Los prerrequisitos son tener instalado NodeJS y el gestor npm. Dependiendo de la versión de angular requiere una u otra versión de Node (<https://angular.io/guide/setup-local> ).

Comenzamos instalando el cliente de Angular que nos permite por consola crear proyectos, componentes, servicios y otra serie de funcionalidades que nos facilitan la tarea de desarrollo.

npm install -g @angular/cli

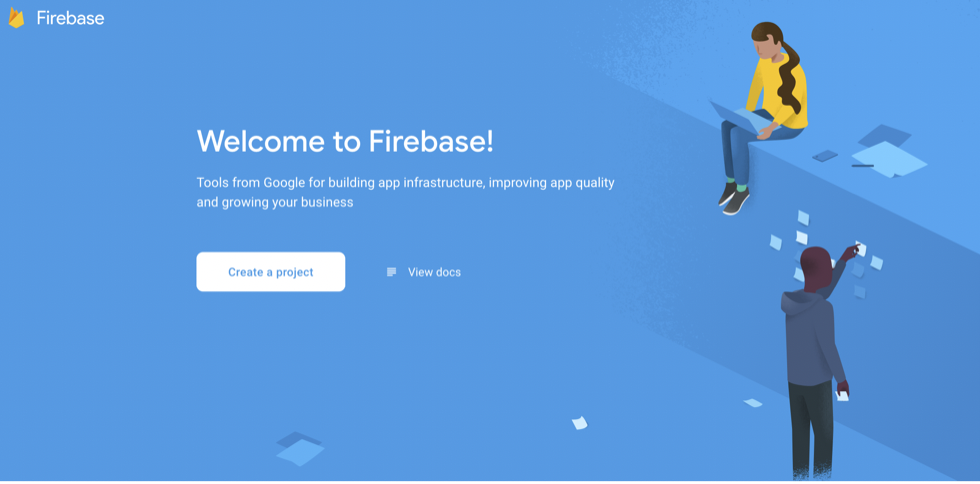
Creamos una carpeta Angular, accedemos a ella e iniciamos nuestro proyecto con el comando:

ng new MyAwesomeProject

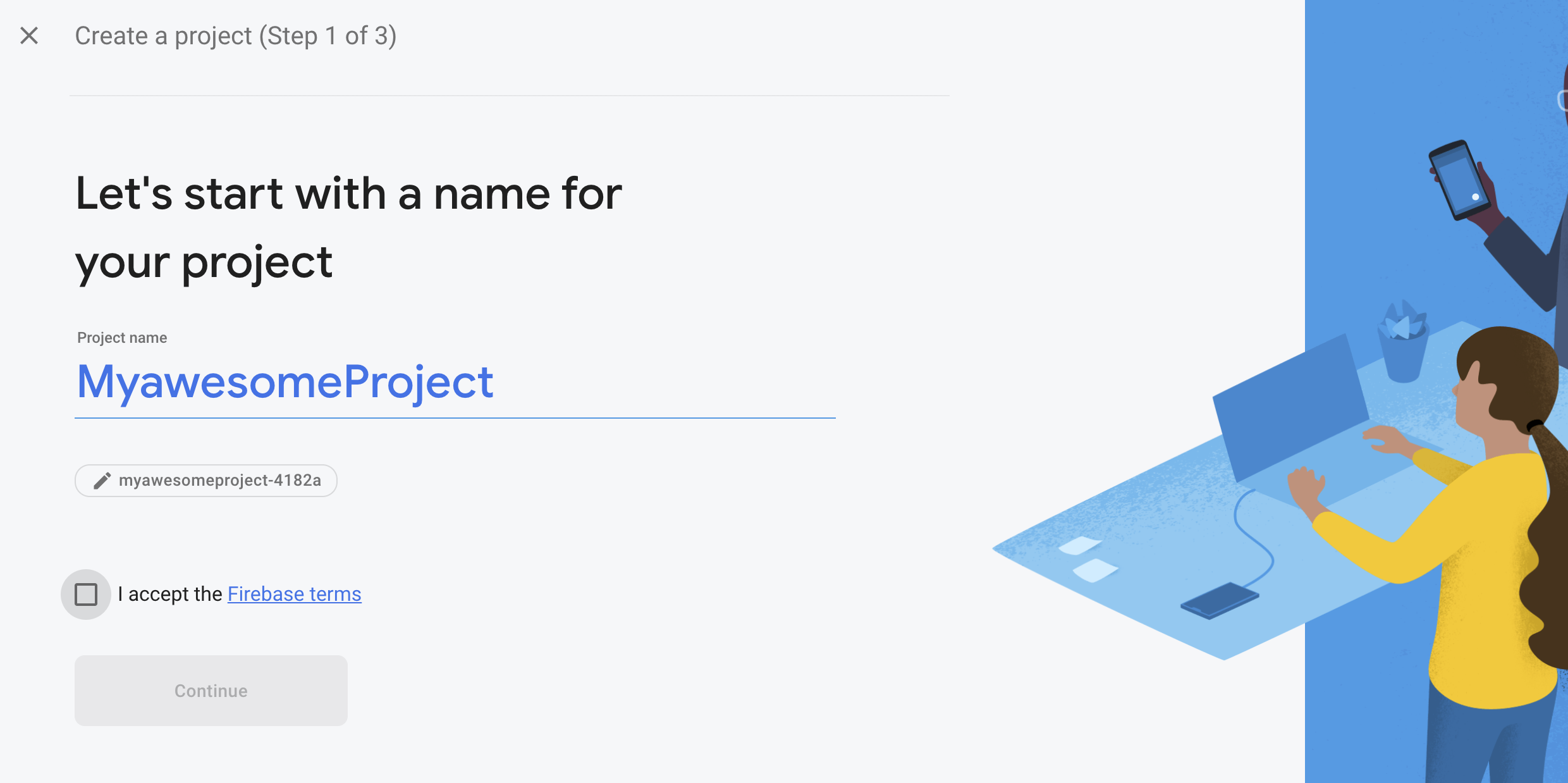
A partir del proyecto anterior vamos a ir realizando algunos cambios para ilustrar de modo básico el funcionamiento de Angular. Para este proyecto emplearemos el servicio Firebase de Google que nos permite bases de datos orientados a objetos, almacenamiento de archivos, autenticación y servicios de despliegue de apps.

