INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

ING. DE SOFTWARE

PRÁCTICA 3

MÓDELO DE DISEÑO

NOMBRE:

PARDO GÓMEZ ISAAC

PROFESOR:

Gabriel Hurtado Avilés

GRUPO:

6CV3

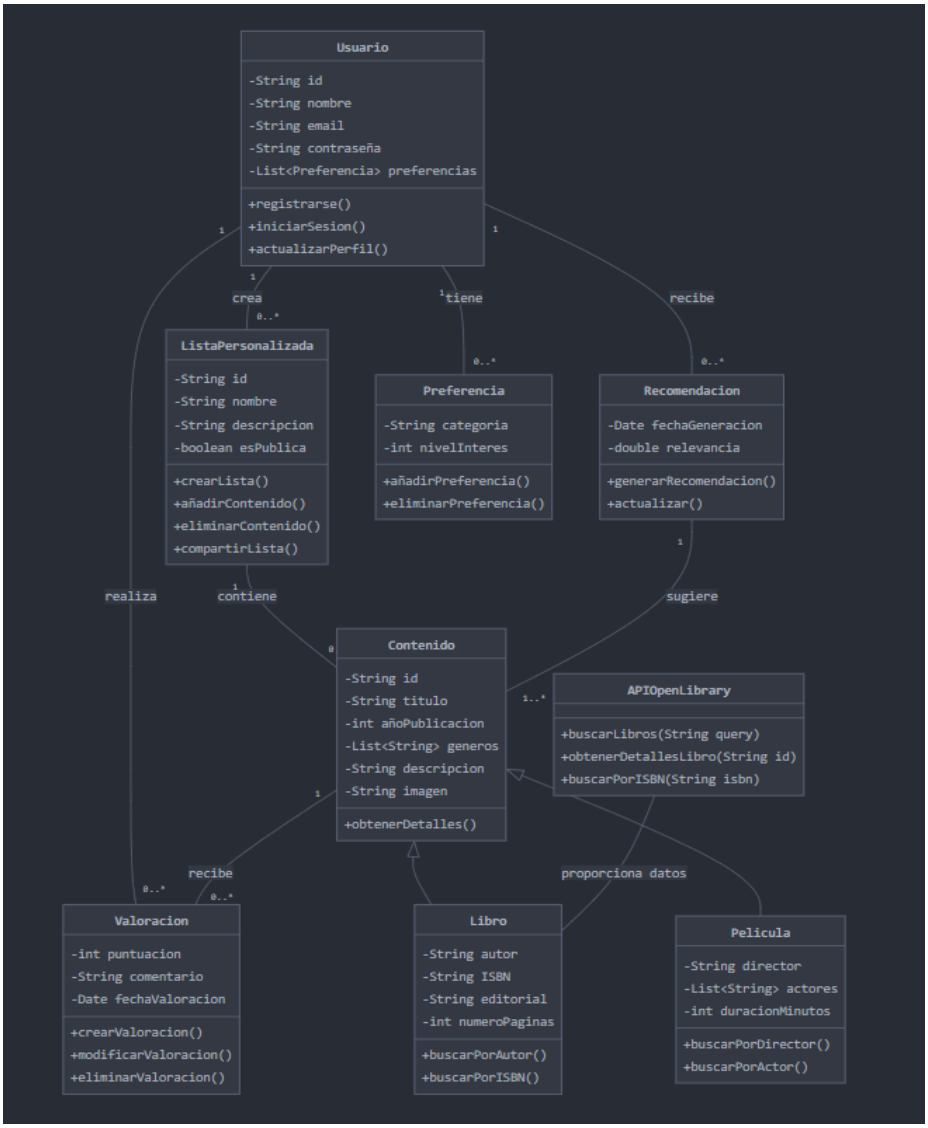
FECHA: 28/04/25

Contenido

1. Correspondencia entre modelos de análisis, diseño e implementación

1.1 Modelo de Dominio y Diagrama de Clases Conceptuales

El modelo de dominio del Sistema de Recomendación de Libros y Películas (SRLP) se basa en el análisis de los requerimientos funcionales identificados, estableciendo una clara correspondencia entre las entidades del negocio y su implementación en el sistema.



1.2. Reglas de negocio e invariantes.

Reglas de Negocio

Usuario

1. Un usuario debe registrarse con un email único en el sistema.

2. Las contraseñas deben almacenarse de forma encriptada.
3. Un usuario puede tener múltiples listas personalizadas.
4. Un usuario puede valorar cada contenido una sola vez, pero puede modificar su valoración.

Contenido (Libros y Películas)

1. Todo contenido debe tener al menos un género asociado.
2. Los libros deben tener un ISBN único cuando esté disponible.
3. El año de publicación no puede ser posterior al año actual.
4. La descripción debe tener un mínimo de 50 caracteres.

Valoraciones

1. La puntuación debe estar en un rango de 1 a 5 estrellas.
2. Los comentarios opcionales no pueden exceder los 1000 caracteres.
3. Solo usuarios registrados pueden realizar valoraciones.

Listas Personalizadas

1. Una lista debe tener un nombre único para cada usuario.
2. Una lista puede ser pública (visible para todos) o privada (visible solo para el usuario).
3. Una lista puede contener tanto libros como películas.

Recomendaciones

1. Las recomendaciones se generan basadas en preferencias del usuario y valoraciones previas.
2. Las recomendaciones se actualizan cuando el usuario modifica sus preferencias o realiza nuevas valoraciones.
3. El algoritmo prioriza contenido similar a los mejor valorados por el usuario.

API Open Library

1. Todas las búsquedas a la API deben ser cacheadas para reducir llamadas repetidas.
2. Se debe implementar manejo de errores para cuando la API no esté disponible.

Invariantes del Sistema

1. Integridad de Datos: Todo contenido en el sistema debe tener un título, año y al menos un género.
2. Unicidad de Usuarios: No pueden existir dos usuarios con el mismo email.
3. Unicidad de Valoraciones: Un usuario solo puede tener una valoración activa por cada contenido.
4. Consistencia de Puntajes: Todas las valoraciones deben tener puntuaciones dentro del rango establecido (1-5).
5. Validez Temporal: Las fechas relacionadas con valoraciones y creación

de listas no pueden ser futuras.

6. Integridad Referencial: Si se elimina un contenido, se deben eliminar todas sus valoraciones asociadas y referencias en listas.
7. Seguridad de Acceso: Solo moderadores y administradores pueden modificar o eliminar contenido del catálogo general.
8. Privacidad de Datos: La información personal del usuario solo es accesible por el propio usuario y administradores.

