GRADO EN INGENIERÍA DE COMPUTADORES



**Trabajo Fin de Grado**

Desarrollo de una aplicación web para la visualización e interacción de contenidos multimedia

**Autor:**

David Bachiller Vela

**Tutor:**

Salvador Otón Tortosa

Escuela Politécnica Superior

Universidad de Alcalá

2024

**UNIVERSIDAD DE ALCALÁ**



**Escuela Politécnica Superior**

**Grado en Ingeniería de Computadores**

Trabajo Fin de Grado

Desarrollo de una aplicación web para la visualización e interacción de contenidos multimedia

**David Bachiller Vela**

**mayo / 2024**

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería de computadores

Trabajo Fin de Grado

DESARROLLO DE UNA APLICACIÓN WEB PARA LA VISUALIZACIÓN E INTERACCIÓN DE CONTENIDOS MULTIMEDIA

**Autor: David Bachiller Vela**

**Tutor: Salvador Otón Tortosa**

Tribunal:

Presidente: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Vocal 1º: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Vocal 2º: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

FECHA: mayo / 2024

# Introducción

La aplicación web desarrollada en este proyecto se basa en el marco de trabajo Spring Boot y se centra en proporcionar una plataforma interactiva para los usuarios. La aplicación permite a los usuarios registrarse y asumir roles específicos, lo que a su vez determina sus niveles de acceso y funcionalidades disponibles. Los usuarios pueden interactuar con una variedad de contenidos multimedia y también tienen la capacidad de comentar sobre estos contenidos, proporcionando así un espacio para la discusión y el intercambio de opiniones. Además, la aplicación incluye la funcionalidad de comunidades y chats, permitiendo a los usuarios formar grupos y tener conversaciones en tiempo real. Las dependencias necesarias para el proyecto se manejan a través de Maven. En resumen, este proyecto proporciona una plataforma completa para la interacción del usuario y la compartición de contenido multimedia.

## Propósito del documento

El propósito de este documento es proporcionar una descripción detallada y completa del proyecto de desarrollo de una aplicación web. Este proyecto, desarrollado con Spring Boot y Java, ofrece una plataforma interactiva para los usuarios, permitiéndoles visualizar y comentar contenido multimedia, así como interactuar con otros usuarios a través de comunidades de chat.

Este documento está diseñado para ser una guía completa tanto para la comunidad académica como para el público en general. Su objetivo es explicar de manera clara y concisa el funcionamiento de la aplicación, desde su arquitectura y funcionalidades hasta los detalles de su implementación.

Además, este documento también tiene como objetivo ser una herramienta de aprendizaje para aquellos interesados en el desarrollo de aplicaciones web con Spring Boot y Java. Al proporcionar una visión detallada de un proyecto real, espero que este documento pueda ayudar a los lectores a entender mejor los conceptos y prácticas de la ingeniería de software

## 1.2 Alcance del proyecto

Este proyecto se centra en el desarrollo de una aplicación web interactiva que permite a los usuarios visualizar y comentar contenido multimedia, así como interactuar con otros usuarios a través de comunidades de chat. La aplicación está construida con Spring Boot y Java en el backend, y utiliza una base de datos MySQL para almacenar la información del usuario, los detalles de las comunidades de chat, los mensajes de chat y los comentarios sobre el contenido multimedia. En el frontend, se utiliza una combinación de HTML, CSS, JavaScript y Thymeleaf para crear una interfaz de usuario dinámica y atractiva. Se utiliza AJAX para realizar solicitudes asíncronas al servidor y actualizar partes de la página web sin necesidad de recargarla por completo. Esto mejora la experiencia del usuario al hacer que la aplicación sea más rápida y receptiva. Las dependencias del proyecto se manejan a través de Maven en el backend y npm en el frontend. Esto incluye bibliotecas y marcos como Spring Boot para el desarrollo del servidor, Thymeleaf para la plantilla del lado del servidor, y Bootstrap para el diseño y los componentes de la interfaz de usuario. Además, se utilizan diversas herramientas y prácticas de desarrollo de software para garantizar la calidad del código y facilitar la colaboración entre los desarrolladores. Esto incluye el uso de un sistema de control de versiones (Git), pruebas unitarias y de integración, y la integración continua/despliegue continuo (CI/CD). En resumen, el alcance de este proyecto abarca tanto el desarrollo del frontend como del backend de una aplicación web completa, incluyendo la gestión de usuarios, la visualización y el comentario de contenido multimedia, y la interacción en tiempo real a través de comunidades de chat.

## 1.3 Definiciones, acrónimos y abreviaturas

**Usuario:** Persona que se ha registrado en la aplicación y puede interactuar con el contenido y otros usuarios.

**Comunidad de chat:** Grupo de usuarios que pueden enviar y recibir mensajes en tiempo real.

**Contenido multimedia:** Cualquier forma de contenido, como videos, imágenes o texto, que los usuarios pueden ver y comentar en la aplicación.

**Spring Boot:** Marco de trabajo de Java que simplifica la configuración y el despliegue de aplicaciones Spring.

**Java:** Lenguaje de programación de alto nivel utilizado para desarrollar la aplicación.

**MySQL:** Sistema de gestión de bases de datos relacional utilizado para almacenar la información de la aplicación.

**Maven:** Herramienta de gestión de proyectos y comprensión de software. Se utiliza para manejar las dependencias del proyecto.

**HTML:** Lenguaje de marcado utilizado para estructurar el contenido en la web.

**CSS:** Lenguaje de hojas de estilo utilizado para describir la apariencia de los documentos HTML.

**JavaScript:** Lenguaje de programación interpretado que se utiliza para crear contenido dinámico en la web.

**Thymeleaf:** Motor de plantillas Java para procesar vistas HTML en aplicaciones web.

**AJAX:** Conjunto de técnicas de desarrollo web para crear aplicaciones asíncronas. Se utiliza para realizar solicitudes al servidor sin tener que recargar la página completa.

**Git:** Sistema de control de versiones distribuido utilizado para rastrear cambios en el código fuente durante el desarrollo de software.

**Pruebas unitarias:** Tipo de prueba que verifica la correcta funcionalidad de una unidad de código específica.

**Pruebas de integración:** Tipo de prueba que verifica la correcta interacción entre varias unidades de código.

**CI/CD:** Prácticas de integración y despliegue continuos. Se utilizan para automatizar el despliegue de la aplicación.

**IDE:** Entorno de desarrollo integrado. Es una aplicación de software que proporciona servicios integrales para facilitar el desarrollo de software.

**API:** Interfaz de programación de aplicaciones. Conjunto de reglas y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas.

**HTTP:** Protocolo de transferencia de hipertexto. Es el protocolo de comunicación que permite las transferencias de información en la web.

**URL:** Localizador uniforme de recursos. Es una referencia a un recurso web que especifica su ubicación en una red de computadoras y un mecanismo para recuperarlo.

**Nickname:** Apodo que usan los usuarios en la aplicación es como un identificador, el resto de usuarios te reconocerán por tu nickname.

## 1.4 Referencias

**Spring Boot:** Spring Boot es un marco de trabajo de Java que simplifica la configuración y el despliegue de aplicaciones Spring. La documentación oficial de Spring Boot ha sido una referencia valiosa durante el desarrollo del proyecto.

[https://spring.io/projects/spring-boot](https://spring.io/projects/spring-boot%20%20)

**Java:** Java es el lenguaje de programación utilizado para desarrollar la aplicación. La documentación oficial de Java proporciona una descripción detallada de las características del lenguaje y su API estándar.

[https://docs.oracle.com/en/java/](https://docs.oracle.com/en/java/%20%20)

**MySQL:** MySQL es el sistema de gestión de bases de datos utilizado para almacenar la información de la aplicación. La documentación oficial de MySQL proporciona información sobre cómo utilizar este sistema de gestión de bases de datos.

[https://dev.mysql.com/doc/](https://dev.mysql.com/doc/%20%20)

**Maven:** Maven es una herramienta de gestión de proyectos y comprensión de software. Se utiliza para manejar las dependencias del proyecto. La documentación oficial de Maven proporciona información sobre cómo utilizar esta herramienta.

[https://maven.apache.org/guides/](https://maven.apache.org/guides/%20%20)

**HTML, CSS, JavaScript:** Estos son los lenguajes utilizados para desarrollar el frontend de la aplicación. La documentación oficial de estos lenguajes proporciona una descripción detallada de sus características y cómo utilizarlos.

[https://developer.mozilla.org/](https://developer.mozilla.org/%20%20)

**Thymeleaf:** Thymeleaf es un motor de plantillas Java para procesar vistas HTML en aplicaciones web. La documentación oficial de Thymeleaf proporciona información sobre cómo utilizar este motor de plantillas.

[https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html](https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html%20%20)

**AJAX:** AJAX es un conjunto de técnicas de desarrollo web para crear aplicaciones asíncronas. Se utiliza para realizar solicitudes al servidor sin tener que recargar la página completa. [https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX](https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX%20%20)

**Git:** Git es un sistema de control de versiones distribuido utilizado para rastrear cambios en el código fuente durante el desarrollo de software.

<https://git-scm.com/doc>

## 1.5 Visión general del documento

Este documento consta de varias secciones que proporcionan una descripción detallada del proyecto de desarrollo de una aplicación web interactiva.

En la sección "**Descripción general del proyecto**", se proporciona una visión general de la aplicación y sus funcionalidades, incluyendo la gestión de usuarios, la visualización y el comentario de contenido multimedia, y la interacción en tiempo real a través de comunidades de chat.

La sección "**Requisitos específicos**" detalla los requisitos funcionales y no funcionales de la aplicación, incluyendo las interfaces de usuario, las funcionalidades esperadas y los requisitos de rendimiento.

La sección "**Arquitectura del sistema**" describe la arquitectura de la aplicación y los componentes del sistema, incluyendo las tecnologías y bibliotecas utilizadas, como Spring Boot, Java, MySQL, Maven, HTML, CSS, JavaScript y Thymeleaf.

La sección "**Diseño detallado**" proporciona detalles sobre el diseño de la aplicación, incluyendo diagramas de clases y de secuencia.

La sección "**Implementación**" describe el entorno de desarrollo, las dependencias y el proceso de instalación. También se incluye el código fuente de la aplicación.

La sección "**Pruebas**" describe la estrategia de pruebas y los casos de prueba utilizados para verificar la correcta funcionalidad de la aplicación.

La sección "**Uso de la aplicación**" proporciona una guía para los usuarios sobre cómo utilizar la aplicación, incluyendo cómo registrarse, cómo visualizar y comentar contenido multimedia, y cómo interactuar con otros usuarios a través de comunidades de chat. Así como una guía para administradores sobre cómo gestionar el contenido.

La sección "**Mantenimiento y soporte**" describe los procedimientos de mantenimiento y soporte al usuario.

Finalmente, la sección "**Conclusiones y trabajos futuros**" resume los resultados del proyecto y sugiere posibles mejoras para el futuro.

Este documento está diseñado para ser una guía completa tanto para la comunidad académica como para el público en general. Su objetivo es explicar de manera clara y concisa el funcionamiento de la aplicación, desde su arquitectura y funcionalidades hasta los detalles de su implementación.

# Descripción general del proyecto

El proyecto es una aplicación web interactiva desarrollada en Java con el marco de trabajo Spring Boot. La aplicación ofrece a los usuarios la posibilidad de registrarse y asumir roles específicos, permitiéndoles interactuar con una variedad de contenidos multimedia. Las funcionalidades principales de la aplicación incluyen la visualización y comentario de contenido multimedia, la interacción en tiempo real a través de comunidades de chat, la gestión de perfiles de usuario y un panel de administración para la gestión de contenidos y comunidades. Los usuarios pueden formar grupos y tener conversaciones en tiempo real, y según su plan de suscripción, pueden tener acceso a diferentes niveles de contenido y funcionalidades.

En el backend, la aplicación utiliza Spring Boot y Java, y una base de datos MySQL para almacenar la información del usuario, los detalles de las comunidades de chat, los mensajes de chat y los comentarios sobre el contenido multimedia. En el frontend, se utiliza una combinación de HTML, CSS, JavaScript y Thymeleaf para crear una interfaz de usuario dinámica y atractiva. AJAX se utiliza para realizar solicitudes asíncronas al servidor y actualizar partes de la página web sin necesidad de recargarla por completo.

Las dependencias del proyecto se manejan a través de Maven en el backend. Esto incluye bibliotecas y marcos como Spring Boot para el desarrollo del servidor, Thymeleaf para la plantilla del lado del servidor, y Bootstrap para el diseño y los componentes de la interfaz de usuario. Además, se utilizan diversas herramientas y prácticas de desarrollo de software para garantizar la calidad del código y facilitar la colaboración entre los desarrolladores. Esto incluye el uso de un sistema de control de versiones (Git), pruebas unitarias y de integración, y la integración continua/despliegue continuo (CI/CD).