Lo primero que realizaremos es, empleando nuestra cuenta Gmail, crear un proyecto en la web: <https://console.firebase.google.com>

Paso 1: Crear un nuevo proyecto

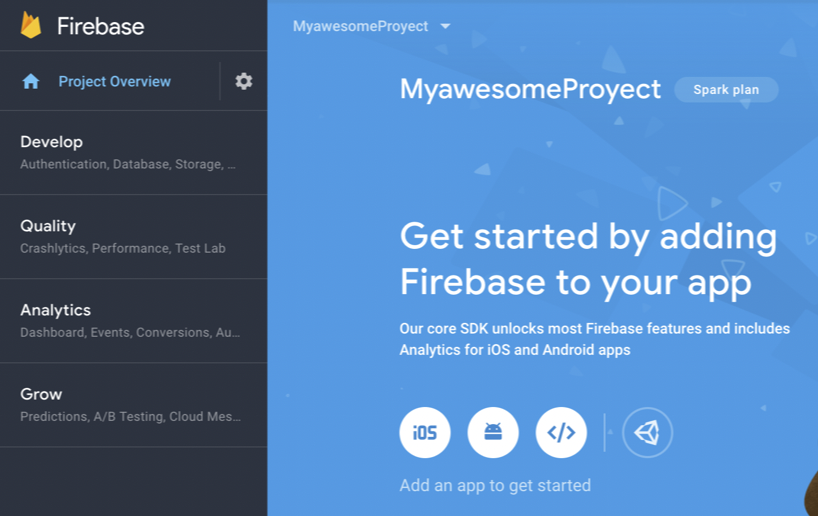


Paso 2: configurar el proyecto

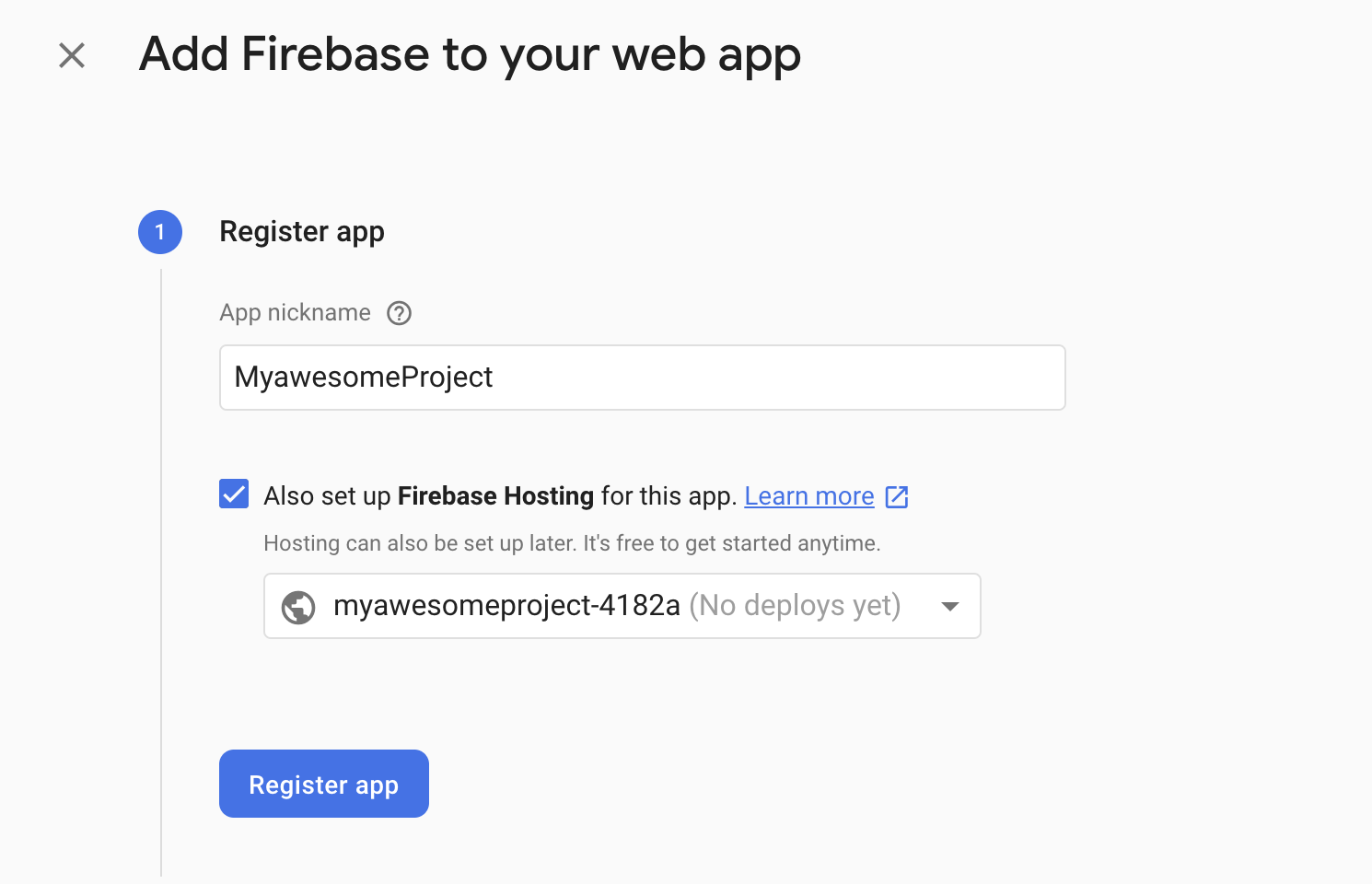


Tras aceptar las condiciones y habilitar Google analytics opcionalmente, accedemos a nuestra consola.

Paso 3: accedemos a la consola. Es interesante ver las condiciones de los planes de precio (por defecto, spark)



Paso 4: accedemos a la configuración web de la app (junto al icono IOS, Android y Unity)  y copiamos los datos. Registramos nuestra aplicación web y permitimos despliegue.



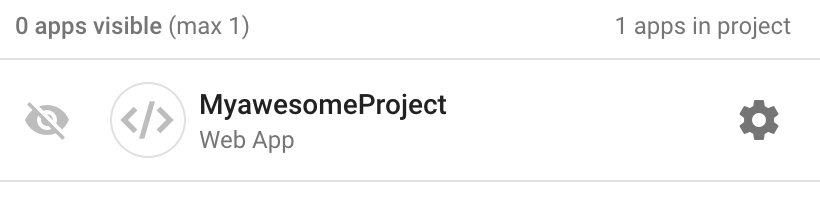
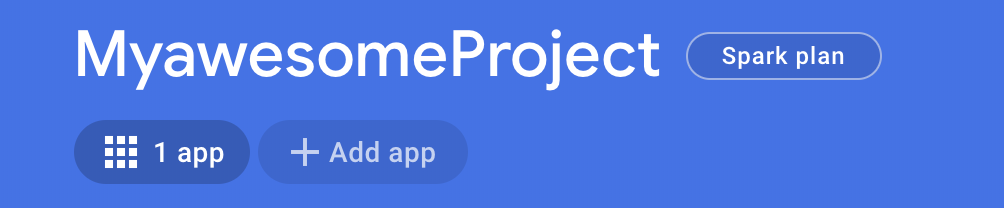
Pasamos al paso 3 donde nos indican que instalemos el cliente de firebase que nos permitirá al final de nuestro proyecto desplegar la aplicación, entre otras muchas funcionalidades.

Paso 5: instalar la librería mediante npm:

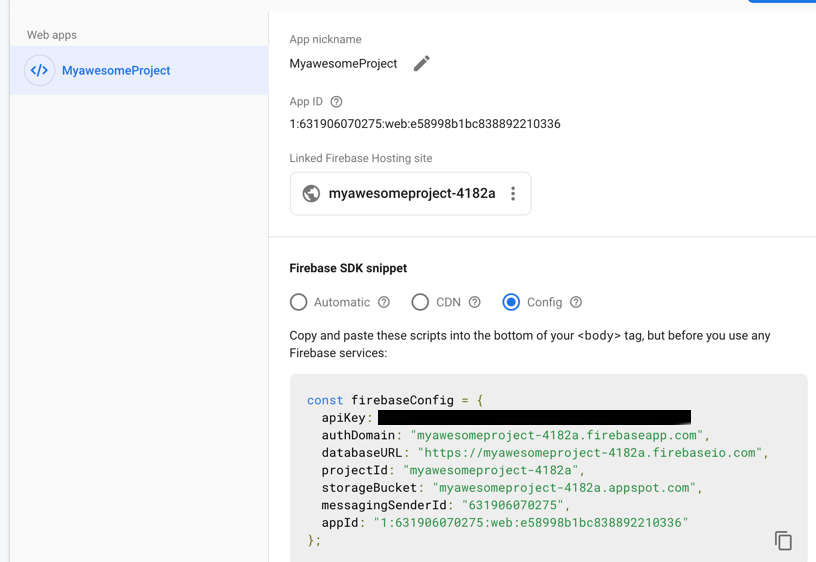
npm install -g firebase-tools

Por ahora, no haremos uso de esta herramienta.