1.3. Diccionario de Datos

Usuario

Atributo	Tipo	Descripción	Restricciones
id	String	Identificador único del usuario	Clave primaria, generado automáticamente
nombre	String	Nombre completo del usuario	Obligatorio, máximo 100 caracteres
email	String	Correo electrónico del usuario	Obligatorio, único, formato email válido
contraseña	String	Contraseña encriptada	Obligatorio, mínimo 8 caracteres, almacenada con hash
preferencias	List<Preferencia>	Lista de preferencias del usuario	Puede ser vacía

Relaciones:

- Un Usuario puede realizar múltiples Valoraciones (1:N)
- Un Usuario puede crear múltiples ListasPersonalizadas (1:N)
- Un Usuario puede tener múltiples Preferencias (1:N)
- Un Usuario puede recibir múltiples Recomendaciones (1:N)

Contenido (Clase abstracta)

Atributo	Tipo	Descripción	Restricciones
id	String	Identificador único del	Clave primaria, generado automáticamente

		contenido	
titulo	String	Título del libro o película	Obligatorio, máximo 200 caracteres
añoPublicacion	Integer	Año de publicación o estreno	Obligatorio, no puede ser posterior al año actual
generos	List<String>	Lista de géneros asociados	Al menos un género obligatorio
descripcion	String	Sinopsis o resumen	Obligatorio, entre 50 y 2000 caracteres
imagen	String	URL de la portada o cartel	Opcional

Relaciones:

- Un Contenido puede recibir múltiples Valoraciones (1:N)
- Un Contenido puede pertenecer a múltiples ListasPersonalizadas (N:M)

Libro (Hereda de Contenido)

Atributo	Tipo	Descripción	Restricciones
autor	String	Autor del libro	Obligatorio, máximo 100 caracteres
ISBN	String	International Standard Book Number	Opcional, único si existe, formato válido
editorial	String	Editorial del libro	Opcional, máximo 100 caracteres
numeroPaginas	Integer	Número de páginas	Opcional, mayor que 0

Película (Hereda de Contenido)

Atributo	Tipo	Descripción	Restricciones
director	String	Director de la película	Obligatorio, máximo 100 caracteres
actores	List<String>	Lista de actores principales	Al menos un actor recomendado
duracionMinutos	Integer	Duración en minutos	Opcional, mayor que 0

Valoración

Atributo	Tipo	Descripción	Restricciones
puntuacion	Integer	Puntuación asignada	Obligatorio, entre 1 y 5

comentario	String	Reseña textual	Opcional, máximo 1000 caracteres
fechaValoracion	Date	Fecha de la valoración	Obligatorio, no puede ser futura

Relaciones:

- Una Valoración pertenece a un Usuario (N:1)
- Una Valoración se asocia a un Contenido (N:1)

ListaPersonalizada

Atributo	Tipo	Descripción	Restricciones
id	String	Identificador único de la lista	Clave primaria, generado automáticamente
nombre	String	Nombre de la lista	Obligatorio, único por usuario, máximo 50 caracteres
descripcion	String	Descripción de la lista	Opcional, máximo 500 caracteres
esPublica	Boolean	Indica si la lista es visible para otros usuarios	Obligatorio, por defecto false

Relaciones:

- Una ListaPersonalizada pertenece a un Usuario (N:1)
- Una ListaPersonalizada puede contener múltiples Contenidos (N:M)

Recomendación

Atributo	Tipo	Descripción	Restricciones
fechaGeneracion	Date	Fecha de generación de la recomendación	Obligatorio, no puede ser futura
relevancia	Double	Puntuación de relevancia para el usuario	Obligatorio, entre 0.0 y 1.0

Relaciones:

- Una Recomendación se genera para un Usuario (N:1)
- Una Recomendación sugiere uno o más Contenidos (1:N)

Preferencia

Atributo	Tipo	Descripción	Restricciones
categoria	String	Categoría de la preferencia (género, autor, actor, etc.)	Obligatorio

nivelInteres	Integer	Nivel de interés del usuario	Obligatorio, entre 1 y 5

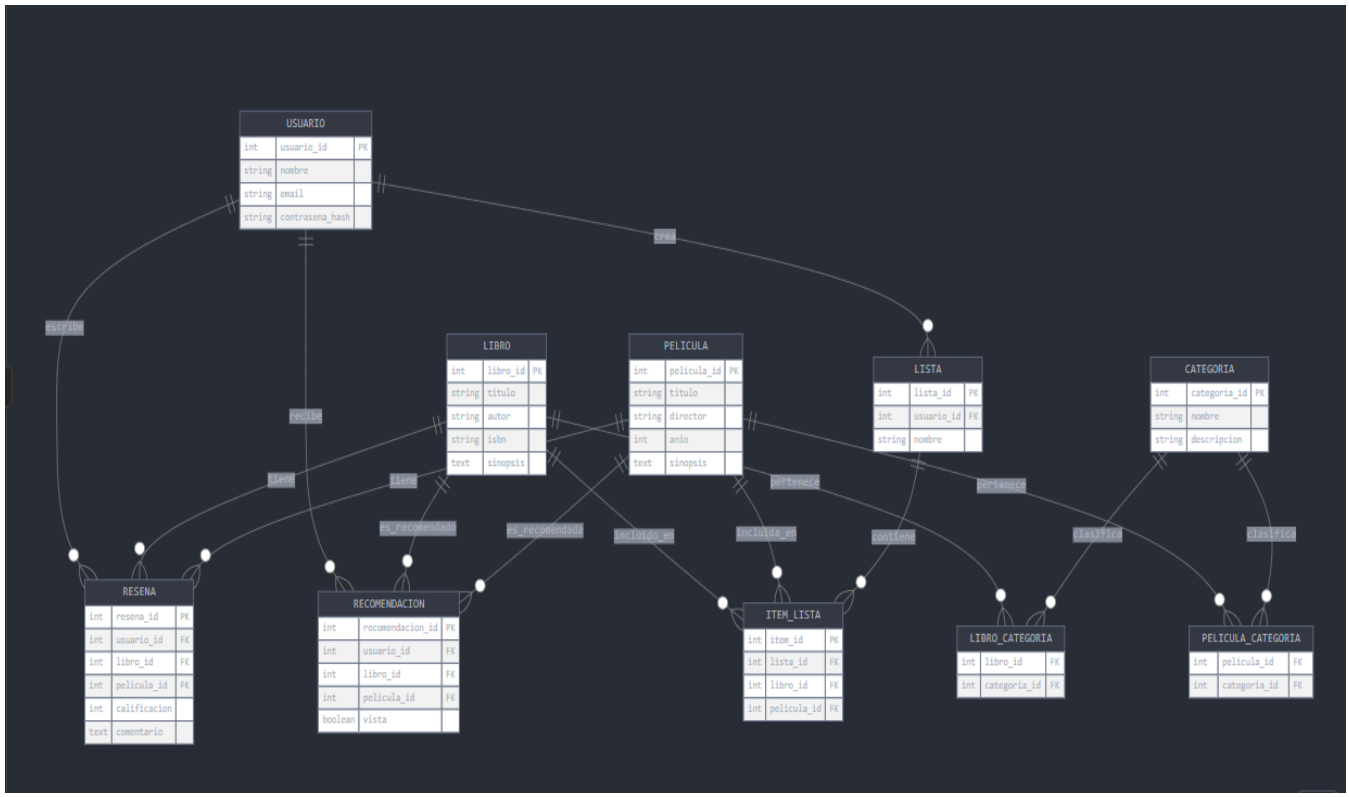
Relaciones:

- Una Preferencia pertenece a un Usuario (N:1)

APIOpenLibrary

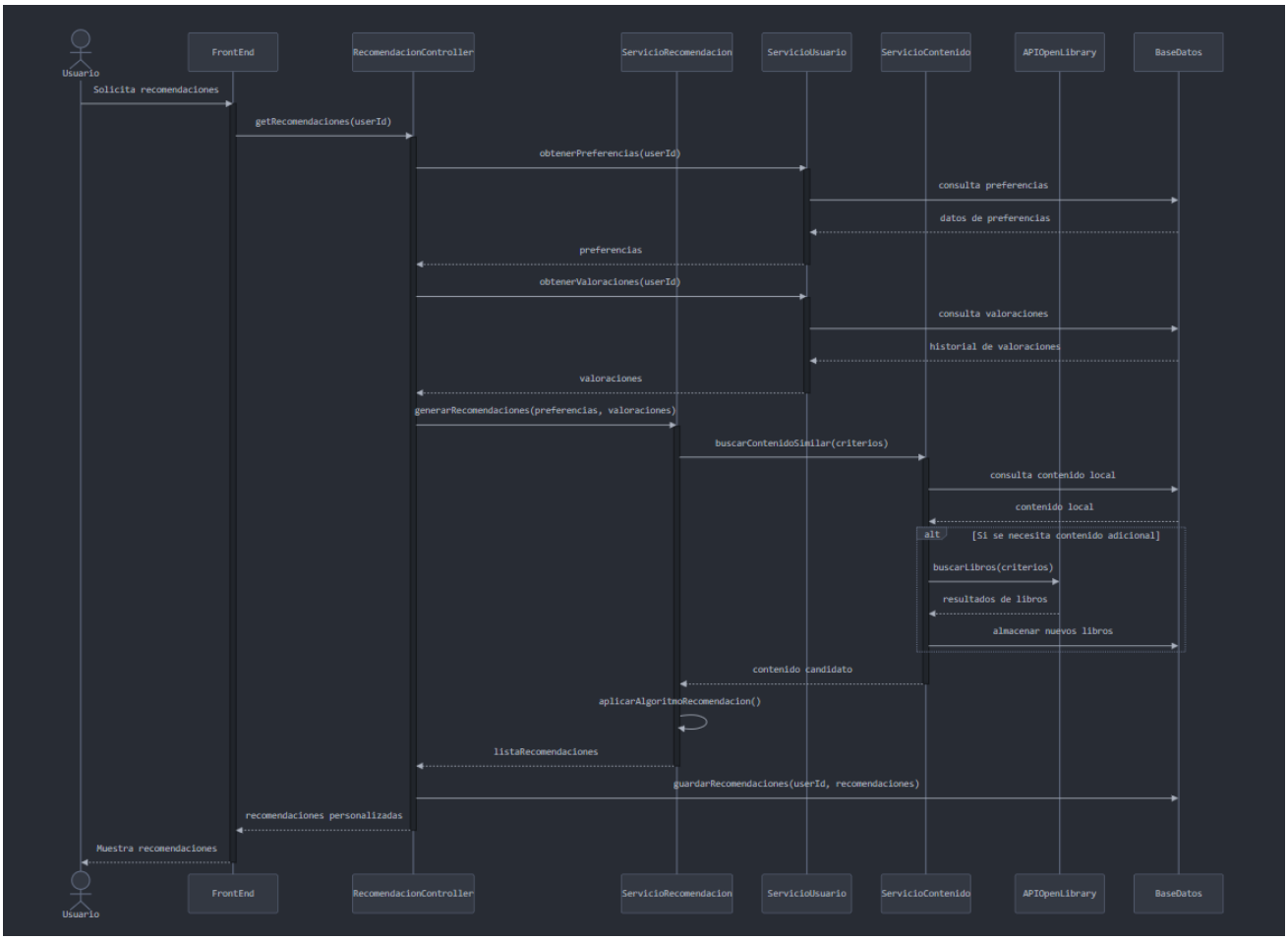
Esta clase representa la integración con la API externa y no tiene una tabla en la base de datos.

