Universidad Politécnica
de Madrid

Escuela Técnica Superior de

Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de un Cliente Móvil Multiplataforma para Servidores de Música Subsonic

Autor: Xila Cai

Tutor(a): Jose Antonio Calvo-Manzano Villalón

Madrid, Julio 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Desarrollo de un Cliente Móvil Multiplataforma para Servidores de Música Subsonic
Julio 2025

Autor: Xila Cai

Tutor: Jose Antonio Calvo-Manzano Villalón
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente Trabajo de Fin de Grado aborda el desarrollo de una aplicación móvil multiplataforma moderna para servidores de música Subsonic, utilizando React Native como tecnología base. El proyecto surge de la necesidad de superar las limitaciones identificadas en las aplicaciones cliente existentes, que presentan interfaces anticuadas, funcionalidades incompletas y problemas de mantenimiento.

La aplicación desarrollada implementa todas las funcionalidades esenciales del protocolo Subsonic, incluyendo navegación por canciones, álbumes, artistas, playlists y géneros, además de un sistema integral de búsqueda. Una de las principales contribuciones del proyecto es la implementación de un sistema robusto de caché offline que permite la descarga automática de canciones durante la reproducción y garantiza la continuidad del servicio sin conexión a internet.

La arquitectura de la aplicación se basa en una gestión de estado centralizada mediante Zustand, una base de datos SQLite para metadatos locales y el sistema de archivos para almacenamiento de contenido multimedia. El proyecto incluye un pipeline completo de CI/CD configurado con GitHub Actions y una suite exhaustiva de 252 casos de prueba que validan tanto la lógica de negocio como los componentes de interfaz de usuario.

Los resultados demuestran que la aplicación cumple exitosamente con todos los objetivos planteados, ofreciendo una experiencia de usuario moderna y estable en dispositivos iOS y Android. El proyecto representa una alternativa viable a las aplicaciones cliente existentes y contribuye al ecosistema de software descentralizado, promoviendo el control de datos personales y la privacidad en el consumo de música digital.

Abstract

This Final Degree Project addresses the development of a modern cross-platform mobile application for Subsonic music servers, using React Native as the base technology. The project emerges from the need to overcome limitations identified in existing client applications, which present outdated interfaces, incomplete functionalities, and maintenance issues.

The developed application implements all essential functionalities of the Subsonic protocol, including navigation through songs, albums, artists, playlists, and genres, plus a comprehensive search system. One of the main contributions of the project is the implementation of a robust offline cache system that enables automatic song downloads during playback and ensures service continuity without internet connection.

The application architecture is based on centralized state management using Zustand, an SQLite database for local metadata, and the file system for multimedia content storage. The project includes a complete CI/CD pipeline configured with GitHub Actions and a comprehensive test suite of 252 test cases that validate both business logic and user interface components.

Results demonstrate that the application successfully meets all established objectives, offering modern and stable user experience on iOS and Android devices. The project represents a viable alternative to existing client applications and contributes to the decentralized software ecosystem, promoting personal data control and privacy in digital music consumption.

Índice de Contenidos

Índice de Tablas

Índice de Figuras

1. Introducción

En este capítulo, se presentarán el contexto y las razones que han motivado a la realización de este Trabajo de Fin de Grado. A continuación, se detallarán los objetivos principales y secundarios, junto a las tareas requeridas para alcanzarlos. Finalmente, se incluirá un resumen de los capítulos que componen este documento.

1.1 Contexto y Motivación

En las últimas décadas, se ha visto una evolución considerable en el método de consumo de música, pasando de formatos físicos, como los CD y vinilos, a servicios digitales de streaming. Estos servicios han ganado una gran popularidad debido a su sencilla accesibilidad y conveniencia, permitiendo a sus usuarios disfrutar de millones de canciones desde cualquier lugar [1].

Sin embargo, estas plataformas comerciales, como Spotify y Apple Music, también presentan algunas importantes limitaciones. La principal limitación es que estos servicios suelen poner ciertas características esenciales detrás de un modelo premium de suscripción mensual/anual, pudiendo generar un obstáculo para algunos [2]. Otro inconveniente puede ser la falta de control al acceso a la música, ya que el catálogo de cada servicio puede variar en cualquier momento dependiendo de las decisiones de la plataforma, estas en cualquier momento pueden decidir eliminar canciones o introducir restricciones regionales [3]. Por último, muchas de estas plataformas no ofrecen la máxima calidad de audio, ya que carecen de soporte para formatos sin pérdida, como FLAC (Free Lossless Audio Codec), un formato que permite almacenar música sin comprimir los datos de audio [4].

Por estas razones, el concepto de self-hosting ha ganado relevancia, sobre todo para los usuarios más técnicos, como una solución que permite a los usuarios gestionar sus bibliotecas musicales de manera independiente. Una de las opciones más populares en este ámbito es el uso de Subsonic, un protocolo diseñado para el streaming de música desde servidores personales [5]. Este protocolo ofrece una experiencia descentralizada, permitiendo a los usuarios transmitir su música desde cualquier lugar mientras mantienen el control total sobre sus datos. Adicionalmente, es compatible con múltiples formatos de audio y dispositivos, lo que lo convierte en una herramienta versátil y atractiva para los entusiastas del self-hosting [6].

Pero a pesar de las ventajas mencionadas, el ecosistema de Subsonic enfrenta desafíos importantes debido a las limitaciones de las aplicaciones cliente disponibles. Estas aplicaciones, especialmente en dispositivos iOS, suelen carecer de interfaces modernas y funcionalidades avanzadas como la gestión offline o la sincronización eficiente [7]. Esto afecta negativamente a la experiencia del usuario, haciéndola menos intuitiva y funcional en comparación con servicios comerciales. Además, la ausencia de un sistema robusto de caché local y problemas de rendimiento dificultan su uso en situaciones cotidianas, limitando su adopción por parte de un público más amplio [8]. A esto se le suma que algunas de estas aplicaciones ya no son mantenidas, lo que ha llevado a la presencia de bugs sin corregir.

1.2 Objetivos

El objetivo principal de este trabajo es desarrollar una aplicación móvil multiplataforma moderna para servidores Subsonic, utilizando React Native como tecnología base. La aplicación busca resolver las limitaciones actuales de las aplicaciones cliente disponibles, ofreciendo un diseño intuitivo, funcionalidades avanzadas y un rendimiento optimizado. Para alcanzar este objetivo, se han definido los siguientes objetivos específicos:

1. **Diseñar e implementar una interfaz de usuario moderna e intuitiva**: Se desarrollará una interfaz atractiva y funcional, compatible con dispositivos iOS y Android, que facilite la navegación y gestión de bibliotecas musicales. El diseño se centrará en la usabilidad, garantizando una experiencia fluida y consistente en ambas plataformas.

2. **Implementar las funcionalidades principales del protocolo Subsonic**: La aplicación incluirá navegación por canciones, álbumes, artistas y playlists, además de la reproducción de música. Estas funcionalidades serán optimizadas para manejar grandes bibliotecas musicales de forma eficiente y estable.

3. **Diseñar un sistema robusto de caché offline**: Se implementará un sistema que permita a los usuarios descargar y reproducir música sin conexión a internet, asegurando sincronización eficiente con el servidor al restablecer la conexión.

4. **Optimizar la aplicación para bibliotecas extensas**: Se garantizará que la aplicación pueda manejar grandes volúmenes de datos con estabilidad y rendimiento fluido, minimizando tiempos de carga y consumo de recursos del dispositivo.

5. **Configurar un pipeline completo de CI/CD (integración y entrega continua)**: Se establecerá un pipeline de Integración y Entrega Continua para automatizar pruebas, análisis de calidad del código y despliegues, asegurando un desarrollo ágil y controlado.

6. **Implementar pruebas unitarias y de integración**: Se desarrollarán pruebas para validar el correcto funcionamiento de los componentes y su interacción, integrándolas en el pipeline de CI/CD para detectar errores de forma temprana.

7. **Documentar el diseño, desarrollo y pruebas**: Se elaborará una memoria final que detalle la arquitectura, las decisiones técnicas y los resultados obtenidos. Además, se incluirá un manual de usuario para facilitar el uso de la aplicación.

Estos objetivos están diseñados para abordar las deficiencias actuales de las aplicaciones cliente de Subsonic, ofreciendo una solución moderna que mejore la experiencia de los usuarios y promueva la adopción del protocolo como alternativa a los servicios comerciales de streaming.

1.3 Tareas y Diagrama Gantt

Para completar la lista de objetivos anteriormente definidos, se elaboró la siguiente lista de tareas:

- Análisis y diseño de la arquitectura.
    - Realizar un análisis del estado de la cuestión.
    - Analizar los requisitos funcionales y no funcionales del sistema.
    - Familiarizarme con las herramientas y tecnologías necesarias.
    - Definir un estándar de proceso de desarrollo y seleccionar una metodología de desarrollo adecuada.
- Implementación del core de la aplicación y características avanzadas.
    - Implementar las funcionalidades principales.
    - Diseñar e implementar un sistema de caché offline para reproducción sin conexión.
    - Desarrollar la funcionalidad de reproducción en segundo plano.
    - Optimizar la aplicación para manejar bibliotecas de música extensas con estabilidad y eficiencia.
- Configuración y desarrollo de pipeline CI/CD.
    - Configurar un pipeline de CI/CD utilizando GitHub Actions.
    - Automatizar el despliegue y las pruebas.
- Testing y control de calidad.
    - Implementar y realizar pruebas unitarias y de integración para validar la interacción entre componentes.
    - Monitorizar la calidad del código mediante herramientas de análisis estático.
- Elaboración de la Memoria Final.
    - Documentar el diseño, desarrollo y pruebas realizadas.
    - Analizar los resultados obtenidos.
    - Evaluar la aplicación en términos de funcionalidad y estabilidad.
    - Realizar un análisis de impacto del proyecto

La Tabla 1 muestra el Diagrama de Gantt del Trabajo Fin de Grado.

Semana	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Formación																
Análisis																
Diseño																
Desarrollo																
Pruebas																
Memoria																

Tabla 1: Diagrama de Gantt

1.4 Estructura

El trabajo se organiza en ocho capítulos que abarcan desde la introducción del problema hasta las conclusiones y el análisis del impacto. A continuación, se describe brevemente el contenido de cada capítulo:

- **Capítulo 1: Introducción.** Se presenta el contexto general del proyecto, el problema a resolver y los objetivos planteados. También se proporciona una visión global de la estructura del documento.
- **Capítulo 2: Estado de la Cuestión.** Se analiza el panorama actual de las aplicaciones cliente para Subsonic, evaluando sus características, fortalezas y limitaciones. Adicionalmente, se contextualiza el proyecto dentro del ámbito del self-hosting y se justifica la necesidad de una solución moderna.
- **Capítulo 3: Entorno de Desarrollo y Metodología**. Se describen las herramientas y tecnologías utilizadas, como React Native, y se detalla la metodología de desarrollo adoptada. También se explican las decisiones técnicas relacionadas con la arquitectura y el pipeline de CI/CD.
- **Capítulo 4: Proceso de Desarrollo**. Se documenta el proceso de desarrollo de la aplicación, desde el análisis de requisitos hasta la implementación de las funcionalidades principales y avanzadas. Se incluyen diagramas y ejemplos que ilustran las decisiones tomadas.
- **Capítulo 5: Resultados**. Se presentan los resultados obtenidos, evaluando el cumplimiento de los objetivos y analizando el rendimiento de la aplicación. También se incluyen métricas relevantes y ejemplos prácticos del funcionamiento del producto final.
- **Capítulo 6: Conclusiones y Trabajo Futuro**. Se exponen las conclusiones generales del proyecto, destacando los logros alcanzados y las limitaciones encontradas. Además, se proponen posibles líneas de trabajo futuro para mejorar y expandir la aplicación.
- **Capítulo 7: Análisis de Impacto.** Se analiza el impacto potencial del proyecto en los ámbitos tecnológico y social, destacando su contribución al software descentralizado y los beneficios que ofrece a los usuarios en términos de control y privacidad.
- **Capítulo 8: Referencias**. Se incluyen todas las fuentes bibliográficas y recursos utilizados durante el desarrollo del trabajo, organizados de manera clara y siguiendo un formato estándar.

Esta estructura permite abordar de manera clara y concisa todos los aspectos relevantes del proyecto, desde la identificación del problema hasta la evaluación de los resultados y su impacto.

2. Estado de la Cuestión

En este capítulo se analiza el estado actual de las aplicaciones cliente para el protocolo de Subsonic, evaluando sus características, ventajas y limitaciones. Además, se justifica la necesidad de una solución que aborde las carencias identificadas.

2.1 Otras Aplicaciones

El protocolo Subsonic cuenta con diversas aplicaciones móviles que permiten a los usuarios acceder a sus bibliotecas musicales en sus servidores. Sin embargo, todas estas aplicaciones presentan una combinación de ventajas y desventajas que afectan a su funcionalidad y experiencia de usuario. A continuación, se realiza un análisis de las principales aplicaciones disponibles:

- **Substreamer:** Es una aplicación gratuita que se destaca por su interfaz moderna y atractiva, diseñada para mostrar todo el contenido de manera organizada e intuitiva. Sin embargo, carece de algunas funciones esenciales. Aunque permite escuchar música sin conexión, las canciones deben descargarse manualmente una por una, ya que no es posible almacenarlas automáticamente. La aplicación permite visualizar listas de artistas o álbumes y acceder a las canciones asociadas a cada uno, pero no ofrece la opción de ver un listado completo de todas las canciones ni de reproducir una lista temporal con todas ellas en orden aleatorio [9].

- **iSub:** Esta es también una aplicación gratuita y es una de las más veteranas, existiendo desde hace más de 15 años. Es de código abierto y ha pasado por las manos de varios mantenedores a lo largo del tiempo. Las características esenciales están muy bien pulidas, ofreciendo un rendimiento robusto incluso con bibliotecas que superan las 50.000 canciones en el servidor. No obstante, también presenta algunos problemas, ya que la interfaz es anticuada y poco atractiva. El proyecto ha sufrido discontinuaciones debido a la pérdida de interés de los mantenedores, existiendo varias ocasiones en la que el proyecto se ha abandonado y eliminado de la App Store durante años, hasta que un nuevo desarrollador decide retomarlo. Actualmente lleva sin recibir actualizaciones por más de 4 años [10].

- **Amperfy:** Otra alternativa gratuita y de código abierto, destacando por su interfaz minimalista y moderna. Ofrece las funciones básicas que se esperan de un reproductor de música. Sin embargo, el desarrollador ha enfocado sus esfuerzos en convertirla también en una aplicación para manejar podcasts, lo que ha llevado a añadir elementos no relacionados con la música en su interfaz, generando cierta confusión en algunas funcionalidades [11]. Además, presenta varios inconvenientes que no han sido resueltos durante años, como problemas de rendimiento con una librería extensa, un uso excesivo de batería y que el desarrollador rechazó la idea de implementar el modo offline automático cuando no hay conexión a internet [12].

- **play:Sub:** Esta es una opción de pago y de código cerrado, cuenta con una interfaz simple e intuitiva, y ofrece todas las funciones esenciales. Puede cachear tus canciones automáticamente, facilitando su uso en modo offline de manera cómoda y eficiente [13]. A pesar de esto, posee varios bugs importantes sin resolver, como la imposibilidad de reproducir determinadas canciones en ocasiones y tener un algoritmo de reproducción aleatoria deficiente, que genera listas temporales con canciones repetidas. A su vez, la aplicación lleva más de un año sin recibir actualizaciones, y el desarrollador permanece inactivo, sin responder a las consultas en su página web.

2.2 Justificación del desarrollo

El análisis previo de las aplicaciones existentes muestra que, a pesar de que cada una de estas ofrecen soluciones para acceder a las bibliotecas musicales alojadas en servidores propios, ninguna de ellas logra ofrecer una solución completa que combine un diseño moderno, funcionalidades avanzadas, un rendimiento óptimo y que siga mantenida para poder solucionar los bugs existentes. Esto demuestra la necesidad de desarrollar una nueva solución para abordar estas limitaciones, brindando una experiencia de usuario superior y aprovechando las tecnologías modernas garantizar su calidad y estabilidad.

Adicionalmente, este proyecto sería de código abierto, una característica altamente valorada dentro de la comunidad de self-hosting. Estos tipos de proyectos están normalmente enfocados a la privacidad, un tema especialmente relevante en el ámbito del self-hosting, donde proteger los datos personales es una de las razones principales por la que optar por alojar aplicaciones en servidores propios. Los proyectos de código abierto ofrecen una mayor transparencia, permitiendo a los usuarios inspeccionar el código para comprobar que cumple con los estándares de privacidad y seguridad.

Por último, este proyecto representa una excelente oportunidad para aprender y aplicar React Native, una tecnología moderna y versátil que permite desarrollar aplicaciones móviles multiplataformas con un solo código base, así optimizando el tiempo y los recursos del desarrollo.

3. Tecnologías empleadas

Este capítulo presenta las tecnologías, herramientas y bibliotecas utilizadas durante el desarrollo del proyecto. Estas tecnologías se seleccionaron para garantizar la compatibilidad multiplataforma, la eficiencia de desarrollo y la calidad del producto final.

3.1 Frameworks y Lenguajes de Programación

3.1.1 React Native

React Native es el framework principal utilizado para el desarrollo de la aplicación. Es una tecnología que permite crear aplicaciones móviles que funcionan tanto en plataformas iOS como Android utilizando una única base de código. Cuenta con una amplia comunidad y un ecosistema de bibliotecas bien desarrollado, facilitando encontrar soluciones a problemas comunes y extender la funcionalidad de la aplicación [14].

Las principales razones por la que elegir React Native, en lugar de otras opciones, como Flutter, son las siguientes:

- **Compatibilidad con componentes nativos:** React Native permite utilizar componentes de interfaz nativos de cada plataforma con JavaScript o TypeScript. Durante el tiempo de ejecución, React Native genera las vistas de Android o iOS correspondientes, permitiendo desarrollar aplicaciones con un rendimiento y experiencia de usuario comparables a las de las aplicaciones nativas [15].
- **Uso de lenguajes ampliamente utilizados:** React Native es compatible con JavaScript y TypeScript, lenguajes muy populares y ampliamente utilizados en el desarrollo moderno [16]. En cambio, Flutter emplea Dart, un lenguaje que tiene hoy en día un uso más limitado fuera del ecosistema de Flutter [17].
- **Integración con Expo**: Por último, Expo, un framework construido sobre React Native que facilita el desarrollo y despliegue de las aplicaciones móviles. Más adelante, se profundizará en las ventajas que ofrece esta herramienta.

