

Patrones Software



Universidad
de Alcalá

Aplicación de Contenido Multimedia

David Bachiller Vela 03221211S

david.bachiller@edu.uah.es

Contenido

Enunciado y requisitos	3
Instrucciones de ejecución y manual de usuario	5
Instrucciones para la instalación y ejecución.	8
Guía detallada de las funcionalidades.	9
Gestión/Administración de la aplicación de Contenido Multimedia.	9
Gestión de Películas.	10
Gestión de Informes.....	11
Funcionalidades para los Clientes.	12
Registro.....	12
Panel principal de la aplicación	13
Comentarios	14
Diseño completo de la aplicación (UML) y patrones utilizados	16
Patrones de Creación	16
PATRÓN SINGLETON.....	16
PATRÓN PROTOTYPE.....	18
Patrones estructurales	22
PATRÓN ADAPTER.....	22
PATRÓN DECORATOR	28
Patrones de comportamiento	35
PATRÓN OBSERVER.....	35
PATRÓN STRATEGY	43
Aplicación	48
Base de datos	48
Esquema de información y modelo E/R propuesto.	48
Programación en JAVA con el patrón MVC.....	50
Uso de HTML5, CSS3 y JavaScript.	66

Enunciado y requisitos

1. Introducción:

La aplicación "Bachi Cines" se ha desarrollado con el objetivo de proporcionar a los usuarios una plataforma integral para explorar, gestionar y compartir información sobre películas y otros contenidos multimedia.

2. Objetivos:

Facilitar a los usuarios la búsqueda y exploración de películas, con información detallada y comentarios de la comunidad.

Integrar patrones de diseño para mejorar la estructura y funcionalidad del sistema.

Proporcionar herramientas de gestión de contenido, como la clasificación por edad y estadísticas de visualización.

3. Descripción de la Aplicación:

Bachi Cines es una plataforma de contenido multimedia diseñada para ofrecer a los usuarios una experiencia de entretenimiento multimedia. Inspirada en modelos de servicios de streaming como Netflix y HBO, Amazon Prime Video, proporcionará acceso a una amplia variedad de películas. La aplicación permitirá a los usuarios explorar y calificar visualizar contenido.

4. Características Principales:

Catálogo Multimedia: Bachi Cines contará con un extenso catálogo de películas, clasificadas por género, año de lanzamiento, y puntuación.

Perfiles de Usuario: Los usuarios una vez registrados en la aplicación podrán ver todos los contenidos de esta misma, y comentar en cada película.

Reproducción Local: Ya que se trata de una aplicación local, los usuarios podrán ver una imagen representativa y una breve descripción de las películas y series en el catálogo, en vez del contenido en video.

Gestión de Contenido: La aplicación permitirá a los administradores agregar, eliminar y modificar contenido en el catálogo.

5. Requisitos Funcionales:

Autenticación: Los usuarios deberán iniciar sesión o registrarse para acceder a las funcionalidades de la aplicación.

Exploración de Contenido: La aplicación ofrecerá una interfaz intuitiva para que los usuarios exploren y encuentren películas y series.

Calificación y Comentarios: Los usuarios podrán calificar el contenido y dejar comentarios.

6. Requisitos Técnicos:

Desarrollo en Java: La aplicación en Java, aprovechando las capacidades del lenguaje y su versatilidad y será desarrollada en el entorno **Apache NetBeans IDE 20**, ya que vamos a usar servlets y jsp, además del servidor **Glassfish** y la base de datos que vamos a usar va a ser la propia del entorno **Apache Derby**.

Persistencia de Datos: Utilización de mecanismos de persistencia de datos para almacenar información sobre usuarios, contenido.

Interfaces Gráficas: Diseño de interfaces gráficas intuitivas utilizando herramientas como jsp, bootstrap para la interacción del usuario.

Manejo de Imágenes: Implementación de la visualización de imágenes y videos de películas, las cuales se almacenarán en la base de datos.

7. Innovaciones:

Implementación de Patrones de Diseño: Introducción de patrones de diseño como Singleton, Prototype, Observer, Decorator, Adapter, Strategy, y Observer para mejorar la estructura y flexibilidad del código.

Aparte ya que estamos usando servlets, jsp y bases de datos vamos a utilizar la modelo vista-controlador, donde la vista será la interfaz de usuario, y desde ahí se lanzaran peticiones al controlador y este devolverá los datos solicitados, también permitirá la gestión de sesiones lo cual se usara en todo el desarrollo de la aplicación para saber que usuario está utilizando la aplicación en cada momento o si es el administrador.

Experiencia del Usuario: Enfoque en una interfaz de usuario atractiva y fácil de usar para una experiencia de usuario fluida y satisfactoria.

Seguridad: Mediante la gestión de sesiones vamos a controlar en cada momento la sesión del usuario, por si copias el url del navegador con una sesión iniciada, sobre todo una vez que estas dentro del panel del administrador, controlaremos eso para evitar una vez que este el servidor arrancado.

Instrucciones de ejecución y manual de usuario

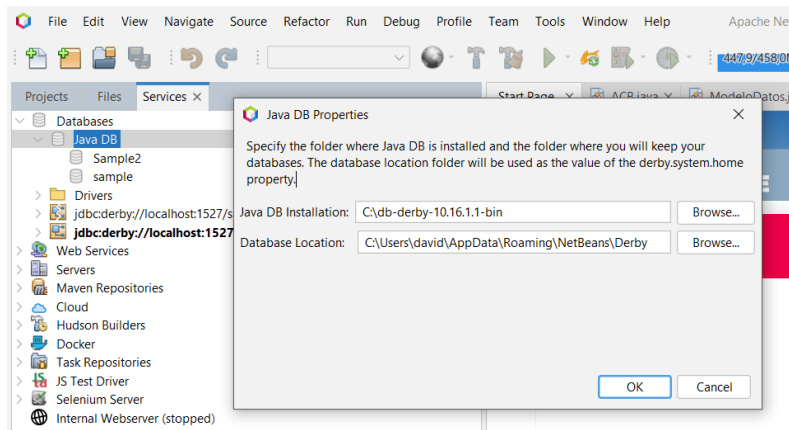
Lo primero de dato es saber como instalar la base de datos **Aache Derby** dentro de **Apache Neatbeans**:

Primero necesitamos descargar Derby desde la web:

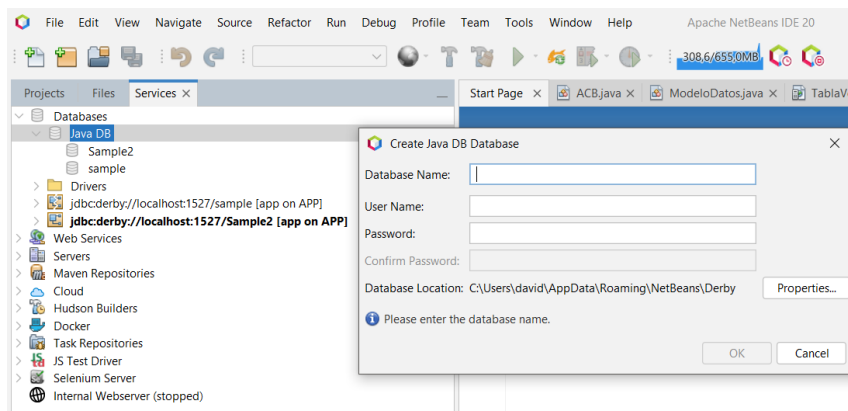
https://db.apache.org/derby/derby_downloads.html

Una vez aquí descargamos aquella compatible para nuestra versión de JDK.

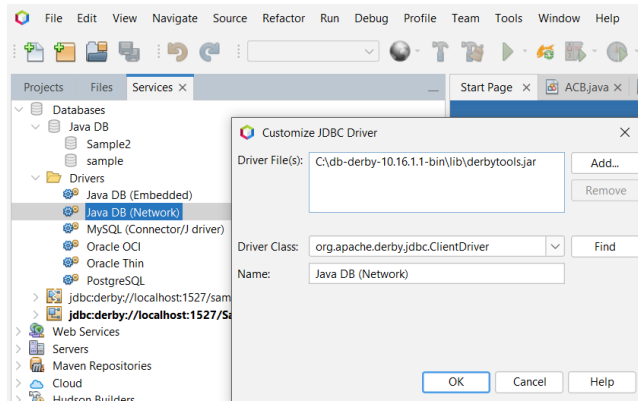
PASO 1) DENTRO DE SERVICES EN SELECCIONAMOS LA BASE DE DATOS Y EN PROPIEDADES INCLUIOS EL ARCHIVO DE DERBY INSTALADO EN ESTE CASO LA VERSION 16:



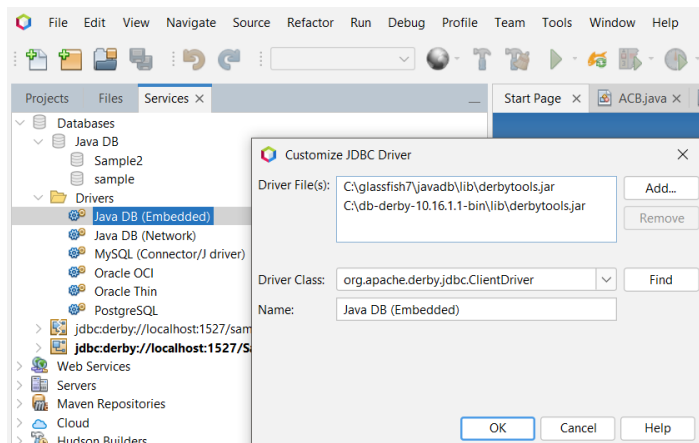
PASO 2) LE DAMOS A CREAR LA BASE DE DATOS, LE PONEMOS UN NOMBRE UN USUARIO Y UNA CONTRASEÑA, EN NUESTRO CASO HEMOS CREADO UNA QUE SE LLAMA SAMPLE2, Y EL USUARIO Y CONTRASEÑA ES admin admin:



PASO 3) DENTRO DE LOS DRIVERS EN JAVA DB (NETWORK) LE DAMOS A CUSTOMIZE Y INCLUIAMOS EL JAR DE DERBY LLAMADO DERBYTOOLS, SE ENCUENTRA EN LA CARPETA INSTALADA DENTRO DE LIB:

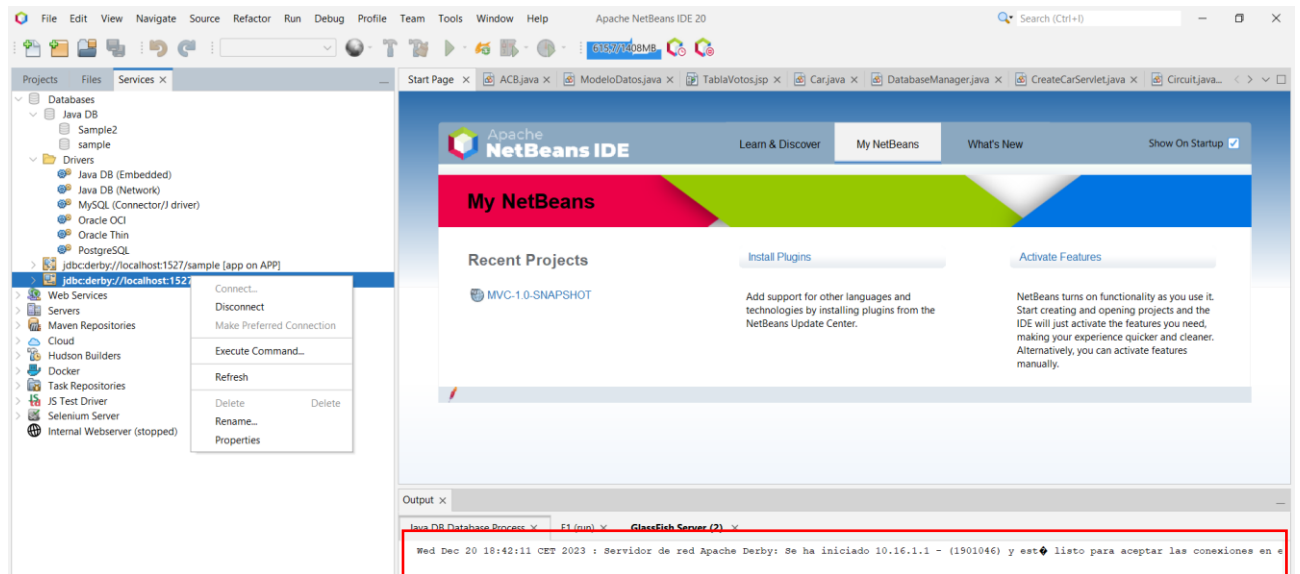


PASO 4) EN JAVA DB (EMBEDDED) AÑADIMOS EL DERBYTOOLS DE DERBY Y EL DE GLASFISH

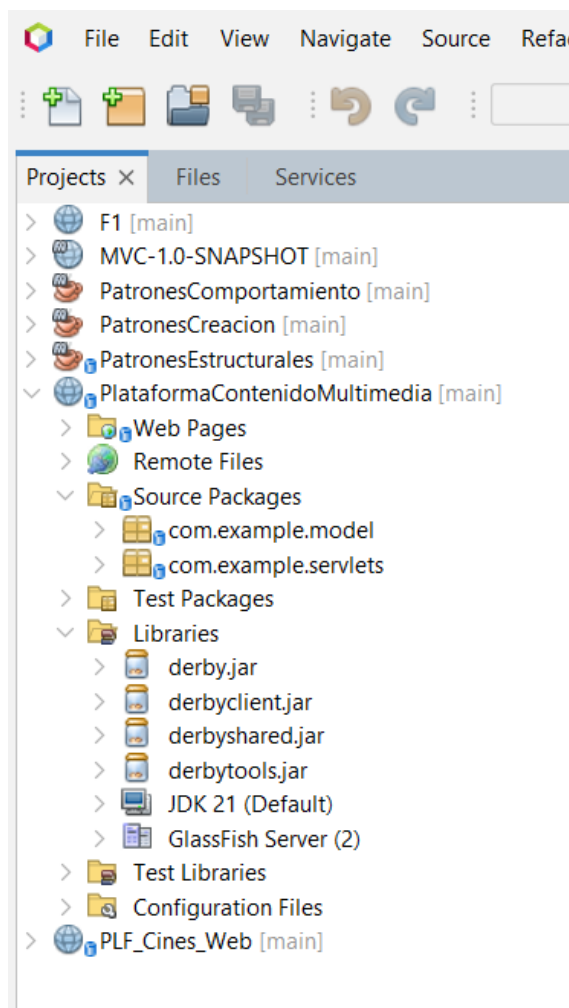


EN EL RESTO DE LOS DRIVERS NO HAY QUE TOCAR NADA SE DEJA EL VALOR POR DEFECTO

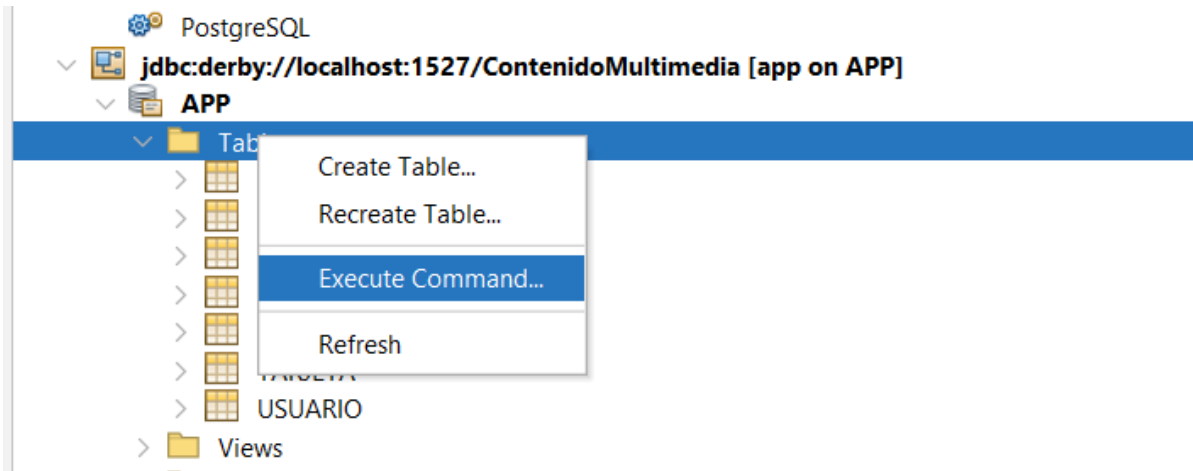
PASO 5) UNA VEZ TODO AÑADIDO CORRECTAMENTE LE DAMOS A CONECTAR LA BASE DE DATOS Y NOS DEBERIA SALIR ABAJO UN MENSAJE DE CONFIRMACION DE SERVICIO:



AHORA DENTRO DEL PROYECTO QUE ES DE TIPO JAVA WITH ANT- JAVA WEB- WEB APPLICATION, EN LAS LIBRERIAS INCLUIAMOS ESTOS .JAR:



Una vez que este todo creado correctamente, dentro de nuestra base de datos insertamos el código SQL para la creación de las tablas y meter contenido a las mismas, el código SQL está en dentro de la carpeta del proyecto, con el nombre **multicines.sql**:



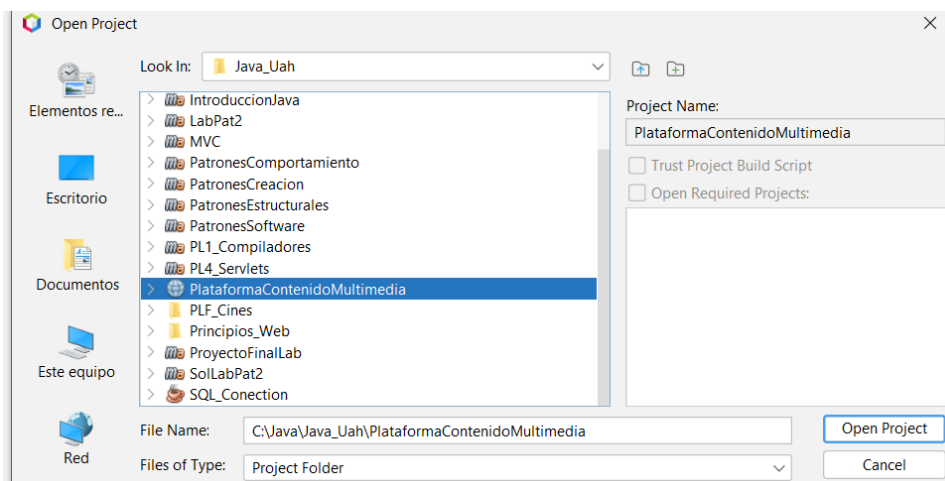
Ahora pasamos a ver como se instala y se configura el servidor que vamos a utilizar en este caso **GlassFish Server**:

<https://youtu.be/zx6QWs2T6w4?si=G29ITCFUJ-UOrs-B>

Ahora si pasamos con ejecución de la aplicación para su correcto funcionamiento, con un manual de usuario explicativo.

Instrucciones para la instalación y ejecución.

Una vez descomprimido el zip de la descarga, el archivo que nos interesa se llama PlataformaContenidoMultimedia, por lo que entramos en NetBeans que es el entorno en el que se ha llevado a cabo la práctica y seleccionamos la opción de **open Project**.



Seguidamente en nuestra base de datos en este caso apache Derby insertamos el código SQL que viene en la carpeta Base de datos, para rellenar las tablas y poder comenzar a trabajar, hay que tener en cuenta que es un proyecto web por lo que cada vez que hacemos

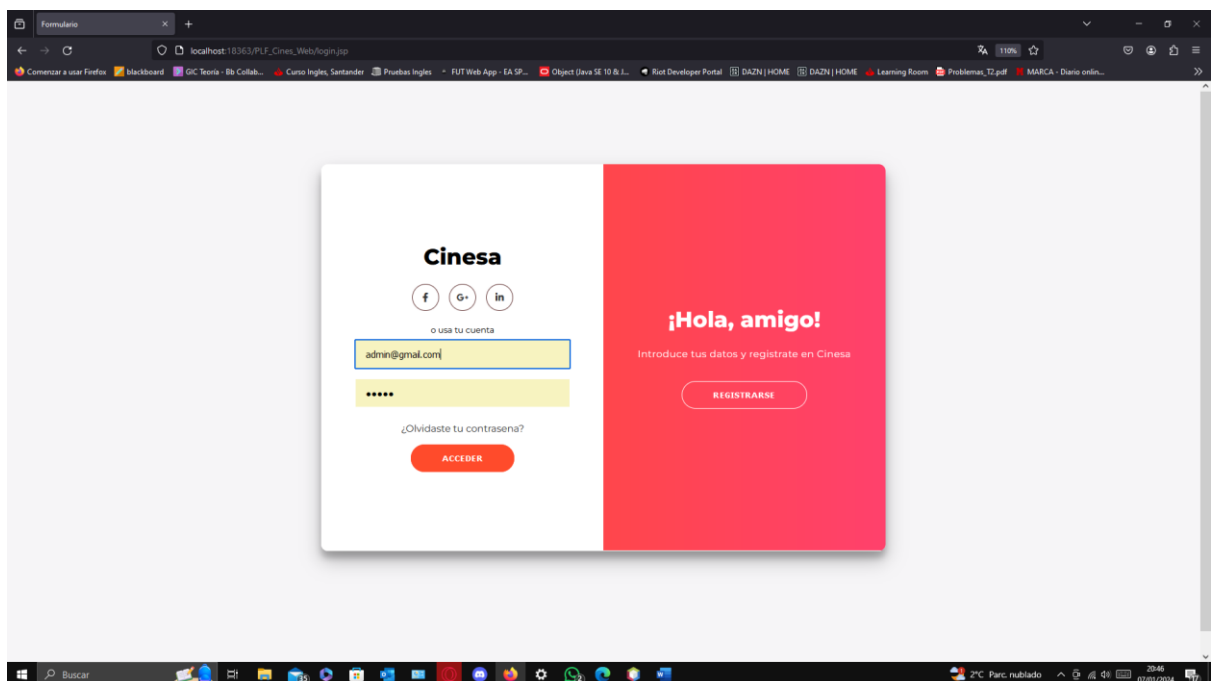
un cambio se auto compila, le damos a la opción de **clean and build** y si todo está bien pasamos a la ejecución del programa.

