MetaMusic: The Music Database System

Authors: More Ajinkya, Kumar Sumedh, Ramesh Aditya *Purdue University Fort Wayne*

Supervisor: Prof. Jin soung Yoo *Purdue University Fort Wayne*

ABSTRACT: *Discovering new music is hard, as most of the music platforms suggest music based on what user already listen and pre-existing pattern. Also, In today's music industry, stakeholders face challenges in navigating the vast amount of music content and identifying emerging trends accurately. Existing solutions often rely on subjective metrics and lack comprehensive analysis capabilities, hindering informed decision-making and hindering the discovery of new talent and opportunities. There is a pressing need for a platform that offers objective analysis of music trends, providing users with actionable insights and industry stakeholders with decision support tools to thrive in the competitive music market.*

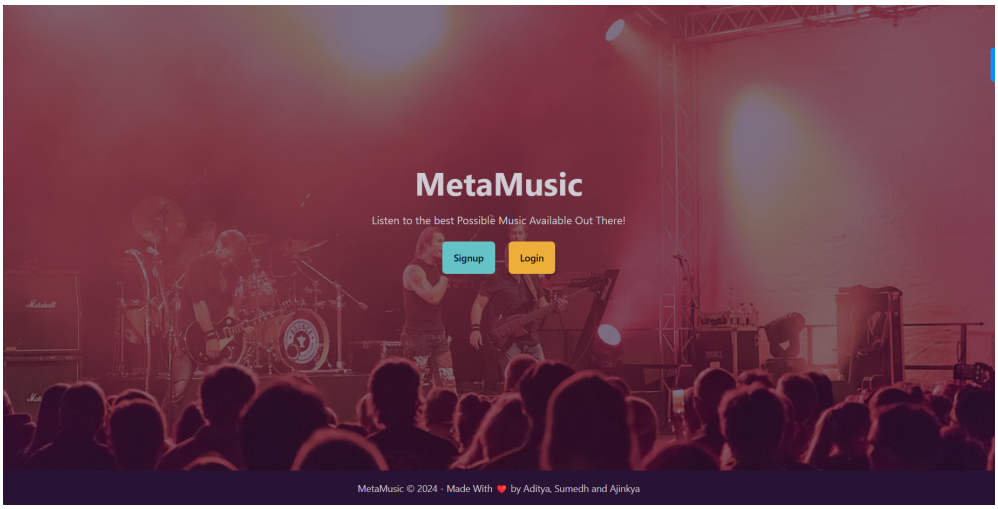

Figure 1: MetaMusic - The Music Database System

1 INTRODUCTION

1.1 Project Description

MetaMusic is a comprehensive open-source music database project aimed at creating a centralized platform for storing, organizing, and accessing music-related information. The project encompasses features such as artist profiles, album and track listings, genre classification, and lyrics integration. Leveraging modern technologies, MetaMusic aims to provide an intuitive user interface for users to search, discover music, and also view analytics related to music and artists.

Central to MetaMusic's mission is its intuitive user interface, crafted to ensure effortless navigation and exploration. Through a seamless blend of functionality and user experience design, MetaMusic empowers users to embark on a journey of musical discovery like never before. Whether delving into the depths of obscure genres or tracking the rise of promising new artists, MetaMusic offers a gateway to unparalleled insight and exploration in the ever-evolving realm of music.

- Design an interactive dashboard that is easy to use;
- Implement logic to handle live real-time data;
- Provide notification with the parametric reason to trigger these notifications;

1.2 Background

Through our research of available solutions in the market, we've identified several options:

- **Spotify:** These platforms offer an intuitive dashboard interface for presenting music streaming. However, they encounter challenges in providing real-time calculations on audience scores for songs.
- **Youtube Music** Similar to Spotify Dashboard platforms, YouTube music tools also present song information. However, they lack the functionality for users to define specific parameters for receiving specified songs based on score.

- **Apple Music:** Apple Music provides a sleek user interface and seamless integration with Apple devices. However, its analytics dashboard may lack depth compared to other platforms, potentially limiting detailed insights into audience engagement.
- **Tidal:** Tidal distinguishes itself with high-fidelity audio streaming and exclusive content. Nevertheless, its analytics tools might not offer the same level of customization and real-time data as competitors, affecting decision-making for content creators.

2 Proposed Solution

2.1 Solution

Our proposed approach entails the analysis of musical data through the utilization of SQL and sophisticated data analytics methodologies. By harnessing contemporary development tools like React, PostgreSQL, Express, and Node, we envision the development of an interactive dashboard. This dashboard will enable users to select favorite songs from various APIs and determine the score for each track, subsequently organizing them into distinct categories based on their scores. Additionally, it will facilitate user registration and session creation to securely store login information.

2.2 Functional Requirement

Music Data Management:

- CRUD operations for managing music tracks, albums, artists, and genres.
- Users can add, edit, delete, and view details of music entities.
- Data validation ensures the integrity and consistency of music data.

Search and Browse Functionality:

- Users can search for music tracks, albums, and artists based on various criteria such as title, artist name, genre, and release year.
- Browsing options allow users to explore music collections by genre or artist.

Trend Analysis and Reporting:

- The system stores data related to music trends, which can be displayed to users and industry stakeholders for analysis and reporting purposes.

3 Data Description

3.1 About the Dataset

We've compiled a dataset consisting of songs by various artists worldwide, along with pertinent statistics retrieved from Spotify and YouTube. This dataset provides insights into the popularity and performance of songs across these platforms.

