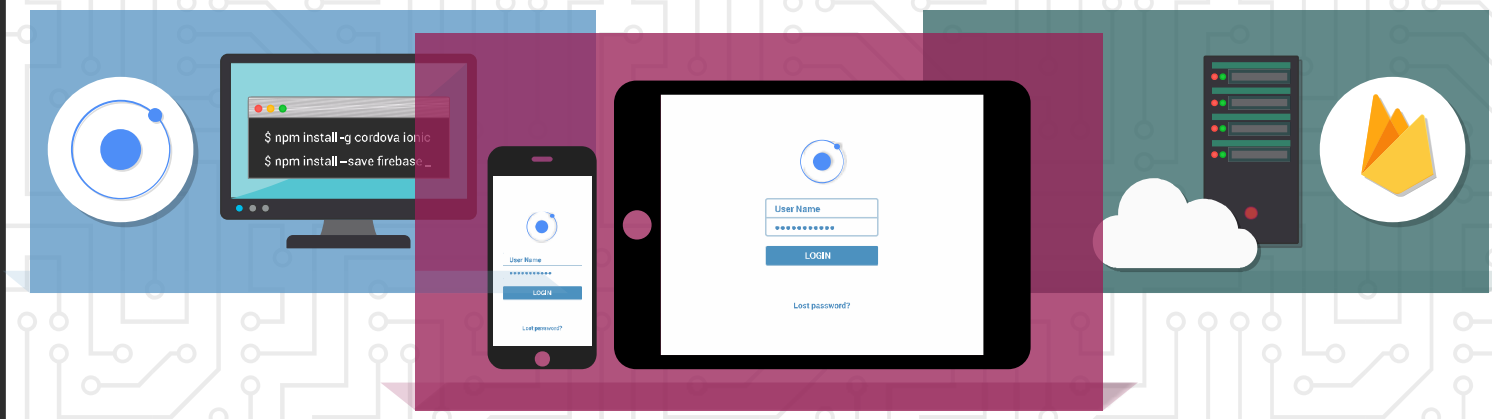
Building Firebase Powered Ionic Apps

By Jorge Vergara



@javebratt

To my wife Evelyn and my son Emmanuel

You made this happen

Contents

Introduction 2
Updates & Errata 2

1 The Setup Process **3**
What we will build 3
Install Firebase 5
Installing the camera plugin 8

2 User Authentication **9**
Initializing the Firebase App. 9
Listening to the authentication state 11
Building the Authentication Provider 13
The Authentication Pages 16

3 Promises **30**
What is a Promise? 30

4 Create, Read, Update, and Delete your data. **33**
Setup 33
Creating the User Profile Provider 34
Creating the Profile Page. 38

5 Working with data as a List **45**
Creating the Event Provider 45
Creating new events 47
Listing the events 50
The Event Detail Page 52

6 Firebase Storage **58**
Taking pictures 58

7 Security Rules **63**
Database Security 63
Storage Security 64
Data Validation 64

8 Next Steps **65**

9 CHANGELOG **66**
1.0.4 Updates to latest versions (*No breaking changes*.) 66
1.0.2 Fixes firebase type setting bugs: 66

Introduction

First of all, **Thank you!**

The goal of this book is for you to use the knowledge you have got about the Ionic framework to start building more robust, responsive, interactive applications, using Firebase as your backend.

Tho this book does not explain the basis of the Ionic framework if you are up for it, I am an email away j@javebratt.com, and I am more than happy to help you overcome any obstacles you find on your journey.

This book is not going to be very theoretical. Instead, we will build an application together, and I will guide you through the development of the application explaining new concepts as soon as they pop up. (*If you find something that you think should be described better in the book, please shoot me an email, I will be glad to see how we can add it*).

-*Jorge Vergara*, Chief Everything Officer, **JAVEBRATT**

Updates & Errata

Both Ionic and Firebase are under active development, that means that from time to time things will change, things are very stable at the moment so that the core principles will not change.

Whenever an update happens, I will reflect it in the book as soon as I can, and you will have it available for download for free.

Also, if you find a bug in the book (*code or typo*) let me know, I will mark it for a fix in the next update, the best way to get in touch with me is via email, you can find me at j@javebratt.com.

Chapter 1

The Setup Process

What we will build

In this first chapter, we are going to be creating an app for managing events.

I was thinking a lot about what to make, and I decided this would be a good start because it was one of the first apps I built with Ionic, an app for managing events for a particular niche (*it was my wife's startup, now dead*).

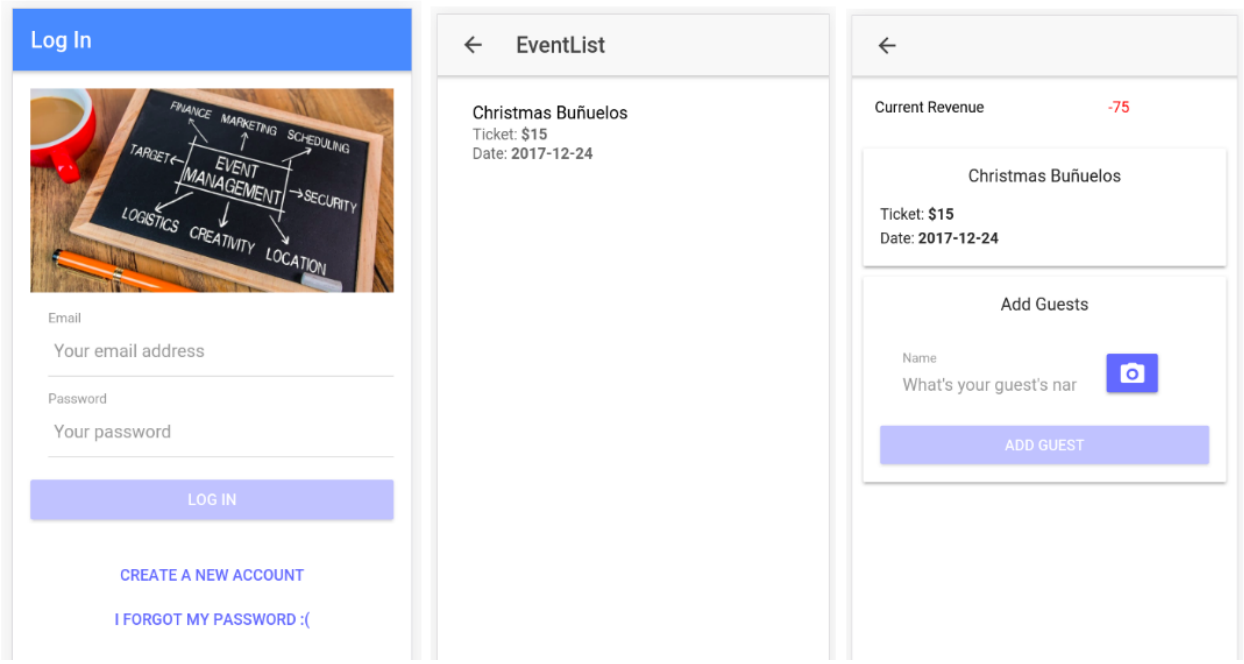


Figure 1.1: Event Manager Work flow

The goal of this app is for you to get comfortable with:

- User authentication.
- Create, Read, Update, and Delete (CRUD) data.
- Use Firebase `.transaction()`.
- Take pictures using `ionic-native`.
- Upload files to Firebase Storage.
- Understand Security Rules.

The entire source code for the app is available here https://github.com/javebratt/event-tutorial/, use it so you can follow along if you get stuck.

Is your development environment up to date?

Before writing any code, we are going to take a few minutes to install everything you need to be able to build this app, that way you will not have to be switching context between coding and fixing.

The first thing you will do is install node.js make sure you get version 6.x.

The second thing you will do is ensure that you have Ionic, and Cordova installed, you will do that by opening your terminal and typing:

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

Depending on your operating system (*mostly if you run on Linux or Mac*) you might have to add `sudo` before the `"npm install"` command.

Create the App

Now that you installed everything, you are ready to create your new Ionic app.

To do this, go ahead and open your terminal, move to wherever it is that you save your projects and start the app:

```
$ cd Development
$ ionic start EventManager blank
$ cd EventManager
```

If you are new to the terminal, what those commands do is to:

- Move you into the `Development` folder.
- Create a new Ionic app using the blank template and calling it `EventManager`.
- Move into the new app's folder.

From now on, whenever you are going to type something on the command line, it is going to be in your app's folder unless I say otherwise.

The "npm" packages that come with the project

When you use the Ionic CLI to create a new project, it is going to do many things for you, one of those things is making sure your project has the necessary `npm` packages it needs.

That means, the `start` command is going to install all of the requirements and more, here's what `package.json` should look like:

```
"dependencies": {
  "@angular/common": "5.0.3",
  "@angular/compiler": "5.0.3",
  "@angular/compiler-cli": "5.0.3",
  "@angular/core": "5.0.3",
  "@angular/forms": "5.0.3",
  "@angular/http": "5.0.3",
  "@angular/platform-browser": "5.0.3",
  "@angular/platform-browser-dynamic": "5.0.3",
  "@ionic-native/core": "4.4.0",
  "@ionic-native/splash-screen": "4.4.0",
  "@ionic-native/status-bar": "4.4.0",
  "@ionic/storage": "2.1.3",
  "ionic-angular": "3.9.2",
  "ionicons": "3.0.0",
  "rxjs": "5.5.2",
  "sw-toolbox": "3.6.0",
  "zone.js": "0.8.18"
},
"devDependencies": {
  "@ionic/app-scripts": "3.1.4",
  "typescript": "2.4.2"
}
```

Depending on when you read this, these packages might change (*specially version numbers*) so keep that in mind; also you can always email me at j@javebratt.com if you have any questions/issues/problems with this.

Install Firebase

Since all we are going to talk about in this book is Firebase, now we need to install… *You guessed it!* Firebase :)

To install a new package open your Terminal again and run:

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

That will install the latest version of the Firebase JavaScript SDK, which is what we will use in this first example. (*By the way, if you're using npm 5 you won't need to add the –save flag*.)

Create and Import Pages and Providers

There's much cognitive overhead in the brain when you are switching tasks consistently, so we are going to do something a bit different, we are going to do all the setup the app requires before we start writing functionality code.

That way, when it is time to start writing the app's functionality we can focus on doing that, and we will not have to switch back and forth between functionality and setup.

So we are going to take a moment to create every page and provider we are going to use on our app. First, the pages, go ahead and open your terminal and start generating them:

```
$ ionic generate page EventDetail
$ ionic generate page EventCreate
$ ionic generate page EventList
$ ionic generate page Login
$ ionic generate page Profile
$ ionic generate page ResetPassword
$ ionic generate page Signup
```

The `generate page` command creates a folder named after the class you created, let's go into detail with the first one `EventDetail`.

`ionic generate page EventDetail` will create a folder named `event-detail`, and inside that folder, it will create four files:

`event-detail.html` is the view file, where we are going to write our HTML code, what our users will eventually see.

