

Music Playlist Application - Design Specification

Table of Contents

Introduction.....	1
Architecture.....	2
Database Schema	3
API Endpoints (Servlets)	5
Authentication & User Management.....	5
Songs	6
Playlists.....	10
Frontend Components (Conceptual)	14
Key Features (SPA Specifics)	16
Frontend JavaScript Architecture and Structure	17
Core Modules & Responsibilities	17
Directory Structure and Component Roles (<code>src/main/webapp/js/</code>).....	18
Data Flow and State Management.....	19
Key Architectural Characteristics	19
Client-Side Feature Implementation	20
Testing tools.....	21

Introduction

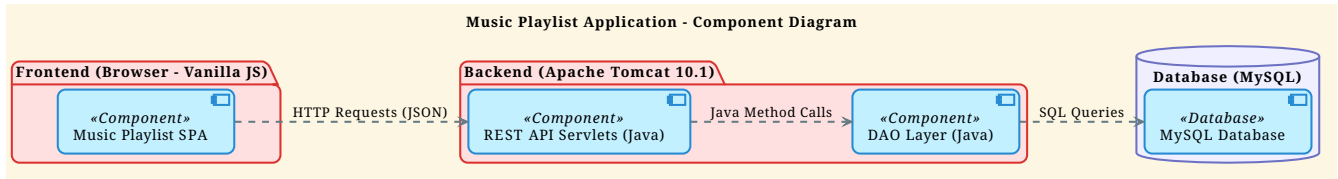
This document outlines the design for a web-based music playlist management application. Users can upload songs, organize them into playlists, and play them. This specification focuses on the Single Page Application (SPA) version, providing a seamless user experience without full page reloads.

Technology Stack:

- **Frontend:** HTML, CSS, JavaScript (Vanilla JS)
- **Backend:** Java Servlets (running on Apache Tomcat 10.1)
- **Database:** MySQL
- **API Format:** RESTful APIs exchanging mostly JSON data but `multipart` or other media types for audio and images

Architecture

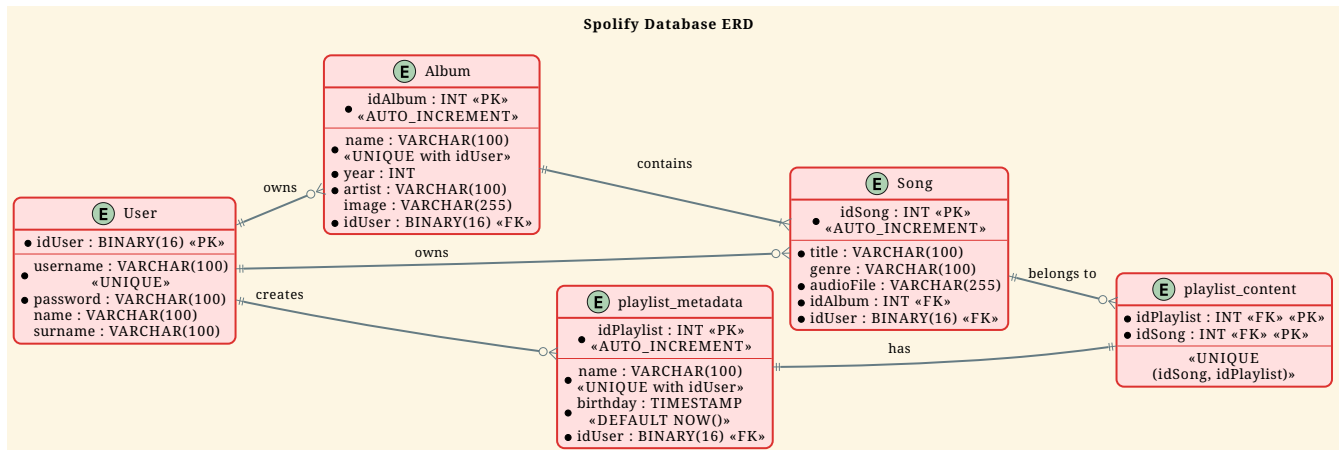
The application follows a standard client-server SPA architecture:



