<SoundJack>

Software Architecture Document

Version <1.1>

Revision History

| **Date** | **Version** | **Description** | **Author** |
| --- | --- | --- | --- |
| <24/11/2024> | <1.0> | Filled Introduction, Architecture goal and constraints, Use-case model and Logical view | Đỗ Lương Nhật Minh Võ Thị Hồng Minh Nguyễn Gia Nguyễn Nguyễn Lâm Nhã Uyên |
| <11/12/2024> | <1.1> | Update section 5 and 6 for PA4 | Võ Thị Hồng Minh Đỗ Lương Nhật Minh |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[**1. Introduction 4**](#_heading=h.gjdgxs)

[**2. Architectural Goals and Constraints 5**](#_heading=h.30j0zll)

[**3. Use-Case Model 6**](#_heading=h.1fob9te)

[**4. Logical View 7**](#_heading=h.3znysh7)

[4.1 Component: Front-End 7](#_heading=h.2et92p0)

[4.3 Component: External-Services 13](#_heading=h.htwv7scaiabf)

[4.4 Component: Application Server: 15](#_heading=h.qyjn23juspso)

[**5. Deployment 20**](#_heading=h.tyjcwt)

[**6. Implementation View 20**](#_heading=h.3dy6vkm)

Software Architecture Document

# Introduction

1. **Purpose**

The purpose of this document is to provide an overview of the software architecture for the "SoundJack", which is a music streaming project.Specifically, it describes architectural decisions, key modules, the components of the project. It is also used as a guide for development, testing, and maintenance teams to ensure system consistency and efficiency.

1. **Scope**

SoundJack is a music website that allows users to

* Listen to music
* Reaction to songs
* Create and manage playlist
* Upload music (for artist accounts)
* Search for songs, artists, albums
* Suggest songs based on users preferences

The system consists of three main components: client applications (web), a backend server, and the database.

1. **Definitions, Acronyms, and Abbreviations**

* **API (Application Programming Interface):** Interface for application interaction.
* **UI (User Interface):** Interface for user interaction.
* **UX (User Experience):** Overall experience of a user when interacting with a web.
* **DBMS (Database Management System):** Database management system.

1. **References**

*HTML / CSS tutorial*

<https://youtube.com/playlist?list=PL_-VfJajZj0U9nEXa4qyfB4U5ZIYCMPlz&si=km1IwdoAk3nfSn6q>

<https://youtu.be/wRNinF7YQqQ?si=OugMEIYrVs-r7VTb>

*MongoDB tutorial*

[*https://youtu.be/8Nx7cdwT86c?si=G7FRkGQ7aE4lbfJI*](https://youtu.be/8Nx7cdwT86c?si=G7FRkGQ7aE4lbfJI)

*Spotify Clone Using Mern Stack*

[*https://youtube.com/playlist?list=PLY7exrvAQSeuh1\_V-b4Sj-4Fhe03noob1&si=KxsBy-vx\_ggYH\_jH*](https://youtube.com/playlist?list=PLY7exrvAQSeuh1_V-b4Sj-4Fhe03noob1&si=KxsBy-vx_ggYH_jH)

1. **Overview**

This document consists of 6 sections:

* **Section 1:** Introduction
* **Section 2:** Architectural Goals and Constraints
* **Section 3:** Use-case model
* **Section 4:** Logical view
* **Section 5:** Deployment
* **Section 6:** Implementation view

# Architectural Goals and Constraints

1. **Performance:**

* Support up to 10,000 concurrent users with minimal latency.
* Ensure response time for user actions (searching, loading songs) does not exceed 2 seconds.
* Guarantee music tracks load within 3 seconds on standard broadband connections.

1. **Security:**

* Encrypt all user data using industry-standard protocols (e.g., AES-256).
* Implement multi-factor authentication (MFA) for enhanced account security.
* Define role-based access control for Guests, Logged-in Users, Artists, and Admins.

1. **Usability:**

* Provide a user-friendly and intuitive interface.
* Ensure WCAG compliance for accessibility to users with disabilities.
* Include help documentation (FAQs, manuals, tutorials) for user assistance.

1. **Reliability and Availability:**

* Achieve a minimum 99.9% uptime for consistent platform access.
* Implement fault-tolerant mechanisms to recover from failures without data loss.

1. **Scalability:**

* Enable horizontal scaling to manage increasing user demand.
* Utilize a Content Delivery Network (CDN) for efficient global content distribution.

1. **Compliance:**

* Adhere to copyright laws and licensing agreements for streamed or uploaded music.
* Comply with GDPR and CCPA for user data privacy and management.

1. **Design Constraints:**

* Technology Stack: Use HTML & CSS for frontend and Javascript, Node,js, Express for backend development.
* Platform Compatibility: Ensure compatibility with major web browsers (Chrome, Firefox, Safari) and mobile OS (iOS, Android).