3.1.2 Expo

Expo es un framework que extiende las capacidades de React Native, simplificando el desarrollo de aplicaciones móviles [18]. Proporciona un conjunto de herramientas y servicios preconfigurados que simplifican el desarrollo, la prueba y el despliegue de las aplicaciones. Incluye librerías que permiten acceder fácilmente a las APIs nativas de iOS y Android, como **expo-av** para el manejo de audio y vídeo, o **expo-file-system** para la gestión de archivos [19].

Una de las características más destacadas de Expo es la facilidad para probar aplicaciones en dispositivos reales o emuladores. Sin Expo, probar una aplicación

React Native en iOS requiere contar con XCode, una herramienta exclusiva de MacOS. En cambio, Expo permite ejecutar la aplicación simplemente escaneando un código QR desde el móvil, agilizando el proceso de pruebas y eliminando la necesidad de configuraciones complejas [20].

3.1.3 Typescript

React Native permite desarrollar aplicaciones únicamente utilizando dos lenguajes de programación, JavaScript y TypeScript. En este proyecto se ha optado por usar TypeScript, un lenguaje desarrollado por Microsoft basado en JavaScript que añade tipado estático, lo que mejora la calidad de código y facilita el mantenimiento del proyecto [21].

3.2 Herramientas de desarrollo

En el desarrollo del proyecto se emplearon diversas herramientas para facilitar la creación, prueba y mantenimiento de la aplicación. Estas herramientas se seleccionaron para optimizar la productividad y garantizar la calidad del producto final:

- **Expo CLI:** Herramienta de línea de comandos utilizada para inicializar, desarrollar y desplegar la aplicación. Simplifica la configuración del entorno de desarrollo y permite ejecutar la aplicación en dispositivos reales o emuladores.
- **Expo Go:** Aplicación móvil que permite probar la aplicación directamente en dispositivos iOS y Android, escaneando un código QR generado por Expo CLI. Esto agiliza el proceso de pruebas sin necesidad de configuraciones complejas.
- **Android Studio:** Herramienta utilizada para emular dispositivos Android, permitiendo probar la aplicación en diferentes configuraciones y tamaños de pantalla.
- **Git y GitHub:** Sistema de control de versiones y plataforma de colaboración que permitieron gestionar el código fuente, realizar seguimiento de cambios y colaborar de manera eficiente.

3.3 Bibliotecas y Dependencias Clave

Para el desarrollo de la aplicación se utilizaron varias bibliotecas y dependencias que extendieron las capacidades de React Native y Expo, proporcionando funcionalidades esenciales:

- **expo-secure-store:** Biblioteca para el manejo seguro de datos sensibles, como credenciales de usuario.
- **expo-file-system:** Proporciona acceso a las APIs del sistema de archivos de los dispositivos iOS y Android. En este proyecto, se utilizó para guardar canciones

e imágenes en la memoria caché, optimizando el rendimiento de la aplicación y permitiendo el uso sin conexión.

- **@react-native-async-storage/async-storage:** Solución para el almacenamiento de datos clave-valor de manera local en los dispositivos. En este proyecto, se utilizó para guardar las configuraciones de cada usuario, como su límite de almacenamiento en caché.
- **expo-sqlite**: Base de datos SQLite local usado para el almacenamiento de metadatos de canciones, artistas, álbumes y géneros. Estos datos se utilizan cuando no hay conexión a internet, permitiendo la navegación y búsqueda offline en la biblioteca musical.
- **expo-audio:** Biblioteca utilizada para la reproducción y gestión de audio en la aplicación, permitiendo una experiencia fluida al interactuar con contenido sonoro.
- **@react-native-community/netinfo:** Biblioteca para detectar y manejar el estado de la conexión a internet.

3.4 Herramientas de Calidad y Automatización

Para garantizar la calidad del código y automatizar procesos clave durante el desarrollo, se emplearon las siguientes herramientas:

- **ESLint:** Herramienta de análisis estático utilizada para identificar problemas en el código y garantizar que se sigan las mejores prácticas y estándares establecidos.
- **Prettier:** Formateador de código utilizado, asegurando que el código tenga una estructura uniforme y legible en todo el proyecto.
- **Jest:** Framework de pruebas unitarias utilizado para verificar la funcionalidad de los componentes y asegurar la estabilidad del código.
- **React Native Testing Library**: Biblioteca que facilita las pruebas de integración, proporcionando herramientas para renderizar componentes, simular interacciones del usuario y realizar aserciones sobre el comportamiento de la interfaz de usuario de manera más natural y centrada en el usuario.
- **GitHub Actions:** Plataforma de integración y entrega continua (CI/CD) que automatizó pruebas, revisiones y despliegues del proyecto.
- **Dependabot:** Herramienta integrada en GitHub que monitorea las dependencias del proyecto y sugiere actualizaciones para mantenerlas seguras y actualizadas.

4. Metodología y Enfoque

En este capítulo se describe el enfoque metodológico adoptado para el desarrollo del proyecto, así como las estrategias utilizadas para organizar y gestionar las tareas. Dado que el desarrollo fue realizado de manera individual, se optó por un enfoque ágil y flexible, priorizando la calidad del producto final, la eficiencia en la implementación y la satisfacción de los requisitos planteados.

4.1 Metodología de Desarrollo

Se utilizó un enfoque basado en **Desarrollo Iterativo e Incremental**, el cual permitió dividir el proyecto en pequeñas fases o iteraciones. Cada iteración incluyó las siguientes etapas:

1. **Planificación:** Definición de objetivos específicos y tareas a realizar durante cada iteración.
2. **Diseño y Desarrollo:** Implementación de nuevas funcionalidades o mejoras, siguiendo las prioridades establecidas.
3. **Pruebas:** Verificación del correcto funcionamiento de las funcionalidades desarrolladas, tanto en dispositivos reales como en emuladores.
4. **Revisión y Ajustes:** Identificación de posibles errores o áreas de mejora, y ajustes necesarios antes de avanzar a la siguiente iteración.

Este enfoque permitió mantener un progreso constante, asegurando que cada funcionalidad estuviera completamente implementada y probada antes de continuar con la siguiente.

4.2 Enfoque de Desarrollo

El enfoque adoptado para el desarrollo del proyecto se centró en garantizar la calidad del producto final, la eficiencia en la implementación y la satisfacción de los requisitos planteados. Los principales aspectos del enfoque son los siguientes:

- **Desarrollo centrado en el usuario:** Se priorizó la experiencia del usuario (UX) al diseñar la interfaz y las funcionalidades de la aplicación. Esto incluyó pruebas constantes para asegurar que la aplicación fuera intuitiva y fácil de usar.
- **Desarrollo iterativo:** Cada funcionalidad se desarrolló, probó y mejoró en iteraciones, permitiendo detectar y corregir errores de manera temprana.
- **Pruebas continuas:** Durante todo el proceso de desarrollo, se realizaron pruebas unitarias, funcionales y de integración para garantizar la estabilidad y el correcto funcionamiento de la aplicación.

- **Optimización del rendimiento:** Se dedicó especial atención a optimizar el rendimiento de la aplicación, asegurando tiempos de carga rápidos y un uso eficiente de los recursos del dispositivo.

4.3 Organización del Trabajo

Dado que el desarrollo fue realizado por una sola persona, se emplearon herramientas para organizar y gestionar las tareas de manera eficiente:

- **Git:** Se utilizó para el control de versiones, permitiendo realizar un seguimiento detallado de los cambios en el código.
- **GitHub:** Se usó para almacenar el repositorio del proyecto y gestionar issues relacionados con las tareas pendientes.

4.4 Gestión del Tiempo

Para garantizar el cumplimiento de los objetivos y plazos establecidos, se dividió el proyecto en fases principales:

1. **Fase de Investigación y Planificación:** Investigación de tecnologías, definición de requerimientos y diseño inicial de la arquitectura de la aplicación.
2. **Fase de Desarrollo:** Implementación de las funcionalidades principales de la aplicación, siguiendo las iteraciones planificadas.
3. **Fase de Pruebas:** Pruebas exhaustivas en dispositivos reales y emuladores para garantizar el correcto funcionamiento de la aplicación.
4. **Fase de Documentación:** Elaboración de la documentación técnica y del informe final del proyecto.

5. Implementación

En este capítulo se detalla los requisitos funcionales, los diseños y las implementaciones adoptadas para el desarrollo del proyecto. Se busca proporcionar una visión clara y estructurada de los elementos clave que conforman la aplicación.

5.1 Estructura

El proyecto está desarrollado con Expo, siguiendo una arquitectura modular que permite una organización clara y mantenible del código. En la Figura 1, se puede observar la organización jerárquica de carpetas y archivos que conforman la aplicación.

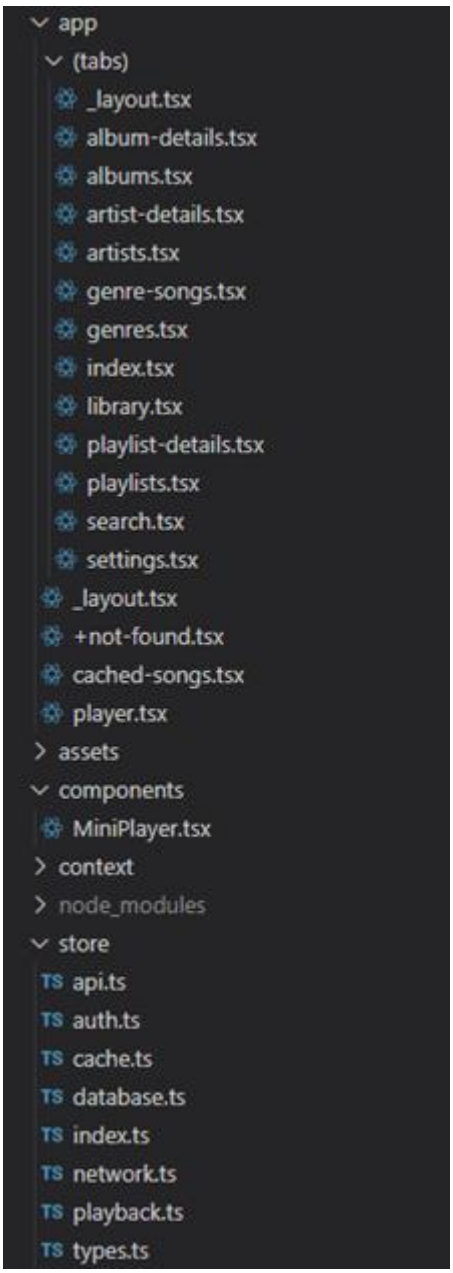

Figura 1: Estructura del proyecto.

La estructura principal se divide en varias carpetas especializadas:

- **app:** Contiene todas las pantallas y rutas de la aplicación, organizadas siguiendo el patrón de Expo Router.
- **components**: Contiene los componentes reutilizables de la interfaz de usuario, destacando el MiniPlayer.tsx que proporciona controles de reproducción persistentes en toda la aplicación.
- **store**: Implementa la gestión de estado utilizando Zustand, con módulos especializados para diferentes aspectos de la aplicación como la API (api.ts), autenticación (auth.ts), caché (cache.ts), red (network.ts), reproducción (playback.ts) y tipos de datos (types.ts).

Esta organización modular facilita el mantenimiento del código, permite la reutilización de componentes y proporciona una separación clara de responsabilidades entre las diferentes funcionalidades de la aplicación.

5.2 Conexión a servidor Subsonic

Al acceder por primera vez a la aplicación, el usuario es recibido con un formulario intuitivo para configurar su servidor Subsonic, similar al que se muestra en la Figura 2. Este formulario incorpora campos claramente etiquetados para la URL del servidor, nombre de usuario y contraseña, priorizando la simplicidad visual sin comprometer la funcionalidad.

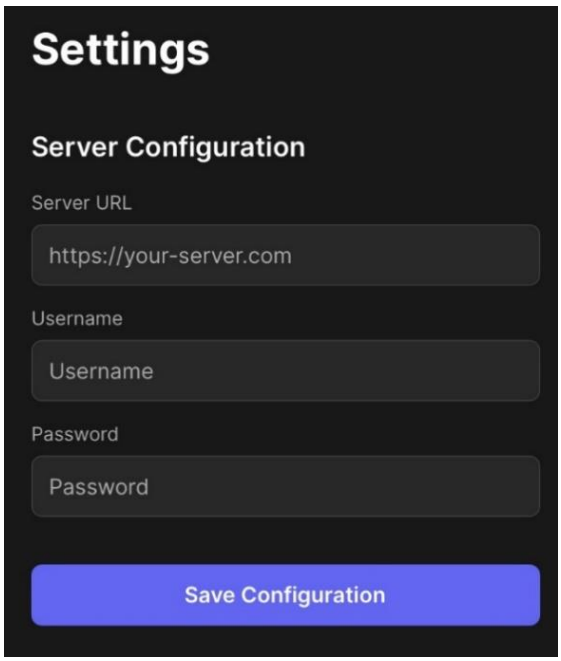

Figura 2: Conexión a servidor.

Una vez que el usuario completa y envía el formulario de configuración, el sistema inicia un flujo de autenticación estructurado como se ilustra en la Figura 3.

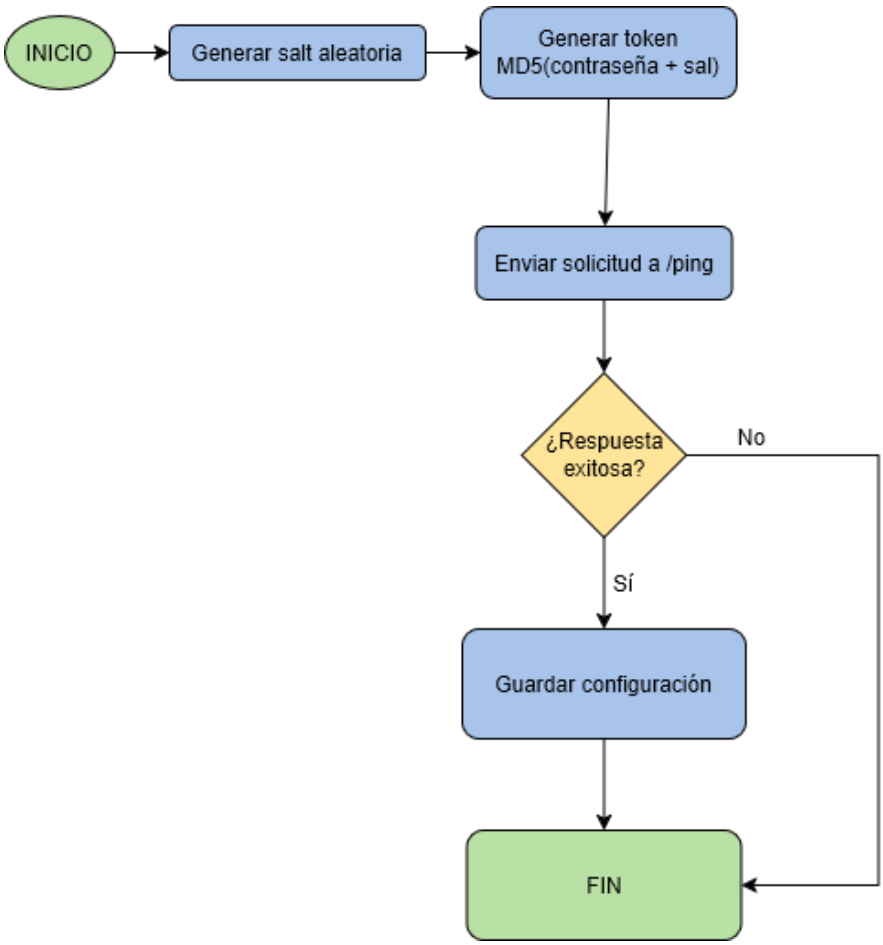

Figura 3: Flujo de conexión al servidor Subsonic.

El flujo comienza con la generación de una sal criptográfica aleatoria compuesta por un mínimo de seis caracteres alfanuméricos, elemento esencial que actúa como factor de aleatorización en el proceso de cifrado. A continuación, el sistema procede a crear el token de autenticación aplicando el algoritmo de hash MD5 sobre la concatenación de la contraseña proporcionada por el usuario y la sal previamente generada, produciendo una cadena hexadecimal de 32 caracteres que representa de forma única las credenciales sin revelar la contraseña original. Es importante señalar que el uso de MD5 constituye una limitación de la API de Subsonic, dado que este algoritmo ha sido considerado criptográficamente inseguro. No obstante, esta implementación resulta preferible al envío de contraseñas en texto plano y debe complementarse con conexiones HTTPS para garantizar una transmisión segura. Esta metodología de autenticación es obligatoria para mantener la compatibilidad con la especificación oficial de la API de Subsonic para versiones posteriores a 1.13.0 [22].

El proceso continúa con el envío de una solicitud al endpoint /ping del servidor Subsonic, incluyendo los parámetros de autenticación generados (usuario, token y sal) junto con la versión de la API y el identificador del cliente, siguiendo el protocolo estándar de Subsonic. Durante este proceso, la interfaz proporciona feedback visual mediante indicadores de carga que mantienen al usuario informado del progreso de

la operación. El sistema evalúa la respuesta del servidor mediante un punto de decisión crítico que determina el flujo posterior del proceso. En caso de error de autenticación o conectividad, como se ejemplifica en la Figura 4, el sistema presenta mensajes de error claros y no intrusivos que incluyen la respuesta específica devuelve el servidor Subsonic, facilitando al usuario la identificación y corrección rápida de problemas. Estos mensajes de error implementan un manejo robusto que abarca desde problemas de red hasta errores de autenticación o configuración incorrecta.

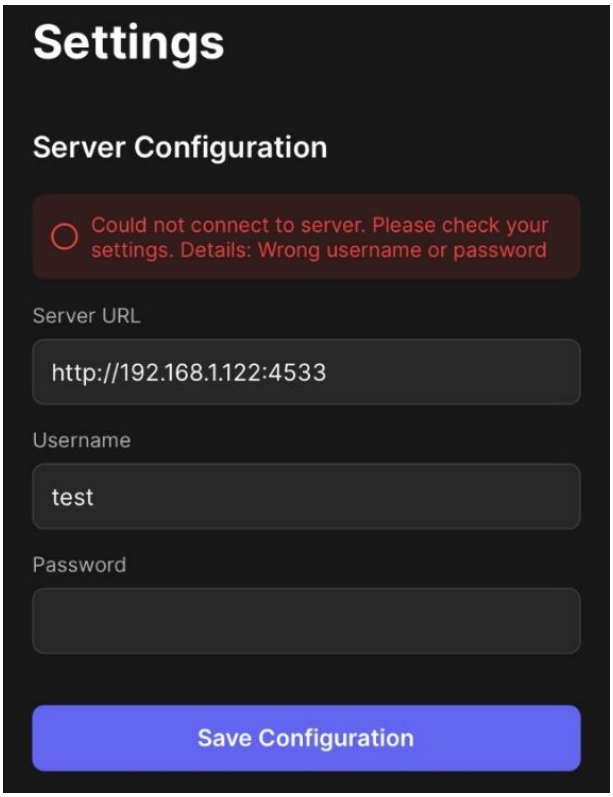

Figura 4: Error al conectarse al servidor.