1.4. Diagrama entidad-relación

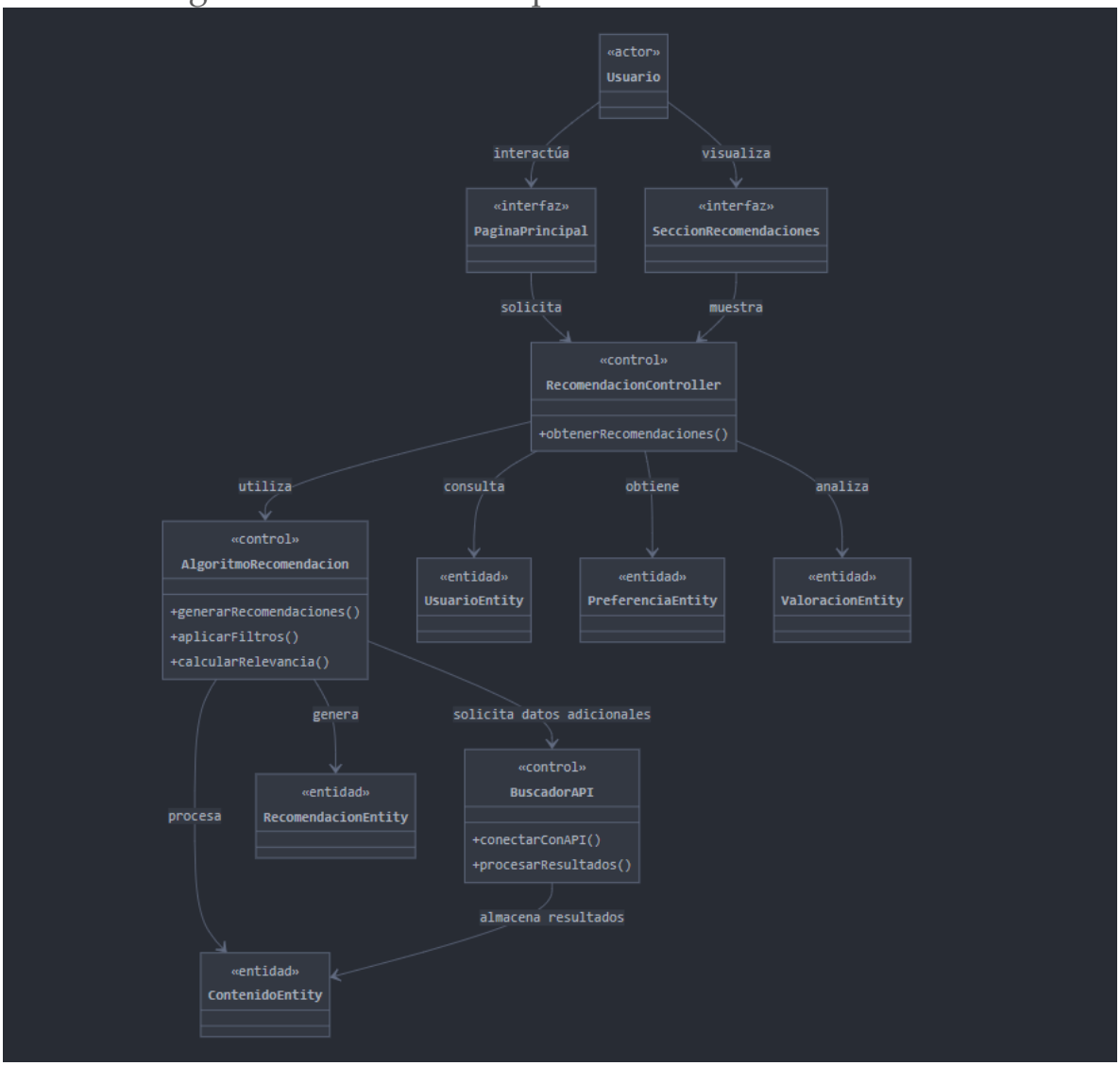


2. Diagramas de Secuencia y Robustez para los Casos de Uso

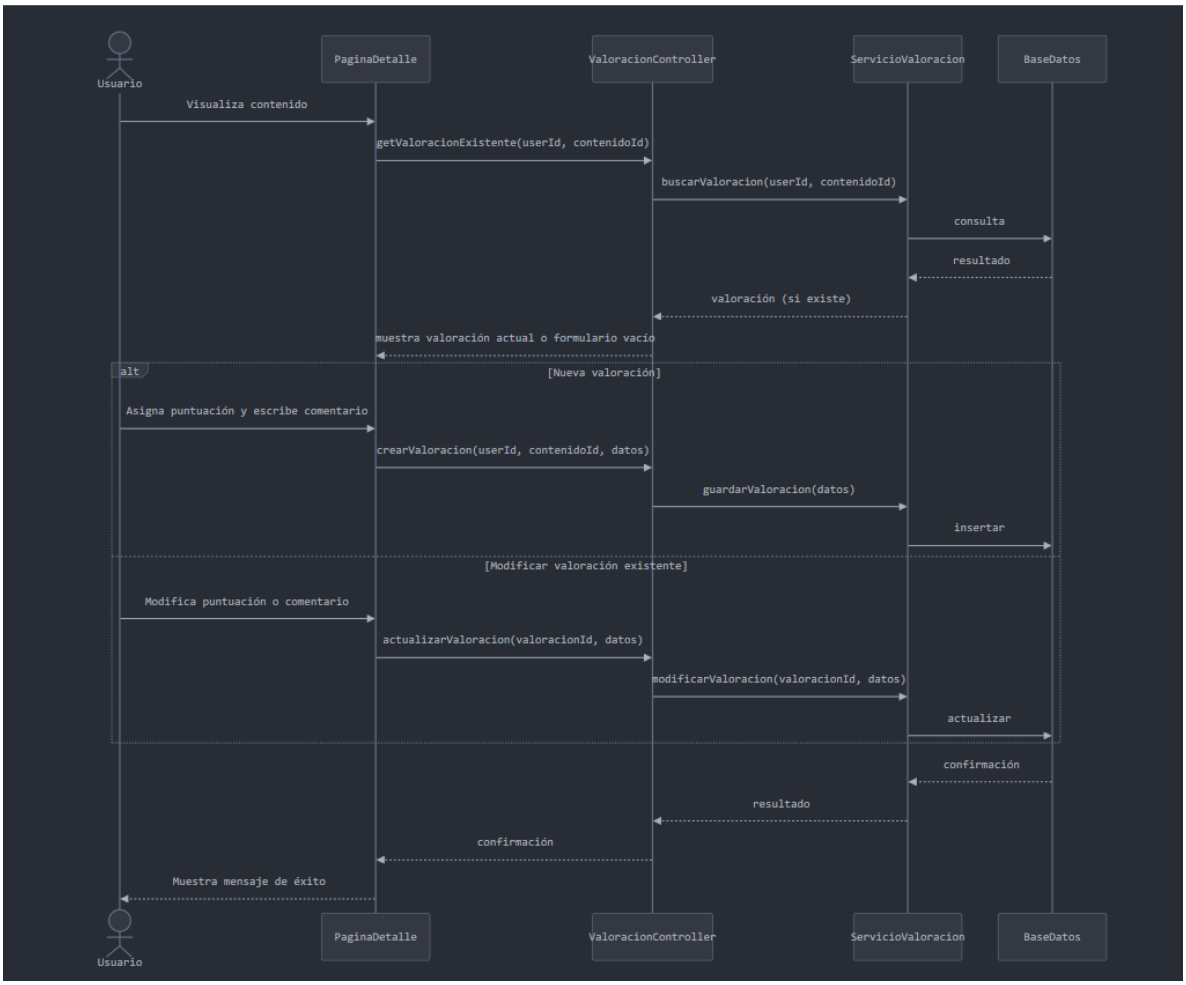
2.1. Diagrama de Secuencia para el Caso de Uso Principal: Recibir Recomendaciones