1. **Documentation Requirements:**

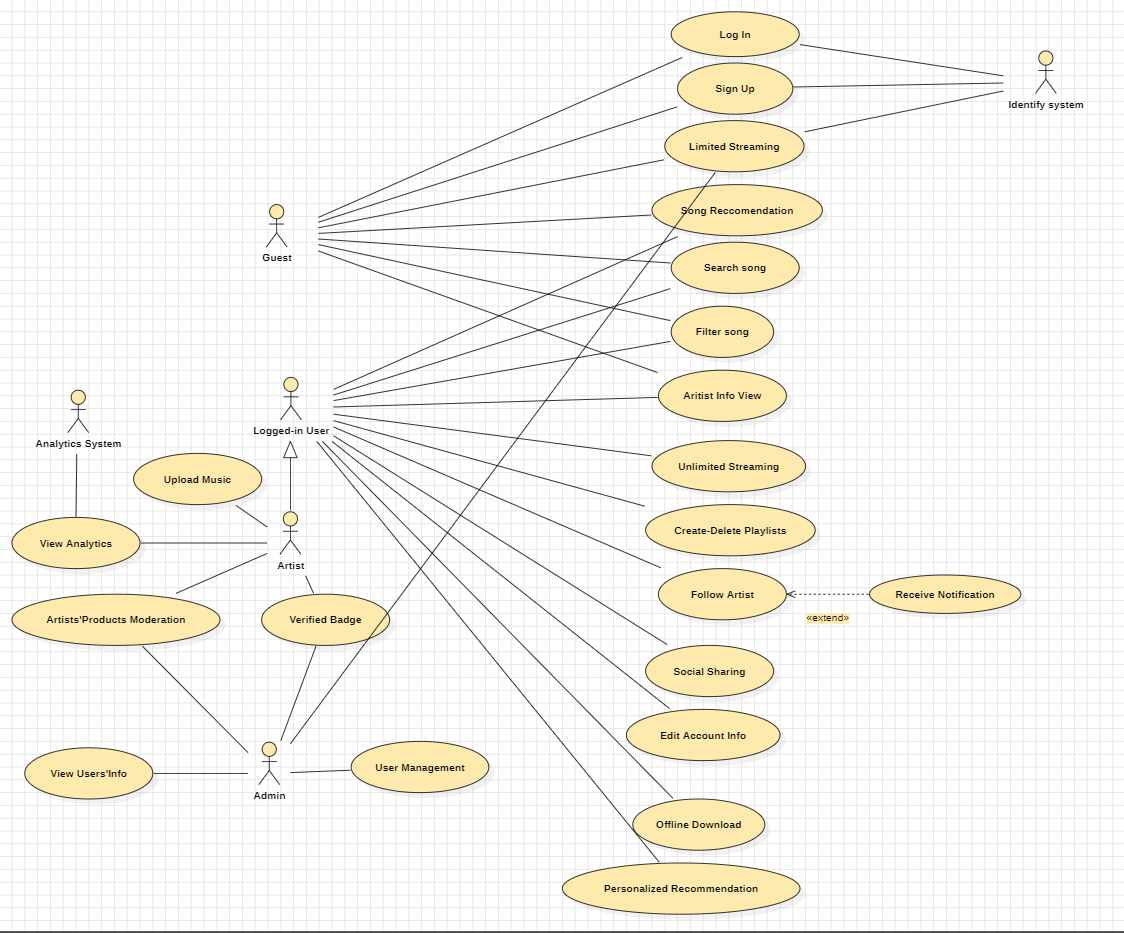
* Provide comprehensive user manuals detailing all features.
* Offer online help with FAQs and troubleshooting guides.
* Include installation instructions for any required components.

**k) Priority of Non-Functional Requirements:**

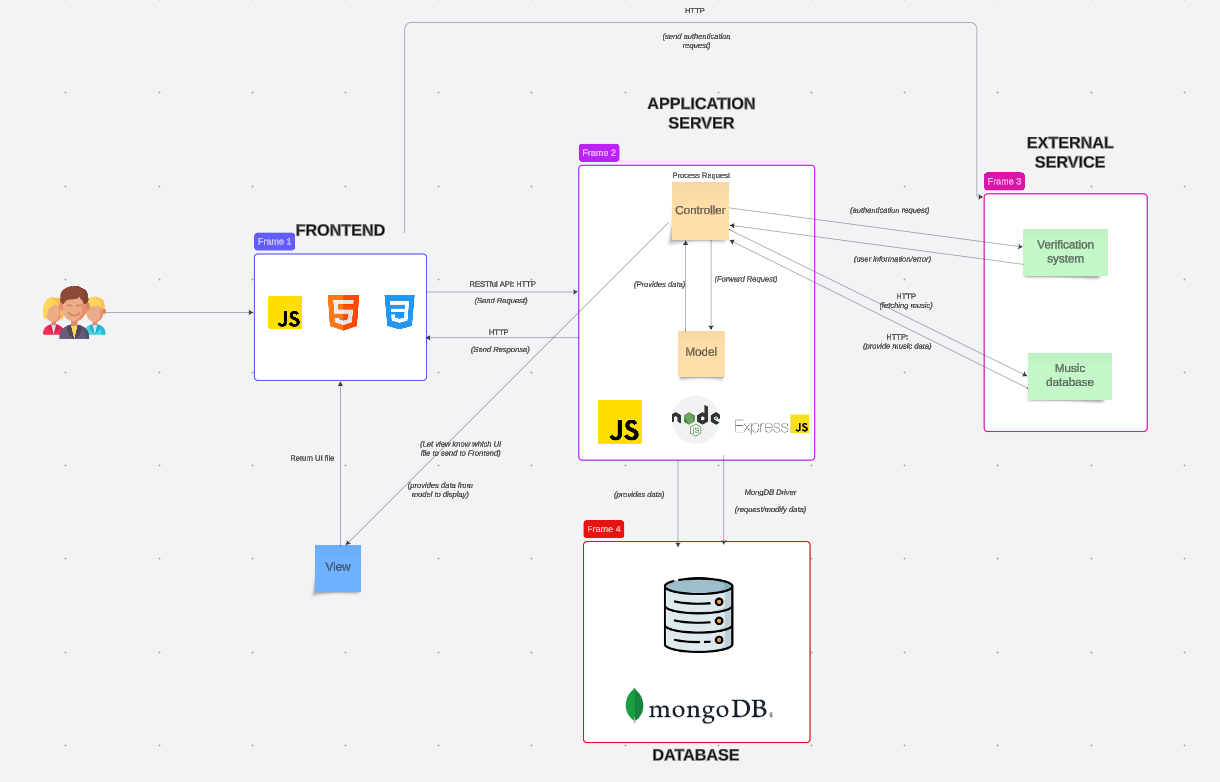
**Stability**: High priority for reliable operations under varying loads.

* **Benefit**: High priority as usability and security are critical for user retention.
* **Effort:** Medium priority, addressable in phases.
* **Risk:** High priority to prevent legal or operational repercussions from unmet compliance or security requirements.

