Community Experience Distilled

Ionic 2 Blueprints

Build real-time, scalable, and interactive mobile apps with the Ionic framework

Indermohan Singh

[PACKT] PUBLISHING open source* community experience distilled

Ionic 2 Blueprints

Build real-time, scalable, and interactive mobile apps with the Ionic framework

Indermohan Singh

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Ionic 2 Blueprints

First published: August 2016

Production reference: 1220816

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78646-373-9

www.packtpub.com

Credits

Authors	**Copy Editors**
Indermohan Singh	Safis Editing
Reviewer	**Project Coordinator**
Richard Shergold	Ulhas Kambali
Commissioning Editor	**Proofreader**
Amarabha Banerjee	Safis Editing
Acquisition Editor	**Indexer**
Reshma Raman	Tejal Daruwale Soni
Content Development Editor	**Graphics**
Onkar Wani	Disha Haria
Technical Editor	**Production Coordinator**
Sushant S Nadkar	Arvindkumar Gupta

About the Author

Indermohan Singh is a freelance JavaScript developer with an interest in AngularJS, React, Ionic, TypeScript, and ES6. For the last two years, he has worked mostly on Ionic, creating mobile apps for various start-ups and organizations.

Almost two years ago, he started a blog (`https://inders.in`) where he writes about Ionic, AngularJS, and JavaScript. As an open source enthusiast, he frequently create demos and publishes them through Github (`https://github.com/imsingh`).

He also gave a talk on Ionic 2 at the *Mobile App Europe Conference*, Berlin, last year. He frequently teaches about mobile development, AngularJS, and Ionic at Codementor, where he is a featured expert on Ionic. He was also invited by his alma mater to teach programming as a guest lecturer.

Apart from coding, Indermohan is very interested in music. When he is not coding, he is composing a new song. He sometimes posts his songs on SoundCloud (`http://soundcloud.com/indermohan-singh-2/stats`). He is also the creator of an *Indian Classical Music Search Engine* for Ragas named Ragakosh. For his idea of a search engine, he was select by MIT Media Lab for the Entrepreneurship Bootcamp 2015.

He was also a technical reviewer of Learning Ionic, Arvind Ravulavaru, Packt Publishing.

Acknowledgements

First I would like to thank God, for giving me strength all the time, especially when I was feeling down. I can't see you but I can feel you somewhere inside me.

A lot of people are involved in making me the person, I am today. I would like to thank Davinder pal Singh and Bhupinder Kaur, and my parents for supporting me in all my good decisions and pouring wisdom when I made bad decisions. Thanks to my brother Gursharan Singh, who did all my other tasks when I was busy working on the book. I would like to thank all of my teachers, especially Hardeep Singh Jawanda, who gave me so many unique projects to work on during my study.

I would like to thank Reshma Raman and Nikhil Nair from Packt, who gave me this opportunity, Anish Durat and Onkar Wani (Content Development Editors) for their valuable feedback during writing process, Sushant Nadkar (Technical Editor) for giving the book, final touch and all other folks at Packt who worked hard to make this book a reality.

Special thanks to Richard Shergold, Technical Reviewer, who gave his insights and feedback to make this book a lot better.

I would also like to thank the team at Ionic for creating such a wonderful framework, especially Adam Bradley, who was always there to answer my questions regarding Ionic.

And last, I want to thank all the open source community for creating wonderful piece of software, blog posts, and demos, which helped me learn a lot about software development.

About the Reviewer

Richard Shergold is a mobile applications developer based in the UK and currently working with the Ionic 2/Angular 2 frameworks. Most recently, he has been developing mobile applications for the Salesforce platform using Ionic in combination with the MobileCaddy framework (`www.mobilecaddy.net`).

Richard also has mobile development experience with native iOS development (Objective C), native Android development (Java), native app development with Appcelerator Titanium, and hybrid app development with Sencha Touch. Richard also has a blog about Ionic 2 development at `www.ionicallyspeaking.com`.

Table of Contents

Preface

Ionic 2 Blueprint will take you to a splendid journey of creating seven mobile apps using the Ionic 2 Framework, starting from simple apps using REST APIs and gradually increasing the complexity of the apps, which involves lots advanced Ionic features and native device features.

The whole idea of this book is to help developers jump start creating elegant mobile apps using Ionic 2 beta 10. The book deals with real-world apps, using REST APIs, Backend as a Service (BaaS), and assumes that you have a fair knowledge of Angular 2 and bit of Ionic 2. You will learn a lot about managing bigger projects, how to deal with complexity, and creating reusable components.

This book will be an invaluable resource for all those developers who seeks to level up their Ionic 2 skills.

What this book covers

`Chapter 1`, *Chat App with Firebase*, will show you to create a real-time chat application using Firebase. You will learn how to use tabs in Ionic 2, how to set up Firebase and how to use Firebase with AngularFire2 to create a chat application.

`Chapter 2`, *E-Commerce App with Marketcloud*, will introduce you to a fairly new BaaS platform for e-commerce, Marketcloud. You will create a complete e-commerce app using the Marketcloud SDK and dashboard, process payments using payment gateway integrations, use the Ionic 2 menu and navigation, learn how to use a loader using Ionic Loading.

`Chapter 3`, *Conference App*, will show you how to create a conference app. We will use Lanyrd's compatible JSON data to build an offline conference app, which will leverage native device capabilities using Ionic native, Leverages LocalStorage, using both Ionic menu and tabs together. We will also use RxJS to build a search filter.

`Chapter 4`, *StockMarket App*, will show you how to create a stock market app, which shows information about the stock of your choice. You will create a reusable UI component using the Angular 2 component API, utilize RxJS to get stock data continuously from Yahoo API, create a custom Angular Pipe, and create responsive charts to display stock history.

`Chapter 5`, *WordPress Client App*, will show you how to create a mobile client for your WordPress blog or site. You will be introduced to the WordPress REST API, Ionic toast, infinite scrolling in Ionic 2, and push notification, with Ionic native, using Google Analytics. The end product will be a WordPress client with blog posts, comments, categories, and WordPress pages.

`Chapter 6`, *Media Player App*, will show you how to create a media player. It is one of the special apps in this book. You will create a complete player with a seek bar for changing the playing position, a play and pause button, and playlist options with a next and previous button. You will also create a file browser to get media from the device.

`Chapter 7`, *Social App with Firebase*, will show you how to create a social app using Firebase. You will use the Firebase database store information and Firebase storage store binary information such as images. Our social app will be like Twitter, in which users can follow other users and see their posts. The chapter will also deal with creating the database structure for our app.

What you need for this book

To build Ionic 2 mobile apps, you need to have a basic knowledge of HTML, CSS, and Angular 2. You also need to have knowledge of Cordova and the Ionic CLI.

You will need to install Node.js, Cordova, the Ionic CLI, Git, and native SDKs for mobile platforms. Instructions of how to install these tools is given in `Chapter 1`, *Chat App with Firebase*.

Who this book is for

This book is for intermediate-level Ionic developers who have some experience of working with Ionic 2, but don't yet have a complete idea of how powerful Ionic can be to create real-time apps with dynamic functionality.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "When a user logs in, it will go to a tabs interface with the `UsersPage` opened."

A block of code is set as follows:

```
<ul *ngFor="let item of items | async">
  <li class="text">
    {{item}}
  </li>
</ul>
```

Any command-line input or output is written as follows:

```
npm install -g ionic@beta cordova
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select **Auth** in the side bar, then click on the **SIGN-IN METHOD** tab on that page."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Updating to latest Ionic 2 versions

While writing the chapters of this book, the latest release of Ionic 2 was Ionic 2 beta 10. So, every chapter and code is written against Ionic 2 beta 10. However, at the time of writing this, Ionic has a new beta 11 release. So, we have decided to publish an update to the book when the final version of Ionic 2 is releases.

However, I want to make this process easier for you. The Ionic team releases changes for each release on GitHub. You can get all the information about changes in latest releases at `https://github.com/driftyco/ionic/blob/master/CHANGELOG.md`. You can follow this guide to update chapter code accordingly, until we release the updated book.

To ease this update process, I am writing this simple guide (as an example) to update from beta 10 to beta 11. Follow these simple steps:

- Beta 10 uses Angular 2 RC4 with latest form modules. In order to use the latest Angular 2 form module, refer to this awesome blog post at `http://blog.thoughtram.io/angular/216/6/22/model-driven-forms-in-angular-2.html`.

- Change in using various Overlay components such as Ionic Alerts, Loading, and others

  In the previous versions till beta 10, this is how we used the Overlay components:

```
import {Component} from '@angular/core';
import {Alert, NavController} from 'ionic-angular';
@Component({
    templateUrl: 'build/pages/home.html'
})
export class UtilProvider {
    constructor(public nav:NavController) {}
    doAlert(title, message, buttonText) {
      let alert = Alert.create({
          title: title,
          subTitle: message,
          buttons: [buttonText]
      });
      this.nav.present(alert);
    }
}
```

  And here is how we use them in beta 11:

```
import {Component} from '@angular/core';
import {AlertContoller} from 'ionic-angular';
@Component({
    templateUrl: 'build/pages/home.html'
})
export class UtilProvider {
    constructor(public nav:NavController, public
      alertController:AlertController){}
    doAlert(title, message, buttonText) {
        let alert = this.alertController.create({
            title: title,
            subTitle: message,
            buttons: [buttonText]
        });
        alert.present();
    }
}
```

Note the following changes in both code samples:

- The Overlay component package names now have Controller appended at the end. For example, `Alert` is now `AlertController`. Similarly, `Loading` is now `LoadingController`.
- You have to create a class instance of the Overlay component controller in order to use them. Refer to the constructor of the latter example code, where we have created the `alertController` instance. In the previous version, this was not required.
- Each instance of the Overlay component has a present method, which is used, instead of the `NavController` class' `present` method.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on Packt' GitHub repository at `https://github.com/PacktPublishing/Ionic-2-Blueprints.git` and also at author's GitHub repository at `https://github.com/ionic2blueprints`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Chat App with Firebase

Ionic 2 is the latest version of Ionic, the super-heroic, hybrid mobile framework. Without any doubt, Ionic 1 was one of best tools for creating cross-platform mobile applications. Several million apps were built with Ionic 1. But it had some shortcomings. Ionic 2 is a completely rethought and rewritten version of Ionic based on **Angular 2** and **TypeScript**, with improved performance, and reduced complexity for writing code.

In this chapter, we will create a real-time chat application using Ionic 2 and Firebase.

We will learn the following things:

- How Ionic navigation works
- How to use tabs in Ionic
- Firebase as a backend using **AngularFire2** in Ionic
- Data modeling in Firebase
- Structuring an Ionic project

This book assumes that you have basic knowledge of Ionic 2 and Angular 2. If you are not familiar with Angular 2, go to Angular's official site at `https://angular.io/docs/ts/latest/quickstart.html`, and if you want to learn the basics of Ionic 2, go to `http://ionicframework.com/docs/v2/`.

The chat application that we will build in this chapter is a real-time chat application. By real-time, I mean that messages will be sent and received instantly. This chapter and app aim to make you familiar with Ionic 2 and AngularFire2, and also aim to show you how to use Cordova plugins inside your Ionic app.

We will first discuss Firebase and, more importantly, we will discuss the AngularFire2 library. Since we are using Firebase as our database, we will also go through the process of defining data structure, which is a very important process for Firebase applications.

Our chat application will show a login page with e-mail and password fields. The user will be able to log in or create a new account. When a user logs in, it will go to a tabs interface with the `UsersPage` opened. This users page will have a list of all the users of the application. The second tab, `ChatsPage`, will have a list of all previous chats. When a user clicks on any other user, or any previous chat, it will open a `ChatViewPage`, which allows users to send text and picture messages. The last tab, `AccountsPage`, will allow the user to upload a profile picture and log out of the app.

Firebase

Firebase is a **Backend as a Service** (**BaaS**) by Google. Firebase provides a real-time backend for various kinds of applications. It provides a data store, authentication, static hosting, and lots of other features. It comes with official SDKs for various platforms including web, Android, iOS, REST API, and server-side libraries.

You can learn more about Firebase at `https://firebase.google.com/features/`.

Firebase also comes with an Angular-specific library for Firebase called **AngularFire**. We will be using AngularFire2 for our Ionic 2 application, which is written in Angular 2 and can work with any Angular 2-based application.

AngularFire2

According to official documentation, AngularFire2 integrates Firebase's real-time observers and authentication with Angular 2. This basically means that it provides various Angular 2 providers that we can use in our Angular 2 application to make a real-time application.

The most important step in using AngularFire2 with an Angular 2 application is injecting and configuring Firebase. The following code demonstrates this:

```
import {bootstrap} from '@angular/core';
import {
  FIREBASE_PROVIDERS,
  defaultFirebase,
  firebaseAuthConfig,
  AuthProviders,
  AuthMethods
} from 'angularfire2';
bootstrap(MyApp, [
  FIREBASE_PROVIDERS,
  defaultFirebase({
    apiKey: "AIzaSyC2gX3jlrBugfnBPugX2p0U1XiSqXhrRgQ",
    authDomain: "chat-app-1e137.firebaseapp.com",
    databaseURL: "https://chat-app-1e137.firebaseio.com",
    storageBucket: "chat-app-1e137.appspot.com",
  }),
  firebaseAuthConfig({
    provider: AuthProviders.Facebook,
    method: AuthMethods.Redirect
  });
])
```

Take a look; first, we have imported stuff from `angularfire2` modules, and in the `bootstrap` process of the application, we have used the `defaultFirebase` method to provide the Firebase `apikey`, `authDomain`, the root endpoint of our Firebase database, and a storage bucket. You will get all these from the Firebase dashboard. We also have to configure authentication in the bootstrap method. For configuring Firebase authentication in AngularFire2, we have the `firebaseAuthConfig` method. In the preceding example, we are using Facebook as the authentication provider.

List and object

AngularFire2 comes with list and object constructs. A list is similar to `$firebaseArray` and an object is similar to `$firebaseObject` from the original AngularFire. Both list and object use RxJS observables under the hood. Take a look at the example codefor list, as follows:

```
import {Component} from 'angular2/core';
import {AngularFire, FirebaseObjectObservable} from 'angularfire2';

@Component({
  selector: 'app',
  template: `
  <h1>{{ (item | async)?.name }}</h1>
  `,
})
export class AppComponent {
  item: FirebaseObjectObservable<any>;
  constructor(af: AngularFire) {
    this.item = af.database.lists('/items');
  }
}
```

Template

Let's take a look at the following example code:

```
<ul *ngFor="let item of items | async">
  <li class="text">
    {{item}}
  </li>
</ul>
```

Notice that we have used an `async` pipe with `ngFor`, which unwraps the item's `Observable` when an item arrives.

To read more about AngularFire 2, check out the official documentation at `https://angularfire2.com/api/`.

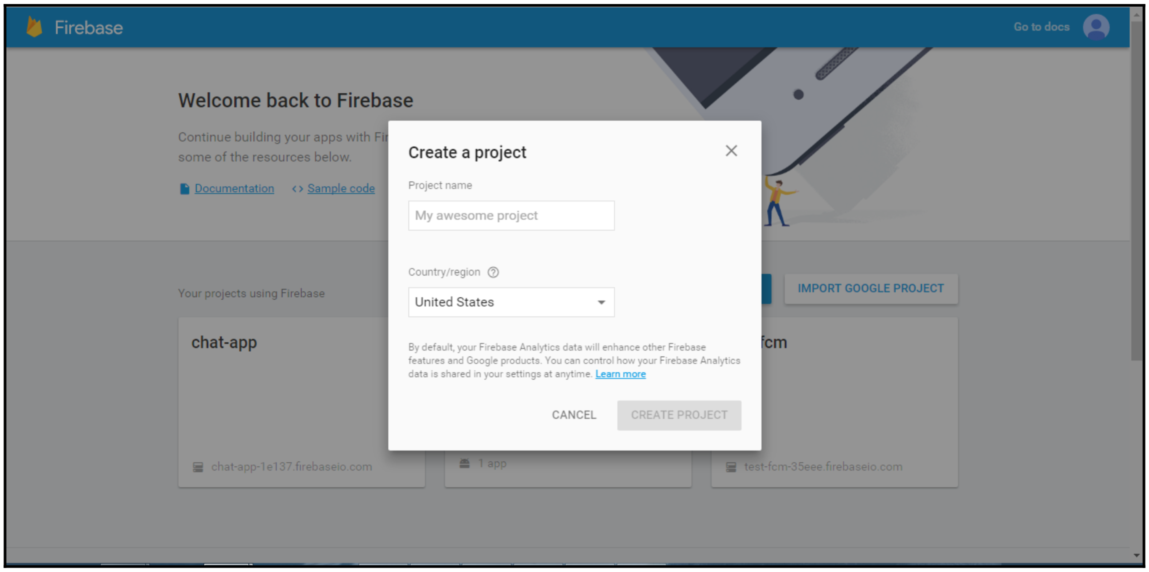
Firebase setup

First we need to create a Firebase app, so let's do it.

Creating our Firebase app

Open `https://firebase.google.com`, click on the **SIGN IN** button and proceed with your Google account. Then, click on **Go to Console** in the top right-hand corner of the page.

This will open the Firebase console. Click on **CREATE NEW PROJECT** on the page and fill in the name for your project and your region (geographical place), as shown in the following screenshot:


Firebase
Go to docs
Welcome back to Firebase
Continue building your apps with Fi
some of the resources below.
Documentation
Sample code
Your projects using Firebase
IMPORT GOOGLE PROJECT
chat-app
fcm
chat-app-1e137.firebaseio.com
1 app
test-fcm-35eee.firebaseio.com

Create a project

Project name

My awesome project

Country/region

United States

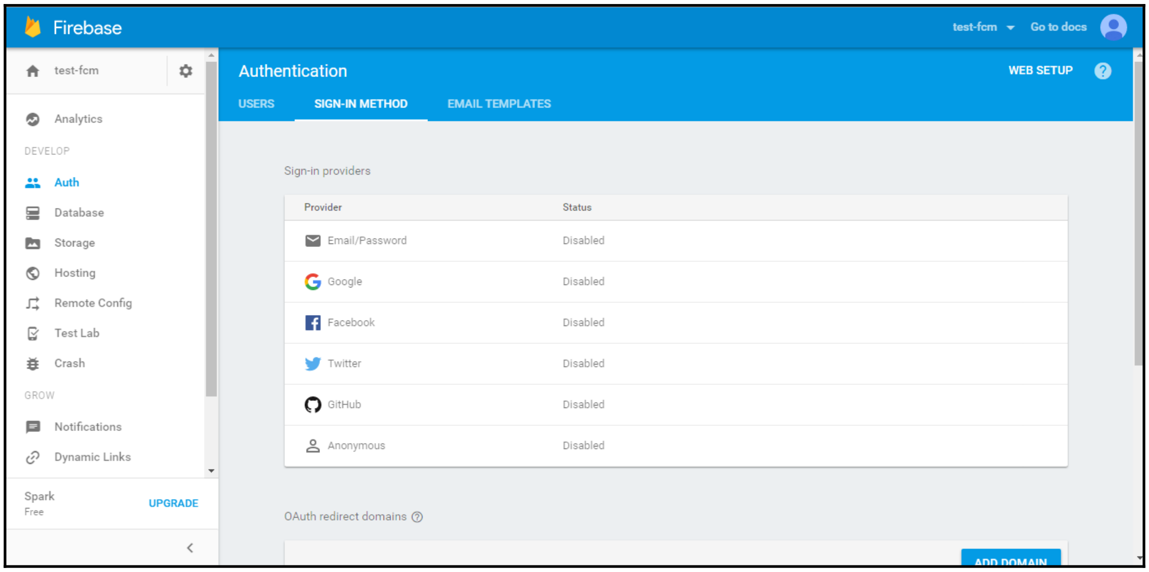
By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more

CANCEL CREATE PROJECT

Enabling password authentication

We will be using Firebase's password authentication system, so let's enable that too.

Open your app's dashboard. Select **Auth** in the side bar, then click on the **SIGN-IN METHOD** tab on that page. Next, enable the **Email/Password** provider, as shown in the following screenshot:



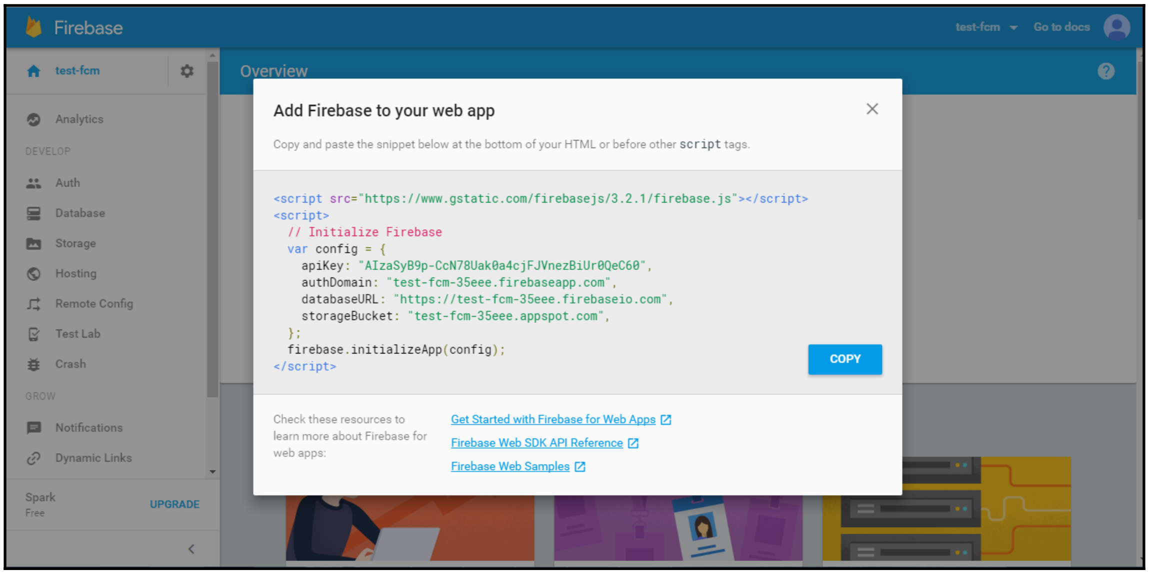
Sign-in providers

Provider	Status
Email/Password	Disabled
Google	Disabled
Facebook	Disabled
Twitter	Disabled
GitHub	Disabled
Anonymous	Disabled

Getting Firebase configuration data

We need Firebase configuration data in order to use it in our Ionic application.

On your app's **Overview** screen, you will see buttons to configure your Firebase app. Click on the **Add Firebase to your web app** button and you will get your Firebase **apiKey**, **authDomain**, **databaseURL**, and **storageBucket** link, as shown in the following screenshot:



Add Firebase to your web app

Copy and paste the snippet below at the bottom of your HTML or before other `script` tags.

```
<script src="https://www.gstatic.com/firebasejs/3.2.1/firebase.js"></script>
<script>
  // Initialize Firebase
  var config = {
    apiKey: "AIzaSyB9p-CcN78Uak0a4cjFJVnezBiUr0QeC60",
    authDomain: "test-fcm-35eee.firebaseapp.com",
    databaseURL: "https://test-fcm-35eee.firebaseio.com",
    storageBucket: "test-fcm-35eee.appspot.com",
  };
  firebase.initializeApp(config);
</script>
```

Check these resources to learn more about Firebase for web apps:

Get Started with Firebase for Web Apps
Firebase Web SDK API Reference
Firebase Web Samples

Setting up the software

Before we actually start working on our app, we need the following software tools and libraries to get started.

Installing node and npm

You need to have Node.js and npm installed on your computer. Go to `https://nodejs.org` and follow the instructions based on your operating system to install Node.js and npm.

You can verify the installation of Node.js and npm by running the following commands in the Terminal (Unix systems) or command prompt (Windows systems):

```
node -v
npm -v
```

Installing Git

Git is a free and open source version control system created by Linus Torvalds. Ionic uses Git to download project templates.

To install Git, navigate to `http://git-scm.com/downloads` and follow the instructions to download Git for your specific operating system. You can also verify the installation of Git by running the following command:

```
git --version
```

Installing platform SDKs

Throughout this book, we will be creating apps that will run on an actual device, so we need to set up platform SDKs in order to build a platform-specific installer.

iOS users can follow the guide at `https://cordova.apache.org/docs/en/latest/guide/platforms/ios/index.html` to set up iOS SDK on their local machine.
Android users can follow the guide at `https://cordova.apache.org/docs/en/latest/guide/platforms/android/index.html`.Windows users can follow the guide at `https://cordova.apache.org/docs/en/latest/guide/platforms/win8/index.html`. It is important to note that you need to have an OS X machine to develop an iOS application, and you need a Windows 8 or 10 machine to develop Windows phone applications.

Alternatively, you can also use Ionic's package service to build a platform-specific installer without installing platform SDKs.

Read the official documentation of the Ionic package service at `http://docs.ionic.io/docs/package-overview`.

Installing the Ionic CLI and Cordova

You also need to install the Ionic CLI and Cordova for developing Ionic 2 applications, using the following command:

```
npm install -g ionic@beta cordova
```

Note that we are using a `@beta` tag with Ionic to install the Ionic CLI. Only the Beta version of Ionic CLI supports Ionic 2 at the time of writing this book.

Defining our app

In this section, we will define our application structure and architecture. I think this is probably the best way to start creating an application. We will discuss various things, starting with the functionalities we are providing in our app, how to structure our database, and how the app will be structured visually.

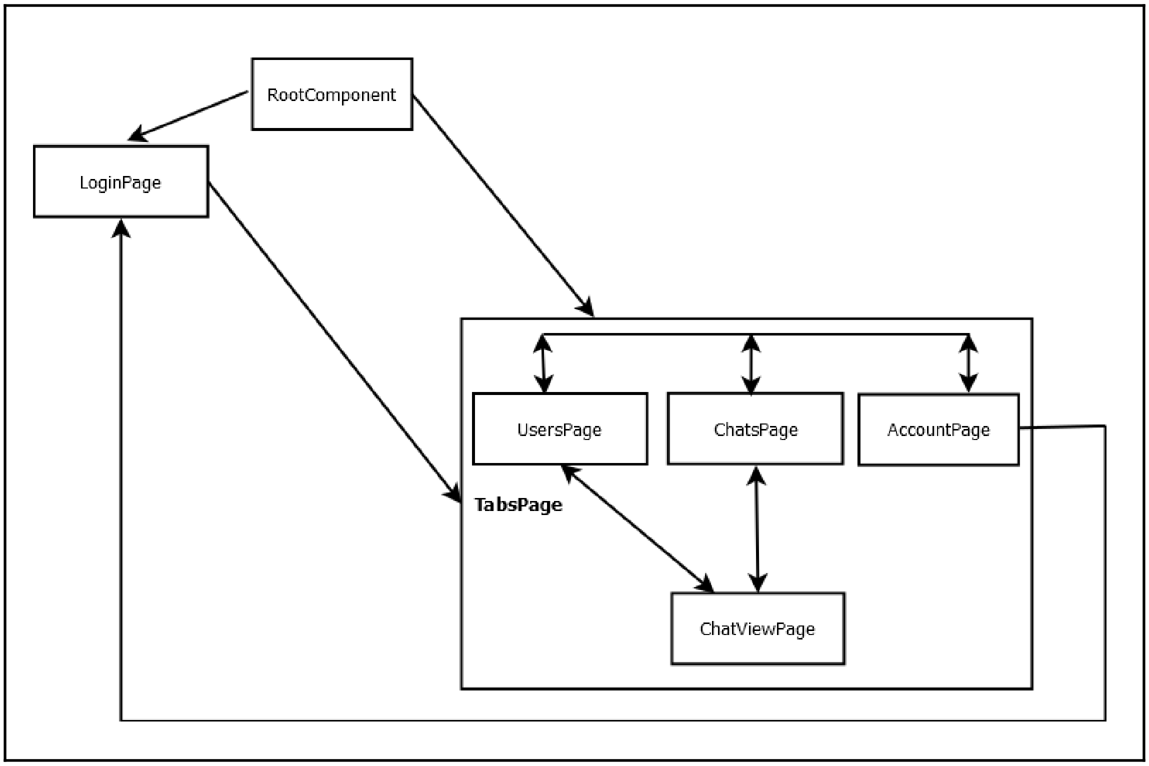
Functionalities

We will be including the following functionalities in our application:

- E-mail password authentication
- A list of all users of the application
- A list of previous chats
- Real-time text and picture messages
- Profile picture

App flow

If we define how the app works and goes from one page to another page, it will become quite easy to understand. So, let's define our applications flow:



Let's understand the flow:

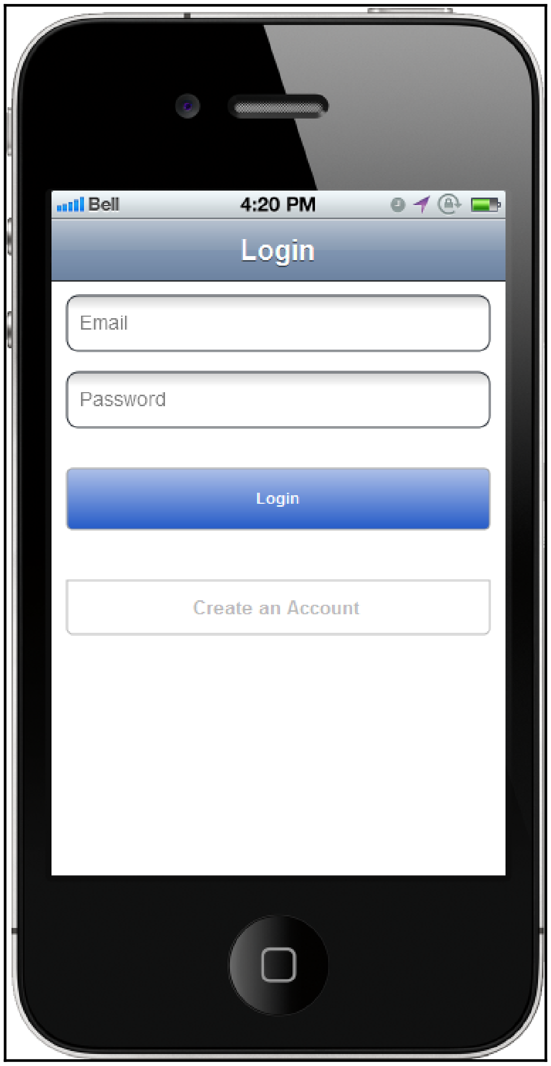
- **RootComponent**: This is the root Ionic component. It is defined inside the `/app/app.ts` file. If a user is already logged in, the `TabsPage` will appear. Otherwise, it will show the `LoginPage`.
- **LoginPage**: This provides a user with the ability to log in or create a new account. In either case, the user then goes to the `TabsPage`.
- **TabsPage**: This is an *abstract* page. By abstract, I mean it will never open alone and always has a child page opened. This `TabsPage` has three tabs, called `UsersPage`, `ChatsPage`, and `AccountPage`; each tab has its own navigation stack.
- **UsersPage**: This lists all the users of our application. By clicking on any user, it will open a `ChatViewPage` for chatting with that specific user.
- **ChatsPage**: This lists all the previous chats. By clicking on any chat, it will also open a `ChatViewPage` for chatting.
- **ChatViewPage**: This allows a user to chat with other users. It lists all the chat messages between two users, an input field to send a text message, and a button to send a picture message. We can go back to either the `UsersPage` or `ChatsPage`, depending on where we came from.
- **AccountPage**: This allows a user to upload a profile picture and to log out of the application. When a user logs out, it will then go to the `LoginPage`.

It is important to understand that a user can go from any tab to any other tab. So, `UsersPage` | `ChatsPage` or `ChatsPage` | `AccountPage` are also part of the app flow. This is how tabs work!

User Interface (UI) mock-ups

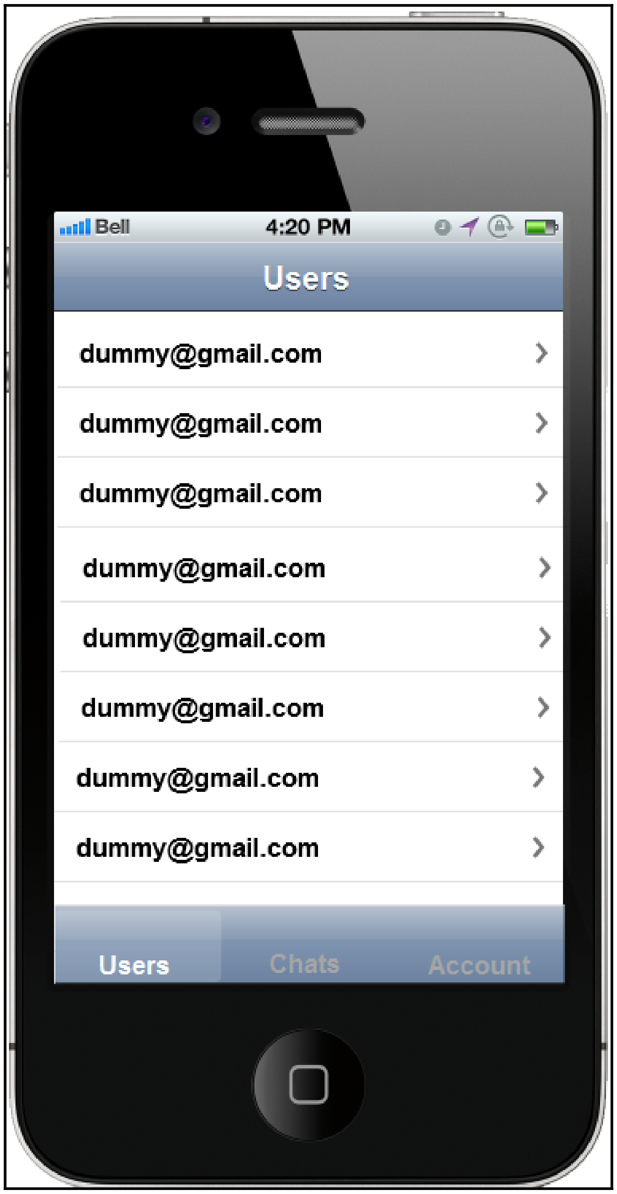
Over the last couple of years, I have found it really useful to have UI mock-ups before starting to code an application. It makes it really easy to think about how you are going to write your templates. The following are the UI mock-ups for every page of our application.

The following screenshot of is of our `LoginPage` UI:



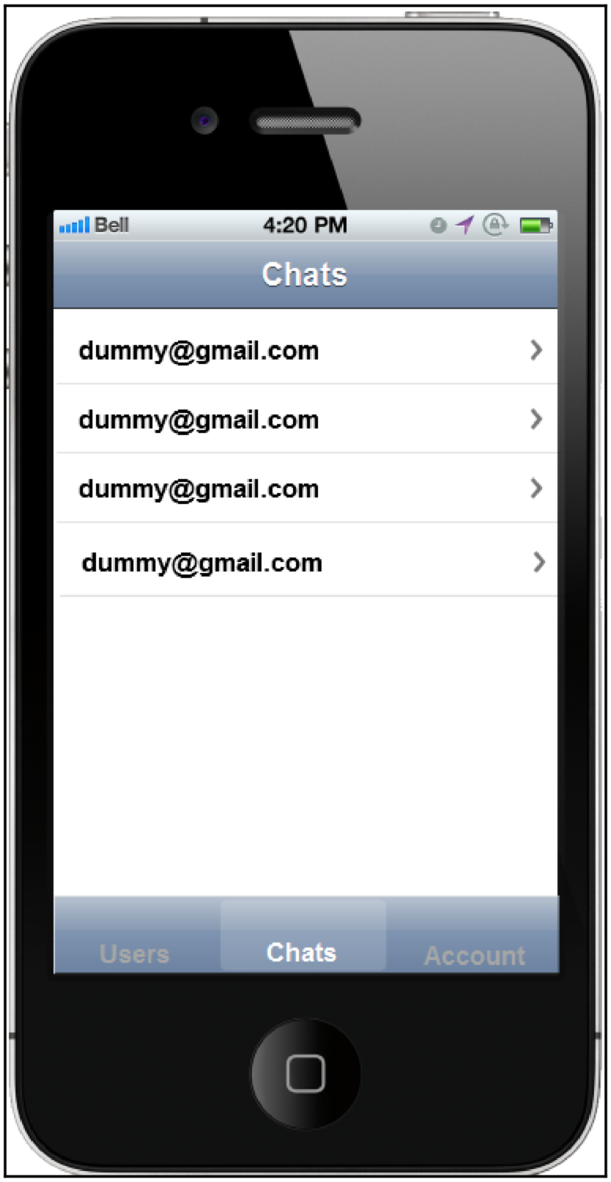
We have **Email** and **Password** fields, and two buttons. One button is for logging in and the other button is for creating an account. If the **Login** button is clicked, it will log in with the information provided and if the **Create an Account** button is clicked, it will create a new account with the same information.

The following screenshot is of the `UserPage`, showing a list of all the users of our application:



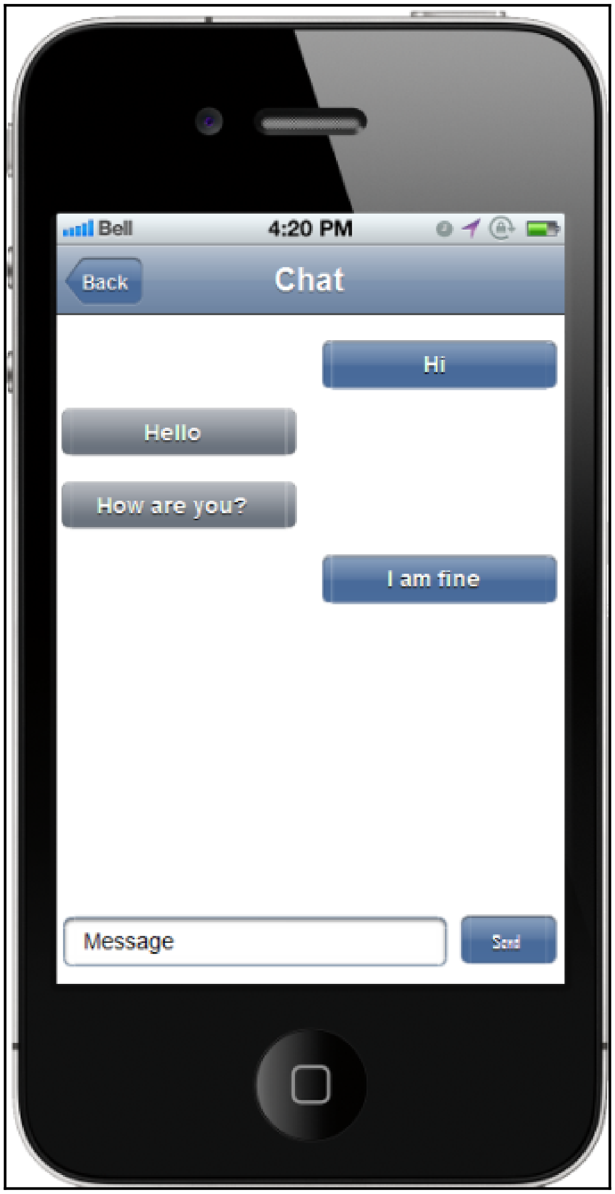
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com

Clicking on any user will open the `ChatViewPage`. The following screenshot shows the `ChatsPage`, showing a list of all our previous chats:

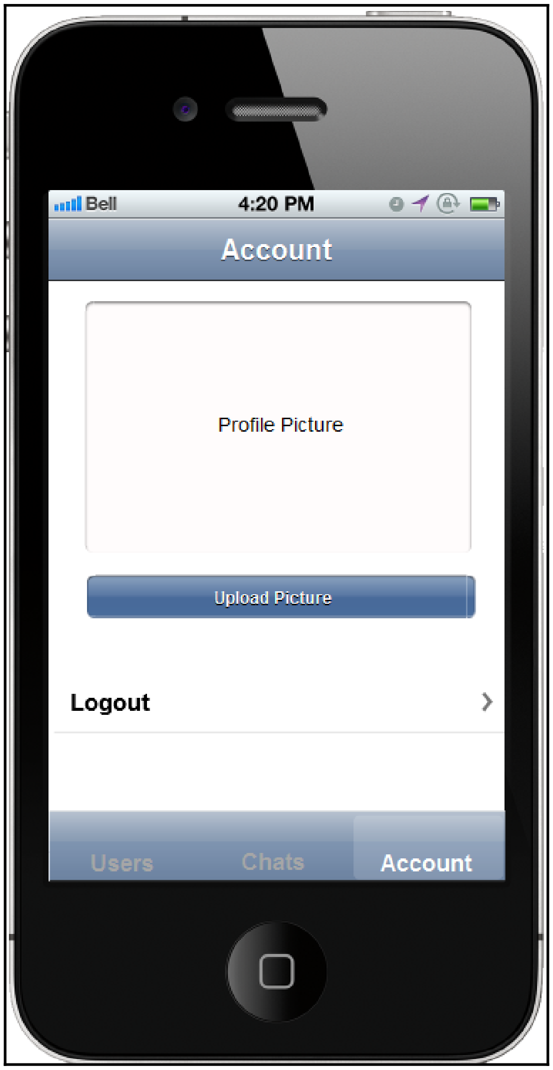


dummy@gmail.com
dummy@gmail.com
dummy@gmail.com
dummy@gmail.com

Clicking on any chat will open the `ChatViewPage`, as shown in the following screenshot:



The `ChatViewPage` will have chats from both users, and an input box and send button at the bottom. The following is how we will design our `AccountPage` UI, with a profile picture and button to update it, and a logout button:



These mock-ups are created with the open source tool **Pencil Project** (`http://pencil.evolus.vn/`).

Firebase's data structure

Deciding on the data structure for the Firebase app before coding it is a crucial step. Since we are using Firebase as our backend, we have to structure our database in such a way that we get all the functionalities without too much redundancy of data. Since Firebase stores data as JSON, we will define our data structure using JSON.

First, we need to store users' information. Let's define it, as shown:

```
users: {
}
```

This will store the information of all users of the application. To save the information of a particular user, we need a unique key. Firebase password authentication gives us a unique ID (`uid`), which we will use as our unique key for each user, as follows:

```
uid: {
}
```

Each user will have information such as an e-mail and a picture, so we need to include these in our data structure, as follows:

```
uid: {
  email: "user@email.com",
  picture: "user's base64 encode picture"
}
```

We also need to track each user's chat history, so we create a chats key for each user, which stores the `uid` of the other user. Our complete user data structure will be as follows:

```
users: {
        uid: {
            email: "dummy@email.com"
            picture: "base64Encoded Picture",
            chats: {
                uid1: true,
                uid2: true
            }
        }
}
```

Here, `uid` is the unique ID of the user and `uid1` and `uid2` are the unique IDs of the other users to whom our user has talked previously.

So far we have stored information about the users. Now we also need to store the chat messages. Let's define the chats data structure step by step:

1. The following is the basic structure for chats to store messages:

```
chats: {
}
```

2. Now we need to define the data structure of chats in such a way that it creates a relationship between two users who are having a conversation. To do this, we will create a key inside our chats data structure, with the `uid` of both users separated by a comma, as follows:

```
uid1,uid2: {
}
```

3. This means that this key, or endpoint, will have chat messages between user `uid1` and user `uid2`.
4. Each chat message is just added into this object using the AngularFire2 `add` method, so each chat message will have its own Firebase-generated, unique key. Each chat message will have from, message or picture, and type keys, as shown:

```
unique-key: {
  from: "UID of User",
  message: "Chat Message",
  picture: "Chat Picture Message",
  type: "message type"
}
```

   In this, `from` stores the `uid` of the user who sent the message, `message` stores the text message, `picture` stores the picture messages, and `type` stores the type of message.

There will be either a message or a picture key, depending on the type of message.

5. Our final chats data structure will look something like the following:

```
chats: {
    uid1,uid2: {
        unique-id: {
            from: uid,
            message: "Chat Message",
            type: "message"
        },
      unique-id: {
            from: uid,
            picture: "Picture Message",
            type: "picture"
        },
    }
}
```

Scaffolding and setting up our app

Ionic CLI eases the process of scaffolding and setting up an app for us, so we will be using the Ionic CLI that we installed earlier. We will start by scaffolding a blank application. Run the following command:

```
ionic start firebase-chat blank --v2 --ts
```

Notice the `--v2` and `--ts` tag for scaffolding our Ionic 2 app based on TypeScript.

Using the `cd` command, go to the `firebase-chat` folder and run the following command to check how the blank app looks:

```
ionic serve
```

Before we start writing the code for our app, we need to install some project-specific dependencies.

Installing Cordova plugins

We will be using a camera plugin to get pictures from a mobile device and upload them to Firebase. The following command will install the plugin:

```
ionic plugin add cordova-plugin-camera
```

Installing Firebase and AngularFire2

Since we are using AngularFire2 for interacting with Firebase, we need to install both of these dependencies. This can be done using the following command:

```
npm install angularfire2 && firebase -save
```

Installing Typings

Typings is the TypeScript definition manager, which is required to install TypeScript definitions. This can be done using the following command:

```
npm install typings -g
```

Installing TypeScript definitions for Firebase

For Typings versions lower than 1, use `--ambient` instead of `--global` in the following command:

```
typings install file:node_modules/angularfire2/firebase3.d.ts --save --
global && typings install
```

Coding our app

Now we have everything set up to start working on our application, there are three important steps for coding our app. They are as follows:

- Defining our main `app.ts` file
- Creating providers/services for various functionalities
- Creating Ionic pages for various views

Defining app.ts

This is the root of our application and it defines the `root` component using the `@App` decorator. This is where we inject all of our dependencies. The following should be present in `app.ts`:

```
/* /app/app.ts */
import {NavController, Platform, ionicBootstrap} from 'ionic-angular';
import {StatusBar} from 'ionic-native';
import {Component, Inject} from '@angular/core';
import {LoginPage} from './pages/login/login';
import {TabsPage} from './pages/tabs/tabs';
import {AuthProvider} from './providers/auth-provider/auth-provider';
import {ChatsProvider} from './providers/chats-provider/chats-provider';
import {UserProvider} from './providers/user-provider/user-provider';
import {UtilProvider} from './providers/utils';
import {
    FIREBASE_PROVIDERS,
    defaultFirebase,
    firebaseAuthConfig,
    FirebaseRef,
    AngularFire,
    AuthProviders,
    AuthMethods
} from 'angularfire2';

@Component({
  template: '<ion-nav id="nav" [root]="rootPage" #content></ion-nav>'
})
class MyApp {
  message: string;
  rootPage: any;
  constructor(public authProvider:AuthProvider, public platform:Platform) {
        let auth = authProvider.getAuth();
        auth.onAuthStateChanged(user => {
          if(user) {
            this.rootPage = TabsPage;
          } else {
            this.rootPage = LoginPage;
          }
        });
  }
}

ionicBootstrap(MyApp, [FIREBASE_PROVIDERS,defaultFirebase({
    apiKey: "AIzaSyC2gX3jlrBugfnBPugX2p0U1XiSqXhrRgQ",
    authDomain: "chat-app-1e137.firebaseapp.com",
    databaseURL: "https://chat-app-1e137.firebaseio.com",
    storageBucket: "chat-app-1e137.appspot.com",
  }),
  firebaseAuthConfig({
    provider: AuthProviders.Password,
    method: AuthMethods.Password,
    remember: 'default',
    scope: ['email']
  }),
  AuthProvider,
  ChatsProvider,
  UserProvider,
  UtilProvider] )
```

This is the entry point of our application. In it, we are initializing Firebase and configuring it to use password authentication. We are also checking that, if a user is authenticated, it goes to the `TabsPage`; otherwise, it will go to the `LoginPage`. We are also providing all of our dependencies. We haven't yet created our dependencies, so TypeScript will show errors. Just ignore it.

ionicBootstrap is an alternative to Angular's bootstrap method for Ionic 2 applications.

Note that we have written a template inside this file. We have used `ion-nav` to create a navigation stack, where we will push our views.

To read more about what is going on under the hood inside IonicBootstrap, check this blog post at `http://inders.in/blog/215/1/28/introduction-to-ionic-2/`.

Providers for our application

We will have the following providers for our application:

- AuthProvider
- UserProvider
- ChatsProvider
- UtilProvider

Defining AuthProvider

`AuthProvider` is the provider that we will use for authentication purposes in our app. With the Ionic CLI, we can now generate pages and providers with the command line.

Generating the provider

The following command will create `auth-provider.js` files in the `app/provider/auth-provider` directory, with some default content:

```
ionic g provider AuthProvider
```

Changing the extension

Now, we have to change the extension of the file to `.ts` for using TypeScript in our code.

AuthProvider code

This is the module where we handle all of our authentication work. The following code should be present in `auth-provider.ts`:

```
/* /app/provider/auth-provider/auth-provider.ts */
import {Injectable, Inject} from '@angular/core';
import {FirebaseAuth, FirebaseRef, AngularFire} from 'angularfire2';
import {LocalStorage, Storage} from 'ionic-angular';

@Injectable()
export class AuthProvider {
  local = new Storage(LocalStorage);
  constructor(public af:AngularFire) {}
  getAuth() {
    return firebase.auth();
  };
  signin(credentails) {
    return this.af.auth.login(credentails);
  }
  createAccount(credentails) {
    return this.af.auth.createUser(credentails);
  };
  logout() {
     var auth = firebase.auth();
     auth.signOut();
  }
}
```

Let's understand the preceding code:

- `getAuth()` returns the Firebase SDK's `firebase.auth()` method.
- `signin()` does the login process for us. Since we will be using Firebase password authentication. We are using AngularFire2's `auth.login` method by passing it login credentials.
- `createAccount()` takes an object with e-mail and password values and creates a Firebase account for the user. Again, we are using AngularFire2's `auth.createUser` with credentials to create a Firebase user account.
- `logout()` logs the user out. We are using Firebase SDK's `auth.signOut()` function here.

Defining UserProvider

First we will generate our provider using the following command:

```
ionic g generate UserProvider
```

Then, change the extension of `user-provider.js` to `user-provider.ts`.

UserProvider code

`UserProvider` is for doing user-related work in our application, such as creating a user, getting a list of users, updating the profile of a user, and other things. The following code should be present in `user-provider.ts`:

```
/* /app/providers/user-provider/user-provider.ts */
import {Injectable, Inject} from '@angular/core';
import {FirebaseRef, AngularFire} from 'angularfire2';
import {LocalStorage, Storage} from 'ionic-angular';
import {Camera} from 'ionic-native';

@Injectable()
export class UserProvider {
  local = new Storage(LocalStorage);
  constructor(public af:AngularFire) { }
  // Get Current User's UID
  getUid() {
    return this.local.get('userInfo')
    .then(value => {
      let newValue = JSON.parse(value);
      return newValue.uid;
    });
  }
  // Create User in Firebase
  createUser(userCredentails) {
    this.getUid().then(uid => {
      let currentUserRef = this.af.database.object(`/users/${uid}`);
      currentUserRef.set({email: userCredentails.email});
    });
  }
  // Get Info of Single User
  getUser() {
    // Getting UID of Logged In User
    return this.getUid().then(uid => {
      return this.af.database.object(`/users/${uid}`);
    });
  }
```

```
    // Get All Users of App
    getAllUsers() {
        return this.af.database.list('/users');
    }
    // Get base64 Picture of User
    getPicture() {
        let base64Picture;
        let options = {
            destinationType: 0,
            sourceType: 0,
            encodingType:0
        };
        let promise = new Promise((resolve, reject) => {
             Camera.getPicture(options).then((imageData) => {
                  base64Picture = "data:image/jpeg;base64," + imageData;
                  resolve(base64Picture);
              }, (error) => {
                  reject(error);
            });
        });
        return promise;
    }
    // Update Provide Picture of User
    updatePicture() {
      this.getUid().then(uid => {
        let pictureRef =
         this.af.database.object(`/users/${uid}/picture`);
        this.getPicture()
        .then((image) => {
            pictureRef.set(image);
        });
      });
    }
  }
```

Let's understand the preceding code:

- `getUid()` returns a `promise`, which resolves intothe `uid` of the logged-in user. It gets the `uid` from the `userInfo` key of `LocalStorage`.
- `createUser()` creates a new user in the `users` endpoint in the Firebase database.
- `getUser()` returns the `Observable`, which has the information of the logged-in user from the Firebase database.

- `getAllUsers()` returns an AngularFire2 list `Observable`, which lists all the users of our application.
- `getPicture()` gets a picture from the user's mobile device and returns a `promise`. This promise resolves into a *base64 Encoded JPEG Image*.
- `updatePicture()` updates the user's profile picture. It takes the picture using the `getPicture` method and sets the `picture` key of the logged-in user.

In this app, we are storing images as a *base64* string. It is not a very efficient method. Instead, Firebase provides us with a storage bucket to store binary data such as images, videos, and other binary data. We have used the Firebase storage-bucket approach in `Chapter 7`, *Social App with Firebase* of this book.

Defining ChatsProvider

First, we need to generate our provider using the following command:

```
ionic g provider ChatsProvider
```

Then, change the extension of the file `chats-provider.js` to `chats-provider.ts`.

ChatsProvider code

`ChatsProvider` is used to get a list of previous chats and check if a chat already exists between two users. The following code should be present in `chats-provider.ts`:

```
/* /app/providers/chats-provider/chats-provider.ts */
import {Injectable, Inject} from '@angular/core';
import {AngularFire, FirebaseRef} from 'angularfire2';
import {Observable} from 'rxjs/Observable';
import {UserProvider} from '../user-provider/user-provider';

@Injectable()
export class ChatsProvider {
  constructor(public af: AngularFire, public up: UserProvider) {}
  // get list of Chats of a Logged In User
  getChats() {
     return this.up.getUid().then(uid => {
        let chats = this.af.database.list(`/users/${uid}/chats`);
        return chats;
     });
  }
  // Add Chat References to Both users
```

```
      addChats(uid,interlocutor) {
          // First User
          let otherUid = interlocutor;
          let endpoint =
           this.af.database.object(`/users/${uid}/chats/${interlocutor}`);
          endpoint.set(true);
          // Second User
          let endpoint2 =
           this.af.database.object(`/users/${interlocutor}/chats/${uid}`);
          endpoint2.set(true);
      }

      getChatRef(uid, interlocutor) {
          let firstRef =
           this.af.database.object(`/chats/${uid},${interlocutor}`,
           {preserveSnapshot:true});
          let promise = new Promise((resolve, reject) => {
              firstRef.subscribe(snapshot => {
                    let a = snapshot.exists();
                    if(a) {
                        resolve(`/chats/${uid},${interlocutor}`);
                    } else {
                        let secondRef =
                         this.af.database.object(`/chats/${interlocutor},
                          ${uid}`, {preserveSnapshot:true});
                        secondRef.subscribe(snapshot => {
                            let b = snapshot.exists();
                            if(!b) {
                                this.addChats(uid,interlocutor);
                            }
                        });
                        resolve(`/chats/${interlocutor},${uid}`);
                    }
                });
          });
          return promise;
      }
  }
```

Let's understand the preceding code:

- `getChats()` gets a list of chats of the logged-in user-chats that a user has already initiated.
- `addChats()` takes two input values. The first is the `uid` of the logged-in user, and the second is the `uid` of the other user (interlocutor). This function adds chat references (`uid` of the other user) to both users' information in the Firebase database.

- `getChatRef()` takes two arguments. One is the `uid` of the logged-in user and the other is the `uid` of the other user. It returns a `promise`, which resolves to the Firebase database URL of the chats between these two users. If this is the first time these two users are chatting, it creates a URL in the form of `/chats/${interlocutor},${uid}`, where `${interlocutor}` is the `uid` of the other user and `${uid}` is the `uid` of the logged-in user.

Defining UtilProvider

`UtilProvider` is a module for abstracting some functionalities that we will use repeatedly, such as the Ionic alert. The following code should be present in `utils.ts`:

```
/* /app/providers/utils.ts */
import {Injectable, Inject} from '@angular/core';
import {Alert} from 'ionic-angular';
@Injectable()
export class UtilProvider {
    doAlert(title, message, buttonText) {
      console.log(message);
      let alert = Alert.create({
          title: title,
          subTitle: message,
          buttons: [buttonText]
      });
      return alert;
    }
}
```

In `UtilProvider`, we have created a `doAlert` function that takes a `title`, `message`, and `buttonText` as an input, and creates an Ionic alert box for us. This function becomes very useful for displaying alert messages without writing alert code again and again.

Application pages

Now we have to define all the providers and services for our application. First let's define the pages of our application.

We will have the following pages in our application:

- LoginPage
- TabsPage
- UsersPage

- ChatsPage
- AccountPage
- ChatViewPage

Each page has two or three files. One is the `.ts` file, which controls the page. The other is the `.html` file, which is the template of the page and, if present, the last is the `.scss` file, which is the styling file for the page.

Defining the LoginPage

The `LoginPage` includes the login template and a controller to handle that template. This is the page where the user creates an account or logs in. The following code should be present in `login.ts`:

```
/* /app/pages/login/login.ts*/
import {Component} from '@angular/core';
import {NavController, Storage, LocalStorage} from 'ionic-angular';
import {TabsPage} from '../tabs/tabs';
import {FormBuilder, Validators} from '@angular/common';
import {validateEmail} from '../../validators/email';
import {AuthProvider} from '../../providers/auth-provider/auth-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {UtilProvider} from '../../providers/utils';
import {FirebaseAuth} from 'angularfire2';

@Component({
  templateUrl: 'build/pages/login/login.html'
})
export class LoginPage {
  loginForm:any;
    storage = new Storage(LocalStorage);
    constructor(public nav:NavController,
      form:FormBuilder,
      public auth: AuthProvider,
      public userProvider: UserProvider,
      public util: UtilProvider) {
        this.loginForm = form.group({
            email: ["",Validators.compose([Validators.required,
             validateEmail])],
            password:["",Validators.required]
        });
    }
  signin() {
      this.auth.signin(this.loginForm.value)
      .then((data) => {
          this.storage.set('userInfo', JSON.stringify(data));
          this.nav.push(TabsPage);
      }, (error) => {
          let errorMessage = "Enter Correct Email and Password";
          let alert = this.util.doAlert("Error",errorMessage,"Ok");
          this.nav.present(alert);
      });
    };
    createAccount() {
        let credentails = this.loginForm.value;
        this.auth.createAccount(credentails)
        .then((data) => {
           this.storage.set('userInfo', JSON.stringify(data));
           this.userProvider.createUser(credentails);
        }, (error) => {
            let errorMessage = "Account Already Exists";
            let alert = this.util.doAlert("Error",errorMessage,"Ok");
            this.nav.present(alert);
        });
    };
}
```

In the constructor, we have created a login form using Angular2's form builder, and we are using a custom validator that we have defined in the following code to validate the e-mail ID, since Angular2 doesn't have a validator for e-mail ID.

The `signin` function takes the user's e-mail and password, and authenticates the user, using the `signin` member function of `AuthProvider`. If the user is authenticated successfully they navigate to the `TabsPage`. Otherwise, it shows an error message using the `doAlert` member function of `UtilProvider`.

Similarly, the `createAccount` function takes the user's e-mail and password and creates a new user account. It also adds the user's information to the `users` key in the Firebase database. If the user already exists it shows an error message.

Template

The following code should be present in `login.html`:

```
<!--  /app/pages/login/login.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Login</ion-title>
    </ion-navbar>
</ion-header>
```

```
<ion-content class="padding">
   <form [ngFormModel]="loginForm">
    <ion-list>
        <ion-item>
            <ion-label floating>Email</ion-label>
            <ion-input type="text" ngControl="email"></ion-input>
        </ion-item>