3.1.1 Content:

The dataset encompasses 26 variables for each song, sourced from Spotify. Here's a concise description of these variables:

- **Track:** Name of the song as displayed on Spotify.
- **Artist:** Name of the artist who recorded the song.
- **Url_spotify:** URL of the artist's page on Spotify.
- **Album:** Album containing the song on Spotify.
- **Album_type:** Indicates if the song is released as a single or part of an album on Spotify.
- **Uri:** Spotify link used to locate the song via the API.
- **Danceability:** Numerical score describing the song's suitability for dancing.

This dataset provides a comprehensive view of the music landscape on Spotify, allowing for a detailed analysis of song performance and trends.

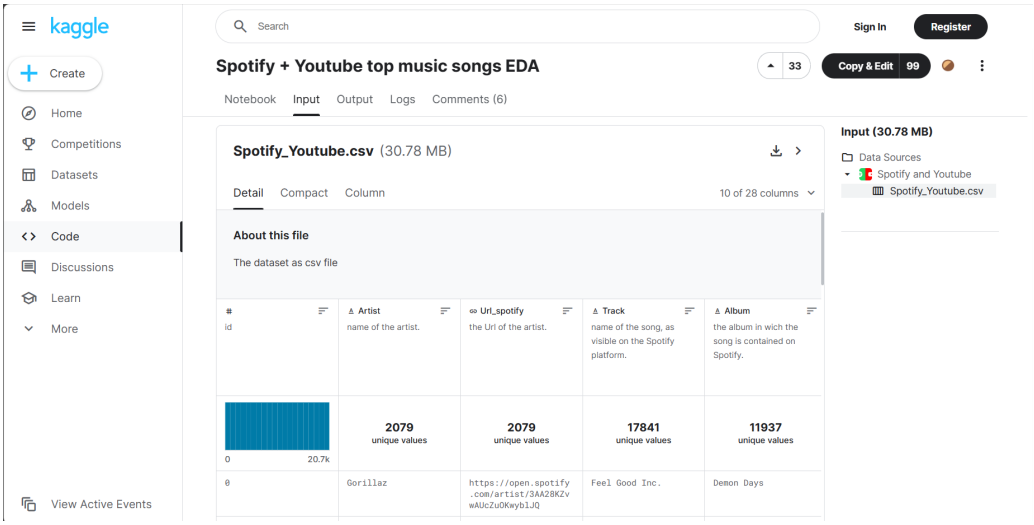

Figure 2: Dataset from Kaggle used for creating the Database

3.2 Prototype System Functionality

1. User Authentication and Authorization:

- Users can register, log in, and manage their profiles, ensuring personalized experiences within the system.
- Robust authentication mechanisms guarantee secure access to the system's features, safeguarding user data and privacy.

2. Music Data Management:

- Comprehensive CRUD (Create, Read, Update, Delete) operations empower users to efficiently manage music tracks, albums, artists, and genres.
- Users have the flexibility to seamlessly add, edit, delete, and view details of music entities, enhancing their control and customization options.
- Stringent data validation protocols ensure the integrity and consistency of music data, maintaining the accuracy and reliability of the system.

3. Search and Browse Functionality:

- Advanced search capabilities enable users to explore music tracks, albums, and artists based on diverse criteria such as title, artist name, genre, and release year.
- Intuitive browsing options facilitate effortless exploration of music collections by genre or artist, providing users with tailored and engaging experiences.

4. Trend Analysis and Reporting:

- The system captures and stores data related to music trends, enabling comprehensive analysis and reporting functionalities.
- Users and industry stakeholders gain valuable insights into music trends, allowing informed decision-making and strategic planning.

3.3 Frontend Components

i) User Interface (UI):

- Responsible for presenting data and interacting with users.
- Includes components for browsing music, searching, viewing trends, and accessing educational resources.
- Developed using HTML, CSS, and React.js for dynamic and responsive UI.

ii) Visualization Tools:

- Utilized for presenting trend analysis reports, charts, and graphs.
- Implemented with the Chart.js library for creating interactive and visually appealing data visualizations.

iii) Authentication and Authorization:

- Handles user authentication and authorization processes.
- Utilizes JSON Web Tokens (JWT) and OAuth for secure user authentication.

3.4 Backend Components

i) Application Server:

- Acts as the intermediary between the frontend and backend systems.
- Implements the business logic, handles requests from the frontend, and communicates with the database.
- Developed using server-side programming language: Node.js with Express.js.

ii) Database Management System (DBMS):

- Stores and manages the music data, user profiles, and system configurations.
- Uses a relational database: PostgreSQL.

iii) External API Integration:

- Facilitates integration with external services for data aggregation (e.g., streaming platforms).
- Implements API endpoints for data retrieval and synchronization.
- Utilizes the Fetch API for making HTTP requests and handling API responses.