# Use-Case Model

**

# Logical View

**

## Component: Front-End

The **frontend component** is responsible for providing an intuitive and responsive user interface for users to interact with the music streaming platform. It communicates with backend services to fetch and send data, render content dynamically, and ensure user experience

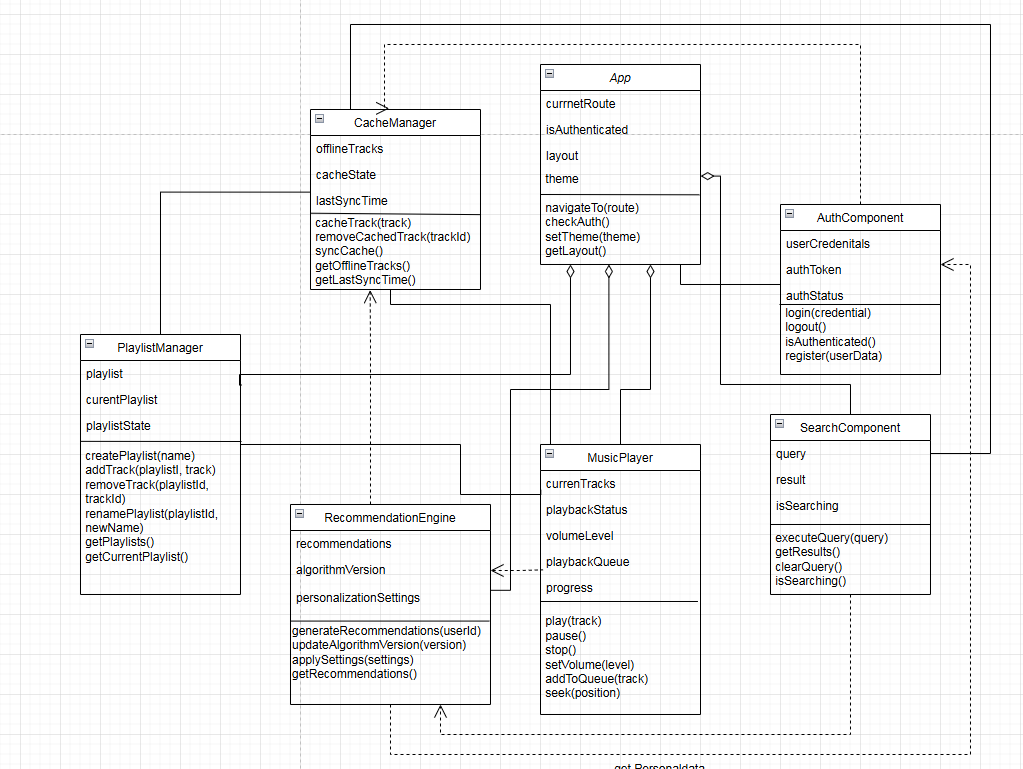
1. **Responsibilities and Services**

* **User Authentication and Authorization**
  + Allow users to log in, register, and manage their accounts securely.
  + Service Communication: **HTTPS** to Backend Authentication API.
* **Music Search and Discovery**
  + Enable users to search for tracks, albums, or artists.
  + Provide recommendations based on preferences and browsing history.
  + Service Communication: **HTTPS** to Search and Recommendation APIs.
* **Media Playback Control**
  + Offer controls like play, pause, skip, repeat, and volume adjustment.
  + Display song metadata, progress bars, and album art.
  + Service Communication: **Socket** for real-time media streaming and control updates.
* **Playlist Management**
  + Allow users to create, update, and delete playlists.
  + Provide the ability to like/dislike tracks or add/remove them from playlists.
  + Service Communication: **HTTPS** to Playlist Management API.
* **Offline and Cache Management**
  + Allow downloading tracks for offline access.
  + Manage caching for faster reloading of frequently accessed data.
  + Service Communication: **HTTPS** and Local Storage.

1. **Key Classes:**

* **App**:
  + The root component managing routes and layout.
  + Interacts with other classes to render the main UI.
* **AuthComponent**:
  + Handles login, registration, and authentication states.
  + Interfaces with the Authentication API.
* **MusicPlayer**:
  + Core class managing playback controls and audio rendering.
  + Includes subcomponents for controls, album art, and progress display.
* **SearchComponent**:
  + Handles user input for search queries and displays search results.
  + Communicates with the Search API.
* **PlaylistManager**:
  + Allows playlist creation, updates, and management.
  + Interfaces with Playlist APIs.
* **RecommendationEngine**:
  + Displays recommended tracks or albums.
  + Utilizes machine learning APIs for suggestions.
* **CacheManager**:
  + Manages local storage for offline tracks and cached data.
  + Synchronizes with the backend to refresh data periodically.

**UML:**



* 1. **Component Database**

The **database component** is responsible for securely storing, managing, and retrieving all critical data for the music streaming platform, including user information, track metadata, playlists, recommendations,...It ensures data consistency, scalability, and fast query execution. The database communicates with backend services through APIs using HTTPS and integrates with caching mechanisms to enhance performance and support offline functionalities.

