PROJECT DOCUMENTATION

INTRODUCTION

Project Title: Rhythmic Tunes - A Music Player

TEAM MEMBERS	EMAIL ID
Umamageswari S	umamageswari2905@gmail.com
Sanjjana SR	Sanjjanasunil12@gmail.com
Swathi R	swathiselvi21@gmail.com
Sivameena P	Psivameena32@gmil.com
Swetha N	swesne7@gmail.com

PROJECT OVERVIEW:

The purpose of rhythmic tunes is to create a structured and organized pattern of sounds, often serving to enhance the emotional impact, convey a sense of movement, or provide a sense of time in music.

PURPOSE:

Rhythmic Tunes is a **modern music player** built using **React.js**, designed to provide a seamless and interactive audio experience. The project aims to create a **lightweight, user-friendly**, and **visually appealing** music player that allows users to enjoy their favorite tracks with intuitive controls.

The main goal of this project is to:

☑ Provide a **smooth** and **responsive** music playback experience

☑ Implement **essential** music player controls like play, pause, next, previous, and volume adjustment

☑ Ensure a **simple and clean UI** for easy navigation

☑ Offer **playlist management** for organizing and playing multiple tracks

FEATURES:

✔ Play/Pause Music – Easily start or stop playback with a single click
✔ Track Navigation – Skip to the next or previous song in the playlist
✔ Volume Control – Adjust audio levels using a slider
✔ Seek Bar – Track the progress of the currently playing song and seek to a specific part
✔ Playlist Management – Display and manage a list of songs for playback
✔ Responsive UI – Works smoothly across different screen sizes and devices

ARCHITECTURE

Component Structure:

Rhythmic Tunes/

```
├── src/
│   ├── components/
│   │   ├── PlayerControls.js   # Play, Pause, Next, Previous controls
│   │   ├── Playlist.js         # Displays available songs
│   │   ├── SongCard.js         # UI for each song
│   │   ├── AudioPlayer.js      # Handles music playback
│   │   ├── Navbar.js           # Navigation bar
│   │   ├── Sidebar.js          # Sidebar navigation
│   ├── pages/
```

│ │ │── Home.js # Home page

│ │ │── Library.js

Saved songs

│ │ │── Search.js # Search functionality

State Management:

- **Context API** is used for managing the global state (current song, play status, etc.).
- **useState & useContext Hooks** handle state updates.

Routing:

- **React Router** is used for navigation.
- Routes include: / (Home), /library, /search

SETUP INSTRUCTIONS

Prerequisites:

- Node.js (v16+)
- npm or yarn

Installation:

1. Clone the repository:
2. git clone https://github.com/your-repo/rhythmic-tunes.git

cd rhythmic-tunes

3. Install dependencies:

npm install

4. Start the development server:

npm start

- **Installation of required tools**:

1. Open the project folder to install necessary tools In this project, we use:

  - React Js o React Router Dom o React Icons o Bootstrap/tailwind css
  - Axios

- For further reference, use the following resources o https://react.dev/learn/installation