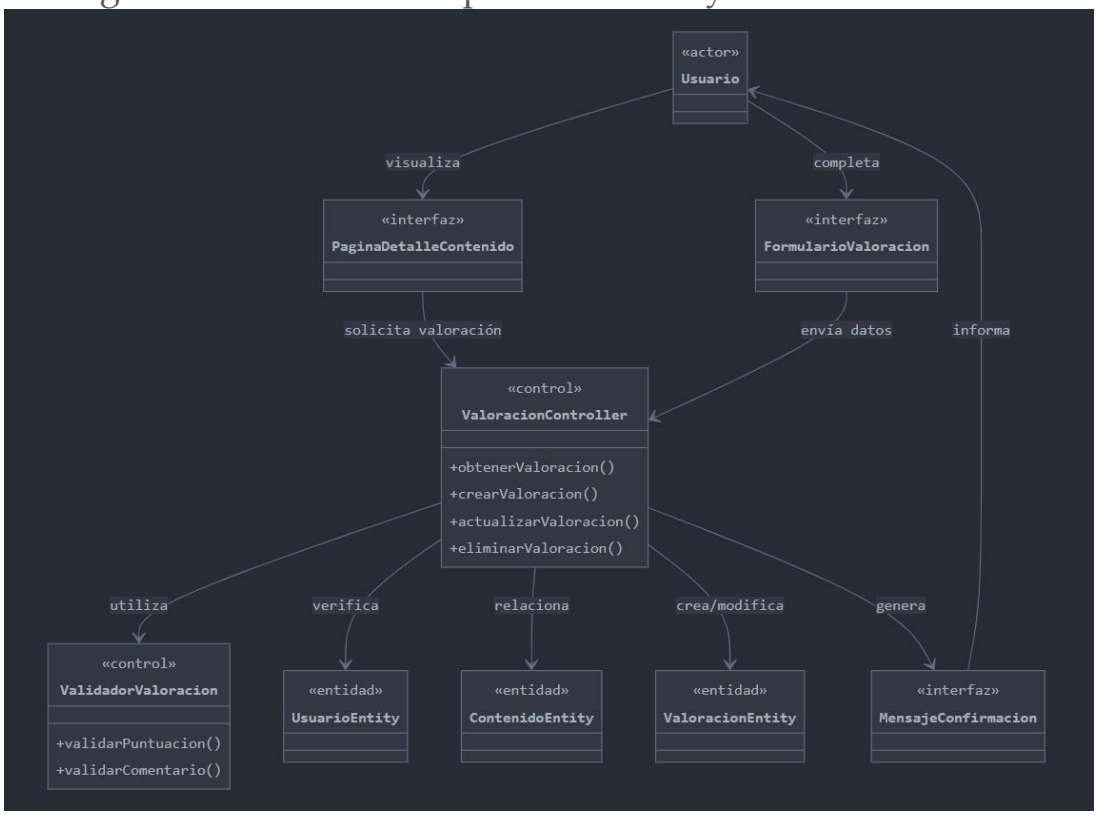
2.2. Diagrama de Robustez para Recibir Recomendaciones



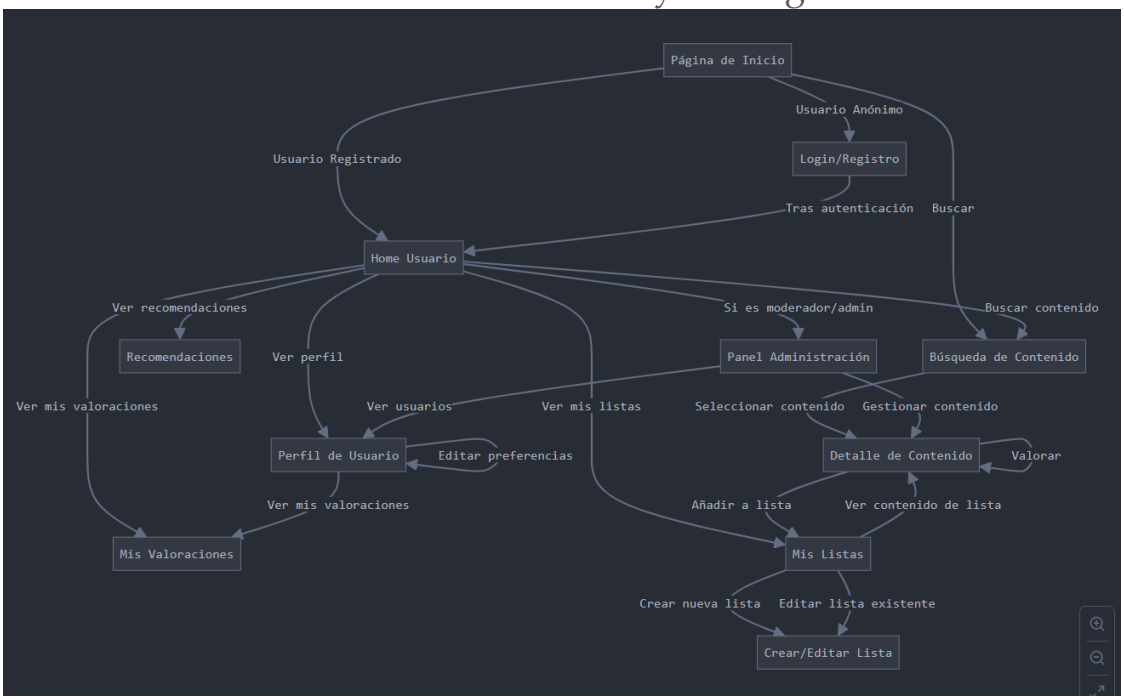
2.3. Diagrama de Secuencia Simplificado para Valorar y Comentar Contenido



