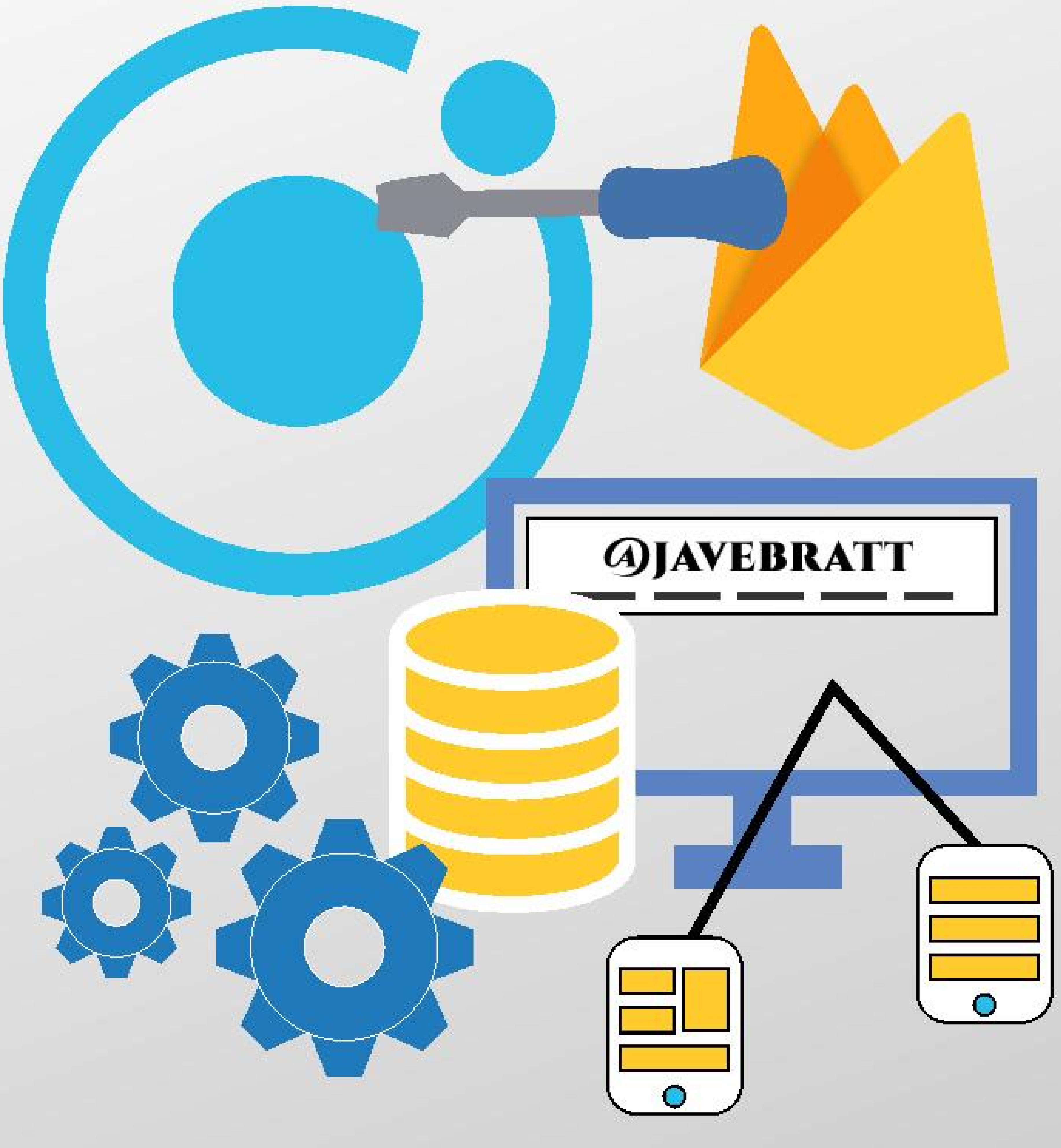
Building Firestore Powered Ionic Apps



Jorge Vergara

To my wife Evelyn and my son Emmanuel

You made this happen

Contents

Chapter 1

Introduction

First of all, **Thank you!**

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

Though 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 several applications together, and I will guide you through the development of the apps 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

This book uses **Ionic 4 and Firebase 5** for the applications we'll build together.

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 add it to 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 2

The FireStore Database Explained

In this chapter we'll go through the core concepts of the FireStore database, and how it differs from the Real-time database.

We'll cover things like Documents and Collections and then go into a bit more detail on specific tasks like creating, reading, updating, and deleting data from Firestore.

A Document-oriented database

FireStore is a NoSQL document-oriented database. It's a big difference from the Real-time database (*going to refer to that one as RTDB from now on*) which is also NoSQL.

The RTDB is a gigantic JSON tree where anything goes, and you can set up the information however you want.

Firestore instead is a document-oriented database, which offers more structure, all of your data is stored in objects called documents, and these documents can have any number of things inside (*booleans, strings, or even other objects*).

And documents are grouped into *collections*.

For example, let's imagine we want to store user information in our database, we would create a collection called `users` or in our app, we're calling it `userProfile`.

Now, inside the collection we'd find objects that hold the information to each user's profile, those are the documents.

One crucial thing, documents can't have other documents stored inside of them, for example, the user `javebratt` can't have the user `booya` as one of its properties.

But documents can store other sub-collections inside, for example, the user `javebratt` can have a collection of `tasks` as one of its properties, and that sub-collection can hold the task documents.

Reading Data from FireStore

To read data from the database, we have two options, we can either get a collection of items (*think of it as a list*), or we can get a specific document from the database (*like an object*).

To read an object from the database, all you need to do is to create a reference pointing to that document, for example:

```
constructor(private fireStore: AngularFirestore) {
  this.userDoc = fireStore.doc<any>('userProfile/we45tfgy8ij');
}
```

We're pointing to the document with the ID of `we45tfgy8ij` inside the `userProfile` collection.

If you want to fetch the entire user collection it would be something like:

```
constructor(private fireStore: AngularFirestore) {
  this.userProfileCollection = fireStore.collection<any>('userProfile');
}
```

You can also query users based on specific properties, let's say our users have a property called `teamAdmin` and we want to fetch the profiles of all the users who are admins of a team.

```
constructor(private fireStore: AngularFirestore) {
  this.teamAdminCollection = fireStore.collection<any>('userProfile', ref =>
    ref.where('teamAdmin', '==', true));
}
```

Adding data to FireStore

To read data from the database we first need to have some data there, D'OH! (*Imagine Homer Simpson's voice there*).

To push objects to the database we have two main options, if we know the ID we want to give to the document, we can use something like this:

```
constructor(private fireStore: AngularFirestore) {
  this.userDoc = fireStore.doc<any>('userProfile/we45tfgy8ij');
  this.userDoc.set({
    name: 'Jorge Vergara',
    email: 'j@javebratt.com',
    // Other info you want to add here
  })
}
```

If we don't care about the generated ID, we can just push the documents to the collection and let Firebase autogenerate those IDs.

```
constructor(private fireStore: AngularFirestore) {
  this.userProfileCollection = fireStore.collection<any>('userProfile');
  this.userProfileCollection.push({
    name: 'Jorge Vergara',
    email: 'j@javebratt.com',
    // Other info you want to add here
  });
}
```

Keep in mind that this way the ID for the user document won't be the same as the user's `uid`, Firebase will autogenerate an ID for that document.

Updating data from FireStore

You've seen that the database API is very straight-forward, so, can you imagine how the method of updating is called?

… *Dramatic Silence* …

Yeah, it's called `.update()`, if we want to change the user's name, for example, we will do something like this:

```
constructor(private fireStore: AngularFirestore) {
  this.userDoc = fireStore.doc<any>('userProfile/we45tfgy8ij');

  this.userDoc.update({
    name: 'Jorge Vergara',
    email: 'j@javebratt.com',
    // Other info you want to add here
  })
}
```

Remove data from FireStore

There are a few ways we can remove data:

- Removing a specific document (*an object*).
- Removing a property or field from a document.
- Removing an entire collection.

