Module Guide for GenreGuru

Team 8 – Rhythm Rangers

Ansel Chen
Muhammad Jawad
Mohamad-Hassan Bahsoun
Matthew Baleanu
Ahmed Al-Hayali

April 5, 2025

1 Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
GenreGuru	Music Feature Engineering & Recommendation Program
UC	Unlikely Change
API	Application Program Interface
(G)UI	(Graphical) User Interface
BPM	Beats Per Minute
WAV	Waveform Audio File Format
REST(ful)	REpresentational State Transfer
JSON	JavaScript Object Notation
ID	IDentifier

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: Format and constraints of the initial input data

AC2: The APIs used to fetch data

AC3: Feature extraction algorithms

AC4: Song Recommendation algorithm

AC5: Database Strucutre and Content

AC6: GUI Implementation

AC7: Addition and/or removal of new/old features

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Source of the input is always from the user

UC2: the usse of Spotify and Deezer as data sources

UC3: the goal of the system is to extract features and recommend songs

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 2. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

Hardware-Hiding Module

...

Level 1	**Level 2**
Hardware-Hiding Module	
Behaviour-Hiding Module	? ? ? ? ? ? ? ?
Software Decision Module	? ? ?

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 3.

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *GenreGuru* means the module will be implemented by the GenreGuru software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented.

Level 1	Level 2
Hardware-Hiding	
Behaviour-Hiding	GUI Module Audio File Upload Module Search Query Module Client Communication Module Server Communication Module Driver Module Tempo (BPM) Feature Extraction Module Key and Scale Feature Extraction Module ~~Instrument Type Feature Extraction Module~~ ~~Vocal Gender Feature Extraction Module~~ ~~Mood Feature Extraction Module~~ Dynamic Range Feature Extraction Module RMS feature Module Instrumentalness Feature Extraction Module ~~Contour Feature Extraction Module~~ Spectral Centroid Feature Module Spectral Bandwidth Feature Module Spectral Rolloff Feature Module Spectral Flux Feature Module Spectral Contrast Feature Module Recommendation Module Program Results Interface Module
Software Decision	Database Spotify API Deezer API ~~Genre Feature Module~~ Spleeter Audio Splitter Module

Table 2: Module Hierarchy

7.1 Hardware Hiding Modules

N/A

7.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

7.2.1 GUI Module (M1)

Secrets: The format and structure of the input data.

Services: Handles user interaction, calling of the sub-input modules.

Implemented By: Frontend UI development framework

Type of Module: Abstract Data Type

7.2.2 Audio File Upload Module (M2)

Secrets: The format and structure of the input data.

Services: Takes in an audiofile and grabs metadata from the user.

Implemented By: Frontend framework file upload widget

Type of Module: Abstract Data Type

7.2.3 Search Query Module (M3)

Secrets: The format and structure of the input data.

Services: Takes in a user input, query searches it and then allows a user to select one of the query results.

Implemented By: Frontend Framework text input widget

Type of Module: Abstract Data Type

7.2.4 Client Communication Module (M4)

Secrets: The format and structure of the input data.

Services: Takes in a user input, query searches it and then allows a user to select one of the query results.

Implemented By: ~~Python~~ JavaScript File

Type of Module: Abstract Data Type

7.2.5 Server Communication Module (M5)

Secrets: The format and structure of the input data.

Services: Takes in a user input, query searches it and then allows a user to select one of the query results.

Implemented By: Python File

Type of Module: Abstract Data Type

7.2.6 Driver Module (M6)

Secrets: The format and structure of the input data.

Services: Takes in a user input, query searches it and then allows a user to select one of the query results.

Implemented By: Python File

Type of Module: Abstract Data Type

7.2.7 Tempo (BPM) Feature Extraction Module (M7)

Secrets: The algorithm for calculating the tempo of the track.

