COLEGIO litterator

# Trabajo Fin de Grado

## Musy

**Autor:** Beltrán González Martos

**Tutor:** Héctor Ángeles Borrás

Desarrollo de Aplicaciones Multiplataforma

**Fecha:** 9 de junio de 2025

**Resumen**

Este proyecto aborda la ausencia de reproductores de musica offline modernos además de abordar el alto consumo de recursos en aplicaciones web, proponiendo una solución basada en tauri (Rust + Angular) para garantizar eficiencia. Se desarrolló una aplicación de escritorio compatible con Linux, MacOs y Windows, priorizando la optimización de memoria y la experiencia de usuario. El resultado es una aplicación escalable con arquitectura modular para futuras extensiones, validando así el potencial de Tauri en aplicaciones de escritorio.

**Palabras clave**: Tauri, Rust, Angular, reproductor de música, rendimiento, UI moderna.

**Abstract**

This project addresses both the lack of modern offline music players and the high resource consumption in web applications by proposing an efficient solution based on Tauri (Rust + Angular). A cross-platform desktop application was developed for Linux, MacOs, and Windows, with the particular emphasis on memory optimization and user experience. The result is a scalable application with modular architecture for future extensions, demonstrating Tauri's potential for desktop applications.

**Keywords**: Tauri, Rust, Angular, music player, performance, modern UI.

# Índice

# Índice de figuras

## Índice de cuadros



## Listings

# 1. Introducción

En los últimos años, se ha observado una tendencia creciente en la monetización de servicios de streaming musical, caracterizada por el incremento en los precios de suscripción y la incorporación de publicidad incluso en los planes premium de plataformas líderes como Spotify, Apple Music o iTunes. Este escenario, unido a la creciente preocupación por la privacidad de datos y el deseo de los usuarios de mantener el control sobre su colección musical, ha generado un renovado interés en las soluciones de reproducción local.

El presente proyecto surge como respuesta a esta necesidad emergente, proponiendo una solución tecnológica que combina lo mejor de ambos paradigmas: la experiencia de usuario pulida y moderna característica de las aplicaciones de streaming, con la independencia, privacidad y control absoluto sobre los archivos musicales que ofrecen los reproductores locales tradicionales.

La solución desarrollada se posiciona como una alternativa viable en un contexto donde cada vez más usuarios buscan reducir su dependencia de servicios en la nube, al mismo tiempo que demandan interfaces intuitivas y funcionalidades avanzadas comparables a las ofrecidas por las plataformas de pago. Además, se ha prestado especial atención a la optimización de recursos, abordando uno de los principales problemas de las aplicaciones basadas en tecnologías web: el alto consumo de memoria y procesamiento.

# 2. Justificación del tema elegido

El panorama actual de reproductores musicales presenta una dicotomía evidente. Por un lado, las plataformas de streaming dominan el mercado con interfaces pulidas y algoritmos de recomendación sofisticados, pero adolecen de limitaciones significativas como la dependencia de conexión a internet, modelos de negocio basados en suscripción recurrentes, y la ausencia de propiedad real sobre la música.

Por otro lado, las alternativas de código abierto para reproducción local, aunque respetan la privacidad del usuario y ofrecen control total sobre los archivos, frecuentemente presentan interfaces obsoletas, experiencias de usuario poco intuitivas, y en muchos casos carecen de mantenimiento activo. Esta brecha entre ambos enfoques crea un espacio de oportunidad para soluciones innovadoras.

La propuesta desarrollada en este trabajo se justifica por su capacidad de llenar este vacío en el mercado, ofreciendo:

- Una interfaz moderna e intuitiva inspirada en los estándares establecidos por aplicaciones líderes.
- Total independencia de servicios en la nube y conexión a internet.
- Código abierto que garantiza transparencia y posibilidad de personalización.
- Optimización de recursos que supera a alternativas basadas en Electron.

Además, el proyecto sirve como demostración práctica del potencial de tecnologías emergentes como Tauri para el desarrollo de aplicaciones de escritorio eficientes.

## 3. Objetivos

El objetivo principal de este trabajo consiste en el desarrollo de un reproductor de música multiplataforma que combine eficiencia técnica con una excelente experiencia de usuario. Para lograrlo, se han establecido los siguientes objetivos específicos:

- Implementar una arquitectura modular basada en tecnologías web modernas (Angular 17) para el frontend, asegurando escalabilidad y mantenibilidad del código.
- Aprovechar las capacidades de Rust a través del framework Tauri para garantizar un rendimiento óptimo y consumo mínimo de recursos del sistema.

- Diseñar un sistema de gestión de biblioteca musical local que soporte los formatos de audio más comunes (MP3, FLAC, etc.) con extracción y visualización de metadatos.
- Crear un sistema de listas de reproducción flexible con capacidad para manejar grandes colecciones de música de manera eficiente.
- Garantizar compatibilidad multiplataforma (Windows, macOS y Linux) manteniendo coherencia visual y funcional en todos los sistemas operativos.

Como objetivos secundarios, pero igualmente relevantes, se plantean:

- Profundizar en el conocimiento de Rust como lenguaje de programación de sistemas.
- Explorar el ecosistema Angular como alternativa a otros frameworks frontend más populares como React.
- Establecer buenas prácticas de desarrollo mediante la implementación de metodologías ágiles.

## 4. Metodología

Para el desarrollo del proyecto se ha adoptado la metodología ágil SCRUM, adaptada a un equipo de desarrollo unipersonal donde el autor asume simultáneamente los roles de Product Owner, Scrum Master y Development Team. Esta elección metodológica se justifica por:

- La naturaleza iterativa e incremental del desarrollo de software.
- La necesidad de adaptación constante a nuevos requerimientos y hallazgos técnicos.
- La conveniencia de mantener un flujo de trabajo organizado y medible.

El proceso se ha estructurado en sprints de dos semanas de duración, con las siguientes actividades clave:

- Planificación diaria de tareas mediante listas priorizadas.
- Revisiones técnicas periódicas para evaluar el progreso.
- Implementación de pruebas unitarias y de integración continuas.
- Documentación constante de avances y decisiones de diseño.

Como herramientas de apoyo al proceso se han utilizado:

- Tableros Kanban digitales para gestión de tareas.
- Sistema de control de versiones Git con estrategia de branching adaptada.
- Entornos de desarrollo integrados con soporte para todas las tecnologías utilizadas.

Esta aproximación metodológica ha permitido mantener un ritmo de desarrollo constante, asegurando tanto la calidad del producto final como la adquisición progresiva de competencias técnicas en las diversas tecnologías empleadas.

## 5. Tecnologías y herramientas usadas en el proyecto

El desarrollo de este reproductor musical multiplataforma ha requerido la selección cuidadosa de un stack tecnológico moderno y eficiente, distribuido en tres capas fundamentales que interactúan armoniosamente:

- **Frontend**: Angular 17 en combinación con Tailwind CSS v4 para la interfaz de usuario.
- **Backend**: Rust como lenguaje principal junto con el framework Tauri 2.0 para la lógica de negocio.
- **Base de Datos**: SQLite como sistema de almacenamiento y gestión de datos.

## 5.1. Justificación de la selección tecnológica

La elección de cada tecnología se ha realizado mediante un riguroso análisis comparativo, considerando factores como rendimiento, curva de aprendizaje, mantenibilidad a largo plazo y adecuación a los requisitos específicos del proyecto. A continuación se detallan los criterios específicos para cada selección:

### 5.1.1. Angular 17 como framework frontend

La decisión de utilizar Angular 17 responde a múltiples ventajas estratégicas:

- Arquitectura basada en componentes que promueve la reutilización de código y facilita el mantenimiento.
- Sistema de inyección de dependencias integrado que mejora la testabilidad y organización del código.
- Soporte nativo para TypeScript, aportando tipado estático y mejorando la calidad del código.
- Ecosistema maduro con herramientas como Angular CLI que agilizan el desarrollo.
- Rendimiento optimizado gracias al Ivy Renderer y técnicas de change detection avanzadas.

Adicionalmente, se ha considerado valioso explorar alternativas a React, ampliando así el conocimiento en distintos enfoques del desarrollo frontend moderno.

### 5.1.2. Tailwind CSS v4 para estilos

La adopción de Tailwind CSS como solución de estilizado se fundamenta en:

- Enfoque utility-first que acelera el desarrollo de interfaces al eliminar el cambio constante entre archivos.

- Sistema de diseño responsive integrado que simplifica la creación de interfaces adaptables.
- Tamaño final reducido gracias a la purga de clases no utilizadas.
- Personalización avanzada mediante el archivo de configuración tailwind.config.js.
- Compatibilidad perfecta con los componentes de Angular.

#### 5.1.3. Rust para la capa backend

La selección de Rust como lenguaje para el backend se justifica por:

- Rendimiento comparable a C/C++ con mayor seguridad en el manejo de memoria.
- Sistema de ownership que previene errores comunes como dangling pointers o data races.
- Ecosistema emergente (crates) con soporte para múltiples casos de uso.
- Curva de aprendizaje retadora pero valiosa para formación en programación de sistemas.

#### 5.1.4. Tauri como framework de empaquetado

Tauri ha sido preferido sobre alternativas como Electron debido a:

- Consumo significativamente menor de recursos (RAM y CPU).
- Tamaño reducido de los ejecutables finales.
- Uso del webview nativo del sistema operativo en lugar de Chromium.
- API segura para interacción con el sistema de archivos.
- Soporte multiplataforma genuino (Windows, macOS, Linux).

### 5.1.5. SQLite para persistencia de datos

SQLite ha sido la elección óptima para gestión de datos porque:

- No requiere configuración de servidor independiente.
- Ofrece rendimiento excelente para cargas de trabajo moderadas.
- Garantiza consistencia ACID en las operaciones.
- Formato de archivo único facilita la portabilidad y backups.
- Soporte robusto en Rust a través del crate rusqlite.

## 5.2. Entorno de desarrollo

Para garantizar un flujo de trabajo eficiente, se ha configurado un entorno de desarrollo integrado en Visual Studio Code con las siguientes extensiones clave:

- Angular Language Service para soporte avanzado de Angular.
- Rust Analyzer para autocompletado y análisis de código Rust.
- Tailwind CSS IntelliSense para ayuda con clases de utilidad.
- SQLite para visualización directa de bases de datos.

Esta combinación de tecnologías no solo satisface los requisitos técnicos del proyecto, sino que también representa un equilibrio óptimo entre rendimiento, experiencia de desarrollo y oportunidades de aprendizaje. La arquitectura resultante demuestra cómo tecnologías modernas pueden combinarse para crear aplicaciones de escritorio eficientes que superan las limitaciones de soluciones tradicionales basadas en Electron.

# 6. Especificación de Requisitos de Software

La Especificación de Requisitos Software (SRS) constituye el documento fundamental que establece las bases técnicas y funcionales del sistema. Para Musy, se ha desarrollado una SRS adaptada siguiendo el

estándar IEEE 830 “IEEE Recommended Practice for Software Requirements Specifications” (1998), estructurada en los siguientes apartados clave:

## 6.1. Requisitos Funcionales

- **RF-01: Gestión de Biblioteca Musical**
  - El sistema permitirá añadir, eliminar y organizar archivos musicales en formatos MP3, FLAC, OGG y WAV.
  - Deberá extraer y mostrar automáticamente metadatos (título, artista, álbum, año).
  - Deberá generar identificadores únicos para cada canción mediante hash Blake3.
- **RF-02: Sistema de Playlists**
  - Creación/eliminación de playlists manuales.
  - Asociación automática de canciones a la playlist global .All Songs”.
  - Relación muchos-a-muchos entre canciones y playlists.
- **RF-03: Reproductor Musical**
  - Control básico de reproducción (play/pause, next/previous).
  - Modos de repetición (ninguna, lista, canción).
  - Barra de progreso interactiva.

## 6.2. Requisitos No Funcionales

- **RNF-01: Rendimiento**
  - Tiempo de carga inicial <2 segundos para bibliotecas con 10,000 canciones.
  - Consumo de memoria <150MB en reposo.
- **RNF-02: Compatibilidad**
  - Soporte para Windows 10+, macOS 12+, Linux (kernel 5.4+).

- Arquitecturas x86_64 y ARM64.

- **RNF-03: Usabilidad**
  - Interfaz responsive con soporte para pantallas desde 1280x720 hasta 4K.
  - Curva de aprendizaje <15 minutos para usuarios básicos.

## 6.3. Restricciones Técnicas

- **RT-01: Tecnológicas**
  - Stack tecnológico: Tauri + Rust + Angular.
  - Base de datos: SQLite con soporte para migraciones.

