AllMusic: Mashup de Eventos Musicales

Por

María Begoña Martínez Martín

Camila Belén Pérez Alániz

Paula Piñuela Monjas

Trabajo de fin de grado del Grado en Ingeniería Informática
FACULTAD DE INFORMÁTICA

Director: Pablo Manuel Rabanal Basalo

MADRID, 2020–2021

AllMusic: Musical Events' Mashup

By

María Begoña Martínez Martín

Camila Belén Pérez Alániz

Paula Piñuela Monjas

UNIVERSIDAD COMPLUTENSE
MADRID

Final degree project in Computer Science Degree
COMPUTER SCIENCE FACULTY

Director: Pablo Manuel Rabanal Basalo

MADRID, 2020–2021

Autorización de difusión

Los abajo firmantes, matriculados en el Grado en Ingeniería del Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo Fin de Grado: "AllMusic: Mashup de Eventos Musicales", realizado durante el curso académico 2020-2021 bajo la dirección de Pablo Manuel Rabanal Basalo en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

María Begoña Martínez Martín, Camila Belén Pérez Alániz y Paula Piñuela Monjas
15 de Junio de 2021

Dedicatorias

A mi madre, por nuestro día a día, por su apoyo incondicional en cada paso de mi vida.
A mi padre, por ser un pilar de seguridad y calma.
A Alberto, por ser ejemplo de esfuerzo y perseverancia.
De todo corazón, muchas gracias.

- Mª Begoña Martínez Martín

A mi familia, por animarme a emprender este proyecto y no darme por vencida.
A mis amigos, por apoyarme cuando más los necesitaba.
A aquellas personas que se han ido de mi vida por enseñarme a ser más fuerte.
Muchas gracias, sin vosotros no estaría donde estoy ahora mismo.

- Camila Belén Pérez Alaniz

A mi padre, a mi madre y a mi hermano, que han formado parte de todas las etapas de mi vida, siendo un ejemplo de superación y contando siempre con su amor y apoyo.
A mis amigos y compañeros, por compartir sus conocimientos y ser una fuente de inspiración.
A todos aquellos que me han ayudado en esta etapa y han estado ahí cuando lo he necesitado.
Muchas gracias.

- Paula Piñuela Monjas

Agradecimientos

A Pablo Rabanal por guiarnos en todo este proceso, dándonos la libertad para llevar a cabo nuestras ideas, siempre bajo su supervisión. También agradecer sus constantes correcciones que nos ha permitido desarrollar el proyecto siguiendo el cauce correcto.

Muchas gracias por el trabajo realizado y la ayuda aportada en nuestro Trabajo de Fin de Grado.

Resumen

Con el paso de los años, la tecnología avanza de manera exponencial, teniendo un impacto sustancial en la sociedad y formando parte de ella. La industria musical es uno de los ejemplos de este avance social al desplazar su modelo de negocio a un entorno digital, permitiendo nuevos métodos de marketing, publicación y consumo de música a través de múltiples plataformas.

Entre las aplicaciones más demandadas por la población se encuentran aquellas que permiten reproducir música desde cualquier lugar, conocer información sobre los artistas más escuchados en el mundo o comprar entradas para eventos musicales como conciertos o festivales. Sin embargo, el principal problema referente a este tipo de plataformas es su especialización en un tema concreto. Al no integrar todo el contenido necesario, relacionado con artistas, provocan al usuario la necesidad de interactuar con una colección de servicios totalmente diferentes.

La plataforma implementada en este trabajo resuelve este problema integrando todos los servicios más populares del contexto musical digital. De esta forma, el usuario puede acceder a todos ellos de forma ágil y sencilla. Esto se consigue gracias a la conexión con aquellas plataformas que ya cuentan en sus bases de datos con toda la información necesaria y que están expuestas a acceso programático. Por lo tanto, cualquier usuario puede conocer nuevos artistas, escuchar sus mejores canciones y acceder a la compra de entradas para asistir a sus próximos conciertos a través de una única aplicación.

Palabras clave

Mashup, web híbrida, API, Django, React, Redux, Redis, MongoDB, Heroku, frontend, backend.

Abstract

Over the years, technology advances exponentially, having a substantial impact on society and becoming part of it. The music industry is one of the examples of this social progress by shifting its business model to a digital environment, allowing new methods of marketing, publishing and consumption of music through multiple platforms.

Among the applications most in demand by the population are those that allow to play music from anywhere, to know information about the most listened artists in the world or to buy tickets for musical events such as concerts or festivals. However, the main problem with this type of platform is its specialization in a specific topic. By not integrating all the necessary content related to artists, they provoke the user the need to interact with a collection of completely different services.

The platform implemented in this work solves this problem by integrating all the most popular services in the digital music context. In this way, the user can access all of them in an agile and simple way. This is achieved thanks to the connection with those platforms that already have in their databases all the necessary information and that are exposed to programmatic access. Therefore, any user will be able to learn about new artists, listen to their best songs and buy tickets to attend their upcoming concerts through a single application.

Keywords

Mashup, hybrid web, API, Django, React, Redux, Redis, MongoDB, Heroku, frontend, backend.

Índice general

Resumen VII

Abstract IX

Página

1. Introducción **1**

1.1. Contexto 1

1.2. Motivación 2

1.3. Objetivos 3

1.4. Plan de Trabajo 4

1.5. Estructura del documento 7

2. Introduction **9**

2.1. Context 9

2.2. Motivation 10

2.3. Objectives 11

2.4. Work Plan 11

2.5. Document structure 14

3. Estado del arte **17**

3.1. Plataformas y aplicaciones musicales 17

3.1.1. Historia de las aplicaciones de música 17
3.1.2. Tipos de aplicaciones de música 18
3.1.3. Análisis de aplicaciones y plataformas de música 19
3.2. APIs 21
3.2.1. Estudio de las APIs 23

4. Requisitos 27

4.1. Descripción General 27
4.1.1. Perspectiva del proyecto 27
4.1.2. Funciones del proyecto 28
4.1.3. Requisitos Futuros 29
4.2. Requisitos Específicos 30
4.2.1. Requisitos Funcionales 30
4.2.2. Requisitos No Funcionales 35

5. Arquitectura software 37

5.1. Plataforma web - React 39
5.1.1. Redux 40
5.2. Integración con Django 41
5.3. Bases de Datos - MongoDB 42
5.3.1. MongoDB Atlas 42
5.4. Base de Datos Caché - Redis 43
5.4.1. Redis Enterprise Cloud 43
5.5. Despliegue con contenedores Docker 44
5.5.1. Despliegue del proyecto 44

5.6. Despliegue en Heroku 45
5.7. Evaluación del Sistema 45

6. Desarrollo de la aplicación 47

6.1. Frontend 47
6.1.1. Enrutamiento del sistema 47
6.1.2. Gestión del flujo de datos de la aplicación 48
6.1.3. Desarrollo de la interfaz gráfica 50
6.1.4. Cambio de idiomas 51
6.1.5. Implementación del mapa 52
6.2. Backend 52
6.2.1. Envío de correos electrónicos desde el backend 52
6.2.2. Validación del correo electrónico 53
6.2.3. Cifrado de contraseñas 53
6.2.4. Base de datos de usuarios 54
6.2.5. Cruce de información de entidades con diferentes identificadores en cada servicio 54
6.2.6. Conexión entre Django y MongoDB 55
6.2.7. Interconexión con servicios externos 55
6.3. Obtención de datos 57
6.3.1. Elección de APIs para la aplicación 57
6.4. Despliegue del proyecto 60
6.4.1. Gunicorn 61
6.4.2. WhiteNoise 61

7. Conclusión y líneas futuras 63

7.1. Conclusión 63

7.2. Trabajo futuro 64

8. Conclusion and future work **65**

8.1. Conclusion 65

8.2. Future work 66

A. Código fuente **81**

A.1. Backend 81

A.1.1. Uso de la librería de Python urllib para la obtención de datos . . 81

A.1.2. Ejemplo del procesamiento y la limpieza de los datos 81

A.1.3. Ejemplo de uso del paquete *EmailMultiAlternatives* para la gestión de correos electrónicos 82

A.1.4. Uso del paquete *re* para la validación de correos electrónicos . . . 83

A.1.5. Uso del paquete *crypt* para el cifrado de contraseñas 84

A.1.6. Uso del paquete *crypt* para la comprobación de contraseñas . . . 84

A.2. Frontend 85

A.2.1. Función action de la página de inicio 85

A.2.2. Función reducer de la página de inicio 86

Capítulo 1

Introducción

1.1. Contexto

La música siempre ha sido una parte fundamental en la vida de las personas. Cuando antiguamente se escuchaba en vinilos, cintas o discos, la única forma de acceder a información relacionada con artistas o grupos era a través de los medios de comunicación. Actualmente, a consecuencia de la revolución tecnológica en la que vivimos, ocurre al contrario, se puede acceder a contenido musical a través de cualquier dispositivo. Este avance tecnológico, que afecta prácticamente a todos los aspectos de la vida de las personas, ha provocado uno de los mayores cambios en la industria discográfica. En la última década, la digitalización ha permitido reproducir música a través de internet, con dispositivos electrónicos, vía móvil o web [1]. De esta forma, se reducen los costes de la copia y difusión de los discos permitiendo al usuario beneficiarse de todo un amplio catálogo musical de forma cómoda y con bajo coste.

Esta transformación también afecta en gran medida a la forma de seguir a los artistas o grupos musicales. Hace cuatro décadas, la forma de enterarte de un nuevo concierto o del lanzamiento de un disco era a través de medios como la radio, la televisión o el periódico. Es decir, el conocimiento de la información se encontraba condicionado por los medios de comunicación y sus horarios. Hoy en día, tanto el escuchar música como el acceder a ese tipo de información se realiza bajo demanda, es decir, cuando el usuario quiere. Comprender el modo en el que la tecnología ha cambiado la industria musical es vital para entender el marco en el que se sitúa este proyecto.

Uno de los servicios consecuentes a este cambio son plataformas que suplen la necesidad implícita que tenían las personas de poder escuchar música desde cualquier dispositivo y en cualquier momento. Estas aplicaciones no sólo han cambiado el modelo de negocio, sino también la forma de interactuar con la música y los artistas. Se ha convertido en algo directo, el usuario tiene la posibilidad de acceder a contenido musical y tener información

actualizada cuando desea [2].

Actualmente existen plataformas muy populares que son usadas de forma diaria por millones de personas con este fin. Estas plataformas proporcionan información y servicios muy relevantes en el contexto de la experiencia musical. Debido a su actividad, hace que generen una gran cantidad de datos, muy valiosos. Ejemplo de dichos servicios son: *Spotify* [3], en el ámbito de la reproducción de música por vía *streaming*, *Ticketmaster* [4], en el contexto de la compra o venta de entradas de conciertos o *YouTube* [5] en el campo de reproducción de vídeos musicales.

1.2. Motivación

Debido a que ninguna de estas webs o aplicaciones móviles ofrecen información completa, el usuario se ve obligado a acceder a varias de ellas para poder obtener toda la información. Por ejemplo, si el usuario desea comprar entradas para un concierto y reproducir las canciones de su artista favorito, debe acceder a diferentes servicios. Por tanto, tal cantidad de plataformas, tan específicas y para diferentes usos, crean una problemática para el usuario, generando una pérdida de tiempo y una complejidad innecesaria al tener que acceder a múltiples servicios para conseguir la información íntegra, en este caso, relacionada con un artista.

Este tipo de situación no solo ocurre en el ámbito musical, sino que también es muy común en otros campos. Un claro ejemplo es el caso de los viajes, donde existen aplicaciones que muestran información sobre alojamiento, transporte y puntos de interés. Al final, la vida de las personas queda condicionada por tener que buscar aquello que queremos en varias aplicaciones o servicios web, a pesar de que el contenido esté relacionado.

La problemática introducida genera, no sólo el problema de la inversión que requiere de tiempo, sino también la necesidad de estar registrado en todas las plataformas y de compartir tus datos o información personal con múltiples aplicaciones. Esta cuestión puede derivar en la falta de seguridad en la protección de tus datos, al ser más probable una filtración en alguna de las numerosas plataformas que pueden contar con información sensible como son los datos bancarios de tarjetas de crédito.

Con el fin de solucionar este problema, existen aplicaciones que consumen datos aportados por las plataformas mencionadas anteriormente. Este tipo de servicios se dedican a gestionar, combinar y mostrar la información para facilitar el consumo del usuario. Ejemplo de estas aplicaciones son: *Songkick* [6], que muestra información sobre festivales o conciertos o *Last.fm* [7], que combina información sobre artistas con conciertos.

El consumo de tal cantidad de datos de estas aplicaciones requiere una interconexión entre las plataformas y la propia aplicación. Este tipo de conexión y de consumo de datos genera un problema que se debe gestionar. A nivel de eficiencia, el tiempo de respuesta de las fuentes que ofrecen información de las plataformas es elevado. Lo que provoca una pérdida de rendimiento en la aplicación por un motivo externo a la misma. Además,

este tipo de aplicaciones muestran información sobre la misma entidad, pero obtenidas a través de diferentes fuentes. Esto crea un problema de congruencia a la hora de relacionar y gestionar datos del mismo activo. Por este motivo, es necesaria una forma de relación entre todas las fuentes de información.

Al mismo tiempo, con la digitalización se ha producido un gran avance en el diseño de interfaces de usuario. Esto ha permitido poner foco en lo que resulta más atractivo para el usuario. Un estudio [8] que evalúa qué características son las más valoradas por los consumidores ha establecido unos parámetros deseados en este tipo de aplicación; entre estas variables se incluyen la facilidad de uso y el valor tanto monetario como emocional o social.

1.3. Objetivos

Por todo lo expuesto anteriormente, la solución ofrecida en este proyecto es un agregador de datos de aplicaciones y plataformas que unifica toda la información relacionada con artistas y la muestra de forma atractiva e intuitiva para el usuario. Además, tiene en cuenta toda la problemática mencionada anteriormente y expone una arquitectura específica para este tipo de servicios.

Los objetivos definidos al inicio del proyecto estaban relacionados con aplicar lo aprendido a lo largo de nuestros estudios, poner en práctica los conocimientos adquiridos sobre programación, bases de datos y aplicaciones web. Además, queríamos que el diseño arquitectural de la aplicación cumpliera con los requisitos estudiados, entre otros, eficiencia, portabilidad, escalabilidad y usabilidad, para ello, investigamos y evaluamos los *frameworks* que cumplieran las especificaciones del proyecto a desarrollar. En base a ese estudio, decidimos aprender e implementar las siguientes tecnologías: *React* [9], *Django* [10], *Docker* [11], *Heroku* [12], *MongoDB* [13] y *Redis* [14], explicadas de forma más extensa en el capítulo de arquitectura 5. Otro de los objetivos fijados para el buen desarrollo de la aplicación fue el exhaustivo estudio de todas las APIs relacionadas con el panorama musical, para poder decidir qué datos consumiríamos de cada una y de qué forma conectaríamos toda la información. A nivel de desarrollo de funcionalidades del servicio, se determinó el objetivo principal, permitir al usuario acceder a contenido de diversa índole, desde información de conciertos, listados con recomendaciones de artistas, hasta videoclips populares por países de forma atractiva y cómoda. Además, en base al artículo mencionado con anterioridad [8], el valor social es muy relevante para los consumidores. Se valora el poder crear una identidad en la página y compartir gustos con el resto de usuarios registrados, para lo que se creará una pequeña red social. Otro de los valores mencionados en dicho artículo, es la utilidad que ofrece la página web. Actualmente, la planificación cobra un papel fundamental en la vida de las personas [15], y con la gran oferta cultural existente, poder gestionar el tiempo y la agenda de forma eficiente es fundamental para los consumidores. En base a estas dos premisas, otro de los objetivos a satisfacer es el hecho de crear una forma de visualizar los conciertos, tanto a través de un mapa que mostrará de forma atractiva eventos basados en una localización, como a nivel personal mostrando los eventos a los que asistirá el usuario en su propio calendario.

1.4. Plan de Trabajo

El presente plan de trabajo pretende abarcar toda la información de cómo debe ser realizado el proyecto. Este plan incluye los objetivos a lograr, los procesos a integrar y, con mayor importancia, los tiempos de entrega que se han de cumplir. Estos plazos se deben tener en cuenta a lo largo de todo el proceso, incluso si los objetivos cambian, los tiempos establecidos no deben ser modificados. Por tanto, es una herramienta que permite generar estrategias para completar con éxito el proyecto a través del trabajo en equipo.

Como se ha mencionado anteriormente, la finalidad del proyecto es crear una aplicación web híbrida o mashup que mediante la integración de componentes de otros servicios genere uno nuevo con información originaria de estos. De este modo, el usuario es capaz de conseguir todo lo que necesita del ámbito musical en una única página web. Algunos de los aspectos que se desean conseguir van desde tener un listado con artistas seleccionados, poder escuchar su contenido musical, conocer los eventos que se celebrarán cerca de su lugar de residencia, hasta generar una red de amigos.

Teniendo una visión definida del proyecto se especifican una serie de objetivos concretos a conseguir, tanto técnicos como funcionales. En el ámbito técnico, el objetivo principal se basa en el diseño de una arquitectura que gestione de forma eficiente la comunicación con servicios externos teniendo en cuenta cualidades como la usabilidad, escalabilidad, portabilidad o fiabilidad. Con el fin de cumplir dichos requisitos, se establece la separación entre la lógica de la aplicación y la interfaz de usuario; lo que permite gestionar de manera independiente cada parte. También son especificados los servicios básicos de los que se obtendrá la información, entre ellos, *Spotify* [3], *Youtube* [5] o *Google* [16], concluyendo así la previsión técnica de objetivos. Con respecto a la parte funcional entre los objetivos acordados están los basados en mostrar aquellos artistas con mayor índice de escuchas en las principales aplicaciones de música, así como sus canciones más reproducidas, videoclips e información sobre sus redes sociales. Por otro lado, es considerado primordial gestionar los eventos que se realizarán en una ubicación próxima a la del usuario, pudiendo, tanto acceder de forma sencilla a la compra de entradas como contando con la posibilidad de almacenar esta información en la cuenta personal del usuario del calendario de *Google* [16]. Otro objetivo se basa en la posibilidad de mostrar los eventos en un mapa, facilitando así una vista rápida de su localización.

A continuación, se divide el proyecto en trabajos de modo que cada miembro del equipo tenga tareas diferentes asignadas, ya que la centralización de todos los esfuerzos en una única tarea retrasaría el proceso de desarrollo. En el siguiente diagrama de Gantt 1.1 exponemos el tiempo de dedicación estimado para las actividades a lo largo del proyecto.

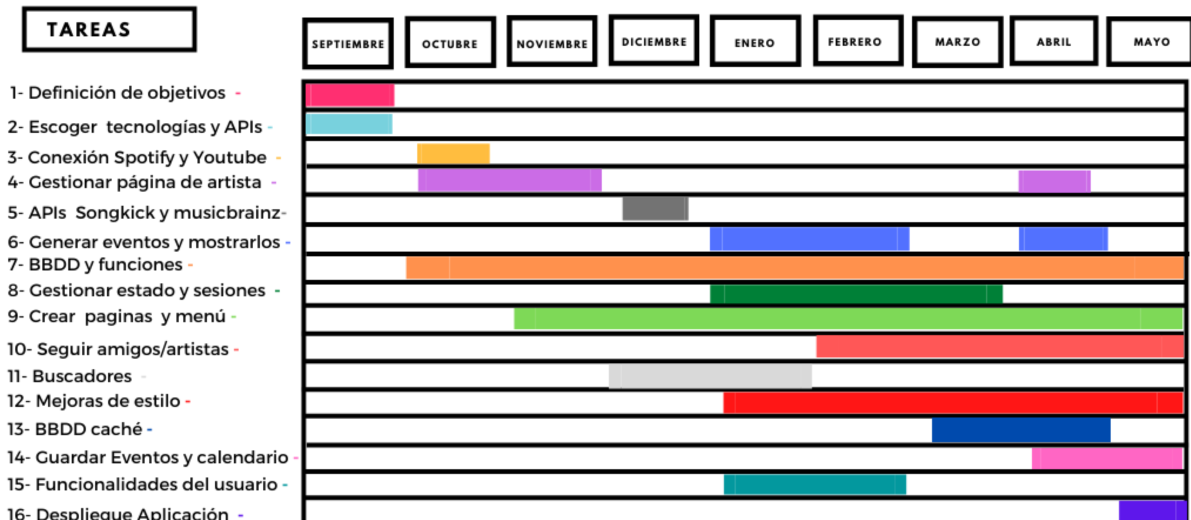

Figura 1.1: Diagrama de Gantt

Las tareas definidas en este diagrama son las siguientes:

1. **Definición de objetivos**: Definición de objetivos a considerar como base del proyecto.

2. **Escoger el entorno y APIs**: Realización de un estudio y elección de las diferentes tecnologías para abordar el desarrollo del backend y el frontend en el proyecto, exponiendo los beneficios e inconvenientes de cada una. También se estudia qué APIs integrar en la aplicación.

3. **Conexión *Spotify* y *Youtube***: Investigación y realización de la conexión y obtención de datos haciendo uso de las APIs de *Spotify* [3] y *Youtube* [5].

4. **Gestionar página de artista**: Selección de los datos a mostrar sobre el artista y diseño de la entrada donde se muestra la misma.

5. **APIs Songkick y Musicbrainz**: Estudio del modo de relacionar la información devuelta por *Spotify* [3] con *Songkick* [6] y *Musicbrainz* [17] ,las cuales ofrecían mayor diversidad de contenido.

6. **Generar eventos y mostrarlos**: Obtención de datos sobre eventos e ilustrarlos en la página principal de la web.

7. **BBDD y funciones**: Integración de un servidor *MongoDB* [13].

8. **Gestionar estado y sesiones**: Investigación del almacenamiento de estados y las sesiones de usuarios.

9. **Crear páginas y menú**: Creación de nuevas páginas con diversos contenidos y un menú para gestionarlas.

10. **Seguir amigos/artistas**: Implementación de la funcionalidad de añadir artistas y amigos al perfil del usuario, para acceder a estos de forma rápida.

11. **Buscadores**: Desarrollo de buscadores tanto de artistas como de eventos para poder filtrar aquellos que se deseen.

12. **Mejoras de estilo**: Realización de mejoras estéticas mediante CSS y componentes específicos de diseño.

13. **BBDD caché**: Inclusión de una base de datos caché para agilizar la carga de la información.

14. **Guardar Eventos y calendario**: Creación de la funcionalidad de guardar eventos y mostrarlos en un calendario.

15. **Funcionalidades de usuario**: Creación del perfil de usuario, el registro de estos, la lista de artistas seguidos y la de amigos.

16. **Despliegue Aplicación**: Despliegue de la aplicación web en un servidor.

Tomando de apoyo el plan de trabajo previamente definido y el diagrama de tiempos establecido, el modo de actuación se basará en la realización de un seguimiento quincenal como forma de cerciorar que se llevaba a cabo una correcta ejecución del plan de trabajo original, para este fin se pondrán en conjunto las tareas llevadas a cabo con éxito entre las establecidas, los inconvenientes encontrados,si los había, y de ser así,el modo de solventarlos, en caso de ser posible. Además se considerará la posible necesidad de realizar cambios en los objetivos para adaptarlos a los recursos disponibles. Por último, serán comentados, en el caso de existir, los aspectos pendientes. Tras ello se se propondrán nuevas tareas y objetivos.

