UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

SpotMyFM
Documentación Técnica

Presentado por Jorge Ruiz Gómez
en Universidad de Burgos — 7 de julio de 2022
Tutor: Bruno Baruque Zanón

Índice general

Índice de figuras

Índice de tablas

Apéndice A

Plan de Proyecto Software

A.1. Introducción

En este anexo ser va a realizar un análisis de la planificación temporal junto con la viabilidad legal y económica del proyecto.

A.2. Planificación temporal

Introducción

Para la planificación temporal se ha utilizado la metodología Scrum. El proyecto ha tenido dos fases muy diferenciadas. La primera fase está comprendida entre el Sprint 1 (13/10/2021) y el Sprint 10(04/03/2022), y se centra en el diseño y desarrollo del servicio web. La segunda fase comprendida entre el Sprint 11 (05/03/2022) y el Sprint Final (07/07/2022) se centra en la investigación, desarrollo e implementación de un sistema de recuperación de información musical.

Estas dos fases se pueden diferenciar en las contribuciones del proyecto Fig.A.1, ya que la segunda fase apenas tuvo impacto en el repositorio al desarrollarse desde Kaggle.

El proyecto se ha dividido en 5 Épicas:

- **Spotify Frontend:** Esta épica contiene todas las tareas relacionada con el gestor de biblioteca / playlists / álbumes de Spotify.

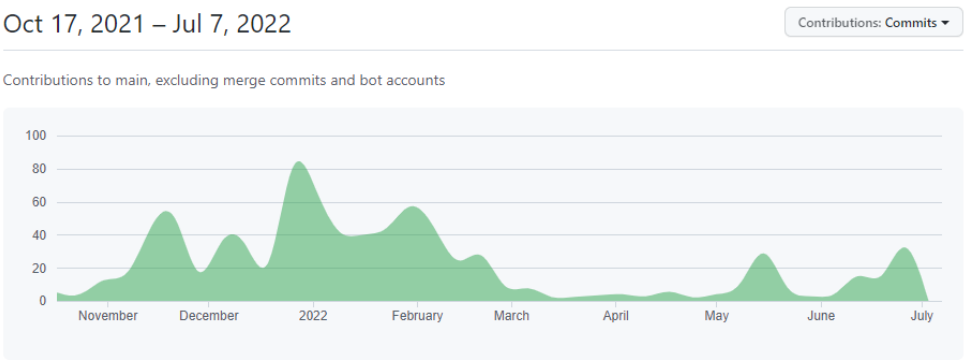


Figura A.1: Contribuciones del Proyecto

- **Database Frontend:** Esta épica contiene todas las tareas relacionadas con la interacción con la base de datos desde el Frontend. Además, se ha utilizado esta épica para la comunicacióon general con el backend.
- **Backend Principal:** Esta épica contiene todas las tareas relacionada con el Backend NextJS, como la autenticación, transacciones de la base de datos, SSR, o comunicación con el analizador de tareas.
- **Analizador de Canciones:** Esta épica contiene todas las tareas relacionadas con el analizador de canciones, desde la investigación de técnicas y modelos, entrenamiento y experimentación con modelos hasta la implementación del backend.

Sprints

Este apartado contine las distintas tareas organizadas por Sprints. No ha sido posible añadir una gráfica con el burndown debido a que Zenhub no permite generar el gráfico para los sprints anteriores a febrero.

Sprint 1

13/10/21 - 26/10/21

- Requisitos Funcionales. Coste inicial 1 punto, coste final 8 puntos.
- Requisitos No Funcionales. Coste inicial 2 puntos, coste final 2 puntos.
- Buscar proveedores de Cloud. Coste inicial 2, Coste final 2.

- Diagrama de casos de uso. Coste inicial 3 puntos, coste final 8 puntos.
- Crear guía de estilo. Coste inicial 1 punto, coste final de 1 punto.

Sprint 2

30/10/21 - 12/11/21

- Botón cambiar el Tema. Coste inicial 2, coste final 2.
- Diseño de la caché local. Coste inicial 2, coste final 2.
- Contexto de la API de Spotify, para que el cliente sea accesible desde todos los punto. Coste inicial de 2 puntos, coste final de 2 puntos.
- Rutas Protegidas. Wrapper que impide a un usuario sin ciertas características acceder a ciertas rutas (por ejemplo /admin). Coste inicial 1 punto, coste final de 1 punto.

Sprint 3

13/11/21 - 26/11/21

- Inicio de Sesión y Gestor de Cookies. Implementación del inicio de sesión y persistencia de credenciales en cookies. Coste inicial 8 puntos, coste final 8 puntos.
- Caché Local. Se ha creado una caché local con DexieDB. Coste inicial 3, coste final 3.
- Modal. Se ha creado un componente modal que contiene una vista anidad. Coste inicial 3, coste final 3.
- Track Card + Proxy de Carga. Se ha creado el componente carta que muestra una canción. Coste inicial 4, coste final 4.
- Vista Genérica de Cartas. Se ha creado una vista genérica de cartas que pueda representar de forma eficiente cualquier tipo de tarjeta (canciones, álbumes, playlists o artistas). Coste inicial 8+5, coste final 8+5.
- Clientes REST Spotify y Lastfm. Coste inicial 2, coste final 2.
- Selector de Canciones para Playlist. Coste inicial 3, coste final 3.
- Implementación de los dátos homogéneos de Spotify. Coste inicial 2, Coste final 2.

Sprint 4

27/11/21 - 10/12/21

- Actualizar la versión de Typescript. Este cambio obligó a refactorizar los *catch* ya que ahora necesitan estar tipados. Coste inicial 2, coste final 2.
- Configuración inicial de las acciones de Github para los tests de Jest.
- Vista detallada de canciones. Coste inicial 3, coste final 3.
- Inicio de los filtros avanzados.
- Inicio de la página de Inicio. Permite cargar las canciones y artistas favoritos del usuario.
- Vista de artistas. Coste inicial 3, coste final 3.
- Tarjeta de Artistas. Coste inicial 3, coste final 3.

Sprint 5

11/12/21 - 24/12/21

- Cambio en las proporciones de los botones para un diseño responsive. Coste inicial 1, coste final 1.
- Finalizado la integración de JEST con Github Actions. Coste inicial 2, coste final 2.
- Finalizados los filtros avanzados. Coste inicial 5, coste final 9.

Sprint 6

11/12/21 - 24/12/21

- Terminar Home Page, integrando las distintas vistas. Coste inicial 2, coste final 13.
- Caché Manager. Implementación de la fachada como Hook de react. Coste inicial 5, coste final 5.
- Mejorar el rendimiento de la fachada. Las peticiones se realizan en paralelo. Coste inicial

- Sistema de Notificaciones Genérico. Coste inicial 5, coste final 5.
- Caché de Notificaciones. Coste inicial 5, coste final 5.
- Vista de Playlist. Coste inicial 5, coste final 5.
- Implementación de JWT. Coste final 5, costes final 5.
- Implementación de Vista de Estadísticas. Coste inicial 8, coste final 8.
- Detalles de Playlist. Coste inicial 5, coste final 5.

Sprint 7

8/01/22 - 21/01/22

- Estado de la Fachada mediante un spinner. Coste inicial 2, coste final 2.
- Menú de navegación de Escritorio. Coste inicial 5. Coste final 8.
- Barra de navegación móviles. Coste inicial 5, coste final 5.
- Página de Opciones. Coste inicial 8. Coste final 8.
- Reproductor / Suena Ahora. Coste inicial 8, coste final 8.
- Selector de lenguaje. Coste inicial 4, coste final 4.

Sprint 8

22/01/22 - 04/02/22

- Cliente de la base de datos. Coste inicial 4, coste final 4.
- Implementación de transacciones de etiquetas. Coste inicial 3, coste final 3.
- Exposición de las etiquetas mediante API REST. Coste inicial 3, coste final 3.
- Documentar la especificación de la API mediante OpenAPI. Coste inicial 1, coste final 1.
- Test de las APIs mediante Cypress. Coste inicial 3, coste final 3.

- Solucionar el cambio de la URL de la API al cambiar el idioma. Coste inicial 1, coste final 1.
- Solucionar problemas de re-renderizado indeseados. Coste inicial 3, coste final 3.
- Vista de álbumes. Coste inicial 3, coste final 3.

Sprint 9

05/02/22 - 18/02/22

- Detalles de álbumes. Coste inicial 2, coste final 2.
- Gestor de Álbumes. Coste inicial 8, coste final 8.
- Editor de etiquetas de álbumes. Coste inicial 3, coste final 3.
- Investigar sobre técnicas de clasificación musical. Coste inicial 4, coste final 4.
- Demo de clasificador con Keras y Gtzan. Coste inicial 8, coste final 8.
- Página de Búsqueda. Coste inicial 5, coste final 5.
- Investigar técnicas y resultados de audio augmentation. Coste inicial 2, coste final 2.
- Ocular el fondo del modal con CSS. Coste inicial 1, coste final 1.

Sprint 10

19/02/22 - 04/03/22

- Entrenar GTZAN a partir de modelos preentrenados. Coste inicial 5, coste final 5.
- Añadir vistas anidadas en los detalles. Coste inicial 8, coste final 8.
- Desarrollo de una aplicación en Node que permita descargar canciones de la API de Spotify. Coste inicial 3, coste final 5.
- Extensión del conjunto de datos GTZAN con 200 canciones más por cada género. Coste inicial 5, coste final 8.

Sprint 11

12/03/22 - 01/04/22

- Normalización de Playlist mediante Node. Coste inicial 3, coste final 3.
- Modelado del Dataset que incluya subgéneros. Coste inicial 1, coste final 1.
- Obtención de las etiquetas de Discogs. Coste inicial 2, coste final 2.
- Investigar Gtzan con técnias de clustering mediante DB SCAN y Kmeans. Coste inicial 2, coste final 2.
- Herramienta que extraiga los MFCCs y los almacene en fichero .npy. Coste inicial 2, coste final 2.
- Creación del Ludwig Dataset. Coste inicial 13, coste final 13.

Sprint 12

2/04/22 - 22/04/22

- Implementación de una OVA para mejorar el resultado del clasificador en Ludwig. Coste inicial 13, coste final 13.
- Eliminar Reggae como género principal. Coste inicial 1, coste final 1.
- Implementar una arquitectura OVO de 32 CNNs para mejorar el rendimiento del clasificador. Coste inicial 3, coste final 8.
- Implementar Adaboost para mejorar el rendimiento. Coste inicial 3, coste final 3.
- Clasificar la salida de CNN mediante SVM para mejorar el rendimiento. Coste inicial 3, coste final 3.
- Entrenar todos los subgéneros. Coste inicial 5, coste final 5.
- Entrenar una OVA como clasificador de estados de ánimo. Coste inicial 8, coste final 8.
- Iniciar el Backend de MIR. Coste inicial 2.

Sprint 13

23/04/22 - 06/05/22

- Implementar el motor de inferencia basado en ONNX. Coste inicial 5, coste final 5.
- Terminar el Backend de MIR. Coste final 8.
- Mejorar el rendimiento mediante inferencia en bloque. Coste inicial 3, coste final 3.
- Configurar la integración continua de GCP Run par que publique el Backend MIR. Coste inicial 2, coste final 2.
- Sistema de recomendación basado en contenidos. Coste inicial 8, coste final 8.
- Sistema de recomendación colaborativo. Coste inicial 8, coste final 8.

Sprint 14

07/05/22 - 20/05/22

- Integración Backend MIR con Frontend. Coste inicial 12, coste final 16.
- Cuantización de Modelos para reducir el consumo de memoria. Coste inicial 1, coste final 1.
- Internacionalización completa al castellano. Coste inicial 8, coste final 8.

Sprint 15

Sprint 14

21/05/22 - 03/06/22

- Desarrollo de los conceptos teóricos. Coste inicial 8, coste final 8.
- Protección del backend MIR mediante un código secreto. Coste inicial 3, coste final 3.
- Actualización de Estadísticas para incluir estados de ánimo. Coste inicial 3, coste final 3.

Sprint Final

03/06/22 - 07 / 07 / 22

- Documentación de técnicas y herramientas. Coste inicial 8, coste final 8.
- Documentación de aspectos relevantes. Coste inicial 8, coste final 8.
- Documentación de Objetivos. Coste inicial 1, coste final 1.
- Documentación de Introudcción y Abstract. Coste inicial 1, coste final 1.
- Creación de Docker Compose. Coste inicial 4, coste final 4.
- Crear página de inicio. Coste inicial 4, coste final 4.
- Finalizar anexos de diseño y requisitos. Coste inicial 8, coste final 8.
- Crear una vista de sistema mediante Figma. Coste inicial 2, coste final 2.
- Documentación del Plan de Proyecto. Coste inicial 8, coste final 8.

A.3. Estudio de viabilidad

En este apartado se va realizar un estudio sobre la viabilidad económica y legal del proyecto, teniendo en cuenta distintos apartados como los

Viabilidad económica

El proyecto no es económicamente viable ya que los términos y condiciones de Spotify [5] y LastFM [1]. Estos servicos solo permiten utilizar la API pública para **uso personal**, por lo que no se puede monetizar.

Costes

En esta sección se van a desglosar y analizar los distintos costes del proyecto.

Costes de Hardware

Los únicos costes hardware son un ordenador portátil junto con sus periféricos.