## 2.1 Funciones del producto

Ofrece varias funciones clave para los usuarios. Estas incluyen:

- Registro de usuarios: Los usuarios pueden registrarse en la aplicación, lo que les permite acceder a todas las funcionalidades disponibles. Dentro del registro se incluye la elección del plan de suscripción y el pago de este, sin un registro previo no se podrá acceder al contenido y según sea el plan de suscripción de podrá o no ver determinados contenidos y realizar determinadas opciones.

- **Visualización de contenido multimedia**: Los usuarios pueden ver una variedad de contenido multimedia en la aplicación, desde películas, carreras de f1, partidos de fútbol y contenido en directo el cual solo se podrá ver con el plan de suscripción mas avanzado el cual se ira actualizando semanalmente igual que el resto del contenido de la aplicación.

- **Comentarios sobre contenido multimedia**: Los usuarios tienen la capacidad de comentar sobre el contenido multimedia, proporcionando un espacio para la discusión y el intercambio de opiniones.

- **Comunidades de chat**: Los usuarios pueden formar y unirse a comunidades de chat, lo que les permite tener conversaciones en tiempo real con otros usuarios, sobre temas relacionados con los contenidos de la aplicación para poder comentar ahí que tener el plan básico, con el plan gratuito solo se podrán ver las conversaciones de otros usuarios, pero no se podrán participar en ellas.

- **Perfil del usuario:** Dentro de la aplicación se encuentra un apartado del perfil del usuario donde este podrá ver sus datos personales, y actualizarlos tanto su foto de perfil, como su nickname, así como también podrá actualizar su contraseña y su plan de suscripción si lo desea.

- **Panel de administrador:** Los usuarios pueden asumir roles específicos en la aplicación, lo que determina sus niveles de acceso y las funcionalidades disponibles para ellos. El rol de administrador permite crear, borrar y actualizar los diferentes contenidos de la aplicación igual con las comunidades.

## 2.2 Objetivo del proyecto

El objetivo del proyecto es desarrollar una aplicación web interactiva que permita a los usuarios registrarse, visualizar y comentar contenido multimedia, y participar en comunidades de chat en tiempo real. La aplicación está diseñada para proporcionar una plataforma completa para la interacción del usuario y la compartición de contenido multimedia.

Además, el proyecto también tiene como objetivo ser una herramienta de aprendizaje para aquellos interesados en el desarrollo de aplicaciones web con Spring Boot y Java. Al proporcionar una visión detallada de un proyecto real, se espera que este proyecto pueda ayudar a los lectores a entender mejor los conceptos y prácticas de la ingeniería de software.

La aplicación se desarrollará utilizando Java y el marco de trabajo Spring Boot en el backend, y HTML, CSS, JavaScript y Thymeleaf en el frontend. Se utilizará una base de datos MySQL para almacenar la información del usuario, los detalles de las comunidades de chat, los mensajes de chat y los comentarios sobre el contenido multimedia. Las dependencias del proyecto se manejarán a través de Maven en el backend y npm en el frontend.

Además, se utilizarán diversas herramientas y prácticas de desarrollo de software para garantizar la calidad del código y facilitar la colaboración entre los desarrolladores. Esto incluye el uso de un sistema de control de versiones (Git), pruebas unitarias y de integración, y la integración continua/despliegue continuo (CI/CD).

## 2.3 Características del usuario

En este proyecto, los usuarios pueden tener varias características y roles:

1. **Usuarios Gratis:** Los usuarios con el plan de suscripción "Gratis" pueden acceder a todo el contenido multimedia de la plataforma. Sin embargo, no pueden hacer comentarios sobre los contenidos, participar en las comunidades de usuarios, ver contenido en directo exclusivo.

2. **Usuarios Básicos:** Los usuarios con el plan de suscripción "Básico" pueden acceder a todo el contenido multimedia de la plataforma, hacer comentarios sobre el contenido multimedia, y participar en las comunidades de usuarios. No pueden ver contenido en directo exclusivo.

3. **Usuarios Pro:** Los usuarios con el plan de suscripción "Pro" tienen acceso a todas las funcionalidades de la aplicación. Pueden acceder a todo el contenido multimedia de la plataforma, hacer comentarios sobre el contenido multimedia, participar en las comunidades de usuarios, y ver contenido en directo exclusivo.

4. **Administradores:** Los usuarios con el rol de administrador tienen acceso a funcionalidades adicionales. Pueden crear, borrar y actualizar los diferentes contenidos de la aplicación, así como las comunidades de chat. Para ser administrador, un usuario debe tener asignado el rol de administrador.

Estas características del usuario se reflejan en la clase ‘User’ en el archivo:

src/main/java/com/example/cursospringboot/entity/User.java.

Además, el rol de administrador se define en la clase `Role` en el archivo:

src/main/java/com/example/cursospringboot/entity/Role.java.

En el apartado de implementación se podrá ver todo el codigo al detalle.

## 2.4 Restricciones

Las restricciones del proyecto son las limitaciones o reglas que deben seguirse durante el desarrollo y uso de la aplicación. En este proyecto, las restricciones incluyen:

**1. Sistema Operativo:** La aplicación asume que los usuarios están utilizando Windows como su sistema operativo.

**2. Registro de Usuarios:** Los usuarios deben registrarse para acceder a la mayoría de las funcionalidades de la aplicación. Además, el acceso a ciertos contenidos y la capacidad de realizar ciertas acciones dependen del plan de suscripción del usuario.

**3. Planes de Suscripción:** Los usuarios pueden elegir entre varios planes de suscripción, cada uno con diferentes niveles de acceso y funcionalidades. Por ejemplo, los usuarios con el plan "Gratis" no pueden hacer comentarios sobre los contenidos ni participar en las comunidades de usuarios. Los usuarios con el plan "Básico" pueden hacer comentarios y participar en las comunidades, pero no pueden ver contenido en directo exclusivo. Los usuarios con el plan "Pro" tienen acceso a todas las funcionalidades, incluyendo el contenido en directo exclusivo.

**4. Roles de Usuario:** Los usuarios pueden asumir roles específicos en la aplicación, como el rol de administrador. Los usuarios con el rol de administrador tienen acceso a funcionalidades adicionales, como la capacidad de crear, borrar y actualizar los diferentes contenidos de la aplicación y las comunidades de chat.

**5. Gestión de Dependencias:** Las dependencias del proyecto se manejan a través de Maven en el backend y npm en el frontend. Esto incluye bibliotecas y marcos como Spring Boot para el desarrollo del servidor, Thymeleaf para la plantilla del lado del servidor, y Bootstrap para el diseño y los componentes de la interfaz de usuario.

**6. Prácticas de Desarrollo de Software:** Se utilizan diversas herramientas y prácticas de desarrollo de software para garantizar la calidad del código y facilitar la colaboración entre los desarrolladores. Esto incluye el uso de un sistema de control de versiones (Git), pruebas unitarias y de integración, y la integración continua/despliegue continuo (CI/CD). Además de JavaDoc y el uso de patrones de diseño.

## 2.5 Dependencias

El proyecto tiene varias dependencias que son esenciales para su funcionamiento:

**1. Spring Web:** Esta dependencia proporciona funcionalidades para construir aplicaciones web, incluyendo RESTful, utilizando Spring MVC. Ayuda en la manipulación de solicitudes web, lo que es esencial para la interacción del usuario con la aplicación.

**2. Spring Data JPA:** Esta dependencia simplifica la implementación de la capa de persistencia de datos. Proporciona un marco para implementar repositorios que reducen la cantidad de código de boilerplate. Esto es crucial para el almacenamiento y recuperación de datos de usuario, detalles de las comunidades de chat, mensajes de chat y comentarios sobre el contenido multimedia.

**3. MySQL Driver:** Esta es la dependencia que permite a la aplicación interactuar con la base de datos MySQL. Es esencial para todas las operaciones de la base de datos, como la creación, lectura, actualización y eliminación de datos.

**4. Spring Boot Dev Tools:** Esta dependencia proporciona herramientas de desarrollo útiles, como el reinicio automático de la aplicación cada vez que se cambia un archivo. Esto mejora la eficiencia del desarrollo y la depuración.

**5. Spring Security:** Esta dependencia es un marco de seguridad que proporciona autenticación, autorización y protección contra ataques comunes. Es esencial para garantizar que solo los usuarios autorizados puedan acceder a ciertas funcionalidades de la aplicación.

**6. Spring Session:** Esta dependencia proporciona una API y una implementación para administrar la sesión del usuario y del administrador. Esto es crucial para mantener el estado del usuario entre las solicitudes y para permitir a los usuarios permanecer conectados a la aplicación.

**7. Thymeleaf:** Esta dependencia es un motor de plantillas Java para el desarrollo web en el lado del servidor. Facilita la creación de vistas dinámicas utilizando HTML estándar. Esto es esencial para la creación de la interfaz de usuario de la aplicación.

Estas dependencias se reflejan en el archivo `pom.xml`, que es el archivo de configuración de Maven que define todas las dependencias del proyecto.

Requisitos específicos

3.1 Requisitos de interfaces externas

1. Interfaz de usuario: La aplicación proporciona una interfaz de usuario basada en la web que permite a los usuarios registrarse, iniciar sesión, visualizar y comentar contenido multimedia, y participar en comunidades de chat. La interfaz de usuario es responsiva y compatible con varios navegadores web, incluyendo Edge, Opera GX, Firefox y Chrome. La interfaz de usuario se ha desarrollado utilizando HTML, CSS, JavaScript y Thymeleaf. AJAX se utiliza para realizar solicitudes asíncronas al servidor y actualizar partes de la página web sin necesidad de recargarla por completo.

Los archivos HTML en src/main/resources/templates/ definen la interfaz de usuario de tu aplicación.

2. Interfaz de software: La aplicación se comunica con una base de datos MySQL para almacenar y recuperar datos. La aplicación utiliza el marco de trabajo Spring Boot y el lenguaje de programación Java en el backend para manejar las solicitudes del usuario e interactuar con la base de datos. Las dependencias del proyecto se manejan a través de Maven.

Las clases en src/main/java/com/example/cursospringboot/controller/ definen los controladores que manejan las solicitudes HHTTP de los usuarios.

3. Interfaz de hardware: La aplicación es compatible con dispositivos que tienen un navegador web moderno y una conexión a Internet. No se requiere hardware específico.

3.2 Requisitos funcionales

1. Registro de usuarios: Los usuarios pueden registrarse proporcionando su nombre, correo electrónico y contraseña, además de datos secundarios como fecha de nacimiento, apellidos. También pueden elegir un plan de suscripción durante el registro. La aplicación valida la información proporcionada por el usuario durante el registro y proporciona mensajes de error apropiados si la información no es válida. También ahí manejo de errores en el caso de que el usuario deje el registro a medias o en el caso de que complete el registro de datos, pero no llegue a validar el pago.

2. Inicio de sesión de usuarios: Los usuarios pueden iniciar sesión utilizando su correo electrónico y contraseña. La aplicación autentica al usuario y proporciona mensajes de error apropiados si las credenciales proporcionadas no son válidas.

3. Visualización de contenido multimedia: Los usuarios pueden visualizar una variedad de contenido multimedia en la aplicación. La aplicación proporciona una interfaz de usuario que permite a los usuarios navegar por el contenido multimedia y seleccionar el contenido que desean ver.

4. Comentarios sobre contenido multimedia: Los usuarios pueden comentar sobre el contenido multimedia si tienen un plan de suscripción que lo permite. La aplicación proporciona una interfaz de usuario que permite a los usuarios escribir y publicar comentarios.

5. Comunidades de chat: Los usuarios pueden formar y unirse a comunidades de chat si tienen un plan de suscripción que lo permite. La aplicación proporciona una interfaz de usuario que permite a los usuarios crear, unirse y participar en comunidades de chat.

6. Gestión de perfiles de usuario: Los usuarios pueden ver y actualizar su perfil, incluyendo su foto de perfil, nickname, contraseña y plan de suscripción. La aplicación proporciona una interfaz de usuario que permite a los usuarios ver y modificar su información de perfil.

7. Panel de administrador: Los usuarios con el rol de administrador pueden crear, borrar y actualizar los diferentes contenidos de la aplicación y las comunidades de chat. La aplicación proporciona una interfaz de usuario que permite a los administradores gestionar el contenido y las comunidades de la aplicación. Esta interfaz se complementa también con la del usuario para que el administrador cuando realice alguna modificación en el contenido pueda ver como quedaría el resultado.

Estos requisitos funcionales y de interfaces externas se reflejan en el diseño y la implementación de la aplicación. La implementación de estos requisitos se ha realizado siguiendo las mejores prácticas de desarrollo de software, incluyendo el uso de un sistema de control de versiones (Git), pruebas unitarias y de integración, y la integración continua/despliegue continuo (CI/CD).

3.3 Requisitos de rendimiento

Tiempo de respuesta: La aplicación responde a las solicitudes del usuario en un tiempo razonable. La carga de una página o el procesamiento de una solicitud de usuario no tarda más de unos pocos segundos.