Let's explore them in that order, for example, I deleted a user from the Auth part of Firebase, and now I want to remove the user profile from the database:

```
constructor(private fireStore: AngularFirestore) {
  fireStore.doc<any>('userProfile/we45tfgy8ij').delete();
}
```

Or let's say we don't want to delete our user, but the user wrote and said to please delete his age because he didn't want anyone to know how old he was, in that case, we would fetch the user and then remove the date field:

```
constructor(private fireStore: AngularFirestore) {
  fireStore.doc<any>('userProfile/we45tfgy8ij').update({
    age: firebase.firestore.FieldValue.delete()
  });
}
```

Deleting collections is a bit trickier, right now there's no way to remove an entire collection in bulk, this is because a fail-safe FireStore did.

It was designed to avoid several apparent outages: if a `delete()` call has to delete huge swaths of data, all other activity is effectively locked out until it completes. As a result even for RTDB users that want to delete large quantities of data the Firebase team recommends recursively finding and deleting documents in groups.

There's a workaround for this, but it's something to do carefully because it's a DESTRUCTIVE operation, you can use the **Firebase CLI:**

```
firebase firestore:delete [options] <<path>>
```

It would work fine for when you're removing a bunch of test data, but my advice would be to avoid it when you're working with production data.

Now that we have a better understanding of the APIs we're going to see them in action in the next chapter when we build our first app together.

Chapter 3

Event Manager – The Setup Process

What we will build

In this first module, 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, it covers a lot of what you need to build your own apps.

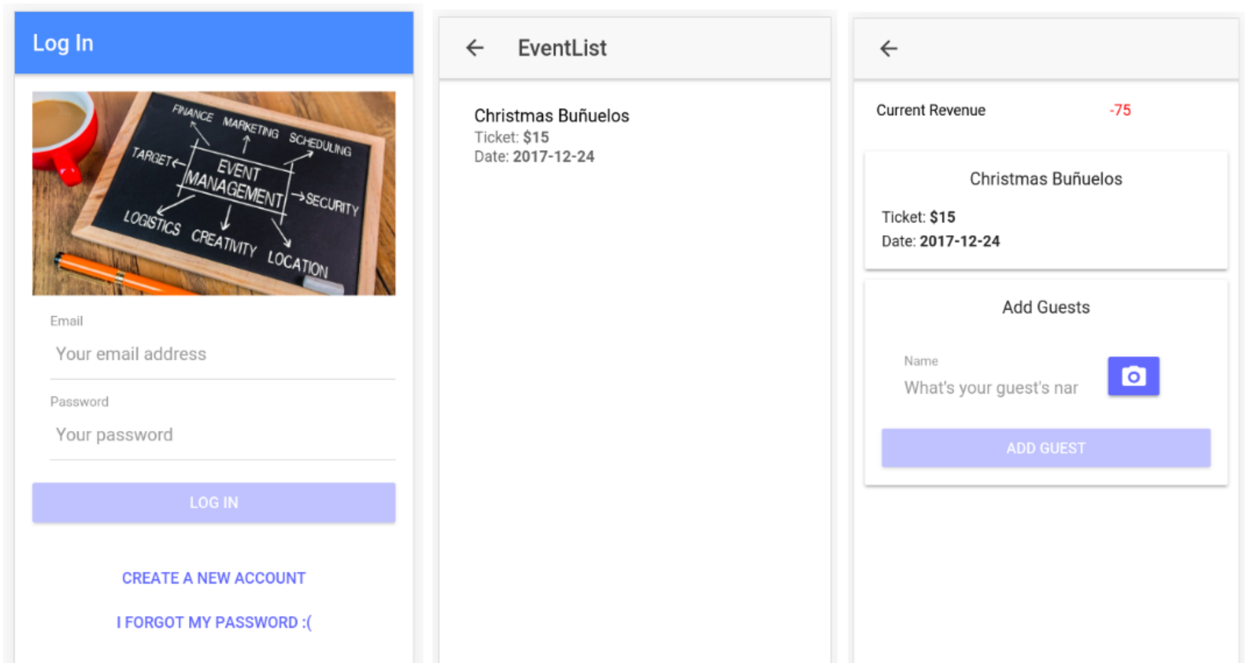


Figure 3.1: Event Manager Work flow

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

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

You can find the source code for this app in the GitHub Repo.

We're going to use the regular Web SDK for Firestore in our first application, and later move to AngularFire.

The reason behind this is simple, AngularFire provides wrappers to the Web SDK, so it's a good idea to know how the Web SDK works.

Make sure your development environment is 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.

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

```
npm install -g ionic
```

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 event-manager blank --type=angular
cd event-manager
```

The `start` command will create a new app, and the `--type=angular` flag will create an Ionic app using angular as a framework for handling the logic.

The CLI is going to ask you if you want to add Cordova and Ionic Appflow SDK, **choose no** in both prompts, we're using Capacitor for the native functionality and we'll cover everything we need to know about it soon.

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 `event-manager`.

- 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": "^7.2.2",
  "@angular/core": "^7.2.2",
  "@angular/forms": "^7.2.2",
  "@angular/http": "^7.2.2",
  "@angular/platform-browser": "^7.2.2",
  "@angular/platform-browser-dynamic": "^7.2.2",
  "@angular/router": "^7.2.2",
  "@ionic-native/core": "^5.0.0",
  "@ionic-native/splash-screen": "^5.0.0",
  "@ionic-native/status-bar": "^5.0.0",
  "@ionic/angular": "^4.1.0",
  "core-js": "^2.5.4",
  "rxjs": "~6.5.1",
  "tslib": "^1.9.0",
  "zone.js": "~0.8.29"
},
```

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.

Let’s take a moment to remove the `@ionic-native` packages, open the terminal and type:

```
npm rm @ionic-native/core @ionic-native/splash-screen @ionic-native/status-bar
```

Capacitor has working APIs to replace those plugins so there’s no need to keep them installed anymore.

Don’t forget to go into the `app.module.ts` and the `app.component.ts` files and remove all the reference for `@ionic-native` there.

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 services

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 service we are going to use on our app. First, the pages, go ahead and open your terminal and start generating them:

```
ionic generate page pages/event-create
ionic generate page pages/event-detail
ionic generate page pages/event-list
ionic generate page pages/login
ionic generate page pages/profile
ionic generate page pages/reset-password
ionic generate page pages/signup
```

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

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

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

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

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

`event-create.module.ts` is the module file, in the past, we used to declare and initialize pages in `app.module.ts`, Angular 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 { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { IonicModule } from '@ionic/angular';
import { EventCreatePage } from './event-create.page';

const routes: Routes = [
  {
    path: '',
    component: EventCreatePage
  }
];

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    IonicModule,
    RouterModule.forChild(routes)
  ],
  declarations: [EventCreatePage]
})
export class EventCreatePageModule {}
```

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-create.scss` is a blank style file, where we will be making our apps “prettier.”

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

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

@Component({
  selector: 'app-event-create',
  templateUrl: './event-create.page.html',
  styleUrls: ['./event-create.page.scss']
})
export class EventCreatePage implements OnInit {
  constructor() {}

  ngOnInit() {}
}
```

And lastly, it creates a `event-create.page.specs.ts` that is used for testing purposes:

```
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';
import { async, ComponentFixture, TestBed } from '@angular/core/testing';

import { EventCreatePage } from './event-create.page';

describe('EventCreatePage', () => {
  let component: EventCreatePage;
  let fixture: ComponentFixture<EventCreatePage>;

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [EventCreatePage],
      schemas: [CUSTOM_ELEMENTS_SCHEMA]
    }).compileComponents();
  }));

  beforeEach(() => {
    fixture = TestBed.createComponent(EventCreatePage);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

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

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

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

```
ionic generate service services/user/auth
ionic generate service services/user/profile
ionic generate service services/event/event
```

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

When you run `ionic generate service services/user/auth` the CLI is going to create a folder called `services/user`, and that folder will have a file called `auth.service.ts` here's what you will see in `auth.service.ts`.

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

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  constructor() {}
}
```

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

Capacitor

We're going to use Capacitor to communicate with the native layer of the phone, for this app we need to talk to 3 different native parts.

The Statusbar so we can style it depending on the colors of our app, the Splash Screen so we can hide it when our app is ready to run, and the camera so our users can take pictures using our app.

You can find Capacitor's official docs here, but I'll do my best to explain every bit we need to get our functionality ready.