Services: Extracts the Tempo (BPM) from the audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.8 Key and Scale Feature Extraction Module (M8)

Secrets: The algorithm for calculating the key and scale of the track.

Services: Extracts the **key** and **scale** from the audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.9 ~~Instrument Type Feature Extraction Module~~

~~**Secrets:** The algorithm(s) for detecting the type(s) of instruments present within the track.~~

~~**Services:** Detects the presence of different instrument types within the input audio signal.~~

~~**Implemented By:** Python Function~~

~~**Type of Module:** Abstract Data Type~~

7.2.10 ~~Vocal Gender Feature Extraction Module~~

~~**Secrets:** The algorithm for detecting the overall vocal gender of the track.~~

~~**Services:** Extracts the overall vocal gender of the input audio signal.~~

~~**Implemented By:** Python Function~~

~~**Type of Module:** Abstract Data Type~~

7.2.11 Dynamic Range Feature Extraction Module (M9)

Secrets: The format and structure of the input data.

Services: Extracts the Dynamic Range (difference between peak and ~~through~~ median in decibels) of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.12 RMS feature Module (M10)

Secrets: The algorithm for computing the RMS of the track

Services: Computes the RMS of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.13 Instrumentalness Feature Extraction Module (M11)

Secrets: The algorithm for detecting the instrumentalness of the track.

Services: Detects the presence level of instrumentals from the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.14 ~~Contour Feature Extraction Module~~

~~**Secrets:** The algorithm to detect the musical contour of the track.~~

~~**Services:** Extracts the musical contour of the input audio signal.~~

~~**Implemented By:** Python Function~~

~~**Type of Module:** Abstract Data Type~~

7.2.15 ~~Mood Feature Extraction Module~~

~~**Secrets:** The algorithm for detecting the mood of the track.~~

~~**Services:** Extracts the mood of the input audio signal.~~

~~**Implemented By:** Python Function~~

~~**Type of Module:** Abstract Data Type~~

7.2.16 Spectral Centroid Feature Extraction Module (M12)

Secrets: The algorithm for detecting the Spectral centroid of the track.

Services: Computes the Spectral Centroid of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.17 Spectral Bandwidth Feature Extraction Module (M13)

Secrets: The algorithm for detecting the Spectral Bandwidth of the track.

Services: Computes the spectral bandwidth of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.18 Spectral Rolloff Feature Extraction Module (M14)

Secrets: The algorithm for detecting the Spectral Rolloff of the track.

Services: Computes the Spectral Rolloff of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.19 Spectral Flux Feature Extraction Module (M15)

Secrets: The algorithm to compute the spectral flux of the track.

Services: Computes the spectral flux of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.20 Spectral Contrast Feature Extraction Module (M16)

Secrets: The algorithm to compute the spectral contrast of the track.

Services: Computes the spectral contrast of the input audio signal.

Implemented By: Python Function

Type of Module: Abstract Data Type

7.2.21 Recommendation Module (M17)

Secrets: The recommendation machine learning algorithm.

Services: Generates a list of recommended songs.

Implemented By: ~~Machine Learning Algorithm~~ K-NN Algorithm

Type of Module: SKLearn Python Library

7.2.22 Program Results Interface Module (M18)

Secrets: The recommended songs widget and features display element.

Services: Generates widget for recommended song(s) with preview snippet and UI element containting associated features for input song.

Implemented By: Spotify API Calls,

Type of Module: Abstract Data Type

7.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: –

7.3.1 Database Module (M19)

Secrets: The data contained in the tables.

Services: Storages for generated features.

Implemented By: Database Service

Type of Module: Abstract Data Object

7.3.2 Spotify API (M20)

Secrets: Spotify API methods, credentials and metadata.

Services: Song Metadata

Implemented By: Spotify

Type of Module: API Library

7.3.3 Deezer API (M21)

Secrets: Deezer API methods, credentials and metadata.

Services: Converts the input data into the data structure used by the input parameters module.

