IES Francisco de los Ríos

Alumno	**Antonio Delgado Portero**
Asignatura	**Acceso a datos**
Curso	**2 DAM**
Año	**2025-2026**
Título de la práctica	**Proyecto: CineSphere**
GIT	**Enlace**

1. Introducción

La presente memoria técnica documenta el desarrollo de CineSphere, una aplicación de escritorio diseñada para resolver la problemática de la dispersión de contenidos multimedia. El sistema proporciona una herramienta centralizada que permite a los usuarios catalogar, valorar y analizar su consumo de películas de forma eficiente.

A diferencia de las aplicaciones web dependientes de conexión, CineSphere está diseñada para funcionar en entornos híbridos, soportando tanto bases de datos locales (SQLite) para un despliegue rápido y portable, como servidores empresariales (PostgreSQL) para entornos de producción.

El desarrollo ha puesto un énfasis especial en el rendimiento. Dado que los catálogos de cine pueden contener miles de registros, se han implementado técnicas avanzadas de paginación y carga diferida para evitar la saturación de la memoria, un problema común en este tipo de desarrollos.

2. Tecnologías Utilizadas

El ecosistema tecnológico se ha seleccionado buscando un equilibrio entre robustez empresarial y modernidad:

- **Java 21**: Lenguaje base, aprovechando características modernas como los Text Blocks para escribir consultas SQL legibles y mantenibles.
- **JavaFX:** Framework para la interfaz gráfica.
- **AtlantaFX (Tema Dracula):** Librería de estilos que moderniza JavaFX, proporcionando una estética oscura ("Dark Mode") profesional y acorde al contexto cinematográfico.
- **JDBC (Java Database Connectivity):** API estándar para la persistencia de datos. Se ha optado por JDBC puro en lugar de ORMs (como Hibernate) para tener un control absoluto sobre la optimización de las consultas SQL.
- **SGBDs (Sistemas Gestores de Bases de Datos):** PostgreSQL y SQLite.
- **Maven:** Gestión del ciclo de vida del proyecto y dependencias.
- **jBCrypt:** Biblioteca de seguridad para el hashing de contraseñas.
- **Apache Commons CSV:** Para la lógica de importación masiva de datos (ETL).

3. Arquitectura del Sistema

El proyecto sigue estrictamente el patrón Modelo-Vista-Controlador (MVC), lo que desacopla la lógica de negocio de la interfaz de usuario, facilitando el mantenimiento y la escalabilidad.

3.1. Detalle: Infraestructura de Datos y Utilidades

Dentro de la arquitectura, existen componentes transversales críticos para el funcionamiento del sistema:

Gestión de Conexiones (Conexion.java)

Esta clase implementa el patrón Singleton para garantizar una única instancia del gestor de conexiones.

- **Configuración vía .properties:** Se ha decidido utilizar archivos .properties (config-sqlite.properties) en lugar de XML para la configuración de la base de datos.

  Los archivos de propiedades (clave-valor) son el estándar en Java para configuraciones planas. Son más ligeros de procesar que un XML (que requiere un parser DOM/SAX) y menos propensos a errores de sintaxis humana, lo que facilita el cambio rápido entre entornos de desarrollo y producción.

```
// Carga dinámica de propiedades según el entorno seleccionado
Properties properties = new Properties();
properties.load(getClass().getResourceAsStream("/config/" +
configFile));

String url = properties.getProperty("db.url");
// ... lectura de credenciales ...

// Detección de motor para configuraciones específicas
if (url.startsWith("jdbc:sqlite")) {
    // Lógica específica para SQLite: activar Foreign Keys manualmente
```

```
    connection = DriverManager.getConnection(url);
    try (Statement st = connection.createStatement()) {
        st.execute("PRAGMA foreign_keys = ON");
    }
} else {
    // Conexión estándar PostgreSQL
    connection = DriverManager.getConnection(url, user, password);
}
```

Inicialización del Esquema y Abstracción DDL(DataBaseSchema.java)

La clase **DatabaseSchema** es la encargada de garantizar que la estructura de la base de datos (tablas, claves primarias y foráneas) exista antes de que la aplicación intente realizar cualquier operación. Implementa una estrategia en la que **el código define la base de datos** y **cero configuración**, permitiendo que la aplicación se despliegue en un entorno nuevo sin necesidad de ejecutar scripts SQL manuales externamente.

