Readio: Book inspired playlist generation

Clara Luiza Alemar Ribas

June 29, 2025

Resumen– La música ha demostrado ser eficaz para influir en la percepción emocional y profundizar la inmersión narrativa, como se observa en el impacto de las bandas sonoras en las películas. En el contexto literario, la presencia de playlists inspiradas en libros en plataformas de música refleja el interés de los lectores por trasladar esa dimensión emocional a la lectura. Sin embargo, aunque existen herramientas para generar playlists, ninguna vincula la música directamente con el contenido de los libros de forma contextualizada. Este proyecto presenta una aplicación web que crea listas personalizadas a partir de libros, utilizando un Modelo de Lenguaje de Gran Escala para analizar el tono, los temas y las emociones de cada capítulo. Tras seleccionar un libro y un género musical, el sistema devuelve una playlist curada acorde al desarrollo narrativo. Al usar inteligencia artificial para integrar literatura y música, la herramienta enriquece la lectura de forma única e inmersiva.
Palabras clave– Inteligencia Artificial Generativa, Análisis literario, Modelos de Lenguaje de Gran Escala, Aplicación web

Abstract– Music has proven to be effective in influencing emotional perception and deepening narrative immersion, as seen in the impact of soundtracks in films. In the context of literature, the presence of book-inspired playlists on music streaming platforms reflects the interest of readers in applying this same emotional dimension to their reading experience. However, while there are various tools for playlist generation, none of them link music directly to the content of books in a contextualized way. This project presents a web application that creates personalized playlists based on books, using a Large Language Model to analyze each chapter's tone, themes, and emotions. After selecting a book and a musical genre, the system returns a curated playlist aligned with the narrative flow. By using AI to integrate literature and music, the tool enhances the reading experience in a unique and immersive manner.
Keywords– Generative AI, Literary analysis, Large Language Models, Web Application

1 Introduction

Music and text are both known to be powerful carriers of emotion, shaping experiences and conceptions. This project derives from the premise that, when combined, they can intensify emotional perception, making a narrative feel more immersive and engaging. Studies such as Hauck and Hecht [1] validate said thesis by demonstrating the influence of the emotional mood of the music on the perceived mood conveyed by the text.

The research consisted of an experiment to explore how background music affects the way people perceive written texts. During the procedure, participants read text extracts while listening to songs that were either happy or sad, aiming to determine whether the accompanying audio could transfer its emotional tone to the written passage.

• Contact e-mail: claraluiza.alemar@autonoma.cat
• Specialization: Software Engineering
• Work tutored by: Xim Cerdá Company (Computer Science)
• Academic year 2024/25

Their results showed that music had a clear influence on text perception, as passages were rated as happier when paired with happy music and sadder when accompanied by sad music. Apart from mood, the songs also affected how much readers enjoyed and engaged with the text. When the music and text shared the same emotional tone, participants felt more immersed in the story and rated it as higher quality.

As Hauck and Hecht concluded, listening to mood-congruent music fosters text immersion, making soundtracks a valuable enrichment for passionate readers. As someone who deeply values both literature and music, the idea for this project originated from the combination of these two personal interests. From that, the proposal emerged: to create an application capable of enhancing a reader's connection to a story through the use of emotionally fitting music, as the research demonstrates.

The way in which the project was conducted will be explained in the sections that follow, beginning with the methodology and planning, followed by the system design and development process, and concluding with the results, economic evaluation, business plan, and final reflections.

2 State of the art

There exists a significant number of playlist-generating tools available online, each offering different ways to discover and select music. However, none appears to be generally recognized specifically for book inspired playlist generation, and that is where this project comes in. To better understand the prospect, two existing tools were selected for comparison, Chosic and Playlistable, as they share some similarities with the project proposal but ultimately serve different purposes.

2.1 Chosic

Chosic is an online music recommendation platform that, among other functionalities, allows users to create Spotify playlists based on a song, artist, genre, mood, or existing playlist. The tool adopts an intelligent algorithm to recommend similar songs. While it does not provide detailed insights into how its recommendation system works, it mainly operates through musical similarity, considering factors such as danceability, energy, and instrumentalness for further personalization.

The platform features a simple and intuitive interface, where one simply input their preferences into a search box with an autocomplete feature. Upon search, the tool generates a playlist that can be refreshed to provide new recommendations. Additionally, Chosic enables preview of song snippets, direct listen on Spotify, access to a Youtube video for the song, as well as to save the playlist for future use.

2.2 Playlistable

Playlistable is an AI playlist generator that utilizes ChatGPT (GPT-3.5) to create personalized playlists based on the user's mood, occasion, or style. The tool analyzes the individual's listening history or a text prompt to generate a selection of up to 50 songs, which are automatically added to their Spotify account.

The platform's interface resembles a chat, where one can input prompts, including similar artists or songs, and receive tailored recommendations. In addition, the tool displays the user's top tracks, artists, and previously created playlists. Although the free version is limited to two playlists, additional playlists can be purchased for further exploration.

2.3 What sets the project apart

While both Chosic and Playlistable offer ways to generate custom playlists, and share some features with the proposed project, such as the autocomplete input field and the ability to save the playlist to a music library, neither tool focuses on analyzing book content or delving into its themes or mood. For instance, Playlistable can generate playlists inspired by books upon request, but it does not analyze the actual content of the text. This distinction is what sets the proposed application apart, as it aims to enhance the reading experience by aligning music with the narrative and emotional tone of a book.

3 Objectives

The project proposal consists of a web application that generates playlists based on a book specified by the user. Rather than branching from a great issue to be solved, the idea for the project originated from a combination of two personal passions: music and literature. Its main goal is to provide people who share those interests a platform that allows them, in addition to potentially discovering new music, to expand their reading experience. By suggesting songs that relate to the plot of the selected book, in terms of lyrics and mood, the application will allow the reader to immerse themselves more deeply in the story. In such manner, the user can enrich their reading experience as they read, or afterwards, if they wish to revisit the novel through the generated playlist.

With this goal in mind, the specific objectives of this project are:

- To analyze book chapters using a language model in order to generate music suggestions that reflect their content, mood, and themes.
- To store and present those songs as a cohesive playlist to the user, integrating the process with Spotify so users can preview and save the playlist.
- To build a functional web application using a limited dataset of books and predefined genres, leaving the system open for future scaling.

4 Methodology

The methodology adopted for the development of this project follows an **incremental paradigm**. The system was planned and built in successive phases, with each one delivering a functional part of the final product. This approach allowed the gradual construction of the application, from setting up the database and LLM integration, to backend logic and finally the frontend interface, where each stage complemented and expanded the previous one. While the overall goals and system concept were clearly defined early on, the structure allowed for flexibility and adjustments throughout the process, such as refining the playlist schema or adapting prompt formats. This adaptability was a key reason for adhering to the incremental approach, as it allowed integrating feedback from the tutor and refining intermediate results without needing to rework the entire system. As well as the ability to validate and test each part of the application as it was built, reducing the risk of late-stage issues.