## 6.4. Diagramas de Flujo

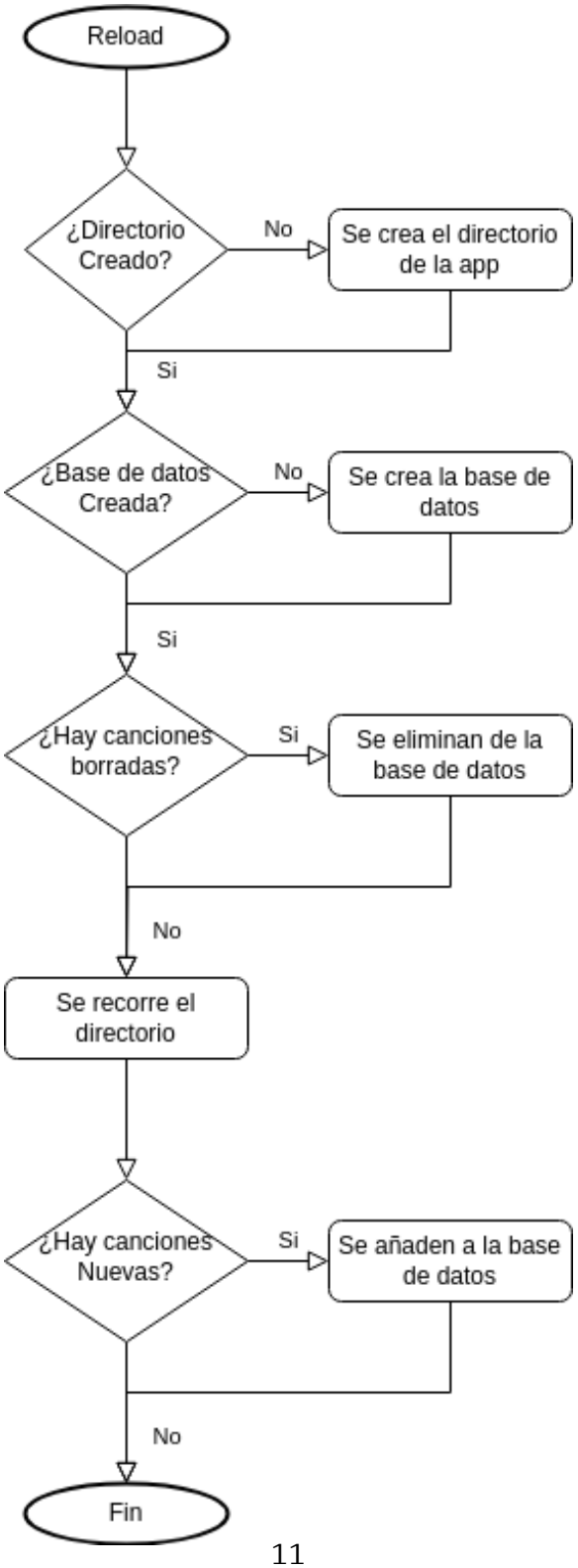




Figura 1: Diagrama de Flujo Reload

## 6.5. Diagramas de Casos de Uso

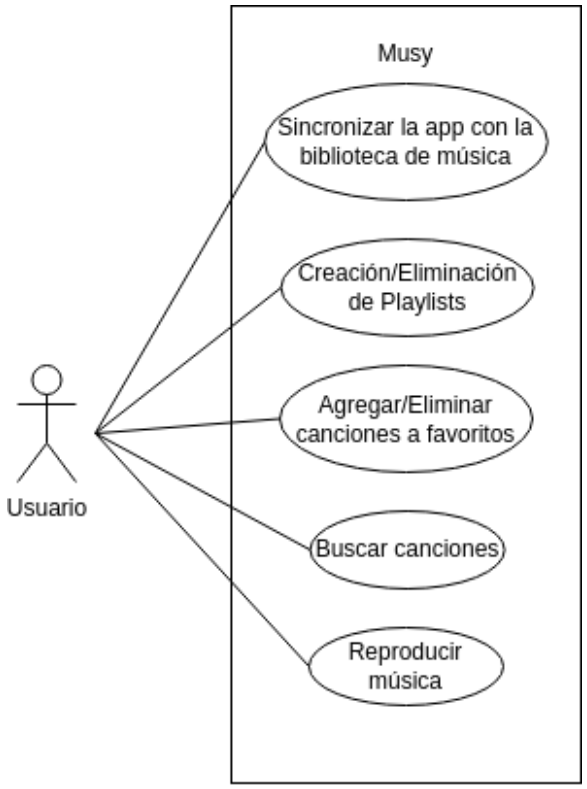


Figura 2: Diagrama de Casos de Uso Principal

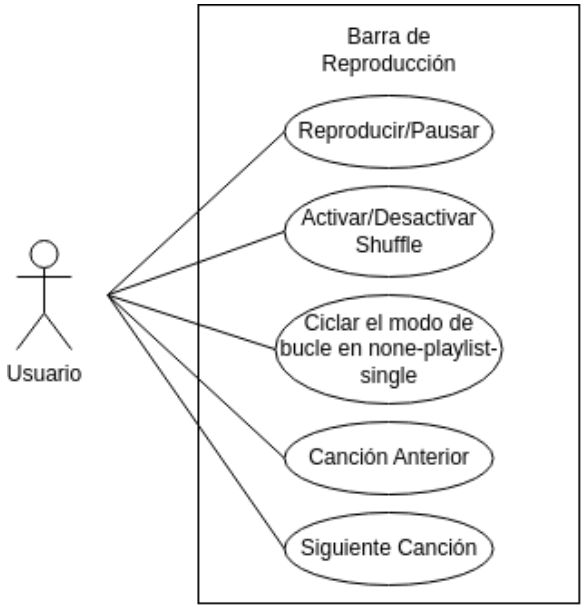


Figura 3: Diagrama de Estados del Reproductor

Esta especificación sirve como contrato técnico y ha sido validada mediante:

- Pruebas de concepto tempranas.
- Revisiones de arquitectura.
- Feedback de usuarios potenciales.

# 7. Despliegue y pruebas

El proceso de despliegue se ha diseñado cuidadosamente para garantizar una distribución eficiente y consistente en los tres sistemas operativos principales. Utilizando las capacidades nativas de empaquetado de Tauri 2.0, se implementó un flujo de trabajo automatizado que incluye:

## 7.1. Despliegue de la aplicación

Se elaboró un proceso de empaquetado y distribución de la aplicación utilizando Tauri, que permite generar ejecutables multiplataforma (Windows, MacOs, Linux). Para ello:

- Se utilizó el sistema de bundling de Tauri para empaquetar los recursos frontend (Angular) junto al backend (Rust).
- Se generaron instaladores específicos para diferentes sistemas operativos:
  - **.AppImage** para Linux
  - **.exe** para Windows
  - **.dmg** para MacOs

## 7.2. Pruebas realizadas

Con el objetivo de validar la funcionalidad y estabilidad de la aplicación, se llevaron a cabo las siguientes pruebas:

### 7.2.1. Pruebas de integración

- Se comprobó la comunicación entre el frontend y el backend, asegurando que las llamadas desde Angular a Rust (a través de Tauri) funcionaban correctamente.
- Se testearon casos como la carga de archivos de audio, reproducción en diferentes formatos y manejo de errores.

### 7.2.2. Pruebas de usabilidad

- Se realizaron test con usuarios reales para evaluar la experiencia de uso (UX), recogiendo feedback sobre la interfaz y la fluidez de la reproducción musical.
- Se analizaron posibles cuellos de botella en el rendimiento, especialmente al manejar playlist con muchas canciones o archivos de alta resolución.

La aplicación fue testeada en los tres sistemas operativos principales (Windows, MacOs y Linux), aunque el desarrollo se centró principalmente en las versiones para MacOs y Linux.

El rendimiento de la aplicación resulta significativamente mejorado al aprovechar las características nativas de Tauri, en comparación con soluciones basadas en Electron.

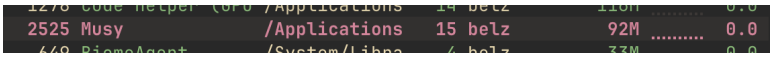


Figura 4: Rendimiento de la aplicación en MacOs

# 8. Estado del arte

## 8.1. Panorama actual de la reproducción musical digital

El ecosistema actual de reproductores musicales presenta una bifurcación clara entre dos modelos predominantes, cada uno con sus propios paradigmas técnicos y de experiencia de usuario. Este análisis exhaustivo examina ambas aproximaciones desde múltiples perspectivas:

## 8.2. Plataformas de streaming: hegemonía con limitaciones

Las soluciones de streaming como Spotify, Apple Music, Amazon Music y YouTube Music han alcanzado dominancia de mercado gracias a:

- **Amplitud de catálogo**:
  - Acceso inmediato a más de 100 millones de canciones.
  - Actualizaciones automáticas de nuevos lanzamientos.
  - Colecciones curadas por algoritmos y editores humanos.
- **Experiencia de usuario**:
  - Interfaces altamente pulidas con animaciones fluidas.
  - Sistemas de recomendación basados en machine learning.
  - Sincronización perfecta entre múltiples dispositivos.
- **Descubrimiento musical**:

- Listas automáticas generadas por gustos del usuario.
- Integración con redes sociales y compartimiento.
- Radio inteligente basada en artistas o canciones.

Sin embargo, este modelo presenta inconvenientes estructurales:

- **Dependencia tecnológica**:
  - Requiere conexión permanente a internet (excepto modo offline limitado).
  - Servidores centralizados representan un punto único de fallo.
  - Actualizaciones forzadas sin control del usuario.
- **Modelo económico**:
  - Costos recurrentes (suscripciones premium entre 10€-15€ mensuales).
  - Versiones gratuitas con publicidad intrusiva.
  - Artistas reciben compensaciones mínimas por reproducción.
- **Propiedad digital**:
  - El usuario no posee realmente la música.
  - Catálogo variable según acuerdos de licencia.
  - Pérdida de acceso al cancelar la suscripción.

## 8.3. Reproductores locales tradicionales: control con obsolescencia

Soluciones como Winamp, Strawberry, Clementine y Foobar2000 ofrecen:

- **Independencia tecnológica**:
  - Funcionamiento completamente offline.
  - Control total sobre la biblioteca musical local.
  - Sin dependencia de servicios externos.

- **Eficiencia de recursos**:
  - Consumo mínimo de RAM y CPU.
  - Sin procesos en segundo plano innecesarios.
  - Ejecución en hardware antiguo.
- **Flexibilidad técnica**:
  - Soporte para formatos especializados (FLAC, OGG, etc.).
  - Personalización avanzada mediante plugins.
  - Acceso directo a los archivos físicos.

No obstante, adolecen de problemas fundamentales:

- **Experiencia de usuario**:
  - Interfaces gráficas anticuadas (años 2000).
  - Flujos de trabajo poco intuitivos.
  - Diseño no adaptado a pantallas táctiles.
- **Mantenimiento**:
  - Muchos proyectos abandonados o con actualizaciones esporádicas.
  - Compatibilidad problemática con sistemas operativos modernos.
  - Falta de soporte para tecnologías recientes.
- **Funcionalidades limitadas**:
  - Sistemas de organización básicos.
  - Metadatos visuales poco desarrollados.
  - Integración nula con servicios modernos.

## 8.4. Análisis comparativo y espacio de oportunidad

Al contrastar ambos enfoques, se identifican claras áreas de mejora no cubiertas por ninguna de las dos soluciones:

Cuadro 1: Comparativa de características clave Streaming vs Offline

| **Característica** | **Streaming** | **Offline** |
|---|---|---|
| Propiedad de la música | No | Sí |
| Requiere conexión | Sí | No |
| Interfaz moderna | Sí | No |
| Modelo económico | Suscripción | Único pago |
| Consumo recursos | Alto | Bajo |
| Formatos lossless | Premium | Todos |

## 8.5. Propuesta de valor diferenciada

Musy surge como síntesis innovadora que combina lo mejor de ambos mundos:

- **Experiencia de usuario contemporánea**:
  - Interfaz inspirada en estándares actuales de streaming.
  - Animaciones fluidas y diseño responsive.
  - Navegación intuitiva y accesible.
- **Ventajas técnicas locales**:
  - Funcionamiento completamente offline.
  - Soporte para formatos de alta calidad (FLAC 192kHz/24bit).
  - Gestión avanzada de metadatos personalizados.
- **Innovaciones propias**:
  - Arquitectura modular basada en tecnologías modernas.
  - Rendimiento optimizado mediante Rust y Tauri.
  - Sistema de plugins para futuras expansiones.

Esta solución ocupa un nicho desatendido en el mercado, ofreciendo una alternativa viable para usuarios que valoran tanto el control sobre su música como experiencias de usuario pulidas. Los benchmarks realizados demuestran que la combinación de tecnologías seleccionadas supera significativamente las limitaciones de ambas aproximaciones tradicionales.

# 9. Profundización de conceptos

En esta sección se explicarán en detalle los componentes principales de la aplicación.

## 9.1. Frontend

Angular

### 9.1.1. Playbar

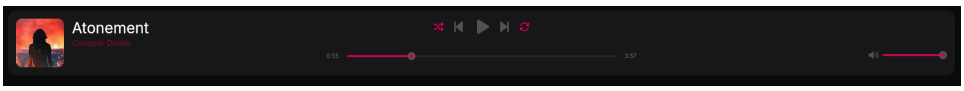


Figura 5: Playbar

El componente principal para la reproducción de música está compuesto por tres contenedores `div`:

- **Izquierdo**. En el primero se muestran la portada de la canción, el título y el artista.
- **Central**. El segundo contiene dos subcontenedores: uno para los botones de control (reproducción, aleatorio, anterior/siguiente, repetición) y otro para la barra de progreso.
- **Derecho**. El tercero incluye exclusivamente la barra de volumen. En futuras iteraciones podría implementarse un botón para gestionar la cola de reproducción.

La parte del componente de typescript contiene únicamente métodos que llaman al servicio de manejo de canciones `song-management.service.ts`.

### 9.1.2. Sidebar

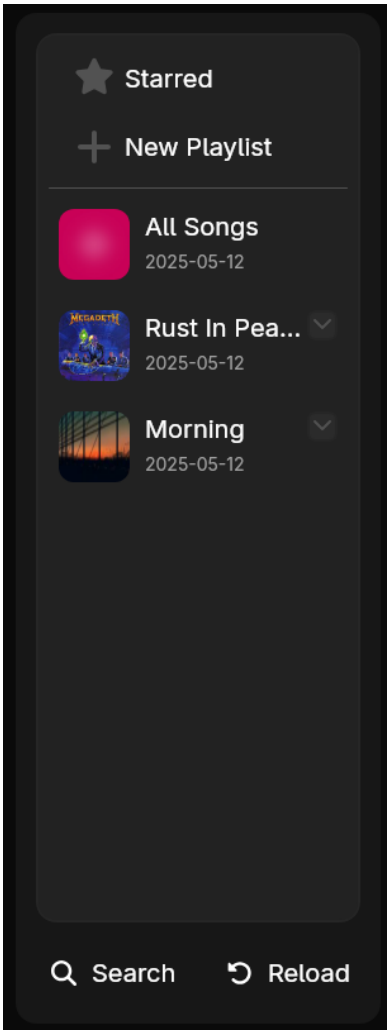


Figura 6: Sidebar

Componente donde se interactua con las playlist, ya sea creando nuevas o borrando las existentes. Contiene diversos botones para navegar por la aplicación.