Cuando la validación es exitosa, el sistema procede al almacenamiento seguro de la configuración utilizando Expo SecureStore, guardando la URL del servidor, nombre de usuario y parámetros necesarios para futuras conexiones. El proceso concluye con una confirmación visual de conexión exitosa como se muestra en la Figura 5, que presenta elementos visuales positivos incluyendo iconos de verificación y colores confirmatorios que proporcionan feedback instantáneo al usuario. Esta confirmación actualiza simultáneamente el estado global de la aplicación mediante Zustand, completando así el ciclo de configuración del servidor Subsonic y habilitando el acceso a la biblioteca musical remota.

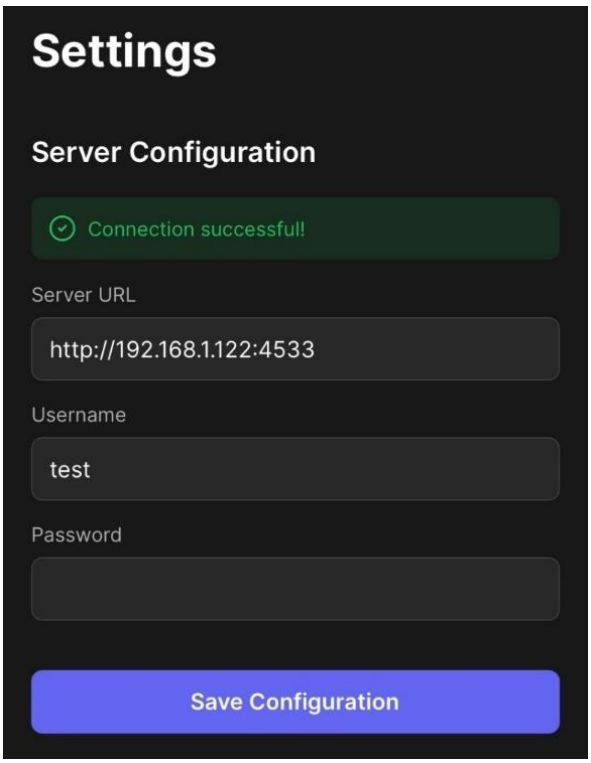

Figura 5: Conexión satisfactoria al servidor.

5.3 Gestión de caché

El sistema de gestión de caché se fundamenta en una arquitectura híbrida que combina almacenamiento de archivos mediante expo-file-system con una base de datos SQLite normalizada para metadatos. La configuración inicial del sistema se presenta a través de la pantalla mostrada en la Figura 6, donde los usuarios pueden establecer el tamaño máximo de caché en gigabytes mediante controles intuitivos de incremento y decremento, complementados con un campo de entrada numérica editable que permite ajustes precisos. Esta configuración se almacena utilizando AsyncStorage y determina el límite máximo para el directorio unificado CACHE_DIRECTORY, que centraliza tanto archivos de audio como imágenes de portada.

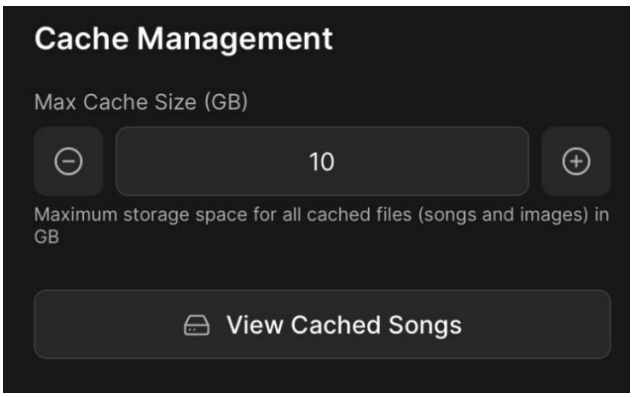

Figura 6: Configuración de caché.

Para soportar esta funcionalidad de manera eficiente, la arquitectura de datos subyacente implementa la estructura normalizada representada en la Figura 7. El diseño comprende cuatro tablas principales: artists, albums, songs y genres, donde la tabla songs actúa como entidad central estableciendo relaciones mediante claves foráneas (artistId, albumId y genre) hacia las demás tablas. Esta normalización facilita las consultas para navegación offline y optimiza las operaciones de búsqueda.

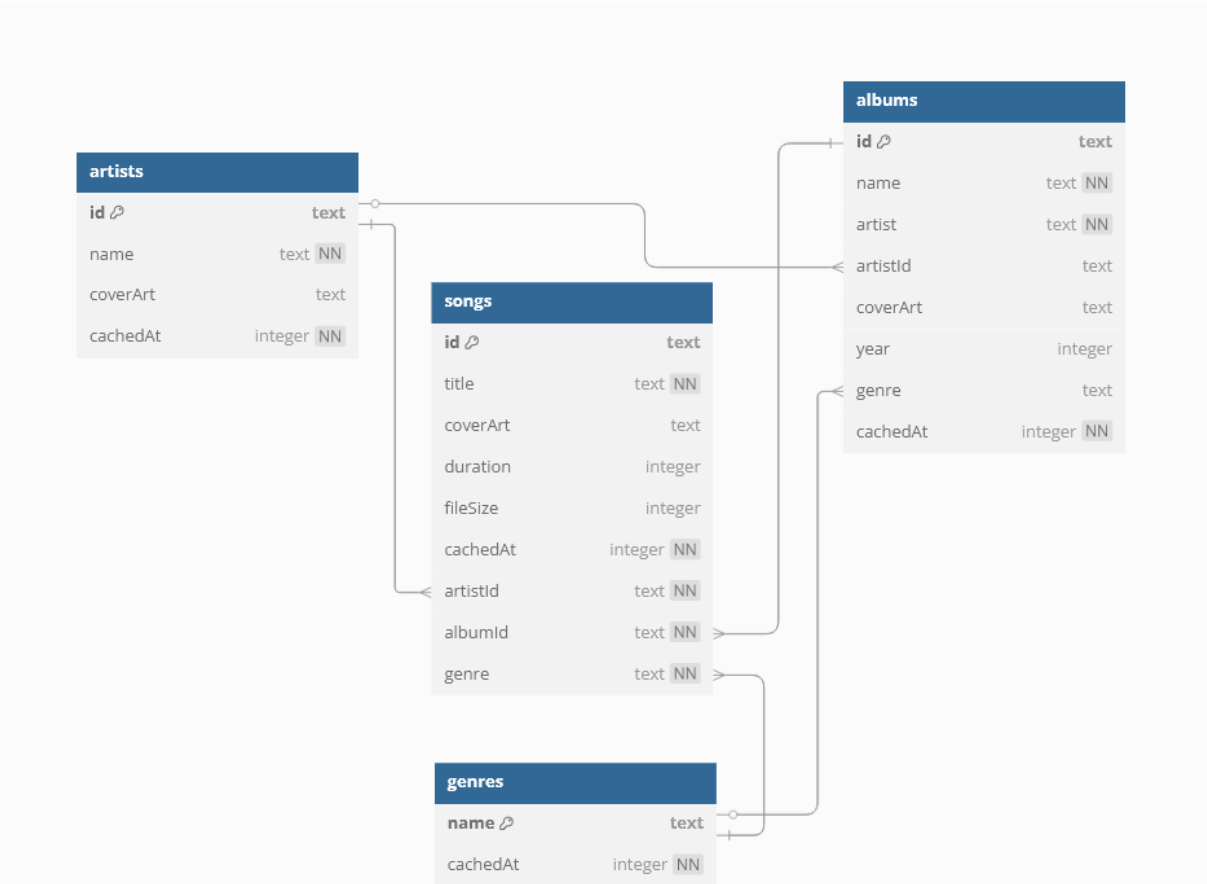

Figura 7: Diagrama de tablas.

En cuanto al proceso de descarga, este sigue la secuencia estructurada detallada en la Figura 8, comenzando con la evaluación del modo offline y continuando con la verificación de espacio disponible mediante hasEnoughCacheSpace(), función que compara el tamaño actual del caché más el archivo a descargar contra el límite configurado por el usuario. Cuando el espacio resulta insuficiente, el sistema activa automáticamente freeUpCacheSpace(), que implementa una estrategia de limpieza basada en antigüedad consultando la base de datos para obtener canciones ordenadas por fecha de último acceso y eliminando archivos hasta liberar el espacio necesario.

Una vez garantizado el espacio suficiente para la canción, la descarga utiliza FileSystem.downloadAsync() para obtener contenido desde el endpoint /stream de la API Subsonic, almacenando archivos de audio con formato {songId}.mp3 e imágenes como {imageId}.jpg. Paralelamente, el sistema ejecuta saveSongMetadata(), función que utiliza la información proporcionada por la API Subsonic para normalizar y almacenar metadatos en la base de datos local, verificando la existencia previa de

artistas, álbumes y géneros mediante consultas por nombre, creando registros básicos cuando es necesario, y estableciendo automáticamente las relaciones normalizadas entre entidades mientras mantiene la integridad referencial de la base de datos.

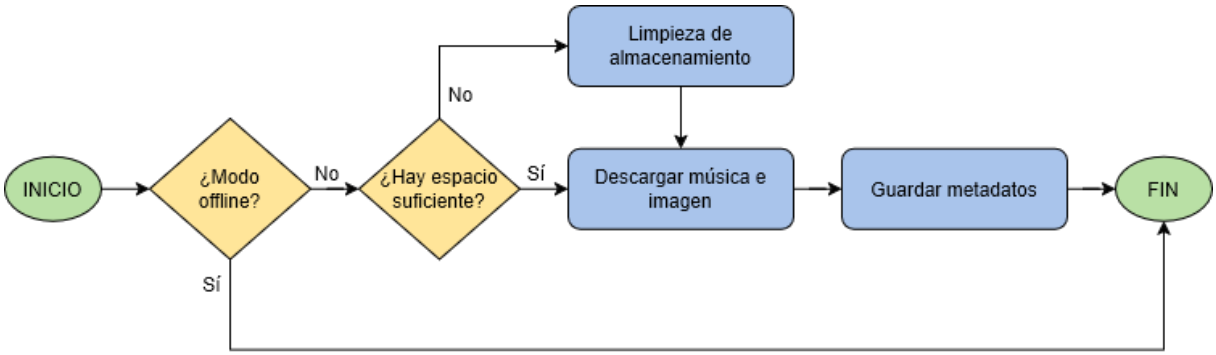

Figura 8: Flujo de descarga de canción.

Complementando estas funcionalidades, también se ha implementado una pantalla para la gestión detallada del caché, accesible mediante un botón dedicado que conduce a la vista comprehensiva ilustrada en la Figura 9.

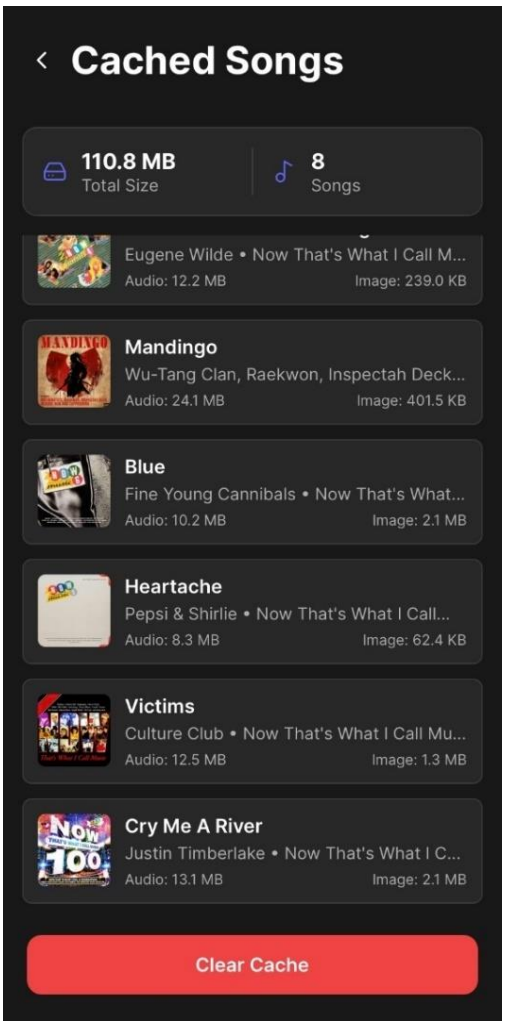

Figura 9: Gestión de caché.

Esta interfaz presenta una lista completa de todos los archivos almacenados localmente, mostrando información específica como título, artista, álbum y tamaños individuales tanto del archivo de audio como de la imagen de portada asociada. En la parte superior de la pantalla se muestran estadísticas consolidadas del caché, incluyendo el tamaño total ocupado (expresado en MB o GB según corresponda) y el número total de canciones almacenadas, proporcionando al usuario una visión general inmediata del estado de su almacenamiento local. La implementación técnica de esta funcionalidad se basa en getCachedFiles(), que consulta tanto el sistema de archivos como la base de datos SQLite para proporcionar información completa y actualizada sobre el contenido cacheado, calculando dinámicamente las estadísticas agregadas mediante la suma de los tamaños individuales de archivos y el conteo de registros en la base de datos.

Adicionalmente, la gestión de limpieza manual se facilita a través del botón "Clear Cache" visible en la Figura 9, que activa un proceso de confirmación de seguridad mostrado en la Figura 10 para prevenir eliminaciones accidentales. La implementación técnica de clearCache() ejecuta una limpieza completa eliminando el directorio completo mediante FileSystem.deleteAsync(CACHE_DIRECTORY) y ejecutando clearAllCacheMetadata() para limpiar todos los registros de la base de datos, proporcionando un reinicio total del sistema de caché.

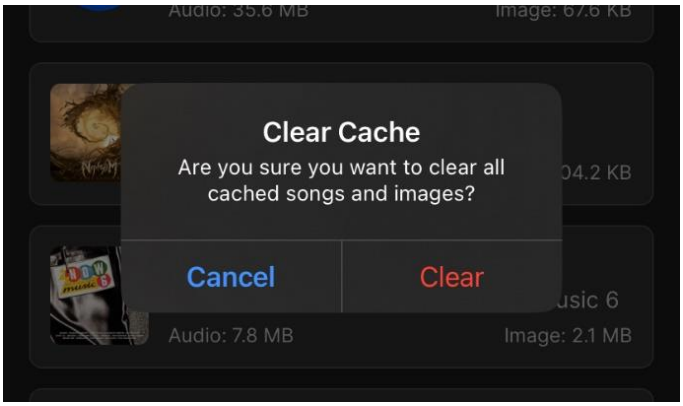

Figura 10: Mensaje de confirmación para la limpieza de caché.

5.4 Modo offline

El requisito para el modo offline es esencial para garantizar la continuidad del servicio musical cuando no existe conectividad a internet. Este sistema permite acceso a los datos almacenados en la base de datos local y archivos cacheados establecidos por el sistema de gestión de caché previamente implementado, aprovechando tanto la estructura normalizada de metadatos en SQLite como los archivos de audio e imágenes almacenados en el directorio unificado CACHE_DIRECTORY.

Para implementar esto, se ha desarrollado una configuración del modo offline que se presenta a través de la interfaz mostrada en la Figura 11, donde los usuarios pueden habilitar manualmente el modo offline mediante un interruptor intuitivo. Esta sección incluye mensajes informativos que explican el comportamiento del modo

offline, así como alertas contextuales que notifican cuando el modo se activa automáticamente debido a la pérdida de conexión a internet.

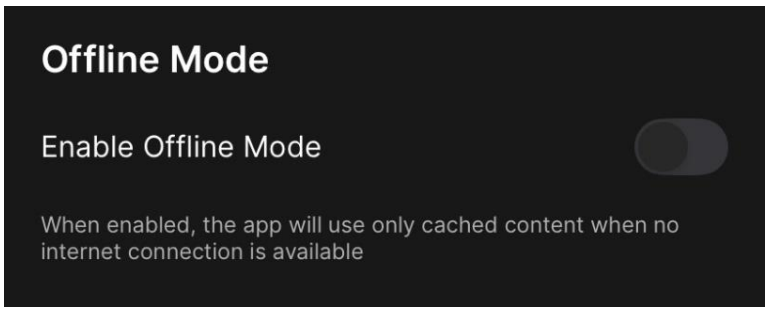

Figura 11: Configuración del modo offline.

También se ha implementado una detección automática cuyo funcionamiento técnico se fundamenta en la función initializeNetworkMonitoring(), que establece un listener continuo usando NetInfo.addEventListener() de la biblioteca @react-native-community/netinfo para detectar cambios en el estado de la red. Posteriormente, la función updateNetworkState() procesa estos cambios y determina automáticamente si debe activarse el modo offline mediante la lógica userSettings.offlineMode || hasNoInternet, asegurando que la aplicación responda inmediatamente a pérdidas de conectividad. Cuando se detecta pérdida de conexión, el sistema actualiza automáticamente las configuraciones del usuario habilitando el modo offline.

Finalmente, el sistema adapta automáticamente todas las vistas de la aplicación cuando se activa el modo offline, modificando su comportamiento para mostrar únicamente el contenido disponible localmente. Esta adaptación se realiza a través de consultas a la base de datos SQLite y verificación de archivos cacheados, garantizando que los usuarios puedan acceder a su contenido musical incluso sin conectividad a internet.

5.5 Vista de canciones

Este requisito define una vista principal que muestra una lista de canciones obtenidas del servidor Subsonic en un orden aleatorio, permitiendo al usuario interactuar de manera dinámica con el contenido. Esta vista actúa como la pantalla de inicio de la aplicación, donde el usuario puede visualizar una selección de canciones al entrar, seleccionar cualquiera para iniciar la reproducción de música y recargar la lista mediante un gesto como deslizar hacia arriba para obtener un nuevo orden aleatorio.

Para cumplir con este requisito, la implementación se fundamenta en la interacción con el endpoint /getRandomSongs de la API Subsonic, que devuelve una selección aleatoria de canciones según criterios especificados. Este endpoint permite parámetros como size para limitar el número de canciones retornadas, con un máximo de 500. Inicialmente se probó con size=500, pero se observó que la carga era lenta, especialmente en redes de baja velocidad, por lo que se optó por un valor más conservador como size=100 para la carga inicial, equilibrando la eficiencia y la responsividad.

En términos de diseño visual, como se muestra en la Figura 12, la interfaz adopta una distribución que organiza la lista de canciones en tarjetas individuales. Cada tarjeta muestra detalles clave como el título, artista, álbum y duración, facilitando la lectura rápida y la selección. La estructura visual se compone de tres secciones principales: una imagen de portada, información textual de la canción organizada verticalmente, y controles de acción que incluyen la duración formateada y un botón de reproducción circular.

