UC | Facultad de Ciencias

Desarrollo de un juego vía web para adivinar canciones

(Development of a web-based game to guess songs)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Rodrigo Fernández Silió

Director: Pablo Sánchez Barreiro

Septiembre - 2025

Resumen

En este Trabajo Fin de Máster se ha desarrollado una aplicación web de carácter lúdico que permite a los usuarios competir por adivinar canciones a partir de breves fragmentos de audio. La aplicación permite crear partidas personalizadas, buscar partidas existentes, unirse a ellas y jugar simultáneamente con otros jugadores.

Aunque existen plataformas similares, presentan ciertos inconvenientes. Por un lado, se encuentra la complejidad de uso, que dificulta la participación de los usuarios y genera barreras innecesarias para disfrutar del juego. Por otro lado, la oferta de listas de canciones suele ser limitada y poco variada, lo que restringe la personalización y disminuye la diversidad de las partidas.

En busca de superar estas limitaciones, la aplicación ofrece un entorno sencillo que facilita la participación y se integra con una plataforma de streaming musical, poniendo a disposición de los usuarios su amplio catálogo de listas de reproducción.

La aplicación se ha desarrollado siguiendo una arquitectura cliente-servidor, con un backend implementado en Spring Boot y un frontend desarrollado con Angular. El backend se encarga de gestionar las partidas y el desarrollo del juego. Por su parte, el frontend se limita a presentar la información y gestionar la interacción de los usuarios. Esta separación simplifica el desarrollo y mantenimiento de la aplicación, mejora su escalabilidad y garantiza una experiencia de usuario fluida y coherente.

Concretamente, el backend gestiona el ciclo de vida de las partidas, desde su creación hasta su finalización, pasando por su desarrollo, durante el cual establece las rondas y, en cada una, envía el fragmento de audio con las opciones de respuesta, valida las respuestas, calcula las puntuaciones y comunica estas y la respuesta correcta.

El frontend, por su parte, se centra en ofrecer una experiencia de usuario intuitiva y atractiva. Se encarga de la navegación entre menús, la gestión de formularios, la reproducción de fragmentos de audio, la captura de respuestas y la visualización tanto de la solución como de las puntuaciones.

La comunicación entre cliente y servidor se realiza principalmente mediante peticiones HTTP estándar. No obstante, durante el transcurso de una partida, se establece una conexión WebSocket por jugador, permitiendo un intercambio bidireccional: el servidor envía los datos de cada ronda, el cliente responde y, finalmente, el servidor devuelve la respuesta correcta y las puntuaciones.

Durante el desarrollo del proyecto, se realizaron múltiples pruebas para garantizar el correcto funcionamiento de la aplicación. Además, se desplegó en una plataforma de hosting, asegurando su accesibilidad desde cualquier equipo con conexión a Internet.

En resumen, este Trabajo Fin de Máster presenta una aplicación web completa que permite disfrutar de un juego musical, que supera las limitaciones de otras plataformas y proporciona una experiencia fluida, intuitiva y personalizable.

Palabras clave:
Aplicación web, Spring Boot, Angular, WebSocket, Despliegue en la nube.

Abstract

In this Master's Thesis, a playful web application was developed, allowing users to compete by guessing songs from short audio clips. The application offers the possibility to create custom games, search for existing games, join them and play simultaneously with other players.

Although similar platforms exist, they present certain drawbacks. On the one hand, the complexity of use makes it difficult for users to participate, creating unnecessary barriers to enjoying the game. On the other hand, the song lists offered are often limited and little varied, which restricts customization and reduce the diversity of the games.

Seeking to overcome these limitations, the application offers a simple environment that facilitates participation and integrates with a music streaming platform, making its extensive catalog of playlists available to users.

The application was developed following a client-server architecture, with a backend implemented in Spring Boot and a frontend developed with Angular. The backend manages the games and game flow, while the frontend focuses on presenting information and handling user interactions. This separation simplifies development and maintenance, improves scalability, and ensures a smooth and coherent user experience.

Specifically, the backend manages the game lifecycle, from creation to completion, including development, during which it establishes the rounds and, for each round, sends the audio clip with the answer options, validates the answers, calculates the scores, and communicates both the scores and the solution.

The frontend, in turn, focuses on offering an intuitive and engaging user experience. It handles menu navigation, form management, audio clip playback, response capture, and the display of both the correct answer and scores.

Communication between client and server is primarily done through standard HTTP requests. However, during the course of a game, a WebSocket connection is established for each player, allowing bidirectional exchange: the server sends the round data, the client responds, and finally, the server returns the correct answer and scores.

During the project's development, multiples tests were conducted to verify the correct functioning of the application. Furthermore, it was also deployed on a hosting platform, ensuring its accessibility from any device with an Internet connection.

In summary, this Master's Thesis presents a complete web application that enables users to enjoy a musical game, overcoming the limitations of other platforms and providing a fluid, intuitive, and customizable experience.

Keywords:
Web application, Spring Boot, Angular, WebSocket, Cloud deployment.

Agradecimientos

Con la finalización de este trabajo doy por concluida mi etapa académica, desde infantil hasta el máster, desde los 3 hasta los 24 años, 21 años de aprendizaje, que se dice pronto.

En este momento tan significativo, me gustaría aprovechar esta sección para expresar mi más sincero agradecimiento a todas las personas que, de una u otra manera, me han acompañado y apoyado a lo largo de esta etapa.

En primer lugar, agradezco profundamente a mis padres por su apoyo incondicional y por la educación en valores que me han transmitido. A mi hermano, por los momentos de alegría compartidos, y a mi hermano perruno, Rocky, que ya no está conmigo pero vivirá siempre en mi memoria, por su fiel compañía. Y, por supuesto, al resto de mi familia, por su cariño y cuidado constantes.

Quiero reconocer también la labor de mis profesores, cuya enseñanza ha sido fundamental para mi formación y desarrollo profesional.

A mis amigos y compañeros de clase, que muchas veces son lo mismo, gracias por las risas compartidas y por ayudarme a superar los desafíos a lo largo de esta etapa.

Por último, agradezco a mi novia, a quien quiero infinito, por su comprensión y motivación, que me han ayudado a mantenerme enfocado hasta la finalización de este trabajo.

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Introducción

En el Grado en Ingeniería Informática opté por la mención en Computación, lo que me permitió profundizar en áreas como algoritmos, inteligencia artificial, big data, etc. Sin embargo, esta elección también implicó dejar de lado otras áreas como el desarrollo web. Aunque durante mis estudios tuve contacto con tecnologías como Java, HTML o PHP, nunca llegué a desarrollar una aplicación web completa. Esta carencia se convirtió en una motivación personal. Al comenzar el Máster en Ingeniería Informática, tenía claro que uno de mis objetivos era completar mi formación construyendo una aplicación web de principio a fin como Trabajo Fin de Máster.

Durante el máster, esta motivación inicial se reforzó aún más gracias a una asignatura centrada en el desarrollo de aplicaciones web. En ella pude adquirir las bases fundamentales del desarrollo full stack utilizando tecnologías modernas, y además descubrí que disfrutaba especialmente del proceso. Gracias a esa asignatura, no solo confirmé que el desarrollo web era un área que me apasionaba, sino que también adquirí los conocimientos técnicos necesarios para afrontar con seguridad la creación de una aplicación web compleja.

Dado que aquella asignatura estaba limitada a seis créditos, su alcance estaba acotado a una aplicación sencilla. El Trabajo Fin de Máster, al contar con quince créditos, representaba una oportunidad perfecta para ir más allá: podía diseñar una aplicación más compleja, con más funcionalidades, y además, desplegarla en un servidor accesible públicamente. Este reto técnico y personal fue el punto de partida del trabajo que aquí se presenta.

Tras establecer el enfoque general del proyecto, el siguiente paso fue concretar una idea específica sobre la que construir la aplicación. La inspiración surgió de forma natural de una experiencia personal. Desde hace tiempo, tengo por costumbre jugar con un grupo de amigos a adivinar canciones escuchando únicamente los primeros segundos de cada una.

Pensé que sería interesante convertir este juego en una aplicación web interactiva, que permitiera recrear esa experiencia de forma online, y así nació la idea del proyecto.

El siguiente apartado describe de manera más detallada los objetivos concretos que debía tener la aplicación que quería construir.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Máster es desarrollar y desplegar una aplicación web funcional que permita a los usuarios jugar a adivinar canciones a partir de breves fragmentos de audio.

Para alcanzar este propósito, se han definido los siguientes objetivos específicos:

- Permitir la creación de partidas personalizadas, definiendo parámetros como la lista de reproducción utilizada, el modo de juego, el número de rondas, el número máximo de jugadores y la visibilidad de la partida (pública o privada, con código de acceso en este último caso).

- Ofrecer la posibilidad de visualizar las partidas disponibles, de manera que el usuario pueda unirse a una en la que ya haya participantes y, por tanto, exista una alta probabilidad de que se inicie en breve.
- Permitir a los usuarios unirse a partidas, ya sean públicas o privadas mediante la introducción del código de acceso.
- Desarrollar una experiencia de juego interactiva en la que, en cada ronda, los usuarios escuchen un fragmento musical, seleccionen entre varias opciones la que consideren correcta y, al finalizar la ronda, se muestre la canción correcta junto con la puntuación obtenida por cada jugador,
- Presentar un resumen al concluir la partida, mostrando la puntuación obtenida por cada jugador en cada ronda y en total, permitiendo así la comparación de resultados.

1.3. Metodología

Durante el desarrollo del proyecto se ha seguido una metodología ágil, inspirada en Scrum [11], adaptada a un entorno individual. La planificación y seguimiento de tareas se han llevado a cabo mediante el uso de Trello, permitiendo una gestión visual y ordenada del avance del proyecto. Además, se establecieron reuniones semanales en las que se revisaba el trabajo realizado, se evaluaban los avances, se ajustaban prioridades y se planificaban las tareas siguientes, garantizando así un desarrollo continuo y estructurado del proyecto.

En la fase inicial, se desarrolló un prototipo funcional, o *walking skeleton* en términos de metodologías ágiles, cuyo propósito principal era validar la viabilidad técnica del proyecto. Una de las necesidades clave de este proyecto era que existiese un sistema de comunicación fluido entre cliente y servidor que permitiese el intercambio rápido de información. Inicialmente se probó el uso de *WebHooks*, pero se descartaron por no ser adecuados para una comunicación bidireccional, ya que se basan en HTTP y requieren que ambos extremos actúen como servidores. Finalmente, se optó por *WebSockets*, que permiten una comunicación bidireccional entre servidor y cliente y proporcionan el comportamiento esperado para una experiencia de juego fluida en tiempo real.

Durante esta misma etapa también se llevaron a cabo pruebas con APIs de plataformas de streaming musical, necesarias para obtener los fragmentos de audio y la información de las canciones que se utilizarían en el juego. Estas pruebas permitieron evaluar aspectos como el formato de los datos, la disponibilidad del contenido y las restricciones de uso de cada API.

Se ha utilizado *Git* [2] como sistema de control de versiones para gestionar el código fuente y la configuración, y *GitHub* como repositorio remoto. Inicialmente, se trabajó en un repositorio *monorepo*, que incluía tanto el *backend* como el *frontend* de la aplicación. Más adelante, y con el objetivo de facilitar el proceso de despliegue en la nube, se crearon dos repositorios independientes, uno para el backend y otro para el frontend, permitiendo así una gestión más clara y modular en la etapa final del proyecto [1].

[1] Los tres repositorios del proyecto son los accesibles desde los siguientes enlaces:

1.4. Infraestructura

El backend de la aplicación se ha desarrollado con *Spring* [14], un *framework* basado en Java, mientras que el *frontend* se ha implementado utilizando *Angular*, un *framework* de desarrollo de interfaces web basado en *TypeScript* [6].

Durante el desarrollo, se realizaron pruebas de diferentes tipos. Para el backend y la comunicación entre cliente y servidor se emplearon herramientas como *Swagger* y *Bruno*, que facilitan la simulación y verificación de peticiones HTTP. Para el frontend se utilizaron herramientas de testing de Angular, como *ComponentFixture* y *TestBed*

Se han elegido estas tecnologías por tener familiaridad con ellas y ser adecuadas y suficientes para el desarrollo del proyecto.

Para el despliegue en producción, se ha utilizado la plataforma *Render*, un proveedor de servicios en la nube que permite alojar aplicaciones web. El *backend* y el *frontend* se han desplegado de forma independiente: el backend como un Web Service y el frontend como un Static Site.

2. Análisis de requisitos

2.1. Requisitos funcionales

Dado que el proyecto corresponde a un juego desarrollado de forma personal, no se realizó un proceso formal de captura de requisitos, como entrevistas, encuestas o análisis de usuarios reales. Básicamente, el proyecto debía satisfacer las necesidades que tenía en mente. Por tanto, simplemente se especificaron una serie de funcionalidades iniciales que se fueron refinando y complementando a través del desarrollo del proyecto.

