

Trabajo de Fin de Grado
Grado en Ingeniería Informática
Ingeniería de Software

**Desarrollo de una Plataforma Web Interactiva de
Datos Musicales Obtenidos de la API de Spotify**

Jon Ortega Goikoetxea

Dirección
Miren Bermejo Llopis

13 de febrero de 2025

Agradecimientos

En caso de querer añadir agradecimientos, escribir aquí el texto.

En caso de no querer este apartado, comentalo en el fichero *main.tex*.

Resumen

Escribe aquí el resumen.

Objetivos de Desarrollo Sostenible

Este proyecto se alinea con varios Objetivos de Desarrollo Sostenible (ODS) de las Naciones Unidas. En concreto, se toma como referencia el marco de la *EHUagenda 2030* de la UPV/EHU. Esta agenda, “recoge la contribución de la UPV/EHU a 12 de los 17 ODS de la Agenda 2030, al que ha sumado el su compromiso con la diversidad lingüística y cultural a través del ODS 17+1” [1]. A continuación, se detallan los objetivos específicos relacionados:



Figura 0.1: Los ODS alineados con este trabajo.

- **ODS 3: Salud y Bienestar.** La música juega un papel importante en el bienestar emocional y mental. La capacidad que ofrece esta aplicación de mejorar la experiencia musical y permitir a los usuarios conectar más profundamente con su música contribuye positivamente a su bienestar general.
- **ODS 9: Industria, Innovación e Infraestructura.** Como este proyecto implica la utilización de tecnologías modernas como React, Next.js y Vercel durante el desarrollo, se fomenta la innovación tecnológica y se contribuye al desarrollo de infraestructuras digitales eficientes.
- **ODS 18 (17+1): Garantizar la diversidad lingüística y cultural.** Al permitir que los usuarios exploren y aprecien música de diferentes culturas y en diversos idiomas, se facilita la exposición a estas, fomentando así el entendimiento y la apreciación cultural, alineándose así con el objetivo 18 propuesto por la UPV/EHU.

Índice de contenidos

Índice de contenidos	IV
Índice de figuras	viii
Índice de tablas	xii
Índice de algoritmos	xiii
1 Introducción	1
1.1. Contexto	1
1.2. Motivación	2
1.3. Objetivos del Proyecto	2
1.4. Estructura de la Memoria	3
2 Contexto Competitivo	4
3 Planificación	6
3.1. Alcance	6
3.1.1. Funcionalidades Incluidas	6
3.1.2. Exclusiones	6
3.1.3. Limitaciones	7
3.2. Gestión de Tareas	7
3.2.1. Descripción de Tareas	7
3.2.2. Dedicaciones	11
3.2.3. Dependencias entre Tareas	12
3.2.4. Periodos de Desarrollo e Hitos	13
3.3. Gestión de Riesgos	14
3.4. Herramientas y Tecnologías	16
4 Estudio de la API de Spotify	20
4.1. Descripción General	20
4.2. Autenticación y Autorización	21
4.3. Principales Endpoints Relevantes para el Proyecto	23
4.3.1. Endpoints de Autenticación	23

4.3.2. Endpoints de Datos	25
4.4. Scopes Necesarios	34
4.5. Limitaciones y Consideraciones	34
4.6. Política y Términos de Uso de la API	35
4.6.1. Restricciones y Obligaciones	35
4.6.2. Protección de Datos y Seguridad	36
4.6.3. Implicaciones para el TFG	36
5 Análisis	37
5.1. Requisitos Funcionales	37
5.1.1. Autenticación y Sesión	37
5.1.2. Navegación	37
5.1.3. Estadísticas Generales (Home)	38
5.1.4. Estadísticas Avanzadas (Stats)	38
5.2. Requisitos No Funcionales	40
5.3. Casos de Uso	41
5.3.1. Actores	41
5.3.2. Modelo de Casos de Uso	41
6 Diseño	49
6.1. Arquitectura del Sistema	49
6.1.1. Rutas del Frontend	50
6.1.2. Endpoints del Backend	50
6.2. Diagrama de Componentes de React	51
6.3. Interfaz de Usuario	53
6.3.1. Principios de Diseño	53
6.3.2. Tailwind CSS	54
6.3.3. Páginas y Componentes Principales	55
6.4. Diagramas de Secuencia	58
6.5. Diseño de la Seguridad	61
6.5.1. Gestión de Credenciales	61
6.5.2. Routing y Conexiones Seguras	62
6.5.3. Otras Medidas	62
7 Implementación	63
7.1. Entorno de Desarrollo Local	63
7.1.1. Tecnologías y Versiones	63
7.1.2. Instalación y Configuración del Proyecto	64
7.2. Registro de la Aplicación en Spotify	65
7.3. Ciclo de Desarrollo	66
7.4. Gestión Segura de los Tokens	69
7.5. Implementación del Frontend	70
7.5.1. React Hooks	70
7.5.2. Estadísticas Básicas (Home)	71
7.5.3. Estadísticas Avanzadas (Stats)	72
7.6. Implementación del Backend	77
7.6.1. Implementaciones Destacadas	77
7.6.2. Resto de Endpoints	78

7.7.	Middleware	79
7.7.1.	Funcionamiento del Middleware	79
7.7.2.	Limitaciones y Consideraciones de Seguridad	79
7.8.	Optimizaciones	80
7.8.1.	Peticiones a Spotify y Procesado de Datos	80
7.8.2.	Interfaz y Experiencia de Usuario	80
7.8.3.	Optimización en Estadísticas Específicas	81
7.8.4.	Impacto General de las Optimizaciones	82
8	Pruebas	83
8.1.	Implementación de Pruebas	83
8.1.1.	Framework y Librerías	83
8.1.2.	Consideraciones a Tener en Cuenta	83
8.2.	Resultados de Pruebas	84
9	Despliegue	85
9.1.	Vercel	85
9.2.	Límites y Características	85
9.3.	Proceso de Despliegue	86
9.3.1.	Configuración de Variables de Entorno	87
9.3.2.	Análisis y Monitoreo	87
9.4.	Ejecución Automática de Tests	88
10	Seguimiento y Control	89
10.1.	Metodología de Seguimiento	89
10.2.	Incidencias	89
10.2.1.	Control de Riesgos	89
10.2.2.	Control de Alcance	89
10.2.3.	Control de Planificación	89
10.3.	Revisión de Objetivos	89
11	Conclusiones	90
11.1.	Conclusiones Técnicas	90
11.2.	Conclusiones Personales	90
11.3.	Líneas Futuras	90
Anexos		91
A	Capturas de la Interfaz de Usuario	92
A.1.	Interfaz de Estadísticas	92
A.2.	Componentes Secundarios, de Carga y de Errores	94
B	Diagramas de Secuencia Adicionales	97
C	Contenidos Relacionados con la Implementación	99
D	Información del Backend de las Estadísticas Avanzadas	111
D.1.	Información del Backend del Hall Of Fame	111
D.2.	Información del Backend de Estaciones Musicales	112

ÍNDICE DE CONTENIDOS

vii

D.3. Información del Backend de Huella del Día	114
D.4. Información del Backend de La Bitácora	115
D.5. Información del Backend de Tus Décadas	116
D.6. Información del Backend de Índice de Interferencia	118
Bibliografía	119

Índice de figuras

2.1. Ejemplo de estadística de “oscuridad” de <i>Obscurify</i>	5
2.2. Imagen generada por MusicScapes.	5
3.1. Diagrama EDT con los paquetes de trabajo del proyecto.	8
3.2. Diagrama de dependencias entre las tareas y paquetes de trabajo del proyecto.	12
3.3. Diagrama Gantt con los tiempos de realización de las tareas y paquetes de trabajo.	13
4.1. Pantalla de autorización de scopes en Spotify.	21
4.2. Plantilla visual para representar los endpoints.	23
4.3. Endpoint de <i>Request User Authorization</i>	24
4.4. Endpoint de <i>Request Access Token</i>	25
4.5. Grupos de endpoints ofrecidos y cuáles se van a usar.	26
4.6. Endpoint de <i>Get Current User’s Profile</i>	26
4.7. Endpoint de <i>Get User’s Top Items (Tracks)</i>	27
4.8. Endpoint de <i>Get User’s Top Items (Artists)</i>	28
4.9. Endpoint de <i>Get Recently Played Tracks</i>	29
4.10. Endpoint de <i>Get User’s Saved Tracks</i>	30
4.11. Endpoint de <i>Get Several Artists</i>	31
4.12. Endpoint de <i>Create Playlist</i>	32
4.13. Endpoint de <i>Add Items to Playlist</i>	33
4.14. Endpoint de <i>Add Custom Playlist Cover Image</i>	33
4.15. Gráfica del funcionamiento de la tasa de peticiones, obtenida de la documentación oficial de Spotify.	34
5.1. Modelo de casos de uso del sistema.	42
6.1. Diagrama de la arquitectura del sistema haciendo uso de <i>Next.js</i>	49
6.2. Diagrama de la jerarquía de componentes de React creados en el proyecto.	52
6.3. Colores seleccionados para la paleta de colores de la aplicación.	54
6.4. Muestra de caracteres de la tipografía <i>Inter</i>	54
6.5. Página de inicio de sesión del proyecto.	55
6.6. Barra de navegación fijada en la parte superior de la web.	55
6.7. Página de <i>Home</i> con las estadísticas básicas.	56
6.8. Página de <i>Stats</i> con las tarjetas de estadísticas avanzadas.	56

6.9.	Ventana modal que contendrá la estadística a mostrar (en este ejemplo, <i>Huella Del Día</i>).	57
6.10.	<i>Footer</i> anclado al pie de todas las páginas de la web.	57
6.11.	Diagrama de secuencia: Iniciar Sesión.	58
6.12.	Diagrama de secuencia: Acceder a <i>Home</i>	59
6.13.	Diagrama de secuencia: Acceder a <i>Stats</i>	59
6.14.	Diagrama de secuencia: Ver Hall Of Fame.	60
6.15.	Diagrama de secuencia: Ver La Bitácora.	60
7.1.	Panel de creación de app en la plataforma de <i>Spotify</i>	65
7.2.	Panel de ajustes con el <i>Client ID</i> y el <i>Client Secret</i>	65
7.3.	Panel de gestión de usuarios que tienen acceso a la aplicación.	66
7.4.	Diagrama explicativo de la actualización de las estadísticas básicas del <i>Home</i>	72
7.5.	Diagrama explicativo de la actualización de la página <i>Stats</i> para detectar la selección de la estadística.	73
9.1.	Pantalla de creación del proyecto tras conectarlo con el repositorio de <i>GitHub</i>	86
9.2.	Panel con la visión general de los recursos consumidos.	87
A.1.	Interfaz de la estadística <i>Hall Of Fame</i>	92
A.2.	Interfaz de la estadística <i>La Bitácora</i> (año).	92
A.3.	Interfaz de la estadística <i>La Bitácora</i> (mes).	92
A.4.	Interfaz de la estadística <i>La Bitácora</i> (día).	92
A.5.	Interfaz de la estadística <i>La Bitácora</i> , en los tres diferentes niveles de detalle.	92
A.6.	Interfaz de la estadística <i>Huella Del Día</i>	93
A.7.	Interfaz de la estadística <i>Tus Décadas</i> (zoom out).	93
A.8.	Interfaz de la estadística <i>Tus Décadas</i> (zoom in).	93
A.9.	Interfaz de la estadística <i>Tus Décadas</i> , en diferentes niveles de zoom.	93
A.10.	Interfaz de la estadística <i>Estaciones Musicales</i> (cerrado).	93
A.11.	Interfaz de la estadística <i>Estaciones Musicales</i> (abierto).	93
A.12.	Interfaz de la estadística <i>Estaciones Musicales</i> , en los estados de cerrado y abierto.	93
A.13.	Interfaz de la estadística <i>Índice De Interferencia</i> (originales).	93
A.14.	Interfaz de la estadística <i>Índice De Interferencia</i> (combinados).	93
A.15.	Interfaz de la estadística <i>Índice De Interferencia</i> , con las ondas originales y su forma combinada.	93
A.16.	Detalle de los tres <i>tops</i> junto con el selector de periodo de tiempo.	94
A.17.	Detalle de <i>Recently Played</i> con el botón para ampliar o contraer la lista.	94
A.18.	Panel del usuario con la opción de <i>Log Out</i>	95
A.19.	Componente que se muestra mientras se cargan los datos de las estadísticas, con texto dinámico.	95
A.20.	Componente de error que se muestra cuando el usuario no tiene ninguna canción guardada en su lista de favoritos.	95
A.21.	Componentes de <i>loading</i> para los tres <i>tops</i>	96
B.1.	Diagrama de secuencia: Cerrar Sesión	97
B.2.	Diagrama de secuencia: Ver Huella Del Día	97
B.3.	Diagrama de secuencia: Ver Estaciones Musicales	98
B.4.	Diagrama de secuencia: Ver Tus Décadas	98

ÍNDICE DE FIGURAS

x

B.5. Diagrama de secuencia: Ver Índice De Interferencia.	98
C.1. Panel de monitoreo de la actividad de la aplicación registrada.	100

Índice de tablas

2.1.	Comparativa de funcionalidades ofrecidas por otros servicios afines.	4
3.1.	Tabla con las estimaciones de tiempo por paquete de trabajo y tarea del proyecto.	11
4.1.	Authorization flows definidos por OAuth 2.0 y cuáles implementa Spotify. . . .	22
7.1.	Dependencias usadas en el desarrollo, junto con sus versiones.	63
7.2.	Librerías de renderizado seleccionadas para cada estadística avanzada. . . .	74
7.3.	Acciones del <i>middleware</i> en todos los casos posibles de existencia de tokens. .	79
9.1.	Límites y características por mes del plan <i>Hobby</i> de Vercel.	85
9.2.	Variables de entorno necesarias en producción.	87

Lista de Algoritmos

6.1.	Entorno Suspense para mostrar un fallback mientras se carga el componente <code>UserProfile</code>	52
6.2.	Ejemplo de aplicación de estilos a un componente JSX usando <i>Tailwind CSS</i>	54
7.1.	Comandos de instalación y ejecución inicial del proyecto.	64
7.2.	Carga dinámica de componentes de estadísticas utilizando <code>dynamic</code> de Next.js.	74
7.3.	Cálculo de la onda combinada en <i>Índice de Interferencia</i> sumando las ondas de la frecuencia habitual y la frecuencia actual.	76
C.1.	Variables de entorno necesarios en el fichero <code>.env.local</code>	99
C.2.	Definición del <i>hook</i> personalizado <code>useFetch</code> para la obtención de datos de la API con gestión de estado y abortos de petición.	101
C.3.	Definición del componente <code>Home</code> , encargado de renderizar la página principal con las estadísticas de usuario en Spotify.	102
C.4.	Definición del componente <code>Stats</code> , encargado de gestionar y renderizar la interfaz de las estadísticas avanzadas.	103
C.5.	Definición del componente <code>StatWrapper</code> , encargado de gestionar la visualización dinámica de estadísticas en un modal.	104
C.6.	Middleware global en Next.js para la gestión y renovación de tokens de autenticación.	105
C.7.	Función para la renovación del <code>access_token</code> utilizando el <code>refresh_token</code> en la API de Spotify.	106
C.8.	Función <code>optimizeImage()</code> de optimización de imágenes con ajuste de calidad y escala.	107
C.9.	Cálculo de la trayectoria de los segmentos en el gráfico de <i>Estaciones Musicales</i>	108
C.10.	Configuración del componente <code>Stage</code> en <i>Tus Décadas</i> , permitiendo navegación y zoom dinámico.	108
C.11.	Manejo del evento de desplazamiento con la rueda del ratón en <i>Tus Décadas</i> para aplicar zoom dinámico.	109
C.12.	Generación de datos de onda sinusoidal en <i>Índice de Interferencia</i> a partir de la frecuencia de escucha del usuario.	109
C.13.	Función <code>drawWave()</code> para dibujar ondas sinusoidales animadas en <i>Índice de Interferencia</i> utilizando D3.js.	110
D.1.	Pseudocódigo del procesamiento de datos en el endpoint Hall Of Fame.	111
D.2.	Ejemplo de estructura de datos enviada en el endpoint Hall Of Fame.	112
D.3.	Pseudocódigo del procesamiento de datos en el endpoint Estaciones Musicales.	112

D.4. Ejemplo de estructura de datos enviada en el endpoint Estaciones Musicales.	113
D.5. Pseudocodigo del procesamiento de datos en el endpoint Huella del Dia.	114
D.6. Ejemplo de estructura de datos enviada en el endpoint Huella del Dia.	114
D.7. Pseudocodigo del procesamiento de datos en el endpoint La Bitacora.	115
D.8. Ejemplo de estructura de datos enviada por año en el endpoint La Bitacora.	115
D.9. Ejemplo de estructura de datos enviada por mes en el endpoint La Bitacora.	115
D.10. Ejemplo de estructura de datos enviada por dia en el endpoint La Bitacora.	116
D.11. Pseudocodigo del procesamiento de datos en el endpoint Tus Decadas.	116
D.12. Ejemplo de estructura de datos enviada en el endpoint Tus Decadas.	117
D.13. Pseudocodigo del procesamiento de datos en el endpoint Indice de Interfe-	
rencia.	118
D.14. Ejemplo de estructura de datos enviada en el endpoint Indice de Interferencia.	118

Introducción

1.1. Contexto

El uso de datos personalizados es un componente esencial en el desarrollo de aplicaciones y servicios digitales actuales. Las plataformas buscan ofrecer experiencias ajustadas a las preferencias de los usuarios, utilizando grandes volúmenes de datos para generar contenido adaptado. En este contexto, *Spotify* se ha consolidado como una de las plataformas más destacadas de *streaming* musical, no solo por su extenso catálogo, sino también por su capacidad de ofrecer recomendaciones y estadísticas de uso personalizadas.

Una de las características más valoradas por los usuarios de *Spotify* es la posibilidad de acceder a estadísticas personales que les permiten comprender mejor sus hábitos de escucha. En 2016, como producto de una campaña de marketing viral, *Spotify* lanzó *Spotify Wrapped*, un resumen anual de los datos musicales de cada usuario, que generó un gran interés y participación en las redes sociales. Sin embargo, el acceso a estas estadísticas solo está disponible en el mes de diciembre, lo que genera una oportunidad para desarrollar herramientas complementarias que ofrezcan este tipo de análisis de manera más frecuente.

El acceso a estos datos es posible gracias a la *Web API* pública que *Spotify* pone a disposición de los desarrolladores. De esta manera, es posible obtener una amplia gama de datos sobre el comportamiento del usuario, como sus canciones más escuchadas, artistas favoritos y playlists. Además, la API ofrece acceso a muchos datos adicionales no utilizados en *Spotify Wrapped*, pero que, realizando diferentes combinaciones, permiten crear nuevas formas enriquecidas de análisis que presenten la información de manera más atractiva y personalizada.

También es necesario mencionar, que el crecimiento de las aplicaciones web interactivas ha sido posible gracias a tecnologías y frameworks modernos, los cuales permiten crear interfaces rápidas y eficientes, optimizando el rendimiento y la experiencia de usuario y simplificando el desarrollo. Además, los sistemas de integración y despliegue continuo (CI/CD) facilitan la entrega de aplicaciones de manera automatizada y escalable, garantizando que estén siempre actualizadas y accesibles para los usuarios.

1.2. Motivación

La elección de este Trabajo de Fin de Grado surge de un interés personal que he tenido desde el inicio de mi carrera universitaria. Desde el primer año, he tenido la idea base para implementar un servicio en torno a la *Web API de Spotify*. Sin embargo, en ese momento no contaba con los conocimientos necesarios para llevarlo a cabo. Ahora, tras completar la carrera, me siento con las capacidades necesarias para poder realizar dicho proyecto y materializar esta idea.

Como usuario habitual de *Spotify*, siempre me ha fascinado la función de *Spotify Wrapped*. No obstante, me resulta limitante que esta información solo esté disponible durante un breve periodo del año. Existen otras herramientas y páginas web que analizan datos de *Spotify*, pero suelen ofrecer estadísticas genéricas y demasiado básicas que carecen de profundidad. Estoy convencido de que es posible obtener estadísticas mucho más interesantes y, conversando con compañeros y otros usuarios, existe un interés general por acceder a estas estadísticas en cualquier momento, no solo una vez al año. La música que cada individuo escucha es algo personal y, a menudo, forma parte de su identidad. Por ello, considero que este proyecto no solo es interesante y motivador para mí, sino que también puede aportar valor a otros usuarios que comparten esta misma idea.

La posibilidad de crear un proyecto que me interese de principio a fin, y que yo mismo desearía utilizar, es una gran motivación. Además, la selección de tecnologías que he decidido emplear, como se detallará más adelante, no solo son ampliamente demandadas en el mercado actual, sino que también contribuyen a mi crecimiento profesional y enriquecen mis conocimientos.

1.3. Objetivos del Proyecto

Para empezar a caracterizar el proyecto de manera concreta, en este apartado se definen los objetivos que guiarán el desarrollo. El **objetivo general** de este proyecto es desarrollar una plataforma web interactiva que permita a los usuarios de *Spotify* acceder a las visualizaciones y análisis de sus datos musicales personales cuando lo deseen. Para alcanzar este objetivo general, se plantean los siguientes **objetivos específicos**:

- Analizar la API de *Spotify* para obtener los datos necesarios.
- Desarrollar la lógica necesaria para filtrar, organizar y transformar los datos obtenidos, preparándolos para su representación gráfica.
- Diseñar una interfaz de usuario intuitiva, interactiva y adaptable a diferentes dispositivos.
- Adoptar tecnologías modernas como *Next.js* y *TypeScript* para beneficiarse de las ventajas que ofrecen.
- Implementar prácticas de CI/CD usando la plataforma de *Vercel*.
- Implementar políticas de seguridad con respecto a los datos de usuarios establecidas por *Spotify* para desarrolladores.
- Documentar el proceso de desarrollo.

1.4. Estructura de la Memoria

La estructura de esta memoria refleja las diferentes etapas por las que atraviesa un proyecto de desarrollo software. Cada capítulo aborda aspectos clave que contribuyen al logro de los objetivos propuestos.

En la [Introducción](#) se ha establecido el contexto del proyecto, la motivación que impulsa su realización y los objetivos. Para terminar de definir el contexto en el que se está desarrollando, en el [Contexto Competitivo](#) se realizará un análisis de las soluciones ya existentes. A continuación, en la [Planificación](#), se define el alcance del proyecto y se aborda todo lo relacionado con la gestión de tareas, riesgos y calidad del proyecto. Además, se presentan las tecnologías que se han utilizado durante todo el desarrollo.

El [Análisis](#) se centra en el estudio de la API de *Spotify*, el cual es **fundamental para el desarrollo del proyecto**. Se especifican los requisitos y se presentan los casos de uso que guiarán el desarrollo de la aplicación. En conjunto con el capítulo anterior, en el [Diseño](#) se describe la arquitectura del sistema y la estructura de la interfaz de usuario (UI), tanto a nivel técnico como visual. Además, se abordan aspectos de seguridad y se planifica el diseño de las pruebas.

La [Implementación](#) detalla el proceso de desarrollo de la aplicación, incluyendo las decisiones tomadas durante esta fase. Se describen los retos enfrentados y cómo se han resuelto. Seguido, en el capítulo de las [Pruebas](#) se exponen las pruebas realizadas y se analizan los resultados obtenidos. Como consecuencia lógica, en [Despliegue](#) se explica el proceso de puesta en producción de la aplicación y las estrategias de CI/CD implementadas.

Finalmente, en [Conclusiones](#) se hace una reflexión sobre los resultados alcanzados, evaluando el cumplimiento de los objetivos iniciales, el seguimiento en comparación a la planificación inicial y se discuten las lecciones aprendidas y se proponen líneas futuras de trabajo.

Mediante este flujo, se espera que el público lector no tenga problemas para seguir la línea de pensamiento que se ha llevado a cabo y que cada apartado quede aclarado o, en menor medida, justificado por los capítulos anteriores.

CAPÍTULO 2

Contexto Competitivo

Antes de proceder con la planificación del proyecto, es un buen ejercicio analizar las alternativas similares que actualmente existen en el mercado. Como se menciona en la sección de [Contexto](#), la mayoría de otras webs ofrecen estadísticas muy básicas, como visualizar los *top géneros, artistas y álbumes* del usuario, además de su *historial de reproducción* más reciente. Cada una de estas funcionalidades corresponde a llamadas directas a la *Web API de Spotify*, siendo estadísticas triviales de implementar, con una mínima necesidad de procesamiento de datos.

Estadísticas	stats.fm	Obscurify	Stats for Spotify	Replayify	Zodiac Affinity	Music Scapes
Básicas	Top canciones	Top canciones	Top canciones	Top canciones		
	Top artistas	Top artistas	Top artistas	Top artistas		
	Top géneros	Top géneros	Top géneros	Top géneros		
	Historial reciente		Historial reciente	Historial reciente		
Avanzadas	Tiempo escuchado					
	Número de artistas					
	Número de álbumes					
Creativas	Porcentaje por décadas					
	Escuchas por hora del día					
	Rating de "obscurity"					
	Top artistas "oscuros"					
	Top canciones "oscuras"					
	Tus estados de ánimo					
					5 canciones según tu signo zodiacal	Imagen basada en las propiedades de las canciones que más escuchas

Tabla 2.1: Comparativa de funcionalidades ofrecidas por otros servicios afines.

Una de las opciones más populares es la aplicación web *stats.fm*¹ (anteriormente *Spotistats*). Esta ofrece las funcionalidades básicas mencionadas a todos los usuarios, pero, mediante un plan de pago, se habilitan gráficas más avanzadas. Estas gráficas no se obtienen directamente de la API, sino que requieren que el usuario descargue manualmente los datos históricos guardados por *Spotify* y los suba a la página web como un archivo comprimido.

¹stats.fm: <https://spotistats.app/>

De esta manera, se generan estadísticas relativamente más complejas, pero que aún carecen de “creatividad” en su presentación.

Otro servicio notable es *Obscurify*², que se centra en la “oscuridad” o “rareza” de la música que el usuario escucha. El concepto principal de *Obscurify* es identificar las canciones y artistas que son menos populares entre otros usuarios, clasificándolos como más “oscuros”. Esta funcionalidad permite que el usuario se sienta especial al escuchar música que no es común. Sin embargo, su enfoque está limitado en su mayoría a este concepto, lo que deja fuera otras formas de visualización o análisis más amplios y variados.

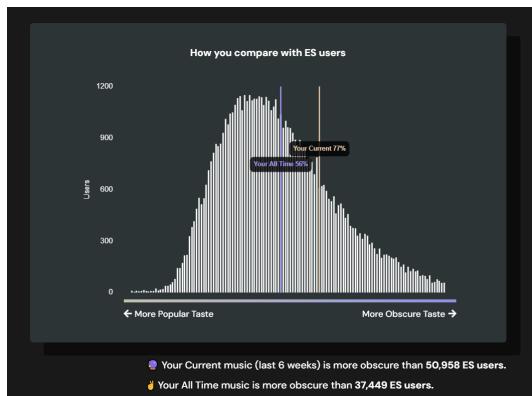


Figura 2.1: Ejemplo de estadística de “oscuridad” de *Obscurify*.

Además de estas páginas principales, existen otras que se enfocan toda su funcionalidad en una única característica original. Dos ejemplos destacados son *Zodiac Affinity*³ y *MusicScapes*⁴. La primera genera una recomendación de cinco canciones en base a los hábitos de escucha del usuario y su signo zodiacal, mientras que la segunda crea de manera procedural una imagen basada en diferentes propiedades de las canciones que el usuario ha escuchado recientemente. Estas páginas no ofrecen ninguna funcionalidad adicional, limitándose a esa única característica. Aunque hay otros servicios similares, he elegido estos dos ejemplos por su aspecto más “acabado”, ya que muchas otras alternativas se presentan más como una prueba de concepto o una demo, en lugar de páginas completamente desarrolladas.

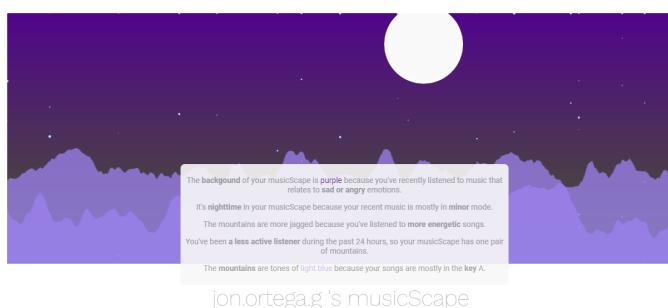


Figura 2.2: Imagen generada por MusicScapes.

²Obscurify: <https://www.obscurifymusic.com/>

³Zodiac Affinity: <https://zodiacaffinity.eu/>

⁴MusicScapes: <https://musicscapes.herokuapp.com/>

Planificación

3.1. Alcance

Definir con precisión el alcance es fundamental para asegurar que el desarrollo se ajuste a los objetivos propuestos y se realice dentro de los recursos y tiempos establecidos. Para ello, en esta sección se delimitarán las funcionalidades, exclusiones y limitaciones que se esperan en este proyecto.

3.1.1. Funcionalidades Incluidas

En la plataforma web se ofrecen las siguientes funcionalidades principales:

- Autenticación segura mediante las credenciales de *Spotify*.
- Home o Panel Inicial donde se muestran la información y estadísticas básicas de la cuenta.
- Gráficos interactivos para representar los datos del usuario obtenidos en tiempo real.
- Interfaz intuitiva y responsive.
- Cierre de sesión seguro.

3.1.2. Exclusiones

Para establecer expectativas claras sobre el alcance del proyecto, se detallan a continuación las funcionalidades que **no** serán incluidas en la plataforma web:

- No se desarrollarán aplicaciones nativas de otras plataformas como móvil, PC, Mac o Linux; el acceso será exclusivamente a través de la web.
- Aunque se seguirá un diseño intuitivo, no se implementarán funcionalidades específicas de accesibilidad avanzadas como compatibilidad con lectores de pantalla o navegación por teclado.
- La plataforma se enfoca exclusivamente en la integración con *Spotify*; se excluyen todos los otros servicios de streaming como *Apple Music*, *Deezer*, etc.

- No se almacenarán de forma persistente datos personales del usuario en servidores propios más allá de lo necesario para la sesión actual; todos los datos se obtendrán directamente de la API de Spotify y se manejarán en tiempo real.
- Se excluye el desarrollo de funcionalidades relacionadas con la interacción social (envío de mensajes, compartir estadísticas, rankings entre usuarios, etc.) dentro o a través de la plataforma, ya que superarían el alcance recogido dentro de un TFG.
- La interfaz de usuario estará disponible únicamente en español y no se ofrecerá soporte para otros idiomas.

3.1.3. Limitaciones

Durante el desarrollo del proyecto, se han identificado las siguientes limitaciones que han afectado al alcance y a las funcionalidades de la web:

- Las políticas de seguridad de *Spotify* impiden el almacenamiento persistente de datos personales, limitando funcionalidades que requieran conservar información del usuario entre sesiones.
- El procesamiento de los datos se ve limitado por los recursos computacionales que la nube de *Vercel* ofrece, descartando técnicas avanzadas como el aprendizaje automático.
- El tiempo y los recursos disponibles para el desarrollo del proyecto son finitos, lo que ha obligado a priorizar funcionalidades esenciales y descartar características adicionales.
- Al hacer uso de una API de terceros, todas las funcionalidades necesitan una conexión activa a Internet para poder funcionar de manera correcta.

3.2. Gestión de Tareas

Una vez definido el alcance, es necesario detallar las tareas requeridas para el desarrollo del proyecto. En esta sección se caracterizará todo lo necesario en relación a las tareas, incluyendo su definición, relaciones y tiempos asignados, para asegurar una gestión estructurada y alineada con los objetivos del proyecto.

3.2.1. Descripción de Tareas

Para gestionar de manera efectiva el conjunto de actividades, se ha elaborado una Estructura de Desglose de Trabajo (EDT). Esta EDT (figura 3.1) proporciona una visión general de las principales áreas de trabajo, desglosando el proyecto en paquetes específicos que abarcan cada tarea esencial, facilitando así la gestión.

En este caso, el proyecto se organiza en cinco áreas principales, que abarcan todas las fases del desarrollo de la aplicación; abordando tanto las tareas relacionadas con la creación de la aplicación en sí misma (el producto final) como aquellas enfocadas en la gestión y redacción del proyecto para su documentación. Esta estructura garantiza una distribución clara de las tareas, cubriendo tanto los aspectos técnicos como los organizativos.

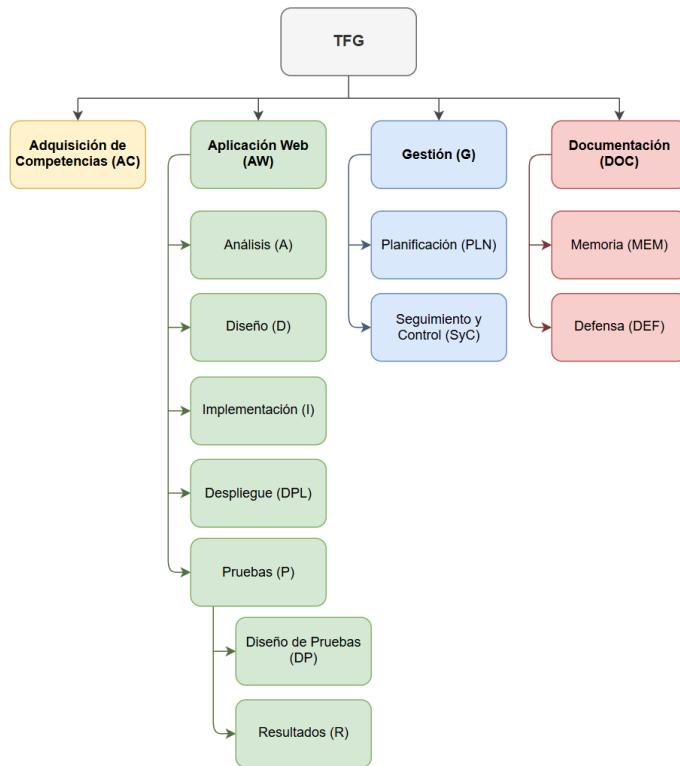


Figura 3.1: Diagrama EDT con los paquetes de trabajo del proyecto.

A continuación se detalla cada paquete de trabajo y las tareas correspondientes contenidas en cada una:

3.2.1.1. Adquisición de Competencias (AC):

Este paquete de trabajo incluye todas las tareas necesarias para adquirir el conocimiento sobre las tecnologías y herramientas clave para el desarrollo del proyecto.

