Moseeqify: Music Streaming Service
Project Documentation
Database Systems
Spring-2024

Submitted To:

Mr. Farhan Sarwar

Submitted By:

Ghulam Mustafa	Fa-2022/BSCS/188
Ammad Rasheed	Fa-2022/BSCS/199
Faizan Ali	Fa-2022/BSCS/187
Ahsan Ilahi	Fa-2022/BSCS/210

Department of Computer Science,

Lahore Garrison University, Lahore, Main Campus.

[June 12th, 2024]

Abstract

Moseeqify is a music streaming service developer for users to access a library of songs, create personalized playlists, and an amazing user interface. The focus of this project is on the database and backend development, ensuring a scalable foundation for an application.
This documentation explains the design and implementation of SQL database, which manages userdata, songs data, playlists, and other data. By using database design principles, authentication, and efficient data management techniques, Our Project Moseeqify aims to deliver a enjoyable music streaming experience.

Introduction

Project Overview:

Moseeqify is a music streaming application designed to show the principles of database design and management under a three-tier architecture. This project focuses on the development of the relational database in Microsoft SQL Server to store and manage user data,music information, and user interactions(history). A backend API (using Flask) and a React JS frontend are also developed for a complete fullstack project with CRUD Operations.

This documentation explains the functionalities of the relational database in Microsoft SQL Server and the Flask API that interacts that with it and also the React.js Frontend

3-Tier Architecture:

Moseeqify has a three-tier architecture for modularity and scalability:

Data Layer (Database): (Microsoft SQL Server)

- Stores all application data using a relational database.
- Uses a well defined db schema with tables for users, artists, genres, albums, songs, playlists,and user interactions like histroy and following.
- Enforces data integrity through constraints for data consistency.

Business Logic Layer (Backend API): (Implemented using Flask)

- Handles user requests and interactions coming from the frontend.
- Connects with the Microsoft SQL Server to manage user data, music library,playlists, and listening history.
- Implementing features like user authentication, searching, playlist management.

Presentation Layer (React JS Frontend):

- Provides a user friendly interface for browsing music, managing playlists, and controlling the music playback.
- Interacts with the backend API.

Technologies Used

Database:

- **Microsoft SQL Server:** For storing and managing all application data.

Backend:

- **Flask**: Micro web framework for Python for backend API.
- **SQLAlchemy**: 3rd Party Library for Python to interact with the database.
- **Flask-Login:** Flask library for user session management.
- **Flask-CORS:** Flask library for Cross Origin Resource Sharing.

Frontend:

- **React JS:** JavaScript frontend library for UI.

Development Tools:

- **Git/GitHub:** Version control system for collaboration of our team.

Database Design

This document outlines the design of the Moseeqify music streaming database. The database uses relational tables with relationships to store and manage music data,user information, and their interactions with the application.

Database Schema:

User Table:

- **username:** Unique identifier for each user, primary key.
- **email:** User's email address, unique and not nullable.
- **name:** User's full name, not nullable.
- **password:** User's password, not nullable.
- **dob:** User's date of birth, not nullable.
- **dateJoined:** Date and time when the user joined, default set to the current timestamp.
- **CHK_User_Age:** Constraint ensuring users are at least 18 years old.

Artist Table:

- **artistID:** Unique identifier for each artist, auto-incremented primary key.
- **name:** Artist's name, unique and not nullable.
- **bio:** Biography or description of the artist.
- **profilepiclink:** Link to the artist's profile picture.

Genre Table:

- **genreName:** Name of the music genre, primary key.

Album Table:

- **albumID:** Unique identifier for each album, auto-incremented primary key.
- **name:** Name of the album, not nullable.
- **artistID:** Foreign key referencing the Artist table.
- **releasedate:** Release date of the album.
- **coverimagelink:** Link to the album's cover image, not nullable.

Song Table:

- **songID:** Unique identifier for each song, auto-incremented primary key.
- **title:** Title of the song, not nullable.

- **artistID:** Foreign key referencing the Artist table.
- **albumID:** Foreign key referencing the Album table.
- **genreName:** Foreign key referencing the Genre table.
- **duration:** Duration of the song, not nullable.
- **audiolink:** Link to the audio file of the song, not nullable.
- **releaseDate:** Date and time when the song was released, default set to the current timestamp.

AlbumSongs Table:

- **albumID:** Foreign key referencing the Album table.
- **songID:** Foreign key referencing the Song table.
- **Composite primary key:** (albumID, songID).

Playlist Table:

- **playlistID:** Unique identifier for each playlist, auto-incremented primary key.
- **name:** Name of the playlist, not nullable.
- **creationdate:** Date and time when the playlist was created, default set to the current timestamp.
- **username:** Foreign key referencing the User table.

PlaylistSongs Table:

- **playlistID**: Foreign key referencing the Playlist table.
- **songID:** Foreign key referencing the Song table.
- **Composite primary key:** (playlistID, songID).

UserListeningHistory Table:

- **username:** Foreign key referencing the User table.
- **songID:** Foreign key referencing the Song table.
- **listeningDate:** Date and time when the user listened to the song, default set to the current timestamp.

user_follows_artists Table:

- **username:** Foreign key referencing the User table.
- **artistID:** Foreign key referencing the Artist table.
- **Composite primary key:** (username, artistID).

Relationships:

One-to-Many:

- An Artist can have many Songs. (Foreign key: artist_id in the Song table)
- An Album belongs to one Artist. (Foreign key: artist_id in the Album table)
- A Song can belong to one Album (optional). (Foreign key: album_id in the Song table)
- A User can create many Playlists. (Foreign key: user_id in the Playlist table)
- A Playlist can contain many Songs (many-to-many relationship implemented through the PlaylistSongs table).
- A User can have a listening history of many Songs. (Foreign key: user_id in the UserListeningHistory table)
- A User can follow many Artists (many-to-many relationship implemented through the user_follows_artists table).

Many-to-Many:

- Albums and Songs are connected through the AlbumSongs table. This allows an album to have many songs, and a song can be included in multiple albums (optional).
- Playlists and Songs are linked through the PlaylistSongs table. A playlist can contain many songs, and a song can be included in multiple playlists.
- Users and Artists follow relationship is represented by the user_follows_artists table. This allows a user to follow multiple artists, and an artist can be followed by many users.

Triggers:

The database uses triggers to automate some actions when data is inserted,updated, or deleted in tables. These triggers perform tasks like:

- **Enforcing data validation rules:** Ensures a user age is 18 or older upon signup(based on the *date_of_birth* in the *User* table).
- **Printing data changes:** Used to log information about inserted, updated, or deleted records.

Stored Procedures:

The stored procedures are categorized into four main types: Deletion, Retrieval, Update, and Insert.

Deletion Procedures:

- **delete_from_user:** Deletes a user from the User table based on username.
- **delete_from_Artist:** Deletes an artist from the Artist table based on *artistID*.
- **delete_from_Genre:** Deletes a genre from the Genre table based on *genreName*.

- **delete_from_album:** Deletes an album from the Album table based on *albumID*.
- **DeleteSong:** Deletes a song from the Song table based on *songID*.
- **DeleteAlbumSong:** Deletes a song from an album in the *AlbumSongs* table based on *albumID* and *songID*.
- **DeletePlaylist:** Deletes a playlist from the Playlist table based on *playlistID*.
- **DeletePlaylistSong:** Deletes a song from a playlist in the *PlaylistSongs* table based on *playlistID* and *songID*.
- **DeleteUserListeningHistory:** Deletes a listening history record from the *UserListeningHistory* table based on *username*, *songID*, and *listeningDate*.
- **DeleteUserFollowsArtist:** Deletes a user-artist follow relationship from the *user_follows_artists* table based on *username* and *artistID*.

Retrieval Procedures:

- **get_table_content:** Dynamically retrieves the content of any table based on the provided table name.

Update Procedures:

- **update_in_User:** Updates user information in the *User* table.
- **update_in_Artist:** Updates artist information in the *Artist* table.
- **update_in_Genre:** Updates genre information in the *Genre* table.
- **update_in_album:** Updates album information in the *Album* table.
- **UpdateSong:** Updates song information in the *Song* table.
- **UpdateAlbumSong:** Updates the album-song relationship in the *AlbumSongs* table.
- **UpdatePlaylist:** Updates *playlist* information in the Playlist table.
- **UpdatePlaylistSong:** Updates the playlist-song relationship in the *PlaylistSongs* table.
- **UpdateUserListeningHistory:** Updates user listening history in the *UserListeningHistory* table.
- **UpdateUserFollowsArtist:** Updates user-artist follow relationship in the *user_follows_artists* table.

Insert Procedures:

- **insert_in_User:** Inserts a new user into the *User* table.
- **insert_in_Artist:** Inserts a new artist into the *Artist* table.
- **insert_in_Genre:** Inserts a new genre into the *Genre* table.
- **insert_in_album:** Inserts a new album into the *Album* table.
- **InsertSong:** Inserts a new song into the *Song* table.
- **InsertAlbumSong:** Inserts a new album-song relationship into the *AlbumSongs* table.
- **InsertPlaylist:** Inserts a new playlist into the Playlist table.
- **InsertPlaylistSong:** Inserts a new playlist-song relationship into the *PlaylistSongs* table.

- **InsertUserListeningHistory:** Inserts a new user listening history record into the UserListeningHistory table.
- **InsertUserFollowsArtist:** Inserts a new user-artist follow relationship into the *user_follows_artists* table.