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

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

`event-detail.module.ts` is the module file, in the past, we used to declare and initialize pages in `app.module.ts`, with Ionic 3, the Ionic team introduced code splitting and lazy loading. Meaning each page gets its module declaration file, this way, instead of loading every page of the app when our users are launching it, we only load the home page, and then we can load each page as we need them, instead of all at once.

```
import { NgModule } from "@angular/core";
import { IonicPageModule } from "ionic-angular";
import { EventDetailPage } from "./event-detail";

@NgModule({
  declarations: [EventDetailPage],
  imports: [IonicPageModule.forChild(EventDetailPage)]
})
export class EventDetailPageModule {}
```

You do not need to mess with that file, and it is a trimmed down version of what we have in `app.module.ts` only for this page.

`event-detail.scss` is out styles file, where we will be making our apps "prettier."

```
page-event-detail {}
```

Also, lastly, `event-detail.ts` is our class, where we are going to be declaring all the functionality the `EventDetail` class will have.

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

@IonicPage()
@Component({
  selector: "page-event-detail",
  templateUrl: "event-detail.html"
})
export class EventDetailPage {
  constructor(public navCtrl: NavController, public navParams: NavParams) {}
}
```

If you have any questions about the files that we generated there, you can email me.

After we create all the pages, we are going to create our providers, if you do not know what a provider is, in the most simple terms providers are recipes that know how to create dependencies. That way we can have different providers for the various aspects of our code.

For example, we can create an authentication provider that handles everything related to authentication, that way when from our page we call the authentication's provider `login()` function, and our page does not care how the provider gets it to work, it only cares that it works.

For this application, we will need three providers, one for authentication, one to handle all the event management, and one provider to manage the user's profile.

```
$ ionic generate provider Auth
$ ionic generate provider Event
$ ionic generate provider Profile
```

Each time you use the `generate provider` command, the Ionic CLI will do a few things for you, let's examine one of those providers to see what got created.

When you run `ionic generate provider Auth` the CLI is going to create a folder called `auth` inside the `providers` folder, and that folder will have a file called `auth.ts` here's what you will see in `src/providers/auth/auth.ts`.

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

@Injectable()
export class AuthProvider {
  constructor() {}
}
```

It has a few other things, but those are the things we care about, the `@Injectable()` decorator is what makes it an actual provider/module that we can inject into our classes.

If you go now and check `app.module.ts` you will also notice that the CLI created an import for your new providers, and added them to the provider's array inside the `@NgModule`. *And you did not have to do any of that!*

Right there your app should be able to run without any errors in the browser when you do `ionic serve`.

If you are running into trouble getting this to work you can send me an email and I will be happy to help you troubleshoot.

Installing the camera plugin

We are going to use `ionic-native` for the Camera. You can find more information about it in Ionic Official Docs, but I will cover everything we need for this project in the book.

To install it open your terminal and type:

```
$ ionic cordova plugin add cordova-plugin-camera --save
$ npm install --save @ionic-native/camera
```

We are installing two things there, the cordova plugin to use the native functionality, and the `ionic-native` package that will allow us to use the plugin writing "*angular kind*" code, meaning we can get our plugin code to look pretty similar to everything else we are writing.

There are two things we are going to do now to get the plugin to work, the first thing we need to do is inject it into the provider's array in `app.module.ts`, for that we can go ahead and open the `app.module.ts` file and import the plugin, then inject it:

```
import { Camera } from '@ionic-native/camera';
@NgModule({
  ...
  providers: [ Camera ]
})
export class AppModule {}
```

Injecting the Camera class from Ionic Native inside the providers' array will let us use the camera anywhere inside our app.

> *Don't forget that the camera plugin only works when you install the app on an actual phone, it won't work while we're testing in the browser*.

Right now you should have the skeleton of your app, and it should run. Go ahead and try it in your browser and see how it looks.

The app should run in the browser without any errors showing you the HomePage with a placeholder that says something like "*the world is your oyster and go to the docs*."

If it does not run, copy the stack trace or take screenshots and shoot me an email so I can help you debug what's going wrong.

Next Steps

When you are ready, move to the next part of this chapter, there you will initialize your app and create the auth module.

Chapter 2

User Authentication

If you've ever built an authentication system you know it can be a pain, setting up secure servers, building the entire back-end, it can take a while when all you want is to focus on making your app great.

That right there is the main reason I chose Firebase as my backend.

In this module you'll learn how to create an email and password authentication system, it will let your users do the three primary things every app needs to do:

- Create a new account.
- Login to an existing account.
- Send a password reset email.

Initializing the Firebase App.

The first thing we need to do for our app is to initialize it, that means telling Ionic how to connect to our Firebase database.

For that, we are going to be working on `src/app/app.component.ts` file.

The first thing you're going to do in that file is to import Firebase, that way the file knows it needs to use it and has access to the methods we need.

We need to add the import line at the top of the file with the other imports:

```
import firebase from 'firebase';
```

After that, we need to go into the constructor and call the initialize app function:

```
firebase.initializeApp();
```

That function takes a config object as a parameter, and the config object has all the API keys to connect to Firebase.

To get that config object for your app, go to your Firebase's Console, click the button that says "*Add Firebase to your web app*".

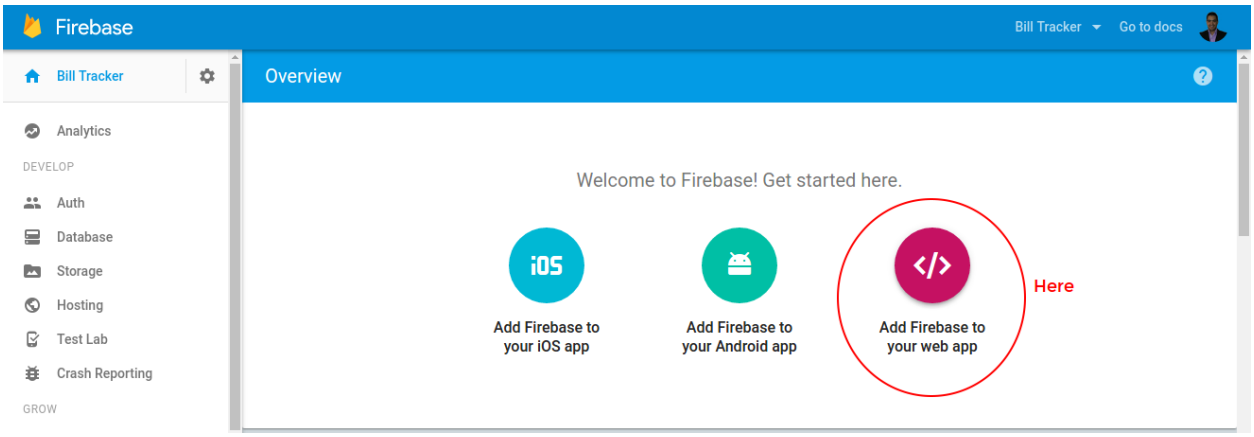


Figure 2.1: Firebase Console

It will show you some initialization code, but focus just on this bit:

```
var config = {
  apiKey: "",
  authDomain: "",
  databaseURL: "",
  projectId: "",
  storageBucket: "",
  messagingSenderId: ""
};
```

In the Firebase Console you need to enable the authentication method for your app, go to **Authentication > Sign-In Method > Email and Password**

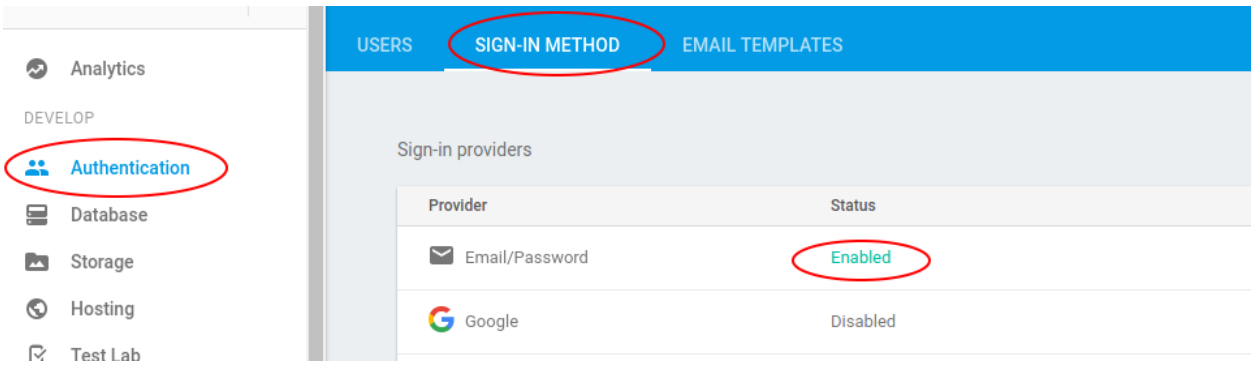


Figure 2.2: Auth Methods in the console

Going back to `app.component.ts`, you can either pass that object to the `.initializeApp()` function or you can create a different file to hold your credentials and then import them inside the `component` page.

Keeping the credentials in a separate file is a good practice if you plan to keep a public repo of your app, that way you can keep that file out of source control.

To do this, create a file called `credentials.ts` inside the `src/app/` folder, the entire content of the file will be this:

```
// Initialize Firebase
export const firebaseConfig = {
  apiKey: "",
  authDomain: "",
  databaseURL: "",
  projectId: "",
  storageBucket: "",
  messagingSenderId: ""
};
```

Then, inside the `app.component.ts` file you can import the `"firebaseConfig"` object and pass it to the firebase initialization function:

```
import { firebaseConfig } from './credentials';
constructor(
  platform: Platform,
  statusBar: StatusBar,
  splashScreen: SplashScreen
) {
  firebase.initializeApp(firebaseConfig);
}
```

Listening to the authentication state

Now that Ionic knows how to talk to our Firebase application, we are going to create an authentication listener.

It connects to Firebase Authentication and listens to changes in the user state to respond to them.

For example, if you create an account and start using the app for a while, then close it, when you come back to use the app again, it should understand that you're already a user and send you to the home page, instead of having you log in again.

The first thing we need to do is to remove the assignment from `rootPage` (*I will explain why in a minute*) inside `app.component.ts`.

```
rootPage: any;
```

After that, we are going to create the authentication listener, for that, we are going to be using Firebase's `onAuthStateChanged()`, you can read more about it in Firebase Official Documentation.

However, the TL;DR is that it adds an observer for auth state changes, meaning that whenever an authentication change happens, it will trigger the observer and the function inside it will run again:

```
const unsubscribe = firebase.auth().onAuthStateChanged(user => {
  if (!user) {
    this.rootPage = 'LoginPage';
    unsubscribe();
  } else {
    this.rootPage = HomePage;
    unsubscribe();
  }
});
```

OK, let me explain a bit more about how the `onAuthStateChanged()` function works:

It is listening to the authentication state, *how?* when you log in or signup using Firebase, it will store an auth object inside your localStorage.

This object has information about the user's profile, such as user's email, name, ID, what kind of authentication used, among others.

So the `onAuthStateChanged()` function looks for that object to see if a user already exists or not.

If the user doesn't exist, the `user` variable will be null, which will trigger the `if` statement and make the `rootPage = 'LoginPage'`.

However, if there's a user, it will return the user's information, at that point the listener is going to send the user to the HomePage since the user should not need to re-login inside the app.

