Published in Better Programming

Estefanía García Gallardo Follow

Dec 29, 2021 · 12 min read ★ · Listen

Angular 13 Firebase Authentication Tutorial With AngularFire 7

Implementing social and traditional login strategies with Angular and Firebase

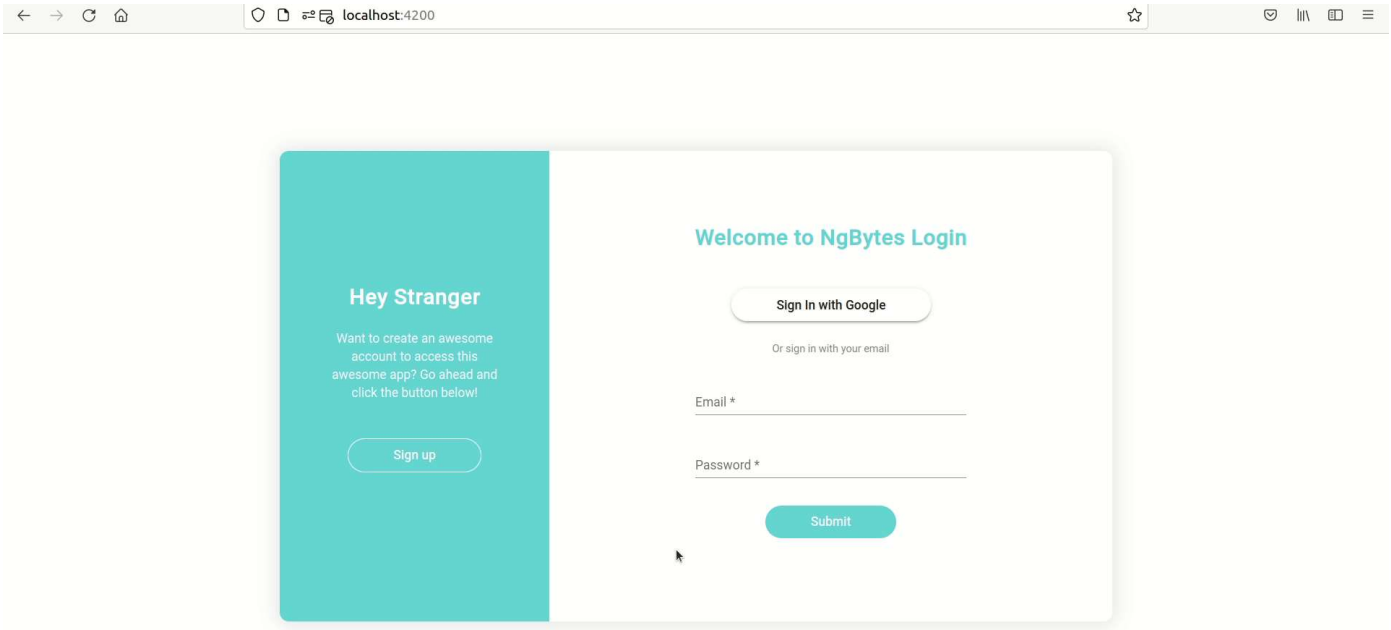
Introduction

Authentication is key to protecting our apps from unregistered users. In this tutorial, you will learn how to implement both a traditional email/password login and a Google social login in Firebase, with the new tree-shakable AngularFire v.7.0 modular SDK, which allows us to take full advantage of the new tree-shakable Firebase JS SDK (v9).

You'll also learn how to protect routes with the AngularFire Auth Guard, and how to redirect users to certain pages based on whether they're authenticated or not.

> *Please note that, at the time of writing this tutorial, the Angular fire v.7.0 API is still in development, and isn't feature complete; and the docs aren't yet updated to the latest version. AngularFire provides a compatibility layer that will allow you to use the latest version of the library, while fully supporting the AngularFire v6.0 API. This said, if you're as curious as I am, and enjoy trying out the latest versions of all your libraries, go ahead and enjoy this tutorial.*

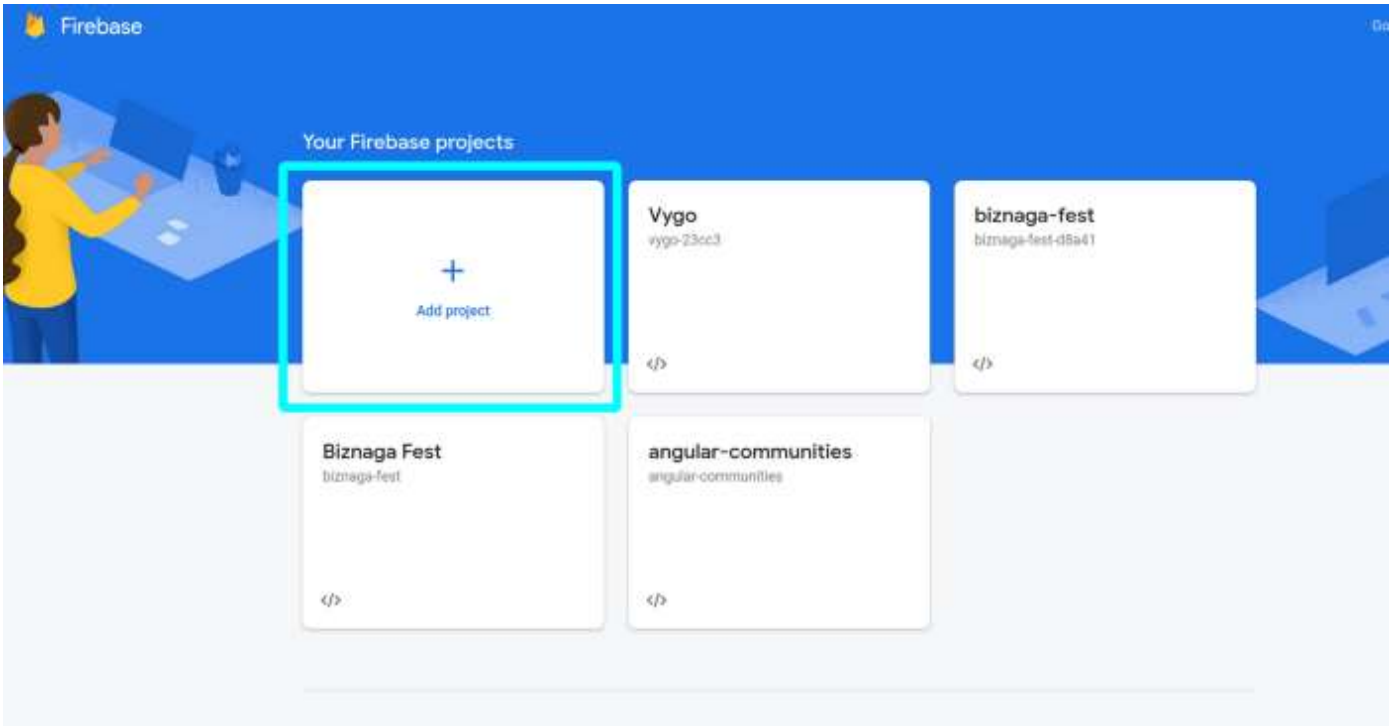
Here's a preview of the app we'll be building today:



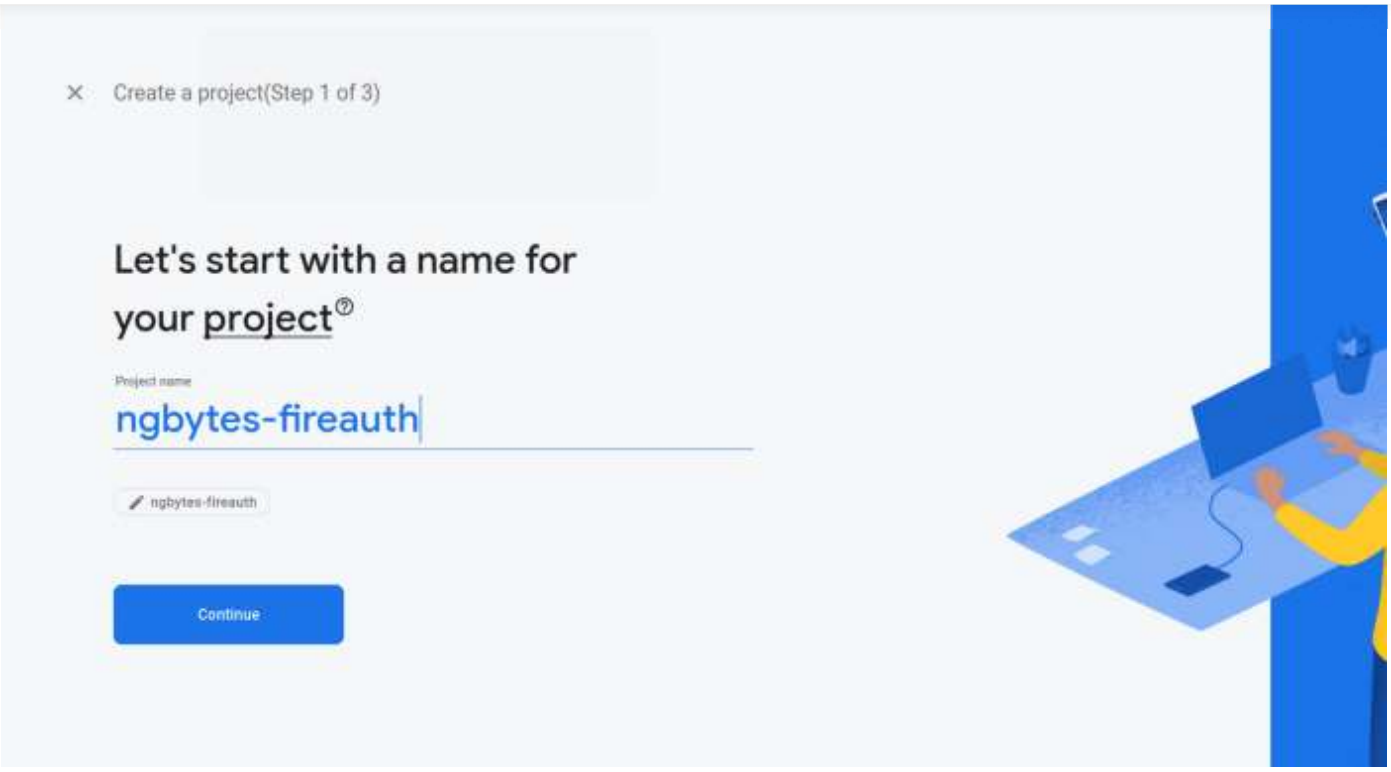
- Setting up an Angular project with AngularFire.
- Building the App.
- Creating the dashboard.
- Creating the auth module.
- Adding Google sign-in.
- Protecting the routes.