        <ion-item>
            <ion-label floating>Password</ion-label>
            <ion-input type="password" ngControl="password"></ion-
             input>
        </ion-item>
    </ion-list>
    <div padding>
      <button primary block (click)="signin()"
       [disabled]="!loginForm.valid">Sign In</button>
    </div>
    <div padding>
      <button full clear favorite (click)="createAccount()"
       [disabled]="!loginForm.valid">
          <ion-icon name="person"></ion-icon>
          Create an Account</button>
    </div>
   </form>
</ion-content>
```

Both the **Sign In** and **Create Account** buttons will be enabled only when the form is valid.

Defining a custom e-mail validator

We need to define a custom validator, which validates the e-mail input from users because Angular 2 doesn't have a default e-mail validator. The following code should be present in `email.ts`:

```
/* /app/validators/email.ts  */
import {Control} from '@angular/common';

export function validateEmail(c: Control) {
 let EMAIL_REGEXP = /^[a-z0-9!#$%&'*+\/=?^_`{|}~.-]+@[a-z0-9]([a-z0-
  9-]*[a-z0-9])?(\.[a-z0-9]([a-z0-9-]*[a-z0-9])?)*$/i;
  return EMAIL_REGEXP.test(c.value) ? null : {
    validateEmail: {
      valid: false
    }
  };
}
```

This just takes the control's value and validates it against the regular expression provided using `EMAIL_REGEXP`.

Defining the TabsPage

The `TabsPage` handles the tabs of our app. This is the place where we define the root page of each tab. The following code should be present in `tabs.ts`:

```
/*  /app/pages/tabs/tabs.ts  */
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {ChatsPage} from '../chats/chats';
import {AccountPage} from '../account/account';
import {UsersPage} from '../users/users';

@Component({
  templateUrl: 'build/pages/tabs/tabs.html'
})
export class TabsPage {
  chats = ChatsPage;
  users = UsersPage;
    profile = AccountPage;
}
```

Template

The following code should be present in `tabs.html`:

```
<!-- /app/pages/tabs/tabs.html -->
<ion-tabs light>
    <ion-tab [root]="users" tabTitle="Users" tabIcon="people"></ion-
     tab>
    <ion-tab [root]="chats" tabTitle="Chats" tabIcon="chatboxes"></ion-
     tab>
    <ion-tab [root]="profile" tabTitle="Account" tabIcon="person"></ion-
tab>
</ion-tabs>
```

In the `TabsPage`, we have just defined the root pages of all our tabs. The `[root]` property is used to set the root page of a tab. By default, the first tab is opened when the `TabsPage is pushed into the navigation stack.`

Defining the UsersPage

The `UserPage` is the page where we will list all of our users. The following code should be present in `users.ts`:

```
/*  /app/pages/users/users.ts  */
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {Observable} from 'rxjs/Observable';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {ChatViewPage} from '../chat-view/chat-view';

@Component({
    templateUrl: 'build/pages/users/users.html'
})
export class UsersPage {
    users:Observable<any[]>;
    uid:string;
    constructor(public nav: NavController, public userProvider:
     UserProvider) {
        userProvider.getUid()
        .then(uid => {
            this.uid = uid;
            this.users = this.userProvider.getAllUsers();
        });
    }
    openChat(key) {
        let param = {uid: this.uid, interlocutor: key};
        this.nav.push(ChatViewPage,param);
    }
}
```

In the constructor, we got the `uid` of the logged-in user and a list of all users of the app as an `Observable`.

The `openChat` function opens the `ChatViewPage` by passing the `uid` of both users as navigation parameters.

Template

The following code should be present in `user.html`:

```
<!-- /app/pages/users/users.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Users</ion-title>
        <ion-buttons end>
            <ion-spinner *ngIf="!(users | async)"></ion-spinner>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list>
      <span *ngFor="let user of users | async">
        <a ion-item (click)="openChat(user.$key)" *ngIf="user.$key !==
         uid">
             <ion-avatar item-left>
                <img *ngIf="!user.picture" src="img/default.jpg">
                <img *ngIf="user.picture" src="{{user.picture}}">
             </ion-avatar>
             <h2>{{user.email}}</h2>
        </a>
      </span>
    </ion-list>
</ion-content>
```

We have used ngFor to iterate over the users list and we have excluded the logged-in user from the list using ngIf. Basically, we check if the uid of the logged-in user and the user in the iteration is the same, then exclude it from the list. We are also showing the avatar of each user. If the user has uploaded his picture, we show it from Firebase; otherwise, we show a default image.

We are also showing a loading spinner in the navbar until we get the list of users.

Defining the ChatsPage

The ChatsPage lists all previous chats. The following code should be present in chats.ts:

```
/* /app/pages/chats/chats.ts */
import {Component} from '@angular/core';
import {NavController, NavParams} from 'ionic-angular';
import {Observable} from 'rxjs/Rx';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {ChatsProvider} from '../../providers/chats-provider/chats-
provider';
import {AngularFire} from 'angularfire2';
import 'rxjs/add/operator/map';
import {ChatViewPage}  from '../chat-view/chat-view';

@Component({
    templateUrl: 'build/pages/chats/chats.html'
})
export class ChatsPage {
    chats:Observable<any[]>;
    constructor(public chatsProvider: ChatsProvider,
        public userProvider: UserProvider,
        public af:AngularFire,
        public nav: NavController) {
            this.chatsProvider.getChats()
            .then(chats => {
                this.chats = chats.map(users => {
                    return users.map(user => {
                        user.info =
                        this.af.database.object(`/users/${user.$key}`);
                        return user;
                    });
                });
            });
        }
    openChat(key) {
        this.userProvider.getUid()
        .then(uid => {
            let param = {uid: uid, interlocutor: key};
            this.nav.push(ChatViewPage,param);
        });
    }
}
```

The `ChatsPage` is similar to the `UsersPage`. The `openChat` function does exactly the same thing as it does in the `UsersPage`. The only difference is that instead of showing all the users of the application, we show only those users who have already had a conversation with the logged-in user. First we get all the references of his previous chats from his Firebase endpoint, which contains all the `uid` of other people. Then we map all those `uid` to keys inside the users endpoint to get the e-mails of those users. It is like a join. This is an asynchronous process, and this is what `Observables` are capable of. You can filter, map, search, and do lots of other stuff on `Observables`.

Template

The following code should be present in `chats.html`:

```
<!-- /app/pages/chats/chats.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Chats</ion-title>
        <ion-buttons end>
```

```
            <ion-spinner primary *ngIf="!(chats | async)"></ion-
             spinner>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
     <a ion-item *ngFor="let chat of chats | async"
      (click)="openChat(chat.$key)">
        <ion-avatar item-left>
                <img *ngIf="!(chat.info | async).picture"
                 src="img/default.jpg">
                <img *ngIf="(chat.info | async).picture" src="
                 {{(chat.info | async).picture}}">
        </ion-avatar>
        <span>{{(chat.info | async).email}}</span>
     </a>
</ion-content>
```

Defining the ChatViewPage

The `ChatViewPage` is the place where the actual chatting takes place. The following code should be present in `chat-view.ts`:

```
/* /app/pages/chat-view/chat-view.ts */
import {Component, ViewChild} from '@angular/core';
import {NavController, NavParams, Content} from 'ionic-angular';
import {Observable} from 'rxjs/Observable';
import {AngularFire, FirebaseListObservable} from 'angularfire2';
import {ChatsProvider} from '../../providers/chats-provider/chats-
provider';
import {UserProvider} from '../../providers/user-provider/user-provider';

@Component({
  templateUrl: 'build/pages/chat-view/chat-view.html',
})
export class ChatViewPage {
  message: string;
  uid:string;
  interlocutor:string;
  chats:FirebaseListObservable<any>;
  @ViewChild(Content) content: Content;
  constructor(public nav:NavController,
  params:NavParams,
  public chatsProvider:ChatsProvider,
  public af:AngularFire,
  public userProvider:UserProvider) {
    this.uid = params.data.uid;
    this.interlocutor = params.data.interlocutor;
    // Get Chat Reference
    chatsProvider.getChatRef(this.uid, this.interlocutor)
    .then((chatRef:any) => {
        this.chats = this.af.database.list(chatRef);
    });
  }

  ionViewDidEnter() {
    this.content.scrollToBottom();
  }

  sendMessage() {
      if(this.message) {
          let chat = {
              from: this.uid,
              message: this.message,
              type: 'message'
          };
          this.chats.push(chat);
          this.message = "";
      }
  };
  sendPicture() {
      let chat = {from: this.uid, type: 'picture', picture:null};
      this.userProvider.getPicture()
      .then((image) => {
          chat.picture =  image;
          this.chats.push(chat);
      });
  }
}
```

In the constructor, we get the `uid` of both users and then we get the Firebase URL of their chat's endpoint, which is something like this: `/chats/uid1,uid2`. With this URL, we get a list of all the messages between these two users' AngularFire2 lists.

In the `sendMessage` function, we send chat messages using the push function of the AngularFire2 list. Similarly, in the `sendPicture` function, we get a picture from the user's gallery and send it as a base64 encoded string.

The `ionViewDidEnter()` function is an Ionic page hook. It fires each time a page is pushed in the navigation stack. We scroll to the bottom in this function using the `scrollToBottom()` method of `ion-content`.

It is important to note that we are using a `@ViewChild` decorator to get hold of `ion-content`.

For further information about Ionic page life cycle hooks, take a look at `http://ionicframework.com/docs/v2/api/components/nav/NavController/`.

Template

The following code should be present in `chat-view.html`:

```
<!-- /app/pages/chat-view/chat-view.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Chat</ion-title>
    <ion-buttons end>
      <button (click)="sendPicture()"><ion-icon name="image" ></ion-
       icon>Send Image</button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content padding class="chat-view" id="chat-view">
  <div class="messages">
      <div class="message" *ngFor="let chat of chats | async"
       [ngClass]="{'me': uid === chat.from}">
            <span *ngIf="chat.message">{{chat.message}}</span>
            <img *ngIf="chat.picture" src="{{chat.picture}}"
             class="picture">
      </div>
  </div>
</ion-content>

<ion-footer>
  <ion-toolbar>
    <ion-row>
      <ion-col width-10>
          <ion-spinner *ngIf="!(chats)"></ion-spinner>
      </ion-col>
      <ion-col width-70 [hidden]="!chats">
          <ion-input type="text"  placeholder="Enter Message"
              [(ngModel)]="message">
          </ion-input>
      </ion-col>
      <ion-col width-20 [hidden]="!chats">
```

```
          <button full (click)="sendMessage()"><ion-icon name="send">
           </ion-icon></button>
        </ion-col>
      </ion-row>
    </ion-toolbar>
  </ion-footer>
```

Take a look at how we have used a different class for the logged-in user's chat message, and how we have used an Ionic grid in `ion-toolbar`.

Style sheet

The following code should be present in `chat-view.scss`:

```
/* /app/pages/chat-view/chat-view.scss */
.chat-view {
  .messages {
    width: 100%;
    position: absolute;
    .message {
      width: 70%;
      padding: 5px 10px;
      background: #3F51B5;
      color: #fff;
      border-radius: 5px;
      margin: 5px;
      float: left;
    }
    .message.me {
      float: right;
      background: #fff;
      border: 1px solid #3F51B5;
      color: #222;
      text-align: right;
    }
  }
}
```

Defining the AccountPage

The `AccountPage` is the place where the user updates their profile picture and logs out. The following code should be present in `accounts.ts`:

```
/* /app/pages/account/account.ts */
import {Component} from '@angular/core';
import {NavController, LocalStorage, Storage} from 'ionic-angular';
import {AuthProvider} from '../../providers/auth-provider/auth-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';

@Component({
    templateUrl: 'build/pages/account/account.html'
})
export class AccountPage {
    rootNav;
    user = {};
    local = new Storage(LocalStorage);
    constructor(public nav: NavController, public auth: AuthProvider,
     public userProvider: UserProvider) {
        this.userProvider.getUser()
        .then(userObservable => {
            userObservable.subscribe(user => {
                this.user = user;
            });
        });
    }
    //save user info
    updatePicture() {
        this.userProvider.updatePicture();
    };

    logout() {
        this.local.remove('userInfo');
        this.auth.logout();
    }
}
```

In the `AccountPage`, we are just updating the user's profile picture using the `updatePicture` function, and we are logging out the user using the `logout` function. When the user logs out, we also remove the value of the `userInfo` key from `LocalStorage`.

Template

The following code should present in `accounts.html`:

```
<!-- /app/pages/account/account.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Account</ion-title>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list>
        <ion-card>
            <img *ngIf="!user.picture" src="img/default.jpg"/>
            <img *ngIf="user.picture" src="{{user.picture}}"/>
        </ion-card>
        <ion-item>
            <button full (click)="updatePicture()">Change
             Picture</button>
        </ion-item>
        <a ion-item primary (click)="logout()">
            Logout
        </a>
    </ion-list>
</ion-content>
```

Adding style

Ionic 2 uses **Syntactically Awesome Style Sheets** (**SASS**) for styling applications, so it's very easy to change the overall look and feel of an application just by changing some values. We will change the values of some default variables to make our application look unique.

Styles are defined in the `app/theme` folder. Inside this folder, there is an `app.variable.scss` file. In this file, Ionic has defined the default colors. Let's change the color values of the primary and secondary Ionic colors to this:

```
primary:    #3F51B5,
secondary:  #32db64,
favorite:   #7986CB
```

Running our app

So far, we have completed all the stages of building our app, but we haven't yet tested in on any actual devices, so let's build it for a mobile device.

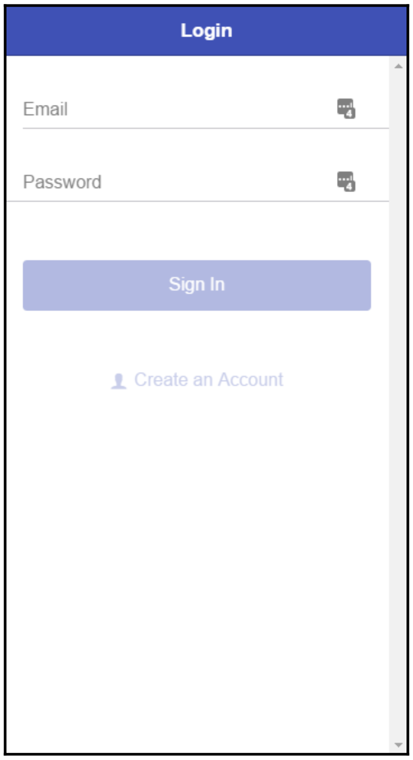
The following commands will add a platform and run the app on a mobile device:

```
ionic platform add android
ionic run android
```

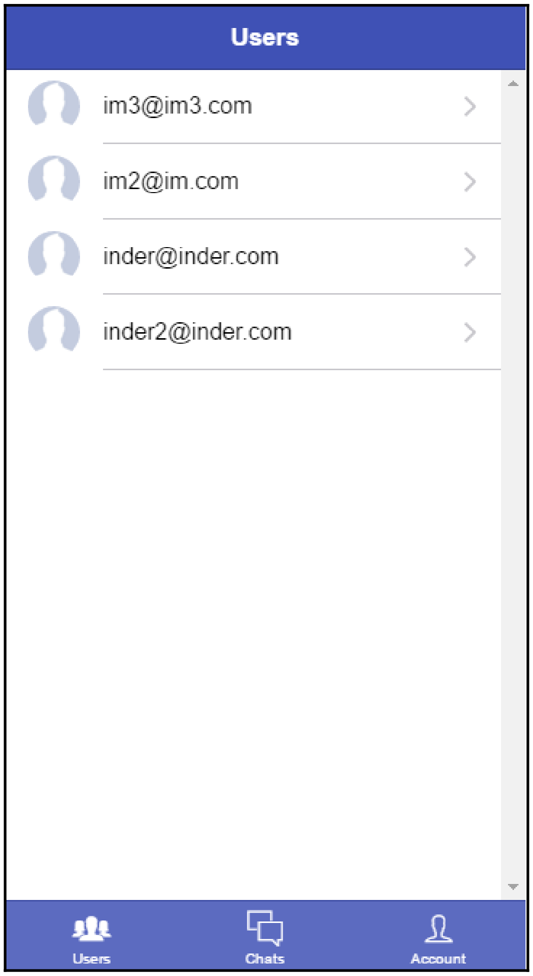
App screenshots

The following are actual screenshots of our chat application.

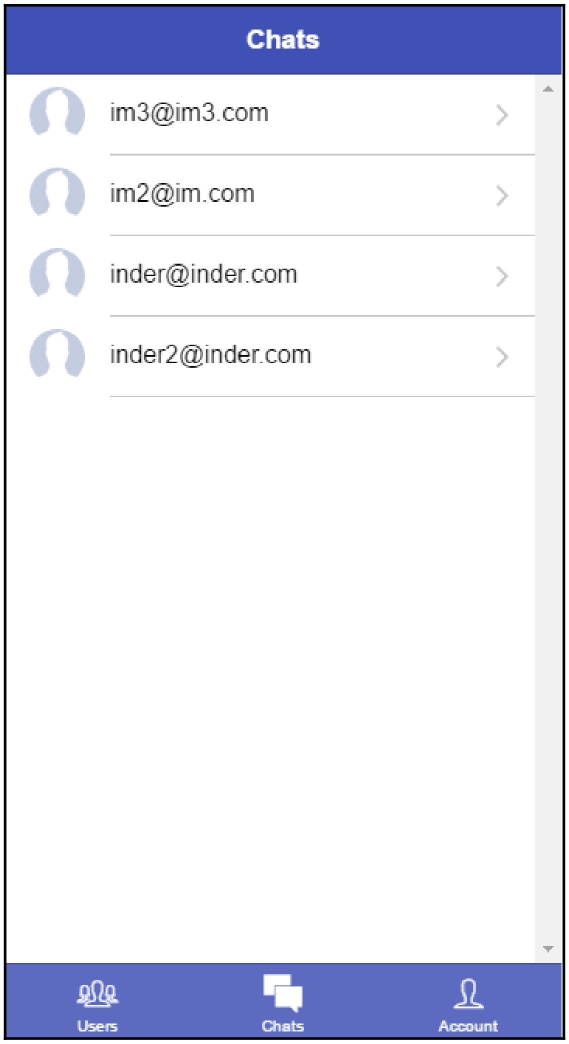
The following screenshot is of the `LoginPage`:



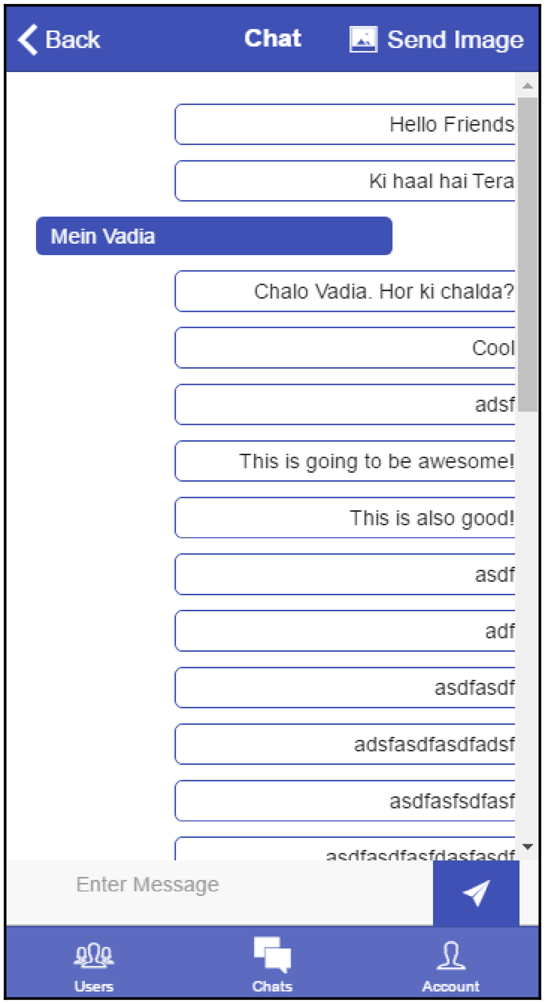
The following screenshot is of the `UsersPage`:



The following screenshot is of the `ChatsPage`:



The following screenshot is of the `ChatsViewPage`:



Hello Friends

Ki haal hai Tera

Mein Vadia

Chalo Vadia. Hor ki chalda?

Cool

adsf

This is going to be awesome!

This is also good!

asdf

adf

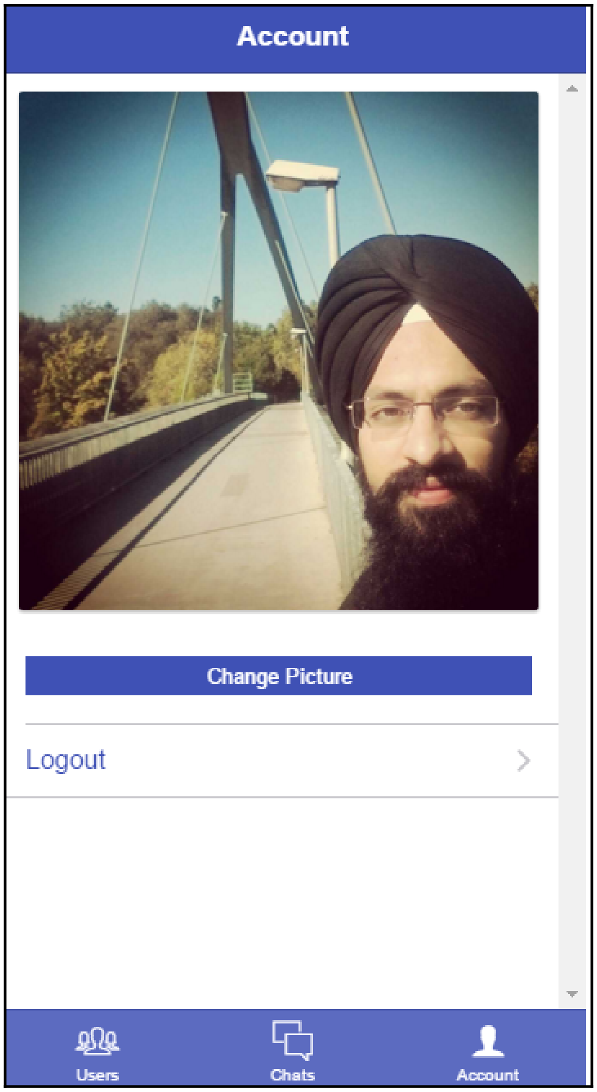
asdfasdf

adsfasdfasdfadsf

asdfasfsdfasf

asdfasdfasfdasfasdf

The following screenshot is of the `AccountPage`:



Summary

In this chapter, we have learned how to use Firebase as a backend for our Ionic application. Along with that, we have also organized our app by dividing our app's functionality into different providers. We have used our mobile device's gallery to get pictures. Along with all these things, we have used Ionic 2's UI elements and services to build a complete app.

In the next chapter, we will create an e-commerce application using Ionic and Marketcloud. Interestingly, we will still not be using any custom backend for our e-commerce app, just like in this chat application.

2
E-Commerce App with Marketcloud

In the previous chapter, we built a real-time chat application using Ionic 2 and Firebase and we utilized the device's gallery to send picture messages. In this chapter, we will create an e-commerce application using **Marketcloud** and Ionic 2. We will create a full e-commerce application, which includes categories, product lists, acart, and checkout.

We will learn the following:

- Marketcloud JavaScript SDK
- Marketcloud Dashboard
- Using the Ionic menu and navigation
- Ionic loading
- Payments

Introduction

In the chapter, we are building an e-commerce application that supports multiple categories, product lists, a cart, and a checkout. So, it is a full e-commerce application.

First we will understand what Marketcloud is and how we can utilize it in our Ionic 2 applications. Then we will create our Marketcloud store and add categories and products. Then we will create our frontend application using Ionic.

Our e-commerce application will first show `StorePage`, which will list all the categories of the store. It will also have a `CategoryPage`, which will list all the products for a particular category. Each product will have its own `ProductPage`. There will be a `CartPage`, where users can edit their cart and see the total, and there will be an `OrderPage`, where users will order the items from their cart.

Marketcloud

Marketcloud is an e-commerce **Backend as a Service** (**BaaS**). It took me almost five full days to think about the best backend for this e-commerce application. I started with **WooCommerce**, then **Magento**, then **Moltin**, which is a similar service, and finally, I found this amazing service: Marketcloud. Marketcloud allows us to create categories, brands, products, and carts, and process orders.

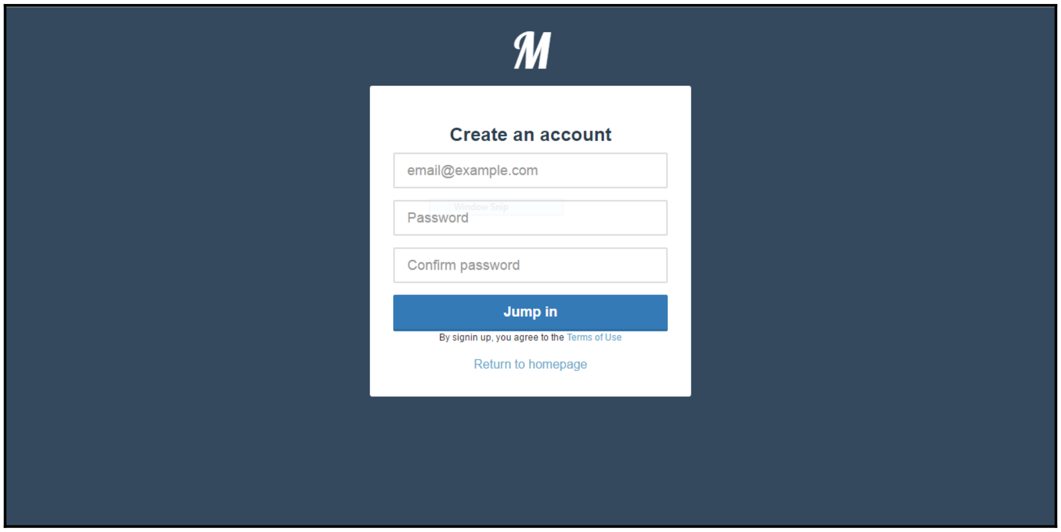
Some features of Marketcloud are:

- Ready-made REST API for your store
- SDKs for various languages and platforms
- Excellent dashboard for doing backend work
- Easy payments with **Braintree**

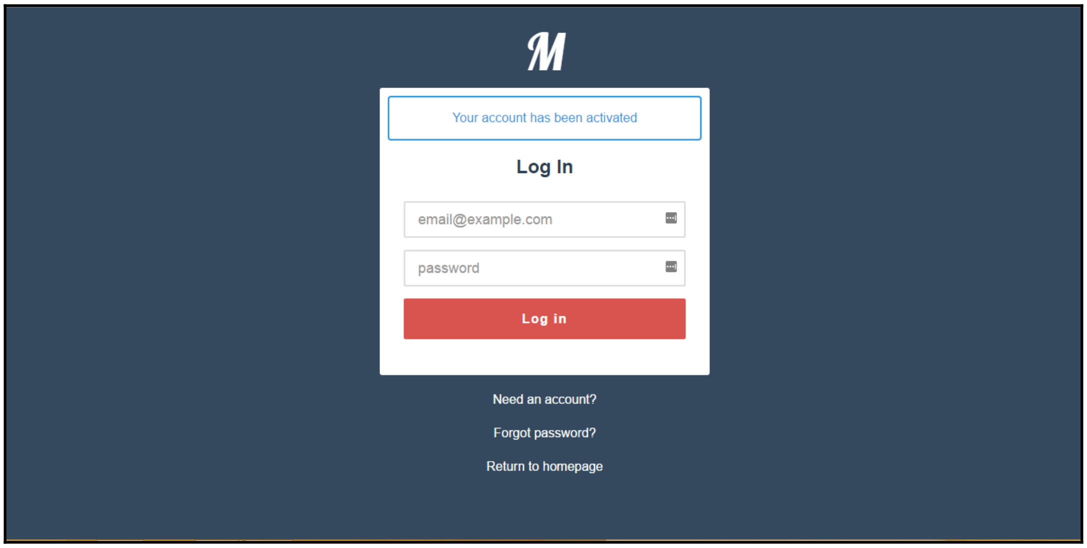
You can learn more about Marketcloud at `http://marketcloud.it/`.

Creating a Marketcloud application

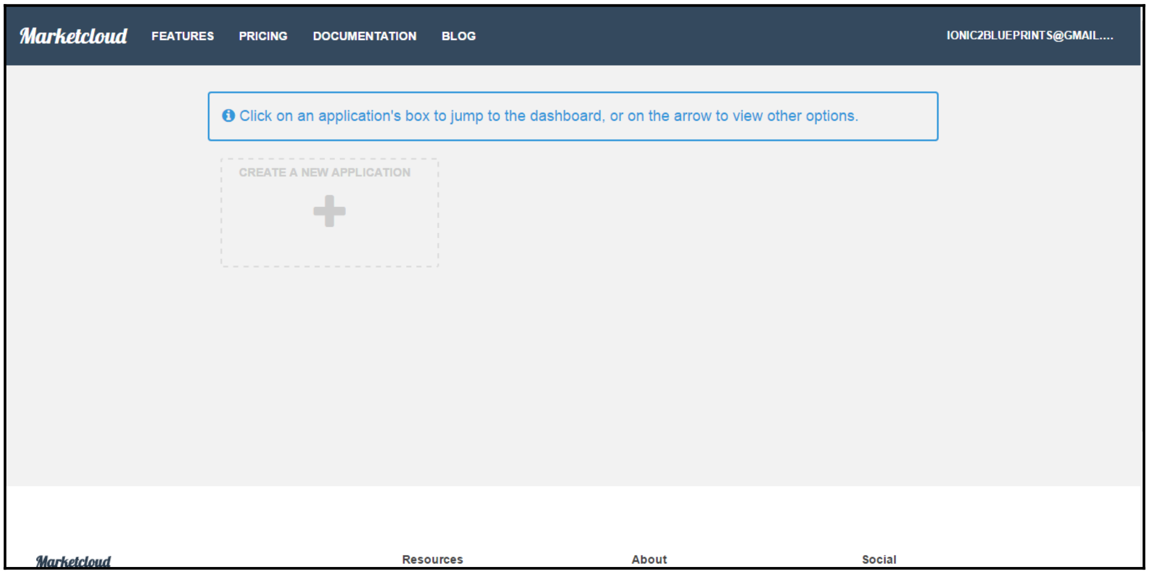
First we need a Marketcloud account. Creating a Marketcloud account is super easy. Just go to `http://marketcloud.it/account/signup`, fill in the form, and click on **Jump in**, as shown in the following screenshot:



An activation e-mail to your e-mail will be sent address. Activate your Marketcloud account by clicking on the link in the e-mail. This will take you to the login page shown in the following screenshot:

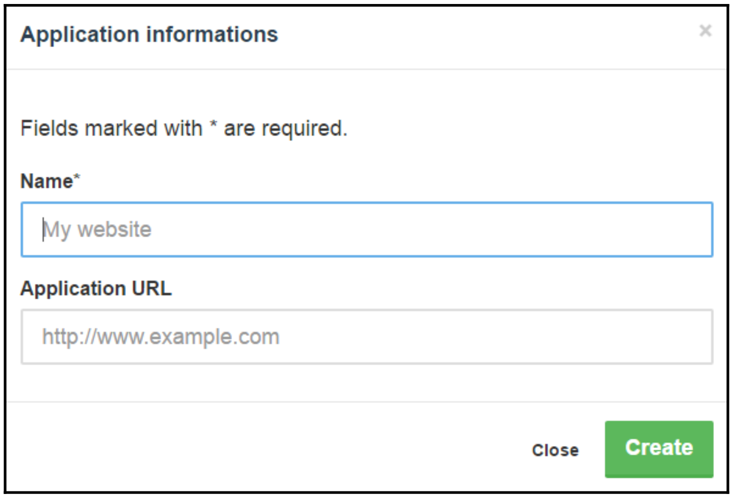


Logging in, will take you to a new page, as shown in the following screenshot:



Click on an application's box to jump to the dashboard, or on the arrow to view other options.

Click on **CREATE A NEW APPLICATION** to open a dialog box, as follows:



Application informations

Fields marked with * are required.

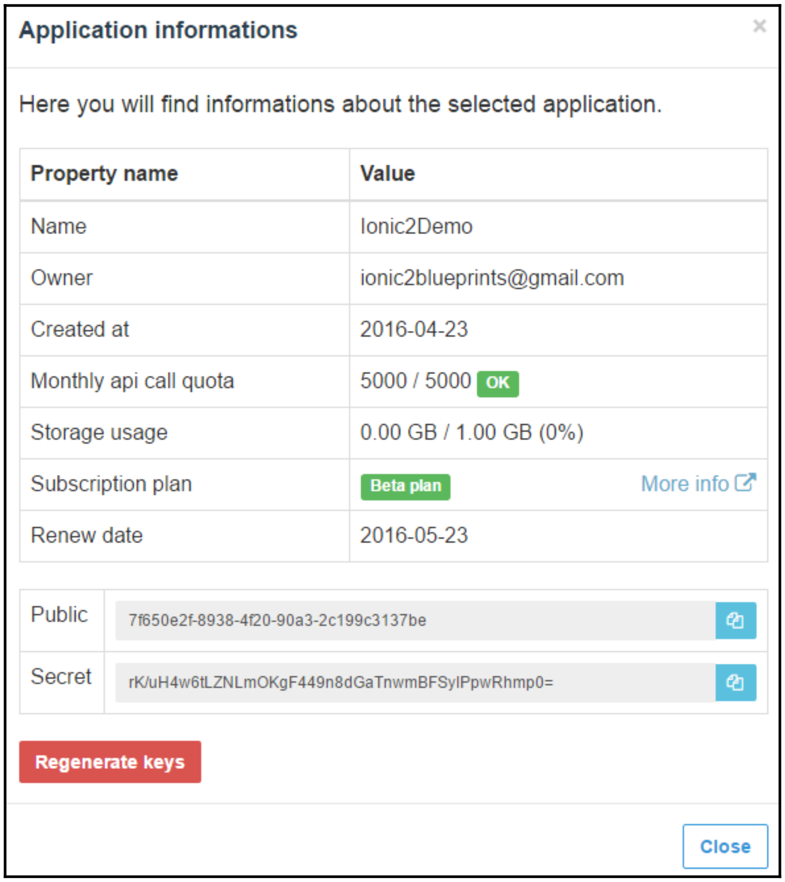
Name*

My website

Application URL

http://www.example.com

Now enter the name of the application and click on **Create**. You just created your new application. Cool! You will see your app as shown in the following screenshot. Click on the down arrow and a drop-down menu will appear. Click on the **Informations** menu item. This will open a new dialog box, as follows:



Application informations

Here you will find informations about the selected application.

Property name	Value
Name	Ionic2Demo
Owner	ionic2blueprints@gmail.com
Created at	2016-04-23
Monthly api call quota	5000 / 5000 OK
Storage usage	0.00 GB / 1.00 GB (0%)
Subscription plan	Beta plan More info
Renew date	2016-05-23

Public	7f650e2f-8938-4f20-90a3-2c199c3137be
Secret	rK/uH4w6tLZNLmOKgF449n8dGaTnwmBFSylPpwRhmp0=

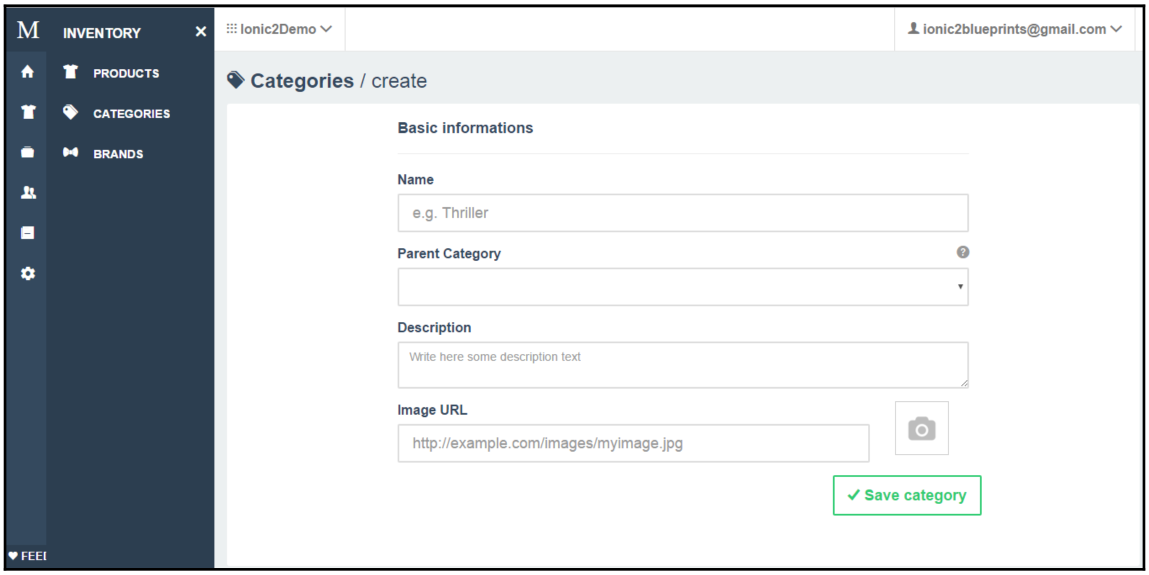
You will see a **Public** field at the bottom. Copy that key and save it somewhere. That key is required to access the Markecloud API. Now open your application by clicking on it. This will take you to dashboard.

Using the Marketcloud dashboard

Marketcloud's dashboard is the place where we will create our categories, products, currency, and other options.

Creating categories

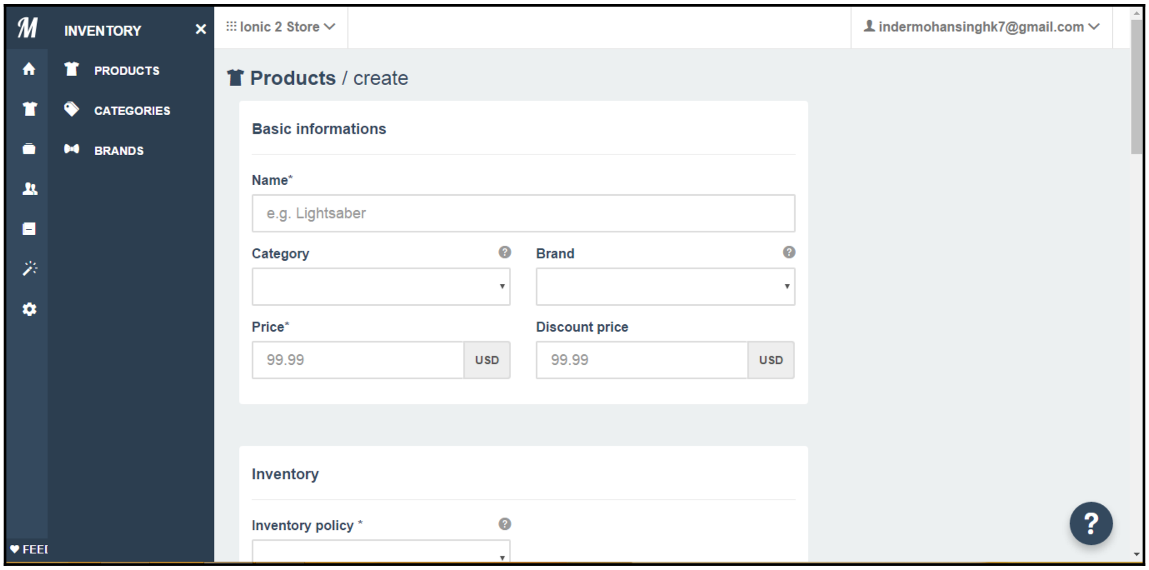
In the side menu, click on **Inventory** and then click on **Category**. This will open a new page. Now click on **Create category** on the right of the page. This will open a new page with a form to create a new category, as follows:



Fill in the form and click on **Save category**.

Creating products

Similarly, in the side menu, click on **INVENTORY** and then **PRODUCTS**. This will open a new page. Now click on **Create Product**. It will again open a new form for you to enter information about your new product. You can also upload images of your product to the form shown in the following screenshot:



Products / create

Basic informations

Name*

e.g. Lightsaber

Category

Brand

Price*

99.99 USD

Discount price

99.99 USD

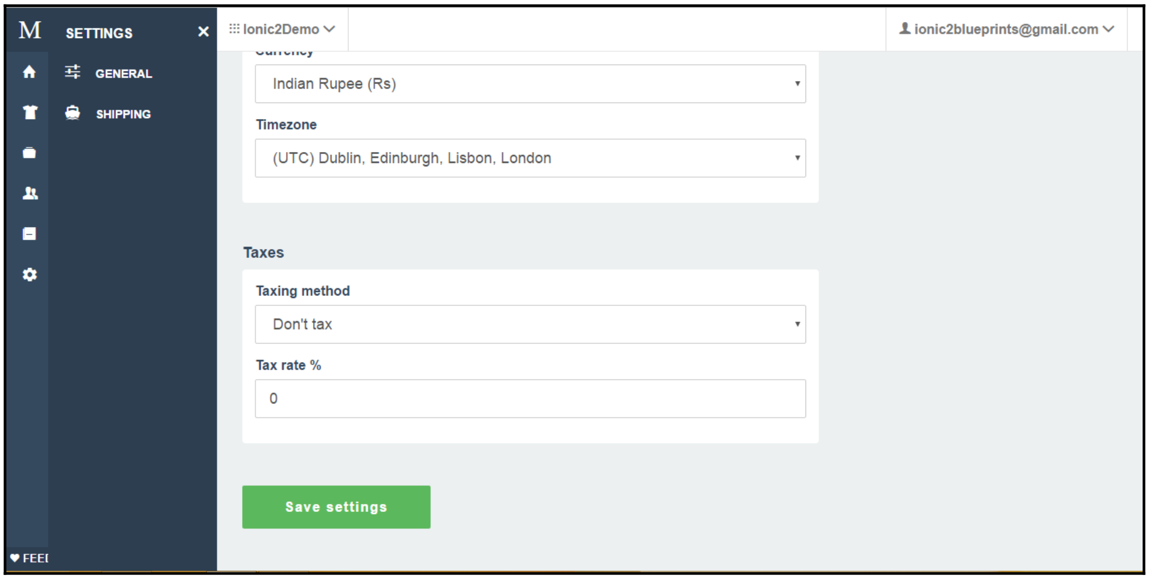
Inventory

Inventory policy *

Similarly, you can create **BRANDS**.

Selecting the currency for our app

You can also select the currency of your application. In the side menu, click on **SETTINGS** and then click on **GENERAL**:



Currency: Indian Rupee (Rs)

Timezone: (UTC) Dublin, Edinburgh, Lisbon, London

Taxes

Taxing method: Don't tax

Tax rate %: 0

In the page, you will see the **Currency** dropdown. Select the currency you need and click on **Save settings**. Change the other settings according to your requirements.

Marketcloud JS SDK

Before we start coding our next million-dollar e-commerce app, let's first dive into the JavaScript SDK of Marketcloud.

Marketcloud SDK is defined in a global variable named `marketcloud`. Before we start querying `marketcloud`, we have to add a public key to our `marketcloud` global variable.

```
marketcloud.public = "your public key here";
```

Now we can query the `marketcloud`. The interesting thing about the Marketcloud JS SDK is its structure. It defines various functionalities in neat member functions, which wrap the REST API calls for us. Here is the list of these functions:

- `marketcloud.categories` has functions to work with the categories of our store
- `marketcloud.product` has functions to work with products
- `marketcloud.carts` has functions to work with carts
- `marketcloud.orders` has functions to work with orders
- `marketcloud.users` allows us to work with users of our app
- `marketcloud.payments` allows us to update payment of an order
- `marketcloud.addresses` allows us to work with the addresses of users

API reference

Here we provide a brief explanation of the Marketcloud JS SDK API.

Listing all categories

The following code will list all of the categories of our application:

```
marketcloud.categories.list({}, (error, categories) => {
  // category variable has our list of categories
  // error variable stores the errors
});
```

Listing all products under a category

The following code will list all of our products under a specific category:

```
marketcloud.products.list({category_id: ctegoryID}, (error, products) => {
  //products variable will have all our products.
})
```

Here, `categoryID` is the the ID of the category, which we will provide.

Creating a cart

The following code will create a new `cart`, where we can add new items for checkout:

```
arketcloud.carts.create({items:[]}, (err, cart) => {
  // Return Cart information is in cart variable
});
```

The `items` array is the array where we will add our products, as a member.

Similarly, we have other functions for carts, as follows:

- `marketcloud.carts.add` adds items to the cart
- `marketcloud.carts.remove` removes items from the cart
- `marketcloud.carts.update` updates the cart

Creating an order

The following code snippet will create a new `order`:

```
marketcloud.orders.create(order, (error, orderData) => {
});
```

Here, the `order` argument is an object with the following structure:

```
order: {
  shipping_address: address,
  billing_address:address,
  items:items
}
```

Creating a payment record

The following code snippet will update the payment of a particular record:

```
marketcloud.payments.create(obj, (err, data) => {
});
```

`obj` is an object with three values:

- `method`: This is the type of payment integration. For example, Braintree, Stripe, and others.
- `order_id`: This will be the `order_id` of that particular order.
- `nonce/source`: This third parameter is either nonce or source, depending upon the payment gateway.

For example: `{ method: 'Braintree', order_id: 76767, nonce: 'Braintree Generated nonce here'}`

To have a complete look at the API, go to `http://www.marketcloud.it/documentation/javascript`.

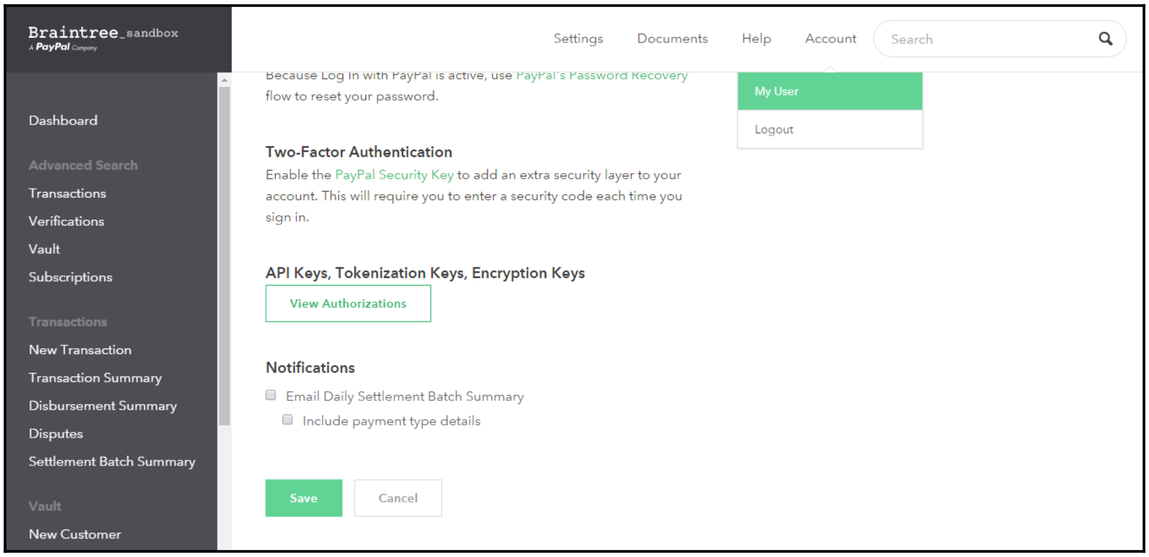
Payment integration

Marketcloud supports integration with payment gateways. So, we don't have to write any server side code for payment processing too. That is one of the best things about Marketcloud. Currently, Marketcloud supports Braintree and Stripe. We will be using Braintree in this application, but you can easily switch to Stripe too. To do so, take the following steps:

1. Go to `https://www.braintreepayments.com/` and create an account.

I highly recommend creating a sandbox account while in the development stages of your application.

2. Now, log in to your account and click on **Account** | **My User**. Then click on **View Authorizations**, as shown in the following screenshot:



Because Log In with PayPal is active, use PayPal's Password Recovery flow to reset your password.

Two-Factor Authentication

Enable the PayPal Security Key to add an extra security layer to your account. This will require you to enter a security code each time you sign in.

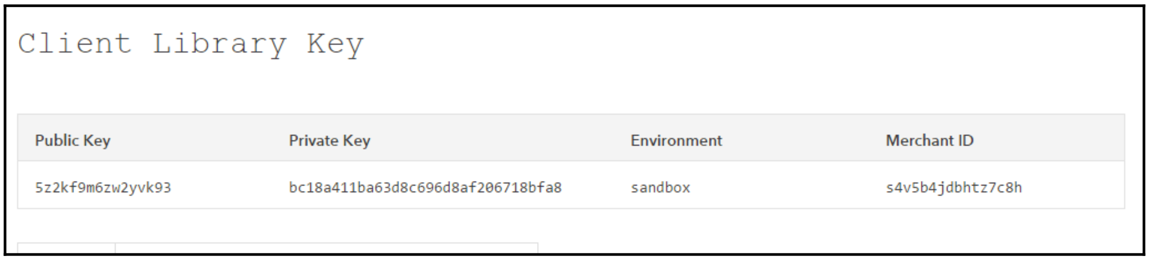
API Keys, Tokenization Keys, Encryption Keys

Notifications

☐ Email Daily Settlement Batch Summary

☐ Include payment type details

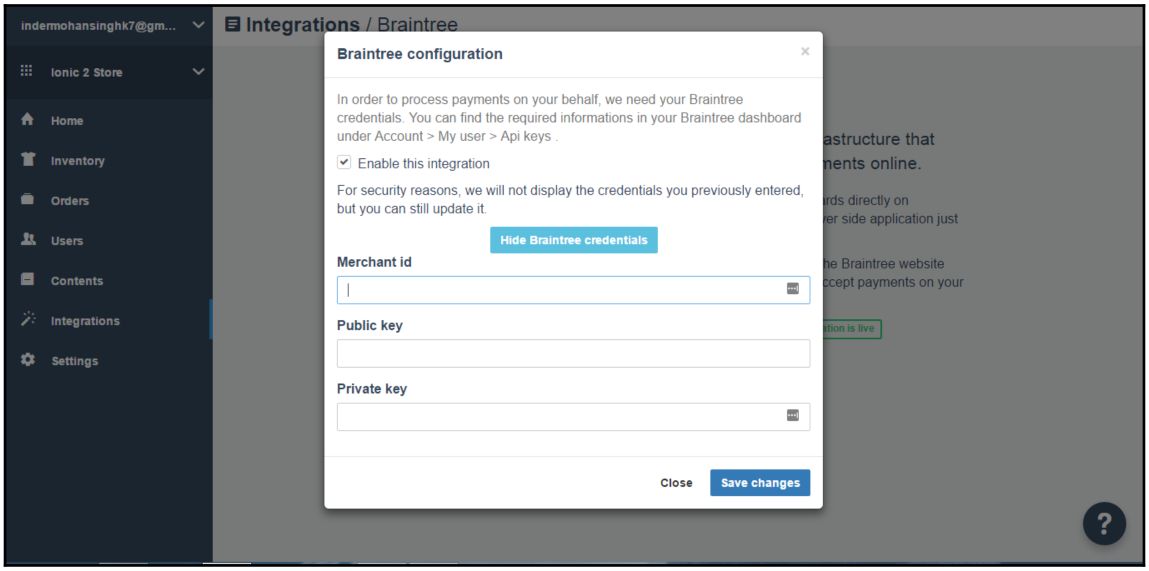
3. It will open a new page. That page has a **Client Library Key** section, as shown in the following screenshot. For there, we need **Merchant ID**, **Public Key**, and **Private Key**:



Client Library Key

Public Key	Private Key	Environment	Merchant ID
5z2kf9m6zw2yvk93	bc18a411ba63d8c696d8af206718bfa8	sandbox	s4v5b4jdbhtz7c8h

4. Now open your Marketcloud dashboard and click on Integrations in the side menu.
5. Choose Braintree and configure it by providing the **Merchant ID**, **Public Key**, and **Private Key**, which we got from Braintree, as shown in the following screenshot:



Braintree configuration

In order to process payments on your behalf, we need your Braintree credentials. You can find the required informations in your Braintree dashboard under Account > My user > Api keys .

☑ Enable this integration

For security reasons, we will not display the credentials you previously entered, but you can still update it.

To know more go to `http://www.marketcloud.it/documentation/integrations/braintree` and read the official integration procedure for Braintree.

Defining our app

In this section we will define various functionalities of our application.

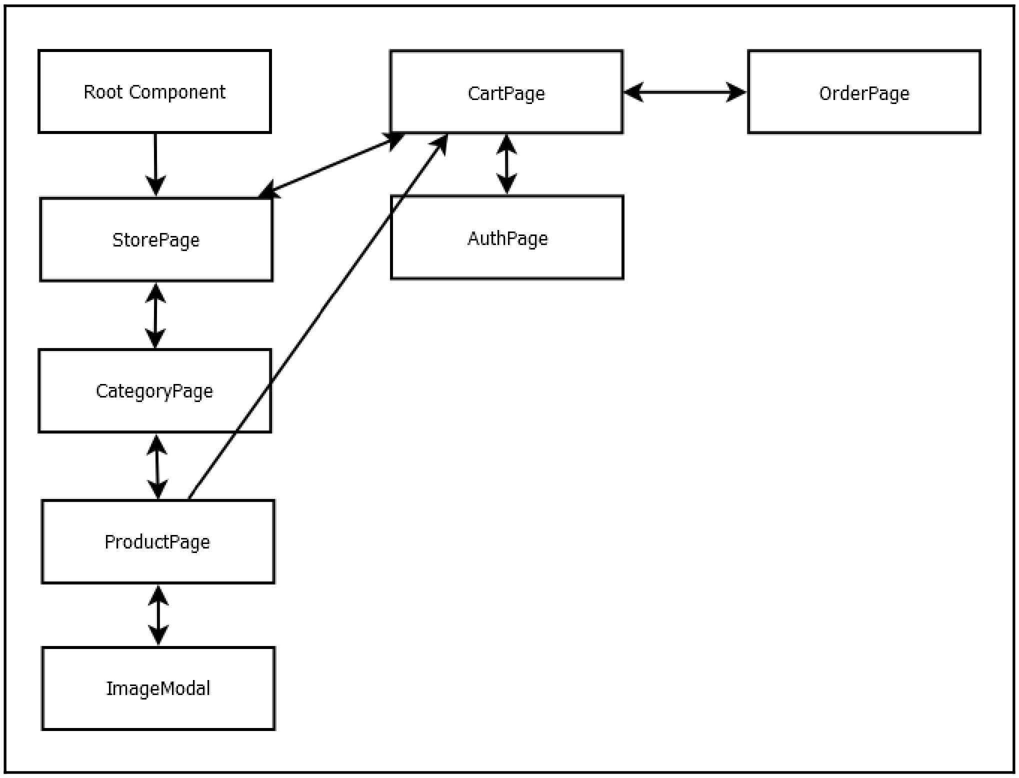
Functionalities

We will be including the following functionalities in our application:

- Product categories
- Product list
- Client-side product search
- Product image gallery
- User authentication
- Cart
- Order

App flow

The following figure shows how the control will flow inside our application:



Let's understand the flow:

- **RootComponent**: This is the root ionic component. It is defined inside the `/app/app.ts` file.
- **StorePage**: This shows all the categories of our application. When you click on a **Category**, you will go to `CategoryPage`.
- **CategoryPage**: This shows all the products of the selected category. When you click on a **Product**, you will go to `ProductPage`.
- **ProductPage**: This shows the details of a product. It allows you to add a product to the cart and see the images of the product. It also allows you to go to `CartPage`.
- **ImageModal**: This is the Ionic modal, which shows all the images of a product. You can go back to `ProductPage` from `ImageModal`.
- **CartPage**: This shows the items in the cart and the total. It allows you to check out. When the user checks out, if he is logged in, then `OrderPage` appears; otherwise, `AuthPage` appears.
- **AuthPage**: This allows the user to log in and register for our application. When the user logs in, `AuthPage` takes you back to `CartPage`.
- **OrderPage**: This shows the address form; when the user clicks on the **Order** button, it allows the user to add credit/debit card details and then pay the money. It then, goes back to `StorePage`.

Scaffolding and setting up our app

Ionic CLI eases the process of scaffolding and setting up an app for us. So, we will be using Ionic CLI, which we installed earlier. We will start by scaffolding a blank application. Run this command:

```
ionic start ionic2-marketcloud sidemenu --v2 --ts
```

Notice the `--v2` and `--ts` tag, which scaffolds the Ionic 2 app based on TypeScript.

Using the `cd` command, go to the `ionic2-marketcloud` folder and run the following command to check how the side menu app looks:

```
ionic serve
```

Before we start writing the code for our app, we need to install some project-specific dependencies.

Installing dependencies

Following are some project-specific dependencies that we need to install.

Installing MarketCloud JS SDK

The following command installs `MarketCloud JS`:

```
bower install marketcloud-js
```

This file will be downloaded in the `bower_components` folder.

Create a folder named `vendor` inside `/www` and move the `marketcloud-min.js` file into it.

Installing the BrainTree Cordova plugin

We are using BrainTree as our payment gateway, so we need to install their Cordova plugin to show a drop-in payment form; that can be done by the following command:

```
ionic plugin add cordova-plugin-braintree
```

For more information on plugin, go to `https://github.com/Justin-Credible/cordova-plugin-braintree/`.

Coding our app

Now we have everything set up to start working on our application. Let's define the following:

- Our `index.html` for the app
- Our main `app.ts` file
- Providers/services for various functionalities
- Ionic pages for various views.

Our index.html file

The `index.html` file is the main file of our application. Ionic generates this file automatically when we scaffold our application. We just need to add the reference to Marketcloud SDK. It is in the `www` folder of our app and references the Marketcloud SDK, just after `cordova.js`, as follows:

```
<script src="vendor/marketcloud.min.js"></script>
```

Defining app.ts

This is the root of our application and it defines the `root` component. This is where we inject all of our dependencies. The following code should be present in `app.ts`:

```
/* /app/app.ts */
import {Component, ViewChild, provide, enableProdMode} from
'@angular/core';
import {Platform, Nav, Events, ionicBootstrap} from 'ionic-angular';
import {StatusBar} from 'ionic-native';
import {StorePage} from './pages/store/store';
import {CartPage} from './pages/cart/cart';
import {MarketProvider} from './providers/market-provider/market-provider';
import {CartProvider} from './providers/cart-provider/cart-provider';
import {StorageProvider} from './providers/storage-provider/storage-
provider';
import {UserProvider} from './providers/user-provider/user-provider';
import {UtilProvider} from './providers/util-provider/util-provider';
declare let marketcloud:any;
@Component({
  templateUrl: 'build/app.html',
})
class MyApp {
  rootPage: any = StorePage;
  pages: Array<{title: string, component: any}>
  isLoggedIn = false;
  @ViewChild(Nav) nav:Nav;
  constructor(private platform: Platform,
  public cartProvider:CartProvider,
  public market: MarketProvider,
  public storage: StorageProvider,
  public events:Events,
  public userProvider: UserProvider) {
    this.initializeApp();
    this.pages = [
      { title: 'Categories', component: StorePage },
      { title: 'Cart', component: CartPage}
    ];

    events.subscribe('user:logged_out', () => {
       this.isLoggedIn = false;
    });
    events.subscribe('user:logged_in', () => {
      this.isLoggedIn = true;
    });
  }

  initializeApp() {
    this.platform.ready().then(() => {
      StatusBar.styleDefault();
      let user = this.storage.getObject('user');
      if(user) {
        this.market.getMarketCloud().token = user.token;
        this.market.getMarketCloud().user = user.user;
        this.isLoggedIn = true;
      }
      this.cartProvider.intializePayments();
    });
  }

  openPage(page) {
    this.nav.setRoot(page.component);
  }
  logout() {
    this.userProvider.logout();
    this.events.publish('user:logged_out', {});
  }
}

ionicBootstrap(MyApp, [MarketProvider, CartProvider, StorageProvider,
UserProvider, UtilProvider]);
```

This is the controller of our `root` component. We have defined the `rootPage` of our app as `StorePage`. Each function does its work as follows:

- `constructor()` is where we create variables such as `pages`, which shows a list of pages for the side menu list. We are also subscribing to Ionic custom events using `this.event`. Take a look at Ionic's events service to learn more about it at `http://ionicframework.com/docs/v2/api/util/Events/`. We are also initializing our app here by calling `initializeApp()`.

- `initializeApp()`is for doing stuff when the platform is ready. In it, we are checking if the user is logged in already by querying `localStorage`. If the user is logged in, we add that information to the global `marketcloud` object using the `this.market.getMarketCloud()` function of `market-provider`. We are also initializing our BrainTree plugin for payments here.
- `openPage(page)` is for opening a page from the side menu. It basically gets the page from the `this.pages` list and sets that page as the root page.
- `logout()` logs out our application by removing the reference from `localStorage` and broadcasting a logout event using Ionic's `Events.publish`.
- `ionicBootstrap` bootstraps our Ionic 2 app. We have also injected the providers of our application.

Template

The following code should be present in `app.html`:

```
<!--  Template app/app.html -->
<ion-menu [content]="content">

  <ion-toolbar primary>
    <ion-title>Menu</ion-title>
  </ion-toolbar>

  <ion-content>
    <ion-list>
      <button menuClose ion-item *ngFor="let p of pages"
       (click)="openPage(p)">
        {{p.title}}
      </button>
      <button menuClose ion-item *ngIf="isLoggedIn" (click)="logout()">
        Logout
      </button>
    </ion-list>
  </ion-content>
</ion-menu>

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

We have just created an `ion-toolbar` and created our side menu using `ion-menu`. Finally, we have created a navigation component using `ion-nav` and set the root to `rootPage`. It is really important to notice that we have created a local reference `#content` for our `ion-nav` and we have used that in our `ion-menu` tag's content property.