Capacidad: La aplicación puede manejar un gran número de usuarios simultáneos sin degradar significativamente el rendimiento. Esto se logra mediante la implementación de técnicas de escalado, como el balanceo de carga.

Eficiencia de recursos: La aplicación hace un uso eficiente de los recursos del sistema, incluyendo la CPU, la memoria y el almacenamiento. Esto se logra mediante la optimización del código y la gestión eficiente de la memoria.

Disponibilidad: La aplicación está disponible para los usuarios la mayor parte del tiempo. Esto se logra mediante la implementación de técnicas de alta disponibilidad, como la redundancia y la recuperación ante desastres.

3.4 Atributos del sistema de software

Seguridad: La aplicación proporciona mecanismos de seguridad robustos para proteger los datos del usuario y prevenir accesos no autorizados. Esto se logra mediante la autenticación de usuarios, la encriptación de datos y la protección contra ataques comunes, como la inyección de SQL y el cross-site scripting.

Usabilidad: La aplicación es fácil de usar y proporciona una experiencia de usuario agradable. Esto se logra mediante la implementación de una interfaz de usuario intuitiva y la provisión de documentación y soporte al usuario.

Mantenibilidad: La aplicación es fácil de mantener y actualizar. Esto se logra mediante la implementación de un código limpio y bien estructurado, la utilización de patrones de diseño y la provisión de documentación del código.

Portabilidad: La aplicación puede funcionar en diferentes sistemas operativos y navegadores web. Esto se logra mediante la implementación de estándares web y la realización de pruebas en diferentes plataformas.

Escalabilidad: La aplicación puede manejar un aumento en la carga de trabajo aumentando su capacidad. Esto se logra mediante la implementación de técnicas de escalado, como el balanceo de carga y la escalabilidad horizontal.

Arquitectura del sistema

4.1 Descripción de la arquitectura

La arquitectura del sistema se basa en una arquitectura de tres capas, que incluye la capa de presentación, la capa de lógica de negocio y la capa de datos.

**1. Capa de presentación:** Esta capa es la interfaz de usuario y se encarga de interactuar con el usuario. Se ha desarrollado utilizando HTML, CSS, Ajax, JavaScript y Thymeleaf.

**2. Capa de lógica de negocio:** Esta capa se encarga de procesar las solicitudes del usuario, aplicar las reglas de negocio y coordinar las respuestas. Se ha desarrollado utilizando Java y el marco de trabajo Spring Boot.

**3. Capa de datos:** Esta capa se encarga de interactuar con la base de datos para almacenar y recuperar datos. Se ha desarrollado utilizando Java, Spring Data JPA y MySQL.

Spring Boot ha sido mi elección para este tipo de arquitectura debido a varias razones:

**- Simplicidad:** Spring Boot simplifica la configuración y el despliegue de aplicaciones Spring, lo que permite a los desarrolladores centrarse en el código en lugar de en la configuración.

**- Integración:** Spring Boot se integra bien con una variedad de tecnologías, incluyendo JPA para el acceso a datos y Thymeleaf para la creación de vistas.

**-Productividad:** Spring Boot proporciona una serie de características que aumentan la productividad, como la recarga automática de cambios y una serie de herramientas de desarrollo.

**- Escalabilidad:** Spring Boot es altamente escalable, lo que permite a la aplicación manejar un gran número de usuarios simultáneos.

En este proyecto, la arquitectura de tres capas se implementa de la siguiente manera:

- Los controladores en la capa de presentación manejan las solicitudes HTTP del usuario y utilizan los servicios para procesar estas solicitudes.

Aparte de todo el manejo de sesiones del usuario para verificar en todo momento si el usuario está o no registrado, si ha validado o no los datos. Y una vez superado ese filtro hacer las solicitudes HTTP, ya sean métodos GetMapping, PostMapping.

Código de ejemplo para entender la lógica, viendo la función que realiza el controlador.

@GetMapping("/{nombrePelicula}")  
public String getPelicula(@PathVariable String nombrePelicula, Model model, HttpSession session) {  
 Pelicula pelicula = peliculaService.getPeliculaByNombre(nombrePelicula)  
 .orElseThrow(() -> new RuntimeException("Pelicula not found"));  
 model.addAttribute("pelicula", pelicula);  
 User user = (User) session.getAttribute("user");  
 if (user == null || user.getPagoValidado().equals(false) || "Sin Plan".equals(user.getPlanSuscripcion())) {  
 return "login"; // Redirect the user to the login page if not authenticated  
 }  
  
 // Fetch the list of comments for the specific movie  
 List<ComentarioPelicula> comentarios = comentarioPeliculaService.getAllComentariosByPelicula(pelicula);  
 model.addAttribute("comentarios", comentarios);  
 model.addAttribute("session", user);  
  
 // Add the roles of the user to the model  
 Set<Role> roles = user.getRoles();  
 model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.*toList*()));  
 return "indexDetallado";  
}

El controlador también añade atributos al modelo que se envía a la vista.

- Los servicios en la capa de lógica de negocio aplican las reglas de negocio y utilizan los repositorios para interactuar con la base de datos.

Seguimos con el ejemplo de las películas para ver como funciona la lógica, en este caso el servicio:

Dentro del controlador llamamos al servicio y ahí ya podemos usar sus metodos

@Autowired  
private ComentarioPeliculaService comentarioPeliculaService;

Dentro del servicio tenemos una interfaz para definir los métodos y luego su implementación en este caso vemos el método de obtener todos los comentarios de una película, pasándole el nombre de esta.

List<ComentarioPelicula> getAllComentariosByPelicula(Pelicula pelicula);

@Override  
public List<ComentarioPelicula> getAllComentariosByPelicula(Pelicula pelicula) {  
 return comentarioPeliculaRepository.findByPelicula(pelicula);  
}

- Los repositorios en la capa de datos interactúan con la base de datos utilizando Spring Data JPA para simplificar las operaciones de la base de datos.

Seguimos con el ejemplo de las películas para ver cómo funciona la lógica, en este caso el repositorio:

Dentro del servicio invocamos al repositorio, para poder usar JPA y tener acceso a la base de datos.

@Autowired  
private ComentarioPeliculaRepository comentarioPeliculaRepository;

public interface ComentarioPeliculaRepository extends JpaRepository<ComentarioPelicula, Long> {

Interfaz de usuario gráfica, Aplicación, Word

Descripción generada automáticamente

Esta arquitectura permite separar las responsabilidades, lo que facilita la mantenibilidad y la escalabilidad de la aplicación.

4.2 Descripción de los componentes

Controladores: Los controladores son parte de la capa de presentación y se encargan de manejar las solicitudes HTTP del usuario y utilizan los servicios para procesar estas solicitudes.

Servicios: Los servicios son parte de la capa de lógica de negocio y se encargan de aplicar las reglas de negocio y utilizan los repositorios para interactuar con la base de datos.

Repositorios: Los repositorios son parte de la capa de datos y se encargan de interactuar con la base de datos utilizando Spring Data JPA para simplificar las operaciones de la base de datos.

Entidades: Las entidades son objetos de dominio que representan las tablas de la base de datos. Las entidades son utilizadas por los repositorios para mapear los datos de la base de datos a objetos Java.

DTOs (Data Transfer Objects): Los DTOs se utilizan para transferir datos entre las capas de la aplicación. Los DTOs pueden contener datos de múltiples entidades y se utilizan para encapsular los datos que se envían al usuario.

Configuración de seguridad: La configuración de seguridad se encarga de configurar las políticas de seguridad de la aplicación, incluyendo la autenticación y la autorización de los usuarios.

Configuración de la base de datos: La configuración de la base de datos se encarga de configurar la conexión a la base de datos y las propiedades de JPA.

Pruebas: Las pruebas se utilizan para verificar la correcta funcionalidad de la aplicación. Las pruebas incluyen pruebas unitarias y de integración.

Diseño detallado

5.1 Descripción de los módulos

El sistema se divide en varios módulos, cada uno con una funcionalidad específica.

Módulo de Usuario: Este módulo se encarga de todas las operaciones relacionadas con los usuarios, como el registro, inicio de sesión, actualización de perfil y gestión de suscripciones. Este módulo se implementa principalmente en la clase User y utiliza el repositorio UserRepository para interactuar con la base de datos.

Módulo de Contenido: Este módulo se encarga de todas las operaciones relacionadas con el contenido multimedia, como la visualización, adición y actualización de contenido. Este módulo se implementa en las clases Pelicula, F1Content, FootballContent y LiveContent y utiliza los repositorios correspondientes para interactuar con la base de datos.

Módulo de Comentarios: Este módulo se encarga de todas las operaciones relacionadas con los comentarios, como la adición, visualización y eliminación de comentarios. Este módulo se implementa en las clases ComentarioPelicula, ComentarioF1, ComentarioFootball y utiliza los repositorios correspondientes para interactuar con la base de datos.

Módulo de Chat: Este módulo se encarga de todas las operaciones relacionadas con las comunidades de chat, como la creación, unión y gestión de comunidades, así como el envío y recepción de mensajes. Este módulo se implementa en las clases ChatCommunity y ChatMessage y utiliza los repositorios correspondientes para interactuar con la base de datos.

Módulo de Tarjeta de Crédito: Este módulo se encarga de todas las operaciones relacionadas con las tarjetas de crédito de los usuarios, como la adición y actualización de tarjetas de crédito. Este módulo se implementa en la clase TarjetaCredito y utiliza el repositorio TarjetaCreditoRepository para interactuar con la base de datos.

5.2 Diagramas de clases

Interfaz de usuario gráfica

Descripción generada automáticamente

5.2.1 Entidades



User: Esta entidad tiene relaciones con varias otras entidades. Tiene una relación uno a muchos con TarjetaCredito, ComentarioPelicula, ChatCommunity y ChatMessage. También tiene una relación muchos a muchos con Role.

Role: Esta entidad tiene una relación muchos a muchos con User.

TarjetaCredito: Esta entidad tiene una relación muchos a uno con User.

ComentarioPelicula: Esta entidad tiene una relación muchos a uno con User y Pelicula.

ChatCommunity: Esta entidad tiene una relación muchos a uno con User y una relación uno a muchos con ChatMessage.

ChatMessage: Esta entidad tiene una relación muchos a uno con User y ChatCommunity.

Pelicula: Esta entidad es una subclase de Contenido y tiene una relación uno a muchos con ComentarioPelicula.

F1Content: Esta entidad es una subclase de Contenido y tiene una relación uno a muchos con ComentarioF1.

FootballContent: Esta entidad es una subclase de Contenido y tiene una relación uno a muchos con ComentarioFootball.

LiveContent: Esta entidad es una subclase de Contenido.

ComentarioF1: Esta entidad es una subclase de Comentario y tiene una relación muchos a uno con F1Content y User.

ComentarioFootball: Esta entidad es una subclase de Comentario y tiene una relación muchos a uno con FootballContent y User.

Comentario: Esta entidad es una superclase para ComentarioPelicula, ComentarioF1 y ComentarioFootball.

Contenido: Esta entidad es una superclase para Pelicula, F1Content, FootballContent y LiveContent.

5.2.2 Repositorios



Extienden JpaRepository y proporcionan métodos para interactuar con la base de datos. Cada repositorio está asociado a una entidad específica y puede proporcionar métodos personalizados para consultas específicas.

UserRepository: Este repositorio está asociado a la entidad User. Proporciona métodos para verificar si existe un usuario con un correo electrónico o nickname específico.

RoleRepository: Este repositorio está asociado a la entidad Role. Proporciona un método para buscar un rol por su nombre.

TarjetaRepository: Este repositorio está asociado a la entidad TarjetaCredito. Proporciona un método para buscar todas las tarjetas de crédito asociadas a un correo electrónico específico.

ChatCommunityRepository: Este repositorio está asociado a la entidad ChatCommunity. Proporciona un método para buscar una comunidad de chat por su nombre.

ChatMessageRepository: Este repositorio está asociado a la entidad ChatMessage. Proporciona métodos para buscar todos los mensajes enviados por un usuario específico en una comunidad específica, y para buscar todos los mensajes en una comunidad específica.

ComentarioF1Repository: Este repositorio está asociado a la entidad ComentarioF1. Proporciona un método para buscar todos los comentarios asociados a un contenido de F1 específico.

F1ContentRepository: Este repositorio está asociado a la entidad F1Content. Proporciona métodos para buscar un contenido de F1 por su nombre y para buscar contenidos de F1 cuyos nombres comiencen con una cadena específica.

FootballContentRepository: Este repositorio está asociado a la entidad FootballContent. Proporciona métodos para buscar un contenido de fútbol por su nombre, para buscar contenidos de fútbol cuyos nombres comiencen con una cadena específica, y para buscar contenidos de fútbol por su competición.

LiveContentRepository: Este repositorio está asociado a la entidad LiveContent. Proporciona métodos para buscar contenidos en directo por su hora de inicio y de fin, para buscar un contenido en directo por su nombre, para verificar si existen contenidos en directo que se superpongan con un intervalo de tiempo específico, y para buscar contenidos en directo que estén activos o que vayan a emitirse en el futuro.