Setting up Firebase project

To create a new Firebase project, we need to log in to the Firebase console. Once we've done that, we can click on Add Project to create a new project:

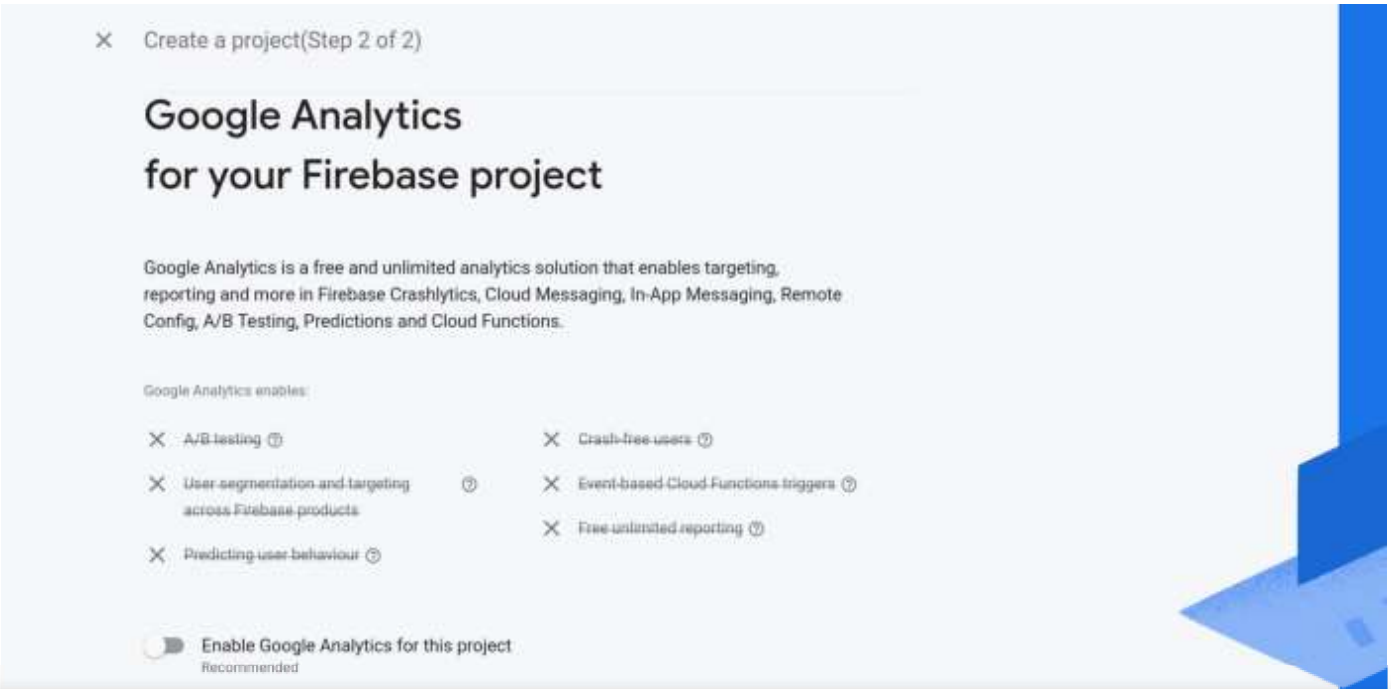


Firebase — Creating a new project



Firebase — Choosing a name

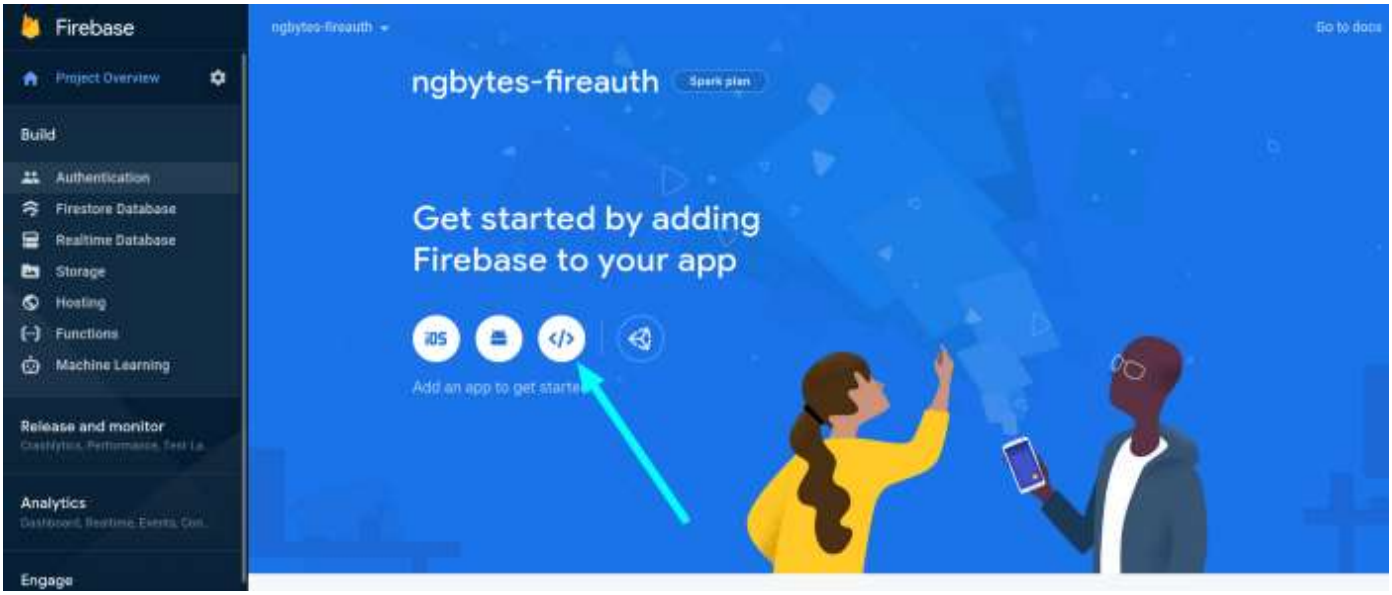
Decide whether or not we want Google Analytics in our project (your choice, I'll go with no):



We're done! We now have a lovely new Firebase project.

