Music Playlist Manager – Full Software Documentation

1. System Architecture

The architecture of the Music Playlist Manager follows a **three-tier architecture model** that separates the system into layers to promote modularity, scalability, and ease of maintenance. This design ensures that the responsibilities of user interaction, business logic, and data persistence are clearly separated. The three layers are:

A. Presentation Layer (Frontend)

This layer is responsible for interacting directly with the users. Built using **HTML5**, **CSS3**, and **Vanilla JavaScript**, it presents a responsive and intuitive interface. The frontend design follows a mobile-first approach, ensuring a seamless experience on both mobile devices and desktops.

Key Features:

- Dynamic and responsive UI components for user registration, login, dashboard, and music management
- Form input validation on the client side to reduce unnecessary API calls
- Audio playback using the HTML5 <audio> tag with custom controls for play, pause, and skip
- Clean, accessible navigation between different parts of the application
- Frontend logic that interacts with the backend APIs via asynchronous JavaScript (AJAX/fetch)

This ensures the user always experiences fast response times and minimal page reloads.

B. Application Layer (Backend)

The application logic is implemented using **Node.js** along with the **Express.js** framework. This layer handles all the computation and coordination between the frontend and the data layer.

Responsibilities:

- Handles user authentication and secure session management using **JWT tokens**
- Receives file uploads using **Multer**, validating the type, size, and format of music files
- Implements business logic using **Object-Oriented Programming** for maintainability and reusability
- Exposes well-structured **RESTful API endpoints** to manage users, songs, playlists, and recent activity
- Handles all CRUD (Create, Read, Update, Delete) operations for songs and playlists
- Includes middleware for validation, error handling, and request logging

This layer serves as the brain of the application, orchestrating interactions between the user interface and the stored data.

C. Data Layer

This layer is responsible for data storage, retrieval, and caching. It uses a combination of **MongoDB** and **Redis** to store both persistent and temporary data efficiently.

Components:

- **MongoDB** stores permanent records of users, songs, and playlists. It is a NoSQL document database that offers flexibility in storing structured data without rigid schemas.

- **Mongoose** acts as an ODM (Object Document Mapper) to define schemas and interact with the MongoDB collections in a structured way.

- **Redis** is used for high-speed caching and storing temporary session data such as recently played songs. Redis stores lightweight data in-memory, which greatly improves performance for repeated accesses.

This layered architecture promotes better system organization, fault isolation, and allows developers to work in parallel on different parts of the system.

2. UML Diagrams

UML diagrams visually describe the structure and behavior of the system. This section includes a **Class Diagram** and a **Use Case Diagram**.

A. Class Diagram

The class diagram uses inheritance and encapsulation to represent the different entities in the system and how they relate to each other.

Key Highlights:

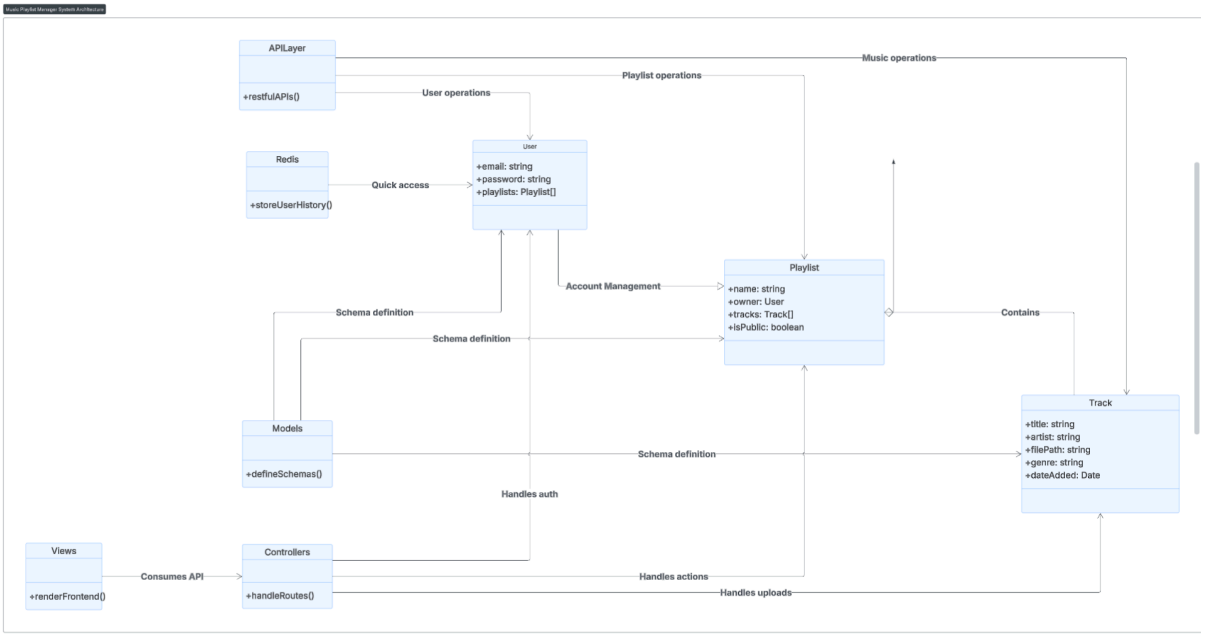
- The BasePlaylist is an abstract class with shared properties and methods.