Task management throughout the project was organized using the **Kanban method**,which suited the project by allowing tasks to be prioritized and adjusted as needed, without imposing rigid time structures. To represent work status, standard columns such as '*Backlog*', '*In Progress*', and '*Done*', along with additional '*On Hold*' and '*Canceled*' columns were implemented to improve clarity and control over task flow. This setup provided a clear overview of progress at all times and supported a more responsive and adaptive workflow.

4.1 Planning

The planning of the project was structured into six main phases, each corresponding to a stage in the system's development. These phases were grouped following the checkpoints defined by the TFG schedule, such as the reports and follow-up meetings. As mentioned Section 4, the phases were aligned with an incremental development approach, where each group of tasks contributed progressively to building the final product. The Table 1 summarizes the main project phases and their associated milestones.

TABLE 1: PROJECT PHASES AND KEY MILESTONES

Phase	Key Milestones
1 - Planning & Setup	Stack selection, environment setup, Initial Report
2 - Backend Development	DB schema, LLM integration, backend logic, Progress Report I
3 - Frontend Development	UI design, API integration, Spotify integration, Progress Report II
4 - System Refinement	Adjustments, issue fixing, data expansion
5 - Final Documentation	Final document draft and revisions
6 - Presentation & Delivery	Presentation prep and final submission

A detailed Gantt chart of the original planning is included in Appendix A.1. Although the overall structure remained consistent, some minor refinements, such as the addition of additional unit tests, extended slightly into the documentation period. However, this did not affect the planned delivery of required outputs.

4.2 Tech Stack

The following tools were employed throughout the development of the project, each selected based on their suitability to the project's needs and prior experience using them.

- **Node.js** and **NestJS** were used to build the backend of the application. **NestJS** was chosen for its structured architecture and native **TypeScript** support, which aligns well with the project's modular needs.
- **React** was selected for the frontend due to its component-based model and prior experience from academic and work-related contexts.
- **Spotify Web API** was used for playlist creation, preview, and storage directly in the user's Spotify account, supporting **OAuth 2.0** for secure user authentication.
- **PostgreSQL** was adopted as the database solution, with its choice justification detailed in Section 6.2.
- **DeepSeek V3** via **OpenRouter API** was selected after comparison with alternatives, as explored in Section 6.3.
- **TypeORM** was utilized as the Object-Relational Mapper, with its implementation details in Section 6.4.
- **Git** and **GitHub** were used for version control, ensuring safe iteration throughout the project lifecycle.
- **Visual Studio Code** was the IDE used during development due to its vast extension support and seamless integration with the stack.

5 SYSTEM DESIGN

In order to implement the described application, the overall established approach involves dividing the book into chapters and generating song suggestions for each of them. This path was chosen in contemplation of the accuracy of the recommended music, considering a novel's content goes beyond its synopsis and metadata. Therefore, a data set has been created to store the accordingly structured books, from which the user can select.

Every chapter of the book is analyzed by an LLM that abstracts from each one aspects such as its content, tone and general mood. Then, the same model, based on the result of the previous analysis, returns a restricted amount of songs, which are stored, filtered, so that there are no repetitions, and combined into a single list that is sent to the user.

Finally, an integration with the API of the digital music service, Spotify, has been implemented so as to allow the user to play a snippet of the suggested music and save it to their library in the streaming application.

A general overview of the system's design, illustrating its main modules, their responsibilities, and the interactions between them, is presented in Figure 1.

6 DEVELOPMENT

6.1 Database schema

The database schema, as illustrated in Figure 2, contains four entities, with self-explanatory properties. These properties were included based on their necessity for both for backend operations and for frontend visualization (for example, **Book**'s **cover**).

The **Book** entity serves as the foundation, storing the works users can select to generate their playlists. **Chapter** entity is a weak entity dependent on the **Book**, with its primary key being a composite of **Book**'s **ID** and **Chapter**'s **number**. Additionally, a **Genre** entity was introduced to define the available musical categories, which will help guide the songs suggestion generated by LLM.

Each **Playlist** is associated with one **Book** and has one influential **Genre**. Initially, a many-to-many relationship between **Playlist** and **Genre** was considered. However, the core purpose of the **Playlist** entity is to minimize token usage by storing the final output of the LLM analysis. Allowing different genres would lead to countless combinations, ultimately defeating the main purpose of the entity. The entity was later extended to also minimize requests to the Spotify Web API. Therefore, a column was added to store Spotify URIs, unique resource identifiers for the tracks [2], so that these can be added to the generated playlist.

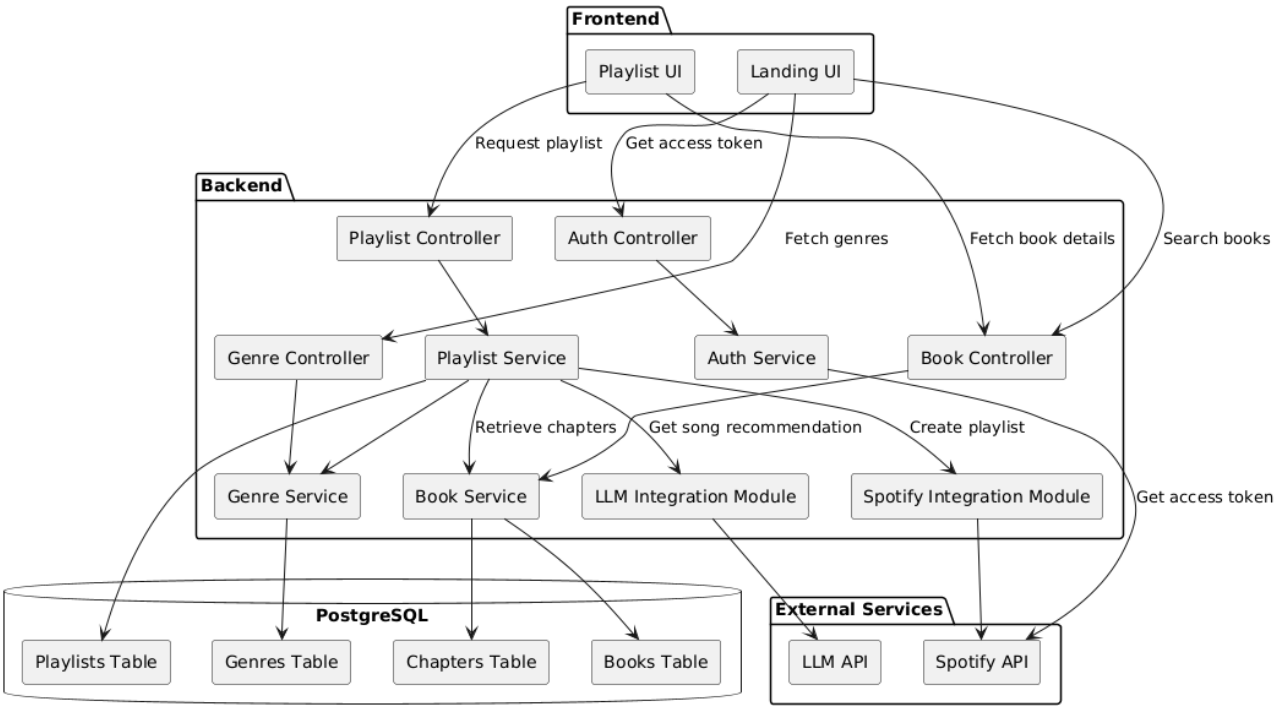


Fig. 1: Component Diagram

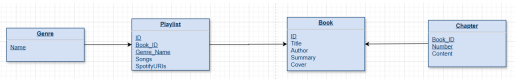


Fig. 2: Database Schema

6.2 Database implementation

A relational model was chosen due to greater familiarity from past college coursework. The database selected was PostgreSQL, not only because it is widely adopted and recognized, but also due to its strong adherence to data integrity standards. In addition, it offers native support for arrays [3], which is useful for storing the playlists, and support for extensions that enable fuzzy text matching, which is essential for the application's autocomplete behavior.