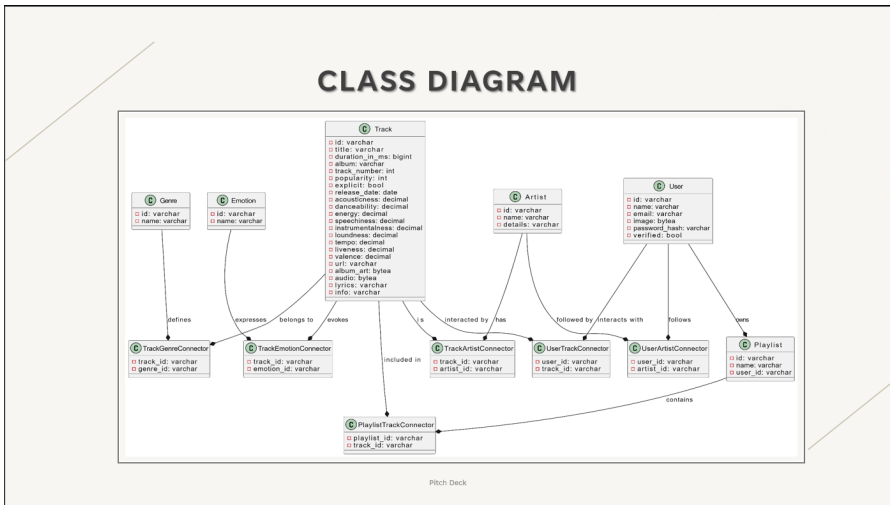

Figure 3: Database System Class Diagram

3.5 Database Schema Description for Music Platform

The class diagram provided illustrates the relational database schema for a comprehensive music platform, detailing the entities and their relationships essential for managing a robust music streaming service. Below is an analysis of each entity and their interconnections:

Entities

- **Track**: Serves as the central entity with attributes such as ID, title, duration, album name, and various musical characteristics (e.g., danceability, energy, loudness). These attributes capture both the metadata and musical properties of music tracks.
- **Genre and Emotion**: These categorical entities classify tracks. Tracks can belong to multiple genres and can express various emotions, facilitated by `TrackGenreConnector` and `TrackEmotionConnector`.
- **Artist**: Contains details about musical artists. Each artist may be associated with multiple tracks, linked through `TrackArtistConnector`, representing the collaboration or solo performances in the recorded tracks.
- **User**: Includes user-specific information like name, email, and password hash. Users can interact with tracks and artists, which is managed through connectors like `UserTrackConnector` and `UserArtistConnector`, enabling functionalities such as track likes or artist follows.
- **Playlist**: Represents user-curated collections of tracks. Each playlist is associated with a user and contains multiple tracks, managed through the `PlaylistTrackConnector`.

Connectors

- `TrackGenreConnector` and `TrackEmotionConnector`: Map tracks to their respective genres and emotions, allowing for many-to-many relationships.
- `TrackArtistConnector`: Links tracks to their artists, indicating artist contributions to each track.
- `UserTrackConnector` and `UserArtistConnector`: Manage the interactions between users and tracks/artists, such as likes and follows.
- `PlaylistTrackConnector`: Connects playlists to their constituent tracks, enabling users to compile and manage their playlists.

This schema effectively supports the data management needs of a music streaming service by organizing information about tracks, artists, users, and their interactions in a structured manner.

3.6 Database Schemas

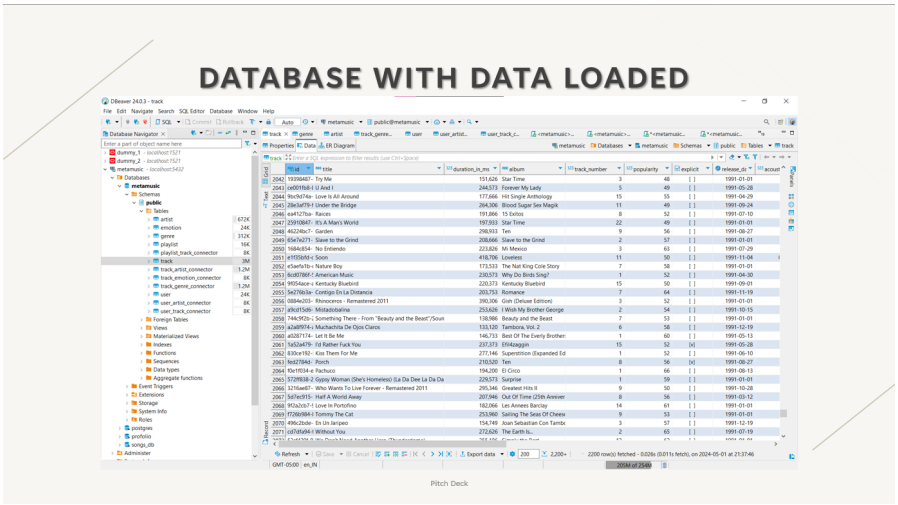

Figure 4: Database with loaded data

3.6.1 Artist Table Definition

The `artist` table stores information about music artists. It includes columns for the artist's unique identifier (`id`), name (`name`), image (`image`), and additional details (`details`).

3.6.2 Emotion Table Definition

The `emotion` table represents different emotions associated with music. It contains columns for the emotion's unique identifier (`id`) and name (`name`).

3.6.3 Genre Table Definition

The `genre` table holds information about music genres. It consists of columns for the genre's unique identifier (`id`) and name (`name`).

3.6.4 Track Table Definition

The `track` table stores details about individual music tracks. It includes columns for the track's unique identifier (`id`), title (`title`), duration (`duration_in_ms`), album (`album`), track number (`track_number`), popularity (`popularity`), explicit content indicator (`explicit`), release date (`release_date`), acousticness (`acousticness`), danceability (`danceability`), energy (`energy`), speechiness (`speechiness`), instrumentalness (`instrumentalness`), loudness (`loudness`), tempo (`tempo`), liveness (`liveness`), valence (`valence`), URL (`url`), album art (`album_art`), audio data (`audio`), lyrics (`lyrics`), and additional information (`info`).

