Seducción
ENTRE LÍNEAS

BOOK REVIEW

Autor: Santiago González Londoño
Curso: 2 de DAM
Año: 2021/2022
Ciclo formativo: Desarrollo de Aplicaciones Multiplataforma
Nombre del centro: IES La Mar
Tutor: Juan José Pedraza

BOOK REVIEW
Santiago González Londoño

Índice

1. INTRODUCCIÓN

1.1. Abstract

Este proyecto es una página web, el objetivo es poder compartir las reseñas en un formato accesible para cualquier dispositivo, sin la necesidad de tener que instalar ningún externo. La página web está diseñada para funcionar en cualquier tipo de pantalla, adaptandose al tamaño necesario para poder mostrar de manera correcto su contenido.

La página web hace uso de spring para crear un servidor rest donde se guardan todos los datos, para el cliente se utiliza angular, usando la librería angular material para la mayoría de elementos visuales de la página web, también se ha hecho uso de Swiper para la creación del carrusel de la página principal. Swagger fue utilizado para hacer tests y probar funcionalidades del servidor. También he usado MapStruct para el mapeo de entidades.

This project is a web page, the objective is to be able to share the reviews in an accessible format for any device, without the need to install any external. The website is designed to work on any type of screen, adapting to the necessary size to be able to display its content correctly.

The web page makes use of spring to create a rest server where all the data is saved, angular is used for the client, using the material angular library for most of the visual elements of the web page, Swiper has also been used for the creation of the carousel of the main page. Swagger was used to test and use functionality of the server. I also used MapStruct to map entities.

1.2. Módulos implicados

Programación:

Este módulo ha sido la base necesaría para el desarrollo del servidor en Java.

Base de datos – Acceso a datos:

Estos dos módulos han sido importantes para el desarrollo del servidor, los conomientos de Base de datos me sirvieron para el diseño de la base de datos SQL y el proyecto de Acceso a datos me sirvio para aprender spring, el framework usado para diseñar el servidor.

1.3. Tipo de proyecto

El tipo de proyecto experimental y de aprendizaje, ya que me tenído que aprender nuevas tecnologías para poder desarrollarlo (Angular) y también he tenído que trabajar en un frontend donde tengo poca experiencia y conomiento, diseñar la página y hacerla funcionar ha sido todo un reto.

2. MEMORIA EXPLICATIVA

2.1. Descripción técnica

Para el proyecto principalmente he tenído que aprender **Angular** para el frontend, también he de decir que el backend ha sido bastante influenciado por mis prácticas, ya que he trabajado con un servidor spring utilizado en una aplicación comercial y me ha permitido aprender nuevas técnicas útiles para el desarrollo.

2.1.1. Requisitos mínimos

Software:

- **Software**:
  - La página web soporta la gran mayoría de navegadores, pero deben tener las últimas versiones para asegurar completa funcionalidad.
- **Hardware**:
  - Un dispositivo capaz de ejecutar un navegador web actual

2.1.2. Herramientas y librerías

Software:

- **Servidor**:
  - **JDK 15** para el funcionamiento del servidor Java.
  - **Maven** para el manejo de las dependencias.
  - **Servidor SQL (ej: xampp)** para guardar la información
- **Cliente**:
  - **Node.js** para el manejo de las dependencias.

2.1.3. Compatibilidad

Navegador	Supported versions
Chrome	Última versión
Firefox	Última version y ESR
Edge	Últimas 2 versiones
Safari	Últimas 2 versiones
iOS	Últimas 2 versiones
Android	Últimas 2 versiones

2.2. Funcionamiento y desarrollo del sistema

La aplicación está dividad principalmente en dos partes:

- **Servidor**:

  El servidor se encarga de manejar la información y ofrecer diferentes endpoints que permiten al cliente acceder a esta información y modificarla.

- **Cliente**:

  El cliente se encarga de realizar peticiones a los endpoints del servidor para recibir la información necesaría para su funcionamiento, también utiliza los endpoints para realizar cambios en la información