As the system is in a relatively early stage, only a limited number of books are stored. Regarding the available genres, those are based on widely recognized and commonly used music categories, with some deliberate omissions of genres considered less relevant to the use case, for example, gospel.

6.3 LLM integration

The selection of the LLM model for this project was, for the most part, guided by cost-benefit, having efficiency and affordability as key factors. After extensive research and testing with different models, including *Llama 3.3*, *Gemma 3*, *Reka Flash 3*, *Qwen 2 7B Instruct*, and several variants of *Mistral*, **DeepSeek V3** was ultimately chosen.

DeepSeek V3 stands out as the only open source model that offers performance comparable to *GPT-4*. It demonstrates strong capabilities in analyzing and interpreting textual contexts and supports a large context window [4], making it suitable for processing extensive texts, such as book chapters. In the specific context of this project, it offered better results than the other tested models.

The tests between the different models were carried out through the **OpenRouter API**, which provides unified access to various models through a single endpoint. The platform also automatically handles fallbacks and selects the most cost-effective options [5]. Since **DeepSeek** had already been integrated through this tool, it was maintained for the project's development.

An integration module was developed in the backend, within which a service function makes dynamic requests to the API leveraging the **HttpService** provided by NestJS's **HttpModule**, a wrapper around the popular HTTP client, Axios [6].

6.4 Backend integration

The integration between the backend and the database was done leveraging **TypeORM**, considering it is the most mature Object Relational Mapper available for TypeScript and it integrates well with Nest [7]. A module was created for each database table, each containing an entity with its respective columns and relations properly defined in order to make queries easier to write and manage. In the services of each module, methods were implemented to retrieve and save data as needed, making use of **TypeORM**'s **Repository** abstraction.

6.5 Chapter analysis and music suggestion generation

This overall process to analyze book chapters, generate music suggestions, and process the output for storage was done in a single flow, still, it is organized into separate functions. The flow begins by checking if a playlist already exists for

the specified book and genre, and if found, it is returned immediately. Otherwise, the system retrieves the chapters of the specified book, and passes their total count to a helper function, which returns how many songs should be recommended per chapter, based on the number of chapters.

Next, the system iterates through each chapter, sending its content, along with the number of songs, target genre, and the book's title and author, to a utility function that builds the LLM prompt. The prompt instructs the model to analyze the chapter and suggest songs that align with its themes, emotions, and events. Although each playlist is associated with a single genre, the prompt indicates that this genre should be treated as a preference rather than a strict requirement, and other factors, such as emotional and lyrical alignment, take priority. The expected output format is also specified to ensure consistent results.

The raw response from the LLM is then passed to a parsing function, which uses JSON parsing and verifies that the result is a valid string array before returning it. The parsed song list is then filtered by another utility function to remove duplicates. Although it was considered to include already recommended songs in the prompt, this was avoided to keep the prompt concise and avoid decreasing output quality, especially as longer lists could negatively affect the LLM's performance. Finally, the array of unique songs is stored in the database.

6.6 Backend API design

There are four API endpoints exposed to the frontend, grouped under **Book**, **Genre** and **Playlist** categories.

The **GET /book/search** endpoint allows searching through the database for a book by title or author. It requires a string query parameter, which can be either the book's title or the author's name.

Initially, full-text search was used due to its capacity to filter out filler words and handle word variations, such as "read", "reading", and "reads". However, this approach is more suitable for full-page search scenarios, which are not planned for the current scope of the application. Since the intended functionality is an autocomplete dropdown, the full-text search is not appropriate, considering the user would have to type the entire word for any result to appear. Later on, if a full search page is added, full-text search can be reintegrated to rank and refine the results.

To support the autocomplete behavior, the query has been updated to use **pg_trgm** extension [8], which enables fast, fuzzy text matching using trigrams, sequences of overlapping three characters. This improves search flexibility, allowing matches even with typos or partial input. After enabling the extension, trigram indexes were created on the **author** and **title** fields of the **Book** table. The search query compares the query string against both fields, ranking the result based on the higher similarity scores.

The next two endpoints are more straightforward. The **GET book/:id** takes book ID as a path parameter and returns the book's stored details: author, title, summary and cover. Meanwhile, the **GET genre/list** requires no parameters and returns all available music genres stored in the database.

Lastly, the **POST playlist/generate** endpoint initiates the complete playlist generation process. It starts by analyzing the book's chapters, generating music suggestions, and formatting the results for storage, as described in the previous section. At this stage, a list with the songs recommended by the LLM was generated and stored in the database. The next step, detailed in the following section, involves using those songs to create the actual Spotify playlist.

6.7 Spotify integration

The process continues by using the recommended songs to create a Spotify playlist. Yet, before interacting with Spotify's Web API, user authentication must be conducted through **OAuth 2.0**. The authentication flow involves exchanging an authorization code for an access token, which the app uses to make API requests on behalf of the user [9]. The OAuth flow implicated two endpoints, which were intentionally omitted from the Swagger documentation as they are not meant to be manually tested or interacted via Swagger UI. These are **GET /auth**, which initiates the authentication process, and **GET /auth/callback**, which handles Spotify's redirect after user authorization.

With the authentication complete, the application proceeds to interact with the Spotify Web API. The first call is to **GET /me** [10], which retrieves detailed profile information about the current user, including their user ID, which will be required to create the playlist under the user's account.

Next, if the song URIs are not already stored in the database, the system iterates through the list of suggested songs and calls **GET /search** [11] for each one. The search request includes different query parameters and selects the top matching result when available. Each returned URI is then added to an array, which is stored in the database once the loop finishes.

Afterward, the application sends a request to **POST /user/:userId/playlists** [12] using the previously obtained user ID. This creates a new playlist under the user's account, with a name and description derived from the book the playlist was generated for.

Finally, the playlist ID is returned from the previous step and used to call **POST /playlists/:playlistId/tracks** [13], where the array of track URIs is sent to populate the created playlist. The playlist ID is then returned.

A sequence diagram illustrating in further detail this main flow, as well as the interactions related to the application's other endpoints, can be found in Appendix A.2.