Cada reunión estaba regida por los pasos indicados a continuación:

- **Revisión proyecto**: Como comienzo cada miembro del equipo informará al resto de componentes del estado de su trabajo en dicho momento, las tareas concluidas, las problemáticas encontradas y la forma en que estas han sido resueltas, en caso de ser posible. También se mencionará si existe la necesidad de realizar cambios importantes en alguno de los aspectos definidos en el plan de proyecto original, así como indicar el motivo que origina dicho cambio.

- **Definición de objetivos**: Tras concluir la revisión del estado del proyecto, se comenzará a definir los siguientes objetivos a conseguir para la próxima revisión, se pondrán en común los posibles nuevos casos de uso, que no estuviera barajado en la propuesta original. Será necesario exponer los beneficios que se obtendrían (técnicos o funcionales) al aceptar dicho caso de uso. Las modificaciones podrán consistir en cambiar la API desde la cual se obtienen datos, un cambio en los datos a obtener o el modo en que se expondrán estos al usuario.

- **Localización de recursos**: Tras decidir las tareas a realizar se pasará a poner en común los recursos disponibles y se contemplará la posibilidad de llevarlas a cabo con dichos recursos. En caso de no ser posible, se explorará en conjunto los posibles

recursos a integrar para la correcta realización del cometido. Esto será hecho de manera general ya que la persona adjudicada para dicha tarea se encargará finalmente de decidir qué recurso se ajusta mejor a lo necesitado, siendo fundamental que sean tenidas en cuenta las prestaciones ofrecidas por cada uno, la agilidad en la carga de datos, el modo en que se ofrecen los datos y la dificultad de uso.

- **División de tareas**: Por último, se repartirán dichas tareas considerando la carga de trabajo, la dificultad de desarrollo y los conocimientos previos en el tema. Se tendrá presente el tiempo a dedicar a cada tarea, consiguiendo así, un reparto equilibrado de las mismas.
- **Organización de tiempos**: Como punto final de cada reunión, se acordará conjuntamente la fecha de la siguiente reunión de seguimiento y, por tanto, el plazo para la realización de los quehaceres.

1.5. Estructura del documento

La memoria se divide en los siguientes apartados:

En el capítulo 1, se ofrece una descripción del proyecto donde, además de hacer un breve resumen, se explican las motivaciones que han existido para su realización, qué objetivos se quieren conseguir y el plan de trabajo que se ha seguido. En el siguiente capítulo 3 se encuentra la sección de "estado del arte", donde se cuenta de manera breve el recorrido de las aplicaciones de música a lo largo de la historia y se explica detalladamente en qué estado se encuentra actualmente la tecnología que está directa o indirectamente relacionada con el proyecto. En el capítulo 4, se encuentra la "definición de requisitos", donde se explican todos los servicios o funcionalidades que el proyecto debe ofrecer para cumplir con lo esperado del mismo.

En el capítulo de "arquitectura" se describen las tecnologías empleadas en el desarrollo del proyecto. Y en el último capítulo, se encuentra el apartado de "desarrollo" que contiene toda la documentación del proceso que se ha seguido para llevar a cabo el desarrollo de la aplicación, así como la información de los elementos que se han usado, los problemas encontrados y las soluciones propuestas. Para finalizar, en el documento se añade una sección de "conclusiones y líneas futuras" donde se explica el conocimiento que hemos adquirido, las conclusiones a las que hemos llegado y la explicación de todo aquello que puede resultar interesante incluir en versiones futuras de la aplicación.

Capítulo 2

Introduction

2.1. Context

Music has always been a fundamental part of people's lives. Historically tt used to be listened to through vinyls, tapes or records, the only way to access information related to artists or groups was through the media. Nowadays, as a result of the technological revolution in which we live in, the opposite occurs, music content can be accessed through virtually any device. This technological advance, which affects practically all aspects of people's lives, has brought about one of the biggest changes in the recording industry. In the last decade [1], digitization has made possible to play music over the Internet, with electronic devices, via mobile or web. In this way, the costs of copying and distribution of records are reduced, allowing the user to benefit from a wide range of music catalogs in a convenient and low-cost way.

This transformation also greatly affects the way we follow artists or musical groups. Four decades ago, find out about a new concert or the release of a record was through media such as radio, television or newspapers. That is to say, the knowledge of the information was conditioned by the media and their schedules. Today, both listening to music and accessing this type of information is done on demand, that is, when the user wants it. Understanding the way in which technology has changed the music industry is vital to understand the framework in which this project is situated.

One of the services resulting from this change are platforms that meet people's implicit need to be able to listen to music from any device at any time. These applications have not only changed the business model, but also the way of interacting with music and artists. It has become something direct, the user has the possibility to access music content and have updated information whenever he/she wants [2].

Currently there are very popular platforms that are used on a daily by millions of people for this purpose. These platforms provide very relevant information and services in the context of the music experience. Due to their activity, they generate a large amount

of valuable data. Examples of these services are: *Spotify* [3], in the field of music playback via streaming, *Ticketmaster* [4], in the context of the purchase or sale of concert tickets or *YouTube* [5] in the field of music video playback.

2.2. Motivation

Since none of these websites or mobile applications offer complete information, the user is forced to access several of them in order to obtain all the information. For example, if the user wants to buy tickets for a concert and play the songs of their favorite artist, he must access different services. Therefore, such a large number of platforms, so specific and for different uses, creates a problem for the user, generating a waste of time and unnecessary complexity by having to access multiple services to get the full information, in this case, related to an artist.

This type of situation does not only occur in the music field, but it is also very common in other fields. A clear example is the case of travel, where there are applications that display information about accommodation, transportation, and other points of interest. At the end, people's lives are conditioned by having to look for what they want in several applications or web services, even though the content is related.

The introduced problem generates not only the time-consuming investment problem, but also the need to be registered on all platforms and to share your data or personal information with multiple applications. This issue can lead to a lack of security in the protection of your data, as it is more likely to be leaked on one of the many platforms that may have sensitive information such as bank or credit card details.

In order to solve this problem, there are applications that consume input data by the platforms mentioned above. These types of services are dedicated to manage, combine and display the information to facilitate the user's consumption. Examples of these applications are: *Songkick*, which shows information about festivals or concerts or *Last.fm*, which combines information about artists with concerts.

The consumption of such a large amount of data from these applications requires an interconnection between the platforms and the application itself. This type of connection and data consumption creates a problem that must be managed. In terms of efficiency, the response time of the sources that provide information from the platforms is high. This causes a loss of performance in the application for a reason external to it. Furthermore, this type of applications show information about the same entity, but obtained through different sources. This creates a problem of congruence when relating and managing data from the same asset. For this reason, a form of relationship between all sources of information is necessary.

At the same time, digitization has led to a breakthrough in user interface design. This has made it possible to focus on what is most appealing to the user. A study [8] assessing which features are most valued by consumers has established desired parameters in this

type of application; these variables include ease of use and both monetary and emotional or social value.

2.3. Objectives

For all of the above, the solution offered in this project is an application and platform data aggregator that unifies all the information related to artists and displays it in an attractive and intuitive way for the user. In addition it takes into account all the issues mentioned before and exposes an architecture specific for this type of services.

The objectives defined at the beginning of the project were related to apply what we had learned throughout our studies, to put into practice the knowledge acquired about programming, databases and web applications. In addition, we wanted the architectural design of the application to meet the requirements studied, among others, efficiency, portability, scalability and usability, for this reason, we investigated and evaluated the *frameworks* that met the specifications of the project to be developed. Based on this study, we decided to learn and implement the following technologies: *React* [9], *Django* [10], *Docker* [11], *Heroku* [12], *MongoDB* [13] y *Redis* [14], explained more extensively in the architecture chapter 5. Another of the objectives set for the proper development of the application was the exhaustive study of all the API's related to the music scene, in order to decide what data we would consume from each one, and in what way we would connect all the information. At the level of development of service functionalities, the main objective was determined, to allow the user to access content of various kinds, from concert information, listings with artist recommendations, to popular video clips by countries in an attractive and comfortable way. Furthermore, based on the aforementioned article [8], the social value is very relevant to consumers. Being able to create an identity on the page and share likes with the rest of the registered users, we will create a small social network. Another of the values mentioned in this article is the utility offered by the website. Nowadays, planning plays a fundamental role in people's lives [15], and with all the great existing cultural offers, being able to manage time and efficiently scheduling is critical for consumers. Based on these two premises, another of the objectives to satisfy is the fact of creating a way to visualize concerts, both through a map that will attractively show events based on a location, and at a personal level showing the events that the user will attend in his own calendar.

2.4. Work Plan

This work plan aims to cover all the information on how the project should be completed. This plan includes the objectives to be achieved, the processes to be integrated and, most importantly, the delivery times to be met. These deadlines must be taken into account throughout the entire process, even if the objectives change, the established times should not be modified. Therefore, it is a tool that allows the generation of strategies to successfully complete the project through working in a team.

As mentioned above, the purpose of the project is to create a hybrid web application or mashup that through the integration of components of other services, creates a new application that generates all the requested data. In this way, the user is able to get everything they need in the musical field on a single web page. Some aspects that we would like to achieve range from having a list with selected artists, being able to listen to its musical content, knowing the events that will be held near one's place of residence, and generating a network of friends.

Having a defined vision of the project, a series of technical and functional requirements were determined for the project's success.

In the technical field, the main objective is based on the design of an architecture to efficiently manage communication with external services taking into account qualities such as functionality, scalability, portability or reliability. In order to meet these requirements, an establishment of separation between the application logic and the user interface has to be done; allowing each part to be managed independently. The basic services from which the information will be obtained are also specified, including, *Spotify* [3], *Youtube* [5] or *Google* [16], thus concluding the technical forecast of the objectives. Regarding the functional part, the following closed objectives aim to show those artists with the highest rate of listeners in the main applications of music, as well as their videos and information on their social networks. On the other hand, it is considered essential to manage the events that will take place in a location close to the user, being able, in addition, to access through these the purchase of entry tickets, with the possibility of storing this information in the personal account of the *Google* [16] calendar user. Another objective is based on the possibility of showing events on a map, thus facilitating a quick view of your location. The project is then divided into jobs so that each team member has different tasks assigned to them, since the centralization of all efforts in one single task would delay the development process. In the following Grantt diagram 2.1 we present the estimated time for dedication of the activities throughout the project.

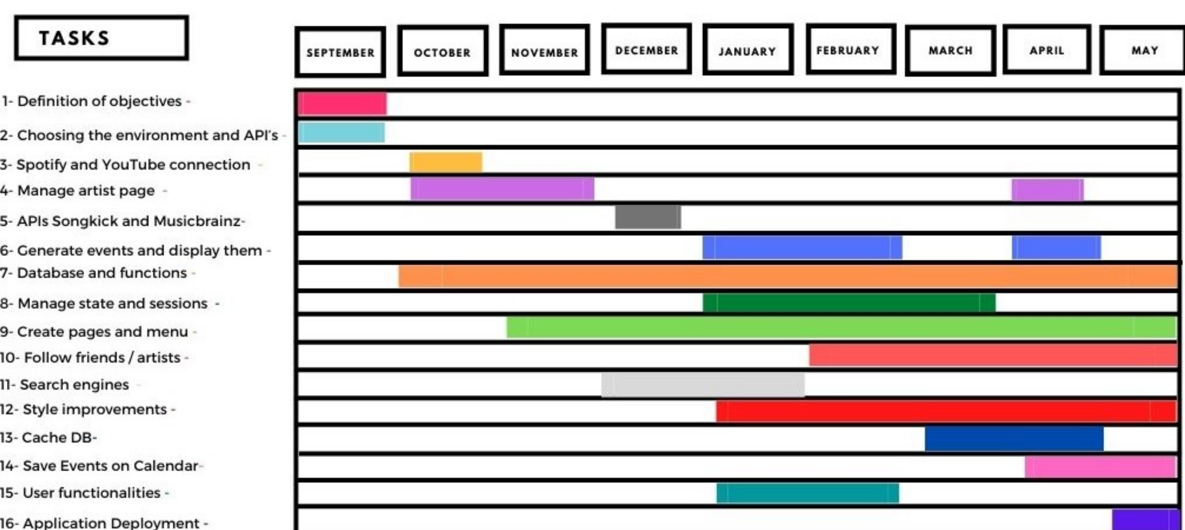

Figura 2.1: Gantt diagram

The tasks defined in this diagram are:

1. **Definition of objectives**: Definition of objectives to be considered as the basis of the project.
2. **Choosing the environment and API's**: Carrying out a study and choosing the different technologies to address the backend and frontend in the project, exposing the benefits and drawbacks of each. A study would also be carried out on which API's to integrate in the project.
3. **Spotify and YouTube connection**: Investigation and realization of the connection and data collection using the APIs of *Spotify* [3] and *YouTube* [5].
4. **Manage artist page**: Selection of the data to show about the artist and design of the entrance where it is shown.
5. **APIs Songkick and Musicbrainz**: Study of the way of relating information given by *Spotify* [3] with *Songkick* [6] and *Musicbrainz* [17], which offer more diversity of content.
6. **Generate events and display them**: Obtain data about events and illustrate them on the main webpage.
7. **Database and functions**: Integration of a *MongoDB* server [13].
8. **Manage state and sessions**: Investigation of state storage and user sessions.
9. **Create pages and menu**: Creation of new pages with various content and a menu to manage them.
10. **Follow friends / artists**: Implementation of the functionality of adding artists and friends to the user's profile, to access them quickly.
11. **Search engines**: Development of search engines for both artists and events in order to be able to filter those that are desired.
12. **Style improvements**: Making aesthetic improvements using CSS and component specific design.
13. **Cache DB**: Inclusion of a Cache Database to speed up the loading of the information.
14. **Save Events on Calendar**: Creation of the functionality to save events and display them on a calendar.
15. **User functionalities**: Creation of the user profile, their registration, the list of artists followed and that of friends.
16. **Application Deployment**: Deployment of the web application on a server.

Getting support from the previously defined work plan and the time diagram established, the mode of action will be based on carrying out a biweekly monitoring as a way to ensure that a correct execution of the original work plan is carried out. For this purpose the tasks put together will be carried out successfully between the established ones, the inconveniences encountered, if any, and if so, the way of solving them, if possible. In addition, the possible need to carry out changes in objectives to adapt them to available resources. Finally, if there are any pending aspects, they will be commented on. Thereafter that new tasks and goals will be proposed.

Each meeting was governed by the steps listed below:

- **Project review**: To start off, each team member will inform the rest of the team members of the state in which their job is at that time, the tasks completed, the problems encountered and the way in which these have been resolved, if possible. It will also be mentioned if there is a need to make important changes in any of the aspects defined in the original project, as well as indicate the reason for the change.
- **Definition of objectives**: After completing the review of the project status, we will begin to define the following objectives to be achieved for the next review, and the possible new use cases will be shared, no considered in the original proposal. It will be necessary to expose the benefits that would be obtained (technical or functional) by accepting said use case. Modifications may consist of changing the API from which data is obtained, changing the data to obtain, or the way in which these will be exposed to the user.
- **Location of resources**: After deciding the tasks to be carried out, the resources available, and the possibility of carrying them out will be considered with these resources. If this is not possible, we will explore the possible resources to integrate for the correct performance of the task. This will be done in a general way since the person assigned to this task will be in charge of deciding which resource best suits what is needed, being fundamental that the benefits offered, the agility in the data load, the way in which the data is offered and the difficulty of use of each one are taken into account.
- **Division of tasks**: Finally, these tasks will be distributed considering the workload, the difficulty of development and the previous knowledge in the subject, keeping in mind the time to dedicate to each task, thus achieving a distribution balanced to each of them.
- **Time organization**: At the end of each meeting, it will be agreed upon together the date of the next follow-up meeting and, therefore, the deadline to carry out the planned tasks.

2.5. Document structure

The report is divided into the following sections:

In chapter 1, a description of the project is given where, in addition to making a brief summary, the motivations that have existed for its realization, what objectives are to be achieved and the work plan that has been followed are explained. In the next chapter 3 there is a section on the "state of the art", where the history of music applications is briefly described and the current state of the art of the technology directly or indirectly related to the project is explained in detail. In chapter 4, there is the "requirements definition", where all the services or functionalities that the project must offer in order to fulfill the project's expectations are explained.

The "architecture" chapter describes the technologies used in the development of the project. And in the last chapter, the "development" section contains all the documentation of the process that has been followed to carry out the development of the application, as well as the information of the elements that have been used, the problems encountered and the proposed solutions. Finally, the document includes a section of "conclusions and future lines" where we explain the knowledge we have acquired, the conclusions we have reached and the explanation of everything that may be interesting to include in future versions of the application.

Capítulo 3

Estado del arte

La música, tal y como ahora la conocemos, ha sufrido grandes avances a lo largo de los años. Desde sus primeros formatos físicos, donde la gente debía conformarse con reproducir su *cassette* en un *Walkman* o, posteriormente, la evolución a los aparatos electrónicos como los famosos *iPod*, hasta que llegaron las miles de plataformas web que existen hoy en día donde, además de poder escuchar música, se ofrece gran variedad de información relacionada con cualquier tipo de cantante o grupo musical.

Por este motivo, es importante analizar cada gran acontecimiento y aplicación que ha hecho que se llegue hasta donde nos encontramos actualmente, ya sea por su interconexión con otras plataformas, el tipo de información que ofrecen o el diseño de sus interfaces.

3.1. Plataformas y aplicaciones musicales

3.1.1. Historia de las aplicaciones de música

La primera aplicación P2P (*Peer-to-peer*) fue desarrollada por Adam Hinkley en el año 1996. *Hotline Connect* [18], como se llamaba esta aplicación, tenía como objetivo la distribución de archivos. Esta plataforma estaba dirigida a universidades y grandes empresas, sin embargo, no tardó en utilizarse para el intercambio de casi todo tipo de archivos, incluyendo contenido ilegal.

Más adelante, en 1999, surgió *Napster* [19], servicio cuya finalidad era el almacenamiento de listas de equipos y archivos que proporcionaba cada uno. De esta manera, se convirtió en la primera aplicación para ordenador especializada en archivos de música MP3. MP3 es un formato de compresión de audio y vídeo desarrollado por el *Moving Picture Experts Group (MPEG)* [20] y cuyo objetivo fue facilitar su emisión y almacenamiento.

La polémica sobre los derechos de autor no tardó en aparecer ya que se trata de un derecho fundamental para todo tipo de artistas y que todos los que quisieran difundir su música debían tener en cuenta. A comienzos del año 2000, varias empresas discográficas iniciaron un juicio en contra de *Napster*, hasta que en julio de 2001 un juez ordenó el cierre de los servidores para prevenir más violaciones de derechos de autor. El 1 de diciembre de 2011, *Napster*, [19] se fusionó con *Rhapsody* [21] y comenzó a operar en diversos países de América y Europa como un nuevo servicio de pago. Actualmente tiene un convenio con la empresa telefónica *Movistar* como servicio de streaming para Latinoamérica.

La siguiente plataforma en aparecer fue *iTunes* [22] convirtiéndose en la primera "tienda online" de música. Más adelante, se produjo el lanzamiento de *Ares*, la cual obtuvo un gran éxito debido a que las descargas eran gratuitas y a que añadían funciones innovadoras como la previsualización o las colas de espera para la descarga. A pesar de todos los aspectos positivos que nos ofrecía *Ares*, también quedaban latentes una gran cantidad de inconvenientes, entre ellos, la calidad del audio, la incomodidad de tener que dedicar tiempo a descargar la música o la dificultad de uso de su interfaz. La combinación de los avances en *streaming* con la problemática explicada anteriormente propició la aparición de otros grandes servicios con subscripción de pago como *Spotify* [3] o *Deezer* [23]. Estas plataformas nos permiten acceder al contenido de forma cómoda y sin la necesidad de realizar ninguna descarga. Con posterioridad han ido surgiendo aplicaciones que no generaban su propio contenido, sino que incorporaban en sí mismas la información que le proporcionaban otros servicios. En este punto surgen plataformas como *Songkick* [6] que ofrece información sobre próximos eventos y artistas musicales.

En la figura 3.1 se puede observar, mediante una línea del tiempo, la evolución de las tecnologías mencionadas anteriormente con el paso de los años.

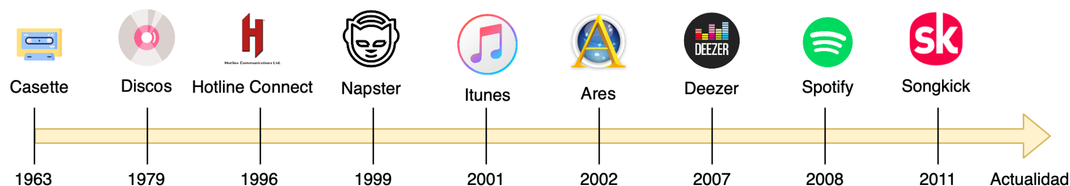

Figura 3.1: Evolución de las formas de reproducir música y aplicaciones musicales

3.1.2. Tipos de aplicaciones de música

Actualmente, la música está integrada en la sociedad como una parte muy importante de nuestras vidas, es por eso que existen múltiples alternativas de páginas webs y aplicaciones para móvil en las que se puede disfrutar de todo tipo de música en cualquier momento.

Además, podemos encontrar opciones tanto gratuitas como de pago, ya que la mayoría de las aplicaciones las podemos obtener gratuitamente, aunque con limitaciones en el tiempo de escucha, el número de canciones o con la incorporación de anuncios publicitarios. Sin embargo, en estos casos podemos optar por la subscripción, la cual nos permite disfrutar de todas las ventajas que nos llegan a ofrecer las aplicaciones.

Al mismo tiempo, la música en directo es una de las actividades más populares y por eso también se han desarrollado a lo largo de los años aplicaciones para que planificar este tipo de eventos sea mucho más rápido y simple. Es el caso de *Ticketmaster* [4] o *Taquilla.com* [24], donde se puede comprar entradas para todo tipo de conciertos y espectáculos hasta el mismo día de comienzo del evento.

3.1.3. Análisis de aplicaciones y plataformas de música

A continuación, se muestran detalladamente algunos de los servicios que encabezan la lista de más utilizados hoy en día. Como se puede observar en la figura 3.2, estos servicios se pueden dividir en plataformas, si generan su propio contenido, y en aplicaciones, si su funcionalidad reside en el consumo de la información generada por otros servicios. Además, se pueden clasificar en base al tipo de contenido que ofrecen, en este caso: visualización de vídeos, reproducción de canciones y eventos musicales.

- **Aplicaciones y plataformas para visualización de vídeos**

  1. Vimeo [25]

     Vimeo es una plataforma que proporciona servicios de visualización de vídeos de todo tipo, libres y de alta calidad. Además, ofrece otras funcionalidades como la transmisión en vivo y la grabación de pantalla. Un aspecto muy importante en el funcionamiento de esta plataforma en la creación de vídeos profesionales es que permite mantener un control de versiones, lo cual no está implementado en el resto de aplicaciones y plataformas posteriormente mencionadas. Respecto al ámbito de la música, en *Vimeo* se puede disfrutar de videoclips, pero no se puede obtener otro tipo de información relacionada con los artistas.

  2. YouTube [5]

     YouTube es una plataforma similar a la anterior, aunque la diferencia es que esta es de uso totalmente gratuito (aunque incorpora publicidad) y es de las más populares en nuestro país. Los artistas recurren en numerosas ocasiones a esta plataforma de visualización de todo tipo de contenidos para hacer públicas sus canciones o vídeos y darse a conocer a un público nuevo.