Paso 6: Volvemos a la consola del proyecto y accedemos a la aplicación que acabamos de crear (haciendo clic en 1 app) y su configuración, en el icono de la tuerca.



Paso 7: accedemos a la sección de tus aplicaciones y seleccionamos la opción de Config.



Paso 8: copiamos todo el objeto firebaseConfig y lo pegamos en el objeto environment del archivo enviroment/environment.ts. (ojo con el formato Objetos de Javascript, los = son sustituidos por :).

export const environment = {

production: false,

firebaseConfig : {

apiKey: "AIzaSxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",

authDomain: "myawesomeproyect-3c109.firebaseapp.com",

databaseURL: "https://myawesomeproyect-3c109.firebaseio.com",

projectId: "myawesomeproyect-3c109",

storageBucket: "",

messagingSenderId: "998713355227",

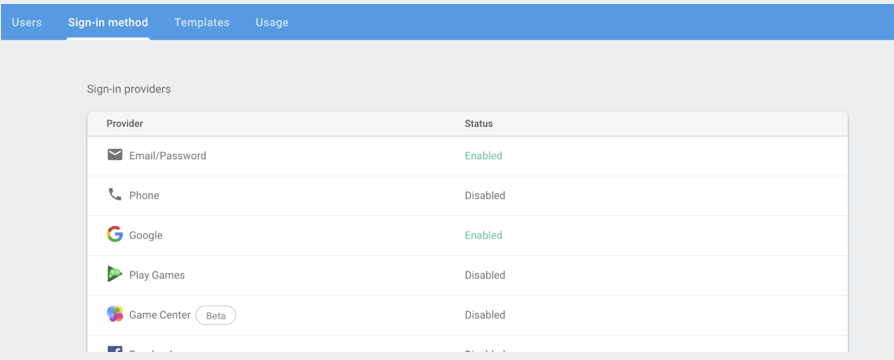
appId: "1:998713355227:web:1b5ea536837ed09c"

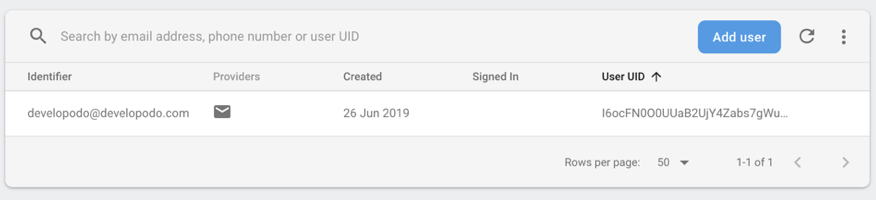
}

};

A partir de aquí ya tenemos acceso a la API de firebase. Ahora vamos a habilitar las dos funcionalidades que vamos a emplear en este proyecto: la base de datos en tiempo real Firebase y el servicio de Autenticación.

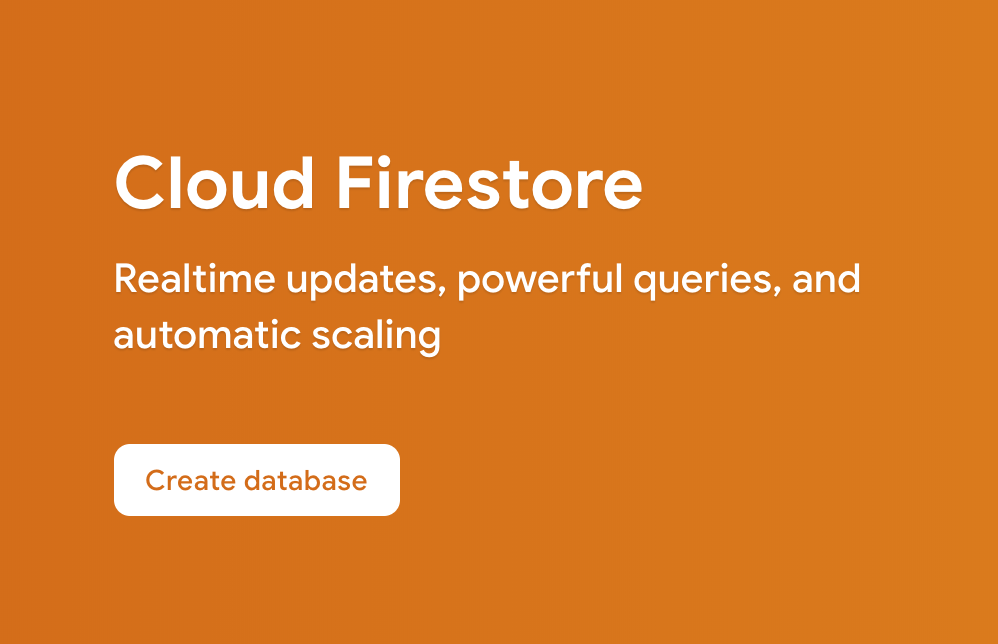
Para activar el servicio de Autenticación accedemos desde la consola a dicha opción en el panel lateral. Al activarlo, nos solicita los métodos de autenticación que se permiten. En nuestro caso vamos a habilitar por medio de las cuentas de Google y email/password. En la pestaña de users creamos un usuario para probar este método de autenticación.





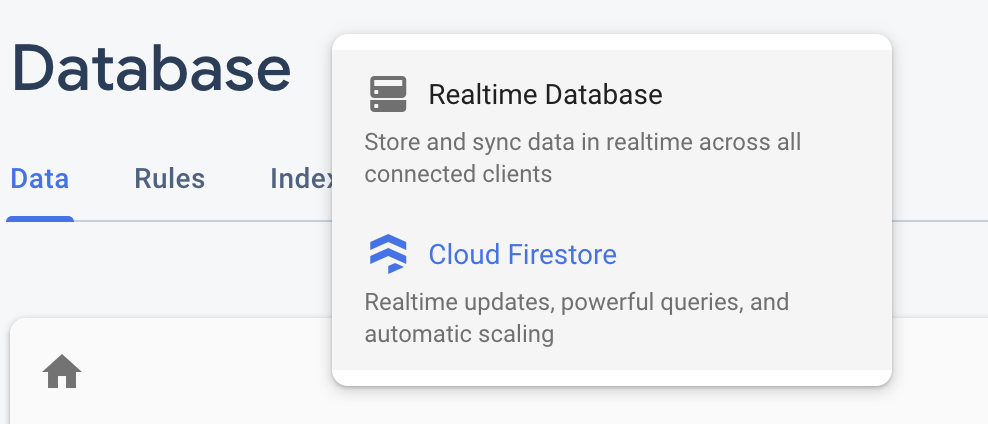
Es interesante ver las opciones de añadir usuarios manualmente o configurar la plantilla de automatización de mensajes a la hora de registrar la cuenta, incluso habilitando la necesidad de confirmar el correo para habilitar al usuario.

