

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
**ESCUELA POLITÉCNICA
SUPERIOR**

2024

UNIVERSIDAD DE ALCALÁ



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

julio / 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: julio / 2024

ÍNDICE RESUMIDO

1. INTRODUCCIÓN	1
1.1 PROPÓSITO DEL DOCUMENTO	1
1.2 ALCANCE DEL PROYECTO.....	1
1.3 DEFINICIONES, ACRÓNIMOS Y ABREVIATURAS	2
1.4 VISIÓN GENERAL DEL DOCUMENTO	3
2. DESCRIPCIÓN GENERAL DEL PROYECTO	4
2.1 FUNCIONES DEL PRODUCTO.....	4
2.2 OBJETIVO DEL PROYECTO	5
2.3 CARACTERÍSTICAS DEL USUARIO	5
2.4 RESTRICCIONES	6
2.5 DEPENDENCIAS.....	6
3. REQUISITOS ESPECÍFICOS	7
3.1 REQUISITOS DE INTERFACES EXTERNAS	7
3.2 REQUISITOS FUNCIONALES	8
3.3 REQUISITOS DE RENDIMIENTO	8
3.4 ATRIBUTOS DEL SISTEMA DE SOFTWARE	9
4. ESTADO DEL ARTE.....	9
4.1 INTRODUCCIÓN.....	9
4.2 HISTORIA Y EVOLUCIÓN	9
4.3 TECNOLOGÍAS ACTUALES	9
4.4 APLICACIONES Y USOS	10
4.5 VENTAJAS Y DESVENTAJAS	11
4.6 FUTURO DE LA TECNOLGÍA	11
4.7 CONCLUSIÓN FINAL.....	11
5. ARQUITECTURA DEL SISTEMA	11
5.1 DESCRIPCIÓN DE LA ARQUITECTURA.....	11
5.2 DESCRIPCIÓN DE LOS COMPONENTES	13
6. DISEÑO DETALLADO.....	13
6.1 DESCRIPCIÓN DE LOS MÓDULOS.....	14
6.2 DIAGRAMAS DE CLASES	14
7. IMPLEMENTACIÓN.....	21
7.1 ENTORNO DE DESARROLLO	21

7.2	PROCESO DE INSTALACIÓN	22
8.	CÓDIGO FUENTE	28
8.1	Modelos	28
8.2	Controladores y Vista	61
9.	PRUEBAS.....	154
9.1	ESTRATEGIA DE PRUEBAS	154
9.2	CASOS DE PRUEBA	155
9.3	Resultados de las pruebas	161
10.	USO DE LA APLICACIÓN.....	173
10.1	GUÍA DEL USUARIO.....	173
10.2	GUÍA DE ADMINISTRADOR	184
11.	MANTENIMIENTO Y SOPORTE	195
11.1	PROCEDIMIENTOS DE MANTENIMIENTO	195
11.2	SOPORTE AL USUARIO	195
12.	PRESUPUESTO DEL DESARROLLO DE LA APLICACIÓN	196
12.1	COSTE DEL DESARROLLO	196
12.2	HERRAMIENTAS Y SOFTWARE.....	196
12.3	INFRAESTRUCTURA	196
12.3	COSTES INDIRECTOS.....	197
12.4	RESUMEN DE COSTOS	197
12.5	CONSIDERACIONES ADICIONALES	197
13.	CONCLUSIONES Y TRABAJOS FUTUROS.....	198
14.	BIBLIOGRAFÍA.....	199

ÍNDICE DETALLADO

1. INTRODUCCIÓN	1
1.1 PROPÓSITO DEL DOCUMENTO	1
1.2 ALCANCE DEL PROYECTO.....	1
1.3 DEFINICIONES, ACRÓNIMOS Y ABREVIATURAS	2
1.4 VISIÓN GENERAL DEL DOCUMENTO	3
2. DESCRIPCIÓN GENERAL DEL PROYECTO	4
2.1 FUNCIONES DEL PRODUCTO.....	4
2.2 OBJETIVO DEL PROYECTO	5
2.3 CARACTERÍSTICAS DEL USUARIO	5
2.4 RESTRICCIONES	6
2.5 DEPENDENCIAS	6
3. REQUISITOS ESPECÍFICOS	7
3.1 REQUISITOS DE INTERFACES EXTERNAS	7
3.2 REQUISITOS FUNCIONALES	8
3.3 REQUISITOS DE RENDIMIENTO	8
3.4 ATRIBUTOS DEL SISTEMA DE SOFTWARE	9
4. ESTADO DEL ARTE.....	9
4.1 INTRODUCCIÓN.....	9
4.2 HISTORIA Y EVOLUCIÓN	9
4.3 TECNOLOGÍAS ACTUALES	9
4.4 APLICACIONES Y USOS	10
4.5 VENTAJAS Y DESVENTAJAS	11
4.6 FUTURO DE LA TECNOLGÍA	11
4.7 CONCLUSIÓN FINAL.....	11
5. ARQUITECTURA DEL SISTEMA	11
5.1 DESCRIPCIÓN DE LA ARQUITECTURA.....	11
5.2 DESCRIPCIÓN DE LOS COMPONENTES	13
6. DISEÑO DETALLADO.....	13
6.1 DESCRIPCIÓN DE LOS MÓDULOS.....	14
6.2 DIAGRAMAS DE CLASES	14
6.2.1 <i>Entidades</i>	15
6.2.2 <i>Repositorios</i>	17
6.2.3 <i>Servicios</i>	19

6.2.4	<i>Controlador</i>	20
7.	IMPLEMENTACIÓN	21
7.1	ENTORNO DE DESARROLLO	21
7.1.2	<i>¿Por qué he decidido usar Spring boot y no otras tecnologías?</i>	22
7.2	PROCESO DE INSTALACIÓN	22
7.2.1	<i>Configuración Básica de IntelliJ</i>	22
7.2.2	<i>Configuración de las dependencias</i>	23
7.2.3	<i>Estructurar el proyecto</i>	24
7.2.4	<i>Descargar y correr el proyecto de GitHub</i>	27
8.	CÓDIGO FUENTE	28
8.1	Modelos	28
8.1.1	<i>Contenidos</i>	28
8.1.1.1	<i>Clase Contenido</i>	28
8.1.1.2	<i>Contenido Película</i>	29
<i>Clase Película</i>	29	
<i>Repositorio Película</i>	29	
<i>Servicio Película</i>	30	
8.1.1.3	<i>Contenido F1</i>	31
<i>Clase F1Content</i>	31	
<i>Repositorio F1Content</i>	31	
<i>Servicio F1Content</i>	32	
8.1.1.4	<i>Contenido Fútbol</i>	34
<i>Clase FootballContent</i>	34	
<i>Repositorio FootballContent</i>	34	
<i>Servicios FootballContent</i>	35	
8.1.1.5	<i>Contenido en Directo</i>	37
<i>Clase LiveContent</i>	37	
<i>Repositorio LiveContent</i>	37	
<i>Servicio LiveContent</i>	38	
8.1.2	<i>Comentarios</i>	40
8.1.2.1	<i>Clase Comentario</i>	40
8.1.2.2	<i>Contenido Comentario Película</i>	41
<i>Clase Comentario de Películas</i>	41	
<i>Repositorio Comentario de Película</i>	42	
<i>Servicio Comentario de Película</i>	42	
8.1.2.3	<i>Contenido Comentario F1</i>	44
<i>Clase Comentario de Contenido de F1</i>	44	

<i> Repository Comentario de Contenido de F1</i>	44
<i> Servicio Comentario de Contenido de F1</i>	45
8.1.2.4 Contenido Comentario Fútbol	46
<i> Clase Comentario de Contenido de Fútbol.....</i>	46
<i> Repository Comentario de Contenido de Fútbol</i>	47
<i> Servicio Comentario de Contenido de Fútbol.....</i>	47
8.1.3 <i> Usuario</i>	49
8.1.3.1 Contenido Usuario.....	49
<i> Clase Usuario.....</i>	49
<i> Repository Usuario</i>	50
<i> Servicio User</i>	51
8.1.3.2 Contenido Usuario Rol	53
<i> Clase Role.....</i>	53
<i> Repository Roles.....</i>	53
<i> Servicios Roles.....</i>	54
8.1.4 <i> Comunidades de Usuarios</i>	56
8.1.4.1 Contenido Comunidades de Usuarios.....	56
<i> Clase ChatCommunity.....</i>	56
<i> Repository ChatCommunity.....</i>	57
<i> Servicio ChatCommunity.....</i>	57
<i> Clase ChatMessage</i>	59
<i> Repository ChatMessage</i>	59
<i> Servicio ChatMessage</i>	60
8.2 Controladores y Vista	61
8.2.1 <i> Controladores y Vista Contenidos</i>	63
Controlador/Vista Película.....	63
Controlador/Vista Contenido de F1	77
Controlador/Vista Contenido de Fútbol	86
Controlador/Vista Contenido en directo	96
8.2.2 <i> Controladores/Vista Comentarios</i>	108
Controlador/Vista Comentarios Películas	108
Controlador/Vista Comentarios F1	111
Controlador/Vista Comentarios Fútbol	114
8.2.3 <i> Controladores/Vista Usuario</i>	117
Controlador/Vista Inicio de Sesión del Usuario.....	117
Controlador/Vista Tarjeta de crédito.....	128
Controlador/Vista Perfil de Usuario	131

8.2.4 <i>Controladores/Vista Comunidades de Usuarios</i>	142
Controlador/Vista Inicio Comunidades/Chats Usuarios	143
9. PRUEBAS.....	154
9.1 ESTRATEGIA DE PRUEBAS	154
9.2 CASOS DE PRUEBA	155
9.2.1 <i>Casos de prueba Usuarios</i>	155
9.2.2 <i>Casos de Pruebas Películas</i>	157
9.2.3 <i>Casos de Prueba Comentarios</i>	159
9.3 Resultados de las pruebas	161
9.3.1 <i>Entidad Usuario</i>	161
9.3.2 <i>Entidad Película</i>	165
9.3.2 <i>Entidad Comentario</i>	169
10. USO DE LA APLICACIÓN.....	173
10.1 GUÍA DEL USUARIO	173
10.2 GUÍA DE ADMINISTRADOR	184
11. MANTENIMIENTO Y SOPORTE	195
11.1 PROCEDIMIENTOS DE MANTENIMIENTO	195
11.2 SOPORTE AL USUARIO	195
12. PRESUPUESTO DEL DESARROLLO DE LA APLICACIÓN	196
12.1 COSTE DEL DESARROLLO	196
12.2 HERRAMIENTAS Y SOFTWARE.....	196
12.3 INFRAESTRUCTURA	196
12.3 COSTES INDIRECTOS.....	197
12.4 RESUMEN DE COSTOS	197
12.5 CONSIDERACIONES ADICIONALES	197
13. CONCLUSIONES Y TRABAJOS FUTUROS.....	198
14. BIBLIOGRAFÍA.....	199

ÍNDICE DE FIGURAS

FIGURA 1: FUNCIONAMIENTO DEL PATRÓN MODELO-VISTA-CONTROLADOR EN SPRING BOOT	13
FIGURA 2: ESTRUCTURA DE LOS DIFERENTES PAQUETES PARA LOS DIAGRAMAS DE CLASE.....	14
FIGURA 3: DIAGRAMA DE CLASES ENTIDADES	15
FIGURA 4: DIAGRAMA DE CLASES REPOSITORIOS	17
FIGURA 5: DIAGRAMA DE CLASES SERVICIOS.....	19
FIGURA 6: DIAGRAMA DE CLASES CONTROLADOR.....	20
FIGURA 7: RUTA DE CREACIÓN DEL PROYECTO EN SPRING BOOT.....	22
FIGURA 8: COMO AGREGAR LAS DEPENDENCIAS EN SPRING BOOT.....	23
FIGURA 9: ORGANIZACIÓN DE PAQUETES DENTRO DE SPRING BOOT.....	24
FIGURA 10: ORGANIZACIÓN CÓDIGO FRONT-END SPRING BOOT	24
FIGURA 11 ORGANIZACION DE CLASES EN EL CONTROLADOR	25
FIGURA 12: ORGANIZACIÓN DE CLASES DENTRO DEL SERVICIO.....	25
FIGURA 13: ORGANIZACIÓN DE CLASES EN EL REPOSITORIO.....	26
FIGURA 14: ORGANIZACIÓN DE CLASES EN LAS ENTIDADES	26
FIGURA 15: PANEL QUE IMPLEMENTA EL MÉTODO GETALLPELICULAS	64
FIGURA 16: PANEL QUE IMPLEMENTA EL METODO GETPELICULA	66
FIGURA 17: PANEL QUE IMPLEMENTA EL MÉTODO AÑADIRPELICULA	67
FIGURA 18: PANEL QUE IMPLEMENTA EL MÉTODO CREATEPELICULA	69
FIGURA 19: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEPELICULA	71
FIGURA 20: PANEL QUE IMPLEMENTA EL MÉTODO DELETEPELICULA.....	74
FIGURA 21: PANEL QUE IMPLEMENTA EL MÉTODO REALTIMESEARCH PARA PELÍCULAS.....	76
FIGURA 22: PANEL QUE IMPLEMENTA EL MÉTODO GETALLF1CONTENT	79
FIGURA 23: PANEL QUE IMPLEMENTA EL MÉTODO GETNOMBRECARRERAF1	79
FIGURA 24: PANEL PARA IMPLEMENTAR EL MÉTODO AÑADIRCARRERA	80
FIGURA 25: PANEL PARA IMPLEMENTAR EL MÉTODO CREATEF1CONTENT	81
FIGURA 26: PANEL PARA IMPLEMENTAR EL MÉTODO DELETECARRERAF1.....	82
FIGURA 27: PANEL PARA IMPLEMENTAR EL MÉTODO UPDATEF1CONTENT	84
FIGURA 28: PANEL PARA IMPLEMENTAR EL MÉTODO REALTIMESEARCH	84
FIGURA 29: PANEL PARA IMPLEMENTAR EL MÉTODO GETALLFOOTBALLCONTENT	87
FIGURA 30: PANEL PARA IMPLEMENTAR EL METODO GETNOMBREFOOTBALLCONTENT.....	88
FIGURA 31: PANEL PARA IMPLEMENTAR EL MÉTODO AGREGARPARTIDO.....	89
FIGURA 32: PANEL PARA IMPLEMENTAR EL MÉTODO CREATEFOOTBALLCONTENT.....	91
FIGURA 33: PANEL PARA IMPLEMENTAR EL MÉTODO UPDATEFOOTBALLCONTENT	93
FIGURA 34: PANEL PARA IMPLEMENTAR EL MÉTODO DELETEFOOTBALLCONTENT	94
FIGURA 35: PANEL PARA IMPLEMENTAR EL MÉTODO REALTIMESEARCH DE F1	95
FIGURA 36: PANEL PARA IMPLEMENTAR EL MÉTODO GETLIVECONTENT	99
FIGURA 37: PANEL PARA IMPLEMENTAR EL MÉTODO GETLIVECONTENT CON CONTENIDO EN VIVO	101
FIGURA 38: PANEL PARA IMPLEMENTAR EL MÉTODO AGREGARLIVECONTENT	102
FIGURA 39: PANEL PARA IMPLEMENTAR EL METODO CREATELIVECONTENT	104
FIGURA 40: PANEL PARA IMPLEMENTAR EL MÉTODO UPDATERLIVECONTENT	107
FIGURA 41: PANEL PARA IMPLEMENTAR EL MÉTODO DELETELIVECONTENT	108
FIGURA 42: PANEL QUE IMPLEMENTA EL MÉTODO CREATECOMENTARIOPELICULA.....	111
FIGURA 43: PANEL QUE IMPLEMENTA EL MÉTODO CREATECOMENTARIOF1	114
FIGURA 44: PANEL PARA IMPLEMENTAR EL MÉTODO CREATECOMENTARIOF1	117
FIGURA 45 PANEL QUE IMPLEMENTA EL MÉTODO ROOT	119
FIGURA 46: PANEL QUE IMPLEMENTA EL MÉTODO PLANSUSCRIPCION	119

FIGURA 47: PANEL QUE IMPLEMENTA EL MÉTODO LOGOUT	120
FIGURA 48: PANEL QUE IMPLEMENTA EL MÉTODO CREATEUSER	122
FIGURA 49: PANEL QUE IMPLEMENTA EL MÉTODO AUTHENTICATEUSER	124
FIGURA 50: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEPLAN	127
FIGURA 51: PANEL QUE IMPLEMENTA EL MÉTODO PROCESARPAGO	131
FIGURA 52: PANEL QUE IMPLEMENTA ERRORES REDIRECTATTRIBUTES	133
FIGURA 53: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEINFO	135
FIGURA 54: PANEL 2 QUE IMPLEMENTA ERRORES REDIRECTATTRIBUTES	136
FIGURA 55: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEPASSWORD	137
FIGURA 56: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEPLAN	139
FIGURA 57: PANEL QUE IMPLEMENTA EL MÉTODO UPDATEPLAN USERPROFILE	140
FIGURA 58: PANEL QUE IMPLEMENTA EL MÉTODO SELECTPROFILEPICTURE	140
FIGURA 59: PANEL DE SELECCIÓN DE IMÁGENES DE PERFIL	142
FIGURA 60: PANEL DE COMUNIDADES DE CHATS	142
FIGURA 61: PANEL DE SELECCIÓN DE OPCIONES DE LA PLATAFORMA	143
FIGURA 62: PANEL QUE IMPLEMENTA EL MÉTODO SHOWCHATCOMMUNITY PARA PELÍCULAS	147
FIGURA 63: FIGURA QUE IMPLEMENTA EL MÉTODO SHOWCHATCOMMUNITY, PARA FÚTBOL	147
FIGURA 64: PANEL QUE IMPLEMENTA EL MÉTODO CREATECOMUNIDAD	149
FIGURA 65: PANEL QUE IMPLEMENTA EL MÉTODO UPDATECOMUNIDAD	152
FIGURA 66: PANEL QUE IMPLEMENTA EL MÉTODO DELETECOMUNIDAD	154
FIGURA 67: IMAGEN DE FUNCIONAMIENTO DE API REST	154
FIGURA 68: IMAGEN 2, FUNCIONAMIENTO DE API REST	155
FIGURA 69: OBTENER TODOS LOS USUARIOS CON POSTMAN	162
FIGURA 70: OBTENER UN USUARIO POR MAIL POSTMAN	163
FIGURA 71: CREAR UN NUEVO USUARIO POSTMAN	164
FIGURA 72: ACTUALIZAR UN USUARIO EXISTENTE POSTMAN	164
FIGURA 73: ELIMINAR UN USUARIO POSTMAN	165
FIGURA 74: OBTENER TODAS LAS PELÍCULAS CON POSTMAN	166
FIGURA 75: OBTENER UNA PELÍCULA POR NOMBRE POSTMAN	166
FIGURA 76: CREAR UNA NUEVA PELÍCULA POSTMAN	167
FIGURA 77: ACTUALIZAR UNA PELÍCULA EXISTENTE POSTMAN	168
FIGURA 78: ELIMINAR UNA PELÍCULA POSTMAN	168
FIGURA 79: OBTENER TODOS LOS COMENTARIOS POSTMAN	169
FIGURA 80: OBTENER UN COMENTARIO POR ID POSTMAN	170
FIGURA 81: ELIMINAR UN COMENTARIO POSTMAN	171
FIGURA 82: BASE DE DATOS MySQL, TRAS INYECCIÓN POSTMAN DE USUARIOS	172
FIGURA 83: BASE DE DATOS MySQL, TRAS INYECCIÓN POSTMAN DE PELICULAS	172
FIGURA 84: PANEL CON LOS DIFERENTES CONTENIDOS DE LA APLICACIÓN	173
FIGURA 85: PANEL DE INICIO DE SESIÓN DE UN USUARIO QUE NO ESTÁ REGISTRADO	174
FIGURA 86: PANEL DE INICIO DE SESIÓN, CON LOS POSIBLES ERRORES	174
FIGURA 87: PANEL DE REGISTRO CON ERRORES EN TODOS LOS CAMPOS	175
FIGURA 88: PANEL DE SELECCIÓN DE LOS PLANES DE SUSCRIPCIÓN	175
FIGURA 89: PANEL DE PASARELA DE PAGO CON TARJETA, MASTERCARD	176
FIGURA 90: PANEL DE PASARELA DE PAGO CON TARJETA, VISA	176
FIGURA 91: PANEL DE PELÍCULAS EN LA APLICACIÓN	177
FIGURA 92: PANEL DE BUSCADOR DE PELÍCULAS EN LA APLICACIÓN	177
FIGURA 93: PANEL DE UNA PELÍCULA PARA SER VISUALIZADA, EN LA APLICACIÓN	178
FIGURA 94: PANEL DE CARRERAS DE F1 EN LA APLICACIÓN	178
FIGURA 95: PANEL DE BUSCADOR DE CARRERAS DE F1, EN LA APLICACIÓN	179
FIGURA 96: PANEL DE UNA CARRERA PARA SER VISUALIZADA, EN LA APLICACIÓN	179

FIGURA 97: PANEL DE PARTIDOS DE FÚTBOL EN LA APLICACIÓN.....	180
FIGURA 98: PANEL DEL BUSCADOR DE PARTIDOS DE FÚTBOL EN LA APLICACIÓN.....	180
FIGURA 99: PANEL PARA LA VISUALIZACIÓN DE UN PARTIDO DE FÚTBOL EN LA APLICACIÓN.	181
FIGURA 100: ERROR DE ACCESO AL CONTENIDO EN DIRECTO, EN LA APLICACIÓN.	181
FIGURA 101: PANEL DE CONTENIDO EN DIRECTO, SIN UN CONTENIDO EN DIRECTO.	182
FIGURA 102: PANEL DE ACTUALIZACIÓN DE UN CONTENIDO EN DIRECTO EN LA APLICACIÓN.	182
FIGURA 103: PANEL DE LAS COMUNIDADES DE CHATS, EN LA APLICACIÓN.....	183
FIGURA 104: PANEL DE ACTUALIZACIÓN DE DATOS DEL USUARIO EN LA APLICACIÓN.	183
FIGURA 105: MENSAJE DE ACTUALIZACIÓN DE DATOS EXITOSOS EN EL PERFIL DEL USUARIO.....	184
FIGURA 106: MENSAJE DE ACTUALIZACIÓN DE CONTRASEÑA CON ÉXITO EN LA APLICACIÓN.	184
FIGURA 107: PANEL DE AGREGAR PELÍCULAS, ADMINISTRADOR.....	185
FIGURA 108: PANEL PARA CREAR UNA NUEVA PELÍCULA, ADMINISTRADOR.	185
FIGURA 109: PANEL PARA ACTUALIZAR PELÍCULAS, ADMINISTRADOR.....	186
FIGURA 110: PANEL PARA BORRAR PELÍCULAS, ADMINISTRADOR.....	186
FIGURA 111: PANEL PARA AÑADIR CARRERAS, ADMINISTRADOR.....	187
FIGURA 112: PANEL PARA AÑADIR CARRERAS DE F1, ADMINISTRADOR.	187
FIGURA 113: PANEL PARA ACTUALIZAR CARRERAS Y BORRARLAS, ADMINISTRADOR.....	188
FIGURA 114: PANEL PARA AÑADIR PARTIDOS DE FÚTBOL, ADMINISTRADOR.	188
FIGURA 115: PANEL PARA AGREGAR CONTENIDO MULTIMEDIA, ADMINISTRADOR.....	189
FIGURA 116: PANEL PARA ACTUALIZAR Y BORRAR PARTIDOS DE FÚTBOL, ADMINISTRADOR.	189
FIGURA 117: PANEL PARA AGREGAR CONTENIDO MULTIMEDIA, ADMINISTRADOR.....	190
FIGURA 118: PANEL PARA AGREGAR CAMPOS DE CONTENIDO EN DIRECTO, ADMINISTRADOR.....	190
FIGURA 119: PANEL PARA ACTUALIZAR Y BORRAR UN CONTENIDO EN DIRECTO, ADMINISTRADOR.	191
FIGURA 120: MENSAJE DE ERROR DE TRAMO HORARIO, ADMINISTRADOR.....	191
FIGURA 121: MENSAJE DE CONTENIDO EN DIRECTO ACTUALIZADO CON ÉXITO, ADMINISTRADOR.	192
FIGURA 122: PANEL PARA CREAR UNA COMUNIDAD, ADMINISTRADOR.....	192
FIGURA 123: PANEL PARA ACTUALIZAR LOS DATOS DE UNA COMUNIDAD, ADMINISTRADOR.	193
FIGURA 124: PANEL DE UNA COMUNIDAD CREADA DE EJEMPLO.	193
FIGURA 125: PANEL PARA BORRAR UNA COMUNIDAD, ADMINISTRADOR.	194
FIGURA 126: MENSAJE DE ÉXITO AL CREAR UNA COMUNIDAD, ADMINISTRADOR.	194
FIGURA 127: MENSAJE DE ERROR AL BORRAR UNA COMUNIDAD 1, ADMINISTRADOR.	194
FIGURA 128: MENSAJE DE ERROR AL BORRAR UNA COMUNIDAD 2, ADMINISTRADOR.	194
FIGURA 129: MENSAJE DE CREACIÓN DE COMUNIDAD CON ÉXITO, ADMINISTRADOR.	194

1. INTRODUCCIÓN

La aplicación web desarrollada en este proyecto se basa en el marco de trabajo Spring Boot [2] y se centra en proporcionar una plataforma de visualización de contenido multimedia interactiva para los usuarios. La aplicación proporciona un sistema de registro de usuarios con fases para la selección del plan de suscripción y del pago del mismo. Los usuarios tendrán diferentes roles lo que definirá las funcionalidades dentro de la aplicación, así como el plan de suscripción de cada usuario. Una vez registrado y completado los pagos se podrá acceder a la aplicación donde estarán los diferentes contenidos multimedia que el usuario podrá ver y comentar, otra de las funcionalidades de la aplicación es la de los contenidos en directo donde los usuarios podrán ver contenido exclusivo o de estreno. También la aplicación tiene comunidades de chats, un espacio público donde los usuarios pueden hablar entre ellos sobre los diferentes contenidos de la aplicación. Por último, cada usuario tendrá un espacio reservado con su información personal, y la posibilidad de actualizar sus datos, medios de pago, imagen de perfil o plan de suscripción. Las dependencias del proyecto se gestionan con Maven [18].

1.1 PROPÓSITO DEL DOCUMENTO

El propósito de este documento es proporcionar una descripción detallada y completa del proyecto que ayude a entender el funcionamiento y la metodología de spring boot [2] tanto a nivel de usuario como a nivel más avanzado, 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.

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.

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 [2] y Java [16] en el backend, y utiliza una base de datos MySQL [13] [7] 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 [1] [15] [11]) y Thymeleaf [2] [6] para crear una interfaz de usuario dinámica y atractiva. Se utiliza AJAX [17] 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 [18] en el backend y npm en el frontend. Esto incluye bibliotecas y marcos como Spring Boot [2] para el desarrollo del servidor, Thymeleaf [2] [6] para la plantilla del lado del servidor, y Bootstrap [8] 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 [4]), 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 [16] que simplifica la configuración y el despliegue de aplicaciones Spring [2] [5].
- **Java:** Lenguaje de programación de alto nivel utilizado para desarrollar la aplicación [16].
- **MySQL:** Sistema de gestión de bases de datos relacional utilizado para almacenar la información de la aplicación [13] [7].
- **Maven:** Herramienta de gestión de proyectos y comprensión de software. Se utiliza para manejar las dependencias del proyecto [18].
- **HTML:** Lenguaje de marcado utilizado para estructurar el contenido en la web [1] [11] [15].
- **CSS:** Lenguaje de hojas de estilo utilizado para describir la apariencia de los documentos HTML [1] [11] [15].
- **JavaScript:** Lenguaje de programación interpretado que se utiliza para crear contenido dinámico en la web [1].
- **Thymeleaf:** Motor de plantillas Java [16] para procesar vistas HTML [1] en aplicaciones web [2] [6].
- **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 [17].
- **Git:** Sistema de control de versiones distribuido utilizado para rastrear cambios en el código fuente durante el desarrollo de software [4].
- **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 [5].
- **HTTP:** Protocolo de transferencia de hipertexto. Es el protocolo de comunicación que permite las transferencias de información en la web [6] [7].
- **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.
- **CRUD:** Create (Crear), Read (Leer), Update (Actualizar) y Delete (Borrar). Este concepto se utiliza para describir las cuatro operaciones básicas que pueden realizarse en la mayoría de las bases de datos y sistemas de gestión de información [6].

- **API REST:** Utilizan el protocolo HTTP [6] [7] para enviar y recibir datos. Las API web, por otro lado, se basan en múltiples protocolos de comunicación como SOAP, XML-RPC y JSON-RPC. Son las más utilizadas para la integración de datos, lo que facilita la transferencia eficiente de datos.
- **NoSQL:** Bases de datos no relacionales. Las bases de datos NoSQL usan un formato no tabular para almacenar datos en lugar de tablas relacionales basadas en reglas, como las bases de datos relacionales.
- **EndPoints:** Son la ubicación de la API en la que un sistema interactúa con una API web. También es el punto de comunicación entre dos sistemas. Es la URL específica que se utiliza para acceder a un recurso proporcionado por una aplicación web desde una API
- **Boilerplate:** código que se repite en varios bloques a lo largo del proyecto o inclusive en diferentes proyectos.

1.4 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, desde la gestión de usuarios, la visualización y los comentarios de los contenidos multimedia, la interacción en tiempo real a través de comunidades de chat, entre otras funcionalidades.
- 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.
- La sección "**Diseño detallado**" proporciona detalles sobre el diseño de la aplicación, incluyendo diagramas de clases, tanto de las entidades, como de los repositorios, servicios y controladores.
- La sección "**Implementación**" describe el entorno de desarrollo, las dependencias y el proceso de instalación. También se incluye el código fuente de la aplicación, que es la parte más extensa de esta documentación ya que se detalla y se explica viendo el código y como se implementa en la vista, cada clase y cada funcionalidad 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, mediante pruebas con Postman [7].
- 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.

2. DESCRIPCIÓN GENERAL DEL PROYECTO

El proyecto es una aplicación web interactiva desarrollada en Java [16] con el marco de trabajo Spring Boot [2]. 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 [2] y Java [16], y una base de datos MySQL [13] [7] para almacenar la información del usuario, los diferentes contenidos multimedia, 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 [1] [15] [11]) y Thymeleaf [2] [6] para crear una interfaz de usuario dinámica y atractiva. AJAX [17] 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 [18] en el backend. Esto incluye bibliotecas y marcos como Spring Boot [2] para el desarrollo del servidor, Thymeleaf [2] [6] para la plantilla del lado del servidor, Jpa [2] [9] para el manejo de los datos, http sesión [6] [7] para el control de las sesiones de los usuarios dentro de la aplicación y Bootstrap [8] para el diseño y los componentes de la interfaz de usuario entre otros. 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 [4]), 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 se 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 más avanzado el cual se irá 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, hay 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:** Dentro de la aplicación habrá usuarios con un rol más avanzado, que permitirá tener funcionalidades exclusivas dentro del aplicación, estas funcionalidades se contratarán en el manejo de contenido multimedia, para realizar operaciones CRUD, estos paneles de administrador estarán implementados dentro de la propia plataforma y se tendrá control gracias a thymeleaf [2] [6] con los roles del usuario.

2.2 OBJETIVO DEL PROYECTO

El objetivo del proyecto es desarrollar una aplicación web completa e interactiva que permita a los usuarios registrarse, visualizar y comentar contenido multimedia, y participar en comunidades de chat en tiempo real, generar diferentes roles para los usuarios con diferentes funcionalidades. Algo que todos hacemos diariamente en diferentes plataformas como los chats en WhatsApp, el contenido en directo en Youtube, la visualización de contenido en Netflix, Hbo, etc y que puede parecer muy complejo o no entender como se hace o en que se basan estas funcionalidades. Y con este proyecto poner al alcance sobre todo a nivel educativo funciones tan comunes en el día a día. Y que la comunidad del mundo software pueda tener en este documento y en este proyecto una guía y una forma de aprender algo nuevo si lo desea. Ya que la plataforma como objetivo ser una herramienta de aprendizaje para aquellos interesados en el desarrollo de aplicaciones web con Spring Boot [2] y Java [16]. 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.

2.3 CARACTERÍSTICAS DEL USUARIO

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

- **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.
- **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.
- **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.
- **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 código 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:

- **Sistema Operativo:** La aplicación se ha hecho utilizando Windows como su sistema operativo. Aunque con IOs, con spring boot [2] instalado y las dependencias instaladas correctamente también funciona.
- **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. Lo único que se puede hacer sin estar registrado es ver la lista de los contenidos que hay en la plataforma, pero en el momento que se intenten ver ya se necesita de registro, también es importante tener el pago validado y el plan seleccionado correctamente.
- **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.
- **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. El rol super admin será exclusivamente para el manejo de roles para hacer que un usuario sea administrador y viceversa.
- **Gestión de Dependencias:** Las dependencias del proyecto se manejan a través de Maven [18] en el backend y npm en el frontend. Cuando se trata del uso de un framework las dependencias adquieren una importancia mayor, y es fundamental si se quiere usar este proyecto tener todas las dependencias instaladas correctamente para poder usarlo y ponerlo en funcionamiento, no obstante, esta documentación recoge una sección de instalación y habilitación del proyecto.
- **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 [4]), 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:

- **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 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 [2] [9].

- **MySQL Driver:** Esta es la dependencia que permite a la aplicación interactuar con la base de datos MySQL [13] [7]. Es esencial para todas las operaciones de la base de datos, como la creación, lectura, actualización y eliminación de datos [7].
- **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 [2].
- **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 [2] [3].
- **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 [2].
- **Thymeleaf:** Esta dependencia es un motor de plantillas Java [16] para el desarrollo web en el lado del servidor. Facilita la creación de vistas dinámicas utilizando HTML [1] estándar. Esto es esencial para la creación de la interfaz de usuario de la aplicación [2] [6].

Estas dependencias se reflejan en el archivo `pom.xml`, que es el archivo de configuración de Maven [18] que define todas las dependencias del proyecto.

3. REQUISITOS ESPECÍFICOS

3.1 REQUISITOS DE INTERFACES EXTERNAS

- **Interfaz de usuario:** La aplicación proporciona una interfaz de usuario, que permite a los usuarios registrarse, iniciar sesión, visualizar y comentar contenido multimedia, y participar en comunidades de chat. La interfaz de usuario es responsive y compatible con varios navegadores web, incluyendo Edge, Opera GX, Firefox y Chrome. La interfaz de usuario se ha desarrollado utilizando (HTML, CSS, JavaScript [1] [15] [11]) y Thymeleaf [2] [6]. AJAX [17] 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 la aplicación.

- **Interfaz de software:** La aplicación se comunica con una base de datos MySQL [13] [7] para almacenar y recuperar datos. La aplicación utiliza el marco de trabajo Spring Boot [2] 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 [18].

Las clases en src/main/java/com/example/cursospringboot/controller/ definen los controladores que manejan las solicitudes HTTP [6] [7] de los usuarios.

- **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

- **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 hay 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.
- **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.
- **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.
- **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.
- **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 administradores crear, actualizar y borrar, mientras que a los usuarios les permite unirse y participar en comunidades de chat.
- **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.
- **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 [4]), 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.

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. Dentro de la aplicación se incorpora en muchas funcionalidades doble capa de seguridad y es que se hacen las comprobaciones tanto en el controlador como en la vista.
- **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 esta documentación que incluye aparte una sección de 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.

4. ESTADO DEL ARTE

4.1 INTRODUCCIÓN

El estado del arte para este proyecto, al ser sobre el desarrollo de una plataforma web se centra en la revisión y análisis de las tecnologías y prácticas utilizadas en el desarrollo de la aplicación, así como en la identificación de trabajos y proyectos similares en el campo.

El proyecto utiliza una combinación de lenguajes de programación (Java y JavaScript), un marco de trabajo basado en Java [16] (Spring Boot), un gestor de dependencias (Maven [18]), un sistema de control de versiones (Git [4]), y una base de datos (MySQL [13] [7]).

4.2 HISTORIA Y EVOLUCIÓN

Java [16], lanzado en 1995, ha sido uno de los lenguajes de programación más populares durante más de dos décadas. JavaScript, lanzado en 1995, ha evolucionado desde un lenguaje de scripting del lado del cliente a un lenguaje de programación completo que se utiliza tanto en el lado del cliente como en el servidor. Spring Boot [2], lanzado en 2014, ha ganado popularidad rápidamente debido a su enfoque en la simplicidad y la productividad. Maven [18], lanzado en 2004, se ha convertido en uno de los gestores de dependencias más utilizados en el desarrollo de Java [16]. Git [4], lanzado en 2005, es ahora el sistema de control de versiones más popular. MySQL [13] [7], lanzado en 1995, es una de las bases de datos más populares del mundo.

4.3 TECNOLOGÍAS ACTUALES

Las tecnologías actuales en el desarrollo de aplicaciones web incluyen lenguajes de programación como Java [16] y JavaScript, marcos de trabajo como Spring Boot [2], gestores de dependencias como Maven [18], sistemas de control de versiones como Git [4], y bases de datos como MySQL [13] [7]. Estas tecnologías son ampliamente utilizadas y tienen una amplia comunidad de desarrolladores y una gran cantidad de recursos de aprendizaje disponibles.

Java es un lenguaje de programación de alto nivel, orientado a objetos y de propósito general, diseñado para tener pocas dependencias de implementación. Es uno de los lenguajes de programación más populares debido a su simplicidad y legibilidad. En este proyecto, Java [16] se utiliza para la lógica del servidor, específicamente para definir las entidades, servicios y controladores.

JavaScript es un lenguaje de programación interpretado que se utiliza principalmente en el desarrollo web del lado del cliente. Permite agregar interactividad a las páginas web y crear experiencias de usuario ricas y dinámicas. En este proyecto, aunque no se muestra en los fragmentos de código proporcionados, es probable que JavaScript se utilice para la lógica del cliente.

Spring Boot [2] es un marco de trabajo de Java [16] que simplifica la configuración y el arranque de aplicaciones Spring. Proporciona una forma de crear aplicaciones Spring independientes que pueden "simplemente ejecutarse", eliminando gran parte del trabajo de configuración manual. En este proyecto, Spring Boot [2] se utiliza para crear controladores y servicios, como se puede ver en PeliculaController.java y LiveContentController.java.

Maven [18] es una herramienta de gestión de proyectos y comprensión de software. Puede gestionar la construcción, el informe y la documentación de un proyecto a partir de una pieza central de información. En este proyecto, Maven [18] se utiliza para gestionar las dependencias del proyecto. Estas tecnologías actuales son fundamentales para el desarrollo y la operación de la aplicación. Cada una de ellas desempeña un papel específico y juntas permiten la creación de una aplicación web robusta y eficiente.

4.4 APLICACIONES Y USOS

En este proyecto, las tecnologías Java [16], JavaScript, Spring Boot [2] y Maven [18] se utilizan de diversas maneras para desarrollar una aplicación web de gestión de contenido en vivo.

- Java: En este proyecto, Java [16] se utiliza para la lógica del servidor. Los controladores, servicios y entidades están escritos en Java [16]. Por ejemplo, en el archivo PeliculaController.java, Java [16] se utiliza para definir los métodos que manejan las solicitudes HTTP [6] [7], como obtener todas las películas, obtener una película por su nombre, actualizar una película, etc [2].

- JavaScript: Se utiliza para la lógica del cliente en este proyecto. Para actualizar la interfaz de usuario en respuesta a las interacciones del usuario, la realización de solicitudes AJAX [17] para recuperar datos del servidor sin tener que recargar la página, y la validación de los datos del formulario antes de enviarlos al servidor [1].

- Spring Boot: Se utiliza para simplificar la configuración y el arranque de la aplicación. En este proyecto, se utiliza para definir los controladores y servicios. Por ejemplo, en el archivo PeliculaController.java, Spring Boot [2] se utiliza para definir un controlador que maneja las solicitudes HTTP [6] [7] relacionadas con las películas. También se utiliza para inyectar dependencias, como el servicio de películas, en el controlador [2].

- Maven: Se utiliza para gestionar las dependencias del proyecto. Esto significa que se utiliza para descargar y gestionar las bibliotecas que el proyecto necesita para funcionar. También se utiliza para construir el proyecto, ejecutar pruebas y generar informes [18].

- MySQL: Se utiliza como base de datos para almacenar todo el contenido multimedia de la ampliación, así como los chats de las comunidades e información de los usuarios, el principal motivo de su uso es debido a

que con Spring Data Jpa [2] [9] el proceso de consulta o envío de datos a MySQL [13] [7] se facilita y se hace muy sencillo y cómodo de integrar [7].

Todas estas tecnologías se utilizan conjuntamente para desarrollar una aplicación web que permite a los usuarios ver información sobre películas, añadir nuevas películas, actualizar películas existentes y eliminar películas. También proporciona funcionalidades para gestionar contenido en vivo y comunidades de chat.

4.5 VENTAJAS Y DESVENTAJAS

Las ventajas de estas tecnologías incluyen su madurez, su amplia comunidad de desarrolladores, y la gran cantidad de recursos de aprendizaje disponibles públicamente. También son altamente configurables y flexibles, lo que permite a los desarrolladores adaptarlas a sus necesidades específicas. Luego son tecnologías que la curva de aprendizaje es sencilla una vez que ya tienes una base sólida en Java [16] o en manejo web a nivel del cliente y la adaptación es en poco tiempo, luego la documentación disponible en internet, sobre todo de la comunidad inglesa es casi ilimitada lo que facilita todo mucho.

Sin embargo, también tienen desventajas. Por ejemplo, Java [16] y Spring Boot [2] tienen una curva de aprendizaje relativamente alta y pueden ser excesivos para proyectos pequeños. Maven [18] puede ser complicado de configurar correctamente. Git [4], aunque poderoso, puede ser difícil de entender para los principiantes. MySQL [13] [7], aunque popular, no es la base de datos más rápida ni la más escalable.

4.6 FUTURO DE LA TECNOLGÍA

El futuro de estas tecnologías parece prometedor. Java [16] y JavaScript continúan evolucionando y adaptándose a las nuevas necesidades de los desarrolladores. Spring Boot [2] sigue siendo popular y está adoptando nuevas tendencias, como la programación reactiva y cada día más y más aplicaciones web tienen como base Spring Boot [2]. Maven [18] sigue siendo una herramienta esencial en el desarrollo de Java [16], aunque está siendo desafiado por otras herramientas como Gradle. Git [4] sigue siendo el sistema de control de versiones más popular y sigue mejorando con nuevas características. MySQL [13] [7] sigue siendo una opción sólida para muchas aplicaciones, aunque está siendo desafiado por otras bases de datos como PostgreSQL y bases de datos NoSQL.

4.7 CONCLUSIÓN FINAL

En conclusión, las tecnologías utilizadas en este proyecto son maduras, populares y ampliamente utilizadas en el desarrollo de aplicaciones web. Aunque tienen sus desventajas, sus ventajas superan con creces a estas. Con la continua evolución de estas tecnologías y la aparición de nuevas tendencias y herramientas, el futuro del desarrollo de aplicaciones web parece prometedor.

5. ARQUITECTURA DEL SISTEMA

5.1 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.

- **Capa de presentación:** Esta capa es la interfaz de usuario y se encarga de interactuar con el usuario. Se ha desarrollado utilizando (HTML, CSS, JavaScript [1] [15] [11]), Ajax [17] y Thymeleaf [2] [6].

- **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 [16] y el marco de trabajo Spring Boot [2].

- **Capa de datos:** Esta capa se encarga de interactuar con la base de datos para almacenar y recuperar datos. Se ha desarrollado utilizando Java [16], Spring Data JPA [2] [9] y MySQL [13] [7].

Al final la base lógica para el tratamiento de datos y el flujo de la información es un patrón Modelo-Vista-Controlador, en donde la vista envía solicitudes al controlador ya sea de petición de datos o para pasarle datos, el controlador maneja esa información recibida y usa el modelo para obtener esos datos, actualizarlos o borrarlos en base a la petición que llegue por parte de la vista. Y una vez el modelo actúe, el controlador devuelve la petición inicial a la vista. Todo este flujo se puede ver de forma correcta y con casos reales dentro de la documentación en la parte de código fuente.

Spring Boot [2] 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 [2] [9] para el acceso a datos y Thymeleaf [2] [6] 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 [2] 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 [6] [7] 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 [6] [7], ya sean métodos GetMapping, PostMapping, PutMapping.

- 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.

- Dentro del controlador llamamos al servicio para poder usar sus métodos.

- Dentro del servicio tenemos una interfaz para definir los métodos y luego su implementación.

- Los repositorios en la capa de datos interactúan con la base de datos utilizando Spring Data JPA [2] [9] para simplificar las operaciones de la base de datos.

- Dentro del servicio invitamos al repositorio, para poder usar JPA [2] [9] y tener acceso a la base de datos.

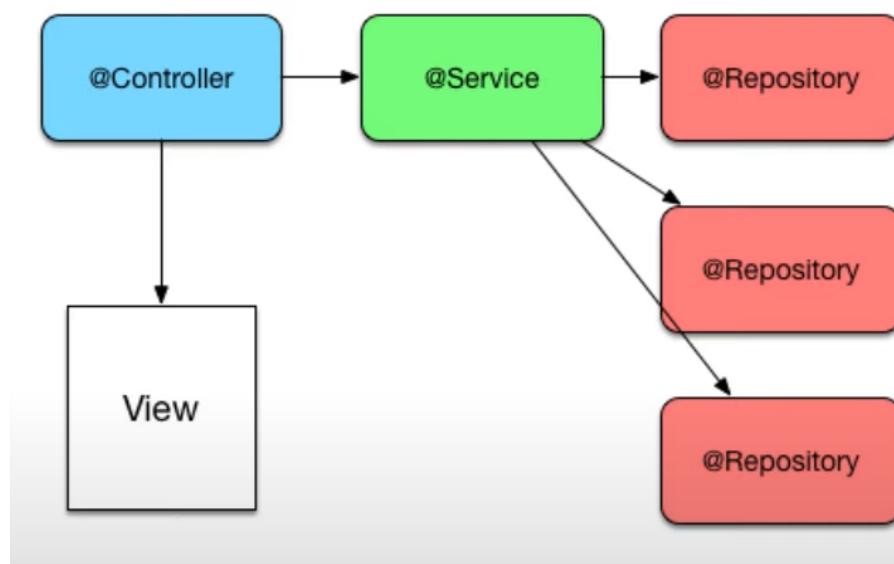


Figura 1: Funcionamiento del patrón Modelo-Vista-Controlador en Spring Boot.

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

5.2 DESCRIPCIÓN DE LOS COMPONENTES

- **Controladores:** Los controladores son parte de la capa de presentación y se encargan de manejar las solicitudes HTTP [6] [7] del usuario y utilizan los servicios para procesar estas solicitudes [2] [6].
- **Servicios:** Los servicios son parte de la capa de lógica de negocio y se encargan de aplicar las reglas de negocio y utilizar los repositorios para interactuar con la base de datos [6].
- **Repositorios:** Los repositorios son parte de la capa de datos y se encargan de interactuar con la base de datos utilizando Spring Data JPA [2] [9] para simplificar las operaciones de la base de datos [6].
- **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 [16].
- **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 [2] [5].
- **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 [2] [3].
- **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 [2] [9].
- **Pruebas:** Las pruebas se utilizan para verificar la correcta funcionalidad de la aplicación. Las pruebas incluyen pruebas unitarias y de integración [2].

6. DISEÑO DETALLADO

6.1 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, gestión de suscripciones y de pagos. 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 ComentarioPelícula, 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.

6.2 DIAGRAMAS DE CLASES

El proyecto se divide en cuatro paquetes, por un lado tenemos los controladores, por otro los servicios, por otro los repositorios y por otro las entidades, y el flujo de comunicación es entre la vista, el controlador y este se comunica con el modelo que son el repositorio y el servicio lo que facilitan el tratamiento de datos con MySQL [13] [7] y luego están las entidades que definen los atributos y las relaciones entre tablas, por lo que para un correcto entendimiento vamos a ver el diagrama de clases uno por uno, desde las entidades para entender la lógica más a nivel usuario de la web, donde se ven las relaciones entre las entidades y luego ya la parte más técnica que son los repositorios, servicios y controladores que ya no es a nivel usuario.

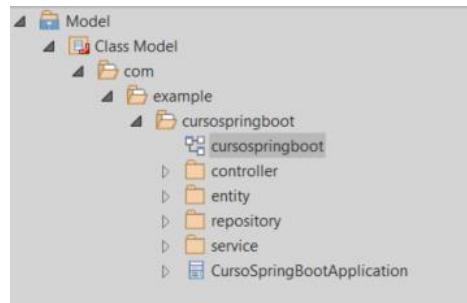


Figura 2: Estructura de los diferentes paquetes para los diagramas de clase.

6.2.1 Entidades

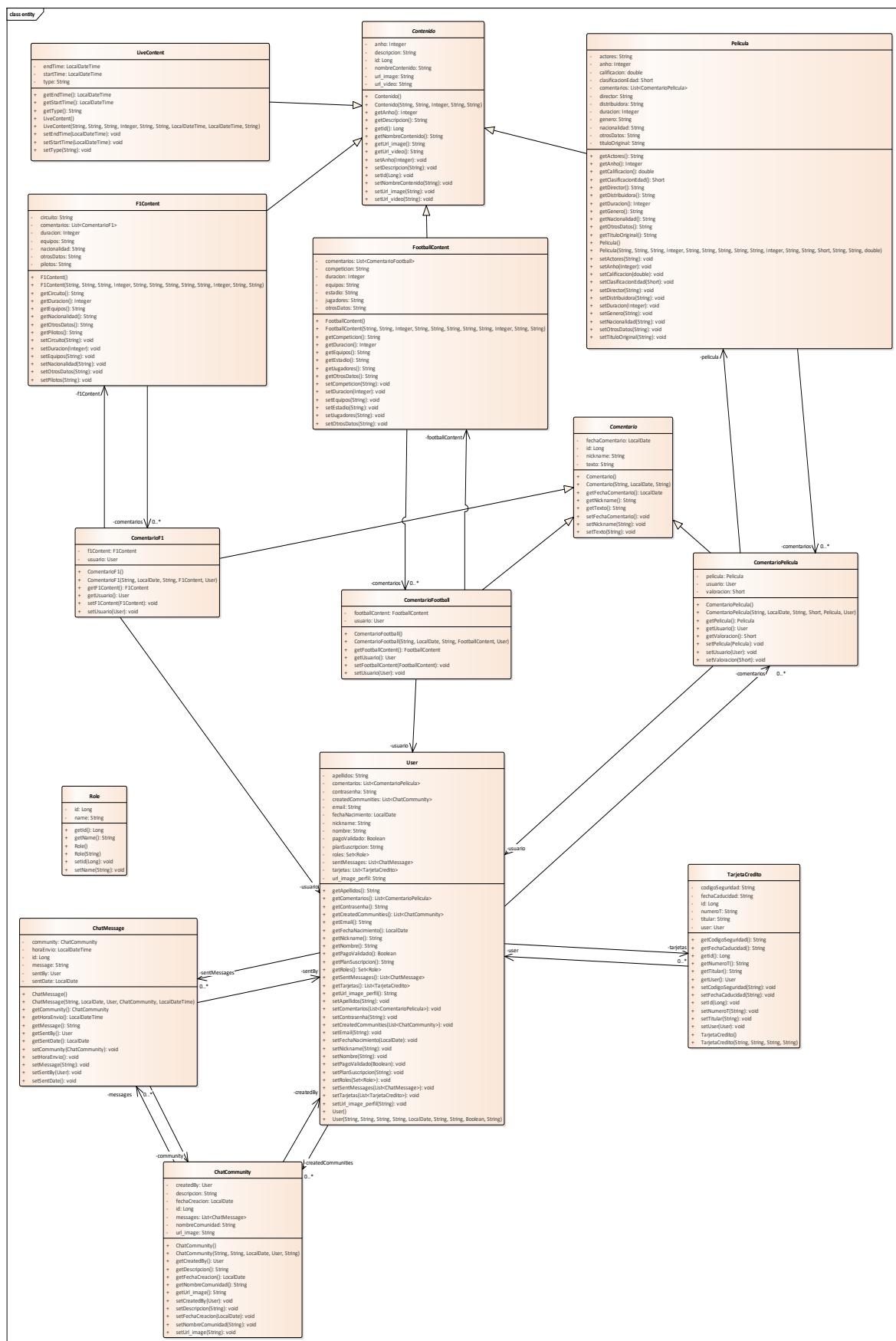


Figura 3: Diagrama de clases Entidades

- **User:** Esta entidad tiene relaciones con varias otras entidades. Tiene una relación uno a muchos con TarjetaCredito, ComentarioPelícula, 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.
- **ComentarioPelícula:** Esta entidad tiene una relación muchos a uno con User y Película.
- **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.
- **Película:** Esta entidad es una subclase de Contenido y tiene una relación uno a muchos con ComentarioPelícula.
- **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 ComentarioPelícula, ComentarioF1 y ComentarioFootball.
- **Contenido:** Esta entidad es una superclase para Película, F1Content, FootballContent y LiveContent.

6.2.2 Repositorios

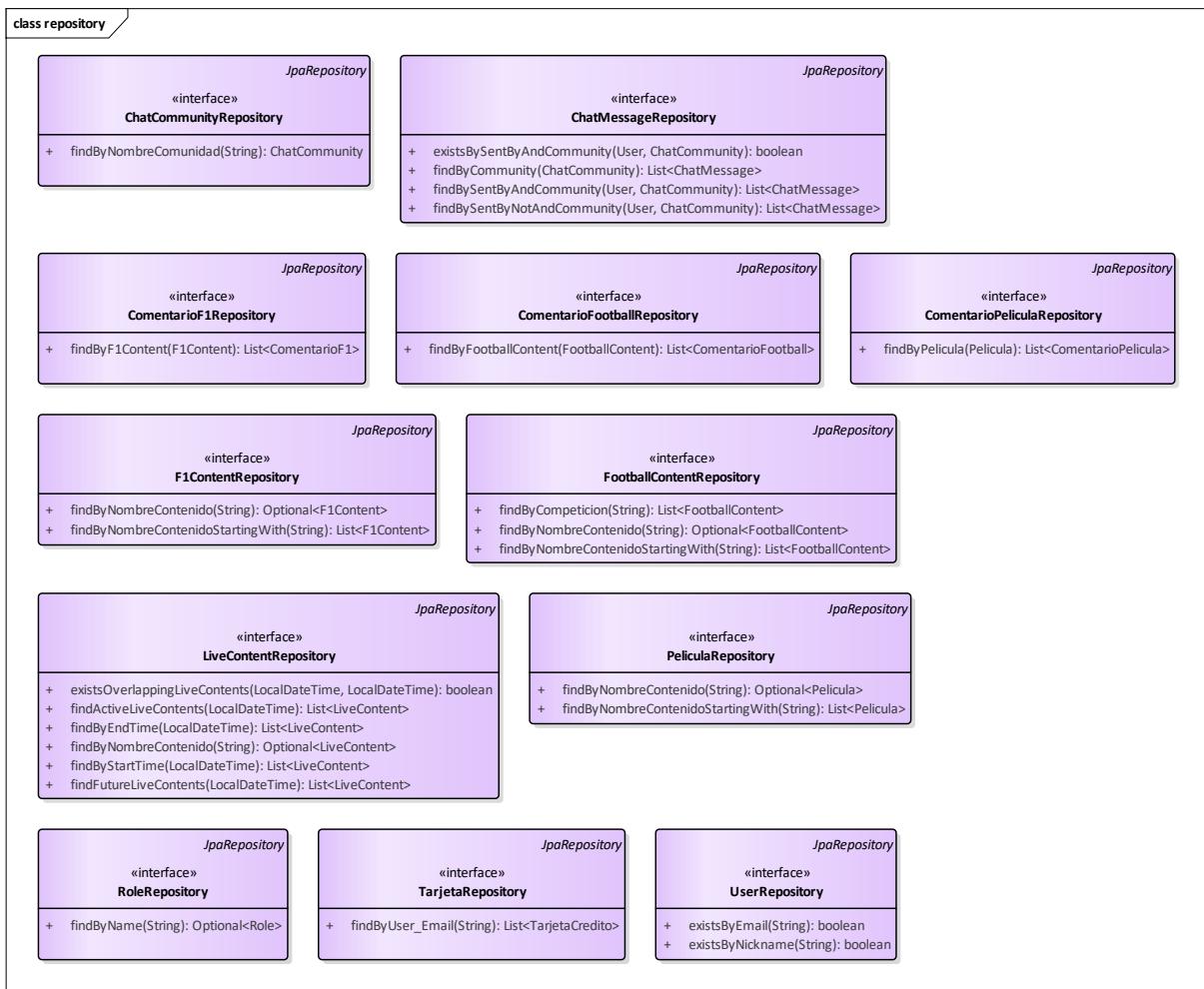


Figura 4: Diagrama de clases Repositorios.

Extienden JpaRepository [9] [2] y proporcionan métodos para interactuar con la base de datos [11]. 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.