2.2.1. Wireframe

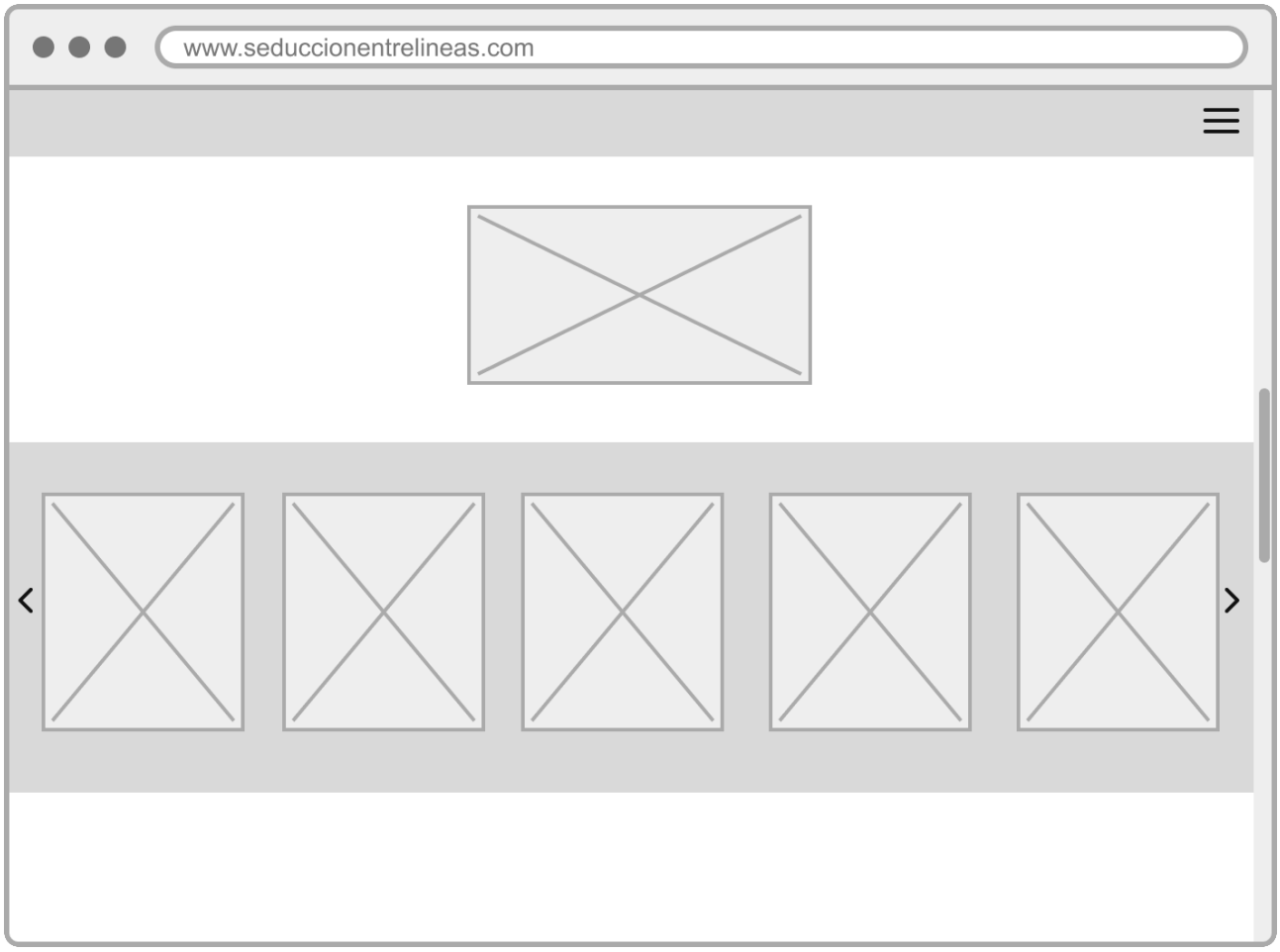


Figura 1: Wireframe de la página web en ordenador

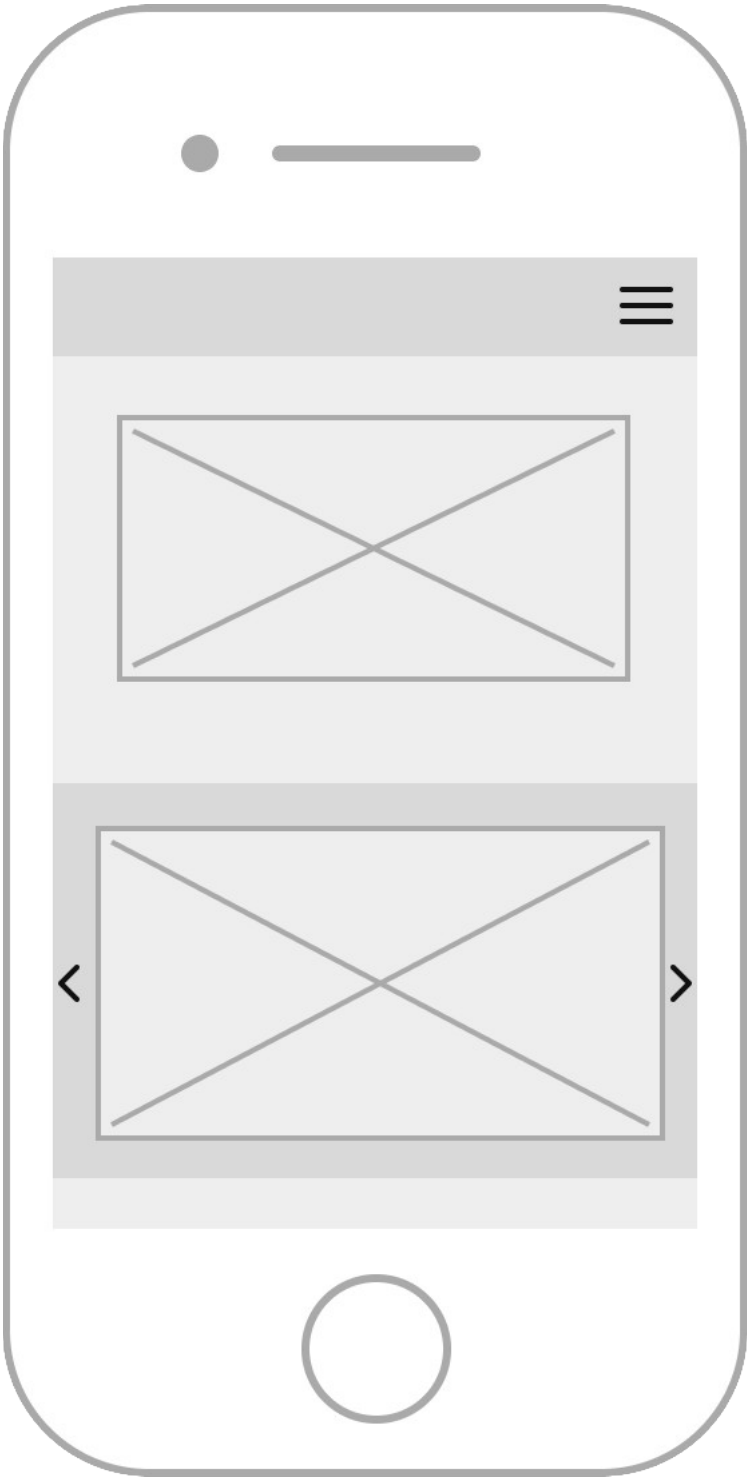


Figura 2: Wireframe del diseño en móvil

2.2.2. Spring

Con spring he creado un flujo de información entre el servidor y el cliente, este hace uso de DTOs para enviar la información entre ellos, en vez de crear muchos **end points** he creado un sistema dinamico de filtrado, esto permite al cliente a través de una única petición solicitar todo tipo de información al clienta, este filtrado también tiene opción de paginación, lo que he permitido crear los menús con todos los datos de una menera fácil.