2.4. Diagrama de Robustez para Valorar y Comentar Contenido

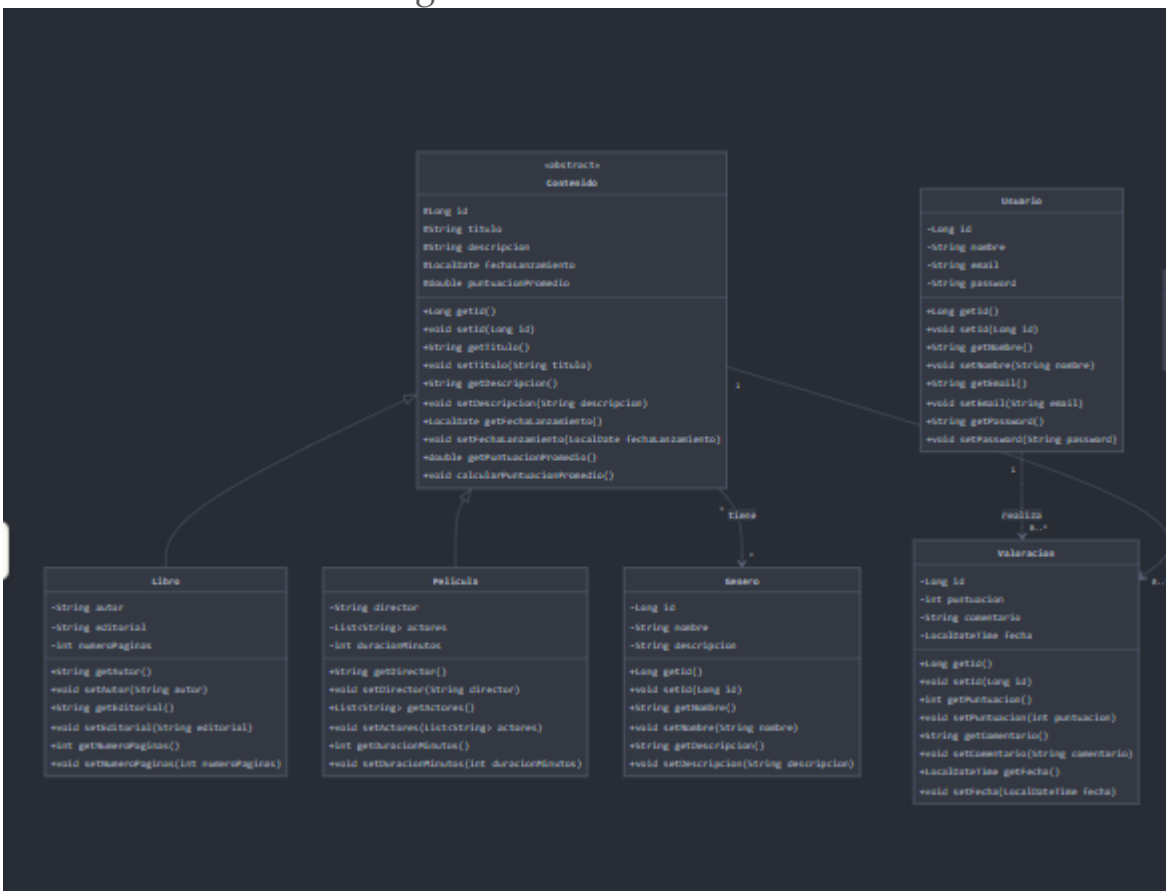


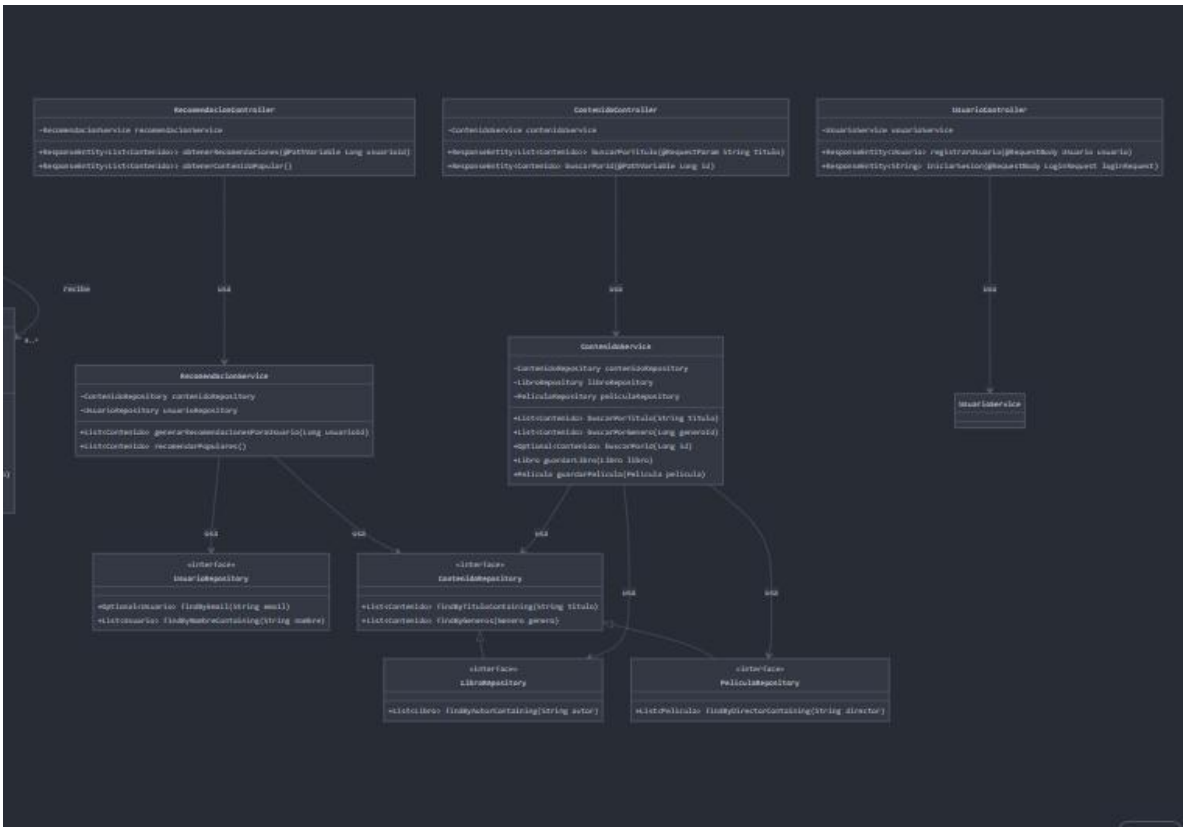
2.5. Modelo de Interfaz y Navegación



3. Implementación del Modelo de Diseño y Patrones

3.1. Diagrama de Clases de Diseño





3.2. Patrón de diseño

Patrón Strategy para el Sistema de Recomendación

Descripción del Patrón

El patrón Strategy define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Este patrón permite que el algoritmo varíe independientemente de los clientes que lo utilizan.