6.2.3 Servicios

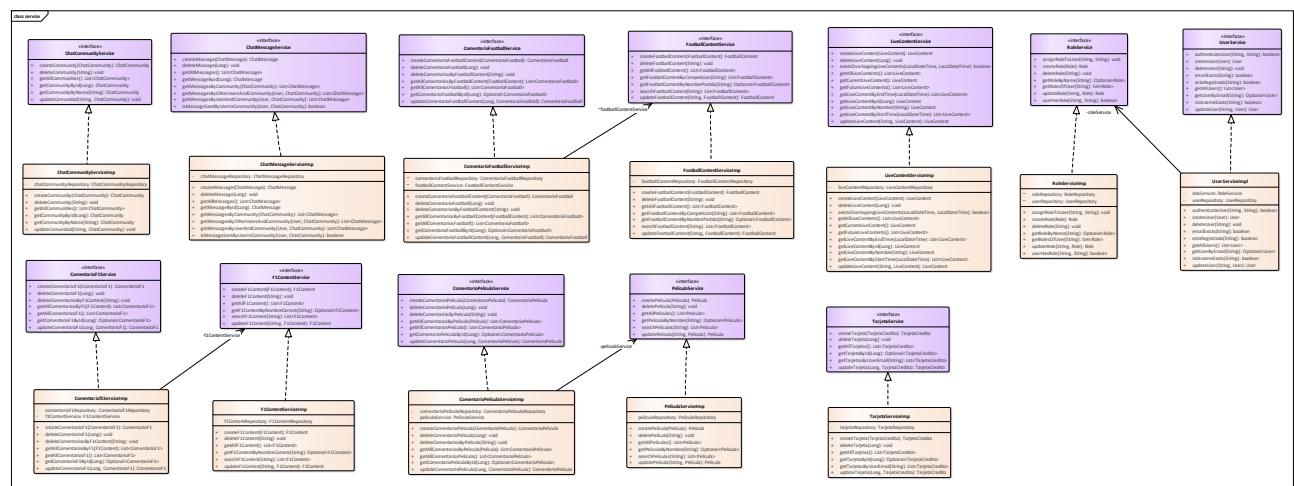


Figura 5: Diagrama de clases Servicios.

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.
- **ComentarioPelículaService**: 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 ComentarioPelículaRepository 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.

6.2.4 Controlador

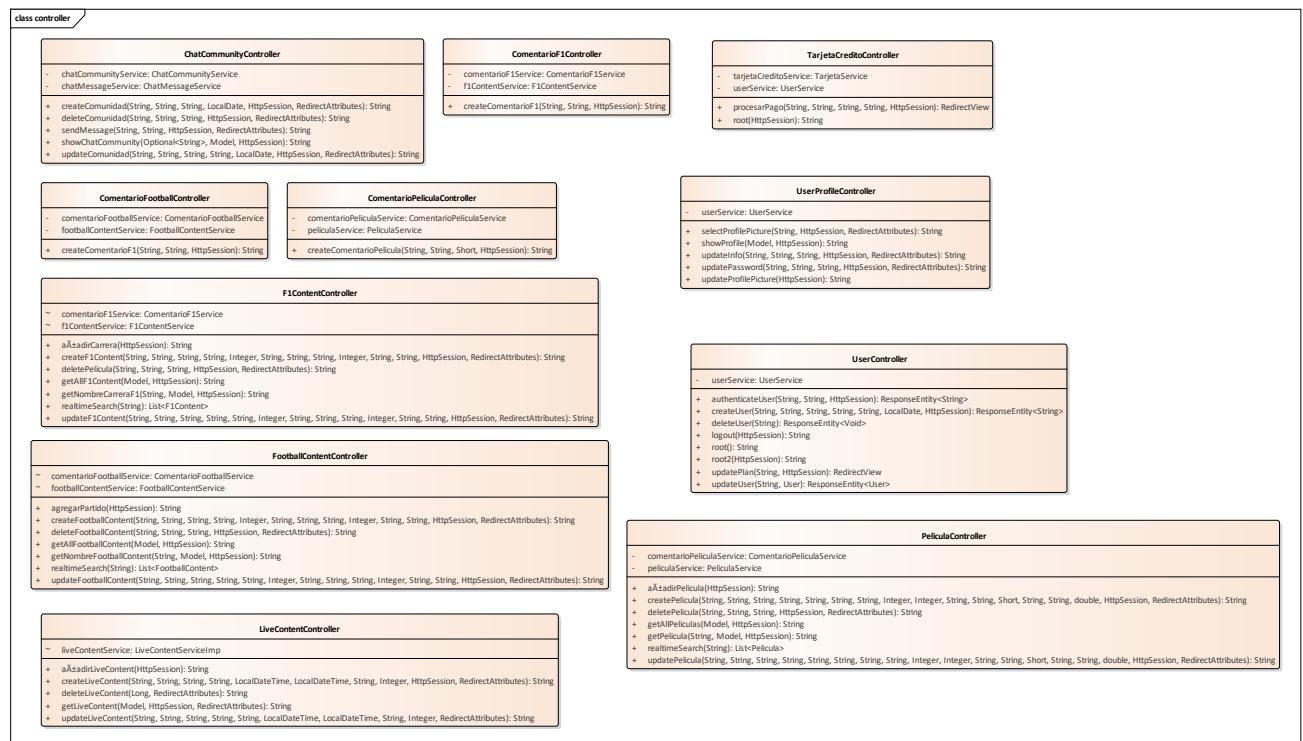


Figura 6: Diagrama de clases Controlador.

Son clases que manejan las solicitudes HTTP [6] [7] 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.
- **ComentarioPelículaController:** 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 ComentarioPelículaService 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.
- **PelículaController:** 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.

7. IMPLEMENTACIÓN

7.1 ENTORNO DE DESARROLLO

El entorno de desarrollo para este proyecto incluye:

Sistema Operativo: Windows

IDE: IntelliJ IDEA 2023.2.5

Lenguaje de Programación: Java [16] y JavaScript [1]

Marco de trabajo: Spring Boot [2]

7.1.2 ¿Por qué he decidido 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 [2] 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 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 [2] 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 entres y todo lo que hay después.

Gestor de Dependencias: Maven [18]

Sistema de Control de Versiones: Git [4]

Base de Datos: MySQL [13] [7]

7.2 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 [4].

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 [2] y Java [16] como Eclipse, NetBeans, Visual Studio Code, entre otros.

7.2.1 Configuración Básica de IntelliJ

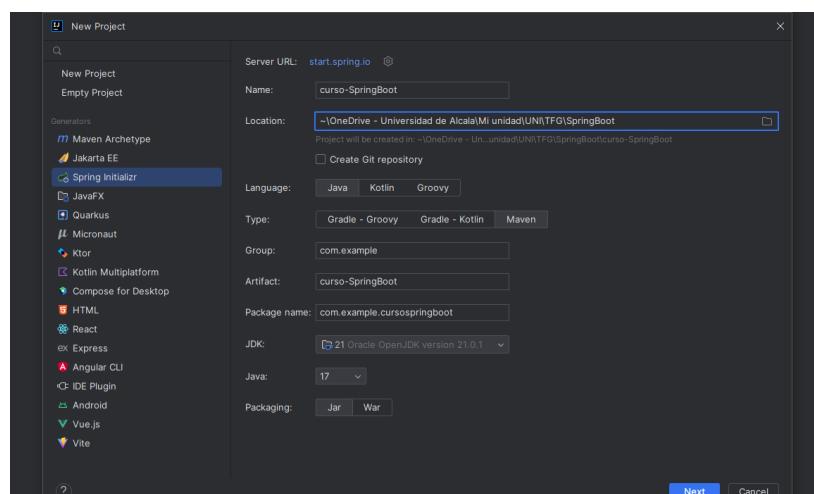


Figura 7: Ruta de creación del proyecto en Spring Boot.

Creamos un nuevo proyecto y seleccionamos la opción de Spring Initializr, el lenguaje Java, de tipo Maven [18], y si es posible la última versión de Java [16] y del JDK, en mi caso inicie el proyecto en enero de 2024.

7.2.2 Configuración de las dependencias

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

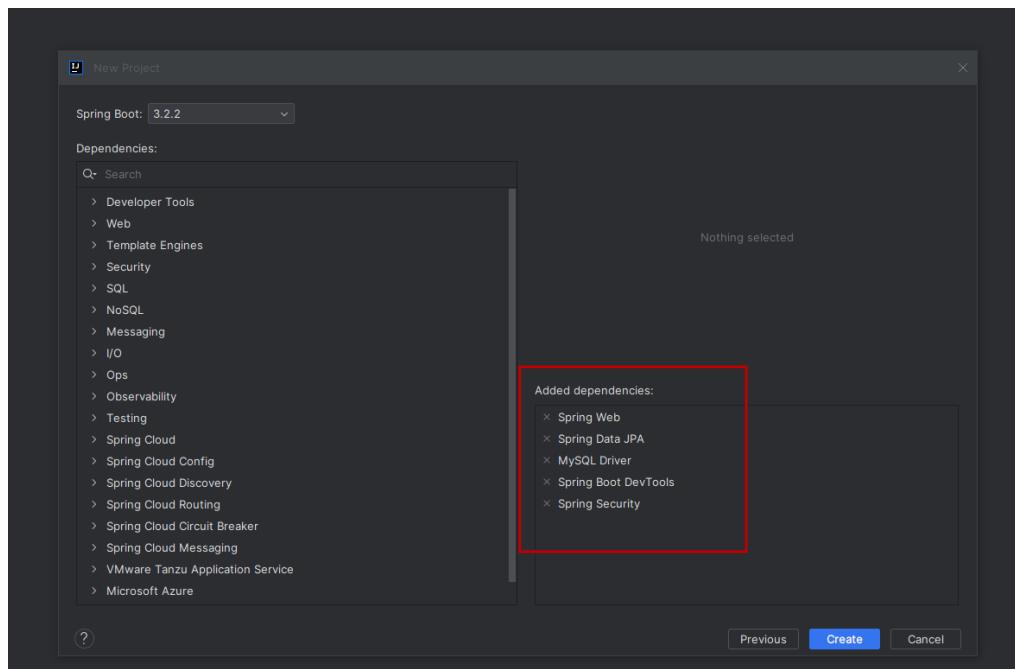


Figura 8: Como agregar las dependencias en Spring Boot.

En total son:

spring-boot-starter-data-jpa [2] [9]

spring-boot-starter-thymeleaf [2] [6]

spring-session-core [2]

spring-boot-starter-websocket [2]

spring-boot-starter-web [2]

spring-boot-devtools [2]

mysql-connector-j [2] [7]

spring-boot-starter-test [2]

spring-security-test [2] [3]

7.2.3 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 el código fuente de Java [16]. Aquí es donde se crean los controladores, servicios, repositorios, etc.

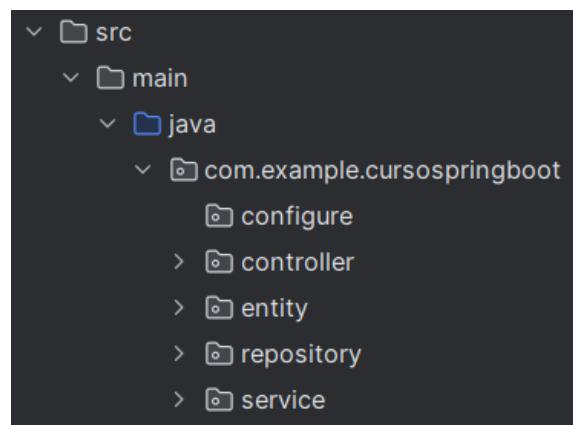


Figura 9: Organización de paquetes dentro de Spring Boot.

- **src/main/resources**: Este directorio contiene recursos como archivos de propiedades, archivos de configuración de Spring Boot [2], pero principalmente gestiona las clases que corresponden a la vista que son archivos HTML , Css y Js [1], de Thymeleaf [2] [6].

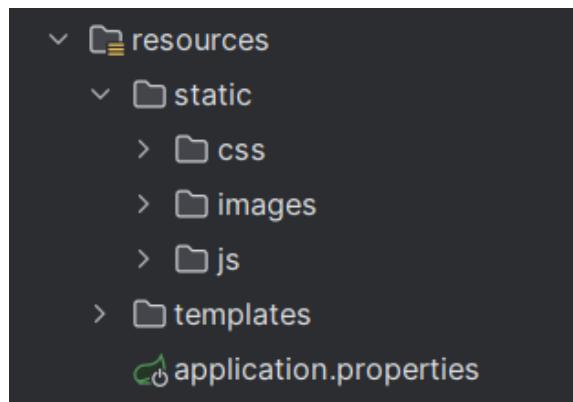


Figura 10: Organización código front-end Spring Boot.

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

- **Paquetes**: Dentro de **src/main/java**, se crean paquetes para organizar el código. En este caso siguiente la arquitectura entidad, repositorio, servicio y controlador.

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

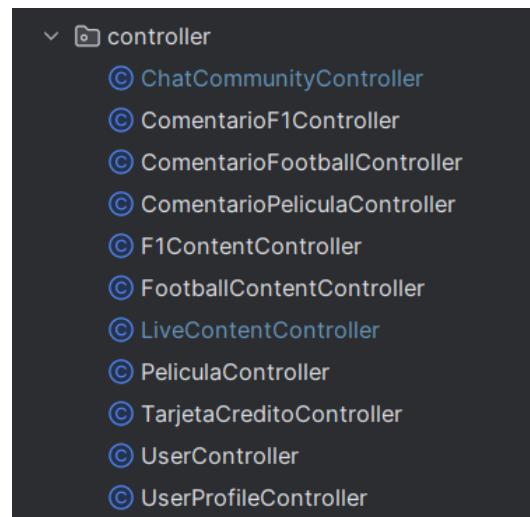


Figura 11 Organización de clases en el Controlador

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

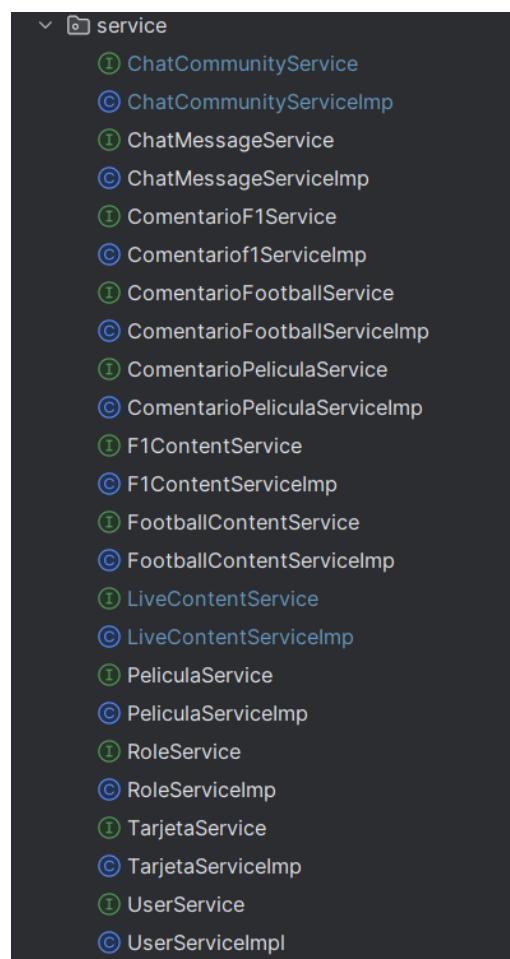


Figura 12: Organización de clases dentro del Servicio

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

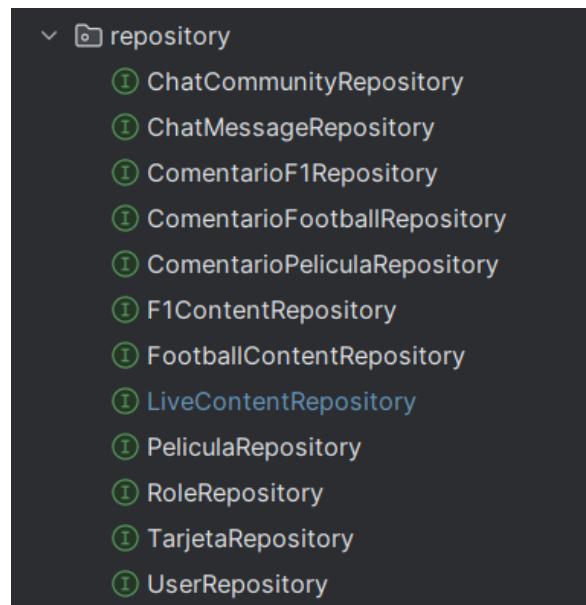


Figura 13: Organización de clases en el Repositorio

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

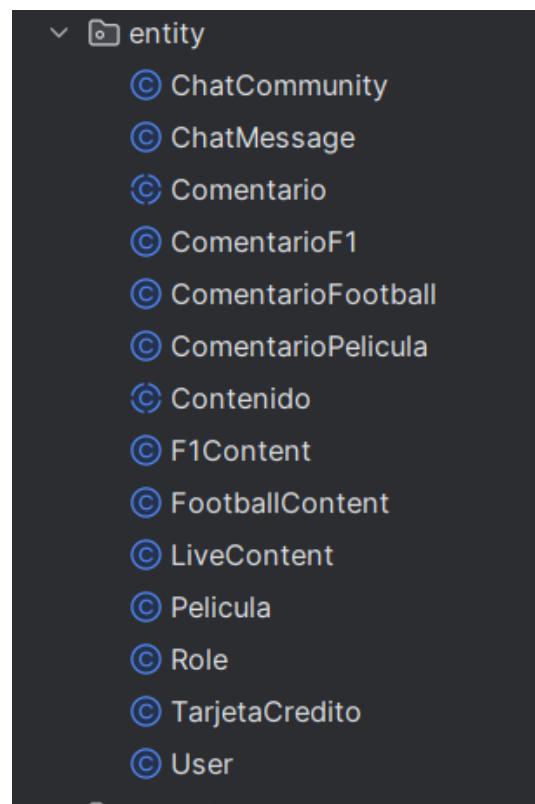


Figura 14: Organización de clases en las Entidades

- **Clases:** Dentro de estos paquetes, se crearán las clases para los controladores, servicios, repositorios y entidades. Por ejemplo, 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, y métodos para el arranque del servidor.
- **Archivos HTML [1] de Thymeleaf [2] [6]:** 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 [2], Spring Data JPA [2] [9], Thymeleaf [2] [6], 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 [18]. 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.

7.2.4 Descargar y correr el proyecto de GitHub

Uno de los requisitos a la hora de hacer este proyecto es que una vez acabado cualquier persona pudiera ejecutar la aplicación en su ordenador personal, o cualquier persona con algún interés más profundo en Spring Boot [2], en GitHub [4] también puede ver los commits y como se ha ido evolucionando el proyecto desde 0, hasta su despliegue total.

El repositorio se encuentra en <https://github.com/davidbachii/SpringBoot>, una vez ahí, dependiendo del sistema operativo en el que queramos descargar el proyecto o las herramientas que se usen para GitHub [4], lo más cómodo es clonar mediante https.

<https://github.com/davidbachii/SpringBoot.git>

O clonar mediante ssh.

<git@github.com:davidbachii/SpringBoot.git>

Pero GitHub también da la opción de descargarse el proyecto en formato zip o rar.

Una vez que se tenga el proyecto desplegado se necesita un entorno para ejecutarlo, yo en mi caso he usado IntelliJ que es con diferencia sobre el resto el mejor entorno, pero es de pago en mi caso con la cuenta educativa de la universidad lo tenía a mi disposición de manera gratuita, la opción gratuita para desplegar el proyecto es Spring Tools 4, que está disponible para eclipse <https://spring.io/tools>, estos entornos llevan un servidor tomcat integrado por lo que con descargar el proyecto, las dependencias, podríamos lanzarlo.

El último punto es la base de datos que está en MySQL [13], que realmente da un poco igual ya que lo único necesario es que sea relacional, por lo que se podría usar MongoDB, PostgreSQL, entre otras, yo en mi caso uso SQL por la dependencia de Spring Boot [2] MySQL Driver, lo mejor es crear una base de datos desde 0, luego en el archivo de configuración del proyecto ponemos el nombre de la base de datos y el usuario y la contraseña que hayas establecido y pones esta configuración:

```
spring.datasource.url=jdbc:mysql://localhost:3306/multimedia?useSSL=false&serverTimezone=Europe/Madrid
spring.datasource.username=root
spring.datasource.password=user
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.show-sql=true
```

```
spring.jpa.open-in-view=false
spring.jpa.hibernate.ddl-auto=update
logging.level.org.hibernate.SQL=debug
```

Que está en la ruta [curso-SpringBoot/src/main/resources/application.properties](#).

No obstante, en la bibliografía [7] y [6] se ve detalladamente y de forma visual como se hace la conexión con mysql [13] y el proyecto de Spring Boot [2].

Una vez hecha, arrancamos el proyecto y las tablas se crean con las relaciones definidas en las entidades, y para llenar las tablas de contenido, dentro del repositorio hay un archivo llamado [multimedia.sql](#) que está repleto de contenido para agregarlo en la aplicación de forma rápida, pero siempre está la opción de que el administrador lo agregue a su gusto.

8. CÓDIGO FUENTE

Spring Boot y la generación de la base de datos:

Spring Boot [2] utiliza el framework **Hibernate** [6] 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 [6] 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 [6] 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.

8.1 Modelos

8.1.1 Contenidos

8.1.1.1 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.

8.1.1.2 Contenido Película

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.

Repository Película

El repositorio PeliculaRepository es una interfaz que extiende JpaRepository, [2] [9] 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 [2] [9] 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 **createPelícula** toma un objeto Película como parámetro y lo guarda en la base de datos utilizando el método save del repositorio.

El método **updatePelícula** toma el nombre de una película y un objeto Película 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 **deletePelícula** 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 **searchPelículas** 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.

8.1.1.3 Contenido F1

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, otrosDatos, 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, [2] [9] 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 [2] [9] 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.

8.1.1.4 Contenido Fútbol

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, otrosDatos, 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, [2] [9] 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 [2] [9] 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 lo 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.

8.1.1.5 Contenido en Directo

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, [2] [9] 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 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 hay 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 está 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 hay 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 directo, 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 **getLiveContentByName** 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.

8.1.2 Comentarios

8.1.2.1 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 = "userNickname", 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.

8.1.2.2 Contenido Comentario Película

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 @OneToOne 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.

Repository Comentario de Película

El repositorio `ComentarioPeliculaRepository` es una interfaz que extiende JpaRepository, [2] [9] 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 [2] [9] 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.getPeliculaByName(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 **getAllComentariosByPelícula** 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.

8.1.2.3 Contenido Comentario F1

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 **@OneToOne** 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, [2] [9] 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 [2] [9] 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);
    }
}
```

```

public void deleteComentariosByF1Content(String nombreCarreraF1) {
    // Fetch the F1 content by its name
    F1Content f1Content = f1ContentService.getF1ContentByName(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);
}
}

```

Esta 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.

8.1.2.4 Contenido Comentario Fútbol

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 los partidos de fútbol. La anotación `@Entity` indica que esta clase se mapeará a una tabla en la base de datos. La anotación `@ManyToOne` 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.

Repository Comentario de Contenido de Fútbol

El repositorio `ComentarioFootballRepository` es una interfaz que extiende JpaRepository, [2] [9] proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de ComentarioFootballContent 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 [2] [9] 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 `ComentarioFootballService`, 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/ComentarioFootballServiceImp.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.

8.1.3 Usuario

8.1.3.1 Contenido 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 está el administrador que tiene los 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.

Repository Usuario

El repositorio `UserRepository` es una interfaz que extiende JpaRepository, [2] [9] 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 [2] [9] genera automáticamente a partir de su nombre. Estos métodos permiten tener un

control para saber si el nickname y el usuario ya está 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 `UserServiceImpl` implementa la interfaz `UserFootballService`, 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.getContraseña().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 [2] [9] 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.

8.1.3.2 Contenido Usuario Rol

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`, [2] [9] 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 [2] [9] 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.

8.1.4 Comunidades de Usuarios

8.1.4.1 Contenido Comunidades de Usuarios

Clase ChatCommunity

La clase [ChatCommunity](#) representa las comunidades de usuarios en el sistema. Dentro de la aplicación los usuarios con un plan de suscripción pro tienen la posibilidad de participar en las comunidades de usuarios basadas en los contenidos multimedia de la aplicación, que permiten a los usuarios poder hablar con otros de forma cotidiana, compartir ideas, opiniones. Con el plan básico se podrán ver las conversaciones de las comunidades, pero no se podrá participar en ellas. En concreto esta clase se centra solo en las comunidades luego habrá otra para los chats.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/entity/ChatCommunity.java.`

```
@Entity
@Table(name = "ChatCommunity")
public class ChatCommunity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "nombreComunidad", length = 30, nullable = false)
    private String nombreComunidad;
    @Column(name = "descripcion", length = 1000, nullable = false)
    private String descripcion;
    @Column(name = "fechaCreacion", length = 12, nullable = false)
    private LocalDate fechaCreacion;
    @Column(name = "url_image", length = 100, nullable = false)
    private String url_image;
    @ManyToOne
    private User createdBy;
    @OneToMany(mappedBy = "community", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<ChatMessage> messages;
```

Almacena información como el nombre de la Comunidad, una breve descripción y una fecha de creación. Esta clase mantiene una relación uno a muchos con el usuario para que quede recogido el usuario que crea la comunidad en cada momento y una relación uno a muchos con los mensajes de los chats para tener recogidos los mensajes que correspondan a cada comunidad.

Repository ChatCommunity

El repositorio `ChatCommunityRepository` es una interfaz que extiende `JpaRepository`, [2] [9] proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de `ChatCommunity` en la base de datos.

Esta interfaz se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/repository/ChatCommunityRepository.java.
```

```
@Repository
public interface ChatCommunityRepository extends JpaRepository<ChatCommunity, Long> {
    ChatCommunity findByNombreComunidad(String nombreComunidad);
}
```

El método `findByNombreComunidad` es un ejemplo de un método de consulta derivados, que Spring Data JPA [2] [9] genera automáticamente a partir de su nombre. Este método permite tener un control para poder tener todos los datos de la comunidad pasándole solo el nombre de esta. El cuales usare en la creación de los servicios que a su vez serán usados por el controlador.

Servicio ChatCommunity

El servicio `ChatCommunityServiceImp` implementa la interfaz `ChatCommunityRepository`, proporcionando la lógica de negocio para las operaciones relacionadas con las comunidades de usuarios.

Esta clase se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/service/ChatCommunityServiceImp.java.
```

```
@Service
public class ChatCommunityServiceImp implements ChatCommunityService{
    @Autowired
    private ChatCommunityRepository chatCommunityRepository;
    @Override
    public List<ChatCommunity> getAllCommunities() {
        return chatCommunityRepository.findAll();
    }
    @Override
    public ChatCommunity getCommunityById(Long id) {
        return chatCommunityRepository.findById(id).orElse(null);
    }
    @Override
    public ChatCommunity getCommunityByName(String nombreComunidad) {
        return chatCommunityRepository.findByNombreComunidad(nombreComunidad);
    }
}
```

```
    }

    @Override
    public ChatCommunity createCommunity(ChatCommunity chatCommunity) {
        return chatCommunityRepository.save(chatCommunity);
    }

    @Override
    public void deleteCommunity(String nombreComunidad) {
        ChatCommunity existingCommunity = chatCommunityRepository.findByNombreComunidad(nombreComunidad);
        if (existingCommunity != null) {
            chatCommunityRepository.delete(existingCommunity);
        }
    }

    @Override
    public void updateComunidad(String nombreComunidad, ChatCommunity detallesComunidad) {
        ChatCommunity existingCommunity = chatCommunityRepository.findByNombreComunidad(nombreComunidad);
        if (existingCommunity != null) {
            existingCommunity.setDescripcion(detallesComunidad.getDescripcion());
            existingCommunity.setNombreComunidad(detallesComunidad.getNombreComunidad());
            existingCommunity.setUrl_image(detallesComunidad.getUrl_image());
            existingCommunity.setFechaCreacion(detallesComunidad.getFechaCreacion());
            chatCommunityRepository.save(existingCommunity);
        }
    }
}
```

Esta clase implementa la interfaz ChatCommunityService y proporciona métodos para interactuar con la entidad ChatCommunity en la base de datos. Utiliza ChatCommunityRepository para acceder a los datos de las comunidades. Principalmente operaciones CRUD básicas.

El método **getAllCommunities**, devuelve una lista que recoge todas las comunidades usando el método findAll del repositorio que nos proporciona JPA [2] [9].

El método **getCommunityByName**, nos devuelve la entidad de la comunidad pasándole el nombre es el método que hemos creado en el repositorio.

El método **createCommunity**, nos sirve para crear una comunidad usando el método save del repositorio que nos proporciona JPA [2] [9].

El método **deleteCommunity**, nos sirve para borrar las comunidades, dentro del método buscamos primero si la comunidad que se desea borrar existe si es así, usamos el método delete que del repositorio que nos proporciona JPA [2] [9]. Si esa comunidad no existe devuelve null.

El método **updateComunidad**, nos sirve para actualizar una comunidad, pasamos como atributo el nombre de la comunidad y su entidad, hacemos el chequeo para verificar que es una comunidad que ya existe y si es así actualizamos sus atributos uno por uno.

Clase ChatMessage

La clase `ChatMessage` representa los chats de las comunidades de usuarios en el sistema. Dentro de la aplicación los usuarios con un plan de suscripción pro tienen la posibilidad de participar en las comunidades de usuarios basadas en los contenidos multimedia de la aplicación, que permiten a los usuarios poder hablar con otros de forma cotidiana, compartir ideas, opiniones. Con el plan básico se podrán ver las conversaciones de las comunidades, pero no se podrá participar en ellas. En concreto esta clase se centra solo en los mensajes que envían los usuarios en las comunidades

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/entity/ChatMessage.java.`

```
@Entity
@Table(name = "ChatMessage")
public class ChatMessage {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "message", length = 1000, nullable = false)
    private String message;
    @Column(name = "sentDate", length = 12, nullable = false)
    private LocalDate sentDate;
    @Column(name = "horaEnvio", nullable = false)
    private LocalDateTime horaEnvio;
    @ManyToOne
    private User sentBy;
    @ManyToOne
    private ChatCommunity community;
```

Almacena información como el contenido del mensaje que los usuarios envían, la fecha en la que se envía el mensaje y la hora del mismo, hay una relación muchos a uno con los usuarios para tener registrado al usuario que escribe el mensaje en cada momento. Los usuarios con el plan pro no tienen límite de mensajes, y una relación muchos a uno con las comunidades.

Repositorio ChatMessage

El repositorio `ChatMessageRepository` es una interfaz que extiende `JpaRepository`, [2] [9] proporcionando métodos para realizar operaciones CRUD (crear, leer, actualizar, eliminar) en la tabla de `ChatMessage` en la base de datos.

Esta interfaz se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/repository/ChatMessageRepository.java.`

```
@Repository
public interface ChatMessageRepository extends JpaRepository<ChatMessage, Long> {
    List<ChatMessage> findBySentByAndCommunity(User user, ChatCommunity community);
    List<ChatMessage> findBySentByNotAndCommunity(User user, ChatCommunity community);
```

```
    List<ChatMessage> findByCommunity(ChatCommunity community);
    boolean existsBySentByAndCommunity(User user, ChatCommunity community);
}
```

El método **findBySentByAndCommunity** es un ejemplo de un método de consulta derivados, que Spring Data JPA [2] [9] genera automáticamente a partir de su nombre. Este método nos permite obtener los mensajes concretos de una comunidad escritos por un usuario, el cual usare en la creación de los servicios que a su vez serán usados por el controlador. También está el método de **findByCommunity** que te devuelve todos los mensajes de una comunidad concreta y el método de **existsBySentByAndCommunity** que es un booleano que uso para verificar si hay mensajes escritos por un determinado usuario antes de hacer la búsqueda de los mismos.

Por último, está el método de **findBySentByNotAndCommunity**, que nos devolverá todos los mensajes de todos los usuarios que hayan escrito en la comunidad menos el usuario que ha iniciado la sesión. Que es el usuario que vamos a pasarle al método.

Servicio ChatMessage

El servicio ChatMessageServiceImp implementa la interfaz ChatMessageRepository, proporcionando la lógica de negocio para las operaciones relacionadas con los mensajes de las comunidades de usuarios.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/service/ChatMessageServiceImp.java.`

Esta clase implementa la interfaz ChatMessageService y proporciona métodos para interactuar con la entidad ChatMessage en la base de datos. Utiliza ChatCommunityRepository para acceder a los datos de las comunidades a través de los métodos de JPA [2] [9] y ChatMessageRepository para acceder a los mensajes de los chats. Aparte de los métodos CRUD básicos, creamos otros métodos que nos serán útiles en el controlador para un buen diseño en el front, ya que a la izquierda están los mensajes que escriben el resto de usuarios y a la derecha los mensajes del usuario que tiene la sesión iniciada, de forma que necesitamos recoger los mensajes del usuario que ha iniciado la sesión para mostrarlos y por otro lado todos los mensajes menos los de usuario que no ha iniciado sesión para mostrarlos en el lado derecho.

El método **getAllMessages**, devuelve una lista que recoge todos los mensajes usando el método findAll del repositorio que nos proporciona JPA [2] [9].

El método **createMessage**, nos sirve para crear un nuevo mensaje usando el método save del repositorio que nos proporciona JPA [2] [9].

El método **deleteMessage**, nos sirve para borrar un determinado mensaje, usamos el método delete del repositorio que nos proporciona JPA [2] [9].

El método **getMessagesByCommunity**, nos devuelve todos los mensajes de una determinada comunidad usando el método findByCommunity que hemos definido en el repositorio.

El método **getMessagesByUserAndCommunity**, devuelve los mensajes enviados por un determinado usuario en una determinada comunidad usando el método que hemos definido en el repositorio de findBySentByAndCommunity, este método se usara para que tener los mensajes del usuario que ha iniciado la sesión.

El método `getMessagesByOtherUsersAndCommunity`, devuelve los mensajes enviados todos los usuarios menos el usuario que pasamos como atributo en una determinada comunidad usando el método que hemos definido en el repositorio de `findBySentByNotAndCommunity`, este método se usara para que tener los mensajes menos los del usuario que ha iniciado la sesión.

El método `isMessageSentByUserInCommunity`, devuelve un valor booleano el cual va a ser útil en el controlador para saber si un determinado mensaje ha sido enviado por un determinado usuario en una determinada comunidad.

8.2 Controladores y Vista

Como concepto básico para entender la función del controlador es básicamente el eje que mueve todo , su función es recibir peticiones por parte de las vistas ya sea para proporcionar datos, para almacenar datos, para verificar información y este solicita al modelo esos datos y los devuelve la petición a la vista, en Spring Boot [2] la parte del modelo la gestiona JPA [2] [9] mediante el uso de repositorios y servicios y el controlador tiene acceso a esos servicios los cuales usa para completar las peticiones que le lleguen por parte de la vista, cuando hablo de vista me refiero a la interfaz de usuario, al frontend.

Siguiendo con el controlador dentro SpringBoot utilizo la dependencia `HttpSession` [6] [7] para tener un control de las sesiones de los usuarios cuando inician y cierran sesión, también es muy útil ya que puedo acceder a los datos del usuario cuando ha iniciado sesión así puedo ver sus planes de suscripción para controlar al contenido que accede o acceder a cualquier dato del usuario, se usa principalmente para hacer chequeo de que el usuario está registrado, tiene el pago validado y todo en orden.

```
if (user == null || user.getPagoValidado().equals(false) || "Sin
Plan".equals(user.getPlanSuscripcion())) {
    return "login"; // Redirect the user to the login page if not authenticated
}
```

De esta forma verificamos que el usuario si tenga un plan, que exista y que tenga el pago validado, si no es así le mandamos al panel de registro e inicio de sesión.

También utilizamos `Model` que sirve para enviar datos a la vista lo que nos va a servir de mucha utilidad para pasar listas de contenido, pasar usuarios, comunidades, chats, etc.

```
List<LiveContent> liveContentsAll = liveContentService.getAllLiveContents();
model.addAttribute("liveContents", liveContentsAll);
```

Esto sería un ejemplo de implementación donde restas recogiendo todos los contenidos en directo en el controlador usando el método del repositorio y estamos pasando ese contenido a la vista con el nombre de `liveContents`.

También utilizamos `RedirectAttributes` para pasarle mensajes de errores y o de acierto a la vista, en todas las clases de la vista tenemos un atributo recogido de Bootstrap [8] para mostrar mensajes de error o de validación.

```
<div th:if="${successMessage}" class="alert alert-success wc_notice"
th:text="${successMessage}"></div>
```

```
<!-- Mensaje de error -->
<div th:if="${errorMessage}" class="alert alert-danger wc_notice"
th:text="${errorMessage}"></div>
```

Desde el controlador le pasamos a la vista los mensajes.

```
if (user.getPlanSuscripcion().equals("Gratis") || user.getPlanSuscripcion().equals("Basico")) {
    redirectAttributes.addFlashAttribute("errorMessage", "Actualiza el plan de
suscripción para acceder a este contenido");
    return "redirect:/api/userProfile/";
}
```

Aquí vemos un ejemplo en caso de que el plan de suscripción sea gratis, enviamos el mensaje de error que el cliente va a ver en su interfaz.

@GetMapping y **@PostMapping** son anotaciones que en Spring Boot [2] que se utilizan para mapear solicitudes HTTP [6] [7] a métodos específicos en un controlador.

@GetMapping: Esta anotación maneja las solicitudes HTTP GET [6] [7]. Se utiliza para leer datos del servidor y puede ser considerada segura e idempotente ya que no modifica ningún recurso en el servidor. Por ejemplo, el método en el controlador **@GetMapping("/ {nombrePelicula}")**, se encargará de manejar todas las solicitudes GET que lleguen a la ruta /api/peliculas/{nombrePelicula}, donde {nombrePelicula} es una variable de ruta que se pasará al método.

@PostMapping: Esta anotación maneja las solicitudes HTTP POST [6] [7]. Se utiliza para enviar datos al servidor para crear un nuevo recurso. No es ni segura ni idempotente, ya que modifica el estado del servidor al crear un nuevo recurso. Por ejemplo, **@PostMapping("/crear")**, este método manejará todas las solicitudes POST que lleguen a la ruta /api/peliculas/crear.

Estas anotaciones permiten un manejo completo de los tipos de solicitudes HTTP [6] [7] de manera diferente. Yo en mi caso he tengo un método que maneje las solicitudes GET para leer datos y otro método que maneje las solicitudes POST para crear nuevos datos.

Cuando hablamos de vista se trata de la interfaz con la que el usuario maneja la aplicación, en esta interfaz está repleta de funcionalidades que el usuario según sus gustos y preferencias o según su nivel de usuario puede hacer unas cosas u otras. Cuando el usuario solicita hacer cualquier acción, por ejemplo, iniciar sesión, clicar en un contenido para verlo, enviar un mensaje, esta enviado una petición al controlador este la tramita y le devuelve esa petición.

Una clave muy importante dentro de la vista es el diseño de una interfaz fluida y elegante que haga al usuario final más amena su experiencia, las tecnologías que se han usado para esto son (HTML, CSS, JavaScript [1] [15] [11]), Ajax [17] y todo basado en Bootstrap [8].

```
<link th:href="@{/css/bootstrap.css}" href="../static/css/bootstrap.css" rel="stylesheet">
```

También se han utilizado diseños hechos ya en telegram como el de la tarjeta de crédito, o el panel de inicio de sesión, cuando se hable de ellos se proporcionarán enlaces.

La dependencia más importante para la parte de la vista es **Thymeleaf** [2] [6], 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 [1] estándar. Esto es esencial para la creación de la interfaz de usuario de la aplicación.

```
<p class="price" style="font-size:1.1em; padding: 5px;">
```

```
th:text="${pelicula.getNombreContenido()}"></p>
```

Aquí vemos un ejemplo de thymeleaf [2] [6] donde en un campo de texto estamos pasando el nombre de la película. Esto nos va ayudar ya que nosotros muchas veces pasamos valores a la vista, con thymeleaf [2] [6] los podemos recoger y los podemos interpretar a nuestro gusto.

8.2.1 Controladores y Vista Contenidos

Controlador/Vista Película

La clase `PeliculaControllerJava` se encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para la sección de películas que en la parte de la vista tendremos un listado de todas las películas con un buscador integrado y luego cuando el usuario seleccione la película que quiere ver dentro de esta podrá ver datos y ver su contenido.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/controller/ PeliculaController.java.`

La clase index es la encargada de la vista por parte de las películas donde se ven todas las películas disponibles en la plataforma y indexDetallado es para cuando se selecciona un contenido en index, verlo y ver su información aparte también se incluye la sección de los comentarios

Estas clases se encuentran en el archivo:

`src/main/resources/templates/index.html`

`src/main/resources/templates/indexDetallado.html`

El flujo va a ser igual en esta documentación para la vista y el controlador, vamos a ver los métodos como se solicitan los métodos get y post desde la vista al controlador y como actúa el controlador en cada caso, acompañado de código y de imágenes.

Definimos la etiqueta de `@Controller`, para que Spring entienda que se trata del controlador y definimos la ruta del controlador `@RequestMapping("/api/peliculas")`.

Con la etiqueta `@Autowired` podemos implementar los servicios que incluyen los métodos de los repositorios.

Métodos GET

Definimos la ruta del controlador

```
@RequestMapping("/api/peliculas")
```

Método **getAllPelículas(Model model [2], HttpSession sesión [6] [7])**: Este método está anotado con **@GetMapping("/")**, lo que significa que maneja las solicitudes GET a la ruta /api/películas/. Este método recupera todas las películas de la base de datos y las añade al modelo para ser mostradas en la vista.

```
@GetMapping("/")
public String getAllPelículas(Model model, HttpSession session) {
    List<Película> listaPelículas = películaService.getAllPelículas();
    model.addAttribute("listaPelículas", listaPelículas);
    // Add the roles of the user to the model
    User user = (User) session.getAttribute("user");
    if (user != null) {
        Set<Role> roles = user.getRoles();
        model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.toList()));
    }
    return "index";
}
```

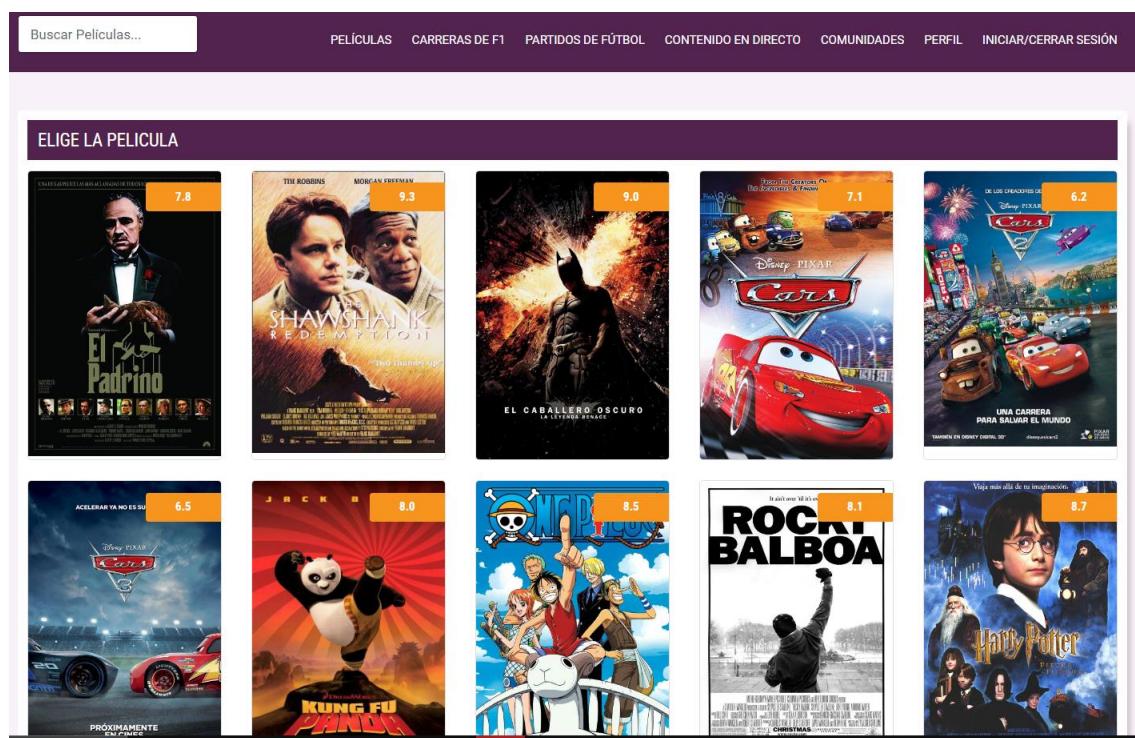


Figura 15: Panel que implementa el método `getAllPelículas`

A la vista le pasamos el rol del usuario y le pasamos una lista con todas las películas luego dentro de la vista lo mostramos tal que así:

```
<div th:each="pelicula : ${listaPeliculas}" class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
    <div class="card card-event info-overlay">
        <h1 class="tag" th:text="${pelicula.getCalificacion()}></h1>
        <div class="img has-background">
            <a th:href="@{'/api/peliculas/' + ${pelicula.getNombreContenido()}">
                class="event-pop-link">
                    <div class="event-pop-info">
                        <p class="price" style="font-size:1.1em; padding: 5px;">
                            th:text="${pelicula.getNombreContenido()}"</p>
                        <span style="font-size: 0.90em; background-color: #53234f;">
                            class="badge badge-primary">VER CONTENIDO</span><br/><br/>
                        </div>
                    </a>
                    <a th:href="@{'/' + ${pelicula.getUrl_image})">
                        </a>
                </div>
            </div>
        </div>
    </div>
```

Con thymeleaf [2] [6] recorremos todas las películas y vamos mostrando datos de estas como la valoración, el nombre y su imagen.

Método **getPelicula(@PathVariable String nombrePelicula , Model model, HttpSession sesión [6] [7])**: Este método está anotado con `@GetMapping("/{nombrePelicula}")`, lo que significa que maneja las solicitudes GET a la ruta /api/peliculas/{nombrePelicula}. Este método recupera una película específica de la base de datos por su nombre y la añade al modelo para ser mostrada en la vista. El cual se va a usar cuando en la vista el usuario seleccione una película concreta para ver.

```
@GetMapping("/{nombrePelicula}")
public String getPelicula(@PathVariable String nombrePelicula, Model model, HttpSession session) {
    Pelicula pelicula = peliculaService.getPeliculaByNombre(nombrePelicula)
        .orElseThrow(() -> new RuntimeException("Película 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
    }
    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";
}
```

}

Dentro de la vista cuando el usuario seleccione la película que desea ver salta al indexDetallado.

```
<a th:href="@{'/api/peliculas/' + ${pelicula.getNombreContenido()}}"
```

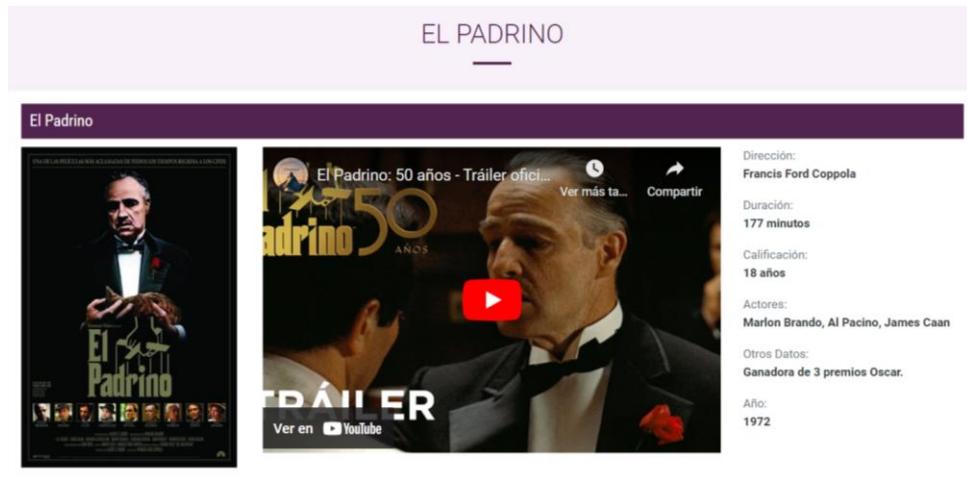


Figura 16: Panel que implementa el método getPelícula

En la parte de la vista con thymeleaf [2] [6] se recoge la película y solo se muestran sus datos específicos como la sinopsis, el año, los actores y con la ayuda de Bootstrap [8] lo colocamos todo.

Método **añadirPelícula(HttpServletRequest sesión)** anotado con `@GetMapping("/añadirPelícula")`, lo que significa que maneja las solicitudes GET a la ruta /api/películas/añadirPelícula. Este método verifica si el usuario actual está autenticado y es un administrador. Si es así, redirige al usuario a la página de agregar contenido. Si no, redirige al usuario a la página de inicio de sesión.

```
@GetMapping("/añadirPelícula")
public String añadirPelícula(HttpServletRequest session) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login";
    }
    return "agregarContenido";
}
```

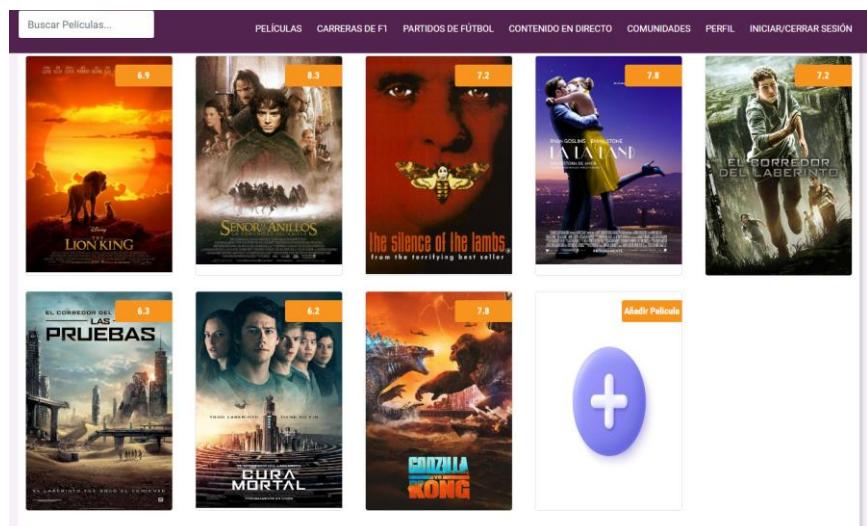


Figura 17: Panel que implementa el método añadirPelícula

```
<div th:if="${roles != null and #lists.contains(roles, 'ADMIN')}" class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20" >
    <div class="card card-event info-overlay">
        <h1 class="tag">Añadir Película</h1>
        <div class="img has-background">
            <a th:href="@{'/api/peliculas/añadirPelícula'}" class="event-pop-link">
                <div class="event-pop-info">
                    <p class="price" style="font-size:1.1em; padding: 5px;"> Pulsa para agregar un
                        nuevo contenido</p>
                    <span style="font-size: 0.90em; background-color: #53234f;" class="badge badge-primary">AÑADIR CONTENIDO</span><br/><br/>
                </div>
            </a>
            <a th:href="@{/images/Peliculas/agregar.png}"></a>
        </div>
    </div>
</div>
```

Cuando cliquemos nos enviará a la ruta `=@{'/api/peliculas/añadirPelícula'}` que nos enviará al panel de agregar contenido.

Métodos POST

Método `createPelícula()`: Este método está anotado con `@PostMapping("/crear")`, lo que significa que maneja las solicitudes POST a la ruta /api/peliculas/crear. Este método crea una nueva película en la base de datos. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad de película.

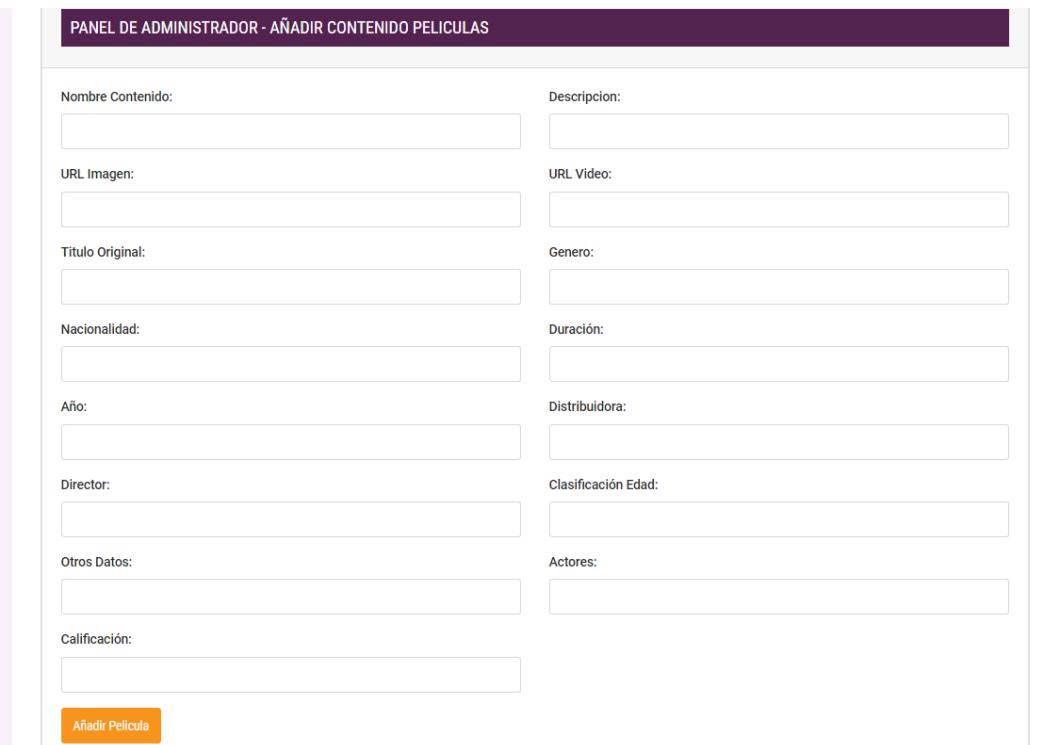
```
@PostMapping("/crear")
```

```

public String createPelícula(
    @RequestParam String nombreContenidoP,
    @RequestParam String descripciónP,
    @RequestParam String url_imageP,
    @RequestParam String url_videoP,
    @RequestParam String títuloOriginal,
    @RequestParam String género,
    @RequestParam String país,
    @RequestParam Integer duraciónP,
    @RequestParam Integer añoP,
    @RequestParam String distribuidora,
    @RequestParam String director,
    @RequestParam Short clasificaciónEdad,
    @RequestParam String otrosDatosP,
    @RequestParam String actores,
    @RequestParam double calificación,
    HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        Película películaNew = new Película();
        películaNew.setNombreContenido(nombreContenidoP);
        películaNew.setDescripción(descripciónP);
        películaNew.setUrl_image(url_imageP);
        películaNew.setUrl_video(url_videoP);
        películaNew.setTítuloOriginal(títuloOriginal);
        películaNew.setGénero(género);
        películaNew.setNacionalidad(país);
        películaNew.setDuración(duraciónP);
        películaNew.setAño(añoP);
        películaNew.setDistribuidora(distribuidora);
        películaNew.setDirector(director);
        películaNew.setClasificaciónEdad(clasificaciónEdad);
        películaNew.setOtrosDatos(otrosDatosP);
        películaNew.setActores(actores);
        películaNew.setCalificación(calificación);
        películaService.createPelícula(películaNew);
        redirectAttributes.addFlashAttribute("successMessage", "Película creada con éxito");
        return "redirect:/api/películas/"; // Redirect to the main movie page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al crear la película");
        return "redirect:/api/películas/añadirPelícula"; // Redirect back to the add movie page
    }
}

```

Una vez que dentro del panel de index se selecciona el contenido propio, si se trata de un usuario administrador va a poder añadir nuevo contenido. Este método va relacionado con el de añadirPelícula.



PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO PELICULAS	
Nombre Contenido:	Descripción:
URL Imagen:	URL Video:
Titulo Original:	Genero:
Nacionalidad:	Duración:
Año:	Distribuidora:
Director:	Clasificación Edad:
Otros Datos:	Actores:
Calificación:	
Añadir Película	

Figura 18: Panel que implementa el método createPelícula

Método updatePelícula(): anotado con `@PostMapping("/update/{nombrePelícula}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/películas/update/{nombrePelícula}`. Este método actualiza los detalles de una película específica en la base de datos. Recogiendo todos los parámetros de la película de la vista y comprobando en todo momento que se trata de un usuario administrador.