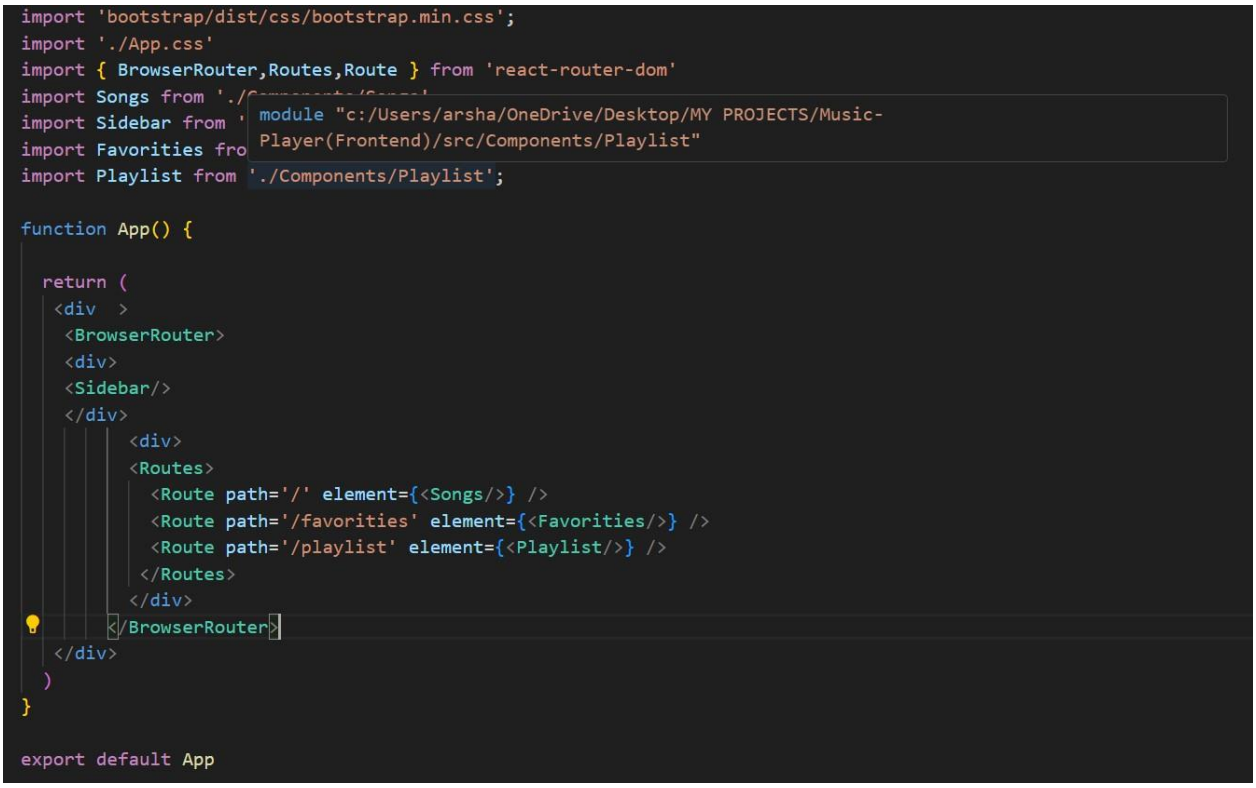
o https://react-bootstrap-v4.netlify.app/getting-started/introduction/ o https://axios-http.com/docs/intro o https://reactrouter.com/en/main/start/tutorial

Milestone 2: Project Development:

1. Setup React Application:

- Create React application.
- Configure Routing.
- Install required libraries.

Setting Up Routes:-



Code Description:-

- Imports Bootstrap CSS (bootstrap/dist/css/bootstrap.min.css) for styling components.
- Imports custom CSS (./App.css) for additional styling.
- Imports BrowserRouter, Routes, and Route from react-router-dom for setting up client-side routing in the application.

- Defines the App functional component that serves as the root component of the application.
- Uses BrowserRouter as the router container to enable routing functionality.
- Includes a div as the root container for the application.
- Within BrowserRouter, wraps components inside two div containers:
    - The first div contains the Sidebar component, likely serving navigation or additional content.
    - The second div contains the Routes component from React Router, which handles rendering components based on the current route.
    - Inside Routes, defines several Route components:
    - Route with path='/' renders the Songs component when the root path is accessed (/).
    - Route with path='/favorities' renders the Favorities component when the /favorities path is accessed.
    - Route with path='/playlist' renders the Playlist component when the /playlist path is accessed.
- Exports the App component as the default export, making it available for use in other parts of the application.

