



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	16
9.1.4. Servicios	24
9.2. Backend	24
9.2.1. Tauri API	24
9.2.2. Sync method	24
9.2.3. SQL CRUD	24

10.Conclusiones	24
10.1.Logros principales	24
10.2.Dificultades clave	25
10.3.Valoración global	25

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	11
6. Menú de buscar canción	15
7. Mainscreen	16
8. Song Button	17

Índice de cuadros

Listings

1. Atributos Y Constructora Sidebar	8
2. Atributos y Constructora Playlist	8
3. getCoverPath()	9
4. Dropdown	9
5. removePlaylist()	10
6. selectCover()	11
7. createThumbnail()	12
8. createPlaylist()	14
9. searchSong()	15
10. Atributos Song Button	17
11. Song	18
12. getCoverPath()	18
13. Dropdown menu	19
14. playSong() y addSongToQueue()	20
15. addSongToPlaylist()	20

16.	removeSongFromPlaylist()	21
17.	toggleStarred()	21
18.	Atributos Playlist Button	22
19.	getCoverPath()	23
20.	addSongToPlaylist()	23

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

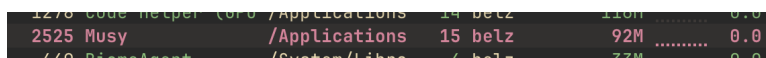
- Se comprobó la comunicación entre el frontend y el backend, asegurando que las llamadas desde Angular a Rust (a través de Tauri) funcionaban correctamente.
- Se testearon casos como la carga de archivos de audio, reproducción en diferentes formatos y manejo de errores.

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.



1278	code helper	(64)	/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

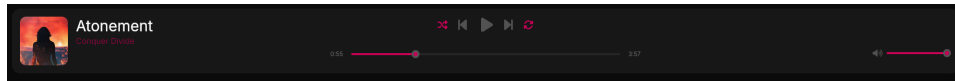


Figura 2: Playbar

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

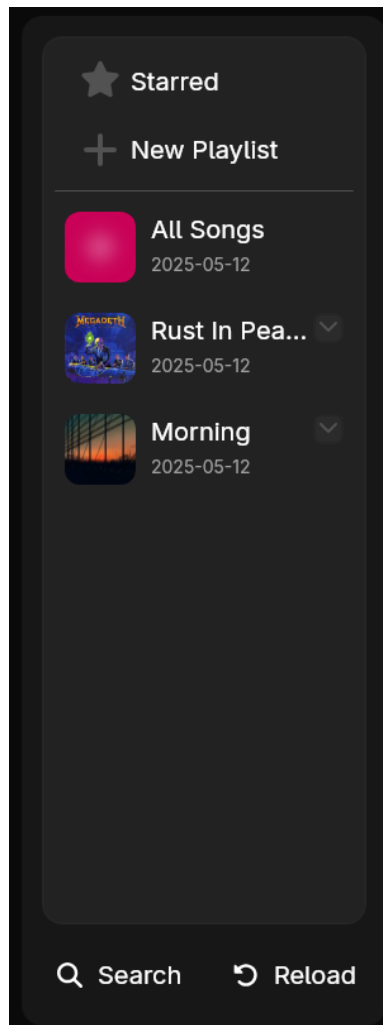


Figura 3: Sidebar

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

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

```
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 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.

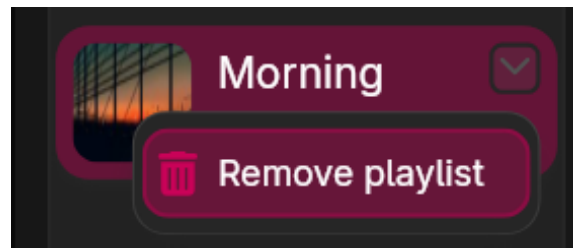


Figura 4: Eliminar playlist

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();
6  }

```

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

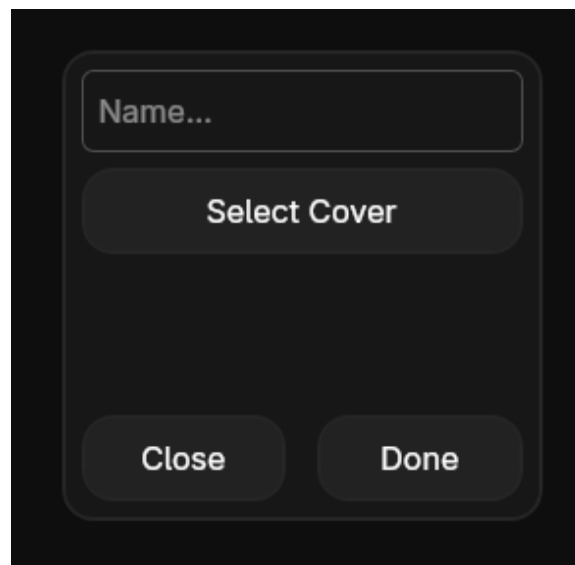


Figura 5: Menú de nueva playlist

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     try {
16         await mkdir('pcovers', { baseDir: BaseDirectory.AppData });
17     } catch {
18         console.log("Already Created")
19     }
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 }

```

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:
2      number): Promise<Uint8Array> {
3      return new Promise((resolve, reject) => {
4          const img = new Image();
5          img.onload = () => {
6              const canvas = document.createElement('canvas');
7              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    img.onerror = () => {
34        reject(new Error("Failed to load image"));
35    };
36
37    img.src = URL.createObjectURL(blob);
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()

