

Prueba Técnica

Web

MERN/MEAN STACK

Vacante: Desarrollador

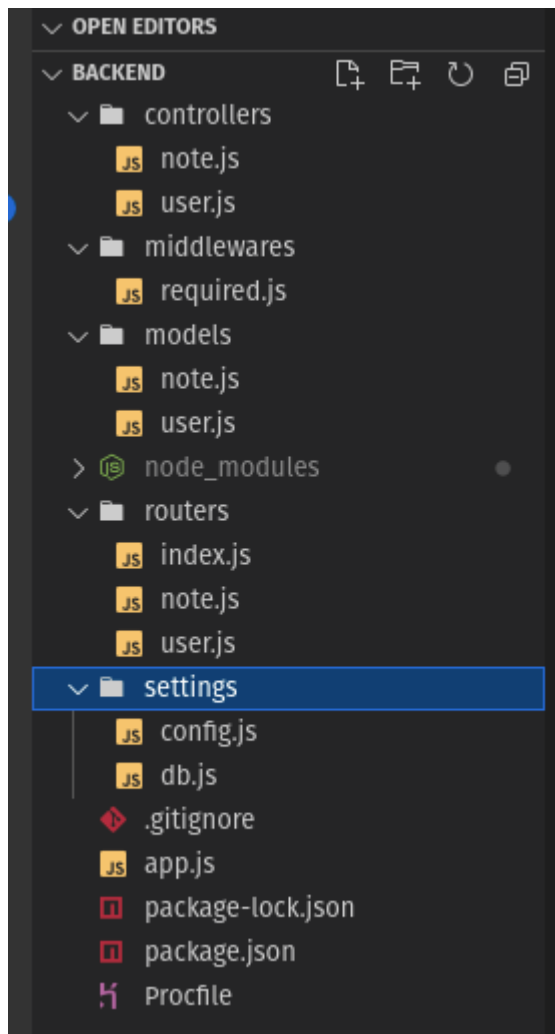
Front End

Preparación de los repositorios.

BACK-END.

El lenguaje utilizado para el desarrollo de APIs fue Express Js conectado a una Base de Datos no relacional (MongoDB).

El proyecto tiene la siguiente estructura, puede parecer a MVC.



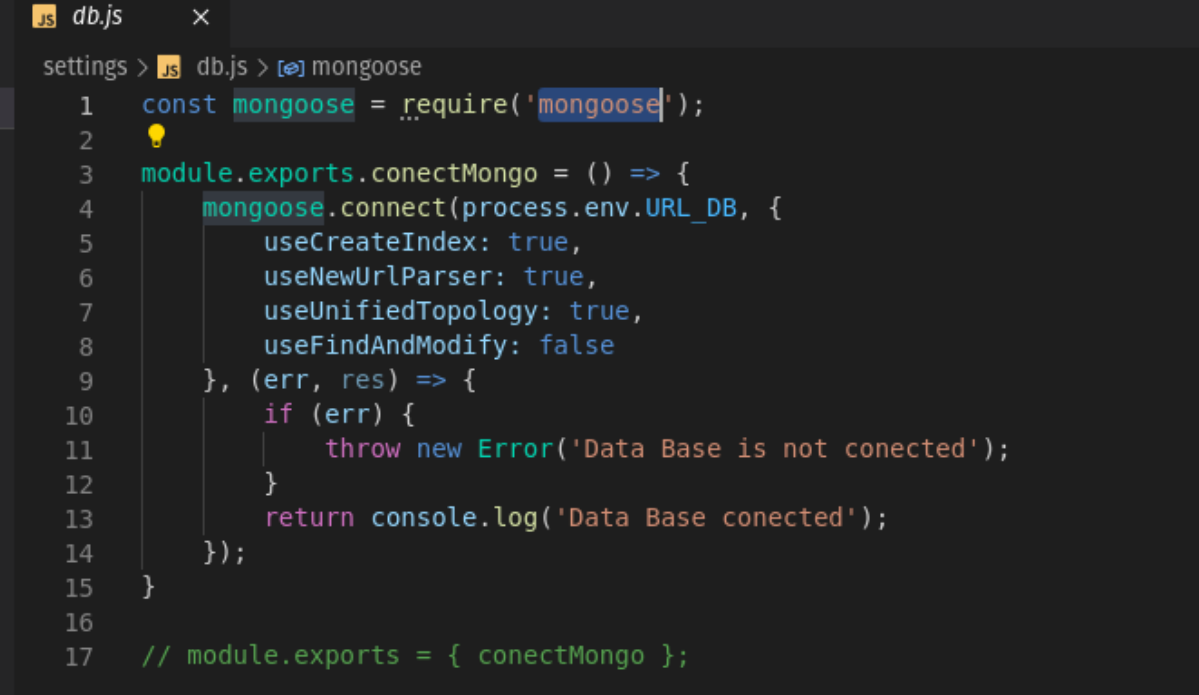
La carpeta controllers almacena toda la lógica de negocio. En este caso tenemos en cada archivo un CRUD.