Guía detallada de las funcionalidades.

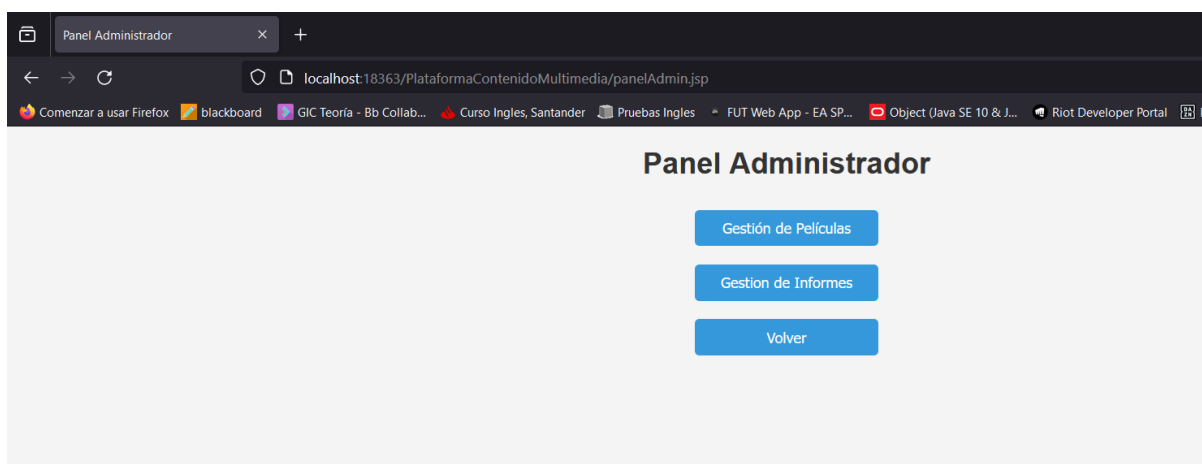
Gestión/Administración de la aplicación de Contenido Multimedia.

El administrador es una especie de super usuario creado en el sistema, pero que no está dentro de la base de datos para evitar si nos hackean o consiguen acceso a la base de datos, que no nos puedan controlar nuestro panel de administración.

Pasamos a ver la parte visual, para poder acceder a la administración tenemos que iniciar sesión como administradores, por lo que nos dirigimos a la opción de registrarse y escribimos dentro del panel del login admin@gmail.com admin.



Y accedemos:

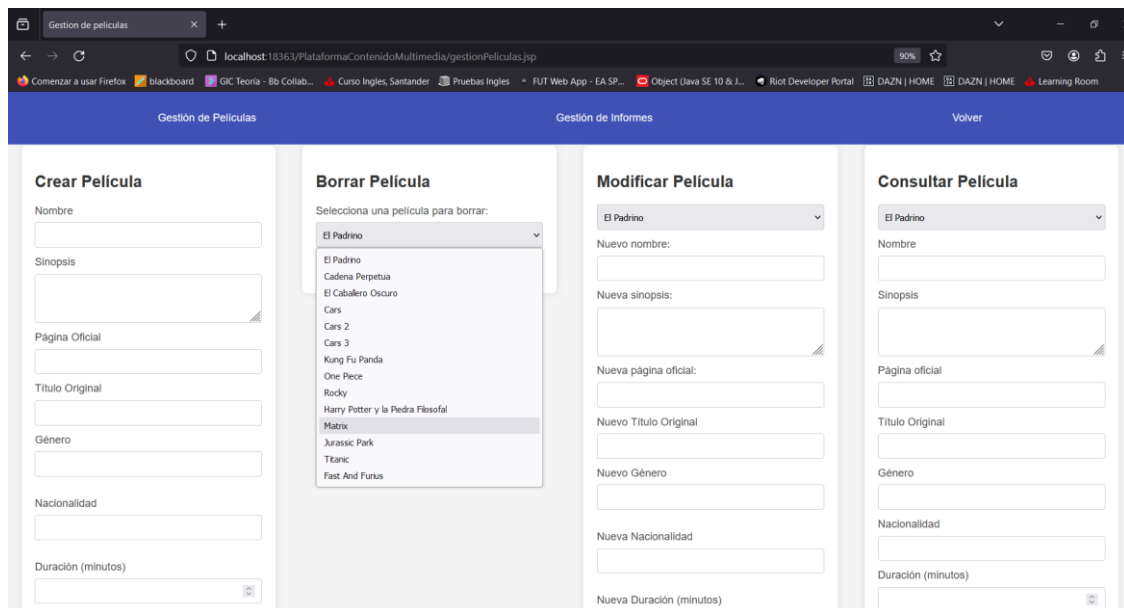


Ahora el administrador decide qué operación realizar.

Gestión de Películas.

Todo el panel del administrador se ha hecho llevando a cabo el patrón de diseño MVC, la implementación y el código interno explicado está en la sección de implementación, pasamos a ver el panel de gestión de películas:

Tenemos 4 posibles operaciones, insertar película, borrar película, modificar película y consultar película:



Información para el usuario: En el campo Url imagen se ha de poner `imagenes/"NombreImagen".png`, esto es debido a que en el proyecto tenemos una carpeta llamada `imagenes` donde guardamos todas las portadas de las películas.

Crear Películas:

Hay un chequeo creado con javascript en el caso de que se creen datos que no existen

Borrar Película:

Sale una lista desplegable con todas las películas, el administrador selecciona la que desee y la borra.

Modificar película:

Se selecciona la película y se cambia los campos que se desee.

Consultar película:

Se selecciona de la lista desplegable la película y se muestran los datos en los campos.

Consultar Película

El Padrino ▼

Nombre

El Caballero Oscuro

Sinopsis

Batman se enfrenta a su némesis, el Joker, en un juego de ingenio que amenaza Gotham City.

Página oficial

<https://example.com/darkknight>

Título Original

The Dark Knight

Género

Acción, Crimen, Drama

Nacionalidad

Estados Unidos

Duración (minutos)

152

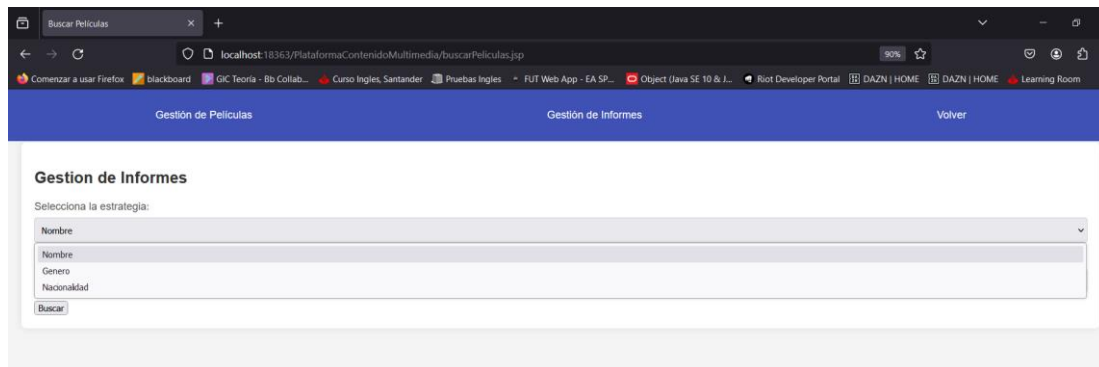
Año

2008

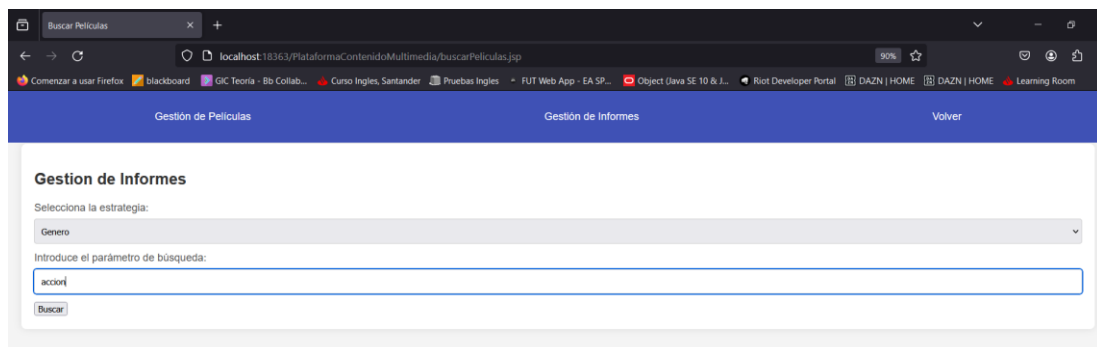
Distribuidora

Gestión de Informes.

Este panel está asociado al patron strategy y podremos hacer consultas de información seleccionando una estrategia u otra.



Una vez seleccionado la estrategia establecemos el parámetro que queremos consultar:



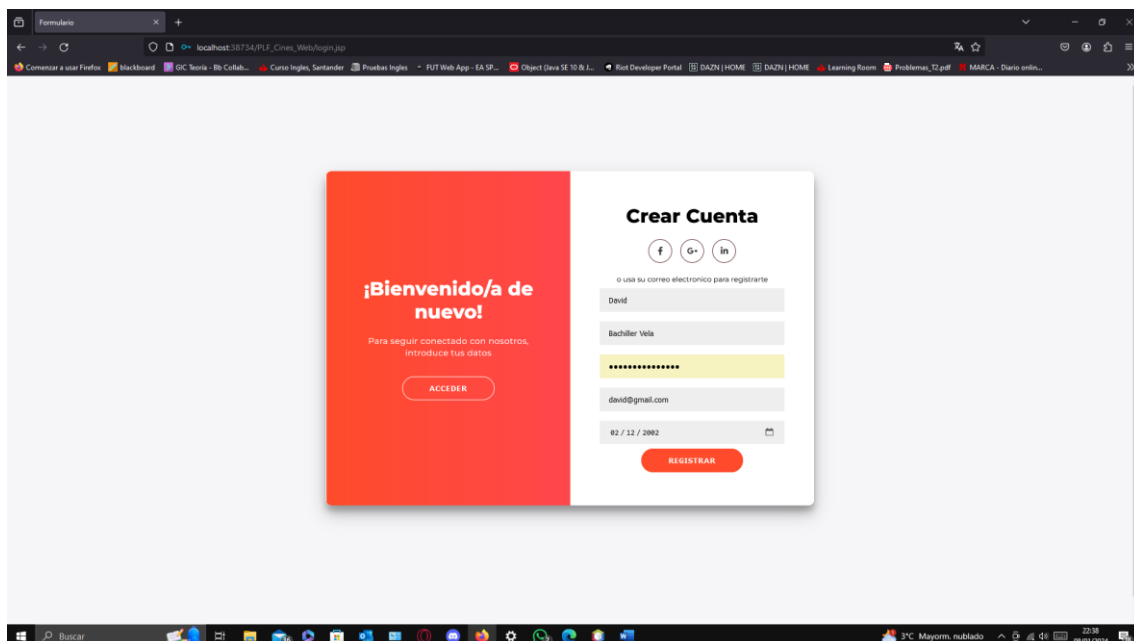
Simplemente veremos de las películas que coincidan con esos parámetros un **toString()**.



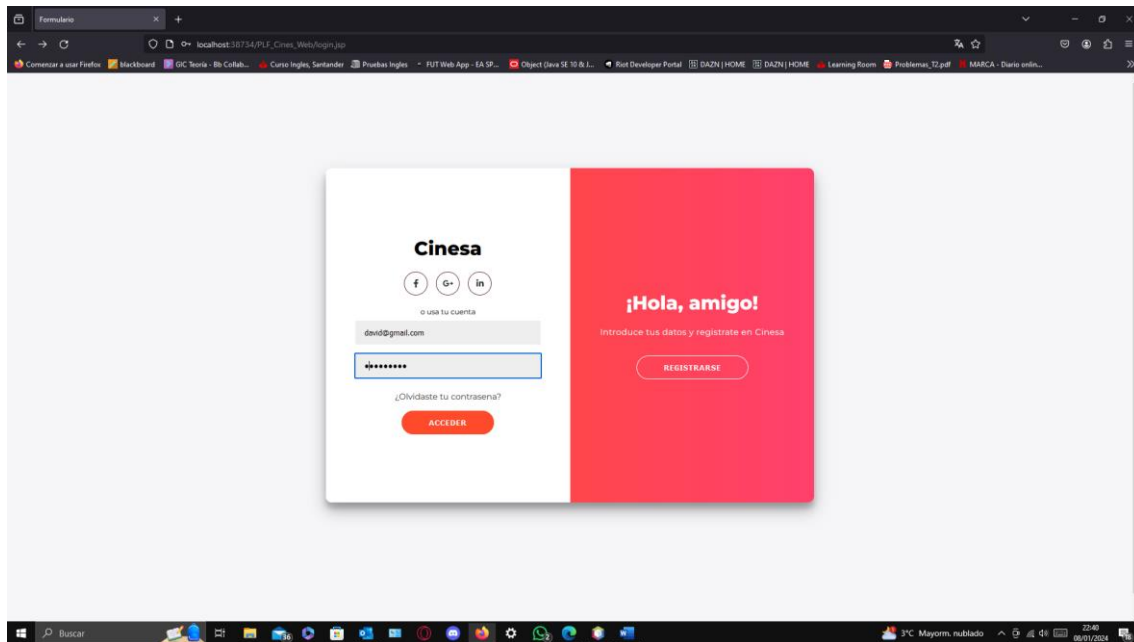
Funcionalidades para los Clientes.

Registro.

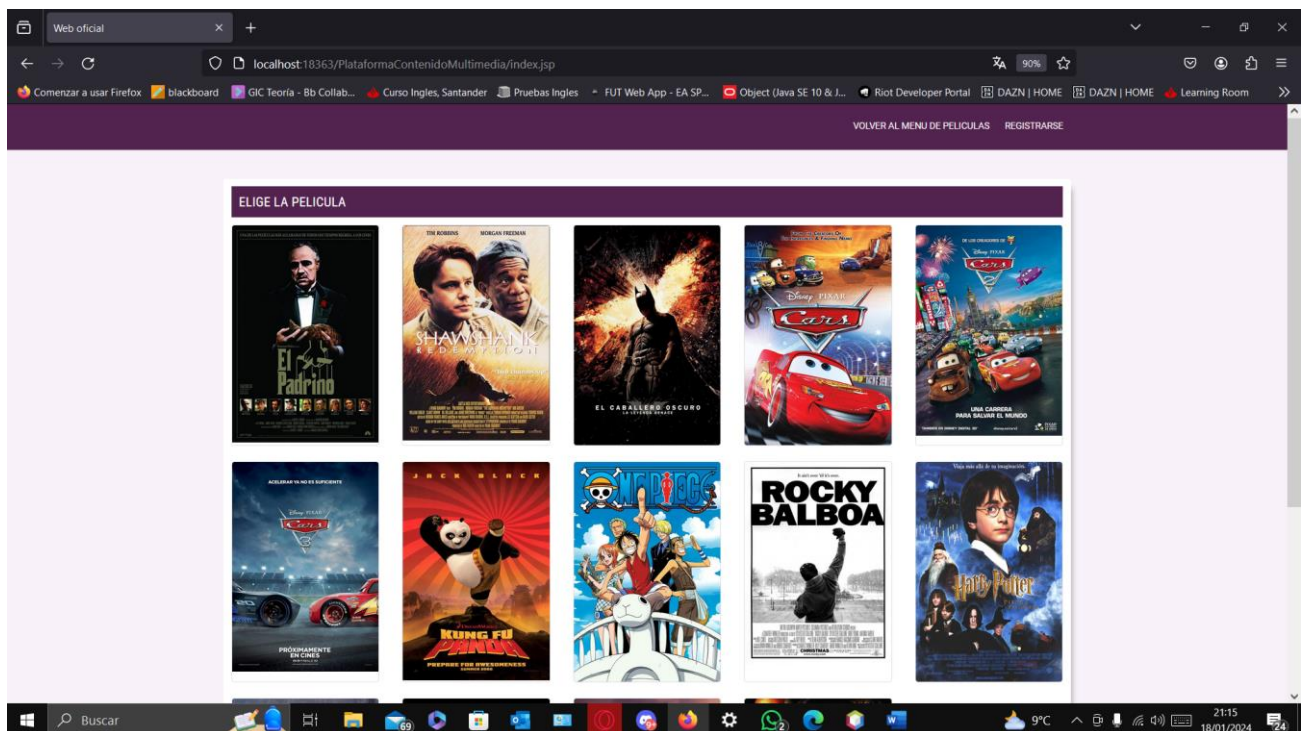
Dentro del registro tenemos la posibilidad de registrarnos o de iniciar sesión si ya estamos registrados, simplemente rellenamos los datos, para darnos de alta en el sistema.



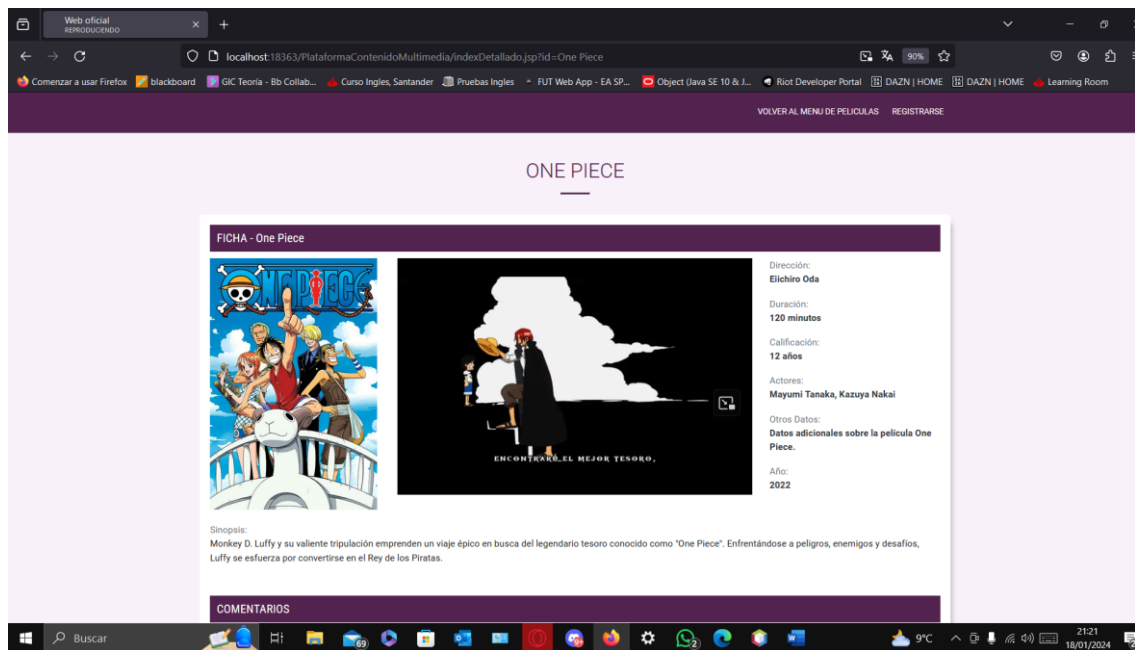
El panel de login nos permite iniciar sesión y si todo se ha creado correctamente mediante gestión de sesiones accedemos al sistema, y ya podremos hacer comentarios y hacer una reserva.



Panel principal de la aplicación



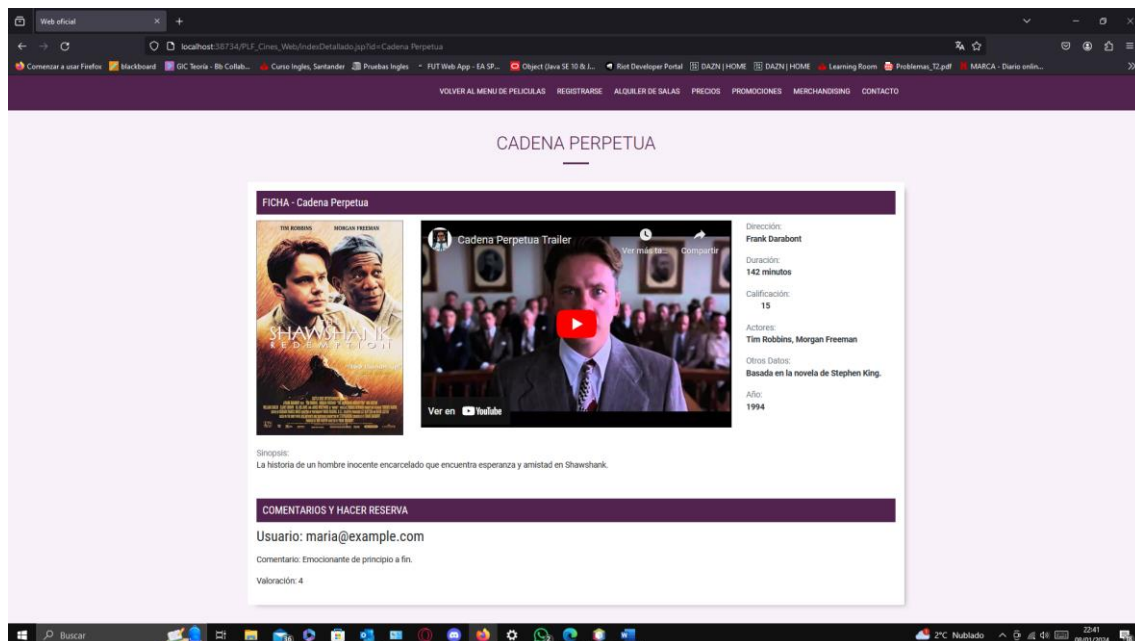
Tendremos la opción de registrarnos para poder hacer comentarios de las películas que mas nos gusten y compartir nuestras opiniones con la comunidad.