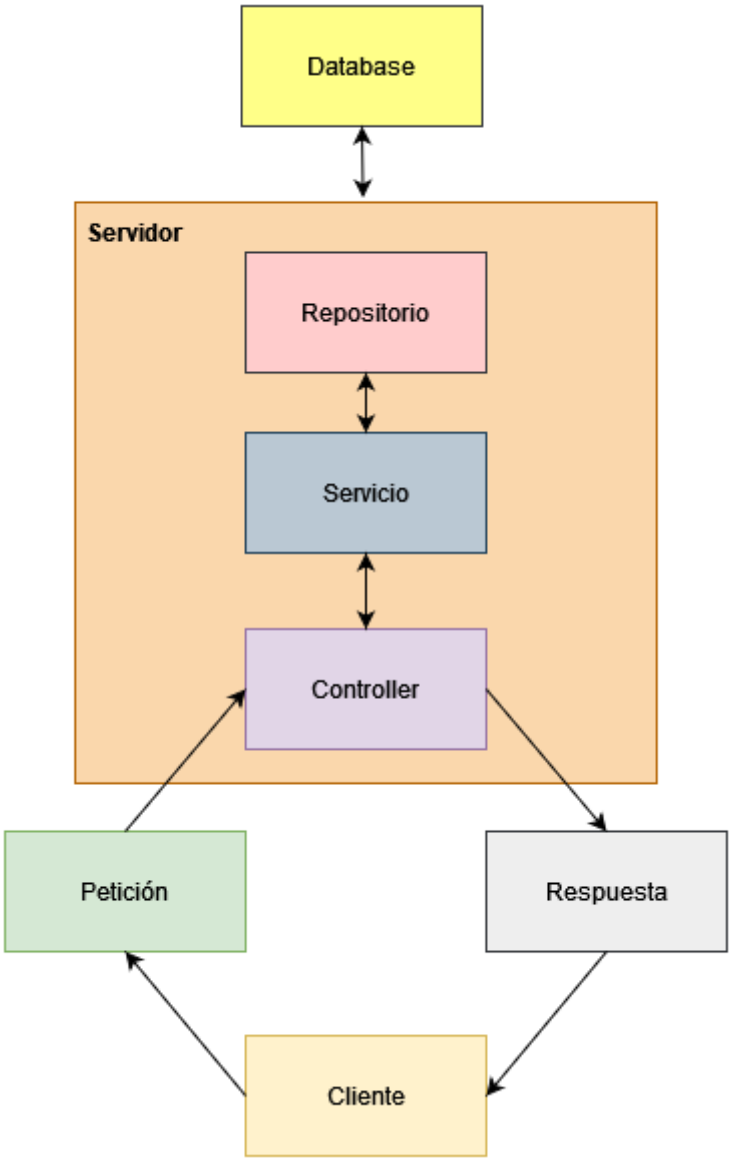


Figura 3: Flujo de información entre cliente y servidor

2.2.2.1 Mapper

El diseño de este flujo ha sido muy influenciado por el conocimiento que he conseguido durante mis prácticas. Inicialmente el servidor recibia una petición del cliente, buscaba la información (**entidad**) y devolvía esta directamente. Pero esto es problemático por varias razones. El cliente de Angular desconoce que es un **Long**, no es un tipo de dato, solo soporta **number** y este no dispone de toda la capacidad para almacenar todos los datos de un **Long**. Esto causa que el cliente al recibir la información no pueda encontrar un tipo de dato para guardarla y falle.

Esta no es la única razón por la que el diseño ha cambiado, al enviar devolver una **entidad** al cliente directamente, estamos enviado datos que pueden no ser deseados, tanto por seguridad como por eficiencia. Por ejemplo, si nuestro cliente solicita la información de un usuario y devolvemos la entidad completa, este tendría acceso al **hash** de la contraseña. La solución a este problema es más fácil de lo que parece.

Ejemplo del funcionamiento:

```java
@Entity
@Getter
@Setter
@ToString
@NoArgsConstructor
@Table(name = "categories")
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @EqualsAndHashCode.Include
    private Long id;

    @NotBlank
    @Size(min = 3, max = 64)
    @Column(name = "name")
    private String name;

    public Category(String name) { this.name = name; }
}
```

Figura 4: Entidad Categoria

Usando la entidad anterior, ahora crearemos una clase **DTO**, en mi caso la nomenclatura que he utilizado para nombrar estas clases es el nombre de la clase + **Response**.

```java
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class CategoryResponse {

    private String id;
    private String name;

}
```

Figura 5: DTO Categoria

Existe una librería fácil y pontente de utilizar llamada **MapStruct**, esta nos permite coger la información de una clase y pasarla a otra clase. Este proceso es conocido como **mapear** la información, lo único que nosotros debemos hacer es crear una clase con la información que queremos enviar al cliente, este tipo de clase suele ser llamado **DTO**. Con nuestro DTO creado, crearemos una interfaz anotada con **@Mapper**, en esta crearemos un metodo que sera utilizado para mapear la información. Al compilar la aplicación MapStruct crea la implementación de manera automatica.

```java
@Mapper(
        componentModel = "spring",
        injectionStrategy = InjectionStrategy.CONSTRUCTOR,
        nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.IGNORE)
public interface CategoryMapper {

    Category toEntity(CategoryRequest request);

    CategoryFilter toFilter(CategoryRequest request);

    CategoryResponse toResponse(Category entity);

}
```

Figura 6: Mapper Categoria

Como podemos ver lo único que debemos hacer es crear un metodo que reciba como parametro la clase original que queremos usar como origen de datos, el valor que devuelve el metodo es clase a la que se le insertara los datos. La implementación de esta clase se genera de manera automatica al compilar nuestra aplicación.

Si la entidad que deseamos mapear a su vez contiene otra entidad, podemos especificar en la anotación **@Mapper** otro mapper para mapear esta nueva entidad.

También nos podemos fijar que en el **Response** las **ids** son te tipo **String**, esto permite que el cliente pueda recibirla sin ningún tipo de problema, no debemos olvidar que el id tan solo se utilza como identificador, no se realizan operaciones con este, por tanto no hay problema en devolverlo como String en vez de Long.

2.2.2.2 Filtrado

Cada controlador cuenta con solo 3 endpoints para realizar todas las operaciones necesarías, esto es posible gracias a un sistema potente para el filtrado de información, de esta forma el cliente puede solicitar exactamente el dato que quiere a través de un único endpoint, este concepto también lo aprendí haciendo las prácticas, usualmente yo creaba un endpoint para cada tipo de búsqueda, por nombre, apellido, etc... Con un sistema dinamico de filtrado podemos reducir esto a un solo endpoint, también nos permite realizar búsquedas más complejas, así como fácilmente limitar el tipo de búsquedas.