Implemented By: Deezer

Type of Module: API Library

7.3.4 Spleeter Vocal Audio Splitter (M22)

Secrets: Machine learning model, training data.

Services: Isolates vocal component of the song from non-vocal components.

Implemented By: Spleeter Library

Type of Module: Abstract Data Object

~~**Genre Feature Module (M20)**~~

~~**Secrets:** Genre label for a song~~

~~**Services:** Fetches a genre label for a required song~~

~~**Implemented By:** Deezer API~~

~~**Type of Module:** Abstract Data Type~~

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Requirement	Modules
R1 (Input queries)	GUI Module, Search Query Module, Audio File Module
R2 (Display search results)	GUI Module, Results Display Module
R3 (Play previews)	GUI Module, Results Display Module, Spotify Module
R4 (Validate audio files)	Audio File Module
R5 (Convert audio to WAV)	Audio File Module, Featurizer Module
R6 (Construct Spotify query)	Search Query Module
R7 (Extract features)	Featurizer Module, Driver Module, Genre Feature Module
R8 (Support 8 features)	Featurizer Module, Genre Feature Module
R9 (Generate recommendations)	Recommendation Module, Driver Module
R10 (Transmit user input)	Client-Side Communication Module, Server-Side Communication Module
R11 (Receive recommendations)	Client-Side Communication Module, Server-Side Communication Module

Table 3: Trace Between Requirements and Modules

8.1 Explanation of Traceability

- **R1 (Input queries):** Linked to the GUI Module, Search Query Module, and Audio File Module to handle user inputs efficiently.
- **R2 (Display search results):** Connected to the GUI Module and Results Display Module to display song information accurately.
- **R3 (Play previews):** Associated with the GUI Module, Results Display Module, and Spotify Module to enable audio playback.
- **R4 (Validate audio files):** Linked to the Audio File Module to ensure only valid files are processed.
- **R5 (Convert audio to WAV):** Handled by the Audio File Module and Featurizer Module to standardize inputs.

- **R6 (Construct Spotify query):** Involves the Search Query Module to generate and send API requests.
- **R7 (Extract features):** Requires the Featurizer Module, Driver Module, and Genre Feature Module to process audio files and extract features.
- **R8 (Support 8 features):** Ensures the Featurizer Module and Genre Feature Module handle all required features.
- **R9 (Generate recommendations):** Relies on the Recommendation Module and Driver Module to compute and return suggestions.
- **R10 (Transmit user input):** Linked to the Client-Side Communication Module and Server-Side Communication Module for transmitting data.
- **R11 (Receive recommendations):** Managed by the Client-Side Communication Module and Server-Side Communication Module for receiving processed results.