- **AC.1:** Aprender *TypeScript*, *React.js* y *Next.js* para el desarrollo de la aplicación web.
- **AC.2:** Estudiar el uso de *Vercel* para el hosting y despliegue de la aplicación.
- **AC.3:** Hacer un reconocimiento inicial de la *Web API* de *Spotify*.

3.2.1.2. Aplicación Web (AW):

En este paquete se engloban todas las fases de desarrollo de la aplicación web, desde la planificación inicial hasta el despliegue final.

■ Análisis (A):

- **AW.A.1:** Estudiar y analizar en profundidad la *Web API* de *Spotify* para determinar el alcance y sus limitaciones.
- **AW.A.2:** Definir los requisitos funcionales y no funcionales del sistema.
- **AW.A.3:** Desarrollar los principales casos de uso del sistema.

■ Diseño (D):

- **AW.D.1:** Diseñar la arquitectura del sistema.
- **AW.D.2:** Crear un diagrama de componentes React para establecer la jerarquía y realizar un diseño general de la interfaz de usuario.
- **AW.D.3:** Definir los diagramas de secuencia de los casos principales.
- **AW.D.4:** Realizar una gestión de la seguridad y asegurar que se implementarán las medidas definidas por *Spotify* para el uso de la API.

■ Implementación (I):

- **AW.I.1:** Implementar el login de la página web, usando el protocolo OAuth 2.0 implementado por *Spotify*.
- **AW.I.2:** Implementar el panel inicial (dashboard) de la web.
- **AW.I.3:** Implementar la sección principal de estadísticas.
- **AW.I.4:** Implementar la funcionalidad de cerrar sesión.
- **AW.I.5:** Realizar optimizaciones y correcciones en la implementación.

■ Despliegue (DPL):

- **AW.DPL.1:** Configurar el despliegue en *Vercel* para crear un proceso automático de despliegue.
- **AW.DPL.2:** Monitorear el funcionamiento del despliegue y los logs.

■ Pruebas (P):

- **AW.P.DP:** Planificar y diseñar pruebas unitarias, de integración y de carga para evaluar el rendimiento y la estabilidad de la aplicación.
- **AW.P.R:** Realizar las pruebas planificadas y documentar los errores encontrados. En caso de que sea posible, implementar las correcciones necesarias.

3.2.1.3. Gestión (G):**■ Planificación (PLN):**

- **G.PLN.1:** Realizar una primera estimación de tiempos de las tareas generales.
- **G.PLN.2:** Establecer el alcance inicial del proyecto según las características del producto seleccionadas.
- **G.PLN.3:** Definir la planificación del proyecto.
- **G.PLN.4:** Revisar y, si fuera necesario, modificar la planificación.

■ Seguimiento y Control (SyC):

- **G.SyC.1:** Conversaciones y comentarios de la tutora a lo largo del desarrollo.
- **G.SyC.2:** Elaboración de un documento para registrar las actividades y dedicaciones realizadas a lo largo del proyecto.
- **G.SyC.3:** Comparación de los datos del seguimiento con los de la planificación, identificación de las desviaciones y riesgos significativos.

3.2.1.4. Documentación (DOC):

Este paquete agrupa las tareas necesarias para la elaboración de la memoria y la preparación de la defensa del proyecto.

■ **Memoria (MEM):**

- **DOC.MEM.1:** Preparar el entorno de trabajo en L^AT_EX utilizando *Visual Studio Code* y establecer la estructura básica de la memoria a partir de la plantilla proporcionada por la facultad.
- **DOC.MEM.2:** Redactar la memoria.

■ **Defensa (DEF):**

- **DOC.DEF.1:** Identificar los puntos y conceptos clave que se presentarán en la defensa.
- **DOC.DEF.2:** Crear los elementos visuales de apoyo para la defensa.
- **DOC.DEF.3:** Preparar y ensayar la defensa.

3.2.2. Dedicaciones

A continuación, en la tabla 3.1, se presentan las horas estimadas para las tareas descritas en el apartado anterior. También se muestran las sumas de las dedicaciones por paquete de trabajo y la suma total de horas que se espera que lleve el desarrollo del proyecto completo.

Tarea	Horas
Adquisición de Competencias (AC)	24
AC.1: TS, React, Next.js	18
AC.2: Vercel	2
AC.3: Spotify API	4
Análisis (A)	30
AW.A.1: Estudio de Spotify API	16
AW.A.2: Captura de requisitos	8
AW.A.3: Casos de uso	6
Diseño (D)	37
AW.D.1: Arquitectura	5
AW.D.2: Interfaz de usuario	10
AW.D.3: Lógica de negocio	18
AW.D.4: Seguridad	4
Implementación (I)	80
AW.I.1: Login	7
AW.I.2: Home	15
AW.I.3: Estadísticas	40
AW.I.4: Logout	2
AW.I.5: Optimizaciones	16
Despliegue (DPL)	3
AW.DPL.1: Configurar CI/CD	2
AW.DPL.2: Monitorear logs	1
Pruebas (P)	30
AW.P.DP: Diseñar pruebas	18
AW.P.R: Resultados	12
Gestión (G)	20
G.PLN: Planificación	10
G.SyC: Seguimiento y Control	10
Documentación (DOC)	90
DOC.MEM: Memoria	80
DOC.DEF: Defensa	10
TOTAL	314

Tabla 3.1: Tabla con las estimaciones de tiempo por paquete de trabajo y tarea del proyecto.

3.2.3. Dependencias entre Tareas

En la figura 3.2 se muestra un diagrama representando las dependencias que existen entre las diferentes tareas. De esta manera, se puede apreciar de forma visual las tareas que requieren la finalización de una o varias tareas para su comienzo.

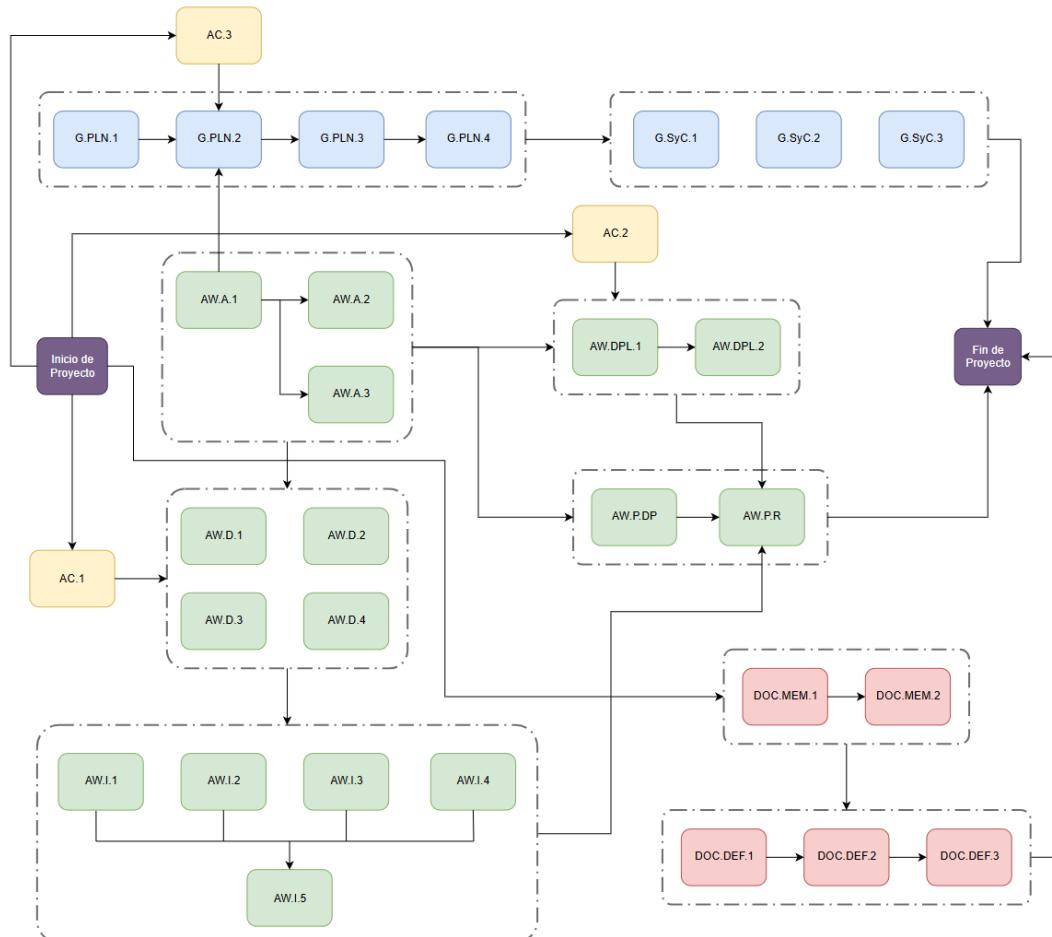


Figura 3.2: Diagrama de dependencias entre las tareas y paquetes de trabajo del proyecto.

Como se puede apreciar, el proyecto debe iniciarse con las tareas de la planificación (paquete de trabajo **G.PLN**) y análisis (**AW.A**), al igual que el desarrollo de las tareas relacionadas con la memoria (**DOC.MEM**), que se realizan desde el inicio del proyecto hasta casi la finalización del TFG. Para ordenar temporalmente todas las tareas, en la siguiente sección se tratarán los períodos de desarrollo de cada una.

3.2. Gestión de Tareas

3.2.4. Periodos de Desarrollo e Hitos

En esta sección se presenta el diagrama Gantt del proyecto (figura 3.3), el cual ilustra los períodos de realización de las tareas y los paquetes de trabajo; siempre teniendo en cuenta las dedicaciones asignadas y las dependencias entre las tareas que se han establecido en la sección anterior. Además, se destacan los hitos del proyecto, permitiendo una visualización clara y organizada del cronograma planificado.

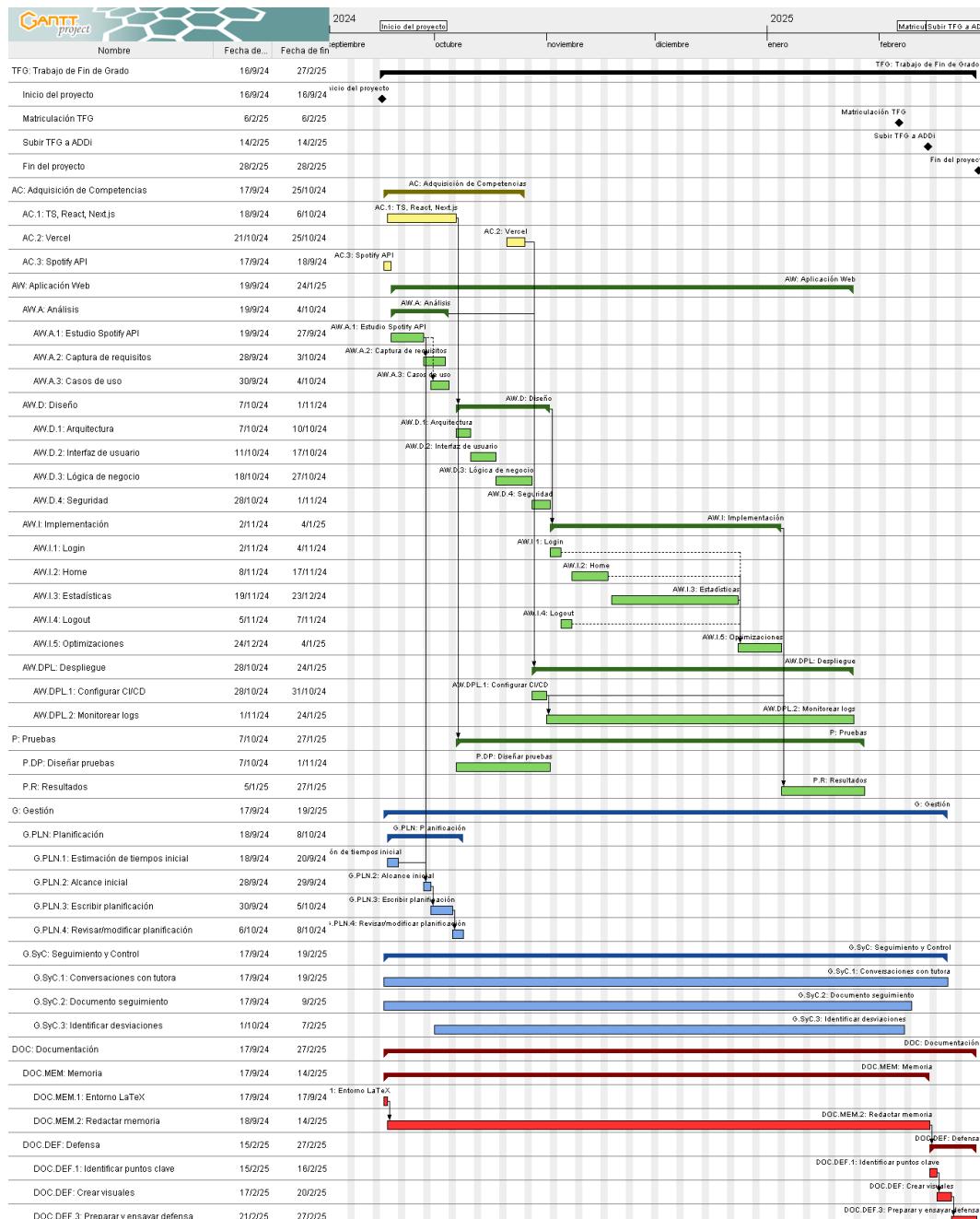


Figura 3.3: Diagrama Gantt con los tiempos de realización de las tareas y paquetes de trabajo.

3.3. Gestión de Riesgos

La gestión de riesgos desempeña un papel crucial en cualquier proyecto, ya que permite anticiparse a posibles inconvenientes, definiendo estrategias adecuadas para minimizar su impacto sobre el proyecto.

A continuación, se presentan los riesgos identificados, especificando para cada uno de ellos la probabilidad de que se materialice, el impacto que podría ocasionar, las medidas implementadas para prevenirlos y las acciones que se llevarían a cabo en caso de que se acabe materializando alguno de ellos.

R01: Limitaciones con respecto a los datos disponibles de la API

Este riesgo se refiere a la posible falta de datos suficientes o adecuados para implementar ciertas funcionalidades previstas, como estadísticas o visualizaciones concretas.

- **Probabilidad:** Media.
- **Impacto:** Medio.
- **Prevención:** Analizar detenidamente los endpoints de la API antes de planificar funcionalidades dependientes de datos específicos.
- **Plan de mitigación:** Proponer gráficos o funcionalidades alternativas que no dependan de esos datos.

R02: Cambios en la política de acceso a la API

Como *Spotify* se reserva el derecho de modificar en cualquier momento las políticas de uso de su API, existe la posibilidad de que algunas funcionalidades del proyecto se vean afectadas o limitadas debido a cambios en los permisos o en la disponibilidad de ciertos endpoints.

- **Probabilidad:** Baja.
- **Impacto:** Alto.
- **Prevención:** Revisar con frecuencia la documentación oficial y adaptar el proyecto a los permisos disponibles desde el inicio.
- **Plan de mitigación:** Ajustar el alcance del proyecto para trabajar con los datos que sigan siendo accesibles y documentar los cambios en la memoria del TFG.

R03: Interrupciones en la disponibilidad de servicios de terceros

Este riesgo engloba posibles problemas de disponibilidad en servicios externos críticos para el proyecto, como la API de *Spotify* o el servicio de hosting de *Vercel*.

- **Probabilidad:** Baja.
- **Impacto:** Alto.

- **Prevención:** Mantener una planificación que contemple margen suficiente para posibles retrasos causados por la indisponibilidad de estos servicios. Además, probar despliegues en un entorno local para avanzar en el desarrollo mientras el servicio de hosting se restablece.
- **Plan de mitigación:** Continuar con el desarrollo de los aspectos que no dependan de estos servicios y retomar las tareas una vez se solucione la interrupción.

R04: Incompatibilidad de versiones de las tecnologías a utilizar

Este riesgo está relacionado con posibles problemas de compatibilidad entre *TypeScript*, *React*, *Next.js* y las librerías utilizadas.

- **Probabilidad:** Media.
- **Impacto:** Alto.
- **Prevención:** Investigar y verificar compatibilidades antes de seleccionar versiones específicas. Evitar usar, en la medida de lo posible, versiones muy recientes que puedan tener problemas de estabilidad y de soporte por parte de otras herramientas.
- **Plan de mitigación:** Actualizar o cambiar las herramientas o librerías problemáticas por alternativas compatibles y realizar pruebas exhaustivas después de cada cambio.

R05: Dificultad en el aprendizaje de las herramientas a utilizar

Al trabajar con herramientas, frameworks y tecnologías con los que no se ha tenido un contacto previo, se pueden ocasionar retrasos por el proceso de aprendizaje.

- **Probabilidad:** Media.
- **Impacto:** Medio.
- **Prevención:** Reservar un tiempo de adquisición de conocimientos y realizar pruebas antes de comenzar con las implementaciones críticas.
- **Plan de mitigación:** Consultar documentación oficial, foros o buscar ayuda en comunidades de desarrollo en línea para resolver dudas rápidamente. Si no es suficiente, se puede avanzar con otra tarea que no dependa de la resolución del problema actual, permitiendo ganar tiempo mientras se sigue investigando la solución al inconveniente técnico.

R06: Dificultad para compaginar el proyecto con las obligaciones académicas

Este riesgo hace referencia a la posibilidad de tener problemas a la hora de gestionar el tiempo disponible para dedicar al proyecto, ya que se está cursando una asignatura en paralelo.

- **Probabilidad:** Media.
- **Impacto:** Alto.
- **Prevención:** Planificar un calendario detallado y realista que reserve horas específicas para trabajar en el proyecto, priorizando las tareas críticas.
- **Plan de mitigación:** Ajustar la planificación redistribuyendo tareas menos prioritarias y minimizando así el impacto en el desarrollo del proyecto.

3.4. Herramientas y Tecnologías

Durante el desarrollo del TFG se usarán diferentes herramientas, tanto para la implementación de la aplicación web como en la planificación y redacción de la memoria. Cada una de ellas ha sido seleccionada por su idoneidad para la tarea en cuestión. A continuación se mencionan dichas herramientas y tecnologías, su función en el proyecto y una justificación de su selección.

3.4.1. TypeScript

Lenguaje de programación que extiende JavaScript, añadiendo un sistema de tipos estáticos y funcionalidades adicionales que mejoran la robustez y mantenibilidad del código. Permite detectar errores durante el desarrollo, en lugar de en tiempo de ejecución, lo que reduce significativamente los problemas en aplicaciones grandes y complejas.

En este proyecto, *TypeScript* se utiliza como lenguaje principal para implementar la aplicación web, gracias a sus múltiples ventajas:

- **Tipado estático:** Permite definir explícitamente los tipos de variables, funciones y componentes, evitando errores comunes como el mal uso de datos o la incompatibilidad entre módulos.
- **Mejor autocompletado y documentación:** Herramientas como *VSCode* ofrecen un autocompletado más preciso y una navegación clara del código, mejorando la productividad del desarrollo.
- **Compatibilidad con JavaScript:** Al ser un superconjunto de JavaScript, es totalmente compatible con cualquier código JavaScript existente, facilitando la integración.
- **Detección temprana de errores:** Como ya se ha mencionado, gracias a su compilador, los errores se detectan antes de ejecutar el código, garantizando una mayor calidad en las entregas.

3.4.2. React.js

Biblioteca de JavaScript desarrollada por *Facebook*, diseñada para la creación de interfaces de usuario modernas y dinámicas. Es ampliamente reconocida por su enfoque declarativo, que facilita la construcción de componentes reutilizables. En este proyecto, *React.js* sirve como base para desarrollar la interfaz de usuario de la aplicación web, aprovechando su capacidad para gestionar de manera eficiente la interacción entre los componentes y el estado de la aplicación.

3.4.3. Next.js

Next.js es un framework de desarrollo web basado en *React.js*, diseñado para facilitar la creación de aplicaciones modernas, escalables y de alto rendimiento. En este proyecto, se utiliza para implementar la página web principal, proporcionando funcionalidades avanzadas como el renderizado híbrido (estático y dinámico), rutas dinámicas y optimización automática de recursos.

El framework emplea varias tecnologías clave para garantizar un entorno de desarrollo eficiente y una construcción optimizada de la aplicación:

- **SWC:** Compilador de alto rendimiento escrito en *Rust* utilizado durante el proceso de construcción.
- **Turbopack:** Empaquetador moderno que reemplaza a *Webpack*, ofreciendo tiempos de desarrollo significativamente más rápidos.
- **ESLint:** Herramienta integrada para el análisis estático de código, que garantiza la detección de errores y el cumplimiento de buenas prácticas.
- **Node.js:** Entorno de ejecución de JavaScript necesario tanto para el desarrollo como para la compilación y el despliegue de la aplicación.

3.4.4. TailwindCSS

Framework de CSS de utilidad que permite crear interfaces de usuario de manera rápida mediante clases predefinidas. Su integración con *Next.js* permite una optimización automática de los estilos, eliminando clases no utilizadas durante el proceso de compilación para reducir el tamaño final del archivo CSS.

3.4.5. Spotify Web API

Siendo el principal servicio utilizado en este proyecto, es una interfaz de programación que permite acceder a diversos datos de usuario y a información relacionada con el contenido disponible en la plataforma de *Spotify*. Esta API ofrece múltiples endpoints que proporcionan acceso a los datos, los cuales gozan de una muy buena documentación en la página web oficial.

3.4.6. Git y GitHub

Git es un sistema de control de versiones ampliamente utilizado en el desarrollo de software, que permite gestionar y realizar un seguimiento de los cambios en el código fuente del proyecto de manera eficiente. Por su parte, *GitHub* es una plataforma basada en la nube que facilita el almacenamiento y la colaboración en proyectos gestionados con *Git*. Juntas, estas herramientas forman una combinación esencial en cualquier proceso de desarrollo de software, proporcionando un entorno robusto para el control de versiones y el respaldo seguro del código.

3.4.7. Vercel

Plataforma de hosting y despliegue continuo, optimizada para aplicaciones web basadas en frameworks como *Next.js*, *React.js*, *Vue.js* y otros. En este proyecto, *Vercel* se utiliza para alojar y mantener la página web, asegurando un proceso de despliegue automatizado gracias a su integración nativa con *GitHub*.

Una de las principales ventajas de *Vercel* es su integración directa con repositorios en *GitHub*. Esto permite que cada vez que se realiza un cambio en el código fuente (como un *push* en la rama principal), se active un flujo de trabajo automatizado para desplegar la versión más reciente de la aplicación. Aunque *Vercel* cuenta con flujos de trabajo internos para automatizar los despliegues, también es posible personalizarlos utilizando *GitHub Actions*, lo que ofrece un mayor control sobre el proceso de CI/CD.

Además de la facilidad de despliegue, *Vercel* proporciona funcionalidades avanzadas que resultan beneficiosas para este proyecto:

- **Renderizado optimizado:** Soporte nativo para el renderizado estático (*Static Site Generation*) y dinámico (*Server-Side Rendering*), fundamentales para proyectos basados en *Next.js*.
- **Red global de distribución de contenido (CDN):** Los recursos de la aplicación se sirven desde una red global de servidores, garantizando tiempos de carga rápidos para los usuarios en diferentes ubicaciones.
- **Escalabilidad automática:** La plataforma ajusta automáticamente los recursos según la demanda, lo que elimina la necesidad de gestionar manualmente la infraestructura.

3.4.8. Visual Studio Code (VSCode)

Editor de código empleado para el desarrollo del proyecto. Se trata de un editor de código abierto y gratuito, creado por Microsoft y diseñado para soportar una amplia variedad de lenguajes de programación y tecnologías. Para el beneficio del proyecto, cuenta con la integración nativa de *TypeScript* y *React.js*.

Además, también se emplea para la edición en local de la memoria del TFG escrita en *L^AT_EX*, siendo una mejor alternativa a otros editores específicos del lenguaje y la preferencia personal a los editores locales en comparación con las herramientas online como *Overleaf*. La posibilidad de gestionar tanto el código de la aplicación como la memoria en un mismo entorno mejora significativamente la organización y productividad durante el desarrollo.

3.4.9. L^AT_EX (Mik^TeX)

Sistema de preparación de documentos ampliamente utilizado en entornos académicos y técnicos por su capacidad para generar documentos de alta calidad tipográfica. En este proyecto, *L^AT_EX* se utiliza para la escritura y edición de la memoria del TFG.

Para la compilación de los documentos, se emplea *Mik^TeX*, una distribución de *L^AT_EX* que incluye las herramientas necesarias para gestionar paquetes y generar archivos en formato PDF. Como ya se ha mencionado, esta distribución se combina con *VSCode* para una agradable experiencia de edición.

3.4.10. Chart.js, D3.js y Konva

Librerías de visualización utilizadas en la sección de estadísticas de la aplicación web. *Chart.js* permite crear gráficos interactivos de forma sencilla, como barras, líneas y áreas. *D3.js* ofrece mayor control sobre la manipulación de datos y gráficos personalizados. Por último, *Konva* se centra en la gestión avanzada de gráficos en *canvas*, facilitando interacciones y animaciones en los elementos visuales.

3.4.11. Jest.js y K6

Herramientas utilizadas para realizar las pruebas de la aplicación. *Jest.js* es un framework de testing desarrollado por *Facebook*, diseñado para realizar pruebas unitarias e integración en aplicaciones JavaScript y *TypeScript*, permitiendo detectar y corregir errores antes del despliegue. Por otro lado, *K6* es una herramienta de pruebas de carga (*load testing*) que evalúa el rendimiento de la aplicación simulando distintos niveles de tráfico. Aunque no se espera un tráfico elevado inicialmente, estas pruebas son una buena práctica para garantizar la calidad del producto final.

3.4.12. Microsoft Excel

Herramienta muy popular para la edición de hojas de cálculo, ampliamente utilizada en diferentes ámbitos debido a su versatilidad y funcionalidades avanzadas. Gracias a su capacidad para generar tablas claras y visuales, se ha empleado para crear las tablas que se incluyen en la memoria del TFG, las cuales se exportan y adaptan fácilmente para su posterior integración en el documento de *LATEX*, complementando el contenido técnico con representaciones más visuales.

3.4.13. Draw.io

Herramienta web utilizada para la creación de diagramas incluidos en la memoria del TFG. Su interfaz intuitiva permite elaborar diagramas técnicos con facilidad. Estos diagramas ayudan a ilustrar conceptos del proyecto, facilitando la comprensión de los aspectos técnicos por parte del lector.

Estudio de la API de Spotify

4.1. Descripción General

La API de Spotify es el núcleo del proyecto, ya que proporciona acceso a los datos necesarios para generar las estadísticas y visualizaciones que constituyen el objetivo principal de este trabajo. Es una interfaz que permite a las aplicaciones externas interactuar con la plataforma de manera programática. Su propósito principal es proporcionar acceso estructurado a los datos, permitiendo a los desarrolladores integrar funcionalidades avanzadas en sus propias aplicaciones. A través de esta API, es posible obtener información sobre las canciones, artistas, álbumes, listas de reproducción y **datos personalizados del usuario**, como sus preferencias musicales o sus hábitos de escucha. Gracias a estos últimos, es posible ofrecer una experiencia personalizada para cada usuario.

Entre las características técnicas más destacadas se encuentran:

- **Tipo de API:** RESTful.
- **Protocolo:** HTTPS.
- **Métodos soportados:** GET, POST, PUT y DELETE.
- **Formato de respuesta:** JSON
- **Seguridad:** Acceso protegido mediante OAuth 2.0.

Gracias a estas características, la API de Spotify facilita la manipulación de datos por las respuestas en formato JSON, al tiempo que garantiza la seguridad y privacidad del usuario mediante el uso del protocolo OAuth 2.0. Estas cualidades la convierten en una herramienta versátil y segura, capaz de adaptarse a una amplia variedad de proyectos.

4.2. Autenticación y Autorización

Antes de avanzar, es importante entender que en este proceso hay dos conceptos clave: **autenticación** y **autorización**. Aunque están relacionados, existen ciertas diferencias importantes:

- **Autenticación:** Es el paso en el que se verifica la identidad del usuario. En este caso, ocurre cuando el usuario inicia sesión en Spotify para confirmar que es quien dice ser. Este proceso es transparente para el desarrollador ya que Spotify, mediante OAuth 2.0, se encarga de gestionarlo.
- **Autorización:** Es el paso en el que el usuario concede permisos para que la aplicación acceda a ciertos recursos de su cuenta. Este proceso es clave, ya que sin estos permisos, la aplicación no podría acceder a los datos necesarios para ofrecer sus funcionalidades.

Una vez que la aplicación obtiene la autorización, Spotify permite el acceso a los datos solicitados de forma controlada. Un concepto fundamental en esta etapa son los **scopes**, que determinan exactamente qué datos y funcionalidades están disponibles para la aplicación.

Scopes: Controlando el Acceso a los Recursos

Los scopes (alcances) permiten a los usuarios tener la tranquilidad de que únicamente compartirán la información que han autorizado explícitamente. Cuando un programador configura el flujo de autorización, debe especificar los scopes necesarios (de un total de 24) para que la aplicación pueda acceder a los recursos protegidos. En función de los scopes solicitados, Spotify mostrará al usuario una pantalla indicando qué permisos específicos se están requiriendo (figura 4.1). El usuario puede entonces aceptar o rechazar estos permisos.

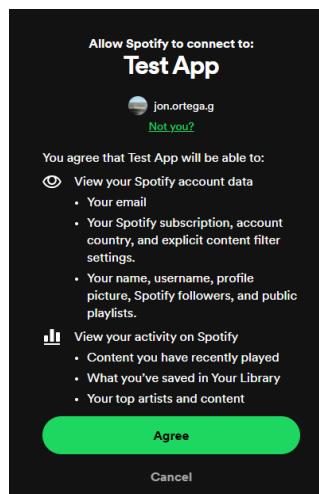


Figura 4.1: Pantalla de autorización de scopes en Spotify.

Una vez que la aplicación obtiene la autorización, el siguiente paso es conseguir el `access_token`, necesario para acceder a los recursos autorizados. Un concepto clave para entender cómo se realiza este proceso es el **authorization flow**, que define las interacciones entre la aplicación, el usuario y Spotify.

OAuth Flows: Elegir el Camino Adecuado

En el marco de OAuth 2.0, el término **flow** (flujo) se refiere a los diferentes procesos diseñados para obtener un `access_token`, dependiendo de las necesidades y características de la aplicación. Estos flujos existen para cubrir una variedad de escenarios, desde aplicaciones web con servidores backend hasta aplicaciones móviles o servicios que no requieren acceso a datos del usuario.

OAuth 2.0 define seis tipos principales de flows, sin embargo, Spotify implementa solo cuatro de ellos (tabla 4.1):

OAuth 2.0	Spotify
Authorization Code	✓
PKCE	✓
Client Credentials	✓
Device Code	✗
Implicit Flow [legacy]	✓
Password Grant [legacy]	✗

Tabla 4.1: Authorization flows definidos por OAuth 2.0 y cuáles implementa Spotify.

Cada uno de estos flujos está diseñado para un escenario específico de uso, por lo que, para poder tomar una buena elección, analizaremos las situaciones en las que cada uno resulta más adecuado:

- **Authorization Code Flow:** Ideal para aplicaciones web que cuentan con un backend seguro donde almacenar el `client_secret`. Proporciona tanto un `access_token` como un `refresh_token`, lo que permite mantener el acceso sin requerir que el usuario se autentique nuevamente. Es el flujo recomendado para aplicaciones con servidores backend que necesitan acceso a datos específicos del usuario.
- **Authorization Code Flow con PKCE:** Es una extensión del anterior, diseñado para escenarios donde no es seguro almacenar el `client_secret`, como en aplicaciones móviles o *Single Page Applications* (SPA). Añade una capa de seguridad utilizando un `code_verifier` y un `code_challenge` para evitar que el código de autorización sea interceptado y utilizado de manera malintencionada.
- **Client Credentials Flow:** Adecuado para aplicaciones backend o servicios que no requieren acceso a datos específicos del usuario, sino que interactúan con recursos de Spotify de manera general. No incluye un proceso de autorización por parte del usuario y no proporciona datos personales.
- **Implicit Grant Flow:** En desuso debido a limitaciones de seguridad. Fue diseñado para aplicaciones cliente que no tienen un backend, pero carece de soporte para `refresh_tokens` y expone el `access_token` en la URL, lo que lo hace menos seguro.

4.3. Principales Endpoints Relevantes para el Proyecto

De los cuatro flujos de autorización implementados por Spotify, este proyecto utilizará el **Authorization Code Flow** (sin PKCE). Esta elección se debe a que la aplicación cuenta con un backend seguro implementado con *Route Handlers* de Next.js, lo que permite almacenar de forma segura el `client_secret`. Además, este flujo proporciona tanto un `access_token` como un `refresh_token`, lo que asegura un acceso continuo a los datos del usuario sin necesidad de repetir el proceso de autenticación. Dado que es el flujo recomendado por Spotify para aplicaciones web que necesitan acceder a datos específicos del usuario, garantiza un balance óptimo entre seguridad, funcionalidad y cumplimiento de estándares.

4.3. Principales Endpoints Relevantes para el Proyecto

Para facilitar la comprensión de los endpoints utilizados en este proyecto, se ha diseñado un sistema visual que resume la información clave de cada uno, evitando la sobrecarga de texto y permitiendo al lector identificar rápidamente los elementos esenciales.

Cada endpoint se presenta como una interacción entre la solicitud (*request*) y la respuesta (*response*). A la izquierda, la *request* incluye el método HTTP (GET, POST, PUT, DELETE), la URL de la petición y los parámetros requeridos, que pueden estar en la URL (para métodos GET) o en el cuerpo de la solicitud (*body*, para métodos como POST o PUT). Si existen parámetros en los *headers*, se mostrarán sobre los parámetros del *body*/URL. A la derecha se encuentra la *response*, con los campos principales que estarán presentes en el JSON devuelto por la API de Spotify, si la solicitud se procesa correctamente.

En la figura 4.2 se muestra una plantilla genérica que sirve como referencia para entender este sistema visual, que se llenará con la información específica de cada endpoint en las siguientes secciones.