Abstracción de Dialectos SQL

Uno de los mayores desafíos al soportar múltiples motores de base de datos (Polimorfismo de SGBD) es que el estándar SQL tiene variaciones, especialmente en la definición de datos (DDL).

- **El Problema:** PostgreSQL utiliza el tipo serial SERIAL PRIMARY KEY para columnas autoincrementales, mientras que SQLite utiliza INTEGER PRIMARY KEY AUTOINCREMENT. Una sentencia estática fallaría en uno de los dos motores.
- **La Solución:** La clase detecta en tiempo de ejecución qué motor está activo y construye las sentencias SQL dinámicamente inyectando la sintaxis correcta.

```
public static void inicializar() {
    // Detección del motor a través de la instancia de conexión
    boolean isSQLite = Conexion.getInstance().isSQLite();

    // Definición dinámica del tipo de dato para IDs autoincrementales
    String AUTO_INCREMENT = isSQLite ?
        "INTEGER PRIMARY KEY AUTOINCREMENT" :
        "SERIAL PRIMARY KEY";
```

```
    // Inyección de la sintaxis correcta en la sentencia CREATE TABLE
    stmt.executeUpdate("""
        CREATE TABLE IF NOT EXISTS usuario (
            idusuario %s,  -- Aquí se inyecta la variante correcta
            nombreusuario VARCHAR(100) UNIQUE NOT NULL,
            email VARCHAR(150) NOT NULL,
            passw VARCHAR(255) NOT NULL,
            borndate DATE,
            rol VARCHAR(20) DEFAULT 'USER'
        )
    """.formatted(AUTO_INCREMENT));

    // ... creación del resto de tablas ...
}
```

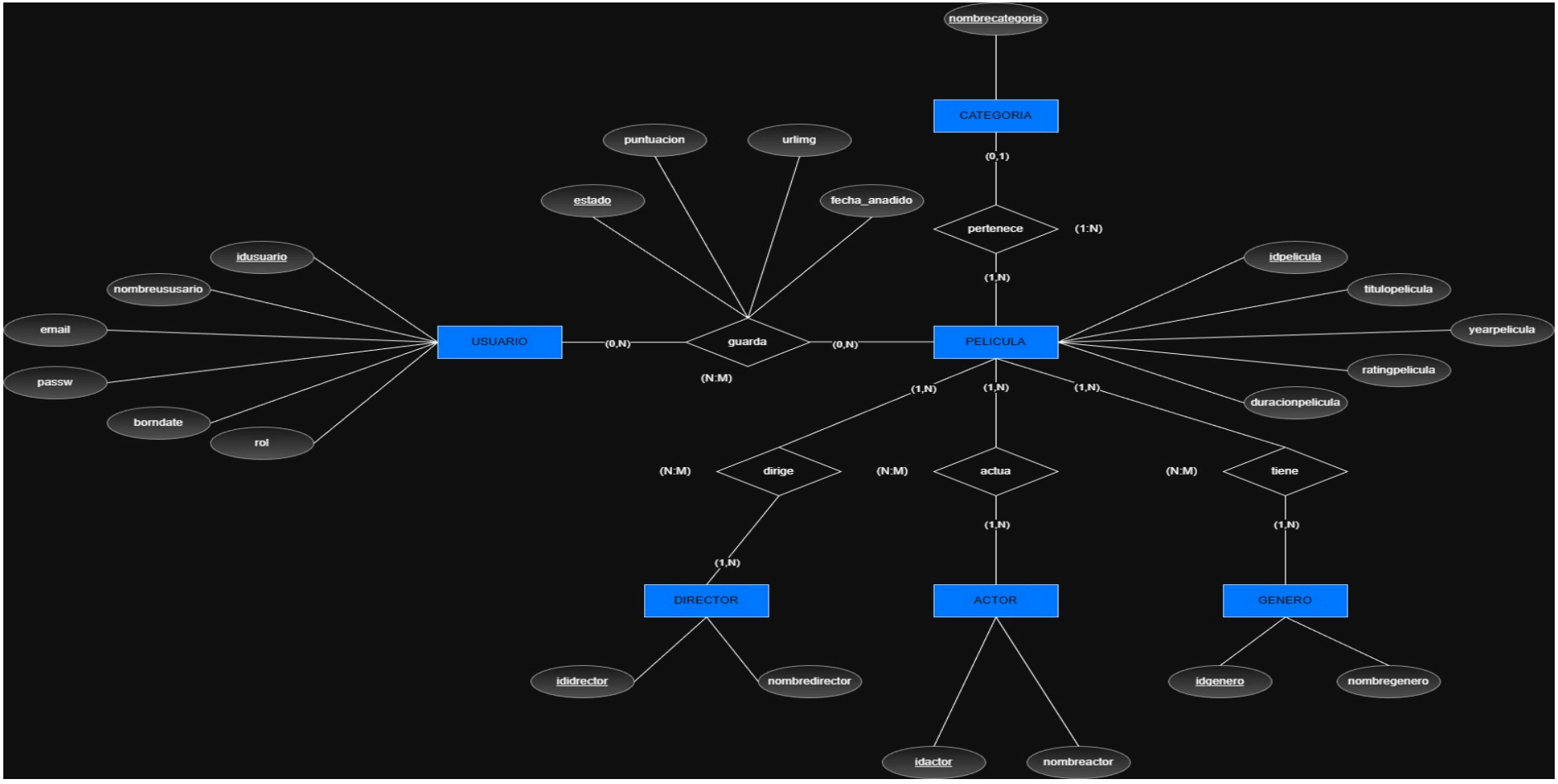
Motor de Importación (CsvImporter.java)