AC	**Modules**
AC1	M1
AC2	M2
AC3	M3
AC4	M4
AC5	M5
AC6	M6
AC7	M7

Table 4: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

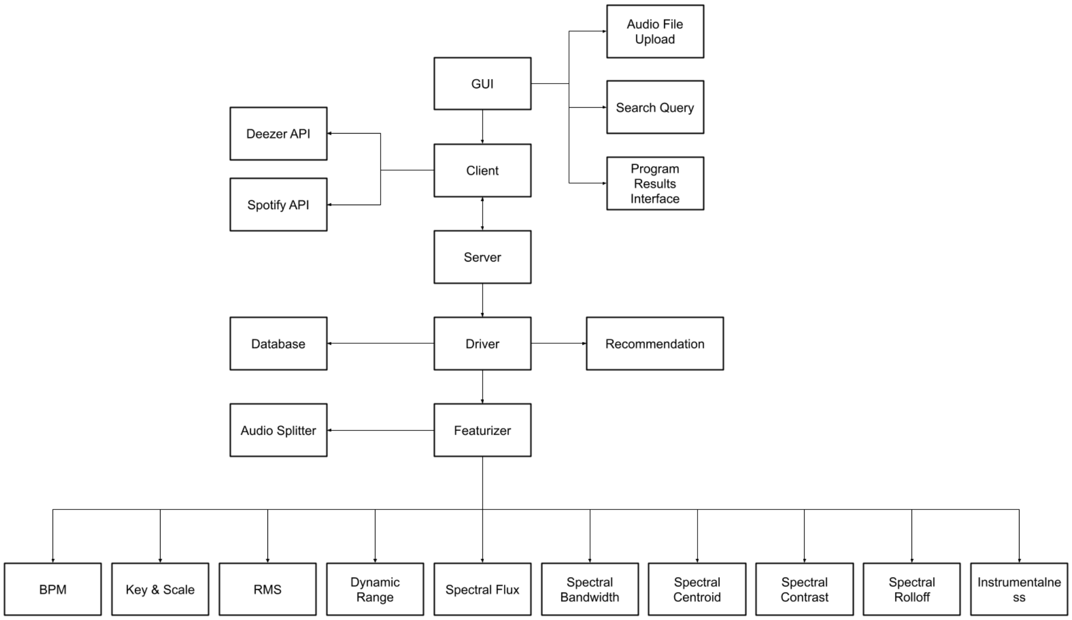

Figure 1: Use hierarchy among modules

10 User Interfaces

11 Design of Communication Protocols

Our design involves two key modules that manage communication between the client and server:

- **Client Communication Module**
- **Server Communication Module**

~~The communication between the frontend systems is done simply through funtion calls. In order to send data (input type, audio data, song ID) from the frontend the user interacts with to the backend (the server) this will be done through encrypted data sent to the hosted server. This would be done by using a RESTFUL interface as this is currently the most common method. Using standard REST API, the payload passed is a regular JSON object with a label for search or file input (the type of input) as that changes the behavior of the backend and the actual song ID itself.~~

The system is architected around a RESTful API that facilitates communication between the frontend (client) and backend (server). This communication protocol is responsible for transferring input data such as the song ID or uploaded audio file, as well as receiving processed recommendations. Data is transmitted over HTTP in the form of JSON payloads.

Figure 2: Home Page - The initial screen where users start their interaction.

Figure 3: Search Results - Shows the top matches for a user's search query.

Frontend-to-Backend Communication

Users interact with the client-facing frontend to either search for a song or upload an audio file. Based on the selected input method, the frontend sends a POST request to the backend