Para organizar y especificar los requisitos funcionales se emplearon *historias de usuario*, una técnica ampliamente utilizada en metodologías ágiles para especificar requisitos. Estas historias describen funcionalidades atómicas que aportan valor a algún usuario, lo que las hace conceptualmente similares a los casos de uso tradicionales, pero con un enfoque más práctico y centrado en la experiencia del usuario.

A continuación, en las Tablas 1 y 2 se muestran dos ejemplos de historias de usuario empleadas durante el desarrollo del proyecto.

La primera describe la funcionalidad de creación de partidas. Permite al usuario configurar partidas públicas o privadas, seleccionar una lista de reproducción, establecer el número de rondas y el máximo de jugadores, e introducir una contraseña en caso de partida privada. Se contemplaron también escenarios de validación, como campos incompletos o incorrectos, y la gestión de errores relacionados con la disponibilidad del servidor o fallos internos.

- Repositorio monorepo
- Repositorio backend
- Repositorio frontend

La segunda se centra en la visualización de partidas disponibles. El usuario puede consultar partidas activas y no iniciadas, ordenadas según la cantidad de jugadores restantes, y acceder a su información esencial. Al igual que en la primera historia, se verificaron casos de fallo del servidor y errores internos, garantizando que el sistema mostrara los mensajes de error adecuados.

El conjunto completo de historias de usuario que guiaron el desarrollo está documentado en el tablero de Trello utilizado durante el proyecto[2].

Tabla 1: US001 - Crear una partida

Historia de usuario	Yo, como usuario, quiero crear una partida con unas características determinadas, de manera que pueda pasar un rato agradable jugando con otras personas.
Prueba 1. Éxito. Crear una partida pública.	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario busca y selecciona una playlist. 4. El usuario completa los campos relativos a la configuración de la partida (número de rondas y número máximo de jugadores). 5. El usuario no selecciona la opción “Partida privada”. 6. El usuario crea la partida. 7. Se verifica que la partida esté correctamente registrada en el sistema.
Prueba 2. Éxito. Crear una partida privada.	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario busca y selecciona una playlist. 4. El usuario completa los campos relativos a la configuración de la partida (número de rondas y número máximo de jugadores). 5. El usuario selecciona la opción “Partida privada”. 6. El sistema habilita el campo “Contraseña”. 7. El usuario ingresa la contraseña. 8. El usuario crea la partida. 9. Se verifica que la partida esté correctamente registrada en el sistema. Yo, como usuario, quiero crear una partida con unas características determinadas, de manera que pueda pasar un rato agradable jugando con otras personas.
Prueba 3. Campos en blanco	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario completa el formulario dejando uno o más campos en blanco. 4. El usuario intenta crear la partida. 5. Se verifica que el sistema informe correctamente sobre los campos vacíos y que la partida no esté creada.

[2]Accesible en el siguiente enlace: Tablero de Trello

Prueba 4. Campos completados erróneamente	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario completa el formulario introduciendo datos erróneos en uno o más campos. 4. El usuario intenta crear la partida. 5. Se verifica que el sistema informe correctamente sobre los campos mal completados y que la partida no esté creada.
Prueba 6. Servidor no disponible.	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario completa el formulario. 4. El usuario intenta crear la partida. 5. Se simula que el servidor no está disponible. 6. Se verifica que el sistema muestre un mensaje de error indicando “Servidor no disponible” y que la partida no esté creada.
Prueba 7. Error 500	1. El usuario selecciona la opción “Crear partida”. 2. El sistema muestra el formulario para crear una partida. 3. El usuario completa el formulario. 4. El usuario intenta crear la partida. 5. Se simula que el servidor devuelve un error 500. 6. Se verifica que el sistema muestre un mensaje de error indicando “Error 500” y que la partida no esté creada.

Tabla 2: US002 - Ver partidas disponibles

Historia de usuario	Yo, como usuario, quiero ver partidas disponibles, de manera que pueda unirme a una donde ya haya gente y haya una cierta probabilidad de iniciarse en breve
Prueba 1. Éxito. Ver partidas disponibles	1. El usuario selecciona la opción “Buscar partidas”. 2. El sistema muestra una lista de partidas disponibles (públicas y no iniciadas), ordenadas según la cantidad de jugadores restantes para completar el aforo. 3. Para cada partida, el sistema muestra su información esencial.
Prueba 2. Servidor no disponible	1. El usuario selecciona la opción “Buscar partidas”. 2. Se simula que el servidor no está disponible. 3. Se verifica que el sistema muestre un mensaje de error indicando “Servidor no disponible” y que no se cargue la lista de partidas.
Prueba 3. Error 500	1. El usuario selecciona la opción “Buscar partidas”. 2. Se simula que el servidor devuelve un error 500. 3. Se verifica que el sistema muestre un mensaje de error indicando “Error 500” y que no se cargue la lista de partidas.

2.2. Requisitos no funcionales

En cuanto a los requisitos no funcionales, se revisó la norma ISO 25010 [7], llegándose a la conclusión de que, con carácter general, la aplicación no tenía que satisfacer ningún requisito no funcional de manera especial. Es decir, el sistema debía tener buen rendimiento, usabilidad o mantenibilidad en la misma medida que cualquier otra aplicación informática adecuadamente diseñada, pero sin que existiese ninguna característica o restricción concreta que la aplicación tuviese que satisfacer en especial.

No obstante, si era importante que la aplicación fuese capaz de mantener una comunicación con el servidor fluida, de manera que las partidas se pudiesen desarrollar de manera dinámica y sin interrupciones. Por tanto, era importante mantener unos tiempos de respuesta en la comunicación con el servidor bajos, aunque sin que existiese un umbral concreto para ello, y asegurar una cierta tolerancia a fallos, permitiendo que las partidas pudiesen continuar aún cuando hubiese pequeñas pérdidas de conexión con el servidor.

3. Arquitectura

La aplicación consiste en un juego multijugador interactivo donde los usuarios deben adivinar canciones a partir de un fragmento de audio. Está compuesta por dos servicios principales: un backend en Spring Boot y un frontend en Angular, ambos desplegados de forma independiente en Render.

En la Figura 1 se ilustra la estructura general del sistema, incluyendo los componentes principales y las comunicaciones entre ellos.

A continuación, se describen con mayor detalle los componentes del backend y del frontend.

3.1. Backend

El backend, desarrollado con Spring Boot y desplegado en Render como un Web Service a través de Docker, es responsable de toda la lógica del juego. Expone una API RESTful que permite la gestión de partidas, incluyendo operaciones como crear una nueva partida, buscar partidas disponibles o unirse a una partida existente.

Durante el transcurso de una partida, la comunicación entre clientes y servidor se realiza mediante WebSockets. El backend marca el ritmo del juego: al inicio de cada ronda, envía a los jugadores un fragmento de audio junto con cuatro opciones de canciones. Recibe las respuestas de los jugadores, las almacena, espera a que la ronda finalice y, a continuación, calcula las puntuaciones en función de la precisión y la rapidez. Finalmente, envía a los jugadores los resultados de la ronda. Este proceso se repite durante un número definido de rondas hasta finalizar la partida.

El backend se comunica con la API externa de la plataforma de streaming musical en dos momentos clave. El primero ocurre durante la creación de la partida: el usuario que la configura puede buscar listas de reproducción por nombre, lo que genera una solicitud al backend. Este, a su vez, realiza una petición a la API del servicio de streaming musical, obtiene las listas coincidentes con esos datos, las adapta a objetos

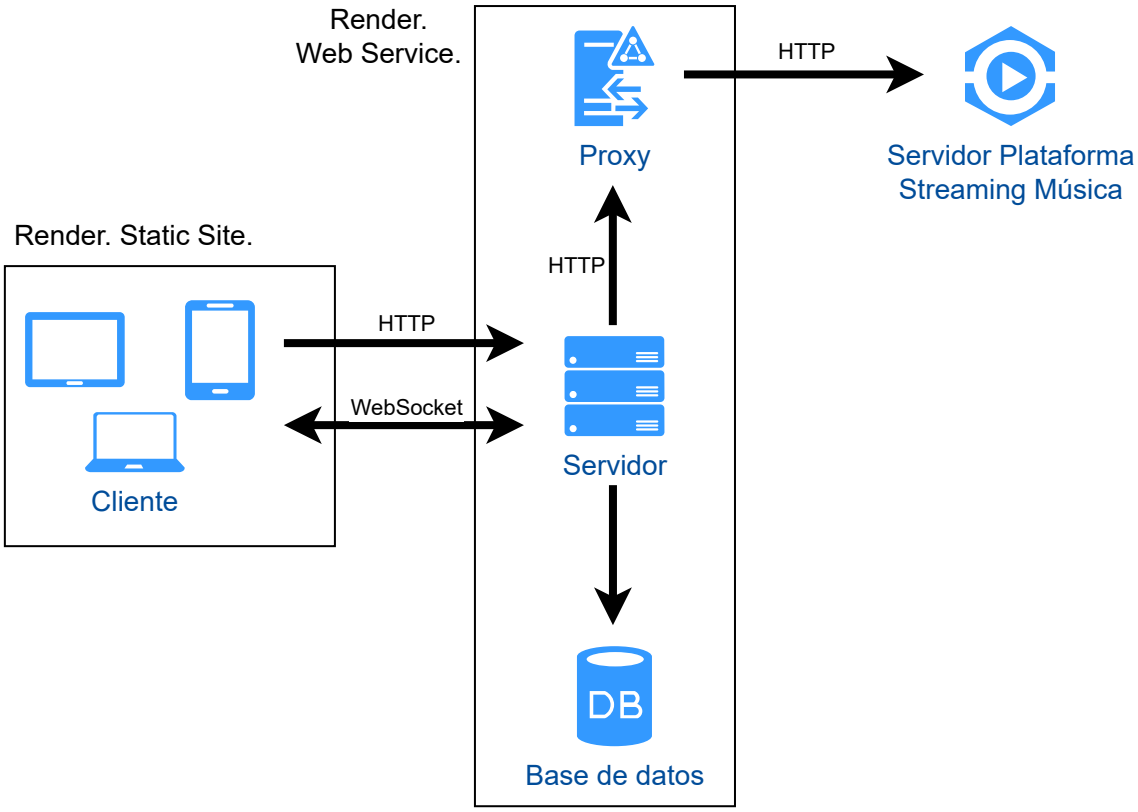

Figura 1: Arquitectura

internos de dominio con solo la información relevante y las envía al frontend para su visualización. El usuario puede repetir la búsqueda tantas veces como desee hasta seleccionar una lista de reproducción concreta con la que iniciar la partida.

El segundo momento de comunicación con la API externa ocurre justo antes de comenzar la partida: el backend utiliza el identificador de la lista de reproducción para obtener las canciones que la componen. Con esta información, genera todas las rondas de la partida, seleccionando cuatro canciones por ronda y determinando cuál será la opción correcta. Este proceso se realiza una única vez por partida.

Al desplegar en Render, la API de la plataforma de streaming musical dejó de aceptar cualquier solicitud, probablemente debido a que su firewall o políticas de seguridad bloquean las peticiones provenientes de Render y servicios similares. Por este motivo, fue necesario añadir un proxy intermedio que actúa como puente entre el backend y la API, permitiendo que las solicitudes se realicen correctamente y se mantenga la funcionalidad del sistema. En concreto, se utilizó el proxy público thingproxy.

Para gestionar el estado de las partidas activas y los datos asociados al juego en curso, se emplea una base de datos H2 en memoria. Esta solución resulta adecuada dado que no existe registro de usuarios ni necesidad de persistencia a largo plazo; es suficiente con conservar la información durante la duración de cada partida.

3.2. Frontend

El frontend está desarrollado con Angular y desplegado en Render como Static Site. Es responsable de la interfaz de usuario y de gestionar la experiencia completa del jugador durante la partida.

La comunicación con el backend se realiza por dos canales complementarios. Por un lado, utiliza peticiones HTTP para operaciones como la creación de partidas, la búsqueda de partidas disponibles y la unión a una de ellas. Por otro lado, emplea WebSockets para toda la interacción durante la partida, incluyendo la recepción de rondas, el envío de respuestas y la recepción de los resultados y puntuaciones tras cada ronda.

El frontend controla el flujo del juego desde la perspectiva del usuario: muestra las rondas a medida que se reciben, permite escuchar los fragmentos de audio, facilita la selección de una respuesta y presenta de forma dinámica los resultados y la puntuación acumulada.