Nombre del Endpoint

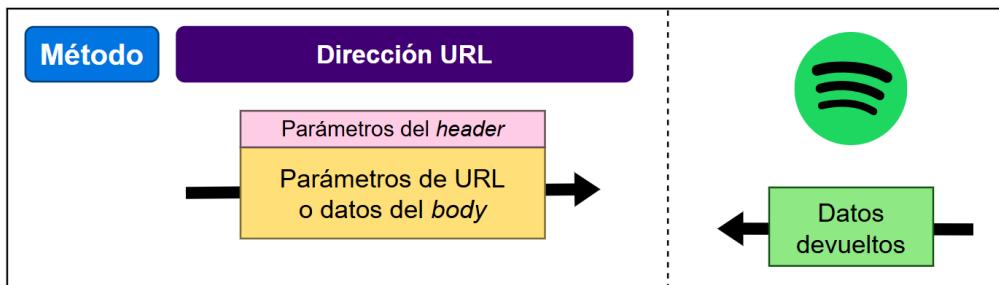


Figura 4.2: Plantilla visual para representar los endpoints.

4.3.1. Endpoints de Autenticación

La interacción con Spotify comienza con un endpoint dedicado al proceso de autorización: **Request User Authorization** (figura 4.3). Este endpoint genera la pantalla de autorización que se le muestra al usuario y, en el caso de que acepte, se le redirige a la `redirect_uri` indicada en la *request*. En esta URI siempre se suele implementar la llamada al segundo endpoint llamado **Request Access Token** (figura 4.4), necesario para finalizar el proceso de autorización. Este permite intercambiar el code obtenido en el paso anterior

4.3. Principales Endpoints Relevantes para el Proyecto

por un `access_token`, el cual es requerido para realizar cualquier otra solicitud posterior a la API.

Request User Authorization

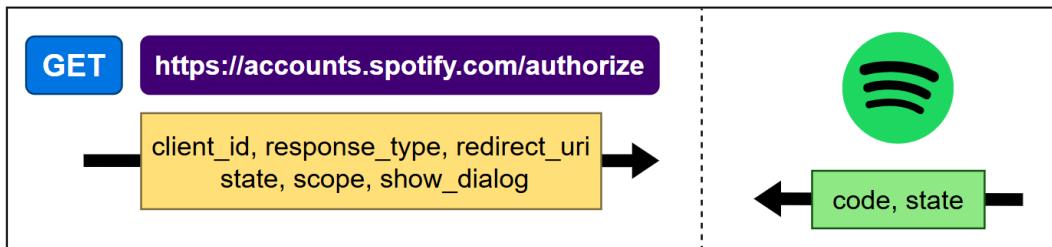


Figura 4.3: Endpoint de `Request User Authorization`.

■ Parámetros del Request

- `client_id`: El ID de cliente generado al registrar la aplicación.
- `response_type`: Se establece en “code”, indicando que se solicita un código de autorización.
- `redirect_uri`: URI a la que se redirige al usuario después de aceptar o rechazar los permisos. Debe coincidir exactamente con uno de los valores configurados al registrar la aplicación.
- `state`: Parámetro utilizado para proteger contra ataques como *cross-site request forgery* (CSRF). Su valor debe ser validado al recibir la respuesta.
- `scope`: Lista de scopes separados por espacios, indicando los permisos requeridos por la aplicación. Si no se especifican, solo se concederá acceso a información pública.
- `show_dialog`: Determina si se fuerza al usuario a aprobar nuevamente la aplicación, incluso si ya lo hizo previamente. Si se establece en `true`, el usuario verá el diálogo de autorización; de lo contrario, será redirigido automáticamente.

■ Campos del Response

- `code`: Código de autorización que puede intercambiarse posteriormente por un `access_token`.
- `state`: El valor del parámetro `state` enviado originalmente en la solicitud. Su valor debe ser comparado para garantizar la validez de la respuesta.

Request Access Token

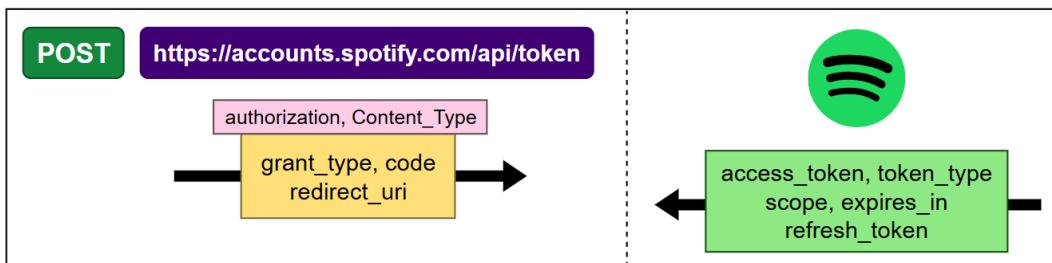


Figura 4.4: Endpoint de *Request Access Token*.

■ Parámetros del Request

- **Body**
 - **grant_type**: Este campo debe contener el valor “`authorization_code`”.
 - **code**: El código de autorización devuelto de la solicitud previa.
 - **redirect_uri**: Este parámetro se utiliza únicamente para validación (no se realiza una redirección real). El valor debe coincidir exactamente con el valor de `redirect_uri` utilizado al solicitar el código de autorización.
- **Headers**
 - **Authorization**: Cadena codificada en Base64 con el siguiente formato: `Basic <base64 encoded client_id:client_secret>`.
 - **Content-Type**: Establecido en “`application/x-www-form-urlencoded`”.

■ Campos del Response

- **access_token**: Token de acceso que permite hacer las posteriores llamadas a la API.
- **token_type**: Indica cómo se puede usar el token de acceso; siempre tiene el valor “`Bearer`”.
- **scope**: Lista de scopes separados por espacios que han sido concedidos para este `access_token`.
- **expires_in**: Período de tiempo (en segundos) durante el cual el token de acceso es válido. Siempre es de 1 hora.
- **refresh_token**: Token utilizado para renovar el `access_token` cuando este expira.

4.3.2. Endpoints de Datos

Una vez completado el proceso de autorización y obtenido el `access_token`, la aplicación puede interactuar con los endpoints de datos proporcionados por Spotify. Hay un total de 88 endpoints agrupados en 14 grupos. En la figura 4.5 se indican los grupos cuyos endpoints se van a utilizar en este proyecto.

4.3. Principales Endpoints Relevantes para el Proyecto

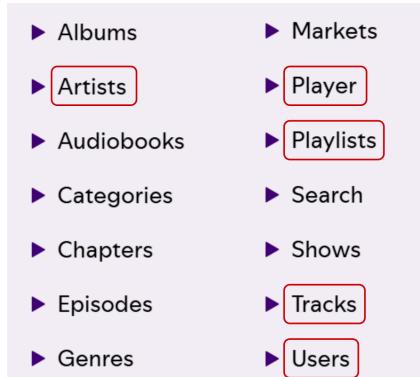


Figura 4.5: Grupos de endpoints ofrecidos y cuáles se van a usar.

Por razones prácticas, solo se van a mencionar los campos más relevantes en los *request* y *response* de los endpoints, ya que los objetos devueltos por la API suelen ser extensos y contener una gran cantidad de información, además de redundante en algunos casos. También se omite el campo de *Authorization*, que contiene el *access_token* en el *header* del *request*, ya que es necesario incluirlo en todas las peticiones.

Users

En el grupo **Users** se encuentran los endpoints relacionados con la información de la cuenta del usuario. Usaremos el endpoint de **Get Current User's Profile** (figura 4.6) para obtener datos como el nombre, correo electrónico y la imagen de perfil de la cuenta. Por otro lado, el endpoint de **Get User's Top Items** permite obtener los “elementos” más escuchados por el usuario, que en este caso podemos elegir entre *tracks* (figura 4.7) o *artists* (figura 4.8).

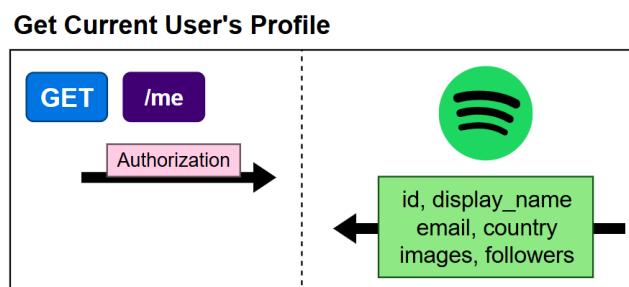


Figura 4.6: Endpoint de *Get Current User's Profile*.

■ Parámetros del Request

- No requiere parámetros adicionales.

■ Campos del Response

- **country**: El país del usuario, en formato ISO.
- **display_name**: Nombre mostrado en el perfil del usuario.
- **email**: Dirección de correo del usuario, ingresada al crear la cuenta.

- **id:** ID del usuario en Spotify.
- **images:** Array conteniendo las URLs de la imagen del perfil del usuarios en distintos tamaños.
- **followers:** Objeto con la información sobre los seguidores del usuario.

Get User's Top Items (Tracks)

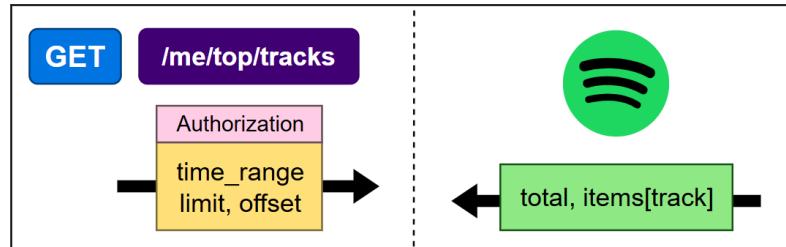


Figura 4.7: Endpoint de *Get User's Top Items (Tracks)*.

■ Parámetros del Request

- **Body**
 - `time_range`: El marco temporal para calcular las afinidades ("long_term", "medium_term" (por defecto), "short_term").
 - `limit`: Máximo número de elementos a devolver. Rango: 1-50.
 - `offset`: Índice del primer elemento a devolver.

■ Campos del Response

- `next`: URL de la página siguiente de elementos. `null` si no hay más elementos.
- `total`: Número total de elementos disponibles.
- `items`: Array de objetos de los datos de cada track.
 - `name`: Nombre del track.
 - `popularity`: Su popularidad (0-100).
 - `duration_ms`: Su duración en milisegundos.
 - `explicit`: Valor booleano indicando si el track contiene letras explícitas.
 - `album`: Información sobre el álbum donde aparece.
 - ◊ `id`: ID del álbum en Spotify.
 - ◊ `images`: Array conteniendo las URLs de la imagen de la portada del álbum en distintos tamaños.
 - ◊ `name`: Nombre del álbum.
 - ◊ `release_date`: Fecha de lanzamiento del álbum.
 - `artists`: Array con los objetos relacionados con los artistas que participaron en el track.
 - ◊ `name`: Nombre del artista.
 - ◊ `id`: ID del artista en Spotify.

Get User's Top Items (Artists)

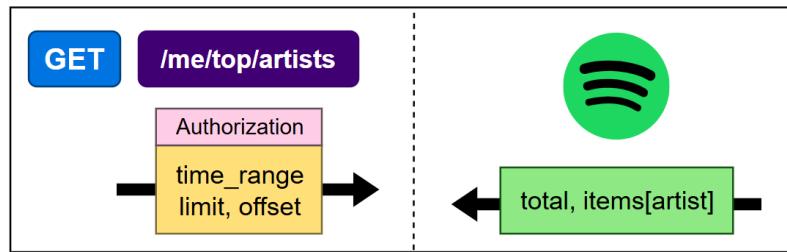


Figura 4.8: Endpoint de *Get User's Top Items (Artists)*.

■ Parámetros del Request

- Los mismos que en el endpoint de *Get User's Top Items (Tracks)*.

■ Campos del Response

- id: ID del artista en Spotify.
- name: Nombre del artista.
- popularity: Su popularidad (0-100).
- followers: Objeto con la información sobre los seguidores del artista.
- genres: Lista de géneros asociados al artista. Si el artista aún no está clasificado, el array estará vacío.
- images: Array conteniendo las URLs de la imagen del artista en distintos tamaños.

Player

En el grupo **Player** podemos encontrar endpoints para controlar y obtener el estado de reproducción de la cuenta. Además de poder iniciar, pausar, adelantar o controlar el volumen de la reproducción, también podemos saber las últimas canciones escuchadas por el usuario mediante el endpoint de **Get Recently Played Tracks** (figura 4.9).

Cabe destacar que este endpoint tiene una limitación muy considerable: **solo permite obtener las últimas 50 canciones escuchadas por el usuario**. Es decir, no es posible obtener un historial de escucha en base a un periodo de tiempo establecido, ya que esas 50 canciones pueden haber sido escuchadas en un periodo largo o en un solo día, según los hábitos de escucha del usuario. Esta limitación ha afectado en el diseño de algunas estadísticas de la aplicación, en concreto aquellas que requieren conocer los gustos musicales del usuario a lo largo del tiempo.

Get Recently PlayedTracks

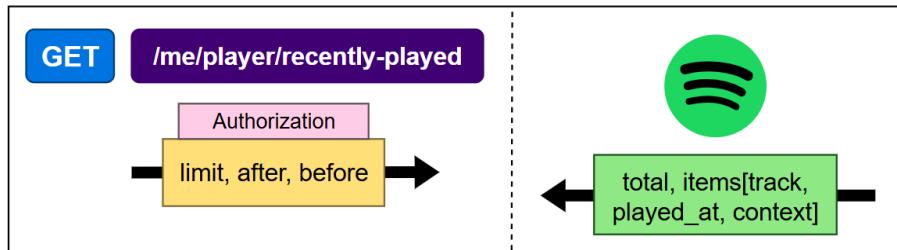


Figura 4.9: Endpoint de *Get Recently Played Tracks*.

■ Parámetros del Request

- **Body**

- **limit**: Número máximo de elementos a devolver. Rango: 1-50.
- **after**: Marca de tiempo Unix en milisegundos. Devuelve todos los elementos posteriores (excluyendo el indicado). Si se especifica **after**, no se debe especificar **before**.
- **before**: Marca de tiempo Unix en milisegundos. Devuelve todos los elementos anteriores (excluyendo el indicado). Si se especifica **before**, no se debe especificar **after**.

■ Campos del Response

- **next**: URL a la página siguiente de elementos. **null** si no hay más elementos.
- **total**: Número total de elementos disponibles.
- **items**: Objetos conteniendo la información sobre los tracks del historial de reproducción.
 - **name**: Nombre del track.
 - **duration_ms**: Duración en milisegundos.
 - **played_at**: Fecha y hora en la que se reprodujo el track.
 - **explicit**: Indica si el track contiene contenido explícito.
 - **popularity**: Popularidad del track (0-100).
 - **album**: Información sobre el álbum en el que se encuentra.
 - ◊ **name**: Nombre del álbum.
 - ◊ **release_date**: Fecha de lanzamiento.
 - ◊ **images**: Las URLs de la portada del álbum en varios tamaños.
 - **artists**: Información sobre los artistas que participaron.
 - ◊ **name**: Nombre del artista.
 - ◊ **id**: ID del artista en Spotify.

Tracks

Mediante los endpoints del grupo **Tracks**, se pueden obtener datos sobre canciones específicas. El endpoint **Get User's Saved Tracks** (figura 4.10) devuelve las canciones guardadas en la lista de favoritos del usuario.

Este endpoint, además de permitir identificar las canciones favoritas del usuario, **proporciona una alternativa para analizar sus gustos musicales a lo largo del tiempo**. Aunque no se trata de un historial de escucha, ofrece una lista de canciones que el usuario ha decidido guardar junto con la fecha correspondiente, lo que puede servir como indicativo de sus preferencias. Esta alternativa ha sido la solución adoptada para suplir la limitación mencionada en el endpoint de **Get Recently Played Tracks**. Gracias al valor proporcionado en el campo `added_at`, es posible considerar que el acto de añadir una canción a favoritos funciona como un proxy para representar los gustos musicales y las canciones escuchadas en torno a ese periodo de tiempo.

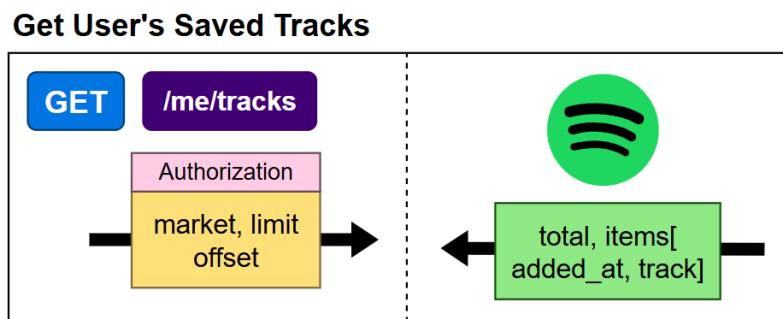


Figura 4.10: Endpoint de *Get User's Saved Tracks*.

■ Parámetros del Request

- **Body**
 - `market`: Código de país para indicar el mercado, en formato ISO.
 - `limit`: Número máximo de elementos a devolver. Rango: 1-50.
 - `offset`: Índice del primer elemento a devolver.

■ Campos del Response

- `next`: URL a la página siguiente de elementos. `null` si no hay más elementos.
- `total`: Número total de elementos disponibles.
- `items`: Array de objetos con la información de los tracks guardados.
 - `added_at`: Fecha y hora en la que se guardó el track en formato ISO.
 - `track`: Objeto con la información sobre el track.
 - ◊ Contiene los mismos campos que en el endpoint de *Get Recently Played Tracks*.

Artists

Al igual que el grupo anterior, en **Artists** se pueden consultar datos sobre artistas específicos. **Get Several Artists** (figura 4.11) permite obtener información sobre varios artistas a la vez, reduciendo el número de peticiones realizadas a la API.

Get Several Artists

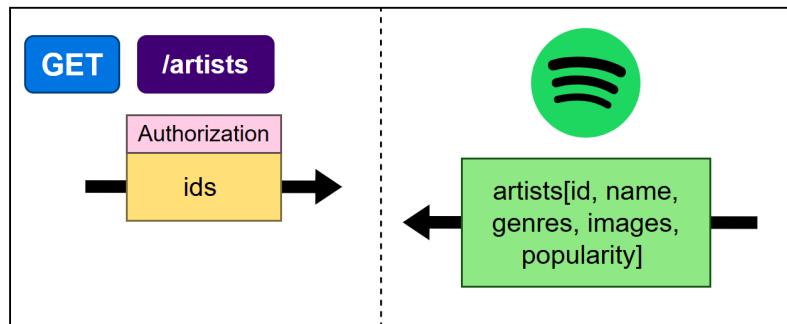


Figura 4.11: Endpoint de *Get Several Artists*.

▪ Parámetros del Request

- URL params
 - ids: Lista separada por comas de los IDs de Spotify de los artistas.

▪ Campos del Response

- artists: Lista de objetos con la información de los artistas.
 - name: Nombre del artista.
 - id: Su ID en Spotify.
 - popularity: Su popularidad (0-100).
 - followers: Información sobre sus seguidores.
 - genres: Lista de géneros asociados al artista.
 - images: Array de URLs de la imagen del artista en varios tamaños.

Playlists

En **Playlists** se encuentran los endpoints que permiten trabajar con las listas de canciones del usuario. Con **Create Playlist** (figura 4.12) podemos crear una nueva playlist, **Add Items To Playlist** (figura 4.13) para añadir canciones y con **Add Custom Playlist Cover Image** (figura 4.14) podemos cambiar la portada de la playlist por una imagen personalizada. Suele ser muy común trabajar con varios endpoints de este grupo en conjunto.

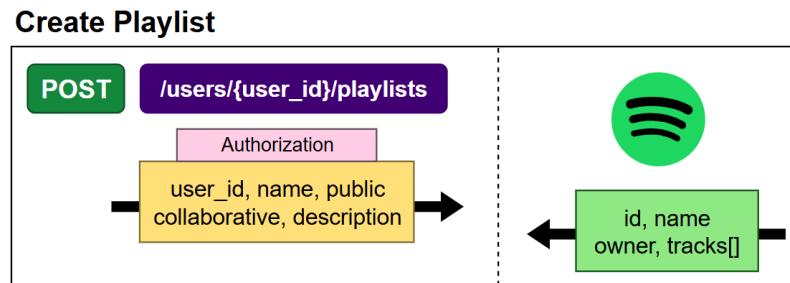


Figura 4.12: Endpoint de *Create Playlist*.

■ Parámetros del Request

- **URL params**
 - `user_id`: ID del usuario de Spotify que creará la playlist.
- **Body**
 - `name`: Nombre de la nueva playlist.
 - `public`: Si la playlist será pública (`true`) o privada (`false`).
 - `collaborative`: Si la playlist será colaborativa o no.
 - `description`: Descripción de la playlist.

■ Campos del Response

- `id`: ID de la playlist.
- `name`: Nombre de la playlist.
- `href`: Enlace al endpoint de la playlist en la API de Spotify.
- `owner`: Objeto con información del usuario propietario de la playlist.
 - `id`: ID del usuario.
 - `display_name`: Nombre del usuario.
- `snapshot_id`: Identificador de la versión actual de la playlist.
- `tracks`: Objeto con información sobre las pistas en la playlist.

Add items to Playlist

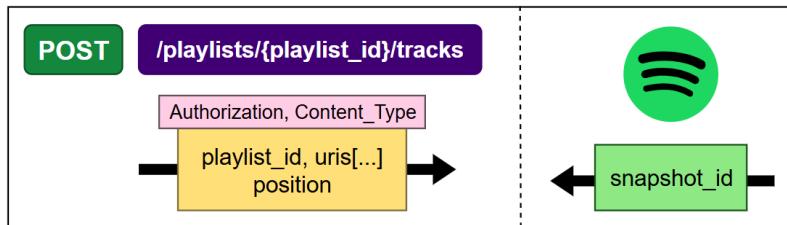


Figura 4.13: Endpoint de *Add Items to Playlist*.

■ Parámetros del Request

- URL params

- playlist_id: ID de la playlist a la que se añadirán los elementos.

- Body

- uris: Array de URIs de Spotify de tracks o episodios a añadir.
- position: Índice donde se insertarán los elementos.

■ Campos del Response

- snapshot_id: Identificador de la nueva versión de la playlist tras la modificación.

Add Custom Playlist Cover Image

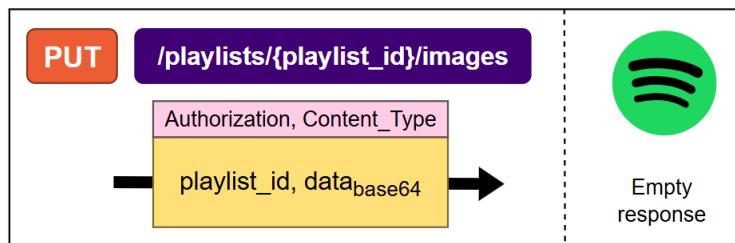


Figura 4.14: Endpoint de *Add Custom Playlist Cover Image*.

■ Parámetros del Request

- URL params

- playlist_id: ID de la playlist a la que se subirá la imagen personalizada.

- Body

- Imagen en formato JPEG codificada en Base64 (máx 256 KB) .

■ Campos del Response

- No devuelve datos adicionales.

Los datos proporcionados por estos endpoints serán utilizados para poblar de información la aplicación, procesándolos y adaptándolos cuando sea necesario para ofrecer una experiencia personalizada al usuario. Además, estos datos influirán directamente en el diseño de la aplicación, tanto en los componentes del frontend, como en la estructura del servidor y la comunicación entre ambos.

4.4. Scopes Necesarios

Como ya se ha mencionado en el apartado 4.2, para poder tratar los datos necesarios, el usuario deberá autorizar el acceso a ciertos recursos de su cuenta. Para ello, es necesario indicar los scopes adecuados al solicitar la autorización. En este proyecto, se necesitarán los siguientes scopes:

- user-top-read
- user-read-private
- user-read-email
- user-read-recently-played
- user-library-read
- playlist-modify-private
- ugc-image-upload

Cada uno de ellos añadirá una descripción asociada en la pantalla de autorización que se muestra al usuario durante el inicio de sesión.

4.5. Limitaciones y Consideraciones

Además del límite de 50 canciones en el endpoint de **Get Recently Played Tracks** y otras limitaciones relacionadas con el acceso a datos concretos, la principal limitación impuesta por la API de *Spotify* es la **tasa de peticiones** o *rate limit*. Esta limitación se establece para evitar la sobrecarga de los servidores y garantizar un funcionamiento estable del servicio. La tasa de peticiones de Spotify se calcula en una ventana de 30 segundos (figura 4.15). Si la aplicación supera este límite en dicho periodo, recibirá una respuesta de error 429 Too Many Requests y los recursos solicitados quedarán temporalmente inaccesibles.

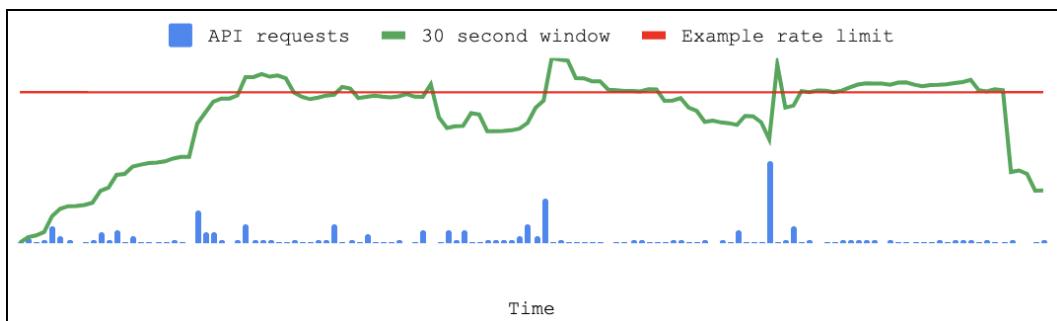


Figura 4.15: Gráfica del funcionamiento de la tasa de peticiones, obtenida de la documentación oficial de Spotify.

Para evitar alcanzar este límite, se pueden implementar técnicas para optimizar el número de peticiones. Algunas de las que se implementarán en este proyecto son:

- Batch APIs: Utilizar endpoints que permiten obtener lotes de datos en una sola petición, como el endpoint **Get Several Artists**.
- Lazy Loading: Retrasar las solicitudes de datos hasta que el usuario interactúe con un elemento específico, como al abrir una estadística.

4.6. Política y Términos de Uso de la API

El uso de la API de *Spotify* está regulado por un marco legal detallado, compuesto por los *Spotify Developer Terms* y la *Spotify Developer Policy*. Estas normativas establecen las condiciones y restricciones bajo las cuales se puede acceder, procesar y utilizar los datos proporcionados por la API. A continuación, se analizan los aspectos que afectan el desarrollo del proyecto en el marco del TFG.

4.6.1. Restricciones y Obligaciones

Se subrayan dos principios básicos en sus términos: **la protección de los derechos de los usuarios y la garantía de que el contenido ofrecido en su plataforma está debidamente licenciado**. Esto implica que cualquier aplicación desarrollada debe respetar la privacidad de los usuarios y manejar los datos conforme a las preferencias que ellos definan. Además, está prohibido utilizar los datos o contenidos de *Spotify* para fines no autorizados, como entrenar modelos de inteligencia artificial o crear bases de datos derivadas.

El uso de la API está sujeto a restricciones específicas, que se verán de cumplir en este proyecto. Entre las más relevantes se destacan:

- **Acceso y uso de datos:** Solo se pueden solicitar los datos necesarios para operar la aplicación y deben eliminarse si un usuario decide desconectar su cuenta de *Spotify*.
- **Prohibiciones explícitas:** Está prohibido transferir datos a terceros, crear funcionalidades que permitan *stream ripping*, o desarrollar servicios destinados al uso empresarial o con contenido dirigido a menores de edad.
- **Caché local:** Aunque se permite el almacenamiento temporal de ciertos datos para mejorar el rendimiento, no se debe almacenar contenido indefinidamente.
- **Mecanismos de desconexión:** La aplicación debe ofrecer a los usuarios una forma accesible y funcional de desconectar su cuenta de *Spotify* en cualquier momento.

4.6.2. Protección de Datos y Seguridad

Los desarrolladores son responsables de implementar medidas de seguridad estándar para proteger los datos personales de los usuarios. Esto incluye:

- Proporcionar una política de privacidad clara y accesible que explique cómo se recopilan, procesan y comparten los datos.
- Asegurar la confidencialidad de los datos personales y notificarlos a *Spotify* en caso de incidentes de seguridad que puedan comprometer esta información.
- Respetar las solicitudes de los usuarios relacionadas con sus datos, como la rectificación o eliminación de la información.

4.6.3. Implicaciones para el TFG

La implementación de la aplicación debe garantizar el cumplimiento de las políticas mencionadas, especialmente la correcta gestión de los datos personales de los usuarios y el uso correcto de los recursos. En este sentido, es importante solicitar únicamente los *scopes* necesarios para las funcionalidades definidas, asegurando que cada autorización concedida por los usuarios esté justificada. También queda claro que el único uso válido del almacenamiento de los datos es el de la caché temporal en el servidor, con el propósito de optimizar el rendimiento. Además, en caso de que el usuario decida desconectar su cuenta de la aplicación, se debe implementar procesos claros y que aseguren la eliminación completa de sus datos.

Spotify se reserva el derecho de revisar las aplicaciones que acceden a su API para garantizar que cumplan con los términos establecidos. En caso de detectar alguna infracción, como el incumplimiento de la normativa descrita, pueden limitar el acceso, revocar permisos o incluso suspender completamente la funcionalidad de la aplicación. Por lo tanto, el diseño y desarrollo de la aplicación deben incorporar estas consideraciones desde el principio, para evitar cualquier tipo de conflicto con la política de *Spotify*.

Análisis

5.1. Requisitos Funcionales

Los requisitos funcionales describen las funcionalidades esenciales que debe cumplir la aplicación para satisfacer las necesidades del usuario. En el caso de esta aplicación, todos los requisitos funcionales están orientados al cumplimiento del objetivo principal: permitir a los usuarios acceder al análisis de sus datos musicales cuando lo deseen. Estas funcionalidades están organizadas en diferentes categorías según su ámbito.

5.1.1. Autenticación y Sesión

- El usuario debe poder iniciar sesión utilizando su cuenta de *Spotify* mediante OAuth 2.0.
- El usuario debe poder cerrar sesión desde cualquier página.
- El cierre de sesión debe eliminar cualquier dato asociado al usuario.
- La aplicación debe obtener y gestionar adecuadamente e token de acceso para interactuar con la API de *Spotify*.
- La sesión del usuario debe permanecer activa mientras este interactúe con la aplicación web. En caso de que el tiempo de validez del token expire, el sistema debe renovar el token de forma transparente para el usuario.

5.1.2. Navegación

- La aplicación debe tener una barra de navegación con acceso a las secciones:
 - **Home:** Vista general de las estadísticas generales.
 - **Stats:** Estadísticas más avanzadas y originales.
 - **Panel de Usuario:** Panel desplegable con la opción de cerrar sesión.

5.1.3. Estadísticas Generales (Home)

- El usuario debe ver una página inicial con estadísticas generales relacionadas con su cuenta, siendo las siguientes:
 - **Top Tracks:** Muestra los top 5 canciones del usuario.
 - **Top Artists:** Muestra los top 5 artistas del usuario.
 - **Top Genres:** Muestra los top 5 géneros del usuario,
 - **Recently Played:** Muestra una lista reducida de las últimas 10 canciones escuchadas, en orden cronológico. Si el usuario quiere, podrá ver la lista completa de las 50 canciones pulsando un botón.
- El usuario podrá cambiar el periodo de tiempo en el que se basan los datos de los tres tops mediante un menú desplegable.

5.1.4. Estadísticas Avanzadas (Stats)

En esta página, el usuario podrá explorar una gama de seis estadísticas más avanzadas. Cada estadística ofrece funcionalidades específicas, descritas a continuación:

Hall Of Fame

- La estadística mostrará las **top 16 canciones** del usuario en formato de cuadrícula (4x4), utilizando las portadas de los álbumes de cada canción como elemento visual.
- Al pasar el ratón por encima de cada una de las portadas, se mostrará el nombre de la canción y el artista.
- El usuario, mediante un botón, podrá crear una playlist de manera automática en su cuenta:
 - Se generará una nueva playlist.
 - Se añadirán las canciones del top 16.
 - Se actualizará la imagen de la playlist con la representación visual de la cuadrícula de portadas generada para esta estadística.

Huella Del Día

- El usuario verá un gráfico de líneas, donde el eje horizontal representa las horas del día y el eje vertical los minutos de música escuchados (correspondientes a 1 hora).
- Al pasar el ratón por encima de un nodo de la gráfica, se mostrará la cantidad exacta de minutos escuchados en esa hora.
- Se indica la hora del día con más minutos de escucha.

Estaciones Musicales

- Se mostrará un gráfico en forma de anillo dividido en cuatro segmentos, cada uno representando una estación del año: invierno, primavera, verano y otoño.
- Al pasar el ratón por encima de un segmento del gráfico, se abrirá un panel que mostrará, basado en la actividad del usuario, el artista y el género musical destacados durante ese periodo.
- La información en el panel se actualiza dinámicamente mientras el usuario pasa el ratón por las distintas secciones del gráfico.

Tus Décadas

- Se presentará un histograma que muestra el número de álbumes correspondientes a cada año, agrupados por décadas. La cantidad de álbumes se muestra de manera visual, mediante imágenes de portadas apiladas en columnas. Los álbumes están asociados a las canciones favoritas del usuario, mostrando una única repetición de cada álbum, independientemente de la cantidad de canciones guardadas de ese álbum.
- El gráfico es explorable, permitiendo al usuario desplazarse horizontal y verticalmente.
- El usuario podrá hacer *zoom in* y *zoom out* para ajustar el nivel de detalle, permitiéndole observar las portadas de los álbumes con mayor claridad.

La Bitácora

El usuario podrá explorar detalladamente el historial completo de sus canciones guardadas en favoritos, organizadas por fechas. Se estructura de la siguiente manera:

- Se mostrará la visualización principal; un gráfico de barras con el número total de canciones guardadas, agrupadas por años, desde la creación de la cuenta hasta la fecha actual.
- Al hacer clic en una barra que represente un año, se profundizará en el gráfico para mostrar la distribución de canciones guardadas por meses dentro de ese año.
- De forma similar, al hacer clic en una barra que represente un mes, se desglosará la información para mostrar las canciones guardadas por días dentro de ese mes específico.
- El usuario podrá navegar libremente entre los niveles (años, meses y días), pudiendo retroceder hacia niveles superiores en cualquier momento.
- Al pasar el ratón por encima de cualquier barra, se mostrará el número de canciones guardadas para ese período.
- En el nivel de días, al pasar el ratón sobre una barra, se mostrarán además los nombres y artistas de las canciones guardadas en esa fecha específica.