The `unsubscribe();` is because we are telling the function to call itself once it redirects the user, this is because the `onAuthStateChanged()` returns the unsubscribe function for the observer.

Meaning it will stop listening to the authentication state unless you rerun it (it runs every time someone opens the app).

You could technically remove the `unsubscribe()` and let the function keep listening, but I have found that it is a good practice to unsubscribe in most of the cases, because if there's a sudden change in the authentication state that function will trigger.

For example, when we get to the user profile chapter we are going to be changing the user's email and password, to be able to change those the users need to add their old password to verify it is them.

When you use the email and password to re-authenticate the user, that change will send the user directly to the `HomePage` if the authentication listener was still active.

So yeah, even tho you can technically stay subscribed, think about the full flow of your app and see if that is a good idea.

Now, remember when I said:

> remove the assignment from `rootPage` (*I will explain why in a minute*)

Did you notice that we are assigning the home page by its class name, but the login page is a string?

That is because of how code splitting works, all of our pages, except `HomePage`, are declared in their page modules. To call those pages, we need to use the string name they have, by default, that name is the same their class name.

Now, move to the next page, where we will be creating the authentication provider.

Building the Authentication Provider

We are going to create an authentication service, in the setup part of the app we created all the providers we need, for this one, we are going to use the `AuthProvider` to handle all the authentication related interactions between our app and Firebase.

Open the file `providers/auth/auth.ts`, delete the methods it has and leave it like this:

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

@Injectable()
export class AuthProvider {
  constructor() {}
}
```

We are going to start building on top of it, the first thing we will do is to import Firebase:

```
import firebase from 'firebase';
```

We need to create four functions inside this file. We need the user to be able to:

- Login to an existing account.
- Create a new account.
- Send a password reset email.
- Logout from the app.

The first function we will create is the LOGIN function, for that go ahead and create a `loginUser()` function that takes two string parameters (email, password):

```
loginUser(email:string, password:string): Promise<any> {...}
```

And inside the function we'll create the Firebase login:

```
loginUser(email: string, password: string): Promise<any> {
  return firebase.auth().signInWithEmailAndPassword(email, password);
}
```

We're using the Firebase `signInWithEmailAndPassword()` method. The method takes an email and a password and logs in the user.

The way Firebase works, it does not have a regular "username" login, your users will need to use a valid email as a username.

If the function has an error, it will return the error code and message. For example, invalid email or password.

If the function goes through, the user will log in, Firebase will store the authentication object in localStorage, and the function will return the user to a promise.

> **NOTE:** If you are new to promises (like I was when I started working with Ionic) don't worry after we are done building the authentication module I will do my best to explain what promises are and how they work.

The second function we need is a signup feature, but that is not all it has to do when a new user creates an account, we want to store the user's email in our database.

SIDE-NOTE: Database and authentication are not "connected," like that, creating a user does not store its information inside the database, it saves it in the authentication module of our app, so we need to copy that data inside the database manually.

```
signupUser(email: string, password: string): Promise<any> {
  return firebase
    .auth()
    .createUserWithEmailAndPassword(email, password)
    .then(newUser => {
      firebase
        .database()
        .ref(`/userProfile/${newUser.uid}/email`)
        .set(email);
    })
    .catch(error => {
      console.error(error);
      throw new Error(error);
    });
}
```

We are using the `createUserWithEmailAndPassword()` to create our new user (*the name kinda says it all*, right?)

This function is cool because after it creates the user, the app also logs the user in automatically (*this wasn't always true*) meaning we do not have to call the login function again.

The function returns a Promise (*we'll talk about them later*) that will run some code for us when it's done creating the new user and login him into the app:

```
.then(newUser => {
  firebase
    .database()
    .ref(`/userProfile/${newUser.uid}/email`)
    .set(email);
})
```

That is a reference to the userProfile node inside our database.

That reference is creating a new node inside the `userProfile` node, and the UID identifies the node, the UID is Firebase automatic id generated for the user, it is its unique identifier.

Also, it is adding a property to that node called email, filling it with the new user's email address.

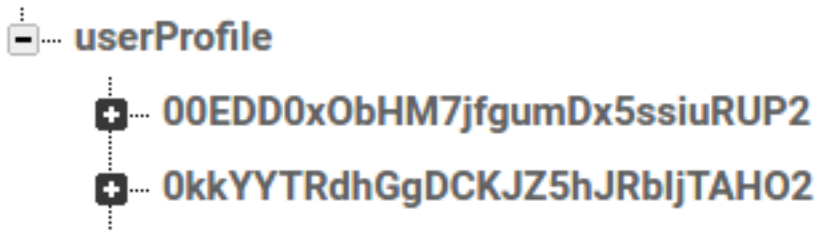


Figure 2.3: User profile node inside Firebase database

SIDE-NOTE: Notice that I'm using template strings, inside `.ref()`, if you don't know what those are, it's a new property of ES6 and TypeScript, notice that I'm not using double or single quotes, I'm using back ticks, in previous versions of JavaScript you'd need to do something like this to concatenate a string with a variable:

```
const firstName = "Jorge";
const lastName = "Vergara";
const myName = "My name is " + firstName + " " + lastName;
```

With ES6 or TypeScript you can use template strings, and it becomes:

```
const myName = `My name is ${firstName} ${lastName}`;
```

So you see, the `${}` indicates that the thing inside needs to be evaluated, you can add any kind of JS inside, like: `${2 + 2}` and it will output 4. Does that make it clearer?

We now need a function to let our users reset their passwords when they cannot remember them.

```
resetPassword(email:string): Promise<void> {
  return firebase.auth().sendPasswordResetEmail(email);
}
```

We are using the `sendPasswordResetEmail()` it returns a `void` Promise, meaning that even tho it does return a Promise, the promise is empty, so you mainly use it to perform other actions once it sends the password reset link.

And Firebase will take care of the reset login. They send an email to your user with a password reset link, the user follows it and changes his password without you breaking a sweat.

And lastly we'll need to create a logout function:

```
logoutUser():Promise<void> {
  return firebase.auth().signOut();
}
```

That one does not take any arguments it checks for the current user and logs him out.

It also returns a void promise. You will mainly use it to move the user to a different page (*probably to LoginPage*).

There's one thing people struggle with when logging out, sometimes the app is still listening to the database references, and it creates errors when your security rules are set up, for that, we need to turn the reference off before logging out.

```
logoutUser(): Promise<void> {
  const userId: string = firebase.auth().currentUser.uid;
  firebase
    .database()
    .ref(`/userProfile/${userId}`)
    .off();
  return firebase.auth().signOut();
}
```

And there we have all of our functions ready to use. Next, we will be creating the actual pages: login, signup, and password reset.

The Authentication Pages

By now we have a complete service or provider called AuthProvider that it is going to handle all the Firebase <<>> Ionic authentication related communications, now we need to create the actual pages the user is going to see.

THE LOGIN PAGE

Open `pages/login/login.html` and create a login form inside the ion-content tags to capture email and password:

```
<form [formGroup]="loginForm" (submit)="loginUser()" novalidate>
  <ion-item>
    <ion-label stacked>Email</ion-label>
    <ion-input formControlName="email" type="email"
    placeholder="Your email address"
    [class.invalid]="!loginForm.controls.email.valid && blur">
    </ion-input>
  </ion-item>
  <ion-item class="error-message" *ngIf="!loginForm.controls.email.valid &&
   loginForm.controls.email.dirty">
    <p>Please enter a valid email address.</p>
  </ion-item>

  <ion-item>
    <ion-label stacked>Password</ion-label>
    <ion-input formControlName="password" type="password"
    placeholder="Your password"
    [class.invalid]="!loginForm.controls.password.valid && blur">
    </ion-input>
  </ion-item>
  <ion-item class="error-message" *ngIf="!loginForm.controls.password.valid
  && loginForm.controls.password.dirty">
    <p>Your password needs more than 6 characters.</p>
  </ion-item>

  <button ion-button block type="submit" [disabled]="!loginForm.valid">
    Log In
  </button>
</form>

<button ion-button block clear (click)="goToSignup()">
  Create a new account
</button>

<button ion-button block clear (click)="goToResetPassword()">
  I forgot my password :(
</button>
```

That is a basic HTML form using Ionic components, and there's also form validation going on (*and we will add more form validation stuff to the TypeScript file*). We will not cover that in the book, but you can read about it in full detail in this post.

Now it is time to give it some style, and we are not going crazy with this, some margins and adding the `.invalid` class (which is a red border)

Open your `login.scss` and add the styles:

```
page-login {
  form {
    margin-bottom: 32px;
    button {
      margin-top: 20px !important;
    }
  }

  p {
    font-size: 0.8em;
    color: #d2d2d2;
  }

  ion-label {
    margin-left: 5px;
  }

  ion-input {
    padding: 5px;
  }

  .invalid {
    border-bottom: 1px solid #ff6153;
  }

  .error-message {
    min-height: 2.2rem;
    p {
      font-size: 60%;
    }
    ion-label {
      margin: 2px 0;
    }
    .item-inner {
      border-bottom: 0 !important;
    }
  }
}
```

Like I said, nothing too fancy, a few margins to make everything look a bit better.

And finally it is time to jump into the actual TypeScript code, open your `login.ts` file, you should have something similar to this:

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

@IonicPage()
@Component({
  selector: 'page-login',
  templateUrl: 'login.html',
})
export class LoginPage {
  constructor(public navCtrl: NavController) {...}
}
```

We will add the imports first, so everything is available when you need it:

```
import { Component } from '@angular/core';
import {
  Alert,
  AlertController,
  IonicPage,
  Loading,
  LoadingController,
  NavController
} from 'ionic-angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { EmailValidator } from '../../validators/email';
import { AuthProvider } from '../../providers/auth/auth';
import { HomePage } from '../home/home';
```

This is the breakdown of what we are importing:

- `Loading`, `FormGroup`, and `Alert` because I am a TypeScript junkie, we will use those to declare the types of the loading component's variable and the form variable.
- `LoadingController` and `AlertController` because we are going to be using an alert pop up and a loading component inside our page.
- `FormBuilder`, `Validators` those are used to get form validation going on with angular.
- `AuthProvider` is the authentication provider we created, we will be using it to call the login function.
- `EmailValidator` is a custom validator I created, it makes sure that when the user is going to type an email address, it follows this format: something @ something . something.

Let's get that out of the way right now, create a file: `src/validators/email.ts` and populate it with the following:

```
import { FormControl } from '@angular/forms';

export class EmailValidator {
  static isValid(control: FormControl) {
    const re = /^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$/
    .test(
      control.value
    );

    if (re) {
      return null;
    }

    return {
      invalidEmail: true
    };
  }
}
```

Explaining angular validators is out of the scope of this book, but you can check this post.

