MINISTRY OF EDUCATION AND TRAINING
CAN THO UNIVERSITY
COLLEGE OF INFORMATION AND COMMUNICATION TECHNOLOGY

FINAL PROJECT REPORT

CT313H: WEB TECHNOLOGIES AND SERVICES

MUZIK ONLINE

Student ID 1: B2206004
Student Name 1: Tran Thi Kim Phung

Student ID 2: B2206005
Student Name 2: Huynh Tu Phuong

FINAL PROJECT REPORT

SEMESTER 3, ACADEMIC YEAR: 2024-2025

CT313H: WEB TECHNOLOGIES AND SERVICES

- **Project/Application name: MUZIK**
- **Youtube link:** *https://www.youtube.com/watch?v=MjR5t9blds4*
- **GitHub link (for both frontend and backend):**
  - *https://github.com/24-25Sem3-Courses/ct313hm01-project-hfu186*
- **Student ID 1:** B2206005
- **Student Name 1:** Huynh Tu Phuong
- **Student ID 2:** B2206004
- **Student Name 2:** Tran Thi Kim Phung
- **Class/Group Number (e.g, CT313HM01):** CT313HM01

I. Introduction

- **Description:** This is a web-based music streaming application inspired by Spotify. It allows users to discover, play, and manage music online. Key features include user authentication, playlist creation, song favorites, album and artist management, and audio file uploading. The system supports different user roles such as guests, registered users, and administrators, and provides a RESTful API documented with OpenAPI standards.
- Functions:

Playlist Management

- Create, update, delete, and manage playlists.
- Add cover images and a list of songs to each playlist.

Favorites & Likes

- Mark songs as favorites.
- View your liked songs for quick access and playback.

Album & Artist Management

- Browse and search albums by name or artist.
- Manage artist profiles including name, avatar, and biography.

- Each album can contain multiple songs and be linked to specific artists.

Advanced Search & Browse

- Search for songs, albums, artists, or playlists by keyword.
- Get suggestions based on genres, trends, or recent activity.

Audio Upload (Admin & Artist)

- Upload audio/music files with metadata (song title, artist, album, etc.).
- Support for common formats such as MP3.

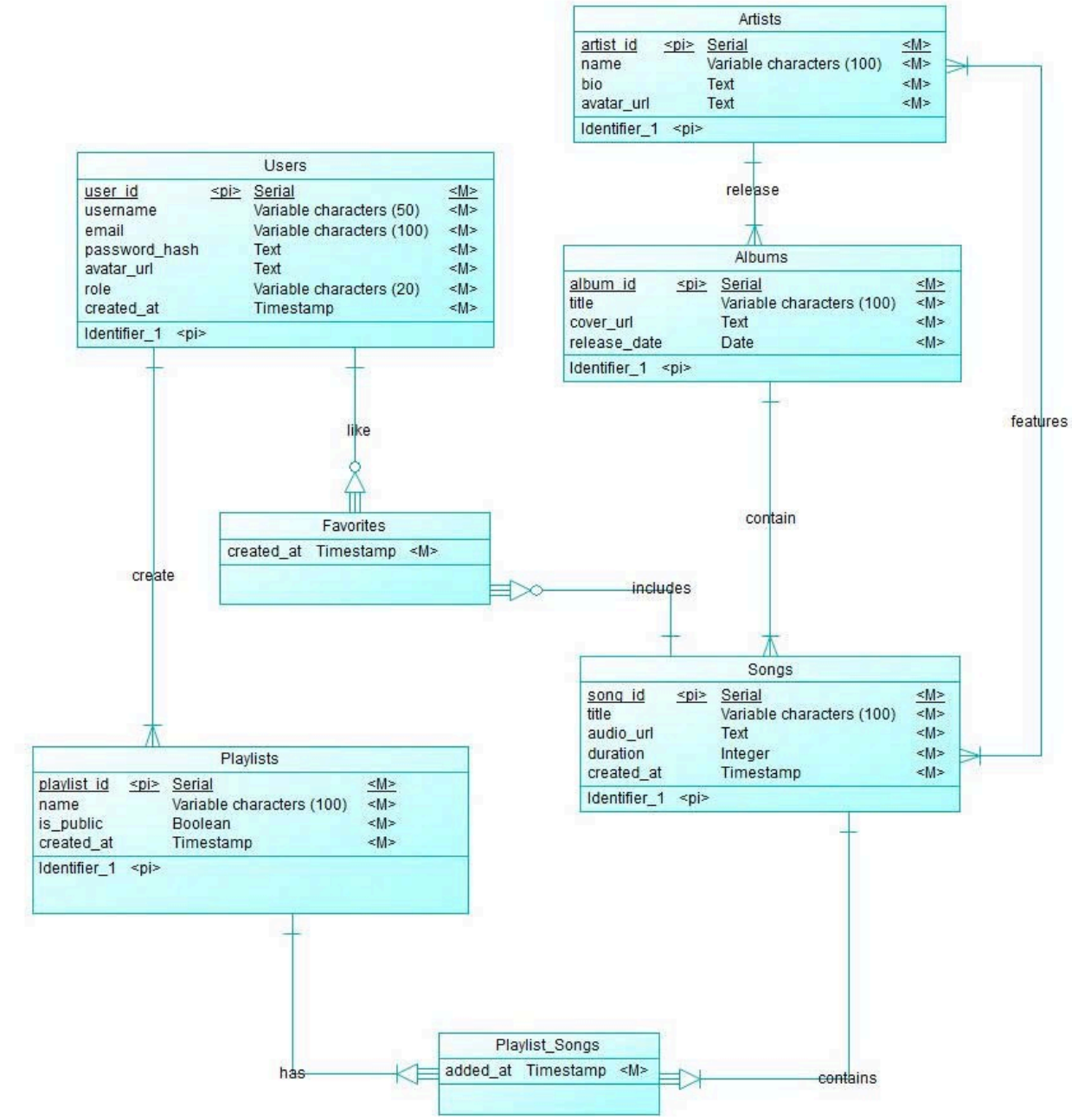
User Roles & Authentication

- Secure user registration and login with JWT authentication.
- Role-based access control: User, Admin.
  Each user has a profile and can interact with content.

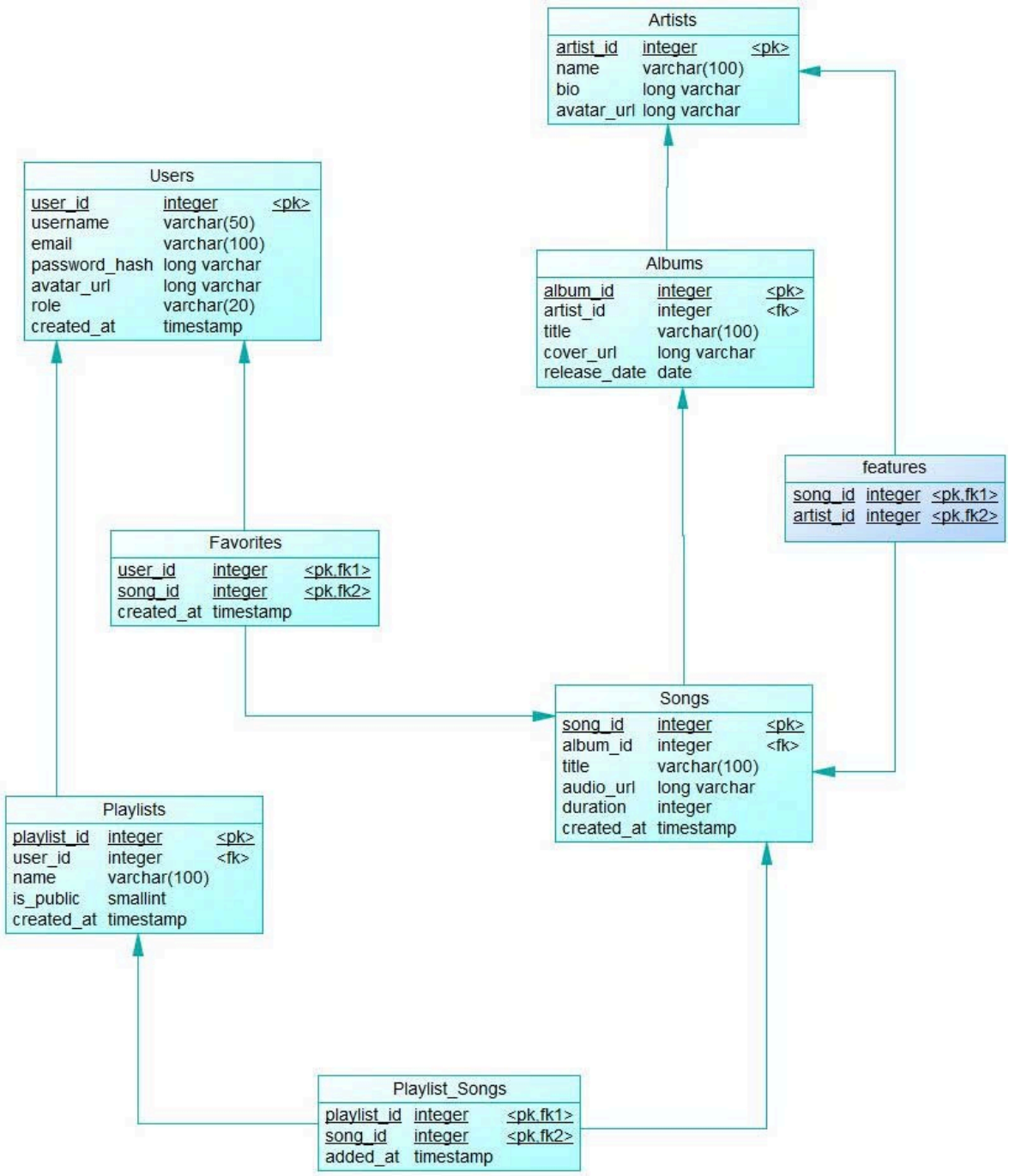
Admin Dashboard

- Manage users, songs, albums, and artists.
- View system-wide statistics such as total counts of songs, users, artists, and albums.
- Admin-only permission to delete all users.

- **Database schema**
  *** CDM**


Artists
artist_id <pi> Serial <M>
name Variable characters (100) <M>
bio Text <M>
avatar_url Text <M>
Identifier_1 <pi>
release
Users
user_id <pi> Serial <M>
username Variable characters (50) <M>
email Variable characters (100) <M>
password_hash Text <M>
avatar_url Text <M>
role Variable characters (20) <M>
created_at Timestamp <M>
Identifier_1 <pi>
Albums
album_id <pi> Serial <M>
title Variable characters (100) <M>
cover_url Text <M>
release_date Date <M>
Identifier_1 <pi>
features
like
Favorites
created_at Timestamp <M>
contain
create
includes
Songs
song_id <pi> Serial <M>
title Variable characters (100) <M>
audio_url Text <M>
duration Integer <M>
created_at Timestamp <M>
Identifier_1 <pi>
Playlists
playlist_id <pi> Serial <M>
name Variable characters (100) <M>
is_public Boolean <M>
created_at Timestamp <M>
Identifier_1 <pi>
Playlist_Songs
added_at Timestamp <M>
has
contains

*** PDM**



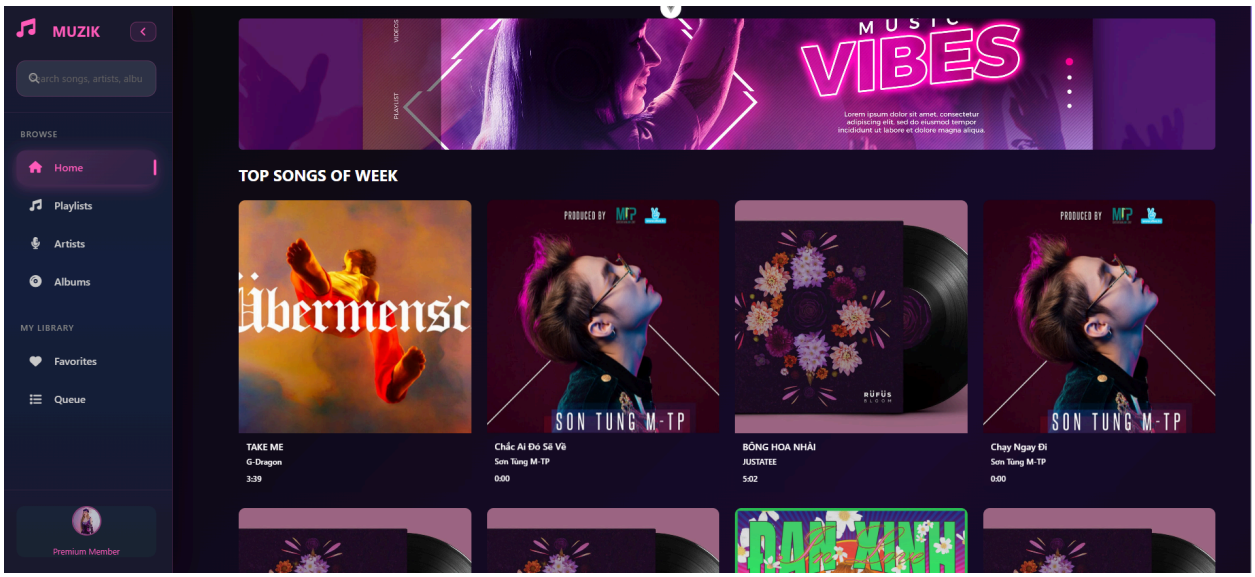
- A task assignment sheet for each member if working in groups.

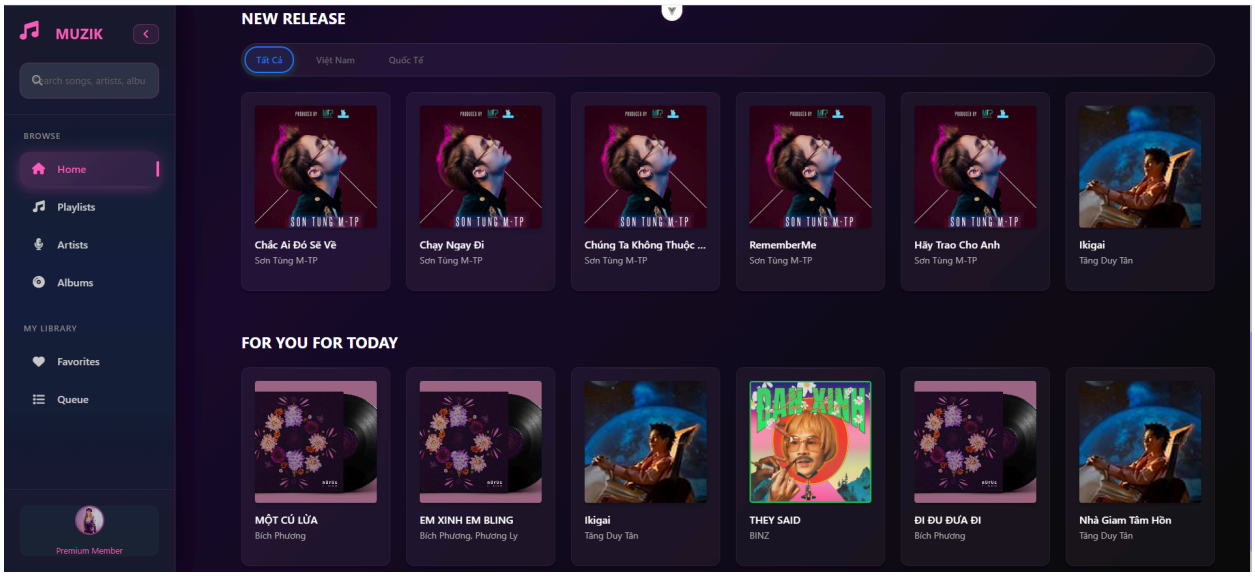
No.	Task	Member
1	Design database and data processing	Huynh Tu Phuong
2	Initialize the API Server	Tran Thi Kim Phung

3	Implement REST API for CRUD and document with OPEN API (Swagger UI)	Huynh Tu Phuong Tran Thi Kim Phung
4	Create functions to get data from server using Vue/Query;Tanstack	Huynh Tu Phuong
5	Implement client-side state for playing song and client side with route	Tran Thi Kim Phung
6	Implement UI admin	Tran Thi Kim Phung
7	Implement UI user	Huynh Tu Phuong
8	Design UI for VUe components using Bootstrap grid system	Huynh Tu Phuong Tran Thi Kim Phung
9	Build and deploy SPA & API using Caddy	Huynh Tu Phuong

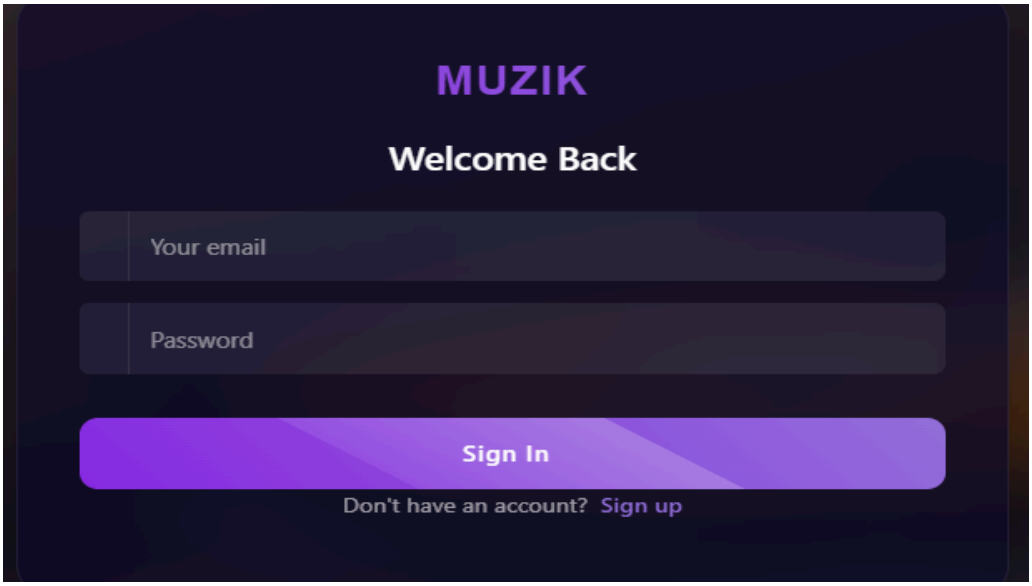
II. Details of implemented features

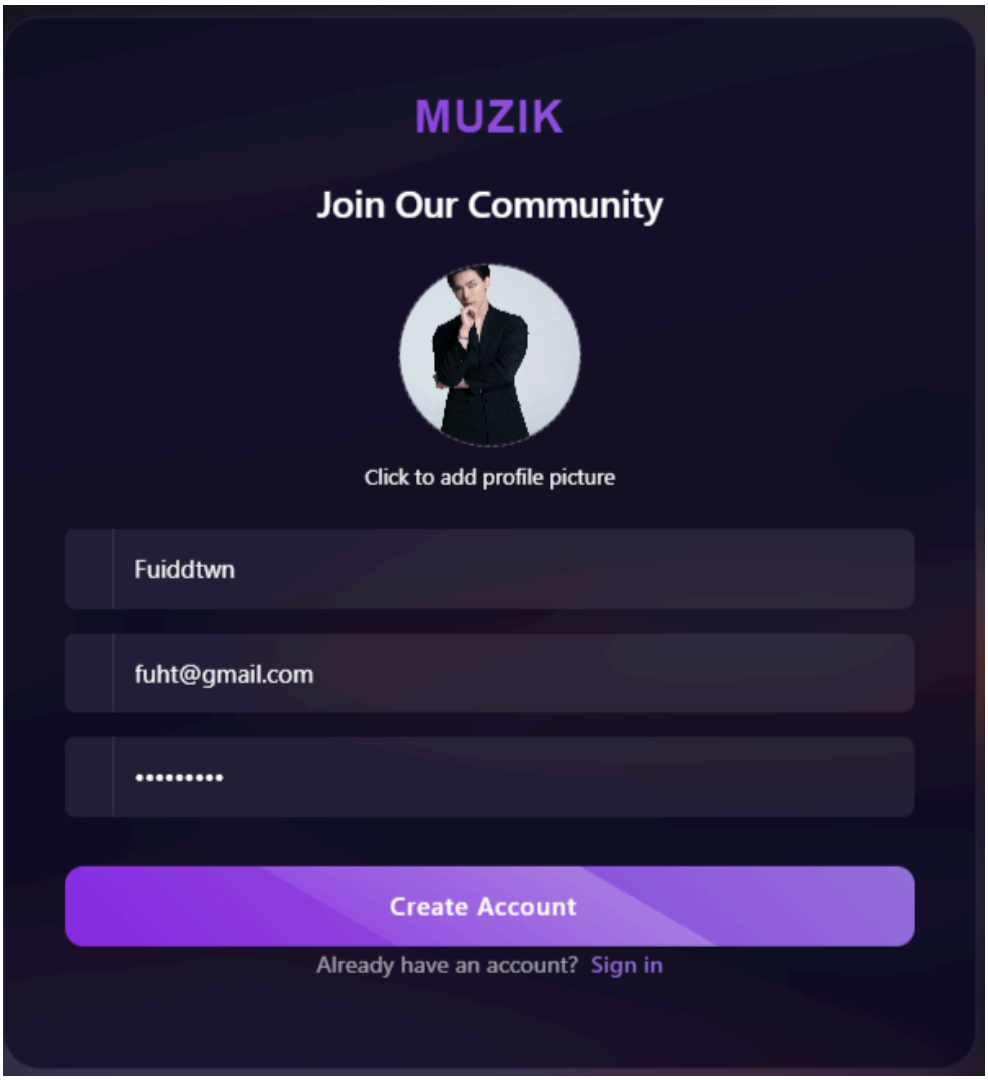
Main HOME PAGE - USER





1. **Feature: Log In Account**
   Description:
   - Allows users to register a new account and log in using their credentials. Passwords are securely hashed before being stored, and a JSON Web Token is issued upon successful login to handle authentication in subsequent requests.
   - This feature consists of two main components:
     + **Register:** Allows new users to create an account by providing necessary information such as: user name, email address, password.Optionally includes additional fields like profile picture
     + **Log In**: Lets existing users access their account by submitting: email, password