Índice de Resonancia

Se presentará al usuario, de forma visual, dos métricas relacionadas con sus preferencias musicales: la popularidad media de sus canciones guardadas en favoritos y la popularidad media de las últimas 50 canciones escuchadas. Estas métricas se implementarán a través de las siguientes funcionalidades:

- Cada valor estará asociado a una onda sinusoidal animada, cuya frecuencia se ajustará proporcionalmente al valor numérico correspondiente.
- El usuario podrá pulsar un botón que generará una nueva onda. Esta nueva onda se calcula mediante la suma matemática de las frecuencias de las dos ondas originales, mostrando la interferencia entre las dos. Además, junto a esta nueva onda, se mostrará la diferencia numérica entre los dos valores originales.
- El usuario podrá pasar el ratón sobre la tarjeta de cualquiera de los valores para resaltar visualmente la onda correspondiente, facilitando la identificación.
- Los cálculos y la generación de datos necesarios para esta estadística se realizarán principalmente en el servidor.

5.2. Requisitos No Funcionales

Los requisitos no funcionales de la aplicación son fundamentales para garantizar no solo su funcionalidad, sino también su rendimiento, seguridad, escalabilidad y usabilidad. A continuación, se detallan los aspectos clave que la aplicación debe cumplir para asegurar una experiencia de usuario satisfactoria y el cumplimiento de los estándares de calidad.

Rendimiento y Escalabilidad

La aplicación debe ofrecer tiempos de respuesta aceptables en las operaciones críticas, como la carga inicial del *Home* o la presentación de las estadísticas. El tiempo de respuesta debe ser inferior a **1 s** bajo condiciones normales de uso, con un máximo de **3 s** bajo picos de carga. Este requisito se medirá utilizando herramientas de pruebas de carga como K6, simulando hasta **50 usuarios simultáneos** realizando operaciones comunes. El objetivo es garantizar que el rendimiento del sistema satisfactorio para los usuarios, incluso en picos de carga inusuales.

Seguridad

Es esencial garantizar la protección de los datos del usuario y las comunicaciones entre el frontend y el backend. Para ello, todas las conexiones deben utilizar **HTTPS**, y los tokens de acceso deben almacenarse de forma segura para prevenir ataques comunes como *Cross-Site Scripting* (XSS) y *Man-in-the-Middle* (MITM). Además, es necesario implementar medidas de seguridad en procesos críticos, como la autenticación de usuarios, incluyendo protección frente a ataques de tipo *Cross-Site Request Forgery* (CSRF).

Testabilidad

Para garantizar la calidad del sistema, se deben implementar pruebas automatizadas que cubran las funcionalidades clave de la aplicación, priorizando las partes críticas del backend y las interacciones más relevantes del frontend. Esto incluirá pruebas unitarias y de integración. La cobertura mínima objetivo será del **60 % del código**, con un enfoque en las secciones esenciales. Las pruebas se ejecutarán regularmente mediante integración continua (CI) utilizando GitHub Actions.

Usabilidad

La aplicación debe ser intuitiva y fácil de usar para el público general. Todas las estadísticas y funcionalidades deben presentarse de manera clara. Este requisito se evaluará mediante pruebas de usabilidad con al menos **5 usuarios representativos**, asegurando que puedan completar tareas concretas sin dificultad ni confusión.

5.3. Casos de Uso

En esta sección se describen los principales casos de uso de la aplicación, relacionados con los requisitos funcionales mencionados. Cada caso de uso se enfoca en un objetivo específico que un usuario puede alcanzar utilizando las funcionalidades del sistema.

5.3.1. Actores

En este sistema, interactúan tres actores diferentes con la aplicación: el usuario anónimo, el usuario autenticado y la API de *Spotify*. A continuación, se describen los roles de cada uno:

- **Usuario Anónimo:** Este actor representa a los usuarios que no han iniciado sesión en la aplicación. Su única interacción con el sistema es el de inicio de sesión.
- **Usuario Autenticado:** Este actor es el usuario que ha iniciado sesión correctamente en la aplicación. Tiene acceso a todas las funcionalidades. Es considerado como el actor principal.
- **Web API de Spotify:** Este actor es el servicio de *Spotify* que proporciona un sistema de autenticación y los datos necesarios para generar las estadísticas. Interactúa principalmente con el backend de la aplicación.

5.3.2. Modelo de Casos de Uso

En el modelo presentado en la figura 5.1, se destacan las funcionalidades principales de la aplicación y su relación con los actores: **Usuario Anónimo** y **Usuario Autenticado**, que interactúan directamente con el sistema, y la **API de Spotify**, que actúa como proveedor de datos.

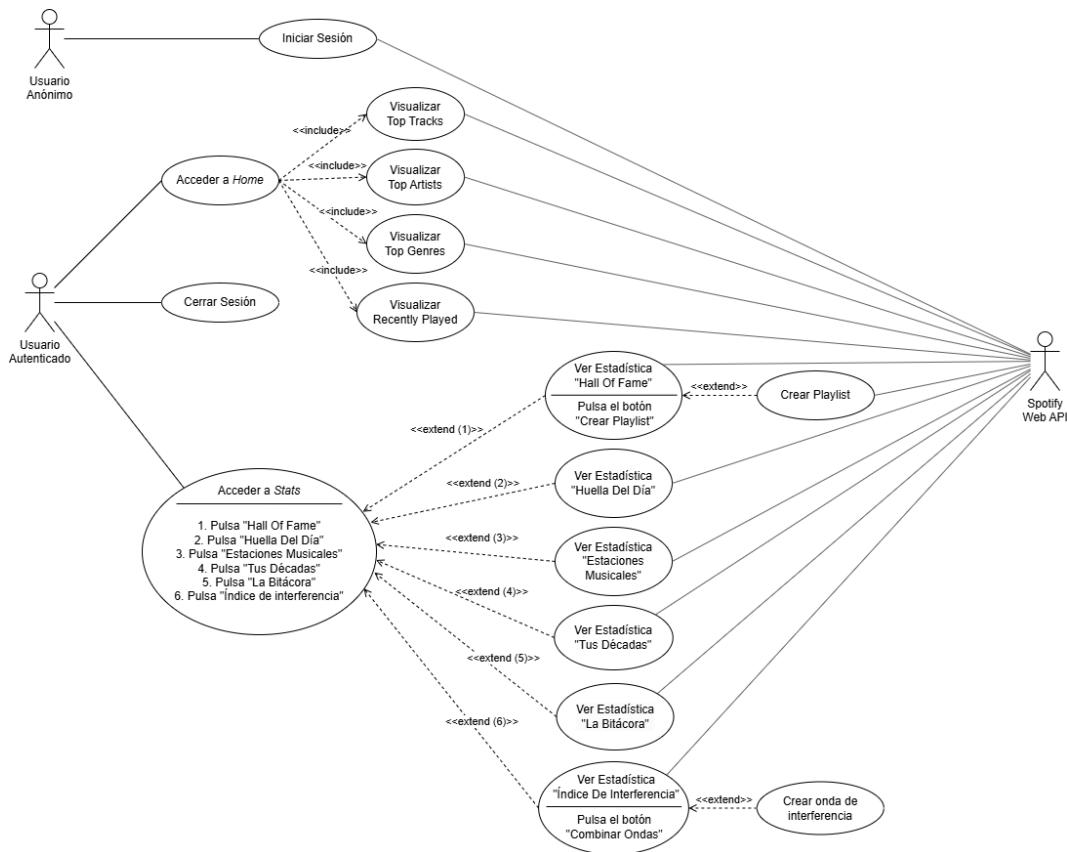


Figura 5.1: Modelo de casos de uso del sistema.

Iniciar Sesión

Permite al usuario anónimo iniciar sesión en la aplicación utilizando sus credenciales de *Spotify* a través del sistema de autenticación OAuth 2.0.

- Flujo principal (Inicio de Sesión):

1. El usuario anónimo selecciona la opción de “Iniciar Sesión” en la pantalla principal.
2. El sistema redirige al usuario a la página de inicio de sesión de *Spotify*.
3. El usuario introduce sus credenciales de *Spotify* en el formulario proporcionado por *Spotify*.
4. *Spotify* verifica las credenciales y, si son correctas, muestra la pantalla de solicitud de autorización.
5. El usuario autoriza a la aplicación a acceder a sus datos de *Spotify* indicados.
6. *Spotify* genera un token de acceso y redirige al usuario de vuelta a la aplicación.
7. El sistema recibe el token de acceso y lo almacena de forma segura.
8. El sistema redirige al usuario autenticado a la pantalla principal de la aplicación (*Home*).

- Flujo alternativo (Error de Autenticación):
 1. El usuario introduce credenciales incorrectas en la página de inicio de sesión de *Spotify*.
 2. *Spotify* rechaza las credenciales y muestra un mensaje de error al usuario.
 3. El sistema redirige al usuario de vuelta a la pantalla principal, donde podrá volver a intentar iniciar sesión.
- Flujo alternativo (Denegación de Autorización):
 1. *Spotify* verifica las credenciales del usuario y muestra la pantalla de solicitud de autorización.
 2. El usuario decide no conceder la autorización y selecciona la opción de “Cancelar”.
 3. *Spotify* informa al sistema de que la autorización ha sido rechazada.
 4. El sistema redirige al usuario de vuelta a la pantalla principal.

Acceder a Home

Permite al usuario autenticado visualizar la pantalla principal de la aplicación, donde se muestran las estadísticas básicas de *Top Tracks*, *Top Artists*, *Top Genres* y *Recently Played*.

- Flujo principal (Visualización de Home):
 1. El usuario autenticado accede a la aplicación tras iniciar sesión correctamente.
 2. El sistema solicita los datos relevantes de la API de *Spotify* para generar las estadísticas.
 3. La API de *Spotify* responde con los datos necesarios para cada estadística.
 4. El sistema procesa y organiza los datos para presentarlos en el frontend.
 5. El sistema muestra al usuario las siguientes secciones en la pantalla principal:
 - *Top Tracks*: Canciones más escuchadas por el usuario.
 - *Top Artists*: Artistas más escuchados.
 - *Top Genres*: Géneros favoritos.
 - *Recently Played*: Lista de reproducción reciente.
 6. El usuario puede cambiar el periodo de tiempo de las estadísticas de los tres *Tops* seleccionando entre los últimos 30 días, 6 meses o 1 año.
 7. El usuario puede expandir la lista de *Recently Played* para ver hasta 50 canciones en lugar de las 10 iniciales, pulsando un botón de "Ver más".
 8. El usuario puede contraer nuevamente la lista de *Recently Played* para reducirla a 10 canciones, pulsando un botón de "Ver menos".
- Flujo alternativo (Error al obtener datos de la API):
 1. El sistema solicita datos a la API de *Spotify*, pero ocurre un error en la comunicación.
 2. El sistema muestra un mensaje de error al usuario, indicando que no se pudieron cargar algunas estadísticas.
 3. El usuario puede intentar recargar la página o esperar a que el sistema reintente la solicitud.

Cerrar Sesión

Permite al usuario autenticado cerrar su sesión en la aplicación, eliminando cualquier dato relacionado con su sesión activa.

- Flujo principal (Cerrar Sesión):
 1. El usuario autenticado selecciona la opción de "Cerrar Sesión" en la interfaz de la aplicación.
 2. El sistema elimina todas las cookies relacionadas con la sesión, incluido el *access_token* del usuario.
 3. El sistema borra cualquier dato del usuario almacenado temporalmente en caché.
 4. El sistema redirige al usuario a la página de inicio de sesión.

Acceder a Stats

Permite al usuario autenticado acceder a la sección de estadísticas avanzadas de la aplicación, donde puede interactuar con diferentes visualizaciones de sus datos musicales.

- Flujo principal (Acceso a Stats):

1. El usuario autenticado selecciona la opción de “Stats” en la barra de navegación.
2. El sistema muestra una pantalla con las opciones de estadísticas disponibles:
 - **Hall of Fame.**
 - **Huella del Día.**
 - **Estaciones Musicales.**
 - **Tus Décadas.**
 - **La Bitácora.**
 - **Índice de Interferencia.**
3. El usuario selecciona una estadística específica para interactuar con ella.
4. El sistema carga la visualización correspondiente.

Para todas las siguientes estadísticas avanzadas, se sigue un flujo alternativo cuando ocurre algún error en la carga. Para evitar repetir la misma descripción, a continuación se mencionará el flujo alternativo. En cada estadística, solo se describirá el flujo principal.

- Flujo alternativo (Error en la carga de datos de una estadística avanzada):

1. El sistema solicita a la API de *Spotify* los datos necesarios para generar la estadística.
2. La API responde con un error, ya sea por una conexión fallida, un token de acceso inválido o una respuesta incompleta.
3. El sistema muestra un mensaje de error en la pantalla, indicando que no se pudo cargar la estadística.

Ver Estadística: Hall of Fame

Permite al usuario visualizar las 16 canciones más destacadas según su actividad, representadas en forma de cuadrícula con las portadas de los álbumes correspondientes.

- Flujo principal (Visualización de Hall of Fame):

1. El usuario selecciona la opción “Hall of Fame” en la pantalla de *Stats*.
2. El sistema solicita a la API de *Spotify* las 16 canciones más destacadas del usuario.
3. La API responde con los datos necesarios, incluyendo nombres, artistas y portadas de los álbumes.
4. El sistema genera una cuadrícula con las portadas de los álbumes correspondientes.

5. El usuario puede pasar el ratón sobre una portada para visualizar el nombre de la canción y el artista.
6. Si el usuario pulsar el botón “Crear Playlist”, se envía la portada generada y los datos de las canciones a la API de *Spotify* para crear una nueva playlist.

Ver Estadística: Huella del Día

Permite al usuario visualizar un gráfico de línea que representa los minutos escuchados por cada hora del día, mostrando las horas de mayor actividad musical.

- Flujo principal (Visualización de Huella del Día):
 1. El usuario selecciona la opción “Huella del Día” en la pantalla de *Stats*.
 2. El sistema solicita a la API de *Spotify* los datos de escucha del usuario necesarios para calcular los minutos por hora.
 3. La API responde con los datos correspondientes.
 4. El sistema genera el gráfico de línea.
 5. El usuario puede pasar el ratón sobre los puntos de la línea para ver los minutos escuchados en una hora específica.
 6. El sistema destaca visualmente la hora con más minutos escuchados.

Ver Estadística: Estaciones Musicales

Permite al usuario visualizar, de forma gráfica, el artista y género más representativo de cada estación del año según su actividad musical.

- Flujo principal (Visualización de Estaciones Musicales):
 1. El usuario selecciona la opción “Estaciones Musicales” en la pantalla de *Stats*.
 2. El sistema solicita a la API de *Spotify* los datos necesarios para calcular el artista y género destacados de cada estación.
 3. La API responde con los datos correspondientes.
 4. El sistema calcula los datos y los agrupa por estaciones.
 5. El sistema genera un gráfico de tipo anillo dividido en cuatro segmentos, representando cada estación del año.
 6. El usuario puede pasar el ratón sobre uno de los segmentos del gráfico para que el sistema muestre, en un panel adicional, el artista y género destacados de la estación seleccionada.
 7. El sistema actualiza el panel dinámicamente según el segmento que el usuario seleccione con el ratón.

Ver Estadística: Tus Décadas

Permite al usuario visualizar en formato de histograma los álbumes de sus canciones favoritas organizados por décadas, destacando las épocas más representativas de su actividad musical.

- Flujo principal (Visualización de Tus Décadas):

1. El usuario selecciona la opción “Tus Décadas” en la pantalla de *Stats*.
2. El sistema solicita a la API de *Spotify* los datos de las canciones guardadas por el usuario en su biblioteca.
3. La API responde con los datos, incluyendo las fechas de lanzamiento de los álbumes asociados a las canciones.
4. El sistema organiza los datos por décadas y genera el histograma correspondiente.
5. El usuario puede desplazarse sobre el gráfico arrastrando con el ratón y explorar con mayor detalle las portadas, ampliando o reduciendo el aumento, mediante la rueda de desplazamiento.

Ver Estadística: La Bitácora

Permite al usuario explorar el historial completo de sus canciones guardadas en favoritos, organizadas cronológicamente.

- Flujo principal (Visualización de La Bitácora):

1. El usuario selecciona la opción “La Bitácora” en la pantalla de *Stats*.
2. El sistema solicita a la API de *Spotify* los datos de las canciones guardadas en favoritos, incluyendo la fecha en que fueron añadidas.
3. La API responde con los datos correspondientes.
4. El sistema genera un gráfico de barras que muestra, inicialmente, el número de canciones guardadas por año.
5. El usuario puede interactuar con el gráfico para explorar diferentes niveles de detalle:
 - Al hacer clic en una barra que representa un año, el sistema actualiza el gráfico para mostrar los meses dentro de ese año.
 - Al hacer clic en una barra que representa un mes, el sistema desglosa la información para mostrar los días dentro de ese mes.
6. Al pasar el ratón sobre cualquier barra, el sistema muestra un tooltip con el número de canciones guardadas en ese periodo.
7. El usuario puede navegar libremente entre los niveles (años, meses y días) y retroceder en cualquier momento para cambiar de periodo.

Ver Estadística: Índice de Interferencia

Permite al usuario comparar el cambio en sus gustos sobre la popularidad de las canciones que escucha.

- Flujo principal (Visualización de Índice de Interferencia):

1. El usuario selecciona la opción “Índice de Interferencia” en la pantalla de Stats.
2. El sistema solicita a la API de *Spotify* los datos necesarios.
3. La API responde con los datos correspondientes.
4. El sistema hace el cálculo de las dos popularidades medias y los presenta visualmente en forma de ondas sinusoidales animadas.
5. Si el usuario pulsa un botón de “Combinar Ondas”, el sistema genera una nueva onda, resultado de la suma matemática de las frecuencias de las dos ondas originales (interferencia).
6. El sistema muestra esta nueva onda junto a la diferencia numérica entre las dos popularidades.

Diseño

6.1. Arquitectura del Sistema

La arquitectura del sistema se basa por completo en el uso de *Next.js*, un framework de *React* que permite la creación de aplicaciones web, con la funcionalidad clave de un backend integrado. Desde la versión 13 de *Next.js* se introdujo el concepto de **App Router**, que crea las rutas de la web en base a la organización de las carpetas dentro del proyecto. Junto a esto, se introdujeron los **Route Handlers**, que permiten la creación de endpoints API REST de la misma manera, creando un backend integrado que hace de intermediario entre el cliente y los servicios externos. De esta manera, se consigue una arquitectura notablemente más simplificada, además de segura, ya que se puede controlar la exposición de datos sensibles al cliente.

Otra característica muy útil de *Next.js*, son los **Server Components**, que permiten renderizar componentes en el servidor en lugar del cliente. Solo aquellos componentes que sean necesarios serán enviados y ejecutados en el cliente, mejorando el rendimiento y la seguridad. Estos componentes pueden realizar llamadas a los endpoints proporcionados por los *Route Handlers*, recreando los roles de un sistema cliente-servidor tradicional. En el diagrama 6.1 se pueden ver representadas estas interacciones.

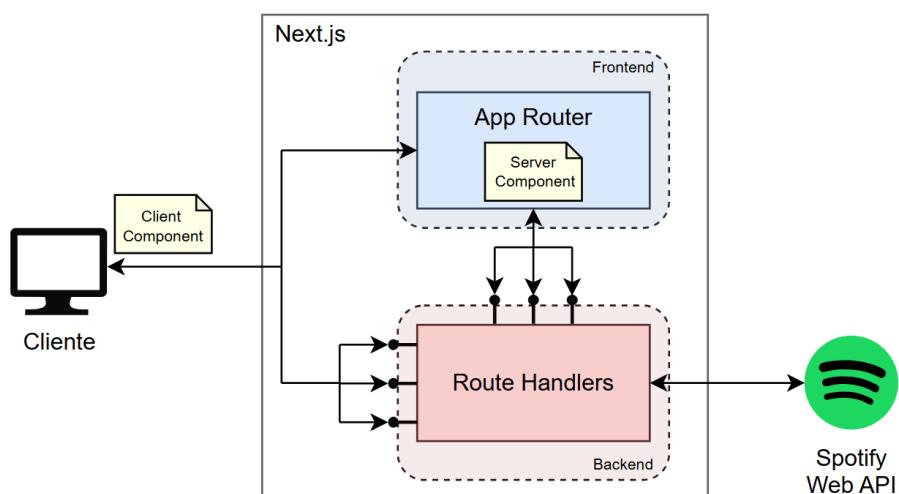


Figura 6.1: Diagrama de la arquitectura del sistema haciendo uso de *Next.js*.

6.1.1. Rutas del Frontend

Usando el *App Router*, las rutas del frontend se organizan en la carpeta `app/`, donde cada subcarpeta representa una ruta específica en la web. Dentro de cada carpeta se pueden encontrar archivos específicos que siguen una convención de nombres, y que definen su función. Las más importantes son:

- `layout.tsx`: Define el diseño compartido de los componentes que se encuentran anidados en las subcarpetas interiores.
- `page.tsx`: Contiene el contenido principal de una ruta específica. Representa la página renderizada cuando el usuario accede a esa ruta. **Para que una página sea accesible, debe existir un archivo `page.tsx` en la carpeta correspondiente.**
- `loading.tsx`: Muestra un indicador de carga mientras se obtienen datos o se renderizan componentes en una página.
- `error.tsx`: Gestiona errores específicos de una página, mostrando mensajes o interfaces para el usuario en caso de fallos.

En el caso de este proyecto, la jerarquía de carpetas que genera las páginas routeables de la web es la siguiente:

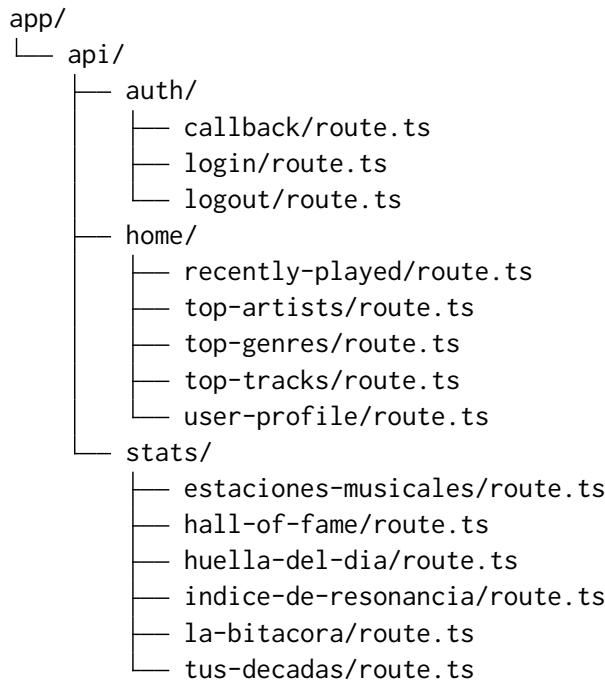
```
app/
  └── home/
    └── page.tsx (Ruta: /home)
  └── stats/
    └── page.tsx (Ruta: /stats)
  └── page.tsx (Ruta: /)
```

La ruta `/` (raíz) es la página principal, en donde el usuario puede iniciar sesión. Las rutas `/home` y `/stats` representan las páginas de estadísticas básicas y estadísticas avanzadas, respectivamente. En cada caso, el archivo `page.tsx` define el contenido principal de la página y es posible añadir otros archivos como `layout.tsx`, `loading.tsx` o `error.tsx` para mejorar la experiencia del usuario.

6.1.2. Endpoints del Backend

En el caso de la creación de endpoints mediante los *Route Handlers*, la estructura de carpetas es similar a la de las rutas del frontend, pero con la diferencia de que cada subcarpeta dentro de `app/api/` contiene un archivo `route.ts` que define el comportamiento del endpoint asociado. **Para que un endpoint sea accesible, debe existir un archivo `route.ts` en la carpeta correspondiente.**

La estructura de carpetas para los endpoints de este proyecto es la siguiente:



Los endpoints en `/api/auth/` gestionan el inicio de sesión del usuario, mientras que los de `/api/home/` y `/api/stats/` se encargan de obtener y procesar los datos necesarios para las estadísticas. El contenido del fichero `route.ts` sigue una convención concreta que `Next.js` reconoce y utiliza para gestionar las peticiones, la cual se explicará con más detalle en el capítulo de [Implementación](#).

6.2. Diagrama de Componentes de React

Los componentes en *React* son las piezas fundamentales para construir interfaces de usuario. Cada componente puede ser reutilizable, contener su propio estado y lógica, y ser combinado con otros para formar estructuras más complejas. Para describir estas interfaces de manera declarativa, *React* utiliza **JSX** (JavaScript XML), una extensión de sintaxis de JavaScript, que permite escribir el código de los componentes con una sintaxis similar a *HTML*. Aunque no sea obligatorio, su uso es ampliamente adoptado en proyectos *React*.

Las interfaces se suelen diseñar en forma de jerarquías de componentes, en las que los componentes “padre” organizan y controlan el comportamiento y la presentación de los componentes “hijo”. Esta organización jerárquica facilita el mantenimiento y la escalabilidad del código, ya que cada componente tiene una responsabilidad definida. En este proyecto, se ha creado una jerarquía de componentes (diagrama 6.2) que refleja las diferentes funcionalidades de la aplicación. A la hora de construir la web, `Next.js` y *React* se encargan de anidar los componentes necesarios y renderizarlos, o enviarlos al cliente, según sea necesario.

6.2. Diagrama de Componentes de React

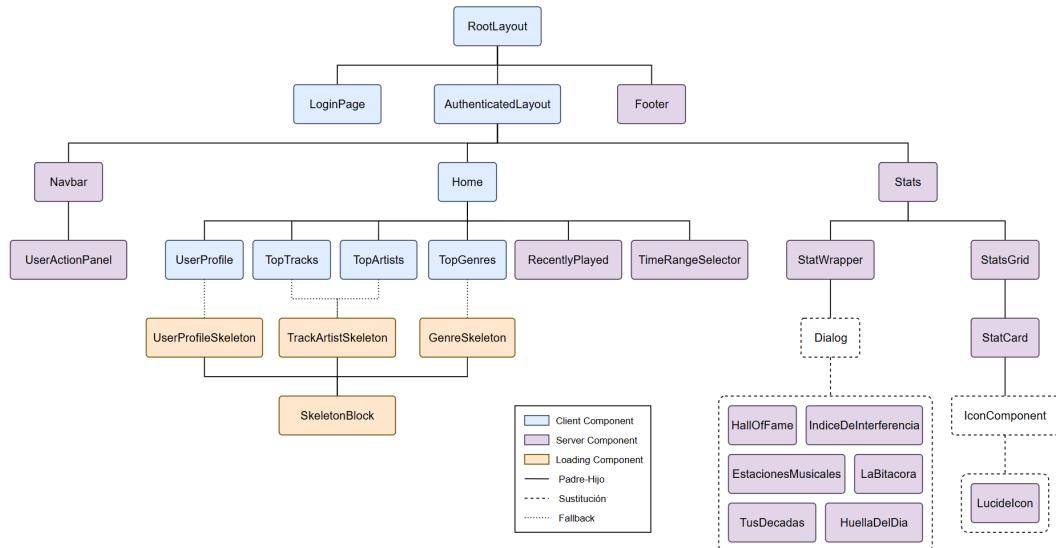


Figura 6.2: Diagrama de la jerarquía de componentes de React creados en el proyecto.

Como se muestra en el diagrama, se pueden diferenciar tres tipos diferentes de componentes, según sus características. *React* define dos principales clases: los **Client Components** y, los ya mencionados, **Server Components**. *Next.js* hace uso concreto de esta diferenciación para optimizar las páginas. Como su nombre indica, los *client components* son componentes de *React* que se ejecutan en el navegador del cliente. Para ello, es necesario enviar toda la lógica junto con el HTML y los estilos, haciendo que este tipo de componentes sean más pesados y requieran más computación por parte de la máquina del usuario. Como respuesta, se introdujeron los *server components*, para que los componentes que no contuviesen funcionalidades de interacción complejas pudieran ser renderizados en el servidor, enviando solamente el HTML y los estilos necesarios al cliente. Con esto se reduce considerablemente la carga del navegador y mejora la rapidez de la web.

El tercer tipo de componente, los **Loading Components**, no son una clasificación oficial, si no que se ha querido hacer una diferenciación con respecto a los componentes “comunes”. No añaden funcionalidad ni información concreta a la página, solamente se usan como sustitutos temporales mientras se cargan los componentes finales. Son especialmente útiles para usarlos junto con los *server components*, ya que estos pueden tardar en ser enviados al cliente por el proceso de renderizado (y carga de datos) en el servidor. Hasta entonces, *Next.js* recurre a mostrar el *loading component* para dar un feedback al usuario (figura 6.1). Además, estos componentes pueden modelarse de tal forma que representen el “esqueleto” del estilo del componente final. Todo esto permite que el proceso de carga se sienta más rápido e informativo, mejorando la experiencia del usuario.

```

1 <Suspense fallback={<UserProfileSkeleton />}>
2   <UserProfile />
3 </Suspense>
```

Algoritmo 6.1: Entorno Suspense para mostrar un fallback mientras se carga el componente *UserProfile*.

Por último, hay algunos componentes que se han definido para realizar cargas dinámicas. El contenido no será enviado hasta que el usuario realice alguna acción que dispare una petición a *Next.js*, sustituyendo en el lugar indicado uno de los componentes preestablecidos. En este proyecto, se han definido dos componentes que se comportan así:

- **Dialog:** Se utiliza para cargar en una ventana modal una de las seis estadísticas disponibles, cada una siendo un componente autocontenido. Dependiendo de la selección del usuario, se cargará uno de estos subcomponentes en el lugar correspondiente. También se puede inferir que solo se podrá mostrar una estadística a la vez.
- **IconComponent:** Está asociado con los iconos representativos de cada estadística, que se muestran en las tarjetas (*StatCard*) y que el usuario puede clicar para ver. Utiliza el paquete *LucideIcon* para cargar diferentes iconos, ya que, en este paquete, cada ícono es representado como un componente de *React*.

Utilizando de manera consciente estas herramientas del framework, se reduce la cantidad de código a enviar al cliente. En el caso de este proyecto, se ha procurado usar, en la medida de lo posible, los *server components*; solo recurriendo al uso de los *client components* en casos necesarios, ya sea por la gestión de estados para interactividad avanzada o llamadas a endpoints de datos.

6.3. Interfaz de Usuario

La interfaz de usuario es uno de los aspectos más flexibles dentro del desarrollo de software, ya que permite una gran libertad creativa y de implementación. En este proyecto, se plantearon varias opciones: utilizar una base de diseño preexistente, como guías visuales o sistemas de diseño consolidados, o desarrollar una interfaz completamente personalizada. Debido a que este trabajo pertenece al ámbito de la informática y no del diseño gráfico, muchas decisiones se orientaron hacia la practicidad, priorizando herramientas y enfoques que permiten simplificar el desarrollo, sin sacrificar la experiencia de usuario. A continuación, se describen las decisiones tomadas al respecto.

6.3.1. Principios de Diseño

Como *Spotify* es una plataforma ampliamente conocida y tiene una identidad visual muy reconocible, se decidió inspirar la interfaz en su diseño. La plataforma proporciona una guía para los desarrolladores que incluye recomendaciones muy útiles [1]. De entre ellas, se han seleccionado aquellos elementos que resultan relevantes para el uso y alcance de esta aplicación, que se sintetizan en los siguientes puntos:

Colores

El principal identificador visual de *Spotify* es su color verde, llamado **Spotify Green**. Este color es el que se ha utilizado como el color primario para la página web. Además, se han incorporado el blanco y negro oficiales y también algunos colores complementarios, como el zul y el rojo, para aportar variedad en las situaciones en las que se necesite.

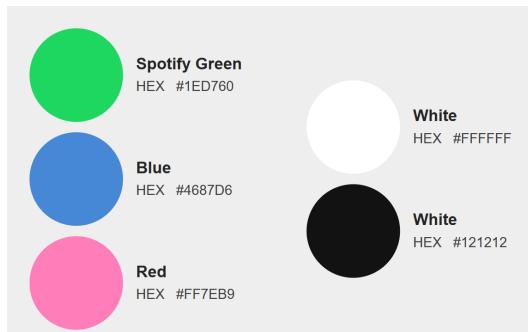


Figura 6.3: Colores seleccionados para la paleta de colores de la aplicación.

Estas combinaciones respetan las recomendaciones de la plataforma, que desaconseja el uso de colores sobresaturados, y también se ha ajustado claridad de los colores en ciertas ocasiones para mejorar el contraste en la lectura de texto.

Tipografía

Aunque *Spotify* no proporciona acceso público a su tipografía oficial, sugieren usar fuentes sans-serif como *Helvetica* o *Arial*. En el caso de este proyecto, se ha optado por hacer uso de la tipografía **Inter** (figura 6.4), ya que se asemeja más a la original que las recomendadas. Podemos encontrarla en *Google Fonts*, pero *Next.js* nos evita la necesidad de tener que descargarla, ya que está incluida de forma predeterminada en el framework.

```
abcdefghijklmnoprstuvwxyz
ABCDEFGHIJKLMNPQRSTUVWXYZ
0123456789 (!#$%&/.|*`@',?:;)
```

Figura 6.4: Muestra de caracteres de la tipografía *Inter*.

6.3.2. Tailwind CSS

Para el estilizado de los elementos de la interfaz se ha utilizado *Tailwind CSS*, una librería de utilidades que permite desarrollar estilos de manera rápida y eficiente. En vez de crear un estilo agrupado que se asocie a cada elemento, el enfoque de *Tailwind* se basa en agregar pequeñas clases utilitarias predefinidas a cada elemento mediante el atributo `className` en JSX (figura 6.2), aplicandole así el resultado de la combinación de todos esos estilos.

```
1 <main className="flex flex-col items-center justify-center">
2   <h1 className="text-2xl font-bold mb-4">Bienvenido</h1>
3   <a href="/api/auth/login"
4     className="px-4 py-2 bg-green-500 text-white rounded">
5     Sign In </a>
6   </main>
```

Algoritmo 6.2: Ejemplo de aplicación de estilos a un componente JSX usando *Tailwind CSS*.

De esta manera se evita la necesidad de definir manualmente grandes cantidades de estilos en archivos CSS. Además de los estilos estéticos, *Tailwind* aporta muchas clases relacionadas con la adaptabilidad de los componentes a diferentes dispositivos, para generar interfaces *responsive* con facilidad.