PeliculaRepository: Este repositorio está asociado a la entidad Pelicula. Proporciona métodos para buscar una película por su nombre y para buscar películas cuyos nombres comiencen con una cadena específica.

ComentarioFootballRepository: Este repositorio se asocia a la entidad ComentarioFootball Proporciona un método para buscar todos los comentarios asociados a un contenido de fútbol específico.

ComentarioPeliculaRepository: Este repositorio se asocia a la entidad ComentarioPelicula. Proporciona un método para buscar todos los comentarios asociados a una película específica.

5.2.3 Servicios



Son clases que contienen la lógica de negocio y coordinan las operaciones entre los controladores y los repositorios.

UserService: Este servicio se encarga de todas las operaciones relacionadas con los usuarios, como el registro, inicio de sesión, actualización de perfil y gestión de suscripciones. Este servicio utiliza el repositorio UserRepository para interactuar con la base de datos.

RoleService: Este servicio se encarga de todas las operaciones relacionadas con los roles de los usuarios. Este servicio utiliza el repositorio RoleRepository para interactuar con la base de datos.

TarjetaCreditoService: Este servicio se encarga de todas las operaciones relacionadas con las tarjetas de crédito de los usuarios, como la adición y actualización de tarjetas de crédito. Este servicio utiliza el repositorio TarjetaCreditoRepository para interactuar con la base de datos.

ComentarioPeliculaService: Este servicio se encarga de todas las operaciones relacionadas con los comentarios de las películas, como la adición, visualización y eliminación de comentarios. Este servicio utiliza el repositorio ComentarioPeliculaRepository para interactuar con la base de datos.

ChatCommunityService: Este servicio se encarga de todas las operaciones relacionadas con las comunidades de chat, como la creación, unión y gestión de comunidades, así como el envío y recepción de mensajes. Este servicio utiliza el repositorio ChatCommunityRepository para interactuar con la base de datos.

ChatMessageService: Este servicio se encarga de todas las operaciones relacionadas con los mensajes de chat, como el envío y recepción de mensajes. Este servicio utiliza el repositorio ChatMessageRepository para interactuar con la base de datos.

ComentarioF1Service: Este servicio se encarga de todas las operaciones relacionadas con los comentarios de los contenidos de F1, como la adición, visualización y eliminación de comentarios. Este servicio utiliza el repositorio ComentarioF1Repository para interactuar con la base de datos.

ComentarioFootballService: Este servicio se encarga de todas las operaciones relacionadas con los comentarios de los contenidos de fútbol, como la adición, visualización y eliminación de comentarios. Este servicio utiliza el repositorio ComentarioFootballRepository para interactuar con la base de datos.

F1ContentService: Este servicio se encarga de todas las operaciones relacionadas con los contenidos de F1, como la adición, visualización y actualización de contenidos. Este servicio utiliza el repositorio F1ContentRepository para interactuar con la base de datos.

FootballContentService: Este servicio se encarga de todas las operaciones relacionadas con los contenidos de fútbol, como la adición, visualización y actualización de contenidos. Este servicio utiliza el repositorio FootballContentRepository para interactuar con la base de datos.

LiveContentService: Este servicio se encarga de todas las operaciones relacionadas con los contenidos en directo, como la adición, visualización y actualización de contenidos. Este servicio utiliza el repositorio LiveContentRepository para interactuar con la base de datos.

PeliculaService: Este servicio se encarga de todas las operaciones relacionadas con las películas, como la adición, visualización y actualización de películas. Este servicio utiliza el repositorio PeliculaRepository para interactuar con la base de datos.

5.2.4 Controlador



Son clases que manejan las solicitudes HTTP de los usuarios y coordinan las operaciones entre los servicios y las vistas.

UserController: Este controlador se encarga de todas las operaciones relacionadas con los usuarios, como el registro, inicio de sesión, actualización de perfil y gestión de suscripciones. Este controlador utiliza el servicio UserService para procesar estas operaciones.

RoleController: Este controlador se encarga de todas las operaciones relacionadas con los roles de los usuarios. Este controlador utiliza el servicio RoleService para procesar estas operaciones.

TarjetaCreditoController: Este controlador se encarga de todas las operaciones relacionadas con las tarjetas de crédito de los usuarios, como la adición y actualización de tarjetas de crédito. Este controlador utiliza el servicio TarjetaCreditoService para procesar estas operaciones.

ComentarioPeliculaController: Este controlador se encarga de todas las operaciones relacionadas con los comentarios de las películas, como la adición, visualización y eliminación de comentarios. Este controlador utiliza el servicio ComentarioPeliculaService para procesar estas operaciones.

ChatCommunityController: Este controlador se encarga de todas las operaciones relacionadas con las comunidades de chat, como la creación, unión y gestión de comunidades, así como el envío y recepción de mensajes. Este controlador utiliza el servicio ChatCommunityService para procesar estas operaciones.

ChatMessageController: Este controlador se encarga de todas las operaciones relacionadas con los mensajes de chat, como el envío y recepción de mensajes. Este controlador utiliza el servicio ChatMessageService para procesar estas operaciones.

ComentarioF1Controller: Este controlador se encarga de todas las operaciones relacionadas con los comentarios de los contenidos de F1, como la adición, visualización y eliminación de comentarios. Este controlador utiliza el servicio ComentarioF1Service para procesar estas operaciones.

ComentarioFootballController: Este controlador se encarga de todas las operaciones relacionadas con los comentarios de los contenidos de fútbol, como la adición, visualización y eliminación de comentarios. Este controlador utiliza el servicio ComentarioFootballService para procesar estas operaciones.

F1ContentController: Este controlador se encarga de todas las operaciones relacionadas con los contenidos de F1, como la adición, visualización y actualización de contenidos. Este controlador utiliza el servicio F1ContentService para procesar estas operaciones.

FootballContentController: Este controlador se encarga de todas las operaciones relacionadas con los contenidos de fútbol, como la adición, visualización y actualización de contenidos. Este controlador utiliza el servicio FootballContentService para procesar estas operaciones.

LiveContentController: Este controlador se encarga de todas las operaciones relacionadas con los contenidos en directo, como la adición, visualización y actualización de contenidos. Este controlador utiliza el servicio LiveContentService para procesar estas operaciones.

PeliculaController: Este controlador se encarga de todas las operaciones relacionadas con las películas, como la adición, visualización y actualización de películas. Este controlador utiliza el servicio PeliculaService para procesar estas operaciones.

Implementación

6.1 Entorno de desarrollo

El entorno de desarrollo para este proyecto incluye:

Sistema Operativo: Windows

IDE: IntelliJ IDEA 2023.2.5

Lenguaje de Programación: Java y JavaScript

Marco de trabajo: Spring Boot

**6.1.2 ¿Por qué he decidio usar Spring boot y no otras tecnologías?**

Cuando creamos una aplicación tenemos que gestionar las dependencias y tenemos que conocer que librerías vamos a utilizar en nuestro proyecto, también tenemos que desarrollar la aplicación y desplegarla.

Lo que hace Spring Boot es gestionar las librerías que vamos a necesitar de forma automática, simplemente diciéndole que tipo de proyecto a grandes rasgos vamos a crear y el y se encarga automáticamente de según sea la aplicación usar unas librerías u otras. Para que solo nos tengamos que centrar en el desarrollo de la aplicación.

También, para desplegar la aplicación Spring Boot nos ayuda teniendo un servidor apache tomcat embebido y solamente con un comando podemos desplegar nuestro proyecto y correrlo en el servidor tomcat.

Con todo esto solo nos tenemos que centrar en el desarrollo de la aplicación y hace que crear una aplicación sea mucho más rápido y principalmente solo tengamos que preocuparnos en su desarrollo y no en todo lo que hay entes y todo lo que hay después.

Gestor de Dependencias: Maven

Sistema de Control de Versiones: Git

Base de Datos: MySQL

6.2 Proceso de instalación

Primero voy a enseñar como se instala todo partiendo de cero el proyecto, luego explicare como instalarlo con los repositorios de GitHub.

El entorno de desarrollo que he utilizado para el proyecto es IntelliJ IDEA 2023.2.5, pero este es de pago, con cuentas educativas es gratuito, se pueden utilizar otros entornos para el manejo de Spring Boot y java como Eclipse, NetBeans, Visual Studio Code, entre otros.

6.2.1 Configuración Básica de IntelliJ

Captura de pantalla de computadora

Descripción generada automáticamente

Creamos un nuevo proyecto y seleccionamos la opción de Spring Initializr, el lenguaje Java, de tipo Maven, y si es posible la última versión de Java y del JDK, en mi caso inicie el proyecto en enero de 2024.

6.2.2 Configuración de las dependencias

En el siguiente panel se gestionan las dependencias, que es ir añadiendo una por una.

Captura de pantalla de computadora

Descripción generada automáticamente

En total son:

spring-boot-starter-data-jpa

spring-boot-starter-thymeleaf

spring-session-core

spring-boot-starter-websocket

spring-boot-starter-web

spring-boot-devtools

mysql-connector-j

spring-boot-starter-test

spring-security-test

6.2.3 Estructurar el proyecto

Una vez realizadas las configuraciones básicas y las configuraciones de dependencias pasamos a la estructura de directorios:

**src/main/java:** Este directorio contiene todo tu código fuente de Java. Aquí es donde se crearán los controladores, servicios, repositorios, etc.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

src/main/resources: Este directorio contiene recursos como archivos de propiedades, archivos de configuración de Spring Boot y archivos HTML de Thymeleaf.

Texto

Descripción generada automáticamente

src/test/java: Este directorio contiene el código de prueba.

**Paquetes:** Dentro de **src/main/java**, debes crear paquetes para organizar tu código. Aquí hay algunos paquetes que podrías crear:

com.example.cursospringboot.controller: Este paquete contendrá todos tus controladores.

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

com.example.cursospringboot.service: Este paquete contendrá todos tus servicios.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

com.example.cursospringboot.repository: Este paquete contendrá todos tus repositorios.

Imagen que contiene Texto

Descripción generada automáticamente

com.example.cursospringboot.entity: Este paquete contendrá todas tus entidades o modelos.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

**Clases:** Dentro de estos paquetes, se crearán las clases para los controladores, servicios, repositorios y entidades. Por ejemplo, puedes la clase UserController en el paquete de controladores, la clase UserService en el paquete de servicios, la interfaz UserRepository en el paquete de repositorios, y la clase User en el paquete de entidades.

**Recursos:** En **src/main/resources**:

application.properties: Este archivo contiene la configuración de la aplicación, como la cadena de conexión a la base de datos.

Archivos HTML de Thymeleaf: Estos archivos definen las vistas de la aplicación.

**Dependencias:** En el archivo **pom.xml**, se debe definir todas las dependencias que el proyecto necesita, como Spring Boot, Spring Data JPA, Thymeleaf, etc.

**Pruebas:** En **src/test/java**, para crear pruebas para el código. Se puede tener un paquete para cada tipo de prueba (por ejemplo, pruebas de controlador, pruebas de servicio, etc.).

**Compilación y ejecución:** Para poder compilar y ejecutar la aplicación utilizando Maven. El comando **mvn clean install** compila la aplicación y crea un archivo JAR en el directorio target. El comando **mvn spring-boot:run** ejecuta la aplicación.

6.3 Código fuente

Spring Boot y la generación de la base de datos

Spring Boot utiliza el framework **Hibernate** para mapear las clases de entidad a tablas en la base de datos. Las anotaciones como **@Entity**, **@Table**, **@Column** y **@Id** se utilizan para definir cómo se realiza el mapeo. Cuando la aplicación se inicia, Hibernate genera automáticamente las tablas de la base de datos basándose en las clases de entidad. Esto significa que no es necesario crear manualmente las tablas en la base de datos; en su lugar, simplemente se definen las clases de entidad en el código e Hibernate se encarga del resto. Esto facilita el desarrollo, ya que los cambios en la estructura de la base de datos se pueden realizar simplemente modificando las clases de entidad. Además, como el esquema de la base de datos se genera a partir del código, siempre está sincronizado con el código de la aplicación.

6.3.1 Modelos

6.3.1.1 Contenidos

**Clase Contenido**

La clase Contenido es una clase abstracta que sirve como superclase para todas las clases de contenido (películas, contenido de f1 y fútbol, contenido en direct). Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/Contenido.java.

@MappedSuperclass  
public abstract class Contenido {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
 @Column(name = "nombreContenido", nullable = false, length = 100)  
 private String nombreContenido;  
 @Column(name = "descripcion", length = 1000, nullable = false)  
 private String descripcion;  
 @Column(name = "anho", length = 5, nullable = false)  
 private Integer anho;  
 @Column(name = "url\_image", length = 100, nullable = false)  
 private String url\_image;  
 @Column(name = "url\_video", length = 100, nullable = false)  
 private String url\_video;  
 // getters and setters  
}

Esta clase define varios campos comunes a todos los tipos de contenido, como nombreContenido, descripcion, anho, url\_image y url\_video. La anotación @MappedSuperclass indica que esta clase no se mapeará a una tabla en la base de datos, sino que sus campos se incluirán en las tablas de sus subclases.