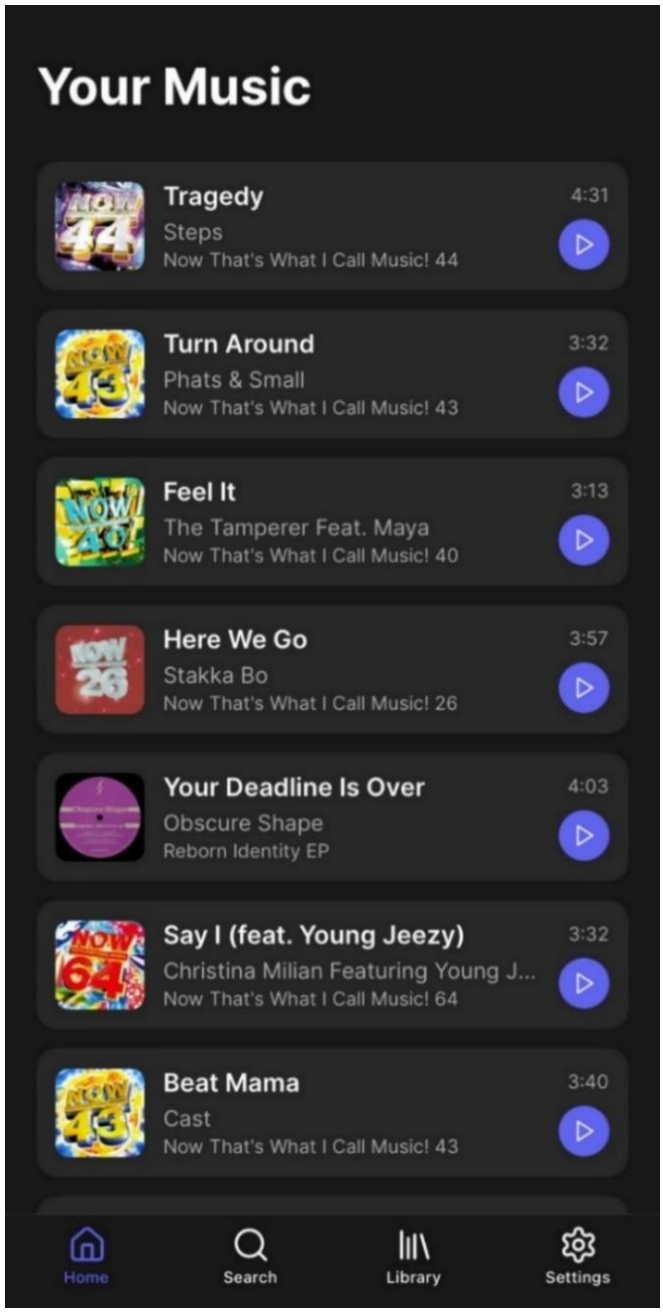

Figura 12: Lista de canciones.

Desde el punto de vista de la interactividad, la interfaz incorpora elementos que mejoran la experiencia del usuario, incluyendo un RefreshControl que responde al gesto de deslizamiento hacia arriba para generar un nuevo conjunto de canciones. Este control ejecuta la función onRefresh, que limpia la lista existente mediante

clearSongs() y solicita una nueva selección aleatoria, proporcionando feedback visual durante el proceso de actualización.

Adicionalmente, el sistema implementa una estrategia de carga progresiva mediante el hook onEndReached del componente FlatList, que detecta cuando el usuario se aproxima al final de la lista y ejecuta automáticamente la función handleLoadMore. Esta función realiza una nueva llamada al endpoint con size=50, agregando más canciones al conjunto existente y proporcionando una experiencia de navegación infinita similar a un paginado continuo, sin requerir acciones adicionales del usuario. Para gestionar la duplicación de contenido, inherente al carácter aleatorio de las respuestas del servidor, se mantiene un conjunto (Set) de identificadores únicos de canciones en el estado fetchedSongIds. Este mecanismo filtra y descarta cualquier elemento repetido antes de actualizar la lista, asegurando que solo se agreguen canciones nuevas.

No obstante, cuando se activa el modo offline o cuando no hay conexión a internet, la aplicación adapta automáticamente su comportamiento para mostrar únicamente el contenido disponible localmente. Esta adaptación se implementa técnicamente a través de la función getAllCachedSongs(), que ejecuta la consulta SQL mostrada en la Figura 13, utilizando JOIN para combinar información de las tablas normalizadas previamente establecidas en el sistema de gestión de caché.

```
`SELECT s.*, a.name as artistName, al.name as albumName
 FROM songs s
 JOIN artists a ON s.artistId = a.id
 JOIN albums al ON s.albumId = al.id
 ORDER BY s.cachedAt DESC`
```

Figura 13: Consula SQL para obtener canciones disponibles localmente.

Esta consulta permite recuperar todos los metadatos necesarios para mostrar las canciones, aprovechando las relaciones establecidas mediante claves foráneas entre las tablas songs, artists y albums. Consecuentemente, esta implementación garantiza que la vista mantenga la misma estructura de datos que en modo online, pero alimentándose exclusivamente de la base de datos SQLite local en lugar de realizar peticiones HTTP al servidor Subsonic. Como resultado de esta adaptación técnica, la interfaz se modifica visualmente según se muestra en la Figura 14, donde se observa un indicador visual prominente de "Offline" junto al título, un banner informativo que notifica sobre la falta de conexión a internet, y la lista filtrada de canciones cacheadas disponibles para reproducción sin conexión, proporcionando así una experiencia de usuario consistente independientemente del estado de conectividad.

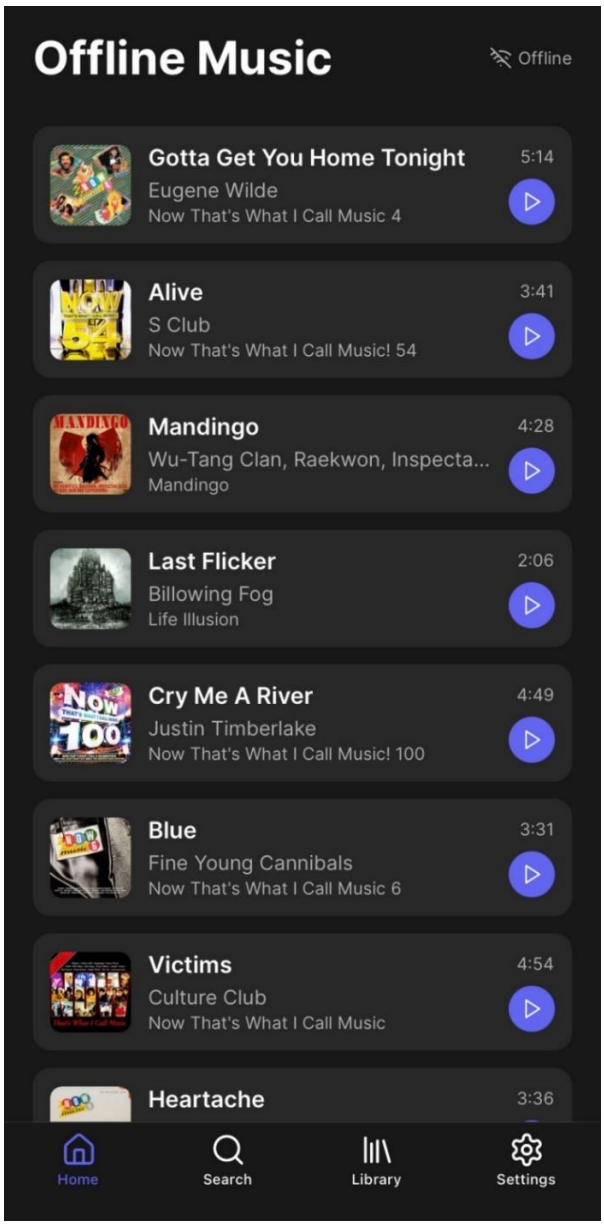

Figura 14: Vista de canciones con modo offline activado.

5.6 Reproducción de audio

El requisito de reproducción de audio permite al usuario iniciar la reproducción de canciones directamente al presionar en una canción específica o al activar el botón "Play All" en las distintas vistas de la aplicación. Como se observa en la Figura 15, el diseño integra un mini reproductor flotante en la barra inferior que muestra la portada, título, artista y controles de reproducción, junto con una barra de progreso superior que indica el avance de la canción actual, asegurando accesibilidad constante sin interrumpir la navegación.

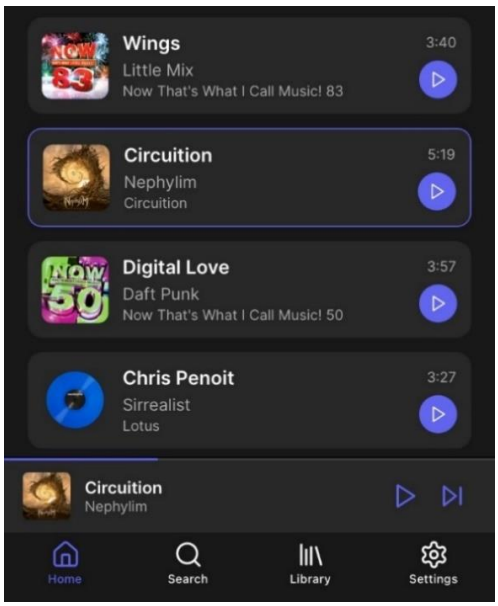

Figura 15: Mini reproductor.

Desde el punto de vista técnico, el funcionamiento de este requisito involucra un proceso centrado en la creación y gestión de sesiones de playback mediante la biblioteca expo-audio, junto con el endpoint /stream de la API Subsonic. Como se ilustra en la Figura 16, el flujo de reproducción sigue una secuencia estructurada que evalúa el contexto de reproducción, gestiona los recursos de audio y establece la configuración necesaria para una experiencia continua.

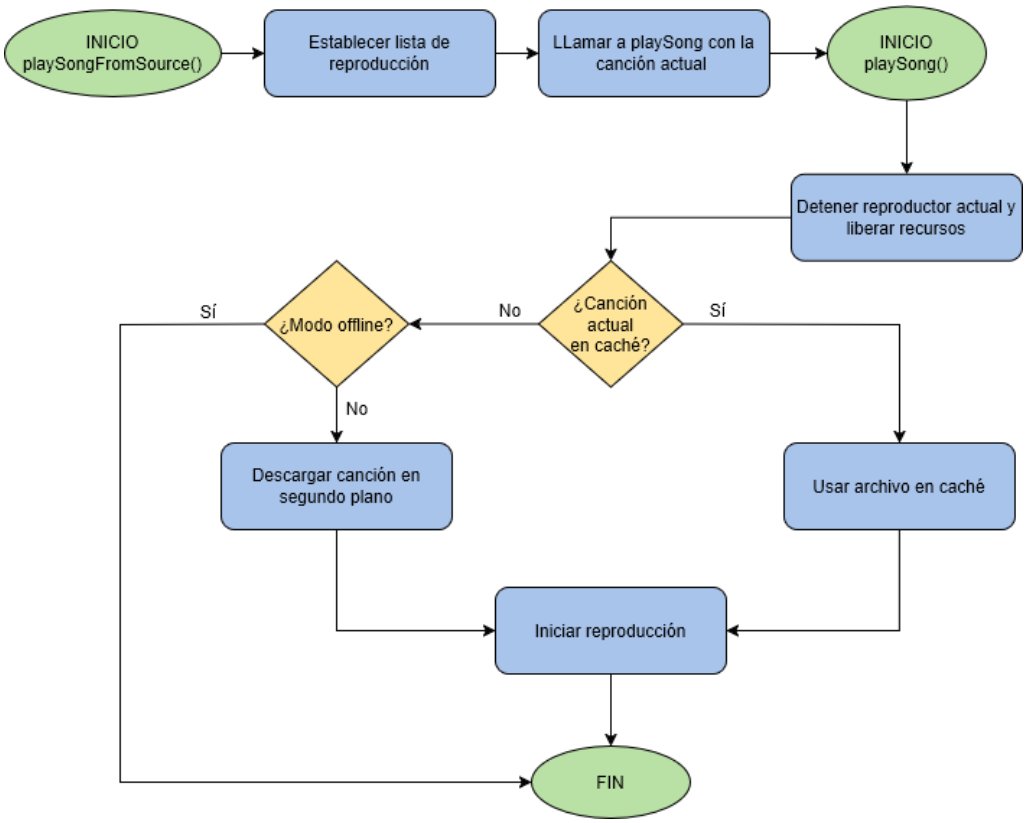

Figura 16: Flujo de reproducción de audio.

La función playSongFromSource() actúa como punto de entrada principal, recibiendo una canción específica junto con la lista completa de canciones disponibles. Esta función establece el contexto de reproducción actualizando el estado currentSongsList, que mantiene un registro de la lista de reproducción activa. Una vez configurado este contexto, delega la reproducción individual a la función playSong(), asegurando que el sistema tenga conocimiento de la lista completa para operaciones posteriores como navegación secuencial.

Por su parte, la función playSong() constituye el núcleo de la reproducción de audio, implementando un proceso estructurado que aprovecha las capacidades de expo-audio para gestionar la reproducción musical. El proceso inicia con la liberación de recursos del reproductor anterior mediante player.pause() y player.remove(), evitando conflictos de audio y liberando memoria. En cuanto a la gestión de fuentes de audio, el sistema implementa una estrategia híbrida que prioriza el contenido en caché cuando está disponible. Si la canción no está almacenada localmente, utiliza el endpoint /stream, construyendo solicitudes como /stream.view?id=<songId>&<authParams> para asegurar la transmisión autenticada del audio. Esta flexibilidad permite reproducir contenido tanto desde el servidor remoto como desde archivos almacenados localmente, optimizando el rendimiento y reduciendo el uso de datos.

Finalmente, la configuración del reproductor utiliza createAudioPlayer(audioSource) para crear la instancia de reproducción, seguido de setAudioModeAsync() con parámetros críticos como playsInSilentMode: true y shouldPlayInBackground: true, permitiendo que la reproducción continúe en segundo plano o en modo silencioso. El sistema también implementa listeners para playbackStatusUpdate que monitorizan el progreso de reproducción y detectan automáticamente el final de las canciones mediante didJustFinish, activando la función skipToNext() para continuar con la siguiente canción en la lista establecida por currentSongsList.

5.7 Vista de playlists

El requisito para la vista de playlists permite al usuario visualizar una lista de playlists disponibles desde el servidor Subsonic, con detalles como el nombre, el número de canciones y el propietario. Esta vista facilita la exploración temática de contenido agrupado por el propio usuario, permitiendo seleccionar una playlist para ver sus canciones específicas y reproducirlas directamente.

Para implementar esta funcionalidad, se ha utilizado el endpoint /getPlaylists de la API Subsonic, que devuelve todas las playlists accesibles para el usuario autenticado. Este endpoint permite un parámetro opcional username para recuperar playlists de otro usuario si se tiene rol de administrador, aunque en esta implementación se utiliza para el usuario autenticado, limitando así la respuesta a playlists relevantes y mejorando la eficiencia al evitar consultas innecesarias. El diseño de la vista se centra en una interfaz modular que prioriza la visualización clara y la interacción directa, como se muestra en la Figura 17, donde la pantalla presenta una lista de tarjetas para cada playlist, incorporando una portada (o ícono si no está disponible),

el título de la playlist, y el número de canciones. Esta disposición facilita una navegación intuitiva, con cada elemento diseñado para resaltar la información esencial en un formato escalable y visualmente atractivo.

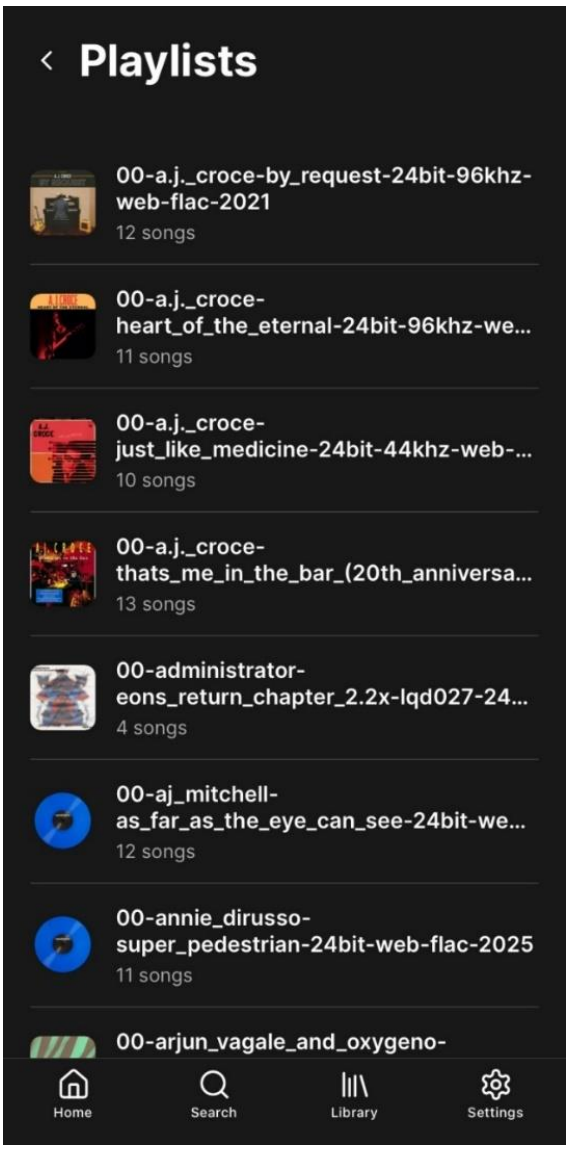

Figura 17: Lista de playlists.

Al presionar en una de las tarjetas de playlist, el usuario es redirigido a la vista detallada mediante el uso del endpoint /getPlaylist, que requiere el parámetro id de una playlist y devuelve la lista de canciones asociadas, permitiendo una carga asíncrona que maneja errores como configuraciones de servidor faltantes. Como se presenta en la Figura 18, esta pantalla de detalles incluye la portada de la playlist en la parte superior, seguido del número de canciones, la duración total de la playlist (calculada y mostrada en minutos), y una lista de canciones que conforman la playlist, con detalles como título, artista y duración individual.