Suponiendo que la vida útil de un ordenador portátil son 6 años podemos estimar su coste amortizado durante el primer año. Estos costes están desglosados en A.1

Concepto	Coste (€)	Amortización (€)
Portátil y Periféricos	850	631
Total	850	631

Tabla A.1: Costes Hardware

Costes Fijos de Software

En este apartado A.3 se van a analizar los costes anuales de los distintos servicios y suscripciones.

Concepto	Coste (€)	Amortización (€)
Vercel[1]	240	163.7
Codacy	170	113
Github Premium	48	32.2
Github Copilot	120	163.7
Total	578	488.4

Tabla A.2: Costes Fijos

Costes por Usuario

Según los análisis de Google Cloud Platform, el coste medio por usuario por usuario para cada uno de los servidores webs son unos 13.26 céntimos al mes, siendo el servidor web de análisis de canciones el más cara de mantener por los altos tiempos de ejecución y consumos de memoria.

AWS DynamoDB tiene un coste mensual de 0.072 céntimos por cada unidad de lectura y escritura en la base de datos, y un coste de 0.283 céntimos por cada GB/mes. Se estima que es necesaria 1 unidad de lectura y escritura por cada 10000 usuarios. Cada canción almacenada tiene un tamaño medio de 412 Bytes y cada usuario almacena de media 1.2KB en etiquetas.
Serían necesarios 200 millones de usuarios (más usuarios que suscriptores de Spotify [7]) o 625 millones de canciones almacenadas. Como es prácticamente

imposible alcanzar estos números de usuarios o canciones, se ha decidido obviar el coste del almacenamiento debido a que el uso comercial entra dentro del plan gratuito de AWS.

Podemos estimar que el coste por usuario es de unos **14 céntimos al mes**. Además, a medida que se analicen más canciones, es posible que el coste se vea reducido debido al aumento de lecturas en la base de datos y el menor uso del analizador de canciones.

Costes Personal

Para este coste se va a tener en cuenta el coste de un único empleado encargado del desarrollo, mantenimiento y soporte de usuario de la página web hasta que se decida dejar de dar soporte al servicio.

En este caso se ha escogido un suelo base bruto de 21.000 euros, considerando el salario de un desarrollador junior fullstack sin apenas experiencia, que ha sido desglosado en A.3

Concepto	Coste (€)
Salario Mensual Neto (12 pagas)	1.425,4
Retenciones IRPF	2.562,2
Cuotas Seguridad Social	1.333,5
Total	21.000,0

Tabla A.3: Desglose Coste anual del Webmaster

Otros Costes

La tabla A.3 contiene otros costes que han aparecido a lo largo del proyecto.

Concepto	Coste (€)
3 USBs	14,5
Memoria	23,5
Total	38,0

Tabla A.4: Coste anual del Webmaster

Coste Primer Año

La tabla A.3 contiene el coste del proyecto durante el primer año.

Concepto	Coste (€)
Salarios	21.000,0
Licencias	488,4
Hardware	631
20 Usuarios	3,33
Total	22122.73

Tabla A.5: Coste durante el primer año

Viabilidad legal

En esta sección se va a analizar las distintas licencias de las dependencias, así como términos y condiciones de las APIs que hemos utilizado.

Api de Spotify

La api de Spotify detalla sus términos y condiciones en [6]. Los términos más importantes están reflejados en la tabla A.3.

Condición	Estado
Uso No Comercial	✓
No se almacenan datos de Spotify que puedan quedarse obsoletos	✓
Los resultados tienen en cuenta el mercado del usuario / no permite a usuarios saltarse restricción geográficas.	✓
La plataforma cumple con las recomendaciones de la guía de diseño de Spotify	✓
No se modifican los metadatos de Spotify	✓
No se utiliza la API para uso malicioso	✓
No se daña la imagen de Spotify	✓
Incumplimiento de la propiedad intelectual	✓
Se oculta la implementación de características *Premium* a usuarios que no son suscriptores	✓
No se cachean los resultados de la API	~

Tabla A.6: Términos y Condiciones más importantes de la API de Spotify

El punto más interesante es la sección IV apartado 3b, que prohíbe el uso de caches. Si detallamos este punto se puede observar como únicamente se permite el uso de cachés si **se utilizan para el rendimiento**, **son temporales**. SpotMyFM cumple con ambos casos de uso.

Api de LastFM

Los términos y condiciones de la API pública de LastFM pueden encontrarse en [1]. En este caso, se pueden resaltar los términos y condiciones en la siguiente tabla A.3.

Condición	Estado
Se acredita el origen de los datos a LastFM	✓
Uso No Comercial	✓
No se licencian los datos de LastFM	✓
No se modifican los datos de LastFM de manera que dañe la imagen de marca[2]	✓

Tabla A.7: Términos y Condiciones más importantes de la API de LastFM

En este caso se han cumplido con todos los requisitos que especifican los términos y condiciones de la API.

Licencias Software

Una licencia software es una definición legal vinculante que indica los límites y condiciones para el uso de cada una de las dependencias. Cada dependencia puede tener un tipo de licencia distinto, por lo que es necesario revisar si algunas de las licencias es incompatible con el proyecto.

Las licencias de cada una de las dependencias han sido detalladas en A.8. Todas las licencias admiten los distintos usos que va a tener el producto.

Dependencia	Licencia
Babel	MIT
Cypress	MIT y Apache
Jest	MIT
npmcli	ISC
popperjs	MIT
@uiball/loader	MIT
axios	MIT
base64	MIT
dexie	Apache 2.0
dotenv	BSD-2-Clause
Framer Motion	MIT
js-cookies	MIT
jsonwebtoken	MIT
lodash	MIT
next	MIT
next-translate	MIT
pretty-ms	MIT
react-device-detect	MIT
react-dom	MIT
react-icons	MIT
react-switch	MIT
react-is	MIT
react-toastify	MIT
react-use	MIT
recharts	MIT
spotify-web-api-js	MIT
spotify-web-api-node	MIT
styled-components	MIT
tiny-async-pool	MIT
twin.macro	MIT
estlint	MIT
ONNX	Apache 2.0
FastAPI	MIT
Tensorflow	Apache 2.0
Python	PSFN[3]

Tabla A.8: Licencias

Apéndice B

Especificación de Requisitos

B.1. Introducción

Este anexo recoge los distintos requisitos del proyecto. Está organizado en dos apartados:

- Catálogo de Requisitos: Recoge los requisitos funcionales y no funcionales del proyecto.

- Casos de Uso: Detallan la visión del proyecto y permiten realizar una implementación de la aplicación.

B.2. Objetivos generales

1. Diseñar e implementar un servicio web que extienda la capacidades de un servicio ya existente.

2. Diseñar e implementar una arquitectura que interconecte múltiples servicios.

3. Diseñar e implementar un servicio que permita analizar y recomendar canciones a partir de un fichero de audio musical.

B.3. Catálogo de Requisitos

Requisitos Funcionales

- **RF-1 Gestión de Biblioteca:** Se requiere que la web sea capaz de mostrar la biblioteca de música personal del usuario.
  - **RF-1.1 Listar Biblioteca:** Se requiere que el usuario pueda visualizar su biblioteca de usuario
    - **RF-1.1.1 Filtrar Vista:** Se requiere que el usuario pueda filtrar la vista de su biblioteca mediante filtros avanzados.
    - **RF-1.1.2 Seleccionar Vista:** Se requiere que el usuario pueda marcar como seleccionados los elementos filtrados.
    - **RF-1.1.3 Ordenar Vista:** Se requiere que el usuario pueda ordenar la vista actual a partir de parámetros.
    - **RF-1.1.3 Detallar Canción:** Se requiere que el usuario pueda ver detalles de cualquier item de la vista.
  - **RF-1.2 Crear Playlist:** Se requiere que el usuario pueda crear playlists a partir de una selección de canciones.
    - **RF-1.2.1 Configurar Playlist:** Se requiere que el usuario pueda escoger el título, descripción y opciones de privacidad de cada playlist.
    - **RF-1.2.2 Dividir Playlist:** Se requiere que el usuario pueda dividir una playlist en múltiples playlists.
    - RF-1.2.3 Expandir con Recomendaciones: Se requeire qu el usuario pueda añadir a la playlist recomendaciones.
  - **RF-1.3 Ampliar Playlists:** Se requiere que el usuario pueda ampliar sus playlists ya creadas a partir de una selección de canciones.
    - **RF-1.3.1 Buscar Playlist:** Se requiere que el usuario pueda seleccionar su playlist a partir de un listado.
    - **RF-1.3.2 Detallar Playlist:** Se requiere que el usuario pueda ver los detalles de una playlist.
  - **RF-1.4 Detallar Playlist:** Se requiere que el usuario pueda ver los detalles de su playlist.
- **RF-2 Gestión de Álbumes:** Se requiere que la web sea capaz de mostrar los álbumes del usuario.
  - **RF-2.1 Listar Álbumes:** Se requiere que el usuario pueda visualizar sus álbumes.

        - **RF-2.1.1 Álbumes Favoritos:** Se requiere que el usuario pueda visualizar sus álbumes marcados como favoritos.
        - **RF-2.1.2 Álbumes Etiquetados:** Se requiere que el usuario pueda visualizar sus álbumes etiquetados.
        - **RF-2.1.3 Filtrar Vista:** Se requiere que el usuario pueda filtrar la vista actual de sus álbumes mediante filtros avanzados.
        - **RF-2.1.4 Ordenar Vista:** Se requiere que el usuario pueda ordenar la vista actual a partir de parámetros.
        - **RF-2.1.5 Detallar Álbum:** Se requiere que el usuario pueda ver detalles de cualquier álbum en la vista.
    - **RF-2.2 Buscar Álbumes:** Se requiere que el usuario pueda visualizar álbumes a partir de una cadena.
    - **RF-2.3 Gestionar Álbumes Favoritos:** Se requiere que el usuario pueda marcar o desmarcar cualquier álbum como favorito.
    - **RF-2.4 Etiquetar Álbumes:** Se requiere que el usuario Añadir etiquetas a cualquier álbum.
        - **RF-2.4.1 Etiquetas Personalizadas:** Se requiere que el usuario pueda crear sus propias etiquetas.
- **RF-3 Detallar Elementos:** Se requiere que el usuario pueda ver detalles de un álbum, playlist o seleccionado por el usuario.
    - **RF-3.1 Detallar Canción:** Se requiere que el usuario pueda ver detalles de una canción específica.
        - **RF-3.1.1 Previsualizar Canción:** Se requiere que el usuario pueda escuchar un fragmento de la canción.
        - **RF-3.1.2 Analizar Canción:** Se requiere que el usuario pueda obtener detalles especiales a partir de un análisis de un fragmento de la canción.
        - **RF-3.1.3 Reproducir Canción:** Se requiere que el usuario pueda reproducir o añadir a la cola una canción en un cliente de Spotify.
        - **RF-3.1.4 Detallar Álbum:** Se requiere que el usuario obtenga los detalles del álbum al que pertenece dicha canción.
    - **RF-3.2 Detallar Álbum:** Se requiere que el usuario pueda ver detalles de un álbum específico.
        - **RF-3.2.1 Gestionar Etiquetas:** Se requiere que el usuario pueda visualizar o gestionar las etiquetas de un álbum

        - **RF-3.2.2 Conocer Estadísticas:** Se requiere que el usuario pueda ver las estadísticas del álbum mediante LastFM.
        - **RF-3.2.3 Reproducir Álbum:** Se requiere que el usuario pueda reproducir el álbum en un cliente Spotify.
        - **RF-3.2.3 Detallar Artistas:** Se requiere que el usuario pueda conocer los detalles de los artistas que han participado en el álbum.
    - **RF-3.3 Detallar Playlist:** Se requiere que el usuario pueda conocer los detalles de una playlist.
        - **RF-3.3.1 Detallar Canciones** Se requiere que el usuario pueda explorar las canciones de una playlist.
    - **RF-3.4 Detallar Artistas:** Se requiere que el usuario pueda conocer los detalles de un artista en específico.
        - **RF-3.4.1 Géneros Musicales** Se requiere que el usuario pueda conocer los géneros musicales de un artista.
- **RF-4 Gestión de Tema:** Se requiere que el usuario pueda cambiar el tema actual en cualquier punto de la web.
    - **RF-4.1 Tema Persistente:** Se requiere que el tema seleccionado se mantenga entre sesiones.
- **RF-5 Gestión de Sesión:** Se requiere que el usuario tenga el control de la sesión actual.
    - **RF-5.1 Cerrar Sesión:** Se requiere que el usuario pueda cerrar su sesión desde la web.
        - **RF-5.1.1 Limpieza de Sesión:** Se requiere que todos los datos locales sean borrados al cerrar sesión.
    - **RF-5.2 Limpiar Datos Locales:** Se requiere que los datos locales puedan borrarse desde la web.
- **RF-5 Estadísticas:** Se requiere que el usuario pueda visualizar las estadísticas relacionadas con su cuenta.
    - **RF-5.1 Listar Canciones**: Se requiere que el usuario pueda conocer sus canciones más escuchadas en diversos periodos de tiempo.
    - **RF-5.2 Listar Artistas**: Se requiere que el usuario pueda conocer sus artistas más escuchadas en diversos periodos de tiempo.
- **RF-6 Cachear Peticiones:** Se requiere que la aplicación realice el mínimo número de peticiones a la API para no sobrepasar los límites.