Crear los middlewares por aparte, en una carpeta y sus archivos ayuda a no tener un código muy robusto, con muchas líneas. Una ventaja es que puedes reutilizar las funciones, si así fuera.

Dentro de la carpeta modelo tenemos todo los objetos que necesitamos. El archivo index.js nos ayuda a centralizar todos los modelos en un solo archivo, lo cual es bueno, así el archivo app.js quedará más legible. Esta misma acción se realiza con los routers, los routers son ayuda con la comunicación para las APIs.

Por último tenemos la carpeta settings. Dentro encontramos dos archivos config.js y db.js. En config.js podremos agregar las variables de entornos. db.js esta la coneccion a la base de datos.

Coneccion a la base de datos: Para conectarnos a MongoDB se implementó la librería Mongoose.



```
JS db.js ×
settings > JS db.js > [e] mongoose
1  const mongoose = require('mongoose');
2
3  module.exports.connectMongo = () => {
4    mongoose.connect(process.env.URL_DB, {
5      useCreateIndex: true,
6      useNewUrlParser: true,
7      useUnifiedTopology: true,
8      useFindAndModify: false
9    }, (err, res) => {
10     if (err) {
11       throw new Error('Data Base is not conected');
12     }
13     return console.log('Data Base conected');
14   });
15 }
16
17 // module.exports = { connectMongo };
```

La librería nos provee de un método para hacer una coneccion, pero debemos pasarle ciertos parámetros como la URL_DB (para este caso es una cadena de coneccion que MongoAtlas no proporciona). Esta función nos devuelve una callback para verificar la conexión.

Para crear los modelos mongoose ofrece un esquema donde podemos definir los modelos. Además podemos validar los atributos.

Rutas y Controlador: El uso de rutas es fundamental para la comunicación entre clientes.

Una de las ventajas es que el Framework Express js ofrece un método llamado Router, esto facilita la creación de rutas.

A Continuación se muestra el uso de los routers.

```
note.js x
routers > node note.js > ...
1  const router = require('express').Router();
2  const { validate, existId } = require('../middlewares/required');
3  const { check } = require('express-validator');
4  const { show, showById, create, update, destroy } = require('../controllers/note');
5
6
7  router.get('/', show);
8  router.get('/:id', [
9      check('id', 'Esto no es una nota!').isMongoId(),
10     check('id').custom(existId),
11     validate
12 ], showById);
13
14 router.post('/', [
15     check('user', 'Las notas no se crean solas!').not().isEmpty(),
16     check('title', 'Una nota sin titulo no es una nota!').not().isEmpty(),
17     check('content', 'Las notas necesitan palabras!').not().isEmpty(),
18     check('date', 'Las notas necesitan fechas!').not().isEmpty(),
19     validate
20 ], create);
21
22 router.put('/:id', [
23     check('id', 'Esto no es una nota!').isMongoId(),
24     check('id').custom(existId),
25     validate
26 ], update);
27
28 router.delete('/:id', [
29     check('id', 'Esto no es una nota!').isMongoId(),
30     check('id').custom(existId),
31     validate
32 ], destroy);
33
34 module.exports = router;
```

El uso de Router se importa de express. En este caso al ser un CRUD tenemos las siguientes rutas y sus métodos correspondientes, además a las rutas las estamos validando, con los famosos Middleware.

