

Trabajo fin de máster

Desarrollo de una aplicación de recomendación de música para grupos



Miguel García González

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR



Máster Universitario en Ingeniería Informática

TRABAJO FIN DE MÁSTER

**Desarrollo de una aplicación de recomendación de
música para grupos**

Autor: Miguel García González
Tutor: Alejandro Bellogín Kouki

febrero 2024

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución comunicación pública y transformación de esta obra sin contar con la autorización de los titulares de la propiedad intelectual.

La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. del Código Penal*).

DERECHOS RESERVADOS

© 12 de Febrero de 2024 por UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, nº 1
Madrid, 28049
Spain

Miguel García González

Desarrollo de una aplicación de recomendación de música para grupos

Miguel García González

C\ Ana María N° 51

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mis padres, por su apoyo incondicional.

*Quien recibe un beneficio nunca debe olvidarlo;
quien lo otorga, nunca debe recordarlo.*

Pierre Charron

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mi familia, en especial a mis padres, todo el apoyo que me brindan día a día de manera incondicional. Las oportunidades que me han dado, como la de irme a estudiar al extranjero durante este máster, me han dado momentos, experiencias y aprendizaje que no olvidaré. A mis hermanos quiero agradecerles su experiencia y consejo, los cuales en ciertas ocasiones me han ayudado a tomar decisiones importantes, que me han llevado hasta aquí. Su reciente experiencia en estas etapas de la vida ha sido siempre y será un apoyo imposible de ignorar.

Quiero dar las gracias a mis amigos, por ser el círculo de influencia que son, porque me ayudan a seguir motivado y concentrado en lo que hay que hacer, y porque me han dado momentos de diversión y desconexión que me han ayudado a seguir adelante.

Quiero dar las gracias también por los compañeros del máster, con los que he aprendido mucho y he compartido muchas horas. Al final, el máster te brinda la experiencia de vivir la universidad de una forma bastante diferente al grado en mi opinión, y doy gracias por haberla vivido. Quiero dar las gracias a en general a todas las personas en las cuales consigo encontrar motivación, ya sea por su ejemplo, por su experiencia o por su consejo.

Por último, quiero dar las gracias a Alejandro Bellogín, mi tutor, por haberme dado la oportunidad de trabajar en este proyecto, por su apoyo, disponibilidad absoluta y por sus propuestas y guía durante la realización del trabajo.

RESUMEN

Que la tecnología inunda nuestras vidas tras haber progresado de manera vertiginosa es una obviedad. La música no es una excepción, y la aparición de plataformas de *streaming* como *Spotify* ha cambiado la forma en la que escuchamos música. La obtención de datos de escucha de los usuarios permite que se puedan perfilar sus gustos de música, dejando de estar todo el peso de la elección en el usuario, que antes tendría que buscar manualmente la música que él previamente tendría que haber descubierto que le gustase; a estar una gran parte del peso en el sistema, que recomendará la música que más se ajuste a los gustos registrados del usuario. A día de hoy *Spotify* traduce esta recomendación en listas de reproducción (*playlists*) generadas automáticamente y que se actualizan automáticamente. Los usuarios han pasado a tener un papel totalmente pasivo en estas recomendaciones, y aunque no se quiere poner en duda su atractivo o utilidad y eficacia, en este trabajo se quiere explorar una alternativa intermedia, en la cual, usuario y sistema tengan un papel en la generación de recomendaciones.

Para ello, se expondrá en este documento el desarrollo de una aplicación web que permite a los usuarios de *Spotify* generar listas de reproducción (*playlists*) de música que se basen en los gustos de un grupo de usuarios de *Spotify*. Se utilizará la API de *Spotify* para obtener datos de usuarios, canciones más escuchadas, y recomendaciones. Como *framework* de desarrollo se utilizará *Flutter* (desarrollado por *Google*), que permite desarrollar aplicaciones multiplataforma, con una misma base de código. En este caso se hará uso de la versión web de la aplicación.

Se llevarán a cabo pruebas con usuarios para evaluar la aplicación y distintas formas de generar *playlists*. Tras ello, se analizarán los resultados obtenidos para obtener *feedback*, conclusiones y posible trabajo futuro sobre el trabajo realizado.

EXPLICAR UN POCO LOS RESULTADOS Y CONCLUSIONES DE FORMA BREVE

PALABRAS CLAVE

Flutter, *Spotify*, recomendación, recomendación de grupos, estrategias de agregación

ABSTRACT

That technology inundates our lives after having progressed rapidly is an obvious fact. Music is no exception, and the emergence of streaming platforms like Spotify has changed the way we listen to music. Obtaining listening data from users allows their music preferences to be profiled, shifting the weight of choice from the user, who previously had to manually search for music they would have previously discovered they liked, to the system, which recommends music that best matches the user's registered preferences. Currently, Spotify translates this recommendation into automatically generated and updated playlists. Users have become completely passive in these recommendations, and although their appeal, usefulness, and effectiveness are not questioned, this work aims to explore an intermediate alternative; one in which the user and the system play a role in generating recommendations.

To this end, this document presents the development of a web application that allows Spotify users to generate playlists based on the preferences of a group of Spotify users. The Spotify API will be used to obtain user data, most listened songs, and recommendations. The development framework used will be Flutter (developed by Google), which allows for cross-platform application development with a single codebase. In this case, the web version of the application will be used.

Tests will be conducted with users to evaluate the application and different ways of generating playlists. Afterwards, the obtained results will be analyzed to gather feedback, draw conclusions, and identify possible future work based on the work done.

KEYWORDS

Flutter, Spotify, recommendation, group recommendation, aggregation strategies

ÍNDICE

1	Introducción	1
1.1	Motivación del proyecto	1
1.2	Propuesta y objetivos	2
1.3	Estructura del documento	2
2	Estado del arte	3
2.1	Framework de desarrollo de aplicaciones: <i>Flutter</i>	3
2.1.1	<i>Dart</i> , el lenguaje detrás de <i>Flutter</i>	4
2.1.2	Instalación y primer uso básico	4
2.1.3	<i>Widgets</i>	5
2.2	<i>Spotify</i> y su API Web	6
2.2.1	La API Web de <i>Spotify</i>	7
2.2.2	Contribución de <i>Spotify</i> a la investigación y desarrollo en el ámbito del análisis musical y la recomendación	7
2.3	Recomendación	8
2.3.1	Métodos y técnicas de recomendación	9
2.3.2	Tipos de sistemas de recomendación	11
2.3.3	Estrategias de agregación	13
2.3.4	Evaluación de un sistema de recomendación a grupos	14
2.4	Implicaciones para el trabajo	15
3	Desarrollo de nuestra aplicación	17
3.1	Análisis	17
3.1.1	Análisis de requisitos	17
3.1.2	Diagramas de casos de uso principales	18
3.2	Diseño de la aplicación	19
3.2.1	Diseño de la interfaz de usuario	19
3.2.2	Arquitectura del sistema	19
3.3	Implementación de la aplicación	28
3.3.1	Flujo de autenticación	30
3.3.2	Llamadas a la API de Spotify y procesamiento de respuestas	33
3.3.3	Generación de <i>playlist</i> combinada	35
3.3.4	Pruebas de evaluación	41
3.3.5	Resumen	44

4 Pruebas y resultados	47
4.1 Evaluaciones <i>offline</i>	47
4.1.1 Resultados obtenidos	47
4.1.2 Decisiones tras los resultados	51
4.2 Estudios con usuarios	52
4.2.1 Gestión de usuarios	52
4.2.2 Procedimiento de las pruebas con los usuarios	53
4.2.3 Contexto sobre el grupo de personas	53
4.2.4 Comparación de estrategias	54
4.2.5 Encuesta de usabilidad	54
4.2.6 Posible nueva funcionalidad en la aplicación	54
5 Conclusiones y trabajo futuro	55
Bibliografía	58

LISTAS

Listas de algoritmos

Listas de códigos

3.1	Lanzar autenticación en <i>Spotify</i>	30
3.2	Obtener <code>access_token</code> y <code>refresh_token</code>	31
3.3	Guardar información del usuario	32
3.4	Acceder a la información del usuario	33
3.5	Procesar objeto en formato <i>JSON</i>	34
3.6	Constructor personalizado de la clase <code>User</code>	34
3.7	Generar <i>playlist</i> combinada	35
3.8	Definición de los argumentos <code>seedArtists</code> y <code>seedTracks</code>	36
3.9	Obtención de recomendaciones para un usuario	37
3.10	Proceso de agregación	38
3.11	Obtención de <i>ratings</i> individuales	38
3.12	Diccionario de estrategias de agregación	39
3.13	Estrategia de agregación <i>average</i>	39
3.14	Ordenar recomendaciones	40
3.15	Cortar recomendaciones	40
3.16	Generación de estadísticas	41
3.17	Diccionario de similitud entre usuarios	42
3.18	Diccionario de recomendaciones	43
3.19	Diccionario de similitud entre estrategias para una duración	43
3.20	Formato del mapa <code>results</code>	44

Listas de cuadros

Listas de ecuaciones

4.1	Esto es un ejemplo de título de ecuación incluida la propia ecuación $\sum c_{ij} = \frac{a}{J^{adx}}$...	54
-----	--	----

Lista de figuras

2.1	Ejecución de un proyecto <i>Flutter</i>	5
2.2	Ejemplo de árbol de <i>widgets</i>	5
2.3	Implementación en código de un <i>StatelessWidget</i> y un <i>StatefulWidget</i>	6
3.1	Diagramas de casos de uso principales de la aplicación	19
3.2	Maquetas de la aplicación a desarrollar	20
3.3	Pantallas de la aplicación desarrollada	20
3.4	Diagrama de secuencia de interacción con la API de <i>Spotify</i>	21
3.5	Visión general <i>Spotify for Developers</i>	24
3.6	Flujo de autenticación PKCE.....	25
3.7	Obtener canciones o artistas más escuchados	26
3.8	Obtener recomendaciones	27
3.9	Crear playlist	28
3.10	Añadir canciones a <i>playlist</i>	29
4.1	Gráficas de similitud entre estrategias para un grupo de 2 personas con similitud en semillas	48
4.2	Gráficas de similitud entre estrategias para un grupo de 3 personas sin similitud en semillas	49
4.3	Gráficas de similitud entre estrategias para un grupo de 2 personas sin similitud en semillas	50
4.4	Gráficas de similitud entre estrategias para grupos de 3, 4 y 5 personas con similitud en semillas	51

Lista de tablas

3.1	Tabla resumen de los <i>endpoints</i> a utilizar	29
-----	--	----

Lista de cuadros

INTRODUCCIÓN

En este capítulo se expondrán la motivación que ha hecho que este trabajo se lleve a cabo, los objetivos que se proponen lograr con él, y finalmente, se explicará la estructura que seguirá este documento.

1.1. Motivación del proyecto

La motivación de este trabajo viene por dos vertientes principales. Por un lado, *Flutter* [1] es un framework de desarrollo de aplicaciones multiplataforma bastante nuevo, desarrollado por Google que ha ganado cierta popularidad y que ha llamado nuestra atención. Así que hacer un proyecto con él es una buena forma de ponerlo a prueba y ver qué tal funciona.

Por el otro lado, escuchar música forma parte de nuestro día a día, de manera directa, a través de la aplicación de *Spotify* [2]. Añadido a esto, percibimos que hoy en día en muchas aplicaciones, productos o servicios, el usuario tienen un papel muy pasivo, y que en el caso de *Spotify* esto se ha acentuado con el paso del tiempo. Los usuarios reciben distintos tipos de *playlists* generadas periódicamente: novedades de la semana, fusionadas con otros usuarios que se actualizan semanalmente, predeterminadas para estados de ánimo, etc. Por supuesto, los usuarios pueden crear sus propias *playlists*, pero la forma de hacerlo es bastante manual, y desde luego la adición de canciones se hace de una en una.

Por ello, planteamos explorar una alternativa intermedia, en la cual, usuario y sistema comparten el peso de la creación de las *playlists*. Podrán juntarse varios usuarios a la vez para crear una *playlist* que trate de ajustarse a los gustos de todos. Incluso, generar una *playlist* que se base en sus, por ejemplo, tres canciones más escuchadas recientemente. La principal motivación es ofrecer al usuario, y sobre todo a aquel que le guste cobrar un papel más activo en sus elecciones, una forma fácil de obtener canciones a escuchar.

Finalmente, en este trabajo entra en escena la recomendación, y más específicamente, la dirigida a grupos, ya que se buscará también hacer un pequeño estudio sobre las distintas formas de generar *playlists* a partir de los gustos de un grupo de usuarios. Los experimentos e investigación recaerán

principalmente en las distintas estrategias de agregación, las cuales se diferencian en cómo tienen en cuenta las preferencias individuales de los usuarios para generar una *playlist* que se ajuste a todos. Se utilizará la API de Spotify [3] para obtener datos de usuarios, canciones más escuchadas, y recomendaciones, que aportarán la base para la generación de *playlists*, ya que, a partir de ellas, aplicando distintas estrategias de agregación, se obtendrán las *playlists* finales.

1.2. Propuesta y objetivos

Para este trabajo se tiene como objetivo principal el desarrollo de una aplicación web que permita a grupos de usuarios de Spotify crear *playlists* que combinen sus gustos musicales, ofreciendo una alternativa a las *playlists* que la aplicación de Spotify les pueda ofrecer. Tras ello, queremos llevar a cabo el estudio y posterior comparación de distintas estrategias de agregación para la generación de *playlists*.

Derivados de este objetivo principal, se plantean los siguientes objetivos específicos:

- Estudiar sobre el desarrollo de aplicaciones con *Flutter*, y familiarizarse con la base de su desarrollo, los *widgets*.
- Estudiar la API de Spotify, leyendo su documentación y haciendo pruebas, para acabar desarrollando el módulo de llamadas a la API.
- Desarrollar la aplicación web, que consistirá en un *frontend* que se sirva de la API de Spotify, que permita a grupos de usuarios de Spotify crear *playlists* que combinen sus gustos musicales.
- Realizar un pequeño estudio sobre la aplicación desarrollada, llevando a cabo pruebas con usuarios reales, para obtener conclusiones sobre la usabilidad de la aplicación y las potenciales estrategias para la generación de *playlists*.

1.3. Estructura del documento

ESTRUCTURA DEL DOCUMENTO

ESTADO DEL ARTE

En este capítulo se hará un recorrido por los actores principales de este trabajo: *Flutter*, la API de *Spotify*, y la recomendación dirigida a grupos. Se explicará qué son, cómo funcionan, y una aproximación de cómo hemos aplicado estos conceptos en el trabajo. Estos componentes son los que han conformado el proyecto de aplicación que finalmente se ha desarrollado, y que se explicará en el Capítulo 3.

2.1. Framework de desarrollo de aplicaciones: *Flutter*

Flutter [1] es un *framework* de desarrollo de código abierto de aplicaciones multiplataforma desarrollado por *Google*. Es una alternativa que tiene cierta popularidad, y que, como comentábamos en la introducción, ha llamado nuestra atención.

Con este *framework*, se pueden desarrollar aplicaciones para diversas plataformas, con una misma base de código. Desde aplicaciones móviles en *iOS* y *Android*, hasta aplicaciones de escritorio para *Windows*, *Linux* o *MacOS*. Sin olvidarnos de las aplicaciones *Web* (como la aplicación desarrollada en este trabajo), o incluso dispositivos embebidos, como los que pueden ser el infoentretenimiento de un coche [4].

Aunque solamente este hecho aporta un atractivo muy considerable a esta tecnología, *Flutter* cuenta con otras características a tener en cuenta:

- **Rendimiento:** utiliza el lenguaje de programación *Dart* [5], que se compila a código nativo, lo que aporta una ventaja en rendimiento clara. Comentaremos más sobre las características de este lenguaje en la Sección 2.1.1.
- **Productividad:** cuenta con *Hot Reload* [6], que permite ver los cambios en la aplicación en tiempo real, sin necesidad de reiniciar la aplicación, y sin perder el estado de esta, agilizando mucho el desarrollo.
- **Personalización:** su diseño de la interfaz basada en *wIDGETS* personalizables, que permiten crear interfaces de usuario atractivas y consistentes.
- **Comunidad:** existe una comunidad con buena actividad, la cual hace que prolifere el desarrollo de *plugins* que se pueden encontrar en *pub.dev* [7].

2.1.1. *Dart, el lenguaje detrás de Flutter*

Flutter se basa en el lenguaje de programación *Dart* [5], también desarrollado por *Google*. Aunque este es un lenguaje relativamente nuevo, y con un uso que no está tan extendido como otros lenguajes, como *Java*, *Python* o *JavaScript*, podemos destacar varias características de este lenguaje que lo hacen atractivo. Por otro lado, haberse familiarizado con este lenguaje no ha sido un problema para el desarrollo en este trabajo, debido a una sintaxis que recuerda a otros lenguajes como *Java*, *C++* o *JavaScript*. Algunas características para una visión general de *Dart* [8] son:

- **Tipado estático:** es un lenguaje de tipado estático, lo que significa que una vez las variables se declaran de un tipo, este no puede cambiar durante la ejecución del programa.
- **Tipado dinámico:** aunque sea un lenguaje de tipado estático, también permite el uso del tipo `dynamic`, que permite flexibilidad en el tipo que deba tener una variable.
- **Programación orientada a objetos:** es un lenguaje orientado a objetos, lo que lo hace muy amigable si se tiene experiencia con otros lenguajes orientados a objetos.
- **Seguridad frente a nulos:** tiene un sistema de seguridad frente a nulos, que permite evitar que variables puedan tener un valor nulo en cualquier momento en tiempo de ejecución.
- **Asincronía:** permite el uso de funciones asíncronas, que permiten ejecutar código en segundo plano, sin bloquear la ejecución del programa.
- **Compilación a código nativo:** se compila a código nativo, lo que aporta un rendimiento muy bueno.
- **Compilación nativa:** permite la compilación *JIT (just-in-time)* y *AOT (ahead-of-time)*, que permite producir código máquina para plataformas nativas.
- **Compilación Web:** para aplicaciones *Web*, se compila a código *JavaScript*, que es lo que ocurre con la aplicación desarrollada en este trabajo.

En resumen, Dart es un lenguaje optimizado para el cliente que permite desarrollar aplicaciones rápidas en cualquier plataforma. Su objetivo es ofrecer el lenguaje de programación más productivo para el desarrollo multiplataforma, junto con una plataforma de ejecución flexible para marcos de aplicaciones. Dart también constituye la base de Flutter. Dart proporciona el lenguaje y los tiempos de ejecución que impulsan las aplicaciones de Flutter, pero Dart también admite muchas tareas básicas del desarrollador, como formatear, analizar y probar el código [8].