After everything is imported, we're going to inject all the providers in the constructor, so they become available inside the class:

```
constructor(
  public navCtrl: NavController,
  public loadingCtrl: LoadingController,
  public alertCtrl: AlertController,
  public authProvider: AuthProvider,
  formBuilder: FormBuilder
) {...}
```

While we are at it, we are going to create two global variables for this class right before the constructor, one to handle our login form, and the other one to handle our loading component:

```
public loginForm: FormGroup;
public loading: Loading;
constructor(...) {...}
```

Inside the constructor, we need to initialize our form:

```
constructor(
  public navCtrl: NavController,
  public loadingCtrl: LoadingController,
  public alertCtrl: AlertController,
  public authProvider: AuthProvider,
  formBuilder: FormBuilder
) {
  this.loginForm = formBuilder.group({
    email: [
      '',
      Validators.compose([Validators.required, EmailValidator.isValid])
    ],
    password: [
      '',
      Validators.compose([Validators.required, Validators.minLength(6)])
    ]
  });
}
```

We are using `formBuilder` inside the constructor to initialize the fields and give them a required validator.

If you want to know more about `FormBuilder` check Angular's docs.

Now let's create our functions:

```
goToSignup():void {
  this.navCtrl.push('SignupPage');
}

goToResetPassword():void {
  this.navCtrl.push('ResetPasswordPage');
}
```

Those two should be easier to comprehend; we are sending the user to the `SignupPage` or the `ResetPasswordPage`.

Our login function takes the values of the form fields and passes them to our `loginUser` function inside our `AuthProvider` service.

```
loginUser(): void {
  if (!this.loginForm.valid) {
    console.log(
      `Form is not valid yet, current value: ${this.loginForm.value}`
    );
  } else {
    const email = this.loginForm.value.email;
    const password = this.loginForm.value.password;

    this.authProvider.loginUser(email, password).then(
      authData => {
        this.loading.dismiss().then(() => {
          this.navCtrl.setRoot(HomePage);
        });
      },
      error => {
        this.loading.dismiss().then(() => {
          const alert: Alert = this.alertCtrl.create({
            message: error.message,
            buttons: [{ text: 'Ok', role: 'cancel' }]
          });
          alert.present();
        });
      }
    );
    this.loading = this.loadingCtrl.create();
    this.loading.present();
  }
}
```

It is also calling Ionic's loading component since the app needs to communicate with the server to log the user in there might be a small delay in sending the user to the HomePage so we are using a loading component to give a visual so the user can understand that it is loading :P

The Password Reset Page

The first app (web app) I built didn't have a password reset function. I was building it with PHP without any frameworks, hacking around stuff while learning on the go.

So every time someone needed to reset their password they needed to email me so I could manually reset it and then send them to their email (*that was terrible!*).

Firebase handles this for us. We create a page where the user inputs the email address, and we call the reset password function we set up in our authentication provider.

We are going to handle this the same way we did the login page (view, style, code).

Open your `reset-password.html` file, and create almost the same form we set up for the login page, with the email field (*don't forget to change the form name!*)

```
<ion-content padding>
  <img src="assets/img/banner.jpg" />
  <form [formGroup]="resetPasswordForm" (submit)="resetPassword()" novalidate>
    <ion-item>
      <ion-label stacked>Email</ion-label>
      <ion-input formControlName="email" type="email"
      placeholder="Your email address"
      [class.invalid]="!resetPasswordForm.controls.email.valid && blur">
      </ion-input>
    </ion-item>
    <ion-item class="error-message"
    *ngIf="!resetPasswordForm.controls.email.valid  &&
    resetPasswordForm.controls.email.dirty">
      <p>Please enter a valid email.</p>
    </ion-item>

    <button ion-button block type="submit"
    [disabled]="!resetPasswordForm.valid">
      Reset your Password
    </button>

  </form>
</ion-content>
```

We'll add basic margins and borders on our `reset-password.scss` file:

```
page-reset-password {
  form {
    margin-bottom: 32px;

    button {
      margin-top: 20px;
    }
  }

  p {
    font-size: 0.8em;
    color: #d2d2d2;
  }

  ion-label {
    margin-left: 5px;
  }

  ion-input {
    padding: 5px;
  }

  .invalid {
    border-bottom: 1px solid #ff6153;
  }

  .error-message {
    min-height: 2.2rem;
    p {
      font-size: 60%;
    }
    ion-label {
      margin: 2px 0;
    }
    .item-inner {
      border-bottom: 0 !important;
    }
  }
}
```

And now it's time to code the functionality, open `reset-password.ts`, it should look like this:

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

@IonicPage()
@Component({
  selector: 'page-reset-password',
  templateUrl: 'reset-password.html',
})
export class ResetPasswordPage {
  constructor(public navCtrl: NavController) {...}
}
```

Like we did before, we are going to be adding the imports and injecting our services into the constructor:

```
import { Component } from "@angular/core";
import {
  Alert,
  AlertController,
  IonicPage,
  NavController
} from "ionic-angular";
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { AuthProvider } from "../../providers/auth/auth";
import { EmailValidator } from "../../validators/email";

@IonicPage()
@Component({
  selector: "page-reset-password",
  templateUrl: "reset-password.html"
})
export class ResetPasswordPage {
  public resetPasswordForm: FormGroup;

  constructor(
    public navCtrl: NavController,
    public authProvider: AuthProvider,
    public alertCtrl: AlertController,
    formBuilder: FormBuilder
  ) {
    this.resetPasswordForm = formBuilder.group({
      email: [
        "",
        Validators.compose([Validators.required, EmailValidator.isValid])
      ]
    });
  }
}
```

Then we create the password resetting function:

```
resetPassword(): void {
  if (!this.resetPasswordForm.valid) {
    console.log(
      `Form is not valid yet, current value: ${this.resetPasswordForm.value}`
    );
  } else {
    const email: string = this.resetPasswordForm.value.email;
    this.authProvider.resetPassword(email).then(
      user => {
        const alert: Alert = this.alertCtrl.create({
          message: "Check your email for a password reset link",
          buttons: [
            {
              text: "Ok",
              role: "cancel",
              handler: () => {
                this.navCtrl.pop();
              }
            }
          ]
        });
        alert.present();
      },
      error => {
        const errorAlert = this.alertCtrl.create({
          message: error.message,
          buttons: [{ text: "Ok", role: "cancel" }]
        });
        errorAlert.present();
      }
    );
  }
}
```

Same as login, it takes the value of the form field, sends it to the `AuthProvider` service and displays a loading component while we get Firebase's response.

If there's something about that file that you do not understand don't hesitate to shoot me an email and I will be happy to help you.

The Signup Page

We are missing our signup page. This is the page that will be used by new users to create a new account, and I am going to be a bit more fast paced with this one (*basically pasting the code*) since it is the same as the login page but changing the forms name.

The first thing we will create is the view. This is how you want your `signup.html` to look like:

```
<ion-content padding>
  <img src="assets/img/banner.jpg" />
  <form [formGroup]="signupForm" (submit)="signupUser()" novalidate>
    <ion-item>
      <ion-label stacked>Email</ion-label>
      <ion-input formControlName="email" type="email"
      placeholder="Your email address"
      [class.invalid]="!signupForm.controls.email.valid && blur">
      </ion-input>
    </ion-item>
    <ion-item class="error-message" *ngIf="!signupForm.controls.email.valid
    && signupForm.controls.email.dirty">
      <p>Please enter a valid email.</p>
    </ion-item>

    <ion-item>
      <ion-label stacked>Password</ion-label>
      <ion-input formControlName="password" type="password"
      placeholder="Your password"
      [class.invalid]="!signupForm.controls.password.valid && blur">
      </ion-input>
    </ion-item>
    <ion-item class="error-message"
    *ngIf="!signupForm.controls.password.valid  &&
    signupForm.controls.password.dirty">
      <p>Your password needs more than 6 characters.</p>
    </ion-item>

    <button ion-button block type="submit" [disabled]="!signupForm.valid">
      Create an Account
    </button>
  </form>
</ion-content>
```

Now add some margins on the `signup.scss` file:

```
page-signup {
  form {
    margin-bottom: 32px;
    button {
      margin-top: 20px;
    }
  }

  p {
    font-size: 0.8em;
    color: #d2d2d2;
  }

  ion-label {
    margin-left: 5px;
  }

  ion-input {
    padding: 5px;
  }

  .invalid {
    border-bottom: 1px solid #ff6153;
  }

  .error-message {
    min-height: 2.2rem;
    p {
      font-size: 60%;
    }
    ion-label {
      margin: 2px 0;
    }
    .item-inner {
      border-bottom: 0 !important;
    }
  }
}
```

And finally open your `signup.ts` file, import the services you'll need and inject them to your constructor:

```
import { Component } from "@angular/core";
import {
  Alert,
  AlertController,
  IonicPage,
  Loading,
  LoadingController,
  NavController
} from "ionic-angular";
import { FormBuilder, FormGroup, Validators } from "@angular/forms";
import { AuthProvider } from "../../providers/auth/auth";
import { EmailValidator } from "../../validators/email";
import { HomePage } from "../home/home";

@IonicPage()
@Component({
  selector: "page-signup",
  templateUrl: "signup.html"
})
export class SignupPage {
  public signupForm: FormGroup;
  public loading: Loading;

  constructor(
    public navCtrl: NavController,
    public authProvider: AuthProvider,
    public loadingCtrl: LoadingController,
    public alertCtrl: AlertController,
    formBuilder: FormBuilder
  ) {
    this.signupForm = formBuilder.group({
      email: [
        "",
        Validators.compose([Validators.required, EmailValidator.isValid])
      ],
      password: [
        "",
        Validators.compose([Validators.minLength(6), Validators.required])
      ]
    });
  }
}
```

You have done this twice now so there shouldn't be anything new here, now create the signup function:

```
signupUser(): void {
  if (!this.signupForm.valid) {
    console.log(
      `Need to complete the form, current value: ${this.signupForm.value}`
    );
  } else {
    const email: string = this.signupForm.value.email;
    const password: string = this.signupForm.value.password;

    this.authProvider.signupUser(email, password).then(
      user => {
        this.loading.dismiss().then(() => {
          this.navCtrl.setRoot(HomePage);
        });
      },
      error => {
        this.loading.dismiss().then(() => {
          const alert: Alert = this.alertCtrl.create({
            message: error.message,
            buttons: [{ text: "Ok", role: "cancel" }]
          });
          alert.present();
        });
      }
    );
    this.loading = this.loadingCtrl.create();
    this.loading.present();
  }
}
```

And there you have it. You have a fully functional auth system working on your app now.

If you run the app, you shouldn't have any errors, and you can test the entire authentication flow.

A piece of advice, store what you have right now in a Github repository, you will be able to clone it every time you need to start an app that uses Firebase as an authentication backend, saving tons of time.

This lesson was long, go for a walk, grab a cookie, or whatever you want to do for fun and then come back to the next one, we are going to talk a little about Promises.

Chapter 3

Promises

Promises are a big part of JS, especially when you are building cloud-connected applications since you use them when fetching a JSON API or doing AJAX work, I am going to use the next page or two to explain why.

What is a Promise?

A promise is something that will happen between now and the end of time. It is something that will occur in the future, but not immediately.

But what does that mean?

To understand this, you will need to learn something else; first, **JS is almost entirely asynchronous**, this means that when you call a function, it does not stop everything to run it.

It will keep running the code below that function even if the function is not done yet.

To explain this better let's look at an example, let's say you are logging in a user, and after login the user you want to log the user and a random string to the console.