- **Frontend:** A single HTML page dynamically updated using JavaScript. It handles user interactions, makes asynchronous requests (e.g., using `fetch` API) to the backend, and updates the DOM accordingly.
- **Backend:** Java Servlets expose RESTful API endpoints. They handle business logic, interact with the Data Access Object (DAO) layer, and manage user sessions.
- **DAO Layer:** Java classes responsible for interacting with the MySQL database (CRUD operations).
- **Database:** MySQL stores user data, song metadata, album information, and playlist structures.

Database Schema

The database consists of the following tables:



- **User:** Stores user credentials and basic information.
 - **idUser** (PK, BINARY(16), NOT NULL)
 - **username** (VARCHAR(100), UNIQUE, NOT NULL)
 - **password** (VARCHAR(100), NOT NULL - *Store hashed passwords*)
 - **name** (VARCHAR(100))
 - **surname** (VARCHAR(100))
- **Album:** Stores album details. Each album is associated with a user.
 - **idAlbum** (PK, INT, Auto-Increment, NOT NULL)
 - **name** (VARCHAR(100), NOT NULL) - *Unique per user*
 - **year** (INT, NOT NULL)
 - **artist** (VARCHAR(100), NOT NULL)
 - **image** (VARCHAR(255)) - *Optional Album cover image*
 - **idUser** (FK, BINARY(16), NOT NULL - References User.idUser)
- **Song:** Stores song metadata and file paths. Each song belongs to an album and is associated with a user.
 - **idSong** (PK, INT, Auto-Increment, NOT NULL)
 - **title** (VARCHAR(100), NOT NULL)
 - **idAlbum** (FK, INT, NOT NULL - References Album.idAlbum)
 - **genre** (VARCHAR(100))
 - **audioFile** (VARCHAR(255), NOT NULL - *Audio filename*)
 - **idUser** (FK, BINARY(16), NOT NULL - References User.idUser)
 - *Note: The 'year' for a song is derived from its associated Album's year.*
- **playlist_metadata:** Stores playlist metadata. Each playlist is created by a user.

- **idPlaylist** (PK, INT, Auto-Increment, NOT NULL)
- **name** (VARCHAR(100), NOT NULL) - *Unique per user*
- **birthday** (TIMESTAMP, NOT NULL, DEFAULT CURRENTTIMESTAMP) - *_Creation timestamp*
- **idUser** (FK, BINARY(16), NOT NULL - References User.idUser)
- **playlist_content**: Joining table for the N-N relationship between **playlist_metadata** and **Song**.
 - **idPlaylist** (PK, FK, INT, NOT NULL - References playlist_metadata.idPlaylist)
 - **idSong** (PK, FK, INT, NOT NULL - References Song.idSong)
 - *Unique constraint on (idSong, idPlaylist)*

API Endpoints (Servlets)

The backend will expose RESTful API endpoints, all prefixed with `/api/v1/`. The primary servlets and their functionalities are:

Authentication & User Management

- **POST /auth/login**: Authenticates an existing user.
 - Request: JSON.

Login Request

username	user123
password	securepassword123

- Response (200 OK): On success, returns JSON and sets an HTTP session cookie.

User Information Response

username	user123
name	John
surname	Doe

- Error Responses:
 - **400 Bad Request**: Invalid input (e.g., missing fields, invalid format).
 - **401 Unauthorized**: Incorrect credentials.
 - **500 Internal Server Error**: Server-side error.
 - **POST /users**: Registers a new user.
- Request: JSON.

User Creation Request

username	newuser
password	newpassword456
name	Jane
surname	Doe

- Response (201 CREATED): On success, returns JSON and sets an HTTP session cookie.