2.1.2. *Instalación y primer uso básico*

Para poder desarrollar con *Flutter*, es necesario instalar el *SDK* de *Flutter*, que se explica en [9] para diferentes plataformas. Una vez instalado, se puede comprobar que todo funciona correctamente ejecutando el comando `flutter doctor`. Para crear un proyecto, se ejecuta el comando `flutter create <nombre_proyecto>`. Una vez creado, tras situarse en el directorio del nuevo proyecto, se puede ejecutar con el comando `flutter run`. Ejecutando el comando `flutter devices` se puede comprobar qué dispositivos están conectados y disponibles para la ejecución. En la Figura 2.1 se puede ver un ejemplo de ejecución de un proyecto *Flutter*.

**Figura 2.1:** Ejecución de un proyecto *Flutter*.

Imagen extraída de flutter.dev

2.1.3. *Widgets*

Los *widgets* son el elemento básico de la interfaz de usuario en *Flutter*. Todo lo que se ve en pantalla es un *widget*, y estos se combinan para crear elementos gráficos más complejos; pero siempre hay que tener en mente que todo está formado por *widgets*, los cuales se organizan en forma de árbol. En la Figura 2.2 se puede ver un ejemplo de vista de árbol de *widgets*.

**Figura 2.2:** Ejemplo de árbol de *widgets*.

Según [10], los *widgets* de *Flutter* se construyen utilizando un marco moderno que se inspira en *React* [11]. La idea central es construir la interfaz de usuario a partir de *widgets*. Los *widgets* describen el aspecto que debería tener su vista en función de su configuración y estado actuales. Cuando cambia el estado de un *widget*, este reconstruye su descripción, que el framework compara con la descripción anterior para determinar los cambios mínimos necesarios en el árbol de renderizado subyacente para pasar de un estado al siguiente.

Existen dos tipos de *widgets*: los *StatelessWidget* y los *StatefulWidget*. Los primeros son aquellos que no contemplan cambios en su estado porque no tienen; es decir, no almacenan un estado que

pueda variar. Mientras que los segundos sí almacenan un estado que pueda variar, y por ejemplo, pueden cambiar su aspecto en función de este estado.

Para hacer ver esto de una forma más cercana a lo técnico y al código, vamos a aclarar un par de cosas. Cuando se crea un *widget*, se define una clase que hereda de *StatelessWidget* o de *StatefulWidget*, y se implementa el método `build()`. Este método es el que se encarga de devolver un *widget* (que, como ya hemos comentado, puede estar formado por numerosos *widgets* estructurados en un árbol) que será lo que se muestre en la interfaz de usuario.

En el caso de los *StatefulWidget*, debe implementarse el método `createState()`, que devolverá un objeto que hereda de *State*. Este será el que se encargue de almacenar el estado del *widget*, y actualizarlo cuando se necesite. En el caso de un *StatefulWidget*, el método `build()` puede implementarse tanto en el *widget* como en el *state*, pero es recomendable hacerlo en el *state*. Veamos la comparación en código de un *StatelessWidget* y un *StatefulWidget* en la Figura 2.3.

```
1  class Mi StatelessWidget extends StatelessWidget {
2    @override
3    Widget build(BuildContext context) {
4      return Container(
5        // Aquí va el contenido del widget
6      );
7    }
8  }
```

```
1  class Mi StatefulWidget extends StatefulWidget {
2    @override
3    _Mi StatefulWidget createState() => _Mi StatefulWidget();
4  }
5
6  class _Mi StatefulWidget extends State<Mi StatefulWidget> {
7    @override
8    Widget build(BuildContext context) {
9      return Container(
10        // Aquí va el contenido del widget
11      );
12    }
13  }
```

(a) *StatelessWidget*(b) *StatefulWidget*

Figura 2.3: Implementación en código de un *StatelessWidget* y un *StatefulWidget*.

2.2. Spotify y su API Web

La plataforma *Spotify*, creada por Daniel Ek y Martin Lorentzon en Estocolmo durante el año 2006, ha transformado significativamente el panorama del streaming musical desde su lanzamiento en octubre de 2008. Con una oferta que abarca más de 100 millones de temas y 3 millones de vídeos, disponible en más de 184 países, *Spotify* ha logrado posicionarse como uno de los líderes del sector gracias a su modelo de negocio freemium y su constante innovación en la experiencia de usuario. [12]

También en algunos artículos como [13] se destaca que *Spotify* se ha caracterizado por su enfoque en la innovación, introduciendo *podcasts*, vídeos musicales y avanzando en la personalización del contenido con iniciativas como "Descubrimiento Semanal", adaptadas a los gustos individuales de cada usuario. Este compromiso con la evolución constante, junto a una API de desarrollo abierta y amigable, resalta la posición de *Spotify* no solo como líder en el ámbito de la música en streaming, sino también como una plataforma preferente para el desarrollo de aplicaciones musicales innovadoras.

2.2.1. La API Web de Spotify

A diferencia de otras plataformas como *Deezer*, *Apple Music*, *Amazon Music* o *Youtube Music*, que también ofrecen catálogos amplios y funcionalidades de personalización, *Spotify* destaca por la accesibilidad de su API Web para desarrolladores. Esta interfaz permite una diversidad de integraciones y desarrollos de aplicaciones que enriquecen la experiencia del usuario final, ofreciendo acceso a información musical, perfiles y listas de reproducción de forma única.

Aunque competidores como *Deezer* [14] y *Apple Music* [15] proporcionan APIs para desarrolladores, la documentación exhaustiva y la comunidad activa de *Spotify* facilitan de manera muy considerable la integración y el desarrollo, posibilitando el surgimiento de aplicaciones de terceros que amplían las funcionalidades de la plataforma. Por otro lado, aunque plataformas como *Apple Music*, *Amazon Music* o *Youtube Music* se benefician del respaldo de grandes corporaciones tecnológicas y ofrecen integraciones únicas dentro de sus ecosistemas, la apertura y accesibilidad de la API de *Spotify* marca una diferencia significativa, proporcionando a los desarrolladores una herramienta poderosa para la creación de experiencias musicales novedosas. Este enfoque ha consolidado a *Spotify* como una plataforma líder, no solo por su contenido, sino por la flexibilidad y oportunidades que ofrece a los creadores y desarrolladores en el mundo de la música digital.

2.2.2. Contribución de Spotify a la investigación y desarrollo en el ámbito del análisis musical y la recomendación

Spotify se distingue en el ámbito de la música en streaming no solo por su extenso catálogo y funcionalidades de usuario, sino también por su innovador uso de tecnologías de análisis de música, recomendación y aprendizaje automático. La plataforma ha liderado el campo en la personalización de la experiencia musical, empleando algoritmos avanzados para analizar preferencias de escucha y comportamientos de sus usuarios. Esto permite a *Spotify* ofrecer listas de reproducción altamente personalizadas como la ya mencionada "Descubrimiento Semanal", que se ha convertido en una característica emblemática de la plataforma.

El compromiso de *Spotify* con la investigación y el desarrollo se puede ver reflejado, entre otros, en su portal de I+D e ingeniería [16], donde se publican artículos sobre temas de los proyectos de ingeniería actuales de la empresa, y también se exponen multitud de proyectos de código abierto que han sido impulsados por los ingenieros de *Spotify*. Este enfoque en la investigación y el desarrollo en el ámbito del análisis musical y la recomendación ha colocado a *Spotify* en la vanguardia de la innovación en la industria musical. Al invertir en estas tecnologías, *Spotify* no solo mejora la experiencia del usuario en su plataforma, sino que también contribuye significativamente al campo del conocimiento musical, abriendo nuevas vías para el descubrimiento y la apreciación de la música en todo el mundo.

2.3. Recomendación

Como introducción a esta sección, un fragmento del prefacio del libro *Recommender Systems Handbook* [17] traducido por nosotros al español aporta una visión bastante actual de qué debe venirnos a la cabeza cuando hablamos de sistemas de recomendación:

“Los sistemas de recomendación son herramientas y técnicas de software que ofrecen sugerencias de artículos útiles para un usuario. Las sugerencias de un sistema de recomendación están pensadas para ayudar al usuario a tomar decisiones como qué comprar, qué música escuchar o qué noticias leer. Los sistemas de recomendación son un valioso medio para que los usuarios de Internet hagan frente a la sobrecarga de información y les ayuden a elegir mejor. En la actualidad son una de las aplicaciones más populares de la inteligencia artificial, ya que ayudan a descubrir información en la Red. Se han propuesto varias técnicas de generación de recomendaciones y, en las dos últimas décadas, muchas de ellas se han implantado con éxito en entornos comerciales. Hoy en día, todos los grandes actores de Internet adoptan técnicas de recomendación. El desarrollo de sistemas de recomendación es un esfuerzo multidisciplinar en el que participan expertos de diversos campos, como la inteligencia artificial, la interacción persona-ordenador, la minería de datos, la estadística, los sistemas de apoyo a la decisión, el marketing y el comportamiento del consumidor.”

El estudio sobre sistemas de recomendación es un campo bastante más nuevo que otras técnicas y herramientas clásicas de los sistemas de información, como las bases de datos, o los motores de búsqueda. A pesar de ser un área de estudio independiente relativamente joven, el interés en estos sistemas ha crecido de manera muy significativa. Un claro signo de ello es cómo se hace uso de los sistemas de recomendación en la actualidad, como por ejemplo en las plataformas de *streaming* de música o vídeo (como Spotify o Netflix), o en las plataformas de comercio electrónico (como Amazon o eBay). Reflexionemos sobre el motivo de su extendido uso.

Partes interesadas en un sistema de recomendación

Según [18], en un sistema de recomendación influyen tres partes interesadas: los usuarios (o consumidores), los proveedores (o suministradores) y los propietarios del sistema. Los usuarios son los que reciben las recomendaciones (basándonos en este trabajo, digamos que los usuarios de Spotify), los proveedores son los que ofrecen los productos o servicios que se recomiendan (digamos que los artistas que publican su música en Spotify), y el propietario del sistema (en este caso, Spotify) que ofrece la plataforma en la que los usuarios son expuestos a los proveedores (artistas). Sabiendo esto, podemos ver que las tres partes se ven beneficiadas ante un escenario ideal: los usuarios reciben recomendaciones de productos o servicios que les interesan (descubren nuevas canciones, géneros y artistas que les gustan), los proveedores ven aumentada su visibilidad (los artistas pueden volverse más conocidos) y, por tanto, sus ventas, y el propietario del sistema ve aumentado su número de usuarios y, por tanto, sus ingresos (más suscripciones de Spotify). Este beneficio claro para las tres

partes diríamos que es uno de los mayores motivos por los que los sistemas de recomendación son tan populares y utilizados en la actualidad.

Fuentes de datos y conocimiento en un sistema de recomendación

Por otro lado, no podemos dejar de lado la suma importancia que tienen los datos de los que hace uso un sistema de recomendación. En [18], se destacan tres elementos como las fuentes de conocimiento y datos de las que un sistema de recomendación se nutre: los usuarios, los productos (o ítems) y las interacciones entre los dos anteriores. Los **ítems**, que son los objetos de las recomendaciones, deben contar con una descripción detallada y exhaustiva, abarcando desde características generales hasta metadatos específicos. Algoritmos especializados extraen y analizan estas características, los cuales son especialmente necesarios y útiles para ítems complejos como imágenes o textos, utilizando técnicas avanzadas de procesamiento de imágenes y lenguaje natural. En cuanto a los **usuarios**, cada usuario se modela de manera única, integrando una diversidad de factores como intereses, edad y género, entre otros. Pero más allá de las preferencias que se puedan mostrar explícitamente (como que un usuario indique sus géneros de música favoritos de manera activa), el modelo de un usuario también se enriquece con información contextual, permitiendo así que el sistema de recomendación ajuste sus sugerencias a las circunstancias específicas y momentáneas del usuario (como un descubrimiento de un nuevo artista o género que está escuchando recientemente). Por último, las **interacciones** entre usuarios e ítems son la fuente de datos más importante para un sistema de recomendación. Desde calificaciones explícitas hasta comportamientos implícitos de navegación, cada acción del usuario aporta información valiosa, contribuyendo a la comprensión profunda de sus preferencias y comportamientos.

2.3.1. Métodos y técnicas de recomendación

Para que un sistema de recomendación haga bien su trabajo, que es sugerir ítems que puedan interesar al usuario, tiene que ser capaz de adivinar o predecir cómo de útiles o atractivos serán ciertos ítems para cada persona. Esto se hace evaluando y comparando lo que se conoce sobre los ítems y los usuarios. A veces, por ejemplo, cuando no hay mucha información específica sobre los gustos de un usuario, el sistema puede optar por recomendar lo que es popular o lo que ha gustado a muchos, pensando que hay más posibilidades de que también le guste a ese usuario.

En términos técnicos, se habla de calcular la 'utilidad' (también lo llamaremos *rating* en este trabajo) de un ítem para un usuario, que no es más que tratar de prever cuánto le va a gustar. No todos los sistemas hacen este cálculo de manera completa y obtienen un *rating* como tal; algunos se basan en métodos más generales o en suposiciones para decidir qué recomendar. Además, hay que considerar que lo que a una persona le puede parecer atractivo o interesante puede depender de muchas cosas, como la actividad que puede estar haciendo, su compañía (si la tiene) o el lugar, así que las recomendaciones también deberían tener en cuenta esos detalles idealmente, para que fueran realmente acertadas.

Por supuesto, existen diversas formas en las cuales un sistema de recomendación puede calcular estas predicciones de utilidad o atractivo para los usuarios. Trataremos tres tipos principales: basado en contenido, el filtrado colaborativo y los sistemas híbridos.

Basado en contenido

La recomendación basada en contenido se centra en las características de los ítems para hacer recomendar a los usuarios. Funciona analizando los detalles de los ítems que a un usuario le han gustado en el pasado y buscando otros ítems que sean similares en cuanto a esas características. Por ejemplo, si a alguien le gustan las películas de un género en particular o de un director específico, el sistema recomendará otras películas que coincidan con esas preferencias. De manera muy sencilla, la recomendación basada en contenido mira lo que te ha gustado antes y te sugiere cosas parecidas. Imaginemos que nos encantan ciertas canciones de pop español o las canciones de un artista en concreto. El sistema observa esto y luego te muestra más películas del mismo tipo.

Este método utiliza algoritmos para procesar y entender las características de los ítems, como las palabras en una descripción, los elementos visuales en una imagen, o el análisis de audio de una canción. Con esta información, el sistema crea un perfil de lo que le gusta al usuario basado en lo que ha elegido anteriormente y luego busca y recomienda nuevos ítems que encajen con ese perfil. Así, la recomendación basada en contenido permite que las sugerencias sean personalizadas y relevantes para los intereses específicos de cada usuario.

Filtrado colaborativo

El filtrado colaborativo es el enfoque original y más simple de los sistemas de recomendación. Predice la utilidad de un ítem para un usuario basándose en cómo de atractivo o útil les ha parecido ese ítem a otros usuarios. El fundamento detrás de esto es que, si dos usuarios han tenido intereses parecidos en el pasado, desde luego es probable que tengan intereses parecidos en el futuro. Imaginemos que dos usuarios han escuchado las mismas canciones de un artista en el pasado, o de un género en particular. Si uno empieza a escuchar una nueva canción de manera recurrente (le gusta), es probable que al otro usuario también le vaya a gustar esta canción. Podemos imaginarlo como si tenemos un amigo con el que compartimos gustos musicales; lógicamente, tendremos en cuenta sus recomendaciones, ya que sabemos que tenemos gustos parecidos.

Debe destacarse que el filtrado colaborativo no necesita información detallada sobre los ítems, como la recomendación basada en contenido. Existen dos tipos de filtrado colaborativo: basado en usuarios y basado en ítems. El basado en usuarios compara a los usuarios entre sí y recomienda ítems que les han parecido atractivos a otros usuarios identificados como similares al usuario al que se le recomienda; mientras que, el basado en ítems compara ítems entre sí. Debemos dejar clara la diferencia entre el filtrado colaborativo basado en ítems, y la recomendación basada en contenido.

En la recomendación basada en contenido (que hemos comentado en la Sección 2.3.1), se modelan con procesos y algoritmos especiales las características de los ítems que consume un usuario para, tras ello, recomendarle más ítems que se asemejen a las características de ese modelo producido. En cambio, en el filtrado colaborativo enfocado en ítems, la diferencia fundamental se encuentra en cuándo se considera a dos ítems similares entre sí (obviando la diferencia de que aquí se comparan ítems directamente, y en la basada en contenido se comparan ítems con un perfil o modelo de características extraído de haber procesado los ítems consumidos previamente). Y es que, si un conjunto de usuarios ha calificado o interactuado de manera similar con ciertos ítems, entonces esos ítems se consideran similares entre sí. No se necesitan detalles sobre las características de los ítems, como género de una canción, artista o análisis de audio. La similitud entre los ítems se calcula en función de las calificaciones o interacciones de los usuarios, no en función de sus características intrínsecas. Aún así, estas características intrínsecas podrían tenerse en cuenta para, posiblemente, mejorar la calidad de las recomendaciones, tratando de coger lo mejor de los dos tipos de sistemas. Veamos a continuación los Sistemas Híbridos.

Sistemas híbridos

Estos sistemas nacen de la necesidad de cubrir los inconvenientes con los que nos podemos encontrar con los anteriores métodos. No entraremos en más detalles, pero por ejemplo, mientras que los métodos de filtrado colaborativo tienen dificultades para recomendar nuevos ítems sin calificaciones, los enfoques basados en contenido pueden hacer predicciones para estos ítems usando sus descripciones.

Además, es importante considerar el contexto del usuario para personalizar más las recomendaciones. Por ejemplo, las sugerencias para vacaciones en invierno deberían ser distintas a las de verano, o las recomendaciones de canciones deberían ser diferentes para una sesión de ejercicio, de estudio, o una fiesta. Los sistemas de recomendación sensibles al contexto (CARS) [19] utilizan esta información para afinar sus sugerencias.

2.3.2. Tipos de sistemas de recomendación

Tras haber visto como base los métodos más comunes de recomendación, vamos a pasar a hablar de los dos ámbitos o tipos de recomendación que más nos interesan en este trabajo por razones obvias si queremos desarrollar una aplicación como la descrita en la introducción: la recomendación de música y la recomendación a grupos.