     Una de las funcionalidades que nos ofrece la plataforma son las listas de éxitos en 44 países diferentes basada en lo que los usuarios consumen para que puedas disfrutar de las mejores canciones y vídeos, los artistas más escuchados y las últimas tendencias en distintas partes del mundo. Actualmente, *Youtube* está posicionada en la primera posición como plataforma para visualizar vídeos gracias a la gran cantidad y variedad de contenido que se puede encontrar.

- **Aplicaciones y plataformas para escucha de música**

  1. Spotify [3]

     Sin duda *Spotify* es de las plataformas más demandadas a lo largo de los años para disfrutar de la música vía streaming. Este servicio dispone de un

amplio catálogo musical, ya que cuenta con discografías completas de millones de artistas. Además, ofrece multitud de listas de reproducción clasificadas en base a la popularidad en diferentes países o globalmente, al género musical o incluso a los gustos personales. El contenido al que accede cada usuario es totalmente personalizable ya que ofrece la posibilidad de crear listas de reproducción (*playlist*) propias, generadas con canciones encontradas en el perfil de los artistas, obtenidas a través del buscador o de otras playlists. Esto te da la posibilidad de guardar aquellas canciones que tras escucharlas de forma casual son de tu agrado, sin la necesidad de buscarlas. Otra característica interesante es la posibilidad de compartirlas con quien desees, cada vez es más común que los famosos creen listas de reproducción personalizadas que luego comparten con sus seguidores.

2. Deezer [23]

   Deezer es una página web donde puedes encontrar todo tipo de artistas y acceder a su amplio catálogo musical. Los usuarios pueden escuchar las listas de reproducción de otros, replicarlas o marcarlas como favoritas para acceder a ellas con más facilidad, igual que para los artistas y los álbumes. También cuenta con un blog y un foro para comentar y compartir opiniones musicales. Una característica que diferencia esta página de otras como puede ser Spotify y que lleva a muchos usuarios a usarla en su lugar. Ofrece a los usuarios la oportunidad de subir sus propias canciones sin necesidad de contar con un distribuidor, es decir, sin que sea una tercero el encargado de gestionar el contenido y los pagos, sino que cualquier persona es libre de subir su contenido siempre que cumpla determinados estándares de calidad.

   Por otro lado, *Deezer* da la opción a los usuarios de hacerse con una versión *Premium* con la cual podrán disfrutar de ventajas como la eliminación de anuncios o tener acceso a la música incluso sin conexión a internet.

   A pesar de todas las ventajas que ofrece la aplicación, no podrás encontrar información sobre los eventos de los artistas ni tener la posibilidad de comprar entradas.

3. Last.fm [7]

   Last.fm es una aplicación utilizada principalmente para la reproducción de música. El contenido que ofrece se adapta totalmente al usuario ya que almacena los datos de las reproducciones realizadas para posteriormente realizar recomendación de música. también incluye una red social en la que es posible navegar a través de los perfiles de los usuarios para conocer sus gustos musicales y de este modo ampliar el suyo propio.

- **Aplicaciones y plataformas para entradas de eventos musicales**

  1. Ticketmaster [4]

     Esta conocida plataforma es de las primeras en el ranking para comprar entradas de eventos de manera online. Ya que además de ofrecer total seguridad en la compra de entradas a cualquier tipo de evento, da la posibilidad de crear avisos para notificar que las entradas ya están disponibles. Busca conectar a los fans con las experiencias que están buscando, ofreciendo un servicio sencillo y efectivo.

En el ámbito de la música, *Ticketmaster* cuenta con información sobre eventos que pueden ser de interés. Sin embargo, no proporciona suficiente información sobre los artistas relacionados para poder conocerle un poco mejor antes de asistir a uno de sus conciertos.

2. Songkick [6]

   Esta aplicación permite a los usuarios seguir a sus artistas favoritos, ver información sobre sus próximos conciertos, además de algunos posts de fotos y vídeos. De igual forma te permite recibir alertas personalizadas de eventos y comprar entradas de una manera segura. Otra de las ventajas de esta aplicación es la posibilidad de realizar una fácil búsqueda de festivales, conciertos cerca de tu localización y música en directo.

   No obstante, *Songkick* no cuenta con la opción de reproducción de música de los artistas subscritos en la web ni de poder acceder a sus redes sociales de manera ágil, por lo que sería necesario el uso de otra aplicación o página web en el caso de que el usuario, por ejemplo, desee escuchar un fragmento de una canción del artista que dará un concierto en su ciudad.

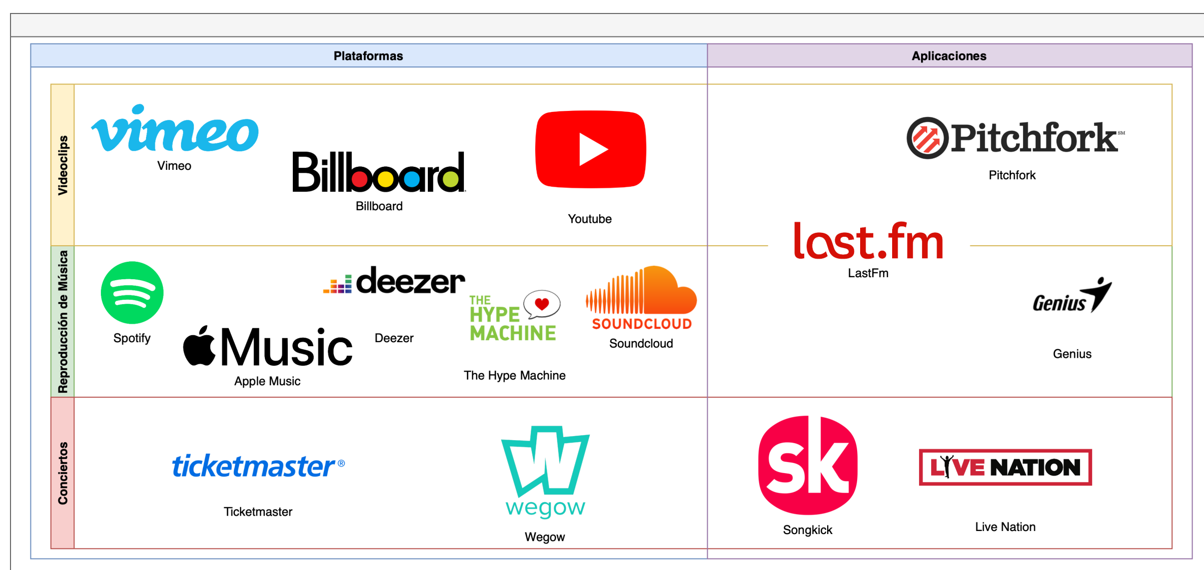

Figura 3.2: Plataformas y aplicaciones de música

3.2. APIs

Uno de los aspectos principales en el desarrollo de la aplicación es la obtención de datos a través de las diferentes fuentes de información. Toda la información que mostramos para hacer el servicio web más atractivo y completo procederá de la conexión con APIs.

¿Qué es una API?

De forma técnica, se define API (**A**pplication **P**rogramming **I**nterfaces) como el contrato de un componente de software en términos de protocolo, formato de datos y endpoint para que dos aplicaciones informáticas se comuniquen entre sí a través de una red [26].

Por tanto, en dicho contrato se definen un conjunto de protocolos y definiciones usadas en el desarrollo e integración del software que permiten que dos servicios hablen entre ellos sin la necesidad de saber cómo están implementados y sin interacción del usuario. Un proveedor de API debe proporcionar la siguiente información: la funcionalidad que se ofrece, la localización para acceder a ella, los parámetros de entrada y de salida, el acuerdo de contrato de nivel ser servicio, requerimientos técnicos sobre el límite de peticiones en un periodo de tiempo y documentación detallada para entender el funcionamiento. Las principales ventajas de las APIs son la flexibilidad que estas otorgan en cuanto al diseño y al uso. Esto simplifica el desarrollo de las aplicaciones logrando un mayor ahorro en tiempo y coste. El formato de intercambio de datos normalmente es JSON o XML.

A modo de resumen, el esquema de una API consiste en la conexión con internet y envío de datos al servidor. El servidor recupera esos datos, los interpreta, realiza las acciones necesarias y los envía de vuelta a la aplicación. Posteriormente, esta interpreta esos datos y presenta la información deseada de manera legible.

Las APIs han adquirido algunas características que las hacen extraordinariamente valiosas y útiles:

- Se adhieren a los estándares (generalmente HTTP y REST), que son amigables para los desarrolladores, de fácil acceso y comprensibles ampliamente.
- Están diseñadas para el consumo de audiencias específicas (por ejemplo, desarrolladores móviles), están documentados y están versionados de manera que los usuarios puedan tener ciertas expectativas de su mantenimiento y ciclo de vida.
- Debido a que están mucho más estandarizadas, cuentan con una disciplina mucho más sólida para la seguridad y la gobernanza.
- Tiene su propio ciclo de vida de desarrollo de software (SDLC) de diseño, prueba, construcción, administración y control de versiones. Además, están bien documentadas para su fácil consumo y control de versiones.

Tipos de API

Cada API cuenta con unas especificaciones de estructuras, clases y variables concretas e incluso pueden estar diseñadas en diferentes lenguajes de programación. Se pueden distinguir cuatro tipos principales de API [27]:

1. API de servicios web

   Ofrecen el intercambio de información entre un servicio web y una aplicación a través de una URL. Normalmente ese intercambio se produce a través de peticiones HTTP o HTTPS que contienen todo tipo de información y normalmente se envían en formato XML o JSON. Dentro de estas APIs se distinguen dos tipos, públicas o con autenticación, comúnmente aquellas que solicitan una APIKEY o clave de acceso para hacer uso de ellas es debido a que parte o todo de su contenido es de pago.

   Este es el tipo de API que usaremos para el desarrollo de *AllMusic*.

2. APIs basadas en bibliotecas

   Con este tipo de API una aplicación puede importar una biblioteca de otro software que dará acceso a las funciones, rutinas y métodos necesarios para así realizar el intercambio de información. Estas bibliotecas suelen estar diseñadas con lenguaje *JavaScript*. Un ejemplo de API basada en bibliotecas es *Google Maps*.

3. APIs basadas en clases

   Este tipo de API permite la conexión con los datos entorno a las clases en lenguaje Java. Las clases proporcionarán la información necesaria para desarrollar en la propia aplicación cualquier tipo de función.

4. APIs de funciones en sistemas operativos

   Permite a los programas software interactuar con los sistemas operativos como *Windows* o *Linux*. Algunas de las funcionalidades son, acceso y almacenamiento de datos e interfaz de usuario.

3.2.1. Estudio de las APIs

A continuación, se muestra detalladamente un estudio de las principales APIs que son de interés para el desarrollo del proyecto, ya que nos ofrecen variedad de información relacionada con la música y sus artistas.

Spotify API

La API web de *Spotify* [3] se basa en principios REST (REpresentational State Transfer), devuelve metadatos JSON sobre artistas musicales, álbumes y canciones, directamente desde el catálogo de datos de *Spotify*. Otra ventaja es que ofrece la posibilidad de obtener información como perfiles de usuario y sus listas de reproducción, contando con la seguridad de que el usuario siempre debe dar autorización a través del servicio de cuentas de *Spotify*. Todos los datos almacenados en este servicio están definidos mediante identificadores, desde los artistas, las categorías o incluso los usuarios, lo que facilita el acceso a los datos desde la aplicación desarrollada. Esto se debe a que no es necesario conocer el nombre que *Spotify* le asigna a cada elemento sino, únicamente su identificador.

La dirección base de la API es https://api.spotify.com. Además, proporciona un conjunto de elementos finales, cada uno con su propia ruta única. La forma de acceder a los recursos de datos es a través de solicitudes HTTPS estándar en formato UTF-8. La limitación de la tasa se aplica según la aplicación, en función de la identificación del cliente e independientemente del número de usuarios que utilizan la aplicación simultáneamente.

Last.fm API

Al igual que ocurre con *Spotify*, esta API se basa en principios REST (REpresentational State Transfer), pero las respuestas pueden ser a través de metadatos JSON o XML. Además, permite que cualquier desarrollador cree sus propios programas utilizando datos ilimitados. *Last.fm* ofrece una amplia gama de funcionalidades respecto a los artistas, canciones y álbumes.

La URL raíz de la API se encuentra en http://ws.audioscrobbler.com/2.0/. Sus métodos responden en XML estilo REST basados en la codificación UTF-8.

Deezer API

La API proporcionada por *Deezer* permite acceder a gran variedad de contenido que va desde la información general de los artistas, ya sea de sus álbumes o canciones, hasta la posibilidad de modificar estos, es decir, permite decidir a qué elementos queremos tener acceso y a cuáles no.

El método de acceso a dichos datos es muy flexible ya que se puede realizar a través tanto del nombre del artista, del álbum o la canción así como a través de un identificador único asignado a cada elemento del link de acceso que proporciona la propia API para cada elemento. También cabe destacar que ofrece diversos formatos de respuesta para ajustarse mejor al formato que necesite el desarrollador para tratar de forma más cómoda los datos. La URL base es la siguiente: https://api.deezer.com/.

Ticketmaster API

Esta API se basa en el descubrimiento de eventos a nivel global con funcionalidades de todo tipo como la búsqueda por palabras clave, obtener eventos de un artista en particular o búsqueda a través de un género determinado.

La URL base de las llamadas es https://app.ticketmaster.com/. Todas las claves de API se emiten con una cuota predeterminada de 5000 llamadas a la API por día y una limitación de velocidad de 5 solicitudes por segundo.

Songkick API

La API de *Songkick* brinda fácil acceso a una de la base de datos de música más grande del mundo, ya que cuenta con más de 6 millones de conciertos pasados y futuros. Cada punto final de la API le permite especificar si desea resultados en JSON o XML.

Las funcionalidades que ofrece este servicio se separan en tres grandes grupos. En el primero se encuentran las búsquedas, que pueden ser por artista, evento, área o localización; después se encuentran todas las funciones que se centran en los usuarios y la gestión de las lista de eventos próximos o pasados que tiene almacenados; y en el ultimo bloque se gestiona el acceso al historial completo de los eventos que se han producido de cada artista. La URL raíz de esta API es https://api.songkick.com/api/3.0/.

Musicbrainz API

Permite el acceso completo a la base de datos de *Musicbrainz*. Algunas de las consultas que se pueden realizar son: información sobre un artista, todos sus lanzamientos e información que permite realizar búsquedas de artistas manera sencilla.

La arquitectura de la API sigue los principios de diseño REST. La interacción se realiza mediante HTTP y todo el contenido se ofrece en un formato simple pero flexible, ya sea en XML o JSON (XML es el formato predeterminado).

Youtube API

Con la API de datos de *YouTube* se puede agregar una gran variedad de funciones de *YouTube* a la aplicación, tales como, cargar vídeos, administrar listas de reproducción y suscripciones, actualizar la configuración del canal, etc. El formato en el que esta API envuelve sus respuestas es JSON.

Positionstack API

Esta API proporciona información referente a ubicaciones. Entre sus principales funciones se encuentran: transformar direcciones de localización en coordenadas latitud y longitud o mostrar un listado de ubicaciones en base a un parámetro dado. Además, ofrece otras funcionalidad como la posibilidad de ver cómo llegar de un sitio a otro.

Mapbox API

Esta API permite visualizar mapas de manera escalable y personalizada. Además de otros servicios como la navegación, la búsqueda y administración de cuentas.

La conexión se realiza mediante el método HTTP, el cual incluye *GET*, *POST*, *PUT*, *PATCH* y *DELETE*. La ruta base es https://api.mapbox.com y la ruta del punto final es /directions/v5/profile/coordinates.

Ipstack API

Esta API permite obtener información referente a la ubicación a partir de una dirección IP. Incluye datos con IPv4 e IPv6 para sí poder alcanzar más de 2 millones de ubicaciones únicas. Estos datos pueden ser obtenidos en formato JSON o XML. Además, se encuentran protegidos mediante cifrado SSL de 256 bits y la conexión se realiza mediante HTTPS.

Capítulo 4

Requisitos

La especificación de requisitos que se detalla a continuación se ha estructurado basándose en las directrices marcadas por el estándar IEEE 830. [28]

Se explica de manera general los servicios que ofrece la aplicación, así como las ventajas que pueden obtener los usuarios. Como se ha explicado con anterioridad, este proyecto se basa en el concepto de una página web híbrida o mashup. Al combinar información de diferentes fuentes y mostrarla de forma atractiva para el usuario, el proyecto debe cumplir una serie de requisitos y especificaciones.

Los requisitos se pueden dividir principalmente en funcionales y no funcionales. Los funcionales, centrados en el comportamiento del servicio web, son aquellos que especifican qué debe hacer la aplicación. Mientras que los requisitos no funcionales se centran en el nivel operacional y se refieren a las cualidades que debe cumplir el sistema.

4.1. Descripción General

4.1.1. Perspectiva del proyecto

La aplicación *All music* está diseñada para que el usuario disfrute de cualquier tipo de contenido musical, incluyendo la visualización de los datos principales, canciones y videoclips de artistas, el descubrimiento de nuevos grupos y de eventos a los que poder asistir. Se diseña con la intención de simplificar todas las funcionalidades que proporcionan diferentes aplicaciones y plataformas de música.

4.1.2. Funciones del proyecto

All music está focalizada en la música, por lo que las funciones principales de la plataforma son:

- Registrase en la aplicación para obtener mayores ventajas como la personalización de la página según la localización y otras funciones especiales.
- Conocer información detallada de artistas, escuchar sus canciones más populares o incluso ver sus últimos videoclips. También cuenta con una sección para consultar los próximos eventos del propio artista existiendo la posibilidad de, si el usuario se encuentra logueado, añadir dicho artista a su lista personal.
- Mostrar los próximos conciertos o eventos en el país o ciudad que el usuario desee. Para facilitar la búsqueda de dichos conciertos se pone a disposición un buscador según la ubicación a la que desee asistir a un evento. Con el fin de recordar y facilitar la planificación de eventos se encuentra la funcionalidad de añadir el evento a un calendario personal.
- A su vez, *All music* hace la función de una pequeña red social, esto quiere decir que permite al usuario interactuar con otras personas, añadirlas a su lista de amigos, y conocer sus gustos musicales.

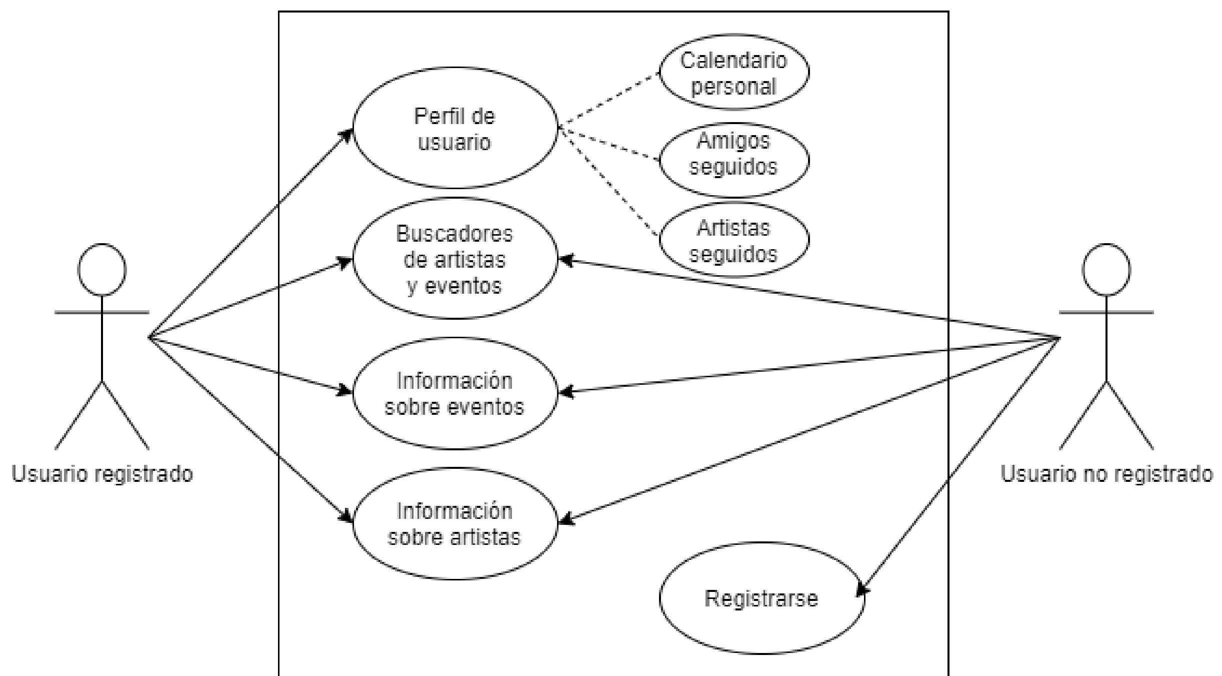

Figura 4.1: Gráfico de relaciones

4.1.3. Requisitos Futuros

En esta subsección se esbozarán futuras mejoras al sistema, que podrán analizarse e implementarse en un futuro.

1. Convertir el sistema en una API : Ya que la aplicación cuenta con una lista muy completa de requisitos funcionales sería favorable convertirla en un servicio que pudiera ser consumido por terceros de forma sencilla.

2. Sincronizar funcionalidades con *Spotify* [3]: Esto permitirá que los usuarios que ya cuenten con una sesión activa en *Spotify* puedan sincronizar de forma automática aquellos artistas a los que siguen en la plataforma, y añadirlos su perfil en *Allmusic*. Además, permitirá personalizar completamente el servicio, mostrando los eventos de las canciones que el usuario ha escuchado recientemente o las que ha seleccionado que le gustan.

3. Compartir contenido con *Google* [16]: Podría resultar muy favorecedor para la página ofrecer a los usuarios la posibilidad de sincronizar los datos de la web con la de los servicios *Google* [16], ya sea iniciar sesión haciendo uso de la cuenta de *Gmail* o sincronizar los datos del calendario de eventos con *Google-Calendar* [29].

4. Listas de favoritos: Para aquellos usuarios registrados en *All Music* se podría añadir una lista personalizada para guardar aquellas canciones que más les gusten y quieran poder escucharlas en cualquier momento de forma rápida. De la misma manera ocurriría para los vídeos relacionados con la música con los que cuentan los artistas.

5. Reproductor de música: Al acceder al perfil de un artista sería interesante contar con un reproductor de canciones que permita escucharlas todas seguidas sin la necesidad de pulsar una por una. Además, contaría con otras típicas funcionalidades como avanzar unos segundos la canción o pasar a la siguiente.

6. Búsqueda de videoclips: Para mejorar aún más la experiencia del usuario de podría implementar un buscador de videoclips a través del nombre del propio artista o de la canción deseada.

7. Ampliación de diferentes idiomas: La página actualmente cuenta con la posibilidad de navegar por ella en dos idiomas: inglés y español. En el futuro se podría ampliar este número para que cada usuario pueda elegir cualquiera de los existentes en el mundo. Incluso estableciendo por defecto el idioma del país con el que está registrado en la página.

8. Página “responsive”: Esto quiere decir, adaptar la página para que sea visible y legible desde cualquier dispositivo ya sea, ordenador, tablet o móvil.

4.2. Requisitos Específicos

4.2.1. Requisitos Funcionales

A continuación, enumeramos los requisitos funcionales, clasificados en tres principales subsecciones: usuarios, artistas y eventos.