6.8 Web application UI

The UI of the application is relatively simple and consists of two pages. Each page is divided into two sections displayed side by side on large screens, with the intention to mimic the appearance of an open book. Following React's official documentation and based on familiarity from work environment, routing and navigation is handled using React Router. [14]

Upon visiting the application, the landing page (Figure 3) presents a welcome section with a brief description of the project and instructions on how to use it. Next to it is the selection section, where users can choose a book and a preferred genre for playlist generation.

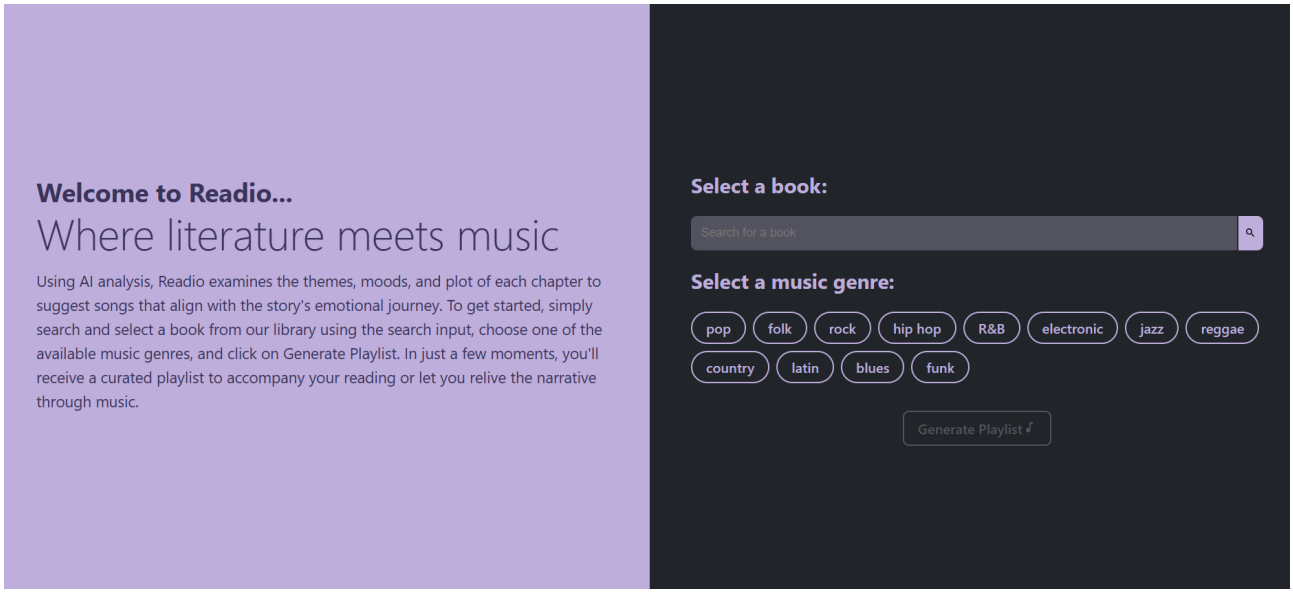


Fig. 3: Landing Interface

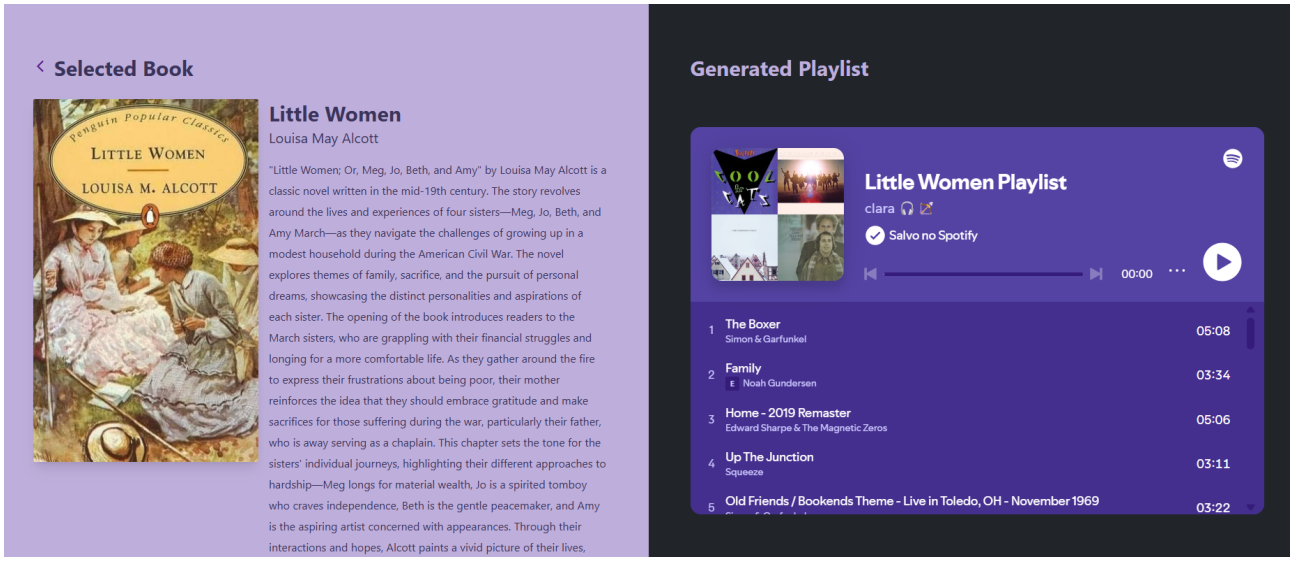


Fig. 4: Playlist Interface

The book search input queries the previously described search endpoint and displays the results in a dropdown, autocomplete-style. The list of genres is fetched on page load, using also a formerly mentioned endpoint, ensuring users can only choose from valid, available options. The "Generate Playlist" button remains disabled until both a book and a genre have been selected. Their values are managed globally throughout the application using React's Context API [15] in combination with useState, allowing state to be passed deeply through the component tree without prop drilling.

When the user clicks the button, it initiates the Spotify authentication flow discussed earlier. Upon completion, the interface automatically redirects to the playlist page.

This second page (Figure 4) displays information about the selected book, retrieved via **GET /book/id**, and the generated playlist. The playlist data is fetched on page load using the authorization token included as a query param, and the returned playlist ID is used to render Spotify's embedded player with the newly created playlist.

6.9 Testing

Testing for both the frontend and backend of the application was conducted using **Jest**, a widely adopted JavaScript testing framework known for its simplicity, speed, and built-in support for assertions, mocks, and coverage reports. Jest integrates seamlessly with TypeScript, making it particularly suitable for projects developed with Node.js and React, as is the case for this application. Unit tests were written for each backend service and controller, as well as for individual frontend components, ensuring that core functionalities behave as expected.

6.10 Deployment

The frontend was deployed using **Vercel**, a platform well-suited for dynamic applications and optimized for frameworks like React, as in this case. Vercel's integration with Git allows for continuous deployment directly from the repository.

For the backend server and database, **Render** was chosen as the hosting platform. It provides a flexible environment for deploying web services and persistent databases. Like Vercel, it supports Git-based deployments, making the process straightforward and easy to manage.