Este componente implementa la lógica de Extracción, Transformación y Carga.

- **Librería Especializada (Apache Commons CSV):** Se ha optado por esta librería para delegar el parsing de bajo nivel. Esto es crítico porque permite manejar correctamente registros complejos (como títulos de películas que contienen comas o comillas) que romperían una implementación ingenua basada en String.split(). Además, permite el mapeo de columnas por nombre, desacoplando el código del orden físico de las columnas en el archivo.
- **Normalización en Vuelo:** Lee datos desnormalizados y los distribuye en tablas relacionales (Actor, Director, Genero).
- **Caché en Memoria:** Para optimizar el rendimiento, mantiene mapas en memoria (HashMap<String, Actor>) durante la ejecución para evitar miles de consultas SELECT redundantes a la base de datos al verificar si un actor ya existe.

4. Modelo de Datos

El modelo relacional se ha diseñado en Tercera Forma Normal para garantizar la integridad de los datos y evitar redundancias.



- **Entidades Fuertes:** Usuario (credenciales y rol) y Pelicula (datos técnicos).
- **Entidades de Catálogo:** Actor, Director, Genero y Clasificacion.
- **Tablas Intermedias (N:M):** peliculaactor, peliculagenero, peliculadirector. Estas tablas rompen la relación muchos a muchos, permitiendo que una película tenga varios actores y viceversa.
- **Entidad Asociativa (MiLista):** Tabla que relaciona Usuario y Pelicula, pero añade atributos propios de esa relación: estado (Pendiente, Viendo, Terminada), puntuacion (0-10) y fecha_anadi

