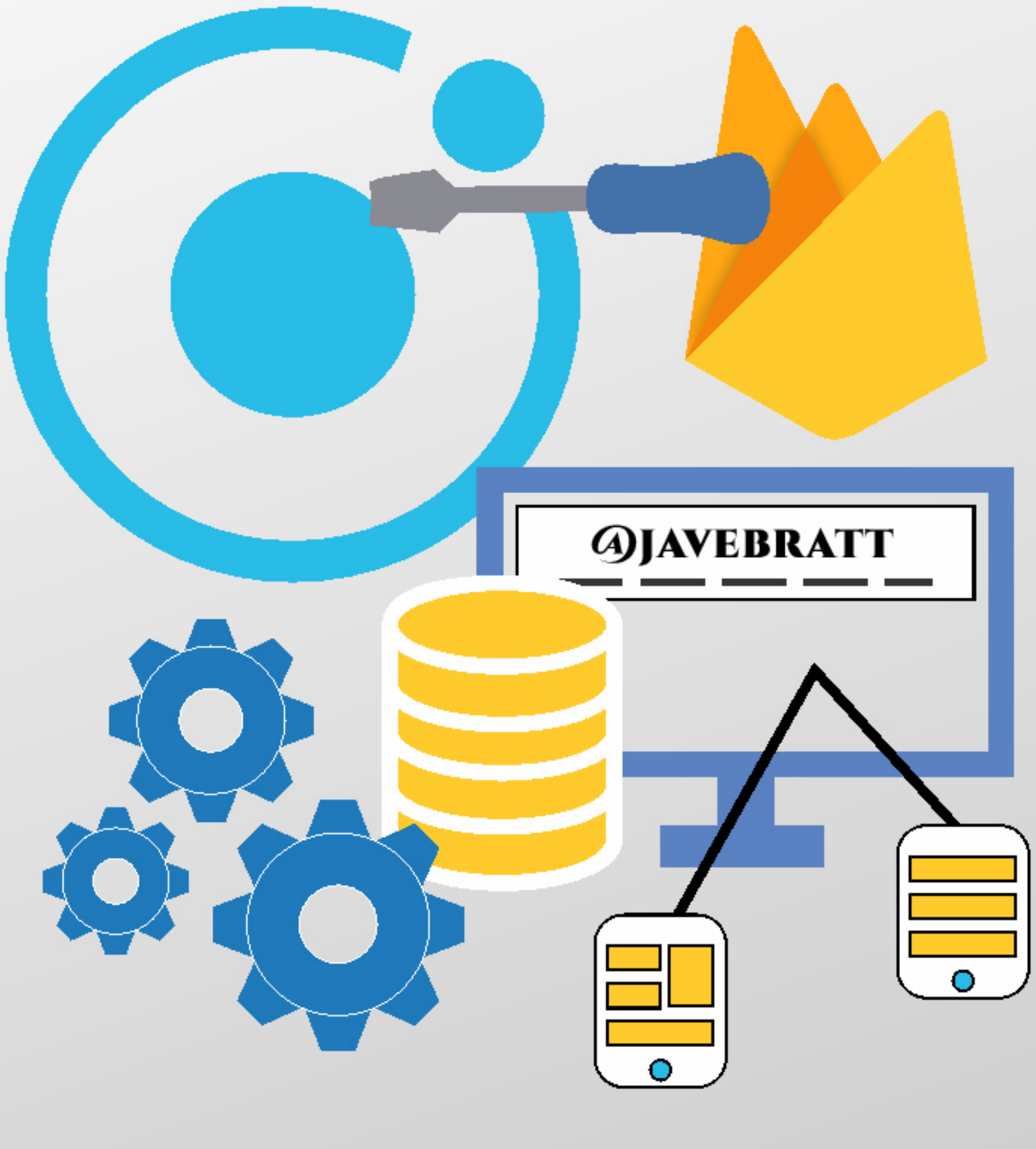
Building Firebase Powered Ionic Apps



Jorge Vergara

Contents

To my wife, Evelyn, and my kids, Emmanuel and Isabella.

You made this happen.

Build your First Firebase powered Ionic app using `@angular/fire`

First of all, thank you!

There are several resources out there, so it means a lot to me you're trusting me to guide you through this journey.

The goal of this book is for you to start scaling your Ionic applications with the help of Firebase.

What you'll learn

Throughout this book, you'll learn how to use Firebase as the backend for your Ionic applications.

We'll work together and use Firebase Authentication to create and manage users and the Firestore database to Create, Read, Update, and Delete items from it.

Who this book is for

This book is for Ionic developers who want to use Firebase as their backend. The book isn't an Ionic 101 course, but I'll do my best to explain every bit of Ionic code we'll use during the book.

The Tools we'll use

For this book, we'll use Ionic with Angular and Firebase with AngularFire.

The goal is to use the latest versions available:

- `@ionic/angular`: version 6

- `firebase` web SDK: version 9
- `@angular/fire`: version 7

We'll use AngularFire for a particular reason, they wrap zones and handle a bit of the behind the scenes for you, so we have fewer things to worry about.

What we'll build

We'll build a party manager application. The idea is that you can first handle users, as in, sign up, log in, and password reset.

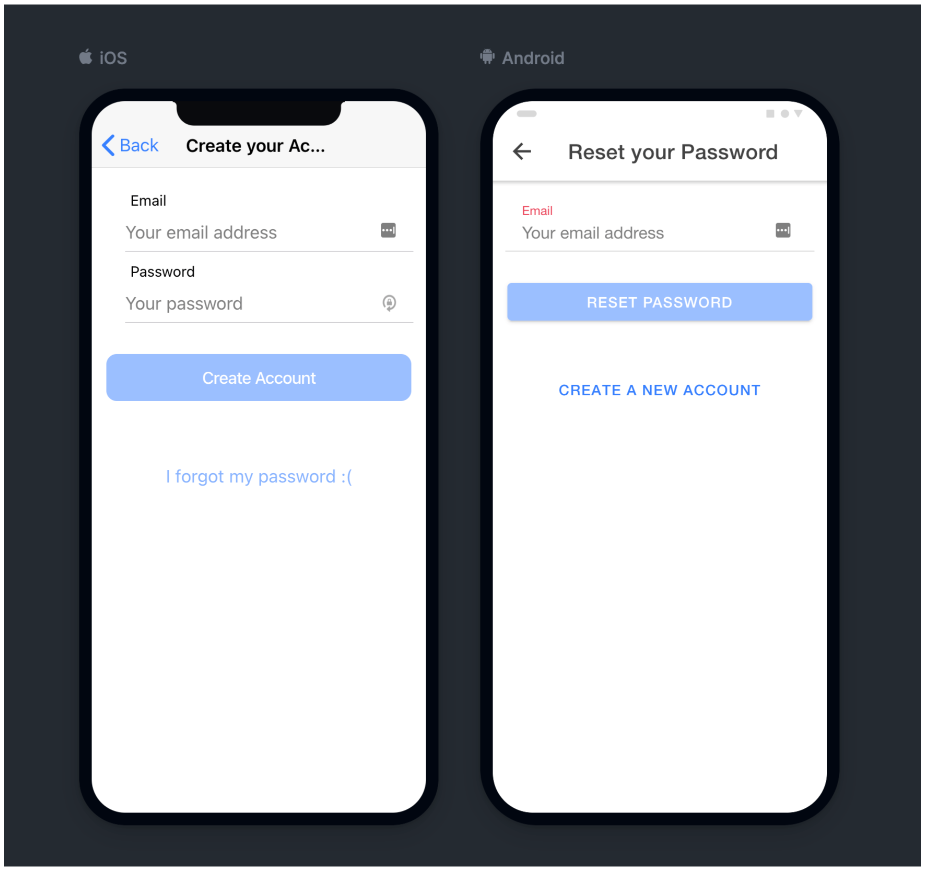
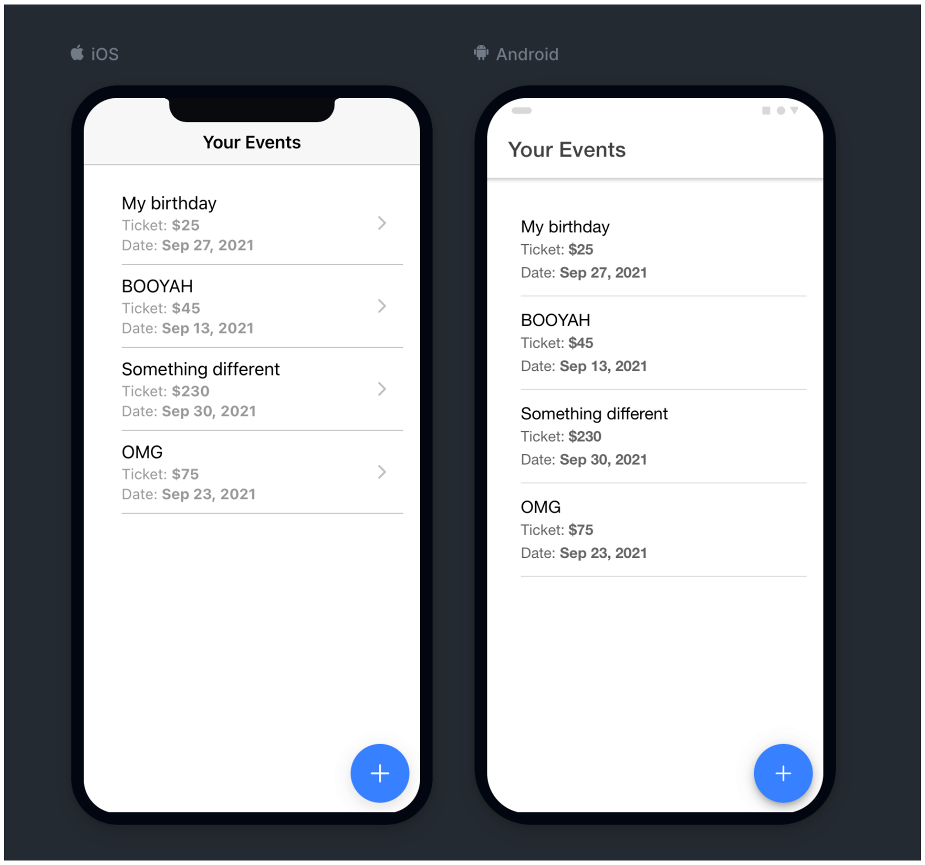


Figure 1: Authentication Pages

And then, we'll move to things like creating events, updating them, deleting them, etc.



Your Events

My birthday
Ticket: **$25**
Date: **Sep 27, 2021**

BOOYAH
Ticket: **$45**
Date: **Sep 13, 2021**

Something different
Ticket: **$230**
Date: **Sep 30, 2021**

OMG
Ticket: **$75**
Date: **Sep 23, 2021**

Your Events

My birthday
Ticket: **$25**
Date: **Sep 27, 2021**

BOOYAH
Ticket: **$45**
Date: **Sep 13, 2021**

Something different
Ticket: **$230**
Date: **Sep 30, 2021**

OMG
Ticket: **$75**
Date: **Sep 23, 2021**

Figure 2: List of events

Next Step

If at any point you have any questions, don't hesitate to ask, you can reach me at jorge@jsmobiledev.com.

Once you're ready, you can move to the next chapter, where we'll set up our development environment and install everything we need to get the application working.

Create our first app

The first thing we'll do is making sure we've got everything ready to start building Ionic and Firebase applications, and for that, we'll install a few things.

Let's ensure we have node and npm installed on our computer. We can do that directly from the nodejs website. Or we can install nvm (node version manager) to handle different versions of node and npm depending on the needs of our projects.