Otra razón por la que se ha decidido utilizar este sistema, es su integración con *React* y *Next.js*. Es un stack de tecnologías frontend muy popular, por lo que existe mucha documentación al respecto y se garantiza una gran compatibilidad. Además, *Tailwind* es muy flexible y permite personalizar los estilos de manera sencilla para ajustarlos a cualquier tipo de proyecto.

6.3.3. Páginas y Componentes Principales

Para finalizar con la caracterización de la interfaz de usuario, se presentarán los componentes de la web, la distribución de los elementos dentro de ella y las decisiones tomadas al respecto. Dado el enfoque más técnico de este documento, solo se han incluido las figuras de los componentes principales. Los detalles específicos de cada estadística y elementos secundarios de la interfaz se han trasladado al [Anexo A](#).

La primera página presentada al usuario es la página de **inicio de sesión** (figura 6.5), que se encuentra en la ruta raíz. Solo se presenta una interacción posible, que es con el botón de *Log In*. Además de la parte puramente funcional, esta pantalla inicial permite una introducción visual al estilo de la web, centrado en los colores ya presentados y la decisión de usar un fondo negro, al igual que la aplicación nativa de *Spotify*.

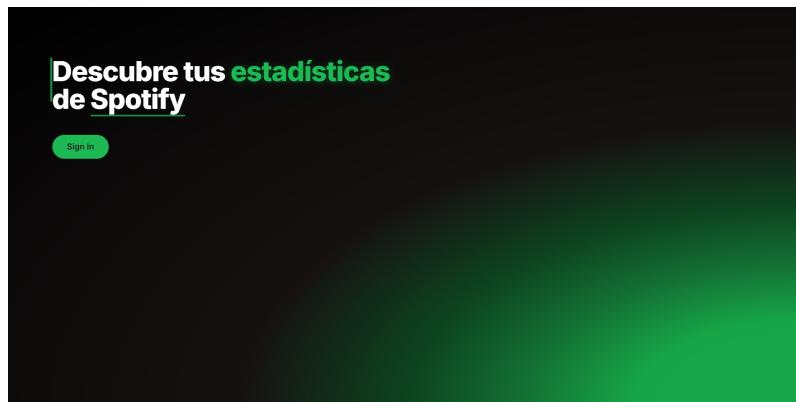


Figura 6.5: Página de inicio de sesión del proyecto.

Una vez dentro, la navegación entre las diferentes secciones se realiza a través de una **barra de navegación** (figura 6.6), accesible en todo momento. En esta barra se incluyen el logo y nombre de la aplicación, los botones de navegación principales y el perfil del usuario. Desde este último, el usuario puede cerrar sesión en cualquier momento mediante la opción de *Log Out*. Este elemento fijo en la interfaz facilita la orientación dentro de la aplicación y permite un acceso rápido a las secciones esenciales.

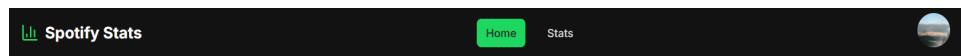


Figura 6.6: Barra de navegación fijada en la parte superior de la web.

La primera sección tras el inicio de sesión es el **Home** (figura 6.7), donde se presentan las estadísticas básicas del usuario. Para organizar la información, se ha optado por un diseño basado en módulos, que encapsulan cada una de las estadísticas. Demás del los datos del perfil del usuario, se muestran los tres tops (*Top Tracks*, *Top artists* y *Top Genres*). Justo debajo, manteniendo una separación, se encuentra la sección de *Recently Played*, que se puede ver al desplazar la ventana.

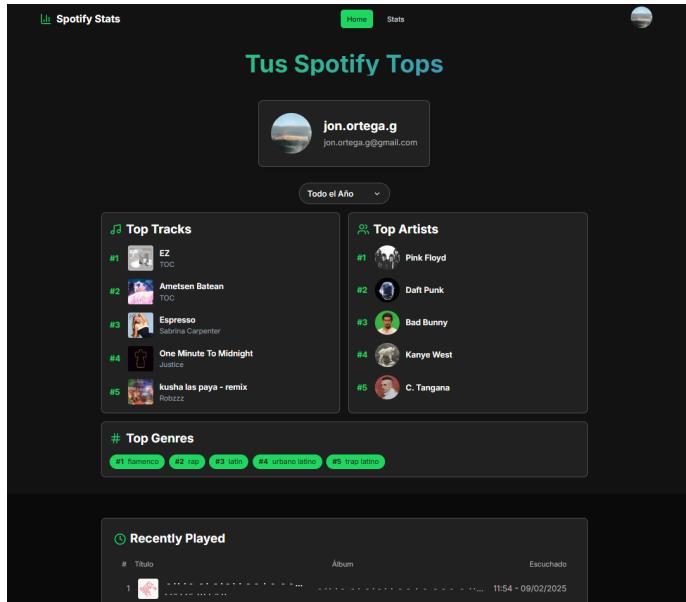


Figura 6.7: Página de *Home* con las estadísticas básicas.

La segunda gran sección es la de **Stats** (figura 6.8), donde se han implementado las estadísticas avanzadas. Al igual que en el *Home*, se mantiene el diseño modular basado en tarjetas, pero en este caso con una disposición más flexible para generar una presentación más interesante. Al hacer clic sobre cualquiera de ellas, se abre una **ventana modal** (figura 6.9) sobre la página, en la que se carga y renderiza la estadística seleccionada. Oscureciendo y difuminando el fondo, se da una sensación de profundidad y crea una separación para foco en la estadística. El usuario puede cerrar la ventana modal en cualquier momento haciendo clic fuera de ella o mediante el botón de cierre.

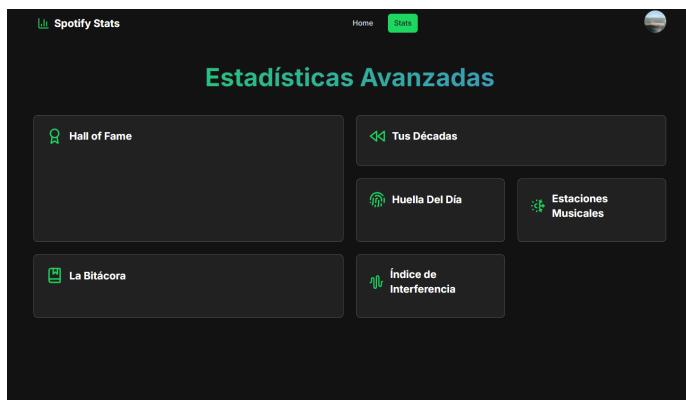


Figura 6.8: Página de *Stats* con las tarjetas de estadísticas avanzadas.

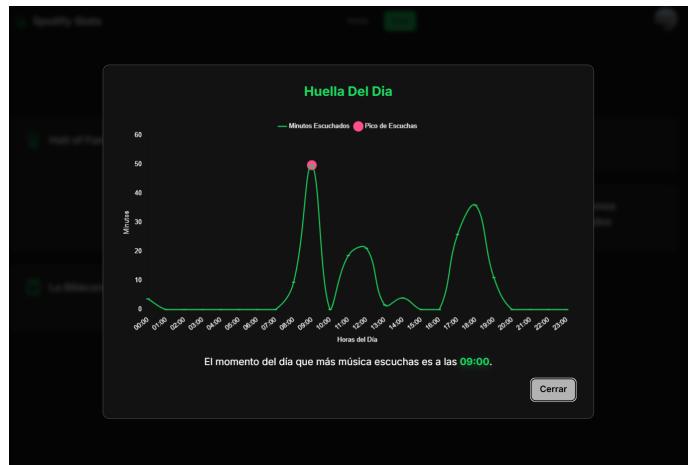


Figura 6.9: Ventana modal que contendrá la estadística a mostrar (en este ejemplo, *Huella Del Día*).

Finalmente, en la parte inferior de la web se encuentra el **footer** (6.10), un elemento presente en todas las páginas, incluyendo la pantalla de inicio de sesión. Su función principal es proporcionar un botón para el acceso a la política de privacidad, requisito de los términos de *Spotify* para el uso de su API. Al igual que en las estadísticas, la información se muestra en una ventana modal, evitando la necesidad de cargar una nueva página y manteniendo la coherencia en la interacción del usuario con la aplicación.



Figura 6.10: Footer anclado al pie de todas las páginas de la web.

Aunque las figuras incluidas en el documento hayan sido capturadas en una pantalla de escritorio, todos los elementos dentro de los componentes han sido implementados con un diseño *responsive* en mente, asegurando una visualización clara tanto en pantallas grandes como en dispositivos móviles con pantallas más reducidas.

6.4. Diagramas de Secuencia

En esta sección se presentan los diagramas de secuencia más importantes. Se han seleccionado aquellos que presentan una complejidad particular o requieren una explicación más detallada para su correcta interpretación. El resto de los diagramas correspondientes a los casos de uso estarán disponibles en el [Anexo B](#).

Iniciar Sesión

Aunque la acción para el usuario sea simple, este caso de uso requiere varios pasos y llamadas a la API de *Spotify* por parte de la lógica de la aplicación. La complejidad viene principalmente de la gestión de los errores, ya que, durante este proceso, existen dos puntos en los que el usuario puede rechazar (o ser rechazado) el inicio de sesión. Esos dos casos son: si el usuario introduce las credenciales incorrectas en la autenticación o si el usuario decide no dar la autorización necesaria a la aplicación para trabajar con sus datos. En el diagrama [6.11](#) se caracterizan todas estas posibilidades con los marcos de interacción alternativas.

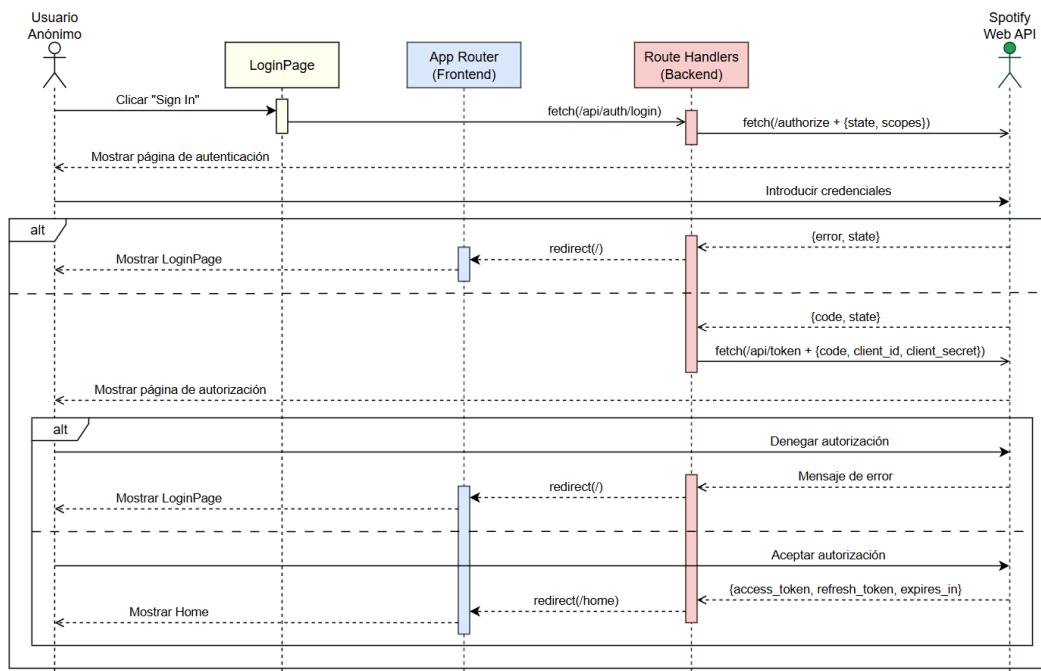


Figura 6.11: Diagrama de secuencia: Iniciar Sesión.

Acceder a Home

En este caso de uso, la aplicación carga la página de *Home* junto con todos los componentes de las estadísticas básicas. En el diagrama [6.12](#) se muestra un marco que representa un proceso paralelo. Se ha querido remarcar esta característica de la carga, ya que todos los componentes se renderizan de manera asíncrona, gracias a la funcionalidad del entorno *Suspense* de *Next.js* mencionado en la sección [Diagrama de Componentes de React](#).

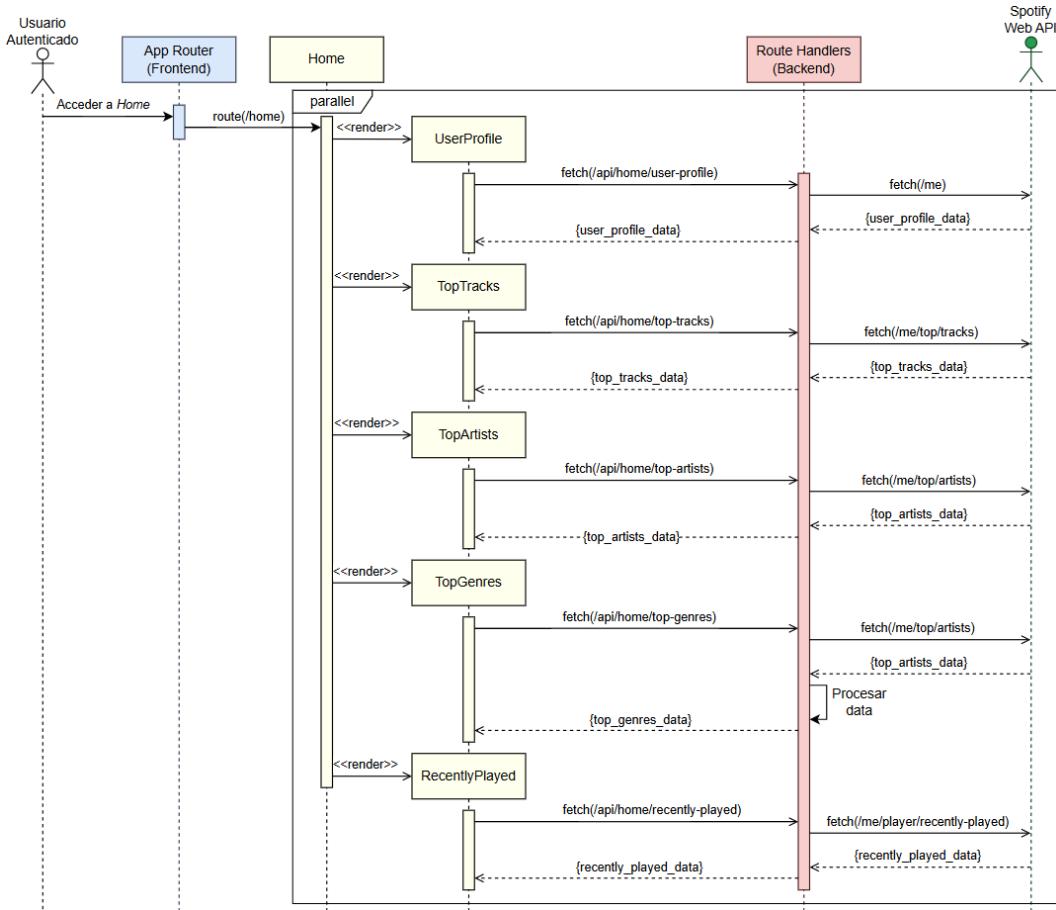


Figura 6.12: Diagrama de secuencia: Acceder a Home.

Acceder a Stats

Al acceder a la página de *Stats*, se renderizan varias instancias del mismo componente *StatCard*, una para cada estadística. Luego, cuando el usuario selecciona una de ellas, el componente envía el *statId* de la estadística correspondiente al componente *StatWrapper*. Por último, el *StatWrapper* renderizará el componente asociado al identificador de manera dinámica, cuyos diagramas de secuencia se han definido de manera independiente. El acceso a *Stats* se ha caracterizado en el siguiente diagrama 6.13:

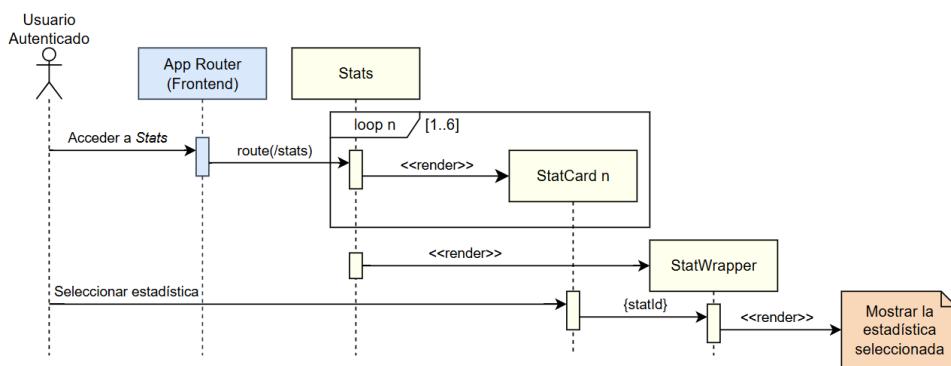
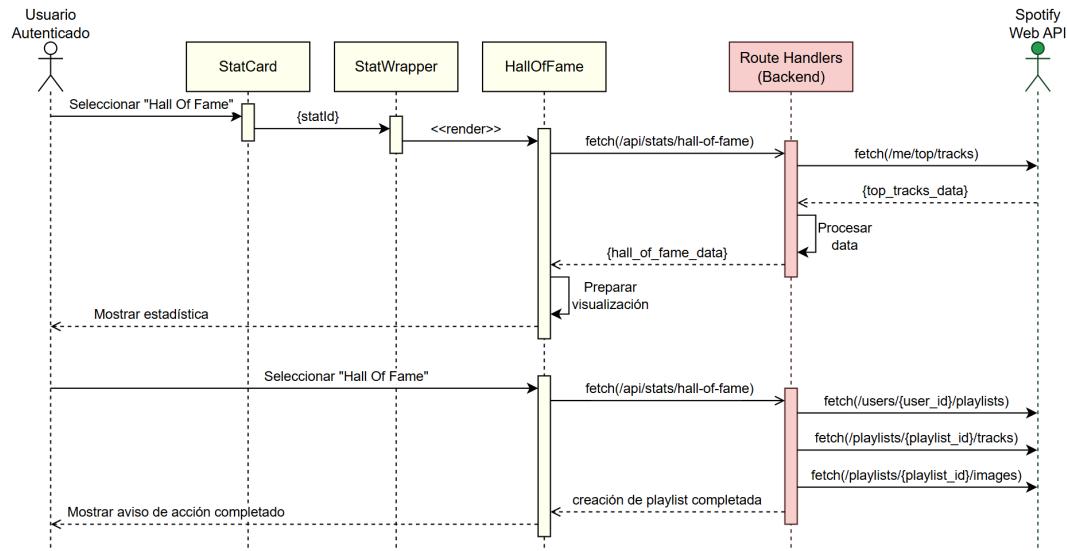


Figura 6.13: Diagrama de secuencia: Acceder a Stats.

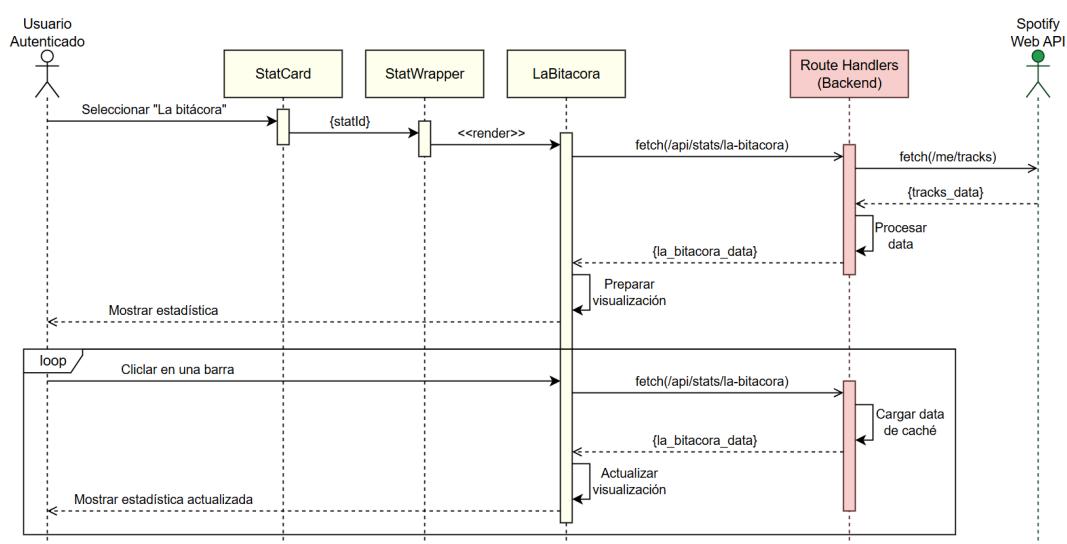
Ver Hall Of Fame

El renderizado y la carga de la estadística *Hall Of Fame* se realiza de manera estándar, pero ofrece una funcionalidad añadida, identificado con el caso de uso extendido **Crear Playlist** (diagrama 6.14). Al clicar sobre el botón, se realizan varias interacciones en cadena con la API de *Spotify* para crear la playlist, añadir canciones y establecer una nueva portada.



Ver La Bitácora

En el caso de esta estadística, se permite al usuario navegar entre los distintos niveles temporales de datos. Para evitar consumir constantemente la API de *Spotify*, los datos se cargan en un caché temporal en el servidor. De esta manera, cuando el usuario solicite los datos, se enviarán con menos latencia y se evita la sobrecarga de llamadas a *Spotify*. En el diagrama 6.15 se muestra esa interacción en el marco de bucle.



6.5. Diseño de la Seguridad

Como esta aplicación maneja información privada de los usuarios, un aspecto crítico en el diseño y desarrollo de la misma es la seguridad. En esta sección se describen las medidas implementadas para proteger los datos de posibles ataques comunes y reducir las vulnerabilidades del sistema.

6.5.1. Gestión de Credenciales

La mala gestión de credenciales es un punto de vulnerabilidad común en aplicaciones web. En el caso del proyecto, se manejan varias credenciales sensibles, especialmente durante el flujo de inicio de sesión. A continuación se detallan las medidas aplicadas a cada una.

Client ID y Client Secret

En la creación de una aplicación que hace uso de la API de *Spotify*, se obtiene un **client_id** y un **client_secret** que son necesarios para la autenticación con la API. Estas credenciales pertenecen al desarrollador y deben ser protegidas adecuadamente para evitar su exposición a terceros no autorizados. Se almacenan exclusivamente en el servidor, en un archivo de variables de entorno (`.env.local`), que no se incluye en el control de versiones. En el momento del despliegue, estas variables se cargan en el entorno de producción, que han de ser anteriormente indicadas de manera manual a través de un panel de configuración de *Vercel*. Se puede acceder a estas variables en el lógica del servidor mediante el objeto `process.env`.

Prevención de CSRF y Scopes

Para prevenir ataques de *Cross-Site Request Forgery* (CSRF), cada solicitud de inicio de sesión incluye un valor de estado (**state**) generado de forma aleatoria. Este valor es único para cada sesión y se valida al recibir la respuesta del servidor de autenticación. Este mecanismo, recomendado explícitamente en la documentación de *Spotify* [2], garantiza que las solicitudes sean legítimas y originadas únicamente desde el cliente autorizado, evitando que actores maliciosos puedan ejecutar acciones en nombre del usuario.

Por otro lado, durante el paso de la autorización en el proceso de inicio de sesión, se respeta el principio de mínimos privilegios, donde solo se solicitan los permisos necesarios y ninguno más. Los usuarios son informados claramente sobre los **scopes** solicitados y su propósito antes de otorgar los permisos. De esta manera se consigue reducir el riesgo de exposición innecesaria de datos.

Access Token y Refresh Token

Una vez que el usuario ha iniciado sesión y la aplicación ha obtenido un **access_token** y un **refresh_token**, estos se almacenan en unas cookies correspondiente. La configuración de estas cookies es muy importante para protegerlas frente a ataques comunes como *Cross-Site Scripting* (XSS) o *Man-In-The-Middle* (MITM). Las siguientes marcas (**flags**) permiten configurar la seguridad necesaria:

- **httpOnly**: Indicando esta opción como “true”, asegura que las cookies no puedan ser accedidas por código JavaScript en el navegador, protegiéndolas contra ataques de XSS.
- **secure**: Indicando esta opción como “true”, las cookies solo pueden ser transmitidas a través de conexiones HTTPS, protegiéndolas contra ataques de MITM. Esta opción solo es necesaria en un entorno de producción, es posible desactivarlo durante el desarrollo.
- **maxAge** Estableciendo un tiempo de vida limitado, se garantiza que se las cookies se eliminen automáticamente una vez cumplido el periodo concretado, minimizando el impacto de un posible compromiso.

6.5.2. Routing y Conexiones Seguras

Next.js proporciona un sistema de **middleware** que permite ejecutar procesos antes de que se manejen las solicitudes de las rutas. En este proyecto, se ha implementado un middleware que verifica la validez de las cookies de sesión antes de permitir el acceso a las rutas protegidas. Si no existe una cookie con un access token, el usuario es redirigido a la página de inicio de sesión, garantizando que solo los usuarios autenticados puedan acceder a las estadísticas y los datos personales.

Para la comunicación entre el cliente y la aplicación, al desplegarse en *Vercel*, éste proporciona automáticamente **conexiones HTTPS** para todas las solicitudes. Esto garantiza la confidencialidad de los datos transmitidos entre el cliente y el servidor, protegiendo contra ataques de intercepción como el MITM.

6.5.3. Otras Medidas

Del mismo modo que se han implementado mecanismos explícitos para prevenir ataques comunes, la seguridad de una aplicación no solo depende de configuraciones específicas, sino también del entorno de desarrollo y de las prácticas adoptadas a lo largo del ciclo de vida del software. La seguridad implícita, aquella que surge como resultado de buenas prácticas y herramientas adecuadas, desempeña un gran papel.

- **Análisis estático del código con ESLint**: Un linter como *ESLint* permite detectar errores, malas prácticas y patrones potencialmente inseguros en el código TypeScript antes de que lleguen a producción. Integrarlo en el flujo de desarrollo facilita la identificación temprana de vulnerabilidades y la posibilidad de corregirlos antes de que se conviertan en problemas de seguridad.
- **Dependencias actualizadas**: La aplicación se construye sobre versiones recientes y estables de *Next.js*, *React* y *Node.js LTS*, lo que permite beneficiarse de mejoras en seguridad y reducir la exposición a vulnerabilidades conocidas; un riesgo muy común derivado del uso de software obsoleto.
- **Ejecución de pruebas**: Realizar pruebas sistemáticas permite identificar fallos inesperados, comportamientos anómalos o configuraciones incorrectas en la aplicación.
- **Prácticas recomendadas**: Seguir las guías oficiales de seguridad, tanto de la API de *Spotify* como de *Next.js*, garantiza que se cumplen los estándares más recientes.

Implementación

7.1. Entorno de Desarrollo Local

Para garantizar la reproducibilidad del proyecto, en esta sección se describirá el entorno de desarrollo en el que se ha realizado y el acceso al repositorio con el código fuente.

7.1.1. Tecnologías y Versiones

El proyecto se ha desarrollado en un entorno **Windows 10**, aunque esto no influye en la compatibilidad, ya que todas las tecnologías utilizadas son multiplataforma, lo cual permite realizar el desarrollo en cualquier sistema operativo sin restricciones. En la tabla 7.1 se presentan las tecnologías principales empleadas en el proyecto junto con sus versiones correspondientes.

	Tecnología	Versión
Principales	Next.js	15.1.2
	React	19.0.0
	Node.js	22.12.0
	Axios	1.7.9
	Canvas	3.1.0
	Chart.js	5.3.0
	D3.js	7.9.0
	Konva	19.0.2
	RadixUI	1.1.5
	Lucide	0.469.0
Desarrollo	TypeScript	5
	Tailwind CSS	3.4.1
	ESLint	9
	Jest	29.7.0
	Testing Library	16.1.0

Tabla 7.1: Dependencias usadas en el desarrollo, junto con sus versiones.

Cabe destacar que el gestor de paquetes utilizado en el proyecto ha sido **pnpm** (*Performant Node Package Manager*), en lugar de npm (*Node Package Manager*) o yarn (*Yet Another Resource Negotiator*). La elección de pnpm se debe a sus mejoras en la gestión de dependencias y optimización del uso del espacio de almacenamiento. A diferencia de npm, pnpm utiliza un sistema de enlaces en lugar de duplicar archivos en cada proyecto, lo que reduce significativamente el consumo de espacio. Además, pnpm es completamente compatible con los paquetes del registro de npm, lo que garantiza su interoperabilidad con la mayoría de los ecosistemas de desarrollo basados en *Node.js*.

7.1.2. Instalación y Configuración del Proyecto

El proyecto se encuentra alojado en un repositorio de *GitHub*¹ para evitar pérdidas y tener disponibilidad completa al código. Para ejecutarlo localmente, además de haber instalado *Node.js*, hay que clonar el repositorio, acceder a la carpeta y ejecutar los comandos de instalación y ejecución. A continuación, se muestran los comandos necesarios para realizar estos pasos:

```

1  # Clonar el repositorio
2  git clone https://github.com/jonortega/tfg-app-spotify.git
3
4  # Acceder al directorio del proyecto
5  cd tfg-app-spotify
6
7  # Instalar dependencias
8  pnpm install
9
10 # Ejecutar el servidor de desarrollo
11 pnpm run dev

```

Algoritmo 7.1: Comandos de instalación y ejecución inicial del proyecto.

Tras estos pasos, la página web estará accesible en localhost:3000. Es necesario agregar un fichero de variables de entorno .env.local, ya que, por seguridad, no se registra en el sistema de control de versiones. El contenido de dicho fichero se muestra en el [Anexo C](#) (algoritmo C.1).

Dos de las variables de entorno necesarias para poder tratar con la API de Spotify, son el **Client ID** y el **Client Secret**. Estos dos valores se obtienen al realizar el registro de la aplicación en la plataforma de desarrollo de Spotify. En la siguiente sección, se explicará cómo realizar este registro y dónde obtener dichas variables.

¹Repositorio del proyecto: <https://github.com/jonortega/tfg-app-spotify>

7.2. Registro de la Aplicación en Spotify

Para poder obtener datos de la Web API de *Spotify*, es necesario registrar la aplicación en su plataforma de desarrollo². Tras iniciar con una cuenta, se nos presentará un panel donde podremos crear una nueva app. *Spotify* pedirá algunos datos (figura 7.1), que tendremos que rellenar. Los campos como *Redirect URIs* pueden ser modificados posteriormente, ya que tendremos que actualizarlo con el dominio indicado tras el despliegue de la aplicación.

Figura 7.1: Panel de creación de app en la plataforma de *Spotify*.

Al aceptar los términos y crear la app, tendremos acceso al *Client ID* y *Client Secret*. Estos se encuentran en los ajustes (*settings*) y tendremos que expandir el panel para poder ver los dos valores (figura 7.2).

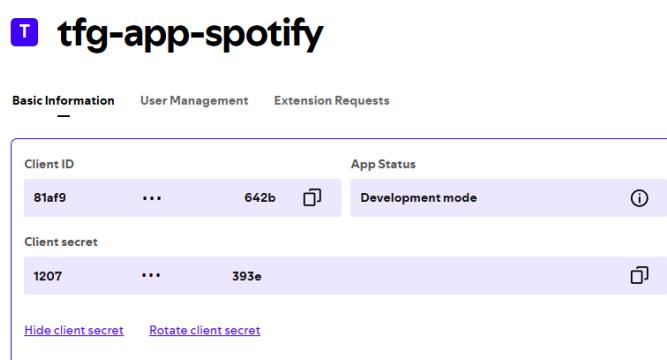


Figura 7.2: Panel de ajustes con el *Client ID* y el *Client Secret*.

²Spotify for Developers: <https://developer.spotify.com/>

En el caso en el que el *Client Secret* haya sido comprometido, es posible generar uno nuevo, anulando el anterior y evitando tener que desechar la aplicación registrada. También se muestra un campo llamado *App Status* con el valor **Development Mode**. Esto significa que la aplicación registrada está en “desarrollo”, por lo que existen las siguientes restricciones:

- Un máximo de 25 usuarios (cuentas verificadas de *Spotify*) pueden usar la aplicación.
- Cada usuario tiene que estar registrado en una lista de permitidos (*allowlist*) de la plataforma.

Es posible eliminar estas restricciones, enviando una solicitud a *Spotify* para cambiar el estado de *Development Mode* a **Extended Quota Mode**. En el caso de que sea aceptada, se elimina cualquier restricción sobre el número de usuarios permitidos y no será necesario registrarlos anteriormente, además de ampliar el umbral de la tasa de peticiones (*rate limit*). En este trabajo **no se va a realizar dicha solicitud**, por lo que tendremos que acogernos a las limitaciones impuestas en el modo de desarrollo. Esto requiere que se registren las cuentas de usuarios que vayan a probar la aplicación en la pestaña de gestión de usuarios (figura 7.3).

#	Name	Email	Date	...
1	[REDACTED]	[REDACTED]@gmail.com	January 4, 2025	...
2	[REDACTED]	[REDACTED]@gmail.com	January 4, 2025	...

Figura 7.3: Panel de gestión de usuarios que tienen acceso a la aplicación.

Con estos pasos, se habrá realizado correctamente el registro de la aplicación y obtención del *Client ID/Secret*, imprescindibles para la comunicación con la API de *Spotify*. En la plataforma de desarrollo, además, tenemos accesible un panel para poder monitorear la actividad de la aplicación (Anexo C, figura C.1).

7.3. Ciclo de Desarrollo

El ciclo de desarrollo del proyecto ha seguido un enfoque híbrido entre los modelos **incremental** e **iterativo**. En concreto, se ha construido en tres incrementos, realizados en diferentes puntos del proyecto. En cada uno, se han añadido más funcionalidades (incremental) y refinando las ya existentes (iterativo). A continuación se describe cada incremento con más detalle.

Incremento I: Prototipado Rápido

Durante la fase inicial de adquisición de competencias, se desarrolló un prototipo rápido para validar la viabilidad técnica del proyecto. No se buscaba crear la base desde donde seguir trabajando una vez iniciase el desarrollo formal; si no que, utilizando herramientas más simples, se quería obtener una idea preliminar de cómo se podrían implementar las funcionalidades principales. No se utilizaron las tecnologías finales del desarrollo, por lo que gran parte del código implementado no fue reutilizado.