**Clase Película**

La clase Pelicula es una subclase de Contenido que representa las películas en la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/Pelicula.java.

@Entity  
@Table(name = "Pelicula")  
public class Pelicula extends Contenido {  
 @Column(name = "tituloOriginal", length = 50, nullable = false)  
 private String tituloOriginal;  
 @Column(name = "genero", length = 50, nullable = false)  
 private String genero;  
 @Column(name = "nacionalidad", length = 50, nullable = false)  
 private String nacionalidad;  
 @Column(name = "duracion", length = 5, nullable = false)  
 private Integer duracion;  
 @Column(name = "distribuidora", length = 50, nullable = false)  
 private String distribuidora;  
 @Column(name = "director", length = 50, nullable = false)  
 private String director;  
 @Column(name = "clasificacionEdad", length = 3, nullable = false)  
 private Short clasificacionEdad;  
 @Column(name = "otrosDatos", length = 150, nullable = false)  
 private String otrosDatos;  
 @Column(name = "actores", length = 200, nullable = false)  
 private String actores;  
 @Column(name = "calificacion", length = 3, nullable = false)  
 private double calificacion;  
 // getters and setters  
}

Esta clase añade varios campos específicos de las películas, como tituloOriginal, genero, nacionalidad, duracion, distribuidora, director, clasificacionEdad, otrosDatos, actores y calificacion. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos.

**Repositorio Película**

El repositorio PeliculaRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de películas en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/PeliculaRepository.java.

@Repository  
public interface PeliculaRepository extends JpaRepository<Pelicula, Long> {  
 Optional<Pelicula> findByNombreContenido(String nombreContenido);  
 List<Pelicula> findByNombreContenidoStartingWith(String query);  
}

Los métodos findByNombreContenido y findByNombreContenidoStartingWith son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten buscar películas por su nombre de contenido.

Servicio Película

El servicio PeliculaServiceImp es una clase que implementa la interfaz PeliculaService, proporcionando la lógica de negocio para las operaciones relacionadas con las películas. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/PeliculaServiceImp.java.

@Service  
public class PeliculaServiceImp implements PeliculaService{  
  
 @Autowired  
 private PeliculaRepository peliculaRepository;  
  
 @Override  
 public List<Pelicula> getAllPeliculas() {  
 return peliculaRepository.findAll();  
 }  
  
 @Override  
 public Optional<Pelicula> getPeliculaByNombre(String nombrePelicula) {  
 return peliculaRepository.findByNombreContenido(nombrePelicula);  
 }  
  
 @Override  
 @Transactional  
 public Pelicula createPelicula(Pelicula pelicula) {  
 return peliculaRepository.save(pelicula);  
 }  
  
 @Override  
 @Transactional  
 public Pelicula updatePelicula(String nombrePelicula, Pelicula detallesPelicula) {  
 // Implementación del método  
 }  
  
 @Override  
 @Transactional  
 public void deletePelicula(String nombrePelicula) {  
 // Implementación del método  
 }  
  
 @Override  
 public List<Pelicula> searchPeliculas(String query) {  
 return peliculaRepository.findByNombreContenidoStartingWith(query);  
 }  
}

Esta clase utiliza PeliculaRepository para interactuar con la base de datos. Proporciona métodos para obtener todas las películas, obtener una película por su nombre, crear, actualizar y eliminar películas, y buscar películas por una cadena de consulta.

Los métodos están anotados con @Transactional para indicar que deben ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

El método **createPelicula** toma un objeto Pelicula como parámetro y lo guarda en la base de datos utilizando el método save del repositorio.

El método **updatePelicula** toma el nombre de una película y un objeto Pelicula con los detalles actualizados como parámetros. Primero busca la película en la base de datos utilizando el método findByNombreContenido del repositorio. Si la película se encuentra, actualiza sus detalles y luego la guarda en la base de datos.

El método **deletePelicula** toma el nombre de una película como parámetro, busca la película en la base de datos y, si la encuentra, la elimina utilizando el método delete del repositorio.

El método **searchPeliculas** toma una cadena de consulta como parámetro y devuelve una lista de películas cuyos nombres comienzan con esa cadena, lo cual nos va a servir para la implementación de un buscador utilizando el método findByNombreContenidoStartingWith del repositorio.

**Clase F1Content**

La clase F1Content es una subclase de Contenido que representa el contenido de F1 en la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/ F1Content.java.

@Entity

@Table(name = "F1Content")

public class F1Content extends Contenido {

    @Column(name = "circuito", length = 50, nullable = false)

    private String circuito;

    @Column(name = "equipos", length = 500, nullable = false)

    private String equipos;

    @Column(name = "nacionalidad", length = 50, nullable = false)

    private String nacionalidad;

    @Column(name = "duracion", length = 5, nullable = false)

    private Integer duracion;

    @Column(name = "otrosDatos", length = 150, nullable = false)

    private String otrosDatos;

    @Column(name = "pilotos", length = 500, nullable = false)

    private String pilotos;

    @OneToMany(mappedBy = "f1Content", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<ComentarioF1> comentarios;

// getters and setters   
}

Esta clase añade varios campos específicos del contenido de F1, como circuito, equipos, nacionalidad, duracion, otrosDatoa, pilotos. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos. La anotación @OneToMany establece la relación con la tabla de comentarios de contenido de F1.

**Repositorio F1Content**

El repositorio F1ContentRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de de F1Contet en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/ F1ContentRepository.java.

@Repository

public interface F1ContentRepository extends JpaRepository<F1Content, Long> {

    Optional<F1Content> findByNombreContenido(String nombreContenido);

    List<F1Content> findByNombreContenidoStartingWith(String query);

}

Los métodos findByNombreContenido y findByNombreContenidoStartingWith son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten buscar carreras de F1 por su nombre de contenido. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicio F1Content**

El servicio F1ContentServiceImp es una clase que implementa la interfaz F1ContentService, proporcionando la lógica de negocio para las operaciones relacionadas con el contenido de F1. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/F1ContentServiceImp.java.

@Service

public class F1ContentServiceImp  implements F1ContentService{

    @Autowired

    private F1ContentRepository f1ContentRepository;

    // Implementación de los métodos del repositorio

    @Override

    public List<F1Content> getAllF1Content() {

        return f1ContentRepository.findAll();

    }

    @Override

    public Optional<F1Content> getF1ContentByNombreCarrera(String nombreCarrera) {

        return f1ContentRepository.findByNombreContenido(nombreCarrera);

    }

    @Override

    public F1Content createF1Content(F1Content f1Content) {

        return f1ContentRepository.save(f1Content);

    }

    @Override

    public F1Content updateF1Content(String nombreCarrera, F1Content f1ContentDetails) {

        F1Content f1Content = f1ContentRepository.findByNombreContenido(nombreCarrera)

                .orElseThrow(() -> new RuntimeException("F1Content not found"));

        f1Content.setAnho(f1ContentDetails.getAnho());

        f1Content.setCircuito(f1ContentDetails.getCircuito());

        f1Content.setDuracion(f1ContentDetails.getDuracion());

        f1Content.setEquipos(f1ContentDetails.getEquipos());

        f1Content.setNacionalidad(f1ContentDetails.getNacionalidad());

        f1Content.setDescripcion(f1ContentDetails.getDescripcion());

        f1Content.setPilotos(f1ContentDetails.getPilotos());

        f1Content.setNombreContenido(f1ContentDetails.getNombreContenido());

        f1Content.setDuracion(f1ContentDetails.getDuracion());

        f1Content.setOtrosDatos(f1ContentDetails.getOtrosDatos());

        f1Content.setUrl\_image(f1ContentDetails.getUrl\_image());

        f1Content.setUrl\_video(f1ContentDetails.getUrl\_video());

        // Actualiza otros campos según sea necesario

        return f1ContentRepository.save(f1Content);

    }

    @Override

    public void deleteF1Content(String nombreCarrera) {

        F1Content f1 = f1ContentRepository.findByNombreContenido(nombreCarrera)

                .orElseThrow(() -> new RuntimeException("Carrera de F1 not found"));

        f1ContentRepository.delete(f1);

    }

    @Override

    public List<F1Content> searchF1Content(String query) {

        return f1ContentRepository.findByNombreContenidoStartingWith(query);

    }

}

Esta clase utiliza F1ContentRepository para interactuar con la base de datos. Proporciona métodos para obtener todos los contenidos de F1, obtener un contenido por su nombre, crear, actualizar y eliminar carreras de F1, y buscar carreras de F1 por una cadena de consulta.

Los métodos están anotados con @Transactional para indicar que deben ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

El método **createF1Content** toma un objeto de tipo f1Content como parámetro y lo guarda en la base de datos utilizando el método save del repositorio.

El método **updateF1Content** toma el nombre de un contenido de F1 y un objeto de tipo f1Content con los detalles actualizados como parámetros. Primero busca el contenido en la base de datos utilizando el método findByNombreContenido del repositorio. Si el contenido se encuentra, actualiza sus detalles y luego la guarda en la base de datos.

El método **deleteF1Content** toma el nombre de un contenido de F1 como parámetro, busca el contenido en la base de datos y, si lo encuentra, lo elimina utilizando el método delete del repositorio.

El método **searchF1Content** toma una cadena de consulta como parámetro y devuelve una lista de contenidos de F1 cuyos nombres comienzan con esa cadena, lo cual nos va a servir para la implementación de un buscador utilizando el método findByNombreContenidoStartingWith del repositorio.

El método **getAllF1Content** simplemente devuelve todos los contenidos de F1 almacenados en la base de datos.

**Clase FootballContent**

La clase FootballContent es una subclase de Contenido que representa el contenido de fútbol en la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/ FootballContent.java.

@Entity

@Table(name = "FootballContent")

public class FootballContent extends Contenido{

    @Column(name = "estadio", length = 50, nullable = false)

    private String estadio;

    @Column(name = "equipos", length = 50, nullable = false)

    private String equipos;

    @Column(name = "competicion", length = 50, nullable = false)

    private String competicion;

    @Column(name = "duracion", length = 5, nullable = false)

    private Integer duracion;

    @Column(name = "otrosDatos", length = 150, nullable = false)

    private String otrosDatos;

    @Column(name = "jugadores", length = 200, nullable = false)

    private String jugadores;

    @OneToMany(mappedBy = "footballContent", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<ComentarioFootball> comentarios;

// getters and setters   
}

Esta clase añade varios campos específicos del contenido de fútbol, como estadio, equipos, competicion, duracion, otrosDatoa, jugadores. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos. La anotación @OneToMany establece la relación con la tabla de comentarios de contenido de fútbol.

**Repositorio FootballContent**

El repositorio FootballContentRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de de FootballContent en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/ FootballContentRepository.java.

@Repository

public interface FootballContentRepository extends JpaRepository<FootballContent, Long> {

    Optional<FootballContent> findByNombreContenido(String nombreContenido);

    List<FootballContent> findByNombreContenidoStartingWith(String query);

    List<FootballContent> findByCompeticion(String competicion);

}

Los métodos findByNombreContenido y findByNombreContenidoStartingWith son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten buscar partidos de fútbol por su nombre de contenido. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador. El método de findByCompeticion , sirve clasificar el contenido de fútbol en base a la competición.

**Servicios FootballContent**

El servicio FootballContentServiceImp implementa la interfaz FootballContentService, proporcionando la lógica de negocio para las operaciones relacionadas con el contenido de fútbol. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/FootballContentServiceImp.java.