Fetching Songs:-

```
import React, { useState, useEffect } from 'react';
import { Button, Form, InputGroup } from 'react-bootstrap';
import { FaHeart, FaRegHeart, FaSearch } from 'react-icons/fa';
import axios from 'axios';
import './sidebar.css'

function Songs() {
  const [items, setItems] = useState([]);
  const [wishlist, setWishlist] = useState([]);
  const [playlist, setPlaylist] = useState([]);
  const [currentlyPlaying, setCurrentlyPlaying] = useState(null);
  const [searchTerm, setSearchTerm] = useState('');

  useEffect(() => {
    // Fetch all items
    axios.get('http://localhost:3000/items')
      .then(response => setItems(response.data))
      .catch(error => console.error('Error fetching items: ', error));

  // Fetch favorities items
      axios.get('http://localhost:3000/favorities')
      .then(response => setWishlist(response.data))
      .catch(error => {
        console.error('Error fetching Favvorities:', error);
        // Initialize wishlist as an empty array in case of an error
        setWishlist([]);
      });

    // Fetch playlist items
    axios.get('http://localhost:3000/playlist')
      .then(response => setPlaylist(response.data))
      .catch(error => {
        console.error('Error fetching playlist: ', error);
        // Initialize playlist as an empty array in case of an error
        setPlaylist([]);
      });
      // Function to handle audio play
    const handleAudioPlay = (itemId, audioElement) => {
        if (currentlyPlaying && currentlyPlaying !== audioElement) {
          currentlyPlaying.pause(); // Pause the currently playing audio
        }
        setCurrentlyPlaying(audioElement); // Update the currently playing audio

      };
```

```
      // Event listener to handle audio play
      const handlePlay = (itemId, audioElement) => {
        audioElement.addEventListener('play', () => {
          handleAudioPlay(itemId, audioElement);
        });
      };

      // Add event listeners for each audio element
      items.forEach((item) => {
        const audioElement = document.getElementById(`audio-${item.id}`);
        if (audioElement) {
          handlePlay(item.id, audioElement);
        }
      });

      // Cleanup event listeners
      return () => {
        items.forEach((item) => {
          const audioElement = document.getElementById(`audio-${item.id}`);
          if (audioElement) {
            audioElement.removeEventListener('play', () => handleAudioPlay(item.id, audioElement));
          }
        });
      };
  }, [items,currentlyPlaying, searchTerm]);

  const addToWishlist = async (itemId) => {
    try {
      const selectedItem = items.find((item) => item.id === itemId);
      if (!selectedItem) {
        throw new Error('Selected item not found');
      }
      const { title, imgUrl, genre, songUrl, singer, id: itemId2 } = selectedItem;
      await axios.post('http://localhost:3000/favorities', { itemId: itemId2, title, imgUrl, genre, songUrl, singer });
      const response = await axios.get('http://localhost:3000/favorities');
      setWishlist(response.data);
    } catch (error) {
      console.error('Error adding item to wishlist: ', error);
    }
  };
```

Code Description:-

- **useState:**
  - items: Holds an array of all items fetched from http://localhost:3000/items.

- wishlist: Stores items marked as favorites fetched from http://localhost:3000/favorities.

- playlist: Stores items added to the playlist fetched from http://localhost:3000/playlist. o currentlyPlaying: Keeps track of the currently playing audio element. o searchTerm: Stores the current search term entered by the user.

- **Data Fetching:**

  - Uses useEffect to fetch data:

▪ Fetches all items (items) from http://localhost:3000/items.

▪ Fetches favorite items (wishlist) from

http://localhost:3000/favorities.

▪ Fetches playlist items (playlist) from

http://localhost:3000/playlist. o Sets state variables (items, wishlist, playlist) based on the fetched data.

- **Audio Playback Management:**

  - Sets up audio play event listeners and cleanup for each item:

▪ handleAudioPlay: Manages audio playback by pausing the currently playing audio when a new one starts.

▪ handlePlay: Adds event listeners to each audio element to trigger handleAudioPlay.

  - Ensures that only one audio element plays at a time by pausing others when a new one starts playing.

- **addToWishlist(itemId):**

  - Adds an item to the wishlist (favorities) by making a POST request to http://localhost:3000/favorities. o Updates the wishlist state after adding an item.

- **removeFromWishlist(itemId):**

- Removes an item from the wishlist (favorities) by making a DELETE request to http://localhost:3000/favorities/{itemId}. o Updates the wishlist state after removing an item.

- **isItemInWishlist(itemId):**

  - Checks if an item exists in the wishlist (favorities) based on its itemId.

- **addToPlaylist(itemId):**

o Adds an item to the playlist (playlist) by making a POST request to http://localhost:3000/playlist. o Updates the playlist state after adding an item.

- **removeFromPlaylist(itemId):**

o Removes an item from the playlist (playlist) by making a DELETE request to http://localhost:3000/playlist/{itemId}. o Updates the playlist state after removing an item.

- **isItemInPlaylist(itemId):**

o Checks if an item exists in the playlist (playlist) based on its itemId.

- **filteredItems:**

  - Filters items based on the searchTerm.

  - Matches title, singer, or genre with the lowercase version of searchTerm.

- **JSX:**

  - Renders a form with an input field (Form, InputGroup, Button, FaSearch) for searching items.

  - Maps over filteredItems to render each item in the UI.

