



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

1. Introducción	1
2. Justificación del tema elegido	1
3. Objetivos	1
4. Metodología	1
5. Tecnologías y herramientas usadas en el proyecto	2
5.1. Razones	2
6. Especificación de Requisitos de Software	3
7. Despliegue y pruebas	3
7.1. Despliegue de la aplicación	3
7.2. Pruebas realizadas	3
7.2.1. Pruebas de integración	3
7.2.2. Pruebas de usabilidad	4
8. Estado del arte	4
8.1. Plataformas de streaming	4
8.2. Reproductores locales tradicionales	5
8.3. Mi propuesta	5
9. Profundización de conceptos	6
9.1. Frontend	6
9.1.1. Playbar	6
9.1.2. Sidebar	7
9.1.3. Mainscreen	15
9.1.4. Servicios	18
9.2. Backend	18
9.2.1. Tauri API	18
9.2.2. Sync method	18
9.2.3. SQL CRUD	18

10.Conclusiones	18
10.1.Logros principales	18
10.2.Dificultades clave	18
10.3.Valoración global	19

Índice de figuras

1. Rendimiento de la aplicación en MacOS	4
2. Playbar	6
3. Sidebar	7
4. Eliminar playlist	10
5. Menú de nueva playlist	10
6. Menú de buscar canción	14
7. Mainscreen	15
8. Song Button	17

Índice de cuadros

Listings

1. Atributos Y Constructora Sidebar	8
2. Atributos y Constructora Playlist	8
3. getCoverPath()	8
4. Dropdown	9
5. removePlaylist()	10
6. selectCover()	11
7. createThumbnail()	12
8. createPlaylist()	13
9. searchSong()	14
10. Atributos Playlist Button	15
11. getCoverPath()	16
12. addSongToPlaylist()	16
13. Atributos Song Button	17

1. Introducción

Con el aumento de precios y la inclusión de anuncios en las suscripciones premium de plataformas multimedia como Spotify o iTunes, se prevé un resurgimiento de la reproducción local de música. Este proyecto tiene como objetivo ofrecer una alternativa moderna, intuitiva y sencilla frente a las aplicaciones de reproducción local existentes.

2. Justificación del tema elegido

Aunque existen alternativas a las aplicaciones de reproducción en streaming, resulta difícil encontrar una que combine una interfaz moderna, atención al detalle y código abierto (open-source). Por ello, se ha desarrollado esta propuesta, centrada en una aplicación de reproducción local con una interfaz similar a Spotify o iTunes, lo que facilitará la transición de nuevos usuarios.

3. Objetivos

El objetivo principal es desarrollar una aplicación multiplataforma basada en tecnologías web, eficiente en la gestión de recursos del sistema y fácil de usar. Adicionalmente, se busca que el proyecto sirva como aprendizaje en el uso de Angular y Rust.

4. Metodología

Para la realización del proyecto se empleará la metodología ágil SCRUM. En este caso, el autor asumirá los roles de product owner, scrum master y development team. Las tareas serán asignadas diariamente, y se realizará un seguimiento continuo del progreso.

5. Tecnologías y herramientas usadas en el proyecto

Este proyecto está desarrollado mediante un conjunto de tecnologías modernas distribuidas en tres capas principales:

- **Frontend:** Angular 17, Tailwind CSS V4
- **Backend:** Rust, Tauri 2.0
- **Base de Datos:** SQLite

5.1. Razones

La selección de tecnologías para este proyecto se ha basado en los siguientes criterios técnicos y de eficiencia:

- **Angular 17:** Se ha elegido este framework por su arquitectura basada en componentes, que favorece la escalabilidad y mantenibilidad del código. Además, se ha considerado relevante la oportunidad de explorar alternativas a React, ampliando así el conocimiento en ecosistemas frontend modernos.
- **Tailwind CSS v4:** Se ha optado por esta herramienta debido a su eficiencia en el desarrollo de interfaces responsive, así como a la familiaridad previa con su paradigma utility-first, lo que permite agilizar el proceso de diseño.
- **Rust:** Se ha seleccionado este lenguaje por su rendimiento optimizado y seguridad. Además, representa una valiosa oportunidad de aprendizaje de un lenguaje de programación de bajo nivel.
- **Tauri:** Se ha preferido sobre alternativas como Electron debido a su menor consumo de recursos y mejor rendimiento en aplicaciones de escritorio multiplataforma.
- **SQLite:** Se ha implementado este sistema de gestión de bases de datos por su ligereza, y adecuación a los requisitos del proyecto.

El entorno de desarrollo está configurado en Visual Studio Code, utilizando extensiones específicas para cada tecnología mencionada, lo que garantiza un flujo de trabajo eficiente.

6. Especificación de Requisitos de Software

7. Despliegue y pruebas

En este apartado se detallan los pasos seguidos para el despliegue de la aplicación de escritorio, así como las pruebas realizadas para garantizar su correcto funcionamiento.

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

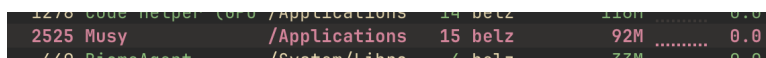
- Se comprobó la comunicación entre el frontend y el backend, asegurando que las llamadas desde Angular a Rust (a través de Tauri) funcionaban correctamente.
- Se testearon casos como la carga de archivos de audio, reproducción en diferentes formatos y manejo de errores.

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.



A screenshot of the macOS Activity Monitor application. The table lists running processes with columns for PID, Name, Location, CPU, Memory, and Energy. The process 'Musy' is highlighted in red, showing it is using 15% of the CPU and 92M of memory.

PID	Name	Location	CPU	Memory	Energy
1278	code helper	/Applications	14	belz	110M 0.0
2525	Musy	/Applications	15	belz	92M 0.0
449	BiomeAgent	/System/Library	4	belz	33M 0.0

Figura 1: Rendimiento de la aplicación en MacOS

8. Estado del arte

En la actualidad, el panorama de reproducción musical se divide principalmente en dos enfoques: plataformas de streaming y reproductores locales tradicionales. A continuación, se analizan sus características, ventajas y desventajas.

8.1. Plataformas de streaming

Las plataformas de streaming (Spotify, Apple Music, etc...) dominan el mercado actual debido a:

- Acceso instantáneo a catálogos musicales extensos (más de 100 millones de canciones).

- Interfaces modernas con recomendaciones basadas en algoritmos.
- Sincronización multiplataforma (dispositivos móviles, web y escritorio).

Sin embargo, presentan limitaciones significativas:

- Dependencia de conexión a internet permanente.
- Modelo de negocio basado en suscripciones o publicidad invasiva.
- Falta de propiedad real sobre la música (acceso condicionado al pago).

8.2. Reproductores locales tradicionales

Los reproductores de música offline (Winamp, Strawberry, etc...) ofrecen:

- Control completo sobre los archivos musicales (propiedad permanente).
- Uso sin restricciones de conectividad o cuentas de usuario.
- Menor consumo de recursos al evitar dependencias en la nube.

No obstante, adolecen de problemas críticos:

- Interfaces obsoletas y experiencias de usuario poco intuitivas.
- Dificultad para encontrar versiones actualizadas (muchos proyectos estan abandonados).
- Limitada compatibilidad con formatos modernos o sistemas operativos recientes.

8.3. Mi propuesta

Este análisis evidencia la necesidad de reproductores locales que combinen:

- Diseño contemporáneo (similar al streaming).
- Independencia de infraestructuras en la nube.
- Soporten estándares actuales (formatos lossless, metadatos avanzados).

La solución propuesta en este proyecto busca ocupar este espacio, aprovechando tecnologías modernas (Tauri, Rust, Angular) para superar las limitaciones de ambas aproximaciones.