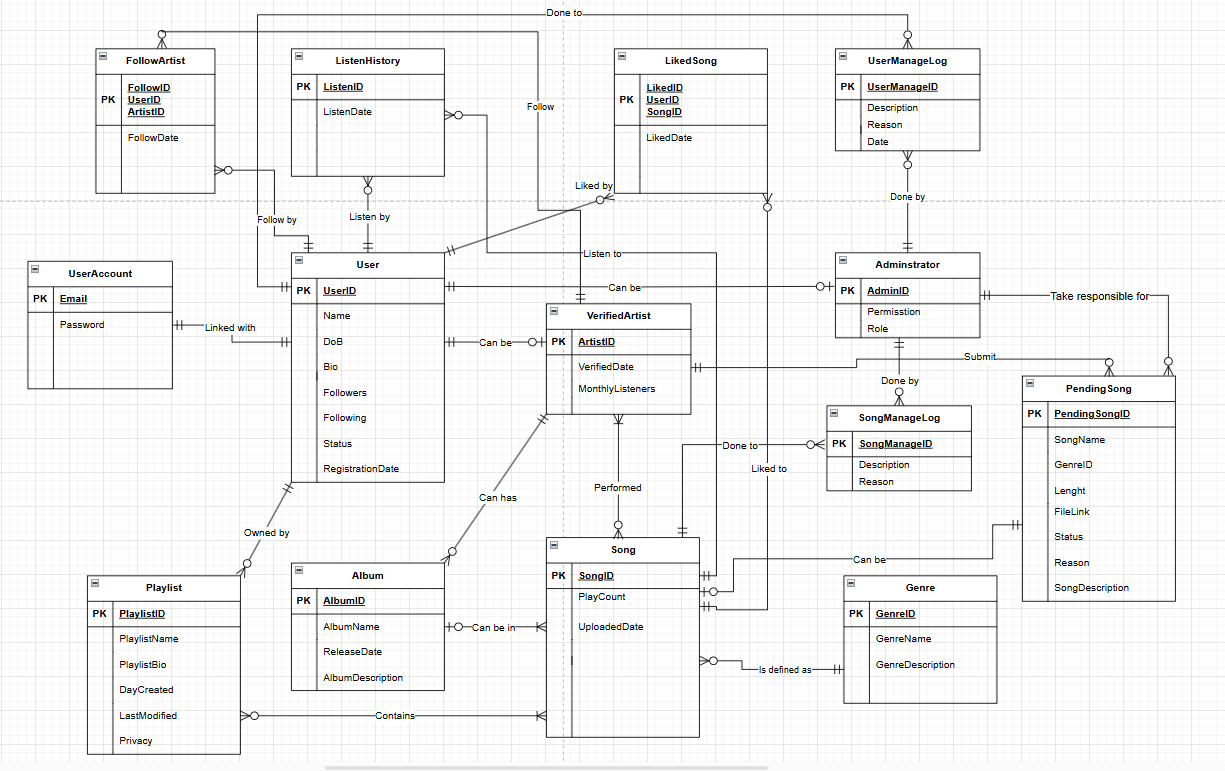
1. **Responsibilities and Services**

* **User Data Management**
  + Store and manage user details such as login credentials, preferences, subscription plans, and profile information.
  + Service Communication: HTTPS with Authentication API and User Profile API.
* **Music Data Management**
  + Maintain metadata about tracks, including title, artist, album, genre, release date, and streaming URL.
  + Enable efficient search and retrieval of tracks.
  + Service Communication: HTTPS with Media Streaming API and Search API.
* **Playlist Data Management**
  + Store user-created playlists with associated tracks and metadata.
  + Provide CRUD operations for playlists.
  + Service Communication: HTTPS with Playlist Management API.
* **Recommendations Data**
  + Store and retrieve personalized song recommendations for users.
  + Log algorithm versions and timestamps for analytics.
  + Service Communication: HTTPS with Recommendation API.
* **Caching and Offline Storage**
  + Maintain temporary data like cached queries and offline tracks.
  + Manage expiration and synchronization with the backend.
  + Service Communication: Local Storage, HTTPS with Cache Manager.
* **Analytics Logging**
  + Log user actions like playback, like, follow for tracking and reporting.
  + Provide insights for recommendations and performance monitoring.
  + Service Communication: HTTPS with Analytics Engine and Recommendation API.