Installing the dependencies

After we have node and npm installed, we want to install the Ionic and the Firebase CLI. For that, we're going to open our terminal and type:

```
npm install -g @ionic/cli firebase-tools
```

To make sure both packages are installed, go to our terminal and type:

```
ionic --version
firebase --version
```

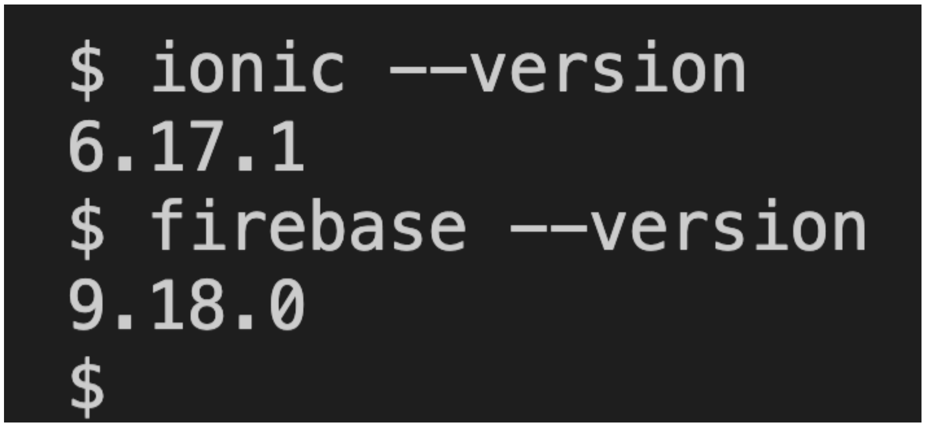
we should see the versions (*Figure 3*) if we see something like `command not found: ionic` or `command not found: firebase` means the package isn't installed.

Create our Application

Now that everything is installed and ready to go, it's time to create our first application. To do that, let's go to the terminal again, navigate to whatever folder we use for our projects, and type:

```
ionic start party-planner
```

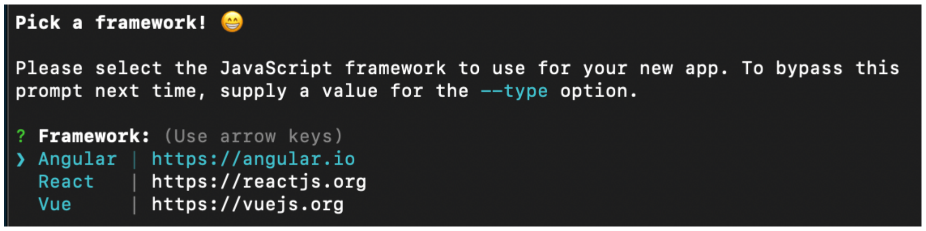
Where `party-planner` is the application's name, it will start a helper that will ask us a few questions.



```
$ ionic --version
6.17.1
$ firebase --version
9.18.0
$
```

Figure 3: Ionic and Firebase package versions

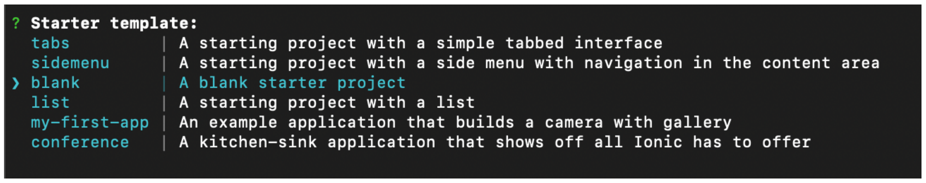
First, it will ask us the UI framework we'll use for the application, Angular, React, or Vue. For this example, we'll pick Angular.



```
Pick a framework! 😁

Please select the JavaScript framework to use for your new app. To bypass this
prompt next time, supply a value for the --type option.

? Framework: (Use arrow keys)
❯ Angular | https://angular.io
  React   | https://reactjs.org
  Vue     | https://vuejs.org
```

Figure 4: Pick your UI framework

Then it will ask us the starter template we want to pick. We'll start from scratch, so select `blank`.



```
? Starter template:
  tabs         | A starting project with a simple tabbed interface
  sidemenu     | A starting project with a side menu with navigation in the content area
❯ blank        | A blank starter project
  list         | A starting project with a list
  my-first-app | An example application that builds a camera with gallery
  conference   | A kitchen-sink application that shows off all Ionic has to offer
```

Figure 5: Pick the starter template

Now we need to give the CLI a few moments to create everything. Once it's done, we can open the application folder in our preferred code editor, I use VS Code, but you can use whatever works best for you.

While we're checking out the app, we'll open the `package.json` file. We'll see all the packages we have installed and their versions look something like this:

```json
{
  "name": "party-planner",
  "version": "0.0.1",
  "author": "Ionic Framework",
  "homepage": "https://ionicframework.com/",
  "scripts": {
    ...
  },
  "private": true,
  "dependencies": {
    "@angular/common": "~12.1.1",
    "@angular/core": "~12.1.1",
    ...
    "@ionic/angular": "^5.5.2",
    "rxjs": "~6.6.0",
    ...
  },
  "devDependencies": {
    ...
    "typescript": "~4.2.4"
  },
  "description": "An Ionic project"
}
```

> Note, when I wrote this book, the current `@ionic/angular` version was 5, so to install version 6, we need to install the BETA, if when you create the app, you get version 6 already, feel free to ignore this note. To Install the BETA, open your terminal and type `npm install @ionic/angular@next`.

After we make sure everything looks ok, we want to install the two main packages we'll use to build our application.

We'll use the Firebase Web SDK with the help of AngularFire. The AngularFire team created an Angular schematic that does a lot of the heavy lifting for us:

- It installs the packages we need.
- Initializes the parts of Firebase we'll use.
- Gets all the needed files.
- Connects to our online Firebase instance to get the credentials.
- Initializes Firebase in our Ionic application.

To use this schematic, open your terminal, inside of the project root folder, and type:

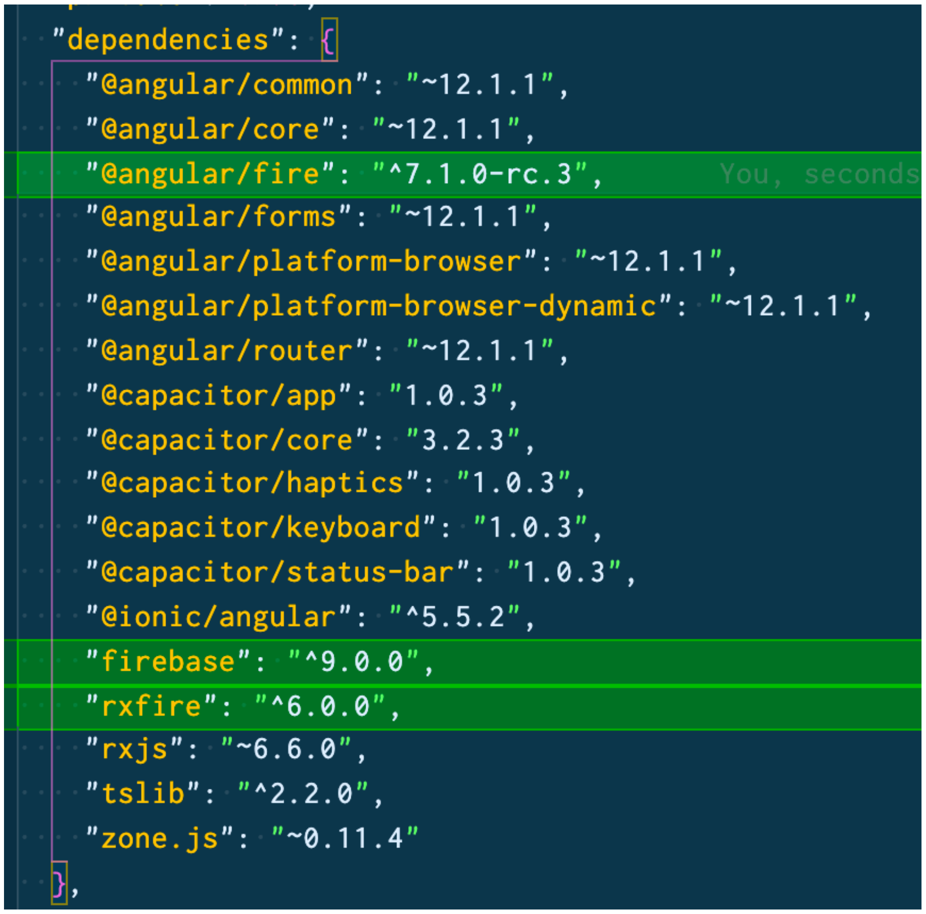
```
ng add @angular/fire
```

Now, let's look at what it did for us.

First, if you check the `package.json` file, you'll note that it installed the packages we need:

```
"dependencies": {
  "@angular/fire": "^7.1.0-rc.3",
  "firebase": "^9.0.0",
  "rxfire": "^6.0.0",
},
```

> NOTE: If by the time you read this book, the version of `@angular/fire` installed is 6.x.x then please, discard the changes and run the schematic with the `@next` flag: `ng add @angular/fire@next`.



```
  "dependencies": {
    "@angular/common": "~12.1.1",
    "@angular/core": "~12.1.1",
    "@angular/fire": "^7.1.0-rc.3",          You, seconds
    "@angular/forms": "~12.1.1",
    "@angular/platform-browser": "~12.1.1",
    "@angular/platform-browser-dynamic": "~12.1.1",
    "@angular/router": "~12.1.1",
    "@capacitor/app": "1.0.3",
    "@capacitor/core": "3.2.3",
    "@capacitor/haptics": "1.0.3",
    "@capacitor/keyboard": "1.0.3",
    "@capacitor/status-bar": "1.0.3",
    "@ionic/angular": "^5.5.2",
    "firebase": "^9.0.0",
    "rxfire": "^6.0.0",
    "rxjs": "~6.6.0",
    "tslib": "^2.2.0",
    "zone.js": "~0.11.4"
  },
```

Figure 6: Installed Dependencies

After installing the files, it will start a prompt in the CLI asking you a few questions to finish the rest of the process.

It will ask you what Firebase packages you want to initialize. You can use the arrow keys on your keyboard to move between them and use the space bar to select them.

For this book, you'll choose Firebase Authentication and Firestore.

Next, it will ask you to log in to see what Firebase projects you have access. Please, log into the CLI with the google account you want to use with Firebase.

Next, it will ask you which app you want to use, select the Firebase project you'll be using.

Once you're done answering the questions, it will generate multiple files for us. Let's double-check that everything is ready to use.

First, go to your `src/environments/environment.ts` file, you'll notice it added a new property to the environment variable called `firebase`, and it will have your firebase credentials:

```
export const environment = {
  firebase: {
    projectId: '',
    appId: '',
    storageBucket: '',
    locationId: '',
    apiKey: '',
    authDomain: '',
    messagingSenderId: '',
  },
  production: false,
};
```

Next, it will go into the `app.module.ts` file and initialize Firebase for you, so let's open that file and review it.

First, it will import what you need:

```
import { initializeApp, provideFirebaseApp } from
    '@angular/fire/app';
import { environment } from '../environments/environment';
import { provideAuth, getAuth } from '@angular/fire/auth';
import { provideFirestore, getFirestore } from
    '@angular/fire/firestore';
```

It is importing the functions it needs to initialize Firebase, Authentication, and Firestore. It is also importing the `environment` variable, since it has our firebase configuration.