Recomendación de música

Aunque en [20] se habla de manera bastante extensa sobre la recomendación en este ámbito, trataremos de hacer un resumen con los puntos que más creemos que nos interesan para este trabajo. Está claro que el consumo de música ha cambiado mucho en los últimos años, llegando a la situación actual, donde la música se consume de manera digital, generalmente en *streaming*. Esto ha llevado a que estas plataformas que dan este tipo de servicio lleguen a ofrecer unas cantidades de música (o contenido, si incluimos los *podcasts*) tan inmensas y tan variadas que son inabarcables para nadie. Y aquí es donde entra en juego la recomendación de música, que se convierte en una herramienta muy útil para los usuarios, al ayudarles a navegar por el inmenso catálogo de música que ofrecen estas plataformas.

Por otro lado, como se expone en [20], hay unos cuantos factores que hacen que los sistemas de recomendación de música se diferencien considerablemente de otros tipos de sistemas de recomendación:

- **Duración de Consumo:** La música se consume más rápidamente que otros medios (como películas o libros), lo que significa que los usuarios pueden formar opiniones sobre los ítems musicales en menos tiempo. Esto puede hacer que los ítems musicales se consideren más 'desechables'.
- **Tamaño del Catálogo:** Los catálogos de música son considerablemente más grandes que los de películas o series. Por lo tanto, la escalabilidad de los algoritmos de recomendación es una consideración más crítica en el dominio de la música.
- **Consumo Repetido:** Los ítems musicales a menudo se consumen repetidamente, incluso varias veces seguidas, a diferencia de otros medios como películas o libros.
- **Consumo Secuencial:** Las canciones suelen consumirse en secuencia (como en *playlists*), lo que da importancia a problemas de recomendación que tienen en cuenta la secuencia, como la continuación automática de listas de reproducción o la recomendación de la próxima canción.
- **Consumo Pasivo:** La música a menudo se consume de manera pasiva, lo que puede afectar la calidad de los indicios de preferencia, especialmente obteniéndolos de manera implícita (como el tiempo de escucha).
- **Importancia del Contenido:** En la recomendación musical, los enfoques basados en contenido son más relevantes que en otros dominios debido a la influencia de campos como el procesamiento y análisis de audio, y la recuperación de información musical. Estos enfoques intentan extraer información semántica de la música y usarla para hacer recomendaciones, a menudo debido a la escasez de datos de calificación explícitos en este dominio.

Recomendación a grupos

También se habla de forma extensa sobre este ámbito de los sistemas de recomendación en [21], pero trataremos de destacar los puntos que más nos pueden interesar para el trabajo y el desarrollo de nuestra idea de aplicación presentada en la introducción. A continuación, exponemos una serie de características que permiten clasificar los sistemas de recomendación a grupos:

- **Conocimiento de Preferencias:** Las preferencias individuales pueden ser conocidas de antemano o desarrollarse con el tiempo a través de interacciones de grupo u obtención de valoraciones.

- **Experiencia de los ítems:** Los ítems pueden ser experimentados directamente por el grupo (como música o anuncios) o presentados como opciones en una lista de recomendaciones.
- **Interactividad del Grupo:** El grupo puede ser pasivo, sin influir en la agregación de preferencias, o activo, participando en la negociación o influenciando las calificaciones.
- **Negociación vs. Recomendaciones Directas:** Algunos sistemas permiten negociación y discusión sobre las recomendaciones, mientras que otros entregan recomendaciones de forma directa sin interacción.
- **Recomendación de Ítem Único vs. Secuencia:** Algunos sistemas recomiendan un único ítem, mientras que otros sugieren secuencias de ítems, como listas de reproducción o rutas turísticas.
- **Estimación de Preferencias:** Las preferencias individuales pueden ser estimadas usando filtrado basado en contenido o colaborativo.
- **Agregación de Perfiles o Recomendaciones:** Los sistemas pueden agregar perfiles de usuarios, que expresan preferencias numéricas por ítem, o directamente las recomendaciones, ya sea como listas clasificadas o como estimaciones numéricas.

Un aspecto importante en la recomendación dirigida a grupos es la agregación de preferencias, y en nuestro caso la agregación de las recomendaciones individuales, en una sola para todo el grupo. Este tema lo trataremos en la siguiente sección, ya que, es el factor sobre el que han recaído algunas pruebas, investigación y experimentos con usuarios a la hora de probar distintas generaciones de *playlists*.

2.3.3. Estrategias de agregación

En la recomendación dirigida a grupos, el principal reto a resolver es cómo conseguir tratar a un grupo de usuarios como una única entidad, basándonos en los gustos individuales de cada uno de los miembros de este. En [21] se habla de varias estrategias de agregación de preferencias, estas sirven de la misma forma para agregar perfiles, y agregar recomendaciones (lo que haremos en nuestro sistema de recomendación). Veamos algunas de las estrategias de agregación más comunes, las nombraremos en inglés, para minimizar confusiones entre la bibliografía, esta sección y el desarrollo:

- **Average:** Promedia las calificaciones de los usuarios.
- **Multiplicative:** Multiplica las calificaciones de los usuarios.
- **Borda:** Asigna puntos a cada ítem en función de su clasificación por cada usuario, y luego suma los puntos para cada ítem.
- **Least Misery:** Asigna a cada ítem el valor mínimo de las calificaciones de los usuarios.
- **Most Pleasure:** Asigna a cada ítem el valor máximo de las calificaciones de los usuarios.
- **Average without Misery:** Asigna a cada ítem el valor promedio de las calificaciones de los usuarios, pero excluye las calificaciones más bajas (se establece un umbral para decidir cuándo es demasiado baja una puntuación o *rating*).
- **Fairness:** Los ítems se ordenan como si cada usuario eligiera un ítem por turnos. Así, si hay tres usuarios, el los primeros tres ítems de la lista serán los que más gusten a cada usuario, y así sucesivamente.

2.3.4. Evaluación de un sistema de recomendación a grupos

Cómo evaluar un sistema de recomendación sigue siendo un tema de investigación muy importante a día de hoy, ya que, a pesar de que existen métricas o pruebas con usuarios, no hay un método de evaluación que sea el más adecuado para todos los sistemas de recomendación. En [21] se habla sobre la evaluación de sistemas de recomendación dirigidos a grupos, y se destacan dos tipos: estudios con usuarios, y evaluaciones *offline*.

Estudios con usuarios

Se lleva a cabo una investigación con usuarios cuando queremos evaluar cómo rinde un sistema basándonos en aspectos como lo que fácil es de usar o cuánto les gusta a los usuarios (por ejemplo, si están satisfechos con él o cómo de buenas son las recomendaciones que reciben). Para recoger esta información, a veces hablamos directamente con las personas involucradas o utilizamos plataformas online que permiten la participación masiva. Aunque estos estudios son esenciales para entender si la gente realmente valora y acepta el sistema, no debemos depender únicamente de ellos para medir la efectividad del sistema, ya que su alcance es limitado. Sin embargo, entender la percepción y la experiencia del usuario es fundamental para confirmar si nuestra solución realmente cumple con las necesidades de quienes la utilizan.

Evaluaciones *offline*

En la investigación sobre sistemas de recomendación para grupos, al igual que en los individuales, se emplean evaluaciones *offline*. No obstante, la falta de datos reales sobre las preferencias de grupos complica estos estudios. Una solución común es el uso de grupos sintéticos, creados artificialmente a partir de datos estándar. El problema principal es cómo validar efectivamente las recomendaciones para estos grupos sintéticos, dado que las preferencias reales del grupo son desconocidas.

A veces, se evalúan las recomendaciones de grupo comparándolas con una valoración media de las preferencias individuales, pero esta aproximación no refleja el complejo funcionamiento de un grupo real. Otros métodos comparan las recomendaciones con las preferencias individuales, asumiendo que la satisfacción de cada miembro del grupo depende exclusivamente de su propia preferencia, ignorando la influencia de los demás miembros del grupo. A pesar de esto, las investigaciones suelen favorecer estrategias que se asemejan a la media.

No obstante, existen estudios, como en [22] o [23] que se apartan de la creación de grupos sintéticos, recopilando en cambio datos reales de individuos y de sus elecciones cuando están en grupos. Estos enfoques ofrecen una visión más precisa y realista de las dinámicas de grupo en los sistemas de recomendación, pero son más difíciles de conseguir.

2.4. Implicaciones para el trabajo

Teniendo en cuenta el contexto de la aplicación que queremos desarrollar, y las opciones que nos da la API de *Spotify*, podríamos tratar de clasificar en base a las características expuestas en la Sección 2.3.2 el sistema de recomendación que podemos obtener. Por ello, en cuanto al que va a ser nuestro sistema de recomendación, podemos decir que:

- Conoceremos las preferencias individuales de los usuarios de antemano gracias a los datos de *Spotify*.
- Los ítems serán experimentados directamente por el grupo, ya que se trata de música.
- Buscamos que el grupo pueda ser activo, pudiendo elegir distintas estrategias de agregación de preferencias.
- Nuestro sistema suministrará recomendaciones directamente tras una previa 'negociación'.
- Nuestro sistema recomendará secuencias de ítems, las *playlists*.
- Las preferencias individuales de los usuarios son estimadas de forma implícita, a través de sus interacciones con *Spotify*.
- En el caso de nuestro sistema de recomendación, agregaremos recomendaciones individuales de ítems en una recomendación grupal. No agruparemos perfiles de usuarios en un único perfil grupal.

En cuanto a las estrategias expuestas en la Sección 2.3.3, la utilización de ellas será fundamental. Tal y como las utilizaremos será para obtener un *rating* grupal para cada canción que sea susceptible de entrar en la *playlist* que se acabará generando para el grupo. Así que, lo que haremos básicamente, será obtener recomendaciones de individuales de canciones para cada usuario, y luego, a través de una estrategia de agregación, obtener un *rating* grupal para cada canción. Ahondaremos más sobre el funcionamiento e implementación en la Sección 3.3.

DESARROLLO DE NUESTRA APLICACIÓN

Este capítulo haremos un recorrido por el proceso de desarrollo de la aplicación, pasando por unas primeras aproximaciones de análisis de requisitos y diseño de interfaz y arquitectura, para acabar con los detalles de implementación de los actores fundamentales en la aplicación.

3.1. Análisis

En esta sección, haremos una aproximación al análisis de requisitos de la aplicación que vamos a desarrollar, definiendo también los casos de uso fundamentales de la aplicación.

3.1.1. Análisis de requisitos

Requisitos Funcionales

Expondremos los requisitos funcionales de la aplicación, definiendo lo que esperamos que nuestra aplicación desarrollada cumpla. Estos requisitos se han obtenido a partir de la motivación del proyecto y de las funcionalidades que ofrece la API de Spotify. A continuación, se detallan los requisitos funcionales de la aplicación:

RF-1.- El sistema debe permitir que los usuarios hagan *login* a través de Spotify.

RF-1.1.- El sistema hará uso del servicio de autenticación de Spotify.

RF-2.- El sistema debe permitir que los usuarios obtengan información de sus cuentas de Spotify.

RF-3.- El sistema debe permitir que múltiples usuarios estén dados de alta (hayan hecho *login* con su cuenta de Spotify) al mismo tiempo en la aplicación.

RF-4.- El sistema debe ser capaz de comunicarse con la API de Spotify, y recibir y enviar información con éxito al servicio.

RF-5.- El sistema debe ser capaz de obtener los artistas o canciones más escuchados de los usuarios en el corto, medio o largo plazo.

RF-5.1.- El sistema obtendrá hasta 50 artistas o canciones más escuchados de los usuarios.

RF-6.- El sistema debe ser capaz de ofrecer recomendaciones de canciones a los usuarios basándose en cancio-

nes, géneros o artistas que seleccionen los usuarios.

RF-6.1.- El sistema debe permitir a los usuarios seleccionar canciones, géneros o artistas para obtener recomendaciones.

RF-6.2.- El sistema recomendará hasta 50 canciones a los usuarios.

RF-7.- El sistema debe permitir que múltiples usuarios dados de alta en la aplicación creen una playlist colaborativa.

RF-7.1.- El sistema debe permitir a los usuarios elegir la duración de la playlist colaborativa.

RF-7.2.- El sistema debe ofrecer distintas estrategias de agregación para la selección de canciones en la playlist colaborativa.

RF-7.3.- El sistema debe mostrar la playlist generada en pantalla.

RF-7.4.- El sistema debe permitir a los usuarios guardar la playlist colaborativa en sus cuentas de Spotify.

Requisitos No Funcionales

A continuación, se detallan los requisitos no funcionales de la aplicación:

RNF-1.- Rendimiento: El sistema debe responder a las solicitudes del usuario en menos de 3 segundos bajo condiciones normales de carga.

RNF-2.- Disponibilidad: El sistema debe estar disponible el 99 % del tiempo, asegurando el acceso de los usuarios a sus funcionalidades.

RNF-3.- Compatibilidad: El sistema debe ser compatible con las versiones más recientes de navegadores web populares como Chrome, Firefox, Safari y Edge.

RNF-4.- Mantenibilidad: El código del sistema debe seguir las buenas prácticas de programación, siendo legible, bien documentado y fácil de mantener.

RNF-5.- Usabilidad: La interfaz de usuario debe ser intuitiva y fácil de usar, asegurando que los usuarios puedan realizar todas las operaciones sin necesidad de instrucciones adicionales.

RNF-6.- Almacenamiento: El sistema debe almacenar la información de los usuarios de manera local, no existirá una base de datos centralizada.

3.1.2. Diagramas de casos de uso principales

Se presentan en la Figura 3.1 los diagramas de casos de uso principales de la aplicación. Estos diagramas representan las interacciones en las cuales se sustenta la funcionalidad de la aplicación. En primer lugar se muestra el caso de uso de *Login*, donde mostramos los pasos para que el usuario termine viendo en su pantalla la información de su cuenta de *Spotify*. En segundo lugar, se muestra el caso de uso de *Crear Playlist*, donde se detallan los pasos que lleva a cabo el sistema para que los usuarios puedan crear una playlist colaborativa.

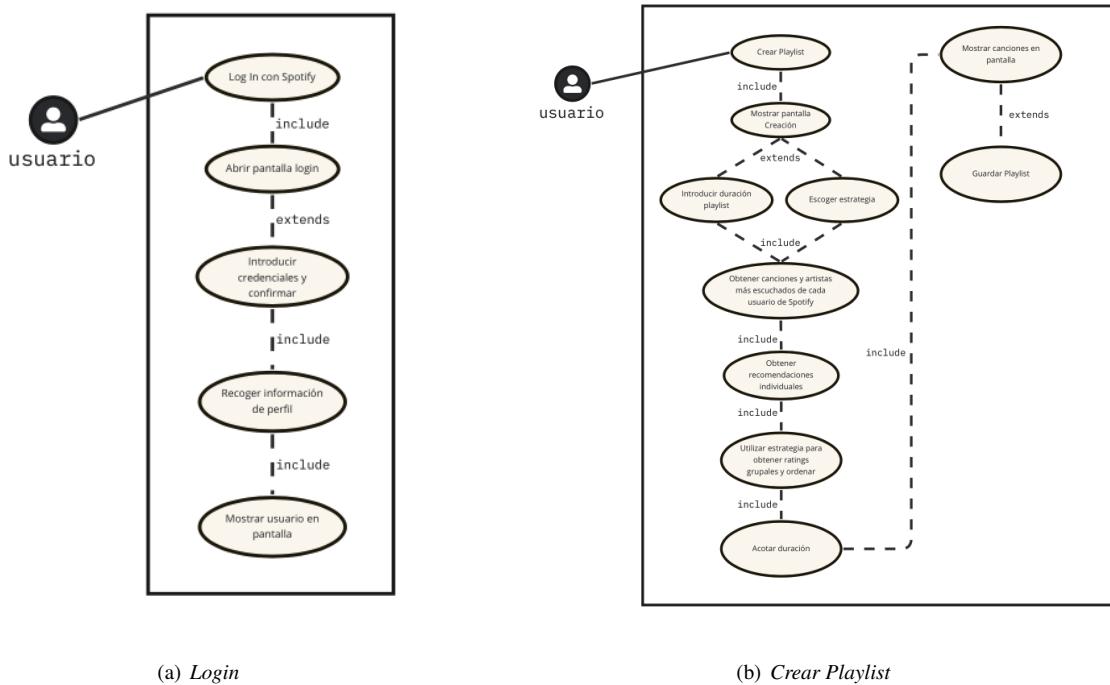


Figura 3.1: Diagramas de casos de uso principales de la aplicación

3.2. Diseño de la aplicación

En esta sección se expondrá el diseño de la interfaz de usuario, así como el de la arquitectura.

3.2.1. Diseño de la interfaz de usuario

En la Figura 3.2 se pueden ver las maquetas de la interfaz de usuario de la aplicación que hemos tenido en mente a la hora de llevar a cabo la implementación de nuestra aplicación. En estas maquetas se exponen las pantallas con las que el usuario podrá interactuar, contando así con un diseño previo de interfaz de la aplicación que hemos tratado de seguir durante el desarrollo, siempre teniendo en cuenta que estas maquetas eran una guía, y que podían sufrir modificaciones durante el desarrollo.

Tras el desarrollo de la aplicación, hemos conseguido un resultado muy satisfactorio respecto al aspecto de la interfaz de usuario, que se asemeja mucho a las maquetas que se han presentado en la Figura 3.2. Véase en la Figura 3.3 capturas de pantalla de las pantallas de la aplicación desarrollada.