7 RESULTS

The results achieved by the project can be considered satisfactory. As previously outlined, the designed database stores the system's output. Specifically, the **songs** field in the **Playlist** entity holds the array of songs recommended by the LLM based on each chapter, while the corresponding array of Spotify URIs is stored within the **spotify_uris** field, which is essential to link the suggestions with actual tracks that can be played and saved.

As shown in previous figures, the final product presented to the user is a Spotify playlist, displayed on the frontend and saved to their personal Spotify account. This playlist is composed of unique songs, which is ensured through the filtering process that eliminates duplicates. An example of such a playlist, generated for the book *Little Women*, with *folk* set as preferred genre, is shown in Figure 5a.

Remarkably, the playlist includes tracks like "Family" and "Home", which line up well with the book's strong focus on family. From the very beginning, *Little Women* shows that despite challenges such as financial struggle, personal differences among the sisters, and the absence of their father, the March family remain close and grounded in their love for one another.

When generating another playlist for the same book but with a different genre preference, the influence of genre, even if not treated as a constraint, is evident. It noticeably shapes the model's choices, aligning the musical tone of the final playlist with user preference.

In this variation, with the preferred genre set to *hip hop*, the playlist features songs by artists such as Kanye West and 2Pac, as can be seen in Figure 5b. In spite of the musical tone change, the tracks continue to reflect themes of the book. It portrays, as previously discussed, the resilience and warmth despite separation and hardships faced by the March family, with songs such as "Family Business".

Similarly, when generating a playlist for *The Picture of Dorian Gray* with *folk* as the preferred genre (Figure 5c), the resulting songs also reflect the book's core themes, despite sharing the same genre as the Little Women example. While the previous playlist highlighted warmth and family unity, this one leans into a more somber and introspective tone, matching the novel's focus on vanity, youth, and moral decay. Tracks such as "The Trapeze Swinger," "The Sound of Silence," and "Fade Into You" carry a reflective and melancholic atmosphere, resonating with Dorian's emotional isolation and the consequences of his obsession with eternal beauty. This contrast illustrates how the model adapts its songs suggestions to remain sensitive to the narrative and emotional direction of each book, in spite of the genre remaining constant.

Regarding the chapter-level analysis, although the Spotify UI presents the playlist as a continuous list, the system's backend output logs each recommendation alongside its corresponding chapter. This allows verifying that the LLM's suggestions accurately align with key events and emotional tones of the specific chapter. Returning to the *Little Women* example, this time with *pop* as the selected genre, Chapter 4, which focuses on the sisters' daily lives and Marmee's reflections on gratitude, inspired songs like "Brave" by Sara Bareilles and "The Climb" by Miley Cyrus, tracks that reflect themes of perseverance and strength. In the following chapter, where Jo and Laurie begin their friendship, led to the suggestion of songs such as "You've Got a Friend in Me" by Randy Newman and "Lean On Me" by Bill Withers, reinforcing the relationship that develops between them. These examples illustrate how each song can trace back to specific events and emotions from the story.

8 ECONOMIC EVALUATION

The development of the project involved minimal real expenses. However, to simulate a potential real-world scenario, based on a development team and infrastructure, an estimated cost breakdown is also provided.

8.1 Real cost

As development was carried out independently using free-tier services and personal hardware, the actual costs associated with the project were minimal. The only direct cost was the purchase of 20 US dollars, equivalent to approximately 18 euros, in API credits, used for accessing the language model.

8.2 Estimated cost

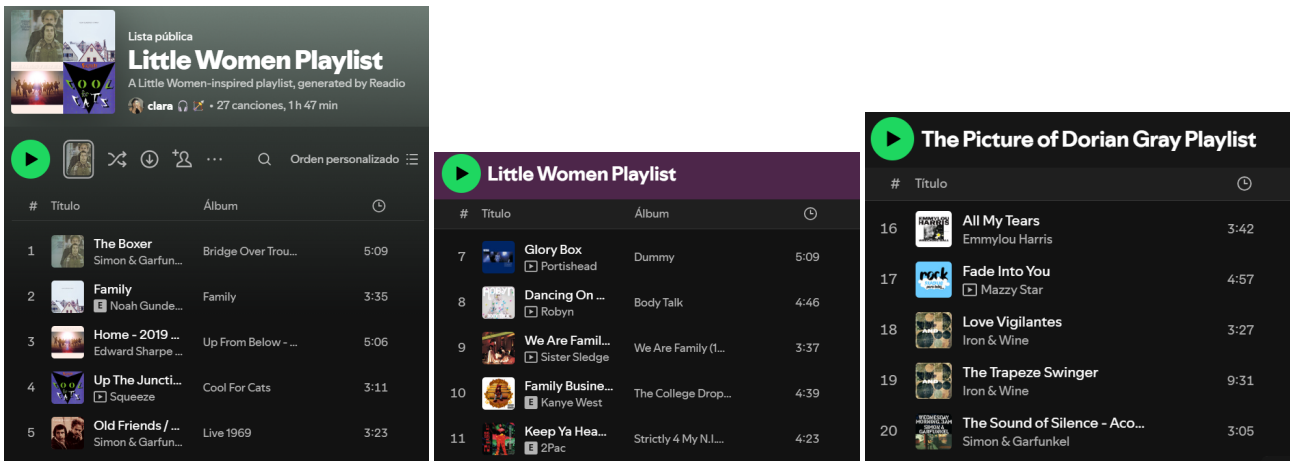
To estimate the cost of scaling this project in a real-world setting with a small team, the following roles and resources were considered:

- **Personnel Costs**: The following estimation is based on average hourly salaries for the relevant technical roles in the Spanish job market. It also takes into account the total workload of 300 hours, as defined by the academic guidelines, for the final degree project. Those hours were distributed among the main roles estimating the effort each one would involve in a project like this.

TABLE 2: ESTIMATED PERSONNEL COSTS

Role	Effort (h)	€/h	Cost (€)
Frontend Developer	90	17	1,530
Backend Developer	100	19	1,900
QA Tester	30	12	360
UI/UX Designer	40	15	600
Project Manager	40	21	840

- **LLM credits**: Since the LLM recommendations are stored to avoid repeated requests, the need for additional credits would remain low even with moderate use. For the estimated scenario, a small buffer of extra credits is considered.
- **Hardware**: The development was conducted using a single personal laptop valued at approximately 700 euros. For estimation purposes, the full team setup of five



(a) Little Women Folk Playlist (b) Little Women Hip-Hop Playlist (c) Dorian Gray Folk Playlist

Fig. 5: Playlist Examples

similar machines would result in an approximate hardware cost of 3,500 euros.

- **Software and services**: The project relied on free-tier plans provided by platforms such as *Vercel* and *Render*, which were sufficient during development. Therefore, there was no direct software service cost, but potential costs for extended usage or scaling are acknowledged.

As a result, the estimated total cost of the project amounts to €8,760, as detailed in Table 3.