Let's start with understanding a bit more about Capacitor and its role on mobile development. The easiest way to "get it" is by seeing it as a replacement for Cordova.

Capacitor allows you to wrap your app into a native container and helps you communicate with the phone's native capabilities. You can use it to create a native build of your application (iOS and Android).

Configuring Capacitor

The first thing we need to do to 'activate' capacitor is to enable it through the Ionic CLI, so open your terminal (*while inside of the project's folder*) and type:

```
ionic integrations enable capacitor
```

It will create the necessary files and install the capacitor packages, one thing to keep in mind, it creates a file called `capacitor.config.json` where you need to go and edit your `appId`.

By default it sets it up to be `io.ionic.starter`, it's a good idea to change it to something more on-brand. The usual convention is to go with `com.myDomainName.myProjectName` so I'll name it `com.javebratt.eventManager`.

Before adding the android/ios platforms we need to generate a build of our project or else Capacitor will have unexpected errors. Open your terminal and type:

```
ionic build
```

That command will generate a build of our application inside the `www/` folder which is the folder Capacitor is watching.

> SIDE-NOTE: That command generates a development build, if you want to generate a production build for your application add the `--prod` flag.

Now we can add our platforms, you can add either iOS or Android by using the commands:

```
ionic cap add ios
ionic cap add android
```

Remember that you need XCode installed on a Mac to build for iOS. Once you add your platforms it's time to sync them and copy over the build to the platform respective folder. You do that by opening up the terminal and typing:

```
ionic cap sync
```

Now Capacitor is ready to use, we will cover the Camera plugin in the **Firebase Storage** chapter, but for now, let's replace the `@ionic-native` integration for the Status bar and the Splash Screen.

Open your `app.component.ts`, replace all Ionic Native references for the plugins and replace them with the Capacitor ones, it should look like this:

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

const { SplashScreen, StatusBar } = Plugins;

@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html'
})
export class AppComponent {
  constructor() {
    this.initializeApp();
  }

  initializeApp() {
    SplashScreen.hide().catch(error => {
      console.error(error);
    });

    StatusBar.hide().catch(error => {
      console.error(error);
    });
  }
}
```

We're importing the plugins and then hiding both the status bar and the splash screen, if there's an error (*like if the app is running on the web*) we can handle it gracefully without annoying our users.

Right now you should have the skeleton of your app ready, go ahead and run `ionic serve` in the terminal and make sure it's running. If it's not and you can't figure out why, feel free to shoot me an email and I'll be more than happy to help you debug what's going wrong.

Next Steps

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

Chapter 4

Event Manager – 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 chapter 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 `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 * as firebase from 'firebase/app';
```

That line of code right there imports the Firebase core functionality in the 'firebase' namespace, we'll use it to initialize our app.

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.

Now we need to go to our Firebase Console on the web, open a browser and navigate to https://console.firebase.google.com.

You'll be asked to log in with your Google account, and then you'll see a dashboard like this.

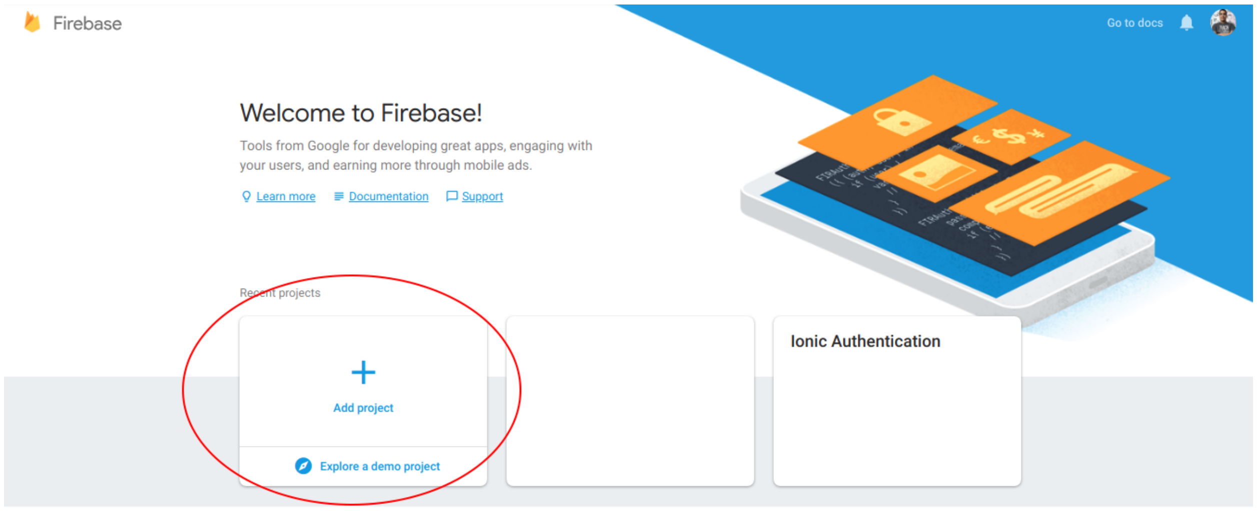


Figure 4.1: Firebase Dashboard

Click the create new app button, fill in the form and then you'll be taken to your app's dashboard.

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

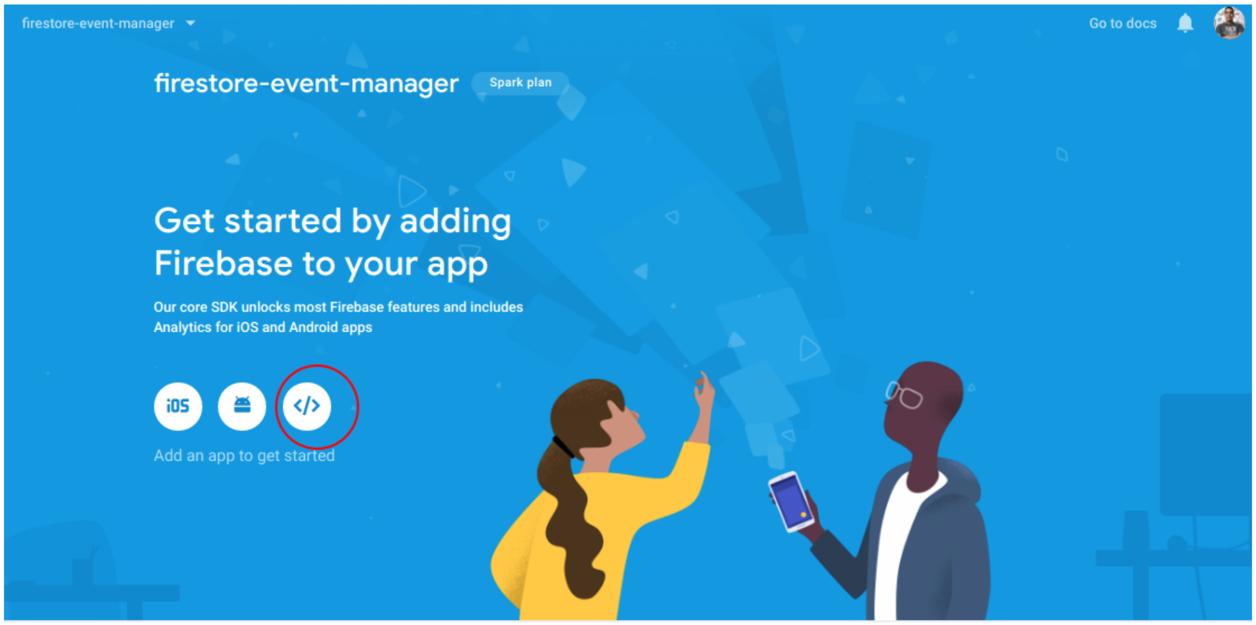
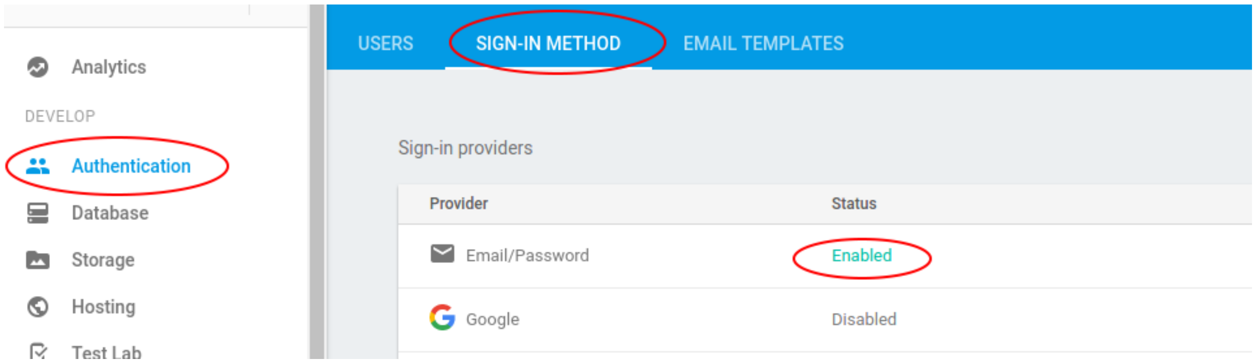

firestore-event-manager
Go to docs
firestore-event-manager
Spark plan
Get started by adding Firebase to your app
Our core SDK unlocks most Firebase features and includes Analytics for iOS and Android apps
iOS
Add an app to get started