```
@PostMapping("/update/{nombrePelícula}")
public String updatePelícula(@PathVariable String nombrePelícula,
                             @RequestParam String nombreContenido,
                             @RequestParam String descripción,
                             @RequestParam String url_image,
                             @RequestParam String url_video,
                             @RequestParam String títuloOriginal,
                             @RequestParam String género,
                             @RequestParam String país,
                             @RequestParam Integer duración,
                             @RequestParam Integer año,
                             @RequestParam String distribuidora,
                             @RequestParam String director,
                             @RequestParam Short clasificaciónEdad,
                             @RequestParam String otrosDatos,
                             @RequestParam String actores,
                             @RequestParam double calificación,
                             HttpSession session, RedirectAttributes redirectAttributes ) {
```

```
User user = (User) session.getAttribute("user");
if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
    return "login"; // Redirect the user to the login page if not authenticated or not an admin
}
try {
    Pelicula detallesPelicula = new Pelicula();
    detallesPelicula.setNombreContenido(nombreContenido);
    detallesPelicula.setDescripcion(descripcion);
    detallesPelicula.setUrl_image(url_image);
    detallesPelicula.setUrl_video(url_video);
    detallesPelicula.setTituloOriginal(tituloOriginal);
    detallesPelicula.setGenero(genero);
    detallesPelicula.setNacionalidad(pais);
    detallesPelicula.setDuracion(duracion);
    detallesPelicula.setAnho(anho);
    detallesPelicula.setDistribuidora(distribuidora);
    detallesPelicula.setDirector(director);
    detallesPelicula.setClasificacionEdad(clasificacionEdad);
    detallesPelicula.setOtrosDatos(otrosDatos);
    detallesPelicula.setActores(actores);
    detallesPelicula.setCalificacion(calificacion);
    peliculaService.updatePelicula(nombrePelicula, detallesPelicula);
    redirectAttributes.addFlashAttribute("successMessage", "Película actualizada con éxito");
    return "redirect:/api/peliculas/" + nombreContenido; // Redirect to the updated movie page
} catch (RuntimeException e) {
    redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar la película");
    return "redirect:/api/peliculas/" + nombreContenido; // Redirect to the updated movie page
}
}
```

Dentro de indexDetallado si accede un administrador a ver el contenido va a tener la opción de poder actualizarlo.

VOLVER AL MENU PRINCIPAL

PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO

Nombre Contenido:	Descripción:
El Padrino	Un poderoso drama criminal que sigue la vida de la familia Corleone en el mu
URL Imagen:	URL Video:
Images/Películas/ElPadrino.jpeg	https://www.youtube.com/embed/oYQx7MXaz0?si=nuCJfNhIJoPzuQTb
Título Original:	Genero:
Il Padrino	Crimen, Drama
Nacionalidad:	Duración:
Estados Unidos	177
Año:	Distribuidora:
1972	Paramount Pictures
Director:	Clasificación Edad:
Francis Ford Coppola	18
Otros Datos:	Actores:
Ganadora de 3 premios Oscar.	Marlon Brando, Al Pacino, James Caan
Calificación:	
7,8	
Actualizar Película	

Figura 19: Panel que implementa el método updatePelícula.

```
<div th:if="#{lists.contains(roles, 'ADMIN')}" class="container mt-5">
    <!-- Contenido visible solo para administradores -->
    <div class="card">
        <div class="card-header">
            <h3 class="titulo-cine">PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO</h3>
        </div>
        <div class="card-body">
            <form th:action="@{'/api/peliculas/update/' + ${pelicula.nombreContenido}}"
method="post"
                class="row g-3">
                <div class="col-md-6 mb-3">
                    <label for="nombreContenido" class="form-label">Nombre Contenido:</label>
                    <input type="text" id="nombreContenido" name="nombreContenido"
                           th:value="${pelicula.nombreContenido}" required class="form-control">
                </div>
                <div class="col-md-6 mb-3">
                    <label for="descripcion" class="form-label">Descripción:</label>
                    <input type="text" id="descripcion" name="descripcion"
                           th:value="${pelicula.descripcion}" required class="form-control">
                </div>
                <div class="col-md-6 mb-3">
                    <label for="url_image" class="form-label">URL Imagen:</label>
                    <input type="text" id="url_image" name="url_image"
                           th:value="${pelicula.url_image}" required class="form-control">
                </div>
                <div class="col-md-6 mb-3">
                    <label for="url_video" class="form-label">URL Video:</label>
                    <input type="text" id="url_video" name="url_video"
                           th:value="${pelicula.url_video}" required class="form-control">
                </div>
            </form>
        </div>
    </div>

```

```

        required class="form-control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="tituloOriginal" class="form-label">Titulo Original:</label>
        <input type="text" id="tituloOriginal" name="tituloOriginal"
            th:value="${pelicula.tituloOriginal}" required class="form-
control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="genero" class="form-label">Genero:</label>
        <input type="text" id="genero" name="genero" th:value="${pelicula.genero}"
required
            class="form-control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="pais" class="form-label">Nacionalidad:</label>
        <input type="text" id="pais" name="pais"
            th:value="${pelicula.nacionalidad}" required
            class="form-control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="duracion" class="form-label">Duración:</label>
        <input type="number" id="duracion" name="duracion"
            th:value="${pelicula.duracion}" required
            class="form-control">
    </div>

    <div class="col-md-6 mb-3">
        <label for="anho" class="form-label">Año:</label>
        <input type="number" id="anho" name="anho" th:value="${pelicula.anho}"
required
            class="form-control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="distribuidora" class="form-label">Distribuidora:</label>
        <input type="text" id="distribuidora" name="distribuidora"
            th:value="${pelicula.distribuidora}" required class="form-control">
    </div>

    <div class="col-md-6 mb-3">
        <label for="director" class="form-label">Director:</label>
        <input type="text" id="director" name="director"
            th:value="${pelicula.director}" required
            class="form-control">
    </div>
    <div class="col-md-6 mb-3">
        <label for="clasificacionEdad" class="form-label">Clasificación
Edad:</label>
    </div>

```

```

<input type="number" id="clasificacionEdad" name="clasificacionEdad"
       th:value="${pelicula.clasificacionEdad}" required class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="otrosDatos" class="form-label">Otros Datos:</label>
    <input type="text" id="otrosDatos" name="otrosDatos"
           th:value="${pelicula.otrosDatos}"
           required class="form-control">
</div>

<div class="col-md-6 mb-3">
    <label for="actores" class="form-label">Actores:</label>
    <input type="text" id="actores" name="actores"
           th:value="${pelicula.actores}"
           required class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="calificacion" class="form-label">Calificación:</label>
    <input type="number" id="calificacion" name="calificacion" step="0.1"
           th:value="${pelicula.calificacion}" required class="form-control">
</div>
<div class="col-12">
    <button type="submit" class="btn btn-primary">Actualizar Película</button>
</div>
</form>
</div>
</div>

```

Con el uso de thymeleaf [2] [6] para mostrar los datos y la petición al controlador actualizamos el contenido.

```

<form th:action="@{'/api/peliculas/update/' + ${pelicula.nombreContenido}}" method="post"
      class="row g-3">

```

Método **deletePelícula(...)**: anotado con `@PostMapping("/delete/{nombrePelícula}")`, lo que significa que maneja las solicitudes POST a la ruta /api/peliculas/delete/{nombrePelícula}. Este método elimina una película específica de la base de datos. Comprueba que se trata de un usuario administrador, también para la incorporación de seguridad el administrador introduce su contraseña y la confirma en caso de no ser validas no se podrá borrar el contenido en este caso una película.

```

@PostMapping("/delete/{nombrePelícula}")
public String deletePelícula(@PathVariable String nombrePelícula,
                           @RequestParam String password,
                           @RequestParam String confirmPassword,
                           HttpSession session, RedirectAttributes redirectAttributes) {

```

```

User user = (User) session.getAttribute("user");
if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
    return "login"; // Redirect the user to the login page if not authenticated or not an admin
}

if (!password.equals(user.getContrasenha())) {
    redirectAttributes.addFlashAttribute("errorMessage", "La contraseña no coincide con la del usuario actual");
    return "redirect:/api/peliculas/" + nombrePelicula; // Redirect back to the movie page
}

if (!password.equals(confirmPassword)) {
    redirectAttributes.addFlashAttribute("errorMessage", "Las contraseñas no coinciden");
    return "redirect:/api/peliculas/" + nombrePelicula; // Redirect back to the movie page
}

try {
    // Delete the comments of the movie
    comentarioPeliculaService.deleteComentariosByPelicula(nombrePelicula);
    // Delete the movie
    peliculaService.deletePelicula(nombrePelicula);
    redirectAttributes.addFlashAttribute("successMessage", "Película borrada con éxito");
    return "redirect:/api/peliculas/"; // Redirect to the main movie page
} catch (RuntimeException e) {
    redirectAttributes.addFlashAttribute("errorMessage", "Error al borrar la película");
    return "redirect:/api/peliculas/" + nombrePelicula; // Redirect back to the movie page
}
}
}

```



PANEL DE ADMINISTRADOR - BORRAR CONTENIDO

Contraseña:

Confirmar Contraseña:

Borrar Película

Figura 20: Panel que implementa el método deletePelicula

```

<div th:if="#{lists.contains(roles, 'ADMIN')}" class="container mt-5">
    <!-- Contenido visible solo para administradores -->
    <div class="card">
        <div class="card-header">
            <h3 class="titulo-cine">PANEL DE ADMINISTRADOR - BORRAR CONTENIDO</h3>

```

```

        </div>
        <div class="card-body">
            <form th:action="@{/api/peliculas/delete/' + ${pelicula.nombreContenido}))"
method="post"
                class="row g-3">
                <div class="col-12 mb-3">
                    <label for="password" class="form-label">Contraseña:</label>
                    <input type="password" id="password" name="password" required class="form-control">
                </div>
                <div class="col-12 mb-3">
                    <label for="confirmPassword" class="form-label">Confirmar
                    Contraseña:</label>
                    <input type="password" id="confirmPassword" name="confirmPassword"
required
                        class="form-control">
                </div>
                <div class="col-12">
                    <button type="submit" class="btn btn-danger">Borrar Película</button>
                </div>
            </form>
        </div>
    </div>

```

De nuevo con el uso de thymeleaf [2] [6] y con los dos atributos de la contraseña, hacemos la comprobación en el controlador y enviamos el mensaje de error o de acción realizada correctamente.

Método `realtimeSearch(@RequestParam String query)`: Este método está anotado con `@GetMapping("/search/realtime")`, lo que significa que maneja las solicitudes GET a la ruta `/api/peliculas/search/realtime`. Este método realiza una búsqueda en tiempo real de películas cuyos nombres comienzan con la cadena de consulta proporcionada. Es usado en el buscador que incorpora la aplicación para poder encontrar contenidos cuando hay mucha variedad.

```

@GetMapping("/search/realtime")
@ResponseBody
public List<Película> realtimeSearch(@RequestParam String query) {
    return peliculaService.searchPelículas(query);
}
}

```

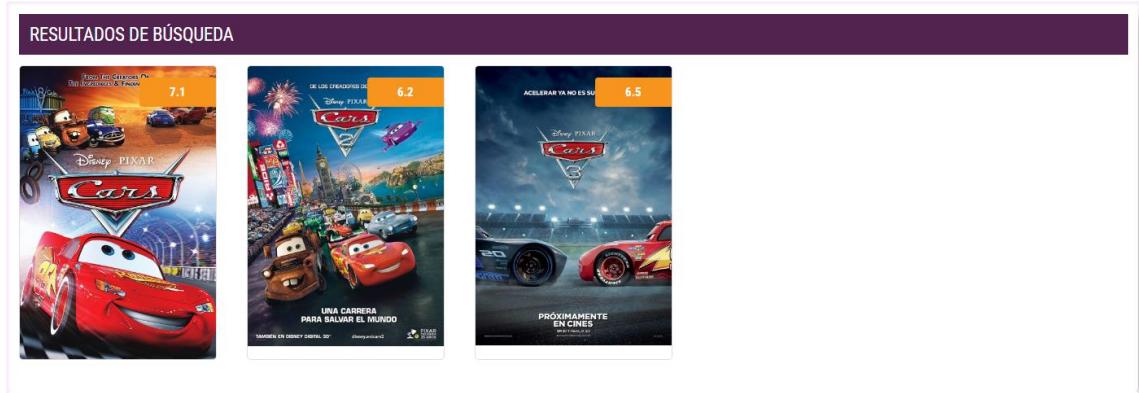


Figura 21: Panel que implementa el método `realtimeSearch` para Películas.

Dentro de la vista añadimos un nuevo campo de buscador que con Ajax [17] iremos completando según se vaya escribiendo contenido.

```
<!-- Resultados de búsqueda -->
<div id="search-results"></div>
<script>
$(document).ready(function () {
    $('#search-input').on('input', function () {
        var query = $(this).val();
        var url = query ? '/api/peliculas/search/realtime' : '/api/peliculas/';
        $.ajax({
            url: url,
            data: {
                query: query
            },
            success: function (data) {
                // Vacía el contenedor de resultados de búsqueda
                $('#search-results').empty();
                // Crea un nuevo elemento de película para cada resultado de la búsqueda
                var peliculasElement = `
                    <div class="white-box">
                        <h3 class="titulo-cine">RESULTADOS DE BÚSQUEDA</h3>
                        <div class="row">
                `;
                $.each(data, function (index, pelicula) {
                    peliculasElement += `
                        <div class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
                            <div class="card card-event info-overlay">
                                <h1 class="tag">${pelicula.calificacion}</h1>
                                <div class="img has-background">
                                    <a href="/api/peliculas/${pelicula.nombreContenido}" class="event-pop-link">
                                        <div class="event-pop-info">
                                            <p class="price" style="font-size:1.1em; padding: 5px;">${pelicula.nombreContenido}</p>
                                            <span style="font-size: 0.90em; background-color: #53234f;" class="badge badge-primary">VER CONTENIDO</span><br/><br/>
                                        </div>
                                    </a>
                                </div>
                            </div>
                        </div>
                `;
            }
        });
    });
});
```

```

        </div>
    </a>
    <a href="/api/peliculas/${pelicula.nombreContenido}"></a>
        </div>
    </div>
</div>
`;
});
peliculasElement += `

</div>
</div>
`;
// Agrega todos los elementos de película al contenedor de resultados de búsqueda
$('#search-results').append(peliculasElement);
}
});
});
});
});
</script>
```

Cuando se escribe texto en el buscador empieza la búsqueda y se recogen los contenidos que empiezan por ese nombre por parte del controlador.

```
var url = query ? '/api/peliculas/search/realtme' : '/api/peliculas/';
```

Si no hay nada en el buscador devuelve el primero método que simplemente muestra todas las películas. Y añade al campo de peliculaElement las películas que le pasa el controlador.

Controlador/Vista Contenido de F1

La clase F1ContentController se encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para la sección de carreras de F1, que en la parte de la vista tendremos un listado de todas las carreras con un buscador integrado y luego cuando el usuario seleccione la carrera que quiere ver dentro de esta podrá ver datos y ver su contenido.

Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/controller/ F1ContentController.java.

La clase indexF1 es la encargada de la vista por parte de los contenidos de F1 donde se ven todas las carreras disponibles en la plataforma y indexDetalladoF1 es para cuando se selecciona un contenido en indexF1, verlo y ver su información aparte también se incluye la sección de los comentarios

Estas clases se encuentran en el archivo:

```
src/main/resources/templates/indexF1.html  
src/main/resources/templates/indexDetalladoF1.html
```

Para las clases de contenido que son las de películas, carreras de F1 y partidos de fútbol, la metodología es igual en las 3 es el mismo código y solo cambia la entidad con la que se trata, como en películas ya se ha puesto el código detallado al completo sobre todo el de la vista que ocupa mucha y es lo mismo, pero con los campos de cada entidad, se va a omitir el código de la vista en los contenidos de f1 y fútbol, pero si se van a explicar.

Métodos GET

Definimos la ruta del controlador

```
@RequestMapping("/api/F1")
```

Método **getAllF1Content (Model model, HttpSession session)**: Este método está anotado con `@GetMapping("/")`, lo que significa que maneja las solicitudes GET a la ruta `/api/F1/`. Este método recupera todas las carreras de F1 de la base de datos y las añade al modelo para ser mostradas en la vista. También le pasa los roles a vista ya que dentro del panel de `indexF1` si eres administrador se incluye la opción de añadir nuevo contenido.

```
@GetMapping("/")  
public String getAllF1Content(Model model, HttpSession session) {  
    List<F1Content> listaf1 = f1ContentService.getAllF1Content();  
    model.addAttribute("listaf1", listaf1);  
    User user = (User) session.getAttribute("user");  
    if (user != null) {  
        Set<Role> roles = user.getRoles();  
        model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.toList()));  
    }  
    return "indexF1";  
}
```

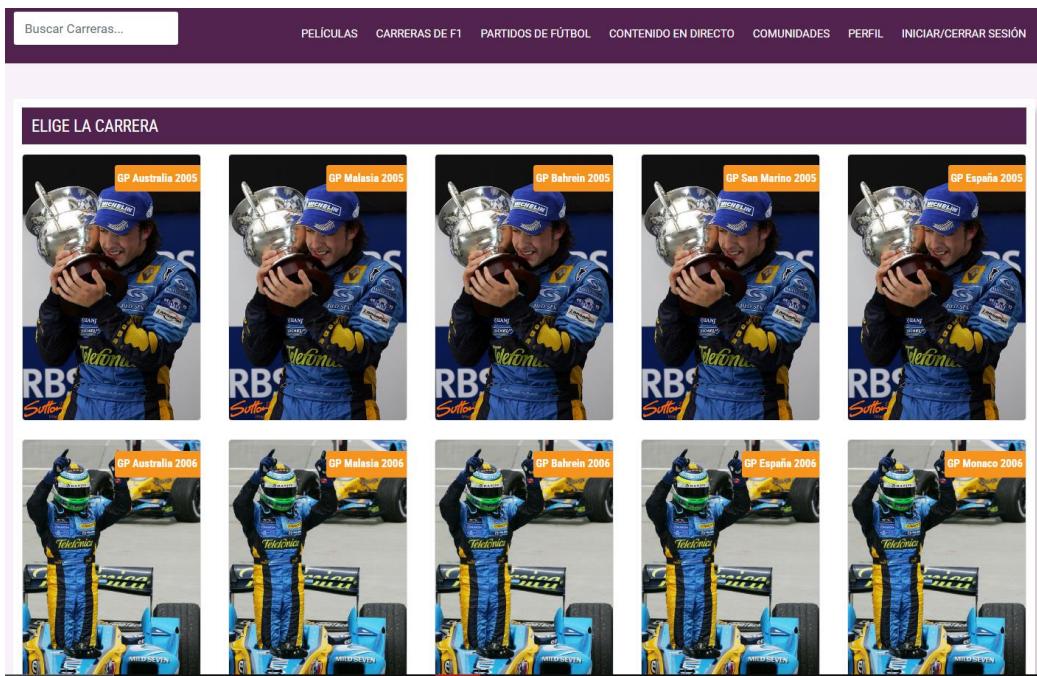


Figura 22: Panel que implementa el método `getAllF1Content`.

Método `getNombreCarreraF1(@PathVariable String nombreCarreraF1, Model model, HttpSession session)`: Este método está anotado con `@GetMapping("/ {nombreCarreraF1}")`, lo que significa que maneja las solicitudes GET a la ruta `/api/F1/{nombreCarreraF1}`. Este método recupera un contenido de F1 específico de la base de datos por su nombre y la añade al modelo para ser mostrada en la vista. El cual se va a usar cuando en la vista el usuario seleccione una carrera de F1 concreta para ver. Se comparte la sesión del usuario, su rol y el contenido exclusivo que se quiere ver.

Figura 23: Panel que implementa el método `getNombreCarreraF1`.

Método `añadirCarrera (HttpSession session)` anotado con `@GetMapping("/añadirCarrera")`, lo que significa que maneja las solicitudes GET a la ruta `/api/F1/ añadirCarrera`. Este método verifica si el usuario

actual está autenticado y es un administrador. Si es así, redirige al usuario a la página de agregar contenido. Si no, redirige al usuario a la página de inicio de sesión.

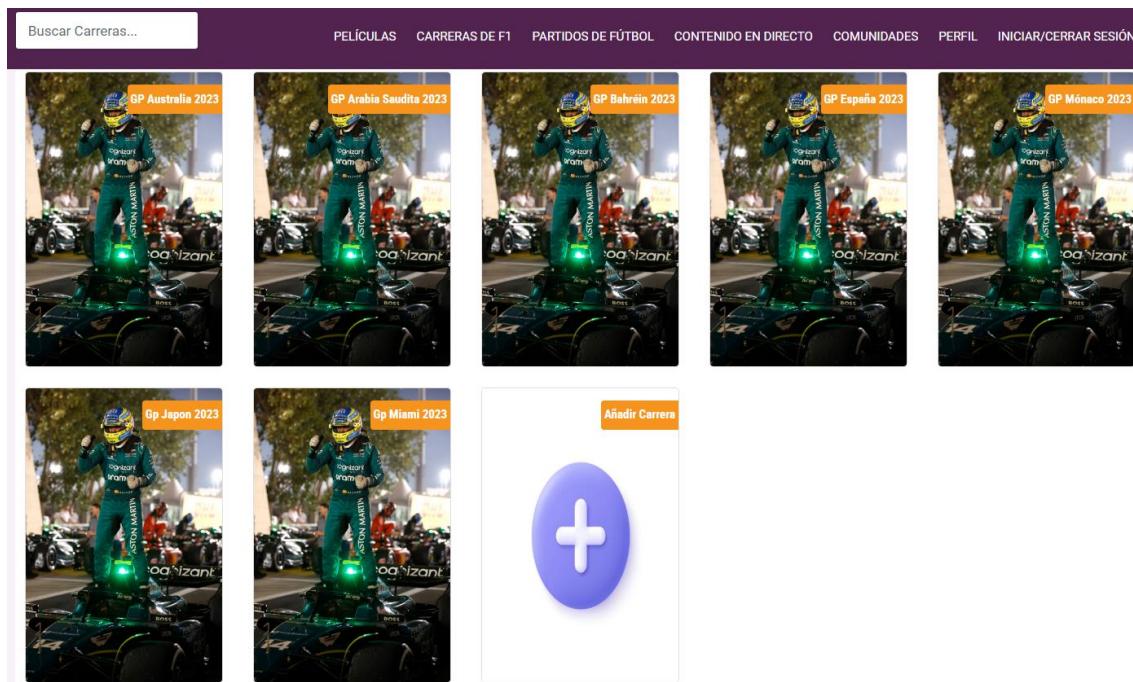


Figura 24: Panel para implementar el método `añadirCarrera`.

MÉTODOS POST

Método `createF1Content()`: Este método está anotado con `@PostMapping("/create")`, lo que significa que maneja las solicitudes POST a la ruta `/api/f1/create`. Este método crea una nueva carrera de F1 en la base de datos. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad de película.

```
@PostMapping("/create")
public String createF1Content(
    @RequestParam String nombreContenidoF1,
    @RequestParam String descripcionF1,
    @RequestParam String url_imageF1,
    @RequestParam String url_videoF1,
    @RequestParam Integer anhoF1,
    @RequestParam String circuitoF1,
    @RequestParam String equiposF1,
    @RequestParam String nacionalidadF1,
    @RequestParam Integer duracionF1,
    @RequestParam String pilotos,
    @RequestParam String otrosDatosF1,
    HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
}
```

```

    }

    try {
        F1Content f1New = new F1Content();
        f1New.setNombreContenido(nombreContenidoF1);
        f1New.setDescripcion(descripcionF1);
        f1New.setUrl_image(url_imageF1);
        f1New.setUrl_video(url_videoF1);
        f1New.setAnho(anhoF1);
        f1New.setCircuito(circuitoF1);
        f1New.setEquipos(equiposF1);
        f1New.setNacionalidad(nacionalidadF1);
        f1New.setDuracion(duracionF1);
        f1New.setPilotos(pilotos);
        f1New.setOtrosDatos(otrosDatosF1);
        f1ContentService.createF1Content(f1New);
        redirectAttributes.addFlashAttribute("successMessage", "Carrera creada con éxito");
        return "redirect:/api/F1/"; // Redirect to the main F1 content page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al crear la carrera");
        return "redirect:/api/F1/añadirCarrera"; // Redirect back to the add F1 content page
    }
}
}

```

PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO F1

<div style="margin-bottom: 10px;"> Nombre Contenido: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> URL Imagen: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Año: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Equipos: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Duración: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Otros Datos: <input style="width: 100%; height: 25px;" type="text"/> </div>	<div> Descripción: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> URL Video: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Circuito: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Nacionalidad: <input style="width: 100%; height: 25px;" type="text"/> </div> <div> Pilotos: <input style="width: 100%; height: 25px;" type="text"/> </div>
<input style="background-color: #f0a000; color: white; border: none; padding: 5px; width: auto;" type="button" value="Añadir Carrera"/>	

Figura 25: Panel para implementar el método createF1Content

Se envía la solicitud al controlador con todos los campos de la nueva carrera de f1 y si la acción se hace correctamente devuelve el mensaje de éxito y si no devuelve el mensaje de error. Destacar que es una operación solo disponible para administradores.

Método **deleteCarreraF1(...)**: anotado con `@PostMapping("/delete/{nombreCarreraF1}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/F1/delete/{nombreCarreraF1}`. Este método elimina una carrera específica de la base de datos. Comprueba que se trata de un usuario administrador, también para la incorporación de seguridad el administrador introduce su contraseña y la confirma en caso de no ser validas no se podrá borrar el contenido en este caso una carrera de F1.

```
@PostMapping("/delete/{nombreCarreraF1}")
public String deletePelícula(@PathVariable String nombreCarreraF1,
                             @RequestParam String password,
                             @RequestParam String confirmPassword,
                             HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    if (!password.equals(user.getPassword())) {
        redirectAttributes.addFlashAttribute("errorMessage", "La contraseña no coincide con la del usuario actual");
        return "redirect:/api/F1/" + nombreCarreraF1; // Redirect back to the movie page
    }
    if (!password.equals(confirmPassword)) {
        redirectAttributes.addFlashAttribute("errorMessage", "Las contraseñas no coinciden");
        return "redirect:/api/F1/" + nombreCarreraF1; // Redirect back to the movie page
    }
    try {
        // Delete the comments of the movie
        comentarioF1Service.deleteComentariosByF1Content(nombreCarreraF1);
        // Delete the movie
        f1ContentService.deleteF1Content(nombreCarreraF1);
        redirectAttributes.addFlashAttribute("successMessage", "Carrera borrada con éxito");
        return "redirect:/api/F1/"; // Redirect to the main movie page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al borrar la carrera");
        return "redirect:/api/F1/" + nombreCarreraF1; // Redirect back to the movie page
    }
}
```

PANEL DE ADMINISTRADOR - BORRAR CONTENIDO

Contraseña:

Confirmar Contraseña:

Figura 26: Panel para implementar el método `deleteCarreraF1`.

Método **updateF1Content ()**: anotado con `@PostMapping("/update/{nombreCarreraF1}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/peliculas/update/ { nombreCarreraF1}`. Este método actualiza los detalles de una **carrera de F1** específica en la base de datos. Recogiendo todos los parámetros de la **carrera** de la vista que introduzca el administrador y comprobando en todo momento que se trata de un usuario administrador.

```

@PostMapping("/update/{nombreCarreraF1}")
public String updateF1Content(@PathVariable String nombreCarreraF1,
                             @RequestParam String nombreContenido,
                             @RequestParam String descripcion,
                             @RequestParam String url_image,
                             @RequestParam String url_video,
                             @RequestParam Integer anho,
                             @RequestParam String circuito,
                             @RequestParam String equipos,
                             @RequestParam String nacionalidad,
                             @RequestParam Integer duracion,
                             @RequestParam String pilotos,
                             @RequestParam String otrosDatos,
                             HttpSession session, RedirectAttributes redirectAttributes ) {

    User user = (User) session.getAttribute("user");
    if (user == null) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        F1Content detallesF1Content = new F1Content();
        detallesF1Content.setNombreContenido(nombreContenido);
        detallesF1Content.setDescripcion(descripcion);
        detallesF1Content.setUrl_image(url_image);
        detallesF1Content.setUrl_video(url_video);
        detallesF1Content.setAnho(anho);
        detallesF1Content.setCircuito(circuito);
        detallesF1Content.setEquipos(equipos);
        detallesF1Content.setNacionalidad(nacionalidad);
        detallesF1Content.setDuracion(duracion);
        detallesF1Content.setPilotos(pilotos);
        detallesF1Content.setOtrosDatos(otrosDatos);
        f1ContentService.updateF1Content(nombreCarreraF1, detallesF1Content);
        redirectAttributes.addFlashAttribute("successMessage", "Carrera actualizada con éxito");
        return "redirect:/api/F1/" + nombreContenido; // Redirect to the updated F1 content page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar la carrera");
        return "redirect:/api/F1/" + nombreContenido; // Redirect to the updated F1 content page
    }
}

```

PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO

Nombre Contenido:	Descripción:
GP Australia 2005	El Gran Premio de Australia de 2005 marcó el inicio de la temporada de Fórmula 1.
URL Imagen:	URL Video:
images/f1/2005.png	https://www.youtube.com/embed/zscZdvxOQ4A?si=19JozFQrKmGzoTW1
Año:	Circuito:
2005	Melbourne Grand Prix Circuit
Equipos:	Nacionalidad:
Renault, Ferrari, McLaren, Williams, Toyota, BAR, Red Bull, Sauber, Jordan, Minardi	Australia
Duración:	Pilotos:
58	Alonso, Fisichella, Räikkönen, Schumacher, Barrichello, Montoya, Trulli, Webber
Otros Datos:	
1. Alonso, 2. Fisichella, 3. Räikkönen	
Actualizar Carrera	

Figura 27: Panel para implementar el método updateF1Content.

Siempre van a salir los datos que están actualmente en la base de datos y el administrador va a poder modificar los que quiera y enviar la solicitud al controlador.

Método realtimeSearch (@RequestParam String query): Este método está anotado con @GetMapping("/search/realtime"), lo que significa que maneja las solicitudes GET a la ruta /api/películas/search/realtime. Este método realiza una búsqueda en tiempo real de carreras de F1 cuyos nombres comienzan con la cadena de consulta proporcionada. Es usado en el buscador que incorpora la aplicación para poder encontrar contenidos cuando hay mucha variedad.

```
@GetMapping("/search/realtime")
@ResponseBody
public List<F1Content> realtimeSearch(@RequestParam String query) {
    return f1ContentService.searchF1Content(query);
}
```

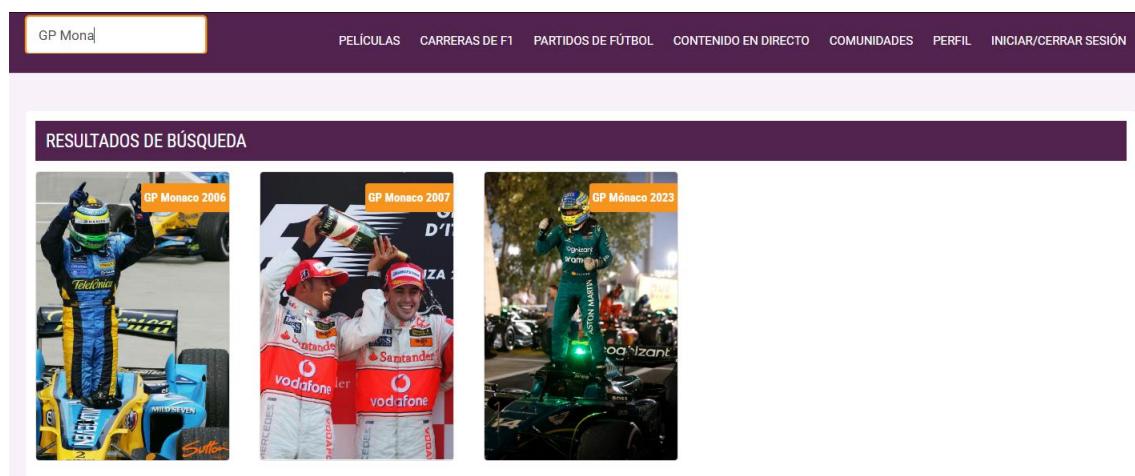


Figura 28: Panel para implementar el método realtimeSearch.

Dentro de la vista añadimos un nuevo campo de buscador que con Ajax [17] iremos completando según se vaya escribiendo contenido.

```
<!-- Resultados de búsqueda -->
<div id="search-results"></div>
<script>
$(document).ready(function () {
    $('#search-input').on('input', function () {
        var query = $(this).val();
        var url = query ? '/api/F1/search/realtim' : '/api/F1/';
        $.ajax({
            url: url,
            data: {
                query: query
            },
            success: function (data) {
                // Vacía el contenedor de resultados de búsqueda
                $('#search-results').empty();

                // Crea un nuevo elemento de película para cada resultado de la búsqueda
                var F1Element = `

                    <div class="white-box">
                        <h3 class="titulo-cine">RESULTADOS DE BÚSQUEDA</h3>
                        <div class="row">
                    `;

                $.each(data, function (index, f1) {
                    F1Element += `

                        <div class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
                            <div class="card card-event info-overlay">
                                <h1 class="tag">${f1.nombreContenido}</h1>
                                <div class="img has-background">
                                    <a href="/api/F1/${f1.nombreContenido}" class="event-pop-link">
                                        <div class="event-pop-info">
                                            <p class="price" style="font-size:1.1em; padding: 5px;">${f1.nombreContenido}</p>
                                            <span style="font-size: 0.90em; background-color: #53234f; color: white; padding: 2px 5px;">VER CONTENIDO</span><br/><br/>
                                        </div>
                                    </a>
                                    <a href="/api/F1/${f1.nombreContenido}"></a>
                                </div>
                            </div>
                        `;
                });
                F1Element += `

                </div>
            `;
        });
    });
});
```

```

        `;
        // Agrega todos los elementos de película al contenedor de resultados de búsqueda
        $('#search-results').append(F1Element);
    }
});
});
});
});

```

Cuando se escribe texto en el buscador empieza la búsqueda y se recogen los contenidos que empiezan por ese nombre por parte del controlador.

```
var url = query ? '/api/F1/search/realtime' : '/api/F1/' ;
```

Si no hay nada en el buscador devuelve el primero método que simplemente muestra todas las carreras de F1. En el caso de que el usuario empiece a buscar contenidos en el buscador añade al campo de F1Element las carreras de F1 que le pasa el controlador.

Controlador/Vista Contenido de Fútbol

La clase FootballContentController se encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para la sección de partidos de fútbol, que en la parte de la vista tendremos un listado de todos los partidos con un buscador integrado y luego cuando el usuario seleccione el partido que quiere ver dentro de esta podrá ver datos y ver su contenido.

Esta clase se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/controller/ FootballContentController.java.
```

La clase indexFootball es la encargada de la vista por parte de los contenidos de fútbol donde se ven todos los partidos disponibles en la plataforma y indexDetalladoFootbl es para cuando se selecciona un contenido en indexFootball, verlo y ver su información aparte también se incluye la sección de los comentarios.

Estas clases se encuentran en el archivo:

```
src/main/resources/templates/indexFootabll.html
```

```
src/main/resources/templates/indexDetalladoFotball.html
```

Para las clases de contenido que son las de películas, carreras de F1 y partidos de fútbol, la metodología es igual en las 3 es el mismo código y solo cambia la entidad con la que se trata, como en películas ya se ha puesto el código detallado al completo sobre todo el de la vista que ocupa mucha y es lo mismo, pero con los campos de cada entidad, se va a omitir el código de la vista en los contenidos de f1 y fútbol, pero si se van a explicar.

Métodos GET

Definimos la ruta del controlador

```
@RequestMapping("/api/FootballController")
```

Método **getAllFootballContent (Model model, HttpSession session)**: Este método está anotado con `@GetMapping("/")`, lo que significa que maneja las solicitudes GET a la ruta `/api/FootballController/`. Este método recupera todos los partidos de fútbol de la base de datos y las añade al modelo para ser mostradas en la vista. También le pasa los roles a vista ya que dentro del panel de `indexFootball` si eres administrador se incluye la opción de añadir nuevo contenido.

```
@GetMapping("/")
public String getAllFootballContent(Model model, HttpSession session) {
    List<FootballContent> listaFootballContent = footballContentService.getAllFootballContent();
    model.addAttribute("listaFootballContent", listaFootballContent);
    User user = (User) session.getAttribute("user");
    if (user != null) {
        Set<Role> roles = user.getRoles();
        model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.toList()));
    }
    return "indexFootball"; // nombre de el archivo Thymeleaf sin la extensión .html
}
```

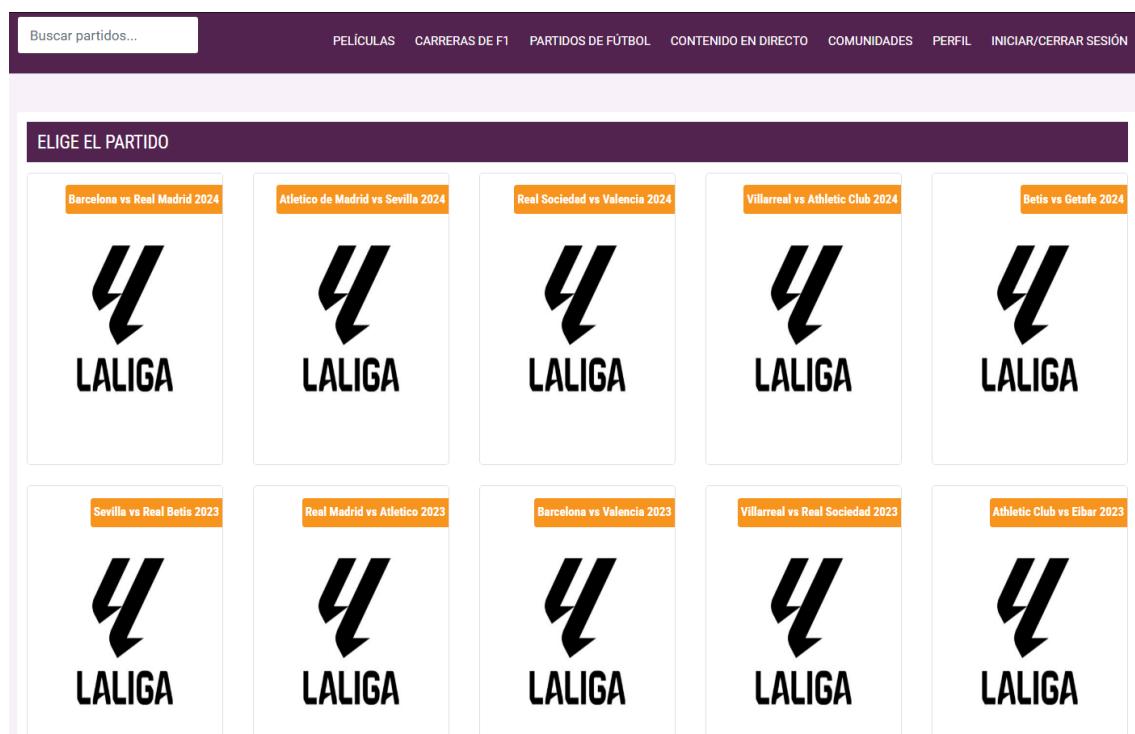


Figura 29: Panel para implementar el método `getAllFootballContent`.

Método **getNombreFootballContent (@PathVariable String nombrePartido, Model model, HttpSession session)**:

Este método está anotado con `@GetMapping("/{nombreFootballContent}")`, lo que significa que maneja las solicitudes GET a la ruta `/api/F1/{nombreFootballContent}`. Este método recupera un partido de fútbol específico de la base de datos por su nombre y la añade al modelo para ser mostrada en la vista. El cual se va a

usar cuando en la vista el usuario seleccione un partido de fútbol concreto para ver. Se comparte la sesión del usuario, su rol y el contenido exclusivo que se quiere ver.

```
@GetMapping("/{nombreFootballContent}")
public String getNombreFootballContent(@PathVariable String nombreFootballContent, Model model,
HttpSession session) {
    FootballContent footballContent =
footballContentService.getFootballContentByNombrePartido(nombreFootballContent)
.orElseThrow(() -> new RuntimeException("Contenido de Futbol not found"));
model.addAttribute("footballContent", footballContent);
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<ComentarioFootball> comentarios =
comentarioFootballService.getAllComentariosByFootballContent(footballContent);
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 "indexDetalladoFootball";
}
```

VOLVER AL MENU PRINCIPAL REGISTRARSE

REAL MADRID VS VARSOVIA 2017

Real Madrid vs Varsovia 2017





MEMORIAS DEL FÚTBOL

masterca

2016-2017

MEMORIAS

UEFA CHAMPIONS LEAGUE

SUSCRÍBETE

Año:
2017

Estadio:
Estadio Santiago Bernabéu

Competición:
Champions League

Equipos:
Real Madrid, Legia de Varsovia

Jugadores:
Cristiano Ronaldo, Gareth Bale, Vadis Odjidja-Ofoe, Keylor Navas

Otros Datos:
Fase de grupos de la Champions League
2017. Partido 3/6

Descripción:
El Real Madrid se enfrentó al Legia de Varsovia en la fase de grupos de la Champions League 2017. Fue un emocionante encuentro que terminó con la victoria del Real Madrid.

Figura 30: Panel para implementar el metodo getNombreFootballContent.

Método agregarPartido(HttpServletRequest session):

Anotado con `@GetMapping("/agregarPartido")`, lo que significa que maneja las solicitudes GET a la ruta `/api/F1/ agregarPartido`. Este método verifica si el usuario actual está autenticado y es un administrador. Si es así, redirige al usuario a la página de agregar contenido. Si no, redirige al usuario a la página de inicio de sesión.

```
@GetMapping("/agregarPartido")
public String agregarPartido(HttpServletRequest session) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    return "agregarContenido"; // nombre de el archivo Thymeleaf sin la extensión .html
}
```

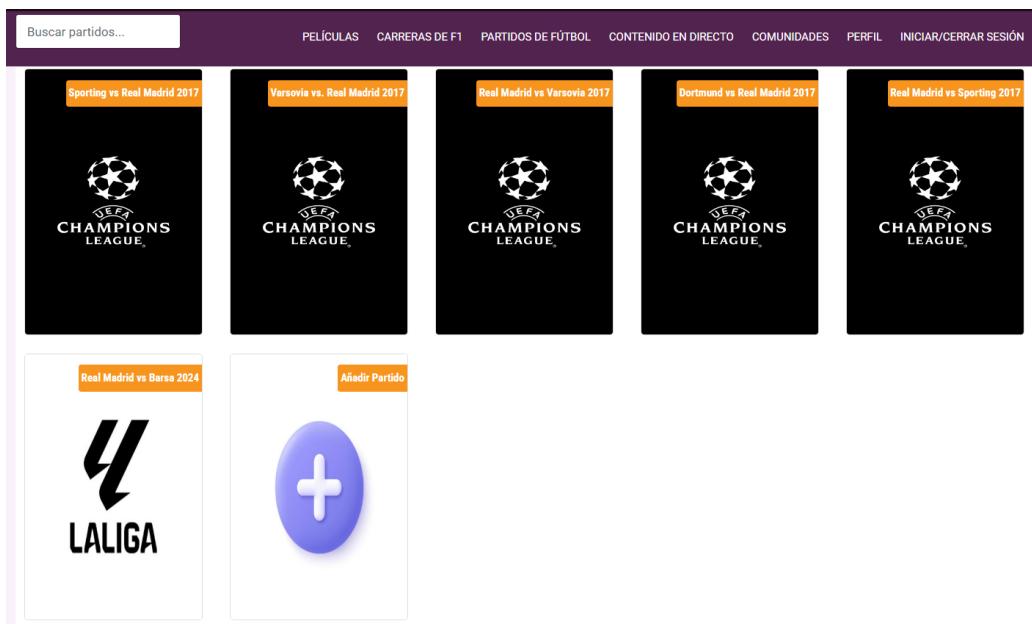


Figura 31: Panel para implementar el método `agregarPartido`.

```
<div th:if="${roles != null and #lists.contains(roles, 'ADMIN')}" class="col-6 col-sm-4 col-md-3 col-xl-2dot4
mb20" >
    <div class="card card-event info-overlay">
        <h1 class="tag">Añadir Partido</h1>
        <div class="img has-background">
            <a th:href="@{'/api/FootballContent/agregarPartido'}" class="event-pop-link">
                <div class="event-pop-info">
                    <p class="price" style="font-size:1.1em; padding: 5px;"> Pulsa para
                        agregar un
                    </p>
                    <p> nuevo contenido</p>
                    <span style="font-size: 0.90em; background-color: #53234f;">
                        <span>AÑADIR CONTENIDO</span><br/><br/>
                    </span>
                </div>
            </a>
            <a th:href="@{/images/Peliculas/agregar.png}"><img class="img-responsive" alt="Icono para agregar contenido"></a>
        </div>
    </div>

```

```

        th:src="@{/images/Peliculas
/agregar.png}"/></a>
    </div>

</div>
</div>
```

Cuando cliquemos nos enviara a la ruta = "@ {'/api/ FootballContent/agregarPartido'}" que nos enviara al panel de agregar contenido.

METODOS POST

Método **createFootballContent ()**: Este método está anotado con `@PostMapping("/create")`, lo que significa que maneja las solicitudes POST a la ruta /api/football/create. Este método crea un nuevo partido en la base de datos. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad de fútbol.

```

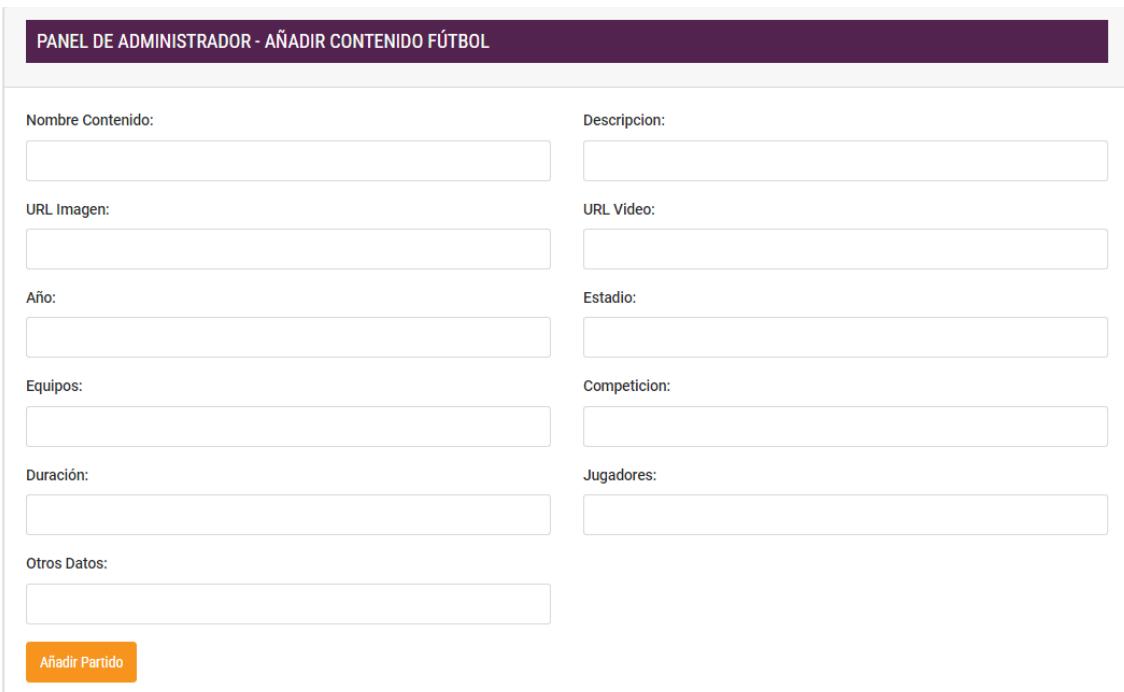
@PostMapping("/create")
public String createFootballContent(
    @RequestParam String nombreContenido,
    @RequestParam String descripcion,
    @RequestParam String url_image,
    @RequestParam String url_video,
    @RequestParam Integer anho,
    @RequestParam String estadio,
    @RequestParam String equipos,
    @RequestParam String competicion,
    @RequestParam Integer duracion,
    @RequestParam String jugadores,
    @RequestParam String otrosDatos,
    HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        FootballContent footballContentNew = new FootballContent();
        footballContentNew.setNombreContenido(nombreContenido);
        footballContentNew.setDescripcion(descripcion);
        footballContentNew.setUrl_image(url_image);
        footballContentNew.setUrl_video(url_video);
        footballContentNew.setAnho(anho);
        footballContentNew.setEstadio(estadio);
        footballContentNew.setEquipos(equipos);
    }
}
```

```

footballContentNew.setCompeticion(competicion);
footballContentNew.setDuracion(duracion);
footballContentNew.setJugadores(jugadores);
footballContentNew.setOtrosDatos(otrosDatos);
footballContentService.createFootballContent(footballContentNew);
redirectAttributes.addFlashAttribute("successMessage", "Partido creado con éxito");
return "redirect:/api/FootballContent/"; // Redirect to the main football content page
} catch (RuntimeException e) {
    redirectAttributes.addFlashAttribute("errorMessage", "Error al crear el partido");
    return "redirect:/api/FootballContent/agregarPartido"; // Redirect back to the add football
content page
}
}
}

```

Se envía la solicitud al controlador con todos los campos del nuevo partido de fútbol y si la acción se hace correctamente devuelve el mensaje de éxito y si no devuelve el mensaje de error. Destacar que es una operación solo disponible para administradores.



PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO FÚTBOL	
Nombre Contenido:	Descripcion:
<input type="text"/>	<input type="text"/>
URL Imagen:	URL Video:
<input type="text"/>	<input type="text"/>
Año:	Estadio:
<input type="text"/>	<input type="text"/>
Equipos:	Competicion:
<input type="text"/>	<input type="text"/>
Duración:	Jugadores:
<input type="text"/>	<input type="text"/>
Otros Datos:	<input type="text"/>
<input type="button" value="Añadir Partido"/>	

Figura 32: Panel para implementar el método `createFootballContent`.

Método `updateFootballContent ()`:

Anotado con `@PostMapping("/update/{nombreFootballContent}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/FootballContent/update{nombreFootballContent}`. Este método actualiza los detalles de un partido de fútbol específico en la base de datos. Recogiendo todos los parámetros del partido, ya sean jugadores, equipos, año de la vista que introduzca el administrador y comprobando en todo momento que se trata de un usuario administrador.

```

@PostMapping("/update/{nombreFootballContent}")

```

```

public String updateFootballContent(@PathVariable String nombreFootballContent,
                                    @RequestParam String nombreContenido,
                                    @RequestParam String descripcion,
                                    @RequestParam String url_image,
                                    @RequestParam String url_video,
                                    @RequestParam Integer anho,
                                    @RequestParam String estadio,
                                    @RequestParam String equipos,
                                    @RequestParam String competicion,
                                    @RequestParam Integer duracion,
                                    @RequestParam String jugadores,
                                    @RequestParam String otrosDatos,
                                    HttpSession session, RedirectAttributes redirectAttributes ) {

    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        FootballContent detallesFootballContent = new FootballContent();
        detallesFootballContent.setNombreContenido(nombreContenido);
        detallesFootballContent.setDescripcion(descripcion);
        detallesFootballContent.setUrl_image(url_image);
        detallesFootballContent.setUrl_video(url_video);
        detallesFootballContent.setAnho(anho);
        detallesFootballContent.setEstadio(estadio);
        detallesFootballContent.setEquipos(equipos);
        detallesFootballContent.setCompeticion(competicion);
        detallesFootballContent.setDuracion(duracion);
        detallesFootballContent.setJugadores(jugadores);
        detallesFootballContent.setOtrosDatos(otrosDatos);
        footballContentService.updateFootballContent(nombreFootballContent, detallesFootballContent);
        redirectAttributes.addFlashAttribute("successMessage", "Partido actualizado con éxito");
        return "redirect:/api/FootballContent/" + nombreContenido; // Redirect to the updated Football
content page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar el partido");
        return "redirect:/api/FootballContent/" + nombreContenido; // Redirect to the updated Football
content page
    }
}

```

PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO

Nombre Contenido:	Descripción:
Real Madrid vs Varsovia 2017	El Real Madrid se enfrentó al Legia de Varsovia en la fase de grupos de la Ch
URL Imagen:	URL Video:
images/FootballContent/Champions.jpg	https://www.youtube.com/embed/OnAklkKj13Q?si=OfGvlq2o-1ibPftP
Año:	Estadio:
2017	Estadio Santiago Bernabéu
Equipos:	Competición:
Real Madrid, Legia de Varsovia	Champions League
Duración:	Jugadores:
90	Cristiano Ronaldo, Gareth Bale, Vadis Odjidja-Ofoe, Keylor Navas
Otros Datos:	
Fase de grupos de la Champions League 2017. Partido 3/6	
<input style="background-color: #f08040; color: white; padding: 5px; border-radius: 5px; border: none; width: 100%;" type="button" value="Actualizar Partido"/>	

Figura 33: Panel para implementar el método updateFootballContent.

Siempre van a salir los datos que están actualmente en la base de datos y el administrador va a poder modificar los que quiera y enviar la solicitud al controlador.

Método deleteFootballContent (...):

Anotado con `@PostMapping("/delete/{nombreFootballContent}")`, lo que significa que maneja las solicitudes POST a la ruta /api/ FootballContent/delete/ {nombreFootballContent}. Este método elimina un partido de fútbol específica de la base de datos. Comprueba que se trata de un usuario administrador, también para la incorporación de seguridad el administrador introduce su contraseña y la confirma en caso de no ser validas no se podrá borrar el contenido en este caso un partido de fútbol.

```

@PostMapping("/delete/{nombreFootballContent}")
public String deleteFootballContent(@PathVariable String nombreFootballContent,
                                     @RequestParam String password,
                                     @RequestParam String confirmPassword,
                                     HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    if (!password.equals(user.getContrasenha())) {
        redirectAttributes.addFlashAttribute("errorMessage", "La contraseña no coincide con la del usuario actual");
        return "redirect:/api/FootballContent/" + nombreFootballContent; // Redirect back to the Football content page
    }
    if (!password.equals(confirmPassword)) {
        redirectAttributes.addFlashAttribute("errorMessage", "Las contraseñas no coinciden");
        return "redirect:/api/FootballContent/" + nombreFootballContent; // Redirect back to the Football content page
    }
}

```

```

    }

    try {
        // Delete the comments of the Football content
        comentarioFootballService.deleteComentariosByFootballContent(nombreFootballContent);

        // Delete the Football content
        footballContentService.deleteFootballContent(nombreFootballContent);

        redirectAttributes.addFlashAttribute("successMessage", "Contenido de Futbol borrado con éxito");
        return "redirect:/api/FootballContent/"; // Redirect to the main Football content page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al borrar el contenido de Futbol");
        return "redirect:/api/FootballContent/" + nombreFootballContent; // Redirect back to the Football
content page
    }
}

```

PANEL DE ADMINISTRADOR - BORRAR CONTENIDO

Contraseña:

Confirmar Contraseña:

Borrar Carrera

Figura 34: Panel para implementar el método deleteFootballContent.

Método realtimeSearch(@RequestParam String query): Este método está anotado con `@GetMapping("/search/realtime")`, lo que significa que maneja las solicitudes GET a la ruta /api/peliculas/search/realtime. Este método realiza una búsqueda en tiempo real de partidos de fútbol cuyos nombres comienzan con la cadena de consulta proporcionada. Es usado en el buscador que incorpora la aplicación para poder encontrar contenidos cuando hay mucha variedad.

```

@GetMapping("/search/realtime")
@ResponseBody
public List<FootballContent> realtimeSearch(@RequestParam String query) {
    return footballContentService.searchFootballContent(query);
}

```

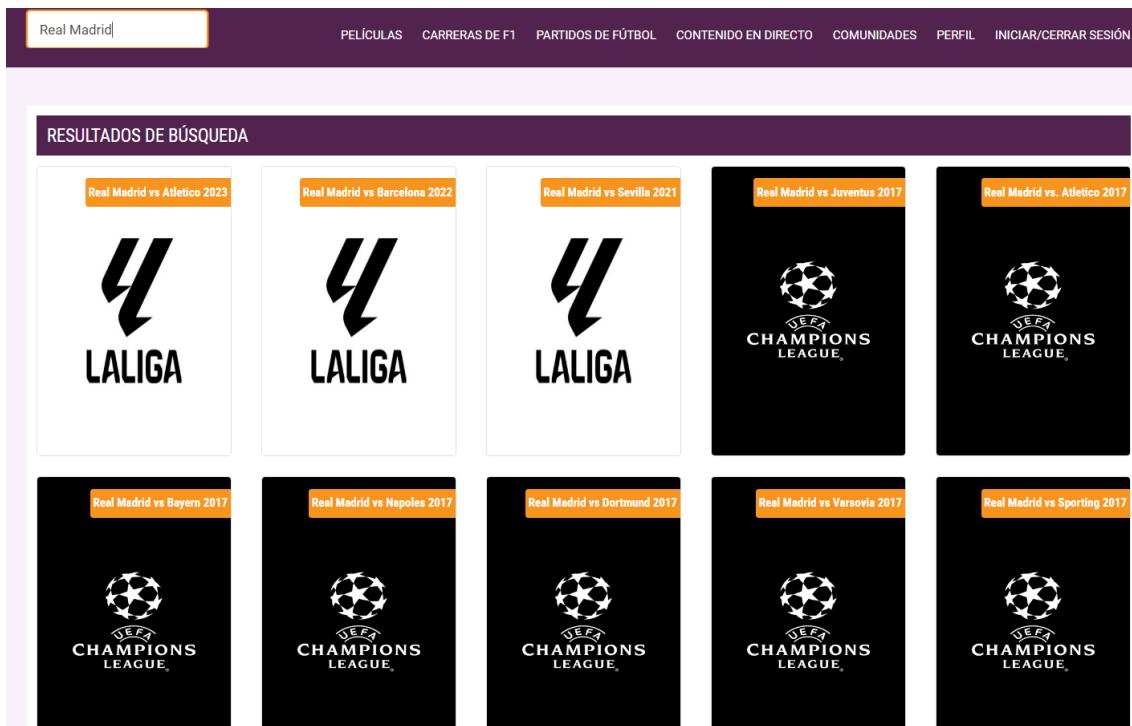


Figura 35: Panel para implementar el método `realtimeSearch` de F1.