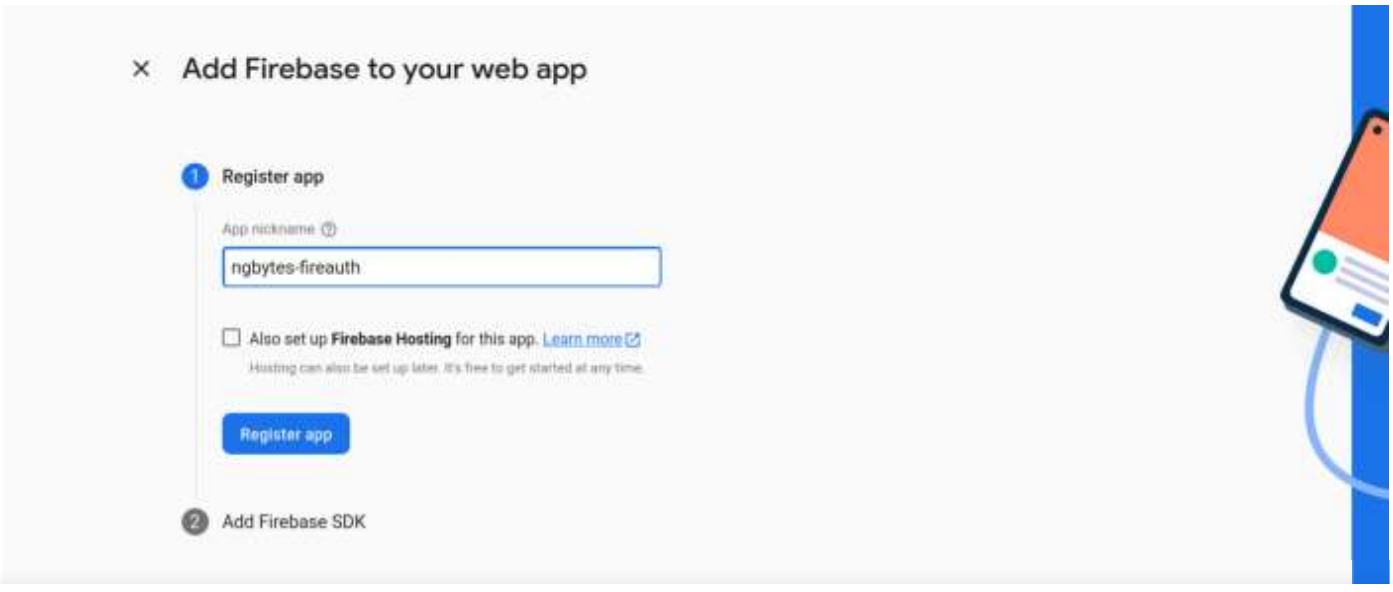
Adding Firebase to a Web app

Now we need to add Firebase to our Angular app. To do so, we need to click on the Web app button, in the dashboard:

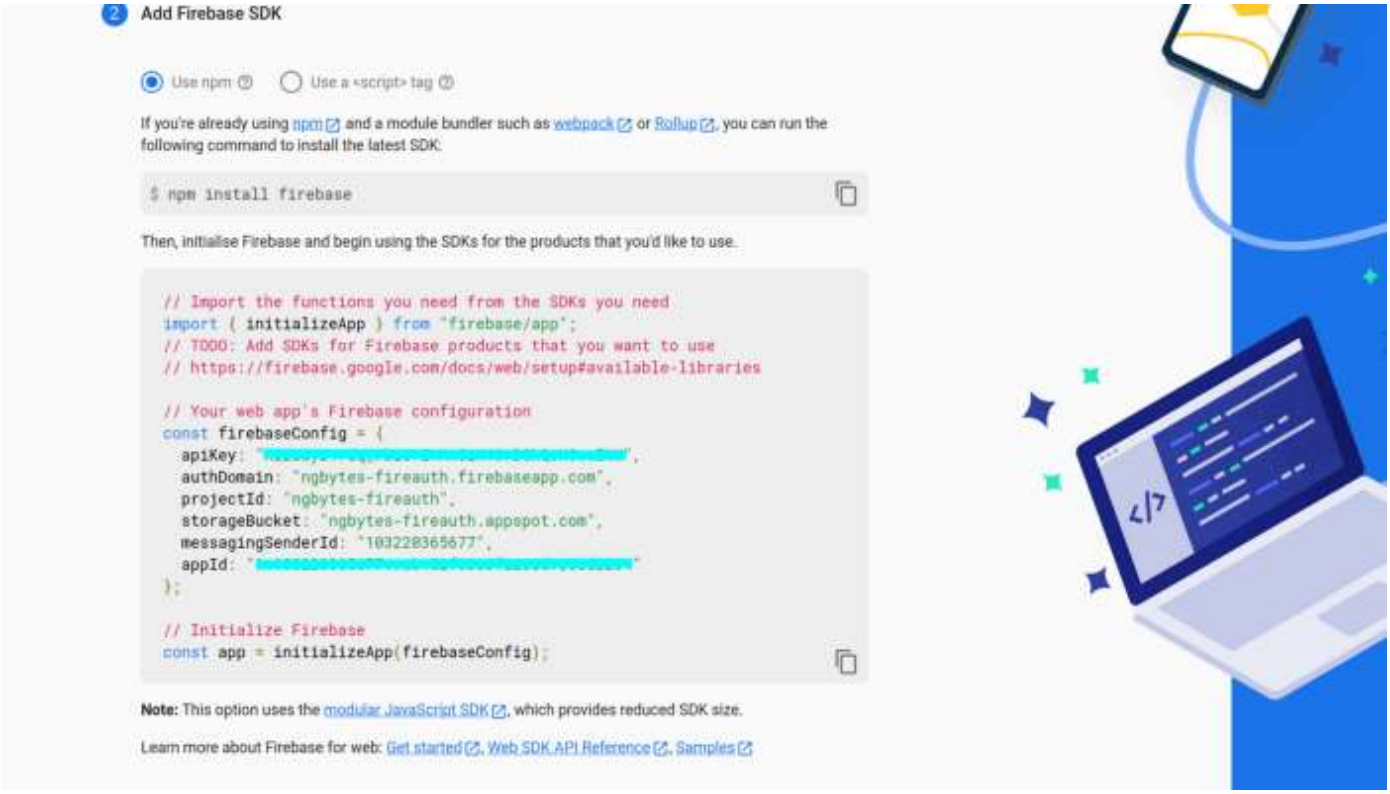


Firebase — Adding a web app

We then need to choose a name:

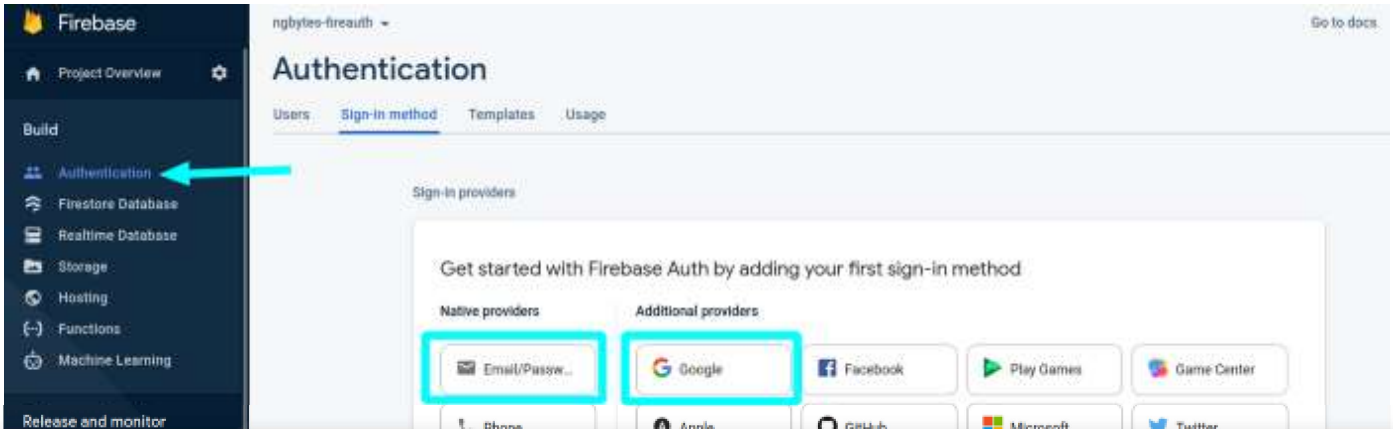


Finally, we now have the configuration that will be used later on to add Firebase to our Angular application:

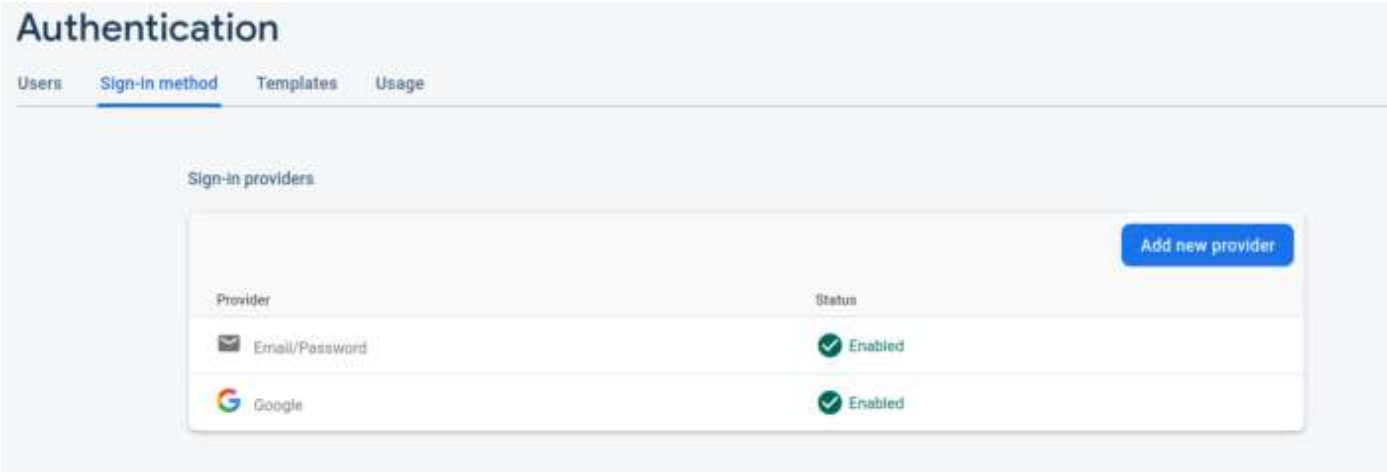


Adding Authentication

It's time to add authentication and choose the sign-in methods that we want to implement. In this tutorial, we'll add a Google social login, and a traditional email/password login. Head over to the Authentication page, and choose the providers:



Once we've enabled both sign-in methods, our Authentication dashboard should look like this:



Firebase — Enabled sign-in methods

That's all! We now have our Firebase app fully set up and ready. It's time to get started on our Angular app.