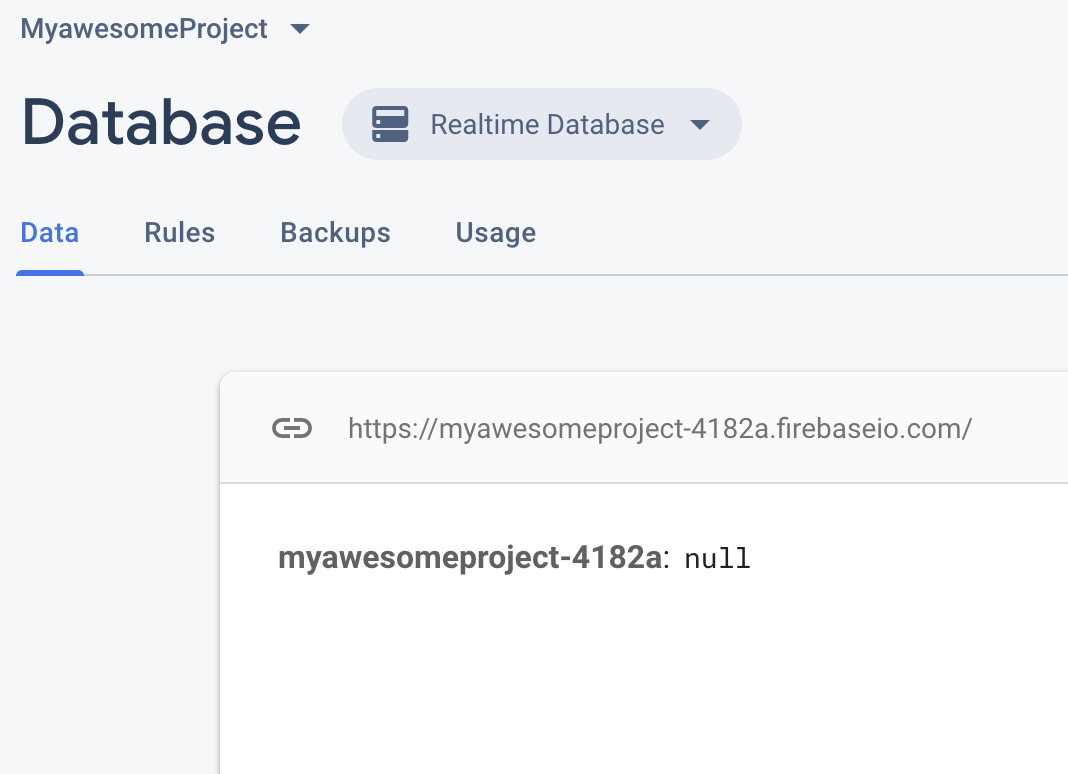
Para activar el almacenamiento de Firebase accedemos a la opción de base de datos (Database). Creamos una base de datos Real Time Database: Firebase



En los siguientes pasos aceptamos la creación en modo producción y elegimos la región que más nos convenga según nuestro país.

Finalmente, tras unos instantes aparece la consola de la base de datos, en el menú superior escogemos Realtime Database.





Este tipo de base de datos es ideal para proyectos con consultas sencillas que requieren baja latencia para mantener sincronizados usuarios conectados en tiempo real. Para proyecto con una estructura de datos más compleja se aconseja el uso de Cloud Firestore.

En rules, mientras hacemos pruebas, lo dejamos con el siguiente código:

{

/\* Visit https://firebase.google.com/docs/database/security to learn more about security rules. \*/

"rules": {

".read": true,

".write": true

}

}

Finalmente, pulsamos en Publicar (Publish). Ya tenemos nuestra base de datos lista para comenzar.

Descargamos el proyecto del repositorio

No debemos olvidar realizar los siguientes pasos:

* Ejecutar npm install para instalar las dependencias.
* Actualizar el valor de enviroment.ts y enviroment.prod.ts con los valores antes descritos de firebase para que la aplicación se pueda conectar a nuestra base de datos.

A partir de aquí, vamos a configurar el proyecto anterior para que tenga un router (hay que recordar que son SPA y se gestiona la URL de forma que, en función de la configuración del módulo router, carga una u otro módulo) y gestione 3 páginas:

* Inicio de sesión (login)
* Página principal (home)
* Sobre (about)

Entramos en la carpeta de nuestro proyecto y añadimos las páginas (componentes):

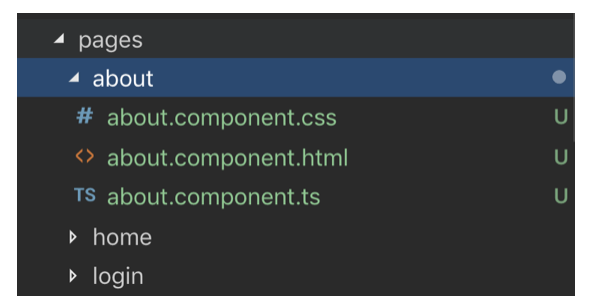
ng generate component pages/login --spec=false

ng generate component pages/home --spec=false

ng generate component pages/about --spec=false

Nota: ng help

El resultado es que se crea una carpeta pages donde existe una subcarpeta por cada componente con los tres archivos asociados: css, html y ts.



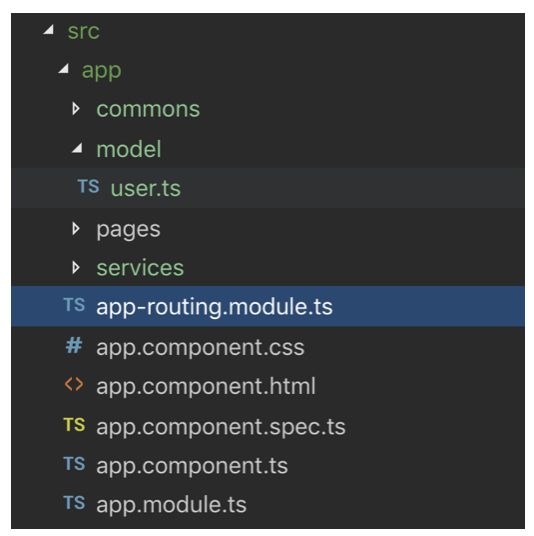
A continuación creamos el módulo que nos servirá para definir el servicio de enrutamiento:

ng generate module app-routing --flat --module=app --spec=false

--flat no crea una carpeta para el componente

--module=app importa el módulo directamente AppModule

--spec evita crear el archivo de pruebas



El resultado es que se crea el archivo:

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

import { CommonModule } from '@angular/common';

@NgModule({

imports: [

CommonModule

],

declarations: []

})

export class AppRoutingModule { }

Y automáticamente se ha actualizado el archivo module app.module.ts para que lo importe:

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { AppRoutingModule } from './app-routing.module';

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent],

})

export class AppModule { }

Modificamos app-routing.module.ts para que exporte el modulo router:

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

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

import { AboutComponent } from './pages/about/about.component';

import { LoginComponent } from './pages/login/login.component';

import { HomeComponent } from './pages/home/home.component';

import { AuthService } from './services/auth.service';

const routes: Routes = [

{ path: 'login', component: LoginComponent },

{ path: 'home', component: HomeComponent, canActivate: [AuthService] },

{ path: 'about', component: AboutComponent, canActivate: [AuthService] },

{ path: '', redirectTo: '/home', pathMatch: 'full' }, //default route

{ path: '\*\*', redirectTo: ''} //any else

];

@NgModule({

imports: [ RouterModule.forRoot(routes) ], //root level of app

exports: [ RouterModule ]

})

export class AppRoutingModule { }

Es importante destacar la variable creada routes que define las diferentes páginas. En los imports se define el módulo para la raíz del path (rutas principales) y se pasa esta variable como parámetros.