Para el sistema de filtrado también he utilizado **MapStruct**, el cliente envía una clase **Request**, la cual se convierte en un **Filter** a través de este filtrado.

```
@Getter
@Setter
@ToString
@NoArgsConstructor
@AllArgsConstructor
public class CategoryRequest {

    @ValidId
    private String id;

    @Size(min = 1, max = 64)
    private String name;

    private FilterCriteria nameCriteria = FilterCriteria.EQUAL;

}
```

Figura 7: Request de Categoria

Esta request contiene la validación necesaría, esto nos permite asegurarnos que el cliente no nos envía información que pueda causar un error o sea incorrecta. Esta clase se convierte en un Filter usando un mapper.

```java
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class CategoryFilter implements RepositoryFilter<Category> {

    private Long id;
    private String name;

    @Builder.Default
    private FilterCriteria nameCriteria = FilterCriteria.EQUAL;

    @Override
    public Specification<Category> getSpecifications() {
        return new SpecificationBuilder<Category>()
                .add(getIdSpecification())
                .add(getNameSpecification())
                .build();
    }

    private Specification<Category> getIdSpecification() {
        if (this.id == null) {
            return null;
        }

        return ((root, query, criteriaBuilder) -> criteriaBuilder.equal(root.get(Category_.id), this.id));
    }

    private Specification<Category> getNameSpecification() {
        if (this.name == null || this.name.isEmpty()) {
            return null;
        }

        return ((root, query, criteriaBuilder) -> {
            if (FilterCriteria.EQUAL.equals(this.nameCriteria)) {
                return criteriaBuilder.equal(root.get(Category_.name), this.name);
            } else if (FilterCriteria.LIKE.equals(this.nameCriteria)) {
                return criteriaBuilder.like(root.get(Category_.name), like(this.name));
            } else {
                throw new IllegalArgumentException("Criteria for the attribute 'name' is not valid: " + this.nameCriteria);
            }
        });
    }
}
```

Figura 8: Filter Categoria

Este filtrado devuelve un **Specification<T>**, este es usado para que **JPA** haga su filtrado interno con los valores que le hemos dado. Como nos podemos fijar el propio filtre tiene diferentes **FilterCriteria**, estos son usados para permitir al cliente que tipo de búsqueda desea realizar para cada campo que lo pueda permitir, de esta forma pude deciri si quiere que la búsqueda por el nombre de la categoría sea de tipo **equal** o **like**.

El sistema de filtrado permite anidar estos filtrados, con entidades más complejas como es **Book**, uno puede desear buscar un **Book** en base a su autor, el sistema permite usar un **AutorFilter** dentro del **BookFilter** para buscar todos los libros que pertenezcan al autor en base a su filtro.

El sistema de filtrado también cuenta con un sistema para el paginado, las peticiones de filtrado solo permiten busquedas acompañadas por un páginado, esto está pensado para limitir la cantidad de información que el cliente puede solicitar así como por eficiencia.

El cliente no puede mostrar toda la información que contiene el servidor, por esto tampoco afecta o limita las funcionalidades del cliente, de hecho esto permite fácilmente implementar un menú páginado para mostrar información.

```java
@Getter
@ToString
@AllArgsConstructor
public class FilterRequest<T> {

    @NotNull
    private T filter;

    @NotNull
    private PageableRequest page;

}
```

Figura 9: Page request

Esto permite fácilmente añadir paginado a cualquier petición de filtrado y paginado.

2.2.2.3 Imágenes

Las imágenes son una parte esencial de la aplicación, el cliente muestra muchas imagenes de los libros, reseñas, etc... Por tanto era imperativo crear un sistema propio de almacenaje de imagenes.

Para la creación de este sistema he utilizado dos sistemas de almacenamiento, una **Entidad** para guardar información realacionada a la imagen, como la id, nombre o formato de la imagen. Este sistema viene acompañado de un sistema de archivos donde se guarda la imagene comprimida.

Cuando el cliente quiere una imagen se carga su información de un repositorio, si esta información existe se busca la imagen en un sistema de archivos.

```java
@Entity
@Getter
@Setter
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "img")
@ToString
@EqualsAndHashCode
public class Image {

    @Id
    @Column(name = "id")
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

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

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

}
```

Figura 10: Imagen

Esta entidad contiene la información necesaria de la imagen. Cuando el endpoint de subida de imágenes recibe una nueva imagen, crea una entidad con la información de la imagen, luego comprime la imagen usando un **Inflater** y guarda los bytes de la imagen en un archivo sin formato.

2.2.2.4 Autentificación

Para una comunicación segura entre el servidor y el cliente, las peticiones del cliente a ciertos endpoints deben ir acompañadas de un **token** que verifica que el usuario está registrado en la página web y tiene los permisos adecuados. Para la creación de este sistema he hecho uso de **JWT** (JSON Web Tokens).

Para la creación de este sistema he utilizado una llave pública y una llave privada, estas se van a utilizar para crear tokens para los usuarios cuando inician sesión.

```
@Value("classpath:public.pub")
RSAPublicKey publicKey;

@Value("classpath:private.key")
RSAPrivateKey privateKey;

@Bean
JwtDecoder jwtDecoder() { return NimbusJwtDecoder.withPublicKey(this.publicKey).build(); }

@Bean
JwtEncoder jwtEncoder() {
    JWK jwk = new RSAKey.Builder(this.publicKey).privateKey(this.privateKey).build();
    JWKSource<SecurityContext> jwks = new ImmutableJWKSet<>(new JWKSet(jwk));
    return new NimbusJwtEncoder(jwks);
}
```

Figura 11: Llave privada y publica

Este sistema se utiliza en conjunto con el sistema de autentificación de spring, básicamente primero se debe autentificar el usuario, luego se genera un token con esa autentificación que sera usado por cliente para demostrar que ha iniciado sesión.

```
public String generateToken(Authentication authentication) throws JwtException {
    String scope = authentication.getAuthorities().stream() Stream<capture of ? extends GrantedAut
            .map(GrantedAuthority::getAuthority) Stream<String>
            .collect(Collectors.joining( delimiter: " "));

    Instant now = Instant.now();
    JwtClaimsSet claims = JwtClaimsSet.builder()
            .issuer("self")
            .issuedAt(now)
            .expiresAt(now.plusSeconds(JWT_TOKEN_VALIDITY))
            .subject(authentication.getName())
            .claim( name: "scope", scope)
            .build();

    return this.jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
}
```