```
firebase.auth().signInWithEmailAndPassword("email", "password");
console.log(firebase.auth().currentUser);
console.log("This is a random string");
```

If you come from a different language you would expect the code to work like this:

1. It logs in our user.
2. It logs the current user to the console.
3. It logs the random string to the console.

However, that is not what's going on since JS is asynchronous, it is doing something like this:

1. It calls the function to log in the user.
2. It logs `null` or `undefined` because there's no user yet.
3. It logs the random string to the console.
4. It finished logging in the user.
5. It updates the log to the console to reflect the new user (sometimes).

It is running everything it can find without waiting for it to complete.

That is where Promises come in if a Promise could talk it would tell you something like:

> Hey, I do not have the data right now, here's an IOU and as soon as the data is back, I will make sure to give it to you.

Moreover, you can catch those promises with `.then()`. So in the above example, if we wanted things to happen in this order:

1. It logs in the user.
2. It logs the current user to the console.
3. It logs the random string to the console.

We'd have to write it this way:

```
firebase.auth().signInWithEmailAndPassword("email", "password")
.then( user => {
    console.log(user);
    console.log("This is a random string");
});
```

That way JS is waiting until the function is completed before running the rest of the code.

Another example from Firebase would be sending a user to a new page on login (which we covered in the last chapter), if you know nothing about promises you might write something like this:

```
firebase.auth().signInWithEmailAndPassword("j@javebratt.com", "123456");
this.navCtrl.setRoot(HomePage);
```

This makes sense right? Log the user in, and then set the HomePage as root, and in most cases, you will not even notice there's a problem there.

However, what that is doing is calling the login function and immediately sending the user to the `HomePage` without waiting for the login's response, so basically, you are letting someone into your house without knowing if that person has permission to go inside.

Instead, we'd write something like this:

```
firebase.auth().signInWithEmailAndPassword("j@javebratt.com", "123456")
.then( user => {
    if (user) { this.navCtrl.setRoot(HomePage); }
});
```

So, when the login function returns, check if there's a real user there, and then send him to the `HomePage`.

Create your Promises

You do not have to rely on Promises being baked into almost everything in JS; you can also make your promises.

Let's say you are writing a data provider or service and you want to pull some data from Firebase (*yeah, I know it returns a Promise by default*), but instead of returning Firebase's promise to the class, what if you want to manipulate the data and then return it?

Then it is:

```
firebase.auth().signInWithEmailAndPassword("j@javebratt.com", "123456")
.then( user => {
  return new Promise( (resolve, reject) => {
    if (user) {
      user.newPropertyIamCreating = "New value I'm adding";
      resolve(user);
    } else {
        reject(error);
    }
  });
});
```

Right there you are catching Firebase's promise, modifying the user object and then returning a new promise with the modified object (*or returning an error*).

The promise takes two arguments, `resolve` and `reject`, we use `resolve` to tell it what we want to return inside that promise, and `reject` is used as a `'catch'` if there are any errors.

So that is it, that was my short intro to Promises, hope you learned as much or more that I found out while researching for this :)

On the next chapter we are going to be doing some CRUD (Create, Read, Update, and Delete) to help you understand how to work with the real-time database.

Chapter 4

Create, Read, Update, and Delete your data.

We learned about authentication and how Promises work, now it is time to add some more functionality to our app, we are going to get to work with objects from the database, while we create a profile page for our users.

I decided to go with a user profile because it can attack two main problems at once, working with Objects and updating the data in our Firebase authentication.

I think that is enough for an intro, so let's jump into business!

Setup

The first thing we are going to do is to set up everything we will need for this part of the tutorial. We will be creating a profile page and a profile data provider.

Remember that we created the actual files in the first chapter, now we need to start building on top of them.

The first thing we will do is to create a link to the profile page, so go to `home.html` and create a button in the header that calls the `goToProfile()` function:

```
<ion-header>
  <ion-navbar>
    <ion-title>
      Ionic Blank
    </ion-title>
    <ion-buttons end>
      <button ion-button icon-only (click)="goToProfile()">
        <ion-icon name="person"></ion-icon>
      </button>
    </ion-buttons>
  </ion-navbar>
</ion-header>
```

Now we'll go to the `home.ts` file and create a function to send our users to the profile page:

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

@Component({
  selector: "page-home",
  templateUrl: "home.html"
})
export class HomePage {
  constructor(public navCtrl: NavController) {}

  goToProfile(): void {
    this.navCtrl.push("ProfilePage");
  }
}
```

Creating the User Profile Provider

Now into a bit more complicated pieces, we are going to create our profile provider, the idea for this provider is that it lets us store our profile information in Firebase's real-time database and also change our email and password from our profile data.

This provider should have a function for:

- Getting the user profile
- Updating the user's name
- Updating the user's date of birth (*I always save this stuff so I can surprise my users later*)
- Updating the user's email both in the real-time database and the auth data
- Changing the user's password.

The first thing we need to do is to import Firebase, so open `providers/profile/profile.ts` and add firebase:

```
import firebase from 'firebase';
```

We are going to create and initialize two variables:

```
public userProfile:firebase.database.Reference;
public currentUser:firebase.User;

constructor() {
  firebase.auth().onAuthStateChanged( user => {
    if(user){
      this.currentUser = user;
      this.userProfile = firebase.database().ref(`/userProfile/${user.uid}`);
    }
  });
}
```

We are going to use `userProfile` as a database reference to the current logged in user, and `currentUser` will be the user object.

We're wrapping them inside an `onAuthStateChanged()` because if we get the user synchronously, there's a chance it will return `null` when the provider is initializing.

With the `onAuthStateChanged()` function we make sure to resolve the user first before assigning the variable.

Now we can start creating our functions, let's start with a function that returns the user's profile from the database:

```
getUserProfile(): firebase.database.Reference {
  return this.userProfile;
}
```

Since we already initialized `userProfile`, we can return it in this function, and we will handle the result inside the profile page.

Next we're going to create a function to update the user's name:

```
updateName(firstName: string, lastName: string): Promise<any> {
  return this.userProfile.update({ firstName, lastName });
}
```

We're using `.update()` here because we only want to update the `firstName` and `lastName` properties, if we were to use `.set()` to write to the database, it would delete everything under the user's profile and replace it with the first and last name.

`.update()` also returns a promise, but it is void, meaning it has nothing inside, so you use it to see when the operation was completed and then perform something else.

Next function in line would be the one to update the user's birthday, this is pretty much the same thing as the `updateName()` function, with the slight difference that we are updating a different property:

```
updateDOB(birthDate:string): Promise<any> {
  return this.userProfile.update({ birthDate });
}
```

Now is where things get a little trickier, we are going to change the user's email address, *why is it tricky?* Because we are not only going to alter the email from the database, we are going to change it from the authentication service too.

That means that we are changing the email the user uses to log into our app, and you cannot call the change email function and have it magically work.

This is because some security-sensitive actions (*deleting an account, setting a primary email address, and changing a password*) require that the user has recently signed-in.

If you perform one of these actions, and the user signed in too long ago, the operation fails with an error. When this happens, re-authenticate the user by getting new sign-in credentials from the user and passing the credentials to reauthenticate.

I am going to go ahead and create the function and then break it down for you to understand better:

```
updateEmail(newEmail: string, password: string): Promise<any> {
  const credential: firebase.auth.AuthCredential = firebase.auth.
    EmailAuthProvider.credential(
      this.currentUser.email,
      password
    );
  return this.currentUser
    .reauthenticateWithCredential(credential)
    .then(user => {
      this.currentUser.updateEmail(newEmail).then(user => {
        this.userProfile.update({ email: newEmail });
      });
    })
    .catch(error => {
      console.error(error);
    });
}
```

Here's what's going on:

- We are using `firebase.auth.EmailAuthProvider.credential();` to create a credential object, Firebase uses this for authentication.
- We are passing that credential object to the re-authenticate function. My best guess is that Firebase does this to make sure the user trying to change the email is the actual user who owns the account. **For example,** if they see the user added email and password recently they let it pass, but if not they ask for it again to avoid a scenario where the user leaves the phone unattended for a while, and someone else tries to do this.
- After the re-authenticate function is completed we're calling `.updateEmail()` and passing the new email address, the `updateEmail()` does as its name implies, it updates the user's email address.
- After the user's email address is updated **in the authentication service** we proceed to call the profile reference from the database and also refresh the email there.

The good thing about that being tricky is that now the `updatePassword()` function will be smooth for you!

```
updatePassword(newPassword: string, oldPassword: string): Promise<any> {
  const credential: firebase.auth.AuthCredential = firebase.auth
    .EmailAuthProvider.credential(
      this.currentUser.email,
      oldPassword
    );

  return this.currentUser
    .reauthenticateWithCredential(credential)
    .then(user => {
      this.currentUser.updatePassword(newPassword).then(user => {
        console.log('Password Changed');
      });
    })
    .catch(error => {
      console.error(error);
    });
}
```

You should probably get yourself a cookie, that was a lot of code, and your sugar levels need a refill, I am taking a 20-minute break myself to get some food…

…Alright, I am back, let's analyze what you have now, right now you have a fully functional provider that will handle all the profile related interactions between your application and Firebase.

It is great because you can call those functions from anywhere inside your app now, without copy-/pasting a bunch of functions to make it work.

Now that it is working, we are going to be creating the profile page, and it is going to be the page where we display, add, and update our user's profile information.

Creating the Profile Page.

We are going to break this down into three parts, the view, the design, and the code.

The first thing we are going to do is the view before we get started I want to explain the logic behind it first.

Instead of having multiple views where you go to update pieces of the profile, I decided to create a single view of it, basically, whenever the user needs to update a property, she can click on it, and a small pop-up appears where she can add the information without leaving the page.

So, with that in mind, here's the HTML for the header:

```
<ion-header>
  <ion-navbar color="primary">
    <ion-title>Profile</ion-title>
    <ion-buttons end>
      <button ion-button icon-only (click)="logOut()">
        <ion-icon name="log-out"></ion-icon>
      </button>
    </ion-buttons>
  </ion-navbar>
</ion-header>
```

Like on the home page, we're creating a header button to handle the logout functionality.

And inside your `<ion-content>` we're going to create a list to start adding our update items:

```
<ion-content padding>
  <ion-list>
    <ion-list-header>
      Personal Information
    </ion-list-header>
</ion-content>
```

Right after the list header, let's add the item to update our user's name:

```
<ion-item (click)="updateName()">
  <ion-grid>
    <ion-row>
      <ion-col col-6> Name </ion-col>
      <ion-col col-6 *ngIf="userProfile?.firstName || userProfile?.lastName">
        {{userProfile?.firstName}} {{userProfile?.lastName}}
      </ion-col>
      <ion-col col-6 class="placeholder-profile"
      *ngIf="!userProfile?.firstName">
        <span> Tap here to edit. </span>
      </ion-col>
    </ion-row>
  </ion-grid>
</ion-item>
```

It should be easy to understand, the labels take the left half of the grid, and the values take the right half.

> **SIDE-NOTE:** The question mark used in the main object `userProfile?.firstName` is called the Elvis operator, it tells the template first to make sure the object is there before accessing or trying to access any of its properties.

If there's no value, we will show a placeholder that says: "Tap here to edit" this will let our user know that they need to touch there to be able to select the profile items.