User Information Response

username	user123
name	John
surname	Doe

- Error Responses:
 - **400 Bad Request**: Invalid input or validation errors.
 - **409 Conflict**: Username already exists.
 - **500 Internal Server Error**: Server-side error.
 - **POST /auth/logout**: Logs out the currently authenticated user.
- Request: No body required.
- Response (200 OK): Returns JSON. Invalidates the user's HTTP session.

Logout Successful Response

message	Logout successful.
----------------	--------------------

- Error Responses:
 - **500 Internal Server Error**: If an unexpected server error occurs during logout.
 - **GET /auth/me**: Checks if the current user has an active session.
- Request: No body required.
- Response (200 OK): If a session is active, returns JSON.

User Information Response

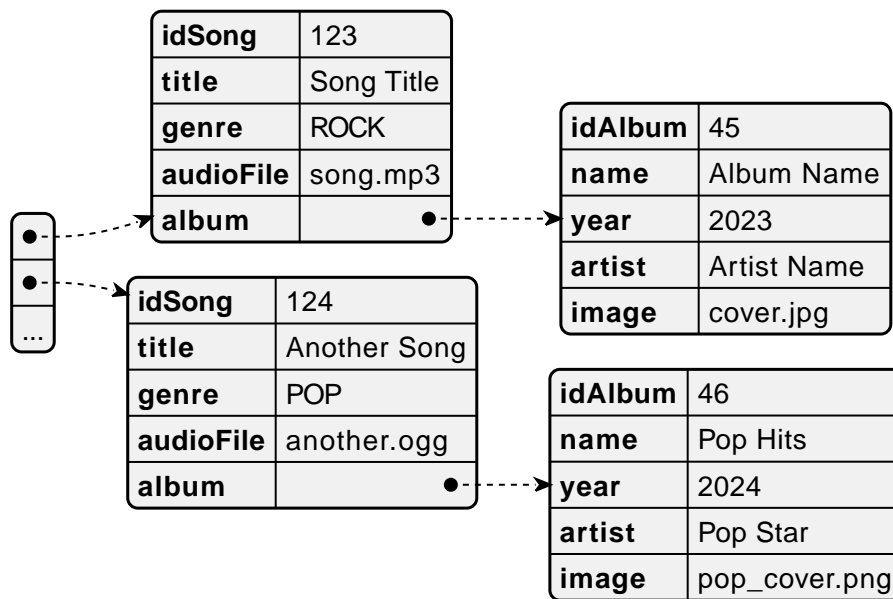
username	user123
name	John
surname	Doe

- Error Responses:
 - **401 Unauthorized**: No active session.

Songs

- **GET /songs**: Fetches all songs for the authenticated user.
 - Request: No body required.
 - Response (200 OK): JSON array of **SongWithAlbum** objects. Each object includes full song details and associated album details.

List of Songs with Album Details Response



- Error Responses:
 - **401 Unauthorized**: User not authenticated.
 - **500 Internal Server Error**: Server-side error.
 - **POST /songs**: Uploads a new song. If an album with the provided **albumTitle** doesn't exist for the user, a new album is created.
- Request: **multipart/form-data** containing:
 - **title** (text, required): The title of the song.
 - **genre** (text, required): The genre of the song (must be one of the predefined values, see **GET /songs/genres**).
 - **albumTitle** (text, required): The title of the album.
 - **albumArtist** (text, required): The artist of the album.
 - **albumYear** (number, required): The year of the album.
 - **audioFile** (file, required): The audio file for the song (e.g., **audio.mp3**).
 - **albumImage** (file, optional): The cover image for the album (e.g., **cover.jpg**). This is used if a new album is being created and this part is provided.
- Response (201 CREATED): JSON **SongWithAlbum** object representing the newly created song and its (potentially new) album.