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

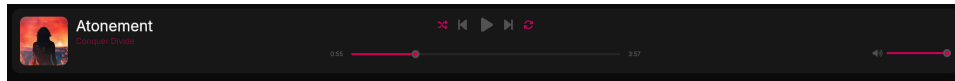


Figura 2: Playbar

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

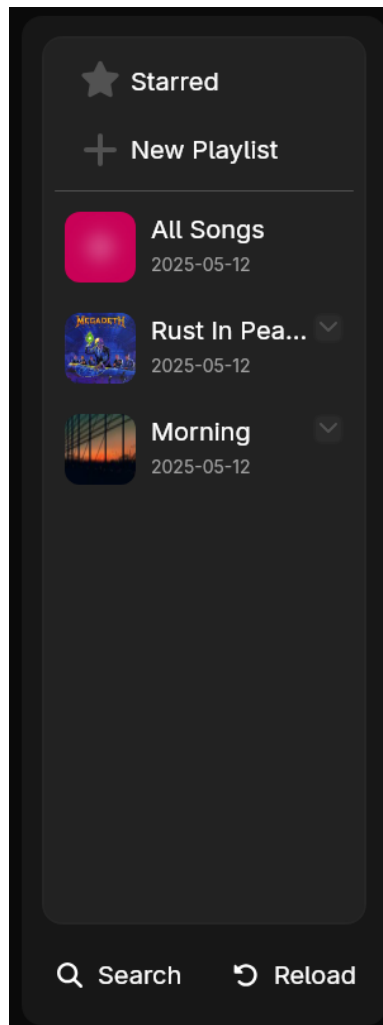


Figura 3: Sidebar

Componente donde se interactúa con las playlists, 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

```
1  playlists: Playlist[] = []
2  name:string = ""
3  isModalOpen:boolean = false;
4  isModalOpenSearch:boolean = false;
5  songs:Song[] = []
6  filteredSongs:Song[] = []
7  coverPath: string = "assets/black.jpg"
8
9  constructor (public mainScreenStatus:MainScreenStatusService,
               public songManagement:SongManagementService) {}
```

Serán explicados más adelante.

Playlist

Listing 2: Atributos y Constructora Playlist

```
1  coverPath = 'assets/black.jpg';
2
3  isDropDownOpen: boolean = false;
4
5  constructor (public mainScreenStatus:MainScreenStatusService) {}
6
7  @Input() playlistId!: number;
8
9  @Input() playlistName!: string;
10
11 @Input() playlistDate!: string;
12
13 @Input() playlistCoverPath!: string;
14
15 @Input() playlistIsStarred!: boolean;
16
17 @Input() refreshFn!: () => void;
```