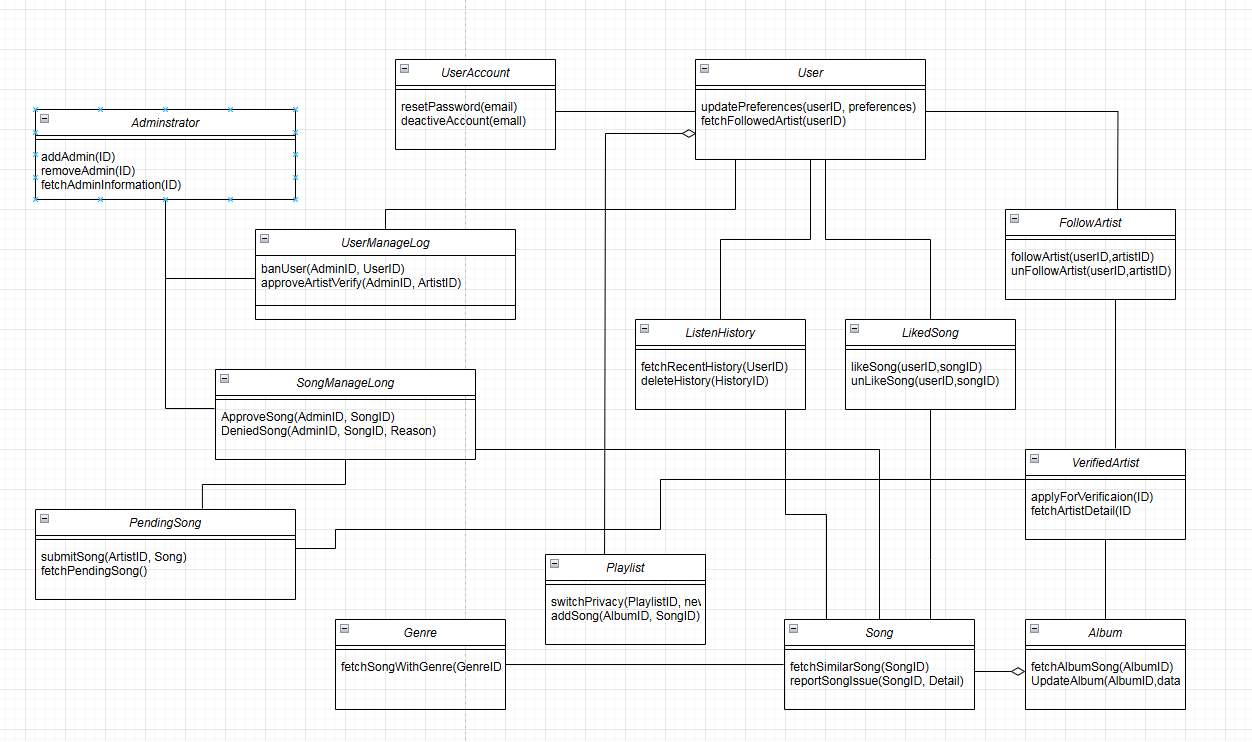
1. **Key Tables (Classes):**

* **UserAccount**
  + **Responsibilities: Manage user login credentials and account-related metadata.**
  + **Interactions: Authentication API, User Management System.**
* **User:** 
  + **Store user profile information and preferences.**
  + **Interactions: User Profile API, Playlist API, Recommendation API.**
* **Verified Artist**
  + Manage information about verified artists and their public profiles.
  + **Interactions**: Artist Verification System, Song API, Album API.
* **Song**
  + Store metadata for individual songs and provide streaming information.
  + **Interactions**: Media Streaming API, Playlist Management API, Search API.
* **Album**
  + Store album details and manage relationships with songs.
  + **Interactions**: Song API, Media Streaming API.
* **Playlist**
  + Manage user-created playlists and associated tracks.
  + **Interactions**: Playlist Management API, User Database.
* **Genre**
  + Organize and classify songs into genres for search and recommendations.
  + **Interactions**: Search API, Recommendation API.
* **FollowArtist**
  + Manage relationships between users and the artists they follow.
  + **Interactions**: Artist API, Recommendation API.
* **ListenHistory**
  + Track user listening activity for recommendations and analytics.
  + **Interactions**: Analytics Engine, Recommendation API.
* **LikedSong**
  + Track songs liked by users for personalized recommendations.
  + **Interactions**: Recommendation API, User Profile API.
* **UserManageLog**
  + Log actions performed by administrators on user accounts.
  + **Interactions**: Administrator Dashboard, User Management System.
* **Administrator**
  + Manage administrative user information and roles.
  + **Interactions**: Administrator Dashboard, User and Song Management APIs.
* **Pending Song**
  + Track songs submitted by artists for review and approval.
  + **Interactions**: Song Management API, Administrator Dashboard.
* **SongManageLog (For administrator accept new music)**
  + Log administrative actions related to managing songs, such as approvals or removals.
  + **Interactions**: Administrator Dashboard, Song Management API.

**ERD:**

****

**UML (Class’s attributes is the same as Entity’s attributes)**

****

## Component: External-Services

The **External Services** component facilitates seamless interaction between the music streaming platform and third-party services such as Spotify's music database and external verification systems. It ensures secure and efficient communication, manages authentication, and provides necessary data transformations to integrate external services into the application workflow.

### **Responsibilities and Services**

#### **Music Database Integration**

#### Fetch music metadata, including track, album, artist, playlist details, and streaming URLs.

#### Manage user-specific data like saved tracks, playlists, and preferences.

#### Enable search and real-time streaming functionalities.

#### Service Communication: HTTPS with Spotify API (Music Metadata API, Playback API), HTTPS with Controller (Search Music, Fetch Track Metadata, Manage Playlists, Play Music,Track Playback Status)

#### **Verification System**

* + Handle user and application authentication using OAuth 2.0.
  + Manage and refresh access tokens for secure interaction with Spotify.
  + Validate third-party requests and ensure secure access to external resources.
  + Service Communication: HTTPS with Spotify Accounts API and Identity Provider APIs. HTTPS with Controller (User Authentication, User Registration, Token Validation, Password Reset, Session Management, Role-Based Access Control, Two-Factor Authentication, Session Expiry and Renewal)

#### **SystemError Handling and Rate Limiting**

#### Monitor and handle errors from external APIs (e.g., invalid tokens, rate limits).

* + Implement retry mechanisms and fallback strategies.
  + Service Communication: HTTPS with Spotify API Error Handling Endpoints.

#### **Third-Party Service Integration**

#### Abstract and manage interactions with additional third-party APIs, such as lyrics providers or metadata enrichment services.

#### Service Communication: HTTPS with Lyrics API or Metadata API.

#### **Data Transformation**

#### Standardize and format external data to match internal database schema.

#### Parse complex responses into structured data for backend services.

#### Service Communication: Internal APIs (Backend Service API).

#### **Key Tables (Classes)**

#### **ExternalService**

#### A base class for managing communication with external APIs.

#### Handles authentication, API request/response, and error management.

#### **SpotifyService**

#### Extends ExternalService to manage music-related interactions via the Spotify API.

#### Retrieves track details, user playlists, and streaming URLs.

#### Provides integration for track metadata and playback functionalities.

#### **VerificationService**

#### Extends ExternalService to handle user authentication and token validation with external identity providers (e.g., OAuth).

#### Ensures secure access by validating tokens and managing user sessions.

#### **MusicMetadata**

#### Represents a track's metadata, including title, artist, album, and genre.

#### Interacts with SpotifyService to fetch additional details and streaming information.

#### **Artist**

#### Models an artist or band, including their name and biography.

#### Communicates with SpotifyService to retrieve related information like tracks and albums.

#### **Track**

#### Represents an individual music track with attributes like duration, genre, and release date.

#### Uses SpotifyService to fetch streaming URLs and other track metadata.

#### **Playlist**

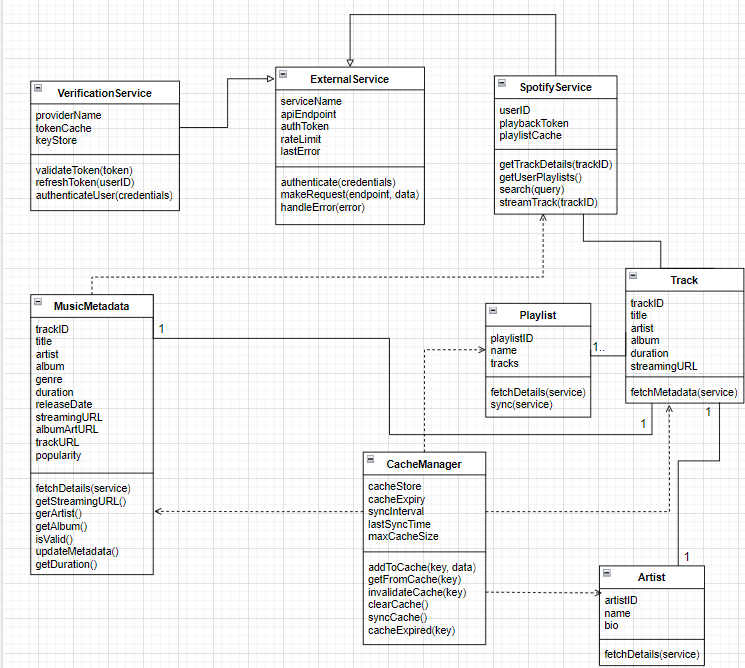
#### Represents a user-created playlist, containing a collection of tracks.

#### Supports CRUD operations and synchronizes data with SpotifyService.

#### **CacheManager**

#### Manages temporary data storage for cached API responses and offline functionality.

#### Synchronizes periodically with external services to ensure data freshness.



## Component: Application Server:

In general, this is the core processing unit for handling requests from the frontend and coordinating data exchange with the database and external services. In this implementation, the Application Server consists of two main elements: Controller and Model. Below is a detailed breakdown of these two elements.

**Controller:**

The Controller acts as an intermediary between the Frontend and the models. It processes incoming HTTP requests, applies business logic, and determines the appropriate response. The controller is divided into smaller classes/modules, each focused on specific functionalities.

* ***Auth Controller:***

The AuthController is responsible for managing user authentication-related tasks. It processes requests for user login, registration, and logout. This controller ensures that only authorized users can access certain parts of the system and helps manage user sessions or tokens. It uses the UserModel to interact with the database and verify user credentials during login and to register new users during the registration process.

*Responsibilities:*

* Authenticating users based on their provided credentials (e.g., username and password).
* Handling the registration of new users, including validating the provided information and saving user data to the database.
* Managing user sessions or generating tokens for authenticated users.
* This controller ensures that the application’s authentication logic is centralized and consistent, providing secure access to the system’s features.

*Methods:*

* register(): Handles user registration. It receives input from the user (e.g., email, password) and calls UserModel to create a new user in the database.
* login(): Handles user login. It receives login credentials (e.g., email, password), validates them through UserModel, and returns a response.
* validateToken(): Verifies the validity of the provided authentication token. It interacts with dbService to check the user's status.

*Inputs:*

* UserModel: Contains user-related data such as email, password, and other user details.
* dbService: Used to check if the user exists, validate passwords, and store user data in the database.
* AuthenticationService: Used to generate and validate authentication tokens (used in both the login() and validateToken() methods).
* ***Music Controller:***The MusicController manages the music-related functionality of the application, including searching for songs, managing playlists, and interacting with external music services like YouTube and Spotify. This controller acts as a middleman between the user’s requests related to music and the corresponding data models and services that handle the actual retrieval and storage of music data.

*Responsibilities:*

* Handling user requests related to music, such as searching for songs or artists, and fetching results from external services or the database.
* Managing playlists, allowing users to create, update, and delete playlists.
* Interacting with external music services to fetch or stream music content.
* The MusicController ensures that all music-related actions in the application are properly handled and makes sure that the user experience with music content is seamless and intuitive.

*Methods:*

* searchMusic(): Handles searching for music. It receives search parameters (e.g., song name, artist) and interacts with dbService to retrieve matching songs from the database.
* playSong(): Handles playing a song. It receives the song's ID and fetches the corresponding song data for playback.
* getSongInfo(): Retrieves detailed information about a specific song, such as title, artist, duration, and other metadata.

*Inputs:*

* songModel: Contains data for searching music (e.g., song title, artist, genre).
* dbService: Used to check if the user exists, validate passwords, and store user data in the database.
* ***User Controller:***

The UserController is responsible for handling user-related actions, such as updating user information, viewing profiles, and managing actions like changing passwords or updating account details. It interacts with the UserModel to perform these changes and store the data in the database.

*Responsibilities:*

* Handle requests for updating personal information (name, email, profile picture) from users.
* Provide APIs for users to view their profile information.
* Support actions like password changes and account updates.
* Manage user-related requests via various APIs.

*Methods:*

* getUserProfile(): Fetches the profile information of a logged-in user. Retrieves the user's profile data from the database using UserModel.
* updateUserProfile(): Allows the user to update their personal information (e.g., name, email, profile picture). Validates the new data and calls UserModel to update the database.
* changePassword(): Purpose: Allows a user to change their password. Validates the current password and updates the new password in the database via UserModel.

*Inputs:*

* UserModel: Contains user data such as credentials.
* dbService: Used to interact with the database to retrieve or save user data.
* ***Playlist Controller:***

The PlaylistController is responsible for managing user playlists. This controller handles requests like creating new playlists, adding/removing songs from playlists, and deleting playlists. It uses the PlaylistModel to manipulate playlist data in the database.

*Responsibilities:*

* Handle requests to create new playlists for users.
* Add and remove songs from the user's playlists.
* Handle requests to view the user's playlist collection.
* Delete or update playlist information.

*Methods:*

* createPlaylist(): Takes user input for a new playlist (e.g., playlist name, description) and calls PlaylistModel to create and store the playlist.
* getPlaylists(): Retrieves the user’s playlists by interacting with PlaylistModel and returns them to the frontend.
* updatePlaylist(): Update playlist. Call dbService to save this change

*Inputs:*

* PlaylistModel: Contains playlist-related data such as name, description, and songs.
* dbService: Used to retrieve or save playlist data in the database.

**Model:**

* ***User Model:***

*Description:*

The UserModel is a model dedicated to managing user data in the application. It is responsible for querying and modifying user-related information, including managing login credentials, profile details, and user preferences. The UserModel interacts directly with the database to retrieve, save, and update user data, including handling sensitive information like passwords.

*Responsibilities:*

* Querying the database for user information, such as verifying login credentials.
* Creating new user records in the database during user registration.
* Updating user details such as profiles or settings.
* Deleting or modifying user data when required (e.g., during account deletion or password reset).
* The UserModel provides the logic necessary for managing users and ensures that the user-related operations are handled securely and efficiently.

*Methods:*

* createUser(): Creates a new user in the database based on provided user data (email, password, etc.).
* getUser(): Searches the database to find a user by email or username.
* updateUser(): Searches the user on the database and modifies.

*Inputs:*

* userData: Contains the user's details such as email, password, and other profile information.
* dbService: Used to query and interact with the database for user-related operations.
* ***Song Model:***

The SongModel is responsible for managing song data, including retrieving music information from the database or external services like YouTube and Spotify. This model encapsulates the logic for fetching songs, saving new songs, updating song metadata, and deleting songs. It allows for easy integration with external music services and ensures that the application’s music database is kept up-to-date.