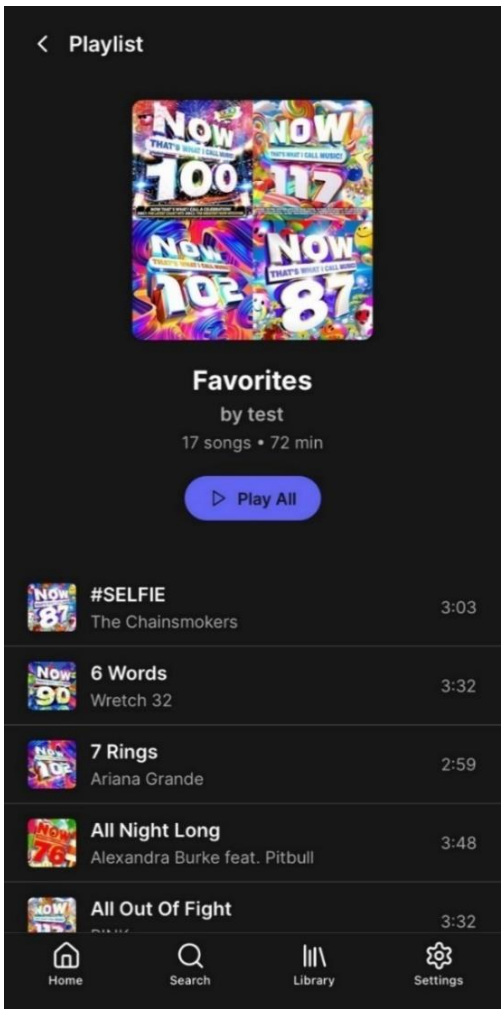

Figura 18: Detalles de una playlist en específico.

La pantalla incorpora un botón "Play all" que permite iniciar la reproducción de toda la playlist comenzando por la primera canción de forma secuencial. Esta funcionalidad recopila todas las canciones de la playlist y reproduce la primera, iniciando una secuencia continua para las restantes mediante la función handlePlaySong(), asegurando una reproducción fluida. Adicionalmente, el usuario puede seleccionar cualquier canción de la playlist para iniciar la reproducción directamente desde esa pista, proporcionando flexibilidad en la experiencia de reproducción.

5.8 Vista de álbumes (R04)

El requisito para la vista de álbumes permite al usuario explorar y acceder a una lista de álbumes desde el servidor Subsonic. Esta vista permite la navegación temática por álbumes, permitiendo seleccionar uno para ver sus canciones detalladas y reproducirlas, lo que enfatiza la organización por colecciones musicales. Como puede verse en la Figura 19, la pantalla exhibe una cuadrícula de tarjetas, cada una con la portada del álbum (o un ícono alternativo), el nombre del álbum y el nombre del artista, facilitando la selección intuitiva y una transición directa a la vista detallada al presionar un elemento.

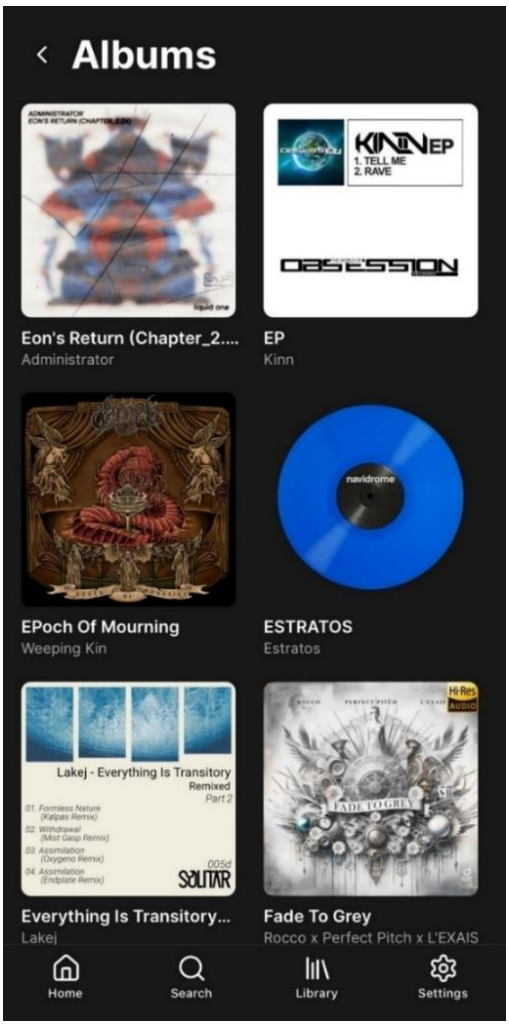

Figura 19: Lista de álbumes.

La lógica técnica detrás de la vista de álbumes depende del endpoint /getAlbumList2 de la API Subsonic, que devuelve una lista de álbumes organizada por ID3 tags, los cuales son metadatos estandarizados en archivos de audio (como MP3) para almacenar información como título, artista y álbum [23], facilitando así una organización eficiente basada en estos datos. Este endpoint utiliza el parámetro type para determinar el tipo de orden, con opciones como "starred", "alphabeticalByName" o "alphabeticalByArtist"; en esta aplicación, se emplea "alphabeticalByName" para ordenar los álbumes alfabéticamente por nombre. Por el contrario, cuando se activa el modo offline, la vista adapta su funcionamiento para trabajar exclusivamente con datos locales mediante una consulta SQL que obtiene la lista de álbumes cacheados ordenados alfabéticamente desde la base de datos SQLite local.

En la Figura 20, se presenta la pantalla de detalles del álbum, que se muestra cuando el usuario selecciona un álbum específico desde la vista principal. Esta pantalla incluye la portada ampliada del álbum, el número de canciones, la duración total, y una lista de canciones con detalles como título, artista y duración. Al igual que en la pantalla de playlist, esta vista incorpora un botón "Play All" prominente que permite iniciar la reproducción de todo el álbum comenzando por la primera pista de forma

secuencial. Los usuarios también pueden seleccionar cualquier canción específica de la lista para iniciar la reproducción directamente desde esa pista.

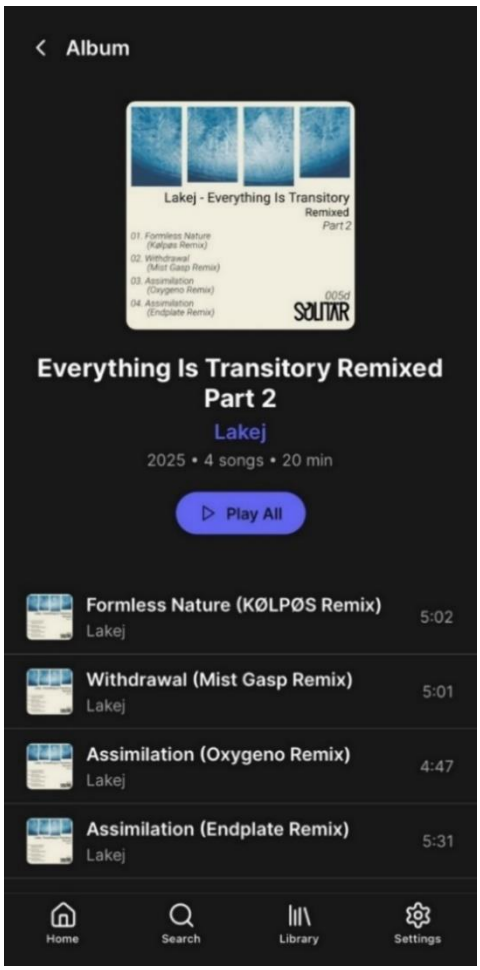

Figura 20: Detalles de un álbum en específico.

Para los detalles específicos de cada álbum, se emplea el endpoint /getAlbum, que requiere el parámetro id de un álbum para obtener la lista de canciones asociadas. En modo offline, esta funcionalidad se implementa a través de la función getCachedAlbumDetails(), que ejecuta la consulta SQL optimizada mostrada en la Figura 21, utilizando LEFT JOIN para combinar las tablas albums, songs y artists en una sola operación.

```
`SELECT
    al.id as albumId, al.name as albumName, al.artist as albumArtist,
    al.coverArt as albumCoverArt, al.year, al.genre as albumGenre,
    s.id, s.title, s.coverArt, s.duration, s.fileSize, s.cachedAt, s.artistId, s.albumId, s.genre,
    a.name as artistName
 FROM albums al
 LEFT JOIN songs s ON s.albumId = al.id
 LEFT JOIN artists a ON s.artistId = a.id
 WHERE al.id = ?
 ORDER BY s.title`,
```

Figura 21: Consulta SQL para obtener detalles de álbum.

Esta consulta permite recuperar tanto la información completa del álbum como todas sus canciones asociadas con los nombres de artistas correspondientes en una sola operación. La implementación utiliza una función de ventana para calcular dinámicamente el número total de canciones del álbum, garantizando la precisión de los datos. Esta optimización mejora el rendimiento al evitar múltiples consultas separadas y proporciona una experiencia de navegación fluida en modo offline.

5.9 Vista de artistas (R05)

El requisito para la vista de artistas permite al usuario navegar por una lista de artistas desde el servidor Subsonic. Esta vista permite la exploración por creadores musicales, utilizando metadatos para organizar el contenido de manera jerárquica y facilitar la transición a las vistas de detalles de álbumes o canciones. Tal como se presenta en la Figura 22, la pantalla principal exhibe una lista de elementos, cada uno con el nombre del artista, una portada o ícono representativo, y el conteo de álbumes, lo que permite una selección rápida. Cuando el usuario presiona cualquier elemento de artista, se produce una transición fluida hacia la vista de detalles correspondiente.

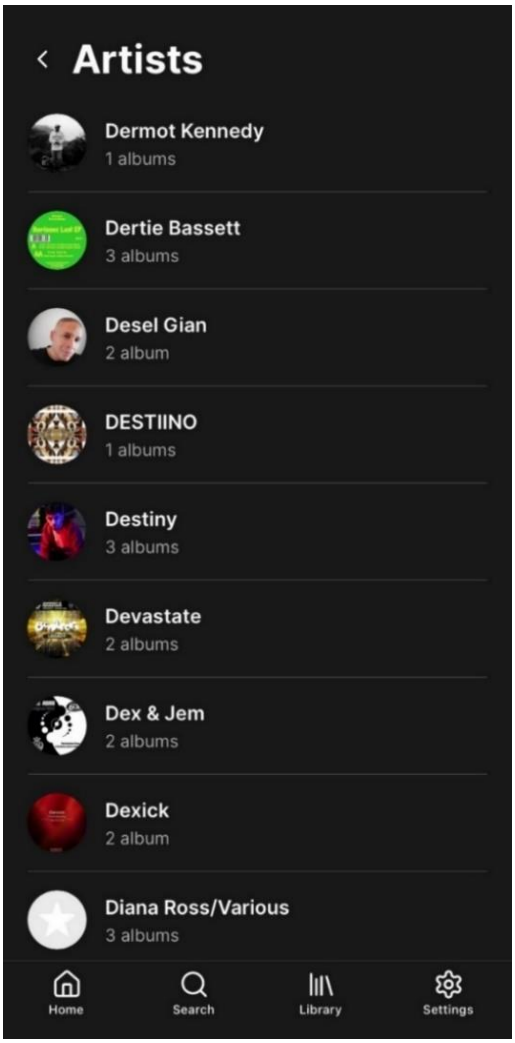

Figura 22: Lista de artistas.

El funcionamiento técnico de la vista de artistas se basa en el endpoint /getArtists de la API Subsonic, que devuelve una lista de artistas organizada por ID3 tags. Este endpoint no incluye parámetros para especificar el orden, por lo que en la implementación se ordena localmente en orden alfabético. En contraste, cuando se activa el modo offline, la vista adapta su funcionamiento utilizando la función getAllCachedArtists(), que ejecuta una consulta simple de SQL para recuperar todos los artistas almacenados en la base de datos SQLite local ordenados alfabéticamente por nombre.

La Figura 23 muestra la pantalla de detalles de un artista, que se presenta cuando el usuario selecciona un artista específico desde la vista principal. Esta pantalla presenta una imagen del artista junto con su nombre y el número total de álbumes disponibles. La interfaz incluye un botón "Play All" prominente que permite reproducir todas las canciones del artista de forma continua. Debajo se muestra una cuadrícula visual de los álbumes del artista, cada uno con su información relevante. Los usuarios pueden interactuar con cualquiera de estos álbumes para acceder a su vista detallada, donde podrán explorar las pistas individuales y obtener información más específica sobre cada álbum.

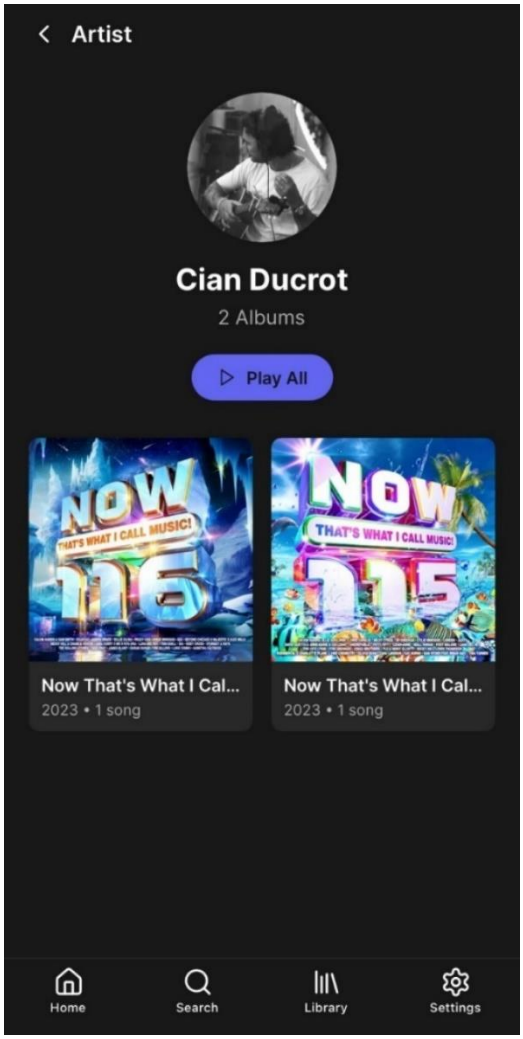

Figura 23: Detalles de un artista en específico.

Para los detalles específicos de cada artista, se emplea el endpoint /getArtist que requiere el parámetro id para obtener la lista de álbumes asociadas. En la vista detallada del artista, el botón "Play All" facilita la reproducción de todas las canciones del artista; esto se implementa recopilando las canciones de los álbumes relevantes mediante el endpoint /getAlbum, y luego iniciando la reproducción desde la primera canción en la lista, gestionando una cola para una secuencia continua. En modo offline, esta funcionalidad se implementa a través de la función getCachedArtistDetails(), que ejecuta la consulta SQL optimizada mostrada en la Figura 24, utilizando LEFT JOIN para combinar las tablas de artistas, álbumes y canciones en una sola operación.

```
`SELECT
    a.id            as artistId,
    a.name          as artistName,
    a.coverArt      as artistCoverArt,

    al.id           as albumId,
    al.name         as albumName,
    al.artist       as albumArtist,
    al.coverArt     as albumCoverArt,

    s.id            as songId,
    s.title         as songTitle,
    s.coverArt      as songCoverArt,
    s.duration      as songDuration
FROM artists a
LEFT JOIN albums al ON al.artistId = a.id
LEFT JOIN songs s ON s.albumId = al.id
WHERE a.id = ?
ORDER BY al.name, s.title`,
```

Figura 24: Consulta SQL para obtener detalles de artista.

5.10 Vista de géneros (R06)

Este requisito permite categorizar las canciones según su género musical, facilitando la exploración temática y ayudando al usuario a descubrir contenido basado en preferencias como rock, pop o jazz. Esta vista utiliza metadatos para organizar y mostrar géneros con su conteo de canciones, permitiendo una navegación intuitiva hacia listas detalladas. Como se muestra en la Figura 25, la pantalla muestra una cuadrícula de géneros, cada uno con el nombre y el número de canciones asociadas, lo que facilita la selección rápida y visual. Cuando el usuario presiona cualquier elemento de género, se produce una transición hacia la vista detallada correspondiente.

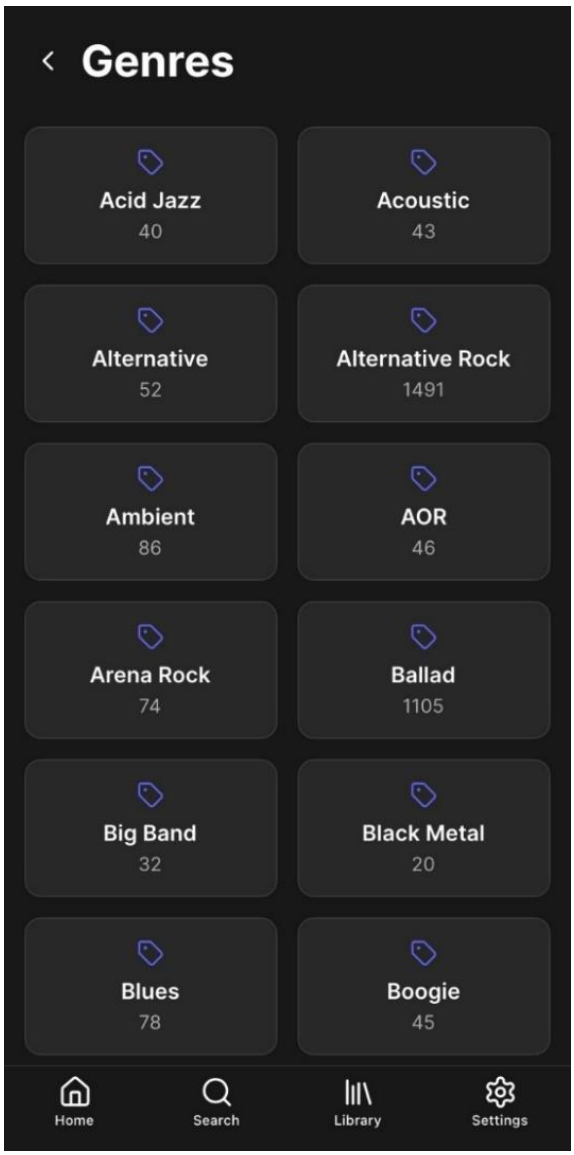

Figura 25: Lista de géneros.

La vista de géneros, desde el punto de vista técnico, se basa en el endpoint /getGenres de la API Subsonic, que devuelve una lista de géneros disponibles para el usuario autenticado, incluyendo el nombre y el conteo de canciones, lo que facilita una categorización temática basada en metadatos. Este endpoint, al igual que /getArtists, no incluye parámetros para especificar el orden, por lo que se ha decidido implementarlo localmente. En contraste, cuando se activa el modo offline, la vista adapta su funcionamiento utilizando la función getAllCachedGenres(), que ejecuta la consulta SELECT * FROM genres ORDER BY name para obtener la lista de géneros cacheados ordenados alfabéticamente desde la base de datos SQLite local.

La Figura 26 muestra la vista detallada de un género, que se presenta cuando el usuario selecciona un género específico desde la vista principal. Esta pantalla incluye una lista de canciones con detalles como título, artista y duración, junto con opciones de reproducción, como un botón para reproducir todas las canciones del género.