Dentro de la vista añadimos un nuevo campo de buscador que con Ajax [17] iremos completando según se vaya escribiendo contenido.

```
<script>
$(document).ready(function () {
    $('#search-input').on('input', function () {
        var query = $(this).val();
        var url = query ? '/api/FootballContent/search/realtime' : '/api/FootballContent/';
        $.ajax({
            url: url,
            data: {
                query: query
            },
            success: function (data) {
                // Vacía el contenedor de resultados de búsqueda
                $('#search-results').empty();
                // Crea un nuevo elemento de película para cada resultado de la búsqueda
                var footballElement =
                    <div class="white-box">
                        <h3 class="titulo-cine">RESULTADOS DE BÚSQUEDA</h3>
                        <div class="row">
                            `;
                $.each(data, function (index, football) {
                    footballElement += `
                        <div class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
                            <div class="card card-event info-overlay">
                                <h1 class="tag">${football.nombreContenido}</h1>

```

```

        <div class="img has-background">
            <a href="/api/FootballContent/${football.nombreContenido}"
            class="event-pop-link">
                <div class="event-pop-info">
                    <p class="price" style="font-size:1.1em; padding:
5px;">${football.nombreContenido}</p>
                    <span style="font-size: 0.90em; background-color: #53234f;">
                        VER CONTENIDO</span><br/><br/>
                </div>
            </a>
            <a href="/api/FootballContent/${football.nombreContenido}"></a>
        </div>
    </div>
</div>
`;
});
footballElement += `

</div>
</div>
`;
// Agrega todos los elementos de película al contenedor de resultados de búsqueda
$('#search-results').append(footballElement);
}
});
});
});
});
</script>
```

Cuando se escribe texto en el buscador empieza la búsqueda y se recogen los contenidos que empiezan por ese nombre por parte del controlador.

```
var url = query ? '/api/FootballContent/search/realtme' : '/api/FootballContent/';
```

Si no hay nada en el buscador devuelve el primero método que simplemente muestra todos los partidos de fútbol que hay recogidos en la base de datos. En el momento que se le pase algún elemento al buscador empieza la búsqueda y todos los nombres que empiecen por ese valor se añaden al campo de [footballElement](#).

Controlador/Vista Contenido en directo

La clase [LiveContentController](#) se encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para la sección de contenido en directo. Dentro de la vista se recogen funciones de administrador como crear contenido en directo, modificar el contenido en directo y borrarlo, luego está la vista general para usuarios podrá ver los futuros contenidos en directo para que sepan cuando se van a emitir y cuando el tramo horario coincida con uno de los contenidos en directo se emite, solo se emite un contenido en directo a la vez, dentro de la creación de contenidos está restringida la opción de que los contenidos se pisen entre sí.

Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/controller/ LiveContentController.java.

La clase liveContent es la encargada de la vista por parte del contenido en directo donde se ven todos los contenidos disponibles en la plataforma, no incluye una sección de comentarios.

Esta clase se encuentran en el archivo:

src/main/resources/templates/ liveContent.html

Definimos la ruta del controlador

```
@RequestMapping("/api/LiveContent")
```

Desde los paneles de la vista para acceder a los diferentes contenidos, para acceder a el contenido en directo usamos esa ruta del controlador.

```
<li class="nav-item"><a th:href="@{/api/LiveContent}" class="nav-link">CONTENIDO EN DIRECTO</a>
```

METODOS GET

Método **getLiveContent (Model model, HttpSession session)**: Este método está anotado con **@GetMapping("")**, lo que significa que maneja las solicitudes GET a la ruta /api/ LiveContent. Este método por un lado comprueba el plan de suscripción del usuario ya que para poder ver el contenido en directo se necesita tener el plan de Suscripción PRO, si el usuario no lo tiene se le envía directamente a la sección dentro del perfil del usuario para actualizar el plan de suscripción.

Por otro lado, le pasa a la vista tres listas, una con todos los contenidos en directo, otra con los futuros contenidos en directo y otra con el contenido en directo si lo hay. Retorna la página de contenido en directo. También se comparten los roles de los usuarios.

```
@GetMapping
public String getLiveContent(Model model, HttpSession session, RedirectAttributes redirectAttributes) {
    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
    }
    if (user.getPlanSuscripcion().equals("Gratis") || user.getPlanSuscripcion().equals("Basico")) {
        redirectAttributes.addFlashAttribute("errorMessage", "Actualiza el plan de suscripción para
acceder a este contenido");
        return "redirect:/api/userProfile/";
    }
    if (user != null) {
        Set<Role> roles = user.getRoles();
        model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.toList()));
    }
    List<LiveContent> liveContentsAll = liveContentService.getAllLiveContents();
    model.addAttribute("liveContents", liveContentsAll);
    // Lógica para obtener el contenido en directo que coincide con la hora actual
    LiveContent liveContent = liveContentService.getCurrentLiveContent();
```

```

        model.addAttribute("liveContentLive", liveContent);

        List<LiveContent> futureLiveContents = liveContentService.getFutureLiveContents();
        model.addAttribute("futureLiveContents", futureLiveContents);
        return "liveContent";
    }
}

```

Dentro de la vista definimos los campos que son exclusivamente para el administrador y los que son para el usuario, también hacemos esa comprobación cuando le llegan peticiones al controlador para que haya una doble capa de seguridad.

```

<div th:if="${roles != null and #lists.contains(roles, 'ADMIN')}" class="col-6 col-sm-4 col-md-3 col-xl-2dot4
mb20" >

```

Vista para futuros contenido en directo:

```

<h3 class="titulo-cine">PROXIMOS CONTENIDOS EN DIRECTO</h3>
<div class="row">
    <!-- Iteración sobre películas -->
    <div th:each="content : ${futureLiveContents}">
        <div class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
            <div class="card card-event info-overlay">
                <h1 class="tag" th:text="${content.type}"></h1>
                <div class="img has-background">
                    <a class="event-pop-link">
                        <div class="event-pop-info">
                            <p class="price" style="font-size:1.1em; padding: 5px;">
                                th:text="${content.nombreContenido}"</p>
                            <span style="font-size: 0.90em; background-color: #53234f;
color: #ffffff;">
                                th:text="'Hora de inicio: ' + content.startTime + ''
                            '></span><br/>
                        </div>
                    </a>
                    <a th:href="@{'/' + ${content.url_image}}">
                </a>
            </div>
        </div>
    </div>

```

Un bucle con thymeleaf [2] [6] en el que recorremos los futuros y con la ayuda de Bootstrap [8] los mostramos de la forma más estética posible.

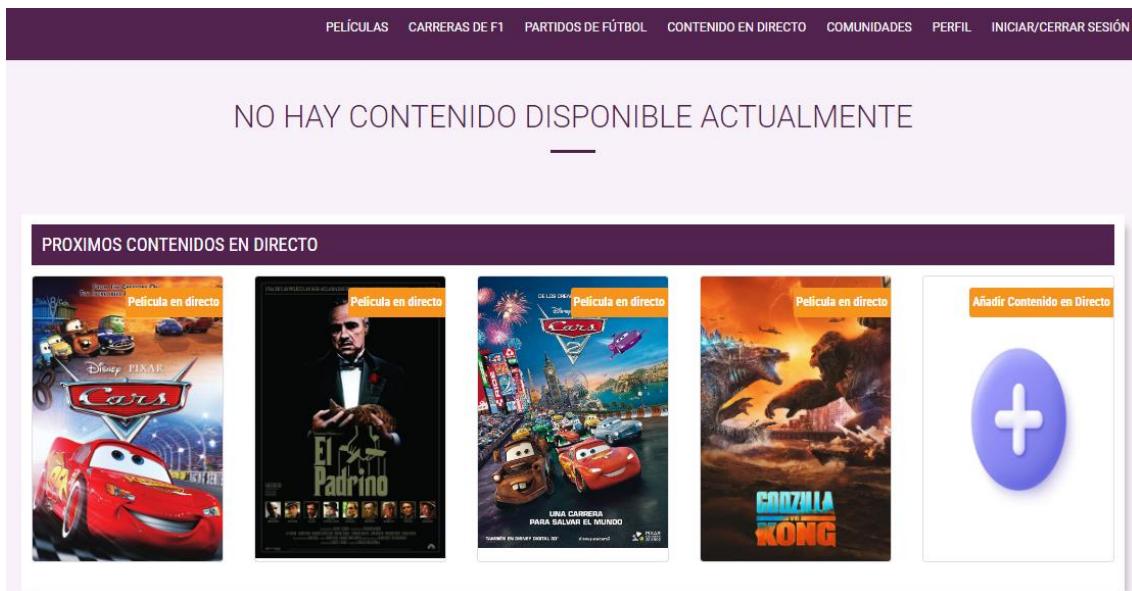


Figura 36: Panel para implementar el método `getLiveContent`.

Vista para contenido en directo:

En el caso de que en ese tramo horario no haya contenido en directo se muestra:

```
<div th:if="${errorMessage}" class="alert alert-danger wc_notice" th:text="${errorMessage}"></div>
<div class="row clearfix">
    <h1 class="text-center title-1" th:if="${liveContentLive != null}">
        th:text="${liveContentLive.nombreContenido}"</h1>
    <h1 class="text-center title-1" th:if="${liveContentLive == null}">No hay contenido disponible
        actualmente</h1>
    <hr class="mx-auto small text-hr" style="margin-bottom: 30px !important">

    <div style="clear:both">
        <hr>
    </div>
</div>
```

En el caso de que si haya:

```
<div class="white-box" style="padding:15px" id="liveContentContainer" th:if="${liveContentLive != null}">
    <h3 class="titulo-cine" th:text="${liveContentLive.nombreContenido}"></h3>
    <div class="row mb20">
        <div class="col-3 d-none d-md-block">
            <a th:src="@{'/' + ${liveContentLive.url_image}}" data-toggle="lightbox">
                <img class="img-responsive" style="width: 262px;">
                th:src="@{'/' + ${liveContentLive.url_image}}"/>
            </a>
        </div>
        <div class="col-6 d-none d-md-block">
            <iframe width="100%" height="350" th:src="${liveContentLive.url_video}">
        </div>
    </div>
</div>
```

```

        title="YouTube video player" frameborder="0"
        allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope;
picture-in-picture; web-share"

        allowfullscreen
        controlsList="nodownload"
        th:attr="data-start-time=${liveContentLive.startTime},data-end-
time=${liveContentLive.endTime}">
    </iframe>
</div>
<div class="col-3 d-none d-md-block" style="padding-left: 10px;">
    <ul class="list-unstyled">
        <li>
            <p class="text-muted">Hora de inicio:<br/>
            <strong class="text-dark" th:text="${liveContentLive.getStartTime() + '}'></strong>
        </p>
        </li>
        <li>
            <p class="text-muted">Hora de fin:<br/>
            <strong class="text-dark" th:text="${liveContentLive.getEndTime() + '}'></strong>
        </p>
        </li>
        <li>
            <p class="text-muted">Tipo:<br/>
            <strong class="text-dark" th:text="${liveContentLive.type}"></strong>
        </p>
        </li>
    </ul>
</div>

<div class="row mb30">
    <div class="col-sm-12">
        <span class="text-muted">Descripción:</span>
        <p th:text="${liveContentLive.descripcion}"></p>
    </div>
</div>
</div>

```

Recoge el contenido que se pasa desde el controlador que es el que coincide con el tramo horario y con la ayuda de bootstrap [8] para un buen diseño, muestra información destacada de ese contenido como el año de fundación, una descripción, los actores, la imagen de portada y como no el video.

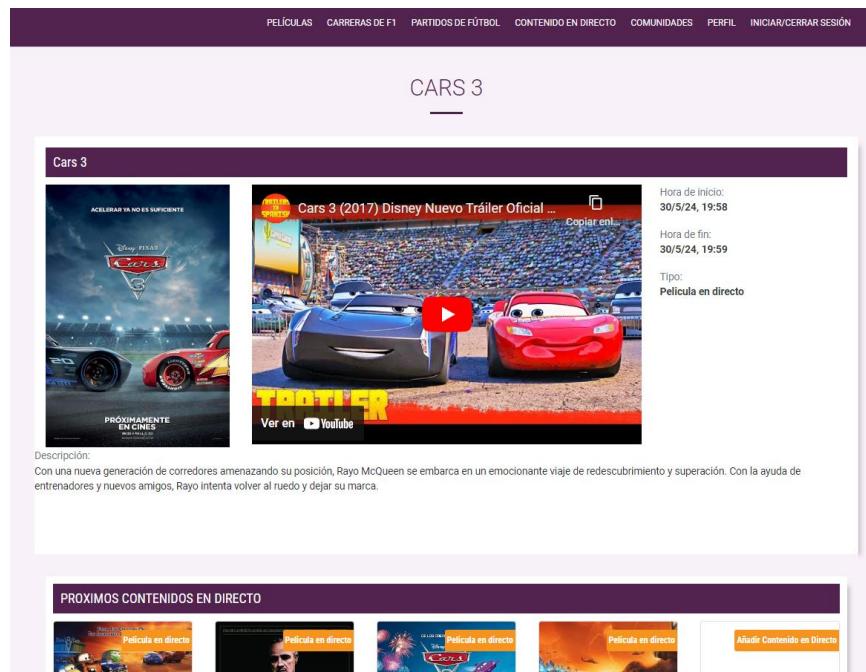


Figura 37: Panel para implementar el método `getLiveContent` con contenido en Vivo.

Método `agregarLiveContent (HttpSession session)`:

Anotado con `@GetMapping("/agregarLiveContent")`, lo que significa que maneja las solicitudes GET a la ruta /api/ LiveContent/ agregarLiveContent. Este método verifica si el usuario actual está autenticado y es un administrador. Si es así, envía al usuario al fichero de agregar contenido donde aparte de la creación de contenido en directo, también se crean el resto de la aplicación. Cabe resaltar que este método no es de la creación de contenido en directo, solo envía a la ruta o pestaña donde se crean usando un método post.

```
@GetMapping("/agregarLiveContent")
public String añadirLiveContent(HttpSession session) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    return "agregarContenido";
}
```

Dentro de la vista, la opción de acceder al panel para poder crear nuevo contenido en directo se verá solo si eres administrador y en la sección de futuros contenidos en directo.

```
<div th:if="${roles != null and #lists.contains(roles, 'ADMIN')}" class="col-6 col-sm-4 col-md-3 col-xl-2dot4
mb20" >
    <div class="card card-event info-overlay">
        <h1 class="tag">Añadir Contenido en Directo</h1>
        <div class="img has-background">
            <a th:href="@{'/api/LiveContent/agregarLiveContent'}" class="event-
pop-link">
                <div class="event-pop-info">
                    <p class="price" style="font-size:1.1em; padding: 5px;"> Palsa
para agregar un
```

```

        nuevo contenido en directo</p>
        <span style="font-size: 0.90em; background-color: #53234f;">
            <span>AÑADIR</span>
        </span>
    CONTENIDO</span><br/><br/>
        </div>
    </a>
    <a th:href="@{/images/Peliculas/agregar.png}"></a>
        </div>
    </div>
</div>

```

Se agrega una imagen con un “+” para que cuando el usuario final con el uso de thymeleaf [2] [6] cuando clique le envié a la ruta del controlador, que le va a enviar al panel de agregar contenido.



Figura 38: Panel para implementar el método `agregarLiveContent`.

METODOS POST

Método `createLiveContent()`: Este método está anotado con `@PostMapping("/create")`, lo que significa que maneja las solicitudes POST a la ruta `/api/LiveContent/create`. Este método crea un nuevo contenido en la base de datos. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad de contenido en directo. He creado dentro del método unos requisitos para que los contenidos en directo no se puedan pisar entre ellos a la hora de creación y así evitar tener varios contenidos en directo a la vez.

```

@PostMapping("/create")
public String createLiveContent(
    @RequestParam String nombreContenidoL,
    @RequestParam String descripcionL,
    @RequestParam String url_imageL,
    @RequestParam String url_videoL,
    @RequestParam LocalDateTime startTime,
    @RequestParam LocalDateTime endTime,
    @RequestParam String type,
    @RequestParam Integer anhoL,
)

```

```

        HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        LiveContent liveContentNew = new LiveContent();
        liveContentNew.setNombreContenido(nombreContenidoL);
        liveContentNew.setDescripcion(descripcionL);
        liveContentNew.setUrl_image(url_imageL);
        liveContentNew.setUrl_video(url_videoL);
        liveContentNew.setStartTime(startTime);
        liveContentNew.setEndTime(endTime);
        liveContentNew.setType(type);
        liveContentNew.setAnho(anhoL);

        // Verificar que la hora de inicio no sea mayor que la hora de fin
        if (startTime.isAfter(endTime)) {
            redirectAttributes.addFlashAttribute("errorMessage", "La hora de inicio no puede ser mayor que la hora de fin.");
            return "redirect:/api/LiveContent/agregarLiveContent";
        }
        List<LiveContent> allLiveContents = liveContentService.getAllLiveContents();
        for (LiveContent existingContent : allLiveContents) {
            if (!existingContent.getNombreContenido().equals(nombreContenidoL) &&
                ((existingContent.getStartTime().isBefore(endTime) &&
existingContent.getEndTime().isAfter(startTime)) ||
                (existingContent.getStartTime().isBefore(startTime) &&
existingContent.getEndTime().isAfter(endTime)))) {
                redirectAttributes.addFlashAttribute("errorMessage", "El horario del contenido en directo se superpone con otro contenido en directo existente.");
                return "redirect:/api/LiveContent/agregarLiveContent";
            }
        }
        liveContentService.createLiveContent(liveContentNew);
        redirectAttributes.addFlashAttribute("successMessage", "Contenido en directo creado con éxito");
        return "redirect:/api/LiveContent"; // Redirect to the main live content page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al crear el contenido en directo");
        return "redirect:/api/LiveContent/agregarLiveContent"; // Redirect back to the add live content page
    }
}

```

PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO EN DIRECTO

Nombre Contenido:	Descripción:
<input type="text"/>	<input type="text"/>
URL Imagen:	URL Video:
<input type="text"/>	<input type="text"/>
Hora de inicio:	Hora de fin:
<input type="text"/> dd/mm/aaaa --:-	<input type="text"/> dd/mm/aaaa --:-
Tipo:	Año:
<input type="text"/>	<input type="text"/>
Añadir Contenido en Directo	

Figura 39: Panel para implementar el método `createLiveContent`.

Método **updateLiveContent ()**: Anotado con `@PostMapping("/update/{nombreLiveContenido}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/ LiveContent/update{nombreLiveContenido}`. Este método actualiza los detalles de un contenido en directo específico en la base de datos. Recogiendo todos los parámetros que al administrador cambie del contenido en directo, ya sea la descripción, año, hora de inicio y de fin de la vista comprobando en todo momento que se trata de un usuario administrador. He creado dentro del método unos requisitos para que los contenidos en directo no se puedan pisar entre ellos a la hora de creación y así evitar tener varios contenidos en directo a la vez.

```

@PostMapping("/update/{nombreLiveContenido}")
public String updateLiveContent(
    @PathVariable String nombreLiveContenido,
    @RequestParam String nombreContenido,
    @RequestParam String descripcion,
    @RequestParam String url_image,
    @RequestParam String url_video,
    @RequestParam LocalDateTime startTime,
    @RequestParam LocalDateTime endTime,
    @RequestParam String type,
    @RequestParam Integer anho,
    RedirectAttributes redirectAttributes) {
    try {
        LiveContent detallesLiveContent = new LiveContent();
        detallesLiveContent.setNombreContenido(nombreContenido);
        detallesLiveContent.setDescripcion(descripcion);
        detallesLiveContent.setUrl_image(url_image);
        detallesLiveContent.setUrl_video(url_video);
        detallesLiveContent.setStartTime(startTime);
        detallesLiveContent.setEndTime(endTime);
        detallesLiveContent.setType(type);
        detallesLiveContent.setAnho(anho);
        // Verificar que la hora de inicio no sea mayor que la hora de fin
        if (startTime.isAfter(endTime)) {
    }
}

```

```

        redirectAttributes.addFlashAttribute("errorMessage", "La hora de inicio no puede ser mayor que
la hora de fin.");
        return "redirect:/api/LiveContent";
    }
    List<LiveContent> allLiveContents = liveContentService.getAllLiveContents();
    for (LiveContent existingContent : allLiveContents) {
        if (!existingContent.getNombreContenido().equals(nombreLiveContenido) &&
            ((existingContent.getStartTime().isBefore(endTime) &&
existingContent.getEndTime().isAfter(startTime)) ||
            (existingContent.getStartTime().isBefore(startTime) &&
existingContent.getEndTime().isAfter(endTime))) {
            redirectAttributes.addFlashAttribute("errorMessage", "El horario del contenido en directo
se superpone con otro contenido en directo existente.");
            return "redirect:/api/LiveContent";
        }
    }
    liveContentService.updateLiveContent(nombreLiveContenido, detallesLiveContent);
    redirectAttributes.addFlashAttribute("successMessage", "Contenido en directo actualizado con
éxito");
} catch (RuntimeException e) {
    redirectAttributes.addFlashAttribute("errorMessage", e.getMessage());
}
return "redirect:/api/LiveContent";
}

```

Dentro de la vista usando thymeleaf [2] [6] completamos los campos con los datos actuales y si el administrador actualiza algún dato hacemos la solicitud al controlador y este actualiza los datos en la base de datos.

```

<div class="card">
    <div class="card-header">
        <h3 class="titulo-cine">PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO EN
DIRECTO</h3>
    </div>
    <div class="card-body">
        <form th:action="@{/api/LiveContent/create}" method="post" class="row g-3">
            <div class="col-md-6 mb-3">
                <label for="nombreContenidoL" class="form-label">Nombre
                Contenido:</label>
                <input type="text" id="nombreContenidoL" name="nombreContenidoL"
required
                    class="form-control">
            </div>
            <div class="col-md-6 mb-3">
                <label for="descripcionL" class="form-label">Descripción:</label>
                <input type="text" id="descripcionL" name="descripcionL" required
                    class="form-control">
            </div>
        </form>
    </div>
</div>

```

```

<div class="col-md-6 mb-3">
    <label for="url_imagenL" class="form-label">URL Imagen:</label>
    <input type="text" id="url_imagenL" name="url_imagenL" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="url_videoL" class="form-label">URL Video:</label>
    <input type="text" id="url_videoL" name="url_videoL" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="startTime" class="form-label">Hora de inicio:</label>
    <input type="datetime-local" id="startTime" name="startTime" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="endTime" class="form-label">Hora de fin:</label>
    <input type="datetime-local" id="endTime" name="endTime" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="type" class="form-label">Tipo:</label>
    <input type="text" id="type" name="type" required class="form-
control">
</div>
<div class="col-md-6 mb-3">
    <label for="anhoL" class="form-label">Año:</label>
    <input type="number" id="anhoL" name="anhoL" required class="form-
control">
</div>
<div class="col-12">
    <button type="submit" class="btn btn-primary">Añadir Contenido en
    Directo
    </button>
</div>
</form>
</div>
</div>

```

Rellenamos cada uno de los campos de la entidad de contenido en directo y recogemos los cambios del administrador si los hay, cuando pulsa el botón de actualizar, enviamos la solicitud post al controlador este actualiza esos datos en la base de datos y vuelve de nuevo a la vista ya con los datos nuevos actualizados.

PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO EN DIRECTO: CARS 3

Nombre Contenido:	Cars 3	Descripción:	Con una nueva generación de corredores amenazando su posición, Rayo McQu...
URL Imagen:	images/Peliculas/Cars3.png	URL Video:	https://www.youtube.com/embed/wtmW9rSRlzU?si=MttED7g2UgQk5A08
Hora de inicio:	30/05/2024 19:58	Hora de fin:	30/05/2024 19:59
Tipo:	Película en directo	Año:	2024
<input style="background-color: #f0ad4e; color: white; border: none; padding: 5px 10px; border-radius: 5px; font-weight: bold; margin-bottom: 5px;" type="button" value="Actualizar Contenido en Directo"/> <input style="background-color: #d9534f; color: white; border: none; padding: 5px 10px; border-radius: 5px; font-weight: bold;" type="button" value="Borrar Contenido en Directo"/>			

Figura 40: Panel para implementar el método `updateLiveContent`.

Método **deleteLiveContent (...)**: Anotado con `@PostMapping("/delete/{id}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/LiveContent/delete/ {id}`. Este método elimina un contenido en directo específico de la base de datos. Comprueba que se trata de un usuario administrador, también para la incorporación de seguridad el administrador introduce su contraseña y la confirma en caso de no ser validas no se podrá borrar el contenido en directo.

```
@PostMapping("/delete/{id}")
public String deleteLiveContent(@PathVariable Long id, RedirectAttributes redirectAttributes) {
    try {
        liveContentService.deleteLiveContent(id);
        redirectAttributes.addFlashAttribute("successMessage", "Contenido en directo borrado con éxito");
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", e.getMessage());
    }
    return "redirect:/api/LiveContent";
}
```

Dentro de la vista solo si eres puedes eliminar el contenido en directo, como en todas las acciones post, saldrá el mensaje de error si lo hay y si la acción se ha realizado correctamente también saldrá el mensaje de éxito.

```
<div th:if="#{lists.contains(roles, 'ADMIN')}" class="container mt-5">
<form th:action="@{/api/LiveContent/delete/" + ${liveContentAll.id}"/}" method="post">
    <button type="submit" class="btn btn-danger">Borrar Contenido en Directo</button>
</form>
```

PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO EN DIRECTO: CARS 3

Nombre Contenido:	Cars 3	Descripción:	Con una nueva generación de corredores amenazando su posición, Rayo McQu...
URL Imagen:	images/Peliculas/Cars3.png	URL Video:	https://www.youtube.com/embed/wtmW9rSRlzu?si=MttED7g2UgQk5A08
Hora de inicio:	30/05/2024 19:58	Hora de fin:	30/05/2024 19:59
Tipo:	Película en directo	Año:	2024
<input style="background-color: #e69138; color: white; padding: 5px; width: 100%; height: 30px; border-radius: 5px; font-weight: bold; margin-bottom: 10px;" type="button" value="Actualizar Contenido en Directo"/>		<input style="background-color: #c00000; color: white; padding: 5px; width: 100%; height: 30px; border-radius: 5px; font-weight: bold;" type="button" value="Borrar Contenido en Directo"/>	

Figura 41: Panel para implementar el método deleteLiveContent

8.2.2 Controladores/Vista Comentarios

Cada contenido de la aplicación menos el contenido en directo tiene la posibilidad de ser comentado por los usuarios de la aplicación, siempre y cuando el plan de suscripción sea mínimo básico ya que con la versión gratuita solo se pueden ver los comentarios de otros usuarios, pero no hacer uno. El controlador sigue la misma lógica y estructura para los comentarios de los tres contenidos de la aplicación que son fútbol, f1 y películas. Desde la vista se envía el comentario al controlador, este hace las comprobaciones pertinentes, como que el usuario tiene validado el pago, que tiene un plan de suscripción accesible para hacer comentarios y si todos estos requisitos se cumplen el controlador guarda ese comentario en el modelo y desde la vista se puede ver, una vez hecho.

Controlador/Vista Comentarios Películas

La clase `ComentarioPeliculaController` encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para el envío de comentarios de contenido relacionado con películas por parte de los usuarios. Cabe resaltar que es una opción para usuarios con plan básico o superior, para poder controlar eso usaremos thymeleaf [2] [6] y restricciones en los controladores. Con el plan gratis si se podrán ver los comentarios de otros usuarios, pero no hacer.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/controller/ ComentarioPeliculaController.java.`

La clase `indexDetallado` es la encargada de la vista es la encargada tanto de la visualización del contenido de películas y la parte de operaciones CRUD del administrador, pero también incluye la sección de comentarios de los contenidos y el control del rol del usuario y de su plan de suscripción con el panel de administrador.

```
<div th:if="#{lists.contains(roles, 'USER')}">
    <h3 class="titulo-cine">COMENTARIOS</h3>
    <div class="comentarios">
```

```

<!-- Sección para hacer comentario -->
<div th:if="${session.user.planSuscripcion != 'Gratis'}">
    <form th:action="@{/api/ComentariosPelícula/CrearComentarioPelícula/" +
${película.nombreContenido}">
        method="post"
        <!-- Otros campos del formulario -->
        <label for="textoComentario">Comentario:</label>
        <textarea id="textoComentario" name="textoComentario" required></textarea>

        <label for="valoracionComentario">Valoración:</label>
        <input type="number" id="valoracionComentario" name="valoracionComentario"
min="0"
max="10"
required>
        <button type="submit">Enviar Comentario</button>
    </form>
</div>
<br><br>
<!-- Sección para mostrar comentarios -->
<div th:each="comentario : ${comentarios}" class="comentario">
    <h4 th:text="${comentario.getUsuario().getNickname()}"></h4>
    <div class="comentario-content">
        <p th:text="${comentario.texto}" class="comentario-texto"></p>
        <p th:text="${comentario.valoracion}" class="comentario-valoracion"></p>
    </div>
</div>

</div>
</div>

```

Comprobamos primero que el plan del usuario no sea el plan gratuito ya que este no puede enviar comentarios, y enviamos la petición post con el texto del comentario y la valoración.

Dentro de la vista en este caso para las películas el comentario está formado por un texto mostrando la opinión del usuario y una valoración, para que el resto de usuarios puedan ver las opiniones de otros usuarios antes de ver el contenido. Dentro de cada comentario recogemos la fecha con el uso de LocalDate y el usuario que ha escrito el usuario con el uso de Thymeleaf [2] [6].

Esta clase se encuentran en el archivo:

`src/main/resources/templates/indexDetallado.html`

Definimos la ruta del controlador

```
@RequestMapping("/api/ComentariosPelícula")
```

MÉTODOS GET

En este caso como se trata solo de una acción de enviar comentarios no hay métodos get ya que el panel de hacer comentarios se encuentra dentro del panel de ver el contenido de las películas, que para acceder a este la ruta es:

```
<li class="nav-item"><a th:href="@{/api/peliculas/}" class="nav-link">PELÍCULAS</a></li>
```

MÉTODOS POST

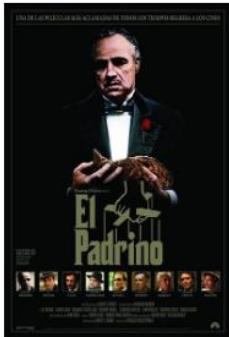
```
@PostMapping("/CrearComentarioPelícula/{nombrePelícula}")
createComentarioPelícula(@PathVariable String nombrePelícula, @RequestParam String textoComentario,
@RequestParam Short valoracionComentario, HttpSession session)
```

Este método maneja las solicitudes POST a la ruta:

/api/ComentariosPelícula/CrearComentarioPelícula/{nombrePelícula}. Crea crea un nuevo comentario sobre un contenido de películas a la base de datos, se pasa el nombre de la película. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad comentarioPelícula.

```
@PostMapping("/CrearComentarioPelícula/{nombrePelícula}")
public String createComentarioPelícula(@PathVariable String nombrePelícula, @RequestParam String
textoComentario, @RequestParam Short valoracionComentario, HttpSession session) {
    // Obtener el usuario de la sesión
    User user = (User) session.getAttribute("user");
    if (user == null || user.getPagoValidado().equals(false) || "Sin
Plan".equals(user.getPlanSuscripción())) {
        return "login"; // Redirect the user to the login page if not authenticated
    }
    // Obtener la película por su nombre
    Película película = películaService.getPelículaPorNombre(nombrePelícula)
        .orElseThrow(() -> new RuntimeException("Película not found"));
    // Obtener el nickname del usuario
    String nickname = user.getNickname();
    // Crear el comentario
    ComentarioPelícula comentarioPelícula = new ComentarioPelícula();
    comentarioPelícula.setTexto(textoComentario);
    comentarioPelícula.setNickname(nickname);
    comentarioPelícula.setValoración(valoracionComentario);
    comentarioPelícula.setFechaComentario();
    comentarioPelícula.setUsuario(user);
    comentarioPelícula.setPelícula(película);
    // Guardar el comentario
    comentarioPelículaService.createComentarioPelícula(comentarioPelícula);
    // Redirigir a la página de detalles de la película
    return "redirect:/api/peliculas/" + nombrePelícula;
}
```

El Padrino





El Padrino: 50 años - Tráiler oficial

Ver más ta... Compartir

TRÁILER

Ver en YouTube

Dirección:
Francis Ford Coppola

Duración:
177 minutos

Calificación:
18 años

Actores:
Marlon Brando, Al Pacino, James Caan

Otros Datos:
Ganadora de 3 premios Oscar.

Año:
1972

Sinopsis:

Un poderoso drama criminal que sigue la vida de la familia Corleone en el mundo del crimen organizado.

COMENTARIOS

Comentario:

Valoración:

luna33
Me encanta 9

Juan99
Ami esta película me ha gustado mucho 9

Figura 42: Panel que implementa el método `createComentarioPelícula`

Controlador/Vista Comentarios F1

La clase `ComentarioF1Controller` encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para él envío de comentarios de contenido relacionado con carreras de f1 por parte de los usuarios. Cabe resaltar que es una opción para usuarios con plan básico o superior, para poder controlar eso usaremos thymeleaf [2] [6] y restricciones en los controladores. Con el plan gratis si se podrán ver los comentarios de otros usuarios, pero no hacer.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/controller/ ComentarioF1Controller.java.`

La clase `indexDetalladoF1` es la encargada de la vista es la encargada tanto de la visualización del contenido de carreras de F1 y la parte de operaciones CRUD del administrador, pero también incluye la sección de comentarios de los contenidos y el control del rol del usuario y de su plan de suscripción con el panel de administrador.

```
<div th:if="#{lists.contains(roles, 'USER')}">
    <h3 class="titulo-cine">COMENTARIOS</h3>
    <div class="comentarios">
        <!-- Sección para hacer comentario -->
        <div th:if="${session.user.planSuscripcion != 'Gratis'}">
            <form th:action="@{/api/ComentariosF1/CrearComentarioF1/" +
${f1Content.nombreContenido}">
                method="post"
                <!-- Otros campos del formulario -->
                <label for="textoComentario">Comentario:</label>
                <textarea id="textoComentario" name="textoComentario" required></textarea>

                <button type="submit">Enviar Comentario</button>
            </form>
        </div>
        <br><br>
        <!-- Sección para mostrar comentarios -->
        <div th:each="comentario : ${comentarios}" class="comentario">
            <h4 th:text="${comentario.getUsuario().getNickname()}"></h4>
            <div class="comentario-content">
                <p th:text="${comentario.texto}" class="comentario-texto"></p>
            </div>
        </div>
    </div>
</div>
```

Comprobamos primero que el plan del usuario no sea el plan gratuito ya que este no puede enviar comentarios, y enviamos la petición post con el texto del comentario.

Dentro de la vista en este caso para las carreras de F1 el comentario está formado por un texto mostrando la opinión del usuario, para que el resto de usuarios puedan ver las opiniones de otros usuarios antes de ver el contenido. Dentro de cada comentario recogemos la fecha con el uso de LocalDate y el usuario que ha escrito el usuario con el uso de Thymeleaf [2] [6].

Esta clase se encuentran en el archivo:

`src/main/resources/templates/indexDetalladoF1.html`

Definimos la ruta del controlador

```
@RequestMapping("/api/ComentariosF1")
```

MÉTODOS GET

En este caso como se trata solo de una acción de enviar comentarios no hay métodos get ya que el panel de hacer comentarios se encuentra dentro del panel de ver el contenido de F1, que para acceder a este la ruta es:

```
<li class="nav-item"><a th:href="@{/api/F1/}" class="nav-link">CARRERAS DE F1</a></li>
```

MÉTODOS POST

```
@PostMapping("/CrearComentarioF1/{nombreContenido}")
public String createComentarioF1(@PathVariable String nombreContenido, @RequestParam String textoComentario,
HttpSession session)
```

Este método maneja las solicitudes POST a la ruta:

/api/ComentariosF1/CrearComentarioF1/{nombreContenido}. Crea un nuevo comentario sobre un contenido de carreras de F1 a la base de datos, se pasa el nombre de la carrera. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad comentarioF1.

```
@PostMapping("/CrearComentarioF1/{nombreContenido}")
public String createComentarioF1(@PathVariable String nombreContenido, @RequestParam String
textoComentario, HttpSession session) {
    // Obtener el usuario de la sesión
    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
    }
    // Obtener la película por su nombre
    F1Content f1Content = f1ContentService.getF1ContentByNombreCarrera(nombreContenido)
        .orElseThrow(() -> new RuntimeException("Película not found"));
    //Obtener el nickname del usuario
    String nickname = user.getNickname();
    // Crear el comentario
    ComentarioF1 comentarioF1 = new ComentarioF1();
    comentarioF1.setTexto(textoComentario);
    comentarioF1.setNickname(nickname);
    comentarioF1.setFechaComentario();
    comentarioF1.setUsuario(user);
    comentarioF1.setF1Content(f1Content);
    // Guardar el comentario
    comentarioF1Service.createComentarioF1(comentarioF1);
    // Redirigir a la página de detalles de la película
    return "redirect:/api/F1/" + nombreContenido;
}
```

GP España 2005





Ver más ta... Compartir

Ver en  YouTube

CAMPEONATOS DE FERNANDO ALONSO

Año:
2005

Circuito:
Circuit de Barcelona-Catalunya

Nacionalidad:
España

Equipos:
McLaren, Renault, Ferrari, Williams, Toyota, BAR, Red Bull, Sauber, Jordan, Minardi

Pilotos:
Räikkönen, Alonso, Schumacher, Montoya, Webber, Barrichello, Coulthard, Button, Heidfeld, Villeneuve, Fisichella, Massa, Liuzzi, Karthikeyan, Albers, Monteiro, Sato, Klien, de la Rosa, Davidson

Otros Datos:
1. Räikkönen, 2. Alonso, 3. Schumacher

Duración:
66 minutos

Descripción:

El Gran Premio de España de 2005 se celebró en el Circuit de Barcelona-Catalunya el 8 de mayo. Kimi Räikkönen, al volante de su McLaren, se llevó la victoria en una carrera llena de acción y emociones. La competencia estuvo marcada por batallas intensas en la pista y estrategias tácticas entre los equipos principales. Alonso y Schumacher también ofrecieron una actuación memorable, luchando por los lugares del podio. La carrera fue un gran espectáculo para los aficionados, con momentos emocionantes que mantuvieron a todos al borde de sus asientos hasta la bandera a cuadros.

COMENTARIOS

Comentario:

Enviar Comentario

Juan99
 La lucha entre Schumacher y Alonso de las mejores en la historia de la F1

Figura 43: Panel que implementa el método createComentarioF1

Controlador/Vista Comentarios Fútbol

La clase `ComentarioFootballController` encarga de recibir solicitudes por parte de la vista, tratar esas solicitudes usando los servicios para el envío de comentarios de contenido relacionado con partidos de fútbol por parte de los usuarios. Cabe resaltar que es una opción para usuarios con plan básico o superior, para poder controlar eso usaremos thymeleaf [2] [6] y restricciones en los controladores. Con el plan gratis si se podrán ver los comentarios de otros usuarios, pero no hacer.

Esta clase se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/controller/ComentarioFootballController.java.
```

La clase indexDetalladoFootball es la encargada de la vista es la encargada tanto de la visualización del contenido de partidos de fútbol y la parte de operaciones CRUD del administrador, pero también incluye la sección de comentarios de los contenidos y el control del rol del usuario y de su plan de suscripción con el panel de administrador.

```
<div th:if="#{lists.contains(roles, 'USER')}">
    <h3 class="titulo-cine">COMENTARIOS</h3>
    <div class="comentarios">
        <!-- Sección para hacer comentario -->
        <div th:if="${session.user.planSuscripcion != 'Gratis'}">
            <form th:action="@{'/api/ComentariosFootball/CrearComentarioFootabll/' +
${footballContent.nombreContenido}}"
                method="post">
                <label for="textoComentario">Comentario:</label>
                <textarea id="textoComentario" name="textoComentario" required></textarea>
                <button type="submit">Enviar Comentario</button>
            </form>
        </div>
        <br><br>
        <!-- Sección para mostrar comentarios -->
        <div th:each="comentario : ${comentarios}" class="comentario">
            <h4 th:text="${comentario.getUsuario().getNickname()}"></h4>
            <div class="comentario-content">
                <p th:text="${comentario.texto}" class="comentario-texto"></p>
            </div>
        </div>
    </div>
```

Comprobamos primero que el plan del usuario no sea el plan gratuito ya que este no puede enviar comentarios, y enviamos la petición post con el texto del comentario y la valoración.

Dentro de la vista en este caso para las películas el comentario está formado por un texto mostrando la opinión del usuario y una valoración, para que el resto de usuarios puedan ver las opiniones de otros usuarios antes de ver el contenido. Dentro de cada comentario recogemos la fecha con el uso de LocalDate y el usuario que ha escrito el usuario con el uso de Thymeleaf [2] [6].

Esta clase se encuentran en el archivo:

`src/main/resources/templates/indexDetalladoFootball.html`

Definimos la ruta del controlador

```
@RequestMapping("/api/ComentariosFootball")
```

MÉTODOS GET

En este caso como se trata solo de una acción de enviar comentarios no hay métodos get ya que el panel de hacer comentarios se encuentra dentro del panel de ver el contenido de fútbol, que para acceder a este la ruta es:

```
<li class="nav-item"><a th:href="@{/api/FootballContent/}" class="nav-link">PARTIDOS DE  
FÚTBOL</a></li>
```

MÉTODOS POST

```
@PostMapping("/CrearComentarioFootball/{nombreContenido}")  
    public String createComentarioF1(@PathVariable String nombreContenido, @RequestParam String  
    textoComentario, HttpSession session)
```

Este método maneja las solicitudes POST a la ruta:

/api/ComentariosFootball/CrearComentarioFootball/{nombreContenido}. Crea un nuevo comentario sobre un contenido de partidos de fútbol a la base de datos, se pasa el nombre del partido específico. Recoge los atributos que se le pasan de vista, comprueba mediante las sesiones que se trata de un usuario administrador y crea una nueva entidad comentarioFootball.

```
@PostMapping("/CrearComentarioFootabll/{nombreContenido}")  
    public String createComentarioF1(@PathVariable String nombreContenido, @RequestParam String  
    textoComentario, HttpSession session) {  
        // Obtener el usuario de la sesión  
        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  
        }  
        // Obtener la película por su nombre  
        FootballContent footballContent =  
footballContentService.getFootballContentByNombrePartido(nombreContenido)  
            .orElseThrow(() -> new RuntimeException("Película not found"));  
        //Obtener el nickname del usuario  
        String nickname = user.getNickname();  
        // Crear el comentario  
        ComentarioFootball comentarioFootball = new ComentarioFootball();  
        comentarioFootball.setTexto(textoComentario);  
        comentarioFootball.setNickname(nickname);  
        comentarioFootball.setFechaComentario();  
        comentarioFootball.setUsuario(user);  
        comentarioFootball.setFootballContent(footballContent);  
        // Guardar el comentario  
        comentarioFootballService.createComentarioFootballContent(comentarioFootball);  
        // Redirigir a la página de detalles de la película  
        return "redirect:/api/FootballContent/" +  
comentarioFootball.getFootballContent().getNombreContenido();
```

ATLETICO DE MADRID VS SEVILLA 2024

Atletico de Madrid vs Sevilla 2024





DAZN

LALIGA

Atletico de Madrid vs Sevilla FC ...

Ver más ta...

Compartir

Año:
2024

Estadio:
Wanda Metropolitano

Competición:
Liga Española

Equipos:
Atlético de Madrid, Sevilla

Jugadores:
Luis Suárez, João Félix, Youssef En-Nesyri, Jesús Navas, Ivan Rakitić

Otros Datos:
Partido jugado el 27 de marzo de 2024

Descripción:
 Un emocionante enfrentamiento entre el Atlético de Madrid y el Sevilla promete un choque de estilos. Con ambos equipos luchando por el título, cada gol y cada jugada serán cruciales en este partido de alto vuelo.

COMENTARIOS

Enviar Comentario

Figura 44: Panel para implementar el método createComentarioF1

8.2.3 Controladores/Vista Usuario

Dentro del controlador tenemos dos clases diferentes para la gestión de usuarios, por un lado, el controlador para el panel de registro y de inicio de sesión del usuario, donde aparte de rellenar los datos personales del usuario también se completa la selección del plan de suscripción y la validación del pago en función de ese plan. Por otro lado, está el controlador del perfil del usuario que permite este conocer sus datos, actualizarlos si desea, cambiar su imagen de perfil o contratar un plan de suscripción mayor si lo desea.

Controlador/Vista Inicio de Sesión del Usuario

La clase UserController maneja dos paneles por un lado el de inicio de sesión y por otro lado el de registro del usuario, los datos que le lleguen desde la vista los almacena en base a ciertos requisitos como que el mail o el nickname no exista ya, que haya validado el pago o que haya seleccionado un plan, hasta que no se haga eso correctamente el controlador no va a dejar acceder a la plataforma para ver el contenido.

pág. 117

Esta clase se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/controller/ UserController.java.
```

La clase login es la encargada de la vista por parte del inicio de sesión del usuario en este caso para la gestión de errores ya sea de datos no introducidos, de contraseñas o correos incorrectos se incorpora el uso de javascript y de Ajax [17] con el uso del controlador.

Estas clases se encuentran en el archivo:

```
src/main/resources/templates/ login.html  
src/main/resources/js/ login.js
```

Definimos la ruta del controlador

```
@RequestMapping("/api/users")
```

Desde los paneles de la vista para acceder al registro o inicio de sesión o bien desde el panel de la aplicación en la opción de INICIAR/CERRAR SESIÓN.

```
<li class="nav-item"><a th:href="@{/api/users/logout}" class="nav-link">INICIAR/CERRAR SESIÓN</a></li>
```

O bien al querer ver un contenido sin estar registrado, la aplicación te envía directamente.

```
if (user == null || user.getPagoValidado().equals(false) || "Sin Plan".equals(user.getPlanSuscripcion())) {  
    return "login"; // Redirect the user to the login page if not authenticated  
}
```

MÉTODOS GET

Método **root ()**: Este método está anotado con `@GetMapping("/")`, lo que significa que maneja las solicitudes GET a la ruta `/api/users/`. Este método simplemente envía al panel de `login.html` para ya poder iniciar los registros.

```
@GetMapping("/")  
public String root() {  
    return "login";  
}
```

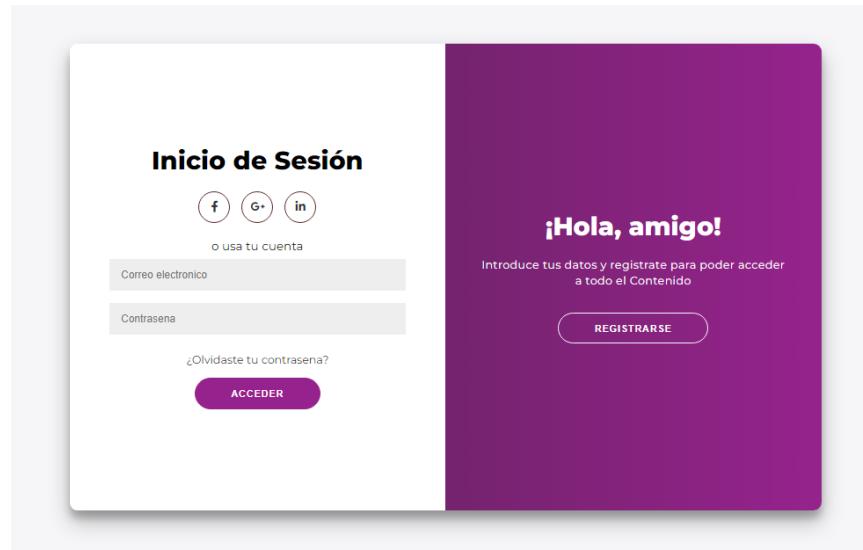


Figura 45 Panel que implementa el método root.

Método **root2 ()**: Este método está anotado con `@GetMapping("/planSuscripcion")`, lo que significa que maneja las solicitudes GET a la ruta `/api/users/ planSuscripcion`. Este método lleva al panel de selección del plan de suscripción, pero primero comprueba que se haya creado una sesión del usuario que se crea cuando se registra, si no es así envía al cliente de nuevo al panel del login para que se registre.



Figura 46: Panel que implementa el método `planSuscripcion`.

Método **logout ()**: Este método está anotado con `@GetMapping("/logout")`, lo que significa que maneja las solicitudes GET a la ruta `/api/users/ logout`. Este método sirve para cerrar la sesión del usuario en el caso de que este lo desee y exista esa sesión y envía de nuevo al panel de inicio de sesión.

```
@GetMapping("/logout")
public String logout(HttpSession session) {
    if (session.getAttribute("user") != null) {
        session.invalidate();
    }
}
```

```

        return "redirect:/api/users/";
    }
}

```

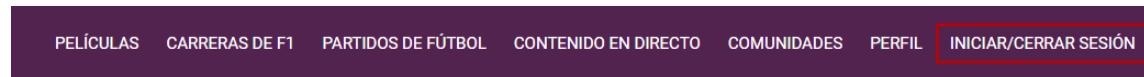


Figura 47: Panel que implementa el método logout.

Desde los paneles de la vista para acceder al registro o inicio de sesión o bien desde el panel de la aplicación en la opción de INICIAR/CERRAR SESIÓN.

```
<li class="nav-item"><a th:href="@{/api/users/logout}" class="nav-link">INICIAR/CERRAR SESIÓN</a></li>
```

METODOS POST

Método **createUser ()**: Este método está anotado con `@PostMapping("/CrearUsuario")`, lo que significa que maneja las solicitudes POST a la ruta /api/users/ CrearUsuario. Recoge todos los datos que le llegan desde la vista, y establece al usuario como “Sin plan” ya que el siguiente panel es el de selección del plan, la imagen de perfil también establece una por defecto, y luego en el panel del perfil de usuario ya la puede cambiar. Una vez obtenidos todos los datos crea en la base de datos la nueva entidad de usuario y crea la sesión del usuario y envía al usuario directamente al panel de selección de plan de suscripción.

```

@PostMapping("/CrearUsuario")
    public ResponseEntity<String> createUser(@RequestParam String Name, @RequestParam String nickname,
@RequestParam String apellido, @RequestParam String pswd, @RequestParam String mail, @RequestParam LocalDate
FechaNacimiento, HttpSession session) {

    boolean registrado = userService.estaRegistrado(mail);
    if(registrado){
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("El usuario ya esta registrado");
    } else {
        User user = new User();
        user.setEmail(mail);
        user.setApellidos(apellido);
        user.setNombre(Name);
        user.setNickname(nickname);
        user.setContrasenha(pswd);
        user.setFechaNacimiento(FechaNacimiento);
        user.setPlanSuscripcion("Sin Plan");
        user.setPagoValidado(false);
        user.setImage_perfil("https://static.turbosquid.com/Preview/001292/481/WV/_D.jpg");
        User createdUser = userService.createUser(user);
        session.setAttribute("user", createdUser);
        return ResponseEntity.status(HttpStatus.OK).body("Usuario creado");
    }
}

```

Dentro de la vista por un lado está la gestión con Ajax [17], ya que desde el controlador enviamos mensajes Http ok o error no como en otras ocasiones que cogemos la ruta directamente.

```
$.ajax({  
    url: '/api/users/CrearUsuario',  
    type: 'POST',  
    data: {  
        'Name': nombre,  
        'nickname': nickname,  
        'pswd': contraseña,  
        'apellido': apellido,  
        'mail': emailRegistro,  
        'FechaNacimiento': fechaNacimiento  
    },  
    success: function(response) {  
        // Si la autenticación es exitosa, redirige al usuario a la API de películas  
        window.location.href = '/api/users/planSuscripcion';  
    },  
    error: function(jqXHR, textStatus, errorThrown) {  
        // Si la autenticación falla, muestra el mensaje de error al usuario  
        if(jqXHR.status == 401) { // 401 es el código de estado para no autorizado  
            document.getElementById('error-usuario-registrado').textContent = 'El correo ya está  
registrado';  
        } else {  
            document.getElementById('error-usuario-registrado').textContent = 'Error desconocido.  
Por favor, inténtelo de nuevo más tarde.';  
        }  
    }  
});
```

Contemplamos la posibilidad de que el correo ya exista, en ese caso enviamos a la vista el mensaje de error, y si dentro del controlador todas las gestiones se han producido correctamente enviamos a la sección de selección del plan de suscripción.

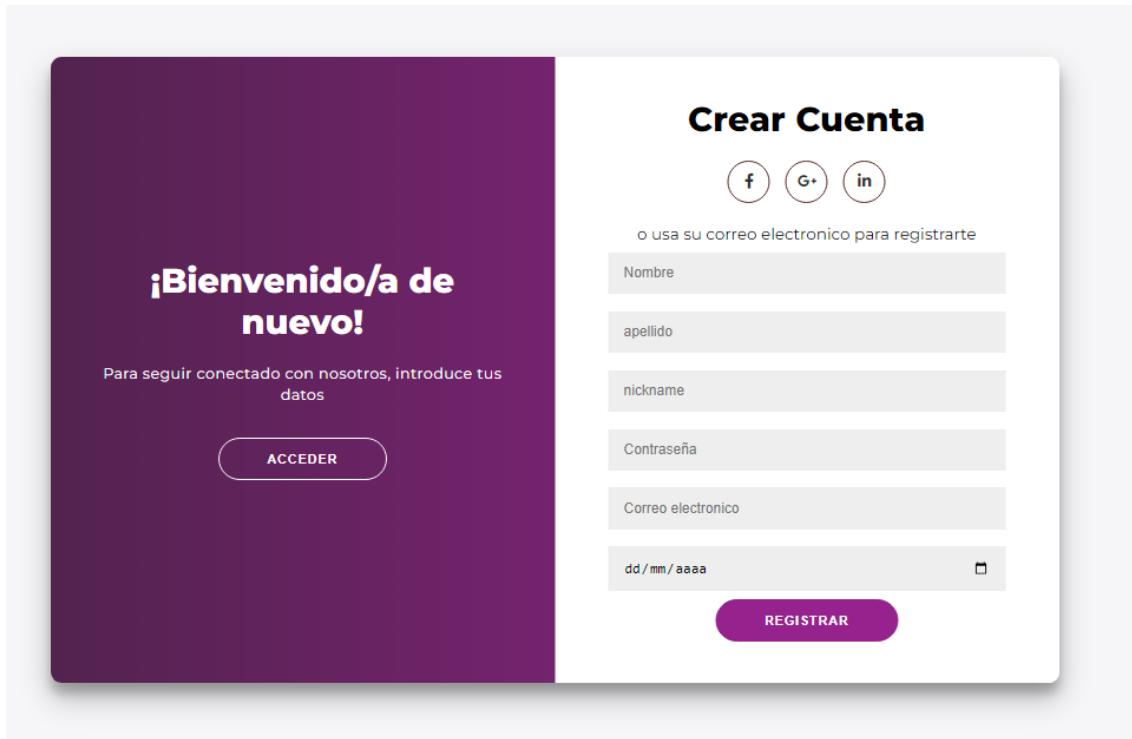


Figura 48: Panel que implementa el método createUser.