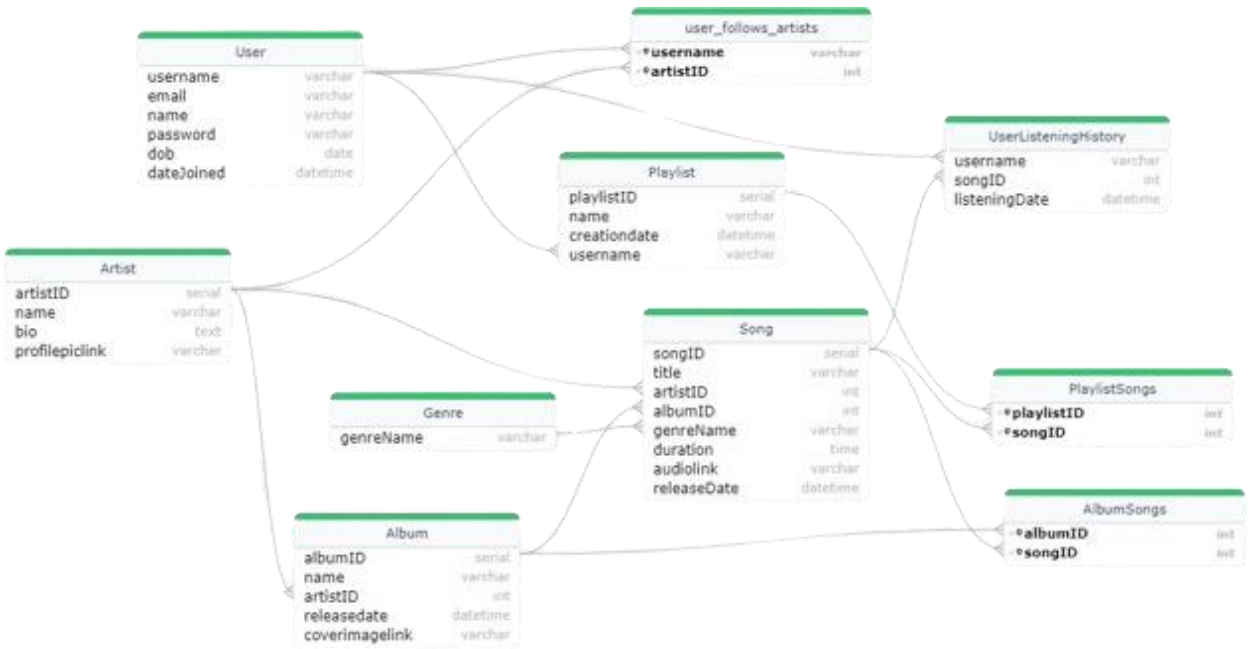
These procedures help in managing the Moseeqify database efficiently by allowing CRUD (Create, Read, Update, Delete) operations on entities such as users, artists, genres,albums, songs, playlists, listening history, and user-artist relationships.

Constraints and Data Integrity:

To ensure the consistency of data, a lot of different constraints are implemented in our the database tables:

- **Primary Key Constraints:** Ensure each record in a table is uniquely identifiable.
- **Foreign Key Constraints:** Maintain integrity between tables.
- **Not Null Constraints:** Ensure fields are not left empty.
- **Unique Constraints:** Prevent duplicate entries where uniqueness is required, such as: *usernames* and *email addresses*.
- **Check Constraints:** Enforce specific rules on the data, such as the *age constraint* in the *User table*.

Entity Relationship Diagram (ERD):



Logical Layer (Backend API)

The backend API of Moseeqify is implemented using Flask, a micro web framework for Python. It serves as the business logic layer in the application's three-tier architecture, handling user requests, interactions, and data processing and interacting with the database, this API manages user authentication, playlists and listening history.

Architecture Overview:

Moseeqify's backend API is built on Flask, SQLAlchemy, Flask-Login, Flask-CORS, and various Python libraries. It uses RESTful principles for handling requests and responses, The architecture is designed with modularity and scalability for good user experience.

API Routes and Functionality:

User Registration (/register - POST):

- Request Body: Accepts username, email, name, password, and date of birth.
- Functionality: Validates input data, creates a new user, and stores it in the database.
- Response: Returns a success message along with the user's serialized data or error messages for missing fields or existing user conflicts.

User Login (/login - POST):

- Request Body: Requires email and password for authentication.
- Functionality: Verifies user credentials against stored data.
- Response: Returns user data upon successful login or authentication error messages.

User Logout (/logout - GET):

- Functionality: Clears the user's session data, logging them out.
- Response: Returns a success message indicating successful logout.

Save Song to Listening History (/save-to-history/int- POST):

- Request Body: Expects the username of the user saving the song.
- Functionality: Logs the song ID in the user's listening history for future reference.
- Response: Indicates successful saving of the song or error messages for missing data.

Create Playlist (/playlists - POST):

- Request Body: Requires the playlist name and username of the creator.

- Functionality: Creates a new playlist associated with the user and stores it in the database.
- Response: Confirms successful creation of the playlist or provides error messages for invalid input.

Add Song to Playlist (/playlists/int:playlist_id/add-song/int:song_id - POST):

- Functionality: Adds a song to a specific playlist identified by playlist_id.
- Response: Indicates successful addition of the song to the playlist or reports if the song already exists in the playlist.

Get Specific Album (/albums/int:album_id/ - GET):

- Functionality: Retrieves detailed information about a specific album identified by album_id.
- Response: Returns album data including ID, name, release date, cover image link, artist, and associated songs.

Get All Songs (/songs - GET):

- Functionality: Retrieves a list of all songs in the database.
- Response: Returns a JSON array containing information about each song, including ID, title, artist, URL, and album.

Get All Albums (/albums - GET):

- Functionality: Retrieves a list of all albums in the database.
- Response: Returns a JSON array containing information about each album, including ID, name, release date, cover image link, artist, and associated songs.

Manage Playlists (/playlists - GET):

- Functionality: Retrieves a list of all playlists in the database.
- Response: Returns a JSON array containing information about each playlist, including ID, name, username of the creator, and songs in the playlist.

Get a Specific Playlist (/playlists/int:playlist_id - GET):

- Functionality: Retrieves detailed information about a specific playlist identified by playlist_id.
- Response: Returns playlist data including ID, name, username of the creator, and songs in the playlist.

Search Songs (/search - POST):

- Request Body: Expects a search query for songs.
- Functionality: Searches for songs based on the provided query.
- Response: Returns a JSON array of matching songs with their ID, title, artist, and audio link.

Follow Artist (/follow-artist/int:artist_id - POST):

- Functionality: Allows a user to follow an artist identified by artist_id.
- Response: Indicates successful follow action or reports if the user is already following the artist.

Unfollow Artist (/unfollow-artist/int:artist_id - POST):

- Functionality: Allows a user to unfollow an artist identified by artist_id.
- Response: Indicates successful unfollow action or reports if the user is not following the artist.
- Fetch User Listening History (/users/<username>/listening-history - GET):
- Functionality: Retrieves the listening history of a specific user identified by username.
- Response: Returns a JSON array of songs in the user's listening history, including ID, title, and artist.

Request and Response Structures:

- **Request Format**: All requests follow the JSON format, ensuring consistency and ease of data transfer.
- **Response Format:** Responses are also in JSON format, containing relevant messages, data, and status codes for client-side handling.

Backend Technologies and Features:

- **Flask Framework:** Utilizes Flask for routing, request handling, and response generation, providing a lightweight and scalable solution.
- **SQLAlchemy:** Interacts with the underlying database, performs CRUD operations, and ensures data integrity through ORM mappings.
- **Flask-Login Library:** Manages user sessions, authentication, and authorization, enhancing security and user experience.
- **Flask-CORS Extension:** Enables Cross-Origin Resource Sharing, allowing communication with the frontend application hosted on a different domain.

Business Logic and Functionality

User Authentication and Authorization:

Implements secure user registration, login, logout and management

Data Management and CRUD Operations:

Handles Create, Read, Update, and Delete (CRUD) operations for user data, playlists, songs, and other entities, maintaining data consistency.

Error Handling and Validation:

Provides great error handling, validation of data, and ensures data integrity to prevent unauthorized access.

Playlist and Song Management:

Features like playlist creation, deletion, and song management , for users toorganize, their music collections.

Listening History Tracking:

Logs user interactions with songs tracks listening history.

Backend Conclusion

The backend API design of Moseeqify represents advanced technologies, good architecture, efficient data management, and secure user authentication. It forms the foundation for music streaming platform, ensuring a good experience for users while maintaining scalability.

Presentation Layer (Frontend Design)

The View Level Schema/Frontend of Moseeqify is developed using React JS, a popular JavaScript library for building user interfaces. This section outlines the structure, components, and functionalities of the React application, detailing how it interacts with the backend API to provide a smooth music streaming experience to users.

Frontend Architecture:

The Moseeqify frontend is built using React JS, a popular JavaScript library for building user interfaces. It uses React's component-based architecture to create reusable and maintainable UI elements

Key Components

The frontend consists of various React components that work together to deliver a cool user experience. Here are some key components and their functionalities:

1. **Login/Signup:** Manages user authentication through login and signup forms. Validates user input and interacts with the backend API for authentication.
2. **Home**: Displays a welcome message, the hero section, and user listening history.
3. Album: Displays detailed information about a specific album, including its cover art, song list, and artist information. Retrieves album data from the backend API.
4. **Song:** Presents details about a particular song, including artist and audio controls. Fetches song data and related content from the backend API.
5. **Playlist**: Allows users to create and view playlists. Manages playlist content by adding songs. Interacts with the backend API for playlist CRUD (Create, Read, Update, Delete) operations.
6. **Search**: Provides a search bar functionality for users to find songs by title or artist.

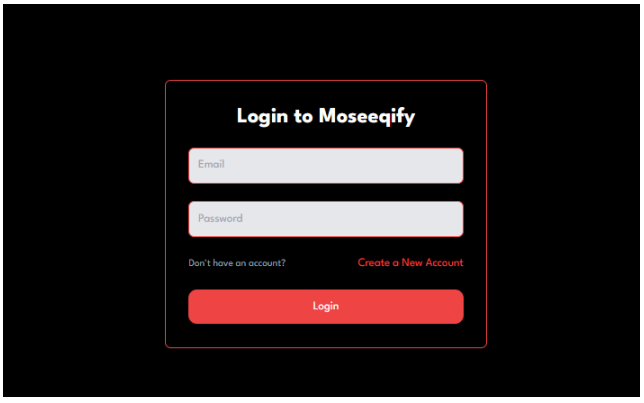
7. **Player**: Renders the music player component, handling playback controls, and volume adjustments.
8. **Navigation Bar**: Provides navigation links to different sections/pages of the application, and also includes an option for signing out.

