TRABAJO DE FIN DE GRADO
(TFG)

CFGS Desarrollo de Aplicaciones Multiplataforma

Informática y Comunicaciones

<Library of Ohara>
Memoria del Proyecto de Fin de Grado (TFG)

Año: *<2025>*

Fecha de presentación: *(23/05/2025)*

Nombre y Apellidos: Jaime González Bravo

Email Corporativo: jaime.gonbra@educa.jcyl.es

Email personal: jaimegonbra1789@gmail.com

Índice de la Memoria

1. Organización de la memoria:

1.1 Descripción general del proyecto:

-Consistirá en definir lo que hace el proyecto o lo que se buscaba a la hora de diseñar el proyecto.

Además, se expondrán todos los objetivos del mismo, ya sean principales o secundarios.

Por otro lado, se explicarán todas aquellas cuestiones metodológicas a lo largo del desarrollo.

En este apartado, también se hablan del entorno de trabajo referido a que elementos de desarrollo se han empleado a la hora de desarrollar la aplicación como los IDEs o los lenguajes de desarrollo.

1.2 Descripción general del producto:

-Consistirá en definir aquellas funcionalidades que permite realizar el programa junto a una breve descripción de los métodos, técnicas o arquitecturas empleadas.

Por último, añado dos manuales, el manual de despliegue y el manual de instalación. Dichos manuales, los encuentro muy útiles debido a la división de la aplicación en dos partes y cada una se despliega de una manera y debe de tener un orden establecido.

1.3 Planificación y presupuesto:

-Consistirá en definir aquella planificación del proyecto, así como un hipotético presupuesto del desarrollo del mismo.

Como no tenía ni idea de cómo realizar un presupuesto, hice lo que creí conveniente.

1.4 Documentación técnica:

-En este apartado, se explicarán más detalladamente los aspectos del proyecto.

Dichos aspectos son:

- El análisis del sistema, teniendo en cuenta los requisitos funcionales de la aplicación.

- El diseño del sistema hablando de cómo está diseñada la base de datos, mostrando en todo momento el cómo se relacionan las distintas tablas de la aplicación.

  Además de mostrar el diseño de la interfaz de la aplicación a nivel visual, véase, lo que va a ver el usuario cuando arranque la aplicación y el diseño de todos los procesos seguidos por la aplicación.

- La implementación del programa hablando de los entornos de desarrollo y el software que se ha empleado para su desarrollo (no se si esto ya lo comenté, pero ahí va otra vez).

  También incluye algunas librerías empleadas como el modelo a seguir durante el desarrollo.

  Además, comentar algunas cuestiones de diseño de porque decidí hacer las cosas así o de otra forma o alguna cosa curiosa que he aprendido durante el desarrollo.

  Por último, está el apartado de las pruebas donde se hablan de aquellos procedimientos a la hora de realizar el testing de la aplicación.

1.5 Manuales:

-En este apartado están todos aquellos manuales que explican tanto lo que un usuario puede hacer como instalar y desplegar la aplicación. (creo que ya lo indico en otro punto de esta memoria, pero en caso de que se busquen en específico, los 3 están ahí también.)

1.6 Conclusiones y posibles ampliaciones:

-En este apartado, se añade información sobre aquellos elementos que, por falta de tiempo, falta de conocimiento no han podido salir a la luz.

1.7 Bibliografía:

-En este apartado, se comentarán o se añadirán enlaces a los cuales el autor de esta memoria (véase yo) he visto importantes o relevantes a la hora de dar más contexto o explicar algunos conceptos.

1.8 Anexos

-En este apartado, se añadirán anexos a diferentes enlaces, documentos de suma importancia para el proyecto en sí, ya sean para tener comunicado las dos partes del proyecto (en el Git, tengo la parte Back por un lado y la parte Front por otra).

2. Descripción general del proyecto:

- “Library Of Ohara” es una aplicación web que emplea un cliente front-end desarrollado en angular que realiza peticiones a una API que actúa de servidor desarrollada en Spring Boot.

Dicha aplicación permitirá a los usuarios gestionar aquellos libros que quieran leer o que ya hayan leído.

2.1 Objetivos:

2.1.1 Objetivo principal:

-Conseguir emplear una aplicación que esté diferenciada en dos partes (la parte front-end y la parte back-end) y conectarlas para que se puedan comunicar.

-Realizar una aplicación de gestión de usuarios con libros, permitiendo así que un usuario disponga de su lista de libros en la que podrá añadir nuevos libros, quitarlos. Además, permitir que pueda modificar el estado de la lectura de un libro permitiéndole cambiar entre sin empezar, leyendo, pausado completado y dejado (en caso de que lo haya empezado pero que no le haya gustado).

1.1.2 Objetivos secundarios:

-Demostrar que los conocimientos aprendidos tanto en el curso de DAM (diseño de aplicaciones multiplataforma) como el de DAW (diseño de aplicaciones web) que realice hace unos años me han servido y he podido aplicarlos en esta aplicación.

-Crear una aplicación "real". Cuando me refiero a real, me refiero a una aplicación que se pueda utilizar perfectamente y que se asemeje a una aplicación que de verdad exista.

1.2 Cuestiones metodológicas:

-A la hora de realizar el proyecto, me he basado en el modelo de desarrollo en cascada. Dicho modelo, se basa en el desarrollo de un proyecto de forma secuencial dividida en varias fases. Además, este modelo permite el uso de "rollbacks" en caso de que una de las fases sea incoherente o tenga algún error. Estas fases son:

-Analítica o búsqueda de requisitos: en ella, como su nombre indica, se analizará que se quiere realizar y como se podría realizar. En mi caso, sería la búsqueda de lenguajes de desarrollo en el que realizar el TFG y buscar que funcionalidades quiero para el proyecto, que datos quiero tratar, etc.

-Diseño: en esta fase, se realizan unos "mockups" o bocetos de la aplicación. En mi caso, sería definir que campos requiero emplear, como mostrarlos y utilizarlos.

-Desarrollo o Implementación: en esta fase, se desarrolla el código del proyecto.

-Pruebas: en esta fase, se desarrollarán las pruebas que el desarrollador vea necesarias. En el caso del TFG, no sé si he realizado alguna, no he visto la necesidad de hacer testing ya que yo mismo hacia las pruebas para probar que todo funcionara.

-Instalación: en esta fase, una vez comprobada toda la aplicación, se comprueba que en los equipos del cliente funcione correctamente. En el caso del TFG, como no tengo "clientes", solo tengo que comprobar que en mi máquina funciona.

Gracias a este modelo, pude organizarme a la hora de desarrollar el proyecto.

1.3 Entorno de trabajo:

1.3.1 Tecnologías:

-He seguido la ideología del modelo cliente-servidor. Dicho modelo es un concepto de arquitectura más común en el desarrollo de software en el que se requiere de un servidor que reciba las peticiones y un cliente que realice las peticiones.

En mi caso, he dividido el desarrollo en dos, la parte de front-end haciendo de cliente y la de back-end haciendo de servidor respectivamente. Lo realicé así debido a que me gustaba la idea de hacer las 2 partes de una aplicación y que se pareciera a una aplicación real.

1.3.2 Herramientas del software

-Debido a la división de la aplicación en dos, para la parte back, he empleado Intellij como IDE (entorno de desarrollo), ya que me siento más cómodo que con NetBeans y Eclipse no lo he empleado casi nada. además, durante las practicas, he utilizado este IDE afianzando así aún más los conocimientos que tenia de este. para la parte front, he tenido en cuenta lo aprendido durante el ciclo de grado superior de DAW empleando así EL IDE de Visual Studio Code. Elegí VSC ya que es gratuito y te permite gestionarlo añadiendo un montón de extensiones para facilitar el desarrollo. En mi caso, como lo tenía instalado para cuando hice dicho ciclo, tenía algunas extensiones interesantes para desarrollar el TFG.

Como lenguajes de programación, para back, he decidido utilizar Java, en especial el framework de Spring Boot, visto en el ciclo actual (DAM). lo elegí, debido a que me siento más cómodo con ello, Java es mi primer lenguaje de programación, (como muchos) por lo tanto, es el que tengo más horas invertidas y más me he "peleado" con él.

Para el front, quería hacerlo en otro lenguaje, primero para variar de java, y porque conozco algún que otro lenguaje dedicado al front. por ello, decidí utilizar Angular, que es el framework de TypeScript, que a su vez es un lenguaje que utiliza por debajo JavaScript. Angular fue uno de los lenguajes que más problemas me dio cuando estuve haciendo DAW. durante ese curso, estuve dando Angular-16, y cuando estuve trabajando en Icon-Multimedia con Angular precisamente, utilicé la versión 16 y estaban a punto de sacar la versión 17. Actualmente están por la 19, pero desde la 17, cambiaron algunas cosas respecto a cómo se crean los proyectos, así que he mantenido la versión 16 o, mejor dicho, estoy trabajando como si fuera la versión 16. Además, he de comentar que yo empleo angular-js, véase, que utilizo las librerías de Node para ejecutarlo. esto es así porque es la manera que aprendí.