Setting up an Angular project

The first thing we're going to do is create a new project with the Angular CLI.

Tip: If you haven't installed the Angular CLI, you can do so by running the following command:

```
npm i -g @angular/cli
```

To create a new Angular project, we can run the following command:

```
ng new ngbytes-fireauth
```

Once the CLI has worked its magic, we can open the newly created project with our favorite IDE (I suggest VSCode, which is the one I normally use).

Adding Firebase and AngularFire

Let's add Firebase and Angularfire to our project. To do so, we'll use the AngularFire schematic, that will take care of setting everything up for us. Let's run the following command:

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

We'll be asked a series of questions, like which Firebase features we'd like to setup. For this tutorial, we only need to add authentication, so let's select that:

We'll then be asked about the Firebase account we'd like to use, and which project we want to setup. Select the project we created previously, and then select the app we also created earlier.

Once we've done all this, you will see that the schematic has taken care of all the Firebase configuration for us. Awesome!

Adding Angular Material

We'll also add Angular Material. Once again, we'll use a schematic:

Disabling strictPropertyInitialization

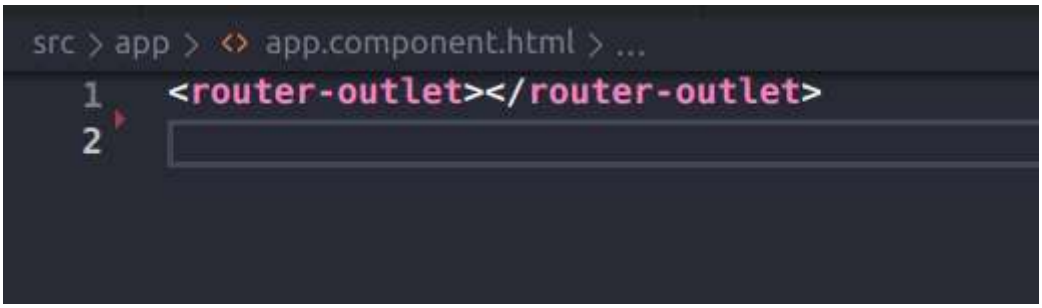
We'll need to set the `strictPropertyInitialization` property in the `tsconfig.json` file to false. We're doing this because the strict mode is enabled by default in all new Angular apps starting from version 12, which means that TypeScript will complain if we declare any class properties without setting them in the constructor (a common practice in Angular). Here's what our `tsconfig.json` file should look like:

```
/* To learn more about this file see: https://angular.io/config/tsconfig. */
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "sourceMap": true,
    "strictPropertyInitialization": false,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2017",
    "module": "es2020",
    "lib": [
      "es2018",
      "dom"
    ]
  },
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false,
    "strictInjectionParameters": true,
    "strictInputAccessModifiers": true,
    "strictTemplates": true
  }
```


Deleting the Angular boilerplate

Last, but not least, we'll delete the boilerplate code that Angular automatically generates in the `app.component.html`. Be very careful and make sure that you do not delete the `<router-outlet></router-outlet>` tags when you delete the boilerplate code, or the router won't work.

After deleting everything, your `app.component.html` should only contain the router-outlet tags:



router-outlet tags in the app.component.html

Creating the dashboard

It's time to start building our Angular app. Let's begin by creating a `dashboard` module. This is the module that will contain the pages which we want to protect from unauthenticated users. Let's begin by creating a `features` directory inside the `src/app` directory, which will contain all of our feature modules:

```
mkdir features
```

Inside this new directory, we'll create our `dashboard` module with the Angular CLI:

Tip: We're using the `--route` option, which creates a component in the new module, and adds the route to that component in the `Routes` array declared in the module provided in the `-m` option.

You'll see that the CLI has created a `dashboard` module inside the `features` directory, with its corresponding `dashboard-routing.module.ts` and component. It's also modified the `app-routing.module.ts`, and added a `dashboard` route, lazy loading the `DashboardModule`.

Let's add some information to our the `dashboard` component, by modifying the `dashboard.component.html` file:

```
<mat-toolbar></mat-toolbar>
<div class="background">
  <h1>Welcome to the NgBytes Dashboard</h1>
  <h2>Built with &lt;3 by Dottech</h2>

  <a target="_blank" class="btn" href="https://github.com/puntotech/ngbytes-fireauth">
    Click here to view the source code
  </a>
</div>
```

dashboard.component.html hosted with ❤ by GitHub view raw

The dashboard component template

Feel free to add whatever you want to this file.

Tip: Don't forget to import the `MatToolbarModule` to the `DashboardModule`, if you decide to use it too.

Let's add a little CSS, to make the dashboard look pretty:

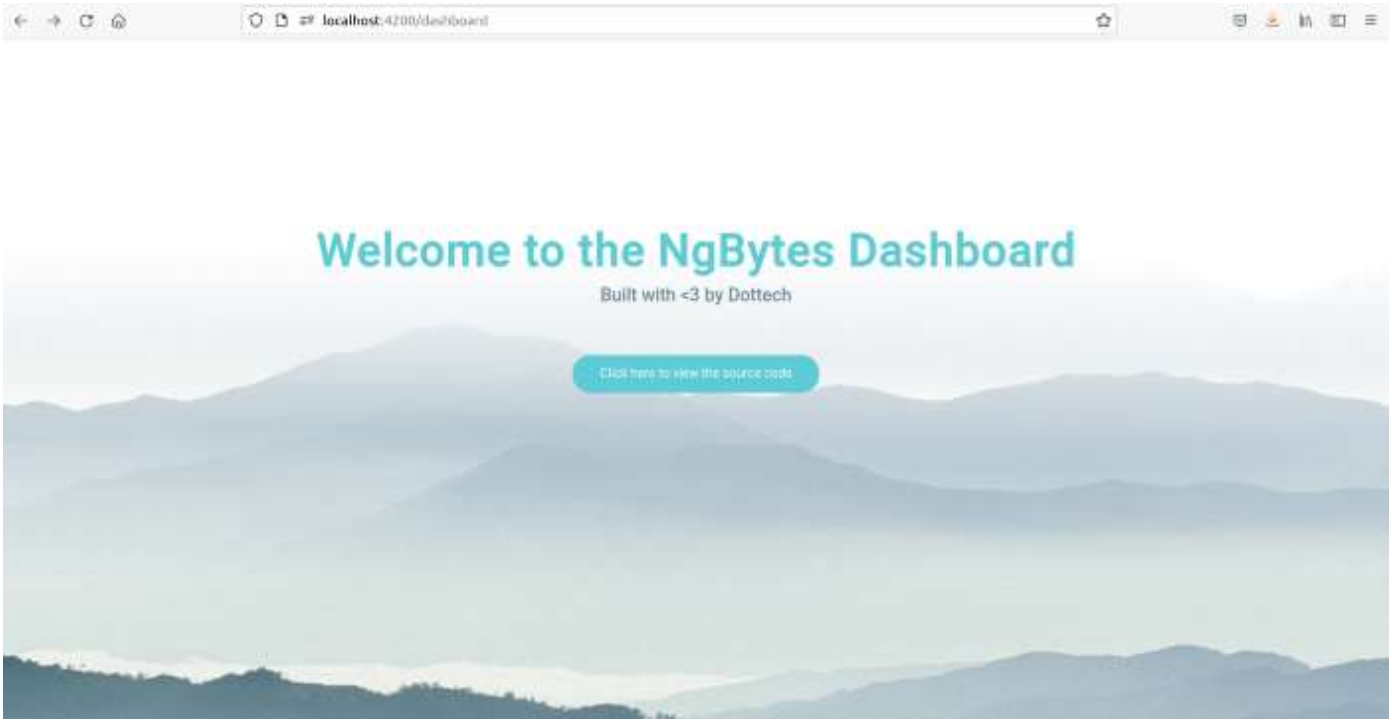
```
h1 {
  color: #5eccd6;
  font-size: 50px;
```

```
  margin-bottom: 50px;
}

a {
  color: #5eccd6;
  border-bottom: 2px solid transparent;
  transition: 0.5s border-bottom;
}

mat-toolbar {
  background-color: transparent;
  display: flex;
  justify-content: right;
}

.background {
  align-items: center;
  background-image: url("/assets/mountains.png");
  box-sizing: border-box;
  display: flex;
  flex-direction: column;
  height: calc(100vh - 64px);
  padding-top: 10%;
}

.btn {
  background-color: #5eccd6;
  border-radius: 20px;
  color: #fff;
  padding: 10px 30px;
  text-decoration: none;
  transition: 0.5s background-color;
}

a:hover {
  border-bottom: 2px solid #5eccd6;
  cursor: pointer;
}

.btn:hover {
```


The dashboard component CSS

Et voilà, our `dashboard` module is ready! To see it, we can navigate to `http://localhost:4200/dashboard` :



Dashboard preview

It's time to get started on our authentication.