Usuarios

RF1: Registro de usuarios

Con el fin de mejorar la experiencia del usuario, se ofrecerá la posibilidad de registrarse en la página haciendo que se convierta en una web personalizada y adaptada al usuario. Esto se conseguirá transformando completamente la información que nos ofrece la aplicación para mostrar contenido en función de la localización y los gustos del usuario. Los datos necesarios para registrarse serán: el nombre de usuario, que servirá como identificativo único, datos personales como el nombre personal, apellidos, correo electrónico, el cual deberá seguir el siguiente formato "correo@servidorCorreos.dominio", país de residencia, y región en la que se encuentra dentro de dicho país, además de una contraseña de al menos 8 caracteres para poder acceder de forma segura a su perfil.

Tras comprobar que tanto el usuario como el correo electrónico no se encuentran ya registrados y que el formato de los datos será correcto, la información se debe almacenar en una base de datos persistente. Si todo será correcto se añadirá la información del usuario a los registros, dándole acceso a la página web. En caso contrario, se informará al usuario que ha habido un problema con los datos, indicando qué dato será incorrecto.

RF2: Inicio de sesión

El usuario ya registrado podrá acceder a su cuenta mediante la página de inicio de sesión indicando el nombre de usuario y la contraseña previamente definidos durante el registro. El sistema debe cotejar los datos y en caso de que estos sean incorrectos se debe enviar un mensaje al usuario indicando que ha fallado el login y que debe volver a intentarlo. Si por el contrario, los datos introducidos son correctos, se efectuará el inicio de sesión y se redirigirá al usuario a la página principal para que pueda disfrutar de su contenido.

RF3: Editar datos de usuario

Cada usuario contará con una página de perfil desde donde podrá editar sus datos personales, siendo estos el nombre, correo electrónico, país de residencia y región. Tras modificar los datos en el formulario pre-completado y pulsar sobre editar, los datos se actualizarán en el sistema persistente de almacenamiento. En caso de que el correo electrónico no sea válido o se encuentre ya registrado con otro nombre de usuario, se notificará al usuario de que el perfil no ha podido ser actualizado por dicho motivo. Si por el con-

trario, todos los datos son correctos se informará de que se ha realizado la modificación de forma correcta a través de una alerta.

RF4: Cambio de contraseña

Desde la página del perfil de usuario también se otorgará la posibilidad de cambiar la contraseña de manera independiente generando un formulario independiente donde el usuario deberá introducir la contraseña antigua y la contraseña nueva junto con la repetición de esta. Los datos enviados estarán bajo dos niveles de comprobación, por un lado, en caso de que la contraseña antigua no concuerde con la almacenada, será el propio servidor de base de datos quien envíe como respuesta un código de error que se traducirá al usuario en forma de mensaje. En el caso de que las contraseñas nuevas no coincidan, será la propia aplicación quien gestione e informe al usuario del error a través de una alerta. Si se ha podido realizar la actualización sin incidencias, se notificará al usuario mediante una alerta.

RF5: Buscar un usuario

Tras registrarse, el usuario podrá acceder a un buscador en el cual podrá indicar el nombre de un usuario. Los resultados de la búsqueda se mostrarán en una lista con todos los usuarios que contengan la palabra introducida. En caso de no existir se devolverá un mensaje indicando que no existe ningún perfil que coincida con la búsqueda realizada.

RF6: Ver los artistas seguidos por un usuario Un usuario registrado podrá acceder a información relativa a los artistas que siguen otros usuarios en la aplicación. A través del buscador de usuarios, se podrá acceder a la página donde aparecen los artistas que sigue el usuario seleccionado.

RF7: Añadir un usuario a la lista de amigos

A través de un botón se podrá añadir a un miembro a la lista de amigos. La comprobación de que no se encuentra ya en dicha lista se hará a nivel de aplicación, por lo que solo se dará la posibilidad de añadirlo cuando no esté almacenado. Tras pulsar el botón de añadir, la aplicación responderá con la modificación de la funcionalidad y del texto de dicho botón, que pasará de “seguir” a “dejar de seguir”.

RF8: Eliminar un usuario de la lista de amigos

Al igual que ocurrirá cuando se sigue a un amigo, se mostrará la posibilidad de dejar de seguir a una persona sólo cuando ya se encuentra en dicha lista. Si se deja de seguir a un usuario desde la lista de amigos la aplicación responderá eliminando la información mostrada de este, pero si por el contrario la acción se realiza desde el perfil del propio amigo, se responderá únicamente modificando la funcionalidad y texto de dicho botón, pasará de ”dejar de seguir´´ a ”seguir´´.

RF9: Ver lista de amigos

Al igual que se dispondrá de una página de artistas seguidos también se ofrecerá la

posibilidad de gestionar la lista de amigos. Para cada amigo se mostrará la posibilidad de dejar de seguirlo, o visualizar los artistas a los que este sigue.

RF10: Añadir un artista a la lista de artistas seguidos

El usuario podrá añadir cualquier artista a su propia lista de artistas seguidos que se encontrará en el perfil del usuario. La comprobación de que no se encuentra ya en dicha lista se efectuará a nivel de aplicación por lo que solo se dará la posibilidad de añadirlo cuando no esté almacenado. Tras pulsar el botón de añadir la aplicación responderá modificando la funcionalidad y texto de dicho botón, pasará de ”seguir´´ a ”dejar de seguir´´.

RF11: Eliminar un artista de la lista de artistas seguidos

Al igual que ocurrirá cuando se sigue a un artista, se mostrará la posibilidad de dejar de seguirlo sólo cuando ya se encuentre en dicha lista. Únicamente será posible dejar de seguir a un artista desde el perfil del propio artista. La aplicación responderá modificando la funcionalidad y texto de dicho botón, pasando de ser dejar “de seguir” a “ seguir”.

RF12: Ver lista de artistas seguidos

El usuario registrado también contará con una página de artistas seguidos donde podrá observar información de cada artista seguido e ir al perfil del mismo de forma sencilla, simplemente pinchando sobre su imagen.

RF13: Añadir un evento al calendario del usuario

Desde cada evento que aparece, se mostrará un botón que te permitirá añadir ese evento al calendario del usuario para poder consultarlo cuando este quiera. Tras pulsar en el botón se modificará este mismo y su funcionalidad pasará de poder añadir un evento a eliminarlo del calendario.

RF14: Ver un evento en el calendario

Cada usuario dispondrá de un calendario propio donde se mostrarán los eventos que se han almacenado, pudiendo ver la información de cada uno simplemente pinchando sobre el día en el que se celebre dicho evento. Tras pulsar sobre el día, se generará una alerta donde se mostrará la información solicitada junto con las opciones de eliminar evento y enviar la información visualizada por correo.

RF15: Eliminar un evento del calendario el usuario

Para eliminar un evento del calendario existirán dos posibilidades. Por un lado, desde la información del propio evento que se mostrará en las diferentes páginas de la aplicación, ya que el botón que lo acompaña estará modificado para que lo elimine, en cuyo caso, tras realizar la modificación, únicamente se producirá un cambio en dicho botón. Por otro lado, la otra forma será accediendo a la información del evento desde el calendario personal, donde además de mostrar los datos del evento se da la opción de eliminar dicho

evento. El sistema responderá mostrando un mensaje indicando que el evento se ha podido eliminar de forma correcta.

RF16: Enviar la información del evento al correo electrónico del usuario

Para poder exportar los datos de los conciertos o festivales que son considerados más relevantes por el usuario, será posible enviar un email a la cuenta de correo del usuario con esta información. Para ello simplemente se deberá elegir esta opción al visualizar la información del evento desde el calendario. Esto hará que el sistema genere el mensaje, lo envíe y notifique al usuario si la acción se ha realizado con éxito o no.

RF17: Ver artistas que pueden ser de interés para el usuario

Otra sección de interés será aquella donde se podrá observar una lista de artistas recomendados en base a los gustos musicales y a los artistas seguidos por usuario. Esto permitirá descubrir nuevos artistas que se desconocían o que se habían olvidado de seguir.

Artistas

RF18: Buscar un artista

Para buscar un artista se deberá introducir el nombre de dicho artista o grupo y la aplicación mostrará un lista con aquellos artistas que más se adaptan a los criterios de búsqueda introducidos. Para obtener más información y visualizar la página específica de un artista se proporcionará un acceso sencillo pinchando sobre su foto en el listado. De esta forma, el usuario será redirigido al perfil de dicho artista.

RF19: Consulta de información de artistas

Cada artista contará con una página específica de perfil que mostrará información relevante como el número de seguidores o los géneros a los que pertenece. Además, para cada uno de ellos se mostrarán sus canciones más escuchadas de *Spotify* [3], sus videoclips musicales o vídeos relacionados con sus canciones más vistos en *YouTube* [5], los eventos en los que participará en las próximas fechas en cualquier ciudad y, por último, enlaces directos al perfil de aquellas redes sociales en las que tiene una cuenta activa.

RF20: Clasificaciones de artistas

La página contará con diversas listas de artistas para facilitar al usuario información relevante y mejorar su experiencia. Las principales son: los artistas más populares y escuchados en un país, los más escuchado en el mundo, o listados en base a géneros musicales. Esta será una forma de descubrir al usuario nuevos artistas o grupos que no conociera previamente.

RF21: Consulta de últimos lanzamientos

Otro contenido que se ofrecerá serán los últimos lanzamientos de álbumes y canciones en base a su localización. De esta forma el usuario podrá visualizar en una página todas las novedades del momento.

Eventos

RF22: Consulta de información de eventos

Con el fin de facilitar la búsqueda de eventos, se mostrarán aquellos que se celebran próximamente en el país seleccionado por el usuario. De manera añadida, contarán con enlaces directos para realizar la compra de entradas a los mismos de manera segura. De esta manera se facilitará todo lo requerido para la gestión de asistencia a eventos.

RF23: Búsqueda de eventos

La búsqueda de eventos debe resultar sencilla y muy útil para encontrar no sólo conciertos en la propia ciudad del usuario sino en destinos cercanos que puedan ser de interés. Para realizar la búsqueda se deberá indicar una localización y se mostrará una lista con aquellos que concuerden con el criterio de búsqueda.

RF24: Consultar eventos en el mapa

Si se desea buscar eventos en cualquier lugar del mundo, será posible a través del mapa donde se introduce la localización de la que se desea conocer dicha información. Como resultado, se redirigirá el mapa de forma dinámica para mostrar la ubicación geográfica de la localización seleccionada por el usuario con los eventos como puntos rojos.

RF25: Consultar información detallada de un evento en el mapa A través del mapa, el usuario podrá pulsar en los puntos rojos que muestran los eventos y obtener una información detallada de los mismos. Esta información incluirá: el nombre del establecimiento del concierto, fecha, hora, artistas que participan y estado en el que se encuentra. Además, pulsando sobre el componente se redirigirá al usuario a la página de venta de entradas de forma sencilla.

Funcionalidades generales

RF26: Cambio de idioma

Se pondrá a disposición del usuario la posibilidad de cambiar el idioma en el que se muestre la página. Para realizar dicho cambio de idioma únicamente será necesario hacer click sobre el icono del idioma deseado y la aplicación de forma automática cambiará la información a la lengua seleccionada.

RF27: Personalización de las páginas en función de la IP

Con el fin de mejorar la experiencia del usuario, si este no se encuentra registrado, se accederá a su IP para ofrecer un servicio personalizado. Desde la página principal, los nuevos lanzamientos y el mapa hasta el buscador de eventos mostrarán información que pueda ser de interés referente al lugar de procedencia. Esta información incluirá eventos, artistas y vídeos de música.

RF28: Reproducción de vídeos

Navegando a través de la web se encontrarán varias secciones donde se mostrarán diversos vídeos de Youtube [5]. Para reproducir dichos vídeos el usuario deberá hacer click sobre el botón de play. Además, contará con las mismas funcionalidades que el propio reproductor *Youtube* [5], pudiendo: parar, adelantar unos segundo, cambiar la velocidad o poner subtítulos, entre las múltiples funcionalidades que este ofrece.

RF29: Reproducción de canciones

En las secciones donde se muestren canciones de Spotify [3], será posible reproducir dichas canciones simplemente pinchando sobre el botón de reproducir y el sistema las reproducirá de forma continua hasta que el usuario decida hacer click en el botón de parar.

4.2.2. Requisitos No Funcionales

Los requisitos no funcionales especificados en nuestro proyecto son los siguientes:

Interfaces de usuario sencillas y atractivas

Se determina que la interfaz de usuario está bien diseñada cuando se comporta exactamente como el usuario piensa que lo haría. Al ser un proyecto cuya finalidad es que sea utilizado por usuarios, la interfaz es una de las partes fundamentales. El diseño debe ser interactivo y ofrecer la posibilidad de acceder fácilmente al contenido en cualquier lugar y en cualquier momento. Además, la aplicación debe integrar varios servicios en una sola plataforma de forma sencilla y atractiva para el usuario.

Rendimiento

En este proyecto el rendimiento cobra vital importancia. Al extraer datos de servicios externos se genera un tiempo de respuesta no controlable. Para evitar esta latencia, la aplicación implementará un sistema de almacenamiento de datos basado en memoria y se realizarán las llamadas necesarias para que el tiempo obtenido no sea significativo ni afecte de forma negativa a la experiencia de los usuarios. El límite establecido para este tiempo de espera es de 3 segundos.

Escalabilidad

La finalidad de este requisito es asegurar mantener el buen rendimiento de la aplicación cuando se incremente la carga de trabajo. Por ello, una de las especificaciones del desarrollo de este proyecto será la implementación de la arquitectura del proyecto en contenedores. Al ser unidades encapsuladas, permiten, de forma ágil, escalar la aplicación tanto a nivel vertical como horizontal. Al ser unidades portables es posible trasladar el sistema a una máquina con mayor poder de cómputo de forma sencilla. Además, al ser servicios completamente independientes, se pueden desplegar en una arquitectura distribuida con varias instancias de los recursos que balancean la carga.

Alta disponibilidad

La disponibilidad está reflejada en la confiabilidad continua de la aplicación, es decir, la aplicación debe estar en funcionamiento y disponible para los usuarios las 24 horas de día y los 7 días de la semana. Además, se garantizará un esquema de recuperación ante posibles fallos, especialmente con las plataformas de las que obtenemos la información y que por lo tanto, al no depender de nosotros, pueden dar problemas en cualquier momento. En este caso, el sistema debe continuar en funcionamiento mitigando y encapsulando las averías externas.

Seguridad

Hacer un sistema seguro es un requisito esencial en este proyecto ya que genera confianza y credibilidad a los usuarios. La aplicación garantizará a los usuarios que su información será manejada de forma segura y que los datos proporcionados en la aplicación serán también de confianza. De esta manera, los datos sensibles, como las contraseñas, se guardarán cifrados en la base de datos.

Capítulo 5

Arquitectura software

La arquitectura que subyace a este proyecto está dividida en varias capas. La distribución de la lógica de la aplicación en estas capas, así como las interacciones y sus estructuras están descritas en este capítulo. Estas capas, que componen una función y una responsabilidad específica en el sistema, se corresponden con las siguientes [30]:

- **Presentación**: es la capa encargada de la lógica de interacción del usuario. Su responsabilidad consiste en tratar todas las funciones de comunicación tanto de entrada como de salida con el usuario. Además, muestra toda la información ofrecida por la aplicación. Esta capa es volátil, es decir, requiere cambios constantemente, por lo que no es una buena práctica poner lógica que sea importante para otros componentes de la aplicación.

- **Negocio**: esta capa se encarga del tratamiento y limpieza de los datos. Es responsable de que los requisitos funcionales se cumplan. Orquesta las llamadas con la capa de datos y toma las decisiones referentes a las APIs: de qué API obtener los datos y en qué momento. Es la capa principal en la centralización de la lógica de la aplicación. La capa de presentación pasa la llamada a la capa de negocio que es la encargada de comunicarse con la capa de datos para formar la respuesta respetando las reglas de negocio para posteriormente devolverla a la capa de presentación.

- **Acceso a datos**: es la capa encargada de encapsular la comunicación con los servicios que proveen datos a la aplicación. En nuestro caso, estos servicios consisten en contactar con los servidores de las bases de datos que están siendo utilizadas y con las APIs previamente mencionadas.

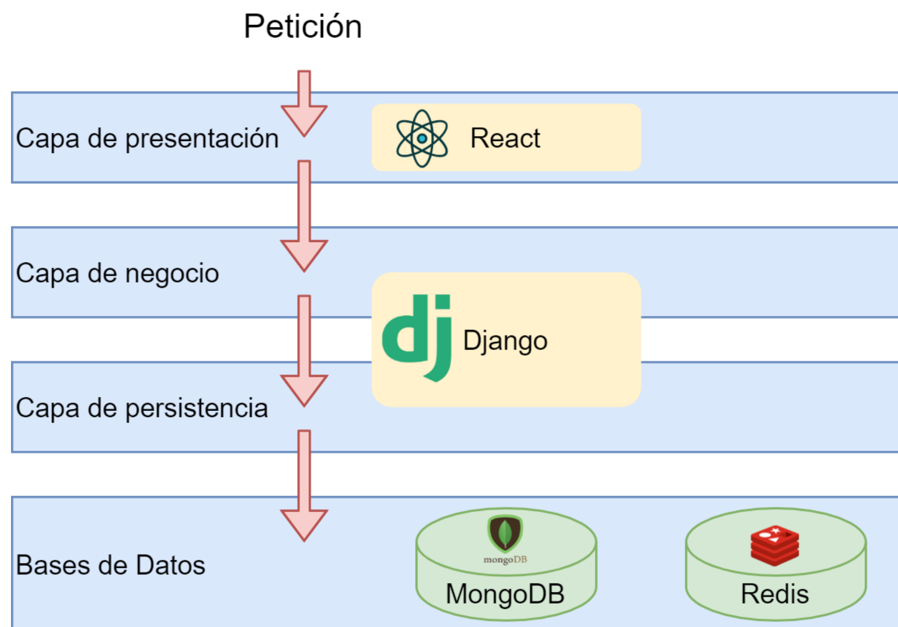

Figura 5.1: Distribución de la lógica de la aplicación por una arquitectura de capas [30]

Como se puede ver en la figura 5.1, en la capa de presentación se ha utilizado *React* [9], una librería de *JavaScript*, para diseñar y componer la parte visual de la página web. Esta librería se encarga de mostrar información al usuario y de interactuar con él. En la capa de negocio, se ha utilizado *Django* [10], un *framework* de Python que se encarga de realizar la recolección y limpieza de los datos para posteriormente pasarlos a la capa de presentación tras una petición del usuario. Para almacenar la información de los usuarios registrados, se usa la base de datos NoSQL *MongoDB* [13]. Esto permite guardar en documentos de forma flexible y eficiente toda la información que recopilamos de los usuarios.

Una de las complejidades a nivel arquitectónico del proyecto es la conexión con diferentes APIs para obtener información. De esta situación surgen dos principales problemáticas. La primera es obtener información sobre el mismo artista de diferentes APIs, ya que cada una de estas utiliza identificadores propios para representar la misma entidad. Dada la información que se quería obtener, se necesitaba una forma de relacionar el identificador de la API de *Spotify* [3] y el de la API de *Musicbraiz* [17]. Para ello, de forma dinámica, cada vez que se accede a un artista, se guardan en una base de datos *MongoDB* ambos identificadores. De esta manera, se logra reducir el número de llamadas a las APIs. La segunda problemática se refiere al tiempo de respuesta de las APIs. Esto es algo que no se puede controlar, por eso, para minimizar el tiempo de espera del usuario se ha instalado una base de datos caché. *Redis* [14] guarda toda la información expuesta en las páginas principales para mejorar el rendimiento y la eficiencia de estas. En la resolución de estas problemáticas logramos reducir notablemente la latencia del sistema. El sistema implementado para desarrollar este proyecto se compone de varios servicios conectados.

5.1. Plataforma web - React

Para el desarrollo web del proyecto se ha optado por una librería de *JavaScript* [31] denominada *React* [9]. Esta librería es popular debido a su eficiencia y flexibilidad.

Su estructura está basada en componentes que consisten en partes independientes y reutilizables de código. Dichos componentes se renderizan de manera individual, haciendo que sean totalmente encapsulables y estén fuertemente desacoplados. Esto permite realizar cambios en toda la aplicación de manera más sencilla favoreciendo el mantenimiento de la página web. La información que procesa *React* es almacenada en una colección de datos en memoria conocida como *estado*. Este estado, otorga una interfaz de acceso unificado a los datos que consumen las peticiones del usuario. En caso de que se produzca algún cambio en la página, este se actualiza y son los componentes los que se encargan de transformar dichos cambios en en una descripción para la interfaz de usuario. *React* comunica los cambios de estado entre componentes de forma unidireccional. Esto significa, que las propiedades que conforman el estado de la página únicamente se envían de componentes padre a componentes hijo. *React* organiza de forma inteligente el tiempo para optimizar los cambios de estado, mejorando la experiencia del usuario.

De forma tradicional, la implementación de estos componentes se dividía en tres grandes grupos, en base a la complejidad. Los más básicos son los llamados componentes elementales, los más sencillos al no poseer estado ni propiedades. Un segundo grupo son los componentes funcionales, que reciben propiedades en un parámetro conocido como *props*. En último lugar, los más utilizados son componentes de clase, aquellos que manejan su propio estado, tienen propiedades y ciclo de vida.

En 2018 se publicó una versión de *React* que cambió completamente la perspectiva de implementación de dichos componentes. Debido a que las futuras versiones de *React* mantendrán este enfoque, se ha decidido integrar la nueva versión de *React* 16.8, donde se introduce lo que denominan *Hooks*. Esta nueva funcionalidad permite añadir estado a la página de manera local. Un *Hook* es una función que permite tanto acceder como modificar el estado de la página. Otra de las funcionalidades de esta nueva modalidad que resulta muy relevante en el proyecto es poder ejecutar acciones denominadas *side effects*. Se considera *side effect* a todas las acciones que están fuera del foco de *React*, es decir, que extienden la funcionalidad principal de convertir el estado en una interfaz de usuario; estas acciones pueden ser, por ejemplo, ejecutar llamadas a una API a través de la red. *React* ofrece sus propios *Hooks* [32]: *useState*, *useReducer*, entre otros, que manejan el estado de los componentes. Además, te permite crear los tuyos, aunque para los casos de uso definidos en el proyecto no fue necesario. En resumen, esta actualización de *React* ofrece la misma utilidad que los componentes etiquetados como clases de manera elegante, clara y limpia; permite encapsular los *side effects* y reutilizar el código de todo el proyecto.

Utilizando este enfoque, se organiza la aplicación en componentes *smart* y *dumb* [33]. Los componentes *smart* son aquellos que manipulan datos y acceden al estado mientras que los componentes *dumb* se centran en la interfaz. Favorece la escalabilidad y facilita

el continuo desarrollo del proyecto hacia una versión más completa [31]. Además, *React* cuenta con una gran comunidad y con un amplio soporte, facilitando la solución de problemas a través de foros. Por último, otra de sus ventajas es que ofrece compatibilidad con múltiples librerías.

5.1.1. Redux

Una de las características de *React* mencionadas anteriormente, es la forma de comunicar el estado entre componentes de forma unidireccional, como se puede ver en la figura 5.2. Para aplicaciones con poco flujo de comunicación de datos, este sistema es correcto. Sin embargo, para aplicaciones con gran cantidad de información sincronizada y expuesta en varias páginas, como es el caso, no es viable. Entre los principales inconvenientes de esta arquitectura, se encuentra la necesidad de pasar información inútil para ciertos componentes con el fin de que se pueda acceder desde componentes hijos. Esto puede resultar en extensas cantidades de datos moviéndose a través de la aplicación. Con el fin de resolver esta problemática surge *Redux* [34]. Según su lema, se define como un contenedor de estado predecible para aplicaciones *JavaScript*. A pesar de ser una librería *standalone*, es decir, que se puede ejecutar en un proceso independiente, en la mayoría de las ocasiones se utiliza junto a *React*. Esta librería sigue el patrón decorador, ya que proporciona componentes que envuelven el acceso al estado implementando la lógica de gestión de manera transparente para el cliente. De esta manera, se guarda el estado de toda la aplicación en un objeto al que se accede a través de estos decoradores. Se encapsulan los datos y se centraliza su acceso pudiendo ser utilizados y modificados desde cualquier componente de la aplicación, facilitando la mantenibilidad y extensibilidad.