4.1. Puntos de Impedancia Objeto-Relación

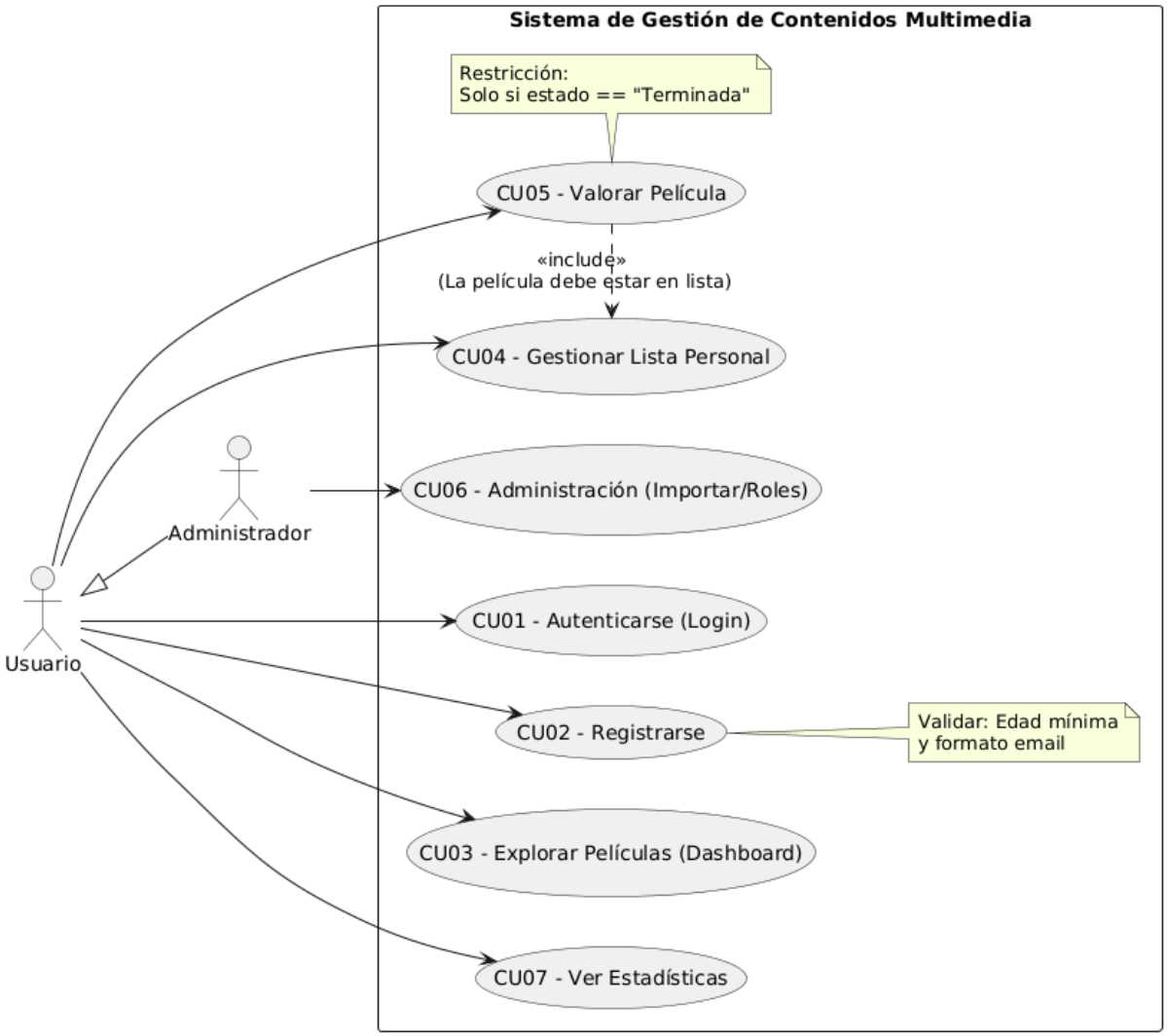
Durante el desarrollo de CineSphere, se ha abordado explícitamente el problema de la Impedancia Objeto-Relación, derivado de las diferencias estructurales y conceptuales entre el paradigma Orientado a Objetos (Java) y el Modelo Relacional (SQL). Al renunciar al uso de ORMs automáticos (como Hibernate) en favor de JDBC puro, se han resuelto manualmente los siguientes puntos de fricción:

- **Diferencia de Granularidad y Estructura:** Mientras que en Java una entidad Pelicula es un objeto complejo que contiene listas de otros objetos (List<Actor>, List<Genero>), en la base de datos esta información está atomizada y dispersa en múltiples tablas normalizadas (pelicula, actor, peliculaactor). La capa DAO reconstruye esta granularidad mediante consultas JOIN y lógica de agregación en memoria.
- **Discordancia de Tipos de Datos:** Se ha implementado un mapeo estricto entre los tipos nativos de los SGBD y los tipos fuertes de Java, prestando especial atención a la conversión entre java.sql.Date / TIMESTAMP y la API moderna java.time.LocalDate utilizada en el modelo de dominio.
- **Asociatividad y Navegación:** El modelo relacional utiliza claves foráneas (FK) para establecer relaciones, las cuales no son navegables directamente. En la aplicación, estas relaciones se han transformado en referencias en memoria (listas y objetos anidados), permitiendo navegar el grafo de objetos (ej. pelicula.getActores()) sin necesidad de realizar consultas SQL redundantes tras la carga inicial.