También debemos hacer hincapié en un servicio AuthService del cual se hace uso de su método canActivate en las rutas que queremos proteger para comprobar si el usuario ha iniciado o no sesión. Posteriormente analizamos este servicio. Si deseamos realizar pruebas antes de codificar este archivo debemos eliminar las llamadas a canActivate y las referencias a AuthService.

Modificamos el archivo app.component.html para que muestre el enrutamiento y una barra de navegación básica:

<h1>{{title}}</h1>

<nav>

<a routerLink="/home">home</a> |

<a routerLink="/login">Login</a> |

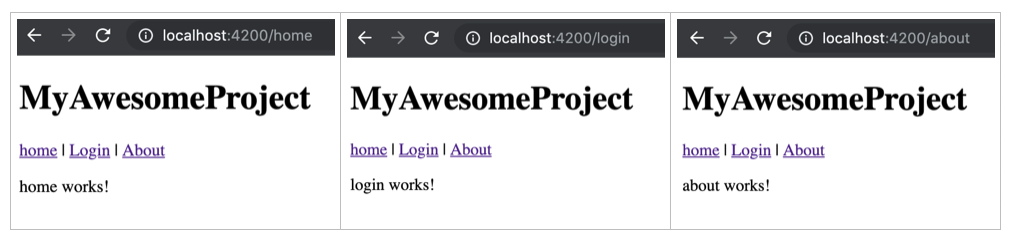
<a routerLink="/about">About</a>

</nav>

<router-outlet></router-outlet>

Como se observa, mediante la sintaxis {{}} se realiza una inyección unidireccional controlador -> vista. Por otra, parte se emplea el componente router-outlet para inyectar los diferentes componentes definidos por el router.

Si probamos la aplicación (ng serve), observamos:



A partir de aquí iremos realizando modificaciones para adaptar la interfaz.

Vamos a emplear la librería Bootstrap para la interfaz, pero una versión que no depende de jQuery para evitar bugs entre Angular y Jquery: ngx-bootstrap (<https://valor-software.com/ngx-bootstrap/#/>).

Para instalarlo:

npm install ngx-bootstrap --save

Ahora creamos un componente que consiste en una barra de navegación:

ng g c commons/navbar --module=app --spec=false

En el archivo commons/navbar/navbar-component.html:

<nav class="navbar navbar-light bg-light justify-content-between">

<ul class="nav nav-pills">

<li class="nav-item">

<a class="nav-link" routerLink="/login" routerLinkActive="active">Login</a>

</li>

<li class="nav-item">

<a class="nav-link" routerLink="/home" routerLinkActive="active">Home</a>

</li>

<li class="nav-item">

<a class="nav-link" routerLink="/about" routerLinkActive="active">About</a>

</li>

</ul>

</nav>

RouterLinkActive aplica la clase asignada cuando el router detecta que es el link activo (la ruta indicada por el link). Es decir, añade la clase active (ver bootstrap) si la aplicación se encuentra en la ruta asignada por routerLink de ese elemento. Lo empleamos para que automáticamente la barra de navegación marque como activa la página correspondiente.

Modificamos app.component.html para incorporar este nuevo componente:

<h1>{{title}}</h1>

<app-navbar></app-navbar>

<router-outlet></router-outlet>

La etiqueta app-navbar se define en el archivo navbar.component.ts.

Antes de poder emplear componentes de bootstrap debemos importar en index.html su hoja de estilos (vamos a emplear arhivo remoto, pero podría ser local):

<link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" rel=“stylesheet">

Nota: si queremos el archivo en local podemos descargarlo o moverlo desde node\_modules

En este momento podemos volver a probar nuestra aplicación:

ng serve (http://localhost:4200, por defecto).

Iniciamos FireBase y creamos el servicio de autenticación (podemos añadir de nuevo el código relacionado con canActivate en el archivo del módulo de enrutamiento: app-routing.module.ts).

{ path: 'home', component: HomeComponent, canActivate: [AuthService] },

{ path: 'about', component: AboutComponent, canActivate: [AuthService] },

Instalamos la librería de firebase:

npm install firebase @angular/fire --save

En el archivo app.module.ts

imports: [

…

AngularFireModule.initializeApp(environment.firebaseConfig),

AngularFireAuthModule

],

Creamos el servicio:

ng g s services/auth --spec=false

Nos permitirá inyectar este módulo en otros componentes. Más adelante lo codificaremos.

Ahora creamos nuestro modelo de datos (por ahora):

ng g interface model/User

Y en el archivo model/user.ts introducimos el siguiente código:

export interface User {

email:string;

}

Programando el servicio de autenticación en el archivo service/auth.ts:

import { Injectable } from "@angular/core";

import {auth} from "firebase";

import { AngularFireAuth } from "@angular/fire/auth";

import {

Router,

CanActivate

} from "@angular/router";

import { User } from "../model/user";

//this service is created by root application injector (for all app modules)

@Injectable({

providedIn: "root"

})

export class AuthService implements CanActivate {

//keep in RAM user credentials (email if it's logged in, null if it's not logged in, see user model interface)

public user: User;

constructor(private authS: AngularFireAuth, private router: Router) {

this.isLogged(); //check user at the beggining

}

async login(user: string, pass: string) {

return new Promise(async (resolve, reject) => {

try {

//it uses Google auth service by user and password.

const res = await this.authS.auth.signInWithEmailAndPassword(user, pass);

this.user = res.user;

resolve(true);

} catch (err) {

reject(err);

}

});

}

async loginGoogle() {

return new Promise(async (resolve, reject) => {

try {

//it uses Google auth by google credentials with pop up window

let provider = new auth.GoogleAuthProvider();

let res = await this.authS.auth.signInWithPopup(provider);

this.user = res.user;

resolve(true);

} catch (err) {

reject(err);

}

});

}

/\*\*

\* it returns true is user is logged in.

\*/

public isLogged(): boolean {

if (this.user!=null) {

return true; //it is already logged and stored in RAM (user variable)

} else {

//is it stored in cookies (we keep user logged in by default)?

if (localStorage.getItem("user")) {

this.user = JSON.parse(localStorage.getItem("user")); //ok, let's store in variable

return true;

} else {

//if not in cookies, is it logged in cloud (server keeps cookies too)?

this.authS.authState.subscribe(user => {

if (user) {

this.user = user; //YES, let's store in cookies and in RAM (variable)

localStorage.setItem("user", JSON.stringify(this.user));

return true;

} else {

//it is definitely not logged id. Let's remove from cookies to avoid bugs

localStorage.removeItem("user");

return false;

}

});

}

}

}

/\*\*

\* Sign out in Cloud, in Cookies and in RAM. Then, user is redirected to login page.