*Responsibilities:*

* Querying the database for song details, such as title, artist, and genre.
* Inserting new songs into the database when new music is added.
* Updating song metadata when necessary (e.g., updating song title or artist details).
* Deleting songs from the database or handling the removal of music records.
* The SongModel works closely with the MusicController to provide users with music content, ensuring that the application’s music data is accurate and up-to-date.

*Methods:*

* createSong(): Adds a new song to the database, including metadata like artist, genre, and song title.
* getSongInfo(): Retrieves song details from the database based on song name, artist, or other criteria.
* updateSong(): Updates song information in the database.

*Inputs:*

* songData: Contains song-related information like title, artist, genre, and any other necessary details.
* dbService: Used to perform CRUD operations (Create, Read, Update, Delete) on song data in the database.
* ***Playlist model:***

The PlaylistModel handles all database interactions related to playlists, such as creating new playlists, adding or removing songs, and managing relationships between songs and playlists. This model ensures that playlist data is stored accurately in the database.

*Responsibilities:*

* Store and retrieve playlist data from the database.
* Add and remove songs from playlists.
* Handle creation, deletion, or updating of playlists.
* Establish relationships between users and their playlists in the database.

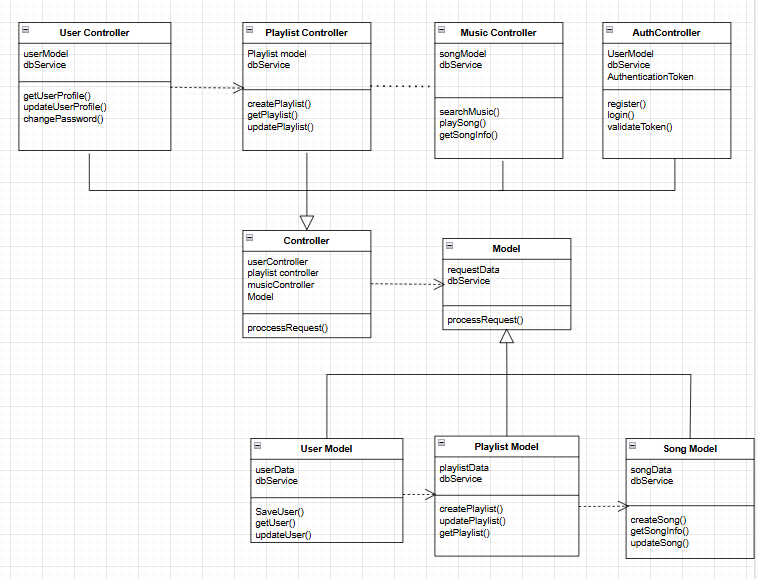
*Methods:*

* **createPlaylist()**: Creates a new playlist for the user and adds songs to it.
* **updatePlaylist():** Updates the playlist, adding or removing songs.
* **getPlaylist()**: Retrieves a user's playlist based on user ID or playlist name.

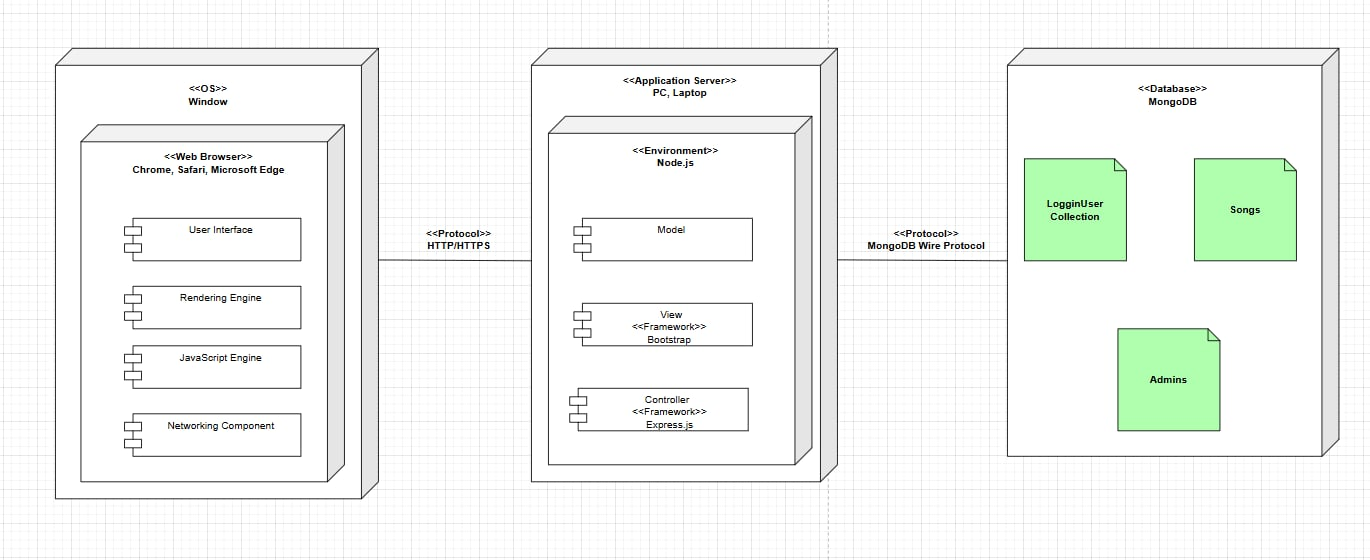
*Inputs:*

* **playlistData**: Contains playlist details such as the name of the playlist and the list of songs within it.
* **dbService**: Interacts with the database to create, update, and delete playlists.