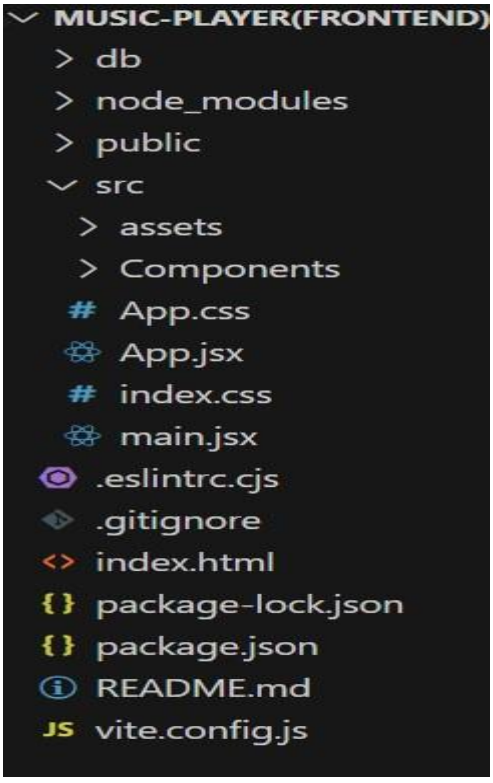
  - Includes buttons (FaHeart, FaRegHeart) to add/remove items from wishlist and playlist. o Renders audio elements for each item with play/pause functionality.

- **Error Handling:**

o Catches and logs errors during data fetching (axios.get). o Handles errors when adding/removing items from wishlist and playlist.