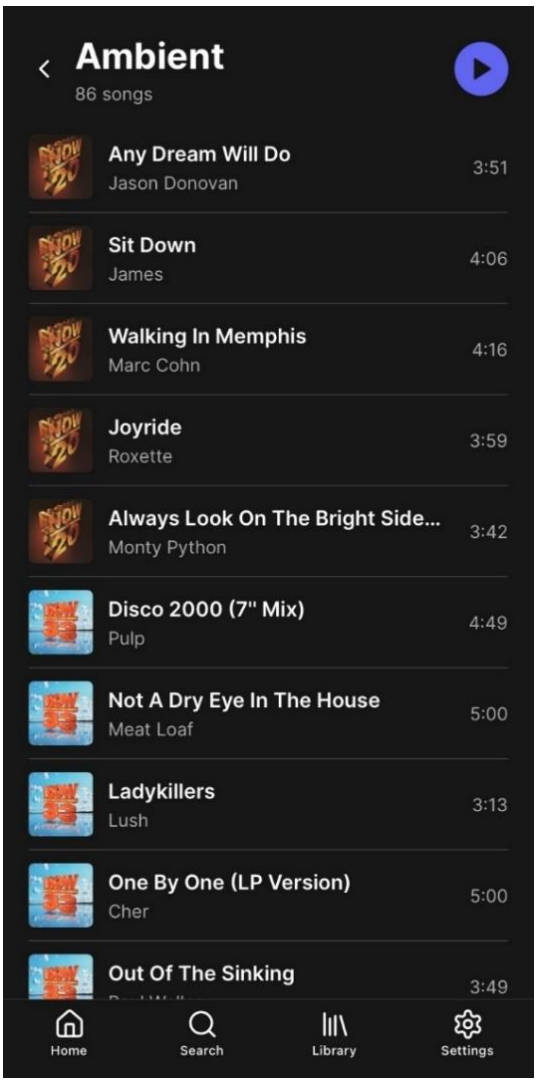

Figura 26: Detalles de un género en específico.

Para obtener las canciones de un género específico, se emplea el endpoint /getSongsByGenre, que requiere el parámetro genre que acepta el nombre de un género devuelto por /getGenres. Como con las anteriores vistas detalladas, el botón "Play All" recopila todas las canciones del género y reproduce la primera, iniciando una secuencia continua para el resto. En modo offline, esta funcionalidad se implementa a través de la función getCachedSongsByGenre(), que ejecuta la consulta SQL mostrada en la Figura 27, combinando las tablas songs, artists y albums mediante JOIN para recuperar todas las canciones pertenecientes al género especificado.

```
`SELECT s.*, a.name as artistName, al.name as albumName
FROM songs s
JOIN artists a ON s.artistId = a.id
JOIN albums al ON s.albumId = al.id
WHERE s.genre = ?
ORDER BY s.title, a.name, al.name`,
```

Figura 27: Consulta SQL para obtener canciones por género.

Esta consulta permite obtener las canciones ordenadas por título, artista y álbum, proporcionando una experiencia de navegación consistente independientemente del estado de conectividad.

5.11 Vista de la lista de reproducción

Este requisito es esencial en cualquier aplicación de música, ya que proporciona una interfaz dedicada para gestionar la reproducción actual, permitiendo a los usuarios visualizar y controlar la lista de canciones en reproducción en tiempo real, ajustar la reproducción y monitorear el progreso. Desde el punto de vista del diseño, como se muestra en la Figura 28, la interfaz se centra en una presentación intuitiva y visualmente atractiva. Esta vista presenta una lista de canciones con detalles como la portada de arte, el título y el artista de cada una, destacando la canción actualmente en reproducción para una fácil identificación. Además, incluye una barra deslizante para controlar el progreso de la reproducción, indicando simultáneamente la duración de la canción, junto con una serie de botones para acciones como reproducir/pausar, avanzar/retroceder, repetir, barajar y cambiar la velocidad de reproducción.

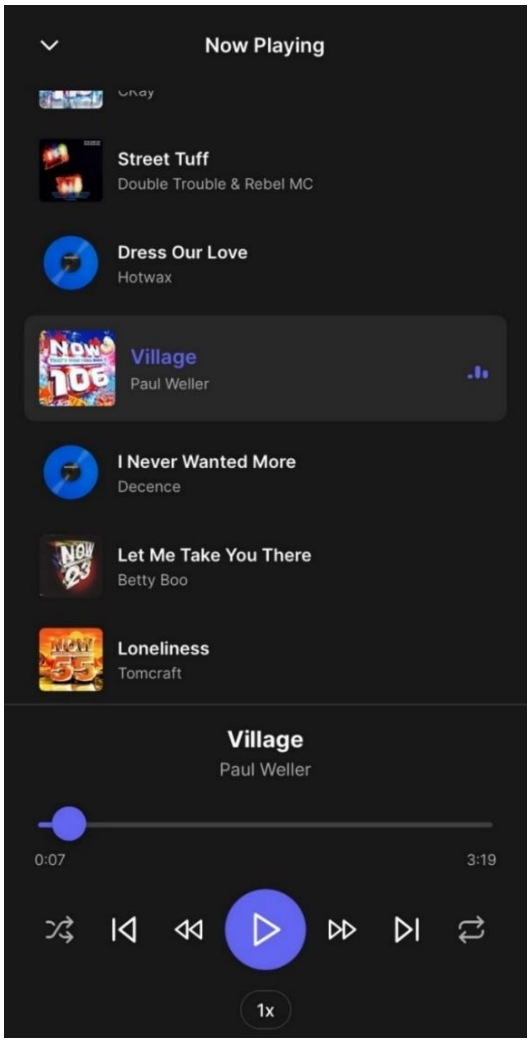

Figura 28: Lista de reproducción.

En cuanto a la implementación técnica, el funcionamiento se basa en una integración profunda con las funcionalidades de reproducción descritas anteriormente, donde se aprovechan funciones como playSong() y playSongFromSource() para inicializar y gestionar el audio. Sin embargo, esta vista enfatiza la interacción avanzada con controles específicos, profundizando en funciones como seekToPosition(), toggleRepeat() y toggleShuffle() para una gestión más detallada de la reproducción.

Para la navegación dentro de una canción, la función seekToPosition() constituye el núcleo del control de posición, recibiendo un parámetro positionSeconds que especifica la ubicación exacta en la pista. La implementación incluye validaciones robustas que utilizan Math.min(Math.max(positionSeconds, 0), songDuration) para asegurar que la posición solicitada esté dentro de los límites válidos, evitando valores negativos o que excedan la duración total de la canción. Una vez validada, la función invoca player.seekTo(clampedPosition) del reproductor de Expo AV y actualiza inmediatamente el estado local para proporcionar retroalimentación visual instantánea en la barra de progreso.

Complementariamente, las funciones seekForward() y seekBackward() implementan navegación rápida en incrementos fijos de 10 segundos. La función seekForward() calcula la nueva posición mediante Math.min(currentTime + 10, songDuration), asegurando que no se exceda la duración total, mientras que seekBackward() utiliza Math.max(currentTime - 10, 0) para evitar posiciones negativas. Ambas funciones delegan la operación de búsqueda real a seekToPosition(), manteniendo la consistencia en la validación y actualización del estado. Estas funciones son esenciales para la interactividad de la barra deslizante en la vista, permitiendo navegación precisa tanto mediante gestos táctiles como botones de control.

Respecto a los modos de reproducción, el sistema implementa un estado repeatMode que puede adoptar valores "off", "all" o "one", gestionado a través de toggleRepeat() que cicla entre estos estados de manera secuencial. La función utiliza una estructura switch que evalúa el estado actual y determina la siguiente configuración: desde "off" pasa a "all", de "all" a "one", y de "one" regresa a "off". Cuando se activa cualquier modo de repetición, automáticamente se desactiva el modo aleatorio para evitar conflictos, asegurando comportamientos predecibles. De manera similar, toggleShuffle() controla el estado isShuffle mediante una simple negación del valor actual, pero implementa la misma lógica de exclusión mutua, desactivando la repetición cuando se habilita la reproducción aleatoria.

En relación con la navegación entre canciones, las funciones skipToNext() y skipToPrevious() implementan una lógica compleja que considera múltiples factores de reproducción. Como se ilustra en la Figura 29, el flujo de skipToNext() sigue un proceso de decisión estructurado que evalúa múltiples condiciones para determinar la siguiente canción a reproducir.

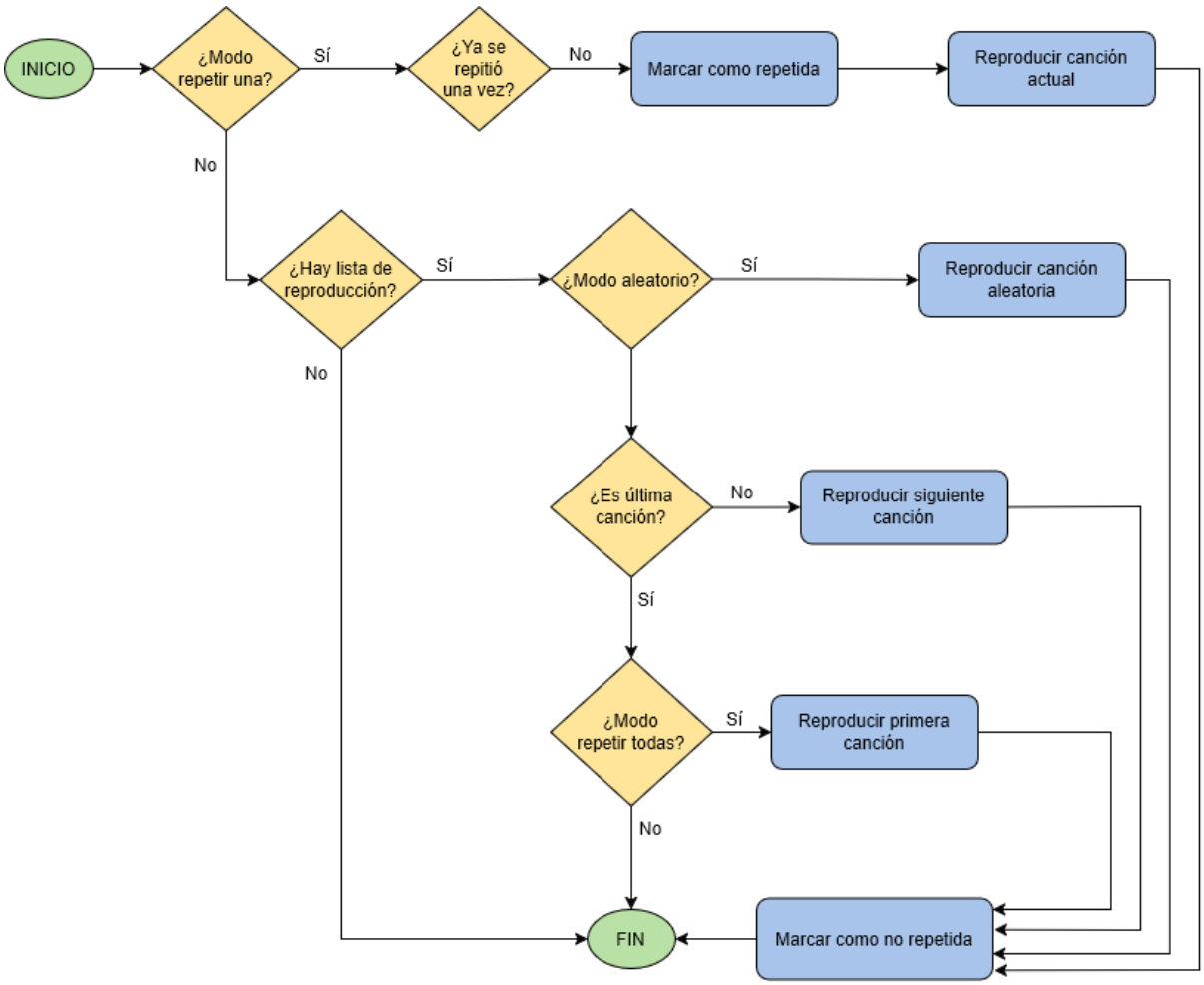

Figura 29: Flujo de navegación a la siguiente canción.

El proceso inicia evaluando el modo "repetir una" mediante una verificación de repeatMode === "one". En este caso, consulta el estado hasRepeatedOnce para determinar si la canción actual ya se reprodujo una vez adicional. Si no se ha repetido (!hasRepeatedOnce), marca hasRepeatedOnce: true y reproduce la misma canción mediante playSong(playback.currentSong). Si ya se repitió una vez, el sistema procede con la lógica normal de siguiente canción, permitiendo avanzar en la lista después de una repetición.

Posteriormente, el sistema verifica la existencia de una lista de reproducción disponible, si no existe ninguna lista válida, la función retorna sin acción. Una vez confirmada la lista, localiza el índice de la canción actual usando findIndex() con comparación de IDs. Para la selección de la siguiente canción, el sistema evalúa si está activado el modo aleatorio mediante el estado isShuffle. En modo aleatorio, crea un array filtrado availableSongs que excluye la canción actual usando filter() con comparación de índices, luego selecciona una canción aleatoria mediante Math.floor(Math.random() * availableSongs.length).

En modo normal, implementa una navegación secuencial verificando si currentIndex < songsToUse.length - 1 para determinar si es la última canción. Si no es la última, avanza al siguiente índice. Si es la última canción, evalúa si está habilitado el modo "repetir todas" (repeatMode === "all"), en cuyo caso reinicia desde el principio de la

lista seleccionando songsToUse[0]. Si no está activado el modo "repetir todas", el sistema finaliza la reproducción marcando el estado como no repetida.

Finalmente, la función setPlaybackRate() permite modificar la velocidad de reproducción aprovechando las capacidades avanzadas de Expo AV. La implementación utiliza player.setPlaybackRate(speed, "medium") donde el primer parámetro especifica la velocidad (0.75x, 1.0x, 1.25x, 1.5x) y el segundo parámetro "medium" activa la corrección de tono automática, manteniendo la calidad del audio y evitando distorsiones. Esta funcionalidad se integra con un botón que cicla entre las diferentes velocidades disponibles, proporcionando flexibilidad en la experiencia de escucha del usuario sin comprometer la fidelidad del audio.

5.12 Búsqueda (R11)

Este requisito implementa un sistema de búsqueda integral que permite a los usuarios localizar contenido de manera rápida y eficiente dentro de la aplicación. Esta funcionalidad incluye la capacidad de buscar canciones, artistas y álbumes, con resultados categorizados y organizados para facilitar la navegación. La búsqueda se actualiza dinámicamente a medida que el usuario escribe en el campo de búsqueda, proporcionando una experiencia de usuario fluida e intuitiva.

Desde el punto de vista del diseño, como se muestra en la Figura 30, la vista de búsqueda presenta un campo de entrada prominente en la parte superior de la pantalla, seguido de resultados categorizados en secciones claramente diferenciadas para artistas, álbumes y canciones. Cada categoría muestra información relevante: los artistas se presentan con iconos de usuario, los álbumes incluyen portadas y detalles como el número de canciones, y las canciones muestran portadas, títulos, artistas y controles de reproducción. La interacción es intuitiva dependiendo de la categoría, por ejemplo, al seleccionar un artista se navega a la vista detallada del artista, seleccionando un álbum se lleva a la vista del álbum detallado, y seleccionando una canción inicia su reproducción.

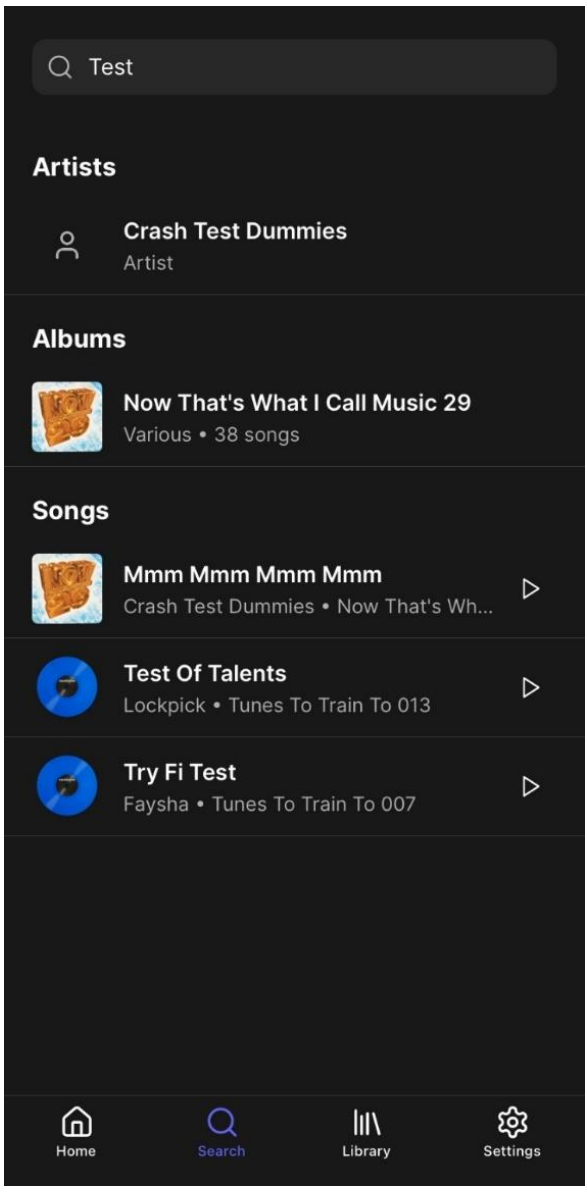

Figura 30: Ejemplo de búsqueda

En cuanto a la implementación técnica, el sistema implementa una arquitectura híbrida que soporta tanto búsqueda en línea como offline, adaptándose automáticamente al estado de conectividad de la aplicación. Para la búsqueda offline, el sistema implementa consultas SQL optimizadas que aprovechan la estructura relacional de la base de datos local. Como se muestra en la Figura 31, el sistema ejecuta tres consultas especializadas para cada tipo de contenido. La primera consulta combina las tablas de canciones, artistas y álbumes mediante JOINs para obtener información completa de las canciones, buscando únicamente por título de canción. La segunda consulta busca directamente en la tabla de artistas por nombre, mientras que la tercera consulta busca en la tabla de álbumes tanto por nombre del álbum como por nombre del artista asociado.