  - **RF-6.1 Cachear Canciones:** Se requiere que los datos de las canciones se almacenen de forma local.
  - **RF-6.2 Cachear Artistas:** Se requiere que los datos de los artistas se almacenen de forma local.
  - **RF-6.3 Cachear Álbumes:** Se requiere que los datos de los álbumes se almacenen de forma local.
  - **RF-6.4 Refrescar Caché:** Se requiere que los distintos datos almacenados puedan actualizarse para evitar inconsistencias.
- **RF-7 Analizar Canciones:** Se requiere que el usuario pueda conocer detalles de cada una de las canciones a partir de un fragmento de la canción.

Requisitos no Funcionales

- **RNF-1 Diseño Responsive:** Se requiere un diseño responsive"para poder utilizarla en dispositivos con distintos tamaños de pantalla o relaciones de aspecto sin perder información.
- **RNF-2 Minimizar Peticiones:** Se requiere minimizar el número de peticiones posibles a las distintas APIs para evitar alcanzar los límites de cada API.
- **RNF-3 Internacionalización:** Se requiere que la web esté disponible en, al menos, dos idiomas.
- **RNF-4 Compatibilidad:** Se requiere que la web sea funcional a lo largo de los motores webs más utilizados (Chromium, Firefox y Apple Webkit).
- **RNF-5 Carga Diferida:** Se requiere que la web evite cargar un exceso de datos si el usuario no va a necesitarlos, por ejemplo aplicando el patón de Carga Diferida para paginar la carga de recursos remotos.
- **RNF-6 Accesibilidad:** Se requiere que la web cumpla con el mayor número de recomendaciones posible de Web Content Accessibility Guidelines (WCAG) 2.1
- **RNF-7 Rendimiento y Buenas Prácticas:** Se requiere que la web funcione correctamente en dispositivos móviles y en escritorio, minimizando el tamaño de la aplicación para acelerar las cargas de JavaScript y evitar las esperas en el navegador.

- **RNF-8 Seguridad:** Se requiere que la web sea segura, utilizando TSL o SSL.
- **RNF-9 Privacidad:** Se requiere almacenar el mínimo de información que puede identificar a un usuario para mejorar la privacidad de los datos.

B.4. Especificación de requisitos

Esta sección contiene el diagrama de casos de uso y la especificación individual de los casos de uso más importantes.

Actores

Únicamente hay un actor, el usuario final.

Diagrama de Casos de Uso

El diagrama de casos de uso únicamente contiene un actor, y está dividido en dos imágenes.

Figura B.1: Casos de Uso - Parte 1

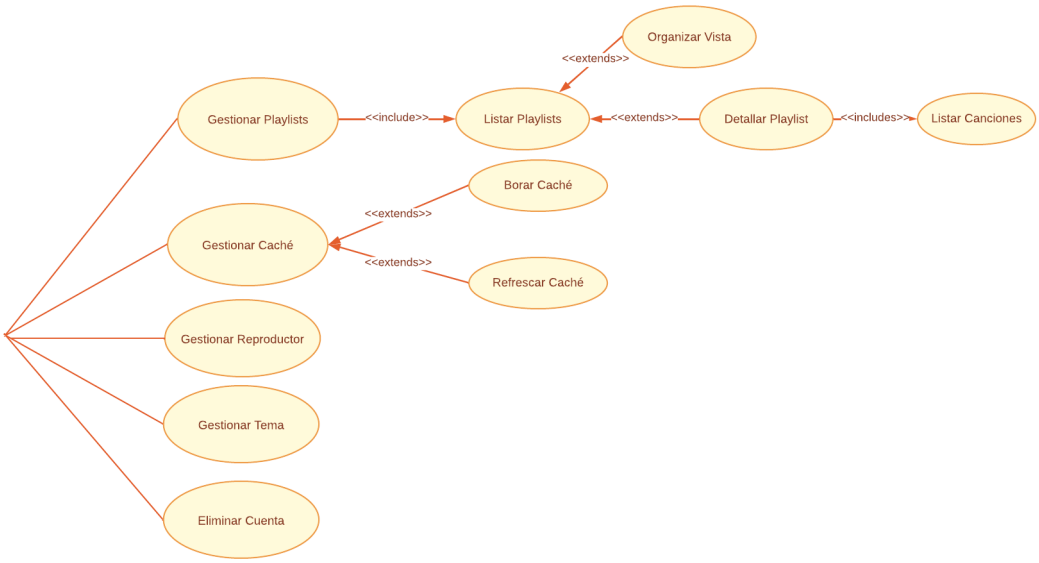


Figura B.2: Casos de Uso - Parte 2

Casos de Uso

En este apartado se van a detallar los distintos casos de uso.

CU-01	**Gestionar Tema**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-4
Descripción	Permite al usuario cambiar el tema general de la página
Precondición	Ninguna
Acciones	▪ El Usuario entra en la página. ▪ La web carga el tema almacenado. ▪ El usuario pulsa el botón de cambio de tema. ▪ El Tema se alterna. ▪ La selección del tema se almacena de forma local.
Postcondición	El tema ha cambiado
Excepciones	Ninguna
Importancia	Baja

Tabla B.1: CU-01

CU-02	**Gestionar Estadísticas**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-5
Descripción	Permite al usuario conocer de manera visual estadísticas calculadas a partir de la vista de canciones actual. Estas estadísticas incluyen los géneros y estados de ánimo favoritos, evolución de géneros musicales o gráficos con la actividad de la cuenta del usuario.
Precondición	El usuario ha iniciado sesión.
Acciones	▪ El Usuario abre una vista de canciones. ▪ El usuario espera a que se carguen las canciones. ▪ El usuario pulsa el botón .Estadísticas". ▪ El usuario visualiza las estadísticas de la vista actual.
Postcondición	Ninguna
Excepciones	Ninguna
Importancia	Baja

Tabla B.2: CU-02

CU-03	**Gestionar Caché**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-6.4 y RF-5.2
Descripción	Permite al usuario tener control sobre la caché local
Precondición	El usuario ha iniciado sesión
Acciones	- El Usuario entra en la página. - El usuario va a a la sección de opciones. - El usuario accede al apartado *CachéLocal*. - El usuario selecciona entre *Borrar* o *RefrescarCaché*.
Postcondición	Aparece una notificación con el estado de la operación.
Excepciones	Ninguna
Importancia	Baja

Tabla B.3: CU-03

CU-04	**Eliminar Cuenta**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RNF-4
Descripción	Permite al usuario eliminar todos los datos asociados con su usuario de Spotify en la base de datos de SpotMyFM.
Precondición	El usuario ha iniciado sesión
Acciones	▪ El Usuario entra en la página. ▪ El usuario va a a la sección de *opciones*. ▪ El usuario accede al apartado *Cuenta*. ▪ El usuario selecciona *EliminarCuenta*. ▪ El usuario confirma la operación.
Postcondición	Aparece una notificación con el estado de la operación.
Excepciones	Ninguna
Importancia	Media

Tabla B.4: CU-04

CU-05, CU-06, CU-07	**Gestionar Biblioteca/Albumes/Playlists**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-1, F-1.3, RF-1.4, RF-2
Descripción	Permite al usuario explorar su biblioteca de canciones / álbumes / playlists / artistas desde la aplicación, con funcionalidades extendidas.
Precondición	El usuario ha iniciado sesión
Acciones	▪ El Usuario entra en la página. ▪ El usuario va a a la sección *GestionarBiblioteca* desde la barra de navegación. ▪ El usuario descarga su biblioteca localmente.
Postcondición	Se elimina el bloqueo y están disponibles todas las canciones del usuario
Excepciones	Ninguna
Importancia	Alta

Tabla B.5: CU-05, CU-06 y CU-07

CU-08	**Organizar Vista**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-1.1.1, RF-1.1.3, RF-2.1.3, RF-2.1.4
Descripción	Permite al usuario filtrar u ordenar una vista.
Precondición	El usuario ha iniciado sesión. Estamos trabajando con una vista de canciones, álbumes, artistas o playlists.
Acciones	▪ El usuario abre una vista. ▪ El usuario pulsa sobre un desplegable con el que puede seleccionar un atributo por el que ordenar. ▪ El usuario escoge un atributo y se refresca la vista con el nuevo orden. ▪ El usuario escribe en un campo. ▪ La vista muestra únicamente los elementos cuyos atributos coinciden con los valores del campo.
Postcondición	Ninguna
Excepciones	Ninguna
Importancia	Baja

Tabla B.6: CU-08

CU-09	**Listar**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-1.1, RF-1.3.2 RF-2.1, RF-5.1, RF-5.2
Descripción	Permite al usuario listar todas las canciones/artistas/álbumes/playlist.
Precondición	El usuario ha iniciado sesión.
Acciones	▪ El usuario abre una vista a partir de un gestor u otra biblioteca anidada. ▪ Si se realiza desde un móvil aparece una lista con canciones. Si no, aparecen tarjetas con la carátula del álbum de cada canción en grande. ▪ Cada elemento en la lista se puede detallar. Al detallar un elemento se abre un Modal con los detalles. ▪ Listar Canciones: Se puede acceder a esta vista a partir de la gestión de la biblioteca B.5, así como los detalles de un álbum (canciones que componen un álbum) o los detalles de una playlist (canciones que componente una playlist) ▪ Listar Álbumes: Se puede acceder a esta vista a partir de los detalles de un artista (álbumes que ha publicado un artista) y la gestión de los álbumes de usuario B.5 ▪ Listar Playlists: Se puede acceder a esta vista desde el gestor de playlists. B.5 ▪ Listar Artistas: Se puede acceder a esta vista desde los detalles de un álbum.
Postcondición	Ninguna
Excepciones	Hay un fallo en la comunicación con Spotify.
Importancia	Alta

Tabla B.7: CU-09

CU-10, CU-10.1, CU-10.2	**Seleccionar Canciones**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-1.2
Descripción	Permite al usuario seleccionar una serie de canciones que pueden ser utilizada para la creación o ampliación de playlists.
Precondición	El usuario ha iniciado sesión. Estamos trabajando con una vista de canciones. B.7
Acciones	▪ El usuario abre una con canciones. ▪ El usuario pulsa sobre el botón *AñadiralaPlaylist*. ▪ El botón cambia su etiqueta a *EliminardelaPlaylist* ▪ Un menú para configurar la playlist es visible. **Ampliar Playlist** • El usuario selecciona una playlist de la lista. • El usuario añade las canciones a la playlist. **Crear Playlist** • El usuario introduce el nombre de la playlist • El usuario introduce el numero de playlists que desea generar (un número mayor que 0) • El usuario pulsa sobre el botón crear.
Postcondición	No hay canciones seleccionadas y aparece un mensaje confirmado el estado de la creación.
Excepciones	Hay un fallo con la API de Spotify.
Importancia	Media

Tabla B.8: CU-10

CU-10.3	**Recomendar Canciones**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RNF-1.2.3
Descripción	Permite al usuario conseguir una lista con canciones similares.
Precondición	El usuario ha seleccionado una o más canciones
Acciones	▪ El usuario pulsa sobre el botón *Recomendaciones*. ▪ Se abre una nueva vista con nuevas canciones. ▪ El usuario selecciona una o varias canciones. ▪ El usuario vuelve a lista anterior.
Postcondición	Se han añadido las nuevas canciones a la vista original.
Excepciones	Hay un fallo en la comunicación con el servidor de canciones.
Importancia	Baja

Tabla B.9: CU-10.3

CU-11	**Modificar Etiquetas**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-2.4
Descripción	Permite al usuario modificar las etiquetas de un álbum.
Precondición	El usuario ha detallado un álbum.
Acciones	▪ El usuario pulsa sobre el botón *EditarEtiqutetas*. ▪ Se abre una nueva vista con una lista de etiquetas y un campo de texto. ▪ El usuario pulsa sobre una etiqueta para eliminarla. ▪ El usuario introduce una nueva etiqueta desde el campo de texto. ▪ El usuario guarda los cambios.
Postcondición	Se ha notificado el estado de la operación.
Excepciones	Hay un fallo en la comunicación con el servidor/base de datos.
Importancia	Baja

Tabla B.10: CU-11

CU-12	**Gestionar Reproductor**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-3.1.4, RD-3.2.3
Descripción	Permite al usuario controlar la canción que está sonando
Precondición	El usuario tiene una sesión activa de Spotify.
Acciones	▪ El usuario pulsa sobre el botón Reproducir Canción / Álbum. ▪ Se modifica la canción actual. ▪ El usuario decide si desea Pausar, Avanzar o Retroceder la cola de canciones.
Postcondición	Se ha notificado el estado de la operación.
Excepciones	Hay un fallo en la comunicación con Spotify.
Importancia	Baja

Tabla B.11: CU-12

CU-13	**Detallar Canción**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-3.1
Descripción	Permite al usuario visualizar todos los atributos de una canción. Esta vista incluye datos como el número de reproducciones de la canción, botón de reproducción, géneros, subgéneros y estados de ánimo[1]. Además, contiene todos los detalles del álbum.
Precondición	El usuario tiene una sesión activa de Spotify y está en una lista de canciones.
Acciones	▪ El usuario pulsa sobre el botón (+) ▪ Se abre un modal con los detalles.
Postcondición	Se ha notificado el estado de la operación.
Excepciones	Hay un fallo en la comunicación con Spotify.
Importancia	Alta

Tabla B.12: CU-13

CU-14	**Detallar Álbum**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-3.2
Descripción	Permite al usuario visualizar todos los atributos de un álbum. Esta vista incluye una pequeña descripción del álbum, sus etiquetas de usuario, un botón para poder editar las etiquetas de usuario y una lista con todos los artistas B.7 que han participado en el álbum. Además, se pueden listar todas las canciones que forman el álbum B.7.
Precondición	El usuario tiene una sesión activa de Spotify y está en una lista de álbumes.
Acciones	▪ El usuario pulsa sobre el botón (+) ▪ Se abre un modal con los detalles.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Alta

Tabla B.13: CU-14

CU-15	**Detallar Playlist**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-3.3
Descripción	Permite al usuario visualizar todos los atributos de una playlist. Esta vista incluye el tipo de playlist, autor, y una lista anidada con las canciones de usuario B.7.
Precondición	El usuario tiene una sesión activa de Spotify y está en una lista de playlist.
Acciones	▪ El usuario pulsa sobre el botón "Detalles". ▪ Se abre un modal con los detalles.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Alta

Tabla B.14: CU-15

CU-16	**Detallar Artista**
Versión	1.0
Autor	Jorge Ruiz Gómez
Requisitos	RF-3.4
Descripción	Permite al usuario visualizar todos los atributos de un artista de Spotify. Los detalles incluyen los géneros del artista, así como un listado con todos los álbumes que ha publicado el artista B.7.
Precondición	El usuario tiene una sesión activa de Spotify y está en una lista de artistas.
Acciones	▪ El usuario pulsa sobre el botón "Detalles". ▪ Se abre un modal con los detalles.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Alta

Tabla B.15: CU-16

Apéndice C

Especificación de diseño

C.1. Introducción

En este apartado se van a documentar las decisiones de diseño más relevantes del proyecto, como el diseño de datos o el diseño de clases de algunos componentes del sistema.