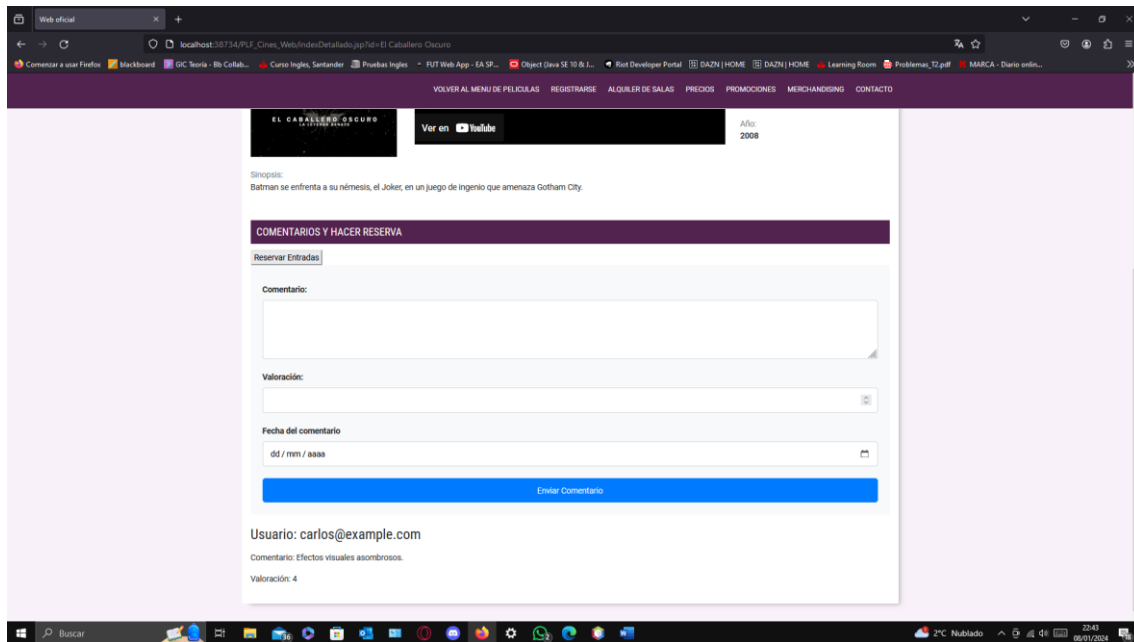
Esto es lo que se ve dentro de cada película, en vez de poner un video de la película, pongo el tráiler para que se pueda ver contenido.

Comentarios

Si accedemos a la aplicación sin estar registrados podremos ver los comentarios de las películas, pero no podremos hacer comentarios y se verá tal que así:



Solo veremos los comentarios de la gente, ahora para poder hacer comentarios nos registramos:



Web oficial

localhost:38734/PLF_Cines/Web/indexDetalle.jsp?id=El Caballero Oscuro

VOLVER AL MENÚ DE PELÍCULAS REGISTRARSE ALQUILER DE SALAS PRECIOS PROMOCIONES MERCHANDISING CONTACTO

EL CABALLERO OSCURO

Ver en YouTube

Año: 2008

Sinopsis:
Batman se enfrenta a su némesis, el Joker, en un juego de ingenio que amenaza Gotham City.

COMENTARIOS Y HACER RESERVA

Reservar Entradas

Comentario:

Valoración:

Fecha del comentario
dd / mm / aaaa

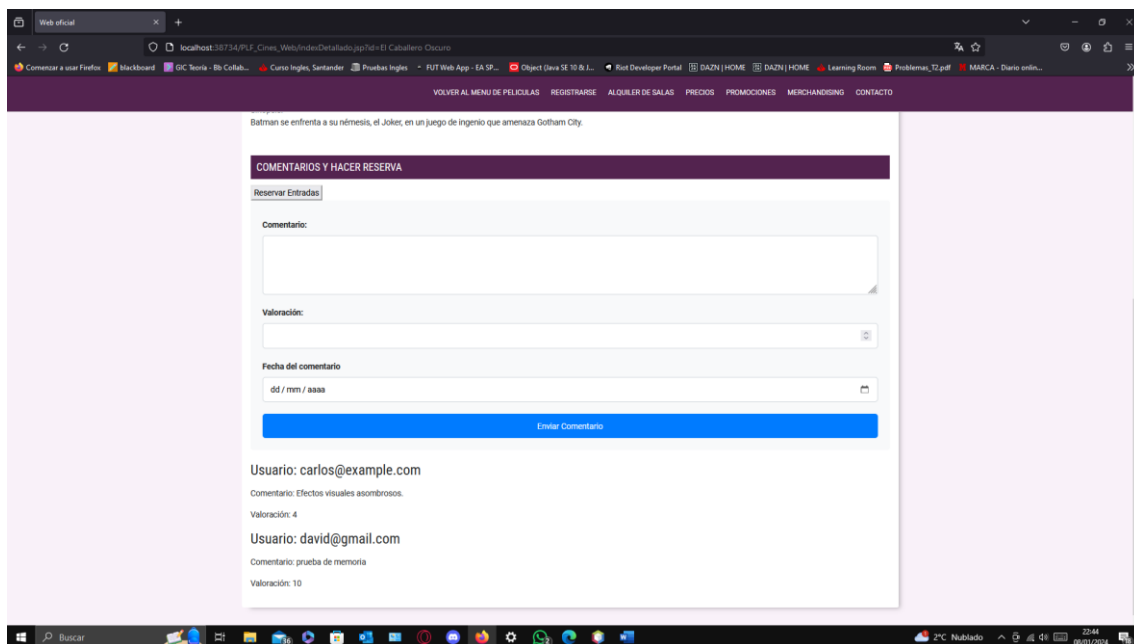
Enviar Comentario

Usuario: carlos@example.com

Comentario: Efectos visuales asombrosos.

Valoración: 4

Ahora si podemos hacer un comentario, vamos a hacer uno de prueba para el caballero oscuro:



Web oficial

localhost:38734/PLF_Cines/Web/indexDetalle.jsp?id=El Caballero Oscuro

VOLVER AL MENÚ DE PELÍCULAS REGISTRARSE ALQUILER DE SALAS PRECIOS PROMOCIONES MERCHANDISING CONTACTO

EL CABALLERO OSCURO

Ver en YouTube

Año: 2008

Sinopsis:
Batman se enfrenta a su némesis, el Joker, en un juego de ingenio que amenaza Gotham City.

COMENTARIOS Y HACER RESERVA

Reservar Entradas

Comentario:

Valoración:

Fecha del comentario
dd / mm / aaaa

Enviar Comentario

Usuario: carlos@example.com

Comentario: Efectos visuales asombrosos.

Valoración: 4

Usuario: david@gmail.com

Comentario: prueba de memoria

Valoración: 10

Y si consultamos la base de datos vemos también que se ha almacenado mediante gestión de sesiones el correo del usuario y el nombre de la película.

SELECT * FROM APP.COMENTA... X					
Max. rows: 100		Fetched Rows: 7			
#	TEXTO	VALORACION	FECHACOMENTARIO	EMAIL_USUARIO	NOMBREPELICULA_PELICULA
1	Una obra maestra, actuaciones increíbles.	5	2023-01-05	rafael@example.com	El Padrino
2	Emocionante de principio a fin.	4	2023-01-06	maria@example.com	Cadena Perpetua
3	Efectos visuales asombrosos.	4	2023-01-07	carlos@example.com	El Caballero Oscuro
4	Larga pero intensa perfecta para una noche ...	10	2024-01-05	david@gmail.com	El Padrino
5	Siempre sera mi pelicula favorita de todos lo...	10	2024-01-05	david@gmail.com	Cars
6	Super divertida y entretenida de ver	10	2024-01-07	david@gmail.com	One Piece
7	prueba de memoria	10	2024-01-08	david@gmail.com	El Caballero Oscuro

Diseño completo de la aplicación (UML) y patrones utilizados

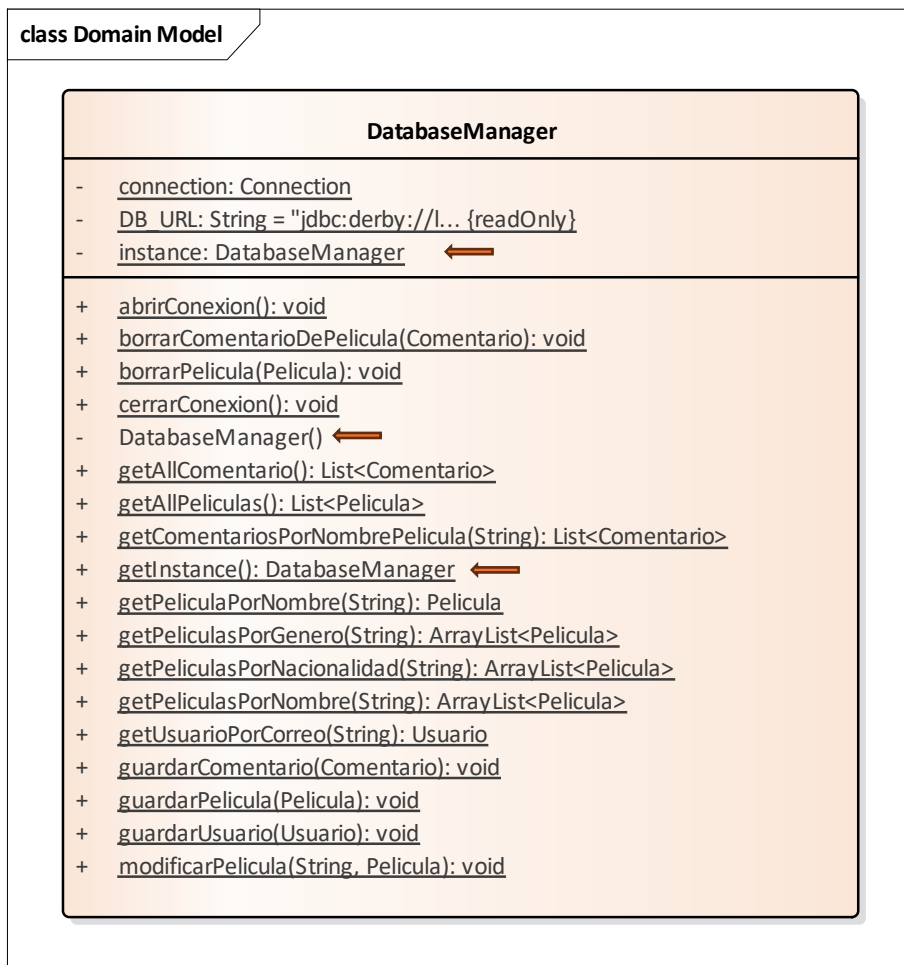
Patrones de Creación

PATRÓN SINGLETON

Objetivo: Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

Se utiliza cuando: Sólo puede haber una instancia de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido. La única instancia debería ser extensible por herencia y los clientes deberían poder usar una instancia extendida sin modificar su código.

En este caso tenemos una clase llamada **DataBaseManager** desde la que vamos a hacer todas las gestiones de la base de datos, desde almacenar usuarios, películas, comentarios hasta obtener datos de estos mismos según nos interese en la aplicación.




```
private static DatabaseManager instance;

private DatabaseManager() {
    // Constructor privado para evitar instancias múltiples
}

public static synchronized DatabaseManager getInstance() {
    if (instance == null) {
        instance = new DatabaseManager();
    }
    return instance;
}
```

Como funciona:

Variable de instancia: `private static DatabaseManager instance;` es una variable de clase (estática) que mantiene la única instancia de `DatabaseManager`. Es privada para que ninguna otra clase pueda acceder a ella directamente.

Constructor privado: `private DatabaseManager() {...}` es un constructor privado. Al hacer privado el constructor, otras clases no pueden crear nuevas instancias de `DatabaseManager` utilizando el operador `new`.

Método `getInstance`: `public static synchronized DatabaseManager getInstance() {...}` es un método de clase (estático) que devuelve la única instancia de `DatabaseManager`. Si la instancia no existe (es decir, es `null`), crea una nueva. Si ya existe, simplemente la devuelve. El método es `synchronized` para garantizar que solo un hilo pueda ejecutarlo a la vez, lo que es importante en entornos multihilo para evitar la creación de múltiples instancias.

En resumen, el patrón Singleton garantiza que solo exista una instancia de `DatabaseManager` en toda la aplicación, y proporciona un punto de acceso global a esa instancia. Esto es útil en tu caso porque solo quieres una conexión a la base de datos, y quieres que todas las partes de tu aplicación utilicen esa misma conexión.

Usos:

```
// Guarda la película modificada en la base de datos
DatabaseManager.getInstance().modificarPelícula(nombrePelículaAModificar, película);
```

```
// Obtén la película por su nombre
Película película =
DatabaseManager.getInstance().getPelículaPorNombre(nombrePelículaAModificar);
```

```
// Guardar el usuario en la base de datos  
DatabaseManager.getInstance().guardarUsuario(user);
```

PATRÓN PROTOTYPE

Objetivo: Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.

Este patrón se usa en los casos en los que crear una instancia de una clase sea un proceso muy complejo y requiera mucho tiempo. Lo que hace es crear una instancia original, y, cuando necesitemos otra, en lugar de volver a crearla, copiar esa original y modificarla.

Se utiliza cuando: Crear una instancia de una clase sea muy complejo y tarda mucho, lo que hace es crear una instancia original y cuando necesitemos otra, copia esa original y la modifica.

Un sistema debe ser independiente de cómo se crean, se componen y se representan sus productos.

Que las clases a instanciar sean especificadas en tiempo de ejecución, para evitar construir una jerarquía de clases de fábrica paralela a la jerarquía de clases de los productos.

Las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes. Puede ser más adecuado tener un número equivalente de prototipos y clonarlos, en vez de crear manualmente instancias de cada clase cada vez con el estado apropiado.

Uso en la aplicación: He aplicado el patrón Prototype en la gestión de comentarios de tu aplicación multimedia. Al clonar un comentario existente, estamos creando una nueva instancia del comentario con los mismos valores iniciales. Esto es útil para mantener la consistencia y el formato de los comentarios.

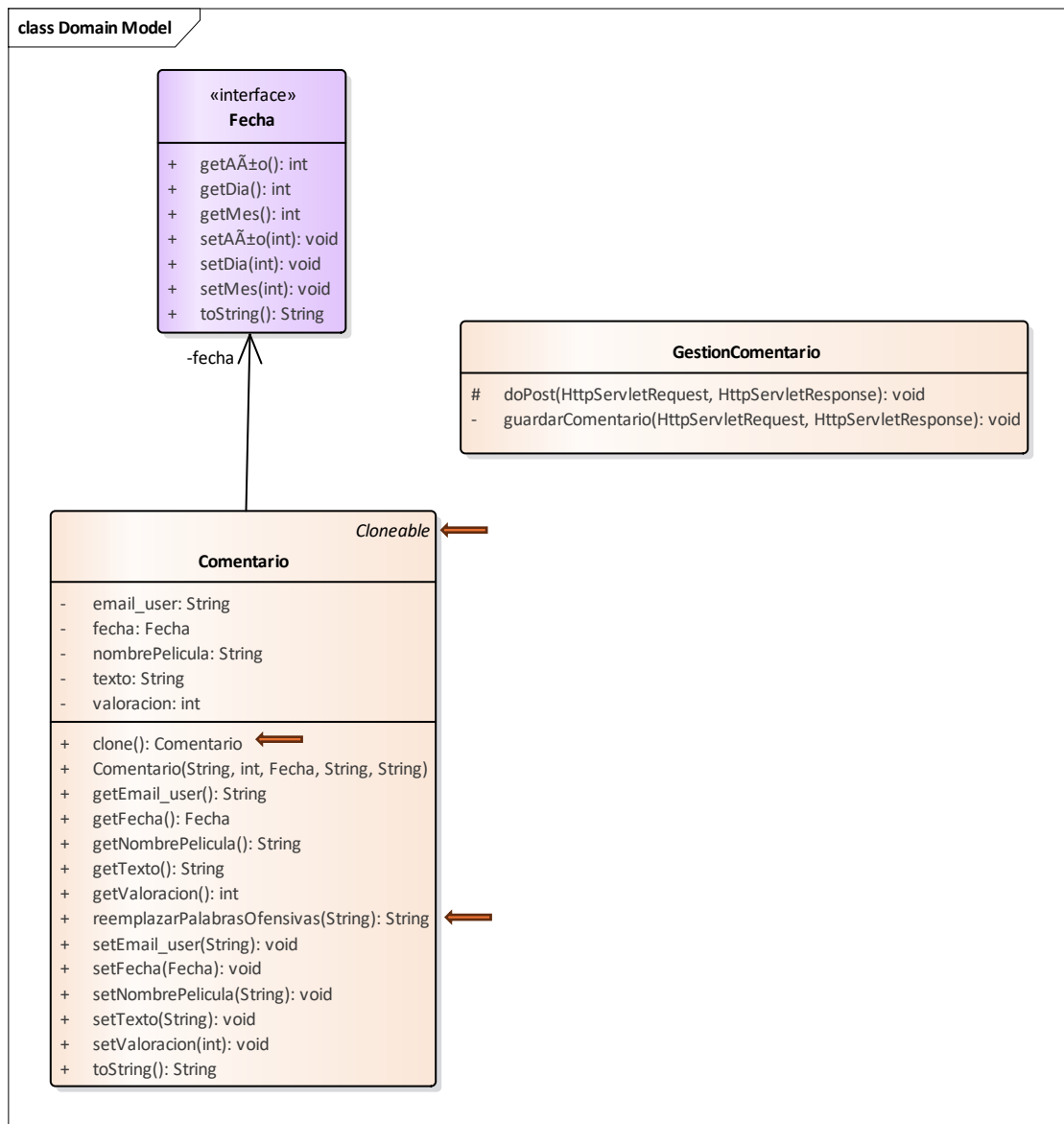
Beneficios en la Aplicación:

Eficiencia en la Creación de Comentarios: Al clonar comentarios existentes en lugar de crear nuevos desde cero, se mejora la eficiencia y se reduce la duplicación de código.

Consistencia en la Estructura de Comentarios: Al clonar un prototipo de comentario, te aseguras de que la nueva instancia tenga la misma estructura y formato que el comentario original.

Facilita la Implementación de Funcionalidades Adicionales: Puedes implementar fácilmente nuevas funcionalidades relacionadas con los comentarios en el prototipo y estas se reflejarán automáticamente en todas las instancias clonadas.

Diagrama UML y Código:



Comentario.java

```

public class Comentario implements Cloneable {

    private String texto;
    private int valoracion;
    private Fecha fecha;
    private String email_user;
    private String nombrePelicula;
  
```



```
public Comentario(String texto, int valoracion, Fecha fecha,
String email_user, String nombrePelicula) {
    this.texto = texto;
    this.valoracion = valoracion;
    this.fecha = fecha;
    this.email_user = email_user;
    this.nombrePelicula = nombrePelicula;
}
```

```
@Override
public Comentario clone() {
    try {
        return (Comentario) super.clone();
    } catch (CloneNotSupportedException e) {
        // Manejar la excepción, por ejemplo, imprimir el
error
        e.printStackTrace();
        return null;
    }
}

public String reemplazarPalabrasOfensivas(String texto) {
    // Definir un mapa de palabras ofensivas y sus
reemplazos
    Map<String, String> reemplazos = new HashMap<>();
    reemplazos.put("mierda", "maravilla");
    reemplazos.put("estupido", "asombroso");
    reemplazos.put("estupida", "asombrosa");
    reemplazos.put("idiota", "inteligente");
    reemplazos.put("odio", "amo");
    // Agregar más palabras ofensivas y sus reemplazos según
sea necesario

    // Realizar reemplazos en el texto
    for (Map.Entry<String, String> entry :
reemplazos.entrySet()) {
        String palabraOfensiva = entry.getKey();
        String reemplazo = entry.getValue();
        texto = texto.replaceAll("\\b" + palabraOfensiva +
"\\b", reemplazo);
    }
}
```

```
        return texto;  
    }
```

Como funciona:

Comentario: Es la clase que actúa como prototipo. Tiene la responsabilidad de ser clonada para crear nuevas instancias de comentarios. En este caso hemos creado una función para evitar que haya palabras ofensivas en los comentarios, y mantener sobre todo el respeto en la comunidad.

GestionComentario (Servlet): En el servlet, he utilizado el patrón Prototype al clonar un comentario existente, y con ese comentario clonado hacemos variaciones la primera es que haga la función de detectar contenido ofensivo y sustituirlo si es necesario y la otra es en el caso que ponga una valoración menos de 5 para romper las estadísticas.

Uso:

GestionComentario.java

```
Comentario comentario = new Comentario(texto, valoracion, fecha,  
emailUsuario, nombrePelícula);  
  
        //Prueba del patron prototype para mejorar las  
estadísticas del contenido  
        //Clonamos el comentario que ha escrito el usuario  
Comentario nuevoComentario = comentario.clone();  
  
        //Si el usuario ha puesto una valoración menor de 5  
if (valoracion < 5) {  
    nuevoComentario.setValoracion(5);  
}  
  
        // Realizamos la modificación del texto para  
reemplazar palabras ofensivas  
String textoModificado =  
nuevoComentario.reemplazarPalabrasOfensivas(nuevoComentario.getTexto());  
nuevoComentario.setTexto(textoModificado);  
  
        //Guardamos ese comentario en la base de datos  
DatabaseManager.getInstance().guardarComentario(nuevoComentario);
```

Patrones estructurales

PATRÓN ADAPTER

Objetivo: Sirve de intermediario entre dos clases, convirtiendo las interfaces de una clase para que pueda ser utilizada por otra.