En este prototipo se implementaron las siguientes funcionalidades, en diferentes grados de completado:

- Inicio de sesión de un usuario y obtención de los tokens.
- Renovación del `access_token` mediante el `refresh_token`.
- Obtener los *top items* del usuario.
- Obtener las canciones favoritas del usuario.
- Obtener los *track features* (**no implementado en el producto final**).
- Prototipos de las estadísticas:
 - *Hall Of Fame*
 - *Huella Del Día*
 - *La Bitácora*

Incremento II: Producto Mínimo Viable (PMV)

Una vez terminada la planificación del proyecto, se dio comienzo a una primera implementación del sistema, desarrollando una base de código que formaría parte del producto final. En esta fase, se utilizaron todas las tecnologías finales seleccionadas para el desarrollo, incluyendo *Next.js*, *React*, *TypeScript* y *Tailwind CSS*.

Durante este incremento, se completaron las siguientes partes fundamentales del sistema:

- Estructura completa de páginas de la aplicación web.
- Implementación de los endpoints mediante *Route Handlers*.
- Inicio de sesión y cierre de sesión seguros.
- Gestión segura de los tokens.
- Implementación de las estadísticas básicas *Top Tracks*, *Top Artists*, *Top Genres* y *Recently Played*.
- Desarrollo inicial de las seis estadísticas avanzadas finales en un estado funcional básico.

Si bien estas estadísticas se implementaron de manera inicial y sin refinamientos, su propósito principal fue definir la interacción entre el frontend y el backend, establecer los tipos de datos y endpoints exactos requeridos, y detectar posibles problemas en la presentación de la información. Asimismo, esta fase permitió identificar la necesidad de herramientas adicionales, como bibliotecas de gráficos, en caso de que las opciones disponibles no fueran suficientes.

En términos generales, este incremento sirvió para caracterizar con mayor precisión el alcance del proyecto y definir con claridad los aspectos que deberían considerarse en las siguientes etapas de análisis y diseño. A diferencia del prototipo inicial, gran parte del código desarrollado en esta iteración se integró directamente en la versión final del sistema.

Incremento III: Producto Final

Finalmente, en el último incremento se completaron todas las funcionalidades secundarias pendientes, junto con una serie de optimizaciones de rendimiento, ajustes en las estadísticas avanzadas y mejoras en la interfaz y experiencia de usuario. Esta fase marcó la consolidación del sistema y la preparación del código para su despliegue.

Las principales funcionalidades implementadas y correcciones realizadas fueron:

- Posibilidad de cambiar el rango temporal en los módulos *Top Tracks*, *Top Artists* y *Top Genres*.
- Implementación de la funcionalidad para crear playlists en *Hall Of Fame*.
- Desarrollo de la estadística *Tus Décadas* utilizando la librería *Konva* para una mayor optimización.
- Mejora en la presentación y rendimiento de la estadística *Índice de Interferencia* con *D3.js*.
- Corrección en la estructura de datos enviada desde el backend a la estadística *La Bitácora*.
- Implementación de la ventana informativa sobre la *Política de Privacidad*.
- Reemplazo de la implementación manual de la ventana modal de las estadísticas por la librería *Radix UI*.
- Creación de componentes de carga (*loading components*).
- Gestión de errores encontrados al probar con diferentes cuentas de usuario.
- Desarrollo de una pantalla de carga específica para las estadísticas avanzadas.
- Refactorización del código para crear funciones reutilizables, como en la obtención de datos (*fetch*), facilitando la gestión de los componentes.
- Preparación del código para su despliegue en *Vercel*, especialmente la gestión de variables de entorno.

Este último incremento permitió completar todas las funcionalidades planeadas y refinar el sistema en su conjunto. Además de las mejoras implementadas, se documentaron posibles optimizaciones futuras, aunque, dentro del alcance definido para el proyecto, la aplicación alcanzó un estado final funcional y estable.

Cabe destacar que las listas de funcionalidades y mejoras descritas en cada incremento no son exhaustivas, sino que recogen las implementaciones y cambios más relevantes realizados durante el desarrollo. A lo largo del proceso, se llevaron a cabo numerosas correcciones, ajustes e implementaciones adicionales que no se han detallado en su totalidad. No obstante, estas listas permiten ofrecer una visión general de los aspectos más significativos de cada incremento en el desarrollo del proyecto.

7.4. Gestión Segura de los Tokens

Antes de poder interactuar con la API de *Spotify*, es fundamental obtener y poder gestionar correctamente los tokens de autenticación que, como se ha descrito anteriormente en la sección de [Diseño de la Seguridad](#), se han decidido tomar ciertas medidas. En dicha sección, se abordó el tema desde una perspectiva de diseño y un nivel más alto de abstracción, mientras que en esta, se profundizará en los aspectos más técnicos.

En primer lugar, el *Client ID* y *Client Secret* se almacenan en el servidor, en respectivas variables de entorno, y solo se acceden desde el objeto `process.env`. Estos tokens nunca son enviados al cliente y se usan en casos concretos, como en el inicio de sesión de un nuevo usuario o la renovación del `access_token`.

Por otro lado, la gestión del propio `access_token` requiere más atención. Se evaluaron varias opciones, pero finalmente se decidió en hacer uso de las **cookies**. Esto se debe a que, de esta manera, no era necesario implementar un sistema de almacenamiento en el servidor (manual o mediante herramientas como *Redis*) para guardar los `access_token` de los diferentes usuarios y asociarlos a un identificador único. El propio `access_token` hace de identificador de cada usuario, por lo que el servidor solo tiene que hacer uso del token recibido mediante las cookies agregadas en la petición.

No obstante, es importante utilizar los sistemas de seguridad que las propias cookies ofrecen, para limitar cualquier tipo de uso malintencionado del token. Existen ciertas opciones (*flags*) de configuración que se han establecido:

- **httpOnly**: Impide que el token sea accesible mediante JavaScript en el navegador, protegiéndolo contra ataques de tipo *Cross-Site Scripting* (XSS).
- **secure**: Se asegura de que las cookies solo sean enviadas a través de conexiones **HTTPS**, protegiéndolas contra ataques de interceptación de tráfico (*Man-in-the-Middle*, MITM). Esta opción solo se habilita cuando la aplicación se ejecuta en un entorno de producción.
- **maxAge**: Define el tiempo de **expiración de las cookies**. El `access_token` tiene una validez de una hora, mientras que el `refresh_token` se mantiene activo durante 24 horas, permitiendo la renovación del token de acceso sin necesidad de solicitar credenciales nuevamente.

Estableciendo de esta manera el `access_token` y el `refresh_token` en las cookies y enviándolas en la respuesta al cliente, se mantiene la posibilidad de que cada usuario envíe su propio token como identificador, pero manteniendo la seguridad frente a ataques comunes. También se debe mencionar que, por el hecho de haber habilitado el flag `httpOnly`, **ningún componente de cliente puede realizar peticiones directamente desde el navegador**. Todas las peticiones de datos a la API de *Spotify* deben de ser realizadas a través del backend seguro.

Implicaciones de la Decisión del uso de las Cookies

El uso de las cookies como forma de almacenamiento del identificador también requiere el conocimiento del alcance efectivo de este sistema, que permite identificar posibles limitaciones que este método impone. Las cookies son compartidas entre todas las pestañas y ventanas del mismo navegador mientras se mantengan dentro del mismo dominio, lo que permite que la autenticación y la sesión del usuario se mantengan consistentes en diferentes contextos de navegación. Estas no se comparten entre distintos navegadores, ya que cada uno gestiona su propio almacenamiento de cookies.

Sin embargo, este sistema es **vulnerable si un atacante tiene acceso físico a la máquina** del usuario con la sesión abierta. A través del panel de desarrollador del navegador, el atacante podría visualizar directamente los valores de los tokens almacenados en las cookies. Aunque este método de almacenamiento impide el acceso mediante sistemas automatizados, un atacante con acceso manual al dispositivo podría extraer y utilizar estos valores.

Una posible medida de mitigación sería cifrar los tokens antes de almacenarlos en las cookies. No obstante, **se ha decidido no implementar esta solución** debido a varias razones: en primer lugar, por las limitaciones de alcance del proyecto; en segundo lugar, por la carga computacional adicional que supondría para el servidor el proceso de cifrado y descifrado en cada solicitud del usuario; y, finalmente, porque se considera que la probabilidad de que un atacante tenga acceso físico a una sesión abierta del usuario es extremadamente baja, lo que no justifica la implementación de este mecanismo en un sistema de estas características.

7.5. Implementación del Frontend

En los siguientes apartados, se describe el flujo de datos y la estructura utilizada para cargar y renderizar las estadísticas dentro de la aplicación, así como los aspectos más relevantes, como el paso de información entre componentes y la actualización de estados en componentes externos.

7.5.1. React Hooks

Para comprender mejor la implementación de los componentes, se considera importante hacer un inciso para introducir un concepto muy importante en *React*: los **hooks**. Estos son funciones especiales proporcionadas por la API de *React* que permiten a un componente funcional “engancharse” al estado y al ciclo de vida del mismo, sin necesidad de utilizar clases, y su uso está limitado a los componentes de cliente.

En el desarrollo de esta aplicación, se han empleado diversos *hooks*. A continuación, se describen los principales *hooks* utilizados y su aplicación en el proyecto:

- **useState**: Permite manejar el **estado local** dentro de los componentes. Es uno de los *hooks* más utilizados en desarrollos de *React*.
- **useEffect**: Se emplea para **gestionar efectos secundarios** dentro de los componentes, como la obtención de datos o la suscripción a eventos. En este caso, se ha utilizado para realizar llamadas a los *Route Handlers* del backend y poder obtener los datos en el cliente.
- **useRef**: Facilita el acceso a elementos del DOM y la **persistencia de valores entre renderizados** sin provocar re-renderizaciones innecesarias. Se ha utilizado en algunos componentes para manejar referencias a elementos gráficos, como un *canvas* o contenedores, permitiendo obtener sus dimensiones y ajustar los renderizados en consecuencia.

Además, se ha desarrollado un **hook personalizado** denominado **useFetch**, el cual encapsula la lógica de obtención de datos de la API dentro de un **useEffect**. Este *hook* es utilizado por los componentes de cliente para realizar peticiones de manera reutilizable. Se incluye el código detallado en el anexo, en el algoritmo C.2.

Cabe destacar que **Next.js** incorpora el modo de desarrollo **StrictMode**, el cual ejecuta los efectos definidos en **useEffect** dos veces consecutivas. Esto significa que, cuando un efecto se dispara, se ejecuta y se detiene inmediatamente para volver a ejecutarse desde cero. Este comportamiento permite detectar efectos mal programados y verificar si se están limpiando correctamente, evitando posibles errores que puedan aparecer en producción. Para gestionar correctamente este comportamiento, es necesario definir una **función de limpieza** dentro del **return** de **useEffect**. En este proyecto, la principal aplicación de este mecanismo ha sido la cancelación de las peticiones al backend. Para ello, se ha empleado la señal de aborto (**AbortController**) de la API **fetch**, la cual permite interrumpir una solicitud incluso si ya ha sido iniciada.

7.5.2. Estadísticas Básicas (Home)

Las estadísticas básicas se encuentran en la pantalla de inicio (*Home*) y se implementan como **componentes de servidor**. Dado que estos componentes no se ejecutan en el navegador, pueden realizar peticiones al backend obteniendo directamente el *access_token* desde las cookies, sin necesidad de gestionarlo manualmente en el cliente. Por otro lado, el selector de rango temporal (**TimeRangeSelector**) permite modificar la visualización de las estadísticas en función de tres intervalos: *short_term*, *medium_term* y *long_term*, siendo *short_term* la opción por defecto. Este selector es un **componente de cliente** que detecta cambios en su estado y, al modificarse, realiza una nueva petición al endpoint */home* con el *time_range* seleccionado como parámetro en la URL.

El componente **Home** recibe este parámetro y lo propaga a los componentes de estadísticas mediante **props** (ver código en el anexo, algoritmo C.3), propiedades (o parámetros) de los componentes. Cuando se detecta un cambio en los datos, los componentes vuelven a renderizarse en el servidor y se envían al frontend con un nuevo estado preprocesado.

Mientras la actualización se procesa, el usuario visualiza un fallback (*loading component*), garantizando una experiencia fluida. Gracias a este enfoque, únicamente los componentes afectados (los **TopTracks**, **TopArtists** y **TopGenres**) se actualizan, evitando recargas innecesarias. En la siguiente figura 7.4 se sintetiza esta interacción:

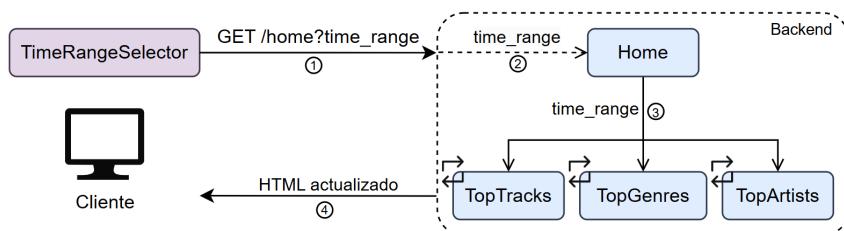


Figura 7.4: Diagrama explicativo de la actualización de las estadísticas básicas del *Home*.

El componente **RecentlyPlayed** funciona de manera diferente, ya que no se ve afectado por el selector de tiempo. Al ser un componente de cliente, no puede acceder a las cookies directamente. En su lugar, utiliza el `useFetch` para obtener los datos desde el backend. Este componente también gestiona su propio estado para alternar entre mostrar las últimas 10 o 50 canciones reproducidas.

7.5.3. Estadísticas Avanzadas (Stats)

Las estadísticas avanzadas se encuentran en la página `/stats`, la cual está compuesta completamente por **componentes de cliente**. Debido a su mayor complejidad y carga de procesamiento, estas estadísticas se **cargan dinámicamente**, evitando que se envíen todas al navegador desde el inicio y mejorando el rendimiento.

Paso de Información entre Componentes del Cliente

El componente **Stats** contiene la estructura principal de la página, incluyendo el **StatsGrid** (que muestra las tarjetas de estadísticas) y el **StatWrapper** (que gestiona la visualización de la estadística seleccionada). La gestión de los estados se centraliza en el componente padre **Stats**, mientras que, a través de `props`, se pasa a los componentes hijos las funciones definidas que habilitan la actualización de esos estados (ver código en el anexo, algoritmo C.4).

Cuando un **StatCard** es clicado, ejecuta dicha función y actualiza el estado `activeStat` con el valor del `statId` que contiene cada tarjeta y el estado `isModalOpen`, indicando que se abra la ventana modal. Al detectar un cambio en su estado, **Stats** se vuelve a renderizar, provocando, a su vez, el renderizado de sus componentes hijos. Ahora, **StatWrapper** obtiene el valor `activeStat` como un `prop` y puede cargar la estadística apropiada.

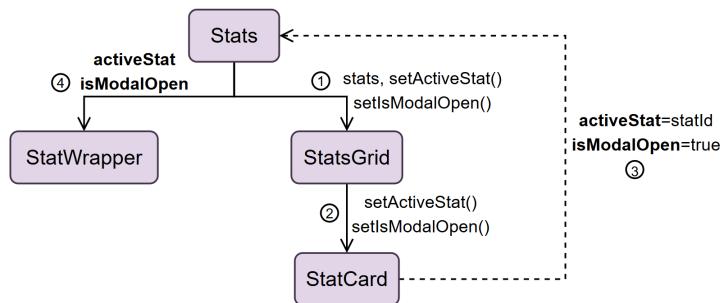


Figura 7.5: Diagrama explicativo de la actualización de la página *Stats* para detectar la selección de la estadística.

Como se puede ver en la figura 7.5, los componentes **StatCard** y **StatWrapper** no se comunican directamente, si no que lo hacen de manera indirecta a través del componente superior común **Stats**. Esto es un flujo de datos típico entre componentes de cliente de *React*.

Carga Dinámica de las Estadísticas

Además del `statId`, **StatWrapper** recibe, mediante **props**, el estado de visibilidad del modal (`isModalOpen`) y una función de cierre (`handleCloseModal()`) que permite resetear el estado al cerrar la ventana. Para simplificar la gestión de la ventana modal, se ha utilizado la librería **Radix UI**, la cual no afecta el proceso de selección y carga de las estadísticas.

Dentro del componente **StatWrapper** existe un objeto que asocia cada `statId` con su respectivo componente (algoritmo 7.2). Mediante la función **dynamic()** de *Next.js*, se realiza una carga dinámica en el momento en el que sea necesario renderizar el componente indicado. Cada vez que se selecciona una nueva estadística, se desmonta el componente anterior y se renderiza el nuevo en su lugar. También se elimina el componente cuando ya no sea necesario renderizarlo (al cerrar la ventana modal), por lo que el código relacionado con las estadísticas solo existe en el navegador el tiempo necesario, liberándolo de todo procesamiento innecesario.

Para más detalle del proceso de carga dinámico, se puede encontrar el código completo del componente **StatWrapper** en el anexo, algoritmo C.5.

```

1  const statComponents = {
2    "hall-of-fame": dynamic(() => import("@/components/stats/
3      ↪ HallOfFame")),
4    "estaciones-musicales": dynamic(() => import("@/components/stats/
5      ↪ /EstacionesMusicales")),
6    "huella-del-dia": dynamic(() => import("@/components/stats/
7      ↪ HuellaDelDia")),
8    "la-bitacora": dynamic(() => import("@/components/stats/
9      ↪ LaBitacora")),
10   "tus-decadas": dynamic(() => import("@/components/stats/
11     ↪ TusDecadas")),
12   "indice-de-interferencia": dynamic(() => import("@/components/
13     ↪ stats/IndiceDeInterferencia"))),
14 };
15
16 const DynamicComponent = statComponents[statId as keyof typeof
17   ↪ statComponents];

```

Algoritmo 7.2: Carga dinámica de componentes de estadísticas utilizando `dynamic` de Next.js.

A continuación se detallarán algunos aspectos concretos de la implementación de cada estadística. Para mantener la redacción compacta, no se va a dar una descripción exhaustiva de todas las funcionalidades, si no que se centrará solamente en aquellos puntos notables o partes que requieran una mención especial. Tampoco se explicarán los aspectos de la gestión de errores o de carga, ya que recaen más en el ámbito de las [Optimizaciones](#) y todas lo hacen de manera muy similar.

Para cada estadística, se ha usado el sistema de renderizado de gráficos más conveniente y óptimo para la situación. En la siguiente tabla 7.2 se recogen dichas librerías:

Estadística	Librería de Renderizado
Hall Of Fame	Canvas
Estaciones Musicales	SVG personalizado
Huella Del Día	Chart.js
La Bitácora	Chart.js
Tus Décadas	Konva
Índice De Interferencia	D3.js

Tabla 7.2: Librerías de renderizado seleccionadas para cada estadística avanzada.

i. Hall Of Fame

La estadística *Hall Of Fame* permite al usuario visualizar una cuadrícula con los álbumes más representativos de su historial de reproducción. Para su renderizado, se emplea el componente **Image** de *Next.js*, que optimiza la carga y presentación de las imágenes de manera automática. Además, se ha implementado un sistema de generación de imagen que permite capturar la cuadrícula generada en **canvas** mediante la librería **html2canvas**.

Antes de enviar la imagen al backend, la imagen se somete a un proceso de optimización (función `optimizeImage()` en el algoritmo C.8) para reducir su tamaño y que ocupe el límite máximo de datos a enviar (256 KB) con la máxima calidad posible bajo dicha restricción.

ii. Estaciones Musicales

La estadística *Estaciones Musicales* muestra la relación entre las estaciones del año y los hábitos de escucha del usuario mediante un gráfico de tipo “donut”, donde cada estación se representa con un color distintivo y un ícono característico.

La interfaz es interactiva y responde a los movimientos del cursor. Al pasar sobre una estación, el segmento correspondiente se resalta con una animación de expansión, y se despliega una tarjeta emergente con información sobre el artista y género más representativos de ese periodo. La posición del *tooltip* se ajusta dinámicamente en función de la ubicación del cursor. La creación de estos segmentos y la gestión de las animaciones se realiza de forma manual mediante SVGs personalizados con la función en el algoritmo C.9.

iii. Huella Del Día

La estadística *Huella del Día* muestra la distribución de los minutos de escucha a lo largo del día mediante un gráfico de líneas, permitiendo al usuario identificar sus horas de mayor consumo musical. Para su implementación, la librería `react-chartjs-2` ayuda con la integración de `chart.js` en un entorno *React*.

El gráfico cuenta con dos elementos:

- **Línea principal:** Representa la cantidad de minutos de reproducción en cada hora del día.
- **Punto destacado:** Resalta la hora con mayor actividad, facilitando su identificación visual.

Además, se incluye un mensaje dinámico que indica al usuario su hora de mayor escucha, resaltando este dato con un color distintivo para mejorar su visibilidad.

iv. La Bitácora

La estadística *La Bitácora* permite visualizar la evolución de las canciones guardadas por el usuario a lo largo del tiempo mediante un gráfico de barras interactivo que permite navegar entre distintos niveles de granularidad: **años, meses y días**.

Para su implementación, se ha utilizado la biblioteca `chart.js` integrada también con `react-chartjs-2`.

v. Tus Décadas

La estadística *Tus Décadas* permite visualizar la distribución de las canciones guardadas por el usuario a lo largo de diferentes décadas mediante una línea de tiempo interactiva que organiza las portadas de los álbumes por año, agrupándolos en bloques de diez años. Al igual que la estadística anterior, para integrar la biblioteca `Konva` en un entorno *React*, se ha utilizado la biblioteca `react-konva`.

El sistema incorpora las siguientes opciones de interacción con la gráfica:

- **Navegación mediante desplazamiento:** Permite moverse horizontal y verticalmente a lo largo de la línea de tiempo para explorar los álbumes de cada década. Esto se gestiona con el componente Stage de Konva, como se muestra en el algoritmo C.10
- **Zoom dinámico:** Se ha implementado un sistema de *zoom* con la rueda del ratón, facilitando la exploración de los datos en distintos niveles de detalle. La función encargada se puede encontrar en el algoritmo C.11.

vi. Índice De Interferencia

La estadística *Índice de Interferencia*, la más abstracta de todas, representa la diferencia entre los hábitos de escucha del usuario y su comportamiento actual mediante una visualización basada en ondas sinusoidales. Los datos de base corresponden a la **media de la popularidad** de las canciones que el usuario suele escuchar, determinada a partir de su biblioteca de canciones guardadas, y la popularidad de lo que está escuchando en el momento, calculada a partir de sus últimas 50 reproducciones. Estos valores de popularidad, que son numéricos, se traducen a la frecuencia de cada onda representada en la visualización.

Para su implementación, se ha utilizado la biblioteca D3.js, más capaz que chart.js, y que facilita la generación de gráficos SVG optimizados. Los datos de frecuencia se obtienen desde el backend y se interpolan (algoritmo C.12) para construir las ondas representadas en la visualización (algoritmo C.13).

El sistema permite visualizar las siguientes ondas:

- **Frecuencia habitual:** Representa el patrón de escucha estándar del usuario.
- **Frecuencia del momento:** Refleja la frecuencia de escucha actual.
- **Onda combinada:** Muestra la interferencia entre ambas frecuencias, resaltando las diferencias entre el comportamiento habitual y el actual.

El cálculo de la interferencia entre las dos ondas originales se hace mediante la suma los respectivos puntos en cada una. Esto se realiza con la siguiente función sencilla que se muestra en el algoritmo 7.3, que genera los datos para la función de dibujado de onda:

```
1 const combinedData = normalData.map((d, i) => ({  
2     x: d.x,  
3     y: d.y + actualData[i].y,  
4 }) );
```

Algoritmo 7.3: Cálculo de la onda combinada en *Índice de Interferencia* sumando las ondas de la frecuencia habitual y la frecuencia actual.

7.6. Implementación del Backend

El backend de la aplicación se encarga de gestionar la comunicación con la API de *Spotify*, procesar los datos recibidos y servir la información al frontend de manera eficiente. En esta sección, se describen algunas implementaciones destacadas dentro del backend, enfocándose en aquellos endpoints que presentan particularidades interesantes o que han requerido una solución específica. El resto de la implementación detallada se encuentra en el [Anexo D](#), donde se incluyen todas las estructuras de datos definidas para el fácil consumo del frontend y lógica aplicada (en forma de pseudocódigo, para facilitar la compresión) en el procesamiento de estadísticas.

7.6.1. Implementaciones Destacadas

A continuación, se explican dos de las implementaciones más relevantes dentro del backend:

Estaciones Musicales

Como ya se ha mencionado en el [Estudio de la API de Spotify](#), la API no permite acceder al historial completo de reproducciones pasadas de un usuario, por lo que utilizar los elementos “destacados” de cada estación, se realizan en base a las canciones favoritas del el usuario.

Primero se obtiene la lista de todas las canciones guardadas en el último año. A cada una de estas canciones se le asigna una estación específica en función de su fecha de guardado. Posteriormente, se contabiliza cuántas veces aparece cada artista dentro de cada estación. **El artista con mayor frecuencia en una estación es considerado el artista destacado de ese período.**

El proceso de selección del género representativo de cada estación sigue una lógica similar, pero con una particularidad adicional: los géneros están asociados a los artistas y no a las canciones individualmente. Para obtener esta información, se recopilan los artistas correspondientes a cada estación y se consultan sus géneros mediante la API de *Spotify*. Como un mismo artista puede pertenecer a varios géneros, se agrupan todos los géneros de todos los artistas de la estación y se contabilizan sus apariciones. Finalmente, **el género con más repeticiones se considera el género representativo de esa estación**. Es importante notar que el artista más destacado y el género más representativo de una estación no tienen por qué coincidir, ya que cada uno se obtiene mediante un proceso de conteo independiente.

Una vez determinado el artista y el género, se estructura la información en un objeto JSON con los datos completamente procesados. Este formato permite que el frontend reciba los datos de manera limpia y lista para su visualización, sin necesidad de realizar cálculos adicionales.

Índice de Interferencia

El propósito de esta estadística es representar dos valores que reflejan los hábitos de escucha del usuario en diferentes escalas de tiempo: un valor “normal”, que describe sus preferencias musicales generales, y un valor “actual”, que representa lo que está escuchando en un periodo corto de tiempo reciente. La comparación entre estos valores permite observar si el usuario está escuchando música en línea con sus hábitos o si ha cambiado significativamente sus preferencias.

Para obtener el valor “actual”, se utiliza la lista de las últimas 50 canciones reproducidas por el usuario, ya que *Spotify* permite acceder a este historial reciente. Cada canción tiene un valor de popularidad entre 0 y 100, donde 100 representa la música más popular en la plataforma. Se calcula la media de la popularidad de estas canciones para obtener un indicador del nivel de popularidad de la música que el usuario ha estado escuchando recientemente.

Para determinar el valor “normal”, se usan como referencia las canciones guardadas en la biblioteca del usuario, asumiendo que reflejan de manera aproximada sus preferencias a lo largo del tiempo. Sin embargo, en usuarios con muchas canciones guardadas, calcular la media de popularidad sobre toda la colección sería un proceso costoso en términos de tiempo y consumo de API. Para optimizar este cálculo, **se emplea una estrategia de muestreo**: si el usuario tiene más de 500 canciones guardadas, en lugar de recorrer toda la biblioteca, se extrae una muestra del 20 % de las canciones, con un mínimo de 500 y un máximo de 1000 canciones. Este muestreo se realiza seleccionando canciones desde posiciones aleatorias dentro de la biblioteca, aprovechando que *Spotify* permite acceder a cualquier parte de la colección directamente mediante un índice de desplazamiento. Este método proporciona una estimación muy cercana al valor real sin necesidad de procesar todas las canciones guardadas.

Una vez obtenidos ambos valores, se estructuran en un objeto JSON con dos números simples que se envían al frontend. Mediante esta optimización, se ha podido ahorrar mucho tiempo de procesamiento y se obtiene un resultado muy parecido al que sería con todas las canciones.

7.6.2. Resto de Endpoints

El resto de los endpoints se encuentran en el [Anexo D](#), donde se puede consultar el pseudocódigo de todas las estadísticas, incluyendo las dos explicadas anteriormente. Allí se detallan los pasos de la lógica de cada endpoint, por si se quiere revisar con más precisión cómo están implementados.

- **Hall Of Fame:** [D.1](#)
- **Estaciones Musicales:** [D.2](#)
- **Huella Del Día:** [D.3](#)
- **La Bitácora:** [D.4](#)
- **Tus Décadas:** [D.5](#)
- **Índice De Interferencia:** [D.6](#)

7.7. Middleware

El *middleware* en *Next.js* permite interceptar y modificar las solicitudes HTTP antes de que lleguen a las rutas del *App Router* o los endpoints de los *Route Handlers*. En esta aplicación, se usa como un sistema de separación entre las rutas para usuarios no autenticados (/), la raíz) y las rutas protegidas tras la autenticación y autorización (/home y /stats). Su código completo se encuentra disponible en el anexo, en el algoritmo C.6.

7.7.1. Funcionamiento del Middleware

El *middleware* se aplica de manera global a las rutas /home, /stats y sus subrutas, ejecutándose antes de que las peticiones alcancen el backend. Su función principal es **verificar si el usuario está autenticado y autorizado** para acceder a estas secciones protegidas. Para ello, realiza una comprobación de la existencia de las cookies de autenticación: el access_token y el refresh_token. Si ambas están presentes, la solicitud se considera válida y se permite el acceso a la página correspondiente.

Dado que el access_token tiene una validez limitada de una hora, el *middleware* también se encarga de **gestionar su renovación automática** cuando expira. En caso de no detectar un access_token pero sí un refresh_token válido, se invoca la función renovarAccessToken() (algoritmo C.7), la cual interactúa con la API de Spotify utilizando las credenciales de la aplicación (*Client ID* y *Client Secret*) para solicitar un nuevo token de acceso. Como el *middleware* se ejecuta en el servidor, el acceso a estas credenciales es seguro, ya que nunca son expuestas en el cliente. Esta estrategia permite extender la sesión del usuario sin requerir una nueva autenticación manual, mejorando la experiencia de usuario.

En la siguiente tabla 7.3 se han resumido todas las posibilidades y las acciones respectivas que el *middleware* llevaría a cabo:

access_token	refresh_token	Acción	
No	No	Denegar acceso	→ Redirigir a /
No	Sí	Renovar access_token	→ Continuar petición
Sí	No	Permitir acceso	
Sí	Sí	Permitir acceso	

Tabla 7.3: Acciones del *middleware* en todos los casos posibles de existencia de tokens.

7.7.2. Limitaciones y Consideraciones de Seguridad

Si bien el sistema de autenticación definido es funcional y práctico para este tipo de aplicación, existe una vulnerabilidad potencial: el *middleware* solo verifica la existencia de los tokens, **pero no su validez**. Esto significa que un usuario malintencionado podría crear manualmente cookies con los nombres access_token y refresh_token y valores aleatorios, lo que le permitiría acceder a la interfaz protegida. No obstante, cualquier intento de obtener datos desde la API de Spotify fallaría, ya que los tokens falsificados no serían reconocidos por el servicio.

Para mitigar esta vulnerabilidad, se podría implementar una validación adicional enviando una solicitud a la API desde el *middleware* para comprobar si el token es válido antes de conceder acceso. Sin embargo, esta solución implicaría una llamada extra a *Spotify* en cada solicitud, lo que aumentaría la carga en la API y afectaría el tiempo de carga de la aplicación. Dado que la probabilidad de explotación de esta vulnerabilidad es baja en el contexto de este proyecto, **se ha decidido no implementar esta verificación adicional** para evitar un consumo innecesario de recursos.

7.8. Optimizaciones

Como sección final del capítulo de implementación, se presentan algunas de las optimizaciones más relevantes realizadas en la última iteración del desarrollo del producto. Estas mejoras han permitido optimizar el rendimiento del sistema, mejorar la experiencia del usuario y reducir el consumo de recursos, tanto en el backend como en el frontend.

7.8.1. Peticiones a Spotify y Procesado de Datos

El acceso a la API de *Spotify* es un aspecto crítico dentro del funcionamiento de la aplicación, ya que la cantidad de datos necesarios para generar cada estadística puede ser considerable. Con el objetivo de reducir la carga en la API y mejorar la eficiencia del sistema, se han aplicado distintas estrategias.

Para aquellos casos en los que se requiere obtener una gran cantidad de información, como en *La Bitácora*, *Tus Décadas* o *Índice De Interferencia*, se ha implementado un **sistema de paginación** que permite recuperar los datos en lotes, evitando bloqueos y reduciendo el tiempo de espera. Además, en el caso de *La Bitácora*, se ha implementado un sistema de **caché en memoria** que almacena los datos agregados, lo que evita que se realicen cálculos repetitivos y mejora significativamente la velocidad de respuesta del servidor.

Por otro lado, en *Índice de Interferencia*, donde el objetivo es comparar la frecuencia de escucha habitual del usuario con su comportamiento actual, se decidió emplear un **muestreo de datos** en lugar de analizar toda la biblioteca musical al completo. Al detectar una cantidad mayor de 500 canciones guardadas, el sistema toma una muestra representativa del 20 % del total, que permite calcular una buena aproximación sin comprometer el rendimiento.

A nivel general de todas las estadísticas, también se ha optimizado el filtrado de datos antes de enviarlos al frontend, eliminando aquellos valores innecesarios y asegurando que las respuestas sean lo más ligeras posible.

7.8.2. Interfaz y Experiencia de Usuario

Desde el punto de vista del frontend, una de las principales mejoras implementadas ha sido la carga dinámica de componentes ya mencionada en la sección [Carga Dinámica de las Estadísticas](#), así como el uso del componente `next/image` para mejorar la carga y optimización de las portadas de álbumes sin afectar la calidad visual.

Para mejorar la experiencia del usuario, se han incorporado *loaders* progresivos en aquellas estadísticas que requieren la recuperación de grandes volúmenes de datos. A través del uso del entorno `Suspense`, se garantiza que la interfaz no quede congelada mientras se procesan las solicitudes. Adicionalmente, en estadísticas que incluyen gráficos

interactivos, como *Índice de Interferencia* y *Estaciones Musicales*, se han añadido animaciones y transiciones suaves, mejorando la fluidez de las interacciones sin afectar el rendimiento.