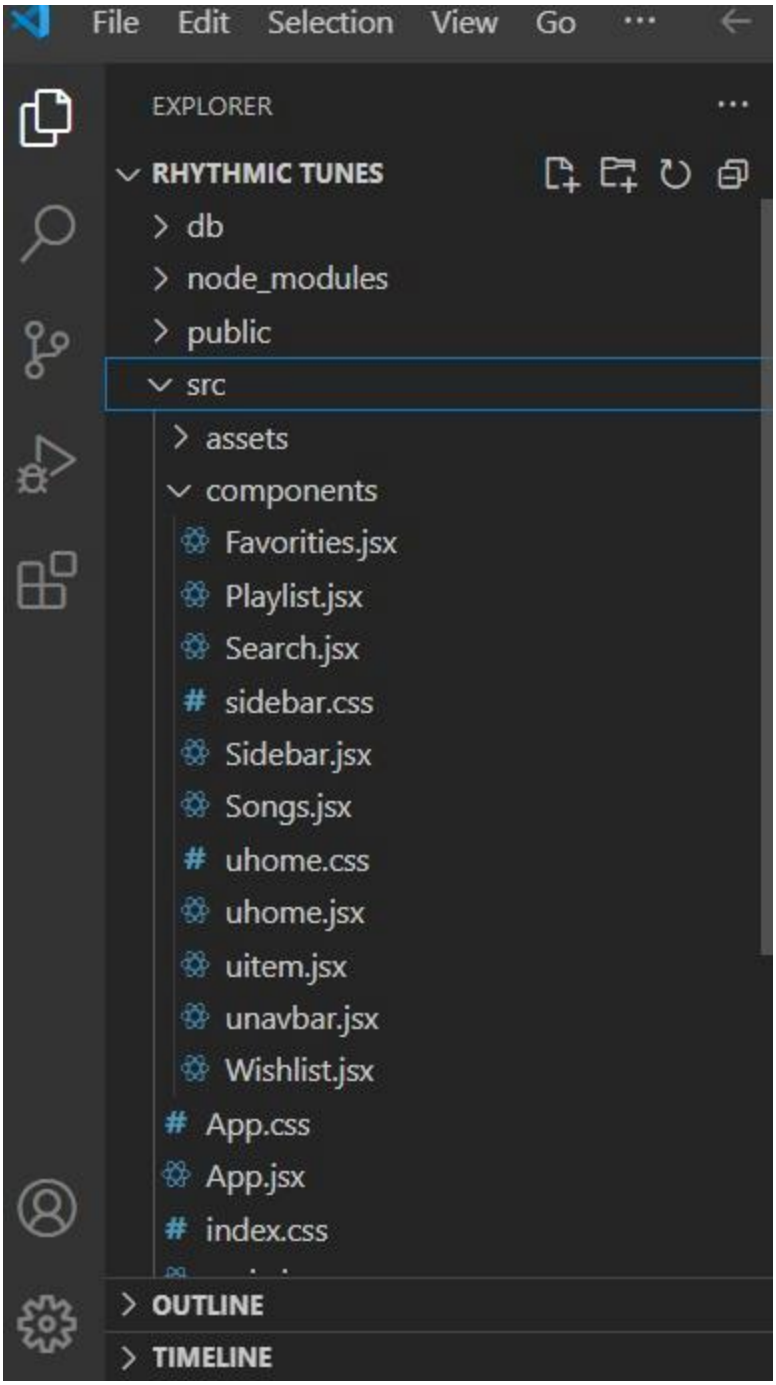
FOLDER STRUCTURE

Project structure:



The project structure may vary depending on the specific library, framework, programming language, or development approach used. It's essential to organize the files and directories in a logical and consistent manner to improve code maintainability and collaboration among developers.

app/app.component.css, src/app/app.component: These files are part of the main AppComponent, which serves as the root component for the React app. The component handles the overall layout and includes the router outlet for loading different components based on the current route.



RUNNING THE APPLICATION

To start the frontend server,
run: npm start
Then, open http://localhost:3000 in your browser.

COMPONENT DOCUMENTATION

KEY COMPONENTS

Below are the major components of the **CookBook** application, along with their purpose and props:

1. Navbar Component (Navbar.js)

Purpose:

- Provides site-wide navigation.
- Contains links to **Home**, **Categories**, and **Search**.

Props:

- logo (string): Path to the logo image.
- menuItems (array): List of menu items for navigation.

2. Hero Component (Hero.js)

Purpose:

- Displays an introduction to the application.
- Contains a call-to-action button to explore recipes.

Props:

- title (string): Main heading text.
- subtitle (string): Supporting description text.

- buttonText (string): Label for the action button.

3. Recipe Card Component (RecipeCard.js)

Purpose:

- Displays a brief summary of a recipe, including an image, title, and category.
- Redirects users to the detailed recipe page when clicked.

Props:

- recipe (object): Contains id, title, image, and category.

4. Category Component (Category.js)

Purpose:

- Displays different categories of meals.
- Allows users to filter recipes by category.

Props:

- categoryName (string): Name of the category.
- image (string): Category thumbnail.

5. Recipe Details Component (RecipeDetails.js)

Purpose:

- Displays a full recipe, including ingredients, instructions, and a demo video.

Props:

- recipeId (string): Unique ID to fetch the recipe details.

REUSABLE COMPONENTS

- **AudioPlayer.js** - Handles playing songs.
- **Playlist.js** - Displays songs.
- **PlayerControls.js** - Controls music playback.

Reusable Components:

- **SongCard.js** - Represents a song item.
- **Navbar.js** - Navigation bar for routing.

STATE MANAGEMENT

Global State:

- Context API is used to manage the current song, playback state, and playlist.

Local State:

- Components use useState for local UI interactions (e.g., toggling search bar).

USER INTERFACE

- **Container Setup:**
    - Uses a div with inline styles (style={{display:"flex", justifyContent:"flex-end"}}) to align the content to the right.
    - The main container (songs-container) has a fixed width (width:"1300px") and contains all the UI elements related to songs.
- **Header:**
    - Displays a heading (<h2>) with text "Songs List" centered (className="text-3xl font-semibold mb-4 text-center").

- **Search Input:**
    - Utilizes InputGroup from React Bootstrap for the search functionality.
    - Includes an input field (Form.Control) that allows users to search by singer, genre, or song name.
    - Binds the input field value to searchTerm state (value={searchTerm}) and updates it on change (onChange={(e) => setSearchTerm(e.target.value)}).
    - Styled with className="search-input".
- **Card Layout:**
    - Uses Bootstrap grid classes (row, col) to create a responsive card layout

(className="row row-cols-1 row-cols-md-2 row-cols-lg-3 row-cols-xl-4 g-4").

    - Maps over filteredItems array and renders each item as a Bootstrap card

( ). ● **Card Content:**

    - Displays the item's image (<img>), title (<h5

className="card-title">), genre (<p className="card-text">), and singer (<p className="card-text">).