Providers for our application

We will have the following providers.

- `StorageProvider`
- `MarketProvider`
- `CartProvider`
- `UserProvider`
- `UtilProvider`

Defining StorageProvider

The `StoreageProvider` class is a very simple class. It allows us to interact with `LocalStorage`. You can get data to and from `localStorage` and remove data from `localStorage`.

Ionic also provides the `Storage` and `Localstorage` class to work with `localStorage`. Their API is based on `Promise`. But we need very simple functionalities, so I created a new class. You can also use that one.

Generating a provider

The following code generates the `storageProvider`:

```
ionic g provider storageProvider
```

This will create `storage-provider.js` files in the `app/provider/storage-provider` directory, with some default content.

Changing the extension

Now, we have to change the extension of the file to `.ts` to use TypeScript in our code.

StorageProvider code

In this class, we abstracted our interaction with `localStorage`:

```
/* /app/providers/storage-provider/storage-provider.ts */
import {Injectable} from '@angular/core';

@Injectable()
export class StorageProvider {
  storage = localStorage;
  get(key) {
    return this.storage.getItem(key);
  }
  set(key, value) {
    this.storage.setItem(key, value);
  }
  getObject(key) {
    let value = this.get(key);
    let returnValue;
    if(value) {
      returnValue = JSON.parse(value);
    } else {
      returnValue = null;
    }
    return returnValue;
  }
  setObject(key, value) {
    this.storage.setItem(key, JSON.stringify(value));
  }
  remove(key) {
    this.storage.removeItem(key);
  }
}
```

Let's understand the preceding code:

- `constructor()` creates a new class member named `storage` and assigns `localStorage` to it
- `get(key)` gets the content of `key` and returns it
- `set(key, value)` sets the content of `key` to a given `value`
- `getObject(key)` gets the content of `key` and parses the content with `JSON.parse` to convert it to an object
- `setObject(key, value)` assigns the content of `key` to a given object in `value` and that value to a string with `JSON.stringify`
- `remove(key)` removes the value at `key` from `localStorage`

Defining UtilProvider

We generate our provider `UtilProvider` with the following command:

```
ionic g provider utilProvider
```

We then change its extension to `.ts`:

UtilProvider code

The `UtilProvider` module is for abstracting features that we will repeatedly use, such as loading dialogs and alerts. The following code should be present in `util-provider.ts`:

```
// app/providers/util-provider/util-provider.ts
import {Injectable} from '@angular/core';
import {Alert, Loading, Toast} from 'ionic-angular';
@Injectable()
export class UtilProvider {
  doAlert(title, message, buttonText) {
      let alert = Alert.create({
          title: title,
          subTitle: message,
          buttons: [buttonText]
      });
      return alert;
  }
  presentLoading(content) {
    let loading = Loading.create({
      dismissOnPageChange: true,
      content: content
    });
    return loading;
  }

  getToast(message) {
    let toast = Toast.create({
      message: message,
      duration:2000
    });
    return toast;
  }
}
```

Let's understand the preceding code:

- `doAlert(title, message, buttonText)` takes three arguments, the `title` for alert box, its `message`, and `buttonText`, which is the text of the button, and then returns the Ionic alert box
- `presentLoading(content)` takes `content` to display as an argument and returns a loading service's instance
- `getToast(message)` takes a message as an argument and returns an Ionic toast with that message

Defining MarketProvider

First we will generate our provider using the following command:

```
ionic g provider marketProvider
```

Then, change the extension of `user-provider.js` to `market-provider.ts`.

MarketProvider code

The `MarketProvider` module is for interacting with Marketcloud to get categories, products, and all that stuff. The following code should be present in `market-provider.ts`:

```
/* /app/providers/market-provider/market-provider.ts */
import {Injectable} from '@angular/core';
declare let marketcloud:any;
@Injectable()
export class MarketProvider {
  market:any;
  constructor() {
    marketcloud.public = "9c7ef560-5f8b-4d53-a12f-e9d9333a3cef";
    this.market = marketcloud;
  };
  getMarketCloud() {
    return this.market;
  }
  getCategories() {
    let promise = new Promise((resolve, reject) =>{
     this.market.categories.list({}, (error, categories) => {
       if(categories) {
         resolve(categories);
       } else {
         reject(error);
```

```
          }
        });
      })
      return promise;
    };
    getProducts(categoryID) {
      let promise = new Promise((resolve, reject) =>{
        this.market.products.list({category_id: categoryID}, (error,
         products) => {
          if(products) {
            resolve(products);
          } else {
            reject(error);
          }
        });
      });
      return promise;
    };
  }
```

Let's understand the preceding code:

- `constructor()` sets the public key of our Marketcloud and also assigns a global `marketcloud` object to `this.market`
- `getMarketCloud()` returns `this.market`
- `getCategories()` gets all the categories from `marketcloud` using the `this.market.categories.list` function
- `getProducts(categoryID)` gets `categoryID` and returns all the products of that category from `marketcloud` using the `this.market.products.list` function

These functions use **ES6 Promises** for asynchronous operations and use Marketcloud's JS SDK.

Defining CartProvider

First, generate our provider using the following command:

```
ionic g provider cartProvider
```

Then change the extension of file `carts-provider.js` to `cart-provider.ts`:

CartProvider code

The `CartProvider` module is used to create the cart, get the contents of the cart, add items to the cart, create an order, and other things. The following code should be present in `cart-provider.ts`:

```
/* /app/providers/cart-provider/cart-provider.ts */
import {Injectable} from '@angular/core';
import {Http, Headers} from '@angular/http';
import {MarketProvider} from '../market-provider/market-provider';
import {StorageProvider} from '../storage-provider/storage-provider';
import 'rxjs/add/operator/map';
declare let BraintreePlugin:any;
@Injectable()
export class CartProvider {
  cartID:string;
  market:any;
  constructor(public marketProvider: MarketProvider, public storage:
   StorageProvider, public http:Http) {
    this.market = this.marketProvider.getMarketCloud();
    let local_cart_id = this.storage.get('cart_id');
    if(local_cart_id) {
      this.cartID = local_cart_id;
    }
  }
```

`constructor()` creates a market member, which is a reference to the global `marketcloud` object. We are using the `getMarketCloud()` function of `market-provider` to get the global `marketcloud` object. We are also setting the `cartID` in the following code, if the cart has already been created, by using `localStorage`:

```
  setCartID(value) {
    console.log('set cart id', value);
    this.cartID = value;
    if(value) {
      this.storage.set('cart_id', value);
    } else {
      this.storage.remove('cart_id');
    }
  }
```

`setCardID(value)` sets `this.cartID` to a given value and also updates it in `localstorage`. The following is the code for `initializePayments()`:

```
intializePayments() {
  let marketcloud_id = this.marketProvider.getMarketCloud().public;
  let promise = new Promise((res, rej)=> {
    this.market.payments.braintree.createClientToken((err, data) => {
      if(err) {
        rej(err);
      } else {
        let token = data.clientToken;
        BraintreePlugin.initialize(token, () => res('done'), (error)
         => rej(error));
        res(data);
      }
    });
  });
  return promise;
}
```

`intializePayments()` initializes our Braintree payments. Every payment gateway uses a token for initializing. Marketcloud allow us to create a token using a `marketcloud.payments.braintree.createClientToken` function and then we initilize our Braintree plugin, using that **token** value. The following code is for `addToCart()`:

```
addToCart(productID, quantity) {
  console.log(this.isCartExist());
  let promise = new Promise((resolve, reject) => {
    // if Cart not already exists
    if(!this.isCartExist()) {
      this.market.carts.create({
        items:[{'product_id' : productID, 'quantity': quantity}]
      }, (err, cart) => {
        if(err) {
          reject(err);
        } else {
          this.setCartID(cart.id);
          resolve(cart);
        }
      });
    } else {  // if Cart exists
        this.market.carts.add(this.cartID, [{'product_id' :
         productID, 'quantity': quantity}], (err, cart) => {
          if(err) {
            reject(err);
          } else {
            resolve(cart);
          }
        });
    }
  });
  return promise;
}
```

addToCart(productID, quantity) adds the product with productID with the quantity given. If the cart exists, it just adds these items to it using this.market.carts.addfunction. If the cart doesn't exist, it creates a new cart using this.market.carts.create. When it creates a new cart, it also adds the cart_id to localStorage. The following code is for isCartExist():

```
    isCartExist() {
      if(this.cartID) {
        return true;
      } else {
        return false
      };
    }
```

isCartExist() basically tells us if the cart already exists or not. The following code is for getCartContents():

```
    getCartContents() {
      let promise = new Promise((resolve, reject) => {
        if(this.cartID !== undefined) {
          this.market.carts.getById(this.cartID, function(err, cart) {
            if(cart) {
              resolve(cart);
            } else {
              reject(err);
            }
         });
        } else {
          reject("no cart created yet");
        }
      });
      return promise;
    }
```

`getCartContents()` gets the content of the cart using `this.market.carts.getById` and `cart_id`, which we have saved in `localStorage`. The following code is for `updateCart()`:

```
updateCart(items) {
  items = items.map((item) => {
      return {product_id: item.product_id, quantity: item.quantity};
  });
  let promise = new Promise((resolve, reject) => {
    this.market.carts.update(this.cartID, items, (error, cart) =>{
      if(cart) {
        resolve(cart);
      } else {
        reject(error);
      }
    })
  });
  return promise;
}
```

`updateCart(items)` takes items as its input and updates the current cart with those items. It is important to note that we use the array's `map` function to create a new array of items, which only has the `product_id` and `quantity` of each item. The following code is for `removeItem()`:

```
removeItem(item_id) {
  let promise = new Promise((resolve, reject) => {
    this.market.carts.remove(this.cartID,[{product_id: item_id}],
     (error, cart) =>{
      if(cart) {
        resolve(cart);
      } else {
        reject(error);
      }
    })
  });
  return promise;
}
```

`removeItem(item_id)` takes `item_id` as an argument and removes that item from the current cart. The following code is for `createOrder()`:

```
createOrder(items, address) {
  items = items.map(function(item) {
    return {product_id:item.product_id, quantity: item.quantity};
  });
  let order = {
    shipping_address: address,
    billing_address: address,
    items: items
  };
  let promise = new Promise((resolve, reject) => {
    this.market.orders.create(order,(error, data) =>{
      if(data) {
        resolve(data);
      } else {
        reject(error);
      }
    })
  });
  return promise;
}
```

`createOrder(items, address)` takes a list of `items` and `address` as an argument and creates a new `order` using the `this.market.orders.create` function. It is also important to note that we are using `address` as `shipping_address` and `billing_address`. The following code is for `getPayments()`:

```
  getPayments(amount, user) {
    var options = {
      cancelText: "Cancel",
      title: "Purchase",
      ctaText: "Select Payment Method",
      amount: "$" + amount.toString(),
      primaryDescription: "Your Item",
      secondaryDescription :"Free shipping!"
    };
    var promise = new Promise((res, rej) => {
      console.log('braintree payment called');
      console.log(BraintreePlugin);
      BraintreePlugin.presentDropInPaymentUI(options, function (result)
      {
        if (result.userCancelled) {
            rej('user cancelled');
        }
        else {
           res(result);
        }
      });
    });
    return promise;
  }
```

getPayment(amount, user) takes amount and user as an argument and then, using the BraintreePlugin.presentDropInPaymentUI function, we show the payment form, which gets payment from the user. In options, amount is the order's total cost, which is most relevant here. The rest of the options are just a bunch of string values, shown in the payment screen. You can read about this in the plugin documentation. The following code is for createPayments():

```
      createPayments(order_id, nonce) {
        let promise = new Promise((res, rej) => {
          this.market.payments.create({
            method: 'Braintree',
            order_id: order_id,
            nonce: nonce
          }, (err, result) => {
            if(err) {
              rej(err);
            } else {
              res(result);
              this.setCartID(undefined);
            }
          });
        });

        return promise;
      }
    }
```

createPayment() actually updates the payment status of any order to be paid, so that we have a record of payment of each order in our Marketcloud dashboard.

Defining UserProvider

First, generate our provideruserProvider using the following command:

```
ionic g provider userProvider
```

Then, change the extension of the file user-provider.js to user-provider.ts.

UserProvider code

The UserProvider class provides user-related work such as creating a user and authenticating a user. The following code should be present in userProvider.ts:

```
// /app/providers/user-provider/user-provider.ts
import {Injectable} from '@angular/core';
import {MarketProvider} from '../market-provider/market-provider';
import {StorageProvider} from '../storage-provider/storage-provider';
declare let marketcloud:any;
@Injectable()
export class UserProvider {
  market:any;
  constructor(public marketProvider: MarketProvider, public
   storage:StorageProvider) {
   this.market = this.marketProvider.getMarketCloud();
  }

  isLoggedIn() {
    let user = this.storage.getObject('user');
    if(user) {
      return true;
    } else return false;
  }
  createUser(user) {
    let promise = new Promise((resolve, reject) => {
      console.log(user);
      this.market.users.create(user, (err, user) => {
        if(user) {
          resolve(user);
        } else {
          reject(err);
        }
      })
    });
    return promise;
  }

  authUser(user) {
    let promise = new Promise((resolve, reject) => {
        this.market.users.authenticate(user.email, user.password, (err,
         data) => {
          if(err) {
            reject(err);
          } else {
            resolve(data);
          }
        })
      });
     return promise;
  }

  logout() {
    this.storage.remove('user');
    marketcloud.token = null;
```

```
      delete marketcloud.user;
    }

  getCurrentUser() {
    let promise = new Promise((resolve, reject) => {
        this.market.users.getCurrent((err, user) => {
          if(user) {
            resolve(user);
          } else {
            console.log(err);
            reject(err);
          }
        })
      });
      return promise;
    }
   getAddress() {
     let promise = new Promise((resolve, reject) => {
        this.market.addresses.list({},(err, address) => {
          console.log(err,address)
          if(address) {
            resolve(address);
          } else {
            reject(err);
          }
        })
     });
     return promise;
   }
   createAddress(address) {
     let promise = new Promise((resolve, reject) => {
       this.market.addresses.create(address, (err, address) => {
         if(address) {
           resolve(address);
         } else {
           reject(err);
         }
       });
     })
     return promise;
   }
  }
```

Let's understand the preceding code:

- `constructor()` just creates a new reference to the global `marketcloud` object and assigns it to `this.market`.
- `isLoggedIn()` checks if the user is already logged in or not, by using `LocalStorage`.
- `createUser(user)` creates a new `user` using the `this.market.users.create` function.
- `authUser(user)` authenticates a user using their e-mail and password by calling the `this.market.users.authenticate` function.
- `Logout()` logs out the user from the application.
- `getCurrentUser()` returns a `promise`, which resolves to the currently logged-in user and, if there is any error, `promise` rejects with the error.
- `getAddress()` returns a promise. It uses the `this.market.addresses.list` function to get the address of the current user. It resolves with `address` and rejects with an error.
- `createAddress(address)` returns the `promise`. It creates a new `address` for the currently logged-in user.

Application pages

Now we have to define all the providers and services for our application. Let's define application pages.

We will have the following pages in our application:

- `StorePage`
- `CategoryPage`
- `ProductPage`
- `ImageModal`
- `CartPage`
- `AuthPage`
- `OrderPage`

Defining StorePage

`StorePage` lists all the categories for store. The following code should be present in `store.ts`:

```
// /app/pages/store/store.ts
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {MarketProvider} from '../../providers/market-provider/market-
provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import {CategoryPage} from '../category/category';
@Component({
  templateUrl: 'build/pages/store/store.html'
})
export class StorePage {
  categories:any;
  constructor(public market: MarketProvider, public nav:NavController,
   public util:UtilProvider) {
    this.market.getCategories()
    .then((data) => {
       this.categories = data;
    });
  }

  openProducts(category) {
    this.nav.push(CategoryPage, {category: category});
  }
}
```

We get a list of categories using the `getCategories` member of `market-provider` in the constructor and assign those categories to `this.categories`. The `openProduct` function pushes `CategoryPage` with the category as `NavParams` to the navigation stack.

Template

The following code should be present in `store.html`:

```
<!-- Template: /app/pages/store/store.html -->
<ion-header>
  <ion-navbar primary>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Ionic 2 Store</ion-title>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-list>
     <a ion-item text-wrap *ngFor="let category of categories"
      (click)="openProducts(category)">
        <ion-thumbnail item-left>
          <img [src]="category.image_url" *ngIf="category.image_url">
          <img src="images/default_category.jpg"
           *ngIf="!category.image_url">
        </ion-thumbnail>
        <h2>{{category.name}}</h2>
        <p [innerHTML]="category.description"></p>
     </a>
  </ion-list>

</ion-content>
```

We show the list of categories using `ion-list` and `ngFor`. We show a category thumbnail if it has an image, otherwise we show the default image. We also show the category's name and description.

Defining CategoryPage

`CategoryPage` shows a list of products for a particular category. The following code should be present in `category.ts`:

```
// /app/pages/category/category.ts
import {Component} from '@angular/core';
import {NavController, NavParams} from 'ionic-angular';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import {MarketProvider} from '../../providers/market-provider/market-
provider';
import {ProductPage} from '../product/product';
@Component({
  templateUrl: 'build/pages/category/category.html'
})
export class CategoryPage {
  category:any;
  loading:any;
  products:any;
  totalProducts:{};
  constructor(public nav: NavController,
              public params: NavParams,
              public market: MarketProvider,
              public util:UtilProvider) {
     this.category = this.params.get('category');
     this.loading = this.util.presentLoading("Loading Products ...");
     this.nav.present(this.loading);
     this.market.getProducts(this.category.id)
     .then((data) => {
       this.products = data;
       this.totalProducts = data;
       this.loading.dismiss();
    });
  }
  openProduct(product) {
     console.log("Opening Product");
     this.nav.push(ProductPage, {product: product});
  }
  searchProducts(searchbar) {
    console.log(searchbar.value);
    this.products = this.totalProducts;
    let searchValue = searchbar.value;
    if(searchValue.trim() == "") {
      return;
    }
    this.products =  this.products.filter((product) => {
      if(product.name.toLowerCase().indexOf(searchValue.toLowerCase())
       > -1) {
        return true;
      }
      return false;
    });
  }
}
```

We get the category from `NavParams` and then get all the products using the `getProducts` member of `market-provider`. The `openProduct` function pushes `ProductPage` to the navigation stack and also passes the `product` object of the selected product as `NavParam`.

The `searchProduct` function is quite a handy function. It allows us to search the product list by filtering it using the user's input. We return only those products that have user input in their name.

Template

The following code is present in `category.html`:

```
<!-- Template /app/pages/category/category.html -->
<ion-header>
  <ion-navbar primary>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>{{category.name}}</ion-title>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-searchbar (input)="searchProducts($event)"></ion-searchbar>
  <ion-list [virtualScroll]="products">
    <ion-list-header>
      Products
    </ion-list-header>
    <a ion-item *virtualItem="let prod" (click)="openProduct(prod)">
        <ion-thumbnail item-left>
          <img [src]="prod.images[0]" *ngIf="prod.images[0]">
        </ion-thumbnail>
        <h2>{{prod.name}}</h2>
        <p [innerHTML]="prod.description"></p>
    </a>
  </ion-list>
</ion-content>
```

First we created a search bar with which to search the list, and then we listed our product list.

This search feature is totally client-side, saving our valuable API requests.

Defining ProductPage

`ProductPage` shows our product and allows us to add it to the cart. The following code should be present in `product.ts`:

```
// /app/pages/product/product.ts
import {Component} from '@angular/core';
import {Modal, Loading, NavController, NavParams} from 'ionic-angular';
import {MarketProvider} from '../../providers/market-provider/market-
provider';
import {CartProvider} from '../../providers/cart-provider/cart-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import {ImageModal} from '../image-modal/image-modal';
import {CartPage} from '../cart/cart';

@Component({
  templateUrl: 'build/pages/product/product.html',
})
export class ProductPage {
  quantity:number;
  product:any;
 constructor(public nav: NavController,
             public params: NavParams,
             public market: MarketProvider,
             public cartProvider:CartProvider,
             public util:UtilProvider) {
    this.quantity = 1;
    // This is the Actual Product Content
    this.product = this.params.get('product');
  }
  addToCart(productId) {
    let loading = this.util.presentLoading("Adding to Cart");
    this.nav.present(loading);
    this.cartProvider.addToCart(productId, this.quantity)
    .then((data)=>{
      loading.dismiss();
      console.log(data);
    });
  }
  openImages() {
    let modal = Modal.create(ImageModal,{images: this.product.images});
    this.nav.present(modal);
  }
  openCart() {
    this.nav.push(CartPage);
  }
}
```

The `addToCart()` function adds the product to the cart with the given quantity. The `openImages()` function creates a `ImageModal` and opens it with our product images. The `openCart()` function opens the cart.

The `openImages()` function shows images of the product using the Ionic modal.

The `openCart()` function opens the `CartPage`.

Template

The following code should be present in `product.html`:

```
<!-- Template /app/pages/product/product.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>{{product.name}}</ion-title>
    <ion-buttons start>
      <button menuToggle><ion-icon name="menu"></ion-icon></button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content class="product">
   <ion-item text-wrap>
      <ion-thumbnail item-left>
          <img [src]="product.images[0]" *ngIf="product.images[0]">
          <img src="images/default_category.jpg"
            *ngIf="!product.images[0]">
      </ion-thumbnail>
       <h2>{{product.name}}</h2>
       <p [innerHTML]="product.description"></p>
    </ion-item>
   <ion-item>
    <ion-icon name="pricetag" primary item-left></ion-icon>
    <h3 primary> Price ${{product.price_discount}}</h3>
    <ion-note item-right>
      <span>{{product.stock_status.toUpperCase()}}</span>
    </ion-note>
   </ion-item>
   <ion-item>
    <ion-note item-left>Quantity</ion-note>
    <ion-input type="number" [(ngModel)]="quantity"></ion-input>
    <ion-note item-right>
      <button full primary (click)="addToCart(product.id)">
        <ion-icon name="cart"></ion-icon>
        Add to Cart
      </button>
    </ion-note>
   </ion-item>
   <ion-item><button full primary (click)="openImages()">See
Images</button></ion-item>
</ion-content>

<ion-footer>
  <ion-toolbar>
      <button primary full clear (click)="openCart()">
```

```
        <ion-icon name="cart"></ion-icon> Go to Cart
      </button>
    </ion-toolbar>
  </ion-footer>
```

In the template, we have shown the product's first image, name, and description. The **Add to Cart** button adds items to the cart with the given quantity. The **See images** button takes you to `ImageModal`, and the **Go to Cart** button takes you to the cart.

Defining ImageModal

`ImageModal` shows an image modal of our product. The following code is present in `image-model.ts`:

```
// /app/pages/image-modal/image-model.ts
import {Component} from '@angular/core';
import { Modal,NavParams, ViewController,NavController} from 'ionic-
angular';

@Component({
  templateUrl: 'build/pages/image-modal/image-modal.html',
})
export class ImageModal {
  images:any;
  constructor(public viewCtrl: ViewController, public params:
   NavParams) {
   this.images = this.params.get('images');
  }
  close() {
    console.log("Closing Modal");
    this.viewCtrl.dismiss();
  }
}
```

Template

The following code should be present in `image-modal.html`:

```
<!-- Template /app/pages/image-modal/image-modal.html -->
<ion-content padding class="image-modal">
  <button fab fab-right fab-bottom (click)="close()"><ion-icon
   name="close" ></ion-icon></button>
  <ion-slides>
     <ion-slide *ngFor="let image of images">
       <img [src]="image">
```

```
      </ion-slide>
   </ion-slides>
 </ion-content>
```

StyleSheet

The following code should be present in `image-modal.scss`:

```
/* Stylesheet /app/pages/image-modal/image-modal.scss */
.image-modal {
    ion-slides {
        height:75%;
    }
}
```

The `ImageModal` page shows images of the product using `ion-slides` and a close button to define the modal.

Defining CartPage

`CartPage` does all cart- and checkout-related work. The following code should be present in `cart.ts`:

```
// /app/pages/cart/cart.ts
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {AuthPage} from '../auth/auth';
import {OrderPage} from '../order/order';
import {CartProvider} from '../../providers/cart-provider/cart-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {StorageProvider} from '../../providers/storage-provider/storage-
provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';

@Component({
  templateUrl: 'build/pages/cart/cart.html',
})
export class CartPage {
  cart:any;
  isExist:Boolean;
  items:Array<any>;
  total:number = null;
  constructor(public cartProvider:CartProvider,
              public userProvider: UserProvider,
              public  nav: NavController,
              public storage: StorageProvider,
              public util:UtilProvider) {
  }

  ionViewWillEnter() {
    this.isExist = this.cartProvider.isCartExist();
    if(this.isExist) {
       this.cartProvider.getCartContents()
      .then((cartContent:any) => {
        this.cart = cartContent;
        this.items = cartContent.items;
        this.total = this.getTotal(this.items);
      }).catch(error => {
        this.total = this.getTotal();
      });
    } else {
      this.reset();
    }
  }
  changeQuantity(index, quantity) {
    this.items[index].quantity = quantity;
    this.cartProvider.updateCart(this.items)
    .then((cart:any) => {
      this.total = this.getTotal(cart.items);
    });
  }
  removeItem(id) {
    this.cartProvider.removeItem(id)
    .then((cart:any) => {
      this.cart = cart;
      this.items = cart.items;
      this.total = this.getTotal(this.items);
    })
    .catch(() =>{
      this.total = this.getTotal();
    });
  }
  checkout() {
   if(this.userProvider.isLoggedIn()) {
     this.nav.push(OrderPage, {items: this.items});
   } else {
     this.nav.push(AuthPage);
   }
  }
  getTotal(items?) {
    if(items) {
      let total = items.map((x) => {
        return x.price_discount * x.quantity;
```

```
        })
        .reduce((pre, curr) =>{
           return pre + curr;
       });
       return total;
      } else {
        return null;
      }
    }
    reset() {
      this.cart = null;
      this.items = null;
      this.total = null;
    }
  }
```

In the constructor, we get the contents of the cart, if it exists, and show all items in the page. The `changeQuantity` function changes the quantity of the selected item. The `removeItem` function removes items from the cart. The `checkout` function takes the user to `AuthPage`, if not logged in; otherwise, it takes the user to `OrderPage`.

Template

The following code should be present in `cart.html`:

```
<!-- Template /app/pages/cart/cart.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Cart</ion-title>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
  </ion-navbar>
</ion-header>

<ion-content class="cart">
  <ion-list>
    <ion-card *ngFor="let item of items; let i = index">
      <ion-item text-wrap>
        <ion-thumbnail item-left>
          <img [src]="item.images[0]" *ngIf="item.images[0]">
        </ion-thumbnail>
        <h2>{{item.name}}</h2>
        <p [innerHTML]="item.description"></p>
      </ion-item>
      <ion-item>
```

```
          <ion-note item-left>Quantity</ion-note>
          <ion-input type="number" [value]="item.quantity" #quantity>
           </ion-input>
          <ion-note item-right><button full primary
           (click)="changeQuantity(i, quantity.value)">Change</button>
           </ion-note>
          <ion-note item-right><button full secondary
           (click)="removeItem(item.id)">Remove</button></ion-note>
        </ion-item>
        <ion-item>
          <ion-note item-left>Sub Total</ion-note>
          <ion-note item-left primary>${{item.price_discount}} x
           {{item.quantity}} = ${{item.price_discount * item.quantity}}
           </ion-note>
        </ion-item>
      </ion-card>
    </ion-list>
  </ion-content>
  <ion-footer *ngIf="total">
    <ion-toolbar>
      <ion-title>Total is ${{total}}</ion-title>
    </ion-toolbar>
    <ion-toolbar>
      <button full primary clear (click)="checkout()">Checkout</button>
    </ion-toolbar>
  </ion-footer>
```

In the template, we have the product's image, name, and description, a **Change** button for changing the quantit,y and a **Remove** button for removing items from the cart. We are also showing the total cost of all items. In the bottom, the **Checkout** button is used for checking out.

Defining AuthPage

`AuthPage` is for login and registering the user. The following code should be present in `auth.ts`:

```
// /app/pages/auth/auth.ts
import {Component} from '@angular/core';
import {NavController, Events} from 'ionic-angular';
import {Validators, FormBuilder} from '@angular/common';
import {validateEmail} from '../../validators/email';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import {StorageProvider} from '../../providers/storage-provider/storage-
provider';
```

```
@Component({
  templateUrl: 'build/pages/auth/auth.html',
})
export class AuthPage {
  auth:string;
  loginForm:any;
  regForm:any;
  constructor(public nav:NavController,
              public form:FormBuilder,
              public util:UtilProvider,
              public userProvider:UserProvider,
              public storage: StorageProvider,
              public events:Events) {
    this.auth = 'login';
    this.form = form;
    this.loginForm = form.group({
        email: ["",Validators.compose([Validators.required,
         validateEmail])],
        password:["",Validators.required]
    });
    this.regForm = form.group({
        name: ["", Validators.required],
        email: ["",Validators.compose([Validators.required,
         validateEmail])],
        password:["", Validators.compose([Validators.required,
         Validators.minLength(8)])]
    });
  }
  login(userData?) {
    // If User logins explicitly
    if(!userData) {
      userData = this.loginForm.value;
    }

    this.userProvider.authUser(userData)
    .then((user)=> {
      this.storage.setObject('user', user);
      this.events.publish('user:logged_in');
      this.nav.pop();
    })
    .catch((error) => {
      let message = error.message;
      let toast = this.util.getToast(message);
      this.nav.present(toast);
    });
  }
  register() {
    let user = this.regForm.value;
```

```
      this.userProvider.createUser(user)
      .then(() => {
        delete user.name;
        this.login(user);
      });
    }
  }
```

The `login()` function authenticates the user. It stores the user's information in `localStorage` at the user key. It also emits an Ionic event, named `user:logged_in`.

The `register()` function registers the user and then logs the user in using the login function.

Template

The following code should be present in `auth.html`:

```
<!--  Template /app/pages/auth/auth.html -->
<ion-content padding class="auth">
  <div padding>
  <ion-segment [(ngModel)]="auth" primary>
    <ion-segment-button value="login">
      Login
    </ion-segment-button>
    <ion-segment-button value="register">
      Register
    </ion-segment-button>
  </ion-segment>
</div>

<div [ngSwitch]="auth">
  <div *ngSwitchWhen="'login'">
    <form [ngFormModel]="loginForm">
    <ion-list>
        <ion-item>
          <ion-label floating>Email</ion-label>
          <ion-input type="text" ngControl="email"></ion-input>
        </ion-item>

        <ion-item>
          <ion-label floating>Password</ion-label>
          <ion-input type="password" ngControl="password"></ion-input>
        </ion-item>
      </ion-list>
```

```
        <div padding>
          <button block (click)="login()"
           [disabled]="!loginForm.valid">Sign In</button>
        </div>
      </form>
    </div>

    <div *ngSwitchWhen="'register'">
      <form [ngFormModel]="regForm">
        <ion-list>
          <ion-item>
            <ion-label floating>Name</ion-label>
            <ion-input type="text" ngControl="name"></ion-input>
          </ion-item>
          <ion-item>
            <ion-label floating>Email</ion-label>
            <ion-input type="text" ngControl="email"></ion-input>
          </ion-item>

          <ion-item>
            <ion-label floating>Password</ion-label>
            <ion-input type="password" ngControl="password"></ion-input>
          </ion-item>
        </ion-list>

        <div padding>
          <button block (click)="register()"
           [disabled]="!regForm.valid">Register</button>
        </div>
      </form>
    </div>
  </div>
  </ion-content>
```

We have used `ion-segment` to show two forms: the login form for the user's login, and the register form for registration.

We are using the custom e-mail validator explained in the first chapter.

Defining OrderPage

`OrderPage` processes our order. The following code is present in `order.ts`:

```
// /app/pages/order/order.ts
import {Page, NavController, NavParams} from 'ionic-angular';
import {Validators, FormBuilder} from '@angular/common';
import {validateEmail} from '../../validators/email';
import {CartProvider} from '../../providers/cart-provider/cart-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';

@Page({
  templateUrl: 'build/pages/order/order.html',
})
export class OrderPage {
  address = {};
  items:any;
  addressForm:any;
  constructor(public nav:NavController,
              public form: FormBuilder,
              public cartProvider: CartProvider,
              public params: NavParams,
              public userProvider: UserProvider,
              public util: UtilProvider) {
    this.items = params.get('items');
    this.userProvider.getAddress()
    .then((address) => {
      this.address = address[0];
    }).catch((data)=> {
      console.log("No Address Added for User", data);
    });
    this.addressForm = form.group({
      full_name: ["", Validators.required],
      email: ["",Validators.compose([Validators.required,
       validateEmail])],
      country:["", Validators.required],
      state: ["", Validators.required],
      city: ["", Validators.required],
      address1:["", Validators.required],
      postal_code:["", Validators.required]
    });
  }
  processOrder() {
    let promise;
    let order_id;
    let nonce;
    if(this.address) {
      Object.keys(this.address).forEach(key => {
        if(this.address[key] == null) {
          delete this.address[key];
        }
      });
      promise = this.cartProvider.createOrder(this.items,
       this.address);
    } else {
      promise = this.userProvider.createAddress(this.addressForm.value)
      .then((address) => {
        return this.cartProvider.createOrder(this.items, address);
      });
    }
    promise.then((order) => {
      order_id = order.id;
      this.cartProvider.setCartID(null);
      return this.cartProvider.getPayment(order.total,
       this.addressForm.value.full_name);
    })
    .then((data) => {
      nonce = data.nonce;
      this.cartProvider.createPayment(order_id, nonce)
      .then(data => {
        let toast = this.util.getToast('Order is successfull');
        this.nav.present(toast);
        this.address = {};
        this.nav.popToRoot();
      });
    });
  }
}
```

In the constructor, we get the address of the logged-in user and also create an address form. The `processOrder` function creates an order using the `createOrder` member of `CartProvider` and, if it is successful, then we charge the user using the `getPayment` member of `CartProvider`.

Template

The following code should be present in `order.html`:

```
<!-- Template /app/pages/order/order.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Order</ion-title>
  </ion-navbar>
```

```
</ion-header>

<ion-content padding class="order">
 <form [ngFormModel]="addressForm">
  <ion-list>
     <ion-item>
        <ion-label>Full Name</ion-label>
        <ion-input type="text" ngControl="full_name"
         [(ngModel)]="address.full_name"></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>Email</ion-label>
        <ion-input type="text" ngControl="email"
         [(ngModel)]="address.email"></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>Country</ion-label>
        <ion-input type="text" ngControl="country"
         [(ngModel)]="address.country"></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>State</ion-label>
        <ion-input type="text" ngControl="state"
         [(ngModel)]="address.state"></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>City</ion-label>
        <ion-input type="text" ngControl="city"
         [(ngModel)]="address.city" ></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>Address</ion-label>
        <ion-input type="text" ngControl="address1"
         [(ngModel)]="address.address1"></ion-input>
     </ion-item>
     <ion-item>
        <ion-label>Postal Code</ion-label>
        <ion-input type="text" ngControl="postal_code"
         [(ngModel)]="address.postal_code"></ion-input>
     </ion-item>
  </ion-list>
  <div>
    <button full primary (click)="processOrder()"
     [disabled]="!addressForm.valid">Order</button>
  </div>
 </form>
</ion-content>
```

We show the address form. If the logged-in user has an address saved in `marketcloud`, then we automatically fill the form in. When the user clicks on the **Order** button, it opens a dialog to input credit/debit card details to make payment.

Adding style

Ionic 2 uses **Syntactically Awesome Style Sheets** (**SASS**) to style applications. So, it's very easy to change the overall look and feel of an application just by changing some values. We will change the values of some default variables to make our application look unique.

Styles are defined in the `app/theme` folder. Inside this folder, there is an `app.variable.scss` file. In this file, Ionic has defined the default colors. Let's change the color values of primary and secondary Ionic colors to this:

```
primary:    #009688,
secondary:  #212121,
```

We also need to add our `ImageModal` modal's stylesheet in `/app/theme/app.core.scss` as follows:

```
@import "../pages/image-modal/image-modal";
```

Running our app

Let's run our application on an actual device.

Adding a platform and running the app on our device

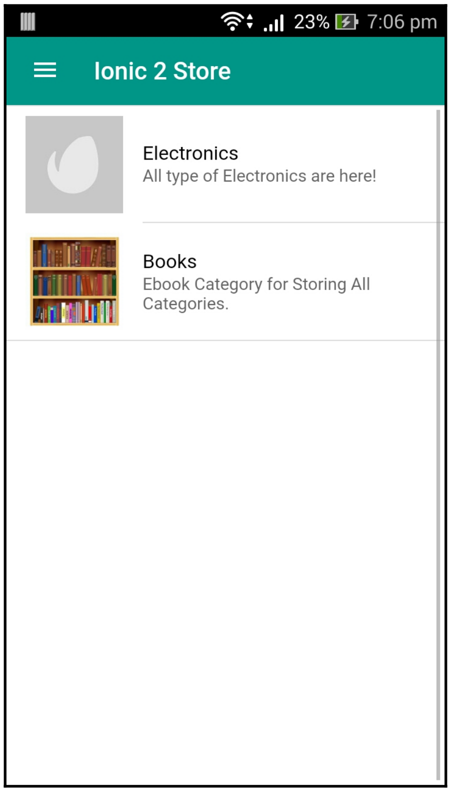
The following adds a platform and runs the app on the device:

```
ionic platform add android
ionic run android
```

App screenshots

These are actual screenshots of our e-commerce application.

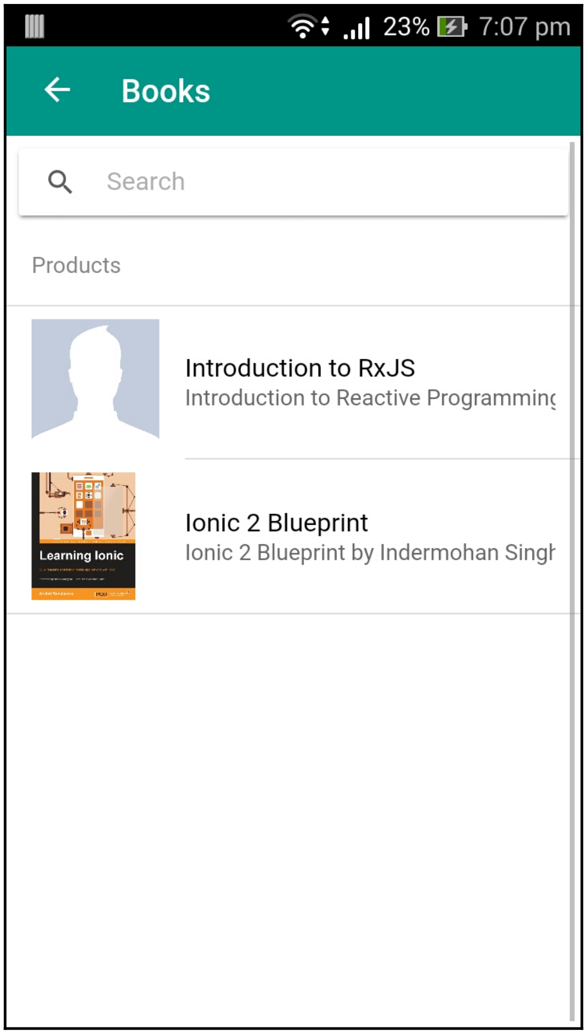
The following screenshot shows `StorePage` with a list of categories:



Electronics
All type of Electronics are here!

Books
Ebook Category for Storing All Categories.

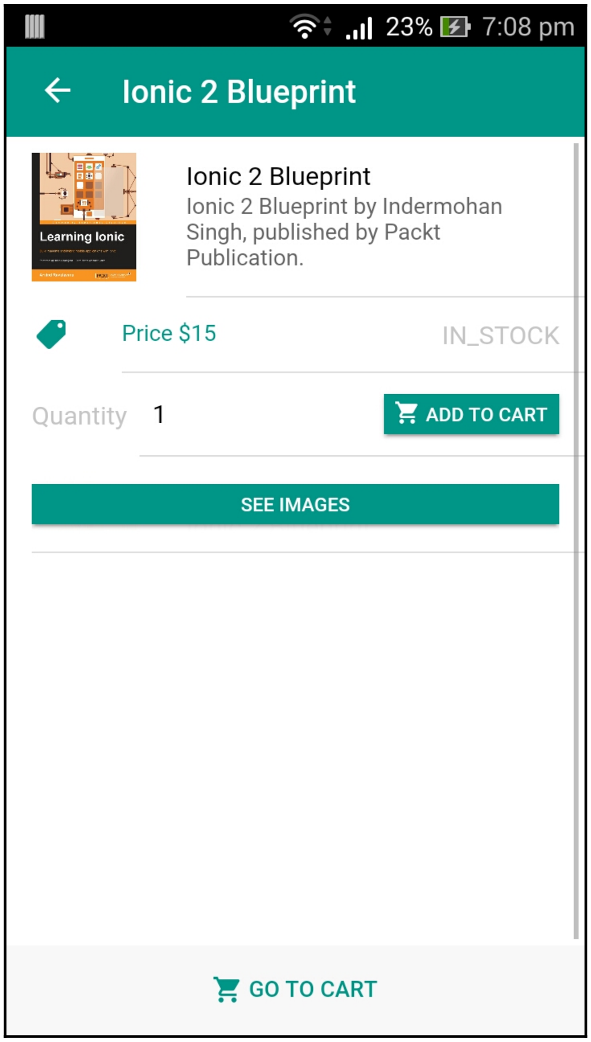
The following screenshot shows the `Category` page with a list of products:



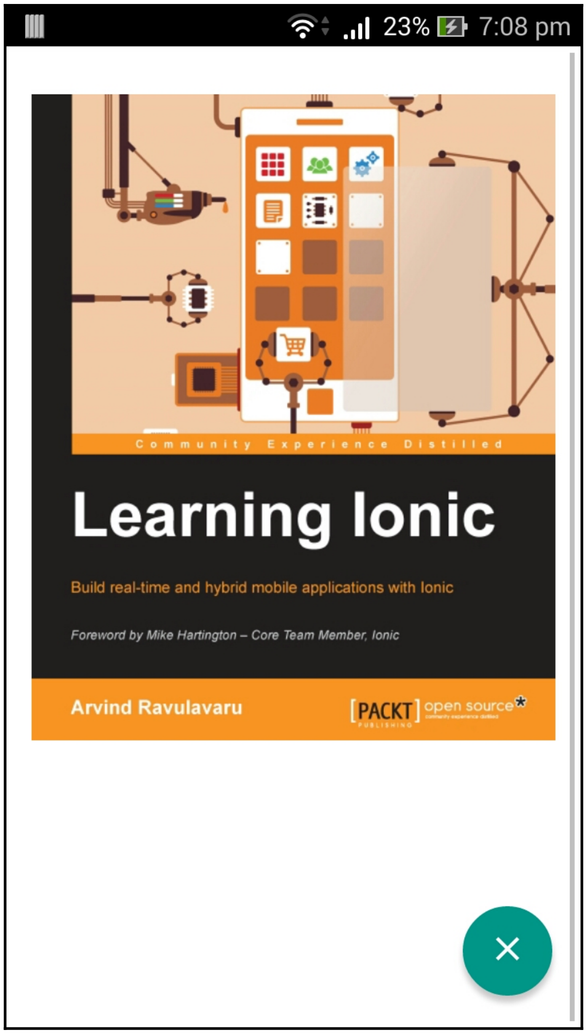
Introduction to RxJS
Introduction to Reactive Programminç

Ionic 2 Blueprint
Ionic 2 Blueprint by Indermohan Singh

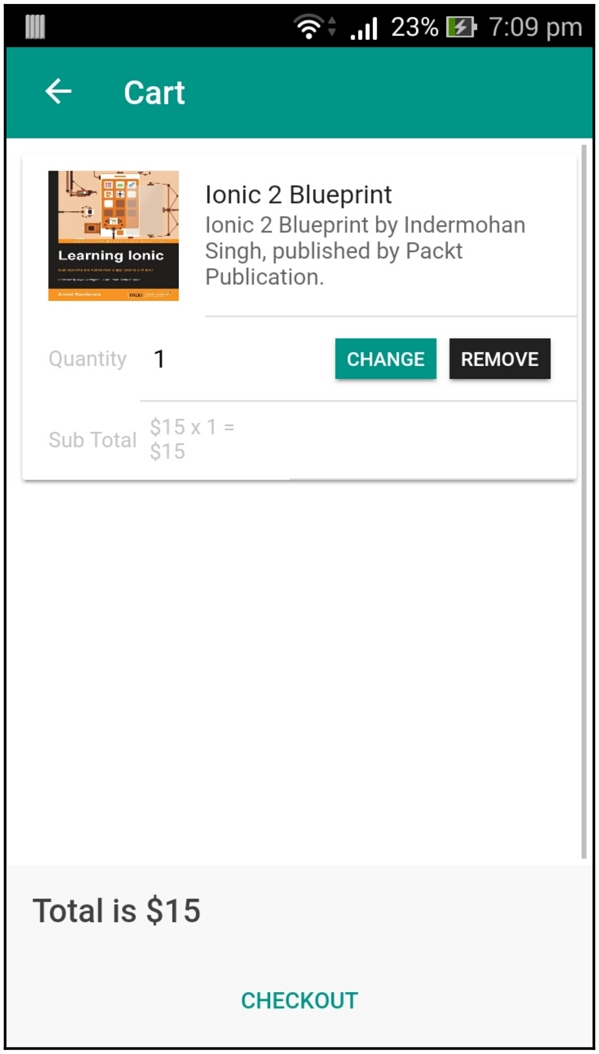
The following screenshot shows `ProductPage`:



The following screenshot shows `ImageModal` with images of our product:


23% 7:08 pm
Community Experience Distilled
Learning Ionic
Build real-time and hybrid mobile applications with Ionic
Foreword by Mike Hartington – Core Team Member, Ionic
Arvind Ravulavaru
[PACKT] open source

The following screenshot shows `CartPage` with items in the cart:



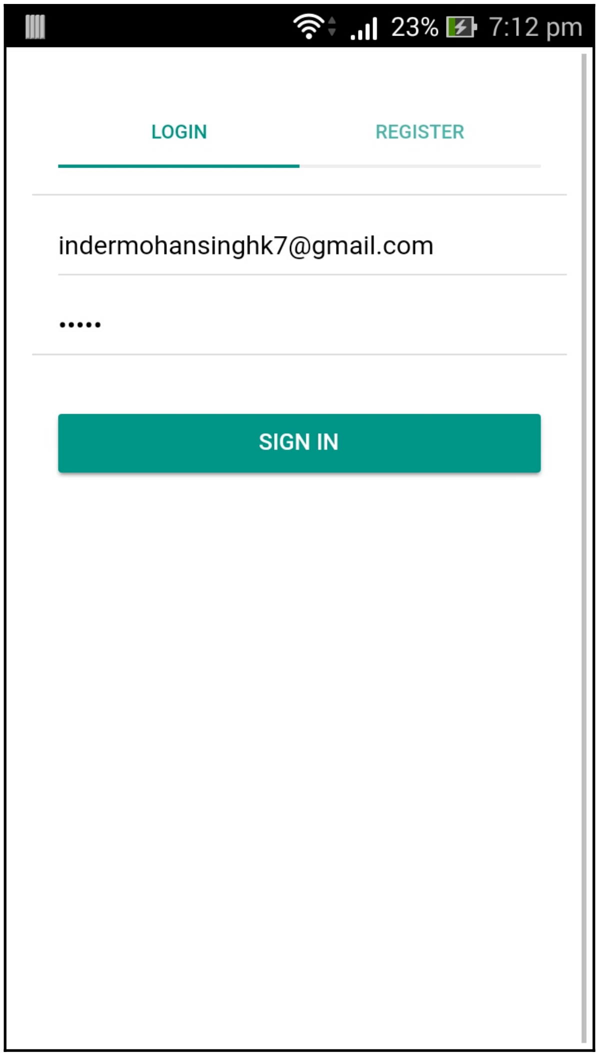
Ionic 2 Blueprint
Ionic 2 Blueprint by Indermohan Singh, published by Packt Publication.

Quantity 1

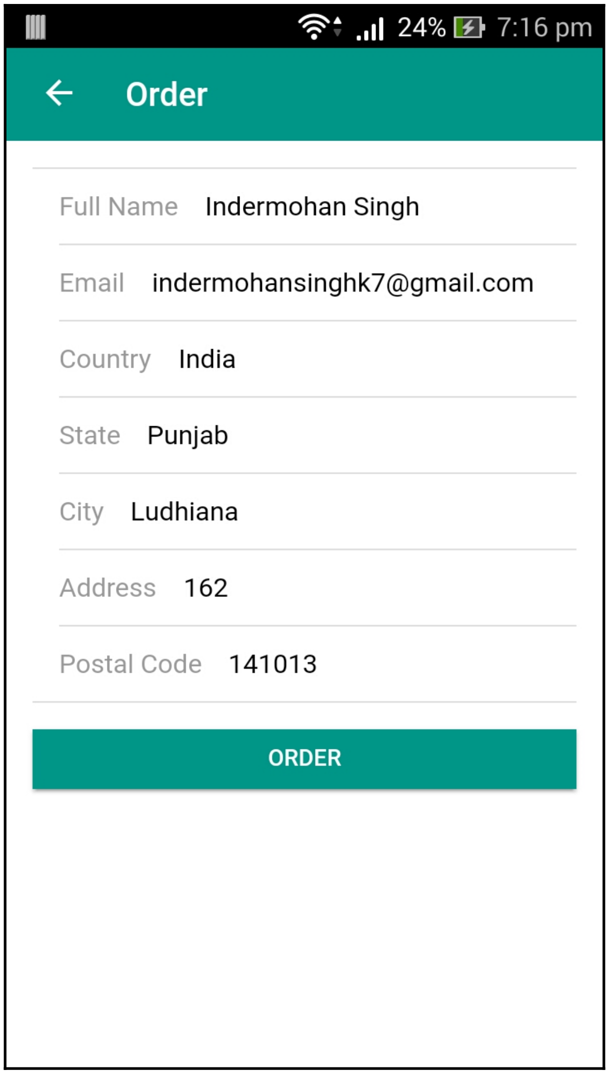
Sub Total $15 x 1 = $15

Total is $15

The following screenshot shows AuthPage with a login and register segment:



The following screenshot shows `OrderPage`, where the user orders and pays for items:



Full Name	Indermohan Singh
Email	indermohansinghk7@gmail.com
Country	India
State	Punjab
City	Ludhiana
Address	162
Postal Code	141013

Summary

In this chapter, we have created a complete e-commerce application using Ionic 2 and Marketcloud. We have used various Ionic features such as menus, segments, loading, alerts, and lots of other features. We used the Marketcloud JS SDK and dashboard to create our products and categories, and we used Braintree to get payments from the user. More interestingly, we created an e-commerce application without using any kind of custom backend.

3
Conference App

In the previous chapter, we created an e-commerce application by using Ionic 2 and MarketCloud. In this chapter, we will create a conference app. We will create an app which will provide a list of speakers, a schedule, directions to the venue, ticket booking, and lots of other features.

We will learn the following things:

- Using the device's native features
- Leveraging **localStorage**
- Ionic menu and tabs
- Using RxJS to build a perfect search filter

Introduction

A conference app is a companion application for conference attendees. In this application, we will use a **Lanyrd** JSON Exportand hardcoded JSON file as our backend. We will have tabs and a side menu interface, just like our e-commerce application.

When a user opens our app, the app will show a tab interface with the `SpeakersPage` open. It will have `SchedulePage` for conference schedule and `AboutPage` for information about conference.

We will also make this app work offline, without any Internet connection. So, your user will still be able to view speakers, see the schedule, and do other stuff without using the Internet at all.

JSON data

In the application, we have used a hardcoded JSON file as our Database. But in the truest sense, we are actually using a JSON export of a Lanyrd event. I tried to design this chapter using Lanyrd as the backend, but unfortunately Lanyrd is mostly in maintenance mode. So I was not able to use it. In this chapter, I am still using a JSON export from Lanyrd, from a previous event. So, if you are able to get a JSON export for your event, you can just swap the URL and you are good to go.

Those who don't want to use Lanyrd, and instead want to use their own backend, must have a look at the next section. I have described the structure of JSON, which I have used to make this app. You can create your REST API accordingly.

Understanding JSON

Let's understand the structure of the JSON export.

- The whole JSON database is an object with two keys, `timezone` and `sessions`, like the following:

```
{
  timezone: "Australia/Brisbane",
  sessions: [..]
}
```

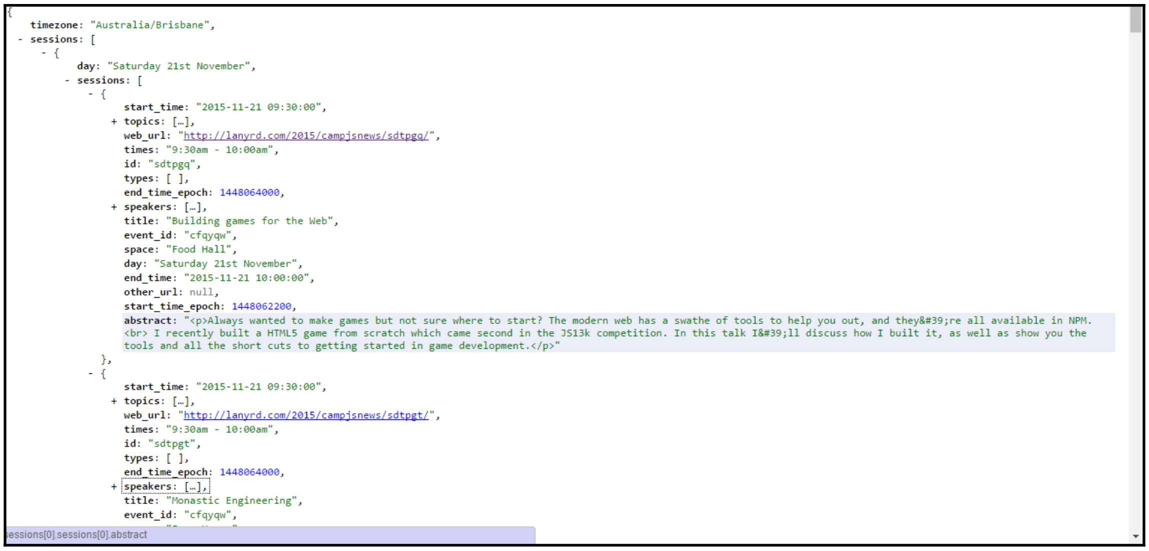
- `Timezone` is just a string, but the `sessions` key is an array of lists of all the sessions of our conference. Items in the sessions array are divided according to days at the conference. Each item represents a conference day and has the following structure:

```
{
  day: "Saturday 21st November",
  sessions: [..]
}
```

- So, the sessions array of each day has actual sessions as items. Each item has the following structure:

```
{
  start_time: "2015-11-21 09:30:00",
  topics: [],
  web_url: "url of event
  times: "9:30am - 10:00am",
  id: "sdtpgq",
  types: [ ],
  end_time_epoch: 1448064000,
  speakers: [],
  title: "Talk Title",
  event_id: "event_id",
  space: "Space",
  day: "Saturday 21st November",
  end_time: "2015-11-21 10:00:00",
  other_url: null,
  start_time_epoch: 1448062200,
  abstract: "<p>Abstract of Talk</p>"
},
```

Here, the `speakers` array has a list of all speakers. We will use that `speakers` array to create a list of all speakers in an array. You can see the whole structure here:



```
{
   timezone: "Australia/Brisbane",
 - sessions: [
    - {
         day: "Saturday 21st November",
       - sessions: [
          - {
               start_time: "2015-11-21 09:30:00",
             + topics: […],
               web_url: "http://lanyrd.com/2015/campjsnews/sdtpgq/",
               times: "9:30am - 10:00am",
               id: "sdtpgq",
               types: [ ],
               end_time_epoch: 1448064000,
             + speakers: […],
               title: "Building games for the Web",
               event_id: "cfqyqw",
               space: "Food Hall",
               day: "Saturday 21st November",
               end_time: "2015-11-21 10:00:00",
               other_url: null,
               start_time_epoch: 1448062200,
               abstract: "<p>Always wanted to make games but not sure where to start? The modern web has a swathe of tools to help you out, and they&#39;re all available in NPM.
               <br> I recently built a HTML5 game from scratch which came second in the JS13k competition. In this talk I&#39;ll discuss how I built it, as well as show you the
               tools and all the short cuts to getting started in game development.</p>"
            },
          - {
               start_time: "2015-11-21 09:30:00",
             + topics: […],
               web_url: "http://lanyrd.com/2015/campjsnews/sdtpgt/",
               times: "9:30am - 10:00am",
               id: "sdtpgt",
               types: [ ],
               end_time_epoch: 1448064000,
             + speakers: […],
               title: "Monastic Engineering",
               event_id: "cfqyqw",
```

essions[0].sessions[0].abstract

That's all we need to understand about JSON.

Defining the app

In this section, we will define various functionalities of our application. We will also show the architecture of our app using an app flow diagram.