MUZIK
Welcome Back
Your email
Password
Sign In
Don't have an account? Sign up


MUZIK
Join Our Community
Click to add profile picture
Fuiddtwn
fuht@gmail.com
Create Account
Already have an account? Sign in

Library - use:

- **Bootstrap, Font Awesome icon**

Server - side APIs:

- **Register:**
  - POST { /api/auth/register }: This endpoint is used to register a new user.
  - Request Body: multipart/form-data
    Fields:
    - ❖ username (string, required): User's display name (min 3 characters).
    - ❖ email (string, required): A valid email address.
    - ❖ password (string, required): Password (min 6 characters).
    - ❖ avatar_url (file, optional): Profile image.
  - Response: application/json
- **Log in**
  - POST { /api/auth/login }: This endpoint authenticates a user by their email and password
  - Request Body: multipart/form-data
    - ❖ email (string, required): User's email address.
    - ❖ password (string, required): User's password.
  - Response: application/json

Table: Users

Data Handling:

- Write: Register stores new user data with hashed password.
- Read: Login verifies email and compares hashed password.

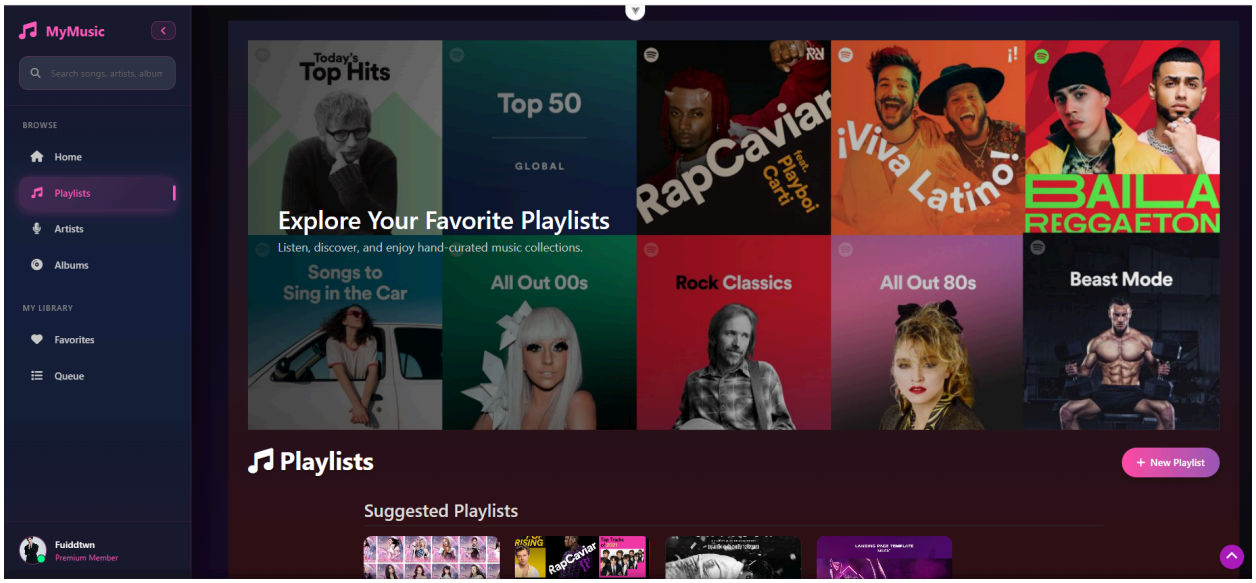
Client-side States:

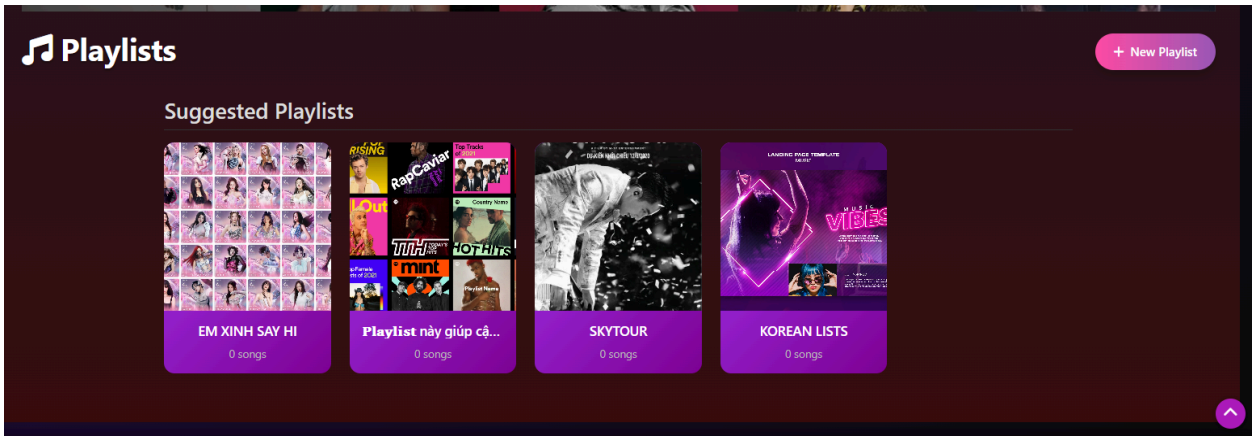
isLogin, email, password, avatarFile, username, avatarPreview, toast

2. Playlists

- **Feature: View all playlists**

Description: Retrieve a list of all playlists available in the system. This endpoint returns an array of playlists, each including basic information such as playlist name, cover image, creator, and number of songs.This endpoint is typically used by users to browse all available playlists.





Server - side APIs:

- **GET /api/playlists:** Retrieve a list of all playlists in the system.
  Request Body: None
  Response: application/json

Library - use:

- **Bootstrap, Font Awesome icon**

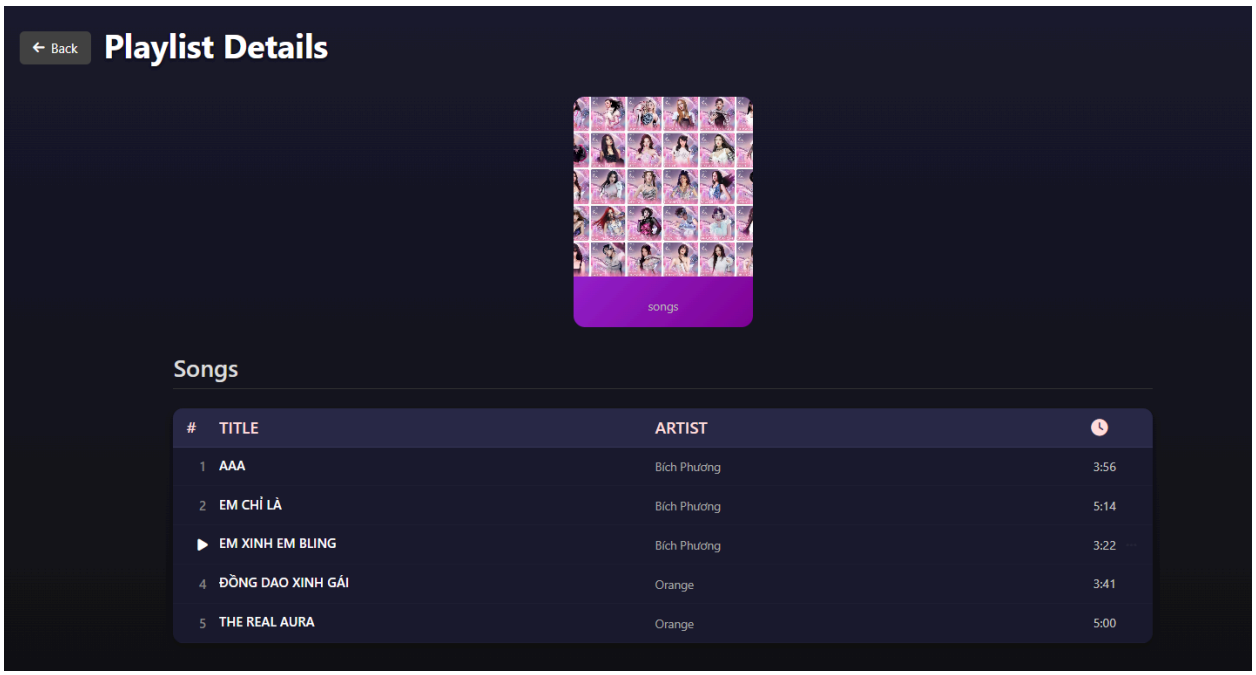
Table: Playlists, Songs, Artists

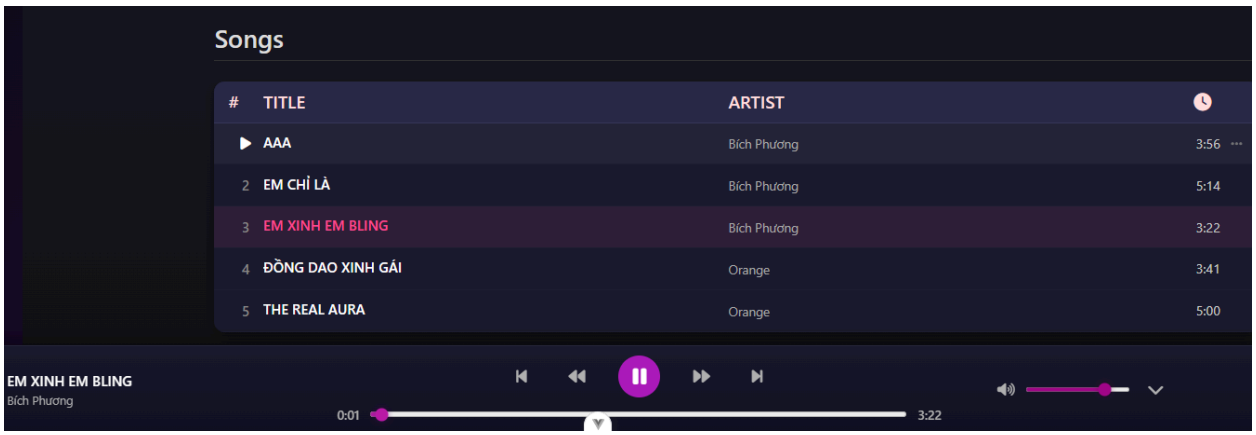
- ***Read data*:** Retrieve data from the system to get a list of all playlists in the database, including:Basic playlist information (ID, name, cover image).List of songs contained in each playlist.Artist information for each song.

Typical Client-side states: loading, error:, showCreateModal, defaultPlaylistImage:, playlists:, topPlaylists: [], userPlaylists: [], systemPlaylists

- **Feature: View a playlist detail**

Description: Get detailed information about a specific playlist by ID.





Server - side APIs:

- **GET/api/playlists/:id:**Get details of a specific playlist

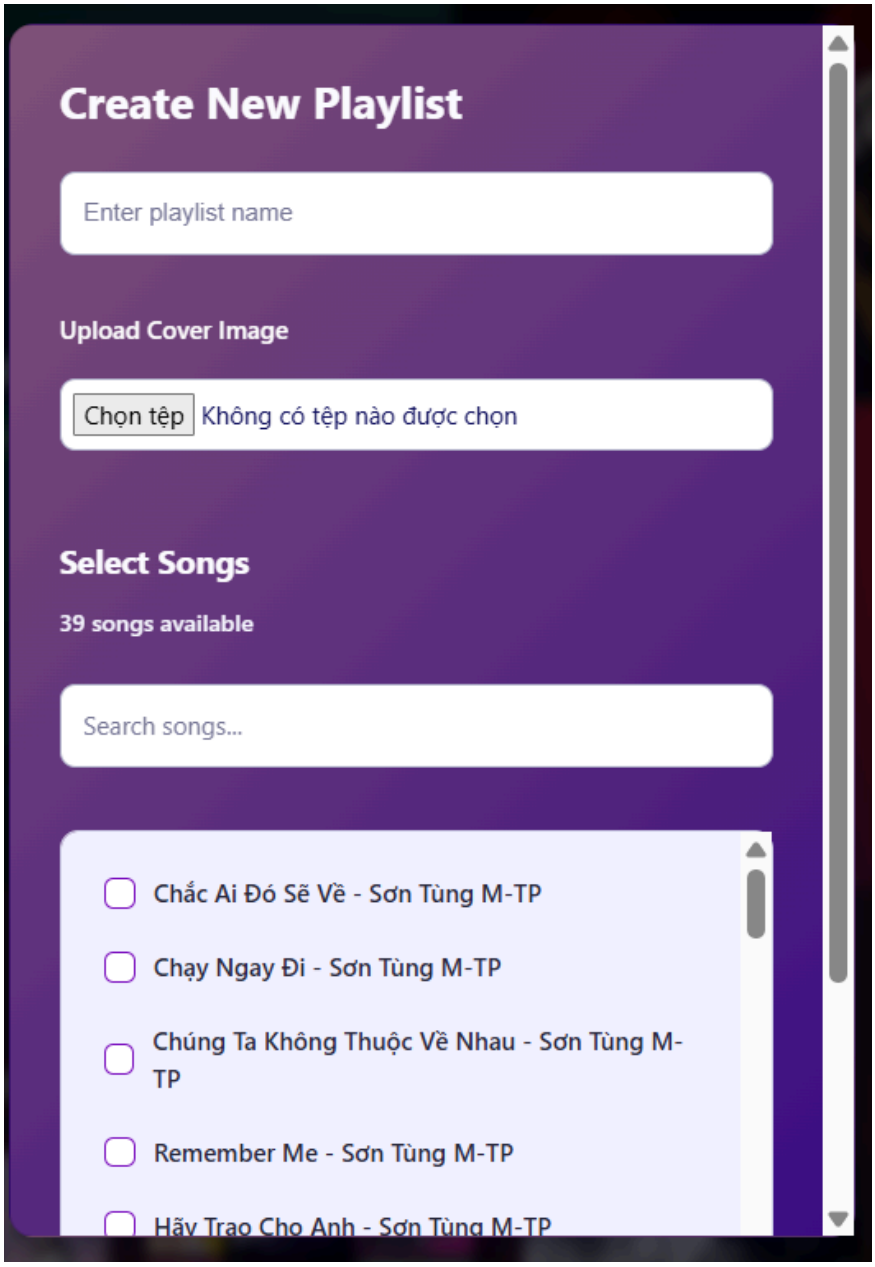
  Request Body: playlist_id: integer

  Response: application/json

Tables:**Playlists,PlaylistSongs,Songs,Artists**

Read data: Retrieve data from the system to get a playlist in the database, including:Basic playlist information (ID, name, cover image).List of songs contained in each playlist.Artist information for each song.
Typical Client-side states: playlistId, playlist[], isLoading, error, player, route, router

- **Feature: Create a playlist**
  Description**:** Create a new playlist in the system. This endpoint allows a user to submit a playlist name, optional cover image, and a list of songs (by title or ID). Upon successful creation, the new playlist is saved and associated with the authenticated user. This endpoint is typically used when users want to organize their favorite songs into custom playlists.



Library - use:

- **Bootstrap, Font Awesome icon**

Server - side APIs:

- **POST /api/playlists:** Create a new playlist with a list of songs.

  Request Body: multipart/form-data

  - ❖ name (string, required): Playlist name.
  - ❖ song_ids (array<integer>, required): List of song IDs to include in the playlist.
  - ❖ cover (string(binary), optional): Playlist cover image file.

  Response body: application/json.Information of added playlist

Table: **Playlists, Songs, Artists, Playlistsongs**

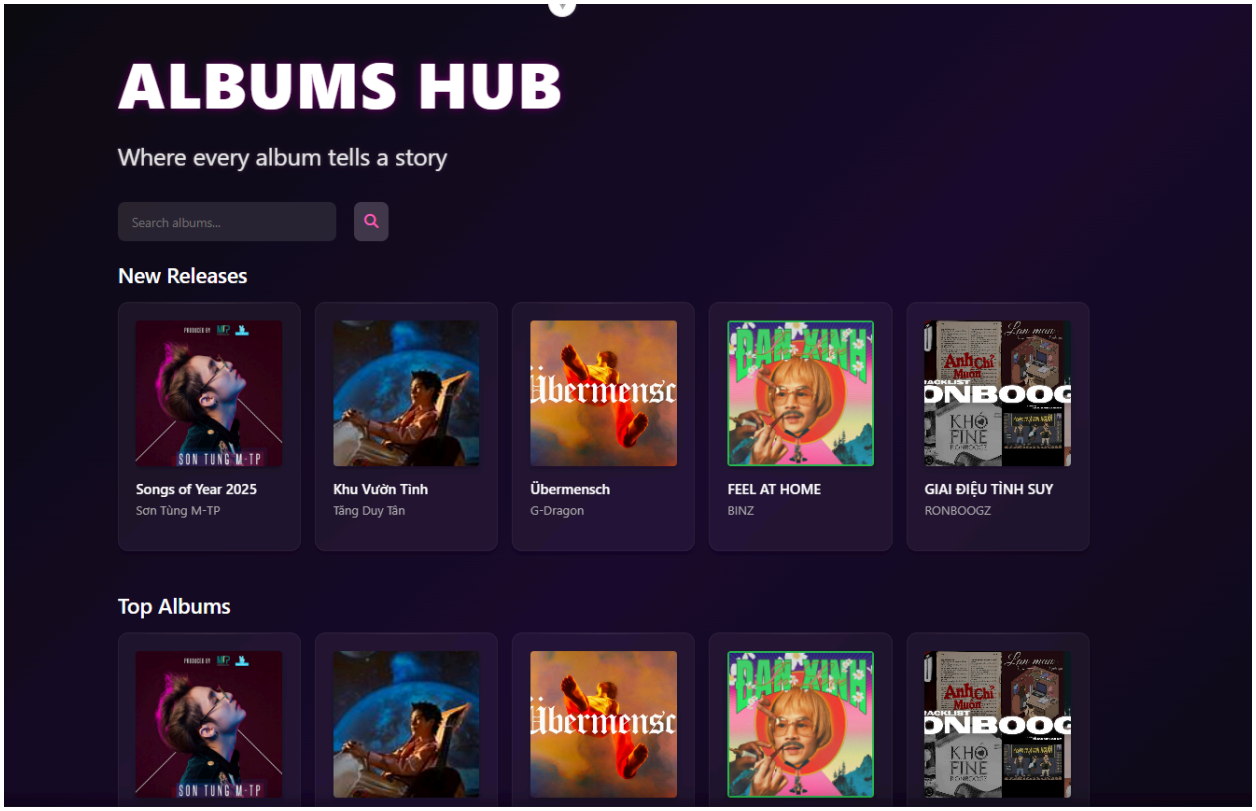
- **Read data:** Retrieve data from the system to get a list of all playlists in the database, including:Basic **playlist** information (ID, name, cover image).List of **songs** contained in each playlist.**Artist** information for each song.
- **Write data:**Insert new data into the system by creating playlists and associating them with songs and artists. Creating a new **playlist** with a name, optional cover image, and the ID(s) or title(s) of songs.Inserting records into the playlists table (for the playlist info).Inserting related entries into the **playlistsongs** junction table to link songs to the playlist.