    - Includes an audio player (<audio controls className="w-100" id={audio-${item.id}}>) for playing the song with a source (<source src={item.songUrl} />).

- **Wishlist and Playlist Buttons:**
    - Adds a heart icon button (<Button>) to add or remove items from the wishlist

(isItemInWishlist(item.id) determines which button to show).

    - Includes an "Add to Playlist" or "Remove From Playlist" button (<Button>) based on whether the item is already in the playlist

(isItemInPlaylist(item.id)).

- **Button Click Handlers:**
    - Handles adding/removing items from the wishlist

(addToWishlist(item.id), removeFromWishlist(item.id)).

    - Manages adding/removing items from the playlist

(addToPlaylist(item.id), removeFromPlaylist(item.id)).

- **Card Styling:**
    - Applies Bootstrap classes (card, card-body, card-footer) for styling the card components.
    - Uses custom styles (rounded-top, w-100) for specific elements like images and audio players.

STYLING

The Recipe Application is designed with a modern, responsive, and clean aesthetic using industry-standard styling techniques.

1. CSS Frameworks & Libraries Used

Bootstrap/Tailwind CSS:

- Bootstrap is a widely used CSS framework that provides a grid system, predefined styling components, and responsiveness without writing extensive custom CSS.
- Tailwind CSS is a utility-first framework that allows for highly customized and flexible styling, reducing the need for external stylesheets.
- The combination of these frameworks ensures:
    - Consistent layouts across all screen sizes (mobile, tablet, desktop).
    - Reusable components such as buttons, cards, modals, and forms.

Faster development due to pre-built styles.

React Icons:

- Used for UI elements such as search icons, navigation arrows, social media links, and buttons.
- Helps improve user navigation and interaction, making the interface more visually appealing**.**

2. **Theming & Custom Design**

Consistent Color Scheme**:**

- The application follows a uniform color palette for a clean and readable UI • Colors are chosen to provide good contrast and accessibility.
- Example:
  - Primary Color: Used for buttons and highlights.
  - Secondary Color: Used for backgrounds and supporting elements.
  - Text Color: Optimized for readability.

Dark Mode & Light Mode**:**

- Dark Mode is implemented to provide a comfortable user experience, especially in low-light environments.
- Users can toggle between Light Mode (default) and Dark Mod**e**, reducing eye strain.
- Future Enhancement: Saving user preference in local storage to persist across sessions.

Custom CSS Modules & Styled Components**:**

CSS Modules**:**

- Ensures that styles are scoped to specific components, preventing unwanted global style conflicts.

Styled Components**:**

- Allows writing CSS directly within JavaScript files.
- Improves maintainability by keeping styles within relevant components.
- Enables dynamic styling based on **props** (e.g., different styles for different recipe categories).

3. Animations & Transitions

Smooth UI Interactions**:**

- CSS animations and React libraries (like Framer Motion) are used to create a seamless browsing experience.
- Examples:
- Hover effects on buttons & images to indicate interactivity.
- Fade-in animations for content appearing dynamically.
- Slide transitions when navigating between pages.

Performance Optimization:

- Avoiding excessive animations to keep the app lightweight.
- Using lazy loading for images and content to enhance page load speed.

TESTING

Ensuring the reliability and performance of the application is critical. The testing approach includes multiple testing strategies

- Jest provides code coverage reports to ensure all critical functionalities are tested.
- Run tests with coverage using:
  npm test -- --coverage
- The coverage report includes:
  - **Statements Coverage:** Ensures all executable statements are tested.
  - **Branch Coverage:** Checks all code paths and conditional statements.
  - **Function Coverage:** Ensures all functions are invoked in tests.
  - **Line Coverage:** Validates that all lines of code execute as expected.