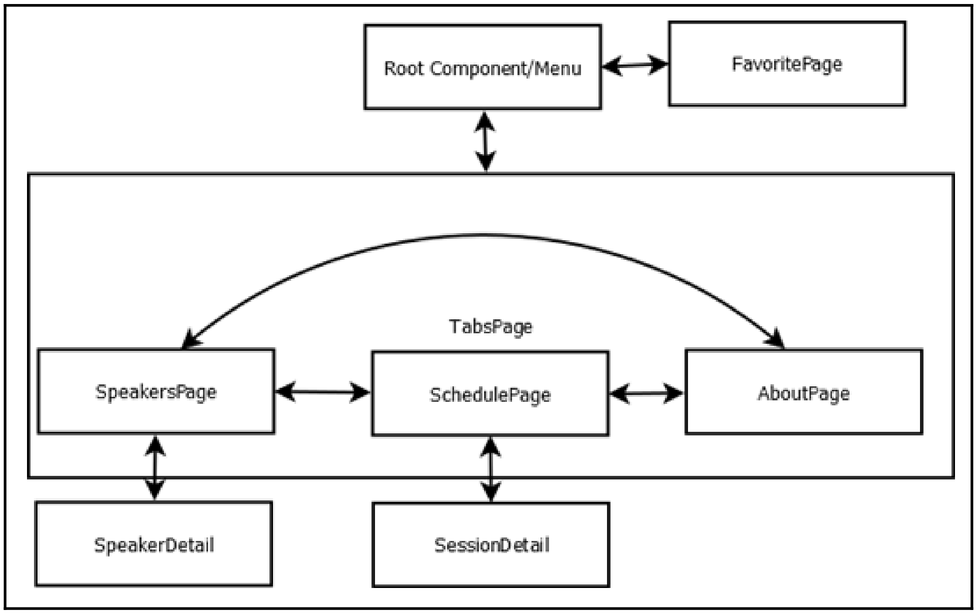
Functionalities

We will be including the following functionalities in our application:

- List of speakers
- Schedule detail
- Search functionality using session title, abstract, and speaker's names
- Hide/Show any day of the schedule
- Favorite list for sessions
- Adding favorite sessions to the device calendar
- Ability to share sessions to other applications
- Directions to venue
- Offline working

App flow

This is how the control will flow inside our application:



Let's understand the flow:

- **RootComponent**: This is the root Ionic component. It is defined inside the `/app/app.ts` file.
- **TabsPage**: This acts as a container for our `SpeakersPage`, `SchedulePage`, and `AboutPage`.
- **SpeakersPage**: This shows a list of all the speakers at our conference.
- **SchedulePage**: This shows us our conference schedule and allows us various filter features.
- **AboutPage**: This provides us with information about the conference.
- **SpeakersDetail**: This shows the details of the speaker and a list of his/her presentations in this conference.
- **SessionDetail**: This shows the details of a session with the title and abstract of the session.
- **FavoritePage**: This shows a list of the user's favorite sessions.

Scaffolding and setting up the app

We will start by scaffolding a tabs application. Run the following command:

```
ionic start conference-app tabs --v2 --ts
```

Notice the `--v2` and `--ts` tag for scaffolding the Ionic 2 app based on **TypeScript**.

Using the `cd` command, go to the `conference-app` folder and run the `ionic serve` command as follows:

```
ionic serve
```

To check how the tabs app looks before we start writing code for it, we need to install some project-specific dependencies.

Installing dependencies

The following are some project-specific dependencies that we need to install.

Use the following to install**Cordova** plugins:

```
ionic plugin add cordova-plugin-x-socialsharing
ionic plugin add cordova-plugiin-x-toast
ionic plugin add cordova-plugin-calender
ionic plugin add cordova-plugin-inappbrowser
ionic plugin add uk.co.workingedge.phonegap.plugin.launchnavigator
ionic plugin add cordova-plugin-network-information
```

Coding our app

Now we have everything set up to start working on our application, let's define the following:

- Structure of the tabs starter
- Our main `app.ts` file
- `ConfProvider`
- Ionic pages for various views

We will not change `index.html`. It already has everything required for this project.

Structure of tabs starter

When we use tabs starter, it creates four pages for us. One is `TabsPage` and the three others are children of `TabsPage`, named `page1`, `page2`, and `page3`.

We will refactor these pages as follows:

- `page1` will be `SpeakersPage`
- `page2` will be `SchedulePage`
- `page3` will be `AboutPage`

We have also changed their file names accordingly.

Defining app.ts

This is the root of our application and it defines the root component using the `@App` decorator. This is where we inject all of our dependencies:

```
/* /app/app.ts */
import {Component, ViewChild} from '@angular/core';
import {Platform, Nav, ionicBootstrap} from 'ionic-angular';
import {JSONP_PROVIDERS, Jsonp} from '@angular/http';
import {StatusBar, InAppBrowser} from 'ionic-native';
import {ConfProvider} from './providers/conf-provider';
import {TabsPage} from './pages/tabs/tabs';
import {FavoritePage} from './pages/favorite/favorite';

@Component({
  templateUrl: 'build/app.html'
})
export class MyApp {
  rootPage: any = TabsPage;
  pages:any;
  @ViewChild(Nav) nav: Nav;
  constructor(platform: Platform) {
    platform.ready().then(() => {
      StatusBar.styleDefault();
    });
    this.pages = [
      {page:TabsPage, title: 'Home'},
      {page:FavoritePage, title: 'Favorite'}
    ];
  }
  openPage(page) {
    this.nav.push(page);
  }
  openLink(url,target, options="location=no") {
    InAppBrowser.open(url,target,options)
  }
}

ionicBootstrap(MyApp, [ConfProvider, JSONP_PROVIDERS]);
```

Each function does its work as follows:

- The `constructor()` function initializes `this.pages`, which is a list of pages for the side menu. Also, it fires a callback, when the platform is ready. In the callback, we used the `StatusBar` plugin.

- The `openPage(page)` function pushes the respective page to the navigation stack.
- The `openLink(url, target, option)` function opens the given URL in **InAppBrowser**.
- `ionicBoostrap` starts our application.

The following code should be present in `app.html`:

```
<!--  Template app/app.html -->
<ion-menu [content]="content">
    <ion-toolbar primary>
        <ion-title>Menu</ion-title>
    </ion-toolbar>
    <ion-content>
        <ion-list>
            <ion-item menuClose *ngFor="let page of pages; let i =
             index" (click)="openPage(page.page)">{{page.title}}</ion-
             item>
            <ion-item menuClose (click)='openLink("https://inders.in",
             "_blank")'>Tickets</ion-item>
        </ion-list>
    </ion-content>
</ion-menu>

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

Again, nothing fancy. This structure is similar to our e-commerce application. We created an `ion-toolbar` and `ion-menu` and finally we used `ion-nav` so that we can use navigation.

Defining ConfProvider

Create a file named as `conf-provider.ts` in the `app/provider` folder.

ConfProvider is the only provider of our application. It provides us with the speakers list and schedule of our conference. The following code should be present in conf-provider.ts:

```
/* /app/providers/conf-provider.ts */
import {Injectable} from '@angular/core';
import {Http, Jsonp} from '@angular/http';
import {Storage, LocalStorage} from 'ionic-angular';

@Injectable()
export class ConfProvider {
    public speakers:any;
    public schedule:any;
    storage = new Storage(LocalStorage);
    url:string =
  'http://lanyrd.com/2015/campjsnews/schedule/481ea3897063c7d5.v1.json?
  callback=JSONP_CALLBACK';
    offline_url:string = 'data/data.json';
    request:any;
    data:any;
    constructor(public jsonp: Jsonp, public http:Http) {
      this.request =  this.jsonp.request(this.url);
    }
    load() {
      if(this.data) {
        return Promise.resolve(this.data);
      }
      let promise = new Promise((resolve) => {
          this.request.subscribe(data => {
            this.data = this.processData(data.json());
            this.storage.set('offline_data',
             JSON.stringify(this.data));
            resolve(this.data);
          },
          error => {
             this.storage.get('offline_data')
             .then(data => {
                if(data) {
                  this.data = JSON.parse(data);
                  resolve(this.data);
                } else {
                  this.http.get(this.offline_url).subscribe(data => {
                    this.data = this.processData(data.json());
                    this.storage.set('offline_data',
                     JSON.stringify(this.data));
                    resolve(this.data);
                  });
                }
              });
          });
      });
      return promise;
    }
    processData(data) {
      data.speakers = [];
      data.speakersName = [];
      data.sessions.forEach(day => {
        day.display = true;
        day.sessions.forEach(session => {
          session.speakers.forEach(newSpeaker => {
             let index = data.speakersName.indexOf(newSpeaker.name);
             if( index < 0) {
                 newSpeaker.sessions = [];
                 newSpeaker.sessions.push({title: session.title,
                  abstract:session.abstract});
                 data.speakersName.push(newSpeaker.name);
                 data.speakers.push(newSpeaker);
             } else {
                 data.speakers[index].sessions.push({title:
                  session.title, abstract:session.abstract});
             }
          });
        });
      })
      delete data.speakersName;
      return data;
    }
    getSpeakers() {
      return this.load().then(data => {
        let newSpeakerArray =
         this.addAlphabets(this.sort(data.speakers));
        return newSpeakerArray;
      });
    }
    getSchedule() {
      return this.load().then(data => {
        return data.sessions;
      });
    }
    sort(speakers) {
      let list = speakers;
      list.sort(function(a, b) {
        if(a.name > b.name) {
          return 1;
        }
        if(a.name < b.name) {
          return -1;
        }
        return 0;
      });
      return list;
    }

    addAlphabets(items) {
      let currentChar = "";
      let newArray = [];
      items.forEach(function(item, index) {
        var char = item.name[0].toLowerCase();
        if(currentChar != char) {
            newArray.push({type:"title", value: char.toUpperCase()});
            currentChar = char;
        }
        newArray.push(item);
      });
      return newArray;
    }
  }
```

Let's understand the preceding code:

- `constructor()` – This initializes `this.storage` to use `localStorage` and `this.request`, an `Observable` for the JSONP request to the Lanyrd JSON file.
- `load()` – Basically, this function returns the promise, which resolves to our conference schedule and speaker list. Our conference data is in the `this.data` member. It checks if `this.data` is initialized. If not, it calls the Lanyrd backend, and resolves with that data, and stores the new data to `localStorage`. If there is an error, it will check for fallback solutions, such as conference data stored in `localStorage` or a JSON file inside the app.
- `processData()` – This function basically create a list of speakers from the schedule and attaches that list of `this.data.speakers.`
- `getSpeakers()` – This function returns the list of speakers in the form of a promise.
- `getSchedule()` – This function returns the schedule of our conference in the form of a promise.
- `sort()` – This function sorts a list based on the name key of an item.
- `addAlphabets()` – This function adds an alphabet item in a sorted list. For example, if a list item has names, it adds *A* before names starting with *A* and *B* in front of names starting with *B*, and so on.

Application Pages

Now we have to define all the providers and services for our application. Let's define the pages of our application.

We will have the following pages in our application:

- `TabsPage`
- `SpeakersPage`
- `SchedulePage`
- `AboutPage`
- `SpeakersDetail`
- `ScheduleDetail`
- `FavoritePage`

Defining the TabsPage page

`TabsPage` is automatically created by `ionic-cli` since we have used the tabs starter.

So, we will just import our pages in `tabs.ts`:

```
import {SpeakersPage} from '../speakers/speakers';
import {SchedulePage} from '../schedule/schedule';
import {AboutPage} from '../about/about';
```

Then, we assign them to their respective member values in the constructor, as follows:

```
tab1Root: any = SpeakersPage;
tab2Root: any = SchedulePage;
tab3Root: any = AboutPage;
```

The following code should be present in `tabs.html`:

```
<!-- Template: /app/pages/tabs/tabs.html -->
<ion-tabs primary>
  <ion-tab [root]="tab1Root" tabTitle="Speakers" tabIcon="people"></ion-
tab>
  <ion-tab [root]="tab2Root" tabTitle="Schedule" tabIcon="calendar"></ion-
tab>
  <ion-tab [root]="tab3Root" tabTitle="About" tabIcon="briefcase"></ion-
tab>
</ion-tabs>
```

In tabs.html, we have just changed the tabTitle to Speakers, Schedule, About, and tabIcon to people, calendar, and briefcase, respectively, for the second, and third tabs, according to our requirements.

Defining the SpeakersPage page

SpeakersPages shows the list of speakers of our application. As we already know, a JSON database doesn't have a list of speakers. But, in ConfProvider, we processed our schedule and added a new speakers array in the JSON data. We will use that array in the following code:

```
// /app/pages/speakers/speakers.ts //
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {ConfProvider} from '../../providers/conf-provider';
import {SpeakerDetail} from '../speaker-detail/speaker-detail';
import {Network, Connection} from 'ionic-native';
import {Subscription} from 'rxjs/Rx';

@Component({
  templateUrl: 'build/pages/speakers/speakers.html',
})
export class SpeakersPage {
  offline:any;
  speakers:Array<any>;
  disconnectSubscription:Subscription;
  onlineSubscription:Subscription;
  constructor(public confProvider:ConfProvider, public nav:
   NavController) {
    confProvider.getSpeakers().then(speakers => {
      this.speakers = speakers;
    });
    this.disconnectSubscription = Network.onDisconnect().subscribe(()=> {
      this.offline = true;
    });
    this.onlineSubscription = Network.onConnect().subscribe(()=> {
      this.offline = false;
    });
  }
  speakerDetail(speaker) {
    this.nav.push(SpeakerDetail, {speaker: speaker});
  }
  isOnline () {
        let networkState = Network.connection;
        return networkState !== Connection.UNKNOWN && networkState !==
         Connection.NONE;
  }
  ionViewDidEnter() {
     this.offline = !this.isOnline();
  }
}
```

In `SpeakersPage`, we get a list of speakers from `ConfProvider`. We also created two `Observables` to get the network's online and offline status.

The `speakerDetail` function pushes the `SpeakerDetail` page to Navigation Stack. The `isOnline` function checks the connectivity of the network, when called. When we enter the page, the `ionViewDidEnter` function is called and we check if the user is online or not.

`ionViewDidEnter` is a page life cycle hook provided by Ionic on every page.

We check Internet connectivity, so we can provide a better experience to the user based on it.

The following code should be present in `speakers.html`:

```
<!-- Template /app/pages/speakers/speakers.html -->
<ion-header>
  <ion-navbar primary>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>
      Speakers
    </ion-title>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-list>
    <span text-wrap *ngFor="let speaker of speakers">
      <ion-item-divider light *ngIf="speaker.type">
        {{speaker.value}}
      </ion-item-divider>
      <a ion-item *ngIf="!speaker.type"
       (click)="speakerDetail(speaker)">
        <ion-avatar item-left *ngIf="!offline">
          <img [src]="speaker.image_75">
        </ion-avatar>
        <h2>{{speaker.name}}</h2>
        <h3>{{speaker.designation}}</h3>
      </a>
    </span>
  </ion-list>
</ion-content>
```

This is a very straightforward template: a navigation bar, and then a list of all speakers. On clicking on each speaker item, the `SpeakerDetail` page is opened.

Defining the SchedulePage page

`SchedulePage` shows us the schedule of our conference. It also provides a search bar for searching sessions by title, speaker name, and description.

The following code should be present in `schedule.tc`:

```
// /app/pages/schedule/schedule.ts //
import {Page, NavController, Storage, LocalStorage} from 'ionic- import
{Component} from '@angular/core';
import {NavController, Storage, LocalStorage} from 'ionic-angular';
import {Calendar, Toast, SocialSharing} from 'ionic-native';
import {Control} from '@angular/common';
import {ConfProvider} from '../../providers/conf-provider';
import {SessionDetail} from '../session-detail/session-detail';
import 'rxjs/Rx';

@Component({
  templateUrl: 'build/pages/schedule/schedule.html',
})
export class SchedulePage {
  filterControl: Control = new Control('');
  days:any;
  totalDays:Array<any>;
  storage = new Storage(LocalStorage);
  favorite:Array<any>;
  constructor(public nav:NavController, public confProvider:
   ConfProvider) {
     confProvider.getSchedule()
    .then(data => {
      this.days = data;
      this.totalDays = JSON.parse(JSON.stringify(data));
    });
    // Handling Favorite List
    this.storage.get('favorite')
    .then(data => {
        if(!data) {
          data = "[]";
        }
        this.favorite = JSON.parse(data);
    });
    // Observable for Searchbar
    this.filterControl.valueChanges
    .debounceTime(500)
    .distinctUntilChanged()
    .map(v => v.toLowerCase().trim())
    .subscribe(value => {
      this.search(value);
    });
  }
  openSession(session) {
      this.nav.push(SessionDetail, {session: session});
  }
  favoriteSession(session) {
    let favoriteList = this.favorite.map(fav => {
      return fav.title
    });

    if(favoriteList.indexOf(session.title) < 0) {
      this.favorite.push(session);
      this.storage.set('favorite', JSON.stringify(this.favorite));
      let start_date = new Date(session.start_time);
      let end_date = new Date(session.end_time);
      Calendar.createEvent(session.title, session.place,
       session.abstract, start_date, end_date)
      .then(data => {
        Toast.show("Session Added to Calender", "2000", "center")
        .subscribe(() => {
        });
      })
      .catch(error => {
        console.log(error);
      });
    } else {
        Toast.show("Already Added to Calender", "2000", "center")
        .subscribe(()=> {
        });
    }
  }
  shareSession(session) {
    let shareString = `Check this amazing talk, ${session.title} at
     Conference name on ${session.day}`;
    SocialSharing.share(shareString, session.title, null,
     session.web_url);
  }
```

```
      // Search Logic
      search(value) {
        this.days = JSON.parse(JSON.stringify(this.totalDays));
        this.days = this.days.filter(day => {
            day.sessions = day.sessions.filter(session => {
              let selected = false;
              if(session.space.toLowerCase().indexOf(value) >= 0) {
                selected = true;
              }
              if(session.title.toLowerCase().indexOf(value) >= 0) {
                selected = true;
              }
              if(session.abstract &&
               session.abstract.toLowerCase().indexOf(value) >= 0) {
                selected = true;
              }
              if(session.speakers) {
                 session.speakers.forEach(speaker => {
                  if(speaker.name.toLowerCase().indexOf(value) >=0) {
                    selected = true;
                    return;
                  }
                });
              }
              return selected;
            });
            if(day.sessions.length > 0) {
              return true;
            }
          });
      }
    }
```

In `SchedulePage`, we get Schedule from `ConfProvider`. In the constructor, we assign our conference schedule to two member variables; one is `this.days` and the other is `this.totalDays`.

In the constructor, we have also created an observable for the search filter. Angular 2's control `valueChanges` method returns an `Observable`, so we can use that `Observable` with any form control. In our case, we are using it on `ion-searchbar`. We are debouncing for 500 milliseconds, and calling our `search` function, if the current value is distinct from the previous value. We can use `map`, `filter`, and various other functions on asynchronous events. That is the power of RxJS.

The `openSession` function opens the `SessionDetail` page with details of the session. The `favoriteSession` function adds the respective session to a favorite list and saves that list to `localStorage`. At the same time, it also creates an entry in the device's calendar. The `shareSession` function shares the session using the `SocialSharing` plugin.

The `search` function does the actual searching of sessions. This function is called when we subscribe to your `ion-searchbar` member's `Observable`. We are basically filtering the sessions based on the title, abstract, and speaker name.

The following code should be present in `schedule.html`:

```
<!-- Template /app/pages/schedule/schedule.html -->
<ion-header>
  <ion-navbar primary>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
    </button>
    <ion-title>Schedule</ion-title>
  </ion-navbar>
</ion-header>

<ion-content class="schedule">
  <ion-searchbar [ngFormControl]="filterControl"></ion-searchbar>
  <ion-card *ngFor="let day of days">
    <ion-item>
      <ion-label>{{day.day}}</ion-label>
      <ion-toggle [(ngModel)]="day.display"></ion-toggle>
    </ion-item>

   <ion-list>
   <ion-item-sliding text-wrap *ngFor="let session of day.sessions"
    [hidden] = "!day.display" >

        <button ion-item (click)="openSession(session)">
          <h3>{{session.title}}</h3>
          <p>
            {{session.times}}
            {{session.space}}
          </p>
        </button>
        <ion-item-options>
          <button danger (click)="favoriteSession(session)">
            <ion-icon name="heart"></ion-icon>
            Favorite
          </button>
          <button (click)="shareSession(session)">
            <ion-icon name="share"></ion-icon>
```

```
                Share
              </button>
            </ion-item-options>
       </ion-item-sliding>
       </ion-list>

      </ion-card>
    </ion-content>
```

We have an `ion-seachbar` on the top, which allow us to filter the schedule. Then we have `ion-toggle`, which allows us to toggle the display for each day. Then we have a list of sessions. Each session has the title, time, and space. When you swipe to the left on a session, you will get favorite and share buttons.

Defining the SessionDetail page

The `SessionDetail` page shows the details of the session, with title, abstract, and the name of speaker. The following code should be present in `session-detail.ts`:

```
// /app/pages/session-detail/session-detail.ts //
import {Component} from '@angular/core';
import {NavController, NavParams} from 'ionic-angular';

@Component({
    templateUrl: 'build/pages/session-detail/session-detail.html'
})
export class SessionDetail {
    session:any;
    constructor(public nav:NavController, public params: NavParams) {
        this.session = this.params.get('session');
    }
}
```

We get session information from `NavParams` and we just assign that to `this.session` and display it.

The following code should be present in `session-detail.html`:

```
<!-- Template /app/pages/session-detail/session-detail.html -->
<ion-header>
    <ion-navbar>
        <ion-title>
            Session
        </ion-title>
    </ion-navbar>
</ion-header>
```

```
<ion-content>
    <ion-card>
        <ion-card-content>
            <ion-card-title>
                {{session.title}}
            </ion-card-title>
            <h3 *ngFor="let speaker of session.speakers">
             {{speaker.name}}</h3>
            <p [innerHTML]="session.abstract"><p>
        </ion-card-content>
    </ion-card>
</ion-content>
```

We display the session title, speaker name, and the session's abstract.

Defining the SpeakerDetail page

The `SpeakerDetail` page shows the details of the speaker, such as his name, picture, bio, and a list of sessions in this conference.

The following code should be present in `session-detail.ts`:

```
// /app/pages/speaker-detail/speaker-detail.ts //
import {Component} from '@angular/core';
import {NavController, NavParams} from 'ionic-angular';
import {Network, Connection} from 'ionic-native';
import {Subscription} from 'rxjs/Rx';
import {SessionDetail} from '../session-detail/session-detail';

@Component({
    templateUrl: 'build/pages/speaker-detail/speaker-detail.html'
})
export class SpeakerDetail {
    offline:any;
    speaker:any;
    disconnectSubscription:Subscription;
    onlineSubscription:Subscription;
    constructor(public nav:NavController, public params:NavParams) {
        this.speaker = this.params.get('speaker');
        this.disconnectSubscription =
         Network.onDisconnect().subscribe(()=> {
            this.offline = true;
        });
        this.onlineSubscription = Network.onConnect().subscribe(()=> {
            this.offline = false;
        });
    }
```

```
        openSession(session) {
            this.nav.push(SessionDetail, {session: session});
        }
        isOnline () {
            let networkState = Network.connection;
            return networkState !== Connection.UNKNOWN && networkState !==
             Connection.NONE;
        }
        ionViewDidEnter() {
         this.offline = !this.isOnline();
        }
    }
```

In the `SpeakerDetail` page, we get the speaker's information from `NavParam` and display it. We also have two network subscriptions, which tell us about when we go online or offline by changing the value of `this.offline`. The `openSession` function pushes the `SessionDetail` page on the navigation stack.

The following code should be present in `speaker-detail.html`:

```
    <!-- Template /app/pages/speaker-detail/speaker-detail.html -->
    <ion-header>
      <ion-navbar>
          <ion-title>{{speaker.name}}</ion-title>
      </ion-navbar>
    </ion-header>

    <ion-content>
      <ion-card>
      <img [src]="speaker.image_75" *ngIf="!offline"/>
      <ion-card-content>
        <ion-card-title>
          {{speaker.name}}
        </ion-card-title>
        <p>
          {{speaker.role}}
        </p>
      </ion-card-content>
    </ion-card>

    <ion-card>
        <ion-item-divider>
           Sessions
        </ion-item-divider>
        <button ion-item *ngFor="let session of speaker.sessions"
         (click)="openSession(session)">
             {{session.title}}
        </button>
    </ion-card>
    </ion-content>
```

We are displaying the speaker's name, a short bio, and a list of his/her sessions in the conference.

Defining the AboutPage page

The `AboutPage` shows the cover image for our conference, a simple introduction to our conference, and a link to get directions to the venue.

The following code should be present in `about.ts`:

```
// /app/pages/about/about.ts //
import {Component} from '@angular/core';
import {LaunchNavigator} from 'ionic-native';
@Component({
  templateUrl: 'build/pages/about/about.html'
})
export class AboutPage {
  venue:Array<number>;
  constructor() {
    this.venue = [30.8589, 75.8602];
  }
  navigate() {
    LaunchNavigator.navigate(this.venue)
    .then(data => {
    })
    .catch(()=> {
      console.log("Some Error Occured");
    });
  }
}
```

In `AboutPage`, `this.venue` has coordinates of our venue, which we will use for providing navigation. The `navigate` function opens the device's navigation app using our venue as the destination.

The following code should be present in `about.html`:

```
<!-- Template /app/pages/about/about.html -->
<ion-header>
  <ion-navbar primary>
    <button menuToggle>
      <ion-icon name="menu"></ion-icon>
      </button>
      <ion-title>
        About
      </ion-title>
    </ion-navbar>
</ion-header>

<ion-content class="page3">
 <ion-card>
  <img src="images/about.png"/>
  <ion-card-content>
    <p>LudhianJS is the first JavaScript Conference of North India. We
     have talks on various JS topics from Web, Mobile, Hardware, NodeJS
     etc. Join Us for great Learning Experience.</p>
  </ion-card-content>
 </ion-card>
 <ion-card>
   <ion-card-header>
     Venue Detail here
   </ion-card-header>
   <ion-card-content>
    <button ion-item (click)="navigate()">
      Click to Get Directions
    </button>
   </ion-card-content>
 </ion-card>
</ion-content>
```

We have an image, `images/about.png`, which we are using as the cover image for `About Page`. You can use any logo for your conference here. Then we just give some information about our conference and finally a button which opens the device's navigation app to supply directions to the venue.

Defining the FavoritePage page

`FavoritePage` lists all the user's favorite sessions. The following code should be present in `favorite.ts`:

```
// /app/pages/favorite/favorite.ts //
import {Component} from '@angular/core';
import {NavController, LocalStorage, Storage} from 'ionic-angular';
import {SessionDetail} from '../session-detail/session-detail';

@Component({
    templateUrl : 'build/pages/favorite/favorite.html'
})
```

```
export class FavoritePage {
    storage = new Storage(LocalStorage);
    favorites:Array<any>;
    constructor(public nav:NavController) {
        this.storage.get('favorite')
        .then(data => {
           this.favorites = JSON.parse(data);
        })
        .catch(()=> {
            this.favorites = [];
        });
    }
    openSession(session) {
        this.nav.push(SessionDetail, {session: session});
    }
    deleteSession(index) {
        this.favorites.splice(index,1);
        this.storage.set('favorite', JSON.stringify(this.favorites));
    }
}
```

We get a list of favorite sessions from `localStorage` and then we display it. `openSessions` opens the `SessionDetail` page and `deleteSession` deletes the session from the favorites list.

The following code should be present in `favorite.html`:

```
<!-- Template /app/pages/favorite/favorite.html -->
<ion-header>
    <ion-navbar>
        <ion-title>Favorite</ion-title>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list>
     <ion-item-sliding *ngFor="let session of favorites; let i = index"
      >
        <button ion-item (click)="openSession(session)">
          <h3>{{session.title}}</h3>
          <p>
            {{session.times}}
            {{session.space}}
          </p>
        </button>
        <ion-item-options>
            <button (click)="deleteSession(i)">Delete</button>
        </ion-item-options>
     </ion-item-sliding>
```

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

We have a list of `ion-item-sliding`, which shows the title of the session; when you swipe the item, you get a button to delete the session from the favorites list.

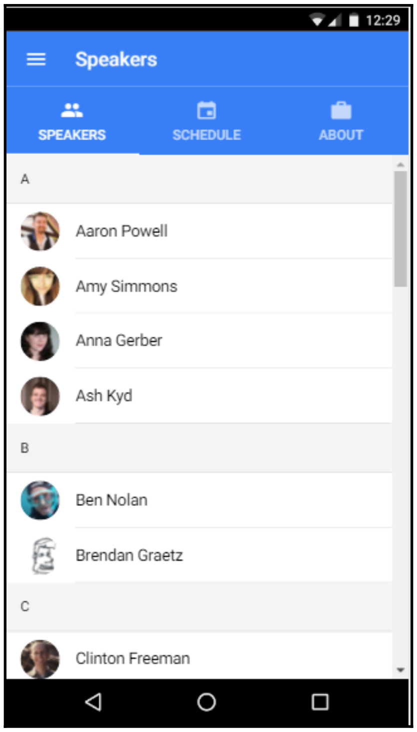
Running our app

Let's run our application on an actual device. Run the following code:

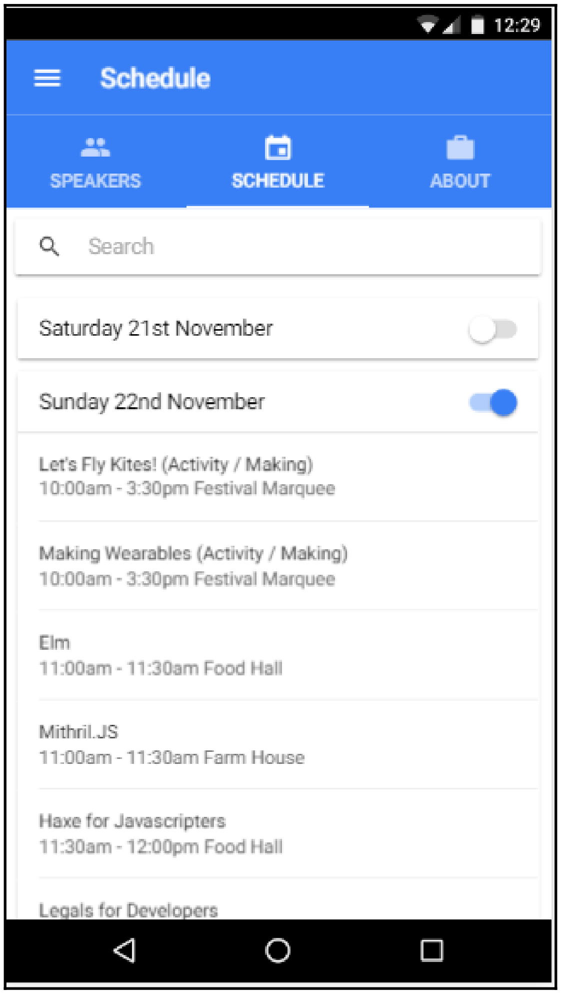
```
    ionic platform add android
    ionic run android
```

App screenshots

The following screenshot shows `SpeakersPage`:



The following screenshot shows `SchedulePage`:



Saturday 21st November

Sunday 22nd November

Let's Fly Kites! (Activity / Making)
10:00am - 3:30pm Festival Marquee

Making Wearables (Activity / Making)
10:00am - 3:30pm Festival Marquee

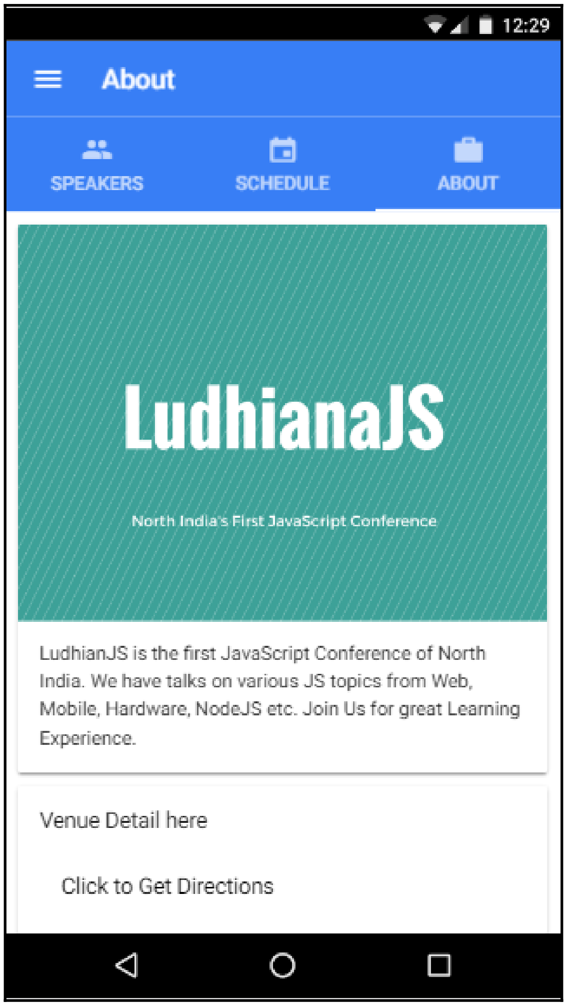
Elm
11:00am - 11:30am Food Hall

Mithril.JS
11:00am - 11:30am Farm House

Haxe for Javascripters
11:30am - 12:00pm Food Hall

Legals for Developers

The following screenshot shows `AboutPage`:

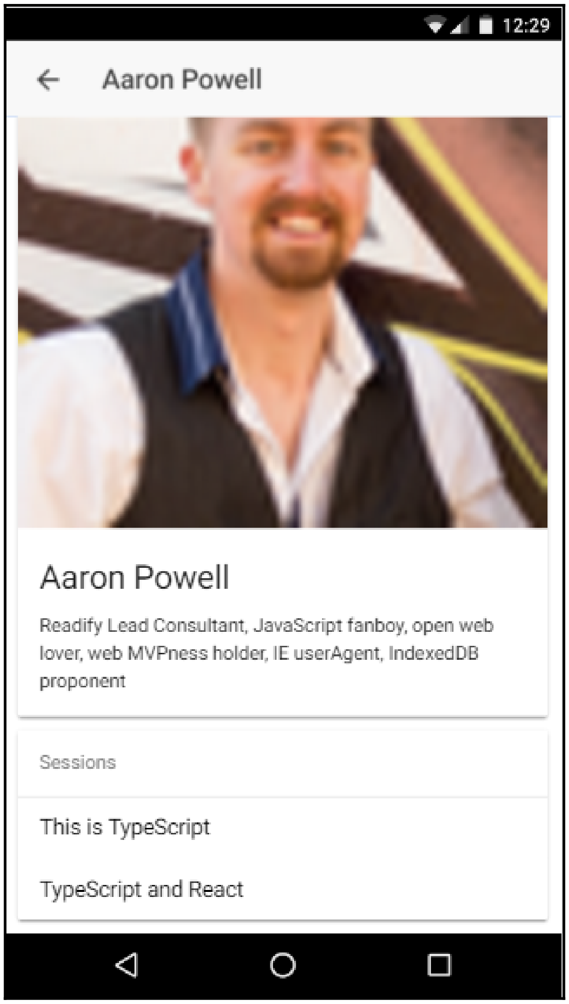


LudhianJS is the first JavaScript Conference of North India. We have talks on various JS topics from Web, Mobile, Hardware, NodeJS etc. Join Us for great Learning Experience.

Venue Detail here

Click to Get Directions

The following screenshot shows `SpeakerDetail`:



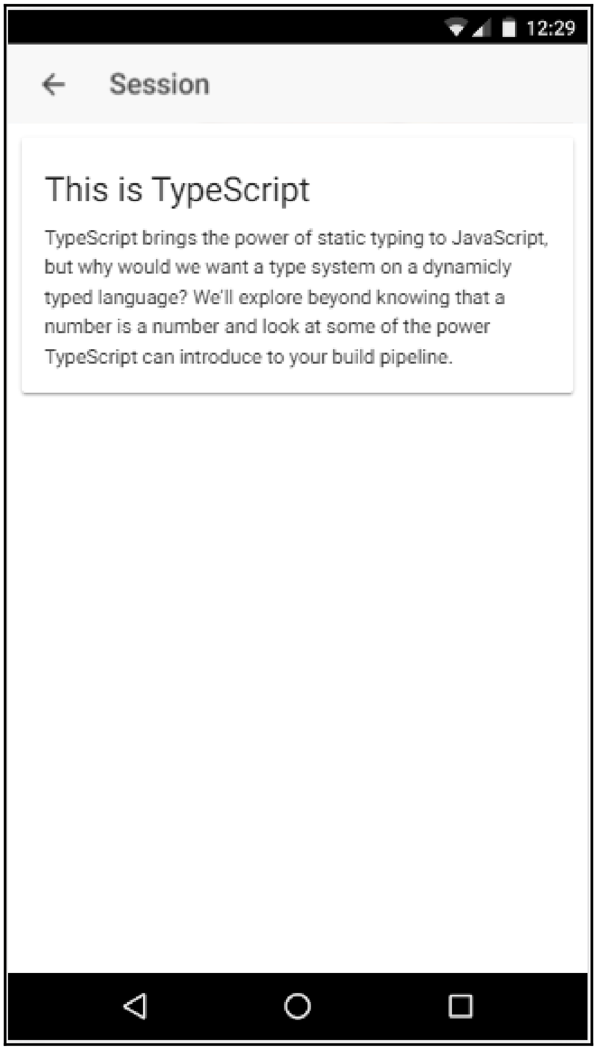
Aaron Powell

Readify Lead Consultant, JavaScript fanboy, open web lover, web MVPness holder, IE userAgent, IndexedDB proponent

This is TypeScript

TypeScript and React

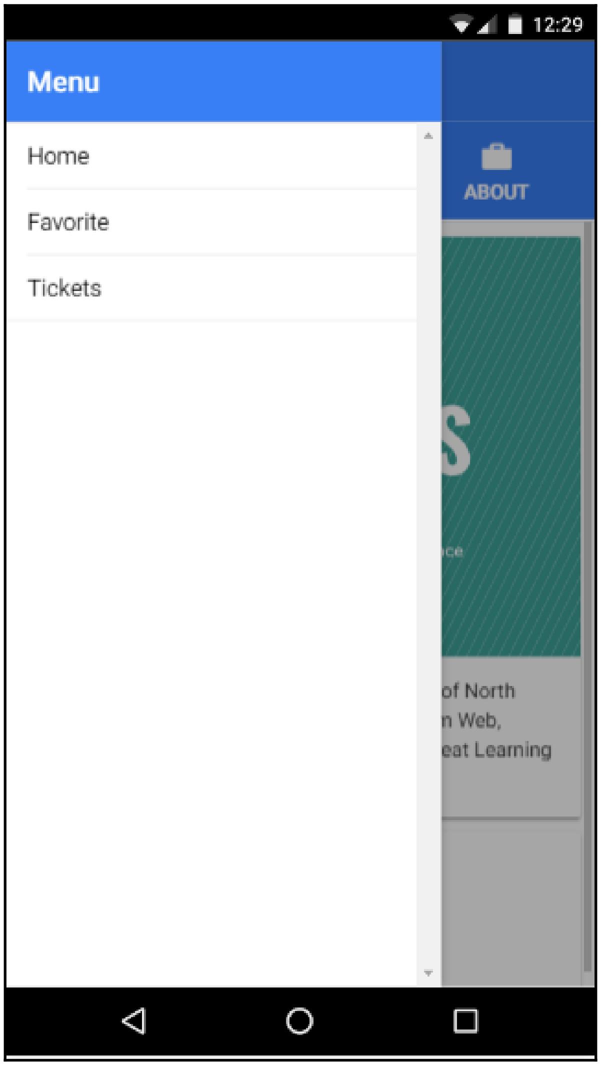
The following screenshot shows `SessionDetail`:



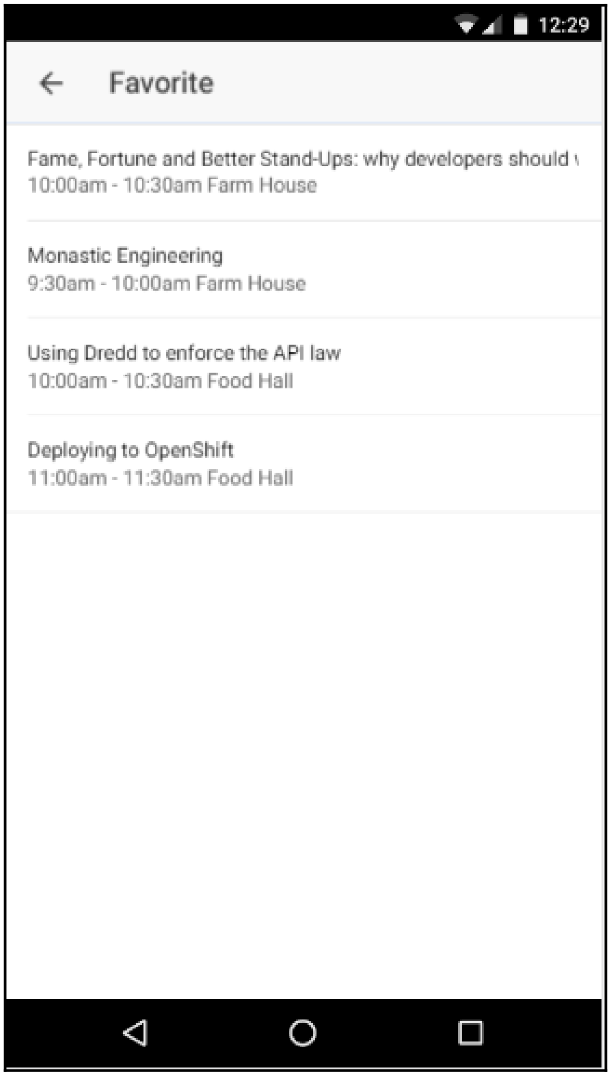
This is TypeScript

TypeScript brings the power of static typing to JavaScript, but why would we want a type system on a dynamicly typed language? We'll explore beyond knowing that a number is a number and look at some of the power TypeScript can introduce to your build pipeline.

The following screenshot shows the side menu:



The following screenshot shows `FavoritePage`:



Fame, Fortune and Better Stand-Ups: why developers should
10:00am - 10:30am Farm House

Monastic Engineering
9:30am - 10:00am Farm House

Using Dredd to enforce the API law
10:00am - 10:30am Food Hall

Deploying to OpenShift
11:00am - 11:30am Food Hall

Summary

In this chapter, we created a conference companion app. Our app shows a list of speakers, a schedule, filtering a schedule, and a favorites list, and works offline. The app allows users to share sessions, get directions to the venue, and create a favorites list. It also adds favorite sessions to a device's calendar.

We also learned how we can use the RxJS library to write functional code for asynchronous events, such as inputs or button clicks.

4
StockMarket App

In the previous chapter, we created a conference app. In this chapter, we are going to create a StockMarket app, which will give us real-time stock information for our favorite companies.

We will learn the following things:

- Component-based app development
- **RxJS** in depth
- Using **SqlStorage** in Ionic apps
- Creating a custom pipe
- Creating responsive charts in mobile app
- Using the Yahoo API to get stock info

Introduction

In all previous chapters, we have used Ionic's pages heavily and created most of our app inside pages. In this chapter, we will create an application that will have reusable Angular 2 components. Not only will we create them, we will also reuse these components inside our application to show how it works. Of course, we will also learn how to use Yahoo's Finance API to get stock information.

Along with this, we will use the Ionic's `SqlStorage` class to create a client-side **SQL** database and query it with SQL commands. We will also create a custom pipe.

In the stock market, there are stock exchanges, which hold the data. Each public company is given a special unique keyword called a **symbol**, which uniquely identifies the company in the exchange. For example, Google has the GOOGL symbol and Yahoo's symbol is **YHOO**. We will be using all this terminology in this chapter.

Yahoo Finance API

In the application, we will just use Yahoo's Finance API and **Yahoo Query Language** (**YQL**). YQL is similar to **Structured Query Language** (**SQL**), but for querying public APIs instead of databases. We will use four endpoints. These are as follows:

- Getting stock information
- Getting symbol names
- Getting historical data
- Getting detailed stock information

Stock information endpoint

We will use YQL to get stock information.

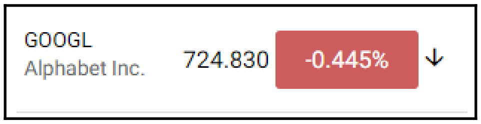
Query is as follows:

```
select * from yahoo.finance.quote where symbol IN('${symbol}')`
```

Do note that we need to replace `${symbol}` with the actual symbol, such as GOOGL, YHOO, and so on. This is the result we will get back from the API.

In order to execute any YQL query, we have to send a HTTP request to the Yahoo servers at `http://query.yahooapis.com/v1/public/yql?q=${query}`.

Here, `${query}` must be replaced with an actual URI-encoded query. We receive the following:

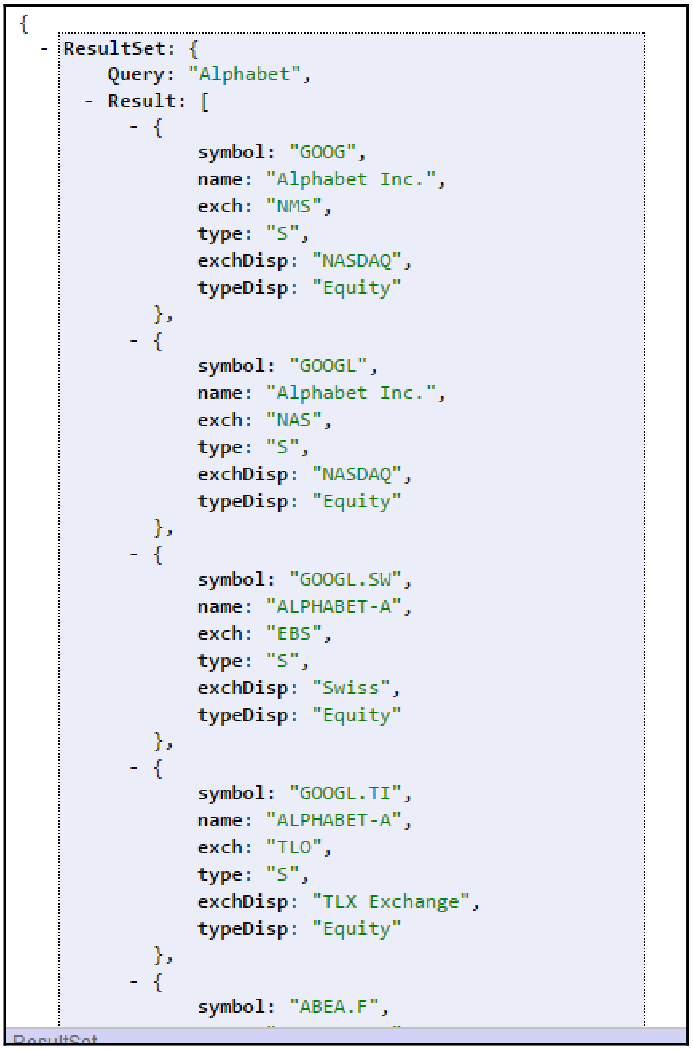

GOOGL
Alphabet Inc.
724.830
-0.445%

Symbol names endpoint

We will use the following endpoint to get the symbol for the company we want: `https://s.yimg.com/aq/autoc?query=${name}&region=CA&lang=en-CA`.

We will replace `${name}` with the name of the company.

The result we will get back will consist of various symbols matching our query, as follows:



```
{
  - ResultSet: {
        Query: "Alphabet",
      - Result: [
          - {
                symbol: "GOOG",
                name: "Alphabet Inc.",
                exch: "NMS",
                type: "S",
                exchDisp: "NASDAQ",
                typeDisp: "Equity"
            },
          - {
                symbol: "GOOGL",
                name: "Alphabet Inc.",
                exch: "NAS",
                type: "S",
                exchDisp: "NASDAQ",
                typeDisp: "Equity"
            },
          - {
                symbol: "GOOGL.SW",
                name: "ALPHABET-A",
                exch: "EBS",
                type: "S",
                exchDisp: "Swiss",
                typeDisp: "Equity"
            },
          - {
                symbol: "GOOGL.TI",
                name: "ALPHABET-A",
                exch: "TLO",
                type: "S",
                exchDisp: "TLX Exchange",
                typeDisp: "Equity"
            },
          - {
                symbol: "ABEA.F",
```

Historical data endpoint

To get historical data, we use the following YQL query:

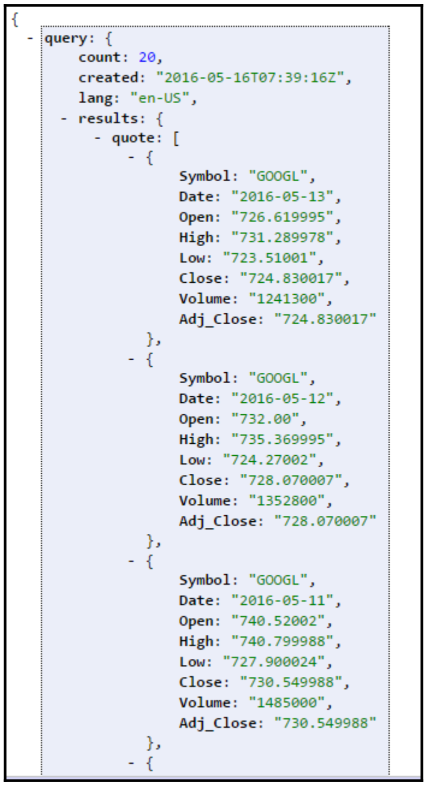
```
select * from yahoo.finance.historicaldata where symbol = "GOOGL" and
startDate = "2016-04-16" and endDate = "2016-05-16";
```

We also have to encode this query to make it work, as mentioned earlier.

The encoded query within a URL will look something like the following:

```
http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20yahoo.financ
e.historicaldata%20where%20symbol%20%3D%20%22GOOGL%22%20and%20startDate%20%
3D%20%222016-04-16%22%20and%20endDate%20%3D%20%222016-05-16%22&format=json&
env=store%3A%2F%2Fdatatables.org%2Falltableswithkeys
```

The response we will get back will have historical stock data for the symbol, for each day:



```
{
  - query: {
        count: 20,
        created: "2016-05-16T07:39:16Z",
        lang: "en-US",
      - results: {
          - quote: [
              - {
                    Symbol: "GOOGL",
                    Date: "2016-05-13",
                    Open: "726.619995",
                    High: "731.289978",
                    Low: "723.51001",
                    Close: "724.830017",
                    Volume: "1241300",
                    Adj_Close: "724.830017"
                },
              - {
                    Symbol: "GOOGL",
                    Date: "2016-05-12",
                    Open: "732.00",
                    High: "735.369995",
                    Low: "724.27002",
                    Close: "728.070007",
                    Volume: "1352800",
                    Adj_Close: "728.070007"
                },
              - {
                    Symbol: "GOOGL",
                    Date: "2016-05-11",
                    Open: "740.52002",
                    High: "740.799988",
                    Low: "727.900024",
                    Close: "730.549988",
                    Volume: "1485000",
                    Adj_Close: "730.549988"
                },
              - {
```

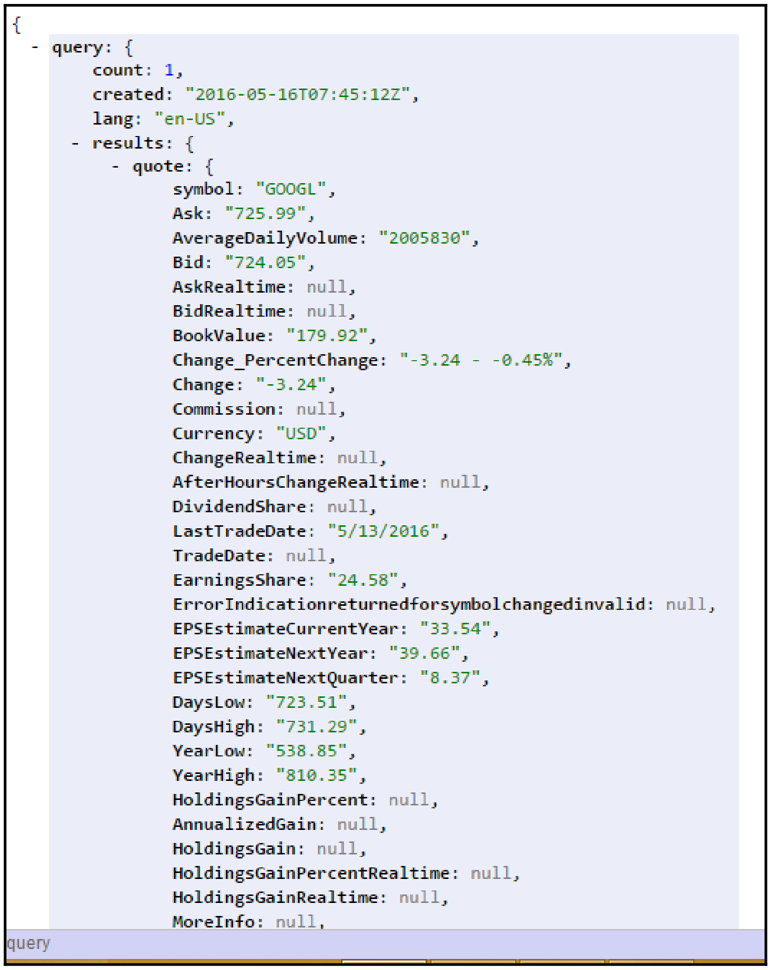
Detailed stock information

Again, we will use YQL here. The URL will remain the same as for the preceding example, but query will be as follows:

```
select * from yahoo.finance.quotes where symbol IN ("' ${symbol} '");
```

Here, `${symbol}` will be replaced with the actual symbol of the company, such as YHOO, GOOGL, and so on.

The response to the query will be as follows:



```
{
  - query: {
        count: 1,
        created: "2016-05-16T07:45:12Z",
        lang: "en-US",
      - results: {
          - quote: {
                symbol: "GOOGL",
                Ask: "725.99",
                AverageDailyVolume: "2005830",
                Bid: "724.05",
                AskRealtime: null,
                BidRealtime: null,
                BookValue: "179.92",
                Change_PercentChange: "-3.24 - -0.45%",
                Change: "-3.24",
                Commission: null,
                Currency: "USD",
                ChangeRealtime: null,
                AfterHoursChangeRealtime: null,
                DividendShare: null,
                LastTradeDate: "5/13/2016",
                TradeDate: null,
                EarningsShare: "24.58",
                ErrorIndicationreturnedforsymbolchangedinvalid: null,
                EPSEstimateCurrentYear: "33.54",
                EPSEstimateNextYear: "39.66",
                EPSEstimateNextQuarter: "8.37",
                DaysLow: "723.51",
                DaysHigh: "731.29",
                YearLow: "538.85",
                YearHigh: "810.35",
                HoldingsGainPercent: null,
                AnnualizedGain: null,
                HoldingsGain: null,
                HoldingsGainPercentRealtime: null,
                HoldingsGainRealtime: null,
                MoreInfo: null,
```

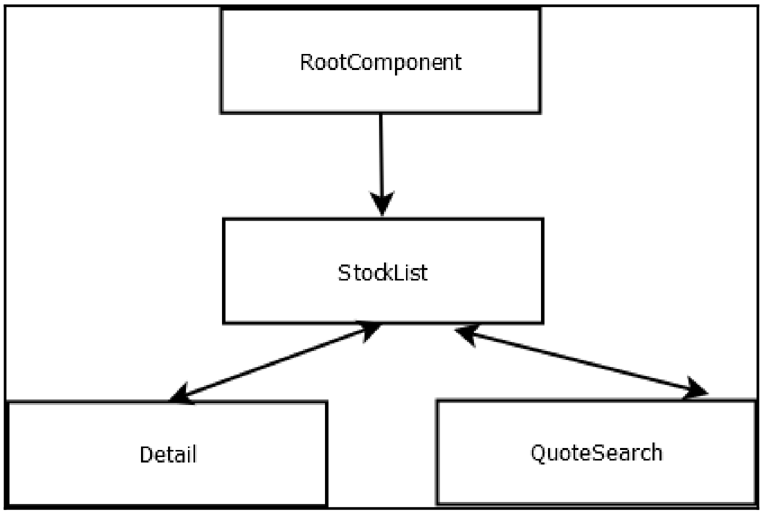
Defining app functionalities

In this section, we will define various functionalities of our application. We will be including the following functionalities in our application:

- Shows list of stocks that you follow
- Stock information must get updated every five seconds
- Historical data of a stock in chart form
- Detailed information of stock
- User can take notes on each stock for later use
- Search symbol by company name

App flow

The following figure shows how the control will flow inside our application:



Let's understand the preceding flow chart:

- **RootComponent**: This is the root of Ionic. It is defined inside the `/app/app.ts` file.

- **StockList page**: This page shows the list of stocks that you watch or follow. Each stock gets updated every five seconds. By clicking on each stock, the user will go to the `detail` page. In the navbar, there is a **+** button. After clicking on the **+** button, the user will go to the `QuoteSearch` page.
- **QuoteSearch page**: This page allows the user to search for companies and watch their stock. The **X** button on top sends the user back to the `StockList` page.
- **Detail page**: This page shows various information about stock, including stock info that updates every five seconds, historical data charts, and user notes. The **Back** button allows the user to go back to the `StockList` page.

Scaffolding and setting up the app

We will start by scaffolding a blank application. Run the following command:

```
ionic start ionic2-stockmarket blank --v2 --ts
```

Notice the `--v2` and `--ts` tags for scaffolding an Ionic 2 app based on **TypeScript**.

Before we start writing code for the app, we need to install some project-specific dependencies.

Installing dependencies

The following are some project-specific dependencies that we need to install.

Installing the Chartist library

In the app, we will use `Chartist.js` to create responsive charts inside our application.

Download the `Chartist.js` and `Chartist.css` files from this link: `https://github.com/gionkunz/chartist-js/tree/develop/dist`. Put them inside `www/vendor` folder.

You can read the documentation for `Chartist.js` at `https://gionkunz.github.io/chartist-js/api-documentation.html`.

Coding part

Now we have everything set up to start working on our application. Let's define the following things:

- The `index.html` page for the app
- Defining our main `app.ts` file
- Creating providers for our application
- Creating reusable components
- Creating pages for our app

Our index.html file

On the `index.html` page, we will only add a reference to our Chartist library.

Add the CSS reference as follows:

```
<link href="vendor/chartist.css" rel="stylesheet">
```

Add the JavaScript reference:

```
<script src="vendor/chartist.js"></script>
```

Defining app.ts

This is the root of our application, and it defines the root component using the `@App` decorator. This is where we inject all of our dependencies using the following code:

```
/* /app/app.ts */
import {Component} from '@angular/core';
import {Platform, ionicBootstrap} from 'ionic-angular';
import {StatusBar} from 'ionic-native';
import {StockList} from './pages/stock-list/stock-list';
import {StockInfo} from './providers/stock-info';
import {StorageProvider} from './providers/storage';