Frontend Functionality:

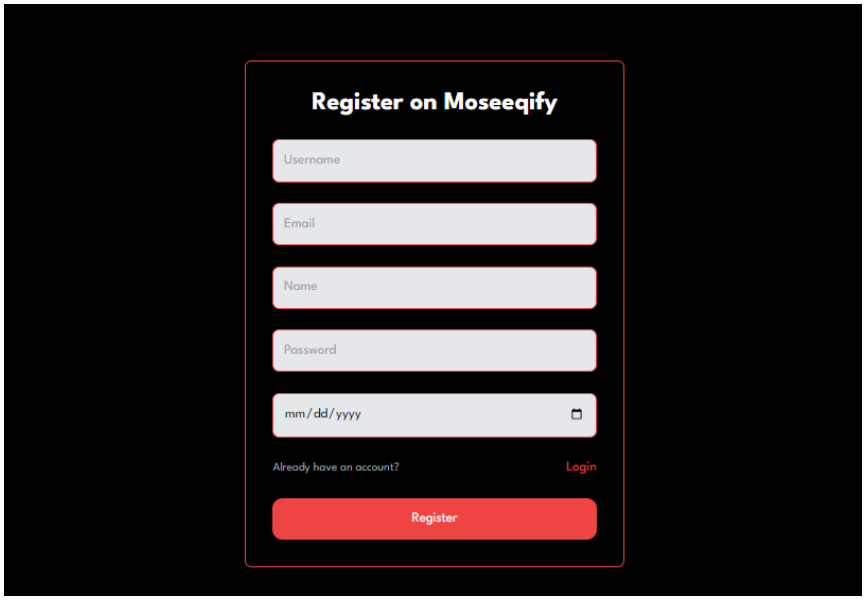
The React JS frontend facilitates various functionalities for users to interact with the Moseeqify music streaming service:

- **User Management**: Allows user registration, login, and logout functionalities. Provides access to logging out of the application as well.
- **Music Browsing:** Allows users to explore the music library by browsing albums, and playlists, or searching for specific songs.
- **Playlist Management**: Allows users to create and manage playlists. Users can add, new songs to their playlists.
- **Playback Control:** Provides audio controls for users to play, pause, reverse, and adjust volume within the music player component.
- **Listening History**: Tracks user listening habits, allowing users to revisit previously played songs.