SCREENSHOTS OR DEMO

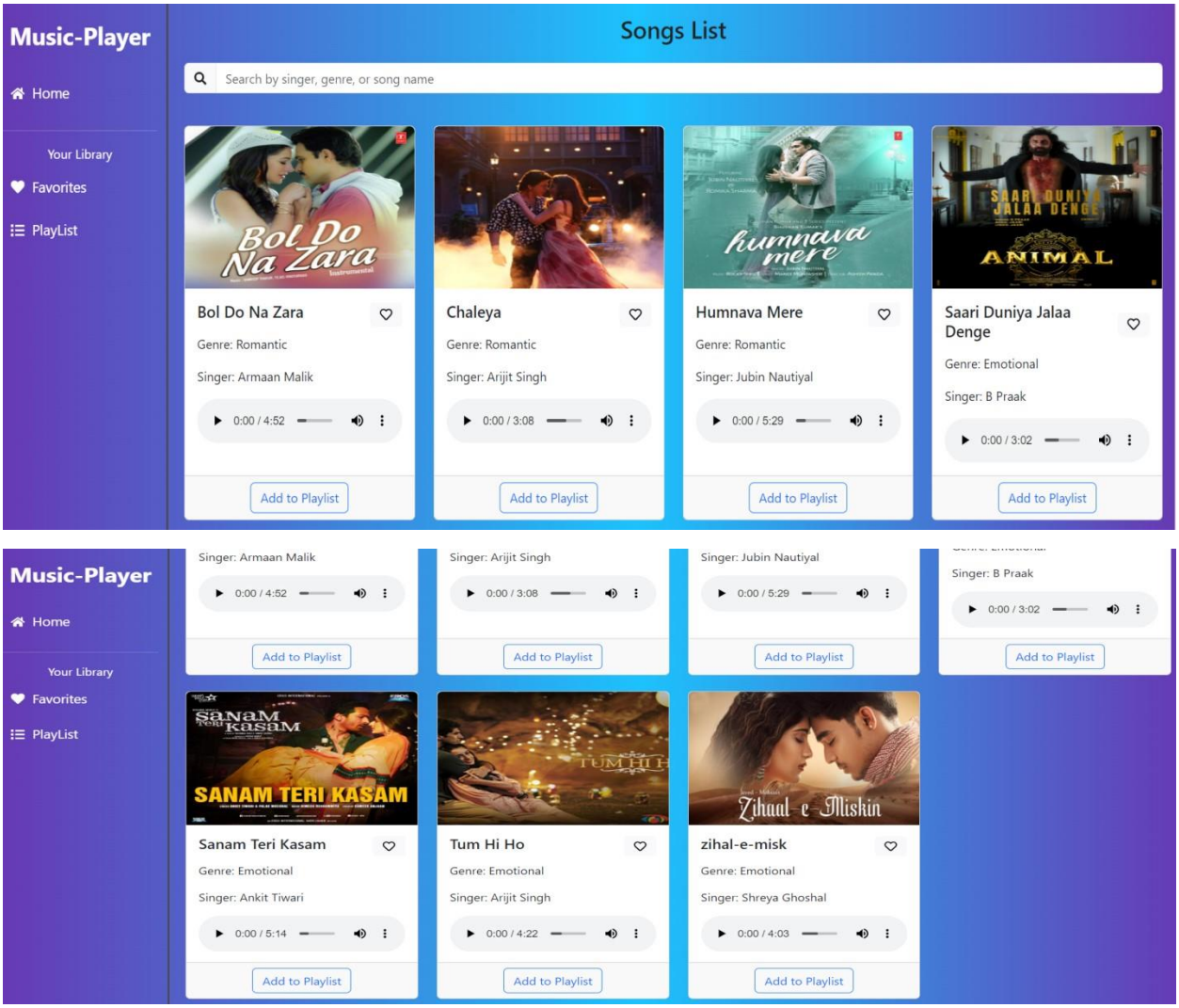
After completing the code, run the react application by using the command “npm start” or “npm run dev” if you are using vite.js

And the Open new Terminal type this command “json-server --watch ./db/db.json” to start the json server too.