Figura 12: Creación del token con las llaves

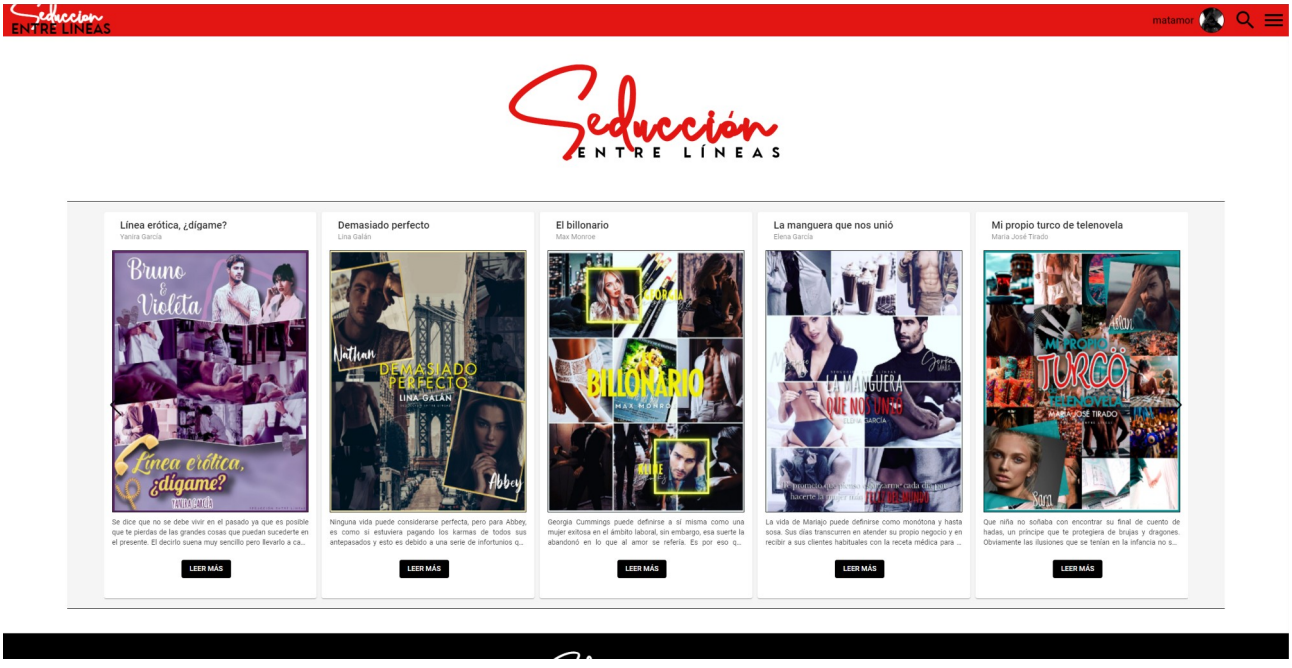
2.2.3 Angular

Para el cliente decidí utilizar **Angular**, de nuevo esta decisión fue influenciada por mis prácticas donde en frontend está creado con esta tecnología. Es una plataforma potente para la creación de aplicaciones web, a su vez esta funciona utilizando **TypeScript**, lo cual es una capa superior a **javascript**, que añade tipado, interfaces, etc. Haciendo javascript una lengua más **OOP**, aunque las clases no son tan potentes como las clases reales que tiene un lenguajes OOP real. No debemos olvidar que las **clases** de javascript no son más que una capa encima del prototype que javascript utiliza.

El desarrollo del cliente ha sido complejo debido a varios factores, uno siendo el conocimiento, ya que tenía cero experiencia con angular, gracias a las practicas he tenido ilimitado a **udemy** lo cual me ha permitido conseguir mucho conocimiento. En general mi conocimiento del frontend es limitado, crear los diseños con **css**/**html** ha sido todo un reto, pero angular tiene librerías que facilitan mucho el desarrollo.

Las librería que mas he usado ha sido **Angular Material**, librería esencial para angular.

2.2.3.1 Página principal



2.2.3.1.1 Swiper

Uno de los mayores resto vino con la porpia página principal, en esta quería mostrar un carrusel con las últimas reseñas, este automaticamente va pasando las diferentes reseñas, creando una bucle infinito. Debido a la complejidad del mismo sistema he utilizado una librería, esto ya en si ha sido un reto, debido a que no hay muchas librerias que funcionaran exactamente como yo quería y muchas de las que hay son de pago.

Finalmente me decidí por una librería que se llama **Swiper**, esta tiene un sistema muy editable que también es reactivo, pudiendo adaptarse a todo tipo de pantallas.

```
<ng-template [ngIf]="reviews">
  <swiper [config]="sliderConfig">
    <ng-template *ngFor="let review of reviews" swiperSlide>
      <app-review-entry [review]="review"></app-review-entry>
    </ng-template>
  </swiper>
</ng-template>
```

Figura 13: Slider HTML

La creación del slider son pocas líneas, toma más espacio su propia configuración. Con el uso de **ng-template** podemos hacer que no se renderize el resto del html hasta que la información no haya sido cargada, también puede ser conveniente un sistema que muestre información estilo **placeholder** mientras se carga la información real.

Como podemos ver se hace un for por las diferentes reseñas y se crear un **entry** por cada una, la información se envia a través del **@Input** que trae angular para enviar información entre componentes.

```
<ng-template [ngIf]="review">
  <mat-card (click)="onClick()" class="my-1 unselectable background-secondary">
    <div fxFlexFill fxLayout="column" fxLayoutAlign="center start">
      <mat-card-header>
        <mat-card-title class="title-clamp">{{ review.book.title }}</mat-card-title>
        <mat-card-subtitle>{{ review.book.autor.name + ' ' + review.book.autor.surnames }}</mat-card-subtitle>
      </mat-card-header>
      <img [src]="review.image.id | image" alt="review" class="card-image" mat-card-image>
      <mat-card-content>
        <p class="line-clamp">{{ review.review }}</p>
      </mat-card-content>
      <mat-card-actions fxFlexFill fxLayout="column" fxLayoutAlign="center center">
        <button mat-button>LEER MÁS</button>
      </mat-card-actions>
    </div>
  </mat-card>
</ng-template>
```

Figura 14: Slider entry HTML

El entry contiene la información que quiero mostrar de una reseña, he utilizad un mat-card para crear el diseño base.

2.2.3.1.2 Navbar

El navbar es fijo y se mantiene siempre en la parte superior de la pantalla, este es reactivo y se adapta al tamaño de la pantalla. Quería conseguir un estilo minimalista, sin tener tantas opciones en la barra, por ello he usado iconos y menús flotantes para crearlo.

```
<header class="sticky-top unselectable">
  <mat-toolbar class="background-primary" fxLayout="row" fxLayoutAlign="space-between space-between">
    <!-- BIG SCREEN -->
    <div fxFlexFill fxHide fxLayout="row" fxLayoutAlign="start space-between" fxShow.gt-sm>
      <div fxFlex="50" fxFlexFill fxLayout="column" fxLayoutAlign="center start">
        <app-title routerLink="/"></app-title>
      </div>
      <div fxFlex="50" fxLayout="row" fxLayoutAlign="end center">
        <ul class="navbar-list" fxFlexFill fxLayout="row" fxLayoutAlign="end center" fxLayoutGap="30px">
          <li class="h-100" fxLayout="row" fxLayoutAlign="center center">
            <app-profile-navbar class="h-100"></app-profile-navbar>
          </li>
          <li fxLayout="row" fxLayoutAlign="center center">
            <mat-icon (click)="openSearch()" class="navbar-icon">search</mat-icon>
          </li>
          <li [matMenuTriggerFor]="linksMenu" class="navbar-item" fxLayout="row" fxLayoutAlign="center center">
            <mat-icon class="navbar-icon">menu</mat-icon>
          </li>
        </ul>
      </div>
    </div>
    <!-- SMALL SCREEN -->
    <div fxFlexFill fxHide fxLayout="row" fxLayoutAlign="start space-between" fxShow.lt-md>
      <ul class="navbar-list" fxFlexFill fxLayout="row" fxLayoutAlign="space-between center" fxLayoutGap="50px">
        <li class="navbar-item" fxLayout="row" fxLayoutAlign="center center">
          <mat-icon (click)="openSearch()" class="navbar-icon">search</mat-icon>
        </li>
        <li class="navbar-item" fxLayout="row" fxLayoutAlign="center center">
          <mat-icon (click)="toggleSidenav()" class="navbar-icon">menu</mat-icon>
        </li>
      </ul>
    </div>
  </mat-toolbar>
</header>
```