Todas las consultas utilizan patrones de búsqueda insensibles a mayúsculas y minúsculas que incluyen el término de búsqueda como subcadena, y los resultados

se ordenan alfabéticamente para proporcionar una presentación coherente y predecible de los resultados.

```
`SELECT s.*, a.name as artistName, al.name as albumName
 FROM songs s
 JOIN artists a ON s.artistId = a.id
 JOIN albums al ON s.albumId = al.id
 WHERE LOWER(s.title) LIKE ?
 ORDER BY s.title`,

`SELECT a.*, COUNT(DISTINCT al.id) as calculatedAlbumCount
 FROM artists a
 LEFT JOIN albums al ON al.artistId = a.id
 WHERE LOWER(a.name) LIKE ?
 GROUP BY a.id
 ORDER BY a.name`,

`SELECT al.*, COUNT(s.id) as calculatedSongCount
 FROM albums al
 LEFT JOIN songs s ON s.albumId = al.id
 WHERE LOWER(al.name) LIKE ? OR LOWER(al.artist) LIKE ?
 GROUP BY al.id
 ORDER BY al.name`,
```

Figura 31: Consulta SQL para la búsqueda de contenido.

Para la búsqueda en línea, el sistema utiliza el endpoint /search3.view de la API Subsonic, construyendo solicitudes con parámetros específicos como artistCount=20, albumCount=20 y songCount=50 para limitar los resultados y optimizar el rendimiento de la red. La URL se construye dinámicamente incluyendo la consulta codificada mediante encodeURIComponent(query) y los parámetros de autenticación generados por generateAuthParams().

Para mejorar la experiencia del usuario, se implementa un sistema de debouncing mediante setTimeout con un retraso de 500 milisegundos, evitando realizar múltiples solicitudes API mientras el usuario está escribiendo. Este mecanismo se gestiona a través de searchTimeoutRef.current que almacena la referencia del timeout activo, cancelando solicitudes previas cuando se detectan nuevos cambios en el campo de búsqueda mediante clearTimeout().

El procesamiento de respuestas incluye validación robusta del formato de datos Subsonic, manejando casos donde ciertas categorías pueden estar ausentes en los resultados. El sistema crea arrays vacíos por defecto y mapea los datos recibidos a las interfaces internas de la aplicación, transformando campos como song.title, album.name y artist.name a la estructura esperada por los componentes de UI.

Respecto a la reproducción de canciones desde los resultados de búsqueda, se utiliza la función playSongFromSource() pasando todas las canciones encontradas como contexto de reproducción mediante el parámetro sourceSongs. Esto permite que las funciones de navegación entre canciones (skipToNext() y skipToPrevious()) operen correctamente dentro del conjunto de resultados de búsqueda, estableciendo currentSongsList con los resultados y manteniendo la coherencia con el comportamiento de reproducción en otras partes de la aplicación.

6. Pruebas y Validación

La fase de pruebas y validación constituye un pilar fundamental en el ciclo de vida del desarrollo de software, garantizando que el producto final no solo cumpla con los requisitos funcionales especificados, sino que también ofrezca una experiencia de usuario robusta, estable y confiable. Para el proyecto, se diseñó e implementó una estrategia de validación integral y multinivel con el objetivo de verificar sistemáticamente cada componente de la aplicación, desde la lógica de negocio más interna hasta la interactividad de la interfaz de usuario y la compatibilidad multiplataforma.

Este capítulo detalla la metodología empleada, las herramientas seleccionadas y los resultados obtenidos, demostrando la calidad y solidez de la aplicación desarrollada a través de un enfoque riguroso y sistemático de validación.

6.1 Metodología y Estructura de Pruebas

Para asegurar una cobertura exhaustiva, la metodología de pruebas se estructuró en múltiples capas complementarias que abarcan desde la validación de componentes individuales hasta la verificación de flujos de trabajo completos. El marco de trabajo principal para la ejecución fue Jest, combinado con React Native Testing Library, que facilita la renderización de componentes y la simulación de eventos de usuario en un entorno controlado. Esta combinación tecnológica permite crear pruebas que no solo verifican la funcionalidad técnica a nivel de lógica de negocio, sino que también validan la experiencia completa desde la perspectiva del usuario final.

En el nivel más granular, las pruebas unitarias se centraron en validar la lógica implementada en el gestor de estado global de Zustand, mientras que las pruebas de integración verificaron la correcta comunicación entre diferentes módulos del sistema. Complementariamente, las pruebas de integración de la interfaz de usuario se diseñaron para simular interacciones reales del usuario, validando el flujo de trabajo completo a través de los componentes visuales y asegurando que la lógica interna se traduzca correctamente en una experiencia de usuario intuitiva y funcional.

La implementación de esta estrategia se materializa en una estructura de archivos de prueba organizada y modular, cada archivo representando una suite de pruebas, como se muestra en la Figura 32.

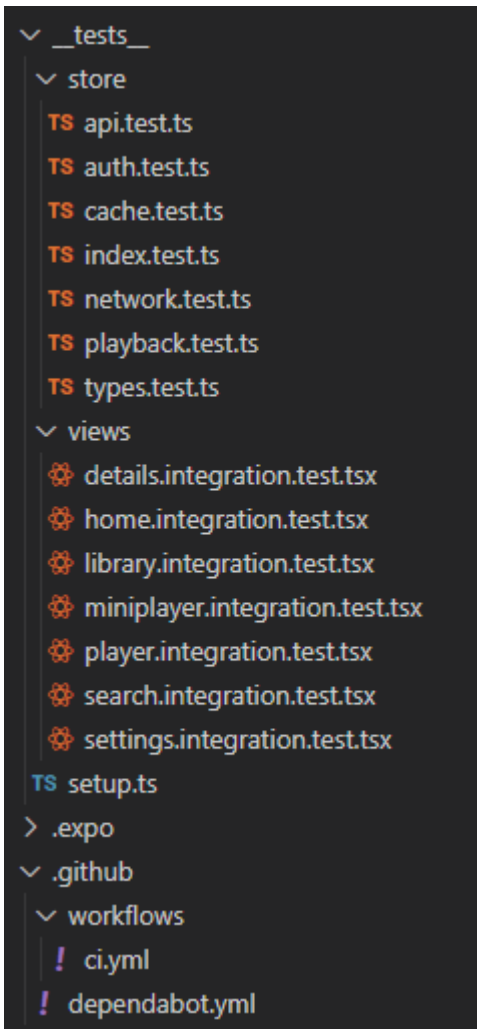

Figura 32: Estructura de los archivos de pruebas y automatización.

Un aspecto crucial de la estrategia fue la creación de un entorno de pruebas determinista y aislado. Mediante la simulación (mocking) de dependencias externas y módulos nativos, se eliminó la necesidad de contar con una conexión a servidor real, acceso al sistema de archivos o un motor de audio funcional durante la ejecución de las pruebas automatizadas. Esta configuración, centralizada en el archivo __tests__/setup.ts, resulta esencial para garantizar la rapidez y confiabilidad de las pruebas, abarcando la simulación de módulos clave como expo-secure-store, expo-file-system, @react-native-community/netinfo y expo-audio.

6.2 Validación de la Lógica de Negocio

La validación del estado global de la aplicación se estructuró mediante suites de pruebas específicas para cada slice de Zustand, totalizando 127 casos de prueba distribuidos en seis módulos principales. El módulo de API, con 32 casos de prueba, valida exhaustivamente la comunicación con el servidor Subsonic, incluyendo la construcción correcta de peticiones HTTP, el manejo de respuestas exitosas y erróneas, y la transformación de datos entre el formato de la API y el formato interno de la aplicación. Particularmente importante es la validación del manejo de errores

de la API, como se presenta en la Figura 33, donde se verifica que las respuestas con estado "failed" se procesen correctamente y que los mensajes de error se propaguen apropiadamente al usuario final.

```
it("should handle API errors", async () => {
  const errorResponse = {
    "subsonic-response": {
      status: "failed",
      error: { message: "Failed to fetch albums" },
    },
  };

  mockFetch.mockResolvedValue({
    json: () => Promise.resolve(errorResponse),
  } as Response);

  await expect(apiSlice.fetchAlbums()).rejects.toThrow(
    "Failed to fetch albums",
  );
});
```

Figura 33: Prueba de manejo de errores de la API Subsonic.

Por otra parte, el módulo de autenticación implementa 17 casos de prueba que validan el ciclo completo de gestión de credenciales, desde su almacenamiento seguro hasta la generación de tokens de autenticación. Un ejemplo crítico es la validación de que los parámetros de autenticación se construyan correctamente, como se documenta en la Figura 34, donde se verifica que la URL resultante contenga el usuario, el token hasheado, la sal y la versión de la API en el formato específico requerido por Subsonic.

```
it("should generate correct auth parameters when config exists", () => {
  mockGet.mockReturnValue({ config: mockConfig });

  const mockMathRandom = jest
    .spyOn(Math, "random")
    .mockReturnValue(0.123456789);

  const params = authSlice.generateAuthParams();

  const expectedSalt = (0.123456789).toString(36).substring(2);
  expect(mockMd5).toHaveBeenCalledWith(`testpass${expectedSalt}`);
  expect(params.toString()).toBe(
    `u=testuser&t=mocked-token&s=${expectedSalt}&v=1.16.1&c=subsonicapp&f=json`,
  );

  mockMathRandom.mockRestore();
});
```

Figura 34: Prueba de generación de parámetros de autenticación.

Asimismo, el sistema de caché, validado mediante 27 casos de prueba, asegura que la gestión del almacenamiento local sea eficiente y confiable. Estas pruebas incluyen la verificación de límites de espacio, la limpieza automática de archivos obsoletos, y

la correcta descarga y almacenamiento de contenido multimedia, como se presenta en la Figura 35.

```
it("should download and cache song when not cached", async () => {
  const mockIsFileCached = mockGet().isFileCached;
  const mockHasEnoughCacheSpace = mockGet().hasEnoughCacheSpace;

  mockIsFileCached.mockResolvedValue(false);
  mockHasEnoughCacheSpace.mockResolvedValue(true);
  mockFileSystem.getInfoAsync.mockResolvedValue({ exists: false } as any);
  mockFileSystem.makeDirectoryAsync.mockResolvedValue();
  mockFileSystem.downloadAsync.mockResolvedValue({ status: 200 } as any);

  const result = await cacheSlice.downloadSong(mockSong);

  expect(mockFileSystem.downloadAsync).toHaveBeenCalledWith(
    "http://example.com/download/songId",
    CACHE_DIRECTORY + "songId.mp3",
  );
  expect(result).toBe(CACHE_DIRECTORY + "songId.mp3");
});
```

Figura 35: Prueba de descarga y almacenamiento de canciones en caché.

En cuanto a la validación del índice del store, con 4 casos de prueba, verifica que la composición del estado global sea correcta y que la lógica de transición de modos de repetición funcione apropiadamente. Estas pruebas incluyen la verificación de que el store principal exporte correctamente todas las funciones necesarias y que las transiciones entre modos (off → one → all → off) sigan la secuencia esperada, como se muestra en la Figura 36.

```
describe("Store Logic Tests", () => {
  it("should define toggleRepeat mode transitions", () => {
    // Test the repeat mode logic directly
    const testRepeatModeTransition = (currentMode: "off" | "one" | "all") => {
      switch (currentMode) {
        case "off":
          return "one";
        case "one":
          return "all";
        case "all":
        default:
          return "off";
      }
    };

    expect(testRepeatModeTransition("off")).toBe("one");
    expect(testRepeatModeTransition("one")).toBe("all");
    expect(testRepeatModeTransition("all")).toBe("off");
  });
```

Figura 36: Prueba de transiciones de modos de repetición en el store.

Además, el módulo de gestión de red, con 10 casos de prueba, valida la capacidad de la aplicación para detectar y responder apropiadamente a cambios en la conectividad. Estas pruebas incluyen la verificación de transiciones automáticas al modo offline cuando se pierde la conectividad, la correcta inicialización del monitoreo de red, y la

gestión del estado de conexión. La validación de la transición automática al modo offline se ejemplifica en la Figura 37, verificando que la aplicación active automáticamente el modo offline cuando no hay conexión a internet disponible.

```
it("should enable offline mode when disconnected", () => {
  const networkState: NetworkState = {
    isConnected: false,
    isInternetReachable: false,
    type: null,
  };

  networkSlice.updateNetworkState(networkState);

  expect(mockSet).toHaveBeenCalledWith({
    networkState,
    isOfflineMode: true,
  });
  expect(mockSetUserSettings).toHaveBeenCalledWith({
    ...mockUserSettings,
    offlineMode: true,
  });
});
```

Figura 37: Prueba de activación automática del modo offline por desconexión.

Por su parte, la complejidad de la lógica de reproducción se refleja en las 33 pruebas del slice de playback, que validan desde la reproducción básica de audio hasta los modos avanzados de repetición y aleatorio. Un aspecto crítico validado es el comportamiento del modo de repetición "one", donde la Figura 38 demuestra que la canción se repita exactamente una vez antes de proceder a la siguiente, y que el estado interno de repetición se gestione correctamente para evitar bucles infinitos.

```
it("should handle repeat one mode", async () => {
  const mockPlaySong = jest.fn();
  mockGet.mockReturnValue({
    playback: {
      isPlaying: true,
      currentSong: mockSong,
      player: mockPlayer,
    },
    currentSongsList: mockSongs,
    repeatMode: "one",
    isShuffle: false,
    hasRepeatedOnce: false,
    playSong: mockPlaySong,
  });

  await playbackSlice.skipToNext();

  // Should set hasRepeatedOnce to true and replay the same song
  expect(mockSet).toHaveBeenCalledWith({ hasRepeatedOnce: true });
  expect(mockPlaySong).toHaveBeenCalledWith(mockSong);
```

Figura 38: Prueba de validación del modo de repetición "one".

Finalmente, la validación del sistema de tipos se realizó mediante 14 casos de prueba que aseguran la integridad y coherencia de las definiciones de tipos TypeScript utilizadas en toda la aplicación. Estas pruebas incluyen la verificación de que todas

las interfaces estén correctamente definidas, que los valores por defecto sean apropiados, y que las relaciones entre tipos sean consistentes, como se muestra en la Figura 39.

```
it("should define Song interface correctly", () => {
  const song: Song = {
    id: "test-id",
    title: "Test Song",
    artist: "Test Artist",
    album: "Test Album",
    duration: 180,
    coverArt: "cover-art-id",
  };

  expect(song.id).toBe("test-id");
  expect(song.title).toBe("Test Song");
  expect(song.artist).toBe("Test Artist");
  expect(song.album).toBe("Test Album");
  expect(song.duration).toBe(180);
  expect(song.coverArt).toBe("cover-art-id");
});
```

Figura 39: Prueba de validación de la interfaz Song en TypeScript.

6.3 Pruebas de Integración de la Interfaz de Usuario

Las pruebas de integración de la interfaz de usuario constituyen el nivel más complejo y exhaustivo de validación en aplicaciones React Native, siendo esenciales para garantizar que los componentes visuales funcionen correctamente tanto de manera aislada como en su interacción coherente con el sistema de gestión de estado. Se implementaron 125 casos de prueba distribuidos en siete módulos principales que cubren todas las pantallas de la aplicación, validando desde navegación básica hasta interacciones complejas de reproducción de audio mediante React Native Testing Library, simulando interacciones reales del usuario para verificar flujos de trabajo completos en condiciones que replican el uso real.

En primer lugar, el módulo de pantallas de detalle, con 15 casos de prueba, valida el correcto funcionamiento de las pantallas de álbum, artista, playlist y canciones por género. La complejidad de estas pruebas radica en la validación de múltiples flujos de datos y navegación, donde se verifica que la información del álbum se renderice correctamente, incluyendo el título, artista, año y lista de canciones, según se evidencia en la Figura 40.

```
it("should fetch and display album details", async () => {
  const { getByText, getAllByText } = render(
    <TestWrapper>
      <AlbumDetailsScreen />
    </TestWrapper>,
  );

  await waitFor(() => {
    expect(getByText("Test Album")).toBeTruthy();
    expect(getAllByText("Test Artist").length).toBeGreaterThan(0);
    expect(getByText("2023 • 3 songs • 9 min")).toBeTruthy();
  });
});
```

Figura 40: Prueba de renderizado de detalles de álbum.

Un aspecto crítico validado en esta suite es la reproducción de canciones desde las pantallas de detalle. La Figura 41 presenta la verificación de que al tocar una canción se active la función de reproducción con los parámetros correctos, incluyendo la canción seleccionada y la lista completa para facilitar la navegación secuencial.

```
it("should play song when song is tapped", async () => {
  const { getByText } = render(
    <TestWrapper>
      <AlbumDetailsScreen />
    </TestWrapper>,
  );

  await waitFor(() => {
    fireEvent.press(getByText("Song 1"));
    expect(defaultMockStore.playSongFromSource).toHaveBeenCalled();
  });
});
```

Figura 41: Prueba de reproducción de canción desde detalle de álbum.

Por otro lado, la pantalla de inicio, con 6 casos de prueba, valida aspectos fundamentales como el modo offline y la gestión de estados de error. Una validación crítica es el comportamiento en modo offline, donde se verifica que la aplicación muestre adecuadamente el indicador de modo offline y las opciones disponibles para el usuario, tal como se refleja en la Figura 41.

```
it("should show offline indicator when in offline mode", () => {
  mockStore.mockReturnValue({
    ...defaultMockStore,
    isOfflineMode: true,
  });

  const { getByText } = render(
    <TestWrapper>
      <HomeScreen />
    </TestWrapper>,
  );

  expect(getByText("Offline Music")).toBeTruthy();
  expect(getByText("Offline")).toBeTruthy();
});
```

Figura 42: Prueba de indicador de modo offline en pantalla principal.

Asimismo, las pantallas de biblioteca, con 22 casos de prueba, validan la correcta visualización y navegación entre diferentes categorías de contenido musical. La complejidad radica en validar múltiples tipos de listas y sus respectivas interacciones. La validación de la pantalla de artistas se presenta en la Figura 43, verificando la correcta carga de datos.

```
it("should fetch and display artists on mount", async () => {
  const { getByText } = render(
    <TestWrapper>
      <ArtistsScreen />
    </TestWrapper>,
  );

  await waitFor(() => {
    expect(defaultMockStore.fetchArtists).toHaveBeenCalled();
    expect(getByText("Test Artist 1")).toBeTruthy();
    expect(getByText("5 albums")).toBeTruthy();
    expect(getByText("Test Artist 2")).toBeTruthy();
    expect(getByText("3 albums")).toBeTruthy();
  });
});
```

Figura 43: Prueba de carga y visualización de artistas en biblioteca.

Igualmente, el componente mini reproductor, con 11 casos de prueba, valida la persistencia de información de reproducción a través de la navegación de la aplicación. Las pruebas incluyen la correcta visualización de información de canciones, manejo de texto largo y comportamiento responsive. La Figura 44 ejemplifica que el componente se renderiza correctamente cuando hay una canción en reproducción, verificando que tanto el título de la canción como el nombre del artista se muestren apropiadamente en la interfaz.

```
describe("Visibility Control", () => {
  test("should render when song is playing", () => {
    render(
      <TestWrapper>
        <MiniPlayer />
      </TestWrapper>,
    );

    expect(screen.getByText("Test Song")).toBeTruthy();
    expect(screen.getByText("Test Artist")).toBeTruthy();
  });
});
```

Figura 44: Prueba de renderizado del mini reproductor durante la reproducción.