@Component({
  template: '<ion-nav [root]="rootPage"></ion-nav>'
})
export class MyApp {
  rootPage: any = StockList;
  constructor(platform: Platform) {
    platform.ready().then(() => {
      StatusBar.styleDefault();
    });
  }
}

ionicBootstrap(MyApp, [StockInfo, StorageProvider]);
```

We have imported `StockList`, `StockInfo`, and `StorageProvider`, which we will create next. We have also set the `rootPage` to the `StockList` page. The rest of the coding is default, Ionic 2 boilerplate code. ionicBootstrap allows us to bootstrap our Ionic 2 application.

Providers for our application

We will have the following providers:

- `StorageProvider`
- `StockInfo`

Defining StorageProvider

`StorageProvider` abstracts our interaction with the client-side SQL database. We will use it to store symbol information and user notes. The following code should be present in `storage.ts`:

```
    /* /app/providers/storage.ts */
    import {Injectable} from '@angular/core';
    import {Storage, SqlStorage} from 'ionic-angular';

    @Injectable()
    export class StorageProvider {
        storage = new Storage(SqlStorage);
        constructor() {
            this.storage.query("create table IF NOT EXISTS quotes(stock
             unique,
             value)");
            this.storage.query("create table IF NOT EXISTS notes(symbol,
             value)");
        }
        setQuote(key, value) {
            value = JSON.stringify(value);
            let query = `insert into quotes values('${key}', '${value}');`;
            return this.storage.query(query);
        }
        getAllQuotes() {
            return this.storage.query("select * from quotes");
        }
        removeQuote(key) {
            let query = `delete from quotes where stock='${key}'`;
            return this.storage.query(query);
        }
        setNotes(symbol, value) {
            let query = `insert into notes(symbol, value)
             values('${symbol}',
             '${value}')`;
            return this.storage.query(query);
        }
        getAllNotes(symbol) {
            let query = `select owed, * from notes where
             symbol='${symbol}';`;
            return this.storage.query(query);
        }
        removeNote(10owed) {
            let query = `delete from notes where owed = ${owed}`;
            return this.storage.query(query);
        }
    }
```

Let's understand the preceding code:

- `constructor()` creates a storage member for accessing **WebSQL**, which is our client-side SQL database, using the `Storage` and `SqlStorage` class. It also creates two tables, `quote` and `notes`, for us.
- `setQuote(key, value)` saves symbol information in the database for later use. We will get this symbol information from the `QuoteSearch` page.
- `getAllQuotes()` returns list of symbols that we have stored in database.
- `removeQuote(key)` removes an entry from the `quote` table using a `key` to find it.
- `setNotes(symbol, value)` inserts a note into the `notes` table for a given symbol using the `symbol` variable.
- `getAllNotes(symbol)` returns a list of notes for a particular symbol.
- `removeNote(10owed)` removes a note from the `notes` table for the specific item `10owed`.

Defining the StockInfo provider

The `StockInfo` provider provides us functions to get information from the Yahoo API.

StockInfo provider code

The following code should be present in `stock-info.ts`:

```
// /app/providers/stock-info.ts
import {Injectable} from '@angular/core';
import {Http} from '@angular/http';
import {Observable} from 'rxjs/Rx';
import 'rxjs/operator/map';

@Injectable()
export class StockInfo {
    constructor(public http:Http) {}
    encodeURI(string) {
        return encodeURIComponent(string).replace(/"/g,
         "%22").replace(/\ /g,
         "%20").replace(/[!'()]/g, encodeURI);
    }

    getPriceInfo(symbol) {
        let query = `select * from yahoo.finance.quote where symbol
         IN('${symbol}')`;
        let url = 'http://query.yahooapis.com/v1/public/yql?q=' +
         this.encodeURI(query) +
         '&format=json&env=http://datatables.org/alltables.env';
         // let url =
      `https://finance.yahoo.com/webservice/v1/symbols/${symbol}/quote?
         format=json&view=detail`;
        let stocks = Observable.interval(5 * 1000)
        .flatMap(()=> {
            return this.http.get(url).retry(5);
        })
        .map(data => {
           let jsonData = data.json();
           return jsonData.query.results.quote;
        });
        return stocks;
    }

    getDetail(symbol) {
        let query = `select * from yahoo.finance.quotes where symbol IN
         ("' ${symbol} '")`;
        let url = 'http://query.yahooapis.com/v1/public/yql?q=' +
         this.encodeURI(query) +
         '&format=json&env=http://datatables.org/alltables.env';
        return this.http.get(url).map(data =>
         data.json().query.results.quote);
    }

    getdate(m) {
        let d = new Date();
        d.setMonth(d.getMonth()-m);
        let year = d.getFullYear().toString();
        let month = (d.getMonth()+ 1).toString();
/* It is good to remember that JavaScript months start from 0. So January
is 0 in JavaScript. That's why month is incremented by 1 to make it
correctly converted into String. */
        let day = d.getDate().toString();
        if(month.length === 1) {
            month = "0" + month;
        }
        if(day.length === 1) {
            day = "0" + day;
        }
        let formatted = `${year}-${month}-${day}`;
        return formatted;
    }
    getQuotes(name) {
        let url = `https://s.yimg.com/aq/autoc?
         query=${name}&region=CA&lang=en-
         CA`;
        return this.http.get(url)
        .map(data => {
            let result = data.json();
            return result.ResultSet.Result;
        });
    }
    getChart(symbol,from=1) {
        let todayDate = this.getdate(0);
        let fromDate = this.getdate(from);
        let query = 'select * from yahoo.finance.historicaldata where
         symbol = "' + symbol + '" and startDate = "' + fromDate + '"
         and endDate = "' + todayDate + '"';
        let url = 'http://query.yahooapis.com/v1/public/yql?q=' +
         this.encodeURI(query) +
         '&format=json&env=http://datatables.org/alltables.env';
        return this.http.get(url).map((data) => {
            console.log(url);
            let result = data.json().query.results;
            if(result) {
                return result.quote
            } else {
                return [];
            }
        });
    }
 }
```

Let's understand the preceding code:

- `encodeURI(string)` encodes the string using a URI encoding scheme.
- `getPriceInfo(symbol)` fetches the stock information for a given stock after an interval of five seconds. Imagine writing this code using **promises**. This is the real advantage of using RxJS. We are not only fetching data every five seconds, this function will also retry five times if it fails to fetch data.
- `getDetail(symbol)` gets the stock details of the symbol provided and returns `Observable`.
- `getDate(m)` returns the date, `m` months from now. In other words, if `m` is 0, it will return the current date; if `m` is `1` it will return date of `1` month from today's date. It is worth noting that the date returned is in the form of a string with the format `yyyy-mm-dd`.
- `getQuotes(name)` returns an observable with the symbol and details of companies that match the `name` string.
- `getChart(symbol, from)` returns an `Observable` with historical data of a specific `symbol` with a given `from` value. `from` is the number that defines the number of months of historical data from today.
- `Observable.interval` returns the `Observable` stream according to given time.
- `flatMap` basically converts each `Observable` returned by `interval` into another observable, which in our case is a HTTP GET request. Then, using the `map` operator, we are just mapping the data returned by HTTP request and returning something different, just like a map of arrays.

If you want to learn about RxJS, check out this wonderful explanation by Andre Staltz at `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`.

Components of our app

We will have three components in our application:

- `StockCmp`
- `StockChart`
- `StockDetailCmp`

StockCmp component

`StockCmp` looks like this:

```
<stock [run]="run" sliding="true" [info]="quote" (delete)="deleteItem(I,
quote.symbol"" > </stock>
```

Inputs

- `run`: It is a Boolean. If `run` is `true`, our stock component will fetch data from server. Otherwise, it will stop.
- `sliding`: It makes a stock item sliding or non-sliding based on its value.
- `info`: It is used to provide some initial data to a component such as a symbol, company name, and so on.

Outputs

- `delete`: Each time the user clicks on the **Delete** button of the stock item, a delete event is fired.

Component pipe

The following code is a very simple pipe that converts the input string value to `Float` value with a precision of three decimal points:

```
// /app/component/stock/pipe.ts
import {Pipe} from ''@angular/core'';
@Pipe({
    name: ''numberString''
})
export class NumberStringPipe {
    transform(val, args) {
        return Number.parseFloat(val).toFixed(3);
    }
}
```

Take a look at Angular 2's documentation on pipes at `https://angular.io/docs/ts/latest/guide/pipes.html` for information.

Component style sheet

The following code should be present in `stock.scss`:

```
// /app/component/stock.scss
.stockcmp {
    .number {
      padding:10px 20px;
      text-align: center;
      min-width: 120px;
      max-width:120px;
      border-radius:3px;
    };
    .percentage_pos {
        @extend .number;
        background: mediumseagreen;
        color:white;
    };
    .percentage_neg {
       @extend .number;
       background: indianred;
       color:white;
    }
}
```

Component controller

The following code should be present in `stock.ts`:

```
// /app/component/stock/stock.ts //
import {Component, Input, Output, OnChanges, OnInit, OnDestroy,
EventEmitter} from ''@angular/core'';
import {ViewController, Events} from ''ionic-angular'';
import {Http} from ''@angular/http'';
import {StockInfo} from ''../../providers/stock-info'';
import {NumberStringPipe} from ''./pipe'';
import {Subscription} from ''rxjs/Rx'';
```

```
@Component({
    selector: ''stock'',
    templateUrl: ''build/component/stock/stock.html'',
    pipes: [NumberStringPipe]
})

export class StockCmp {
    @Input() sliding;
    @Input() info;
    @Input() run;
    @Output() delete =  new EventEmitter();
    data = {};
    down:Boolean;
    loading:Boolean;
    symbolStock:any;
    subscription:Subscription;
    constructor(public http:Http, public ss:StockInfo, public vc:
     ViewController, public events:Events) {
    this.loading = true;
    }
    ngOnInit() {
       this.sliding = (this.sliding ==="true");
       this.symbolStock = this.ss.getPriceInfo(this.info.symbol);
       this.subscription =  this.symbolStock.subscribe(data => {
           this.data = data;
           this.loading = false;
       });
    }
    percentageSign(value) {
        if(value === undefined || value === null) {
            return ''number'';
        } else if(value[0] =="""") {
           this.down = true;
           return ''percentage_neg''
        } else {
            this.down = false;
            return ''percentage_pos'';
        }
    }
    ngOnChanges() {
        if(this.run) {
           if(this.symbolStock && this.subscription) {
                this.subscription =  this.symbolStock.subscribe(data => {
                    this.data = data;
                    this.loading = false;
                });
           }
        } else {
            if(this.subscription) {
              this.subscription.unsubscribe();
            }
        }
    }
    deleteItem() {
        this.delete.emit""delet"");
    }
}
```

Let's understand the preceding code:

- `constructor()` initializes the `loading` member to `true`.
- `ngOnInit()` is the Angular 2 life cycle hook function. It fires when that component initializes. We are basically subscribing to `Observable` returned by `getPriceInfo` function of our `service.ts`.
- `percentageSign(value)` returns a string based on `value`. It is used for applying CSS in a template.
- `ngOnChanges()` is also a life cycle hook function. It runs when any of the inputs to component is changed. With this, we are checking the run input and subscribing or unsubscribing to our `Observable` accordingly.
- `deleteItem()` omits deleted output.

For more information on Angular's life cycle hooks, take a look at `https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html`.

Component template

The following code should be present in `stock.html`:

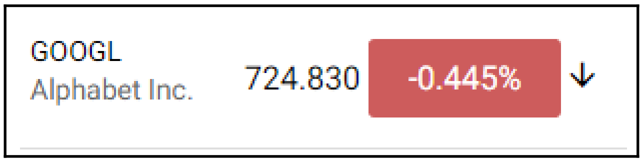
```
<!-- /app/component/stock/stock.html -->
<ion-item-sliding>
  <a ion-item text-wrap class""stockcm"">
      <ion-row>
        <h3>{{info.symbol}}</h3>
      </ion-row>
      <ion-row center width-25>
        <p>{{info.name}}</p>
      </ion-row>
      <ion-spinner item-left *ngIf""loadin""></ion-spinner>
      <span item-right *ngIf""!loadin"">
        <span>{{data.LastTradePriceOnly | numberString}}</span>
```

```
            <ion-note class""pric"" [ngClass]""percentageSign(data.Change"">
             {{data.Change | numberString}}%</ion-note>
            <ion-icon *ngIf""!dow"" name""arrow-u""></ion-icon>
            <ion-icon *ngIf""dow"" name""arrow-dow""></ion-icon>
          </span>
      </a>

      <ion-item-options *ngIf""slidin"">
            <button primary (click)""deleteItem("">
              <ion-icon name""delet""></ion-icon>
              Delete
            </button>
      </ion-item-options>
    </ion-item-sliding>
```

We have created a sliding `ion-item` with information about our stock, which updates every five seconds. We show an `ion-spinner` when our data is loading. There is also a **Delete** button, which fires the deleted output.

Our component looks like the following:



StockChart

`StockChart` shows a chart with the historical data of a particular stock. It looks like this:

```
    <stock-chart class="ct-perfect-fourth" [data]="data"></stock-chart>
```

Input

- `data` provides all historical data that we require to display a chart.

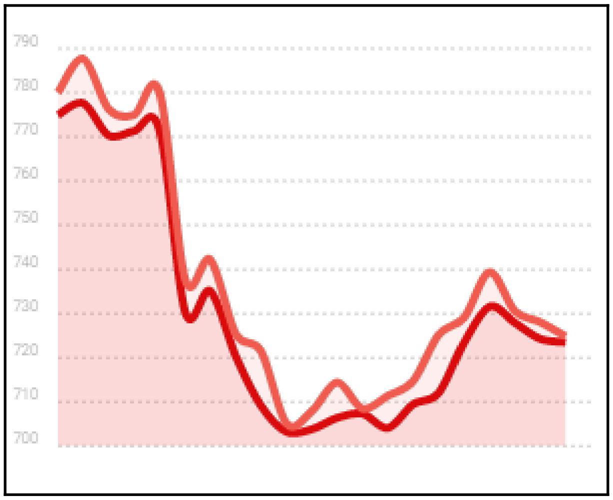
Component controller

The following code should be present in `stock-chart.ts`:

```
// /app/component/stock-chart/stock-chart.ts
import {Component, Input, Output, ViewChild, ElementRef, OnChanges} from
'@angular/core';
declare var Chartist;
@Component({
    selector: 'stock-chart',
    template: ``
})
export class StockChart {
    @Input() data:{};
    constructor(public elm: ElementRef) {}
    ngOnChanges() {
        if(this.data) {
            let options = {
                showPoint: false,
                axisX: {
                    showGrid: true,
                    showLabel:true
                },
                showArea:true,
                chartPadding: {
                    top: 15,
                    bottom: 5,
                    left: 10
                }
            };
            let chart = new Chartist.Line(this.elm.nativeElement,this.data,
             options);
        }
    }
```

In the constructor, `this.elm` is a reference to the `stock-chart` element. The `ngOnChanges()` function creates a new line chart using `Chartist.Line` inside our `stock-chart` element, using data provided by through `this.data`. We are also configuring our chart using options.

Our chart component looks like this:



StockDetailCmp

`StockDetailCmp` shows detailed information about the stock. The markup looks like this:

```
<stock-detail [symbol]="quote.symbol"></stock-detail>
```

Input

- `symbol` is the symbol of the stock whose details we want to find.

Component style sheet

The following code should be present in `stock-detail.scss`:

```
/* /app/component/stock-detail/stock-detail.scss */
.stock-detail-cmp {
    ion-col:nth-child(2n-1) {
        font-weight: bold;
    }
}
```

Component controller

The following code should be present in `detail.ts`:

```
// /app/component/stock-detail/stock-detail.ts
import {Component, Input, OnInit} from '@angular/core';
import {StockInfo} from '../../providers/stock-info';

@Component({
    selector: 'stock-detail',
    templateUrl: 'build/component/stock-detail/stock-detail.html',
})

export class StockDetailCmp {
    @Input() symbol;
    detail = {};
    constructor(public stockInfo: StockInfo) {
    }
    ngOnInit() {
        this.stockInfo.getDetail(this.symbol)
        .subscribe(data => {
            this.detail = data;
        });
    }
}
```

We have created two members: one is `symbol` for the component's input, and the other is `data` for storing the symbol's stock data. When the `ngOnInit()` function is fired, it gets the stock details and assigns these details to `this.data`.

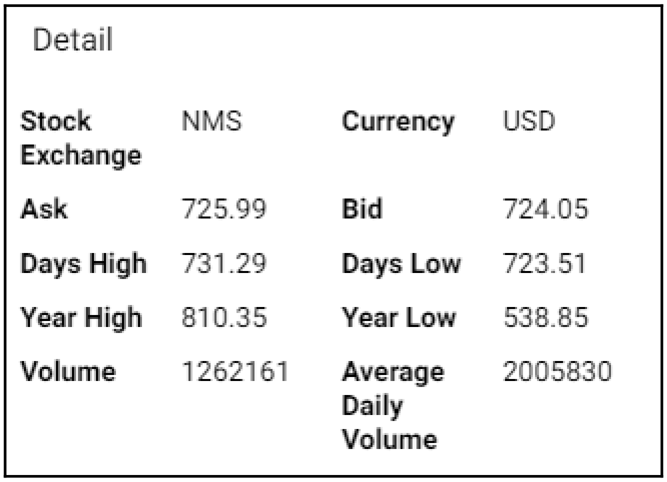
Component template

The following code shows the details of a stock using a tabular structure created with Ionic `grid` using `ion-row` and `ion-col`:

```
<!-- /app/component/stock-detail/stock-detail.html -->
<ion-card class="stock-detail-cmp">
    <ion-item>Detail</ion-item>
    <ion-row text-wrap>
        <ion-col>Stock Exchange</ion-col>
        <ion-col>{{detail.StockExchange}}</ion-col>
        <ion-col>Currency</ion-col>
        <ion-col>{{detail.Currency}}</ion-col>
    </ion-row>
    <ion-row text-wrap>
        <ion-col>Ask</ion-col>
        <ion-col>{{detail.Ask}}</ion-col>
```

```
            <ion-col>Bid</ion-col>
            <ion-col>{{detail.Bid}}</ion-col>
        </ion-row>
        <ion-row text-wrap>
            <ion-col>Days High</ion-col>
            <ion-col>{{detail.DaysHigh}}</ion-col>
            <ion-col>Days Low</ion-col>
            <ion-col>{{detail.DaysLow}}</ion-col>
        </ion-row>
        <ion-row text-wrap>
            <ion-col>Year High</ion-col>
            <ion-col>{{detail.YearHigh}}</ion-col>
            <ion-col>Year Low</ion-col>
            <ion-col>{{detail.YearLow}}</ion-col>
        </ion-row>
        <ion-row text-wrap>
            <ion-col>Volume</ion-col>
            <ion-col>{{detail.Volume}}</ion-col>
            <ion-col>Average Daily Volume</ion-col>
            <ion-col>{{detail.AverageDailyVolume}}</ion-col>
        </ion-row>
    </ion-card>
```

Our details component will look like this:



Detail

Stock Exchange	NMS	**Currency**	USD
Ask	725.99	**Bid**	724.05
Days High	731.29	**Days Low**	723.51
Year High	810.35	**Year Low**	538.85
Volume	1262161	**Average Daily Volume**	2005830

Pages of our application

Now we have to define all the providers and services for our application. Let's define the pages of our application.

We will have the following pages in our application:

- The `StockList` page
- The `QuoteSearch` page
- The `Detail` page

Defining the StockList page

The `StockList` page lists all stock that our user will watch.

StockList Controller

The following code should be present in `stock-list.ts`:

```
// /app/pages/stock-list/stock-list.ts
import {Component} from '@angular/core';
import {NavController, Modal, Events} from 'ionic-angular';
import {StockCmp} from '../../component/stock/stock';
import {StockChart} from '../../component/stock-chart/stock-chart';
import {QuoteSearch} from '../quote-search/quote-search';
import {Detail} from '../detail/detail';
import {StorageProvider} from '../../providers/storage';

@Component({
  templateUrl: 'build/pages/stock-list/stock-list.html',
  directives: [StockCmp, StockChart]
})
export class StockList {
  quoteList = [];
  run:Boolean;
  constructor(public nav:NavController, public storageProvider:
StorageProvider,
   public events: Events) {
    this.storageProvider.getAllQuotes()
    .then(data => {
      let result = data.res.rows;
      for(let i=0; i < result.length; i++) {
       this.quoteList.push(JSON.parse(result[i].value));
      }
```

```
        });
        this.events.subscribe("stock:watch", (stock) => {
          this.quoteList.push(stock[0]);
        });
       }
      searchQuote() {
        let modal = Modal.create(QuoteSearch);
        this.nav.present(modal);
      }
      deleteItem(index, symbol) {
        this.storageProvider.removeQuote(symbol)
        .then(()=>{
          this.quoteList.splice(index, 1);
        });
      }
      ionViewWillEnter() {
       this.run = true;
      }
      ionViewWillLeave() {
        this.run = false;
      }
      openDetail(quote) {
        this.nav.push(Detail, {quote:quote});
      }
    }
```

Let's understand the preceding code:

- `constructor()` gets a list of all quotes that our user watches and assigns the list to `this.quoteList`. We are also watching the `stock:watch` event, which is fired when a user watches new stock. We are also adding that stock in our `quoteList`.
- `searchQuote()` opens a new `QuoteSearch` Ionic modal.
- `deleteItem(index, symbol)` deletes the stock at a given index from the user's watch list.
- `ionViewWillEnter()` is a page life cycle hook that is provided by Ionic. It is fired when the user is about to enter the page. We are changing `this.run` to `true`, which sets the input to our stock component. In other words, our stock component will get stock information only when the page is active.
- `ionViewWillLeave()` is another page life cycle hook, which is fired when the user is about to leave the page. It sets `this.run` to `false`.
- `openDetail(quote)` opens the `Detail` page for a given symbol.

StockList template

The following code should be present in `stock-list.html`:

```
<!-- /app/pages/stock-list/stock-list.html -->
<ion-header>
  <ion-navbar>
    <ion-title>Stocks</ion-title>
    <ion-buttons end>
      <button (click)="searchQuote()">
        <ion-icon name="add"></ion-icon>
      </button>
    </ion-buttons>
  </ion-navbar>
</ion-header>

<ion-content padding class="page1">
  <ion-list>
    <stock (click)="openDetail(quote)" *ngFor="let quote of quoteList; let
i =
     index" [run]="run" sliding="true" [info]="quote"
(delete)="deleteItem(i,
     quote.symbol)">
    </stock>
  </ion-list>
</ion-content>
```

In `ion-navbar`, clicking on the button with the add icon opens the `QuoteSearch` modal.

In `ion-content`, we have `ion-list` with stock components. It is worth noting that stock components require `ion-list` as a parent component.

Defining the QuoteSearch modal

The `QuoteSearch` modalallow us to search for the symbol based on company name.

QuoteSearch controller

The following code should be present in `search.ts`:

```
// /app/pages/quote-search/quote-search.ts
import {Component} from '@angular/core';
import {Control} from '@angular/common';
import {Page, ViewController, Events} from 'ionic-angular';
import {StockInfo} from '../../providers/stock-info';
import {StorageProvider} from '../../providers/storage';
```

```
@Component({
    templateUrl: 'build/pages/quote-search/quote-search.html'
})
export class QuoteSearch {
    searchbar = new Control();
    quoteList:Array<any>;
    constructor(public stockInfo: StockInfo, public vc:ViewController,
public
     storage: StorageProvider, public events:Events) {
        this.searchbar.valueChanges
        .filter(value => value.trim().length > 2)
        .distinctUntilChanged()
        .debounceTime(2000)
        .subscribe(value => {
            this.search(value);
        });
    }
    search(value) {
        this.stockInfo.getQuotes(value)
        .subscribe(list => {
            this.quoteList = list;
        });
    }
    watch(quote) {
        this.events.publish("stock:watch", quote);
        this.storage.setQuote(quote.symbol, quote);
    }
    close() {
        this.vc.dismiss();
    }
}
```

Let's understand the preceding code:

- `constructor()` observes the search bar's values using the `valueChanges` variable of `Observable`. We filter the value less than three and also filtering distinct values; we are debouncing for two seconds and then calling the `search` function.
- `search(value)` gets symbol details such as company name, symbol, and so on, based on a given value.
- `watch(quote)` fires the `stock:watch` event with a given quote, and also stores the quote information in the database.
- `close()` closes the modal.

QuoteSearch template

The following code should be present in `quote-search.html`:

```
<!-- /app/pages/quote-search/quote-search.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Quote Search</ion-title>
        <ion-buttons end>
            <button (click)="close()"><ion-icon name="close"></ion-icon>
      </button>
        </ion-buttons>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-searchbar [ngFormControl]="searchbar">
    </ion-searchbar>
    <ion-list>
        <ion-item *ngFor="let quote of quoteList">
            <ion-row>
              <h3>{{quote.symbol}}</h3>
            </ion-row>
            <ion-row center width-25>
              <p>{{quote.name}}</p>
            </ion-row>
            <button item-right danger (click)="watch(quote)">Watch</button>
        </ion-item>
    </ion-list>
</ion-content>
```

We have a close button in the `ion-navbar` tag, an `ion-searchbar` in content, and `ion-list`, which shows a list of symbols returned from the API. Each `ion-item` has a symbol and company name, along with the watch button.

Defining the Detail page

The `Detail` page is for showing the stock ticker, historical data in chart form, and detailed stock information, and for adding user notes.

Detail page controller

The following should be present in `detail.ts`:

```
// /app/pages/detail/detail.ts
import {Component} from '@angular/core';
import {NavParams} from 'ionic-angular';
import {StockInfo} from '../../providers/stock-info';
import {StockChart} from '../../component/stock-chart/stock-chart';
import {StockCmp} from '../../component/stock/stock';
import {StockDetailCmp} from '../../component/stock-detail/stock-detail';
import {StorageProvider} from '../../providers/storage';

@Component({
    templateUrl: 'build/pages/detail/detail.html',
    directives:[StockChart, StockCmp, StockDetailCmp]
})
export class Detail {
    notes = [];
    month:number = 1;
    details:{};
    quote:any;
    data:any;
    constructor(params: NavParams, public stockinfo:StockInfo, public
     storageProvider: StorageProvider) {
        this.quote = params.get('quote');
        this.getChartData();
        this.getNotes();
    }
    addNote(noteElm) {
        if(noteElm.value) {
            this.storageProvider.setNotes(this.quote.symbol, noteElm.value)
            .then( (data)=> {
                this.notes.push({rowid: data.res.insertId,
value:noteElm.value,
                 symbol:this.quote.symbol});
                noteElm.value = "";
            })
        }
    }
    deleteNote(rowid,i) {
        this.storageProvider.removeNote(rowid);
        this.notes.splice(i,1);
    }
    getChartData() {
            this.stockinfo.getChart(this.quote.symbol, this.month)
            .subscribe(data=> {
            let newData:any = {};
            newData.series = [];
            newData.series.push({meta: 'Low', name: 'Low', data:[]});
            newData.series.push({meta: 'Close',name: 'Close', data:[]});
            data.forEach(day=> {
                newData.series[0].data.unshift(Number.parseFloat(day.Low));
newData.series[1].data.unshift(Number.parseFloat(day.Close));
            });
                this.data = newData;
            });
    }
    getNotes() {
         this.storageProvider.getAllNotes(this.quote.symbol)
         .then(data => {
           let set = data.res.rows;
           this.notes = Array.from(set);
        });
    }
}
```

Let's understand the preceding code:

- `constructor()` gets a quote from `NavParams` and calls `this.getChartData` and `this.getNotes`, which basically gets the historical data and user notes for a given symbol provided using `this.quote`.
- `addNote(noteElm)` is fired when a user clicks **Add Note** button. It save the note in the current notes array, as well as in the notes table inside database.
- `deleteNote(rowid, i)` deletes a note from the database using `rowid`, and from the current notes array.
- `getChartData()` gets historical data from the `StockInfo` provider and assigns all that data to `this.data`. It is worth noting that we are creating an object that has a key named `series`, which is of array type `series[0].data` and is an array of `Low` stock value, and `series[1].data` has array of high stock value.
- `getNotes()` gets user notes for a given symbol and provides the user notes for that stock.

Detail page template

The following code should be present in `detail.html`:

```
<!-- /app/pages/detail/detail.html -->
<ion-header>
    <ion-navbar>
        <ion-title>Stock Detail</ion-title>
    </ion-navbar>
```

```
</ion-header>

<ion-content class="detail">
    <ion-card>
        <ion-list>
            <stock [info]="quote" sliding="false"></stock>
        </ion-list>
    </ion-card>
    <ion-card>
        <ion-item>Historical Data</ion-item>
       <ion-list>
        <ion-item>
            <ion-label>Select No. of Months of Data</ion-label>
            <ion-select [(ngModel)]="month">
                <ion-option value=1>1 Month</ion-option>
                <ion-option value=3>3 Month</ion-option>
                <ion-option value=6>6 Month</ion-option>
                <ion-option value=12>12 Month</ion-option>
            </ion-select>
            </ion-item>
        </ion-list>
        <ion-item>
            <button (click)="getChartData()">Update Chart</button>
        </ion-item>
        <ion-item>
            <ion-label class="ct-series-a"><hr/>Low</ion-label>
            <ion-label class="ct-series-b"><hr/> Close</ion-label>
        </ion-item>
        <stock-chart class="ct-perfect-fourth" [data]="data"></stock-chart>
    </ion-card>
    <stock-detail [symbol]="quote.symbol"></stock-detail>
    <ion-card text-wrap>
        <ion-item>
            <ion-label stacked>Enter Notes</ion-label>
            <ion-input type="text" #newNote></ion-input>
        </ion-item>
        <div padding>
            <button full (click)="addNote(newNote)">Add Note</button>
        </div>
        <ion-item *ngFor="let note of notes;let i = index;">
            {{note.value}}
            <button item-right danger (click)="deleteNote(note.rowid,
             i)">Delete</button>
        </ion-item>
    </ion-card>
</ion-content>
```

In the `Detail` page, we have a stock component, which gets stock information every five seconds. It is interesting to know that we have reused our stock component in this page too, without writing a single extra line for it. This is the benefit of component-based programming.

Then we have a historical data chart, which shows the historical data for one month in chart form. Users can change the number of months of historical data using `ion-select`.

Then we have the details of the symbol, using the `stock-detail` component.

Last, we have an `ion-input` tag and the **Add Note** button for adding new notes and a list of notes for a particular symbol.

Adding core styles

We need to import individual stylesheets and some styles in `/app/themes/app.core.scss`:

```
@import '../component/stock/stock';
@import '../component/stock-detail/stock-detail';

.ct-series-a {
    color:#d70206;
}

.ct-series-b {
    color: #f05b4f;
}
```

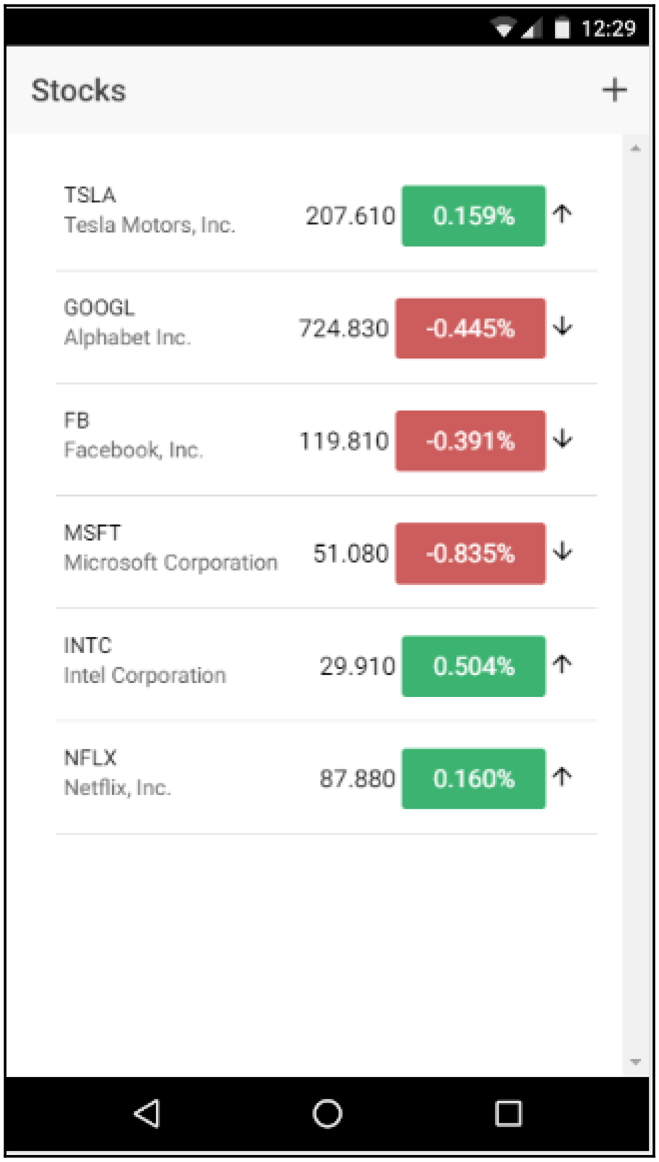
Running our app

Let's run our application on an actual device:

```
ionic platform add android
ionic run android
```

App screenshots

The `StockList` page will look like this:

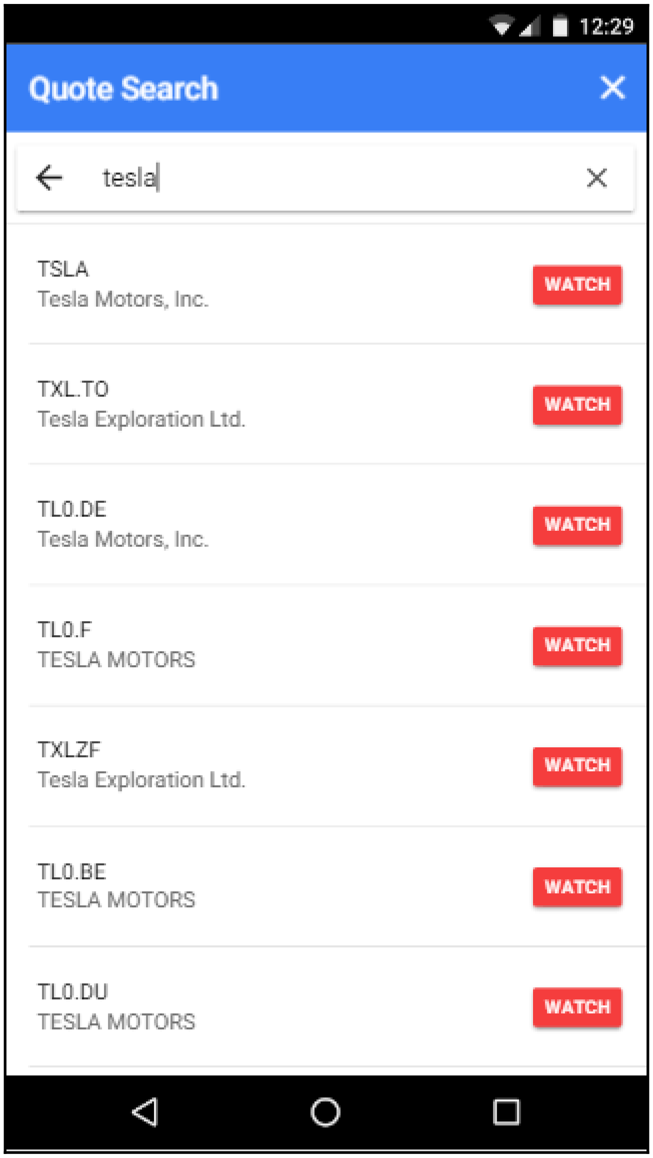


12:29

Stocks +

TSLA Tesla Motors, Inc.	207.610	0.159%	↑
GOOGL Alphabet Inc.	724.830	-0.445%	↓
FB Facebook, Inc.	119.810	-0.391%	↓
MSFT Microsoft Corporation	51.080	-0.835%	↓
INTC Intel Corporation	29.910	0.504%	↑
NFLX Netflix, Inc.	87.880	0.160%	↑

The `QuoteSearch` modal will look like this:



TSLA
Tesla Motors, Inc.
WATCH

TXL.TO
Tesla Exploration Ltd.
WATCH

TL0.DE
Tesla Motors, Inc.
WATCH

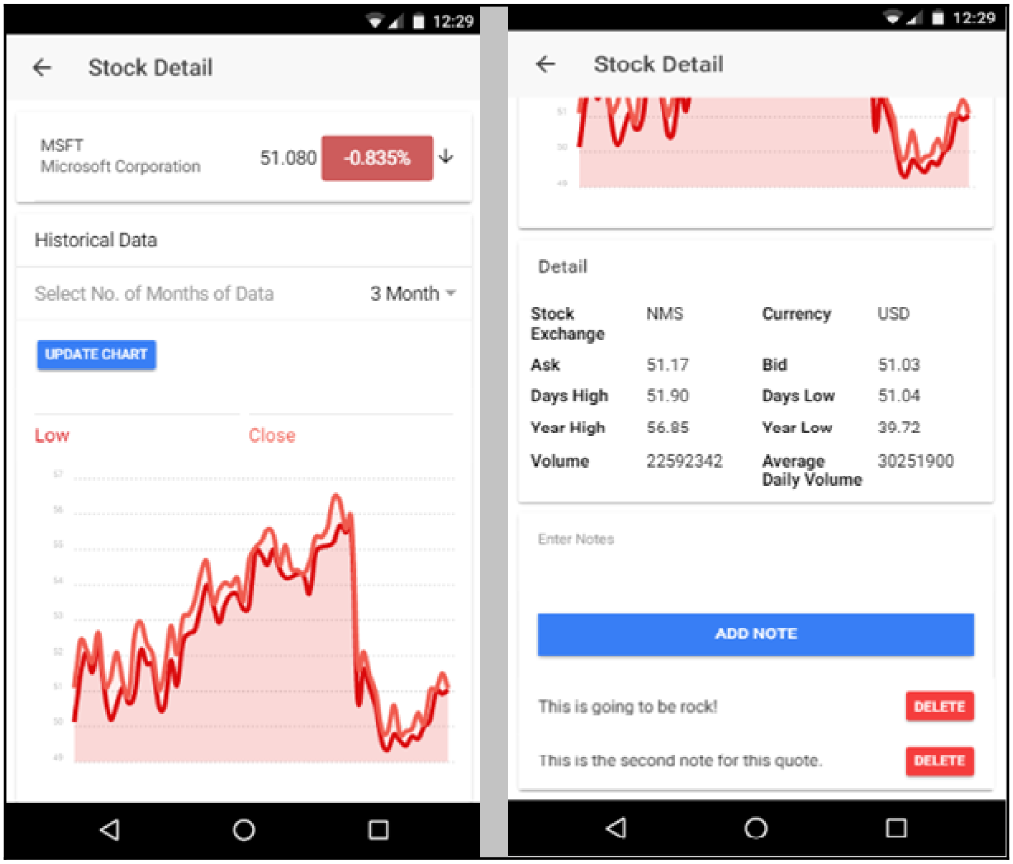
TL0.F
TESLA MOTORS
WATCH

TXLZF
Tesla Exploration Ltd.
WATCH

TL0.BE
TESLA MOTORS
WATCH

TL0.DU
TESLA MOTORS
WATCH

The `Detail` page will look like this:



MSFT
Microsoft Corporation
51.080 -0.835%

Historical Data

Select No. of Months of Data: 3 Month

Detail

Stock Exchange	NMS	Currency	USD
Ask	51.17	Bid	51.03
Days High	51.90	Days Low	51.04
Year High	56.85	Year Low	39.72
Volume	22592342	Average Daily Volume	30251900

Enter Notes

This is going to be rock!

This is the second note for this quote.

Summary

In this chapter, we have created a StockMarket app, which uses RxJS's `Observable` and `Chartist.js` for creating charts using Yahoo Finance API and YQL. Along with it, we have used component-based programming, which allows us to reuse the component.

In the next chapter, we will build a WordPress client for mobile. We will have posts with comments, pages, categories, and other features, such as push notifications.

5
WordPress Client App

WordPress client app is a mobile app for WordPress site owners and bloggers. WordPress is one of the most famous applications created for the web. WordPress powers nearly 24% of the world's websites. So, having a mobile app for your WordPress website seems like a cool idea!

In the process of developing this application, we will use lots of new Ionic 2 functionalities such as **toast** and **infinite scroll**. When the user opens the app, it will show a tab bar with four tabs: `Posts`, `Categories`, `Pages`, and `Favorites`. We will also utilize component-based programming and reactive programming to make our life easier.

The best part of this app is that you can use it with any number of WordPress sites. You just need to change a bunch of variables to do that. Cheers!

In the previous chapter, we created a Stock Market app. In this chapter, we will create a WordPress client, which we can use as a blog or newspaper app. The app will show a list of posts, a list of pages and their contents, a favorites list, push notifications, and lots of other stuff.

We will learn the following things:

- WordPress REST API
- Ionic Toast
- Infinite scrolling in Ionic
- Push notification in the WordPress app
- Component-based programming
- **Google Analytics** and **Google Cloud Messaging** (**GCM**) setup

Backend

In this section, we are going to discuss various backend items required for our WordPress client app. We will be doing the following stuff:

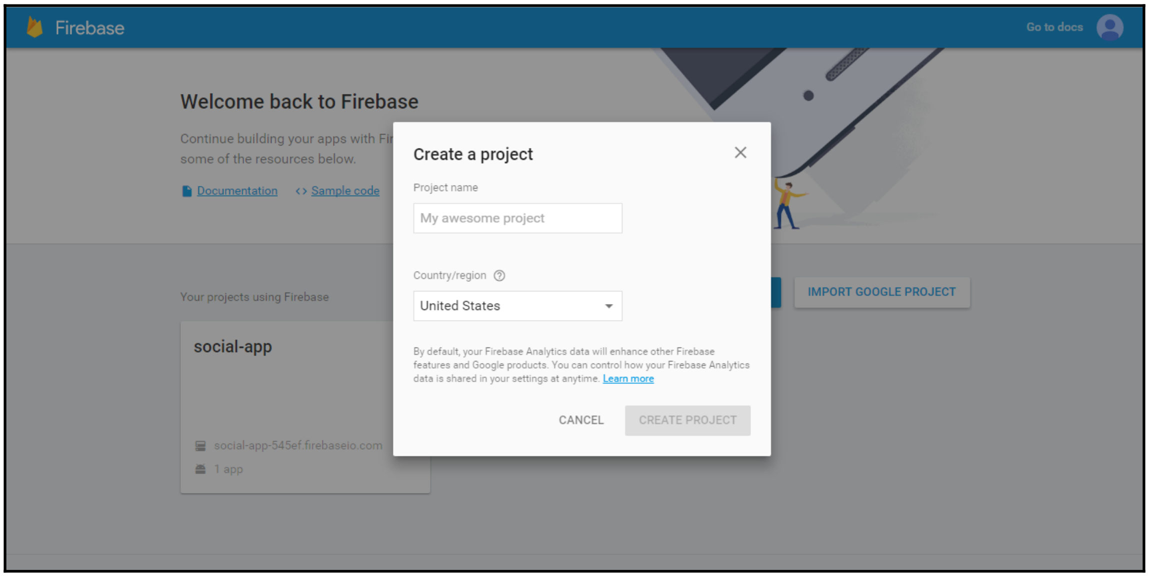
- Configuring **Firebase Cloud Messaging** (**FCM**) for push notification
- Creating a project in Google Analytics
- Configuring WordPress

Push notifications

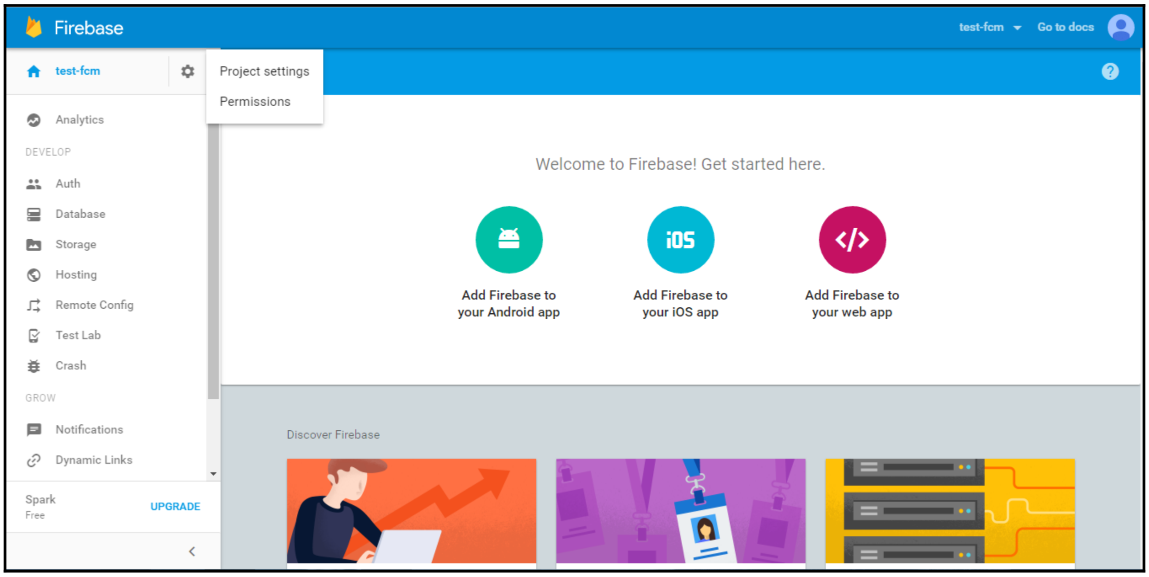
In order to use push notifications with WordPress, we are using the WordPress **Push Notification Lite** plugin and the **phonegap push plugin**, which will allow us to send push notifications when we publish a post.

For Android, we will be using **Firebase Cloud Messaging** (**FCM**), which is an updated version of GCM. To configure FCM, take the following steps:

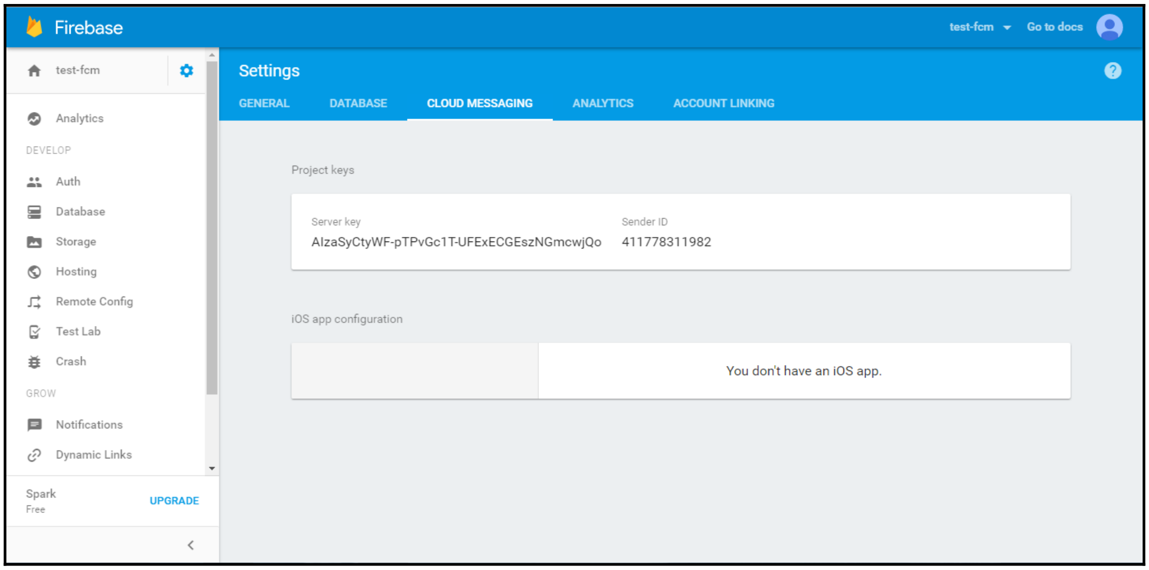
1. Open your Firebase Console at `https://console.firebase.google.com` and click on **CREATE NEW PROJECT**.
2. Then, enter the **Project name,** select your **Country/region,** and click on **CREATE PROJECT**, as shown in the following screenshot:


Firebase
Go to docs
Welcome back to Firebase
Continue building your apps with Fi
some of the resources below.
Documentation
Sample code
Your projects using Firebase
social-app
social-app-545ef.firebaseio.com
1 app
IMPORT GOOGLE PROJECT
Create a project
Project name
My awesome project
Country/region
United States
By default, your Firebase Analytics data will enhance other Firebase features and Google products. You can control how your Firebase Analytics data is shared in your settings at anytime. Learn more
CANCEL
CREATE PROJECT

3. Now, click on **Project settings**, as shown in the following screenshot. This will take you to the Settings page:



4. Now click on the **CLOUD MESSAGING** tab. You will see **Server Key** and **Sender ID**, as shown in the screenshot. We require both of these values to make our push notifications work:



Server key
AIzaSyCtyWF-pTPvGc1T-UFExECGEszNGmcwjQo

Sender ID
411778311982

You don't have an iOS app.

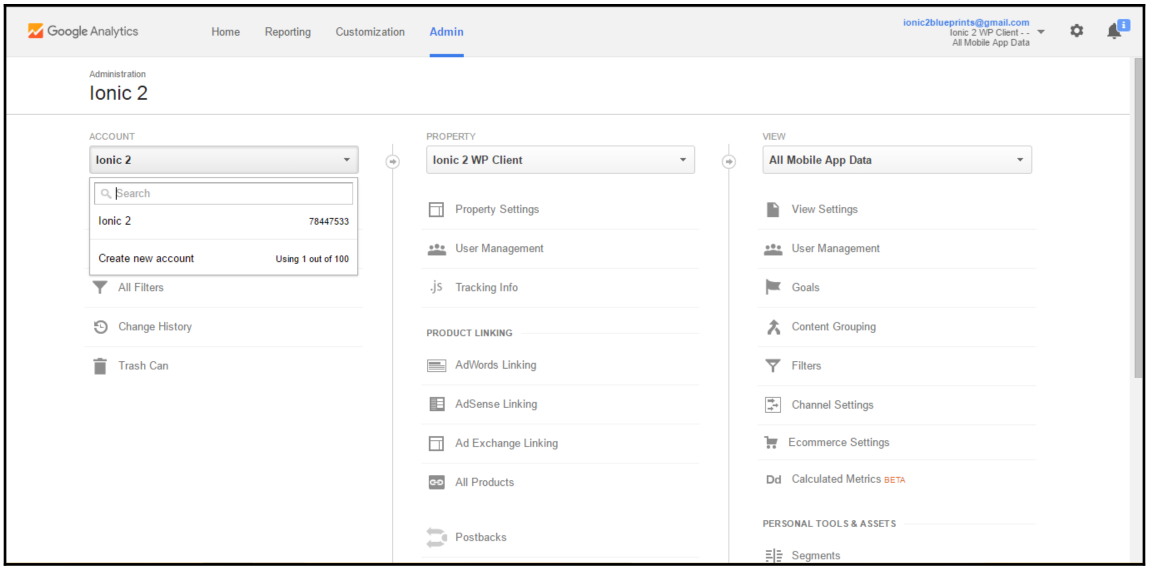
For iOS, we need an SSL certificate for push notifications from the dev center. Explaining the whole procedure here is outside the scope of this book. You can read the instructions at `http://www.delitestudio.com/WordPress/push-notifications-for-WordPress/configuring-ios-push-notifications/`.

Google Analytics

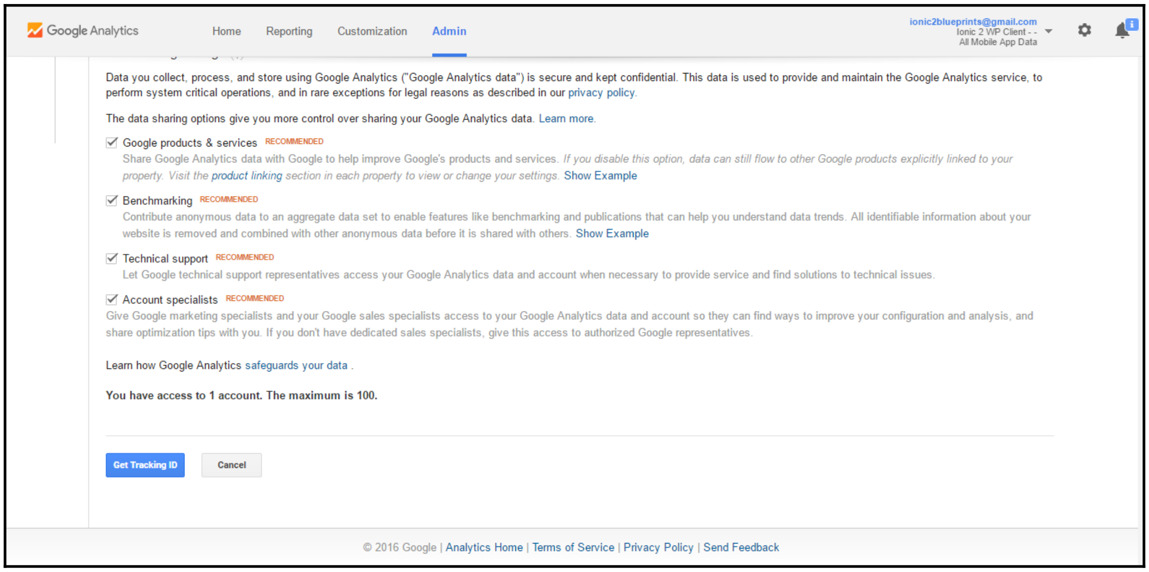
You need to have a Google Analytics account to use it. Just a reminder!

For Google Analytics to work, we need a tracking ID for our app. To configure it, take the following steps:

1. Open your Analytics account and go to the **Admin** tab.
2. Then, create an account by clicking on **Create new account**:



3. A new page will open with a form to fill in. After completing it click on **Get Tracking ID**, you will get a tracking ID; which we require for our application:



Data you collect, process, and store using Google Analytics ("Google Analytics data") is secure and kept confidential. This data is used to provide and maintain the Google Analytics service, to perform system critical operations, and in rare exceptions for legal reasons as described in our privacy policy.

The data sharing options give you more control over sharing your Google Analytics data. Learn more.

✓ Google products & services RECOMMENDED
Share Google Analytics data with Google to help improve Google's products and services. *If you disable this option, data can still flow to other Google products explicitly linked to your property. Visit the product linking section in each property to view or change your settings.* Show Example

✓ Benchmarking RECOMMENDED
Contribute anonymous data to an aggregate data set to enable features like benchmarking and publications that can help you understand data trends. All identifiable information about your website is removed and combined with other anonymous data before it is shared with others. Show Example

✓ Technical support RECOMMENDED
Let Google technical support representatives access your Google Analytics data and account when necessary to provide service and find solutions to technical issues.

✓ Account specialists RECOMMENDED
Give Google marketing specialists and your Google sales specialists access to your Google Analytics data and account so they can find ways to improve your configuration and analysis, and share optimization tips with you. If you don't have dedicated sales specialists, give this access to authorized Google representatives.

Learn how Google Analytics safeguards your data .

You have access to 1 account. The maximum is 100.

Get Tracking ID
Cancel
© 2016 Google | Analytics Home | Terms of Service | Privacy Policy | Send Feedback

WordPress JSON API

For quite a long time, WordPress lacked a REST API. Our prayers were answered with a special project: the mighty WP-REST API. It was created as a WordPress plugin, which creates REST API endpoints for a WordPress site.

The API actually exposes an easy interface to get posts, comments, users, pages, media, and lots of other stuff.

An important thing to remember is that the REST API is located at `${siteUrl}/wp-json/wp/v2`, where `${siteUrl}` will be the actual URL of your WordPress site.

You want to get a post from your site, right? Send a GET Request to `http://${siteUrl}/wp-json/wp/v2/` and you will get list of posts.

For example, let's assume that `https://dummy.com` is the site URL. You can get posts through a GET request to `https://dummy.com/wp-json/wp/v2/posts`.

We will be using this API in order to get posts, pages, and more.

See more details of the API at `http://v2.wp-api.org/`.

Configuring WordPress

Before we actually do anything, we need to have a WordPress installation on our server. So, go ahead and install WordPress on any hosting site. Make sure it is live on the Internet. If you don't have a hosting, go to `http://openshift.redhat.com` to host a WordPress site.

Installing plugins

We need to install two plugins for our app. Install the following two plugins:

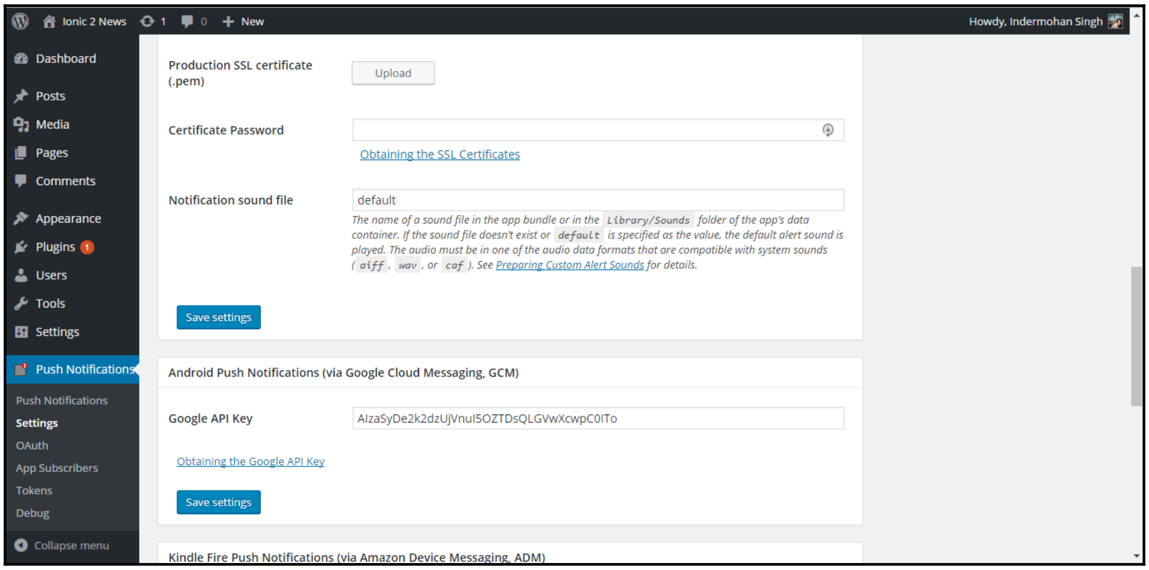
- WP REST API
- WordPress Push Notification Lite

Configuring Push Notification

In the WordPress side menu, click on **Push Notification** | **Settings**.

For iOS, upload the production SSL certificate (`.pem`), fill in the password, and click **Save Settings**.

For Android, complete the Google API keyfield with the server key that we mentioned earlier and click **Save settings**:



Production SSL certificate (.pem)

Certificate Password

Obtaining the SSL Certificates

Notification sound file: default

The name of a sound file in the app bundle or in the `Library/Sounds` folder of the app's data container. If the sound file doesn't exist or `default` is specified as the value, the default alert sound is played. The audio must be in one of the audio data formats that are compatible with system sounds (`aiff`, `wav`, or `caf`). See Preparing Custom Alert Sounds for details.

Android Push Notifications (via Google Cloud Messaging, GCM)

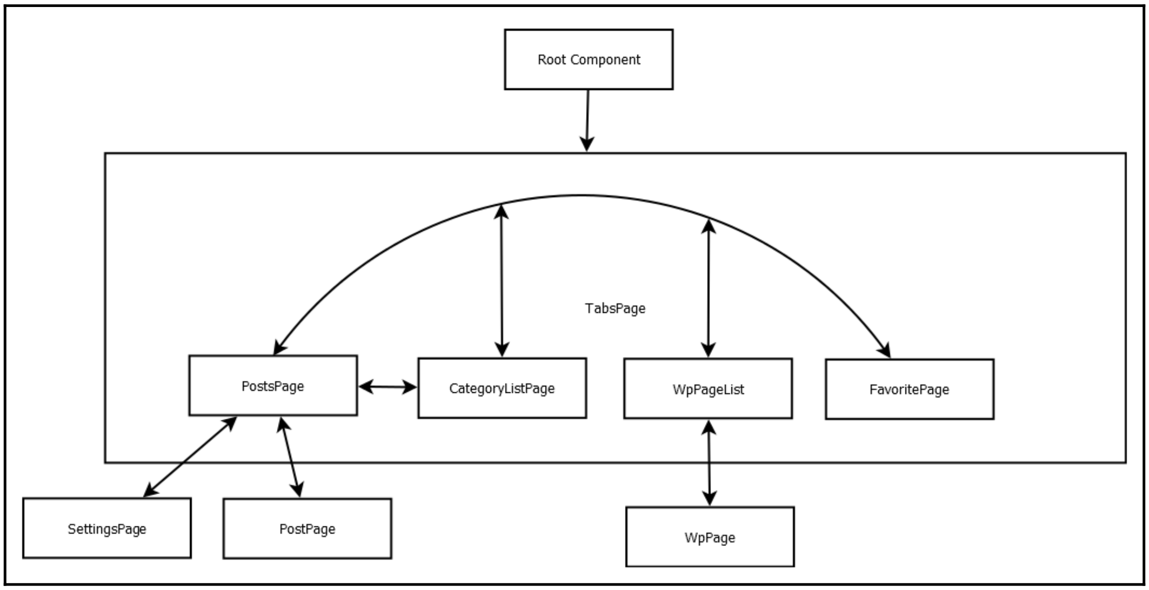
Google API Key: AIzaSyDe2k2dzUjVnuI5OZTDsQLGVwXcwpC0ITo

Obtaining the Google API Key

Kindle Fire Push Notifications (via Amazon Device Messaging, ADM)

App flow

The following figure shows how the control will flow inside our application:



Let's understand the flow:

- **RootComponent**: This is the root Ionic appcomponent. It is defined inside the `/app/app.ts` file.
- **TabsPage**: This acts as a container for our four tabs: posts, categories, pages, and favorites.
- **PostsPage**: This lists all posts for our WordPress site and it is the root page of our posts tab.
- **PostPage**: This shows a single post in detail, with content and comments.
- **CategoryListPage**: This shows the posts category of our WordPress site. Clicking on any category takes the user to PostsPage. But in this flow, only posts of that particular category will be shown.
- **WpPageList**: This shows a list of WordPress pages.
- **WpPage**: This shows the content of a particular WordPress page.
- **FavoritePage**: This shows a list of posts that are saved as favorites by the user.
- **SettingsPage**: This shows settings such as Push Notification, toggle and sorting of posts, and many more.

Scaffolding and setting up the app

We will start by scaffolding a blank application. Run the following command:

```
ionic start ionic2-wp-client blank --v2 --ts
```

Notice the `--v2` and `--ts` tags for scaffolding the Ionic 2 app based on **TypeScript**.

Before we start writing code for our app, we need to install some project-specific dependencies.

Installing Cordova plugins

Following are some Cordova plugins that we need to install:

```
ionic plugin add phonegap-plugin-push --variable SENDER_ID="XXXXXXX"
ionic plugin add cordova-plugin-google-analytics
ionic plugin add cordova-plugin-network-information
ionic plugin add cordova-plugin-x-socialsharing
```

It is worth noting that Ionic installs some plugins automatically, such as `StatusBar`, `Device`, and `WhiteList`. We are also using these plugins. Also, you need to replace the `XXXXXXX` with your `SENDER_ID` from Firebase Developer Console.

CORS issue

While developing this application, you might end up running into a CORS issue. If you are using Google Chrome, just install this CORS plugin from `https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi?hl=en`.

The coding part

Now that we have everything set up to start working on our application, let's define the followings things:

- App configuration
- Our main app component
- The components of our app
- Providers/services for various functionalities
- Ionic pages for various views

App configuration

We need to configure our app. We will create a file that exports some constants required for configuration, for instance, the URL of our WordPress site and Push Notification sender ID.

Create a `constants.ts` inside `/app/providers` and create the required constants as follows:

```
// /app/providers/constants.ts
export const SITE_URL:string = "WordPress_Site_URL";
export const GOOGLE_ANALYTICS_ID:string ="TRACKING_ID";
export const GCM_SENDER_ID:string = "SENDER_ID";
```

Replace `WordPress_Site_URL` with the URL of your WordPress site and replace `Tracking_ID` with your Analytics `Tracking_ID`; similarly, replace `SENDER_ID`.

Defining app.ts

This is the root of our application and it defines the root component using the `@App` decorator. This is where we inject all of our dependencies. The following code should be present in `app.ts`:

```
/* /app/app.ts */
import {ViewChild, Component} from '@angular/core';
import {Platform,Nav, Storage, LocalStorage, ionicBootstrap} from 'ionic-
angular';
import {StatusBar, Network, GoogleAnalytics} from 'ionic-native';
import {TabsPage} from './pages/tabs/tabs';
import {CategoryListPage} from './pages/category-list/category-list';
import {WpProvider} from './providers/wp-provider/wp-provider';
import {PushProvider} from './providers/push-provider/push-provider';
import {UtilProvider} from './providers/util-provider/util-provider';
import {GOOGLE_ANALYTICS_ID} from './providers/constants';

@Component({
  templateUrl: 'build/app.html',
})
class MyApp {
  @ViewChild(Nav) nav: Nav;
  rootPage: any = TabsPage;
  storage:Storage = new Storage(LocalStorage);
  settings = {};
  constructor(private platform: Platform, private push: PushProvider,
   public up: UtilProvider) {
    this.initializeApp();
  }

  initializeApp() {
    this.platform.ready().then(() => {
      let toast = this.up.getToast("You are not connected to
       Internet.");
      let disconnectSubscription = Network.onDisconnect().subscribe(()
       => {
        this.nav.present(toast);
      });
      GoogleAnalytics.startTrackerWithId(GOOGLE_ANALYTICS_ID);
      this.storage.get('settings')
      .then(data => {
        if(data === null) {
          let settings = {push: true, sort: 'desc'};
          this.storage.set('settings', JSON.stringify(settings));
        }
      });
      StatusBar.styleDefault();
      if(this.platform.is('mobile')) {
        this.push.init();
      }
    });
  }
}
ionicBootstrap (MyApp, [WpProvider,PushProvider, UtilProvider]);
```

Let's understand the preceding code:

- `constructor()`: We create a storage variable to use `LocalStorage` and then call `initializeApp()`.
- `initializeApp()`: In this, we check if the platform is ready and then we start Google Analytics tracker with our `Tracker_ID`. Along with that, we also create a disconnect subscription, which creates a `toast` message when the user goes offline. We also initialize the settings of our app, store them in `LocalStorage` , and finally, initialize the Push Notification service.
- `ionicBootstrap`: We bootstrapped our Ionic 2 app and also injected our array of providers in ionicBootstrap.
- The following code should be present in `app.html`:

```
      <!--  Template app/app.html -->
      <ion-nav id="nav" [root]="rootPage" #content
swipeBackEnabled="false"></ion-
       nav>
```

Providers for our application

We will have the followingproviders:

- `UtilProvider`
- `WpProvider`
- `PushProvider`

Defining UtilProvider

`UtilProvider` basically abstracts the `Toast` and `Loading` API for us. We are using `Toast` and `Loading` heavily, so we abstracted their creation. The following code should be present in `util-provider.ts`:

```
/* /app/providers/util-provider/util-provider.ts */
import {Injectable} from '@angular/core';
import {Loading, Toast} from 'ionic-angular';

@Injectable()
export class UtilProvider {
    getLoader(content) {
        let loading = Loading.create({
            content: content,
            duration: 3000
        });
        return loading;
    }
    getToast(message) {
        let toast = Toast.create({
            message: message,
            duration: 2000
        });
        return toast;
    }
}
```

Let's understand the preceding code:

- `getLoader(content)`: The `getLoader` function returns a loader message using a `content` argument.
- `getToast(content)`: This returns a `Toast`. The message displayed in `Toast` is given by the user through the `content` argument.

Defining WpProvider

`WpProvider` is used to interact with the WordPress backend using the WP REST API. We will fetch posts, pages, media, and lots of other stuff. The following should be present in `wp-provider.ts`:

```
    // /app/providers/wp-providder/wp-provider.ts
    import {Injectable} from '@angular/core';
    import {Http} from '@angular/http';
    import {Observable} from 'rxjs/Rx';
    import 'rxjs/add/operator/map';
    import {SITE_URL} from '../constants';

    @Injectable()
    export class WpProvider {
      apiURL:string = SITE_URL + '/wp-json/wp/v2';
      constructor(public http:Http) {}
      getPosts(query) {
        query = this.transformRequest(query);
        let url = this.apiURL + `/posts?` + query + '&_embed';
        return this.http.get(url)
          .map(data => {
            let posts = data.json();
            posts.forEach(post => {
              if(post.featured_media) {
                post.featuredMedia = this.getMedia(post.featured_media);
              }
            });
            return posts;
          });
      }
      getMedia(id:number) {
        return this.http.get(this.apiURL + `/media/${id}`).map(data =>
         data.json());
      }
      getPages() {
        return this.http.get(this.apiURL + '/pages').map(data =>
         data.json());
      }
      getCategories() {
        return this.http.get(this.apiURL + '/categories').map(data =>
         data.json());
      }
      transformRequest(obj) {
        let p, str;
        str = [];
        for (p in obj) {
            str.push(encodeURIComponent(p) + '=' +
             encodeURIComponent(obj[p]));
        }
        return str.join('&');
      }

    }
```

Let's Understand the preceding code:

- `constructor()`: We have initialized our `apiURL` to the WP REST API endpoint.
- `getPosts(query)`: This fetches the posts from WordPress, according to the given query, and returns the `Observable` to the function caller. It also fetches the required media for each post through `getMedia()`.
- `getMedia(id)`: This fetches the media according to the given `id` and returns an `Observable`.
- `getPages()`: This fetches all the `pages` of the WordPress site and returns the `observable`.

- `getCategories()`: This fetches all the `Categories` and returns the `Observable`.
- `transformRequest(obj)`: This transforms the request given as an object and returns a request in the form of a string. For example, here, we have supplied the following object as input:

  ```
  {page: 2, order: "asc"}
  ```

  It will return a string, as follows:

  ```
  page=2&order=asc
  ```

Defining PushProvider

`PushProvider` helps us to use Push Notification in our application. It allows us to register and unregister the mobile device to the WordPress site. The following code should be present in `push-provider.ts`:

```
// /app/providers/push-provider/push-provider.ts
import {Injectable} from '@angular/core';
import {Http, Headers} from '@angular/http';
import {Observable} from 'rxjs/Rx';
import 'rxjs/add/operator/map';
import {Events, LocalStorage, Storage} from 'ionic-angular';
import {Push, Device} from 'ionic-native';
import {SITE_URL, GCM_SENDER_ID} from '../constants';

@Injectable()
export class PushProvider {
  apiURL:string = SITE_URL;
  storage = new Storage(LocalStorage);
  push:any;
  constructor(public http:Http, public events: Events) {}
  init() {
     this.push = Push.init({
        android: { senderID: GCM_SENDER_ID },
        ios: {
            alert: "true",
            badge: true,
            sound: 'false'
        },
        windows: {}
     });
     this.push.on('registration', (data) => {
         this.storage.set('token', data.registrationId);
```

```
            this.registerDevice()
            .then(data => {});
        });
        this.push.on('notification', (data) => {
            console.log(data);
        });
    }
    transformRequest(obj) {
      let p, str;
      str = [];
      for (p in obj) {
          str.push(encodeURIComponent(p) + '=' +
           encodeURIComponent(obj[p]));
      }
      return str.join('&');
    }
    registerDevice() {
      let url = this.apiURL + '/pnfw/register';
      let os = Device.device.platform;
      return this.storage.get('token')
      .then(data => {
           let request = this.transformRequest({token: data, os: os});
           let headers = new Headers({'Content-Type':'application/x-www-
            form-urlencoded'});
           return this.http.post(url, request,
            {headers:headers}).toPromise();
      });
    }
    unregisterDevice() {
        let url = this.apiURL + '/pnfw/unregister';
        let os = Device.device.platform;
        return this.storage.get('token')
        .then(data => {
            let request = this.transformRequest({token: data, os: os});
            let headers = new Headers({'Content-Type':'application/x-www-
             form-urlencoded'});
            return this.http.post(url, request,
             {headers:headers}).toPromise();
        });
    }
  }
```

Let's understand the preceding code:

- `constructor()`: We have initialized our `apiURL` according to our WordPress Push Notification plugin's endpoint.
- `init()`: In the `init` function, we initialize our push plugin and create an event handler for `registration` and `notification` events. When a user registers, we save the `registrationId` to `LocalStorage` and this allows us to send this ID to the WordPress site using `this.registerDevice`
- `transformRequest(obj)`: This does the same thing as it does in `WpProvider`.
- `registerDevice()`: This registers our device token to the WordPress site.
- `unregisterDevice()`: This unregisters our `Device` from the WordPress site.

You can read more about the plugin's API at `http://www.delitestudio.com/WordPress/push-notifications-for-WordPress/documentation`.

HtmlPipe

`HtmlPipe` pipe basically either excludes the link from an HTML template, or changes it. If `args` is `delete`, then it will delete the first link from the given string. If `args` is equal to `change`, then it will add `target="_system"` to the link.

More importantly, why do we need this? Because if we add any link inside our blog posts, it will be open inside our app's **WebView**, which we don't want. So, we add `target="_system"` to open those links in the system's browser. The following code should be present in `htmlPipe.ts`:

```
// /app/pipes/htmlPipe.ts
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
    name: 'htmlPipe'
})
export class HtmlPipe implements PipeTransform{
  transform(value:string, args) {
      if(args === "delete") {
        let test = value.match("<a");
        if(test) {
            value = value.substring(0,test.index-1);
        }
        return value;
      } else if (args === "change") {
          let test = value.replace("<a","<a target='_system'");
          return test;
      }
  }
}
```

Post component

The `Post` component is the only custom component in our app. It displays the post title, post date, author name and image, post excerpt, and three buttons named **Read**, **Favorite**, and **Share**.

Inputs

We have only one input, `postData`, which contains all the data of a post.

Outputs

The following are the outputs of our app:

- **read**: This emits post information, author information, the featured media, and comments when the **Read** button is clicked
- favorite: This emits the same data as read, but when the **Favorite** button is clicked

The following code should be in `posts.ts`:

```
// /app/components/posts/posts.ts
import {Component, Input,Output, EventEmitter, OnInit} from
'@angular/core';
import {Storage, LocalStorage} from 'ionic-angular';
import {GoogleAnalytics} from 'ionic-native';
import {HtmlPipe} from '../../pipes/htmlPipe';
import {WpProvider} from '../../providers/wp-provider/wp-provider';
import {SocialSharing} from 'ionic-native';

@Component({
    selector: 'post',
    templateUrl: 'build/components/post/post.html',
    pipes: [HtmlPipe]
})

export class PostCmp {
    @Input() postData;
```

```
        @Output() read = new EventEmitter();
        @Output() favorite = new EventEmitter();
        featuredMedia = {};
        authorData = {};
        comments = [];
        favoriteList = [];
        constructor(public wp:WpProvider) {
        }
        ngOnInit() {
          this.authorData = this.postData["_embedded"].author[0];
          this.postData.featuredMedia = this.postData.featuredMedia ||
           false;
          if(this.postData.featuredMedia) {
             this.postData.featuredMedia.subscribe(data => {
              this.featuredMedia = data;
             });
          }
          if(this.postData["_embedded"].replies) {
              this.comments = this.postData["_embedded"].replies[0];
          }
        }
        readBtn() {
           GoogleAnalytics.trackView(this.postData.link);
           this.read.emit({post: this.postData, author: this.authorData,
            media: this.featuredMedia, comments: this.comments});
        }
        favoriteBtn() {
            this.favorite.emit({post: this.postData, author:
             this.authorData, media: this.featuredMedia, comments:
             this.comments});
        }
        shareBtn() {
            let title = this.postData.title.rendered;
            let author = this.authorData['name'];
            let message = `Read this post on ${title} by ${author}.`;
            let url = this.postData.link;
            SocialSharing.share(message,"Read this post", null, url);
        }
    }
```

Let's understand the preceding code:

- `ngOnInit()`: In this, we set `this.authorData` to the information of the author. If there is featured media for this post, we set it to `this.featuredMedia`. If the post has any comments, we set it to `this.comments`.

- `readBtn()`: This is called when the user clicks on the **Read** button. We track this post in Google Analytics through our Google Analytics plugin and also output an object with post, author, media, and comment data.
- `favoriteBtn()`: This is called when the **Favorite** Button is clicked. It emits a favorite event.
- `shareBtn()`: This is called when the **Share** button is clicked. It allows the user to share the post to any other app on a mobile using the SocialSharing Cordova plugin.

The following code should be present in `post.html`:

```
<!-Template /app/components/post/post.html à
<ion-card cla"s="post-"mp">
    <img [sr"]="featuredMedia.source_"rl"
     *ng"f="featuredMedia.source_"rl">
    <ion-item cla"s="post-ti"le" text-wrap>
        <h1>{{postData.title.rendered}}</h1>
        <h4 cla"s="d"te">{{postData.date.substr(0,10)}}</h4>
    </ion-item>
    <ion-item>
        <ion-avatar item-left>
            <img [sr"]="authorData.avatar_urls["6]">
        </ion-avatar>
        <h2>{{authorData.name}}</h2>
    </ion-item>

    <ion-card-content>
        <p [innerHtm"]="postData.excerpt.rendered | htmlPip':
         'del'"e'"></p>
    </ion-card-content>

     <ion-row no-padding>
      <ion-col>
        <button clear small primary (clic")="readBt"()">
          <ion-icon na'e='b'ok'></ion-icon>
          Read
        </button>
      </ion-col>
      <ion-col text-center>
        <button clear small primary (clic")="favoriteBt"()">
          <ion-icon na'e='bookm'rk'></ion-icon>
          Favorite
        </button>
      </ion-col>
      <ion-col text-right>
```

```
            <button clear small primary (clic")="shareBt"()">
              <ion-icon na'e='share-'lt'></ion-icon>
              Share
            </button>
          </ion-col>
        </ion-row>
    </ion-card>
```

This is a simple Ionic markup, but there are some important things to understand here. We are using custom styles, so we are using a `post-cmp` class for it. `postDate` is trimmed to `10` characters, because that's all we need. In post content, we have used the `htmlPipe`, which deletes WordPress's Read More URL; otherwise, it will open the URL in the App's Web View.

The following code should be present in `posts.scss`:

```
    /* /app/components/posts/posts.scss */
    .post-cmp {
        .post-title {
            text-align:center;
            background: #ECEFF1;
            color:#263238;
            .date {
                font-weight: bold;
            }
        }
        border-radius:1px;
    }
```

Application Pages

We have defined all the providers and services for our application. Let's define the pages of our application. We will have the following pages in our application:

- `TabsPage`
- `PostsPage`
- `PostPage`
- `CategoryListPage`
- `WpPageList`
- `WpPage`
- `FavoritePage`
- `SettingsPage`

Defining TabsPage

`TabsPage` acts as a container for our other pages in the app. The following code should be present in `tabs.ts`:

```
// /app/pages/tabs/tabs.ts
import {Component} from '@angular/core';
import {CategoryListPage} from '../category-list/category-list';
import {PostsPage} from '../posts/posts';
import {FavoritePage} from '../favorite/favorite';
import {WpPageList} from '../wp-page-list/wp-page-list';

@Component({
    templateUrl: 'build/pages/tabs/tabs.html'
})
export class TabsPage {
    firstTab = PostsPage;
    secondTab = CategoryListPage;
    thirdTab = WpPageList;
    fourthTab = FavoritePage;
}
```

This is almost self-explanatory. We have initialized the root page for our four tabs through the `firstTab`, `secondTab`, `thirdTab`, and `fourthTab` variables and set it to `PostsPage`, `CategoryListPage`, `WpPageList`, and `FavoritePage`, respectively.

The following code should be present in `tabs.html`:

```
<!-- Template: /app/pages/tabs/tabs.html -->
<ion-tabs primary tabbarPlacement="bottom">
    <ion-tab [root]="firstTab" tabTitle="Posts" tabIcon="create"></ion-tab>
    <ion-tab [root]="secondTab" tabTitle="Categories" tabIcon="list"></ion-
tab>
    <ion-tab [root]="thirdTab" tabTitle="Pages" tabIcon="document"></ion-
tab>
    <ion-tab [root]="fourthTab" tabTitle="Favorites"
tabIcon="bookmark"></ion-tab>
</ion-tabs>
```

Defining PostsPage

`PostsPage` lists all the posts of a WordPress site. We will use the posts component to show the post. We will also use infinite scroll to load more posts when the user scrolls to the end. The following code should be present in `posts.ts`:

```
// /app/pages/posts/posts.ts
```

```
import {Component} from '@angular/core';
import {Control} from '@angular/common';
import {NavController, NavParams, Modal, Storage, LocalStorage, Loading,
Toast, Events} from 'ionic-angular';
import {Observable} from 'rxjs/Rx';
import {PostCmp} from '../../components/post/post';
import {PostPage} from '../post/post';
import {SettingsPage} from '../settings/settings';
import {WpProvider} from '../../providers/wp-provider/wp-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/filter';
import 'rxjs/add/operator/distinctUntilChanged';

@Component({
  templateUrl: 'build/pages/posts/posts.html',
  directives: [PostCmp]
})
export class PostsPage {
  hideSearch:Boolean = true;
  searchbar:Control = new Control();
  pageCount:number = 1;
  favoriteList = [];
  getData = true;
  sort:string;
  noMore:Boolean = false;
  posts:any;
  storage = new Storage(LocalStorage);
  category:any = {};
  query:{};
  constructor(public params:NavParams, public nav:NavController, public
   wp:WpProvider, public up:UtilProvider, public events: Events) {
    this.category = this.params.get('category');

    // Getting Favorite List
    this.storage.get('favorite')
    .then(data => {
        if(data === null) {
            data = "[]";
        }
        this.favoriteList = JSON.parse(data);
    });
    // Getting Settings
    this.storage.get('settings')
    .then(data => {
      this.sort = JSON.parse(data).sort;
      let query = this.createQuery();
      this.getPosts(query);
```

```
        });
        // Search Subscription
        this.searchbar.valueChanges
        .debounceTime(2000)
        .filter(value => value.length === 0 || value.trim().length > 2)
        .distinctUntilChanged()
        .subscribe((value)=> {
          this.resetSettings();
          let query = this.createQuery();
          this.getPosts(query);
        });
        // If Sort Order Changed
        this.events.subscribe("sort:changed", value => {
          this.resetSettings();
          this.sort = value;
          let query = this.createQuery();
          this.getPosts(query);
        });
      }
```

`Constructor()` initializes various variables. Then we get a list of favorite posts in `this.favoriteList`. Then we get settings from `LocalStorage` in `this.settings`. We also subscribed to the search bar's `valueChanges` object's `Observable` and query the WP API accordingly. Finally, we listen to the `sort:changed` event and get posts accordingly.

```
      getPosts(query) {
        let loader = this.up.getLoader("Loading Posts...");
        this.nav.present(loader);
        this.wp.getPosts(query)
        .subscribe(posts => {
          this.posts = posts;
          loader.dismiss();
        }, (error) => {
          loader.dismiss();
        });
      }
```

`getPosts(query)` fetches posts according to the given query using the `WpProvder` class's `getPosts` method and assigns all the posts to `this.posts`, which we display on the page. We are also showing Ionic `Loading` while getting posts.

```
      loadMore(infinteScroll) {
        this.pageCount++;
        let query = this.createQuery();
        let toast = this.up.getToast("There are no more posts.");
        this.wp.getPosts(query)
        .subscribe(posts => {
```

```
        infinteScroll.complete();
        if(posts.length < 1) {
         this.noMore = true;
         infinteScroll.enable(!this.noMore);
         this.nav.present(toast);
        } else {
          this.posts = this.posts.concat(posts);
        }
      });
    }
```

`loadMore(infiniteScroll)` is called when the user reaches the bottom of the screen. When called, it increases the `pageCount`, then calls the `WpProvider` class's `getPosts`, and adds the retrieved posts to the current list of posts.

```
    toggleSearch() {
      this.hideSearch = !this.hideSearch;
    }
```

`toggleSearch()` toggles the `hideSearch` variable, which in turns displays or hides the search bar that we have on the page.

```
    read(data) {
      this.nav.push(PostPage, {postData: data});
    }
```

`read(data)` opens the `PostPage` with the given data. It is called when the user clicks on the **Read** button.

```
    favorite(post) {
      let newPost:Boolean = true;
      let toast;
      let message:string;
      this.favoriteList.forEach(favPost => {
        if(JSON.stringify(favPost) === JSON.stringify(post)) {
          newPost = false;
        }
      });
      if(newPost) {
        this.favoriteList.push(post);
        this.storage.set('favorite', JSON.stringify(this.favoriteList));
        message = "This Post is saved in Favorite List";
      } else {
        message = "This Post is already in Favorite List";
      }
      toast = this.up.getToast(message);
      this.nav.present(toast);
    }
```

`favorite(post)` is fired when the favorite button is clicked. It adds the current post to `favoriteList`, if it is not in the list.

```
      openSettings() {
        this.nav.push(SettingsPage);
      }
```

`openSettings()` is fired when the search button on **Navbar** is clicked. It opens the `SettingsPage`.

```
      resetSettings() {
        this.noMore = false;
        this.pageCount = 1;
      }
```

`resetSettings()` resets the `pageCount` and `noMore` variables.

```
      createQuery() {
        let query = {};
        query['page'] = this.pageCount;
        if(this.sort) {
          query['order'] = this.sort;
        }
        if(this.searchbar.value) {
          query['search'] = this.searchbar.value;
        }
        if(this.category) {
          query['categories'] = this.category.id;
        }
        return query;
      }
    }
```

`createQuery()` creates a query object to get posts from WordPress.

Template

The following code should be present in `posts.html`:

```
    <!-- Template: /app/pages/posts/posts.html -->
    <ion-header>
     <ion-navbar primary>
     <ion-title *ngIf="!category">Posts</ion-title>
     <ion-title *ngIf="category">{{category.name}}</ion-title>
     <ion-buttons end>
     <button (click)="toggleSearch()"><ion-icon name="search"></ion-
```

```
icon></button>
 <button (click)="openSettings()"><ion-icon name="settings"></ion-
icon></button>
 </ion-buttons>
 </ion-navbar>
<ion-toolbar [hidden]="hideSearch">
 <ion-searchbar [ngFormControl]="searchbar"></ion-searchbar>
 </ion-toolbar>
</ion-header>
```

We have a Navbar with two buttons: one to toggle `searchbar`, and one to open `SettingsPage`. Then we have a toolbar with a search bar in it. It is good to know that using the toolbar here ensures our search bar always sticks to the top of the page.

In `ion-content`, we have a list of posts using the post component. Then we have an `ion-infinte-scroll` component for infinite scrolling.

Defining PostPage

`PostPage` shows a single post in detail. It shows author and post details, post content, and comments. The following code should be in `post.ts`:

```
// /app/pages/post/post.ts
import {Component} from '@angular/core';
import {Page, NavController, NavParams} from 'ionic-angular';
import {HtmlPipe} from '../../pipes/htmlPipe';
@Component({
  templateUrl: 'build/pages/post/post.html',
  pipes: [HtmlPipe]
})
export class PostPage {
  postData:any;
  constructor(public nav:NavController, public params: NavParams) {
    this.postData = this.params.get('postData');
  }
}
```

It is quite simple here. We get `postData` from `NavParams`. We display all that data on the page's view. The following code should be present in `post.html`:

```
<!-- Template: /app/pages/post/post.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Post</ion-title>
  </ion-navbar>
</ion-header>

<ion-content class="post">
  <ion-list>
    <ion-card>
        <img [src]="postData.media.source_url"
         *ngIf="postData.media.source_url">
        <ion-item class="post-title" text-wrap>
            <h1>{{postData.post.title.rendered}}</h1>
            <h4 class="date">{{postData.post.date.substr(0,10)}}</h4>
        </ion-item>
        <ion-item>
            <ion-avatar item-left>
                <img [src]="postData.author.avatar_urls[96]">
            </ion-avatar>
            <h2>{{postData.author.name}}</h2>
        </ion-item>
    </ion-card>

    <ion-card text-wrap *ngIf="postData.post.content.rendered">
        <p padding [innerHtml]="postData.post.content.rendered |
         htmlPipe:'change'"></p>
    </ion-card>

<ion-list *ngIf="postData.comments">
  <ion-item-divider light>Comments</ion-item-divider>
  <ion-item text-wrap *ngFor="let comment of postData.comments">
    <ion-avatar item-left>
      <img [src]="comment.author_avatar_urls['96']">
    </ion-avatar>
    <h2>{{comment.author_name}}</h2>
    <p [innerHTML]="comment.content.rendered | htmlPipe: 'change'"></p>
  </ion-item>
</ion-list>
  </ion-list>
</ion-content>
```

This template is almost similar to the `Post` component's template. Some notable differences are the use of `htmlPipe` with the `change` argument and showing the post's comments.

Defining CategoryListPage

`CategoryListPage` shows the list of categories of a WordPress site. The following code should be present in `category-list.ts`:

```
// /app/pages/category-list/category-list.ts
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {PostsPage} from '../posts/posts';
import {WpProvider} from '../../providers/wp-provider/wp-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider/';

@Component({
  templateUrl: 'build/pages/category-list/category-list.html',
})
export class CategoryListPage {
  list:Array<any>;
  constructor(public nav:NavController, public wp:WpProvider, public
   up:UtilProvider) {
    let loader = this.up.getLoader("Loading Categories");
    this.nav.present(loader);
    this.wp.getCategories()
    .subscribe(data => {
      this.list = data;
      loader.dismiss();
    }, ()=> {
      loader.dismiss();
    })
  }
  openCategory(category) {
    this.nav.push(PostsPage, {"category": category});
  }
}
```

In the constructor, we initialize variables, get all the categories using the `WpProvider` class's `getCategories` method, and set all categories to `this.list`. The `openCategory` method opens `PostsPage` with the given category as a parameter. The following code should be present in `category-list.html`:

```
<!-- Template: /app/pages/category-list/category-list.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Categories</ion-title>
  </ion-navbar>
</ion-header>

<ion-content class="category-list">
  <ion-list>
    <ion-item *ngFor="let category of list"
     (click)="openCategory(category)">
      <h2>{{category.name}}</h2>
      <ion-badge item-right primary>{{category.count}}</ion-badge>
    </ion-item>
  </ion-list>
</ion-content>
```

We are showing a list of categories, with their name and number of posts in the category.

Defining WpPageList

`WpPageList` shows a list of the WordPress pages of our WordPress site. The following code should be present in `wp-page-list.ts`:

```
// /app/pages/wp-page-list/wp-page-list.ts
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {WpProvider} from '../../providers/wp-provider/wp-provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';
import {WpPage} from '../wp-page/wp-page';

@Component({
    templateUrl: 'build/pages/wp-page-list/wp-page-list.html'
})
export class WpPageList {
    pages:Array<any>;
    constructor(public nav:NavController, public wp: WpProvider, public
     up: UtilProvider) {
        let loader = this.up.getLoader('Loading Pages ...');
        this.nav.present(loader);
        this.wp.getPages()
```

```
            .subscribe(pages => {
                this.pages = pages;
                loader.dismiss();
            }, ()=> {
                loader.dismiss();
            });
        }
        openPage(page) {
            this.nav.push(WpPage, {page: page});
        }
    }
```

We get a list of pages using the `WpProvider` class's `getPages` method, assign those pages to `this.pages`, and show them in the page's view. The `openPage` function opens the `WpPage` Ionic page. The following code should be present in `wp-page-list.html`:

```
<!-- Template: /app/pages/wp-page-list/wp-page-list.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Pages</ion-title>
    </ion-navbar>
</ion-header>
<ion-content>
    <ion-list>
        <a ion-item *ngFor="let page of pages"
         (click)="openPage(page)">
            {{page.title.rendered}}
        </a>
    </ion-list>
</ion-content>
```

Defining WpPage

`WpPage` shows the content of a WordPress page. The following code should be present in `wp-page.ts`:

```
// /app/pages/wp-page/wp-page.ts
import {Component} from '@angular/core';
import {NavParams} from 'ionic-angular';
import {HtmlPipe} from '../../pipes/htmlPipe';
@Component({
    templateUrl: 'build/pages/wp-page/wp-page.html',
    pipes: [HtmlPipe]
})
export class WpPage {
    page:any;
```

```
        constructor(public params:NavParams) {
            this.page = this.params.get('page');
        }
    }
```

We get the page content as `NavParam` and we assign that value to `this.page`. The following code should be present in `wp-page.html`:

```
    <!-- Template: /app/pages/wp-page/wp-page.html -->
    <ion-header>
        <ion-navbar primary>
            <ion-title>{{page.title.rendered}}</ion-title>
        </ion-navbar>
    </ion-header>

    <ion-content>
        <ion-item text-wrap>
            <p [innerHTML]="page.content.rendered | htmlPipe:'change'"></p>
        </ion-item>
    </ion-content>
```

We have displayed the page title and page content in the view.

Defining FavoritePage

`FavoritePage` shows a list of favorite posts and allows us to delete any post from the favorites list. The following code should be present in `favorite.ts`:

```
    // /app/pages/favorite/favorite.ts
    import {Component} from '@angular/core';
    import {Storage,LocalStorage, NavController} from 'ionic-angular';
    import {PostPage} from '../post/post';

    @Component({
        templateUrl: 'build/pages/favorite/favorite.html'
    })
    export class FavoritePage {
        favoriteList = [];
        storage = new Storage(LocalStorage);
        constructor(public nav:NavController) {}
        ionViewWillEnter() {
            this.storage.get('favorite')
            .then(data => {
                if(data !== null) {
                    this.favoriteList = JSON.parse(data);
                }
            });
        }
        read(post) {
            this.nav.push(PostPage, {postData:post});
        }
        removeFavorite(index) {
           this.favoriteList.splice(index, 1);
           this.storage.set('favorite', this.favoriteList);
        }
    }
```

Each time the user enters this page, we will get a list of favorite posts from `LocalStorage` and assign them to `this.favoriteList`. The `read` method opens the `PostPage` with the given post. The `removeFavorite` method removes the post at the given `index` value from `favoriteList` array and also from `LocalStorage`. The following code should be present in `favorite.html`:

```
<!-- Template: /app/pages/favorite/favorite.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Favorites</ion-title>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list>
        <ion-item-sliding *ngFor="let favorite of favoriteList; let i =
         index">
            <ion-item>
                {{favorite.post.title.rendered}}
                <button item-right outline
                 (click)="read(favorite)">Read</button>
            </ion-item>
            <ion-item-options>
                <button primary
                 (click)="removeFavorite(i)">Delete</button>
            </ion-item-options>
        </ion-item-sliding>
    </ion-list>
</ion-content>
```

We have used `ion-item-sliding`, which shows the list title and a **Read** button. When the user slides the item, we have displayed a **Delete** button to delete the item from `favoriteList`.

Defining SettingsPage

`SettingsPage` allows us to change some settings for our app. The following code should be present in `settings.ts`:

```
// /app/pages/settings/settings.ts
import {Component} from '@angular/core';
import {ViewController, LocalStorage, Storage, Events} from 'ionic-
angular';
import {Control} from '@angular/common';
import {PushProvider} from '../../providers/push-provider/push-provider';
@Component({
   templateUrl: 'build/pages/settings/settings.html'
})
export class SettingsPage {
    pushToggle:Control = new Control();
    storage = new Storage(LocalStorage);
    settings:any;
    constructor(public pushProvider: PushProvider, public
     events:Events) {
          this.storage.get('settings')
          .then(data => {
              this.settings = JSON.parse(data);
              this.pushToggle.updateValue(this.settings.push);
          });
          this.pushToggle.valueChanges.skip(1)
          .subscribe(value => {
              this.changePush(value);
          });
    }
    saveSettings() {
        this.storage.set('settings', JSON.stringify(this.settings));
    }
    changeSort() {
        this.events.publish("sort:changed", this.settings.sort);
        this.saveSettings();
    }
    changePush(push) {
        if(push === true) {
            this.pushProvider.registerDevice()
            .then(() => {})
        } else {
            this.pushProvider.unregisterDevice()
            .then(() => {})
        }
        this.saveSettings();
    }
}
```

We use ion-toggle to enable and disable the push notification, and we use ion-select to set the sort order. In the constructor, we get the previous settings and change the UI accordingly. We have also used reactive programming on the push toggle and we skip the first change that occurs when we change our view due to the previous settings from LocalStorage. The saveSettings method saves settings to LocalStorage. The changeSort method publishes the sort:changed event and calls the saveSettings method. The changePush method registers or unregisters the user's device for push notification. The following code should be present in settings.html:

```
<!-- Template: /app/pages/settings/settings.html -->
<ion-header>
    <ion-navbar primary>
    <ion-title>Settings</ion-title>
    </ion-navbar>
</ion-header>

<ion-content>
    <ion-list>
        <ion-item>
            <ion-label> Push Notifications</ion-label>
            <ion-toggle [ngFormControl]="pushToggle"></ion-toggle>
        </ion-item>
        <ion-item>
            <ion-label>Posts Sorting</ion-label>
            <ion-select [(ngModel)]="settings.sort"
             (change)="changeSort()">
                <ion-option value="desc">Latest</ion-option>
                <ion-option value="asc">Oldest</ion-option>
            </ion-select>
        </ion-item>
     </ion-list>
</ion-content>
```

Styling our app

We need to add our Post component's SCSS file to /app/themes/app.core.scss as follows:

```
@import '../components/post/post';
```

We also need to change the colors to make our app look a little better. Change the primary color value in /app/themes/app.variables.scss as follows:

```
primary:    #607D8B
```

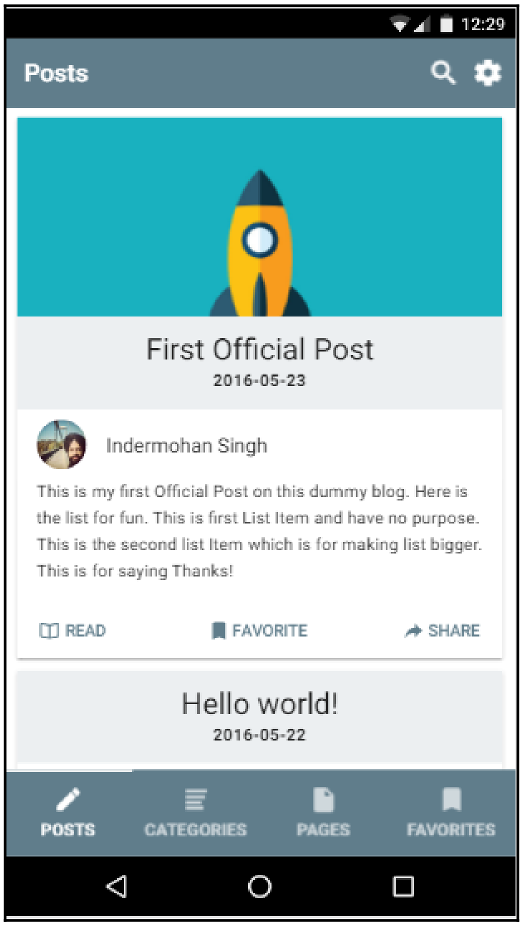
Running our app

Let's run our application on an actual device using the following command:

```
ionic platform add android
ionic run android
```

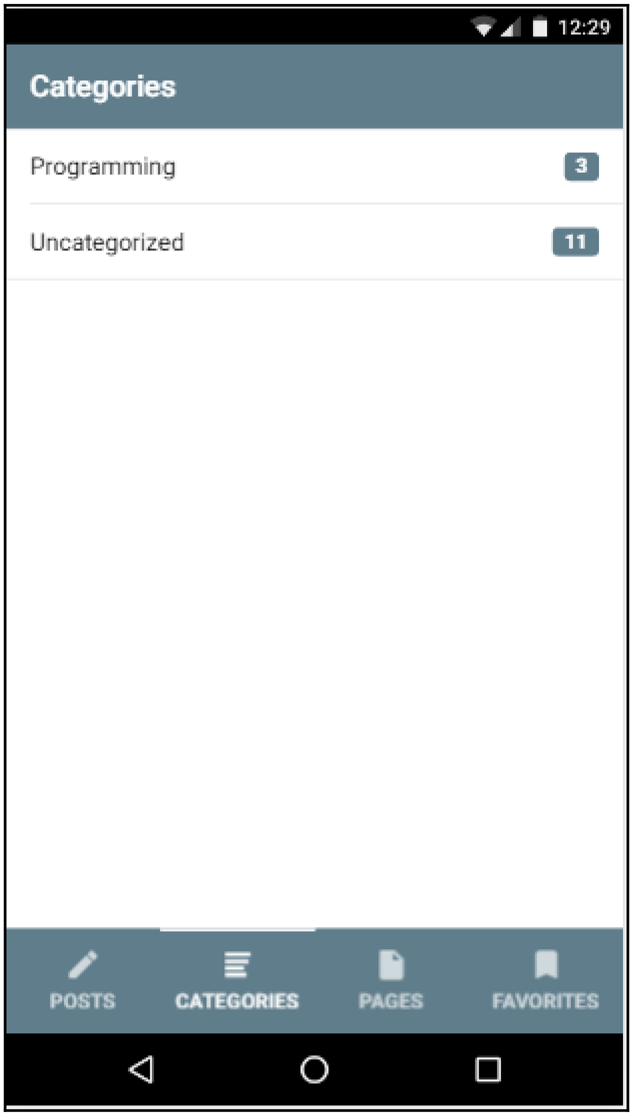
App screenshots

The following screenshot shows `PostsPage`:

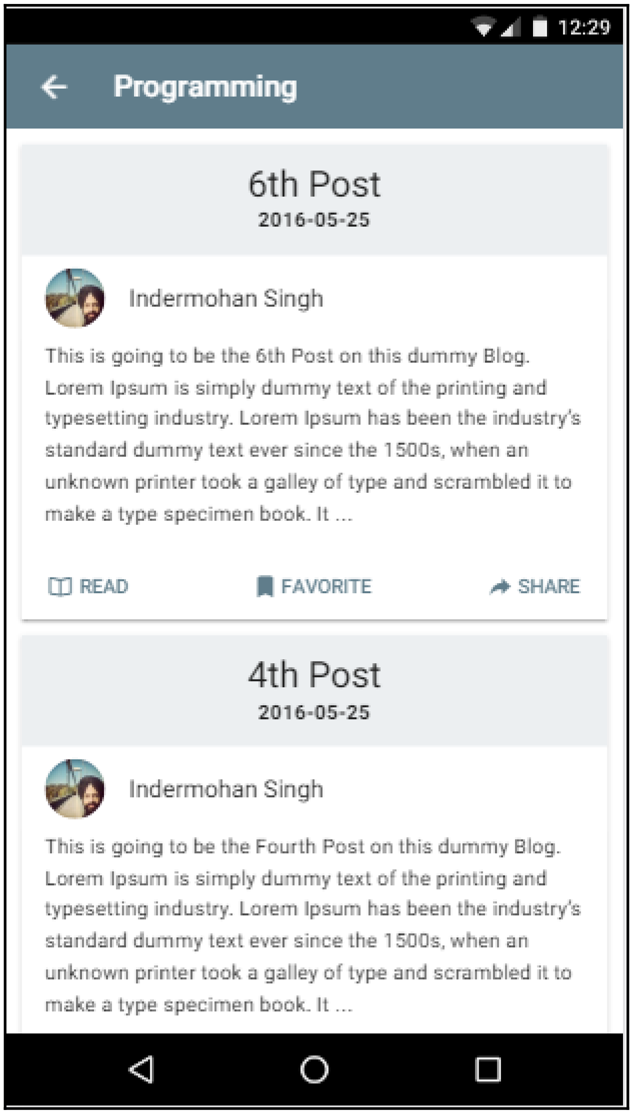


This is my first Official Post on this dummy blog. Here is the list for fun. This is first List Item and have no purpose. This is the second list Item which is for making list bigger. This is for saying Thanks!

The following screenshot shows `CategoryListPage`:



The following screenshot shows `CategoryPage`:



6th Post

2016-05-25

Indermohan Singh

This is going to be the 6th Post on this dummy Blog. Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It ...

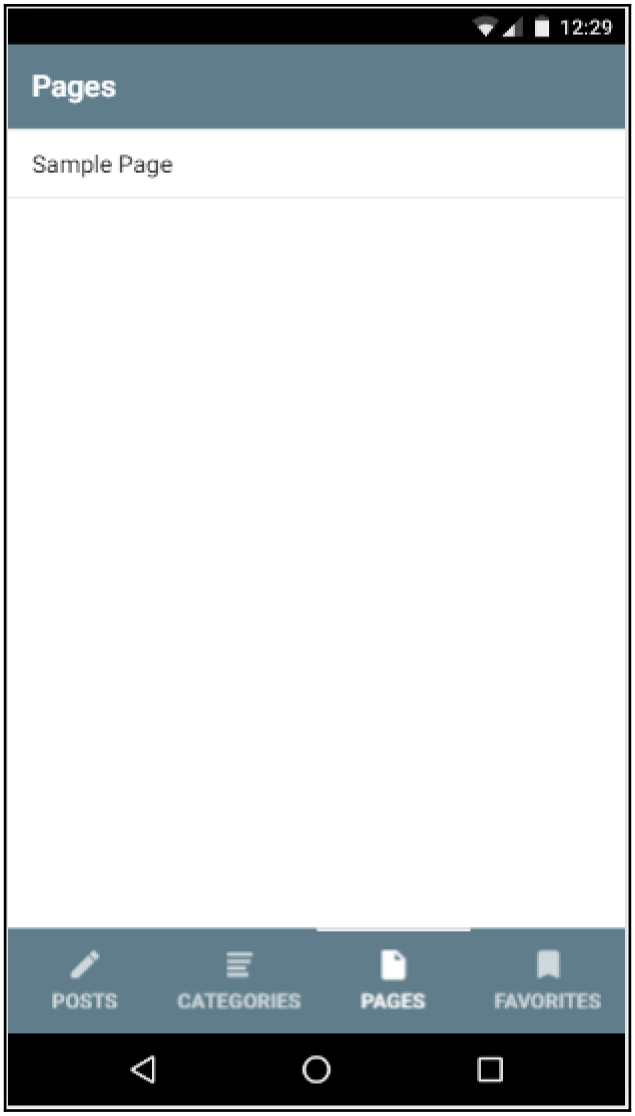
4th Post

2016-05-25

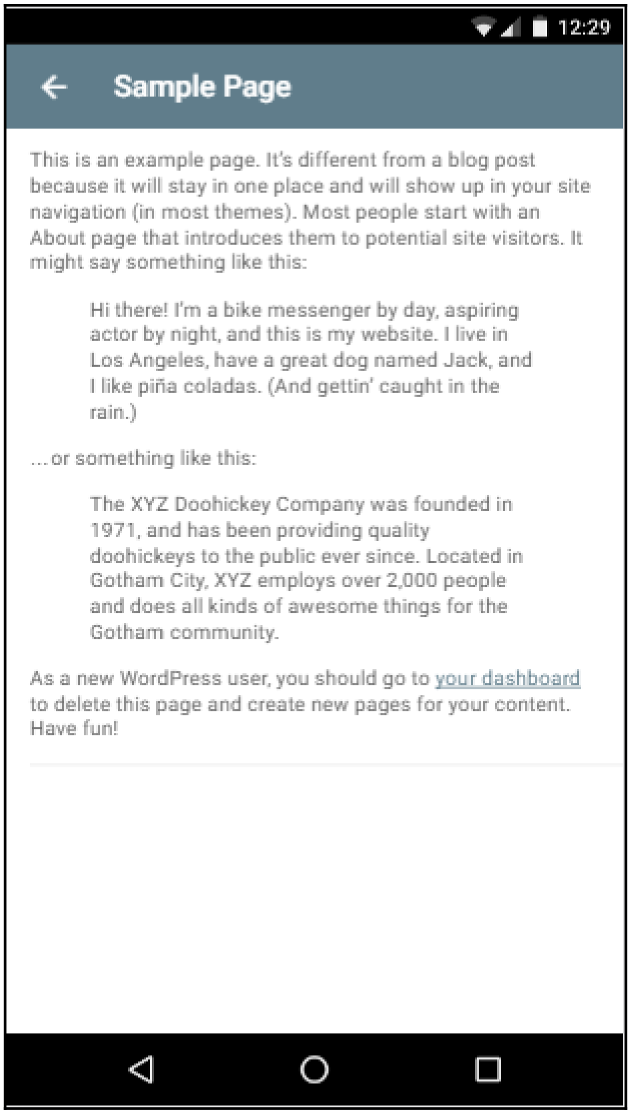
Indermohan Singh

This is going to be the Fourth Post on this dummy Blog. Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It ...

The following screenshot shows `WpPageList`:



The following screenshot shows `WpPage`:



12:29

Sample Page

This is an example page. It's different from a blog post because it will stay in one place and will show up in your site navigation (in most themes). Most people start with an About page that introduces them to potential site visitors. It might say something like this:

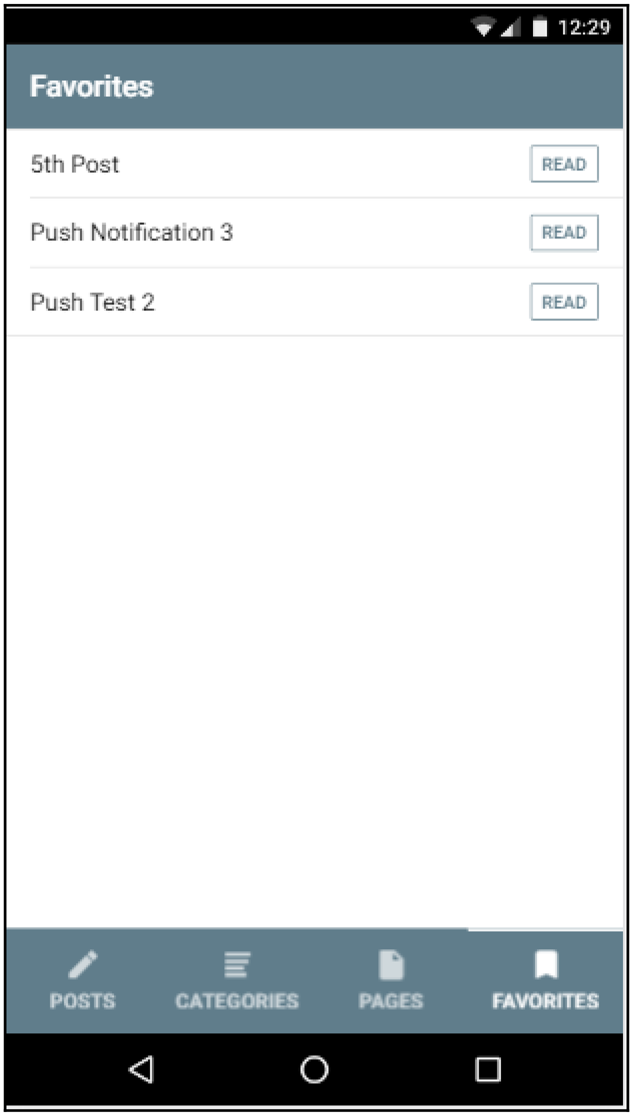
> Hi there! I'm a bike messenger by day, aspiring actor by night, and this is my website. I live in Los Angeles, have a great dog named Jack, and I like piña coladas. (And gettin' caught in the rain.)

...or something like this:

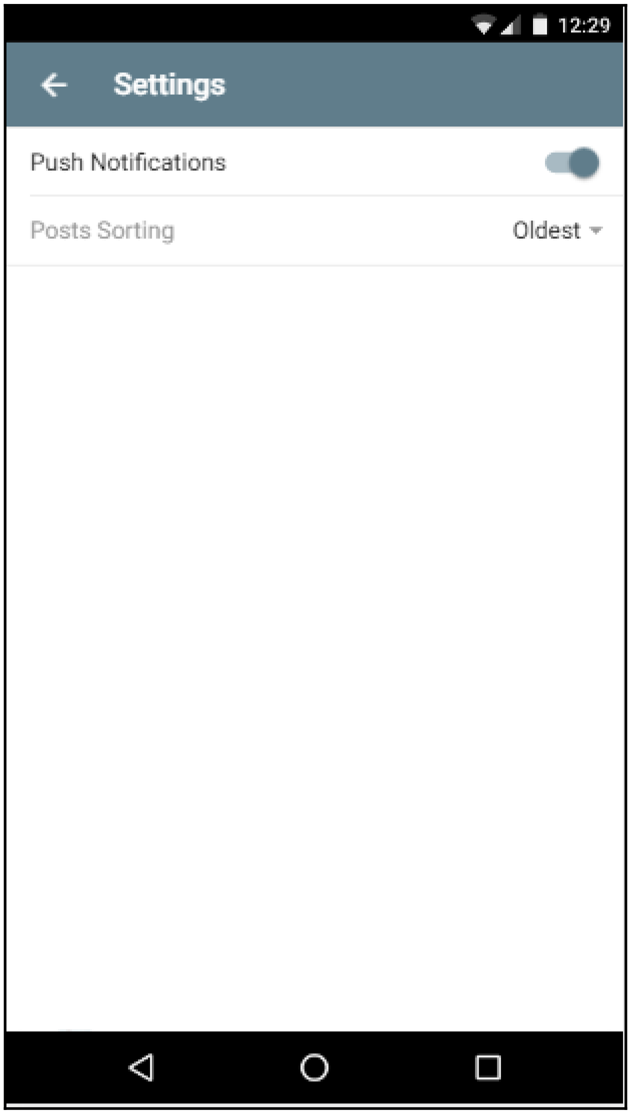
> The XYZ Doohickey Company was founded in 1971, and has been providing quality doohickeys to the public ever since. Located in Gotham City, XYZ employs over 2,000 people and does all kinds of awesome things for the Gotham community.

As a new WordPress user, you should go to your dashboard to delete this page and create new pages for your content. Have fun!

The following screenshot shows `FavoritePage`:



The following screenshot shows `SettingsPage`:



Push Notifications

Posts Sorting Oldest

Summary

In this chapter, we learned how to use the WordPress REST API to create a WordPress mobile client. We used Ionic's toast, loader, infinite scroll, and lots of other components and APIs. We utilized reactive programming and Ionic events, along with `LocalStorage` to save data locally. We also learned how to use push notification inside a Cordova/Ionic app using the push notification plugin.

In the next chapter, we will create a media player using Ionic, which will support audio and video playback.

6
Media Player App

In the previous chapter, we created a WordPress client, with posts, categories, and pages. We focused on component-based programming and used various advanced Ionic 2 features like infinite scroll.

In this chapter, we will create a media player with audio and video playback support. Audio and video playback support is added using **Cordova** plugins. We will have play, pause, next, and previous buttons for playlists, and a seek bar for changing the current time of playback media.

This app is different from most of the previous applications we have worked on because we can't test this application in a browser. This is because we are using native capabilities, which do not work in the browser. So, an important take away from this chapter is how to build apps that work on the device only.

We will learn the following things:

- Using native audio and video using plugins
- Using the file system of a device
- Ionic `Range`
- Sliding a list item
- Creating a file browser

Key things

In the application, there are a few keys things to be noted and learned before we start working on coding the player.

Playback

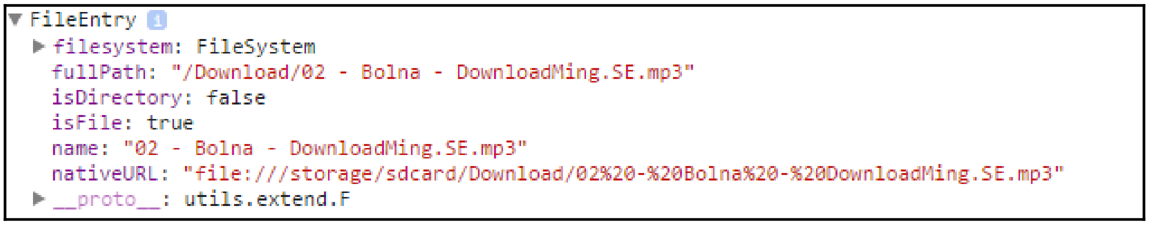
There are two ways to create a playback functionality for this application:

- Using **HTML5** audio and video
- Using the device's native audio and video using plugins

I have used the latter way of providing this functionality, but it is possible to create a media player using HTML5. In fact, during the development of this application, I created an HTML5 version too. However, it was buggy in providing the actual duration of an MP3 file, so I chose the native method.

File System

Since client-side JavaScript engines can't directly access the file system of devices, we need to use the **Cordova file plugin** to access the file system. For each file or folder, the Cordova file plugin returns a `FileEntry` object, which looks like the following:



```
▼ FileEntry
  ▶ filesystem: FileSystem
    fullPath: "/Download/02 - Bolna - DownloadMing.SE.mp3"
    isDirectory: false
    isFile: true
    name: "02 - Bolna - DownloadMing.SE.mp3"
    nativeURL: "file:///storage/sdcard/Download/02%20-%20Bolna%20-%20DownloadMing.SE.mp3"
  ▶ __proto__: utils.extend.F
```

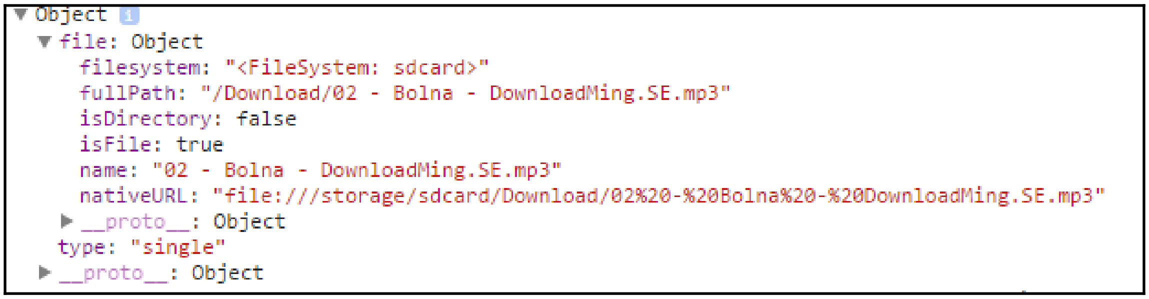
We will refer to this structure as `FileEntry` in our chapter:

- `isDirectory` is Boolean. It is `true` if entry is a directory; otherwise, it is `false`
- `isFile` is `true`, if entry is a file; otherwise, it is `false`
- `nativeURL` is the actual URL of the file
- `name` is the name of the file

Local storage

We are using local storage to store the data of the currently playing file and playlist in the `current` and `playlist` keys of local storage, respectively.

If a single file is playing, we will have something like the following in the current key:

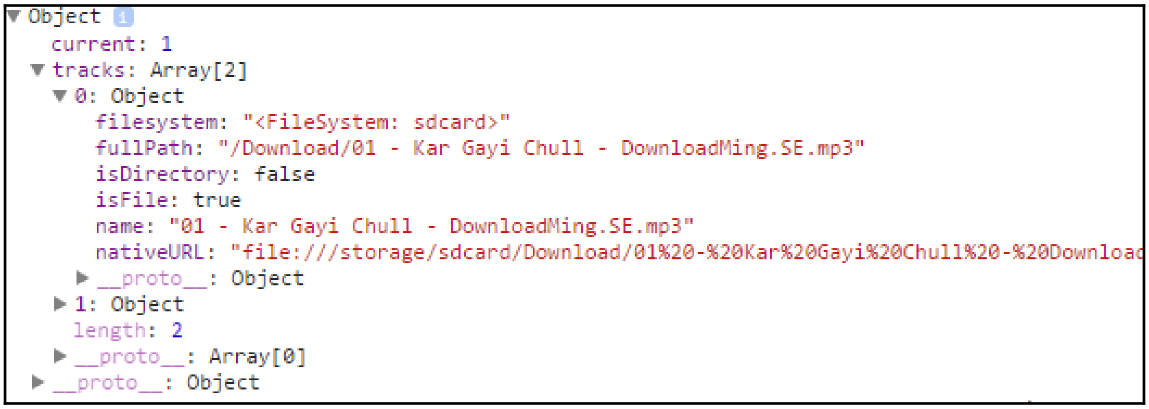