TABLE 3: ESTIMATED COST BREAKDOWN

Cost Category	Value (€)
Personnel	5,230
LLM Credits	30
Hardware	3,500
Software	0
Total	**8,760**

9 BUSINESS PLAN

In order to recoup the initial investment and ensure long-term sustainability, a business strategy is designed below. It begins with growth-oriented actions and gradually progresses toward optional monetization methods, maintaining the personal goal of keeping the application free to use for all users.

9.1 Community Growth

This initial step focuses on growing the user base by engaging communities that are already aligned with the app's purpose. Educational institutions, literature teachers, and students represent a strong entry point, as the app can serve as a free enrichment tool in classrooms. At the same time, this strategy can be extended by connecting with online reading communities. Reaching out to book-related content creators can help amplify exposure and generate interest among wider audiences. While this stage is non-monetary, it offers strong value by placing the app in direct contact with frequent readers, building credibility and visibility.

9.2 Publisher Partnerships

With growing visibility, the next step consists in establishing partnerships with publishers, especially independent or smaller ones. Initially, these partnerships can be informal, offering exposure to selected books in exchange for mutual visibility. Over time, this model can evolve into paid sponsorships to promote certain books through playlists.

9.3 Optional Monetization

Once the platform has stable traffic and diverse content, light monetization mechanisms can be introduced without compromising the free model. These may consist of **affiliate links** to buy books or music, generating a small commission per purchase, as well as donation options, such as a "**Buy Me a Coffee**" button, for users who wish to voluntarily support the project.

10 CONCLUSIONS

Despite the overall positive results achieved, there have been challenges along the way. The initial proposal emphasized chapter-by-chapter analysis to enrich the musical suggestions beyond a generic summary-level interpretation. To avoid broad or vague recommendations, only the content of the individual chapter was sent for analysis. This approach allowed the LLM to suggest songs that aligned closely with each chapter's events, making the resulting playlist feel like an evolving soundtrack that follows the book's narrative flow.

However, this introduced a challenge: some suggestions might feel too narrow or disconnected from the overall tone of the book. For instance, in a book with scenes set during Christmas, the LLM suggests holiday-themed songs that fit the moment perfectly but might feel out of place when viewed in the context of the book as a whole. To address this, additional information, such as the book's title and author, was added to the prompt to give the model more con-

text and help it better balance what is happening in each chapter with the overall story.

Another challenging aspect has been the influence of genre. Even though the prompt tells the LLM to treat the selected genre as a preference rather than a strict rule, the playlists still show a clear distinction between genres. This raised concerns that the model might be prioritizing less pertinent cues, such as the musical tone, over more critical narrative elements.

Still, even narrow suggestions can enhance the reading experience, acting like a movie soundtrack that captures the mood of a specific moment. Additionally, in spite of the variation in style, the recommended songs remain generally aligned with the book's themes.

From a technical perspective, the project applies core engineering skills like front and backend development, API design, database modeling and integration of third-party services. A variety of tools were used to cover the project's requirements,including NestJS, React and PostgreSQL, a stack that aligned with technologies explored during coursework and provided practical experience with tools commonly used in professional settings. Regarding the third-party integrations, both Spotify and the LLM were accessed through well-documented public APIs. The use of OpenRouter, in particular, enabled access to advanced models like DeepSeek V3 at low cost, making high-quality results achievable without expensive infrastructure. Overall, the choices reflect a balance between project complexity and available resources, making the results achievable for a computer engineering student within the given timeframe.

Looking ahead, there are areas where the project could be further developed. First, the current pipeline could be expanded to support books beyond the public domain by implementing fallback strategies, such as generating suggestions based on summaries when full content isn't available. Additionally, as the book dataset grows, the current autocomplete search could evolve into a full-page search experience, with support for ranking and filtering using full-text search. Finally, separating the playlist generation into two distinct steps, LLM song recommendation and Spotify playlist creation, would allow users to review and optionally save the playlist, offering greater control and flexibility.

Overall, the three specific objectives defined for the project have been achieved: the language model successfully analyzes and extracts meaning from the chapters, the system stores the suggestions and presents playlists via Spotify integration, and a fully functional application has been built using a limited, yet expandable, dataset.

ACKNOWLEDGMENTS

First and foremost, I would like to express gratitude to my family: to my mother, without whom I would not have had the opportunity to pursue a degree at this institution, and to my father, grandparents, and aunts, whose love and support, even from oceans away, made this journey possible.

To my friends: thank you for standing by me through the highs and lows of the university life.

Finally, I thank my tutor, Xim Cerdá, for believing in the idea from the start and for guiding me through its execution. I truly appreciate all the time you committed to answer my questions.

REFERENCES

[1] Hauck, Pia & Hecht, Heiko. (2023). Emotionally congruent music and text increase immersion and appraisal. *PLOS ONE*, 18, e0280019.

[2] Spotify for Developers, "Spotify URIs and IDs," Spotify Web API Concepts. Available: https://developer.spotify.com/documentation/web-api/concepts/spotify-uris-ids. [Accessed: May 19, 2025].

[3] PostgreSQL Global Development Group, "About PostgreSQL," PostgreSQL. Available: https://www.postgresql.org/about/. [Accessed: Mar. 15, 2025].

[4] PopAi, "Everything About DeepSeek: Key Features, Usage, and Technical Advantages," PopAi. Available: https://www.popai.pro/resources/everything-about-deepseek/. [Accessed: Apr. 12, 2025].

[5] OpenRouter, "OpenRouter Quickstart Guide," OpenRouter Documentation. Available: https://openrouter.ai/docs/quickstart. [Accessed: Mar. 17, 2025].

[6] NestJS, "HTTP Module," NestJS Documentation. Available: https://docs.nestjs.com/techniques/http-module. [Accessed: Mar. 17, 2025].

[7] NestJS, "Database," NestJS Documentation. Available: https://docs.nestjs.com/techniques/database. [Accessed: Mar. 20, 2025].

[8] PostgreSQL, "F.33. pg_trgm," PostgreSQL Documentation. Available: https://www.postgresql.org/docs/current/pgtrgm.html. [Accessed: May 1, 2025].

[9] Spotify for Developers, "Authorization Code Flow," Spotify Web API Tutorials. Available: https://developer.spotify.com/documentation/web-api/tutorials/code-flow. [Accessed: May 4, 2025].

[10] Spotify for Developers, "Get Current User's Profile," Spotify Web API Reference. Available: https://developer.spotify.com/documentation/web-api/reference/get-current-users-profile. [Accessed: May 5, 2025].

[11] Spotify for Developers, "Search for Item," Spotify Web API Reference. Available: https://developer.spotify.com/documentation/web-api/reference/search. [Accessed: May 5, 2025].

[12] Spotify for Developers, "Create Playlist," Spotify Web API Reference. Available: https://developer.spotify.com/documentation/web-api/reference/create-playlist. [Accessed: May 5, 2025].