Creating the Auth Module

Inside the `features` directory, we'll use the Angular CLI to create an `auth` module:

```
ng g m auth --routing
```

Tip: We're using the `--routing` flag, which generates an `auth-routing.module.ts` as well as

Once we've created our module, we need to add its corresponding route to the `app-routing.module.ts` file.

```
1  import { RouterModule, Routes } from '@angular/router';
2
3  import { NgModule } from '@angular/core';
4
5  const routes: Routes = [
6    {
7      path: '',
8      loadChildren: () =>
9        import('./features/auth/auth.module').then((m) => m.AuthModule),
10   },
11   {
12     path: 'dashboard',
13     loadChildren: () =>
14       import('./features/dashboard/dashboard.module').then(
15         (m) => m.DashboardModule
16       ),
17   },
18   {
19     path: '**',
20     redirectTo: '',
21     pathMatch: 'full',
22   },
23 ];
24
25 @NgModule({
26   imports: [RouterModule.forRoot(routes)],
27   exports: [RouterModule],
28 })
29 export class AppRoutingModule {}
```

app-routing.module.ts hosted with ❤ by GitHub view raw

Adding routes the app-routing module

Note that we're adding a wildcard route, that redirects to the default route. We do this so


All of our interactions with Firebase and AngularFire will take place within the `AuthService`. To create our service, we'll once again make use of the Angular CLI. Inside our `app` directory:

```
mkdir core
cd core
mkdir services
cd services
ng g s auth
```

Inside our `authService`, the first thing we'll need to do is inject the AngularFire `Auth` instance in the constructor. Then, we'll create a `login` method, which will receive the login data (email and password), and use the `signInWithEmailAndPassword` AngularFire method to sign in:

```
1  import { Auth, signInWithEmailAndPassword } from '@angular/fire/auth';
2
3  import { Injectable } from '@angular/core';
4
5  @Injectable({
6    providedIn: 'root',
7  })
8  export class AuthService {
9    constructor(private auth: Auth) {}
10
11   login({ email, password }: any) {
12     return signInWithEmailAndPassword(this.auth, email, password);
13   }
14 }
```

auth.service.ts hosted with ❤ by GitHub view raw

AuthService with untyped login method

However, doesn't something about our `login` function seem a little off? We have a very

```
mkdir interfaces
cd interfaces
touch login-data.interface.ts
```

Here's what out interface will look like:

```
export interface LoginData {
  email: string;
  password: string;
}
```

login-data.interface.ts hosted with ❤ by GitHub view raw

The LoginData interface https://gist.github.com/NyaGarcia/2ec63c5fe25c1d281fcdb011722e9712

Now we can use our new interface to properly type our `login` function's parameters:

```
import { Auth, signInWithEmailAndPassword } from '@angular/fire/auth';

import { Injectable } from '@angular/core';
import { LoginData } from '../interfaces/login-data.interface';

@Injectable({
  providedIn: 'root',
})
export class AuthService {
  constructor(private auth: Auth) {}

  login({ email, password }: LoginData) {
    return signInWithEmailAndPassword(this.auth, email, password);
  }
}
```

auth.service.ts hosted with ❤ by GitHub view raw

AuthService with typed login function

Looking much better, isn't it?

We'll also create a `logout` method, which, as you can probably guess, will allow our users to log out of our application:

```
1  import {
2    Auth,
3    createUserWithEmailAndPassword,
4    signInWithEmailAndPassword,
5    signOut,
6  } from '@angular/fire/auth';
7
8  import { Injectable } from '@angular/core';
9  import { LoginData } from '../interfaces/login-data.interface';
10
11 @Injectable({
12   providedIn: 'root',
13 })
14 export class AuthService {
15   constructor(private auth: Auth) {}
16
17   login({ email, password }: LoginData) {
18     return signInWithEmailAndPassword(this.auth, email, password);
19   }
20
21   register({ email, password }: LoginData) {
22     return createUserWithEmailAndPassword(this.auth, email, password);
23   }
24
25   logout() {
26     return signOut(this.auth);
27   }
28 }
```

auth.service.ts hosted with ❤ by GitHub view raw

Adding the register and logout functions to the AuthService

That's all we have to do in our `AuthService` (we'll be adding the Google social login option later on) for now.

Inside the `auth` directory, we'll create a `login-page` component, which will display a welcome message, the login form so that users can sign in, and later on, the google login button:

```
mkdir pages
cd pages
ng g c login-page
```

Once we've created our page, let's head over to the `login-page.component.html` file and add a simple title:

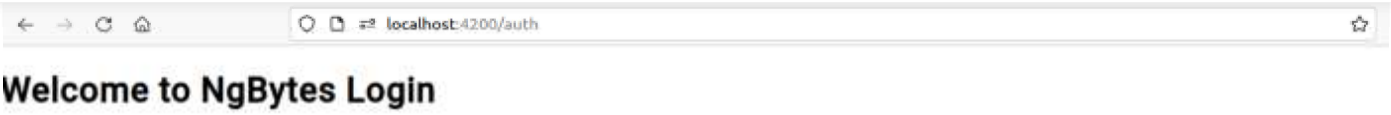
```
<h1>Welcome to NgBytes Login</h1>
```

Since pages are routed components, we need to add a route to our `auth-routing.module.ts` file that points to the `LoginPageComponent`, like so:

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

import { LoginPageComponent } from './pages/login-page/login-page.component';
import { NgModule } from '@angular/core';

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

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

auth-routing.module.ts hosted with ❤ by GitHub view raw

Routing the LoginPageComponent in the AuthRoutingModule

LoginPageComponent preview

Awesome! It's time to create a login form, so that our users can sign into our app:

Creating the login form

Now we'll create a `login-form` component, inside our `auth` directory:

```
mkdir components
cd components
ng g c login-form
```

Tip: Since we won't be writing any tests in this tutorial, feel free to delete the `login-form.component.spec.ts` file.

To create our form, we'll be using the Reactive Forms module. We'll also be using the Angular Material modules, to style our form. Since it isn't the goal of this tutorial, I won't go into any details about how to use reactive forms.

First, we'll import the `ReactiveFormsModule` , the `MatInputModule` , the `MatFormFieldModule` and the `MatButtonModule` modules into our `AuthModule` :

```
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { NgModule } from '@angular/core';
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  declarations: [LoginPageComponent, LoginFormComponent],
  imports: [
    CommonModule,
    AuthRoutingModule,
    ReactiveFormsModule,
    MatInputModule,
    MatFormFieldModule,
    MatButtonModule,
  ],
})
export class AuthModule {}
```

auth.module.ts hosted with ❤ by GitHub view raw

Importing modules into the AuthModule

Then, we'll create the form in our `login-form.component.ts` file:

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

@Component({
  selector: 'ngbytes-login-form',
  templateUrl: './login-form.component.html',
  styleUrls: ['./login-form.component.css'],
})
export class LoginFormComponent implements OnInit {
  @Output() formData: EventEmitter<{
    email: string;
    password: string;
  }> = new EventEmitter();

  form: FormGroup;
```


```
21        email: ['', [Validators.required, Validators.email]],
22        password: ['', Validators.required],
23      });
24    }
25
26    get email() {
27      return this.form.get('email');
28    }
29
30    get password() {
31      return this.form.get('password');
32    }
33
34    onSubmit() {
35      this.formData.emit(this.form.value);
36    }
37  }
```

login-form.component.ts hosted with ❤ by GitHub view raw

The LoginFormComponent logic

As you can see, we've added basic validation to our form: both the email and password fields are required, and the email field only allows valid email addresses.

Tip: Find it odd that we're using an `EventEmitter` *to emit our form's data upon submission? This is because we're using the smart/dumb component pattern, which decouples business logic from purely presentational components.*

Lastly, in our `login-form.component.html` file:

```
1  <form class="form" [formGroup]="form" (ngSubmit)="onSubmit()">
2    <mat-form-field class="form-control">
3      <mat-label>Email</mat-label>
4      <input matInput formControlName="email" type="text" required />
5      <mat-error *ngIf="email?.hasError('required')">
6        Email is required
7      </mat-error>
```