Para la base de datos, empleé MySQL ya que es el que más he dado, y el que me siento más cómodo. es un sistema gestor de bases de datos relacional. dichas relaciones, me permitían poder integrar unos datos relacionados entre sí.

2. Descripción general del producto

2.1 Funcionalidad básica:

-La aplicación permite al usuario agregar a su lista aquellos libros que haya leído o que quiera leer siempre y cuando, dichos libros estén en la propia biblioteca que te da el programa.

para poder acceder, se requerirá de loggearse / iniciar sesión o en contraparte, registrarse / crea una cuenta rellenando un formulario respectivamente. una vez iniciada la sesión / registrado, el usuario podrá ver los libros y añadirlos a su lista.

2.2 Arquitectura:

-Para gestionar la aplicación, he empleado el modelo de Modelo-Vista-Controlador (MVC) porque me es más cómodo emplearlo ya que lo he visto tanto durante el curso de DAM como durante el curso de DAW.

En mi caso, el modelo estaría íntegramente en la aplicación back-end, la vista estaría íntegramente en el front (teniendo la típica del swagger para comprobar todas las peticiones). Para el controlador, sería una combinación entre la aplicación front-end con los servicios y la aplicación back-end con los controladores permitiéndome así, conectar las comunicaciones entre las dos aplicaciones.

2.3 Manual de despliegue:

-Para desplegar la aplicación, en caso de que no dispongamos del código, mirar el manual de instalación más adelante.

Una vez teniendo cargado el sql, podremos lanzar la aplicación back-end, podremos comprobar que todo va bien si no ha dado ningún error o si nos vamos a la url del swagger y carga todos los datos.

una vez cargada la aplicación back-end, tendemos que lanzar la aplicación front-end, yo utilizo el comando en la terminal de la carpeta padre del proyecto (cuando se abre con VSC, -con pulsar “ctrl+ñ” nos abre la terminal o bien, yendo a la terminal directamente) y poner

el comando de ng serve -o para lanzar la aplicación (si es la primera vez, nos pide una confirmación, lo hace siempre que creamos un nuevo proyecto de angular, le decimos que Sí) le pongo el -o para que nos abra directamente el navegador con la página inicial del proyecto en vez de tener nosotros que irnos al navegador y escribir la url del proyecto de front (localhost:4200).

2.4 Manual de instalación

-Para instalar la aplicación, primero, al igual que en el despliegue, debemos bajarnos del repositorio Git, las dos partes, la de front, como la de back.

enlace parte front TFG

enlace parte back TFG

para descargarlos, tan solo hay que pulsar a donde pone lo de “code” en verde y a “download ZIP” (preferiblemente hacerlo con el zip, también se puede clonarlo), una vez descargados, hay que descomprimirlos, para ello, utilice la herramienta que tenga disponible (recomendable en zip) y colóquelos en donde quiera.

Después, tendríamos que ejecutar el código del sql para cargar la base de datos, para localizarla, está en la parte back-end, en resources/bd/libraryOfOhara.sql cargar el SQL en MySQL (mejor en el mysqlWorkbench). si ya se tiene cargada la sql, podremos ver que esta la db libraryOfOhara con las tablas de usuario, libro, genero, autor y librosUsuarios.

Después de que las tablas estén creadas, debemos lanzar la aplicación back-end (en principio no debería de dar ningún problema).

Una vez lanzado la aplicación back, tendremos que preparar la aplicación front, que es la que más problemas puede darnos. En primer lugar, tendremos el código de la aplicación, pero debemos de irnos a la consola “ctrl+ñ” y confirmar que tenemos estar en la carpeta del proyecto (algo como: ./libraryOfOhara/) y aquí escribir el comando “npm -install”. este comando nos instalará todas las librerías necesarias para que la aplicación pueda ejecutarse.

3. Planificación y presupuesto

3.1 Panificación:

-Como ya os he explicado, dividí el proyecto en dos partes, la front-end y la back-end. Para poder planificar el tiempo, decidí dividir las 30 horas del TFG en 2.

Para la parte back-end, decidí emplear 10 horas ya que era "mas" sencilla. con más sencilla, me refiero a que es más directa (Ej. una petición para añadir, otra para eliminar, otra para modificar, otra para obtener datos, etc.).

Para la parte front-end, dejé como 20 horas aproximadamente. para esta parte, no sé cuánto tiempo me va a llevar ya que es la más subjetiva, en el sentido que depende de cómo quiera poner el estilo, pueda tardar más o menos.

3.2 Presupuestos

(los datos que voy a emplear, se basan en una nómina que tuve mientras trabajaba hace unos años.)

-Vamos a suponer que para un desarrollador de software junior cobra 930 €/mes y que, en un mes, el desarrollador realiza unas 40 horas semanales, y que un mes, tiene 4 semanas, véase que, en un mes, el desarrollador realiza 160 horas/mes.

En teoría, tenemos 30 horas aproximadamente de desarrollo del proyecto, pero debido a complicaciones durante el desarrollo, (previsible, es un junior quien realiza la aplicación) se nos va a +50 por lo que nos indica que por lo menos, ha estado más de 1 mes en desarrollo la aplicación. En su caso, requerimos de hacer 2 nominas diferentes.

Si contamos que las bases de contingencias comunes son de 4,8% y el IRPF es del 2%.

Nomina mes 1:

Total, devengado es de 930 € (he decidido no poner ningún plus ya que, al ser un junior, apenas tendría)

El descuento por las contingencias comunes es de: 930 * 4,8%= 44,64 €

El descuento de IRPF es de: 930 * 2%= 18,6€

La total deducción será de: 44,6 + 18,6= 63,24 €

Total, a percibir serán de: 866,77 €

Nomina mes 2:

Suponiendo que solo realice las horas faltantes véase (50-40) 10 horas, si en 1 mes se cobraba 930 € en 40 horas.

Total, de devengo: 930 * 10 / 40= 232,5 €

El descuento por las contingencias comunes es de: 232,5 * 4,8% = 11,16 €

El descuento de IRPF es de: 232,5 * 2% = 4,65€

La total deducción será de: 11,16 + 4,65 = 15,81 €

Total, a percibir serán de: 216,69 €

En total, en los 2 meses habrá percibido: 1083,46 €

4. Documentación técnica:

4.1. Análisis de sistema:

4.1.1. Especificación de requisitos:

Requisitos:

-Aplicación funcional (por supuesto) que emplee aquellos conocimientos que se hayan aprendido durante el curso y algunos nuevos por parte del usuario. En mi caso he utilizado Spring boot para el tema del servidor realizando peticiones que es lo que hemos visto durante las clases de Acceso a Datos (AAD) y angular js que es un lenguaje de desarrollo que aprendí por mi cuenta mientras realizaba el ciclo de grado superior de Desarrollo de Aplicaciones Web (DAW)

-Aplicación que gestione usuarios y libros que tenga la aplicación permitiendo que los usuarios puedan agregar aquellos libros que quieran leer o que ya hayan leído.

4.2. Diseño del sistema:

4.2.1. Diseño de la Base de Datos:

-La base de datos consta de 5 tablas como se ven en la imagen inferior. Las tablas son las de Genero, Autor, Libros, Usuarios y LibrosUsuarios respectivamente.