@Service

public class FootballContentServiceImp implements FootballContentService{

    @Autowired

    private FootballContentRepository footballContentRepository;

    // Implementación de los métodos del repositorio

    @Override

    public List<FootballContent> getAllFootballContent() {

        return footballContentRepository.findAll();

    }

    @Override

    public Optional<FootballContent> getFootballContentByNombrePartido(String nombrePartido) {

        return footballContentRepository.findByNombreContenido(nombrePartido);

    }

    @Override

    public FootballContent createFootballContent(FootballContent footballContent) {

        return footballContentRepository.save(footballContent);

    }

    @Override

    public FootballContent updateFootballContent(String nombrePartido, FootballContent footballContentDetails) {

        FootballContent footballContent = footballContentRepository.findByNombreContenido(nombrePartido)

                .orElseThrow(() -> new RuntimeException("FootballContent not found"));

        footballContent.setAnho(footballContentDetails.getAnho());

        footballContent.setDescripcion(footballContentDetails.getDescripcion());

        footballContent.setDuracion(footballContentDetails.getDuracion());

        footballContent.setEquipos(footballContentDetails.getEquipos());

        footballContent.setEstadio(footballContentDetails.getEstadio());

        footballContent.setJugadores(footballContentDetails.getJugadores());

        footballContent.setCompeticion(footballContentDetails.getCompeticion());

        footballContent.setAnho(footballContentDetails.getAnho());

        footballContent.setOtrosDatos(footballContentDetails.getOtrosDatos());

        footballContent.setNombreContenido(footballContentDetails.getNombreContenido());

        footballContent.setUrl\_image(footballContentDetails.getUrl\_image());

        footballContent.setUrl\_video(footballContentDetails.getUrl\_video());

        // Actualiza otros campos según sea necesario

        return footballContentRepository.save(footballContent);

    }

    @Override

    public void deleteFootballContent(String nombrePartido) {

        FootballContent footballContent = footballContentRepository.findByNombreContenido(nombrePartido)

                .orElseThrow(() -> new RuntimeException("Carrera de F1 not found"));

        footballContentRepository.delete(footballContent);

    }

    @Override

    public List<FootballContent> searchFootballContent(String query) {

        return footballContentRepository.findByNombreContenidoStartingWith(query);

    }

    @Override

    public List<FootballContent> getFootballContentByCompeticion(String competicion) {

        return footballContentRepository.findByCompeticion(competicion);

    }

}

Esta clase utiliza FootballContentRepository para interactuar con la base de datos. Proporciona métodos para obtener todos los contenidos de fútbol, obtener un contenido por su nombre, crear, actualizar y eliminar partidos de fútbol, y buscar partidos por una cadena de consulta.

Los métodos están anotados con @Transactional para indicar que deben ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

El método **createFootballContent** toma un objeto de tipo footballContent como parámetro y lo guarda en la base de datos utilizando el método save del repositorio.

El método **updateFootballContentt** toma el nombre de un contenido de fútbol y un objeto de tipo footballContent con los detalles actualizados como parámetros. Primero busca el contenido en la base de datos utilizando el método findByNombreContenido del repositorio. Si el contenido se encuentra, actualiza sus detalles y luego la guarda en la base de datos.

El método **deleteFootballContent** toma el nombre de un contenido de fútbol como parámetro, busca el contenido en la base de datos y, si lo encuentra, lo elimina utilizando el método delete del repositorio.

El método **searchFootballContent** toma una cadena de consulta como parámetro y devuelve una lista de contenidos de fútbol cuyos nombres comienzan con esa cadena, nos sirve para la implementación de un buscador utilizando el método findByNombreContenidoStartingWith del repositorio.

El método **getAllFoorballContent** simplemente devuelve todos los contenidos de fútbol almacenados en la base de datos.

**Clase LiveContent**

La clase LiveContent es una subclase de Contenido que representa el contenido en directo en la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/LiveContent.java.

@Entity

@Table(name = "LiveContent")

public class LiveContent extends Contenido {

    @Column(name = "startTime",  nullable = false, length = 20)

    private LocalDateTime startTime;

    @Column(name = "endTime",  nullable = false, length = 20)

    private LocalDateTime endTime;

    @Column(name = "type", nullable = false, length = 50)

    private String type;

// getters and setters   
}

Esta clase añade varios campos específicos del contenido en directo, como startTime, endTime, type. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos.

**Repositorio LiveContent**

LiveContentRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar), tambien operaciones para el manejo del contenido en directo como devolver los contenidos futuros, saber si un contenido está en una determinada franja horaria. En la tabla de de LiveContent en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/LiveContentRepository.java.

@Repository

public interface LiveContentRepository extends JpaRepository<LiveContent, Long> {

    List<LiveContent> findByStartTime(LocalDateTime startTime);

    List<LiveContent> findByEndTime(LocalDateTime endTime);

    Optional<LiveContent> findByNombreContenido(String nombreContenido);

    // Verificar si ahi otros contenidos en directo en ese tramo horario

    @Query("SELECT CASE WHEN COUNT(lc) > 0 THEN true ELSE false END FROM LiveContent lc WHERE " +

            "(lc.startTime <= :endTime AND lc.endTime >= :startTime)")

    boolean existsOverlappingLiveContents(LocalDateTime startTime, LocalDateTime endTime);

    @Query("SELECT lc FROM LiveContent lc WHERE lc.startTime <= :now AND lc.endTime >= :now")

    List<LiveContent> findActiveLiveContents(@Param("now") LocalDateTime now);

    @Query("SELECT lc FROM LiveContent lc WHERE lc.startTime > :now ORDER BY lc.startTime")

    List<LiveContent> findFutureLiveContents(@Param("now") LocalDateTime now);

}

Los métodos findActiveLiveContents y findFutureLiveContents sirven para tener un controlor de los contenidos en directo, obteniendo por un lado el contenido que esta en directo en ese instante y los futuros contenidos en directo, los cuales son muy valiosos para el diseño final de la aplicación. El otro método existsOverlappingLiveContents, simplemente nos devuelve un booleano para saber si ahí contenido o no.

**Servicio LiveContent**

El servicio LiveContentServiceImp implementa la interfaz LiveContentService, proporcionando la lógica de negocio para las operaciones relacionadas con el contenido en directo. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/LiveContentServiceImp.java.

@Service

public class LiveContentServiceImp implements LiveContentService {

    @Autowired

    private LiveContentRepository liveContentRepository;

    @Override

    public List<LiveContent> getAllLiveContents() {

        return liveContentRepository.findAll();

    }

    @Override

    public LiveContent getLiveContentById(Long id) {

        return liveContentRepository.findById(id).orElse(null);

    }

    @Override

    public LiveContent createLiveContent(LiveContent liveContent) {

        return liveContentRepository.save(liveContent);

    }

    @Override

    public LiveContent getLiveContentByNombre(String nombreContenido) {

        return liveContentRepository.findByNombreContenido(nombreContenido).orElseThrow(() -> new RuntimeException("Contenido en directo no encontrado"));

    }

    @Override

    @Transactional

    public LiveContent updateLiveContent(String nombreContenido, LiveContent detallesLiveContent) {

        LiveContent liveContent = liveContentRepository.findByNombreContenido(nombreContenido).orElseThrow(() -> new RuntimeException("Contenido en directo no encontrado"));

        liveContent.setNombreContenido(detallesLiveContent.getNombreContenido());

        liveContent.setDescripcion(detallesLiveContent.getDescripcion());

        liveContent.setUrl\_image(detallesLiveContent.getUrl\_image());

        liveContent.setUrl\_video(detallesLiveContent.getUrl\_video());

        liveContent.setStartTime(detallesLiveContent.getStartTime());

        liveContent.setEndTime(detallesLiveContent.getEndTime());

        liveContent.setType(detallesLiveContent.getType());

        liveContent.setAnho(detallesLiveContent.getAnho());

        return liveContentRepository.save(liveContent);

    }

    @Override

    public void deleteLiveContent(Long id) {

        liveContentRepository.deleteById(id);

    }

    @Override

    public List<LiveContent> getLiveContentByStartTime(LocalDateTime startTime) {

        return liveContentRepository.findByStartTime(startTime);

    }

    @Override

    public List<LiveContent> getLiveContentByEndTime(LocalDateTime endTime) {

        return liveContentRepository.findByEndTime(endTime);

    }

    @Override

    public List<LiveContent> getFutureLiveContents() {

        LocalDateTime now = LocalDateTime.now();

        return liveContentRepository.findFutureLiveContents(now);

    }

    @Override

    public boolean existsOverlappingLiveContents(LocalDateTime startTime, LocalDateTime endTime) {

        return liveContentRepository.existsOverlappingLiveContents(startTime, endTime);

    }

@Override

    public LiveContent getCurrentLiveContent() {

        LocalDateTime now = LocalDateTime.now();

        List<LiveContent> liveContents = liveContentRepository.findActiveLiveContents(now);

        for (LiveContent liveContent : liveContents) {

            LocalDateTime startTime = liveContent.getStartTime();

            LocalDateTime endTime = liveContent.getEndTime();

            if (now.isAfter(startTime) && now.isBefore(endTime)) {

                return liveContent;

            }

        }

        return null;

    }

}

Esta clase utiliza LiveContentService para interactuar con la base de datos. Proporciona métodos para obtener todos los contenidos en drecto, obtener un contenido por su nombre, crear, actualizar y eliminar contenidos en directo, obtener contenidos en base al horario.

Los métodos están anotados con @Transactional para indicar que deben ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

El método **getAllLiveContents** recupera una lista de todos los contenidos en directo almacenados en la base de datos. Es útil para mostrar todos los contenidos disponibles a los usuarios y retorna una lista que contiene todos los contenidos en directo.

El método **createLiveContent** permite la creación de un nuevo contenido en directo y su almacenamiento en la base de datos. Recibe los detalles del nuevo contenido y lo guarda. Recibe como parámetros liveContent: Un objeto que contiene los detalles del contenido en directo que se va a crear y retorna El contenido en directo recién creado.

El método **getLiveContentByNombre** busca y recupera un contenido en directo utilizando su nombre. Si no se encuentra un contenido con el nombre especificado, lanza una excepción indicando que no se encontró el contenido. Tiene como parámetros El nombre del contenido en directo que se desea recuperar y retorna El contenido en directo correspondiente al nombre proporcionado.

El método **updateLiveContent** permite la actualización de los detalles de un contenido en directo existente. Busca el contenido por su nombre, actualiza sus atributos y guarda los cambios en la base de datos. Tiene como parámetros el nombre del contenido en directo que se desea actualizar y un objeto con los nuevos detalles del contenido en directo. Retorna el contenido en directo actualizado.

El método **deleteLiveContent** elimina un contenido en directo específico de la base de datos utilizando su ID y tiene como parámetros El identificador único del contenido en directo que se desea eliminar.

El método **getLiveContentByStartTime** recupera una lista de contenidos en directo que comienzan en una hora específica. Tiene como parámetros la hora de inicio de los contenidos en directo que se desean recuperar y retorna Una lista de contenidos en directo que comienzan en la hora especificada.

El método **getLiveContentByEndTime** recupera una lista de contenidos en directo que terminan en una hora específica. Tiene como parámetros La hora de finalización de los contenidos en directo que se desean recuperar y retorna Una lista de contenidos en directo que terminan en la hora especificada.

El método **getCurrentLiveContent** busca y recupera el contenido en directo que está activo en el momento actual. Un contenido se considera activo si la hora actual está entre su hora de inicio y su hora de finalización y retorna el contenido en directo que está actualmente activo, o null si no hay ningún contenido activo.

El método **getFutureLiveContents** recupera una lista de contenidos en directo que están programados para empezar en el futuro, ordenados por su hora de inicio y retorna Una lista de contenidos en directo que comenzarán en el futuro.

El método **existsOverlappingLiveContents** verifica si existe algún contenido en directo que se solape con un intervalo de tiempo específico. Es útil para evitar conflictos de programación. Tiene como parámetros la hora de inicio del intervalo de tiempo a verificar y la hora de finalización del intervalo de tiempo a verificar. Retorna un valor booleano que indica si existen o no contenidos en directo que se solapen con el intervalo de tiempo proporcionado.

6.3.1.2 Comentarios

Clase Comentario

La clase Comentario es una clase abstracta que sirve como superclase para todas las clases de comentario de películas, contenido de f1 y fútbol. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/Comentario.java.

@MappedSuperclass

public abstract class Comentario{

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @Column(name = "texto",length = 500, nullable = false)

    private String texto;

    @Column(name = "fechaComentario", length = 12, nullable = false)

    private LocalDate fechaComentario;

    @Column(name = "user\_nickname", nullable = false)

    private String nickname;

// getters and setters   
}

Esta clase define varios campos comunes a todos los tipos de comentarios, como Id, texto, fechaComentario, userNickname. La anotación @MappedSuperclass indica que esta clase no se mapeará a una tabla en la base de datos, sino que sus campos se incluirán en las tablas de sus subclases.

**Clase Comentario de Películas**

La clase ComentarioPelicula es una subclase de Comentario que representa los comentarios de los diferentes contenidos de películas de la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/ComentarioPelicula.java.

@Entity

@Table(name = "ComentarioPelicula")

public class ComentarioPelicula extends Comentario {

    @Column(name = "valoracion", nullable = false)

    private Short valoracion;

    @ManyToOne

    @JoinColumn(name = "pelicula\_id", referencedColumnName = "id", nullable = false)

    private Pelicula pelicula;

    @ManyToOne

    @JoinColumn(name = "user\_email", referencedColumnName = "email",  nullable = false)

    private User usuario;

// getters and setters   
}

Esta clase añade varios campos específicos a los comentarios de las películas, como valoración. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos. La anotación @OneToMany establece la relación con la tabla de películas recogiendo el atributo del id de la película y con la tabla de los usuarios cogiendo el atributo del email de usuario.

**Repositorio Comentario de Película**

El repositorio ComentarioPeliculaRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de ComentarioPelicula en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/ComentarioPeliculaRepository.java.