\*/

public async logOut() {

try {

await this.authS.auth.signOut(); //next line executed when is loggout in cloud

this.user = null;

localStorage.removeItem("user");

console.log("everyThing is removed");

this.router.navigate(['login']);

} catch (err) {

console.log(err);

}

}

/\*\*

\* Router Guard. This method is called from routing module.

\* If it returns true, then the page that user is trying to visit will be recheable.

\* The page won't be loaded if it returns false

\*/

public canActivate(): boolean {

if (!this.isLogged()) {

//this avoid bugs if user enters the url http//localhost:4200/home directly.

//The page wouldn't be recheable (thanks to routing module and canActivate call), but a blank

//page'd be loaded. To avoid this, we make sure to redirect user to login page.

this.router.navigate(['login']);

return false;

}

return true;

}

/\*\*

\* we don't use other type of Router Guard, there are here only for demostration

\*/

public canActivateChild(): boolean {

return true;

}

public canDeactivate(): boolean {

return true;

}

public canLoad(): boolean {

return true;

}

public resolve(): boolean {

return true;

}

}

Es importante fijarse en cómo se implementa el método isLogged() y sobre todo cómo se gestionan las llamadas asíncronas al servicio de autenticación de Google. Este servicio es empleado tanto por la página de login como por la barra de navegación (para mostrar el usuario y llamar al cierre de sesión).

**El código de la página de login es:**

pages/login/login.component.css:

.error{

color:red;

}

form{

margin:8%;

}

.gSignInWrapper{

text-align: center;

border-top:1px solid #888;

margin:5%;

padding:10px;

}

#customBtn {

display: inline-block;

background: white;

color: #444;

width: 190px;

border-radius: 5px;

border: thin solid #888;

box-shadow: 1px 1px 1px grey;

white-space: nowrap;

}

#customBtn:hover {

cursor: pointer;

}

span.label {

font-family: serif;

font-weight: normal;

}

span.icon {

display: inline-block;

padding-top:10px;

text-align: center;

width: 42px;

height: 42px;

}

span.buttonText {

display: inline-block;

vertical-align: text-bottom;

font-size: 14px;

font-weight: bold;

/\* Use the Roboto font that is loaded in the <head> \*/

font-family: 'Roboto', sans-serif;

}

pages/login/login.component.html:

<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">

<div class="form-group">

<label for="exampleInputEmail1">Email address</label>

<input type="email" formControlName="email" #email class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp" placeholder="Enter email">

<div class="error" \*ngIf="this.submited && this.f.email.invalid">Email is required</div>

</div>

<div class="form-group">

<label for="exampleInputPassword1">Password</label>

<input type="password" formControlName="password" #password class="form-control" id="exampleInputPassword1" placeholder="Password">

<div class="error" \*ngIf="this.submited && this.f.password.invalid">Password is required</div>

</div>

<button type="submit" class="btn btn-primary float-right">

<span class="spinner-grow spinner-grow-sm" \*ngIf="isLogging()"></span>

<span \*ngIf="!isLogging()">Log in</span>

</button>

</form>  <!-- Button Google Login -->

<div class="gSignInWrapper">

<div id="customBtn" class="customGPlusSignIn" (click)="this.logingGoogle()">

<span class="icon">

<svg version="1.1" xmlns="http://www.w3.org/2000/svg" width="18px" height="18px" viewBox="0 0 48 48" class="abcRioButtonSvg"><g><path fill="#EA4335" d="M24 9.5c3.54 0 6.71 1.22 9.21 3.6l6.85-6.85C35.9 2.38 30.47 0 24 0 14.62 0 6.51 5.38 2.56 13.22l7.98 6.19C12.43 13.72 17.74 9.5 24 9.5z"></path><path fill="#4285F4" d="M46.98 24.55c0-1.57-.15-3.09-.38-4.55H24v9.02h12.94c-.58 2.96-2.26 5.48-4.78 7.18l7.73 6c4.51-4.18 7.09-10.36 7.09-17.65z"></path><path fill="#FBBC05" d="M10.53 28.59c-.48-1.45-.76-2.99-.76-4.59s.27-3.14.76-4.59l-7.98-6.19C.92 16.46 0 20.12 0 24c0 3.88.92 7.54 2.56 10.78l7.97-6.19z"></path><path fill="#34A853" d="M24 48c6.48 0 11.93-2.13 15.89-5.81l-7.73-6c-2.15 1.45-4.92 2.3-8.16 2.3-6.26 0-11.57-4.22-13.47-9.91l-7.98 6.19C6.51 42.62 14.62 48 24 48z"></path><path fill="none" d="M0 0h48v48H0z"></path></g></svg>

</span>

<span class="buttonText">Singin with Google</span>

</div>

</div>

Es importante la línea:

<form [formGroup]="registerForm" (ngSubmit)=“onSubmit()">

Que por un lado realiza un binding de eventos (onsubmit del form) y por otro, crea un formulario reactivo y lo asocia a la variable registerForm. De esta forma, las variables del formulario tendrán doble binding y un sistema de validación muy intuitivo.

La línea:

<input type="email" formControlName="email" #email class="form-control" id="exampleInputEmail1" aria-describedby="emailHelp" placeholder="Enter email”>

Indica que el campo está asociado al (formControlName) campo email del formullario reactivo.

La línea:

<div class="error" \*ngIf="this.submited && this.f.email.invalid">Email is required</div>

Indica que esa capa solo será rendeerizada (\*ngIf) si la variable submited y la variable f.email.invalid son true.

La línea:

<div id="customBtn" class="customGPlusSignIn" (click)=“this.logingGoogle()">

Realiza un binding del evento onclick con el método loginGoogle().

pages/login/login.component.ts:

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

import { FormBuilder, FormGroup, Validators } from "@angular/forms";

import { AuthService } from "../../services/auth.service";

import { Router } from "@angular/router";

@Component({

selector: "app-login",

templateUrl: "./login.component.html",

styleUrls: ["./login.component.css"]

})

export class LoginComponent implements OnInit {

public registerForm: FormGroup;

public submited: boolean;

public submitting: boolean;

constructor(

private formBuilder: FormBuilder,

private auth: AuthService,

private router: Router

) {

this.submitting = false;

}

ngOnInit() {

if (this.auth.isLogged()) {

this.router.navigate(["home"]);

}

this.registerForm = this.formBuilder.group({

email: ["", [Validators.required, Validators.email]],

password: ["", [Validators.required, Validators.minLength(6)]]

});

this.submited = false;

}

public isLogging() {

return this.submitting;

}

// convenience getter for easy access to form fields

public get f() {

return this.registerForm.controls;

}

onSubmit() {

if (this.submitting) {

return;

}

this.submited = true;

this.submitting = true;

// stop here if form is invalid

if (this.registerForm.invalid) {

this.submitting = false;

return;

}

this.auth

.login(this.f.email.value, this.f.password.value)

.then(res => {

this.router.navigate(["/home"]);

this.submitting = false;

})

.catch(err => {

alert(err);

this.submitting = false;

});

}

public logingGoogle() {

this.submitting = true;

this.auth

.loginGoogle()

.then(res => {

this.submitting = false;

this.router.navigate(["/home"]);

})

.catch(err => {

this.submitting = false;

alert(err);

console.log(err);

});

}

}

La línea:

this.registerForm = this.formBuilder.group({

email: ["", [Validators.required, Validators.email]],

password: ["", [Validators.required, Validators.minLength(6)]]

});

Crea el formulario reactivo, indicando cuales son los campos (deben estar asociados a los formControlName del HTML y una serie de opciones. En nuestro caso, valor inicial (“”) y el validado que se ejecuta por defecto, por ejemplo para el email sería que debe estar relleno y con formato email. Se pueden crear validadores personalizados.