```
13      <mat-label>Password</mat-label>
14      <input matInput formControlName="password" type="password" required />
15      <mat-error *ngIf="password?.hasError('required')">
16        Password is required
17      </mat-error>
18    </mat-form-field>
19    <div class="form-footer">
20      <button [disabled]="form.invalid" mat-raised-button type="submit">
21        Submit
22      </button>
23    </div>
24  </form>
```

login-form.component.html hosted with ❤ by GitHub view raw

The LoginFormComponent template

Note: See how we're disabling the submit button if the form isn't valid? That's a bad practice (poor UX), and the only reason I'm doing it is for the sake of brevity, since this tutorial should be focused on AngularFire and not on best practices in Angular forms. However, be warned, don't do this at home folks!

Let's add a little CSS in our `login-form.component.css` file to make the form prettier (you can skip this step if you wish):

```
1   button {
2     background-color: #5eccd6;
3     border-radius: 20px;
4     color: #fff;
5     padding: 0 50px;
6   }
7
8   form {
9     align-items: center;
10    display: flex;
11    flex-direction: column;
12    justify-content: space-evenly;
13    height: 200px;
```

```
19  }
```

login-form.component.css hosted with ❤ by GitHub view raw

The LoginFormComponent CSS

Great! We now have a lovely login form. All that's left is to add it to to the `login-page.component.html` template:

```
1  <div class="form-container">
2        <h1>Welcome to NgBytes Login</h1>
3        <h5>Sign in with your email</h5>
4        <ngbytes-login-form (formData)="login($event)"></ngbytes-login-form>
5  </div>
```

login-page.component.html hosted with ❤ by GitHub view raw

Adding the login form to the login page

We also need to inject the `authService` we created earlier into our `login-page.component.ts` file and create a `login` method, which will use the `authService` to log the user in, redirecting to the dashboard once it's done. Don't forget to catch any possible errors!

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { AuthService } from 'src/app/core/services/auth.service';
4  import { LoginData } from 'src/app/core/interfaces/login-data.interface';
5  import { Router } from '@angular/router';
6
7  @Component({
8    selector: 'app-login-page',
9    templateUrl: './login-page.component.html',
10   styleUrls: ['./login-page.component.css'],
11 })
12 export class LoginPageComponent implements OnInit {
13   constructor(
14     private readonly authService: AuthService,
```

```
20    login(loginData: LoginData) {
21      this.authService
22        .login(loginData)
23        .then(() => this.router.navigate(['/dashboard']))
24        .catch((e) => console.log(e.message));
25    }
26  }
```

login-page.component.ts hosted with ❤ by GitHub view raw

Creating a login method in the login-page.component

Let's test what we've done so far by going over to our browser, and filling in the form. As you can see, if we try to leave the email field blank, or to enter an invalid email, an error will be shown, which means our validation is working splendidly. However, if we try to press the submit button…

We're getting a User not found error in our console. Well, of course we are. We haven't registered any users yet, have we? Let's create a register form so that we can do so!

Creating a register page

Just like we did for our login page, we're going to create a register page, which will contain the register form. Inside the `auth/pages` directory:

```
ng g c register-page
```

Let's add a title to our register page, and some styles:

```
1  <div class="container">
2    <h1>Welcome to NgBytes Register</h1>
3  </div>
```

register-page.component.html hosted with ❤ by GitHub view raw

The RegisterPageComponent template


```
5
6   .container {
7     align-items: center;
8     display: flex;
9     flex-direction: column;
10    justify-content: center;
11    margin-top: 80px;
12  }
```

register-page.component.css hosted with ❤ by GitHub view raw

The RegisterPageComponent CSS https://gist.github.com/NyaGarcia/f6f3f5892f711fa31fa4acac1c886968

Last, but not least, we need to add a `register` route to the `auth-routing.module.ts` file:

```
1   import { RouterModule, Routes } from '@angular/router';
2
3   import { LoginPageComponent } from './pages/login-page/login-page.component';
4   import { NgModule } from '@angular/core';
5   import { RegisterPageComponent } from './pages/register-page/register-page.component';
6
7   const routes: Routes = [
8     { path: '', component: LoginPageComponent },
9     { path: 'register', component: RegisterPageComponent },
10  ];
11
12  @NgModule({
13    imports: [RouterModule.forChild(routes)],
14    exports: [RouterModule],
15  })
16  export class AuthRoutingModule {}
```

auth-routing.module.ts hosted with ❤ by GitHub view raw

Adding the RegisterPageComponent route

Since our register page is done, all that we need to do is create a register form… Or do we?

form is used to log in, the other to register a new user; two completely different pieces of logic.

However…

Remember how we used the smart/dumb component pattern to decouple the login form, from the login page? Our login form isn't in charge of performing any logic at all, our login page is. Which means we are free to reuse the login form in our register page:

```
<div class="container">
  <h1>Welcome to NgBytes Register</h1>
  <ngbytes-login-form (formData)="register($event)"></ngbytes-login-form>
</div>
```

register-page.component.html hosted with ❤ by GitHub view raw

Adding the login form to the register page

Neat, isn't it? Now comes the time to implement the logic to register a new user. We will do so, of course, in our `register-page.component.ts` file:

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

import { AuthService } from 'src/app/core/services/auth.service';
import { LoginData } from 'src/app/core/interfaces/login-data.interface';
import { Router } from '@angular/router';

@Component({
  selector: 'app-register-page',
  templateUrl: './register-page.component.html',
  styleUrls: ['./register-page.component.css'],
})
export class RegisterPageComponent implements OnInit {
  constructor(
    private readonly authService: AuthService,
    private readonly router: Router
  ) {}
```

```
22          .register(data)
23          .then(() => this.router.navigate(['/login']))
24          .catch((e) => console.log(e.message));
25      }
26  }
```

register-page.component.ts hosted with ❤ by GitHub view raw

Adding the register logic to the RegisterPageComponent

As you can see, we're calling the `authService.register` method, which will register the new user in our Firebase app, and then we're navigating to the login page, so that our newly registered user can log into our app.

Let's add some final touches to the login page: a button that will redirect new users to the register page, and change the layout a bit to make it look decent:

```
1   <div class="background">
2     <div class="login-container">
3       <div class="text-container">
4         <h1>Hey Stranger</h1>
5         <p>
6           Want to create an awesome account to access this awesome app? Go ahead
7           and click the button below!
8         </p>
9         <button mat-button routerLink="/register">Sign up</button>
10      </div>
11      <div class="form-container">
12        <h1>Welcome to NgBytes Login</h1>
13        <h5>Sign in with your email</h5>
14        <ngbytes-login-form (formData)="login($event)"></ngbytes-login-form>
15      </div>
16    </div>
17  </div>
```

login-page.component.html hosted with ❤ by GitHub view raw

Improving the login page


```
  margin-top: 30px;
  text-decoration: none;
}

h1 {
  color: #5eccd6;
  font-weight: bolder;
  margin-bottom: 20px;
}

h5 {
  color: #838383;
}

button {
  border-radius: 20px;
  padding: 0 50px;
}

p {
  margin-bottom: 20px;
  text-align: center;
}

.background {
  align-items: center;
  display: flex;
  height: 100%;
  justify-content: center;
}

.login-container {
  border-radius: 10px;
  box-shadow: 0 0 20px 2px #e4e4e4;
  display: flex;
  width: 60%;
}

.text-container {
  align-items: center;
```


```
    justify-content: center;
    padding: 50px;
    width: 30%;
  }

  .form-container {
    align-items: center;
    background-color: #fff;
    border-radius: 0 10px 10px 0;
    display: flex;
    flex-direction: column;
    justify-content: center;
    padding: 80px;
    width: 70%;
  }

  .text-container h1 {
    color: #fff;
  }

  .text-container button {
    background-color: transparent;
    border: 1px solid #fff;
  }
```

login-page.component.css hosted with ❤ by GitHub view raw

Adding styles to the login page

Since we've made our login page look pretty, let's take a moment and do the same for our register page:

```
<div class="background">
  <div class="login-container">
    <div class="form-container">
      <h1>Welcome to NgBytes Register</h1>
      <h5>Register with your email</h5>
      <ngbytes-login-form (formData)="register($event)"></ngbytes-login-form>
    </div>
```