5. Casos de Uso

La aplicación cubre el ciclo de vida completo de la gestión de contenidos:

- **CU01 - Autenticación:** Login seguro con soporte para múltiples motores de BBDD.
- **CU02 - Registro**: Creación de usuarios con validación de edad mínima y formato de correo.
- **CU03 - Exploración (Dashboard):** Visualización de películas con paginación y filtros dinámicos (Año, Rating, Género).
- **CU04 - Gestión Personal**: Añadir películas a "Mi Lista", cambiar su estado visual y eliminarlas.
- **CU05 - Valoración:** Puntuar películas (solo disponible si el estado es "Terminada").
- **CU06 - Administración:** Importar catálogos CSV y gestionar roles de usuarios (funcionalidad exclusiva del rol ADMIN).
- **CU07 - Análisis:** Visualización de estadísticas gráficas sobre hábitos de consumo.



6. Interfaces de Usuario (GUI)

La interfaz gráfica se ha desarrollado utilizando FXML para la estructura y CSS para el estilizado, priorizando la usabilidad y la experiencia de usuario (UX). La arquitectura de la vista se organiza de la siguiente manera:

- **Layout Principal (Shell):** Implementado sobre un BorderPane (en main.fxml). Cuenta con una barra lateral estática (VBox) para la navegación persistente y un StackPane central para la carga dinámica de vistas, asegurando transiciones fluidas sin recargar la ventana completa.
- **Autenticación (Login y Registro):**
    - Diseño encapsulado en un StackPane que centraliza el formulario.
    - Gestión de Conexión: Separa la lógica de conexión a la base de datos (ComboBox para elegir motor) de la autenticación de usuario, inhabilitando los campos de credenciales hasta que se establece una conexión exitosa con el servidor .
    - Incluye navegación fluida hacia la vista de registro (register.fxml), que incorpora controles específicos como DatePicker para la validación de edad.
- **Explorador y Mi Lista (Dashboard):**
    - Utiliza un diseño responsivo basado en TilePane dentro de un ScrollPane, permitiendo que las tarjetas de las películas se reorganicen automáticamente según el tamaño de la ventana.
    - Barra de Herramientas: Integra una barra superior completa con TextField para búsquedas y múltiples ComboBox para el filtrado simultáneo por año, rating y género .
    - Paginación Personalizada: Se ha implementado una barra de controles inferior (HBox) con botones de navegación (First, Prev, Next, Last) en lugar del paginador por defecto de JavaFX, para tener control total sobre las consultas SQL paginadas .
- **Detalle de la Película:**
    - Diseñada con un GridPane asimétrico para destacar el póster frente a la información textual .
    - Contenido Dinámico: Uso extensivo de FlowPane para mostrar listas de longitud variable (géneros, directores y actores) mediante etiquetas tipo "chip", asegurando que el contenido no se desborde .
    - Contexto de Usuario: Controles interactivos para gestión de estado (ComboBox de estado y puntuación) y visibilidad condicional de botones administrativos (ej. botón "Eliminar" solo visible para admins).

- **Estadísticas (Dashboard Analítico):**
    - Estructura basada en tarjetas (VBox con estilo card) para mostrar KPIs clave: Total guardadas, Películas vistas y Tiempo total consumido .
    - Visualización de datos mediante gráficos nativos de JavaFX: PieChart para la distribución por estado y BarChart con ejes categorizados para el análisis de géneros favoritos .
- **Formulario de Gestión (Nueva Película):**
    - Interfaz dedicada (pelicula_form.fxml) para la creación manual de registros. Organiza los campos mediante un GridPane para datos técnicos (Año, Duración, Rating) y TextArea para la entrada de listas separadas por comas (Actores, Directores) .
- **Panel de Ajustes:**
    - Organizado mediante un TabPane que segrega claramente las responsabilidades:
        1. **Perfil:** Gestión de credenciales y "Zona de Peligro" para el borrado de cuenta.
        2. **Importación:** Interfaz para la carga masiva de datos (CSV local o Datasets predefinidos) y acceso al formulario de creación manual .
        3. **Gestión de Usuarios (Admin):** Visualización tabular (TableView) de todos los usuarios registrados con controles directos para la promoción de roles (Ascender/Degradar) y eliminación de usuarios .

