Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Ingeniería del Software

Aplicación Android/Web de redes sociales en Elixir – Cine

Autor
David Cuñado Gil

Tutor
Dr. César Llamas Bello

Agradecimientos

A mis padres, Leandro y Ana, sin ellos nada de esto hubiera sido posible. Gracias por creer en mí cuando ni yo mismo lo hacía.

A mi tutor, Don César Llamas Bello, por su dedicación y guía a lo largo de todo el proyecto.

A mi pareja, Isabel, por apoyarme y soportarme incluso en las partes más duras del proceso.

Y, por último, me gustaría agradecérselo también a los usuarios de *Stack Overflow* y *Elixir Forum* que tuvieron la valentía de plantear sus dudas; y, sobre todo, a los que tuvieron la bondad de resolverlas.

Resumen

En el presente documento se analizará la aplicación 'Filmoteca', creada como trabajo de fin de grado por el alumno de ingeniería informática, especializado en ingeniería del software, David Cuñado Gil. Su concepción nace de la necesidad de reducir el tiempo que tarda una persona en elegir qué película ver y para sustituir la típica nota del móvil en la que la mayoría de las personas apuntan las recomendaciones cinematográficas que se les hacen, y que frecuentemente acaba olvidada entre el resto de notas.

Esta aplicación se desarrolla bajo el modelo de diseño software cliente-servidor. El cliente será realizado en Java para la plataforma Android y será el punto de acceso y de interacción de los usuarios con el sistema.

El servidor será desarrollado en el lenguaje de programación concurrente Elixir, e implementado gracias al framework Phoenix.

Abstract

In the present document the 'Filmoteca' app will be analized, which has been made as the final degree project of the computer engineer student, specialized in software engineering, David Cuñado Gil. Its conception borns of the need to reduce the time it takes for a person to choose which movie to see and to replace the typical phone's note where most of people write films recommendations, which frequently get lost among the other notes.

This app has been developed under the client-server software design model. The cliente will be made in Java for the Android platform and it will be the point of access and interaction of the users with the system.

The server will be developed in the concurrent programming language Elixir, and implemented by the framework Phoenix.

Índice de contenidos

Capítulo 1 - Introducción 1
1.1. Introducción 1
1.2. Motivación 1
1.3. Objetivos 2
1.4. Metodología utilizada 2
1.4.1. Planificación 2
1.5. Estructura del documento 4
Capítulo 2 – Antecedentes y herramientas utilizadas 5
2.1. Esquema general 5
2.2. APIs de consulta 6
2.2.1. TMDB 6
2.2.2. OMDB 7
2.3. Android 7
2.3.1. Breve historia de Android 7
2.3.2. Cuota de mercado 9
2.4. Phoenix framework 10
2.4.1. Antecedentes 10
2.4.2. Características del framework 13
2.4.3. Componentes 13
2.5. Herramientas utilizadas 16
2.5.1. Cliente 17
2.5.2. Servidor 17
Capítulo 3 - Análisis 19
3.1. Presentación del sistema 19
3.2. Captura de requisitos 19
3.2.1. Requisitos funcionales 19

3.2.2. Requisitos no funcionales 21

3.3. Casos de uso 22

3.3.1. Descripción de los casos de uso 25

3.4. Modelo de dominio del sistema 34

Capítulo 4 - Diseño 35

4.1. Tecnología utilizada 35

4.1.1. Tecnología utilizada en el cliente 35

4.1.2. Tecnología utilizada en el servidor 37

4.2. Diagramas de diseño 37

4.2.1. Diagrama de paquetes del cliente 37

4.2.2. Diagrama de clases del servidor 39

4.2.3. Diagrama de despliegue del sistema 40

4.3. Diseño de las bases de datos 41

4.3.1. Base de datos del servidor 41

4.3.2. Base de datos del cliente 42

4.4. Diseño de la interfaz de usuario 42

4.4.1. Watchlists personales 44

4.4.2. Buscador de películas 45

4.4.3. Watchlists sociales 46

Capítulo 5 - Implementación 47

5.1. Interfaces de conexión 47

5.2. Adaptadores de vista 50

5.3. Actividad principal 53

5.4. Login y registro 56

5.4.1. Gestión de la sesión 57

5.4.2. Login 58

5.4.3. Registro 64

5.5. Administración de watchlists personales 67

5.6. Búsqueda de películas 72

5.7. Gestión de suscripciones del usuario 79

Capítulo 6 - Pruebas 86

6.1. Pruebas de caja blanca 86

6.2. Pruebas de caja negra 86

Capítulo 7 - Conclusiones 91

7.1. Conclusiones 91

7.2. Líneas de trabajo futuro 92

Capítulo 1

Introducción

1.1. Introducción

El presente documento constituye la memoria realizada para el trabajo de fin de grado de ingeniería informática de David Cuñado Gil bajo la tutela del Doctor César Llamas Bello. En él, se documenta el desarrollo de un sistema basado en la arquitectura cliente-servidor desde la etapa de análisis hasta la de pruebas.

El producto resultante consistirá en una aplicación Android en la que los usuarios podrán crearse una cuenta para crear listas de películas (en adelante *'watchlists'*) y tener acceso a ellas en cualquier terminal compatible con la aplicación. También implementará funcionalidades de red social, permitiendo así a los usuarios compartir sus listas con el resto de la comunidad y suscribirse a las que les interesen.

1.2. Motivación

Desde que se inició el estado de alarma en España y fuimos confinados en nuestros hogares, la forma de ocio de la sociedad sufrió importantes cambios, limitándose exclusivamente a los "planes caseros". Incluso con el fin del confinamiento, debido a las restricciones y a cuestiones sanitarias, se siguió optando por este tipo de plan.

La realización de este trabajo nace motivada por esta situación, con el objetivo de reducir el tiempo que tarda una persona en elegir qué película ver. Por ello, se pensó que sería de gran utilidad disponer de una aplicación en el móvil en la que crear distintas listas de películas de entre las que se podrá elegir una de forma aleatoria. También se consideró interesante que los usuarios de la aplicación pudiesen compartir sus listas creadas como "públicas" y suscribirse a las de los demás para poder ampliar su conocimiento cinematográfico.

Otra gran motivación para la realización de este trabajo es conocer el funcionamiento del framework Phoenix, escrito en Elixir, sus ventajas y sus limitaciones. Este framework nos permitirá implementar la parte del servidor y gestionar los accesos concurrentes de los usuarios al mismo.

1.3. Objetivos

A continuación, se enumerarán los objetivos que se desea lograr con la realización de este trabajo:

- Aprender nociones básicas sobre el lenguaje de programación Elixir
- Implementar un servidor basado en el framework Phoenix escrito en Elixir
- Desarrollar una aplicación cliente para terminales Android
- Proporcionar a los usuarios una herramienta para crear y compartir listas de películas.

1.4. Metodología utilizada

Para el desarrollo de este proyecto se ha decidido emplear la metodología incremental. El objetivo de este modelo consiste en la producción de un sistema software generado en una sucesión de entregables en los que se va ampliando su funcionalidad. Cada entregable será el resultado de un ciclo de programación que denominaremos incremento.

Una de las principales ventajas de este modelo es que cada incremento, incluido el primero, constituye una versión reducida del sistema final, permitiéndonos realizar pruebas de caja blanca específicas para cada funcionalidad.

A continuación, detallaremos los incrementos establecidos para nuestro sistema y la funcionalidad implementada en cada uno de ellos.

1.4.1. Planificación

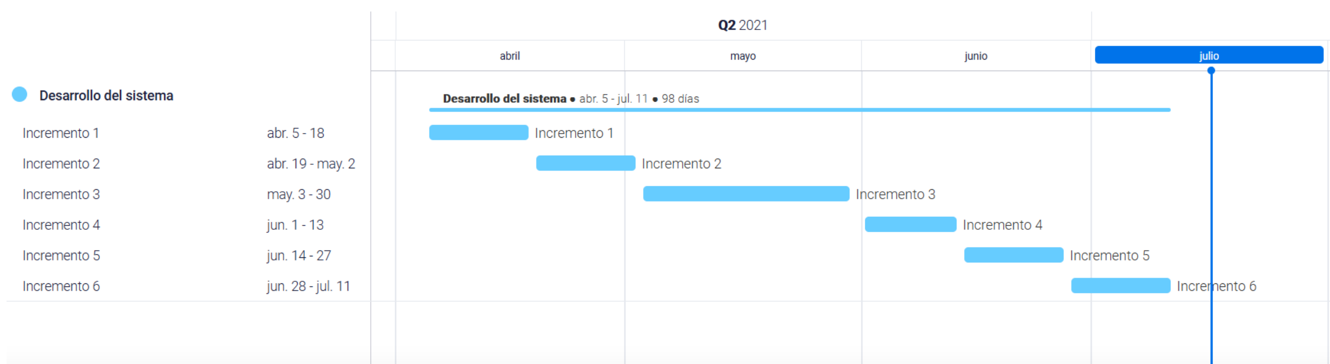

Figura 1.1 - Diagrama de Gantt simplificado del proyecto

Incremento 1: búsqueda de películas

Para comenzar con el desarrollo del sistema se fijó como primer objetivo encontrar una fuente de datos de películas fiable. Para ello, se realizó un pequeño trabajo de investigación sobre las diferentes APIs de consulta de datos existentes en el mercado, y finalmente se eligieron dos.

También fue el punto de toma de contacto con la herramienta Retrofit, una librería para Android que nos permite realizar peticiones HTTP y gestionar su respuesta.

Como resultado de este incremento, obtenemos una aplicación Android constituida tan solo por una actividad que implementa un buscador para introducir el término de la petición a realizar a las APIs. Los títulos de las películas del resultado de la petición se mostrarán por pantalla en forma de texto plano.

Incremento 2: watchlists y persistencia en el cliente

El siguiente paso a realizar consiste en introducir la entidad 'watchlist' al modelo de dominio. Con este fin, se traslada la lógica de la actividad resultante del incremento anterior a un fragmento y se establece la actividad principal como un contenedor de fragmentos, añadiendo una barra de navegación inferior con los tres elementos que se utilizarán en el sistema final.

Una vez tenemos diferenciados los tres espacios que albergarán las distintas funcionalidades de la aplicación, establecemos que el primero será para las listas personales, el de en medio para buscar las películas y el último, que permanecerá vacío hasta el quinto incremento, para las watchlists públicas.

En este incremento se implementará la lógica de crear y eliminar watchlists y la de añadir películas a las mismas. Estas operaciones se verán registradas en la base de datos SQLite creada para la aplicación.

El entregable producido en este incremento ya constituye en sí mismo una aplicación para la administración de listas de películas personales.

Incremento 3: registro, login y logout de usuarios

Es en este incremento cuando afrontamos el reto de trabajar con un nuevo framework, Phoenix, y el de aprender el lenguaje de programación Elixir. Este es el motivo principal de la larga duración de este incremento.

Como resultado de esta iteración, obtendremos nuestra primera versión del sistema con arquitectura cliente-servidor. El cliente contará con dos nuevas actividades, la de login y la de registro, y se habrá introducido un botón para finalizar la sesión. Además, dispondremos de un servidor operativo capaz de responder y validar las peticiones de login y registro por parte del cliente.

Incremento 4: persistencia en el servidor

Una vez se ha logrado establecer la conexión entre el cliente y el servidor, procedemos a implementar la funcionalidad que garantiza la persistencia de los datos del usuario en el servidor. Es decir, se introduce la lógica de negocio de la aplicación en el servidor y se crea la base de datos. También se establecerán los métodos necesarios en el cliente para realizar las peticiones al servidor.

Al finalizar esta iteración tendremos un servidor que responderá satisfactoriamente a las peticiones de inserción y borrado de las watchlists y películas de los usuarios en una base de datos PostgreSQL, y podremos obtener y ver estos datos en el cliente al iniciar sesión.

Incremento 5: watchlists públicas

Es en esta fase cuando se introducirá el concepto de "watchlist pública". Para ello, introduciremos un *checkbox* en el cuadro de diálogo de creación de la lista y el atributo "pública", de tipo booleano, en la clase watchlist.

Se desarrollará el fragmento perteneciente al último apartado de la aplicación y se implementará la funcionalidad para la búsqueda, suscripción y baja de watchlists públicas.

Como resultado, tendremos un sistema que implementa la totalidad de la funcionalidad propuesta.

Incremento 6: depuración del código, sistema final

Esta última iteración se corresponderá con la fase de depuración del código. Es decir, se corregirán errores, se mejorará el rendimiento mediante la simplificación del código, se añadirán los mensajes de error posibles para cada actividad y se terminará de pulir la interfaz de usuario, la cual se ha ido desarrollando y refinando en cada incremento.

Una vez terminado este incremento, contaremos con el sistema final listo para ser desplegado en producción.

1.5. Estructura del documento

En el *Capítulo II* introduciremos el marco de trabajo necesario para el desarrollo de este proyecto. Se profundizará en el entorno software donde tiene lugar y se detallarán las herramientas necesarias para llevarlo a cabo.

En el *Capítulo III* se ceñirá a la etapa de análisis del proyecto, estableciendo los requisitos, casos de uso y el modelo de dominio del sistema.

El *Capítulo IV* se dedicará a la fase de diseño, donde trataremos el middleware utilizado y se expondrán distintos diagramas para facilitar la comprensión del sistema. También se tratará aquí el diseño de la interfaz de usuario.

En el *Capítulo V* hablaremos de la implementación de nuestro sistema. Expondremos fragmentos de código necesarios para entender su funcionalidad y el flujo del programa.

En el *Capítulo VI* se especificarán los casos de prueba llevados a cabo para validar la funcionalidad del sistema.

Por último, en el *Capítulo VII*, expondremos las conclusiones obtenidas tras la realización del proyecto y las líneas de trabajo futuro que se podrían tomar.

Capítulo 2

Antecedentes y herramientas utilizadas

En este capítulo se tratará todo lo referente a la tecnología necesaria para la construcción de la aplicación. En primer lugar, se describirá el sistema de una forma general para luego profundizar sobre cada parte. En la siguiente sección hablaremos de las APIs de consulta empleadas, argumentando su elección. A continuación, analizaremos la elección tecnológica para el cliente y para el servidor, y, finalmente, detallaremos las herramientas utilizadas para la construcción y pruebas del sistema.

2.1. Esquema general

Esta aplicación, como muchas otras de redes sociales, se ha desarrollado utilizando una arquitectura cliente-servidor.

Una arquitectura cliente-servidor es un modelo de diseño de software que reparte la funcionalidad entre estos dos agentes, siendo el cliente el encargado de realizar peticiones y el servidor el de proporcionar una respuesta a estas. Según el número de servidores que emplee el sistema se catalogarán en arquitecturas cliente-servidor de N capas, siendo N el número de servidores empleados más uno (el cliente). También se clasificarán estas arquitecturas según dónde recaiga el peso de procesar la información, dando lugar a los clientes y servidores activos o pasivos.

En nuestro caso, nuestra arquitectura cliente-servidor será de 2 capas y de tipo cliente activo - servidor activo, pues los dos se encargarán de procesar la información. El servidor responderá a las peticiones del cliente procesando la información de su base de datos y sintetizando la respuesta. Por otro lado, el cliente recibirá esta respuesta y, además de mostrarla al usuario, la almacenará en su base de datos local SQLite.

El cliente se ha construido para la plataforma Android para la versión 28 de su API, decisión motivada por la necesidad de funcionar en dispositivos móviles y por el gran alcance que tiene este sistema operativo. Este será el encargado de comunicarse con el servidor y de realizar las consultas a las distintas bases de datos de películas.

Para acceder a estas bases de datos se trabajará con dos APIs de consulta: TMDB (*The Movie Database*) y OMDB (*Open Movie Database*). Estas APIs nos ofrecen una interfaz para acceder a los datos de las películas en formato JSON mediante peticiones GET.

Para la construcción del servidor se ha elegido el framework Phoenix, que correrá en una máquina Linux. Aquí se gestionará toda la información referente a los usuarios y sus watchlists, y la forma de acceder a ella.

2.2. APIs de consulta

Una API de consulta es la interfaz que se proporciona a los usuarios para acceder a la información de una determinada base de datos. En nuestro caso, nos interesa consultar bases de datos que contengan información sobre películas. Para ello se tuvieron en cuenta distintas APIs, y se eligieron dos en función de las necesidades de nuestra aplicación.

Los requisitos a satisfacer son:

- Poder realizar búsquedas de películas a partir de sus títulos en su idioma original y en español.
- Obtener varios resultados coincidentes con la búsqueda y no una única película.
- Tener acceso a información sobre películas de distintas nacionalidades,
- Obtener información relevante para la película buscada: título y año de lanzamiento, director, reparto, duración, sinopsis y las imágenes de carátula y fondo.

En un principio se quería utilizar IMDB (*Internet Movie Database*) como proveedor de la información de las películas, pero debido a la escasa información que obtenemos con su API, se optó por otras de acceso libre (previo registro).

Estas dos APIs elegidas son TMDB y OMDB. Si bien había más alternativas, se optó por estas ya que conjuntamente cumplen con todos nuestros requisitos.

2.2.1. TMDB

TMDB (The Movie Database [1] es una base de datos de películas y series originada en 2008, disponible en 39 idiomas y utilizada por más de 400.000 desarrolladores y compañías.

Si bien esta no fue la primera opción por su escasez de datos y el gran tamaño de las imágenes de las carátulas (aumenta el caché generado), es a la que se realiza la primera petición por permitir búsquedas de títulos en español y obtener varios resultados coincidentes.

De esta API obtenemos la siguiente información:

- Título de la película.
- Fecha de lanzamiento.
- Imagen de la carátula.
- Imagen de fondo para el detalle de la película.
- Sinopsis.
- Puntuación de TMDB.
- Título de la película en el lenguaje original.
- Popularidad.

Esta interfaz nos devuelve las coincidencias con el título introducido por el usuario en varias páginas si fuese necesario, con hasta 20 películas por página. Por tanto, se realizará una petición por cada página y se almacenarán

las películas en una lista de objetos de tipo 'TmdbMovie'. Por motivos de eficiencia y calidad se han limitado las peticiones a las tres primeras páginas.

Esta estructura será posteriormente filtrada, pues muchos títulos no son de interés, y se procederá a hacer una petición a la siguiente API por cada película restante en la lista.

2.2.2. OMDB

OMDB (Open Movie Database) [3] es otra base de datos de películas y series que lanzó su API pública en 2014. Al contrario que la API anterior, esta nos devuelve una única película por cada petición. Para realizarla utilizaremos el título original y el año de lanzamiento obtenidos previamente.

La razón del uso de esta segunda API es lo mucho que amplía la información de las películas, aportando los siguientes datos:

- Una imagen de carátula de tamaño más reducido, que se usará siempre y cuando sea posible.
- Duración.
- Director.
- Reparto.
- Puntuación de IMDB (más popular y, por tanto, la que se optará por usar).

Sin embargo, en base a las pruebas realizadas, esta interfaz presenta carencias de contenido en cuanto a películas de origen español se refiere. Por tanto, se decidió ofrecer una solución híbrida entre las dos APIs mencionadas, mostrando películas sin la información extra que nos brinda OMDB.

2.3. Android

Android es un sistema operativo diseñado para dispositivos móviles con pantalla táctil, como smartphones y tabletas entre otros. Está basado en el kernel de Linux y es un sistema de código abierto, lo cual facilita su implementación en una gran variedad de dispositivos de diferentes marcas.

2.3.1. Breve historia de Android

La historia de Android comienza en 2003, cuando Andy Rubin, Rich Miner, Chris White y Nick Sears fundan Android Inc. en Palo Alto, California. En 2005 la compañía es comprada por Google, pero no sería hasta el 5 de noviembre de 2007 cuando se forma la Open Handset Alliance y se anuncia la primera versión del sistema operativo: Android 1.0 Apple Pie. Actualmente se encuentra en su versión 11, y la 12 está en fase beta.

Los primeros terminales en incorporar Android salieron al mercado en 2008, y desde entonces ha recibido múltiples actualizaciones y ha diversificado hacia distintos dispositivos, como relojes, tablets, televisiones e incluso automóviles.

Code name	Version numbers	API level	Release date
No codename	1.0	1	September 23, 2008
No codename	1.1	2	February 9, 2009
Cupcake	1.5	3	April 27, 2009
Donut	1.6	4	September 15, 2009
Eclair	2.0 - 2.1	5 - 7	October 26, 2009
Froyo	2.2 - 2.2.3	8	May 20, 2010
Gingerbread	2.3 - 2.3.7	9 - 10	December 6, 2010
Honeycomb	3.0 - 3.2.6	11 - 13	February 22, 2011
Ice Cream Sandwich	4.0 - 4.0.4	14 - 15	October 18, 2011
Jelly Bean	4.1 - 4.3.1	16 - 18	July 9, 2012
KitKat	4.4 - 4.4.4	19 - 20	October 31, 2013
Lollipop	5.0 - 5.1.1	21- 22	November 12, 2014
Marshmallow	6.0 - 6.0.1	23	October 5, 2015
Nougat	7.0	24	August 22, 2016
Nougat	7.1.0 - 7.1.2	25	October 4, 2016
Oreo	8.0	26	August 21, 2017
Oreo	8.1	27	December 5, 2017
Pie	9.0	28	August 6, 2018
Android 10	10.0	29	September 3, 2019
Android 11	11	30	September 8, 2020

Figura 2.1 - Timeline de las versiones de Android [4]

2.3.2. Cuota de mercado

Actualmente, según StatCounter [bibliografía], en abril de 2021 un 72.19% de los smartphones implementaban Android como sistema operativo; frente al 26.99% que funcionaban con iOS. Estas cifras se deben a la condición de código abierto de Android, lo cual permite que distintas marcas puedan acceder a él y usarlo para sus terminales, ofreciendo así múltiples alternativas dentro de todos los rangos de precio del mercado.