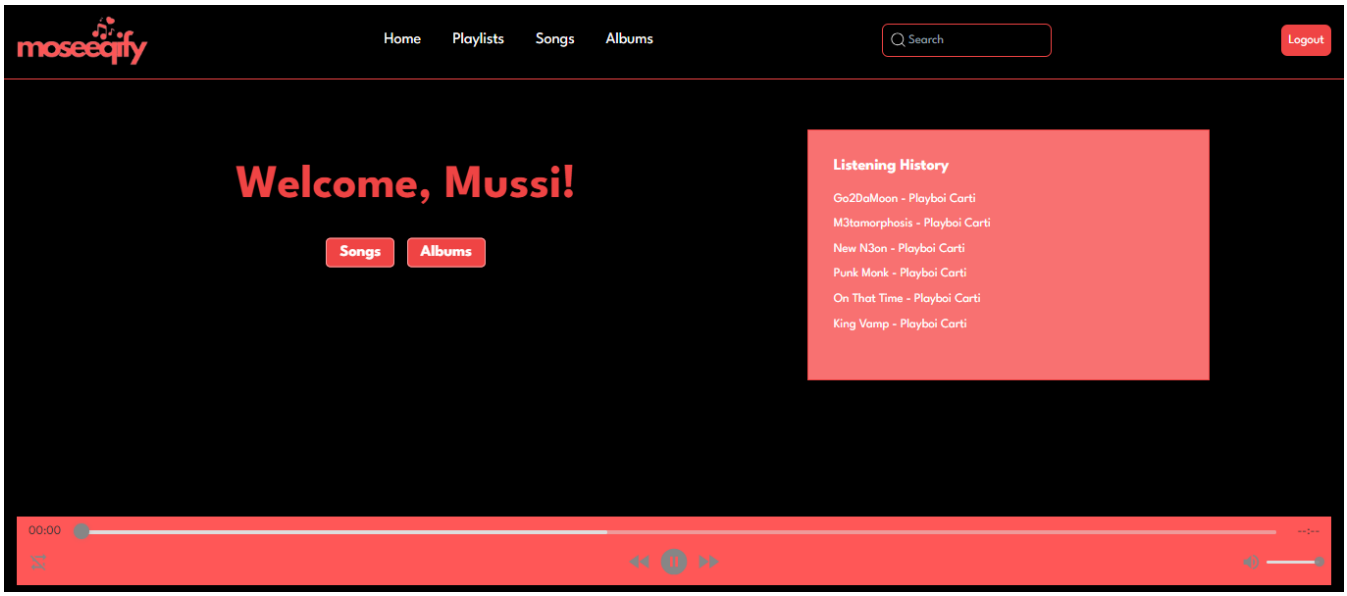
Frontend Screenshots



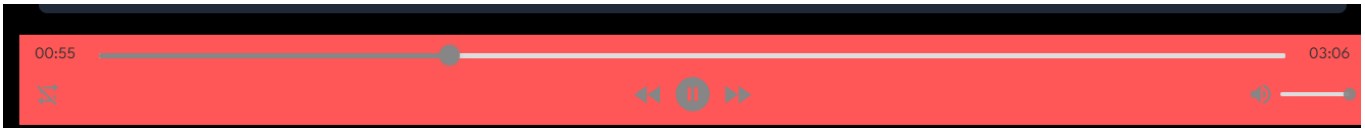
Login Page



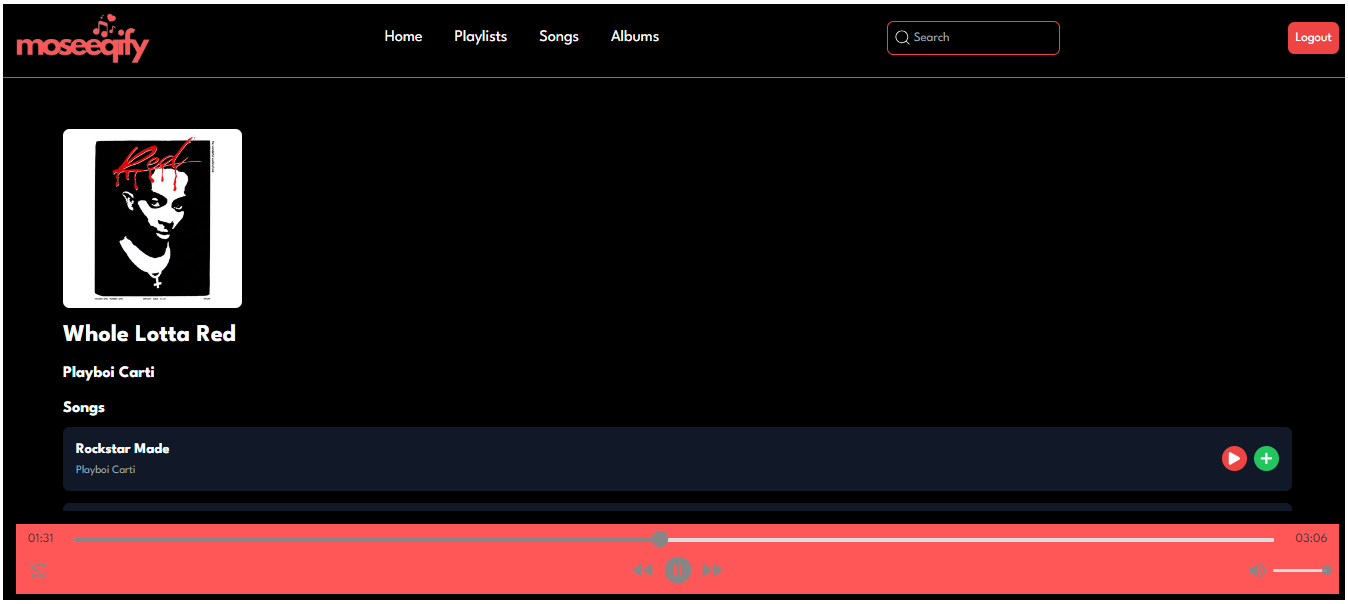
Signup Page



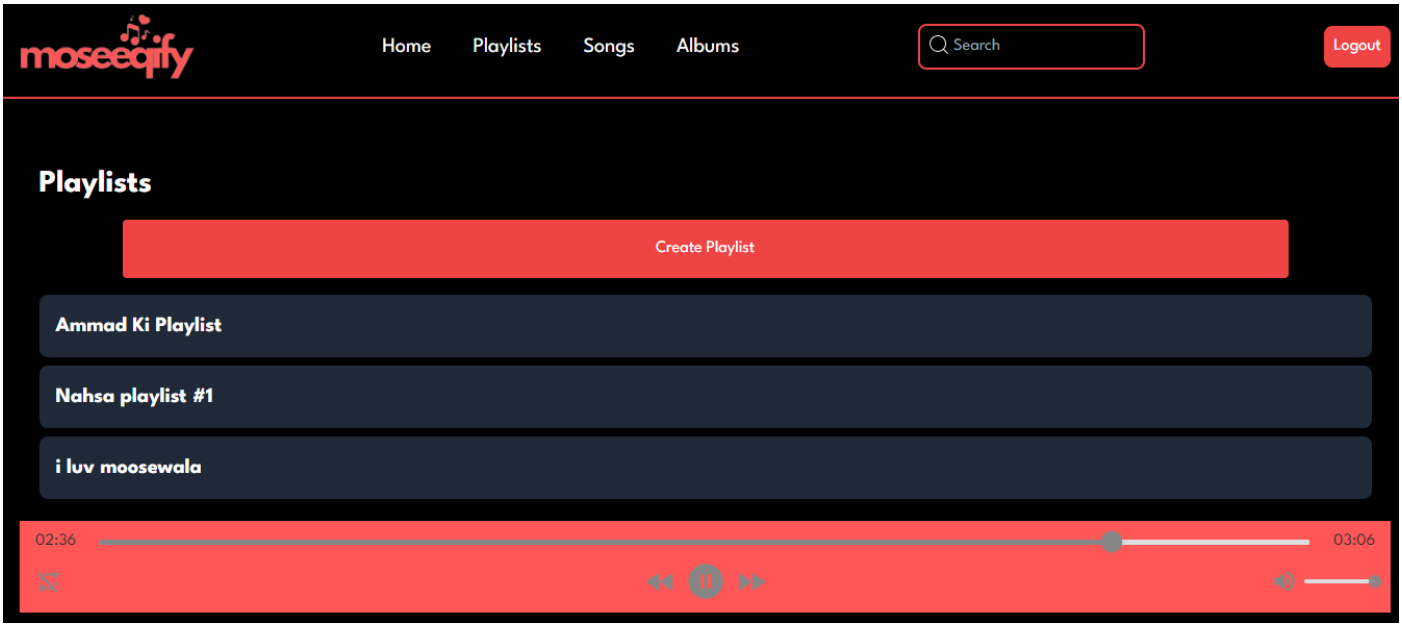
Home Page



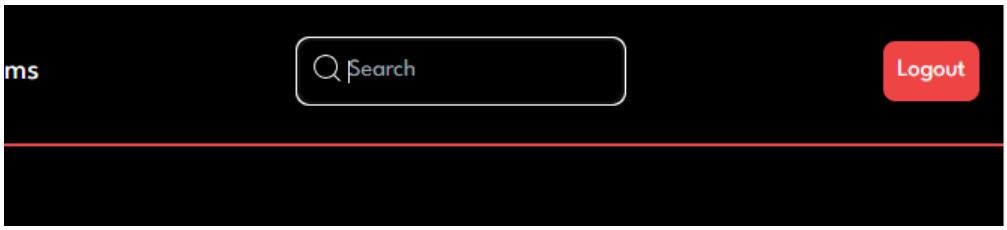
Player Component



Album Page



Playlist Page



Search Component

Conclusion

Moseeqify aims to deliver a feature-rich music streaming experience by leveraging a robust and scalable database design. The integration of Microsoft SQL Server, Flask API, and React JS frontend ensures a seamless user experience while maintaining high performance, security, and scalability. The detailed documentation of the database schema, API endpoints, user requirements, and security considerations serves as a comprehensive guide for understanding and extending the system.

References

- **Flask:** *https://flask.palletsprojects.com/*
- **Flask-Login:** *https://flask-login.readthedocs.io/*
- **Flask-CORS:** *https://flask-cors.readthedocs.io/*
- **SQLAlchemy:** *https://docs.sqlalchemy.org/*
- **React:** *https://reactjs.org/docs/getting-started.html*
- **Microsoft SQL Server:** *https://docs.microsoft.com/sql/sql-server/*
- **GitHub:** *https://docs.github.com/*

Appendix

Complete Code

Database Schema

Tables:

Create the Database:

```
CREATE DATABASE moseeqify;
USE moseeqify;
```

User:

```
CREATE TABLE [User] (
   username VARCHAR(50) PRIMARY KEY,
   email VARCHAR(100) UNIQUE NOT NULL, -- Added UNIQUE constraint
   name VARCHAR(100) NOT NULL,
   password VARCHAR(100) NOT NULL,
   dob DATE NOT NULL,
   dateJoined DATETIME DEFAULT CURRENT_TIMESTAMP,
   CONSTRAINT CHK_User_Age CHECK (DATEDIFF(YEAR, dob, GETDATE()) >= 18)
);
```

Artists:

```
CREATE TABLE Artist (
   artistID INT PRIMARY KEY IDENTITY(1,1),
   name VARCHAR(100) UNIQUE NOT NULL, -- Added UNIQUE constraint
   bio TEXT,
   profilepiclink VARCHAR(255)
);
```

Genre:

```
CREATE TABLE Genre (
   genreName VARCHAR(50) PRIMARY KEY
);
```

Album:

```
CREATE TABLE Album (
   albumID INT PRIMARY KEY IDENTITY(1,1),
   name VARCHAR(100) NOT NULL,
   artistID INT NOT NULL REFERENCES Artist(artistID), -- Added NOT NULL constraint
   releasedate DATETIME,
   coverimagelink VARCHAR(255) NOT NULL -- Added NOT NULL constraint
);
```

Song:

```
CREATE TABLE Song (
   songID INT PRIMARY KEY IDENTITY(1,1),
   title VARCHAR(100) NOT NULL,
   artistID INT NOT NULL REFERENCES Artist(artistID),
   albumID INT REFERENCES Album(albumID),
   genreName VARCHAR(50) REFERENCES Genre(genreName),
   duration TIME NOT NULL, -- Added NOT NULL constraint
   audiolink VARCHAR(500) NOT NULL,
   releaseDate DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

AlbumSongs:

```
CREATE TABLE AlbumSongs (
   albumID INT REFERENCES Album(albumID) ,
   songID INT REFERENCES Song(songID),
   PRIMARY KEY (albumID, songID)
);
```

Playlist:

```
CREATE TABLE Playlist (
   playlistID INT PRIMARY KEY IDENTITY(1,1),
   name VARCHAR(100) NOT NULL, -- Added NOT NULL constraint
   creationdate DATETIME DEFAULT CURRENT_TIMESTAMP,
   username VARCHAR(50) NOT NULL REFERENCES [User](username)
);
```

playlistSongs:

```
CREATE TABLE PlaylistSongs (
   playlistID INT REFERENCES Playlist(playlistID),
   songID INT REFERENCES Song(songID),
   PRIMARY KEY (playlistID, songID)
);
```

UserListeningHistory:

```
CREATE TABLE UserListeningHistory (
   username VARCHAR(50) NOT NULL REFERENCES [User](username),
   songID INT NOT NULL REFERENCES Song(songID),
   listeningDate DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

user_follows_artists:

```
   CREATE TABLE user_follows_artists (
      username VARCHAR(50) NOT NULL,
      artistID INT NOT NULL,
      PRIMARY KEY (username, artistID),
      FOREIGN KEY (username) REFERENCES [User](username),
      FOREIGN KEY (artistID) REFERENCES Artist(artistID)
   );
```

Triggers:

Insert trigger (User):

```
CREATE TRIGGER trgAfterInsertUser ON [User]
AFTER INSERT
AS
BEGIN
   DECLARE @username VARCHAR(50), @name VARCHAR(100), @email VARCHAR(100),@password VARCHAR(100), @dob DATE, @dateJoined DATETIME;

   SELECT @username = INSERTED.username, @name = INSERTED.name, @dob = INSERTED.dob, @dateJoined
        = INSERTED.dateJoined, @email = INSERTED.email,@password = INSERTED.password
   FROM INSERTED;

   PRINT 'New user inserted: Username = ' + @username + ', Name = ' + @name + ', Email = ' + @email + ', password = '+@password +', Date of Birth = ' +
CONVERT(VARCHAR, @dob, 120) + ', Date Joined = ' + CONVERT(VARCHAR, @dateJoined, 120);
END;
```

Update trigger (User):

```
CREATE TRIGGER trgAfterUpdateUser ON [User]
AFTER UPDATE
AS
BEGIN
   DECLARE @username VARCHAR(50), @oldEmail VARCHAR(100), @newEmail VARCHAR(100), @oldName VARCHAR(100), @newName VARCHAR(100), @oldDob
DATE, @newDob DATE, @oldDateJoined DATETIME, @newDateJoined DATETIME;

   SELECT @username = INSERTED.username,
        @newEmail = INSERTED.email, @oldEmail = DELETED.email,
        @newName = INSERTED.name, @oldName = DELETED.name,
        @newDob = INSERTED.dob, @oldDob = DELETED.dob,
        @newDateJoined = INSERTED.dateJoined, @oldDateJoined = DELETED.dateJoinedFROM
   INSERTED
   INNER JOIN DELETED ON INSERTED.username = DELETED.username;

   PRINT 'the record of ' + @username +' is updated'+
        ', Old Email = ' + @oldEmail + ', New Email = ' + @newEmail +
        ', Old Name = ' + @oldName + ', New Name = ' + @newName +
        ', Old Date of Birth = ' + CONVERT(VARCHAR, @oldDob, 120) + ', New Date of Birth = ' + CONVERT(VARCHAR, @newDob, 120) +
        ', Old Date Joined = ' + CONVERT(VARCHAR, @oldDateJoined, 120) + ', New Date Joined = ' + CONVERT(VARCHAR, @newDateJoined, 120);
END;
```

Delete trigger (User):

```
CREATE TRIGGER trgAfterDeleteUser ON [User]
AFTER DELETE
AS
```

```
    BEGIN
    PRINT 'A user record has been deleted.';
END;
```

Insert trigger (Artist):

```
CREATE TRIGGER trgAfterInsertArtist ON Artist
AFTER INSERT
AS
BEGIN
DECLARE @artistID INT, @name VARCHAR(100), @bio VARCHAR(MAX), @profilepiclink VARCHAR(255); SELECT
    @artistID = INSERTED.artistID,
        @name = INSERTED.name,
        @bio = INSERTED.bio,
        @profilepiclink = INSERTED.profilepiclink
    FROM INSERTED;

    PRINT 'New artist inserted: ArtistID = ' + CAST(@artistID AS nVARCHAR(10)) +',
        Name = ' + @name +
        ', Bio = ' + ISNULL(@bio, 'NULL') +
        ', Profile Pic Link = ' + ISNULL(@profilepiclink, 'NULL');
END;
```