@Repository

public interface ComentarioPeliculaRepository extends JpaRepository<ComentarioPelicula, Long> {

    List<ComentarioPelicula> findByPelicula(Pelicula pelicula);

}

Los métodos findByPelicula son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten obtener los comentarios de una determinada película. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicio Comentario de Película**

El servicio ComentarioPeliculaServiceImp implementa la interfaz ComentarioPeliculaService, proporcionando la lógica de negocio para las operaciones relacionadas con los comentarios de los contenidos que sean películas. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/ComentarioPeliculaServiceImp.java.

@Service

public class ComentarioPeliculaServiceImp implements ComentarioPeliculaService {

    @Autowired

    private ComentarioPeliculaRepository comentarioPeliculaRepository;

    @Autowired

    private PeliculaService peliculaService;

    @Override

    public List<ComentarioPelicula> getAllComentariosPelicula() {

        return comentarioPeliculaRepository.findAll();

    }

    @Override

    public Optional<ComentarioPelicula> getComentarioPeliculaById(Long id) {

        return comentarioPeliculaRepository.findById(id);

    }

    @Override

    public ComentarioPelicula createComentarioPelicula(ComentarioPelicula comentarioPelicula) {

        return comentarioPeliculaRepository.save(comentarioPelicula);

    }

    @Override

    public ComentarioPelicula updateComentarioPelicula(Long id, ComentarioPelicula comentarioPeliculaDetails) {

        ComentarioPelicula comentarioPelicula = getComentarioPeliculaById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioPelicula not found with id " + id));

        comentarioPelicula.setTexto(comentarioPeliculaDetails.getTexto());

        comentarioPelicula.setValoracion(comentarioPeliculaDetails.getValoracion());

        comentarioPelicula.setNickname(comentarioPeliculaDetails.getNickname());

        return comentarioPeliculaRepository.save(comentarioPelicula);

    }

    @Override

    public void deleteComentarioPelicula(Long id) {

        ComentarioPelicula comentarioPelicula = getComentarioPeliculaById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioPelicula not found with id " + id));

        comentarioPeliculaRepository.delete(comentarioPelicula);

    }

    @Override

    public List<ComentarioPelicula> getAllComentariosByPelicula(Pelicula pelicula) {

        return comentarioPeliculaRepository.findByPelicula(pelicula);

    }

    @Override

    public void deleteComentariosByPelicula(String nombrePelicula) {

        // Fetch the movie by its name

        Pelicula pelicula = peliculaService.getPeliculaByNombre(nombrePelicula)

                .orElseThrow(() -> new RuntimeException("Pelicula not found with name " + nombrePelicula));

        // Fetch all comments associated with the movie

        List<ComentarioPelicula> comentarios = comentarioPeliculaRepository.findByPelicula(pelicula);

        // Delete all fetched comments

        comentarioPeliculaRepository.deleteAll(comentarios);

    }

}

Esta clase utiliza ComentarioPeliculaRepository para interactuar con la base de datos. Proporciona métodos para obtener todos los comentarios de películas, obtener un comentario por su ID, crear, actualizar y eliminar comentarios, y obtener comentarios en base a la película asociada. Además, incluye métodos para eliminar todos los comentarios asociados a una película específica.

El método **getAllComentariosPelicula** recupera una lista de todos los comentarios de películas almacenados en la base de datos. Es útil para mostrar todos los comentarios disponibles a los usuarios y administradores.

El método **getComentarioPeliculaById** recupera un comentario específico de una película utilizando su ID único. Si el comentario no se encuentra, se retorna un Optional vacío.

El método **createComentarioPelicula** permite la creación de un nuevo comentario de película y su almacenamiento en la base de datos.

El método **updateComentarioPelicula** permite la actualización de los detalles de un comentario de película existente. Busca el comentario por su ID, actualiza sus atributos y guarda los cambios en la base de datos.

El método **deleteComentarioPelicula** elimina un comentario de película específico de la base de datos utilizando su ID.

El método **getAllComentariosByPelicula** recupera una lista de comentarios asociados a una película específica.

Cada método está anotado con @Transactional para indicar que debe ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

**Clase Comentario de Contenido de F1**

La clase ComentarioF1 es una subclase de Comentario que representa los comentarios de los diferentes contenidos de F1 de la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/ComentarioF1.java.

@Entity

@Table(name = "ComentarioF1")

public class ComentarioF1 extends Comentario{

    @ManyToOne

    @JoinColumn(name = "F1Content\_id", referencedColumnName = "id", nullable = false)

    private F1Content f1Content;

    @ManyToOne

    @JoinColumn(name = "user\_email", referencedColumnName = "email",  nullable = false)

    private User usuario;

// getters and setters   
}

Esta clase añade varios campos específicos a los comentarios de las carreras de F1. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos. La anotación @OneToMany establece la relación con la tabla de F1Content recogiendo el atributo del id del contenido de F1 y con la tabla de los usuarios cogiendo el atributo del email de usuario.

**Repositorio** **Comentario de Contenido de F1**

El repositorio ComentarioF1Repository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de ComentarioF1Content en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/ComentarioF1Repository.java.

@Repository

public interface ComentarioF1Repository extends JpaRepository<ComentarioF1, Long> {

    List<ComentarioF1> findByF1Content(F1Content f1Content);

}

Los métodos findByf1Content son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten obtener los comentarios de un determinado contenido de F1. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicio Comentario de Contenido de F1**

El servicio Comentariof1ServiceImp implementa la interfaz Comentariof1Service, proporcionando la lógica de negocio para las operaciones relacionadas con los comentarios de los contenidos que sean carreras de F1. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/ComentarioF1ServiceImp.java.

@Service

public class Comentariof1ServiceImp implements ComentarioF1Service{

    @Autowired

    private ComentarioF1Repository comentarioF1Repository;

    @Autowired

    private F1ContentService f1ContentService;

    @Override

    public List<ComentarioF1> getAllComentariosF1() {

        return comentarioF1Repository.findAll();

    }

    @Override

    public Optional<ComentarioF1> getComentarioF1ById(Long id) {

        return comentarioF1Repository.findById(id);

    }

    @Override

    public ComentarioF1 createComentarioF1(ComentarioF1 comentarioF1) {

        return comentarioF1Repository.save(comentarioF1);

    }

    @Override

    public ComentarioF1 updateComentarioF1(Long id, ComentarioF1 comentarioF1Details) {

        ComentarioF1 comentarioF1 = comentarioF1Repository.findById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioF1 not found with id " + id));

        comentarioF1.setTexto(comentarioF1Details.getTexto());

        comentarioF1.setNickname(comentarioF1Details.getNickname());

        return comentarioF1Repository.save(comentarioF1);

    }

    @Override

    public void deleteComentarioF1(Long id) {

        ComentarioF1 comentarioF1 = comentarioF1Repository.findById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioF1 not found with id " + id));

        comentarioF1Repository.delete(comentarioF1);

    }

    @Override

    public List<ComentarioF1> getAllComentariosByF1(F1Content f1Content) {

        return comentarioF1Repository.findByF1Content(f1Content);

    }

    @Override

    public void deleteComentariosByF1Content(String nombreCarreraF1) {

        // Fetch the F1 content by its name

        F1Content f1Content = f1ContentService.getF1ContentByNombreCarrera(nombreCarreraF1)

                .orElseThrow(() -> new RuntimeException("F1Content not found with name " + nombreCarreraF1));

        // Fetch all comments associated with the F1 content

        List<ComentarioF1> comentarios = comentarioF1Repository.findByF1Content(f1Content);

        // Delete all fetched comments

        comentarioF1Repository.deleteAll(comentarios);

    }

}

clase utiliza ComentarioF1Repository para interactuar con la base de datos. Proporciona métodos para obtener todos los comentarios de Fórmula 1, obtener un comentario por su ID, crear, actualizar y eliminar comentarios, y obtener comentarios en base al contenido de Fórmula 1 asociado. Además, incluye un método para eliminar todos los comentarios asociados a un contenido de Fórmula 1 específico.

El método **getAllComentariosF1** recupera una lista de todos los comentarios de Fórmula 1 almacenados en la base de datos. Es útil para mostrar todos los comentarios disponibles a los usuarios y administradores.

El método **getComentarioF1ById** recupera un comentario específico de Fórmula 1 utilizando su ID único. Si el comentario no se encuentra, se retorna un Optional vacío.

El método **createComentarioF1** permite la creación de un nuevo comentario de Fórmula 1 y su almacenamiento en la base de datos.

El método **updateComentarioF1** permite la actualización de los detalles de un comentario de Fórmula 1 existente. Busca el comentario por su ID, actualiza sus atributos y guarda los cambios en la base de datos.

El método **deleteComentarioF1** elimina un comentario de Fórmula 1 específico de la base de datos utilizando su ID.

El método **getAllComentariosByF1** recupera una lista de comentarios asociados a un contenido de Fórmula 1 específico.

Cada método está anotado con @Transactional para indicar que debe ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

**Clase Comentario de Contenido de Fútbol**

La clase ComentarioFootball es una subclase de Comentario que representa los comentarios de los diferentes contenidos de fútbol de la aplicación. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/ComentarioFootball.java.

@Entity

@Table(name = "ComentarioFootball")

public class ComentarioFootball  extends Comentario{

    @ManyToOne

    @JoinColumn(name = "FootballContent\_id", referencedColumnName = "id", nullable = false)

    private FootballContent footballContent;

    @ManyToOne

    @JoinColumn(name = "user\_email", referencedColumnName = "email",  nullable = false)

    private User usuario;

// getters and setters   
}

Esta clase añade varios campos específicos a los comentarios de las partidos de fútbol. La anotación@Entity indica que esta clase se mapeará a una tabla en la base de datos. La anotación @OneToMany establece la relación con la tabla de F1Content recogiendo el atributo del id del contenido de F1 y con la tabla de los usuarios cogiendo el atributo del email de usuario.

**Repositorio Comentario de Contenido de Fútbol**

El repositorio ComentarioFootballRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de ComentarioFootaballContent en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/ComentarioFootballRepository.java.

@Repository

public interface ComentarioFootballRepository extends JpaRepository<ComentarioFootball, Long> {

    List<ComentarioFootball> findByFootballContent(FootballContent footballContent);

}

Los métodos findByFootballContent son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten obtener los comentarios de un determinado contenido de fútbol. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicio Comentario de Contenido de Fútbol**

El servicio ComentarioFootballServiceImp implementa la interfaz ComentarioFotballService, proporcionando la lógica de negocio para las operaciones relacionadas con los comentarios de los contenidos que sean partidos de fútbol. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/ComentarioFootaballServiceImp.java.

@Service

public class ComentarioFootballServiceImp implements ComentarioFootballService{

    @Autowired

    ComentarioFootballRepository comentarioFootballRepository;

    @Autowired

    FootballContentService footballContentService;

    @Override

    public List<ComentarioFootball> getAllComentariosFootball() {

        return comentarioFootballRepository.findAll();

    }

    @Override

    public Optional<ComentarioFootball> getComentarioFootballById(Long id) {

        return comentarioFootballRepository.findById(id);

    }

    @Override

    public ComentarioFootball createComentarioFootballContent(ComentarioFootball comentarioFootball) {

        return comentarioFootballRepository.save(comentarioFootball);

    }

    @Override

    public ComentarioFootball updateComentarioFootballContent(Long id, ComentarioFootball comentarioFootballDetails) {

        ComentarioFootball comentarioFootball = comentarioFootballRepository.findById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioFootabll not found with id " + id));

        comentarioFootball.setTexto(comentarioFootballDetails.getTexto());

        comentarioFootball.setNickname(comentarioFootballDetails.getNickname());

        return comentarioFootballRepository.save(comentarioFootball);

    }

    @Override

    public void deleteComentarioFootball(Long id) {

        ComentarioFootball comentarioFootball = comentarioFootballRepository.findById(id)

                .orElseThrow(() -> new RuntimeException("ComentarioFootball not found with id " + id));

        comentarioFootballRepository.delete(comentarioFootball);

    }

    @Override

    public List<ComentarioFootball> getAllComentariosByFootballContent(FootballContent footballContent) {

        return comentarioFootballRepository.findByFootballContent(footballContent);

    }

    @Override

    public void deleteComentariosByFootballContent(String nombreFootballContent) {

        // Fetch the Football content by its name

        FootballContent footballContent = footballContentService.getFootballContentByNombrePartido(nombreFootballContent)

                .orElseThrow(() -> new RuntimeException("FootballContent not found with name " + nombreFootballContent));

        // Fetch all comments associated with the Football content

        List<ComentarioFootball> comentarios = comentarioFootballRepository.findByFootballContent(footballContent);

        // Delete all fetched comments

        comentarioFootballRepository.deleteAll(comentarios);

    }

}

Esta clase utiliza ComentarioFootballRepository para interactuar con la base de datos. Proporciona métodos para obtener todos los comentarios de fútbol, obtener un comentario por su ID, crear, actualizar y eliminar comentarios, y obtener comentarios en base al contenido de fútbol asociado. Además, incluye un método para eliminar todos los comentarios asociados a un contenido de fútbol específico.