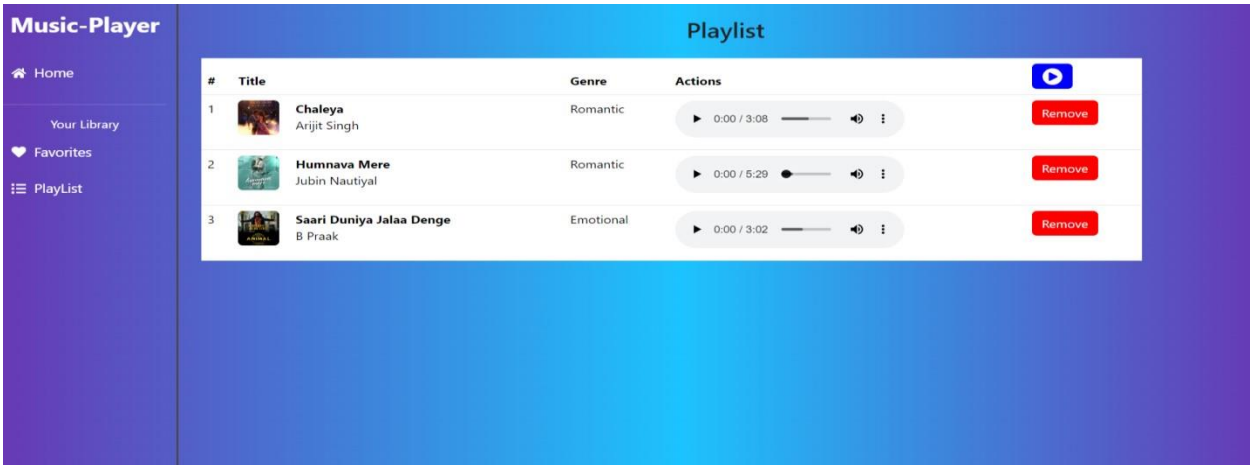
After that launch the Rythimic Tunes.

Here are some of the screenshots of the application.

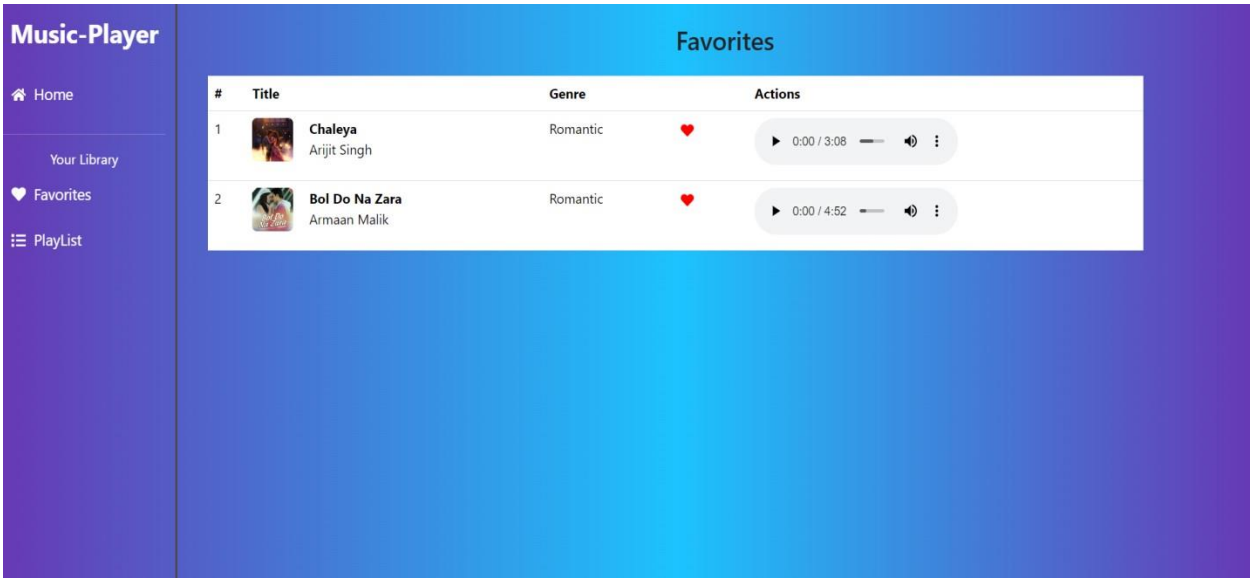
Hero components



Playlist



Favorites



KNOWN ISSUES

Issues:

- Song buffering on slow networks.
- Search optimization needed.

FUTURE ENHANCEMENTS

- **AI-based song recommendations** - Implementing machine learning to suggest songs based on user preferences.
- **Real-time lyrics display** - Syncing lyrics dynamically as the song plays.
- **UI animations and transitions** - Enhancing the user experience with smooth animations.
- **Offline mode** - Allow users to download and listen to music offline.
- **Social sharing** - Enable users to share their favorite songs or playlists on social media.
- **Multi-device synchronization** - Sync playlists and playback across multiple devices.
- **Voice commands** - Integrate voice recognition to play, pause, or search songs.
- **Podcast support** - Expand the player to include podcast streaming functionality.