Update trigger (Artist):

```
CREATE TRIGGER trgAfterUpdateArtist ON Artist
AFTER UPDATE
AS
BEGIN
    DECLARE @artistID INT, @oldName VARCHAR(100), @newName VARCHAR(100), @oldBio VARCHAR(MAX), @newBio VARCHAR(MAX), @oldProfilepiclink
VARCHAR(255), @newProfilepiclink VARCHAR(255);

    SELECT @artistID = INSERTED.artistID,
        @newName = INSERTED.name, @oldName = DELETED.name,
        @newBio = INSERTED.bio, @oldBio = DELETED.bio,
        @newProfilepiclink = INSERTED.profilepiclink, @oldProfilepiclink = DELETED.profilepiclink
    FROM INSERTED
    INNER JOIN DELETED ON INSERTED.artistID = DELETED.artistID;

    PRINT 'Artist updated: ArtistID = ' + CAST(@artistID AS VARCHAR(10)) +',
        Old Name = ' + @oldName + ', New Name = ' + @newName +
        ', Old Bio = ' + ISNULL(@oldBio, 'NULL') + ', New Bio = ' + ISNULL(@newBio, 'NULL') +
        ', Old Profile Pic Link = ' + ISNULL(@oldProfilepiclink, 'NULL') + ', New Profile Pic Link = ' + ISNULL(@newProfilepiclink, 'NULL');
END;
```

Delete trigger (Artist):

```
CREATE TRIGGER trgAfterDeleteArtist ON Artist
AFTER DELETE
AS
BEGIN
    PRINT 'An artist record has been deleted.';
END;
```

Insert trigger (Genre):

```
CREATE TRIGGER trgAfterInsertGenre ON Genre
AFTER INSERT
AS
BEGIN
    DECLARE @genreName VARCHAR(50);

    SELECT @genreName = INSERTED.genreName
    FROM INSERTED;

    PRINT 'New genre inserted: Genre Name = ' + @genreName;
END;
```

Update trigger (Genre):

```
CREATE TRIGGER trgAfterUpdateGenre ON Genre
AFTER UPDATE
AS
BEGIN
```

```
  DECLARE @oldGenreName VARCHAR(50), @newGenreName VARCHAR(50);

  SELECT @oldGenreName = DELETED.genreName, @newGenreName = INSERTED.genreName
  FROM INSERTED, DELETED;

  PRINT 'Genre updated: Old Genre Name = ' + @oldGenreName + ', New Genre Name = ' + @newGenreName;END;
```

Delete trigger (Genre):

```
CREATE TRIGGER trgAfterDeleteGenre ON Genre
AFTER DELETE
AS
BEGIN
  PRINT 'A genre record has been deleted.';
END;
```

Insert trigger (Album):

```
CREATE TRIGGER trgAfterInsertAlbum ON Album
AFTER INSERT
AS
BEGIN
  DECLARE @albumID INT, @name VARCHAR(100), @artistID INT, @releasedate DATETIME, @coverimagelink VARCHAR(255);

  SELECT @albumID = INSERTED.albumID,
      @name = INSERTED.name,
      @artistID = INSERTED.artistID,
      @releasedate = INSERTED.releasedate,
      @coverimagelink = INSERTED.coverimagelink
  FROM INSERTED;

  PRINT 'New album inserted: AlbumID = ' + CAST(@albumID AS VARCHAR(10)) +',
      Name = ' + @name +
      ', ArtistID = ' + CAST(@artistID AS VARCHAR(10)) +
      ', Release Date = ' + ISNULL(CONVERT(VARCHAR, @releasedate, 120), 'NULL') +',
      Cover Image Link = ' + @coverimagelink;
END;
```

Update trigger (Album):

```
CREATE TRIGGER trgAfterUpdateAlbum ON Album
AFTER UPDATE
AS
BEGIN
  DECLARE @albumID INT, @oldName VARCHAR(100), @newName VARCHAR(100), @oldArtistID INT, @newArtistID INT, @oldReleaseDate DATETIME,
@newReleaseDate DATETIME, @oldCoverImageLink VARCHAR(255), @newCoverImageLink VARCHAR(255);

  SELECT @albumID = INSERTED.albumID,
      @newName = INSERTED.name, @oldName = DELETED.name,
      @newArtistID = INSERTED.artistID, @oldArtistID = DELETED.artistID,
      @newReleaseDate = INSERTED.releasedate, @oldReleaseDate = DELETED.releasedate,
      @newCoverImageLink = INSERTED.coverimagelink, @oldCoverImageLink = DELETED.coverimagelinkFROM
  INSERTED
  INNER JOIN DELETED ON INSERTED.albumID = DELETED.albumID;

  PRINT 'Album updated: AlbumID = ' + CAST(@albumID AS VARCHAR(10)) +',
      Old Name = ' + @oldName + ', New Name = ' + @newName +
      ', Old ArtistID = ' + CAST(@oldArtistID AS VARCHAR(10)) + ', New ArtistID = ' + CAST(@newArtistID AS VARCHAR(10)) +
      ', Old Release Date = ' + ISNULL(CONVERT(VARCHAR, @oldReleaseDate, 120), 'NULL') + ', New Release Date = ' + ISNULL(CONVERT(VARCHAR, @newReleaseDate,
120), 'NULL') +
      ', Old Cover Image Link = ' + @oldCoverImageLink + ', New Cover Image Link = ' + @newCoverImageLink;
END;
```

Delete trigger (Album):

```
CREATE TRIGGER trgAfterDeleteAlbum ON Album
AFTER DELETE
AS
BEGIN
  PRINT 'An album record has been deleted.';
END;
```

Insert trigger (Song):

```
CREATE TRIGGER trgAfterInsertSong ON Song
AFTER INSERT
AS
BEGIN
   DECLARE @songID INT, @title VARCHAR(100), @artistID INT, @albumID INT, @genreName VARCHAR(50), @duration TIME, @audiolink VARCHAR(255),
@releaseDate DATETIME;

   SELECT @songID = INSERTED.songID,
        @title = INSERTED.title,
        @artistID = INSERTED.artistID,
        @albumID = INSERTED.albumID,
        @genreName = INSERTED.genreName,
        @duration = INSERTED.duration,
        @audiolink = INSERTED.audiolink,
        @releaseDate = INSERTED.releaseDate
   FROM INSERTED;

   PRINT 'New song inserted: SongID = ' + CAST(@songID AS VARCHAR(10)) +',
        Title = ' + @title +
        ', ArtistID = ' + CAST(@artistID AS VARCHAR(10)) +
        ', AlbumID = ' + ISNULL(CAST(@albumID AS VARCHAR(10)), 'NULL') +
        ', Genre = ' + ISNULL(@genreName, 'NULL') +
        ', Duration = ' + CONVERT(VARCHAR, @duration, 120) +',
        Audio Link = ' + @audiolink +
        ', Release Date = ' + CONVERT(VARCHAR, @releaseDate, 120);
END;
```

update trigger (Song):

```
CREATE TRIGGER trgAfterUpdateSong ON Song
AFTER UPDATE
AS
BEGIN
   DECLARE @songID INT, @oldTitle VARCHAR(100), @newTitle VARCHAR(100), @oldArtistID INT, @newArtistID INT, @oldAlbumID INT, @newAlbumID INT,
@oldGenreName VARCHAR(50), @newGenreName VARCHAR(50), @oldDuration TIME, @newDuration TIME, @oldAudiolink VARCHAR(255), @newAudiolink
VARCHAR(255), @oldReleaseDate DATETIME, @newReleaseDate DATETIME;

   SELECT @songID = INSERTED.songID,
        @newTitle = INSERTED.title, @oldTitle = DELETED.title, @newArtistID
        = INSERTED.artistID, @oldArtistID = DELETED.artistID,
        @newAlbumID = INSERTED.albumID, @oldAlbumID = DELETED.albumID,
        @newGenreName = INSERTED.genreName, @oldGenreName = DELETED.genreName,
        @newDuration = INSERTED.duration, @oldDuration = DELETED.duration,
        @newAudiolink = INSERTED.audiolink, @oldAudiolink = DELETED.audiolink,
        @newReleaseDate = INSERTED.releaseDate, @oldReleaseDate = DELETED.releaseDate
   FROM INSERTED,deleted
   -- INNER JOIN DELETED ON INSERTED.songID = DELETED.songID;

   PRINT 'Song updated: SongID = ' + CAST(@songID AS VARCHAR(10)) +',
        Old Title = ' + @oldTitle + ', New Title = ' + @newTitle +
        ', Old ArtistID = ' + CAST(@oldArtistID AS VARCHAR(10)) + ', New ArtistID = ' + CAST(@newArtistID AS VARCHAR(10)) +
        ', Old AlbumID = ' + ISNULL(CAST(@oldAlbumID AS VARCHAR(10)), 'NULL') + ', New AlbumID = ' + ISNULL(CAST(@newAlbumID AS VARCHAR(10)), 'NULL') +
        ', Old Genre = ' + ISNULL(@oldGenreName, 'NULL') + ', New Genre = ' + ISNULL(@newGenreName, 'NULL') +
        ', Old Duration = ' + CONVERT(VARCHAR, @oldDuration, 120) + ', New Duration = ' + CONVERT(VARCHAR, @newDuration, 120) + ',
        Old Audio Link = ' + @oldAudiolink + ', New Audio Link = ' + @newAudiolink +
        ', Old Release Date = ' + CONVERT(VARCHAR, @oldReleaseDate, 120) + ', New Release Date = ' + CONVERT(VARCHAR, @newReleaseDate, 120);
END;
```

Delete trigger (Song):

```
CREATE TRIGGER trgAfterDeleteSong ON Song
AFTER DELETE
AS
BEGIN
   PRINT 'A song record has been deleted.';
END;
```