```
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

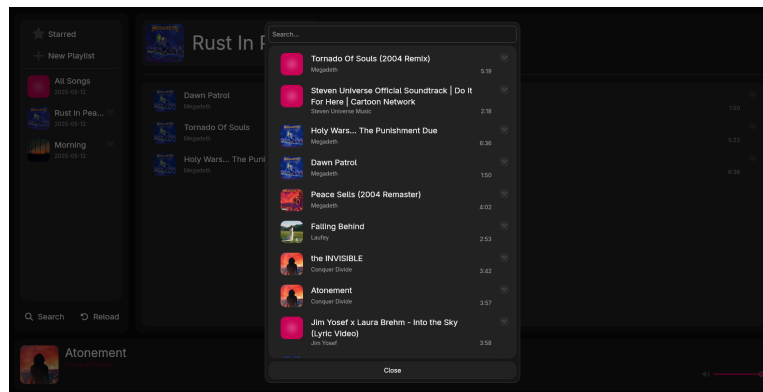


Figura 6: Menú de buscar canción

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 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

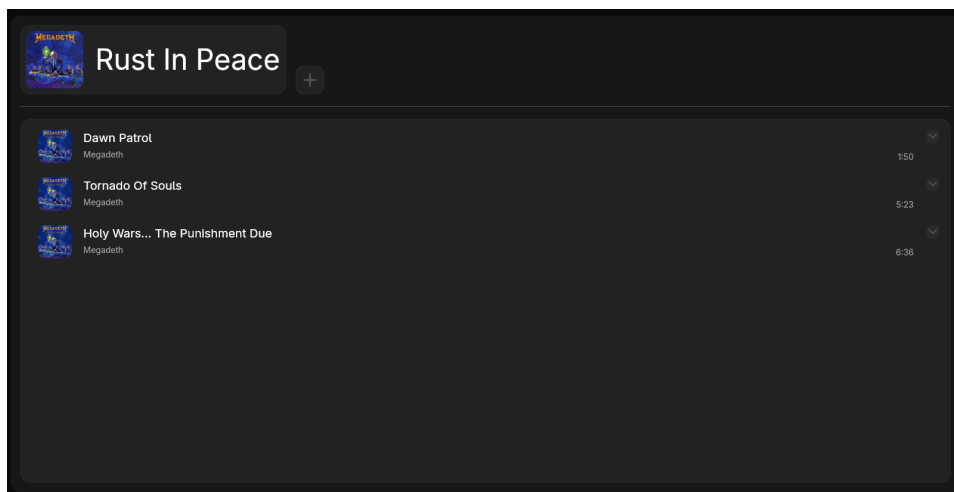


Figura 7: Mainscreen

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

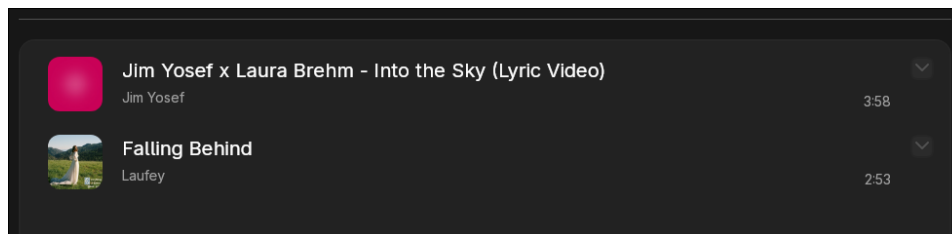


Figura 8: Song Button

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

```
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 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

```

1 interface Song {
2   id:string,
3   path:string,
4   title:string,
5   artist:string,
6   album:string,
7   year:string,
8   duration:string,
9   coverPath:string,
10  isStarred:boolean
11 }

```

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

Listing 12: getCoverPath()

```

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

```

```

10     URL.revokeObjectURL(this.coverUrl);
11   }
12
13   return URL.createObjectURL(blob);
14 }

```

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. (El manejo de la imagen predeterminada garantiza que siempre exista una representación visual, incluso cuando falla la carga de la portada original.).

Listing 13: Dropdown menu

```

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 }

```

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 14: `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 15: `addSongToPlaylist()`

```
1 addSongToPlaylist() {
2     this.songAdding.getAllPlaylists();
3     this.isModalOpen = true;
4 }
5
6 close() {
7     this.isModalOpen = false;
8     this.songAdding.letGo();
9 }
```

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 16: `removeSongFromPlaylist()`

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

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 17: `toggleStarred()`

```
1 async toggleStarred() {
```

```

2     const data_dir = await appDataDir();
3
4     if (this.isStarred) {
5         invoke('remove_is_starred', {song_id: this.id, db_path:
6             data_dir});
7         console.log("Canción !Starred: " + this.id);
8     } else {
9         invoke('add_is_starred', {song_id: this.id, db_path:
10            data_dir});
11         console.log("Canción Starred: " + this.id);
12     }
13
14     this.mainScreenStatus.refresh();
15 }

```

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 18: 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;

```

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 19: `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 }

```

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 20: `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  }

```

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 apli-

cación 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

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.