3.2.2. Arquitectura del sistema

La arquitectura de la aplicación se basa en una arquitectura cliente-servidor, donde el cliente es el *frontend* que hemos desarrollado, que corre en el navegador web, y el servidor es la API de *Spotify*. La aplicación se ejecuta en el lado del cliente (en el navegador) y utiliza opciones de almacenamiento

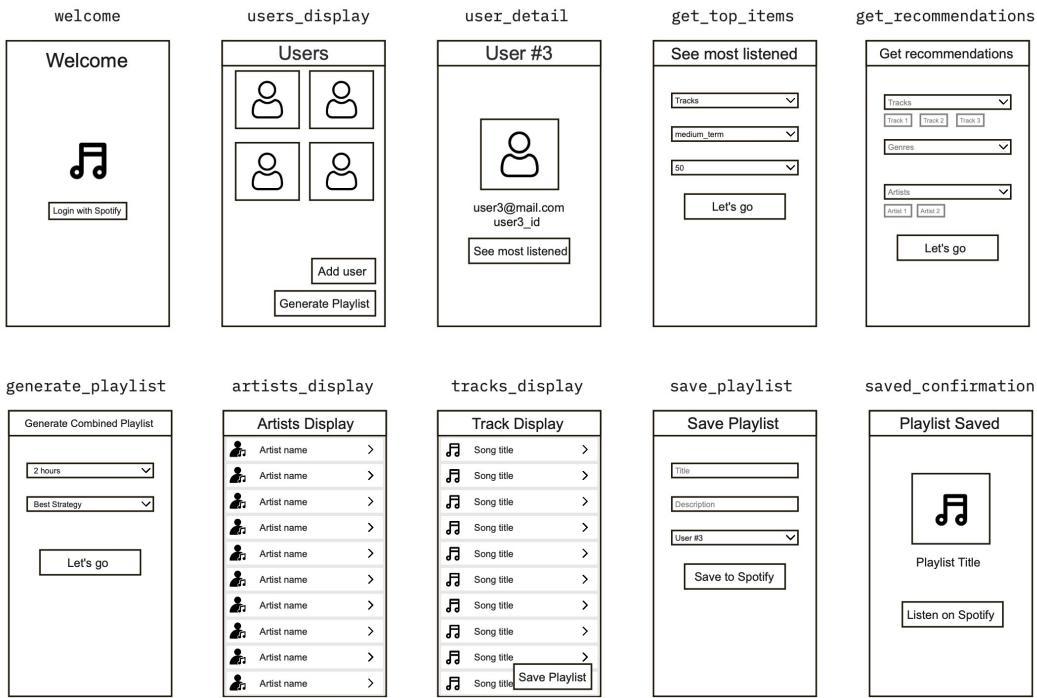


Figura 3.2: Maquetas de la aplicación a desarrollar

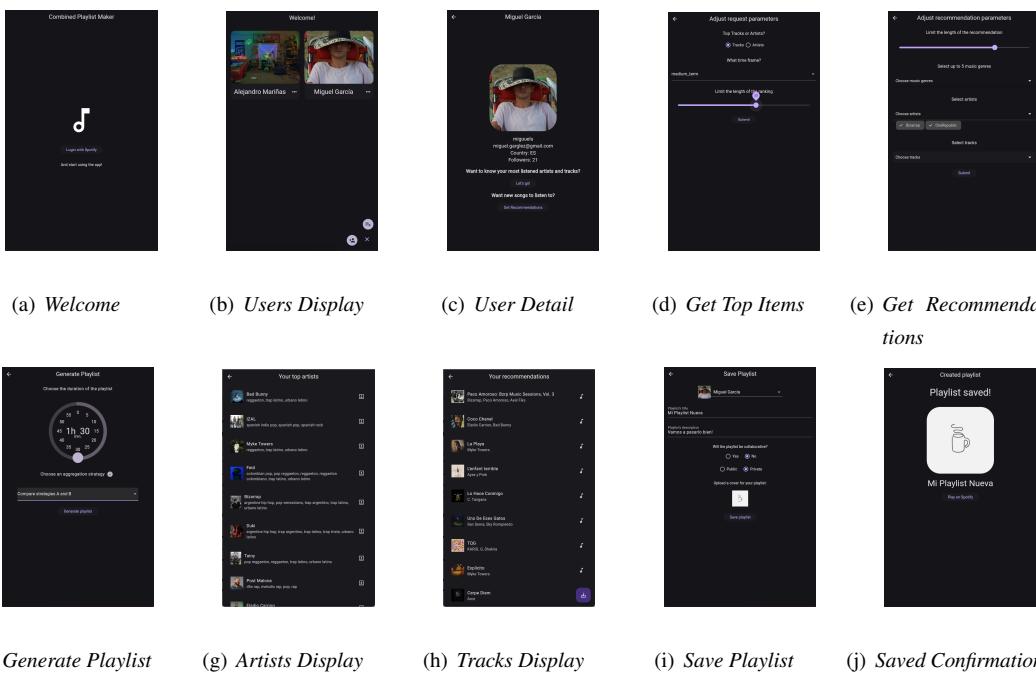


Figura 3.3: Pantallas de la aplicación desarrollada

locales para mantener la seguridad y la integridad de los tokens de acceso a la API de Spotify.

La arquitectura del sistema se compone de los siguientes elementos clave:

- **Frontend:** Interfaz de usuario desarrollada con *Flutter* que ofrece una experiencia de usuario interactiva y dinámica. Se encarga de presentar la información, recoger las interacciones del usuario y comunicarse con la API de Spotify.
- **Almacenamiento Local:** Utilizado para guardar de forma segura los tokens de acceso necesarios para la interacción con la API de Spotify y otros datos necesarios para el funcionamiento de la aplicación, asegurando que la información sensible se maneje correctamente. Utilizaremos la API de *Hive* [24] para su implementación.
- **API de Spotify:** Servicio externo que proporciona acceso a los datos de música, artistas, y *playlists*, así como las funcionalidades para crear y modificar playlists en las cuentas de los usuarios.

La interacción entre los componentes se puede observar en la Figura 3.4 en un diagrama de secuencia y se puede resumir en los siguientes pasos:

- 1.– El **Frontend** envía solicitudes a la API de Spotify para obtener datos de música o realizar operaciones relacionadas con playlists.
- 2.– La **API de Spotify** responde a estas solicitudes, proporcionando los datos o confirmando la realización de las operaciones solicitadas.
- 3.– Durante este proceso, el **Almacenamiento Local** se utiliza para guardar y recuperar los tokens de acceso y otros datos relevantes, asegurando que las solicitudes a la API de Spotify estén autenticadas y sean seguras.

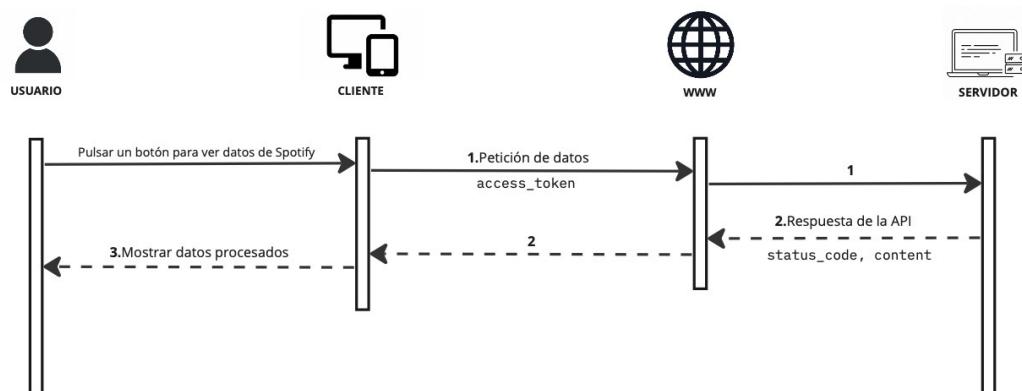


Figura 3.4: Diagrama de secuencia de interacción con la API de Spotify

Frontend

El framework de desarrollo de aplicaciones *Flutter* [1] ha sido el elegido para el desarrollo del *frontend* de la aplicación. Ya hemos comentado en la Sección 2.1 que *Flutter* es un framework basado en el lenguaje de programación *Dart*, cuya base se sienta en los *widgets*. Vamos a explicar cada uno de los actores principales en el desarrollo del *frontend* de la aplicación a continuación.

Pantallas

Tal y como se ha diseñado el *frontend* de la aplicación, cada pantalla de la aplicación se ha desarrollado como un *widget* independiente, el cual, a su vez está formado por widgets más pequeños que se combinan para formar la interfaz de usuario. Los módulos que definen las pantallas son:

- `landing.dart`: pantalla de inicio de la aplicación, que muestra un botón para hacer *login* con *Spotify*.
- `users_display.dart`: pantalla que muestra los usuarios que han hecho *login* en la aplicación.
- `user_detail.dart`: pantalla que muestra la información de un usuario.
- `generate_playlist.dart`: pantalla del formulario para la generación de *playlists*.
- `playlist_display.dart`: pantalla que muestra una *playlist* generada.
- `item_display.dart`: pantalla que muestra ítems (artistas o canciones) en una lista.
- `save_playlist.dart`: pantalla del formulario para guardar una *playlist* en la cuenta de *Spotify* de un usuario.
- `playlist_detail.dart`: pantalla que muestra la información básica de una *playlist* guardada en la cuenta de *Spotify* del usuario.
- `track_detail.dart`: pantalla que muestra la información de una canción.
- `get_top_items.dart`: pantalla del formulario para obtener las canciones más escuchadas.
- `get_recommendations.dart`: pantalla del formulario para obtener recomendaciones de canciones.

Se puede encontrar relación directa entre estos módulos de pantallas y las maquetas de la Figura 3.2.

Widgets

Los *widgets* personalizados (la mayoría de *widgets* utilizados en el desarrollo han sido propios del framework o paquetes de `pub.dev` [7]) que se utilizan para formar las pantallas:

- `artist_tile.dart`: *widget* que muestra la información de un artista en un elemento de lista.
- `track_tile.dart`: *widget* que muestra la información de una canción en un elemento de lista.
- `expandable_fab.dart`: *widget* que muestra un botón flotante que se expande al hacer clicar en él.

Servicios

Por otro lado, agrupados en los 'servicios', se encuentran los módulos que se encargan de realizar las llamadas a la API de *Spotify* y procesar las respuestas para uniformizarlas y hacerlas más fáciles de manejar en el *frontend*. En la Sección 3.3 se ahondará más en el uso de estos módulos. También en este mismo grupo se encuentra el módulo de recomendación de canciones, que se ha desarrollado para llevar a cabo la agregación de las canciones recomendadas por *Spotify* para los usuarios que forman un grupo, así como un módulo de recogida de métricas sobre las recomendaciones de *playlists*:

- `requests.dart`: módulo que se encarga de realizar las llamadas a la API de *Spotify*.
- `recommendator.dart`: módulo que se encarga de llevar a cabo la agregación de las canciones recomendadas por *Spotify* para los usuarios que forman un grupo.

- `statistics.dart`: módulo que se encarga de la realización de pruebas y obtener métricas sobre las recomendaciones de *playlists*.

Modelos

Cuando hablamos de uniformizar las respuestas de la API de *Spotify*, nos referimos a que todas ellas se reducirán a un modelo de objeto determinado, para así uniformizar el manejo de errores en las pantallas y las peticiones exitosas. Se han agrupado los modelos de objetos que se utilizan en la aplicación:

- `artist.dart`: modelo de objeto que representa a un artista.
- `my_response.dart`: modelo de objeto que representa una respuesta de la API de *Spotify*.
- `track.dart`: modelo de objeto que representa a una canción.
- `user.dart`: modelo de objeto que representa a un usuario.

Almacenamiento Local

El almacenamiento local se ha llevado a cabo con *Hive* [24] para *Flutter*. Ha sido utilizada para guardar de forma segura a los usuarios que hagan login en la aplicación, junto con sus respectivos tokens de acceso a la API de *Spotify*. Este ha sido el principal uso que se le ha dado, pero también se ha utilizado para guardar pequeñas variables de forma temporal. Algunas características que han hecho de *Hive* una elección adecuada para el almacenamiento local en la aplicación son:

- **Base de datos No SQL**: *Hive* ofrece una estructura de datos flexible y escalable, ideal para almacenar datos temporales que no requieren un esquema rígido.
- **Fácil de usar**: La API de *Hive* es intuitiva y sencilla, lo que facilita su integración en el código *Flutter*.
- **Alto rendimiento**: *Hive* ofrece un acceso rápido a los datos, lo que es crucial para una experiencia de usuario fluida.
- **Ligera**: La biblioteca de *Hive* tiene un tamaño reducido, lo que minimiza el impacto en el rendimiento general de la aplicación.
- **Multiplataforma**: *Hive* es compatible con *Flutter* web y móvil, lo que permite una mayor reutilización del código.

API de Spotify

En esta sección realizaremos una visión general de la API de *Spotify* [3], e iremos pasando por los *endpoints* más destacados, que serán útiles para el desarrollo de la aplicación. No obstante, en la Sección 3.3 se ahondará más en el uso de cada uno de ellos.

Visión general de la API

La API de *Spotify* es una API *REST*, que utiliza el formato *JSON* para el intercambio de datos. Es importante destacar que para hacer uso de esta API, es necesario darse de alta como desarrollador,

asumiendo que se tiene una cuenta de *Spotify*. Una vez hecho esto, se podrán dar de alta aplicaciones en la cuenta de desarrollador desde el *Dashboard*, y tener así los datos y credenciales necesarios para hacer uso de la API. En el *Dashboard* se pueden ver las aplicaciones dadas de alta, así como crear nuevas. Además, se puede entrar en cada aplicación para ver datos de utilización de *endpoints* y editar ajustes sobre ella, como la dirección del sitio web, las posibles *redirect URLs* para el flujo de autenticación, etc. Obsérvese la Figura 3.5 a modo de orientación.

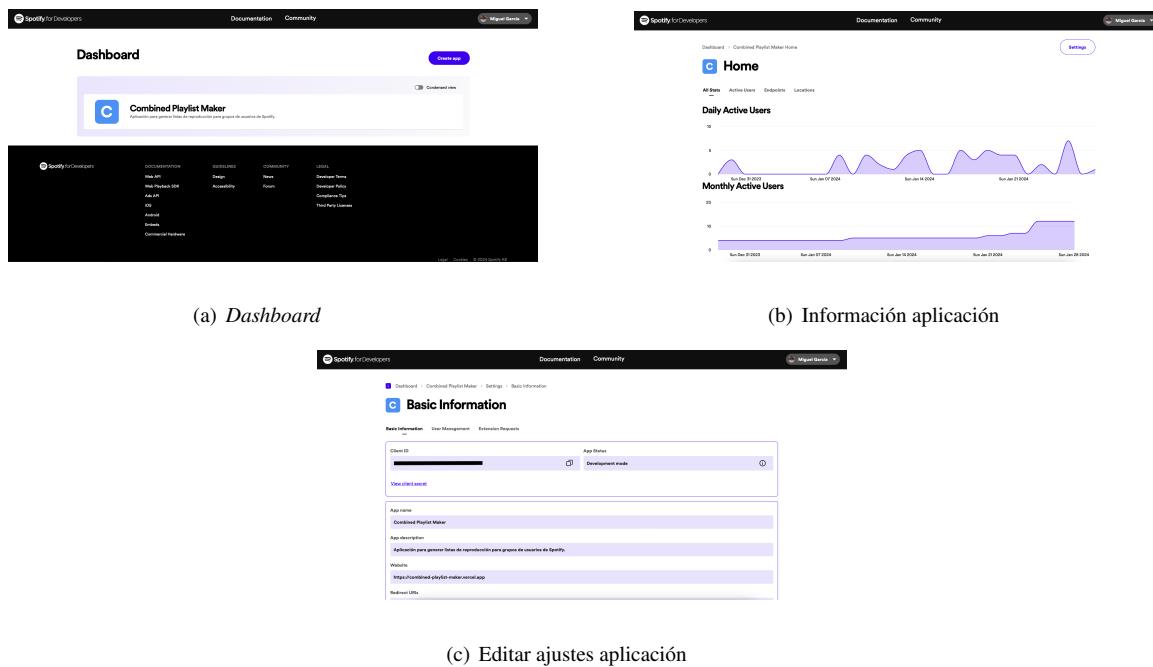


Figura 3.5: Visión general *Spotify for Developers*

Imágenes extraídas de developer.spotify.com

Para hacerse una idea inicial de qué tipos de *endpoints* nos harán falta, se puede hacer una primera aproximación a la aplicación que se quiere desarrollar. Vamos a suponer que un grupo de usuarios de *Spotify* está reunido e interesado en crear una playlist común que combine sus gustos musicales para escuchar en ese momento (o en un futuro). Teniendo en cuenta esta situación, y el resultado que se quiere obtener, podemos listar qué tipos de *endpoints* o interacciones con la API nos harán falta:

- **Datos de cuenta de los usuarios**, es decir, necesitaremos que los usuarios hagan *login* con sus cuentas de *Spotify*, para mostrar su información básica, como su foto de perfil y nombre de usuario, y obtener un token de acceso para poder hacer posteriores llamadas a la API.
- **Canciones más escuchadas** porque es necesario saber sobre los gustos de los usuarios y, obtener las canciones más escuchadas de cada uno de los usuarios que forman el grupo, parece una buena forma de hacerlo.
- **Recomendaciones**, porque así, podremos aprovecharnos de las estrategias de recomendación individuales de *Spotify*, y obtener recomendaciones de canciones que puedan gustar a los usuarios que forman el grupo, para así generar la playlist común.
- **Crear playlist**, porque así los usuarios podrán guardar en sus cuentas de *Spotify* la playlist que se haya generado.

Todos estos *endpoints* están disponibles en la API de Spotify, cuya URL base es "<https://api.spotify.com/v1>". Vamos a hacer un recorrido por cada uno de ellos.

Datos de cuenta de los usuarios

Para que los usuarios puedan autenticarse en la aplicación con sus cuentas de Spotify, y así poder crear playlists combinadas con su grupo, se llevará a cabo el flujo de autenticación PKCE (*Proof Key for Code Exchange*) definido en [25], el cual es el recomendado por Spotify para aplicaciones móviles u aplicaciones web de una sola página, donde la clave secreta del cliente no puede ser protegida de manera segura. Se puede ver un gráfico del flujo en la Figura 3.6.

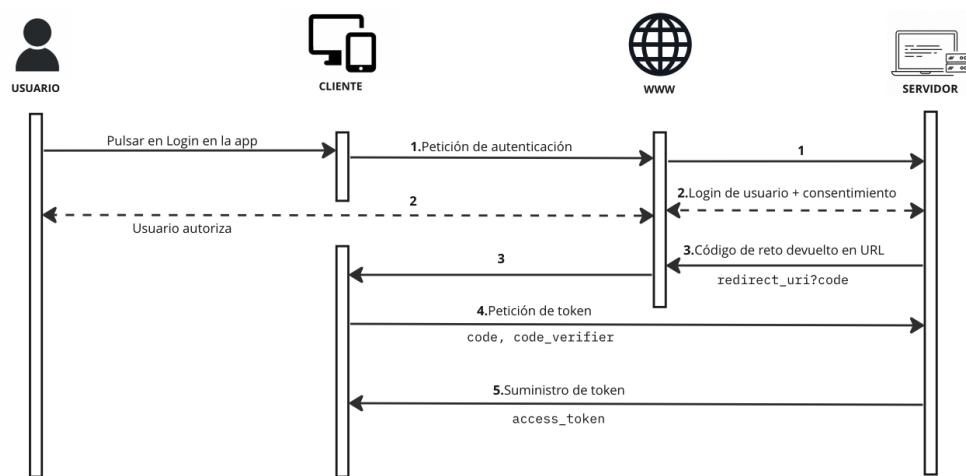


Figura 3.6: Flujo de autenticación PKCE

Para llevar a cabo esta autenticación, la app tendrá que lanzar la URL de autenticación de Spotify ("<https://accounts.spotify.com/authorize>"). Se explicará más en detalle la implementación de este flujo de autenticación en la Sección 3.3.1.