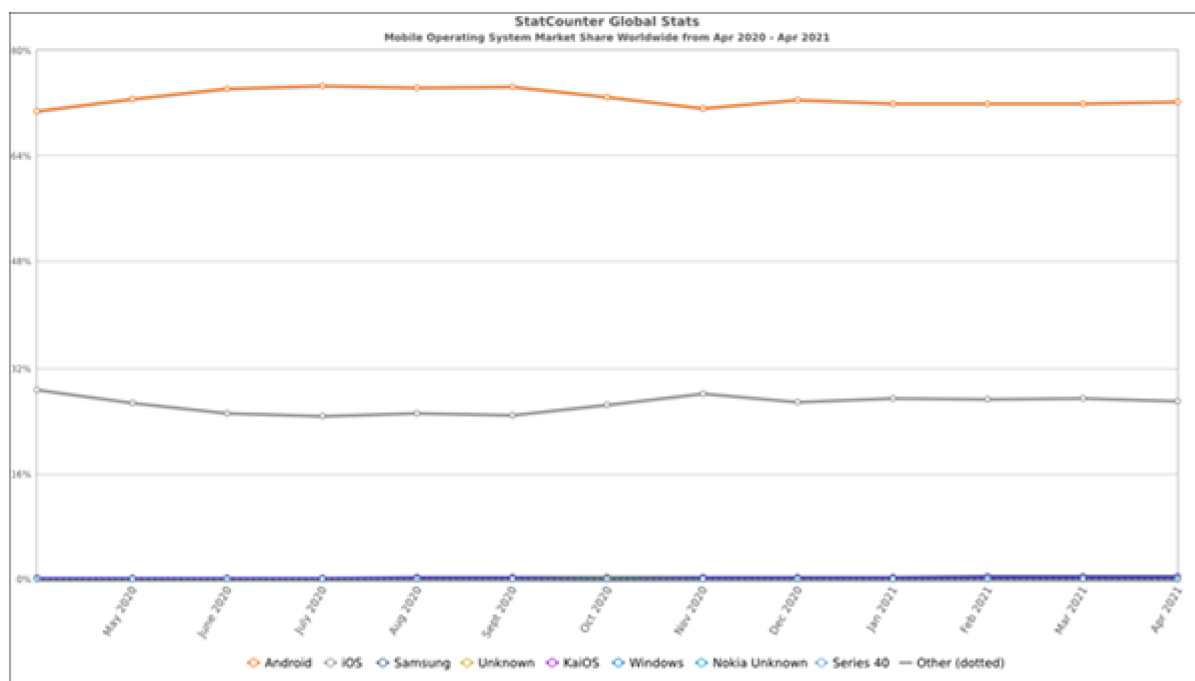

Figura 2.2 - Cuota de mercado de los sistemas operativos móviles [5]

Por otra parte, esta inmensa variedad de modelos dificulta el proceso de mantenimiento y actualización del sistema, pues involucra no solo a Google sino también a los fabricantes de los dispositivos. Esto radica en uno de los grandes problemas de Android para los desarrolladores: la fragmentación de versiones.

Actualmente, Android se encuentra en su versión 11, pero tan solo un 12.38% de los dispositivos están actualizados a esta versión.

Como ya se mencionó previamente, para el desarrollo de este cliente se ha elegido la API 28, correspondiente a Android 9 “Pie”, pues se trata de una versión relativamente reciente que permitirá que la aplicación funcione en el 69.09% de móviles Android (versiones 9, 10 y 11); además tampoco echamos en falta funcionalidad de versiones superiores.

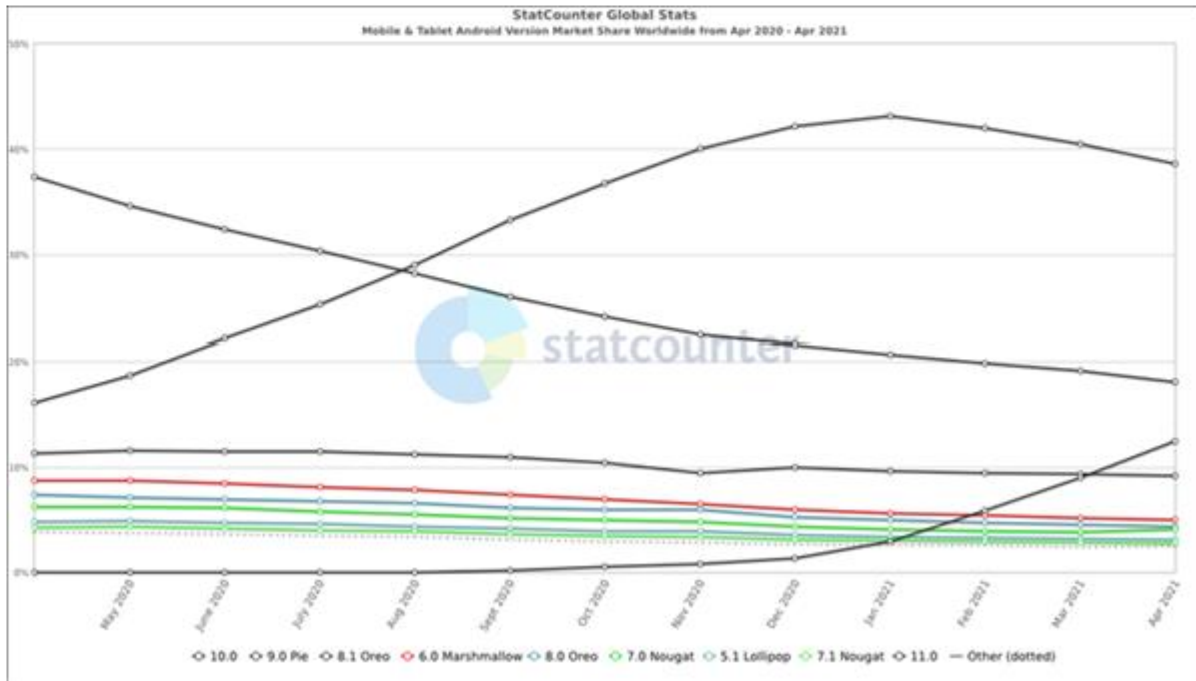

Figura 2.3 - Distribución de las distintas versiones de Android en móviles y tablets [6]

2.4. Phoenix framework

Phoenix es un framework web especializado para el desarrollo de la parte referente al servidor. Está escrito en Elixir y se ejecuta sobre la máquina virtual de Erlang (BEAM).

En esta sección profundizaremos en este framework, su funcionamiento y en las ventajas que nos brinda.

2.4.1. Antecedentes

Antes de hablar del framework Phoenix es preciso entender el contexto en el que se origina. Por ello, empezaremos describiendo Erlang (y su máquina virtual BEAM) y Elixir.

Erlang/OTP

Erlang/OTP (en adelante 'Erlang') es un lenguaje de programación funcional orientado a la concurrencia en sistemas distribuidos. Las siglas OTP hacen referencia a '*Open Telecom Platform*', y son un conjunto de librerías y principios de diseño para los programas Erlang. OTP cuenta también con su propia base de datos distribuida, aplicaciones que proporcionan interfaces para interactuar con otros lenguajes y herramientas de *debugging* y control de versiones.

Fue desarrollado por Ericsson y Ellemtel ComputerScience Laboratories en 1986 (y cedido como software de código abierto en 1988) por la necesidad de construir grandes sistemas distribuidos en tiempo real y escalables dentro del ámbito empresarial.

Un sistema de tiempo real es aquel que garantiza que la respuesta resultante de una petición se realice en un plazo de tiempo determinado.

Cuando hablamos de la escalabilidad de un sistema nos referimos a su capacidad de incrementar la cantidad de operaciones concurrentes manteniendo un nivel de servicio aceptable ante un aumento de la demanda.

Erlang trata la concurrencia a través de los procesos, dejando al programador definir cuáles de ellos se ejecutarán secuencialmente y cuáles en paralelo. Estos se comunicarán entre ellos únicamente mediante el paso de mensajes, lo cual facilita el desarrollo de aplicaciones distribuidas.

A parte de las ya mencionadas, otras características de Erlang son:

- Tolerante a fallos: es la propiedad de un sistema que le permite seguir funcionando a pesar de que uno o varios de sus componentes fallen. Erlang cuenta además con mecanismos para detectar errores en tiempo de ejecución.
- Sistema '*non-stop*': dispone de primitivas para reemplazar el código de un sistema en ejecución permitiendo correr al mismo tiempo versiones de código anteriores. Esto supone una gran ventaja para aquellos sistemas de tiempo real que no pueden ser interrumpidos para realizar actualizaciones, como por ejemplo sistemas de control de tráfico aéreo.
- Gestión de memoria: dispone de un colector de basura en tiempo real que libera los datos de la memoria que no están siendo utilizados. Esto reduce significativamente los errores asociados a la gestión de memoria.
- Integración: Erlang puede utilizar con facilidad programas escritos en otros lenguajes de programación.

Erlang incluye su propio sistema de ejecución, una máquina virtual denominada BEAM (*Bogdan's Erlang Abstract Machine)*, al igual que otros lenguajes como pueden ser Java, NodeJS, o Ruby. Esta máquina se basa en los registros como unidad operacional, los cuales pueden contener cualquier término Erlang (como por ejemplo un entero o una tupla).

Todas estas ventajas que proporciona Erlang no han pasado desapercibidas, pues es utilizado por un gran número de compañías de relevancia mundial [10]. Como, por ejemplo: Amazon, Facebook, WhatsApp, Yahoo o Cisco, de entre una larga lista.

Elixir

Elixir es un lenguaje funcional escrito sobre Erlang que también se ejecuta en la máquina virtual BEAM. Nace como proyecto open-source en 2011 por iniciativa de José Valim, aunque el resultado final es fruto del trabajo colaborativo de más de 700 contribuyentes.

Este lenguaje hereda todas las características y parte de la sintaxis de Erlang, pudiendo hacer uso de sus librerías y viceversa. No hay nada que puedas hacer en uno de estos lenguajes que no se pueda hacer en el otro, con lo cual, podríamos ver Elixir como una actualización de Erlang.

A continuación, listaremos algunas de las mejoras que nos proporciona Elixir frente a Erlang:

- Simplificación del código: reduce en gran medida la cantidad de código a escribir para implementar la misma función.
- Adición de "azúcar sintáctico" para mejorar la comprensión del código.
- Sintaxis provista para trabajar con estructuras de datos.
- Reasignación de variables: en Erlang no se pueden asignar variables más de una vez.
- Mejora en el tratamiento de cadenas de texto.
- Soporte para manipular Unicode
- 'Mix': herramienta que nos facilita tareas como crear aplicaciones y librerías, gestionar las dependencias y compilar y probar código.
- 'Hex': gestor de paquetes.

Como resultado de estas mejoras obtenemos un código más fácil de leer y, por tanto, de mantener.

```
1 -module(sum_server).
2 -behaviour(gen_server).
3
4 -export([
5   start/0, sum/3,
6   init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,
7   code_change/3
8 ]).
9
10 start() -> gen_server:start(?MODULE, [], []).
11 sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).
12
13 init(_) -> {ok, undefined}.
14 handle_call({sum, A, B}, _From, State) -> {reply, A + B, State}.
15 handle_cast(_Msg, State) -> {noreply, State}.
16 handle_info(_Info, State) -> {noreply, State}.
17 terminate(_Reason, _State) -> ok.
18 code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

Figura 2.4 - Ejemplo de código escrito en Erlang que define un proceso de servidor que suma dos números [11]

```
1 defmodule SumServer do
2   use GenServer
3
4   def start do
5     GenServer.start(__MODULE__, nil)
6   end
7
8   def sum(server, a, b) do
9     GenServer.call(server, {:sum, a, b})
10   end
11
12   def handle_call({:sum, a, b}, _from, state) do
13     {:reply, a + b, state}
14   end
15 end
```

Figura 2.5 - Código análogo al de la figura anterior escrito en Elixir [11]

Elixir es utilizado en multitud de páginas web y aplicaciones [12], como por ejemplo Discord o Pinterest. La primera mencionada lo lleva utilizando desde su creación, mientras que Pinterest dio el salto a Elixir en 2014, consiguiendo reducir a una décima parte el código de su back-end y a la mitad los servidores usados. Otros ejemplos relevantes serían PepsiCo, *Financial Times, Toyota Connected* y Moz.

2.4.2. Características del framework

Phoenix es un framework de desarrollo web escrito en Elixir que implementa el patrón de diseño modelo-vista-controlador (MVC) en el lado del servidor. Como ya hemos mencionado, se ejecutará en una máquina virtual Erlang BEAM por lo que se beneficiará de todas sus cualidades. Esto hace que sea un framework ideal para el desarrollo del back-end de sistemas concurrentes, mantenibles, escalables, tolerantes a fallos y con muy buen rendimiento.

Otra característica relevante que nos ofrece Phoenix son los '*channels'*, los cuales permiten establecer conexiones bidireccionales a tiempo real entre millones de usuarios. Su funcionamiento reside en los '*topics'* o temas, a los cuales se suscriben los usuarios que se conectan al servidor para acceder a una determinada sala de chat o recibir las actualizaciones de un servicio de noticias, por ejemplo. A diferencia de las conexiones HTTP los channels permiten conexiones de larga duración, dedicando un proceso ligero a cada una de ellas en la máquina BEAM. Estos trabajan en paralelo y cada uno mantiene su propio estado.

Aun con todas estas ventajas que nos proporciona Phoenix, no cuenta con la popularidad que sí tienen sus competidores directos, como por ejemplo 'Ruby on Rails' o 'Django', los cuales son bastante más utilizados entre la comunidad de desarrolladores. Esto se debe principalmente a la escasez de software de terceros y al reducido tamaño de la comunidad de Phoenix. Sin embargo, Phoenix ofrece un rendimiento superior al de los otros frameworks mencionados y una mejor gestión de la concurrencia.

2.4.3. Componentes

A continuación, procederemos a explicar las partes que componen nuestra aplicación Elixir utilizando el framework Phoenix, intentando lograr así una mejor comprensión de su funcionamiento.

Antes de empezar, debemos tener claro el concepto de 'Plug', componente que constituye la base de la capa HTTP de Phoenix. Plug es una especificación para la composición de módulos entre aplicaciones web, además de una capa de abstracción para los adaptadores de conexión de distintos servicios web. Su función consiste en unificar los conceptos de 'petición' y 'respuesta' en uno único: la conexión.

Los Plugs pueden ser de dos tipos: funciones o módulos. Para que una función actúe como un Plug deberá recibir como primer parámetro una estructura de tipo '*Plug.connection{}'* y devolverla con las modificaciones pertinentes, como pueden ser el estado de la respuesta o incluso la adición de un objeto JSON. Si lo que queremos es que un módulo actúe como Plug deberá implementar las funciones '*init*' y '*call*'.

Sabiendo lo que es un Plug en Phoenix y para qué sirve es fácil deducir que para establecer las distintas conexiones entre nuestro cliente y nuestro servidor hemos creado distintos Plugs (en concreto de tipo función). Estos serán vistos en detalle en el apartado 'Implementación'.

Modelo - Vista - Controlador

El modelo-vista-controlador es un patrón de arquitectura de software altamente conocido entre la comunidad de desarrolladores. Este patrón separa los datos de la aplicación y la lógica de negocio, su interfaz y la lógica de control de control en estos tres componentes.

Modelo: contiene la representación de los datos y la lógica de negocio de la aplicación. También se definirán los mecanismos de persistencia.

Vista: define la representación de estos datos al usuario y la forma que tendrá de interactuar con ellos, dando lugar a la interfaz de usuario.

Controlador: actuará como intermediario entre el modelo y la vista, gestionando el flujo de información y realizando las transformaciones necesarias sobre ella. Será el encargado de responder a los distintos eventos.

Phoenix implementa esta arquitectura de una manera simple y eficaz, separando en directorios distintos los modelos de sus vistas y controladores. Al definir un modelo en Phoenix podremos determinar directamente su esquema en base de datos, lo cual simplifica en gran medida el tratamiento de los datos. También se podrá realizar aquí el filtrado, la validación, casting y definir restricciones para los datos del modelo mediante el uso de '*changesets'*.

La principal función de las vistas en Phoenix será generar el cuerpo de las respuestas a las peticiones HTTP. Estarán fuertemente ligadas a las '*templates'* o plantillas en las que se define el formato de la respuesta. Una plantilla puede ser por ejemplo una página web o la definición de un objeto JSON. Para nuestra aplicación no ha sido necesario crear ni vistas ni plantillas, pues el objeto JSON guarda relación directa con su representación en la base de datos local del cliente, por lo que no es necesario realizar ninguna adaptación.

Los controladores en Phoenix determinan las acciones a realizar ante las distintas peticiones HTTP mediante el uso de funciones. De esta forma se agrupará la funcionalidad común en un mismo módulo. En la siguiente sección veremos mediante un ejemplo cómo se asocian los controladores con las peticiones al servidor.

Router

El enrutador es el eje central de las aplicaciones Phoenix. Sirve para enlazar peticiones HTTP con las acciones correspondientes de los controladores. Este enrutador se define en '*router.ex*', archivo en el que se establecerán las tuberías *(pipelines)* a utilizar y su alcance *(scope)*.

La tubería por defecto es '*:browser'* (navegador), pero el desarrollador puede crear las que considere oportunas. En nuestra aplicación no utilizaremos la tubería por defecto, al no necesitar generar ninguna página web en el navegador; en su lugar utilizaremos la tubería '*:api'* . Dentro de las pipelines se indicarán los distintos Plugs que utilizarán.

En el scope será donde se definan las rutas, la tubería a utilizar y el controlador y la función asociados a cada petición HTTP. También se establecerá el carácter de la petición: *GET, POST, UPDATE* y/o *DELETE.*

A continuación podremos ver un ejemplo real de un archivo *router.ex,* donde veremos cómo se relacionan estos elementos previamente descritos. Este enrutador será el resultado de generar una aplicación Phoenix llamada 'Ejemplo' y tan solo se ha modificado para borrar una parte comentada en la que te indica que puedes crear scopes alternativos para tus tuberías.

```
defmodule EjemploWeb.Router do
  use EjemploWeb, :router

  pipeline :browser do
    plug :accepts, ["html"]
    plug :fetch_session
    plug :fetch_flash
    plug :protect_from_forgery
    plug :put_secure_browser_headers
  end