Dentro de la vista enviamos los datos al controlador con el método post.

```
<div class="form-container sign-up-container">
    <form th:action="@{/api/users/CrearUsuario}" method="post">
        <h1>Crear Cuenta</h1>
        <div class="social-container">
            <a href="#" class="social"><i class="fab fa-facebook-f"></i></a>
            <a href="#" class="social"><i class="fab fa-google-plus-g"></i></a>
            <a href="#" class="social"><i class="fab fa-linkedin-in"></i></a>
        </div>
        <span>o usa su correo electronico para registrarte</span>
        <input type="text" placeholder="Nombre" id="Name" name="Name"/>
        <span id="error-nombre" class="error"></span>
        <input type="text" placeholder="Apellido" id="apellido" name="apellido"/>
        <span id="error-apellido" class="error"></span>
        <input type="text" placeholder="nickname" id="nickname" name="nickname"/>
        <span id="error-nickname" class="error"></span>
        <input type="password" placeholder="Contraseña" id="pswd" name="pswd"/>
        <span id="error-contrasena" class="error"></span>
        <input type="email" placeholder="Correo electronico" id="mail" name="mail"/>
        <span id="error-email" class="error"></span>
        <input type="date" placeholder="FechaNacimiento" id="FechaNacimiento" name="FechaNacimiento"/>
        <span id="error-fecha" class="error"></span>
        <span id="error-usuario-registrado" class="error"></span>
        <button type="submit" id="Registrar">Registrar</button>
    </form>
</div>
```

Y en todo momento recogemos un campo para mensajes de error que van a llegar desde el js, por casos de que el contenido este vacío, o que por ejemplo en el correo no esté la arroba, o la fecha no sea en el formato adecuado.

```

var emailPattern = /^[^@\s]+@[^\s]+\.[^\s]+$/;
var fechaPattern = /^(\d{4})-(\d{2})-(\d{2})$/; // Define un patrón de fecha (AAAA-MM-DD)

var isValid = true; // Variable para verificar si todos los campos son válidos
// Verificar si el nombre está lleno
if (nombre.trim() === '') {
    document.getElementById('error-nombre').textContent = 'Por favor, ingresa tu nombre';
    isValid = false;
} else {
    document.getElementById('error-nombre').textContent = '';
}

// Verificar si el nickname está lleno
if (nickname.trim() === '') {
    document.getElementById('error-nickname').textContent = 'Por favor, ingresa un nickname';
    isValid = false;
} else {
    document.getElementById('error-nickname').textContent = '';
}

// Verificar si el apellido está lleno
if (apellido.trim() === '') {
    document.getElementById('error-apellido').textContent = 'Por favor, ingresa un apellido';
    isValid = false;
} else {
    document.getElementById('error-apellido').textContent = '';
}

// Verificar si el correo electrónico de registro es válido
if (!emailRegistro.match(emailPattern)) {
    document.getElementById('error-email').textContent = 'Por favor, ingresa un correo electrónico';
    isValid = false;
} else {
    document.getElementById('error-email').textContent = '';
}

```

Método **authenticateUser()**: Este método está anotado con `@PostMapping("/AccederUsuario")`, lo que significa que maneja las solicitudes POST a la ruta /api/users/AccederUsuario. Se centra en lo que es el inicio de sesión puro recoge el usuario y la contraseña y contempla todo tipo de posibilidades, en primer lugar, que el usuario este sin plan, luego se le envía al panel pertinente, también la posibilidad de que el usuario no haya validado el pago, de nuevo se envía al panel de suscripción pertinente. Otra de las posibilidades es que el usuario no este registrado por lo que se le envía directamente al panel de registro y la última posibilidad es que el usuario y contraseña no sean validos o no coincidan.

```
@PostMapping("/AccederUsuario")
```

```

public ResponseEntity<String> authenticateUser(@RequestParam("mail-2") String email, @RequestParam("pswd-2") String password, HttpSession session) {
    Optional<User> userOptional = userService.getUserByEmail(email);
    if (userOptional.isPresent()) {
        User user = userOptional.get();
        if (userService.authenticateUser(email, password)) {
            if (user.getPlanSuscripcion().equals("Sin Plan")) {
                // Si el usuario no tiene un plan de suscripción, crear una sesión y redirigir a la página
                // de actualización de plan
                session.setAttribute("user", user);
                return ResponseEntity.status(HttpStatus.OK).body("Sin plan de suscripción, por favor
selecciona uno.");
            } else if(user.getPagoValidado().equals(false)){
                session.setAttribute("user", user);
                return ResponseEntity.status(HttpStatus.OK).body("Pago no validado, por favor valida tu
pago.");
            } else {
                // El usuario tiene un plan de suscripción, redirigir a la página de películas
                session.setAttribute("user", user);
                return ResponseEntity.status(HttpStatus.OK).body("Autenticación exitosa");
            }
        } else {
            // Autenticación fallida, devuelve un mensaje de error
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Usuario o contraseña
incorrectos");
        }
    } else {
        // Usuario no encontrado, devuelve un mensaje de error
        return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Usuario no encontrado");
    }
}

```

Inicio de Sesión



o usa tu cuenta

Correo electrónico

Contraseña

[¿Olvidaste tu contraseña?](#)

ACCEDER

Figura 49: Panel que implementa el método authenticateUser.

Todo el manejo de estos posibles errores se maneja de nuevo con Ajax [17], desde el controlador se envía la acción http según el caso que se dé a la hora de iniciar sesión con Ajax [17] se recoge y se envían los errores si los hay y si el usuario y la contraseña son correctos da acceso a la aplicación para ver el contenido.

```

$.ajax({
    url: '/api/users/AccederUsuario',
    type: 'POST',
    data: { 'mail-2': emailAcceso, 'pswd-2': contrasena },
    success: function(response) {
        if (response === "Autenticación exitosa") {
            // Si la autenticación es exitosa, redirigir al usuario a la API de películas
            window.location.href = '/api/peliculas/';
        } else if (response === "Sin plan de suscripción, por favor selecciona uno.") {
            // Si el usuario no tiene un plan de suscripción, redirigir a la página de actualización
            // de plan
            window.location.href = '/api/users/planSuscripcion';
        } else if(response == "Pago no validado, por favor valida tu pago."){
            // Si el usuario no ha validado su pago, redirigir a la página de validación de pago
            window.location.href = '/api/pago/';
        } else {
            // Otro caso de respuesta (como error de autenticación)
            document.getElementById('error-contraseña-acceso').textContent = response;
        }
    },
    error: function(jqXHR, textStatus, errorThrown) {
        // Si la autenticación falla, muestra el mensaje de error al usuario
        var responseText = jqXHR.responseText;
        if(responseText === "Usuario o contraseña incorrectos") { // 401 es el código de estado para
            // no autorizado
            document.getElementById('error-contraseña-acceso').textContent = 'Usuario o contraseña
            incorrectos';
        } else if(responseText === "Usuario no encontrado"){
            document.getElementById('error-contraseña-acceso').textContent = 'Por favor registrese.';
        } else{
            document.getElementById('error-contraseña-acceso').textContent = 'Error desconocido. Por
            favor, inténtelo de nuevo más tarde.';
        }
    }
});

```

Por parte de la vista solo se envían los campos al controlador y se incluyen los campos de los mensajes de errores.

```

<form th:action="@{/api/users/AccederUsuario}" method="post">
    <h1>Inicio de Sesión</h1>
    <div class="social-container">
        <a href="#" class="social"><i class="fab fa-facebook-f"></i></a>
        <a href="#" class="social"><i class="fab fa-google-plus-g"></i></a>

```

```

        <a href="#" class="social"><i class="fab fa-linkedin-in"></i></a>
    </div>
    <span>o usa tu cuenta</span>
    <input type="email" placeholder="Correo electronico" id="mail-2" name="mail-2"/>
    <span id="error-email-acceso" class="error"></span>
    <input type="password" placeholder="Contrasena" id="pswd-2" name="pswd-2"/>
    <span id="error-contrasena-acceso" class="error"></span>
    <a href="#">¿Olvidaste tu contrasena?</a>
    <button type="submit" id="Acceder">Acceder</button>
</form>

```

Método **updatePlan()**: Este método está anotado con `@PostMapping("/updatePlan")`, lo que significa que maneja las solicitudes POST a la ruta `/api/users/updatePlan`. Este método sirve tanto para el panel de inicio de sesión para la selección de un plan como para el panel del perfil de usuario por si quiere contratar un plan mejor. Hay que resaltar una cosa y es que el pago es único es decir que el plan que seleccionas te quedas con el siempre no es renovación mensual, lo único que sí puedes hacer es actualizar el plan a uno superior cuando deseas. Si seleccionas un plan inferior o igual al que tienes no te va a dejar. Lo primero que hace es obtener mediante la sesión del usuario el plan actual que en este caso va a ser “Sin Plan” y en base a la opción que el usuario decida, se le va a actualizar el plan. Una vez que se actualice el o se seleccione un plan por primera vez, se va a enviar al panel del pago.

```

@PostMapping("/updatePlan")
public RedirectView updatePlan(@RequestParam String plan, HttpSession session) {
    User user = (User) session.getAttribute("user");
    switch (user.getPlanSuscripcion()) {
        case "Pro" -> {
            return new RedirectView("/api/userProfile/");
        }
        case "Basico" -> {
            if (plan.equals("Gratis") || plan.equals("Basico")) {
                return new RedirectView("/api/userProfile/");
            }
        }
        case "Gratis" -> {
            if (plan.equals("Gratis")) {
                return new RedirectView("/api/userProfile/");
            }
        }
    }
    user.setPlanSuscripcion(plan);
    userService.updateUser(user.getEmail(), user);

    if (plan.equals("Gratis")) {
        user.setPagoValidado(true);
        userService.updateUser(user.getEmail(), user);
        session.invalidate();
        return new RedirectView("/api/users/");
    } else {

```

```

        return new RedirectView("/api/pago/");
    }
}

```

Desde la vista el usuario vera los 3 planes disponibles y las condiciones que ofrece cada plan y elegirá uno, si es para actualizarlo saldrá la misma vista solo que solo dejará acceder a un plan superior al actual para llegar al pago.



Figura 50: Panel que implementa el método updatePlan.

Desde la vista para la selección del plan de suscripción tenemos la clase PlanSuscripcion.html que envía al controlador el plan de suscripción seleccionado y ya este gestiona la acción.

```

<div class="cards__inner">
    <div class="cards__card card">
        <h2 class="card__heading">Gratis</h2>
        <p class="card__price">€0.00€</p>
        <ul role="list" class="card__bullets flow">
            <li>Acceso a todo el contenido multimedia de la plataforma</li>
            <li>No se pueden hacer comentarios de los contenidos</li>
            <li>No se puede participar en las comunidades de usuarios</li>
            <li>No se puede ver contenido en directo exclusivo</li>
            <li>Un único dispositivo</li>
        </ul>
        <form action="/api/users/updatePlan" method="post" class="card__cta cta">
            <input type="hidden" name="plan" value="Gratis">
            <a href="#basic" onclick="event.preventDefault(); this.parentNode.submit();">Get Started</a>
        </form>
    </div>
    <div class="cards__card card">
        <h2 class="card__heading">Basico</h2>

```

```
<p class="card__price">1.99€</p>
<ul role="list" class="card__bullets flow">
    <li>Acceso a todo el contenido multimedia de la plataforma</li>
    <li>Acceso a los comentarios del contenido multimedia</li>
    <li>Acceso a las comunidades de usuarios</li>
    <li>No puede ver contenido exclusivo en directo</li>
    <li>Tres dispositivos</li>
</ul>
<form action="/api/users/updatePlan" method="post" class="card__cta cta">
    <input type="hidden" name="plan" value="Basico">
    <a href="#pro" onclick="event.preventDefault(); this.parentNode.submit()">Upgrade to
Basic</a>
</form>
</div>
```

Controlador/Vista Tarjeta de crédito

La clase TarjetaCreditoController maneja el pago en la aplicación, es un pago europeo con nombre y apellidos, número de tarjeta, fecha de caducidad y código de seguridad. No es un pago real, ojalá. Simplemente dentro del usuario hay una variable booleana llamada pagoValidado y cuando se realice el pago correctamente ponemos esa variable a true.

Esta clase se encuentra en el archivo:

src/main/java/com/example/cursospringboot/controller/TarjetaCreditoController.java.

La clase TarjetaCredito en la vista se divide en su parte de html [1] donde desde la vista le pasamos los datos al controlador y luego en el js recogemos el diseño de los diferentes tipos de tarjeta.

Estas clases se encuentran en el archivo:

src/main/resources/templates/tarjetaCredito.html

src/main/resources/js/tarjetaCredito.js

Definimos la ruta del controlador

```
@RequestMapping("/api/pago")
```

MÉTODOS GET

Método **root ()**: Este método está anotado con `@GetMapping("/")`, lo que significa que maneja las solicitudes GET a la ruta `/api/pago/`. Que simplemente comprueba si hay una sesión del usuario ya creada, es decir que ya se ha registrado y envía a la vista de `tarjetaCredito.html`.

```
@GetMapping("/")
public String root(HttpServletRequest session) {
    if (session.getAttribute("user") == null) {
        return "login";
    }
    return "tarjetaCredito";
}
```

METODOS POST

Método **procesarPago ()**: Este método está anotado con `@PostMapping("/ProcesarPagoServlet")`, lo que significa que maneja las solicitudes POST a la ruta `/api/pago/ ProcesarPagoServlet`. Recogemos los campos que nos llegan desde la vista, y cuando estén todos los datos correctos y se haya guardado en la base de datos la tarjeta de crédito, ponemos la variable del usuario de pago validado a true.

```
@PostMapping("/ProcesarPagoServlet")
public RedirectView procesarPago(@RequestParam("titular") String titular,
                                @RequestParam("numeroT") String numeroTarjeta,
                                @RequestParam("fechaCaducidad") String fechaCaducidad,
                                @RequestParam("codigoSeguridad") String codigoSeguridad, HttpSession
session) {
    User user = (User) session.getAttribute("user");
    if (user == null) {
        return new RedirectView("/api/users/");
    }
    TarjetaCredito tj = new TarjetaCredito();
    tj.setTitular(titular);
    tj.setNumeroT(numeroTarjeta);
    tj.setFechaCaducidad(fechaCaducidad);
    tj.setCodigoSeguridad(codigoSeguridad);
    tj.setUser(user);
    //Si se ha podido crear la tarjeta se valida el pago del usuario
    if (user.getPagoValidado().equals(false)) {
        tarjetaCreditoService.createTarjeta(tj);
        user.setPagoValidado(true);
        userService.updateUser(user.getEmail(), user);
    }
    session.invalidate();
}

return new RedirectView("/api/users/");
}
```

Por parte de la vista le pasamos los datos al controlador a la ruta del controlador

```
<form action="/api/pago/ProcesarPagoServlet" method="post">  
    <div class="form-container">  
        <div class="field-container">  
            <label for="name">Nombre del titular</label>  
            <input id="name" name="titular" maxlength="20" type="text" required>  
        </div>  
        <div class="field-container">  
            <label for="cardnumber">Número de tarjeta</label><span id="generatecard">Selecciona tipo de  
tarjeta</span>  
            <input id="cardnumber" name="numeroT" type="text" pattern="\d{4}\s?\d{4}\s?\d{4}\s?\d{4}"  
                inputmode="numeric">  
            <img alt="Credit card icon" data-bbox="225 315 475 375" />  
        </div>  
        <div class="field-container">  
            <label for="expirationdate">Fecha de caducidad (mm/yy)</label>  
            <input id="expirationdate" name="fechaCaducidad" type="text" pattern="[0-9]{2}/[0-9]{2}"  
                inputmode="numeric"  
                required="">  
        </div>  
        <div class="field-container">  
            <label for="securitycode">Código de seguridad</label>  
            <input id="securitycode" name="codigoSeguridad" type="text" pattern="[0-9]*" inputmode="numeric"  
                required>  
        </div>  
        <button type="submit">Procesar pago</button>  
    </div>  
</form>
```



Figura 51: Panel que implementa el método `procesarPago`.

Controlador/Vista Perfil de Usuario

Es un espacio personal del usuario donde puede ver su información más personal y tiene la posibilidad de actualizar datos sobre su cuenta, como el nickname, cambiar la contraseña, cambiar la imagen de perfil que va a ser vista por el resto de usuarios en las comunidades de chat y también permite al usuario actualizar su plan de suscripción a una versión mayor. La clase `UserProfileController` recoge los datos que le pasa la vista de los nuevos valores de los campos para actualizar el modelo.

Esta clase se encuentra en el archivo:

`src/main/java/com/example/cursospringboot/controller/ UserProfileController.java`.

La clase userProfile es la encargada de la vista por parte del panel de perfil del usuario en este caso para la gestión de datos personales del usuario, de contraseñas, correos o planes de suscripción, el diseño es con el uso de bootstrap [8] mediante enlace.

```
<link rel='stylesheet'  
      href='https://themes.getbootstrap.com/wp-content/themes/bootstrap-  
marketplace/style.css?ver=1590611604' />
```

Estas clases se encuentran en el archivo:

[src/main/resources/templates/userProfile.html](#)

Definimos la ruta del controlador

```
@RequestMapping("/api/userProfile")
```

Para acceder al perfil de usuario desde el panel de la aplicación en la opción de PERFIL.

```
<li class="nav-item"><a th:href="@{/api/userProfile}" class="nav-link">PERFIL</a></li>
```

Si accedes al perfil sin estar registrado, la aplicación te envía directamente al registro.

```
if (user == null || user.getPagoValidado().equals(false) || "Sin Plan".equals(user.getPlanSuscripcion())) {  
    return "login"; // Redirect the user to the login page if not authenticated  
}
```

MÉTODOS GET

Método **showProfile (Model model, HttpSession session)**: Este método está anotado con `@GetMapping("/")`, lo que significa que maneja las solicitudes GET a la ruta `/api/userProfile/`. Este método simplemente envía al panel de userProfile.html si hay una sesión de usuario ya creado, es decir si se ha registrado o ha iniciado sesión en caso de que este registrado y si es así pasa esa sesión del usuario a la vista para mostrar sus datos.

Método **updateProfilePicture (HttpSession session)**: Este método está anotado con `@GetMapping("/updateProfilePicture")`, lo que significa que maneja las solicitudes GET a la ruta `/api/userProfile/updateProfilePicture`. Simplemente envía al panel profileImage.html donde el usuario puede actualizar su foto de perfil.

MÉTODOS POST

Método **updateInfo ()**: Este método está anotado con `@PostMapping("/updateInfo")`, lo que significa que maneja las solicitudes POST a la ruta `/api/users/updateInfo`.

```
public String updateInfo(@RequestParam("nombre") String nombre, @RequestParam("apellidos")
String apellidos, @RequestParam("nickname") String nickname, HttpSession session,
RedirectAttributes redirectAttributes)
```

Este método recoge desde la vista el nombre, apellidos y nickname del usuario para actualizarlos en el caso de que el usuario lo deseé, una de las condiciones que implementa el método es que el nuevo nickname no puede coincidir con el de otros usuarios, si todo se ha implementado correctamente se envía mensaje de acción correcta y si se produce algún error como que el nickname ya existe se también se notifica con el uso de errores, usando redirectAttributes. En este caso cogemos los atributos que llegan desde la vista con RequestParam y lo llamamos a la derecha como queramos para darle uso dentro del controlador.

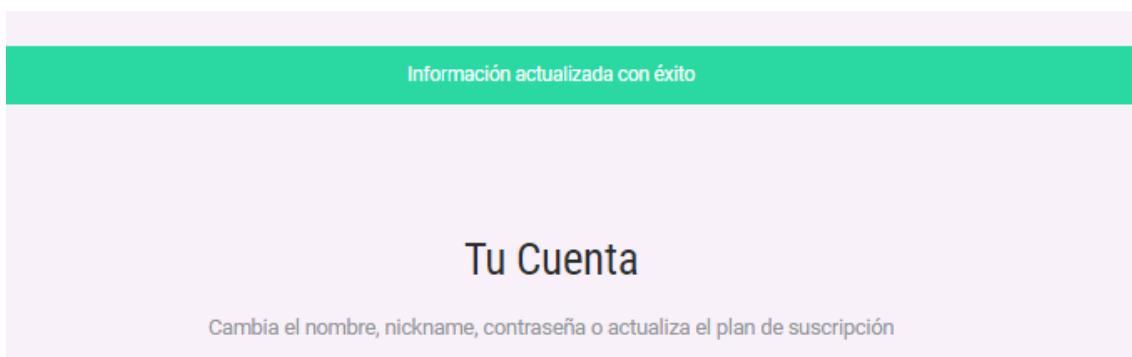


Figura 52: Panel que implementa errores redirectAttributes.

```
@PostMapping("/updateInfo")
public String updateInfo(@RequestParam("nombre") String nombre, @RequestParam("apellidos") String
apellidos, @RequestParam("nickname") String nickname, HttpSession session, RedirectAttributes
redirectAttributes) {
    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
    }
    if (nickname != null && !nickname.equals(user.getNickname())) {
        if (userService.nicknameExists(nickname)) {
            redirectAttributes.addFlashAttribute("errorMessage", "El nickname ya está en uso");
            return "redirect:/api/userProfile/";
        }
        user.setNickname(nickname);
    }
    user.setNombre(nombre);
    user.setApellidos(apellidos);
    try {
        userService.updateUser(user.getEmail(), user);
        redirectAttributes.addFlashAttribute("successMessage", "Información actualizada con éxito");
    } catch (Exception e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar la información");
    }
}
```

```

    }
    return "redirect:/api/userProfile/";
}

```

Por parte de la vista, con el uso de thymeleaf [2] [6] y gracias a la información del usuario que le pasamos desde el controlador, mostramos toda la información del usuario y dejamos también margen a actualizar el campo si se desea.

```

<div class="container container--xs">
    <h1 class="mb-1 text-center">Tu Cuenta</h1>
    <p class="fs-14 text-gray text-center mb-5">Cambia el nombre, nickname, contraseña  

        actualiza el plan de suscripción</p>
    <form class="woocommerce-EditAccountForm edit-account" action="/api/userProfile/updateInfo" method="post">
        <div class="form-group row">
            <div class="col-sm-6 mb-4 mb-sm-0">
                <label for="nombre">Nombre <span class="required">*</span></label>
                <input type="text" class="form-control woocommerce-Input woocommerce-Input--text input-text" name="nombre" id="nombre" th:value="${user.nombre}"/>
            </div>
            <div class="col-sm-6">
                <label for="apellidos">Apellidos <span class="required">*</span></label>
                <input type="text" class="form-control woocommerce-Input woocommerce-Input--text input-text" name="apellidos" id="apellidos" th:value="${user.apellidos}"/>
            </div>
        </div>
        <div class="form-group">
            <label for="nickname">Nickname <span class="required">*</span></label>
            <input type="text" class="form-control woocommerce-Input woocommerce-Input--email input-text" name="nickname" id="nickname" th:value="${user.nickname}"/>
        </div>
        <input type="submit" class="btn btn-outline-brand btn-block mb-4" name="save_account_details" value="Actualizar Información"/>
    </form>

```

Tu Cuenta

Cambia el nombre, nickname, contraseña o actualiza el plan de suscripción

Nombre *	Apellidos *
<input type="text" value="Luna"/>	<input type="text" value="Rodriguez Viejo"/>
Nickname *	
<input type="text" value="luna33"/>	
<input type="button" value="Actualizar Información"/>	
<hr/>	
Email *	Fecha de Nacimiento *
<input type="text" value="luna@gmail.com"/>	<input type="text" value="02/12/2002"/>

Figura 53: Panel que implementa el método updateInfo.

Método **updatePassword ()**: Este método está anotado con `@PostMapping("/updatePassword")`, lo que significa que maneja las solicitudes POST a la ruta /api/users / updatePassword.

```
public String updatePassword(@RequestParam String password_current, @RequestParam String
password_1, @RequestParam String password_2, HttpSession session, RedirectAttributes
redirectAttributes )
```

Este método se encarga de actualizar la contraseña del usuario en caso de que este lo dese, se encuentra dentro de la clase UserProfileController. Recoge desde la vista tres campos para la implementación de mayor seguridad que son el de actual contraseña, nueva contraseña y confirmar nueva contraseña. Mediante el uso de redirectAttributes vamos notificando a la vista de la evolución del proceso de actualizar la contraseña con mensajes de los posibles errores que vayan surgiendo o si todo se ha realizado correctamente igual.



Figura 54: Panel 2 que implementa errores redirectAttributes.

```

@PostMapping("/updatePassword")
public String updatePassword(@RequestParam String password_current, @RequestParam String password_1,
@RequestParam String password_2, HttpSession session, RedirectAttributes redirectAttributes) {
    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
    }
    else if (password_current.trim().isEmpty() || password_1.trim().isEmpty() ||
password_2.trim().isEmpty()) {
        redirectAttributes.addFlashAttribute("errorMessage", "Por favor complete todos los campos de
contraseña");
        return "redirect:/api/userProfile/";
    }
    else if (!userService.authenticateUser(user.getEmail(), password_current)) {
        redirectAttributes.addFlashAttribute("errorMessage", "Contraseña actual incorrecta");
        return "redirect:/api/userProfile/";
    }
    else if (!password_1.equals(password_2)) {
        redirectAttributes.addFlashAttribute("errorMessage", "Las contraseñas no coinciden");
        return "redirect:/api/userProfile/";
    }
    else {
        user.setContrasenya(password_1);
        userService.updateUser(user.getEmail(), user);
        redirectAttributes.addFlashAttribute("successMessage", "Contraseña actualizada con éxito");
        return "redirect:/api/userProfile/";
    }
}

```

Por parte de la vista pasamos al controlador tres campos con las contraseñas actuales y las nuevas para que este las maneje y las actualice si es necesario, en este caso no se hace uso de thymeleaf [2] [6] ya que los 3 campos están vacíos y ya se gestionan desde el controlador.

```

<form class="woocommerce-EditAccountForm edit-account"
      action="/api/userProfile/updatePassword" method="post">

```

```

<!-- Rest of the form fields -->
<div class="form-group">
    <label for="password_current">Contraseña Actual </label>
    <input type="password"
        class="form-control woocommerce-Input woocommerce-Input--password
input-text"
        name="password_current" id="password_current"/>
</div>
<div class="form-group">
    <label for="password_1">Nueva Contraseña </label>
    <input type="password"
        class="form-control woocommerce-Input woocommerce-Input--password
input-text"
        name="password_1" id="password_1"/>
</div>
<div class="form-group">
    <label for="password_2">Confirmar nueva contraseña</label>
    <input type="password"
        class="form-control woocommerce-Input woocommerce-Input--password
input-text"
        name="password_2" id="password_2"/>
</div>
<input type="submit" class="btn btn-outline-brand btn-block mb-4"
        name="save_account_details" value="Actualizar Contraseña"/>
</form>

```



The screenshot shows a user interface for updating a password. It consists of three input fields labeled "Contraseña Actual", "Nueva Contraseña", and "Confirmar nueva contraseña". Below these fields is a large blue button labeled "Actualizar Contraseña".

Figura 55: Panel que implementa el método `updatePassword`.

Método **updatePlan (@RequestParam String plan, HttpSession session)**: Este método está anotado con **@PostMapping("/updatePlan")**, lo que significa que maneja las solicitudes POST a la ruta /api/users/updatePlan.

Este método sirve tanto para el panel de inicio de sesión para la selección de un plan como para el panel del perfil de usuario por si quiere contratar un plan mejor. Hay que resaltar una cosa y es que el pago es único es decir que el plan que seleccionas te quedas con él siempre no es renovación mensual, lo único que si puedes hacer es actualizar el plan a uno superior cuando desees. Si seleccionas un plan inferior o igual al que tienes no te va a dejar. Lo primero que hace es obtener mediante la sesión del usuario el plan actual que en este caso va a ser “Sin Plan” y en base a la opción que el usuario decida, se le va a actualizar el plan. Una vez que se actualice el o se seleccione un plan por primera vez, se va a enviar al panel del pago.

```
@PostMapping("/updatePlan")
public RedirectView updatePlan(@RequestParam String plan, HttpSession session) {
    User user = (User) session.getAttribute("user");
    switch (user.getPlanSuscripcion()) {
        case "Pro" -> {
            return new RedirectView("/api/userProfile/");
        }
        case "Basico" -> {
            if (plan.equals("Gratis") || plan.equals("Basico")) {
                return new RedirectView("/api/userProfile/");
            }
        }
        case "Gratis" -> {
            if (plan.equals("Gratis")) {
                return new RedirectView("/api/userProfile/");
            }
        }
    }
    user.setPlanSuscripcion(plan);
    userService.updateUser(user.getEmail(), user);

    if (plan.equals("Gratis")) {
        user.setPagoValidado(true);
        userService.updateUser(user.getEmail(), user);
        session.invalidate();
        return new RedirectView("/api/users/");
    } else {

        return new RedirectView("/api/pago/");
    }
}
```

Desde la vista el usuario verá los 3 planes disponibles y las condiciones que ofrece cada plan y elegirá uno, si es para actualizarlo saldrá la misma vista solo que solo dejará acceder a un plan superior al actual para llegar al pago.

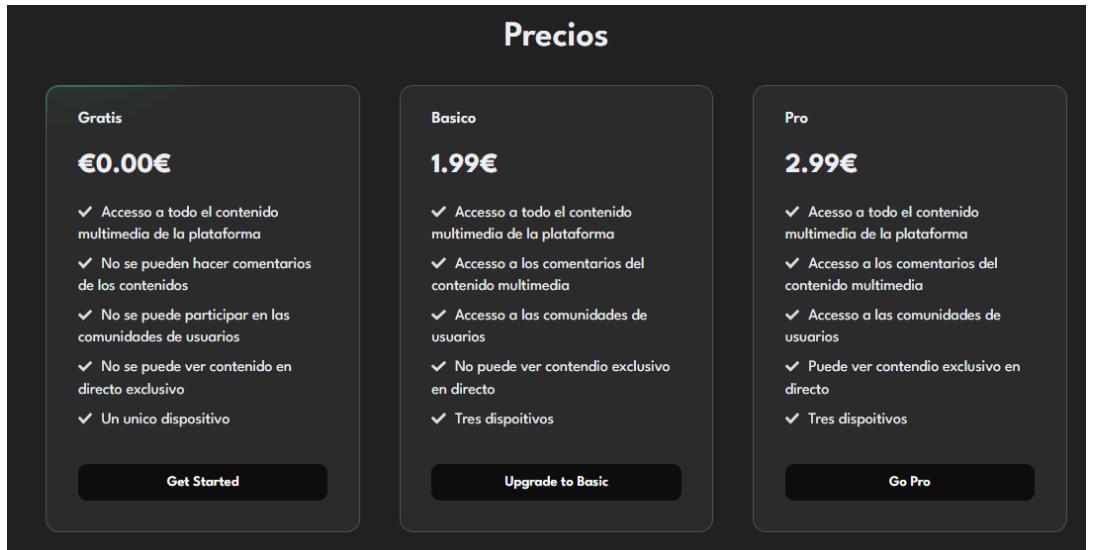


Figura 56: Panel que implementa el método updatePlan.

Desde la vista para la selección del plan de suscripción tenemos la clase PlanSuscripcion.html que envía al controlador el plan de suscripción seleccionado y ya este gestiona la acción.

```
<div class="cards__inner">
  <div class="cards__card card">
    <h2 class="card__heading">Gratis</h2>
    <p class="card__price">€0.00€</p>
    <ul role="list" class="card__bullets flow">
      <li>Acceso a todo el contenido multimedia de la plataforma</li>
      <li>No se pueden hacer comentarios de los contenidos</li>
      <li>No se puede participar en las comunidades de usuarios</li>
      <li>No se puede ver contenido en directo exclusivo</li>
      <li>Un unico dispositivo</li>
    </ul>
    <form action="/api/users/updatePlan" method="post" class="card__cta cta">
      <input type="hidden" name="plan" value="Gratis">
      <a href="#basic" onclick="event.preventDefault(); this.parentNode.submit();">Get Started</a>
    </form>
  </div>
  <div class="cards__card card">
    <h2 class="card__heading">Basico</h2>
    <p class="card__price">1.99€</p>
    <ul role="list" class="card__bullets flow">
      <li>Acceso a todo el contenido multimedia de la plataforma</li>
      <li>Acceso a los comentarios del contenido multimedia</li>
      <li>Acceso a las comunidades de usuarios</li>
      <li>No puede ver contenido exclusivo en directo</li>
      <li>Tres dispositivos</li>
    </ul>
    <form action="/api/users/updatePlan" method="post" class="card__cta cta">
      <input type="hidden" name="plan" value="Basico">
```

```

        <a href="#pro" onclick="event.preventDefault(); this.parentNode.submit();">Upgrade to
Basic</a>
</form>
</div>

```

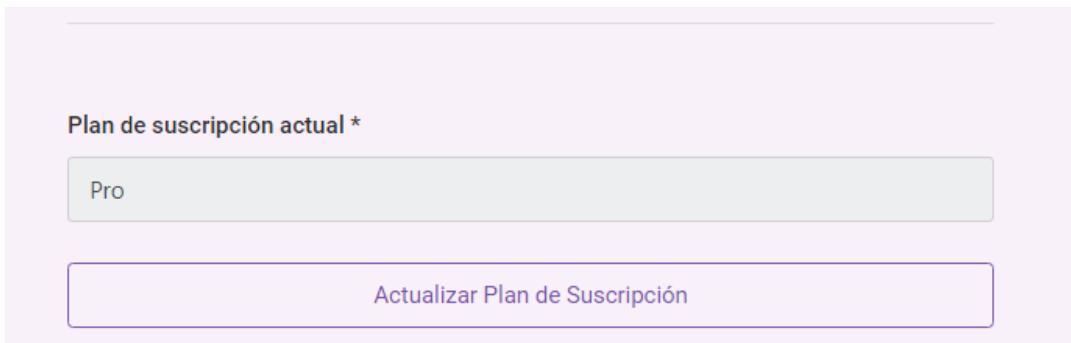


Figura 57: Panel que implementa el método `updatePlan UserProfile`.

Método `selectProfilePicture ()`: Este método está anotado con `@PostMapping (" /selectProfilePicture/{imageName}")`, lo que significa que maneja las solicitudes POST a la ruta `/api/userProfile/selectProfilePicture/{imageName}`.

```

public String selectProfilePicture(@PathVariable String imageName, HttpSession session,
RedirectAttributes redirectAttributes)

```

Este método se encarga de actualizar la imagen de perfil del usuario, que es la que se va a ver en las comunidades de chat, la propia aplicación implementa una serie de imágenes de perfil para que el usuario selecciona la que más le guste.

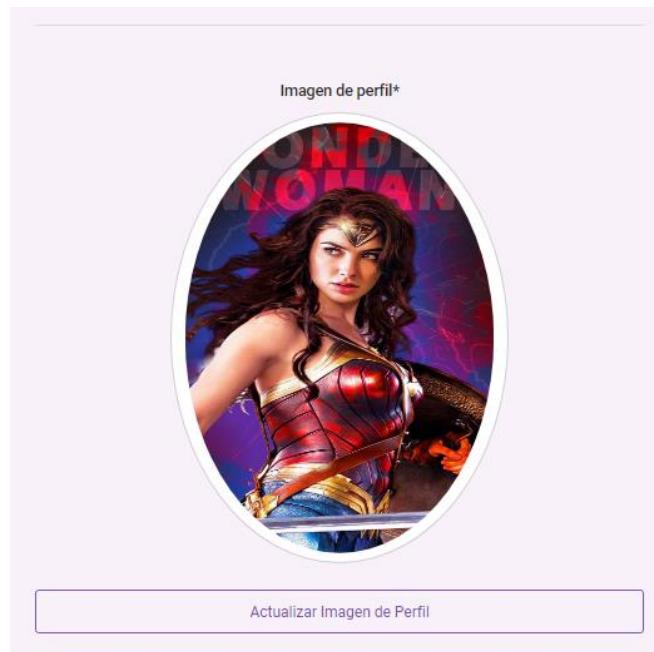


Figura 58: Panel que implementa el método `selectProfilePicture`.

```

@GetMapping("/selectProfilePicture/{imageName}")
    public String selectProfilePicture(@PathVariable String imageName, HttpSession session, RedirectAttributes
redirectAttributes) {
        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
        }
        user.setUrl_image_perfil("/images/Perfil/" + imageName);
        try {
            userService.updateUser(user.getEmail(), user);
            redirectAttributes.addFlashAttribute("successMessage", "Imagen de perfil actualizada con éxito");
        } catch (Exception e) {
            redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar la imagen de perfil");
        }
        return "redirect:/api/userProfile/";
    }
}

```

Por parte de la vista se envía al controlador el nombre de la imagen seleccionada, las imágenes se almacenan en la ruta src/main/resources/images/Perfil. Cuando el usuario selecciona una imagen de las 10 que hay se busca en la ruta esa imagen y se pasa como imagen de perfil.

```

<div class="card" data-color="red">
    <a th:href="@{/api/userProfile/selectProfilePicture/nano.png}">
        
        <div class="card-faders">
            
            
            
            
            
            
            
            
            
            
        </div>
    </a>
</div>
<div class="card" data-color="green">
    <a th:href="@{/api/userProfile/selectProfilePicture/ironMan.png}">
        
        <div class="card-faders">
            
            
            
            
            
            
            
            
            
            
        </div>
    </a>
</div>

```



Se sigue la misma metodología con las 10 imágenes. Donde se selecciona y hace el efecto de panel 3d.

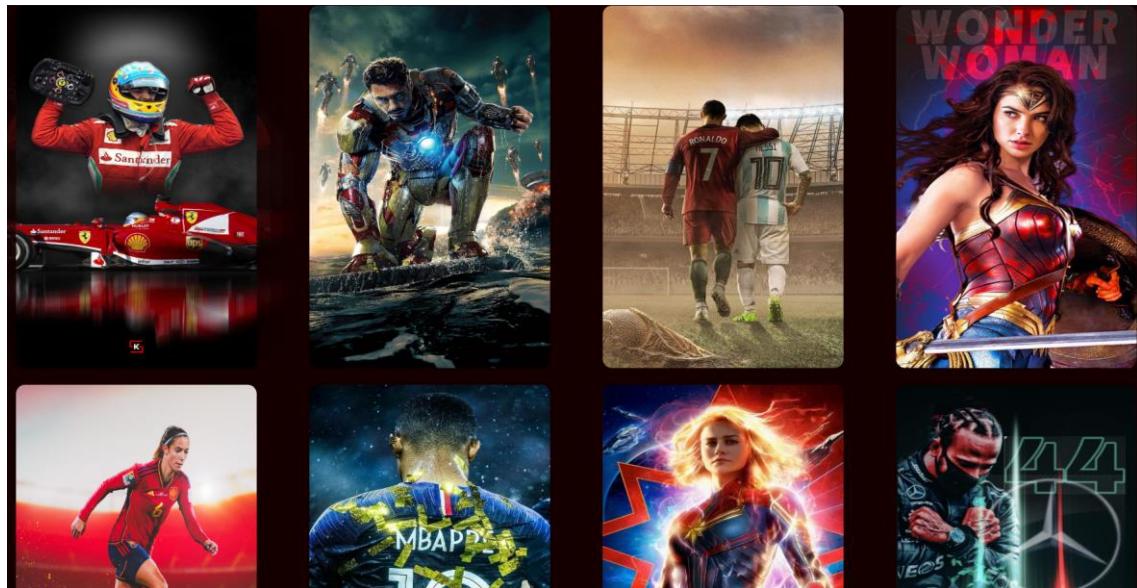


Figura 59: Panel de selección de imágenes de perfil.

8.2.4 Controladores/Vista Comunidades de Usuarios

Es la parte más compleja de la aplicación y a la vez la más interesante y completa, se trata de la gestión interna de las comunidades de chats de los usuarios de la aplicación, el controlador gestiona dos entidades a la vez por un lado las comunidades de usuarios y por otro lado los chats propios de cada comunidad. También se habilita el panel de administrador para la realización de operaciones CRUD con las comunidades.

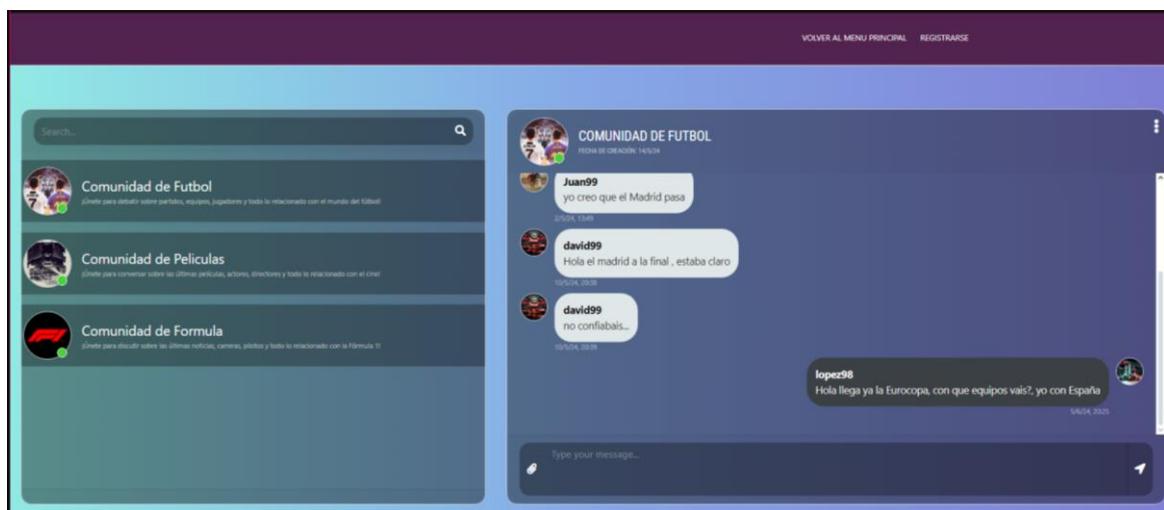


Figura 60: Panel de comunidades de Chats

Como se ve en la imagen a la izquierda está el panel desplegable donde se selecciona la comunidad y a la derecha los chats de cada comunidad, no hay restricciones en el sentido de tener que pertenecer en la comunidad, es decir que solo con estar dado de alta en la aplicación y tener el plan de suscripción básico o superior puedes participar en las comunidades de usuarios sin límites de mensajes ni restricciones, ya que siempre se busca la máxima comodidad de los usuarios, en caso de los usuarios con el plan gratis al igual que con los comentarios solo se pueden ver los chats pero no se pueden enviar mensajes para interaccionar con otros usuarios.

Controlador/Vista Inicio Comunidades/Chats Usuarios

La clase chatCommunityController gestiona todo lo relacionado con las comunidades y chats de la plataforma desde mostrar los chats, las comunidades habilitadas, la selección de las comunidades, recoger los mensajes de los usuarios, la gestión del panel de administrador para operaciones CRUD con las comunidades.

Esta clase se encuentra en el archivo:

```
src/main/java/com/example/cursospringboot/controller/ChatCommunityController.java.
```

La clase chat es la encargada de la vista por parte de las comunidades de usuarios, en este caso lo que es el diseño es recogido de una web donde el diseño ya estaba hecho, en la bibliografía se puede ver el enlace, la función de la vista en este caso es enviarle los chats al controlador para que estos los gestione, enviarle la comunidad que se selecciona para que el controlador muestre los mensajes de la comunidad o para las operaciones CRUD, informamos al controlador en todo momento con qué comunidad se está tratando para hacer la operación que el administrador deseé.

Esta clase se encuentran en el archivo:

```
src/main/resources/templates/ chat.html
```

Definimos la ruta del controlador

```
@RequestMapping("/api/ChatCommunity")
```

Desde los paneles de la vista para acceder desde el panel de la aplicación en la opción de COMUNIDADES.

```
<li class="nav-item"><a th:href="@{/api/ChatCommunity}" class="nav-link">COMUNIDADES</a></li>
```

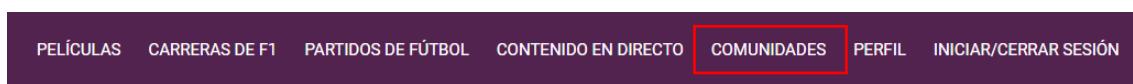


Figura 61: Panel de selección de opciones de la plataforma.

MÉTODOS GET

Definimos la ruta del controlador

```
@RequestMapping("/api/ChatCommunity")
```

Método **showChatCommunity ()**: Este método está anotado con `@GetMapping (""/", "/{communityName}")`, lo que significa que maneja las solicitudes GET a la ruta /api/ ChatCommunity/ o /api/ChatCommunity/communityName.

```
public String showChatCommunity(@PathVariable Optional<String> communityName, Model model,
HttpSession session)
```

Eso es porque la primera vez que entramos en la aplicación y vamos a la sección de comunidades de usuario no hay ninguna comunidad seleccionada como tal ósea que la ruta es /api/ChatCommunity/ en el momento que el usuario desde la vista selecciona una comunidad para poder tratar con ella usamos la ruta /api/ChatCommunity/communityName.

Lo que se pasa por parte del controlador a la vista cuando se accede a esa ruta del controlador es, por un lado, el rol del usuario ya que tenemos un panel de administración para operaciones con las comunidades, también se pasa todas las comunidades disponibles en la base de datos con los métodos de los repositorios, la comunidad seleccionada por el usuario dentro de la vista y los mensajes de la comunidad seleccionada. En el caso de que sea la primera vez que se entra como no ha dado tiempo a seleccionar una comunidad, se hace una función random con todas las comunidades de los usuarios para que entre por defecto a una y ya el usuario entre a la que quiera, esto es debido a que como he comentado antes están los dos paneles, tanto el del chat como el de selección de comunidades en el mismo fichero y es la única forma de hacerlo así.

```
@GetMapping("/{}/{}")
public String showChatCommunity(@PathVariable Optional<String> communityName, Model model, HttpSession
session) {
    User user = (User) session.getAttribute("user");
    if (user == null || user.getPagoValidado().equals(false)) {
        return "login"; // Redirect the user to the login page if not authenticated
    }
    if (user != null) {
        Set<Role> roles = user.getRoles();
        model.addAttribute("roles", roles.stream().map(Role::getName).collect(Collectors.toList()));
    }
    // Fetch the list of comments for the specific movie
    List<ChatCommunity> comunidades = chatCommunityService.getAllCommunities();
    model.addAttribute("comunidades", comunidades);
    //Selecciona una comunidad aleatoria
    String nombreAleatorio = comunidades.get((int) (Math.random() *
comunidades.size())).getNombreComunidad();
    // Fetch the community and the list of messages for the specific community
    String communityNameToUse = communityName.orElse(nombreAleatorio);
    ChatCommunity community = chatCommunityService.getCommunityByName(communityNameToUse);
    List<ChatMessage> allMessages = chatMessageService.getMessagesByCommunity(community);
    model.addAttribute("comunidadSeleccionada", community);
    model.addAttribute("AllMessages", allMessages);
    model.addAttribute("user", user); // Add the user object to the model
    return "chat";
}
```

}

En este caso desde el controlador enviamos al fichero chat.html donde está la vista de los chats y comunidades de usuarios. Para tratar el tema de la comunidad seleccionada se recoge un campo llamado communityName, si ese campo no existe se usa el campo de la comunidad random.

MÉTODOS POST

Método **sendMessage ()**: Este método está anotado con `@PostMapping("/sendMessage")`, lo que significa que maneja las solicitudes POST a la ruta /api/ChatCommunity/sendMessage.

```
public String sendMessage(@RequestParam("content") String content,@RequestParam("communityName")
String communityName, HttpSession session, RedirectAttributes redirectAttributes)
```

Este método se encarga de recoger el contenido de los mensajes de la comunidad correspondiente que se envían desde la vista al controlador para que este los almacene en la base de datos y devuelva a la vista el chat actualizado con ese nuevo mensaje. Comprueba como todos los métodos que el usuario este registrado correctamente, que tenga los pagos validados para poder realizar la acción. Se crea una nueva entidad del mensaje del chat, la comunidad se recoge de la vista también, el usuario que envía el mensaje se recoge desde la vista y el resto de datos son la hora y la fecha en la que se envía el mensaje. Una vez guardado el mensaje volvemos de nuevo a la vista con pasándole la ruta del controlador con la comunidad seleccionada.

```
@PostMapping("/sendMessage")
public String sendMessage(@RequestParam("content") String content, @RequestParam("communityName") String
communityName, HttpSession session, RedirectAttributes redirectAttributes) {
    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
    }
    ChatCommunity community = chatCommunityService.getCommunityByName(communityName);
    if (community == null) {
        redirectAttributes.addFlashAttribute("errorMessage", "Comunidad no encontrada");
        return "redirect:/api/ChatCommunity/";
    }
    try {
        ChatMessage newMessage = new ChatMessage();
        newMessage.setMessage(content);
        newMessage.setSentBy(user);
        newMessage.setCommunity(community);
        newMessage.setSentDate();
        newMessage.setHoraEnvio();
        chatMessageService.createMessage(newMessage);
        redirectAttributes.addFlashAttribute("successMessage", "Mensaje enviado con éxito");
    } catch (Exception e) {
```

```
        redirectAttributes.addFlashAttribute("errorMessage", "Error al enviar el mensaje: " +
e.getMessage());
    }
    return "redirect:/api/ChatCommunity/" + communityName;
}
```

Por parte de la vista solo enviamos el contenido del mensaje y la comunidad seleccionada en este caso la función de enviar mensajes solo está habilitada para los usuarios con un plan de suscripción básico o superior, para crear esta restricción se usa la dependencia thymeleaf [2] [6] de donde recogemos la sesión del usuario que pasa el controlador en el método get y así tenemos un control del plan de suscripción del usuario.

```
<div th:if="${user.getPlanSuscripcion() != 'Gratis'}">
    <form th:action="@{/api/ChatCommunity/sendMessage}" method="post">
        <div class="input-group">
            <div class="input-group-append">
                <span class="input-group-text attach_btn"><i class="fas fa-paperclip"></i></span>
            </div>
            <textarea name="content" class="form-control type_msg"
placeholder="Type your message..." required></textarea>
            <input type="hidden" name="communityName"
th:value="${comunidadSeleccionada.nombreComunidad}" />
            <div class="input-group-append">
                <button type="submit" class="input-group-text send_btn"><i
class="fas fa-location-arrow"></i></button>
            </div>
        </div>
    </form>
</div>
```

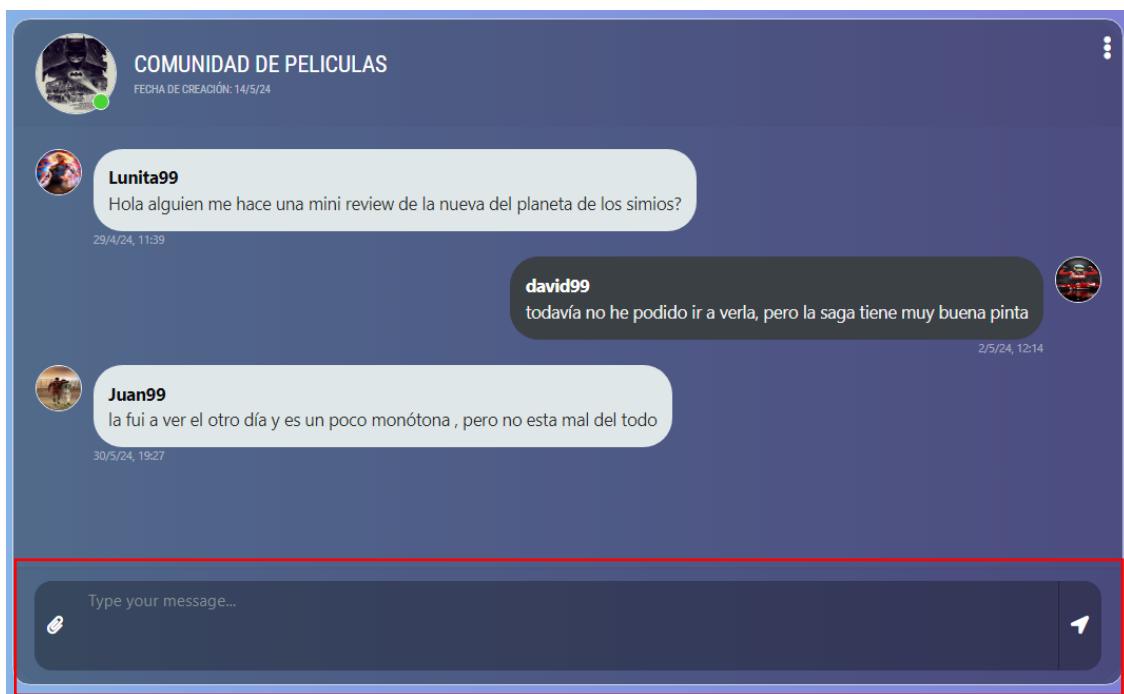


Figura 62: Panel que implementa el método `showChatCommunity` para películas.

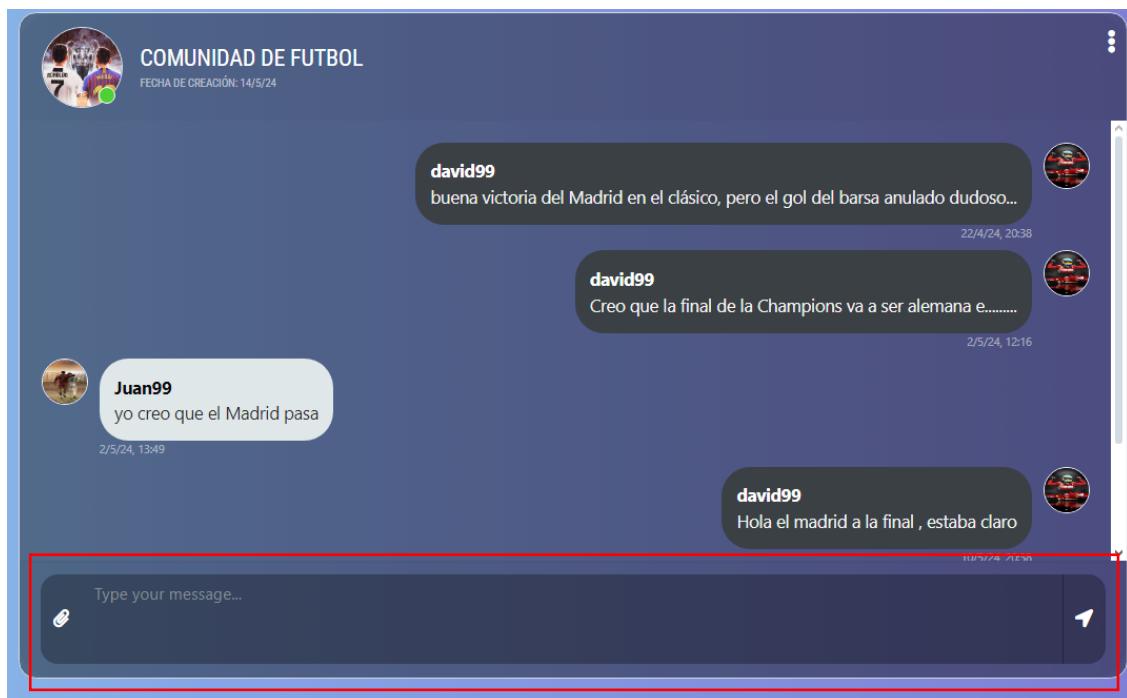


Figura 63: Figura que implementa el método `shonChatCommunity`, para fútbol.

Método **createComunidad ()**: Este método está anotado con `@PostMapping (" / create ")`, lo que significa que maneja las solicitudes POST a la ruta `/api/ChatCommunity/create`.

```
public String createComunidad(
    @RequestParam String nombreComunidad,
    @RequestParam String descripcionComunidad,
    @RequestParam String url_imageComunidad,
    @RequestParam LocalDate fechaCreacion,
```

```
HttpSession session, RedirectAttributes redirectAttributes)
```

Este método está pensado para las funciones que tiene el administrador dentro de la aplicación en este caso la de crear una comunidad nueva, por lo que se comprueba recogiendo la sesión del usuario que ha iniciado sesión que este tiene rol de administrador y así tener control de las acciones que realiza o no el administrador. Luego se crea una nueva entidad de la clase chatCommunity y se rellena con los atributos que se pasan de la vista. De nuevo y como en el resto de métodos gracias a redirectAttributes enviamos los mensajes de error, por si hay algún fallo en el proceso de creación de una nueva comunidad o el mensaje de acción realizada correctamente.

```
@PostMapping("/create")
public String createComunidad(
    @RequestParam String nombreComunidad,
    @RequestParam String descripcionComunidad,
    @RequestParam String url_imageComunidad,
    @RequestParam LocalDate fechaCreacion,
    HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null || (!user.getRoles().stream().anyMatch(role -> role.getName().equals("ADMIN")))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        ChatCommunity community = new ChatCommunity();
        community.setNombreComunidad(nombreComunidad);
        community.setDescripcion(descripcionComunidad);
        community.setFechaCreacion(fechaCreacion);
        community.setCreatedBy(user);
        community.setUrl_image(url_imageComunidad);

        chatCommunityService.createCommunity(community);
        redirectAttributes.addFlashAttribute("successMessage", "Comunidad creada con éxito");
        return "redirect:/api/ChatCommunity/"; // Redirect to the main movie page
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al crear la comunidad");
        return "redirect:/api/ChatCommunity/"; // Redirect back to the add movie page
    }
}
```

Por parte de la vista usamos thymeleaf [2] [6] para que el campo de crear comunidad solo se vea para administradores, aparte dentro del controlador se vuelve a hacer la comprobación para implementar mayor seguridad y robustez a la aplicación. Pasamos los campos de descripción, nombre de la comunidad, fecha de creación y la imagen al controlador para que este cree la nueva entidad de chatCommunity. El diseño de los campos es basado en Bootstrap [8].

```
<h3 class="titulo-cine">PANEL DE ADMINISTRADOR - CREAR COMUNIDAD</h3>
</div>
<div class="card-body">
    <form th:action="@{/api/ChatCommunity/create}" method="post" class="row g-3">
        <div class="col-md-6 mb-3">
```

```

<label for="nombreComunidad" class="form-label">Nombre de la comunidad:</label>
<input type="text" id="nombreComunidad" name="nombreComunidad" required class="form-control">

</div>
<div class="col-md-6 mb-3">
    <label for="descripcionComunidad" class="form-label">Descripción:</label>
    <input type="text" id="descripcionComunidad" name="descripcionComunidad" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="url_imageComunidad" class="form-label">URL de la imagen:</label>
    <input type="text" id="url_imageComunidad" name="url_imageComunidad" required
        class="form-control">
</div>
<div class="col-md-6 mb-3">
    <label for="fechaCreacion" class="form-label">Fecha de creación:</label>
    <input type="date" id="fechaCreacion" name="fechaCreacion" required class="form-control">
</div>
<div class="col-12">
    <button type="submit" class="btn btn-primary">Crear Comunidad</button>
</div>
</form>

```



PANEL DE ADMINISTRADOR - CREAR COMUNIDAD

Nombre de la comunidad:

Descripción:

URL de la imagen:

Fecha de creación:

Crear Comunidad

Figura 64: Panel que implementa el método `createComunidad`.