Por su parte, el componente reproductor principal, validado mediante 32 casos de prueba, constituye el núcleo funcional más complejo de la aplicación. Las pruebas abarcan desde la renderización básica de información hasta controles avanzados como velocidad de reproducción, modos de repetición y gestión de la cola de reproducción. La validación de controles de reproducción se ilustra en la Figura 45,

verificando que los botones de play/pausa respondan correctamente al estado actual de reproducción.

```
describe("Playback Controls", () => {
  test("should toggle play/pause when play button pressed", async () => {
    render(
      <TestWrapper>
        <PlayerScreen />
      </TestWrapper>,
    );

    expect(screen.getByText("Now Playing")).toBeTruthy();

    const store = mockUseMusicPlayerStore();
    await store.pauseSong();
    expect(mockPauseSong).toHaveBeenCalled();
  });
```

Figura 45: Prueba de controles de reproducción en el reproductor principal.

Además, la gestión de la cola de reproducción requiere pruebas específicas que verifican la correcta visualización y selección de canciones. En la Figura 46 se ejemplifica la comprobación de que al seleccionar una canción de la cola se active la reproducción con la lista actual manteniendo el contexto de reproducción.

```
describe("Queue Management", () => {
  test("should play song from queue when pressed", () => {
    render(
      <TestWrapper>
        <PlayerScreen />
      </TestWrapper>,
    );

    const queueItem = screen.getByText("Another Song");
    fireEvent.press(queueItem);

    expect(mockPlaySongFromSource).toHaveBeenCalledWith(
      mockSongs[1],
      mockSongs,
    );
  });
```

Figura 46: Prueba de reproducción desde la cola en el reproductor principal.

Del mismo modo, el sistema de búsqueda, con 17 casos de prueba, valida uno de los flujos de interacción más complejos de la aplicación. Las pruebas incluyen desde la funcionalidad básica de entrada de texto hasta la debounce de búsquedas y la navegación hacia resultados. La Figura 47 demuestra la validación de la funcionalidad de debounce, crucial para optimizar el rendimiento al evitar búsquedas excesivas durante la escritura.

```
describe("Search Functionality", () => {
  it("should call searchSongs when search query changes with debounce", async () => {
    jest.useFakeTimers();

    const { getByPlaceholderText } = render(
      <TestWrapper>
        <SearchScreen />
      </TestWrapper>,
    );

    const searchInput = getByPlaceholderText(
      "Search songs, albums, or artists",
    );
    fireEvent.changeText(searchInput, "test");

    jest.advanceTimersByTime(500);

    await waitFor(() => {
      expect(defaultMockStore.search).toHaveBeenCalledWith("test");
    });

    jest.useRealTimers();
  });
```

Figura 47: Prueba de funcionalidad de búsqueda con debounce.

Complementariamente, la navegación desde resultados de búsqueda se valida mediante pruebas específicas que verifican que las selecciones dirijan correctamente a las pantallas de detalle correspondientes. La Figura 48 ilustra la comprobación de navegación hacia detalles de artista con parámetros específicos incluyendo el contexto de búsqueda.

```
it("should navigate to artist details when artist is tapped", async () => {
  const { getByPlaceholderText, getByText } = render(
    <TestWrapper>
      <SearchScreen />
    </TestWrapper>,
  );

  const searchInput = getByPlaceholderText(
    "Search songs, albums, or artists",
  );
  fireEvent.changeText(searchInput, "test");

  await waitFor(() => {
    const artistName = getByText("Test Artist");
    const artistItem = artistName.parent?.parent;
    expect(artistItem).toBeTruthy();
    fireEvent.press(artistItem!);

    expect(mockRouter.push).toHaveBeenCalledWith({
      pathname: "/(tabs)/artist-details",
      params: { id: "artist1", source: "search" },
    });
  });
});
```

Figura 48: Prueba de navegación desde resultados de búsqueda.

Por último, la pantalla de configuración, con 22 casos de prueba, valida aspectos críticos como la conexión al servidor, gestión de caché y configuraciones offline. Un

componente especialmente importante es la validación de conexión al servidor, donde se verifica tanto la validación de campos obligatorios como el proceso completo de prueba de conexión y almacenamiento de configuración, según se detalla en la Figura 49.

```
test("should test connection and save configuration", async () => {
  (global.fetch as jest.Mock).mockResolvedValue({
    json: () =>
      Promise.resolve({
        "subsonic-response": { status: "ok" },
      }),
  });

  render(
    <TestWrapper>
      <SettingsScreen />
    </TestWrapper>,
  );

  const serverUrlInput = screen.getByPlaceholderText(
    "https://your-server.com",
  );
  const usernameInput = screen.getByPlaceholderText("Username");
  const passwordInput = screen.getByPlaceholderText("Password");
  const saveButton = screen.getByText("Save Configuration");

  fireEvent.changeText(serverUrlInput, "https://test.subsonic.org");
  fireEvent.changeText(usernameInput, "testuser");
  fireEvent.changeText(passwordInput, "testpass");
  fireEvent.press(saveButton);

  await waitFor(() => {
    expect(mockSetConfig).toHaveBeenCalledWith({
      serverUrl: "https://test.subsonic.org",
      username: "testuser",
      password: "testpass",
      version: "1.16.1",
    });
  });

  expect(screen.getByText("Connection successful!")).toBeTruthy();
});
```

Figura 49: Prueba de configuración y conexión al servidor.

6.4 Resultados y Validación de Compatibilidad

La ejecución de la suite de pruebas automatizadas constituye la primera línea de validación cuantitativa del proyecto. Como se muestra en la Figura 50, la totalidad de los casos de prueba definidos fueron superados exitosamente, confirmando la solidez del código desarrollado. Los resultados demuestran que las 14 suites de pruebas, que agrupa un total de 252 casos de prueba individuales, se completaron satisfactoriamente, validando así la correcta implementación tanto de la lógica de negocio como de los componentes de interfaz de usuario.

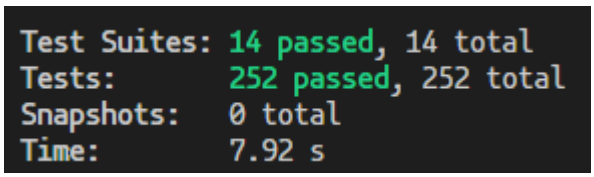

Figura 50: Resultados de la ejecución de la suite de pruebas automatizadas.

No obstante, las pruebas automatizadas se complementaron con una fase de validación manual de compatibilidad para asegurar una experiencia de usuario consistente en los dos sistemas operativos móviles dominantes. Para iOS, la aplicación fue probada en dispositivos físicos (iPhone) utilizando la aplicación cliente Expo Go, lo que permitió verificar el comportamiento en condiciones reales de uso. Para Android, se utilizó el emulador oficial proporcionado por Android Studio con múltiples configuraciones de dispositivos virtuales, permitiendo evaluar el funcionamiento de la aplicación en diferentes tamaños de pantalla y versiones del sistema operativo Android.

En ambas plataformas se verificó meticulosamente el correcto renderizado de la interfaz, prestando especial atención a la consistencia visual entre temas claro y oscuro, la fluidez de las animaciones de transición, la respuesta táctil de los controles y la correcta reproducción de audio en segundo plano. Los resultados confirmaron que la aplicación ofrece un rendimiento óptimo y una apariencia consistente en ambos ecosistemas, manteniendo la fidelidad del diseño y la funcionalidad independientemente de la plataforma utilizada.

Adicionalmente, se realizaron pruebas de rendimiento con un servidor Subsonic real conteniendo más de 1000 canciones. Estas pruebas validaron la capacidad de la aplicación para manejar bibliotecas musicales extensas, confirmando que el algoritmo implementado para mostrar canciones aleatorias funciona correctamente sin degradación del rendimiento. No obstante, se identificó que algunas pestañas como playlists y artistas experimentan una disminución en el rendimiento a medida que se incrementa el volumen de contenido, debido a que la API de Subsonic no proporciona funcionalidad de paginación para estos endpoints, requiriendo la carga completa de los datos en una sola petición. Esta limitación inherente de la API impacta en los tiempos de respuesta y en la gestión de memoria durante la navegación por colecciones musicales extensas en estas secciones específicas.

6.5 Calidad de Código e Integración Continua

Para garantizar no solo la funcionalidad sino también la mantenibilidad y calidad del código fuente a largo plazo, se integraron dos procesos fundamentales en el flujo de desarrollo. En primer lugar, se configuró un sistema de análisis de código estático mediante ESLint y Prettier, herramientas que analizan el código en tiempo de desarrollo para detectar errores comunes, identificar malas prácticas y asegurar un estilo de código homogéneo en todo el proyecto. Esta configuración incluye reglas específicas para React Native, TypeScript y las mejores prácticas de desarrollo móvil.

En segundo lugar, este proceso de validación se automatizó mediante un flujo de Integración Continua (CI) implementado con GitHub Actions, definido en el archivo ci.yml. Este flujo se ejecuta automáticamente ante cualquier cambio en las ramas principales del repositorio, realizando una secuencia estructurada de pasos que incluye la instalación de dependencias del proyecto, la ejecución del análisis estático con ESLint para identificar problemas de código, la verificación del formato con Prettier y, finalmente, la ejecución de la suite completa de pruebas con Jest.

Este sistema actúa como una red de seguridad robusta, garantizando que cualquier cambio nuevo no introduzca regresiones funcionales y que la base del código se mantenga siempre en un estado funcional y de alta calidad. Además, proporciona retroalimentación inmediata a los desarrolladores sobre el impacto de sus cambios, facilitando la detección temprana de problemas y mejorando la eficiencia del proceso de desarrollo.

7. Conclusiones y Trabajo Futuro

Este capítulo presenta las conclusiones derivadas del desarrollo del proyecto, evaluando el cumplimiento de los objetivos planteados y analizando los resultados obtenidos. Adicionalmente, se identifican las limitaciones encontradas durante el proceso y se proponen líneas de trabajo futuro para expandir las funcionalidades de la aplicación.

7.1 Conclusiones

El desarrollo ha cumplido exitosamente con los objetivos principales establecidos al inicio del proyecto, proporcionando una solución moderna y funcional para el acceso a bibliotecas musicales alojadas en servidores Subsonic. La aplicación aborda de manera efectiva las limitaciones identificadas en las aplicaciones cliente existentes, ofreciendo una experiencia de usuario superior que combina un diseño contemporáneo con funcionalidades avanzadas.

La integración con el protocolo Subsonic se implementó de manera robusta, cubriendo todas las funcionalidades esenciales como navegación por canciones, álbumes, artistas, playlists y géneros, además de un sistema de búsqueda integral que maneja eficientemente bibliotecas musicales extensas. El sistema de caché offline representa una de las contribuciones más valiosas del proyecto, proporcionando funcionalidad que muchas aplicaciones cliente de Subsonic carecen, con capacidad de descargar automáticamente canciones durante la reproducción, gestionar inteligentemente el espacio de almacenamiento y permitir reproducción sin conexión.

El pipeline de CI/CD establecido automatiza pruebas, análisis de calidad y validaciones, mientras que la suite de 266 casos de prueba desarrollada cubre tanto componentes individuales como flujos de trabajo completos, ejecutándose exitosamente y validando la estabilidad del sistema. Sin embargo, se identificaron limitaciones como la falta de paginación en algunos endpoints de la API Subsonic que resultan en degradación del rendimiento con colecciones muy extensas.

7.2 Trabajo Futuro

Para continuar mejorando la aplicación, se proponen las siguientes mejoras y funcionalidades adicionales que se pueden implementar:

- **Sistema de caché de llamadas API**: Implementar un sistema de caché inteligente para las respuestas de la API Subsonic que mitigue las limitaciones de paginación en endpoints como playlists y artistas, almacenando temporalmente los resultados.
- **Gestión completa de playlists**: Permitir a los usuarios crear, editar y eliminar playlists directamente desde la aplicación, aprovechando los endpoints correspondientes de la API Subsonic para proporcionar una experiencia de gestión musical más completa y autónoma.

- **Ecualizador integrado**: Agregar un sistema que ofrezca control granular sobre la calidad de audio según preferencias personales, permitiendo ajustes personalizados de frecuencias para optimizar la experiencia auditiva.
- **Modo de conducción**: Implementar una interfaz simplificada con controles de gran tamaño optimizados para uso seguro durante la conducción, mejorando la accesibilidad en contextos específicos de uso.
- **Compresión adaptativa**: Añadir sistemas de compresión dinámica que ajusten automáticamente la calidad de audio según la velocidad de conexión disponible, optimizando el balance entre calidad sonora y rendimiento de red para diferentes condiciones de conectividad.
- **Integración con asistentes virtuales**: Desarrollar compatibilidad nativa con Siri, Google Assistant y Alexa para control por voz de la reproducción musical, mejorando significativamente la accesibilidad mediante una experiencia de uso completamente manos libres.

8. Análisis de Impacto

El objetivo de este Trabajo de Fin de Grado es tener un impacto en los ámbitos personal, económico y social mediante el desarrollo de una aplicación cliente moderna para servidores Subsonic que promueva la descentralización del streaming musical y la privacidad de datos.

8.1 Impacto Tecnológico

El impacto personal del proyecto se centra en cómo la aplicación mejora la experiencia individual de los usuarios en el consumo de música digital. La aplicación permite a los usuarios acceder de manera eficiente a sus bibliotecas musicales personales desde dispositivos móviles, proporcionando una experiencia de streaming personalizada que se adapta a sus preferencias específicas. El sistema de caché offline garantiza que los usuarios puedan disfrutar de su música sin depender de una conexión constante a internet, mejorando significativamente la autonomía en el consumo musical. Adicionalmente, la aplicación empodera a los usuarios al proporcionarles control total sobre sus datos musicales, eliminando la dependencia de algoritmos externos y permitiendo una experiencia musical más auténtica mientras mantienen la privacidad de sus hábitos de consumo.

8.2 Impacto Económico

El impacto económico del proyecto se evalúa en términos de reducción de costes para usuarios individuales y pequeñas organizaciones. La aplicación permite a los usuarios evitar suscripciones mensuales a servicios de streaming comerciales, representando un ahorro económico significativo a largo plazo. Adicionalmente, la aplicación fomenta el ecosistema de servidores Subsonic, potencialmente generando oportunidades económicas para desarrolladores de software de servidor y proveedores de hosting especializado.

8.3 Impacto en Objetivos de Desarrollo Sostenible

Este TFG hace hincapié en el ODS 9, Industria, Innovación e Infraestructura. Este proyecto demuestra cómo las nuevas tecnologías móviles pueden utilizarse de forma innovadora para crear alternativas descentralizadas a servicios centralizados, mejorando la infraestructura tecnológica personal y promoviendo la innovación en el ámbito del streaming musical. La aplicación contribuye al desarrollo de soluciones tecnológicas que empoderan a los usuarios individuales, fomentando un ecosistema más diverso y resiliente en el consumo de contenido digital.

9. Bibliografía

[1] R. Stoner, «An Economic Analysis of The Impact of Digital Music Streaming,» Abril 2023. [En línea]. Available: https://dima.org/wp-content/uploads/2023/04/An-Economic-Analysis-of-the-Impact-of-Digital-Music-Streaming_April-2023.pdf.

[2] «Spotify Premium,» 2025. [En línea]. Available: https://www.spotify.com/us/premium/.

[3] L. Aguiar, «Let the music play? Free streaming and its effects on digital music,» *Information Economics and Policy,* pp. 1-14, 2017.

[4] J. Coalson, «What is FLAC?,» 2009. [En línea]. Available: https://xiph.org/flac/.

[5] TopoRUS, «What's yout preferred selfhosted music streaming suite?,» 2023. [En línea]. Available: https://old.reddit.com/r/selfhosted/comments/10g2bqr/whats_yout_preferred_selfhosted_music_streaming/.

[6] S. Mehus, «Subsonic Features,» [En línea]. Available: https://www.subsonic.org/pages/features.jsp.

[7] J. Willis, «Adventures in Self-hosting HiFi Audio Streaming,» 8 Diciembre 2024. [En línea]. Available: https://www.jimwillis.org/2024/12/08/adventures-in-self-hosting-hifi-audio-streaming/.

[8] DataHoards, «Moving from Spotify to a self-hosted music streaming server,» 1 Junio 2022. [En línea]. Available: https://www.datahoards.com/moving-from-spotify-to-a-self-hosted-music-streaming-server/.

[9] G. Henry, «SubStreamer,» 5 Mayo 2023. [En línea]. Available: https://apps.apple.com/us/app/substreamer/id1012991665.

[10] B. Baron, «iSub Music Streamer,» 11 Marzo 2021. [En línea]. Available: https://apps.apple.com/us/app/isub-music-streamer/id362920532.

[11] D. Hildebrand, «Amperfy Music,» 2 Diciembre 2024. [En línea]. Available: https://apps.apple.com/us/app/amperfy-music/id1530145038?platform=iphone.

[12] BLeeEZ, «[Feature request] Automatic Offline Mode + Selectable Carplay Offline Category,» 2024. [En línea]. Available: https://github.com/BLeeEZ/amperfy/issues/278.

[13] M. Hansen, «play:Sub Music Streamer,» 15 Mayo 2024. [En línea]. Available: https://apps.apple.com/ca/app/play-sub-music-streamer/id955329386.

[14] facebook, «react-native,» 3 Febrero 2025. [En línea]. Available: https://github.com/facebook/react-native.

[15] Meta, «Core Components and Native Components,» 19 Febrero 2025. [En línea]. Available: https://reactnative.dev/docs/intro-react-native-components.

[16] E. Yepis, «Developers want more, more, more: the 2024 results from Stack Overflow's Annual Developer Survey,» 1 Enero 2025. [En línea]. Available: https://stackoverflow.blog/2025/01/01/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey/.

[17] Flutter, «Intro to Dart,» 10 Enero 2025. [En línea]. Available: https://docs.flutter.dev/get-started/fundamentals/dart.

[18] Expo, «Introduction,» 1 Julio 2024. [En línea]. Available: https://docs.expo.dev/get-started/introduction/.

[19] Expo, «Refence,» 14 Abril 2025. [En línea]. Available: https://docs.expo.dev/versions/latest/.

[20] B. Moedano, «Expo Go vs Development Builds: Which should you use?,» 12 Septiembre 2024. [En línea]. Available: https://expo.dev/blog/expo-go-vs-development-builds.

[21] «TypeScript for the New Programmer,» 15 Abril 2025. [En línea]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.

[22] «Subsonic API,» [En línea]. Available: https://www.subsonic.org/pages/api.jsp.

[23] DanONeill, «ID3,» 21 Abril 2020. [En línea]. Available: https://id3.org/.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jul 02 15:51:54 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)