  pipeline :api do
    plug :accepts, ["json"]
  end

  scope "/", EjemploWeb do
    pipe_through :browser

    get "/", PageController, :index
  end

  if Mix.env() in [:dev, :test] do
    import Phoenix.LiveDashboard.Router
    scope "/" do
      pipe_through :browser
      live_dashboard "/dashboard", metrics: EjemploWeb.Telemetry
    end
  end
end
```

Tabla 2.1 - Archivo 'router.ex' de un proyecto de ejemplo

Como podemos observar hay definidas dos tuberías, *':browser'* y *':api',* y tan solo un scope para la primera. En las tuberías se establecen los Plugs por defecto que van a necesitar, indicando el formato en el que se devolverá el resultado (HTML para el navegador y JSON para la tubería api) entre otras cosas.

En el scope lo primero que se define es la ruta, '/' en este caso´, pero que podrá ser modificada (al igual que todo lo demás) según las necesidades de nuestra aplicación. Lo siguiente que se establece es la tubería a utilizar con la función '*pipe_through',* incluso se puede definir que el scope sea para varias pipelines utilizando corchetes: '*pipe_through [:browser, :api]' ,* para nuestro ejemplo.

Una vez definida la ruta base y la/s tubería/s a utilizar se establecen las peticiones HTTP y las acciones a realizar para cada una de ellas. En este caso solo tendremos una petición GET a la ruta '/' y que utilizará la función '*index'* del controlador '*PageController'.* Phoenix también nos ofrece la opción de agrupar todos los tipos de petición HTTP para un determinado modelo con la función '*resources',* donde podremos también especificar si queremos tan solo alguno de ellos. Por ejemplo: con *'resources "/pages", PageController, only[:create, :delete, :index]'* indicaremos que para las peticiones *POST, DELETE* y *GET* para el modelo '*pages'* utilizaremos su controlador y accederemos a las funciones indicadas entre los corchetes. Para utilizar este recurso las funciones creadas tendrán que tener el nombre reservado para ellas y no el que les queramos dar.

Las últimas líneas de este ejemplo definirán el scope a utilizar si la aplicación se ejecuta en el entorno '*dev'.* Y es que Phoenix nos permite ejecutar nuestra aplicación en tres entornos definidos por defecto: desarrollo, producción y pruebas (*'dev', 'prod'* y *'test',* respectivamente). Cada uno de ellos tendrá asociado su fichero de configuración, y, como no podía ser de otra manera en Phoenix, también se permite al desarrollador crear los entornos que considere oportunos.

Endpoint

Un endpoint (o punto de acceso final) es un tipo de nodo de red de comunicación, es decir, un punto de conexión de varios elementos que confluyen en el mismo lugar: en este caso nuestro servidor. Para ello, proporciona una interfaz que definirá los requisitos operacionales y de seguridad que se deberán seguir para establecer las conexiones.

Al crear una aplicación Phoenix se generará un endpoint automáticamente en el archivo *'endpoint.ex'.* Este tendrá tres responsabilidades:

- Proporcionar un envoltorio *('wrapper')* para iniciar y parar las conexiones dentro del árbol de supervisión de Phoenix.
- Establecer la pipeline-plug inicial por la que pasarán las peticiones.
- Hospedar la configuración web específica de la aplicación.

2.5. Herramientas utilizadas

En este apartado se describirá el conjunto de herramientas y software utilizado para el desarrollo de la aplicación. Para una mejor clasificación de ellas, se dividirán en dos grupos: las utilizadas para realizar la parte del cliente y aquellas usadas para desarrollar el servidor.

2.5.1. Cliente

Android Studio 4.2.1

Se trata de un IDE (entorno de desarrollo integrado) proporcionado por Google para el desarrollo específico de aplicaciones Android en Java y Kotlin. Fue anunciado el 16 de mayo de 2013 en el Google IO y desde entonces reemplazó a Eclipse como el IDE oficial de desarrollo de aplicaciones Android. Este IDE contiene características que lo hacen muy práctico, como pueden ser: soporte para construcciones basadas en Gradle, renderización en tiempo real, consola de desarrollador y herramientas como Android Virtual Device. Por todo esto y por la fiabilidad que ofrece su carácter oficial, se ha elegido este IDE.

AVD (Android Virtual Device) Manager

Emulador nativo de Android Studio que ofrece distintos terminales Pixel para probar las aplicaciones. Esto ayuda a ver cómo quedaría la aplicación en terminales con distintos tamaños de pantalla. En concreto se han utilizado los terminales virtuales Pixel 3 XL, Pixel 4 y Pixel 5; todos ellos utilizando Android en su API 28.

Terminal Oneplus Nord (Android 11)

Utilizado para realizar pruebas de la aplicación en un terminal físico.

Mozilla Firefox 89.0.2

Navegador web de código abierto utilizado para realizar las distintas consultas a las APIs. También se le instaló el *add-on* 'Json LITE' para facilitar la lectura de las respuestas.

2.5.2. Servidor

Oracle VM VirtualBox 6.1.22

Software de virtualización altamente conocido desarrollado por Oracle Corporation. Este programa nos permite crear y ejecutar máquinas virtuales. El servidor de nuestro sistema fue alojado en una máquina virtual Linux, con la distribución Ubuntu Mate 20.04.2.

Atom 1.57.0

Editor de texto utilizado para desarrollar el servidor. Se ha elegido por la familiaridad con él y sus múltiples paquetes desarrollados por la comunidad que aportan herramientas muy útiles para todo tipo de desarrollo software. En concreto se ha utilizado el paquete *'ide-elixir',* aportado por el usuario Jake Beker y que reconocerá

la sintaxis de cualquier código escrito en elixir estableciendo un código de colores para ella, además de aportarnos opciones de autocompletado. Esto nos facilitará en gran medida el desarrollo de aplicaciones en Elixir.

Node.js v10.19.0

Se trata de un entorno en tiempo de ejecución multiplataforma para la capa del servidor. Se basa en JavaScript y es de código abierto. En concreto utilizaremos su gestor de paquetes, npm ('node package manager'), para la instalación de dependencias.

Inotify-tools v3.14-8

Necesario para la ejecución de Phoenix en un entorno Linux. Esta herramienta nos permitirá realizar cambios en el servidor en tiempo de ejecución.

PostgreSQL 12.7

Sistema gestor de base de datos empleado por defecto para cualquier aplicación Phoenix. Se ha utilizado para hospedar nuestra base de datos y poder ver los cambios que las peticiones a nuestro servidor originaban en ella.

Mozilla Firefox 89.0.1

Utilizado para las primeras pruebas de toma de contacto con el framework Phoenix. Si bien nuestra aplicación finalmente no generará ninguna página web, en un primer momento sí que se hicieron para lograr un mejor entendimiento del framework

Capítulo 3

Análisis

3.1. Presentación del sistema

El sistema permite a los usuarios finales crear y modificar listas de películas (en lo *sucesivo 'watchlists')* y compartirlas con otros usuarios mediante una interfaz implementada en dispositivos móviles Android. De esta forma, el sistema trata de ser una herramienta para compartir preferencias cinematográficas.

3.2. Captura de requisitos

A la hora de desarrollar cualquier proyecto software el primer paso a realizar es la captura de requisitos. Estos se dividen en requisitos funcionales y no funcionales, y nos indicarán qué debe hacer nuestra aplicación y bajo qué restricciones, respectivamente.

A continuación, procederemos a enumerar los distintos requisitos que se han establecido para nuestra aplicación en base a los objetivos que se busca conseguir y a las líneas generales que siguen la mayoría de las aplicaciones, con el fin de homogeneizar la experiencia de usuario y facilitar su aprendizaje.

3.2.1. Requisitos funcionales

Los requisitos funcionales describen los servicios que nuestro sistema debe incorporar y establecen las salidas que devuelve ante determinadas entradas.

RF1 - Registro

- **RF1.1 -** El sistema deberá permitir a los usuarios registrarse indicando su email, nombre de usuario y contraseña.

- **RF1.2 -** El sistema deberá validar que el email y nombre de usuario introducidos en el registro no se correspondan con los de un usuario ya registrado.
- **RF1.3 -** El sistema deberá comprobar que la contraseña elegida por el usuario en el registro tenga al menos 8 caracteres.
- **RF1.4 -** El sistema deberá mostrar un mensaje de error al usuario si los datos introducidos en el formulario de registro no son válidos, indicando la naturaleza del error.

RF2 - Gestión de la sesión

- **RF2.1 -** El sistema deberá permitir a los usuarios iniciar sesión con su email y contraseña.
- **RF2.2 -** El sistema deberá mantener la sesión del usuario iniciada mientras este no indique lo contrario, aunque se finalice la aplicación.
- **RF2.3 -** El sistema deberá cargar los datos del usuario (sus watchlists y suscripciones) del servidor en el cliente cuando este inicie sesión.
- **RF2.4 -** El sistema deberá permitir al usuario finalizar la sesión.

RF3 - Búsqueda de películas

- **RF3.1 -** El sistema deberá permitir a los usuarios buscar películas, por su título, en una base de datos y mostrarles la información.

RF4 - Watchlists personales

- **RF4.1 -** El sistema deberá permitir a los usuarios crear watchlists, públicas o privadas, con un nombre
  RF4.2 - de entre uno y 16 caracteres en formato UTF8.
- **RF4.3 -** El sistema deberá permitir a los usuarios borrar watchlists de su creación.
- **RF4.4 -** El sistema deberá permitir a los usuarios añadir películas a sus watchlists creadas.
- **RF4.5 -** El sistema deberá mostrar las películas de una watchlist ordenadas por su año de lanzamiento.
- **RF4.6 -** El sistema deberá permitir a los usuarios crear watchlists en el momento de añadir una película.
- **RF4.7 -** El sistema deberá permitir a los usuarios retirar películas de una determinada watchlist de su creación.
- **RF4.8 -** El sistema deberá proporcionar una película elegida aleatoriamente de las watchlists personales cuando el usuario lo requiera.
- **RF4.9 -** El sistema deberá almacenar en la base de datos del servidor las watchlists creadas por los usuarios y las películas asociadas a ellas.
- **RF4.10 -** El sistema deberá almacenar en una base de datos local en el cliente las watchlists de los usuarios y sus películas asociadas.
- **RF4.11 -** El sistema deberá actualizar la base de datos del servidor cuando se realice una modificación, o borrado, en cualquier watchlist.
- **RF4.12 -** El sistema deberá actualizar la base de datos del cliente cuando se realice alguna modificación o borrado de sus watchlists personales.

RF5 - Watchlists públicas

- **RF5.1 -** El sistema deberá permitir a los usuarios buscar watchlists públicas por su nombre o por el nombre de usuario de su creador.
- **RF5.2 -** El sistema deberá permitir a los usuarios suscribirse a watchlists públicas.
- **RF5.3 -** El sistema no deberá permitir a los usuarios suscribirse a sus propias watchlists públicas.
- **RF5.4 -** El sistema deberá permitir a los usuarios dar de baja sus suscripciones a watchlists públicas.
- **RF5.5 -** El sistema deberá actualizar la información de las watchlists públicas suscritas cuando su creador las modifique.
- **RF5.6 -** El sistema deberá mostrar el nombre de usuario del creador de la watchlist pública.
- **RF5.7 -** El sistema deberá almacenar en la base de datos del servidor las suscripciones de los usuarios a las watchlists públicas.
- **RF5.8 -** El sistema deberá borrar de la base de datos del servidor las suscripciones de los usuarios cuando se den de baja.
- **RF5.9 -** El sistema deberá almacenar en la base de datos local del cliente las suscripciones del usuario.
- **RF5.10 -** El sistema deberá borrar de la base de datos local del cliente las suscripciones que se den de baja.
- **RF5.11 -** El sistema no permitirá a los usuarios suscribirse a sus propias watchlists públicas.

3.2.2. Requisitos no funcionales

Los requisitos no funcionales definen las limitaciones que afectan a nuestro sistema, además de propiedades emergentes del mismo tales como su rendimiento, seguridad y usabilidad, entre otras.

RNF1 - Seguridad

- **RNF1.1 -** El sistema solo permitirá el acceso a usuarios registrados.
- **RNF1.2 -** El sistema no permitirá a los usuarios acceder a información sensible, como el email o cualquier otra información que pueda ser añadida en otras versiones, de otros usuarios.
- **RNF1.3 -** El sistema encriptará la contraseña del usuario con una función hash.
- **RNF1.4 -** El sistema no requerirá de permisos de ningún tipo en el dispositivo.
- **RNF1.5 -** Toda la información referente a un usuario podrá ser borrada del servidor a petición suya.

RNF2 - Producto

- **RNF2.1 -** El sistema deberá funcionar en dispositivos móviles con el sistema operativo Android en su versión 9 como mínimo.
- **RNF2.2 -** El instalable de la aplicación no superará los 10Mb.

RNF3 - Interfaz y usabilidad

- **RNF3.1 -** El sistema constará de una interfaz simple y atractiva evitando la sobrecarga de elementos en pantalla.
- **RNF3.2 -** El sistema mostrará una guía introductoria para facilitar el entendimiento de su uso tras un registro satisfactorio.

RNF4 - Rendimiento

- **RNF4.1 -** Los tiempos de respuesta del servidor en un estado normal no deben superar los dos segundos.
- **RNF4.2 -** El servidor debe ser escalable ante un posible aumento de usuarios.

3.3. Casos de uso

En el contexto de la ingeniería del software, un caso de uso es la descripción de una secuencia de acciones que un sistema ejecuta, en respuesta a la interacción de un usuario o actor, arrojando un resultado observable y valioso para el mismo.

Para nuestro sistema hemos distinguido dos tipos de actores: registrados y no registrados, y se han establecido distintos casos de uso para ellos que capturan el detalle de los requisitos funcionales definidos para el sistema. Los casos de uso resultantes son los siguientes:

Figura 3.1 - Caso de uso para el actor 'usuario no registrado'

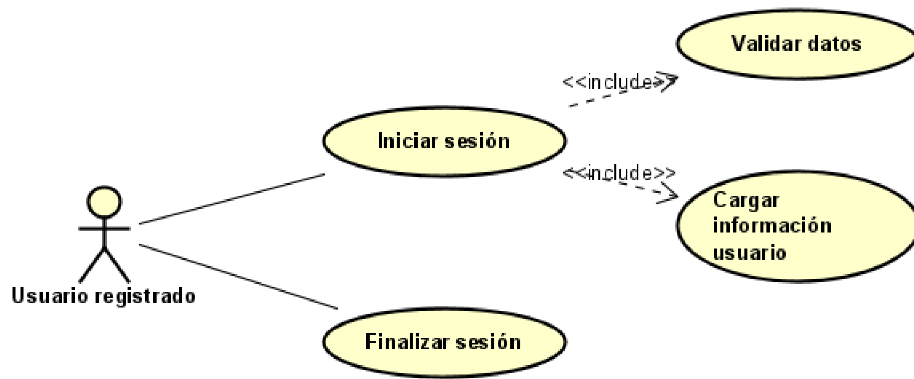

Figura 3.2 - Casos de uso relacionados con el estado de la sesión del actor 'usuario registrado'

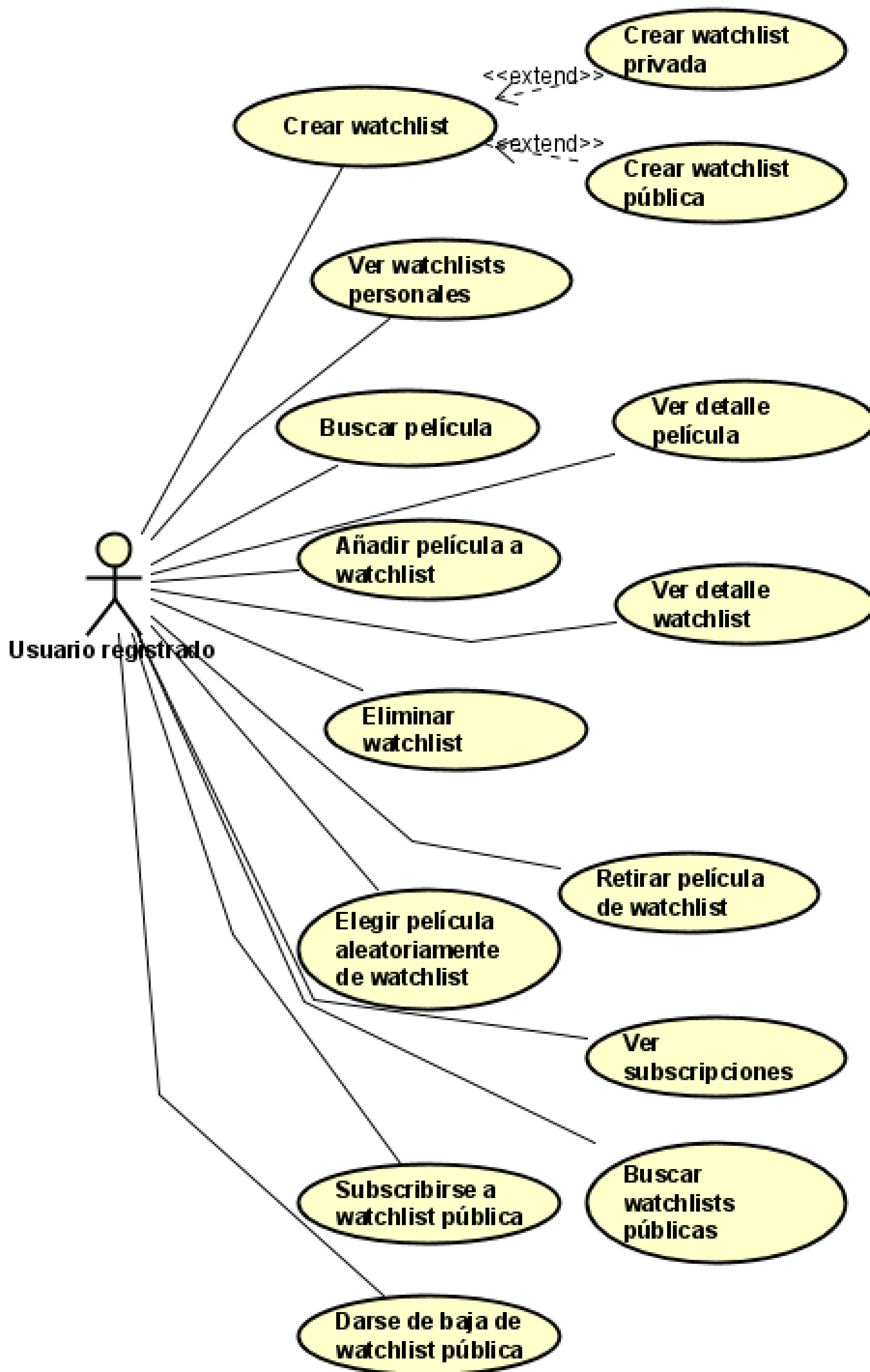

Figura 3.3 - Casos de uso que detallan el resto de la funcionalidad del sistema para el actor 'Usuario registrado'

3.3.1. Descripción de los casos de uso

A continuación, procederemos a detallar cada caso de uso mencionado en los diagramas anteriores.

UC1 – Registro

Descripción	Proceso de un usuario para darse de alta en el sistema
Actor	Usuario no registrado
Precondiciones	-
Postcondiciones	El usuario debe quedar registrado en la base de datos del servidor.
Secuencia base	1. El usuario pulsa el link "registrarse" de la pantalla inicial de login. 2. El actor introduce un email válido sin registrar en el sistema. 3. El actor introduce un nombre de usuario de hasta 12 caracteres. 4. El actor introduce la contraseña y su verificación, con un mínimo de 8 caracteres. 5. El actor pulsa el botón de registrarse. 6. El sistema valida los datos. 7. El sistema muestra la pantalla 'Watchlists personales'. 8. El sistema muestra un cuadro de diálogo con indicaciones para el uso de la aplicación.
Excepciones	1. La validación de datos no es satisfactoria y el sistema muestra el mensaje de error correspondiente.

Tabla 3.1 – UC1: Registro

UC2 - Inicio de sesión

Descripción	Proceso que permite al usuario acceder al sistema y a sus datos.
Actor	Usuario registrado
Precondiciones	El actor debe tener una cuenta activa en el sistema.
Postcondiciones	-
Secuencia base	1. El actor introduce su email. 2. El actor introduce su contraseña. 3. El sistema valida los datos.

	4. El sistema realiza una serie de peticiones al servidor y obtiene los datos referentes a las watchlists y suscripciones del usuario. 5. El sistema muestra la pantalla 'Watchlists personales' con los datos obtenidos del servidor.
Excepciones	1. La validación de datos no es satisfactoria y el sistema muestra el mensaje de error correspondiente.

Tabla 3.2 - UC2: Inicio de sesión

UC3 - Finalizar sesión

Descripción	Proceso que permite al actor finalizar la sesión en la aplicación.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	La base de datos local del cliente es borrada.
Secuencia base	1. El actor accede a la pantalla 'Social WatchLists' 2. El actor pulsa el botón de la esquina superior derecha con el icono que representa 'desconexión. 3. El sistema muestra un cuadro de diálogo para confirmar la acción. 4. El actor pulsa 'Aceptar'. 5. El sistema borra los datos de sesión del usuario y muestra la pantalla de login.
Excepciones	1. El actor pulsa 'Cancelar' y el caso de uso termina sin efecto.

Tabla 3.3 - UC3: Finalizar sesión

UC4 - Crear watchlist privada

Descripción	Proceso que permite al actor crear una lista de películas a la que solo él podrá acceder.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	La watchlist creada por el actor queda registrada en la base de datos del servidor y en la del cliente.
Secuencia base	1. El actor accede a la pantalla (por defecto) 'Watchlists personales'. 2. El actor pulsa en el botón '+' situado en la esquina inferior derecha. 3. El sistema muestra un cuadro de diálogo con un campo para el nombre de la lista y un checkbox que indica su naturaleza (pública o privada).

	4. El actor introduce un nombre de entre 1 y 16 caracteres. 5. El actor pulsa 'Aceptar'. 6. El sistema finaliza el cuadro de diálogo mostrando en la pantalla 'Watchlists personales' la nueva lista creada.
Excepciones	1. El actor accede a la pantalla 'Añadir película a watchlist' y el caso de uso sigue la secuencia base. 2. El actor no ha introducido ningún nombre para la lista y no surte ningún efecto al pulsar 'Aceptar'. 5'. El actor pulsa 'Cancelar' y el caso de uso finaliza sin efecto.

Tabla 3.4 - UC4: Crear watchlist privada

UC5 - Crear watchlist pública

Descripción	Proceso que permite al actor crear una lista de películas de acceso público.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	La watchlist creada por el actor queda registrada en la base de datos del servidor y en la del cliente.
Secuencia base	1. El actor accede a la pantalla (por defecto) 'Watchlists personales'. 2. El actor pulsa en el botón '+' situado en la esquina inferior derecha. 3. El sistema muestra un cuadro de diálogo con un campo para el nombre de la lista y un checkbox que indica su naturaleza (pública o privada). 4. El actor introduce un nombre de entre 1 y 16 caracteres. 5. El actor pulsa el checkbox situado a la derecha de la etiqueta 'Pública'. 6. El actor pulsa 'Aceptar'. 7. El sistema finaliza el cuadro de diálogo mostrando en la pantalla 'Watchlists personales' la nueva lista creada.
Excepciones	1. El actor accede a la pantalla 'Añadir película a watchlist' y el caso de uso sigue la secuencia base. 2. El actor no ha introducido ningún nombre para la lista y no surte ningún efecto al pulsar 'Aceptar'. 6'. El actor pulsa 'Cancelar' y el caso de uso finaliza sin efecto.

Tabla 3.5 - UC4: Crear watchlist pública

UC6 - Ver watchlists personales

Descripción	Proceso que permite al actor ver las listas creadas por él.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	-
Secuencia base	1. El actor accede a la pantalla 'Watchlists personales'. 2. El sistema carga en pantalla la información referente a las watchlists del actor.
Excepciones	1. El actor no tiene creada ninguna watchlist y el sistema lo notifica con un mensaje en pantalla.

Tabla 3.6 - UC6: Ver watchlists personales

UC7 - Buscar película

Descripción	Proceso que permite al actor buscar películas por su título para ver su información o añadirla a alguna de sus listas.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	-
Secuencia base	1. El actor accede a la pantalla 'Buscar película'. 2. El actor introduce un término en la barra de búsqueda superior de al menos dos caracteres. 3. El actor pulsa el botón de buscar situado en la parte derecha de la barra de búsqueda o el 'intro' de su teclado. 4. El sistema realiza una serie de peticiones a las APIs de consulta previamente mencionadas para mostrar hasta 60 resultados.
Excepciones	1. La petición introducida por el actor es menor de dos caracteres y el caso de uso queda sin efecto. 2. El sistema no encuentra ninguna película para el termino introducido y notifica al actor con un mensaje en pantalla.

Tabla 3.7 - UC7: Buscar película

UC8 - Ver detalle película

Descripción	Proceso que permite al actor visualizar información de una película.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe encontrarse en una pantalla que muestre películas, las cuales pueden ser: 'Detalle de watchlist' y 'Buscar película'.
Postcondiciones	-
Secuencia base	1. El actor pulsa en la tarjeta que representa la película de la que desea obtener información. 2. El sistema muestra la pantalla 'Detalle de película' con toda la información que se dispone de ella.
Excepciones	-

Tabla 3.8 - UC8: Ver detalle película

UC9 - Añadir película a watchlist

Descripción	Proceso que permite al actor añadir una película a una de sus listas personales.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe encontrarse en la pantalla 'Detalle película' originada a raíz de una búsqueda de película o desde la pantalla 'Detalle watchlist pública'.
Postcondiciones	La adición de la película a la watchlist debe quedar reflejada en la base de datos del servidor y en la del cliente.
Secuencia base	1. El actor pulsa el botón '+' situado en la esquina inferior derecha de la pantalla 'Detalle película'. 2. El sistema muestra al usuario las watchlists creadas por él dándole la opción de crear las listas nuevas que desee. 3. El actor pulsa en la tarjeta que representa la lista a la que desea añadir la película. 4. El sistema añade la película a la lista y regresa a la pantalla anterior ('Detalle película').
Excepciones	1. El actor pulsa el botón '+' situado en la esquina inferior derecha y comienza el caso de uso UC4 o UC5.

Tabla 3.9 - UC9: Añadir película a watchlist

UC10 - Ver detalle watchlist

Descripción	Proceso que permite al actor ver las películas pertenecientes a una watchlist.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe tener creada alguna watchlist • El actor debe encontrarse en una pantalla que muestre listas, las cuales son: 'WatchLists personales' y 'Social WatchLists'.
Postcondiciones	-
Secuencia base	1. El actor pulsa en la tarjeta que representa la watchlist de la que quiere obtener más información. 2. El sistema muestra la pantalla 'Detalle watchlist' con las películas que se encuentren en ella (si las hubiera).
Excepciones	-

Tabla 3.10 - UC10: Ver detalle watchlist

UC11 - Retirar película de watchlist

Descripción	Proceso que permite al actor retirar películas de una lista creada por él.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe tener creada alguna lista con películas en ella.
Postcondiciones	La retirada de la película de la watchlist debe quedar reflejada en la base de datos del servidor y en la del cliente.
Secuencia base	1. El actor accede a la pantalla 'Detalle watchlist' mediante el caso de uso UC9 y desde la pantalla 'WatchLists personales'. 2. El actor mantiene pulsado sobre la tarjeta que representa la película que desea retirar de la lista. 3. El sistema muestra un cuadro de diálogo para confirmar la acción. 4. El actor pulsa 'Sí'. 5. El sistema vuelve a la pantalla 'Detalle watchlist' en la que ya no se mostrará esta película.
Excepciones	1. El actor pulsa 'No' y el caso de uso queda sin efecto.

Tabla 3.11 - UC11: Retirar película de watchlist

UC12 - Eliminar watchlist

Descripción	Proceso que permite al actor borrar una lista creada por él.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe tener creada alguna watchlist.
Postcondiciones	La eliminación de la watchlist debe quedar reflejada en la base de datos del servidor y en la del cliente.
Secuencia base	1. El actor accede a la pantalla 'WatchLists personales'. 2. El actor mantiene pulsado sobre la tarjeta que representa la lista que desea borrar. 3. El sistema muestra un cuadro de diálogo para confirmar la acción. 4. El actor pulsa 'Sí'. 5. El sistema vuelve a la pantalla 'WatchLists Personales' en la que ya no se mostrará esta lista.
Excepciones	1. El actor pulsa 'No' y el caso de uso queda sin efecto.

Tabla 3.12 - UC12: Eliminar watchlist

UC13 - Elegir película aleatoriamente de watchlist

Descripción	Proceso que permite al actor obtener aleatoriamente una de las películas de dentro de una lista.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe tener creada alguna lista con al menos dos películas en ella.
Postcondiciones	-
Secuencia base	1. El actor accede a la pantalla 'Detalle watchlist' mediante el caso de uso UC9 y desde la pantalla 'WatchLists personales'. 2. El actor pulsa el botón situado en la esquina inferior derecha con el icono que representa 'aleatorio'. 3. El sistema muestra la pantalla 'Detalle película' con la información de una de las películas que se encuentran dentro de la lista elegida mediante números aleatorios.
Excepciones	-

Tabla 3.13 - UC13: Elegir película aleatoriamente de watchlist

UC14 - Ver suscripciones.

Descripción	Proceso que permite al actor ver las listas a las que se ha suscrito.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	•
Secuencia base	1. El actor accede a la pantalla 'Social WatchLists'. 2. El sistema realiza una petición al servidor para obtener las watchlists de las suscripciones del usuario. 3. El sistema carga la información en pantalla.
Excepciones	1. El sistema no detecta suscripciones activas del actor y le notifica con un mensaje en pantalla.

Tabla 3.14 - UC14: Ver suscripciones.

UC15 - Buscar watchlists públicas

Descripción	Proceso que permite al actor buscar watchlists públicas, creadas por el resto de la comunidad, por su nombre o por el nombre de usuario de su creador.
Actor	Usuario registrado
Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	-
Secuencia base	1. El actor accede a la pantalla 'Social WatchLists'. 2. El actor introduce un término en la barra de búsqueda superior. 3. El sistema envía una petición al servidor para obtener los resultados de la búsqueda. 4. El sistema muestra al actor los resultados de la búsqueda.
Excepciones	1. El sistema no encuentra coincidencias con el término introducido y notifica al actor con un mensaje en pantalla.

Tabla 3.15 - UC15: Buscar watchlists públicas

UC16 - Suscribirse a watchlist pública

Descripción	Proceso que permite al actor suscribirse a una watchlist pública para poder acceder a sus datos y ver las modificaciones que realice su autor.
Actor	Usuario registrado

Precondiciones	El actor debe estar previamente logueado en la aplicación.
Postcondiciones	La suscripción del actor a la lista debe quedar reflejado en la base de datos del servidor y del cliente.
Secuencia base	1. El actor realiza la búsqueda de watchlists mediante el caso de uso UC15. 2. El actor accede al detalle de la watchlist mediante el caso de uso UC10 desde la pantalla de los resultados de la búsqueda. 3. El actor pulsa el botón de la esquina inferior derecha con el icono de 'suscribirse'. 4. El sistema muestra un cuadro de diálogo para confirmar la acción. 5. El actor pulsa 'Sí'. 6. El sistema vuelve a mostrar el detalle de la watchlist eliminando el botón de seguir y añadiendo la etiqueta *'Siguiendo'* en la parte superior.
Excepciones	1. La watchlist seleccionada ya se está siguiendo y se muestra como se describe en el paso 6. 3'. La watchlist ha sido creada por el actor y se indica en el campo del autor, además de eliminarse el botón de suscribirse. 2. El actor pulsa 'No' y el caso de uso queda sin efecto.

Tabla 3.16 - UC16: Suscribirse a watchlist pública

UC17 - Darse de baja de watchlist pública

Descripción	Proceso que permite al actor darse de baja de una watchlist a la que se encuentra suscrito.
Actor	Usuario registrado
Precondiciones	• El actor debe estar previamente logueado en la aplicación. • El actor debe estar suscrito a alguna watchlist.
Postcondiciones	La baja del actor a la lista debe quedar reflejado en la base de datos del servidor y del cliente.
Secuencia base	1. El actor accede a sus suscripciones mediante el caso de uso UC14. 2. El actor mantiene pulsado sobre la tarjeta que representa a la lista de la que se quiere dar de baja. 3. El sistema muestra un cuadro de diálogo para confirmar la acción. 4. El actor pulsa 'Sí'. 5. El sistema vuelve a la pantalla 'Social WatchLists' eliminando la lista seleccionada de las suscripciones.
Excepciones	1. El actor pulsa 'No' y el caso de uso queda sin efecto.

Tabla 3.17 - UC17: Darse de baja de watchlist pública

3.4. Modelo de dominio del sistema

En este apartado presentaremos el modelo de dominio del sistema, mostrando sus entidades y la relación establecida entre ellas.

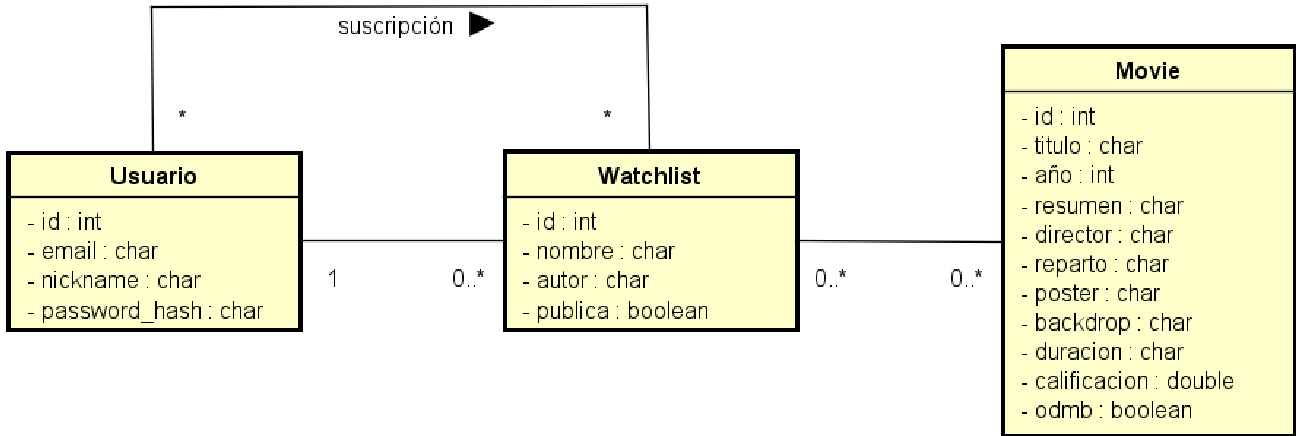

Figura 3.4 - Modelo de dominio del sistema

Como podemos ver, el dominio del sistema estará conformado por tan solo tres entidades: usuario, watchlist y película *('movie')*. Las películas podrán estar añadidas en varias listas, por lo que necesitaremos de una tabla intermedia en nuestra base de datos para establecer las relaciones de pertenencia.

Las suscripciones de los usuarios a las watchlists serán vistas como un tipo de relación entre ellos de tipo muchos a muchos.

Capítulo 4

Diseño

En este capítulo trataremos el diseño de la aplicación. En primer lugar, describiremos el middleware utilizado, tanto para el desarrollo del cliente como del servidor. A continuación, expondremos los diagramas de clases del sistema y analizaremos la estructura de las bases de datos que utiliza a través de sus respectivos diagramas relacionales. Por último, detallaremos la interfaz de usuario de nuestra aplicación cliente.

4.1. Tecnología utilizada

En este apartado hablaremos del conjunto de librerías, middleware, y, en definitiva, software de terceros que hemos empleado para el desarrollo de nuestro sistema. Para facilitarlo distinguiremos entre la tecnología empleada en el cliente y la utilizada en el servidor.

4.1.1. Tecnología utilizada en el cliente

Retrofit

Se trata de un cliente HTTP con seguridad de tipado desarrollado para Android y Java. Ha sido utilizado para realizar peticiones de tipo GET, tanto a las API's previamente mencionadas como al servidor, y de tipo POST y DELETE a este último.

Para el tratamiento de las respuestas se ha usado la librería GSON Converter, que nos proporciona un conversor de objetos JSON a objetos Java (y viceversa).

Además, se ha añadido un 'HttpLoggingInterceptor' a cada petición realizada para una mejor visualización de los datos de petición y respuesta.