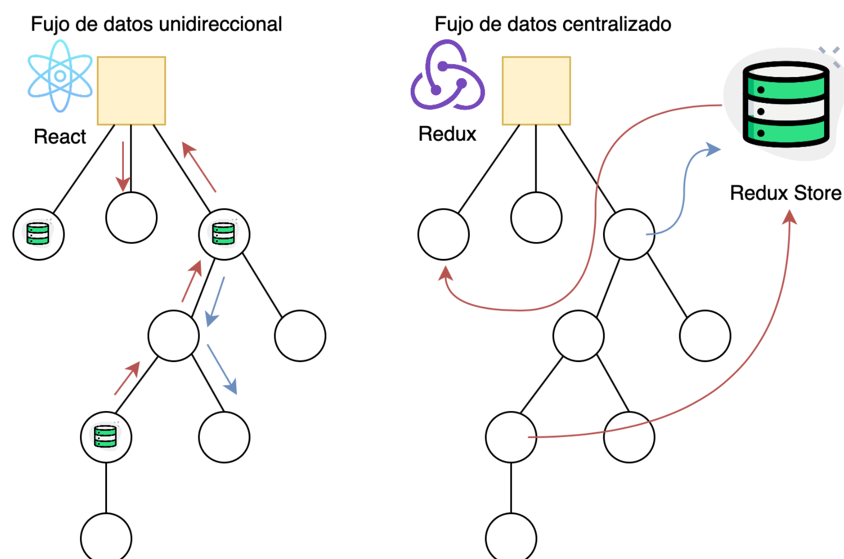

Figura 5.2: Comparativa del flujo de datos entre *React* y *Redux* [35]

Redux-Persist

Esta librería, que extiende de *Redux*, guarda el estado de la aplicación como variable local en el navegador. De esta forma, otorga persistencia a los datos de la aplicación. Además, permite seleccionar la información concreta que se quiere persistir. Esto favorece el rendimiento a la hora de recargar la página, ya que no es necesario que vuelva a hacer las llamadas a las APIs. Además, permite gestionar de forma eficaz asuntos la sesión activa de un usuario.

5.2. Integración con Django

La base de este proyecto se sustenta en ofrecer información a los usuarios a través de un servicio web. El *framework* donde se tratan los datos y se hacen peticiones a las diferentes APIs, haciendo uso de *Python*, es *Django*, el cual se corresponde con el *backend* del sistema. Este *framework* permite el desarrollo de servicios web de forma rápida, segura y con fácil mantenimiento [36]. El flujo de trabajo de Django sigue el camino expuesto en el diagrama 5.3. El navegador manda una petición al servidor web; este servicio transmite la petición al Web Server Gateway Interface (WSGI), y el WSGI comunica las aplicaciones web y los *frameworks*. En este proyecto cobra vital importancia su uso, ya que envuelve las peticiones del servidor web para que la comunicación llegue a la aplicación de *Python*, en nuestro caso *Django*. Para esto, transforma las peticiones del servidor web, en objetos de *Python HttpRequest* y al contrario, convierte los objetos *HttpResponse* en respuestas para el servidor web.

El uso del WSGI, en lugar de apuntar directamente a la aplicación concreta, favorece la flexibilidad al permitir el intercambio de servidores web sin que afecte al proyecto. Además, favorece la escalabilidad al decidir cómo se comunican las peticiones que recibe el *framework* de la aplicación. *Django* cuenta con un módulo de *Middleware* que es la capa que gestiona las peticiones y respuestas de manera segura. Siguiendo el trascurso de la petición, ésta se transmite al módulo URL *(Uniform Resource Locator)*. El *URL conf* mapea las peticiones al componente de la lógica, que es responsable de ejecutar la función que resuelve dicha petición. Este componente, seleccionado en *Django*, es una función denominada vista. La función vista puede realizar las siguientes acciones: "hablar" con la base de datos a través de los modelos, renderizar el HTML usando plantillas o *templates*, devolver una respuesta de texto o lanzar una excepción. En el caso de que la petición requiera respuesta, se devuelve un objeto con formato *HttpResponse*, posteriormente se renderiza como un *string* que abandona la aplicación *Django*.

Una de las principales características de *Django* es que sigue el patrón de diseño MVC (Modelo-Vista-Controlador). Este patrón es una arquitectura software que separa la presentación de los datos de la lógica que maneja las interacciones con usuarios. *Django* personaliza este patrón y lo convierte en Modelo-*Template*-Vista. La capa de modelos es la que sirve para estructurar y manipular los datos de la aplicación web. La capa de vista

es la encargada de encapsular la lógica de la aplicación necesaria para producir aquello que se le muestra al usuario. Y, por último, la capa *template* proporciona una sintaxis simple para renderizar la información que se presenta al usuario.

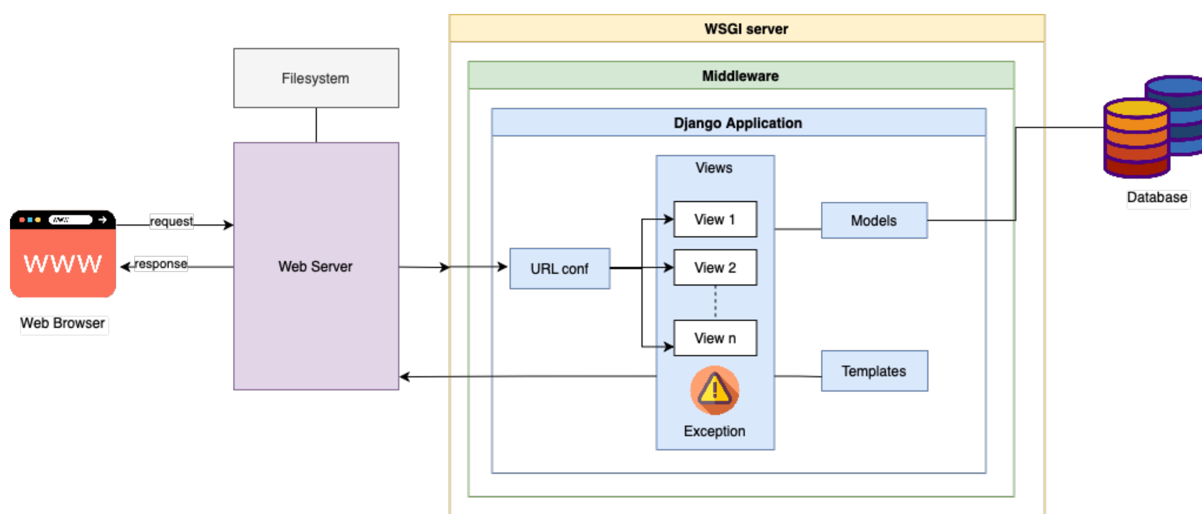

Figura 5.3: Flujo de *Django* [36]

5.3. Bases de Datos - MongoDB

Para poder cumplir satisfactoriamente con la funcionalidad requerida del servicio web, es necesario guardar información en bases de datos. Una base de datos es una plataforma para almacenar datos de forma segura, confiable y fácilmente accesible. Existen dos tipos, relacionales y no relacionales. En este caso usamos las no relacionales, también llamadas no-sql, que se caracterizan por poder almacenar grandes cantidades de información compleja y diversa. Este tipo de bases de datos almacenan la información en documentos con formato clave-valor. Para el usuario, *MongoDB* proporciona la información en formato JSON, aunque internamente se convierte a un formato binario llamado BSON. Estos documentos no cuentan con una estructura definida, son flexibles. En el caso del inicio de una aplicación, esa flexibilidad se traduce en poder realizar cambios de la estructura de forma dinámica, rápida e iterativa. En el caso de una aplicación ya desarrollada, permite la escalabilidad horizontal, ya que se puede desplegar en múltiples nodos. Además, esta funcionalidad también proporciona alta disponibilidad. En el proyecto gestionamos dos bases de datos *MongoDB*: en una guardamos datos referentes a usuarios y en otra aquellos relacionados con los identificadores de las APIs que gestionamos. El uso de estas bases de datos se explica de manera detallada en el siguiente capítulo.

5.3.1. MongoDB Atlas

Con el objetivo de pasar el proyecto a un entorno de producción, las bases de datos *MongoDB* [13] se desplegaron en *MongoDBAtlas* [37], una plataforma considerada DBaaS,

Database as a Service, es decir, Base de Datos como servicio. Esto presenta grandes ventajas como la disponibilidad que asegura el servicio, ya que se monta en modo cluster con un mínimo de tres nodos desplegados en localidades geográficas diferentes. Otra de las ventajas notables de este servicio es el rendimiento y la escalabilidad que ofrece, ya que se puede aumentar de forma dinámica el poder de cómputo asignado a la aplicación. Además, es altamente seguro con controles de acceso y *firewalls* para restringirlos.

5.4. Base de Datos Caché - Redis

Redis (***Re**mote **Di**ctionary **S**erver*) [14] es una base datos que elimina la necesidad de acceder a disco ya que trabaja únicamente en memoria. Una de las ventajas que ofrece frente a sus competidores es la posibilidad de realizar *snapshots*, es decir, permite guardar una captura de la base de datos en un determinado momento. De esta forma se puede tener una copia y no perder el servicio ante cualquier tipo de fallo que ocurra en la base de datos. También incluye funciones de réplica que facilitarían en un futuro la necesidad de escalar la aplicación.

Para mejorar el rendimiento y la eficiencia del sistema, los datos más accesibles de la página se guardan en una base de datos caché. Las llamadas a las diferentes APIs, sobre todo si se realizan múltiples en una petición del usuario y de forma síncrona, crean una latencia de varios segundos. La latencia viene determinada por el tiempo de respuesta de las APIs con las que el sistema se comunica. Puede ser causada por varios motivos, entre ellos la distancia que existe entre el servidor que hace la petición y en los que se sitúan las APIs, que normalmente se encuentran en USA. Esta latencia es ligeramente notable cuando se carga la página que muestra la información de un artista, ya que se compone de datos de las APIs de *Spotify*, *Youtube*, *Songkick* y *Musicbrainz*. Para solventar este problema, los detalles de los artistas que aparecen en páginas principales se guardan en *Redis*, una base de datos clave-valor que elimina cualquier latencia y tarda escasos microsegundos en acceder al dato ya que trabaja en memoria.

5.4.1. Redis Enterprise Cloud

Para el despliegue del proyecto en un entorno de producción, la base de datos Caché se implementó con *Redis Enterprise Cloud* [38], una plataforma basada en cloud. Esto nos ofrece prestaciones muy valoradas como la garantía de alta disponibilidad, con un SLA (Service Level Agreement o Conformidad del Nivel de Servicio) de 99.999 % o la posibilidad de escalar el poder de cómputo de manera sencilla.

5.5. Despliegue con contenedores Docker

Docker [11] es una herramienta útil en la creación, implementación y despliegue de entornos de producción. Al mantener encapsuladas las capas de nuestro proyecto en contenedores, mantenemos separado el *backend* del *frontend*. Esto hace que el proyecto sea más limpio y claro. Además, facilita el despliegue de los contenedores de manera independiente, incluso para realizar modificaciones, se puede re-desplegar el contenedor modificado sin que afecte al resto del proyecto. En este proyecto, se incorpora *Docker* en la gestión del despliegue de la aplicación. A través de la herramienta *Docker Compose*, se han diseñado diferentes contenedores que dividen y desacoplan las responsabilidades de la lógica del proyecto.

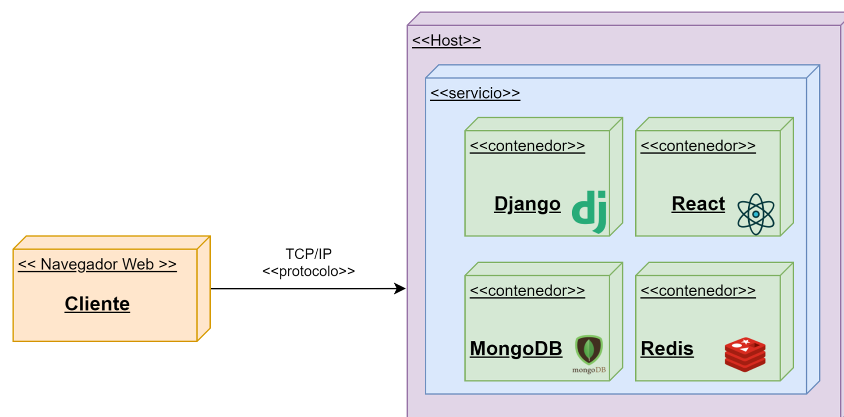

Figura 5.4: Despliegue orquestado con *Docker*

5.5.1. Despliegue del proyecto

En este proyecto, el despliegue de los contenedores se ejecuta a través de un script denominado *up.sh*. A través de comandos *bash* se ejecuta la herramienta *docker-compose* que es la encargada de montar los cuatro contenedores de los que se compone el proyecto:

- **Backend**: este es el contenedor dedicado a montar *Django*, y todas las librerías usadas para realizar conexiones con las diferentes APIs.
- **Frontend**: este contenedor se dedica a montar *React*, y todas las librerías utilizadas para el desarrollo de la web.
- **Base de Datos MongoDB**: este contenedor se dedica a montar las dos bases de datos de *MongoDB*, la que guarda información de los usuarios y la que relaciona los identificadores de las APIs.
- **Base de Datos Redis**: este contenedor se dedica a montar *Redis*, la base de datos usada como caché para mejorar el rendimiento de la página.

5.6. Despliegue en Heroku

Para desplegar el sistema en un entorno de producción se ha utilizado *Heroku*, [12] una plataforma basada en cloud. Este servicio cuenta con una opción gratuita para uso personal que otorga al desarrollador la posibilidad de desplegar aplicaciones ligeras que al comprimirse, ocupen menos de 512 Mb. Esta versión cuenta con algunas limitaciones, entre ellas, la más destacada es la suspensión temporal del servicio cuando lleva más de 30 minutos sin ser accedido. Esto genera un tiempo de respuesta y una latencia considerablemente alta para la primera vez que se accede al servicio tras dicha suspensión. Pese a esta restricción el servicio cumple todos los requisitos para el correcto despliegue de este proyecto. Contando con la posibilidad de escalar el servicio aumentando la capacidad de cómputo y el rendimiento del sistema de forma sencilla y ágil.
El entorno desarrollado cuenta con dos aplicaciones separadas, el frontend y el backend, conectadas a través de sus respectivas URLs. Como se puede ver en la figura 5.5, el proyecto se desplegó usando dos *buildpacks* predefinidos por *Heroku*: *Nodejs Heroku Buildpack* para la aplicación de React y *Python Heroku Buildpack* para la aplicación de *Django*. Cada *buildpack* se corresponde con un proceso de sistema. Estos *buildpacks* son herramientas que automatizan el proceso de construcción de la aplicación en base al lenguaje de programación utilizado. Las bases de datos ya se encontraban desplegadas en otros servicios, con lo que no fue necesaria su integración en *Heroku*.

El proyecto es accesible a través de la siguiente url: https://mashup-tfg.herokuapp.com

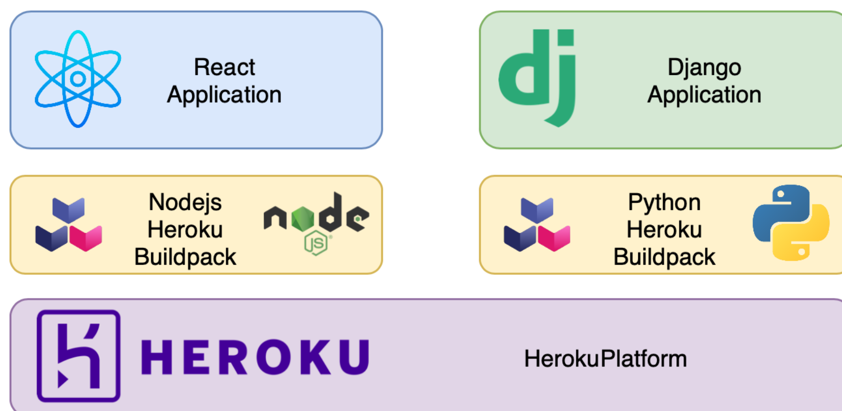

Figura 5.5: Despliegue en *Heroku*

5.7. Evaluación del Sistema

Evaluamos la calidad de nuestro sistema en base a la ISO/IEC 9126 [39]. Estas normas evalúan la calidad en base a las siguientes cualidades: funcionalidad, fiabilidad, usabilidad, eficiencia, y portabilidad [40].

- **Funcionalidad**: se considera que la aplicación es útil y funcional ya que el diseño

implementado logra que la aplicación cumpla con su cometido, ofrece al usuario acceso a información agregada de diferentes fuentes.

- **Fiabilidad**: la aplicación se ha desarrollado centrada en la tolerancia a fallos y a la alta disponibilidad. Está diseñada para evitar la pérdida del servicio incluso si es producida por recursos externos como las APIs.
- **Usabilidad**: el proyecto se compone de páginas web sencillas e intuitivas para facilitarle al usuario su utilización.
- **Eficiencia**: para mejorar el rendimiento y la eficiencia de la aplicación se implementa una base de datos caché. Esto hace que la información que se le ofrece al usuario se cargue de forma rápida.
- **Portabilidad**: al ser una aplicación implementada usando *Docker*, el despliegue es muy sencillo. Al encapsular los servicios en contenedores definidos, el trasladado a cualquier otro equipo o servidor no requiere de configuración especial.

Capítulo 6

Desarrollo de la aplicación

En este apartado de la memoria se expone el desarrollo completo de la aplicación, dividiéndola en las dos principales capas: frontend (*React*) y backend (*Django*). Se especifica una visión general de la conexión entre clases y componentes, las principales funcionalidades de la aplicación, así como aquellas implementaciones consideradas de mayor interés.

6.1. Frontend

Como se ha mencionado anteriormente, *React* [9] es una librería de *JavaScript* cuya estructura está basada en componentes. En esta sección hablaremos de aquellos más relevantes para el funcionamiento de la aplicación.

6.1.1. Enrutamiento del sistema

Una de las partes fundamentales en la aplicación es el enrutamiento de peticiones del usuario a funcionalidades del sistema, una librería que ha resultado vital para obtener una navegación fluida en la página ha sido *react-router-dom* [41]. En otras plataformas de desarrollo de *JavaScript*, el enrutamiento se hace de forma estática, es decir, antes de que se renderice la aplicación ya se han asociado todas las rutas a sus respectivos componentes. Esto, que contribuye a que no se puedan realizar modificaciones de forma ágil, hace que la página no sea flexible. Frente a esta problemática, en el desarrollo de este proyecto se optó por el uso de *react-router-dom* [41], que permite generar un enrutamiento dinámico entre componentes de forma que se puedan hacer redirecciones de unos a otros mientras que la aplicación se renderiza. Esto es vital para funcionalidades como el cambio de idioma o los servicios que sólo están disponibles cuando el usuario está logueado. Es cobra vital importancia cuando la asociación de una ruta con un componente viene determinada por variables, en este caso, de estado. Ahondando en el caso de cambio de idioma, cuando el usuario hace click en el botón para que la página se muestre en inglés, los componentes se renderizan con unas variables completamente distintas de cuando

está en español. Esto es posible gracias a componentes predefinidos por esta librería como *Router*,*Route* o *Switch*. Además, se hace uso de otros componentes como *Link* que permite redireccionar a otra ventana al interactuar con un elemento definido de la página o *Redirect* que permite redireccionar a otra ventana tras una acción determinada. También se utiliza el componente *useLocation*, que hace que al cambiar de página dentro del servicio web, esta se muestre desde su inicio y no desde el punto en el que estábamos en la página anterior.

6.1.2. Gestión del flujo de datos de la aplicación

Una de las principales librerías usadas junto a *React* [9] es *Redux*. De forma específica usamos la librería *React-Redux* [42] que permite una integración total entre ambas tecnologías. Su uso fue referente a la centralización del acceso a la información a través de funciones. De esta forma se reducía el número de relaciones y dependencias entre componentes de la aplicación favoreciendo el mantenimiento. De esta forma, permite un acceso y flujo de datos sencillo a través de todo el sistema.

El flujo de datos mediante el cual un componente solicita cierta información para después mostrarla y que el usuario pueda interactuar con ella, se muestra en la figura 6.1. A continuación explicaremos este proceso.

En primer lugar, el usuario es responsable de la interacción con la página, en concreto con un elemento programado en lenguaje *JSX*, que será el que muestre de forma atractiva y sencilla todos los datos que el propio usuario desee. *JSX* se trata de una extensión de *JavaScript* que produce elementos de *React*. Además, estas clases hacen uso de *Hooks*, que permiten usar el estado y otras características de *React* [9], sin que sea necesaria la implementación de una nueva clase.

El *Hook* más utilizado y con más relevancia en el desarrollo de la aplicación es *useEffect*. Este "hook de efecto" cuenta con la capacidad de llevar a cabo acciones en segundo plano desde el componente de función y después realizar el renderizado. Para la página de inicio de la aplicación hicimos uso de *useEffect* para realizar las llamadas a aquellas funciones que devuelven la información de los próximos eventos (teniendo en cuenta el país de procedencia del usuario en el caso de encontrarse registrado), los artistas más escuchados en el momento según la base de datos de *Spotify* [3] y, por último, los videoclips más visualizados de *YouTube* [5].

Por lo tanto, *useEffect* es el encargado de crear una acción y enviarla al objeto *Dispatcher*, mediante una única función. Para ello, se hace uso de la función *useDispatch*, que devuelve una referencia a la función *dispatch*. Esta referencia se trata de un *Hook* de *Redux*.

Además de estos *Hooks*, en la vista principal (y en la gran mayoría de páginas) se hace uso de la función *useState*, que es el llamado "hook de estado". Esto quiere decir que se llama dentro de un componente de función para agregarle un estado local. Devuelve dos valores, por un lado, el del estado actual y, por otro lado, una función que permite

actualizarlo en cualquier momento. En este caso concreto se usa *useState* para saber el elemento pulsado sobre el menú de clasificación de eventos y mostrar la información relativa a esa acción.

Una vez creada la acción desde la vista, el *Dispatcher* la envía al *Action* A.2.1. Esta clase, implementada en lenguaje *JavaScript*, tiene la responsabilidad de realizar las llamadas a las funciones que devuelven la información directamente de las APIs. Para ello, hace uso de *Axios*, que se trata de un cliente HTTP basado en promesas. Una promesa es un objeto que es devuelto y determina la terminación con éxito o el fracaso de una operación. Esto permite la utilización de las funciones *async* y *await* de *JavaScript* para obtener un código asíncrono más legible. En concreto, la función *async* (función asíncrona) se utiliza para que una función devuelva una promesa y *await* hace que la función espere la promesa. En este punto cabe destacar que se ha implementado un sistema de control de errores para que, en el caso de que alguna de las APIs utilizadas deje de estar disponible en un momento determinado, se capture la excepción y la página pueda seguir funcionando con normalidad.