Insert trigger (AlbumSong):

```
CREATE TRIGGER trgAfterInsertAlbumSongs ON AlbumSongs
AFTER INSERT
AS
BEGIN
   DECLARE @albumID INT, @songID INT;
```

```
  SELECT @albumID = albumID, @songID = songID
  FROM INSERTED;

  PRINT 'New AlbumSong inserted: AlbumID = ' + CAST(@albumID AS VARCHAR(10)) +',
      SongID = ' + CAST(@songID AS VARCHAR(10));
END;
```

Update trigger (AlbumSong):

```
CREATE TRIGGER trgAfterUpdateAlbumSongs ON AlbumSongs
AFTER UPDATE
AS
BEGIN
  DECLARE @oldAlbumID INT, @newAlbumID INT, @oldSongID INT, @newSongID INT;

  SELECT @oldAlbumID = DELETED.albumID, @newAlbumID = INSERTED.albumID,@oldSongID
      = DELETED.songID, @newSongID = INSERTED.songID
  FROM INSERTED,DELETED

  PRINT 'AlbumSong updated: Old AlbumID = ' + CAST(@oldAlbumID AS VARCHAR(10)) + ', New AlbumID = ' + CAST(@newAlbumID AS VARCHAR(10)) + ',
      Old SongID = ' + CAST(@oldSongID AS VARCHAR(10)) + ', New SongID = ' + CAST(@newSongID AS VARCHAR(10));
END;
```

Delete trigger (AlbumSong):

```
CREATE TRIGGER trgAfterDeleteAlbumSongs ON AlbumSongs
AFTER DELETE
AS
BEGIN
  PRINT 'An album-songs record has been deleted.';
END;
```

Insert trigger (Playlist):

```
CREATE TRIGGER trgAfterInsertPlaylist ON Playlist
AFTER INSERT
AS
BEGIN
  DECLARE @playlistID INT, @name VARCHAR(100), @creationdate DATETIME, @username VARCHAR(50);

  SELECT @playlistID = playlistID,
      @name = name,
      @creationdate = creationdate,
      @username = username
  FROM INSERTED;

  PRINT 'New Playlist inserted: PlaylistID = ' + CAST(@playlistID AS VARCHAR(10)) +',
      Name = ' + @name +
      ', Creation Date = ' + CONVERT(VARCHAR, @creationdate, 120) +',
      Username = ' + @username;
END;
```

Update trigger (Playlist):

```
CREATE TRIGGER trgAfterUpdatePlaylist ON Playlist
AFTER UPDATE
AS
BEGIN
  DECLARE @oldPlaylistID INT, @newPlaylistID INT,
      @oldName VARCHAR(100), @newName VARCHAR(100),
      @oldCreationdate DATETIME, @newCreationdate DATETIME,
      @oldUsername VARCHAR(50), @newUsername VARCHAR(50);

  SELECT @oldPlaylistID = DELETED.playlistID, @newPlaylistID = INSERTED.playlistID,
      @oldName = DELETED.name, @newName = INSERTED.name,
      @oldCreationdate = DELETED.creationdate, @newCreationdate = INSERTED.creationdate,
      @oldUsername = DELETED.username, @newUsername = INSERTED.username
  FROM INSERTED,DELETED

  PRINT 'Playlist updated: Old PlaylistID = ' + CAST(@oldPlaylistID AS VARCHAR(10)) + ', New PlaylistID = ' + CAST(@newPlaylistID AS VARCHAR(10)) + ',
      Old Name = ' + @oldName + ', New Name = ' + @newName +
      ', Old Creation Date = ' + CONVERT(VARCHAR, @oldCreationdate, 120) + ', New Creation Date = ' + CONVERT(VARCHAR, @newCreationdate, 120) + ',
      Old Username = ' + @oldUsername + ', New Username = ' + @newUsername;
END;
```

Delete trigger (Playlist):

```
CREATE TRIGGER trgAfterDeletePlaylist ON Playlist
```

```
AFTER DELETE
AS
BEGIN
  PRINT 'A playlist record has been deleted.';
END;
```

Insert trigger (PlaylistSongs):

```
  CREATE TRIGGER trgAfterInsertPlaylistSongs ON PlaylistSongs
  AFTER INSERT
  AS
  BEGIN
    DECLARE @playlistID INT, @songID INT;

    SELECT @playlistID = playlistID, @songID = songID
    FROM INSERTED;

    PRINT 'New PlaylistSong inserted: PlaylistID = ' + CAST(@playlistID AS VARCHAR(10)) +',
      SongID = ' + CAST(@songID AS VARCHAR(10));
  END;
```

Update trigger (PlaylistSongs):

```
CREATE TRIGGER trgAfterUpdatePlaylistSongs ON PlaylistSongs
AFTER UPDATE
AS
BEGIN
  DECLARE @oldPlaylistID INT, @newPlaylistID INT, @oldSongID INT, @newSongID INT;

  SELECT @oldPlaylistID = DELETED.playlistID, @newPlaylistID = INSERTED.playlistID,
      @oldSongID = DELETED.songID, @newSongID = INSERTED.songID
  FROM INSERTED,DELETED
  PRINT 'PlaylistSong updated: Old PlaylistID = ' + CAST(@oldPlaylistID AS VARCHAR(10)) + ', New PlaylistID = ' + CAST(@newPlaylistID AS VARCHAR(10)) + ',
     Old SongID = ' + CAST(@oldSongID AS VARCHAR(10)) + ', New SongID = ' + CAST(@newSongID AS VARCHAR(10));
END;
```

Delete trigger (PlaylistSongs):

```
CREATE TRIGGER trgAfterDeletePlaylistSongs ON PlaylistSongs
AFTER DELETE
AS
BEGIN
  PRINT 'A playlist-songs record has been deleted.';
END;
```

Insert trigger (User_follows_artists):

```
CREATE TRIGGER trgAfterInsertUserFollowsArtists ON user_follows_artists
AFTER INSERT
AS
BEGIN
  DECLARE @username VARCHAR(50), @artistID INT;

  SELECT @username = username, @artistID = artistID
  FROM INSERTED;

  PRINT 'User ' + @username + ' started following Artist ' + CAST(@artistID AS VARCHAR(10));END;
```

update trigger (user_follows_artists):

```
CREATE TRIGGER trgAfterUpdateUserFollowsArtists ON user_follows_artists
AFTER UPDATE
AS
BEGIN
  DECLARE @username VARCHAR(50), @artistID INT; DECLARE
  @oldUsername VARCHAR(50), @oldArtistID INT; DECLARE
  @newUsername VARCHAR(50), @newArtistID INT;

  SELECT @oldUsername = DELETED.username, @oldArtistID = DELETED.artistID,
      @newUsername = INSERTED.username, @newArtistID = INSERTED.artistID
  FROM INSERTED,deleted
  PRINT 'User ' + @oldUsername + ' changed following status for Artist ' + CAST(@oldArtistID AS VARCHAR(10)) + '
     to User ' + @newUsername + ' following status for Artist ' + CAST(@newArtistID AS VARCHAR(10));
END;
```

delete trigger (User_follows_artists):

```
CREATE TRIGGER trgAfterDeleteUserFollowsArtists ON user_follows_artists
```

```
AFTER DELETE
AS
BEGIN
   PRINT 'A user follows artists record has been deleted.';END;
```

Stored Procedures:

insert_in_User:

```
create procedure insert_in_User
   @username VARCHAR(50), @name VARCHAR(100)
   ,@password VARCHAR(100)
   ,@email VARCHAR(100)
   , @dob DATE, @dateJoined DATETIME
AS
BEGIN
   insert into [User](username,email,name,[password],dob,dateJoined)
   values
   (@username,@email,@name,@password,@dob,@dateJoined);
END
GO
```

insert_in_Artist:

```
create procedure insert_in_Artist
   @name VARCHAR(100), @bio VARCHAR(MAX), @profilepiclink VARCHAR(255)
AS
BEGIN
   insert into Artist
   values
   (@name,@bio,@profilepiclink);
END
GO
```

insert_in_Genre:

```
create procedure insert_in_Genre
    @genreName VARCHAR(50)
AS
BEGIN
   insert into Genre
   values
   (@genreName);
END
GO
```

insert_in_album:

```
create procedure insert_in_album
   @name VARCHAR(100), @artistID INT, @releasedate DATETIME, @coverimagelink VARCHAR(255)
AS
BEGIN
   insert into Album
   values
   (@name,@artistID,@releasedate,@coverimagelink);
END
GO
```

insertSong:

```
CREATE PROCEDURE InsertSong
   @title VARCHAR(100),
   @artistID INT,
   @albumID INT = NULL,
   @genreName VARCHAR(50) = NULL,
   @duration TIME,
   @audiolink VARCHAR(255),
   @releaseDate DATETIME = NULL
AS
BEGIN
   SET NOCOUNT ON;

   IF @releaseDate IS NULL
      SET @releaseDate = CURRENT_TIMESTAMP;

   INSERT INTO Song (title, artistID, albumID, genreName, duration, audiolink, releaseDate)
   VALUES (@title, @artistID, @albumID, @genreName, @duration, @audiolink, @releaseDate);
END;
```

insertAlbumSong:

```
CREATE PROCEDURE InsertAlbumSong
   @albumID INT,
   @songID INT
AS
BEGIN
   INSERT INTO AlbumSongs (albumID, songID)
   VALUES (@albumID, @songID);
END;
```

insertPlaylist:

```
CREATE PROCEDURE InsertPlaylist
   @name VARCHAR(100),
   @username VARCHAR(50)
AS
BEGIN
   INSERT INTO Playlist (name, creationdate, username)
   VALUES (@name, CURRENT_TIMESTAMP, @username);
END;
```

insertPlaylistSong:

```
CREATE PROCEDURE InsertPlaylistSong
   @playlistID INT,
   @songID INT
AS
BEGIN
   INSERT INTO PlaylistSongs (playlistID, songID)
   VALUES (@playlistID, @songID);
END;
```

insertUserListeningHistory:

```
CREATE PROCEDURE InsertUserListeningHistory
   @username VARCHAR(50),
   @songID INT
AS
BEGIN
   INSERT INTO UserListeningHistory (username, songID, listeningDate)
   VALUES (@username, @songID, CURRENT_TIMESTAMP);
END;
```