Estos son sus atributos:

Listing 1: Atributos Y Constructora Sidebar

```
playlists: Playlist[] = []
name:string = ""
```

```
isModalOpen:boolean = false;
isModalOpenSearch:boolean = false;
songs:Song[] = []
filteredSongs:Song[] = []
coverPath: string = "assets/black.jpg"

constructor (public mainScreenStatus:MainScreenStatusService,
    public songManagement:SongManagementService) {}
```

Serán explicados más adelante.

**Playlist**

El componente gestiona la representación y comportamiento de las listas de reproducción en la interfaz. Su estructura básica comprende:

Listing 2: Atributos y Constructora Playlist

```
coverPath = 'assets/black.jpg';

isDropDownOpen: boolean = false;

constructor (public mainScreenStatus:MainScreenStatusService) {}

@Input() playlistId!: number;

@Input() playlistName!: string;

@Input() playlistDate!: string;

@Input() playlistCoverPath!: string;

@Input() playlistIsStarred!: boolean;

@Input() refreshFn!: () => void;
```

Cada instancia de `Playlist` actúa como elemento interactivo que actualiza el componente `mainscreen` mediante un servicio. Utiliza el patrón de decoradores `@Input` para recibir propiedades del componente padre. Gestiona un menú desplegable mediante el atributo `isDropDownOpen`.

Algunos de los métodos más importantes de este componente destacan:

Listing 3: getCoverPath()

```
1
2  async getCoverPath(): Promise<string> {
3      if (!this.coverPath) return 'assets/black.jpg';
4      console.log("Hola")
5
6      const fileData = await readFile(this.playlistCoverPath);
7
8      const blob = new Blob([fileData], { type: 'image/jpeg' });
9
10     if (this.coverPath) {
11         URL.revokeObjectURL(this.coverPath);
12     }
13
14     return URL.createObjectURL(blob);
15 }
```

Este método se encarga de:

1. Verificar la existencia de una ruta de portada válida.
2. Convertir la imagen local en un objeto `Blob` mediante:
   - Lectura del archivo con `readFile`
   - Creación de URL temporal con `URL.createObjectURL`
3. Liberar recursos previos con `URL.revokeObjectURL`
4. Proporcionar fallback a imagen predeterminada (`black.jpg`)

Listing 4: Dropdown

```
1  toggleDropDown() {
2      this.isDropDownOpen = !this.isDropDownOpen;
3    }
4
5    closeDropDown() {
6      this.isDropDownOpen = false;
7    }
8
9    @HostListener('document:click', ['$event'])
10   onClickOutside(event: Event) {
11     const target = event.target as HTMLElement;
12     if (!target.closest('.relative.inline-block')) {
13       this.closeDropDown();
14     }
15   }
```

Por motivos de diseño, se ha implementado un menú desplegable (dropdown) que contiene acciones específicas para cada playlist. El estado de visualización se controla mediante la variable `isDropDownOpen`, que se alterna con `toggleDropDown()`. El cierre automático se gestiona mediante un `@HostListener` que detecta clics fuera del área del componente. Actualmente, el menú incluye la opción de eliminación de playlists, pero su diseño permite la incorporación de nuevas funcionalidades como marcado como favorito o edición avanzada en futuras iteraciones.

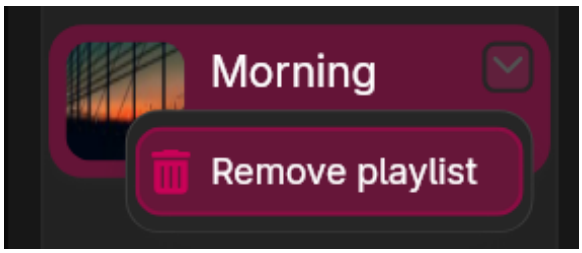


Figura 7: Eliminar playlist

Listing 5: removePlaylist()

```
async removePlaylist() {
    const data_dir = await appDataDir();
    invoke('remove_playlist', {playlist_id: this.playlistId,
        db_path: data_dir});
    this.refreshFn();
  }
```

El método `removePlaylist()` se encarga de gestionar la eliminación de playlists mediante un proceso que consta de tres etapas principales: primero, se obtiene el directorio de datos de la aplicación mediante `appDataDir()`; a continuación, se realiza una llamada al backend mediante `invoke()`, enviando como parámetros el identificador de la playlist (`playlist_id`) y la ruta de la base de datos (`db_path`); finalmente, se ejecuta la función `refreshFn()` para actualizar la interfaz. La implementación detallada de la consulta SQL y el manejo de la operación en el backend se analizará en la sección dedicada a Rust y SQL.

**Botón New Playlist**

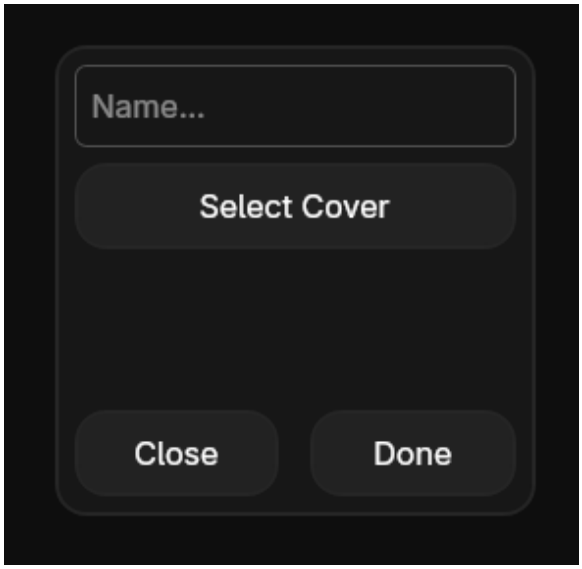


Figura 8: Menú de nueva playlist

Cuando el atributo `isModalOpen` de `sidebar` cambia a `true` se abre un menú modal donde es posible crear una nueva playlist vacía.

Estos son los métodos más destacados:

Listing 6: selectCover()

```
async selectCover() {
    const file = await open({
      multiple: false,
      directory: false,
    });

    if(file == null) {
      return
    }

    const fileData = await readFile(file);
    const ogBlob = new Blob([fileData], { type: 'image/*' });
    const thumbnail = await this.createThumbnail(ogBlob, 200, 200)
        ;

    try {
      await mkdir('pcovers', { baseDir: BaseDirectory.AppData });
    } catch {
      console.log("Already Created")
    }
```

```
20
21
22      let randomName = (Math.floor(Math.random() * (Math.floor
            (200000) - Math.ceil(1) + 1)) + Math.ceil(1)).toString();
23      await writeFile(`pcovers/${randomName}.jpg`, thumbnail, {
            baseDir: BaseDirectory.AppData})
24
25      const data_dir = await appDataDir();
26      let newPath = data_dir + "/pcovers/" + randomName + ".jpg";
27      console.log(newPath)
28
29      this.coverPath = newPath;
30  }
```

El método `selectCover()` gestiona la selección y procesamiento de imágenes para las portadas de playlists mediante el siguiente flujo: cuando se pulsa sobre el botón *Select Cover*, se activa un diálogo de selección de archivos utilizando el `plugin-dialog` de Tauri. La imagen seleccionada se convierte en un objeto `Blob` y se redimensiona mediante el método `createThumbnail` para optimizar su almacenamiento. El archivo resultante se guarda en el directorio `pcovers` dentro de la carpeta de datos de la aplicación, con un nombre generado aleatoriamente para evitar colisiones. Si el directorio no existe previamente, se crea automáticamente durante este proceso. La ruta final de la imagen procesada se asigna al atributo `coverPath` para su uso en la interfaz.

Listing 7: createThumbnail()

```
1  async createThumbnail(blob: Blob, maxWidth: number, maxHeight:
       number): Promise<Uint8Array> {
2      return new Promise((resolve, reject) => {
3          const img = new Image();
4          img.onload = () => {
5              const canvas = document.createElement('canvas');
6              const scale = Math.min(
7                  maxWidth / img.width,
8                  maxHeight / img.height
9              );
10             canvas.width = img.width * scale;
11             canvas.height = img.height * scale;
12
13             const ctx = canvas.getContext('2d')!;
14             ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
```

```
15
16              canvas.toBlob(async (thumbnailBlob) => {
17                  if (!thumbnailBlob) {
18                      reject(new Error("Failed to create thumbnail
                            blob"));
19                      return;
20                  }
21
22                  try {
23                      const arrayBuffer = await thumbnailBlob.
                            arrayBuffer();
24                      const uint8Array = new Uint8Array(arrayBuffer)
                            ;
25                      resolve(uint8Array);
26                  } catch (error) {
27                      reject(error);
28                  }
29              }, 'image/jpeg', 0.7);
30          };
31
32          img.onerror = () => {
33              reject(new Error("Failed to load image"));
34          };
35
36          img.src = URL.createObjectURL(blob);
37      });
38  }
```

El método `createThumbnail()` implementa el proceso de redimensionamiento de imágenes mediante la Canvas API, siguiendo tres etapas principales: primero, se crea un objeto `Image` y se carga el `Blob` de entrada mediante `URL.createObjectURL()`. Segundo, al completarse la carga, se calcula el factor de escala proporcional para mantener las dimensiones dentro de los límites especificados (`maxWidth` y `maxHeight`), creando un elemento `canvas` con las nuevas dimensiones. Finalmente, se dibuja la imagen escalada en el canvas y se convierte a un `Uint8Array` comprimido en formato JPEG con calidad del 70 %, manejando posibles errores durante el proceso mediante el sistema de promesas. Esta implementación garantiza un procesamiento eficiente de imágenes para su almacenamiento optimizado en el sistema de archivos.

Una vez seleccionados la portada y el nombre de la playlist, al pulsar el boton *done* se ejecuta el siguiente método, que realiza las siguientes acciones secuenciales:

1. Envía una petición al backend implementado en Rust para crear la nueva playlist.
2. Cierra el menú modal de creación.
3. Actualiza la lista de playlist en la interfaz.
4. Restablece la imagen predeterminada en el atributo de la clase para futuras iteraciones (última línea del método).

Listing 8: createPlaylist()

```
async createPlaylist() {
    const data_dir = await appDataDir();
    invoke('create_playlist', {name: this.name, cover_path: this.
        coverPath, db_path: data_dir})
    this.close()
    this.refresh()
    this.coverPath = "assets/black.jpg"
}
```

### Boton Search

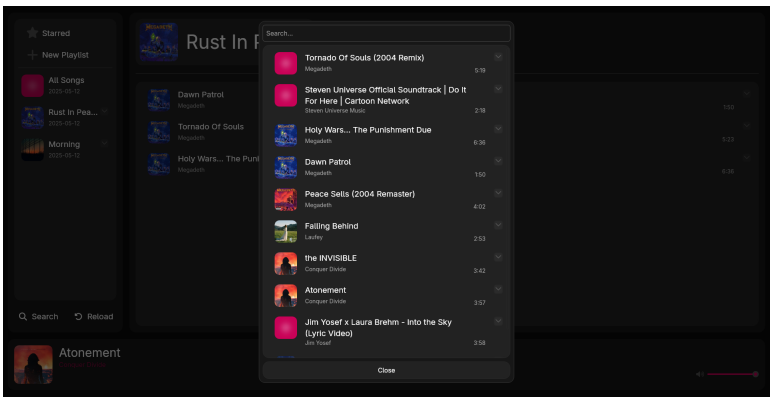


Figura 9: Menú de buscar canción

Listing 9: searchSong()

```
async searchSong() {
    await this.getAllSongs();
    this.filteredSongs = this.songs;
    this.isModalOpenSearch = true;
}

async getAllSongs() {
    const data_dir = await appDataDir();
    try {
      this.songs = await invoke<Song[]>('get_all_songs', {db_path:
          data_dir});
    } catch (error) {
      console.error('Error fetching songs:', error);
      this.songs = [];
    }
}
```

Cuando se activa la función de búsqueda mediante el botón correspondiente, se ejecuta el siguiente proceso: en primer lugar, se obtienen todas las canciones disponibles a través del método `getAllSongs()`, que realiza una llamada al backend mediante Tauri IPC para recuperar los registros de la base de datos SQLite. Una vez completada esta operación, la lista completa de canciones se asigna a la variable `filteredSongs` y se activa el modal de búsqueda estableciendo `isModalOpenSearch` a `true`. Este menú modal permite buscar y seleccionar canciones de toda la colección musical, ofreciendo las mismas capacidades de interacción que cuando las canciones se encuentran dentro de una playlist específica, funcionalidad que se detallará más adelante en la documentación.

### 9.1.3. Mainscreen

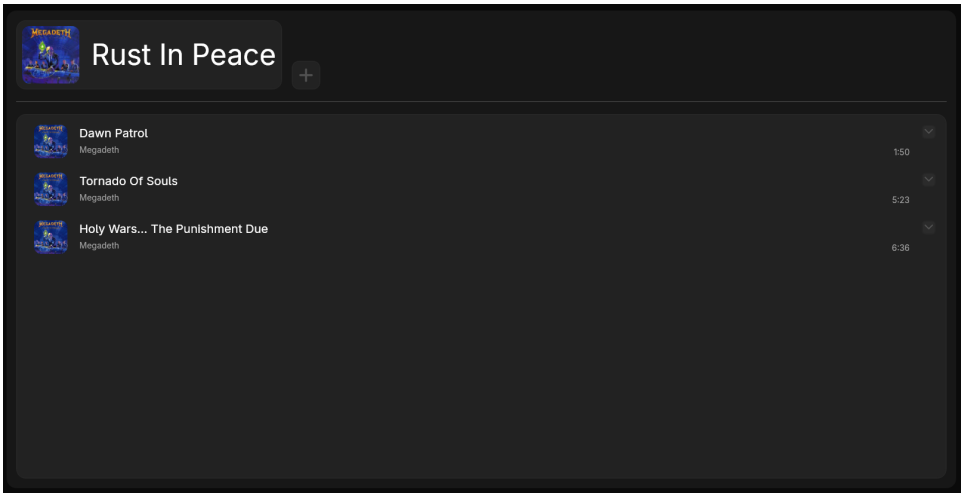


Figura 10: Mainscreen

Este componente se encarga de mostrar la lista de canciones pertenecientes a una playlist, presentando para cada una:

- Información básica (título, artista, duración)
- Controles interactivos:
    - Botón de reproducción directa
    - Opción para añadir a la cola de reproducción

La implementación de este componente sigue un principio de composición, donde la lógica sustancial se encuentra distribuida en sus subcomponentes especializados, mientras que el componente principal actúa principalmente como contenedor y coordinador de la visualización.