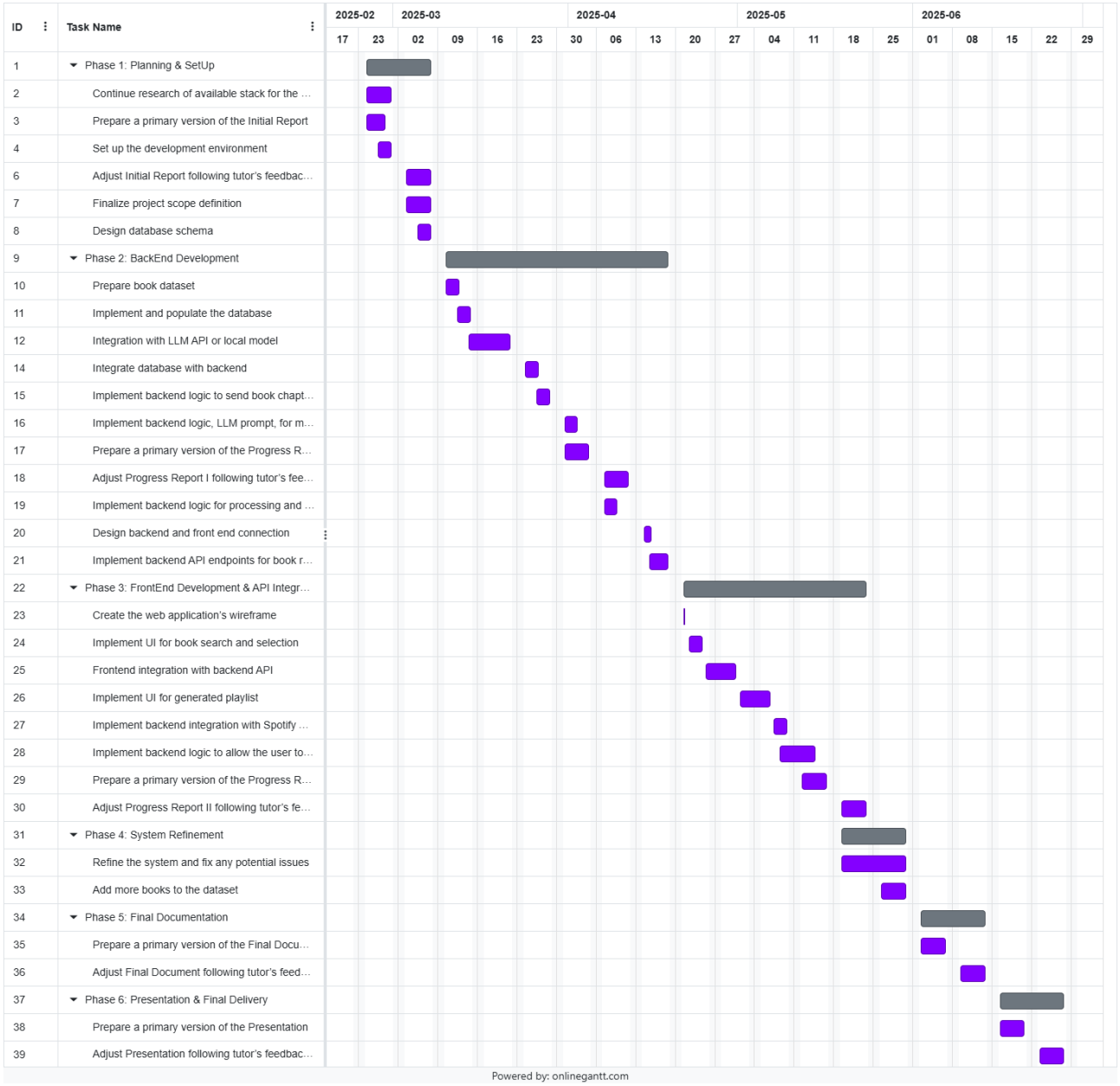
[13] Spotify for Developers, “Add Items to a playlist,” Spotify Web API Reference. Available: https://developer.spotify.com/documentation/web-api/reference/add-tracks-to-playlist. [Accessed: May 5, 2025].

[14] React Router, “Custom Framework,” React Router Documentation. Available: https://reactrouter.com/start/data/custom. [Accessed: May 9, 2025].

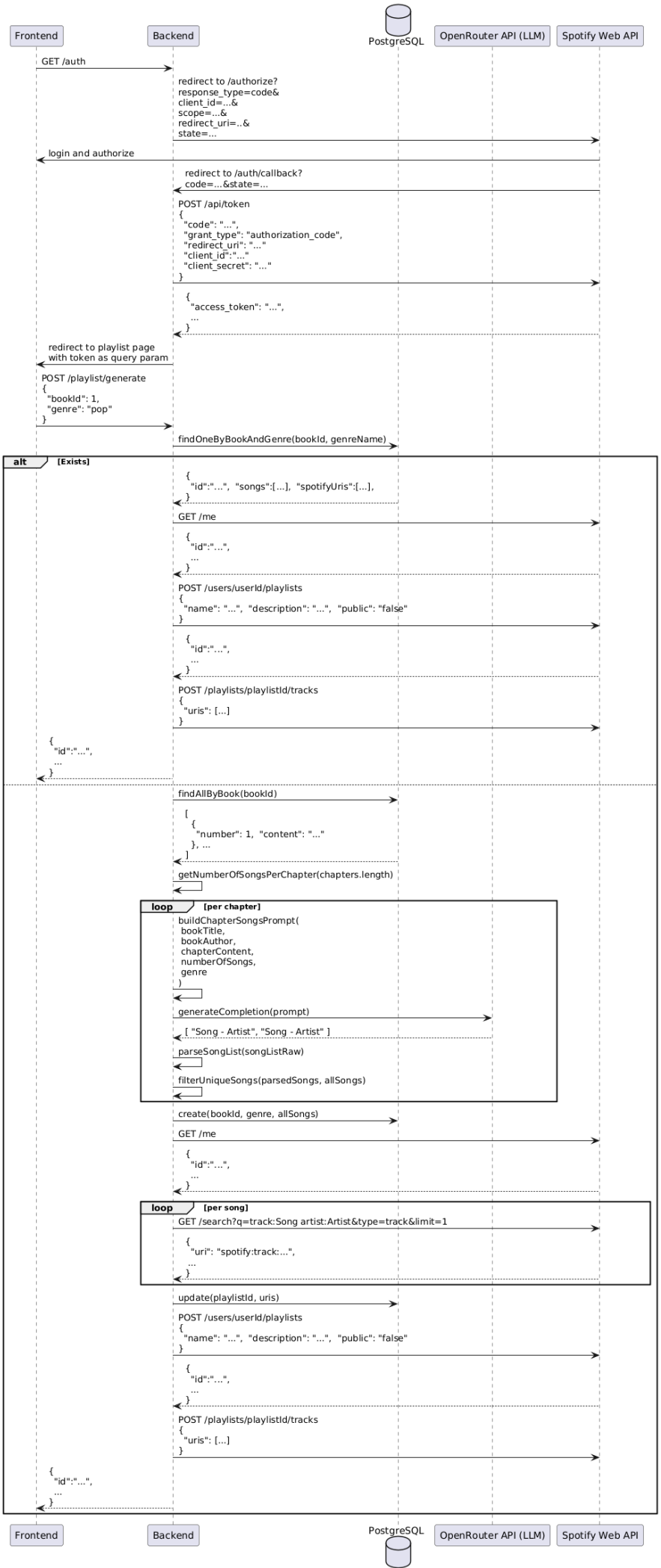
[15] React, “Passing Data Deeply with Context,” React Documentation. Available: https://react.dev/learn/passing-data-deeply-with-context. [Accessed: May 10, 2025].