7.8.3. Optimización en Estadísticas Específicas

Cada estadística ha sido optimizada con enfoques específicos para mejorar su funcionalidad y eficiencia. De entre todas ellas, se mencionan las siguientes:

La Bitácora

En el caso de *La Bitácora*, una de las principales mejoras ha sido la ordenación cronológica de los datos, asegurando que los valores se presenten en el orden adecuado para facilitar su interpretación. Para mejorar la legibilidad de la información, también se han personalizado las etiquetas del gráfico, reemplazando los formatos numéricos estándar por nombres de meses y días más intuitivos. Además, se ha eliminado el periodo de carga entre clics en la navegación entre distintos niveles de granularidad, permitiendo una transición fluida entre las vistas de años, meses y días.

Otra mejora destacable ha sido el cálculo dinámico de porcentajes en los *tooltips* del gráfico de barras, lo que proporciona al usuario información detallada sobre la proporción que representa cada período respecto al total.

Índice de Interferencia

En el caso de *Índice de Interferencia*, se han aplicado varias mejoras para hacer que la visualización de las ondas sinusoidales sea más intuitiva y atractiva, ya que se trata de la estadística más experimental:

- Transiciones progresivas en la generación de las ondas, asegurando que estas aparezcan de manera fluida en la pantalla.
- Efectos de resaltado dinámico mediante filtros de visuales, para que el usuario pueda identificar claramente las diferentes ondas.
- Posibilidad de alternar entre la visualización de ondas individuales y la onda combinada, facilitando el análisis de la diferencia entre ambas frecuencias.

Tus Décadas

En *Tus Décadas*, se ha mejorado la navegación y visualización de los álbumes organizados por año. Siendo la estadística que más recursos demanda del cliente, el cambio de una implementación manual al uso de la librería *Konva* ha sido la pieza clave para que fuese funcional. *Konva* permite dividir los elementos gráficos en capas (*layers*), asegurando que solo se rendericen aquellos elementos que están en pantalla.

También se ha mejorado la experiencia de exploración al añadir soporte para zoom dinámico y desplazamiento, permitiendo que el usuario pueda navegar libremente entre las diferentes décadas y explorar su colección de álbumes de manera intuitiva. En iteraciones anteriores, esta funcionalidad también se había implementado de manera manual, que daba resultado a una experiencia más limitante y muy lenta.

7.8.4. Impacto General de las Optimizaciones

Gracias a la última fase de optimizaciones, la plataforma ha logrado reducir significativamente los tiempos de carga y mejorar la fluidez de la interacción del usuario con las estadísticas. La reducción en el número de peticiones innecesarias a la API de Spotify ha permitido disminuir la latencia y evitar sobrecargas en el servidor. Del mismo modo, la optimización en la renderización de gráficos y componentes ha permitido que la interfaz sea más ágil y responsive, incluso en dispositivos con menor capacidad de procesamiento. En conjunto, **todas estas mejoras han contribuido a ofrecer una experiencia de usuario más rápida, intuitiva y eficiente.**

Pruebas

8.1. Implementación de Pruebas

Para garantizar el correcto funcionamiento de la aplicación, durante el desarrollo se ha realizado una fase de pruebas. Principalmente se han implementado pruebas unitarias, pero también algunas pruebas de integración para verificar el comportamiento de ciertos componentes y su interacción con los componentes vecinos.

8.1.1. Framework y Librerías

Para la ejecución de los tests se ha utilizado **Jest** como framework principal, junto con **React Testing Library** para la prueba de componentes de *React*. *Jest* permite ejecutar tests de forma rápida y aislar cada prueba en un entorno controlado, mientras que *React Testing Library* facilita la interacción y consulta de elementos en el DOM, asegurando que los tests reflejen mejor el comportamiento real de los usuarios.

En el caso de los endpoints del backend, *Next.js* no proporciona un mecanismo directo para testear las API definidas con *Route Handlers*, por lo que ha sido necesario el uso de la librería **next-test-api-route-handler**. Esta herramienta permite simular peticiones HTTP a los endpoints del backend y verificar las respuestas esperadas, facilitando la validación del comportamiento de la lógica del servidor.

8.1.2. Consideraciones a Tener en Cuenta

A la hora de escribir los tests, se han tenido en cuenta diferentes aspectos para asegurar su correcta ejecución:

- **Entorno de ejecución:** *Jest* permite definir diferentes entornos de ejecución, siendo los más relevantes *jsdom*, que simula un entorno de navegador, y *node*, que representa el entorno de servidor. En este proyecto, **se han utilizado ambos entornos** según corresponda, ya que existen tanto componentes de cliente como de servidor. Es importante indicar explícitamente el entorno en la configuración de *Jest* para evitar errores en la ejecución de los tests.

- **Configuración de variables de entorno:** Algunas pruebas requieren acceso a variables. Para ello, se ha utilizado el archivo `.env.test`, donde se definen las variables necesarias en el contexto de pruebas.
- **Cobertura del código:** Se ha utilizado la funcionalidad de cobertura de `Jest` para analizar qué partes del código están siendo verificadas por las pruebas y detectar posibles áreas sin testear.
- **Limitaciones y problemas encontrados:** En algunos casos, no ha sido posible realizar ciertas pruebas debido a problemas con dependencias específicas. Por ejemplo, `Jest` presenta dificultades al manejar módulos basados en `CommonJS` y en la manipulación de canvas en entornos de servidor, lo que ha impedido testear correctamente ciertos gráficos generados con `D3.js`. Se intentó solucionar esto mediante *mocking* de dependencias, pero sin éxito.

Además de la ejecución local de las pruebas, se ha configurado un flujo de integración continua (CI) para ejecutar los tests automáticamente en cada nueva actualización del código. Esto se ha realizado mediante **GitHub Actions**, asegurando que cualquier cambio en el proyecto sea validado antes de desplegarse. Se explican más detalles sobre esta integración en el capítulo de **Despliegue**, la sección [Ejecución Automática de Tests](#).

8.2. Resultados de Pruebas

Despliegue

9.1. Vercel

El despliegue de la aplicación se realiza utilizando *Vercel*, una plataforma de hosting orientada a aplicaciones frontend y serverless. La principal razón por la que se ha seleccionado esta plataforma, de entre las muchas opciones, es porque *Vercel* es la empresa desarrolladora del framework *Next.js*. Con esto se asegura una integración completa y se simplifica significativamente el proceso de despliegue.

También ha sido importante el hecho de que la infraestructura de *Vercel* se apoya en servicios como *Amazon Web Services* y *Cloudflare* [3], entidades muy conocidas y confiadas, que proporcionan características de seguridad, disponibilidad y alto rendimiento. Sobre ellas, *Vercel* ofrece una interfaz muy cómoda para manejar los despliegues, además de un sistema de despliegue continuo muy accesible.

9.2. Límites y Características

Para este proyecto, el plan gratuito **Hobby** de *Vercel* ofrece unas características suficientes, ya que está dirigido a proyectos personales o de pequeña escala. En la tabla 9.1, se listan los principales límites de los recursos disponibles:

Categoría	Descripción	
Computación y Red	Solicitudes HTTP	1 millón
	Transf. de datos al usuario	100 GB
	Transf. de datos internos	10 GB
	Ejecuciones de middleware	1 millón
	Funciones serverless	1000.000 ejecuciones (APIs, procesar datos, etc.)
	Optimización de imágenes	1.000 imágenes
Seguridad	Firewall	Hasta 3 reglas personalizadas
	IP Blocking	Hasta 3 reglas para bloquear IPs
	Mitigación de DoS	Protección integrada contra ataques DoS/DDoS
	SSL/HTTPS	Certificados SSL incluidos para todas las aplicaciones

Tabla 9.1: Límites y características por mes del plan *Hobby* de *Vercel*.

9.3. Proceso de Despliegue

El despliegue de la aplicación en *Vercel* es muy sencillo, ya que la plataforma permite conectar un repositorio *GitHub* para crear un nuevo proyecto y cargar todo el código. Además, al tratarse de un proyecto de *Next.js*, la plataforma se encarga de establecer la configuración óptima para el build (figura 9.1).

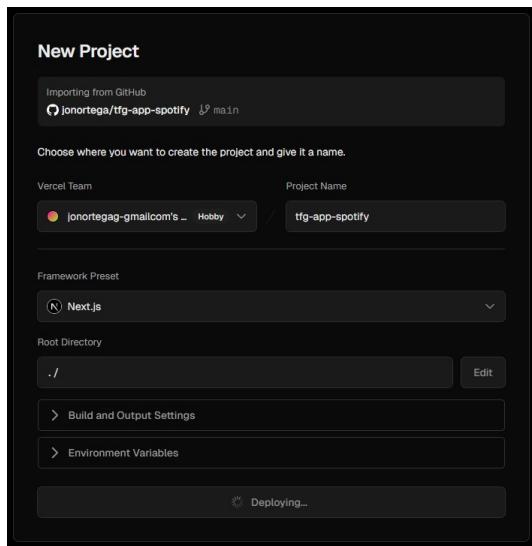


Figura 9.1: Pantalla de creación del proyecto tras conectarlo con el repositorio de *GitHub*.

Tras esperar a que se realice el buid correctamente, nuestra aplicación se encontrará en línea, con certificados SSL y con un dominio personal que *Vercel* establecerá según el nombre del proyecto¹. Existen dos diferentes entornos de despliegue:

- *Production*: Entorno principal, accesible para los usuarios finales desde el dominio establecido en la creación del proyecto. Se toma la rama `main` como la rama de producción.
- *Preview*: Entorno de preproducción que se genera automáticamente al realizar cambios en cualquier rama distinta a la de producción (`main`). Se asigna un dominio único diferente a cada despliegue.

En nuestro caso, solo haremos uso del entorno *production*. Cualquier cambio realizado en la rama `main` será detectado automáticamente por *Vercel*, que creará un nuevo despliegue, realizará un nuevo build y actualizará la página en línea. En caso de que ocurra algún error, se revertirá automáticamente a la versión anterior del proyecto. Este proceso se realiza sin intervención manual, por lo que obtenemos un despliegue continuo (CD) sin necesidad de configuraciones adicionales.

¹Dominio del proyecto de este TFG: <https://tfg-app-spotify.vercel.app>

9.3.1. Configuración de Variables de Entorno

Para finalizar con la configuración del despliegue, es necesario agregar las variables de entorno en el panel de control de *Vercel*. Cada entorno permite establecer variables específicas, pero en nuestro caso solo introduciremos, en el entorno de producción, las siguientes:

Variable	Descripción
DOMAIN_URL	URL principal de la aplicación, establecida en la creación del proyecto.
SPOTIFY_CLIENT_ID	Identificador único de la aplicación, registrado en la plataforma de desarrollo de Spotify .
SPOTIFY_CLIENT_SECRET	Clave secreta asociada con el SPOTIFY_CLIENT_ID.
SPOTIFY_REDIRECT_URI	La URL a la que Spotify redirigirá al usuario después de que complete el proceso de autenticación. También está registrada en la plataforma de desarrollo de Spotify .
NEXTAUTH_URL	La URL base de la aplicación que utiliza NextAuth para manejar la autenticación. Necesaria en todas las aplicaciones de Next.js .

Tabla 9.2: Variables de entorno necesarias en producción.

Es importante destacar que, para garantizar una transición fluida entre el entorno de desarrollo local y el de producción, se debe evitar codificar directamente las URLs en el código. En su lugar, se debe utilizar la variable de entorno DOMAIN_URL, lo que permite que las URLs cambien automáticamente de localhost al dominio público correspondiente.

9.3.2. Análisis y Monitoreo

Vercel ofrece un panel de control desde donde se pueden visualizar diversas métricas y modificar opciones de configuración. Para este proyecto, es importante monitorizar los indicadores de recursos consumidos (figura 9.2), para asegurar que no llegan al límite del plan gratuito.

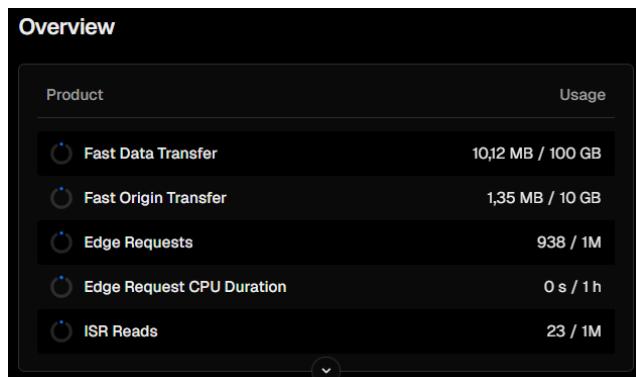


Figura 9.2: Panel con la visión general de los recursos consumidos.

Esto es especialmente relevante porque, en las plataformas de nube, se ha popularizado un ataque conocido como **Ataque de Agotamiento Económico** (*Denial of Wallet* (DoW)) [4]. Son un tipo de ataque DoS, cuyo objetivo es consumir continuamente los recursos más “pesados”, con el fin de obligar el escalado automático y agotar el presupuesto operativo de la aplicación. En el caso de este proyecto, al encontrarse dentro del plan gratuito, la consecuencia sería el consumo completo de los recursos disponibles; haciendo que *Vercel* retirase la aplicación de producción, quedándose inoperativa hasta el siguiente ciclo mensual.

Por esta razón, *Vercel* y otras plataformas similares ofrecen mecanismos de protección contra ataques DoS, permitiendo bloquear temporalmente este tipo de peticiones con el firewall integrado. Además, podemos acceder a ver los logs generados en el servidor durante la última hora y monitorizar el número de invocaciones, datos transferidos y peticiones por cada ruta de nuestra web.

9.4. Ejecución Automática de Tests

Añadir al hacer el apartado de Pruebas.

CAPÍTULO **10**

Seguimiento y Control

10.1. Metodología de Seguimiento

10.2. Incidencias

10.2.1. Control de Riesgos

10.2.2. Control de Alcance

10.2.3. Control de Planificación

10.3. Revisión de Objetivos

CAPÍTULO **11**

Conclusiones

11.1. Conclusiones Técnicas

11.2. Conclusiones Personales

11.3. Líneas Futuras

Anexos

Capturas de la Interfaz de Usuario

A.1. Interfaz de Estadísticas

Cada una de las estadísticas ha sido diseñada de manera independiente y autocontenida. Es por esto por lo que, aunque se sigan las directrices de diseño generales de la página, cada una tiene un carácter y presentación diferenciable. También cabe destacar que, en algunas de ellas, se ha preferido desarrollar la implementación funcional de manera más robusta, antes de invertir el tiempo en mejorar por completo la estética de la presentación.

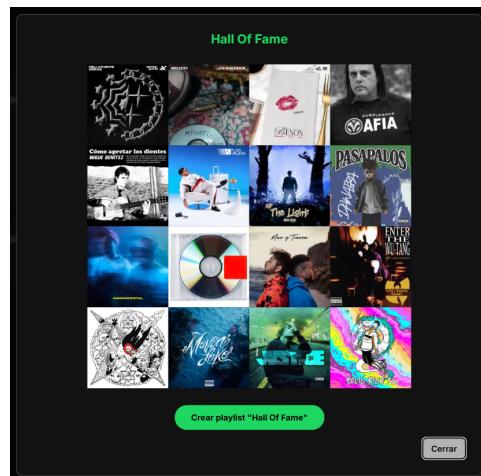


Figura A.1: Interfaz de la estadística *Hall Of Fame*.



Figura A.5: Interfaz de la estadística *La Bitácora*, en los tres diferentes niveles de detalle.

A.1. Interfaz de Estadísticas

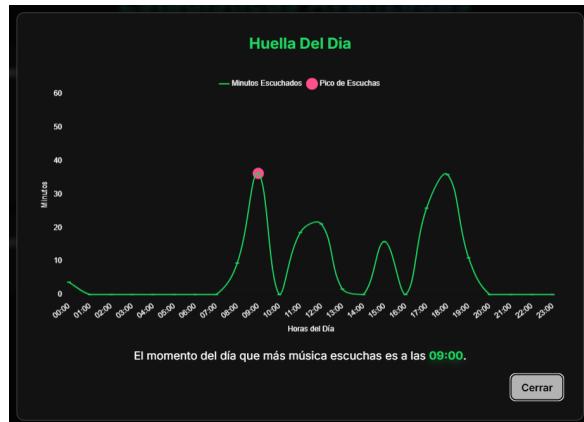


Figura A.6: Interfaz de la estadística *Huella Del Día*.

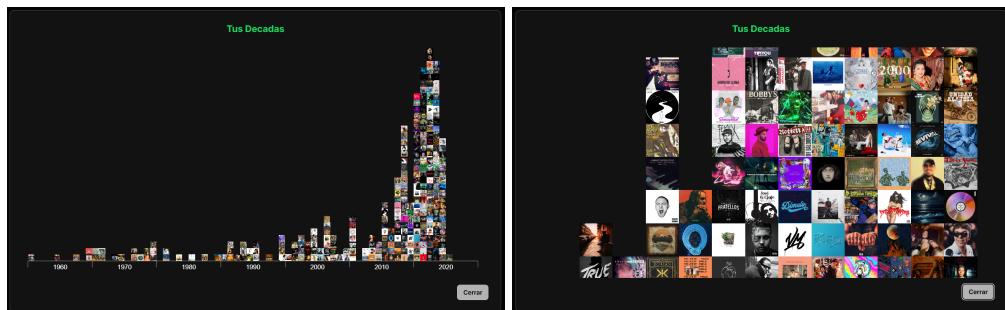


Figura A.9: Interfaz de la estadística *Tus Décadas*, en diferentes niveles de zoom.

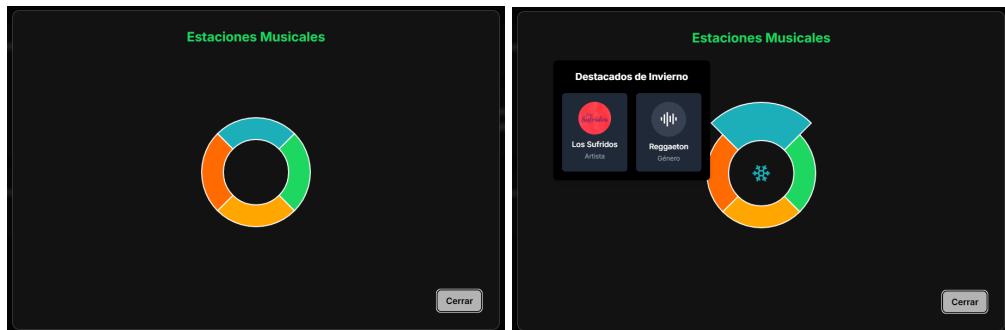


Figura A.12: Interfaz de la estadística *Estaciones Musicales*, en los estados de cerrado y abierto.



Figura A.15: Interfaz de la estadística *Índice De Interferencia*, con las ondas originales y su forma combinada.

A.2. Componentes Secundarios, de Carga y de Errores

Además de los elementos presentados, también se han implementado algunos componentes secundarios para aportar más interactividad a la web y mejorar la experiencia al usuario.

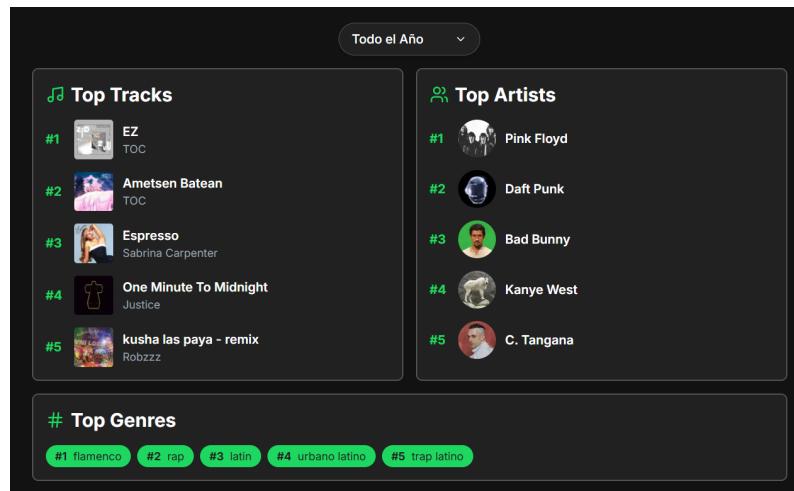


Figura A.16: Detalle de los tres *tops* junto con el selector de periodo de tiempo.

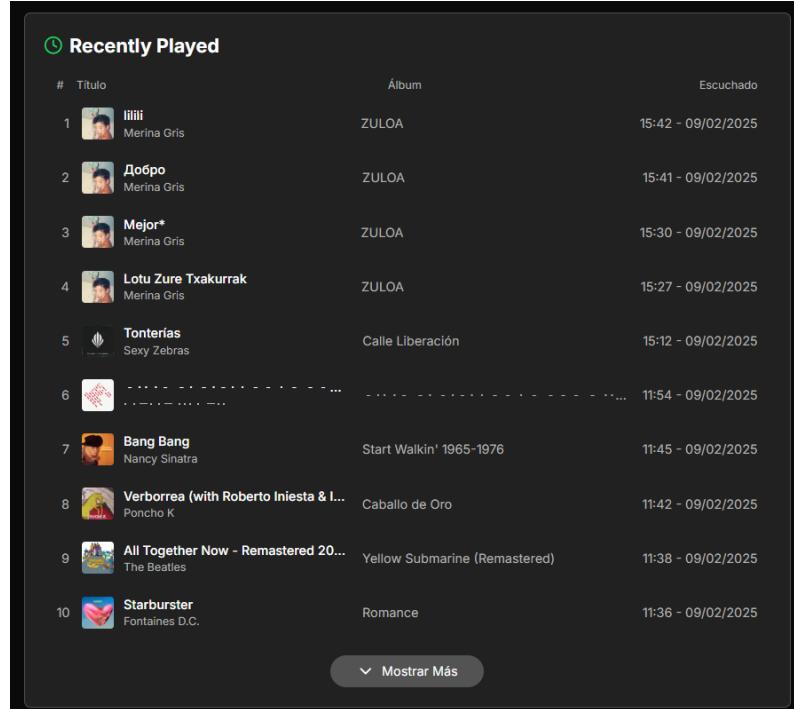


Figura A.17: Detalle de *Recently Played* con el botón para ampliar o contraer la lista.

A.2. Componentes Secundarios, de Carga y de Errores

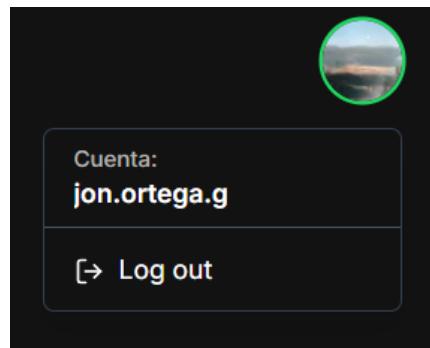


Figura A.18: Panel del usuario con la opción de *Log Out*.

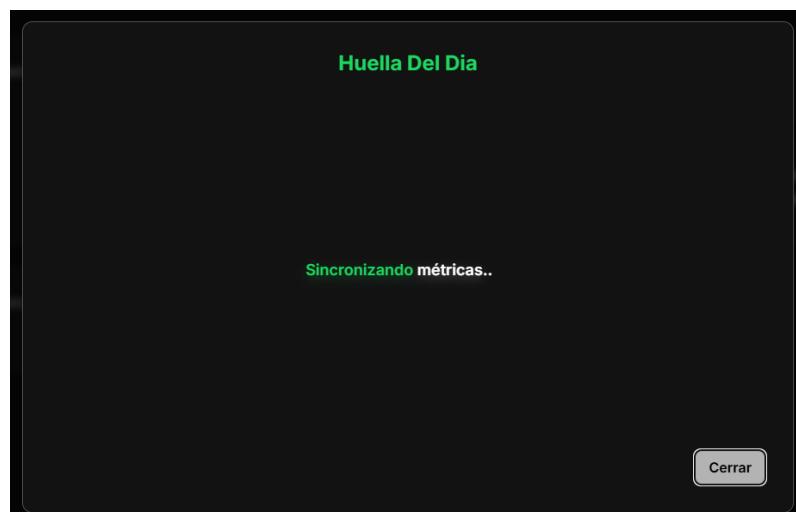


Figura A.19: Componente que se muestra mientras se cargan los datos de las estadísticas, con texto dinámico.

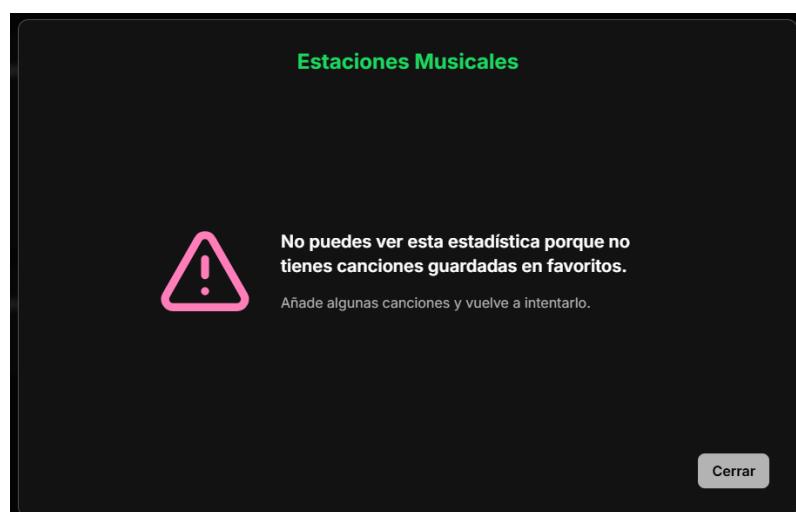


Figura A.20: Componente de error que se muestra cuando el usuario no tiene ninguna canción guardada en su lista de favoritos.

A.2. Componentes Secundarios, de Carga y de Errores

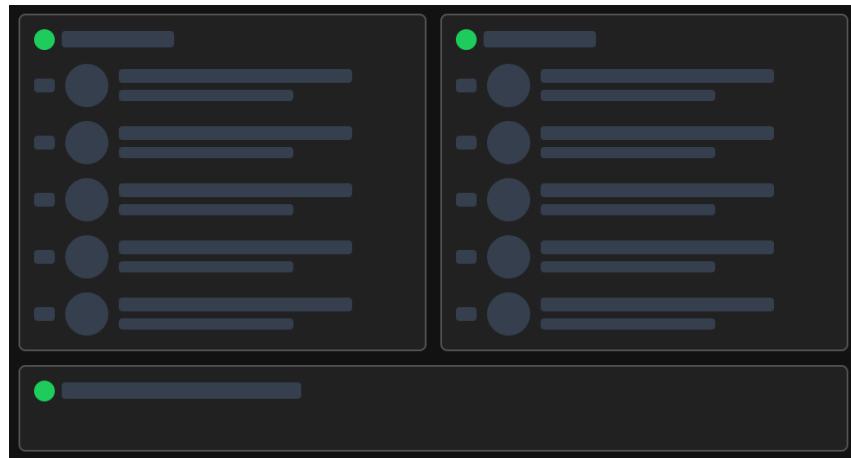


Figura A.21: Componentes de *loading* para los tres *tops*.

Diagramas de Secuencia Adicionales

- **Cerrar Sesión:** figura B.1
- **Ver Huella Del Día:** figura B.2
- **Ver Estaciones Musicales:** figura B.3
- **Ver Tus Décadas:** figura B.4
- **Ver Índice de Interferencia:** figura B.5

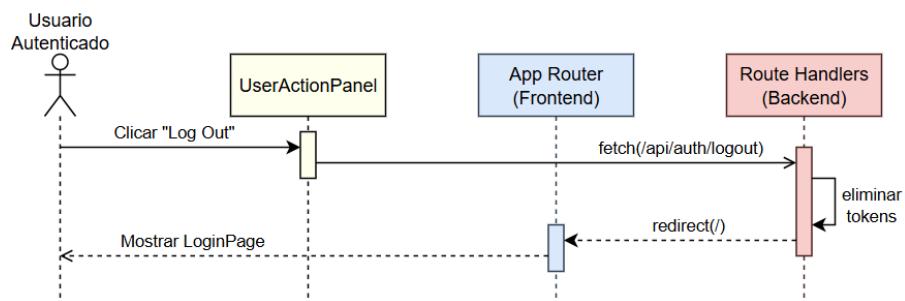


Figura B.1: Diagrama de secuencia: **Cerrar Sesión**.

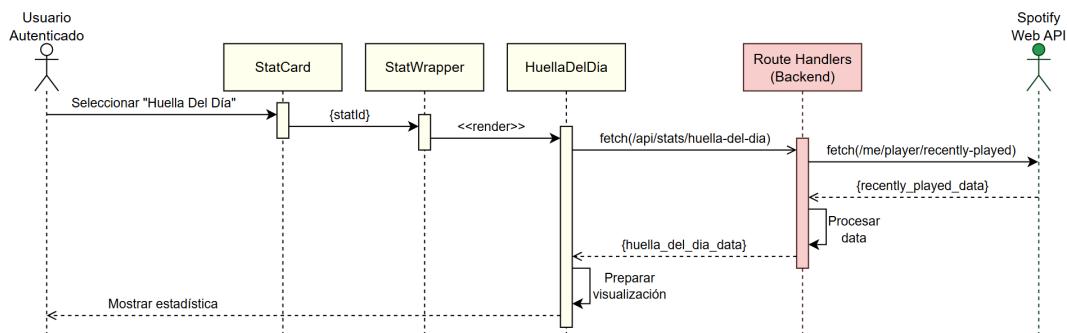


Figura B.2: Diagrama de secuencia: **Ver Huella Del Día**.

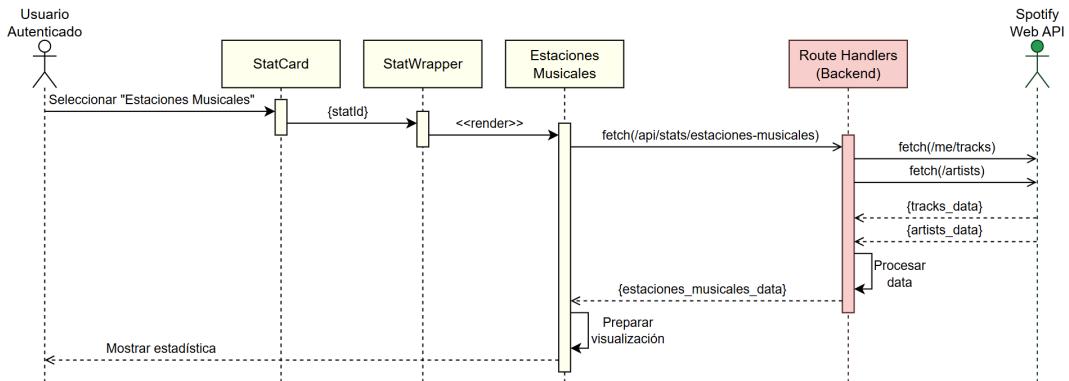


Figura B.3: Diagrama de secuencia: Ver Estaciones Musicales.

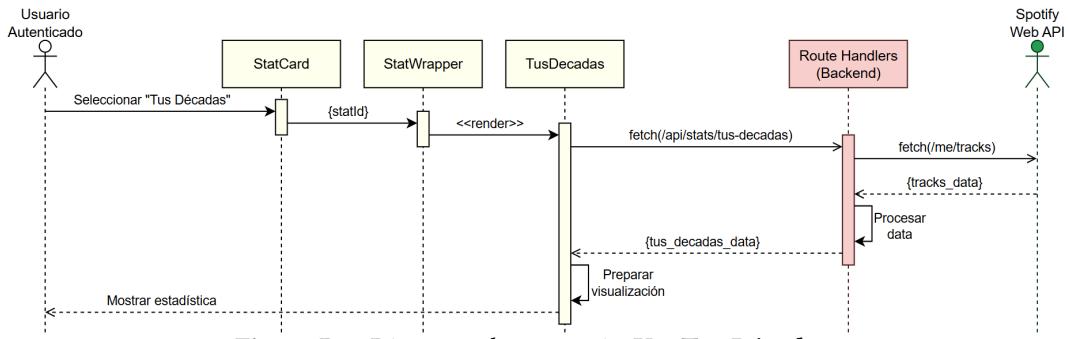


Figura B.4: Diagrama de secuencia: Ver Tus Décadas.

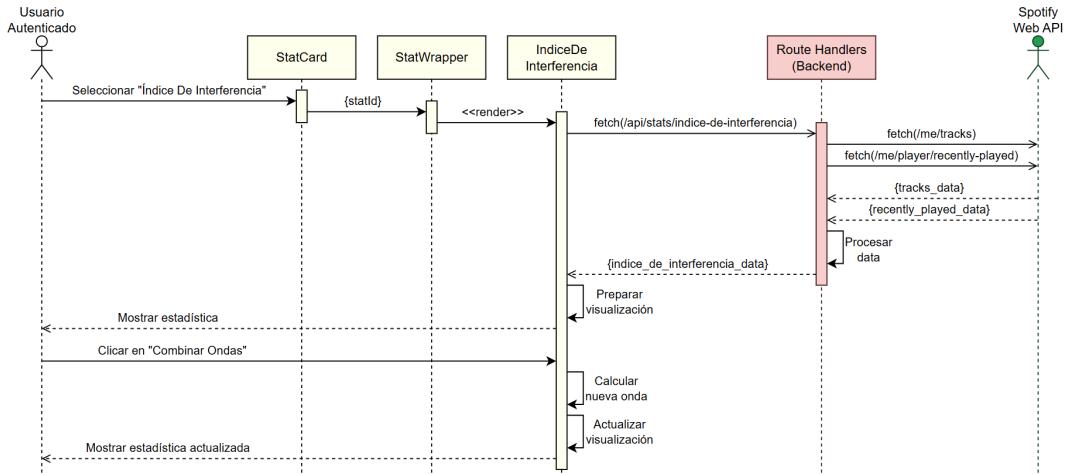


Figura B.5: Diagrama de secuencia: Ver Índice De Interferencia.

Contenidos Relacionados con la Implementación

Fichero .env.local

```
1 # ID del cliente registrado en la API de Spotify
2 SPOTIFY_CLIENT_ID="81af...642b"
3
4 # Secreto del cliente registrado en la API de Spotify no compartir
4   ↪ nunca
5 SPOTIFY_CLIENT_SECRET="1207...393e"
6
7 # URL del dominio donde se ejecuta la aplicacion en desarrollo,
7   ↪ localhost
8 DOMAIN_URL="http://localhost:3000"
9
10 # URI de redireccion configurada en Spotify para la autenticacion
10   ↪ OAuth
11 SPOTIFY_REDIRECT_URI="http://localhost:3000/api/auth/callback"
12
13 # URL utilizada por NextAuth para gestionar la autenticacion en la
13   ↪ aplicacion
14 NEXTAUTH_URL="http://localhost:3000/api/auth/callback"
```

Algoritmo C.1: Variables de entorno necesarios en el fichero .env.local.