Song with Album Details Response

idSong	123
title	Song Title
genre	ROCK
audioFile	song.mp3
album	•

idAlbum	45
name	Album Name
year	2023
artist	Artist Name
image	cover.jpg

- Error Responses:
 - **400 Bad Request**: Invalid input (e.g., missing required fields, invalid genre, invalid year format, file processing error).
 - **401 Unauthorized**: User not authenticated.
 - **409 Conflict**: If a constraint violation occurs (e.g., song title already exists in the album for that user, though this specific check might vary based on DAO implementation).
 - **500 Internal Server Error**: Server-side error (e.g., DAO exception, file storage issue).
 - **GET /songs/genres**: Fetches all available song genres.
- Request: No body required.
- Response (200 OK): JSON array of objects, where each object has a **name** (e.g., "ROCK") and **description** (e.g., "Rock Music") for the genre.

List of Available Genres Response


•	<table><tr><td>name</td><td>ROCK</td></tr><tr><td>description</td><td>Rock Music</td></tr></table>	name	ROCK	description	Rock Music
name	ROCK				
description	Rock Music				
•	<table><tr><td>name</td><td>POP</td></tr><tr><td>description</td><td>Popular Music</td></tr></table>	name	POP	description	Popular Music
name	POP				
description	Popular Music				
•	<table><tr><td>name</td><td>JAZZ</td></tr><tr><td>description</td><td>Jazz Music</td></tr></table>	name	JAZZ	description	Jazz Music
name	JAZZ				
description	Jazz Music				
...					

- Error Responses:
 - **401 Unauthorized**: User not authenticated (if authentication is enforced for this endpoint, though typically it might be public).
 - **500 Internal Server Error**: Server-side error.
 - **GET /songs/{songId}**: Fetches details for a specific song, identified by **songId**.
- Request: No body required.
- Response (200 OK): JSON **SongWithAlbum** object containing full song details and associated album details.

Song with Album Details Response

idSong	123
title	Song Title
genre	ROCK
audioFile	song.mp3
album	●

idAlbum	45
name	Album Name
year	2023
artist	Artist Name
image	cover.jpg



- Error Responses:
 - **400 Bad Request**: Invalid `songId` format.
 - **401 Unauthorized**: User not authenticated.
 - **404 Not Found**: Song not found or not owned by the user.
 - **500 Internal Server Error**: Server-side error.
 - **GET /songs/{songId}/audio**: Fetches the audio file for a specific song.
- Request: No body required.
- Response (200 OK): The audio file stream (e.g., `audio/mpeg`, `audio/ogg`) with appropriate **Content-Type** and **Content-Disposition** headers.
- Error Responses:
 - **400 Bad Request**: Invalid `songId` format.
 - **401 Unauthorized**: User not authenticated.
 - **404 Not Found**: Song not found, not owned by the user, or audio file is missing.
 - **500 Internal Server Error**: Server-side error (e.g., error reading file).
 - **GET /songs/{songId}/image**: Fetches the album cover image for the album associated with a specific song.
- Request: No body required.
- Response (200 OK): The image file stream (e.g., `image/jpeg`, `image/png`) with appropriate **Content-Type** and **Content-Disposition** headers.
- Error Responses:
 - **400 Bad Request**: Invalid `songId` format.
 - **401 Unauthorized**: User not authenticated.
 - **404 Not Found**: Song not found, album not found, not owned by the user, or image file is missing.
 - **500 Internal Server Error**: Server-side error (e.g., error reading file).

Playlists

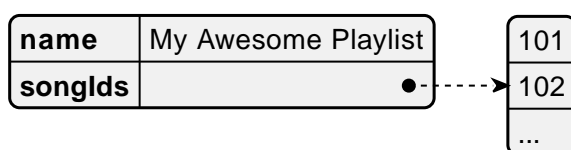
- **GET /playlists**: Fetches all playlists for the authenticated user.
 - Request: No body required.
 - Response (200 OK): JSON array of **Playlist** objects.

List of Playlists Response



- Error Responses:
 - **401 Unauthorized**: User not authenticated.
 - **500 Internal Server Error**: Server-side error.
 - **POST /playlists**: Creates a new playlist.
- Request: JSON (songIds is optional, if provided must be an array of positive integers).