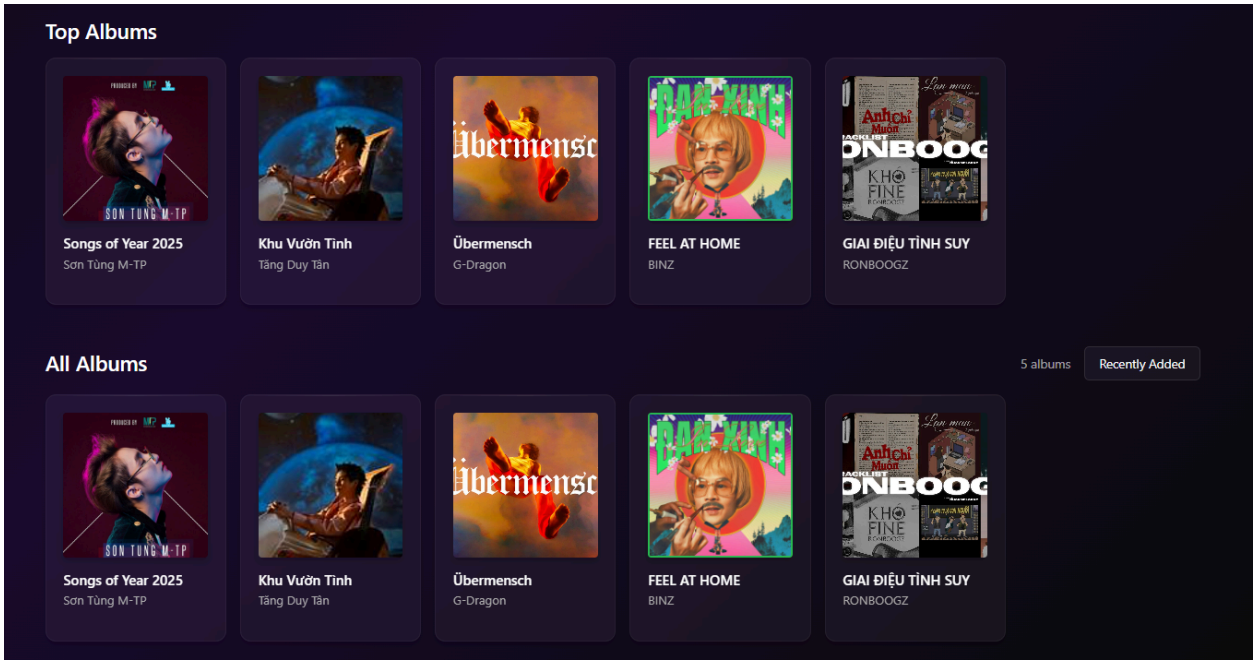
Typical Client-side states: playlistName, songSearch, selectedSongs, file, mutationError, isLoading, fetchError, songs, filteredSongs, isCreating

3. Albums

- **Feature: View all albums**

***Description*:**Start playback of all songs within a specific album. This endpoint retrieves the full list of tracks associated with the given album ID, ordered by track number or release sequence, and initiates playback starting from the first track. It is typically used when a user selects an album to listen to in its entirety.





Server - side APIs:

- **GET /api/albums:** Retrieve a list of all playlists in the system.
  - **Parameter:** none
  - **Response:** application/json

Library - use:

- **Bootstrap, Font Awesome icon**

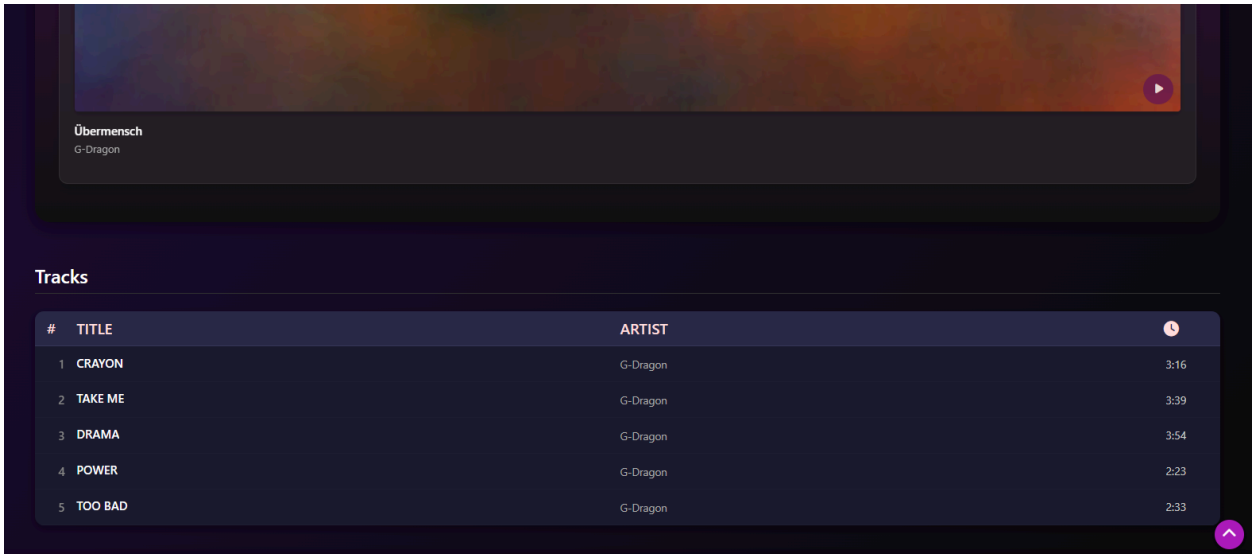
***Table*: Playlists, Songs, Artists**

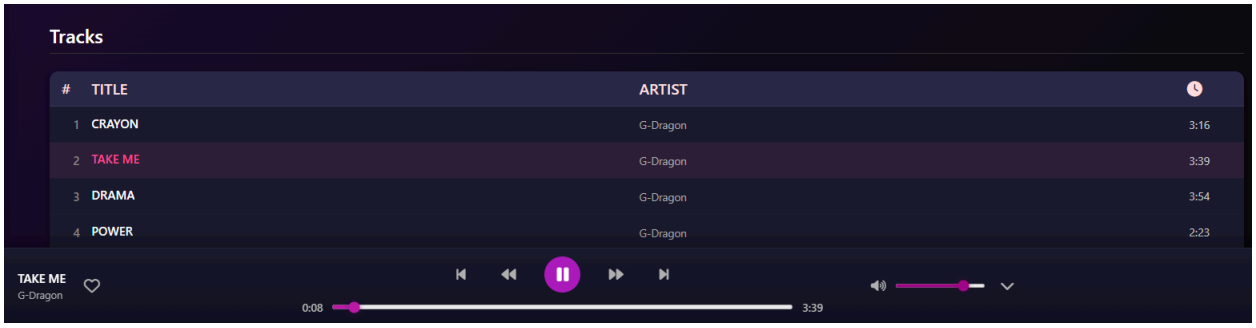
- **Read data:** Retrieve data from the system to get a list of all playlists in the database, including:Basic playlist information (ID, name, cover image).List of songs contained in each playlist.Artist information for each song.

Client-side states: currentSong, playlist, currentIndex, isPlaying, volume, currentTime, duration, isLoading, error, favorites, togglePlay.

- Feature: View songs with Album Details

***Description*:** This feature allows the client to retrieve and display detailed information of a specific album, including album metadata (title, cover image, release date, artist) and a list of all associated songs. The songs are ordered by their track number or release sequence. This is typically used when a user opens an album page to view its content before starting playback.





Server-side APIs:

- **GET /api/albums/{id}:** Retrieve details of a specific album by album ID.
    - **Parameter:** id:integer
    - **Response:** application/json

***Tables*: Albums, Songs, Artists**

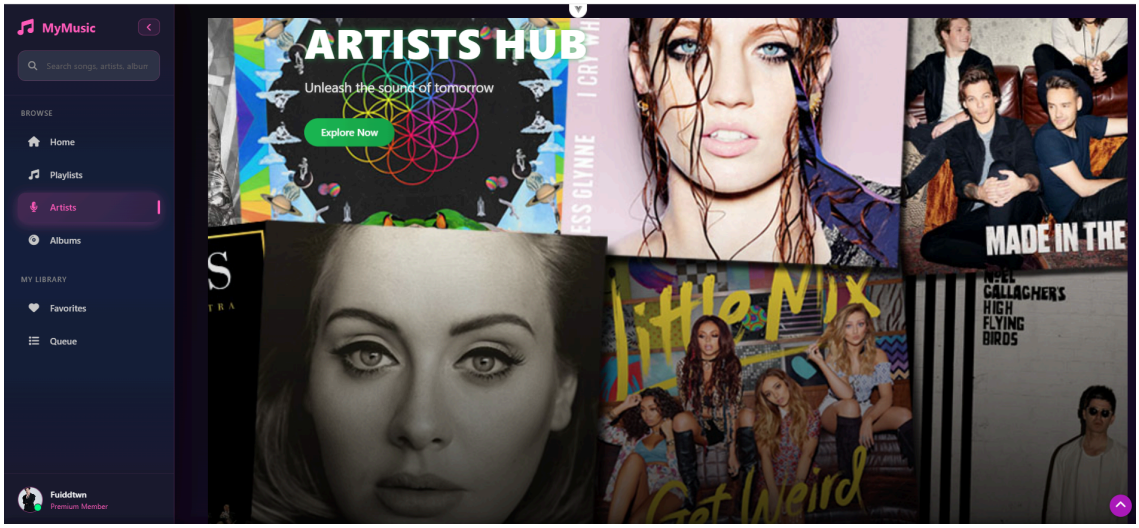
+ **Read data: The system retrieves data from the following sources:**
    - Album information: ID, title, cover image, release date.
    - Artist information: Linked via artist_id in the Albums table.
    - Songs list: All songs where album_id = {id}, sorted by track_number or release date.

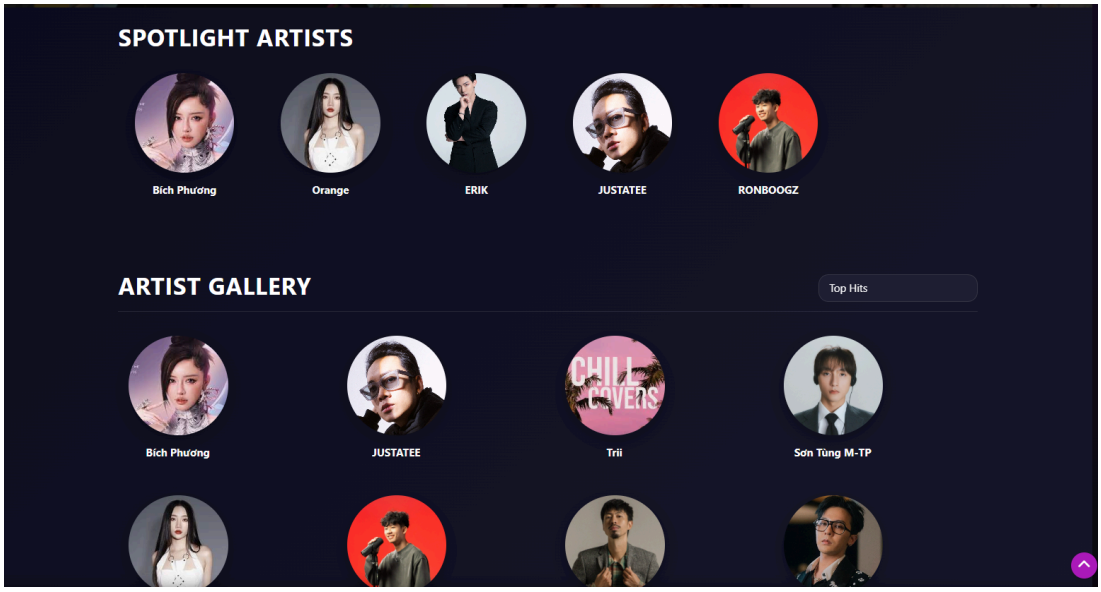
Client-side states: route, audioRef, currentSong, album, isLoading, error, albumSongs, isSongsLoading, songsError, player, route

4. ARTISTS

- **Feature: View all artists**

  ***Description*:** Allows users to view a list of all artists in the system. Each artist is displayed with their avatar and artist name in a grid format (grid gallery). This feature is commonly used in the Artist Gallery section of the user interface to explore art and access each artist's detail page**.**





Library - use:

- **Bootstrap, Font Awesome icon**

Server-side API

- **GET /api/artists**
  - **Parameters:***None*
  - **Response:** Content-Type: application/json

***Tables*:** Artists

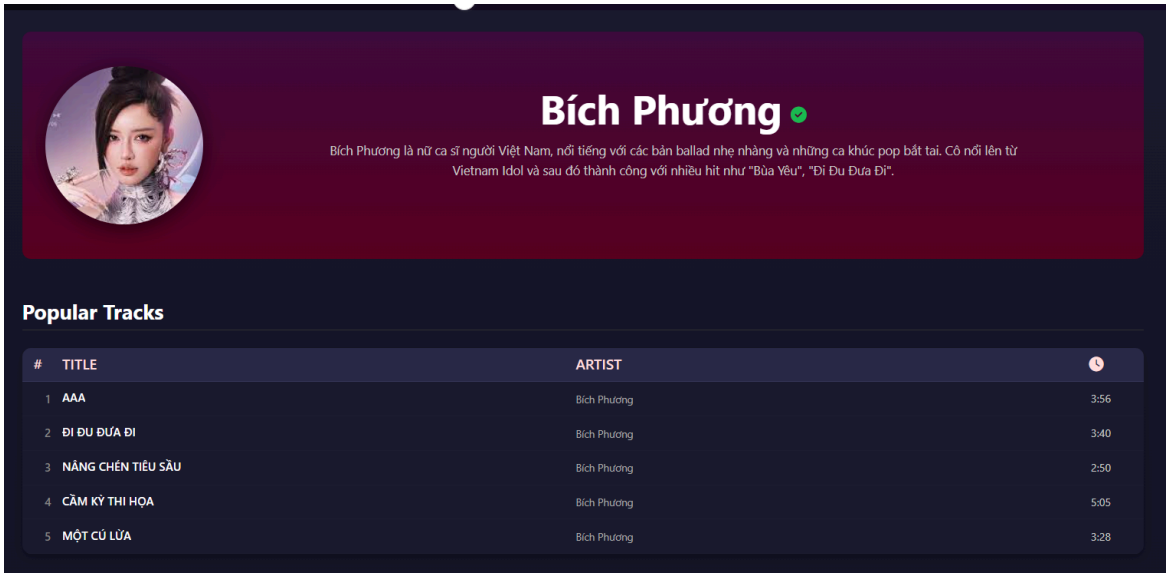
Read data: The system retrieves data from the following sources:

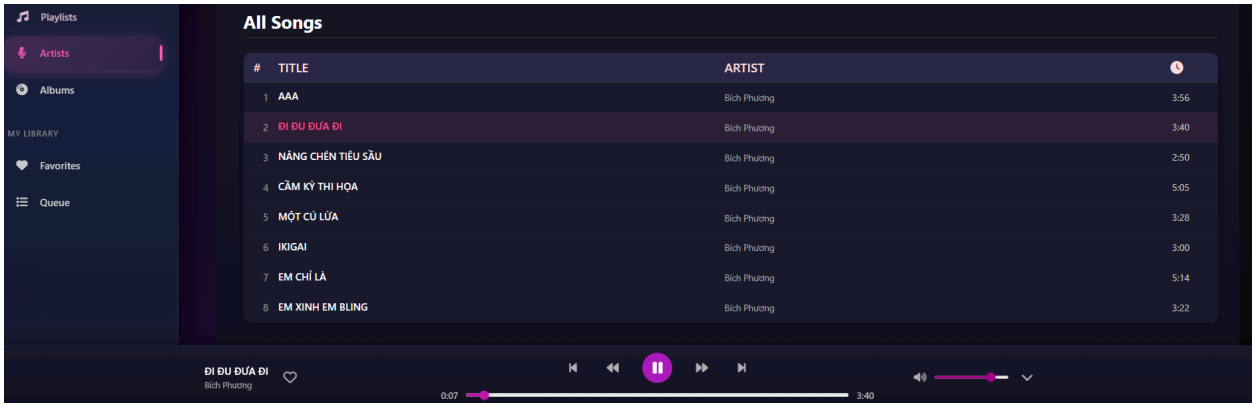
- Artist information: Name, avatar image

Client-side states:router, sortBy, artists, isLoading, error, sortedArtists

- **Feature: View artist detail**

***Description*:**
This feature enables users to view detailed information about a specific artist. Upon clicking on an artist from the gallery, the user is navigated to the artist's detail page. This page includes comprehensive data such as artist bio, avatar, list of released songs, and associated albums.





Library - use:

- **Bootstrap, Font Awesome icon**

Server-side API

- GET /api/artists/{id}
    - **Parameters:**id (path parameter): The unique identifier of the artist
    - **Response:Content-Type:** application/json

Tables :Artists, Songs, Albums, Songartists

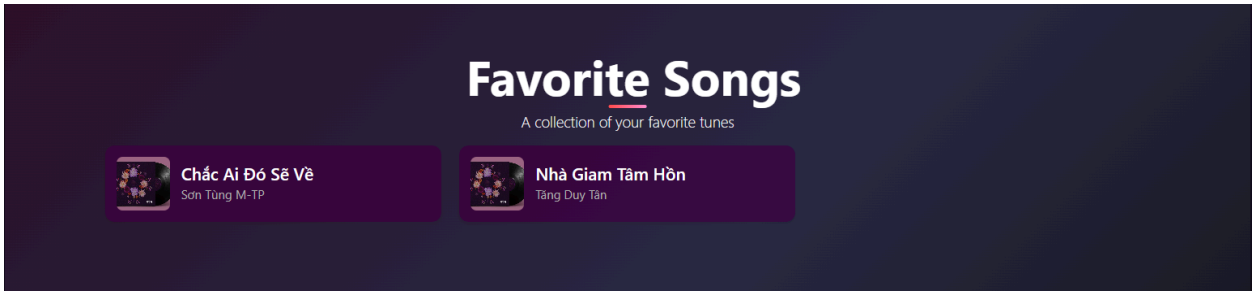
- Read data:
    - Artist info: name, bio, avatar image
    - Songs: all songs where artist_id = {id}
    - Albums: all albums where artist_id = {id}

Client-state:route, currentSong, audioRef, artistId, artist, artistSongs, artistAlbums, isLoading, error, popularSongs, allSongs, player.

5. Favorites

- **Feature: View Favorite Songs by User**

***Description*:** Allows users to view a personalized list of their favorite songs. This feature is commonly accessed from the "Favorites" section of the application, where users can play, manage, or remove songs they've previously marked as favorite. Each song displays its title, artist, album cover, and duration. This feature enhances user engagement by enabling easy access to frequently listened tracks.



Library - use:

- **Bootstrap, Font Awesome icon**

Server-side API

- **GET** /api/users/:user_id/favorites
  Retrieve all favorite songs for a specific user by user_id.

- **Parameters:** user_id: *integer* (from route param)
  Response: Content-Type: application/json. Returns a list of favorite songs, including song details, artist, and album info.