C.2. Diseño de Datos

Para el diseño de datos se ha decidido utilizar el modelo relacional debido a su similitud con los esquemas clave-valor que se utilizan en la base de datos basadas en documentos.

DynamoDB

Se ha utilizado la base de datos NOSQL basada en documentos DynamoDB.
La base de datos contiene 5 tablas, de las cuales 2 son versiones destinadas a ejecutar los tests / desarrollo. Cada uno de los elementos de la base de datos contiene dos atributos obligatorios, *updated_at* y *created_at*, estos atributos son strings que almacenan una fecha en formato ISO 8601.

LudwigDataset

Esta tabla contiene todos los items del conjunto de datos Ludwig C.1. La clave primaria (o clave de partición en DynamoDB), se identifica como PK, y es un string que almacena el id de la canción en Spotify. Por otro lado,

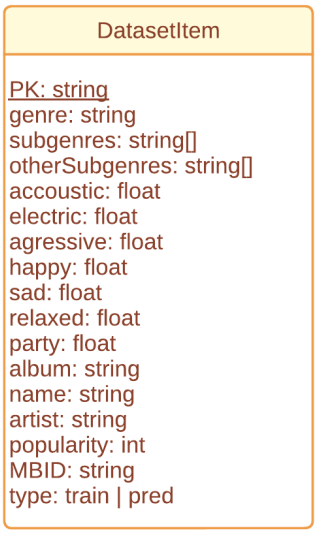


Figura C.1: DynamoDB: Ludwig Dataset

se almacena el $MBID$ de Brainz, un valor único en toda la tabla. Discogs [8] tiene muchos más subgéneros que los listados en la memoria, por ello se almacenan el resto de subgéneros que no se van a utilizar para entrenar los clasificadores en el campo $otherSubgenres$. Los estados de ánimo almacenan un valor entre 0 y 1 que indican la confianza de ese estado de ánimo.

La obtención de este dataset se realiza siguiendo el proceso indicado en el diagrama de la figura C.2

Tracks y Tracks_test

La tabla Tracks C.3 contiene los resultados del clasificador de géneros. El identificador de la tabla, PK, almacena el id de Spotify de una canción. Esta tabla contiene un atributo $version$, que permite identificar la versión del clasificador, ya que si reemplazamos el modelo principal, es posible los resultados almacenados y los resultados del clasificador sean distintos. La implementación de SpotMyFM obliga a DynamoDB a indexar $version$ para que únicamente se puedan obtener los resultados de la versión actual del clasificador.

Users y Users_test

La tabla Users C.4 contiene la información de los usuarios que han utilizado la función de etiquetas de SpotMyFM. Debido a la limitación de

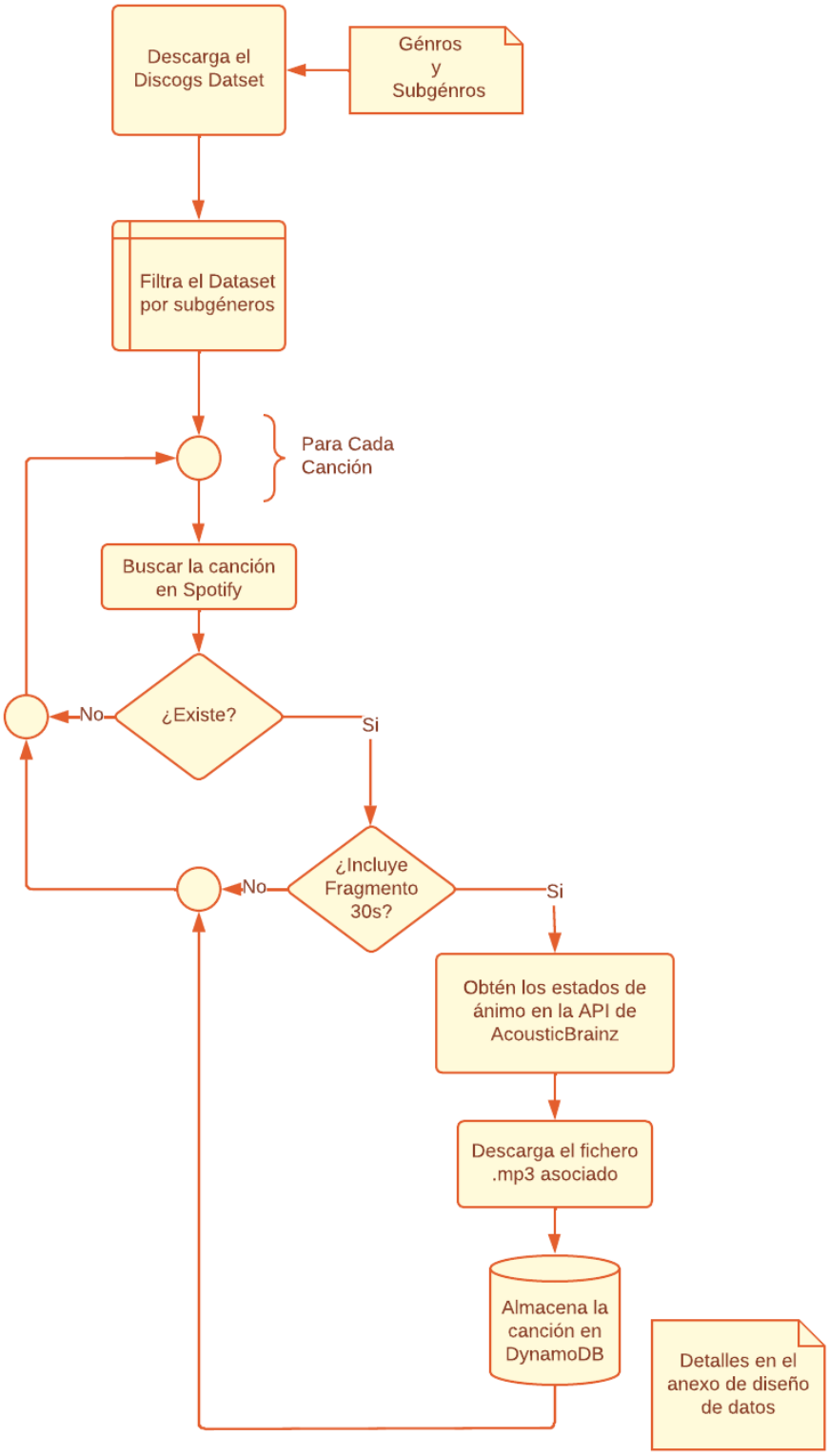


Figura C.2: Proceso de obtención de los datos de Ludwig Dataset

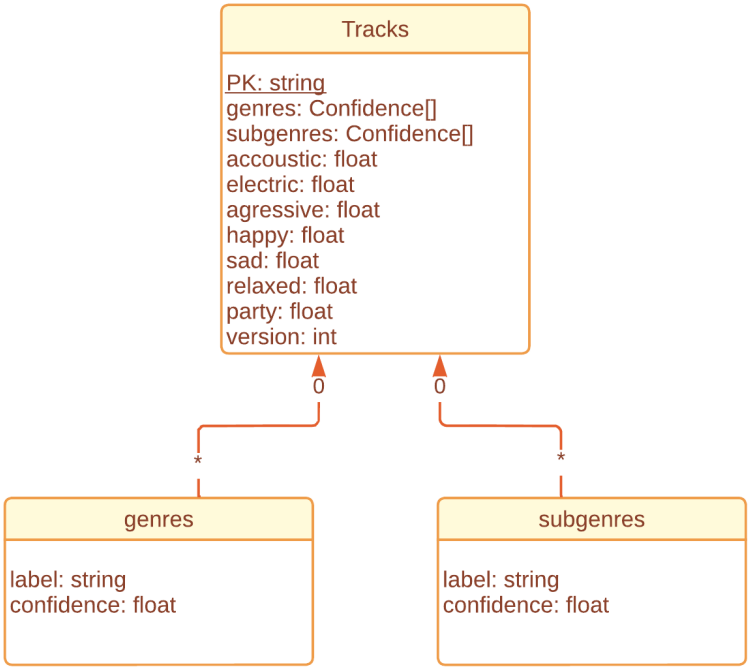


Figura C.3: DynamoDB: Resultados del analizador de canciones

DynamoDB de 40kB por cada atributo, debemos almacenar las etiquetas asociadas a cada álbum mediante atributos dinámicos. Estos campos no se pueden declarar en Dynamoose, por lo que si queremos identificar que un campo es de un tipo específico, es necesario asignarle un prefijo, como por el ejemplo $ALB :< id\_album >$, donde $< id\_album >$ hace referencia al ID de Spotify del álbum.

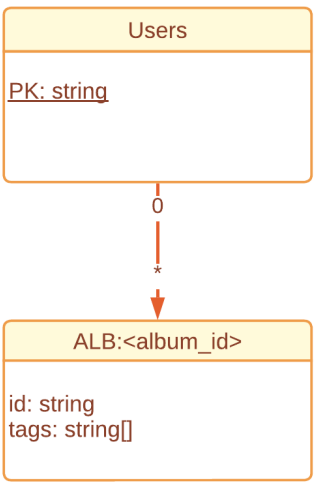


Figura C.4: DynamoDB: Tabla usuarios

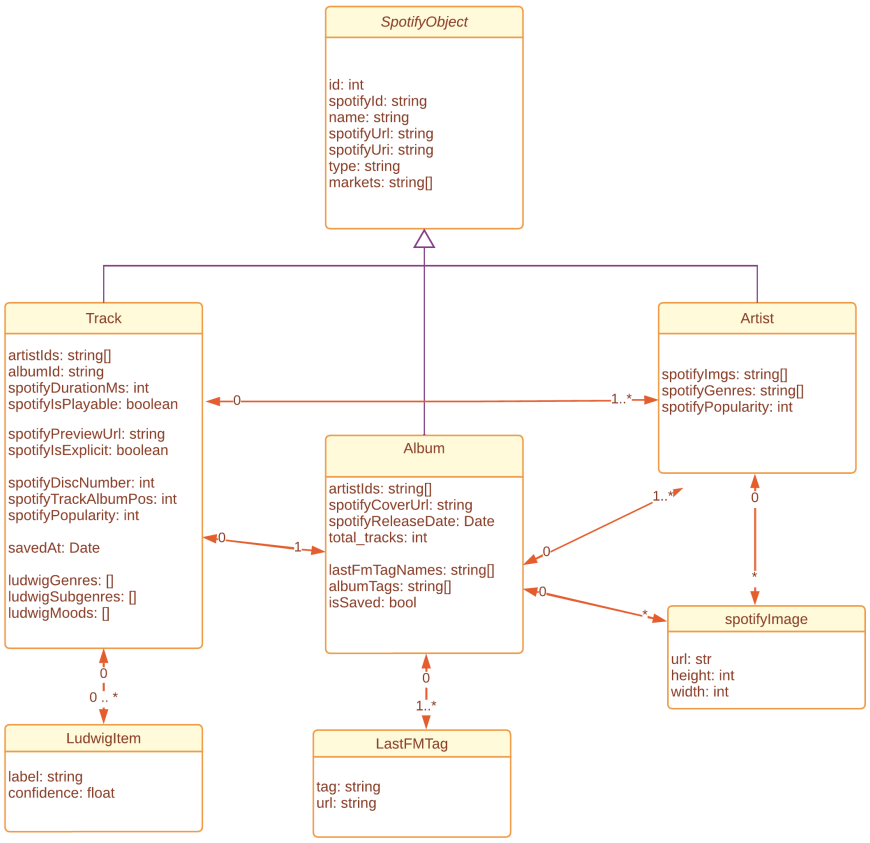


Figura C.5: DexieDB: Canciones, álbumes y artistas

Datos del Frontend

Una de las fuentes de datos de la capa **modelo** C.4 es DexieDB C.5, una base de datos NOSQL que utilizamos como caché intermedia para almacenar los datos obtenidos de las distintas fuentes de datos.