Al final de cada ruta tenemos nuestro controlador quien se encarga de lógica.

```
note.js  ×
controllers > Js note.js > create
const Note = require('../models/note');

const show = (req, res) => {
  Note.find()
    .populate('user', 'name')
    .exec((err, data) => {
      if (err) {
        res.status(500).json({
          ok: false,
          err
        });
      }
      return res.status(200).json({
        ok: true,
        data
      });
    });
}

const showById = (req, res) => {
  let id = req.params.id;
  Note.findOne({ _id: id })
    .populate('user', 'name')
    .exec((err, data) => {
      if (err) {
        return res.status(500).json({
          ok: false,
          err
        });
      }
      return res.status(200).json({
        ok: true,
        data
      });
    });
}

const create = (req, res) => {
  let { user, title, content, date } = req.body;

  let newDate = date.toString().split('T')[0];

  console.log(newDate);
}
```

En primera línea solicitamos a nuestro modelo para poder hacer a los métodos que el esquema nos ofrece por ejemplo “save” nos ayuda crear el objeto, “findOnde” a buscar una solo objeto, “find” obtenemos toda nuestra colección. Una de las ventajas es que cada método nos regresa una CallBack, Promesa.

En la siguiente imagen vemos cómo integrar nuestras rutas al proyecto para poder usar nuestros controladores “Lógica de negocio”

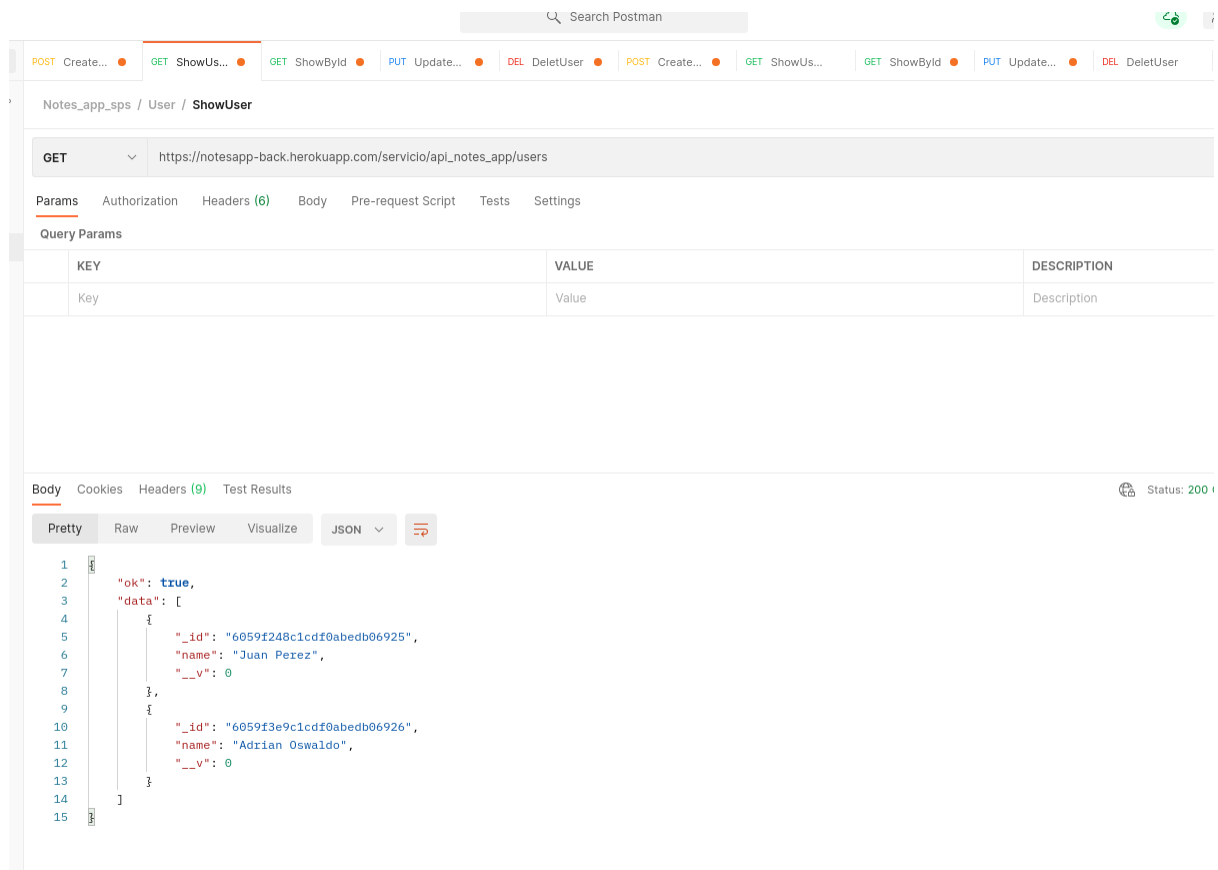
```
users > JS index.js > ...
1  const express = require('express');
2  const app = express();
3
4
5  app.use('/servicio/api_notes_app/users', require('./user'));
6
7  app.use('/servicio/api_notes_app/notes', require('./note'));
8
9  app.use('/notes-app-check', (req, res) => {
10     res.json({
11         ok: 'true',
12         message: 'Deploy is working',
13     });
14 });
15
16 module.exports = app;
17
18 // servicio/api_notes_app/users(para la sección de Usuarios)
19 // servicio/api_notes_app/notes (para la sección de Notas)
```