Figura 15: Navbar HTML

He hecho uso de **mat-toolbar**, **mat-menu** y **mat-ico** para crear el navbar, todo siendo parte de la librería **material**, he dividido el navbar en dos fotosporque no cabe todo en una foto.

```
<mat-menu #linksMenu="matMenu" class="border-dark border border-1 mt-1 menu-container" xPosition="before">
  <div class="p-3 menu-container" fxLayout="column" fxLayoutAlign="center center" fxLayoutGap="10px">
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/">
      <mat-icon class="menu-icon">home</mat-icon>
      Inicio
    </button>
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/books">
      <mat-icon class="menu-icon">book</mat-icon>
      Libros
    </button>
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/reviews">
      <mat-icon>local_library</mat-icon>
      Reviews
    </button>
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/autors">
      <mat-icon class="menu-icon">people_outline</mat-icon>
      Autores
    </button>
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/categories">
      <mat-icon class="menu-icon">local_offer</mat-icon>
      Categorías
    </button>
    <button class="menu-item" fxLayout="row" fxLayoutAlign="start center" mat-menu-item routerLink="/editorials">
      <mat-icon class="menu-icon">business</mat-icon>
      Editoriales
    </button>
  </div>
</mat-menu>
```

Figura 16: Navbar menu HTML

Esta parte es el menú flotante que aparece cuando se clica en el botón del navbar.

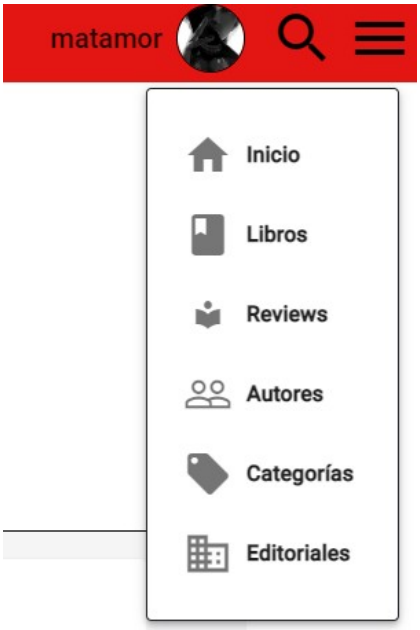
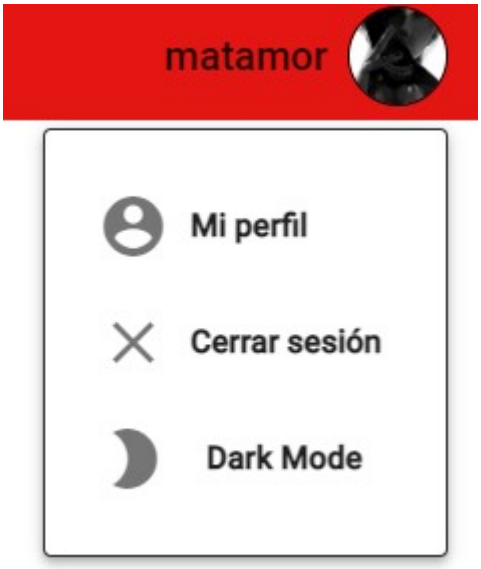


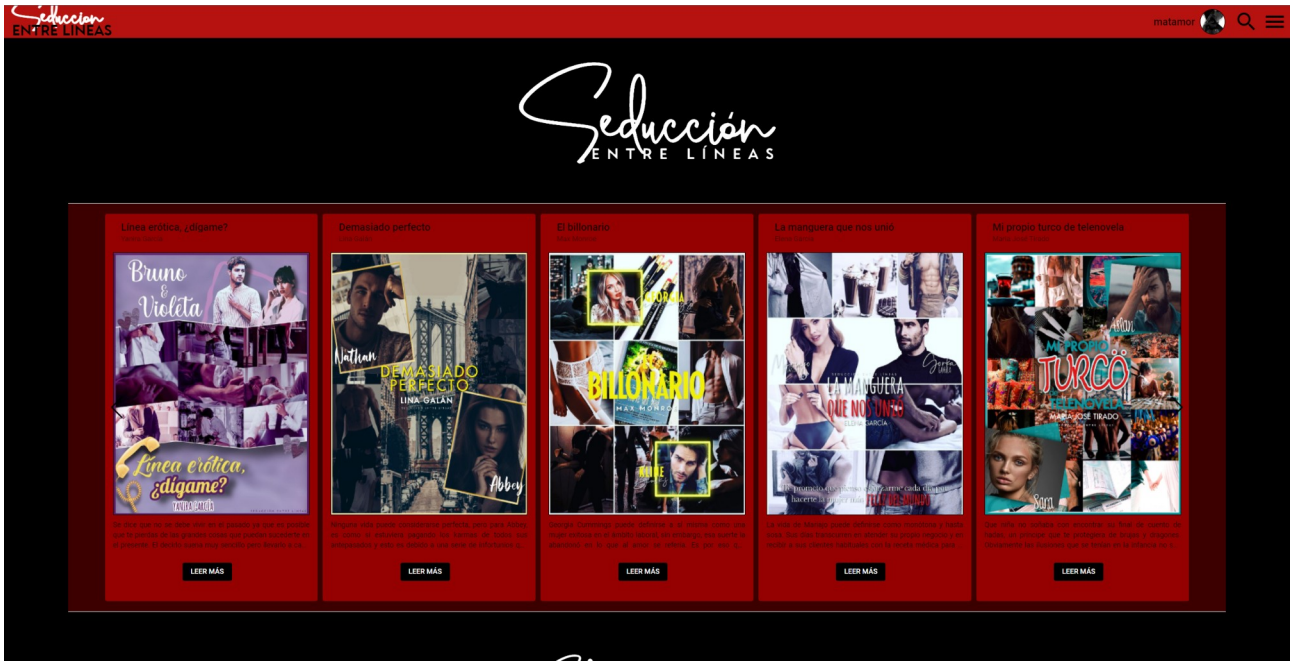
Figura 17: Menu flotante

2.2.3.1.3 Dark Mode

Un modo noche es esencial para una página con tanto brillo, por ello he creado un que se puede activar mediante el menú flotante de tu perfil.