Tables:Users, Songs, Favorites, Artists, Songs

Read Data:

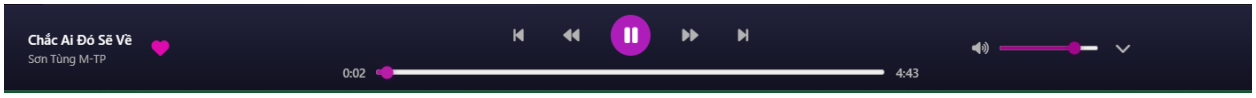
The system performs joins across tables to retrieve:

- User's list of favorite song IDs from Favorites
- Song information: title, duration, audio_url,cover_image
- Artist details: name, avatar_url

Typical Client-side States: userId, favoriteSongs, isLoading, error, currentSong, audioRef, isPlaying, playlist, player, route,isFavorited:

- **Feature: Add Favorite Song**

Description: Allows authenticated users to mark a specific song as their favorite. When a user taps the "favorite" icon on a playing song.



Server-side API

- **POST** /api/users/:user_id/favorites: Add a specific song to the authenticated user's favorites.
  Path Param: user_id: *integer* – ID of the authenticated user (from route)
  Request Body: *song_id :integer*
  Response*:Content-Type: application/json*

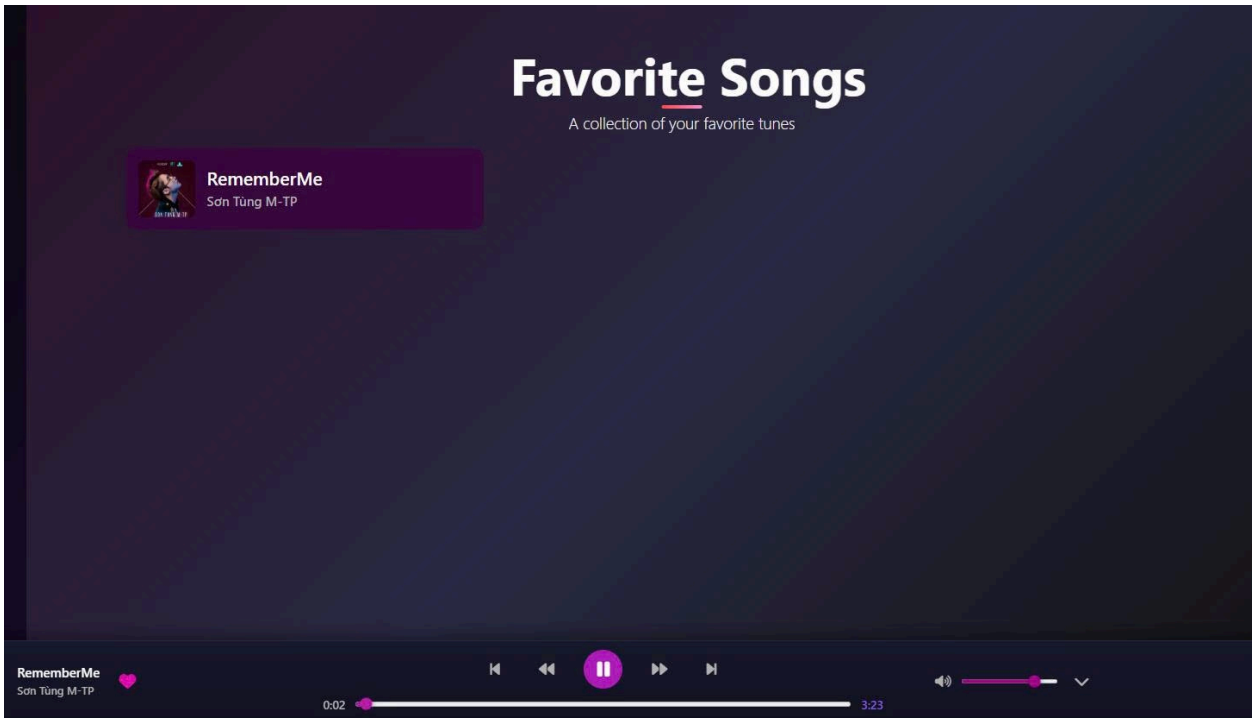
Table: **Users, Favorites ,Songs**

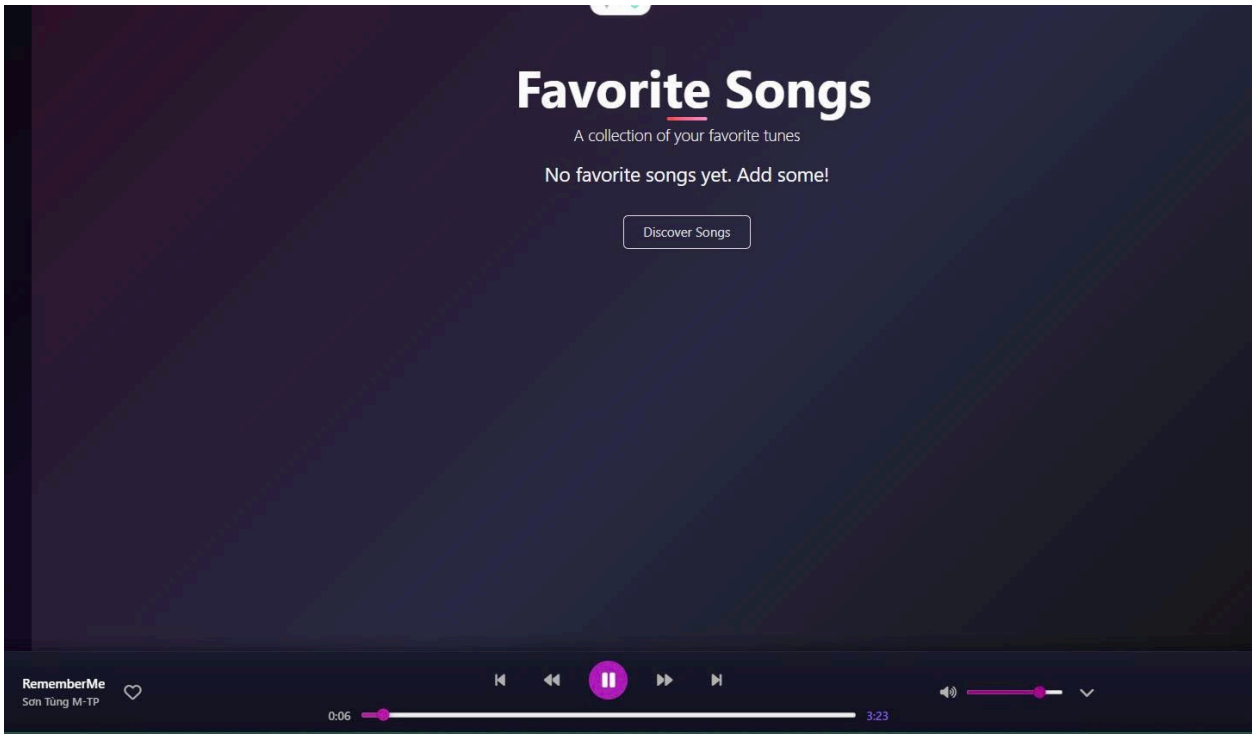
- **Write Data**
  - *Receive user_id from route and song_id from request body. Validate input and check if the song already exists in the user's favorites.*
  - *If not exists, insert a new row into Favorites.*
  - *Return confirmation message and inserted data.*

Client-side States : userId, isFavorite, handleToggleFavorite(song), auth, player, favoriteSongs, error, loading

- **Feature: Remove Favorite Song**

Description**:** Allows authenticated users to remove a previously marked favorite song. When a user taps the "favorite" icon again on a song that is already favorited, the system removes the corresponding record from the Favorites table





Server-side API

- **DELETE** /api/users/:user_id/favorites: Remove a specific song from the user's favorite list.
  Path Param: user_id: *integer* – ID of the authenticated user (from route)
  Request Body: *song_id :integer*
  Response*:Content-Type: application/json*

Table:Users, Favorites, Songs

Write Data:

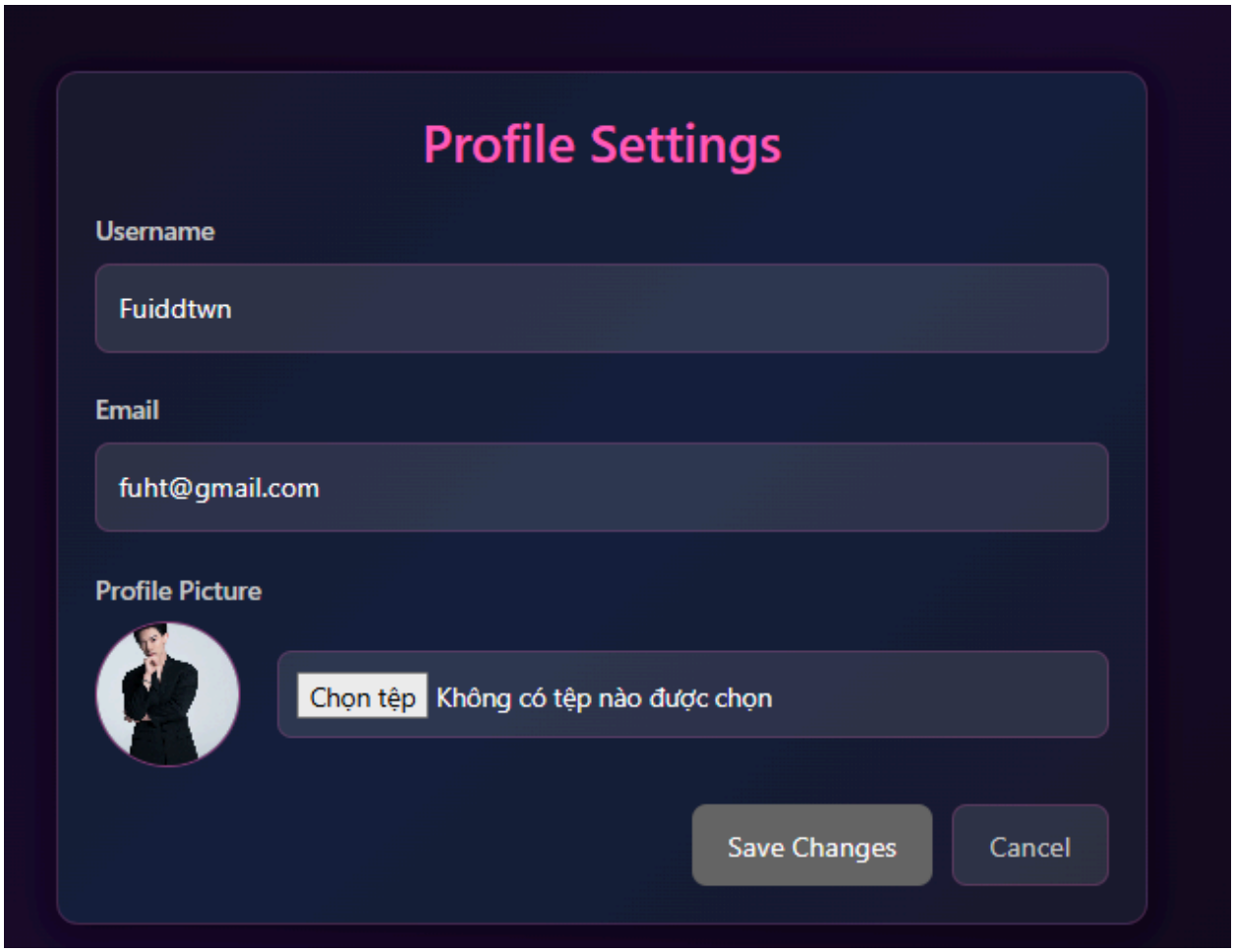
Receive user_id from route and song_id from request body.
Validate input and check if the song exists in the user's favorites.
If exists, remove the corresponding entry from Favorites.
Return a success message confirming removal.

***Client-side States*:** userId, isFavorite, handleToggleFavorite(song), auth, player, favoriteSongs, error, loading.

6. Feature: Update information

Description: Allows an authenticated user to update their personal profile information such as name, email, password, and avatar. This feature helps users

manage their account details to keep them current and personalized. Typically accessed from the "Account Settings" or "Profile" section.



Server-side API

- **PUT** /api/users/:user_id:Update user information by user ID.

Path Param:user_id: *integer* – ID of the authenticated user (from route)
Request Body: Content-Type: multipart/form-data

+ Username,Email,Avatar_url

Response: Content-Type: application/json

Tables:Users

Write Data:

1. **Receive user_id from route and form data from request body.**
2. **Validate and sanitize input fields (name, email, password).**

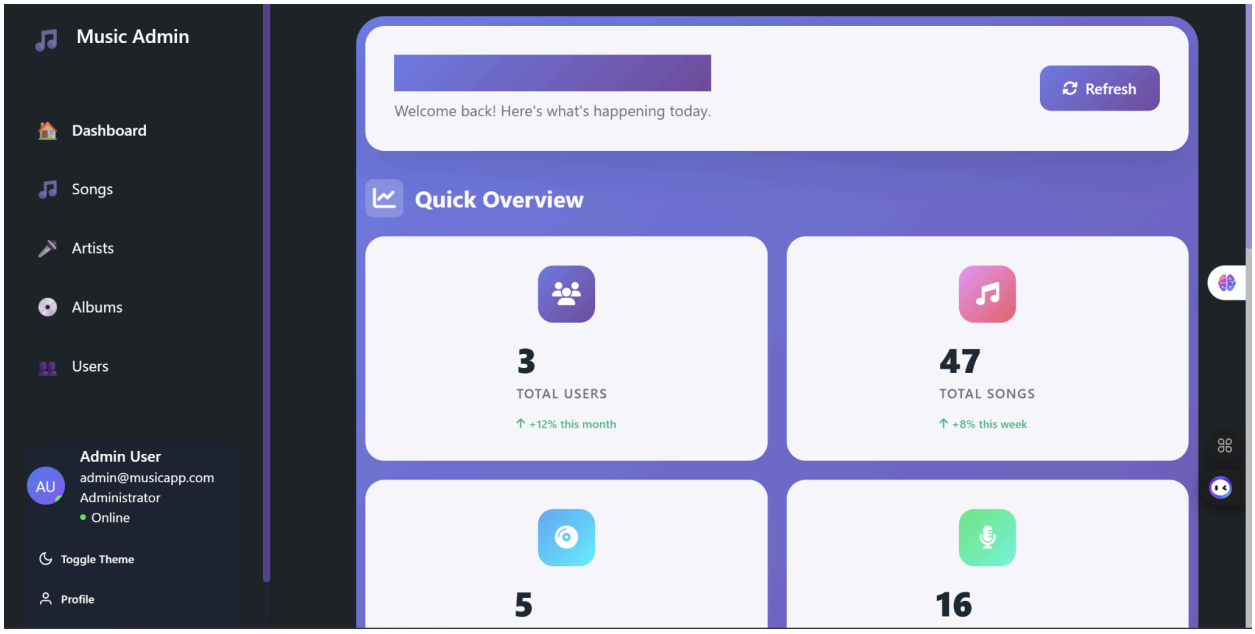
3. **If avatar image is provided, upload and store it.**
4. **Update user's record in the Users table.**
5. **Return updated user info in response.**

***Client-side States*:** router, userId, username, email, avatarFile, avatarPreview,userData, isLoading, isError, isFormChanged, isUpdating, authService, handleUpdateProfile(), handleFileChange(), handleCancel()

ADMIN

Details of implemented features

1. **Main HOME PAGE - ADMIN**



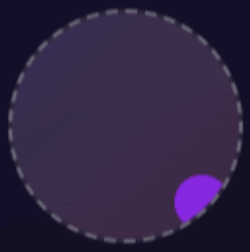
2. **Feature: Log In Account**

Description:

- Allows users to log in using their credentials. Passwords are securely hashed before being stored, and a JSON Web Token is issued upon successful login to handle authentication in subsequent requests.
- This feature consists of one main components:
  + **Log In**: Lets existing users access their account by submitting email and password

MUZIK

Join Our Community



Click to add profile picture

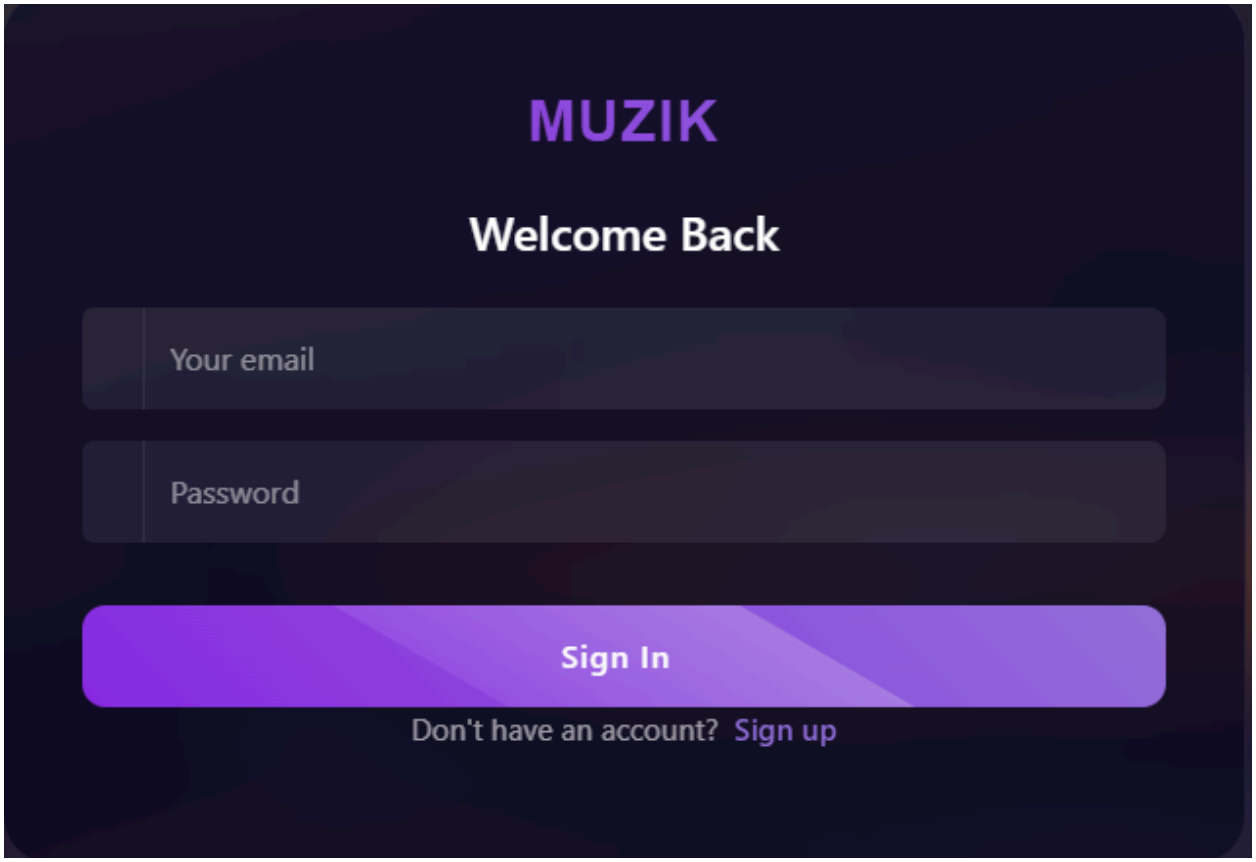
Username

Your email

Password

Create Account

Already have an account? Sign in



Library - use:

- **Bcrypt:** Hash user passwords before storing them in the database
- **Jsonwebtoken:** Generate and verify JWT access tokens after successful login
- **Express-rate-limit:** Limits the number of API requests to prevent spam and protect the system

Server - side APIs:

- **Log in**
  - POST { /api/auth/login }: This endpoint authenticates a user by their email/username and password
  - Request Body: application/json
    - ❖ email (string, required): User's email address or username
    - ❖ password (string, required): User's password.
  - Response: application/json

Table: Users

Data Handling:

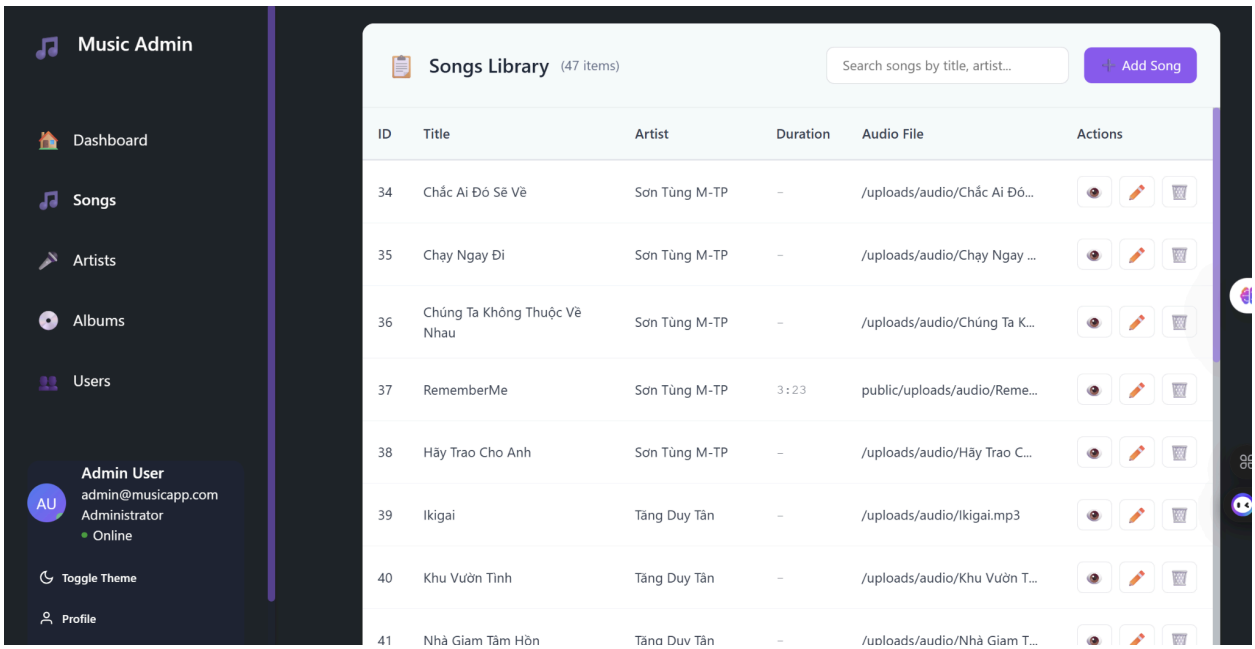
- Read: Login verifies email/username and compares submitted password with hashed password from database
- JWT token validation with automatic expiration checking

Client-side States : email, password, isSubmitting, loginError, adminToken, isLoggedIn, currentAdmin, authService, handleLogin(), handleInputChange(), handleLogout()

3. Songs Management

- Feature: View all songs

Description: Retrieve a list of all songs available in the system. This endpoint returns an array of songs, each including basic information such as song title, artist name, duration, audio file path, and associated metadata. This endpoint is typically used by admins to browse and manage all songs in the music library.