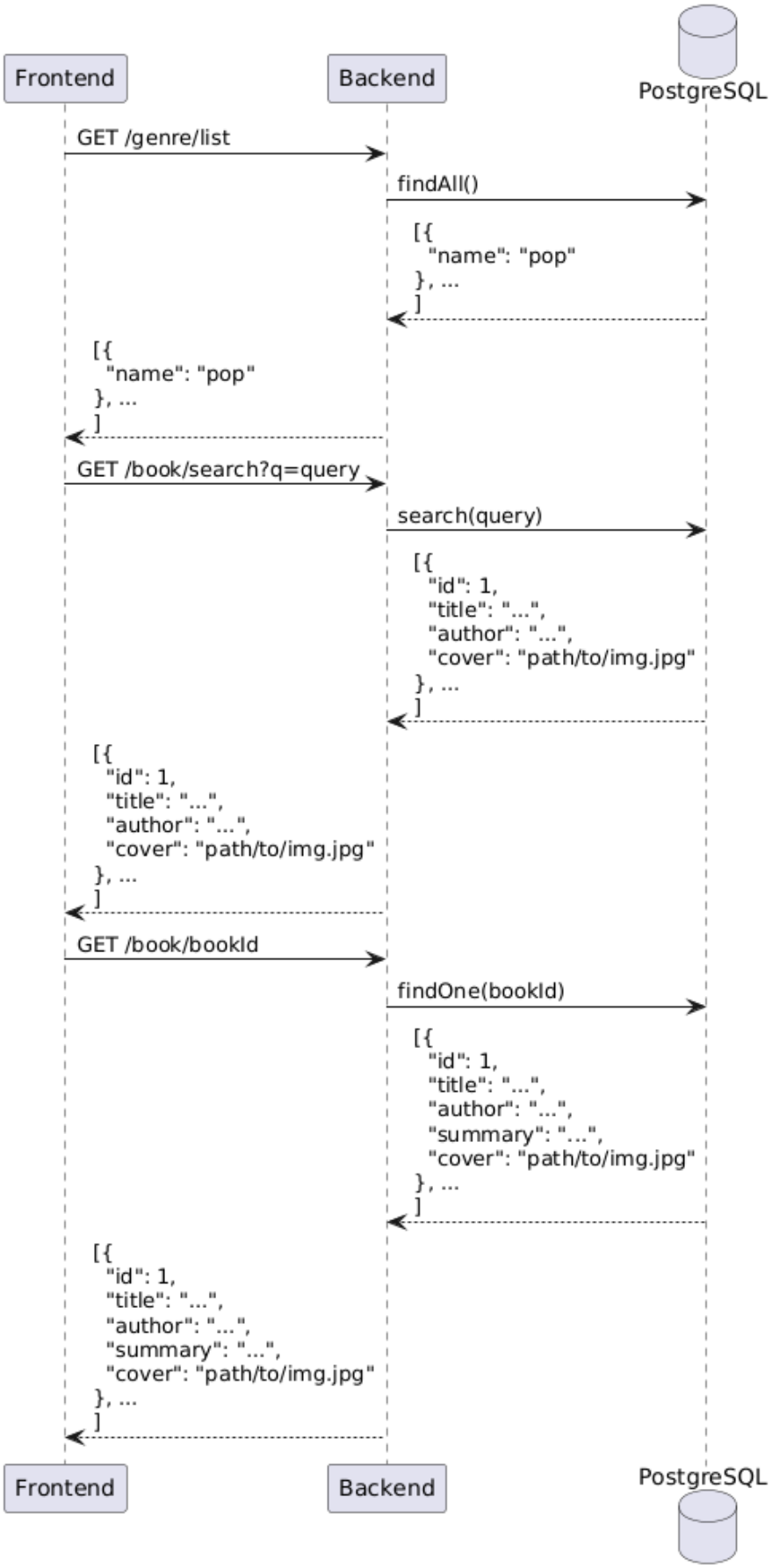
APPENDIX

A.1 Gantt Chart



A.2 Sequence Diagrams


Frontend
Backend
PostgreSQL
OpenRouter API (LLM)
Spotify Web API
GET /auth
redirect to /authorize?
response_type=code&
client_id=...&
scope=...&
redirect_uri=..&
state=...
login and authorize
redirect to /auth/callback?
code=...&state=...
POST /api/token
{
"code": "...",
"grant_type": "authorization_code",
"redirect_uri": "..."
"client_id":"..."
"client_secret": "..."
}
{
"access_token": "...",
...
}
redirect to playlist page
with token as query param
POST /playlist/generate
{
"bookId": 1,
"genre": "pop"
}
findOneByBookAndGenre(bookId, genreName)
alt
[Exists]
{
"id":"...", "songs":[...], "spotifyUris":[...],
}
GET /me
{
"id":"...",
...
}
POST /users/userId/playlists
{
"name": "...", "description": "...", "public": "false"
}
{
"id":"...",
...
}
POST /playlists/playlistId/tracks
{
"uris": [...]
}
{
"id":"...",
...
}
findAllByBook(bookId)
[
{
"number": 1, "content": "..."
}, ...
]
getNumberOfSongsPerChapter(chapters.length)
loop
[per chapter]
buildChapterSongsPrompt(
bookTitle,
bookAuthor,
chapterContent,
numberOfSongs,
genre
)
generateCompletion(prompt)
["Song - Artist", "Song - Artist"]
parseSongList(songListRaw)
filterUniqueSongs(parsedSongs, allSongs)
create(bookId, genre, allSongs)
GET /me
{
"id":"...",
...
}
loop
[per song]
GET /search?q=track:Song artist:Artist&type=track&limit=1
{
"uri": "spotify:track:...",
...
}
update(playlistId, uris)
POST /users/userId/playlists
{
"name": "...", "description": "...", "public": "false"
}
{
"id":"...",
...
}
POST /playlists/playlistId/tracks
{
"uris": [...]
}
{
"id":"...",
...
}


Frontend
Backend
PostgreSQL
GET /genre/list
findAll()
[{
"name": "pop"
}, ...
]
[{
"name": "pop"
}, ...
]
GET /book/search?q=query
search(query)
[{
"id": 1,
"title": "...",
"author": "...",
"cover": "path/to/img.jpg"
}, ...
]
[{
"id": 1,
"title": "...",
"author": "...",
"cover": "path/to/img.jpg"
}, ...
]
GET /book/bookId
findOne(bookId)
[{
"id": 1,
"title": "...",
"author": "...",
"summary": "...",
"cover": "path/to/img.jpg"
}, ...
]
[{
"id": 1,
"title": "...",
"author": "...",
"summary": "...",
"cover": "path/to/img.jpg"
}, ...
]
Frontend
Backend
PostgreSQL