Se ha diseñado el siguiente esquema relacional C.5 mediante interfaces de Typescript. En este caso una canción está compuesta por un único álbum y uno o más artistas. Cada álbum está compuesto por uno o más artistas.

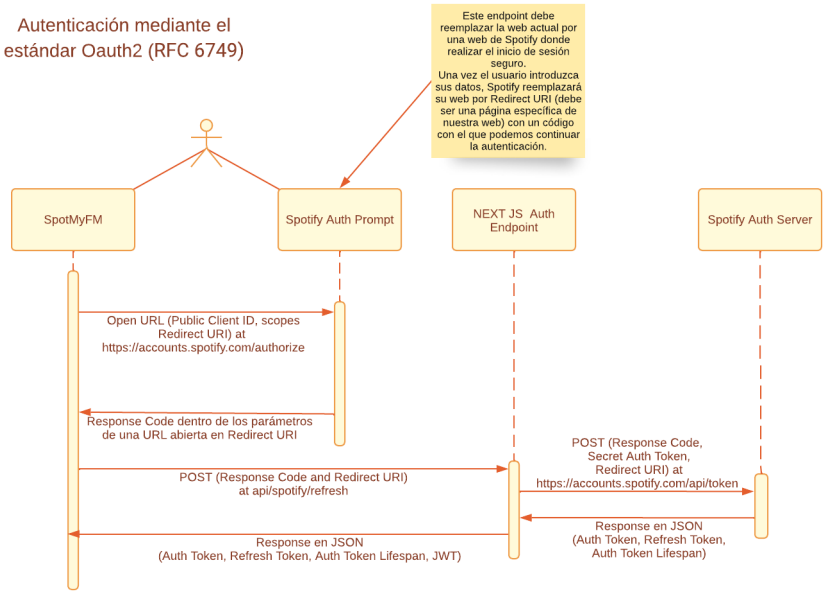


Figura C.6: Flujo de Autenticación OAUTH2

C.3. Diseño Procedimental

Este apartado contiene el diseño detrás del inicio de sesión y la lógica de la fachada de datos C.4.

Inicio de Sesión mediante Oauth2

El inicio de sesión de Spotify utiliza el flujo de autenticación de Oauth2 [4]. En este flujo de identificación necesitamos dos claves, una pública y una privada. La clave pública es accesible desde el frontend y sirve para identificar a la aplicación. La clave privada, gestionada desde el servidor web, permite verificar que un usuario desea utilizar nuestra API, y no se trata de un atacante intentando suplantar la identidad de la aplicación mediante la clave pública.

En este flujo de autenticación C.6, el usuario abre una URL especial con el token público en los parámetros de la URL. Este token permite a Spotify identificar la aplicación, mostrando al usuario el nombre de la aplicación y los permisos que tiene que otorgar a la aplicación.[1]

[1]Estos permisos son conocidos en el estándar OAUTH2 como Scopes.

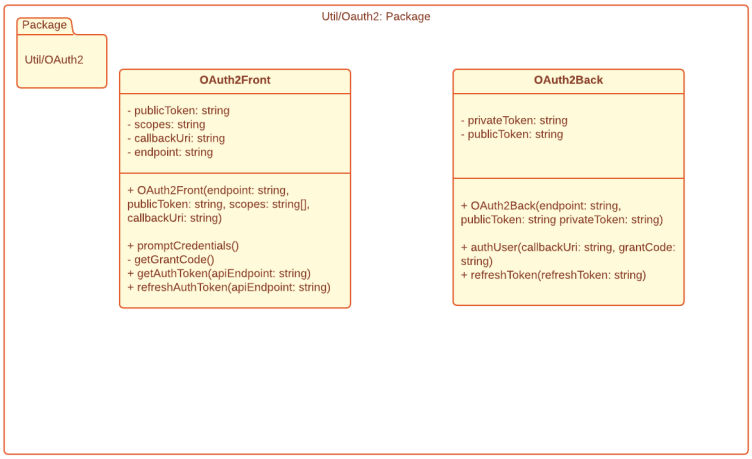


Figura C.7: Clases auxiliares para el flujo Oauth2

Si el usuario acepta los permisos, Spotify abrirá una nueva pestaña con un segundo token de verificación en los parámetros de la URL. Este nuevo token se enviará al servidor web, donde se juntará con el token secreto para generar dos nuevos tokenes: un **token de autenticación** y un **token de refresco**. El token de autenticación permite al usuario interactuar con la API durante 1h, y el token de refresco permite generar nuevos tokenes de autenticación de 1h para que el usuario no tenga que iniciar sesión cada vez que pase 1h de uso.

En este paso aprovechamos a generar un token JWT con una vida útil de 1h con el siguiente contenido:

- Token de Autenticación.
- Nombre de usuario.
- Identificador interno de Spotify.
- Booleano que indica si el usuario está suscrito a Spotify Premium.

Hecho estos devolvemos los 3 Tokens al frontend, donde se almacenarán en varias cookies para que sean accesibles entre sesiones.

Para facilitar la gestión del flujo OAUTH2, se han diseñado dos clases C.7 compatibles con el procedimiento que realizan esta tarea.

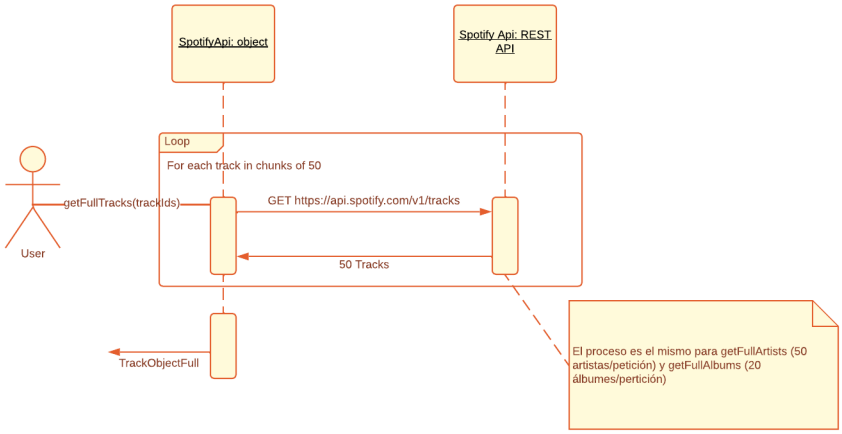


Figura C.8: Obtención de datos de Spotify

Obtención de los Datos

Uno de los objetivos de la fachada de datos C.4 es unificar las distintas fuentes de datos para generar los objetos Fig.C.2 con los que va a trabajar el presentador. Para ello se han planteado los siguientes diagramas de secuencia, que explican como obtener los datos de forma eficiente de cada una de las fuentes de datos.

La figura C.8 hace referencia a como se obtienen todas las canciones completas a partir de una lista de IDs realizando el menor número de peticiones posible.

La figura C.9 explica como se analizan las canciones de Spotify teniendo en cuenta que cada canción analizada se persiste en una base de datos para reducir el tiempo de análisis.

La figura C.10, al igual que las figuras C.11 y C.12 hace referencia a como obtener datos homogéneos a partir de la API de Spotify, LastFM y el Backend Nextjs[2].

La obtención de canciones homogéneas se puede resumir en los siguientes pasos:

1. Carga los elementos de la caché a partir de su ID y calcula los elementos que no están cacheados.

[2]El backend NextJS a su vez se comunica con el Backend de recuperación de información musical y la base de datos

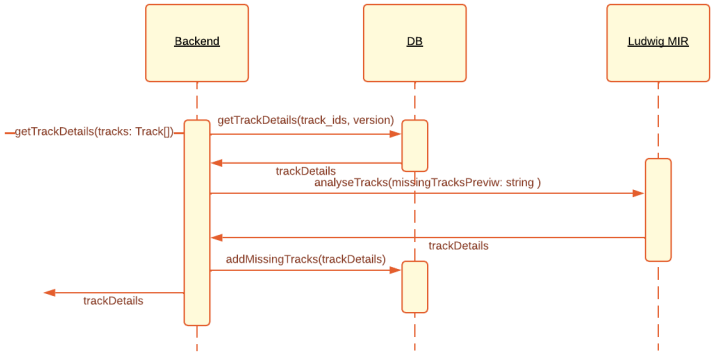


Figura C.9: Obtención de datos de Ludwig MIR

2. Pide a Spotify mediante el proceso de la figura C.8 las canciones completas.

3. Pide al servidor web Ludwig que analice todas las canciones. Este paso se inicia en este punto porque es paso más largo.

4. Realiza el proceso de la fachada con los artistas Fig.C.12 y álbumes Fig.C.11 con todos los álbumes y artistas que ha devuelto la API de Spotify.

5. Una vez acabados los pasos anteriores, realiza una operación de *join* que une cada canción con su álbum y artista asociado, esta operación además persiste los datos en la caché local.

6. Devuelve las canciones homogéneas. Las canciones será actualizadas por referencia una vez se obtengan los resultados del análisis.