Levantando el proyecto y probando: A continuación se muestra el archivo que dara vida al proyecto.

```
JS app.js x
JS app.js > app.listen() callback
1  require('./settings/config');
2  const express = require('express');
3  const cors = require('cors');
4  const db = require('./settings/db');
5
6  const app = express();
7  const PORT = process.env.PORT;
8
9  app.use(cors());
10
11 app.use(express.json());
12 app.use(express.urlencoded({ extended: false }));
13
14 app.use(require('./routers/index'));
15
16 db.connectMongo();
17 app.listen(PORT, () => {
18     console.log(`run server in port: ${PORT}`);
19 });
```

En las primeras líneas solicitamos nuestra conexión de la base de datos, express js, nuestra configuraciones (Variables de entorno). Una de las ventajas de centralizar todos los archivos en uno solo nos ayuda a que nuestro archivo principal se vea limpio y fácil de leer.

Para probar nuestras APIs usamos un Framework Cliente Rest (Postman)



Deploy a Heroku:

Bueno en esta parte solo es hacer configuración en el proyecto y en la página de Heroku. En el proyecto solo agregamos un archivo Profile, nos ayudará a que nuestra aplicación ejecute algunos scripts. En heroku solo agregamos nuestras variables de entorno virtual.

```
> node bin/postinstall || exit 0
```

```
Love nodemon? You can now support the project via the open collective:  
> https://opencollective.com/nodemon/donate
```

```
added 206 packages in 3.94s
```

```
-----> Build
```

```
-----> Caching build  
- node_modules
```

```
-----> Pruning devDependencies  
removed 116 packages and audited 89 packages in 1.972s
```

```
2 packages are looking for funding  
run `npm fund` for details
```

```
found 0 vulnerabilities
```

```
-----> Build succeeded!
```

```
-----> Discovering process types  
Procfile declares types -> web
```

```
-----> Compressing...  
Done: 34.8M
```