Una vez se lleva a cabo la autenticación del usuario con su cuenta de Spotify, ya podemos hacer llamadas a la API gracias al token de acceso que hemos obtenido. Comenzaremos por obtener los datos de la cuenta del usuario, como su foto de perfil, o su nombre de usuario. Para ello, Spotify nos ofrece un *endpoint* llamado *Get Current User's Profile* [26] cuya ruta es /me, GET es el método a utilizar, y no se necesitan parámetros adicionales.

Canciones más escuchadas

Obtener las canciones más escuchadas por los usuarios será un factor fundamental para la generación de playlists. Descubrimos que hay un *endpoint* que nos puede devolver los datos que necesitamos: En la documentación de la API [3] está definido como *Get User's Top Items* [27]. Y es que, resulta que podremos obtener tanto las canciones más escuchadas, como los artistas más escuchados. Veremos

cómo haremos uso de este *endpoint* para la implementación de nuestra app en la Sección 3.3.2.

Como se puede observar en la Figura 3.7, GET es el método a utilizar y se especifican los siguientes parámetros:

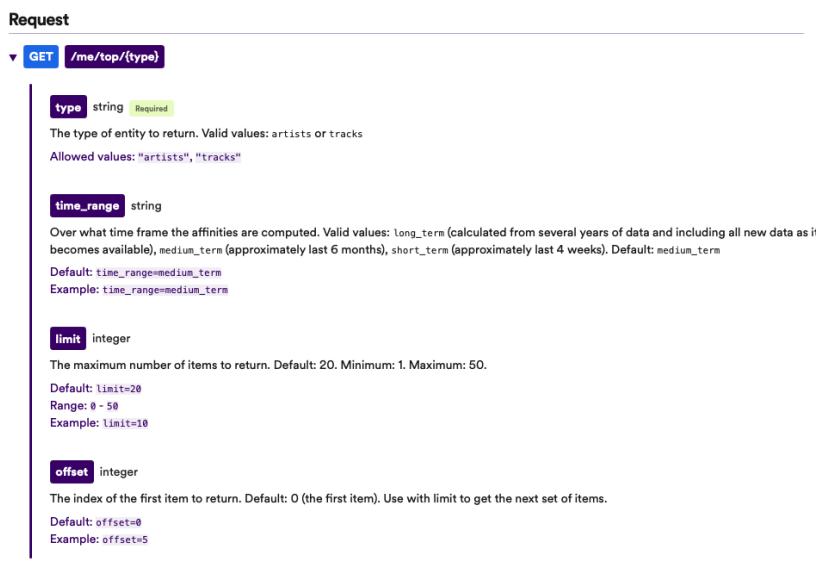


Figura 3.7: Obtener canciones o artistas más escuchados

Imagen extraída de developer.spotify.com

- **type**: para indicar si queremos obtener artistas o canciones.
- **time_range**: el período de tiempo sobre el que queremos obtener la lista *items*.
- **limit**: el número de *items* que querríamos obtener.
- **offset**: por si quisieramos obtener la lista a partir de una determinada posición.

Recomendaciones

Para obtener recomendaciones de canciones, Spotify nos ofrece un *endpoint* llamado *Get Recommendations* [28]. En la Sección 3.3.2 ahondaremos más en detalle sobre cómo se ha hecho uso de este *endpoint* para la implementación de la app. En la Figura 3.8, además de que GET es el método a utilizar, se pueden observar cinco parámetros que se pueden especificar:

- **limit**: el número de canciones que se quieren obtener.
- **market**: el mercado en el que deben estar presentes las canciones recomendadas, si se quisiera restringir.
- **seed_artists**: una lista de identificadores de artistas, a partir de los cuales, Spotify obtendrá canciones para recomendar.
- **seed_genres**: una lista de géneros, que funciona igual que la anterior.
- **seed_tracks**: una lista de identificadores de canciones, que funciona igual que las anteriores.

Para este *endpoint*, existen más parámetros opcionales que se pueden especificar, relacionados con la musicalidad, la energía, el tempo, la popularidad de las canciones, etc. Realmente tienen mucho

potencial para la obtención de recomendaciones, pero no se han utilizado en la aplicación desarrollada. No obstante, pueden ser algo muy a tener en cuenta para trabajo futuro, y lo comentaremos en el Capítulo 5.

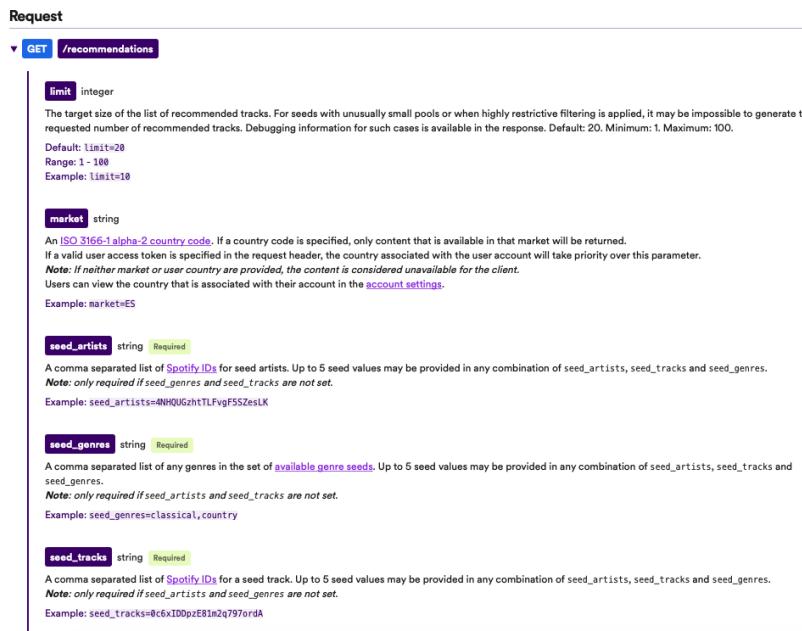


Figura 3.8: Obtener recomendaciones

Imagen extraída de developer.spotify.com

Crear playlist

Para crear una playlist, *Spotify* nos ofrece un *endpoint* llamado *Create Playlist* [29]. En la Figura 3.9, además de que `POST` es el método a utilizar, se pueden observar los parámetros a especificar en el cuerpo de la petición:

- `name`: el nombre de la playlist.
- `public`: si la playlist es pública o no. Es decir, si otros usuarios de *Spotify* podrán verla o no.
- `collaborative`: si la playlist es colaborativa o no. Es decir, si se creará una *playlist* con la capacidad de que se le añadan canciones por parte de otros usuarios, o no.
- `description`: la descripción de la playlist.

Con este *endpoint* se lleva a cabo la creación de la *playlist*, que se guardará en la cuenta de *Spotify* del usuario que se elija. Pero aún faltaría añadir las canciones a la *playlist* recién creada. Para ello, *Spotify* nos ofrece un *endpoint* llamado *Add Items to Playlist* [30]. En la Figura 3.10, además de observar que `POST` es el método a utilizar, se pueden observar de nuevo los parámetros a especificar en el cuerpo de la petición:

- `playlist_id`: el identificador de la *playlist* a la que se quieren añadir las canciones.
- `position`: la posición en la que se quieren añadir las canciones.

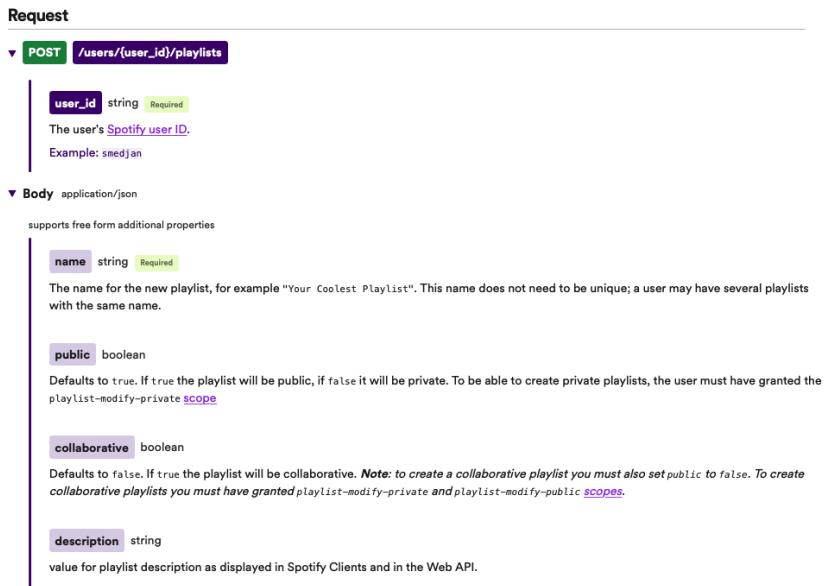


Figura 3.9: Crear playlist

Imagen extraída de developer.spotify.com

- uris: una lista de identificadores de canciones, que se añadirán a la *playlist*.

Debe observarse además, que las *uris* se pueden pasar en la URL de la petición si no son demasiadas; si no, se recomienda pasárlas en el cuerpo de la petición.

Resumen de los *endpoints* a utilizar

En la Tabla 3.1 se resume la información de los *endpoints* de la API de Spotify que se han comentado en esta sección.

3.3. Implementación de la aplicación

En esta sección trataremos la implementación de las partes más críticas e importantes de la aplicación, pasando desde la autenticación de usuarios, hasta la implementación de la agregación y obtención de *ratings* grupales sobre canciones. También se hablará de la implementación de los mecanismos para poder hacer pruebas de evaluación necesarias para el estudio de las estrategias de agregación, a utilizar para la generación de *playlists*, que sirvieron para definir las estrategias a utilizar en las pruebas con usuarios que se explican en la Sección 4.2.

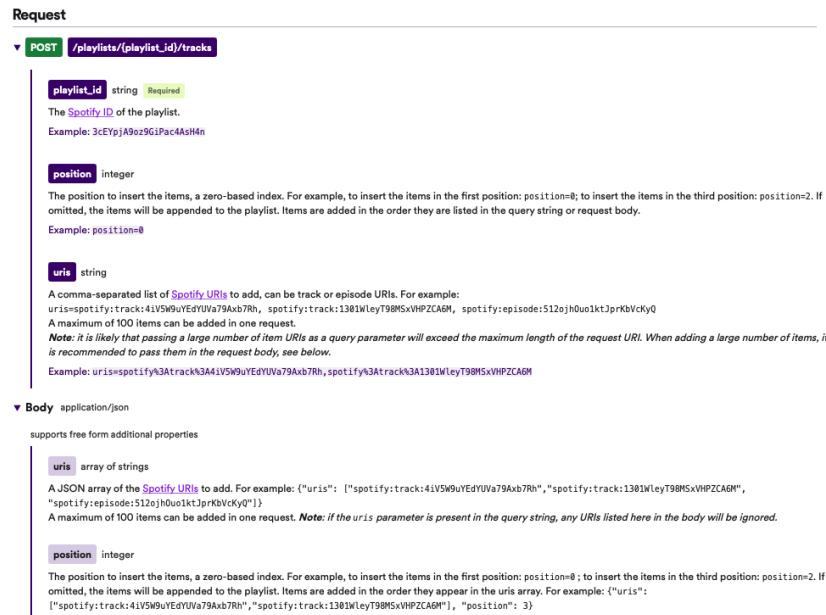
**Figura 3.10:** Añadir canciones a *playlist*

Imagen extraída de developer.spotify.com

URL Base<https://api.spotify.com/v1>

Nombre	Método	Auth Scopes	Ruta
Get Current User's Profile	GET	user-read-private user-read-email	/me
Get User's Top Items	GET	user-top-read	/me/top/{type}
Get Recommendations	GET	-	/recommendations
Get Available Genre Seeds	GET	-	/recommendations/available-genre-seeds
Create Playlist	POST	playlist-modify-public playlist-modify-private	/users/{user_id}/playlists
Add Items to Playlist	POST	playlist-modify-public playlist-modify-private	/playlists/{playlist_id}/tracks
Get Playlist	GET	-	/playlists/{playlist_id}

Tabla 3.1: Tabla resumen de los *endpoints* de la API de Spotify a utilizar.

3.3.1. Flujo de autenticación

La implementación de la autenticación de usuarios se llevó a cabo siguiendo el flujo de autenticación de *Spotify*, que explicábamos en la Sección 3.2.2. Inspirados en las instrucciones que daba *Spotify* en su documentación, se implementó un flujo de autenticación en Dart. A pesar de que encontramos posibles librerías de *Flutter* para llevar a cabo esta tarea, decidimos no utilizarlas para tener una total flexibilidad e independencia a la hora de hacer uso de la API de *Spotify*. Además, creímos que para un lenguaje como Dart, una librería como esta no estaría tan cuidada como podría estarlo para lenguajes mucho más extendidos como *Python* o *JavaScript*.

Como se explicaba en la Sección 3.2.2, después de que un usuario pulse en el botón para hacer login en la aplicación, se le redirige a una página de *Spotify* en la que se le pide que inicie sesión, y que dé permisos a la aplicación para acceder a sus datos. Véase el Código 3.1 para ver cómo se lleva a cabo el lanzamiento de la URL de autenticación en *Spotify*.

Código 3.1: Lanzar autenticación en *Spotify*

```

1 Future<void> requestAuthorization() async {
2     var codeVerifier = generateRandomString(128);
3
4     # guardamos el codeVerifier para el proceso de autenticacion
5     var cvBox = Hive.box('codeVerifiers');
6     await cvBox.put('cv', codeVerifier);
7
8     String codeChallenge = generateCodeChallenge(codeVerifier);
9
10    String state = generateRandomString(16);
11
12    final args = {
13        'response_type': 'code',
14        'client_id': clientId,
15        'scope': scope,
16        'redirect_uri': redirectUri,
17        'state': state,
18        'code_challenge_method': 'S256',
19        'code_challenge': codeChallenge,
20    };
21
22    final authorizationUrl =
23        Uri.https('accounts.spotify.com', '/authorize', args);
24
25    if (await canLaunchUrl(authorizationUrl)) {
26        await launchUrl(authorizationUrl, webOnlyWindowName: '_self');
27    } else {
28        throw 'Could not launch $authorizationUrl';
29    }
30}

```

Podemos ver en la línea 5 del Código 3.1 que se genera un `codeVerifier` aleatorio, que se guarda en nuestro almacenamiento local, pudiendo ver un ejemplo de cómo se guardan elementos en el almacenamiento local de la aplicación. Una vez que el usuario ha dado permisos a la aplicación, *Spotify* redirige al usuario a la URL que se ha especificado en la configuración de la aplicación, junto con un código de autorización. Este código de autorización es el que se utilizará para obtener el

access_token y el refresh_token que nos permitirán hacer la primera llamada a la API de Spotify para obtener los datos del usuario y plasmar en la aplicación el *login* exitoso. Véase el Código 3.2 para ver cómo se obtiene el código de autorización y se intercambia por el access_token y el refresh_token.

Código 3.2: Obtener access_token y refresh_token

```

1 Future<MyResponse> getAccessToken() async {
2     String code;
3
4     MyResponse ret = MyResponse();
5
6     # obtengo el codigo de la url y lo guardo
7     code = await obtainCurrentURLCode();
8
9     var cvBox = Hive.box('codeVerifiers');
10    var codeVerifier = await cvBox.get('cv');
11    cvBox.clear();
12    if (codeVerifier == null) {
13        # no hay un codeVerifier, asi que va a dar error, cortamos ejecucion
14        ret.content = {'error': 'No codeVerifier'};
15        return ret;
16    }
17    final body = {
18        'grant_type': 'authorization_code',
19        'code': code,
20        'redirect_uri': redirectUri,
21        'client_id': clientId,
22        'code_verifier': codeVerifier,
23    };
24
25    try {
26        final response = await post(
27            Uri.parse('https://accounts.spotify.com/api/token'),
28            headers: {'Content-Type': 'application/x-www-form-urlencoded'},
29            body: body,
30        );
31
32        ret.statusCode = response.statusCode;
33        if (response.statusCode == 200) {
34            final data = json.decode(response.body);
35            ret.content = data;
36            return ret;
37        } else {
38            ret.content = {};
39            throw Exception('HTTP status ${response.statusCode} in getAccessToken');
40        }
41    } catch (error) {
42        # solo se imprime el error si ejecutamos en local (en desarrollo)
43        if (kDebugMode) {
44            print('Error $error');
45        }
46        return ret;
47    }
48}

```

Este código se llama solamente una vez por usuario, porque obtendremos la información básica de este, y lo guardaremos en el almacenamiento local como un objeto de la clase User, como comentábamos en la Sección 3.2.2. Si el usuario ha introducido sus credenciales de Spotify correctamente, y ha dado permisos a la aplicación, recogeremos la información básica del usuario, como su nombre, su

email, y su *ID* de *Spotify*, y lo guardaremos en el almacenamiento local, véase el Código 3.3 para ver cómo se obtiene la información del usuario.

Código 3.3: Guardar información del usuario

```

1 Future<MyResponse> retrieveSpotifyProfileInfo() async {
2     MyResponse ret = MyResponse();
3     MyResponse tokenResponse = await getAccessToken();
4     if (tokenResponse.statusCode != 200) {
5         # Error en la obtencion del token
6         ret.statusCode = tokenResponse.statusCode;
7         ret.content = User.notValid();
8         return ret;
9     }
10
11    final Uri uri = Uri.parse('https://api.spotify.com/v1/me');
12    User newUser;
13
14    try {
15        final response = await get(
16            uri,
17            headers: {
18                'Authorization': 'Bearer ${tokenResponse.content["access_token"]}'
19            },
20        );
21        ret.statusCode = response.statusCode;
22        if (response.statusCode == 200) {
23            final data = json.decode(response.body);
24
25            # Creamos un objeto usuario a partir de la informacion que nos devuelve Spotify
26            newUser = User.fromJson(data, tokenResponse.content['access_token'],
27                tokenResponse.content['refresh_token']);
28
29            # Guardamos el nuevo usuario en el almacenamiento local
30            var usersBox = Hive.box<User>('users');
31            await usersBox.put(newUser.id, newUser);
32
33            ret.content = newUser;
34            return ret;
35        } else {
36            ret.content = User.notValid();
37            throw Exception(
38                'HTTP status ${response.statusCode} en retrieveSpotifyProfileInfo');
39        }
40    } catch (error) {
41        # solo se imprime el error si ejecutamos en local (en desarrollo)
42        if (kDebugMode) {
43            print('Error: $error');
44        }
45        return ret;
46    }
47 }
```

Podemos observar en los fragmentos de código expuestos cómo hacemos uso del objeto `MyResponse` para devolver información, donde el `statusCode` y `content` son fundamentales para el manejo de errores y la obtención de información. También puede verse en la línea 3 del Código 3.3 cómo se hace la llamada al código 3.2. Esta llamada se incluye dentro de la misma función porque en nuestra aplicación, obtener el `access_token` y obtener la información del perfil del usuario van de la mano. En las líneas 18 y 26 del Código 3.3 se hace uso de los tokens para hacer la llamada para obtener la información del usuario, y para guardar la información del usuario en el almacenamiento local,

respectivamente.