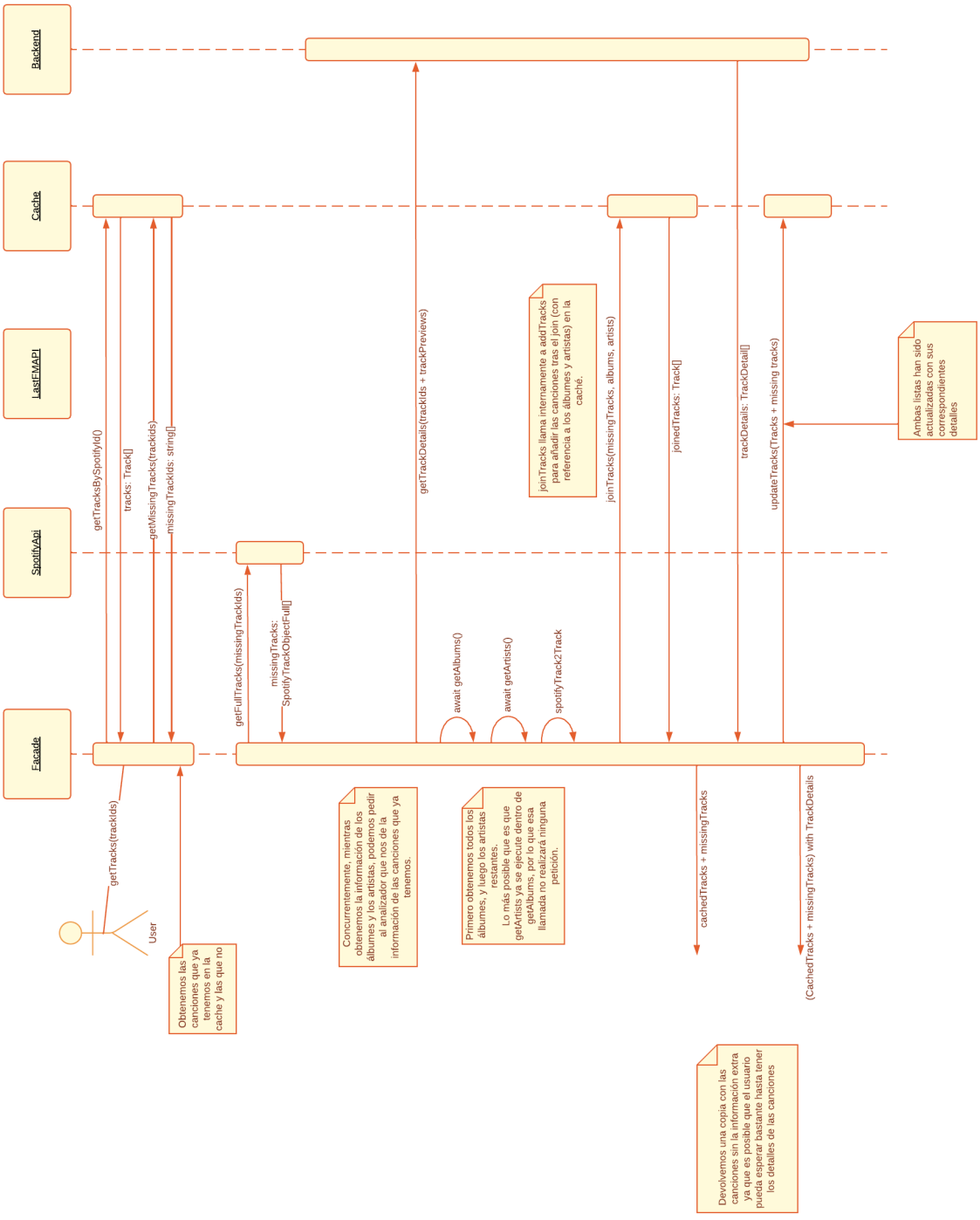


Figura C.10: Obtención de canciones de Spotify

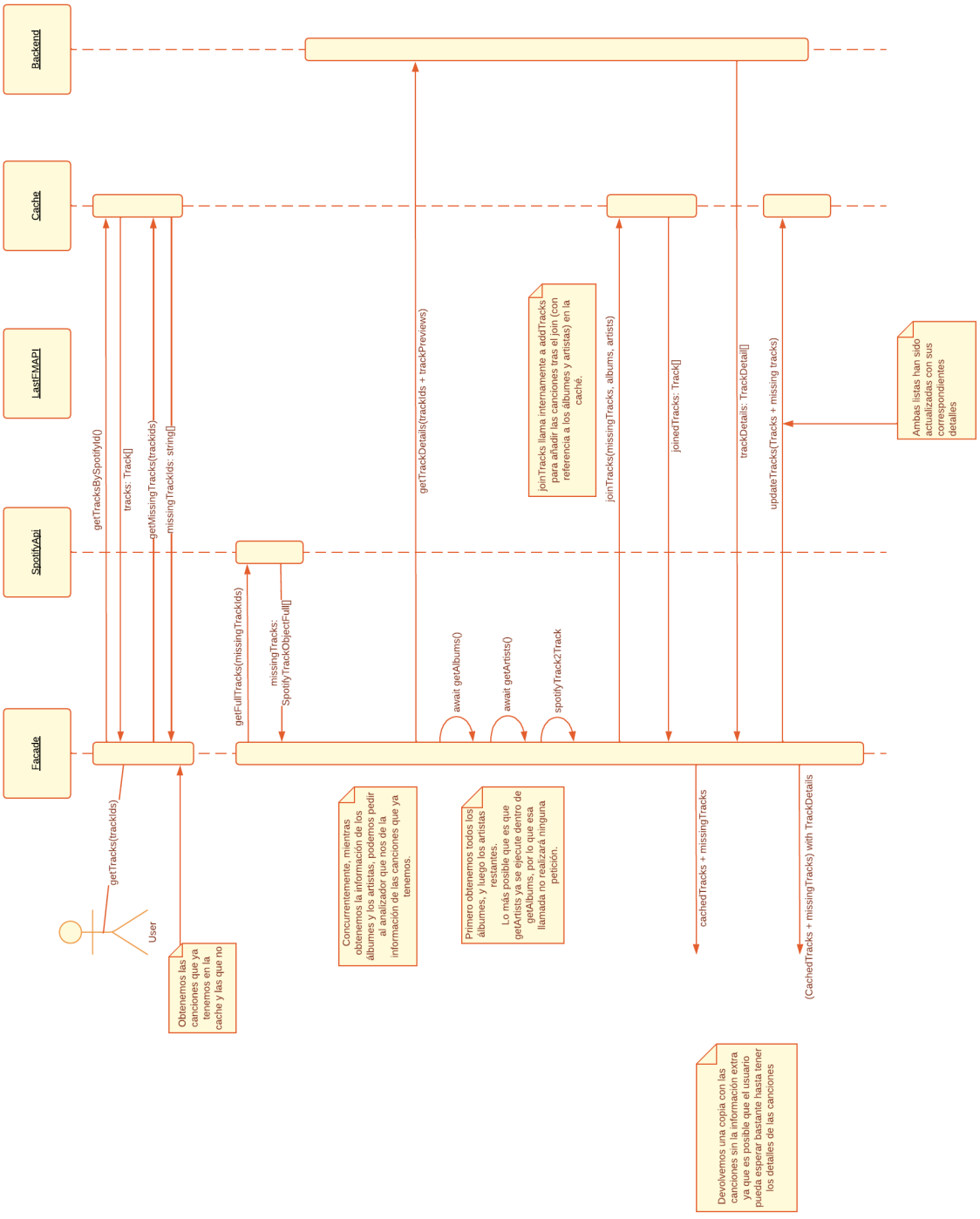


Figura C.11: Obtención de álbumes de Spotify

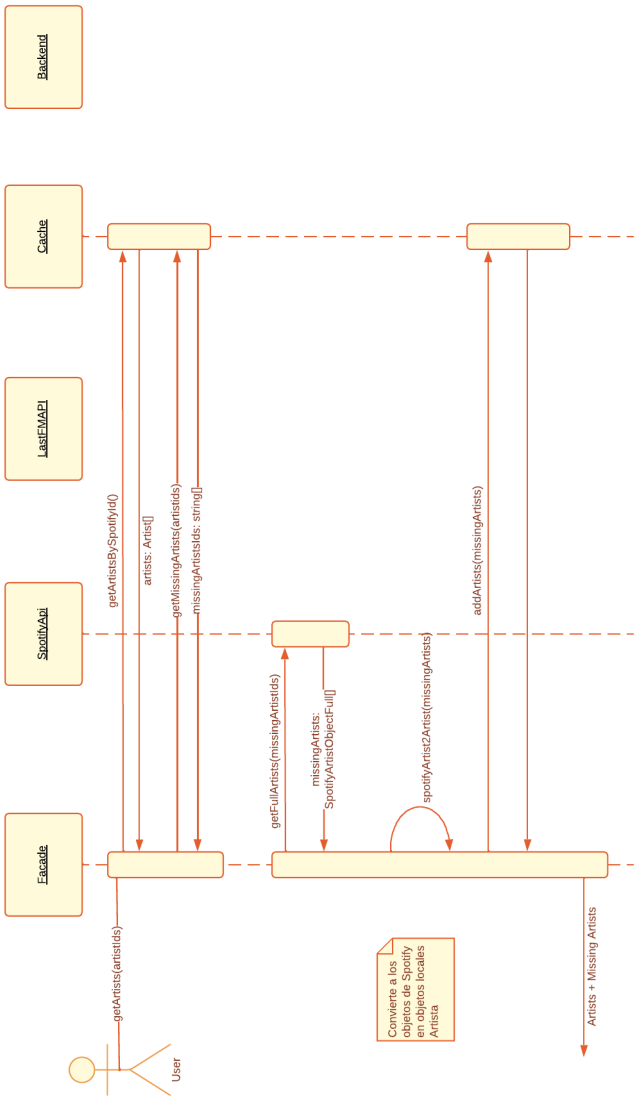


Figura C.12: Obtención de artistas de Spotify

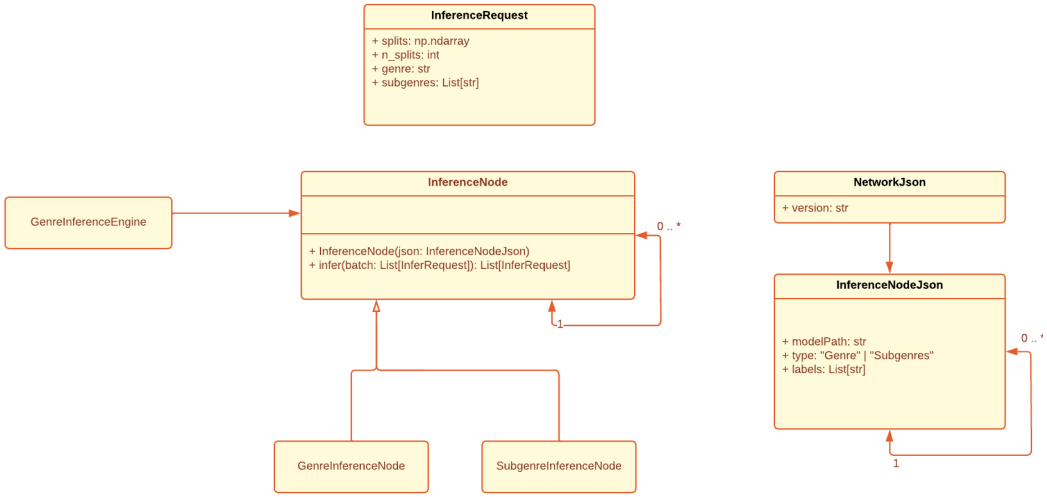


Figura C.13: Motor de Inferencia

C.4. Diseño Arquitectónico

Este apartado concreta los elementos más relevantes sobre la arquitectura del proyecto.

Motor de Inferencia

Para gestionar el elevado número de clasificadores de subgéneros, se ha diseñado el motor de inferencia recursivo de la figura C.13 aplicando el patrón de diseño Composite [3]. Este motor se configura mediante un fichero .json, donde se recogen los géneros y subgéneros de cada uno de los clasificadores, así como la ruta del modelo en formato ONNX.

El motor de inferencia agrupa los distintos MFCCs de 3s de duración en un único lote, minimizando el tiempo de inferencia. Cada lote agrupa las $InferenceRequest$[3] por su género o subgénero. Por ejemplo, el clasificador de géneros agrupará las peticiones de inferencia en 9 grupos, siendo cada grupo un género musical. El nodo pasará cada uno de los lotes a su correspondiente Nodo Hijo, que etiquetará los subgéneros de cada canción actualizando el campo de la petición por referencia.

[3]Peticiones de Inferencia

Patrón MVP

El patrón Modelo Vista Presentador [10] es un patrón de software utilizado para construir interfaces.
En este patrón nos encontramos tres componentes:

1. **Modelo**: Formado por los datos que va a utilizar el Presentador.
2. **Vista**: La interfaz pasiva que muestra los datos procesados por el Presentador.
3. **Presentador**: Es el elemento que interactúa entre el Modelo y la Vista, permite obtener los datos de la vista (entrada de usuario) y del modelo, procesarlos y modificar la vista en función a los cambios.

En este proyecto, el Modelo es la caché de datos y los distintos clientes REST, la vista es el Javascript y HTML generado por ReactJs y el presentador son los Componentes y Hooks de ReactJS escritos en TSX y TS respectivamente.

Fachada de Datos

Para gestionar el flujo de datos especificado en los diagramas de secuencia del aparatado C.3, se ha detallado un diagrama de clases en la figura C.14 con todos los métodos necesarios que detallan las distintas interfaces de los clientes REST, caché local, etc.

En este caso, todos los clientes REST implementan una interfaz común para tratar los errores HTTP, independientemente del cliente HTTP utilizado.

Modelo Serverless

El modelo serverless es un tipo de arquitectura de Cloud Computing que permiten ejecutar y escalar un servicio web sin necesidad de gestionar las máquinas físicas o servidores. En el caso de este proyecto, se han usado dos paradigmas serverless: Functions as a Service (FaaS) y Containers as a Service (CaaS).

Functions as a Service es un paradigma que permite ejecutar funciones sueltas como si fuesen un endpoint de una API. En este caso, por cada petición se levanta un pequeño servicio en los servidores del proveedor cloud, que gestionará la petición HTTP. Una vez se termine la petición, se elimina la

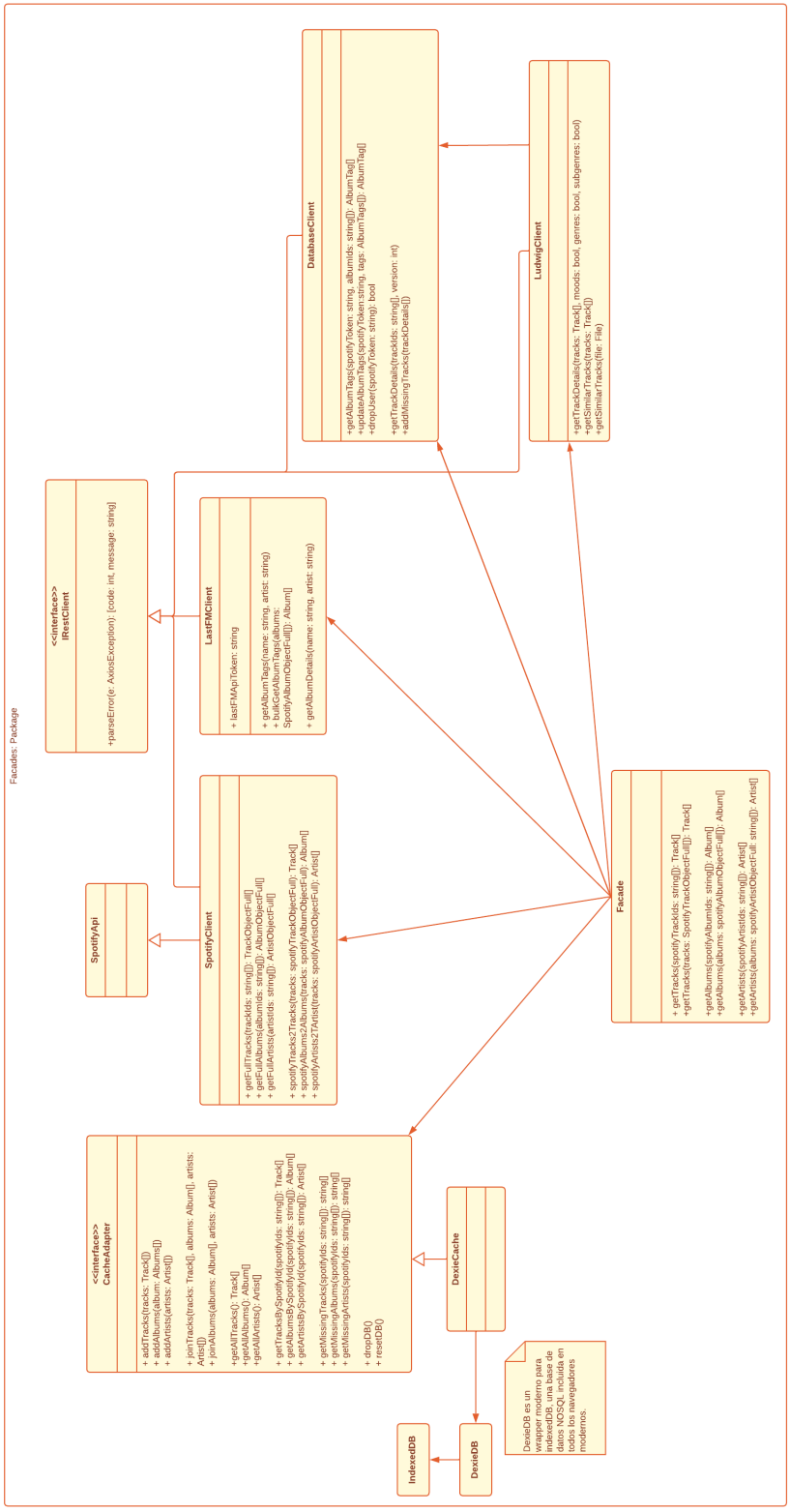


Figura C.14: Fachada de Datos

instancia de la función. Podemos disfrutar de este paradigma si desplegamos la platforma con un proveedor cloud compatible, como Netlify o Vercel.