Playlist Creation Request



- Response (201 CREATED): On success, returns the created **Playlist** object.

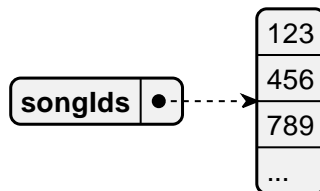
Playlist Details Response



- Error Responses:

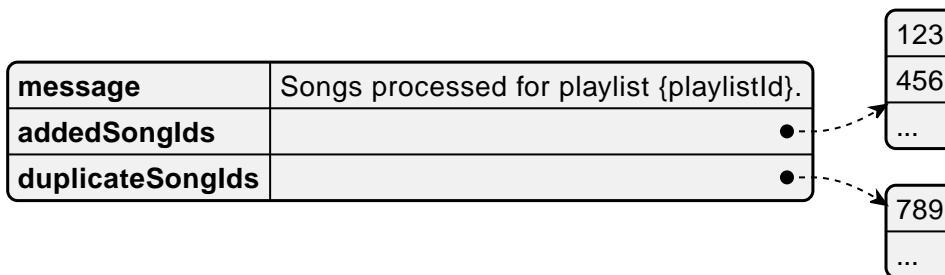
- **400 Bad Request**: Invalid input (e.g., missing name, invalid name format, invalid song IDs).
- **401 Unauthorized**: User not authenticated.
- **409 Conflict**: Playlist name already exists for the user.
- **500 Internal Server Error**: Server-side error (e.g., DAO exception).
 - **POST /playlists/{playlistId}/songs**: Adds one or more songs to an existing playlist.
- Request: JSON. **songIds** must be a non-empty array of positive integers.

Add Songs to Playlist Request



- Response (200 OK): On successful processing, returns JSON:

Add Songs to Playlist Result Response



- Error Responses:
 - **400 Bad Request**: Invalid JSON, missing/empty **songIds**, invalid song ID format.
 - **401 Unauthorized**: User not authenticated.
 - **403 Forbidden**: User does not own the playlist, or a specified song is not owned by the user.
 - **404 Not Found**: Playlist with **{playlistId}** not found, or a specified song ID not found.
 - **500 Internal Server Error**: Other server-side errors.
 - **GET /playlists/{playlistId}/order**: Fetches the current order of songs for a specific playlist.
- Request: No body required.
- Response (200 OK): JSON array of song IDs representing the order.

List of Song IDs Response

101
105
102
...

- Error Responses:
 - **400 Bad Request**: Invalid playlist ID format.
 - **401 Unauthorized**: User not authenticated.
 - **404 Not Found**: Playlist not found or user does not have access.
 - **500 Internal Server Error**: Other server-side errors.
 - **PUT /playlists/{playlistId}/order**: Updates the order of songs in a specific playlist.
- Request: JSON array of song IDs in the desired new order. The list must contain all and only the song IDs currently in the playlist, without duplicates.

Update Playlist Order Request

105
101
102
...

- Response (200 OK): JSON array of song IDs confirming the new order.

List of Song IDs Response

101
105
102
...

- Error Responses:
 - **400 Bad Request**: Invalid JSON format, invalid playlist ID, song ID list does not match current playlist content (e.g., missing songs, extra songs, duplicate songs in request, invalid song IDs).
 - **401 Unauthorized**: User not authenticated.
 - **404 Not Found**: Playlist not found or user does not have access.
 - **500 Internal Server Error**: Other server-side errors.





Error Handling: APIs should return appropriate HTTP status codes (e.g., 200, 201, 400, 401, 403, 404,

500) and JSON error messages.