Libraries: knex (query builder), zod (validation, if used), multer (if response includes audio metadata)

Server - side APIs:

- **GET /api/songs:** Retrieve a list of all songs in the system.
  Request Body: None
  Response: application/json

Table: Songs, Artists, Albums

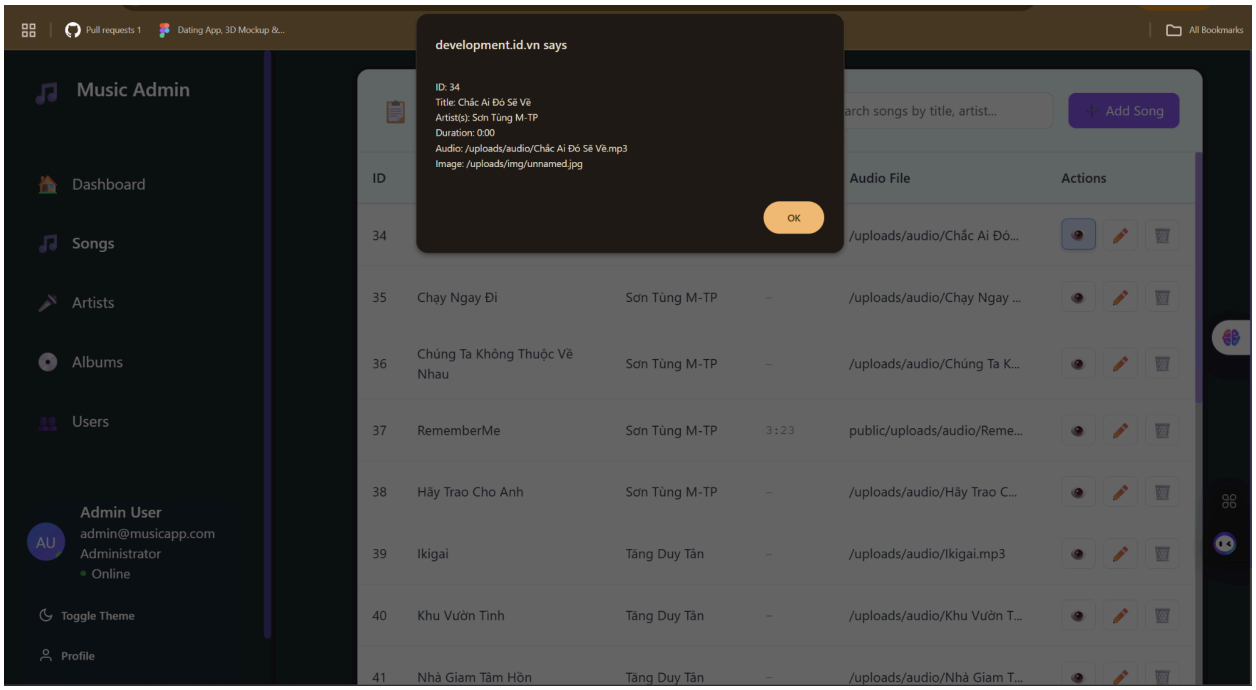
- ***Read data*:** Retrieve data from the system to get all songs with:
  + Basic song information (ID, title, duration)
  + Associated artist information
  + Audio file paths and metadata

+ Creation timestamps

Client-side States : songDetails, isFetching, error, songId, fetchSongDetails(), Router

- **Feature: View a song**

Description: Get detailed information about a specific song by ID, including full metadata, artist details, and file information.



Libraries: knex, zod

Server - side APIs:

- **GET /api/songs/:id:** Get details of a specific song

  Request Body: song_id: integer

  Response: application/json

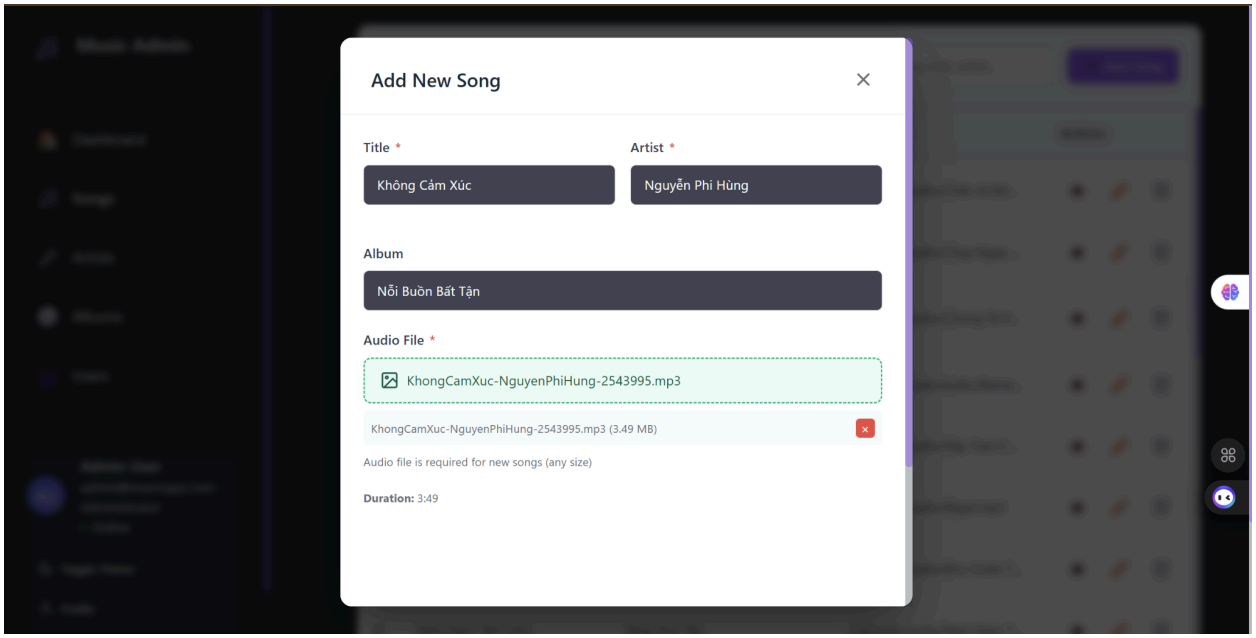
Tables: **Songs, Artists, Albums**

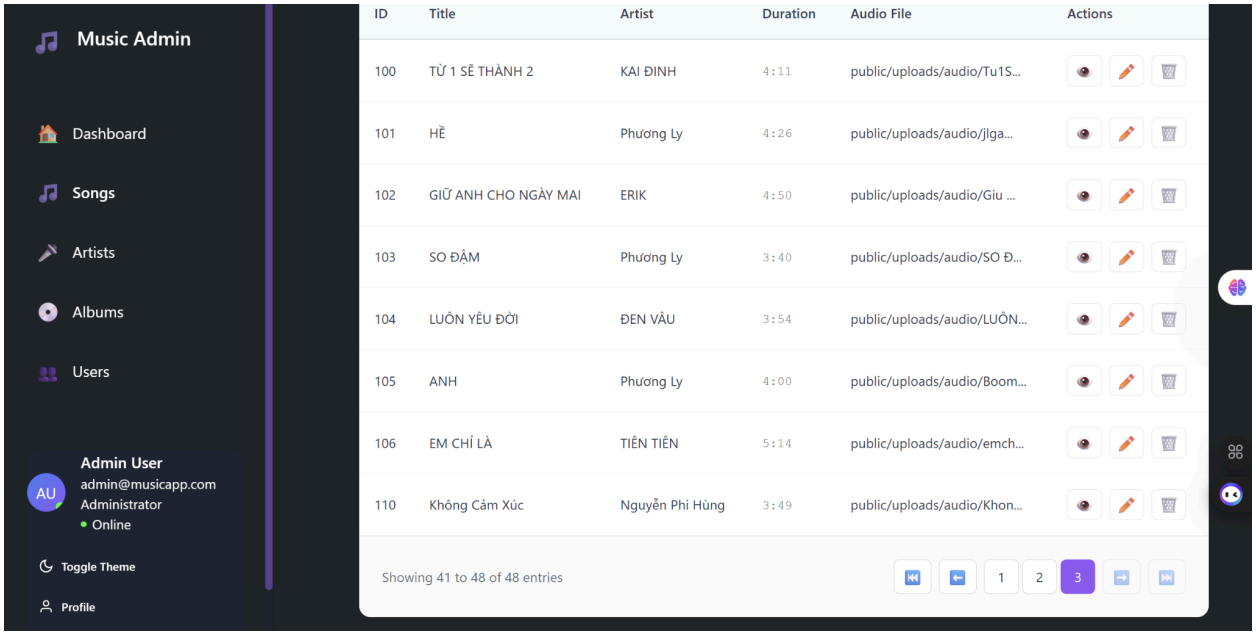
Read data: Retrieve detailed song information including artist relationships and file metadata.

Client-side States : songDetails, isFetching, error, songId, fetchSongDetails(), router

- **Feature: Create a song**

***Description*:** Create a new song in the system. This endpoint allows admins to upload an audio file along with song metadata (title, artist, album). The system processes the audio file, extracts metadata like duration, and stores both the file and metadata in the database.





Libraries: multer (file upload), knex, zod (input validation), ffprobe or music-metadata (audio duration)

Server - side APIs:

- **POST /api/songs/:** Create a new song

  Request Body: FormData containing:

  - title: string (required)
  - artist: string (required)
  - album: string (optional)
  - audio_files: File (required)

- duration: number (auto-calculated)

Response: application/json

Tables: **Songs, Artists, Albums**

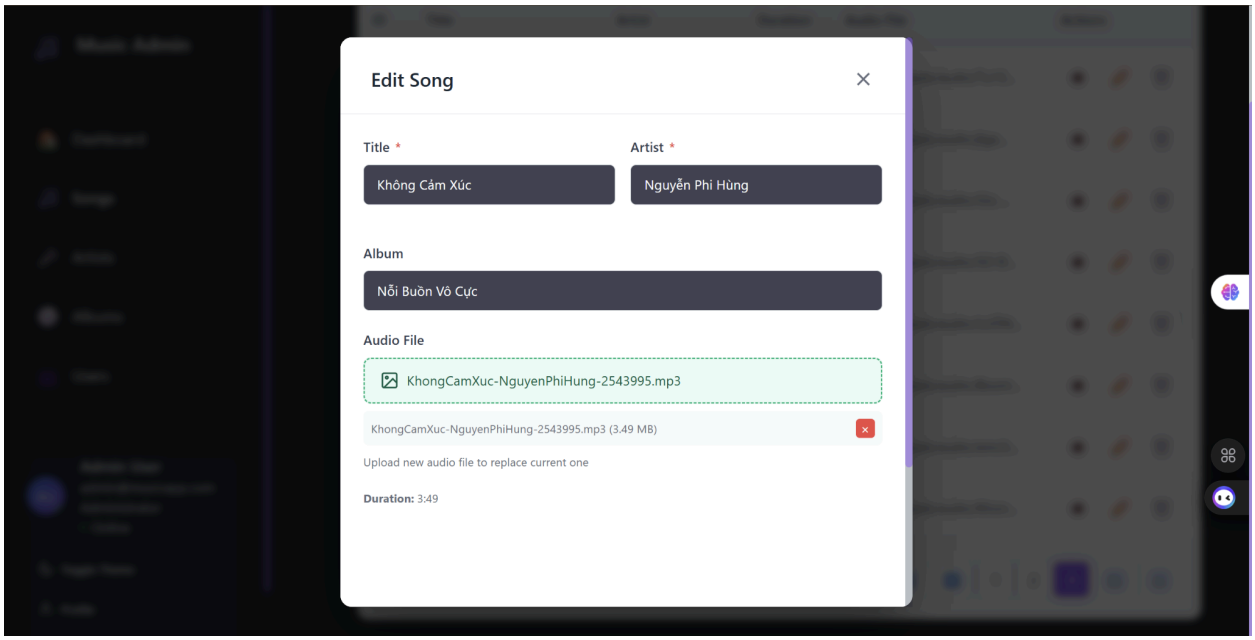
Read data: Fetch artist/album list for form dropdowns.

Write data: Insert new song record with file upload handling and metadata extraction.

Client-side States : title, artist, album, audioFile, uploadProgress, isCreating, validationErrors, successMessage, handleSubmit(), handleFileChange()

- **Feature: Edit a song**

***Description*:** Update an existing song's information including metadata and optionally replace the audio file. Supports partial updates where only changed fields are updated.



Libraries: multer, knex, zod

Server - side APIs:

- **PUT /api/songs/:id:** Update an existing song

  Request Body: FormData containing updated fields

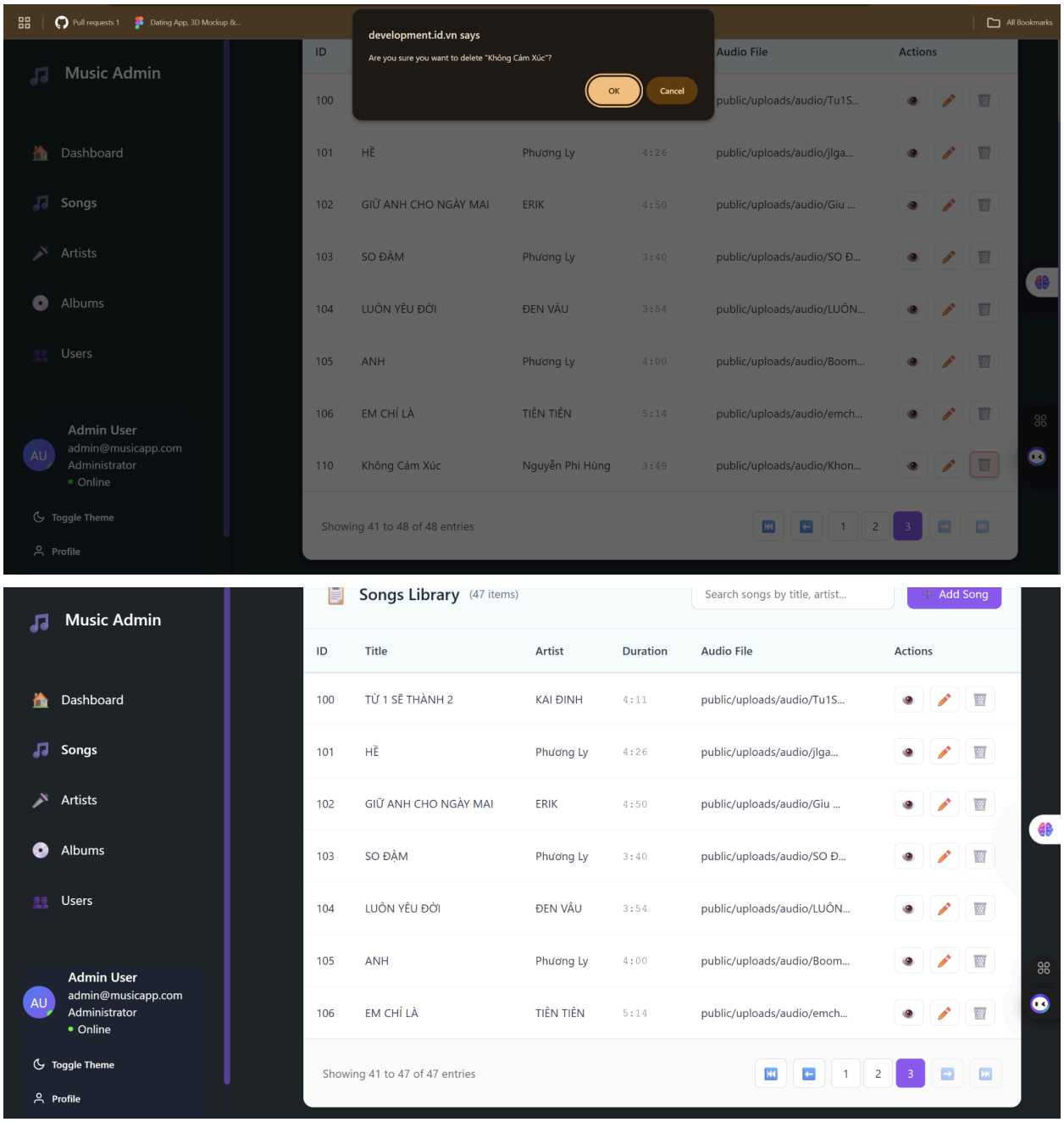
  Response: application/json

Read data: Fetch current song data and reference lists (artists, albums).

Write data: Update only changed fields; optionally update audio file.

Client-side States : editedSong, audioFile, isUpdating, updateError, handleUpdate(), handleInputChange(), handleCancel()

- **Feature: Delete a song**
  ***Description*:** Remove a song from the system including its audio file and database records.