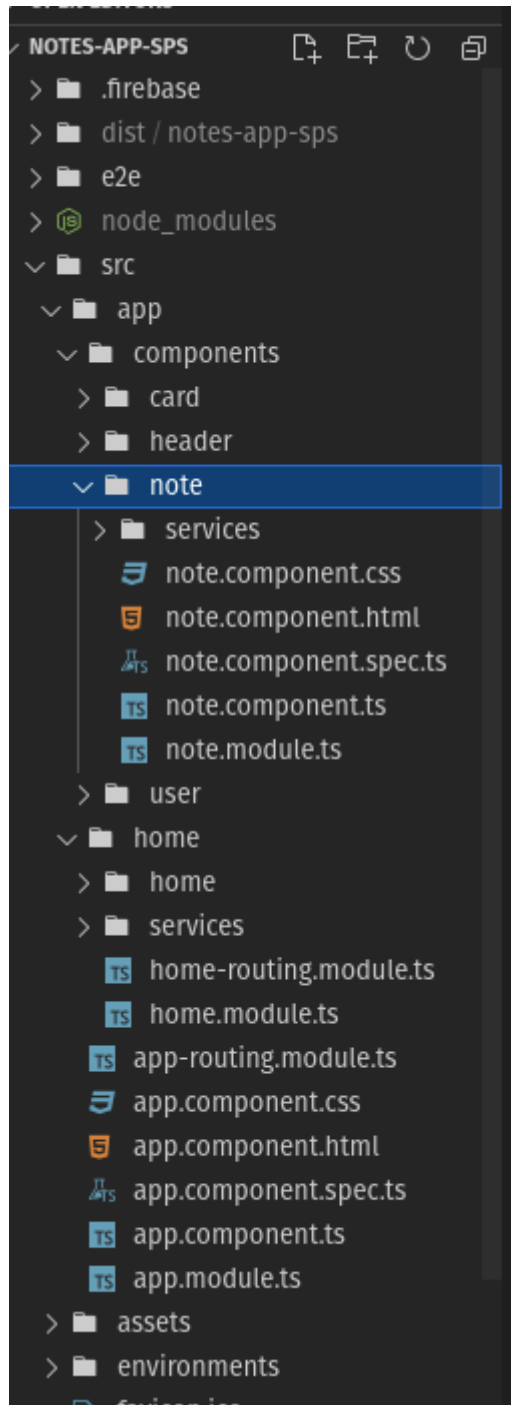
```
-----> Launching...  
Released v4  
https://notesapp-back.herokuapp.com/ deployed to Heroku
```

```
Build finished
```


FRONT-END

El Framework usado fue Angular versión 11.

El desarrollo del proyecto contiene la siguiente estructura.



La estructura consiste en tener módulos principales, esto ayudará a que la aplicación pueda ser flexible y escalable. En este caso el módulo principal es el home. Cada módulo principal cuenta con un routing.modules.ts, modules.ts. y puede contener varios componentes y servicios para cada componente. Además la aplicación cuenta con componentes que pueden ser reutilizables.

Maquetado:

Para el diseño se implementó Bootstrap.

El uso de componentes reutilizables nos ahorra tiempo y código. En este caso el uso de un Navbar, fue de buena ayuda por que se necesitaban en otras pantallas y simplemente llamas o solicitas el componente.

```
src > app > components > note > note.component.html > div.container > div.row. > div.col-  
1 <app-header [selected]="selected"></app-header>  
2  
3 <br>  
4 <br>
```

No solo el Navbar fue un componente reutilizable las tarjeta de información solo se codificó una vez!

```
<app-header [selected]="selected"></app-header>  
<br>  
<br>  
  
<div class="container">  
  <div class="row">  
    <div class="col-12">  
      <section>  
        <div class="row row-cols-1 row-cols-lg-4 g-4 g-lg-5">  
          <div *ngFor="let item of notes">  
            <div class="col">  
              <app-card [data]="item" (delete)="delete($event)"></app-card>  
            </div>  
          </div>  
        </section>  
      </div>  
    </div>  
  </div>  
</div>  
  
<!-- <app-card></app-card> -->  
<!-- <app-user></app-user> -->
```

Prueba sps

Delete

By Adrian Oswaldo - 2021-03-23

Practica de laboratorio de SPS.

Edit

Consumiendo APIs:

Para el uso del cliente de angular se importó el módulo HttpClientModule en cada archivo modules.ts de cada componente. Esto no ayuda a utilizar libremente el cliente de angular y hacer nuestras consultas a las APIs.