Cada playlist funciona como un botón que actualiza el contenido de la mainscreen, que envía la lista de canciones correspondiente a ese componente mediante un servicio. 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 }

```

Debido a que la ruta de la portada no es válida para ser mostrada directamente, ha de convertirse en un blob. Este método se encarga de ello y, en el caso de que no exista una portada se carga una imagen pre-determinada.

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 temas de diseño, se ha decidido crear un dropdown menu (menú desplegable) con opciones como el boton de eliminación de playlist. A futuro es posible que se implementen más botones como por ejemplo para añadir una playlist a favoritos.

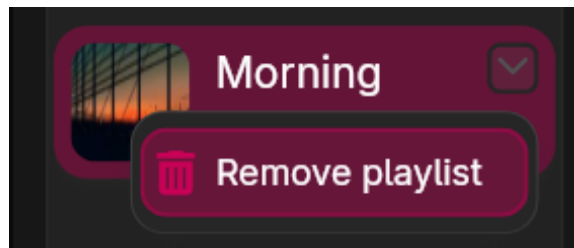


Figura 4: Eliminar playlist

Listing 5: removePlaylist()

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

Para eliminar una playlist de la sidebar se hace una petición al backend donde se realizará una consulta sql para eliminarla de la base de datos (Esto se verá mas adelante en el apartado de Rust y SQL).

Botón New Playlist

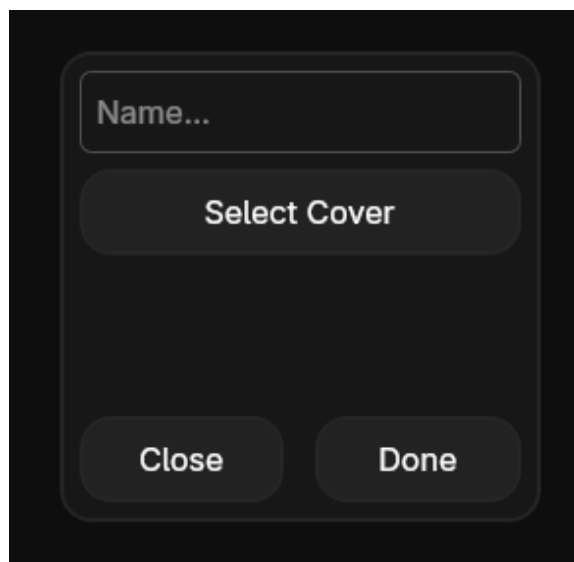


Figura 5: Menú de nueva playlist

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

Estos son los métodos más destacados:

Listing 6: `selectCover()`

```
1  async selectCover() {
2      const file = await open({
3          multiple: false,
4          directory: false,
5      });
6
7      if(file == null) {
8          return
9      }
10
11     const fileData = await readFile(file);
12     const ogBlob = new Blob([fileData], { type: 'image/*' });
13     const thumbnail = await this.createThumbnail(ogBlob, 200, 200)
14         ;
15
16     try {
17         await mkdir('pcovers', { baseDir: BaseDirectory.AppData });
18     } catch {
19         console.log("Already Created")
20     }
21
22     let randomName = (Math.floor(Math.random() * (Math.floor
23         (200000) - Math.ceil(1) + 1)) + Math.ceil(1)).toString();
24     await writeFile(`pcovers/${randomName}.jpg`, thumbnail, {
25         baseDir: BaseDirectory.AppData})
26
27     const data_dir = await appDataDir();
28     let newPath = data_dir + "/pcovers/" + randomName + ".jpg";
29     console.log(newPath)
30
31     this.coverPath = newPath;
32 }
```

Usando la APP de `tauri plugin-dialog` aparece una ventana del sistema que permite la selección de una imagen. Una vez seleccionada, un blob es creado y transformado en una imagen de baja resolución (con el fin de ahorrar espacio y mejorar el rendimiento). Esta nueva ima-

gen es guardad con un nombre aleatorio dentro de *pcovers*, carpeta la cual es creada si esta no existe previamente. A continuación el método `createThumbnail` encontrado en la línea 13.

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
19                         blob"));
20                     return;
21                 }
22                 try {
23                     const arrayBuffer = await thumbnailBlob.
24                         arrayBuffer();
25                     const uint8Array = new Uint8Array(arrayBuffer)
26                         ;
27                     resolve(uint8Array);
28                 } catch (error) {
29                     reject(error);
30                 }
31             }, 'image/jpeg', 0.7);
32         };
33
34         img.onerror = () => {
35             reject(new Error("Failed to load image"));
```



```

36         img.src = URL.createObjectURL(blob);
37     });
38 }

```

La MDN Canvas API es usada para la transformación de la imagen.