```
// Iniciamos retrofit
        loggingInterceptor = new HttpLoggingInterceptor().setLevel(HttpLoggingInter
ceptor.Level.BODY);
        httpClientBuilder = new OkHttpClient.Builder().addInterceptor(loggingInterc
eptor);
        retrofit = new Retrofit.Builder()
                .baseUrl(SERVER_URL)
                .addConverterFactory(GsonConverterFactory.create())
                .client(httpClientBuilder.build())
                .build();
        client = retrofit.create(ServerClient.class);
```

Tabla 4.1 - Ejemplo de código iniciando retrofit

Picasso

Librería que nos permite descargar imágenes de una dirección URL y asignarlas al 'ImageView' correspondiente en una sola línea de código. Se ha utilizado para descargar las carátulas y fondos de las películas.

```
 // Obtenemos la imagen de fondo de la película
        Picasso.get().load(IMG_URL + movie.getBackdropPath()).into(imgBackdrop);
```

Tabla 4.2 - Fragmento de código que ejemplifica el uso de la librería Picasso

SDP - *a scalable unit size*

Librería creada por el usuario de github Elhanan Mishraky que nos proporciona una nueva unidad de medida para desarrollar las vistas de nuestra aplicación. Lo que hace destacar a esta unidad de medida 'sdp' frente a las usadas por defecto ('dp', por ejemplo) es que su tamaño se determina en función de la resolución de la pantalla del terminal que ejecute la aplicación. De esta forma, conseguimos desarrollar una aplicación cuya interfaz escala según el dispositivo en el que se ejecute.

SQLite

Motor de base de datos relacionales SQL integrado por defecto en el SDK de Android. Se ha utilizado para la gestión local de los datos referentes al usuario, tales como sus listas y las películas asociadas a ellas; y las suscripciones del usuario a las listas públicas. Este tema se tratará con mayor profundidad en el apartado de 'Implementación'.

4.1.2. Tecnología utilizada en el servidor

Ecto v3.6.2

Librería proporcionada por Elixir, se encargará de la validación de datos y de la persistencia. Para ello, utilizará el módulo '*Repo*' y proporcionará soporte para trabajar con bases de datos SQL, siendo PostgreSQL la definida por defecto, y nuestra elección. Ecto trabaja directamente con el modelo de la aplicación Phoenix, que se definen directamente como esquemas *('schemas'),* y hará uso de archivos de migraciones para crear la base de datos. Para el tema de la validación empleará las funciones denominadas *'changesets'*.

Pow v1.0.24

Librería escrita en Elixir desarrollada por Dan Schultzer que se encarga de la autenticación y gestión de usuarios. Ha sido utilizada para las funcionalidades de registro y login.

4.2. Diagramas de diseño

Para una correcta comprensión del software desarrollado, se han elaborado los siguientes diagramas de diseño.

4.2.1. Diagrama de paquetes del cliente

Para explicar la parte del cliente se ha definido el siguiente diagrama de paquetes para poder ver las diferentes partes que lo conforman y cómo se relacionan entre ellas. Como para toda aplicación Android, se sigue el patrón de diseño Modelo-Vista-Controlador (MVC).

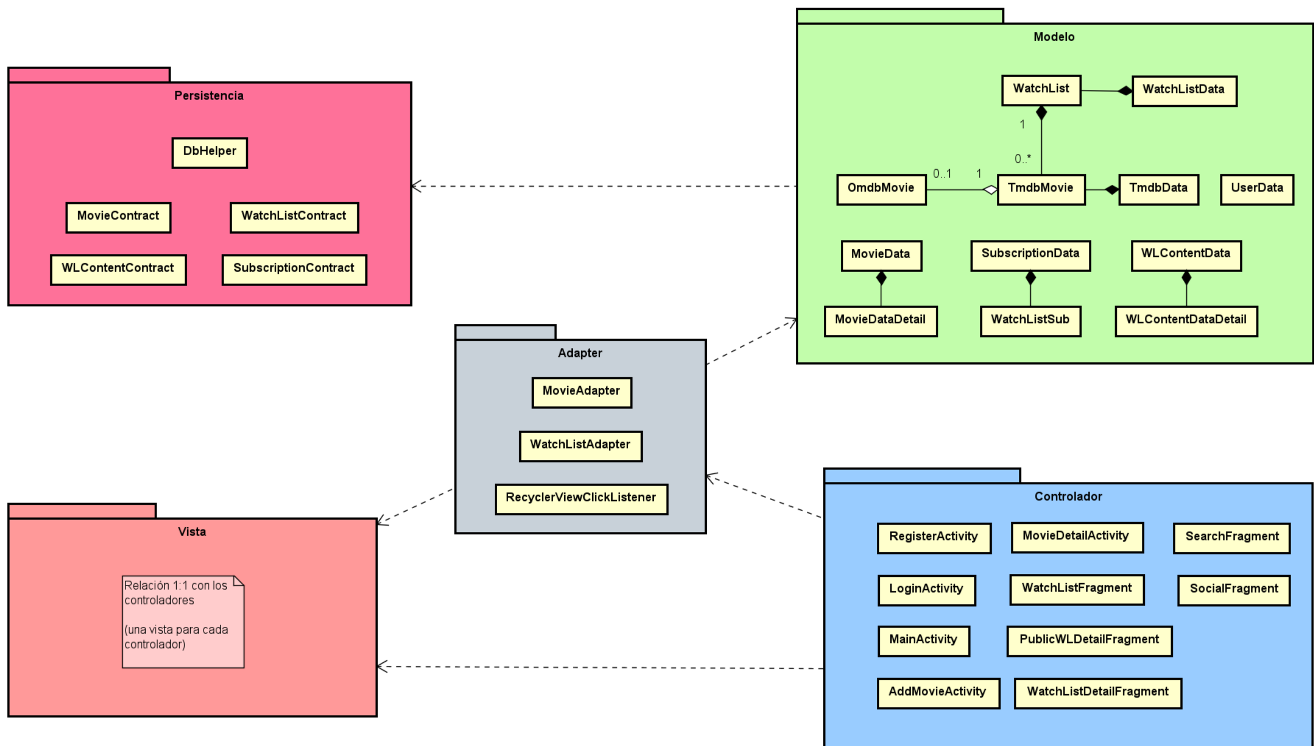

Figura 4.1 - Diagrama de paquetes del cliente

En el modelo del cliente será donde se defina la lógica de negocio de nuestra aplicación. Los datos principales con los que tratamos son las películas y las watchlists, dividiéndose las primeras en dos clases, '*TmdbMovie'* y '*OmdbMovie*', al definirse en base a las dos APIs de consulta mencionadas y de las que heredan su nombre. El resto de las clases acabadas en 'Data' serán las que hagan de intermediarias entre los datos obtenidos de nuestro servidor, o de las APIs de consulta, y la lógica de negocio del cliente. El modelo se comunicará con la capa de persistencia, donde se encuentra la lógica relativa a la base de datos local.

Los controladores en Android hacen referencia a las distintas actividades y fragmentos (comportamientos con interfaz asociada que se ejecutan dentro de una actividad) que generan las vistas de la aplicación. Serán los encargados de crear y manipular las vistas en base a la lógica de negocio del modelo y a los datos de la sesión del usuario. También son los que se comunicarán con el servidor de nuestra aplicación y con los servidores relativos a las APIs de consulta a través de dos interfaces definidas en la capa de transporte.

Las vistas guardan relación directa 1:1 con sus controladores, es decir, cada controlador generará una vista y cada vista será gestionada únicamente por su controlador. La comunicación de la vista con el modelo no es directa, sino que se abstraerá mediante una capa adaptadora, la cual será la responsable de proporcionar los datos del modelo a las vistas y de modificar el modelo por la interacción del usuario.

4.2.2. Diagrama de clases del servidor

Para el servidor se ha elaborado el siguiente diagrama de clases:

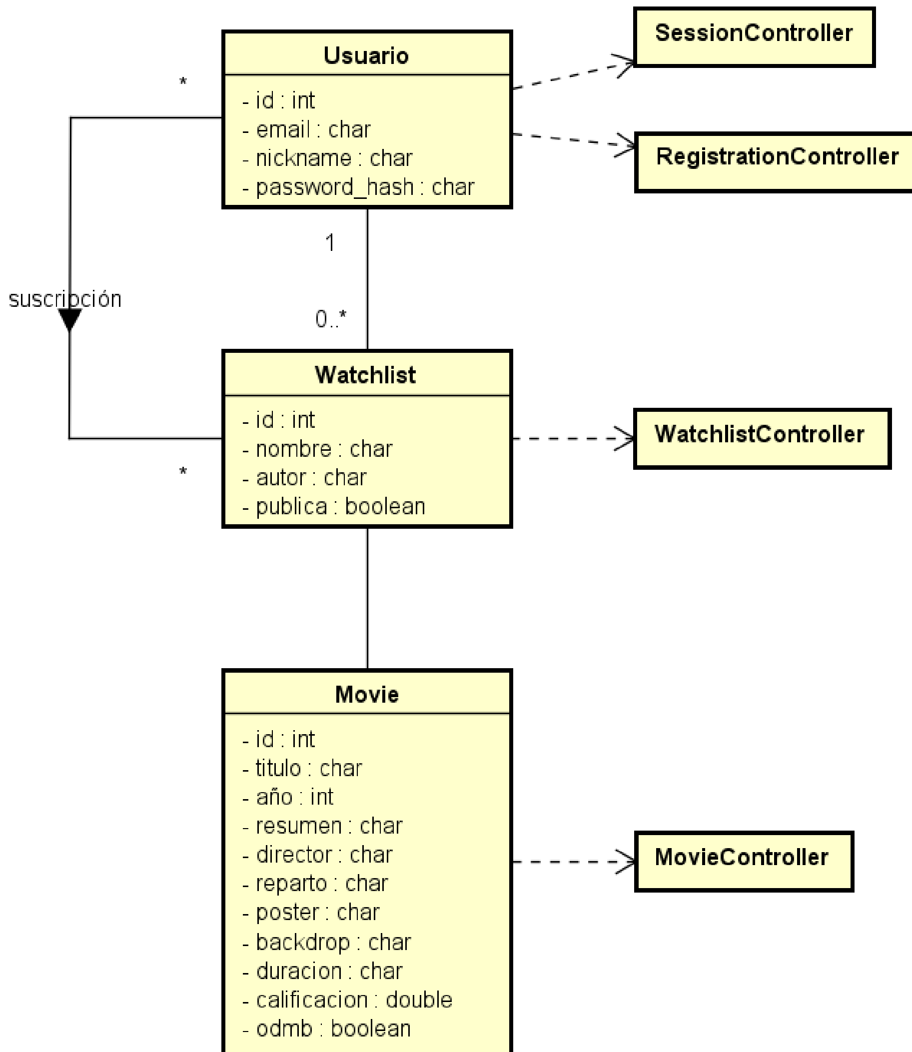

Figura 4.2 - Diagrama de clases del servidor

Como ya hemos mencionado, el framework Phoenix también se basa en el patrón de diseño MVC, sin embargo, para nuestro sistema hemos prescindido de las vistas ya que están más enfocadas a la renderización de páginas web.

En la lógica de negocio del servidor nos encontramos con tres componentes: usuarios, watchlists y películas, cada uno asociado a al menos un controlador. El componente usuario tendrá asociados dos controladores, entre los que se dividirá la funcionalidad de registro y gestión de la sesión.

4.2.3. Diagrama de despliegue del sistema

Una vez definidos los componentes de cada parte del sistema, es interesante entender cómo se relacionan entre sí. Para ello se ha elaborado el siguiente diagrama de despliegue:

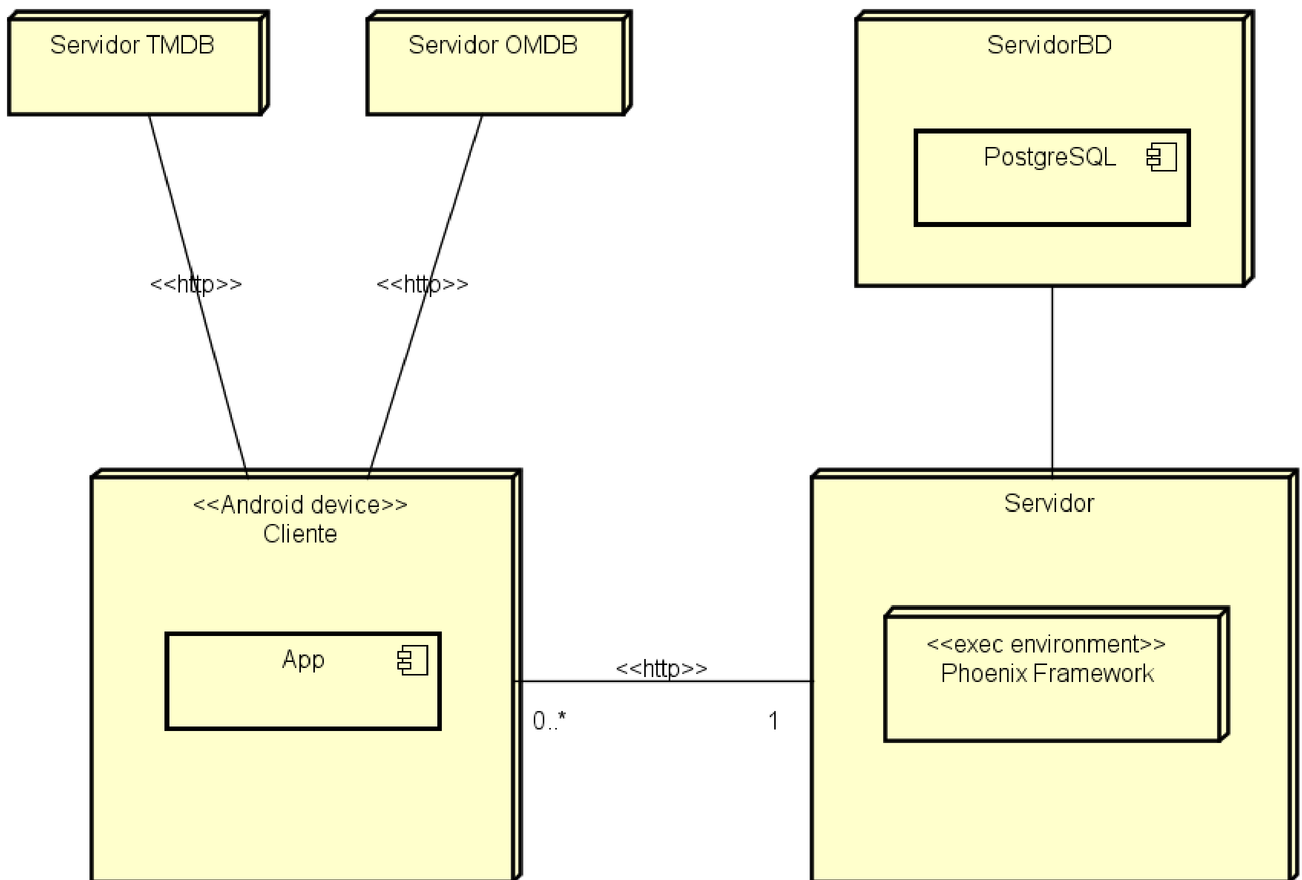

Figura 4.3 - Diagrama de despliegue del sistema

4.3. Diseño de las bases de datos

Como ya se ha mencionado previamente, para este sistema se han establecido dos bases de datos análogas: una en el servidor (PostgreSQL) y otra en el cliente (SQLite). La base de datos del servidor almacenará la información referente a las watchlists de todos los usuarios del sistema, mientras que la del cliente solo almacenará la información referente al usuario que utiliza la aplicación.

4.3.1. Base de datos del servidor

El siguiente diagrama entidad-relación define la estructura de la base de datos del servidor.

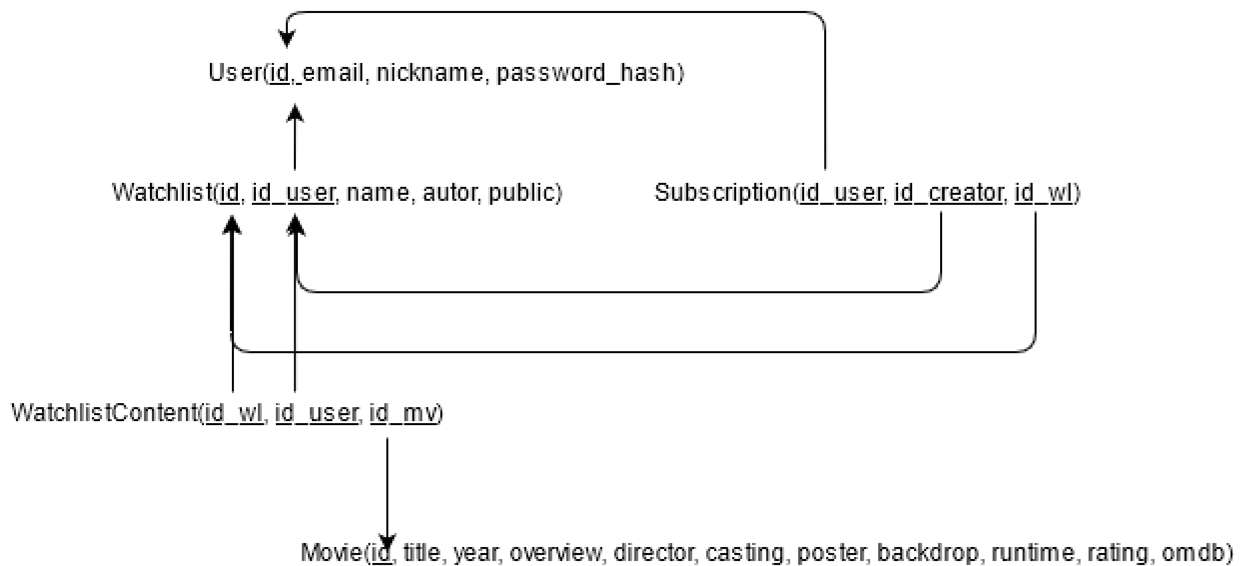

Figura 4.4 - Diagrama entidad-relación de la base de datos del servidor

Como podemos ver, está formado por tres entidades fuertes (*'User', 'Watchlist'* y *'Movie'*) y dos débiles (*'WatchlistContent'* y *'Subscription'*). La traducción de este diagrama sería: "la base de datos del servidor está compuesta por usuarios que pueden tener watchlists con películas asociadas a ellas. Estos usuarios también podrán suscribirse a watchlists creadas por otros".

La entidad débil 'WatchlistContent' relaciona watchlists y películas, pues cada película será almacenada una sola vez en base de datos, independientemente de en cuántas watchlists se encuentre.

La entidad 'Subscription' también es débil ya que solo relaciona a usuarios con watchlists, y necesitará sus tres claves foráneas para establecer un identificador único.

La entidad 'User' cuenta con un atributo 'id' como clave primaria, y será definido por el sistema de forma incremental. A parte de este id, esta entidad contará con los atributos 'email' y 'nickname', que si bien no son claves primarias sí son únicos y claves candidatas en la base de datos.

4.3.2. Base de datos del cliente

Como ya hemos mencionado, las bases de datos del cliente y del servidor son análogas, con la salvedad de que en el servidor se almacena la información de todos los usuarios y no solo la de uno. Por ello, el diagrama entidad-relación definido para la base de datos del cliente será idéntico al del servidor excepto por la entidad 'User', la cual no es necesaria para este contexto.