Finalmente, con la información recopilada, se envía al *Reducer* A.2.2 haciendo uso de *dispatch*, definiendo el tipo de acción a realizar. El *Reducer* captura todos estos datos y actualiza el estado para que la información sea accesible desde la propia página. A continuación, el *Store* es el objeto que los reúne y cuenta con funcionalidades como permitir el acceso al estado y que este sea actualizado. Es importante destacar que sólo se cuenta con un *Store* en la aplicación, ya que para dividir la lógica del manejo de datos, se usa la composición de reductores.

Por último, en este ciclo de flujo de datos en *React*, el *Store* notificará a la vista para que pueda hacer uso de la información almacenada en el estado. Para ello se utiliza el *Hook useSelector*, el cual permite determinar qué información se quiere extraer. Se creará una instancia del selector cada vez que se renderice el componente.

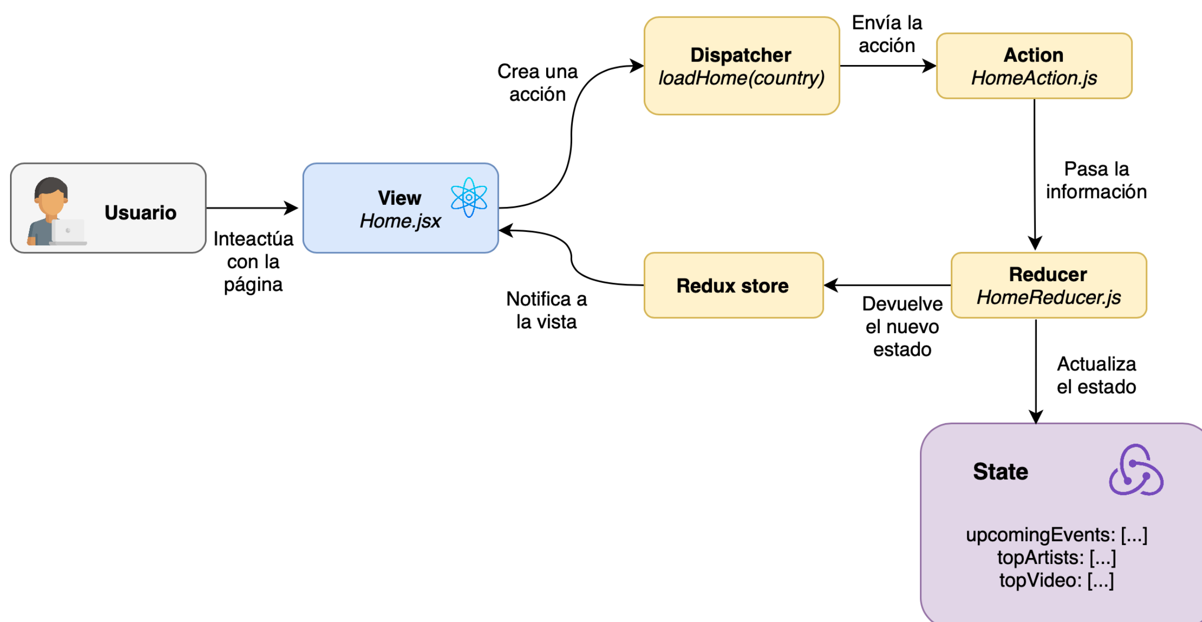

Figura 6.1: Ejemplo de flujo de datos en *Redux* [43]

Como se ha comentado previamente, los componentes permiten dividir la implementación de la interfaz de usuario en piezas independientes, reutilizables,y totalmente desacopladas [44]. Estos componentes no deben ser considerados como simples funciones incluidas en el código, ya que describen todo comportamiento componible, incluyendo el renderizado, el ciclo de vida y el estado, pudiéndose ampliar aún más el ámbito de responsabilidad cuando estos son combinados con ciertas librerías.

En los siguientes apartados se mencionan los componentes y librerías que han sido usados en el desarrollo de la aplicación.

Redux-Persist

Al principio del proceso de desarrollo no era posible guardar el estado de la página y, por tanto, no había forma de crear variables de sesión que permitieran al usuario una navegación cómoda. Además, también existía el problema de no poder refrescar las páginas sin perder la información que ellas albergaban. Este problema se debía a que el estado almacenado por *Redux* no es persistente. Como solución, usamos una librería especial llamada *Redux-Persist*. La ventaja que ofrece esta librería es la capacidad de especificar de qué páginas deseas conservar el estado en una “lista blanca” o “whitelist” y de cuáles no en una “lista negra” o “blacklist”. Para la aplicación, los datos que deseamos que persistan son los de login, registro, perfil y ubicación del usuario e idioma.

Redux-Thunk

Con el fin de poder realizar llamadas asíncronas y evitar el envío de acciones de forma síncrona que retrasa el envío de una acción hasta que se hayan cumplido determinadas condiciones o acciones se usó la librería *Redux-Thunk*. Al crear un middleware, se eliminan los problemas que pueden surgir al comunicarse con las diferentes APIs, ya que se obliga a esperar a que los elementos estén cargados antes de acceder a ellos, evitando así errores del programa como excepciones de acceso de fuera de rango a los array de información.

6.1.3. Desarrollo de la interfaz gráfica

Para añadir valor estético a la web se han utilizado los siguientes elementos de diseño. El componente “*react-country-region*-selector”, proporciona una selección de países y ciudades mediante un desplegable de fácil uso. Este componente es usado tanto en el registro como en la página principal y, en ambos casos, su finalidad es ofrecer un trato más personalizado, dando la posibilidad al usuario de visualizar los eventos de su país e incluso de su región. Como otra forma de acotar el contenido a lo que necesita el usuario se incluye en el proyecto *react-swipeable*, que añade un menú deslizable usado para incluir secciones en la vista general de eventos separando estos entre cancelados, activos, celebrados hoy o en esta semana. Dentro de los elementos que aportan una mejoría visual, se han implementado iconos estandarizados gracias a *React-icons*.

El ámbito de los eventos es considerado un pilar fundamental del proyecto y quedó latente la necesidad de incorporar una forma simple para acceder a los eventos previamente almacenados en el perfil de usuario sin necesidad de buscarlos en listas. Para ello, se ha implementado un calendario en el cual se vieran y fuera posible acceder a la información completa del evento pinchando sobre el día en que se celebra el evento. Tras probar con *react-calendar* y *react-big-calendar* finalmente se eligió *react-full-calendar*, ya que es más accesible para insertar y visualizar los eventos en el día indicado. Para mostrar la información del evento tras pinchar en el día se incluyó el componente *Sweetalert*, el cual nos proporciona alarmas personalizables más elegantes que las simples alarmas que ofrece *JavaScript*. De la mano de *SweetAlert* también se ha introducido otro tipo alertas junto con *React-popup*, para poder tener mayor diversidad en los modos de mostrar el contenido, ya que en determinados casos basta con mensajes indicativos que sean de ayuda al usuario, como mostrar una alerta al seguir o dejar de seguir a un artista o usuario. En otros casos se considera necesario que el usuario interactué con ellas para confirmar una acción.

Como una forma de realizar mejoras a los componentes creados en *React*, existen numerosas librerías que, combinándolas con estos, aporta mejoras significativas en el resultado final. *Style-components* es una librería que permite escribir código CSS en *JavaScript*. En concreto, crea un componente a la vez que define sus estilos y los condiciona a determinadas propiedades. Este componente se asocia con otros componentes a los que se les puede aplicar dicho estilo haciéndolos estéticamente más atractivos. Esto ha resultado útil para definir propiedades diferentes a componentes similares o, por el contrario, crear un diseño global que seguirán todos los componentes utilizados.

Otra librería que ofrece diferentes marcos de CSS es *Bootstrap*, con la diferencia que esta proporciona gran cantidad de componentes de diseño. Para nuestro caso particular, su mayor uso se ha centrado en la incorporación del componente *Button*, el cual nos genera botones con un estilo más elegante. La característica que ha resultado más atractiva para su uso es el estado activo que proporciona, siendo posible cambiar el estilo del botón cuando este es pulsado sin la necesidad de añadir funciones auxiliares. Otro componente que nos ha permitido tener un estilo más limpio es el llamado *Card*, el cual tiene un contenido flexible y extensible, incluyendo opciones para encabezados y pies de página, una amplia variedad de contenido, colores de fondo contextuales y potentes opciones de visualización. Están creados con *flexbox* [45], ofrecen una fácil alineación y se mezclan bien con otros componentes.

6.1.4. Cambio de idiomas

Con el fin de hacer la aplicación internacional, se ha desarrollado en español e inglés. En lugar de hacer los cambios de lengua respectivos en cada clase, la forma de implementar esta funcionalidad ha sido basada en la parametrización de todos los textos mostrados en la aplicación. Los cambios se realizan desde la clase padre “App” y se transfiere de forma unidireccional a los componentes hijos como en el diagrama mostrado en la figura 5.2. Con esto se consigue que los componentes estén encapsulados y sean independientes, favoreciendo la reutilización de los mismos. Además, al centralizar todos los textos en una clase facilita su modificación.

Otra ventaja de esta forma de implementación es que hace que añadir un nuevo idioma no sea una tarea complicada y larga. Sino que simplemente creando una llamada desde la clase padre similar a las existentes, se consiga esta funcionalidad.

6.1.5. Implementación del mapa

Para proporcionar un mapa en el que poder visualizar de forma atractiva los eventos musicales se ha utilizado la API de geolocalización *Mapbox*. De esta API utilizamos dos servicios, el primero es la visualización de un mapa de forma sencilla y el segundo es la herramienta denominada *Geocoder* que permite buscar entre lugares del mundo. Además, se creó un componente que en base a unas coordenadas pasadas como parámetros sitúa un punto en el mapa y le asocia una función para que al hacer click aparezca un desplegable con información relativa al evento ubicado en esa localización.

6.2. Backend

En el desarrollo de este proyecto el *backend* se compone de múltiples servicios interconectados. Una de las principales motivaciones arquitectónicas en el desarrollo de esta capa es desacoplar los componentes y crear una infraestructura de servicios reutilizables. Por tanto, en este proyecto *Django* funciona como una API interna, no está unido a *React* a través de ningún módulo. Como se puede ver en el diagrama 6.2, todos los servicios se encuentran separados y no tienen dependencias. Esto favorece cualidades software muy importantes mencionadas anteriormente como la interoperabilidad, ya que permite que el *backend* sea usado por cualquier otro *framework* dedicado al *frontend*. También favorece el poder realizar cambios en cualquiera de los módulos sin que afecte al resto de la aplicación. El desarrollo de la aplicación referente al *backend* consiste en la implementación de varios módulos encargados de realizar las llamadas a las correspondientes APIs, limpiar los datos y proporcionar una respuesta de forma externa. Para el primer objetivo se han desarrollado varios ficheros de *Python* que contienen funciones encargadas de la comunicación con las APIs, el filtrado y la ordenación de los datos obtenidos. Un ejemplo del proceso del filtrado y la limpieza de los datos se encuentra en el apartado A.1.2, donde a partir de la colección de datos que devuelve la función, en este caso de *Songkick*, se construye el documento *json* con la información seleccionada. También se implementan los ficheros *urls* y *views* pertenecientes a *Django*, que son los encargados de gestionar la interacción externa. Estos ficheros recogen una petición, llaman a la función encargada de resolverla y proporcionan la respuesta adecuada.

6.2.1. Envío de correos electrónicos desde el backend

Una funcionalidad añadida a la página, y desarrollada desde el *backend*, es el envío de notificaciones mediante el correo electrónico. Para enviar un correo al usuario con información acerca de su registro y de los eventos que ha decidido guardar en su calendario,

se siguieron los siguientes pasos.

Lo primero fue la configuración de nuestro proyecto modificando el fichero settings.py para agregar, en variables de entorno, el host, el puerto, el usuario y la contraseña. A continuación, para el envío del mensaje se importó la clase *EmailMultiAlternatives* y el módulo settings. *EmailMultiAlternatives* nos permite definir el asunto y mensaje del correo, el remitente y finalmente, los destinatarios. Para agregar un cuerpo al mensaje se hizo uso de un *template* renderizado, el cual consiste en un fichero HTML que contiene el formato y obtienen de las variables la información que será enviada al usuario. Finalmente, con el método *send* se envía el correo.

6.2.2. Validación del correo electrónico

Cuando un usuario desea registrarse en la página debe introducir un correo que sea válido ya que este será utilizado para el envió de información. Un correo se considera válido cuando: no comienza ni termina con un punto, y únicamente contiene uno en toda la cadena, no contiene caracteres raros, cuenta con una arroba (@) y tiene la forma correcta de escribir un correo [46]. La función en concreto se puede observar en el apartado A.1.4.

Para conseguir todo esto hacemos uso del paquete *re* de *Python*, el cual nos permite utilizar la función *compile()* para compilar un patrón de expresión regular en un objeto de expresión regular, que puede soportar métodos como *match()*, que devuelve si coinciden una serie de caracteres.

6.2.3. Cifrado de contraseñas

El cifrado de contraseñas es fundamental en el desarrollo de una aplicación, por este motivo hemos implementado un sistema de cifrado con ayuda del paquete *crypt* de *Python*. Se trata de una función hash unidireccional basada en un algoritmo DES modificado que permite almacenar en nuestra base de datos contraseñas cifradas y poder verificarlas sin la necesidad de guardar el texto en claro. El uso de algoritmos de "doble sentido" no son recomendables ya que permiten que cualquiera pueda desencriptar las contraseñas de los usuarios quedando estas al descubierto de posibles atacantes.

La función *crypt* (de "un solo sentido") recibe como parámetro la contraseña y la devuelve cifrada incluyendo una *sal* aleatoria. La información que contiene *sal* complica los ataques de diccionario, ya que cada bit de esta duplica la cantidad de almacenamiento y computación requerida. En el ejemplo del apartado A.1.5 se puede observar como, al recibir la contraseña, esta es directamente encriptada. La función incluida en el apartado A.1.6 refleja el caso en el que se necesita comprobar que la contraseña introducida por el usuario es idéntica a la almacenada, esto se consigue de forma similar, pasando a esta misma función la contraseña enviada y la almacenada, de manera que devolverá únicamente la contraseña encriptada en el caso de que ambas coincidan.

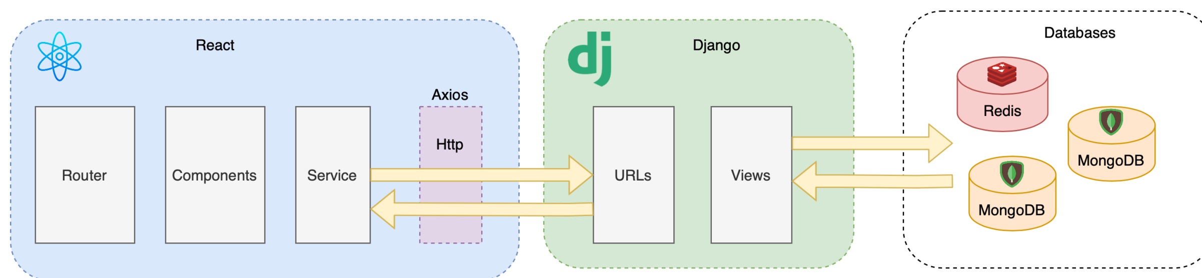

Figura 6.2: Diagrama general de la aplicación

Además de toda la información que proporcionan las APIs, también es necesario almacenar información en bases de datos.

6.2.4. Base de datos de usuarios

Con el fin de proporcionar un servicio personalizado y atractivo para el usuario, le ofrecemos registrarse en la aplicación. La información que se guarda de ese registro son los datos personales, el correo electrónico, su localización y sus gustos musicales. Con la mentalidad de que la aplicación fuera escalable y que en el futuro se pudieran guardar datos del usuario provenientes de otras plataformas, se optó por una base de datos *NoSQL*, como explicamos anteriormente.

6.2.5. Cruce de información de entidades con diferentes identificadores en cada servicio

La aplicación se basa en unificar información de diferentes plataformas musicales. Lograr este objetivo implica acceder al contenido de un mismo artista a través de diferentes servicios internos.

Dedicado a esta función anteriormente existía un servicio llamado *The Echo Nest* [47]. Esta compañía, conocida por ser líder mundial como plataforma de datos inteligente dedicada a la música, proporcionaba un servicio que guardaba información de más de treinta millones de canciones y más de tres millones artistas. Sus desarrolladores buscaban que realizara funciones como identificación de música, recomendación, creación de listas de reproducción, y análisis para consumidores. Dentro de este proyecto existía un servicio concreto llamado *Proyecto Rosetta Stone* [48], un servicio que textualmente buscaba *'hacer el mundo más sencillo para desarrolladores de aplicaciones de música'*. Este proyecto buscaba resolver la problemática atribuida a lidiar con diferentes formatos de identificadores en plataformas proveedoras de datos. Por tanto, consistía en un *mapeador* de identificadores entre varios servicios externos, en concreto, de *Musicbrainz* y *Spotify*. En el año 2014, esta compañía fue adquirida por *Spotify* y en 2016 transformó el proyecto *The Echo Nest* en lo que actualmente conocemos como *Spotify API*. Dentro de este nuevo servicio no se incluyó la funcionalidad que aportaba el proyecto Rosetta.

Actualmente, no se conoce ningún otro proyecto que proporcione una función similar para conectar identificadores en el campo musical. Con el fin de abordar esta situación se ha implementado una base de datos no-sql en *MongoDB*.

Base de Datos de Identificadores

Esta base de datos resuelve la problemática del cruce de información de entidades con diferentes identificadores en cada servicio. Cada API de la que obtenemos datos, guarda la información relacionada con un artista con un identificador específico de la plataforma. En este proyecto, al exponer datos del mismo artista pero de diferentes plataformas se genera un problema de incongruencia. En esta base de datos *NoSQL* se guardan en un documento el identificador del artista para la API de *Spotify* y para la API de *Musicbrainz*. Se almacenan en una base de datos *NoSQL* con el fin de poder realizar cambios estructurales en el futuro y añadir más identificadores de distintas APIs. Debido a la inmensa cantidad de artistas que existen, cargar los datos en un lote no es factible ya que se supera el límite de peticiones a las APIs; esto podría suponer el bloqueo de la cuenta utilizada en el proyecto para acceder a dicha API. Por tanto, los datos en esta colección se guardan de manera progresiva según los usuarios acceden a los artistas. La estructura de la base de datos es clave-valor, usando como clave el identificador del artista en la API de *Spotify* y como valor un JSON con toda la información relacionada con él.

6.2.6. Conexión entre Django y MongoDB

La librería usada para la conexión entre *Django* y *MongoDB* es *PyMongo*. Esta librería de *Python* permite la conexión con bases de datos *MongoDB* a través de una clase denominada *MongoClient*. Para tener una única instancia de esta clase, se ha implementado el patrón de diseño *singleton*. Utilizando este patrón, el sistema restringe la creación de instancias de la clase *MongoClient*. De esta manera, solo existe una instancia en memoria que centraliza todas las comunicaciones con *MongoDB*. Esta decisión de diseño incrementa el rendimiento, ya que cada instancia se corresponde con una conexión con el servidor de *MongoDB*. Si tuviéramos múltiples instancias, realizarías múltiples conexiones, lo cual podría sobrecargar el servicio de *MongoDB*.

6.2.7. Interconexión con servicios externos

Como se ha expuesto anteriormente, la interconexión con servicios externos con el fin de obtener datos es muy común en aplicaciones musicales. El rendimiento y la eficiencia que ofrecen estos servicios webs vienen determinados por el tiempo de respuesta ante peticiones de las plataformas. Este tiempo de respuesta provoca una latencia al recibir la información en la aplicación que puede ocasionar un rendimiento pobre. Esto se traduce en largos tiempos de espera a la hora de cargar los datos y mostrar la información. Actualmente, las aplicaciones cuya funcionalidad se basa en este principio generan acuerdos con las plataformas de las que se nutren para poder obtener tiempos de respuesta bajos.

Por ejemplo, *Songkick* [49] tiene como socios clave a todas las empresas de las que expone información: *YouTube*, *Spotify*, *SoundCloud*, *Foursquare*, *Warner Music Group*, *Mobile Roadie*, *Yahoo! Search*, *BBC* y *Vevo*. Ante la imposibilidad de obtener acuerdos o tener como socios a las empresas que proporcionan la información se consideró el tiempo de respuesta establecido como un cuello de botella. La opción de ir guardando la información cada vez que es accedida en una base de datos no relacional, como en el caso de *MongoDB* con los identificadores, no es viable para esta información. Al ser una información que cambia de forma no periódica habría que acceder a las APIs para comprobar que la información almacenada en la base de datos está actualizada y, en caso de no ser así, re-escribirla con frecuencia. Esta opción no soluciona el problema de latencia, ya que implica múltiples accesos a las APIs frecuentemente.
Para abordar esta situación y tratar de evadir este tiempo predeterminado, se ha implementado una base de datos caché.

Base de Datos Caché

Como explicamos en el apartado anterior, la mayor parte de la información que tratamos en el servicio web se actualiza cada cierto tiempo. Por ejemplo, la información referente a nuevos lanzamientos de canciones o el anuncio de nuevos conciertos. Teniendo en cuenta esta condición de que los datos que guardamos no son estáticos optamos por la solución de implementar una base de datos caché. Como esta base de datos caché es finita, decidimos almacenar la información que se muestra en las páginas principales de la web. De la página de inicio, guardamos la información detallada de los artistas y videoclips más populares por países; de la página del buscador de artistas guardamos la información de los artistas más populares a nivel mundial. Por ello, para el desarrollo del proyecto se implementó un *scheduler* que a las 06:00 de la mañana, hora en la que el servicio es mínimo, ejecuta las llamadas a las correspondientes APIs y carga toda esa información en la caché. Estudiando la frecuencia de los lanzamientos de artistas determinamos que en un porcentaje muy alto de las veces, cuando un artista lanza una canción, álbum o videoclip lo hace a las 00:00, hora local. Como la frecuencia de los nuevos lanzamientos tanto de álbumes como de canciones es diaria, se tomó la decisión de almacenarlo también en la caché. Con esta solución no se pierde información relevante para el usuario y se consigue una mejora en rendimiento y eficiencia. A partir de ese momento, la capa del usuario usa únicamente esta base de datos sin tener ninguna comunicación con las APIs.

Una vez la aplicación fue desplegada en *Heroku*, el entorno de producción se utilizó un *add-on* propio de este entorno llamado *Heroku Scheduler*. Esto permite ejecutar tareas programadas en el servidor donde se aloja el proyecto.

Debido a que la mayoría de las APIs devuelven documentos JSON, hemos estandarizado ese tipo de respuesta siendo el formato en el que viajan los datos a través de la aplicación. Con la librería *Redis-JSON* es posible almacenar objetos de tipo JSON como valores, de esta forma se pueden almacenar los datos con la misma estructura con la que se pasan a *React* si provienen directamente de las APIs.

6.3. Obtención de datos

6.3.1. Elección de APIs para la aplicación

Tras haber realizado un análisis de todas estas API, que nos pueden ofrecen información de interés para nuestra aplicación, concluimos que Last.fm [7]y Ticketmaster [4] eran prescindibles en el proyecto. En base a la problemática mencionada anteriormente del cruce de información de entidades con diferentes identificadores en cada servicio, decidimos acotar la elección de APIs. Para obtener la información relativa a conciertos contábamos con dos opciones. La primera era Ticketmaster [4], debido a la facilidad de obtener una clave para su uso, fue la primera por la que optamos. Sin embargo, usaba un identificador diferente a las demás APIs para los artistas, esto aumentaba la complejidad y el número de llamadas necesarias para obtener la información. Además, no contaba con la funcionalidad de devolver eventos en base a unas coordenadas. La segunda opción fue la API de *Songkick* [6], que permitía buscar a los artistas en base al identificador de *Musicbrainz* [17]. Otra de las APIs que finalmente no fue incluida en el proyecto fue la de *Last.fm* [7], debido a que no aportaba ninguna información relevante a la funcionalidad de la aplicación. El resto de APIs fueron elegidas basándonos en la información complementaria que nos ofrecen, aportando mayor valor y riqueza a la aplicación. En la figura 6.3 se puede ver un diagrama de la conexión de las APIs finalmente elegidas y su conexión con nuestra aplicación *All Music*.