Frontend Components (Conceptual)

The JavaScript SPA will manage different views/components, dynamically rendered within the main application container (`<main id="app">` in `index.html`):

- **Login/Signup View:** Forms for user authentication (`#login` route) and registration (`#signup` route). Managed by `loginHandler.js` and `loginView.js`.
- **Main Application Structure (Single Page):**
 - **Navigation/Header:** Contains navigation links (e.g., "Home", "Songs" - managed by `app.js`) and a "Logout" button. User-specific information might be displayed within the content of authenticated views rather than fixed in the global header.
 - **Home View (`#home` route):** Managed by `homeHandler.js` and `homeView.js`. This is the main landing page after login. It displays:
 - The user's playlists, sorted by creation date (descending). Each playlist entry links to its specific Playlist View and provides access to the Reorder Modal.
 - A form for uploading new songs.
 - A form for creating new playlists, including a list of the user's available songs (sorted alphabetically by artist, then by album year) to select from.
 - **Songs View (`#songs` route):** Managed by `songsHandler.js` and `songsView.js`. This view displays a comprehensive list of all songs uploaded by the user. Selecting a song from this list will trigger the Player Functionality.
 - **Playlist View (`#playlist-:idplaylist` route):** Managed by `playlistHandler.js` and `playlistView.js`. Accessed by selecting a specific playlist from the Home View. It displays:
 - Songs belonging to the selected playlist, presented 5 at a time. The songs are initially displayed according to their default order (alphabetically by artist/group, then by album publication year ascending) or a previously saved custom order.
 - "Previous" and "Next" buttons for client-side pagination through the playlist's songs.
 - A form to add more songs (from the user's collection) to the current playlist. After new songs are added, the view refreshes, typically displaying the first page/block of songs.
 - **Player Functionality:** This is not a static "section" but a dynamic update of the UI that occurs when a user selects a song title (e.g., from the Playlist View or Songs View). It will display:
 - Full details of the selected song.
 - An HTML5 audio player for playback.
 - **Reorder Modal:** Activated from the Home View via a link/button associated with each playlist. Managed by the relevant handler (e.g., `homeHandler.js`) and uses `sharedComponents.js` for the modal structure. It displays:
 - A complete list of songs for the selected playlist, initially shown in their current order (default or custom).
 - Functionality for users to drag-and-drop songs to define a custom order. This reordering happens client-side.

- A "Save Order" button to persist the custom sequence to the server via `apiService.updatePlaylistOrder()`. Once a custom order is saved, it becomes the default display order for that playlist in subsequent views. If new songs are added to a playlist that has a custom order, they are appended to the end of this custom order.
- **Color Palette:**
 - Background color:  #EEEEEE
 - Alternative background color:  #D4BEE4
 - Text:  #9B7EBD
 - Highlight color:  #3B1E54.

Key Features (SPA Specifics)

- **Single Page Experience:** All interactions happen within one HTML page, dynamically updating content via JavaScript without full reloads.
- **Asynchronous Communication:** Uses `fetch` or similar for all backend communication.
- **Client-Side Playlist Pagination:** The "Previous"/"Next" functionality in the Playlist View is handled entirely in JavaScript without server requests.
- **Client-Side Reordering:** Drag-and-drop reordering of songs in the modal happens client-side. The final order is sent to the server only when the user clicks "Save Order".
- **Dynamic Updates:** Forms (song upload, playlist creation, add song to playlist) update relevant sections of the page asynchronously upon success.
- **State Management:** JavaScript will manage the application state (current view, user data, playlists, songs, etc.).

Frontend JavaScript Architecture and Structure

The frontend is a Vanilla JavaScript Single Page Application (SPA) built with a modular structure. It dynamically updates the content of `index.html` without full page reloads. The core JavaScript files (`app.js`, `router.js`, `apiService.js`) and the directory structure (`handlers/`, `views/`, `utils/`) define its architecture.