El método **getAllComentariosFootball** recupera una lista de todos los comentarios de fútbol almacenados en la base de datos. Es útil para mostrar todos los comentarios disponibles a los usuarios y administradores.

El método **getComentarioFootballById** recupera un comentario específico de fútbol utilizando su ID único. Si el comentario no se encuentra, se retorna un Optional vacío.

El método **createComentarioFootballContent** permite la creación de un nuevo comentario de fútbol y su almacenamiento en la base de datos.

El método **updateComentarioFootballContent** permite la actualización de los detalles de un comentario de fútbol existente. Busca el comentario por su ID, actualiza sus atributos y guarda los cambios en la base de datos.

El método **deleteComentarioFootball** elimina un comentario de fútbol específico de la base de datos utilizando su ID.

El método **getAllComentariosByFootballContent** recupera una lista de comentarios asociados a un contenido de fútbol específico.

Cada método está anotado con @Transactional para indicar que debe ejecutarse dentro de una transacción de base de datos. Esto significa que, si ocurre un error durante la ejecución del método, todas las operaciones de base de datos realizadas dentro del método se revertirán.

6.3.1.3 Usuario

**Clase Usuario**

La clase User representa un usuario en el sistema. Este tiene diferentes roles en la aplicación desde usuario normal que solo se limita a ver el contenido, luego esta el administrador que tiene lo permisos para editar como quiera el contenido. También el usuario está asociada a una tarjeta de crédito, a los comentarios que hace y a las comunidades de chat que crea.

Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/User.java.

@Entity

@Table(name = "Usuario")

public class User {

    @Id

    @Column(name = "email", nullable = false, length = 50)

    private String email;

    @Column(name = "nombre", length = 50, nullable = false)

    private String nombre;

    @Column(name = "apellidos", length = 80, nullable = false)

    private String apellidos;

    @Column(name = "nickname", length = 50, nullable = false)

    private String nickname;

    @Column(name = "contrasenha", length = 50, nullable = false)

    private String contrasenha;

    @Column(name = "fechaNacimiento", length = 12, nullable = false)

    private LocalDate fechaNacimiento;

    @Column(name = "planSuscripcion", length = 20, nullable = false)

    private String planSuscripcion;

    @Column(name = "pagoValidado", nullable = false)

    private Boolean pagoValidado;

    @Column(name = "url\_image\_perfil", length = 200)

    private String url\_image\_perfil;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<TarjetaCredito> tarjetas;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<ComentarioPelicula> comentarios;

    @OneToMany(mappedBy = "createdBy", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<ChatCommunity> createdCommunities;

    @OneToMany(mappedBy = "sentBy", cascade = CascadeType.ALL, orphanRemoval = true)

    private List<ChatMessage> sentMessages;

    @ManyToMany(fetch = FetchType.EAGER)

    @JoinTable(name = "user\_roles",

            joinColumns = @JoinColumn(name = "user\_id"),

            inverseJoinColumns = @JoinColumn(name = "role\_id"))

    private Set<Role> roles;

// getters and setters   
}

Almacena información como el correo electrónico, nombre, apellidos, nickname, contraseña, fecha de nacimiento, plan de suscripción, estado de validación del pago, URL de la imagen de perfil, tarjetas de crédito asociadas, comentarios de películas realizados, comunidades creadas y mensajes enviados en el chat.

Además, esta clase establece una relación muchos a muchos con la clase Role para gestionar los roles de los usuarios en el sistema.

**Repositorio Usuario**

El repositorio UserRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de User en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/UserRepository.java.

@Repository

public interface UserRepository extends JpaRepository<User, String> {

    boolean existsByEmail(String email);

    boolean existsByNickname(String nickname);

}

Los métodos existsByEmail y existsByNickname son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten tener un control para saber si el nickname y el usuario ya esta registrado en el sistema. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicio User**

El servicio UserServiceImp implementa la interfaz UserFotballService, proporcionando la lógica de negocio para las operaciones relacionadas con los usuarios. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/UserServiceImp.java.

@Service

public class UserServiceImpl implements UserService{

    @Autowired

    private UserRepository userRepository;

    @Autowired

    private RoleService roleService;

    //Metodo para obtener todos los usuarios

    @Override

    public List<User> getAllUsers() {

        return userRepository.findAll();

    }

    //Metodo para obtener un usuario por su email

    @Override

    public Optional<User> getUserByEmail(String email) {

        return userRepository.findById(email);

    }

    //Metodo para guardar un usuario

    @Override

    public User createUser(User user) {

        User createdUser = userRepository.save(user);

        roleService.assignRoleToUser(createdUser.getEmail(), "USER"); // Asignar el rol de "USER" por defecto

        return createdUser;

    }

    //Metodo para actualizar un usuario

    @Override

    public User updateUser(String email, User userDetails) {

        User user = userRepository.findById(email)

                .orElseThrow(() -> new RuntimeException("User not found"));

        user.setNombre(userDetails.getNombre());

        user.setNickname(userDetails.getNickname());

        user.setContrasenha(userDetails.getContrasenha());

        user.setFechaNacimiento(userDetails.getFechaNacimiento());

        user.setPlanSuscripcion(userDetails.getPlanSuscripcion());

        user.setApellidos(userDetails.getApellidos());

        user.setPagoValidado(userDetails.getPagoValidado());

        user.setUrl\_image\_perfil(userDetails.getUrl\_image\_perfil());

        return userRepository.save(user);

    }

    //Metodo para eliminar un usuario

    @Override

    public void deleteUser(String email) {

        userRepository.deleteById(email);

    }

    // Metodo para la autenticacion de un usuario

    @Override

    public boolean authenticateUser(String email, String password) {

        User user = userRepository.findById(email).orElse(null);

        if(user == null){

            return false;

        }

        return user.getContrasenha().equals(password);

    }

    @Override

    public boolean estaRegistrado(String email) {

        return userRepository.existsById(email);

    }

    @Override

    public boolean emailExists(String email) {

        return userRepository.existsByEmail(email);

    }

    @Override

    public boolean nicknameExists(String nickname) {

        return userRepository.existsByNickname(nickname);

    }

}

Esta clase implementa la interfaz UserService y proporciona métodos para interactuar con la entidad User en la base de datos. Utiliza UserRepository para acceder a los datos de usuario y RoleService para gestionar los roles de los usuarios.

El método **getAllUsers** recupera una lista de todos los usuarios almacenados en la base de datos.

El método **getUserByEmail** recupera un usuario específico utilizando su correo electrónico.

El método **createUser** permite la creación de un nuevo usuario y su almacenamiento en la base de datos. Además, asigna automáticamente el rol de "USER" al nuevo usuario.

El método **updateUser** permite la actualización de los detalles de un usuario existente.

El método **deleteUser** elimina un usuario específico de la base de datos.

El método **authenticateUser** verifica las credenciales de un usuario para autenticar su acceso al sistema.

El **métodoestaRegistrado** verifica si un usuario está registrado en el sistema utilizando su correo electrónico.

El método **emailExists** verifica si ya existe un usuario registrado con un correo electrónico específico.

El método **nicknameExists** verifica si ya existe un usuario registrado con un apodo específico.

Se ha omitido la anotación @Transactional en los métodos del servicio porque JPA implementa automáticamente transacciones para los métodos que interactúan con la base de datos.

El método **authenticateUser** compara la contraseña proporcionada con la contraseña almacenada del usuario para autenticar su acceso.

Los métodos **emailExists** y **nicknameExists** se utilizan para verificar la existencia de un usuario utilizando su correo electrónico o apodo respectivamente.

**Clase Role**

La clase Role representa los roles disponibles en el sistema. Estos roles determinan los permisos y privilegios que tienen los usuarios en la aplicación.

Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/entity/Role.java.

@Entity

@Table(name = "Role")

public class Role {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;

    @Column(name = "name", length = 50, nullable = false)

    private String name;

// getters and setters   
}

Los roles en la aplicación son utilizados para controlar el acceso y los privilegios de los usuarios. Cada usuario puede tener uno o varios roles asignados, lo que define su nivel de acceso y las acciones que pueden realizar en la plataforma.

Habrá 3 tipos de roles: User, Admin y SuperAdmin.

El rol de **User** permite ver el contenido y dependiendo del plan de suscripción puede hacer comentarios o no, ver el contenido en directo y participar en las comunidades de usuarios.

El rol de **Admin** tiene los permisos para modificar el contenido, las comunidades, para crear y borrar contenido y comunidades, pero no para asignar roles a otros usuarios.

El rol **SuperAdmin** se encarga de asignar roles a los usuarios de la plataforma asigna quien es administrador, puede a hacer a un usuario administrador.

**Repositorio Roles**

El repositorio RoleRepository es una interfaz que extiende JpaRepository, proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de Roles en la base de datos. Esta interfaz se encuentra en el archivo:

src/main/java/com/example/cursospringboot/repository/RoleRepository.java.

@Repository

public interface RoleRepository extends JpaRepository<Role, Long> {

    Optional<Role> findByName(String name);

}

Los métodos findByName son ejemplos de métodos de consulta derivados, que Spring Data JPA genera automáticamente a partir de su nombre. Estos métodos permiten tener un control de los usuarios en base a los roles que tengan. Los cuales usaremos en la creación de los servicios que a su vez serán usados por el controlador.

**Servicios Roles**

El servicio RoleServiceImp implementa la interfaz RoleService, proporcionando la lógica de negocio para las operaciones relacionadas con los roles. Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/service/RoleServiceImp.java.

@Service

public class RoleServiceImp implements RoleService{

    @Autowired

    RoleRepository roleRepository;

    @Autowired

    UserRepository userRepository;

    @Override

    public void assignRoleToUser(String email, String roleName) {

        Optional<User> userOptional = userRepository.findById(email);

        Optional<Role> roleOptional = roleRepository.findByName(roleName);

        if (userOptional.isPresent() && roleOptional.isPresent()) {

            User user = userOptional.get();

            Role role = roleOptional.get();

            user.getRoles().add(role);

            userRepository.save(user);

        }

    }

    @Override

    public Set<Role> getRolesOfUser(String email) {

        Optional<User> userOptional = userRepository.findById(email);

        if (userOptional.isPresent()) {

            User user = userOptional.get();

            return user.getRoles();

        }

        return null;

    }

    @Override

    public boolean userHasRole(String email, String roleName) {

        Optional<User> userOptional = userRepository.findById(email);

        if (userOptional.isPresent()) {

            User user = userOptional.get();

            return user.getRoles().stream().anyMatch(role -> role.getName().equals(roleName));

        }

        return false;

    }

    @Override

    public Optional<Role> getRoleByName(String name) {

        return roleRepository.findByName(name);

    }

    @Override

    public Role createRole(Role role) {

        return roleRepository.save(role);

    }

    @Override

    public Role updateRole(String name, Role roleDetails) {

        Role role = roleRepository.findByName(name)

                .orElseThrow(() -> new RuntimeException("Role not found"));

        role.setName(roleDetails.getName());

        return roleRepository.save(role);

    }

    @Override

    public void deleteRole(String name) {

        Role role = roleRepository.findByName(name)

                .orElseThrow(() -> new RuntimeException("Role not found"));

        roleRepository.delete(role);

    }

}

El método **assignRoleToUser** asigna un rol específico a un usuario. Primero, busca el usuario y el rol por su identificador y nombre respectivamente. Si ambos existen, añade el rol al conjunto de roles del usuario y guarda los cambios en el repositorio de usuarios.

El método **getRolesOfUser** recupera el conjunto de roles asociados a un usuario. Busca el usuario por su identificador y, si lo encuentra, devuelve su conjunto de roles. Si el usuario no existe, devuelve null.

El método **userHasRole** verifica si un usuario tiene un rol específico. Busca el usuario por su identificador y, si lo encuentra, comprueba si alguno de sus roles coincide con el nombre del rol especificado.

El método **getRoleByName** busca y recupera un rol por su nombre. Devuelve un Optional que puede contener el rol si se encuentra.

El método **createRole** crea un nuevo rol en la base de datos. Guarda el rol proporcionado en el repositorio de roles y lo devuelve.

El método **updateRole** actualiza los detalles de un rol existente. Busca el rol por su nombre, actualiza sus atributos con los detalles proporcionados y guarda los cambios en el repositorio de roles.

El método **deleteRole** elimina un rol específico de la base de datos. Busca el rol por su nombre y, si lo encuentra, lo elimina del repositorio de roles.

6.3.1.4 Comunidades de Usuarios

6.3.2 Controladores

6.3.2.1 Contenidos

6.3.2.2 Comentarios

6.3.2.3 Usuario

6.3.2.4 Comunidades de Usuarios

6.3.2 Vistas

Pruebas

Estrategia de pruebas

Casos de prueba

Resultados de las pruebas

**Uso de la aplicación**

Guía del usuario

Guía de administrador

**Mantenimiento y soporte**

Procedimientos de mantenimiento

Soporte al usuario

**Conclusiones y trabajos futuros**

**Bibliografía**

**Anexos**

Glosario

Referencias

**Licencia**