En concreto, la información que hemos obtenido de cada una de las APIs es la siguiente:

Spotify API

En la aplicación los datos de la API de *Spotify* se consumen a través de la librería de *Python Spotipy*. Esta librería envuelve el acceso a los datos a través de funciones ya definidas. Además, nuestro sistema inicializa una única instancia de Spotipy, definiendo así un único punto de acceso que centraliza las comunicaciones con la API de *Spotify*.

El uso de estos datos es transversal a todo el proyecto, es la API principal de la que se consume información, para el desarrollo de esta aplicación se han implementado funciones para extraer los siguientes datos:

- Listas con los diez artistas más escuchados tanto globalmente como en un país en concreto, información de los artistas como el nombre, número de seguidores, imágenes y géneros de los artistas que pertenecen a dichas listas.
- Las diez canciones más escuchadas de un artista.
- Listado de artistas cuyo nombre es similar al introducido en el buscador, con información respecto a su nombre, género e imagen.
- Listado de artistas y canciones en base a un género musical concreto.

- Listado de artistas recomendados en base a la lista de artistas seguidos por el usuario.
- Listado de los nuevos lanzamientos, tanto álbumes como canciones en un país específico.

Songkick API

La API de Songkick es la que provee a la aplicación de toda la información relativa a eventos musicales. Se accede a ella gracias a la librería de *Python urllib*, que permite acceder al contenido que proporciona una URL desde una clase de Python. Se puede ver un ejemplo del código implementado con esta función en el apéndice de código en la sección A.1.1.

Dentro de la variedad de datos que *Songkick* [6] ofrece se han implementado funciones para obtener la siguiente información:

- Listado de eventos en los que actuará un artista.
- Listado de eventos en base a una localización geográfica dada.
- Información detallada de los eventos, su fecha, localización, estado, artistas que actúan.

Esta información aparece reflejada en la aplicación en páginas como la de inicio, donde se muestran los próximos eventos en la ubicación del usuario, en el mapa, donde aparecen los conciertos representados por puntos, en el buscador de eventos, donde aparece un listado de los conciertos cercanos a la localización buscada o en la página de los artistas, donde aparecen los próximos eventos en los que actuarán.

Musicbrainz API

Esta API es clave para la interconexión de información de artistas proveniente de diferentes APIs. Al buscar un artista en esta plataforma, Musicbrainz devuelve un identificador que se puede usar para el consumo de datos de la API de *Songkick*. Además, proporciona información añadida de los artistas como los enlaces a sus redes sociales o páginas de fans. De la misma manera que en *Spotify*, los datos se consumen a través de una librería específica de *Python*, en este caso, *musicbrainzngs*.

Youtube API

Para acceder a la información que ofrece la API de *Youtube* se utiliza la librería de *Python youtube_api*. Se han implementado funciones para obtener la lista de los diez

vídeos musicales más vistos en cada país, que se muestra en la página de inicio, y los videoclips más reproducidos de cada artista en concreto, mostrados en la página de los detalles de un artista.

Positionstack API

En concreto la API de *Positionstack* proporciona una lista de ubicaciones, con su información referente a la localización, que contienen un nombre similar al que se le pasa como parámetro. Esta función se utiliza en el buscador de eventos, para obtener la latitud y longitud de la localización seleccionada por el usuario, y así, poder obtener el listado de eventos próximos a dicha ubicación.

Mapbox API

Esta API es la única a la que se accede desde el *frontend*, es decir, desde *React*. Proporciona un componente para la visualización de un mapa, y un buscador asociado a este. También, devuelve la localización, latitud y longitud, de la ubicación buscada por el usuario; información usada para obtener y mostrar los eventos cercanos a dicha ubicación como puntos en el mapa.

Ipstack API

Esta API es utilizada para obtener la localización de un usuario a través de su dirección IP. En el proyecto se implementaron dos funciones, una relativa a la obtención de la latitud y longitud de la ubicación del usuario y otra que aporta información sobre el país y la región en la que se encuentra. Estos datos se emplean en la visualización tanto de los eventos filtrados según ubicación en la página de inicio y en el mapa, como para los artistas más populares y últimos lanzamientos en el país correspondiente.

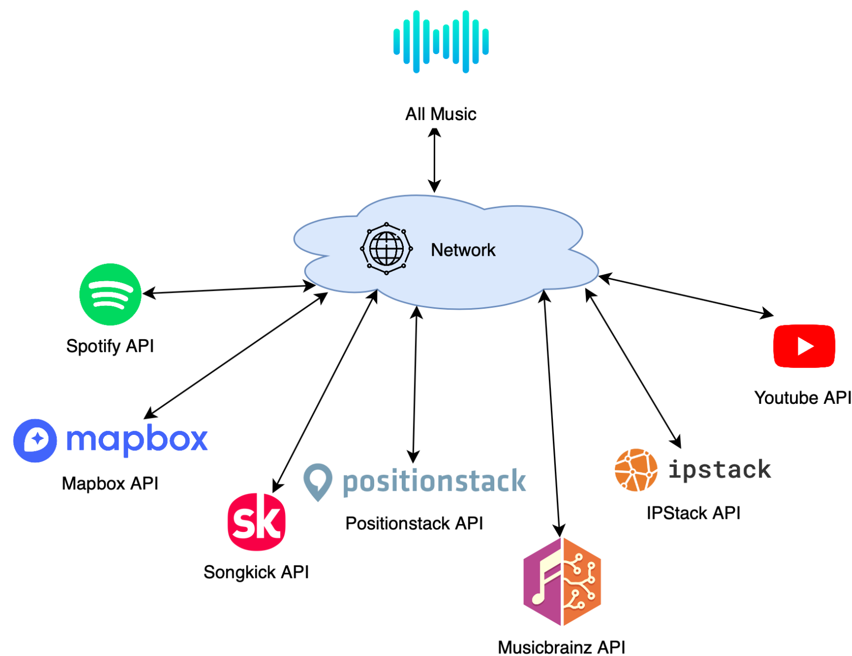

Figura 6.3: Interconexión de las APIs con la aplicación

6.4. Despliegue del proyecto

El proyecto se encuentra desplegado en dos entornos, el de desarrollo y el de producción. El entorno de producción es aquel que se corresponde con el sistema que produce la web que es visible al público. Este entorno ha sido desplegado en *Heroku*, *Redis Cloud* y *MongoDB Atlas*, como se ha explicado anteriormente. En cambio, el entorno de desarrollo, es aquel utilizado para implementar funcionalidades en el sistema, donde se pueden realizar pruebas sin que el usuario las vea. Este entorno utiliza *Docker*, con la arquitectura previamente comentada. Por lo tanto, el entorno de desarrollo sirve como antesala al entorno de producción, en el cual se busca evitar que haya errores, que deben ser detectados en el entorno de desarrollo.

El entorno de desarrollo se desplegó utilizando la tecnología *Docker*, de forma que fuera portable y fácilmente instalable en cualquier dispositivo. Como ya se ha comentado en el capítulo referente a la arquitectura, gracias a la ejecución de un fichero llamado *up.sh*, se montaba todo el entorno necesario para el correcto funcionamiento de la aplicación, con todos los *frameworks* y librerías requeridas. De la misma manera, ha sido desarrollado también el fichero *down.sh* que "desmonta" el entorno y libera el espacio utilizado en la memoria del dispositivo. Este tipo de despliegue es efectivo para entornos de desarrollo donde los cambios implementados pueden ser probados con agilidad sin que afecten a la versión en producción.

Como queríamos seguir contando con el entorno de desarrollo para seguir implementando las funcionalidades del servicio, se creó otro repositorio de *GitHub* con el código

necesario para el despliegue en *Heroku*.

El despliegue en esta plataforma fue complejo al tener varias bases de datos y dos aplicaciones diferentes enlazadas. Los cambios que se tuvieron que realizar son: creación de un fichero *ProcFile* encargado de definir las tareas o los comandos que debe ejecutar la aplicación al inicio, en este caso, para el backend se ejecutaba un fichero llamado *release-tasks.sh*, que ejecuta los siguientes comandos: *python manage.py makemigrations –no-input*, *python manage.py migrate –no-input*, *python manage.py collectstatic –no-input*. *Python manage.py makemigrations* y *Python manage.py migrate*, son los encargados de propagar los cambios hechos a la nueva versión desplegada del proyecto. *python manage.py collectstatic –no-input*, es el encargado de tratar con los ficheros estáticos del proyecto. Para la parte de web se ejecuta el comando: web: *gunicorn mashup.wsgi –log-file* -, para hacer que la web se comunicara de forma eficiente con el backend se usó *Gunicorn*.

6.4.1. Gunicorn

Como ya se comentó en el apartado de la arquitectura relativa a *Django*, 5.3, es necesario que exista un *WSGI* que intermedie la comunicación entre la web y la aplicación de *Python*. Para esto, usamos *Gunicorn* [50] un servidor HTTP puro de *Python* que permite la ejecución de cualquier aplicación de *Python* de forma concurrente, proporcionando un balance perfecto entre rendimiento, flexibilidad y sencillez.

Otro de los aspectos complejos a tratar en el despliegue fueron los ficheros estáticos almacenados en la aplicación, para gestionar esto en el backend se utilizó *WhiteNoise*.

6.4.2. WhiteNoise

WhiteNoise [51] es una librería permite a la aplicación web gestionar de forma sencilla sus propios ficheros estáticos. Entre las ventajas que nos ofrece esta librería en el despliegue en *Heroku* se encuentra la compresión que realiza de forma automática de estos ficheros. Además, los guarda en una base de datos caché con un identificador único de forma que cada versión se guarda de forma persistente.

Capítulo 7

Conclusión y líneas futuras

7.1. Conclusión

La evidencia que presentamos demuestra que, efectivamente, era necesaria la creación de una página web que unificara todos los ámbitos musicales en lugar de hacer al usuario navegar por diversas páginas web y aplicaciones con el fin de conseguir una información completa y clara. Como hemos mencionado en diversos apartados, el hecho de que el usuario tuviera que hacer uso de diversos contenidos, generaba problemas claros de seguridad, ya que este se veía obligado a crear una cuenta diferente en cada aplicación, ampliando las posibilidades de que los datos personales del mismo se vieran expuestos en internet. Por otro lado, también cabe destacar el tiempo que es requerido para satisfacer esta necesidad con las herramientas previamente existentes.

Ante esto, podemos decir con total seguridad que nuestra aplicación ofrece una solución sólida a estos problemas. Por un lado, el usuario solamente necesitará registrarse en una única plataforma desde la cual encontrará todo lo que necesita como, información de sus artistas favoritos, sus eventos, canciones y vídeos. Esto no solo ofrece una solución a este problema, ya que además aporta como valor añadido la posibilidad de guardar contenido en un perfil personalizado e interactuar con otros usuarios.

Este proyecto nos ha permitido ampliar el conocimiento sobre algunos de los lenguajes de programación estudiados a lo largo de la carrera como pueden ser *JavaScript* o HTML, así como aprender el uso de nuevas tecnologías empleadas para el desarrollo. Diseñar una aplicación que orqueste las tecnologías usadas en el backend (*Django* [10], *Redis* [14], *MongoDB* [13]) y las del frontend (*React* [9] y *Redux* [52]) de manera eficiente y extensible ha sido uno de los retos logrados más importantes del proyecto. Además, la aplicación fue desplegada en un servidor de la plataforma *Heroku*, que nos permitió ofrecer el resultado del proyecto a cualquier usuario de manera completamente abierta.

Basándonos en los objetivos iniciales, podemos concluir que el proyecto ha finalizado de manera satisfactoria, puesto que no solo se han obtenido los resultados esperados, sino que se han añadido nuevas funcionalidades no contempladas en un inicio.

7.2. Trabajo futuro

El proyecto se encuentra finalizado y disponible para su uso, sin embargo, consideramos conveniente plantear una lista de mejoras futuras para abarcar mayores necesidades y aumentar el valor a la experiencia del usuario. Como se expone en el apartado de requisitos futuros 4.1.3, teniendo en cuenta el estado actual de la página, se consideran varias posibles líneas futuras de mejora, a nivel de funcionalidades para el usuario y en relación a los requisitos no funcionales especificados. Dentro de la primera línea incluimos funcionalidades para hacer la experiencia más placentera como puede ser añadir listas de favoritos, sincronizar funcionalidades con otros servicios, proporcionar mejoras en el modo de reproducción, tanto de los vídeos como de las canciones, incluir una mayor cantidad de idiomas de traducción, para hacer una adaptación a usuarios de cualquier país y, por último, realizar mejoras estéticas para aportar mayor atractivo.

En relación con los requisitos no funcionales, se plantea como trabajo futuro el despliegue de la aplicación en un orquestador de contenedores como *Kubernetes* [53] u *Openshift* [54]. Estos servicios pueden desplegar varias instancias de cada uno de los servicios, de forma que aseguran cualidades como la alta disponibilidad, escalabilidad y rendimiento. Al balancear el flujo de trabajo en las instancias de dichos contenedores, la aplicación gana eficiencia y rendimiento. Además, si una instancia deja de dar servicio y no responde, de forma automática, todo el flujo de trabajo se traslada a otra sin dejar de dar servicio. Otra futura mejora es convertir el proyecto en una API para que la información que mostramos pueda ser consumida de forma programática por otras plataformas.

Capítulo 8

Conclusion and future work

8.1. Conclusion

The evidence we presented shows that it was indeed necessary to create a web page that would unify all the musical fields instead of making the user navigate through several different web pages and applications in order to obtain complete and clear information. As we have mentioned in various sections, the fact that the user had to make use of different content, generated clear security problems, since the user was forced to create a different account in each application, increasing the possibilities that the user's personal data would be exposed on the internet. On the other hand, it is also worth noting the time required to satisfy this need with the previously existing tools.

Given this, we can safely say that our application offers a solid solution to these problems. On the one hand, the user will only need to register on a single platform from which he will find everything he needs, such as information about his favorite artists, events, songs and videos. This not only offers a solution to this problem, but also brings as an added value the possibility to save content in a personalized profile and interact with other users.

This project has allowed us to expand our knowledge of some of the programming languages studied throughout the course, such as JavaScript or HTML, as well as to learn the use of new technologies used for development. Design an application that orchestrates the technologies used in the backend (*Django* [10], *Redis* [14], *MongoDB* [13]) and those of the frontend (*React* [9] and *Redux* [52]) in an efficient and extensible way has been one of the most important challenges of the project. In addition, the application was deployed on a *Heroku* platform server, which allowed us to offer the result of the project to any user in a completely open way.

Based on the initial objectives, we can conclude that the project has ended satisfactorily, since not only the expected results have been obtained, but also new functionalities not contemplated at the beginning have been added.

8.2. Future work

The project is finished and available for use, however, we consider it convenient to propose a list of future improvements to cover greater needs and increase the value to the user experience. As stated in the future requirements section 4.1.3, taking into account the current state of the page, several possible future lines of improvement are considered, at the level of functionalities for the user and in relation to the specified non-functional requirements. Within the first line we include functionalities to make the experience more pleasant such as adding favorite lists, synchronizing functionalities with other services, providing improvements in the playback mode, both for videos and songs, including a greater number of translation languages, to make an adaptation to users from any country and, finally, making aesthetic improvements to provide greater attractiveness.

In relation to the non-functional requirements, future work includes the deployment of the application in a container orchestrator such as *Kubernetes Kubernete* or *Openshift* [54]. These services can deploy multiple instances of each of the services, thus ensuring qualities such as high availability, scalability and performance. By balancing the workflow across instances of such containers, the application gains efficiency and performance. In addition, if an instance stops providing service and does not respond, automatically, the entire workflow is moved to another one without stopping providing service. Another future improvement is to convert the project into an API so that the information we display can be consumed programmatically by other platforms.

Contribuciones individuales

María Begoña Martínez Martín

Tras la definición de los requisitos del proyecto, se estableció la necesidad de desplegar la aplicación en un entorno de desarrollo que fuera fácil de portar y en el que se pudieran hacer modificaciones de forma ágil. Para ello, monté el sistema utilizando *Docker*, en concreto la herramienta *docker-compose*, donde definí las imágenes de los contenedores necesarios. Al ser un proyecto con librerías y dependencias específicas, implementé dos ficheros *Dockerfile* independientes para cada servicio, *Django* y *React*, donde se configura y se define el contenedor. Para facilitar el despliegue implementé el fichero *up.sh*, para montar el entorno con todas sus dependencias y *down.sh*, para desmontarlo y liberar el espacio en memoria. Para facilitar la iteración ágil y las modificaciones, creé un repositorio en *GitHub*, el cual fue indispensable para la organización del desarrollo del código del proyecto. En la configuración del proyecto creé dos ficheros de entornos: *dev.env*, donde guardábamos en variables todas las claves referentes a las APIs accedidas desde *Django* o a la conexión con los servidores donde se encuentran alojadas las bases de datos y *.env*, donde almacenábamos las variables de acceso a API y configuración de *React*. Añadí la extensión de estos fichero, *.env*, a *.gitignore* de forma que no se subieran con el resto del código al compartir las modificaciones. De esta manera, configuré al completo el repositorio de trabajo para facilitar la etapa de desarrollo.

Tras la creación del entorno, mi función principal fue la implementación de las funciones correspondientes a la conexión con las diferentes APIs. Tras investigar y decidir qué APIs eran las más adecuadas para satisfacer los requisitos definidos en el proyecto, desarrollé las funciones de obtención, filtrado y limpieza de datos. Estas funciones desarrolladas incluyen el procesamiento de datos íntegro realizado por el sistema en el backend, fundamental para el correcto funcionamiento de los servicios integrados. Además, implementé las funciones correspondientes de *Django* en *views* y en *urls* para poder exponer la información obtenida gracias a las APIs al servicio de *React*. Gracias a ello, el backend ofrece una interfaz completa de todas las funcionalidades del sistema al frontend.

Además, con el fin de reducir el tiempo de respuesta y mejorar el rendimiento de la página, me encargué de añadir una base de datos caché usando la tecnología *Redis*. En concreto, como todos el formato en el que viajan los datos a través de la aplicación es

JSON, decidí utilizar *RedisJSON*. Tras añadir la imagen al *docker-compose* previamente creado, implementé las funciones necesarias para el acceso al servicio por lectura y escritura de datos en la caché. Para poder tener la información actualizada sin penalizar el rendimiento en las horas de más servicio, decidí implementar una tarea planificada a las 06:00. Para ello, desarrollé un fichero *cronjob*, utilizando la tecnología Linux de planificación *cron*. Este trabajo ejecuta cada día un fichero con las funciones de escritura de la base de datos.

A pesar de la mejora de eficiencia que produce esta base de datos caché, sigue existiendo un pequeño retardo en la actualización del estado por parte de *React*. Para contar con un punto de acceso a los datos centralizado decidí hacer uso de la tecnología *Redux*, por la cual varias funciones actúan como decoradores encapsulan el acceso al estado. Esto genera una latencia mínimamente notable en los cambios de dicho estado. Para evitar la renderización de la página con información obsoleta, modifiqué: los *actions* y *reducers* añadiendo un nuevo campo denominado *loading* y la implementación de todas las páginas afectadas añadiendo una función condicional. De esta forma, se muestra la información una vez el estado se ha actualizado por completo o, en caso de que esté cargando, un componente que implementé simbolizando dicho estado de carga con un "spinner".

En el frontend, además de contribuir en el desarrollo de las interfaces, implementé el script *store.js* con la funcionalidad de *redux-thunk* para añadir *middleware* y poder ejecutar funciones de actualización de estado asíncronas. Agregado a esto, también integré *redux-persist* para poder mantener el estado de forma persistente. Con el fin de hacer la interfaz más personalizable para el usuario, implementé la funcionalidad de localización en base a la IP de forma transversal a la aplicación. Para ello, implementé el *action* y *reducer* necesarios para: la comunicación con la función que previamente había desarrollado en *Django* y la actualización del estado de la aplicación. Para que los componentes implementados en *React* cambiaran según esta localización, implementé funciones *useEffect* que actualizaban y renderizaban dichos componentes para que el cambio fuera efectivo en la página. También, implementé la funcionalidad referente al cambio de idioma. Desarrollé el *action* y *reducer* necesarios para poder actualizar el estado y parametricé los textos de los componentes para poder hacer cambios de forma sencilla y eficiente según las preferencias del usuario. Además, desarrollé la función del mapa dinámico e interactivo, conectando con la API de *mapbox* para la funcionalidad del *geocoder* y la visualización del mismo. También creé los componentes necesarios para mostrar los eventos en el mapa.

Como dentro de los objetivos se encontraba el despliegue del proyecto en un entorno de producción, desplegué la aplicación gracias a la herramienta *Heroku*. Al ser una aplicación con varios servicios conectados y, entre ellos, varias bases de datos, este despliegue resultó complejo. Las bases de datos se desplegaron en servidores independientes: *MongoDB Atlas* y *Redis Cloud*, de forma que se conectaban al proyecto vía HTTP. Para el despliegue del frontend y el backend como dos servicios separados, utilicé dos *buildpacks* de *Heroku* diferentes. Para la aplicación de *Python*, hice los cambios necesarios en el código del proyecto: modifiqué la configuración de *Django*, instalé e implementé *gunicorn*, para la comunicación entre la aplicación de *Python* y el servidor web, y *WhiteNoise*, para tratar con los ficheros estáticos del sistema. Para el despliegue de la aplicación de Nodejs, realicé los cambios necesarios en los ficheros *package.json* y *package-lock.json*, para asegurar que

estaban todas las librerías y dependencias necesarias instaladas. También, implementé el fichero *Procfile*, necesario para ejecutar las tareas de construcción y configuración del entorno y los ficheros estáticos del proyecto. Además, dentro del despliegue instalé los *addons* necesarios para el correcto funcionamiento de la aplicación, en concreto el *Heroku scheduler*, con el que desarrollé el script con las funciones de actualización de la base de datos caché. Otro *addon* añadido fue *Mailgun*, para enviar correos electrónicos desde la aplicación.

Referente a la memoria, fui la encargada de documentar y desarrollar el capítulo de arquitectura, explicando cuáles habían sido las tecnologías usadas en la aplicación y por qué. También escribí la sección de desarrollo del backend, obtención de datos y despliegue del proyecto. Además, con respecto a la sección de desarrollo frontend, redacté sobre aquellas funcionalidades de la interfaz que había implementado. Por último, revisé todos los capítulos de la memoria y contribuí con aquello que fue necesario.

Camila Belén Pérez Alániz

Cabe destacar antes de comenzar a explicar mi contribución en el proyecto, que he realizado una participación activa en el mismo, ya que las tareas pendientes eran distribuidas entre los diferentes miembros del grupo sin quedar exento ningún miembro del grupo.

Al comienzo del proyecto no existían tareas concretas para cada persona ya que en primer lugar debíamos escoger las tecnologías que se utilizarían para llevar a cabo el proyecto de manera satisfactoria, por ello, llevé a cabo una investigación sobre las diferentes tecnologías que eran usadas para estos fines y qué ventajas e inconveniente tenía cada una.