4. Implementación

En esta sección se describe el desarrollo de la aplicación, comenzando por el modelo de dominio, que define las entidades y sus relaciones fundamentales. A continuación se presentan las capas de la aplicación siguiendo el orden real de implementación: primero la capa de persistencia, luego la de control y, finalmente, la de servicio. Este enfoque, aunque diferente al orden canónico de explicación de aplicaciones multicapa, facilita la comprensión del proceso. Seguidamente, se dedica un apartado independiente a la integración con la API de la plataforma de streaming musical, dada su elevada complejidad y su impacto en todas las capas. Posteriormente se aborda el desarrollo del frontend y, a continuación, la implementación de la lógica de juego, que requiere tareas programadas y escucha de eventos, elementos adicionales al patrón típico de una aplicación Spring. Finalmente, se explica la comunicación bidireccional en tiempo real mediante WebSockets, se presentan las pruebas realizadas y se detalla el despliegue de la aplicación.

Esta estructura permite ofrecer una visión completa tanto del entramado técnico estándar de una aplicación Spring como de las funcionalidades y mecanismos adicionales implementados para cumplir los objetivos del proyecto.

4.1. Modelo de dominio

El proceso de desarrollo comenzó con la elaboración de un modelo de dominio, que se muestra en la Figura 2.

Los datos de listas de reproducción y canciones se obtienen de una base de datos externa mediante la API de la plataforma de streaming musical. Sin embargo, estas dos clases forman parte del modelo de dominio porque representan conceptos fundamentales para el juego. Además, permiten encapsular de forma sencilla la información mínima necesaria recibida desde la API externa, facilitando la gestión de estos datos dentro del sistema.

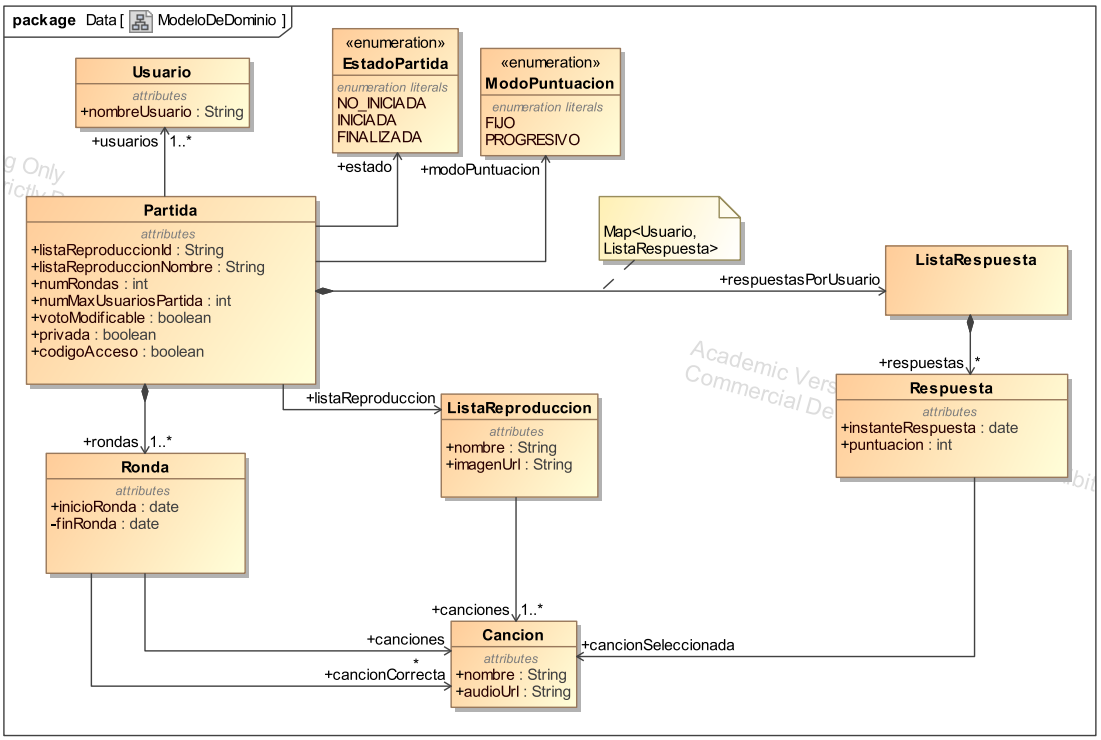

Figura 2: Modelo de dominio

Posteriormente, se implementó el modelo de dominio en Java, utilizando POJOs (Plain Old Java Objects) dentro de un proyecto basado en Spring, sentando así la base para el desarrollo de las capas superiores del sistema.

Como ejemplo, se presenta el POJO de la clase Usuario (Figura 3).

```java
@Data
@RequiredArgsConstructor
public class Usuario {

    @NonNull
    private String nombre;
}
```

Figura 3: Código del POJO Usuario, con anotaciones Lombok

Este POJO se define de manera concisa utilizando anotaciones de la biblioteca Lombok, que permite reducir el código repetitivo y mejorar la legibilidad y mantenibilidad. En particular, la anotación `@Data` genera automáticamente los métodos getters, setters, equals, hashCode y toString. Por otro lado, la combinación de `@RequiredArgsConstructor` y `@NonNull` genera un constructor que incluye únicamente los campos marcados como no nulos, en este caso únicamente el nombre.

4.2. Capa de persistencia

Para habilitar la persistencia del modelo de dominio, se adaptaron los POJOs creados previamente, convirtiéndolos en entidades JPA. Las clases que representan entidades fueron anotadas con `@Entity`, indicando que deben mapearse a tablas en la base de datos. Por otro lado, los *value objects* se marcaron con `@Embeddable`, lo que permite que sus atributos se integren directamente dentro de la entidad que los contiene. Además, se añadió un identificador a cada entidad junto con su correspondiente anotación `@Id`. Para los identificadores de todas las entidades se optó por una estrategia de generación automática mediante `@GeneratedValue`, lo que permite delegar a la base de datos la creación de los valores del identificador, simplificando así la lógica de creación de objetos.

A continuación se muestra la clase Usuario adaptada como entidad JPA (Código 4).

```
@Data
@NoArgsConstructor
@RequiredArgsConstructor
@Entity
public class Usuario {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @NonNull
    private String nombre;
}
```

Figura 4: Código de la clase Usuario adaptada como entidad JPA

Como puede observarse, la clase está anotada con `@Entity`, y se ha incorporado el atributo `id` con sus correspondientes anotaciones `@Id` y `@GeneratedValue`. Se sigue utilizando la biblioteca Lombok para reducir el código repetitivo. La anotación `@NoArgsConstructor` genera un constructor sin argumentos, requisito indispensable para que JPA pueda instanciar la entidad mediante reflexión. Esto permite que el framework cree objetos en tiempo de ejecución sin necesidad de invocar explícitamente un constructor con parámetros. Por otro lado, `@RequiredArgsConstructor` sigue siendo funcional porque solo el atributo `nombre` está anotado con `@NonNull`. Esto asegura que al crear un nuevo objeto se siga proporcionando solo el nombre, mientras que el identificador se genere automáticamente, manteniendo la coherencia del modelo.

Continuando con la habilitación de la persistencia, toda propiedad de una entidad que represente una asociación con otra entidad debe anotarse según la cardinalidad de la relación, utilizando `@OneToOne`, `@OneToMany`, `@ManyToOne` o `@ManyToMany` según corresponda. De este modo, se garantiza el mapeo preciso de las relaciones entre entidades en la base de datos.

Por ejemplo, en mi aplicación, la clase *Partida* tiene una lista de usuarios. Este atributo está anotado con `@ManyToMany` porque una partida puede incluir varios usuarios y, a su vez, un usuario puede participar en múltiples partidas. Por otro lado, la clase *Partida* tiene una lista de rondas anotada con `@OneToMany`, ya que una partida puede contener varias rondas, pero cada ronda pertenece únicamente a una partida.

Es importante indicar, además, para las asociaciones bidireccionales que lo requieran, cuál es la referencia opuesta correspondiente para mantener la integridad y coherencia del modelo. En mi caso particular, no existen asociaciones bidireccionales, por lo que este aspecto no es aplicable.

También, es fundamental elegir una estrategia adecuada de propagación de operaciones cuando se considere necesario para gestionar el ciclo de vida de las entidades relacionadas. En esta aplicación en particular, la entidad *Ronda* depende totalmente de *Partida*, ya que una ronda no tiene sentido sin estar asociada a una partida. Por eso, se utiliza `cascade = CascadeType.ALL` para que todas las operaciones (persistencia, actualización, eliminación, etc.) realizadas sobre una partida se propaguen automáticamente a sus rondas asociadas, garantizando la coherencia y evitando referencias huérfanas.

Por último, se crearon los repositorios necesarios para la gestión de las entidades en la aplicación, específicamente para *Partida* y *Usuario*. Estos repositorios permiten acceder a los datos, guardar nuevas instancias, actualizar información o realizar consultas personalizadas.

A continuación, en la Figura 5, se muestra el repositorio correspondiente a Usuario. Este repositorio extiende `JpaRepository`, lo que permite simplificar la implementación de operaciones CRUD (Create, Read, Update y Delete) y consultas básicas sin necesidad de escribir código adicional, ya que estas operaciones las genera Spring de manera automática simplemente al heredar de la interfaz.

```
package
↪ adivina_la_cancion.prototipo.adivina_la_cancion.repositories;

import org.springframework.data.jpa.repository.JpaRepository;

import
↪ adivina_la_cancion.prototipo.adivina_la_cancion.domain.Usuario;

public interface UsuarioRepository extends JpaRepository<Usuario,
↪ Long> {

}
```

Figura 5: Código del repositorio Usuario

4.3. Capa de control

La capa de control está compuesta por los controladores, cuya función es exponer los endpoints de la aplicación a través de peticiones HTTP. Un controlador se encarga de recibir las peticiones HTTP del cliente, interpretar los parámetros necesarios y delegar el procesamiento en la capa de servicio. Finalmente, devuelve la respuesta correspondiente al cliente.

En esta aplicación se ha desarrollado una API REST (Representational State Transfer), lo que implica que los recursos del sistema se exponen a través de URIs (Universal Resource Identificator) bien definidas y se accede a ellos utilizando los verbos HTTP estándar (GET, POST, PUT, DELETE).

Para implementar esta capa, se identifican los recursos que conforman la API del sistema, los cuales corresponden a las entidades principales del dominio: *Partida*, *Usuario* y *ListaReproduccion*.

Partida es un recurso porque requiere operaciones como la creación de nuevas partidas y la obtención de las partidas para visualizar en la interfaz. De forma similar, para la entidad *Usuario* se necesitan operaciones básicas de creación. *ListaReproduccion* también se considera un recurso, aunque con una particularidad: en lugar de almacenar y consultar sus datos en la base de datos propia, la aplicación recupera la información desde la base de datos externa de la plataforma de streaming musical mediante solicitudes a su API.

A continuación, se diseña una URI adecuada y clara para cada recurso. También, se determinan los verbos HTTP aplicables a cada recurso en función de las operaciones que se deseen ofrecer (por ejemplo, `GET` para obtener información o `POST` para crear nuevos elementos).

En las tablas 3, 4 y 5 se resumen los endpoints de la API REST del sistema, indicando para cada uno la URI correspondiente, el método HTTP que se utiliza y los códigos de respuesta esperados. Esta información proporciona una visión clara de cómo interactuar con el backend y sirve como guía para el desarrollo de clientes que consuman estos servicios.

Tabla 3: API REST del recurso Usuario

Recurso	**Método**	**Código/s de respuesta**
/usuarios/{usuarioNombre}	POST	200

Tabla 4: API REST del recurso Partida

Recurso	**Método**	**Código/s de respuesta**
/partidas	GET	200
/partidas	POST	200, 400, 404
/partidas/{partidaID} /{usuarioID}/anhadirUsuario	PUT	200, 400, 404

Tabla 5: API REST del recurso Música

Recurso	Método	Código/s de respuesta
/musica/playlists	GET	200, 500
/musica/playlists/{playlistId}	GET	200, 400, 404

Para implementar lo anteriormente mencionado, es suficiente con crear tres clases controladoras, cada una correspondiente a un recurso identificado previamente. Estas clases deben estar anotadas con `@RestController`, lo que permite que Spring las reconozca como controladores REST y exponga automáticamente los endpoints definidos en su interior.

Para facilitar el cumplimiento de las convenciones REST, en la definición de cada clase controladora se incluye la anotación `@RequestMapping` con la URI base correspondiente, por ejemplo `@RequestMapping(/partidas")` para *Partida*. Esta URI actúa como ruta inicial común, de modo que todos los endpoints definidos en esa clase comenzarán con dicha ruta, agrupando así de forma coherente las operaciones relacionadas con este recurso.