Right after the update name item, we're going to add an option to update the date of birth:

```
<ion-item>
  <ion-label class="dob-label">Date of Birth</ion-label>
  <ion-datetime displayFormat="MMM D, YYYY" pickerFormat="D MMM YYYY"
  [(ngModel)]="birthDate" (ionChange)="updateDOB(birthDate)"></ion-datetime>
</ion-item>
```

We could have opened a modal to update the DoB, but using the `(ionChange)` function allows us to handle everything on the same page :)

After updating the date of birth, add another function to update the user's email & password:

```
<ion-item (click)="updateEmail()">
  <ion-grid>
    <ion-row>
      <ion-col col-6> Email </ion-col>
      <ion-col col-6 *ngIf="userProfile?.email">
        {{userProfile?.email}}
      </ion-col>
      <ion-col col-6 class="placeholder-profile" *ngIf="!userProfile?.email">
        <span> Tap here to edit. </span>
      </ion-col>
    </ion-row>
  </ion-grid>
</ion-item>

<ion-item (click)="updatePassword()">
    <ion-grid>
      <ion-row>
        <ion-col col-6> Password </ion-col>
        <ion-col col-6 class="placeholder-profile">
          <span> Tap here to edit. </span>
        </ion-col>
      </ion-row>
    </ion-grid>
  </ion-item>
</ion-list>
```

Now that the HTML is in place, we are going to create the styles for it (*remember, CSS is not my thing, so if you can, improve upon this!*)

```
page-profile {
  ion-item ion-label {
    margin: 0 !important;
  }

  ion-list {
    margin: 0;
    padding: 0;
  }

  ion-list-header {
    background-color: #ECECEC;
    margin: 0;
    padding: 0;
  }

  ion-item { padding: 0 !important; }

  ion-datetime { padding-left: 3px !important; }

  .item-inner {
    border: none !important;
    padding: 0;
  }

  .popover-content {
    min-height: 0 !important;
    max-height: 88px !important;
  }

  .profile-popover {
    margin-top: -1px !important;
  }

  .placeholder-profile {
    color: #CCCCCC;
  }

  .dob-label {
    color: #000000 !important;
    padding: 10px !important;
    max-width: 50% !important;
  }
}
```

Nothing too weird, some margins and colors.

And now we're ready to start coding the functionalities for this page, the first thing we'll need to do

is import everything we'll use and inject into the constructor when necessary:

```
import { Component } from "@angular/core";
import {
  Alert,
  AlertController,
  IonicPage,
  NavController
} from "ionic-angular";
import { ProfileProvider } from "../../providers/profile/profile";
import { AuthProvider } from "../../providers/auth/auth";

@IonicPage()
@Component({
  selector: "page-profile",
  templateUrl: "profile.html"
})
export class ProfilePage {
  public userProfile: any;
  public birthDate: string;

  constructor(
    public navCtrl: NavController,
    public alertCtrl: AlertController,
    public authProvider: AuthProvider,
    public profileProvider: ProfileProvider
  ) {}
}
```

We are importing `AlertController` because we will display alerts to capture the data the user is going to update.

We are importing our profile and authentication providers because we need to call functions from both.

The userProfile variable will hold all the data from Firebase, and our birthDate variable will interact with our date picker component.

Now it is time to call our profile provider and ask for the user's profile, so right after our constructor, go ahead and create the function:

```
ionViewDidLoad() {
  this.profileProvider.getUserProfile().on("value", userProfileSnapshot => {
    this.userProfile = userProfileSnapshot.val();
    this.birthDate = userProfileSnapshot.val().birthDate;
  });
}
```

Let's break down what we did here:

- We are using `ionViewDidLoad()`, this is part of Ionic's lifecycle events, it is called after the view rendered.

- We are calling the `getUserProfile()` function from our `ProfileProvider` service, and when it returns we are assigning the value from the object to our `userProfile` variable.
- If the userProfile had a `birthDate` property stored it is going to assign it to the `birthDate` variable to use it in our date picker component.

Do you know what the `value` means in the `.on('value', function())`?

It is a value observer to a list of data and will return the entire list of data as a single snapshot which you can then loop over to access individual children.

Even when there is only a single match for the query, the snapshot is still a list; it contains a single item.

This pattern can be useful when you want to fetch all children of a list in a single operation, rather than listening to additional child added events.

The child events are:

- `child_added` retrieves lists of items or listen for additions to a list of items. This event is triggered once for each existing child and then again every time a new child is added to the specified path.
- `child_changed` Listen for changes to the items in a list. This event is triggered any time a child node is modified. This includes any modifications to descendants of the child node.
- `child_removed` Listen for items being removed from a list. This event is triggered when an immediate child is removed.
- `child_moved` Listen for changes to the order of items in an ordered list.

Each of these together can be useful for listening to changes to a specific node in a database. For example, a social blogging app might use these methods together to monitor activity in the comments of a post, as shown below:

```
const commentsRef = firebase.database().ref('post-comments/' + postId);
commentsRef.on('child_added', function(data) {
  addCommentElement(postElement, data.key, data.val().text,
      data.val().author);
});

commentsRef.on('child_changed', function(data) {
  setCommentValues(postElement, data.key, data.val().text, data.val().author);
});

commentsRef.on('child_removed', function(data) {
  deleteComment(postElement, data.key);
});
```

Ok, now that we have a better understanding of the database event triggers and that our user's profile ready to use, it is time to start adding the functions to add, modify or log out our users.

First, we will create a logout function, since it is probably the easiest one:

```
logOut(): void {
  this.authProvider.logoutUser().then(() => {
    this.navCtrl.setRoot("LoginPage");
  });
}
```

This is what you expected it to be (*since you saw it in the previous chapter*) it calls the `logoutUser` function and then it sets the `LoginPage` as our rootPage, so the user is taken to login without the ability to have a back button.

Now let's move to updating our user's name:

```
updateName(): void {
  const alert: Alert = this.alertCtrl.create({
    message: "Your first name & last name",
    inputs: [
      {
        name: "firstName",
        placeholder: "Your first name",
        value: this.userProfile.firstName
      },
      {
        name: "lastName",
        placeholder: "Your last name",
        value: this.userProfile.lastName
      }
    ],
    buttons: [
      { text: "Cancel" },
      {
        text: "Save",
        handler: data => {
          this.profileProvider.updateName(data.firstName, data.lastName);
        }
      }
    ]
  });
  alert.present();
}
```

We are creating a prompt here to ask users for their first and last name. Once we get them our "Save" button is going to call a handler, that is going to take those first and last name and send them to the `updateName` function of `ProfileProvider`.

The Birth Date is even easier since we created an `(ionChange)` inside the `<ion-datetime>` we need to call that function and use it to pass the birth date to `ProfileProvider`.

```
updateDOB(birthDate:string):void {
  this.profileProvider.updateDOB(birthDate);
}
```

Now email and password are going to be the same as the updateName function, keep in mind that we are changing the input types to email & password to get the browser validation for them:

```
updateEmail(): void {
  let alert: Alert = this.alertCtrl.create({
    inputs: [{ name: 'newEmail', placeholder: 'Your new email' },
    { name: 'password', placeholder: 'Your password', type: 'password' }],
    buttons: [
      { text: 'Cancel' },
      { text: 'Save',
        handler: data => {
          this.profileProvider
            .updateEmail(data.newEmail, data.password)
            .then(() => { console.log('Email Changed Successfully'); })
            .catch(error => { console.log('ERROR: ' + error.message); });
      }}]
  });
  alert.present();
}

updatePassword(): void {
  let alert: Alert = this.alertCtrl.create({
    inputs: [
      { name: 'newPassword', placeholder: 'New password', type: 'password' },
      { name: 'oldPassword', placeholder: 'Old password', type: 'password' }],
    buttons: [
      { text: 'Cancel' },
      { text: 'Save',
        handler: data => {
          this.profileProvider.updatePassword(
            data.newPassword,
            data.oldPassword
          );
        }
      }
    ]
  });
  alert.present();
}
```

That will create both functions and send the separate email & passwords to ProfileProvider.

And that is it, for now, at this point you should have a fully functional profile page, not only that, you also should have a better understanding of working with Objects in firebase.

If you are running into any issues, send me an email, and I will be happy to help you debug it. => j@javebratt.com

Chapter 5

Working with data as a List

We have been learning a lot about Firebase in these few chapters, ranging from authentication to CRUD (*hey, you even learned about Promises*).

In this chapter, we are going to start working with lists of data, reading them from our database to display them in our app, adding more items to those lists, and more.

The idea is to let our user start creating events, so she can keep track of the events she is hosting.

I think that is enough for an intro, so let's jump into business!

Creating the Event Provider

As always we are going to be using a provider to handle all of our event data, the reason we are using providers throughout this book is that they will help you with one of the most common programming principles: DRY, which stands for Don't Repeat Yourself.

For example, we created a user's profile, where the user can log out from the app, if we were not using a provider, we would have had to import Firebase and use Firebase's logout function directly into the `ProfilePage` class.

That is not that bad, but what if you want to add logout buttons elsewhere? Are you going to be copy/pasting the same code? Or are you going to use a provider so you can create the logout function once and then call it from anywhere you want? *Yup, me too*.

Now that we are ready let's dive into `src/providers/event/event.ts` and create the functions that are going to communicate with Firebase.

```
import { Injectable } from '@angular/core';
import firebase from 'firebase';

@Injectable()
export class EventProvider {

  constructor() {}
}
```

The first thing we are going to do here is to create a variable to hold our `userProfile` database reference so that we can use it in all of our functions.

```
public eventListRef: firebase.database.Reference;
constructor() {
  firebase.auth().onAuthStateChanged(user => {
    if (user) {
      this.eventListRef = firebase
        .database()
        .ref(`/userProfile/${user.uid}/eventList`);
    }
  });
}
```

Now it is time to start thinking what kind of features we need in our provider. We want our users to be able to:

- Create new events.
- Get the full list of events.
- Get a particular event from the list.

Knowing that, let's start with creating a new event:

```
createEvent(
  eventName: string,
  eventDate: string,
  eventPrice: number,
  eventCost: number
): firebase.database.ThenableReference {
  return this.eventListRef.push({
    name: eventName,
    date: eventDate,
    price: eventPrice * 1,
    cost: eventCost * 1,
    revenue: eventCost * -1
  });
}
```

A couple of things to note:

- We are using `.push()` on the `eventList` node because we want firebase to append every new object to this list, and to auto-generate a random ID, so we know there aren't going to be two objects with the same ID.
- We are adding the name, date, ticket price and cost of the event (Mostly because in the next chapter I am going to use them for transactions + real-time updates on revenue per event.)

After we have the function that is going to create our events, we'll need one more to list them.

```
getEventList(): firebase.database.Reference {
  return this.eventListRef;
}
```

And one for receiving an event's ID and returning that event:

```
getEventDetail(eventId:string): firebase.database.Reference {
  return this.eventListRef.child(eventId);
}
```

This will be it for now (*we will return to this page for some cool stuff later*), we are going to start playing with our events to see what we can do.

Setting up the HomePage

Now since I have not given much thought to the app's UI, I am going to create two buttons on the HomePage to take me to the EventCreatePage or the EventListPage.