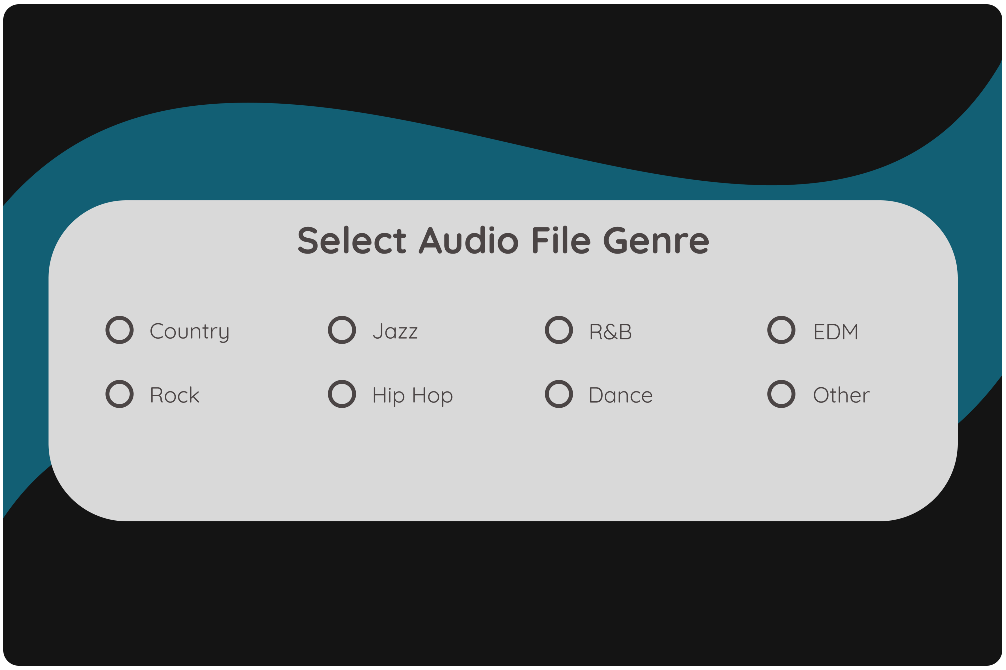

Figure 4: Select Genre - Allows users to specify the genre for uploaded audio files.

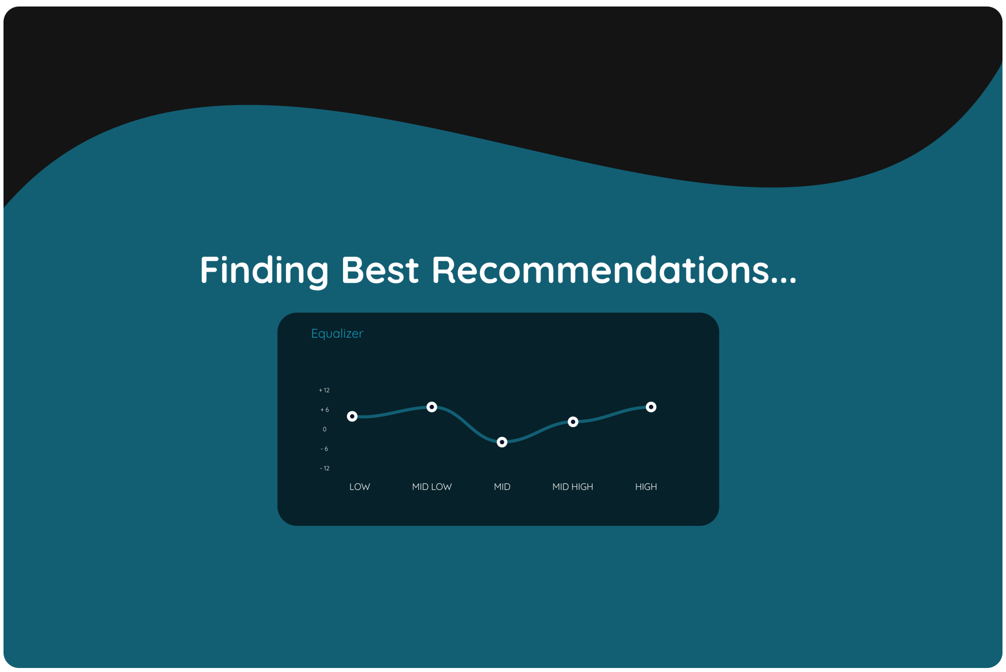

Figure 5: Loading Page - Indicates that the application is processing a request.

API endpoint. The request payload follows a standardized JSON schema containing:

- `type` — either `"search"` or `"upload"`.