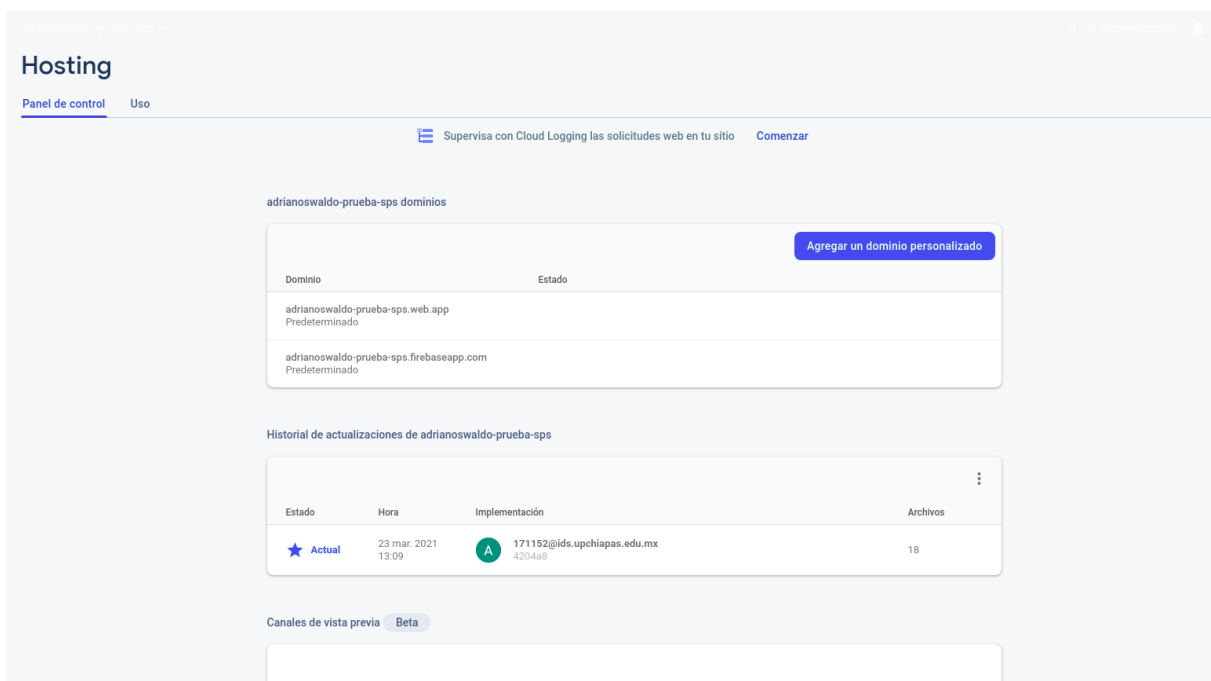
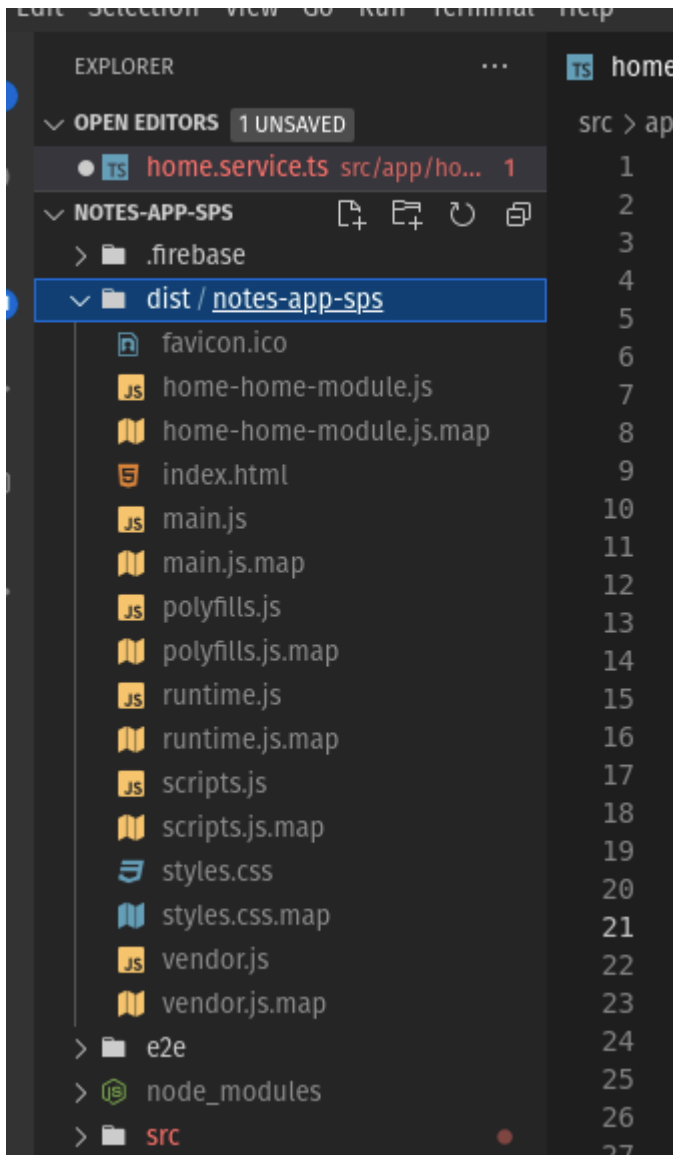
```
pp > home > ts home.modules.ts > ...  
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
  
import { HomeComponent } from '../home/home.component';  
import { HomePageRoutingModule } from '../home-routing.module';  
import { CardModule } from '../components/card/card.module';  
import { UserModule } from '../components/user/user.module';  
import { NoteModule } from '../components/note/note.module';  
import { HeaderModule } from '../components/header/header.module';  
import { HttpClientModule } from '@angular/common/http';  
  
@NgModule({  
  declarations: [HomeComponent],  
  imports: [  
    CommonModule,  
    HomePageRoutingModule,  
    CardModule,  
    UserModule,  
    NoteModule,  
    HeaderModule,  
    HttpClientModule,  
  ],  
})  
export class HomeModule { }
```