InsertUserFollowsArtist:

```
CREATE PROCEDURE InsertUserFollowsArtist
   @username VARCHAR(50),
   @artistID INT
AS
BEGIN
   INSERT INTO user_follows_artists (username, artistID)
   VALUES (@username, @artistID);
END;
```

get_table_content:

```
CREATE PROCEDURE get_table_content
   @TableName NVARCHAR(128)
AS
BEGIN
   DECLARE @Exec_STAT NVARCHAR(MAX)
   SET @Exec_STAT = N'SELECT * FROM ' + QUOTENAME(@TableName)
   EXEC sp_executesql @Exec_STAT
END
GO

create procedure update_in_User
    @username VARCHAR(50)
   ,@name VARCHAR(100),@password VARCHAR(100)
   ,@email VARCHAR(100), @dob DATE
   ,@dateJoined DATETIME
AS
BEGIN
update [USER]
set name = @name,[password]=@password,email = @email, dob=@dob,dateJoined=@dateJoined
   where username=@username
END
GO
```

update_in_Artist:

```
create procedure update_in_Artist
   @artistID INT, @name VARCHAR(100), @bio VARCHAR(MAX), @profilepiclink VARCHAR(255)
AS
BEGIN
update Artist
set name=@name,bio = @bio, profilepiclink=@profilepiclink
   where artistID=@artistID
END
GO
```

update_in_Genre:

```
create procedure update_in_Genre
   @old varchar(50),
   @genreName VARCHAR(50)
AS
BEGIN
update Genre
set genreName=@genreName
   where genreName=@old
END
GO
create procedure update_in_album
   @albumID INT, @name VARCHAR(100), @artistID INT, @releasedate DATETIME, @coverimagelink VARCHAR(255)
AS
BEGIN
update Album
set name=@name,artistID=@artistID,releasedate=@releasedate,coverimagelink=@coverimagelink
   where albumID=@albumID
END
GO
```

UpdateSong:

```
CREATE PROCEDURE UpdateSong
   @songID INT,
   @title VARCHAR(100),
   @artistID INT,
   @albumID INT = NULL,
   @genreName VARCHAR(50) = NULL,
   @duration TIME,
   @audiolink VARCHAR(255),
   @releaseDate DATETIME = NULL
AS
BEGIN
   UPDATE Song
   SET title = @title,
      artistID = @artistID,
      albumID = @albumID,
      genreName = @genreName,
      duration = @duration,
      audiolink = @audiolink,
      releaseDate = ISNULL(@releaseDate, CURRENT_TIMESTAMP)
   WHERE songID = @songID;
END;
```

UpdateAlbumSong:

```
CREATE PROCEDURE UpdateAlbumSong
   @oldAlbumID INT,
   @oldSongID INT,
   @newAlbumID INT,
   @newSongID INT
AS
BEGIN
   -- Delete the old record
   DELETE FROM AlbumSongs
   WHERE albumID = @oldAlbumID AND songID = @oldSongID;
   -- Insert the new record
   INSERT INTO AlbumSongs (albumID, songID)
   VALUES (@newAlbumID, @newSongID);
END;
```

UpdatePlaylist:

```
CREATE PROCEDURE UpdatePlaylist
   @playlistID INT,
   @name VARCHAR(100),
   @username VARCHAR(50)
AS
BEGIN
   UPDATE Playlist
   SET name = @name,
      username = @username
   WHERE playlistID = @playlistID;
END;
```

UpdatePlaylistSong:

```
CREATE PROCEDURE UpdatePlaylistSong
   @oldPlaylistID INT,
   @oldSongID INT,
   @newPlaylistID INT,
   @newSongID INT
AS
BEGIN
   -- Delete the old record
   DELETE FROM PlaylistSongs
   WHERE playlistID = @oldPlaylistID AND songID = @oldSongID;
   -- Insert the new record
   INSERT INTO PlaylistSongs (playlistID, songID)
   VALUES (@newPlaylistID, @newSongID);
END;
```

UpdateUserListeningHistory:

```
CREATE PROCEDURE UpdateUserListeningHistory
   @oldUsername VARCHAR(50),
   @oldSongID INT,
   @oldListeningDate DATETIME,
   @newUsername VARCHAR(50),
   @newSongID INT
AS
BEGIN
   -- Delete the old record
   DELETE FROM UserListeningHistory
   WHERE username = @oldUsername AND songID = @oldSongID AND listeningDate = @oldListeningDate;
   -- Insert the new record
   INSERT INTO UserListeningHistory (username, songID, listeningDate)
   VALUES (@newUsername, @newSongID, CURRENT_TIMESTAMP);
END;
```

UpdateUserFollowsArtist:

```
CREATE PROCEDURE UpdateUserFollowsArtist
   @oldUsername VARCHAR(50),
   @oldArtistID INT,
   @newUsername VARCHAR(50),
   @newArtistID INT
AS
BEGIN
   -- Delete the old record
   DELETE FROM user_follows_artists
   WHERE username = @oldUsername AND artistID = @oldArtistID;

   -- Insert the new record
   INSERT INTO user_follows_artists (username, artistID)VALUES (@newUsername, @newArtistID);
END;
   b.ForeignKey('artist.artistID'), primary_key=True)
```

delete_from_user:

```
create PROCEDURE delete_from_user
  @username varchar(50)
AS
BEGIN
  delete from [User]
  where username=@username;
END
GO
```

delete_from_Artist:

```
create PROCEDURE delete_from_Artist
  @ArtistID int
AS
BEGIN
  delete from Artist
  where artistID=@ArtistID;
END
GO
```

delete_from_Genre:

```
create PROCEDURE delete_from_Genre
  @genreName VARCHAR(50)
AS
BEGIN
  delete from Genre
  where genreName=@genreName;
END
GO
```

delete_from_album:

```
create PROCEDURE delete_from_album
  @albumID INT
AS
BEGIN
  delete from Album
  where albumID=@albumID;
END
GO
CREATE PROCEDURE DeleteSong
  @songID INT
AS
BEGIN
  DELETE FROM Song
  WHERE songID = @songID;
END;
```

DeleteAlbumSong:

```
CREATE PROCEDURE DeleteAlbumSong
  @albumID INT,
  @songID INT
AS
BEGIN
  DELETE FROM AlbumSongs
  WHERE albumID = @albumID AND songID = @songID;
END;
```

DeletePlaylist:

```
CREATE PROCEDURE DeletePlaylist
  @playlistID INT
AS
BEGIN
  DELETE FROM Playlist
  WHERE playlistID = @playlistID;
END;
```

DeletePlaylistSong:

```
CREATE PROCEDURE DeletePlaylistSong
  @playlistID INT,
  @songID INT
AS
BEGIN
  DELETE FROM PlaylistSongs
  WHERE playlistID = @playlistID AND songID = @songID;
END;
```

DeleteUserListeningHistory:

```
CREATE PROCEDURE DeleteUserListeningHistory
  @username VARCHAR(50),
  @songID INT,
  @listeningDate DATETIME
AS
BEGIN
  DELETE FROM UserListeningHistory
  WHERE username = @username AND songID = @songID AND listeningDate = @listeningDate;
END;
```

DeleteUserFollowsArtist:

```
CREATE PROCEDURE DeleteUserFollowsArtist
  @username VARCHAR(50),
  @artistID INT
AS
BEGIN
  DELETE FROM user_follows_artists
  WHERE username = @username AND artistID = @artistID;
END;
```