```
▼ Object
  ▼ file: Object
      filesystem: "<FileSystem: sdcard>"
      fullPath: "/Download/02 - Bolna - DownloadMing.SE.mp3"
      isDirectory: false
      isFile: true
      name: "02 - Bolna - DownloadMing.SE.mp3"
      nativeURL: "file:///storage/sdcard/Download/02%20-%20Bolna%20-%20DownloadMing.SE.mp3"
    ▶ __proto__: Object
    type: "single"
  ▶ __proto__: Object
```

Let's understand the preceding code:

- `type` property has `single` as its value
- `file` property stores `FileEntry` of the file

If we will play a playlist, we will have `{type:'playlist'}` in `current` key and `playlist` key is used to store the playlist data, and has the following structure:



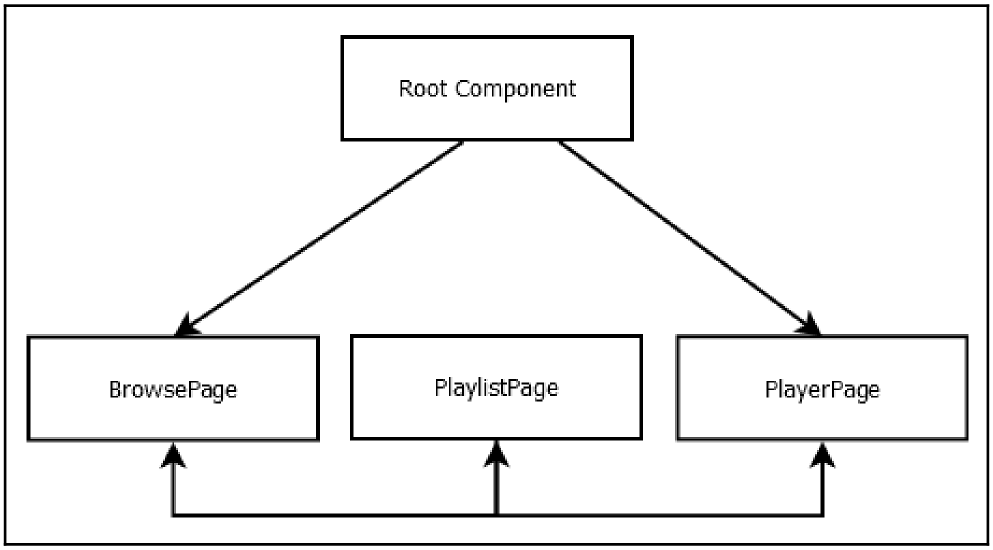
```
▼ Object
    current: 1
  ▼ tracks: Array[2]
    ▼ 0: Object
        filesystem: "<FileSystem: sdcard>"
        fullPath: "/Download/01 - Kar Gayi Chull - DownloadMing.SE.mp3"
        isDirectory: false
        isFile: true
        name: "01 - Kar Gayi Chull - DownloadMing.SE.mp3"
        nativeURL: "file:///storage/sdcard/Download/01%20-%20Kar%20Gayi%20Chull%20-%20Downloac
      ▶ __proto__: Object
    ▶ 1: Object
      length: 2
    ▶ __proto__: Array[0]
  ▶ __proto__: Object
```

Let's understand the preceding code:

- `track` is an array which holds `FileEntry` of each track in the playlist
- `current` property stores the index value of the current track in the `playlist.track` array

App flow

The following is how the control will flow inside our application:



Let's understand the flow:

- **RootComponent**: It is the root ionic component. It is defined inside the `/app/app.ts` file.
- **BrowsePage**: It is the page that is used for browsing the media in the device.
- **PlaylistPage**: It shows the playlist and deletes items from playlist. It also allows you to play the playlist.
- **PlayerPage**: It is the page where you will get player controls and the name of the file.

Scaffolding and setting up the app

We will start by scaffolding a blank application. Run the following command:

```
ionic start media-player blank --v2 --ts
```

Notice the `--v2` and `--ts` tag for scaffolding the Ionic 2 app based on **TypeScript**.

Before we start writing code for the app, we need to install some project-specific dependencies.

Installing dependencies

Following are some Cordova plugins that we need to install. They are as follows:

```
ionic plugin add cordova-plugin-media
ionic plugin add cordova-plugin-streaming-media
ionic plugin add cordova-plugin-file
```

Installing moment library:

```
npm install moment -save
```

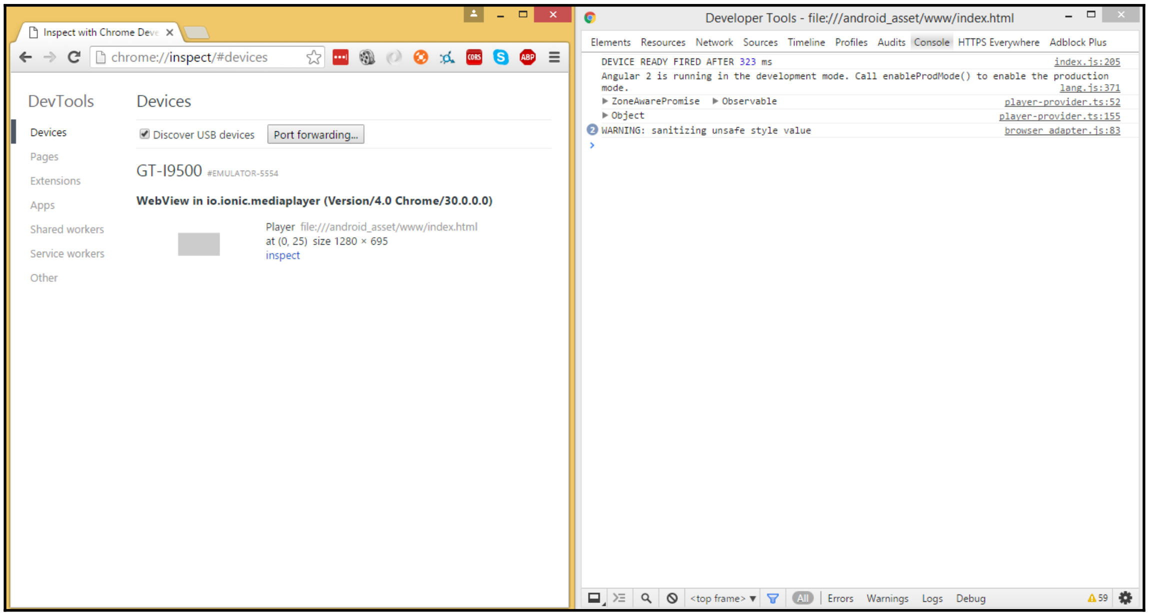
We are using `moment.js` to create a digital timer for displaying the playback time of our media.

Take a look at MomentJS docs at `http://momentjs.com/docs/` for more information.

Running and debugging the app

Our application is dependent on device-specific features like audio and video playback, and displaying the files and folders of our device. So, we can't run our app on a browser. We can run our application on an actual device using the ionic run command, but I prefer emulators like **Genymotion** or **BlueStacks**. It is easy to run our application on these emulators.

Debugging is also difficult when we are running our application on an actual device. But we can use the debugging capabilities of Safari (for iOS) or Chrome (for Android) for this purpose. You can use `chrome://inspect` for debugging. The following shows the `chrome://inspect`:


Inspect with Chrome Deve
chrome://inspect/#devices
DevTools
Devices
Pages
Extensions
Apps
Shared workers
Service workers
Other
Discover USB devices
Port forwarding...
inspect
Developer Tools - file:///android_asset/www/index.html
Elements Resources Network Sources Timeline Profiles Audits Console HTTPS Everywhere Adblock Plus
<top frame>
All Errors Warnings Logs Debug

GT-I9500 #EMULATOR-5554

WebView in io.ionic.mediaplayer (Version/4.0 Chrome/30.0.0.0)

Player file:///android_asset/www/index.html
at (0, 25) size 1280 × 695

```
DEVICE READY FIRED AFTER 323 ms                                       index.js:205
Angular 2 is running in the development mode. Call enableProdMode() to enable the production
mode.                                                                 lang.js:371
▶ ZoneAwarePromise  ▶ Observable                              player-provider.ts:52
▶ Object                                                     player-provider.ts:155
2 WARNING: sanitizing unsafe style value                      browser_adapter.js:83
>
```

Download Genymotion from `https://www.genymotion.com/` and BlueStacks from `http://www.bluestacks.com/`.

Coding our app

Now that we have everything set up to start working on our application, let's define the followings things:

- Defining our root app component
- Creating the components of our app
- Creating providers/services for various functionalities
- Creating Ionic pages for various views

Root app component

The root app component is the main component of our application. Previously, Ionic 2 used `@App` decorator for defining a root app component, but now it is using Angular's `@Component`.

The following code should be present in `app.ts`:

```
/* /app/app.ts */
import {Component, ViewChild} from '@angular/core';
import {Platform, Storage, LocalStorage, NavController, ionicBootstrap}
from 'ionic-angular';
import {StatusBar} from 'ionic-native';
import {FileProvider} from './providers/file-provider/file-provider';
import {UtilProvider} from './providers/util-provider/util-provider';
import {BrowsePage} from './pages/browse/browse';
import {PlayerPage} from './pages/player/player';
import {PlayListPage} from './pages/playlist/playlist';
import {PlayerProvider} from './providers/player-provider/player-provider';

@Component({
  templateUrl: 'build/app.html',
  queries: {
    nav: new ViewChild('content')
  },
  providers: [FileProvider, PlayerProvider, UtilProvider]
})

export class MyApp {
  rootPage: any;
  player:any;
  storage = new Storage(LocalStorage);
  pages:Array<{title:string, component:any}>;
  nav:NavController;
  constructor(platform: Platform, player:PlayerProvider) {
    this.player = player;
    platform.ready().then(() => {
      StatusBar.styleDefault();
      this.storage.get('current')
      .then(current => {
        current = JSON.parse(current);
        if(current === null) {
          this.rootPage = BrowsePage;
        } else if(current.type === "single") {
          let url = current.file.nativeURL;
          this.player.initialize(url);
          this.rootPage = PlayerPage;
        } else {
          this.player.initializePlaylist();
          this.rootPage = PlayerPage;
        }
      });
    });
    this.pages = [
      {title: 'Playing', component: PlayerPage},
      {title: 'Browse', component: BrowsePage},
      {title: 'Playlist', component: PlayListPage}
    ];
  }
  openPage(page) {
    this.nav.setRoot(page.component);
  }
}
ionicBootstrap(MyApp);
```

Let's understand the preceding code:

- `constructor()`: In constructor, we are checking if there was any previously-playing media, then go to `PlayerPage`. Otherwise, go to `BrowsePage`. We are also initializing our page's array.
- `openPage(page)`: This opens the page by using the `setRoot` method of navigation.
- Finally, we have used bootstrap on our Ionic application using `ionicBootstrap`.

Initially, Ionic used `@App` decorator, which automatically bootstrapped our application. It is now deprecated.

The following code is present in `app.html`:

```
<!--  Template app/app.html -->
<ion-menu [content]="content">
    <ion-toolbar>
        <ion-title>Menu</ion-title>
    </ion-toolbar>
    <ion-content>
        <ion-list>
            <a ion-item menuToggle *ngFor="let page of pages"
             (click)="openPage(page)">
                {{page.title}}
            </a>
        </ion-list>
    </ion-content>
</ion-menu>

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

We are showing a side menu here and an `ion-nav` component for creating a navigation stack.

Providers for our application

We will have the following providers:

- `UtilProvider`
- `FileProvider`
- `PlayerProvider`

Defining UtilProvider

`UtilProvider` basically abstracts the `Toast` and `Loading`API for us. We are using them heavily, so we abstracted their creation. The following code should be present in `util-provider.ts`:

```
/* /app/providers/util-provider/util-provider.ts */
import { Injectable } from '@angular/core';
import { Toast, Loading } from 'ionic-angular';

@Injectable()
export class UtilProvider {
    getLoader(content="Loading...") {
        let loading = Loading.create({
            content: content,
            duration: 3000
        });
        return loading;
    }

    getToast(message) {
        let toast = new Toast({
            message: message,
            duration:1000
        });
        return toast;
    }
}
```

Let's understand the preceding code:

- `getLoader(content)`:The `getLoader` function returns a loading component. A message for the loading component is given using the `content` argument.
- `getToast(content)`: This returns a `Toast` component. A message for `Toast` is given by the user using the `content` argument

Defining FileProvider

`FileProvider` is used for getting the files and directories of devices. We will use it to get our media from the device. The following should be present in `file-provider.ts`:

```
// /app/providers/file-providder/file-provider.ts
import {Injectable} from '@angular/core';
import {File} from 'ionic-native';
import 'rxjs/add/operator/map';
declare var cordova:any;
declare var window:any;
@Injectable()
export class FileProvider {
  root:string;
  getEntries(root = cordova.file.externalRootDirectory) {
   let promise = new Promise((resolve, reject) => {
      window.resolveLocalFileSystemURL(root, (entry)=> {
          let directoryReader = entry.createReader();
          directoryReader.readEntries(entries => {
            entries.sort(function(a,b) {
              if(a.isDirectory && !b.isDirectory) {
                return -1;
```

```
                  } else if(!a.isDirectory && b.isDirectory) {
                    return 1;
                  }
                  return 0;
                });
                resolve(entries);
              }, error => {
                reject(error);
              });
            }, (error) => {
              reject(error);
            });
       });
       return promise;
     }
   }
```

Let's understand the preceding code:

- `getEntries(root)`: The `getEntries` function gets files and directories from the given `root` path. If the `root` path is not given, the device's external root directory is used as default. This function also sorts the entries into folders and files.

Defining PlayerProvider

`PlayerProvider` is the most import element of our media player app. This provider has all the methods required for playback functionalities like play, pause, stop, next and previous track, and many more. The following code should be present in `player-provider.ts`:

```
   // /app/providers/player-provider/player-provider.ts
   import {Injectable} from '@angular/core';
   import {Storage, LocalStorage} from 'ionic-angular';
   import {MediaPlugin} from 'ionic-native';
   import {Observable} from 'rxjs/Rx';

   @Injectable()
   export class PlayerProvider {
       file:MediaPlugin;
       public url:String;
       public isPlaying:Boolean = false;
       isPlaylist:Boolean;;
       storage = new Storage(LocalStorage);
       playlist:{tracks:Array<any>, current:number};
       constructor() {
           this.storage.get('playlist')
```

```
            .then(playlist => {
                if(playlist === null) {
                    this.playlist = {tracks:[], current:null}
                } else {
                    this.playlist = JSON.parse(playlist);
                }
            });
        }
```

In constructor, we initialize our playlist array using `localstorage`. Let's look at some further code:

```
        play(url?, isPlaylist = false) {
            this.isPlaylist = isPlaylist;
            // If file is already playing
            if(this.isPlaying) {
                this.stop();
            }
            // If a new file is given to play
            if(url) {
               this.initialize(url);
            }
            this.file.play();
            this.isPlaying = true;

       }
```

`play(url,isPlaylist)` play function plays our file with the given `url`. It stops the previous playback, if it is running. It also sets the `isPlaying` variable to `true`. Let's look at some further code:

```
       isFinished() {
            return this.file.init.then(() => {
                return this.isPlaying = false;
            });
       }
```

`isFinished()` returns the `Promise` to the caller and also sets the `isPlaying` to `false` when playback is finished:

```
       initialize(file) {
            this.url = file;
            this.file = new MediaPlugin(file);
       }
```

`initialize(file)`initialize function initializes our media using the given `file` and media plugin, so that we can use it for other operations on it. Let's look at some further code:

```
pause() {
     this.file.pause();
     this.isPlaying = false;
}
```

`pause()` pauses our media playback and also sets `isPlaying` to `false`. Let's look at some further code:

```
stop() {
     if(this.file) {
         this.file.pause();
     }
     this.isPlaying = false;
}
```

`stop()` stops the playback and sets `isPlaying` to `false`. It is important to know that we can't release the underlying resource using `this.file.release()`, because it will fire a finish event immediately and we don't want that to happen. Let's look at some further code:

```
seekTo(seconds) {
     let milli = seconds * 1000;
     this.file.seekTo(milli);
}
```

`seekTo(seconds)` seeks the media playback to the given time in `seconds`. Let's look at some further code:

```
initializePlaylist() {
     this.storage.set('current', JSON.stringify({type:'playlist'}));
     if(!this.playlist.current) {
         this.playlist.current = 0;
}
     let file =
      this.playlist.tracks[this.playlist.current].nativeURL;
     this.initialize(file);
     this.isPlaylist = true;
 }
```

`intializePlaylist()`: If `playlist` is played, it is initialized using this function. It checks the current file of the playlist and initializes it. Let's look at some further code:

```
        startPlaylist() {
            this.stop();
            this.initializePlaylist();
            this.play(null,true);
        }
```

`startPlaylist()` starts playing the playlist. Let's look at some further code:

```
        getDuration() {
            let obs = Observable.interval(200).flatMap(value => {
                value = this.file.getDuration();
                return Observable.of(value);
            })
            .filter(value => value > 0)
            .take(1);
            return obs;
        }
```

`getDuration()` returns the `Observable` which gets the total time of a media file. Let's look at some further code:

```
        getCurrentPosition() {
            return this.file.getCurrentPosition();
        }
```

`getCurrentPosition()` returns the `Observable`, which gets the current position of playback.

```
        isAudio(ext) {
            let audioExt = ["mp3", "ogg", "amr", "wma", "m4a"];
            return audioExt.indexOf(ext) !== -1? true:false;
        }
```

`isAudio(ext)` returns `true` if `ext` is supported by our media player; otherwise, it returns `false`. It is used for audio files. Let's look at some further code:

```
        isVideo(ext) {
            let videoExt = ["mp4"];
            return videoExt.indexOf(ext) !== -1? true:false;
        }
```

`isVideo(ext)` does the same thing as `isAudio`, but for video files. Let's look at some further code:

```
addToPlayList(file) {
  let index = this.playlist.tracks.map(value =>
   value.nativeURL).indexOf(file.nativeURL);
  if(index < 0) {
    this.playlist.tracks.push(file);
    this.storage.set('playlist', JSON.stringify(this.playlist));
  } else {
      return -1;
  }
}
```

`addToPlaylist(file)` adds the `file` to `playlist` if it is not in the `playlist` already, and saves it to local storage. Let's look at some further code:

```
next() {
    if(this.playlist.current < this.playlist.tracks.length - 1) {
        this.stop();
        this.playlist.current++;
        let url =
         this.playlist.tracks[this.playlist.current].nativeURL;
        this.play(url, true);
        this.storage.set('playlist',
         JSON.stringify(this.playlist));
    } else {
        return -1;
    }
}
```

`next()` function plays the next `file` from the `playlist`. If there is no next `file`, it returns `-1` to the caller. Let's look at some further code:

```
back() {
    if(this.playlist.current > 0) {
        this.stop();
        this.playlist.current--;
        let url =
         this.playlist.tracks[this.playlist.current].nativeURL;
        this.play(url, true);
        this.storage.set('playlist',
         JSON.stringify(this.playlist));
    } else {
        return -1;
    }
}
```

back() function plays the previousfile from the playlist. If is there is no file to play, it returns -1 to the caller. Let's look at some further code:

```
getFileName() {
        let promise = new Promise((res, rej) => {
            if(this.isPlaylist) {
                let current = this.playlist.current;
                res(this.playlist.tracks[current].name);
            } else {
                this.storage.get('current')
                .then(current => {
                    if(current) {
                        current = JSON.parse(current);
                        res(current.file.name);
                    } else {
                        res(null);
                    }
                });
            }
        });

        return promise;
    }
  }
```

getFileName() returns a promise, which resolves into the file name of the currently playing file.

Imclock pipe

Imclock pipe is used to convert seconds into a digital clock format. The following code should be present in clock.ts:

```
// /app/pipes/clock.ts
import {Pipe, PipeTransform} from '@angular/core';
import * as moment from 'moment';

@Pipe({ name: 'imClock' })
export class ImClock implements PipeTransform {
  transform(value: any, ...args: any[]) {
    let duration:any = moment.duration(value, args[0]);
    return this.toClock(duration);
  }
  toClock(duration) {
      let value:any;
      let seconds = duration.seconds();
```

```
        let minutes = duration.minutes();
        let hours = duration.hours();
        value = this.padder(hours) + ":" + this.padder(minutes) + ":" +
         this.padder(seconds);
        return value;
    }
    padder(value) {
        if(value < 10) {
            return "0" + value;
        } else {
            return value.toString();
        }
    }
  }
```

Let's understand it:

- `transform()`: This calls the `toClock` function for processing input, given through `value` arguments, and returns the process value.
- `toClock(duration)`: This function converts the given value in seconds, using `duration` arguments, to a digital clock. For example, if `121` is the value, the return value would be `00:02:01`.
- `padder(value)`: This adds an extra 0 to the value in front, if it is less than `10`.

ImControls component

The `ImControls` component is the component which has controls for playback like play, pause, next, and previous buttons, and the seek bar. It directly interacts with the `PlayerProvider`.

Inputs

The following is the input:

- `isPlaylist`: This is provided as the input to our `ImControls` so that we can enable playlist-related features.

Outputs

The following is the output:

- `next`: When the next button is clicked, it fires the next event as an output
- `back`: When the back button is clicked, it fires the back event as an output

The following code should be present in `im-controls.ts`:

```
// /app/components/im-controls/im-controls.ts
import {Component,Output, Input,EventEmitter,OnInit} from '@angular/core';
import {Control} from '@angular/common';
import {ImClock} from '../../pipes/clock';
import {Observable} from 'rxjs/Rx';
import {PlayerProvider} from '../../providers/player-provider/player-
provider';
@Component({
    selector: 'im-controls',
    pipes:[ImClock],
    templateUrl: 'build/components/im-controls/im-controls.html'
})

export class ImControls {
    @Input() isPlaylist:Boolean = false;
    @Output() next = new EventEmitter();
    @Output() back = new EventEmitter();
    seekbar:Control = new Control();
    maxSlide:number = 0;
    currentTime = 0;
    isPlaying:Boolean;
    timer:any;
    constructor(public player: PlayerProvider) {}

    ngOnInit() {
        this.reset();
        this.player.getDuration()
        .subscribe(value => {
            this.maxSlide = value;
        });
     }
```

`ngOnInit()` is where we we initialize our controls – we start the timer, get the duration of our file, and similar stuff. Let's look at some further code:

```
playBtn() {
   this.player.play();
   this.isPlaying = true;
   this.startTimer();
}
```

The `playBtn()` function is fired when the play button is clicked on the frontend. It starts the playback using the play function of the `PlayerProvider` and starts the timer for displaying the current playback time on the display. Let's look at some further code:

```
pauseBtn() {
    this.player.pause();
    this.isPlaying = false;
    this.stopTimer();
}
```

`pauseBtn()` pauses the playback of our media and stops the timer. Let's look at some further code:

```
startTimer() {
    let observable = Observable.interval(1000);
    this.timer = observable.subscribe(() =>{
        this.player.getCurrentPosition()
        .then(value => {
            this.currentTime = value;
            this.seekbar.updateValue(parseInt(value));
        });
    });
}
```

`startTimer()` creates an `Observable`, which fires a value each second, and we get the current position of our playback and display that on the screen. We also update our seek bar accordingly. Let's look at some further code:

```
stopTimer() {
    if(this.timer) {
        this.timer.unsubscribe();
    }
}
```

With `stopTimer()`, we unsubscribe to the `Observable` created in `startTimer`, to stop the timer. Let's look at some further code:

```
        seekTo(event) {
            this.player.seekTo(event.value);
        }
```

`seekTo(event)` is fired when the user moves the seek bar to a different position. It sets the media playback to time according to the position of the seek bar. Let's look at some further code:

```
        nextBtn() {
            let value = this.player.next();
            if(value !== -1) {
                this.reset();
                this.next.emit({});
            } else {
                console.log("No More File");
            }
        }
```

`nextBtn()` is called when the next button is clicked. It plays the next media in `playlist` and also fires a `next` event. Let's look at some further code:

```
        backBtn() {
            let value = this.player.back();
            if(value !== -1) {
                this.reset();
                this.back.emit({});
            } else {
                console.log("No More File");
            }
        }
```

`backBtn()` is called when the back button is clicked. It plays the previous media in the `playlist` and also fires a `back event`:

```
        reset() {
            this.player.getDuration()
            .subscribe(value => {
                this.maxSlide = value;
            });
            this.isPlaying = this.player.isPlaying;
            this.startTimer();
            this.player.isFinished()
            .then(value => {
                console.log(value);
                this.isPlaying = value;
            });
        }
    }
```

The `reset()` function resets the UI and variables of our controls.

It is important to note that we are using the custom `Pipe` here in this component, so we have to include the custom `Pipe` to the `pipes` array of the `@Component` decorator. In this case, it is `ImClock`.

The following code should be present in `im-controls.html`:

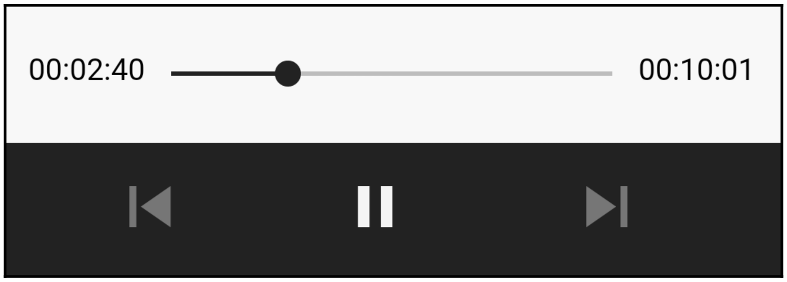
```
<!-- Template /app/components/im-controls/im-controls.html -->
<ion-footer class="im-controls">
  <ion-toolbar>
    <ion-range min="0" snaps="1" [max]="maxSlide"
     [ngFormControl]="seekbar" (ionChange)="seekTo($event)" dark>
      <ion-label range-left>{{currentTime | imClock:'seconds'}}</ion-
       label>
      <ion-label range-right>{{maxSlide | imClock:'seconds'}}</ion-
       label>
    </ion-range>
  </ion-toolbar>

  <ion-toolbar class="control-buttons" dark>
    <ion-row>
      <ion-col>
        <button [disabled]="!isPlaylist" center (click)="backBtn()"
         clear light><ion-icon name="skip-backward"></ion-icon>
         </button>
      </ion-col>
      <ion-col>
        <button *ngIf="isPlaying" clear light (click)="pauseBtn()">
         <ion-icon name="pause"></ion-icon></button>
        <button *ngIf="!isPlaying" clear light (click)="playBtn()">
         <ion-icon name="play"></ion-icon></button>
      </ion-col>
      <ion-col>
        <button [disabled]="!isPlaylist" (click)="nextBtn()" clear
         light><ion-icon name="skip-forward"></ion-icon></button>
      </ion-col>
    </ion-row>
  </ion-toolbar>
</ion-footer>
```

We are displaying two tool bars at the bottom. The primary toolbar at the bottom displays the button for controlling playback. The secondary toolbar shows an Ionic range element, the current time, and the total time of the media.

An important takeaway is the range element. An `ionChange` output event is only fired when the user changes the range value using sliding, not when the value of the seek bar is changed through programming. For example, we are changing the range value each second according to the current playing time, but it won't fire an `ionChange` event.

Our controls will look like the following screenshot:



Pages of our application

We have defined all the providers and services for our application. Let's define the pages of our application. We will have the following pages in our application:

- `BrowsePage`
- `PlaylistPage`
- `PlayerPage`

Defining the BrowsePage

The `BrowsePage` is used for browsing the media content in our mobile devices.

Basically, we are using this page recursively. We get the path to display the files and folders using the path `NavParams` on each display. If the path is not provided through `NavParams`, we will display the content of the `root` directory of our device.

The following code should be present in `browse.ts`:

```
// /app/pages/browse/browse.ts
import {Component} from '@angular/core';
import {LocalStorage,Storage, NavController, Platform, NavParams} from
'ionic-angular';
import {FileProvider} from '../../providers/file-provider/file-provider';
import {PlayerPage} from '../player/player';
import {PlayerProvider} from '../../providers/player-provider/player-
provider';
import {UtilProvider} from '../../providers/util-provider/util-provider';

declare var window:any;
@Component({
  templateUrl: 'build/pages/browse/browse.html',
})
export class BrowsePage {
  file:string;
  path:string;
  files:any;
  storage = new Storage(LocalStorage);
  constructor(public nav:NavController,
  public fp:FileProvider,
  public player: PlayerProvider,
  public platform: Platform,
  public params: NavParams,
  public util:UtilProvider) {
    this.path = params.get('path');
    platform.ready()
    .then(() => {
      let loader = this.util.getLoader();
      this.nav.present(loader);
      this.fp.getEntries(this.path)
      .then(content => {
        this.files = content;
        loader.dismiss();
      })
      .catch(error => {
        console.log("File system error", error);
      });
    });
  };
```

`constructor()` gets the file and folder entries of the given path using the `FileProvider` and sets those values to the `files` array. Let's look at some further code:

```
  open(file) {
    this.file = file;
    this.storage.set('current', JSON.stringify({file:file,
     type:"single"}));
    this.player.play(file.nativeURL, false);
    this.nav.push(PlayerPage, {type:'single'});
  }
```

The `open(file)` function is called when the **Play** button is clicked. It sets the current key in the `LocalStorage` to the given file, starts the media playback of the given file, and also opens the `PlayerPage`. Let's look at some further code:

```
  openDocument(file) {
    this.nav.push(BrowsePage, {path: file.nativeURL}, {animate: true});
  }
```

`openDocument(file)` opens the directory provided through the `file` variable by recursively opening the `BrowsePage`. It provides the `path` through `NavParams`. It is fired when the **Open** button is clicked. Let's look at some further code:

```
  openVideo(file) {
    let url = file.nativeURL;
    let options = {
      successCallback: function() {
        console.log("Video was closed without error.");
      },
      errorCallback: function(errMsg) {
        console.log("Error! " + errMsg);
      },
      orientation: 'landscape'
    };
    window.plugins.streamingMedia.playVideo(url,options);
  }
```

The `openVideo(file)` function starts the playback of the video file given in the `file` variable using the Cordova streaming plugin that we installed earlier. It opens a new `Native Page` or `Activity`, which shows playback controls of its own. Let's look at some further code:

```
  addToPlaylist(file) {
    let value = this.player.addToPlayList(file);
    let message;
    if(value === -1) {
      message = "Track is Already in Playlist."
    } else {
      message = "Track is Added to Playlist";
    }

    let toast = this.util.getToast(message);
```

```
        this.nav.present(toast);
      }
```

`addToPlaylist(file)` is called when the **Add to Playlist** button is clicked. It adds the `file` to the `playlist`, using the `addToPlaylist` function of `PlayerProvider`, and displays a `Toast` message accordingly. Let's look at some further code:

```
      isAudio(file) {
        let ext = file.name.split(".").pop();
        return this.player.isAudio(ext);
      }
```

`isAudio(file)` checks if a file is supported for audio playback and returns `true`; otherwise, it returns `false`. Let's look at some further code:

```
      isVideo(file) {
        let ext = file.name.split(".").pop();
        return this.player.isVideo(ext);
      }
    }
```

`isVideo(file)` checks if a file is supported for video playback and returns `true`; otherwise, it returns `false`.

Template

The following code should be present in `browse.html`:

```
    <!-- Template: /app/pages/browse/browse.html -->
    <ion-header>
      <ion-navbar dark>
        <button menuToggle><ion-icon name="menu"></ion-icon></button>
        <ion-title>
          Browse
        </ion-title>
      </ion-navbar>
    </ion-header>

    <ion-content class="home">
      <ion-list>
        <a ion-item *ngFor="let file of files">
          <ion-icon *ngIf="file.isFile" name="document" item-left></ion-
           icon>
          <ion-icon *ngIf="file.isDirectory" name="folder" item-left></ion-
           icon>
          {{file.name}}
          <button primary item-right *ngIf="isVideo(file)"
           (click)="openVideo(file)">Play Video</button>
          <button primary item-right *ngIf="!file.isFile"
           (click)="openDocument(file)">Open</button>
          <button primary item-right *ngIf="isAudio(file)"
           (click)="open(file)">Play</button>
          <button primary item-right *ngIf="isAudio(file)"
           (click)="addToPlaylist(file)">Add to PlayList</button>
        </a>
      </ion-list>
    </ion-content>
```

We are only showing the list of files and directories/folders and some buttons like **Play**, **Open**, and **Add to Playlist**, to play, open a folder, and add a file to the playlist, respectively.

You can use `virtualScroll` here, which is an Ionic 2 alternative for `collection-repeat`. There were some issues with it during the writing of this part, but it will be workable by the time the book is published. It is pretty straightforward to use it. For example, suppose we have an array of objects named `items` with this structure:

```
{name: 'any string here'}
```

We can display it using virtual scroll, like this:

```
<ion-list [virtualScroll]="items">
  <ion-item *virtualItem="#item">{{item.name}}</ion-item>
</ion-list>
```

Check the official documentation for virtual scroll at `http://ionicframework.com/docs/v2/api/components/virtual-scroll/VirtualScroll/`.

Defining PlaylistPage

The `PlaylistPage` is for showing and playing the playlist. The following code should be present in `playlist.ts`:

```
// /app/pages/playlist/playlist.ts
import {Component, ViewChild} from '@angular/core';
import {LocalStorage, Storage, ViewController, NavController, Platform,
NavParams, List} from 'ionic-angular';
import {PlayerPage} from '../player/player';
import {PlayerProvider} from '../../providers/player-provider/player-
provider';
```

```
@Component({
  templateUrl: 'build/pages/playlist/playlist.html',
})
export class PlayListPage {
    playlist:any = {};
    @ViewChild('playlist') list: List;
    storage = new Storage(LocalStorage);
    constructor(public nav:NavController, public player:PlayerProvider)
     {}

    ionViewWillEnter() {
        this.storage.get('playlist')
        .then(playlist => {
            console.log(playlist);
            if(playlist) {
                this.playlist = JSON.parse(playlist);
                console.log(this.playlist);
            }
        });
    }

    play() {
        this.player.startPlaylist();
        this.nav.push(PlayerPage, {type:'playlist'}, {animate:true});
    }

    deleteTrack(index) {
        this.playlist.tracks.splice(index,1);
        this.storage.set('playlist', JSON.stringify(this.playlist));
        this.list.closeSlidingItems();
    }
}
```

Let's try to understand the preceding code:

- `ionViewWillEnter()`: When we are about to enter the view/page, this function is called. Previously, it was called `ionPageWillEnter`, but it is now renamed. Each time this function is called, we get a playlist from the local storage and assign it to the `playlist` array.
- `play()`: This starts the playback of the playlist and shows the `PlayerPage`.
- `deleteTrack(index)`: This deletes the track given using an index value from the `playlist` array and local storage.

The following code should be present in `playlist.html`:

```
<!-- Template: /app/pages/playlist/playlist.html -->
<ion-header>
<ion-navbar dark>
  <button menuToggle><ion-icon name="menu"></ion-icon></button>
  <ion-title>
    Playlist
  </ion-title>
</ion-navbar>
</ion-header>

<ion-content>
  <div>
    <ion-item *ngIf="!playlist?.tracks">
      Playlist is Empty!
    </ion-item>
    <button *ngIf="playlist?.tracks"full clear
     (click)="play()">Play</button>
    <ion-list #list>
      <ion-item-sliding #slide *ngFor="let track of playlist?.tracks;
       let i = index">
        <ion-item>
            {{track.name}}
        </ion-item>

        <ion-item-options side="right">
          <button (click)="deleteTrack(i)">
            <ion-icon name="trash"></ion-icon>
          </button>
        </ion-item-options>
      </ion-item-sliding>
    </ion-list>
  </div>
</ion-content>
```

We are showing a **Play** button and a list of items. Each item is a sliding item. When we slide an item to the left, it shows the **Delete** button.

Defining PlayerPage

The `PlayerPage` is the page that shows the player controls and file name. The following code should be present in `player.ts`:

```
/ /app/pages/player/player.ts
import {Component} from '@angular/core';
import {ImControls} from '../../components/im-controls/im-controls';
import {PlayerProvider} from '../../providers/player-provider/player-
provider';
@Component({
    templateUrl: 'build/pages/player/player.html',
    directives: [ImControls]
})
export class PlayerPage {
    isPlaylist:Boolean;
    name:any;
    constructor(public player:PlayerProvider) {
        this.isPlaylist = this.player.isPlaylist;
    }

    ionViewWillEnter() {
        this.getFileName();
    }

    getFileName() {
        this.player.getFileName()
        .then(name => {
            this.name = name;
        });
    }
}
```

Let's understand the preceding code:

- `constructor()`: We initialize the `isPlaylist` variable using the `PlayerProvider`.
- `ionViewWillEnter()`: This function is called each time before entering a page. In it, we are calling the `getFileName` function to get the name of the file.
- `getFileName()`: This sets the name variable to the file name of the currently playing media by calling the `getFileName` method of the `PlayerProvider`.

It is also important to note that we are using our custom Angular component, `ImControls`, so we have to add that to the directives array in the `@Component` decorator.

The following code should be present in `player.html`:

```
<!-- Template: /app/pages/player/player.html -->
<ion-header>
  <ion-navbar dark>
    <button menuToggle><ion-icon name="menu"></ion-icon></button>
```

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

  <ion-content>
    <ion-card>
    <img src="img/back.jpg"/>
    <ion-card-content>
      <ion-card-title *ngIf="name">
        {{name}}
      </ion-card-title>
      <ion-card-title *ngIf="!name">
        Select a File to Play
      </ion-card-title>
    </ion-card-content>
  </ion-card>
  </ion-content>

  <im-controls *ngIf="name" [isPlaylist]="isPlaylist" (next)="getFileName()"
  (back)="getFileName()"></im-controls>
```

We are displaying an `ion-navbar`, an `ion-card` with a pre-stored image and a filename. At the bottom we are showing our `ImControls` component.

> We have stored an image at path `img/back.jpg`, which is displayed inside the card.

Running our app

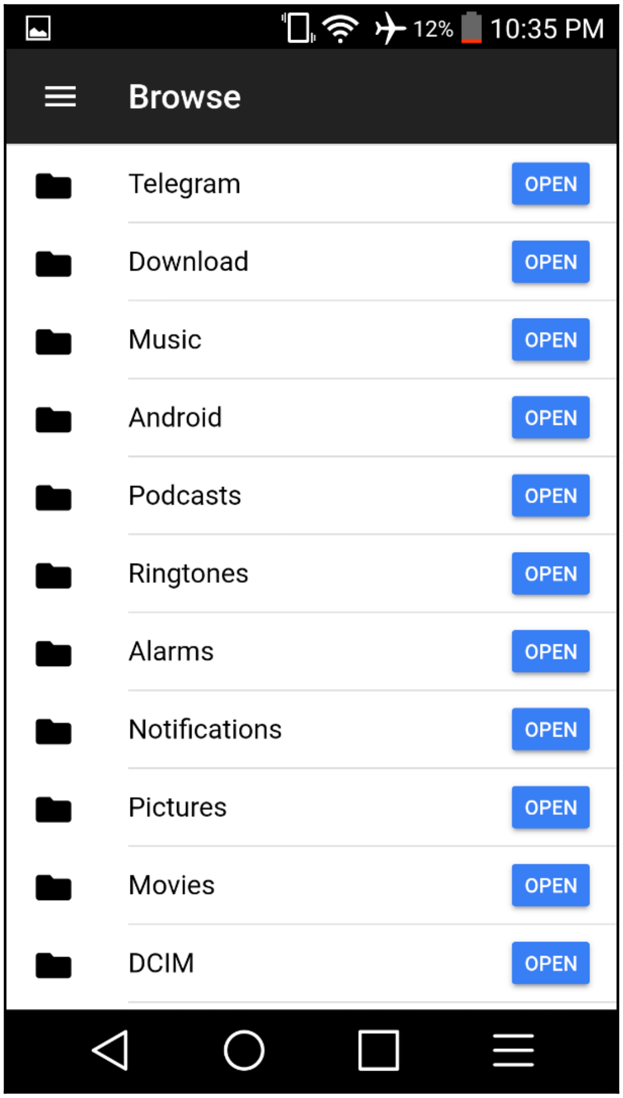
Let's run our application on an actual device using the following command:

```
ionic platform add android
ionic run android
```

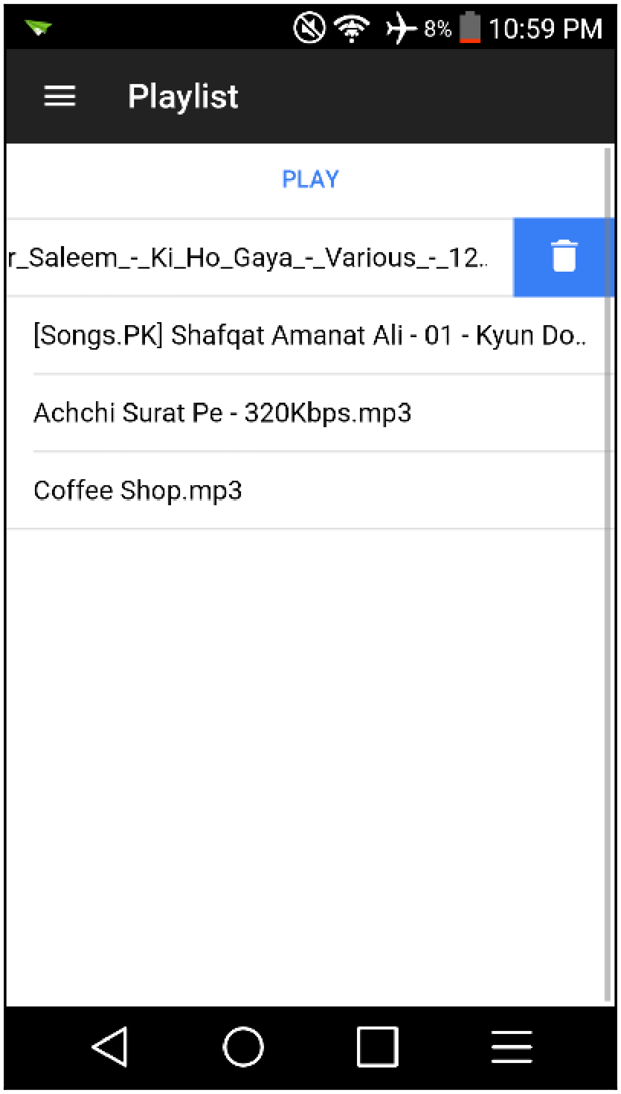
> It is worth noting that the Genymotion and Bluestacks emulators are treated as a real device, so you don't have to use the `emulate` command to run apps on these emulators. Just use the `run` command.

App screenshots

The following screenshot shows the `BrowsePage`:



The following screenshot shows the `PlaylistPage`:



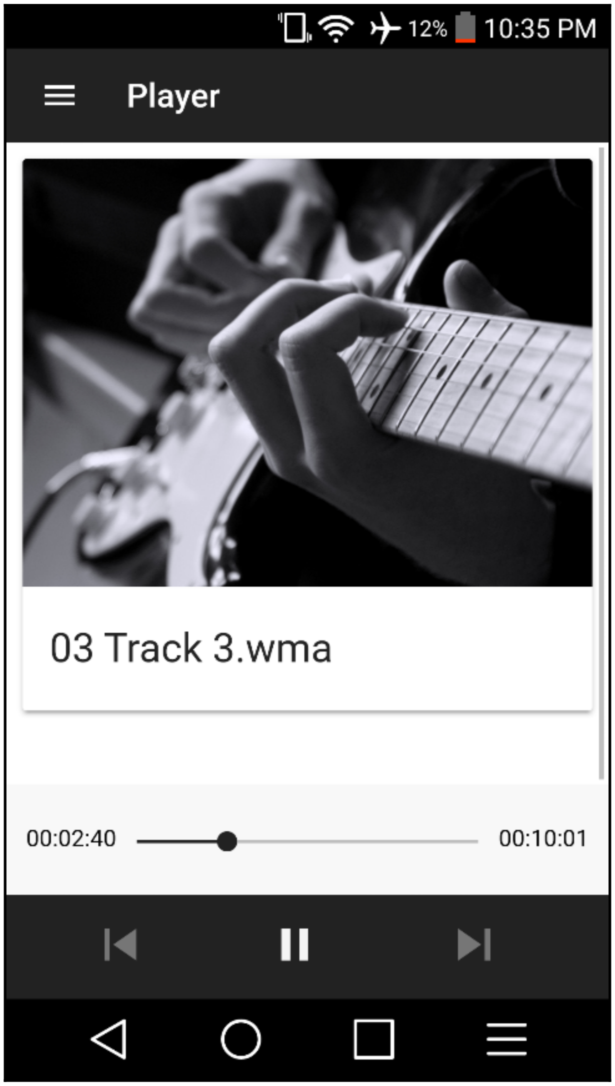
r_Saleem_-_Ki_Ho_Gaya_-_Various_-_12..

[Songs.PK] Shafqat Amanat Ali - 01 - Kyun Do..

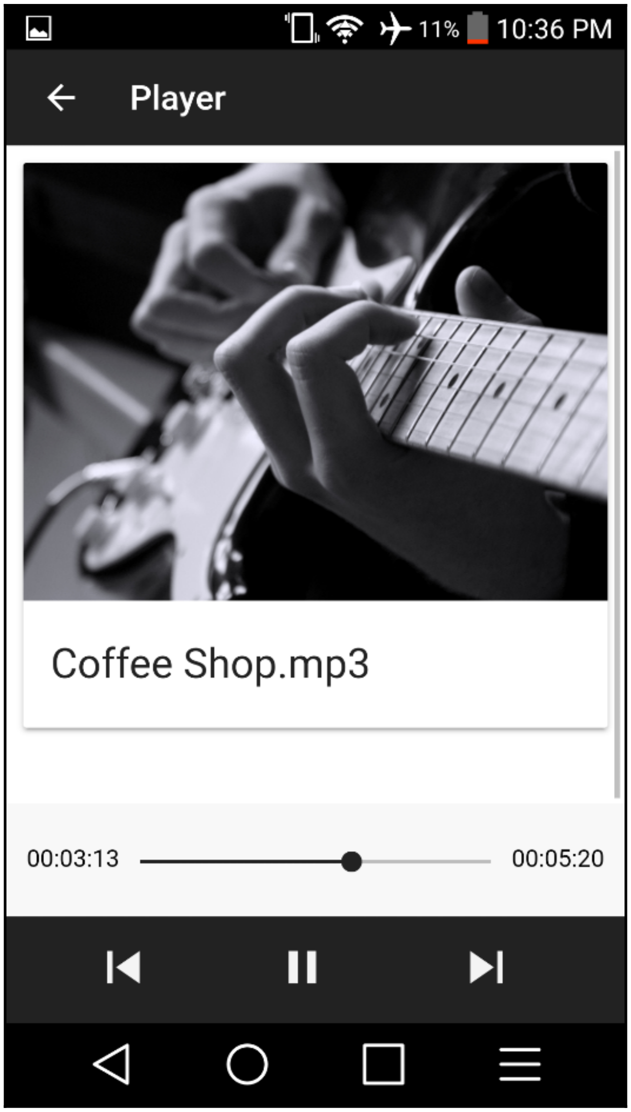
Achchi Surat Pe - 320Kbps.mp3

Coffee Shop.mp3

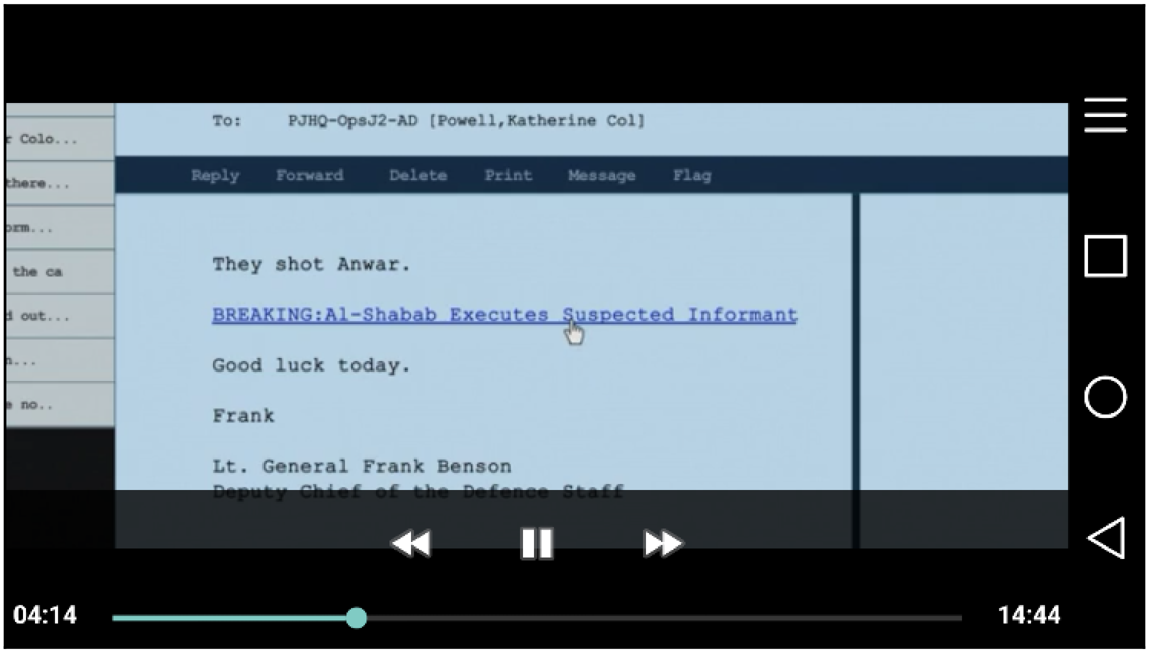
The following screenshot shows the `PlayerPage` without a playlist:



The following screenshot shows the `PlayerPage` with a playlist:



The following screenshot shows the video player:



They shot Anwar.

BREAKING:Al-Shabab Executes Suspected Informant

Good luck today.

Frank

Lt. General Frank Benson
Deputy Chief of the Defence Staff

Summary

In this chapter, we created a media player using Ionic that will support audio and video playback. We have used various Ionic features such as toast, loader, Ionic range, and others.

In the next chapter, we will create a social network mobile app using **Firebase** as our backend. This social network site will support friends, posts, images, profile page, and lots of other features.

7
Social App with Firebase

In this final chapter, we are going to create a social app using **Firebase**. Our application will be a totally client-side application, since we are using Firebase. All the major server-side functionality such as authentication, database, and binary storage will be handled by Firebase for us.

We will learn the following things:

- The Firebase as a backend using **AngularFire2** in Ionic
- Using Firebase storage for hosting binary stuff
- Understanding a neat way of using Firebase with Angular and Ionic
- Creating a social networking app

Defining the app

Our social app will be like *Twitter*. The first step for a user to use our app will be to create an account. We will support e-mail and password login. The user will have to choose a username and provide a name, e-mail, and password. Each user has a timeline where he sees the posts. He can follow other people, see their profile, and change his own profile pic, name, or about me section. Users can also post pictures.

Post logic

It is good to know how our posts actually work. Each user has a list of posts that are shown on his timeline. We will call them feeds. This feeds list is updated when either the user publishes a new post, or any person who our user is following publishes a post. This feeds list stores the reference to the actual post, which is inside a global posts list. Basically, when a user publishes a post, it is broadcast to all his followers. Take a look at the Firebase database structure below to know more about it.

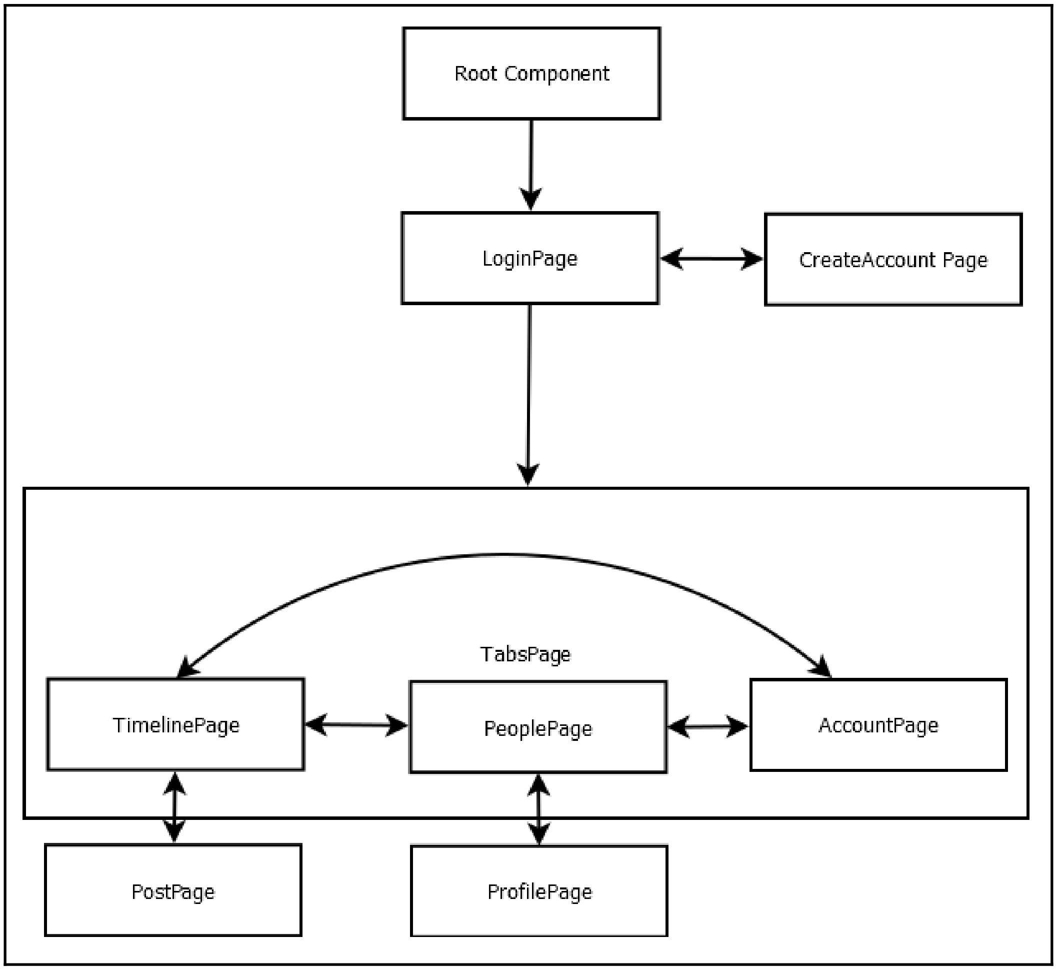
Functionalities

We will be including the following functionalities in our application:

- E-mail and password authentication
- List of all people on app
- Search users with username
- User profile with image, name, and about me
- Follow user
- Timeline with post feeds
- Posting images

App flow

App flow is really important when creating an application. It makes it easy for us to understand the working of application. The following figure shows the app flow for our application:



Let's try to understand the flow:

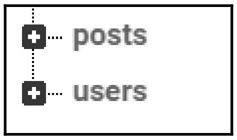
- **RootComponent**: This is the root Ionic component of our app. It is defined inside the `/app/app.ts` file. If the user is already logged in, the `TabsPage` will appear. Otherwise, it will show a `LoginPage`.
- **LoginPage**: This provides a user the ability to log in or go to the create account page.

- **CreateAccount page**: In the `CreateAccount` page, the user will create a new account.
- **TabsPage**: This is an **abstract** page. By abstract, it means it will never open alone and always have a child pages opened. This `TabsPage` has three tabs, called `Timeline`, `People`, and `Account`.
- **TimelinePage**: This is our timeline tab page. It shows the feeds for our user.
- **PostPage**: The post page allows the user to publish a new post.
- **PeoplePage**: This allows our user to search for and follow other people.
- **UserProfilePage**: The user can see the profile of other people.
- **AccountPage**: This allows a user to upload a profile picture and to log out from the application. When the user logs out, it will then go to `LoginPage`.

Firebase data structure

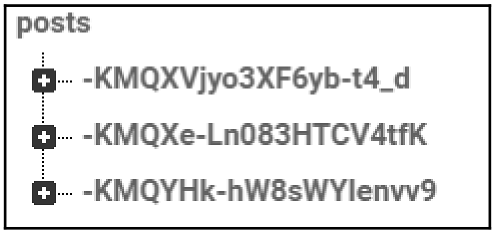
When developing Firebase apps, it is a good idea to develop the data structure first, using a flat data scheme, recommended for the Firebase data structure. Let's take a look at our data structure for our Firebase social app:

- At the root of the database, we have **posts** and **users** keys, shown as follows:



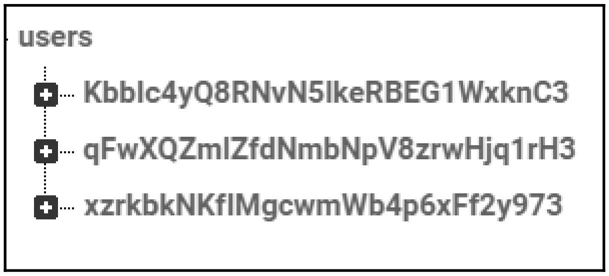
```
posts
users
```

- The **posts** key is list of posts, which our users have posted:



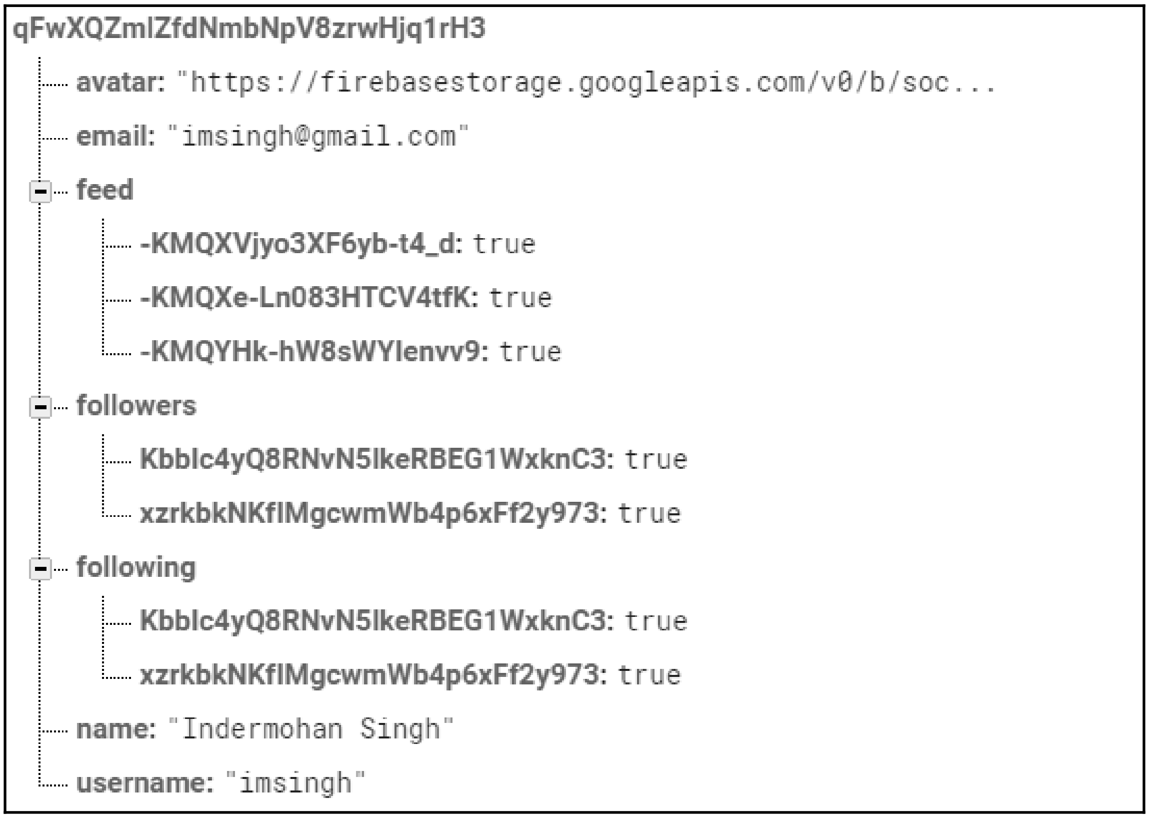
```
posts
  -KMQXVjyo3XF6yb-t4_d
  -KMQXe-Ln083HTCV4tfK
  -KMQYHk-hW8sWYlenvv9
```

- The **users** key is list of users of our application:



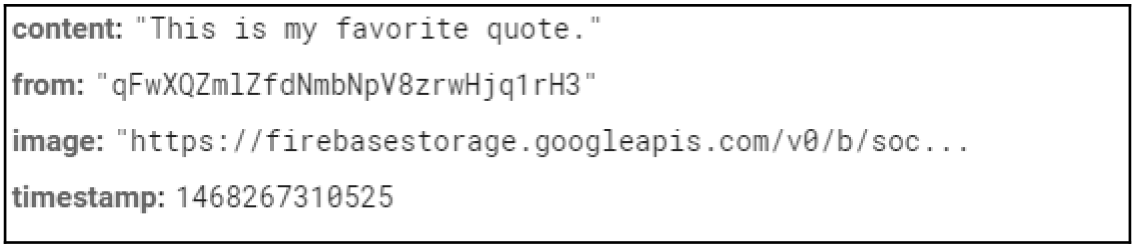
```
users
  + Kbblc4yQ8RNvN5lkeRBEG1WxknC3
  + qFwXQZmlZfdNmbNpV8zrwHjq1rH3
  + xzrkbkNKflMgcwmWb4p6xFf2y973
```

- Each user has the following data structure:



```
qFwXQZmlZfdNmbNpV8zrwHjq1rH3
  avatar: "https://firebasestorage.googleapis.com/v0/b/soc...
  email: "imsingh@gmail.com"
  - feed
      -KMQXVjyo3XF6yb-t4_d: true
      -KMQXe-Ln083HTCV4tfK: true
      -KMQYHk-hW8sWYlenvv9: true
  - followers
      Kbblc4yQ8RNvN5lkeRBEG1WxknC3: true
      xzrkbkNKflMgcwmWb4p6xFf2y973: true
  - following
      Kbblc4yQ8RNvN5lkeRBEG1WxknC3: true
      xzrkbkNKflMgcwmWb4p6xFf2y973: true
  name: "Indermohan Singh"
  username: "imsingh"
```

- Each user has the following keys.
    - **avatar**: This is the Firebase storage URL for the user's profile picture.
    - **email**: User's e-mail address.
    - **feed**: This is the list of post IDs that will show in the user's timeline. Basically, it is the list of post IDs of posts of the user them self and the posts of people who this user follows.
    - **followers**: This is the list of user IDs of the followers of this user.
    - **following**: This is the list of user IDs of the people who this user follows.
    - **name**: Name of our user.
    - **username**: Username of our user. We will use it for searching for people.
- Each user post has the following data structure:



```
content: "This is my favorite quote."
from: "qFwXQZmlZfdNmbNpV8zrwHjq1rH3"
image: "https://firebasestorage.googleapis.com/v0/b/soc...
timestamp: 1468267310525
```

- **content**: The text content of the post.
- **image**: The Firebase storage URL of the image of our post, if there is any.
- **from**: The user ID of the poster.
- **timestamp**: The timestamp value is when the user posts the post.

Scaffolding and setting up our app

Ionic CLI eases the process of scaffolding and setting up an app for us. So, we will be using Ionic CLI, which we installed earlier. We will start by scaffolding a blank application. Run the following command:

```
ionic start social-app blank --v2 --ts
```

Notice the `--v2` and `--ts` tag for scaffolding the Ionic 2 app based on **TypeScript**.

Using the `cd` command, go to the `firebase-chat` folder and run the `ionic serve` command, as the following shows:

```
ionic serve
```

The preceding command lets us check how the blank app looks.

Before we start writing the code for the app, we need to install some project-specific dependencies.

Installing Cordova plugins

The following command will install the Cordova plugins required for our app:

```
ionic plugin add cordova-plugin-camera
```

We will be using the Camera plugin to get pictures from our mobile device and then upload them to Firebase.

Installing Typings

The following command will install **Typings**:

```
npm install -g typings
```

Typings is the **Typescript Definition Manager**, which is required to install TypeScript definitions.

Installing Firebase and AngularFire2

The following command will install Firebase and AngularFire2:

```
npm install angularfire2 && firebase =save
```

Since we are using AngularFire2 for interacting with Firebase, we need to install both of these dependencies.

Installing ng2-moment

The following command will install `ng2-moment`:

```
npm install --save angular2-moment
```

The following command will install Moment for Typings:

```
typings install --save moment
```

Installing TypeScript definitions for Firebase

For Typings versions lower than 1, use `--ambient` instead of `--global` in the following command:

```
typings install file:node_modules/angularfire2/firebase3.d.ts --save --
global && typings install
```

It is temporary and will soon not be required.

Coding our app

Now we have everything set up to start working on our application. There are three important parts in the coding of our app; they are as follows:

- Defining our main `app.ts` file
- Our post component
- Creating providers/services for various functionalities
- Creating Ionic pages for various views

Defining app.ts

`app.ts` defines the root component and also bootstraps our app. The following should be present in `app.ts`:

```
/* /app/app.ts */
import {Component} from '@angular/core';
import {Platform, ionicBootstrap} from 'ionic-angular';
import { FIREBASE_PROVIDERS,
  defaultFirebase,
  firebaseAuthConfig,
  AuthProviders,
  AuthMethods } from 'angularfire2';
import {StatusBar} from 'ionic-native';
import {TabsPage} from './pages/tabs/tabs';
import {LoginPage} from './pages/login/login';
import {AuthProvider} from './providers/auth-provider/auth-provider';
import {UtilProvider} from './providers/utils';
import {UserProvider} from './providers/user-provider/user-provider';
import {SocialProvider} from './providers/social-provider/social-provider';

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

  private rootPage:any;

  constructor(private platform:Platform, authProvider: AuthProvider) {
    platform.ready().then(() => {
      StatusBar.styleDefault();
      let auth = authProvider.getAuth();
        auth.subscribe((result) => {
            if(result) {
              this.rootPage = TabsPage;
            } else {
              this.rootPage = LoginPage;
            }
        });
    });
  }
}

ionicBootstrap(MyApp, [FIREBASE_PROVIDERS,
  // Initialize Firebase app
  defaultFirebase({
    apiKey: "AIzaSyDugmwEYhnPFuy2mYT1xcTC8oINXv__540",
    authDomain: "social-app-545ef.firebaseapp.com",
    databaseURL: "https://social-app-545ef.firebaseio.com",
    storageBucket: "social-app-545ef.appspot.com",
  }), AuthProvider, UtilProvider, UserProvider, SocialProvider,
  firebaseAuthConfig({
    provider: AuthProviders.Password,
    method: AuthMethods.Password,
    remember: 'default',
    scope: ['email']
  })
])
```

In the constructor, we are using our `AuthProvider` to check if the user is logged in or not. If the user is logged in, we will show the `TabsPage`, and if the user is not logged in, we will show the `LoginPage`.

In the `ionicBootstrap` function, we bootstrap our application, initializing our Firebase app and configuring **Firebase Auth** to use e-mail/password authentication.

Providers for our application

We will have the following providers:

- `UtilProvider`
- `AuthProvider`
- `UserProvider`
- `SocialProvider`

Defining UtilProvider

`UtilProvider` serves as a *Swiss army knife* for us. We have abstracted various functionalities in this provider so that we don't have to repeat our code again and again. The following is the `UtilProvider` class:

```
import {Injectable, Inject} from '@angular/core';
import {Alert, Toast} from 'ionic-angular';
import {Camera} from 'ionic-native';

@Injectable()
export class UtilProvider {
    doAlert(title, message, buttonText) {
      let alert = Alert.create({
          title: title,
          subTitle: message,
          buttons: [buttonText]
      });
      return alert;
    };

    getToast(message) {
        let toast = Toast.create({
            message: message,
            duration: 1000
        });
        return toast;
    }

    dataURItoBlob(dataURI) {
        // convert base64 to raw binary data held in a string
        var byteString = atob(dataURI.split(',')[1]);
        var mimeString = dataURI.split(',')[0].split(':')[1].split(';')
         [0];
        var ab = new ArrayBuffer(byteString.length);
        var ia = new Uint8Array(ab);
        for (var i = 0; i < byteString.length; i++) {
            ia[i] = byteString.charCodeAt(i);
        }
        var bb = new Blob([ab], {type: mimeString});
        return bb;
    }

  // Get Picture
  getPicture(sourceType = 0, allowEdit = true) {
      let base64Picture;
      let options = {
          destinationType:0,
          sourceType: sourceType,
          encodingType:0 ,
          mediaType:0,
          allowEdit: allowEdit
      };
      let promise = new Promise((resolve, reject) => {
        Camera.getPicture(options).then((imageData) => {
            base64Picture = "data:image/jpeg;base64," + imageData;
            resolve(base64Picture);
        }, (error) => {
            reject(error);
        });
      });
      return promise;
  }
}
```

Let's try to understand the preceding code:

- `doAlert(title, message, buttonText)`: The `doAlert` function returns an `alert` component with the given `title`, `message`, and `buttonText`.
- `getToast(message)`: The `getToast` function returns Ionic `Toast` with a given message.
- `dataURItoBlob(dataURI)`: This converts the data given by `dataURI` to `Blob` and returns that `Blob`. We are using it for converting our image, provided by the camera to `Blob`.
- `getPicture(sourceType, allowEdit)`: This function is used to get an image from the camera. It returns a promise, which resolves to **base64** image data. `sourceType` is used for selecting the image source, like the camera or phone gallery. `allEdit`, if set to `true`, allows the user to crop the image before sending it to the app.

Defining AuthProvider

`AuthProvider` is used for authentication purpose in our application:

```
import {Injectable, Inject} from '@angular/core';
import {FirebaseAuth, FirebaseApp, FirebaseRef, AngularFire} from
'angularfire2';
import {LocalStorage, Storage} from 'ionic-angular';

@Injectable()
export class AuthProvider {
  local = new Storage(LocalStorage);
  constructor(public af:AngularFire, private fbAuth: FirebaseAuth) {
    this.fbAuth = fbAuth;
  }
  getAuth() {
    return this.af.auth;
  };
  signin(credentails) {
    return this.fbAuth.login(credentails);
  }
  createAccount(credentails) {
   return this.fbAuth.createUser(credentails);
  };
  logout() {
     this.fbAuth.logout();
    }
  }
```

Let's try to understand the preceding code:

- `getAuth()`: This returns the `Observable` of `auth`, which allows us to get the authentication state of the app.
- `signin(credentials)`: This function signs the user in to our application, with given `credentials`. `credentials` is an object with `username` and `password` keys. A `promise` is returned to the caller.
- `createAccount(credentials)`: This function creates a new user with the given credentials in Firebase Auth.
- `logout()`: This function logs out the currently logged-in user.

Defining UserProvider

`UserProvider` is used for providing facilities to the logged-in user of our application. The following is the `UserProvide` class:

```
import {Injectable, Inject} from '@angular/core';
import {AngularFire} from 'angularfire2';
import {LocalStorage, Storage} from 'ionic-angular';
import * as firebase from 'firebase';

@Injectable()
export class UserProvider {
  storage = new Storage(LocalStorage);
  constructor(public af:AngularFire) {
  }
  // Check if username is free
  isUsernameFree(username) {
    let promise = new Promise((res, rej) => {
      this.searchUser(username)
      .subscribe(value => {
        if(value.length === 0) {
          res(true)
        } else {
          res(false);
        }
      });
    });
    return promise;
  }
```

```
    // Save logged in user info in LocalStorage at userInfo key
    saveUser(userData) {
      this.storage.setJson("userInfo",userData);
    }

    // Get Current User's UID
    getUid() {
      let promise = new Promise((res, rej) => {
        this.storage.get('userInfo')
        .then(value => {
          let uid = JSON.parse(value).auth.uid;
          res(uid);
        });
      })
      return promise;
    }
    // Create User in Firebase
    createUser(userData) {
      return this.getUid()
      .then(uid => {
        let url = `/users/${uid}`;
        let user = this.af.database.object(url);
        return user.set(userData);
      });
    }

    updateProfile(obj) {
     return this.getUid()
      .then(uid => {
        return this.af.database.object(`/users/${uid}/`).update(obj);
      });
    }

    // upload profile picture
    uploadPicture(file) {
      return this.getUid()
      .then(uid => {
          let promise = new Promise((res,rej) => {
          let fileName = uid + '.jpg';
          let pictureRef =
           firebase.storage().ref(`/profile/${fileName}`);
          let uploadTask = pictureRef.put(file);

          uploadTask.on('state_changed', function(snapshot) {
          }, function(error) {
            rej(error);
          }, function() {
            var downloadURL = uploadTask.snapshot.downloadURL;
            res(downloadURL);
          });
        });

        return promise;
      });
    }

    // Search User with given username
    searchUser(username) {
      let query = {
        orderByChild: 'username'
      };
      // username is given
      if(username) {
        query['equalTo'] = username;
      }
      let users = this.af.database.list('/users', {
        query: query
      });
      return users;
    }

    // Get All Followers of a Logged In
    getFollowers() {
      return this.getUid()
      .then(uid => {
        return this.af.database.list(`/users/${uid}/followers`);
      });
    }
  }
```

Let's try to understand the preceding code:

- `isUsernameFree(username)`: This function returns a `promise`, which resolves to `true`, if the given username is free to use; otherwise, it resolves to `false`.
- `saveUser(userData)`: This stores the `userData` as a JSON in the `LocalStorage` at the `userInfo` key.
- `getUid()`: It returns a `Promise`, which resolve the `uid` of logged in user. It uses `LocalStorage` to get the `uid`.
- `createUser(userData)`: This basically adds the `userData` into our users list inside the Firebase data, which we described earlier in the Firebase data structure section.
- `updateProfile(obj)`: This function updates the profile information (user information) of the logged-in user, given by `obj`.

- `uploadPicture(file)`: This function uploads the profile picture of the logged-in user into Firebase storage. Profile pictures are stored at the `/images/uid.jpg` path in Firebase storage, where `uid` is the unique ID of the logged-in user.
- `searchUser(username)`: This searches the Firebase database with the given `username` and returns that user.
- `getFollowers()`: This returns the list of followers of the logged-in user.

Firebase storage allow us to upload binary files to the cloud without writing any server-side code. Take a look at `https://firebase.google.com/docs/storage/` for more information.

Defining SocialProvider

`SocialProvider` is used for various social functions, like publishing posts, following a user, and similar stuff. The following code is for the `SocialProvider` class:

```
import {Injectable} from '@angular/core';
import {UserProvider} from '../user-provider/user-provider';
import {AngularFire} from 'angularfire2';
import {Observable} from 'rxjs/Rx';

@Injectable()
export class SocialProvider {
    constructor(private userProvider: UserProvider, private af:
AngularFire) {}

    // Follow the user
    followUser(userData) {
        return this.userProvider.getUid()
        .then((uid:string) => {
            let otherUserID = userData.$key;
            let followingList =
             this.af.database.object(`/users/${uid}/following`);
            followingList.update({[otherUserID]: true});
            let followerList =
            this.af.database.object(`/users/${otherUserID}/followers`);
            return followerList.update({[uid]: true});
        });
    }

    // Get the Post Data given in postID
```

```
getPost(postID) {
    return this.af.database.object(`/posts/${postID}`);
}

// Post Image for a given post via postID
postImage(postID, imageData) {
    let promise = new Promise((res,rej) => {
        let fileName = postID + ".jpg";
        let uploadTask =
       firebase.storage().ref(`/posts/${fileName}`).put(imageData);
        uploadTask.on('state_changed', function(snapshot) {
        }, function(error) {
            rej(error);
        }, function() {
        var downloadURL = uploadTask.snapshot.downloadURL;
            res(downloadURL);
        });
    });
    return promise;
 }
// Update Post given via postID with given object value
updatePost(postID, obj) {
    return this.af.database.object(`/posts/${postID}`).update(obj);
}
// Upload Post
createPost(postData) {
    let uid;
    let posts = this.af.database.list('/posts');
    return this.userProvider.getUid()
    .then(userid => {
        uid = userid;
        postData.from = uid;
        postData.timestamp =
         firebase.database['ServerValue'].TIMESTAMP;
        return posts.push(postData).key;
    })
    .then(postKey => {
        let userFeed =
         this.af.database.object(`/users/${uid}/feed`);
        userFeed.update({[postKey]: true});

        this.af.database.list(`/users/${uid}/followers`)
        .subscribe(followers => {
           followers.forEach(follower => {
            this.af.database.object
            (`/users/${folower.$key}/feed`).update({[postKey]:
            true});
           });
```

```
                });

                return Promise.resolve(postKey);
            });
        }

        // Get User via uid
        getUser(uid) {
            return this.af.database.object(`/users/${uid}`);
        }
    }
```

Let's understand the preceding code:

- `followUser(userData)`: This allows the currently logged-in user to follow the user given by `userData`. It updates the `following` key of the currently logged-in user and also updates the `followers` key of other users in the Firebase database.
- `getPost(postID)`: This returns the post data of a post given by `postID` from the Firebase database.
- `postImage(postID, imageData)`: This function allows the user to upload the image in the post to Firebase storage. The path of images has the following structure: `/posts/postid.jpg`, where `postid` is the Firebase key of the post. It returns a `promise`, which resolves to the URL of the uploaded image.
- `updatePost(postID, obj)`: This allows us to update the post given by the `postID` to the value given by the `obj`.
- `createPost(postData)`: This function creates a new post inside the global `posts` object in our Firebase database. After that, it inserts the `postid` of the post to the logged-in user's feed, as well as to his followers' feeds.
- `getUser(uid)`: This gets the details of a user given by the `uid`.

Post component

The `Post` component renders the actual post on the screen. It has the following I/O structure.

Input

`feed` takes a data feed as input. Feed data is an object of form `{$key, $value}`. Here, the `$key` is the key of the actual post and `$value` is hardcoded as `true`. The following code should be present in `post.ts`:

```
// /app/component/post/post.ts //
import {Component, Input, OnInit, OnChanges} from '@angular/core';
import {TimeAgoPipe, FromUnixPipe} from 'angular2-moment';
import {SocialProvider} from '../../providers/social-provider/social-
provider';
@Component({
    selector: 'post',
    pipes: [TimeAgoPipe, FromUnixPipe],
    templateUrl: 'build/component/post/post.html'
})
export class PostCmp {
    @Input() feed;
    post;
    poster;
    constructor(public socialProvider:SocialProvider) {}

    ngOnInit() {
        let postID = this.feed.$key;
        this.post = this.socialProvider.getPost(postID);
        this.post
        .subscribe(value => {
            this.poster = this.socialProvider.getUser(value.from);
        });
    }
}
```

When the component is initialized, inside `ngOnInit`, we get the `postID` from the feed input. We then get the actual post data using the `SocialProvider.getPost` function and assign that to `this.post`. We also get the info of the poster of this post through the `socialProvider.getuser` function.

The following code should be present in `post.html`:

```
<!-template /app/component/post/post.html -->
<ion-card>
    <ion-item>
      <ion-avatar item-left>
        <img *ngIf="(poster | async)?.avatar" [src]="(poster |
         async)?.avatar">
        <img *ngIf="!(poster | async)?.avatar"
         src="images/defaultAvatar.png">
      </ion-avatar>
      <h2>{{(poster | async)?.name}}</h2>
      <p><p>
    </ion-item>

    <img *ngIf="(post| async)?.image" [src]="(post | async)?.image">

    <ion-card-content>
      <p>{{(post | async)?.content}}</p>
    </ion-card-content>

    <ion-item-divider>
      {{(post | async)?.timestamp/1000 | amFromUnix| amTimeAgo }}
    </ion-item-divider>
</ion-card>
```

We have displaced the avatar and name of the poster. If the poster doesn't have any avatar, we show a default picture. Then we show the image of the post, if there is any published, then the content of the post and time of the post. It is worth noting that we are using the `ng2moment` library to display the time.

Take a look at `ng2moment` at `https://github.com/urish/angular2-moment`.

Pages of our application

Now we have to define all the providers and services for our application. Let's define the pages of our application.

We will have the following pages in our application:

- `LoginPage`
- `CreateAccount Page`
- `TabsPage`
- `TimelinePage`
- `PostPage`
- `PeoplePage`
- `UserProfilePage`
- `AccountPage`

Defining LoginPage

`LoginPage` allows the user to log in to our application. The following code should be present in `login.ts`:

```
/* /app/pages/login/login.ts*/
import {Component} from '@angular/core';
import {NavController, Storage, LocalStorage} from 'ionic-angular';
import {TabsPage} from '../tabs/tabs';
import {FormBuilder, Validators} from '@angular/common';
import {validateEmail} from '../../validators/email';
import {AuthProvider} from '../../providers/auth-provider/auth-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {UtilProvider} from '../../providers/utils';
import {FirebaseAuth} from 'angularfire2';
import {Inject} from '@angular/core';
import {CreateAccount} from '../create-account/create-account';

@Component({
  templateUrl: 'build/pages/login/login.html'
})
export class LoginPage {
  loginForm;
    auth;
    storage = new Storage(LocalStorage);
    constructor(public nav:NavController,
      form:FormBuilder,
      auth: AuthProvider,
      @Inject(FirebaseAuth) public fbAuth: FirebaseAuth,
      public userProvider: UserProvider,
      public util: UtilProvider) {
        this.loginForm = form.group({
            email: ["",Validators.compose([Validators.required,
             validateEmail])],
            password:["",Validators.required]
        });
        this.auth = auth;
    }
  signin() {
      this.auth.signin(this.loginForm.value)
      .then((data) => {
          this.storage.set('userInfo', JSON.stringify(data));
          this.nav.push(TabsPage);
      }, (error) => {
          console.log(error);
          let errorMessage = "Enter Correct Email and Password";
          let alert = this.util.doAlert("Error",errorMessage,"Ok");
          this.nav.present(alert);
      });
    };
    createAccount() {
        this.nav.push(CreateAccount);
    };
}
```

We have created a `loginForm` using the `FormBuilder`, with e-mail and password fields. We are also using the `validateEmail` validator for validating the e-mail (from `Chapter 1`, *Chat App with Firebase*).

The `signin` function is attached to the **Sign In** button in view, and it signs in the user and saves the user information inside `Localstorage` at the `userInfo` key. If there is any error, it is shown through an `alert` message.

The `createAccount` function is attached to the **Create Account** button in the view, and it takes the user to the `CreateAccount` page.

The following code should be present in `login.html`:

```
<!-- template /app/pages/login/login.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title text-center>Login</ion-title>
    </ion-navbar>
</ion-header>

<ion-content class="padding">
    <form [ngFormModel]="loginForm">
        <ion-list>
            <div padding text-center>
```

```
                <img src="images/logo.png">
            </div>

            <ion-item>
                <ion-label floating>Email</ion-label>
                <ion-input type="text" ngControl="email"></ion-input>
            </ion-item>

            <ion-item>
                <ion-label floating>Password</ion-label>
                <ion-input type="password" ngControl="password"></ion-
                 input>
            </ion-item>
        </ion-list>
        <div padding>
            <button primary block (click)="signin()"
             [disabled]="!loginForm.valid">Sign In</button>
        </div>
        <div padding>
            <button full clear secondary (click)="createAccount()">
                <ion-icon name="person"></ion-icon>
                Create an Account
            </button>
        </div>
    </form>
</ion-content>
```

An important thing to notice in the template is that the **Sign In** button is only enabled if the `userForm` is valid.

Defining the CreateAccount page

The `CreateAccount` page allows the user to create a new account. The following code should be present in `create-account.ts`:

```
/* /app/pages/create-account/create-account.ts*/
import {Component} from '@angular/core';
import {NavController, Storage, LocalStorage} from 'ionic-angular';
import {TabsPage} from '../tabs/tabs';
import {FormBuilder, Validators, Control} from '@angular/common';
import {validateEmail} from '../../validators/email';
import {AuthProvider} from '../../providers/auth-provider/auth-provider';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {UtilProvider} from '../../providers/utils';
import {FirebaseAuth} from 'angularfire2';
import {Inject} from '@angular/core';
```

```
@Component({
    templateUrl: 'build/pages/create-account/create-account.html'
})

export class CreateAccount {
  createForm;
    auth;
    password:Control;
    storage = new Storage(LocalStorage);
    constructor(public nav:NavController,
      form:FormBuilder,
      auth: AuthProvider,
      @Inject(FirebaseAuth) public fbAuth: FirebaseAuth,
      public userProvider: UserProvider,
      public util: UtilProvider) {
        this.password = new
         Control("",Validators.compose([Validators.required,
         Validators.minLength(6)]));
        this.createForm = form.group({
            username: ["", Validators.required],
            name: ["", Validators.required],
            email: ["",Validators.compose([Validators.required,
             validateEmail])],
            password:this.password,
            repass: ["", Validators.required]
        });
        this.auth = auth;
    }

    createAccount() {
        let {password, repass, username, name, email} =
         this.createForm.value;
        username = username.toLowerCase();
        if(password !== repass) {
            let alert = this.util.doAlert("Error", "Password doesn't
             matched", "Ok");
            this.nav.present(alert);
        } else {
            this.userProvider.isUsernameFree(username)
            .then(value => {
                if(value === false) {
                    let alert = this.util.doAlert("Error", "Username
                     not available", "Ok");
                    this.nav.present(alert);
                } else {
                    // Create Account
                    this.auth.createUser({email: email, password:
                     password})
                    .then(value => {
                        this.userProvider.saveUser(value);
                        this.userProvider.createUser({email: email,
                         username: username, name: name});
                    });
                }
            });
        }
    }
}
```

In the constructor, we have created `createForm` using the `FormBuilder` and we have used various required validators.

The `createAccount` function is attached to the **Create Account** button on the view. When it is fired, it checks if the chosen `username` is free or not. If the username is free, we create our user in Firebase Auth using the `createAccount` function of `AuthProvider`. When our user is created in Firebase Auth, we save its details in `LocalStorage` and also in the Firebase database.

The following code should be present in `create-account.html`:

```
<!-- template /app/pages/create-account/create-account.html -->
<ion-header>
    <ion-navbar primary>
        <ion-title>Create Account</ion-title>
    </ion-navbar>
</ion-header>

<ion-content>
    <form [ngFormModel]="createForm">
        <ion-list>
            <ion-item>
                <ion-label floating>Username</ion-label>
                <ion-input type="text" ngControl="username"></ion-
                 input>
            </ion-item>
            <ion-item>
                <ion-label floating>Name</ion-label>
                <ion-input type="text" ngControl="name"></ion-input>
            </ion-item>

            <ion-item>
                <ion-label floating>Email</ion-label>
                <ion-input type="text" ngControl="email"></ion-input>
            </ion-item>
            <ion-item>
```

```
                <ion-label floating>Password</ion-label>
                <ion-input type="password" ngControl="password"></ion-
                 input>
            </ion-item>
            <ion-item text-wrap *ngIf="password.dirty &&
             !password.valid">
                Password should have at least 6 characters.
            </ion-item>
            <ion-item>
                <ion-label floating>Enter Password again</ion-label>
                <ion-input type="password" ngControl="repass"></ion-
                 input>
            </ion-item>

        </ion-list>
        <div padding>
            <button primary block (click)="createAccount()"
             [disabled]="!createForm.valid">Create Account</button>
        </div>
    </form>
</ion-content>
```

Defining TabsPage

The following code should be present in `tabs.ts`:

```
/* /app/pages/tabs/tabs.ts*/
import {Component} from '@angular/core'
import {TimelinePage} from '../timeline/timeline';
import {PeoplePage} from '../people/people';
import {AccountPage} from '../account/account';

@Component({
  templateUrl: 'build/pages/tabs/tabs.html'
})
export class TabsPage {

  private tab1Root: any;
  private tab2Root: any;
  private tab3Root: any;

  constructor() {
    this.tab1Root = TimelinePage;
    this.tab2Root = PeoplePage;
    this.tab3Root = AccountPage;
  }
}
```

```
<!-- template /app/pages/tabs/tabs.html -->
<ion-tabs primary>
  <ion-tab [root]="tab1Root" tabTitle="Timeline" tabIcon="pulse"></ion-tab>
  <ion-tab [root]="tab2Root" tabTitle="People" tabIcon="people"></ion-tab>
  <ion-tab [root]="tab3Root" tabTitle="Account" tabIcon="settings"></ion-
tab>
</ion-tabs>
```

We have created our tabs page with three tabs: timeline, people, and account.

Defining TimelinePage

The following code should be present in `timeline.ts`:

```
/* /app/pages/timeline/timeline.ts*/
import { Component } from '@angular/core';
import { NavController, Modal } from 'ionic-angular';
import { AngularFire, FirebaseListObservable } from 'angularfire2';
import { PostPageModal } from '../post/post';
import { SocialProvider } from '../../providers/social-provider/social-
provider';
import { UserProvider } from '../../providers/user-provider/user-provider';
import { PostCmp } from '../../component/post/post';
import * as firebase from 'firebase';

@Component({
  templateUrl: 'build/pages/timeline/timeline.html',
  directives: [PostCmp]
})
export class TimelinePage {
  posts:any;
  userInput;
  feeds:any = [];
  constructor(private navController: NavController, private
   socialProvider: SocialProvider, private userProvider:
   UserProvider,private af: AngularFire) {
    this.userProvider.getUid()
    .then(uid => {
       firebase.database().ref(`/users/${uid}/feed`)
       .on('child_added', (snapshot) => {
         this.feeds.unshift({$key:snapshot.key, $value:
          snapshot.val()});
       });
    });
  }

  openPost() {
    let modal = Modal.create(PostPageModal);
    this.navController.present(modal);
  }
}
```

The timeline page allows a user to see their feed. We are basically getting the list of feed from the user's data and providing feed information to the post component. Then, the post component renders the post as described in the *Post component* section of this book.

The `openPost` function opens the `PostPage`, which allow users to publish a new post.

The following code should be present in `timeline.html`:

```
<!-- template /app/pages/timeline/timeline.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>Timeline</ion-title>
  </ion-navbar>
</ion-header>

<ion-content class="home" fullscreen>
  <post *ngFor="let feed of feeds" [feed]="feed">
  </post>
</ion-content>

<ion-footer>
  <button (click)="openPost()" fab fab-right fab-bottom><ion-icon
name="add"></ion-icon></button>
</ion-footer>
```

Defining PostPage

`PostPage` allows the user to publish a new post. Users can also post an image. The following code should be present in `post.ts`:

```
/* /app/pages/post/post.ts*/
import {Component} from '@angular/core';
import {ViewController, NavController, ActionSheet} from 'ionic-angular';
import {SocialProvider} from '../../providers/social-provider/social-
provider';
import {UtilProvider} from '../../providers/utils';
@Component({
    templateUrl: 'build/pages/post/post.html'
})
export class PostPageModal {
    postContent:string;
    image = null;
    blobImage;
    constructor(
        private viewController:ViewController,
        private navController: NavController,
        private socialProvider: SocialProvider,
        private util:UtilProvider) {
    }

    dismiss() {
        this.viewController.dismiss();
    }

    sendPost() {
        let obj = {content: this.postContent, image:this.image};
        this.socialProvider.createPost(obj)
        .then((postKey) => {
            console.log(postKey);
            // if Image is Added
            if(this.blobImage) {
                this.socialProvider.postImage(postKey, this.blobImage)
                .then(url => {
                    this.socialProvider.updatePost(postKey,
                     {image:url});
                });
            }
            this.reset();
            this.dismiss();
        });
    }

    addImage() {
        this.presentPictureSource()
        .then(source => {
            let sourceType:number = Number(source);
            return this.util.getPicture(sourceType, false);
        })
        .then(imageData => {
            this.blobImage = this.util.dataURItoBlob(imageData);
        });
    }

    presentPictureSource() {
        let promise = new Promise((res, rej) => {
            let actionSheet = ActionSheet.create({
            title: 'Select Picture Source',
            buttons: [
                { text: 'Camera', handler: () => { res(1); } },
                { text: 'Gallery', handler: () => { res(0); } },
                { text: 'Cancel', role: 'cancel', handler: () => {
                 rej('cancel'); } }
            ]
            });
            this.navController.present(actionSheet);
        });
        return promise;
    }

    reset() {
        this.postContent = "";
        this.image = null;
        this.blobImage = null;
    }
}
```

This is basically an Ionic modal. Here, the `dismiss` function is attached to the **X** button in the toolbar. It closes the modal.

The `sendPost` function publishes the post and saves its content to global, the `posts` list in the Firebase database. If there is an image posted along with text, it is uploaded to Firebase's storage.

The `addImage` function saves the user's selected image to the `this.blobImage` variable.

The `presentPictureSource` function shows an `ActionSheet` with camera, gallery, and cancel as options. These are the sources of images in a mobile device.

The `reset` function resets the variable to null or empty values.

The following code should be present in `post.html`:

```
<!-- template /app/pages/post/post.html -->
<ion-header>
  <ion-toolbar primary>
    <ion-title>
      Write Post
    </ion-title>
    <ion-buttons end>
      <button (click)="addImage()"><ion-icon name="image"></ion-icon>
      </button>
      <button (click)="dismiss()">
        <ion-icon name="close"></ion-icon>
      </button>
    </ion-buttons>
  </ion-toolbar>
```

```
</ion-header>

<ion-content class="post-page">
  <ion-item>
    <ion-textarea [(ngModel)]="postContent" rows="20" max-length="140"
     placeholder="Enter your post"></ion-textarea>
  </ion-item>
</ion-content>

<ion-footer>
    <ion-toolbar>
        <button (click)="sendPost()" full>Post</button>
    </ion-toolbar>
</ion-footer>
```

In the toolbar, we are showing a close button and image button. **X** (close button) is used for dismissing the modal, and the image button is for uploading an image.

In `ion-content`, we have `ion-textarea`, where the user can write their post, and in the footer, we have the **Post** button for sending the post.

The following code should be present in `post.scss`:

```
/* stylesheet /app/pages/post/post.scss */
.post-page {
    ion-item {
        height: 100%;
        width:100%;
    };

    ion-item textarea {
        height: 100%;
        width:100%;
        margin-top:10%;
    }
}
```

With this stylesheet, we are setting the `textarea` component's height and width to `100%` so that it covers the whole `ion-content` area.

Defining PeoplePage

`PeoplePage` allows users to search for other users and follow them. The user will see a list of people and profiles, and a follow button for each person. The following code should be present in `people.ts`:

```
/* /app/pages/people/people.ts*/
import {Component} from '@angular/core';
import {NavController} from 'ionic-angular';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {SocialProvider} from '../../providers/social-provider/social-
provider';
import {UtilProvider} from '../../providers/utils';
import {FirebaseListObservable} from 'angularfire2';
import {UserProfilePage} from '../user-profile/user-profile';

@Component({
  templateUrl: 'build/pages/people/people.html'
})
export class PeoplePage {
  users;
  uid;
  followersObservable:FirebaseListObservable<any>;
  followers;
  constructor(private navController: NavController,
              private userProvider:UserProvider,
              private socialProvider: SocialProvider,
              private util: UtilProvider) {
    this.userProvider.getUid()
    .then(uid => {
      this.uid = uid;
    });
    this.users = this.userProvider.searchUser("");
  }

  getUser(ev) {
    let username = ev.target.value;
    this.users = this.userProvider.searchUser(username);
  }

  followUser(user) {
    this.socialProvider.followUser(user)
    .then(()=> {
      let toast = this.util.getToast("You are now following " +
       user.name);
      this.navController.present(toast);
    });
  }
```

```
    userProfile(user) {
      console.log(user);
      this.navController.push(UserProfilePage, {uid:user.$key});
    }
  }
```

In the constructor, we are getting the `uid` of the logged-in user. Then we are getting a list of all users, by providing an empty `username` to `this.userProvider.searchUser`.

The `getUser` function shows the user with the provided `username` given through the `ion-searchbar`.

`followUser(user)` is attached to the **Follow** button in the view and allows the user to follow another user, given by the `user` argument. It also shows a nice Ionic `Toast` when you follow the user.

`userProfile(user)` is attached to the **Profile** button in the view. It opens the `UserProfilePage` of a given user. The following code should be present in `people.html`:

```
<!-- template /app/pages/people/people.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>
      People
    </ion-title>
  </ion-navbar>
</ion-header>

<ion-content class="about">
  <ion-searchbar (ionInput)="getUser($event)" placeholder="Search with
   username">
  </ion-searchbar>

  <ion-list>
    <ion-item text-wrap *ngFor="let user of users | async"
     [hidden]="uid === user?.$key">
      <ion-avatar item-left>
        <img *ngIf="user?.avatar" [src]="user?.avatar">
        <img *ngIf="!user?.avatar" src="images/defaultAvatar.png">
      </ion-avatar>
      <h2>{{(user)?.name}}</h2>
      <button item-right secondary
       (click)="userProfile(user)">Profile</button>
      <button item-right (click)="followUser(user)">Follow</button>
    </ion-item>
  </ion-list>
</ion-content>
```

Nothing fancy here, but an important point to notice is that we are showing all users except the logged-in user. We are using the logged-in user's `uid` to hide their data.

We are showing a list of people with their avatar, name, and a **Follow** and **Profile** button.

Defining UserProfilePage

`UserProfilePage` shows the profile page of any user of our app. The following code should be present in `user-profile.ts`:

```
/* /app/pages/user-profile/user-profile.ts*/
import {Component} from '@angular/core';
import {NavController, ActionSheet, NavParams} from 'ionic-angular';
import {UtilProvider} from '../../providers/utils';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {SocialProvider} from '../../providers/social-provider/social-
provider';

@Component({
  templateUrl: 'build/pages/user-profile/user-profile.html'
})
export class UserProfilePage {
  user = {};
  profile = {};
  uid:String;
  constructor(private navController: NavController,  private util:
   UtilProvider, private params: NavParams, private socialProvider:
   SocialProvider) {
      this.uid = params.get('uid');
      console.log(this.uid);
      this.socialProvider.getUser(this.uid)
      .subscribe(user => {
        this.user = user;
        console.log(user);
      });
  }

  followUser(user) {
       this.socialProvider.followUser(user);
  }
}
```

We are getting the `uid` of the user from `NavParams`, and then we are getting all its detail using the `getUser` function of `SocialProvider`.

`followUser` allow the logged in user to follow this user.

The following code should be present in `user-profile.html`:

```
<!-- template /app/pages/user-profile/user-profile.html -->
<ion-header>
  <ion-navbar primary>
    <ion-title>
      {{user?.username}}
    </ion-title>
  </ion-navbar>
</ion-header>

<ion-content>
  <ion-card>
      <img *ngIf="user?.avatar" [src]="user?.avatar">
      <img *ngIf="!user?.avatar" src="images/defaultAvatar.png">
  </ion-card>

  <ion-card>
        <ion-item text-wrap>{{user?.username}}
         <ion-note item-right>
           Username
         </ion-note>
        </ion-item>
        <ion-item text-wrap><ion-icon secondary item-left
         name="person"></ion-icon>{{user?.name}}</ion-item>
        <ion-item text-wrap><ion-icon secondary item-left name="mail">
</ion-icon>{{user?.email}}</ion-item>
        <ion-item text-wrap *ngIf="user?.about"><ion-icon secondary
         item-left name="create"></ion-icon><p *ngIf="user?.about">
          {{user?.about}}</p></ion-item>
        <div padding>
            <button full primary
             (click)="followUser(user)">Follow</button>
        </div>
</ion-card>
</ion-content>
```

We are showing an image of the user, his/her username, name, e-mail, and about me section. Finally,we are also showing a **Follow** button.

Defining AccountPage

`AccountPage` allows the logged-in user to change his avatar pic, name, and about me sections.

The following code should be present in `account.ts`:

```
/* /app/pages/account/account.ts*/
import {Component} from '@angular/core';
import {NavController, ActionSheet} from 'ionic-angular';
import {FirebaseAuth} from 'angularfire2';
import {UtilProvider} from '../../providers/utils';
import {UserProvider} from '../../providers/user-provider/user-provider';
import {SocialProvider} from '../../providers/social-provider/social-
provider';

@Component({
  templateUrl: 'build/pages/account/account.html'
})
export class AccountPage {
  user = {};
  profile:Object = {};
  constructor(private navController: NavController, private
   afAuth:FirebaseAuth, private util: UtilProvider, private
   userProvider: UserProvider, private socialProvider: SocialProvider)
   {
    this.userProvider.getUid()
    .then(uid => {
      this.socialProvider.getUser(uid)
      .subscribe(user => {
        this.user = user;
      });
    });
  }

  logout() {
    this.afAuth.logout();
  }

  updatePicture() {
    this.presentPictureSource()
    .then(source => {
      let sourceType:number = Number(source);
      return this.util.getPicture(sourceType);
    })
    .then(imageData => {
      var blobImage = this.util.dataURItoBlob(imageData);
      return this.userProvider.uploadPicture(blobImage);
    })
    .then(imageURL => {
      return this.userProvider.updateProfile({avatar: imageURL});
    })
    .then(()=> {
      let toast = this.util.getToast('Your Picture is updated');
      this.navController.present(toast);
    });
  }

  presentPictureSource() {
    let promise = new Promise((res, rej) => {
        let actionSheet = ActionSheet.create({
          title: 'Select Picture Source',
          buttons: [
            { text: 'Camera', handler: () => { res(1); } },
            { text: 'Gallery', handler: () => { res(0); } },
            { text: 'Cancel', role: 'cancel', handler: () => {
             rej('cancel'); } }
          ]
        });
        this.navController.present(actionSheet);
    });
    return promise;
  }

  updateProfile() {
    let toast = this.util.getToast("Your Profile is updated");
    this.userProvider.updateProfile({name: this.user['name'], about:
     this.user['about']})
    .then(()=> {
      this.navController.present(toast);
    });
  }
}
```

In the constructor, we are getting the details of the logged-in user.

The `logout` function logs the user out. It seems easy, right?

`updatePicture` allows the user to update his/her profile picture. It gets the picture from a device, uploads it to Firebase's storage, and saves its entry to the Firebase database.

`presentPictureSource` shows an action sheet with image sources.

`updateProfile` actually updates the user's profile information in the Firebase database.

The following code should be present in `account.html`:

```
  <!-- template /app/pages/account/account.html -->
  <ion-header>
    <ion-navbar primary>
      <ion-title>
```

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

  <ion-content>
    <ion-card>
        <img *ngIf="user?.avatar" [src]="user?.avatar">
        <img *ngIf="!user?.avatar" src="images/defaultAvatar.png">
        <div padding>
          <button full primary (click)="updatePicture()">Update
           Picture</button>
        </div>
    </ion-card>
    <ion-card>
      <ion-item>
          <ion-label floating>Username</ion-label>
          <ion-input type="text" disabled=true [value]="user?.username">
           </ion-input>
      </ion-item>

      <ion-item>
          <ion-label floating>Name</ion-label>
          <ion-input type="text" [(ngModel)]="user.name"></ion-input>
      </ion-item>

      <ion-item>
          <ion-label floating>About me</ion-label>
          <ion-textarea type="text" [(ngModel)]="user.about"></ion-
           textarea>
      </ion-item>
      <div padding>
          <button full primary (click)="updateProfile()">Update
           Profile</button>
      </div>
    </ion-card>

  <div padding>
    <button full dark (click)="logout()">Logout</button>
  </div>
  </ion-content>
```

Adding style

Styles are defined in the `app/theme` folder. Inside this folder, there is an `app.variable.scss` file. In this file, Ionic has defined the default colors. Let's change the color values of the primary and secondary Ionic colors to this:

```
primary:    #3949AB,
secondary:  #E91E63,
```

Also, let's add the following stylesheet of the post component in the `app.core.scss` file:

```
@import "../pages/post/post";
```

Running our app

So, we have done every part so far. But we haven't yet tested our app in any actual devices, so let's build our application for a mobile device. This can be done by the following commands:

```
ionic platform add android
ionic run android
```

Further improving our app

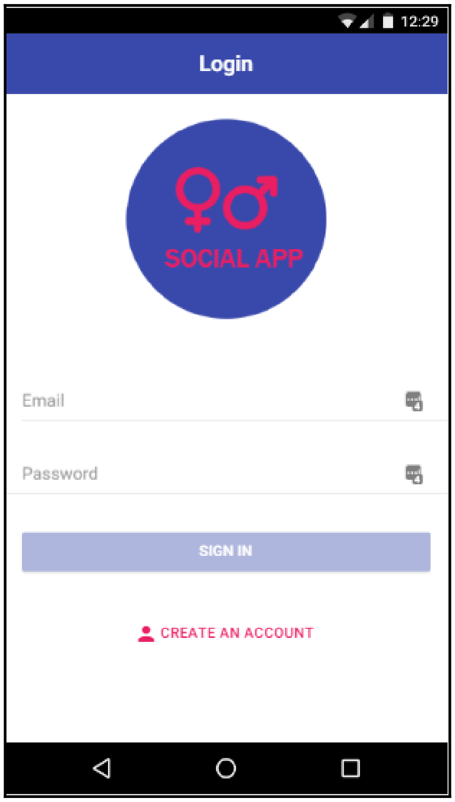
You can add lots of features in this app. I have limited pages, so I can't add lots of functionalities, but here are some hints for you:

- **Allow post editing**: You can allow posts to be edited. In fact, I have written most of the code for it. There is an `updatePost` function in `SocialProvider`. You can use it to edit posts.
- **Unfollow user**: It is fairly simple to add an unfollow feature. You need to get the list of users you are following and then show an unfollow button for these users in the `AccountPage` or the `PeoplePage`.

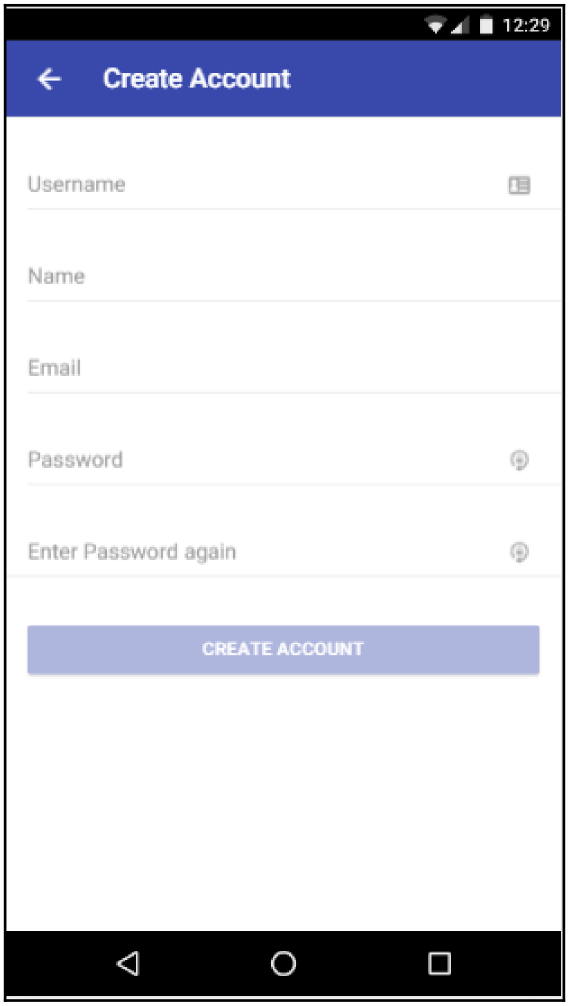
- **Push notification**: You can send a push notification when a new post is published. It can be done inside the `createPost` function in the `SocialProvider`. It is good to note that you have to store the `token` of each user's device inside the Firebase database under the global users list.
- **Limit post counts**: Right now, we are showing all the posts of a user's feed. You can limit it using Firebase queries. It will be a good match with Ionic's infinite scroll.

App screenshots

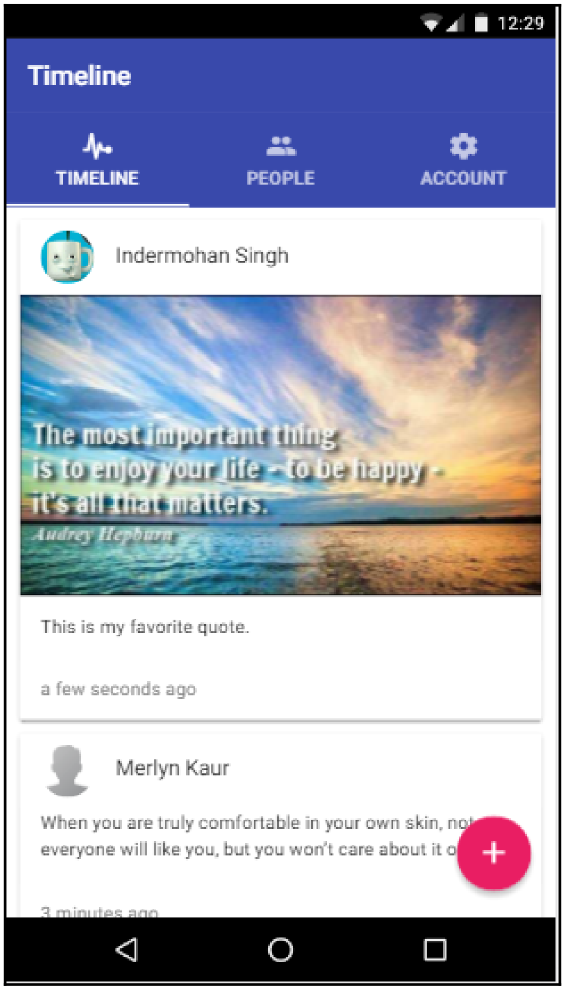
The following is the screenshot of the login page:



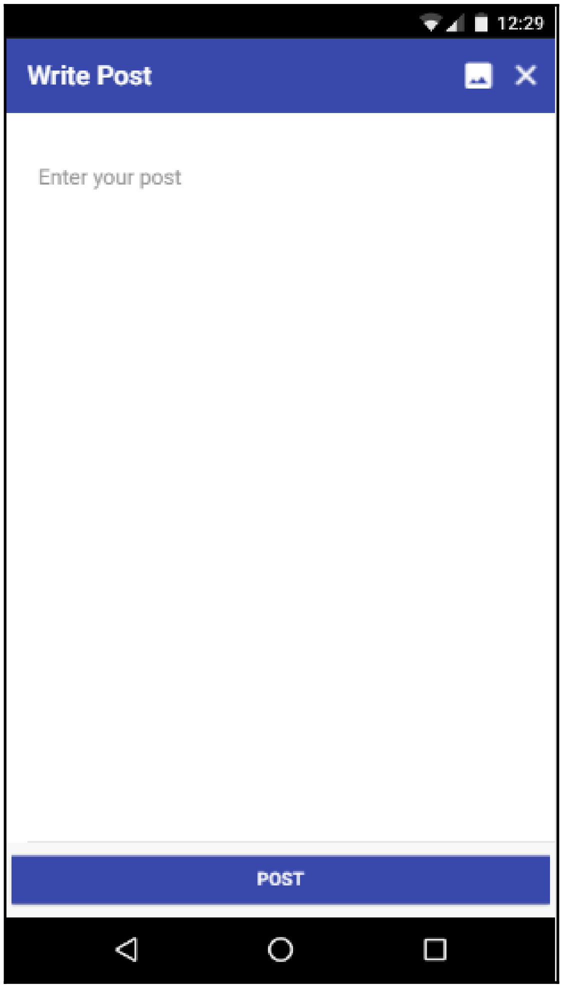
The following is the screenshot of the `CreateAccountPage`:



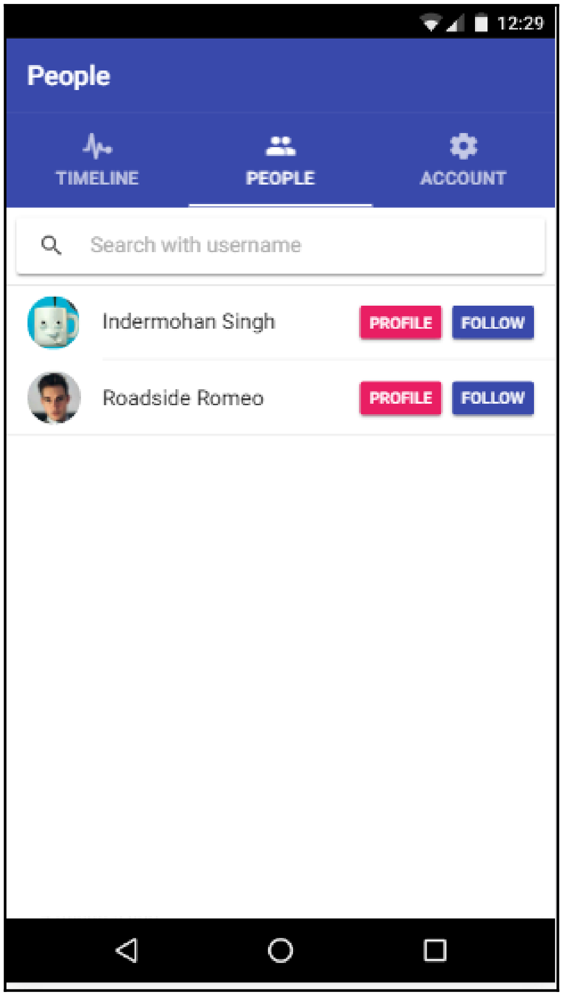
The following is the screenshot of the `TimelinePage`:


12:29
Timeline
TIMELINE
PEOPLE
ACCOUNT
Indermohan Singh
The most important thing is to enjoy your life - to be happy - it's all that matters.
Audrey Hepburn
This is my favorite quote.
a few seconds ago
Merlyn Kaur
When you are truly comfortable in your own skin, not everyone will like you, but you won't care about it o
3 minutes ago

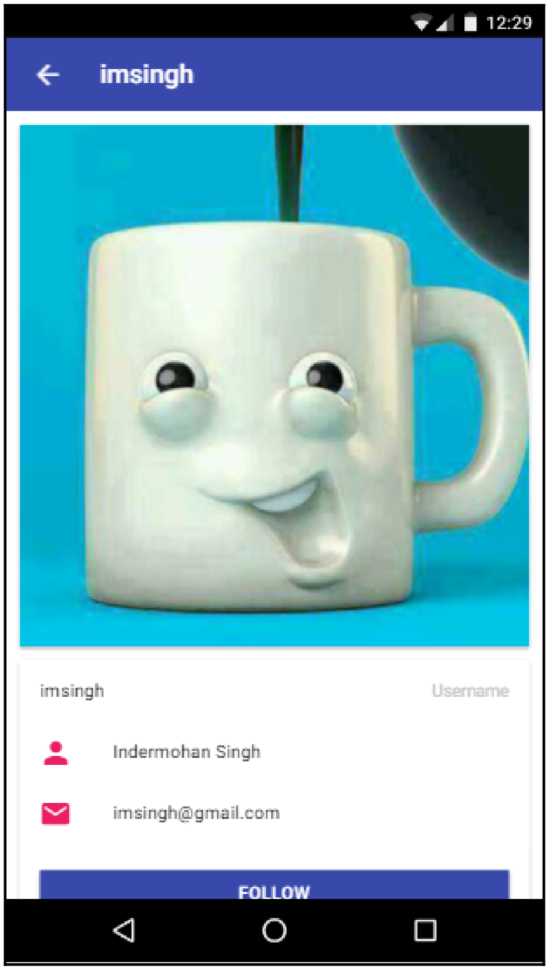
The following is the screenshot of the `PostPage`:



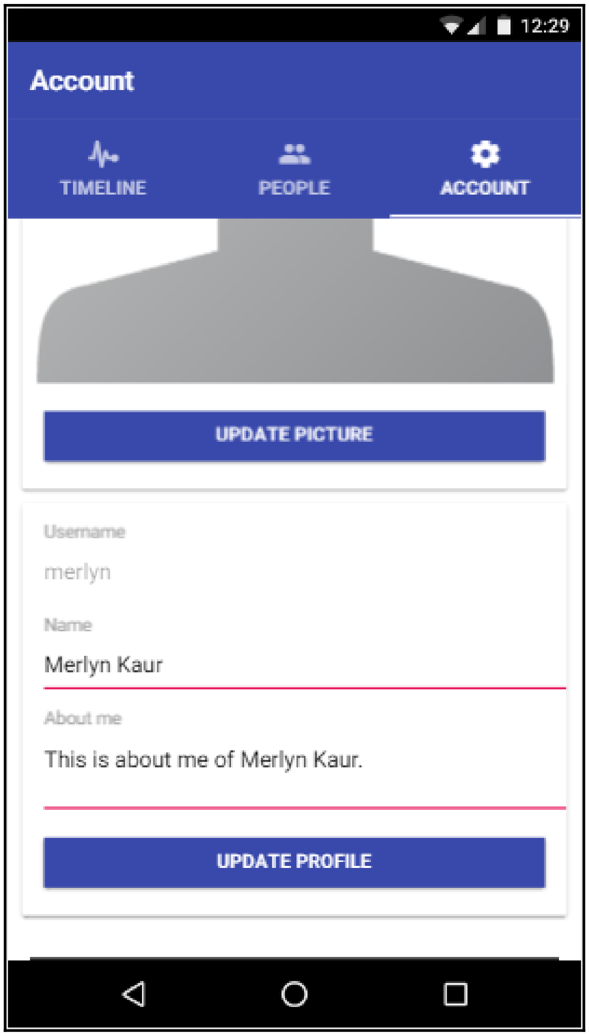
The following is the screenshot of the `PeoplePage`:



The following is the screenshot of the `UserProfilePage`:



The following is the screenshot of the `AccountPage`:



Username

merlyn

Name

Merlyn Kaur

About me

This is about me of Merlyn Kaur.

Summary

In this chapter, we have created a pretty simple social app just like *Twitter*. It is a totally front-end application and we are using the Firebase database, authentication, and storage for our application.

Index