Figure 4.2: 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**



Provider	Status
Email/Password	Enabled
Google	Disabled

Figure 4.3: Auth Methods in the console

Also, let's create a new database, so go into **Database** create a new Firestore database, and start it in test mode.

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';

initializeApp() {
  firebase.initializeApp(firebaseConfig);
  ...
}
```

Preventing anonymous users from accessing pages

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

To explain the concept of an authentication guard, we need to first get a bit of context about how the navigation works on Ionic Framework.

Ionic uses Angular Router for navigation, which moves the user through the application using URLs, if you go ahead and open the `src/app` folder, you'll find the file `app-routing.module.ts`, open it and you'll see the CLI created a route for every page you generated.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', loadChildren: './home/home.module#HomePageModule' },
  {
    path: 'event-create',
    loadChildren:
      './pages/event-create/event-create.module#EventCreatePageModule'
  },
  ...
  ...
  ...
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Those routes use lazy loading, meaning that when the app starts it's not going to load them all, instead, it's going to load them on a "need-this-now" case.

That's why you don't see the URLs pointing to a specific component, but instead to a module.

The URLs work like in a regular website, if you go to `yourdomain.com/home` it will take you to the home page.

Before we move to protecting the URLs we need to fix the detail URL first, go ahead and find the detail URL inside the file:

```
{
  path: 'event-detail',
  loadChildren:
    './pages/event-detail/event-detail.module#EventDetailPageModule',
}
```

To go to a detail page, we want to pass the ID of that object to the URL so that we can know which detail we need to fetch from the database, it should look like this:

```
{
  path: 'event-detail/:id',
  loadChildren:
    './pages/event-detail/event-detail.module#EventDetailPageModule',
}
```

Authentication Guard

Now that we have our URLs ready, we want to create an authentication guard, the job of this guard is to fetch for the user's authentication state and return `true/false` depending on whether or not there's a logged in user.

If it returns `true` the user will be able to navigate to that page, if it returns `false` we should redirect the user somewhere else.

Open your terminal and use the CLI to create the guard:

```
ionic generate guard services/user/auth
```

Since the guard is a service related to authentication we're creating it in the `/user` folder. If you open it you'll find something like this:

```
import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
} from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor() {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | Observable<boolean> | Promise<boolean> {
    return true;
  }
}
```

We're going to use the `onAuthStateChanged()` method, it connects to Firebase Authentication and listens to changes in the user state to respond to them. 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.

```
import { Injectable } from '@angular/core';
import {
  CanActivate,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  Router,
} from '@angular/router';
import { Observable } from 'rxjs';
import * as firebase from 'firebase/app';
import 'firebase/auth';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  constructor(private router: Router) {}
  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | Observable<boolean> | Promise<boolean> {
    return new Promise((resolve, reject) => {
      firebase.auth().onAuthStateChanged((user: firebase.User) => {
        if (user) {
          resolve(true);
        } else {
          console.log('User is not logged in');
          this.router.navigate(['/login']);
          resolve(false);
        }
      });
    });
  }
}
```

We're importing the core of Firebase functionality and then adding the auth functionality to the namespace:

```
import * as firebase from 'firebase/app';
import 'firebase/auth';
```

After that, we're using the `onAuthStateChanged()` function to see if there's a user, if there is, we resolve the promise with `true`, if there isn't, we send back `false` and use the router to redirect the user to the login page.

Now that our Guard is created, we can call it from inside the `app-routing.module.ts` file:

```
import { AuthGuard } from './services/user/auth.guard';

{
  path: 'home',
  loadChildren: './pages/home/home.module#HomePageModule',
  canActivate: [AuthGuard],
},
{
  path: 'event-create',
  loadChildren:
    './pages/event-create/event-create.module#EventCreatePageModule',
  canActivate: [AuthGuard],
},
{
  path: 'event-detail/:id',
  loadChildren:
    './pages/event-detail/event-detail.module#EventDetailPageModule',
  canActivate: [AuthGuard],
}
```

All you need to do is add the `canActivate` property with the value of the `AuthGuard` (*or any other guard you create*) to the routes you want protected.

Now that our routing is complete, we can move to the next section and build the functionality we need in the authentication service.

Building the Authentication Service

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 `AuthService` to handle all the authentication related interactions between our app and Firebase.

Open the file `auth.service.ts`, it should look like this:

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

@Injectable({
  providedIn: 'root'
})
export class AuthService {
  constructor() {}
}
```

We are going to start building on top of it, the first thing we will do is to import Firebase, we're going to need the core functionality, auth, and the database added to the namespace:

```
import * as firebase from 'firebase/app';
import 'firebase/auth';
import 'firebase/firestore';
```

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<firebase.auth.UserCredential> {...}
```

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

```
loginUser(
  email: string,
  password: string
): Promise<firebase.auth.UserCredential> {
  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 a UserCredential object 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:** Firestore and authentication are not "connected", like that, creating a user does not store their information inside the database, it saves it in the authentication module of our app, so we need to copy that data inside the firestore database manually.

```
signupUser(email: string, password: string): Promise<any> {
  return firebase
    .auth()
    .createUserWithEmailAndPassword(email, password)
    .then((newUserCredential: firebase.auth.UserCredential) => {
      firebase
        .firestore()
        .doc(`/userProfile/${newUserCredential.user.uid}`)
        .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:

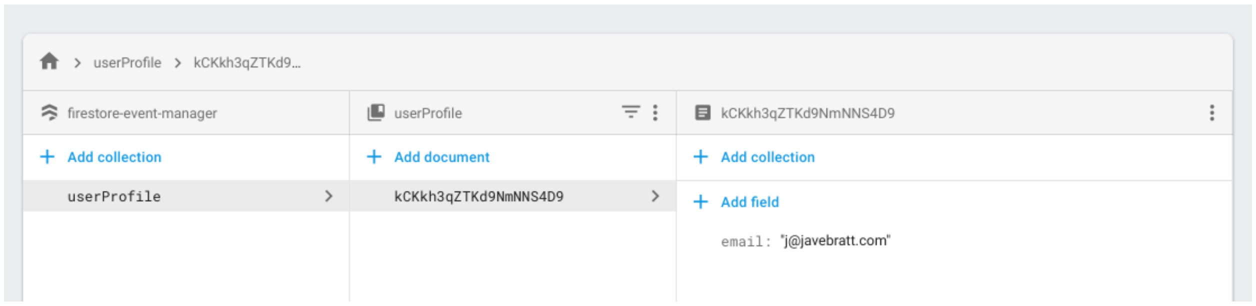
```
.then((newUserCredential: firebase.auth.UserCredential) => {
  firebase
    .firestore()
    .doc(`/userProfile/${newUserCredential.user.uid}`)
    .set({ email });
})
```

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

We're creating new collection called `userProfile`, and the UID identifies the user's document.

Also, we're adding a property called email, filling it with the new user's email address.

> **SIDE-NOTE:** Notice that I'm using template strings, inside `.doc()`, if you don't know what those are, it's a property of ES6 and TypeScript, notice that I'm not using double



email: "j@javebratt.com"

Figure 4.4: User profile node inside Firebase database

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 though 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*).

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 AuthService that 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 `login.page.html` and create a login form inside the ion-content tags to capture email and password:

```
<form [formGroup]="loginForm">
  <ion-item>
    <ion-label position="stacked">Email</ion-label>
    <ion-input
      formControlName="email"
      type="email"
      placeholder="Your email address"
      [class.invalid]="!loginForm.controls['email'].valid &&
      loginForm.controls['email'].touched"
    >
    </ion-input>
  </ion-item>
  <ion-item
    class="error-message"
    *ngIf="!loginForm.controls['email'].valid &&
       loginForm.controls['email'].touched"
  >
    <ion-label>Please enter a valid email address.</ion-label>
  </ion-item>

  <ion-item>
    <ion-label position="stacked">Password</ion-label>
    <ion-input
      formControlName="password"
      type="password"
      placeholder="Your password"
      [class.invalid]="!loginForm.controls['password'].valid&&
         loginForm.controls['password'].touched"
    >
    </ion-input>
  </ion-item>
  <ion-item
    class="error-message"
    *ngIf="!loginForm.controls['password'].valid
      && loginForm.controls['password'].touched"
  >
    <ion-label>Your password needs more than 6 characters.</ion-label>
  </ion-item>
  <ion-button
    (click)="loginUser(loginForm)"
    expand="block"
    [disabled]="!loginForm.valid"
  >
    Log In
  </ion-button>
</form>

<ion-button expand="block" fill="clear" routerLink="/signup">
  Create a new account
</ion-button>

<ion-button expand="block" fill="clear" routerLink="/reset-password">
  I forgot my password :(
</ion-button>
```

That is an HTML form using Ionic components and Angular's form module, 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.

There are two things I want to explain here:

- `routerLink="/signup"` is like using an `<a href=""></a>` tag with Angular router. It will evaluate the URL and navigate the user there.
- You need to open the `login.module.ts` file and add `ReactiveFormsModule` to the `imports` array.

NOTE: You'll also need to add `ReactiveFormsModule` to the import array of any module where you're using reactive forms, for this app it's mainly the authentication forms.

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.page.scss` and add the styles:

```
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;
  ion-label {
    margin: 2px 0;
    font-size: 60%;
    color: #ff6153;
  }
  .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.page.ts` file, you should have something similar to this:

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

@Component({
  selector: 'app-login',
  templateUrl: './login.page.html',
  styleUrls: ['./login.page.scss']
})
export class LoginPage implements OnInit {
  constructor() {}

  ngOnInit() {}
}
```

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

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, Validators, FormBuilder } from '@angular/forms';
import { LoadingController, AlertController } from '@ionic/angular';
import { AuthService } from '../../services/user/auth.service';
import { Router } from '@angular/router';
```

This is the breakdown of what we are importing:

- `LoadingController` and `AlertController` because we are going to be using an alert pop up and a loading component inside our page.
- `FormGroup FormBuilder Validators` are used to get form validation going on with angular.

- `AuthService` is the authentication service we created, we will be using it to call the login function.
- `Router` will handle navigation on this page, we want to send our user to the home page after logging in.

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

```
constructor(
  public loadingCtrl: LoadingController,
  public alertCtrl: AlertController,
  private authService: AuthService,
  private router: Router,
  private 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: HTMLIonLoadingElement;
constructor(...) {...}
```

Inside the constructor, we need to initialize our form:

```
constructor(
  public loadingCtrl: LoadingController,
  public alertCtrl: AlertController,
  private authService: AuthService,
  private router: Router,
  private formBuilder: FormBuilder
) {
  this.loginForm = this.formBuilder.group({
    email: ['',
      Validators.compose([Validators.required, Validators.email])],
    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 login function:

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

```
async loginUser(loginForm: FormGroup): Promise<void> {
  if (!loginForm.valid) {
    console.log('Form is not valid yet, current value:', loginForm.value);
  } else {
    this.loading = await this.loadingCtrl.create();
    await this.loading.present();

    const email = loginForm.value.email;
    const password = loginForm.value.password;

    this.authService.loginUser(email, password).then(
      () => {
        this.loading.dismiss().then(() => {
          this.router.navigateByUrl('home');
        });
      },
      error => {
        this.loading.dismiss().then(async () => {
          const alert = await this.alertCtrl.create({
            message: error.message,
            buttons: [{ text: 'Ok', role: 'cancel' }],
          });
          await alert.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.page.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>
  <form [formGroup]="resetPasswordForm">
    <ion-item>
      <ion-label position="stacked">Email</ion-label>
      <ion-input
        formControlName="email"
        type="email"
        placeholder="Your email address"
        [class.invalid]="!resetPasswordForm.controls['email'].valid &&
        resetPasswordForm.controls['email'].touched"
      >
      </ion-input>
    </ion-item>
    <ion-item
      class="error-message"
      *ngIf="!resetPasswordForm.controls['email'].valid &&
      resetPasswordForm.controls['email'].touched"
    >
      <ion-label>Please enter a valid email.</ion-label>
    </ion-item>

    <ion-button
      expand="block"
      (click)="resetPassword(resetPasswordForm)"
      [disabled]="!resetPasswordForm.valid"
    >
      Reset your Password
    </ion-button>
  </form>
</ion-content>
```

REMEMBER to add the `ReactiveFormsModule` to the page module.

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

```
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;
  ion-label {
    margin: 2px 0;
    font-size: 60%;
    color: #ff6153;
  }
  .item-inner {
    border-bottom: 0 !important;
  }
}
```

And now it's time to code the functionality, open `reset-password.page.ts`, and like we did before, we are going to be adding the imports and injecting our services into the constructor:

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../../services/user/auth.service';
import { AlertController } from '@ionic/angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';

@Component({
  selector: 'app-reset-password',
  templateUrl: './reset-password.page.html',
  styleUrls: ['./reset-password.page.scss'],
})
export class ResetPasswordPage implements OnInit {
  public resetPasswordForm: FormGroup;
  constructor(
    private authService: AuthService,
    private alertCtrl: AlertController,
    private formBuilder: FormBuilder,
    private router: Router
  ) {
    this.resetPasswordForm = this.formBuilder.group({
      email: [
        '',
        Validators.compose([Validators.required, Validators.email]),
      ],
    });
  }

  ngOnInit() {}

}
```

Then we create the password resetting function:

```
resetPassword(resetPasswordForm: FormGroup): void {
  if (!resetPasswordForm.valid) {
    console.log(
      'Form is not valid yet, current value:', resetPasswordForm.value
    );
  } else {
    const email: string = resetPasswordForm.value.email;
    this.authService.resetPassword(email).then(
      async () => {
        const alert = await this.alertCtrl.create({
          message: 'Check your email for a password reset link',
          buttons: [
            {
              text: 'Ok',
              role: 'cancel',
              handler: () => {
                this.router.navigateByUrl('login');
              },
            },
          ],
        });
        await alert.present();
      },
      async error => {
        const errorAlert = await this.alertCtrl.create({
          message: error.message,
          buttons: [{ text: 'Ok', role: 'cancel' }],
        });
        await errorAlert.present();
      }
    );
  }
}
```

Same as login, it takes the value of the form field, sends it to the `AuthService` and waits for 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.page.html` to look like:

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

```
  <form [formGroup]="signupForm">
    <ion-item>
      <ion-label position="stacked">Email</ion-label>
      <ion-input
        formControlName="email"
        type="email"
        placeholder="Your email address"
        [class.invalid]="!signupForm.controls['email'].valid &&
           signupForm.controls['email'].touched"
      >
      </ion-input>
    </ion-item>
    <ion-item
      class="error-message"
      *ngIf="!signupForm.controls['email'].valid
      && signupForm.controls['email'].touched"
    >
      <ion-label>Please enter a valid email.</ion-label>
    </ion-item>

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

    <ion-button
      expand="block"
      (click)="signupUser(signupForm)"
      [disabled]="!signupForm.valid"
    >
      Create an Account
    </ion-button>
  </form>
</ion-content>
```

REMEMBER to add the `ReactiveFormsModule` to the page module.