Al activarlo la página se cambia al modo noche sin necesidad de recargarla, he decidido hacer el modo noche opcional, de tal manera que solo se activa si el usuario lo desea.



2.2.3.1.4 Búsqueda

Para facilitar las búsquedas de información en la página se me ocurrió crear un sistema que soporte todos los diferentes tipos de datos que tiene la aplicación. Para el diseño he utilizado **mat-dialog** lo que permite crear pestaña flotantes que pueden aparecer desde cualquier lado sin necesidad de cambiar de página. También he utilizado **mat-tab**, esto permite crear diferentes pestañas dentro del dialogo para así dividir las búsquedas según su tipo.

Figura 18: Menú de búsquedas

2.2.3.1.5 Inicio de sesión

Para iniciar sesión en la página web he creado un sistema parecido a las búsquedas, he utilizado un **mat-dialog** junto a **mat-tab** lo que ha permitido tener de manera organizada ambos menús en uno solo.

Figura 19: Menú de login/registro

2.2.3.1.6 Perfil de usuario

Quería que los usuarios pudieran modificar parte de su información personal (excluyendo su nombre de usuario y correo electrónico). Para que sea de fácil acceso desde cualquier lugar, he añadido su acceso directamente en el **navbar**, para el menú en si he utilizad **mat-dialog** así desde cualquier lugar se puede ver:

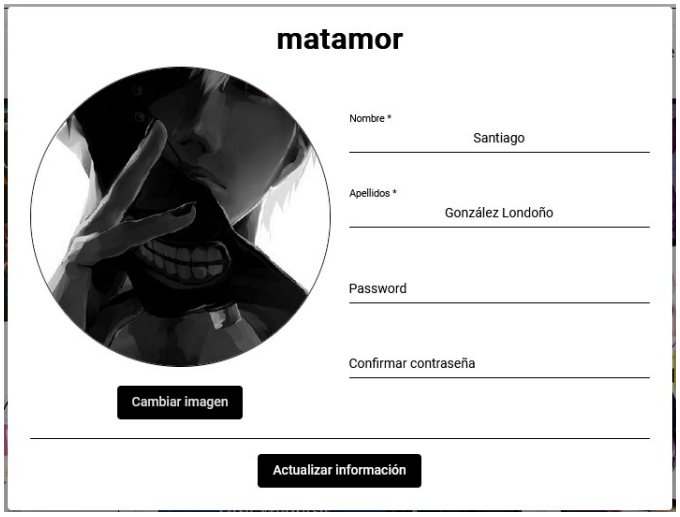


Figura 20: Perfil de usuario

2.2.3.2 Modo admin

Era necesario que se pudiera modificar la información existente así como poder añadir nuevos registros. Para ello he creado un módulo específicamente para el módulo de administración, de esta manera puedo hacer que toda la información pertinente a la administración no se le cargue a todos los usuarios, solo a aquellos que sean administradores y accedan al menú de administración.

Para evitar que usuarios sin permiso puedan acceder al módulo he creado lo que se llamada **Guard**, este elemento se puede asignar a una ruta y permite limitar su acceso.

```
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
  Observable<boolean|UrlTree>|Promise<boolean|UrlTree>|boolean|UrlTree {

  if (!this.authService.hasPrivilege( name: 'EDIT')) {
    this.router.navigateByUrl( url: '/');
    return false;
  }

  return true;
}
```

Figura 21: Admin guard

De esta manera solo aquellos con el permiso **EDIT** pueden acceder al módulo admin.

El menú de administración a parte de ser su propio módulo, tiene su propia interfaz, esta es mucho más simple y accesible, también se adapta al tamaño de tu pantalla, permitiendo el acceso a cualquier tipo de dispositivo.

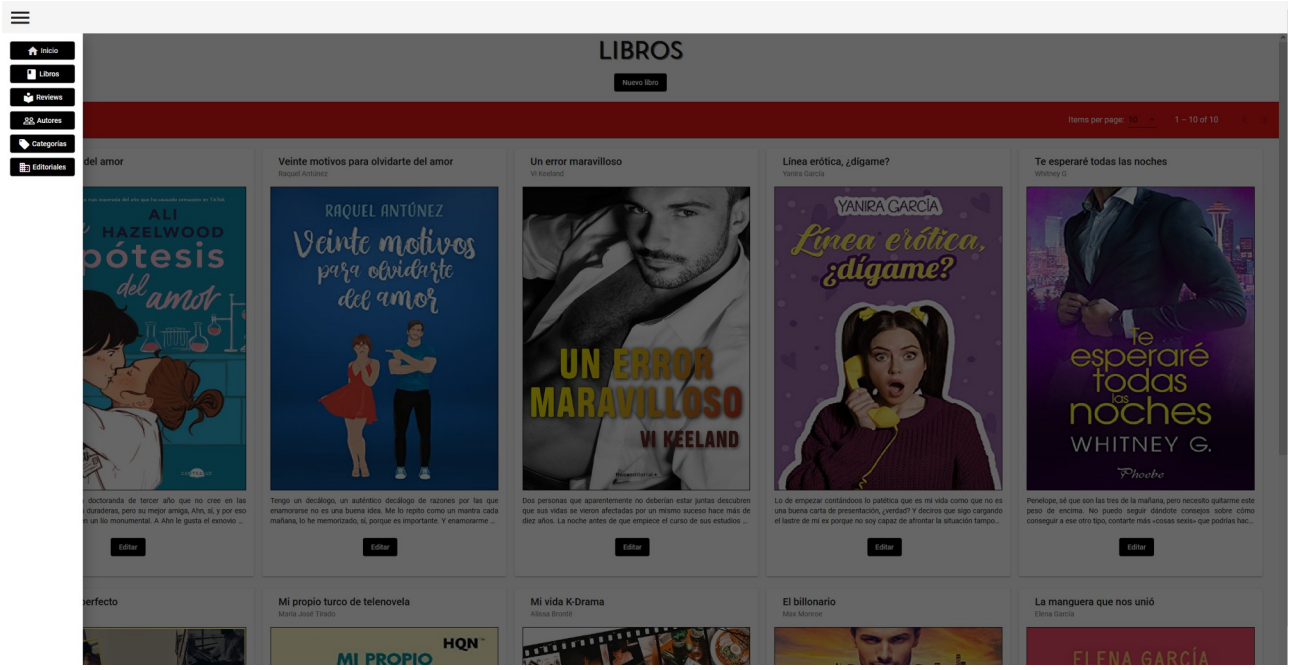


Figura 22: Modo administración

Desde aquí podemos hacer clic en el elemento que deseemos para poder modificar su información, también podemos crear fácilmente un nuevo registro de cualquier tipo.