Se utiliza cuando: Hay una clase existente y su interfaz no concuerda con la que se necesita.

Traducción entre interfaces de varios objetos.

Existe un objeto intermediario reutilizable que se encarga de gestionar todas las clases, pero no es hasta tiempo de ejecución cuando se elige la clase a utilizar.

Adaptador de objetos.

Uso en la aplicación: Tenía un problema con las fechas ya que cuando las recogía del jsp se obtenía una fecha en formato AAAA-MM-DD, y yo quería tener esa fecha en formato español DD/MM/AAAA, por lo que lo mejor para hacer este cambio era implementar un el patron adapter, para adaptar las fechas en formato americano a formato español.

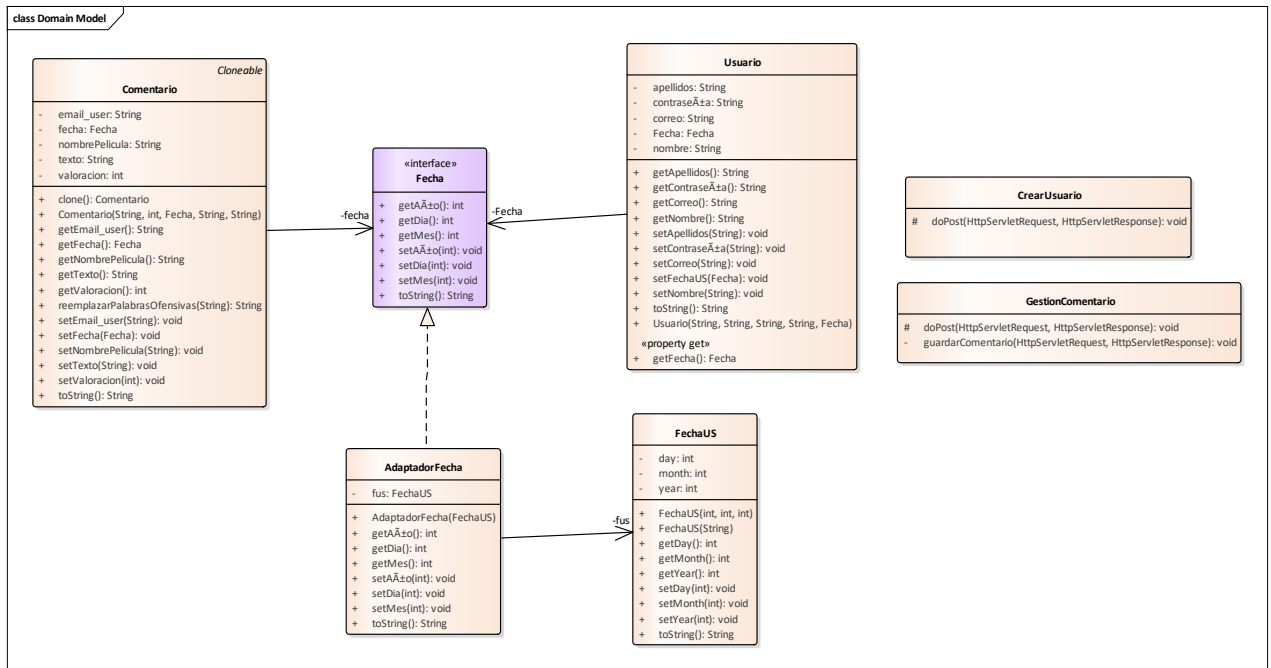
Beneficios en la Aplicación:

Reutilización de código existente: Podemos usar la clase FechaUS existente sin tener que modificarla para que se ajuste a la interfaz Fecha.

Desacoplamiento: El código principal no depende de la clase FechaUS directamente, sino de la interfaz Fecha. Esto hace que tu código sea más flexible y fácil de mantener.

Interoperabilidad: El patrón Adapter permite que clases con interfaces incompatibles trabajen juntas. Esto es especialmente útil cuando necesitas integrar clases que no puedes modificar, como las de bibliotecas de terceros.

Diagrama UML y Código:



FechaUs.java

```

public class FechaUS {
    private int year;
    private int month;
    private int day;

    public FechaUS(int month, int day, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public FechaUS(String date) {
        String[] st = date.split("-");
        String aa = st[0];
        String mm = st[1];
        String dd = st[2];

        this.day = Integer.parseInt(dd);
        this.month = Integer.parseInt(mm);
        this.year = Integer.parseInt(aa);
    }

    public String toString() {

```

```
        return year + "-" + month + "-" + day;
    }

    public int getYear() {
        return this.year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public int getDay() {
        return this.day;
    }

    public void setDay(int day) {
        this.day = day;
    }

    public int getMonth() {
        return this.month;
    }

    public void setMonth(int month) {
        this.month = month;
    }
}
```

Fecha.java

```
public interface Fecha {

    @Override
    public String toString();

    public int getAño();

    public void setAño(int año);

    public int getDia();
```



```
public void setDia(int dia);

public int getMes();

public void setMes(int mes);
}
```

AdaptadorFecha.java

```
public class AdaptadorFecha implements Fecha {

    private FechaUS fus;

    public AdaptadorFecha(FechaUS fus) {
        this.fus = fus;
    }

    @Override
    public String toString() {
        StringTokenizer st = new StringTokenizer(fus.toString(), "-");
        String aa = st.nextToken();
        String mm = st.nextToken();
        String dd = st.nextToken();

        return dd + "/" + mm + "/" + aa;
    }

    @Override
    public int getAño() {
        return this.fus.getYear();
    }

    @Override
    public void setAño(int año) {
        this.fus.setYear(año);
    }

    @Override
    public int getDia() {
```

```
        return this.fus.getDay();
    }

    @Override
    public void setDia(int dia) {
        this.fus.setDay(dia);
    }

    @Override
    public int getMes() {
        return this.fus.getMonth();
    }

    @Override
    public void setMes(int mes) {
        this.fus.setMonth(mes);
    }
}
```

Como funciona:

Creemos una interfaz Fecha que define los métodos que esperas usar en tu aplicación.

Clase FechaUS que no implementa esta interfaz y, por lo tanto, no puedes usarla directamente donde necesitas un objeto Fecha.

Clase AdaptadorFecha que implementa la interfaz Fecha y tiene un objeto FechaUS interno. Esta clase adapta los métodos de FechaUS a los métodos de Fecha, realizando las conversiones necesarias.

Creemos un AdaptadorFecha y lo usas como si fuera un Fecha.

Uso:

GestionUsuarios.java

```
// Obtén los parámetros del formulario de registro
String nombre = request.getParameter("Name");
String apellidos = request.getParameter("Apellidos");
String correo = request.getParameter("mail");
String contraseña = request.getParameter("pswd");
String fechaNacimiento = request.getParameter("Fecha-
nacimiento");
System.out.println(fechaNacimiento);
```

```
// Crea una instancia de la clase Fecha con la fecha de nacimiento
Fecha fecha;

fecha = new AdaptadorFecha(new FechaUS(fechaNacimiento));
System.out.println(fecha);

Usuario user = new Usuario(nombre, apellidos, contraseña, correo, fecha);
System.out.println(user.toString());

// Guardar el usuario en la base de datos
DatabaseManager.getInstance().guardarUsuario(user);
```

También se usa en la gestión de comentarios para almacenar la fecha en la que se ha hecho el comentario.

Una vez que introducimos una fecha veamos un ejemplo:



Vemos como la fecha que obtenemos de ese formulario que la estamos mostrando por pantalla es 2002-12-02, y la adaptamos al formato español 02/12/2002

```
Context path from ServletContext: differs from path from bundle: /|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
2002-12-02|#]
2/12/2002|#]
Usuario{nombre=David, apellidos=Bachiller Vela, contraseña=bachiller, correo=david2024@gmail.com, FechaUS=2/12/2002|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
```

PATRÓN DECORATOR

Objetivo: Añadir nuevas responsabilidades dinámicamente a un objeto sin modificar su apariencia externa o su función, es una alternativa a crear demasiadas subclases por herencia.

Se utiliza cuando: Para añadir funcionalidad a una clase sin las restricciones que implica la utilización de la herencia.

Cuando se quiera añadir funcionalidad a una clase de forma dinámica en tiempo de ejecución y que sea transparente a los usuarios.

Haya características que varíen independientemente, que deban ser aplicadas de forma dinámica y que se pueden combinar arbitrariamente sobre un componente.

Uso de la aplicación: El patrón Decorator ha sido utilizado aquí para extender las capacidades de las películas de manera flexible y modular, permitiendo la fácil incorporación de nuevas funcionalidades sin afectar la estructura existente del código. Para saber en este caso cual es el rango de edad mas recomendado en base a la clasificación por edad de las películas.

Beneficios en la aplicación:

Flexibilidad y Extensibilidad: El patrón Decorator nos permite agregar o quitar responsabilidades a un objeto existente de manera dinámica. En este caso, podemos agregar clasificaciones de edad sin modificar la clase base Pelicula. Si en el futuro necesitas agregar más funcionalidades o clasificaciones, puedes hacerlo fácilmente creando nuevos decoradores sin modificar el código existente.

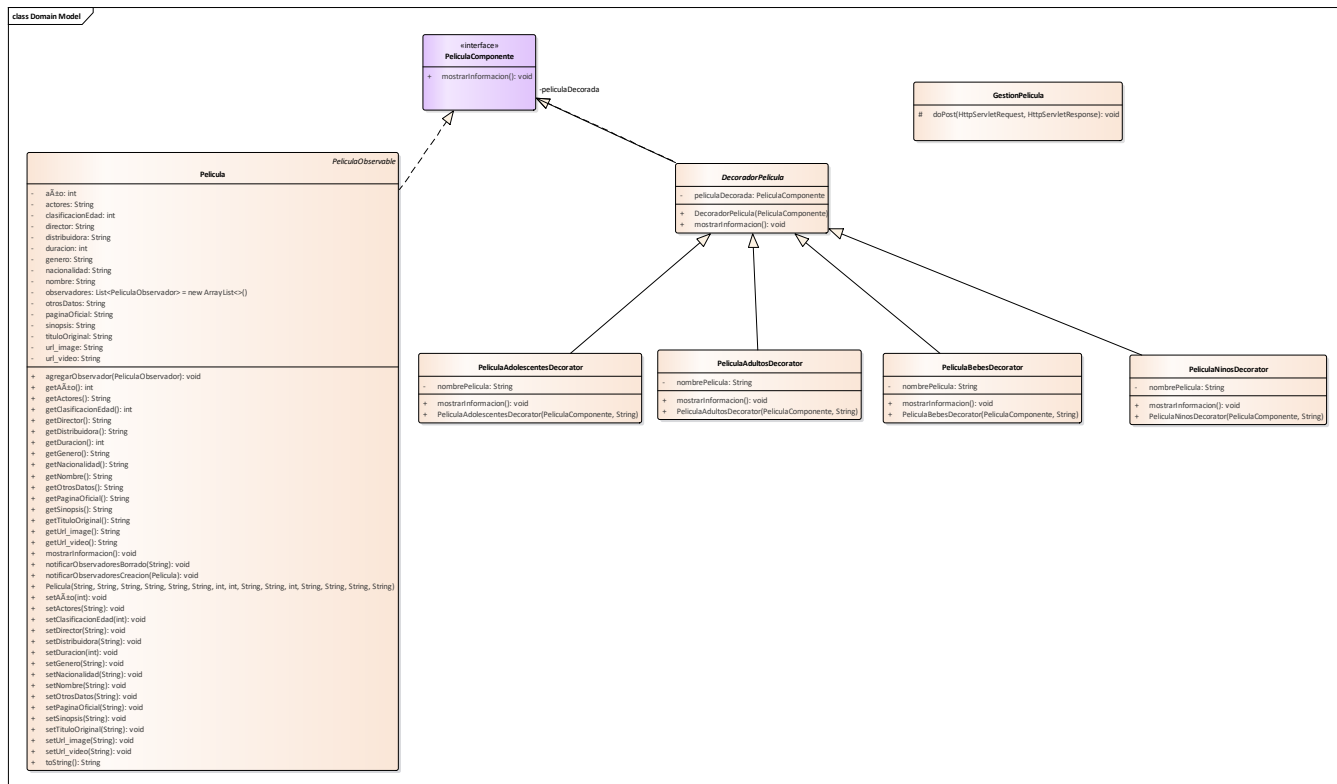
Cumple con el Principio de Responsabilidad Única: Cada clase tiene una única responsabilidad. La clase base Pelicula se encarga de la información básica de la película, mientras que los decoradores se encargan de las clasificaciones de edad específicas. Esto facilita el mantenimiento y la comprensión del código.

Reutilización de Código: Los decoradores son clases independientes y pueden reutilizarse en diferentes contextos o con diferentes componentes. Por ejemplo, si tienes otro tipo de objeto que también necesita clasificaciones de edad, puedes reutilizar los mismos decoradores.

Fácil Configuración: El uso de decoradores permite configurar de manera sencilla y personalizada las películas según sus clasificaciones de edad. Simplemente eliges qué decorador aplicar en función de la clasificación de edad y obtienes la funcionalidad específica.

Mejora de la Mantenibilidad: Al separar las responsabilidades y utilizar el patrón Decorator, el código se vuelve más modular y fácil de entender. Esto facilita futuras modificaciones y mejoras en el sistema.

Diagrama UML y Código:



PeliculaComponente.java

```
// Interfaz Componente
public interface PeliculaComponente {
    void mostrarInformacion();
}
```

Película.java (en este caso el componente concreto)

```
public class Pelicula implements PeliculaObservable, PeliculaComponente {
    //Resto del codigo...

    //Seccion del patron decorator - Componente concreto

    @Override
    public void mostrarInformacion() {
        System.out.println("Información base de la película: " + nombre);
    }
}
```

DecoradorPelicula.java

```
public abstract class DecoradorPelicula implements PeliculaComponente{
```

```
private PeliculaComponente peliculaDecorada;

public DecoradorPelicula(PeliculaComponente peliculaDecorada) {
    this.peliculaDecorada = peliculaDecorada;
}

@Override
public void mostrarInformacion() {
    peliculaDecorada.mostrarInformacion();
}
}
```

PeliculaAdolescentesDecorator.java (actúa como decorador concreto)

```
// Decorador Concreto para Películas de Adolescentes
public class PeliculaAdolescentesDecorator extends DecoradorPelicula {

    private String nombrePelicula;

    public PeliculaAdolescentesDecorator(PeliculaComponente peliculaDecorada,
String nombrePelicula) {
        super(peliculaDecorada);
        this.nombrePelicula = nombrePelicula;
    }

    @Override
    public void mostrarInformacion() {
        super.mostrarInformacion();
        System.out.println("La película '" + nombrePelicula + "' es apta para
adolescentes.");
    }
}
```

PeliculaAdultosDecorator.java (actúa como decorador concreto)

```
// Decorador Concreto para Películas de Adultos
public class PeliculaAdultosDecorator extends DecoradorPelicula {

    private String nombrePelicula;
```

```
public PeliculaAdultosDecorator(PeliculaComponente peliculaDecorada,
String nombrePelicula) {
    super(peliculaDecorada);
    this.nombrePelicula = nombrePelicula;
}

@Override
public void mostrarInformacion() {
    super.mostrarInformacion();
    System.out.println("La película '" + nombrePelicula + "' es apta para
adultos.");
}
}
```

PeliculaBebesDecorator.java (actúa como decorador concreto)

```
// Decorador Concreto para Películas de Bebés
public class PeliculaBebesDecorator extends DecoradorPelicula {

    private String nombrePelicula;

    public PeliculaBebesDecorator(PeliculaComponente peliculaDecorada, String
nombrePelicula) {
        super(peliculaDecorada);
        this.nombrePelicula = nombrePelicula;
    }

    @Override
    public void mostrarInformacion() {
        super.mostrarInformacion();
        System.out.println("La película '" + nombrePelicula + "' es apta para
bebés.");
    }
}
```

PeliculaNinosDecorator.java (actúa como decorador concreto)

```
// Decorador Concreto para Películas de Niños
public class PeliculaNinosDecorator extends DecoradorPelicula {

    private String nombrePelicula;
```

```
public PeliculaNinosDecorator(PeliculaComponente peliculaDecorada, String
nombrePelicula) {
    super(peliculaDecorada);
    this.nombrePelicula = nombrePelicula;
}

@Override
public void mostrarInformacion() {
    super.mostrarInformacion();
    System.out.println("La película '" + nombrePelicula + "' es apta para
niños.");
}
}
```

Uso:

Como hemos visto el patrón decorator nos permite añadir nuevas funcionalidades a la clase película sin modificar su estructura, por lo que cuando creamos una película dependiendo del rango de la edad de esa película vamos a crear un nuevo decorador para saber el rango de edad recomendable.

```
Pelicula pelicula = new Pelicula(nombre, sinopsis, pagina_oficial, titulo_original,
genero, nacionalidad, duracion, año, distribuidora, director, clasificacionEdad,
otrosDatos, actores, url_image, url_video);

// Crear instancias de los decoradores según la clasificación de edad
PeliculaComponente peliculaDecorada = pelicula;
if (clasificacionEdad >= 0 && clasificacionEdad <= 3) {
    //Rango de edad entre 0 y 3, luego pensada para bebes
    peliculaDecorada = new PeliculaBebesDecorator(peliculaDecorada,
nombre);

} else if (clasificacionEdad >= 4 && clasificacionEdad <= 12) {
    //Rango de edad entre 4 y 12, luego pensada para niños
    peliculaDecorada = new PeliculaNinosDecorator(peliculaDecorada,
nombre);

} else if (clasificacionEdad >= 13 && clasificacionEdad <= 16) {
    //Rango de edad entre 13 y 16, luego pensada para adolescentes
    peliculaDecorada = new
PeliculaAdolescentesDecorator(peliculaDecorada, nombre);

} else if (clasificacionEdad >= 17) {
```



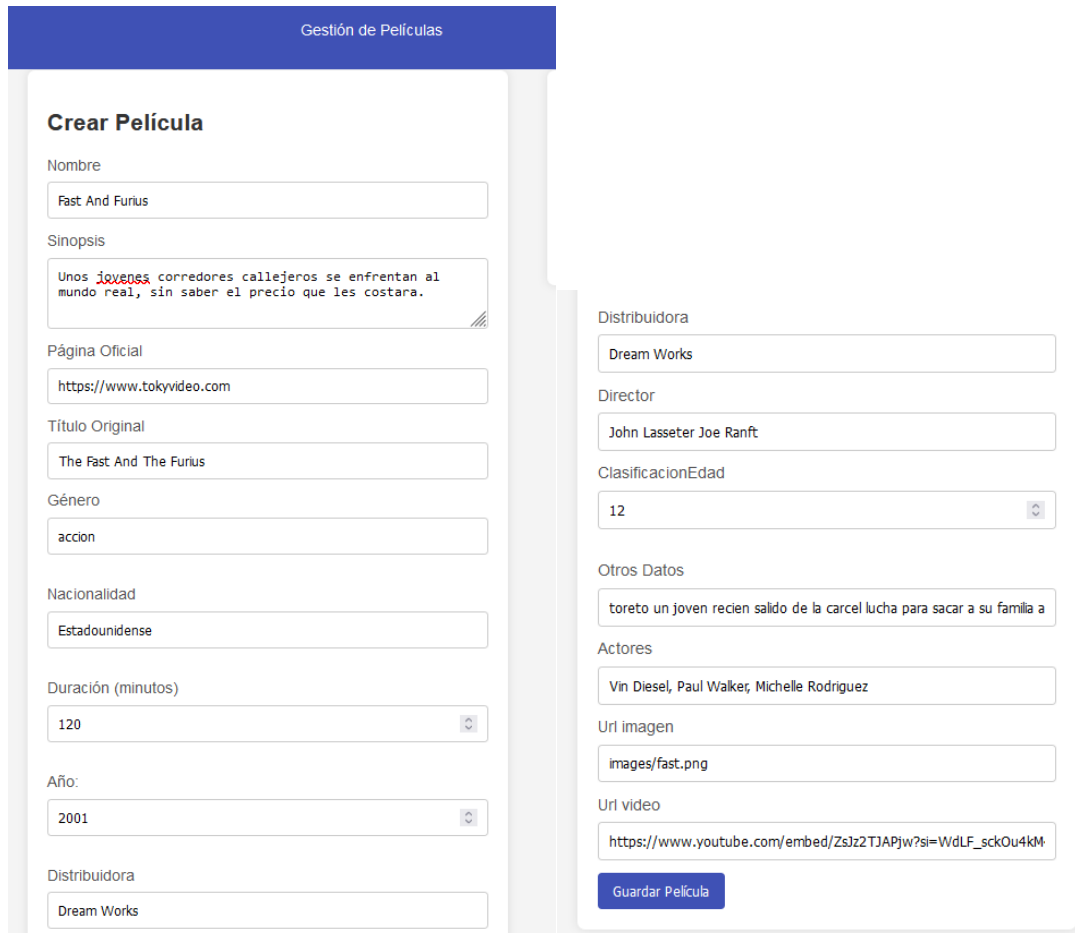
```
//Rango de edad mayor de 17, luego pensada para adultos
peliculaDecorada = new PeliculaAdultosDecorator(peliculaDecorada,
nombre);

} else {
    System.out.println("Edad no valida");
}

// Mostrar la información de la película decorada
peliculaDecorada.mostrarInformacion();
```

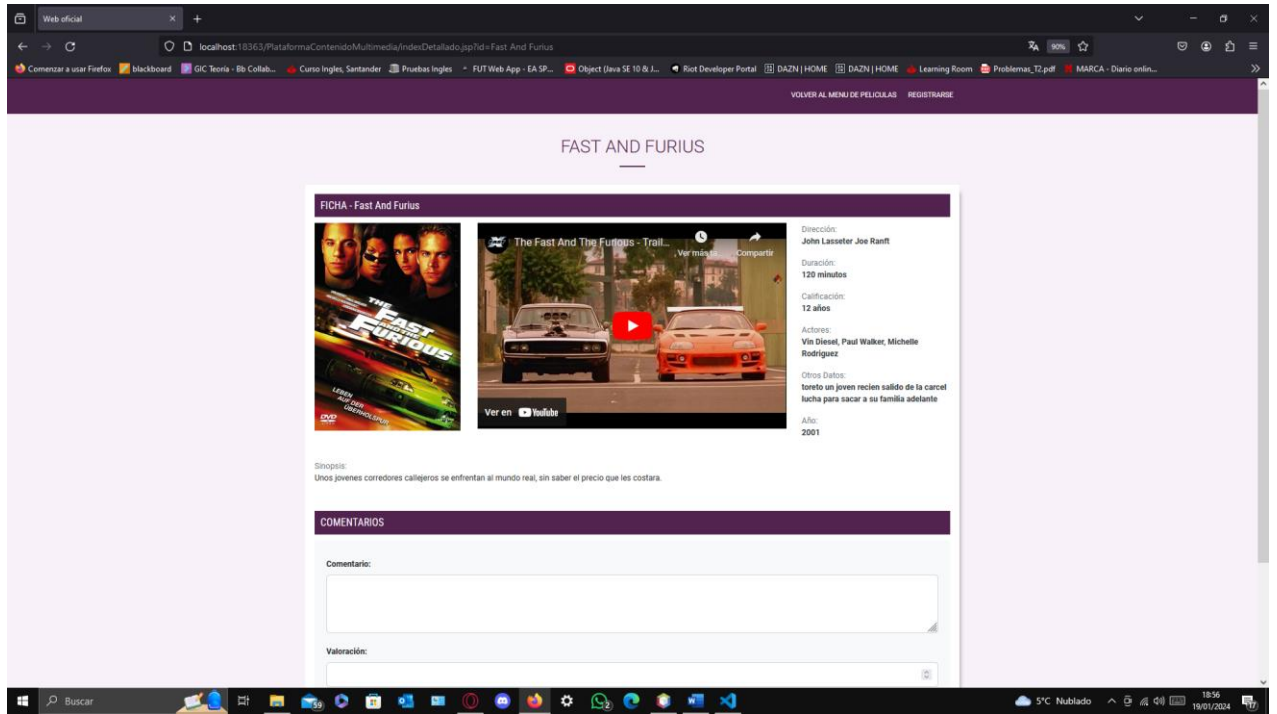
Vamos a hacer la prueba de crear una película en este caso el vamos a meter una edad de 12 años, por lo que vamos a crear un decorador de `peliculaDecorada = new PeliculaAdolescentesDecorator(peliculaDecorada, nombre);`

Y una vez creado vamos a mostrar la información, en este caso vamos a crear la película fast and furius a nuestro catalogo de películas:



Una vez que se guarde la película va a recoger la edad de este mismo y en base a esta va a crear un decorador y en base a el decorador que sea va a mostrar una información u otra.