```
        </p>
        <button mat-button routerLink="/">Sign in</button>
      </div>
    </div>
  </div>
```

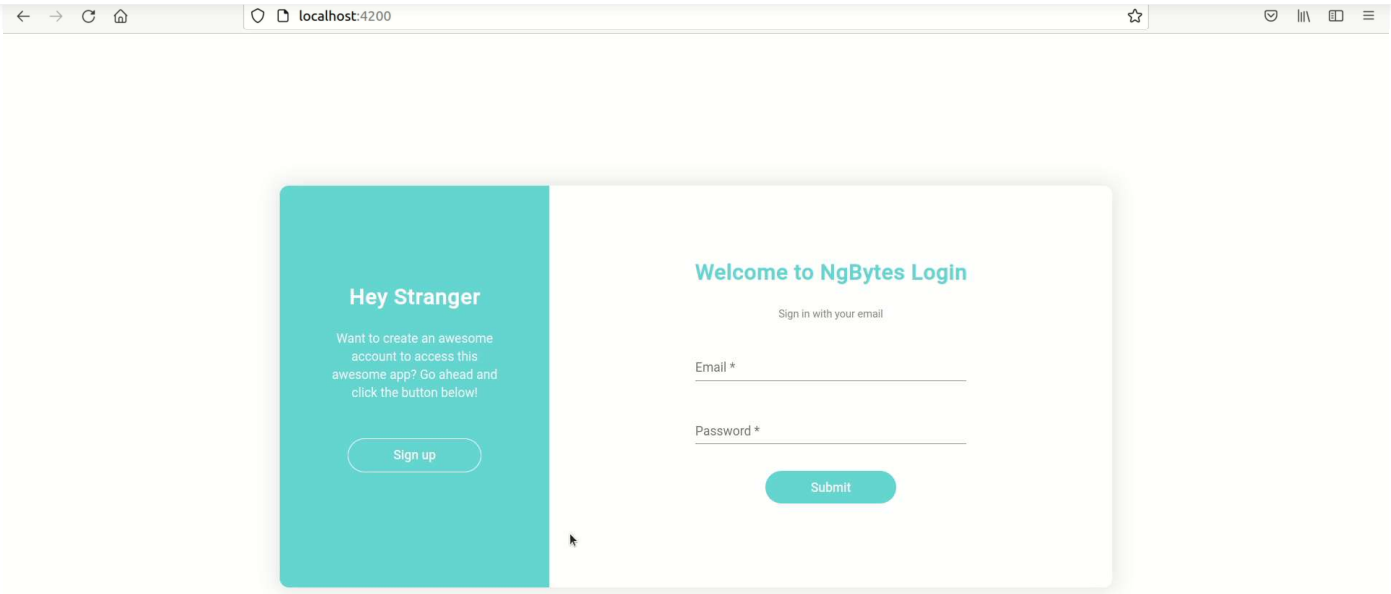
register-page.component.html hosted with ❤ by GitHub view raw

Improving the register page

Some more CSS:

```
a {
  color: #838383;
  margin-top: 30px;
  text-decoration: none;
}

h1 {
  color: #5eccd6;
  font-weight: bolder;
  margin-bottom: 20px;
}

h5 {
  color: #838383;
}

button {
  border-radius: 20px;
  padding: 0 50px;
}

p {
  margin-bottom: 20px;
  text-align: center;
}

.background {
```


```
33
34  .login-container {
35    border-radius: 10px;
36    box-shadow: 0 0 20px 2px #e4e4e4;
37    display: flex;
38    width: 60%;
39  }
40
41  .text-container {
42    align-items: center;
43    background-color: #5eccd6;
44    border-radius: 0 10px 10px 0;
45    color: #fff;
46    display: flex;
47    flex-direction: column;
48    justify-content: center;
49    padding: 50px;
50    width: 30%;
51  }
52
53  .form-container {
54    align-items: center;
55    background-color: #fff;
56    border-radius: 10px 0 0 10px;
57    display: flex;
58    flex-direction: column;
59    justify-content: center;
60    padding: 80px;
61    width: 70%;
62  }
63
64  .text-container h1 {
65    color: #fff;
66  }
67
68  .text-container button {
69    background-color: transparent;
70    border: 1px solid #fff;
71  }
```

Registering a new user and logging in

Awesome! We're logging into our app! However, we're still missing something, aren't we? We need a logout option! Let's add it to our dashboard.

Adding a logout option

Let's head over to our `dashboard.component.html` and create a logout option:

```
<mat-toolbar><a (click)="logout()">Log out</a></mat-toolbar>
<div class="background">
  <h1>Welcome to the NgBytes Dashboard</h1>
  <h2>Built with &lt;3 by Dottech</h2>

  <a
    target="_blank"
    class="btn"
    href="https://github.com/puntotech/ngbytes-fireauth"
  >
    Click here to view the source code
  </a>
```

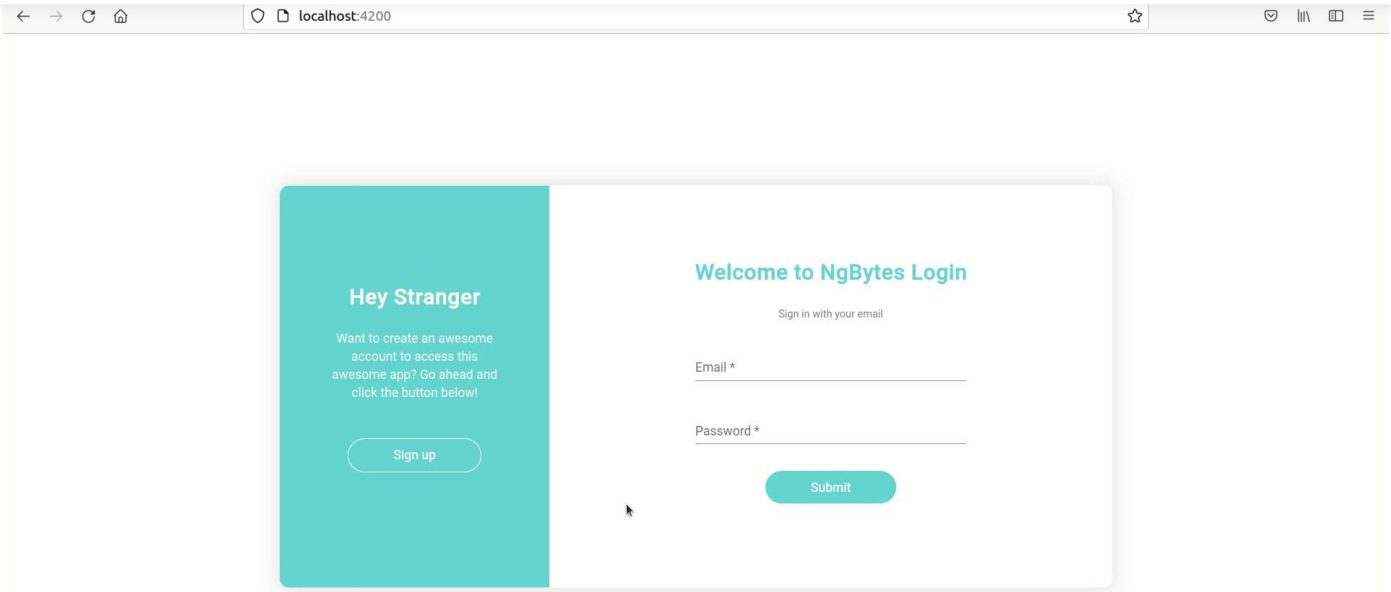
Now that we have our logout option, we need to add some logic to our `dashboard.component.ts`. First, we'll inject the `AuthService` in the constructor, and then we'll create a `logout` method:

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { AuthService } from '../../core/services/auth.service';
4  import { Router } from '@angular/router';
5
6  @Component({
7    selector: 'app-dashboard',
8    templateUrl: './dashboard.component.html',
9    styleUrls: ['./dashboard.component.css'],
10 })
11 export class DashboardComponent implements OnInit {
12   constructor(private authService: AuthService, private router: Router) {}
13
14   ngOnInit(): void {}
15
16   logout() {
17     this.authService
18       .logout()
19       .then(() => this.router.navigate(['/']))
20       .catch((e) => console.log(e.message));
21   }
22 }
```

dashboard.component.ts hosted with ❤ by GitHub view raw

Adding a logout method to the dashboard component

As you can see, once we call the `logout` method, we'll redirect the user to the default route. That's it! We've successfully implemented a way for our users to log out of our app. Once again, let's test our app:



Logging out

Adding Google Sign In

Now that our traditional login has been fully implemented, it's time to add our Google social login. To do so, we'll first create the necessary method in the `auth.service`:

```
1  import {
2    Auth,
3    GoogleAuthProvider,
4    createUserWithEmailAndPassword,
5    signInWithEmailAndPassword,
6    signInWithPopup,
7    signOut,
8  } from '@angular/fire/auth';
9
10 import { Injectable } from '@angular/core';
11 import { LoginData } from '../interfaces/login-data.interface';
```

```
18
19    login({ email, password }: LoginData) {
20      return signInWithEmailAndPassword(this.auth, email, password);
21    }
22
23    loginWithGoogle() {
24      return signInWithPopup(this.auth, new GoogleAuthProvider());
25    }
26
27    register({ email, password }: LoginData) {
28      return createUserWithEmailAndPassword(this.auth, email, password);
29    }
30
31    logout() {
32      return signOut(this.auth);
33    }
34  }
```

auth.service.ts hosted with ❤ by GitHub view raw

Adding a loginWithGoogle method to the AuthService

Then, we'll need to add a Google sign in button to the login page:

```
1   <div class="background">
2     <div class="login-container">
3       <div class="text-container">
4         <h1>Hey Stranger</h1>
5         <p>
6           Want to create an awesome account to access this awesome app? Go ahead
7           and click the button below!
8         </p>
9         <button mat-button routerLink="/register">Sign up</button>
10      </div>
11      <div class="form-container">
12        <h1>Welcome to NgBytes Login</h1>
13        <button class="mat-raised-button" (click)="loginWithGoogle()">
14          Sign In with Google
15        </button>
```