Método **updateComunidad ()**: Este método está anotado con `@PostMapping ("update/ { nombreComunidad} ")`, lo que significa que maneja las solicitudes POST a la ruta /api/ChatCommunity/update/{nombreComunidad}.

```

public String updateComunidad(@PathVariable String nombreComunidad,
                               @RequestParam String nombreComunidadUpdate,
                               @RequestParam String descripcionComunidadUpdate,
                               @RequestParam String url_imageComunidadUpdate,
                               @RequestParam LocalDate fechaCreacionUpdate,
                               HttpSession session, RedirectAttributes redirectAttributes)

```

Este método está pensado para las funciones que tiene el administrador dentro de la aplicación en este caso la de actualizar el contenido de una comunidad, por lo que se comprueba recogiendo la sesión del usuario que ha iniciado sesión que este tiene rol de administrador y así tener control de las acciones que realiza o no el administrador. Se recoge la entidad pasándole el nombre que le llega desde la vista gracias a los métodos declarados en los repositorios, comprobando que esa comunidad exista si no se envía el mensaje de error con el uso de redirectAttributes. Una vez que se tiene la entidad de la comunidad se actualizan todos los campos por los nuevos que le llegan desde la vista y se usa el método del repositorio de actualizar la comunidad y se todas las acciones se han podido realizar correctamente se envía el mensaje a la vista de comunidad actualizada con éxito y si no el error.

```

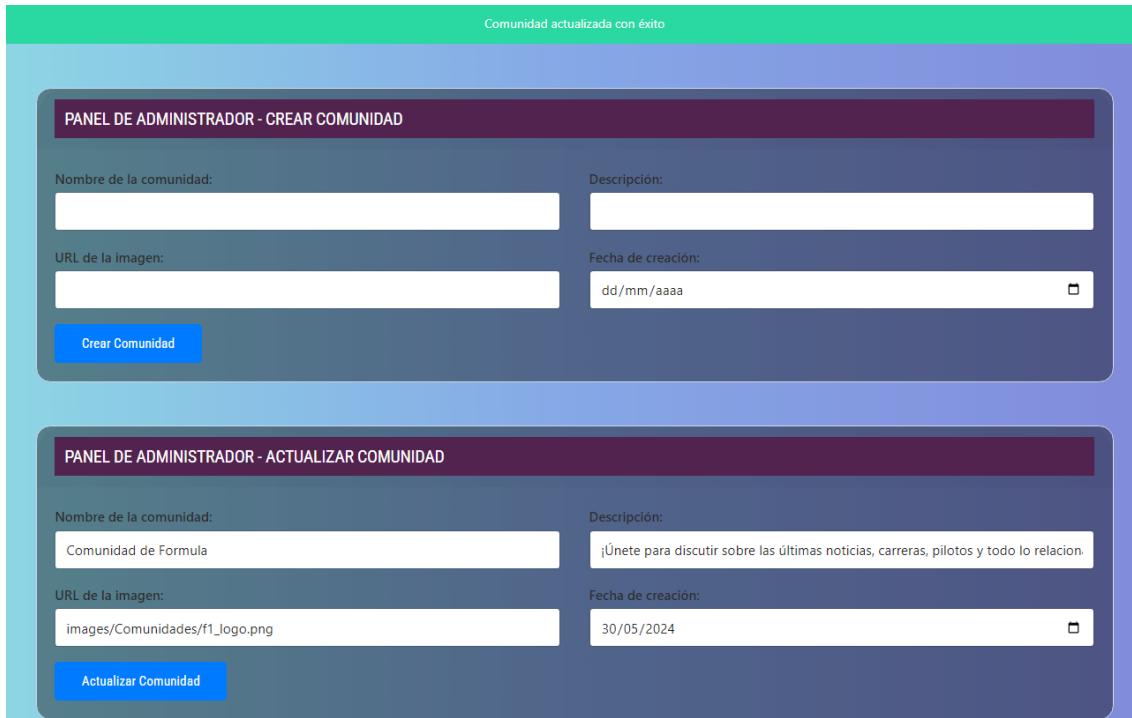
@PostMapping("/update/{nombreComunidad}")
public String updateComunidad(@PathVariable String nombreComunidad,
                               @RequestParam String nombreComunidadUpdate,
                               @RequestParam String descripcionComunidadUpdate,
                               @RequestParam String url_imageComunidadUpdate,
                               @RequestParam LocalDate fechaCreacionUpdate,
                               HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
    if (user == null ||
        (!user.getRoles().stream().map(Role::getName).collect(Collectors.toList()).contains("ADMIN"))) {
        return "login"; // Redirect the user to the login page if not authenticated or not an admin
    }
    try {
        // Obtén la comunidad existente
        ChatCommunity existingCommunity = chatCommunityService.getCommunityByName(nombreComunidad);
        if (existingCommunity == null) {
            redirectAttributes.addFlashAttribute("errorMessage", "Comunidad no encontrada");
            return "redirect:/api/ChatCommunity/";
        }
        // Actualiza los campos necesarios
        existingCommunity.setDescripcion(descripcionComunidadUpdate);
        existingCommunity.setUrl_image(url_imageComunidadUpdate);
        existingCommunity.setFechaCreacion(fechaCreacionUpdate);
        existingCommunity.setNombreComunidad(nombreComunidadUpdate);
        // Guarda la comunidad actualizada
        chatCommunityService.updateComunidad(nombreComunidad, existingCommunity);
        redirectAttributes.addFlashAttribute("successMessage", "Comunidad actualizada con éxito");
        return "redirect:/api/ChatCommunity/" + nombreComunidadUpdate; // Redirect to the updated
    } catch (RuntimeException e) {
        redirectAttributes.addFlashAttribute("errorMessage", "Error al actualizar la comunidad");
        return "redirect:/api/ChatCommunity/" + nombreComunidadUpdate; // Redirect back to the community
    }
}

```

Por parte de la vista usamos thymeleaf [2] [6] para que el campo de actualizar comunidad solo se vea para administradores, aparte dentro del controlador se vuelve a hacer la comprobación para implementar mayor seguridad y robustez a la aplicación. Pasamos los campos de descripción, nombre de la comunidad, fecha de

creación y la imagen al controlador para que este actualice esos nuevos campos si el usuario ha introducido algún cambio. El diseño de los campos es basado en Bootstrap [8].

```
<h3 class="titulo-cine">PANEL DE ADMINISTRADOR - ACTUALIZAR COMUNIDAD</h3>
</div>
<div class="card-body">
    <form th:action="@{/api/ChatCommunity/update/' + ${comunidadSeleccionada.nombreComunidad}''"
          method="post"
          class="row g-3">
        <div class="col-md-6 mb-3">
            <label for="nombreComunidadUpdate" class="form-label">Nombre de la comunidad:</label>
            <input type="text" id="nombreComunidadUpdate" name="nombreComunidadUpdate"
                   th:value="${comunidadSeleccionada.nombreComunidad}"
                   required class="form-control">
        </div>
        <div class="col-md-6 mb-3">
            <label for="descripcionComunidadUpdate" class="form-label">Descripción:</label>
            <input type="text" id="descripcionComunidadUpdate" name="descripcionComunidadUpdate"
                   th:value="${comunidadSeleccionada.descripcion}"
                   required class="form-control">
        </div>
        <div class="col-md-6 mb-3">
            <label for="url_imageComunidadUpdate" class="form-label">URL de la imagen:</label>
            <input type="text" id="url_imageComunidadUpdate" name="url_imageComunidadUpdate"
                   th:value="${comunidadSeleccionada.url_image}"
                   required class="form-control">
        </div>
        <div class="col-md-6 mb-3">
            <label for="fechaCreacionUpdate" class="form-label">Fecha de creación:</label>
            <input type="date" id="fechaCreacionUpdate" name="fechaCreacionUpdate"
                   th:value="${comunidadSeleccionada.fechaCreacion}"
                   required class="form-control">
        </div>
        <div class="col-12">
            <button type="submit" class="btn btn-primary">Actualizar Comunidad</button>
        </div>
    </form>
```



The screenshot shows two panels for managing communities. The top panel is titled 'PANEL DE ADMINISTRADOR - CREAR COMUNIDAD' and contains fields for 'Nombre de la comunidad' (Community name), 'Descripción' (Description), 'URL de la imagen' (Image URL), and 'Fecha de creación' (Creation date). The bottom panel is titled 'PANEL DE ADMINISTRADOR - ACTUALIZAR COMUNIDAD' and contains similar fields, with the description field showing placeholder text: '¡Únete para discutir sobre las últimas noticias, carreras, pilotos y todo lo relacionado con la Fórmula 1.' (Join to discuss about the latest news, races, drivers and everything related to Formula 1.). Both panels have a blue 'Crear Comunidad' (Create Community) button.

Figura 65: Panel que implementa el método updateComunidad.

Método **deleteComunidad** () : Este método está anotado con `@PostMapping` ("`/delete/{nombreComunidad}`"), lo que significa que maneja las solicitudes POST a la ruta `/api/ChatCommunity/delete/{nombreComunidad}`.

```
public String deleteComunidad(@PathVariable String nombreComunidad,
                               @RequestParam String password,
                               @RequestParam String confirmPassword,
                               HttpSession session, RedirectAttributes redirectAttributes)
```

Este método está pensado para las funciones que tiene el administrador dentro de la aplicación en este caso la de borrar una comunidad, por lo que se comprueba recogiendo la sesión del usuario que ha iniciado sesión que este tiene rol de administrador y así tener control de las acciones que realiza o no el administrador. Para la incorporación de mayor seguridad en un método tan delicado como este el administrador tiene que poner su contraseña y confirmarla y ya desde el controlado con los métodos del repositorio y la sesión del usuario administrador comprobamos que coincide con la contraseña. En caso de error con las contraseñas vamos enviando mensajes de error con `redirectAttributes` si las contraseñas no coinciden o si la contraseña no es la del usuario actual, y un mensaje de comunidad creada con éxito si se ha completado el método de borrar correctamente.

```
@PostMapping("/delete/{nombreComunidad}")
public String deleteComunidad(@PathVariable String nombreComunidad,
                               @RequestParam String password,
                               @RequestParam String confirmPassword,
                               HttpSession session, RedirectAttributes redirectAttributes) {
    User user = (User) session.getAttribute("user");
```

```

        if (user == null ||
(!user.getRoles().stream().map(Role::getName).collect(Collectors.toList()).contains("ADMIN"))) {
            return "login"; // Redirect the user to the login page if not authenticated or not an admin
        }
        if (!password.equals(user.getContraseña())) {
            redirectAttributes.addFlashAttribute("errorMessage", "La contraseña no coincide con la del usuario
actual");
            return "redirect:/api/ChatCommunity/"; // Redirect back to the movie page
        }

        if (!password.equals(confirmPassword)) {
            redirectAttributes.addFlashAttribute("errorMessage", "Las contraseñas no coinciden");
            return "redirect:/api/ChatCommunity/"; // Redirect back to the movie page
        }
        try {
            // Implementar la lógica para borrar la comunidad por su nombre
            chatCommunityService.deleteCommunity(nombreComunidad);
            redirectAttributes.addFlashAttribute("successMessage", "Comunidad borrada con éxito");
            return "redirect:/api/ChatCommunity/"; // Redirect to the main community page
        } catch (RuntimeException e) {
            redirectAttributes.addFlashAttribute("errorMessage", "Error al borrar la comunidad");
            return "redirect:/api/ChatCommunity/" + nombreComunidad; // Redirect back to the community page
        }
    }
}

```

Por parte de la vista usamos thymeleaf [2] [6] para que el campo de actualizar comunidad solo se vea para administradores, aparte dentro del controlador se vuelve a hacer la comprobación para implementar mayor seguridad y robustez a la aplicación. Pasamos al controlador los dos campos de contraseña. El diseño de los campos es basado en Bootstrap [8].

```

<h3 class="titulo-cine">PANEL DE ADMINISTRADOR - BORRAR COMUNIDAD</h3>
</div>
<div class="card-body">
    <form th:action="@{'/api/ChatCommunity/delete/' + ${comunidadSeleccionada.nombreComunidad}}"
          method="post"
          class="row g-3">
        <div class="col-md-6 mb-3">
            <label for="password" class="form-label">Contraseña:</label>
            <input type="password" id="password" name="password" required class="form-control">
        </div>
        <div class="col-md-6 mb-3">
            <label for="confirmPassword" class="form-label">Confirmar Contraseña:</label>
            <input type="password" id="confirmPassword" name="confirmPassword" required
                   class="form-control">
        </div>
        <div class="col-12">
            <button type="submit" class="btn btn-danger">Borrar Comunidad</button>
        </div>
    </form>
</div>

```



</form>

Figura 66: Panel que implementa el método deleteComunidad.

9. PRUEBAS

9.1 ESTRATEGIA DE PRUEBAS

Antes del desarrollo de cada entidad con su proceso de crear la entidad, mapearla en la base de datos, crear el repositorio, el servicio y el controlador, he hecho una serie de pruebas con postman [7] para comprobar el funcionamiento de las operaciones CRUD con la entidad y el correcto funcionamiento de la base de datos que es lo que recoge todos los datos y actúa como modelo para que el controlador le haga peticiones de datos, ya sea para guardar nuevos datos, actualizar datos, borrarlos, etc. La estrategia para las pruebas del correcto funcionamiento de cada entidad va a ser dentro del controlador usamos **API REST**.

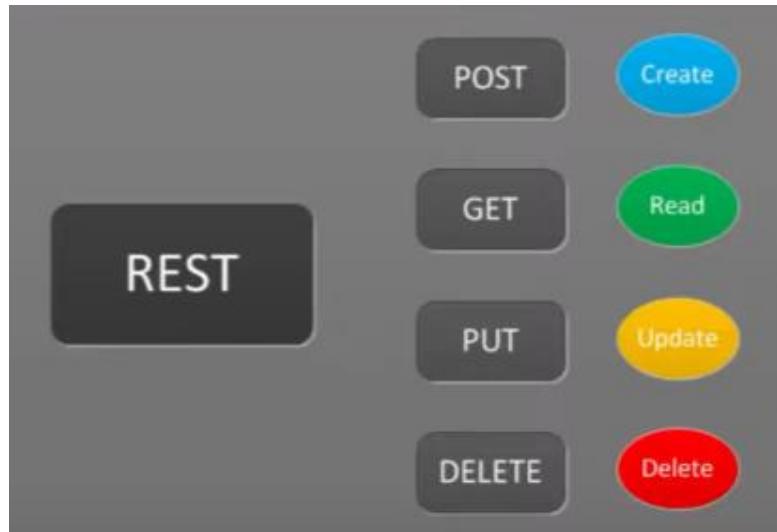


Figura 67: Imagen de funcionamiento de API REST.

Para eso hay de diseñar dentro del controlador los métodos de cada acción CRUD, como hay más de 12 entidades y es todo el rato lo mismo y ya van muchas páginas de memoria voy a enseñar cómo se han hecho las pruebas con la entidad del usuario, la entidad de películas y por último la entidad de comentarios.

Esta es la lógica que se va a seguir vista de forma gráfica.



Figura 68: Imagen 2, funcionamiento de API REST.

El cliente mediante la interfaz de la plataforma o en este caso mediante post solicita la creación, actualización, búsqueda o eliminación en este caso de usuarios, para esta metodología lo mejor es el uso de una Api Rest que con el controlador enviamos las peticiones que nos llegan a la base de datos, la base de datos realiza la acción y si todo se ha realizado correctamente el cliente obtiene lo que ha solicitado inicialmente, ya sea por ejemplo ver todas las películas o una determinada, actualizar una o borrarla. Para esto son cruciales las pruebas antes de ponerse a desarrollar dando palos de ciego y generar problemas tanto a la larga como a la corta, primero hacemos las pruebas y hacemos todos los métodos CRUD y una vez que funcionan ya lo implementamos en la plataforma. Pero esta sería la estrategia que he seguido con respecto a las pruebas.

9.2 CASOS DE PRUEBA

Dentro del controlador definimos las operaciones para las pruebas usando la arquitectura de micro servicios rest, en este caso para las entidades de usuarios, películas y comentarios.

9.2.1 Casos de prueba Usuarios

Dentro del controlador `userController` definimos los métodos para llevar a cabo las operaciones CRUD en este caso con la entidad `user`, gracias a estos métodos que se basan en los repositorios y servicios creados en el modelo podemos hacer las pruebas con postman [7] para verificar que el flujo de comunicación vista-controlador-modelo y viceversa funciona correctamente.

Método para crear un usuario. (C create)

```
@PostMapping
public ResponseEntity<User> createUser(@RequestBody User user) {
    User createdUser = userService.createUser(user);
    return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
```

```
}
```

Para guardar un usuario en la base de datos usamos el método post de http [6] [7] que nos permite enviar datos al servidor

Método para obtener un usuario por su id (R read)

```
@GetMapping("/{id}")
public ResponseEntity<?> read(@PathVariable(value = "id") Long id) {
    Optional<User> oUser = userService.findById(id);
    if (!oUser.isPresent()) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(oUser);
}
```

Para obtener todos los usuarios de la base de datos usamos el método get de http [6] [7] que nos permite obtener datos del servidor

Método para obtener un usuario por su email (R read)

```
@GetMapping("/{email}")
public ResponseEntity<User> getUserByEmail(@PathVariable String email) {
    try {
        Optional<User> user = userService.getUserByEmail(email);
        if (user.isPresent()) {
            return new ResponseEntity<>(user.get(), HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

Método para actualizar un usuario pasando como entrada su email. (U update)

```
@PutMapping("/{email}")
public ResponseEntity<User> updateUser(@PathVariable String email, @RequestBody User
userDetails) {
    try {
        User updatedUser = userService.updateUser(email, userDetails);
        return new ResponseEntity<>(updatedUser, HttpStatus.OK);
    } catch (RuntimeException e) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Para actualizar un usuario de la base de datos usamos el método put de http [6] [7] que nos permite enviar datos al servidor

Método para eliminar un usuario usando como entrada su mail (D delete)

```
@DeleteMapping("/{email}")
public ResponseEntity<Void> deleteUser(@PathVariable String email) {
    try {
        userService.deleteUser(email);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    } catch (RuntimeException e) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Para eliminar un usuario de la base de datos usamos el método delete de http [6] [7] que nos permite enviar datos al servidor

Estos métodos se van a comunicar con el servicio que se encargara de hacer las operaciones en la base de datos y el servicio se va a comunicar con el repositorio que se encargara de hacer las consultas a la base de datos. El repositorio se va a comunicar con la base de datos

Y así se va a hacer el flujo de información:

Controlador -> Servicio -> Repositorio -> Base de datos

Y viceversa

Base de datos -> Repositorio -> Servicio -> Controlador

9.2.2 Casos de Pruebas Películas

Dentro del controlador peliculaController definimos los métodos para llevar a cabo las operaciones CRUD en este caso con la entidad película, gracias a estos métodos que se basan en los repositorios y servicios creados en el modelo podemos hacer las pruebas con postman [7] para verificar que el flujo de comunicación vista-controlador-modelo y viceversa funciona correctamente.

Método para crear una película. (C create)

```
@PostMapping
public ResponseEntity<?> create(@RequestBody Pelicula pelicula) {
    return ResponseEntity.status(HttpStatus.CREATED).body(peliculaService.save(pelicula));
}
```

Para guardar una película en la base de datos usamos el método post de http [6] [7] que nos permite enviar datos al servidor

Método para obtener una película por su nombre (R read)

```
@GetMapping("/{nombrePelicula}")
public ResponseEntity<?> read(@PathVariable(value = "nombrePelicula") String nombrePelicula)
{
    Pelicula pelicula = peliculaService.findByNombrePelicula(nombrePelicula);
```

```

    if (pelicula == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(pelicula);
}

```

Para obtener películas de la base de datos usamos el método get de http [6] [7] que nos permite obtener datos del servidor.

Método para actualizar una película pasando como entrada su nombre. (U update)

```

@PutMapping("/{nombrePelicula}")
public ResponseEntity<?> update(@RequestBody Pelicula peliculaDetails, @PathVariable(value =
"nombrePelicula") String nombrePelicula) {
    Pelicula pelicula = peliculaService.findByNombrePelicula(nombrePelicula);
    if (pelicula == null) {
        return ResponseEntity.notFound().build();
    }

    pelicula.setNombrePelicula(peliculaDetails.getNombrePelicula());
    pelicula.setSinopsis(peliculaDetails.getSinopsis());
    pelicula.setPaginaOficial(peliculaDetails.getPaginaOficial());
    pelicula.setTituloOriginal(peliculaDetails.getTituloOriginal());
    pelicula.setGenero(peliculaDetails.getGenero());
    pelicula.setNacionalidad(peliculaDetails.getNacionalidad());
    pelicula.setDuracion(peliculaDetails.getDuracion());
    pelicula.setAnho(peliculaDetails.getAnho());
    pelicula.setDistribuidora(peliculaDetails.getDistribuidora());
    pelicula.setDirector(peliculaDetails.getDirector());
    pelicula.setClasificacionEdad(peliculaDetails.getClasificacionEdad());
    pelicula.setOtrosDatos(peliculaDetails.getOtrosDatos());
    pelicula.setActores(peliculaDetails.getActores());
    pelicula.setUrl_image(peliculaDetails.getUrl_image());
    pelicula.setUrl_video(peliculaDetails.getUrl_video());

    return ResponseEntity.status(HttpStatus.CREATED).body(peliculaService.save(pelicula));
}

```

Para actualizar una película de la base de datos usamos el método put de http [6] [7] que nos permite enviar datos al servidor.

Método para eliminar una película usando como entrada su nombre (D delete)

```

@DeleteMapping("/{nombrePelicula}")
public ResponseEntity<?> delete(@PathVariable(value = "nombrePelicula") String
nombrePelicula) {
    if (peliculaService.findByNombrePelicula(nombrePelicula) == null) {
        return ResponseEntity.notFound().build();
    }
}

```

```

    }
    peliculaService.deleteByNombrePelícula(nombrePelícula);
    return ResponseEntity.ok().build();
}

```

Para eliminar una película de la base de datos usamos el método delete de http [6] [7] que nos permite enviar datos al servidor.

Estos métodos se van a comunicar con el servicio que se encargara de hacer las operaciones en la base de datos y el servicio se va a comunicar con el repositorio que se encargara de hacer las consultas a la base de datos. El repositorio se va a comunicar con la base de datos

Y así se va a hacer el flujo de información:

Controlador -> Servicio -> Repositorio -> Base de datos

Y viceversa

Base de datos -> Repositorio -> Servicio -> Controlador

9.2.3 Casos de Prueba Comentarios

Dentro del controlador `comentarioPelículaController` definimos los métodos para llevar a cabo las operaciones CRUD en este caso con la entidad `comentarioPelícula`, gracias a estos métodos que se basan en los repositorios y servicios creados en el modelo podemos hacer las pruebas con postman [7] para verificar que el flujo de comunicación vista-controlador-modelo y viceversa funciona correctamente.

Método para crear un comentario. (C create)

```

@PostMapping
public ResponseEntity<?> create(@RequestBody Comentario comentario) {
    Optional<User> user = userService.getUserByEmail(comentario.getUsuario().getEmail());
    Película película =
    películaService.findByNamePelícula(comentario.getPelícula().getNombrePelícula());

    if (user == null || película == null) {
        return ResponseEntity.notFound().build();
    }
    comentario.setUsuario(user.get());
    comentario.setPelícula(película);
    return
    ResponseEntity.status(HttpStatus.CREATED).body(comentarioService.save(comentario));
}

```

Para guardar un comentario en la base de datos usamos el método post de http [6] [7] que nos permite enviar datos al servidor.

Método para obtener un comentario por su id (R read)

```

@GetMapping("/{id}")
public ResponseEntity<?> read(@PathVariable(value = "id") Long id) {

```

```

Optional<Comentario> oComentario = comentarioService.findById(id);
if (!oComentario.isPresent()) {
    return ResponseEntity.notFound().build();
}
return ResponseEntity.ok(oComentario);
}

```

Para obtener todos los comentarios de la base de datos usamos el método get de http [6] [7] que nos permite obtener datos del servidor.

Método para actualizar un comentario pasando como entrada su id. (U update)

```

@PutMapping("/{id}")
public ResponseEntity<?> update(@RequestBody Comentario comentarioDetails,
@PathVariable(value = "id") Long id) {
    Optional<Comentario> comentario = comentarioService.findById(id);
    if (!comentario.isPresent()) {
        return ResponseEntity.notFound().build();
    }
    comentario.get().setTexto(comentarioDetails.getTexto());
    comentario.get().setValoracion(comentarioDetails.getValoracion());
    comentario.get().setFechaComentario(comentarioDetails.getFechaComentario());
    comentario.get().setUsuario(comentarioDetails.getUsuario());
    comentario.get().setPelicula(comentarioDetails.getPelicula());
    return
    ResponseEntity.status(HttpStatus.CREATED).body(comentarioService.save(comentario.get()));
}

```

Para actualizar un comentario de la base de datos usamos el método put de http [6] [7] que nos permite enviar datos al servidor

Método para eliminar un comentario usando como entrada su id (D delete)

```

@DeleteMapping("/{id}")
public ResponseEntity<?> delete(@PathVariable(value = "id") Long id) {
    if (!comentarioService.findById(id).isPresent()) {
        return ResponseEntity.notFound().build();
    }
    comentarioService.deleteById(id);
    return ResponseEntity.ok().build();
}

```

Para eliminar un comentario de la base de datos usamos el método delete de http [6] [7] que nos permite enviar datos al servidor

Estos métodos se van a comunicar con el servicio que se encargara de hacer las operaciones en la base de datos y el servicio se va a comunicar con el repositorio que se encargara de hacer las consultas a la base de datos. El repositorio se va a comunicar con la base de datos

Y así se va a hacer el flujo de información:

Controlador -> Servicio -> Repositorio -> Base de datos

Y viceversa

Base de datos -> Repositorio -> Servicio -> Controlador

9.3 Resultados de las pruebas

Postman [7] proporciona una interfaz gráfica de usuario para enviar solicitudes HTTP [6] [7] a los endpoints de la API y ver las respuestas.

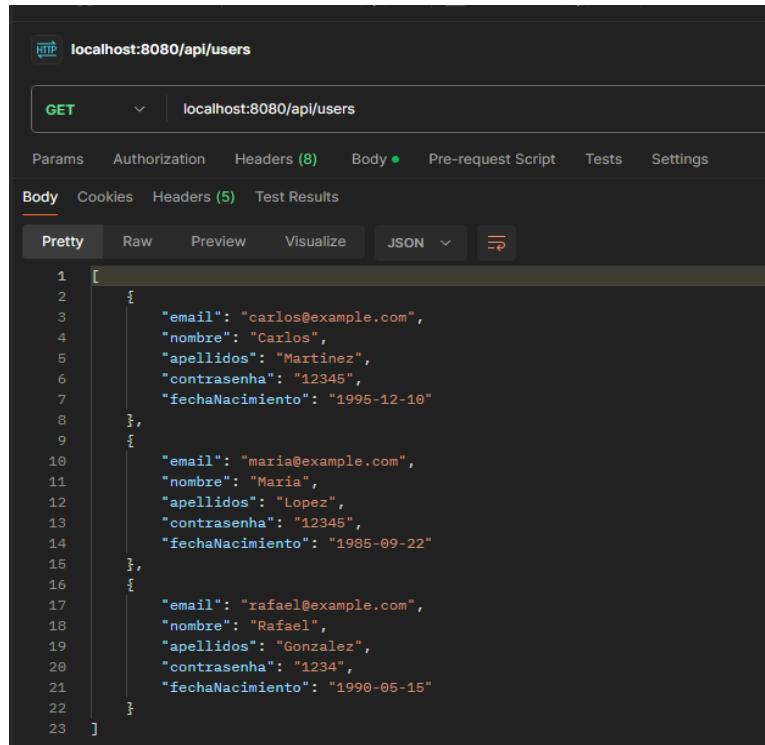
9.3.1 Entidad Usuario

GET - Obtener todos los usuarios:

Método: GET

URL: <http://localhost:8080/api/users>

En Postman [7], se selecciona el método GET y enviar la solicitud. Se recibe una respuesta con una lista de todos los usuarios en la base de datos.



```
1 [  
2 {  
3     "email": "carlos@example.com",  
4     "nombre": "Carlos",  
5     "apellidos": "Martinez",  
6     "contrasenha": "12345",  
7     "fechaNacimiento": "1995-12-10"  
8 },  
9 {  
10    "email": "maria@example.com",  
11    "nombre": "Maria",  
12    "apellidos": "Lopez",  
13    "contrasenha": "12345",  
14    "fechaNacimiento": "1985-09-22"  
15 },  
16 {  
17    "email": "rafael@example.com",  
18    "nombre": "Rafael",  
19    "apellidos": "Gonzalez",  
20    "contrasenha": "1234",  
21    "fechaNacimiento": "1990-05-15"  
22 }  
23 ]
```

Figura 69: Obtener todos los usuarios con Postman.

GET - Obtener un usuario por email:

Método: GET

URL: <http://localhost:8080/api/users/{email}>

Reemplaza {email} con el correo electrónico del usuario que deseas buscar, en Postman [7], selecciona el método GET y envía la solicitud. Se recibe una respuesta con los detalles del usuario correspondiente al correo electrónico proporcionado.

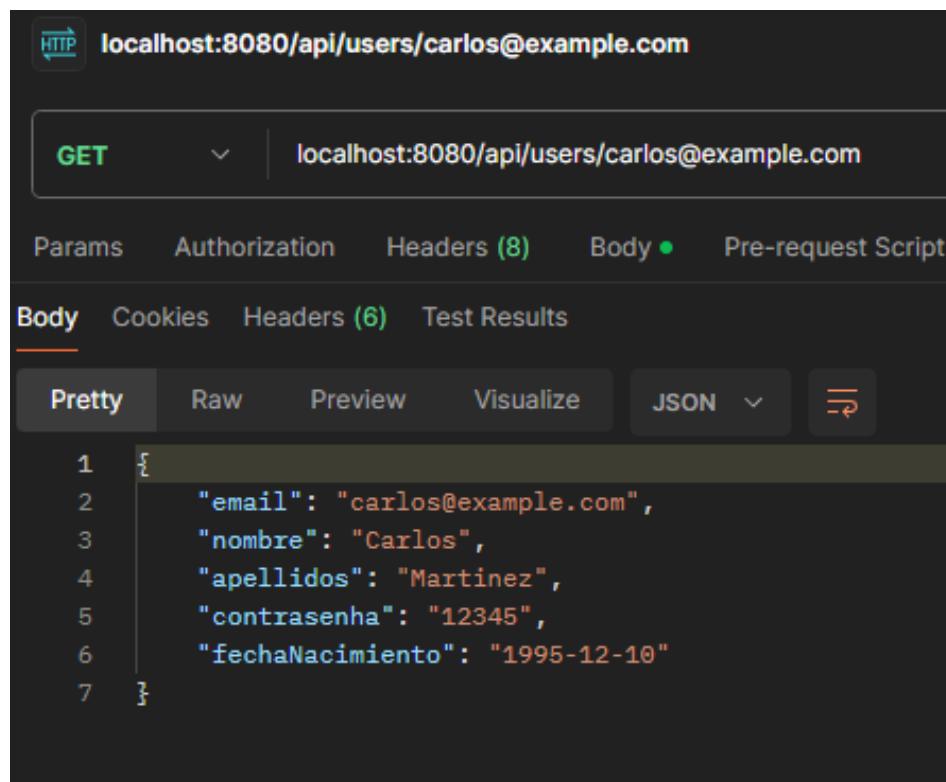


Figura 70: Obtener un usuario por mail Postman.

POST - Crear un nuevo usuario:

Método: POST

URL: <http://localhost:8080/api/users>

En la pestaña Body, selecciona raw y el tipo de datos JSON. Ingresa los datos del nuevo usuario en formato JSON en el cuerpo de la solicitud.

En Postman [7], selecciona el método POST, ingresa los datos del nuevo usuario en formato JSON en el cuerpo de la solicitud y envía la solicitud. Se recibe una respuesta con los detalles del usuario recién creado.

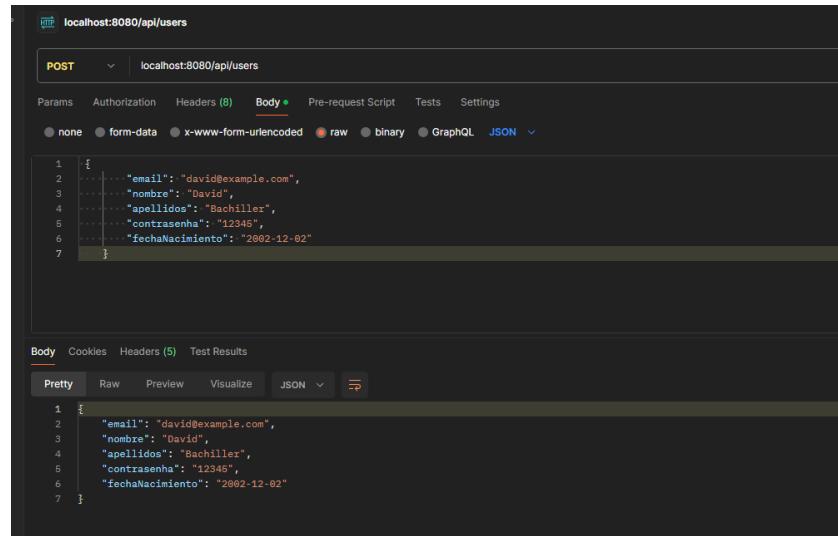


Figura 71: Crear un nuevo Usuario Postman.

PUT - Actualizar un usuario existente:

Método: PUT

URL: <http://localhost:8080/api/users/{email}>

Reemplaza {email} con el correo electrónico del usuario que deseas actualizar.

En la pestaña Body, selecciona raw y el tipo de datos JSON.

Se ingresa los campos que se desea actualizar en formato JSON en el cuerpo de la solicitud.

En Postman [7], seleccionamos el método PUT, ingresamos los datos actualizados en formato JSON en el cuerpo de la solicitud y se envía la solicitud. Se recibe una respuesta con los detalles del usuario actualizado.

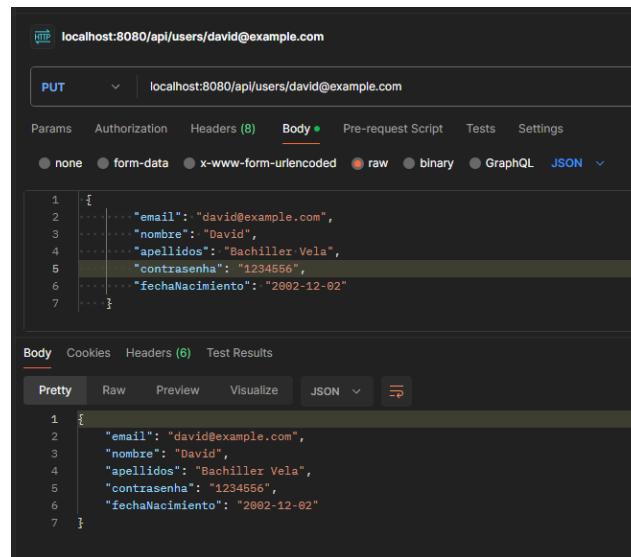


Figura 72: Actualizar un usuario existente Postman.

DELETE - Eliminar un usuario:

Método: DELETE

URL: <http://localhost:8080/api/users/{email}>

Reemplaza {email} con el correo electrónico del usuario que deseas eliminar.

En Postman [7], se selecciona el método DELETE y se envía la solicitud. Se recibe una respuesta indicando que el usuario ha sido eliminado correctamente.

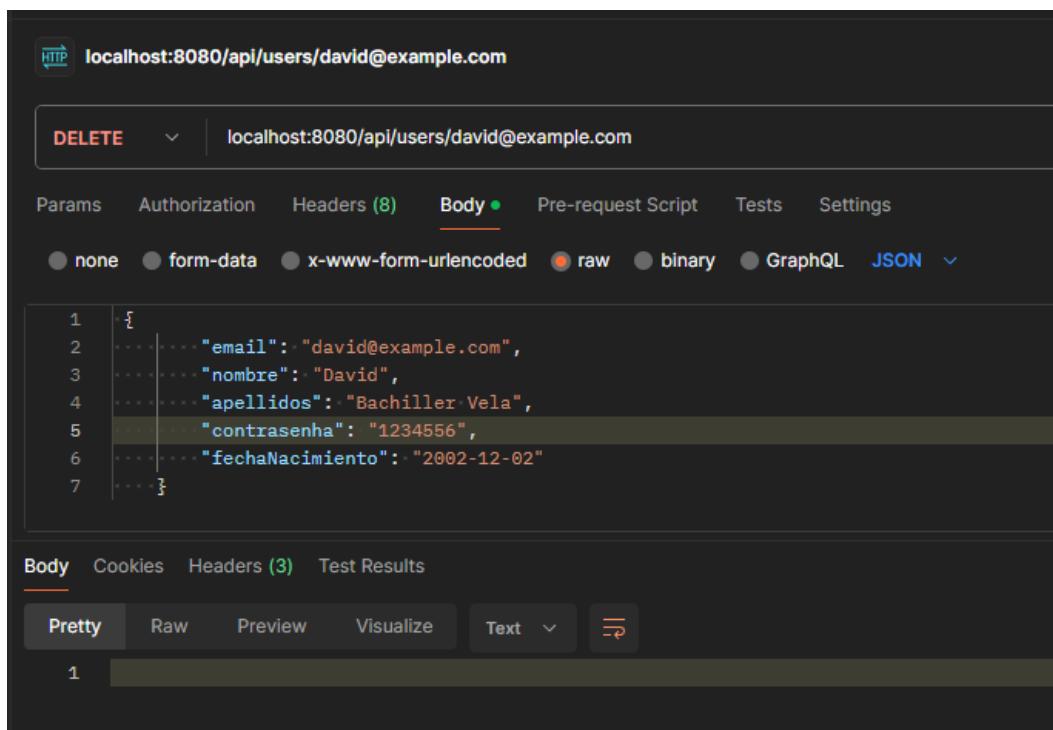


Figura 73: Eliminar un usuario Postman.

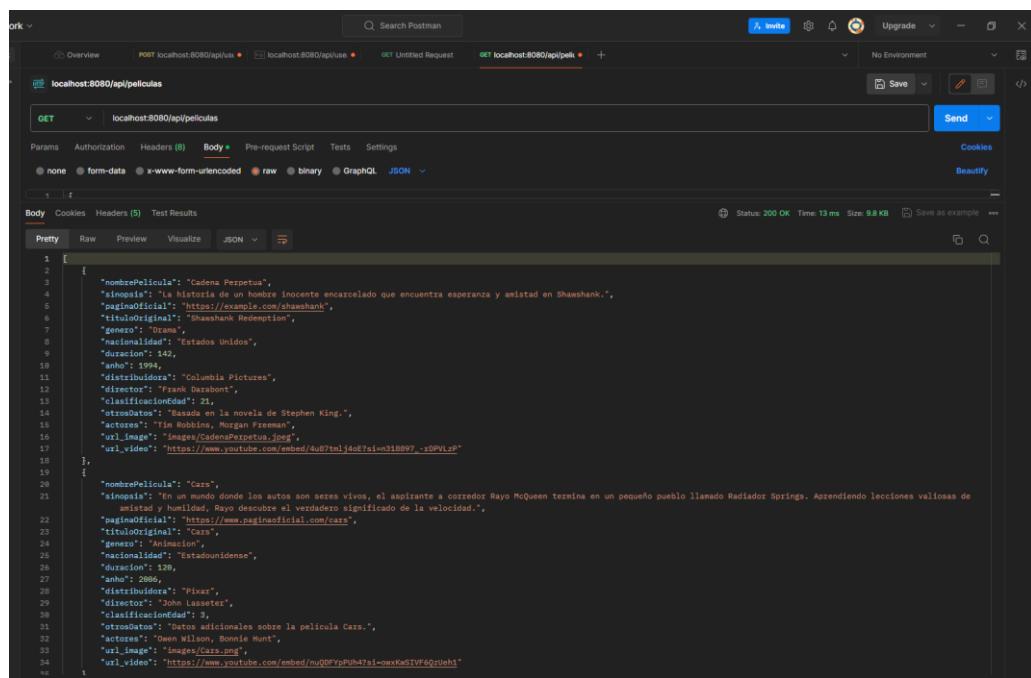
9.3.2 Entidad Película

GET - Obtener todas las películas:

Método: GET

URL: <http://localhost:8080/api/peliculas>

En Postman [7], selecciona el método GET y envía la solicitud. Se recibe una respuesta con una lista de todas las películas en la base de datos.



```

1 [
2   {
3     "nombrePelicula": "Cadena Perpetua",
4     "sinopsis": "La historia de un hombre inocente encarcelado que encuentra esperanza y amistad en Shawshank.",
5     "paginaOficial": "https://example.com/shawshank",
6     "tituloOriginal": "Shawshank Redemption",
7     "genero": "Drama",
8     "nacionalidad": "Estados Unidos",
9     "duracion": 142,
10    "año": 1994,
11    "distribuidora": "Columbia Pictures",
12    "director": "Frank Darabont",
13    "clasificacionEdad": 21,
14    "otrosDatos": "Basada en la novela de Stephen King.",
15    "actores": "Tim Robbins, Morgan Freeman",
16    "url_image": "images/CadenaPerpetua.jpg",
17    "url_video": "https://www.youtube.com/embed/4u87tmj4cE?si=n318897_-rDPVizP"
18  },
19  {
20    "nombrePelicula": "Cars",
21    "sinopsis": "Un día donde los autos con seres vivos, el aspirante a corredor Rayo McQueen termina en un pequeño pueblo llamado Radiador Springs. Aprendiendo lecciones valiosas de amistad y humildad, Rayo descubre el verdadero significado de la velocidad",
22    "paginaOficial": "https://www.paginaoficial.com/cars",
23    "tituloOriginal": "Cars",
24    "genero": "Animación",
25    "nacionalidad": "Estadounidense",
26    "duracion": 120,
27    "año": 2006,
28    "distribuidora": "Pixar",
29    "director": "John Lasseter",
30    "clasificacionEdad": 3,
31    "otrosDatos": "Datos adicionales sobre la película Cars.",
32    "actores": " Owen Wilson, Bonnie Hunt",
33    "url_image": "images/Cars.jpg",
34    "url_video": "https://www.youtube.com/embed/nuQDFYpPUh47si=gexXaGIVF6qzlehi1"
35  }
]

```

Figura 74: Obtener todas las películas con Postman.

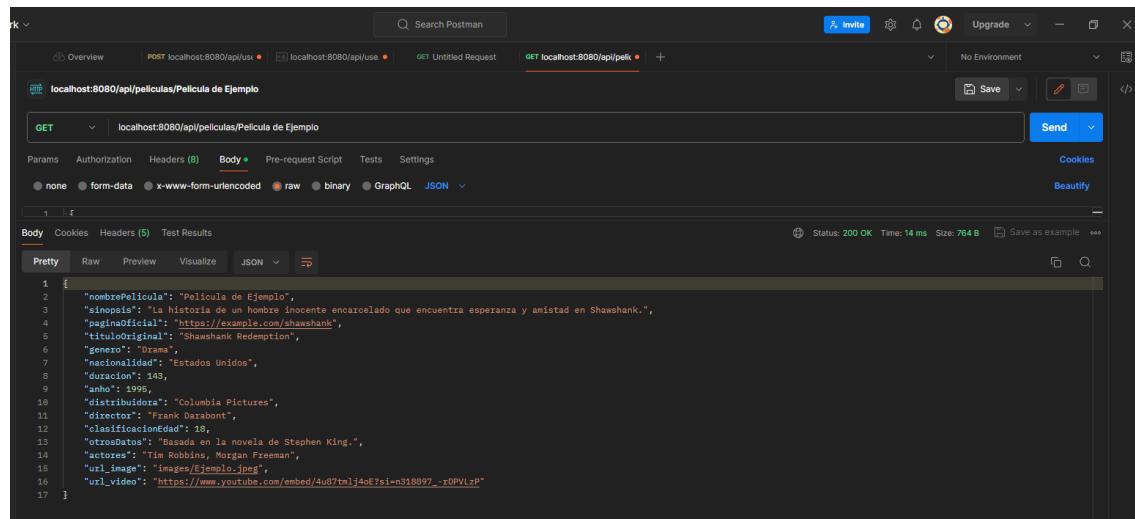
GET - Obtener una película por nombre:

Método: GET

URL: <http://localhost:8080/api/peliculas/{nombrePelicula}>

(Reemplazar {nombrePelicula} con el nombre de la película que deseas buscar)

En Postman [7], se selecciona el método GET y se envía la solicitud. Se recibe una respuesta con los detalles de la película correspondiente al nombre proporcionado.



```

1 {
2   "nombrePelicula": "Película de Ejemplo",
3   "sinopsis": "La historia de un hombre inocente encarcelado que encuentra esperanza y amistad en Shawshank.",
4   "paginaOficial": "https://example.com/shawshank",
5   "tituloOriginal": "Shawshank Redemption",
6   "genero": "Drama",
7   "nacionalidad": "Estados Unidos",
8   "duracion": 143,
9   "año": 1995,
10  "distribuidora": "Columbia Pictures",
11  "director": "Frank Darabont",
12  "clasificacionEdad": 18,
13  "otrosDatos": "Basada en la novela de Stephen King.",
14  "actores": " Tim Robbins, Morgan Freeman",
15  "url_image": "images/Ejemplo.jpg",
16  "url_video": "https://www.youtube.com/embed/4u87tmj4cE?si=n318897_-rDPVizP"
17 }

```

Figura 75: Obtener una película por nombre Postman.

POST - Crear una nueva película:

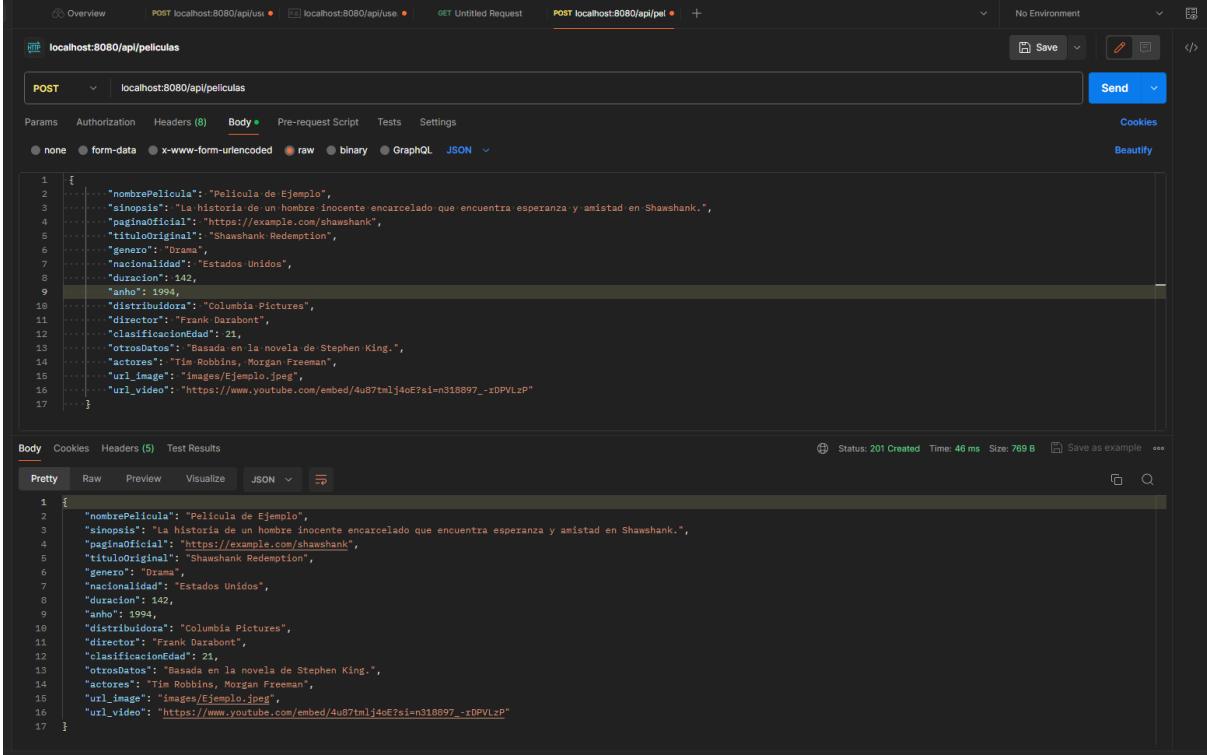
Método: POST

URL: <http://localhost:8080/api/peliculas>

En la pestaña Body, selecciona raw y el tipo de datos JSON.

Ingresar los datos de la nueva película en formato JSON en el cuerpo de la solicitud.

En Postman [7], se selecciona el método POST, se ingresan los datos de la nueva película en formato JSON en el cuerpo de la solicitud y se envía la solicitud. Se recibe una respuesta con los detalles de la película recién creada.



```

1 {
2     "nombrePelicula": "Película de Ejemplo",
3     "sinopsis": "La historia de un hombre inocente encarcelado que encuentra esperanza y amistad en Shawshank.",
4     "paginaOficial": "https://example.com/shawshank",
5     "tituloOriginal": "Shawshank Redemption",
6     "genero": "Drama",
7     "nacionalidad": "Estados Unidos",
8     "duracion": 142,
9     "año": 1994,
10    "distribuidora": "Columbia Pictures",
11    "director": "Frank Darabont",
12    "clasificacionEdad": 21,
13    "otrosDatos": "Basada en la novela de Stephen King.",
14    "actores": "Tim Robbins, Morgan Freeman",
15    "url_image": "images/Ejemplo.jpeg",
16    "url_video": "https://www.youtube.com/embed/4u87tm1j4oE?si=n318897_-rOPVLzP"
17 }

```

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```

1 {
2     "nombrePelicula": "Película de Ejemplo",
3     "sinopsis": "La historia de un hombre inocente encarcelado que encuentra esperanza y amistad en Shawshank.",
4     "paginaOficial": "https://example.com/shawshank",
5     "tituloOriginal": "Shawshank Redemption",
6     "genero": "Drama",
7     "nacionalidad": "Estados Unidos",
8     "duracion": 142,
9     "año": 1994,
10    "distribuidora": "Columbia Pictures",
11    "director": "Frank Darabont",
12    "clasificacionEdad": 21,
13    "otrosDatos": "Basada en la novela de Stephen King.",
14    "actores": "Tim Robbins, Morgan Freeman",
15    "url_image": "images/Ejemplo.jpeg",
16    "url_video": "https://www.youtube.com/embed/4u87tm1j4oE?si=n318897_-rOPVLzP"
17 }

```

Status: 201 Created Time: 46 ms Size: 769 B Save as example

Figura 76: Crear una nueva película Postman.

PUT - Actualizar una película existente:

Método: PUT

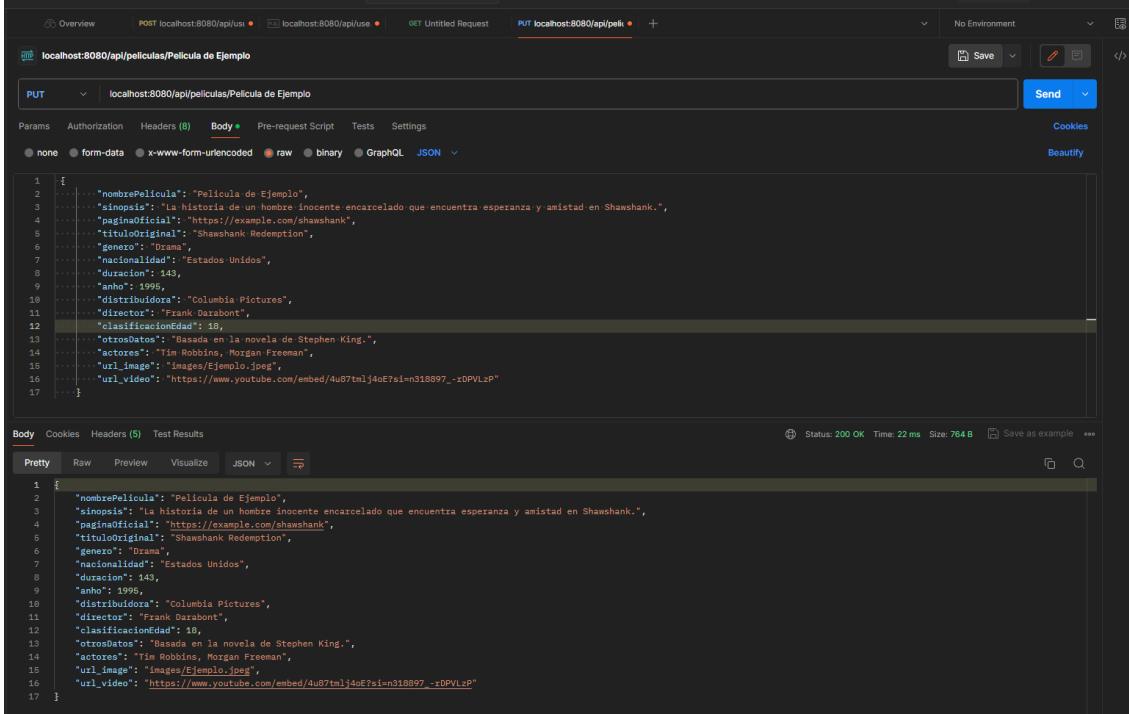
URL: <http://localhost:8080/api/peliculas/{nombrePelicula}>

(Reemplaza {nombrePelicula} con el nombre de la película que deseas actualizar)

En la pestaña Body, seleccionamos raw y el tipo de datos JSON.

Se ingresa los campos que se desea actualizar en formato JSON en el cuerpo de la solicitud.

En Postman [7], seleccionamos el método PUT, ingresamos los datos actualizados en formato JSON en el cuerpo de la solicitud y se envía la solicitud. Se recibe una respuesta con los detalles de la película actualizada.



The screenshot shows the Postman interface with a PUT request to `localhost:8080/api/peliculas/Película de Ejemplo`. The request body contains the following JSON:

```

1  {
2     "nombrePelicula": "Película de Ejemplo",
3     "sinopsis": "La historia de un hombre inocente encarcelado que encuentra esperanza y amistad en Shawshank.",
4     "paginaOficial": "https://example.com/shawshank",
5     "tituloOriginal": "Shawshank Redemption",
6     "genero": "Drama",
7     "nacionalidad": "Estados Unidos",
8     "duracion": 143,
9     "año": 1994,
10    "distribuidora": "Columbia Pictures",
11    "director": "Frank Darabont",
12    "clasificacionEdad": 18,
13    "otroDatos": "Basada en la novela de Stephen King.",
14    "actores": "Tim Robbins, Morgan Freeman",
15    "url_image": "images/Ejemplo.jpg",
16    "url_video": "https://www.youtube.com/embed/4u87tm1j4oE?si=n318897_-xDPLzP"
17  }

```

The response status is 200 OK, with a response body identical to the request body.

Figura 77: Actualizar una película existente Postman.

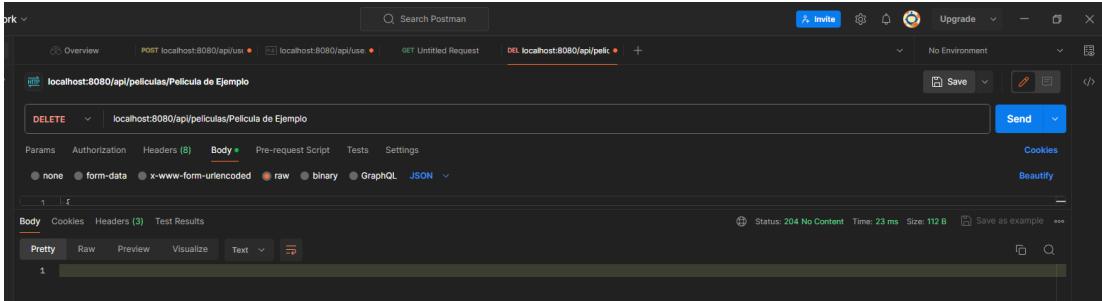
DELETE - Eliminar una película:

Método: DELETE

URL: <http://localhost:8080/api/peliculas/{nombrePelicula}>

(Reemplaza {nombrePelicula} con el nombre de la película que deseas eliminar)

En Postman [7], se selecciona el método DELETE y se envía la solicitud. Se recibe una respuesta indicando que la película ha sido eliminada correctamente.



The screenshot shows the Postman interface with a DELETE request to `localhost:8080/api/peliculas/Película de Ejemplo`. The response status is 204 No Content.

Figura 78: Eliminar una película Postman.