Containers as a Service es un paradigma que permite instanciar APIs almacenadas en un contenedor de manera similar a FaaS. En este caso, el proveedor cloud permite tener una pool de contenedores, y esta se gestiona automáticamente dependiendo de las necesidades del servicio. Los dos servidores se han desplegado siguiendo este paradigma.

Ambos paradigmas permiten escalar de una forma muy directa los distintos servicios, ya que se pueden desplegar tantos servicios como peticiones y en los servidores más cercanos al usuario. El principal inconveniente de estos paradigmas es el tiempo de arranque, ya que si la función o el contenedor tiene un gran número de dependencias, el arranque puede durar varios segundos para gestionar una petición HTTP pequeña que apenas requiere unos pocos cientos de milisegundos de tiempo de ejecución.

Guía de Estilo

Este apartado contiene una guía de estilo con los colores y tamaños de letra utilizados en el frontend. La guía ha sido generada mediante la herramienta Catalog [2]

Guía de Estilo

Esta guía de estilo ha sido creada gracias a la herramienta de código abierto Catalog.

Temas:

MySpotFM está diseñada alrededor de dos temas, un tema claro (Light) y un tema oscuro (Dark).

Colores:

Paleta de Colores:

Se han escogido diez colores para la paleta de colores principal:

Emerald 50
#ECFDF5

Emerald 100
#D1FAE5

Emerald 200
#A7F3D0

Emerald 300
#6EE7B7

Emerald 400
#34D399

Emerald 500
#10B981

Emerald 600
#059669

Emerald 700
#047857

Emerald 800
#065F46

Emerald 900
#064E3B

Colores Principales:

Se han escogido dos colores principales que van a ser utilizados a lo largo de toda la web para crear elementos como Botones o

Tarjetas.

Ambos colores tienen una variante “Hover” que será utilizada cuando el usuario intente interactuar con un elemento.

Light Green (Emerald 500)
#10B981

Dark Green (Emerald 700)
#047857

Green Hover (Emerald 600)
#059669

Colores de Fondo:

Para el fondo de la Web se han utilizado dos colores muy diferentes, cada uno asociado con uno de los temas principales.

Ambos colores han sido inspirados en la guía de estilos Material Design.

Light Background
#e5e7eb

Dark Background
#212121

Colores de Tarjetas

Para representar la información se utilizan tarjetas.

Estas tarjetas permiten resaltar una sección respecto al fondo.

Light Card
#F3F4F6

Dark Card
#4d4d4d

Para resaltar las tarjetas y dar una sensación de profundidad se utiliza una sombra alrededor de ellas.

<>

Tipografía

Para la tipografía hemos usado la fuente ROBOTO.

El color en el modo oscuro es el blanco puro `(#ffffff)`, mientras que en el modo claro usamos un tono algo más grisáceo `(#4B5563)`.

Nota, el `<h7/>` es lo mismo que `<p/>`

h1 (60px)

The quick brown fox jumps over …

h2 (48px)

The quick brown fox jumps over the lazy …

h3 (36px)

The quick brown fox jumps over the lazy dog

h4 (30px)

The quick brown fox jumps over the lazy dog

h5 (24px)

The quick brown fox jumps over the lazy dog

h6 (20px)

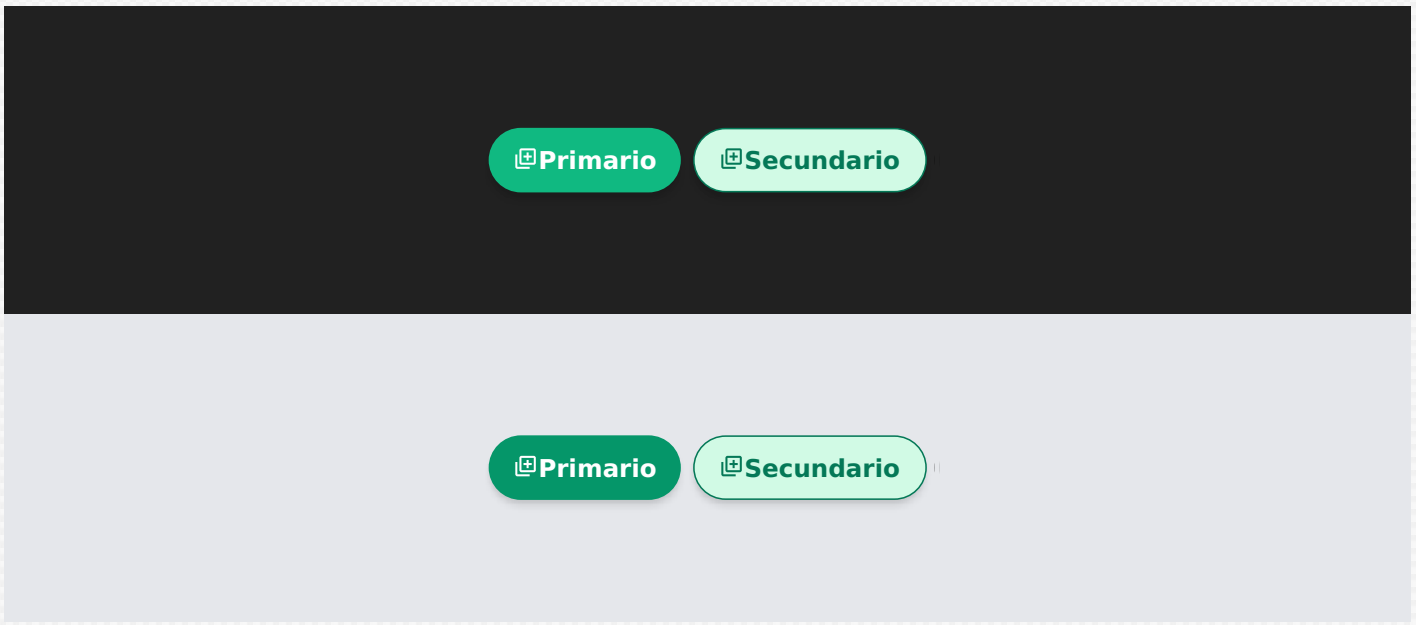
The quick brown fox jumps over the lazy dog

h7 (16px)

The quick brown fox jumps over the lazy dog

color: #4B5563;

Ejemplo de Botones:



C.5. Diseño de Redes Neuronales

Este apartado detalla la arquitectura de las redes neuronales convolucionales que se han utilizado a lo largo del proyecto. Todas la redes tienen como entrada un tensor con dimensión $N \times 130 \times 32$, correspondiente a un bloque de N MFCCs de 3 segundos de duración.

CNN: Clasificador de Géneros

El clasificador de géneros está formado por 2 redes neuronales. Por un lado se ha utilizado la arquitectura EfficientNetB0 [9] como base del clasificador. Esta arquitectura requiere que la entrada tenga 3 canales, ya que está pensada para el tratamiento de imágenes RGB, por lo que es necesario añadir 2 canales a nuestros MFCCs. Para ello se ha usado una capa convolucional como entrada de la red, que permite expandir la entrada con otras 2 copias. A la salida de la red EfficientNetB0, se ha añadido una capa GlobalAveragePooling (GAP), que normaliza la salida de la red al realizar una operación de pooling en cada dimensión, reduciendo la dimensión de la salida de forma similar a una capa Flatten. Por último se añaden dos capas densas, la capa de salida con 9 neuronas equivalentes al número de clases, y una capa intermedia entre GAP y la capa de salida, con activación Softmax. Esta red se corresponde con la figura C.15, donde dense_6 es la salida de la red.

CNN: Clasificador de Subgéneros

El clasificador de subgéneros es una red neuronal convolucional convencional C.16, formada por bloques convolucionales formados por:

- Operación de convolución.
- Operación de normalización.
- Operación de Pooling.
- Segunda operación de normalización.

En este caso, la salida de la red es una capa densa con N neuronas, siendo N el número de subgéneros que es capaz de detectar dicha capa. Esta capa tiene como función de activación la función Sigmoide.

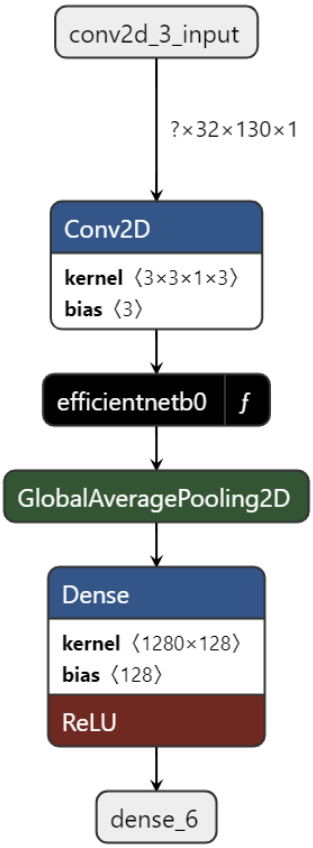


Figura C.15: Arquitectura del Clasificador de Géneros

Ova de CNN: Clasificador de Estados de Ánimo

El clasificador de estados de ánimo C.17 es una OVA de 7 CNNs binarias activadas mediante una Sigmoide. Cada una de las CNNs tiene una arquitectura muy similar a C.5.
La salida de la red es una capa de concatenación que junta todas las salidas en un único tensor.

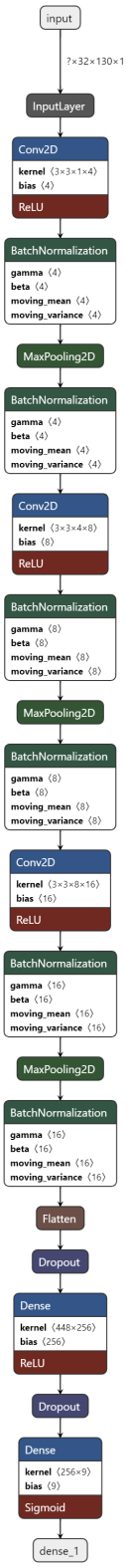


Figura C.16: Arquitectura del Clasificador de Subgéneros

Figura C.17: Arquitectura del Clasificador de Estados de Ánimo

Apéndice D

Documentación técnica de programación

D.1. Introducción

El proyecto está dividido en dos servicios principales, **Nextjs** y **Ludwig/mir-backend**, siendo estas rutas en el USB o Repositorio.

D.2. Estructura de directorios

NextJS

Todo el código se encuentra en subdirectorio **/src**, el resto de ficheros de la raíz son ficheros de configuración de cada uno de los componentes y no deberían ser modificados a menos que se realicen cambios en la estructura general del proyecto.

1. **public/**: Contiene todos los ficheros estáticos que son accesibles desde el navegador.

2. **cypress/**: Contiene los distintos ficheros correspondiente con Cypress, la herramienta de testing E2E.

3. **___mocks___/**: Contiene la configuración de mocks de algunos elementos que no pueden ser ejecutados mediante JEST por requerir de un navegador.

4. **node_modules/**: Contiene todos los módulos de NodeJS, esta carpeta debería modificarse mediante el gestor de paquetes npm, al igual que los ficheros pacage.json y package-lock.json.

Componentes del Proyecto NextJS src/

1. **api/**: Contiene la especificación de la API en formato OpenAPI. La implementación de la API se encuentra bajo pages/api.
2. **backendLogic/**: Contiene ficheros auxiliares que deben ser utilizados desde el Backend, como JWT.
3. **Componentes/**: Contiene todos los componentes de ReactJS que han sido utilizados, se recomienda consultar el apartado D.2
4. **data/**: Contiene la especificación y clases con las transacciones de la capa de datos del cliente (DexieDB) y del servidor (Dynamoose).
5. **enums/**: Contiene varios enums públicos, como por ejemplo el del tema Claro/Oscuro.
6. **hooks/**: Contiene Todos los hooks de ReactJS que han sido definidos.
7. **i18n/**: Contiene la configuración y ficheros de internacionalización.
8. **interfaces/**: Contiene la definición de algunas de las interfaces públicas. Otras de las interfaces se exportan desde el mismo fichero en el que son definidas y utilizadas.
9. **pages/**: Contiene la API pública y todas las rutas de la página. Cada fichero .tsx de este directorio se corresponde con una ruta de la web.
10. **restClients/**: Contiene los distintos clientes rest que se han definido, como el cliente para interactuar con Spotify, LastFM, o el backend de NextJS.
11. **store/**: Contiene los distintos contenedores de datos definidos por Zustand.
12. **syltes/**: Contiene los ficheros CSS y estilos de Styled Components.
13. **Typings/**: Contiene ficheros que declaran los tipos de datos en Typescript de algunas bibliotecas que no tienen definición de tipos. En este caso no hay ninguna biblioteca que no tenga definición de tipos.

14. **util/**: Contiene las definiciones de clases auxiliares e instancias de bibliotecas. Algunas de estas clases son el cliente Oauth2, par iniciar sesión, gestor de cookies, filtros y herramientas para ordenar arrays de canciones, álbumes, etc...

Estructura de src/components

La estructura de este directorio ha seguido dos patrones. Por un lado se han agrupado los componentes según sus características.