El cliente angular permite hacer peticiones hacia alguna API Rest. Se pueden usar los métodos POST, GET, PUT y DELETE cada método solicita argumentos distintos. El uso del patron Observer nos ayuda a estar suscrito a la API y esperar algún cambio y así estar en “tiempo real” cuando haya algún cambio o la API nos responda.

```
src > app > components > note > services > note.service.ts > NoteService > url
1  import { Injectable } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3  import { Observable } from 'rxjs';
4  import { UserService } from '../../user/services/user.service';
5  import { HomeService } from '../../home/services/home.service';
6
7
8  @Injectable({
9    providedIn: 'root'
10 })
11 export class NoteService {
12
13   url = 'https://notesapp-back.herokuapp.com';
14
15   constructor(private http: HttpClient, private user: UserService, private alertService: HomeService) { }
16
17   createNote(body:any): Observable<any> {
18     return this.http.post<any>(`${this.url}/servicio/api_notes_app/notes`, body);
19   }
20
21   getNote(id:any): Observable<any> {
22     return this.http.get<any>(`${this.url}/servicio/api_notes_app/notes/${id}`);
23   }
24
25   deleteNote(id:any):Observable<any> {
26     return this.http.delete<any>(`${this.url}/servicio/api_notes_app/notes/${id}`);
27   }
28
29   updateNote(id:any, body:any) {
30     return this.http.put<any>(`${this.url}/servicio/api_notes_app/notes/${id}`, body);
31   }
32
33   getUsers() {
34     return this.user.getUsers();
35   }
36
37   showAlert(title:any, message:any, type:any) {
38     this.alertService.infoAlert(title, message, type);
39   }
40
41 }
42
```

Deploy en Firebase: Firebase es plataforma para el desarrollo de aplicaciones Web y Moviles. Unos de los grandes beneficios es el hosting.

Para el deploy se necesitó de una cuenta de Firebase y configurar nuestro proyecto de angular. El primer paso fue crear nuestra carpeta pública donde se encuentra toda nuestra Aplicación. Después, fue crear nuestra proyecto en Firebase y realizar nuestra integración con Firebase.



Anexos

Link FRONT-END

<https://adrianoswaldo-prueba-sps.web.app/home>

Link API

<https://notesapp-back.herokuapp.com/notes-app-check>