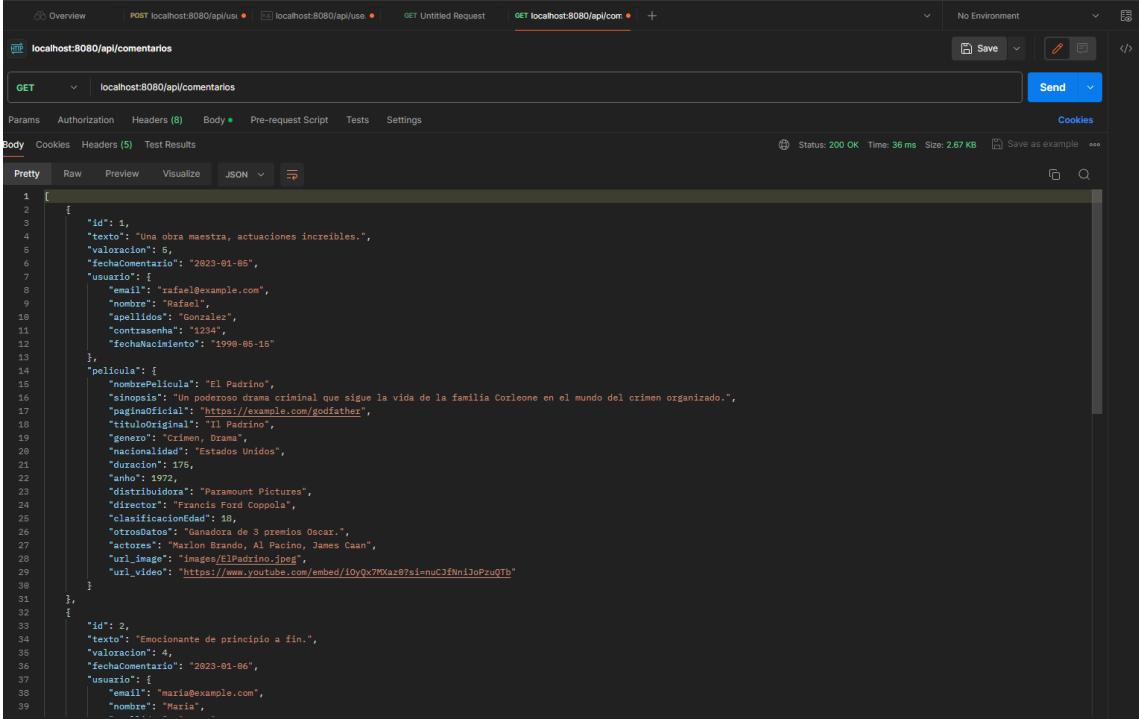
9.3.2 Entidad Comentario

GET - Obtener todos los comentarios:

Método: GET

URL: <http://localhost:8080/api/comentarios>

En Postman [7], se selecciona el método GET y se envía la solicitud. Se recibe una respuesta con una lista de todos los comentarios en la base de datos. Este método se utiliza para obtener una lista de todos los comentarios.



```

1 [
2   {
3     "id": 1,
4     "texto": "Una obra maestra, actuaciones increíbles.",
5     "valoracion": 8,
6     "fechaComentario": "2023-01-05",
7     "usuario": {
8       "email": "rafael@example.com",
9       "nombre": "Rafael",
10      "apellidos": "González",
11      "contraseña": "1234",
12      "fechaNacimiento": "1999-05-15"
13    },
14    "pelicula": {
15      "nombrePelicula": "El Padrino",
16      "sinopsis": "Un poderoso drama criminal que sigue la vida de la familia Corleone en el mundo del crimen organizado.",
17      "paginaOficial": "https://example.com/godfather",
18      "tituloOriginal": "El Padrino",
19      "genero": "Crimen, Drama",
20      "nacionalidad": "Estados Unidos",
21      "duracion": 175,
22      "año": 1972,
23      "distribuidora": "Paramount Pictures",
24      "director": "Francis Ford Coppola",
25      "clasificacionEdad": 18,
26      "otrosDatos": "Ganadora de 3 premios Oscar.",
27      "actores": "Marlon Brando, Al Pacino, James Caan",
28      "url_Imagen": "images/ElPadrino.jpg",
29      "url_Video": "https://www.youtube.com/embed/1oYQx7MKaz0?si=nuCJfNniJgPzuQTb"
30    }
31  },
32  {
33    "id": 2,
34    "texto": "Emocionante de principio a fin.",
35    "valoracion": 7,
36    "fechaComentario": "2023-01-06",
37    "usuario": {
38      "email": "maria@example.com",
39      "nombre": "Maria",
      ...
    }
  }
]

```

Figura 79: Obtener todos los comentarios Postman.

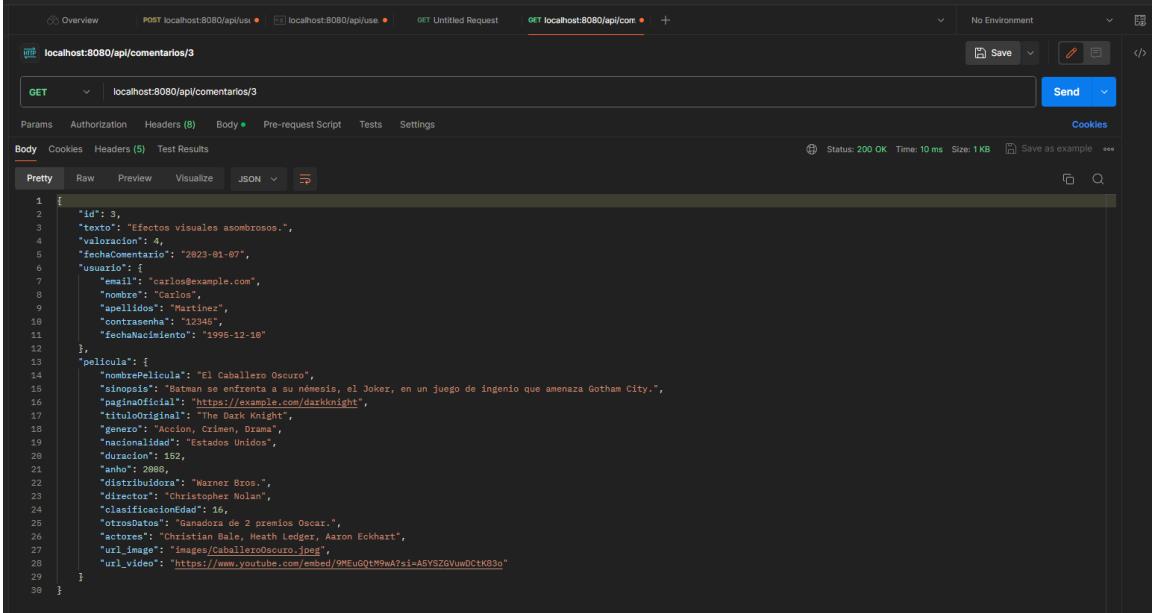
GET - Obtener un comentario por id:

Método: GET

URL: <http://localhost:8080/api/comentarios/{idComentario}>

(Reemplaza {idComentario} con el id del comentario que deseas buscar)

En Postman [7], se selecciona el método GET y se envía la solicitud. Se recibe una respuesta con los detalles del comentario correspondiente al id proporcionado. Este método se utiliza para obtener los detalles de un comentario específico por su id.



```

1 {
2     "id": 3,
3     "texto": "Efectos visuales asombrosos.",
4     "valoracion": 4,
5     "fechaComentario": "2023-01-07",
6     "usuario": [
7         {
8             "email": "carlos@example.com",
9             "nombre": "Carlos",
10            "apellidos": "Martinez",
11            "contraseña": "12345",
12            "fechaNacimiento": "1995-12-10"
13        },
14        {
15            "pelicula": {
16                "nombrePelicula": "El Caballero Oscuro",
17                "sinopsis": "Batman se enfrenta a su nemesis, el Joker, en un juego de ingenio que amenaza Gotham City.",
18                "paginaWeb": "https://example.com/darknight",
19                "trailer": "https://example.com/trailerdarknight",
20                "genero": "Acción, Crimen, Drama",
21                "nacionalidad": "Estados Unidos",
22                "duracion": 152,
23                "año": 2008,
24                "distribuidora": "Warner Bros.",
25                "director": "Christopher Nolan",
26                "clasificacionedad": 16,
27                "otrosDatos": "Ganadora de 2 premios Oscar.",
28                "actores": "Christian Bale, Heath Ledger, Aaron Eckhart",
29                "url_image": "images/CaballeroOscuro.jpg",
30                "url_video": "https://www.youtube.com/embed/9MEuQtM9wA?si=A5YSZGVuw0CtkB3o"
31            }
32        }
33    }
34 }
```

Figura 80: Obtener un comentario por id Postman.

POST - Crear un nuevo comentario:

Método: POST

URL: <http://localhost:8080/api/comentarios>

En la pestaña Body, selecciona raw y el tipo de datos JSON.

Se ingresan los datos del nuevo comentario en formato JSON en el cuerpo de la solicitud.

En Postman [7], se selecciona el método POST, se ingresan los datos del nuevo comentario en formato JSON en el cuerpo de la solicitud y se envía la solicitud. Se recibe una respuesta con los detalles del comentario recién creado. Este método se utiliza para crear un nuevo comentario.

PUT - Actualizar un comentario existente:

Método: PUT

URL: <http://localhost:8080/api/comentarios/{idComentario}>

(Reemplaza {idComentario} con el id del comentario que deseas actualizar)

En la pestaña Body, se selecciona raw y el tipo de datos JSON.

Se ingresan los campos que se quieren actualizar en formato JSON en el cuerpo de la solicitud.

En Postman [7], se selecciona el método PUT, se ingresan los datos actualizados en formato JSON en el cuerpo de la solicitud y se envía la solicitud. Se recibe una respuesta con los detalles del comentario actualizado. Este método se utiliza para actualizar los detalles de un comentario existente.

DELETE - Eliminar un comentario:

Método: DELETE

URL: <http://localhost:8080/api/comentarios/{idComentario}>

(Reemplaza {idComentario} con el id del comentario que deseas eliminar)

En Postman [7], se selecciona el método DELETE y se envía la solicitud. Se recibe una respuesta indicando que el comentario ha sido eliminado correctamente. Este método se utiliza para eliminar un comentario existente.

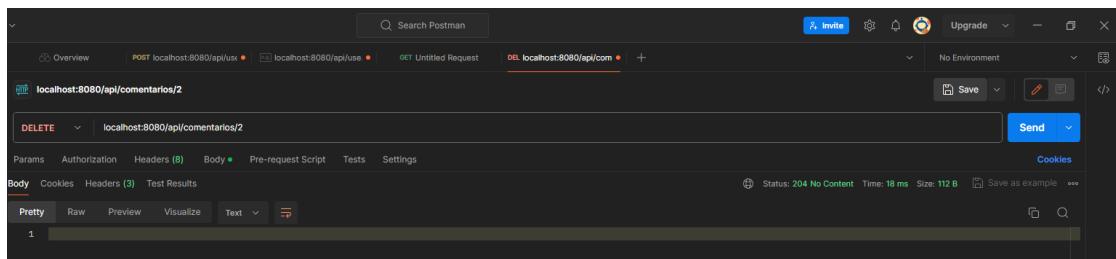


Figura 81: Eliminar un comentario Postman.

La fase de pruebas con Postman [7] ha servido para verificar y validar el correcto funcionamiento de los endpoints de la API. En este caso, he utilizado Postman [7] para probar los métodos GET, POST, PUT y DELETE de la API.

Estas pruebas permiten:

- Comprobar que los endpoints están correctamente configurados y responden como se espera.
- Verificar que los datos se están procesando correctamente, tanto en las solicitudes entrantes como en las respuestas salientes.
- Identificar y corregir errores o problemas en el código antes de que la API se despliegue en un entorno de producción. Que será la segunda fase del proyecto.
- Hay que asegurar que la API es capaz de manejar correctamente diferentes tipos de solicitudes y datos.
- Por último, vamos a comprobar la base de datos para ver si todos los métodos CRUD se han realizado con éxito.

The screenshot shows a MySQL result grid with the following data:

	email	apellidos	contrasenha	fecha_nacimiento	nombre
▶	carlos@example.com	Martinez	12345	1995-12-10	Carlos
▶	maria@example.com	Lopez	12345	1985-09-22	Maria
▶	rafael@example.com	Gonzalez	1234	1990-05-15	Rafael
*	NULL	NULL	NULL	NULL	NULL

Figura 82: Base de datos MySQL, tras inyección Postman de usuarios.

The screenshot shows a MySQL result grid with the following data:

	nombre_pelicula	actores	año	clasificacion_edad	director	distribuidora	duracion	genero	nacionalidad	otros_datos
▶	Cadena Perpetua	Tim Robbins, Morgan Freeman	1994	15	Frank Darabont	Columbia Pictures	142	Drama	Estados Unidos	Basada en la novela de Stephen King.
▶	Cars	Owen Wilson, Bonnie Hunt	2006	3	John Lasseter	Pixar	120	Animacion	Estadounidense	Datos adicionales sobre la película Cars.
▶	Cars 2	Owen Wilson, Larry the Cable Guy	2011	3	John Lasseter	Pixar	130	Animacion	Estadounidense	Datos adicionales sobre la película Cars 2.
▶	Cars 3	Owen Wilson, Cristela Alonzo	2017	3	Brian Fee	Pixar	110	Animacion	Estadounidense	Datos adicionales sobre la película Cars 3.
▶	El Caballero Oscuro	Christian Bale, Heath Ledger, Aaron Eckhart	2008	16	Christopher Nolan	Warner Bros.	152	Accion, Crimen, Drama	Estados Unidos	Ganadora de 2 premios Oscar.
▶	El Padino	Marlon Brando, Al Pacino, James Caan	1972	18	Francis Ford Coppola	Paramount Pictures	175	Crimen, Drama	Estados Unidos	Ganadora de 3 premios Oscar.
▶	Harry Potter y la Piedra Filosofal	Daniel Radcliffe, Emma Watson, Rupert Grint	2001	7	Chris Columbus	Warner Bros.	152	Aventura, Familia, Fantasia	Reino Unido	Basada en la novela de J.K. Rowling.
▶	Jurassic Park	Sam Neill, Laura Dern, Jeff Goldblum	1993	12	Steven Spielberg	Universal Pictures	127	Aventura, Ciencia Ficcion	Estados Unidos	Basada en la novela de Michael Crichton.
▶	Kung Fu Panda	Jack Black, Angelina Jolie	2008	3	Mark Osborne, John Stevenson	DreamWorks Animation	95	Animacion	Estadounidense	Datos adicionales sobre la película Kung Fu Panda.
▶	Matrix	Keanu Reeves, Laurence Fishburne, Carrie-Ann...	1999	15	Lana Wachowski, Lilly Wachowski	Warner Bros.	136	Accion, Ciencia Ficcion	Estados Unidos	Ganadora de 4 premios Oscar.
▶	One Piece	Mayumi Tanaka, Kazuya Nakai	2022	12	Eiichiro Oda	Toei Animation	120	Animacion	Japonesa	Datos adicionales sobre la película One Piece.
▶	Rocky	Sylvester Stallone, Talia Shire, Burgess Meredith	1976	12	John G. Avildsen	United Artists	120	Drama	Estadounidense	Datos adicionales sobre la película Rocky.
▶	Titanic	Leonardo DiCaprio, Kate Winslet	1997	12	James Cameron	20th Century Fox	195	Drama, Romance	Estados Unidos	Ganadora de 11 premios Oscar.
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figura 83: Base de datos MySQL, tras inyección Postman de películas.

Las pruebas realizadas han sido un éxito.

10. USO DE LA APLICACIÓN

10.1 GUÍA DEL USUARIO

En este caso voy a simular la creación de un usuario desde cero y el uso de este en la aplicación como sería el caso común de una persona que quiera usar mi plataforma.

Cuando accedemos a la plataforma lo primero que vemos son los diferentes contenidos disponibles ya sean de futbol, de f1, de películas, de contenidos en directo, las comunidades, la zona de perfil y la zona de iniciar sesión. Cabe resaltar que el buscador de los contenidos si este habilitado sin estar registrado.

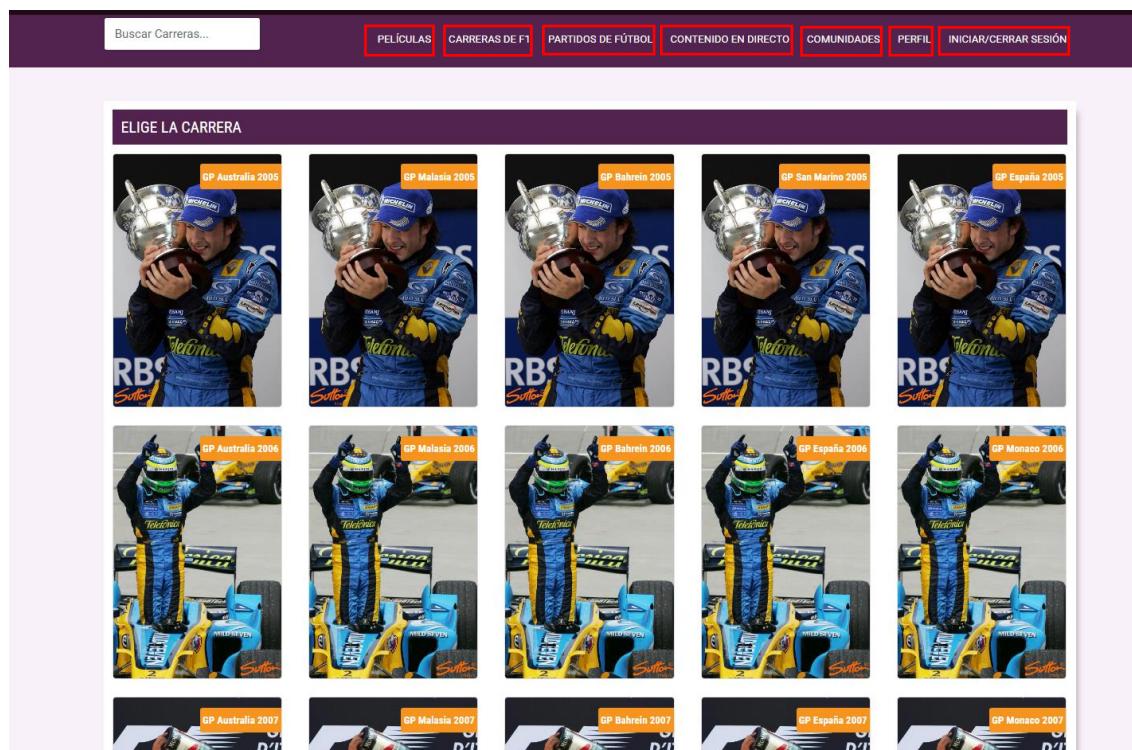


Figura 84: Panel con los diferentes contenidos de la aplicación.

La lógica que esta implementada es que, si seleccionas algún contenido para ver o le das a las comunidades, o contenido en directo o perfil, te envía directamente al panel de inicio de sesión para que te registres.

Dentro del panel de inicio de sesión / registro, si intentamos iniciamos sesión sin estar registrados o iniciamos sesión de forma errónea por poner algún dato mal o lo que sea, la plataforma maneja tres errores. Y también hay control de campos vacíos. Los errores que se maneján son; Por si el correo que se pone para el inicio de sesión e inicias sesión, te pide directamente que te registres.

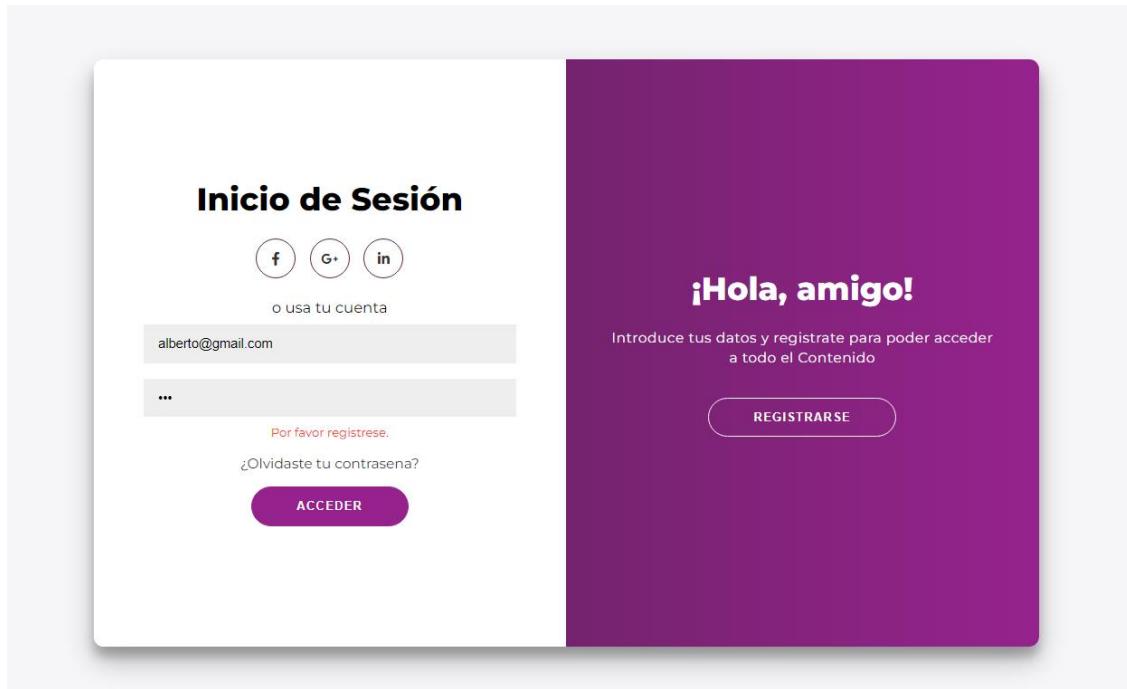


Figura 85: Panel de Inicio de sesión de un usuario que no está registrado.

Otro de los errores es por poner el usuario y/o contraseña incorrecta y por último el control de campo vacío.

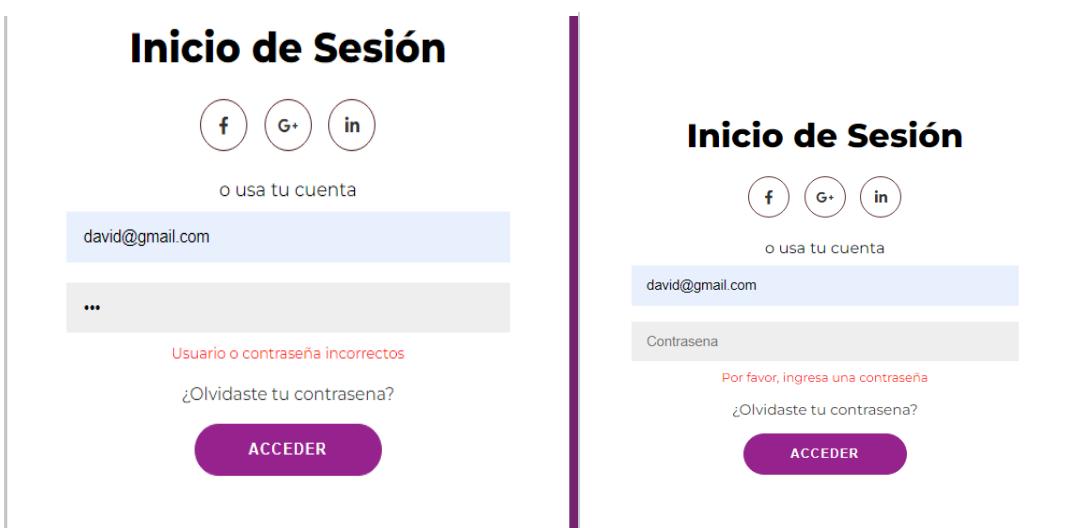


Figura 86: Panel de Inicio de sesión, con los posibles errores.

Dentro del panel de inicio de sesión igual, hay un comprobante para el nickname y para el correo por si pones uno que ya existe en el sistema, y un manejo de errores por campo vacío.

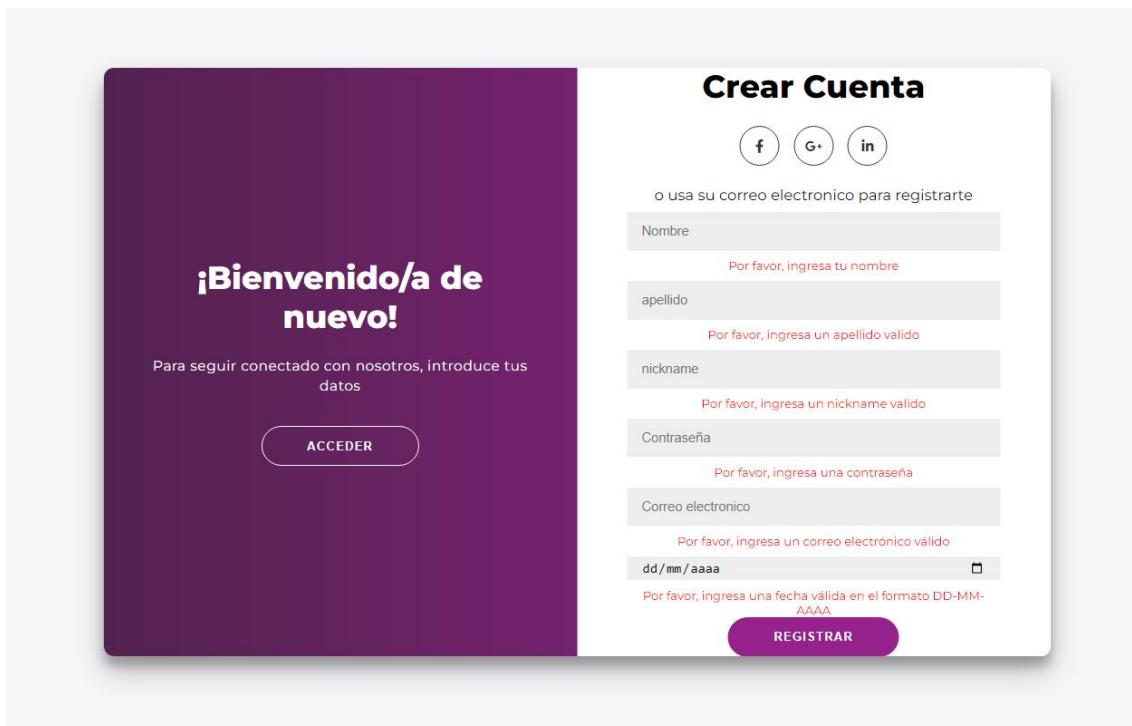


Figura 87: Panel de Registro con errores en todos los campos.

Una vez completado todos los campos del registro, la plataforma nos lanza a un panel de selección del panel de suscripción y en base a nuestros gustos o capacidad económica seleccionamos uno, cabe resaltar que si nos registramos y pasamos de esto cuando vayamos a iniciar sesión nos va a mandar de nuevo al panel de selección del panel de suscripción.



Figura 88: Panel de selección de los planes de suscripción.

Si seleccionamos el plan de suscripción gratis el proceso de registro habrá terminado y estaremos listos para iniciar sesión y empezar a ver el contenido de la plataforma, si por el contrario seleccionamos uno de los dos planes de suscripción de pago nos envía a un panel de pago para que pongamos la tarjeta.



Nombre del titular: ALBERTO LOPEZ

Fecha esp.: 02/12

Numero de tarjeta: 6759 6498 2643 8453

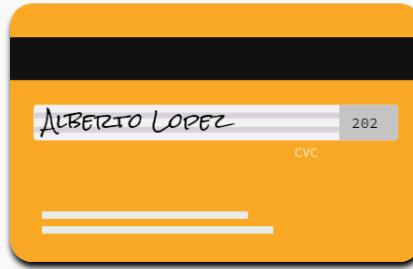
Nombre del titular: Alberto Lopez

Numero de tarjeta: 6759 6498 2643 8453

Fecha de caducidad (mm/yy): 02/12

Codigo de seguridad: 202

Procesar pago



Nombre del titular: ALBERTO LOPEZ

CVC: 202

Nombre del titular: Alberto Lopez

Numero de tarjeta: 6759 6498 2643 8453

Fecha de caducidad (mm/yy): 02/12

Codigo de seguridad: 202

Procesar pago

Figura 89: Panel de pasarela de pago con tarjeta, MasterCard.

Hay cargadas 10 tipos diferentes de tarjeta en ese botón se selecciona el modelo de tarjeta que uses. Aquí dejo otro ejemplo de otro modelo de tarjeta.



Nombre del titular: ALBERTO LOPEZ

Fecha esp.: 02/12

Numero de tarjeta: 4000 0566 5566 5556

Nombre del titular: Alberto Lopez

Numero de tarjeta: 4000 0566 5566 5556

Fecha de caducidad (mm/yy): 02/12

Codigo de seguridad: 202

Procesar pago



Nombre del titular: ALBERTO LOPEZ

CVC: 202

Nombre del titular: Alberto Lopez

Numero de tarjeta: 4000 0566 5566 5556

Fecha de caducidad (mm/yy): 02/12

Codigo de seguridad: 202

Procesar pago

Figura 90: Panel de pasarela de pago con tarjeta, Visa.

La plataforma tiene control de validación de pagos o validación de selección del plan de suscripción por lo que, si dejas el proceso de pago a medias o si al pago no es correcto, cuando inicies sesión te volverá a enviar al proceso de pago o de selección del plan de suscripción de nuevo.

Una vez completado todo el proceso de registro ya podemos iniciar sesión y acceder a la plataforma que se divide en 6 secciones, películas, carreras de f1, partidos de fútbol, contenido en directo, comunidades de usuarios y una sección de perfil para ver y actualizar si se desea los datos personales. Comenzamos con el primero de los contenidos, las películas.

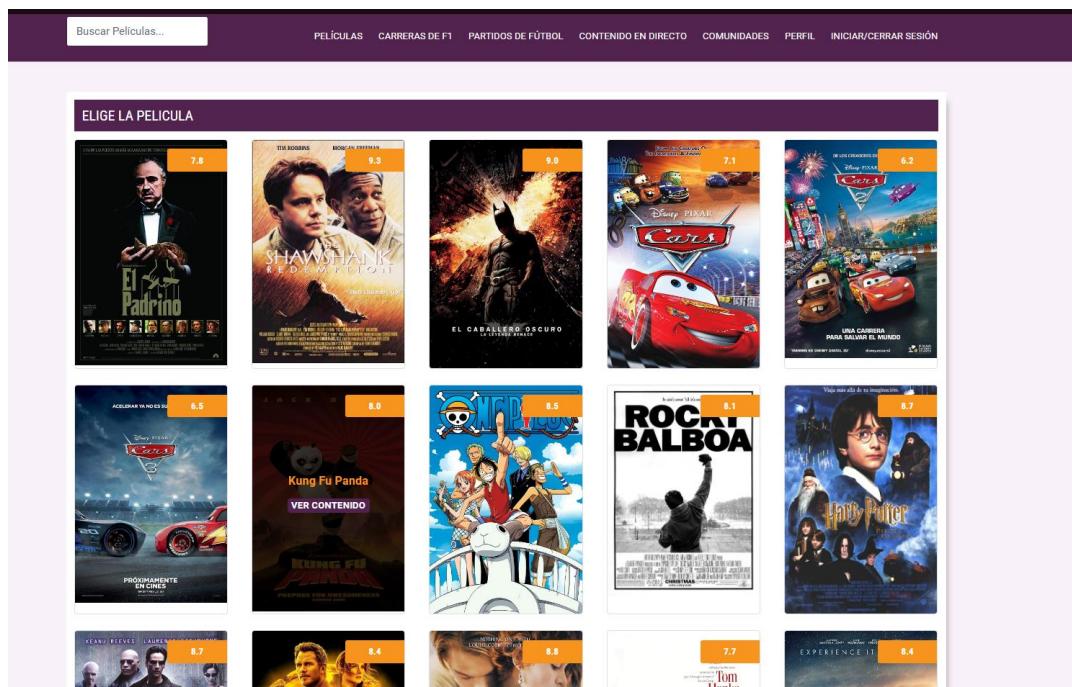


Figura 91: Panel de películas en la aplicación.

Tenemos una serie de películas a gusto del consumidor resaltando sus valoraciones algo que ayuda mucho a la hora de selección de los contenidos. Se implementa también un buscador para hacer más sencillo el proceso de selección de una película. En el que aparecerá una nueva sección llamada resultados de búsqueda con el contenido que se encuentre que encaje con el nombre establecido en el buscador.

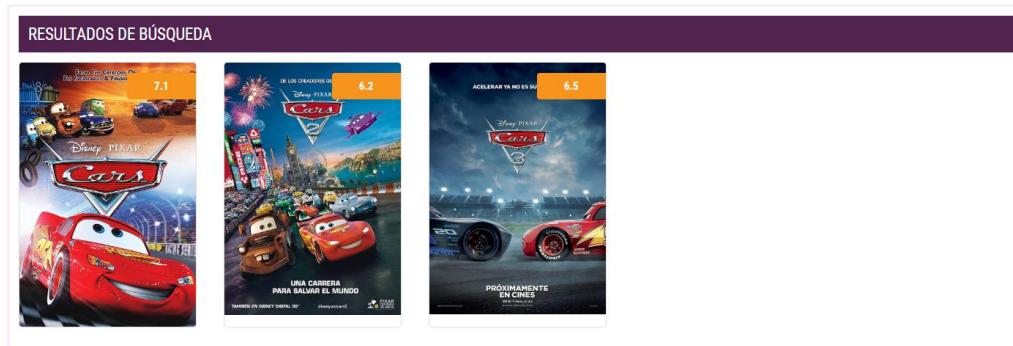


Figura 92: Panel de buscador de películas en la aplicación.

Cuando el usuario clique en cualquiera de las películas entra al panel individual de cada película donde ya podemos ver su contenido, información detallada de esta y la sección de comentarios que si tiene el plan básico o superior podrá hacerlos si no solo ver los comentarios de otros usuarios de la plataforma.

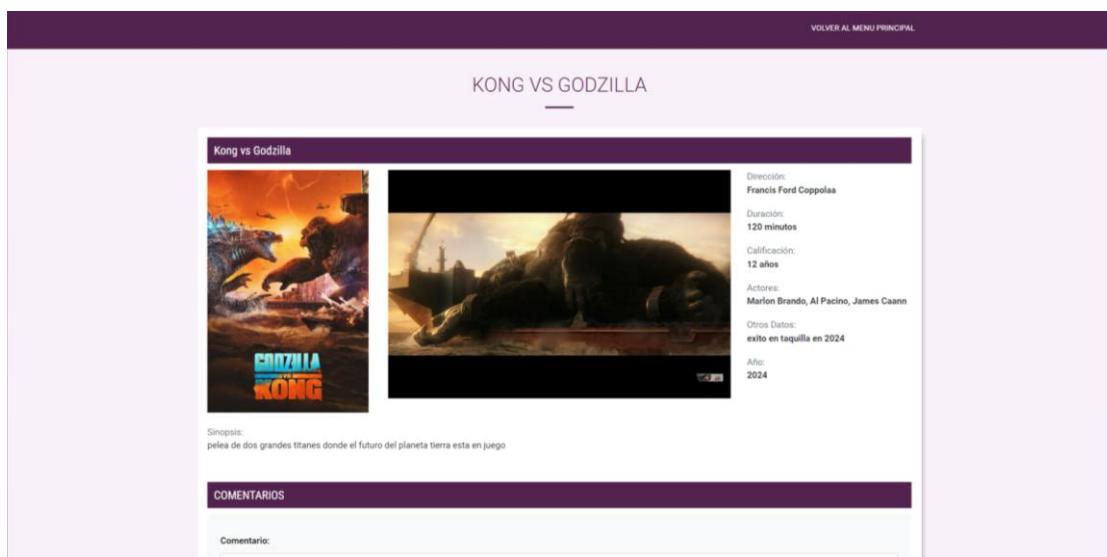


Figura 93: Panel de una película para ser visualizada, en la aplicación.

Pasamos al siguiente contenido las carreras de f1, en este caso tenemos diferentes carreras ordenadas por años.

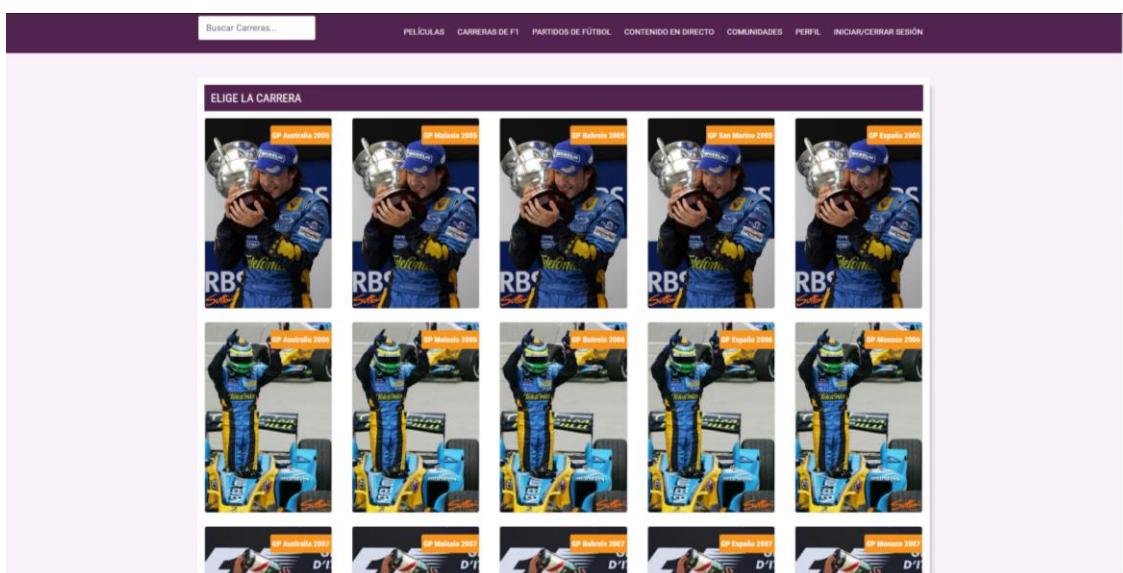


Figura 94: Panel de carreras de F1 en la aplicación.

Se implementa también un buscador para hacer más sencillo el proceso de selección de una carrera de f1. En el que aparecerá una nueva sección llamada resultados de búsqueda con el contenido que se encuentre que encaje con el nombre establecido en el buscador.

Figura 95: Panel de buscador de carreras de F1, en la aplicación.

Cuando el usuario clique en cualquiera de las carreras de f1 entra al panel individual de cada carrera donde ya podemos ver su contenido, información detallada de esta y la sección de comentarios que si tiene el plan básico o superior podrá hacerlos si no solo ver los comentarios de otros usuarios de la plataforma.

Figura 96: Panel de una carrera para ser visualizada, en la aplicación.

Pasamos al siguiente contenido los partidos de fútbol, en este caso tenemos diferentes partidos ordenados por competición, están la liga española y la champions legue y ordenados también por año.

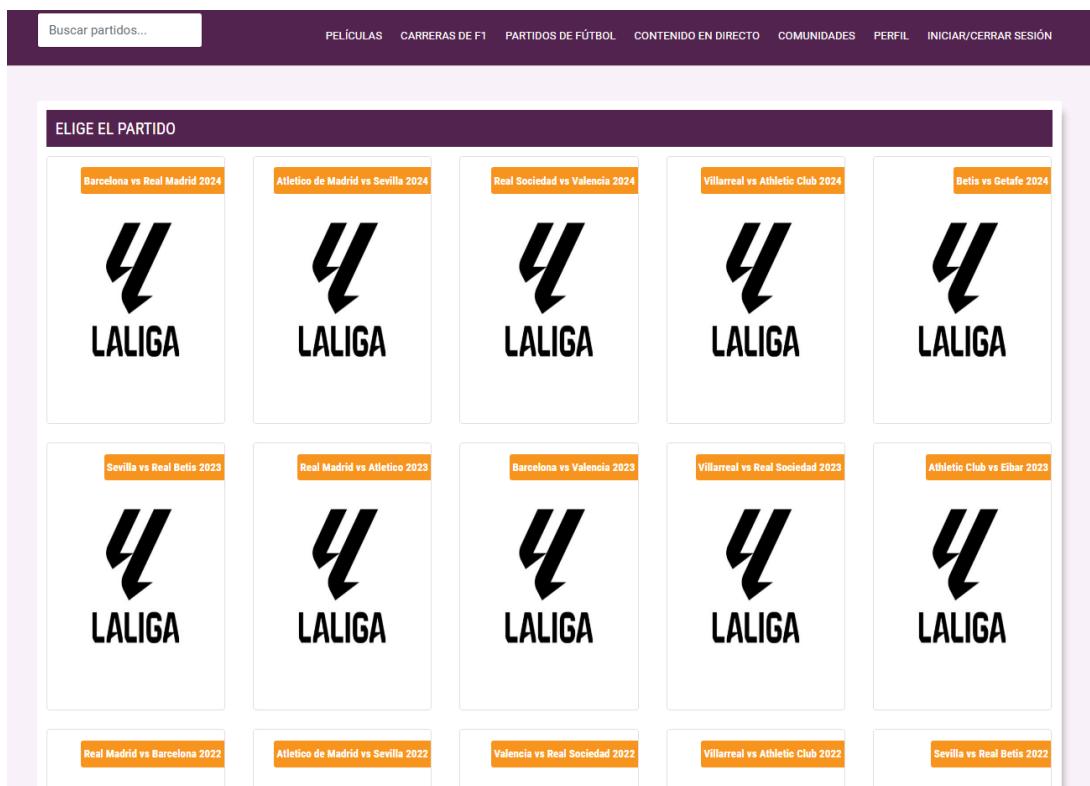


Figura 97: Panel de partidos de fútbol en la aplicación.

Se implementa también un buscador para hacer más sencillo el proceso de selección de un partido de fútbol. En el que aparecerá una nueva sección llamada resultados de búsqueda con el contenido que se encuentre que encaje con el nombre establecido en el buscador.

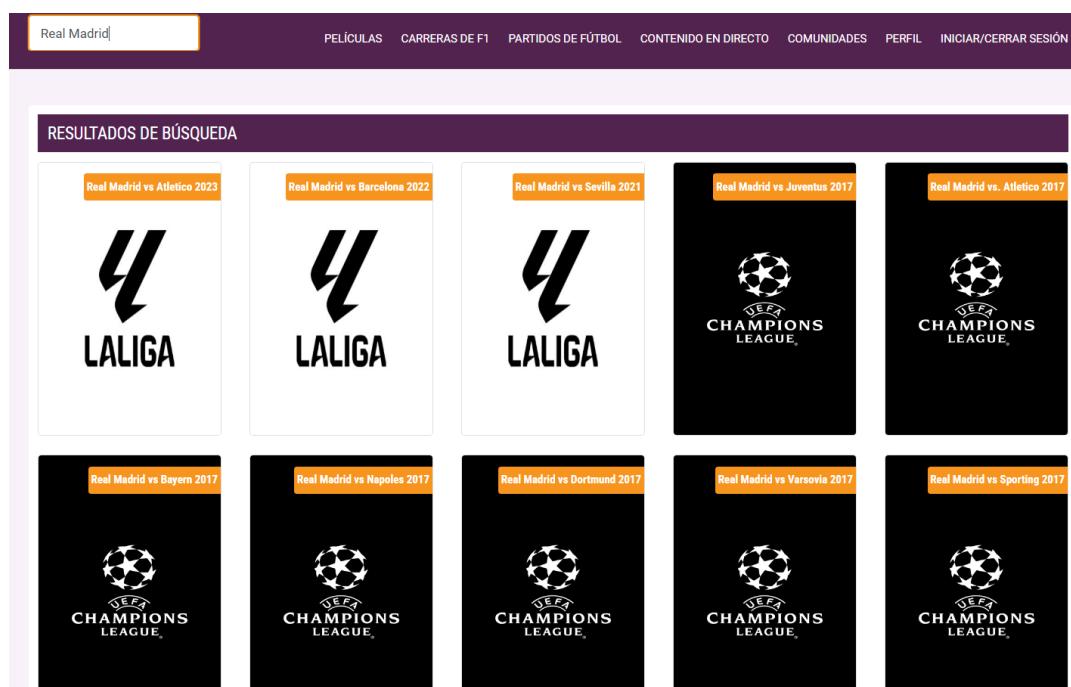


Figura 98: Panel del buscador de partidos de fútbol en la aplicación.

Cuando el usuario clique en cualquiera de los partidos de fútbol disponibles entra al panel individual de cada partido donde ya podemos ver su contenido, información detallada de este y la sección de comentarios que si tiene el plan básico o superior podrá hacerlos si no solo ver los comentarios de otros usuarios de la plataforma.



VOLVER AL MENU PRINCIPAL REGISTRARSE

REAL MADRID VS VARSOVIA 2017

Real Madrid vs Varsovia 2017





Año:
2017

Estadio:
Estadio Santiago Bernabéu

Competición:
Champions League

Equipos:
Real Madrid, Legia de Varsovia

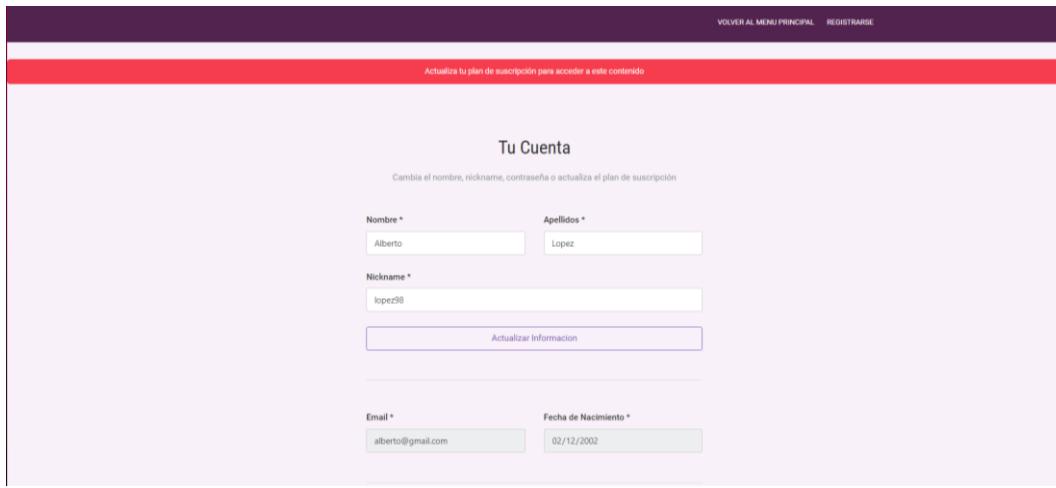
Jugadores:
Cristiano Ronaldo, Gareth Bale, Vadis Odjidja-Ofoe, Keylor Navas

Otros Datos:
Fase de grupos de la Champions League 2017. Partido 3/6

Descripción:
El Real Madrid se enfrentó al Legia de Varsovia en la fase de grupos de la Champions League 2017. Fue un emocionante encuentro que terminó con la victoria del Real Madrid.

Figura 99: Panel para la visualización de un partido de fútbol en la aplicación.

El siguiente panel es el de contenido en directo, donde se verán en caso de que lo haya el contenido en directo y si en ese tramo horario no hay contenido en directo se verá los futuros contenidos en directo para que los usuarios sepan cuando se va emitir el contenido exclusivo. En este caso solo se podrá ver con un plan de suscripción pro ya que es la funcionalidad más exclusiva de la plataforma, si intentas ver el contenido sin el plan pro, la plataforma te envía directamente al perfil para que actualices a él plan pro.



VOLVER AL MENU PRINCIPAL REGISTRARSE

Actualiza tu plan de suscripción para acceder a este contenido

Tu Cuenta

Cambia el nombre, nickname, contraseña o actualiza el plan de suscripción

Nombre *	Apellidos *
Alberto	Lopez

Nickname *
lopez98

Actualizar Información	
------------------------	--

Email *	Fecha de Nacimiento *
alberto@gmail.com	02/12/2002

Figura 100: Error de acceso al contenido en directo, en la aplicación.

Si tenemos el plan de suscripción pro si podemos entrar a el panel de contenido en directo.

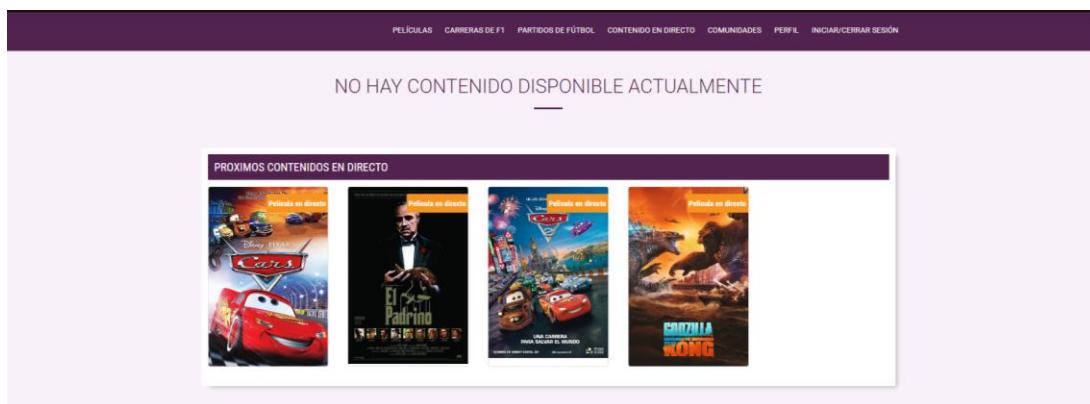


Figura 101: Panel de contenido en directo, sin un contenido en directo.

Este es uno de los casos en el que en ese tramo horario no hay contenidos en directo y podemos ver los futuros contenidos en directo.

This screenshot shows a browser window with the URL 'localhost:8080/api/LiveContent'. The page has a purple header with social media links and navigation options. A green banner at the top says 'Contenido en directo actualizado con éxito'. Below it, a movie entry for 'EL PADRINO' is shown. The movie poster is on the left, and a thumbnail for the 'El Padrino: 50 años - Tráiler oficial' is on the right, featuring Marlon Brando. To the right of the thumbnail, event details are listed: 'Hora de inicio: 10/6/24, 20:58', 'Hora de fin: 10/6/24, 21:00', and 'Tipo: Película en directo'. Below the main entry, a 'PROXIMOS CONTENIDOS EN DIRECTO' section is visible, showing the same four movies as in Figure 101.

Figura 102: Panel de actualización de un contenido en directo en la aplicación.

Por último, en lo que respecta a los contenidos, aunque en este caso no es para verlos si no para comentarlos y debatir con otros usuarios, tenemos las comunidades de chats para que los usuarios de la plataforma hablen entre ellos y comparten sus opiniones y conocimientos sobre los diferentes contenidos de la plataforma.

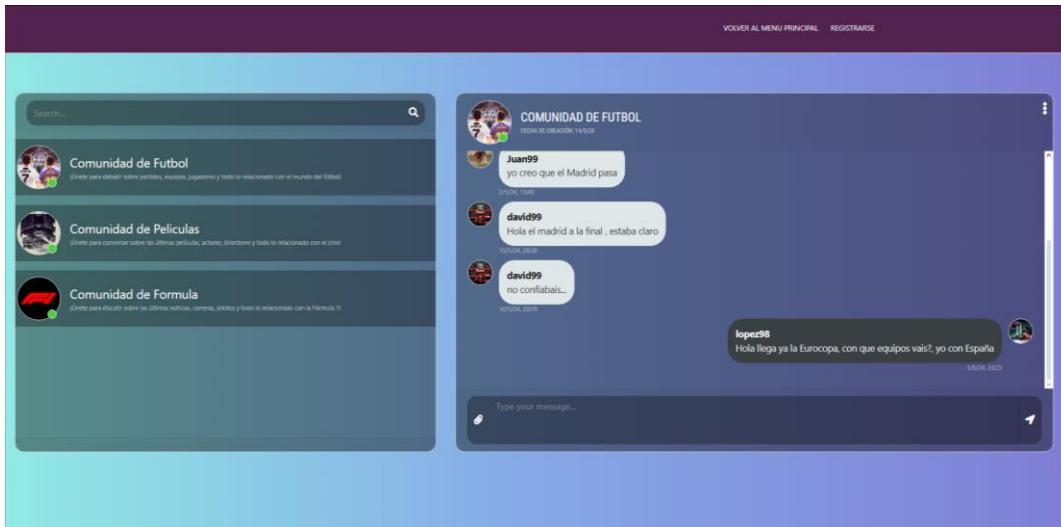


Figura 103: Panel de las comunidades de chats, en la aplicación.

Dentro de las comunidades disponibles podemos hablar en la que queramos siempre que tengamos un plan de suscripción básico o superior, en el caso de tener un plan gratuito solo podremos ver los chats. Aquí muestro un mensaje del usuario que he creado de Alberto en la comunidad de fútbol. Por último, en el manual para usuarios tenemos el panel del perfil donde se puede encontrar la información personal del usuario que ha iniciado sesión en este caso del Alberto, nos permite cambiar el nombre, apellidos y nickname, este último siempre y cuando no coincida con uno que ya existe.

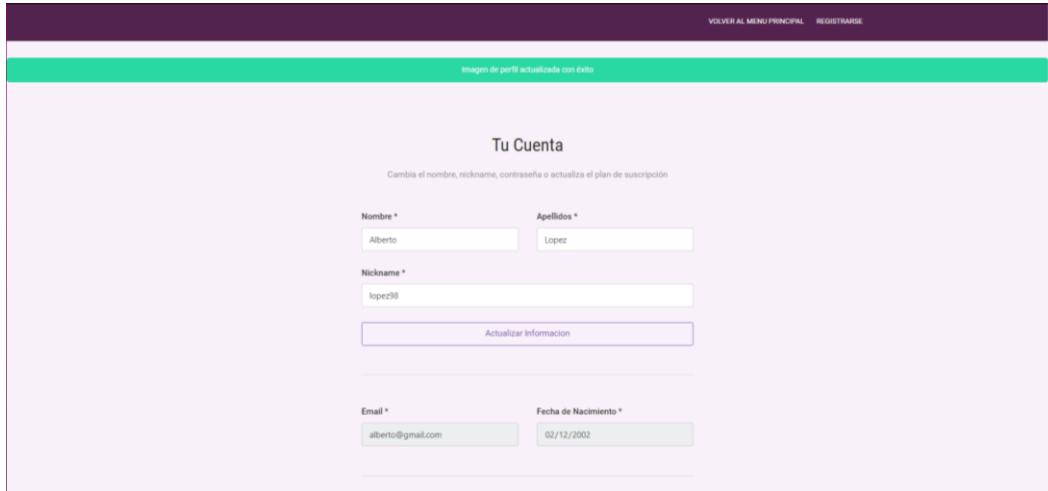


Figura 104: Panel de actualización de datos del usuario en la aplicación.

Para cada acción que se realice la plataforma maneja mensajes que salen en cuando realizamos alguna acción de cambio de datos, notificándonos si la acción se ha realizado correctamente, o si ha surgido algún error y nos explica por qué.



Figura 105: Mensaje de actualización de datos exitosos en el perfil del usuario.

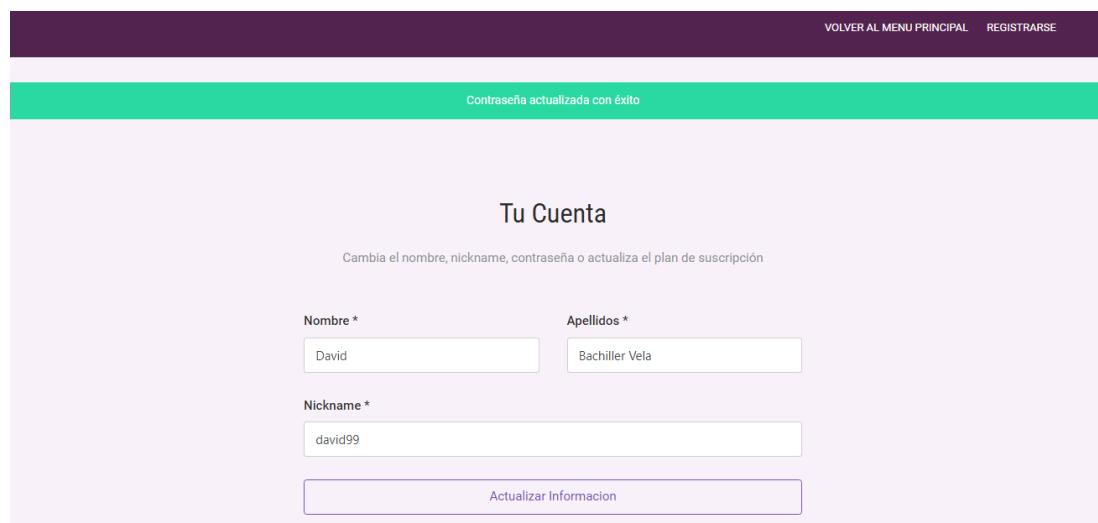


Figura 106: Mensaje de actualización de contraseña con éxito en la aplicación.

10.2 GUÍA DE ADMINISTRADOR

Lo primero de todo es entender el rol del administrador en la plataforma que se basa en dos funcionalidades gestión del contenido multimedia y gestión de usuarios, la gran pregunta es quien asigna los administradores, y es el super admin culla única función es la de asignar o desasignar administradores es decir pasar a un usuario de rol de user a rol de admin o viceversa. En cuanto a las funcionalidades de gestión de contenido el admin tiene secciones especiales dentro de los principales paneles de la aplicación para la gestión del contenido multimedia, que se va a basar en la creación, actualización y borrado del contenido.

