1. **INTRODUCTION**

* **PROJECT TITLE** : RYTHIMIC TUNES
* **TEAM ID** : SWTID1741161330153553
* **TEAM LEADER :** SAI SATHYA S
* **TEAM MEMBER :** RITHISH KUMAR V
* **TEAM MEMBER :** SAKTHIVEL R
* **TEAM MEMBER :** SANJAI KUMAR K

1.NAME: SAISATHYA S

ROLL NO: 2201111033027

GMAIL: ca2201111033027@lngovernmentcollege.com

2.NAME: RITHISH KUMAR V

ROLL NO: 2201111033026

GMAIL: [ca2201111033026@lngovernmentcollege.com](mailto:ca2201111033026@lngovernmentcollege.com)

3.NAME: SAKTHIVEL R

ROLL NO: 2201111033028

GMAIL: [ca2201111033028@lngovernmentcollege.com](mailto:ca2201111033028@lngovernmentcollege.com)

4.NAME: SANJAI KUMAR K

ROLL NO: 2201111033029

GMAIL: ca2201111033029@lngovernmentcollege.com

**2. PROJECT OVERVIEW**

* **Purpose**:  
   Rythimic Tunes aims to transform how users interact with music by providing a cutting-edge, browser-based streaming application. The primary purpose is to offer a platform where users can discover new tracks, revisit their favorite classics, and curate personalized playlists—all within an intuitive and visually appealing interface. Built with React.js, the application prioritizes performance, scalability, and cross-device compatibility, ensuring that users enjoy a consistent experience whether they’re on a desktop at home or a smartphone on the go. The overarching goal is to bridge the gap between traditional music consumption and modern digital convenience, making music more accessible and enjoyable for everyone.
* **Features**:
  + **Song Listings**: A detailed catalog of songs, complete with metadata such as title, artist, genre, and release date, presented in an organized and visually appealing layout.
  + **Playlist Creation**: Empowers users to craft custom playlists by adding, removing, and rearranging songs to suit their moods or activities.
  + **Playback Control**: Offers a robust set of controls including play, pause, skip, and volume adjustment, ensuring smooth and uninterrupted listening.
  + **Offline Listening**: Allows users to download songs for offline playback, making the app usable even without an internet connection—a feature particularly valuable for commuters or travelers.
  + **Search Functionality**: A powerful search tool that lets users quickly find songs, artists, or albums by typing keywords, enhancing discoverability and usability.

By combining these features, Rythimic Tunes caters to both casual listeners and dedicated music aficionados, providing a versatile platform that adapts to diverse user needs.

**3. ARCHITECTURE**

**Component Structure**:  
 The application’s architecture is modular, leveraging React’s component-based design to ensure maintainability and reusability. Key components include:

* + **App**: The root component that serves as the entry point, managing the overall layout and routing logic. It acts as a container for other components and ensures seamless navigation across the app.
  + **Sidebar**: A persistent navigation panel that provides quick access to the Songs, Favorites, and Playlist sections, enhancing user flow and accessibility.
  + **Songs**: The main content area where users browse the song catalog, search for tracks, and interact with playback and curation features.
  + **Favorites**: A dedicated view for songs marked as favorites, offering a quick way to revisit beloved tracks.
  + **Playlist**: A dynamic interface for managing user-created playlists, allowing additions, deletions, and playback directly from the list.  
    These components interact through props and state, with the Songs component serving as the central hub for most user interactions.

**State Management**:

* + **Global State**: The application uses React’s useState and useEffect hooks for state management, keeping it lightweight and straightforward. Global state includes the song catalog (items), user favorites (wishlist), playlists (playlist), currently playing track (currentlyPlaying), and search input (searchTerm). This approach avoids the complexity of libraries like Redux for this project’s scope, though it could be expanded in the future for larger-scale needs.
  + **Local State**: Individual components manage their own local state where necessary. For example, the Songs component maintains searchTerm locally to filter the song list in real time, ensuring a responsive user experience without affecting other parts of the app.

**Routing**:  
 The app employs react-router-dom to handle client-side routing, enabling smooth transitions between views without full page reloads. The routing structure is as follows:

* + /: Loads the Songs component, displaying the main song catalog.
  + /favorites: Renders the Favorites component, showing the user’s favorited songs.
  + /playlist: Displays the Playlist component, where users manage their custom playlists.  
    This setup ensures a single-page application (SPA) feel, with the BrowserRouter wrapping the entire app to enable navigation.