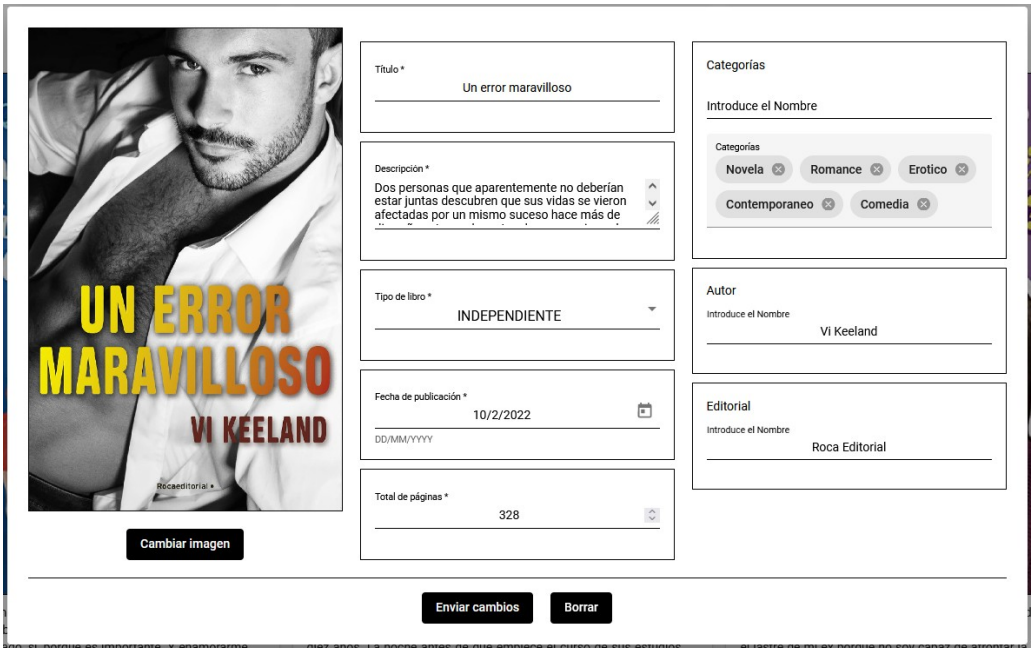


Figura 23: Menú de edición de un libro

3. AMPLICACIONES Y MEJORAS

Aplicación móvil: Una aplicación móvil para mayor accesibilidad, aunque debo decir que la página no necesita una aplicación para el correcto funcionamiento en móvil.

Correos electrónicos: Sistema de verificación por correo electrónico, así como notificaciones de nuevas publicaciones

Sistema de comentarios: Permitir realizar comentarios en las publicaciones a los usuarios.

4. PROBLEMAS ENCONTRADOS

Seguridad: Tuve problemas para crear un sistema de autentificación por token, acabé creando un sistema con JWT.

Peticiones/Respuestas: No podía recibir peticiones o enviar respuestas de cierta información con el mismo tipo de dato que se guarda, tuve que crear un sistema de mapeo.

Diseño: Me costo crear el diseño de la aplicación web por mi poco conocimiento de las diferentes tecnologías.

5. PRESUPUESTO GENERAL DEL PROYECTO

Tecnologías utilizadas			0 €
Mano de obra	Horas	Precio hora	Total
Análisis	20	15 €	300 €
Diseño de clases	15	15 €	250 €
Programación	50	15 €	750 €
Pruebas	15	15 €	250 €
Total	70		1550 €

6. ANEXOS Y DOCUMENTOS COMPLEMENTARIOS

6.1. Prevención de riesgos

Los principales riesgos laborales que experimentan los trabajadores informáticos son:

- **Fatiga visual o muscular**: Es importante mantener a una distancia adecuada la pantalla, un mínimo 40 centímetros, la parte inferior de la pantalla este ligeramente inclinada de modo que el enfoque sea perpendicular a nuestro ángulo de visión. También es importante que el espacio cuente con la luz adecuada: el sistema de iluminación artificial será ambiental para evitar puntos de sombra o un exceso de luz.

- **Fatiga musculas**: Es producida por mantener posturas incorrectas al sentarse y la ubicación incorrecta del equipo. Es también importante cambiar de postura, mantener una postura de manera prolongada conduce a la fatiga.

- **Carga mental**: El estrés en el trabajo o la falta de motivación pueden ser un riesgo laboral y llegar a afectar a la salud de los trabajadores. Por ello, es importante realizar tareas variadas, realizar paradas periódicas para prevenir la fatiga, seguir hábitos de vida saludable y realizar ejercicio de forma habitual.

6.2. Contenido del USB

El USB contiene la siguiente información:

- Una copia digital de la documentación.
- Una copia digital de la presentación.
- Manual de instalación del proyecto.
- El proyecto dividido en dos partes, el **backend** y el **frontend**.

6.3. índice de imágenes

Índice de figuras

7. Conclusión

El desarrollo de este proyecto me ha servido para aumentar y perfeccionar mis conocimientos, a lo largo del desarrollo me he enfrentado a diferentes problemas que ha mejorado mi capacidad solución de problemas así como mi paciencia.

El resultado final ha cumplido con las expectativas del proyecto así como con mis expectativas, crear una aplicación que facilite el acceso a las reseñas que se actualmente se realizaban en una aplicación de terceros y ahora cuenta con su propia plataforma.

El principal reto del proyecto ha sido el tiempo, compaginar el proyecto, las prácticas y el trabajo ha sido todo un reto.

8. BIBLIOGRAFÍA

- https://spring.io/projects/spring-boot
- https://angular.io/
- https://material.angular.io/
- https://stackoverflow.com/