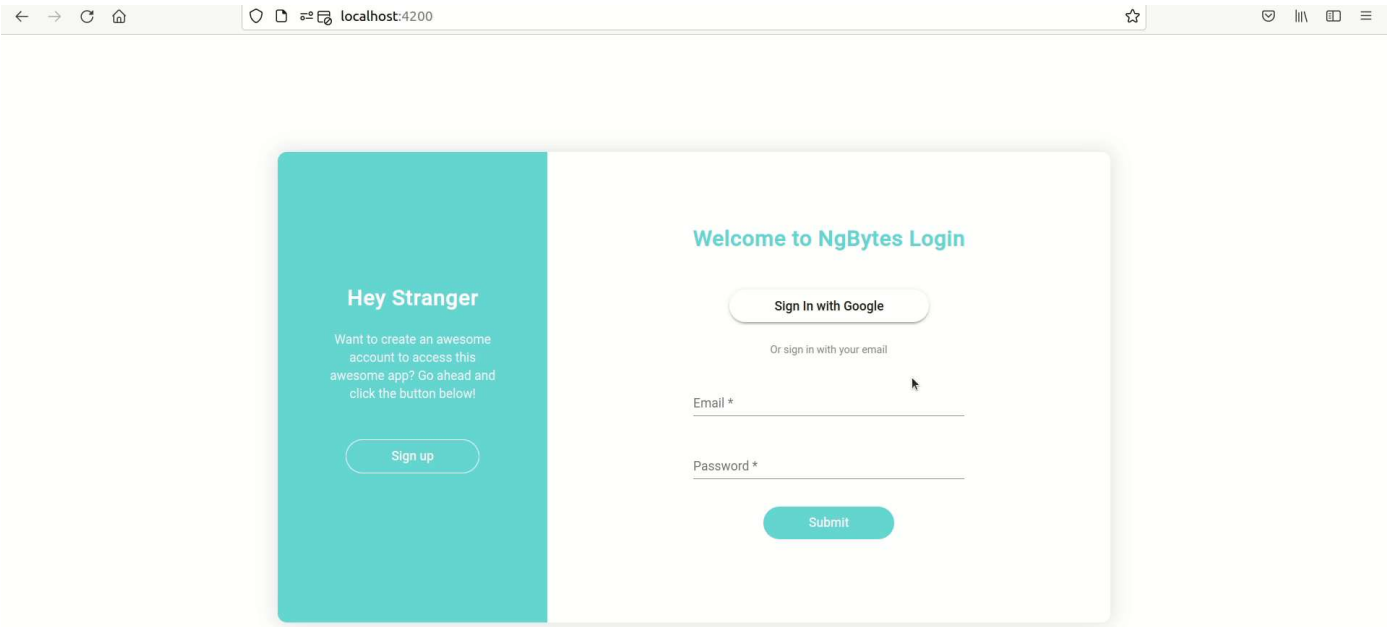

login-page.component.html hosted with ❤ by GitHub view raw

Adding a Google sign in button

Finally, we'll need to add a `loginWithGoogle` function to our `login-page.component.ts`:

```
1  import { Component, OnInit } from '@angular/core';
2
3  import { AuthService } from 'src/app/core/services/auth.service';
4  import { LoginData } from 'src/app/core/interfaces/login-data.interface';
5  import { Router } from '@angular/router';
6
7  @Component({
8    selector: 'app-login-page',
9    templateUrl: './login-page.component.html',
10   styleUrls: ['./login-page.component.css'],
11 })
12 export class LoginPageComponent implements OnInit {
13   constructor(
14     private readonly authService: AuthService,
15     private readonly router: Router
16   ) {}
17
18   ngOnInit(): void {}
19
20   login(loginData: LoginData) {
21     this.authService
22       .login(loginData)
23       .then(() => this.router.navigate(['/dashboard']))
24       .catch((e) => console.log(e.message));
25   }
26
27   loginWithGoogle() {
28     this.authService
29       .loginWithGoogle()
30       .then(() => this.router.navigate(['/dashboard']))
31       .catch((e) => console.log(e.message));
32   }
```

We're done! Let's head over to our browser and try it out:



Signing in with google

Protecting the routes

If you attempt to navigate to the dashboard page `localhost:4200/dashboard`, you'll see that you'll be able to do so, even if you're unauthenticated. To solve this problem, we'll use the auth guard provided by AngularFire, to automatically redirect unauthorized users to the login page. In our `app-routing.module.ts`:

```
1  import { AuthGuard, redirectUnauthorizedTo } from '@angular/fire/auth-guard';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { NgModule } from '@angular/core';
5
6  const redirectUnauthorizedToLogin = () => redirectUnauthorizedTo(['']);
```

```
12          import('./features/auth/auth.module').then((m) => m.AuthModule),
13      },
14      {
15        path: 'dashboard',
16        loadChildren: () =>
17          import('./features/dashboard/dashboard.module').then(
18            (m) => m.DashboardModule
19          ),
20        canActivate: [AuthGuard],
21        data: { authGuardPipe: redirectUnauthorizedToLogin },
22      },
23      {
24        path: '**',
25        redirectTo: '',
26        pathMatch: 'full',
27      },
28    ];
29
30    @NgModule({
31      imports: [RouterModule.forRoot(routes)],
32      exports: [RouterModule],
33    })
34    export class AppRoutingModule {}
```

app-routing.module.ts hosted with ❤ by GitHub view raw

Protecting the routes

As you can see, we've protected the `dashboard` route with the AngularFire `AuthGuard`, and we've built a `redirectUnauthorizedToLogin` pipe, using the AngularFire `redirectUnauthorizedTo` pipe, which we then provide to the `AuthGuard`. This will ensure that any unauthenticated user that tries to access the dashboard page, will automatically be redirected to the login page. Feel free to head over to your browser and try it out.

The same way that we don't want unauthorized users to access the dashboard, if the user is already authenticated, they shouldn't be able to navigate to the login page. Let's create another custom pipe, one that will redirect the already logged in users to the dashboard:

```
5  } from '@angular/fire/auth-guard';
6  import { RouterModule, Routes } from '@angular/router';
7
8  import { NgModule } from '@angular/core';
9
10 const redirectUnauthorizedToLogin = () => redirectUnauthorizedTo(['']);
11 const redirectLoggedInToHome = () => redirectLoggedInTo(['dashboard']);
12
13 const routes: Routes = [
14   {
15     path: '',
16     loadChildren: () =>
17       import('./features/auth/auth.module').then((m) => m.AuthModule),
18     canActivate: [AuthGuard],
19     data: { authGuardPipe: redirectLoggedInToHome },
20   },
21   {
22     path: 'dashboard',
23     loadChildren: () =>
24       import('./features/dashboard/dashboard.module').then(
25         (m) => m.DashboardModule
26       ),
27     canActivate: [AuthGuard],
28     data: { authGuardPipe: redirectUnauthorizedToLogin },
29   },
30   {
31     path: '**',
32     redirectTo: '',
33     pathMatch: 'full',
34   },
35 ];
36
37 @NgModule({
38   imports: [RouterModule.forRoot(routes)],
39   exports: [RouterModule],
40 })
41 export class AppRoutingModule {}
```

app-routing.module.ts hosted with ❤ by GitHub view raw

Adding the redirectLoggedInToHome route guard

The canActivate helper

AngularFire provides us with a `canActivate` helper, which we can use together with the spread operator, to make our routes more readable:

```
import { RouterModule, Routes } from '@angular/router';
import {
  canActivate,
  redirectLoggedInTo,
  redirectUnauthorizedTo,
} from '@angular/fire/auth-guard';

import { NgModule } from '@angular/core';

const redirectUnauthorizedToLogin = () => redirectUnauthorizedTo(['']);
const redirectLoggedInToHome = () => redirectLoggedInTo(['dashboard']);

const routes: Routes = [
  {
    path: '',
    loadChildren: () =>
      import('./features/auth/auth.module').then((m) => m.AuthModule),
    ...canActivate(redirectLoggedInToHome),
  },
  {
    path: 'dashboard',
    loadChildren: () =>
      import('./features/dashboard/dashboard.module').then(
        (m) => m.DashboardModule
      ),
    ...canActivate(redirectUnauthorizedToLogin),
  },
  {
    path: '**',
    redirectTo: '',
    pathMatch: 'full',
  },
];

```


app-routing.module.ts hosted with ❤ by GitHub view raw

Using the canActivate helper to make the routes more readable

Conclusion

That's all folks! Congratulations, you've successfully created an authentication application with Angular 13 and the latest AngularFire 7 modular SDK.

Please remember that the new API isn't complete, so if you decide to go ahead anyway and use it in production, do so at your own risk. Also, bear in mind that AngularFire 7 provides a compatibility layer so that you can continue using the previous AngularFire 6 API.

I hope you found this tutorial useful. Thank you very much for reading.