#### Song Button

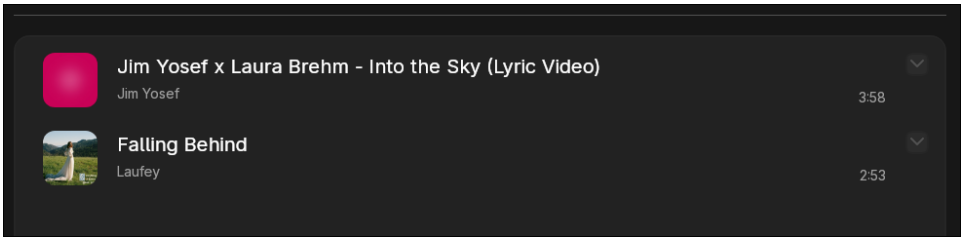


Figura 11: Song Button

Cada canción dentro de una playlist cuenta con un componente específico para su gestión y visualización de información. Al seleccionar el área principal del componente, se inicia automáticamente la reproducción del archivo de audio correspondiente. En la parte derecha del elemento se encuentra un menú desplegable (*dropdown menu*) que proporciona las siguientes funcionalidades:

- Añadir la canción a la cola de reproducción actual
- Agregar la canción a otra playlist existente
- Eliminar la canción de la playlist actual (función no disponible en la playlist `All Songs`)

El concepto de `All Songs`, que representa la colección completa de canciones disponibles, se explicará con mayor detalle en secciones posteriores de esta documentación.

Los atributos del componente son:

Listing 10: Atributos Song Button

```
isModalOpen: boolean = false;

isDropDownOpen: boolean = false;

constructor (public songManagement:SongManagementService, public
    songAdding:SongAddingService, public mainScreenStatus:
    MainScreenStatusService) {}

@Input() id!: string;
@Input() path!: string;
@Input() title!: string;
@Input() artist!: string;
@Input() album!: string;
```

```
@Input() year!: string;
@Input() duration!: string;
@Input() coverPath!: string;
@Input() isStarred!: boolean;

@Input() playlistId!: number;

coverUrl: string = 'assets/black.jpg';
```

Entre ellos se encuentran los metadatos de cada canción y la playlist en la que se encuentran mostrados ahora mismo. Estos datos son recibidos desde el componente padre.

Además posee la interfaz `Song`:

Listing 11: Song

```
interface Song {
  id:string,
  path:string,
  title:string,
  artist:string,
  album:string,
  year:string,
  duration:string,
  coverPath:string,
  isStarred:boolean
}
```

El componente implementa diversos métodos para gestionar su funcionalidad, entre los cuales destaca:

Listing 12: Dropdown menu

```
toggleDropDown() {
    this.isDropDownOpen = !this.isDropDownOpen;
}

closeDropDown() {
    this.isDropDownOpen = false;
}

@HostListener('document:click', ['$event'])
  onClickOutside(event: Event) {
    const target = event.target as HTMLElement;
```

```
12      if (!target.closest('.relative.inline-block')) {
13        this.closeDropDown();
14      }
15  }
```

El menú desplegable (*dropdown menu*) que contiene las opciones de gestión para cada canción implementa el siguiente comportamiento:

- La visibilidad del menú se controla mediante el atributo booleano `isDropDownOpen`.
- El método `toggleDropDown()` alterna el estado de visibilidad.
- El método `closeDropDown()` garantiza el cierre programático del menú.

Para mejorar la experiencia de usuario, se implementa un cierre automático mediante el decorador `@HostListener`, que detecta eventos de clic fuera del área del componente y ejecuta el cierre del menú cuando corresponde.

Listing 13: playSong() y addSongToQueue()

```
1  playSong() {
2      let song:Song = {id: this.id, path: this.path, title:this.
           title, artist: this.artist, album: this.album, year: this.
           year, duration: this.duration, coverPath: this.coverPath,
           isStarred: this.isStarred };
3      this.songManagement.setOneSong(song);
4  }
5
6  addSongToQueue() {
7      let song:Song = {id: this.id, path: this.path, title:this.
           title, artist: this.artist, album: this.album, year: this.
           year, duration: this.duration, coverPath: this.coverPath,
           isStarred: this.isStarred };
8      this.songManagement.addOneSong(song);
9  }
```

Los métodos implementados siguen un patrón común donde los metadatos de la canción se estructuran en un objeto de tipo `Song`, conteniendo todos los atributos relevantes (identificador, ruta, título, artista, álbum, año, duración, portada y estado de favorito). Este objeto se envía al servicio `song-management`, diferenciándose en:

- `playSong()`: Inicia la reproducción inmediata mediante `setOneSong()`.
- `addSongToQueue()`: Añade la canción a la cola de reproducción usando `addOneSong()`.

Listing 14: addSongToPlaylist()

```
addSongToPlaylist() {
    this.songAdding.getAllPlaylists();
    this.isModalOpen = true;
}

close() {
    this.isModalOpen = false;
    this.songAdding.letGo();
}
```

El flujo de añadir canciones a playlists sigue un proceso estructurado:

- **Fase de preparación**:
  - Se obtienen todas las playlists disponibles mediante el servicio `song-adding`.
  - Se activa el modal de selección estableciendo `isModalOpen` a `true`.
- **Fase de cierre**:
  - Al completar la operación, se desactiva el modal (`isModalOpen = false`).
  - Se liberan los recursos del servicio mediante `letGo()` para optimizar memoria.

Este diseño garantiza una gestión eficiente de recursos durante el proceso de agregado de canciones a playlists.

Listing 15: removeSongFromPlaylist()

```
async removeSongFromPlaylist() {
    const data_dir = await appDataDir();
    invoke('remove_song_from_playlist', {playlist_id: this.
        playlistId, song_id: this.id, db_path: data_dir});
    this.mainScreenStatus.refresh();
}
```

Cuando se activa la eliminación de una canción mediante el botón correspondiente, se ejecuta una petición al backend implementado en Rust que procesa la eliminación del registro en la base de datos SQLite. El método obtiene primero la ruta del directorio de datos mediante `appDataDir()`, luego envía los parámetros necesarios (identificador de playlist, identificador de canción y ruta de la base de datos) a través de la función `invoke()`. Finalmente, se actualiza el estado de la interfaz principal llamando al método `refresh()` del servicio `mainScreenStatus`, lo que garantiza que los cambios se reflejen inmediatamente en la vista del usuario.

Listing 16: toggleStarred()

```
async toggleStarred() {
    const data_dir = await appDataDir();

    if (this.isStarred) {
      invoke('remove_is_starred', {song_id: this.id, db_path:
          data_dir});
      console.log("Canción !Starred: " + this.id);
    } else {
      invoke('add_is_starred', {song_id: this.id, db_path:
          data_dir});
      console.log("Canción Starred: " + this.id);
    }

    this.mainScreenStatus.refresh();
}
```

El método `toggleStarred()` gestiona el estado de favorito de las canciones mediante un proceso que alterna entre dos posibles acciones. Cuando una canción está marcada como favorita (`isStarred = true`), se ejecuta la función `remove_is_starred` en el backend de Rust para eliminarla de la lista especial. En caso contrario, se invoca `add_is_starred` para añadirla. En ambos casos, se utiliza `appDataDir()` para obtener la ruta de la base de datos y se actualiza la interfaz mediante `mainScreenStatus.refresh()`. Las canciones marcadas como favoritas se agrupan automáticamente en una playlist especial accesible desde un botón específico en la barra lateral de la aplicación.

**Playlist Button**

Este componente de interfaz se activa exclusivamente cuando se inicia el menú modal para añadir canciones a playlists.

Estos son sus atributos:

Listing 17: Atributos Playlist Button

```
coverPath = 'assets/black.jpg';

constructor (public mainScreenStatus:MainScreenStatusService) {}

@Input() playlistId!: number;

@Input() playlistName!: string;

@Input() playlistCoverPath!: string;

@Input() songId!: string;
```

Su funcionamiento principal consiste en presentar al usuario la lista completa de playlists disponibles en el sistema, permitiendo la selección del destino donde se agregará la canción actual.

Sus métodos más importantes son:

Listing 18: getCoverPath()

```
async ngOnInit() {
    this.coverPath = await this.getCoverPath();
}

async getCoverPath(): Promise<string> {
    if (!this.coverPath) return 'assets/black.jpg';

    const fileData = await readFile(this.playlistCoverPath);

    const blob = new Blob([fileData], { type: 'image/jpeg' });

    if (this.coverPath) {
        URL.revokeObjectURL(this.coverPath);
    }

    return URL.createObjectURL(blob);
}
```

Este método, que sigue la misma implementación ya descrita en el componente anterior, se encarga de gestionar la visualización de las portadas de las playlists.

Listing 19: addSongToPlaylist()

```
async addSongToPlaylist() {
    const data_dir = await appDataDir();
    invoke('add_song_to_playlist', {playlist_id: this.playlistId,
        song_id: this.songId, db_path: data_dir})
    this.mainScreenStatus.refresh();
}
```

El método `addSongToPlaylist()` ejecuta el proceso de añadir una canción a una playlist específica mediante una llamada al backend implementado en Rust. Primero obtiene el directorio de datos de la aplicación usando `appDataDir()`, luego envía los parámetros necesarios (ID de la playlist, ID de la canción y ruta de la base de datos) a través de la función `invoke()`. Finalmente, actualiza la página principal llamando a `mainScreenStatus.refresh()` para reflejar los cambios en la interfaz de usuario. Este flujo garantiza que la canción quede registrada en la base de datos y que la vista se actualice consistentemente.

#### Homeitem

El componente `Homeitem` representa una implementación alternativa al componente estándar `Song`, desarrollado específicamente para cumplir con requerimientos particulares de diseño y funcionalidad en la playlist *Starred Songs*.

### 9.1.4. Servicios

Se han implementado servicios especializados para cada funcionalidad clave, tales como el manejo de listas de reproducción, la gestión de metadatos musicales y la información de la `mainscreen`.

#### Main Screen Status

Listing 20: Atributos Main Screen Status

```
onHome = signal(true);
pId: number = 0;
```

```
pName: string = "";
pDate: string = "";
pCoverPath: string = "";
pStarred: boolean = false;

songs:Song[] = []

constructor(public songManagement:SongManagementService) { }
```

El servicio `MainScreenStatus` se encarga de gestionar y mantener el estado de la información mostrada en la interfaz principal de la aplicación (`mainscreen`). Para ello, se han definido una serie de atributos que almacenan tanto los metadatos de la playlist actualmente visualizada como las canciones asociadas a la misma.

Entre los atributos principales se incluyen señales y variables que controlan el estado de la interfaz (`onHome`), así como información específica de la playlist, como su identificador (`pId`), nombre (`pName`), fecha de creación (`pDate`), ruta de la portada (`pCoverPath`) y estado de favorito (`pStarred`). Adicionalmente, se almacena un arreglo de canciones (`songs`) que pertenecen a la playlist actual. Por supuesto este servicio contiene la interfaz `Song` vista anteriormente.

El servicio se inicializa con una dependencia inyectada del `SongManagementService`, lo que permite una coordinación fluida entre la gestión del estado de la interfaz y las operaciones relacionadas con la reproducción de audio. Esta arquitectura modular facilita el mantenimiento y la escalabilidad del código.

Listing 21: setHome() y setPlaylist()

```
setHome() {
    this.onHome.set(true);
    this.pId = 0;
    this.pName = "";
    this.pDate = "";
    this.pCoverPath = "";
    this.getAllStarred();
}

```

```
10 setPlaylist(id:number, name:string, date:string, coverPath:string,
       isStarred:boolean) {
11     this.onHome.set(false);
12     this.pId = id;
13     this.pName = name;
14     this.pDate = date;
15     this.pCoverPath = coverPath;
16     this.pStarred = isStarred;
17 }
```

Los métodos `setHome()` y `setPlaylist()` gestionan la transición entre las diferentes vistas de la interfaz principal (`mainscreen`). En el primer caso, `setHome()` configura el estado para mostrar la lista de canciones marcadas como favoritas, estableciendo el valor `onHome` a `true` y reiniciando los metadatos de la playlist. Simultáneamente, se invoca el método `getAllStarred()` para cargar las canciones destacadas.

Por otro lado, `setPlaylist()` se utiliza para visualizar una playlist específica, actualizando los metadatos correspondientes (`id`, `name`, `date`, `coverPath` y estado `isStarred`) y desactivando la vista de favoritos mediante `onHome.set(false)`.

Cabe destacar que la nomenclatura original (`Home`) fue heredada de las primeras etapas del desarrollo, cuando la estructura conceptual del proyecto aún estaba en definición. Posteriormente, esta terminología se actualizó a `Starred Songs` para reflejar con mayor precisión su funcionalidad. A pesar de este cambio, se mantuvieron las referencias originales en el código para preservar la compatibilidad con los componentes existentes.

Se ha considerado una refactorización futura para unificar la nomenclatura y mejorar la coherencia del código. Actualmente, aunque los términos `Home` y `Starred Songs` coexisten, esta dualidad no afecta la funcionalidad del sistema.

Listing 22: Obtención de playlists

```
1 async getAllSongs() {
2     const data_dir = await appDataDir();
```

```
    try {
      this.songs = await invoke<Song[]>('get_all_songs', {db_path:
          data_dir});
    } catch (error) {
      console.error('Error fetching songs:', error);
      this.songs = [];
    }
}

async getAllStarred() {
    const data_dir = await appDataDir();
    try {
      this.songs = await invoke<Song[]>('get_all_starred', {
          db_path: data_dir});
    } catch (error) {
      console.error('Error fetching songs:', error);
      this.songs = [];
    }
}

async getPlaylistSongs() {
    const data_dir = await appDataDir();
    try {
      this.songs = await invoke<Song[]>('get_playlist_songs', {
          playlist_id: this.pId, db_path: data_dir});
    } catch (error) {
      console.error('Error fetching songs:', error);
      this.songs = [];
    }
}
```

Los métodos `getAllSongs()`, `getAllStarred()` y `getPlaylistSongs()` gestionan la obtención de canciones desde diferentes fuentes, asignándolas posteriormente a la variable local `songs`. En cada caso, se sigue un patrón similar: primero se obtiene el directorio de datos de la aplicación mediante `appDataDir()`, luego se realiza una llamada al backend mediante `invoke()`, y finalmente se manejan tanto el éxito como los posibles errores de la operación.