**4. SETUP INSTRUCTIONS**

**Prerequisites**:  
 To set up and run Rythimic Tunes locally, ensure the following software is installed:

* + **Node.js and npm**: The backbone of the JavaScript runtime environment, required to execute the app and manage dependencies.
    - Download: <https://nodejs.org/en/download/>
    - Install via package manager: <https://nodejs.org/en/download/package-manager/>
  + **Code Editor**: A development environment like Visual Studio Code (recommended), Sublime Text, or WebStorm for editing code.
    - Visual Studio Code: <https://code.visualstudio.com/download>
  + **JSON Server**: A lightweight tool to simulate a backend API using a db.json file. Install globally via npm:

npm install -g json-server

**Installation**:  
 Follow these steps to get the project running on your machine:

* + - 1. **Clone the Repository**:

git clone <repository-url>

cd rythimic-tunes

Replace <repository-url> with the actual URL of your project repository (e.g., from GitHub).

**2. Install Dependencies**:

npm install

This installs core dependencies like React and Vite (if used as the build tool).

**3.Install Additional Libraries**:

npm install react-router-dom react-icons bootstrap axios

These libraries provide routing, icons, UI styling, and HTTP request capabilities, respectively.

**4.Configure the Mock Backend**:

* + - Create a db folder in the project root and add a db.json file with sample song data (see sample below).
    - Start the JSON server:

json-server --watch ./db/db.json

This runs the mock API at http://localhost:3000.

**5. FOLDER STRUCTURE**

**Client**:  
The frontend codebase is organized as follows:

* + src/
    - Components/
      * Sidebar.js: Navigation component.
      * Songs.js: Song listing and interaction component.
      * Favorites.js: Favorites display component.
      * Playlist.js: Playlist management component.
    - App.js: Root component with routing setup.
    - App.css: Custom styles for the app.
    - index.js: Entry point rendering the app to the DOM.
  + db/
    - db.json: Mock database storing songs, favorites, and playlists.
  + public/
    - Static assets like images or icons (if used).

**Utilities**:

* + **Custom Hooks**: A potential useAudio hook could manage playback logic, though not explicitly implemented here.
  + **Helper Functions**: Axios-based API calls (e.g., fetching songs, updating wishlist serve as reusable utilities, reducing code duplication across components.

This structure promotes modularity, making it easy to locate and update specific parts of the application as it grows.

**6. RUNNING THE APPLICATION**

**Frontend**:  
To launch the React application locally:

npm run dev

* + If using Vite (as suggested by npm run dev), the app will be available at http://localhost:5173.
  + If using Create React App, use npm start and access it at http://localhost:3000.

**Mock Backend**:  
In a separate terminal, start the JSON server:

json-server --watch ./db/db.json

* + This exposes endpoints like http://localhost:3000/items for songs, /favorities for favorites, and /playlist for playlists.

Ensure both the frontend and backend are running simultaneously for full functionality.

**7. COMPONENT DOCUMENTATION**

**Key Components**:

* + **App**:
    - **Purpose**: Acts as the central hub, orchestrating routing and layout. It wraps the app in a BrowserRouter and renders the Sidebar alongside the routed content.
    - **Props**: None; relies on child components for functionality.
  + **Sidebar**:
    - **Purpose**: Provides a consistent navigation experience, linking to the app’s main sections. It’s designed to be persistent across routes, improving usability.
    - **Props**: None; static links to predefined routes.
  + **Songs**:
    - **Purpose**: The core interaction space where users browse songs, search the catalog, manage favorites/playlists, and control playback. It’s the most feature-rich component.
    - **Props**: None; manages its state internally with hooks.

**Reusable Components**:

* + While not explicitly defined, the card layout in Songs.js could be refactored into a SongCard component:
    - **Purpose**: Display individual song details (image, title, genre, singer, audio player).
    - **Props**: item (song data), isInWishlist, isInPlaylist, onWishlistToggle, onPlaylistToggle.
  + This abstraction would improve code reusability across Songs, Favorites, and Playlist views.