Go to home.html and add the buttons inside the <ion-content> tag:

```
<ion-content padding>
  <button ion-button block color="primary" (click)="goToCreate()">
    Create a new Event
  </button>

  <button ion-button block color="primary" (click)="goToList()">
    See your events
  </button>
</ion-content>
```

Now go into home.ts and create those functions:

```
goToCreate(): void {
  this.navCtrl.push('EventCreatePage');
}

goToList(): void {
  this.navCtrl.push('EventListPage');
}
```

Creating new events

Now that we have that it is time to create the event part of the app, we are going to start with adding a new event, for that go to event-create.html and create a few inputs to save the event's name, date, ticket price and costs:

```
<ion-header>
  <ion-navbar>
    <ion-title>New Event</ion-title>
  </ion-navbar>
</ion-header>

<ion-content padding>
  <ion-item>
    <ion-label stacked>Event Name</ion-label>
    <ion-input [(ngModel)]="eventName" type="text"
    placeholder="What's your event's name?">
    </ion-input>
  </ion-item>

  <ion-item>
    <ion-label stacked>Price</ion-label>
    <ion-input [(ngModel)]="eventPrice" type="number"
    placeholder="How much will guests pay?">
    </ion-input>
  </ion-item>

  <ion-item>
    <ion-label stacked>Cost</ion-label>
    <ion-input [(ngModel)]="eventCost" type="number"
    placeholder="How much are you spending?">
    </ion-input>
  </ion-item>

  <ion-item>
    <ion-label>Event Date</ion-label>
    <ion-datetime displayFormat="D MMM, YY" pickerFormat="DD MMM YYYY"
    [(ngModel)]="eventDate">
    </ion-datetime>
  </ion-item>

  <button ion-button block
    (click)="createEvent(eventName, eventDate, eventPrice, eventCost)">
    Create Event
  </button>
</ion-content>
```

Nothing we have not seen in previous examples, we are using a few inputs to get the data we need, and then creating a `createEvent()` function and passing it those values so we can use them later.

After you finish doing this, go to `event-create.ts` first we will need to import our `EventProvider`.

```
import { Component } from "@angular/core";
import { IonicPage, NavController } from "ionic-angular";
import { EventProvider } from "../../providers/event/event";

@IonicPage()
@Component({
  selector: "page-event-create",
  templateUrl: "event-create.html"
})
export class EventCreatePage {
  constructor(
    public navCtrl: NavController,
    public eventProvider: EventProvider
  ) {}
}
```

After doing that we will create our `createEvent` function, it will send the data to the `createEvent()` function we already declared in our EventProvider.

```
createEvent(
  eventName: string,
  eventDate: string,
  eventPrice: number,
  eventCost: number
): void {
  this.eventProvider
    .createEvent(eventName, eventDate, eventPrice, eventCost)
    .then(newEvent => {
      this.navCtrl.pop();
    });
}
```

Nothing too crazy, we are sending the data to our `EventProvider`, and as soon as we create the event, we are using `this.navCtrl.pop()` to go back a page to the `HomePage`.

We use `this.navCtrl.pop();` because it is a good practice to redirect the user after a form submits, this way we avoid the user clicking multiple times the submit button and create several entries.

Listing the events

Now that we can create events, we need a way to see our events, so let's go into the `event-list` folder and inside the `event-list.html` file and create a list of your events:

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


<ion-content padding>
  <ion-list>
    <ion-item *ngFor="let event of eventList"
    (click)="goToEventDetail(event.id)">
      <h2>{{event?.name}}</h2>
      <p>Ticket: <strong>${{event?.price}}</strong></p>
      <p>Date: <strong>{{event?.date}}</strong></p>
    </ion-item>
  </ion-list>
</ion-content>
```

- We are creating an item that will repeat itself for every event we have in our database.
- We are showing necessary event data like the name, the ticket price for guests and the event date.
- When users tap on the event, they are going to be taken to the event's detail page.
- We are sending the event id in the `goToEventDetail()` function so we can pull the specific ID from Firebase.

Now we need the logic to implement all of that so go into `event-list.ts` and first import and declare everything you'll need:

```
import { Component } from "@angular/core";
import { IonicPage, NavController } from "ionic-angular";
import { EventProvider } from "../../providers/event/event";

@IonicPage()
@Component({
  selector: "page-event-list",
  templateUrl: "event-list.html"
})
export class EventListPage {
  public eventList: Array<any>;

  constructor(
    public navCtrl: NavController,
    public eventProvider: EventProvider
  ) {}
}
```

- We are importing `EventProvider` to call the provider's functions.
- We are declaring a variable called `eventList` to hold our list of events.

Now we need to get that list of events from Firebase. We do not want Ionic to cache this view, so we are going to use the same LifeCycle function we used in the previous chapter:

```
ionViewDidLoad() {
  this.eventProvider.getEventList().on("value", eventListSnapshot => {
    this.eventList = [];
    eventListSnapshot.forEach(snap => {
      this.eventList.push({
        id: snap.key,
        name: snap.val().name,
        price: snap.val().price,
        date: snap.val().date
      });
      return false;
    });
  });
}
```

We have done this before. We are:

- Calling the `getEventList()` method from our provider.
- Pushing every record into our `eventList` array.

Now the first part of the HTML will work, it is going to show users a list of their events in the app.

However, we are not done yet. We need to create the function to send users to the event detail, and we will do that right here:

```
goToEventDetail(eventId):void {
  this.navCtrl.push('EventDetailPage', { eventId: eventId });
}
```

The function is receiving the event's ID and sending it as a parameter that we can retrieve in the event detail's page, that way we know what event we are going to pull from the database.

The Event Detail Page

Now that we are sending the user to the event detail page, and we are passing the event ID we have everything we need to show the event details.

Go into `event-detail.ts`, you're going to import and initialize:

```
import { Component } from "@angular/core";
import { IonicPage, NavController, NavParams } from "ionic-angular";
import { EventProvider } from "../../providers/event/event";

@IonicPage({
  segment: "event-detail/:eventId"
})
@Component({
  selector: "page-event-detail",
  templateUrl: "event-detail.html"
})
export class EventDetailPage {
  public currentEvent: any = {};

  constructor(
    public navCtrl: NavController,
    public navParams: NavParams,
    public eventProvider: EventProvider
  ) {}
}
```

We are importing `NavParams` because that is the module that handles navigation parameters (*like the event ID we sent to this page*)

And we are creating `currentEvent` to hold our event's information, and now it is time to pull that information from our Firebase database:

```
ionViewDidLoad() {
  this.eventProvider
    .getEventDetail(this.navParams.get("eventId"))
    .on("value", eventSnapshot => {
      this.currentEvent = eventSnapshot.val();
      this.currentEvent.id = eventSnapshot.key;
    });
}
```

Now that we have our event's information available, we can display it in our `event-detail.html` file:

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

<ion-content>
  <ion-card>
    <ion-card-header>
      {{currentEvent?.name}}
    </ion-card-header>
    <ion-card-content>
      <p>Ticket: <strong>${{currentEvent?.price}}</strong></p>
      <p>Date: <strong>{{currentEvent?.date}}</strong></p>
    </ion-card-content>
  </ion-card>
</ion-content>
```

We are not spending much time or brain power with the UI of this page because it is going to get heavily modified in our next chapter.

And that's it if you felt this was too easy or the chapter was too simple that was the point, I wanted to create a small introduction to lists because we are going to get a little deeper in our next chapter and I wanted you to be ready for it.

And as always, if you run into any problems with the code let me know by emailing j@javebratt.com

Adding guest to an event

We are going to keep working with lists, right now we will start creating a list of guest for an event.

We do not need to create any more pages than there are, we already have everything we need from the previous chapter.

The first thing we will need to do is to change some stuff from the previous chapter.

Go into `src/providers/event/event.ts` and we will add a new function there to add the guests to the event so inside the provider add the function:

```
addGuest(guestName: string, eventId: string, eventPrice: number
): PromiseLike<any> {
  return this.eventListRef.child(`${eventId}/guestList`).push({ guestName });
}
```

Something simple, we are pushing a new guest to the event, it only has the guest's name (*for now*).

This way we are saving an event's guest list, we might need it later.

Now we can focus on the `event-detail` folder, go to `event-detail.ts` and let's create an `addGuest()` function, we will use it for adding new guests to our events, go ahead and add it:

```
addGuest(guestName: string): void {
  this.eventProvider.addGuest(
    guestName,
    this.currentEvent.id,
    this.currentEvent.price
  )
  .then(newGuest => { this.guestName = ""; });
}
```

Nothing weird going on, we are calling the addGuest function from our event provider, then passing it the `guestName`, our event's id and the event's ticket price (*we will use this last one for updating the revenue*).

Remember to declare the the `guestName` variable we're using:

```
public guestName: string = '';
constructor(...){...}
```

Now we are going to be adding some things to our `event-detail.html` file.

The first thing we will add is a row to show our revenue updating in real time (*Yes, we will get to the JS for that part soon*) go ahead and add a row with a label and the event's revenue property:

```
<ion-content>
  <ion-row padding>
    <ion-col col-8>Current Revenue</ion-col>
    <ion-col col-4 [class.profitable]="currentEvent?.revenue > 0"
      [class.no-profit]="currentEvent?.revenue < 0">
      {{currentEvent?.revenue}}
    </ion-col>
  </ion-row>
```

```
  <ion-card>
    <ion-card-header>{{currentEvent?.name}}</ion-card-header>
    <ion-card-content>
      <p>Ticket: <strong>${{currentEvent?.price}}</strong></p>
      <p>Date: <strong>{{currentEvent?.date}}</strong></p>
    </ion-card-content>
  </ion-card>
</ion-content>
```

There we are adding the event's revenue, and we are telling Ionic, hey if the income is less than 0 add the `no-profit` CSS class, and if the income is greater than 0 go ahead and add the `profitable` CSS class.

By the way, `profitable` adds green color and `no-profit` red color, in fact, take the CSS for this file, add it straight into `event-detail.scss`

```
page-event-detail {
  ion-card-header { text-align: center; }

  .add-guest-form {
    button { margin-top: 16px; }
  }

  .profitable { color: #22BB22; }

  .no-profit { color: #FF0000; }
}
```

And lastly we will need a way to add our guests, create a text input for the guest's name and a button to send the info:

```
<ion-card class="add-guest-form">
  <ion-card-header>
    Add Guests
  </ion-card-header>
  <ion-card-content>
    <ion-item>
      <ion-label stacked>Name</ion-label>
      <ion-input [(ngModel)]="guestName" type="text"
      placeholder="What's your guest's name?"></ion-input>
    </ion-item>

    <button ion-button color="primary" block (click)="addGuest(guestName)"
    [disabled]="!guestName">
      Add Guest
    </button>
  </ion-card-content>
</ion-card>
```

This calls the `addGuest()` function from `event-detail.ts` and adds the guest to the event.

Creating a Firebase `.transaction()`

But Jorge, how does this updates the revenue for the event?

I wanted to leave this part for last because it needs a little explanation of **why** first.

I am going to use a feature from Firebase called `transaction()` is a way to update data to ensure there's no corruption when being updated by multiple users.

For example, let's say Mary downloads the app, but she soon realizes that she needs some help at the front door, there are way too many guests and if she is the only one registering them it is going to take too long.

So she asks Kate, Mark, and John for help, they download the app, log in with Mary's password (*Yeah, it could be a better idea to make it multi-tenant* :P), and they start registering users too.