Por ejemplo, pages/ contiene todos los componentes que se utilizan para construir una página o ruta al completo, mientras que core/ contiene todos los componentes que son utilizados entre varios componentes, como botones, menús desplegables, etc.

Dentro de core/ nos encontramos a los componentes repartidos en varias categorías.

1. **cards/**: Son las tarjetas, listas y vistas de la aplicación.
2. **display/**: Son los componentes que muestra información, por ejemplo el Modal o el menú de paginación.
3. **input/**: Son los componentes que reciben una entrada del usuario, por ejemplo los sliders.
4. **navigation/**: Son los componentes utilizados para la navegación, como la barra superior y la barra inferior de la página web.
5. **notification/**: Son los componentes utilizados para notificar al usuario.

Por otro lado, estos grupos se dividen según la relevancia del componente siguiente una **estructura atómica.** En este caso, los componentes se agrupan en átomos, moléculas y organismos según su tamaño.
A medida que un componente utiliza componentes de categorías inferiores para construirse, aumenta la categoría del componente. Si tenemos un componente que utiliza una barra de búsqueda y un botón, pasará de ser un componente atómico a ser un componente molecular.

Estructura de un Componente

Cada componente es un directorio que incluye cuatro ficheros.

1. **index.ts**: Exporta por defecto al componente. Esto permite importar un componente desde cualquier módulo sin necesidad de conocer en que punto del directorio se encuentra.

2. ***.styles.ts**: Contiene la definición de estilos utilizados por ese componente mediante Styled Componentes. Este fichero exporta un único diccionario por defecto.

3. ***.test.ts**: Contiene los tests unitarios del componente para se ejecutados mediante Jest.

4. ***.tsx**: Incluye el componente a renderizar.

Ludwig

Este subdirectorio contiene los datos del proyecto de extracción de características musicales Ludwig.

Está formado por los siguientes directorios:

1. **dataset-tools/**: Contiene las herramientas desarrolladas en NodeJS y Python para la generación de los distintos datasets. Se ha detallado en la sección D.2.

2. **mir-backend/**: Contiene la API de Ludwig en FastAPI. Se ha detallado su uso en D.2

3. **models/**: Contiene el modelo base para inicializar los pesos del clasificador de música. Esta en esta carpeta para poder acceder a él mediante curl/wget desde los notebooks utilizados durante el entrenamiento.

4. **notebooks/**: Contiene los distintos notebooks utilizados para entrenar con los modelos y explorar los conjuntos de datos.

dataset-tools

Por un lado, la herramienta en NodeJS contiene una aplicación de línea de comandos que permite:

1. Descargar una playlist pública / usuario en formato .mp3

2. Normalizar una playlist. Esta normalización elimina aquellas canciones que no contienen 30s de previsualización.

3. Leer Discogs. Lee el dataset de Discogs en formato .csv, lo cruza con Spotify y AcousticBrainZ y lo guarda en DynamoDB. Para más detalles consultar la memoria.

Por otro lado, las herramientas de Python permiten descargar el dataset almacenado en DynamoDB y extraer las características mediante Librosa.

1. **downloadDataset.py**: Descarga el dataset de dynamoDB y genera un fichero .json con todas las canciones.

2. **2mfccs.py**: Dado un directorio con ficheros .wav, divide la canción en segmentos de 3s y genera los MFFCs en cad segmente. Almacena en un fichero .npy con el mismo nombre del fichero .wav un array con los MFCCs.

3. **2sftf.py**: Realiza la misma operación que el paso anterior, pero convierte los segmentos en espectrogramas en escala de mel.

4. **dynamo2labels**: A partir del json obtenido en 2, genera un json que agrupa las distintas canciones por subgéneros.

mir-backend

Este directorio incluye la implementación del backend de recuperación de información musical.

1. **inference_engine/**: Contiene la implementación del motor de inferencia para géneros, subgéneros y estado de ánimo.

2. **models/**: Contiene los modelos finales. Este directorio contiene los binarios de Keras orignales (.h5), así como sus versiones compatibles con ONNX (.onnx y _quantized.onnx)

3. **request_specifications/**: Contiene las especificaciones de los cuerpos de las peticiones HTTP.

4. **util/**: Contiene los módulos para descargar y extraer las características de las canciones. En este caso se dividen las canciones, se convierten a .wav y se remuestrean a 22050Hz. Estas canciones son cargadas por Librosa, divididas en fragmentos de 3s y transformadas en MFCCs.

D.3. Manual del programador

En esta sección se van a detallar los aspectos más importantes del proyecto de cara al programador.

Variables de Entorno

Las variables de entorno son muy utilizadas para configurar valores secretos sin necesidad de incluirlas en el código.

Variables de entorno de NextJS

NextJS permite cargar variables de entorno a partir de un fichero **.env.local** que se encuentra en la raíz del proyecto.

Se diferencian dos tipos de variables de entorno, las variables del frontend y las variables del backend. Las variables del frontend se identifican por empezar por **NEXT_PUBLIC_**, y son accesibles desde el frontend ya que se cargan en tiempo de compilación. Las variables del backend no tienen ningún prefijo y se cargan en tiempo de ejecución, al arrancar el servidor. Si por ejemplo modificamos una variable de entorno en una del frontend, es necesario volver a construir la imagen de Docker.

Para facilitar el acceso a estas variables, se ha declarado un diccionario en el archivo **env.ts**, localizado en el subdirectorio **src/**.

1. **NEXT_PUBLIC_SPOTIFY_ID**: Almacena el código público que debe conocer el frontend para poder iniciar sesión el usuario, se genera desde https://developer.spotify.com/.
2. **SPOTIFY_SECRET**: Almacena el código privado que se utiliza para validar el inicio de sesión desde el backend, se genera junto al ID Público de Spotify.
3. **JWT_SIGN_KEY**: Almacena la clave de firma utilizada para emitir Json Web Tokens.
4. **AWS_ACCESS_KEY_ID_**: Almacena el ID de Amazon Web Services utilizado para conectar Dynamoose con DynamoDB. Se puede obtener desde IAM en AWS.
5. **AWS_SECRET_ACCESS_KEY_** : Almacena la clave de inicio de AWS, se usa en conjunto con el elemento anterior.

6. **AWS_REGION_**: Identifica la región en la que se despliega la base de datos. No todas las regiones están disponibles para Dynamoose, se recomienda consultar la documentación.

7. **DYNAMOOSE_USER_TABLE**: Nombre de la tabla en la que se van a almacenar los datos de usuario, por defecto la tabla se llama TEST_TABLE.

8. **DYNAMOOSE_TRACK_TABLE**: Nombre de la tabla en la que se van a almacenar los datos de ludwig, por defecto la tabla se llama TEST_TABLE.

9. **DYNAMOOSE_LOCAL**: Por defecto está vacía. Puede contener la dirección de una instancia de DynamoDB Local.

10. **NEXT_PUBLIC_LAST_KEY**: Almacena la clave pública de LastFM.

11. **NEXT_PUBLIC_API_BASE_URL**: Almacena la URL al servidor que almacena la API de SpotMyFM. Por defecto usa /, es decir, el mismo servidor que el frontend.

12. **NEXT_PUBLIC_LUDWIG_URL**: Almacena la url al servidor de la api Ludwig.

13. **LUDWIG_SECRET**: Almacena el código secreto que permite al backend de NextJS conectarse con Ludwig.

Variables de entorno de Ludwig Mir Backend

Este backend es mucho más sencillo, las variables de entorno pueden almacenarse en un fichero **.env** en la raíz del proyecto.

Este fichero únicamente contiene una variable de entorno, **SECURITY_TOKEN**, que almacena el mismo valor que 13.

D.4. Compilación, instalación y ejecución del proyecto

Instalación de Docker

GNU/Linux

La instalación para GNU/Linux es independiente para cada distribución, por lo que es recomendable consultar la documentación para conocer los distintos pasos. En nuestro caso vamos a utilizar Ubuntu, por lo que vamos a seguir los pasos con esta distribución.

Ubuntu Vamos a utilizar el script oficial de instalación:

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

Hecho esto, tendremos Docker instlado en nuestro equipo, pero es recomendable añadir a los usuarios que vayan a usar este servicio a el grupo **docker** para poder usarlo sin necesidad de permisos de superusuario.

Para añadir al usuario actual:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
```

Hecho esto es recomendable reiniciar el equipo para que el proceso se complete de forma satisfactoria. Podemos probar que todo está correctamente instalado usando el siguiente comando:

```
$ docker run hello-world
```

Si utilizamos el comando `$ docker ps` podemos comprobar si el contenedor está en ejecución.

Microsoft Windows

La instalación de Docker en Windows es algo más complicada ya que requiere de un servicio de virtualización.

Instalación WSL El primer requisito que necesitamos es instalar el Subsistema de Linux para Windows (Windows Subsystem for Linux) Es recomendable seguir la guía de instalación actualizada para este proceso, ya que ha cambiado mucho a lo largo de los últimos años.

Suponiendo que tengamos una versión relativamente actualizada de Window 10 o Windows 11, deberíamos poder ejecutar el siguiente comando con permisos de administrador para la instalación:

```
wsl --install
```

Hecho esto buscamos la distribución que más nos guste en la tienda de aplicaciones de Windows (En las últimas versiones WSL se puede instalar directamente desde esta tienda). En nuestro caso vamos a escoger Ubuntu 20.04 LTS

Se nos habrá instalado como una aplicación más que podemos encontrar en el menú de inicio, en el PATH (`ubuntu2004.exe`), etc. Podemos abrir la distribución que tengamos por defecto con los comandos

```
wsl
```

ó

```
bash
```

Como únicamente tenemos una distribución instalada, podemos abrir Ubuntu directamente con este proceso. Hecho esto abrimos la distribución y la configuramos como cualquier otro sistema operativo.

Actualizar la distribución a WSL2 Para poder utilizar la distribución con Docker, debemos utilizar la versión 2 de WSL que virtualiza el Kernel de Linux. Este paso es tan sencillo como utilizar el siguiente comando:

```
wsl --set-version <Distro> 2
```

Podemos obtener `<Distro>` a partir del siguiente comando:

```
wsl --list
```

En nuestro caso tenemos las distribuciones que va a instalar Docker ya instaladas, pero podemos ver el nombre exacto de nuestra distribución Ubuntu.

Hecho esto ya podemos instalar Docker.

Instalación de DOCKER Podemos seguir la siguiente guía oficial o usar Winget, en nuestro caso vamos a usar el gestor de paquetes.

```
winget install -e --id Docker.DockerDesktop
```

Configuración de DOCKER Si queremos acceder a los comandos de docker desde Ubuntu, debemos indicar a Docker que pueda utilizar Ubuntu como una interfaz más. Para ello podemos utilizar la interfaz gráfica de Docker Desktop.

Construir y Ejecutar los Contenedores

Se pueden levantar los tres contenedores con el fichero **Docker/compose.yml** mediante el comando **docker-compose up**.

Es necesario definir un fichero **.env** con todas las variables de entorno mencionadas en el apartado D.3.

Se pueden construir y ejecutar los contenedores de forma individual con **docker build -t nombre_del_servicio . && docker run nombre_del_servicio**[1]

[1]Es necesario ejecutarlo sobre la raíz de cada proyecto, NextJS/ y Ludwig/ludwig-mir/

Apéndice E

Documentación de usuario

E.1. Introducción

Se recomienda consultar la documentación desde el propio repositorio al contener una versión más actualizada.
https://github.com/JorgeRuizDev/SpotMyFM/tree/main/Manual/es/readme.md

E.2. Requisitos de usuarios

El usuario necesita una cuenta en Spotify (http://spotify.com). Algunas características de SpotMyFM son exclusivas de usuarios suscritos a Spotify Premium por limitaciones de la API.

Navegadores Recomendados

Se recomienda usar la última versión de cada uno de los navegadores. Si bien SpotMyFM puede funcionar en versiones más antiguas, se ha seleccionado la última versión recomendada de cada navegador. Utilizar una versión más antigua puede ser peligroso debido a los múltiples fallos de seguridad que se han ido acumulando.

1. Navegador basado en Chromium 93 (Google Chrome 93)
2. WebKit 12 (Safari 12)
3. Mozilla Firefox 88

E.3. Manual del usuario

El manual de usuario en castellano se encuentra disponible en la siguiente URL: https://github.com/JorgeRuizDev/SpotMyFM/tree/main/Manual/es/readme.md

Bibliografía

[1] Api terms of service | last.fm. https://www.last.fm/api/tos. (Accessed on 07/06/2022).

[2] Catalog. https://www.catalog.style/. (Accessed on 07/04/2022).

[3] Composite. https://refactoring.guru/design-patterns/composite. (Accessed on 07/04/2022).

[4] Rfc 6749 - the oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749. (Accessed on 07/04/2022).

[5] Spotify developer terms | spotify for developers. https://developer.spotify.com/terms/. (Accessed on 07/04/2022).

[6] Spotify developer terms | spotify for developers. https://developer.spotify.com/terms/. (Accessed on 07/06/2022).

[7] • spotify users - subscribers in 2022 | statista. https://www.statista.com/statistics/244995/number-of-paying-spotify-subscribers/#:~:text=As%20of%20the%20first%20quarter,than%20doubled%20since%20early%202017. (Accessed on 07/06/2022).

[8] Discogs: la base de datos y el mercado online de la música, 2022.

[9] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[10] Wikipedia contributors. Model–view–presenter — Wikipedia, the free encyclopedia, 2021. [Online; accessed 3-July-2022].