Libraries: multer, knex, fs

Server - side APIs:

- **DELETE /api/songs/:id:** Delete a song

  Request Body: song_id: integer

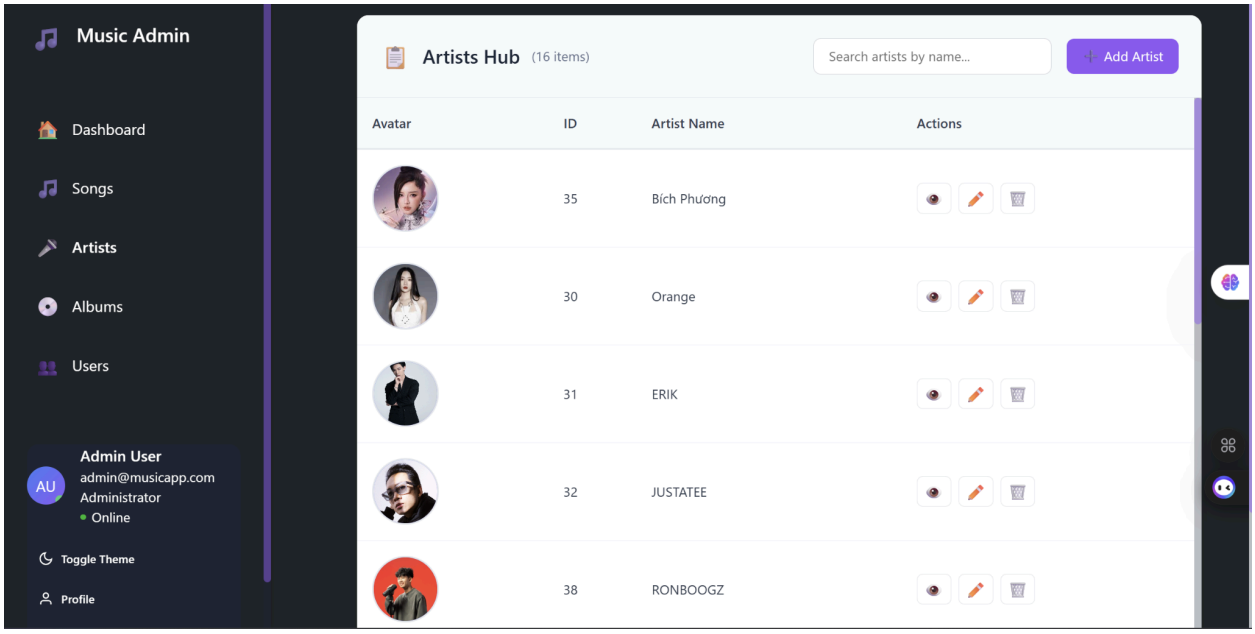
  Response: application/json

Read data: Used to confirm selected song before delete.
Write data: Deletes song record and audio file from storage.
Client-side States : selectedSong, confirmDeleteModalOpen, isDeleting, deleteError, handleConfirmDelete(), handleOpenModal(), handleCloseModal()

4. Artists Management

- Feature: View all artists

Description: Retrieve a list of all artists in the system with their profile information, avatar images, and basic statistics.



Libraries: multer, knex, zod
Server - side APIs:

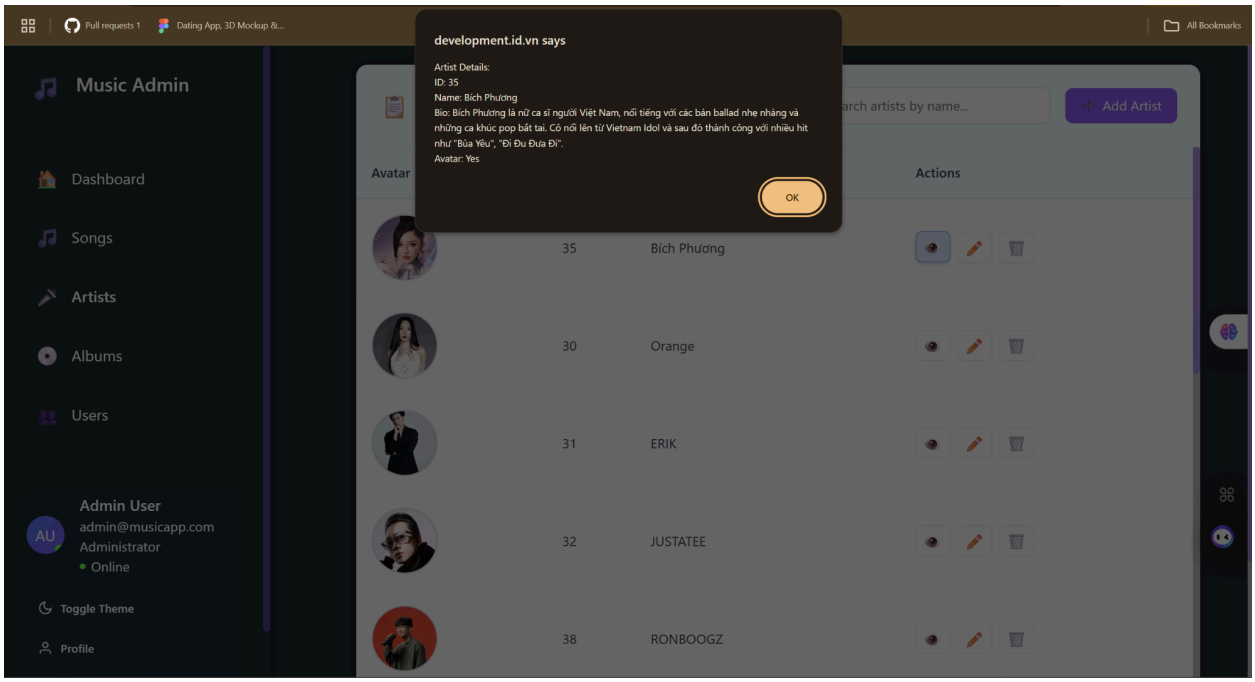
- **GET /api/artists:** Retrieve all artists
  Request Body: None
  Response: application/json

Table: Artists, Users
Read data: Retrieve data from the system to get all artists with:
+ Basic artist information (ID, name, biography)
+ Avatar image paths - Associated user information
+ Creation timestamps

Client-side States : artists, isLoading, error, searchText, selectedArtist, handleSelectArtist(), fetchArtists()

- Feature: View an artist

Description: Get detailed information about a specific song by ID, including full metadata, artist details, and file information.



Libraries: knex, zod

Server - side APIs:

- **GET /api/artists/:id:** Get details of a specific artist

  Request Body: artist_id: integer

  Response: application/json

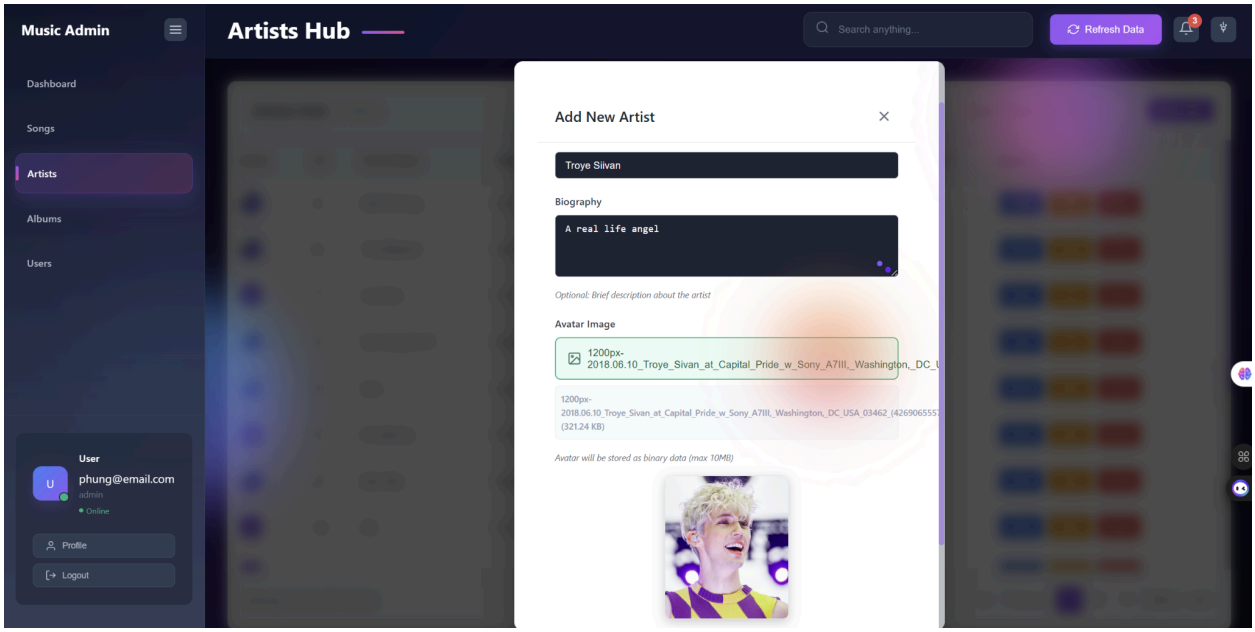
Tables: **Songs, Artists, Albums, Users**

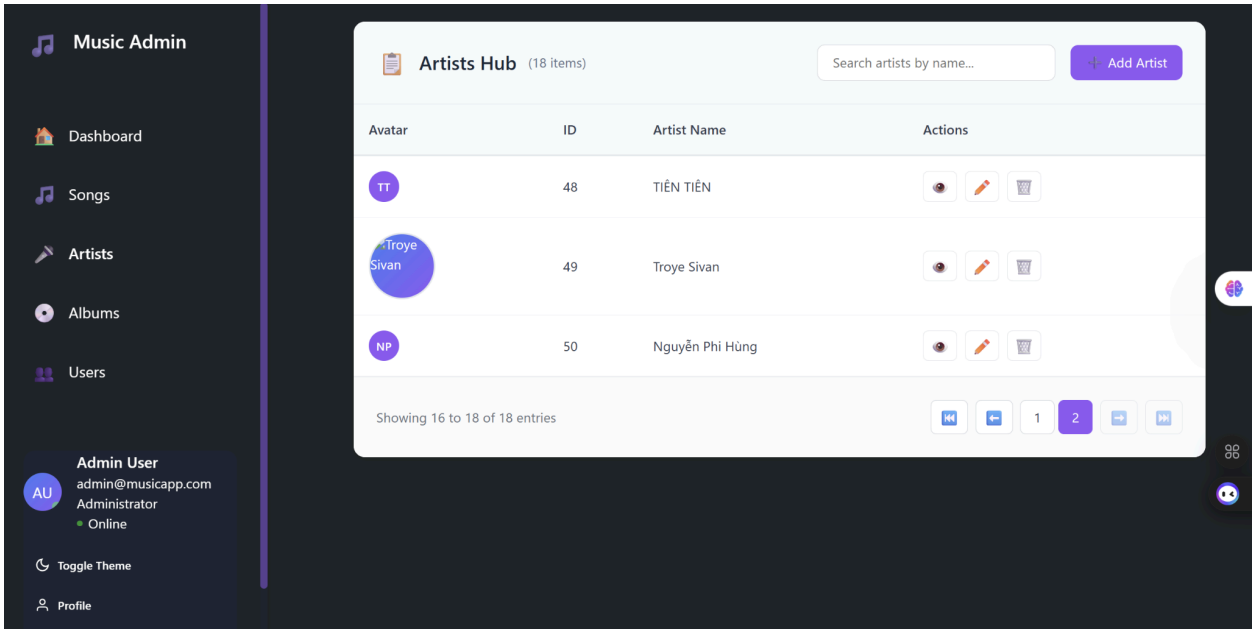
Read data: Retrieve detailed artist information including user relationships and associated content.

Client-side States : formData, avatarFile, previewAvatar, isSaving, formErrors, successMsg, handleInputChange(), handleFileChange(), handleSaveArtist(), handleCancel()

- **Feature: Create an artist**

***Description*:** Create a new artist in the system. This endpoint allows admins to add an artist profile with biographical information and avatar image upload. The system processes the avatar image and stores both the profile data and image in the database.





Libraries: multer (file upload), knex, zod

Server - side APIs:

- **POST /api/artists/:** Create a new artist

  Request Body: FormData containing:

  + name: string (required)
  + bio: string (optional)
  + avatar_url: File (optional)

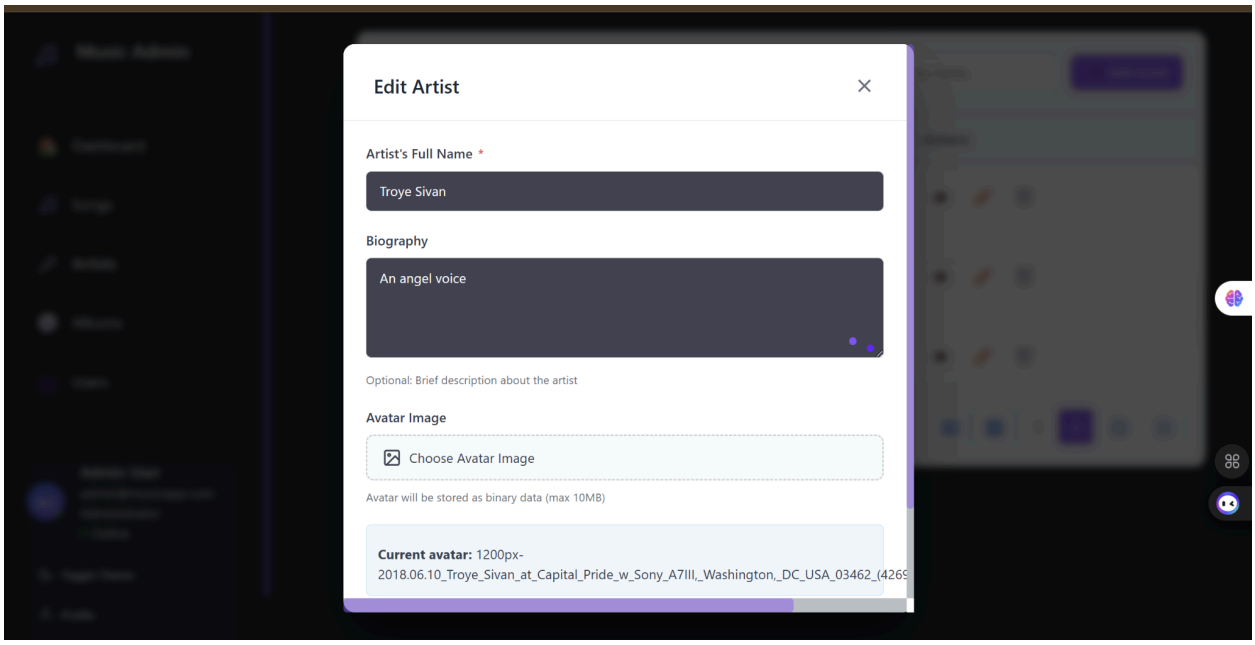
  Response: application/json

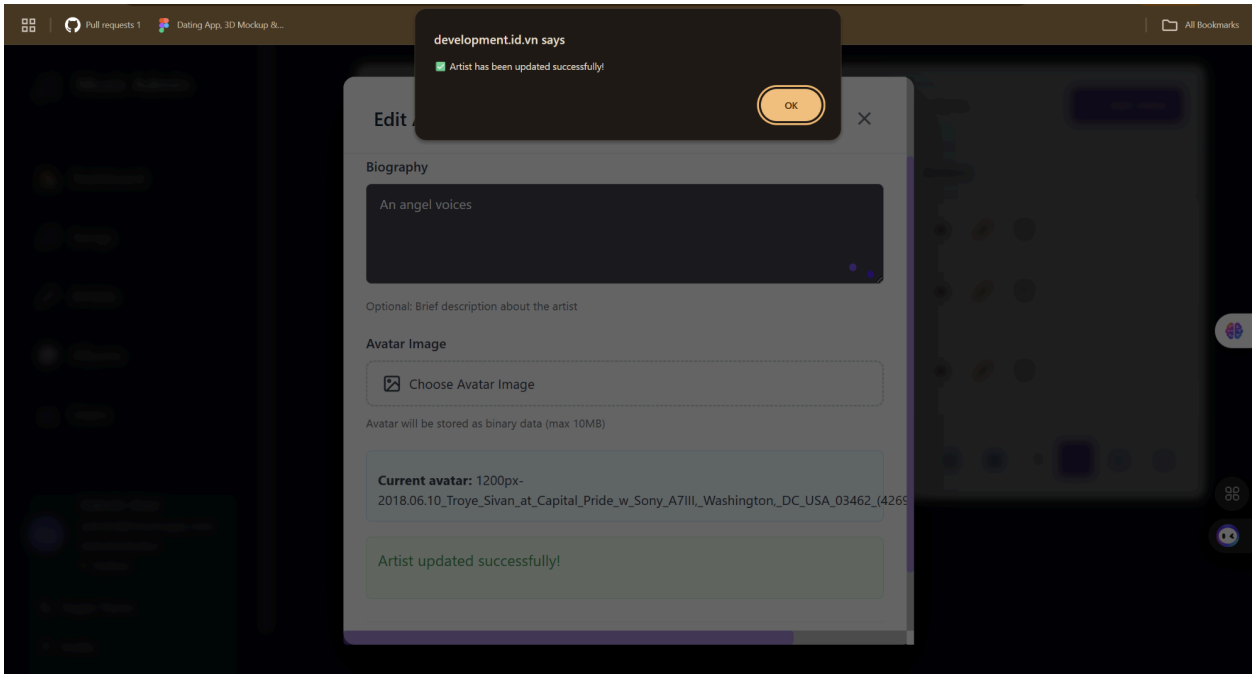
Tables: **Users, Artists**

Read data: Fetch related user list.
Write data: Insert new artist record with avatar image upload handling and profile data storage.
Client-side States : formData, avatarFile, previewAvatar, isSaving, formErrors, successMsg, handleInputChange(), handleFileChange(), handleSaveArtist(), handleCancel()

- **Feature: Edit an artist**

***Description*:** Update an existing artist's information including name, biography, and optionally replace the avatar image. Supports partial updates where only changed fields are updated.





Libraries: multer, knex, zod

Server - side APIs:

- **PUT /api/artists/:id:** Update an existing artist

  Request Body: FormData containing:

  + name: string (required)
  + bio: string (optional)
  + avatar_url: File (optional)

  Response: application/json

Tables: **Users, Artists**

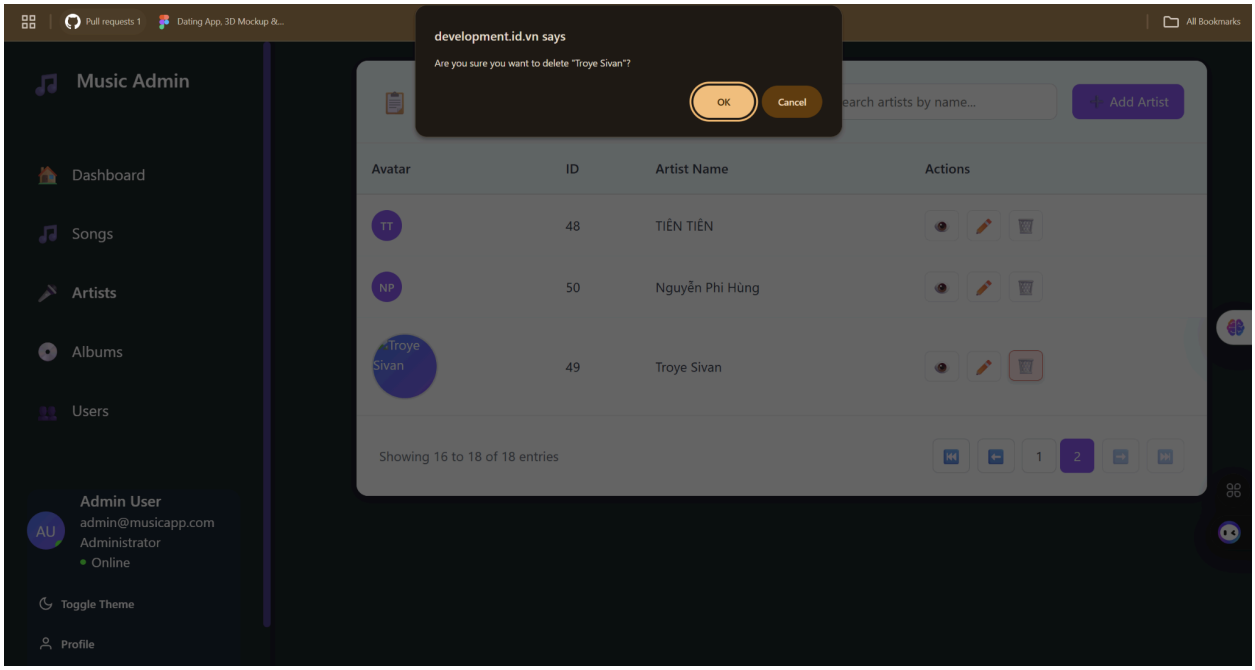
Read data: Fetch current artist data and user mapping.