**8. STATE MANAGEMENT**

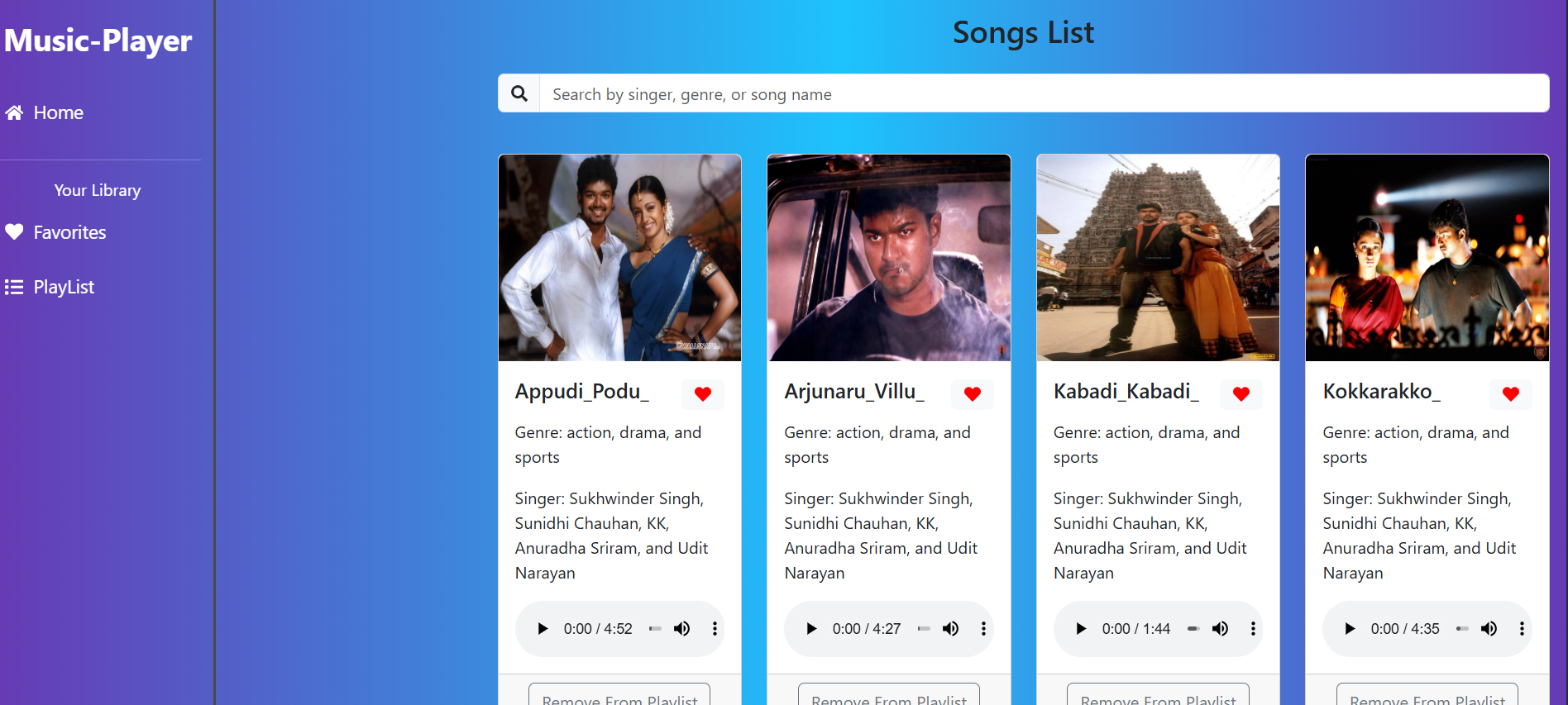
**Global State**:  
The application’s state is managed within the Songs component using React hooks:

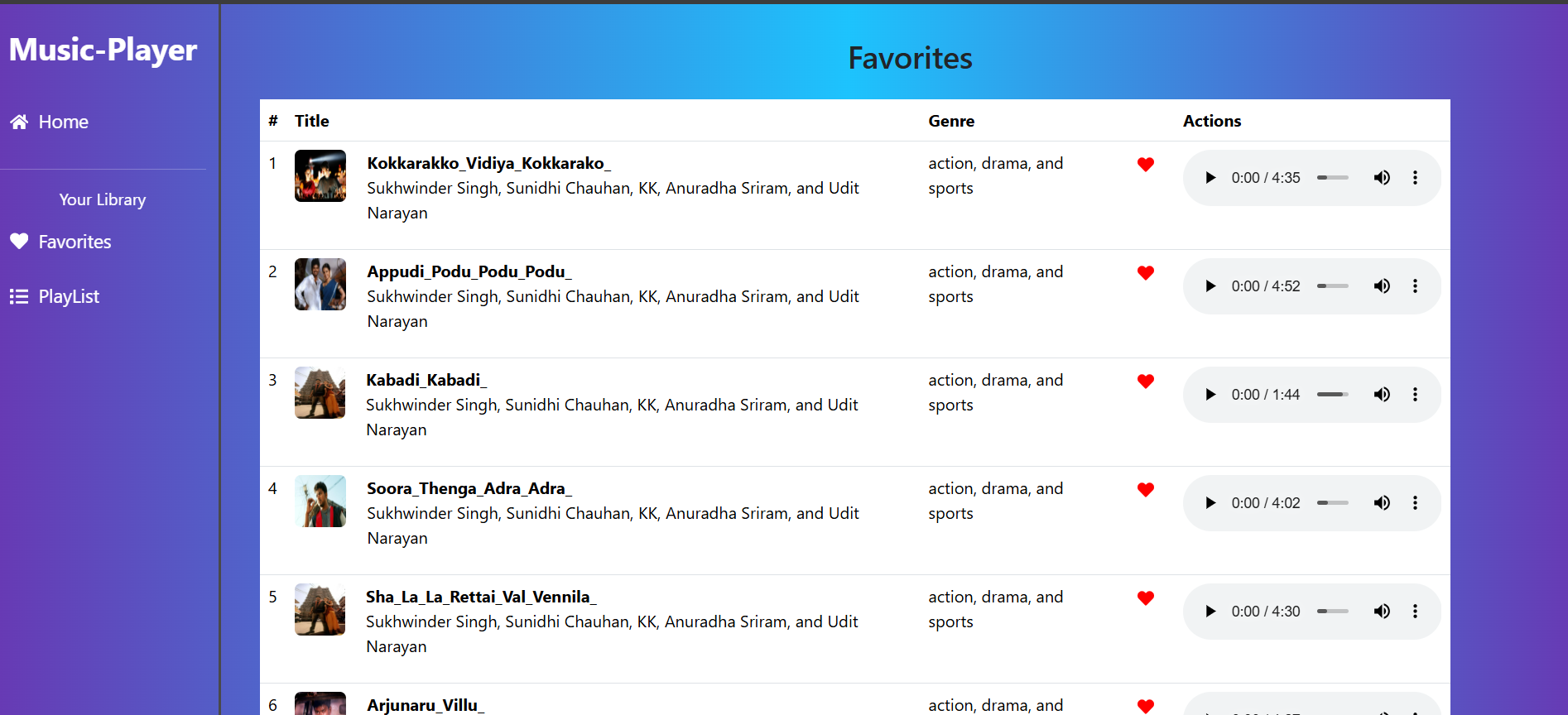
* + items: An array of all available songs fetched from http://localhost:3000/items.
  + wishlist: An array of favorited song IDs from http://localhost:3000/favorities.
  + playlist: An array of playlist song IDs from http://localhost:3000/playlist.
  + currentlyPlaying: A reference to the active <audio> element, ensuring only one song plays at a time.
  + searchTerm: A string capturing the user’s search input, used to filter items.  
    The useEffect hook fetches data on component mount, updating these states asynchronously with Axios. This approach keeps state management simple but may require refactoring (e.g., to Context API) as the app scales.

**Local State**:  
 Local state is minimal, primarily the searchTerm within Songs for real-time filtering. This ensures that search updates are instantaneous, providing a smooth user experience without unnecessary re-renders elsewhere.

**9. USER INTERFACE**

**Screenshots**:





**10. STYLING**

**CSS Frameworks/Libraries**:

* + **Bootstrap**: Used for its grid system (e.g., row, col) and prebuilt components like cards and buttons, ensuring a responsive and polished UI with minimal effort.
  + **Tailwind CSS**: Mentioned as an alternative; if implemented, it would offer utility-first styling for greater customization.
  + **Custom CSS**: App.css contains additional styles for layout tweaks (e.g., flexbox alignment) and component-specific designs not covered by Bootstrap.

**Theming**:

* + The current implementation lacks a formal theming system, relying on Bootstrap’s default palette (e.g., primary buttons, card styles). Future iterations could introduce a custom design system with light/dark modes using CSS variables or a library like Styled-Components for dynamic theming.