Listing 23: refresh()

```
async refresh() {
    if (this.pId != 0) {
```

```
      await this.getPlaylistSongs();
    } else if (this.pId == 0 && this.onHome()) {
      await this.getAllStarred();
    } else {
      await this.getAllSongs();
    }
}
```

El método `refresh()` se encarga de actualizar dinámicamente el contenido mostrado en la `mainscreen` en función del contexto actual de reproducción. Para ello, se implementa un flujo condicional que evalúa el identificador de playlist almacenado en `this.pId` y el estado de la señal `onHome`.

Este enfoque garantiza que la interfaz siempre muestre el contenido más relevante según el estado actual de la aplicación, manteniendo una experiencia de usuario coherente. La implementación mediante llamadas asíncronas (`await`) asegura que las operaciones de actualización se completen antes de proceder con cualquier otra acción, evitando así condiciones de carrera o estados inconsistentes en la interfaz.

Listing 24: playQueue() y addQueue()

```
playQueue() {
    this.songManagement.setQueue(this.songs);
  }

addQueue() {
    this.songManagement.addQueue(this.songs);
}
```

Estos métodos gestionan la interacción con la cola de reproducción a través del servicio `SongManagementService`. Al pulsar sobre el nombre de la playlist, se invoca `playQueue()`, que establece la lista actual de canciones (`this.songs`) como nueva cola de reproducción mediante el método `setQueue()` del servicio. De forma similar, al seleccionar la opción de añadir a cola, `addQueue()` incorpora las canciones a la cola existente utilizando `addQueue()` del mismo servicio.

Esta implementación refleja una arquitectura limpia donde la lógica de gestión de reproducción se delega completamente al servicio,

permitiendo que los componentes se centren únicamente en la presentación de datos y captura de eventos de usuario. El servicio `SongManagementService` será analizado en profundidad en la siguiente sección de esta documentación.

### Song Management

Listing 25: Atributos Song Management

```
songCover:string = ""
songTitle:string = ""
songArtist:string = ""

constructor() { this.setupSongListeners() }

song: HTMLAudioElement = new Audio();

volume:number = 0;

isPlaying:boolean = false;

loopMode:string = "none" //* none - playlist - single
shuffle:boolean = false

queue:Song[] = []
queueSave:Song[] = []
currentIndex:number = 0
currentIndexSave:number = 0
currentISDelay:number = 0

progress:number = 0;
```

El servicio `SongManagementService` centraliza toda la lógica relacionada con la reproducción musical y gestión de colas en la aplicación. Sus atributos principales pueden categorizarse en tres grupos:

1. **Metadatos visuales**: Almacena información de la canción actual (`songCover`, `songTitle`, `songArtist`) para mostrarse en la interfaz.
2. **Control de reproducción**: Incluye el elemento `HTMLAudioElement` para manejo nativo del audio, junto con estados como `isPlaying`, `volume` y `progress` que reflejan la reproducción en tiempo real.

3. **Gestión de cola**: Mantiene dos versiones de la cola (`queue` y `queueSave`) para implementar funcionalidades como shuffle, junto con índices de posición (`currentIndex`, `currentIndexSave`) y modos especiales (`loopMode`, `shuffle`).

El constructor inicializa automáticamente los listeners de eventos mediante `setupSongListeners()`, preparando el servicio para responder a cambios de estado.

Listing 26: setUpSongListeners()

```
private async setupSongListeners() {
    this.song?.addEventListener('ended', () => this.playNext());
    this.song?.addEventListener('timeupdate', () => {
      if (this.song) {
        this.progress = (this.song.currentTime / this.song.
            duration) * 100;
      }
    });
}
```

El método `setupSongListeners()` configura los listeners esenciales para el control de reproducción, implementando dos funcionalidades clave mediante eventos sobre el elemento `HTMLAudioElement`.

1. El evento `'ended'` activa automáticamente `playNext()` al terminar la canción actual, permitiendo una reproducción continua.

2. El evento `'timeupdate'` actualiza constantemente la propiedad `progress` con el porcentaje de avance (calculado como la relación entre `currentTime` y `duration`), lo que se refleja en la barra de progreso de la interfaz.

Listing 27: getCoverPath()

```
async getCoverPath(coverPath:string): Promise<string> {
    let coverUrl: string = 'assets/black.jpg';
    if (!coverPath) return coverUrl;

    const fileData = await readFile(coverPath);

    const blob = new Blob([fileData], { type: 'image/jpeg' });

    if (coverUrl) {
```

```
      URL.revokeObjectURL(coverUrl);
    }

    return URL.createObjectURL(blob);

}
```

El método `getCoverPath()` representa una solución centralizada para la gestión de portadas de las canciones, cuya implementación fue migrada desde el componente `playbar` al servicio actual como parte de una refactorización para simplificar la arquitectura. Esta implementación unificada elimina la duplicación de código que existía cuando la funcionalidad estaba dispersa en varios componentes (a excepción de las playlists), mejorando el mantenimiento y permitiendo una gestión más eficiente de los recursos multimedia.

Listing 28: loadAndPlay()

```
async loadAndPlay(_path:string, _index:number) {
    try {
      this.songTitle = this.queue[_index].title;
      this.songArtist = this.queue[_index].artist;

      this.songCover = await this.getCoverPath(this.queue[_index].
          coverPath);

      const path = _path;

      const fileData = await readFile(path);
      const blob = new Blob([fileData], { type: 'audio/mp3' });
      const audioUrl = URL.createObjectURL(blob);

      this.song.src = audioUrl;
      await this.song.play();
      this.isPlaying = true;

      console.log('Canción cargada correctamente');
    } catch (error) {
      console.error('Error al cargar la canción:', error);
    }
}
```

El método `loadAndPlay()` gestiona todo el proceso de preparación y reproducción de canciones mediante un flujo secuencial. Primero ac-

tualiza los metadatos visuales (`songTitle`, `songArtist` y `songCover`) utilizando los datos de la canción en la posición `_index` de la cola. Para la portada, se emplea el método `getCoverPath()` previamente descrito. Luego carga el archivo de audio desde la ruta especificada (`_path`), lo convierte a un blob y genera una URL temporal para su reproducción. Finalmente asigna esta URL al elemento de audio (`this.song.src`) e inicia la reproducción, actualizando el estado `isPlaying`. Todo el proceso está encapsulado en un bloque try-catch que maneja posibles errores durante la carga.

Listing 29: set y add

```
setQueue(songs:Song[]) {
    this.queue = [...songs];
    this.currentIndex = 0;
    this.loadAndPlay(this.queue[0].path, 0);
}

addQueue(songs: Song[]) {
    this.queue.push(...songs)
}

setOneSong(song:Song) {
    this.queue = [];
    this.queue.push(song);
    this.currentIndex = 0;
    this.loadAndPlay(this.queue[0].path, 0);
}

addOneSong(song:Song) {
    this.queue.push(song);
}
```

Estos métodos constituyen la API fundamental para la gestión de la cola de reproducción, mencionada recurrentemente en secciones anteriores de la documentación. La implementación sigue un patrón dual con variantes ”set”(establecer) y .ªdd”(añadir):

Los métodos `setQueue()` y `setOneSong()` reemplazan completamente la cola existente. En el primer caso, se recibe un array de canciones que se copia mediante el operador de propagación (`...songs`), mientras que el segundo acepta una única canción, reiniciando la cola antes

de añadirla. Ambos métodos comparten el mismo flujo posterior: reinician el índice actual a 0 e invocan `loadAndPlay()` para comenzar la reproducción inmediata.

Por otro lado, `addQueue()` y `addOneSong()` implementan la funcionalidad complementaria de añadir contenido a la cola existente sin modificar el estado de reproducción actual. La versión `addQueue()` utiliza el mismo operador de propagación para incorporar múltiples canciones, mientras que `addOneSong()` trabaja con elementos individuales.

Listing 30: togglePlayPause()

```
async togglePlayPause() {
    if(!this.isPlaying) {
      this.song.play()
    } else {
      this.song.pause()
    }
    this.isPlaying = !this.isPlaying;
}
```

El método `togglePlayPause()` implementa la funcionalidad básica de control de reproducción mediante un mecanismo de alternancia. Cuando se invoca, verifica el estado actual de reproducción a través de la variable `isPlaying`. Si la reproducción está pausada (`isPlaying` es falso), se ejecuta el método `play()` del elemento de audio HTML. En caso contrario, se activa el método `pause()`. Finalmente, se invierte el valor de `isPlaying` para reflejar el nuevo estado.

Listing 31: playNext() y playPrevious()

```
playNext() {
    if ((this.currentIndex < this.queue.length - 1) && (this.
        loopMode == "none" || this.loopMode == "playlist")) {
      this.currentIndex++;
      this.currentISDelay++;
      this.loadAndPlay(this.queue[this.currentIndex].path, this.
          currentIndex);
    } else if ((this.currentIndex == this.queue.length - 1) &&
        this.loopMode == "playlist") {
      this.currentIndex = 0;
```

```
8         this.loadAndPlay(this.queue[this.currentIndex].path, this.
              currentIndex);
9       } else if (this.loopMode == "single") {
10        this.song.currentTime = 0
11      } else {
12        this.stop();
13      }
14      console.log("CurrentIndex: " + this.currentIndex + "
            CurrentIndexSave: " + this.currentIndexSave)
15  }
16
17  playPrevious() {
18      if(this.song.currentTime < 3) {
19        if((this.currentIndex > 0) && this.loopMode != "single") {
20          this.currentIndex--;
21          this.currentISDelay--;
22          this.loadAndPlay(this.queue[this.currentIndex].path, this.
                currentIndex);
23        } else if (this.currentIndex == 0) {
24          if (this.loopMode == "playlist") {
25            this.currentIndex = this.queue.length - 1
26            this.loadAndPlay(this.queue[this.currentIndex].path,
                  this.currentIndex);
27          } else if(this.loopMode == "single") {
28            this.song.currentTime = 0
29          } else {
30            this.stop()
31          }
32        } else {
33          this.song.currentTime = 0
34        }
35      } else {
36        this.song.currentTime = 0
37      }
38
39  }
40
41  stop() {
42      this.song.pause();
43      this.song.currentTime = 0;
44      this.isPlaying = false;
45  }
```

Estos métodos implementan la lógica avanzada de navegación en

la cola de reproducción, considerando diferentes modos de funcionamiento (`none`, `playlist` y `single`). En `playNext()`, se evalúa primero si hay canciones siguientes disponibles y el modo de repetición lo permite, incrementando entonces el índice y cargando la siguiente canción. Cuando se alcanza el final de la lista en modo `playlist`, se reinicia al principio. Para el modo `single`, simplemente se reinicia la canción actual.

El método `playPrevious()` incorpora lógica adicional para diferenciar entre retroceder a la canción anterior (si la reproducción actual lleva menos de 3 segundos) o reiniciar la canción actual. Maneja correctamente los casos especiales como el inicio de la lista en modo `playlist` (saltando al final) o el comportamiento específico del modo `single`. Ambos métodos mantienen actualizados los índices de posición (`currentIndex`, `currentISDelay`) y registran el estado actual para depuración.

La implementación demuestra un manejo robusto de todos los casos posibles de navegación, proporcionando una experiencia de usuario consistente con los reproductores musicales profesionales. El diseño modular permite fácil extensión para añadir nuevos modos de reproducción o comportamientos especiales en futuras iteraciones.

Listing 32: Control de tiempo y volumen

```
onInput(event: Event) {
    const input = event.target as HTMLInputElement;
    const newTime = parseInt(input.value);
    if (this.song) {
      this.song.currentTime = newTime;
    }
}

onVolume(event: Event) {
    const inputVolume = event.target as HTMLInputElement;
    const newVolume = parseInt(inputVolume.value) / 100;
    if (this.song) {
      this.song.volume = newVolume;
    }
    this.volume = parseInt(inputVolume.value);
```

```
    return this.volume
}
```

Estos métodos gestionan la interacción del usuario con los controles de reproducción. El método `onInput()` se encarga de actualizar la posición de reproducción cuando el usuario manipula la barra de progreso. Primero obtiene el valor del elemento HTML que generó el evento, lo convierte a número entero y luego actualiza la propiedad `currentTime` del elemento de audio si este está disponible.

Por su parte, `onVolume()` controla los ajustes de volumen mediante un flujo similar: captura el valor del control deslizante de volumen, lo normaliza dividiendo entre 100 (para adaptarlo al rango 0-1 que espera el API de audio) y aplica este valor tanto al elemento de audio como a la variable local `volume`. La conversión de tipos y normalización de valores garantiza que los controles funcionen consistentemente aunque provengan de diferentes fuentes de entrada.

Ambos métodos implementan una verificación de existencia (`if (this.song)`) como medida de seguridad para evitar errores cuando no hay elemento de audio inicializado.

Listing 33: cycleLoop()

```
cycleLoop() {
    switch (this.loopMode) {
      case "none":
        this.loopMode = "playlist";
        break;
      case "playlist":
        this.loopMode = "single";
        break;
      case "single":
        this.loopMode = "none";
        break;
      default:
        this.loopMode = "none";
        break;
    }
    return this.loopMode;
}
```

Este método se encarga de alternar cíclicamente la variable `loopMode`.