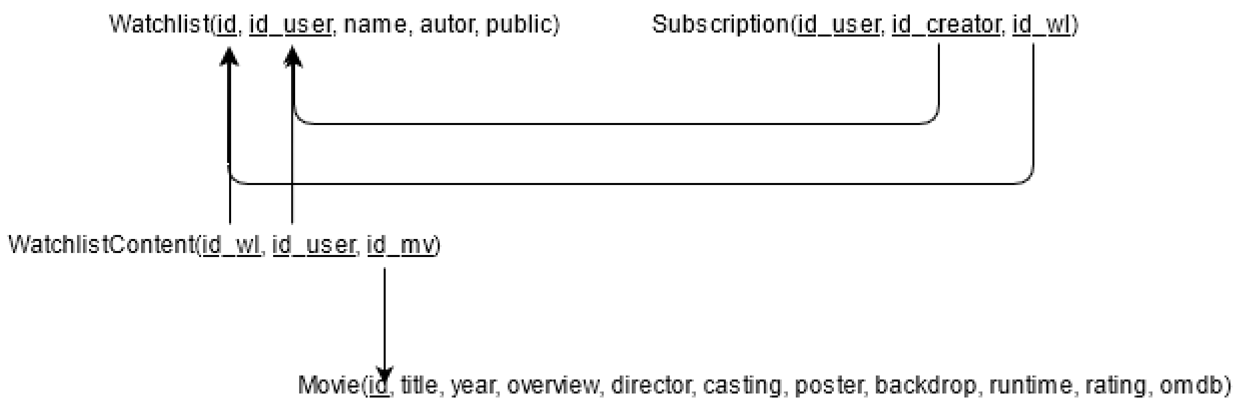

Figura 4.5 - Diagrama entidad-relación de la base de datos del cliente

Para conseguir una correcta actualización de las suscripciones a las watchlists de otros usuarios, estas no serán almacenadas en la base de datos, y se realizará una serie de peticiones al servidor cada vez que se acceda a la pantalla 'Social WatchLists' con los datos de la tabla 'Subscription' del usuario.

4.4. Diseño de la interfaz de usuario

Para el diseño de esta aplicación se ha buscado crear una interfaz limpia y fácil de utilizar, siguiendo las líneas de diseño de *Material Design*.

La primera pantalla que se nos muestra es la del login de usuario, permitiendo acceder a la de registro desde esta. Si el intento de autenticación en el sistema es satisfactorio el usuario accederá a la aplicación y no tendrá que volver a iniciar sesión mientras no borre los datos de la aplicación o la desinstale. En caso de una autenticación fallida, el usuario permanecerá en la misma pantalla y se le notificará la naturaleza del error.

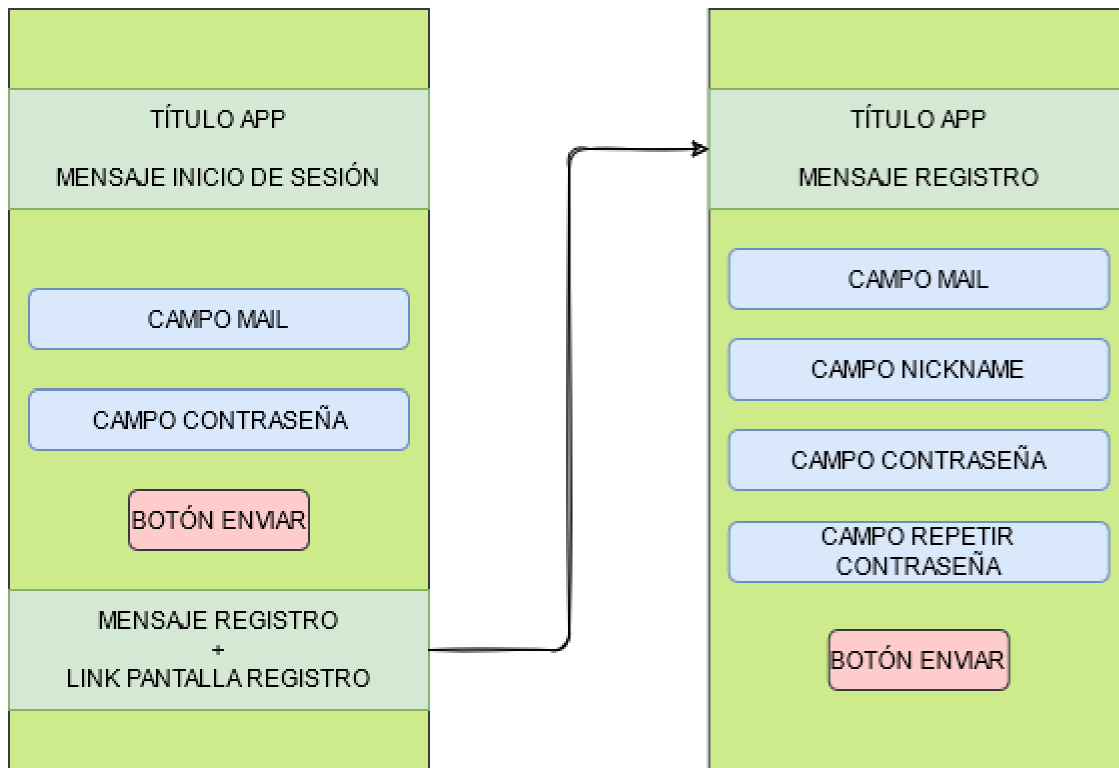

Figura 4.6 - Bocetos de las pantallas de login y registro

Una vez validados los datos del usuario satisfactoriamente podremos hacer uso de la funcionalidad de la aplicación, la cual se distribuirá en tres pantallas:

- WatchLists personales
- Buscador de películas para añadirlas a las distintas listas
- WatchLists sociales

Para facilitar la navegación entre las vistas se ha utilizado una barra de navegación inferior (*'BottomNavigationView'*). Esta permanecerá visible en pantalla en todo momento salvo en la vista del detalle de las películas y en la de añadirlas a una lista.

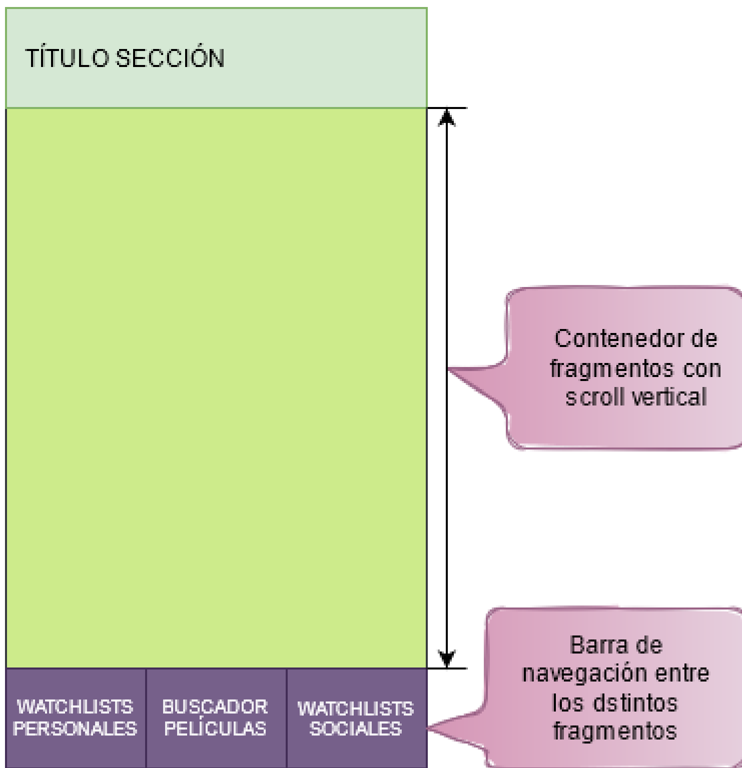

Figura 4.7 - Boceto del esquema general de la actividad principal

A continuación, procederemos a explicar cada pantalla de manera más detallada.

4.4.1. Watchlists personales

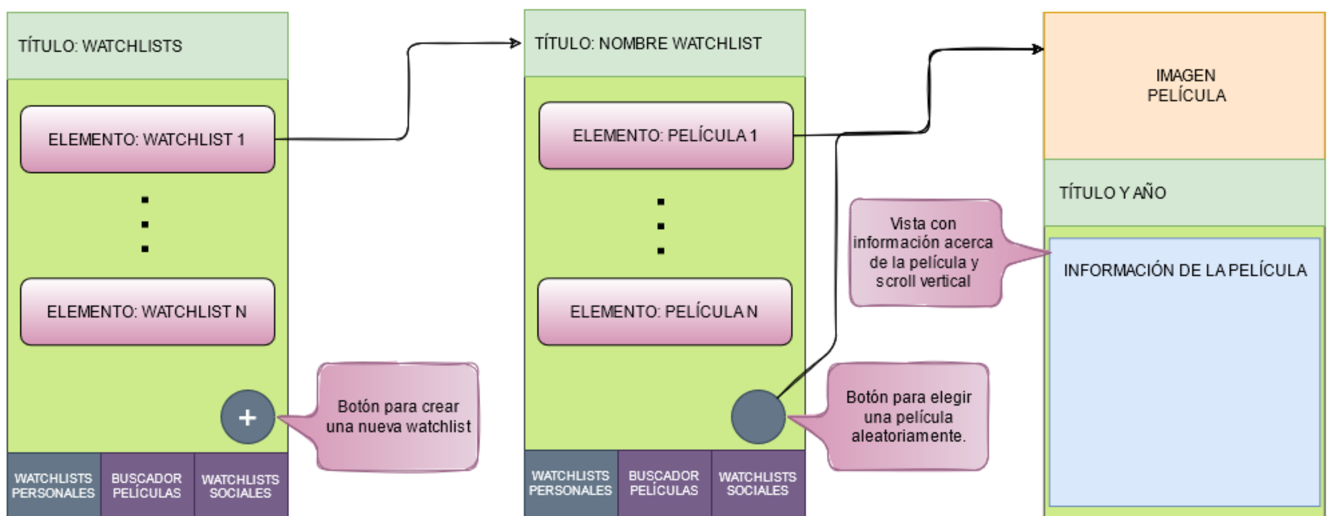

Figura 4.8 - Boceto de la secuencia de navegación entre las distintas pantallas del apartado 'WatchLists personales'

En esta pantalla se encontrarán las listas creadas por el usuario, tanto las públicas como las privadas. Estas se podrán crear desde esta misma pantalla o en la vista de 'añadir película a watchlist'. Para borrar una lista bastará con mantener pulsado sobre ella y darle a "sí" en el cuadro de diálogo emergente.

Cada lista es representada por un '*CardView*', que mostrará:

- El nombre de la lista
- El número total de películas en ella
- Un icono que indicará que la lista es pública, y solo se mostrará en ese caso
- Una imagen escogida aleatoriamente entre las carátulas de las películas de dentro de la lista, la cual podrá cambiar entre cualquiera de ellas cada vez que se acceda a esta pantalla.

Al pulsar sobre la tarjeta de la watchlist accederemos a la vista de su detalle, donde se mostrarán las películas incluidas en esta lista. Además, se podrá elegir una de sus películas aleatoriamente al pulsar un botón. También podremos acceder al detalle de las películas al pulsar sobre ellas.

4.4.2. Buscador de películas

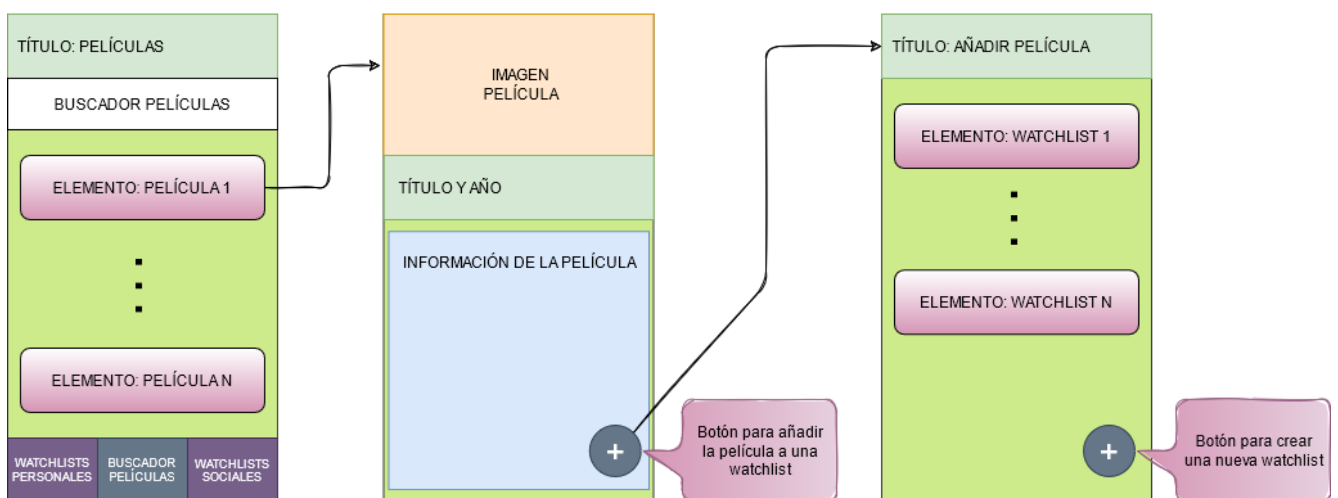

Figura 4.9 - Boceto de la secuencia de navegación entre las distintas pantallas del apartado 'Buscador de películas'

En esta pantalla se recoge la funcionalidad referente a la consulta de las dos APIs previamente mencionadas para la obtención de películas. Constará de un buscador y un apartado con las últimas películas añadidas a las listas del usuario. Dicho apartado se sustituirá por el resultado de la búsqueda una vez el usuario realice la consulta. El orden en el que aparecen las películas de la búsqueda se basará en el valor del campo de popularidad obtenido en la consulta a la API de TMDB.

Al pulsar en cualquier punto de la tarjeta de la película accederemos a la vista de su detalle, donde podremos conocer distintos datos de la película y añadirla a cualquiera de nuestras listas, o a una que decidamos crear.

4.4.3. Watchlists sociales

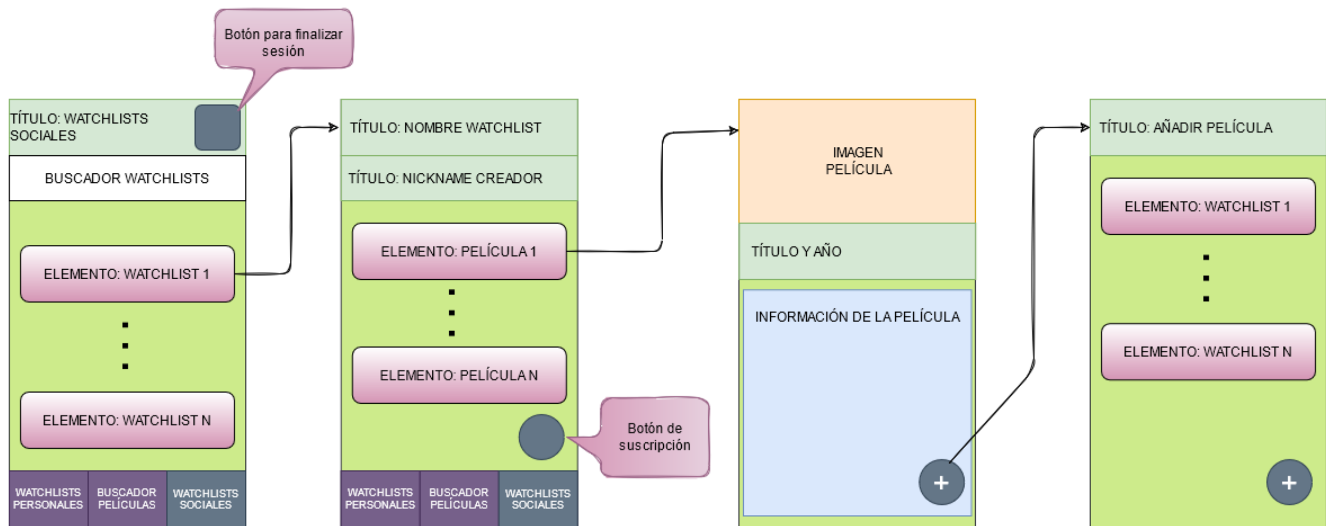

Figura 4.10 - Boceto de la secuencia de navegación entre las distintas pantallas del apartado 'WatchLists sociales'

En esta pantalla encontraremos toda la funcionalidad referente a las suscripciones a las listas públicas (búsqueda, subscripción y baja) así como un botón para finalizar la sesión. Para ello, la vista contendrá un buscador de listas públicas (que enviará la petición al servidor de nuestra aplicación) y un apartado con las subscripciones activas del usuario. En cuanto el usuario realice una consulta, al igual que en la vista del buscador de películas, este apartado será sustituido por los resultados de la búsqueda.

Las tarjetas de las listas públicas son muy similares a las de las personales, añadiendo el campo del nick del creador y obviando la distinción entre lista pública y privada. Una vez pulsemos sobre cualquiera de ellas accederemos a la vista de su detalle. Para ello, comprobaremos si el usuario está ya subscrito o es el propio autor. Si no es así se mostrará un botón para seguir la lista.

Capítulo 5

Implementación

En este capítulo se abordará la fase de implementación del sistema. Con el objetivo de lograr una mayor comprensión de su funcionamiento, tanto de la parte del cliente como de la del servidor, se ha decidido desglosar las distintas clases según la funcionalidad que aporten:

- Interfaces de conexión
- Adaptadores de vista
- Actividad principal
- Login y registro
- Administración de las watchlists personales
- Búsqueda de películas
- Gestión de las suscripciones del usuario

5.1. Interfaces de conexión

En esta sección se expondrán las interfaces implementadas para realizar las conexiones entre el cliente y las APIs y el servidor. En concreto serán dos interfaces: 'APIClient.java' para la conexión con las APIs y 'ServerClient.java' para la conexión con el servidor.

También se detallará el archivo *'router.ex'* de nuestro servidor y cuyo funcionamiento fue explicado en la sección de 'Antecedentes y tecnología usada'.

APIClient.java

```
public interface APIClient {

    @GET("search/movie")
    Call<TmdbData> getTmdbMovie(
            @Query("api_key") String api,
            @Query("language") String lang,
            @Query("query") String query,
            @Query("page") int page,
            @Query("include_adult") boolean adult
    );

    @GET(".")
    Call<OmdbMovie> getOmdbMovie(
            @Query("t") String title,
            @Query("y") String year,
            @Query("apikey") String api,
            @Query("type") String type
    );
}
```

Tabla 5.1 - Interfaz para la conexión del cliente con las APIs

Como podemos ver, tan solo se utilizarán dos métodos para obtener las películas, uno para cada API consultada.

ServerClient.java

```
@POST("api/v1/session")
Call<ResponseBody> login(
            @Query("user[email]") String email,
            @Query("user[password]") String password
   );

@GET("api/v1/getUserData")
Call<UserData> getUserData(
             @Query("email") String email
    );

@DELETE("api/v1/watchlist/deleteMovie")
Call<ResponseBody> deleteMovie(
             @Query("id_user") int idUser,
```

```
            @Query("id_wl") int idWl,
            @Query("id_mv") int idMv
    );
```

Tabla 5.2 - Fragmento de código de la interfaz de conexión del cliente con el servidor

En esta interfaz se definirán todos los métodos necesarios para la conexión del cliente con el servidor. En este caso son 18 métodos los que se implementan, pero por la similitud entre ellos nos bastará con los 4 expuestos para ejemplificarlos. En ellos se pueden apreciar solicitudes de tipo GET, POST y DELETE que nuestro cliente envía al servidor.

router.ex