- RegularPlaylist and SmartPlaylist extend from BasePlaylist, enabling future expansion of features like auto-generation of playlists.

- The Song class encapsulates all music metadata and validation logic.

- The User class handles personal data and their collection of playlists.

This object-oriented structure improves modularity and simplifies testing and maintenance.



B. Use Case Diagram

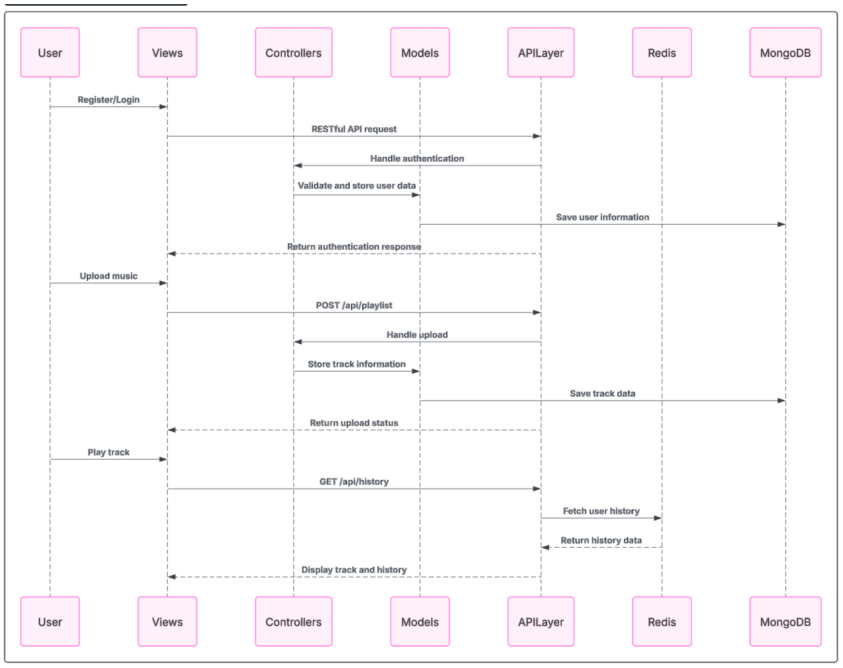
This diagram outlines how different system actors (users) interact with features in the system. The major use cases include:

1. **Authentication**:
    - Register a new account
    - Login to an existing account
    - Logout
2. **Music Management**:
    - Upload songs
    - View, search, and delete songs
    - Play audio tracks
3. **Playlist Management**:
    - Create, view, edit, and delete playlists

- Add or remove songs
- Set playlist privacy settings (public/private)

4. **Recently Played Management**:
   - Track recently played songs
   - View playback history
   - Clear playback history

These use cases ensure that all functional requirements of the system are clearly defined and implemented.



3. API Documentation

The system exposes a RESTful API that allows external clients (the frontend) to interact with backend resources. Each endpoint is well-documented and follows HTTP standards.

Authentication Endpoints

Handles secure user login and registration.

- **POST /api/auth/register**: Registers a new user with username, email, and password.
- **POST /api/auth/login**: Logs in existing users and returns an authentication token.

JWT tokens are used to protect sensitive endpoints and are stored client-side for authenticated requests.