En la tabla Usuario, se guardarán los datos del usuario tales como: El nombre del usuario, el / los apellido/s, el correo electrónico la contraseña y el rol. Todos estos campos son de tipo Varchar ya que son textos siendo la contraseña la más larga debido a que estará cifrada. Además de estos, utilizo un id como identificador del mismo para facilitarme las búsquedas.

```sql
-- -----------------------------------------------------
-- Tabla `Library_of_ohara`.`Usuario`
-- -----------------------------------------------------
CREATE TABLE Usuario
(
    id          INT NOT NULL AUTO_INCREMENT,
    nombre      VARCHAR(30)  DEFAULT NULL,
    apellidos   VARCHAR(50)  DEFAULT NULL,
    gmail       VARCHAR(50)  DEFAULT NULL,
    contrasenna VARCHAR(500) DEFAULT NULL,
    rol         VARCHAR(10)  DEFAULT FALSE,
    PRIMARY KEY (id)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 AUTO_INCREMENT=1;
```

En la tabla Autor, tenemos el nombre, el / los apellido/s y la edad. Tanto el nombre como los apellidos son Varchar ya que son textos. En cambio, la edad es un numero entero. Además de estos campos, guardo también un id para identificar al autor para facilitarme las búsquedas.

```sql
-- -----------------------------------------------------
-- Tabla `Library_of_ohara`.`Autor`
-- -----------------------------------------------------

CREATE TABLE Autor
(
    id        INT NOT NULL AUTO_INCREMENT,
    nombre    VARCHAR(30) DEFAULT NULL,
    apellidos VARCHAR(50) DEFAULT NULL,
    edad      INT(3) DEFAULT 0,
    PRIMARY KEY (id)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 AUTO_INCREMENT=1;
```

En la tabla Género, tenemos el género en sí. Este, será de tipo Varchar ya que será un texto. Además, tengo un id como identificador para facilitarme a la hora de buscar.

```sql
-- -----------------------------------------------------
-- Tabla `Library_of_ohara`.`Genero`
-- -----------------------------------------------------

CREATE TABLE Genero
(
    id     INT NOT NULL AUTO_INCREMENT,
    genero VARCHAR(50) DEFAULT NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 AUTO_INCREMENT=1;
```

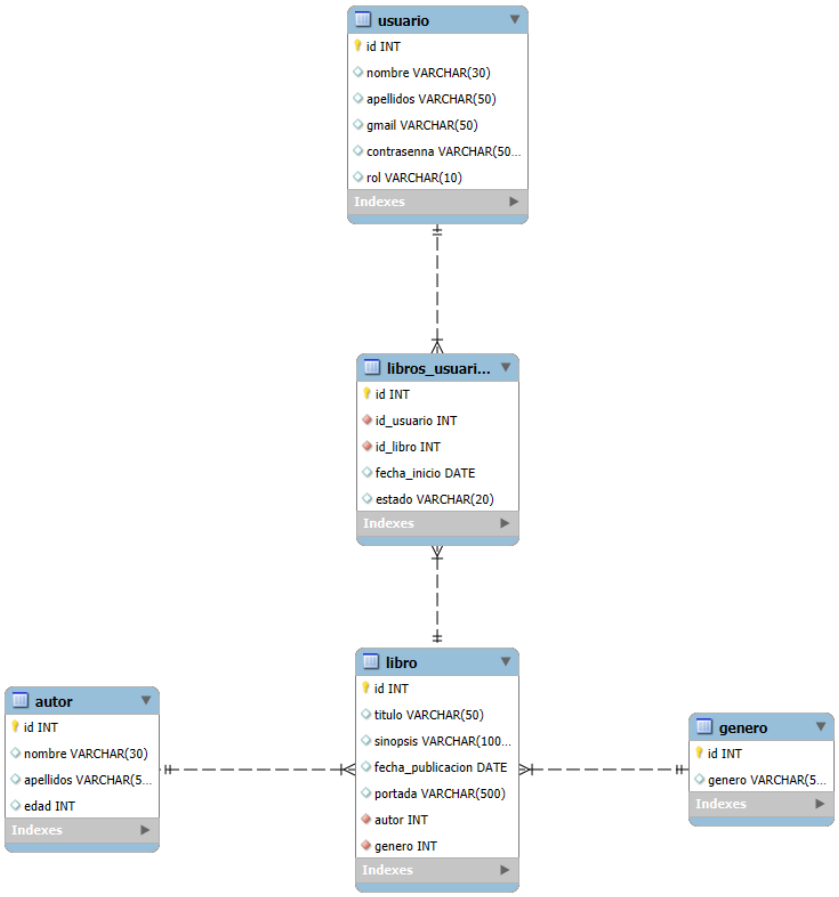
La tabla Libro que es donde se guardan los datos del libro tales como su título, su sinopsis, la fecha de publicación del libro y la portada. Tanto autor y el género serán claves foráneas que hacen referencia a las tablas de Autor y Género respectivamente. En esta tabla solo se guardará el id de dicho Autor o Género. Por último, la fecha de publicación será de tipo Date.

```sql
-- -----------------------------------------------------
-- Tabla `Library_of_ohara`.`Libro`
-- -----------------------------------------------------

CREATE TABLE Libro
(
    id               INT NOT NULL AUTO_INCREMENT,
    titulo           VARCHAR(50)   DEFAULT NULL,
    sinopsis         VARCHAR(1000) DEFAULT NULL,
    fecha_publicacion DATE         DEFAULT NULL,
    portada          VARCHAR(500)  DEFAULT NULL,
    autor            INT NOT NULL,
    genero           INT NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (autor) REFERENCES Autor (id),
    FOREIGN KEY (genero) REFERENCES Genero (id)
)ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 AUTO_INCREMENT=1;
```

En la tabla LibrosUsuarios, he querido realizar una relación entre los libros y los usuarios, pero a su vez, requería guardar otros tipos de datos. Al final, esta tabla guarda la fecha de inicio de lectura del libro por parte del usuario y el estado del mismo. Además, al igual que en la tabla Libro, guardo como una clave foránea tanto una referencia a la tabla Usuario como la tabla Libro respectivamente. De cada referencia, solo guardo el id.

```
-- -----------------------------------------------------
-- Tabla `Library_of_ohara`.`Libros_Usuarios`
-- -----------------------------------------------------

CREATE TABLE Libros_Usuarios
(
    id           INT NOT NULL AUTO_INCREMENT,
    id_usuario   INT NOT NULL,
    id_libro     INT NOT NULL,
    fecha_inicio DATE        DEFAULT NULL,
    estado       VARCHAR(20) DEFAULT "sin empezar",
    PRIMARY KEY (id),
    FOREIGN KEY (id_usuario) REFERENCES Usuario (id),
    FOREIGN KEY (id_libro) REFERENCES Libro (id)

)ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4 AUTO_INCREMENT=1;
```

Diagrama Entidad relación (ER):



4.2.2. Diseño de la Interfaz de Usuario:

-La parte front-end de la aplicación esta desarrollada a partir de Angular JS,faramework que se basa en TypeScript que es un "JavaScript tipado "(entre muchas comillas). Por tanto, se ejecuta en el navegador del cliente. La versión que he empleado es la versión 16 (o al menos, la que he tenido en cuenta), a partir de la versión 17 cambiaba ciertos aspectos de los cuales no llegue a entender bien y ya que conocía bien la versión anterior, pues me dedique a desarrollarlo en esa versión. Además, he de comentar que el estilo esta proporcionado por Bootstrap (en su mayoría) a la última versión (versión 5.3.6).

Una vez que arrancas el proyecto, nos llevará a la ruta localhost:4200/init, donde nos aparecerá el nombre de la aplicación "Library of Ohara" junto con una breve explicación de lo que puedes llegar a hacer. Además, nos saldrá un formulario a rellenar, en primera estancia será el formulario de login/inicio de sesión, pudiendo cambiar al de register/crear cuenta en caso de no se disponga de una cuenta.

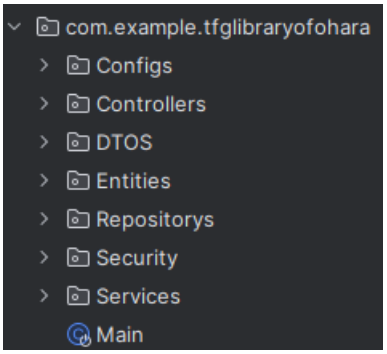
Una vez rellenados los formularios, si es correcto, nos navegará hasta la página del usuario en la que podremos ver todos aquellos libros que tenga en la lista o podremos ver la biblioteca que es donde estarán todos los libros de la aplicación. Al visualizar los libros estarán en formato de cartas ya que es el más visual a la hora de mostrar información. En cualquier carta, podremos navegar pulsándole al botón de "ver más", dicho enlace, nos llevará a la página de detalle del libro en el que nos mostrará más a detalle todos los datos de dicho libro. En esta ventana, nos permitirá pulsar a un botón para añadirlo a la lista o en su defecto, de eliminarlo de la lista si es que ese libro ya estuviera.

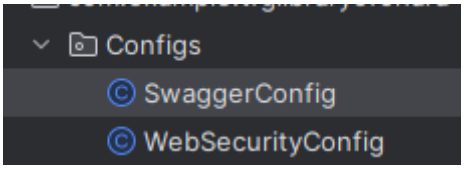
4.2.3. Diseño de la Aplicación:

-Para el diseño de mi aplicación, ya comenté que lo dividí en 2 partes, la parte front-end y la parte back end, por lo que me tendré que extender al explicar cada uno de los diseños. (no preocuparse porque he intentado hacerlo lo más parecido posible, salvando las distancias de ser aplicaciones desarrolladas en distintos lenguajes de desarrollo que funcionan de manera completamente diferente).