Core Modules & Responsibilities

1. `app.js` (Main Entry Point):

- Initializes the application upon `DOMContentLoaded`.
- Sets up the client-side router (`router.js`) by defining route-to-handler mappings.
- Manages the initial user session state by calling `apiService.checkAuthStatus()`. Authenticated user data is stored in `sessionStorage`.
- Redirects users to the login page if they attempt to access protected routes without an active session.
- Dynamically populates and manages the global navigation bar (`#navbar`), including navigation links (e.g., Home, Songs) and the logout button.
- Orchestrates the loading of different views into the main application container (`<main id="app">`) based on the current route and authentication status.

2. `router.js` (Client-Side Routing):

- Implements a hash-based routing system (e.g., `#home`, `#playlist-123`).
- Listens for `hashchange` events to trigger route transitions.
- Parses route parameters from the URL hash (e.g., `idplaylist` from `#playlist:idplaylist`).
- Maps URL patterns to specific handler functions (defined in `app.js` and sourced from `handlers/`).
- Manages a visual loader element during page transitions to indicate activity.
- Controls the visibility of the global navigation bar based on whether the current route is public (e.g., login, signup) or protected.
- Handles unknown routes by displaying a "404 - Page Not Found" message within the application container.

3. `apiService.js` (API Communication Layer):

- Centralizes all HTTP requests to the backend REST API (prefixed with `api/v1`).
- Uses the `fetch` API for asynchronous communication.
- Provides a generic `_fetchApi` helper function that handles:
 - Setting appropriate request headers (`Content-Type: application/json`, `Accept: application/json`).

- Serializing request bodies to JSON (or handling `FormData` for file uploads, e.g., in `uploadSong`).
- Parsing JSON responses from the server.
- Comprehensive error handling: It catches network errors and non-OK HTTP responses, creating custom `ApiError` objects that include status codes, messages, and detailed error information from the server's JSON response.
- Exports dedicated, JSDoc-typed functions for each API endpoint (e.g., `login()`, `getPlaylists()`, `uploadSong()`, `updatePlaylistOrder()`), making API calls clean, type-hinted, and reusable throughout the application.
- Includes URL builder functions for constructing media URLs (e.g., `getSongImageURL()`, `getSongAudioURL()`).

Directory Structure and Component Roles

(`src/main/webapp/js/`)

- **handlers/ (Controller/Presenter Logic):**
 - Modules in this directory (e.g., `homeHandler.js`, `loginHandler.js`, `playlistHandler.js`, `songsHandler.js`) are responsible for the logic associated with specific views or application "pages".
 - They are invoked by the router when a corresponding route is matched.
 - Typical responsibilities include:
 - Fetching necessary data from the backend using functions from `apiService.js`.
 - Processing user input and handling events delegated from the UI elements.
 - Managing view-specific state or data transformations, including client-side state for features like playlist pagination or song reordering within a modal.
 - Coordinating with modules in the `views/` directory to render or update the UI within the main application container (`#app`).
 - `sharedFormHandlers.js` provides reusable logic for common form submission patterns (e.g., handling song uploads, playlist creation).
- **views/ (View Rendering Logic):**
 - Modules here (e.g., `homeView.js`, `loginView.js`, `playlistView.js`, `songsView.js`) are primarily concerned with generating and manipulating the DOM for different sections of the application.
 - They typically export functions that take data (provided by handlers) and return HTML structures (often as DOM elements created via `utils/viewUtils.js`) or directly update existing DOM elements.
 - Event listeners for UI elements are often attached within these modules, delegating actions to handler functions.
 - `playlistView.js`, in conjunction with `playlistHandler.js`, manages the display of paginated songs (e.g., 5 at a time) and the "Previous/Next" buttons for client-side navigation through a

playlist's songs. It also integrates with the reorder modal functionality.