Tras esta etapa inicial, junto con una compañera comenzamos a buscar formación sobre React, tecnología frontend finalmente escogida, ya que queríamos crear una estructura básica donde ya pudiesen distinguirse las diferentes partes de la web, cabecera, pie de página, un menú y otros elementos comunes para todas las páginas que posteriormente se crearían. Tras bastante investigación y diversas pruebas se decidió pasar a React-redux para este fin, con lo que ya se pudo comenzar a añadir elementos a la web.

Cuando la aplicación comenzó a tener un mayor número de funcionalidades principalmente ofrecidas por las diferentes APIs que se habían introducido, quedó latente la necesidad de introducir una forma de almacenar los datos con los que trabajábamos. Ante esto fue mi labor conseguir realizar la conexión con una base de datos ya que yo tenía mayor conocimiento sobre los diferentes tipos de bases de datos.

Tras elegir *MongoDB* como modelo de base de datos a implementar, el siguiente paso fue conseguir instalar el servidor en un contenedor *Docker* independiente y que este pudiera interactuar con el resto de los contenedores, tras conseguirlo comencé a diseñar las colecciones que necesitábamos y la estructura que estas seguirían, teniendo en cuenta

los datos que necesitaríamos almacenar y el modo más cómodo para trabajar con ellos y los datos que contendrían. Para poder trabajar con la información almacenada cree diversas funciones en *Python* que permitían intercambiar esta entre *MongoDB* y *Django* y finalmente *Django* con *React*.

Tras conseguir tener la lógica y los datos correctamente definidos tanto en la base de datos como en *Django* pasé a desarrollar contenido en la web.

Mis labores consistieron principalmente en la gestión de eventos, se quería introducir un mapa y un calendario donde aparecieran los eventos. En primer lugar, añadí el mapa de *Google-Maps* para mostrar todos los eventos cerca de la localización pero, finalmente tuvo que ser descartado por problemas con los permisos.

Tras abandonar temporalmente la idea de introducir un mapa debido a los inconvenientes encontrados, pasé a introducir un calendario que permitiera al usuario gestionar desde ahí sus eventos. Para ello probé diferentes componentes, los cuales al no permitir añadir información en ellos fueron descartados. Finalmente, di con uno que me permitía mostrar los eventos e interactuar con ellos. Además de añadir la parte visual también fue necesario modificar la base de datos y la estructura lógica del programa.

Debido que el proyecto en este punto ya tenía una estructura definida comencé a realizar tareas de mejoras, tanto estéticas como en el ámbito de la lógica del programa y en la base de datos.

Dentro de estas modificaciones me encargué de cambiar el perfil del usuario, en concreto hice que los cambios que se quisieran hacer en los datos se realizaran desde un único formulario común y no campo a campo, también en esta misma página introduje la funcionalidad de eliminar una cuenta y las correspondientes alertas para asegurarnos que esto no era hecho por error. Por otro lado, me encargué de realizar mejoras en la página que contenía la lista de amigos modificando la forma en que estos se ofrecían mediante uso de componentes y CSS.

Se debe tener en cuenta que durante todo el proceso de desarrollo fui la encargada de gestionar la base de datos, añadiendo y modificando las funciones con el fin de adaptar el contenido a lo que se necesitaba en cada momento. Esto incluía desde modificar los datos que se devolvían al realizar las consultas, modificaciones en las colecciones para cambiar los datos que se guardaban, la creación de nuevas colecciones, etc.

En la memoria también he estado totalmente involucrada desde el principio, mi labor en la creación de la memoria consistió en el desarrollo de la introducción y sus subapartados, los requisitos funcionales y la conclusión, está última realizada junto a una compañera. Además de escribir estos apartados realicé las revisiones y correcciones de estos tras cada feedback.

Paula Piñuela Monjas

A lo largo del desarrollo del proyecto, la comunicación y colaboración entre todos los miembros del equipo ha sido imprescindible. Sin embargo, cada una de nosotras trabajamos en partes bien diferenciadas de manera que el avance fuera más rápido y efectivo. Mi contribución ha sido muy equilibrada en todas las fases del proyecto, desde la especificación de requisitos, la elección de las herramientas más adecuadas, hasta el fin del desarrollo. No obstante, se puede considerar que mi aportación ha sido mayoritaria en la parte de *React* (frontend), dando especial importancia a la recogida de la información del estado que las APIs devolvían.

Una vez el plan de trabajo, los objetivos y los requisitos estaban bien definidos, surgió la necesidad de encontrar un entorno de trabajo apto para el desarrollo de todo el proyecto y útil desde el sistema operativo *Windows 10*. Esta fue una tarea compleja ya que se requería de una máquina virtual de *Ubuntu Linux* que obtuviera el proyecto almacenado en un repositorio de *Github* y se conectara, de manera remota, con la herramienta *Visual Studio*, donde finalmente se llevaría a cabo la implementación.

Una vez contaba con todo lo necesario para poder comenzar la parte del desarrollo de la aplicación, y las tecnologías fueron escogidas, dediqué un periodo de tiempo al aprendizaje sobre las mismas, ya que eran completamente nuevas y desconocía su utilización. En concreto, uno de los puntos más importantes fue la comunicación entre *Django* y *React*, así como la utilización de *React-Redux*.

Posteriormente, me centré en toda la implementación relacionada con el frontend. Esto consiste tanto en la distribución y creación de cada página que contiene la aplicación, como en la recogida de la información devuelta por *Django* para su posterior uso. En primer lugar, para obtener esta información, que después sería mostrada por pantalla, fue necesaria crear una lista llamada api.js con aquellas funciones que realizan la comunicación, a través de una URL, con las funciones detalladas en urls.py del backend. A continuación, para el almacenamiento de estos datos en el estado, implementé, mediante *React-Redux*, una serie de actions y reducers separados según las diferentes funcionalidades y que darían paso a cada una de las vistas para el uso de estos datos. En las vistas principalmente fue necesario el uso de *useEffect*, *useSelector* y *useDispatch*.

Una de las partes fundamentales en la aplicación es el enrutamiento de peticiones del usuario a funcionalidades del sistema de manera que ofrezca una navegación fluida. Para ello hice uso de la librería *react-router-dom*, en concreto de los elementos como *Link* y *Router*, que permiten generar un enrutamiento dinámico entre componentes de forma que se puedan hacer redirecciones mientras la aplicación se renderiza.

Para mejorar, aclarar y evitar la repetición de código separé todas las funcionalidades, como los artistas más escuchados, la lista de eventos o la lista de canciones en componentes, de manera que en aquellas páginas donde se requería esta información solo era necesario incluir el componente con aquellas variables que le distinguieran. Además, dediqué gran parte del tiempo a la mejora de la visualización de la aplicación. Esto consiste en

la utilización del componente *styled-components*, para pequeños detalles como los títulos, elementos como *Button* y *Card* que nos ofrece *react-bootstrap* y la creación de todas las clases de estilos CSS, de manera que cada una de ellas se corresponde con una página diferente. Un ejemplo de mejoras en la visualización de la página de inicio es la clasificación de eventos, haciendo uso de un menú mediante *react-swipeable-views*.

Algunos aspectos a destacar son, la creación de alertas mediante *sweetalert* y la librería *Alert* de *reactstrap* con el fin de que la página fuera más dinámica y el usuario recibiera información acerca de sus acciones. Por otro lado, llevé a cabo la implementación de la lógica de registro e inicio de sesión y los buscadores de manera que fueran dinámicos.

Además, contribuí en otros aspectos fundamentales como la funcionalidad de cambio de idiomas parametrizando los textos de los componentes y la creación del calendario mediante el componente *fullcalendar*, donde se muestran los eventos del usuario registrado. También incluí el uso de un "spinner" en algunas de las páginas donde la carga era lenta y producía su renderización con información obsoleta. Esto se consiguió mediante una nueva variable llamada loading la cual comprobaba que el estado haya sido actualizado por completo.

A pesar de haberme dedicado en casi la totalidad a la parte frontend de la página, también he llevado a cabo algunas de las funciones del backend. En concreto las más importantes son, el uso de la clase *EmailMultiAlternatives* y el módulo *settings* para el envío de mensajes al correo electrónico del usuario, tanto con las información sobre sus eventos como al realizar un nuevo registro en la página, consiguiendo así facilitar el acceso a la información. El uso del paquete *re* de *python*, el cual me ha permitido realizar la validación de los correos electrónicos que el usuario introduce. De esta manera nos aseguramos de que el correo siempre es correcto y está bien construido, es decir, contiene todos los caracteres obligatorios para la formación de un *email*. La función implementada para esta finalidad es usada tanto en el registro de nuevos usuarios, como en el caso de que el usuario desee modificar el correo con el que se registró. Otro aspecto fundamental que implementé fue el cifrado de contraseñas de manera que nunca se trabajara con contraseñas en texto en claro y aportará a la página una mayor fiabilidad. La principal finalidad del cifrado de contraseñas es evitar a los atacantes hacerse con ellas, es por eso por lo que el algoritmo escogido es de "un solo sentido" y no permite descifrar las contraseñas almacenadas y encriptadas.

Por otro lado, mientras desarrollaba algunas funcionalidades en *React*, fue necesario hacer modificaciones en las funciones implementadas de *Django* por lo que también pude aportar y conocer en detalle esta parte. Por ejemplo, fui la encargada de realizar las comprobaciones necesarias con la información recibida en el *login* y *register*.

En relación con la memoria, mi principal aportación ha sido en la definición de la estructura del apartado de requisitos en base al estándar IEEE 830-1998 para el ERS (Especificación de Requerimientos de Software), así como la explicación de requisitos futuros, funcionales y no funcionales. Ya que mi trabajo se centró en la implementación de la parte frontend, he detallado todo este apartado en conjunto con la obtención de datos mediante APIs. En cuanto al apartado de backend del desarrollo he explicado aquellas funcionalidades en las que estuve implicada. También he documentado el apartado de

estado del arte y explicado, de manera conjunta, gran parte del resumen y las conclusiones finales. Por último, y de la misma manera que mis compañeras, contribuí a aquellas mejoras y correcciones requeridas en otros apartados.

Índice de figuras

1.1. Diagrama de Gantt 5

2.1. Gantt diagram 12

3.1. Evolución de las formas de reproducir música y aplicaciones musicales . . 18

3.2. Plataformas y aplicaciones de música 21

4.1. Gráfico de relaciones 28

5.1. Distribución de la lógica de la aplicación por una arquitectura de capas [30] 38

5.2. Comparativa del flujo de datos entre *React* y *Redux* [35] 40

5.3. Flujo de *Django* [36] 42

5.4. Despliegue orquestado con *Docker* 44

5.5. Despliegue en *Heroku* 45

6.1. Ejemplo de flujo de datos en *Redux* [43] 49

6.2. Diagrama general de la aplicación 54

6.3. Interconexión de las APIs con la aplicación 60

Bibliografía

[1] L. Aguiar and B. Martens, "Digital music consumption on the internet: Evidence from clickstream data," *Information Economics and Policy*, vol. 34, pp. 27–43, 2016.

[2] P. Vonderau, "The spotify effect: Digital distribution and financial growth," *Television & New Media*, vol. 20, no. 1, pp. 3–19, 2019.

[3] "Spotify." https://www.spotify.com/es/.

[4] "Ticketmaster." https://www.ticketmaster.es/.

[5] "Youtube." https://www.youtube.com/.

[6] "Songkick." https://www.songkick.com/.

[7] "Last.fm." https://www.last.fm/es/.

[8] A. Oyedele and P. M. Simpson, "Streaming apps: What consumers value," *Journal of Retailing and Consumer Services*, vol. 41, pp. 296–304, 2018.

[9] "React." https://reactjs.org/.

[10] "Django." https://docs.djangoproject.com/en/3.2/.

[11] "Docker." https://www.docker.com/.

[12] "Heroku." https://www.heroku.com/.

[13] "Mongodb." https://www.mongodb.com/es/.

[14] "Redis." https://redis.io/.

[15] P. D. Dalcher, "Why planning is more important than plans," 2016.

[16] "Google." https://www.google.es/.

[17] "Musicbrainz." https://musicbrainz.org/.

[18] "Hotline-connect." https://www.macintoshrepository.org/6691-hotline-connect-client-server-tracker-/.

[19] "Napster." https://es.napster.com/.

[20] "Mpeg." https://www.mpegstandards.org/.

[21] "Rhapsody." https://industriamusical.es/rhapsody-ahora-se-llamara-napster/.

[22] "itunes." https://www.apple.com/es/itunes/.

[23] "Deezer." https://www.deezer.com/es/.

[24] "Taquilla.com." https://www.taquilla.com/.

[25] "Vimeo." https://vimeo.com/es.

[26] M. Amundsen, *Design and Build Great Web APIs.* Pragmatic Bookshelf, 2020.

[27] "Qué es una api y qué puede hacer por mi negocio." https://www.bbvaapimarket.com/es/mundo-api/que-es-una-api-y-que-puede-hacer-por-mi-negocio/.

[28] "Ieee-830." https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf.

[29] "Google-calendar." https://calendar.google.com/.

[30] M. Richards, *Software Architecture Patterns.* O'Reilly Media, Inc., 2015.

[31] E. Wohlgethan, *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue. js.* PhD thesis, Hochschule für Angewandte Wissenschaften Hamburg, 2018.

[32] J. Larsen, *React Hooks in Action.* Manning Publications, 2021.

[33] "Smart and dumb components." https://medium.com/@thejasonfile/dumb-components-and-smart-components-e7b33a698d43.

[34] M. Garreau and W. Faurot, *Redux in Action.* USA: Manning Publications Co., 1st ed., 2018.

[35] K. Chinnathambi, *Learning React: A Hands-On Guide to Building Web Applications Using React and Redux, Second edition, year = 2018, isbn = 9780134843582, publisher = Addison-Wesley Professional.*

[36] A. Ravindran, *Django Design Patterns and Best Practices.* Packt Publishing, 2015.

[37] "Mongodb-atlas." https://www.mongodb.com/es/cloud/atlas/.

[38] "Redislabs." https://redislabs.com/redis-enterprise-cloud/overview/.

[39] A. Abran and R. Al-Qutaish, *ISO 9126: Analysis of Quality Models and Measures*, pp. pp. 205–228. 09 2010.

[40] P. Botella, X. Burgués, J. Carvallo, X. Franch, G. Grau, J. Marco, and C. Quer, "Iso/iec 9126 in practice: what do we need to know?," 01 2004.

[41] "React router." https://reactrouter.com/web/guides/philosophy/.

[42] "React-redux." https://react-redux.js.org.

[43] D. Geary, "Introducing redux." https://developer.ibm.com/technologies/web-development/tutorials/wa-manage-state-with-redux-p1-david-geary/.

[44] “Componentes react.” https://es.reactjs.org/docs/components-and-props.html/.

[45] “A complete guide to flexbox.” https://css-tricks.com/snippets/css/a-guide-to-flexbox/.

[46] “Cómo validar una dirección de correo electrónico.” https://micro.recursospython.com/recursos/como-validar-una-direccion-de-correo-electronico.html.

[47] R. Prey, *Musica Analytica: The Datafication of Listening*, pp. 31–48. 01 2016.

[48] E. Nest, “Introducing porject rosetta stone.” https://musicmachinery.com/2010/02/10/introducing-project-rosetta-stone/.

[49] Songkick, “Business model of songkick.” https://www.cleverism.com/company/songkick/.

[50] “Gunicorn.” https://gunicorn.org.

[51] “Whitenoise.” http://whitenoise.evans.io/en/stable/.

[52] “Redux.” https://redux.js.org.

[53] “Kubernetes.” https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[54] “Openshift.” https://www.openshift.com/.

[55] “Componentes y propiedades.” https://es.reactjs.org/docs/components-and-props.html.

[56] J. Kim, C. Nam, and M. H. Ryu, “What do consumers prefer for music streaming services?: A comparative study between korea and us,” *Telecommunications Policy*, vol. 41, no. 4, pp. 263–272, 2017.

[57] “Locationiq.” https://locationiq.com/.

Apéndice A

Código fuente

A.1. Backend

A.1.1. Uso de la librería de Python urllib para la obtención de datos

```
1  def search_location(self,lat,lon):
2          ## Search venues in location
3          try:
4              link = f"https://api.songkick.com/api/3.0/events.json?location=geo:{lat},{l
   ↪   on}&apikey={SK_API_KEY}"
5              f = urllib.request.urlopen(link)
6              data = json.loads(f.read())
7              res_data = []
8              if(data['resultsPage']['totalEntries']==0):
9                  return res_data
10             else:
11                 return self.dt.data_cleaner(data)
12         except:
13             return []
```

A.1.2. Ejemplo del procesamiento y la limpieza de los datos

```
1  def data_cleaner(self,data):
2          res_data = [{
3              'event_name':x['displayName'],
4              'event_type':x['type'],
5              'event_uri':x['uri'],
```

```
6             'event_status':x['status'],
7             'event_start':x['start'],
8             'event_artist':[{
9                 'artist_id':y['artist']['id'],
10                'artist_name':y['artist']['displayName'],
11            }for y in x['performance']],
12            'event_venue': {
13                'venue_id':x['venue']['id'],
14                'venue_name':x['venue']['displayName'],
15                'venue_metro_area':x['venue']['metroArea']['displayName'],
16                'venue_country':x['venue']['metroArea']['country']['displayName'],
17                'venue_city':x['location']['city'],
18                'venue_latitude':x['venue']['lat'],
19                'venue_longitude':x['venue']['lng'],
20            }
21        }for x in data['resultsPage']['results']['event']]
22        return res_data
```

A.1.3. Ejemplo de uso del paquete *EmailMultiAlternatives* para la gestión de correos electrónicos

```
1  def send_email_event(mail,title,date,time,type,venue_name,venue_city):
2      context = {'mail': mail,
3      'title': title,
4      'date': date,
5      'time': time,
6      'type': type,
7      'venue_name': venue_name,
8      'venue_city': venue_city}
9
10     template = get_template('correoEvent.html')
11     content = template.render(context)
12
13     email = EmailMultiAlternatives(
14         'Information about the '+ type+' '+title,
15         'Codigo',
16         settings.EMAIL_HOST_USER,
17         [mail]
18     )
19
20     email.attach_alternative(content, 'text/html')
21     email.send()
```

A.1.4. Uso del paquete *re* para la validación de correos electrónicos

```python
import re

regex_domain = re.compile('''
    ( localhost | ( [a-z0-9]
        ([-\w]*                    # caracteres alfanuméricos, barras baja, puntos y apóstrofes
          [a-z0-9]
        )?
      \. # punto final
      )+ [a-z]{2,} )$
''', re.VERBOSE | re.IGNORECASE)

regex_body = re.compile('''
    ^(?!\.)                        # el nombre no puede empezar por punto
    ( [-a-z0-9!\#$%&'*+/=?^_`{|}~]
      |
      (?<!\.)\.
    )+
    (?<!\.)$                       # el nombre no puede acabar por punto
''', re.VERBOSE | re.IGNORECASE)

def email_validation(mail):
    if not isinstance(mail, str) or not mail or '@' not in mail:
        return False

    name, domain = mail.rsplit('@', 1)

    correct_domain = regex_domain.match(domain)
    correct_body = regex_body.match(name)

    if not correct_domain
        try:
            encoded_domain = domain.encode('idna').decode('ascii')
        except UnicodeError:
            return False

        correct_domain = regex_domain.match(encoded_domain)

    if (correct_body is not None) and (correct_domain is not None):
        return True
    else:
        return False
```

A.1.5. Uso del paquete *crypt* para el cifrado de contraseñas

```
import crypt

def insert_user(request):
    user = request.GET.get("user", "")
    passw=request.GET.get("passw", "")
    name=request.GET.get("name", "")
    last=request.GET.get("last", "")
    email=request.GET.get("email", "")
    country=request.GET.get("country", "")
    region=request.GET.get("region", "")

    passLength = len(passw)

    password_hash = crypt.crypt(passw)

    mongo = UserPersonalDetails()
    info = mongo.insert_user(user,password_hash,passLength,name,last,email,country
        ,region)

    if(info==1):
        send_email(email)

    return JsonResponse(info, safe=False)
```

A.1.6. Uso del paquete *crypt* para la comprobación de contraseñas

```
import crypt

def login_user(self,query,passw):
        numResultados = self.col.find({"user":query},{"_id":0}).count()
        if(numResultados > 0):
            resultadoUser = self.col.find({"user":query},{"_id":0})
            x = []
            for i in resultadoUser:
                x.append(i)
            password_hash = x[0]["password"]

            # To check the password.
            valid_password = crypt.crypt(passw, password_hash)

            result = self.col.find({"user":query,"password":valid_password}).count()
        else:
```

```
17          result=0
18          query=""
19          valid_password=""
20
21      return {'login':result, 'user':query, 'password':valid_password}
```

A.2. Frontend

A.2.1. Función action de la página de inicio

```
1  export const loadHome = (country,city) => async (dispatch) => {
2      let upcomingEvents = [];
3      let videoDetailData = [];
4      let artistTopArtists= [];
5      try{
6          dispatch({type:'LOAD_HOME_REQUEST'});
7          upcomingEvents = await axios.get(getEventsCityURL(city));
8          upcomingEvents = upcomingEvents.data;
9          {console.log(upcomingEvents)}
10     }
11     catch(error){dispatch({type:'LOAD_HOME_ERROR'})}
12     try{
13         dispatch({type:'LOAD_HOME_REQUEST'});
14         videoDetailData = await axios.get(topMusicVideosYoutubeURL(country));
15         videoDetailData = videoDetailData.data;
16         {console.log(videoDetailData)}
17     }
18     catch(error){dispatch({type:'LOAD_HOME_ERROR'})}
19     try{
20         dispatch({type:'LOAD_HOME_REQUEST'});
21         artistTopArtists= await axios.get(topArtistsSpotifyURL(country));
22         artistTopArtists = artistTopArtists.data;
23         {console.log(artistTopArtists)}
24     }
25     catch(error){dispatch({type:'LOAD_HOME_ERROR'})}
26
27     dispatch({
28         type: 'LOAD_HOME',
29         payload: {
30             upcomingEvents: upcomingEvents,
31             topArtists: artistTopArtists,
32             topVideos: videoDetailData,
33             loading:false,
34             error: true
```

```
35         }
36     });
37 };
```

A.2.2. Función reducer de la página de inicio

```
1  const HomeReducer = (state=initialState,action) => {
2      switch(action.type){
3          case "LOAD_HOME_REQUEST":
4              return{
5                  ...state,
6                  topVideos: [],
7                  topArtists: [],
8                  upcomingEvents:[],
9                  loading: true,
10                 error:false
11         };
12         case "LOAD_HOME":
13             return{
14                 ...state,
15                 topVideos: action.payload.topVideos,
16                 topArtists: action.payload.topArtists,
17                 upcomingEvents: action.payload.upcomingEvents,
18                 loading: false,
19                 error:false
20         };
21         case "LOAD_HOME_ERROR":
22             return{
23                 ...state,
24                 topVideos: [],
25                 topArtists: [],
26                 upcomingEvents: [],
27                 loading:false,
28                 error:true
29         };
30         default:
31             return{...state};
32     }
33 }
```

María Begoña Martínez Martín, Camila Belén Pérez Alániz y Paula Piñuela Monjas
Junio 2021
Ult. actualización 15 de junio de 2021
LaTeX lic. LPPL & powered by TEFLON CC-ZERO

Esta obra está bajo una licencia Creative Commons "CC0 1.0 Universal".