Se nos añade la película:

[illegible]

Patrones de comportamiento

PATRÓN OBSERVER

Objetivo: Permite definir dependencias 1-a-muchos de forma que los cambios en un objeto e comuniquen a los objetos que dependen de él.

Se utiliza cuando: Existe al menos un emisor de mensajes.

Uno o más receptores de mensajes podrían variar dentro de una aplicación o entre aplicaciones.

Si se produce un cambio en un objeto, se requiere el cambio de otros y no se sabe cuántos se necesitan cambiar.

No queremos que estén fuertemente acoplados.

Uso de la aplicación:

Cuando se realiza una acción con la película, ya sea que se crea una o se borra

Registro de observadores: Antes de crear la película, se registra un observador específico para el objeto Película. En mi caso, este observador es la clase EstadísticasPelículas. Esto se hace mediante la llamada a `pelicula.agregarObservador(new EstadísticasPelículas())`;

Guardado en la base de datos: Después de registrar los observadores, se procede a guardar la película utilizando `DatabaseManager.getInstance().guardarPelicula(pelicula)`;

Notificación a observadores: Luego de guardar la película, se notifica a todos los observadores registrados que una nueva película ha sido creada. Esto se realiza mediante `pelicula.notificarObservadoresCreacion(pelicula)`; Cada observador (en este caso, EstadísticasPelículas) recibe la notificación.

Acciones del observador: La clase EstadísticasPelículas dentro del paquete model, al implementar PelículaObservador, tiene un método `actualizarCreacion(Película pelicula)`. Este método se ejecuta cuando el observador recibe la notificación de que se ha creado una película. Aquí es donde puedes realizar acciones adicionales relacionadas con las estadísticas de películas, como imprimir un mensaje indicando que se ha creado una película específica.

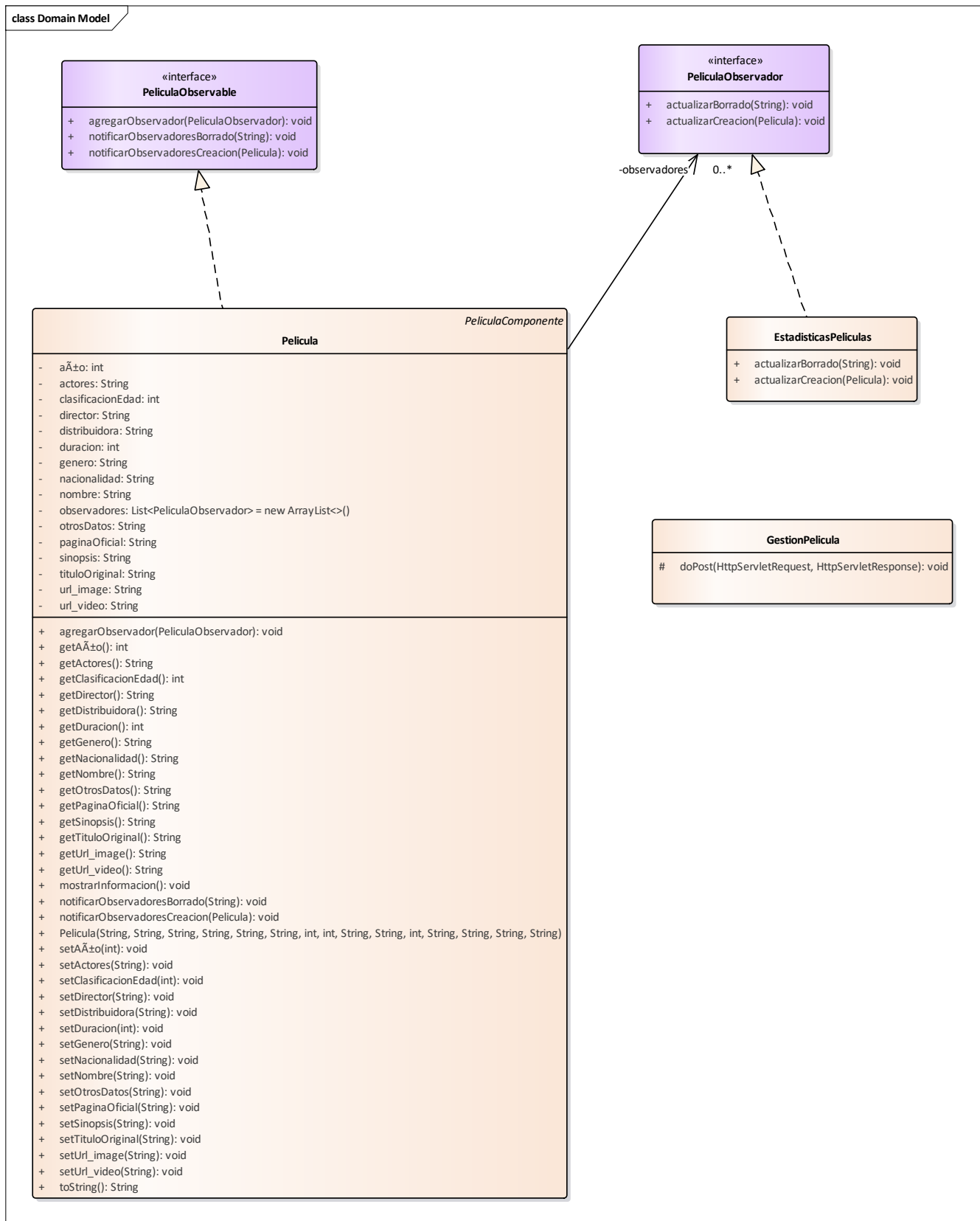
Beneficios en la aplicación:

Desacoplamiento: El patrón Observer permite desacoplar las clases observadoras de la clase que está siendo observada (Película en este caso). Esto significa que las estadísticas no están fuertemente vinculadas a la lógica de creación de películas. Si en el futuro se quiere agregar más observadores para otros eventos relacionados con películas, se puede hacer sin modificar la lógica de la clase Película.

Extensibilidad: Se puede agregar fácilmente nuevos observadores para manejar eventos específicos sin cambiar el código existente.

Mantenimiento: El código es más mantenible, ya que cada clase tiene una responsabilidad clara y se sigue el principio de responsabilidad única. Si se necesita cambiar cómo se manejan las estadísticas de películas, se puede hacer dentro de la clase EstadísticasPelículas sin afectar otras partes del sistema.

Diagrama UML y código:



PeliculaObservable.java (actúa como el observable)

```
/*
Creamos una interfaz llamada PeliculaObservable,
que define los métodos agregarObservador y notificarObservadores.
*/
public interface PeliculaObservable {

    void agregarObservador(PeliculaObservador observador);
    void notificarObservadoresCreacion(Pelicula pelicula);
    void notificarObservadoresBorrado(String nombrePelicula);

}
```

Pelicula.java (actúa como el observable concreto)

```
public class Pelicula implements PeliculaObservable, PeliculaComponente {
    //Hacemos que la clase Pelicula implemente la interfaz PeliculaObservable.

    // Nuevos atributos y método para implementar el patrón Observable
    private List<PeliculaObservador> observadores = new ArrayList<>();

    @Override
    public void agregarObservador(PeliculaObservador observador) {
        observadores.add(observador);
    }

    @Override
    public void notificarObservadoresCreacion(Pelicula pelicula) {
        for (PeliculaObservador observador : observadores) {
            observador.actualizarCreacion(pelicula);
        }
    }

    @Override
    public void notificarObservadoresBorrado(String nombrePelicula) {
        for (PeliculaObservador observador : observadores) {
            observador.actualizarBorrado(nombrePelicula);
        }
    }
}
```

PeliculaObservador.java

```
public interface PeliculaObservador {  
    void actualizarCreacion(Pelicula pelicula);  
    void actualizarBorrado(String nombrePelicula);  
}
```

EstadisticasPleiculas.java (actúa como observador concreto).

```
public class EstadisticasPeliculas implements PeliculaObservador {  
    // Implementamos el método actualizar para realizar acciones cuando se crea una  
    película  
  
    /*  
        La creación de una clase de estadísticas es útil para recopilar y presentar datos  
        estadísticos  
        relacionados con las películas en tu aplicación. Algunos ejemplos de estadísticas  
        que se podrían incluir son:  
  
        Número total de películas, Promedio de duración de películas, Número de películas por  
        género, Año con más películas  
        Clasificación de edad más común  
    */  
  
    @Override  
    public void actualizarCreacion(Pelicula pelicula) {  
        System.out.println("Se ha creado la película: " + pelicula.getNombre());  
        // Realizar acciones adicionales, si es necesario...  
    }  
  
    @Override  
    public void actualizarBorrado(String nombrePelicula) {  
        System.out.println("Se ha borrado la película: " + nombrePelicula);  
        // Realizar acciones adicionales, si es necesario...  
    }  
}
```

Uso:

Esta implementación permite notificar a los observadores específicamente cuando se crea una película (notificarObservadoresCreacion) y cuando se borra una película (notificarObservadoresBorrado).

Los observadores pueden realizar acciones personalizadas en respuesta a estos eventos, como registrar información, actualizar vistas, etc. Puedes tener diferentes tipos de observadores que reaccionen de manera diferente a estas notificaciones. Pero en este caso solo nos hemos centrado en el ámbito de la gestión de películas para saber en cada momento la acción que se realiza ya sea de creación o de borrar.

GestionPelículas.java (actúa como el main donde usamos los observadores)

```
Pelicula pelicula = new Pelicula(nombre, sinopsis, pagina_oficial, titulo_original,
genero, nacionalidad, duracion, año, distribuidora, director, clasificacionEdad,
otrosDatos, actores, url_image, url_video);
```

```
// Registra observadores
```

```
    pelicula.agregarObservador(new EstadisticasPelículas());
```

```
// Guardar la película en la base de datos
```

```
    DatabaseManager.getInstance().guardarPelicula(pelicula);
```

```
// Notificar a los observadores
```

```
    pelicula.notificarObservadoresCreacion(pelicula);
```

```
// Obtén la película por su nombre
```

```
    Pelicula pelicula =
```

```
    DatabaseManager.getInstance().getPeliculaPorNombre(nombrePeliculaABorrar);
```

```
    if (pelicula != null) {
```

```
        //Registra observadores
```

```
        pelicula.agregarObservador(new EstadisticasPelículas());
```

```
// Borra la película de la base de datos
```

```
    DatabaseManager.getInstance().borrarPelicula(pelicula);
```

```
// Notificar a los observadores de borrado de película
```

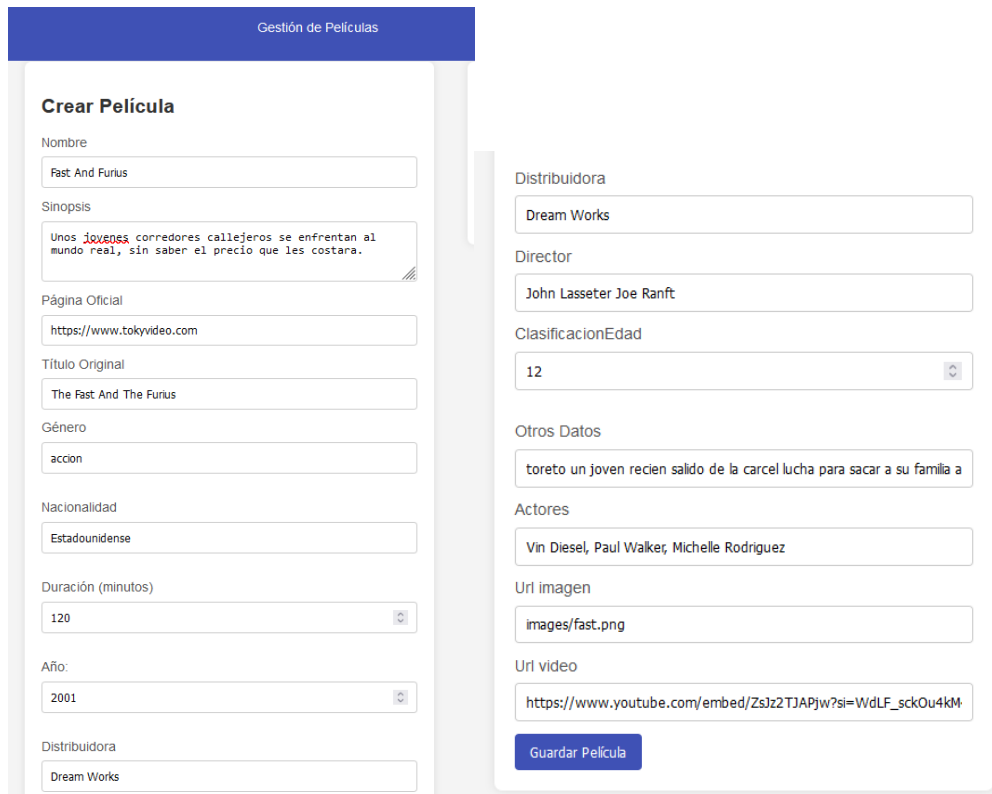
```
    pelicula.notificarObservadoresBorrado(pelicula);
```

```
response.sendRedirect("gestionPelículas.jsp"); // Redirigir a la
página principal

System.out.println("Película borrada con éxito");
} else {
    response.getWriter().println("No se encontró la película a
    borrar.");
}
```

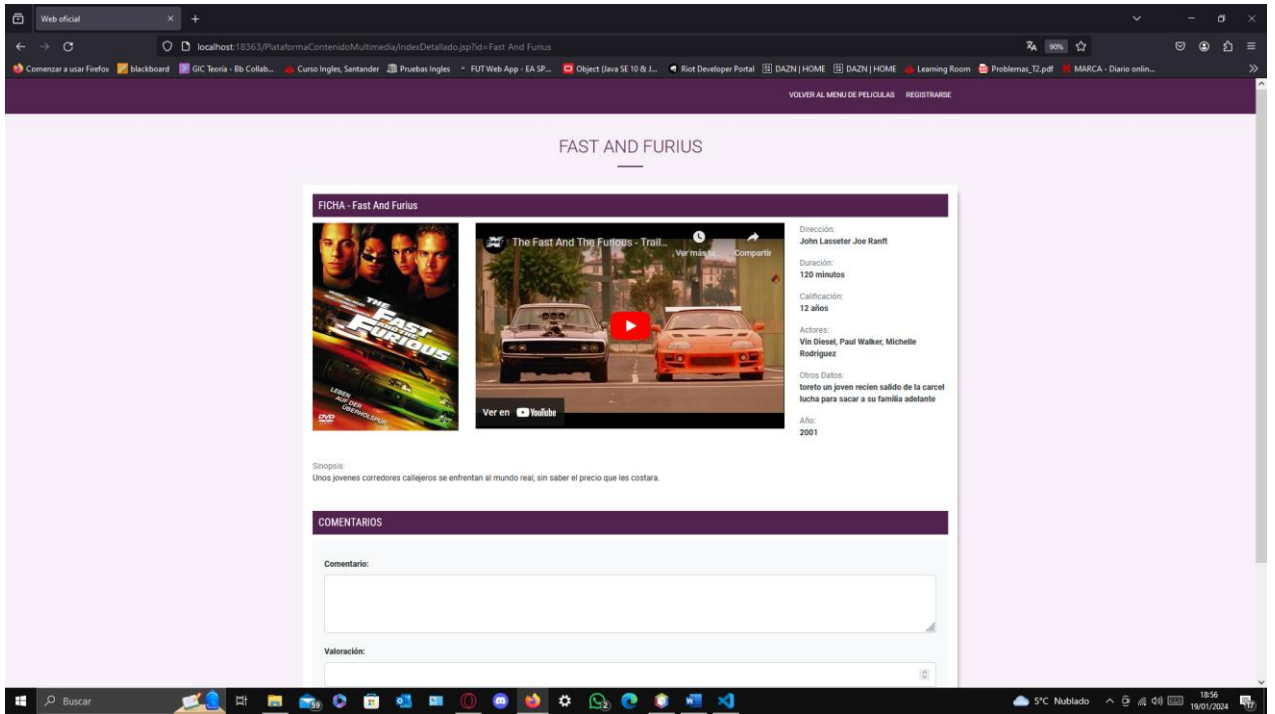
Vamos a hacer la prueba de crear una película en este caso el vamos a meter, para crear un nuevo observador y notificar la acción que se realice, en este caso vamos a crearla y posteriormente vamos a borrarla.

Y una vez creado vamos a mostrar la información, en este caso vamos a crear la película fast and furius a nuestro catalogo de películas:



Una vez que se guarde la película va a recoger la edad de este misma y en base a esta va a crear un decorador y en base a él decorador que sea va a mostrar una información u otra.

Se nos añade la película:

[illegible]

Ahora similar en el panel de administración vamos a borrarla:

Borrar Película

Selecciona una película para borrar:

Fast And Furious

Borrar Película

```
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
He obtenido correctamente el usuario de la bbdd|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha borrado la película: Fast And Furious|#]
Película borrada con éxito|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
Se ha conectado|#]
Se ha cerrado la conexión|#]
```

PATRÓN STRATEGY

Objetivo: Definir un grupo de clases que representan un conjunto de posibles comportamientos. Estos comportamientos pueden ser fácilmente intercambiados en una aplicación, modificando la funcionalidad en cualquier instante.

Se utiliza cuando: Muchas clases relacionadas sólo se diferencian en su comportamiento.

Se necesitan distintas variables de un algoritmo.

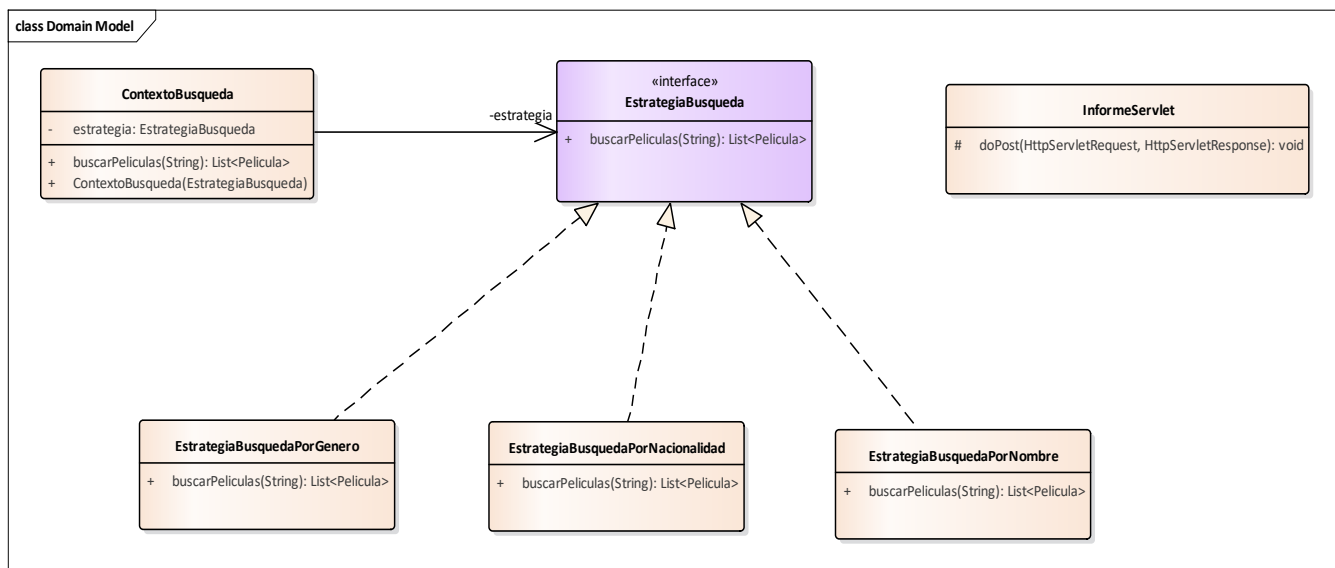
Una clase tiene distintos comportamientos posibles que aparecen como instrucciones condicionales en sus métodos.