```
pipeline :api do
    plug :accepts, ["json"]
    plug :fetch_session
    plug :fetch_flash
    plug FilmotecaWeb.APIAuthPlug, otp_app: :filmoteca
  end

  scope "/api/v1", FilmotecaWeb.API.V1, as: :api_v1 do
    pipe_through :api

    resources "/registration", RegistrationController, singleton: true, only: [:create]
    resources "/session", SessionController, singleton: true, only: [:create, :delete]
    post "/session/renew", SessionController, :renew
    get "/getUserData", SessionController, :getUserData
    resources "/movies", MovieController
    resources "/watchlists", WatchlistController
    post "/watchlist/addMovie", WatchlistController, :addMovie
    get "/watchlist/getWatchlists", WatchlistController, :getWatchlists
    get "/watchlist/getUserMovies", WatchlistController, :getUserMovies
    get "/watchlist/getWatchlistcontents", WatchlistController, :getWatchlistcontents
    delete "/watchlist/deleteMovie", WatchlistController, :deleteWatchlistcontent
    delete "/watchlist/deleteWatchlist", WatchlistController, :deleteWatchlist
    get "/watchlist/public", WatchlistController, :searchPublicWatchlist
    get "/watchlist/getPublicMovies", WatchlistController, :getPublicMovies
    post "/watchlist/subscribe", WatchlistController, :subscribeToWatchlist
    get "/watchlist/getSubscriptions", WatchlistController, :getSubscriptions
    get "/watchlist/getSubscriptionWls", WatchlistController, :getSubscriptionWatchlists
    delete "/watchlist/unsubscribe", WatchlistController, :unsubscribe
  end
```

Tabla 5.3 - Fragmento de código del archivo 'router.ex' del servidor Phoenix

Como ya se mencionó, lo más relevante de este archivo reside en la configuración de tuberías *('pipelines')* y dominios *('scopes')*. Para nuestro sistema se utilizará la tubería ':api', y en su dominio se establecerán todas las peticiones que el cliente puede realizar al servidor. Las rutas definidas coincidirán con las de la interfaz del cliente.

5.2. Adaptadores de vista

Las clases que se tratarán en este punto constituyen la capa de abstracción entre el modelo y la vista. Serán utilizadas por el controlador para mostrar los datos del modelo por pantalla, el cual podrá ser modificado por las acciones del usuario.

Estas clases son necesarias porque para mostrar los datos del modelo utilizamos la vista *'RecyclerView'*, la cual necesitará que se implementen adaptadores para los datos a tratar y un listener para controlar la acción del usuario.

MovieAdapter.java

Esta primera clase constituye el adaptador para los objetos *TmdbMovie,* es decir, las películas. Extenderá la clase *'RecyclerView.Adapter<MovieAdapter.MovieAdapterHolder>'.*

Clase MovieAdapterHolder

```
public class MovieAdapterHolder extends RecyclerView.ViewHolder{
        private TextView tvTitle;
        private TextView tvYear;
        private ImageView imgPoster;
        private ImageView imgRating;
        private TextView tvRating;

        public MovieAdapterHolder (@NonNull View itemView){
            super(itemView);
            tvTitle = itemView.findViewById(R.id.tvTitle);
            tvYear = itemView.findViewById(R.id.tvYear);
            imgPoster = itemView.findViewById(R.id.imgPoster);
            imgRating = itemView.findViewById(R.id.imgRating);
            tvRating = itemView.findViewById(R.id.tvRating);
        }
    }
```

Tabla 5.4 - Clase 'MovieAdapterHolder' contenida en la clase 'MovieAdapter'

Esta clase está contenida en la anterior y se corresponde con la tarjeta que representa cada elemento de la vista, enlazando sus componentes.

Método onBindViewHolder()

```
@Override
public void onBindViewHolder(@NonNull MovieAdapterHolder holder, int position) {

        TmdbMovie tmdbMovie = tmdbMovies.get(position);
        holder.tvTitle.setText(tmdbMovie.getTitle());
        holder.tvYear.setText(tmdbMovie.getReleaseDate());
        if(tmdbMovie.getOmdbMovie() != null){
            setUpOmdb(holder, tmdbMovie);
        }else{
            setUpTmdb(holder, tmdbMovie);
        }
    }
```

Tabla 5.5 - Método 'onBindViewHolder()' de la clase MovieAdapter.java

WatchListAdapter.java

Clase muy similar a la anterior descrita, con la salvedad de que los datos aquí tratados se corresponden con los de las watchlists. Se omitirán los detalles de su implementación al no aportarnos nada nuevo, con la excepción del método *'setRandomPoster()'*.

Método setRandomPoster()

```
private void setRandomPoster(WatchListAdapterHolder holder, int position){
        Random r = new Random();
        int random = r.nextInt(wlists.get(position).getMovies().size());
        TmdbMovie movie = wlists.get(position).getMovies().get(random);
        if(movie.getOmdbMovie()!=null && !"N/A".equals(movie.getOmdbMovie().getPoster()) &&
!movie.getOmdbMovie().getPoster().isEmpty()){
            Picasso.get().load(movie.getOmdbMovie().getPoster()).into(holder.randomPoster);
        }else{
            Picasso.get().load(IMG_URL + movie.getPosterPath()).into(holder.randomPoster);
        }
    }
```

Tabla 5.6 - Método 'setRandomPoster()' en la clase WatchListAdapter.java

Método que establece una imagen a la tarjeta de la watchlist elegida de forma aleatoria de entre las carátulas de las películas añadidas a esta. Este método será llamado desde *'onBindViewHolder()',* y, por tanto, se ejecutará cada vez que se genere la vista, cambiando así las imágenes mostradas.

RecyclerViewClickListener.java

Clase que implementa el listener para las vistas RecyclerView. Nos permitirá establecer los eventos 'click único' y 'mantener pulsado'.

```
public RecyclerViewClickListener(Context context, final RecyclerView recyclerView, OnItemClickLis
tener listener) {
        mListener = listener;
        mGestureDetector = new GestureDetector(context, new GestureDetector.SimpleOnGestureListen
er() {
            @Override
            public boolean onSingleTapUp(MotionEvent e) {
                return true;
            }

            @Override
            public void onLongPress(MotionEvent e) {
                View child = recyclerView.findChildViewUnder(e.getX(), e.getY());
                if (child != null && mListener != null) {
                    mListener.onLongItemClick(child, recyclerView.getChildAdapterPosition(child))
;
                }
            }
        });
    }
```

Tabla 5.7 - Constructor de la clase 'RecyclerViewClickListener.java'

5.3. Actividad principal

En Android, la clase '*Activity*' representa el componente base de cualquier aplicación. Generalmente, una actividad implementa una pantalla en una aplicación, pero también puede usarse para albergar distintos *'Fragments'* en ella, los cuales son porciones de interfaz de usuario que representan un determinado comportamiento.

La primera actividad que se inicia al ejecutar una aplicación Android se denomina actividad principal, y es por ello que para hablar de la implementación de nuestro sistema comenzaremos por la clase 'MainActivity.java'.

MainActivity.java

La actividad principal de nuestra aplicación actuará como un contenedor de distintos fragments que implementarán la funcionalidad de la misma. Por ello, estará presente durante casi todo el ciclo de vida de la aplicación.

Método onCreate()

El primer método en ejecutarse en cualquier actividad es *'onCreate()'*, el cual es heredado de la clase '*AppCompatActivity*', cuya extensión es requisito base para que la clase creada sea una actividad.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        setUpSession();

        fragment = new WatchListFragment();
        boolean fromRegister = getIntent().getBooleanExtra("register", false);
        if(fromRegister){
            Bundle b = new Bundle();
            b.putBoolean("register", true);
            fragment.setArguments(b);
        }

        BottomNavigationView navigation = findViewById(R.id.navigationView);
        navigation.setOnNavigationItemSelectedListener(toolbarListener);
        loadFragment(fragment);
    }
```

Tabla 5.8 - Método 'onCreate()' de la actividad principal

Lo primero que hace este método es fijar su vista correspondiente, *'activity_main.xml'* (línea 20). Tras esto, comprobará el estado de la sesión del usuario en el método *'setUpSession()'*, y en base a esto se seguirá un flujo u otro.

Lo siguiente que hace es comprobar si esta actividad se ha lanzado desde la actividad de registro, lo cual se indica mediante la adición de "extras" al *intent* que origina nuestra actividad. Esta comprobación se realiza para saber si mostrar la guía de bienvenida, la cual solo será mostrada una vez tras un registro satisfactorio.

Por último, se fijará la barra de navegación inferior con su *listener*, la cual se utilizará para desplazarse por las distintas pantallas de la aplicación, y se cargará el fragment por defecto: WatchListFragment.

Método setUpSession()

```
private void setUpSession(){
        SessionManager sm = new SessionManager(getApplicationContext());
        if(!sm.checkLogin()){
            Intent i = new Intent(this, LoginActivity.class);
            startActivity(i);
            finish();
        }
    }
```

Tabla 5.9 - Método 'setUpSession()' de la actividad principal

Su función consiste en comprobar que el usuario haya iniciado sesión en la aplicación para poder usarla, y, si no es así, iniciar la actividad de inicio de sesión (en adelante *login*).

Listener de la BottomNavigationView

```
private BottomNavigationView.OnNavigationItemSelectedListener toolbarListener
            = item -> {
                Fragment fragment;
                switch (item.getItemId()) {
                    case R.id.user_lists_nav:
                        fragment = new WatchListFragment();
                        loadFragment(fragment);
                        return true;
                    case R.id.search_nav:
                        fragment = new SearchFragment();
                        loadFragment(fragment);
                        return true;
                    case R.id.user_group_nav:
                        fragment = new SocialFragment();
                        loadFragment(fragment);
```

```
                    return true;
            }
            return false;
        };
```

Tabla 5.10 - Listener de la 'BottomNavigationView'

La barra de navegación inferior contendrá tres elementos, y el listener implementado para ella registrará la interacción del usuario con estos elementos para cargar un fragment u otro.

Método loadFragment()

```
private void loadFragment(Fragment fragment) {
        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.frame_container, fragment);
        transaction.addToBackStack(null);
        transaction.commit();
    }
```

Tabla 5.11 - Método 'loadFragment()' de la actividad principal

Método que recibirá como parámetro un objeto de tipo *'Fragment'* y lo iniciará dentro de la actividad principal, mostrándolo también en pantalla.

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.filmoteca.MainActivity"
    android:background="@color/teal_700">

    <FrameLayout
        android:id="@+id/frame_container"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_marginBottom="55dp"
        app:layout_behavior="@string/appbar_scrolling_view_behavior" />
```

```
    <com.google.android.material.bottomnavigation.BottomNavigationView
        android:id="@+id/navigationView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:background="?android:attr/windowBackground"
        android:foreground="?attr/selectableItemBackground"
        app:itemBackground="@color/aquamarine"
        app:itemIconTint="@color/bottom_nav_color"
        app:itemTextColor="@android:color/white"
        app:labelVisibilityMode="unlabeled"
        app:menu="@menu/navigation" />
</RelativeLayout>
```

Tabla 5.12 - Vista de la actividad principal

Las vistas en Android se implementan mediante un archivo XML. En este caso podemos observar la definición de la vista de la actividad principal, constituida por un *layout* que alberga el contenedor de fragmentos y la barra de navegación. La implementación del resto de vistas de las distintas actividades y fragmentos se omitirá por su tamaño y la poca información que nos aporta, pero todas seguirán el mismo esquema y usarán *RelativeLayout.*

5.4. Login y registro

Una de las principales características de la aplicación es la posibilidad de crearte una cuenta de usuario para poder disponer de las watchlists que crees en cualquier teléfono Android. Para ello se han desarrollado las actividades de login y registro.

Otro requisito del sistema a cumplir es la necesidad de mantener la sesión iniciada en el dispositivo hasta que el usuario indique lo contrario. Con este objetivo se ha implementado un gestor de la sesión, del cual empezaremos hablando antes de entrar en detalle con las actividades previamente mencionadas.

5.4.1. Gestión de la sesión

SessionManager.java

Se trata de una clase implementada en el cliente que gestionará el estado de la sesión del usuario en el terminal. Para ello creará un archivo de preferencias mediante el uso de un editor, lo cual garantizará su persistencia aun cuando se finalice la aplicación.

```
public class SessionManager {
    SharedPreferences pref;
    SharedPreferences.Editor editor;
    Context _context;

    private static final String PREF_NAME = "FilmotecaPref";
    private static final String IS_LOGIN = "IsLoggedIn";
    public static final String KEY_EMAIL = "email";
    public static final String KEY_ID = "id";
    public static final String KEY_NICKNAME = "nickname";

    public SessionManager (Context context){
        this._context = context;
        pref = _context.getSharedPreferences(PREF_NAME, Private_mode);
        editor = pref.edit();
    }

    public void createLoginSession(String email, int id, String nick){
        editor.putBoolean(IS_LOGIN, true);
        editor.putString(KEY_EMAIL, email);
        editor.putString(KEY_ID, String.valueOf(id));
        editor.putString(KEY_NICKNAME, nick);

        editor.apply();
        editor.commit();
    }

    public boolean checkLogin(){
        return pref.getBoolean(IS_LOGIN, false);
    }

    public HashMap<String, String> getUserDetails(){
        HashMap<String, String> user = new HashMap<String, String>();
        user.put(KEY_EMAIL, pref.getString(KEY_EMAIL,null));
        user.put(KEY_ID, pref.getString(KEY_ID,null));
        user.put(KEY_NICKNAME, pref.getString(KEY_NICKNAME,null));
        return user;
```

```
    }

    public void logoutUser(){
        editor.clear();
        editor.apply();
        editor.commit();

        Intent i = new Intent(_context, LoginActivity.class);
        _context.startActivity(i);
    }
}
```

Tabla 5.13 - Clase 'SessionManager.java' encargada de gestionar la sesión del usuario

Como podemos ver en la figura de arriba, la clase dispondrá de un constructor en el que se fijará un contexto, un objeto de tipo *'SharedPrefereces'* y su correspondiente editor. Seguido de este constructor nos encontramos con las siguientes funciones:

- *'createLoginSession()':* será la encargada de crear el archivo de preferencias indicando los datos del usuario y que la sesión ya está iniciada.
- *'checkLogin()':* función utilizada en la actividad principal para conocer el actual estado de la sesión del usuario.
- *'getUserDetails()':* función que nos devolverá los datos del usuario en una estructura HashMap.
- *'logoutUser()':* con este método indicamos a la aplicación que cierre la sesión de usuario, para ello borrará el archivo de preferencias e iniciará la actividad de login.

5.4.2. Login

En este subapartado se tratará la funcionalidad de login tanto en el cliente como en el servidor. Empezaremos detallando la clase *'LoginActivity.java'* para después adentrarnos en el método invocado en el servidor.

LoginActivity.java

Esta actividad será iniciada por la actividad principal si no hay ninguna sesión registrada en la aplicación. Será la encargada de validar los datos del usuario, tramitar la petición de inicio de sesión y de gestionar los datos del usuario devueltos por el servidor.

Método onCreate()

```
@Override
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
        sm = new SessionManager(getApplicationContext());
        setUpView();
    }
```

Tabla 5.14 - Método 'onCreate()' de la actividad de login

A partir de ahora, para todas las actividades y fragmentos, se realizarán las configuraciones de su vista asociada en el método *'setUpView()'*. Por su extensión y la poca relevancia que aporta se omitirán los detalles de implementación del método, salvo los relativos a los botones de 'login' y de 'registro'.

Método setUpView()

```
signInBtn.setOnClickListener(v -> {
            email = emailEt.getText().toString();
            password = passwordEt.getText().toString();
            sendLogInPetition();
        });

        registerTv.setOnClickListener(v -> {
            email = emailEt.getText().toString();
            password = passwordEt.getText().toString();
            Intent i = new Intent(this, RegisterActivity.class);
            i.putExtra("EMAIL", email);
            i.putExtra("PASSWORD", password);
            startActivityForResult(i, REQUEST_EXIT);
            this.overridePendingTransition(0, 0);
        });
```

Tabla 5.15 - Fragmento del método 'setUpView()' de la actividad de login en el que se pueden ver los listeners establecidos para los botones de 'login' y de 'registro'

Al pulsar en el botón de login, la actividad validará los datos introducidos por el usuario y se iniciará una cadena de peticiones al servidor para obtener todos los datos del usuario. Si, en cambio, se pulsa en el de registro se iniciará la actividad de registro con los campos 'email' y 'password' ya con los valores introducidos en estos mismos campos en la actividad de login.

Método sendLoginPetition()

```
private void sendLogInPetition(){
        Call<ResponseBody> call = client.login(email, password);
        call.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
                Log.d("CONNECTION_SUCCESS", "Se ha establecido conexión con el servidor, login");
                if(response.code()==200) {
                    getUserData();
                }else if(response.code() == 401){
                    Log.d("ERROR 401 ", response.message());
                    errorTv.setText(R.string.login_user_error);
                }
            }

            @Override
            public void onFailure(@NotNull Call<ResponseBody> call, Throwable t) {
                Log.d("CONNECTION_ERROR", t.getMessage());
            }
        });
    }
```

Tabla 5.16 - Método 'sendLoginPetition()' de la actividad de login

En este método se establecerá conexión con el servidor para realizar el inicio de sesión. Si el login no es satisfactorio (error 401) se notificará al usuario. Si, por el contrario, resulta exitoso, se encadenarán el resto de las llamadas al servidor para traer los datos del usuario, en este orden:

- getUserData(): método utilizado para obtener el código identificador del usuario en el sistema.
- getUserWatchLists(): nos devolverá las listas de películas del usuario, con sus películas y las relaciones de pertenencia entre ellas. Si este proceso resulta exitoso se invocará el método *'manageSuccessfulLogin()'*.

Método manageSuccessfulLogin()

```
private void manageSuccessfulLogin(){
        // Creamos la sesión
        Log.d("USER_LOGIN", String.valueOf(idUser));
        sm.createLoginSession(email, idUser, nick);

        // Iniciamos MainActivity
        Intent i = new Intent(getApplicationContext(), MainActivity.class);
        startActivity(i);
        finish();
    }
```

Tabla 5.17 - Método manageSuccessfulLogin() de la actividad de login

En este método se crea la sesión en la aplicación a través del SessionManager y se inicia de nuevo la actividad principal, que ya detectará la sesión del usuario iniciada.

Método onActivityResult()

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        if (requestCode == REQUEST_EXIT) {
            if (resultCode == Activity.RESULT_CANCELED) {
                finish();
            }
        }
    }
```

Tabla 5.18 - Método 'onActivityResult()' de la actividad de login

Para finalizar con la funcionalidad de login en el cliente hablaremos del método *'onActivityResult()'*, heredado de la clase AppCompatActivity y cuya función será finalizar la actividad si se consigue un registro exitoso desde la actividad de registro.

Implementación en el servidor

Como hemos podido ver, esta actividad previamente mencionada realizará una serie de peticiones al servidor. A continuación, veremos los métodos invocados en este.

Método create(), session_controller.ex

```
@spec create(Conn.t(), map()) :: Conn.t()
def create(conn, %{"user" => user_params}) do
    conn
    |> Pow.Plug.authenticate_user(user_params)
    |> case do
      {:ok, conn} ->
        json(conn, %{data: %{access_token: conn.private.api_access_token, renewal_t
oken: conn.private.api_renewal_token}})

      {:error, conn} ->
        conn
        |> put_status(401)
        |> json(%{error: %{status: 401, message: "Invalid email or password"}})
    end
  end
```

Tabla 5.19 - Método utilizado para iniciar sesión en el servidor

Método utilizado en nuestro servidor para iniciar la sesión del usuario. Está ubicado en el controlador de *sesión* (*'session_controller.ex*') y es invocado desde el cliente, concretamente desde la actividad de login en el método *'sendLoginPetitionn()'*. El servidor devolverá los tokens de acceso en caso de una autenticación exitosa o informará de un error 401 en caso contrario.

Método getUserData(), session_cotroller.ex

```
def getUserData(conn, %{"email" => email}) do
    id_query = from u in "users",
            where: u.email == ^email,
            select: u.id
    nick_query = from u in "users",
            where: u.email == ^email,
            select: u.nickname
    id = Repo.all(id_query)
    nick = Repo.all(nick_query)
    conn
```

```
    |> json(%{user_id: id, user_nickname: nick})
  end
```

Tabla 5.20 - Método 'getuserdata()' del controlador de sesión del servidor

Este método nos proporcionará el identificador del usuario en el sistema y su *nick* elegido. Se invocará en las actividades de login y registro del cliente.

Métodos de recogida de datos de las listas de usuario, watchlist_controller.ex

```
def getWatchlists(conn, %{"id_user" => id_user}) do
    query = from wl in Watchlist,
            where: wl.user_id == ^id_user,
            select: wl
    wls = Repo.all(query)|> Repo.preload(:user)
    conn
    |> json(%{watchlists: wls})
  end
```

```
def getUserMovies(conn, %{"id_user" => id_user}) do
    query = from wlc in Watchlistcontent,
            join: m in Movie,
            on: wlc.id_mv == m.id,
            where: wlc.id_user == ^id_user,
            select: m
    mvs = Repo.all(query)
    conn
    |> json(%{movies: mvs})
  end
```

```
def getWatchlistcontents(conn, %{"id_user" => id_user}) do
    query = from wlc in Watchlistcontent,
            where: wlc.id_user == ^id_user,
            select: wlc
    wlcs = Repo.all(query)
    conn
    |> json(%{contents: wlcs})
  end
```

```
def getSubscriptions(conn, %{"id_user" => id_user}) do
    query = from sub in Subscription,
            where: sub.id_user == ^id_user,
            select: sub
    subs = Repo.all(query)
    conn
    |> json(%{subscriptions: subs})
  end
```

Tabla 5.21 - Métodos del servidor encargados de devolver la información del usuario

Los siguientes métodos se encuentran en el controlador *'watchlist_controller.ex',* y son llamados desde la actividad de login del cliente, concretamente desde el método *'getUserWatchLists()'.* Estos devolverán, en orden, las listas de los usuarios, las películas que contienen, las relaciones de pertenencia entre listas y películas, y, por último, las suscripciones del usuario; todo ello en formato JSON.

5.4.3. Registro

A la funcionalidad de registro se accederá a través de la pantalla de inicio de sesión. En esta actividad encontraremos, además de los campos de email y contraseña, los campos de nombre de usuario y de repetir contraseña. También habrá un botón para validar y enviar el formulario al servidor.