Song Endpoints

- **POST /api/songs**: Uploads a new song file and metadata.
- **GET /api/songs**: Fetches a paginated list of all songs in the database.
- **GET /api/songs/search**: Searches for songs based on a query string.
- **GET /api/songs/:id**: Retrieves metadata for a specific song.
- **PUT /api/songs/:id**: Updates song metadata.
- **DELETE /api/songs/:id**: Deletes a specific song.
- **POST /api/songs/:id/play**: Logs a song as recently played and starts playback.

These routes allow users to fully manage their music collection.

Playlist Endpoints

- **POST /api/playlists**: Creates a new playlist.

- **GET /api/playlists/public**: Retrieves all publicly shared playlists.
- **GET /api/playlists/user/:userId**: Fetches playlists created by a specific user.
- **GET /api/playlists/:id**: Gets details of a specific playlist.
- **PUT /api/playlists/:id**: Updates playlist name, description, and visibility.
- **DELETE /api/playlists/:id**: Removes a playlist from the database.
- **POST /api/playlists/:playlistId/songs/:songId**: Adds a song to a playlist.
- **DELETE /api/playlists/:playlistId/songs/:songId**: Removes a song from a playlist.

These endpoints enable both static and dynamic playlist creation and management.

User and History Endpoints

- **GET /api/users**: List all registered users.
- **GET /api/users/:id**: Retrieve a user's profile by ID.
- **GET /api/users/:userId/recently-played**: Fetch songs recently played by the user.
- **DELETE /api/users/:userId/recently-played**: Clears the user's listening history.

These routes provide user profile access and personal listening data.

4. Database Schema Design

MongoDB is used for data persistence, with well-defined schemas for each entity:

Users Collection

Contains:

- username, email, and password fields
- Reference to the playlists created by the user
- Passwords are hashed with **bcrypt**

Songs Collection

Stores:

- Song title, artist, album, genre, duration
- Path to the uploaded audio file
- Timestamps for createdAt

Playlists Collection

Includes:

- Playlist name, description, userId, and a list of song references
- Type (regular or smart) and visibility (isPublic)
- For smart playlists, dynamic criteria for generating songs

Redis: Recently Played

Redis stores a lightweight list of recently played songs for each user:

- Format: recently_played:{userId}
- Data is pushed using LPUSH, accessed with LRANGE
- Songs are stored as compact JSON strings

This makes fetching playback history very fast and efficient.

5. Setup and Deployment Instructions

Prerequisites

- Node.js (v14 or newer)
- MongoDB server
- Redis server
- Git installed on your machine

Installation Steps

git clone https://github.com/Ajang-Deng98/fullstack_assignment_group5.git
cd fullstack_assignment_group5
npm install

Environment Setup

Create a .env file in the root directory with the following:

PORT=3000

```
MONGODB_URI=mongodb://localhost:27017/music_playlist_db
REDIS_URL=redis://localhost:6379
JWT_SECRET=your_jwt_secret_key
```

Run the Application

```
npm run dev       # Runs in development mode
npm start         # Runs in production mode
```

Access it at:

- App UI: http://localhost:3000
- API Docs: http://localhost:3000/api/docs
- Health Check: http://localhost:3000/api/health

6. Testing

Use the following commands:

```
npm test              # Run all test files
npm run test:coverage # Generate coverage report
```

Testing ensures reliability and catches bugs early in development.

7. Project Directory Structure

```
/fullstack_assignment_group5/
├── config/              # DB and Redis setup
├── controllers/          # Business logic handlers
```

```
├── middleware/          # Auth, upload, error handling
├── models/              # Mongoose schemas and OOP logic
├── routes/              # Express route definitions
├── public/              # Static frontend assets (HTML/CSS/JS)
├── uploads/             # Audio file uploads
├── utils/               # Helper functions (e.g., Redis client)
├── tests/               # Unit and integration tests
├── server.js            # Entry point
└── package.json         # Dependencies and scripts
```

This structure ensures separation of concerns and scalability.

Conclusion

The Music Playlist Manager is a fully-featured web application that brings together robust backend systems, an intuitive frontend interface, and smart data handling through Redis and MongoDB. This documentation outlines all the key components, from system architecture and database design to API endpoints and deployment steps. It reflects best practices in full-stack development and can serve as a foundation for future enhancements, such as social features, AI-generated playlists, or mobile app integration.