3.6.5 User Table Definition

The `user` table stores information about system users. It includes columns for the user's unique identifier (`id`), name (`name`), email (`email`), image (`image`), hashed password (`password_hash`), and verification status (`verified`).

3.6.6 Playlist Table Definition

The `playlist` table represents user-generated playlists. It contains columns for the playlist's unique identifier (`id`), name (`name`), and the user's identifier who created the playlist (`user_id`).

3.6.7 Playlist Track Connector Table Definition

The `playlist_track_connector` table establishes a many-to-many relationship between playlists and tracks. It includes columns for the playlist's unique identifier (`playlist_id`) and the track's unique identifier (`track_id`).

3.6.8 Track Artist Connector Table Definition

The `track_artist_connector` table establishes a many-to-many relationship between tracks and artists. It includes columns for the artist's unique identifier (`artist_id`) and the track's unique identifier (`track_id`).

3.6.9 Track Emotion Connector Table Definition

The `track_emotion_connector` table establishes a many-to-many relationship between tracks and emotions. It includes columns for the emotion's unique identifier (`emotion_id`) and the track's unique identifier (`track_id`).

3.6.10 Track Genre Connector Table Definition

The `track_genre_connector` table establishes a many-to-many relationship between tracks and genres. It includes columns for the genre's unique identifier (`genre_id`) and the track's unique identifier (`track_id`).

3.6.11 User Artist Connector Table Definition

The `user_artist_connector` table establishes a many-to-many relationship between users and artists. It includes columns for the artist's unique identifier (`artist_id`) and the user's unique identifier (`user_id`).

3.6.12 User Track Connector Table Definition

The `user_track_connector` table establishes a many-to-many relationship between users and tracks. It includes columns for the track's unique identifier (`track_id`) and the user's unique identifier (`user_id`).

3.7 System Architecture

RESTful API Interaction Diagram Description

The diagram represents a typical interaction flow in a RESTful API system involving a user, user interface (UI), API Gateway, and the backend system. The sequence diagram captures how requests are processed and how system behavior changes based on the state of the API Gateway. Below are the key components and their interactions:

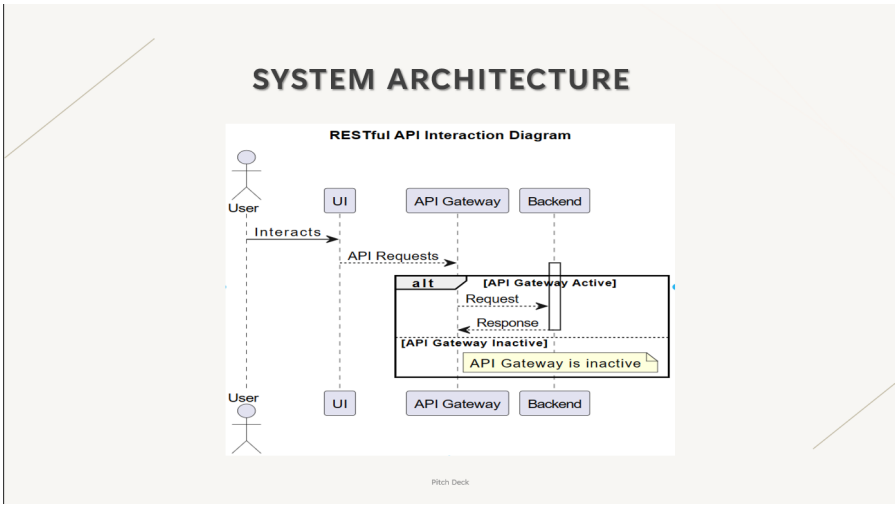

Figure 5: Database System Architecture

Components:

- **User**: The primary actor who initiates interactions by making requests through the UI.
- **UI (User Interface)**: The interface through which the user interacts with the system. It acts as the medium for sending API requests to the API Gateway.
- **API Gateway**: Serves as the intermediary that processes incoming requests and routes them to the appropriate backend services. It is crucial for handling request routing, composition, and protocol translation.
- **Backend**: The backend services perform the necessary operations and generate responses sent back to the user via the API Gateway.

Interaction Flow:

1. **User Interaction**: The user interacts with the system through the UI, which initiates API requests.
2. **API Request Routing**: The UI forwards these requests to the API Gateway, which is responsible for routing them to the appropriate backend services.
3. **Conditional Handling**:
    - **API Gateway Active**: When the API Gateway is active, it processes the requests, forwarding them to the backend, and then routes the backend's response back to the UI.
    - **API Gateway Inactive**: If the API Gateway is inactive, the diagram indicates that the system will handle this condition, likely by notifying the user or the UI of the unavailability of the gateway.

This sequence effectively demonstrates the API request flow and the critical role of the API Gateway in ensuring that requests are processed even when different components have varying availability statuses.