Next, it will provide our firebase initializations to the `imports` array of our `NgModule`:

```
@NgModule({
  declarations: [AppComponent],
  entryComponents: [],
  imports: [
    ...,
    provideFirebaseApp(() => initializeApp(environment.firebase)),
    provideAuth(() => getAuth()),
    provideFirestore(() => getFirestore()),
  ],
  providers: [{ provide: RouteReuseStrategy, useClass:
      IonicRouteStrategy }],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

There's a small change we want to make, the Firestore initialization. Let's change it so that we enable offline persistence:

```
provideFirestore(() => {
  const firestore = getFirestore();
  enableIndexedDbPersistence(firestore);
  return firestore;
}),
```

Don't forget to add `enableIndexedDbPersistence` to the list of imports from `@angular/fire/firestore`.

If, by some weird mistake, it didn't initialize the application for you, or it didn't get the credentials, no worries, you can get those credentials manually from the Firebase console itself.

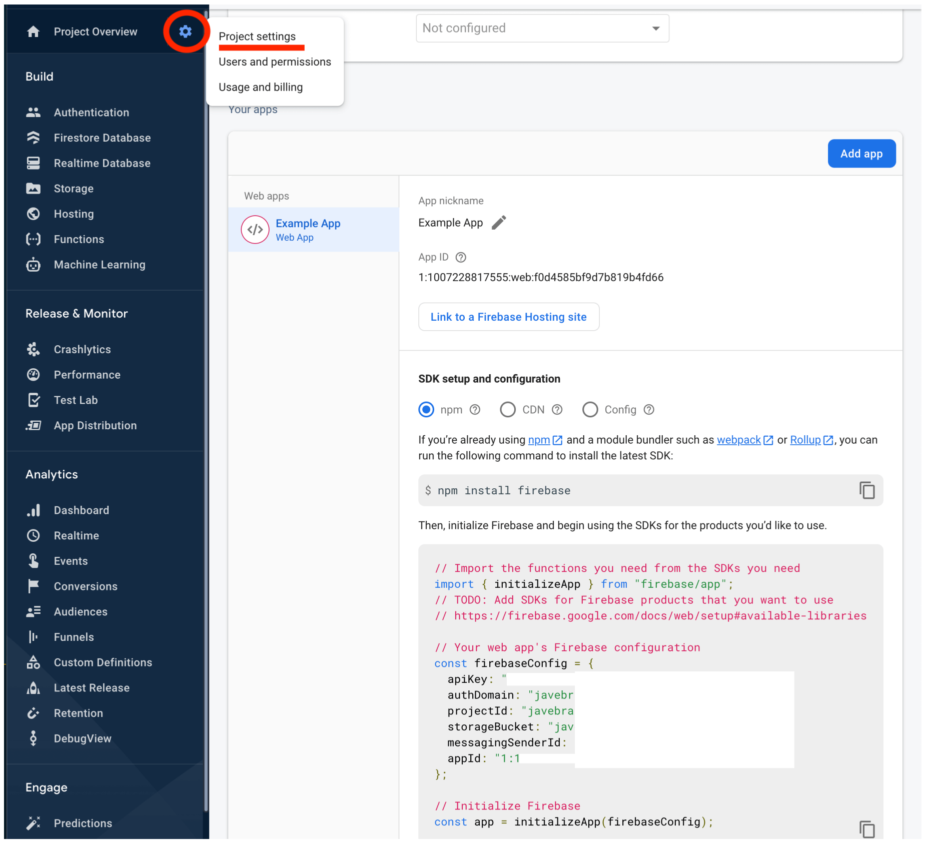
If you don't know where those credentials are, you can open a browser window and go to the Firebase console at https://console.firebase.google.com.

You can go to your Firebase app, and navigate to the settings and find the credentials there (*You'll need to create a new app if you don't have one yet*).

We're also initializing Firebase Authentication, regular authentication with no additional configurations passed.

By the way, while you're in the Firebase Console, go ahead and enable both email/password authentication and the Firestore database. If not, we'll run into issues later.

To enable email & password authentication, we need to go to the Authentication menu, then go to Sign In options, and enable email and password as shown in the image below



App nickname
Example App

App ID
1:1007228817555:web:f0d4585bf9d7b819b4fd66

SDK setup and configuration

[✓ npm] CDN Config

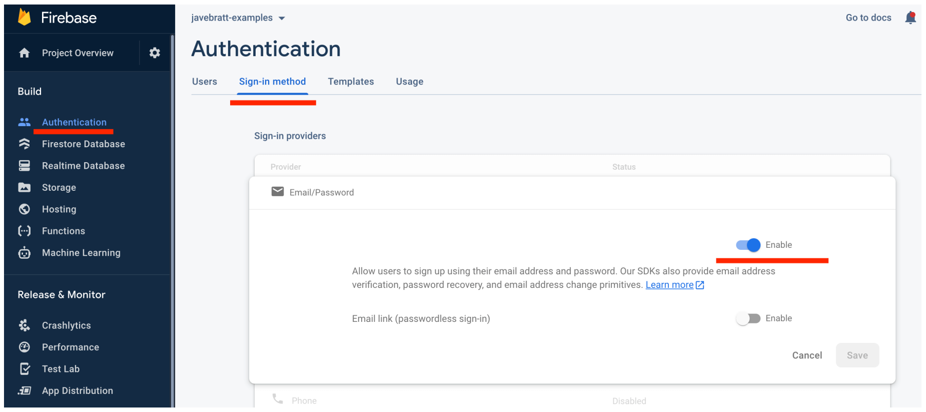
If you're already using npm and a module bundler such as webpack or Rollup, you can run the following command to install the latest SDK:

```
$ npm install firebase
```

Then, initialize Firebase and begin using the SDKs for the products you'd like to use.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "[redacted]
  authDomain: "javebr[redacted]
  projectId: "javebra[redacted]
  storageBucket: "jav[redacted]
  messagingSenderId:[redacted]
  appId: "1:1[redacted]
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Figure 7: Firebase Credentials



Authentication

Sign-in providers

Provider Status

Email/Password

[✓ Enable]

Allow users to sign up using their email address and password. Our SDKs also provide email address verification, password recovery, and email address change primitives. Learn more

Email link (passwordless sign-in) Enable

Phone Disabled

Figure 8: Enable Firebase Authentication

To create your Firestore database, you need to go to the Firestore option in the menu, click on **Create Firestore Database** and select to start with **Test Mode**.

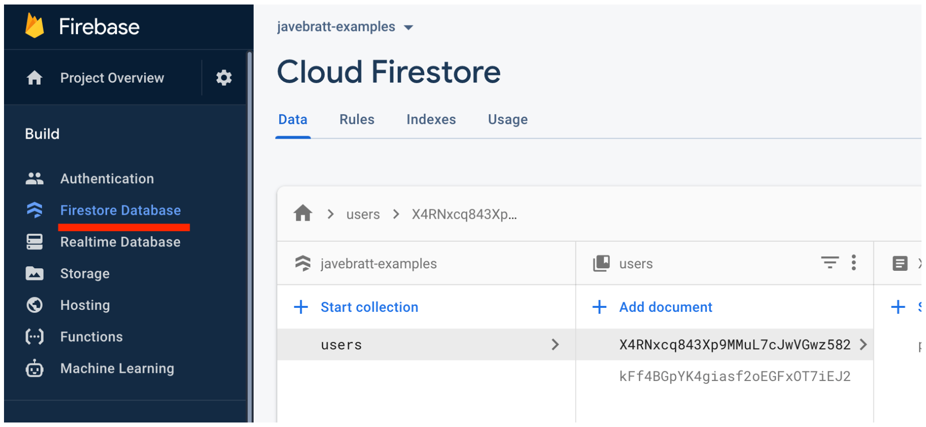


Figure 9: Create Firestore Database

At this point, we should be able to run `ionic serve` in the terminal and have our application running without any compile or build errors.

If you have errors here and want some help, remember you can send me a quick email at jorge@jsmobiledev.com.

Once all of that is done, we're ready to start coding. In the next chapter, we'll build the authentication module of our application, where we'll learn how to create and authenticate users in our app.

User Authentication

One of the most common (*an imoprtant*) parts of a modern application is user authentication. Throughout the next few pages, we'll learn:

- How to create a page to handle user authentication.
- How to create and reuse a component for login, signup, and reset password.
- How to talk to Firebase Authentication functions.
- How to create a guard to prevent unauthenticated users from accessing a page.

Create the Authentication Module

The first thing we want to do, is to create a module where we will handle everything related to authentication.

We can see a module as a group of 'types' of files that concern one functionality, for example, the authentication module would have the authentication components, page, services, etc.

To do that, we open the terminal in the root of our project, and we type:

```
ionic generate page authentication
```

If you want to know what that script does before typing it, you can append the flag `--dry-run`, and call it like this:

```
ionic generate page authentication --dry-run
```

Then you can see the files it will generate, and the files it will update like this:

```
CREATE src/app/authentication/authentication-routing.module.ts (379
    bytes)
CREATE src/app/authentication/authentication.module.ts (528 bytes)
CREATE src/app/authentication/authentication.page.scss (0 bytes)
CREATE src/app/authentication/authentication.page.html (133 bytes)
CREATE src/app/authentication/authentication.page.spec.ts (717 bytes)
CREATE src/app/authentication/authentication.page.ts (288 bytes)
UPDATE src/app/app-routing.module.ts (643 bytes)
```

Once we're comfortable with the result on the console, we can run the script without the `--dry-run` flag and generate those files.

Before we move into the authentication files, let's open the `src/app/app-routing.module.ts`, it should look something like this:

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

const routes: Routes = [
  {
    path: 'home',
    loadChildren: () => import('./home/home.module').then(m =>
        m.HomePageModule),
  },
  {
    path: '',
    redirectTo: 'home',
    pathMatch: 'full',
  },
  {
    path: 'authentication',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
      PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

We want to change it, to remove the home module and to add proper routes for our application, make the changes so that it looks like this:

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

const routes: Routes = [
  {
    path: 'login',
    loadChildren: () =>
```

```
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
  {
    path: 'signup',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
  {
    path: 'reset',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
      PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

You can also remove the `src/app/home/` entire folder since we won't use it.

The changes we made in the `app-routing.module.ts` file means that every time the user goes to the urls: `/login`, `signup`, or `reset`, we're handling the responsibility to the `AuthenticationPageModule`.

Now, we also want to create the authentication form component. We'll use it to load the authentication form depending on the page we're on.

For that, let's open the terminal and type:

```
ionic generate component authentication/auth-form --dry-run
```

Where you can see in the output the files it will generate:

```
CREATE src/app/authentication/auth-form/auth-form.component.scss (0
    bytes)
CREATE src/app/authentication/auth-form/auth-form.component.html (28
    bytes)
CREATE src/app/authentication/auth-form/auth-form.component.spec.ts
    (711 bytes)
CREATE src/app/authentication/auth-form/auth-form.component.ts (279
    bytes)
```

After reviewing that those are the files and paths we were expecting, we can run that command again without the `--dry-run` flag.

Next, let's go to the `authentication.module.ts` file and we need to do two things, first, let's import the authentication form component that we just created:

```
import { AuthFormComponent } from './auth-form/auth-form.component';
```

And then, we add it to the `declarations` array inside of the `NgModule()`

```
@NgModule({
  imports: [CommonModule, FormsModule, IonicModule,
      AuthenticationPageRoutingModule, ReactiveFormsModule],
  declarations: [AuthenticationPage, AuthFormComponent],
})
export class AuthenticationPageModule {}
```

In the end, the `authentication.module.ts` should look like this:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { IonicModule } from '@ionic/angular';

import { AuthenticationPageRoutingModule } from
    './authentication-routing.module';

import { AuthenticationPage } from './authentication.page';
import { AuthFormComponent } from './auth-form/auth-form.component';

@NgModule({
  imports: [CommonModule, FormsModule, IonicModule,
      AuthenticationPageRoutingModule],
  declarations: [AuthenticationPage, AuthFormComponent],
})
export class AuthenticationPageModule {}
```

Now it's an excellent time to test that everything is working, for that, let's open the `src/app/authentication/authentication.page.html` and it should look something like this:

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

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

Let's change the page title, and add our authentication component:

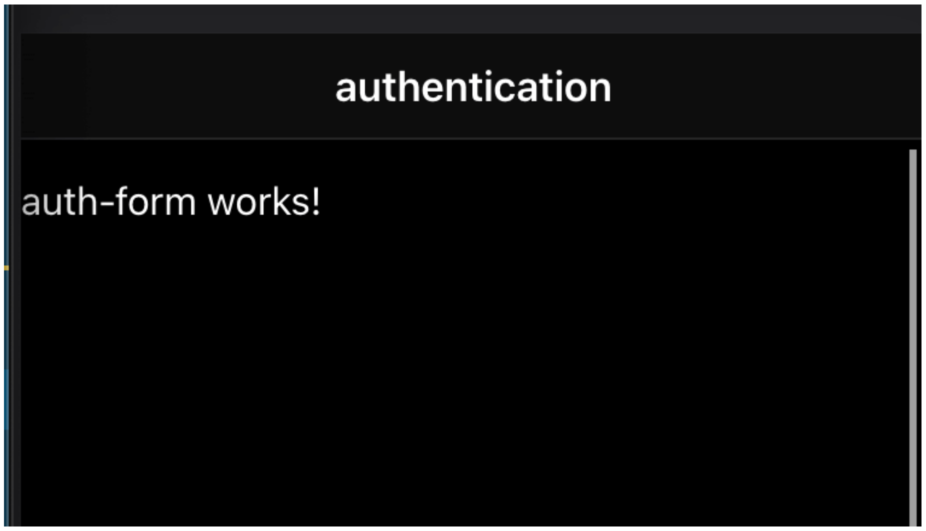
```
<ion-header>
  <ion-toolbar>
    <ion-title>authentication</ion-title>
  </ion-toolbar>
</ion-header>

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

Now go ahead and in the terminal, let's type:

```
ionic serve
```

It will create a local development server to run the application, pay attention to the port so that we can run the app in the browser.

By default, it uses the port `8100`, so you can go to `localhost:8100/login` and you can see a message that says **auth-form works!**.



auth-form works!

Figure 10: Authentication Form Working

The Authentication Component

Now that we have our `authentication-form` component, it is time to create the form itself so that our users can login, signup, or reset their passwords.

First, we want to import the `ReactiveFormsModule`, an angular forms module that lets us build reactive forms.

You can add it to the `authentication.module.ts` in the `imports` array of our `NgModule`:

```
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [..., ReactiveFormsModule]
})
export class AuthenticationPageModule {}
```

Now, we can go into the `auth-form.component.ts` file, and start setting up the functionality we need for the form, for that, we want to import a few things, so go into the `auth-form.component.ts` file and import the following packages:

```
import { Component, Input, Output, EventEmitter, OnInit } from
    '@angular/core';
import { FormGroup, Validators, FormBuilder } from '@angular/forms';
```

We're importing:

- `Input` to take on properties from the parent component.
- `Output`, and `EventEmitter` to send the user credentials back to the authentication page.
- `FormGroup, Validators, FormBuilder` are part of the angular reactive forms and we'll use them to handle the forms in our component.

Then, we'll create our class variables, and inject the Angular's form builder into our constructor:

```
import { Component, Input, Output, EventEmitter, OnInit } from
    '@angular/core';
import { FormGroup, Validators, FormBuilder } from '@angular/forms';
import { UserCredential } from '../authentication.model';

@Component({
  selector: 'app-auth-form',
  templateUrl: './auth-form.component.html',
  styleUrls: ['./auth-form.component.scss'],
})
export class AuthFormComponent implements OnInit {
  @Input() actionButtonText = 'Sign In';
  @Input() isPasswordResetPage = false;
  @Output() formSubmitted = new EventEmitter<any>();
  public authForm: FormGroup;

  constructor(private readonly formBuilder: FormBuilder) {}
}
```

Where:

- `actionButtonText` is the text our form's button will have.
- `isPasswordResetPage` is a flag to determine if we're in the password reset page or not.
- `formSubmitted` is an angular event emitter, this will send the value of our form back to the parent page.
- `authForm` this is out form.

Next, we'll initialize our form, for that we'll create a function to initialize it and call that function from the Angular's `OnInit` lifecycle hook.

```
ngOnInit() {
  this.initializeForm(!this.isPasswordResetPage);
}

initializeForm(showPasswordField: boolean): void {
  this.authForm = this.formBuilder.group({
    email: ['', Validators.compose([Validators.required,
        Validators.email])],
    password: ['', Validators.compose([showPasswordField ?
        Validators.required : null, Validators.minLength(6)])],
  });
}
```

In that function, we're doing three things:

- We're initializing the fields our form will have, in this case, email and password.
- We're adding the validators each field will have.
- We're making sure the required validator on the password field is only active if the page is NOT the password reset page (*Since the password reset page will only have the email field*)

And lastly, we'll create a function called `submitCredentials()`, this function will simply send whatever value the form has to the parent page:

```
submitCredentials(authForm: FormGroup): void {
  if (!authForm.valid) {
    console.log('Form is not valid yet, current value:',
        authForm.value);
  } else {
    const credentials = {
      email: authForm.value.email,
      password: authForm.value.password,
    };
    this.formSubmitted.emit(credentials);
  }
}
```

Now our component is ready to be designed, for that, let's move to the `auth-form.component.html`, right now it should probably look like this:

```
<p>auth-form works!</p>
```

The first thing we want to do is to create our form:

```
<form [formGroup]="authForm"></form>
```

We're binding the `formGroup` property to our authentication form, next, we want to add a couple of fields, one for the email, and another one for the password:

```
<form [formGroup]="authForm">
  <ion-item>
    <ion-label position="stacked">Email</ion-label>
    <ion-input formControlName="email" type="email"
        placeholder="Your email address"> </ion-input>
  </ion-item>

  <ion-item *ngIf="!isPasswordResetPage">
    <ion-label position="stacked">Password</ion-label>
    <ion-input formControlName="password" type="password"
        placeholder="Your password"> </ion-input>
  </ion-item>
</form>
```

Those are regular Ionic inputs, the one particular thing here is the `formControlName`, this binds the fields to the properties we created for our form, email, and password.

And lastly, we'll add a button that will call the `submitCredentials()` function and send the result of the form:

```
<form [formGroup]="authForm">
  <ion-item>
    <ion-label position="stacked">Email</ion-label>
    <ion-input formControlName="email" type="email"
        placeholder="Your email address"> </ion-input>
  </ion-item>
  <ion-item *ngIf="!isPasswordResetPage">
    <ion-label position="stacked">Password</ion-label>
    <ion-input formControlName="password" type="password"
        placeholder="Your password"> </ion-input>
  </ion-item>

  <ion-button (click)="submitCredentials(authForm)" expand="block"
      [disabled]="!authForm.valid">
    {{ actionButtonText }}
  </ion-button>
</form>
```

We're using `[disabled]="!authForm.valid"` to tell our submit button to only be active once the form is valid. To finish this, let's go to the `auth-form.component.scss` file and add some spacing and styles to our form:

```
form {
  padding: 12px;
  margin-bottom: 32px;
  ion-button {
    margin-top: 30px;
  }
}

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

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

ion-input {
  padding: 5px;
}
```

The Authentication Page

Now it's time to use our component, and for that, we'll go into the authentication page itself. Let's open the `authentication.page.ts` file, and add everything we'll need for this first part, I'll cover what everything is inside of the comments:

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

```
@Component({
  selector: 'app-authentication',
  templateUrl: './authentication.page.html',
  styleUrls: ['./authentication.page.scss'],
})
export class AuthenticationPage implements OnInit {
  url: string; // The URL we're at: login, signup, or reset.
  pageTitle = 'Sign In';
  actionButtonText = 'Sign In';
  constructor(private readonly router: Router) {}

  ngOnInit() {
    // First we get the URL, and with that URL we send the
    // proper information to the authentication form component.
    this.url = this.router.url.substr(1);
    if (this.url === 'signup') {
      this.pageTitle = 'Create your Account';
      this.actionButtonText = 'Create Account';
    }

    if (this.url === 'reset') {
      this.pageTitle = 'Reset your Password';
      this.actionButtonText = 'Reset Password';
    }
  }

  handleUserCredentials(userCredentials) {
    // This method gets the form value from the authentication
        component
    // And depending on the URL, it calls the respective method.
    const { email, password } = userCredentials;
    switch (this.url) {
      case 'login':
        this.login(email, password);
        break;
      case 'signup':
        this.signup(email, password);
        break;
      case 'reset':
        this.resetPassword(email);
        break;
    }
  }

  async login(email: string, password: string) {
    // This will hold the logic for the login function.
    console.log(email, password);
  }

  async signup(email: string, password: string) {
    // This will hold the logic for the signup function.
    console.log(email, password);
  }

  async resetPassword(email: string) {
    // This will hold the logic for the resetPassword function.
    console.log(email);
  }
}
```

Now let's visit the `authentication.page.html` file, it looks something like this:

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

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

The first thing we'll do is to replace the header with something more custom:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons *ngIf="url && url !== 'login'" slot="start">
      <ion-back-button defaultHref="/login"></ion-back-button>
    </ion-buttons>
    <ion-title>{{ pageTitle }}</ion-title>
  </ion-toolbar>
</ion-header>
```

It will add a dynamic page title, and it will add the Ionic back button if we're on a page different than the login page.

Next, we want to send all the correct properties to the authentication form:

```
<ion-content>
  <app-auth-form
    *ngIf="url"
    (formSubmitted)="handleUserCredentials($event)"
    [actionButtonText]="actionButtonText"
    [isPasswordResetPage]="url === 'reset'"
  ></app-auth-form>
</ion-content>
```

We're telling our component what function to use to handle the form value, and sending the correct properties depending on the page we're on.

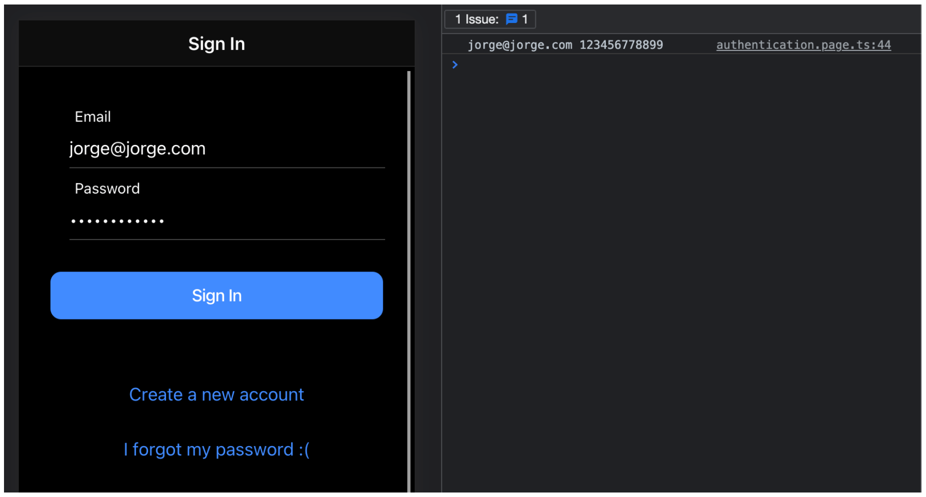
And lastly, we want to add two links to the page, one to create a new account, and the other one to navigate to the password reset page, in the end, the page should look like this:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons *ngIf="url && url !== 'login'" slot="start">
      <ion-back-button defaultHref="/login"></ion-back-button>
    </ion-buttons>
    <ion-title>{{ pageTitle }}</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>
  <app-auth-form
    *ngIf="url"
    (formSubmitted)="handleUserCredentials($event)"
    [actionButtonText]="actionButtonText"
    [isPasswordResetPage]="url === 'reset'"
  ></app-auth-form>

  <ion-button expand="block" fill="clear" routerLink="/signup"
      *ngIf="url && url !== 'signup'">
    Create a new account
  </ion-button>

  <ion-button expand="block" fill="clear" routerLink="/reset"
      *ngIf="url && url !== 'reset'">
    I forgot my password :(
  </ion-button>
</ion-content>
```

Remember, at any point, you can run your app to make sure it's all coming along nicely, and you don't have compile/build errors, for that open the terminal and type:

```
ionic serve
```

And then navigate to `localhost:8100/login`, `localhost:8100/signup`, or `localhost:8100/reset`, that way you'll be able to see the differences in the form, and the page title.



```
jorge@jorge.com 123456778899        authentication.page.ts:44
```

Figure 11: Login Page

Now it is time to give our authentication module actual authentication functionality. For that, we'll start connecting to Firebase.

> NOTE: It is a good practice to separate the business logic into services. That way, we can make changes to those services that affect the entire application.

We'll create a service to handle all of our authentication functionality, for that, stop the terminal if you're serving your application, and type:

```
ionic generate service authentication/authentication
```

It will generate the file `src/app/authentication/authentication.service.ts` where we'll start adding all the Firebase related information, let's go into that file and quickly import everything we need:

```
import { Injectable } from '@angular/core';
import {
  Auth,
  createUserWithEmailAndPassword,
  sendPasswordResetEmail,
  signInWithEmailAndPassword,
  signOut,
  User,
  UserCredential,
} from '@angular/fire/auth';
import { Observable, of } from 'rxjs';
```

I'll go over all the `@angular/fire/auth` as we encounter them. Once everything is imported, let's inject auth into the constructor, this is the AngularFire's auth instance and initialization.