Una vez definida la ruta base, se implementan los distintos endpoints. A continuación, se muestran dos ejemplos ilustrativos. Por un lado, la Figura 6 muestra la obtención de partidas. En este caso, se utiliza la anotación `@GetMapping` dado que la operación se corresponde con el verbo GET, empleado para recuperar información sin modificar el estado del sistema, porque se está obteniendo una lista de partidas. Como se ha mencionado antes, no es necesario añadir `/partidas` en el `@GetMapping`, ya que esta ruta base está definida previamente mediante la anotación `@RequestMapping` a nivel de clase.

```
@GetMapping
@JsonView({ Views.PartidaPreview.class })
public ResponseEntity<List<Partida>> obtenerPartidasDisponibles()
 ↪  {
    return partidaService.obtenerPartidasDisponibles();
}
```

Figura 6: Código del endpoint para obtener partidas

La anotación `@JsonView(Views.PartidaPreview.class)` permite controlar los atributos de *Partida* que serán serializados en la respuesta JSON. Esto es útil para evitar exponer información sensible (como `codigoDeAcceso`) o irrelevante (como `listaReproduccionId`), mostrando únicamente los datos necesarios.

Para que esto funcione correctamente, las propiedades que se deseen incluir deben estar anotadas en la clase *Partida*, tal como se muestra en la Figura 7. Por ejemplo, `listaReproduccionNombre`, que es clave para que el usuario conozca las posibles canciones que sonarán durante la partida, queda anotada para ser enviada como respuesta a la petición HTTP.

Por último, el método retorna un objeto `ResponseEntity<List<Partida>>`, que

```
class Partida {
    ...

    @JsonView({ Views.PartidaPreview.class })
    private String listaReproduccionNombre;

    ...
}
```

Figura 7: Fragmento de código de la clase partida con anotación `@JsonView`

permite devolver tanto el cuerpo de la respuesta como el código de estado HTTP. En caso de éxito, se devuelve una lista de partidas con un código 200 OK. Si ocurre un error, se puede devolver un cuerpo vacío junto con el código de error adecuado.

Por otro lado, la Figura 8 muestra el ejemplo de creación de un usuario. Aquí se emplea `@PostMapping(/usuarioNombre")` porque se trata de una operación de creación, y por tanto se usa el verbo POST. El nombre del usuario se extrae directamente de la URI gracias al segmento {`usuarioNombre`}, y se vincula al parámetro del método mediante la anotación `@PathVariable String usuarioNombre`. Además, se utiliza la anotación `@Transactional`, que garantiza que todas las operaciones realizadas dentro del método se ejecuten en una única transacción.

```
@PostMapping("/{usuarioNombre}")
@Transactional
public ResponseEntity<Long> crearUsuario(@PathVariable String
↪  usuarioNombre) {
    return usuarioService.crearUsuario(usuarioNombre);
}
```

Figura 8: Código del endpoint de creación de usuario

4.4. Capa de servicio

La capa de servicio contiene la lógica de la aplicación y actúa como intermediaria entre la capa de control y la capa de persistencia. Al igual que en la capa de control, cada recurso principal dispone de su propio servicio, encargado de implementar la lógica correspondiente a ese recurso.

Los servicios están anotados con `@Service`, lo que permite a Spring detectarlos automáticamente y gestionarlos como componentes del sistema. La Figura 9 muestra un servicio simplificado con un método básico.

Como se observa, el servicio interactúa con su repositorio correspondiente para realizar, en este caso, una inserción, aunque también podría llevar a cabo otras operaciones como búsquedas, actualizaciones o eliminaciones. El método devuelve un

```java
@Service
public class UsuarioService {
    ...

    public ResponseEntity<Long> crearUsuario(String
    ↪ usuarioNombre) {
        Usuario usuario = new Usuario(usuarioNombre);
        usuarioRepo.save(usuario);
        return new ResponseEntity<Long>(usuario.getId(),
        ↪ HttpStatus.OK);
    }

    ...
}
```

Figura 9: Fragmento de código del servicio UsuarioService

`ResponseEntity`, lo que permite controlar tanto el contenido de la respuesta como el código de estado HTTP.

La Figura 10 muestra un método más complejo para que se aprecie el alcance de lo que se puede gestionar en esta capa. Este método gestiona operaciones de negocio complejas, combinando tanto el acceso a repositorios como la interacción con otros servicios.

El principal objetivo de este método es crear una partida con los datos proporcionados en un objeto de la clase `PartidaDTO`. En primer lugar, el método utiliza una instancia del servicio `MusicaService` para recuperar la lista de reproducción asociada al identificador especificado en `partidaDTO`. A continuación, consulta el repositorio de usuarios para obtener el usuario que solicita la creación de la partida. Estos pasos iniciales permiten garantizar que los recursos necesarios para crear la partida están disponibles y se pueden utilizar en los pasos posteriores.

Una vez obtenidos los datos, el método continúa únicamente si la recuperación de la lista de reproducción ha resultado exitosa y si el usuario existe en la base de datos. Si cualquiera de estas condiciones no se cumple, el método devuelve un `ResponseEntity` con código 404 Not Found, informando al cliente de que alguno de los recursos no estaba disponible.

A continuación, se intenta añadir el usuario creador a la nueva partida mediante el método `partida.anhadirUsuario`. Este método garantiza que la incorporación del usuario respeta todas las reglas de negocio, que son que la partida no haya comenzado, que no se haya alcanzado el número máximo de participantes, que el usuario no esté previamente registrado en la partida y, en caso de partidas privadas, que se proporcione un código de acceso correcto. Si alguna de estas condiciones no se cumple, la operación falla y se devuelve un ResponseEntity con código 400 Bad Request.

Si el usuario se añade correctamente, el método persiste la nueva partida en la

```
public ResponseEntity<Partida> crearPartida(PartidaDTO pDTO) {
    ResponseEntity<ListaReproduccion>
    ↪ listaReproduccionResponseEntity = musicaService
            .obtenerPlaylistValida(pDTO.getPlaylistID());
    Optional<Usuario> usuarioOptional =
    ↪ ur.findById(pDTO.getUsuarioID());

    if (listaReproduccionResponseEntity.getStatusCode() ==
    ↪ HttpStatus.OK && usuarioOptional.isPresent()) {
        ListaReproduccion listaReproduccion =
        ↪ listaReproduccionResponseEntity.getBody();
        Usuario usuario = usuarioOptional.get();

        Partida partida = new Partida(listaReproduccion,
        ↪ pDTO.getNumRondas(), pDTO.getNumMaxUsuariosPartida(),
        ↪ pDTO.isVotoModificable(), pDTO.getModoPuntuacion(),
        ↪ pDTO.isPrivada(), pDTO.getCodigoAcceso());

        if (partida.anhadirUsuario(usuario,
        ↪ pDTO.getCodigoAcceso())) {
            partidaRepo.save(partida);

            // Tras añadirlo, comprobar si se ha llenado
            if (partida.getUsuarios().size() ==
            ↪ partida.getNumMaxUsuariosPartida()) {
                // Si la partida se ha llenado, se inicia
                iniciarPartidaAsync(partida);
            }

            return new ResponseEntity<>(partida, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
        }
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Figura 10: Código del método del servicio para crear una partida

base de datos. Tras la inserción, se verifica si la partida ha alcanzado su capacidad máxima de participantes. En caso afirmativo, la partida se inicia de forma asíncrona, evitando bloquear el flujo de creación y respetando los principios de desacoplamiento entre operaciones.

Finalmente, cuando todo el proceso se completa de manera satisfactoria, el método devuelve la partida recién creada junto con un código 200 OK.

4.5. Integración con la API de la plataforma de streaming musical

Inicialmente se consideró utilizar Spotify como proveedor de contenido musical, ya que es el líder del mercado de streaming y dispone de un catálogo muy amplio de listas de reproducción y canciones. Dada su popularidad, era razonable suponer que su API contaría con desarrolladores experimentados, buen soporte, una comunidad numerosa y abundantes ejemplos de uso.

Durante la fase inicial de validación de la viabilidad del proyecto, se realizaron pruebas de la API de Spotify de manera independiente, utilizando Python y la librería Spotipy. Se comprobó que era posible buscar listas de reproducción por nombre, obtener sus metadatos y las canciones que las componen, incluyendo a su vez los metadatos y la pervisualización de cada canción.

En principio, la API ofrecía todas las funcionalidades necesarias. No obstante, se detectaron comentarios sobre cambios frecuentes y documentación limitada [12].

Tras completar el resto de pruebas aisladas, se inició el desarrollo del prototipo, pero integrando todos los componentes. Entre la finalización de las pruebas aisladas de la API de Spotify y su integración en el prototipo transcurrió un tiempo no despreciable.

Al llevar a cabo esta integración, se detectó que las muestras de las canciones dejaron de estar disponibles, apareciendo todas con valor nulo, lo cual fue confirmado tras una búsqueda en distintas fuentes. Según la información recopilada, esta eliminación se debía, según un usuario, a la intención de impedir su uso en procesos de aprendizaje automático [8], hecho que fue corroborado parcialmente por la comunicación oficial de Spotify [10]. Esta circunstancia suponía un problema grave, ya que la reproducción de fragmentos de audio era esencial para el funcionamiento del juego.

Frente a este problema se contemplaron dos alternativas: aplicar soluciones para extraer previews desde otras fuentes de Spotify o migrar a la API de otra plataforma de streaming musical. La primera opción se consideró frágil y de difícil mantenimiento, por lo que fue descartada, y en consecuencia se optó finalmente por la segunda.

Entre las múltiples plataformas de streaming existentes, se optó por Deezer, que aunque no posee un catálogo tan amplio como Spotify, es suficientemente completo. Además, su API es excelente, ya que es sencilla de utilizar, no requiere autenticación para consultas básicas, como la búsqueda de canciones o listas de reproducción, y proporciona muestras de audio de manera constante. Gracias a Deezer y su API, fue posible mantener la funcionalidad prevista y garantizar la continuidad del proyecto.

Tras explicar la elección de la plataforma, desde la idea inicial con Spotify hasta la decisión final de utilizar Deezer, a continuación se presenta una descripción de su implementación.

Como se mencionó anteriormente, aunque la aplicación accede únicamente a listas de reproducción y no a canciones, la información de las canciones que forman cada lista debe ser procesada. Por tanto, resulta conveniente un servicio que abarque ambos conceptos.

En consecuencia, en la aplicación no existe un servicio exclusivo para la gestión de listas de reproducción. En su lugar, esta responsabilidad recae en `MusicaService`, un componente que cumple un papel más amplio, gestionando no solo las listas de reproducción, sino también las canciones que las componen y la relación entre ambas.

Este componente ofrece funciones de alto nivel que internamente delegan en DeezerService, que es el componente responsable de realizar las llamadas directas a la API de la plataforma de streaming musical Deezer.

La separación entre MusicaService y DeezerService se realiza por motivos de flexibilidad y mantenibilidad. De este modo, si en el futuro se decidiera migrar a otra plataforma, únicamente sería necesario implementar un nuevo servicio específico para esa plataforma y actualizar `MusicaService` para delegar en él, sin afectar al resto de la aplicación.

A continuación, con el fin de facilitar la comprensión de la estructura de la clase y del modo en que se realizan las llamadas a la API, se muestra en la Figura 11 la implementación del método buscarPlaylists, que permite buscar listas de reproducción en Deezer según un término y parámetros de paginación.

```
public PlaylistData buscarPlaylists(String q, int index, int
↪  limit) {
    String url = UriComponentsBuilder.fromHttpUrl(DEEZER_API_URL)
            .path("/search/playlist")
            .queryParam("q", q)
            .queryParam("index", index)
            .queryParam("limit", limit)
            .toUriString();

    try {
        String jsonResponse = restTemplate.getForObject(url,
        ↪  String.class);

        jsonResponse = asegurarCampoNext(jsonResponse);

        Gson gson = new Gson();
        return gson.fromJson(jsonResponse, PlaylistData.class);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Figura 11: Código del método para buscar playlists

Este método utiliza la clase UriComponentsBuilder para ensamblar la URL de la petición de forma sencilla, evitando errores de concatenación manual y manteniendo un código legible. La construcción parte de la URL base, que es una constante con

valor 'https:// api.deezer.com', que se corresponde con la raíz de la API desde la cual se accede a todos sus endpoints. A la URL base se le añade */search/playlist*, que se corresponde con el endpoint de la API encargado de devolver listas de reproducción que coinciden con un término de búsqueda [4].

Por último, se incorporan los tres parámetros de consulta necesarios:

- `q`: especifica el término de búsqueda
- `index`: indica la posición inicial desde la que se desean obtener los resultados
- `limit`: define el número máximo de resultados por página.

Una vez construida la URL, se realiza la solicitud HTTP GET mediante la librería RestTemplate. La respuesta se obtiene como una cadena de texto en formato JSON, lo que facilita su manipulación antes de la deserialización. La deserialización consiste en transformar la cadena JSON en un objeto Java con estructura tipada, lo que permite acceder a los datos de manera ordenada y segura.

Para la deserialización se emplea la clase `PlaylistData` que pertenece a una librería que facilita la interacción con la API de Deezer [13]. Esta clase se utiliza en lugar de implementar una propia, ya que proporciona una representación completa y compatible con el formato devuelto por la API, evitando la necesidad de desarrollar y mantener estructuras redundantes.