Justificación

Para nuestro sistema de recomendación de libros y películas, hemos implementado el patrón Strategy por las siguientes razones:

1. **Flexibilidad en algoritmos de recomendación**: El sistema necesita utilizar diferentes algoritmos de recomendación según el contexto (basado en géneros, popularidad, o filtrado colaborativo).
2. **Mantenibilidad y extensibilidad**: Facilita la adición de nuevos algoritmos de recomendación sin modificar el código existente.
3. **Selección dinámica**: Permite seleccionar el algoritmo más adecuado en tiempo de ejecución según el perfil del usuario o la situación específica.
4. **Separación de responsabilidades**: Cada estrategia se enfoca en un método específico de generación de recomendaciones.

Implementación

Interfaz Strategy

```
public interface RecomendacionStrategy {
   List<Contenido> recomendar(Usuario usuario, List<Valoracion> valoraciones,
List<Preferencia> preferencias);
}
```

Implementaciones Concretas

1. **Recomendación por Género**

```
@Component
public class RecomendacionPorGenero implements RecomendacionStrategy {
   @Autowired
   private ContenidoRepository contenidoRepository;

   @Override
   public List<Contenido> recomendar(Usuario usuario, List<Valoracion>
valoraciones, List<Preferencia> preferencias) {
      // Identificar géneros preferidos basados en preferencias y valoraciones
      Set<String> generosPreferidos = obtenerGenerosPreferidos(preferencias,
valoraciones);

      // Buscar contenido con géneros similares que el usuario no haya valorado
      return contenidoRepository.findByGenerosInAndIdNotIn(
         generosPreferidos,
         valoraciones.stream().map(v ->
v.getContenido().getId()).collect(Collectors.toList())
      );
   }

   private Set<String> obtenerGenerosPreferidos(List<Preferencia> preferencias,
List<Valoracion> valoraciones) {
      // Lógica para extraer géneros preferidos
      // ...
      return generosPreferidos;
   }
}
```

2. **Recomendación por Popularidad**

```
@Component
public class RecomendacionPorPopularidad implements RecomendacionStrategy {
   @Autowired
   private ValoracionRepository valoracionRepository;
   @Autowired
   private ContenidoRepository contenidoRepository;
```

```
    @Override
    public List<Contenido> recomendar(Usuario usuario, List<Valoracion>
valoraciones, List<Preferencia> preferencias) {
        // Obtener contenido popular basado en valoraciones globales
        List<String> contenidosYaValorados = valoraciones.stream()
            .map(v -> v.getContenido().getId())
            .collect(Collectors.toList());

        // Buscar contenido bien valorado que el usuario no haya valorado
        return valoracionRepository.findTopRatedContent(contenidosYaValorados);
    }
}
```

3. **Recomendación Colaborativa**

```
@Component
public class RecomendacionColaborativa implements RecomendacionStrategy {
    @Autowired
    private UsuarioRepository usuarioRepository;
    @Autowired
    private ValoracionRepository valoracionRepository;

    @Override
    public List<Contenido> recomendar(Usuario usuario, List<Valoracion>
valoraciones, List<Preferencia> preferencias) {
        // Encontrar usuarios con gustos similares
        List<Usuario> usuariosSimilares = encontrarUsuariosSimilares(usuario,
valoraciones);

        // Recomendar contenido que a esos usuarios les gustó pero que el usuario
actual no ha visto
        return obtenerContenidoDeUsuariosSimilares(usuariosSimilares,
valoraciones);
    }

    private List<Usuario> encontrarUsuariosSimilares(Usuario usuario,
List<Valoracion> valoraciones) {
        // Algoritmo para encontrar usuarios con valoraciones similares
        // ...
        return usuariosSimilares;
    }
```

```
    private List<Contenido> obtenerContenidoDeUsuariosSimilares(List<Usuario>
usuariosSimilares, List<Valoracion> valoracionesUsuarioActual) {
        // Obtener contenido bien valorado por usuarios similares que el usuario
actual no ha visto
        // ...
        return contenidoRecomendado;
    }
}
```

Contexto que utiliza la Strategy

```
@Service
public class RecomendacionService {
    @Autowired
    private RecomendacionRepository recomendacionRepository;
    @Autowired
    private UsuarioService usuarioService;
    @Autowired
    private ValoracionService valoracionService;

    @Autowired
    private Map<String, RecomendacionStrategy> estrategias;

    public List<Recomendacion> generarRecomendaciones(String userId) {
        Usuario usuario = usuarioService.obtenerPorId(userId);
        List<Valoracion> valoraciones = valoracionService.obtenerPorUsuario(userId);
        List<Preferencia> preferencias = usuarioService.obtenerPreferencias(userId);

        // Seleccionar la estrategia más adecuada según el perfil del usuario
        RecomendacionStrategy estrategiaSeleccionada;

        if (valoraciones.size() < 5) {
            // Para usuarios nuevos, recomendar por popularidad
            estrategiaSeleccionada = estrategias.get("recomendacionPorPopularidad");
        } else if (tienePreferenciasDefinidas(preferencias)) {
            // Si tiene preferencias claras, recomendar por género
            estrategiaSeleccionada = estrategias.get("recomendacionPorGenero");
        } else {
            // Por defecto, usar recomendación colaborativa
            estrategiaSeleccionada = estrategias.get("recomendacionColaborativa");
        }
```

```
        // Generar recomendaciones usando la estrategia seleccionada
        List<Contenido> contenidosRecomendados =
estrategiaSeleccionada.recomendar(usuario, valoraciones, preferencias);

        // Convertir contenidos recomendados en entidades de recomendación
        return crearEntidadesRecomendacion(usuario, contenidosRecomendados);
    }

    private boolean tienePreferenciasDefinidas(List<Preferencia> preferencias) {
        // Lógica para determinar si el usuario tiene preferencias bien definidas
        // ...
    }

    private List<Recomendacion> crearEntidadesRecomendacion(Usuario usuario,
List<Contenido> contenidos) {
        // Crear y guardar entidades de recomendación
        // ...
    }
}
```

Beneficios Conseguidos

1. **Adaptabilidad**: El sistema puede adaptar sus recomendaciones según el perfil de usuario, ofreciendo una experiencia personalizada.
2. **Escalabilidad**: Se pueden añadir nuevos algoritmos de recomendación sin modificar el código existente.
3. **Testabilidad**: Cada estrategia puede ser probada de forma independiente.
4. **Mejora en la calidad de recomendaciones**: Al tener diferentes estrategias disponibles, se puede seleccionar la más adecuada para cada contexto.
5. **Mantenibilidad**: La lógica compleja de cada algoritmo está encapsulada en su propia clase.