Vamos a ver para terminar cómo después se accede a esta información del usuario en el almacenamiento local, para utilizarla. En el Código 3.4 se puede ver cómo se accede a la información del usuario en el almacenamiento local, y cómo se hace uso de ella, en este caso, para utilizar el `access_token` para realizar una petición después.

Código 3.4: Acceder a la información del usuario

```

1 Future<MyResponse> getUsersTopItems(
2     String userId, String type, String timeRange, double limit) async {
3
4     MyResponse ret = MyResponse();
5
6     # Accedemos al almacenamiento local para obtener el usuario
7     var usersBox = Hive.box<User>('Users');
8     User? user = usersBox.get(userId);
9     var accessToken = user!.accessToken;
10
11    final args = {
12        'time_range': timeRange,
13        'limit': limit.round().toString(),
14    };
15
16    final Uri uri = Uri.https('api.spotify.com', '/v1/me/top/$type', args);
17    try {
18        final response = await get(
19            uri,
20            headers: {'Authorization': 'Bearer $accessToken'},
21        );
22        ret.statusCode = response.statusCode;
23        if (response.statusCode == 200) {
24            final data = json.decode(response.body);
25            ret.content = parseItemData(data, tipo);
26            return ret;
27        } else {
28            ret.content = [];
29            throw Exception('HTTP status ${response.statusCode} en getUsersTopItems');
30        }
31    } catch (error) {
32        # solo se imprime el error si ejecutamos en local (en desarrollo)
33        if (kDebugMode) {
34            print('Error: $error');
35        }
36        return ret;
37    }
38}

```

En las líneas 7 y 9 del Código 3.4 se accede al almacenamiento local para obtener el usuario, y en la línea 9 se accede al `access_token`.

3.3.2. Llamadas a la API de Spotify y procesamiento de respuestas

En el Código 3.3 se puede ver cómo se hace uso de la librería `http` para hacer llamadas a la API de *Spotify*. En la línea 15 se hace uso de la función `get` para hacer una llamada GET a la URI correspondiente, y se le pasan los `headers` necesarios para que *Spotify* nos devuelva la información

del usuario. En la línea 23 se hace uso de la función `json.decode` para procesar la respuesta de *Spotify*, con la cual se crea un objeto de la clase `User`, que se guarda en el almacenamiento local.

Este proceso se puede extrapolar al resto de peticiones a la API, donde cambiarán posibles argumentos que haya que pasar, y cambiará la URI a la que se haga la petición, dependiendo del *endpoint* (resumidos en la Tabla 3.1) al que se quiera hacer la petición.

Una cuestión a comentar es el procesamiento de los 'objetos' que devuelve *Spotify*, ya que nos devolverá listas con artistas y canciones en formato JSON. Por ello, vamos a comentar cómo se procesan estos objetos en la aplicación. Por ejemplo, cuando se pide la lista de los artistas o canciones más escuchados por un usuario (véase el Código 3.4), se obtiene un objeto en formato JSON, el cual procesa artistas o usuarios en la línea 25, dependiendo del tipo de objeto que se haya pedido. Véase el Código 3.5, el cual corresponde a la función para procesar artistas o canciones.

Código 3.5: Procesar objeto en formato JSON

```

1  List parseItemData(data, type) {
2      List topItems = [];
3      if (type == 'artists') {
4          for (var item in data['items']) {
5              Artist a = Artist.fromJson(item);
6              topItems.add(a);
7          }
8      } else {
9          for (var item in data['items']) {
10             Track a = Track.fromJson(item);
11             topItems.add(a);
12         }
13     }
14   }
15   return topItems;
16 }
```

Vemos en las líneas 6 y 11 del Código 3.5 que hacemos uso de un constructor personalizado (`fromJson`) para crear objetos de las clases `Artist` y `Track` a partir de los datos que nos devuelve *Spotify*. También tenemos ese constructor personalizado para la clase `User`, el cual se utilizaba en el Código 3.3 en la línea 26, solo que a este se le pasaba además el `access_token` y el `refresh_token` que se obtienen en el Código 3.2. Vamos a ver un ejemplo de cómo se implementa este constructor personalizado en la clase `User` en el Código 3.6, los constructores de las otras clases se implementan de la misma forma, obviando las diferencias que tengan los objetos JSON que devuelve *Spotify*.

Código 3.6: Constructor personalizado de la clase `User`

```

1 factory User.fromJson(
2     Map<String, dynamic> json, String token, String refreshToken) {
3     String imageUrl = '';
4     if ((json['images'] as List).isNotEmpty) {
5         # La lista de imágenes no está vacía y puedes acceder a json['images'][0]
6         imageUrl =
7             json['images'][1]['url']; // Selecciona la segunda imagen de la lista
8     } else {
9         # La lista de imágenes está vacía o no existe la clave 'images' en el JSON
```

```

10     imageUrl = '';
11 }
12
13 return User(
14   displayName: json['display_name'],
15   id: json['id'],
16   email: json['email'],
17   imageUrl: imageUrl,
18   country: json['country'],
19   followers: json['followers']['total'],
20   accessToken: token,
21   refreshToken: refreshToken,
22 );
23 }
```

3.3.3. Generación de *playlist* combinada

En la Secciones 2.3.2 y 2.3.3 comentábamos cómo íbamos a llevar a cabo la agregación y obtención de *ratings* grupales para las recomendaciones obtenidas de los usuarios. Vamos a tratar de explicar este proceso paso a paso en esta sección. Desde la obtención de las recomendaciones de los usuarios, hasta la agregación de estas recomendaciones, y la obtención de los *ratings* grupales, que se utilizarán para obtener las *playlists* generadas para los grupos. Cuando se selecciona en la aplicación la opción de generar una *playlist* para un grupo, se inicia un proceso que se recoge en la función `generateCombinedPlaylist`, véase el Código 3.7.

Código 3.7: Generar *playlist* combinada

```

1 Future<MyResponse> generateCombinedPlaylist(
2   Duration duration, String? type) async {
3   MyResponse ret = MyResponse();
4
5   # Obtenemos las recomendaciones de todos los usuarios del grupo
6   MyResponse recommendations = await obtainAllUsersRecommendations();
7
8   if (recommendations.statusCode != 200) {
9     ret.statusCode = recommendations.statusCode;
10    ret.content = recommendations.content;
11    ret.auxContent = recommendations.auxContent;
12    return ret;
13  }
14  # Generamos la playlist combinada agregando todas las recomendaciones
15  Map<String, List> playlists = generateRecommendedPlaylist(
16    recommendations.content, duration, type);
17
18
19  ret.statusCode = recommendations.statusCode;
20  ret.content = playlists;
21
22  return ret;
23 }
```

Esta función, aparentemente corta, en la línea 6 del Código 3.7 obtiene las recomendaciones de todos los usuarios del grupo, y en la línea 15 se generan las *playlists* recomendadas. En esta última línea es donde se agregan todas las recomendaciones de los usuarios del grupo, asignando primero *ratings* individuales a las recomendaciones de cada usuario, para después aplicar la estrategia de

agregación pertinente y obtener los *ratings* grupales, que se utilizarán para la generación de la *playlist* recomendada. Vamos a cubrir cada uno de los principales pasos de este proceso en las siguientes subsecciones.

Obtención de recomendaciones de los usuarios

En primer lugar se obtienen recomendaciones para los usuarios con la función `getRecommendations`, la cual cuenta con elementos en el código muy similares a fragmentos como el Código 3.4. Lo que merece la pena comentar aquí son los argumentos que se le pasan a la URI, `seedArtists`, `seedTracks` y `seedGenres`, que son los artistas, canciones y géneros a partir de los cuales se obtendrán las recomendaciones. Véase en el fragmento de Código 3.8 el uso y la definición de estos argumentos que se le pasan a la función `getRecommendations`.

Código 3.8: Definición de los argumentos `seedArtists` y `seedTracks`

```

1 Future<MyResponse> getRecommendations(String userId, List<String> seedArtists,
2     List<String> seedTracks, double limit) async {
3
4     # ...
5
6     final args = {
7         'market': 'ES',
8         'limit': limit.round().toString(),
9         'seed_artists': seedArtists,
10        'seed_genres': seedGenres,
11        'seed_tracks': seedTracks,
12    };
13
14     final Uri uri = Uri.https('api.spotify.com', '/v1/recommendations', args);
15     try {
16         final response = await get(
17             uri,
18             headers: {'Authorization': 'Bearer $accessToken'},
19         );
20         # ...
21     }
22 }
```

Los artistas y canciones los podemos obtener de las listas de artistas y canciones más escuchadas de cada uno de los usuarios del grupo con la función `getUsersTopItems` (puede verse en el Código 3.4). Además, es importante saber que la API de Spotify nos permite utilizar hasta 5 semillas en total, en la proporción que decidimos.

Decisión sobre las semillas a partir de las cuales se obtendrán las recomendaciones.

Debemos destacar aquí una decisión más o menos arbitraria, y es que decidimos que para la obtención de las recomendaciones a la hora de generar una *playlist* para un grupo, se utilizarán los 2 artistas más escuchados en el largo plazo y las 3 canciones más escuchadas en el corto plazo por cada usuario del grupo. No utilizaremos semillas de géneros musicales. Esta decisión se tomó teniendo en cuenta dos ideas:

- **Variedad:** se decidió que se utilizarán los 2 artistas más escuchados en el largo plazo para que las recomendaciones no se basaran solamente en los gustos recientes de los usuarios y tuvieran los artistas favoritos de los usuarios, y se decidió que se utilizarán las 3 canciones más escuchadas en el corto plazo para tener en cuenta las canciones a las que un usuario pueda tener una preferencia en el corto plazo.
- **Equilibrio:** se decidió que se utilizarán 2 artistas y 3 canciones para que las recomendaciones no se basaran solamente en un tipo de ítem, teniendo en cuenta algo más amplio como los artistas, y algo más específico como las canciones.
- **Limitaciones en las semillas de género:** se decidió que no se utilizaran semillas de géneros musicales porque no podemos saber los géneros más escuchados por un usuario de manera tan sencilla, y no creímos que fuera a aportar algo muy especial.

Véase el Código 3.9 para ver cómo se obtienen las recomendaciones para todos los usuarios del grupo, teniendo como argumentos por defecto el número de canciones y artistas a utilizar como semillas, y la ventana temporal a utilizar para obtener las semillas de cada tipo.

Código 3.9: Obtención de recomendaciones para un usuario

```

1 Future<MyResponse> obtainAllUsersRecommendations(
2     {double numTracks = 3,
3      double numArtists = 2,
4      String tracksTerm = 'short_term',
5      String artistsTerm = 'long_term'}) async {
6
7     MyResponse ret = MyResponse();
8     var usersBox = Hive.box<User>('users').toMap();
9
10    Map<String, List> recommendations = {};
11
12    for (User user in usersBox.values) {
13        String userId = user.id;
14
15        # Actualización del token de acceso para asegurarnos de que no haya problemas
16        await user.updateToken();
17
18        # Obtención de las semillas de canciones
19        MyResponse topTracks =
20            await getUsersTopItems(userId, 'tracks', tracksTerm, numTracks);
21        # Comprobación de errores...
22        List? trackSeeds = topTracks.content.map((track) => track.id).toList();
23
24        # Obtención de las semillas de artistas
25        MyResponse topArtists =
26            await getUsersTopItems(userId, 'artists', artistsTerm, numArtists);
27        # Comprobación de errores...
28        List? artistSeeds = topArtists.content.map((artist) => artist.id).toList();
29
30        # Obtención de las recomendaciones del usuario en base a las semillas obtenidas
31        MyResponse userRecommendations =
32            await getRecommendations(userId, artistSeeds, trackSeeds, 100);
33        # Comprobación de errores...
34        ret.statusCode = userRecommendations.statusCode;
35
36        # Se añaden las recomendaciones obtenidas para ese usuario
37        recommendations[userId] = userRecommendations.content;
38    }
39    ret.content = recommendations;
40    return ret;
41 }
```

Agregación de recomendaciones

Una vez que se han obtenido las recomendaciones de todos los usuarios del grupo, salimos del módulo de peticiones a la API y se procede a la agregación de estas recomendaciones desde el módulo de recomendación. Véase el Código 3.10 para ver cómo se lleva a cabo este proceso.

Código 3.10: Proceso de agregación

```

1 Map<String, List<dynamic>> generateRecommendedPlaylist(
2     Map<String, List> recommendations,
3     Duration playlistDuration,
4     List<int> seedProp,
5     String? tipo) {
6     Map groupRatings = {};
7     Map sortedTracks = {};
8     Map recommendation = {};
9
10    # Obtenemos los ratings individuales
11    Map<Track, List<double>> ratings =
12        obtainIndividualRatings(recommendations, seedProp);
13
14    # Aplicamos la estrategia de agregación del diccionario strategies correspondiente
15    groupRatings[tipo!] = strategies[tipo]!(recommendations, ratings);
16
17    # Ordenamos las recomendaciones por rating de mayor a menor
18    sortedTracks[tipo] = sortedRecommendation(groupRatings[tipo]!);
19
20    # Cortamos las recomendaciones para que no superen la duración indicada por el usuario
21    recommendation[tipo] =
22        cutOrderedRecommendations(sortedTracks[tipo]!, playlistDuration);
23
24    # Devolvemos la playlist generada
25    return recommendation;
26 }
```

En primer lugar, se obtienen los *ratings* individuales de las recomendaciones de cada usuario, que se obtienen con la función `obtainIndividualRatings`, que se puede ver en el Código 3.11.

Decisión sobre los *ratings* individuales.

Se decidió que el *rating* de una canción para un usuario se basara en el orden de la canción en la lista de recomendaciones para ese usuario. Es decir, si una canción es la primera de la lista de recomendaciones para un usuario, su *rating* será el más alto, si es la segunda, su *rating* será el segundo más alto, y así sucesivamente. Véase el Código 3.11 para ver cómo se obtienen los *ratings* individuales.

Código 3.11: Obtención de *ratings* individuales

```

1 Map<Track, List<double>> obtainIndividualRatings(
2     Map<String, List> recommendations, List<int> seedProp) {
3     # Inicialización de variables
4     Map<Track, List<double>> ratings = {};
5     int userNum = 0;
6     int totalUsers = recommendations.length;
7     List<String> trackIds = [];
8
9     for (List trackList in recommendations.values) {
```

```

10    double subtract = 0; # Valor que se sustrae al valor de ranking invertido
11    for (var trackPos = 0; trackPos < trackList.length; trackPos++) {
12        Track track = trackList[trackPos];
13
14        # Calculo de rating de una cancion
15        double trackRating = trackList.length - subtract;
16
17        if (trackIds.contains(track.id)) {
18            # El track ya se habia recomendado a otro usuario
19            # Se anade el rating del usuario que corresponda
20            ratings[track]![userNum] = trackRating;
21        } else {
22            # El track ha sido analizado por primera vez entre las recomendaciones
23            # Todas las listas de ratings de cada track tendran un rating de 0
24            # De todos los usuarios
25            ratings[track] = List.filled(totalUsers, 0);
26            # Se pone el valor correspondiente para el usuario pertinente
27            ratings[track]![userNum] = trackRating;
28            trackIds.add(track.id);
29        }
30        subtract += 1;
31    }
32    userNum += 1;
33 }
34
35     return ratings;
36 }
```

Tras darles un *rating* a todas las recomendaciones de cada usuario, se aplica la estrategia de agregación correspondiente, que se encuentra en el diccionario `strategies`, y que ha podido ser elegida por el usuario previamente. Véase el fragmento de Código 3.12 con el diccionario de estrategias de agregación.

Código 3.12: Diccionario de estrategias de agregación

```

1 Map<String, Function> strategies = {
2     'average': averageGroupRatings,
3     'multiplicative': multiplicativeGroupRatings,
4     'most_pleasure': mostPleasureGroupRatings,
5     'least_misery': leastMiseryGroupRatings,
6     'borda': bordaGroupRatings,
7     'average_custom': averageCustomGroupRatings,
8 };
```

El diccionario tiene como clave una cadena con el nombre en minúsculas de la estrategia, y como valor, una función que lleva a cabo la estrategia de agregación correspondiente. Véase el Código 3.13 para ver un ejemplo de cómo se implementa una de las estrategias.

Código 3.13: Estrategia de agregación *average*

```

1 Map<Track, double> multiplicativeGroupRatings(
2     Map<String, List> recommendations, Map<Track, List<double>> ratings) {
3
4     Map<Track, double> multGroupRatings = ratings.map((track, ratingsList) {
5         # Multiplicacion de los ratings individuales, para obtener uno grupal
6         double multiplicationRating = ratingsList
7             .where((rating) => rating != 0)
8             .reduce((double a, double b) => a * b);
9
10    return MapEntry(track, multiplicationRating);
11 }
```

```

11     });
12
13     return multGroupRatings;
14 }

```

Todas las estrategias siguen una implementación similar a la del Código 3.13, y devuelven un diccionario con `Track` como clave y `double` como valor, que son los *ratings* grupales.

Obtención de la *playlist* final

Una vez que se han obtenido los *ratings* grupales, se ordenan las recomendaciones por *rating* de mayor a menor, como se puede ver en el Código 3.14, y se cortan las recomendaciones para que no superen la duración indicada por el usuario, como se puede ver en el Código 3.15.

Código 3.14: Ordenar recomendaciones

```

1 List<Track> sortedRecommendation(Map<Track, dynamic> groupRatings) {
2     List<Track> sortedTracks = groupRatings.keys.toList();
3     sortedTracks.sort((a, b) {
4         int c = groupRatings[b].compareTo(groupRatings[a]);
5         # En caso de tener mismo rating, ordenar alfabeticamente
6         if (c == 0) {
7             return a.name.compareTo(b.name);
8         } else {
9             return c;
10        }
11    });
12    return sortedTracks;
13 }

```

Vemos cómo se ordenan las recomendaciones de mayor a menor en la línea 4 del Código 3.14, y se obtiene una lista ordenada de `Track`. En el Código 3.15 se muestra cómo se corta la lista para que no supere la duración deseada.