La línea:

public get f() {

return this.registerForm.controls;

}

Devuelve el estado de todos los campos del formulario. Útil para controlar el valor y el estado de la validación.

Es interesante analizar cómo hace uso de los servicios de autenticación y cómo gestiona la sincronía, por ejemplo, modificando mientras no vuelva la respuesta la apariencia del botón de inicio de sesión, mediante el flag submitting.

Aún no podemos probar nuestra aplicación porque existen múltiples errores. En la página home vamos a emplear un editor WYSIWYG llamado CKeditor (<https://ckeditor.com/> ). Para instalarlo, ejecutamos:

npm install --save @ckeditor/ckeditor5-angular

npm install --save @ckeditor/ckeditor5-build-classic

Ahora debemos modificar app.module.ts para que incluya todos los módulos y servicios necesarios. Su código es:

import { BrowserModule } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

import { AppRoutingModule } from './app-routing.module';

import { LoginComponent } from './pages/login/login.component';

import { HomeComponent } from './pages/home/home.component';

import { AboutComponent } from './pages/about/about.component';

import { NavbarComponent } from './commons/navbar/navbar.component';

import { AngularFireModule } from '@angular/fire';

import { AngularFireAuthModule } from '@angular/fire/auth';

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

import { CKEditorModule } from '@ckeditor/ckeditor5-angular';

import { AuthService } from './services/auth.service';

import { ModalModule } from 'ngx-bootstrap/modal';

import { environment } from '../environments/environment';

@NgModule({

declarations: [

AppComponent,

LoginComponent,

HomeComponent,

AboutComponent,

NavbarComponent

],

imports: [

BrowserModule,

AppRoutingModule,

AngularFireModule.initializeApp(environment.firebaseConfig),

AngularFireAuthModule,

FormsModule, // ngModel for editor

ReactiveFormsModule, // for validator on login page ngmodel form

CKEditorModule, // wysisygg

ModalModule.forRoot() // modal service (from ngx-bootstrap)

],

providers: [AuthService],

bootstrap: [AppComponent]

})

export class AppModule { }

Observamos cómo hemos importado todos los componentes de la aplicación y los módulos. Además, en provider hemos importado los servicios necesarios a nivel de toda la aplicación, en este caso nuestro único servicio de autenticación. Por último, se indica que el componente de entrada principal es AppComponent, cuyo código ya vimos (app.component.ts):

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

})

export class AppComponent {

title = 'MyAwesomeProject';

}

**En cuanto al código necesario para home:**

pages/home/home.component.css:

No tiene ningún código, pero si hemos modificado styles.css (raíz) para modificar la página principal y el editor:

/\* You can add global styles to this file, and also import other style files \*/

body{

padding:10px;

}

.ck-editor\_\_editable\_inline {

min-height: 400px;

}

.ck-file-dialog-button{

display:none !important;

}

pages/home/home.component.html:

<ckeditor

#editor

id="editor"

[editor]="Editor"

[disabled]="isDisabled"

(change)="onChange($event)"

(ready)="onReady($event)"

(keydown)="onKeyDown($event)"

(keyup)="onKeyUp($event)"

></ckeditor>

<div class="float-left">

<span \*ngIf="receiving" class="badge badge-pill badge-success">R</span>

<span \*ngIf="sending" class="badge badge-pill badge-primary">S</span>

<span \*ngIf="updating" class="badge badge-pill badge-warning">U</span>

</div>

<button

type="button"

class="btn btn-primary float-right"

(click)="openModal(template)"

>

View Document (no media embed)

</button>

<ng-template #template>

<div class="modal-header">

<h4 class="modal-title pull-left">Document Preview</h4>

<button

type="button"

class="close pull-right"

aria-label="Close"

(click)="modalRef.hide()"

>

<span aria-hidden="true">&times;</span>

</button>

</div>

<div class="modal-body" [innerHTML]="document"></div>

</ng-template>

Se observa el uso del componente instalado (ckeditor) y del componente modal de la librería ngx-bootstrap (ng-template).

Además, se realiza un properties binding en la línea, es decir, se asocia una propiedad del elemento HTML (en este caso innerHTML) a una variable del controlador (en este caso document):

<div class="modal-body" [innerHTML]="document"></div>

pages/home/home.component.ts:

import { Component, OnInit, TemplateRef } from "@angular/core";

import \* as ClassicEditor from "@ckeditor/ckeditor5-build-classic";

import { database } from "firebase";

//Better than import \* as database from 'firebase'

import { BsModalService, BsModalRef } from "ngx-bootstrap/modal";

@Component({

selector: "app-home",

templateUrl: "./home.component.html",

styleUrls: ["./home.component.css"]

})

export class HomeComponent implements OnInit {

public Editor = ClassicEditor; //Editor constructor,

public WYSIWYG: any; //Editor instance

private db; //firebase database instance

public isDisabled = true; //flag enabled editor

private cursor; //current cursor position

private typing: boolean = false; //user is typing?

private timerToUpdateKey; //delay to update after typing

private virtualDoc; //code in WYSIWYG to check if it needs to be updated

public receiving: boolean = false; //it's recevingn new data from db

public sending: boolean = false; //it's sending new data to db

public updating: boolean = false; //it's updatind WYSIWYG content

public document:string; //modal content

modalRef: BsModalRef; //reference to modal

//modalService inyected

constructor(private modalService: BsModalService) {}

//at the beginning

ngOnInit() {

//nothing to do in this example

}

/\*\*

\* This funcion is executed when editor is ready. We get a reference to instance (WYSISYG) and set focus

\* and the default cursor position

\* @param editor

\*/

public onReady(editor) {

editor.ui

.getEditableElement()

.parentElement.insertBefore(

editor.ui.view.toolbar.element,

editor.ui.getEditableElement()

);

this.WYSIWYG = editor;

this.isDisabled = true; //disabled editor

this.typing = false; //user is not typing

this.db = database(); //we create database object

//it gets init value from data base

this.db

.ref()

.once("value")

.then(d => {

this.WYSIWYG.setData(d.val()); //it updates content

this.enableEditor(); //it enables editor

this.WYSIWYG.editing.view.focus(); //set focus on editor

this.cursor = this.WYSIWYG.model.document.selection.getFirstPosition(); //we get current position: 0

this.listenChanges(); //it starts listener from changes on database

});

//nothing else to do, editor is disabled until previous promise is dispathed (then)

}

/\*\*

\* This function is executed every time we write on editor. We update the realtime database with new content.