**Class diagram:**



# Deployment

**

1. **Web Browser (Client Node)**

The **web browser** acts as the user-facing node that sends requests to the web server and renders the received responses. It consists of the following key components:

* **User Interface (UI)**
  + **Description**: This is the interface where users interact directly with the browser. It includes address bars, buttons for navigation (back, forward, refresh), and any other graphical elements.
  + **Purpose**: Allows users to enter website URLs and interact with the system’s functionalities, such as playing or pausing songs.
* **Rendering Engine**
  + **Description**: This component processes HTML, CSS, and other resources to display the webpage content.
  + **Purpose**: Ensures the UI displays correctly, including layouts, styles, and songs or playlists from the web application.
* **JavaScript Engine**
  + **Description**: Executes JavaScript code for dynamic functionalities like music playback, search, and volume adjustments.
  + **Purpose**: Adds interactivity to the web application, such as real-time updates to song playback or changes in UI elements.
* **Networking Component**
  + **Description**: Manages communication between the browser and the web server via HTTP/HTTPS protocols.
  + **Purpose**: Sends requests for web pages, songs, and other resources, and receives responses.
* **Operating System**: Windows
* **Browser**: Chrome, Safari, Microsoft Edge.

1. **Web Server (Application Node):**

The **web server** processes client requests and delivers appropriate resources (HTML, CSS, JavaScript files, or JSON responses). It includes three sub-nodes:

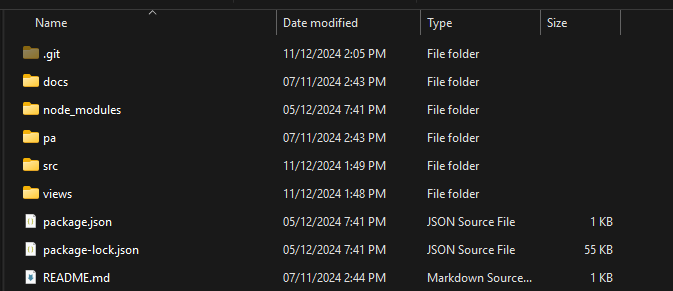
* **View Layer**
  + **Description**: Responsible for serving static files like HTML, CSS, and client-side JavaScript.
  + **Artifacts**:
    - Static files for the user interface (HTML templates, CSS stylesheets, and front-end JavaScript).
* **Controller Layer**
  + **Description**: Handles incoming HTTP requests and maps them to appropriate logic.
  + **Artifacts**:
    - Business logic for managing requests such as “play song,” “pause song,” or “fetch user profile.”
* **Model Layer**
  + **Description**: Handles communication between the application and the database, including fetching and storing data.
  + **Artifacts**:
    - Data structures for ***User, Admin,*** and ***Song*** objects, with corresponding CRUD operations.
* **File Storage**: A local system stores large files, such as audio tracks and album art, separately from the database to optimize performance.

1. **Database (MongoDB):**

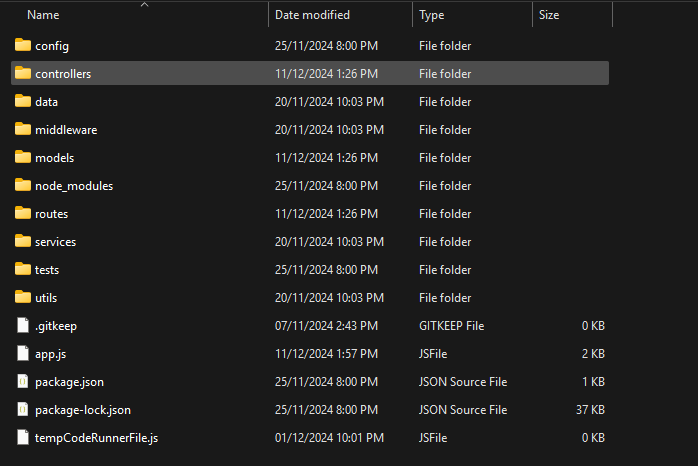
The **database server** stores structured data required for the application, using **MongoDB**. It manages and organizes data into collections.

* **Collections (Artifacts)**:
  + **LogginUser**: Stores user details, such as username, password (hashed), email, playlists, and playback history.
  + **Admin**: Maintains administrator data, including permissions and logs of changes made.
  + **Song**: Contains metadata for songs, such as title, artist, album, duration, and storage location in the file system.
* **Purpose**:
  + Ensures reliable storage and retrieval of data.
  + Provides query support for fast access to songs, playlists, and user profiles.

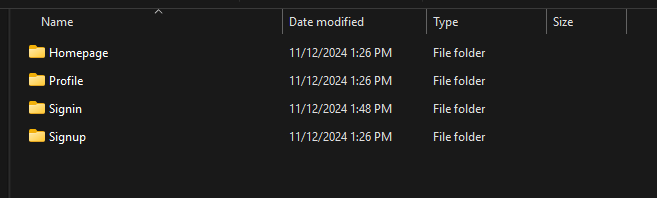
# Implementation View

**

*Overview file structure*

**

*Back-end file structure (src)*

**

*Front-end file structure (views)*

1. **Front-end structure (views)**

* This folder stores the templates for the application’s user interface, structured to enhance user experience and maintain consistent design. It includes subfolders for each individual page. Each subfolder contains its HTML, CSS files and also media files such as images, icons,..

1. **Back-end structure (src)**

* Contains the core backend code of the application, including server logic, routes, controllers, and database interactions.
* File app.js:
  + Purpose: The main entry point of the application.
  + Usage:
    - Set up the Express server.
    - Configures global middleware (e.g., for parsing JSON or handling CORS).
    - Registers route handlers defined in the routes folder.
* Folder routes/:
  + Purpose: Defines the routing layer of the application. Each file here corresponds to specific API endpoints.
  + Usage: Routes direct incoming HTTP requests to appropriate controller functions.
* Folder controllers/:
  + Purpose: Implements the logic for handling requests, interacting with models, and returning responses.
  + Usage:
    - For example, authController.js interacts with user.js to verify user data against the database.
* Folder models/:
  + Purpose: Defines the data structure and schema for the application. Models directly interact with the database.
  + Usage:
    - For example, when a user signs up, their data is validated and stored as a document in the database using this schema.
* Folder config/:
  + Purpose: Stores configuration files necessary for the application setup.
  + For example:
    - dbUsers.js: Configures the database connection, including connection strings and settings.
  + Usage:
    - Automatically initializes the database connection when the application starts.
* Folder node\_modules/:
  + Purpose: Stores external dependencies installed via npm.
  + Usage:
    - Includes libraries like express, mongoose, and others used in the application.