Un detalle a tener en cuenta es que Deezer omite el campo `next` en la última página de resultados, lo que genera un inconveniente, ya que la deserialización con `PlaylistData` requiere que dicho campo siempre esté presente. Este problema se soluciona utilizando el método `asegurarCampoNext` (ver Figura 12), que añade el campo a la cadena JSON de forma manual cuando es necesario, garantizando así que la deserialización se realice sin errores.

```
private String asegurarCampoNext(String jsonResponse) {
    if (!jsonResponse.contains("\"next\"")) {
        return jsonResponse.replace("}", ", \"next\": \"\" }");
    }
    return jsonResponse;
}
```

Figura 12: Código del método `asegurarCampoNext`

Finalmente, si ocurre algún otro error durante la solicitud, como un fallo del servidor de Deezer, o durante la conversión, por ejemplo, debido a una respuesta incompleta o irregular, la excepción se captura y el método devuelve un valor nulo, lo que permite manejar la situación sin interrumpir el flujo de la aplicación.

Posteriormente, `MusicaService` se encarga de transformar este objeto en una lista de reproducción adaptada a la aplicación, conservando únicamente los campos relevantes.

4.6. Interfaz gráfica

Para el desarrollo del frontend se ha empleado Angular, un framework moderno basado en TypeScript que facilita la construcción de aplicaciones web modulares y mantenibles. En concreto, se ha utilizado la versión 17.3, que introduce de manera estable la posibilidad de utilizar componentes standalone, que funcionan de manera independiente, sin necesidad de ser declarados dentro de un módulo, lo que simplifica la estructura de la aplicación y reduce el código repetitivo [5].

Se ha seguido el patrón SPA (Single Page Application), en el que la navegación entre diferentes vistas se realiza dinámicamente sin recargar páginas completas, mejorando la experiencia de usuario al reducir los tiempos de carga y permitiendo una gestión más eficiente del estado de la aplicación.

Para implementar este patrón se ha utilizado Angular Router, el sistema de enrutamiento propio de Angular que permite asociar rutas de la aplicación con los diferentes componentes de la interfaz que se deberán ir mostrando a medida que se navega desde el inicio. Todos estos componentes se descargan al acceder por primera vez a la página, y simplemente se hacen visibles cuando se accede a una ruta determinada, evitando tener que llamar al servidor siempre que se cambia de ruta. De esta manera, al cambiar de sección, se renderiza únicamente el componente correspondiente sin necesidad de recargar toda la página. La configuración de estas rutas se encuentra definida en el archivo *app.routes.ts*, que se muestra en la Figura 13.

```
// Nota: No se muestran las sentencias de importación de Routes
↪  ni del resto de componentes, pero están presentes en el
↪  archivo original.

export const routes: Routes = [
    { path: '', redirectTo: '/inicio', pathMatch: 'full' },
    { path: 'inicio', component: InicioComponent },
    { path: 'menu-principal', component: MenuPrincipalComponent
    ↪  },
    { path: 'buscar-partida', component: BuscarPartidaComponent
    ↪  },
    { path: 'unirse-partida-privada', component:
    ↪  UnirsePartidaPrivadaComponent },
    { path: 'crear-partida', component: CrearPartidaComponent },
    { path: 'partida/:id', component: PartidaComponent },
];
```

Figura 13: Código para la configuración de rutas

En la configuración de rutas, cada entrada define un path y el componente que debe cargarse al acceder a dicha ruta. La primera ruta utiliza `redirectTo` para redirigir automáticamente a /inicio cuando el usuario accede a la raíz de la aplicación, garantizando así que se muestre la pantalla de inicio por defecto. Las rutas restantes funcionan de manera directa: /inicio carga el componente InicioComponent, /menu-principal carga MenuPrincipalComponent y así sucesivamente.

Por último, la ruta /partida/:id incluye un parámetro dinámico (:id), que se obtiene de la URL y permite al componente PartidaComponent recuperar el identificador de la partida correspondiente.

La página principal de la aplicación corresponde al componente AppComponent (ver Figura 14), que actúa como contenedor de la interfaz. Este componente define la posición de los elementos que permanecerán fijos en la página y el área en la que se cargarán dinámicamente los componentes gestionados por el router.

```
<div class="layout">
    <h1>{{title}}</h1>
    <router-outlet class="w-100"></router-outlet>
    <app-error-box></app-error-box>
</div>
```

Figura 14: Código del componente AppComponent

En la implementación actual, incluye la cabecera de la aplicación (`<h1>title</h1>`) y un contenedor de errores (`<app-error-box>`), correspondiente a ErrorBoxComponent, que permanece en todas las vistas pero que solo se muestra cuando se produce algún error. Entre estos elementos, se sitúa el `<router-outlet>`, que indica dónde se insertarán los componentes correspondientes a las distintas rutas de la aplicación. De este modo, la estructura permite que ciertos elementos de la interfaz permanezcan siempre presentes mientras el contenido principal se actualiza dinámicamente según la navegación del usuario.

4.6.1. Servicios Angular

Para gestionar la comunicación entre el cliente y el servidor se han desarrollado varios servicios Angular, encargados de realizar las llamadas al backend mediante HttpClient y RxJS. HttpClient es el módulo de Angular encargado de enviar solicitudes HTTP y recibir respuestas de forma asíncrona [1]. Por su parte, RxJS (Reactive Extensions for JavaScript) permite gestionar flujos de datos, incluyendo secuencias de eventos o respuestas HTTP, mediante observables y otros mecanismos, a los que los componentes pueden suscribirse para reaccionar a cambios y recibir actualizaciones [9].

Se ha creado un servicio por cada controlador del backend, de modo que cada servicio encapsula la lógica de interacción con su correspondiente conjunto de endpoints. Esto ha dado lugar a los siguientes servicios:

- usuario.service.ts, correspondiente a UsuarioController.java
- partida.service.ts, correspondiente a PartidaController.java
- musica.service.ts, correspondiente a MusicaController.java

Un ejemplo representativo de los servicios implementados en el frontend es UsuarioService, cuya definición y método principal se muestran en las Figuras 15 y 16.

```
// Nota: No se muestran las sentencias de importación ni el
↪  decorador @Injectable para definir el servicio a nivel de
↪  aplicación, pero están presentes en el archivo original

export class UsuarioService {

  usuariosURL = `${environment.apiUrl}/usuarios`;

  usuario: Usuario | null = null;

  constructor(private http: HttpClient, private errorBoxService:
  ↪  ErrorBoxService) { }

  // Método para crear usuario
}
```

Figura 15: Código de la definición del servicio UsuarioService

En UsuarioService se observa la propiedad usuariosURL, que corresponde a la URL base del controlador del backend, y la propiedad usuario, que se utiliza para almacenar la información del usuario creado, facilitando su uso posterior dentro de la aplicación. El servicio cuenta con un constructor que recibe como dependencias HttpClient y ErrorBoxService, un servicio auxiliar encargado de mostrar errores generales en el componente ErrorBoxComponent mencionado anteriormente.

El método principal es crearUsuario, que se corresponde con el endpoint /crearUsuario del controlador del backend. Este método realiza la llamada HTTP y devuelve un observable con el identificador del usuario creado. Una vez obtenida la respuesta, utiliza el identificador devuelto y el nombre recibido para actualizar la propiedad `usuario`. La propiedad `usuario` está tipada según el modelo `Usuario`. Un modelo es una representación estructurada de los datos que se utilizan en la aplicación, definiendo su forma y los tipos de cada campo. En este caso, `Usuario` incluye campos como `id` y `nombre`, lo que permite que la información recibida del backend se maneje de manera consistente y segura en los componentes y servicios del frontend, evitando errores derivados de estructuras de datos no definidas o inconsistentes.

Centrándonos nuevamente en el método, este incluye registros en consola para facilitar el seguimiento del flujo y un registro de errores en caso de fallo. Además, gestiona de forma específica los errores 500 (error interno del servidor) y 0 (problemas de conexión), enviando los mensajes correspondientes a `ErrorBoxService`. Otros errores, más específicos del endpoint, se propagan para que sean gestionados por el componente que llama a este método.

En resumen, `UsuarioService` ejemplifica la función de los servicios en el frontend, que es encapsular la lógica necesaria para comunicarse con el backend, asegurar que los datos se manipulen de manera tipada a través de modelos y ofrecer un enfoque centralizado y bien organizado para la gestión de errores.

```
crearUsuario(nombreUsuario: string): Observable<number> {
  console.log("Creando usuario...");

  const url = `${this.usuariosURL}/${nombreUsuario}`;

  return this.http.post<number>(url, null).pipe(
    tap((usuarioID) => {
      this.usuario = {id: usuarioID, nombre: nombreUsuario};
      console.log(`Usuario creado`);
    }),
    catchError((error) => {
      console.error('Error al crear usuario', error);

      // Manejo del error 500
      if (error.status === 500) {
        this.errorBoxService.agregarError('Error interno del
        ↪ servidor. Intenta más tarde.');
      }
      // Manejo de errores cuando no hay respuesta del servidor
      else if (error.status === 0) {
        this.errorBoxService.agregarError('No se pudo conectar al
        ↪ servidor. Verifica tu conexión a internet o intenta
        ↪ más tarde.');
      }
      return throwError(() => new Error(error.message || 'Error
      ↪ desconocido'));
    })
  );
}
```

Figura 16: Código del método crearUsuario del servicio UsuarioService

4.6.2. Componentes

En esta sección se describen los componentes que conforman la interfaz gráfica. Los componentes de la aplicación se dividen principalmente en dos categorías: menús de navegación y formularios de entrada de datos.

Por un lado, los menús se caracterizan por presentar al usuario un conjunto reducido de opciones, cada una acompañada de una breve descripción y un elemento interactivo (por ejemplo, un botón) que permite seleccionarlas. Como ejemplo, en la Figura 17 se muestra el menú principal, que ofrece tres alternativas: buscar partida, unirse a partida privada y crear partida, cada una con su correspondiente información explicativa y el botón asociado.

A continuación, en la Figura 18 se presenta un fragmento del código del componente que implementa el menú principal, concretamente la sección correspondiente a la opción de búsqueda de partidas. Las demás presentan una estructura análoga.

Figura 17: Menú principal

```
<div class="menu-card">
    <h3 class="menu-title">Buscar partidas</h3>
    <p class="menu-description">Encuentra partidas públicas
    ↪  disponibles.</p>
    <button class="btn btn-primary"
    ↪  routerLink="/buscar-partida">Buscar Partidas</button>
</div>
```

Figura 18: Fragmento de código del componente MenuPrincipal

Como se puede observar, la opción se implementa como una tarjeta (menu-card) que contiene un título que identifica la acción principal, un párrafo descriptivo que ofrece al usuario una breve explicación de la funcionalidad, y un botón de navegación que, mediante la directiva `routerLink`, permite redirigir a la ruta correspondiente dentro de la aplicación cuando se activa el botón correspondiente.

Por otro lado, la aplicación incorpora formularios destinados a la introducción de datos por parte del usuario, permitiendo, por ejemplo, registrar un nuevo usuario o crear una partida. Estos formularios validan y envían los datos al servicio correspondiente, que se encargará de comunicarse con el backend. En la Figura 19 se muestra, como ejemplo, el formulario para configurar y crear una partida. El formulario de configuración de partida permite al usuario definir los parámetros principales de una partida.

La selección de la playlist se realiza mediante un menú desplegable que incorpora un campo de búsqueda. Al desplegar la opción, el sistema solicita al usuario introducir el nombre de la lista de reproducción en un cuadro de texto y, posteriormente, ejecutar la búsqueda a través del botón correspondiente. Una vez obtenidos los resultados, el usuario puede seleccionar una de las listas mostradas o realizar nuevas búsquedas con diferentes términos. En caso de no encontrarse coincidencias, el sistema muestra un mensaje informativo indicando la ausencia de resultados y sugiriendo modificar el criterio de búsqueda. Tras seleccionar una, el nombre de esta reemplaza automáticamente el texto inicial 'Seleccione una playlist' en el menú desplegable, confirmando la elección realizada.