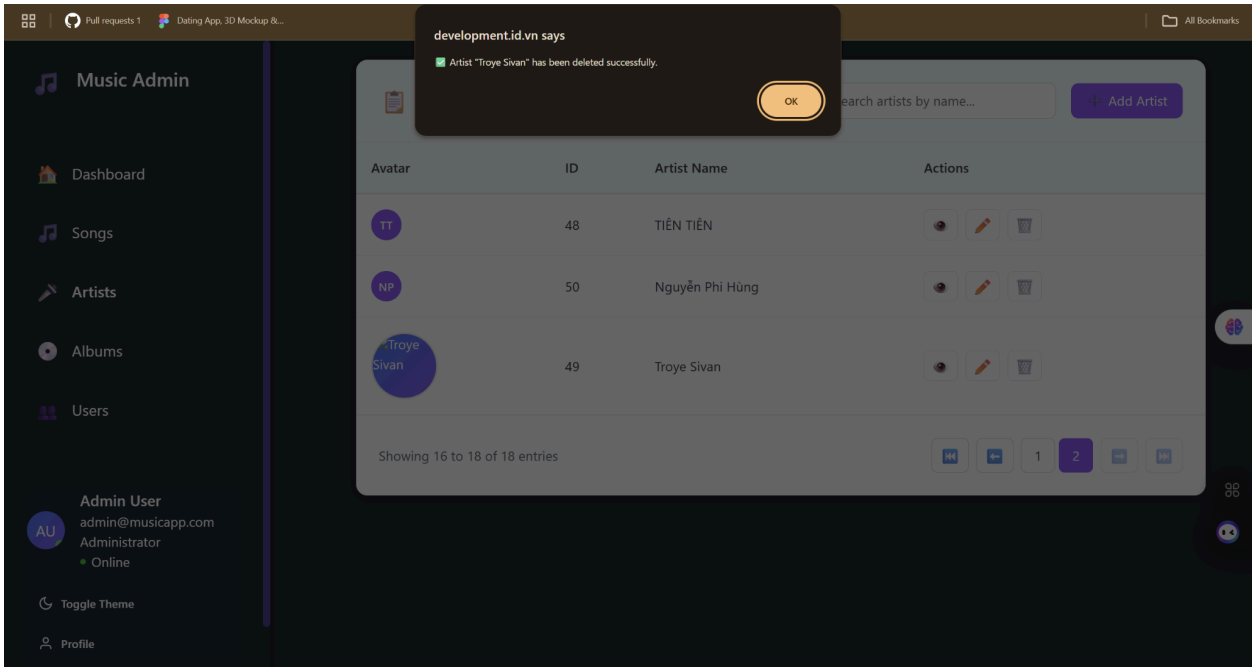
Write data: Update artist record with optional avatar image replacement and profile data updates.

Client-side States : formData, avatarFile, previewAvatar, isSaving, formErrors, successMsg, handleInputChange(), handleFileChange(), handleSaveArtist(), handleCancel()

- **Feature: Delete an artist**

***Description*:** Remove an artist from the system including their profile data, avatar image, and database records.





Libraries: multer, knex, fs

Server - side APIs:

- **DELETE /api/artists/:id:** Delete an artist

  Request Body: artist_id: integer

  Response: application/json

Read data: Fetch current artist data and user mapping.

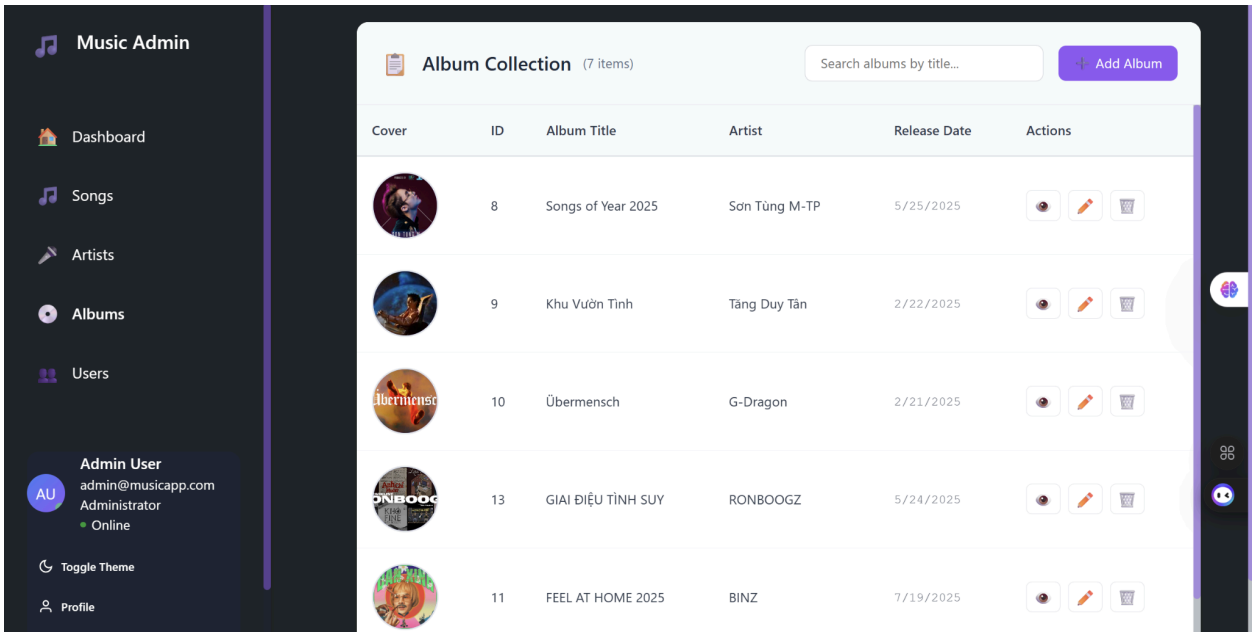
Write data: Remove artist entry and delete avatar if exists.

Client-side States : selectedArtist, confirmDelete, isDeleting, deleteError, successMsg,handleDeleteArtist(),handleOpenDeleteModal(), handleCancelDelete()

5. Albums

- Feature: View all albums

Description: Retrieve a list of all albums available in the system. This endpoint returns an array of albums, each including basic information such as album title, artist name, release date, cover image, and associated metadata. This endpoint is typically used by admins to browse and manage all albums in the music library.



Libraries: zod, knex

Server - side APIs:

- **GET /api/albums:** Retrieve a list of all albums in the system
  Request Body: None
  Response: application/json

Table: Artists, Albums

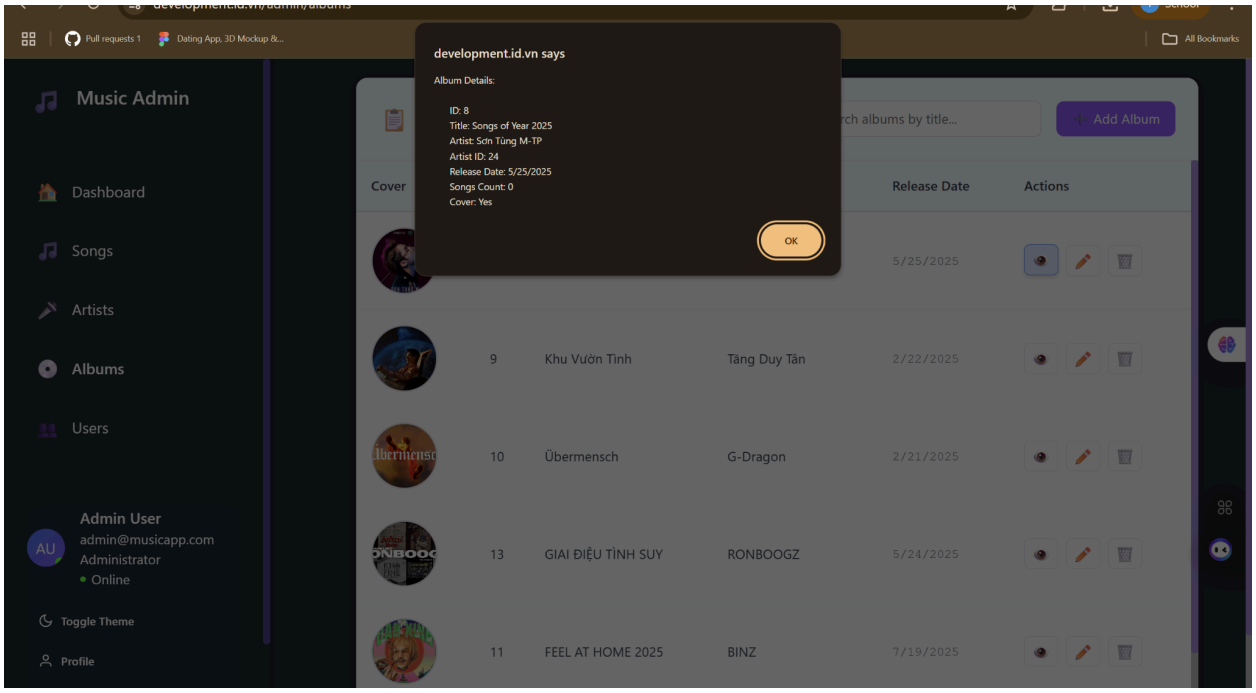
Read data: Retrieve data from the system to get all albums with:

+ Basic album information (ID, title, release date)
+ Associated artist information

Client-side States : albums, isLoading, error, selectedAlbum, searchTerm, handleSelectAlbum(), fetchAlbums()

- Feature: View an album

Description: Get detailed information about a specific album by ID, including full metadata, artist details, track listing, and cover information.



Libraries: knex, zod

Server - side APIs:

- **GET /api/albums/:id:** Get details of a specific album

  Request Body: album_id: integer

  Response: application/json

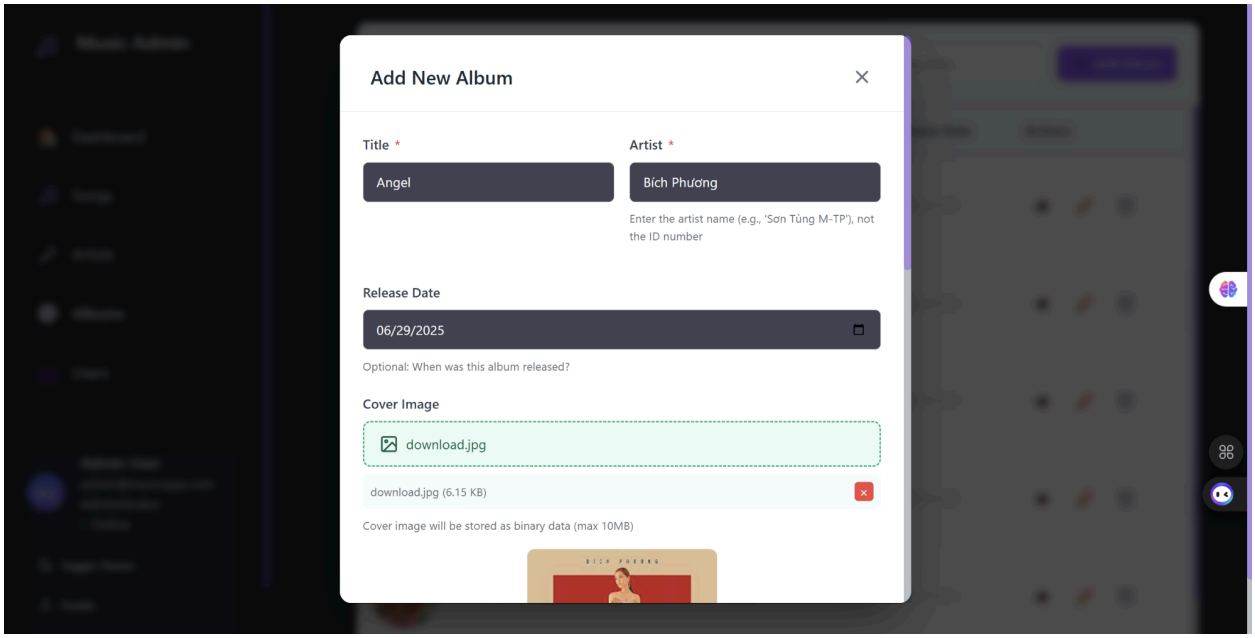
Tables: **Songs, Artists, Albums**

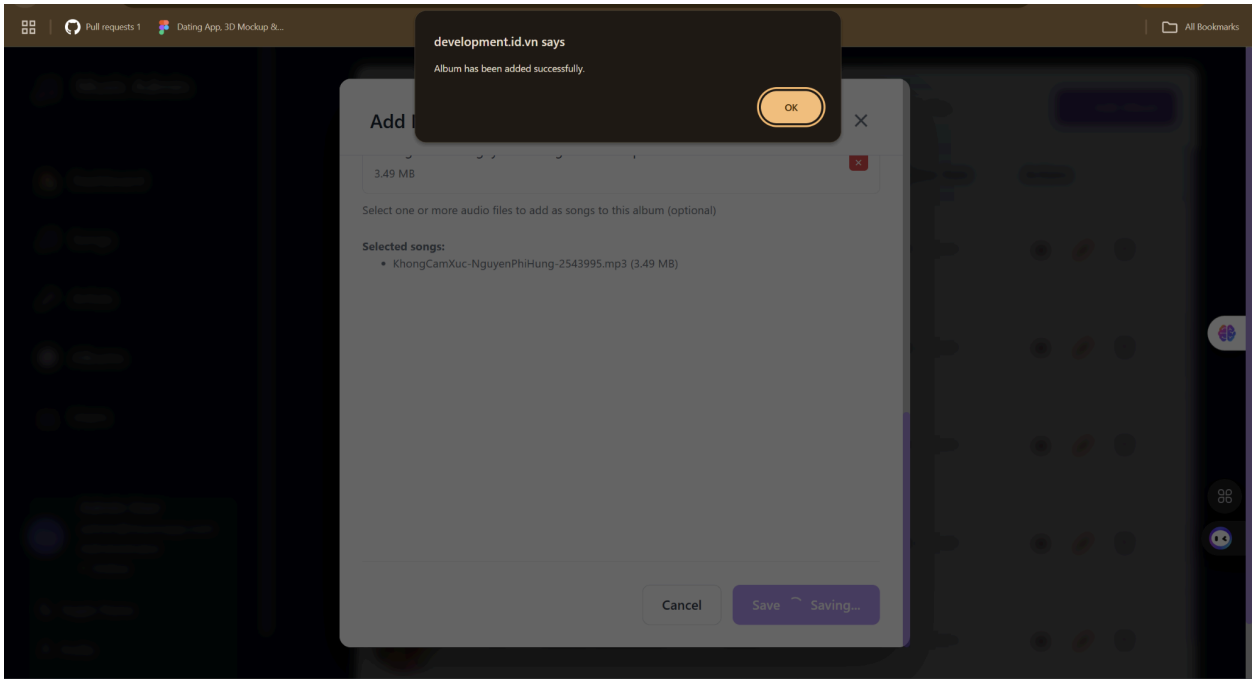
Read data: Retrieve detailed album information including artist relationships and track listings.

Client-side States : albumDetails, albumId, albumSongs, isLoading, error, artistInfo, trackList, fetchAlbumDetails(), router

- **Feature: Create an album**

***Description*:** Create a new album in the system. This endpoint allows admins to add an album with cover art upload and artist association. The system processes the cover image and stores both the album data and cover art in the database.





Libraries: knex, zod, multer

Server - side APIs:

- **POST /api/albums/:** Create a new album

  Request Body: FormData containing:

  + title: string (required)
  + artist: string (required)
  + release_date: date (optional)

+ cover: File (optional)

Response: application/json

Tables: **Albums, Artists**

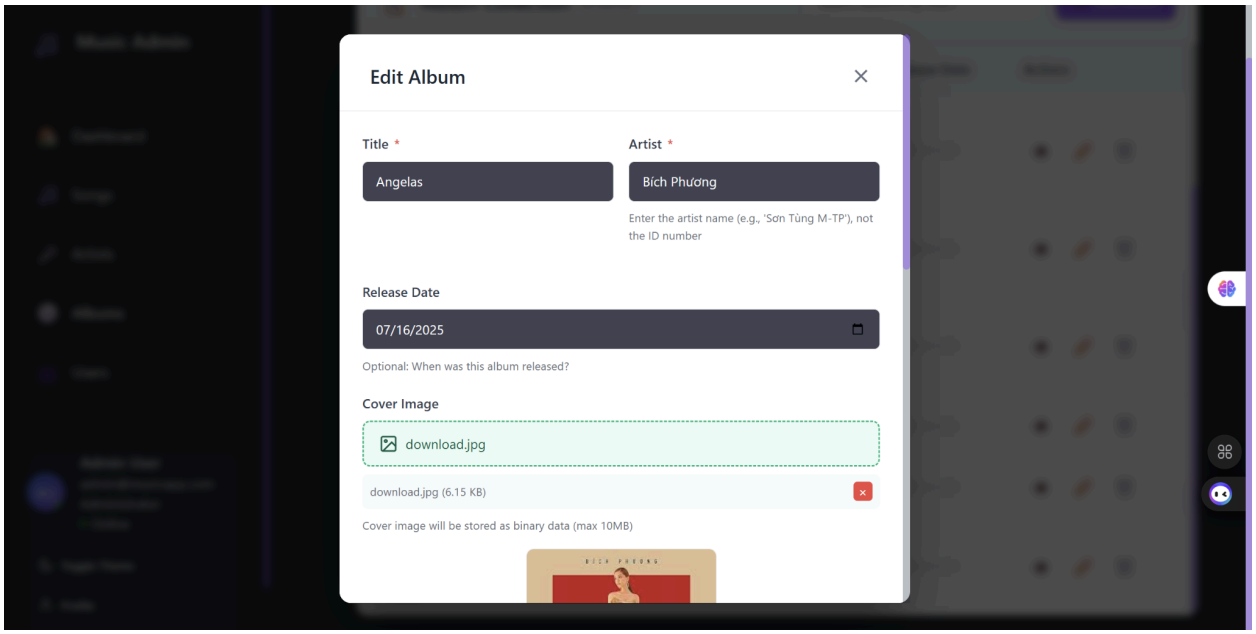
Read data: Retrieve artist list for selection before creation.

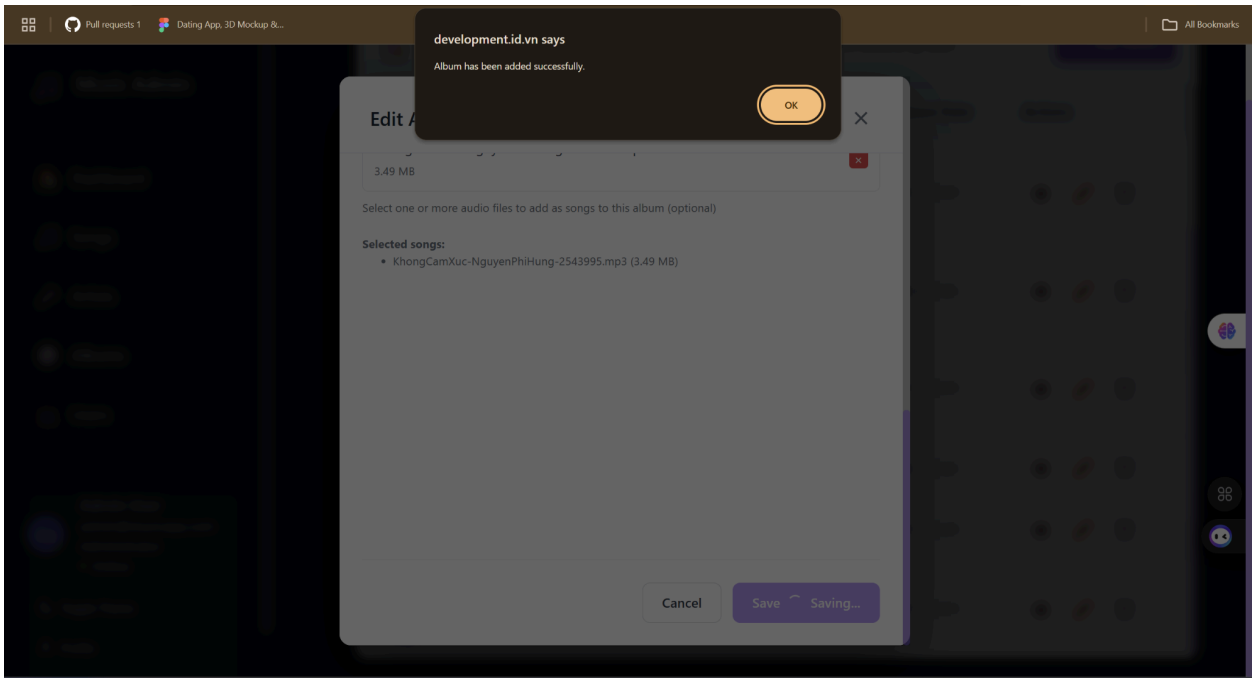
Write data: Insert new album record, upload and store cover image if provided.