**11. TESTING**

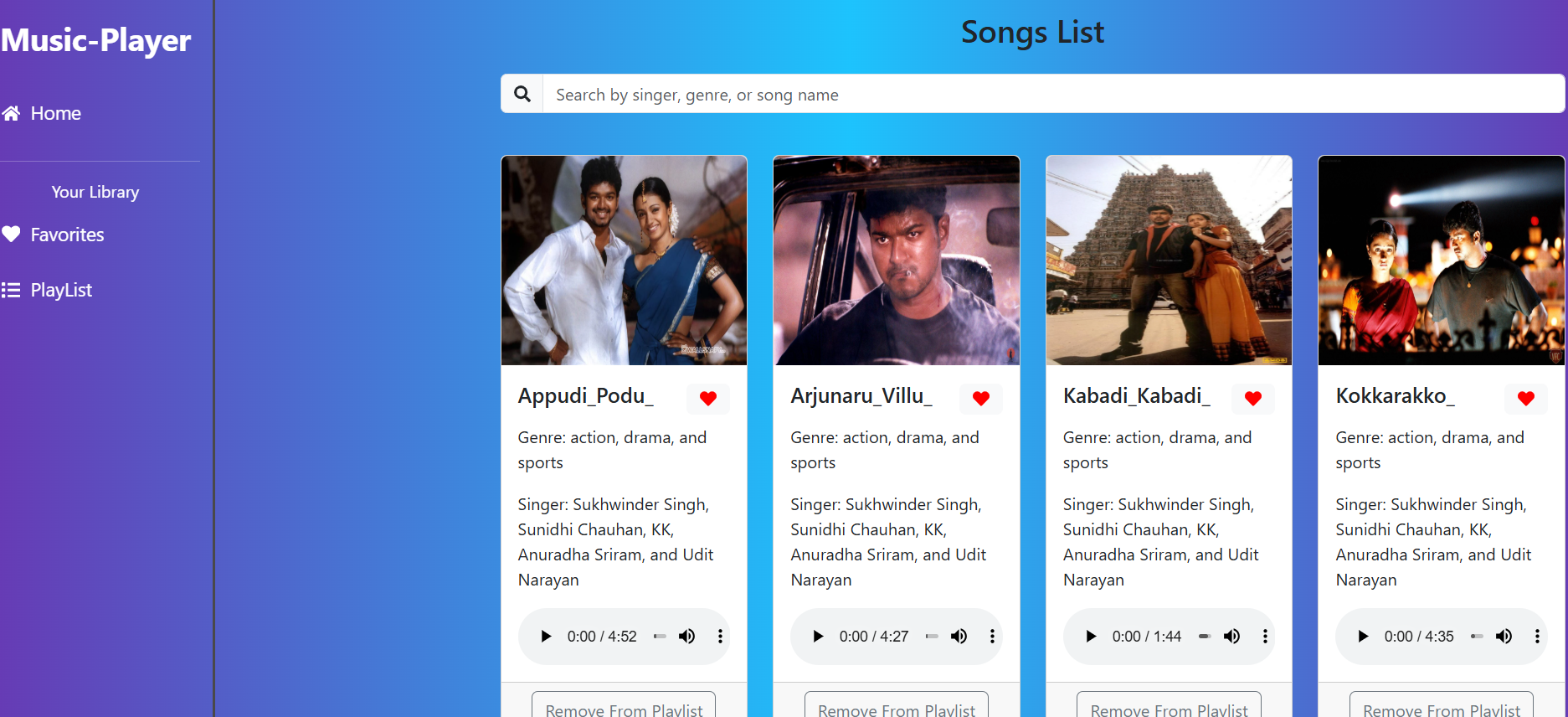
**Testing Strategy**:

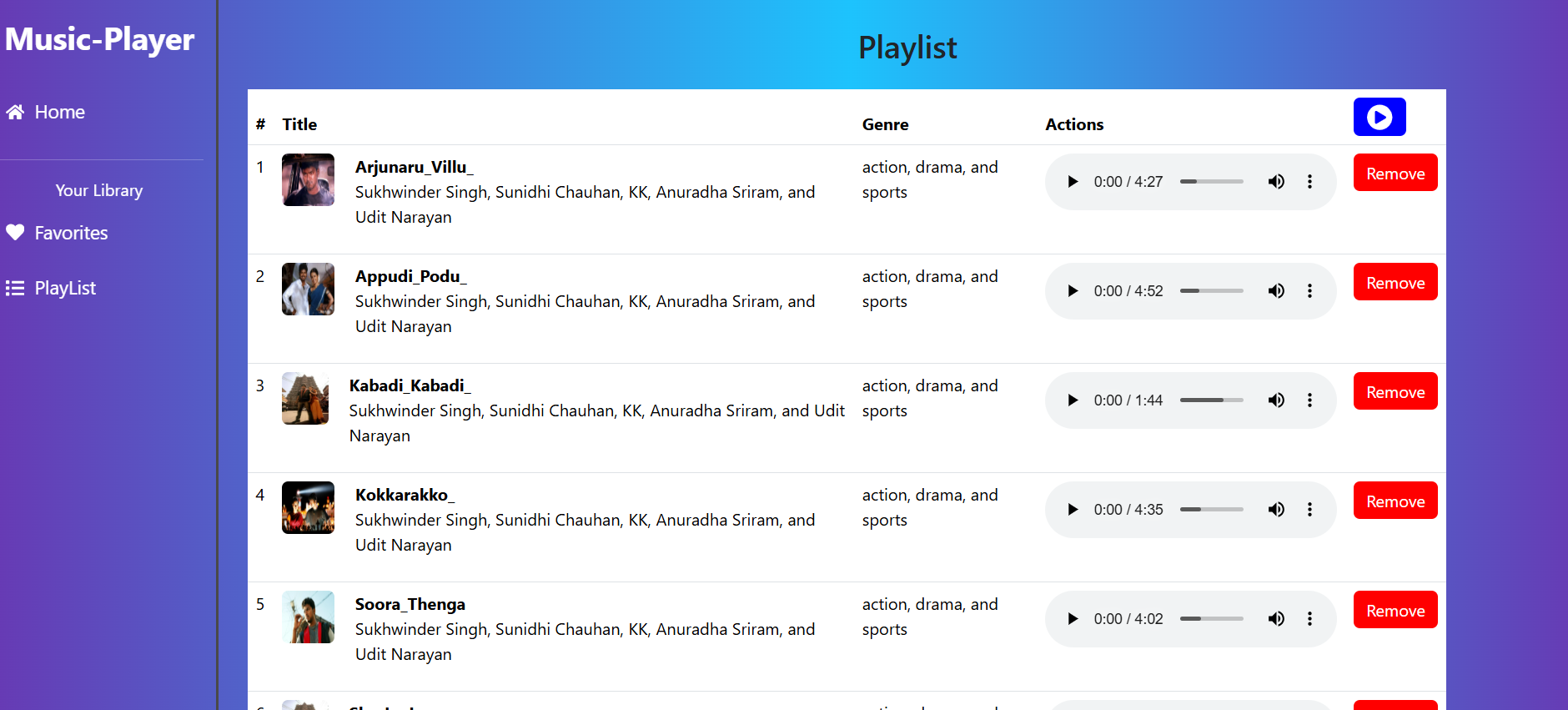
* + The provided document doesn’t detail testing, but a robust strategy would include:
    - **Unit Testing**: Test individual components (e.g., Songs, Sidebar) with Jest to verify rendering and state updates.
    - **Integration Testing**: Use React Testing Library to simulate user interactions (e.g., searching, adding to playlist) across components.
    - **End-to-End Testing**: Tools like Cypress could validate the full app flow (e.g., navigating from Songs to Playlist).

**Code Coverage**:

* + No coverage metrics are currently tracked. Integrating Jest’s --coverage flag would provide insights into tested code, aiming for at least 80% coverage of critical paths (e.g., playback, state updates).

**12. SCREENSHOTS**

****

****

**13. KNOWN ISSUES**

* **Incomplete Code**: Some documentation sections (e.g., Fetching Songs) lack full implementation details, possibly due to OCR errors or unfinished drafts.
* **Audio Playback**: While handleAudioPlay prevents multiple songs from playing simultaneously, edge cases (e.g., rapid clicking) might disrupt playback flow.
* **Error Handling**: API failures (e.g., JSON server offline) are logged but not surfaced to the user, potentially leading to a silent failure experience.
* **Scalability**: The current state management approach may struggle with larger datasets or additional features, necessitating a more robust solution.

**14. Future Enhancements**

* **User Authentication**: Add login/signup functionality to persist user data (favorites, playlists) across devices, possibly with Firebase or a custom backend.
* **Advanced State Management**: Transition to Redux or Context API for centralized state, improving maintainability as the app grows.
* **UI Enhancements**: Introduce animations (e.g., fade-ins for cards) and a custom theme toggle (light/dark mode) to elevate the visual experience.
* **Testing Suite**: Implement a comprehensive test suite with Jest and React Testing Library to ensure reliability and catch regressions.
* **Backend Integration**: Replace JSON Server with a real API (e.g., Spotify API) for dynamic song data and broader music access.