Código 3.15: Cortar recomendaciones

```

1 List cutOrderedRecommendations(
2     List orderedRecommendations, Duration playlistDuration) {
3     List cutRecommendations = [];
4     int currentDuration = 0;
5     # La duracion de cada Track esta guardada en milisegundos
6     int desiredDuration = playlistDuration
7         .inMilliseconds;
8
9     for (Track track in orderedRecommendations) {
10        if (currentDuration + track.durationMs <= desiredDuration) {
11            cutRecommendations.add(track);
12            currentDuration += track.durationMs;
13        } else if (currentDuration + track.durationMs <
14            desiredDuration + (desiredDuration * 0.2)) {
15            # Si la duracion de la lista de reproduccion NO excede en mas de 0.2 la duracion deseada
16            # Agregamos la pista y continuamos con la siguiente
17            cutRecommendations.add(track);
18            currentDuration += track.durationMs;
19            break;
20        } else if (currentDuration + track.durationMs >=
21            desiredDuration + (desiredDuration * 0.2)) {
22            # Si la duracion de la lista de reproduccion excede en mas de 0.2 la duracion deseada

```

```

23     # NO agregamos la pista y continuamos con la siguiente
24     continue;
25 } else {
26     break;
27 }
28 }

30 return cutRecommendations;
31 }
```

Tras esta sucesión de procesos se obtiene la *playlist* final, que se devuelve como una lista de Track, lista para ser mostrada en nuestra aplicación.

TAMBIEN UN DIAGRAMA RESUMEN QUE EXPLIQUE TODO EL PROCESO Y TODAS LAS INTERACCIONES ENTRE LAS FUNCIONES NO SE DE QUE TIPO EL DIAGRAMA AÚN, Y NO SÉ SI METERLO EN EL APÉNDICE.

3.3.4. Pruebas de evaluación

Tras haber hecho un recorrido por todo el proceso de generación de *playlists* para grupos, queda una cuestión importante por tratar, y es la de investigar sobre las distintas estrategias de agregación que se han implementado, y ver cómo se comportan, si son similares, si no, y si merece la pena destacar alguna. Para ello, llevamos a cabo la implementación de un módulo destinado a la recogida y acumulación de datos (*statistics.dart*) sobre las distintas estrategias de agregación, las cuales se pueden ver en el Código 3.12.

Lo que se busca con la función *checkAllStrategiesAllDurations* de este módulo es, como su propio nombre indica, generar, a partir de las mismas recomendaciones de los usuarios, *playlists* para grupos con distintas duraciones, y con todas las estrategias de agregación. Véase el Código 3.16 para ver cómo se lleva a cabo este proceso.

Código 3.16: Generación de estadísticas

```

1 Future<Map> checkAllStrategiesAllDurations() async {
2     List<Duration> durations = [
3         Duration(minutes: 10),
4         Duration(minutes: 20),
5         # ...
6         Duration(minutes: 570),
7         Duration(minutes: 600),
8     ];
9
10    Map<String, dynamic> results = {};
11
12    # Obtener todas las recomendaciones, luego generar una lista de reproducción para cada estrategia y duración
13    MyResponse recommendationsResponse = await obtainAllUsersRecommendations();
14    if (recommendationsResponse.statusCode == 200) {
15        Map<String, List> recommendations = recommendationsResponse.content;
16        # Lista de IDs de usuarios
17        results['users'] = recommendations.keys.toList();
18        # Similaridad entre usuarios (coincidencias en sus semillas para generar recomendaciones)
19        results['users_similarity'] = recommendationsResponse.auxContent;
20        # Aquí se guarda el valor de similaridad entre estrategias para cada duración
```

```

21     results['strategies_overlapping'] = {};
22
23     for (Duration duration in durations) {
24         String durationKey = duration.inMinutes.toString();
25         results['strategies_overlapping'][durationKey] = {};
26         # Generar recomendaciones para cada estrategia con 'all'
27         Map<String, List<dynamic>> recommendation = generateRecommendedPlaylist(
28             recommendations, duration, 'all');
29
30         # Calcular la similitud entre las recomendaciones de cada estrategia
31         Map<String, Map<String, double>> overlapping =
32             calculateOverlappingBetweenPlaylists(recommendation);
33         results['strategies_overlapping'][durationKey] = overlapping;
34     }
35 } else {
36     # Impresión de errores si se ejecuta en local
37     if (kDebugMode) {
38         print('Error obtaining all recommendations');
39         print(recommendationsResponse);
40     }
41 }
42 return results;
43 }
```

En la línea 13 del Código 3.16 podemos ver cómo se obtienen las recomendaciones de los usuarios, a partir de las cuales se generarán las *playlists* para grupos con distintas duraciones y con todas las estrategias de agregación, como comentábamos. En la línea 19 guardamos la similitud entre usuarios, que es un diccionario extra (véase el Código 3.17) cuya obtención implementamos con este propósito dentro de la función `obtainAllUsersRecommendations`, así sabremos si entre usuarios del grupo hay coincidencias en sus semillas de artistas o canciones para generar recomendaciones, lo cual será útil para el análisis de los resultados del Capítulo 4.

Código 3.17: Diccionario de similitud entre usuarios

```

1  {
2      "artists": {
3          "id1": {
4              "id2": value,
5              "id3": value
6          },
7          "id2": {
8              "id1": value,
9              "id3": value
10         },
11         "id3": {
12             "id1": value,
13             "id2": value
14         }
15     },
16     "tracks": {
17         "id1": {
18             "id2": value,
19             "id3": value
20         },
21         "id2": {
22             "id1": value,
23             "id3": value
24         },
25         "id3": {
26             "id1": value,
27             "id2": value
28     }}
```

```

28     }
29 }
30 }
```

En la línea 27 del Código 3.16 se obtienen las recomendaciones para cada estrategia en un mapa llamado `recommendation` (para su estructura, véase el Código 3.18), y en la línea 32 del Código 3.16 se calcula la similitud entre las recomendaciones de cada estrategia, que se guarda en una entrada del mapa `results` llamada `strategies_overlapping`, con la respectiva duración.

Código 3.18: Diccionario de recomendaciones

```

1 {
2   "estrategia1": [
3     track1,
4     track2,
5     ...,
6     track3
7   ],
8   "estrategia2": [
9     track4,
10    track3,
11    ...,
12    track6
13 ],
14   "estrategia3": [
15     track1,
16     track7,
17     ...,
18     track9
19 ]
20 }
```

La obtención de la similitud en las estrategias se lleva a cabo con la función `calculateOverlappingBetweenPlaylists`, la cual no hace más que ir comparando las recomendaciones de cada estrategia con las de las demás, comparando los `Track` que aparecen en cada una de ellas con sus `id`, y devolviendo un mapa con las similitudes entre estrategias con la estructura que se observa en el Código 3.19.

Código 3.19: Diccionario de similitud entre estrategias para una duración

```

1 {
2   "average": {
3     "multiplicative": value,
4     "most_pleasure": value,
5     "least_misery": value,
6     "borda": value,
7     "average_custom": value
8   },
9   "multiplicative": {
10     "average": value,
11     "most_pleasure": value,
12     "least_misery": value,
13     "borda": value,
14     "average_custom": value
15   },
16   "most_pleasure": {
17     "average": value,
18     "multiplicative": value,
19     "least_misery": value,
```

```

20     "borda": value,
21     "average_custom": value
22 },
23 "least_misery": {
24     "average": value,
25     "multiplicative": value,
26     "most_pleasure": value,
27     "borda": value,
28     "average_custom": value
29 },
30 "borda": {
31     "average": value,
32     "multiplicative": value,
33     "most_pleasure": value,
34     "least_misery": value,
35     "average_custom": value
36 },
37 "average_custom": {
38     "average": value,
39     "multiplicative": value,
40     "most_pleasure": value,
41     "least_misery": value,
42     "borda": value
43 }
44 }
```

Así, en el mapa `results` se guardan los resultados de la evaluación de las estrategias de agregación, junto con los usuarios implicados, y la similitud entre ellos, para su posterior análisis en el Capítulo 4. Véase el Código 3.20 para ver el formato del mapa `results`.

Código 3.20: Formato del mapa `results`

```

1  {
2      "users": ["id1", "id2", "id3"],
3      "users_similarity": {
4          # ...
5      },
6      "strategies_overlapping": {
7          "10": {
8              "strategy1": {
9                  "strategy2": value,
10                 # ...
11                 "strategyN": value
12             },
13             # ...
14             "strategyN": {
15                 "strategy1": value,
16                 # ...
17                 "strategyN-1": value
18             }
19         },
20         ...
21     }
22 }
```

3.3.5. Resumen

En esta sección hemos cubierto la implementación de la generación de *playlists* para grupos, desde la autenticación de los usuarios con Spotify en la Sección 3.3.1, pasando por las llamadas a la

API de *Spotify* y el procesamiento de las respuestas en la Sección 3.3.2, hasta la generación de *ratings* grupales y la obtención de la *playlist* final en la Sección 3.3.3. Para terminar, hemos cubierto la implementación de un módulo destinado a la recogida y acumulación de datos sobre las distintas estrategias de agregación en la Sección 3.3.4, que dan pie al siguiente capítulo, en el que analizaremos los resultados obtenidos, y explicaremos las posteriores pruebas con usuarios que hemos realizado.

PRUEBAS Y RESULTADOS

En este capítulo hablaremos sobre los experimentos y pruebas que se han realizado para evaluar la aplicación desarrollada, empezando por las pruebas *offline* que se han llevado a cabo para comparar distintas estrategias de agregación y acabando por las pruebas con usuarios reales, hablando de problemáticas y decisiones a las que nos han llevado estas pruebas.

4.1. Evaluaciones *offline*

Hemos hecho uso de estas evaluaciones *offline* para comparar distintas estrategias de agregación. Así seremos capaces de ver cuáles se comportaban de manera más similar a otras estrategias, y cuáles se comportaban de manera más diferente. Esto nos dará la posibilidad de suprimir estrategias que sean demasiado similares, para así centrar las comparaciones con usuarios en las estrategias que más diferencias presenten entre sí. Exponíamos en la Sección 3.3.4 cómo se implementaron los mecanismos para obtener los datos de similitud entre *playlists* generadas con distintas estrategias de agregación y la similitud entre usuarios con la coincidencia de semillas. En esta sección, expondremos los resultados obtenidos y las conclusiones que hemos sacado de ellos.

4.1.1. Resultados obtenidos

Los datos recogidos de las pruebas implementadas se han recogido en archivos JSON que se han procesado después con *Python* para generar gráficas que muestren la evolución de la similitud entre *playlists* generadas con distintas estrategias de agregación con el aumento de la duración de estas. Hemos realizado pruebas para grupos de 2, 3, 4 y 5 usuarios, en los cuales para algunos no existía ninguna similitud entre semillas, y para otros sí.

Grupo de 2 usuarios con similitud en semillas

En la Figura 4.1 se muestran las gráficas de evolución de la similitud de una estrategia frente a todas las demás para un grupo de 2 personas, en el cual, existía una coincidencia de una semilla

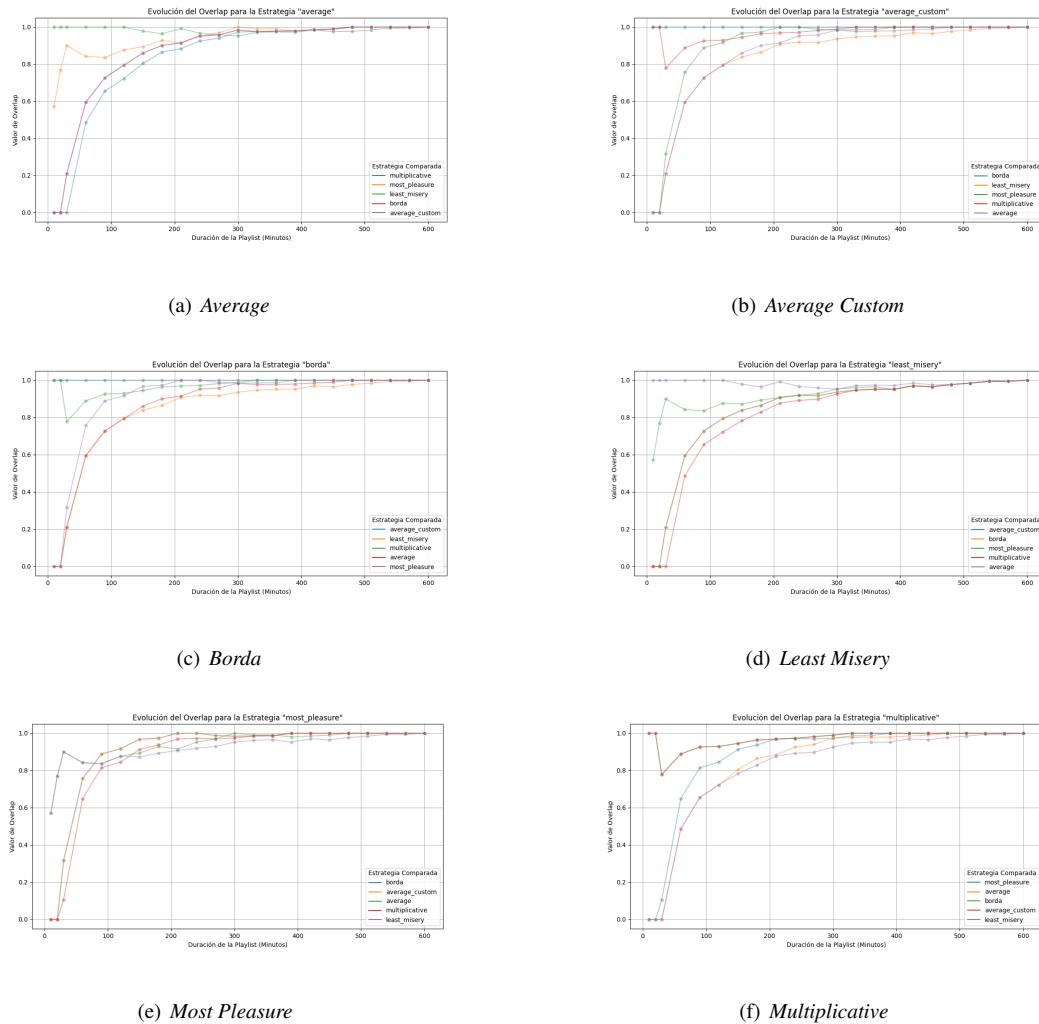


Figura 4.1: Gráficas de similitud entre estrategias para un grupo de 2 personas con similitud en semillas

de artista para la generación de recomendaciones. De estas primeras gráficas podemos sacar las siguientes conclusiones:

- Vemos que, entre algunas estrategias, el valor de similitud es muy alto con duraciones pequeñas, lo que nos indica que estas estrategias tienen un comportamiento similar. Por ejemplo, si nos fijamos en la gráfica de Average, vemos que las líneas de *Least Misery* y *Most Pleasure* se aproximan mucho más que el resto al máximo valor de similitud con duraciones pequeñas.
- Claramente, a medida que se aumenta la duración de la *playlist*, la similitud entre las estrategias aumenta, y tiende a ser total. Ya que, de los usuarios del grupo siempre obtenemos el mismo número de canciones candidatas.

Grupo de 3 usuarios sin similitud en semillas

Tras esta primera prueba con una pareja de usuarios que tienen cierta coincidencia en semillas, vamos a observar qué nos puede mostrar un grupo de 3 usuarios, entre los cuales no exista coincidencia en semillas. En la Figura 4.2 se muestran los resultados obtenidos.

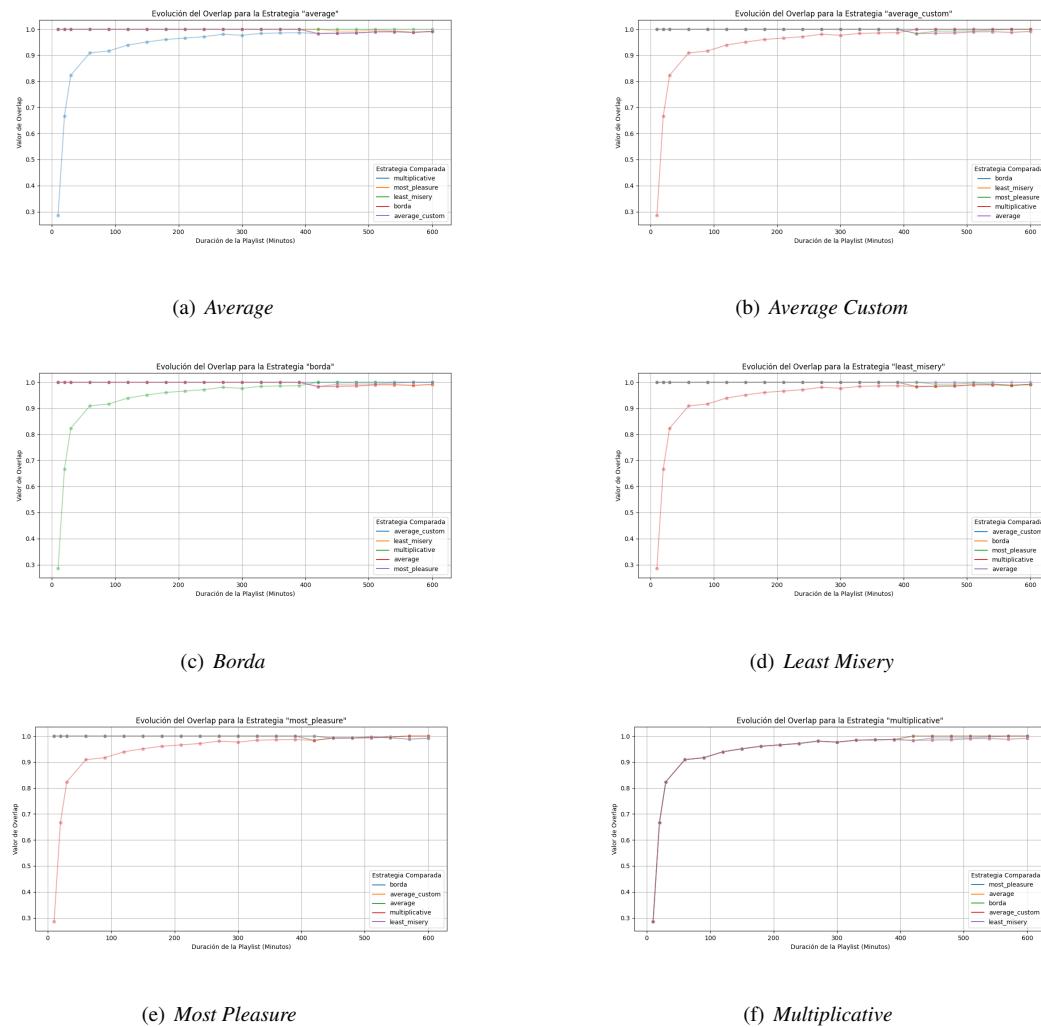


Figura 4.2: Gráficas de similitud entre estrategias para un grupo de 3 personas sin similitud en semillas

Las conclusiones que podemos sacar de las gráficas de la Figura 4.2 son las siguientes:

- Vemos que, parece que la ausencia de coincidencia en semillas de los usuarios hace que las estrategias se comporten de más uniforme, viendo que, ya para duraciones bajas la similitud es prácticamente total entre casi todas las estrategias.
- Es importante el 'casi todas' que hemos mencionado, ya que destaca que la estrategia *Multiplicative* se comporta de manera diferente al resto, y su similitud con el resto de estrategias es mucho menor que la similitud entre las demás estrategias. No obstante, acaba tendiendo a la máxima similitud con el aumento de la duración, como es de esperar.