Client-side States : albums, isLoading, error, selectedAlbum, searchTerm, handleSelectAlbum(), fetchAlbums()

- **Feature: Edit an album**

  Description**:** Update an existing album's information including title, artist, release date, and optionally replace the cover image. Supports partial updates where only changed fields are updated.





Libraries: knex, zod, multer

Server - side APIs:

- **PUT /api/albums/:id:** Update an existing album

  Request Body: FormData containing:

  + title: string (optional)
  + artist: string (optional)
  + release_date: date (optional)
  + cover: File (optional)

  Response: application/json

Tables: **Albums, Artists**

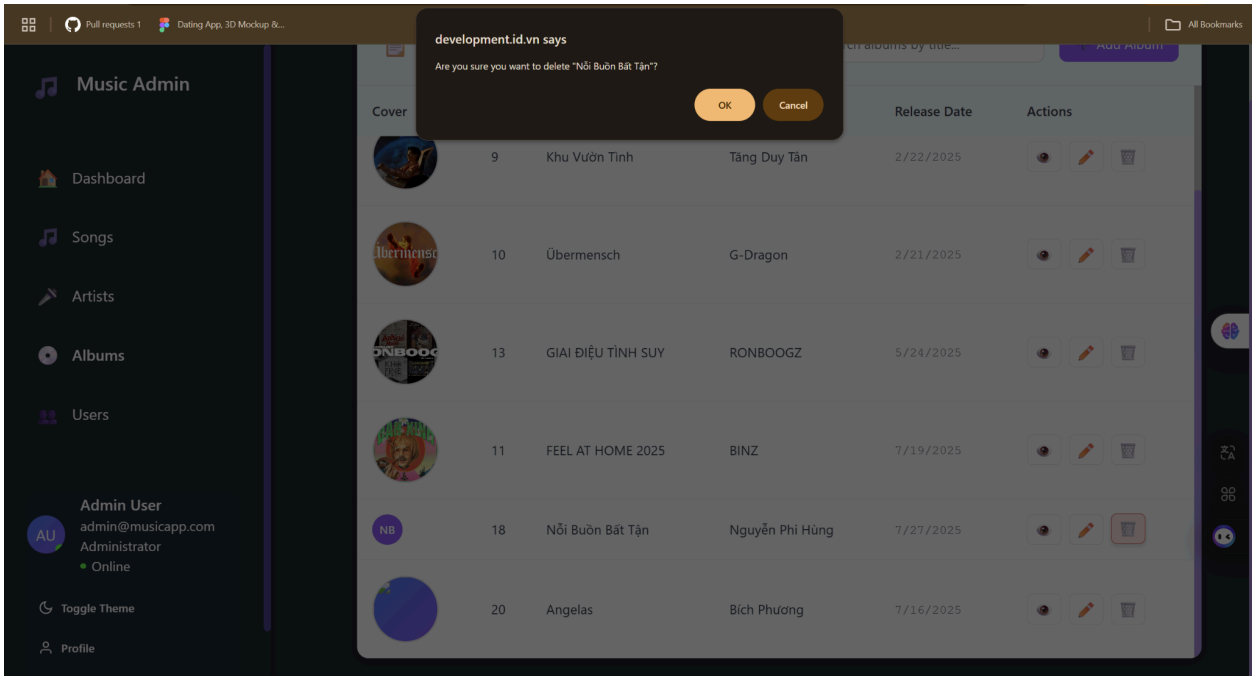
Read data: Fetch current album details and artist list.

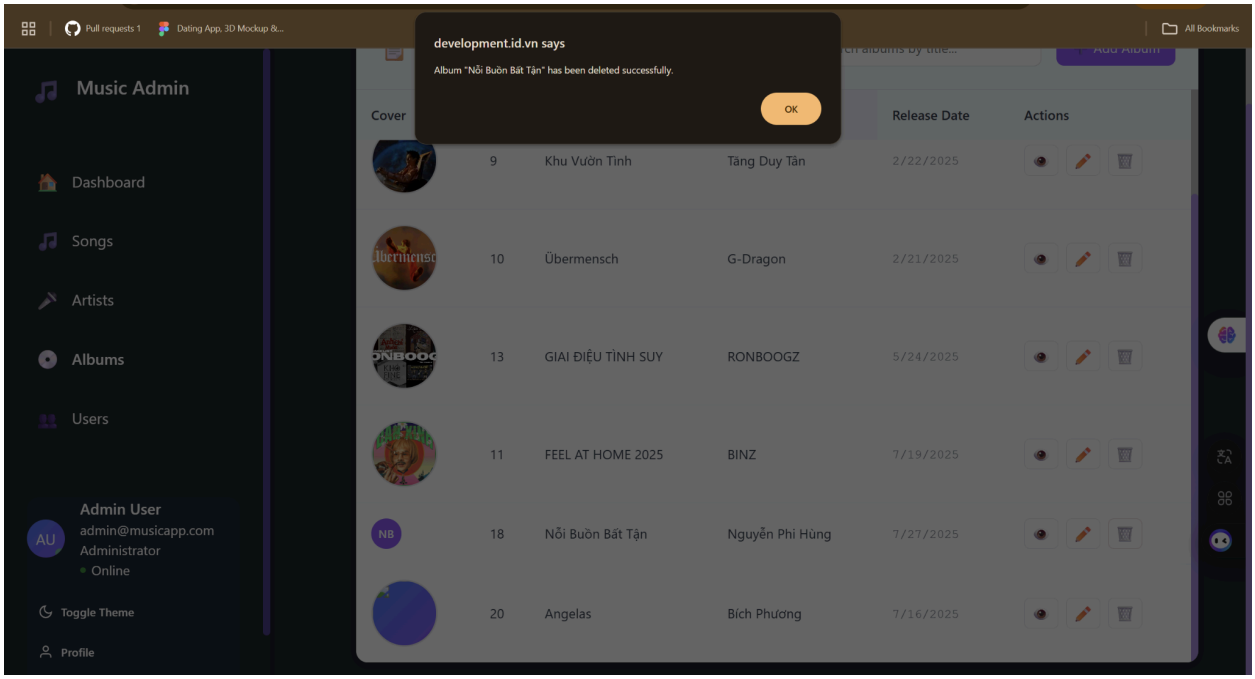
Write data: Update album record with optional cover image replacement and metadata updates.

Client-side States : albums, isLoading, error, selectedAlbum, searchTerm, handleSelectAlbum(), fetchAlbums()

- **Feature: Delete an album**

***Description*:** Remove an album from the system including its cover image and database records.





Libraries: knex, fs, multer

Server - side APIs:

- **DELETE /api/albums/:id:** Delete an album

  Request Body: album_id: integer

  Response: application/json

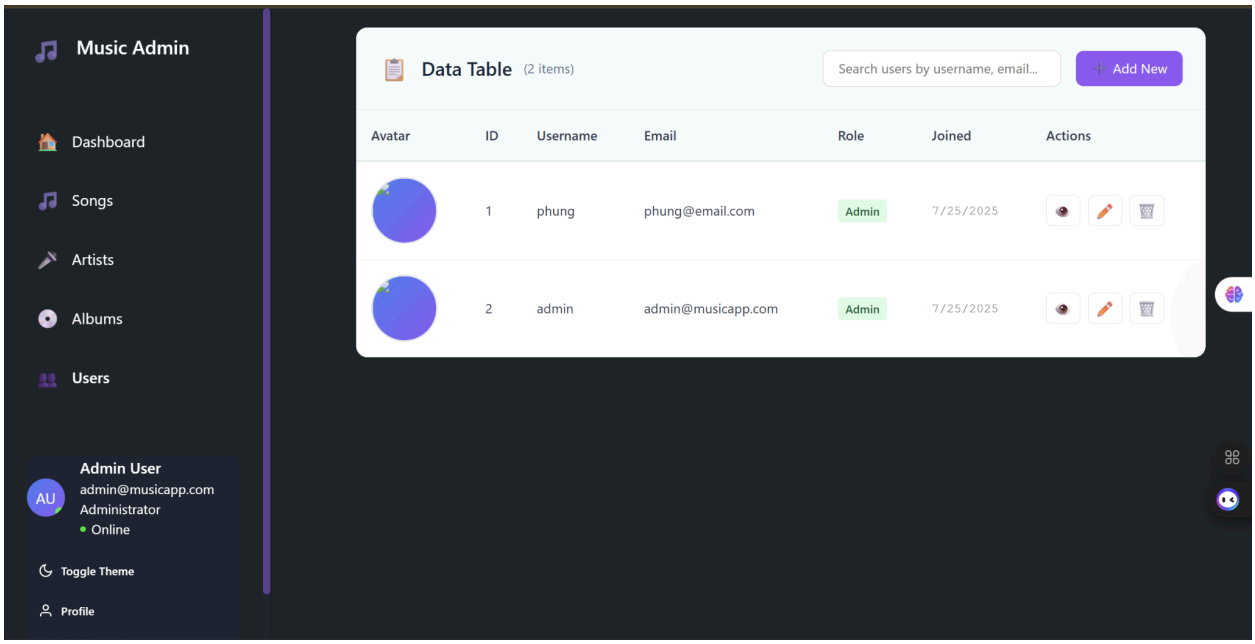
Read data: Confirm selected album before deletion.

Write data: Remove album from DB and delete cover image if exists.
Client-side States : selectedAlbum, confirmDelete, isDeleting, deleteError, successMsg,handleDeleteAlbum(),handleOpenDeleteModal(), handleCancelDelete()

6. Users

- Feature: View all users

Description: Retrieve a list of all users available in the system. This endpoint returns an array of users, each including basic information such as username, email, role, avatar image, and account status. This endpoint is typically used by admins to browse and manage all user accounts in the system.



Libraries: knex, zod
Server - side APIs:

- **GET /api/users:** Retrieve a list of all users in the system
  Request Body: None
  Response: application/json

Table: Users
Read data: Retrieve data from the system to get all users with:

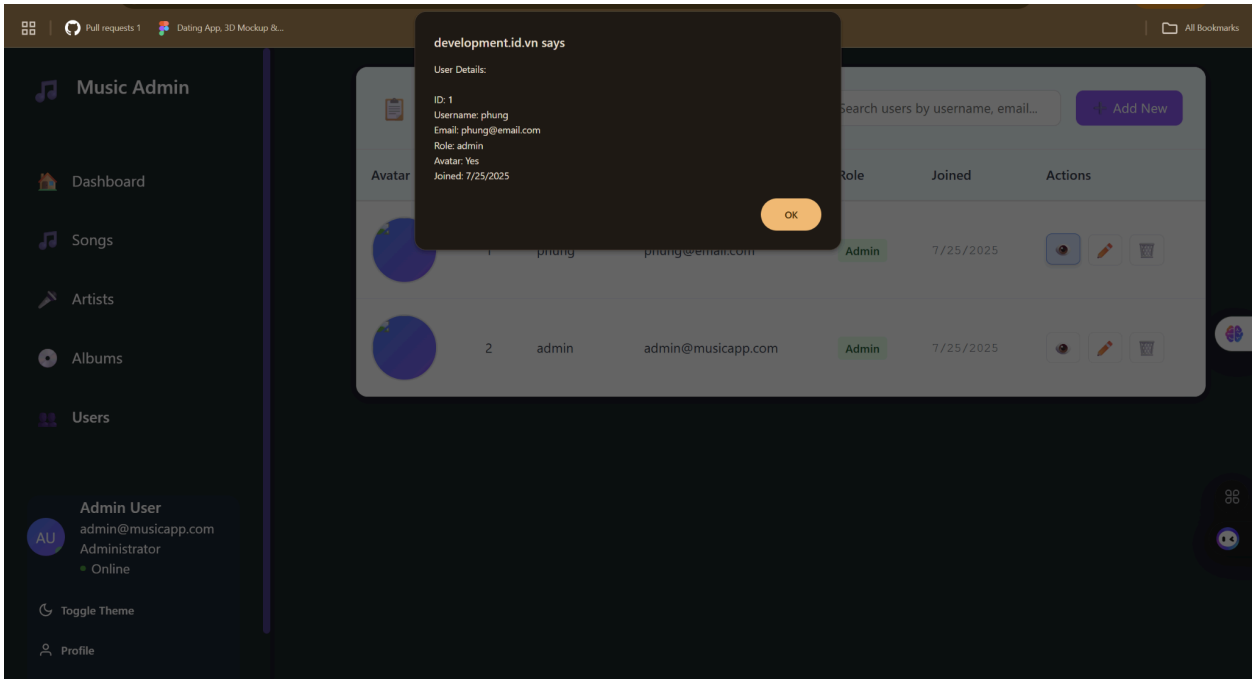
+ Basic album information (ID, username, email)
+ Role and permission levels
+ Avatar image paths
+ Account creation timestamps

Client-side States : users, selectedUser, isLoading, error, searchInput, sortField,

handleSelectUser(), fetchUsers()

- **Feature: View a user**

Description: Get detailed information about a specific user by ID.



Libraries: knex, fs

Server - side APIs:

- **GET /api/users/:id:** Get details of a specific user

  Request Body: user_id: integer

  Response: application/json

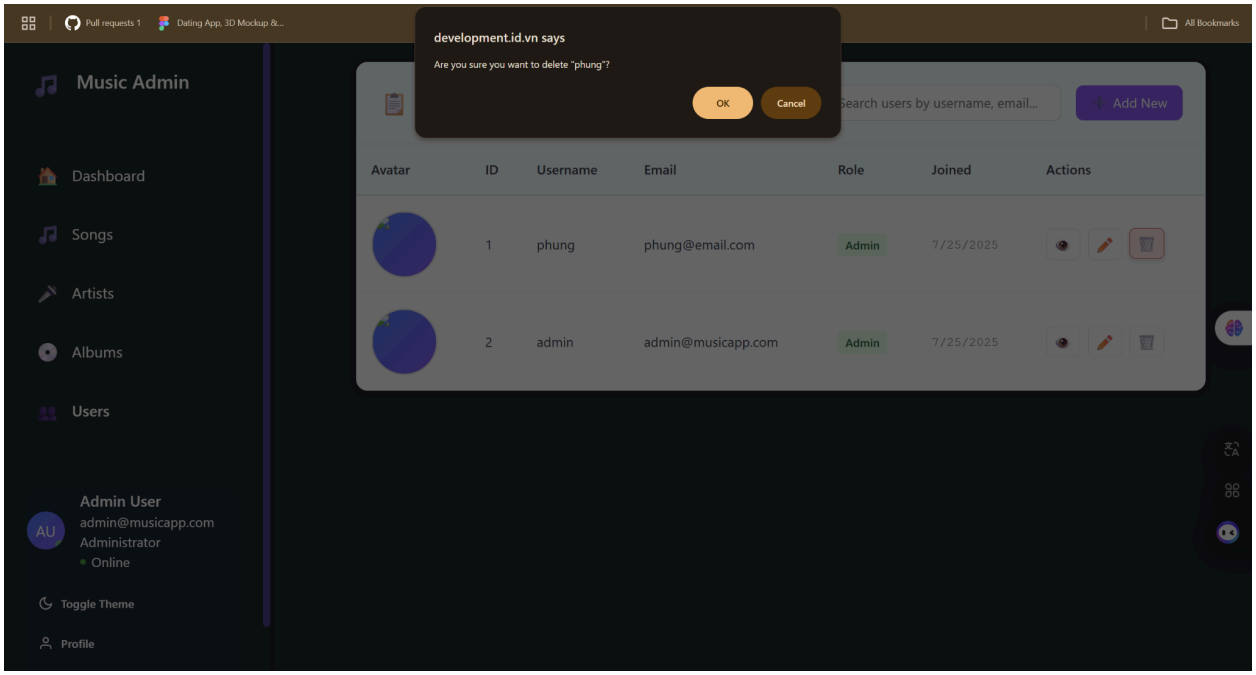
Tables: **Users**

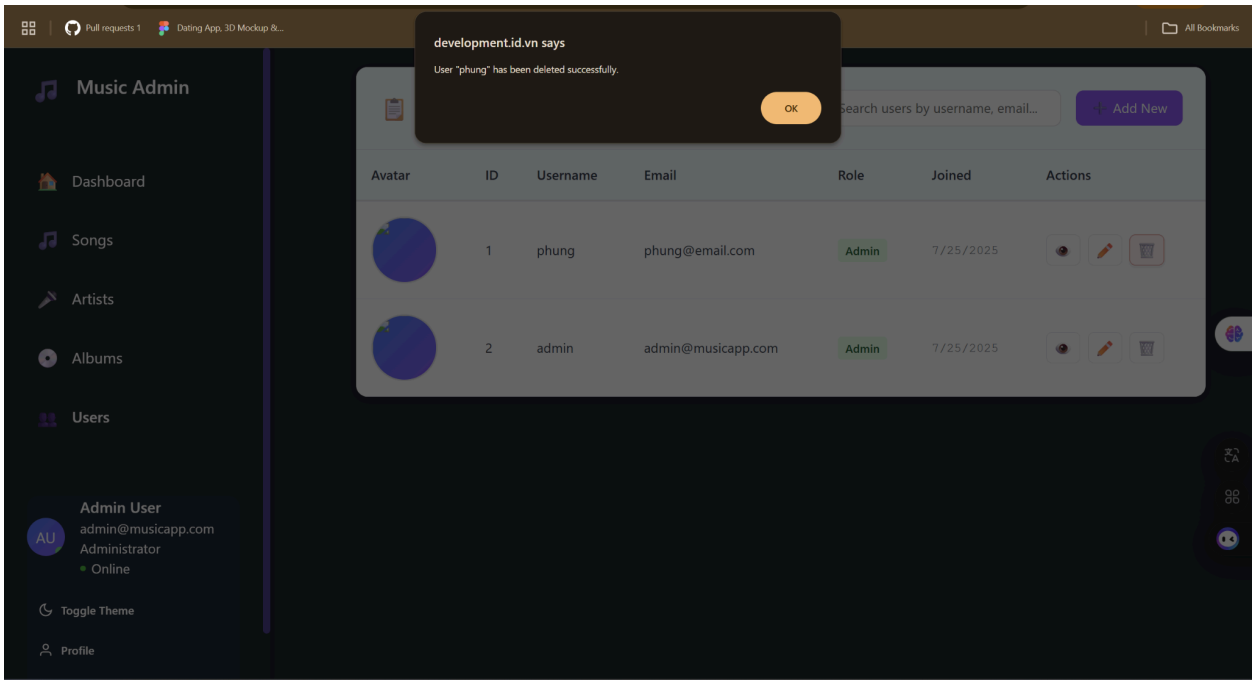
Read data: Retrieves user ID, username, email, role, avatar (true or false), and account creation timestamp

Client-side States : users, selectedUser, isLoading, error, searchInput, sortField, handleSelectUser(), fetchUsers()

- **Feature: Delete a user**

***Description*:** Remove a user account from the system including their profile data, avatar image, and associated records.





Libraries: knex, fs

Server - side APIs:

- **DELETE /api/users/:id:** Delete a user

  Request Body: user_id: integer

  Response: application/json

Read data: Fetch user details for confirmation before deletion.

Write data: Remove user record and associated avatar if exists.

Client-side States : selectedUser, userDetails, isLoading, error, fetchUserDetails(), router