7. Controladores

Los controladores actúan como el cerebro de cada vista, orquestando la comunicación entre la capa de presentación (FXML) y la lógica de datos (DAOs):

- **LoginController:** Gestiona el punto de entrada a la aplicación. Su responsabilidad crítica es la selección dinámica del motor de base de datos (PostgreSQL o SQLite) antes de permitir la autenticación . Utiliza BCrypt para la verificación segura de credenciales y establece la sesión global mediante SessionManager.
- **RegisterController:** Maneja el alta de nuevos usuarios implementando reglas de negocio estrictas: validación de formato de email mediante RegEx y verificación de edad mínima (14 años) utilizando DatePicker y lógica de fechas (LocalDate) .

- **MainController**: Estructura la navegación principal. Implementa un sistema de carga dinámica de vistas sobre un StackPane central, permitiendo cambiar entre pantallas (Lista, Estadísticas, Ajustes) sin cerrar la ventana principal ni perder el contexto de la aplicación .
- **PeliculaListaController**: Es el controlador más complejo en cuanto a visualización. Implementa un algoritmo de paginación responsiva (ajustarPageSize) que escucha los cambios en el ancho de la ventana para recalcular dinámicamente cuántas tarjetas caben por fila, optimizando las llamadas SQL (LIMIT/OFFSET) para solicitar solo los registros visibles .
- **MiListaController**: Especialización de la lista de películas filtrada por la relación usuario-película. Reutiliza la lógica de paginación y filtrado, permitiendo al usuario gestionar sus estados de visualización ("Pendiente", "Viendo", "Terminada") desde una vista centralizada .
- **PeliculaDetalleController**: Maneja la lógica de negocio de la relación individual usuario-película.
    - **Lógica de Interfaz:** Habilita o deshabilita el selector de puntuación dinámicamente; solo permite puntuar si el estado es "Terminada" .
    - **Interactividad:** Genera dinámicamente componentes visuales ("Chips") para géneros y actores, y permite la apertura del tráiler en el navegador externo del sistema.
- **EstadisticasController**: Recopila métricas de negocio en tiempo real. Transforma los datos crudos del DAO en información visual mediante gráficos de JavaFX (PieChart para estados y BarChart para géneros favoritos), calculando KPIs como el tiempo total de visualización .
- **PeliculaFormController**: Gestiona la creación de contenidos bajo un estricto control transaccional (ACID). Al guardar una película, orquesta la inserción en múltiples tablas (película, relaciones con actores, directores y géneros) dentro de una única transacción, realizando un rollback automático si falla cualquier paso para evitar inconsistencias en la base de datos.
- **SettingsController**: Controlador administrativo modularizado mediante pestañas (TabPane):
    - **Gestión de Perfil:** Permite la actualización de credenciales y la eliminación de cuenta ("Zona de peligro").
    - **Administración (Solo Admins):** Facilita la importación masiva de datos (ETL desde CSV) y la gestión de roles de usuarios (ascenso/degradación) mediante una TableView interactiva .

8. DAO (Data Access Object) y Optimización

Esta capa constituye el núcleo técnico de la persistencia de datos. Se ha diseñado para abstraer la complejidad del acceso a datos, garantizando un alto rendimiento y la integridad referencial sin depender de frameworks ORM pesados. A continuación, se detallan las técnicas de ingeniería de software aplicadas:

8.1. Ingeniería de Rendimiento: Paginación y Carga por Lotes

Uno de los desafíos críticos en aplicaciones de gestión de catálogos es la saturación de memoria al manipular miles de registros. Para resolverlo, se han implementado dos estrategias clave en PeliculaDAO:

a. **Paginación en Servidor (LIMIT/OFFSET)** En lugar de recuperar todo el catálogo a la memoria de la JVM (lo cual sería ineficiente), se envían sentencias SQL que solicitan exclusivamente la "página" actual demandada por la interfaz .

```
// PeliculaDAO.java
private static final String SQL_FIND_PAGE = """
    SELECT idpelicula, titulopelicula, yearpelicula, ratingpelicula,
           duracionpelicula, nombreclasificacion
    FROM pelicula
    ORDER BY idpelicula
    LIMIT ? OFFSET ?
""";
```

