Ejercicio Angular – Hiberus:

Sandro López Gil

Para este ejercicio se nos piden varias cosas primero :

- escalabilidad del directorio de trabajo y modularidad, con esto nos referimos a la capacidad de organizar cada una de las funcionalidades y interfaces de tal manera que si en un futuro queremos ampliar el proyecto podamos hacerlo sin necesidad de cambiar todo el código, como mucho solo una parte de el o a veces solo crear nuevos módulos y intercambiarlos por los nuestros dependiendo de la situación. Por ello he pensado en hacer dos módulos uno de ellos se encarga del login y sign up así como de la gestión y obtención del token y otro que se encargue de la zona de usuarios y todo el CRUD involucrado hacia la API REST, cada módulo tiene los siguientes subcomponentes:
    - **Login y signUp**:
        - Tendremos unos dos componentes uno de registro y otro de entrada, en este lo que haremos es obtener el token.
    - **Usuarios**:
        - Tendremos un componente padre con dos componentes hijos que funcionaran como tabla y como formulario, además en este módulo tenemos definidos el DAO, DTO y sus interfaces correspondientes. Por último tendremos un componente independiente llamado home con ciertas estadísticas representadas a través de Google charts y un interceptor que se encargará de configurar las llamadas a la API REST.

- Por otro lado la arquitectura del proyecto, para este proyecto he decidido que vamos a tener distintas partes entre ellas Modelo vista controlador para lo que son los componentes de angular en la aplicación , vamos a tener DTO's y DAO para estructurar los datos y sus propiedades así como el acceso a ellos, también será EDA puesto que sobre todo tenemos una aplicación orientada a eventos, además he pensado el incluir un middleware debido a la necesidad de Logins y tokens así como el uso de pipes o interceptor para configurar las llamadas a la API REST y por último tener un singleton para utilizar el DAO.
- Para la nomenclatura usaremos el camelCase para los nombres y usar las reglas de estandarización de Google así como su estructuración para ficheros y carpetas.

- Para la gestión de datos de sesión usaremos el sessionStorage para el guardado del token Bearer debido a su rápida capacidad de obtención de datos cuando lo queramos así como por su seguridad gracias a la encriptación de los mismos.

- Para la maquetación usaremos la librería de material debido a que también pertenece a Google y es la más recomendable dentro del uso de angular.

  En este ejercicio me encargaré solamente de lo que es la parte de la visualización y la lógica detrás de esta visualización como ordenar usuarios traspasar datos de componentes a componentes y demás. Debido a que tengo una API REST que funciona también como base de datos no gestionaré lo que en líneas generales es el backend solo realizaré las llamados a través de los protocolos HTTP, por ello puedo confirmar que el usuario de este proyecto será al final un thin client.

  Sobre ciertas condiciones que se piden en el ejercicio tengo una página de login, otra de registro y otra de usuarios, pero no de logout. He pensado que al ser solo un botón el logout no necesita una página por sí sola.

  Aquí mostraré ciertas partes del código que me parecen más importantes de explicar:

  Interceptor:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent
<any>> {
    var token = sessionStorage.getItem("token");

    if(token != null)
    {

      var valor_token = JSON.parse(token);
      var request = req.clone({
        setHeaders:{
          authorization: valor_token.tokenType + " " + valor_token.access
Token
        }
      });
      return next.handle(request);
    }
    else
    {
      return next.handle(req).pipe(
        catchError(this.handleError.bind(this))
      );
    }
  }
```

  Lo que hace este interceptor es valga la redundancia interceptar todas las peticiones que hagamos a la API una vez autenticados es decir una vez que

hayamos obtenido el token. Primero recupero el token del sessionStorage luego compruebo si es null o no, si es null realizamos la petición sin cabecera actualizada lo que nos devolverá un error y nos redirigirá a la página del login, si existe el token pueden ocurrir dos cosas, la primera que sea válido y no de error, la segunda que de error porque el token no sea válido o haya expirado y nos redirija a la página del login a través de un route.navigate

Otra cosa que es bastante importante son el DTO y DAO, tendrán este código:

```
export class UsuariosDTO implements UsuariosInterface{
    email:string;
    password:string;
    name:string;
    surname:string;
    id:string;
    constructor(email:string , pass:string,name:string,surn:string,id:str
ing){
        this.email=email;
        this.password=pass;
        this.name=name;
        this.surname=surn;
        this.id=id;
    }
    getName(){
        return this.name;
    }
    getPassword(){
        return this.password;
    }
    getId(){
        return this.id;
    }
    getSurname(){
        return this.surname;
    }
    getEmail(){
        return this.email;
    }
    setName(data:string){
        this.name=data;
    }
    setSurname(data:string){
        this.surname=data;
    }
    setEmail(data:string){
        this.email=data;
    }
```

```
    setId(data:string){
        this.id=data;
    }
    setPassword(data:string){
        this.password=data;
    }
}
```

Esto sería el DTO que contendrá los campos nombre, apellido, email, contraseña y por último id. Además, contiene los métodos internos para la obtención y edición de las propiedades.

```
export class usuariosDAO implements usuariosDAOInterface{
    ip = 'http://51.38.51.187:5050/api/v1/users';

    constructor(private httpClient: HttpClient){}

     crearUsuario(usuario:UsuariosDTO):Observable<any>{
          return this.httpClient.post<any>(this.ip,usuario)
    }
    borrarUsuario(usuario:usuariosSimpDTO):Observable<any>{

     return this.httpClient.delete<any>(this.ip+"/"+usuario.id)
    }
     listarUsuarios():Observable<any>{

         return this.httpClient.get<any>(this.ip)

    }
     buscarUsuario(id:string):Observable<any>{

        return this.httpClient.get<any>(this.ip+"/"+id);

    }
     editarUsuario(usuario:UsuariosDTO):Observable<any>{

        return this.httpClient.put<any>(this.ip+"/"+usuario.id,usuario)

    }
    public onError(error:HttpErrorResponse){
        console.log(error);
    }
}
```

Esto sería el DAO que se encargará de definir las peticiones y los tipos de datos que devuelvan o que se envíen ya sea como parte del cuerpo o la cabecera de la

petición en mi caso cada uno devuelve un observable que podrán ser usados desde el componente que queramos a través de la inyección de este servicio.

Por último, hay ciertas partes del CRUD que creo que tienen falta de coherencia, por ejemplo, que cualquier usuario sea capaz de borrar editar o crear un usuario incluso si no es el suyo, otra cosa que no creo que sea lógico es que a la hora de editar un usuario debas enviar una petición que contenga el nombre, apellido, contraseña, email e id. El problema está en que no hay manera de obtener la contraseña de un usuario a no ser que seas ese usuario por lo que cada vez que edites un usuario debes cambiar la contraseña puesto que no permite pasar un campo de contraseña vacío. Una manera de solventar esto en mi opinión es limitar o restringir al usuario por lo menos en lo que editar respecta de tal manera que un usuario solo puede editar su propio usuario, dado que si que se puede recuperar la contraseña del usuario logeado a través de uno de los endpoints.