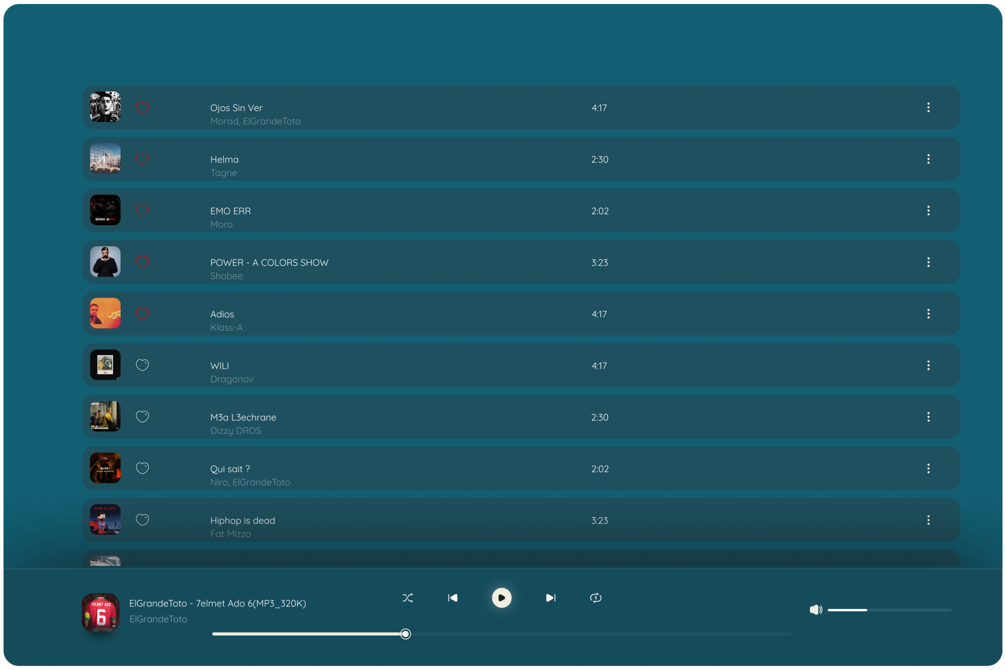

Figure 6: Results Page - Displays the recommendation results.

- `payload` — the song ID (for search) or encoded audio file (for upload).

The backend processes this request and returns a JSON response containing:

- `recommended_songs` — a list of recommended tracks (deezer IDs).

Server Hosting and Ngrok Tunneling

Due to development constraints, the backend server is hosted locally and exposed to the internet using `ngrok`, a tunneling tool that creates a temporary public URL for the localhost server. However, since `ngrok` URLs are not persistent and change each time the tunnel is restarted, we implemented a dynamic solution for maintaining server availability.

Dynamic URL Tracking with Render

To solve the problem of `ngrok`'s impermanence, we created a lightweight `url-store` microservice hosted on Render (which tracks the `render-deploy` branch). The backend server, on startup, sends its current `ngrok` URL to the `url-store` via an API request. This updated URL is then retrieved by the frontend to ensure that all outgoing API requests target the correct endpoint. The `url-store` is continuously deployed by tracking the `render-deploy` branch of our repository.

This design ensures robust communication across environments while enabling rapid testing and iteration during development.

Summary of Flow

1. Frontend fetches the current server URL from `url-store` (available on the `render-deploy` branch).
2. User submits a request (search/upload) from the frontend UI.
3. Frontend sends a JSON POST request to the server via the fetched ngrok URL.
4. Backend processes input, performs feature extraction and recommendation, and responds with JSON data.
5. Frontend displays the results to the user.

12 Timeline

The following timeline outlines the key milestones and activities planned for the development and refinement process:

- **January 6th - January 17th:** Finalize the design document, decide on features and finalize on software architecture.
- **January 18th - January 20th:** Focus on refining the SRS, Hazard Analysis, and VnV documents.
- **January 20th - January 21st:** Conduct a meeting with Dr. MvM with a well-defined agenda to review progress and receive feedback.
- **January 22nd - January 23rd:** Engage with user groups to gather additional insights and validate design decisions.
- **January 23rd onwards:** Begin work on Rev 0 of the project deliverables, incorporating feedback from user groups and the advisor meeting.
- **Feb 4th:** Revision 0 demo: Demonstration of the completed project.
- **March 10th:** VnV Report: Completed testing of the project.
- **March 27th:** Supervisor Final Demonstration.
- **April 4th:** Rev1: Final Documentation Completion
- **April 8th:** Capstone Expo

All team members are responsible for the tasks described in this timeline breakdown are features in the Post-Winter break Check-in Issue.

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.