No se sabe qué aproximación utilizar hasta el momento de ejecución

Uso en la aplicación: Permite al cliente poder elegir el algoritmo que desea utilizar de forma dinámica, sin alterar la estructura del código. En este caso, se utiliza para implementar diferentes estrategias de búsqueda o filtrado de películas. Utilizando los datos que ya tenemos almacenados en la base de datos, para hacer búsquedas por nacionalidad, por género, por nombre y poder obtener información de estas.

Beneficios en la aplicación: Este patrón ofrece flexibilidad al permitir que nuevas estrategias de búsqueda se agreguen fácilmente sin cambiar el código existente. Además, proporciona una separación clara de responsabilidades entre el cliente que utiliza la estrategia y las estrategias individuales que implementan algoritmos específicos

Diagrama UML y código:



EstrategiaBusqueda.java

```

public interface EstrategiaBusqueda {

    List<Película> buscarPelículas(String parametro) throws SQLException;

}
  
```

EstrategiaBusquedaPorGenero.java (Primera de las tres estrategias concretas)

```
public class EstrategiaBusquedaPorGenero implements EstrategiaBusqueda {

    @Override
    public List<Pelicula> buscarPeliculas(String parametro) throws SQLException {
        return DatabaseManager.getPeliculasPorGenero(parametro);
    }
}
```

EstrategiaBusquedaPorNacionalidad.java (Segunda de las tres estrategias concretas)

```
public class EstrategiaBusquedaPorNacionalidad implements EstrategiaBusqueda {

    @Override
    public List<Pelicula> buscarPeliculas(String parametro) throws SQLException {
        return DatabaseManager.getPeliculasPorNacionalidad(parametro);
    }
}
```

EstrategiaBusquedaPorNombre.java (Tercera de las tres estrategias concretas)

```
public class EstrategiaBusquedaPorNombre implements EstrategiaBusqueda {

    @Override
    public List<Pelicula> buscarPeliculas(String parametro) throws SQLException {
        return DatabaseManager.getPeliculasPorNombre(parametro);
    }
}
```

ContextoBusqueda.java

```
public class ContextoBusqueda {

    private EstrategiaBusqueda estrategia;

    public ContextoBusqueda(EstrategiaBusqueda estrategia) {
        this.estrategia = estrategia;
    }

    public List<Pelicula> buscarPeliculas(String parametro) throws SQLException {
        return estrategia.buscarPeliculas(parametro);
    }
}
```

```
}  
}
```

Uso:

Estrategia (EstrategiaBusqueda):

Las clases de EstrategiaBusquedaPor..., ya sea nombre, nacionalidad, o genero son clases concretas que implementan la interfaz EstrategiaBusqueda.

Cada una de estas estrategias define un algoritmo específico para buscar películas según el nombre, género o nacionalidad, respectivamente.

Contexto (ContextoBusqueda):

ContextoBusqueda es la clase que utiliza una estrategia concreta para realizar la búsqueda.

Tiene un método buscarPelículas que toma un parámetro y delega la búsqueda a la estrategia correspondiente.

Cliente (servlet InformeServlet):

El servlet recibe la estrategia seleccionada y el parámetro de búsqueda desde el formulario JSP.

Crea un objeto ContextoBusqueda y le asigna la estrategia seleccionada.

Llama al método buscarPelículas del contexto para realizar la búsqueda.

InformesServlet.java

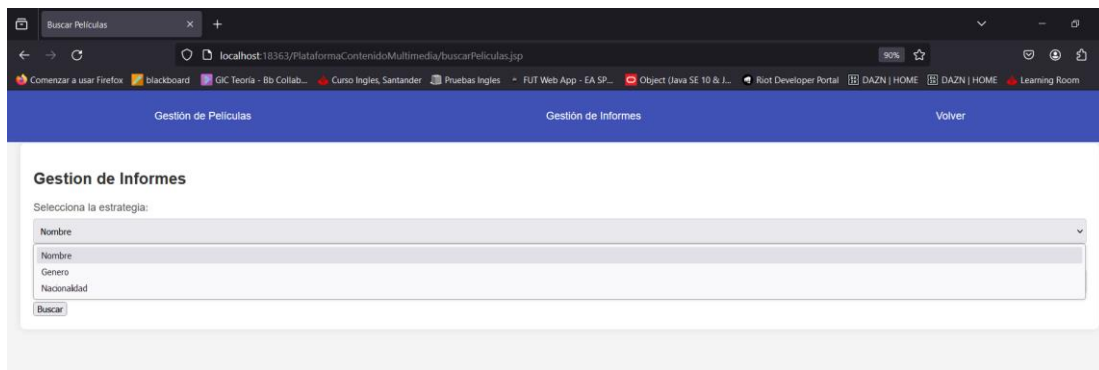
```
@Override  
protected void doPost(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    String estrategia = request.getParameter("estrategia");  
    System.out.println(estrategia);  
    String parametro = request.getParameter("parametro");  
  
    // Crear el contexto con la estrategia seleccionada  
    ContextoBusqueda contexto = null;  
  
    switch (estrategia) {  
        case "nombre":  
            contexto = new ContextoBusqueda(new EstrategiaBusquedaPorNombre());  
            break;  
        case "genero":  
            contexto = new ContextoBusqueda(new EstrategiaBusquedaPorGenero());  
    }
```

```
        break;
    case "nacionalidad":
        contexto = new ContextoBusqueda(new
EstrategiaBusquedaPorNacionalidad());
        break;
    default:
        response.getWriter().println("<p>Estrategia no válido.</p>");
        break;
    }

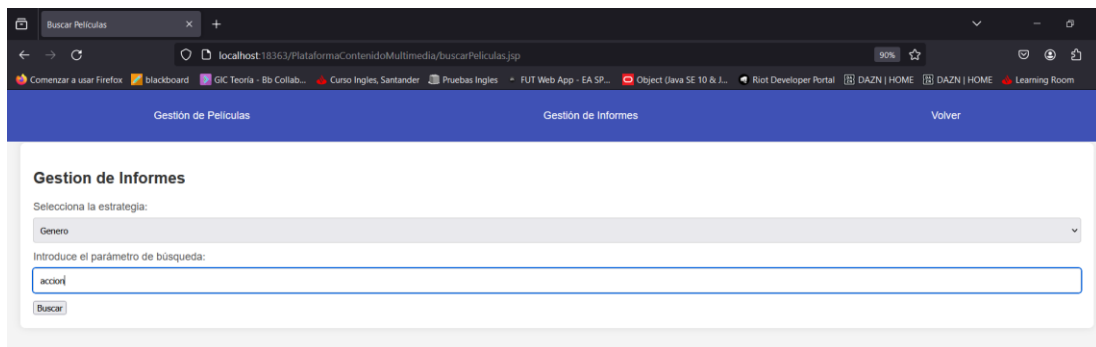
// Validar que contexto no sea null
if (contexto != null) {
    // Realizar la búsqueda con la estrategia seleccionada
    try {
        List<Película> peliculas = contexto.buscarPeliculas(parametro);

        // Puedes mostrar los resultados de la búsqueda en la respuesta
        response.getWriter().println("<h2>Resultados de la búsqueda:</h2>");
        for (Película pelicula : peliculas) {
            response.getWriter().println("<p>" + pelicula.toString() +
"</p>");
        }
    } catch (SQLException e) {
        e.printStackTrace(); // Manejar la excepción adecuadamente
        response.getWriter().println("<p>Error en la búsqueda.</p>");
    }
} else {
    // Manejar el caso donde contexto es null
    response.getWriter().println("<p>Contexto no válido.</p>");
}
}
```

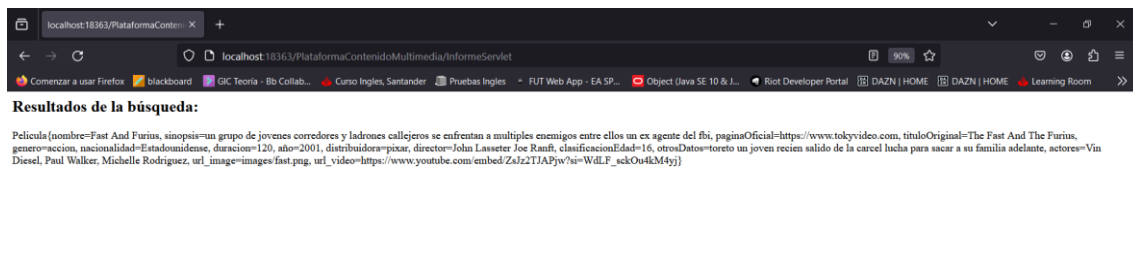
Hacemos consultas de información seleccionando una estrategia u otra.



Una vez seleccionado la estrategia establecemos el parámetro que queremos consultar:



Simplemente veremos de las películas que coincidan con esos parámetros un **toString()**.



```
Pelicula[{nombre="Fast And Furious", sinopsis="un grupo de jóvenes corredores y ladrones callejeros se enfrentan a múltiples enemigos entre ellos un ex agente del fbi", paginaOficial="https://www.tokysvideo.com", tituloOriginal="The Fast And The Furious", genero="accion", nacionalidad="Estadounidense", duracion=120, año=2001, distribuidora="pixar", director="John Lawmeter Joe Rauff", clasificacionEdad=16, otrosDatos="toreto un joven recién salido de la cárcel lucha para sacar a su familia adelante", actores="Vin Diesel, Paul Walker, Michelle Rodriguez", url_image="images/fast.png", url_video="https://www.youtube.com/embed/ZsJz2TJAjw?si=WdLF_sckOu4kM4yzj}"]
```

Aplicación

Base de datos

Se trata de una aplicación web, pero como lo principal de este proyecto son los patrones software de diseño y no la web como tal, he implementado una base de datos sencilla con un usuario que visiona un contenido, en este caso las películas y ese mismo usuario puede hacer comentarios de esas películas, por otro lado estará el panel de administración pero ese no nos interesa en la base de datos, ya que el administrador usará las películas, los usuarios y los comentarios que se vayan guardando en la base de datos para trabajar con estos.

Esquema de información y modelo E/R propuesto.

Para poder elaborar el modelo entidad/relación que haga referencia a la manera que se relacionan los diferentes objetos y personas de nuestra aplicación, hemos empleado la herramienta Pgmodeler, esta se puede encontrar en la siguiente [web](#).

Al analizar detenidamente el texto y haciendo uso de lo expuesto en el enunciado, hemos encontrado las siguientes entidades (con sus respectivos atributos):

Película: entidad que representa la información fundamental de una película que se va a poder ver en nuestra plataforma.

- Atributos:
 - nombrePelícula (Clave Primaria)
 - sinopsis
 - paginaOficial
 - tituloOriginal
 - genero
 - nacionalidad
 - duracion
 - anho
 - distribuidora
 - director
 - clasificacionEdad
 - otrosDatos
 - actores
 - url_image

- url_video
-

Usuario: entidad que almacena la información de los usuarios registrados en el sistema.

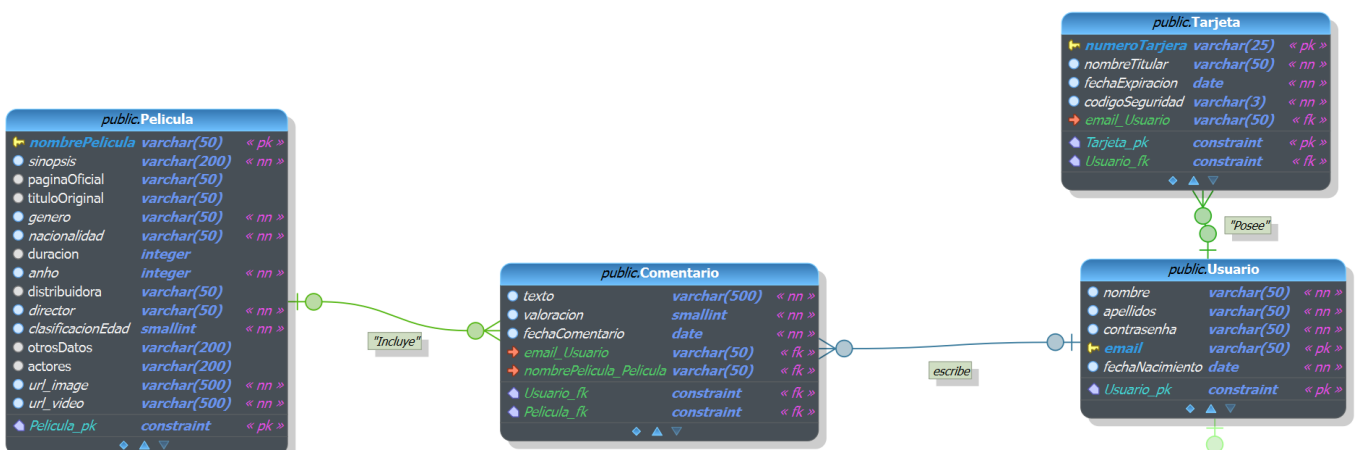
- Atributos:
 - nombre
 - apellidos
 - contraseña
 - email (Clave Primaria)
 - fechaNacimiento

Comentario: entidad que almacena los comentarios de los usuarios sobre las películas.

- Atributos:
 - texto
 - valoracion
 - fechaComentario

Estudiadas las entidades con sus respectivos atributos, vamos a presentar las relaciones que hemos creído convenientes:

- La relación entre "Usuario" y "Comentario" se establece mediante la clave foránea email_Usuario, vinculando los comentarios con los usuarios correspondientes.
- La relación entre "Película" y "Comentario" se establece mediante la clave foránea nombrePelícula_Película, vinculando los comentarios con las películas correspondientes.

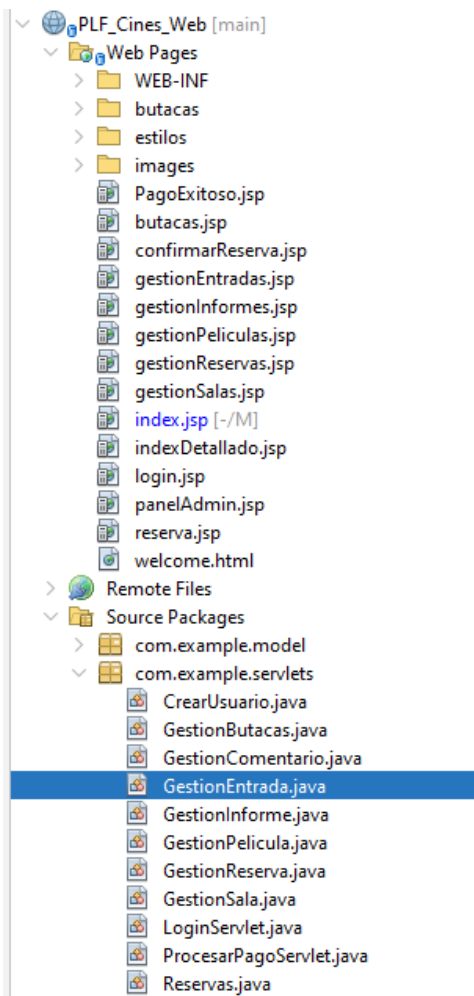


Programación en JAVA con el patrón MVC.

Como se ha comentado en la introducción de este apartado, nosotros hemos implementado en todo momento el patrón MVC, por un lado, tenemos todos los archivos.jsp que van a actuar como vista, lanzando peticiones al controlador, que son todas las clases del paquete **com.example.servlets**.

Luego en los servlets devolvemos la información de esas peticiones si es necesario, o gestionamos información de las sesiones, o por último, la opción que más hemos llevado a cabo que es el contacto con el modelo que es nuestra base de datos, toda esa gestión de la base de datos, se ha llevado a cabo en la clase **DataBaseManager.java** dentro de paquete **com.example.model**, acciones para poder almacenar usuarios películas, salas, para poder consultar estas pasándole un nombre, para poder modificarlas, para poder borrarlas, una función que devuelva un arraylist con todas las películas.

Gestión del administrador:



En todo momento enviamos un método post de la vista al controlador para que este almacene esos datos en el modelo.

Vamos a ver ejemplo de gestión de películas, paso a paso para que se entienda como se aplica el patrón, ya que en el resto de los casos es igual, tanta gestión de salas, gestión de entradas, de informes, etc.

Lo primero es ver la gestión del modelo y es el cómo almacenamos e Utilizamos toda la información de la base de datos.

Clase **DataBaseManager.java**

Lo primero que hacemos es crear la estructura básica con los datos que queramos definiendo el constructor y los getters y setters:

```
private String nombre;

private String sinopsis;

private String paginaOficial;

private String tituloOriginal;


private String genero;

private String nacionalidad;

private int duracion;

private int año;

private String distribuidora;

private String director;

private int clasificacionEdad;

private String otrosDatos;

private String actores;

private String url_image;

private String url_video;


public Pelicula(String nombre, String sinopsis, String paginaOficial, String tituloOriginal,
String genero, String nacionalidad, int duracion, int año, String distribuidora, String
director, int clasificacionEdad, String otrosDatos, String actores, String url_image, String
url_video) {

    this.nombre = nombre;

    this.sinopsis = sinopsis;

    this.paginaOficial = paginaOficial;

    this.tituloOriginal = tituloOriginal;
```

```
this.genero = genero;

this.nacionalidad = nacionalidad;

this.duracion = duracion;

this.año = año;

this.distribuidora = distribuidora;

this.director = director;

this.clasificacionEdad = clasificacionEdad;

this.otrosDatos = otrosDatos;

this.actores = actores;

this.url_image = url_image;

this.url_video = url_video;

}
```