A continuación, se definen tanto el número de rondas como el número máximo de usuarios a través de campos de entrada numéricos. Asimismo, el sistema incorpora

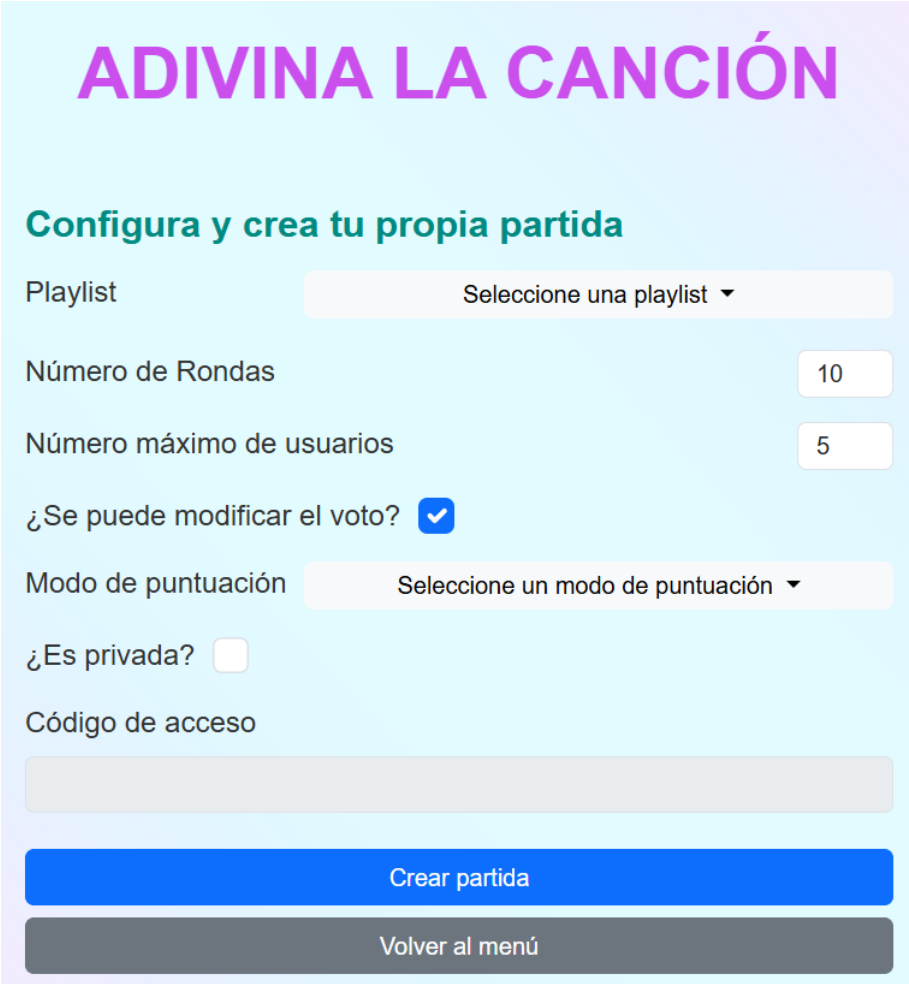

Figura 19: Formulario para la creación de una partida

una casilla de verificación que permite habilitar o deshabilitar la opción de modificar el voto durante las rondas de la partida. El modo de puntuación se define mediante un menú desplegable. Se utiliza esta opción en lugar de una casilla de verificación o un control similar, ya que, aunque actualmente solo existan dos modalidades, esta decisión favorece la escalabilidad al permitir añadir nuevos modos en el futuro sin necesidad de modificar la interfaz del formulario.

El usuario también tiene la opción de marcar la partida como privada mediante otra casilla de verificación. Al activarlo, se habilita automáticamente un campo de texto en el que se puede introducir un código de acceso, el cual será requerido por otros participantes para unirse a esta partida privada. Por último, el formulario incluye dos botones de acción: uno para crear la partida con la configuración establecida y otro para regresar al menú principal sin guardar los cambios realizados.

Para implementar el formulario, en la plantilla HTML se emplea la etiqueta `<form>` para definir el contenedor de los campos de entrada y la directiva `[formGroup]` para asociar el formulario al grupo de controles definido en la clase TypeScript. La directiva `(ngSubmit)` se encarga de interceptar el evento de envío del formulario, ya sea al presionar la tecla *Enter* en un campo de entrada o al hacer clic en el botón de confirmación. Su función es delegar el tratamiento de los datos al método definido

en la clase TypeScript, garantizando así la separación entre la captura del evento y la lógica de procesamiento.

Cada campo del formulario se organiza en un contenedor, cuya complejidad depende del dato a capturar. Algunos, como el número de rondas, se limitan a una etiqueta y un campo numérico con restricciones de validación. Otros, como la lista de reproducción, exige una construcción más elaborada, que combina un menú desplegable, un cuadro de texto, un botón de búsqueda y un área de resultados. En cualquier caso, lo esencial es que cada campo en la vista está enlazado con su control correspondiente en el formulario reactivo, de modo que tanto los datos introducidos como las validaciones se gestionan de forma coherente.

En el archivo TypeScript se utiliza FormBuilder, una herramienta de Angular que simplifica la creación y configuración de formularios reactivos. Internamente, FormBuilder genera los objetos FormGroup y FormControl que representan, respectivamente, el conjunto de campos del formulario y cada campo individual con sus reglas de validación. Gracias a FormBuilder, los campos y sus validaciones se definen de manera concisa. Por ejemplo, la lista de reproducción es obligatoria, el número de rondas tiene un valor por defecto de 10 y un rango entre 1 y 20, y el código de acceso utiliza un validador personalizado que lo hace obligatorio únicamente cuando la partida es privada. Esta implementación garantiza que los datos se validen de manera consistente antes de ser procesados por el servicio correspondiente y enviados al servidor. Además, permite gestionar de forma reactiva el estado del formulario dentro del componente y mostrar mensajes de error en la interfaz. Por ejemplo, si se intenta enviar el formulario si haber seleccionado una lista de reproducción, se muestra un aviso indicando que el campo es obligatorio, evitando enviar información incorrecta al servidor.

Entre los aspectos visuales generales destacan la tipografía clara y legible, el alto contraste entre texto y fondo, y una distribución uniforme de los elementos en la interfaz. La navegación se ha tratado de hacer intuitiva gracias a menús claros y botones descriptivos, y la coherencia se mantiene mediante patrones repetidos en botones, campos de entrada, iconos y colores. Los formularios incorporan validación interactiva que proporciona retroalimentación inmediata y facilita la corrección de errores, y además se minimiza la carga cognitiva mostrando únicamente la información necesaria en cada pantalla.

En cuanto a los elementos específicos de la aplicación en materia de usabilidad, se han incorporado decisiones de diseño poco habituales en interfaces generales, pero que resultan especialmente pertinentes en este caso. El ejemplo más representativo es la visualización de aciertos y errores: en lugar de emplear los colores verde y rojo estándar, se utilizan tonalidades alternativas de ambos que resultan más fácilmente distinguibles por personas con daltonismo, garantizando así una mejor accesibilidad y una experiencia más inclusiva [3].

4.7. Lógica del juego

La lógica del juego constituye el núcleo de la aplicación y difiere de la típica estructura CRUD de Spring. En esta sección se describe cómo el servidor gestiona las partidas durante el juego, incluyendo el control de eventos del juego, la sin-

cronización entre jugadores y la ejecución de tareas programadas (scheduled tasks) necesarias para mantener la dinámica del juego, tales como el cambio automático de canciones, el tiempo límite de cada ronda y la actualización de puntuaciones. Estas funcionalidades requieren un enfoque distinto al de las operaciones habituales de persistencia y servicios, ya que implican coordinación temporal y reacción a eventos.

El servidor debe gestionar múltiples partidas de forma concurrente, ya que varias partidas pueden estar activas al mismo tiempo con diferentes jugadores. Para ello, se utilizan estructuras de datos seguras frente a la concurrencia, como ConcurrentHashMap, que permiten almacenar y actualizar información compartida sin riesgo de inconsistencias. Al iniciarse una partida, se utiliza esta estructuras de datos para almacenar el identificador de la partida junto con el número de ronda actual, asegurando así que el progreso de cada juego se controle de manera consistente. Además de mantener el estado de las rondas, el servidor controla la dinámica del juego mediante dos tareas periódicas y un gestor de eventos.

La primera tarea periódica se encarga de establecer el intervalo de tiempo de la ronda, registrando su inicio y su fin: el inicio corresponde al instante actual y el fin se calcula sumando al instante actual 10 segundos, que es el tiempo dado a los jugadores para responder. A continuación, la tarea envía a los participantes las cuatro canciones de la ronda, el audio de la canción correcta y el instante de fin de la ronda, que puede utilizarse como temporizador en el cliente. El gestor de eventos procesa las respuestas que llegan desde los clientes. Para cada una, comprueba si es correcta, calcula la puntuación correspondiente y persiste el resultado en la partida para que esté disponible cuando la segunda tarea publique los resultados. La segunda tarea periódica se encarga de enviar a los jugadores la canción correcta de la ronda y la puntuación obtenida por cada participante, actualizando así la información del juego antes de iniciar la siguiente ronda.

La planificación de tareas se implementa mediante ThreadPoolTaskScheduler, una clase de Spring que actúa como un planificador de tareas multihilo, gestionando un pool de hilos para ejecutar tareas en momentos o intervalos determinados.

La implementación de la planificación de las dos tareas es muy similar, por lo que la Figura 20 muestra el código para la primera a modo de ejemplo.

```
Instant instanteDeInicio =
 ↪  Instant.now().plusSeconds(TIEMPO_ESPERA_CONEXION_WEBSOCKET);
future = taskScheduler.scheduleAtFixedRate(() ->
 ↪  enviarSiguienteCancion(partida), instanteDeInicio,
TIEMPO_ENTRE_RONDAS_DURATION);
hashMapFuture1.put(partida.getId(), future);
```

Figura 20: Código para la planificación de tareas con ThreadPoolTaskScheduler

Como se puede observar, primero se calcula el instante inicial de la tarea como el instante actual más un tiempo de espera, esto permite que los jugadores establezcan la conexión antes de que comience la ejecución y evita posibles errores. A continuación, con el método scheduleAtFixedRate se programa la ejecución periódica,

indicándole tanto el instante de inicio como el período. Finalmente, el Future devuelto se guarda en un mapa, asociado al identificador de la partida, lo que permite mantener un control independiente sobre las tareas de cada partida. Este Future resulta especialmente útil para cancelar la tarea una vez que se han completado todas las rondas de la partida.

Por otro lado, la implementación del gestor de eventos consiste en que el servicio de la partida implemente la interfaz ApplicationListener<RespuestaEvent>. De este modo, cuando se publica un evento RespuestaEvent se invoca automáticamente el método onApplicationEvent de este servicio, que se encarga de procesar la respuesta del usuario. El evento específico RespuestaEvent, que hereda de ApplicationEvent y encapsula la información de la respuesta enviada por un jugador. Cuando un cliente responde durante una ronda, el servidor crea y publica una instancia de este evento, que pasa automáticamente al subsistema de eventos de Spring, que será escuchado y procesado por el gestor de eventos. Cabe destacar que el uso de eventos evita dependencias directas y acoplamientos circulares. Gracias a esta técnica, el componente que recibe la respuesta del usuario publica el evento sin conocer quién lo procesará, y PartidaService lo escucha y lo gestiona sin mantener referencia al publicador.

4.8. Comunicación bidireccional

Para implementar la dinámica del juego era necesario contar con un mecanismo de comunicación bidireccional entre el servidor y los clientes. No bastaba con que el cliente solicitara datos al servidor, sino que el propio servidor debía poder enviar información de manera proactiva: al inicio de cada ronda para mandar las cuatro canciones, el audio de la canción correcta y el instante de fin, y al finalizar la ronda para enviar la canción correcta y las puntuaciones de la ronda. Para ello se valoraron dos opciones: *webhooks* y *websockets*.

4.8.1. Webhook

En un primer momento se valoró el uso de webhook, un patrón basado en el protocolo HTTP mediante el cual un servidor notifica a otro sistema cuando ocurre un evento. La operación de un webhook se divide en dos fases: configuración y funcionamiento. En la fase de configuración, la aplicación receptora proporciona una URL que la aplicación emisora registra como destino de las notificaciones. En la fase de funcionamiento, cuando ocurre un evento en la aplicación emisora, ésta envía automáticamente una solicitud HTTP POST con los datos del evento a la URL registrada, y la aplicación receptora procesa la información y puede ejecutar alguna acción.

Sin embargo, webhook no resultaba adecuado para resolver la problemática. En primer lugar, requiere exponer un endpoint en el sistema que recibe la información, lo cual no es viable en este caso porque el receptor es un cliente frontend. Además, al basarse en peticiones HTTP puntuales, no permite mantener un canal persistente de comunicación, lo que dificulta el envío continuo de mensajes. Por estas razones, aunque webhook es muy útil para notificaciones asíncronas entre servidores, no puede garantizar la inmediatez ni la bidireccionalidad requeridas por la aplicación.

4.8.2. WebSocket

Dado que webhook no resultaba adecuado, se descartó y se exploraron otras alternativas, entre las cuales se identificó WebSocket como la opción más adecuada. WebSocket es un protocolo que permite establecer un canal de comunicación persistente y bidireccional entre un servidor y un cliente sobre una única conexión TCP. A diferencia de las peticiones HTTP tradicionales, la conexión permanece abierta, lo que permite que el servidor y el cliente intercambien mensajes en tiempo real sin necesidad de abrir nuevas conexiones para cada interacción. Esta tecnología se adapta perfectamente a las necesidades de la aplicación, ya que permite al servidor enviar de manera proactiva los eventos de cada ronda.