Listing 34: Shuffle

```
toggleShuffle() {
    if(!this.shuffle) {
      this.queueSave = [...this.queue];
      this.currentIndexSave = this.currentIndex;
      this.doShuffle();
    } else if(this.shuffle) {
      this.currentIndex = this.queueSave.indexOf(this.queue[this.
          currentIndex]);
      this.queue = this.queueSave;
    }
    this.shuffle = !this.shuffle;
    return this.shuffle;
  }

doShuffle() {
    this.queue.unshift(this.queue[this.currentIndex])
    this.queue.splice(this.currentIndex+1,1)
    this.currentIndex = 0
    let firstElement = this.queue[0]
    let queueToShuffle = this.queue.splice(1);
    let currentIndex2 = queueToShuffle.length;

    while (currentIndex2 != 0) {

      let randomIndex = Math.floor(Math.random() * currentIndex2);
      currentIndex2--;

      [queueToShuffle[currentIndex2], queueToShuffle[randomIndex]]
           = [queueToShuffle[randomIndex], queueToShuffle[
          currentIndex2]];
    }
    queueToShuffle.unshift(firstElement)
    this.queue = queueToShuffle

}
```

La implementación del modo aleatorio (*shuffle*) sigue un enfoque de dos fases mediante los métodos `toggleShuffle()` y `doShuffle()`. Cuando se activa el modo aleatorio, `toggleShuffle()` preserva el estado original de la cola (`queueSave`) y la posición actual (`currentIndexSave`) antes de aplicar la aleatorización. Este diseño permite restaurar el orden

inicial cuando se desactiva la función, manteniendo la coherencia con el comportamiento de reproductores profesionales.

El método `doShuffle()` implementa el algoritmo Fisher-Yates (Wikipedia contributors (2025)) para la mezcla aleatoria, optimizado para garantizar que la canción actual permanezca en primera posición durante la transición. El proceso consta de tres etapas:

1. Extracción y protección de la canción en reproducción
2. Mezcla aleatoria del resto de la cola mediante intercambios indexados
3. Reconstrucción de la cola con la canción actual preservada

Esta implementación demuestra especial atención al detalle en la experiencia de usuario, evitando saltos bruscos en la reproducción durante la activación del modo aleatorio.

El sistema mantiene sincronizados los estados paralelos (colas original y aleatoria, índices de posición) mediante operaciones atómicas, garantizando consistencia incluso en listas extensas. La solución combina eficiencia computacional (complejidad O(n) para la mezcla) con un diseño intuitivo que refleja los patrones de uso establecidos en aplicaciones de referencia del sector.

Listing 35: formatTime()

```
formatTime(seconds: number | undefined): string {
    if (seconds === undefined || isNaN(seconds)) return '0:00';

    const minutes = Math.floor(seconds / 60);
    const remainingSeconds = Math.floor(seconds % 60);

    const paddedSeconds = remainingSeconds.toString().padStart(2,
        '0');

    return `${minutes}:${paddedSeconds}`;
}
```

El método `formatTime()` implementa la conversión de segundos a un formato de tiempo legible (MM:SS) para la interfaz de usuario. Cuando recibe un valor indefinido o no numérico, devuelve por defecto '0:00'

para mantener la consistencia visual. Para valores válidos, realiza tres operaciones matemáticas: primero calcula los minutos completos mediante división entera, luego obtiene los segundos restantes usando el operador módulo, y finalmente aplica un padding de dos dígitos a los segundos para mantener un formato uniforme.

La salida cumple con los estándares de reproductores musicales profesionales, mostrando siempre minutos y segundos con dos dígitos (ej: 3:05 en lugar de 3:5), lo que mejora la legibilidad y experiencia de usuario durante la reproducción.

**Song Adding**

El servicio `SongAddingService` se especializa en la gestión de playlists durante el proceso de añadir canciones. Su diseño minimalista cumple con el principio de responsabilidad única, enfocándose exclusivamente en dos operaciones clave: la obtención de playlists y la liberación de recursos.

Listing 36: song-adding.service.ts

```
export class SongAddingService {
  playlists: Playlist[] = []

  constructor() { }

  async getAllPlaylists() {
    const data_dir = await appDataDir();
    try {
      this.playlists = await invoke<Playlist[]>('get_all_playlists
          ', {db_path: data_dir});
    } catch (error) {
      console.error('Error fetching playlists:', error);
      this.playlists = [];
    }
  }


  letGo() {
    this.playlists = [];
  }
```

```
20
21
22 }
23
24 interface Playlist {
25   id: number;
26   name: string;
27   creation_date: string;
28   cover_path: string;
29   isStarred: boolean;
30 }
```

El método principal `getAllPlaylists()` establece comunicación con el backend mediante Tauri (`invoke`), recuperando el listado completo de playlists desde la base de datos SQLite. Este listado es mapeado en la parte de HTML del componente *Song*. Implementa manejo de errores robusto que garantiza estabilidad incluso ante fallos de conexión, inicializando un array vacío como fallback. La interfaz `Playlist` define la estructura de datos esperada, incluyendo metadatos esenciales como identificador único, nombre, fecha de creación, ruta de la portada y estado de favorito.

La función `letGo()` proporciona un mecanismo de limpieza de memoria, liberando el array de playlists cuando ya no es necesario. Este enfoque refleja buenas prácticas de gestión de recursos en aplicaciones con componentes modales, donde la memoria puede acumularse durante ciclos repetidos de apertura/cierre.

## 9.2. Backend

Rust y SQLite

### 9.2.1. Tauri API

El archivo `lib.rs` constituye el punto de entrada principal del backend desarrollado con Tauri y Rust, donde se configura la aplicación y se definen los manejadores de las operaciones entre el frontend y el sistema. Su estructura sigue un flujo bien definido que combina inicialización de plugins, gestión de permisos y registro de comandos invocables.

Listing 37: lib.rs

```
use std::path::PathBuf;
use logic::add_playlist;
use logic::add_song;
use logic::add_starred;
use logic::remove_song;
use logic::remove_a_playlist;
use logic::remove_starred;
use tauri_plugin_fs::FsExt;

mod logic;
use logic::Playlist;
use logic::Song;

#[tauri::command(rename_all = "snake_case")]
async fn sync_lib(music_dir:String, app_data_dir:String) {
    let _ = logic::sync(PathBuf::from(app_data_dir),PathBuf::from(
        music_dir));
}


#[tauri::command(rename_all = "snake_case")]
async fn get_all_playlists(db_path:String) -> Result<Vec<Playlist
    >, String> {
    let playlists = logic::get_all_playlists(db_path)?;

    Ok(playlists)
}

#[tauri::command(rename_all = "snake_case")]
async fn get_all_songs(db_path:String) -> Result<Vec<Song>, String
    > {
    let all_songs = logic::get_all_songs(db_path)?;

    Ok(all_songs)
}

#[tauri::command(rename_all = "snake_case")]
async fn get_playlist_songs(playlist_id: i64, db_path: String) ->
    Result<Vec<Song>, String> {
    let playlist_songs = logic::get_playlist_songs(playlist_id,
        db_path)?;
```

```
37
38      Ok(playlist_songs)
39  }
40
41  #[tauri::command(rename_all = "snake_case")]
42  async fn get_all_starred(db_path:String) -> Result<Vec<Song>,
        String> {
43      let all_songs = logic::get_all_songs_starred(db_path)?;
44
45      Ok(all_songs)
46  }
47
48  #[tauri::command(rename_all = "snake_case")]
49  async fn create_playlist(name:String, cover_path:String, db_path:
        String) {
50      let _ = add_playlist(name, cover_path, db_path);
51  }
52
53  #[tauri::command(rename_all = "snake_case")]
54  async fn remove_playlist(playlist_id:i64, db_path:String) {
55      let _ = remove_a_playlist(playlist_id, db_path);
56  }
57
58  #[tauri::command(rename_all = "snake_case")]
59  async fn add_song_to_playlist(playlist_id:i64, song_id:String,
        db_path:String) {
60      let _ = add_song(playlist_id, song_id, db_path);
61  }
62
63  #[tauri::command(rename_all = "snake_case")]
64  async fn remove_song_from_playlist(playlist_id:i64, song_id:String
        , db_path:String) {
65      let _ = remove_song(playlist_id, song_id, db_path);
66  }
67
68  #[tauri::command(rename_all = "snake_case")]
69  async fn add_is_starred(song_id:String, db_path:String) {
70      let _ = add_starred(song_id, db_path);
71  }
72
73  #[tauri::command(rename_all = "snake_case")]
74  async fn remove_is_starred(song_id:String, db_path:String) {
75      let _ = remove_starred(song_id, db_path);
76  }
```

```
77
78 #[cfg_attr(mobile, tauri::mobile_entry_point)]
79 pub fn run() {
80     tauri::Builder::default()
81         .plugin(tauri_plugin_opener::init())
82         .plugin(tauri_plugin_dialog::init())
83         .plugin(tauri_plugin_fs::init())
84         .setup(|app| {
85             let scope = app.fs_scope();
86             if let Err(e) = scope.allow_directory("$HOME/Music",
                    true) {
87                 eprintln!("Failed access: {}", e)
88             }
89             if let Err(e) = scope.allow_directory("$APPDATA", true
                    ) {
90                 eprintln!("Failed access: {}", e)
91             }
92             if let Err(e) = scope.allow_directory("$HOME/.config",
                     true) {
93                 eprintln!("Failed access: {}", e)
94             }
95             if let Err(e) = scope.allow_directory("$HOME/Library/
                    Application Support", true) {
96                 eprintln!("Failed access: {}", e)
97             }
98             if let Err(e) = scope.allow_directory("src-tauri",
                    true) {
99                 eprintln!("Failed access: {}", e)
100            }
101
102            Ok(())
103         })
104         .invoke_handler(tauri::generate_handler![sync_lib,
                get_all_playlists, get_all_songs, get_playlist_songs,
                get_all_starred,create_playlist, remove_playlist,
                add_song_to_playlist, remove_song_from_playlist,
                add_is_starred, remove_is_starred])
105         .run(tauri::generate_context!())
106         .expect("error while running tauri application");
107 }
```

En la fase de setup, se definen los directorios accesibles para la aplicación mediante `fs_scope()`, abarcando ubicaciones críticas multiplataforma:

- Directorios de música ($HOME/Music)
- Rutas de configuración según el SO (`$APPDATA` en Windows, `$HOME/.local/share/` en Linux, `Library/Application Support` en macOS)

El método `invoke_handler` asocia las funciones Rust con llamadas desde el frontend mediante `generate_handler!`.

Los handlers registrados incluyen:

- Gestión de playlists (`create_playlist`, `remove_playlist`)
- Consultas a la base de datos (`get_all_songs`)
- Control de canciones destacadas (`add_is_starred`)

Finalmente, el builder se ejecuta con `run()`, propagando cualquier error durante el arranque.

### 9.2.2. Diseño de la base de datos

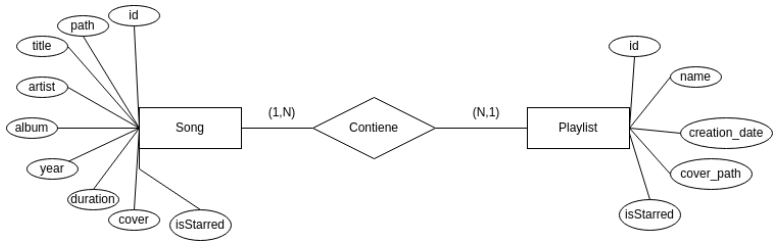


Figura 12: Diagrama Entidad - Relación

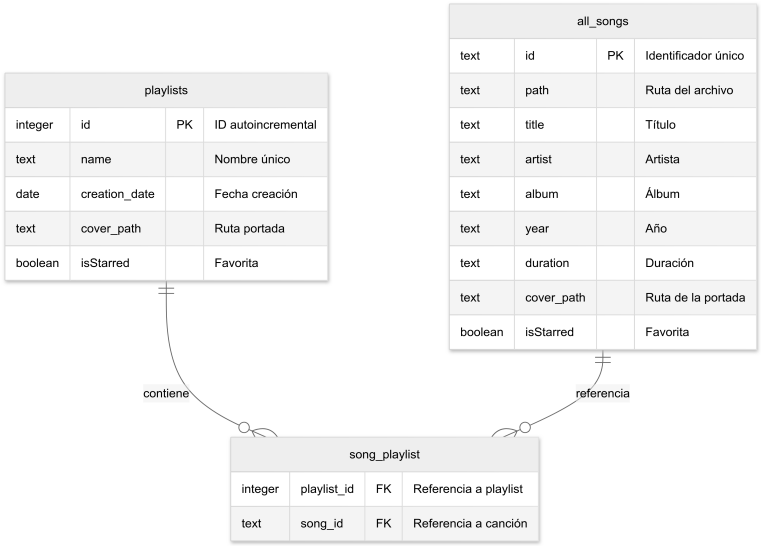


Figura 13: Diagrama Relacional

La base de datos se ha diseñado como un sistema relacional que gestiona dos entidades principales: las canciones y las listas de reproducción. El esquema sigue los principios de normalización para eliminar redundancias, utilizando una tabla intermedia para manejar la relación *muchos-a-muchos* entre estas entidades.

### All Songs

Funciona como repositorio central para todas las canciones disponibles en el sistema. Cada registro contiene:

- **id**: Hash único que actúa como clave primaria.
- **Metadatos**: Incluye título, artista, álbum, y año de lanzamiento.
- **Datos**: Almacena la ruta del archivo, duración en segundos y ubicación de la portada.
- **isStarred**: Indicador booleano que muestra si la canción ha sido marcada como favorita.

Esta tabla cumple una función dual como playlist especial (con ID 0 en la tabla de playlists) y presenta características únicas como la imposibilidad de eliminar canciones directamente desde ella.

#### Playlist

Almacena todas las listas de reproducción creadas por los usuarios con los siguientes campos:

- **id**: Identificador único autoincremental que actúa como clave primaria.
- **Metadatos**: Nombre de la playlist, fecha de creación (formato ISO 8601) y ruta a la imagen de portada (previamente procesada y optimizada).
- **isStarred**: Campo reservado para futuras implementaciones, actualmente no utilizado en la versión 1.0 debido a complejidades en la interacción con la interfaz gráfica.

#### Song Playlist

Establece las relaciones entre canciones y playlists mediante un diseño minimalista que contiene únicamente:

- Clave foránea a `All_Songs.id`.
- Clave foránea a `Playlist.id`.

Esta estructura permite consultas eficientes tanto para recuperar todas las canciones de una playlist como para identificar en qué playlists aparece una canción específica. La integridad referencial se mantiene mediante restricciones `FOREIGN KEY` que garantizan la consistencia de los datos.