Creamos la gestión de la película, acciones como guardar película, modificar película, borrar película, getAllPelículas, las cuales usaremos a lo largo de toda la aplicación:

```
public static void guardarPelícula(Película película) throws SQLException {

    abrirConexion();

    System.out.println("GuardarPelícula");

    try {

        if (película != null) {

            String sql = "INSERT INTO pelicula (nombrepelicula, sinopsis, paginaoficial, titulooriginal,
            genero, nacionalidad, duracion, anho, distribuidora, director, clasificacionEdad, otrosdatos, actores,
            url_image, url_video) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";

            try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {

                preparedStatement.setString(1, película.getNombre());

                preparedStatement.setString(2, película.getSinopsis());

                preparedStatement.setString(3, película.getPaginaOficial());

                preparedStatement.setString(4, película.getTituloOriginal());

                preparedStatement.setString(5, película.getGenero());

                preparedStatement.setString(6, película.getNacionalidad());

            }

        }

    }

}
```

```
        preparedStatement.setInt(7, pelicula.getDuracion());

        preparedStatement.setInt(8, pelicula.getAño());

        preparedStatement.setString(9, pelicula.getDistribuidora());

        preparedStatement.setString(10, pelicula.getDirector());

        preparedStatement.setInt(11, pelicula.getClasificacionEdad());

        preparedStatement.setString(12, pelicula.getOtrosDatos());

        preparedStatement.setString(13, pelicula.getActores());

        preparedStatement.setString(14, pelicula.getUrl_image());

        preparedStatement.setString(15, pelicula.getUrl_video());

        preparedStatement.executeUpdate();

    }

    } else {

        System.out.println("Error: Pelicula es nula.");

    }

} catch (Exception e) {

    e.printStackTrace();

} finally {

    cerrarConexion();

}

}

public static List<Pelicula> getAllPeliculas() throws SQLException {

    abrirConexion();

    List<Pelicula> peliculas = new ArrayList<>();

    try {

        String sql = "SELECT * FROM pelicula";

        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
```

```
try (ResultSet resultSet = preparedStatement.executeQuery()) {

    while (resultSet.next()) {

        Pelicula pelicula = new Pelicula(

            resultSet.getString("nombrepelicula"),

            resultSet.getString("sinopsis"),

            resultSet.getString("paginaOficial"),

            resultSet.getString("titulooriginal"),

            resultSet.getString("genero"),

            resultSet.getString("nacionalidad"),

            resultSet.getInt("duracion"),

            resultSet.getInt("anho"),

            resultSet.getString("distribuidora"),

            resultSet.getString("director"),

            resultSet.getInt("clasificacionEdad"),

            resultSet.getString("otrosdatos"),

            resultSet.getString("actores"),

            resultSet.getString("url_image"),

            resultSet.getString("url_video")

        );

        peliculas.add(pelicula);

    }

}

} finally {

    cerrarConexion();

}

return peliculas;

}
```

```
public static Pelicula getPeliculaPorNombre(String nombre) throws SQLException {  
  
    abrirConexion();  
  
    try {  
  
        String sql = "SELECT * FROM pelicula WHERE nombrepelicula = ?";  
  
        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {  
  
            preparedStatement.setString(1, nombre);  
  
            try (ResultSet resultSet = preparedStatement.executeQuery()) {  
  
                if (resultSet.next()) {  
  
                    return new Pelicula(  
  
                        resultSet.getString("nombrepelicula"),  
  
                        resultSet.getString("sinopsis"),  
  
                        resultSet.getString("paginaOficial"),  
  
                        resultSet.getString("titulooriginal"),  
  
                        resultSet.getString("genero"),  
  
                        resultSet.getString("nacionalidad"),  
  
                        resultSet.getInt("duracion"),  
  
                        resultSet.getInt("anho"),  
  
                        resultSet.getString("distribuidora"),  
  
                        resultSet.getString("director"),  
  
                        resultSet.getInt("clasificacionEdad"),  
  
                        resultSet.getString("otrosdatos"),  
  
                        resultSet.getString("actores"),  
  
                        resultSet.getString("url_image"),  
  
                        resultSet.getString("url_video")  
  
                    );  
  
                }  
  
            }  
  
        }  
  
    }  
}
```

```
    }

    } finally {

        cerrarConexion();

    }

    return null;

}

public static void borrarPelicula(Pelicula pelicula) throws SQLException {

    abrirConexion();

    try {

        String sql = "DELETE FROM pelicula WHERE nombrepelicula = ?";

        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {

            preparedStatement.setString(1, pelicula.getNombre());

            preparedStatement.executeUpdate();

        }

    } finally {

        cerrarConexion();

    }

}

public static void modificarPelicula(String nombreActual, Pelicula nuevaPelicula) throws SQLException {

    abrirConexion();

    try {

        String sql = "UPDATE pelicula SET nombrepelicula=?, sinopsis=?, paginaoficial=?, titulooriginal=?,
genero=?, nacionalidad=?, duracion=?, anho=?, distribuidora=?, director=?, clasificacionedad=?, otrosdatos=?,
actores=?, url_image=?, url_video=? WHERE nombrepelicula=?";

        try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {

            preparedStatement.setString(1, nuevaPelicula.getNombre());

            preparedStatement.setString(2, nuevaPelicula.getSinopsis());
```



```
        preparedStatement.setString(3, nuevaPelícula.getPáginaOficial());

        preparedStatement.setString(4, nuevaPelícula.getTituloOriginal());

        preparedStatement.setString(5, nuevaPelícula.getGenero());

        preparedStatement.setString(6, nuevaPelícula.getNacionalidad());

        preparedStatement.setInt(7, nuevaPelícula.getDuracion());

        preparedStatement.setInt(8, nuevaPelícula.getAño());

        preparedStatement.setString(9, nuevaPelícula.getDistribuidora());

        preparedStatement.setString(10, nuevaPelícula.getDirector());

        preparedStatement.setInt(11, nuevaPelícula.getClasificaciónEdad());

        preparedStatement.setString(12, nuevaPelícula.getOtrosDatos());

        preparedStatement.setString(13, nuevaPelícula.getActores());

        preparedStatement.setString(14, nuevaPelícula.getUrl_image());

        preparedStatement.setString(15, nuevaPelícula.getUrl_video());

        preparedStatement.setString(16, nombreActual); // Condición para actualizar la película
específica

        preparedStatement.executeUpdate();

    }

    } finally {

        cerrarConexion();

    }

}
```

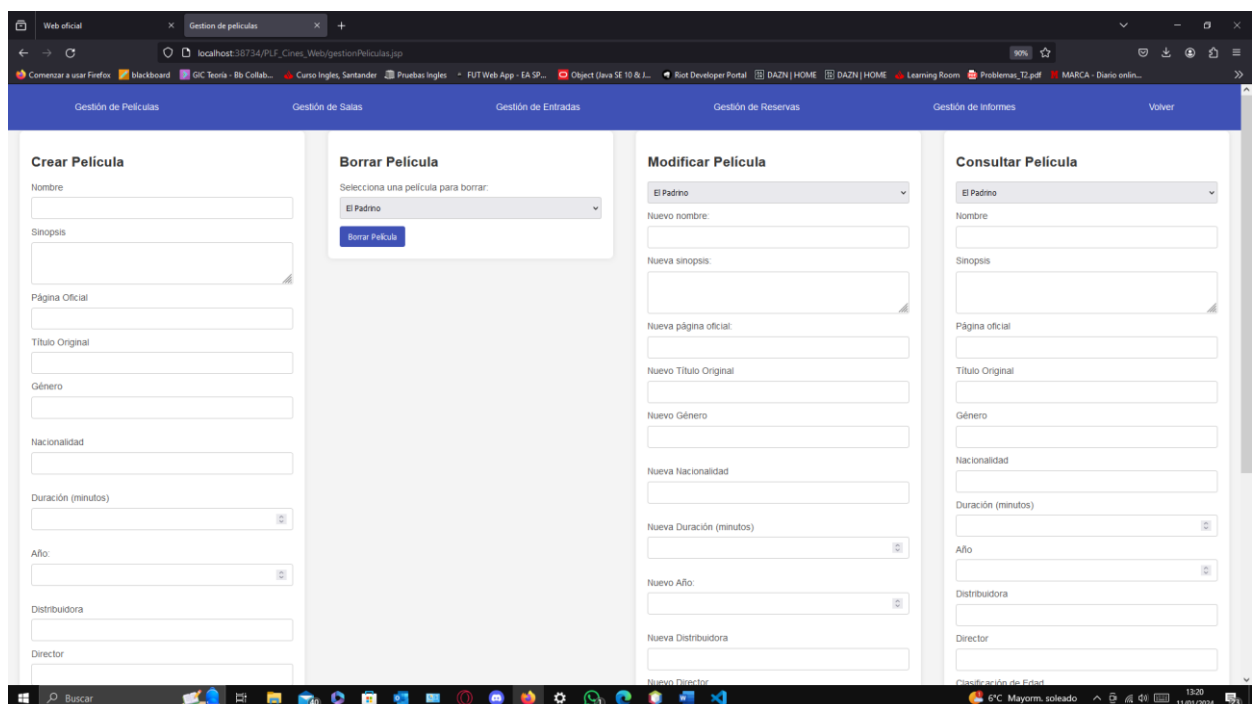
- **guardarPelícula(Película película):** Este método guarda una película en la base de datos. Abre una conexión, prepara una sentencia SQL INSERT, establece los parámetros de la sentencia a las propiedades del objeto película, ejecuta la sentencia y finalmente cierra la conexión.
- **getAllPelículas():** Este método recupera todas las películas de la base de datos. Abre una conexión, prepara una sentencia SQL SELECT, ejecuta la sentencia, itera

sobre el conjunto de resultados para crear una lista de objetos de película, y finalmente cierra la conexión.

- **getPelículaPorNombre(String nombre):** Este método recupera una película de la base de datos por su nombre. Abre una conexión, prepara una sentencia SQL SELECT con una cláusula WHERE, establece el parámetro de la sentencia al nombre proporcionado, ejecuta la sentencia, crea un objeto de película a partir del primer resultado (si lo hay), y finalmente cierra la conexión.
- **borrarPelícula(Película película):** Este método elimina una película de la base de datos. Abre una conexión, prepara una sentencia SQL DELETE, establece el parámetro de la sentencia al nombre del objeto película, ejecuta la sentencia y finalmente cierra la conexión.
- **modificarPelícula(String nombreActual, Película nuevaPelícula):** Este método actualiza una película en la base de datos. Abre una conexión, prepara una sentencia SQL UPDATE, establece los parámetros de la sentencia a las propiedades del nuevo objeto película y al nombre actual, ejecuta la sentencia y finalmente cierra la conexión.

Una vez creado todos los métodos que nos permiten trabajar con el modelo, vamos a pasar a ver la gestión de la vista con el controlador.

Vista:



Para saber que acción estamos llevando a cabo, ya sea crear película, borrarla, modificarla o consultar hemos creado dentro de la clase.jsp unos inputs ocultos que nos permiten saber en cada momento que acción estamos llevando a cabo:

```
<!-- Campos que esta ocultos para saber que accion esta realizando el servlet y no crear un
servlet exclusivo para cada accion de

    boorar de insertar, modificar o mostrar contenido-->

<input type="hidden" name="accion" value="crear">
```

Dentro de cada acción enviamos la información al servlet mediante un doPost

```
<form action="GestionPelicula" method="post">

    <h2>Borrar Película</h2>

    <label>Selecciona una película para borrar:</label>

    <select name="peliculaABorrar">

        <% List<Pelicula> peliculas = new ArrayList<>();

            try {

                peliculas = DatabaseManager.getAllPeliculas();

            } catch (Exception e) {

                e.printStackTrace();

            }

            for (Pelicula pelicula : peliculas) { %>

                <option value="<%= pelicula.getNombre() %>"><%= pelicula.getNombre() %></option>

                <% } %>

            </select><br>

            <!-- Campos que esta ocultos para saber que accion esta realizando el servlet y no
            crear un servlet exclusivo para cada accion de

                boorar de insertar, modificar o mostrar contenido-->

                <input type="hidden" name="accion" value="borrar">

                <button type="submit">Borrar Película</button>

    </form>
```

Ahora vamos a pasar a ver el servlet:

```
@WebServlet("/GestionPelicula")

public class GestionPelicula extends HttpServlet {

    @Override

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
    IOException {

        int duracion = 0;

        int año = 0;

        int clasificacionEdad = 0;

        // Obtén el parámetro de acción desde el formulario

        String accion = request.getParameter("accion");

        // Realiza diferentes acciones según el valor de 'accion'

        if ("crear".equals(accion)) {

            // Obtén los parámetros del formulario de registro

            String nombre = request.getParameter("nombre");

            String sinopsis = request.getParameter("sinopsis");

            String pagina_oficial = request.getParameter("paginaOficial");

            String titulo_original = request.getParameter("tituloOriginal");

            String genero = request.getParameter("genero");

            String nacionalidad = request.getParameter("nacionalidad");

            try {

                duracion = Integer.parseInt(request.getParameter("duracion"));

            } catch (NumberFormatException e) {

                System.out.println(e);

            }

            try {

                año = Integer.parseInt(request.getParameter("anho"));

            }
```

```
    } catch (NumberFormatException e) {

        System.out.println(e);

    }

    String distribuidora = request.getParameter("distribuidora");

    String director = request.getParameter("director");

    try {

        clasificacionEdad = Integer.parseInt(request.getParameter("clasificacionEdad"));

    } catch (NumberFormatException e) {

        System.out.println(e);

    }

    String otrosDatos = request.getParameter("otrosDatos");

    String actores = request.getParameter("actores");

    String url_image = request.getParameter("imagen");

    String url_video = request.getParameter("video");

    try {

        Pelicula pelicula = new Pelicula(nombre, sinopsis, pagina_oficial, titulo_original, genero,
nacionalidad, duracion, año, distribuidora, director, clasificacionEdad, otrosDatos, actores, url_image,
url_video);

        // Guardar el usuario en la base de datos

        DatabaseManager.getInstance().guardarPelicula(pelicula);

        response.sendRedirect("gestionPelículas.jsp"); // Redirigir a la página principal

    } catch (SQLException e) {

        response.getWriter().println("Error al crear la película en el servlet.");

    }

} else if ("borrar".equals(accion)) {

    // Obten el nombre de la película a borrar desde la solicitud

    String nombrePeliculaABorrar = request.getParameter("peliculaABorrar");

    try {

        // Obtén la película por su nombre
```

```
Pelicula pelicula = DatabaseManager.getPeliculaPorNombre(nombrePeliculaABorrar);

if (pelicula != null) {

    // Borra la película de la base de datos

    DatabaseManager.getInstance().borrarPelicula(pelicula);

    response.sendRedirect("gestionPelículas.jsp"); // Redirigir a la página principal

    System.out.println("Pelicula borrada con éxito");

} else {

    response.getWriter().println("No se encontró la película a borrar.");

}

} catch (SQLException e) {

    e.printStackTrace();

    response.getWriter().println("Error al borrar la película.");

}

} else if ("modificar".equals(accion)) {

// Obtén el nombre de la película a modificar desde la solicitud

String nombrePeliculaAModificar = request.getParameter("peliculaAModificar");

try {

    // Obtén la película por su nombre

    Pelicula pelicula =
DatabaseManager.getInstance().getPeliculaPorNombre(nombrePeliculaAModificar);

    System.out.println(nombrePeliculaAModificar);

    if (pelicula != null) {

        // Modifica la película con los nuevos valores

        pelicula.setNombre(request.getParameter("nuevoNombre"));

        pelicula.setSinopsis(request.getParameter("nuevaSinopsis"));

        pelicula.setPaginaOficial(request.getParameter("nuevaPaginaOficial"));

        pelicula.setTituloOriginal(request.getParameter("nuevoTituloOriginal"));

        pelicula.setGenero(request.getParameter("nuevoGenero"));

        pelicula.setNacionalidad(request.getParameter("nuevaNacionalidad"));
```

```
        pelicula.setDuracion(Integer.parseInt(request.getParameter("nuevaDuracion")));

        pelicula.setAño(Integer.parseInt(request.getParameter("nuevoAño")));

        pelicula.setDistribuidora(request.getParameter("nuevaDistribuidora"));

        pelicula.setDirector(request.getParameter("nuevoDirector"));

        pelicula.setClasificacionEdad(Integer.parseInt(request.getParameter("nuevaClasificacionEdad")));

        pelicula.setOtrosDatos(request.getParameter("nuevosDatos"));

        pelicula.setActores(request.getParameter("nuevosActores"));

        pelicula.setUrl_image(request.getParameter("nuevaImagen"));

        pelicula.setUrl_video(request.getParameter("nuevoVideo"));

        // Guarda la película modificada en la base de datos

        DatabaseManager.modificarPelicula(nombrePeliculaAModificar, pelicula);

        response.sendRedirect("gestionPelículas.jsp");

    } else {

        response.getWriter().println("No se encontró la película a modificar.");

    }

} catch (SQLException e) {

    e.printStackTrace();

    response.getWriter().println("Error al modificar la película.");

}

} else if ("Consultar".equals(accion)) {

    String nombrePeliculaAConsultar = request.getParameter("peliculaAConsultar");

    try {

        // Obtén la película por su nombre

        Pelicula pelicula = DatabaseManager.getPeliculaPorNombre(nombrePeliculaAConsultar);

        if (pelicula != null) {

            // Setea los atributos de la película en el request para que puedan ser accesibles en el
```

JSP



```
request.setAttribute("nombreConsultar", pelicula.getNombre());

request.setAttribute("sinopsisConsultar", pelicula.getSinopsis());

request.setAttribute("paginaOficialConsultar", pelicula.getPaginaOficial());

request.setAttribute("tituloOriginalConsultar", pelicula.getTituloOriginal());

request.setAttribute("generoConsultar", pelicula.getGenero());

request.setAttribute("nacionalidadConsultar", pelicula.getNacionalidad());

request.setAttribute("duracionConsultar", pelicula.getDuracion());

request.setAttribute("AnhoConsultar", pelicula.getAño());

request.setAttribute("distribuidoraConsultar", pelicula.getDistribuidora());

request.setAttribute("directorConsultar", pelicula.getDirector());

request.setAttribute("clasificacionEdadConsultar", pelicula.getClasificacionEdad());

request.setAttribute("datosConsultar", pelicula.getOtrosDatos());

request.setAttribute("actoresConsultar", pelicula.getActores());

request.setAttribute("ImagenConsultar", pelicula.getUrl_image());

request.setAttribute("VideoConsultar", pelicula.getUrl_video());


// Redirige a la página del formulario con los campos ya poblados

request.getRequestDispatcher("gestionPelículas.jsp").forward(request, response);

} else {

    response.getWriter().println("No se encontró la película a Consultar.");

}

} catch (SQLException e) {

    e.printStackTrace();

    response.getWriter().println("Error al consultar la película.");

}

} else {

    response.getWriter().println("Acción no reconocida");

}

}
```



```
}  
  
}
```

Este código es un servlet en Java que maneja las solicitudes HTTP POST para la gestión de películas. El servlet se mapea a la URL `"/GestionPelicula"` mediante la anotación `@WebServlet`.

El método `doPost(HttpServletRequest request, HttpServletResponse response)` se sobrescribe para manejar las solicitudes POST. Este método se invoca automáticamente cuando el servidor recibe una solicitud POST para la URL `"/GestionPelicula"`.

Dentro del método `doPost`, se obtiene el parámetro de acción desde la solicitud. Este parámetro determina qué acción se debe realizar: crear, borrar, modificar o consultar una película.

Si la acción es "crear", se obtienen todos los parámetros necesarios para crear una nueva película desde la solicitud. Estos parámetros se utilizan para crear un nuevo objeto `Pelicula`, que se guarda en la base de datos utilizando el método `guardarPelicula` del `DatabaseManager`. Si ocurre un error al guardar la película, se envía un mensaje de error al cliente.

Si la acción es "borrar", se obtiene el nombre de la película a borrar desde la solicitud. Se busca la película en la base de datos utilizando el método `getPeliculaPorNombre` del `DatabaseManager`. Si la película existe, se borra de la base de datos utilizando el método `borrarPelicula` del `DatabaseManager`. Si la película no existe, se envía un mensaje de error al cliente.

Si la acción es "modificar", se obtiene el nombre de la película a modificar desde la solicitud. Se busca la película en la base de datos utilizando el método `getPeliculaPorNombre` del `DatabaseManager`. Si la película existe, se modifican sus atributos con los nuevos valores obtenidos desde la solicitud, y se guarda la película modificada en la base de datos utilizando el método `modificarPelicula` del `DatabaseManager`. Si la película no existe, se envía un mensaje de error al cliente.

Si la acción es "Consultar", se obtiene el nombre de la película a consultar desde la solicitud. Se busca la película en la base de datos utilizando el método `getPeliculaPorNombre` del `DatabaseManager`. Si la película existe, se establecen sus atributos en el objeto `request` para que puedan ser accesibles en la página JSP, y se redirige a la página del formulario con los campos ya rellenos. Si la película no existe, se envía un mensaje de error al cliente.

Si la acción proporcionada en la solicitud no es ninguna de las anteriores, se envía un mensaje de error al cliente diciendo "Acción no reconocida".

De esta forma gestionamos las películas con el patrón MVC, y de la misma forma lo hacemos con las salas, las entradas y las reservas, creamos los 5 métodos en la clase `databaseManager.java`, luego en el `jsp` enviamos los datos al servlet reconociendo la acción que hemos realizado y ya dentro del servlet recogemos y almacenamos esos datos.

Visto el patrón MVC para la gestión del cine, vamos a ver como funciona este patrón para la vista de usuario.

Al igual que con el administrador, la clase que se encarga de gestionar todo el modelo es DatabaseManager.java. Por lo que omitiremos su explicación y veremos el controlador y la vista para generar la experiencia de usuario.

Uso de HTML5, CSS3 y JavaScript.

Uso de HTML5:

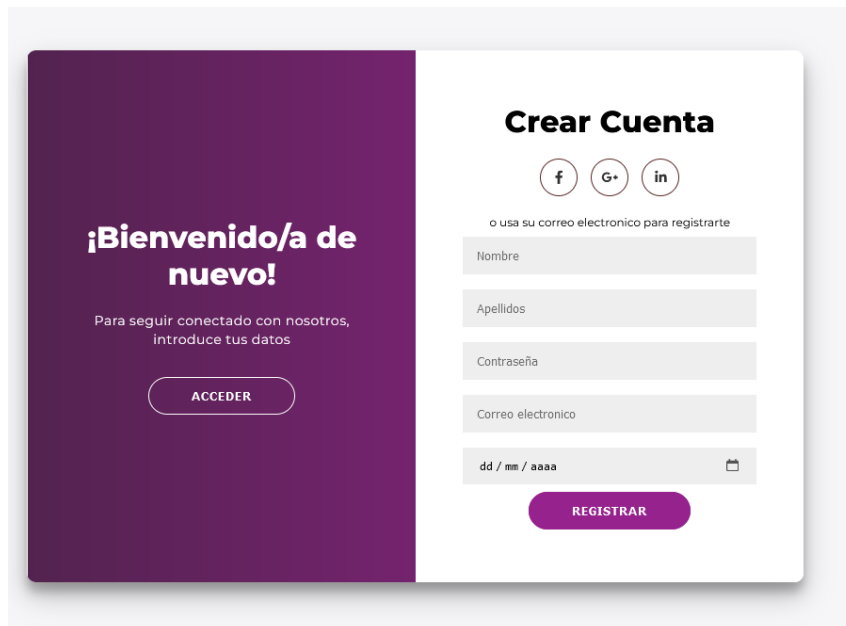
El uso de html en este proyecto ha sido dentro de los archivos.jsp, que permiten la integración de código java, gestión de sesiones y manejos de bases de datos con html5, lo que nos permitía optimizar en gran forma el código.

En este caso el uso ha sido en el apartado de la vista dentro del patrón MVC, tanto para el panel de administrador, como para el registro de los usuarios, como para la propia plataforma de cines y hacer reservas y comentarios.