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()

```

1  async createPlaylist() {
2      const data_dir = await appDataDir();
3      invoke('create_playlist', {name: this.name, cover_path: this.
        coverPath, db_path: data_dir})
4      this.close()
5      this.refresh()
6      this.coverPath = "assets/black.jpg"
7  }

```

Boton Search

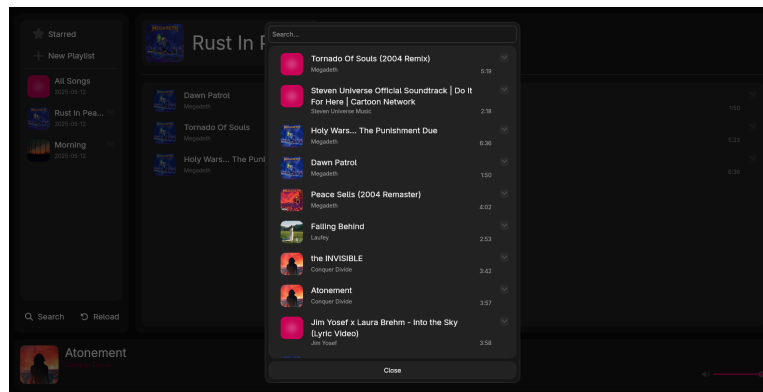


Figura 6: Menú de buscar canción

Listing 9: searchSong()

```

1  async searchSong() {
2      await this.getAllSongs();
3      this.filteredSongs = this.songs;
4      this.isModalOpenSearch = true;
5  }
6
7  async getAllSongs() {
8      const data_dir = await appDataDir();
9      try {
10         this.songs = await invoke<Song[]>('get_all_songs', {db_path:
            data_dir});
11     } catch (error) {
12         console.error('Error fetching songs:', error);
13         this.songs = [];
14     }
15 }

```

Cuando el botón de *search* es pulsado, la sidebar espera a recibir todas las canciones encontradas en la carpeta de música para posteriormente popular la lista de canciones filtradas y abrir el menú modal de búsqueda.

En este menú es posible encontrar y buscar todas las canciones e interactuar con ellas como si estuvieran dentro de una playlist (esto es explicado más adelante).

9.1.3. Mainscreen

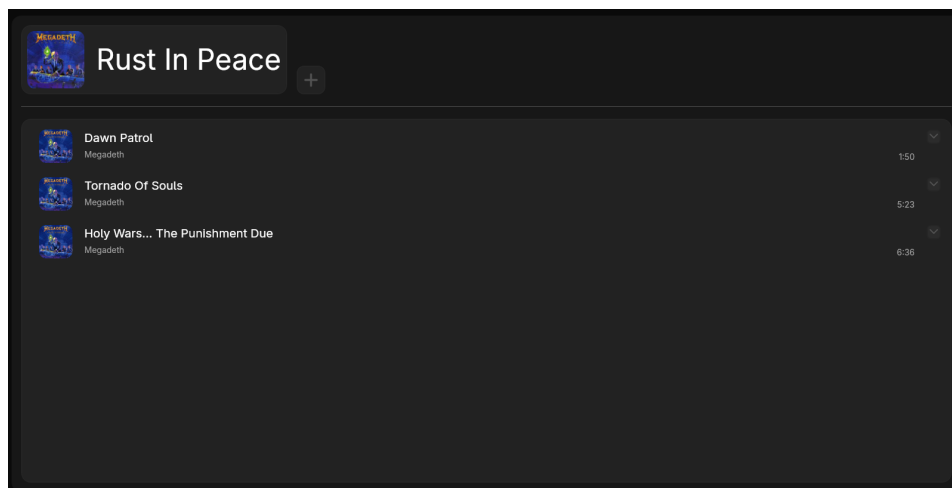


Figura 7: Mainscreen

Componente donde son mostradas las canciones de una playlist junto a su información y los botones para iniciar a escuchar o para añadir a la cola.

El archivo de typescript de este componente no posee nada sustancial, lo interesante es encontrado en sus subcomponentes.

Playlist Button

Este componente es mostrado cuando se inicia el menú modal para añadir una canción a una determinada playlist.

Estos son sus atributos:

Listing 10: Atributos Playlist Button

```
1 coverPath = 'assets/black.jpg';
2
3 constructor (public mainScreenStatus:MainScreenStatusService) {}
4
5 @Input() playlistId!: number;
6
7 @Input() playlistName!: string;
8
9 @Input() playlistCoverPath!: string;
10
11 @Input() songId!: string;
```

El componente recibe los metadatos de una playlist junto al id de la canción para su posterior adición a esta.