- `songsView.js` is responsible for rendering the page that lists all user songs (accessed via the `#songs` route). The "Player Section" functionality, for playing a selected song, is a conceptual component that would be updated with song details and an audio player when a song is selected from any list.
- `sharedComponents.js` provides functions to create reusable UI elements such as modals (e.g., for song reordering), buttons, and lists, ensuring consistency across different views.
- **utils/ (Utility Functions):**
 - This directory contains helper modules that provide common, reusable functionalities to support other parts of the application.
 - Examples include:
 - `viewUtils.js`: DOM manipulation helpers (e.g., `createElement` for creating elements, functions to clear containers).
 - `formUtils.js`: Utilities for form validation, data extraction from forms, or resetting forms.
 - `delayUtils.js`: Functions for adding artificial delays, possibly for UI effects or simulating network latency during development/testing.
 - `orderUtils.js`: Provides utilities to support drag-and-drop reordering logic for song lists, particularly within the reorder modal.

Data Flow and State Management

- **Authentication State:** Primarily managed in `app.js` and `router.js`. User information for an active session is stored in `sessionStorage`. Access to protected routes is conditional on this stored state.
- **View-Specific Data:** Fetched asynchronously by handler modules (from `handlers/`) using `apiService.js` when a view is loaded or requires new data. For features like client-side playlist pagination or reordering, `playlistHandler.js` may fetch the full list of songs for a playlist once and cache it client-side to avoid repeated server requests for these operations.
- **UI Updates:** Data is passed from handlers to view modules. View modules are responsible for rendering this data into the DOM. Updates typically involve clearing and re-rendering sections of the `#app` container or specific components within it.
- **State Management:** There is no centralized state management library (like Redux or Vuex). Application state is primarily managed locally within handler modules (e.g., current page index for pagination, temporary song order during reordering), or passed between modules as function arguments. `sessionStorage` is used for persistent session state (user data).

Key Architectural Characteristics

- **Modularity:** The codebase is organized into distinct JavaScript modules with specific responsibilities (API interaction, routing, view rendering, business logic/handlers, utilities), imported/exported using ES6 module syntax.
- **Single Page Application (SPA):** Achieved through client-side hash-based routing, which

prevents full page reloads and provides a smoother user experience.

- **Asynchronous Operations:** Extensive use of `async/await` and Promises for non-blocking API calls and other asynchronous tasks, ensuring the UI remains responsive.
- **Vanilla JavaScript:** The application is built using plain JavaScript, HTML, and CSS, without relying on large frontend frameworks (like React, Angular, or Vue). DOM manipulation is done directly or via helper utilities.
- **Separation of Concerns (SoC):**
 - API interaction logic is strictly isolated in `apiService.js`.
 - Routing and navigation logic is encapsulated in `router.js`.
 - UI rendering and DOM manipulation are primarily handled by modules in the `views/` directory.
 - Application flow, event handling coordination, and view-specific data management are primarily the responsibility of modules in the `handlers/` directory.

Client-Side Feature Implementation

- **Playlist Pagination:** When viewing a playlist, `playlistHandler.js` fetches the complete list of song IDs (or full song objects if needed for display without further lookups) for that playlist via `apiService.getPlaylistSongOrder()` (or `apiService.getSongs()` filtered by playlist). This list is stored client-side. `playlistView.js` then renders a "page" of songs (e.g., 5 items) based on a current page index managed by `playlistHandler.js`. "Previous" and "Next" button clicks in the view update this index in the handler, which then instructs the view to re-render the appropriate slice of songs, all without further server requests.
- **Song Reordering Modal:** From the Home page (or Playlist page), a "Reorder" action for a playlist (handled by `homeHandler.js` or `playlistHandler.js`) triggers the display of a modal (likely created using `sharedComponents.js`). This modal, managed by the respective handler, lists all songs in the playlist. Users can drag and drop songs to change their order; this reordering is handled client-side (potentially using `utils/orderUtils.js` and native HTML5 drag-and-drop APIs). The temporary new order is maintained in the handler. Upon clicking a "Save Order" button in the modal, the handler calls `apiService.updatePlaylistOrder()` with the new sequence of song IDs to persist the changes on the server.

Testing tools

- **Generating mock data:** `mvn compile exec:java -Pgenerate`
- **Deleting mock data:** `mvn compile exec:java -Pcleanup`