\* @param e event keyup

\*/

onKeyDown(e) {

this.typing = true;

clearTimeout(this.timerToUpdateKey);

this.timerToUpdateKey = setTimeout(() => {

this.typing = false;

this.cursor = this.WYSIWYG.model.document.selection.getLastPosition();

this.onChange(null);

}, 400); //delay to avoid unnecessary changes on database

}

/\*\*

\* This function updates cursor position

\* @param e keypup event

\*/

onKeyUp(e) {

this.cursor = this.WYSIWYG.model.document.selection.getLastPosition();

}

/\*\*

\* This function creates an agent that observes the changes in database

\*/

listenChanges() {

this.stopListenChanges();

this.db.ref().on("value", d => {

/\*\*

\* Let's update data. Editor loses focus and cursor position

\*/

this.receiving = true;

setTimeout(() => {

this.receiving = false;

}, 100); //to turn off receiving icon after 0.1s

if (d.val() != this.virtualDoc) { //it only updates if database sends new content

this.virtualDoc = d.val();

this.WYSIWYG.setData(this.virtualDoc);

this.updating = true;

setTimeout(() => {

this.updating = false;

}, 100);//to turn off updating icon after 0.1s

//Let's focus again

this.WYSIWYG.editing.view.focus();

/\*\*

\* Update position to keep cursor where user was writing on

\* This method is made only for this example and has bugs

\*/

this.WYSIWYG.model.change(writer => {

let posStart = null;

//if (this.WYSIWYG.model.hasContent(this.cursor.parent)) {

try {

if (this.cursor.offset <= this.cursor.parent.maxOffset) {

let posStart = this.WYSIWYG.model.createPositionAt(

this.cursor.parent,

this.cursor.offset

);

const range = writer.createRange(posStart);

writer.setSelection(range, { fake: true });

}

} catch (err) {

console.log(err);

}

// }

});

}

});

}

/\*\*

\* This function is executed on exiting

\*/

ngOnDestroy() {

//it stops agents listening the database changes

this.stopListenChanges();

}

/\*\*

\* This function remove all agent listening on value changes

\*/

stopListenChanges() {

this.db.ref().off("value");

}

/\*\*

\* These functions change the value binded with editor to enable it

\*/

disableEditor() {

this.isDisabled = true;

}

enableEditor() {

this.isDisabled = false;

}

/\*\*

\* This funcion is fired when any change occurs on editor. We don't use it

\* @param editor that fires de change event

\*/

public onChange(e) {

//nothing to do here

if (!this.typing) {

if (this.virtualDoc != this.WYSIWYG.getData()) {

this.sending = true;

setTimeout(() => {

this.sending = false;

}, 100); //to turn off sending icon after 0.1s

this.virtualDoc = this.WYSIWYG.getData();

this.db.ref().set(this.virtualDoc, error => {

if (error) {

console.log(error);

} else {

//nothing to do if success

}

});

}

}

}

/\*\*

\* This function is called from button on view (HTML).

\* The reference to HTML modal is passed.

\*/

openModal(template: TemplateRef<any>) {

this.document=this.WYSIWYG.getData();

this.modalRef = this.modalService.show(template);

}

}

Existe mucha parte del código asociada al manejo del editor WYSIWYG que sale del objetivo de este documento, pero es importante el uso de la base de datos en tiempo real.

La línea:

this.db

.ref()

.once("value")

.then(d => {

Hace referencia a la raíz de la base de datos en tiempo real de firebase (creada anteriormente). Y ejecuta una vez (once) un escuchador del evento value, es decir que devuelve un Promise que es despachado cuando se comprueba el valor de la base de datos. De esta forma realizamos la carga inicial del documento.

La línea:

this.db.ref().on("value", d => {

…

if (d.val() != this.virtualDoc)

Crea un escuchador del evento value que se ejecuta cada vez que el valor es cambiado en la base de datos en tiempo real. Es decir, mantenemos un socket abierto con la base de datos de forma que nos notifica directamente cualquier cambio. La variable d almacena la información que nos devuelve (el contenido de la base de datos, en este caso entera, puesto que estamos haciendo una referencia a la raíz, ref() ). Podemos obtener el objeto con el método d.val().

La línea:

this.db.ref().set(this.virtualDoc, error => {

if (error) {

console.log(error);

} else {

//nothing to do if success

}

});

Realiza una inserción de valor en la raíz de la base de datos y ejecuta una función callback cuando finaliza.

Por último, actualizamos la barra de navegación:

/commons/navbar/navbar.component.html

<nav class="navbar navbar-light bg-light justify-content-between">

<ul class="nav nav-pills justify-content-center">

<li class="nav-item">

<a class="nav-link" \*ngIf="!this.auth.user" routerLink="/login" routerLinkActive="active">Login</a>

</li>

<li class="nav-item">

<a [ngClass]="setMyClasses()" routerLink="/home" routerLinkActive="active">Home</a>

</li>

<li class="nav-item">

<a [ngClass]="setMyClasses()" routerLink="/about" routerLinkActive="active">About</a>

</li>

<li class="nav-item" \*ngIf="this.auth.user">

<button class="nav-link" (click)="this.logOut()">{{this.auth.user.email}}</button>

</li>

</ul>

</nav>

/commons/navbar/navbar.component.ts

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

import { AuthService } from "../../services/auth.service";

@Component({

selector: "app-navbar",

templateUrl: "./navbar.component.html",

styleUrls: ["./navbar.component.css"]

})

export class NavbarComponent implements OnInit {

constructor(public auth: AuthService) {}

ngOnInit() {}

public logOut() {

if (confirm("Log out?")) {

this.auth.logOut();

}

}

public setMyClasses() {

let classes = {

"nav-link": true,

disabled: this.auth.user ? false : true

};

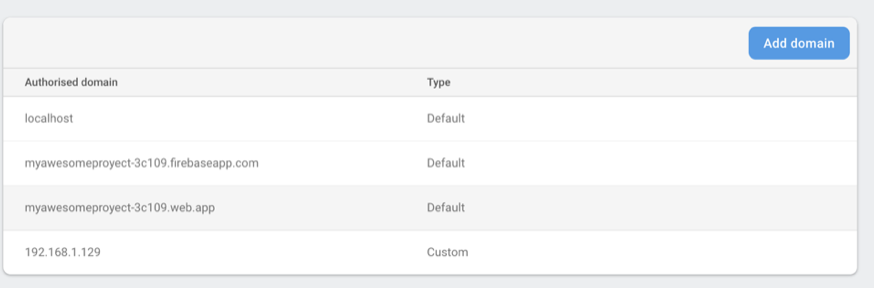
return classes;

}

}

Otra aspecto importante, en Angular es que aquellos componentes y variables que queremos que sean accesibles desde la interfaz (HTML) debemos hacerlos públicos, como es el caso del método setMyClasses.

Para que el servicio de autenticación funcione debemos incluir esta ip como dominio válido, para ello vamos a la consola de Google y en Authentication, accedemos a la pestaña Singin-method para añadir nuestra ip (en mi caso, 192.168.1.129), es importante recordar que si vamos a alojar nuestra app en dominio fuera de firebase deberíamos añadirlo aquí:



Por último, vamos a codificar la página de about, para ello, en nuestro caso vamos a copiar una imagen en la carpeta assets (Developodo\_BG2.jpg):



pages/about.component.css:

.background {

width: 100%;

height: 100%;

position: fixed;

left: 0px;

top: 0px;

z-index: -1; /\* Ensure div tag stays behind content; -999 might work, too. \*/

background-size:cover;

background-image: url(/assets/Developodo\_BG2.jpg)

}

pages/about.component.html:

<div class="background"></div>

pages/about.component.ts:

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

@Component({

selector: "app-about",

templateUrl: "./about.component.html",

styleUrls: ["./about.component.css"]

})

export class AboutComponent implements OnInit {

constructor() {}

ngOnInit() {}

}