b. **Solución al Problema N+1 (Batch Fetching)** Al mostrar un listado de películas, es necesario obtener sus géneros. La aproximación ingenua realizaría 1 consulta para obtener la lista y N consultas adicionales (una por película) para sus géneros. Para evitar este cuello de botella, se ha implementado el método cargarGenerosEnLote. Este extrae los IDs de las películas cargadas en memoria y ejecuta una **única consulta adicional** utilizando la cláusula IN (...), reduciendo drásticamente el tráfico de red .

```java
// PeliculaDAO.java - Optimización N+1
public void cargarGenerosEnLote(List<Pelicula> peliculas) {
    if (peliculas.isEmpty()) return;
    // 1. Recolección de IDs
    List<Integer> ids =
peliculas.stream().map(Pelicula::getIdPelicula).toList();

    // 2. Generación dinámica de la cláusula IN (?,?,?...)
    String placeholders = String.join(",",
Collections.nCopies(ids.size(), "?"));
    String sql = String.format(SQL_FIND_GENEROS_LOTE, placeholders);

    // 3. Ejecución de una ÚNICA consulta agregada
    try (PreparedStatement ps = conn.prepareStatement(sql)) {
        // ... asignación y mapeo en memoria ...
    }
}
```

8.2. Gestión de Transacciones (ACID)

Para garantizar la integridad de los datos, las operaciones de escritura que involucran múltiples tablas se manejan bajo transacciones atómicas. Un ejemplo crítico se encuentra en PeliculaFormController. Al guardar una película, se deben insertar registros en la tabla pelicula y posteriormente en las tablas intermedias (peliculaactor, peliculadirector). Se deshabilita el AutoCommit para permitir un ROLLBACK completo si falla alguna inserción, evitando datos inconsistentes o "huérfanos" .

```java
// PeliculaFormController.java
try {
    conn.setAutoCommit(false); // 1. Inicio de Transacción (Atomicidad)

    peliculaDAO.insert(p); // Insertar entidad padre

    // Insertar relaciones dependientes
```

```
    guardarGeneros(p, txtGeneros.getText());
    guardarDirectores(p, txtDirectores.getText());
    guardarActores(p, txtActores.getText());

    conn.commit(); // 2. Confirmación (Durabilidad)
} catch (Exception e) {
    conn.rollback(); // 3. Reversión en caso de error
    logger.log(Level.SEVERE, "Transacción fallida, rollback ejecutado",
e);
} finally {
    conn.setAutoCommit(true);
}
```

8.3. Construcción Dinámica de Consultas (Dynamic SQL)

El sistema permite filtrar por múltiples criterios simultáneos (Año, Rating, Género, Título). Escribir una consulta estática para cada combinación posible sería inmanejable. Se ha implementado un constructor de consultas dinámico en PeliculaDAO (construirCondicionesFiltro). Este evalúa en tiempo de ejecución qué filtros están activos y concatena las cláusulas AND necesarias, gestionando paralelamente una lista de parámetros para inyectarlos de forma segura en el PreparedStatement .

8.4. Estrategias de Carga: Lazy vs. Eager Loading

Para optimizar el uso de recursos, se han diferenciado dos estrategias de recuperación de objetos:

- **Carga Perezosa (Lazy Loading):** Implementada en findByIdLazy. Recupera únicamente los datos primitivos de la película, sin realizar JOIN con las tablas de relaciones. Se utiliza en los listados generales para maximizar la velocidad .
- **Carga Ansiosa (Eager Loading):** Implementada en findByIdEager. Reutiliza la carga perezosa e invoca a los DAOs auxiliares (PeliculaActorDAO, etc.) para "hidratar" el objeto completo. Se utiliza exclusivamente en la vista de detalle donde se requiere toda la información .

8.5. Consultas Analíticas y Agregación

Para el módulo de estadísticas, se ha evitado traer todos los registros a la memoria de Java para procesarlos. En su lugar, se delega el cálculo al motor de base de datos mediante funciones de

agregación (COUNT, SUM, GROUP BY) en MiListaDAO. Esto permite generar gráficos en tiempo real con un coste computacional mínimo para la aplicación .