Para implementar la comunicación mediante WebSocket es necesario configurar tanto el servidor como el cliente. En el servidor, se debe añadir la dependencia de WebSocket al proyecto Spring y crear una clase de configuración que sobrescriba el método correspondiente para añadir el gestor de eventos y los interceptores, y definir los orígenes permitidos. En el cliente, se crea un servicio dedicado para gestionar la comunicación WebSocket, que incluye métodos para abrir la conexión, procesar los mensajes entrantes y enviar información al servidor.

En el servidor, lo primero es añadir la dependencia de WebSocket al proyecto para que Spring pueda soportar este protocolo, tener los componentes necesarios para crear gestores de eventos, interceptores y gestionar sesiones. Luego, se crea una clase de configuración anotada con @EnableWebSocket para habilitar la infraestructura de WebSocket en la aplicación. La clase implementa WebSocketConfigurer para definir de forma programática los endpoints WebSocket. En concreto, hay que sobrescribir el método registerWebSocketHandlers para añadir el gestor de eventos y los interceptores, y definir los orígenes permitidos. La definición de los orígenes permitidos se hace por motivos de seguridad, para evitar que orígenes no autorizados puedan establecer conexiones. Consiste en especificar la URL del frontend que podrá conectarse. Durante la fase de desarrollo, se utiliza la dirección del servidor que sirve la aplicación frontend en el entorno local, mientras que en producción se debe indicar la dirección del servidor donde está desplegado.

Los interceptores permiten procesar y validar información durante la fase inicial de la conexión WebSocket. En este caso, el interceptor se encarga de extraer el identificador de la partida y del usuario, verificar que la partida y el usuario existen y que el usuario pertenece a la partida y, si todo es correcto, añadir estos datos como atributos de la sesión para que puedan ser usados posteriormente, durante la comunicación.

Para configurar el interceptor, se crea una clase que implemente la interfaz HandshakeInterceptor y se registra como parámetro en addInterceptors dentro del método registerWebSocketHandlers. Su comportamiento principal se define sobrescribiendo el método beforeHandshake, para realizar en él la extracción de los parámetros de la URI, su validación y el almacenamiento como atributos de la sesión.

Por último, el gestor de eventos se encarga de gestionar la comunicación durante la conexión WebSocket, recibiendo los mensajes enviados por los clientes y enviando respuestas o notificaciones de manera proactiva. Además, mantiene el estado de la sesión y de la conexión, lo que permite una interacción en tiempo real fluida durante

el desarrollo de la partida. Para configurarlo, se debe crear una clase que implemente la interfaz WebSocketHandler, en este caso, heredando de TextWebSocketHandler, que ya implementa WebSocketHandler. En la clase se sobrescriben métodos como afterConnectionEstablished, handleTextMessage y afterConnectionClosed. Además, se define un método específico para enviar objetos a los clientes conectados. El método afterConnectionEstablished se ejecuta después de que la negociación haya concluido con éxito y la conexión WebSocket esté abierta y lista para usarse. Se sobrescribe para que, cada vez que se establece una conexión, almacene la sesión en un mapa asociándola a la partida correspondiente, de modo que posteriormente pueda obtenerse fácilmente la lista de sesiones de esa partida.

Por su parte, el método afterConnectionClosed se invoca automáticamente cuando la conexión WebSocket se cierra. Este método obtiene la lista de sesiones asociadas a la partida, elimina la sesión cerrada y, si la lista queda vacía, borra también la entrada de la partida del mapa para no dejar registros sin sesiones. El método handleTextMessage se invoca cuando llega un nuevo mensaje de texto por WebSocket. Primero deserializa el JSON recibido y comprueba que tenga el formato correcto. Si corresponde a una canción, extrae de la sesión el identificador de la partida y del usuario, obtiene el instante actual, crea un RespuestaEvent con toda esa información y lo publica para que sea procesado por el servicio de partida.

Finalmente, el método enviarTextMessage se encarga de la comunicación proactiva del servidor hacia los clientes, enviando mensajes de texto a todas las sesiones asociadas a una partida. Además, se complementa con métodos auxiliares que serializan los objetos para poder enviarlos correctamente.

Por otro lado, en el cliente se crea un servicio dedicado para centralizar toda la comunicación WebSocket y gestionarla de forma más sencilla. Los métodos principales de este servicio son connect, handleMessage y enviarMensaje. El método connect abre la conexión y se suscribe a los mensajes entrantes. La URL utilizada para la conexión es clave, ya que incluye los identificadores de la partida y del usuario, lo que permite al servidor asociar correctamente cada conexión a la partida y al usuario correspondiente. Por su parte, handleMessage se encarga de procesar los mensajes recibidos del servidor a través del WebSocket, deserializarlos si es necesario y emitir nuevos valores al flujo de datos del observable, de modo que todos los componentes que se hayan suscrito previamente reciban automáticamente la información. Por último, el método enviarMensaje se utiliza para enviar información desde el cliente al servidor mediante la conexión WebSocket, asegurándose de que la conexión esté abierta y serializando los objetos cuando sea necesario antes de enviarlos.

5. Pruebas y despliegue

5.1. Pruebas

En este apartado se describen las estrategias y procedimientos seguidos para garantizar el correcto funcionamiento de la aplicación. El objetivo general de las pruebas fue verificar la funcionalidad de la aplicación en sus tres áreas principales: el subsistema CRUD, la lógica del juego y la comunicación mediante WebSockets.

De forma concreta, se pretendió:

- Comprobar que las operaciones CRUD funcionaban correctamente y mantenían la integridad de los datos entre cliente y servidor.
- Verificar que la lógica del juego se ejecuta según lo especificado (inicio de rondas, temporización, cálculo y persistencia de puntuaciones).
- Asegurar que la comunicación WebSocket permite el intercambio proactivo de mensajes de servidor a cliente y la recepción de respuestas cliente a servidor, sin pérdida o inconsistencia de información.

No se persiguieron objetivos de rendimiento o carga (medición de latencia o pruebas de estrés) y las pruebas se centraron en la corrección funcional y en la consistencia de los datos.

5.1.1. Pruebas unitarias

Las pruebas unitarias son un tipo fundamental de pruebas de software que se centran en evaluar unidades individuales de código, como clases o métodos, de forma aislada. Estas pruebas verifican que estas unidades funcionen correctamente en términos de lógica y comportamiento esperado.

Para garantizar la correcta ejecución del software, se llevaron a cabo pruebas unitarias tanto en el servidor como en el cliente. Aunque no se siguió un proceso estandarizado ni se utilizó automatización, estas pruebas resultaron fundamentales para examinar las unidades de código más pequeñas y críticas del sistema.

Al cubrir escenarios normales y casos límite, las pruebas unitarias permitieron detectar y corregir errores en fases tempranas del desarrollo, evitando que los defectos se propagaran.

5.1.2. Pruebas de integración

Las pruebas de integración se centran en validar la interacción y la cooperación entre diferentes componentes, módulos o sistemas dentro de una solución mayor. Su propósito es detectar fallos que aparecen sólo cuando las partes se combinan, por lo que son esenciales para garantizar el correcto funcionamiento global de la aplicación.

En este proyecto no se siguió un plan formalizado ni se automatizaron las pruebas de integración; en su lugar se realizaron pruebas ad hoc y pruebas incrementales a medida que avanzaba el desarrollo. Este enfoque, aunque menos riguroso, resultó adecuado para detectar problemas tempranos y contribuyó a asegurar la coherencia y la correcta coordinación entre los distintos componentes del sistema.

5.1.3. Pruebas de aceptación

Las pruebas de aceptación están orientadas a verificar si una aplicación cumple con los requisitos y expectativas del cliente o usuario final. El objetivo principal de estas pruebas es asegurarse de que el software funcione correctamente en escenarios del mundo real y que cumpla con los criterios de aceptación previamente establecidos.

En este proyecto, los criterios de aceptación se definieron directamente en las historias de usuario. Cada historia de usuario incluye los pasos y condiciones que deben cumplirse para considerar la funcionalidad como correcta.

Estas pruebas se llevaron a cabo en el cliente, aprovechando las herramientas de testing de Angular, como ComponentFixture y TestBed, así como mocks para el enrutador. Fueron pruebas planificadas y sistemáticas, aunque no automatizadas completamente.

Por ejemplo, para la historia de usuario US005, titulada *Identificarse*, cuya descripción es: 'Como usuario, quiero poder identificarme introduciendo un nombre, para que el sistema me reconozca durante la partida', se definió una prueba de éxito y junto con varias orientadas a manejar posibles errores. A continuación, se muestra la correspondiente al caso de éxito:

1. El usuario inicia la aplicación.
2. El sistema muestra el formulario de identificación.
3. El usuario completa el formulario con su nombre.
4. El usuario pulsa 'Continuar' o presiona Enter.
5. Se verifica que el usuario haya sido registrado correctamente en el sistema.

La implementación de esta prueba en Angular se realizó tal como se muestra en la Figura 21.

```
it('Prueba 1: Éxito - Identificar usuario correctamente', () => {
  component.formularioUsuario.setValue({ nombreUsuario: 'Juan'
  ↪  });

  component.onSubmit();

  expect(usuarioServiceMock.crearUsuario)
    .toHaveBeenCalledWith('Juan');

  expect(routerMock.navigate)
    .toHaveBeenCalledWith(['/menu-principal']);
});
```

Figura 21: Código de la prueba de éxito para la historia de usuario US005

En esta prueba se refleja el flujo de la historia de usuario: los pasos 1 y 2 se simulan mediante la inicialización del componente y la visualización del formulario; el paso 3 se realiza introduciendo el nombre 'Juan' en el formulario; el paso 4 se simula invocando el método onSubmit; finalmente, el paso 5 se comprueba mediante las expectativas de que el servicio de usuario haya sido llamado con el nombre correcto y que la navegación haya redirigido al menú principal.

Entre las pruebas de error, se incluyó una prueba destinada a comprobar el comportamiento del sistema cuando el usuario deja campos vacíos:

1. El usuario inicia la aplicación.
2. El sistema muestra el formulario de identificación.

3. El usuario deja uno o más campos en blanco.
4. El usuario intenta continuar.
5. Se verifica que el sistema muestre correctamente los mensajes de error y que el usuario no sea creado.

La implementación en Angular puede verse en la Figura 22.

```
it('Prueba 2. Campos en blanco', () => {
  component.formularioUsuario.setValue({ nombreUsuario: '' });

  component.onSubmit();

  fixture.detectChanges();

  const nombreUsuarioVacio =
  ↪  fixture.nativeElement.querySelector('#nombreUsuarioVacio');
  expect(nombreUsuarioVacio).toBeTruthy();

  expect(usuarioServiceMock.crearUsuario).not.toHaveBeenCalled();
});
```

Figura 22: Código de la prueba de campos vacíos para la historia de usuario US005

En esta prueba se simula que el usuario no completa el formulario y presiona continuar. Luego, se comprueba que el mensaje de validación aparece en la vista y que el servicio de creación de usuario no se ha llamado, garantizando así que no se registren usuarios con datos incompletos.

Al cubrir los criterios de aceptación definidos en las historias de usuario, las pruebas de aceptación verificaron que la aplicación cumpliera con los requisitos funcionales esperados, garantizando que los usuarios pudieran interactuar correctamente con el sistema en escenarios reales.

5.2. Despliegue

El despliegue de software es el proceso mediante el cual una aplicación desarrollada pasa del entorno de desarrollo al entorno en el que será utilizada por los usuarios finales.

Incluye tareas como la configuración del entorno de ejecución, la instalación de dependencias necesarias, la preparación de variables y archivos de configuración, la publicación de los recursos de la aplicación y la verificación de su correcto funcionamiento en el entorno de producción. También puede implicar la implementación de medidas de seguridad, la gestión de la red y de los accesos, así como la monitorización inicial del servicio para garantizar su disponibilidad y estabilidad.

En términos generales, el despliegue de una aplicación puede realizarse en distintos niveles de complejidad, cada uno con implicaciones diferentes en cuanto a control, esfuerzo de configuración y mantenimiento.

La primera opción consiste en montar un servidor físico propio. En este escenario, se debe adquirir y mantener el hardware, instalar el sistema operativo, configurar los servicios de red, cortafuegos y seguridad, además de desplegar y gestionar la aplicación en sí. Aunque esta opción otorga el máximo control sobre todos los aspectos de la infraestructura, también implica un alto coste en tiempo, conocimientos técnicos y recursos de mantenimiento. Por ello, resulta poco práctica para proyectos de pequeña escala, como una aplicación desarrollada en el marco de un Trabajo Fin de Máster.

Una segunda alternativa consiste en contratar una máquina virtual en la nube mediante proveedores como Amazon Web Services, Microsoft Azure u otros servicios similares. En este caso, se elimina la necesidad de gestionar el hardware físico, pero todavía es necesario configurar el entorno de ejecución de la aplicación, así como implementar medidas de seguridad. Aunque este modelo reduce la carga operativa frente a un servidor físico, sigue requiriendo un nivel considerable de tiempo y de conocimientos técnicos.