What happens if Mark & Kate both register new users, when Mark's click reads the revenue it was $300 so his app took those $300, added the $15 ticket price and the new revenue should be $315 right? Wrong!

It turns out that Kate registered someone else a millisecond earlier, so the revenue already was at $315 and Mark set it to $315 again, you see the problem here right?

This is where transactions come in. They update the data safely. The update function takes the current state of the data as an argument and returns the new desired state you would like to write.

If another client writes to the location before you store your new value, your update function is called again with the new current value, and the Firebase retries your write operation.

And they are not even hard to write, go ahead to `src/providers/event/event.ts` and add a `.then()` function for the `addGuest()` it used to look like this:

```
addGuest(guestName:string, eventId:string, eventPrice:number):
 PromiseLike<any> {
  return this.eventListRef.child(`${eventId}/guestList`).push({ guestName })
}
```

Now it should look like this:

```
addGuest(guestName: string, eventId: string, eventPrice: number
): PromiseLike<any> {
  return this.eventListRef
    .child(`${eventId}/guestList`)
    .push({ guestName })
    .then(newGuest => {
      this.eventListRef.child(eventId).transaction(event => {
        event.revenue += eventPrice;
        return event;
      });
    });
}
```

The transaction takes the current state of the event and updates the revenue property for it, and then it returns the new value making sure it is correct.

I need to stop this chapter here, mainly because I ran out of coffee, see you in the next chapter!

Chapter 6

Firebase Storage

In this chapter we are going to be using one of Firebase's best features, Storage, it will let you store binary data in your Firebase application, meaning you can upload files :)

We are also going to use the phone's camera (*that's why we installed the Camera plugin in Chapter #1*), the idea is to let our users take a picture of their guests when they are adding them to the guest list.

Since we already have the plugin installed let's jump into the code.

Taking pictures

Now everything is ready to start working with the Camera Plugin, so go to `event-detail.html` and create a button to take the guest's picture, we'll add a nice camera looking icon next to the guest name:

```
<ion-row>
  <ion-col col-8>
    <ion-item>
      <ion-label stacked>Name</ion-label>
      <ion-input [(ngModel)]="guestName" type="text"
        placeholder="What's your guest's name?"></ion-input>
    </ion-item>
  </ion-col>

  <ion-col col-4>
    <button ion-button icon-only (click)="takePicture()">
      <ion-icon name="camera"></ion-icon>
    </button>
  </ion-col>
</ion-row>
<span *ngIf="guestPicture">Picture taken!</span>
```

Right after the name input I am adding a message that says the picture was taken and it only shows if the guestPicture property exists (*that will make more sense in the `event-detail.ts` file*)

And then a button to call the `takePicture()` function.

Now go to `event-detail.ts` and first import the Camera plugin:

```
import { Camera } from '@ionic-native/camera';
```

Now we need to inject it into our constructor:

```
constructor(
  public navCtrl: NavController,
  public navParams: NavParams,
  public eventProvider: EventProvider,
  public cameraPlugin: Camera
) {}
```

After that you will create a variable to hold the guest's picture, right before the `constructor()` add:

```
public guestPicture: string = null;
```

And add that property as a parameter in the `addGuest` function:

Before:

```
addGuest(guestName) {
  this.eventProvider.addGuest(guestName, this.currentEvent.id,
      this.currentEvent.price).then( () => {
    this.guestName = '';
  });
}
```

Now:

```
addGuest(guestName: string): void {
  this.eventProvider
    .addGuest(
      guestName,
      this.currentEvent.id,
      this.currentEvent.price,
      this.guestPicture
    )
    .then(newGuest => {
      this.guestName = "";
      this.guestPicture = null;
    });
}
```

We are passing the `this.guestPicture` variable to the `addGuest()` function on our event provider, don't worry if it gives you an error, the function is not declared for those parameters, and we will fix that once we move to edit our provider.

Then we are setting `this.guestPicture` to null to make sure the message *"picture taken"* is not shown.

Now we need to create the `takePicture()` function that's going to open the camera and allow us to take a picture of our guest, and it is an extended function, so I am going to paste it here and then explain the different parts of it:

```
takePicture(): void {
  this.cameraPlugin
    .getPicture({
      quality: 95,
      destinationType: this.cameraPlugin.DestinationType.DATA_URL,
      sourceType: this.cameraPlugin.PictureSourceType.CAMERA,
      allowEdit: true,
      encodingType: this.cameraPlugin.EncodingType.PNG,
      targetWidth: 500,
      targetHeight: 500,
      saveToPhotoAlbum: true
    })
    .then(
      imageData => {
        this.guestPicture = imageData;
      },
      error => {
        console.log("ERROR -> " + JSON.stringify(error));
      }
    );
}
```

The first part of the function is the call to the Camera plugin:

```
this.cameraPlugin
  .getPicture({
    quality: 95,
    destinationType: this.cameraPlugin.DestinationType.DATA_URL,
    sourceType: this.cameraPlugin.PictureSourceType.CAMERA,
    allowEdit: true,
    encodingType: this.cameraPlugin.EncodingType.PNG,
    targetWidth: 500,
    targetHeight: 500,
    saveToPhotoAlbum: true
  })
```

There we are calling the Camera plugin and giving it a few options:

- `quality` => The quality we want our picture to have on a scale of 1 - 100.
- `destinationType` => This gives you the return type, DATA_URL is set to return a base64 string of the image, you can also return the image's URI.
- `sourceType` => We are telling it to open the camera, you can change this to get the picture from the photo library.
- `allowEdit` => Allows users to edit the picture, mostly crop it.
- `encodingType`: We set the encoding for the image: png, jpg.

- `targetWidth` & `targetHeight` => This gives you the image size in px.
- `saveToPhotoAlbum` => This saves the image to the gallery after being taken.

The next part of the code is setting that result to `this.guestPicture`.

This will now be sent in the `addGuest()` function from above, so it is time to move to our provider and edit that.

Go to the `src/providers/event/event.ts` and find the `addGuest()` function:

```
addGuest(guestName: string, eventId: string, eventPrice: number
): PromiseLike<any> {
  return this.eventListRef
    .child(`${eventId}/guestList`)
    .push({ guestName })
    .then(newGuest => {
      this.eventListRef.child(eventId).transaction(event => {
        event.revenue += eventPrice;
        return event;
      });
    });
}
```

The first thing to do here is to add the picture parameter:

```
addGuest(
  guestName: string,
  eventId: string,
  eventPrice: number,
  guestPicture: string = null
): PromiseLike<any> {...}
```

I am adding it and setting a default to null in case the guest does not want his picture taken.

Now we are going to add the code that takes the picture, saves it to Firebase Storage and then goes into the guest details and adds the URL to the image we saved:

```
if (guestPicture != null) {
  firebase
    .storage()
    .ref(`/guestProfile/${newGuest.key}/profilePicture.png`)
    .putString(guestPicture, 'base64', { contentType: 'image/png' })
    .then(savedPicture => {
      this.eventListRef
        .child(`${eventId}/guestList/${newGuest.key}/profilePicture`)
        .set(savedPicture.downloadURL);
    });
}
```

We are creating a reference to our Firebase Storage:

```
guestProfile/guestId/profilePicture.png
```

And that is where we store our file, to save it we use the `.putString()` method, and pass it the `base64` string we got from the Camera Plugin.

After we upload the image to Firebase Storage we create a database reference to the guest we created and create a `profilePicture` property, and then we set that property to the picture's download URL, you get that with: `savedPicture.downloadURL`.

And that is it. Now you have a fully functional way of capturing photos with your camera and uploading them to Firebase Storage.

See you in the next section!

Chapter 7

Security Rules

We are going to start preparing our app to go public, so the first thing we will need to do is update our security rules on the server, we do not want people connecting to the app and having access to someone else's data.

Database Security

There's a comprehensive guide to security rules in Firebase Docs, and I have kept them simple for this app because I do believe that if you structure your data correctly, they do not need to be hard.

So, to structure your security rules, you will need to go to your firebase console:

console.firebase.google.com/project/YOURAPPNAMEHERE/database/data

By default the rules are there to allow access to only authenticated users:

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

We need to set them, so it also checks that the user trying to access the information is the **correct** user.

```
{
  "rules": {
    "userProfile": {
      "$uid": {
        ".read": "auth != null && $uid === auth.uid",
        ".write": "auth != null && $uid === auth.uid"
      }
    }
  }
}
```

There we are saying that under the `userProfile` node there's going to be a variable called uid when you add the $ sign in here, it takes the value as a variable.

And we are saying that for a user to have read or write permissions to that node, their auth.uid needs to match the $uid variable.

In here auth is a variable that holds the authentication methods/properties.

There we ensure that only the user who owns the data can write/read it.

Storage Security

You should also set up rules for **Firebase Storage**, that way you can protect your users' files.

You will need to go to:

console.firebase.google.com/project/YOURAPPGOESHERE/storage/rules

Identifying your user is only part of security. Once you know who they are, you need a way to control their access to files in Cloud Storage.

Cloud Storage lets you specify per file and per path authorization rules that live on our servers and determine access to the files in your app. For example, the default Storage Security Rules require Firebase Authentication to perform any read or write operations on all data:

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Data Validation

Firebase Security Rules for Cloud Storage can also be used for data validation, including validating file name and path as well as file metadata properties such as `contentType` and `size`:

```
service firebase.storage {
  match /b/{bucket}/o {
    match /images/{imageId} {
      // Only allow uploads of any image file that's less than 5MB
      allow write: if request.resource.size < 5 * 1024 * 1024
                   && request.resource.contentType.matches('image/.*');
    }
  }
}
```

Chapter 8

Next Steps

By now you should have a greater understanding of how to work with the Firebase JavaScript SDK and how to integrate it with Ionic Framework.

Now, I want you to do two things:

1. Take a break, watch some TV or whatever else you like to do for fun.
2. Use this knowledge to build a simple app with the JavaScript SDK, when you are done upload it to Github and send it to me :)

Congratulations on finishing the book, if you want to go into more in-depth use cases using Ionic and Firebase, like AngularFire2, Cloud Functions, and Push Notifications, make sure to check my PREMIUM course at https://javebratt.com/ionic-firebase-book/

Chapter 9

CHANGELOG

1.0.4 Updates to latest versions (*No breaking changes*.)

1.0.3 This update is dedicated to fixing typos.

1.0.2 Fixes firebase type setting bugs:

In the last update we moved away from `firebase.Promise` to a different set of types from Firebase, this update fixes the ones that were missing and still showed old versions in the book.

1.0.1 Updates to Firebase 4.5.0 and fixes TypeScript Errors

- In the `auth.ts` provider we change all instances of `: firebase.Promise<any>` for `: Promise<any>`.
- In the `profile.ts` provider we change all instances of `: firebase.Promise<any>` for `: Promise<any>`.
- In the `event.ts` provider we change the return type of `createEvent()` to `firebase.database.Thenable`
- In the `event.ts` provider we change the return type of `addGuest()` to `PromiseLike<any>`.
- When adding the `addGuest()` function forgot to also declare the `guestName` variable, so adding `public guestName: string =""` should fix the TypeScript error.

1.0.0

First launch.