Backend API

App.py:

```
from flask import Flask, jsonify, request, session
from flask_sqlalchemy import SQLAlchemy
from flask_cors import CORS, cross_origin
from flask_login import LoginManager, login_user, logout_user, login_required, current_user
from sqlalchemy.exc import IntegrityError
from functools import wraps
from models import User, UserListeningHistory, UserFollowsArtists, Artist, Album, AlbumSongs, PlaylistSongs, Playlist, Song
from models import db
import logging

app = Flask(_name_)
app.config.from_object('config.Config')

db.init_app(app)
login_manager = LoginManager()
login_manager.init_app(app)
CORS(app, resources={r"/*": {"origins": "http://localhost:5173"}})
# Error Handling Middleware
@app.errorhandler(400)
def bad_request(error):
    return jsonify({"error": "Bad request"}), 400

@app.errorhandler(401)
def unauthorized(error):
    return jsonify({"error": "Unauthorized"}), 401

@app.errorhandler(404)
def not_found(error):
    return jsonify({"error": "Not found"}), 404


# Login Required Decorator
def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if not current_user.is_authenticated:
            return jsonify({"message": "Unauthorized"}), 401
        return f(*args, **kwargs)
    return decorated_function

# User Registration
@app.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    email = data.get('email')
    name = data.get('name')
    password = data.get('password')
    dob = data.get('dob')
    if not all([username, email, name, password, dob]):
        return jsonify({"message": "Missing fields"}), 400
    try:

 user = User(username=username, email=email, name=name, password=password, dob=dob)
        db.session.add(user)
        db.session.commit()
        return jsonify({"user": user.serialize(), "message": "User registered successfully"}), 201
    except IntegrityError:
        db.session.rollback()
        return jsonify({"message": "Username or Email already exists"}), 409
# Login
@app.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')
    if not all([email, password]):
        return jsonify({"message": "Missing fields"}), 400
    user = User.query.filter_by(email=email).first()
    if user and user.password == password:
        session['user'] = user.serialize()
```

```
 return jsonify({"user": user.serialize(), "message": "Logged in successfully"}), 200
  return jsonify({"message": "Invalid email or password"}), 401


# User Logout
@app.route('/logout', methods=['GET'])
@login_required
def logout():
  logout_user()
  return jsonify({"message": "Logged out successfully"}), 200

# Play Song
from flask import request


@app.route('/save-to-history/<int:song_id>', methods=['POST'])
def save_to_history(song_id):
  try:
    data = request.get_json()
    username = data.get('username')
    if not username:
      app.logger.error("Username is missing in request payload")
      return jsonify({"error": "Username is missing"}), 400

    # Fetch the song from the database
    song = Song.query.get_or_404(song_id)

    # Log the song ID for debugging
    app.logger.info(f"Received request to save song with ID: {song_id}")

    # Create a new entry in the listening history
    listening_history = UserListeningHistory(username=username, songID=song.songID)
    db.session.add(listening_history)
    db.session.commit()

    # Log success message
app.logger.info("Song saved to listening history successfully")

    return jsonify({"message": "Song saved to listening history successfully"}), 200
  except Exception as e:
    # Log the error message
    app.logger.error(f"Error saving song to listening history: {str(e)}")
    return jsonify({"error": "Internal server error"}), 500

@app.route('/playlists', methods=['POST'])
def create_playlist():
  print(f"Request method: {request.method}")
  data = request.get_json()
  name = data.get('name')
  username = data.get('username')
  if not name:
    return jsonify({"message": "Missing playlist name"}), 400

  playlist = Playlist(name=name, username=username)
  db.session.add(playlist)
  db.session.commit()
  return jsonify(playlist.serialize()), 201


@app.route('/playlists/<int:playlist_id>/add-song/<int:song_id>', methods=['POST'])
def add_song_to_playlist(playlist_id, song_id):
  playlist = Playlist.query.get_or_404(playlist_id)
  song = Song.query.get_or_404(song_id)

  if song not in playlist.songs:
    playlist.songs.append(song)
    db.session.commit()
    return jsonify({"message": "Song added to playlist successfully"}), 200
  else:
    return jsonify({"message": "Song already exists in playlist"}), 409

# Get Specific Album
@app.route('/albums/<int:album_id>/', methods=['GET'])
def get_specific_album(album_id):
  album = Album.query.get_or_404(album_id)
```

```
    if album:
        album_data = {
            "id": album.albumID,
            "name": album.name,
            "release_date": album.releasedate,
            "cover_image_link": album.coverimagelink,
            "artist": album.artist.name,
            "songs": [song.serialize() for song in album.songs]  # Serialize each song
        }
        return jsonify(album_data), 200
    return jsonify({"message": "Song not found in album"}), 404

# Get All Songs
@app.route('/songs', methods=['GET'])
def get_all_songs():

  songs = Song.query.all()

 songs_data = [{

"id": song.songID,
        "title": song.title,
        "artist": song.artist.name,
        "url": song.audiolink,
        "album": song.albumID
    } for song in songs]
    return jsonify(songs_data), 200

# Get All Albums
@app.route('/albums', methods=['GET'])
def get_all_albums():
    albums = Album.query.all()
    albums_data = [{
        "id": album.albumID,
        "name": album.name,
        "release_date": album.releasedate,
        "cover_image_link": album.coverimagelink,
        "artist": album.artist.name,
        "songs": [song.serialize() for song in album.songs]  # Serialize each song
    } for album in albums]
    return jsonify(albums_data), 200

# Manage Playlists
@app.route('/playlists', methods=['GET'])
def get_all_playlists():
    playlists = Playlist.query.all()
    playlists_data = [{
        "id": playlist.playlistID,
        "name": playlist.name,
        "username": playlist.username,
        "songs": [song.serialize() for song in playlist.songs]
    } for playlist in playlists]
    return jsonify(playlists_data), 200

# Get a specific Playlidt
@app.route('/playlists/<int:playlist_id>', methods=['GET'])
def get_playlist(playlist_id):
    playlist = Playlist.query.get_or_404(playlist_id)
    playlist_data = {
        "id": playlist.playlistID,
        "name": playlist.name,
        "username": playlist.username,
        "songs": [song.serialize() for song in playlist.songs]
    }
    return jsonify(playlist_data), 200

# Search Songs
@app.route('/search', methods=['POST'])
def search_songs():
    data = request.get_json()
    query = data.get('query')
    if not query:


 return jsonify({"message": "Missing query"}), 400
```

```
    songs = Song.query.filter(Song.title.ilike(f'%{query}%')).all()
    return jsonify([{"id": song.songID, "title": song.title, "artist": song.artist.name, "audio_link": song.audiolink} for song in songs]), 200
# Follow Artist

@app.route('/follow-artist/<int:artist_id>', methods=['POST'])
@login_required
def follow_artist(artist_id):
    artist = Artist.query.get_or_404(artist_id)
    if artist not in current_user.followed_artists:
        current_user.followed_artists.append(artist)
        db.session.commit()
        return jsonify({"message": "Artist followed successfully"}), 200
    return jsonify({"message": "Already following this artist"}), 409

# Unfollow Artist
@app.route('/unfollow-artist/<int:artist_id>', methods=['POST'])
@login_required
def unfollow_artist(artist_id):
    artist = Artist.query.get_or_404(artist_id)
    if artist in current_user.followed_artists:
        current_user.followed_artists.remove(artist)
        db.session.commit()
        return jsonify({"message": "Artist unfollowed successfully"}), 200
    return jsonify({"message": "Not following this artist"}), 404


# Fetch user listening history
@app.route('/users/<username>/listening-history', methods=['GET'])
def get_listening_history(username):
    print("Fetching listening history for user:", username)  # Add logging
    listening_history = UserListeningHistory.query.filter_by(username=username).all()
    history_data = [{
        "id": item.songID,
        "title": item.song.title,
        "artist": item.song.artist.name
    } for item in listening_history]
    print("Listening history data:", history_data)  # Add logging
    return jsonify(history_data), 200

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(user_id)



if _name_ == '_main_':
app.run(debug=True)
```

Models.py:

```
from flask_sqlalchemy import SQLAlchemy
from flask_login import UserMixin

db = SQLAlchemy()

class User(db.Model, UserMixin):
    username = db.Column(db.String(50), primary_key=True)
    email = db.Column(db.String(100), unique=True, nullable=False)
    name = db.Column(db.String(100), nullable=False)
    password = db.Column(db.String(100), nullable=False)
    dob = db.Column(db.Date, nullable=False)
    dateJoined = db.Column(db.DateTime, default=db.func.current_timestamp())

    def serialize(self):
        return {
            'username': self.username,
            'email': self.email,

  'name': self.name,
            'dob': self.dob,
        }
    def get_id(self):
        return self.username
```

```
class Artist(db.Model):
    artistID = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), unique=True, nullable=False)
    bio = db.Column(db.Text)
    profilepiclink = db.Column(db.String(255))


class Genre(db.Model):
    genreName = db.Column(db.String(50), primary_key=True)

class Album(db.Model):
    albumID = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    releasedate = db.Column(db.DateTime)
    coverimagelink = db.Column(db.String(255), nullable=False)
    artistID = db.Column(db.Integer, db.ForeignKey('artist.artistID'), nullable=False)
    artist = db.relationship('Artist', backref=db.backref('albums', lazy=True))
    songs = db.relationship('Song', backref='album', lazy=True)

class Song(db.Model):
    songID = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    artistID = db.Column(db.Integer, db.ForeignKey('artist.artistID'), nullable=False)
    artist = db.relationship('Artist', backref=db.backref('songs', lazy=True))
    albumID = db.Column(db.Integer, db.ForeignKey('album.albumID'))
    genreName = db.Column(db.String(50), db.ForeignKey('genre.genreName'))
    duration = db.Column(db.Time, nullable=False)

 audiolink = db.Column(db.String(255), nullable=False)
    releaseDate = db.Column(db.DateTime, default=db.func.current_timestamp())
    def serialize(self):
        return {
            'id': self.songID,
            'title': self.title,
            'artist': self.artist.name,
            'url': self.audiolink,
            'album': self.album.name
        }
class Playlist(db.Model):
    playlistID = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100), nullable=False)
    creationdate = db.Column(db.DateTime, default=db.func.current_timestamp())
    username = db.Column(db.String(50), db.ForeignKey('user.username'), nullable=False)
    songs = db.relationship('Song', secondary='PlaylistSongs', backref=db.backref('playlists', lazy=True))

    def serialize(self):
        return {
            "id": self.playlistID,
            "name": self.name,
            "username": self.username,
            "songs": [song.serialize() for song in self.songs]  # Assuming Song model has a serialize method
        }
class PlaylistSongs(db.Model):
    _tablename_ = 'PlaylistSongs'
    playlistID = db.Column(db.Integer, db.ForeignKey('playlist.playlistID'), primary_key=True)
    songID = db.Column(db.Integer, db.ForeignKey('song.songID'), primary_key=True)

class AlbumSongs(db.Model):
    _tablename_ = 'AlbumSongs'
    albumID = db.Column(db.Integer, db.ForeignKey('album.albumID'), primary_key=True)
    songID = db.Column(db.Integer, db.ForeignKey('Song.songID'), primary_key=True)

class UserListeningHistory(db.Model):
    _tablename_ = 'UserListeningHistory'
    username = db.Column(db.String(50), db.ForeignKey('user.username'), nullable=False, primary_key=True)
    songID = db.Column(db.Integer, db.ForeignKey('song.songID'), nullable=False, primary_key=True)
    listeningDate = db.Column(db.DateTime, default=db.func.current_timestamp(), primary_key=True)

    # Define relationship to Song model
    song = db.relationship('Song', backref='listening_history')

class UserFollowsArtists(db.Model):
    _tablename_ = 'user_follows_artists'
    username = db.Column(db.String(50), db.ForeignKey('user.username'), primary_key=True)
artistID = db.Column(db.Integer, db.ForeignKey('artist.artistID'), primary_key=True)
```

Note:
Frontend code is available on: *https://github.com/Musxeto/Moseeqify*
(Code updates will also be available at this repository).