Pasamos a ver por encima el uso principal del html5, en los diferentes paneles:

Panel de Registro e Inicio de sesión:

El usuario lo que ve es esto:



```
<div class="form-container sign-up-container">
    <form action="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>

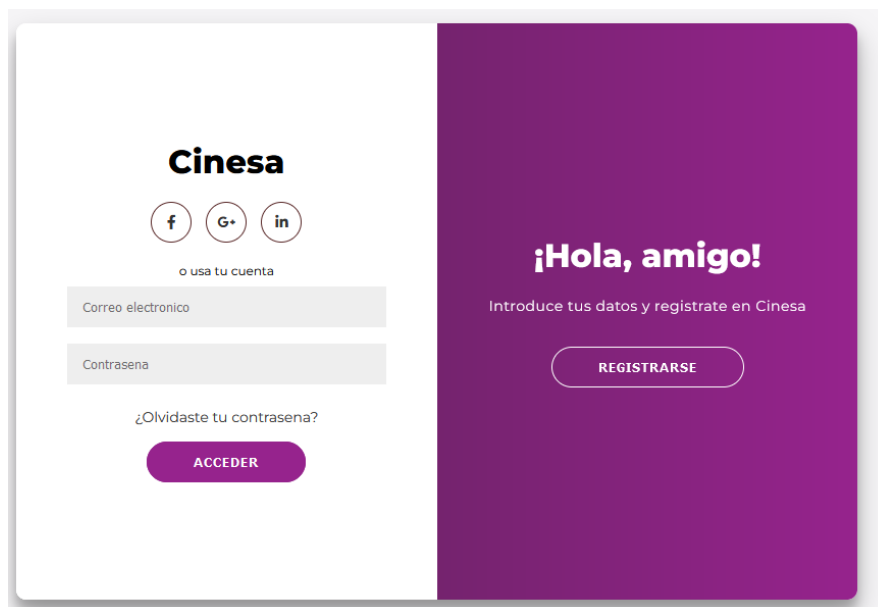
<input type="text" placeholder="Nombre" id="Name" name="Name"/>
<span id="error-nombre" class="error"></span>
<input type="text" placeholder="Apellidos" id="Apellidos" name="Apellidos"/>
<span id="error-apellido" class="error"></span>
<input type="password" placeholder="Contraseña" id="pswd" name="pswd"/>
<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="Fecha-nacimiento" id="Fecha-nacimiento"
name="Fecha-nacimiento"/>
<span id="error-fecha" class="error"></span>
<button type="submit" id="Registrar">Registrar</button>

</form>
</div>

```

Este es un formulario HTML para crear una cuenta de usuario. Cuando el usuario hace clic en el botón "Registrar", los datos del formulario se envían a un servlet llamado "CrearUsuario" utilizando el método POST.

El formulario incluye campos para el nombre, apellidos, contraseña, correo electrónico y fecha de nacimiento del usuario. Cada campo de entrada tiene un elemento `` asociado con un ID específico (por ejemplo, "error-nombre", "error-apellido", etc.) que se puede usar para mostrar mensajes de error relacionados con ese campo.



```

<div class="form-container sign-in-container">

    <form action="AccederUsuario" method="post">

        <h1>Cinesa</h1>

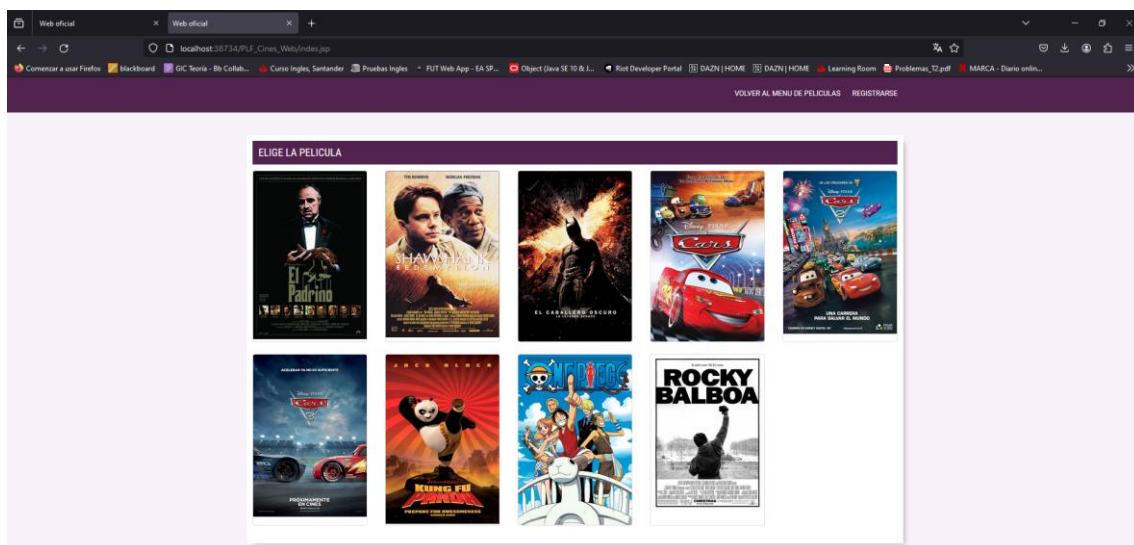
```

```
<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="Contraseña" id = "pswd-2" name="pswd-
2"/>
<span id="error-contrasena-acceso" class="error"></span>
<a href="#">¿Olvidaste tu contraseña?</a>
<button type="submit" id="Acceder">Acceder</button>
</form>
</div>
```

Este es un formulario HTML para iniciar sesión en una cuenta de usuario. Cuando el usuario hace clic en el botón "Acceder", los datos del formulario se envían a un servlet llamado "AccederUsuario" utilizando el método POST.

El formulario incluye campos para el correo electrónico y la contraseña del usuario. Cada campo de entrada tiene un elemento `` asociado con un ID específico (por ejemplo, "error-email-acceso", "error-contrasena-acceso") que se puede usar para mostrar mensajes de error relacionados con ese campo.

Panel de la aplicación principal index.jsp:



```
<div class="container">
    <div class="white-box">
```

```

<h3 class="titulo-cine">ELIGE LA PELICULA</h3>

<div class="row">

    <% // Llamamos al metodo getAllPelículas para iterar sobre todas las películas
        List<Película> listaPelículas DatabaseManager.getInstance().getAllPelículas();
    %>

    <% for(Película película : listaPelículas){ %>
    <div class="col-6 col-sm-4 col-md-3 col-xl-2dot4 mb20">
        <div class="card card-event info-overlay">
            <div class="img has-background">

                <a href="indexDetallado.jsp?id=<%= película.getNombre() %>"
class="event-pop-link">

                    <div class="event-pop-info">
                        <p class="price" style="font-size:1.1em; padding:
5px;"><button type="submit" style="background: none; border: none; padding: 0; color: inherit; cursor:
pointer;"><%= película.getNombre() %></button></p>
                        <span style="font-size: 0.90em; background-color: #53234f;"
class="badge badge-primary">VER FICHA Y PASES</span><br /><br />
                    </div>
                </a>

                <a href="indexDetallado.jsp?id=<%= película.getNombre() %>"></a>

            </div>
        </div>
    </div>
    </div>
    <% } %>
</div>

</div>
</div>

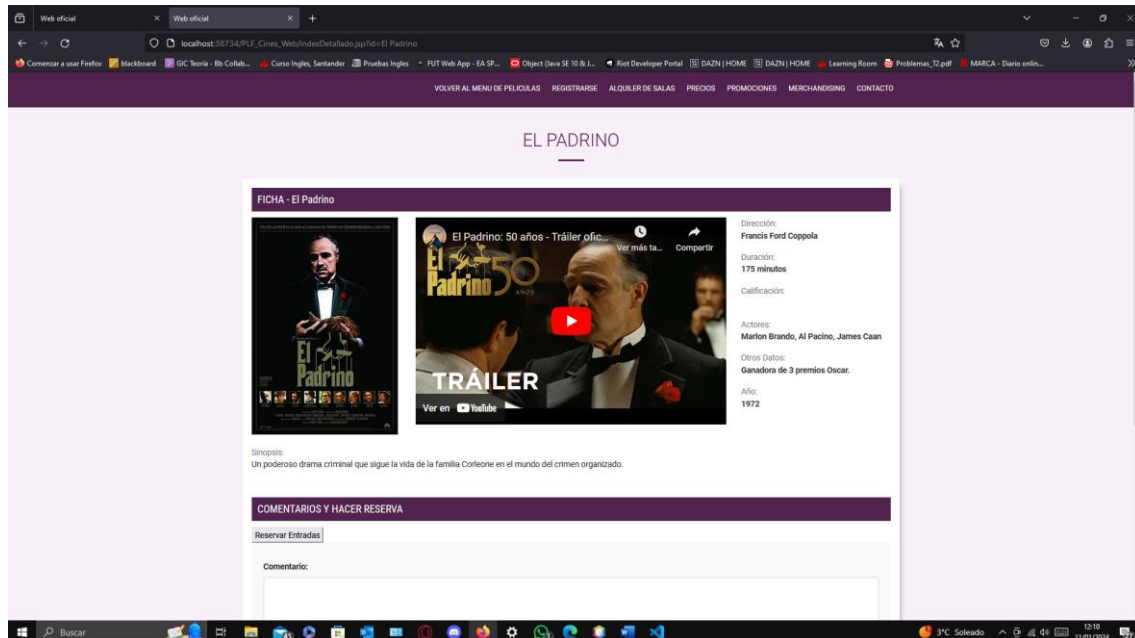
```

Primero, se obtiene una lista de objetos Película llamando al método getAllPelículas() del DatabaseManager.

Luego, se itera sobre esta lista con un bucle for. Para cada película, se genera un bloque de HTML que incluye un enlace a una página de detalles de la película (indexDetallado.jsp), pasando el nombre de la película como un parámetro en la URL.

Dentro de este bloque, se muestra el nombre de la película en un botón y se proporciona un enlace para ver más detalles y pases de la película. También se muestra una imagen de

la película, con la URL de la imagen obtenida llamando al método `getUrl_image()` del objeto `Pelicula`.



Sección dentro de cada película `indexDetallado.jsp`

```
<% String nombrePelicula = request.getParameter("id");
    Pelicula pelicula =
DatabaseManager.getInstance().getPeliculaPorNombre(nombrePelicula);
    session.setAttribute("pelicula",pelicula);
    %>

<% if (pelicula != null) { %>

    <div class="row clearfix">
        <h1 class="text-center title-1"><%= pelicula.getNombre() %></h1>
        <hr class="mx-auto small text-hr" style="margin-bottom: 30px
!important">

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



```
<div class="white-box" style="padding:15px">
    <h3 class="titulo-cine">FICHA - <%= pelicula.getNombre() %></h3>
    <div class="row mb20">
        <div class="col-3 d-none d-md-block">

            <a href="<%= pelicula.getUrl_image() %>" data-
toggle="lightbox"></a>

        </div>
        <div class="col-6 d-none d-md-block"><iframe width="100%"
height="350" src="<%= pelicula.getUrl_video() %>" title="YouTube video player" frameborder="0"
allow="accelerometer; autoplay; clipboard-write; encrypted-media; gyroscope; picture-in-picture;
web-share" allowfullscreen></iframe></div>
        <div class="col-3 d-none d-md-block" style="padding-left: 10px;">
            <ul class="list-unstyled">
                <li><p class="text-muted">Dirección:<br><strong class="text-
dark"> <%= pelicula.getDirector() %></strong></p></li>

                <li><p class="text-muted">Duración:<br><strong class="text-
dark"><%= pelicula.getDuracion() %> minutos</strong></p></li>

                <li><p class="text-muted">Calificación:<br><strong
class="text-dark"><img title="No recomendada para menores de <%= pelicula.getClasificacionEdad()
%>" style="width:20px; margin-right: 5px;"></strong></p></li>

                <li><p class="text-muted">Actores:<br><strong class="text-
dark"><%= pelicula.getActores() %></strong></p></li>

                <li><p class="text-muted">Otros Datos:<br><strong
class="text-dark"><%= pelicula.getOtrosDatos() %></strong></p></li>

                <li><p class="text-muted">Año:<br><strong class="text-
dark"><%= pelicula.getAño() %></strong></p></li>

            </ul>
        </div>
        <div class="col-4 d-md-none">

            <span data-toggle="modal" data-target="#trailer" class="btn btn-
primary" style="padding-top: 10px; margin-top: 10px; display: block; padding-bottom:
10px;">TRAILER <i class="fa fa-play" style="margin-left: 2px;"></i></span>
```

```
</div>

</div>
<div class="row mb30">
  <div class="col-sm-12">
    <span class="text-muted">Sinopsis:</span>
    <p><%= pelicula.getSinopsis() %></p>
  </div>
</div>
```

Primero, se verifica si el objeto película no es nulo. Si no es nulo, se muestra la información de la película.

El nombre de la película se muestra en un encabezado <h1> y también en un encabezado <h3> precedido por la palabra "FICHA".

Se muestra una imagen de la película, con la URL de la imagen obtenida llamando al método getUrl_image() del objeto Pelicula. Esta imagen está vinculada a la URL de la imagen, por lo que al hacer clic en ella se abrirá la imagen en una nueva ventana o pestaña.

También se muestra un video de la película, con la URL del video obtenida llamando al método getUrl_video() del objeto Pelicula.

Se muestra información adicional sobre la película en una lista sin orden, incluyendo el director, la duración, la clasificación de edad, los actores, otros datos y el año.

Finalmente, se muestra la sinopsis de la película, obtenida llamando al método getSinopsis() del objeto Pelicula.

Uso de JavaScript:




Principalmente hemos usado javascript para el chequeo de formularios, tanto el de panel de registro de usuarios como el panel del administrador, para no poder rellenar datos absurdos, pasamos a ver el panel de registro:

¡Bienvenido/a de nuevo!

Para seguir conectado con nosotros, introduce tus datos

ACCEDER

Crear Cuenta

o usa su correo electrónico para registrarte

Nombre

Por favor, ingresa tu nombre

Apellidos

Por favor, ingresa tu apellido

Contraseña

Por favor, ingresa una contraseña

Correo electrónico

Por favor, ingresa un correo electrónico válido

dd / mm / aaaa

Por favor, ingresa una fecha válida en el formato DD-MM-AAAA

REGISTRAR

```
document.addEventListener('DOMContentLoaded', function() {
    document.getElementById('Registrar').addEventListener('click', function(e) {
        var nombre = document.getElementById('Name').value;
        var apellido = document.getElementById('Apellidos').value;
        var emailRegistro = document.getElementById('mail').value;
        var contrasena = document.getElementById('pswd').value;
        var fechaNacimiento = document.getElementById('Fecha-nacimiento').value;
        var emailPattern = /^[^\\s@]+@[^\\s@]+\\.([^\\s@]+)$/;
        var fechaPattern = /^[d{4}-d{2}-d{2}$/; // Define un patrón de fecha (AAAA-MM-DD)

        // Verificar si el nombre está lleno
        if (nombre.trim() === '') {
            document.getElementById('error-nombre').textContent = 'Por favor, ingresa tu nombre';
            e.preventDefault();
        } else {
            document.getElementById('error-nombre').textContent = '';
        }

        // Verificar si el apellido está lleno
        if (apellido.trim() === '') {
            document.getElementById('error-apellido').textContent = 'Por favor, ingresa tu apellido';
            e.preventDefault();
        } 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 válido';
    e.preventDefault();
} else {
    document.getElementById('error-email').textContent = '';
}

// Verificar si la contraseña está lleno
if (contrasena.trim() === '') {
    document.getElementById('error-contrasena').textContent = 'Por favor, ingresa una contraseña';
    e.preventDefault();
} else {
    document.getElementById('error-contrasena').textContent = '';
}

// Verificar si la fecha cumple con el formato esperado
if (!fechaNacimiento.match(fechaPattern)) {
    document.getElementById('error-fecha').textContent = 'Por favor, ingresa una fecha válida en el formato DD-MM-AAAA';
    e.preventDefault();
} else {
    document.getElementById('error-fecha').textContent = '';
}
});

document.getElementById('Acceder').addEventListener('click', function(e) {
    var emailAcceso = document.getElementById('mail-2').value;
    var contrasena = document.getElementById('pswd-2').value;
    var emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    // Verificar si el correo electrónico de acceso es válido
    if (!emailAcceso.match(emailPattern)) {
        document.getElementById('error-email-acceso').textContent = 'Por favor, ingresa un correo electrónico válido';
        e.preventDefault();
    } else {
        document.getElementById('error-email-acceso').textContent = '';
    }
}
```

```
// Verificar si la contraseña está lleno
if (contrasena.trim() === '') {
    document.getElementById('error-contrasena-acceso').textContent = 'Por favor, ingresa una contraseña';
    e.preventDefault();
} else {
    document.getElementById('error-contrasena-acceso').textContent = '';
}
});

document.getElementById('signIn').addEventListener('click', function() {
    document.getElementById('container').classList.remove('right-panel-active');
});

document.getElementById('signUp').addEventListener('click', function() {
    document.getElementById('container').classList.add('right-panel-active');
});
});
```

Se encarga de la validación de los formularios de registro y acceso, y de la interacción con los botones de inicio de sesión y registro.

- Cuando se hace clic en el botón 'Registrar', se recogen los valores de los campos del formulario de registro. Se verifica que cada campo esté lleno y que los campos de correo electrónico y fecha de nacimiento cumplan con los patrones requeridos. Si un campo no cumple con las validaciones, se muestra un mensaje de error y se evita la acción predeterminada del evento de clic (probablemente la presentación del formulario).
- Cuando se hace clic en el botón 'Acceder', se recogen los valores de los campos del formulario de acceso. Se verifica que el correo electrónico y la contraseña estén llenos y que el correo electrónico cumpla con el patrón requerido. Si un campo no cumple con las validaciones, se muestra un mensaje de error y se evita la acción predeterminada del evento de clic.
- Cuando se hace clic en el botón 'signIn', se elimina la clase 'right-panel-active' del elemento con id 'container'. Esto probablemente cambia la vista para mostrar el formulario de inicio de sesión.

- Cuando se hace clic en el botón 'signUp', se agrega la clase 'right-panel-active' al elemento con id 'container'. Esto probablemente cambia la vista para mostrar el formulario de registro.

Panel de guardar películas:

Género

Por favor, ingresa generos validos. Los siguientes generos no son validos:

Nacionalidad

Por favor, ingresa una nacionalidad valida

Duración (minutos)

Año:

Distribuidora

Director

ClasificacionEdad

Por favor, ingresa una clasificacion de edad valida

Otros Datos

```
document.getElementById('botonCrearPelicula').addEventListener('click', function(e) {
    //Cogeremos todos los datos del formulario
    var anho = parseInt(document.getElementById('anho').value, 10);
    var genero = document.getElementById('genero').value.toLowerCase();
    var nacionalidad = document.getElementById('nacionalidad').value;
    var clasificacionEdad = parseInt(document.getElementById('clasificacionEdad').value,
10);

    var duracion = parseInt(document.getElementById('duracion').value, 10);
    var listaGeneros = ['accion', 'animacion', 'aventura', 'ciencia ficcion', 'comedia',
'documental', 'drama', 'fantasia', 'historica', 'musical', 'romantica', 'suspense', 'terror'];
    var listaNacionalidades = ['Espanola', 'Francesa', 'Inglesa', 'Americana', 'Turca',
'Italiana'];
    var listaClasificaciones = [0, 7, 12, 16, 18];

    // Una vez que todos los atributos estén seleccionados, haremos el checking de los
mismos para ver si son validos

    // Verificar si el anno introducido esta entre 1980 y 2024
    if (anho < 1980 || anho > 2024) {
        document.getElementById('mensajeErrorAnho').textContent = 'Por favor, ingresa un
anho entre 1980 y 2024';
        document.getElementById('mensajeErrorAnho').style.color = "red";
        e.preventDefault();
    } else {
```

```
document.getElementById('mensajeErrorAnho').textContent = '';

}

//
//Verificar que los generos introducidos son validos
var generos = genero.split(',').map(g => g.trim());
var generosInvalidos = generos.filter(g => !listaGeneros.includes(g));

if (generosInvalidos.length > 0) {
    //Se verifica que no contiene el genero
    document.getElementById('mensajeErrorGenero').textContent = 'Por favor, ingresa
generos validos. Los siguientes generos no son validos: ' + generosInvalidos.join(', ');
    document.getElementById('mensajeErrorGenero').style.color = "red";
    e.preventDefault();
} else {
    document.getElementById('mensajeErrorGenero').textContent = '';
}

//Verificar si la nacionalidad introducida esta en la lista
if (!listaNacionalidades.includes(nacionalidad)) {
    document.getElementById('mensajeErrorNacionalidad').textContent = 'Por favor,
ingresa una nacionalidad valida';
    document.getElementById('mensajeErrorNacionalidad').style.color = "red";
    e.preventDefault();
} else {
    document.getElementById('mensajeErrorNacionalidad').textContent = '';
}

//Verificar si la clasificacion de edad introducida esta en la lista
if (!listaClasificaciones.includes(clasificacionEdad)) {
    document.getElementById('mensajeErrorClasificacionEdad').textContent = 'Por favor,
ingresa una clasificacion de edad valida';
    document.getElementById('mensajeErrorClasificacionEdad').style.color = "red";
    e.preventDefault();
} else {
    document.getElementById('mensajeErrorClasificacionEdad').textContent = '';
}

//Verificar si la duracion esta entre 0 y 773 minutos
if (duracion < 0 || duracion > 773) {
    document.getElementById('mensajeErrorDuracion').textContent = 'Por favor, ingresa
una duracion entre 0 y 773 minutos';
    document.getElementById('mensajeErrorDuracion').style.color = "red";
    e.preventDefault();
} else {
    document.getElementById('mensajeErrorDuracion').textContent = '';
```

```
}  
});
```

Este script recoge los valores de varios campos de un formulario y realiza una serie de comprobaciones para validar estos valores. Si un valor no cumple con las condiciones de validación, se muestra un mensaje de error y se evita la acción predeterminada del evento de clic (probablemente la presentación del formulario).

Las comprobaciones de validación son las siguientes:

- El año (anho) debe estar entre 1980 y 2024.
- Los géneros (genero) deben estar en la lista de géneros permitidos (listaGeneros).
- La nacionalidad debe estar en la lista de nacionalidades permitidas (listaNacionalidades).
- La clasificación de edad (clasificacionEdad) debe estar en la lista de clasificaciones de edad permitidas (listaClasificaciones).
- La duración debe estar entre 0 y 773 minutos.

Si un valor no cumple con estas condiciones, se muestra un mensaje de error en rojo y se evita la presentación del formulario. Si un valor cumple con las condiciones, se borra cualquier mensaje de error anterior.