Sus métodos más importantes son:

Listing 11: `getCoverPath()`

```
1  async ngOnInit() {
2      this.coverPath = await this.getCoverPath();
3  }
4
5  async getCoverPath(): Promise<string> {
6      if (!this.coverPath) return 'assets/black.jpg';
7
8      const fileData = await readFile(this.playlistCoverPath);
9
10     const blob = new Blob([fileData], { type: 'image/jpeg' });
11
12     if (this.coverPath) {
13         URL.revokeObjectURL(this.coverPath);
14     }
15
16     return URL.createObjectURL(blob);
17 }
```

Como ha sido explicado antes, las rutas de imágenes no son aptas para ser mostradas directamente en Angular, por ello debe de crearse un blob y asignárselo a la variable `coverPath` cuando se inicializa el componente. (Por supuesto con una imagen predeterminada en caso de que no sea posible encontrar una imagen válida).

Listing 12: `addSongToPlaylist()`

```
1  async addSongToPlaylist() {
2      const data_dir = await appDataDir();
3      invoke('add_song_to_playlist', {playlist_id: this.playlistId,
4                                     song_id: this.songId, db_path: data_dir})
5      this.mainScreenStatus.refresh();
6  }
```

Se realiza una llamada a Rust para añadir la canción a la base de datos y se actualiza la `mainpage`.

Song Button

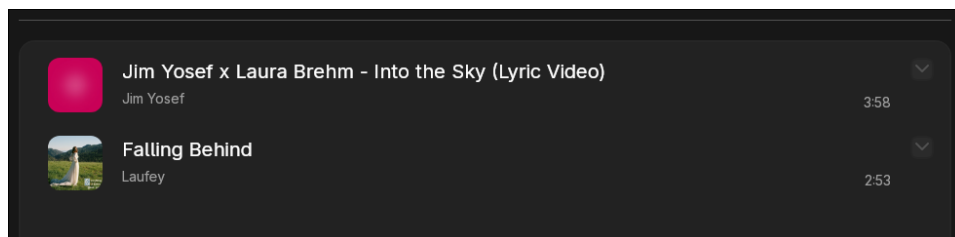


Figura 8: Song Button

Cada canción dentro de una playlist tiene un componente dedicado para su gestión y su información, pulsando sobre este se inicia la reproducción directa de una canción. A la derecha se encuentra un *dropdown menu* el cual permite añadir la canción a la cola, añadirla a otra playlist y, si no se encuentra sobre all songs eliminarla de la playlist (El concepto de All Songs se explicará más adelante).

Los atributos del componente son:

Listing 13: Atributos Song Button

```

1  isModalOpen: boolean = false;
2
3  isDropDownOpen: boolean = false;
4
5  constructor (public songManagement: SongManagementService, public
      songAdding: SongAddingService, public mainScreenStatus:
      MainScreenStatusService) {}
6
7  @Input() id!: string;
8  @Input() path!: string;
9  @Input() title!: string;
10 @Input() artist!: string;
11 @Input() album!: string;
12 @Input() year!: string;
13 @Input() duration!: string;
14 @Input() coverPath!: string;
15 @Input() isStarred!: boolean;
16
17 @Input() playlistId!: number;
18
19 coverUrl: string = 'assets/black.jpg';

```

Entre ellos se encuentran los metadatos de cada canción y la play-

list en la que se encuentran mostrados ahora mismo. Estos datos son recibidos desde el componente padre.

Este componente posee una cantidad interesante de métodos que serán explicados a continuación:

9.1.4. Servicios

9.2. Backend

Rust y SQLite

9.2.1. Tauri API

9.2.2. Sync method

9.2.3. SQL CRUD

10. Conclusiones

10.1. 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.

10.2. 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.

10.3. 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.