```
constructor(private readonly auth: Auth) {}
```

And then let's start adding the functions we'd like to have. First, we want a function to return the current user:

```
getUser(): User {
  return this.auth.currentUser;
}

getUser$(): Observable<User> {
  return of(this.getUser());
}
```

The `getUser()` function returns the current user logged into the application. This is a synchronous function, so we need to make sure the object is ready to be used before calling it.

For example, if we call it first thing when loading the service, we risk it returning `undefined/null` even when there's a logged-in user.

That's where the `getUser()$` function comes in. It creates an observable of that response and subscribes to the result so that we can call it more safely.

Next, we need a function to log our users in, for that, we can use a Firebase function called `signInWithEmailAndPassword()`, this function takes 3 parameters: the current authentication instance, the email, and password.

```
login(email: string, password: string): Promise<UserCredential> {
  return signInWithEmailAndPassword(this.auth, email, password);
}
```

Be careful when you call the `signInWithEmailAndPassword()`, it doesn't return the user, it returns a `UserCredential` To access the user, you need to get it from the `UserCredential` object as: `userCredential.user`.

Next, we want to create a signup function, for that we use the `createUserWithEmailAndPassword()` function (*Firebase has really explicit names uh?*) which also takes the same parameters as login, it takes the authentication instance, the email, and the password.

```
signup(email: string, password: string): Promise<UserCredential> {
  return createUserWithEmailAndPassword(this.auth, email, password);
}
```

Next, we want a function to reset our users' passwords when needed, for that, we'll use the `sendPasswordResetEmail()` function, which takes the authentication instance, and the email.

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

And lastly, we want to add a function to log our users out. For that, we can use the `signOut()` function, which only takes the current authentication instance.

```
logout(): Promise<void> {
  return signOut(this.auth);
}
```

Now we're ready to go back to our auth page and connect that functionality. Open your `authentication.page.ts` file, and inject the auth service into the constructor:

```
constructor(
  private readonly router: Router,
  private readonly auth: AuthenticationService
) {}
```

Next, go into the login function. Right now is a placeholder that doesn't have anything. Let's add the functionality we need.

For login, we want to call the authentication service login function, and after login, redirect our user to the home page.

```
async login(email: string, password: string) {
  try {
    await this.auth.login(email, password);
    // This will give you an error since we don't have the / URL in
        our routes yet.
    // No worries, we'll add it soon enough.
    this.router.navigateByUrl('');
  } catch (error) {
    console.log(
      `Either we couldn't find your user or there was a problem with
          the password`
    );
  }
}
```

For signup and reset password, we want to do the same. The idea is to call the authentication service, and if the operation is successfull, send the user to the home page. If not, log an error to the console:

```
async signup(email: string, password: string) {
  try {
    await this.auth.signup(email, password);
    this.router.navigateByUrl('');
  } catch (error) {
    console.log(error);
  }
}

async resetPassword(email: string) {
  try {
    await this.auth.resetPassword(email);
    console.log('Email Sent');
    this.router.navigateByUrl('login');
  } catch (error) {
    console.log('Error: ', error);
  }
}
```

The Authentication Guard

The last thing we want to add in this module is a guard, a guard is a router protection that won't let users go into pages they don't have access. To create a guard, open your terminal and type:

```
ionic generate guard authentication/authentication
```

You should see the output in the console saying that it created the file:

```
CREATE src/app/authentication/authentication.guard.ts (468 bytes)
```

Go ahead and open it. It should look something like this:

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

@Injectable({
  providedIn: 'root',
})
export class AuthenticationGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> |
      boolean | UrlTree {
    return true;
  }
}
```

We're going to change the imports a bit, to import something from angular fire, and to import the angular router:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router, UrlTree } from '@angular/router';
import { Auth, onAuthStateChanged } from '@angular/fire/auth';
```

Then we'll add a constructor injecting both the authentication instance and the router.

```
constructor(private readonly auth: Auth, private readonly router:
    Router) {}
```

Next, we'll remove everything from the `canActivate()` method and return an empty promise.

```
canActivate(): Promise<boolean | UrlTree> {
  return new Promise((resolve, reject) => {});
}
```

And inside the promise, we'll give the functionality we need. First, we want to use Firebase's `onAuthStateChanged()` function, it adds an observer to the user's sign-in state:

```
canActivate(): Promise<boolean | UrlTree> {
  return new Promise((resolve, reject) => {
    onAuthStateChanged(this.auth, (user) => {
      // TODO
    });
  });
}
```

And, inside the `onAuthStateChanged()`, we want to add a conditional to ask if the user is there. If we have a user, we resolve the promise with a `true` value.

If there's no user, we reject the promise, and we navigate the user back to the login page:

```
canActivate(): Promise<boolean | UrlTree> {
  return new Promise((resolve, reject) => {
    onAuthStateChanged(this.auth, (user) => {
      if (user) {
        resolve(true);
      } else {
        reject('No user logged in');
        this.router.navigateByUrl('/login');
      }
    });
  });
}
```

And that's it. We now have a fully functional authentication system that will take care of most of the needs of an application. In the next chapter, we'll start using the Firestore database, and we'll create some new pages. I'll show you how to use this guard to protect those pages.

Interacting with your Firestore Database

One of the essential parts of an application is CRUD, creating, reading, updating, and deleting data from the database. Firebase provides a NoSQL database for us: **Firestore**.

We'll use the database in a new module, where we have all of that functionality encapsulated.

We'll create the `Party` module, where we'll handle everything related to events, creating them, updating them, deleting them, etc.

For that, let's open the terminal and generate a new Ionic page with the following line:

```
ionic generate page party
```

You can inspect the terminal's output and see the files it generated and the files it updated:

```
CREATE src/app/party/party-routing.module.ts (343 bytes)
CREATE src/app/party/party.module.ts (465 bytes)
CREATE src/app/party/party.page.scss (0 bytes)
CREATE src/app/party/party.page.html (124 bytes)
CREATE src/app/party/party.page.spec.ts (654 bytes)
CREATE src/app/party/party.page.ts (252 bytes)
UPDATE src/app/app-routing.module.ts (923 bytes)
```

Note the last line: it updated the `app-routing.module.ts` file adding a route for our module it looks like this:

```
{
  path: 'party',
  loadChildren: () => import('./party/party.module').then( m =>
      m.PartyPageModule)
},
```

We'll make three changes in the `app-routing.module.ts` file:

- We'll add a new route for `party/:partyId` so that we can go to the detail page of an event.
- We'll protect both routes with the `AuthenticationGuard` we created in the previous chapter.
- We'll create a redirect to take us to the `/party` page when the user doesn't add any URL.

```
import { NgModule } from '@angular/core';
import { PreloadAllModules, RouterModule, Routes } from
    '@angular/router';
import { AuthenticationGuard } from
    './authentication/authentication.guard';

const routes: Routes = [
  {
    path: '',
    redirectTo: 'party',
    pathMatch: 'full',
  },
  {
    path: 'login',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
  {
    path: 'signup',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
  {
    path: 'reset',
    loadChildren: () =>
        import('./authentication/authentication.module').then(m =>
        m.AuthenticationPageModule),
  },
  {
    path: 'party',
    loadChildren: () => import('./party/party.module').then(m =>
        m.PartyPageModule),
    canActivate: [AuthenticationGuard],
  },
  {
    path: 'party/:partyId',
    loadChildren: () => import('./party/party.module').then(m =>
        m.PartyPageModule),
    canActivate: [AuthenticationGuard],
  },
];

@NgModule({
  imports: [RouterModule.forRoot(routes, { preloadingStrategy:
      PreloadAllModules })],
  exports: [RouterModule],
})
export class AppRoutingModule {}
```

Once we have that, let's understand what we need for our module to work:

- We need the Party page where we'll show a list of parties (*this is the one we just created*).
- A component to show the details of a specific party.
- A component to create a new party.
- A service to interact with Firebase.

Let's create the files themselves now so that we can serve our application and focus solely on coding instead of switching from coding to config back and forth.

For that, open your terminal and type:

```
ionic generate service party/party
ionic generate component party/detail-party
ionic generate component party/create-party
```

It will generate a bunch of files. You can see the output in the terminal:

```
CREATE src/app/party/detail-party/detail-party.component.scss (0
    bytes)
CREATE src/app/party/detail-party/detail-party.component.html (31
    bytes)
CREATE src/app/party/detail-party/detail-party.component.spec.ts
    (732 bytes)
CREATE src/app/party/detail-party/detail-party.component.ts (291
    bytes)

CREATE src/app/party/create-party/create-party.component.scss (0
    bytes)
CREATE src/app/party/create-party/create-party.component.html (31
    bytes)
CREATE src/app/party/create-party/create-party.component.spec.ts
    (732 bytes)
CREATE src/app/party/create-party/create-party.component.ts (291
    bytes)
```

```
CREATE src/app/party/party.service.spec.ts (352 bytes)
CREATE src/app/party/party.service.ts (134 bytes)
```

Now we can start coding. First, let's open the `party.module.ts` file and add the newly created components to the `declarations` array inside of the `NgModule`:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';

import { IonicModule } from '@ionic/angular';

import { PartyPageRoutingModule } from './party-routing.module';

import { PartyPage } from './party.page';
import { CreatePartyComponent } from
    './create-party/create-party.component';
import { DetailPartyComponent } from
    './detail-party/detail-party.component';

@NgModule({
  imports: [CommonModule, FormsModule, IonicModule,
      PartyPageRoutingModule],
  declarations: [PartyPage, CreatePartyComponent,
      DetailPartyComponent],
})
export class PartyPageModule {}
```

Now that both components are available for use, let's open the `party-routing.module.ts`. It should look like this:

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

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

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
```

```
export class PartyPageRoutingModule {}
```

We want to have three routes:

- The empty route that will trigger every time a user navigates to `/party` and point to the `PartyPage`.
- The `new` route that will trigger every time a user navigates to `/party/new` and point to the `CreatePartyComponent`.
- The `:partyId` route that will trigger every time a user navigates to `/party/id` and point to the `DetailPartyComponent`.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CreatePartyComponent } from
    './create-party/create-party.component';
import { DetailPartyComponent } from
    './detail-party/detail-party.component';

import { PartyPage } from './party.page';

const routes: Routes = [
  {
    path: '',
    component: PartyPage,
  },
  {
    path: 'new',
    component: CreatePartyComponent,
  },
  {
    path: ':partyId',
    component: DetailPartyComponent,
  },
];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule],
})
export class PartyPageRoutingModule {}
```

Now, you can run your application with `ionic serve`, and when you log in, you'll be redirected to the main route, where you'll see the party page, it's empty right now, but don't worry about it, our next step will be to add the functionality to show the list of parties there.

Listing Parties from Firestore

We'll start with listing items from the database, in this case, the parties. A party has several properties:

```
{
  id: string; // The ID of the document.
  name: string; // The user friendly name.
  date: number; // The date it is happening.
  ticketPrice: number; // The price for people to go into the party.
  cost: number; // The $$ you're spending throwing the party.
  revenue: number; // The income - the expenses.
}
```

In the `PartyPage` we want to show a list of the parties we have in the application. For that, let's open the `party.page.ts` file, and let's create a class variable called `partyList`. Also, remember you can create an interface for the party properties so that it is properly typed.

By now, the party page class should look like this:

```
import { Component } from '@angular/core';
import { Party } from './party.model'; // You can add this
    party.model.ts file in the same /party folder.

@Component({
  selector: 'app-party',
  templateUrl: './party.page.html',
  styleUrls: ['./party.page.scss'],
})
export class PartyPage {
  readonly partyList$: Party[] = [];
  constructor() {}
}
```

We'll initialize the `partyList` property as an empty array, we don't have anything to show in our database at this point, so we'll leave it like that while we work on adding items to the database.

Now, we can create a simple (*but elegant*) template to display the list of parties once it's ready.

First, we want to change the page's header, so open `party.page.html` file and replace the header with this:

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

```
</ion-header>

<ion-content class="ion-padding"></ion-content>
```

Next, we want to create a list, for that we can use Ionic's `ion-list` component, the idea is that we loop through `partyList` and display the most relevant information of each event:

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

<ion-content class="ion-padding">
  <ion-list>
    <ion-item tappable *ngFor="let party of partyList$"
        routerLink="/party/{{ party.id }}">
      <ion-label>
        <h2>{{party?.name}}</h2>
        <p>Ticket: <strong>${{party?.ticketPrice}}</strong></p>
        <p>Date: <strong>{{party?.date | date }}</strong></p>
      </ion-label>
    </ion-item>
  </ion-list>
</ion-content>
```

Note that when you click on an item on the list, it will take you to the `/party/id` URL, which will trigger the `DetailPartyComponent`.

Then, we want a way for the user to add new parties, for that, we're going to create a floating action button the users can click and navigate to the `CreatePartyComponent`.

To create the Floating Action Button, Ionic has a handy component called `ion-fab` where you pass the position, and they take care of the rest :)

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

<ion-content class="ion-padding">
  <ion-list>
    <ion-item tappable *ngFor="let party of partyList$"
        routerLink="/party/{{ party.id }}">
      <ion-label>
```

```
        <h2>{{party?.name}}</h2>
        <p>Ticket: <strong>${{party?.ticketPrice}}</strong></p>
        <p>Date: <strong>{{party?.date | date }}</strong></p>
      </ion-label>
    </ion-item>
  </ion-list>

  <ion-fab vertical="bottom" horizontal="end" slot="fixed">
    <ion-fab-button routerLink="/party/new">
      <ion-icon name="add"></ion-icon>
    </ion-fab-button>
  </ion-fab>
</ion-content>
```

Adding new items to the database

We have our page to show the list of parties, but no party to show, so now would be a good time to start adding them, for that, let's work on our `CreatePartyComponent`. First, let's open our `create-party.component.ts` file, and make it look like this:

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { Party } from '../party.model';

@Component({
  selector: 'app-create-party',
  templateUrl: './create-party.component.html',
  styleUrls: ['./create-party.component.scss'],
})
export class CreatePartyComponent implements OnInit {
  name: string;
  ticketPrice: number;
  cost: number;
  date: any;

  constructor(private readonly router: Router) {}

  ngOnInit() {}

  async createEvent(party: Partial<Party>): Promise<void> {
    // Save the party to the database
    console.log(party);
    await this.router.navigateByUrl('party');
  }
  isValidForm(): boolean {
    return this.name && this.ticketPrice && this.cost && this.date;
  }
}
```

Here's what's going on:

- We're importing the Party model, and the angular router.
- Were creating class variables for `name`, `ticketPrice`, `cost`, and `date`, to use them in the HTML template.
- We're creating a placeholder function call `createParty()` which we'll use to create the party and then send the user back to the party list.
- We're creating the `isValid()` function, this will check if all the items are there, as a way to validate the submit form.

Now we can move to the template, so let's open the `create-party.component.html` file, and the first thing we want to do is add a header.

We'll add a simple title, and a back button using the `ion-back-button` component:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-back-button defaultHref="/party"></ion-back-button>
    </ion-buttons>
    <ion-title>Add new Event</ion-title>
  </ion-toolbar>
</ion-header>

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

Now it is time to start adding our fields, for the previous lesson we used the angular `ReactiveFormsModule`, for this one, we'll try something a bit different so that you're aware of the options you have.

We'll use the angular's `FormsModule`, which lets us bind the inputs with the `[(ngModel)]` property (*that's why we created the class variables for the properties inside the component's class*).

The first one we'll create will be a text input we can bind to the `name` property:

```
<ion-content class="ion-padding">
  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Event Name</ion-label>
    <ion-input [(ngModel)]="name" type="text" placeholder="What's
        your event's name?"> </ion-input>
  </ion-item>
</ion-content>
```

Next, we can add two number inputs, one for the `ticketPrice` and another one for the `cost`:

```
<ion-content class="ion-padding">
  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Event Name</ion-label>
    <ion-input [(ngModel)]="name" type="text" placeholder="What's
        your event's name?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Price</ion-label>
    <ion-input [(ngModel)]="ticketPrice" type="number"
        placeholder="How much will guests pay?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Cost</ion-label>
    <ion-input [(ngModel)]="cost" type="number" placeholder="How
        much are you spending?"> </ion-input>
  </ion-item>
</ion-content>
```

Next, we'll use Ionic's new `datetime` component to show a calendar view inside the form, so that our users can easily pick the date of their events:

```
<ion-content class="ion-padding">
  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Event Name</ion-label>
    <ion-input [(ngModel)]="name" type="text" placeholder="What's
        your event's name?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Price</ion-label>
    <ion-input [(ngModel)]="ticketPrice" type="number"
        placeholder="How much will guests pay?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Cost</ion-label>
    <ion-input [(ngModel)]="cost" type="number" placeholder="How
        much are you spending?"> </ion-input>
  </ion-item>

  <ion-datetime class="party-date" size="fixed" presentation="date"
      [(ngModel)]="date"></ion-datetime>
```

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

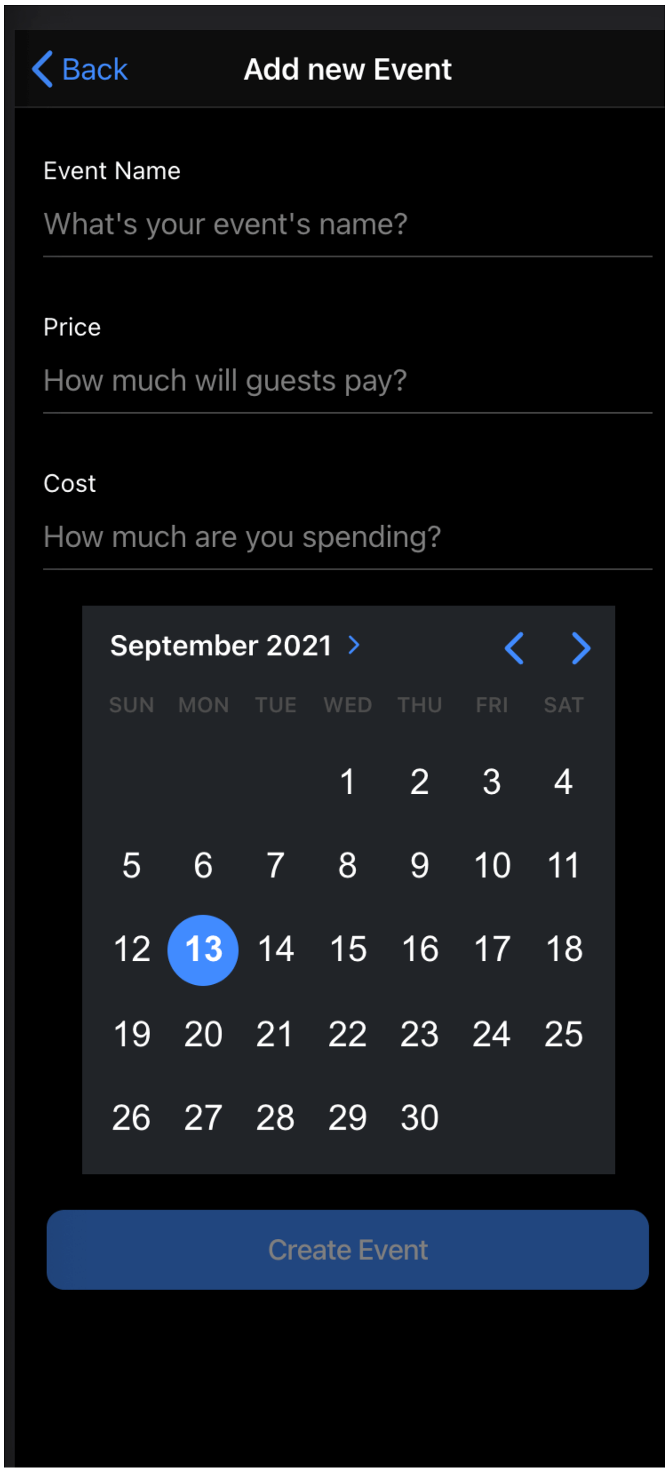
And lastly, we'll add a button to call the `createEvent()` function we created and pass the properties:

```
<ion-content class="ion-padding">
  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Event Name</ion-label>
    <ion-input [(ngModel)]="name" type="text" placeholder="What's
        your event's name?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Price</ion-label>
    <ion-input [(ngModel)]="ticketPrice" type="number"
        placeholder="How much will guests pay?"> </ion-input>
  </ion-item>

  <ion-item class="ion-no-padding">
    <ion-label position="stacked">Cost</ion-label>
    <ion-input [(ngModel)]="cost" type="number" placeholder="How
        much are you spending?"> </ion-input>
  </ion-item>

  <ion-datetime class="party-date" size="fixed" presentation="date"
      [(ngModel)]="date"></ion-datetime>

  <ion-button [disabled]="!isValidForm()" expand="block"
      (click)="createEvent({name, date, ticketPrice, cost})">
    Create Event
  </ion-button>
</ion-content>
```

At this point, you should be able to run the application with `ionic serve`, navigate to the create page, and see the form, if you add properties there and click on `Create Event` you'll be able to see the properties in the browser's console.

So far we've managed to get the UI ready, but we still have no way to add those parties to the database, for that, let's head to the `party.service.ts` file, and import things we'll need:

```
import { Injectable } from '@angular/core';
import { addDoc, collection, Firestore } from
    '@angular/fire/firestore';
import { AuthenticationService } from
    '../authentication/authentication.service';
import { Party } from './party.model';
```



Event Name

What's your event's name?

Price

How much will guests pay?

Cost

How much are you spending?

Figure 12: Create Party Form

Here's what we're importing:

- `AuthenticationService` to connect with the auth service, we'll use this to get the user's ID.
- `addDoc, collection, Firestore` are for adding documents to the database and connecting to Firestore.

Now, let's inject both the authentication service and the Firestore instance into the constructor:

```
constructor(
  private readonly auth: AuthenticationService,
  private readonly firestore: Firestore
) {}
```

And lastly, let's create our `createParty()` function, it will take the party as a parameter, and the first thing that it needs to do is to get the user's ID.

```
createParty(party: Partial<Party>) {
  const userId: string = this.auth.getUser().uid;
}
```

We're getting the user's ID to store the parties under the user's document in the Firestore database. Then, we want to get a reference to the `party` collection. Think of it like the location where we want to add the new document.

```
createParty(party: Partial<Party>) {
  const userId: string = this.auth.getUser().uid;
  const partyCollection = collection(
    this.firestore,
    `users/${userId}/party/`
  );
  return addDoc(partyCollection, party);
}
```

The firestore `collection` method takes two parameters, the Firestore instance, and the path to where we'll store the document.

And lastly, we want to add the `addDoc` function and pass that collection reference, and the object we'll be storing:

```
createParty(party: Partial<Party>) {
  const userId: string = this.auth.getUser().uid;
  const partyCollection = collection(
    this.firestore,
    `users/${userId}/party/`
  );
  return addDoc(partyCollection, party);
}
```

You can test this right now, let's go to the `create-party.component.ts` and inject this party service into the constructor:

```
constructor(
  private readonly router: Router,
  private readonly partyService: PartyService
) {}
```

And let's see our `createEvent()` function, right now it is a placeholder that looks like this:

```
async createEvent(party: Partial<Party>) {
  // Save the party to the database
  console.log(party);
  await this.router.navigateByUrl('party');
}
```

And we'll do two things, first, we want to initialize the revenue as `0`, since no party will start with tickets already sold:

```
async createEvent(party: Partial<Party>) {
  party.revenue = 0;
  // Save the party to the database
  console.log(party);
  await this.router.navigateByUrl('party');
}
```

And next, we'll replace our comment with a call to the party service's new function we just created, the `createParty()` function:

```
async createEvent(party: Partial<Party>) {
  party.revenue = 0;
  await this.partyService.createParty(party);
  console.log(party);
  await this.router.navigateByUrl('party');
}
```

You can go ahead and fill your form now, and you can check your database to see if the document was created or not. If there's any issues and the document wasn't created feel free to shoot me an email at (jorge@jsmobiledev.com)[mailto:jorge@jsmobiledev.com].

Now that we're adding items to the database would be a good time to connect a function that can list them, for that, let's get back to the `party.service.ts` file and create a function called `getPartyList`.

This function needs to get the user's ID, and then get the collection of parties from the database.

First, let's create the function:

```
getPartyList() {}
```

And then we're going to use a bit of `rxjs` to get everything as an observable, instead of trying to get the user's ID, and then getting the path and so on.

First, let's import the `rxjs` operators we'll need:

```
// If you're on rxjs 7+
import { map, switchMap } from 'rxjs';

// If you're on rxjs 6
import { map, switchMap } from 'rxjs/operators';
```

Next, we'll do a sequence of steps:

- We'll subscribe to the `getUser$()` function on the authentication service.
- Once we get the user we'll use the response from there to create a reference to the collection of parties.
- Once we have that reference, we'll return the collection's data.
- Remember you need to import the functions from `@angular/fire/firestore`: `collection`, `collectionData`

```
getPartyList() {
  return this.auth.getUser$().pipe(
    map(({ uid: userId }: User) =>
      collection(this.firestore, `users/${userId}/party`)
    ),
    switchMap((partyCollection) =>
      collectionData(partyCollection, { idField: 'id' })
    )
  );
}
```

Now we can go back to the `PartyPage` and connect this new function so that we can see the list of parties in our application, so let's open the `party.page.ts` file, and first, inject the part service into the constructor:

```
constructor(private readonly partyService: PartyService) {}
```

Next, look for this line where we initialized the parties:

```
readonly partyList$: Party[] = [];
```

And replace it with the initialization of the parties from the part service:

```
readonly partyList$ = this.partyService.getPartyList();
```

At this point, you'll get an error in the template, something like this:

```
Error: Cannot find a differ supporting object '[object Object]' of
    type 'object'. NgFor only supports binding to Iterables such as
    Arrays
```

This is happening because in the template we're trying to look through an array of parties, but the service is not returning an array, it's returning an Observable of an array.

To fix this, we'll go into the `party.page.html` page, and look for the `*ngFor` directive we added, it should look like this:

```
<ion-item *ngFor="let party of partyList$" routerLink="/party/{{
    party.id }}"></ion-item>
```

And right after the `partyList$` variable, we'll add the `async` pipe provided by Angular, it will handle observable unwrapping for us, as well as unsubscribing from it when we leave the component:

```
<ion-item *ngFor="let party of partyList$ | async"
    routerLink="/party/{{ party.id }}"></ion-item>
```

If you navigate to `/party` now you should be able to see the list of parties we created

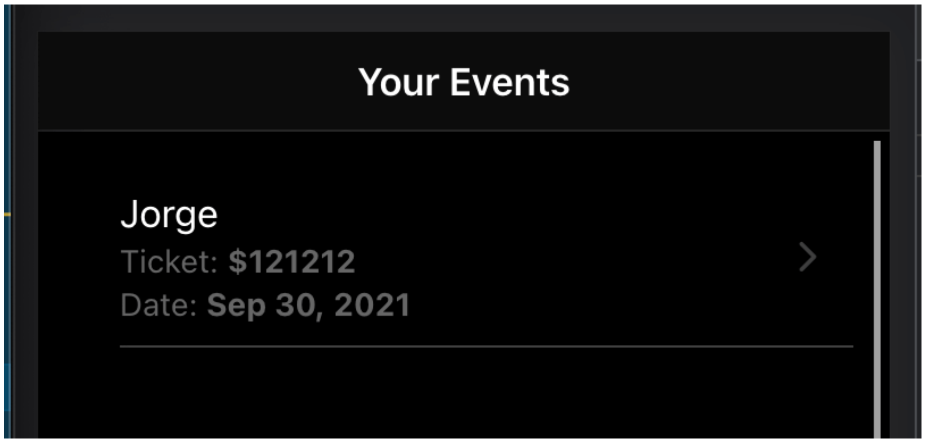


Figure 13: Your Events Page

Navigating to the detail page

If you click on one of the parties in the list, you'll be able to navigate to the `DetailPartyComponent`, right now it doesn't have much, it's the default Angular component placeholder that says:

```
<p>detail-party works!</p>
```

We're going to start by initializing the current object we'll see, the event or party we're navigating to, for that, let's go into the `detail-party.component.ts` file, and make it look like this:

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Party } from '../party.model';

@Component({
  selector: 'app-detail-party',
  templateUrl: './detail-party.component.html',
  styleUrls: ['./detail-party.component.scss'],
})
export class DetailPartyComponent implements OnInit {
  // We're creating a class variable for the currentParty
  currentParty: Party;
  // We're injecting Angular's ActivatedRoute into the constructor.
  constructor(private readonly route: ActivatedRoute) {}

  ngOnInit() {
    // We're using Angular's ActivatedRoute to get the PartyId from
        the URL.
    const partyId: string =
        this.route.snapshot.paramMap.get('partyId');
    // We're passing the partId to the initialize party function
    this.initializeParty(partyId);
  }

  initializeParty(partyId: string): void {
    // This function will get the part so that we can show it in the
        UI.
  }
}
```

Let's focus on the `initializeParty()` function, we want it to do two things:

- Call the party service to get the current party.
- Assign that response to the `currentParty` variable.

But we don't have a function to get the current party from the database, so let's fix that first, go to the `party.service.ts` file, and create a function called `getPartyDetail()`, it should take the `partyId` as a parameter:

```
getPartyDetail(partyId: string) {
  // Get the actual document from Firestore
}
```

We'll use the same approach to when we got the list of parties, we'll get the `userId`, to create the path to the database, then use `rxjs` to chain those together and get the document from the database:

```
getPartyDetail(partyId: string) {
  return this.auth.getUser$().pipe(
    map(({ uid: userId }: User) => doc(this.firestore,
        `users/${userId}/party/${partyId}`)),
    switchMap(partyDocument => docData(partyDocument))
  );
}
```

Notice we are using `doc()` to create a reference to the document in Firestore, and then `docData()` to get that document into our application, so don't forget to add both of them to the `@angular/fire/firestore` imports:

```
import {
  doc,
  docData,
  ...,
} from '@angular/fire/firestore';
```

Now that we have that, we can go back to our `detail-party.component.ts` file, and change our `initializeParty()` function to call the service and get the party:

```
constructor(
  private readonly route: ActivatedRoute,
  private readonly partyService: PartyService
) {}
initializeParty(partyId: string): void {
  this.partyService.getPartyDetail(partyId).subscribe(party => {
    this.currentParty = party;
    if (this.currentParty) {
      this.currentParty.id = partyId;
    }
  });
}
```

Now we can start working on the view. After all, what good does it to have all this data if we can't see it hehe.

For that, let's go to the `detail-party.component.html`, and the first thing we want to add is a header:

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
```

```
      <ion-back-button defaultHref="/party"></ion-back-button>
    </ion-buttons>
    <ion-title>{{ currentParty?.name }}</ion-title>
  </ion-toolbar>
</ion-header>
```

We're adding the regular Ionic header, with a back button and using the party's name as the page title.

One thing you've probably noticed is that we're using `defaultHref` in our back buttons, this is because we want these specific back buttons to work even if the user directly navigates to this page with the URL.

For example, if the user grabs the URL `/party/4567uikmnbvfr6u` and pastes it in the browser, you won't see the back button because there's no page to go back to, that's the first page the user opened.

But, if you add the `defaultHref`, Ionic will show the back button and when clicked it will take the user to the URL we're pointing out there.

After the header, let's add some content to display the properties of the party:

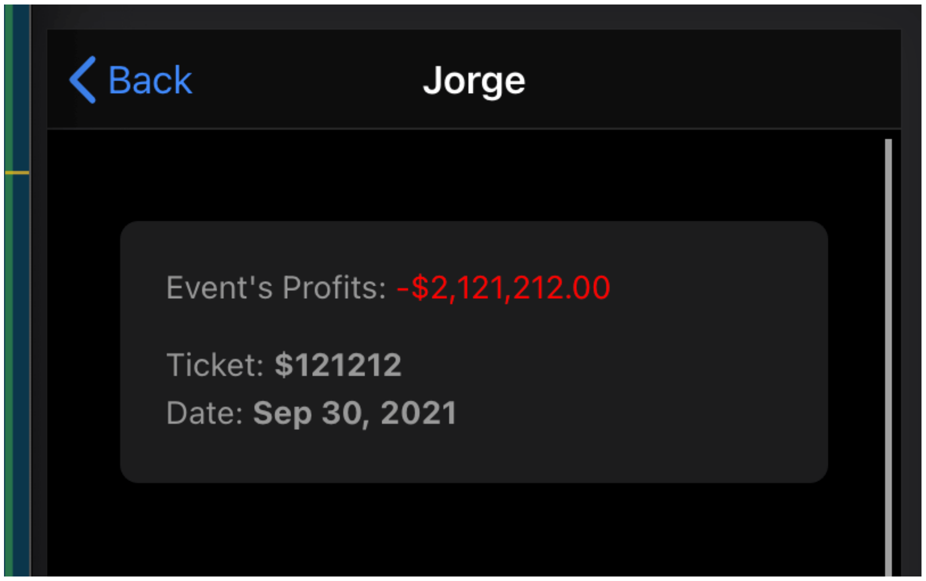
```
<ion-content class="ion-padding">
  <ion-card>
    <ion-card-header>
      Event's Profits:
      <span
        [class.profitable]="currentParty?.revenue -
            currentParty?.cost > 0"
        [class.no-profit]="currentParty?.revenue -
            currentParty?.cost <= 0"
      >
        {{ currentParty?.revenue - currentParty?.cost | currency }}
      </span>
    </ion-card-header>
    <ion-card-content>
      <p>Ticket: <strong>${{ currentParty?.ticketPrice
          }}</strong></p>
      <p>Date: <strong>{{ currentParty?.date | date }}</strong></p>
    </ion-card-content>
  </ion-card>
</ion-content>
```

We're showing whatever properties we have in the party, and we have a card header that shows the current profits. Profit is the revenue minus the cost, and if we're on negative values we're adding a CSS class called `no-profit`; if we're on positive values, we're adding the class `profitable`.

You can go ahead and add both to the CSS file:

```
.profitable {
  color: #22bb22;
}

.no-profit {
  color: #ff0000;
}
```

Now, if you run your app and navigate to the detail page, you should be able to see all the properties right there in the page.



Event's Profits: -$2,121,212.00

Ticket: **$121212**
Date: **Sep 30, 2021**

Figure 14: Detail Page Showing Several Properties

Working with Transactions

It's not fun to see that profits are in the red, so to fix that, we'll add some functionality to be able to sell tickets, that way we can increase the revenue until the profits are no longer in the red.

Since we're on the `detail-party.component.html` file let's handle the view first, right below the card, you'll add a title showing the number of tickets sold, then add a couple of buttons that show a + and a - sign, we'll use them to either add or remove tickets sold (*We can call them sell and refund*).

```
<h5>Tickets Sold: {{ currentParty?.revenue /
    currentParty?.ticketPrice }}</h5>
<ion-button (click)="addTicketOperation('refund')">
```

```
  Refund Ticket
  <ion-icon slot="end" name="remove"></ion-icon>
</ion-button>
<ion-button (click)="addTicketOperation('add')">
  Sell Ticket
  <ion-icon slot="end" name="add"></ion-icon>
</ion-button>
```

You'll see an error because the `addTicketOperation()` function doesn't exist. So let's take a moment and add it. It will get the type as a parameter, the type is if we're doing a refund, or selling a ticket.

Then, it will call the party service, and call the `addTicketOperation()` function from that service (*we'll create it next*), and send a few parameters:

- The party's ID.
- The ticket price.
- The type (*refund or add*).

```
async addTicketOperation(type: string) {
  try {
    await this.partyService.addTicketOperation(
      this.currentParty.id,
      this.currentParty.ticketPrice,
      type
    );
  } catch (error) {
    console.log(error);
  }
}
```

Now, let's move to the `party.service.ts` file and create the method there too:

```
async addTicketOperation(
  partyId: string,
  ticketCost: number,
  type: string = 'add'
) {
  try {
    // We'll add the logic here
  } catch (error) {
    console.log('Transaction failed: ', error);
    throw error;
  }
}
```

Right now, it's not doing much, it takes the parameters you're sending, and it will run the business logic inside a try/catch block in case something fails.

Next, we want to get the user's ID, and create a reference to the document for that party in Firestore:

```
async addTicketOperation(
  partyId: string,
  ticketCost: number,
  type: string = 'add'
) {
  try {
    const userId: string = this.auth.getUser().uid;
    const partyDocRef = doc(this.firestore,
        `users/${userId}/party/${partyId}`);

  } catch (error) {
    console.log('Transaction failed: ', error);
    throw error;
  }
}
```

And lastly, we'll use the `runTransaction()` function to update the revenue.

```
async addTicketOperation(
  partyId: string,
  ticketCost: number,
  type: string = 'add'
) {
  try {
    const userId: string = this.auth.getUser().uid;
    const partyDocRef = doc(this.firestore,
        `users/${userId}/party/${partyId}`);

    await runTransaction(this.firestore, async (transaction) => {
      const partyDoc = await transaction.get(partyDocRef);
      const newRevenue =
        type === 'add'
          ? partyDoc.data().revenue + ticketCost
          : partyDoc.data().revenue - ticketCost;
      transaction.update(partyDocRef, { revenue: newRevenue });
    });
  } catch (error) {
    console.log('Transaction failed: ', error);
    throw error;
  }
}
```

This is something new, so let me break down what happened there:

The `runTransaction()` function takes the Firestore instance and runs a transaction in the database. A transaction will make sure you update the proper value. Let me explain:

What happens if you have 10 people selling tickets?

What happens when they all click the sell button at the same time?

Without a transaction, they would get the current revenue, let's say they all get 0, and add the new ticket value, so when they all update the document, they're all setting the new revenue as 15, for example.

With a transaction, things are a bit different. Here's how it happens:

- Firestore runs the transaction.
- You get the document ready to update whatever property you want to update.
- Firestore checks if the document has changed, if not, you're good, and your update goes through.
- If the document has changed, then Firestore gets you the new version of the document, then it runs your updates again.

That way, if they all sell a ticket at the same time, Firestore will start processing those transactions, and instead of they overwrite each other's work, it will update the proper values.

In the end, you can start clicking the sell and refund buttons and you'll be able to see how the application is working.

Deleting Items from Firestore

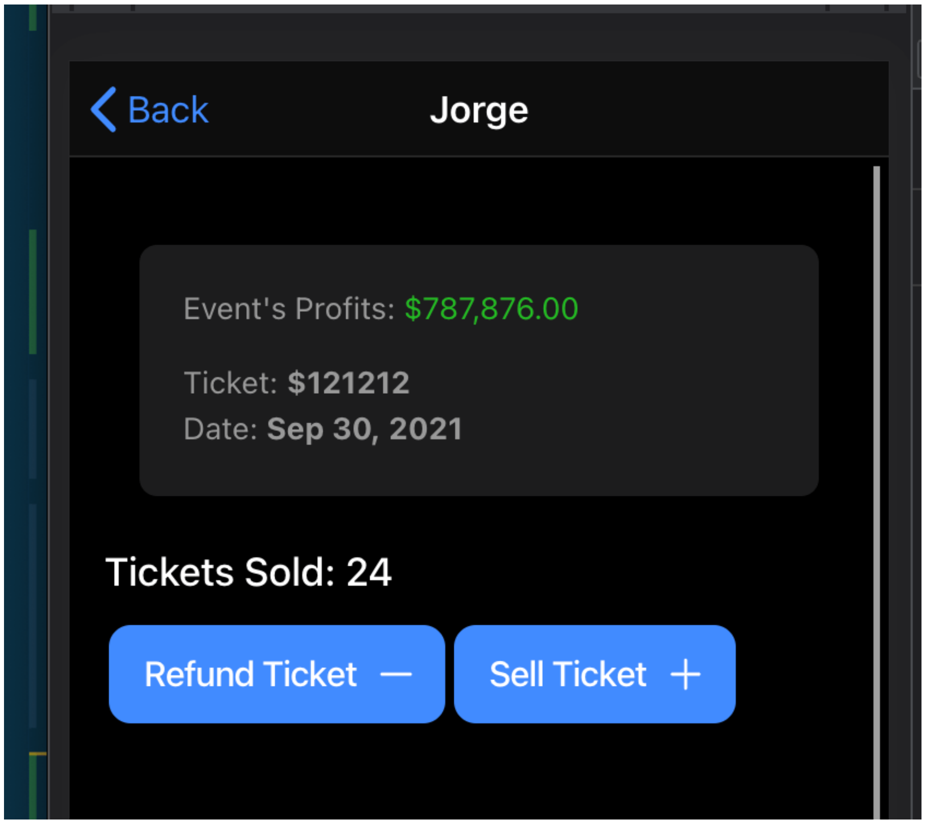
We want to add one last feature to our application, we want to add the ability to remove parties from the system, we'll do it in reverse order this time, since we're already in the `party.service.ts` file, let's create a function called `deleteParty()` it needs to take in the party's ID as a parameter:

```
deleteParty(partyId: string): Promise<void> {}
```

Then, we'll do what we've been doing for a while, we get the user's ID, and we create a reference to the document we want to delete:

```
deleteParty(partyId: string) {
  const userId: string = this.auth.getUser().uid;
  const documentReference = doc(this.firestore,
      `users/${userId}/party/${partyId}`);
}
```

And lastly, we will call Firestore's `deleteDoc()` function, which takes the document's reference as a parameter.



Event's Profits: $787,876.00

Ticket: $121212
Date: Sep 30, 2021

Tickets Sold: 24

Figure 15: Detail page Showing Profitable Party

```
deleteParty(partyId: string): Promise<void> {
  const userId: string = this.auth.getUser().uid;
  const documentReference = doc(this.firestore,
      `users/${userId}/party/${partyId}`);
  return deleteDoc(documentReference);
}
```

Remember that we have to add the `deleteDoc()` function to the firestore imports:

```
import {
  ...,
  deleteDoc,
} from '@angular/fire/firestore';
```

Now let's move to the `detail-party.component.ts` file and create the function that will call the firestore delete functionality:

```
async removeParty() {
  try {
    await this.partyService.deleteParty(this.currentParty.id);
    this.router.navigateByUrl('party');
  } catch (error) {
    console.log(error);
  }
}
```

The function is doing two things:

- It is calling the service to delete this party.
- It takes the user to the party page to show the list of parties available.

Remember we need to import the router and inject it into the constructor:

```
import { ActivatedRoute, Router } from '@angular/router';

constructor(
  private readonly route: ActivatedRoute,
  private readonly partyService: PartyService,
  private readonly router: Router
) {}
```

Now, before moving into the view to create the button to call this function, we need to talk about something. This is a horrible practice.

Deleting an item from the database is a destructive operation. There's no recovering from that one, so what happens when your user clicks by mistake? Do you want to remove the item without hesitation?

Instead of doing that, a better practice is to introduce a confirmation step, where you ask the user if they're sure they want to delete that item.

For that, we can use Ionic's alerts, first, let's import the alert service and inject it into the constructor:

```
import { AlertController } from '@ionic/angular';

constructor(
  private readonly route: ActivatedRoute,
  private readonly partyService: PartyService,
  private readonly router: Router,
  private readonly alertCtrl: AlertController
) {}
```

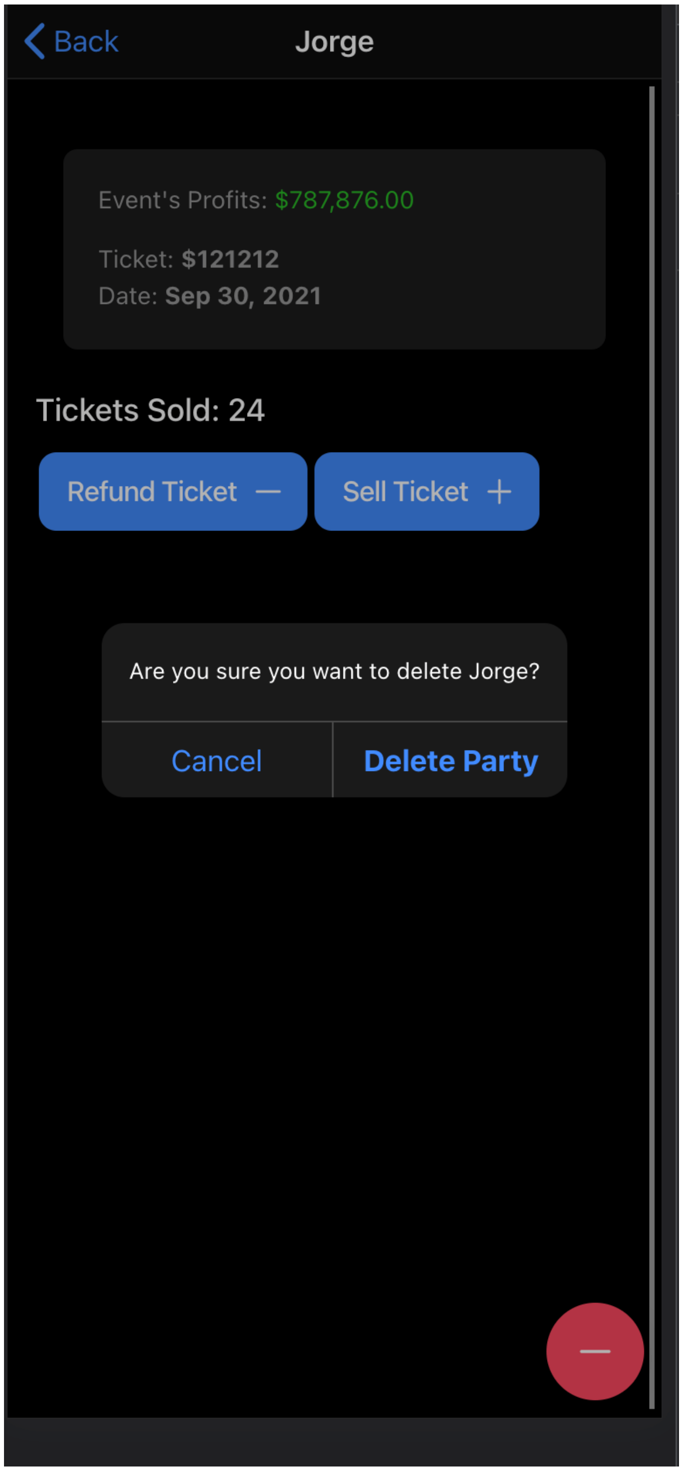
Next, we want to create a new function, let's call it `removePartyAlert()`. This function will display a confirmation alert, and if the user clicks on confirm, it then calls the remove party function we created:

```
async removePartyAlert() {
  const alert = await this.alertCtrl.create({
    message: `Are you sure you want to delete
        ${this.currentParty.name}?`,
    buttons: [
      { text: 'Cancel', role: 'cancel' },
      {
        text: 'Delete Party',
        handler: () => this.removeParty(),
      },
    ],
  });
  await alert.present();

  await alert.onDidDismiss();
}
```

Now we can go ahead and test this, go to the `detail-party.component.html` file, and create a floating action button that calls this `removePartyAlert()` function:

```
<ion-fab vertical="bottom" horizontal="end" slot="fixed">
  <ion-fab-button color="danger" (click)="removePartyAlert()">
    <ion-icon name="remove"></ion-icon>
  </ion-fab-button>
</ion-fab>
```

And that's it, you can click that button and see your alert in action, and once you confirm it, it will delete the item and take you back to the list page.



Event's Profits: $787,876.00

Ticket: **$121212**
Date: **Sep 30, 2021**

Tickets Sold: 24

Are you sure you want to delete Jorge?

Figure 16: Delete Party Confirmation Alert

What’s next?

First of all, congratulations!

This wasn’t a long book, but since Firebase’s modular SDK is so new, most concepts here are probably new to you.

Seriously, you went from 0 to installing and initializing Firebase in your Ionic project, handling user authentication, and full database interactivity.

Please, do something nice for you to celebrate :)

The next thing on your mind should be practice.

And the best way to practice is implementing all of this in a product of your own.

So please, once you’re done with this, shoot me an email. I’m always available at jorge@jsmobiledev.com, and let me know what you learned from this or if you feel it’s missing something. I’m working on adding more content about Firebase, so feedback is always appreciated.