Figura del Panel de Monitoreo de Spotify

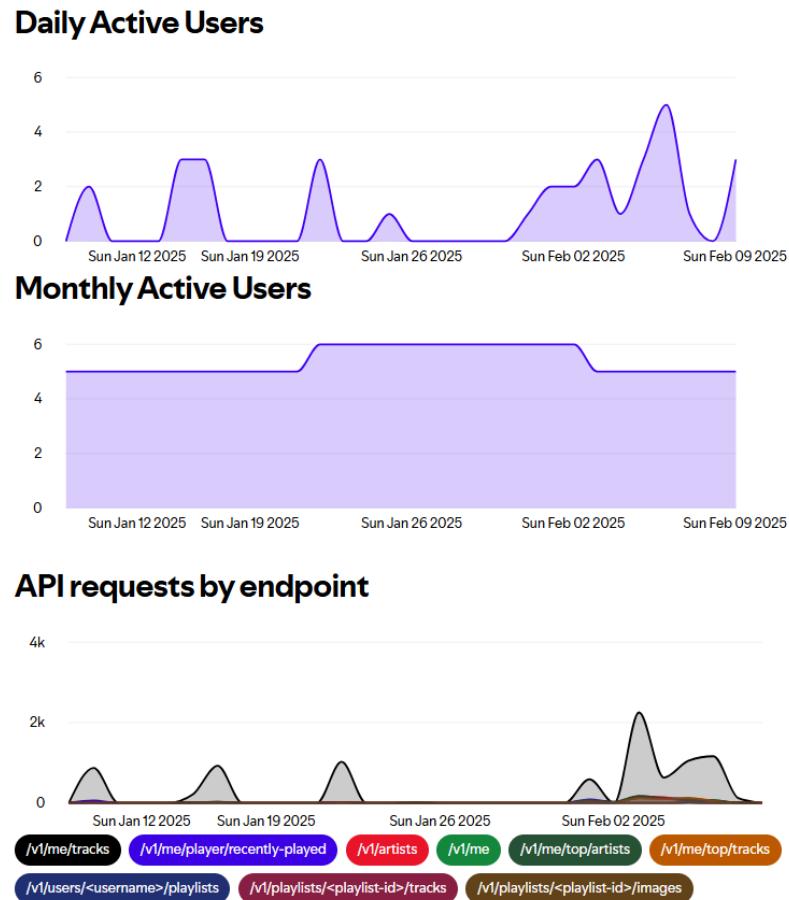


Figura C.1: Panel de monitoreo de la actividad de la aplicación registrada.

Definición del Hook Personalizado useFetch

```
1 import { useEffect, useState } from "react";
2
3 interface UseFetchResult<T> {
4     data: T | null;
5     loading: boolean;
6     error: string | null;
7 }
8
9 export function useFetch<T>(url: string): UseFetchResult<T> {
10     const [data, setData] = useState<T | null>(null);
11     const [loading, setLoading] = useState(true);
12     const [error, setError] = useState<string | null>(null);
13
14     useEffect(() => {
15         const controller = new AbortController();
16         const signal = controller.signal;
17         let isAborted = false;
18
19         console.log("== INICIO DE FETCH, setLoading(true) ==");
20         setLoading(true);
21         setData(null);
22         setError(null);
23
24         fetch(url, { credentials: "include", signal })
25             .then((response) => {
26                 if (!response.ok) throw new Error("Failed to fetch data");
27                 return response.json() as Promise<T>;
28             })
29             .then((result) => {
30                 if (!isAborted) setData(result);
31             })
32             .catch((err) => {
33                 if (err.name !== "AbortError") {
34                     console.error("Error fetching data:", err);
35                     if (!isAborted) setError("Error al cargar los datos. Por
36                         ↪ favor, intentalo de nuevo mas tarde.");
37                 } else {
38                     console.log("== FETCH ABORTADO ==");
39                 }
40             })
41             .finally(() => {
42                 if (!isAborted) {
43                     console.log("== FIN DE FETCH, setLoading(false) ==");
44                     setLoading(false);
45                 }
46             });
47
48         return () => {
49             console.log("== ABORTANDO FETCH ==");
50             isAborted = true;
51             controller.abort();
52         };
53     }, [url]);
54
55     return { data, loading, error };
56 }
```

Algoritmo C.2: Definición del *hook* personalizado useFetch para la obtención de datos de la API con gestión de estado y abortos de petición.

Código de los Componentes Home y Stats

```
1  export default async function Home({
2    searchParams,
3  ): {
4    searchParams: Promise<Record<string, string | string[] | undefined>>;
5  } {
6    const resolvedSearchParams = await searchParams;
7
8    const timeRange = Array.isArray(resolvedSearchParams.time_range)
9      ? resolvedSearchParams.time_range[0]
10     : resolvedSearchParams.time_range || "short_term";
11
12   return (
13     <main className='min-h-screen relative'>
14       <section className='bg-spotify-black'>
15         <div className='max-w-5xl mx-auto px-4 md:px-8 py-8'>
16           <h1 className='text-4xl md:text-5xl font-bold mb-12 text-center bg-gradient-to-r from-spotify-green to-spotify-blue bg-clip-text text-transparent'>
17             Tus Spotify Tops
18           </h1>
19           <Suspense fallback={<UserProfileSkeleton />}>
20             <UserProfile />
21           </Suspense>
22           <div className='flex justify-center mt-8'>
23             <TimeRangeSelector />
24           </div>
25           <div className='grid grid-cols-1 md:grid-cols-2 gap-4'>
26             <Suspense fallback={<TrackArtistSkeleton count={5} />}>
27               <TopTracks timeRange={timeRange} />
28             </Suspense>
29             <Suspense fallback={<TrackArtistSkeleton count={5} />}>
30               <TopArtists timeRange={timeRange} />
31             </Suspense>
32             <div className='md:col-span-2'>
33               <Suspense fallback={<GenreSkeleton count={5} />}>
34                 <TopGenres timeRange={timeRange} />
35               </Suspense>
36             </div>
37           </div>
38         </section>
39       <section className='bg-[#0A0A0A]'>
40         <div className='max-w-5xl mx-auto px-4 md:px-8 py-12'>
41           <RecentlyPlayed />
42         </div>
43       </section>
44     </main>
45   );
46 }
47 }
```

Algoritmo C.3: Definición del componente Home, encargado de renderizar la página principal con las estadísticas de usuario en Spotify.

```
1  "use client";
2
3  import { useState } from "react";
4  import StatsGrid from "@/components/StatsGrid";
5  import StatWrapper from "@/components/StatWrapper";
6
7  // Define the type for a single stat item
8  type StatItem = {
9    id: string;
10   title: string;
11   iconName: keyof typeof import("lucide-react");
12   className?: string;
13 };
14
15 // Define the stats array with the correct types
16 const stats: StatItem[] = [
17   {
18     id: "hall-of-fame",
19     title: "Hall of Fame",
20     iconName: "Award",
21     className: "md:col-span-2 lg:col-span-2 lg:row-span-2",
22   },
23   { id: "tus-decadas", title: "Tus Decadas", iconName: "Rewind",
24     ↪ className: "md:col-span-1 lg:col-span-2" },
25   {
26     id: "huella-del-dia",
27     title: "Huella Del Dia",
28     iconName: "Fingerprint",
29   },
30   { id: "estaciones-musicales", title: "Estaciones Musicales",
31     ↪ iconName: "SunSnow" },
32   {
33     id: "la-bitacora",
34     title: "La Bitacora",
35     iconName: "BookMarked",
36     className: "md:col-span-2 lg:col-span-2",
37   },
38   { id: "indice-de-interferencia", title: "indice de Interferencia
39     ↪ ", iconName: "AudioWaveform" },
40 ];
41
42 export default function Stats() {
43   const [activeStat, setActiveStat] = useState<string | null>(null
44     ↪ );
45   const [isModalOpen, setIsModalOpen] = useState(false);
46
47   const handleStatClick = (statId: string) => {
48     setActiveStat(statId);
49     setIsModalOpen(true);
50   };
51
52   const handleCloseModal = () => {
53     setActiveStat(null);
54     setIsModalOpen(false);
55   };
56
57   return (
58     <main className='bg-spotify-black min-h-screen'>
      <div className='max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-12'>
        <h1 className='text-4xl md:text-5xl font-bold mb-12 text-center bg-gradient-to-r from-spotify-green to-spotify-blue bg-clip-text text-transparent'>
          Estadisticas Avanzadas
        </h1>
      </div>
    </main>
  );
}

```

Código del Componente StatWrapper

```
1  "use client";
2
3  import { Dialog, DialogContent, DialogHeader, DialogTitle } from "
4      ↪ @/components/ui/dialog";
5  import dynamic from "next/dynamic";
6
7  const statComponents = {
8      "hall-of-fame": dynamic(() => import("@/components/stats/
9          ↪ HallOfFame")),
10     "estaciones-musicales": dynamic(() => import("@/components/stats/
11         ↪ /EstacionesMusicales")),
12     "huella-del-dia": dynamic(() => import("@/components/stats/
13         ↪ HuellaDelDia")),
14     "la-bitacora": dynamic(() => import("@/components/stats/
15         ↪ LaBitacora")),
16     "tus-decadas": dynamic(() => import("@/components/stats/
17         ↪ TusDecadas")),
18     "indice-de-interferencia": dynamic(() => import("@/components/
19         ↪ stats/IndiceDeInterferencia")),
20 };
21
22 interface StatWrapperProps {
23     activeStat: string | null;
24     isOpen: boolean;
25     onClose: () => void;
26 }
27
28 export default function StatWrapper({ activeStat, isOpen, onClose
29     ↪ }: StatWrapperProps) {
30     const getStatComponent = (statId: string | null) => {
31         if (!statId || !(statId in statComponents)) {
32             return (
33                 <div className='text-sm text-spotify-gray-100'>
34                     Selecciona una estadística válida para ver los detalles.
35                 </div>
36             );
37         }
38         const DynamicComponent = statComponents[statId as keyof typeof
39             ↪ statComponents];
40         return <DynamicComponent />;
41     };
42
43     return (
44         <Dialog open={isOpen} onOpenChange={(open) => !open && onClose
45             ↪ ()}>
46             <DialogContent
47                 className={`${`${
48                     activeStat === "tus-decadas"
49                         ? "w-[90vw] max-w-7xl"
50                         : "w-[95vw] max-w-4xl"
51                 } min-h-[60vh] max-h-[90vh] overflow-y-auto p-8 flex flex-
52                     ↪ col'}`}
53             >
54                 <DialogHeader>
55                     <DialogTitle className='text-2xl font-bold text-spotify-
56                         ↪ green'>
57                         {activeStat
58                             ? activeStat
59                                 .split("-")
60                                 .map((word) => word.charAt(0).toUpperCase() +
61                                     ↪ word.slice(1))
62                                 .join(" ")
63                         : "Estadística"
64                     }
65                 </DialogTitle>
66             </DialogContent>
67         </Dialog>
68     );
69 }
70
71 
```

Código Completo del Middleware

```
1 import { NextResponse } from "next/server";
2 import type { NextRequest } from "next/server";
3 import { renovarAccessToken } from "@/lib/spotify";
4
5 export async function middleware(req: NextRequest) {
6   const access_token = req.cookies.get("access_token");
7   const refresh_token = req.cookies.get("refresh_token");
8
9   // Caso 1: Si no hay ni access_token ni refresh_token, redirir
10    ↪ al login
11   if (!access_token && !refresh_token) {
12     console.log("No hay tokens, redirigiendo al login...");
13     return NextResponse.redirect(new URL("/", req.url));
14   }
15
16   // Caso 2: Si hay un refresh_token pero no un access_token,
17    ↪ renovar el token
18   if (!access_token && refresh_token) {
19     console.log("No hay access_token, renovando...");
20     const new_tokens = await renovarAccessToken(refresh_token.
21       ↪ value);
22
23     if (!new_tokens) {
24       return NextResponse.redirect(new URL("/", req.url));
25     }
26
27     const res = NextResponse.next();
28
29     res.cookies.set("access_token", new_tokens.access_token, {
30       httpOnly: true,
31       secure: process.env.NODE_ENV === "production",
32       path: "/",
33       maxAge: new_tokens.expires_in, // 1 hora
34     });
35
36     if (new_tokens.refresh_token) {
37       res.cookies.set("refresh_token", new_tokens.refresh_token, {
38         httpOnly: true,
39         secure: process.env.NODE_ENV === "production",
40         path: "/",
41         maxAge: 60 * 60 * 24, // 1 dia
42       });
43     }
44
45     return res;
46   }
47
48   // Caso 3: Si hay un access_token, dejar que la petición
49    ↪ continue
50   return NextResponse.next();
51 }
52
53 // Rutas protegidas
54 export const config = {
55   matcher: ["/home/:path*", "/stats/:path*"],
56};
```

Algoritmo C.6: Middleware global en Next.js para la gestión y renovación de tokens de autenticación.105

```

1  export async function renovarAccessToken(refresh_token: string) {
2      const clientId = process.env.SPOTIFY_CLIENT_ID!;
3      const clientSecret = process.env.SPOTIFY_CLIENT_SECRET!;
4      const auth_header = Buffer.from(`${clientId}:${clientSecret}`).  

5          ↪ toString("base64");
6
7      try {
8          const response = await fetch(`https://accounts.spotify.com/api  

9              ↪ /token`, {  

10                 method: "POST",
11                 headers: {
12                     "Content-Type": "application/x-www-form-urlencoded",
13                     Authorization: `Basic ${auth_header}`,
14                 },
15                 body: new URLSearchParams({  

16                     grant_type: "refresh_token",
17                     refresh_token: refresh_token,
18                 }),
19             });
20
21         if (!response.ok) {
22             console.error("Error al renovar el token:", await response.  

23                 ↪ text());
24             return null;
25         }
26
27         const data = await response.json();
28         console.log("Access token renovado con éxito");
29         console.log("Nuevos tokens:\n", data);
30
31         return {
32             ...data,
33             refresh_token: data.refresh_token || refresh_token,
34         };
35     } catch (error) {
36         console.error("Error durante la renovación del access token:",  

37             ↪ error);
38         return null;
39     }
40 }
```

Algoritmo C.7: Función para la renovación del access_token utilizando el refresh_token en la API de Spotify.

Función de Optimización de Imagen en Hall Of Fame

```
1  const optimizeImage = async (canvas: HTMLCanvasElement, maxSize:
2      ↪ number): Promise<string> => {
3    let quality = 0.9;
4    let imageBase64: string;
5    let size: number;
6
7    do {
8      imageBase64 = canvas.toDataURL("image/jpeg", quality).split(",",
9          ↪ ")[1];
10     size = Math.round((imageBase64.length * 0.75) / 1024);
11     console.log('Trying quality: ${quality.toFixed(2)}, size: ${
12         ↪ size}KB');
13     quality -= 0.1;
14   } while (size > maxSize && quality > 0.1);
15
16   if (size > maxSize) {
17     // If still too large, reduce dimensions
18     const scaledCanvas = document.createElement("canvas");
19     const ctx = scaledCanvas.getContext("2d");
20     const scale = Math.sqrt(maxSize / size);
21
22     scaledCanvas.width = canvas.width * scale;
23     scaledCanvas.height = canvas.height * scale;
24
25     if (ctx) {
26       ctx.drawImage(canvas, 0, 0, scaledCanvas.width, scaledCanvas
27           ↪ .height);
28       imageBase64 = scaledCanvas.toDataURL("image/jpeg", 0.7).
29           ↪ split(",")[1];
30       size = Math.round((imageBase64.length * 0.75) / 1024);
31       console.log('Scaled down image. Final size: ${size}KB');
32     }
33   }
34
35   return imageBase64;
36 }
```

Algoritmo C.8: Función `optimizeImage()` de optimización de imágenes con ajuste de calidad y escala.

Función de Creación de SVGs en Estaciones Musicales

```
1  const calculatePath = (startAngle: number, endAngle: number,
2    ↪ isHovered: boolean) => {
3      const r = isHovered ? expandedRadius : radius;
4
5      const x1 = r * Math.cos((startAngle * Math.PI) / 180);
6      const y1 = r * Math.sin((startAngle * Math.PI) / 180);
7      const x2 = r * Math.cos((endAngle * Math.PI) / 180);
8      const y2 = r * Math.sin((endAngle * Math.PI) / 180);
9
10     const largeArcFlag = endAngle - startAngle > 180 ? 1 : 0;
11
12     return `
13       M ${x1} ${y1}
14       A ${r} ${r} 0 ${largeArcFlag} 1 ${x2} ${y2}
15       L ${innerRadius * Math.cos((endAngle * Math.PI) / 180)} ${
16         ↪ innerRadius * Math.sin((endAngle * Math.PI) / 180)}
17       A ${innerRadius} ${innerRadius} 0 ${largeArcFlag} 0 ${
18         innerRadius * Math.cos((startAngle * Math.PI) / 180)
19       } ${innerRadius * Math.sin((startAngle * Math.PI) / 180)}
20       Z
21     `;
22   };
23 }
```

Algoritmo C.9: Cálculo de la trayectoria de los segmentos en el gráfico de *Estaciones Musicales*.

Funciones de Movimiento y Zoom de Tus Décadas

```
1  <Stage
2    width={stageWidth}
3    height={600}
4    draggable
5    scaleX={scale}
6    scaleY={scale}
7    x={position.x}
8    y={position.y}
9    onWheel={handleWheel}
10   onMouseDown={() => setCursorStyle("grabbing")} // Cambia a "
11     ↪ grabbing"
12   onMouseUp={() => setCursorStyle("grab") } // Vuelve a "grab"
13   onMouseLeave={() => setCursorStyle("grab") } // Asegura que el
14     ↪ cursor vuelva a "grab" al salir
15   style={{
16     cursor: cursorStyle, // Usa el estado del cursor
17   }}
18 >
```

Algoritmo C.10: Configuración del componente Stage en *Tus Décadas*, permitiendo navegación y zoom dinámico.

```

1  const handleWheel = (e: any) => {
2      e.evt.preventDefault();
3
4      const scaleBy = 1.2;
5      const stage = e.target.getStage();
6      if (!stage) return;
7
8      const oldScale = stage.scaleX();
9      const mousePointTo = {
10          x: stage.getPointerPosition()!.x / oldScale - stage.x() /
11              ↪ oldScale,
12          y: stage.getPointerPosition()!.y / oldScale - stage.y() /
13              ↪ oldScale,
14      };
15
16      const newScale = e.evt.deltaY > 0 ? oldScale / scaleBy :
17          ↪ oldScale * scaleBy;
18      setScale(newScale);
19
20      const newPos = {
21          x: -(mousePointTo.x - stage.getPointerPosition()!.x /
22              ↪ newScale) * newScale,
23          y: -(mousePointTo.y - stage.getPointerPosition()!.y /
24              ↪ newScale) * newScale,
25      };
26
27      setPosition(newPos);
28  };

```

Algoritmo C.11: Manejo del evento de desplazamiento con la rueda del ratón en *Tus Décadas* para aplicar zoom dinámico.

Funciones de Generación de Ondas de Índice De Interferencia

```

1  const generateWaveData = (frequency: number, amplitude: number =
2      ↪ 0.5, phase: number = 0) => {
3      return Array.from({ length: 200 }, (_, i) => ({
4          x: i,
5          y: amplitude * Math.sin(((frequency / 25) * i * Math.PI) /
6              ↪ 24 + phase),
7      }));
8  };

```

Algoritmo C.12: Generación de datos de onda sinusoidal en *Índice de Interferencia* a partir de la frecuencia de escucha del usuario.

```

1   const drawWave = (data: { x: number; y: number }[], color: string,
2     ↪ delay: number = 0, waveType: WaveType) => {
3     const path = svg
4       .append("path")
5       .datum(data)
6       .attr("fill", "none")
7       .attr("stroke", color)
8       .attr("stroke-width", 2)
9       .attr("d", lineGenerator)
10      .attr("opacity", 0)
11      .attr("class", `wave-${waveType}`);
12
13      const totalLength = path.node()?.getTotalLength() || 0;
14
15      path
16        .attr("stroke-dasharray", totalLength + " " + totalLength)
17        .attr("stroke-dashoffset", totalLength)
18        .transition()
19        .duration(1000)
20        .delay(delay)
21        .attr("opacity", 1)
22        .transition()
23        .duration(1500)
24        .attr("stroke-dashoffset", 0);
25
26      return path;
27    };

```

Algoritmo C.13: Función drawWave() para dibujar ondas sinusoidales animadas en *Índice de Interferencia* utilizando D3.js.

Información del Backend de las Estadísticas Avanzadas

D.1. Información del Backend del Hall Of Fame

```
1 // * GET /api/stats/hall-of-fame
2 1. Obtener access_token de las cookies.
3 2. Si no hay access_token, devolver error 401.
4 3. Realizar una petición a la API de Spotify para obtener las 16
   ↪ canciones mas escuchadas a largo plazo.
5 4. Si la petición no es exitosa, devolver error 500.
6 5. Extraer la información relevante de cada canción:
7   - URL de la imagen del álbum
8   - Nombre de la canción
9   - Nombre del artista principal
10 6. Estructurar los datos en un array de objetos Album.
11 7. Enviar la respuesta en formato JSON con los datos procesados.
12
13 // * POST /api/stats/hall-of-fame
14 1. Obtener access_token de las cookies.
15 2. Si no hay access_token, devolver error 401.
16 3. Obtener la imagen de portada desde el cuerpo de la petición.
17 4. Realizar una petición a la API de Spotify para obtener las 16
   ↪ canciones mas escuchadas.
18 5. Extraer las URIs de las canciones.
19 6. Obtener el perfil del usuario y su userId.
20 7. Crear una nueva playlist privada llamada "Hall Of Fame".
21 8. Extraer el playlistId de la respuesta.
22 9. Anadir las canciones obtenidas a la playlist recien creada.
23 10. Actualizar la imagen de portada de la playlist con la imagen
    ↪ proporcionada.
24 11. Enviar respuesta en JSON con el playlistId creado si todo ha
    ↪ sido exitoso.
```

Algoritmo D.1: Pseudocódigo del procesamiento de datos en el endpoint Hall Of Fame.

D.2. Informacion del Backend de Estaciones Musicales

```
1  {
2      "title": "Hall of Fame",
3      "albums": [
4          {
5              "albumArtUrl": "https://i.scdn.co/image/abc123",
6              "track": "Bohemian Rhapsody",
7              "artist": "Queen"
8          },
9          {
10             "albumArtUrl": "https://i.scdn.co/image/xyz456",
11             "track": "Billie Jean",
12             "artist": "Michael Jackson"
13         },
14         ...
15     ]
16 }
```

Algoritmo D.2: Ejemplo de estructura de datos enviada en el endpoint Hall Of Fame.

D.2. Informacion del Backend de Estaciones Musicales

```
1 // * GET /api/stats/estaciones-musicales
2 1. Obtener access_token de las cookies.
3 2. Si no hay access_token, devolver error 401.
4 3. Inicializar una estructura para almacenar los datos de artistas
   ↳ y generos por estacion.
5 4. Obtener todos los tracks guardados por el usuario hasta un año
   ↳ atrás, recorriendo la API de Spotify con paginacion.
6 5. Determinar la estacion correspondiente de cada track segun su
   ↳ fecha de agregado.
7 6. Contar la frecuencia de aparicion de cada artista y almacenar
   ↳ sus IDs.
8 7. Obtener datos de artistas en lotes de 50 desde la API de
   ↳ Spotify.
9 8. Actualizar la informacion de cada artista con su imagen y
   ↳ contar los generos musicales asociados.
10 9. Determinar el artista y genero mas representativos en cada
    ↳ estacion segun la frecuencia de aparicion.
11 10. Estructurar los datos en un objeto JSON con la informacion
    ↳ obtenida.
12 11. Enviar la respuesta en formato JSON con los datos procesados.
```

Algoritmo D.3: Pseudocodigo del procesamiento de datos en el endpoint Estaciones Musicales.

D.2. Informacion del Backend de Estaciones Musicales

```
1  {
2      "invierno": {
3          "artist": {
4              "name": "Coldplay",
5              "artistPicUrl": "https://i.scdn.co/image/coldplay123"
6          },
7          "genre": {
8              "name": "Alternative Rock"
9          }
10     },
11     "primavera": {
12         "artist": {
13             "name": "Dua Lipa",
14             "artistPicUrl": "https://i.scdn.co/image/dualipa123"
15         },
16         "genre": {
17             "name": "Pop"
18         }
19     },
20     "verano": {
21         "artist": {
22             "name": "Bad Bunny",
23             "artistPicUrl": "https://i.scdn.co/image/badbunny123"
24         },
25         "genre": {
26             "name": "Reggaeton"
27         }
28     },
29     "otono": {
30         "artist": {
31             "name": "The Beatles",
32             "artistPicUrl": "https://i.scdn.co/image/beatles123"
33         },
34         "genre": {
35             "name": "Rock"
36         }
37     }
38 }
```

Algoritmo D.4: Ejemplo de estructura de datos enviada en el endpoint Estaciones Musicales.

D.3. Información del Backend de Huella del Día

```

1 // * GET /api/stats/huella-del-dia
2 1. Obtener access_token de las cookies.
3 2. Si no hay access_token, devolver error 401.
4 3. Inicializar la URL base para obtener el historial de
   ↳ reproduccion reciente.
5 4. Inicializar una lista para almacenar todas las canciones
   ↳ reproducidas.
6 5. Realizar una peticion a la API de Spotify para obtener las
   ↳ ultimas 50 canciones reproducidas.
7 6. Si la peticion es exitosa:
8     a. Agregar las canciones obtenidas a la lista total.
9     b. Verificar si hay mas paginas de resultados.
10    c. Si hay mas paginas, actualizar la URL y repetir la peticion.
11 7. Si la peticion falla, devolver error con el mensaje
   ↳ correspondiente.
12 8. Inicializar un array de 24 posiciones para almacenar los
   ↳ minutos escuchados por cada hora del dia.
13 9. Iterar sobre todas las canciones obtenidas y extraer:
14     a. La hora en la que se reprodujo la cancion.
15     b. Su duracion en minutos (limitada a un maximo de 60 por hora)
   ↳ .
16 10. Acumular el tiempo de reproduccion en la hora correspondiente
   ↳ dentro del array.
17 11. Limitar cada posicion del array a un maximo de 60 minutos por
   ↳ hora.
18 12. Enviar la respuesta en formato JSON con los datos procesados.

```

Algoritmo D.5: Pseudocodigo del procesamiento de datos en el endpoint Huella del Dia.

```

1 [5, 12, 25, 35, 50, 42, 30, 20, 15, 10, 8, 5, 3, 2, 1, 0, 0, 0, 5,
   ↳ 10, 20, 30, 40, 50]

```

Algoritmo D.6: Ejemplo de estructura de datos enviada en el endpoint Huella del Dia.

D.4. Información del Backend de La Bitácora

```

1  // * GET /api/stats/la-bitacora
2  1. Obtener access_token de las cookies.
3  2. Si no hay access_token, devolver error 401.
4  3. Extraer los parametros de consulta (year y month) de la URL.
5  4. Si la cache no esta inicializada:
6      a. Obtener todas las canciones guardadas del usuario desde la
         ↳ API de Spotify.
7      b. Extraer las fechas de guardado de cada cancion y
         ↳ estructurarlas en objetos con year, month y day.
8      c. Agregar todas las canciones obtenidas a una lista total.
9      d. Crear una estructura de almacenamiento:
10         - Diccionario de anios con el total de canciones guardadas
             ↳ por cada anio.
11         - Diccionario de meses por anio con el total de canciones
             ↳ guardadas por cada mes.
12         - Diccionario de dias por anio-mes con el total de canciones
             ↳ guardadas por dia.
13  5. Completar los periodos faltantes con valor 0:
14      a. Rellenar los anio faltantes desde el primer hasta el ultimo.
15      b. Rellenar los meses faltantes en cada anio.
16      c. Rellenar los dias faltantes en cada mes considerando la
         ↳ cantidad de dias de cada mes.
17  6. Si no se proporciona un anio en la consulta, devolver la
     ↳ estructura de datos por anio.
18  7. Si se proporciona solo un anio, devolver la estructura de datos
     ↳ por mes para ese anio.
19  8. Si se proporciona un anio y un mes, devolver la estructura de
     ↳ datos por dia para ese mes.
20  9. Enviar la respuesta en formato JSON con los datos procesados.

```

Algoritmo D.7: Pseudocódigo del procesamiento de datos en el endpoint La Bitacora.

```

1  {
2      "2020": 150,
3      "2021": 120,
4      "2022": 180
5  }

```

Algoritmo D.8: Ejemplo de estructura de datos enviada por año en el endpoint La Bitacora.

```

1  {
2      "2022-01": 15,
3      "2022-02": 10,
4      "2022-03": 20
5  }

```

Algoritmo D.9: Ejemplo de estructura de datos enviada por mes en el endpoint La Bitacora.

D.5. Información del Backend de Tus Décadas

```
1  {
2    "2022-03-01": 5,
3    "2022-03-02": 3,
4    "2022-03-03": 8
5 }
```

Algoritmo D.10: Ejemplo de estructura de datos enviada por dia en el endpoint La Bitacora.

D.5. Información del Backend de Tus Décadas

```
1 // * GET /api/stats/tus-decadas
2 1. Obtener access_token de las cookies.
3 2. Si no hay access_token, devolver error 401.
4 3. Inicializar una lista para almacenar los albums procesados y un
   ↪ conjunto para evitar duplicados.
5 4. Definir la URL inicial para la paginacion de las canciones
   ↪ guardadas en Spotify.
6 5. Mientras haya una URL valida:
7   a. Realizar una peticion a la API de Spotify para obtener los
      ↪ tracks guardados.
8   b. Iterar sobre los tracks obtenidos:
9     - Si el album ya ha sido procesado, omitirlo.
10    - Extraer la imagen de mayor resolucion disponible.
11    - Si no hay imagen disponible, omitir el track.
12    - Extraer el anio de lanzamiento del album segun su
       ↪ precision de fecha.
13    - Agregar la informacion del album procesado a la lista.
14    - Registrar el album en el conjunto de albums procesados.
15   c. Actualizar la URL con la siguiente pagina de resultados, si
      ↪ existe.
16 6. Ordenar los tracks por anio de lanzamiento.
17 7. Determinar el rango de decadas abarcadas por los datos:
18   - Encontrar el anio minimo y redondearlo al inicio de la decada
     ↪ correspondiente.
19   - Encontrar el anio maximo y redondearlo al final de la decada
     ↪ correspondiente.
20 8. Construir una estructura de datos que agrupe los albums por
     ↪ anio.
21 9. Rellenar los anios faltantes dentro del rango completo de
     ↪ decadas con listas vacias.
22 10. Enviar la respuesta en formato JSON con los datos procesados.
```

Algoritmo D.11: Pseudocodigo del procesamiento de datos en el endpoint Tus Decadas.

D.5. Información del Backend de Tus Décadas

```
1  {
2      "1970": [
3          {
4              "id": "album1",
5              "albumPicUrl": "https://i.scdn.co/image/abc123",
6              "year": 1970
7          }
8      ],
9      "1980": [
10         {
11             "id": "album2",
12             "albumPicUrl": "https://i.scdn.co/image/xyz456",
13             "year": 1980
14         },
15         {
16             "id": "album3",
17             "albumPicUrl": "https://i.scdn.co/image/def789",
18             "year": 1980
19         }
20     ],
21     "1990": [],
22     "2000": [
23         {
24             "id": "album4",
25             "albumPicUrl": "https://i.scdn.co/image/ghi101",
26             "year": 2000
27         }
28     ]
29 }
```

Algoritmo D.12: Ejemplo de estructura de datos enviada en el endpoint Tus Decadas.

D.6. Información del Backend de Índice de Interferencia

```

1  // * GET /api/stats/indice-de-interferencia
2  1. Obtener access_token de las cookies.
3  2. Si no hay access_token, devolver error 401.
4
5  // * Calcular el valor "normal" basado en canciones guardadas
6  3. Definir un porcentaje del 20% de canciones a muestrear, con un
   ↳ minimo de 500 y un maximo de 1000 canciones.
7  4. Realizar una peticion a la API de Spotify para obtener el total
   ↳ de canciones guardadas.
8  5. Si el total de canciones guardadas es menor o igual a 500:
9    a. Obtener todas las canciones paginando la API de Spotify.
10   6. Si hay mas de 500 canciones:
11     a. Generar offsets aleatorios para seleccionar una muestra
        ↳ representativa.
12     b. Realizar peticiones a la API de Spotify usando estos offsets
        ↳ para obtener una muestra aleatoria.
13   7. Calcular la popularidad media de las canciones obtenidas y
        ↳ redondear el resultado.
14
15  // * Calcular el valor "actual" basado en reproducciones recientes
16  8. Realizar una peticion a la API de Spotify para obtener las
   ↳ ultimas 50 canciones reproducidas.
17  9. Calcular la popularidad media de estas canciones y redondear el
   ↳ resultado.
18
19  // * Responder con los datos calculados
20  10. Devolver un JSON con los valores "normal" y "actual".

```

Algoritmo D.13: Pseudocódigo del procesamiento de datos en el endpoint Indice de Interferencia.

```

1  {
2    "normal": 75,
3    "actual": 62
4  }

```

Algoritmo D.14: Ejemplo de estructura de datos enviada en el endpoint Indice de Interferencia.

Bibliografía

- [1] Spotify, “Spotify Design Guidelines,” 2025, Último acceso: 31-01-2025. [Online]. Available: <https://developer.spotify.com/documentation/design> Ver página 53.
- [2] ——, “Authorization Code Flow,” 2025, Último acceso: 30-01-2025. [Online]. Available: <https://developer.spotify.com/documentation/web-api/tutorials/code-flow> Ver página 61.
- [3] L. Hallie, “Behind the scenes of vercel’s infrastructure,” ene 2023, publicado: 27-01-2023. Accedido: 01-02-2025. [Online]. Available: <https://vercel.com/blog/behind-the-scenes-of-vercels-infrastructure> Ver página 85.
- [4] T. Sbano, “Mitigating denial of wallet risks with vercel,” ene 2025, publicado: 24-01-2025. Accedido: 07-02-2025. [Online]. Available: <https://vercel.com/blog/mitigating-denial-of-wallet-risks-with-vercel> Ver página 88.