Vamos a empezar con el panel de películas, una vez que te registres y hayas sido asignado con el rol de administrador vas a ver funcionalidades exclusivas, en el caso de las películas vemos que tenemos un panel que nos permite añadir nuevas películas.

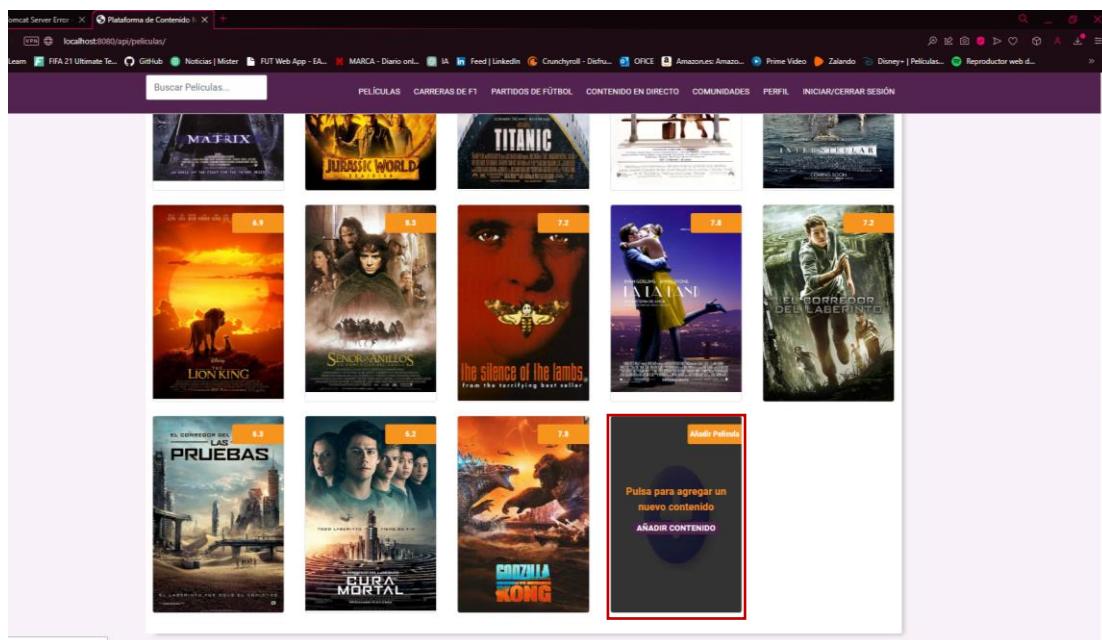


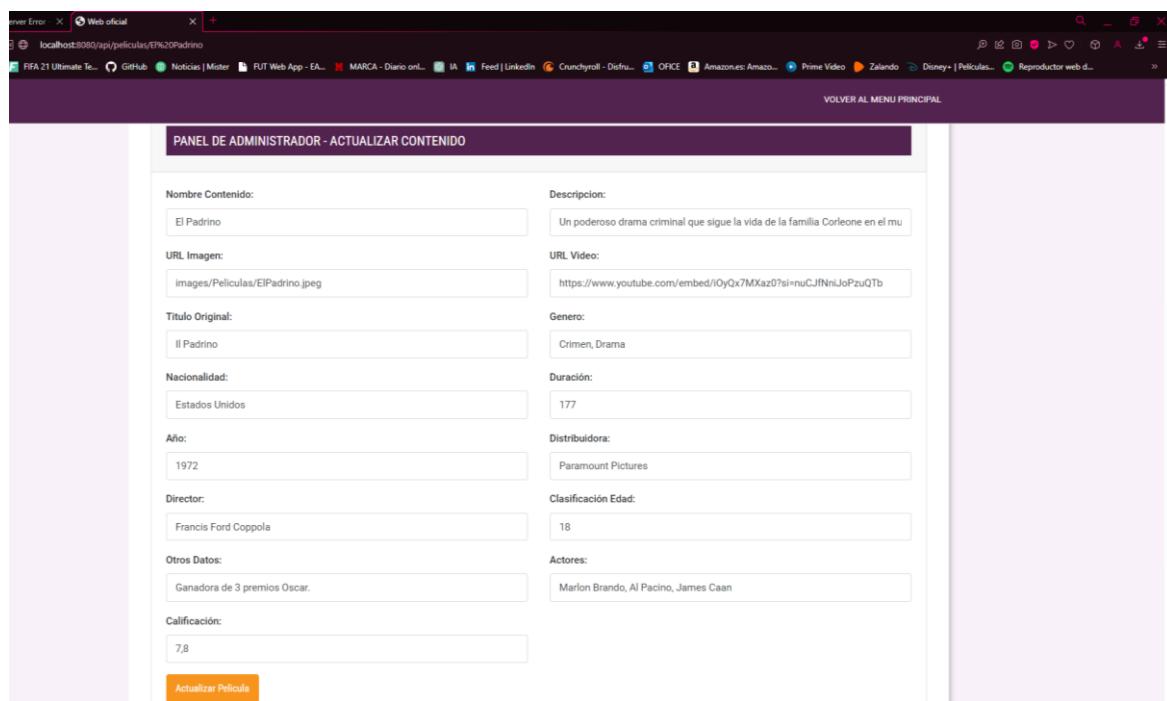
Figura 107: Panel de agregar películas, administrador.

Cuando pulsamos para añadir nuevo contenido nos envía a un panel donde se gestiona toda la creación de nuevos contenidos dentro de la plataforma, tanto contenido en directo, como películas, carreras de f1 y partidos de fútbol, para llenar los datos del nuevo contenido y crearlo. En el caso de que se cree correctamente veremos un mensaje de creación correcta y si se produce algún problema también saldrá un mensaje indicando el motivo del error por lo que ha sucedido.

PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO PELÍCULAS	
Nombre Contenido:	Descripción:
<input type="text"/>	<input type="text"/>
URL Imagen:	URL Video:
<input type="text"/>	<input type="text"/>
Título Original:	Género:
<input type="text"/>	<input type="text"/>
Nacionalidad:	Duración:
<input type="text"/>	<input type="text"/>
Año:	Distribuidora:
<input type="text"/>	<input type="text"/>
Director:	Clasificación Edad:
<input type="text"/>	<input type="text"/>
Otros Datos:	Actores:
<input type="text"/>	<input type="text"/>
Calificación:	
<input type="text"/>	
<input type="button" value="Añadir Película"/>	

Figura 108: Panel para crear una nueva película, administrador.

Dentro del contenido de cada película, debajo del contenido vemos paneles especiales para actualizar el contenido o borrarlo si se desea, en ese caso el admin no verá los comentarios ya que su única función en este caso es gestionar el contenido.



The screenshot shows a web browser window with the URL `localhost:8080/api/peliculas/El%20Padrino`. The page title is "PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO". The form fields are as follows:

Nombre Contenido:	Descripción:
El Padrino	Un poderoso drama criminal que sigue la vida de la familia Corleone en el mu...
URL Imagen:	URL Video:
images/Peliculas/ElPadrino.jpeg	https://www.youtube.com/embed/IoQx7MXazD?si=nuCJNniJoPzuQTb
Título Original:	Género:
Il Padrino	Crimen, Drama
Nacionalidad:	Duración:
Estados Unidos	177
Año:	Distribuidora:
1972	Paramount Pictures
Director:	Clasificación Edad:
Francis Ford Coppola	18
Otros Datos:	Actores:
Ganadora de 3 premios Oscar.	Marlon Brando, Al Pacino, James Caan
Calificación:	
7,8	

Actualizar Película button at the bottom.

Figura 109: Panel para actualizar películas, administrador.



The screenshot shows a web browser window with the URL `localhost:8080/api/peliculas/El%20Padrino`. The page title is "PANEL DE ADMINISTRADOR - BORRAR CONTENIDO". The form fields are as follows:

Contraseña:
<input type="password"/>
Confirmar Contraseña:
<input type="password"/>
Borrar Película button at the bottom.

Figura 110: Panel para borrar películas, administrador.

El administrador ya ve los campos como están actualmente y los puede cambiar si lo desea. También tiene funcionalidades para los contenidos de f1, en el panel principal tiene la opción de agregar nuevos contenidos de f1.

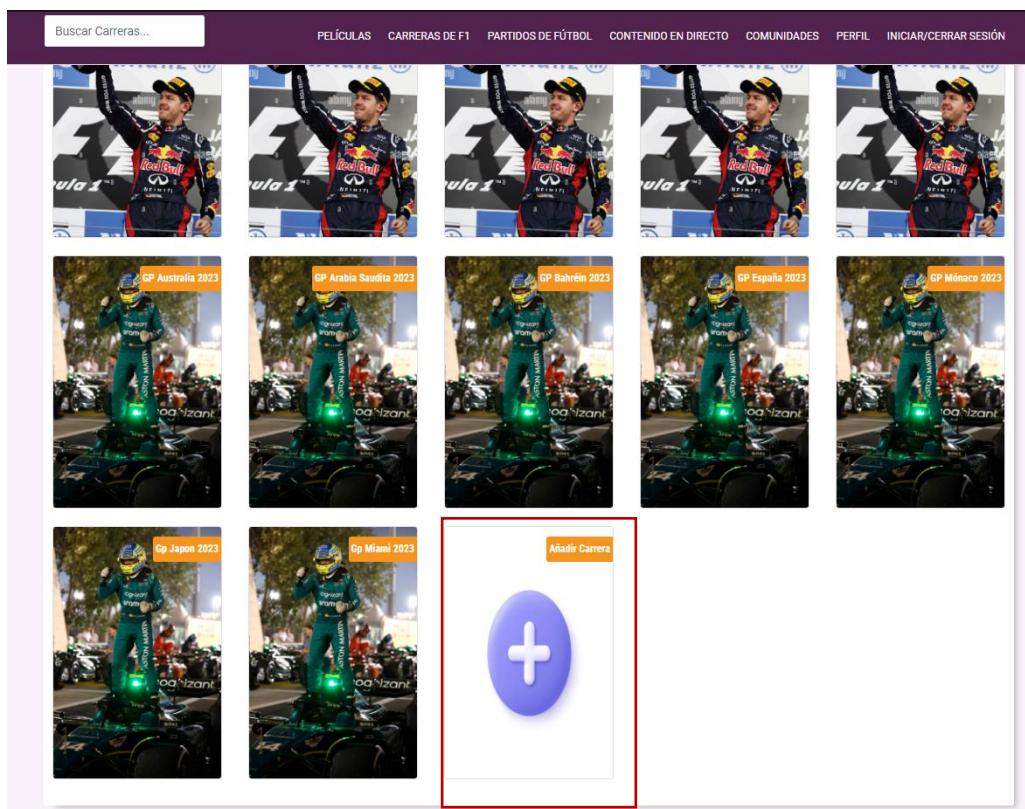


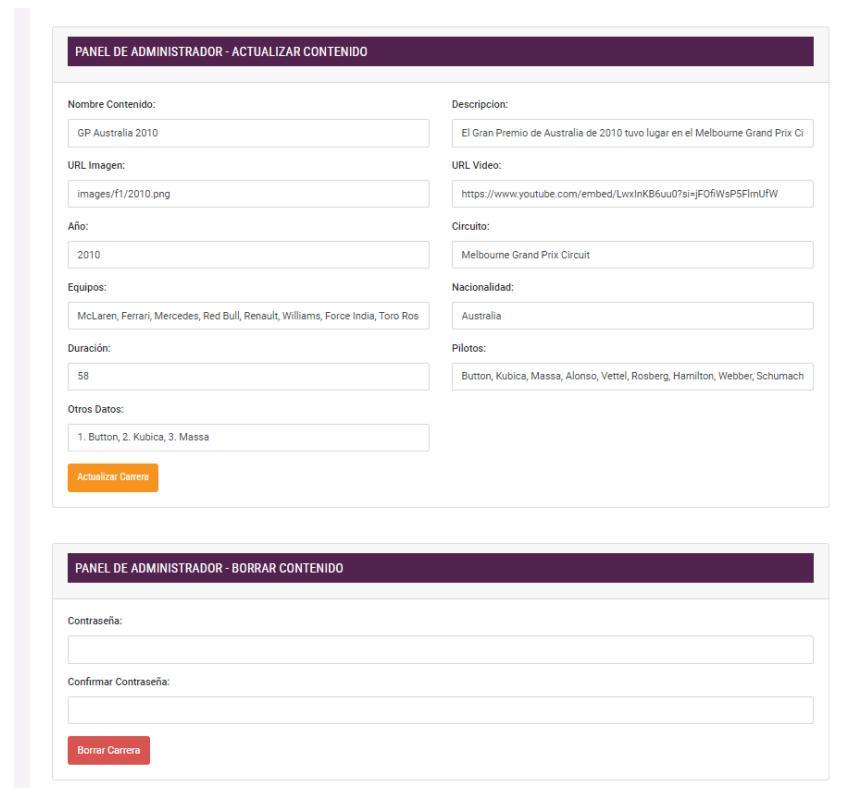
Figura 111: Panel para añadir carreras, administrador.

Cuando pulsamos para añadir nuevo contenido nos envía a un panel donde se gestiona toda la creación de nuevos contenidos dentro de la plataforma, tanto contenido en directo, como películas, carreras de f1 y partidos de fútbol, para llenar los datos del nuevo contenido y crearlo. En el caso de que se cree correctamente veremos un mensaje de creación correcta y si se produce algún problema también saldrá un mensaje indicando el motivo del error por lo que ha sucedido.

The screenshot shows a browser window with a form titled 'PANEL DE ADMINISTRADOR - AÑADIR CONTENIDO F1'. The form has several input fields: 'Nombre Contenido' (Content Name), 'Descripción' (Description), 'URL Imagen' (Image URL) and 'URL Video' (Video URL), 'Año' (Year), 'Circuito' (Circuit), 'Equipos' (Teams), 'Nacionalidad' (Nationality), 'Duración' (Duration), 'Pilotos' (Drivers), 'Otros Datos' (Other Data), and an 'Añadir Carrera' (Add Race) button at the bottom. Above the form, there's a header with the URL 'localhost:8080/api/peliculas/añadirPelicula' and a 'VOLVER AL MENU PRINCIPAL' (Return to Main Menu) link.

Figura 112: Panel para añadir carreras de f1, administrador.

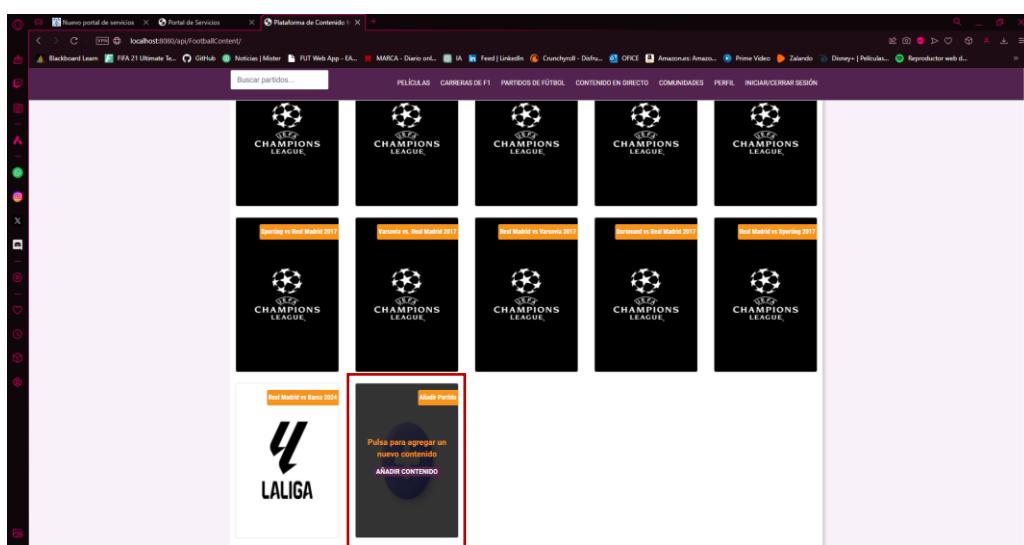
Dentro del contenido de cada carrera de f1, debajo del contenido vemos paneles especiales para actualizar el contenido o borrarlo si se desea, en ese caso el admin no vera los comentarios ya que su única función en este caso es gestionar el contenido.



The image shows two separate panels side-by-side. The left panel is titled 'PANEL DE ADMINISTRADOR - ACTUALIZAR CONTENIDO'. It contains fields for Nombre Contenido (GP Australia 2010), Descripción (El Gran Premio de Australia de 2010 tuvo lugar en el Melbourne Grand Prix Ci...), URL Imagen (images/f1/2010.png), URL Video (https://www.youtube.com/embed/LwxlnKB6uu0?si=jPOfWsP5FlmUW...), Año (2010), Circuito (Melbourne Grand Prix Circuit), Equipos (McLaren, Ferrari, Mercedes, Red Bull, Renault, Williams, Force India, Toro Rosso), Nacionalidad (Australia), Duración (58), Pilotos (Button, Kubica, Massa, Alonso, Vettel, Rosberg, Hamilton, Webber, Schumach), Otros Datos (1. Button, 2. Kubica, 3. Massa), and an 'Actualizar Carrera' button. The right panel is titled 'PANEL DE ADMINISTRADOR - BORRAR CONTENIDO' and has fields for Contraseña and Confirmar Contraseña, with a 'Borrar Carrera' button.

Figura 113: Panel para actualizar carreras y borrarlas, administrador.

Otras de las funcionalidades dentro del panel de administrador son en el contenido de partidos, el administrador tiene un campo para añadir si lo desea nuevos partidos de fútbol, cuando pulse le enviara al panel de creación de nuevos contenidos que es el mismo que el del resto de contenidos de plataforma.



The image shows a grid of football match thumbnails. Most thumbnails are for Champions League matches (Real Madrid vs Barcelona 2021, Valencia vs Real Madrid 2017, Real Madrid vs Valencia 2017, Borussia vs Real Madrid 2017, Real Madrid vs Sporting 2017). One thumbnail is for La Liga (Real Madrid vs Barcelona 2021). A modal window is open in the bottom center, titled 'Nuevo partido...' (New Match...), with the text 'Pulsa para agregar un nuevo contenido' (Press to add new content) and an 'AGREGAR CONTENIDO' (Add Content) button. The background shows a navigation bar with links like PELÍCULAS, CARRERAS DE F1, PARTIDOS DE FÚTBOL, CONTENIDO EN DIRECTO, COMUNIDADES, and PERFIL.

Figura 114: Panel para añadir partidos de fútbol, administrador.

Cuando pulsamos para añadir nuevo contenido nos envía a un panel donde se gestiona toda la creación de nuevos contenidos dentro de la plataforma, tanto contenido en directo, como películas, carreras de f1 y partidos de fútbol, para llenar los datos del nuevo contenido y crearlo. En el caso de que se cree correctamente veremos un mensaje de creación correcta y si se produce algún problema también saldrá un mensaje indicando el motivo del error por lo que ha sucedido.

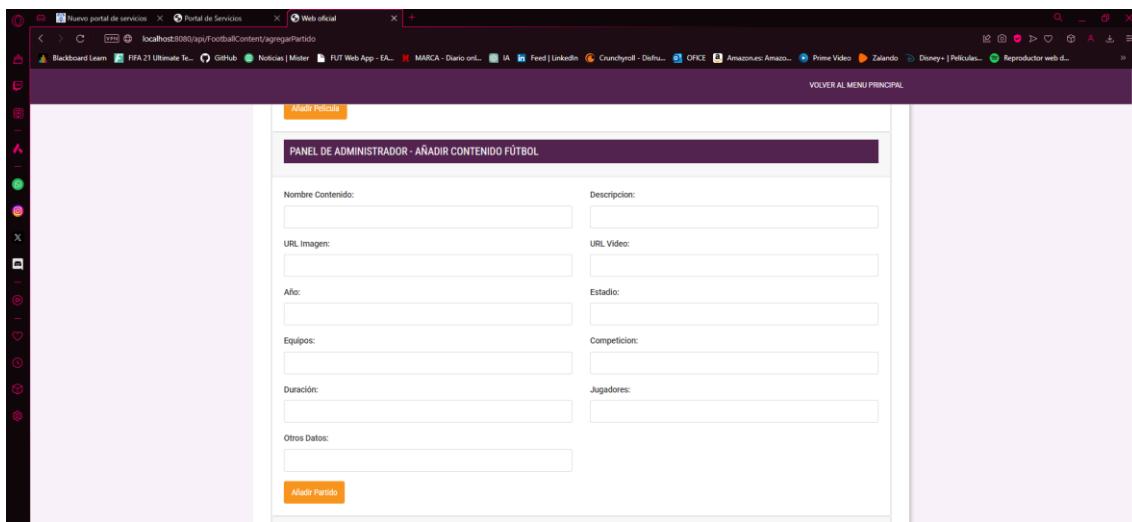


Figura 115: Panel para agregar contenido multimedia, administrador.

Una cosa importante es que las imágenes no están en la base de datos, si no en el propio proyecto en la ruta `src/main/resources/static/images/`, luego cuando vayamos a añadir un contenido en la sección de imagen hay que primero añadir esa imagen a esa ruta, y luego en el campo solo poner el url por ejemplo, `images/Películas/Kong.png`. Dentro del contenido de cada partido de fútbol, debajo del contenido vemos paneles especiales para actualizar el contenido o borrarlo si se desea, en ese caso el admin no vera los comentarios ya que su única función en este caso es gestionar el contenido.

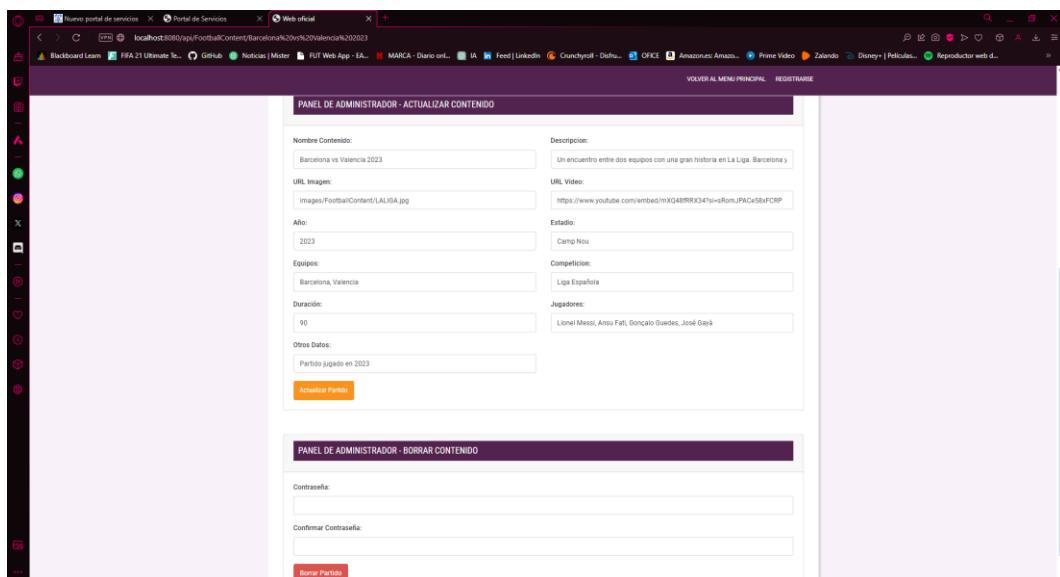


Figura 116: Panel para actualizar y borrar partidos de fútbol, administrador.

La funcionalidad más importante del administrador dentro de la aplicación es la de gestión del contenido en directo ya que el administrador es el encargado de crear el contenido en directo, modificarlo si es necesario, borrarlo, por lo que la gestión completa del contenido en directo en la aplicación se basa en el administrador. Dentro del panel principal de contenido multimedia el usuario tiene un campo especial para añadir un nuevo contenido multimedia.

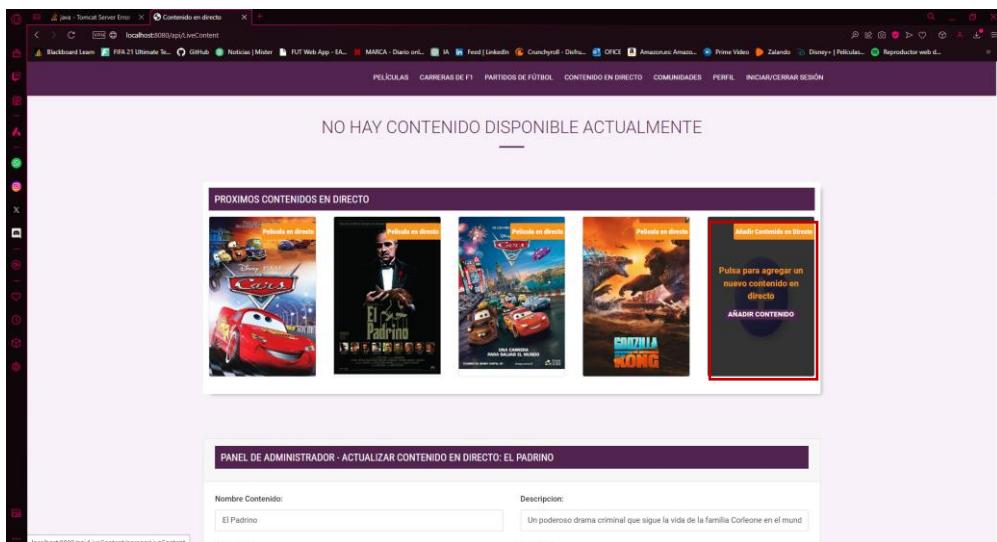


Figura 117: Panel para agregar contenido multimedia, administrador.

Una vez que se acceda al panel especial de creación de contenido en directo que es el mismo panel que el resto de contenidos multimedia en cuanto a la creación se refiere. El administrador debe tener el control de cuando publica el contenido ya que el tramo horario es fundamental para una mayor audiencia, no obstante, la propia plataforma no deja que los contenidos se pisen entre sí, dando error si ocurre el caso.

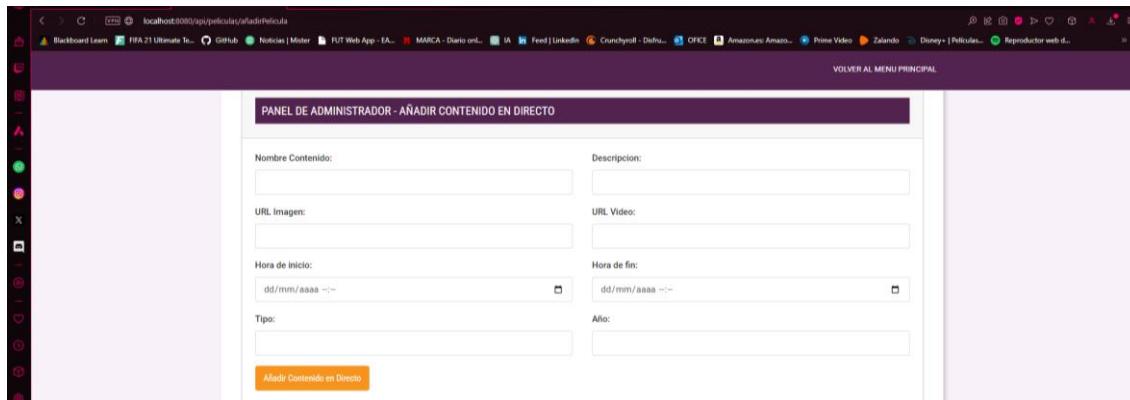


Figura 118: Panel para agregar campos de contenido en directo, administrador.

Otro de los roles importantes del administrador en lo que se refiere a la gestión del contenido en directo es de actualizar datos o borrar contenidos en directo, por lo que si se ha equivocado en la creación de este puede actualizarlo en cualquier momento. La plataforma le proporciona una lista con todos los contenidos en directo creados ya sean futuros o contenidos que ya han sido emitidos y el administrador tiene la función de gestionarlos, siempre pensando en conseguir la mayor visualización posible y siguiendo una estructura correcta de publicación de contenido, ya que este solo está disponible para usuarios con un plan de suscripción pro.

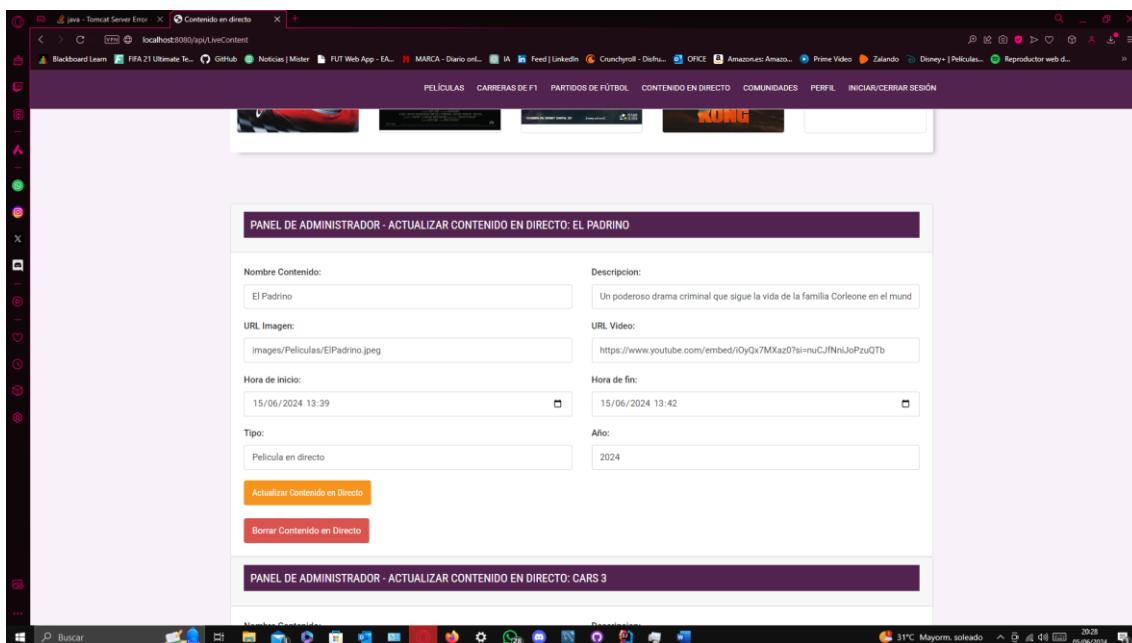


Figura 119: Panel para actualizar y borrar un contenido en directo, administrador.

Aquí vemos un ejemplo de uno de los contenidos que están creados dentro de la plataforma y como el administrador puede editar sus datos o borrarlo si es necesario. Este es el mensaje de error en el caso de que los contenidos se pisen unos con otros.

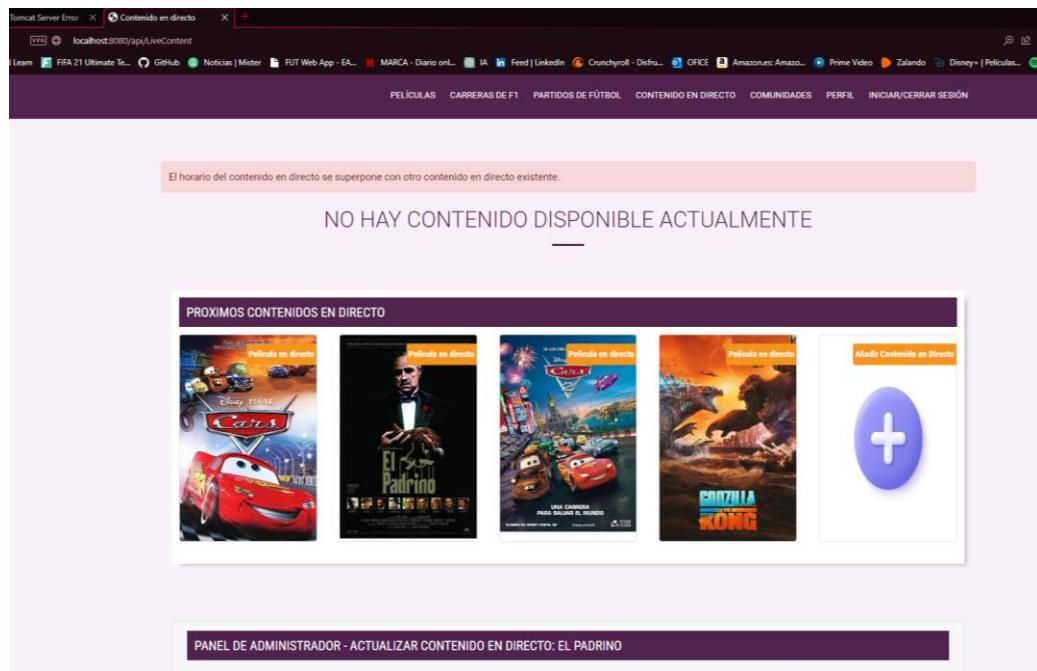


Figura 120: Mensaje de error de tramo horario, administrador.

Y por otro lado este es el mensaje que se enviaría si el contenido se ha actualizado correctamente.

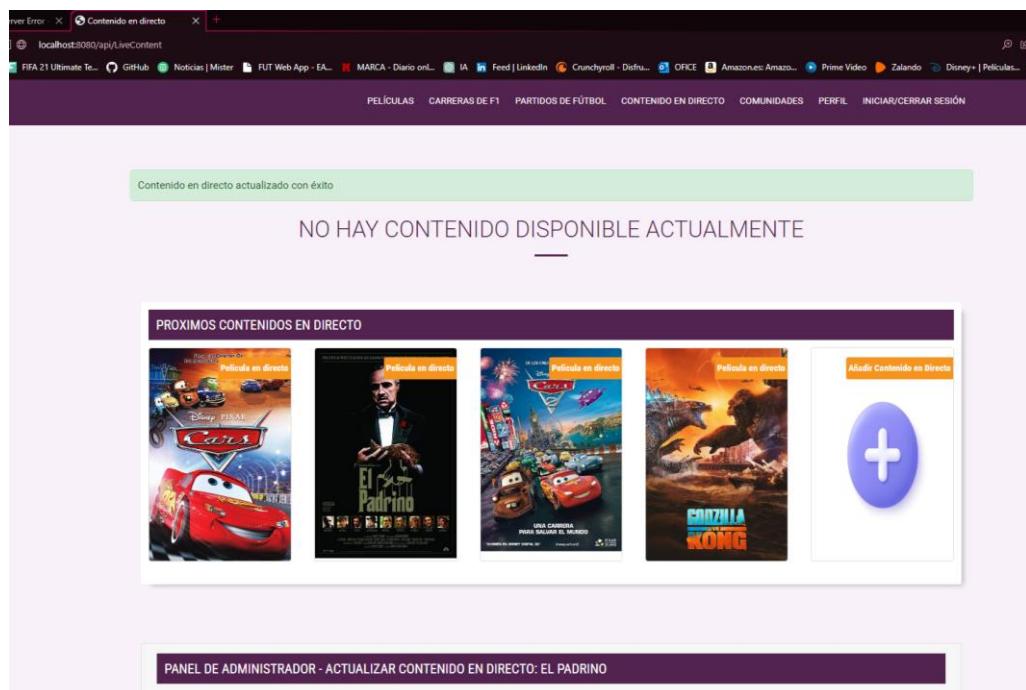


Figura 121: Mensaje de contenido en directo actualizado con éxito, administrador.

Por último, en lo que al panel de administrador se refiere, esta la gestión de las comunidades de chats, en este caso la función del administrador se centra exclusivamente en las comunidades, en la parte de los chats que envían los usuarios no entra, solo crea nuevas comunidades si es necesario en base a los requisitos de los usuarios, o actualizar algún dato de la comunidad como su imagen de perfil, descripción, ya que como he comentado antes no hace falta unirse a las comunidades para participar en ellas, si no que con estar registrado en la aplicación y tener un plan de suscripción básico o superior ya puedes participar en los chats de las comunidades.

El administrador tiene un panel especial para crear nuevas comunidades, que está debajo de los chats de la comunidad así ve en tiempo real si los cambios que él ha hecho se ven bien o se han hecho de forma correcta en las comunidades.

The screenshot shows a form titled 'PANEL DE ADMINISTRADOR - CREAR COMUNIDAD'. It includes fields for 'Nombre de la comunidad:' (Community name:), 'Descripción:' (Description:), 'URL de la imagen:' (Image URL:), and 'Fecha de creación:' (Creation date:). There is also a 'Crear Comunidad' (Create Community) button at the bottom. At the top right of the page are links 'VOLVER AL MENÚ PRINCIPAL' and 'REGISTRARSE'.

Figura 122: Panel para crear una comunidad, administrador.

Una comunidad tampoco tiene muchos datos por lo que su creación no supone grandes dificultades. Voy a hacer el ejemplo de crear una comunidad de ejemplo para ver cómo se haría.

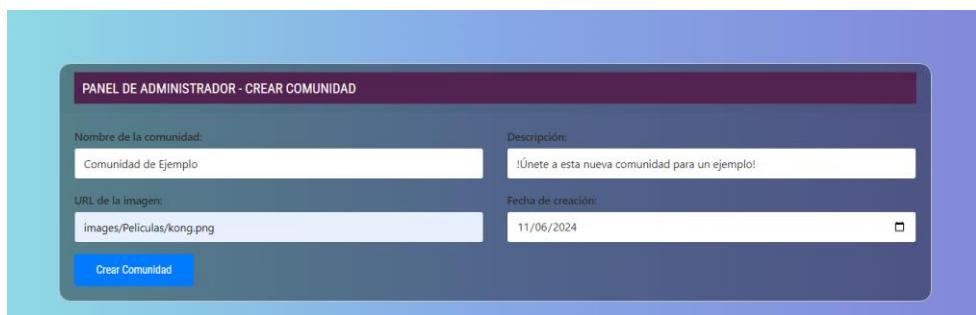


Figura 123: Panel para actualizar los datos de una comunidad, administrador.

Una vez creada dentro del panel de administrador podremos ver el panel que verían los usuarios para ver si la nueva comunidad se ha creado bien y ya abajo en la sección del administrador sale el mensajito si se ha producido éxito o si ha surgido algún tipo de error.

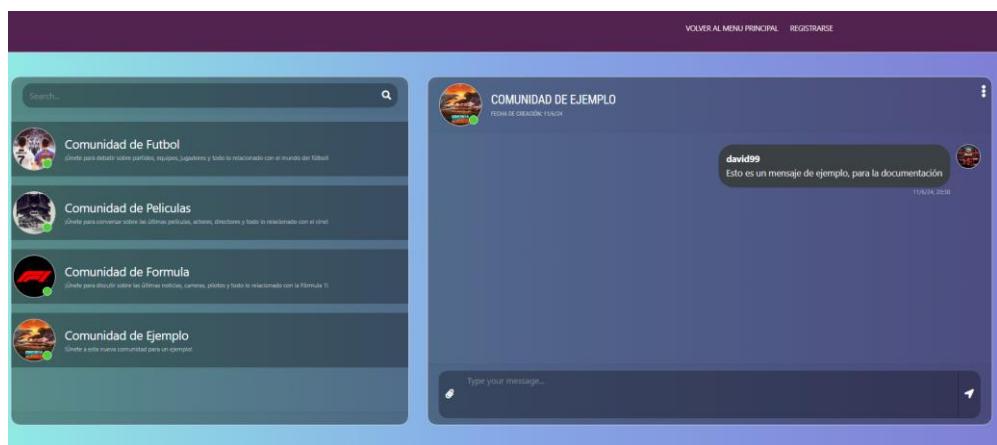
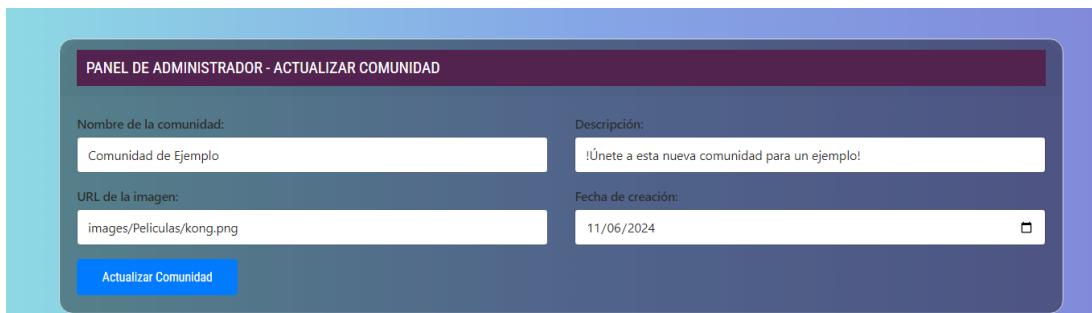


Figura 124: Panel de una comunidad creada de ejemplo.

Vemos como se ha añadido la nueva comunidad y he lanzado un mensaje de ejemplo. Ahora el administrador debajo del panel de las comunidades tiene la sección para actualizarlos y borrarlas luego lo que tiene que hacer primero es seleccionar la comunidad en este caso he seleccionado la comunidad que he creado para el ejemplo.



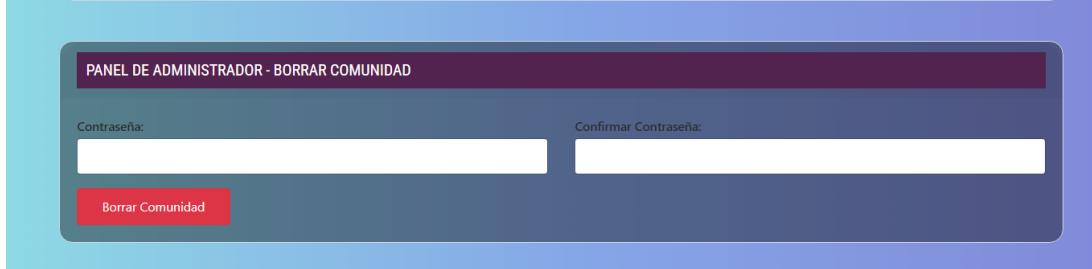


Figura 125: Panel para borrar una comunidad, administrador.

Si la actualizamos se ve el mensaje de éxito o de error para que el administrador tenga constancia en todo momento de lo bien o mal que se están realizando las operaciones.

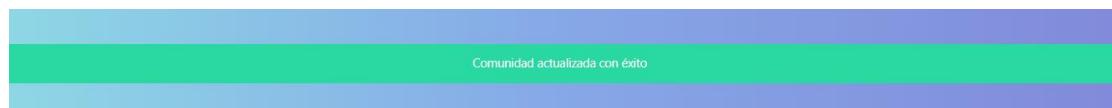


Figura 126: Mensaje de éxito al crear una comunidad, administrador.

Y para borrarla él se añade una capa intermedia de seguridad con la contraseña del administrador, que también maneja mensajes de errores en caso de datos erróneos.



Figura 127: Mensaje de error al borrar una comunidad 1, administrador.

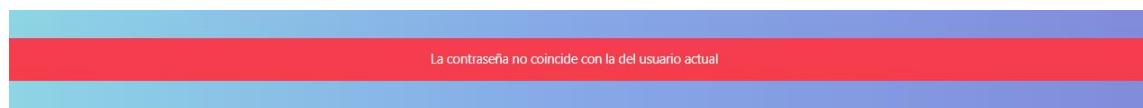


Figura 128: Mensaje de error al borrar una comunidad 2, administrador.



Figura 129: Mensaje de creación de comunidad con éxito, administrador.

11. MANTENIMIENTO Y SOPORTE

Teniendo en cuenta que el proyecto se centra en llegar a un objetivo en cuanto a un desarrollo se refiere para ser evaluado a nivel educativo el mantenimiento y soporte va a ser mínimo, ya que no se trata de un proyecto con fines profesionales a alto nivel. Pero hay una serie de procesos que se van a seguir ya más como personal que como obligación educativa.

11.1 PROCEDIMIENTOS DE MANTENIMIENTO

El mantenimiento de la aplicación se realiza de manera continua para garantizar su correcto funcionamiento antes posibles actualizaciones con las tecnologías utilizadas. Como he comentado antes esto es con fines educativos por lo que voy a poner como sería el mantenimiento que yo haría si esto fuera una plataforma profesional.

- **Actualizaciones regulares:** La aplicación se actualiza regularmente para incorporar nuevas funcionalidades, mejorar las existentes y corregir errores. Las actualizaciones se realizan en un entorno de desarrollo antes de ser desplegadas en el entorno de producción.
- **Monitorización:** La aplicación se monitoriza constantemente para detectar y solucionar problemas de rendimiento y seguridad.
- **Copias de seguridad:** Se realizan copias de seguridad regulares de la base de datos para prevenir la pérdida de datos. Las copias de seguridad se almacenan en un lugar seguro y se pueden utilizar para restaurar la base de datos en caso de fallo del sistema.
- **Pruebas:** Se realizan pruebas unitarias y de integración para garantizar que todas las partes de la aplicación funcionan correctamente. Las pruebas se realizan después de cada actualización para asegurar que no se introducen nuevos errores.

11.2 SOPORTE AL USUARIO

El soporte al usuario incluye:

- **Documentación:** Se proporciona este documento detallado de la aplicación que incluye guías de usuario, preguntas frecuentes, explicación de todas las funciones del código individualizadas, explicación de la arquitectura. La documentación está disponible en línea a través de mi Git Hub [4] <https://github.com/davidbachii/SpringBoot> o a través de la propia universidad.
- **Formación:** La propia documentación está hecha pensando en la formación para los usuarios para ayudarles a entender y utilizar la aplicación de manera eficiente.
- **Comunidad de usuarios:** La comunidad de usuarios donde los usuarios pueden compartir sus experiencias, resolver dudas y aprender unos de otros. Lo que genera foros de discusión, grupos de usuarios.
- **Panel de perfil:** proporcionando la ayuda necesaria para que los usuarios puedan gestionar su perfil de manera efectiva. Lo que permite a los usuarios ver y actualizar su información personal, cambiar su contraseña, gestionar sus métodos de pago y modificar su plan de suscripción. Los usuarios pueden acceder a este panel después de iniciar sesión en la aplicación.

12. PRESUPUESTO DEL DESARROLLO DE LA APLICACIÓN

El desarrollo de esta aplicación requiere de un perfil full stack, que domine en la parte del back bases de datos relacionales en este caso MySql, y Spring Boot, en el front no es necesario que domine frameworks como angular o React, pero si Html, Css, Ajax y Bootstrap. El abanico de trabajo es completo, ya que no solo es desarrollo, también incluye la documentación la fase de testing y el soporte y mantenimiento. Por lo que un perfil junior sería lo más ideal.

12.1 COSTE DEL DESARROLLO

El proyecto tiene una duración estimada de 6-7 meses y teniendo en cuenta que se combina con estudios o trabajo, la dedicación semanal es de 20 horas, ósea media jornada. El precio de la hora va a ser de 15 a 20 euros en base a la experiencia en este proyecto se va a fijar en 20 euros ya que al ser pocas horas al día la dedicación es plena.

Por lo que al mes quedan 88 horas mensuales, teniendo en cuenta que hay días que hay que dedicar más horas o invertir más tiempo en investigación y aprendizaje por lo seis meses que dura el proyecto la duración total 560 horas.

- **Coste de desarrollo:** 620 horas * 20 euros/hora.
- **Total, Costo de Desarrollo:** 12.400 euros.

12.2 HERRAMIENTAS Y SOFTWARE

En el caso del entorno de la base de datos MySql es gratuito, así como git hub para el control de versiones y toda la información es publica en internet, youtube, telegram. Por lo que el único gasto que vamos a tener en lo que a software se refiere es el del entorno de desarrollo, ya que es fundamental tener un buen entorno y en este caso, IntelliJ IDEA es el mejor del mercado.

- **IntelliJ IDEA (versión comercial):** 200 euros/año.
- **Total, Herramientas y Software:** 200 euros.

12.3 INFRAESTRUCTURA

Aunque la aplicación no se ha desplegado en la nube, considerando los costos potenciales para un despliegue inicial.

- **Servidor en la nube (AWS, Azure, Google Cloud) para pruebas y despliegue inicial:** 50 euros/mes * 5 meses.
- **Dominio y SSL:** 20 euros/año.
- **Total, Infraestructura:** 270 euros.

12.3 COSTES INDIRECTOS

El proyecto se puede realizar desde cualquier sitio, solo se necesita conexión a internet para que arranque el servidor y para el control de versiones, luego solo se necesita un equipo con buena Ram, y conexión a internet. Aparte contar con los mínimos gastos de luz.

- **Internet y otros servicios:** 20 euros/mes * 7 meses = 140 euros.
- **Electricidad y servicios:** 20 euros/mes * 7 meses = 140 euros.
- **Total, Costos Indirectos:** 280 euros.

12.4 RESUMEN DE COSTOS

- **Costo de Desarrollo:** 12.400 euros.
- **Herramientas y Software:** 200 euros.
- **Infraestructura:** 270 euros.
- **Costos Indirectos:** 280 euros.
- **Total, Estimado del Presupuesto:** 13.150 euros.

12.5 CONSIDERACIONES ADICIONALES

- **Documentación:** La creación de la documentación es esencial para la mantención y escalabilidad del proyecto, ya que gestiona la creación de documentación técnica, manuales de usuarios, guías de instalación. Supondrá un tiempo estimado de unas 20 a 40 horas, por lo que supondrá de uno a dos meses del proyecto. El precio va incluido en los costes de desarrollo.

- **Testing y QA:** Pruebas unitarias, de integración y pruebas funcionales, también va dentro de los costes de desarrollo.

Este presupuesto se basa en el momento actual del mercado en julio de 2024, todo se trata de dinero en bruto por que lo habría que tener en cuenta impuestos.

13. CONCLUSIONES Y TRABAJOS FUTUROS

El proyecto ya de por si es complejo ya que requiere el manejo de múltiples tecnologías como Java [16] su framework Spring Boot [2], dentro de este dominar Spring Data [2], Spring Security [3], sus dependencias exclusivas en función de las necesidades del proyecto, también hay que dominar las tecnologías frontend (HTML, CSS, JavaScript [1] [15] [11]) y en mi caso Ajax [17] para las búsquedas y operaciones en tiempo real. Cuando te aventuras en un proyecto como este tu solo, como ha sido mi caso, te puedes complicar tanto como quieras, ya que en el mundo web las funcionalidades a implementar son múltiples al igual que las tecnologías que se pueden usar.

Mi objetivo siempre ha sido el de lograr una plataforma completa que cubra todas las principales necesidades de un usuario en cuanto a una plataforma de contenido multimedia se refiere y creo que en ese aspecto si se ha logrado el objetivo, pero como he dicho antes en el mundo web y cuando usas tantas tecnologías y sobre todo cuando trabajas en un proyecto tan grande una persona sola hay que poner límites ya que no se puede hacer todo en apenas 5 meses.

Respecto a los trabajos que se podrían hacer en un futuro, el primero lo tendría claro, pasar el front a Angular y React, ya que este proyecto se basa en (HTML, CSS, JavaScript [1] [15] [11]) con Bootstrap [8] y la complejidad al haber múltiples clases y librerías en cada panel puede suponer problemas a la larga. Respecto a Spring Boot [2] se podría implementar mucha más seguridad con Spring Security [3] [2] sobre todo con el tratamiento de la base de datos, y también el uso de Spring Cloud si se quiere subir el proyecto a la nube para facilitar el manejo y el uso del mismo, pero como comentaba antes cada una de estas tecnologías lleva meses incluso años de aprendizaje y el manejo completo de spring boot [2] requiere de muchos años debido a todo lo que abarca.

Respecto a nuevas funcionalidades que se podrían implementar en la plataforma, es la de crear una zona dentro del perfil de usuario para resolver dudas de los usuarios o un buzón para que los usuarios puedan comentar problemas que les hayan pasado y así los administradores tener conocimiento para pasar al equipo técnico los problemas, luego dentro del contenido de la plataforma en cada sección por ejemplo la de las películas, hacer mejores paneles ya que solo se muestran todas las películas con sus valoraciones y un buscador, podría haber secciones como películas en tendencia, mejores éxitos. Dentro del contenido en directo se podría contemplar la opción de que se emitan varios contenidos en directo a la vez para dar a los usuarios mejores condiciones.

También pienso que esta plataforma se podría hacer realidad y con unas mejoras y un equipo por detrás se podría llegar a publicar e incluso monetizarla, pero ese nunca es el objetivo, lo que a mí me interesa de verdad es que ayude a otros usuarios a entender mejor el uso de Spring Boot [2], que ayude a entender cómo funciona el flujo de comunicación general en el mundo web, la arquitectura y sus funcionalidades individuales por si alguien en el futuro quiere reutilizarlas.

Por lo que se refiere en lo personal, ha sido un proceso complejo ya que yo antes de empezar no tenía conocimientos apenas de javascript, de Ajax [17], y sobre todo de Spring Boot [2] que al final es lo principal y he tenido primero que aprender a utilizarlo y poco a poco ir implementando función a función e ir peleándome mucho con el código día a día, si a alguien le interesa, puede ver en mi git hub [4] el proyecto y los commits para ver la evolución desde 0 del proyecto, que se encuentra en la sección de despliegue de la aplicación. No es un proyecto que recomiendo como Trabajo de fin de grado y sobre todo para hacerlo solo, ya que requiere de muchísimo tiempo no solo de desarrollo, también de aprendizaje y si tienes muchas asignaturas se puede acumular mucha carga de trabajo, yo en mi caso solo tenía una asignatura y trabajaba 5 horas al día en una empresa, por lo que podía invertirle mucho tiempo todas las tardes, el proyecto es un desafío personal pero sobre todo educativo ya que los conocimientos que se van adquiriendo son muy valiosos y más cuando los vas implementando y tienes control completo de cómo funciona cada una de las funcionalidades, la base de datos, que hace cada cosa.

14. BIBLIOGRAFÍA

- [1] 2MuchTech, «Canal de telegram con diseños de interfaces en html, css y js,» 2023. [En línea]. Available: <https://t.me/twomuchtech>.
- [2] J. Hoeller, «Documentación oficial de Spring Boot: Proporciona una guía completa sobre el uso del marco de trabajo Spring Boot, incluyendo la configuración, el desarrollo de aplicaciones y la gestión de dependencias,» 2024. [En línea]. Available: <https://spring.io/projects/spring-boot>.
- [3] B. a. T. L. a. W. R. a. H. G. a. G. J. a. B. J. Alex, «Spring Security Reference,» URL <https://docs.spring.io/springsecurity/site/docs/current/reference/htmlsingle/>. [utoljára megtekintve: 2017. 04. 21.], vol. 12, 2004.
- [4] J. a. C.-A. V. Astigarraga, «!Se puede entender cómo funcionan Git y GitHub!,» *Ecosistemas*, vol. 3, nº 1, pp. 2332--2332, 2022.
- [5] J. A. a. V. Cruz-Alonso, «How to Design a Web Survey Using Spring Boot with Mysql: A Romanian Network Case Study,» *Spiru Haret University, Faculty of Economic Sciences*, vol. 17, nº 2, pp. 63-71, 2017.
- [6] ByteCode, «{Curso Completo de Desarrollo Web con Spring Boot: Este curso de YouTube proporciona una guía paso a paso para desarrollar una aplicación web utilizando Spring Boot,» 2018. [En línea]. Available: <https://www.youtube.com/playlist?list=PLcIHm18h1i4nD4H8tPeID8PNiKsm4VZm5>.
- [7] F. Fuentes, «Curso de Postman, JPA, API REST y MySQL: Este curso de YouTube proporciona una guía detallada sobre el uso de Postman, JPA, API REST y MySQL en el desarrollo de aplicaciones,» 2020. [En línea]. Available: <https://www.youtube.com/playlist?list=PLA7l3GTDnp1YkBwslsdzuJKF55cISdSD>.
- [8] S. S. a. A. P. Gaikwad, «A review paper on bootstrap framework,» *IRE Journals*, vol. 2, nº 10, pp. 349--351, 2019.
- [9] O. a. o. Gierke, «Spring Data JPA-Reference Documentation,» URL <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>. [utoljára megtekintve: 2017. 04. 21.], 2012.
- [10] G. M. González, Aprende a Desarrollar con Spring Framework: 2ª Edición., IT Campus Academy, 2016.
- [11] Harris059, «Coding Notes, Sheets, Tips, and Coding,» 2023. [En línea]. Available: <https://t.me/codehype>.
- [12] R. a. H. J. a. D. K. a. S. C. a. H. R. a. R. T. a. A. A. a. D. D. a. K. D. a. P. M. a. o. Johnson, «The spring framework-reference documentation,» *interface*, vol. 21, p. 27, 2004.
- [13] M. a. C. J. a. G. J. D. Matthews, MySQL and Java developer's guide, John Wiley & Sons, 2003.

- [14] K. S. P. Reddy, Beginning Spring Boot 2: Applications and microservices with the Spring framework, Apress, 2017.
- [15] C. Stella, «Personal Portfolio Zip File & Some Web Development Releted Course in Telegram,» 2023. [En línea]. Available: <https://t.me/codingstella>.
- [16] P. Sznajdleder, Java a fondo: curso de programación, Alpha Editorial, 2016.
- [17] E. E. C. T. a. I. S. Solís., Programació web con CSS, JavaScript, PHP y AJAX, Iván Soria Solís, 2014.
- [18] B. Varanasi, Introducing Maven: A Build Tool for Today's Java Developers, Apress, 2019.