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

```
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;
  ion-label {
    margin: 2px 0;
    font-size: 60%;
    color: #ff6153;
  }
  .item-inner {
    border-bottom: 0 !important;
  }
}
```

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

```
import { Component, OnInit } from '@angular/core';
import { AuthService } from '../../services/user/auth.service';
import { LoadingController, AlertController } from '@ionic/angular';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Router } from '@angular/router';

@Component({
  selector: 'app-signup',
  templateUrl: './signup.page.html',
  styleUrls: ['./signup.page.scss'],
})
export class SignupPage implements OnInit {
  public signupForm: FormGroup;
  public loading: any;
  constructor(
    private authService: AuthService,
    private loadingCtrl: LoadingController,
    private alertCtrl: AlertController,
    private formBuilder: FormBuilder,
    private router: Router
  ) {
    this.signupForm = this.formBuilder.group({
      email: [
        '',
        Validators.compose([Validators.required, Validators.email]),
      ],
      password: [
        '',
        Validators.compose([Validators.minLength(6), Validators.required]),
      ],
    });
  }

  ngOnInit() {}

}
```

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

```
async signupUser(signupForm: FormGroup): Promise<void> {
  if (!signupForm.valid) {
    console.log(
      'Need to complete the form, current value: ', signupForm.value
    );
  } else {
    const email: string = signupForm.value.email;
    const password: string = signupForm.value.password;

    this.authService.signupUser(email, password).then(
      () => {
        this.loading.dismiss().then(() => {
          this.router.navigateByUrl('home');
        });
      },
      error => {
        this.loading.dismiss().then(async () => {
          const alert = await this.alertCtrl.create({
            message: error.message,
            buttons: [{ text: 'Ok', role: 'cancel' }],
          });
          await alert.present();
        });
      }
    );
    this.loading = await this.loadingCtrl.create();
    await 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.

But Jorge, there's no back button

You might have noticed that when you navigate from the login page to either the signup page or the reset password page there's no back button to return to login.

That's because Ionic 4 doesn't include the back button by default, you have to drop it into the HTML in the pages you want it, so, if you want to add it to the signup page, you have to add it to the HTML like this:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/login"></ion-back-button>
    </ion-buttons>

    <ion-title>signup</ion-title>
  </ion-toolbar>
</ion-header>
```

Adding the component does the trick, Ionic handles all the logic in the background. Notice how we added the `defaultHref="/login"` that's to make sure that even if the user reloads that page the back button shows up and redirects the user to the login screen.

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 5

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 firestore database.

Chapter 6

Event Manager – CRUD 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 firestore 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 service.

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.page.html` and create a button in the header that navigates to the profile page:

```
<ion-header>
  <ion-toolbar>
    <ion-title></ion-title>
    <ion-buttons slot="end">
      <ion-button routerLink="/profile">
        <ion-icon slot="icon-only" name="person"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</ion-header>
```

Creating the User Profile service

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

This service 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 firestore database and the auth data
- Changing the user's password.

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

```
import * as firebase from 'firebase/app';
import 'firebase/auth';
import 'firebase/firestore';
```

We are going to create and initialize two variables:

```
public userProfile: firebase.firestore.DocumentReference;
public currentUser: firebase.User;

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

We are going to use `userProfile` as a document 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 service 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.firestore.DocumentReference {
  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(() => {
      this.currentUser.updateEmail(newEmail).then(() => {
        this.userProfile.update({ email: newEmail });
      });
    })
    .catch(error => {
      console.error(error);
    });
}
```

Here's what's going on:

- We are using `firebase.auth.EmailAuthservice.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 firestore 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(() => {
      this.currentUser.updatePassword(newPassword).then(() => {
        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 service 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-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/home"></ion-back-button>
    </ion-buttons>
    <ion-title>Profile Page</ion-title>
    <ion-buttons slot="end">
      <ion-button (click)="logOut()">
        <ion-icon slot="icon-only" name="log-out"></ion-icon>
      </ion-button>
    </ion-buttons>
  </ion-toolbar>
</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-list>
</ion-content>
```

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

```
<ion-item (click)="updateName()">
  <ion-label>
    <ion-grid>
      <ion-row>
        <ion-col class="text-left" size="5"> Name </ion-col>
        <ion-col
          class="text-center"
          size="7"
          *ngIf="userProfile?.firstName || userProfile?.lastName"
        >
          {{userProfile?.firstName}} {{userProfile?.lastName}}
        </ion-col>
        <ion-col
          size="7"
          class="placeholder-profile text-center"
          *ngIf="!userProfile?.firstName"
        >
          <span> Tap here to edit. </span>
        </ion-col>
      </ion-row>
    </ion-grid>
  </ion-label>
</ion-item>
```

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

> **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-label>
    <ion-grid>
      <ion-row>
        <ion-col class="text-left" size="5"> Email </ion-col>
        <ion-col class="text-center" size="7" *ngIf="userProfile?.email">
          {{userProfile?.email}}
        </ion-col>
        <ion-col
          size="7"
          class="placeholder-profile text-center"
          *ngIf="!userProfile?.email"
        >
          <span> Tap here to edit. </span>
        </ion-col>
      </ion-row>
    </ion-grid>
  </ion-label>
</ion-item>

<ion-item (click)="updatePassword()">
  <ion-label>
    <ion-grid>
      <ion-row>
        <ion-col class="text-left" size="5"> Password </ion-col>
        <ion-col class="text-center" size="7" class="placeholder-profile">
          <span> Tap here to edit. </span>
        </ion-col>
      </ion-row>
    </ion-grid>
  </ion-label>
</ion-item>
```

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!*)

```
ion-list-header {
  background-color: #ececec;
}

.text-center {
  text-align: center;
}

.text-left {
  text-align: left;
}

.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, OnInit } from '@angular/core';
import { AlertController } from '@ionic/angular';
import { AuthService } from '../../services/user/auth.service';
import { ProfileService } from '../../services/user/profile.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-profile',
  templateUrl: './profile.page.html',
  styleUrls: ['./profile.page.scss'],
})
export class ProfilePage implements OnInit {
  public userProfile: any;
  public birthDate: Date;
  constructor(
    private alertCtrl: AlertController,
    private authService: AuthService,
    private profileService: ProfileService,
    private router: Router
  ) {}

  ngOnInit() {}
}
```

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 services 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 service and ask for the user's profile, so right after our constructor, go ahead and create the function:

```
ngOnInit() {
  this.profileService
    .getUserProfile()
    .get()
    .then( userProfileSnapshot => {
      this.userProfile = userProfileSnapshot.data();
      this.birthDate = userProfileSnapshot.data().birthDate;
    });
}
```

Let's break down what we did here:

- We are using `ngOnInit()`, this is part of Angular's lifecycle events, it is called after the view rendered.
- We are calling the `getUserProfile()` function from our `ProfileService` 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.

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.authService.logoutUser().then( () => {
    this.router.navigateByUrl('login');
  });
}
```

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:

```
async updateName(): Promise<void> {
  const alert = await this.alertCtrl.create({
    subHeader: 'Your first name & last name',
    inputs: [
      {
        type: 'text',
        name: 'firstName',
        placeholder: 'Your first name',
        value: this.userProfile.firstName,
      },
      {
        type: 'text',
        name: 'lastName',
        placeholder: 'Your last name',
        value: this.userProfile.lastName,
      },
    ],
    buttons: [
      { text: 'Cancel' },
      {
        text: 'Save',
        handler: data => {
          this.profileService.updateName(data.firstName, data.lastName);
        },
      },
    ],
  });
  await 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 `Profileservice`.

For the birthday we have to do a bit more validation, the `(ionChange)` can trigger on page load so we want to make sure it's not `undefined`.

```
updateDOB(birthDate: string): void {
  if (birthDate === undefined) {
    return;
  }
  this.profileService.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:

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

async updatePassword(): Promise<void> {
  const alert = await 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.profileService.updatePassword(
            data.newPassword,
            data.oldPassword
          );
        },
      },
    ],
  });
  await alert.present();
}
```

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

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 Documents in Firestore.

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 7

Event Manager – Working with a List

We have been learning a lot about Firestore 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 Service

As always we are going to be using a service to handle all of our event data, the reason we are using services 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, I just moved this entire application from the Firebase RTDB to Firestore, and I only needed to change the service files and a couple of functions and it works perfectly.