Grupo de 2 usuarios sin similitud en semillas

Después de estas segundas observaciones, vamos a tratar de ver los resultados de otra pareja de usuarios, pero en este caso, sin coincidencia en semillas. Creemos que esto será bastante revelador ya que, con menos usuarios, hay menos canciones entre las que elegir para la estrategia, y por ello, mayor facilidad para que haya una similitud total entre las estrategias. En la Figura 4.3 se muestran los resultados obtenidos, mostraremos la gráfica de la estrategia *Multiplicative* frente al resto.

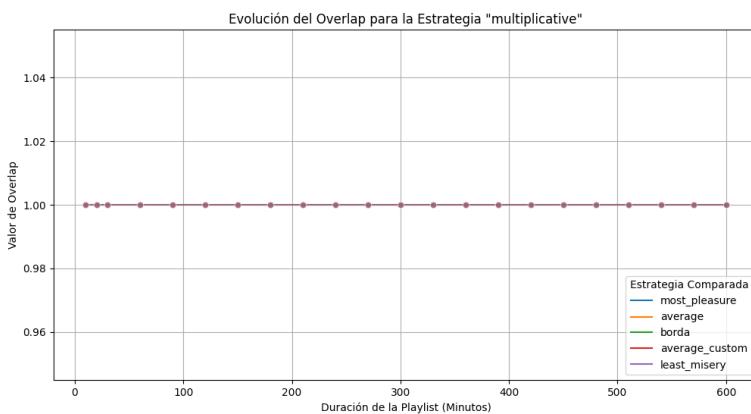


Figura 4.3: Gráficas de similitud entre estrategias para un grupo de 2 personas sin similitud en semillas

Como se puede observar en la Figura 4.3, todas las estrategias se comportan de la misma manera con una pareja que no tiene ninguna coincidencia en semillas. Esto nos hace observar que:

- La menor cantidad en un grupo de usuarios, y la baja coincidencia en sus gustos, contribuyen a que las estrategias de agregación dejen de aportar diferencias a la hora de generar recomendaciones.
- A pesar de que anteriormente habíamos observado que la estrategia *Multiplicative* se comportaba de manera diferente al resto, en este caso, no es así. Deberemos seguir observando el comportamiento de las estrategias en grupos de mayor tamaño para tratar de destacar algunas estrategias.

Grupos de 3, 4 y 5 usuarios con similitud en semillas

Vamos a observar resultados con situaciones algo más favorables para nuestro sistema de recomendación. En la Figura 4.4 se muestran los resultados obtenidos para grupos de 3, 4 y 5 usuarios, en los cuales, existía una coincidencia de una semilla de artista entre tan sólo dos de los usuarios del grupo. Mostraremos las gráficas de la estrategia *Average* frente al resto.

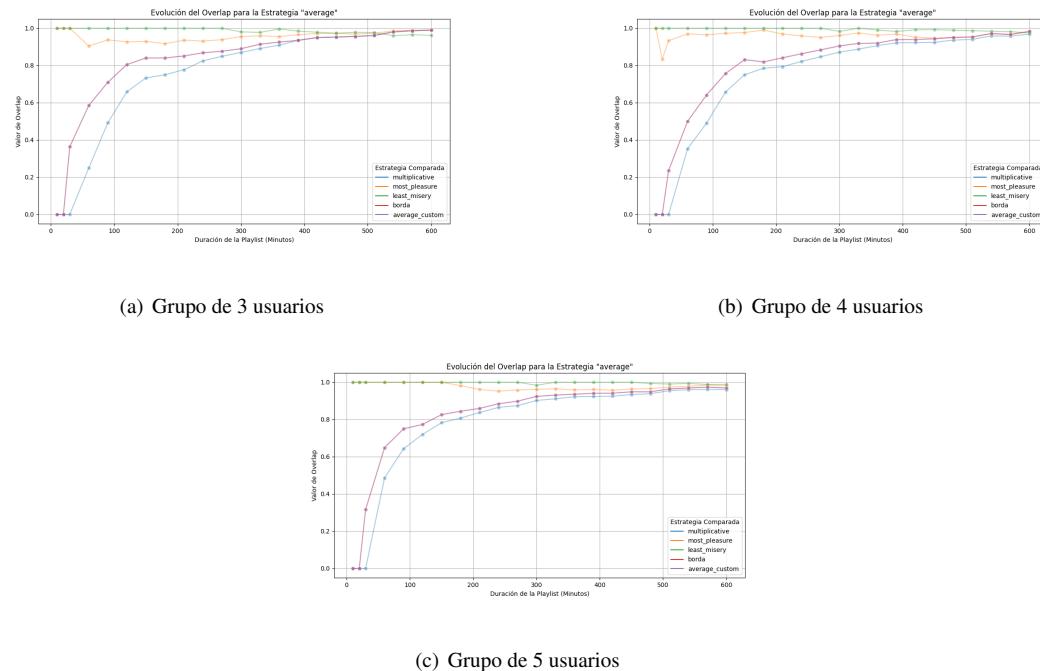


Figura 4.4: Gráficas de similitud entre estrategias para grupos de 3, 4 y 5 personas con similitud en semillas

Viendo las gráficas de la Figura 4.4, podemos destacar lo siguiente:

- Observamos que con grupos más grandes, se llega a la máxima similitud entre todas las estrategias para valores de duración de *playlist* mayores.
- Identificamos dos 'familias' de estrategias, cuyas estrategias se comportan de manera muy similar.
 - **Average, Least Misery** y **Most Pleasure** tienen una similitud cercana a la máxima entre sí desde el principio. No sabríamos destacar una estrategia que pueda ser más diferente que las otras dos.
 - **Average Custom, Borda** y **Multiplicative** tienen una similitud casi máxima entre sí desde el principio. Observamos que, *Borda* y *Average Custom* son idénticas en su comportamiento. Por ello destacaríamos *Multiplicative* como la estrategia que más se diferencia.

4.1.2. Decisiones tras los resultados

Tras haber tratado de explorar distintos escenarios de número de usuarios del grupo y similitud en semillas entre ellos, hemos podido ver indicios de algunas teorías:

- La similitud entre estrategias tiende a ser máxima con el aumento de la duración de la *playlist*. Esto es de esperar, ya que, a mayor duración, con un número fijo de canciones candidatas, la probabilidad de que exista similitud entre estrategias es mayor.
- Un número menor de usuarios en el grupo contribuye a que las estrategias de agregación dejen de aportar diferencias a la hora de generar recomendaciones.
- La coincidencia en semillas entre usuarios del grupo contribuye a que las estrategias sí que aporten diferencias a la hora de generar recomendaciones.
- Las estrategias probadas pueden agruparse en dos grupos, debido a la similitud en sus comportamientos con grupos mayores y con coincidencia de semillas.

Aunque tenemos claro que sería adecuado y deseable tener una muestra mayor de grupos, que aportase más variedad con: mayores números de usuarios y distintas similitudes en semillas; creemos que la información obtenida con nuestras pruebas es suficiente para poder tomar decisiones sobre qué estrategias de agregación vamos a comparar con usuarios en las pruebas con usuarios. Hemos decidido que vamos a centrar las comparaciones en las estrategias *Multiplicative* y *Least Misery*, cada una como representante de un grupo de estrategias:

- ***Multiplicative***: Hemos observado que esta estrategia se comporta de manera diferente a las otras dos estrategias de su grupo, que se comportaban de manera idéntica entre sí.
- ***Least Misery***: podríamos también haber escogido *Average* o *Most Pleasure* como representantes de su grupo, ya que ninguna destacaba por ser más diferente que las otras dos. No obstante, hemos decidido escoger *Least Misery* por ser la estrategia, con la idea más diferente detrás de ella, y que, por tanto, podría ser más interesante de comparar con usuarios.

4.2. Estudios con usuarios

Será la forma en la que trataremos de evaluar nuestro sistema de recomendación, elaborando encuestas para recoger las sensaciones y satisfacción que las *playlists* generadas por nuestro sistema de recomendación han causado en los usuarios. Compararemos estrategias de agregación para tratar de sacar conclusiones sobre qué estrategias tienen mejor acogida, trataremos de sacar una puntuación de usabilidad para nuestra aplicación con una encuesta *SUS* (System Usability Scale) y, por último, trataremos de sacar conclusiones sobre posibles nuevas funcionalidades que podrían ser interesantes para implementar en el futuro.

4.2.1. Gestión de usuarios

Como comentábamos en la Sección 3.2.2, Spotify nos permite dar de alta hasta 25 usuarios en el modo de desarrollo. Esto sin duda ha supuesto una limitación para nuestras pruebas con usuarios, puesto que nosotros teníamos que saber qué personas iban a probar nuestra aplicación para poder darlos de alta previamente. Por esto no hemos podido compartir nuestra aplicación y nuestras encues-

tas con un número elevado de personas, lo que sin duda hubiera sido lo ideal, para tener el mayor número de opiniones posibles, junto con mayor variedad de grupos de personas. Nuestra encuesta ha tenido numerosas preguntas, las cuales tratarán de obtener:

- Contexto sobre el grupo de personas que ha generado una *playlist* con nuestra aplicación.
- Comparación de estrategias de agregación.
- Puntuación de usabilidad de la aplicación.
- Posibles nuevas funcionalidades que podrían ser interesantes para implementar en el futuro.

4.2.2. Procedimiento de las pruebas con los usuarios

Las pruebas con usuarios se han llevado a cabo de manera presencial, reuniéndonos con los usuarios, dándolos de alta en nuestra aplicación para que pudieran utilizarla y explicándoles los pasos a seguir:

- En primer lugar, que todos hicieran login en nuestra aplicación con su cuenta de Spotify.
- Una vez todos aparecieran en la pantalla principal de la aplicación, les explicábamos que tenían que generar una *playlist* con nuestra aplicación, eligiendo la opción: *Comparar estrategias A y B*, e indicando una duración de 1 hora para la *playlist*.
- Una vez generadas las *playlist*, les pedíamos que revisaran las canciones que habían sido añadidas a las *playlists*, que se quedaran con la sensación que les transmitían, o si una le gustaba más que otra.
 - Es importante destacar que, en este paso, si nuestro sistema de recomendación no era capaz de obtener una *playlist* diferente para cada estrategia, solo se mostraría una en pantalla, y esta opción ha sido barajada en el formulario.
- Compartíamos con ellos el formulario vía *link*.
- Por último, les pedíamos que llenaran el formulario de manera totalmente sincera.

4.2.3. Contexto sobre el grupo de personas

En nuestra encuesta, asignamos un nombre identificativo a cada grupo, preguntamos de cuántas personas se componía, y si consideraban si el grupo era homogéneo o heterogéneo. De estas preguntas, obtenemos lo siguiente:

- **Número de personas en el grupo:** La mayoría de grupos han sido parejas, dejando otro grupo de 3 y de 4 personas.
- **Nombre para identificar el grupo:** Les pedíamos que pusieran los nombres de las personas que habían generado la *playlist* en orden alfabético.
- **Homogeneidad del grupo:** Las parejas han coincidido en su percepción de la homogeneidad de sus gustos. Pero en los grupos más numerosos, ha habido opiniones dispares.

Sobre la valoración de la homogeneidad en las parejas, sí que creemos que nos pueden servir para sacar algunas conclusiones sobre las teorías que empezamos a plantear en la Sección 4.1.2, como la

de que un bajo número de personas en el grupo, (como las parejas), y la baja similitud en los gustos, podía provocar que las estrategias de agregación no aportaran diferencias en las recomendaciones.

4.2.4. Comparación de estrategias

Para los grupos que obtuvieron dos *playlists* diferentes, les pedíamos que valoraran por separado las dos *playlists* y que eligieran cuál de las dos preferían. En el caso de que no obtuvieran dos *playlists* diferentes, les pedíamos que valoraran la *playlist* que les había aparecido en pantalla. De esta manera, obtenemos lo siguiente:

- **Comparación de estrategias:** ninguna de las dos estrategias ha sido claramente preferida por los usuarios. Tanto en parejas, como en grupos más grandes, la elección variaba.
- **Valoración de la *playlist*:** tanto comparando dos, como cuando solamente se obtenía una, las puntuaciones de que los usuarios daban se aproximan al 6,5 sobre 10.

4.2.5. Encuesta de usabilidad

Para evaluar la usabilidad de nuestra aplicación, hemos utilizado la encuesta SUS (System Usability Scale) [31]. Esta encuesta son 10 preguntas sobre usabilidad, que se responden con una escala de 5 puntos, y nos permite obtener una puntuación de usabilidad aplicando una fórmula sobre los valores de las respuestas. Los valores de las respuestas que hemos utilizado han sido las medias de las respuestas de los usuarios encuestados. La fórmula para obtener la puntuación de usabilidad se puede ver en la Ecuación 4.1 donde r_i es la puntuación en la pregunta i .

$$SUS = 2,5 \sum_{i=1}^{10} \begin{cases} r_i - 1 & \text{si } i \text{ es impar} \\ 5 - r_i & \text{si } i \text{ es par} \end{cases} \quad (4.1)$$

La puntuación de usabilidad que hemos obtenido ha sido de 68,75 sobre 100. Esta puntuación se considera aceptable, pero

4.2.6. Posible nueva funcionalidad en la aplicación

CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se expondrán las conclusiones que se han obtenido tras la realización de este trabajo y las últimas pruebas con usuarios, y se van a proponer posibles rumbos a seguir para trabajo futuro en el proyecto.

BIBLIOGRAFÍA

- [1] Google, “Flutter.” github.com/flutter/flutter, 2017.
- [2] S. AB, “Spotify.” <https://open.spotify.com/>, 2024.
- [3] S. AB, “Web api, spotify for developers.” <https://developer.spotify.com/documentation/web-api/>, 2024.
- [4] Google, “Toyota showcase.” <https://flutter.dev/showcase/toyota>, 2021.
- [5] Google, “Dart.” <https://github.com/dart-lang>, 2011.
- [6] Google, “Hot reload.” <https://docs.flutter.dev/tools/hot-reload>, 2023.
- [7] Google, “pub.dev.” <https://pub.dev>, 2023.
- [8] Google, “Dart overview.” <https://dart.dev/overview>, 2023.
- [9] Google, “Flutter install.” <https://docs.flutter.dev/get-started/install>, 2024.
- [10] Google, “Building user interfaces with flutter.” <https://docs.flutter.dev/ui>, 2023.
- [11] Meta, “React.” <https://react.dev>, 2024.
- [12] Wikipedia, “Spotify.” <https://es.wikipedia.org/wiki/Spotify>, 2024. Último acceso: 7 de febrero de 2024.
- [13] Marketing4eCommerce, “Historia de spotify: nacimiento y evolución.” <https://marketing4ecommerce.net/historia-de-spotify-del-lider-de-la-musica-en-streaming>, 2022. Último acceso: 7 de febrero de 2024.
- [14] deezerdevs, “Deezer api documentation.” <https://rapidapi.com/deezerdevs/api/deezer-1>, 2024.
- [15] A. Inc., “Apple music api.” <https://developer.apple.com/documentation/applemusicapi>, 2024.
- [16] S. Engineering, “Spotify engineering - spotify’s official technology blog.” <https://engineering.atspotify.com/>, 2024.
- [17] F. Ricci, L. Rokach, and B. Shapira, eds., *Recommender Systems Handbook*. Springer US, 2022.
- [18] F. Ricci, L. Rokach, and B. Shapira, “Recommender systems: introduction and challenges,” in *Recommender systems handbook*, pp. 1–34, Springer, 2022.
- [19] G. Adomavicius, K. Bauman, A. Tuzhilin, and M. Unger, “Context-aware recommender systems: From foundations to recent developments context-aware recommender systems,” in *Recommender Systems Handbook*, pp. 211–250, Springer, 2021.
- [20] M. Schedl, P. Knees, B. McFee, and D. Bogdanov, “Music recommendation systems: Techniques, use cases, and challenges,” in *Recommender Systems Handbook*, pp. 927–971, Springer, 2021.
- [21] J. Masthoff and A. Delić, “Group recommender systems: Beyond preference aggregation,” in *Recommender Systems Handbook*, pp. 381–420, Springer, 2022.

- [22] C. Senot, D. Kostadinov, M. Bouzid, J. Picault, A. Aghasaryan, and C. Bernier, “Analysis of strategies for building group profiles,” in *Proceedings of User Modeling, Adaptation, and Personalization*, (Berlin), pp. 40–51, Springer, 2010.
- [23] L. V. Tran, T.-A. N. Pham, Y. Tay, Y. Liu, G. Cong, and X. Li, “Interact and decide: Medley of sub-attention networks for effective group recommendation,” in *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, (New York), pp. 255–264, ACM, 2019.
- [24] Google, “Hive package.” <https://pub.dev/packages/hive>, 2024.
- [25] S. AB, “Authorization code with pkce flow.” <https://developer.spotify.com/documentation/web-api/tutorials/code-pkce-flow>, 2024.
- [26] S. AB, “Get current user’s profile.” <https://developer.spotify.com/documentation/web-api/reference/get-current-users-profile>, 2024.
- [27] S. AB, “Get user’s top items.” <https://developer.spotify.com/documentation/web-api/reference/get-users-top-artists-and-tracks>, 2024.
- [28] S. AB, “Get recommendations.” <https://developer.spotify.com/documentation/web-api/reference/get-recommendations>, 2024.
- [29] S. AB, “Create playlist.” <https://developer.spotify.com/documentation/web-api/reference/create-playlist>, 2024.
- [30] S. AB, “Add items to playlist.” <https://developer.spotify.com/documentation/web-api/reference/add-tracks-to-playlist>, 2024.
- [31] J. Brooke, “System usability scale (sus),” (2011): *Measuring Usability with the System Usability Scale (SUS)*, 1986.



Universidad Autónoma
de Madrid