8.6. Gestión de Identidad (Generated Keys)

En las operaciones de inserción, se utiliza la característica Statement.RETURN_GENERATED_KEYS de JDBC. Esto permite recuperar el ID autogenerado por la base de datos (independientemente de si es SERIAL en PostgreSQL o AUTOINCREMENT en SQLite) dentro de la misma transacción, actualizando el objeto Java inmediatamente sin necesidad de consultas adicionales .

9. Validaciones y Control de Errores

- **Validación de Datos:** Antes de contactar con la base de datos, se validan formatos (email, fechas no futuras, campos vacíos) para reducir el tráfico innecesario.
- **Integridad Referencial:** Se utiliza el borrado en cascada (ON DELETE CASCADE) en la definición de las tablas. Si se elimina un usuario, el sistema de base de datos se encarga automáticamente de limpiar su lista de películas, garantizando la consistencia sin código Java adicional.
- **Feedback al Usuario:** Los errores técnicos (excepciones SQL) se capturan y se traducen en mensajes amigables mediante la clase AlertUtils, mientras que el error técnico real se vuelca al log para depuración.

10. Instalación y Despliegue

La aplicación se distribuye como un proyecto Maven.

1. **Requisitos:** Tener instalado JDK 21 o superior.
2. **Configuración:**
    a. **Para SQLite:** No requiere acción. La aplicación creará el archivo database/cinesphere.db automáticamente en la primera ejecución.
    b. **Para PostgreSQL:** Se debe configurar el archivo src/main/resources/config/config-postgres.properties con las credenciales del servidor local.
3. **Ejecución:** Ejecutar la clase Main.java desde el IDE o generar el JAR mediante mvn package.

11. Lista de Comprobación de Requisitos Técnicos

[x] Conexión JDBC: Implementada con patrón Singleton.

[x] Soporte Multi-BBDD: Funcional en SQLite y PostgreSQL.

[x] Estructura Relacional: Tablas normalizadas y relaciones N:M implementadas.

[x] CRUD Completo: Implementado para usuarios y películas.

[x] Transacciones: Uso de commit y rollback en operaciones complejas.

[x] Consultas Avanzadas: Uso de JOIN, GROUP BY y funciones de agregación para estadísticas.

[x] Optimización: Paginación en servidor y carga por lotes (Batch fetching).

[x] Arquitectura: MVC estricto con separación de capas.

[x] Interfaz: JavaFX con diseño responsive y CSS avanzado.

12. Patrones de Diseño y Componentes

Se han aplicado patrones de ingeniería de software para asegurar la calidad del código:

- **MVC:** Estructura troncal de la aplicación.
- **Singleton:** Aplicado en Conexion y SessionManager para recursos únicos compartidos.
- **DAO (Data Access Object):** Para abstraer y encapsular el acceso a datos.
- **Factory:** Utilizado implícitamente por JavaFX al crear las celdas de la lista.

13. Futuras Mejoras

El proyecto sienta una base sólida escalable con las siguientes líneas de evolución:

- **API REST:** Migrar el backend a una API REST (Spring Boot) para permitir clientes web y móviles.
- **Sistema Social:** Permitir seguir a otros usuarios y ver sus listas ("Amigos").
- **Exportación PDF:** Generar informes de "Mi Lista" en formato PDF imprimible.

14. Ayuda IA y Herramientas Externas

Se ha utilizado Inteligencia Artificial como herramienta de apoyo a la programación para:

- Generar los datos semilla (datasets CSV) para pruebas masivas.
- Optimizar la sintaxis de las consultas SQL complejas para la lógica de paginación y carga por lote.
- Diseñar la paleta de colores del archivo CSS para asegurar contraste y accesibilidad en el tema oscuro.
- Creación de componentes visuales como clases.

15. Conclusión

CineSphere cumple satisfactoriamente con todos los objetivos planteados. No es solo un gestor de bases de datos, sino una aplicación completa que demuestra un dominio de la arquitectura de software.

La implementación técnica destaca por su eficiencia (resolviendo problemas reales de saturación de memoria) y su robustez (manejo de transacciones y errores). La capacidad de operar indistintamente con bases de datos locales o servidores remotos, junto con una interfaz de usuario pulida, la convierten en un producto de software de alta calidad académica y profesional.