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

The first thing we are going to do here is to import Firestore and create a variable to hold our `eventList` collection so that we can use it in all of our functions.

```
import { Injectable } from '@angular/core';
import * as firebase from 'firebase/app';
import 'firebase/auth';
import 'firebase/firestore';

@Injectable({
  providedIn: 'root',
})
export class EventService {
  public eventListRef: firebase.firestore.CollectionReference;
  constructor() {
    firebase.auth().onAuthStateChanged(user => {
      if (user) {
        this.eventListRef = firebase
          .firestore()
          .collection(`/userProfile/${user.uid}/eventList`);
      }
    });
  }
}
```

Now it is time to start thinking what kind of features we need in our service. 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
): Promise<firebase.firestore.DocumentReference> {
  return this.eventListRef.add({
    name: eventName,
    date: eventDate,
    price: eventPrice * 1,
    cost: eventCost * 1,
    revenue: eventCost * -1,
  });
}
```

A couple of things to note:

- We are using `.add()` on the `eventList` sub-collection because we want firebase to append every new document 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.firestore.CollectionReference {
  return this.eventListRef;
}
```

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

```
getEventDetail(eventId: string): firebase.firestore.DocumentReference {
  return this.eventListRef.doc(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 event create or list pages.

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

```
<ion-content padding>
  <ion-button expand="block" color="primary" routerLink="/event-create">
    Create a new Event
  </ion-button>

  <ion-button expand="block" color="primary" routerLink="/event-list">
    See your events
  </ion-button>
</ion-content>
```

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.page.html` and create a few inputs to save the event's name, date, ticket price and costs:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/home"></ion-back-button>
    </ion-buttons>
    <ion-title>New Event</ion-title>
  </ion-toolbar>
</ion-header>

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

  <ion-item>
    <ion-label position="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
      [(ngModel)]="eventDate"
      displayFormat="D MMM, YY"
      pickerFormat="DD MMM YYYY"
      min="2017"
      max="2020-12-31"
    >
    </ion-datetime>
  </ion-item>

  <ion-button
    expand="block"
    (click)="createEvent(eventName, eventDate, eventPrice, eventCost)"
  >
    Create Event
  </ion-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.page.ts` first we will need to import our event service and the angular router.

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { EventService } from '../../services/event/event.service';

@Component({
  selector: 'app-event-create',
  templateUrl: './event-create.page.html',
  styleUrls: ['./event-create.page.scss'],
})
export class EventCreatePage implements OnInit {
  constructor(private router: Router, private eventService: EventService) {}

  ngOnInit() {}
}
```

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

```
createEvent(
  eventName: string,
  eventDate: string,
  eventPrice: number,
  eventCost: number
): void {
  if (
    eventName === undefined ||
    eventDate === undefined ||
    eventPrice === undefined ||
    eventCost === undefined
  ) {
    return;
  }
  this.eventService
    .createEvent(eventName, eventDate, eventPrice, eventCost)
    .then(() => {
      this.router.navigateByUrl('');
    });
}
```

Nothing too crazy, we are sending the data to our `EventService`, and as soon as we create the event, we are using `this.router.navigateByUrl('');` to go back a page to the `HomePage`.

We use `this.router.navigateByUrl('');` 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 to the `event-list.page.html` file and create a list of your events:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start"> <ion-back-button></ion-back-button>
       </ion-buttons>
    <ion-title>Your Events</ion-title>
  </ion-toolbar>
```

```
</ion-header>

<ion-content padding>
  <ion-list>
    <ion-list-header> <ion-label>Your next events</ion-label>
       </ion-list-header>
    <ion-item
      tappable
      *ngFor="let event of eventList"
      routerLink="/event-detail/{{ event.id }}"
    >
      <ion-label>
        <h2>{{event?.name}}</h2>
        <p>Ticket: <strong>${{event?.price}}</strong></p>
        <p>Date: <strong>{{event?.date}}</strong></p>
      </ion-label>
    </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 send the event id in the `routerLink="/event-detail/{{ event.id }}"` callso we can pull the specific ID from Firebase.

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

```
import { Component, OnInit } from '@angular/core';
import { EventService } from '../../services/event/event.service';

@Component({
  selector: 'app-event-list',
  templateUrl: './event-list.page.html',
  styleUrls: ['./event-list.page.scss'],
})
export class EventListPage implements OnInit {
  public eventList: Array<any>;
  constructor(private eventService: EventService) {}

  ngOnInit() {}
}
```

- We are importing `EventService` to call the service'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.

```
ngOnInit() {
  this.eventService
    .getEventList()
    .get()
    .then(eventListSnapshot => {
      this.eventList = [];
      eventListSnapshot.forEach(snap => {
        this.eventList.push({
          id: snap.id,
          name: snap.data().name,
          price: snap.data().price,
          date: snap.data().date,
        });
        return false;
      });
    });
}
```

We have done this before. We are:

- Calling the `getEventList()` method from our service.
- 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.

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.page.ts`, you're going to import and initialize:

```
import { Component, OnInit } from '@angular/core';
import { EventService } from '../../services/event/event.service';
import { ActivatedRoute } from '@angular/router';

@Component({
  selector: 'app-event-detail',
  templateUrl: './event-detail.page.html',
  styleUrls: ['./event-detail.page.scss'],
})
export class EventDetailPage implements OnInit {
  public currentEvent: any = {};

  constructor(
    private eventService: EventService,
    private route: ActivatedRoute,
  ) {}
  ngOnInit() {}
}
```

We are importing `ActivatedRoute` 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:

```
ngOnInit() {
  const eventId: string = this.route.snapshot.paramMap.get('id');
  this.eventService
    .getEventDetail(eventId)
    .get()
    .then(eventSnapshot => {
      this.currentEvent = eventSnapshot.data();
      this.currentEvent.id = eventSnapshot.id;
    });
}
```

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

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start"> <ion-back-button></ion-back-button>
       </ion-buttons>
    <ion-title>{{currentEvent?.name}}</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content padding>
  <ion-card>
    <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 a bit easier, 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

Chapter 8

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 `event.service.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
): Promise<firebase.firestore.DocumentReference> {
  return this.eventListRef
    .doc(eventId)
    .collection('guestList')
    .add({ 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.eventService
    .addGuest(
      guestName,
      this.currentEvent.id,
      this.currentEvent.price,
    )
    .then(() => 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 `guestName` variable we're using:

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

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

The first thing we will add is a header to show our revenue updating in real time (*Yes, we will get to the JS for that part soon*) go ahead and add a header to the first card:

```
<ion-card>
  <ion-card-header>
    Event's Revenue:
    <span
      [class.profitable]="currentEvent?.revenue > 0"
      [class.no-profit]="currentEvent?.revenue < 0 "
    >
      {{currentEvent?.revenue | currency}}
    </span>
  </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>
```

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.page.scss`

```
.add-guest-form {
  ion-card-header {
    text-align: center;
  }
  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:

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

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

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

Creating a Firestore 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 Firestore 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 then Firestore retries your write operation.

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

```
addGuest(
  guestName: string,
  eventId: string,
  eventPrice: number
): Promise<firebase.firestore.DocumentReference> {
  return this.eventListRef
    .doc(eventId)
    .collection('guestList')
    .add({ guestName });
}
```

Now it should look like this:

```
addGuest(guestName: string, eventId: string, eventPrice: number):
    Promise<void> {
  return this.eventListRef
    .doc(eventId)
    .collection('guestList')
    .add({ guestName })
    .then((newGuest) => {
      return firebase.firestore().runTransaction(transaction => {
        return transaction.get(this.eventListRef.doc(eventId)).then(eventDoc
            => {
          const newRevenue = eventDoc.data().revenue + eventPrice;
          transaction.update(this.eventListRef.doc(eventId), { revenue:
              newRevenue });
        });
      });
    });
}
```

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 9

Event Manager – 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 Capacitor 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 everything installed let's jump into the code.

Taking pictures

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

```html
<ion-card class="add-guest-form">
  <ion-card-header> Add Guests </ion-card-header>
  <ion-card-content>
    <ion-row>
      <ion-col size="8">
        <ion-item>
          <ion-label position="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 size="4">
        <ion-button (click)="takePicture()">
          <ion-icon slot="icon-only" name="camera"></ion-icon>
        </ion-button>
      </ion-col>
    </ion-row>
    <span *ngIf="guestPicture">Picture taken!</span>

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

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.page.ts` file*)