3.3. Modelo de implementación

Arquitectura Técnica

El sistema de recomendación de libros y películas se implementa utilizando una arquitectura de múltiples capas basada en Spring Boot, siguiendo el patrón Modelo-Vista-Controlador (MVC).

Capas de la Arquitectura

1. **Capa de Presentación**
    - Frontend desarrollado con Thymeleaf y Bootstrap
    - Interfaces responsivas para acceso desde diferentes dispositivos

    - Comunicación mediante REST API para funcionalidades dinámicas
2. **Capa de Aplicación**
    - Controladores REST para manejar las solicitudes HTTP
    - Servicios que implementan la lógica de negocio
    - DTOs (Data Transfer Objects) para transferir datos entre capas
3. **Capa de Dominio**
    - Entidades de dominio que representan los conceptos del negocio
    - Interfaces de repositorio que definen operaciones sobre las entidades
    - Servicios de dominio que encapsulan lógica compleja
4. **Capa de Infraestructura**
    - Implementaciones de repositorios JPA
    - Clientes para APIs externas (Open Library)
    - Configuraciones de seguridad y servicios transversales

Tecnologías y Frameworks

Backend

- **Spring Boot 3.1**: Framework principal para desarrollo
- **Spring Data JPA**: Para la persistencia de datos
- **Spring Security**: Para autenticación y autorización
- **JWT (JSON Web Tokens)**: Para manejo de sesiones
- **Lombok**: Para reducir código boilerplate
- **MapStruct**: Para mapeo entre entidades y DTOs
- **OpenAPI/Swagger**: Para documentación de API REST

Base de Datos

- **MongoDB**: Base de datos NoSQL para almacenamiento de datos
- **MongoDB Atlas**: Servicio cloud para hosting de la base de datos
- **Spring Data MongoDB**: Para integración con MongoDB

APIs Externas

- **Open Library API**: Para obtener información de libros
- **RestTemplate**: Para consumir APIs externas
- **WebClient**: Para comunicaciones reactivas con APIs externas (alternativa)

Frontend

- **Thymeleaf**: Motor de plantillas para vistas
- **Bootstrap 5**: Framework CSS para diseño responsivo
- **jQuery**: Para manipulación del DOM y AJAX
- **Font Awesome**: Para iconografía

Herramientas de Desarrollo

- **Maven**: Gestión de dependencias y construcción
- **JUnit 5**: Framework de pruebas unitarias
- **Mockito**: Framework para mocking en pruebas
- **Spring Boot Test**: Utilidades de prueba para Spring Boot

- **Git**: Control de versiones
- **GitHub Actions**: CI/CD

Estructura de Componentes

```
com.recomendacion
├─── config                # Configuraciones de la aplicación
│    ├─── SecurityConfig.java   # Configuración de seguridad
│    ├─── MongoConfig.java      # Configuración de MongoDB
│    └─── OpenLibraryConfig.java # Configuración del cliente de Open Library
├─── controller            # Controladores REST
│    ├─── UsuarioController.java
│    ├─── ContenidoController.java
│    ├─── ValoracionController.java
│    ├─── ListaController.java
│    └─── RecomendacionController.java
├─── service               # Servicios de aplicación
│    ├─── UsuarioService.java
│    ├─── ContenidoService.java
│    ├─── ValoracionService.java
│    ├─── ListaService.java
│    ├─── RecomendacionService.java
│    └─── strategy          # Implementaciones del patrón Strategy
│         ├─── RecomendacionStrategy.java
│         ├─── RecomendacionPorGenero.java
│         ├─── RecomendacionPorPopularidad.java
│         └─── RecomendacionColaborativa.java
├─── model                 # Modelos de dominio
│    ├─── Usuario.java
│    ├─── Contenido.java
│    ├─── Libro.java
│    ├─── Pelicula.java
│    ├─── Valoracion.java
│    ├─── ListaPersonalizada.java
│    ├─── Preferencia.java
│    └─── Recomendacion.java
├─── repository            # Repositorios para acceso a datos
│    ├─── UsuarioRepository.java
│    ├─── ContenidoRepository.java
│    ├─── LibroRepository.java
│    ├─── PeliculaRepository.java
│    ├─── ValoracionRepository.java
```

```
│   ├── ListaRepository.java
│   └── RecomendacionRepository.java
├── dto                 # Data Transfer Objects
│   ├── UsuarioDTO.java
│   ├── LoginDTO.java
│   ├── ContenidoDTO.java
│   ├── LibroDTO.java
│   ├── PeliculaDTO.java
│   ├── ValoracionDTO.java
│   ├── ListaDTO.java
│   └── PreferenciaDTO.java
├── client              # Clientes para APIs externas
│   └── OpenLibraryClient.java
├── exception           # Excepciones personalizadas y manejo
│   ├── ResourceNotFoundException.java
│   ├── DuplicateResourceException.java
│   └── GlobalExceptionHandler.java
└── util                # Utilidades varias
    ├── JwtTokenProvider.java
    ├── MapperUtil.java
    └── ValidationUtil.java
```

Consideraciones sobre Seguridad, Rendimiento y Escalabilidad

Seguridad

- Autenticación basada en JWT para el acceso a API REST
- Cifrado de contraseñas utilizando BCrypt
- Protección CSRF para formularios web
- Validación de entradas en servidor y cliente
- Implementación de CORS para API REST
- Manejo de sesiones seguras
- Auditoría de acciones críticas

Rendimiento

- Caché de datos para reducir llamadas a API externas
- Paginación para conjuntos grandes de datos
- Optimización de consultas a la base de datos
- Carga diferida de recursos pesados
- Compresión de respuestas HTTP
- Minimización de recursos estáticos (JS, CSS)

Escalabilidad

- Arquitectura sin estado para facilitar escalado horizontal
- Separación de la aplicación en servicios independientes

- Balanceo de carga para distribuir tráfico
- Índices eficientes en la base de datos
- Procesamiento asíncrono para tareas pesadas
- Actualización periódica de recomendaciones mediante jobs programados

3.4. Diagrama de despliegue