RegisterActivity.java

Esta actividad del cliente será muy similar a la actividad de inicio de sesión descrita en el apartado anterior. Contará con el método 'onCreate()', que llamará a 'setUpView()' para configurar la vista. De este método destacaremos la implementación del listener del botón de registro.

```
registerBtn.setOnClickListener(v -> {
            email = emailEt.getText().toString();
            nick = nickEt.getText().toString();
            password = passwordEt.getText().toString();
            rePassword = passwordConfirmEt.getText().toString();

            if(email.isEmpty() || nick.isEmpty() || password.isEmpty() || rePassword.isEmpty()){
                errorTv.setText(R.string.empty_error);
            }else if(password.length() < 8){
                errorTv.setText(R.string.password_length_error);
            }else{
                if(password.equals(rePassword)){
                    sendRegisterPetition();
                }else{
                    errorTv.setText(R.string.match_error);
                }
            }
        });
```

Tabla 5.22 - Listener implementado para el botón de la actividad de registro

En este método se realizará una validación en la que se asegurará que estén todos los campos informados y que las contraseñas coincidan y sean de 8 o más caracteres de longitud. Una vez satisfechos todos estos requisitos se enviará la petición de registro al servidor a través del método *'sendRegisterPetition()'*.

Método sendRegisterPetition()

```
public void sendRegisterPetition(){
        Call<ResponseBody> call = client.register(email, nick, password, rePassword);
        call.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
                Log.d("CONNECTION_SUCCESS", "Se ha establecido conexión con el servidor (registro
)");
                if(response.code()==200) {
                    getUserData();
                }else if(response.code() == 500){
                    Log.d("ERROR 500 ", response.message());
                    errorTv.setText(R.string.register_error);
                }
            }

            @Override
            public void onFailure(@NotNull Call<ResponseBody> call, Throwable t) {
                Log.d("CONNECTION_ERROR", t.getMessage());
            }
        });
    }
```

Tabla 5.23 - Método 'sendRegisterPetition()' de la actividad de registro

Este método será el responsable de conectar con el servidor y realizar la petición de registro/alta de nuevo usuario. Si todo es satisfactorio, se realizará otra petición para obtener datos del usuario desconocidos por el cliente (mismo funcionamiento que en LoginActivity.java) para, finalmente, invocar el método *'manageSuccessfulRegister()'*.

Método manageSuccessfulRegister()

```
private void manageSuccessfulRegister(){
        // Creamos la sesión
        sm.createLoginSession(email, id, nick);

        // Cerramos la actividad del login
        Intent intentReturn = new Intent();
```

```
        setResult(LoginActivity.RESULT_CANCELED, intentReturn);

        // Iniciamos MainActivity
        Intent i = new Intent(getApplicationContext(), MainActivity.class);
        i.putExtra("register", true);
        startActivity(i);
        finish();
    }
```

Tabla 5.24 - Método 'manageSuccessfulRegister()' de la actividad de registro

Como podemos observar, es muy similar al método *'manageSuccessfulLogin()'* visto en la actividad anterior. La única diferencia que tiene es que desde este método se finalizará la actividad de login, para que el usuario no pueda acceder a ella navegando hacia atrás en las pantallas

Implementación en el servidor

Para este caso, al ser una cuenta nueva la que se crea, no serán necesarios los métodos de obtener los datos del usuario, pues no tendrá aún en el sistema. Tan solo se invocará el método responsable de dar de alta el nuevo usuario en el sistema.

Método create(), registration_controller.ex

```
@spec create(Conn.t(), map()) :: Conn.t()
def create(conn, %{"user" => user_params}) do
    conn
    |> Pow.Plug.create_user(user_params)
    |> case do
      {:ok, user, conn} ->
        json(conn, %{data: %{access_token: conn.private.api_access_token, renewal_token: conn.pri
vate.api_renewal_token}})

      {:error, changeset, conn} ->
        errors = Changeset.traverse_errors(changeset, &ErrorHelpers.translate_error/1)

        conn
        |> put_status(500)
        |> json(%{error: %{status: 500, message: "Couldn't create user", errors: errors}})
    end
  end
```

Tabla 5.25 - Método 'create()' del controlador de registro del servidor

Al igual que para el login, utilizaremos la librería Pow para la gestión del alta de nuevos usuarios.

5.5. Administración de watchlists personales

Una vez se ha establecido una sesión en el terminal Android mediante el SessionManager se vuelve a invocar la actividad principal. Como ya hemos dicho, esta actividad será principalmente un contenedor de fragmentos, y el primero en ser cargado será *'WatchListFragment.java'*. Aquí se reunirá la funcionalidad relativa a la gestión de las watchlists personales del usuario.

WatchListFragment.java

Este fragmento contendrá las listas del usuario y un botón con el símbolo '+' para poder crear más.

Método onCreateView()

```
@Nullable
@org.jetbrains.annotations.Nullable
@Override
public View onCreateView(@NonNull @NotNull LayoutInflater inflater,
                             @Nullable @org.jetbrains.annotations.Nullable ViewGroup container,
                             @Nullable @org.jetbrains.annotations.Nullable Bundle savedInstanceSt
ate) {
        View view = inflater.inflate(R.layout.fragment_watch_list, container, false);
        setUpView(view);
        return view;
    }
```

Tabla 5.26 - Método 'onCreateView()' del fragmento de búsqueda de películas

Al tratarse de un fragmento, y no de una actividad, ya no se extenderá la clase 'AppCompatActivity', sino que será la clase 'Fragment' la que se extienda. Por ello, los métodos heredados no serán los mismos, y en lugar del método 'onCreate()' se heredará 'onCreateView()'. Su funcionamiento, sin embargo, será muy similar.

Para los métodos onCreateView() de los fragmentos se utilizará la notación de jetbrains proporcionada por IntelliJ IDEA. También se llamará en ellos al método 'setUpView()', encargado de configurar la vista.

Método setUpView()

```
private void setUpView(View view){
        // Nos aseguramos que el botón de la NavigationBar está marcado correctamente.
        nav = getActivity().findViewById(R.id.navigationView);
        if(nav.getSelectedItemId() != R.id.user_lists_nav){
            nav.setSelectedItemId(R.id.user_lists_nav);
            getActivity().getSupportFragmentManager().popBackStack();
        }
...
```

```
//Comprobamos si el fragment se carga desde el registro
        Bundle b = this.getArguments();
        if (b != null) {
            boolean fromRegister = b.getBoolean("register", false);
            if(fromRegister){
                showWelcomeDialog();
            }
        }
…
```

```
rv.addOnItemTouchListener(
                new RecyclerViewClickListener(getActivity(), rv ,new RecyclerViewClickListener.On
ItemClickListener() {
                    @Override public void onItemClick(View view, int position) {
                        Fragment fragment = new WatchListDetailFragment();
                        FragmentTransaction transaction = getFragmentManager().beginTransaction()
;

                        Bundle bundle = new Bundle();
                        bundle.putSerializable("watchlist", adapter.getWatchList(position));
                        fragment.setArguments(bundle);

                        transaction.replace(R.id.frame_container, fragment);
                        transaction.addToBackStack(null);

                        transaction.commit();
                    }

                    @Override public void onLongItemClick(View view, int position) {
                        showDeleteDialog(position);
                    }
                })
        );
…
```

Tabla 5.27 - Fragmentos de código del método 'setUpView()' de la clase 'WatchListFragment.java'

El comportamiento de este método será similar para el resto de los fragmentos implementados. El primer fragmento de código que mostramos se asegurará de que el botón seleccionado en la barra de navegación se corresponda con el que representa esta pantalla. Esto se introdujo al observar que al navegar hacia atrás en la aplicación sí se cambiaban las pantallas, pero no se actualizaba la barra de navegación. También es necesario sacar de la pila de ejecución el fragment anterior, pues se añadirá uno igual al ejecutar la sentencia '`nav.setSelectedItemId(R.id.user_lists_nav);`'.

A continuación, se muestra la comprobación que se realiza para conocer el origen desde donde se inició el fragmento. En concreto, se mirará si se ha iniciado desde la actividad de registro, pues a los nuevos usuarios se les mostrará, mediante un cuadro de diálogo, una breve guía introductoria.

Finalmente, podemos ver la implementación del listener a la vista RecyclerView. Al pulsar sobre una tarjeta que representa a una watchlist, se cargará el fragmento de su detalle, pasándole la misma como argumento. Si mantenemos pulsado, se mostrará un cuadro de diálogo pidiendo confirmación para el borrado de la watchlist.

Método addWatchListServer()

```
private void addWatchListServer(){
        Call<ResponseBody> call = client.addWatchList(idUser, id_wl, wlname, nickname, pub);
        call.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
            }

            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
                Log.d("TAG1", "Error: " + t.getMessage());
            }
        });
    }
```

Tabla 5.28 - Método 'addWatchListServer()' de la clase 'WatchListFragment.java'

Como ya hemos dicho, este fragmento se encargará de la administración de las watchlists personales del usuario, lo cual incluye su creación y borrado (tanto en el cliente como en el servidor). Aquí podemos ver el método empleado para crear la watchlist en la base de datos del servidor. La implementación del método de borrado se omitirá por su similaridad con este, solo difiere en la llamada realizada.

Método onDestroy()

```
@Override
public void onDestroy() {
        super.onDestroy();
        if(getActivity().getSupportFragmentManager().getBackStackEntryCount() == 0){
            getActivity().finish();
        }
    }
```

Tabla 5.29 - Método 'onDestroy()' de la clase 'WatchListFragment.java'

Este método se heredará de la clase Fragment y se implementará para finalizar la actividad principal al mismo tiempo que finalice el último fragment de la *BackStack.* Esto se decidió hacer al observar que en el último paso de la navegación hacia atrás se quedaba la actividad principal sin ningún fragment cargado, mostrando tan solo la barra de navegación y una pantalla vacía.

WatchListDetailFragment.java

Ante el evento de 'click' del usuario sobre una de las tarjetas de nuestra vista, se iniciará un nuevo fragmento para mostrar el detalle de la watchlist elegida, es decir, las películas que en ella se encuentren. Desde este fragmento se podrá acceder al detalle de las películas (lo veremos en el siguiente punto), retirar películas de la watchlist, a través de un cuadro de diálogo al mantener pulsado sobre ellas; y abrir el detalle de una de sus películas pulsando un botón con el símbolo de aleatoriedad.

Método setUpView()

```
if(adapter.getItemCount()<2){
            fabRandom.setVisibility(View.GONE);
        }else{
            fabRandom.setOnClickListener(v -> {
                Random r = new Random();
                int random = r.nextInt(adapter.getItemCount());
                getActivityDetail(random);
            });
        }
```

Tabla 5.30 - Fragmento de código del método 'setUpView() de la clase 'WatchListDetailFragment.java'

En este fragmento de código podemos ver cómo se realiza la comprobación del número de películas en la lista para ocultar el botón de aleatorio al ser menor de dos. Esta comprobación también se realizará en el método de retirada de la película.

Implementación en el servidor

La funcionalidad implementada descrita realizará peticiones al servidor para la creación y borrado de watchlists, y la retirada de películas de ellas. Estas peticiones se llevarán a cabo mediante los siguientes métodos.

Métodos de gestión de watchlists personales, watchlist_controller.ex

```
def create(conn, %{"watchlist" => watchlist_params}) do
    changeset = Watchlist.changeset(%Watchlist{}, watchlist_params)
    {:ok, _watchlist} = Repo.insert(changeset)

    conn
    |> put_flash(:info, "WatchList created!")
    |> put_status(:ok)
    |> send_resp(:ok, "")
  End
```

```
def deleteWatchlist(conn, %{"id_user" => id_user, "id_wl" => id_wl}) do
    from(wlc in Watchlistcontent, where: wlc.id_user == ^id_user and wlc.id_wl == ^id_wl) |> Repo
.delete_all
    from(wl in Watchlist, where: wl.user_id == ^id_user and wl.id_wl== ^id_wl) |> Repo.delete_all
    from(sub in Subscription, where: sub.id_creator == ^id_user and sub.id_wl == ^id_wl) |> Repo.
delete_all
    conn
    |> put_status(:ok)
    |> send_resp(200, "ok")
  end
```

```
def deleteWatchlistcontent(conn, %{"id_user" => id_user, "id_wl" => id_wl, "id_mv" => id_mv}) do
    from(wlc in Watchlistcontent, where: wlc.id_user == ^id_user and wlc.id_wl == ^id_wl and wlc.
id_mv == ^id_mv) |> Repo.delete_all
    conn
    |> put_status(:ok)
    |> send_resp(200, "ok")
  end
```

Tabla 5.31 - Métodos de gestión de watchlists personales, watchlist_controller.ex

Métodos para la inserción, borrado y retirada de películas de una watchlist, en ese orden.

5.6. Búsqueda de películas

Para acceder al fragmento del buscador de películas el usuario debe seleccionar el botón central de la barra de navegación, representado con el icono de una lupa. Será entonces cuando el fragmento *'SearchFragment.java'* sea cargado y mostrado en pantalla.

SearchFragment.java

Este fragmento será el que albergue la funcionalidad relacionada con la búsqueda de películas. Tendrá una barra de búsqueda en la zona superior y el resto de la pantalla estará reservado para mostrar los resultados. Para evitar que se muestre este apartado vacío, antes de realizar ninguna búsqueda se ha introducido una sección que muestra las últimas películas añadidas por el usuario.

Esta clase extenderá *'Fragment'* e implementará *'SearchView.OnQueryTextListener'* para establecer un listener a la barra de búsqueda.

Método setUpView()

```
// Barra de búsqueda y su listener
        svSearch = view.findViewById(R.id.svSearch);
        svSearch.setOnQueryTextListener(this);

        LinearLayoutManager lim = new LinearLayoutManager(getActivity());
        lim.setOrientation(RecyclerView.VERTICAL);
        recyclerView.setLayoutManager(lim);
        recyclerView.setAdapter(adapter);
        recyclerView.addOnItemTouchListener(
                new RecyclerViewClickListener(getActivity(), recyclerView,new RecyclerViewClickLi
stener.OnItemClickListener() {
                    @Override public void onItemClick(View view, int position) {
                        Intent intent = new Intent (getActivity(), MovieDetailActivity.class);
                        intent.putExtra("movie", adapter.getMovie(position));
                        startActivity(intent);
                    }

                    @Override public void onLongItemClick(View view, int position) { }
                })
        );
```

Tabla 5.32 - Fragmento de código del método 'setUpView()' de 'SearchFragment.java'

En este fragmento de código podemos ver cómo iniciamos la barra de búsqueda y establecemos un listener a los elementos del RecyclerView, el cual nos permitirá acceder a la información de las distintas películas mostradas.

Método onQueryTextSubmit()

```
@Override
public boolean onQueryTextSubmit(String query) {
        if(query.length()>=2){
            errorLayout.setVisibility(View.GONE);
            adapter.clearData();
            svSearch.clearFocus();
            recentTv.setVisibility(View.GONE);
            tmdbPetition(query);

            new Handler(Looper.getMainLooper()).postDelayed(() -> {
                if (adapter.getItemCount() == 0){
                    errorLayout.setVisibility(View.VISIBLE);
                }
            }, 1500);
        }
        return false;
    }
```

Tabla 5.33 - Método 'onQueryTextSubmit()' de la clase 'SearchFragment.java'

Este método se ejecutará cuando el usuario busque un término en la barra superior, siempre y cuando tenga una longitud de al menos dos caracteres. Si se cumple este requisito se ocultará el apartado de últimas películas añadidas y se lanzará la petición a la API TMDB. Si en 1,5 segundos no ha encontrado ningún resultado se le notificará al usuario por pantalla.

Método tmdbPetition()

```
private void tmdbPetition(String input){
        int page = 1;

        HttpLoggingInterceptor loggingInterceptor = new HttpLoggingInterceptor().setLevel(HttpLog
gingInterceptor.Level.BODY);
        OkHttpClient.Builder httpClientBuilder = new OkHttpClient.Builder().addInterceptor(loggin
gInterceptor);
        Retrofit retrofit = new Retrofit.Builder()
                .baseUrl(TMDB_BASE_URL)
                .addConverterFactory(GsonConverterFactory.create())
```

```
                .client(httpClientBuilder.build())
                .build();

        client = retrofit.create(APIClient.class);
        Call<TmdbData> call = client.getTmdbMovie(TMDB_API_KEY, language, input, page, adult);
        call.enqueue(new Callback<TmdbData>() {
            @Override
            public void onResponse(Call<TmdbData> call, Response<TmdbData> response) {
                tmdbMovies = response.body().getResults();
                int totalPages = response.body().getTotalPages();
                if(tmdbMovies.size()!=0){
                    for(int page = 2; page <= totalPages && page <= 3; page++){
                        newTmdbPetition(page, input);
                    }
                    omdbPetition(movieFilter(tmdbMovies, input));
                }else{
                    errorLayout.setVisibility(View.VISIBLE);
                }
            }

            @Override
            public void onFailure(Call<TmdbData> call, Throwable t) {
                Log.d("TAG1", "Error: " + t.getMessage());
            }
        });
    }
```

Tabla 5.34 - Método 'tmdbPetition()' de la clase 'SearchFragment.java'

En este método se realizará la primera petición a la API de TMDB, a través de la interfaz APIClient. En el resultado nos devolverá el número de páginas totales con resultados, que contendrán hasta 20 cada una. Para el resto de páginas se llamará al método *'newTmdbPetition()'*, cuya implementación será muy similar a la de este salvo por el tratamiento de las páginas.

También se llamará al método *'omdbPetition()'* con las películas obtenidas y filtradas. La implementación de este método se obviará por su similaridad con este. En este método se realizará una petición a la API de OMDB por cada película obtenida para ampliar la información sobre ella. Para ello, se utilizará el título original de la película, pues no admite búsquedas en otros idiomas.

Función movieFilter()

```
private List<TmdbMovie> movieFilter(List<TmdbMovie> tmdbMovies, String input){

        List<TmdbMovie> filteredMovies = new ArrayList<>();

        tmdbMovies.forEach((TmdbMovie movie) -> {
            String poster = movie.getPosterPath();
            String backdrop = movie.getBackdropPath();
            String overview = movie.getOverview();
            Double popularity = movie.getPopularity();
            Boolean contained = movie.getTitle().toLowerCase().contains(input.toLowerCase()) ||
movie.getOriginalTitle().toLowerCase().contains(input.toLowerCase());
            if(poster!=null && backdrop!=null && overview!=null && popularity>10 && contained) {
                filteredMovies.add(movie);
            }
        });
        return filteredMovies;
    }
```

Tabla 5.35 - Función 'movieFilter()' de la clase 'SearchFragment.java'

Por último hablaremos de la función *'movieFilter()'*, que será la encargada de filtrar los resultados de las peticiones a TMDB para dejar los de más relevancia. Para ello se fijará en que sus campos estén informados y en el valor del atributo 'popularidad'.

MovieDetailActivity.java

Al pulsar sobre la tarjeta que representa una película, el listener implementado en el RecyclerView iniciará la actividad *'MovieDetailActivity.java'*, en la cual se mostrará la información sobre la película en cuestión. La implementación de esta actividad se obviará al carecer de relevancia por ser la representación de una vista meramente informativa. La única posibilidad de interacción con el usuario es mediante un botón con el símbolo '+', que solo se mostrará si se ha accedido a esta actividad desde el fragmento de búsqueda.

Método setUpView()

```
boolean fromWl = getIntent().getBooleanExtra("WatchList", false);
        if(!fromWl) {
            fabMovie.setOnClickListener(v -> {
                Intent intent = new Intent(this, AddMovieActivity.class);
                intent.putExtra("movie", movie);
                startActivity(intent);
            });
        }else{
            fabMovie.setVisibility(View.GONE);
        }
```

Tabla 5.36 - Fragmento de código del método 'setUpView()' de la actividad 'MovieDetailActivity.java'

Como podemos ver, a este actividad se le indica su origen mediante el paso de un argumento como extra. Si se accede a la película desde alguna watchlist no se mostrará el botón. Si, por el contrario, se ha accedido a ella desde el fragmento de búsqueda se mostrará el botón y se le añadirá un listener para detectar el click del usuario e iniciar la actividad *'AddMovieActivity.java'*.

AddMovieActivity.java

Actividad que muestra las watchlists del usuario para que este elija en cuál desea añadir la película. También contará con un botón '+' para poder crear una nueva watchlist al momento. Es muy similar al fragmento *'WatchListFragment.java'* descrito en el apartado anterior, tan solo varía en el listener del RecyclerView y en las funciones invocadas desde este.

Método setUpView()

```
rv.addOnItemTouchListener(
                new RecyclerViewClickListener(this, rv ,new RecyclerViewClickListener.OnItemClick
Listener() {
                    @Override public void onItemClick(View view, int position) {
                        WatchList selected = adapter.getWatchList(position);
                        //  Añadimos la película y su relación con la WatchList a la base de dato
s interna (SQLite)
                        MovieContract.addMovieDb(movie, db);
                        WLContentContract.addMovieToWl(selected, movie, db);
```

```
                    //  Añadimos la película a la BD del servidor
                    addMovieServer();
                    addMovieToWatchListServer(position);
                    finish();
                }

                @Override public void onLongItemClick(View view, int position) { }
            })
        );
```

Tabla 5.37 - Fragmento de código del método 'setUpView()' de la clase 'AddMovieActivity.java'

Como podemos ver, tan solo se atenderá el evento de *click* en el elemento. Esto desembocará en la adición de la película y de su pertenencia a la watchlist elegida a la base de datos local del cliente y a la del servidor, para, por último, finalizar la actividad.

Método addMovieServer()

```
private void addMovieServer(){
        Call<ResponseBody> call = callMovieConstructor();
        call.enqueue(new Callback<ResponseBody>() {
            @Override
            public void onResponse(Call<ResponseBody> call, Response<ResponseBody> response) {
            }

            @Override
            public void onFailure(Call<ResponseBody> call, Throwable t) {
                Log.d("ERROR", t.getMessage());
            }
        });
    }
```

Tabla 5.38 - Método 'addMovieServer()' de la clase 'AddMovieActivity.java'

Este método realizará una petición de tipo POST al servidor con los datos de la película. Debido a la complejidad de fijar estos datos al provenir de dos fuentes distintas (TMDB y OMDB), se ha realizado la construcción de esta llamada en la función *'callMovieConstructor()'*.

El método*'addMovieToWatchListServer()'* es igual que el anterior solo que con un constructor de llamada distinto.

Implementación en el servidor

Para el desarrollo de esta parte de la funcionalidad del sistema se han implementado los siguientes métodos en el servidor.

Método create(), movie_controller.ex

```
def create(conn, %{"movie" => movie_params}) do
    changeset = Movie.changeset(%Movie{}, movie_params)
    {:ok, movie} = Repo.insert(changeset)

    conn
    |> put_flash(:info, "#{movie.title} created!")
    |> put_status(:ok)
    |> send_resp(:ok, "")
  end
```

Tabla 5.39 - Método implementado en el servidor para la adición de películas en base de datos

Este método será llamado desde el cliente en la función *'addMovieServer()'*. Su finalidad es insertar la película en la base de datos. Este método será ejecutado cada vez que un usuario añada una película a una watchlist, pero solo se almacenará una vez en la base de datos. Los posteriores intentos de inserción de una película ya añadida producirán un error SQL pero que, gracias a nuestro entorno tolerante a fallos, no comprometerá el funcionamiento del sistema.