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

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

```
import { Plugins, CameraResultType } from '@capacitor/core';
const { Camera } = Plugins;
```

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

Now:

```
addGuest(guestName: string): void {
  this.eventService
    .addGuest(
      guestName,
      this.currentEvent.id,
      this.currentEvent.price,
      this.guestPicture
    )
    .then(() => {
      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:

```
async takePicture(): Promise<void> {
  try {
    const profilePicture = await Camera.getPhoto({
      quality: 90,
      allowEditing: false,
      resultType: CameraResultType.Base64,
    });
    this.guestPicture = profilePicture.base64String;
  } catch (error) {
    console.error(error);
  }
}
```

There we are calling the Camera API from Capacitor and giving it a few options, most of them are obvious by their names, the most important one is `resultType` because it's the one that will give you the format of the image, either a base64 string or the native path to the actual file.

We're using the base64 string because Firebase Cloud Storage has a `.putString()` method that takes a base64 string and uploads the picture from it.

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 `event.service.ts` and find the `addGuest()` function:

```
addGuest(guestName: string, eventId: string, eventPrice: number):
    Promise<void> {
  return this.eventListRef
    .doc(eventId)
    .collection('guestList')
    .add({ guestName })
    .then((newGuest) => {
      return firebase.firestore().runTransaction(transaction => {
        return transaction.get(this.eventListRef.doc(eventId)).then(eventDoc
            => {
          const newRevenue = eventDoc.data().revenue + eventPrice;
          transaction.update(this.eventListRef.doc(eventId), { revenue:
              newRevenue });
        });
      });
    });
}
```

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

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

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.

It should be inside the `.then()` right after we run the Firebase transaction.

```
if (guestPicture != null) {
  const storageRef = firebase
    .storage()
    .ref(`/guestProfile/${newGuest.id}/profilePicture.png`);

  return storageRef
    .putString(guestPicture, 'base64', { contentType: 'image/png' })
    .then(() => {
      return storageRef.getDownloadURL().then(downloadURL => {
        return this.eventListRef
          .doc(eventId)
          .collection('guestList')
          .doc(newGuest.id)
          .update({ profilePicture: 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.

Remember that you'd need to import the Storage module:

```
import * as firebase from 'firebase/app';
import 'firebase/auth';
import 'firebase/firestore';
import 'firebase/storage';
```

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.

To test this on our phones, we must run it with capacitor, so we're going to do two things right now.

First, open up your terminal and type

```
ionic build
```

It will generate a new build for your app, then type

```
ionic cap sync
```

And now you're ready to test your app, type in the terminal

```
ionic cap open
```

It will prompt you for the OS you want to target, click it and it should either open XCode or Android Studio depending on your choice.

If you're running Linux like myself, you're probably going to need to manually open Android Studio and import the project from the options there.

Once Android Studio is open and the project is loaded you can click the green "*play*" button to run the app on your phone.

If you get a permission denied error from Firebase Storage, make sure you open the Storage tab inside your Firebase Console, click the "*Get Started*" button and set up the Storage portion of your app.

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 10

Event Manager – 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.

Firestore Security

With the **Cloud Firestore Security Rules**, we can focus on building a great user experience, without having to manage infrastructure or write server-side authentication and authorization code.

The idea is to authenticate users through Firebase Authentication and set up rules to determine who has access to data stored in Cloud Firestore.

You can find your security rules in the **Rules tab** in the Cloud Firestore section of the Firebase Console.

To start securing our database we need to understand how the security rules work, let's take a look at the default ones that come when you create the app.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if false;
    }
  }
}
```

The security rules work matching documents in the database, they have two permissions, `read` and `write` which are both false by default, meaning, no one has access to the database.

To start working with them, we tell them to allow all read/write operations, since we're going to be in development mode:

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write;
    }
  }
}
```

The =** symbol is a cascade operator, rules don't cascade by default.

So if you set up a read/write rule for the document `users/{userId}` but don't set up read/write rules for `users/{userId}/tasks/{taskId}` no one will have access to the `taskId` documents.

When you use the =** operator, you're telling Firestore rules that if the user matches the condition to read that document, they should be able to read all the sub-collections and documents below that tree.

The brackets mean we're using a wild-card, for example, if I have a collection called `users` that has the documents for each user's profile, I'd only want the profile owner to be able to have read/write access.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth.uid == userId;
    }
  }
}
```

Notice how we're getting the user's `uid` from the `request` object. When working with Firestore rules, we have two available objects.

The `request` object has information about the request made, such as the authenticated user: `request.auth`, and the time the request was made: `request.time`.

The `resource` object is THE Firestore document we're accessing. For example, let's say we have public and private profiles, each profile has a flag called `public` and it's either set to `true` or `false`.

The `resource` object gives us access to that flag:

```
service cloud.firestore {
  match /databases/{database}/documents {

    match /myCollection/myDocument {
      allow read: if resource.data.public == true;
    }
  }
}
```

In that case, people will only be able to read profiles marked as `public`.

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 11

Thank you

First thing I want to do is to say thanks, you've trusted me enough to invest your time in my book :-)

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.

When you're ready to jump into more advanced topics you might be interested in checking out my complete book "*Building Firebase Powered Ionic Apps*".

It covers topics like:

- AngularFire2
- Observables.
- Role-based Authentication.
- Cloud Functions.
- Push Notifications.
- … and more.

You can check it out here: https://javebratt.com/ionic-firebase-book/.

Also, if you ever get stuck, remember that you can always send me an email to j@javebratt.com, or if you only want to chat. I'm always happy to hear from other Ionic devs :-)

Chapter 12

CHANGELOG

4.0.0 Breaking Changes with Route Guards

- Migrates all the route guards to `AngularFireAuthGuard`.
- Fixes types from new update.
- Fixes error where the `inShoppingList` param was alwas loading as false.
- Adds the 'desc' parameter to the `clientWeightHistory()` query.
- Adds link to the event-manager source code.

3.0.0 Breaking changes

- Updates to Firebase v6.0.2
- In the event manager app the function `reauthenticateAndRetrieveDataWithCredential` has been deprecated and replaced with `reauthenticateWithCredential` instead.
- Upgrades Capacitor to latest version, changes in the Camera API, instead of using `this.guestPicture = profilePicture.base64Data.slice(23);` to get the picture, we now can get the base64String directly with `this.guestPicture = profilePicture.base64String;`. (*Side-note: Really proud of that, it's a PR I made to the Capacitor repository*)
- Fixes bug in the `event-service.addGuest()` function where the function wasn't returning the `newGuest` after adding it.
- Removes the links to the source code repositories and instead adds the code as a downloadable with the book.
- From the `firestore-shopping` app it moves the `shopping-list-add` and the `inventory-add` routes as children inside the `tabs.routing` module.
- Fixes bug from Cloud Functions where it was incorrectly refering to the RTDB instead of Firestore.
- Relaces the FCM Cordova plugin with the Capacitor Push Notifications API.

2.0.3 Updates to Ionic 4.1.0

There are no changes, you can update the `@ionic/angular` package and it won't break anything.

2.0.2 Typo fix

2.0.1 Updates to Ionic 4.0.0 and Firebase 5.8.1

There aren't breaking changes in this update, I just took it as an opportunity to fix some bugs y'all had reported :)

2.0.0 BREAKING CHANGES

In this book update we're updating the apps to `@ionic/angular` 4.0.0-rc.0 which is the final step before V4 final.

Since it would take a lot of space to add all the changes here what I did was to create a file called `CHANGELOG.md` inside the repository of every app where you can see in more detail what changed.

1.0.2 Small Fies

- Removes trailing slash (/) at the end of a firestore reference.
- Corrects spelling mistake "Capera" to "Camera" :-P
- Ionic 4.0.0-beta.13 changed the output of the `<ion-datetime>` to be an ISO string so refactored the code to work.

1.0.1 Fixes bugs and typos

- Adds a note at the beginning of the book to remove reference of `@ionic-native` from the `app.component.ts` file.
- Fixes `ngOnInit()` function in the event detail page, it had some bad code from the previous firestore database example.
- Fixes wrong import in the event manager example, I was importing the `credentials.ts` file from `import { firebaseConfig } from './config/credentials';` when it should have been `import { firebaseConfig } from './credentials';`.

1.0.0

Official Launch