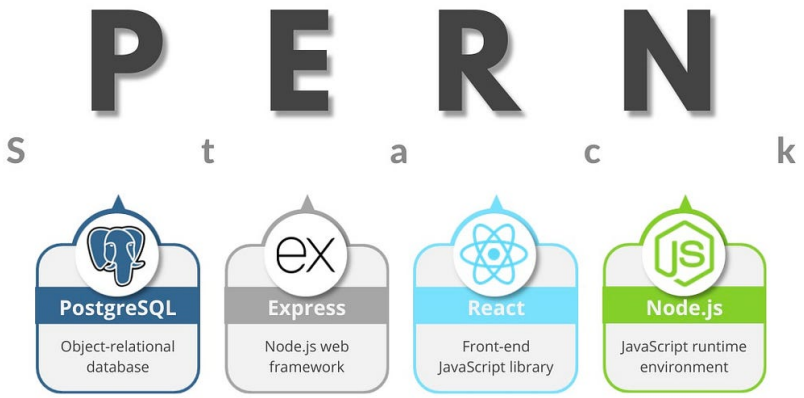

Figure 6: Postgres Express React Node Stack Development

DBMS Technology and Development Methods

Database Management System (DBMS) Technology

For the database management system, our project employs **PostgreSQL**, a powerful, open-source object-relational database system. PostgreSQL was chosen due to its strong reputation for reliability, feature robustness, and performance, especially in handling complex queries and large volumes of data. The choice was also influenced by PostgreSQL's extensive support for SQL standards and advanced features like complex data types, full-text search, and updatable views, which are essential for our application's needs.

3.8 Application Development Technologies

The application is developed using a combination of **Express**, **Node.js**, **React**, and **JavaScript**, forming a stack commonly referred to as the MERN stack (MongoDB, Express, React, Node), with PostgreSQL replacing MongoDB in this instance:

- **Node.js** and **Express** framework are used on the server side. Node.js provides a JavaScript runtime environment that enables asynchronous, event-driven programming, which is crucial for handling numerous concurrent connections inherent in modern web applications. Express, a minimal and flexible Node.js web application framework, is utilized to handle server-side logic, routing, and interaction with the PostgreSQL database, enhancing the server's capability to manage requests efficiently.
- **React** is utilized for the front-end to construct a responsive and dynamic user interface. Its component-based architecture allows for efficient re-rendering of parts of the page when the application's state changes, improving user experience by making the interface more interactive and faster.
- **JavaScript** serves as the backbone for both client-side and server-side code, providing a unified language that streamlines development and reduces context switching for developers.

3.8.1 Development Methodologies

Our development process embraces the **Agile methodology**, characterized by iterative development, regular evaluations, and adaptability to changing project requirements. Agile practices, such as sprint planning, daily stand-ups, and sprint reviews, are integral to our workflow, enabling rapid feature development and continuous improvement based on stakeholder feedback.

Furthermore, elements of **DevOps** are integrated into our project management to enhance the collaboration between development and operations teams. This is achieved through automated pipelines for testing, integration, and deployment, which facilitates a smoother and more reliable delivery process.

3.8.2 Integration and Interface Technologies

The integration between the PostgreSQL database and the Express application is managed through **Node.js** libraries like `pg`, a PostgreSQL client for Node.js. This setup ensures seamless data flow and transactions between the backend server and the database, essential for maintaining data integrity and performance.

3.9 Data and CRUD Operations

3.9.1 Data Management

Our system utilizes PostgreSQL as the primary data storage solution, chosen for its robustness and support for complex data structures. The data managed within our application pertains to users, playlists, songs, and their interrelationships.

3.9.2 CRUD Operations

Create Operations

Users: New user registrations involve creating a new user record in the database, handled by the `/register` endpoint where a new user entry with a hashed password and unique identifier is created after checking for existing email.

Read Operations

Fetching data is managed through various GET requests, such as retrieving user information, fetching top ten popular songs, or specific searches by genre or artist, utilizing SELECT queries to gather data from the database.

Update Operations

Users can update their favorite tracks list, which involves modifying entries in the `user_track_connector` table to reflect new preferences.

Delete Operations

Deletion in the system is handled by endpoints like `/email` for user deletion or `/user/userId/favourite-track/trackId` for removing a track from a user's favorites.

Security and Integrity

Operations such as user registration and login are secured through hashing passwords with bcrypt and using JWT tokens for maintaining session integrity.

3.9.3 Error Handling

Each endpoint is equipped with error handling capabilities to manage exceptions and database errors gracefully. Error responses ensure that the client-side application can respond appropriately to issues like network failures or data consistency errors.

4 Prototype Functionality

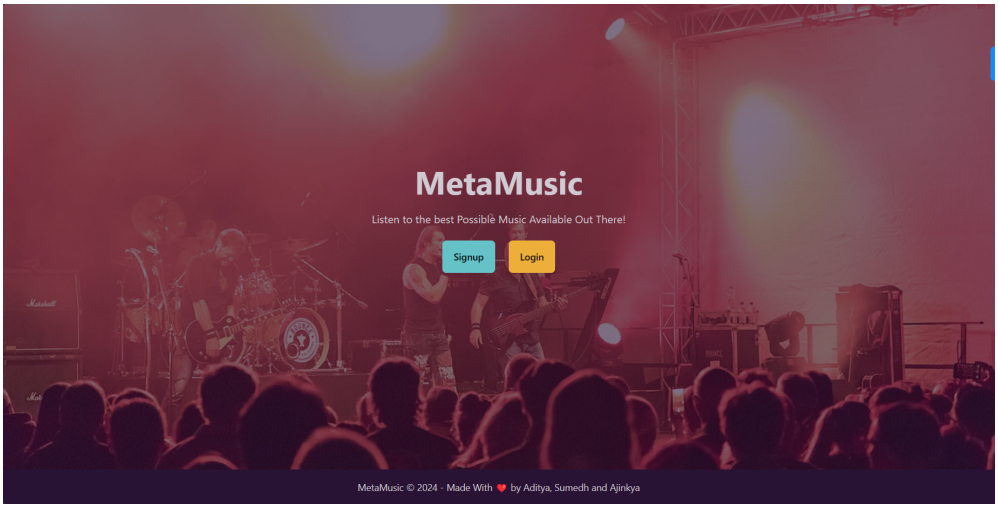

Figure 7: Front Page of Metamusic

The screen shown above is the front page of Metamusic and these page has login and signup action

```
app.post("/register", async (req, res) => {
  const { name, email, password } = req.body;
  const userExistsQuery = 'SELECT * FROM public.user u WHERE u.email = $1';
  const userExistsResult = await pool.query(userExistsQuery, [email]);
  if (userExistsResult.rows.length > 0) {
    return res.status(400).json({ error: 'Email already exists, try logging in' });
  }
  const userId = crypto.randomUUID();
  const hashedPassword = await bcrypt.hash(password, 10);
  const insertUserQuery = 'INSERT INTO public.user (id, name, email, password_hash) VALUES
  await pool.query(insertUserQuery, [userId, name, email, hashedPassword]);
  res.status(201).json({ userId: userId, message: 'User registered successfully' });
});
```

3. User Login

Endpoint: `POST /login`
Purpose: Authenticates returning users and provides them access to their account.
Functionality:

- Verifies the user's credentials against the database.
- Checks if the email exists and if the provided password matches the stored hash.
- Generates a JSON Web Token (JWT) that the user can use to authenticate subsequent requests.

Error Handling: Responds with errors for invalid email or password entries, or internal server errors.
Code:

```
app.post("/login", async (req, res) => {
  const { email, password } = req.body;
  const getUserQuery = 'SELECT * FROM public.user u WHERE u.email = $1';
  const getUserResult = await pool.query(getUserQuery, [email]);
  const user = getUserResult.rows[0];
  if (!user) {
    return res.status(401).json({ error: 'Invalid email' });
  }
  const passwordMatch = await bcrypt.compare(password, user.password_hash);
  if (!passwordMatch) {
    return res.status(401).json({ error: 'Invalid Password' });
  }
  const token = jwt.sign({ email: email }, 'your_secret_key');
  res.json({ token });
});
```

4. Delete User

Endpoint: `DELETE /:email`
Purpose: Allows for the deletion of a user's account from the system.
Functionality:

- Removes the user record associated with the specified email address from the database.

Error Handling: Provides feedback if the deletion was successful or returns an error if the process fails.
Code:

```
app.delete("/:email", async (req, res) => {
  const { email } = req.params;
  const deleteUserQuery = 'DELETE FROM public.user u WHERE u.email = $1';
  await pool.query(deleteUserQuery, [email]);
  res.json({ message: 'User deleted successfully' });
});
```

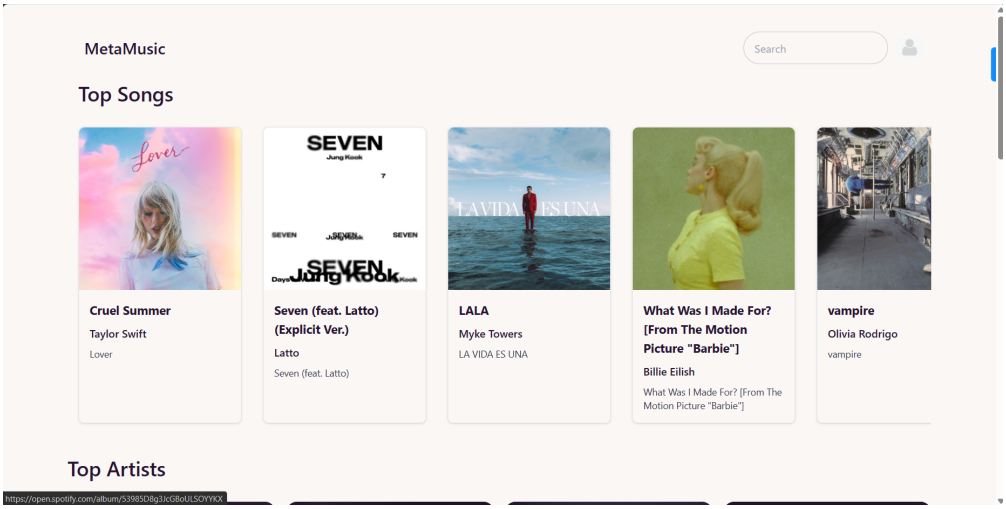

Figure 8: Home of Metamusic

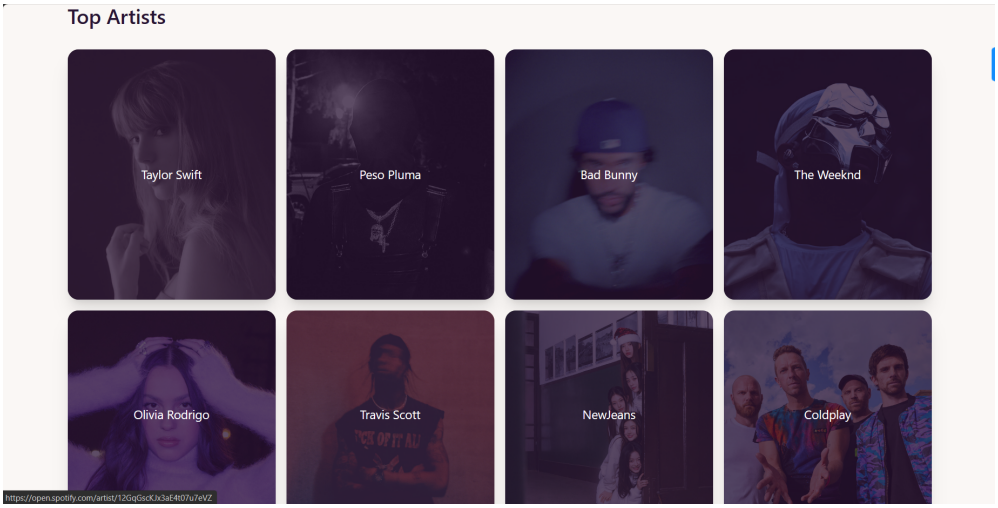

Figure 9: Home of Metamusic

1. User Registration Endpoint

Endpoint: `POST /register`
Purpose: To register new users in the application.
Functionality: This endpoint checks if a user's email already exists to avoid duplicate registrations. If the email is not found in the database, it proceeds to create a new user with a hashed password and a unique identifier (UUID). The new user is then added to the database.
Response on Success: Returns a JSON object with the user ID and a success message.
Error Handling: If the email exists, it returns an error message suggesting the user to log in instead. It also handles server errors by returning a 500 status code.

2. User Login Endpoint

Endpoint: `POST /login`
Purpose: To authenticate users and provide them with a session token.
Functionality: This endpoint retrieves the user's details from the database using the provided email. It verifies the user's password with the stored hash. Upon successful authentication, it generates a JWT token for the session.
Response on Success: Returns a JSON object containing the JWT token.
Error Handling: Returns error messages for invalid email or incorrect password and handles exceptions by returning a server error status.

3. Delete User Endpoint

Endpoint: `DELETE /:email`
Purpose: To delete a user from the application based on their email address.
Functionality: This endpoint removes the user's record from the database using the email address as the key.
Response on Success: Returns a confirmation message indicating successful deletion.
Error Handling: Handles cases where the deletion fails due to server errors by returning a 500 status code.

4. Fetch Top Ten Popular Songs

Endpoint: `GET /top-ten-popular-songs`
Purpose: To retrieve the top ten popular songs based on popularity.
Functionality: This endpoint queries the database for the top ten songs, joins with the artist details, and orders the results by song popularity. It also enriches each song with album art URLs fetched from an external API.
Response on Success: Returns a list of songs with details including title, album, artist, and URLs to album art.
Error Handling: Manages errors during data fetching and external API calls by returning a server error status.

5. Fetch Top Ten Songs by Genre

Endpoint: `GET /top-ten-popular-songs/genre/:genre`
Purpose: To fetch the top ten songs filtered by a specific genre.
Functionality: Queries the database to find top songs in the specified genre using a dynamic genre parameter, orders them by popularity, and limits the results to ten.
Response on Success: Outputs the top ten songs within the requested genre.
Error Handling: Returns error status on failure to execute the query or process the request.

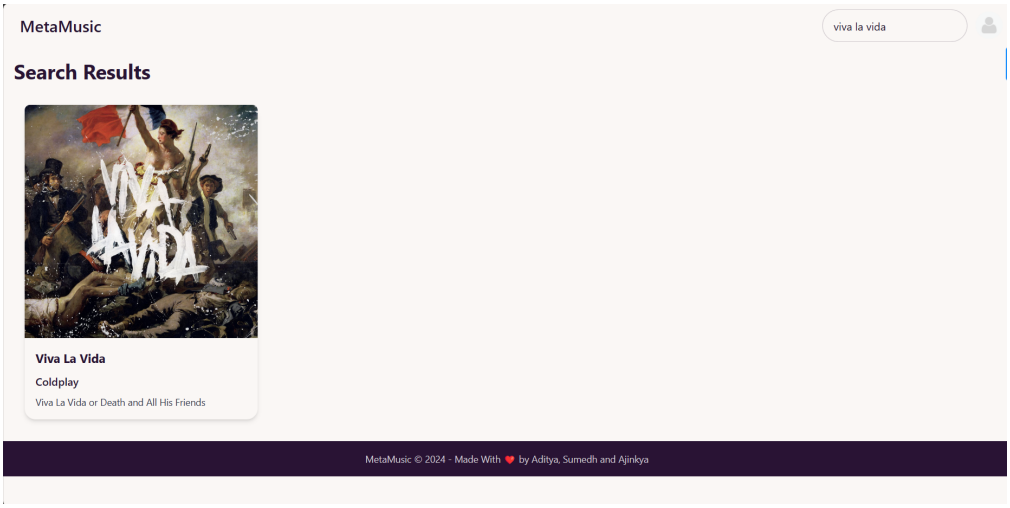

Figure 10: Search functionality for meta music

1. Add Favorite Track

Endpoint: `POST /user/:userId/favourite-track/:trackId`
Purpose: To add a track to a user's list of favorite tracks.
Functionality: This endpoint first verifies the existence of both the user and the track in the database. If both are valid, it adds the track to the user's favorite list.
Error Handling: Returns errors if the user or track does not exist (404) or if there is an internal server error (500).

2. Get Favorite Tracks

Endpoint: `GET /user/:userId/favourite-tracks`
Purpose: To retrieve all favorite tracks of a specific user.
Functionality: Checks if the user exists and then fetches all favorite tracks associated with the user from the database.
Error Handling: Responds with a user not found error (404) or an internal server error (500).

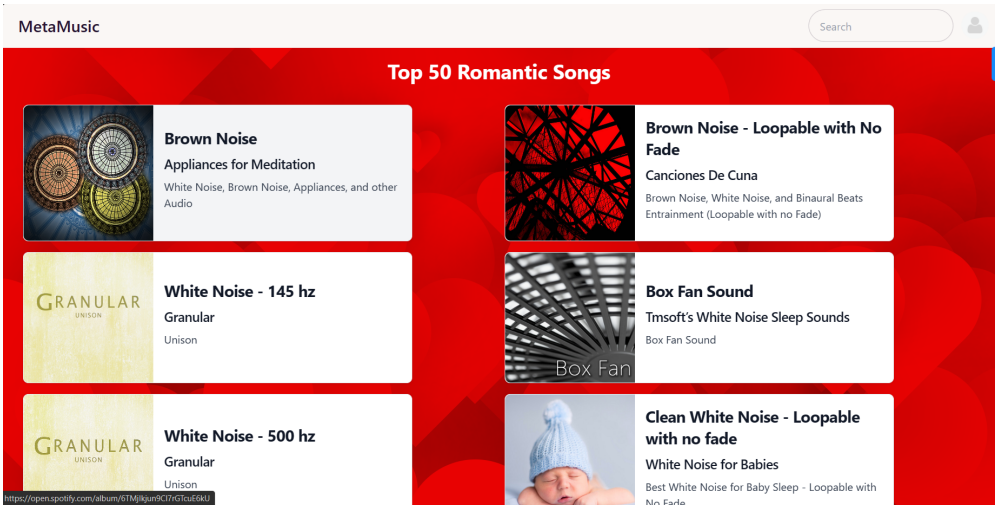

Figure 11: Top Romantic song for metamusic

3. Remove Favorite Track

Endpoint: `DELETE /user/:userId/favourite-track/:trackId`
Purpose: To remove a track from a user's list of favorites.
Functionality: Ensures both the user and the track exist before removing the track from the user's favorites.
Error Handling: Handles not found errors for non-existent users or tracks (404), and server errors (500).

4. Search Tracks

Endpoint: `GET /track/search/:searchTerm`
Purpose: To search for tracks based on a title search term.
Functionality: Performs a database search for tracks matching the search term in the title. Enhances the search results with album art and URLs obtained from an external API.
Error Handling: Manages internal server errors (500) and formats errors related to the search functionality.

These descriptions provide detailed insight into how each endpoint operates within the system, from adding and removing favorite tracks to searching for specific tracks, ensuring that the documentation is both comprehensive and informative.

5 Validation

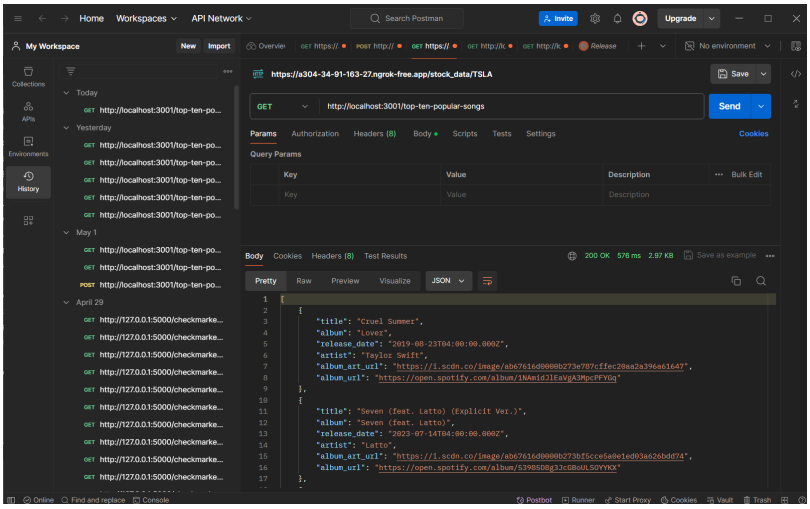

Figure 12: API Testing for meta music

The database is designed with strict constraints including foreign key, primary key, character limit, datatype, and

unique key requirements. If anomalies are detected, the Python script fails and all queries in the transaction are reverted. The backend application includes validation logic for each API, rejecting transactions that fail data requirements. Similarly, the frontend application prevents users from entering invalid values through field validations. APIs have been extensively tested with Postman, ensuring they are free from bugs.

6 Discussion

The project was a success, achieving all features outlined in the scope. A significant challenge was selecting an appropriate database that met all required parameters. Data cleaning was another major challenge encountered. The development of the backend server and the frontend UI proceeded smoothly, supported by the robust design of the relational database and its well-planned data relationships. Future developments might include integrating Machine Learning models to analyze music parameters and lyrics more deeply, enhancing understanding of the emotional context of songs.

7 Conclusion

MetaMusic leverages a blend of modern technologies to create a robust, user-friendly music database platform. The backend uses Node.js with Express.js, optimizing for scalability and real-time data processing in a non-blocking, event-driven architecture. PostgreSQL serves as the relational database management system, supporting ACID transactions and complex queries, suitable for managing large volumes of data. The frontend employs HTML, CSS, and React.js, providing a dynamic and responsive user interface that enriches user experience. MetaMusic also ensures security with JSON Web Tokens (JWT) and OAuth for authentication and authorization. This comprehensive use of advanced technologies enables MetaMusic to meet the needs of users and industry stakeholders effectively.

8 References

- Music Attributes and Popularity Analysis
- SoundCharts: Music Metadata