Método addMovie(), watchlist_controller.ex

```
def addMovie(conn, %{"watchlistcontent" => watchlistcontent_params}) do
    changeset = Watchlistcontent.changeset(%Watchlistcontent{}, watchlistcontent_params)
    {:ok, _watchlist} = Repo.insert(changeset)

    conn
    |> put_flash(:info, "movie added!")
    |> put_status(:ok)
    |> send_resp(:ok, "")
  end
```

Tabla 5.40 - Método implementado en el servidor para añadir películas a una watchlist concreta

Al insertar las películas de manera única y no por cada watchlist tendremos una relación de 'muchas a muchas' entre ellas. Para ello se establece la tabla *'Watchlistcontent'*, en la que se establecerán las relaciones de pertenencia entre las watchlists y las películas añadidas a ellas.

5.7. Gestión de suscripciones del usuario

Esta funcionalidad tiene lugar en la última pantalla de la aplicación, y para acceder a ella pulsaremos en el botón de más a la derecha de la barra de navegación. Al hacerlo, se cargará el fragmento *'SocialFragment.java'*.

SocialFragment.java

En este fragmento el usuario podrá gestionar sus suscripciones. Para ello, contará con una barra de búsqueda en la parte superior (para buscar entre las watchlists públicas del sistema) y mostrará las suscripciones activas del usuario debajo de esta.

Las watchlists suscritas del usuario no serán almacenadas en la base de datos local del cliente, y se realizará una serie de peticiones al servidor para obtenerlas cada vez que se acceda a esta pantalla. Esta decisión fue tomada para conseguir una correcta actualización del estado de las watchlists suscritas del usuario.

Método getSubscriptions()

```
private void getSubscriptions(){
        HashMap<String, String> user = sm.getUserDetails();
        int idUser = Integer.parseInt(user.get("id"));
        Call<WatchListData> call4 = client.getSubscriptionWls(idUser);
        call4.enqueue(new Callback<WatchListData>() {
            @Override
            public void onResponse(Call<WatchListData> call, Response<WatchListData> response) {
                Log.d("RESPONSE_CODE", String.valueOf(response.code()));
                if(response.code()==200) {
                    if(response.body().getWatchLists().size()!=0){
                        errorLayout.setVisibility(View.GONE);
                        List<WatchList> wls = response.body().getWatchLists();
                        if(wls.size()!=0){
                            getPublicMovies(wls);
                        }
                    }else{
                        errorLayout.setVisibility(View.VISIBLE);
```

```
                    }
                }else if(response.code() == 401){
                    Log.d("ERROR 401 ", response.message());
                    errorLayout.setVisibility(View.VISIBLE);
                }
            }
            @Override
            public void onFailure(@NotNull Call<WatchListData> call, Throwable t) {
                Log.d("CONNECTION_ERROR", t.getMessage());
                errorLayout.setVisibility(View.VISIBLE);
            }
        });
    }
```

Tabla 5.41 - Método 'getSubsctiptions()' de la clase 'SocialFragment.java'

Este método será llamado desde *'onCreateView()'*, es decir, al crearse la vista. Realizará una petición al servidor para obtener las watchlists a la que se encuentra suscrito el usuario. Si la respuesta de la petición es satisfactoria, realizará peticiones para obtener las películas que se encuentran en ellas (una por lista suscrita).

Método setUpView()

```
rv.addOnItemTouchListener(
                new RecyclerViewClickListener(getActivity(), rv ,new RecyclerViewClickListener.On
ItemClickListener() {
                    @Override public void onItemClick(View view, int position) {
                        Fragment fragment = new PublicWLDetailFragment();
                        FragmentTransaction transaction = getFragmentManager().beginTransaction()
;

                        Bundle bundle = new Bundle();
                        bundle.putSerializable("watchlist", adapter.getWatchList(position));
                        fragment.setArguments(bundle);

                        transaction.add(R.id.frame_container, fragment);
                        transaction.addToBackStack("SocialFragment");
                        transaction.commit();
                    }

                    @Override public void onLongItemClick(View view, int position) {
                        if(!fromSearch){
                            showUnsubscribeDialog(position);
                        }
                    }
                })
        );
```

Tabla 5.42 - Fragmento de código del método 'setUpViiew()' de la clase 'Socialfragment.java'

En este fragmento de código mostramos el listener implementado para la vista RecyclerView. Esto nos indica que, con un solo toque del usuario sobre la watchlist suscrita podrá acceder a su detalle, mientras que si mantiene pulsado podrá darse de baja a través de un cuadro de diálogo de confirmación.

Método onQueryTextSubmit()

```
@Override
public boolean onQueryTextSubmit(String input) {
        if(input.length()>=2){
            errorLayout.setVisibility(View.GONE);
            adapter.clearData();
            fromSearch = true;
            subsTv.setVisibility(View.GONE);
            svPublic.clearFocus();
            searchPublicWatchLists(input);
        }
        return false;
    }
```

Tabla 5.43 - Método 'onQueryTextSubmit()' de la clase 'SocialFragment.java'

Este método será heredado del listener de la barra de búsqueda y se ejecutará cuando el usuario busque un término. Si dicho término tiene una longitud mayor o igual a dos caracteres, se ocultarán sus suscripciones activas en pantalla y se realizará la petición al servidor. Si esta petición es satisfactoria se le mostrarán los resultados en el mismo fragment, y si no se mostrará un mensaje de aviso al usuario.

Método searchPublicWatchLists()

```
private void searchPublicWatchLists(String input){
        Call<WatchListData> call = client.getPublicWatchLists(input);
        call.enqueue(new Callback<WatchListData>() {
            @Override
            public void onResponse(Call<WatchListData> call, Response<WatchListData> response) {
                publicWls = response.body().getWatchLists();
                if(publicWls.size()!=0){
                    getPublicMovies(publicWls);
                }else{
                    errorLayout.setVisibility(View.VISIBLE);
                }
            }

            @Override
            public void onFailure(Call<WatchListData> call, Throwable t) {
```

```
                Log.d("TAG1", "Error: " + t.getMessage());
            }
        });
    }
```

Tabla 5.44 - Método 'searchPublicWatchLists()' de la clase 'SocialFragment.java'

Con este método enviaremos la petición de búsqueda al servidor. Como ya se ha mencionado, el término de búsqueda puede hacer referencia al nombre de la watchlist o al del usuario que la creo. Ante una respuesta satisfactoria haremos una petición al servidor por cada watchlist para obtener sus películas en el método *'getPublicMovies()'.* Por cuestiones de calidad las watchlists que no contengan películas no serán mostradas al usuario, a pesar de que coincida con el término introducido.

PublicWLDetailFragment.java

Este fragmento mostrará el contenido de las watchlists públicas. Su implementación es muy similar a la del fragmento *'WatchListDetailFragment'* descrito en el punto anterior, pero contará con métodos que gestionen la suscripción y su estado de las watchlists.

En la parte superior se mostrará el *nick* del creador y si se está siguiendo ya la lista. También comprobará si la watchlist es propiedad del usuario. Debajo se mostrarán las películas contenidas en la watchlist en cuestión y un botón para suscribirse en caso de no estarlo ya y de no ser propiedad del usuario.

Implementación en el servidor

Para implementar esta funcionalidad en el servidor, se utilizarán los siguientes métodos. Todos ellos se encontrarán en el controlador de watchlists *'watchlist_controller.ex'.*

Método searchPublicWatchlist()

```
def searchPublicWatchlist(conn, %{"input" => input}) do
    like = "%#{input}%"
    query = from wl in Watchlist,
            where: wl.public == true
            and (like(wl.name, ^like) or like(wl.autor, ^like)),
            select: wl
    publics = Repo.all(query)
    conn
    |>json(%{watchlists: publics})
  end
```

Tabla 5.45 - Método 'searchPublicWatchlists()' del controlador 'watchlist_controller.ex'

Método que responde a la petición de búsqueda y devuelve los resultados que coinciden con el término introducido. Para ello se ha establecido que el término de búsqueda deba estar contenido en el nombre de la watchlist o del usuario, y, por supuesto, solo se devolverán watchlists públicas.

Método getSubscriptionWatchlists()

```
def getSubscriptionWatchlists(conn, %{"id_user" => id_user}) do
    query = from sub in Subscription,
            join: wl in Watchlist,
            on: sub.id_creator == wl.user_id,
            where: sub.id_user == ^id_user
            and sub.id_wl == wl.id_wl,
            select: wl
    subs = Repo.all(query)
    conn
    |> json(%{watchlists: subs})
  end
```

Tabla 5.46 - Método 'getSubscriptionWatchlists()' del controlador 'watchlist_controller.ex'

Método que devuelve todas las watchlists a las que se encuentra suscrito un usuario.

Método getPublicMovies()

```
def getPublicMovies(conn, %{"id_creator" => id_creator, "id_wl" => id_wl}) do
    query = from wlc in Watchlistcontent,
            join: m in Movie,
            on: wlc.id_mv == m.id,
            where: wlc.id_user == ^id_creator and wlc.id_wl == ^id_wl,
            select: m
    mvs = Repo.all(query)
    conn
    |> json(%{movies: mvs})
  end
```

Tabla 5.47 - Método 'getPublicMovies()' del controlador 'watchlist_controller.ex'

Este método devolverá las películas pertenecientes a la watchlist que se pida. Se ejecutará una vez por cada watchlist a la que se encuentre suscrito el usuario.

Método subscribeToWatchlist()

```
def subscribeToWatchlist(conn, %{"subscription" => subscription_params}) do
      changeset = Subscription.changeset(%Subscription{}, subscription_params)
      {:ok, _subscription} = Repo.insert(changeset)

      conn
      |> put_flash(:info, "subscription added!")
      |> put_status(:ok)
      |> send_resp(:ok, "")
  end
```

Tabla 5.48 - Método 'subscribeToWatchlist()' del controlador 'watchlist_controller.ex'

Este método permitirá al usuario suscribirse a una watchlist. Para ello, introducirá un nuevo registro en la tabla de suscripciones indicando el identificador de la watchlist y del usuario suscrito.

Método unsubscribe()

```
def unsubscribe(conn, %{"id_user" => id_user, "id_creator" => id_creator, "id_wl" => id_wl}) do
    from(sub in Subscription, where: sub.id_user == ^id_user and sub.id_creator == ^id_creator an
d sub.id_wl == ^id_wl) |> Repo.delete_all
    conn
    |> put_status(:ok)
    |> send_resp(200, "ok")
  end
```

Tabla 5.49 - Método 'unsubscribe()' del controlador 'watchlist_controller.ex'

Este método permitirá al usuario darse de baja de una watchlist eliminando el correspondiente registro de la tabla de suscripciones.

Capítulo 6

Pruebas

En esta sección hablaremos de las pruebas realizadas para comprobar la calidad de nuestro sistema. Comenzaremos explicando los dos tipos que hay, para después introducir nuestros casos de prueba.

6.1. Pruebas de caja blanca

Las pruebas de caja blanca hacen referencia a aquellas que se centran en los detalles procedimentales del software, es decir, a los flujos de ejecución del código. Estas pruebas se centran principalmente en:

- Garantizar un correcto flujo de control, asegurándose que no existen caminos independientes en nuestro sistema que no lleguen a ejecutarse.
- Comprobación de los condicionales establecidos
- Asegurarse que todos los bucles tienen un límite operacional

Sin embargo, no se entrará en detalle de las pruebas de caja blanca realizadas para nuestro sistema ya que se han ejecutado durante el proceso de desarrollo y no resultan significativas para la comprensión del sistema.

6.2. Pruebas de caja negra

En el contexto de la ingeniería del software se conocen como pruebas de caja negra a aquellas que se encargan de validar los datos de entrada de un sistema y los resultados que genera. Para realizar este tipo de pruebas no se tiene en cuenta la estructura interna del software del sistema, sino que se establecerán en base a los casos de uso establecidos para asegurar el cumplimiento de las especificaciones funcionales.

A continuación, se detallarán los casos de prueba establecidos y ejecutados para comprobar la funcionalidad del sistema.

CP-01	Registro
Descripción	El usuario rellena correctamente el formulario de registro y pulsa el botón de registrarse
Actor	Usuario no registrado
Requisitos	-
Resultado esperado	Se crea un nuevo usuario en el sistema y se le muestra un cuadro de diálogo emergente con información sobre el uso de la aplicación.
Resultado	Ok

Tabla 6.1 - Caso de prueba 1, registro

CP-02	Iniciar sesión
Descripción	El usuario introduce su email y su contraseña en el formulario de login y pulsa el botón de iniciar sesión.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema.
Resultado esperado	Se establece sesión con el servidor y los datos de las watchlists del usuario son mostrados en pantalla.
Resultado	Ok

Tabla 6.2 - Caso de prueba 2, iniciar sesión

CP-03	Finalizar sesión
Descripción	El usuario navega hasta la pantalla 'Social WatchLists' y pulsa el botón de finalizar sesión. Confirmará su acción pulsando 'Sí' en el cuadro de diálogo emergente
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema.
Resultado esperado	La base de datos local es borrada y se redirige al usuario a la pantalla de login.
Resultado	Ok

Tabla 6.3 - Caso de prueba 3, finalizar sesión

CP-04	Crear watchlist privada
Descripción	El usuario pulsa el botón '+' en la pantalla 'WachLists', introduce un nombre en el cuadro de diálogo emergente y pulsa aceptar.

Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema.
Resultado esperado	Se muestra la watchlist creada en la pantalla y se inserta en la base de datos del servidor.
Resultado	Ok

Tabla 6.4 - Caso de prueba 4, crear watchlist privada

CP-05	Crear watchlist pública
Descripción	El usuario pulsa el botón '+' en la pantalla 'WachLists', introduce un nombre en el cuadro de diálogo emergente, selecciona el *checkbox* con etiqueta 'Pública' y pulsa aceptar.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema.
Resultado esperado	Se muestra la watchlist creada en la pantalla indicando su carácter público con un icono y se inserta en la base de datos del servidor. Esta watchlist deberá poder ser encontrada en la pantalla 'Social WatchLists' introduciendo su nombre en el buscador.
Resultado	Ok

Tabla 6.5 - Caso de prueba 5, crear watchlist pública

CP-06	Buscar película y añadirla a una watchlist
Descripción	El usuario navega hasta la pantalla 'Buscar película', introduce el título que desea buscar y selecciona la película deseada. A continuación pulsa el botón '+' y selecciona una watchlist de su elección *(en caso de no haber ninguna podría crear una pulsando el nuevo botón '+')*.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema. El usuario debe tener al menos una watchlist creada.
Resultado esperado	La película elegida se encuentra en la watchlist elegida y tanto la película como su relación con la watchlist se introducirán en la base de datos del servidor.
Resultado	Ok

Tabla 6.6 - Caso de prueba 6, buscar película y añadirla a una watchlist

CP-07	Retirar película de una watchlist
Descripción	El usuario navega hasta la pantalla 'WatchLists' y selecciona la lista de la que desea retirar la película. Mantiene pulsada la tarjeta que representa la película y pulsa 'Sí' en el cuadro de diálogo emergente.
Actor	Usuario registrado

Requisitos	El usuario debe tener una sesión activa en el sistema. El usuario debe tener al menos una watchlist creada.
Resultado esperado	La película ya no se muestra en la watchlist y su relación con ella se borra del servidor.
Resultado	Ok

Tabla 6.7 - Caso de prueba 7, retirar película de una watchlist

CP-08	Eliminar watchlist
Descripción	El usuario navega hasta la pantalla 'WatchLists' y mantiene pulsada la tarjeta que representa la lista que desea borrar. Se le muestra un cuadro de diálogo emergente y pulsa 'Sí'.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema. El usuario debe tener al menos una watchlist creada.
Resultado esperado	La watchlist ya no se mostrará en pantalla y tanto esta como todos sus datos asociados (relaciones de pertenencia de películas, suscripciones activas si fuese pública) son borrados de la base de datos del servidor.
Resultado	Ok

Tabla 6.8 - Caso de prueba 8, eliminar watchlist

CP-09	Elegir película aleatoriamente de una watchlist
Descripción	El usuario navega hasta la pantalla 'WatchLists' y selecciona una lista que contenga al menos dos películas. Pulsa el botón de 'aleatorio' y se le muestra el detalle de una de las películas de la lista.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema. El usuario debe tener al menos una watchlist creada con dos películas en ella como mínimo.
Resultado esperado	La película mostrada varía de entre las de la lista, llegando a mostrar todas, al pulsar varias veces el botón 'aleatorio'.
Resultado	Ok

Tabla 6.9 - Caso de prueba 9, elegir película aleatoriamente de una watchlist

CP-10	Buscar watchlist pública y suscribirse a ella
Descripción	El usuario navega hasta la pantalla 'Social WatchLists' e introduce un término en la barra de búsqueda. Se le muestran todas las listas públicas cuyo nombre o el de su creador contengan el término introducido. Pulsa sobre una de ellas a la que no esté siguiendo y no sea suya y a continuación pulsa el botón 'suscribirse' y confirma la acción en el cuadro de diálogo.

Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema.
Resultado esperado	La nueva watchlist se muestra por pantalla y se registra la suscripción en la base de datos del servidor.
Resultado	Ok

Tabla 6.10 - Caso de prueba 10, buscar watchlist pública y suscribirse a ella

CP-11	Darse de baja de una watchlist pública
Descripción	El usuario navega hasta la pantalla 'Social WatchLists' y mantiene pulsada la tarjeta que representa la lista de la que desea darse de baja. Se le muestra un cuadro de diálogo emergente y pulsa 'Sí'.
Actor	Usuario registrado
Requisitos	El usuario debe tener una sesión activa en el sistema. El usuario debe estar suscrito a al menos una watchlist pública.
Resultado esperado	La watchlist ya no se muestra en pantalla y la se elimina la suscripción de la base de datos del servidor.
Resultado	Ok

Tabla 6.11 - Caso de prueba 11, darse de baja de una watchlist

Capítulo 7

Conclusiones

7.1. Conclusiones

El objetivo principal a la hora de desarrollar este proyecto era entrar en contacto con el lenguaje de programación concurrente Elixir y, más concretamente, levantar un servidor utilizando el framework Phoenix.

Una vez conseguido este objetivo, el que suscribe puede afirmar que la experiencia con este entorno de programación ha sido bastante satisfactoria. El lenguaje es limpio y conciso, y el rendimiento final que ofrece es impecable, pues la mayoría de las peticiones son respondidas en un tiempo inferior a los 10 milisegundos.

Sin embargo, el gran obstáculo encontrado reside en la curva de aprendizaje, la cual puede volverse ardua por la escasa documentación que hay sobre este framework. Si bien se dispone de manuales y libros que ayudan a su comprensión y entendimiento, puedes verte desahuciado a la hora de solucionar tus propios errores buscando información sobre ellos en la red. Esto es fácilmente entendible al ver que la primera versión de este framework fue lanzada en 2014, mientras que las de sus competidores directos, como Ruby on Rails y Django, datan de 2004 y 2005 respectivamente.

En definitiva, el framework Phoenix ofrece una solución muy interesante para implementar el lado del servidor de los sistemas distribuidos, con un rendimiento y una gestión de la concurrencia sobresalientes que lo hacen destacar sobre sus competidores. Tan solo será necesario más tiempo en el mercado y un aumento de su comunidad y sus aportes para terminar de popularizar su uso.

7.2. Líneas de trabajo futuro

Debido a la falta de recursos de tiempo y personal no se han podido implementar todas las funcionalidades deseadas. En este punto, las relataremos como futuras líneas de trabajo que permitirán mejorar la experiencia de los usuarios del sistema.

Una función interesante que se quería implementar es la opción de añadir a otros usuarios como amigos en el sistema para poder compartir películas y crear listas colaborativas, en la que los participantes establecidos por el creador puedan también añadir películas. Esto podría hacerse utilizando los *'channels'* de Phoenix, los cuales permiten establecer una comunicación en tiempo real notificando a los miembros de estos canales de los cambios producidos en ellos.

También se intentaron implementar funcionalidades extra de la librería *'Pow'*, utilizada para la validación de los usuarios en el servidor, que vienen en forma de extensiones para esta librería. Estas nos ofrecen funciones como: validar el registro de los usuarios mediante un mail de confirmación, la posibilidad de cambiar la contraseña del usuario y técnicas de autenticación mediante OAuth 2.0. Como ya hemos dicho, esto no se terminó de implementar por falta de tiempo para integrarlo en nuestro sistema.

Bibliografía

[1] *The Movie DataBase,* URL: https://www.themoviedb.org/ (visitado 19-07-2021)
[2] *The Movie DataBase – Documentation API,* URL: https://www.themoviedb.org/documentation/api (visitado 19-07-2021)
[3] *Online Movie DataBase API,* URL: https://www.themoviedb.org/ (visitado 19-07-2021)
[4] *JavaTpoint – Android versions,* URL: https://www.javatpoint.com/android-versions (visitado 19-07-2021)
[5] *StatCounter – Mobile Operating System Market Share Worldwide,* URL: https://gs.statcounter.com/os-market-share/mobile/worldwide/ (visitado 19-07-2021)
[6] *StatCounter - Mobile & Tablet Android Version Market Share Worldwide,* URL: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/ (visitado 19-07-2021)
[7] *Erlang/OTP,* URL: https://www.erlang.org/ (visitado 19-07-2021)
[8] *HexDocs,* URL: https://hexdocs.pm/ (visitado 19-07-2021)
[9] *HexDocs – Phoenix,* URL: https://hexdocs.pm/phoenix/Phoenix.html (visitado 19-07-2021)
[10] *Stack Overflow – Where is Erlang used and why,* URL: https://stackoverflow.com/questions/1636455/where-is-erlang-used-and-why/ (visitado 19-07-2021)
[11] *Elixir in Action, Second Edition - Saša Jurić, Enero 2019,* URL: https://www.manning.com/books/elixir-in-action-second-edition (visitado 19-07-2021)
[12] *MonteRail – Eight Famous Companes using Elixir,* URL: https://www.monterail.com/blog/famous-companies-using-elixir (visitado 19-07-2021)
[13] *EcuRed – Pruebas de caja blanca,* URL: https://www.ecured.cu/Pruebas_de_caja_blanca / (visitado 19-07-2021)
[14] *PMOinformática – Pruebas de caja negra,* URL: http://www.pmoinformatica.com/2017/02/pruebas-de-caja-negra-ejemplos.html (visitado 19-07-2021)
[15] *Tutorial Retrofit 2 Parte1 en Español*, URL: https://www.youtube.com/watch?v=nwFMTX_ePFg (visitado 19-07-2021)
[16] *Tutorial Retrofit 2 Parte 2 en Español,* URL: https://www.youtube.com/watch?v=xDF4dpSbvhA (visitado 19-07-2021)
[17] *SVG Repo,* URL: https://www.svgrepo.com/ (visitado 19-07-2021)
[18] *Tutorial Android: Búsqueda en RecyclerView - SearchView filter,* URL: https://www.youtube.com/watch?v=TSnWWhUx1V4 (visitado 19-07-2021)
[19] *Stack Overflow – RecyclerView onClick,* URL: https://stackoverflow.com/questions/24471109/recyclerview-onclick (visitado 19-07-2021)
[20] *Android Hive - Android Working with Bottom Navigation,* URL: https://www.androidhive.info/2017/12/android-working-with-bottom-navigation/ (visitado 19-07-2021)