### 9.2.3. Logica

El módulo `logic.rs` constituye el núcleo funcional del backend, donde se implementan las operaciones críticas de gestión de datos y sincronización musical. Su diseño sigue los principios de Rust: seguridad de tipos, manejo explícito de errores y eficiencia en el acceso a recursos del sistema.

Listing 38: fn sync()

```
pub fn sync(path:PathBuf, music_path:PathBuf) -> std::io::Result
    <()> {
    create_dir(path.clone())?;
    create_db(path.clone())?;
    clean(path.clone()).unwrap();
    walk_dir(music_path, path.clone())?;

    Ok(())
}
```

La función `sync()` coordina el proceso completo de inicialización de la biblioteca musical mediante cuatro etapas secuenciales:

1. **Preparación del entorno**.
2. **Inicialización de la base de datos**.
3. **Limpieza de registros obsoletos**.
4. **Exploración del directorio musical**: `walk_dir` recorre recursivamente la ruta especificada para indexar nuevos archivos.

Cada etapa maneja sus errores de forma independiente, siguiendo el patrón `Result` de Rust. La función principal propaga los errores mediante el operador `?`, mientras que `clean` utiliza `unwrap` al considerarse una operación no crítica.

Listing 39: fn create_dir()

```
fn create_dir(path:PathBuf) -> std::io::Result<()> {
    let new_path = path.join("covers");
    fs::create_dir_all(&new_path).expect("Error");
    Ok(())
}
```

Esta función se encarga de inicializar la estructura de directorios necesaria para el almacenamiento de archivos auxiliares.

El proceso sigue tres pasos fundamentales:

1. Construye la ruta completa concatenando el directorio base con la subcarpeta `covers`

2. Ejecuta la creación recursiva de directorios mediante `fs::create_dir_all()`

3. Maneja posibles errores con `expect` para fallos críticos

La implementación garantiza que exista siempre el directorio para almacenar las portadas de las canciones, requisito esencial para el funcionamiento de la interfaz gráfica. El uso de `PathBuf` en lugar de manipulaciones de strings crudas facilita su uso.

Listing 40: fn create_db()

```
fn create_db(path:PathBuf) -> std::io::Result<()> {
    let conn = Connection::open(path.join("playlists.db")).expect
        ("Error");
    conn.execute("create table if not exists all_songs(
        id text primary key not null,
        path text not null,
        title text not null,
        artist text,
        album text,
        year text,
        duration text,
        cover_path text,
        isStarred boolean default false
    );",()).expect("ERROR2");

    conn.execute("create table if not exists playlists (
        id integer primary key autoincrement,
        name text not null unique,
        creation_date DATE default CURRENT_DATE,
        cover_path text default '',
        isStarred boolean default false
    )", ()).expect("ERROR3");

    conn.execute("create table if not exists song_playlist(
        playlist_id integer not null,
        song_id text not null,
        foreign key (playlist_id) references playlists(id) ON
            DELETE CASCADE,
        foreign key (song_id) references all_songs(id) ON DELETE
            CASCADE,
        primary key (playlist_id, song_id)
    );",() ).expect("ERROR4");

```

```
    conn.execute("INSERT OR REPLACE INTO playlists (id, name)
        VALUES (?1,?2);", (0,"All Songs",)).expect("ERROR WHILE
        INSERTING");

    Ok(())
}
```

La función `create_db` genera la estructura SQLite con las tablas necesarias vistas en el punto anterior.

El diseño utiliza `if not exists` para evitar errores en ejecuciones repetidas y maneja errores mediante el operador `?` de Rust.

Listing 41: fn clean()

```
fn clean(database_path: PathBuf) -> Result<()> {
    let mut conn = Connection::open(database_path.join("playlists.
        db"))?;

    let rows: Vec<(String, String)> = {
        let mut stmt = conn.prepare("SELECT id, path FROM
            all_songs")?;
        let rows_iter = stmt.query_map([], |row| {
            Ok((row.get::<_, String>(0)?, row.get::<_, String>(1)
                ?))
        })?;

        let mut collected = Vec::new();
        for row in rows_iter {
            collected.push(row?);
        }
        collected
    };

    let tx = conn.transaction()?;

    for (id, path_str) in rows {
        println!("Checking path: {}", path_str);
        println!("Path exists? {}", Path::new(&path_str).exists())
            ;
        if !Path::new(&path_str).exists() {
            println!("Removing song: {} - File not found: {}", id,
                 path_str);
            tx.execute("DELETE FROM all_songs WHERE id = ?", [id])
                ?;
        }
```

```
26      }
27
28      tx.commit()?;
29      println!("Operation completed.");
30      Ok(())
31  }
```

Esta función implementa el mantenimiento de la base de datos eliminando registros de canciones cuyos archivos ya no existen en el sistema.

La función sigue un flujo de cuatro etapas:

1. **Consulta inicial**: Recupera todos los IDs y rutas de canciones almacenadas mediante una consulta SQL parametrizada. El uso de `query_map()` permite un mapeo seguro de los resultados a tuplas Rust.
2. **Transacción**: Inicia una transacción SQLite para garantizar atomicidad en las operaciones de limpieza. Esto previene estados inconsistentes si ocurren errores durante el proceso.
3. **Verificación**: Para cada canción registrada:
   - Comprueba la existencia del archivo físico usando `Path::exists()`.
   - Elimina el registro de la base de datos si el archivo no existe.
   - Registra la operación en la consola para diagnóstico.
4. **Confirmación**: Finaliza la transacción aplicando todos los cambios de forma atómica.

Características técnicas destacables:

- Manejo de errores unificado con el tipo `Result` de Rust.
- Uso de transacciones para garantizar integridad de datos.
- Consultas parametrizadas para prevenir inyección SQL.

Listing 42: fn walk_dir()

```
fn walk_dir(path:PathBuf, data_path:PathBuf) -> std::io::Result<()> {
    for entry in WalkDir::new(&path).into_iter().filter_entry(|e| is_music(e)) {
        let entry = entry?;
        if entry.file_name().to_str().unwrap_or("").ends_with(".mp3") || entry.file_name().to_str().unwrap_or("").ends_with(".ogg") {
            println!("{}", entry.path().display());
            let data = extract_metadata(entry.path().to_str().unwrap(),data_path.join("covers"))?;
            insert_song(data_path.clone(), entry.path().to_string_lossy().to_string(), data.0, data.1, data.2, data.3, data.4, data.5);
        }
    }
    Ok(())
}
```

Función principal que coordina el recorrido del directorio *music*.

Utiliza `WalkDir` para recorrido recursivo eficiente y filtra archivos mediante `is_music()` durante el recorrido. Cada archivo válido es procesado con `extract_metadata` y almacenado mediante `insert_song()`.

Listing 43: fn is_music()

```
fn is_music(entry: &DirEntry) -> bool {
    entry.file_name().to_str().map(|s| {s.ends_with(".mp3") || s.ends_with(".ogg") || s.ends_with(".wav") || s.ends_with(".flac") || s.ends_with(".aac")}).unwrap_or(false) || entry.file_type().is_dir()
}
```

Filtro para identificar archivos musicales.

Soporta múltiples formatos de audio incluyendo `.mp3`, `.ogg`, `.wav`, `.flac` y `.aac`. Permite recursividad al incluir directorios mediante `is_dir` con manejo seguro de conversión de nombres de archivo.

Listing 44: fn extract_metadata()

```
fn extract_metadata(path:&str, cover_path:PathBuf) -> std::io::Result<(String, String, String, String, u128, String)> {
    let tagged_file = read_from_path(path).expect("Error");
    let first_tag = tagged_file.first_tag();
```

```
    let title = first_tag.unwrap().title().unwrap().to_string();
    let artist = first_tag.unwrap().artist().unwrap_or(Cow::from("
        Unkown")).to_string();
    let album = first_tag.unwrap().album().unwrap_or(Cow::from("
        Unkown")).to_string();
    let year = first_tag.unwrap().year().unwrap_or(0000).to_string
        ();
    let duration = tagged_file.properties().duration().as_millis()
        ;
    let cover = first_tag.unwrap().pictures().first().map(|p| p.
        data().to_vec() );

    let cover_path = create_cover(cover_path, &title, cover).
        expect("Error creating cover");

    Ok((title,artist,album,year,duration,cover_path.
        to_string_lossy().to_string()))
}
```

Extracción de metadatos de archivos musicales.

Lee metadatos usando la crate `lofty` con valores por defecto para campos opcionales. Convierte la duración a milisegundos y gestiona la extracción de imágenes de portada mediante `create_cover()`.

Listing 45: fn create_cover()

```
fn create_cover(path:PathBuf, title:&str, image:Option<Vec<u8>>)
    -> std::io::Result<PathBuf> {

    let hash = blake3::hash(title.as_ref());
    let file_name = format!("{}.jpg",hash);
    let full_path = path.join(file_name);

    let image_data = image.unwrap_or_else(|| DEFAULT_BLACK_IMAGE.
        to_vec());

    let cover = File::create(&full_path);
    cover?.write_all(image_data.as_ref()).expect("ERROR writing");
    Ok(full_path)
}
```

Con el propósito de no estar constantemente extrayendo la portada de la canción, se crea un archivo dentro del directorio del programa

con un hash como nombre.

Listing 46: fn insert_song()

```
fn insert_song(path:PathBuf , song_path:String, title:String,
    artist:String, album:String, year:String, duration: u128,
    cover_path:String) {
    let dur_secs = (duration / 1000) as i64;
    let new_duration = format!("{}:{:02}",dur_secs / 60, dur_secs
        % 60);
    let uuid = blake3::hash(title.as_ref());

    let conn = Connection::open(path.join("playlists.db")).expect
        ("Error");
    conn.execute("INSERT OR IGNORE INTO all_songs (id, path, title
        , artist, album, year, duration, cover_path) VALUES
        (?1,?2,?3,?4,?5,?6,?7,?8);", (uuid.to_string(), song_path,
         title, artist, album, year, new_duration, cover_path)).
        expect("ERROR WHILE INSERTING");
}
```

Genera ID único mediante hash Blake3 del título usando `blake3::hash()`. Formatea la duración a `MM:SS` y evita duplicados con `INSERT OR IGNORE`. Almacena todos los metadatos extraídos en la tabla `all_songs` de la base de datos SQLite.

Listing 47: Operaciones CRUD playlist

```
pub fn add_playlist(name:String, cover_path:String, db_path:String
    ) -> Result<()> {
    let conn = Connection::open(PathBuf::from(db_path).join("
        playlists.db"))?;

    conn.execute("INSERT INTO playlists (name, cover_path) VALUES
        (?1, ?2);", (name, cover_path)).expect("ERROR WHILE
        INSERTING");

    Ok(())
}

pub fn remove_a_playlist(playlist_id:i64, db_path:String) ->
    Result<()> {
    let conn = Connection::open(PathBuf::from(db_path).join("
        playlists.db"))?;

```

```
12      conn.execute("DELETE FROM playlists WHERE id = ?1",(
            playlist_id,)).expect("ERROR WHILE DELETING SONG");
13
14      Ok(())
15  }
```

Métodos encargados de crear y eliminar *playlists* de la base de datos. Primero se crea una conexión y posteriormente se ejecuta la consulta con los parámetros de la playlist.

Listing 48: Operaciones CRUD canciones

```
1   pub fn add_song(playlist_id:i64, song_id:String, db_path:String)
        -> Result<()> {
2       let conn = Connection::open(PathBuf::from(db_path).join("
            playlists.db"))?;
3
4       conn.execute("INSERT OR REPLACE INTO song_playlist (
            playlist_id, song_id) VALUES (?1,?2);",(playlist_id,
            song_id)).expect("ERROR WHILE INSERTING SONG");
5
6       Ok(())
7   }
8
9   pub fn remove_song(playlist_id:i64, song_id:String, db_path:String
        ) -> Result<()> {
10      let conn = Connection::open(PathBuf::from(db_path).join("
            playlists.db"))?;
11
12      conn.execute("DELETE FROM song_playlist WHERE playlist_id = ?1
             AND song_id = ?2",(playlist_id, song_id)).expect("ERROR
            WHILE DELETING SONG");
13
14      Ok(())
15  }
16
17  pub fn add_starred(song_id:String, db_path:String) -> Result<()> {
18      let conn = Connection::open(PathBuf::from(db_path).join("
            playlists.db"))?;
19
20      conn.execute("UPDATE all_songs SET isStarred = 1 WHERE id =
            ?",(song_id,)).expect("ERROR WHILE MODIFIYING STARRED");
21
22      Ok(())
23  }
```

```
24
25 pub fn remove_starred(song_id:String, db_path:String) -> Result<()
       > {
26     let conn = Connection::open(PathBuf::from(db_path).join("
           playlists.db"))?;
27
28     conn.execute("UPDATE all_songs SET isStarred = 0 WHERE id =
           ?",(song_id,)).expect("ERROR WHILE MODIFIYING STARRED");
29
30     Ok(())
31 }
```

Métodos encargados de crear, eliminar y actualizar canciones de la base de datos. Primero se crea una conexión y posteriormente se ejecuta la consulta con los parámetros de la canción. Para las canciones favoritas se ejecuta es una consulta de `UPDATE`.

La implementación de los siguientes métodos utilizan estos `structs`:

Listing 49: structs Song y Playlist

```
1  pub struct Song {
2    id: String,
3    path: String,
4    title: String,
5    artist: String,
6    album: String,
7    year: String,
8    duration: String,
9    #[serde(rename = "coverPath")]
10   cover_path: String,
11   #[serde(rename = "isStarred")]
12   is_starred: bool,
13 }
14
15 pub struct Playlist {
16   id: i64,
17   name: String,
18   creation_date: String,
19   cover_path: String,
20   #[serde(rename = "isStarred")]
21   is_starred: bool,
22 }
```

El `struct Song` contiene todos los datos relevantes de cada canción ya vistos en la sección de la base de datos, incluyendo:

- Identificador único (`id`).
- Ruta del archivo (`path`).
- Metadatos (`title`, `artist`, `album`, `year`).
- Información de reproducción (`duration`).
- Ruta de la portada (`cover_path`).
- Estado (`is_starred`).