Para la parte back, he decidido dividir la aplicación en estas carpetas:

- Carpeta de configuración, que no confundir con la configuración del proyecto. En esta carpeta estarán los ficheros que he visto necesario para meter la configuración que he visto necesaria:





El primer fichero es el de la configuración del swagger en la que defino ciertas características de como se muestra el swagger. Además, las ultimas líneas sirven para dedicar un espacio en el swagger para poder autenticarse. De esta manera, puedo acceder a todos los métodos.

```java
KonoDIODa13
@Configuration
public class SwaggerConfig {
    // URL SWAGER: http://localhost:9999/doc/swagger-ui/index.html
    KonoDIODa13
    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
                .components(new Components())
                .info(new Info().title("library of Ohara API")
                        .description("API de gestión de Libros")
                        .contact(new Contact()
                                .name("Jaime González Bravo")
                                .email("jaime.gonbra@educa.jcyl.es")
                                .url("LibraryOfOharaAPI"))
                        .version("1.0")).addSecurityItem(new SecurityRequirement().addList( name: "JavaInUseSecurityScheme"))
                .components(new Components().addSecuritySchemes( key: "JavaInUseSecurityScheme", new SecurityScheme().name("JavaInUseSecurityScheme")
                        .type(SecurityScheme.Type.HTTP).scheme("bearer").bearerFormat("JWT"))
                );

    }
}
```

El segundo fichero es el de la configuración de la seguridad de la web. En dicho fichero, indico, que rutas están permitidas para todos los usuarios, y que rutas no.

```java
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
            .cors() // Habilita CORS
            .and()
            .addFilterAfter(new JWTAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class)
            .authorizeHttpRequests(authz -> authz
                    .antMatchers(HttpMethod.GET, "/api/usuario/login").permitAll()
                    .antMatchers(HttpMethod.POST, "/api/usuario/registrar").permitAll()

                    .antMatchers(AUTH_WHITELIST).permitAll()
                    .anyRequest().authenticated()
            );

    return http.build();
}
```

Si preguntais por el Auth_Whitelist es una variable que contiene todas las rutas para el swagger.

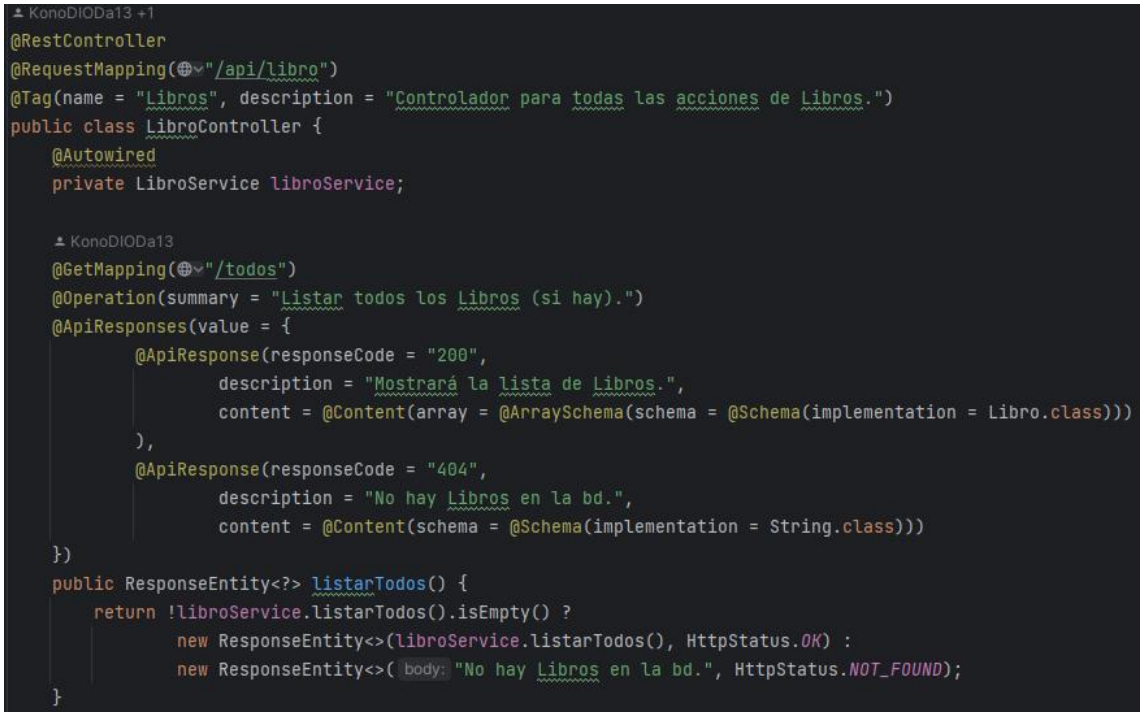
```
    1 usage
    private static final String[] AUTH_WHITELIST = { //SWAGGER
// -- Swagger UI v2
            "/v2/api-docs",
            "/swagger-resources",
            "/swagger-resources/**",
            "/configuration/ui",
            "/configuration/security",
            "/swagger-ui.html",
            "/webjars/**",
// -- Swagger UI v3 (OpenAPI)
            "/v3/api-docs/**",
            "/swagger-ui/**",
            "/doc/**"
// other public endpoints of your API may be appended to this array
    };
```

Por último, en este fichero, permito las peticiones desde la dirección de "localhost:4200" que, si recordáis, es la ruta de mi aplicación Front-end. Además, permito que sean de los tipos importantes de las peticiones (GET, POST, DELETE PUT/PATCH) y para que envíe también en el header, el token de autorización.

```
KonoDIODa13
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of( e1: "http://localhost:4200"));
    config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "PATCH"));
    config.setAllowedHeaders(List.of( e1: "*"));
    config.setExposedHeaders(List.of( e1: "Authorization"));
    config.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration( pattern: "/**", config);
    return source;
}
```

- Carpeta de los controladores: Mi idea es la de crear un controlador por entidad que tenga en la aplicación. Pero debido a que la tabla Librosusuarios dependía demasiado de usuarios, decidí meter todo dentro del controlador de Usuario. En los controladores he metido todas aquellas peticiones que se pueden realizar para cada entidad y de cada uno de los tipos (peticiones GET, POST, PUT/PATCH, DELETE, etc.). Para no extender mucho la explicación, solo detallaré un controlador y algunas peticiones.

En la imagen inferior, podemos ver el inicio de el controlador de libro, en el que se puede ver tanto la ruta general para todas las acciones de dicho constructor (/api/libro) como el servicio de los libros. El autowired es para que se conecte automáticamente en vez de hacer un constructor para dicho servicio.
La primera funcionalidad de este controlador es la de conseguir todos los libros. Como podéis ver, las primeras líneas sirven para decir que tipo de petición es (GET) con que ruta (/todos). Después, muestra las posibles respuestas con sus códigos. Estos, solo harán que, en el swagger, se vea de esta manera. No va a modificar la respuesta, solo que, al ver la petición en el swagger, se muestren las posibles respuestas con sus respectivos códigos. Esto es algo que quise mantener ya que mejorarían el aspecto del swagger. Por último, en la imagen se ve que toda esta configuración esta para listar todos los libros. En función de que respuesta me de la función de listarTodos del servicio, mostrare una lista de libros o un texto diciendo que no hay libros en la lista.



```
KonoDIODa13 +1
@RestController
@RequestMapping("/api/libro")
@Tag(name = "Libros", description = "Controlador para todas las acciones de Libros.")
public class LibroController {
    @Autowired
    private LibroService libroService;

    KonoDIODa13
    @GetMapping("/todos")
    @Operation(summary = "Listar todos los Libros (si hay).")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200",
                    description = "Mostrará la lista de Libros.",
                    content = @Content(array = @ArraySchema(schema = @Schema(implementation = Libro.class)))
            ),
            @ApiResponse(responseCode = "404",
                    description = "No hay Libros en la bd.",
                    content = @Content(schema = @Schema(implementation = String.class)))
    })
    public ResponseEntity<?> listarTodos() {
        return !libroService.listarTodos().isEmpty() ?
                new ResponseEntity<>(libroService.listarTodos(), HttpStatus.OK) :
                new ResponseEntity<>( body: "No hay Libros en la bd.", HttpStatus.NOT_FOUND);
    }
```

En la segunda imagen, podemos ver el método para guardar un nuevo libro en la base de datos. En dicho método, en función de si nos devuelve un libro o no, nos mostrara dicho libro que es el que se ha guardado en base de datos o nos mostrará el texto de que dicho libro ya existe en la base de datos.

```
KonoDIODa13
@PostMapping("/registrar")
@Operation(summary = "Añadir un nuevo Libro")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200",
                description = "Añadirá correctamente el Libro con los datos pasados.",
                content = @Content(schema = @Schema(implementation = Libro.class))
        ),
        @ApiResponse(responseCode = "409",
                description = "Dicho Libro ya existe en la bd.",
                content = @Content(schema = @Schema(implementation = String.class)))
})
public ResponseEntity<?> registrar(@RequestBody LibroDTO libroDTO) {
    Libro libro = libroService.insertar(libroDTO);
    return libro != null ?
            new ResponseEntity<>(libro, HttpStatus.OK) :
            new ResponseEntity<>( body: "Dicho Libro ya existe en la bd.", HttpStatus.CONFLICT);
}
```

No quiero meterme en mas detalles porque si no, alargaría innecesariamente esta memoria.

- En la carpeta de los DTOS: se guardarán unas clases bastante parecidas a las entidades de mi aplicación, solo que no disponen de un identificador y en el caso de Libro, ya que tienen dos referencias a otras clases, solo requerirá de los ids de las otras clases a las que se referencia. En el caso del UsuarioDTO, decidí meter también la codificación de la contraseña para protegerla, sobre todo. Utilizo estas clases a la hora de insertar nuevos campos o a la hora de modificar (debido a que para modificar siempre paso el id por la url, puedo saber si existe o no dicho registro para modificarlo). Aquí abajo, muestro un ejemplo de libroDTO. En dicho DTO, tengo todos los datos del libro, sin el identificador de libro y los ids del autor como del género. En dicha clase, tengo el método para cambiar de DTO a entidad o modelo que lo que me devuelve es el libro. También, tengo el método para crear un nuevo DTO cuando viene de un libro. Dichos métodos, son parte de algo que quería realizar, pero por falta de tiempo y conocimiento, solo empleo el que me devuelve el libro. Quería responder a todas las peticiones con los dtos siendo estos los únicos que tendrían contacto con el exterior protegiendo así los datos internos de la aplicación.

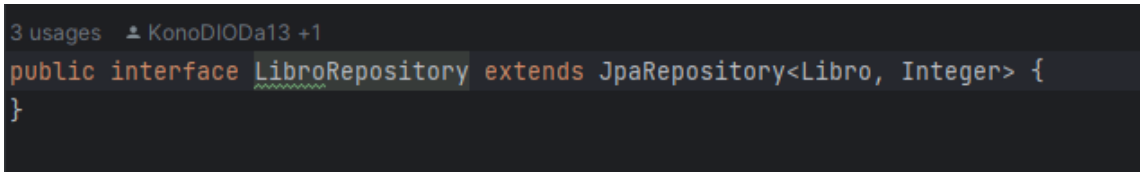
```java
9 usages  KonoDIODa13 *
@Data
@NoArgsConstructor
@AllArgsConstructor
public class LibroDTO {
    private String titulo;
    private String sinopsis;
    private LocalDate fechaPublicacion;
    private String portada;
    private int idAutor;
    private int idGenero;


    1 usage  KonoDIODa13 *
    public Libro DTOtoModel() {
        Libro libro = new Libro();
        libro.setTitulo(titulo);
        libro.setSinopsis(sinopsis);
        libro.setFechaPublicacion(fechaPublicacion);
        libro.setPortada(portada);
        return libro;
    }

    no usages  KonoDIODa13 *
    public LibroDTO modeltoDTO(Libro libro) {
        return new LibroDTO(libro.getTitulo(),
                libro.getSinopsis(),
                libro.getFechaPublicacion(),
                libro.getPortada(),
                libro.getAutor().getId(),
                libro.getGenero().getId()
        );
    }
}
```

- En la carpeta de las entidades: estarán guardadas las entidades de mi aplicación, junto con todas las relaciones entre ellas mismas. Por ello, considero que esta es la parte más complicada de la aplicación ya que dependía de que esto funcionara completamente. Tuve muchos problemas con la entidad de librosusuarios (como no) ya que tenía la idea principal de hacerla como una relación varios a varios (n:n) pero luego vi la oportunidad de guardar más información y decidí cambiarla a una tabla con más campos. Voy a mostrar el de Libro ya que es uno de los más

complicados. En dicha entidad, os muestro, todas las configuraciones que se tienen en cuenta a la hora de crear una entidad de mi aplicación. Como podemos ver, están todas estas etiquetas de lombok que facilitan el uso de esta. En el toString, excluyo esas características ya que eran las que más podían complicar el código. Como se puede ver, esta clase tiene la etiqueta de @Entity para indicar que es una entidad junto con la de @Table que indica a que tabla hace referencia. En cada uno de los campos como podemos ver, tienen la columna de la tabla a la que hacen referencia. El id, además de la columna, indicamos que hibernate, autoincremente a la hora de crear uno nuevo. En autor como en género, ya que es una relación con las otras clases, requiere de @ManyToOne ya que varios libros pueden pertenecer a un género, pero un libro solo tiene un género. En dichas relaciones tengo puesto la etiqueta de @JsonIgnoreProperties ya que, en dichas entidades, esta el campo libros que hace referencia a los libros con los que se tiene en cuenta. Para evitar que se cree un bucle infinito intentado sacar los libros, hago esto para evitarlo. Con el autor, es igual la teoría, pero la relación con librosUsuarios es diferente. Esta, como ya comenté, en principio, esa clase iba a ser una relación varios a varios (n:n) pero al final la planteé como una entidad normal. En el caso de libro, sería una @OneToMany.

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString(exclude = {"autor", "genero", "librosUsuarios"})
@Entity
@Table(name = "Libro")
public class Libro {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "titulo")
    private String titulo;

    @Column(name = "sinopsis")
    private String sinopsis;

    @Column(name = "fecha_publicacion")
    private LocalDate fechaPublicacion;

    @Column(name = "portada")
    private String portada;

    @ManyToOne
    @JoinColumn(name = "autor", referencedColumnName = "id")
    @JsonIgnoreProperties("libros")
    private Autor autor;

    @ManyToOne
    @JoinColumn(name = "genero", referencedColumnName = "id")
    @JsonIgnoreProperties("libros")
    private Genero genero;

    @OneToMany(mappedBy = "libro", cascade = CascadeType.ALL, orphanRemoval = true)
    @JsonIgnore
    private List<LibrosUsuarios> librosUsuarios;
}
```

- La carpeta de repositorios: en dicha carpeta, están los repositorios de la aplicación que dan toda la funcionalidad al programa, ya que, en ellas, están todas las operaciones que se van a realizar en la base de datos, ya sean guardar nuevos registros, borrarlos, modificarlos o visualizarlos. Mostrare el repositorio del Libro. En dicho repositorio, como podéis ver, no empleo funcionalidad más allá de el JpaRepository que es una clase que crea toda la funcionalidad que pueda llegar a tener un crud. Para poder especificarle de que es el crud, solo tienes que pasarle tanto la clase como el tipo de dato que es el identificador de la clase.



- La carpeta de seguridad: en ella, está la creación del token de autorización. En el único fichero que está aquí dentro, genero todo lo que necesita para poder crear un token.

```java
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

    String header = request.getHeader(HttpHeaders.AUTHORIZATION);
    if (header == null || !header.startsWith("Bearer ")) {
        chain.doFilter(request, response);
        return;
    }

    String token = header.replace( target: "Bearer ", replacement: "");
    try {
        Claims claims = Jwts.parser() JwtParser
                .setSigningKey("mySecretKey".getBytes())
                .parseClaimsJws(token) Jws<Claims>
                .getBody();

        String username = claims.getSubject();
        List<GrantedAuthority> authorities;

        Object rawAuthorities = claims.get("authorities");

        if (rawAuthorities instanceof List<?>) {
            List<?> roles = (List<?>) rawAuthorities;
            if (!roles.isEmpty() && roles.get(0) instanceof String) {
                authorities = roles.stream() Stream<capture of ?>
                        .map(role -> new SimpleGrantedAuthority((String) role)) Stream<SimpleGrantedAuthority>
                        .collect(Collectors.toList());
            } else {
                authorities = roles.stream() Stream<capture of ?>
                        .map(role -> new SimpleGrantedAuthority(((Map<?, ?>) role).get("authority").toString())) Stream<SimpleGrantedAuthority>
                        .collect(Collectors.toList());
            }
        } else {
            authorities = List.of(); // vacío, por si acaso
        }
        Authentication auth = new UsernamePasswordAuthenticationToken(username, credentials: null, authorities);
        SecurityContextHolder.getContext().setAuthentication(auth);
    } catch (Exception e) {
        SecurityContextHolder.clearContext();
    }
    chain.doFilter(request, response);
}
```

- Por último, la carpeta de los servicios. Yo utilizo los servicios como conexión entre el controlador y el repositorio, por ello, toda la funcionalidad estará aquí, todas las comprobaciones y los valores de retorno están configurados aquí. Al igual que en los controladores, no tengo uno para la clase LibroUsuarios ya que está dentro del de usuario. Mostraré el servicio de Libro. En dicho servicio, podemos ver que tenemos la etiqueta de @service diciéndole a Spring que esto trátalo como un servicio. Además de todos los repositorios que requiero para el tema de los libros que son los del libro en sí, el del autor y el del género. Además, están los métodos para listar todos los libros mediante un método del repositorio. También, está el método por el cual busca un libro mediante el id que se le pasa.

```java
@Service
public class LibroService {
    @Autowired
    private LibroRepository libroRepository;
    @Autowired
    private AutorRepository autorRepository;
    @Autowired
    private GeneroRepository generoRepository;

    10 usages  KonoDIODa13
    public List<Libro> listarTodos() { return libroRepository.findAll(); }

    6 usages  KonoDIODa13
    public Optional<Libro> buscarXID(int idLibro) {
        return libroRepository.findById(idLibro);
    }
```

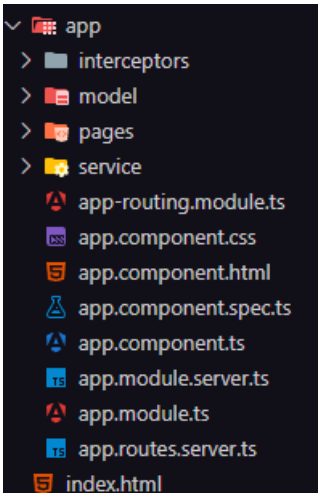
En la imagen inferior, podemos ver como el programa insertaría un nuevo libro. Para ello, primero comprueba si existe el autor y el genero ya que recordemos, en el DTO solo tenemos los ids, no el objeto como tal. Una vez comprobados que dichos datos no sean nulos, le agregamos a un libro que hemos instanciado en dicha función, los datos que tenemos en el DTO, mediante la función del mismo en la que nos devuelve un libro con los datos del DTO. El de modificar libro, es parecido solo que devuelve un boolean. Si ha modificado el registro, devuelve un true y en caso de que no lo ha modificado, devuelve un false.

```java
1 usage  KonoDIODa13
public Libro insertar(LibroDTO libroDTO) {
    Libro libro = null;
    Autor autor = comprobarAutor(libroDTO.getIdAutor());
    Genero genero = comprobarGenero(libroDTO.getIdGenero());
    if (!comprobar(libroDTO) && autor != null && genero != null) {
        libro = libroDTO.DTOtoModel();
        libro.setAutor(autor);
        libro.setGenero(genero);
        save(libro);
    }
    return libro;
}

1 usage  KonoDIODa13 *
public boolean modificar(Libro libro, LibroDTO libroDTO) {
    Autor autor = comprobarAutor(libroDTO.getIdAutor());
    Genero genero = comprobarGenero(libroDTO.getIdGenero());
    if (autor != null && genero != null) {
        libro.setTitulo(libroDTO.getTitulo());
        libro.setSinopsis(libroDTO.getSinopsis());
        libro.setFechaPublicacion(libroDTO.getFechaPublicacion());
        libro.setPortada(libroDTO.getPortada());
        libro.setAutor(autor);
        libro.setGenero(genero);
        save(libro);
        return true;
    }
    return false;
}
```

Por si hubiera alguna duda, a la hora de realizar las comprobaciones, empleo estos métodos en los que, en el caso de que compruebe el libro entero, busco si dicho título existe en algún registro. Si encuentra uno, devuelvo true, en caso contrario devuelvo false. Tanto el compruebaGenero como el compruebaAutor, son iguales pero devuelven el objeto que encuentran a la hora de buscar un genero o un autor mediante su id.

```
1 usage  KonoDIODa13
private boolean comprobar(LibroDTO libroDTO) {
    boolean existe = false;
    Libro optLibro = listarTodos().stream()
            .filter(libro ->
                    libro.getTitulo().equalsIgnoreCase(libroDTO.getTitulo()))
            .findFirst().orElse( other: null);
    if (optLibro != null) {
        existe = true;
    }
    return existe;
}

2 usages  KonoDIODa13
private Autor comprobarAutor(int idAutor) {
    Autor autor = null;
    Optional<Autor> optAutor = autorRepository.findById(idAutor);
    if (optAutor.isPresent()) {
        autor = optAutor.get();
    }
    return autor;
}

2 usages  KonoDIODa13
private Genero comprobarGenero(int idGenero) {
    Genero genero = null;
    Optional<Genero> optGenero = generoRepository.findById(idGenero);
    if (optGenero.isPresent()) {
        genero = optGenero.get();
    }
    return genero;
}
```

En la parte front, decidí dividirlo en estas carpetas:



Ya que utilizo angular, toda la funcionalidad está metida en la carpeta app, no preocuparse por todos esos ficheros que están fuera de las demás carpetas, ya que, son ficheros para definir tanto el html de inicio, como el estilo general, como la configuración respecto a las rutas como a los elementos que tiene dicha aplicación.

- La primera carpeta es la carpeta de los interceptores, algo que nunca había hecho antes así que es una modalidad nueva. Estos archivos, como su nombre indican, interceptan las peticiones que se realizan para poder añadir ciertas características. En mi caso, es el tema del token de autorización para permitir el acceso a las peticiones.

  En la imagen inferior, podemos ver el interceptor que he creado que gestiona las peticiones. Si para cuando una petición es lanzada, este archivo se ejecutará para comprobar si se tiene guardado el token o no y en función de eso, devolverá en la cabecera de la petición, el token de autenticación o no.

```typescript
import { Injectable } from '@angular/core';
import { HttpEvent, HttpHandler, HttpInterceptor, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = sessionStorage.getItem('token');

    if (token) {
      const authReq = req.clone({
        setHeaders: {
          Authorization: token
        }
      });
      return next.handle(authReq);
    }
    return next.handle(req);
  }
}
```

- La carpeta del modelo es la carpeta destinada a todas las clases que utilizo en la aplicación, es en esencia, una copia de la carpeta de entidades de la aplicación back-end solo que con los tipos dedicados de TypeScript.

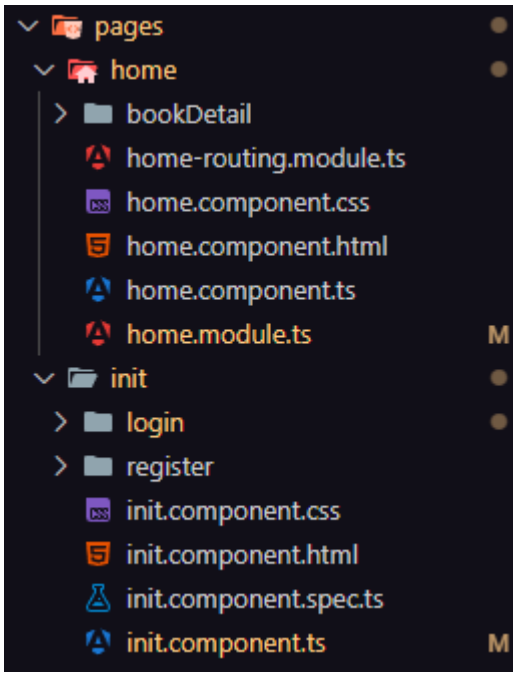
  En mi caso, como se puede ver en la imagen inferior, tenemos la clase Libro que es un espejo de la clase Libro de la aplicación back-end. Quiero dar incapie en el hecho de que el id es “no necesario”, o mejor dicho, puede ser nulo debido a que

en la aplicación front-end, no hay DTOs por lo que mi manera de comunicar la información a lo largo y ancho de la aplicación.

```
export class Libro {
  constructor(
    public titulo: string,
    public sinopsis: string,
    public fechaPublicacion: Date,
    public genero: Genero,
    public autor: Autor,
    public portada: string,
    public id?: number,
  ) { }
```

- La carpeta de páginas que es donde estarán todas las vistas del programa, en mi caso, esta de subdivide en varias otras carpetas que a su vez se dividen en otras subcarpetas. Angular se puede definir lo que se llaman componentes, cada componente realiza una acción o es el resultado de una vista, por ello, se permite el uso de un componente más genérico o incluso la repetición de un componente (véase, una lista de cartas).

  En mi caso, decidí dividir en 2 el tema de las páginas:

  Como podéis ver en la imagen, tenemos el directorio home que es donde están todas las páginas relacionadas a la página del usuario. Reconozco, que debería de haber más componentes, pero por la falta de tiempo, tuve que acortar recursos. Uno de los componentes es el del detalle del libro. He de comentar también que, en dicho directorio, tiene su propio module que sirve para poder encapsular cierta información importante relacionada a que librerías puede utilizar los componentes a los que tiene acceso.

  Además, tengo el directorio de “init” o inicio que es donde comenzara la aplicación, donde veremos dos componentes, el de inicio de sesión y el de crear cuenta. Vamos a ver el ejemplo del componente login. A la hora de crear nuevas vistas, Angular lo llama componente que agrupa tanto el estilo, como la vista, como el fichero que contiene toda la funcionalidad de la vista. Como se puede ver en la imagen inferior, podemos encontrar cierta configuración del propio componente tales como, el nombre de la etiqueta, la url donde estará la vista y el

estilo y lo de estandalone hay que ponerlo en false porque estoy trabajando como si fuera la versión 16.

```
@Component({
  selector: 'init-login',
  standalone: false,
  templateUrl: './login.component.html',
  styleUrl: './login.component.css'
})
```

En la imagen inferior veremos el componente, la parte programática por así decirlo, el JavaScript de la vista. Podemos ver la forma de crear y controlar un formulario reactivo. También podemos ver como se maneja la petición de logear. En caso de que la respuesta sea positiva, comprobamos el token de autorización que recibimos y lo guardamos en la sesión para tenerlo durante el uso de la aplicación junto con el usuario que nos devuelve y navegamos hasta la página de usuario. En caso contrario, mostramos un mensaje de error.

```
export class LoginComponent {

  usuario?: Usuario

  constructor(private service: LibraryOfOharaService,
    private router: Router) { }

  public loginForm = new FormGroup({
    nombre: new FormControl('', Validators.required),
    contrasenna: new FormControl('', Validators.required)
  });


  enviar() {
    const nombre = this.loginForm.value.nombre!
    const contra = this.loginForm.value.contrasenna!

    this.service.login(nombre, contra).subscribe({
      next: response => {
        const authHeader = response.headers.get('Authorization');
        if (authHeader) {
          this.usuario = response.body!;
          sessionStorage.setItem('usuario', JSON.stringify(this.usuario));
          sessionStorage.setItem('token', authHeader);
          this.router.navigate(['/home']);
        } else {
          alert("No se recibió token de autenticación.");
        }
      },
      error: () => {
        alert("Credenciales inválidas.");
      }
    });
  }
}
```

- La última carpeta, es la de los servicios, en ella, estarán todas las peticiones que esta aplicación realiza. Las junte todas en un solo fichero ya que me era más cómodo a la hora de centralizar todas las llamadas a un solo fichero. Como podéis ver en la imagen, necesitamos instanciar el clienteHTTP para poder realizar las peticiones y muestro como ejemplo, alguna de las peticiones. Entre las peticiones, esta la de recibir todos los usuarios que como veis, devuelve un Observable que es un tipo de objeto de angular para las peticiones y le estamos diciendo que cuando acabe la petición, la respuesta ha de ser una lista de usuarios (o al menos en ese caso). En cambio, los otros como necesito acceder a la cabecera de la petición de respuesta, tiene que ser un HttpResponse y dentro de la petición poner el observe:'response'.

```
export class LibraryOfOharaService {

  private url = "http://localhost:9999/api/";

  constructor(private http: HttpClient) { }

  public getUsuarios(): Observable<Usuario[]> {
    return this.http.get<Usuario[]>(this.url + "usuario/todos")
  }

  public login(nombre: string, contra: string): Observable<HttpResponse<Usuario>> {
    const newUrl = this.url + "usuario/login?nombreUsuario=" + nombre + "&contrasenna=" + contra
    return this.http.get<Usuario>(newUrl, {
      observe: 'response'  // Para poder leer las cabeceras
    });
  }

  public register(usuario: Usuario): Observable<HttpResponse<Usuario>> {
    const newUrl = this.url + "usuario/registrar"
    return this.http.post<Usuario>(newUrl, usuario, {
      observe: 'response'  // Para poder leer las cabeceras
    });
  }
```

4.3. Implementación:

4.3.1. Entorno de Desarrollo:

-Como ya he comentado antes, la aplicación está dividida en 2 partes, la parte front-end y la parte back-end. En la parte back-end empleo java, más concretamente el framework de Spring Boot, para utilizarlo como una API a la que hacer peticiones (GET, POST, PUT, DELETE, etc.). Para poder desarrollarlo y ejecutarlo, utilicé Intellij que es uno de los IDEs más utilizados actualmente para desarrollo ya que es libre (aunque la versión que te permite lanzar proyectos de Spring Boot no).

Para la parte front-end, utilicé Angular JS que es un framework de TypeScript que en esencia es un JavaScript tipado. Para poder desarrollarlo y ejecutarlo, usé el IDE de Visual Studio Code (VSC) que es el más ligero y que más conozco, además, debido al ciclo de Desarrollo de Aplicaciones Web (DAW) que realicé con anterioridad, tengo algunas librerías instaladas para facilitarme el desarrollo de código en Angular (y en más lenguajes de desarrollo ya que VSC te permite meter y desarrollar en muchos lenguajes).

4.3.2. Estructura del código:

4.3.2.1. Librerías empleadas:

-Para la parte Back he empleado las librerías para poder emplear APIs.

Una de las librerías que he utilizado es la de Lombok que es una librería para facilitar el código (sobre todo en las clases). Esta librería permite crear automáticamente todos los getters / setters de los atributos de la clase como los constructores, ya sea el constructor vacío como el constructor con todos los parámetros. Como podéis ver en la imagen inferior, veréis algunos de las muchas utilidades de esta librería

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString(exclude = {"libros"})
@Entity
@Table(name = "Autor")
public class Autor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private int id;

    @Column(name = "nombre")
    private String nombre;

    @Column(name = "apellidos")
    private String apellidos;

    @Column(name = "edad")
    private int edad;

    @OneToMany(mappedBy = "autor", cascade = CascadeType.ALL)
    @JsonManagedReference
    private List<Libro> libros;
}
```

Otra de las librerías empleadas es la de “commons-codec” que utilizo principalmente para codificar las contraseñas, de esta manera, protejo dicha información.

```
2 usages   KonoDIODa13
public void codificarContra() { contrasenna = DigestUtils.sha256Hex(contrasenna); }
```

Además, utilizo todas las librerías necesarias para la conexión de base de datos como la de mysql. Por último, utilizo librerías para permitir que mi aplicación sea una API. (openapi)

-Para la parte Front, algunas librerías como la de los formularios reactivos (ReactiveFormsModule) que como su nombre indica que sirve para poder controlar mejor el tema de los formularios.

```
public loginForm = new FormGroup({
  nombre: new FormControl('', Validators.required),
  contrasenna: new FormControl('', Validators.required)
});
```

También utilizo la librería de HTTP_INTERCEPTORS para habilitar los interceptores junto con el de HttpClientModule que me permite realizar peticiones. Esta librería, si os fijas esta “deprecada” debido a que como ya expliqué con anterioridad, empleo una versión anterior a la que está actualmente, dicha versión, emplea la versión actualizada de esta. Además, empleo la famosísima librería de Bootstrap. Yo me considero un negado a la hora de estilizar una aplicación, (a mí me gusta la funcionalidad) por ello, esta librería me ayuda a que mis aplicaciones estén mejores diseñadas o mejor estilizada.

4.3.2.2. Metodología que se ha seguido a la hora de desarrollar:

-He querido seguir el ideal de Programación Orientada a Objetos (POO) ya que es el que, aprendido, pero a su vez, las APIs, requiere de Programación Orientada a Eventos ya que, solo se ejecuta cuando se le llama a la petición. Además, en las aplicaciones web generalmente, funcionan con la metodología orientada a eventos, pero incluso aquí, también he empleado objetos a la hora de organizar la información y, sobre todo, para la hora de recibir información de parte del back e incluso a la hora de enviarla. Por tanto, en esta aplicación, se emplean los dos tipos.

4.3.2.3. Cuestiones de diseño e implementación dignos a mencionar:

-En general, la aplicación es fácil de seguir y se basan en cómo hemos desarrollado las APIs durante el curso (+ algunos añadidos de mi forma de trabajar). Algunos de esos añadidos son:

-El uso de streams para las listas. Stream es una manera de recorrer listas en las cuales puedes modificar la salida para que, en vez de devolver una lista, sea un Set o solo el objeto o un entero o incluso un boolean. Esto, se puede hacer también recorriéndolo con for/foreach pero esta es la forma más actualizada. Esto junto a la decisión de desarrollo de trabajar siempre con listas, yo en vez de buscar uno en concreto, lo que hago es pasarme todos, y después buscar el que se necesite/requiera.

-El CORS: el cors es lo que protege/da problemas cuando intentas comunicar desde una aplicación a otra (en este caso, el front al back). Resulta que se tiene que definir desde que url se puede acceder (en mi caso, localhost:4200) y para un tema de autorización que veremos más adelante.

-El uso de DTOs: a la hora modificar/añadir nuevos objetos para la base de datos, los objetos DTOs serán (en su mayoría) igual que los objetos, pero sin el id del mismo. Esto es así para poder facilitar a la hora de insertar nuevos campos y para editar uno, como yo empleo unas URLs donde siempre se requiere el id pues con eso puedo comprobar si existe.

-El uso de Interceptors: como su nombre indica, interceptan las peticiones que se realicen. En mi caso, todas las peticiones, salvo alguna (login/register) están restringidas para evitar que se empleen de mala manera. Para poder restringirlo, he empleado el uso del token de autenticación Bearer, que es el que hemos aprendido. El problema era que no sabía cómo realizarlo, una de las maneras era meter dicho token en todas y cada una de las peticiones, pero, con la existencia de estos interceptores, me permite interceptar la petición y añadir, en este caso, el token de autorización.

4.3.2.4. Pruebas:

En este apartado no tengo mucho que decir la verdad ya que, si tenía algo que probar, por ejemplo, que todas las peticiones funcionaran, me ponía y las probaba una a una. Se que no es la mejor manera, pero si la más rápida, de esta manera, no tengo ningún test, más allá de los que genera automáticamente el angular (creo que solo es la clase para meter las pruebas que uno quiera, imagínate cómo funciona que ni siquiera sé que es xD).

Por la parte de back, aunque si he hecho algún test, comprobando que si las funciones de un servicio funcionan o que las peticiones de un controlador sí que devuelven lo realizado.

5. Manuales:

5.1. Manual de usuario:

-Antes de empezar indicando toda la funcionalidad que tendrá un usuario empleando esta aplicación, comentar que este manual empezará una vez que ya se tenga la aplicación en total funcionamiento, en caso contrario, se relataran al final de este documento tanto el manual de instalación como el manual de despliegue. Una vez comentado esto, procederé a explicar toda la funcionalidad.

Bienvenidos a todos los nuevos usuarios de la aplicación "Library Of Ohara" con la cual los usuarios podrán gestionar aquellos libros que hayan leído o que quieran leer en un futuro.

Como página inicial, se le dará le bienvenida a esta maravillosa aplicación. A la derecha de la pantalla, habrá un formulario que según si tenemos una cuenta o no, nos pedirá que lo rellenemos.

En caso de que tengamos una cuenta, nos pedirá que rellenemos el formulario de login/iniciar sesión, en el que nos pedirá tanto el nombre del usuario, como la contraseña del usuario. Una vez rellenados los campos, podremos darle a enviar, en caso de que el formulario este correcto, el usuario accederá directamente a la página del usuario.

En caso de que no tengamos una cuenta, tendremos que rellenar el formulario de register/crear cuenta. En este formulario, se requiere de los parámetros de nombre del usuario, apellidos, correo electrónico, y contraseña, todos ellos requeridos. Una vez rellenado y enviado, si la respuesta es correcta, el usuario accederá a la página del usuario.

En caso de que, de error en cualquiera de los formularios, nos mostrará una alerta de error en el formulario correspondiente.

Una vez dentro de la página del usuario, nos mostrará, si tenemos, los libros que hayamos guardados en nuestra lista. En caso de no tener, no preocuparse, ya que, podremos acceder a la parte de biblioteca donde podremos ver todos aquellos libros que la aplicación tenga a su disposición.

En la biblioteca como hemos mencionado ya, tenemos los libros que nos mostrará tanto la imagen de la portada, como el titulo y si queremos ver más detalles del propio libro.

Al ver el detalle del libro, nos mostrará toda la información del libro en cuestión. Dicha información, será la de título del libro, el autor del libro, el género al que pertenece el libro, la fecha de publicación del libro y una breve sinopsis/resumen del mismo (sin spoilers). Además, nos permitiría tanto volver a la ventana anterior y añadir dicho libro. En caso de que dicho libro exista ya en la lista, podremos ver tanto el estado de lectura del mismo y su fecha de inicio, en caso de que se haya empezado a leer.

En la página de usuario también el usuario puede deslogearse / desconectarse (si así lo quiere) pulsando a la imagen arriba a la izquierda (no sé si para este punto, tendré alguna funcionalidad más en este apartado).

5.2. Manual de instalación:

-Para instalar la aplicación, primero, al igual que en el despliegue, debemos bajarnos del repositorio Git, las dos partes, la de front, como la de back.

9. Bibliografía

• Bibliografía y Webgrafía.

Anexos para descargarlos, tan solo hay que pulsar a donde pone lo de “code” en verde y a “download ZIP” (preferiblemente hacerlo con el zip, también se puede clonarlo), una vez descargados, hay que descomprimirlos, para ello, utilice la herramienta que tenga disponible (recomendable en zip) y colóquelos en donde quiera.

Después, tendríamos que ejecutar el código del sql para cargar la base de datos, para localizarla, está en la parte back-end, en resources/bd/libraryOfOhara.sql cargar el SQL en MySQL (mejor en el mysqlWorkbench). si ya se tiene cargada la sql, podremos ver que esta la db libraryOfOhara con las tablas de usuario, libro, genero, autor y librosUsuarios.

Después de que las tablas estén creadas, debemos lanzar la aplicación back-end (en principio no debería de dar ningún problema).

Una vez lanzado la aplicación back, tendremos que preparar la aplicación front, que es la que más problemas puede darnos. En primer lugar, tendremos el código de la aplicación, pero debemos de irnos a la consola “ctrl+ñ” y confirmar que tenemos estar en la carpeta del proyecto (algo como: ./libraryOfOhara/) y aquí escribir el comando “npm -install”. este comando nos instalará todas las librerías necesarias para que la aplicación pueda ejecutarse.

5.3 Manual de despliegue:

-Para desplegar la aplicación, en caso de que no dispongamos del código, mirar el manual de instalación más adelante.

Una vez teniendo cargado el sql, podremos lanzar la aplicación back-end, podremos comprobar que todo va bien si no ha dado ningún error o si nos vamos a la url del swagger y carga todos los datos.

una vez cargada la aplicación back-end, tendemos que lanzar la aplicación front-end, yo utilizo el comando en la terminal de la carpeta padre del proyecto (cuando se abre con VSC, -con pulsar “ctrl+ñ” nos abre la terminal o bien, yendo a la terminal directamente) y poner el comando de ng serve -o para lanzar la aplicación (si es la primera vez, nos pide una confirmación, lo hace siempre que creamos un nuevo proyecto de angular, le decimos que Sí) le pongo el -o para que nos abra directamente el navegador con la página inicial del proyecto.

6. Conclusiones y posibles ampliaciones:

6.1 Conclusiones:

-Como objetivo principal del proyecto, es la de crear una aplicación web que utilice peticiones a un servidor diferenciando así, la parte front-end de la parte back-end. Esa parte, que, para mí, era la más importante a nivel de demostrar mis habilidades, está realizada.

Si es cierto que el desarrollo del mismo está demasiado apresurado debido a las prácticas y a una planificación de la misma es paupérrima. Debido a ello, y a que aun soy un desarrollador novato, no he podido sacar el máximo potencial de las herramientas que he utilizado.

6.2 Posibles ampliaciones:

-Debido la pésima organización del proyecto, hay una serie de mejoras / ampliaciones que me habría gustado añadir, pero no he podido por falta de tiempo / recursos.

Una de ellas, era el panel de admin, tal y como se puede ver en la parte visual del proyecto, no se pueden crear nuevos libros, nuevos géneros o autores, ni siquiera modificarlos. De hecho, para que yo pudiera hacerlo, tenía que hacerlo mediante mis peticiones y reconozco que hubiera sido mejor si lo hubiera podido meter con formularios. El panel de admin tendría esas opciones además de una manera de modificar el rol del usuario permitiendo así crear más administradores (o quitarlos).

Otra implementación sería la de poder recuperar cuenta. Desde el punto de vista de la aplicación, si no te sabes el nombre y la contraseña, no podrías acceder a la cuenta o intentar recuperarla, para eso, había pensado que, al añadir un nuevo usuario, tuvieras que emplear el correo que proporcionaste al crear el usuario.

Otra implementación que me hubiera gustado poner es el tema de comentarios de los libros. Los usuarios, si se han terminado el libro, podrían poner un comentario de lo que les ha parecido el libro. Inclusive, añadir la nota que le pone. De esta manera podría organizar sus libros en función si tienen comentario o por la nota del mismo e incluso, se podría ordenar la lista de libros de la biblioteca según estas características.

7. Bibliografía:

7.1 Enlaces empleados para conceptos:

modelo (MVC)

IDE

7.2 Enlaces empleados para entornos de desarrollo:

Intellij idea

Visual Studio Code (VSC)

7.3 Enlaces empleados para lenguajes de desarrollos y frameworks:

Java

Spring Boot

Angular

TypeScript

JavaScript

NodeJS

MySQL

8. Anexos:

enlace parte front TFG

enlace parte back TFG