Finalmente, existe la opción de utilizar un servicio de alojamiento. En este caso, solo es necesario subir el código o el artefacto de la aplicación, y la plataforma se encarga de todo lo demás. Aunque limita el control y la personalización de los recursos disponibles, para una aplicación sencilla como la desarrollada en este proyecto, es la opción más adecuada.

Para la aplicación desarrollada en este proyecto, se buscó un servicio de alojamiento que cumpliera dos condiciones principales: ser gratuito y permitir el despliegue del servidor, desarrollado con Spring, y del cliente, desarrollado con Angular. Dentro de esta segunda condición, era especialmente importante que la plataforma ofreciera soporte para WebSockets, ya que no todos los servicios de alojamiento gratuitos lo permiten y es fundamental para la aplicación. Tras evaluar varias alternativas, se eligió Render, ya que cumple con todos estos requisitos.

Render se integra con repositorios de código como GitHub, lo que permite automatizar el proceso de despliegue. Aunque es posible desplegar el servidor y el cliente desde el mismo repositorio, teniéndolos separados resulta más sencillo. Inicialmente, ambos estaban en un único repositorio, pero al llegar a la fase de despliegue se decidió separarlos por este motivo.

Por un lado, se debe desplegar el servidor y, por otro, el cliente. Cada componente requiere un enfoque distinto debido a sus características y a la manera en que se sirven y consumen los recursos.

5.2.1. Despliegue del servidor

El servidor se corresponde con la aplicación Spring, que se encarga de la lógica de negocio y la gestión de datos. Su despliegue implica exponer los endpoints que serán consumidos por el cliente. Antes de desplegar el servidor, se modificó la configuración para adaptarlo al nuevo entorno sin dejar de ser compatible con el entorno de desarrollo.

Para ello, se modificó el archivo de configuración para habilitar un puerto dinámico, utilizando la línea server.port=${PORT:8080}. Esto permite que la plataforma de alojamiento asigne automáticamente el puerto en el que Spring debe escuchar,

mientras que, si la variable de entorno no está definida, por ejemplo, en el entorno local, la aplicación sigue utilizando el puerto 8080 por defecto.

Además, se actualizaron los controladores y la configuración de WebSocket para que la aplicación pudiera aceptar conexiones no solo desde la dirección local, sino también desde la dirección del frontend desplegado, garantizando así la correcta comunicación entre cliente y servidor en el entorno de producción.

A continuación se procedió a la dockerización de la aplicación. Docker es una plataforma que permite empaquetar una aplicación junto con todas sus dependencias, configuraciones y entorno de ejecución en una unidad estandarizada llamada contenedor. Frente al despliegue directo, esta estrategia ofrece mayor portabilidad y simplicidad en la gestión, evitando problemas derivados de diferencias entre entornos y facilitando tanto la escalabilidad como el mantenimiento.

Para dockerizar la aplicación, es decir, encapsularla en un contenedor que garantiza que se ejecute de la misma forma en cualquier sistema, es necesario crear un archivo denominado Dockerfile. En él se establecen las instrucciones necesarias para compilar el proyecto y generar el artefacto ejecutable, construir una imagen para la ejecución y configurar el arranque del servidor dentro del contenedor.

El siguiente paso fue crear el Web Service en Render y asociarlo al repositorio de GitHub. Se seleccionó la rama principal como fuente del código a desplegar, de modo que la plataforma tomara siempre la última versión estable del proyecto. Además, se configuró el servicio para que Render compilara y ejecutara automáticamente la aplicación a partir del código publicado, desencadenando un nuevo despliegue en cada push sobre dicha rama. Finalmente, se realizó la configuración del servicio en la plataforma, lo que implicó definir las variables de entorno necesarias y especificar la ruta del Dockerfile.

Tras el despliegue inicial, se realizaron pruebas para verificar que los endpoints funcionaban correctamente y que la aplicación podía gestionar conexiones WebSocket de manera estable. Durante estas pruebas se comprobó que, a diferencia del entorno local donde las peticiones a la API de Deezer se resolvían sin problemas, en producción dichas solicitudes eran rechazadas sistemáticamente. La causa parecía estar en las políticas de restricción de Deezer, que bloquean el tráfico procedente de rangos de direcciones IP asociados a proveedores de alojamiento en la nube como Render, probablemente como medida de seguridad para evitar abusos o usos automatizados.

Ante esta situación, se valoraron distintas soluciones, siendo una de ellas recurrir al uso de proxies intermedios para redirigir las peticiones y evitar el bloqueo. Se realizaron varias pruebas con diferentes proxies: algunos directamente no funcionaban, y otros, al ser muy conocidos y ampliamente utilizados, también estaban bloqueados por Deezer. Finalmente, se encontró thingproxy, un proxy funcional que, al ser relativamente poco conocido, no figuraba en las listas de bloqueo y permitía que las solicitudes se procesaran correctamente. Gracias a este intermediario, las peticiones volvieron a funcionar de manera estable y la aplicación pudo operar en producción sin incidencias.

5.2.2. Despligue del cliente

El cliente es una aplicación Angular que se ejecuta en el navegador y actúa como interfaz web del usuario. Su despliegue consiste en servir el contenido estático y garantizar la comunicación con el servidor. Antes de iniciar el despliegue, se crearon y configuraron los archivos environment.ts y environment.prod.ts para separar las variables específicas de los entornos de desarrollo y producción, de modo que la aplicación pueda adaptarse automáticamente al entorno en el que se ejecute sin modificar el código fuente. Además, se actualizó el archivo angular.json para asegurar que cada compilación utilizara la configuración adecuada en función del entorno seleccionado. Finalmente, se ajustó el código para utilizar las variables definidas en esos archivos, garantizando que la aplicación tome los valores adecuados según se ejecute en desarrollo o en producción.

A diferencia del servidor, en este caso no se recurrió a la dockerización. Dado que Angular genera un conjunto de archivos estáticos tras el proceso de compilación, el despliegue se simplifica a servir dichos archivos en la plataforma de alojamiento, sin necesidad de encapsular la aplicación en un contenedor. Esto hace que el proceso sea más sencillo, rápido y directo.

Al no dockerizar la aplicación cliente, el despliegue en Render se reduce a dos configuraciones esenciales: el comando de compilación y el directorio de publicación. El comando de compilación es ejecutado por Render antes de cada despliegue para generar los activos estáticos de la aplicación. En nuestro caso se utiliza `ng build`, que compila el proyecto y genera los ficheros listos para servir en producción.

El directorio de publicación es la ruta relativa a la carpeta donde están los recursos estáticos generados por el comando de compilación. Este directorio es fundamental, ya que indica a Render dónde se encuentran los archivos que debe servir y desplegar. Para este proyecto se configura como dist/adivina_la_cancion_frontend/browser.

Finalmente, al ser esta página web una SPA es necesario añadir la siguiente regla: `Source: /* Destination: /index.html Action: Rewrite`. Esto provoca que cualquier petición a una ruta interna que no corresponda a un archivo estático real devuelva siempre el archivo index.html. De esta manera, el navegador mantiene la ruta solicitada y es el enrutador de Angular quien interpreta esa ruta y renderiza la vista adecuada. Por ejemplo, gracias a esta regla, si el navegador solicita /inicio, el servidor de Render responde con el contenido de index.html y Angular lee esa ruta y muestra directamente la vista asociada a la ruta /inicio.

Tras completar el despliegue, se realizaron pruebas en el nuevo entorno para comprobar que la aplicación cargaba adecuadamente en el navegador y que las peticiones al servidor se resolvían de manera estable.

6. Conclusiones y trabajos futuros

6.1. Conclusiones

El Trabajo Fin de Máster ha alcanzado sus objetivos principales: se ha puesto a disposición una aplicación web operativa que permite a los usuarios participar en un juego de adivinanza musical a partir de breves fragmentos de audio.

La solución obtenida cumple con los requisitos definidos en la fase de diseño y ofrece una experiencia de uso personalizable, intuitiva, fluida y técnicamente sólida.

Desde el punto de vista arquitectónico, la separación entre un backend implementado en Spring Boot y un frontend desarrollado en Angular ha facilitado el desarrollo, las pruebas y el despliegue, al mismo tiempo que ha mejorado la mantenibilidad y la escalabilidad potencial de la aplicación.

En cuanto a las pruebas, se realizaron múltiples verificaciones orientadas a comprobar la funcionalidad de la aplicación en sus principales áreas.

Respecto al despliegue, el backend se dockerizó y se publicó como un Web Service, mientras que el frontend se sirvió como un Static Site. Durante la puesta en producción se resolvieron las restricciones de comunicación con la API de la plataforma de streaming musical mediante un proxy intermedio, garantizando la operatividad de la aplicación.

En resumen, este proyecto me ha permitido no solo adquirir y consolidar conocimientos técnicos en el desarrollo de aplicaciones web, sino también abordar todas las fases del ciclo de vida de un proyecto software. Ha sido, en definitiva, una experiencia enriquecedora tanto a nivel académico como personal, alineada con mis intereses y con mi desarrollo profesional como ingeniero informático.

6.2. Trabajos futuros

A pesar de haber alcanzado satisfactoriamente los objetivos definidos en este Trabajo Fin de Máster, se han identificado diversas líneas de mejora que podrían enriquecer significativamente la solución desarrollada.

En primer lugar, es necesario solucionar la limitación actual en la comunicación con la plataforma de streaming musical, que bloquea el acceso desde rangos de IP asociados a proveedores en la nube como Render para prevenir usos indebidos. Una posible solución sería gestionar directamente con la plataforma la inclusión de la IP del backend en una whitelist, asegurando así la continuidad del servicio. Otra alternativa sería desplegar la aplicación en un servidor propio cuya IP no esté bloqueada, aunque esto no garantiza que funcione a largo plazo, ya que la IP podría ser eventualmente restringida en el futuro.

Adicionalmente, se contempla la persistencia del historial de partidas, junto con mecanismos de autenticación de usuarios. Esta ampliación mejoraría la experiencia de uso, al permitir guardar preferencias o configuraciones personalizadas, y abriría la puerta a funcionalidades más avanzadas, como estadísticas personalizadas o seguimiento del progreso individual.

Por último, se propone ampliar la experiencia de juego con nuevos modos y sistemas de puntuación alternativos. Estas mejoras fomentarían la rejugabilidad y ofrecerían a los usuarios un mayor grado de personalización en las partidas, consolidando el atractivo de la aplicación a medio y largo plazo.

Referencias

[1] Angular Team. Angular - HttpClient Guide. https://angular.io/guide/http. Accedido el 7 de septiembre de 2025.

[2] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2rd edition, 2014.

[3] David Nichols. Coloring for colorblindness. https://davidmathlogic.com/colorblind/#%23D81B60-%231E88E5-%23FFC107-%23004D40-%23D632A3. Accedido el 30 de mayo de 2025.

[4] Deezer Developers. Deezer API – Search Playlist Endpoint. https://developers.deezer.com/api/search/playlist. Accedido el 10 de marzo de 2025.

[5] Josh Dunn. Angular - Standalone Components vs Modules. https://lynkz.com.au/blog/2024-angular-standalone-vs-modules. Accedido el 3 de septiembre de 2024.

[6] Adam Freeman. *Essential TypeScript 5*. Manning, 3rd edition, 2023.

[7] ISO/IEC 25010. Iso/iec 25010: Modelo de calidad del producto. https://iso25000.com/index.php/normas-iso-25000/iso-25010. Accedido el 5 de septiembre de 2024.

[8] MizAlecto. Spotify API - 30 second preview URLs. https://www.reddit.com/r/spotifyapi/comments/1hayp7j/30_second_preview_urls/?rdt=60143. Accedido el 10 de marzo de 2025.

[9] RxJS Team. RxJS - Reactive Extensions for JavaScript. https://rxjs.dev/. Accedido el 7 de septiembre de 2025.

[10] Spotify Team. Changes to the Web API on 27 November 2024. https://www.reddit.com/r/spotifyapi/comments/1hayp7j/30_second_preview_urls/?rdt=60143. Accedido el 10 de marzo de 2025.

[11] Jeff Sutherland and J. J. Sutherland. *Scrum*. Ariel, 2018.

[12] Jonas Thelemann. spotify-web-api-java: Java wrapper for the spotify web api. https://github.com/spotify-web-api-java/spotify-web-api-java?tab=readme-ov-file#:~:text=Spotify%20has%20a,1%5D%20%5B2%5D. Accedido el 2 de noviembre de 2024.

[13] Yevhen Vasyliev. deezer-api: Java Wrapper for the Deezer API. https://github.com/yvasyliev/deezer-api. Accedido el 10 de marzo de 2025.

[14] Craig Walls. *Spring in Action*. Manning, 2022.