Por otra parte, `Playlist` incluye:

- Identificador único (`id`).
- Nombre de la *Playlist* (`name`).
- Fecha de creación (`creation_date`).
- Portada (`cover_path`).
- Estado (`is_starred`, no implementado por el momento).

Listing 50: fn get_all_playlists()

```
pub fn get_all_playlists(db_path:String) -> Result<Vec<Playlist>, String> {
    let conn = Connection::open(PathBuf::from(db_path).join("playlists.db")).map_err(|e| e.to_string())?;

    let mut stmt = conn.prepare("SELECT id, name, creation_date, cover_path, isStarred FROM playlists").map_err(|e| e.to_string())?;

    let playlists = stmt.query_map([], |row| {
        Ok(Playlist {
            id: row.get(0)?,
            name: row.get(1)?,
            creation_date: row.get(2)?,
            cover_path: row.get(3)?,
            is_starred: row.get(4)?,
```

```
        })
    }).map_err(|e| e.to_string())?
    .collect::<Result<Vec<_>, _>>()
    .map_err(|e| e.to_string())?;

    Ok(playlists)
}
```

Esta función implementa la consulta de todas las playlists almacenadas en la base de datos.

El proceso se ejecuta en tres etapas principales. Primero se establece la conexión con la base de datos ubicada en `playlists.db` dentro del directorio especificado.

Seguidamente se prepara y ejecuta la consulta SQL que selecciona todos los campos de la tabla `playlists`.

Finalmente, los resultados son mapeados a instancias del `struct Playlist` mediante un `query_map()` que procesa cada fila. El mapeo incluye todos los campos relevantes: `id`, `name`, `creation_date`, `cover_path` y `is_starred`.

El vector resultante con todas las playlists es devuelto al frontend a través de `lib.rs`, donde será utilizado para poblar la interfaz de usuario como se ha visto previamente en esta documentación.

Listing 51: fn get_all_songs() y fn get_all_songs_starred()

```
pub fn get_all_songs(db_path:String) -> Result<Vec<Song>, String>
    {
    let conn = Connection::open(PathBuf::from(db_path).join("
        playlists.db")).map_err(|e| e.to_string())?;

    let mut stmt = conn.prepare("SELECT id, path, title, artist,
        album, year, duration, cover_path, isStarred FROM
        all_songs").map_err(|e| e.to_string())?;

    let all_songs = stmt.query_map([], |row| {
        Ok(Song {
            id: row.get(0)?,
            path: row.get(1)?,
            title: row.get(2)?,
            artist: row.get(3)?,
            album: row.get(4)?,
```

```
            year: row.get(5)?,
            duration: row.get(6)?,
            cover_path: row.get(7)?,
            is_starred: row.get(8)?,
        })
    }).map_err(|e| e.to_string())?
    .collect::<Result<Vec<_>, _>>()
    .map_err(|e| e.to_string())?;

    Ok(all_songs)

}

pub fn get_all_songs_starred(db_path:String) -> Result<Vec<Song>, String> {
    let conn = Connection::open(PathBuf::from(db_path).join("playlists.db")).map_err(|e| e.to_string())?;

    let mut stmt = conn.prepare("SELECT id, path, title, artist, album, year, duration, cover_path, isStarred FROM all_songs WHERE isStarred = 1").map_err(|e| e.to_string())?;

    let all_songs = stmt.query_map([], |row| {
        Ok(Song {
            id: row.get(0)?,
            path: row.get(1)?,
            title: row.get(2)?,
            artist: row.get(3)?,
            album: row.get(4)?,
            year: row.get(5)?,
            duration: row.get(6)?,
            cover_path: row.get(7)?,
            is_starred: row.get(8)?,
        })
    }).map_err(|e| e.to_string())?
    .collect::<Result<Vec<_>, _>>()
    .map_err(|e| e.to_string())?;

    Ok(all_songs)
}
```

Estas funciones implementan el acceso a las canciones almacenadas en la base de datos, ambas comparten la misma estructura pero

tienen un propósito diferente.

`get_all_songs()` recupera todas las canciones almacenadas en la tabla `all_songs`, mientras que `get_all_songs_starred()` filtra solo aquellas marcadas como favoritas mediante la condición `WHERE isStarred = 1`.

El flujo de ejecución es idéntico en ambos casos:

1. Establece conexión con la base de datos.
2. Prepara y ejecuta la consulta SQL correspondiente.
3. Mapea los resultados a instancias del `struct Song`.
4. Devuelve el vector de canciones o propaga cualquier error.

Listing 52: fn get_playlist_songs()

```
pub fn get_playlist_songs(playlist_id: i64, db_path: String) ->
    Result<Vec<Song>, String> {
    let conn = Connection::open(PathBuf::from(db_path).join("
        playlists.db"))
        .map_err(|e| e.to_string())?;

    let mut stmt = conn.prepare(
        "SELECT s.id, s.path, s.title, s.artist, s.album, s.year,
            s.duration, s.cover_path, s.isStarred FROM all_songs s
             INNER JOIN song_playlist sp ON s.id = sp.song_id
            WHERE sp.playlist_id = ?1"
    ).map_err(|e| e.to_string())?;

    let playlist_songs = stmt
        .query_map([playlist_id], |row| {
            Ok(Song {
                id:          row.get(0)?,
                path:        row.get(1)?,
                title:       row.get(2)?,
                artist:      row.get(3)?,
                album:       row.get(4)?,
                year:        row.get(5)?,
                duration:    row.get(6)?,
                cover_path:  row.get(7)?,
                is_starred:  row.get(8)?,
            })
        })
```

```
23          .map_err(|e| e.to_string())?
24          .collect::<Result<Vec<_>, _>>()
25          .map_err(|e| e.to_string())?;
26
27      Ok(playlist_songs)
28  }
```

Esta función implementa la consulta de canciones pertenecientes a una playlist específica mediante una operación JOIN.

La función realiza una consulta SQL compleja que combina los datos de las tres tablas:

- `all_songs`: Contiene los metadatos completos de las canciones.
- `song_playlist`: Establece las relaciones entre canciones y playlists.

El JOIN se realiza mediante la condición `s.id = sp.song_id`, filtrando por el parámetro `playlist_id` proporcionado.

Los campos seleccionados incluyen todos los metadatos necesarios para la reproducción y visualización al igual que en los métodos anteriores:

- Identificación única (`id`).
- Ruta del archivo (`path`).
- Metadatos (`title`, `artist`, `album`, `year`).
- Duración formateada (`duration`).
- Ruta de la portada (`cover_path`).
- Estado (`isStarred`).

# 10. Trabajo Futuro

El desarrollo de Musy contempla varias líneas de mejora y expansión de funcionalidades que permitirán enriquecer la experiencia del usuario y ampliar las capacidades técnicas de la aplicación. A continuación se detallan los principales planes de desarrollo futuro:

## 10.1. Implementación de Playlist favoritas

La implementación completa del sistema `is_starred` para playlists incluirá la integración de las *playlists* favoritas en la actual ***Starred***, ya sea mediante un orden concreto o por último añadido.

## 10.2. Personalización de Temas Visuales

Se planea implementar un sistema de temas personalizables que permitirá a los usuarios:

- **Cambio de color de acento**: Los usuarios podrán seleccionar entre una paleta de colores predefinidos o definir valores RGB personalizados para los elementos interactivos de la interfaz.
- **Modos de contraste**: Implementación de temas claros y oscuros.
- **Almacenamiento de preferencias**: Las configuraciones de tema se guardarán en un archivo `config.json` dentro del directorio de datos de la aplicación, utilizando el formato:

Listing 53: config.json

```
{
  "theme": {
    "accent_color": "#ffccff",
    "background_color: "#cccccc",
    "dark_mode": true
  }
}
```

## 10.3. Gestión Flexible de Directorios Musicales

La expansión del sistema de gestión de bibliotecas musicales incluirá la capacidad de añadir y configurar mútiples directorios de música ($HOME/Descargas/Música, /mnt/DISCO/Música, etc...)

## 10.4. Creación automática de Playlist

Al igual que la app crea automáticamente una playlist con todas las canciones, se planea implementar un generador inteligente de play-

lists basado en la estructura de directorios del usuario. Esta funcionalidad está diseñada específicamente para usuarios con bibliotecas musicales offline ya organizadas en carpetas, ofreciendo una transición fluida a Musy.

# 11. Plan de Monetización y Modelo de Negocio

La estrategia de monetización de Musy se ha diseñado para equilibrar la sostenibilidad económica del proyecto con el compromiso de ofrecer una herramienta de calidad a los usuarios. El modelo actual y las proyecciones futuras se detallan a continuación:

## 11.1. Modelo Actual

- **Licencia de pago único**:
  - Precio base: 5€ en itch.io.
  - Incluye todas las funcionalidades actuales sin restricciones.
  - Actualizaciones gratuitas.
- **Ventajas competitivas**:
  - Coste significativamente inferior al de suscripciones anuales de alternativas (ej. 120€/año en otros reproductores premium).
  - Sin publicidad ni telemetría invasiva.
  - Código abierto con licencia comercial.

## 11.2. Métricas de Ventas

Cuadro 2: Proyección de ingresos (primer año)

| Canal | Precio Unitario | Unidades Estimadas | Ingreso Total |
|---|---|---|---|
| itch.io | 5€ | 1,200 | 6,000€ |
| Sponsors GitHub | - | - | 1,500€ |
| Donaciones | - | - | 800€ |
| Total | | | 8,300€ |

### 11.3. Consideraciones Éticas

- **Compromisos garantizados**:
  - Nunca implementar publicidad invasiva.
  - Mantener versión funcional completa offline.
  - No vender datos de usuarios.
- **Transparencia**:
  - Informe público anual de ingresos/gastos.
  - Roadmap público de desarrollo.

Este modelo busca demostrar que es posible monetizar software de calidad manteniendo principios éticos, ofreciendo valor real a cambio de un precio justo. Las estrategias futuras se activarán sólo cuando aporten beneficios tangibles a los usuarios, nunca como meros mecanismos de extracción de valor.

## 12. Conclusiones

### 12.1. Logros principales

Se ha desarrollado con éxito un reproductor de música multiplataforma (Linux, MacOs y Windows) utilizando Tauri, que cumple con los objetivos principales de:

- Optimización de recursos (Mucho menor consumo de ram respecto a otras webapps basadas en Electro).
- Compatibilidad de formatos de audio (FLAC, MP3, etc...).
- Arquitectura modular para el desarrollo futuro.

### 12.2. Dificultades clave

El principal desafío técnico fue la curva de aprendizaje asociada a Rust, lenguaje de programación de sistemas con el que no se contaba experiencia previa. Esta dificultad se resolvió mediante el estudio de

documentación oficial, tutoriales especializados y la implementación de pruebas piloto. Adicionalmente, fue necesario adaptar los conocimientos existentes en desarrollo web al framework Angular, cuyo paradigma de componentes requirió un período de adaptación.

## 12.3. Valoración global

A pesar de las dificultades, el proyecto valida el potencial de Tauri para aplicaciones de audio eficientes, ofreciendo un rendimiento superior al de frameworks tradicionales. La escalabilidad de la arquitectura permite añadir funcionalidades como cambios en la apariencia en futuras iteraciones.

### 12.3.1. Paginas webs visitadas durante el desarrollo

- Angular Team (2025) Documentación de Angular.
- Herrera (2025) Curso web de Angular.
- Font Awesome Team (s.f.) Iconos.
- “Lofty - Rust” (s.f.) Audio en Rust.
- MDN Web Docs (s.f.-a)
- MDN Web Docs (s.f.-b)
- MDN Web Docs (s.f.-c)
- MDN Web Docs (s.f.-d)
- Rusqlite contributors (s.f.) Bases de datos en Rust.
- Stack Overflow contributors (s.f.)
- Tailwind CSS Team (2025) Estilo de HTML.
- Tauri Team (2024) Base del programa.
- “Walkdir - Rust” (s.f.) Recorrer directorios en Rust.
- Programming Channel (2024) Aprendizaje de Rust
- Mermaid Team (s.f.) Para la creación de Diagramas.

## Referencias


Angular Team. (2025). *Home • Angular*. Angular. https://v17.angular.io/docs

Font Awesome Team. (s.f.). *SVGs Used*. Font Awesome. https://fontawesome.com/

Herrera, F. (2025). *Curso Angular*. https://www.udemy.com/course/angular-2-fernando-herrera/

*IEEE Recommended Practice for Software Requirements Specifications*. (1998). IEEE Computer Society.

*Lofty - Rust*. (s.f.). Docs.rs. https://docs.rs/lofty/latest/lofty/

MDN Web Docs. (s.f.-a). *ArrayBuffer*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer

MDN Web Docs. (s.f.-b). *Blob*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/API/Blob

MDN Web Docs. (s.f.-c). *Canvas API*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

MDN Web Docs. (s.f.-d). *String.prototype.padStart()*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/padStart

Mermaid Team. (s.f.). *Mermaid*. https://mermaid.js.org/

Programming Channel. (2024, 21 de mayo). *Rust programming full course /Learn in 2024*. https://youtu.be/rQ_J9WH6CGk?si=GBTKkMXJuvvyq208

Rusqlite contributors. (s.f.). *rusqlite/rusqlite: Ergonomic bindings to SQLite for rust*. GitHub. https://github.com/rusqlite/rusqlite

Stack Overflow contributors. (s.f.). *How to randomize (shuffle) a JavaScript array?* Stack Overflow. https://stackoverflow.com/questions/2450954/how-to-randomize-shuffle-a-javascript-array

Tailwind CSS Team. (2025). *Rapidly build modern websites without ever leaving your HTML*. Tailwind CSS. https://tailwindcss.com

Tauri Team. (2024, 1 de octubre). *What is Tauri?* Tauri. https://v2.tauri.app/start/

*Walkdir - Rust*. (s.f.). Docs.rs. https://docs.rs/walkdir/2.2.9/walkdir/

Wikipedia contributors. (2025, 31 de mayo). *Fisher–Yates shuffle*. Wikipedia, The Free Encyclopedia. Consultado el 8 de junio de 2025,

desde https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle