RhythmicTunes: Your Melodic Companion

1. Introduction:

Project tittle:rhythmic tunes application
Team members: M.Harikrishnan.
R.Dhanush.
R.Imarn.
S.Dhanraj.

1.1 Project Overview

RhythmicTunes is a music streaming application that provides users with a seamless and personalized music experience. The platform allows users to listen to their favorite songs, discover new music, create custom playlists, and share music with friends. By offering music recommendations tailored to individual tastes and preferences, RhythmicTunes aims to create a unique and engaging music experience.

1.2 Problem Statement

With the increasing number of music streaming platforms, there is a growing demand for personalized music experiences. While many music streaming services exist, few effectively tailor recommendations to user preferences and provide an intuitive interface that facilitates music discovery and interaction. RhythmicTunes seeks to fill this gap by focusing on personalized music suggestions, ease of use, and community interaction.

1.3 Project Objectives

- To develop a music streaming application that offers a personalized and enjoyable user experience.
- To incorporate features such as music streaming, playlist creation, and social sharing.
- To implement a recommendation engine that suggests music based on user preferences.
- To provide seamless cross-platform support for mobile and desktop users.

2. System Design and Architecture

2.1 System Architecture

The system follows a **Client-Server** model, where the client (user-facing interface) communicates with the server (back-end system) to deliver music streaming and management functionalities.

2.1.1 Components:

- **Frontend (Client-Side)**: The frontend is built using **React.js** (for web) or **React Native** (for mobile) to provide a responsive, intuitive, and interactive user interface.
- **Backend (Server-Side)**: The backend is developed using **Node.js** with **Express.js** for handling client requests, user authentication, and music streaming functionalities.
- **Database**: A **MongoDB** database is used to store user information, playlists, song metadata, and other necessary data. Alternatively, **MySQL** can be used for relational data storage.
- **Streaming API**: For delivering music content, RhythmicTunes integrates with external music services such as **Spotify API** or **SoundCloud API** to stream tracks and gather metadata.
- **Recommendation System**: A machine learning-based recommendation engine provides personalized song, playlist, and artist recommendations.

2.1.2 Data Flow

1. **User Interaction**: The user interacts with the front-end, selecting songs or searching for artists.
2. **Request Handling**: The frontend sends a request to the backend (server) for specific data (music, playlists, recommendations).
3. **Data Processing**: The backend processes the request, querying the database or external music API for the relevant data.
4. **Response**: The backend returns the requested data to the frontend, which is then presented to the user.

---

3. Features and Functionality

3.1 User Authentication

- **Sign Up / Login**: Users can sign up with an email address or log in through social media accounts (Google/Facebook).

- **User Profile**: Users can manage their profile, including setting preferences and uploading profile pictures.

3.2 Music Streaming

- **Search**: Users can search for songs, albums, or artists.
- **Play Music**: Users can play, pause, skip, shuffle, and adjust the volume of music tracks.
- **Offline Mode**: Option to download music for offline listening.
- **Now Playing**: A section to show the currently playing track, including the album artwork, song title, and artist.

3.3 Playlist Management

- **Create Playlist**: Users can create custom playlists by adding songs of their choice.
- **Playlist Sharing**: Users can share their playlists with friends via a shareable link.
- **Playlist Discovery**: Browse popular playlists and discover new music tailored to the user's interests.

3.4 Recommendations

- **Personalized Recommendations**: Based on user listening history and preferences, the application recommends new songs, albums, and artists.
- **Trending Music**: Displays trending music and albums popular with other users in the community.

3.5 Social Features

- **Follow Users/Artists**: Users can follow artists and other users to stay updated on their activities.
- **Comments & Likes**: Users can comment on songs and playlists, as well as like or dislike tracks to further refine recommendations.
- **Social Sharing**: Share songs and playlists on social media platforms.

3.6 User Interface

- **Responsive UI**: Designed to work across multiple devices, including desktops, smartphones, and tablets.
- **Dark Mode**: A dark mode feature for users who prefer a darker interface during night-time listening.

---

4. Technologies Used

4.1 Front-End

- **React.js**: For building the user interface of the web app.
- **React Native**: For building the mobile application (Android/iOS).
- **Redux**: For state management across components.
- **HTML, CSS, JavaScript**: Core technologies used for building the web UI and styling.

4.2 Back-End

- **Node.js**: For server-side JavaScript execution.
- **Express.js**: For building the RESTful API to handle HTTP requests and responses.
- **MongoDB** / **MySQL**: Database management system used to store user data, playlists, and song information.
- **JWT (JSON Web Token)**: For user authentication and secure API communication.

4.3 Third-Party APIs

- **Spotify API** / **SoundCloud API**: Used to fetch music tracks, album details, and metadata for streaming.
- **Machine Learning**: For building a recommendation system based on user preferences (e.g., collaborative filtering or content-based filtering).

4.4 Deployment

- **Heroku/AWS**: Cloud hosting platforms for deploying the web and backend applications.
- **Git/GitHub**: For version control and collaboration during development.
- **Docker**: Containerization of the application for easier deployment and scalability.

---

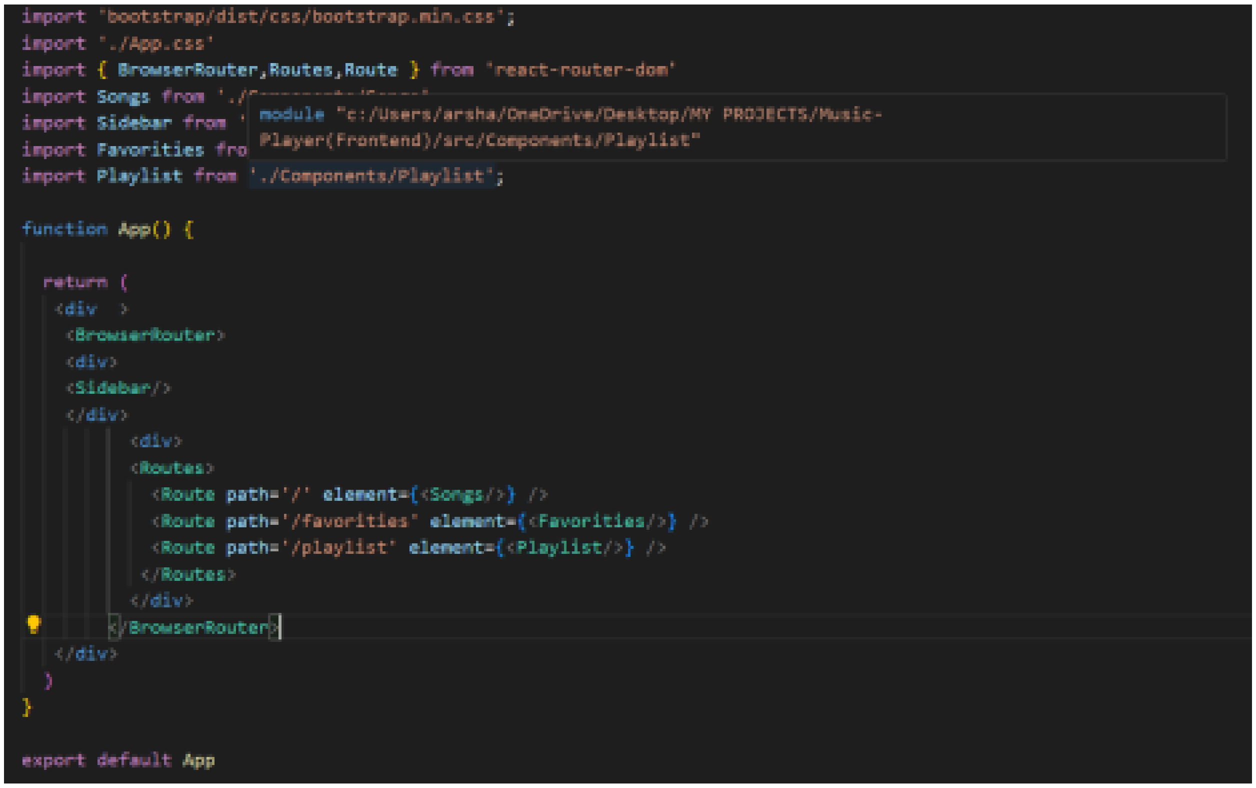
5. Database Design

The database design is essential for storing user data, song metadata, playlists, and user interactions.

5.1 Schema Design

- **Users Table**: Stores user information (ID, name, email, password, etc.).
- **Tracks Table**: Stores information about the tracks, including title, artist, album, genre, and file path.
- **Playlists Table**: Stores user-created playlists, including playlist name, user ID, and song IDs.

- **Listening History Table**: Tracks the user's listening habits, which are used for recommendations.



```
import 'bootstrap/dist/css/bootstrap.min.css';
import './App.css'
import { BrowserRouter,Routes,Route } from 'react-router-dom'
import Songs from './
import Sidebar from '
import Favorities fro
import Playlist from './Components/Playlist';

function App() {

  return (
   <div  >
    <BrowserRouter>
    <div>
    <Sidebar/>
    </div>
          <div>
          <Routes>
            <Route path='/' element={<Songs/>} />
            <Route path='/favorities' element={<Favorities/>} />
            <Route path='/playlist' element={<Playlist/>} />
           </Routes>
          </div>
        </BrowserRouter>
   </div>
  )
}

export default App
```

```
module "c:/Users/arsha/OneDrive/Desktop/MY PROJECTS/Music-
Player(Frontend)/src/Components/Playlist"
```

5.2 Relationships

- **One-to-many**: One user can have multiple playlists.
- **Many-to-many**: A playlist can contain multiple songs, and a song can appear in multiple playlists.

---

6. Testing and Quality Assurance

6.1 Types of Testing

- **Unit Testing**: Testing individual components like the login process, music streaming, and playlist creation.
- **Integration Testing**: Ensuring that different parts of the system (e.g., frontend-backend communication, database integration) work correctly together.
- **End-to-End Testing**: Testing the entire flow of the application, from user sign-up to music playback.
- **Performance Testing**: Checking how the application performs under heavy traffic and multiple concurrent users.

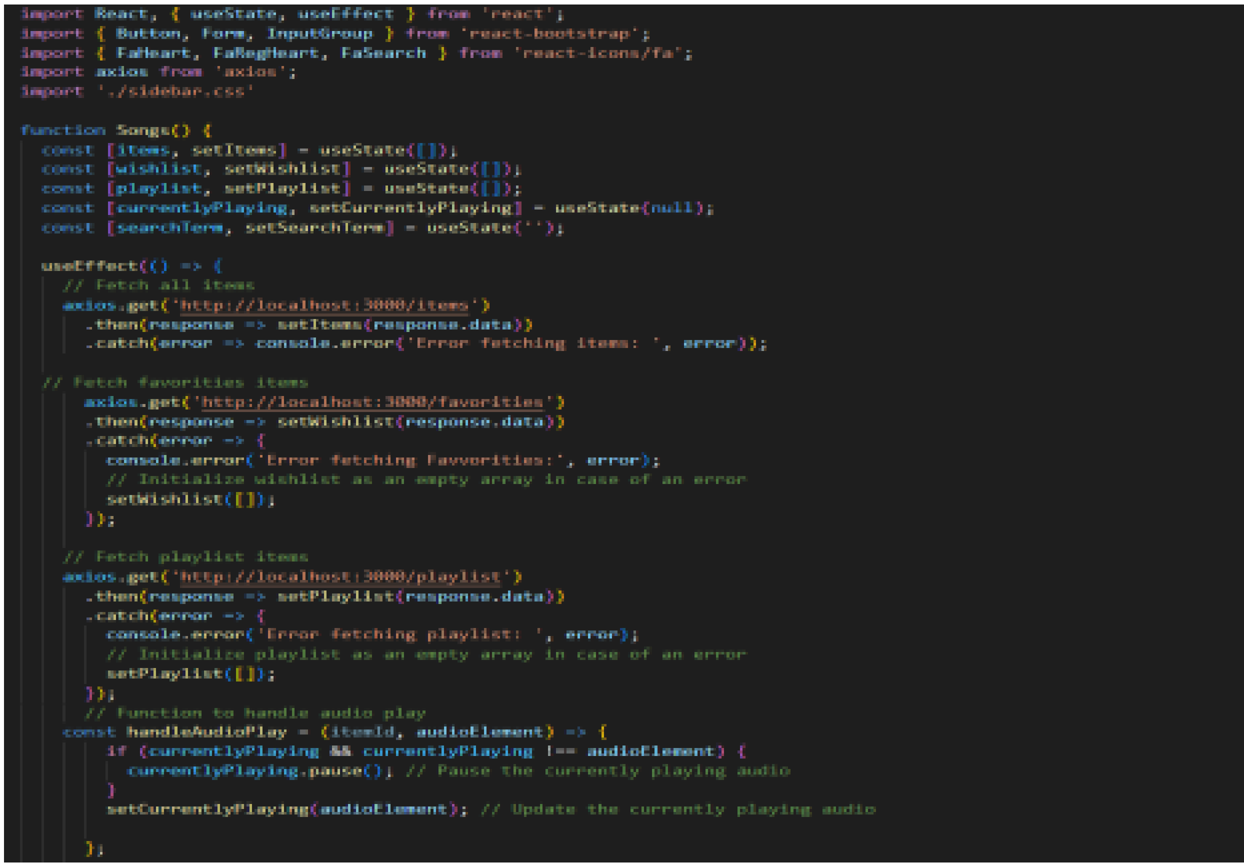
6.2 Tools Used for Testing

- **Jest**: A testing framework for JavaScript to perform unit and integration testing.
- **Mocha/Chai**: For backend testing in Node.js.
- **Selenium**: For automating UI testing.

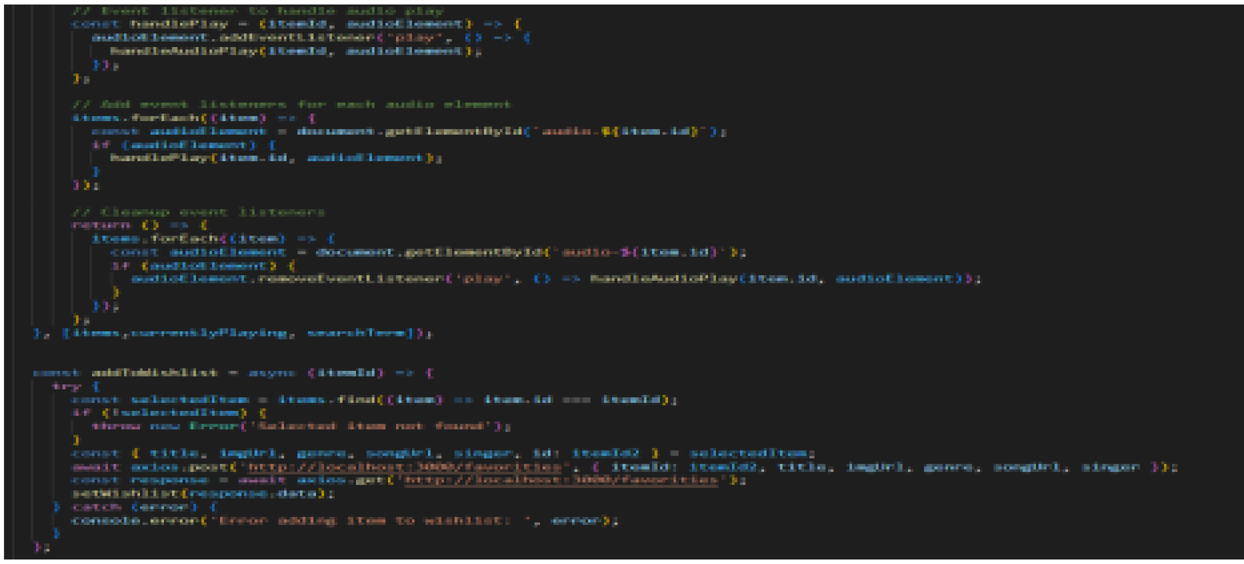
---

7. Future Enhancements

- **Voice Control**: Integrate voice assistants (e.g., Google Assistant, Siri) for hands-free control over music playback.
- **AI-Generated Playlists**: Use AI to create dynamic playlists based on the user’s mood, time of day, or activity.
- **Integration with Smart Devices**: Integrate with smart speakers like Amazon Echo, Google Home, etc.
- **Music Genre Prediction**: Predict user music genre preferences based on listening habits and provide suggestions.
- **Live Concert Streaming**: Add the feature to stream live concerts or exclusive events from artists.



```
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);
    }
  };
```

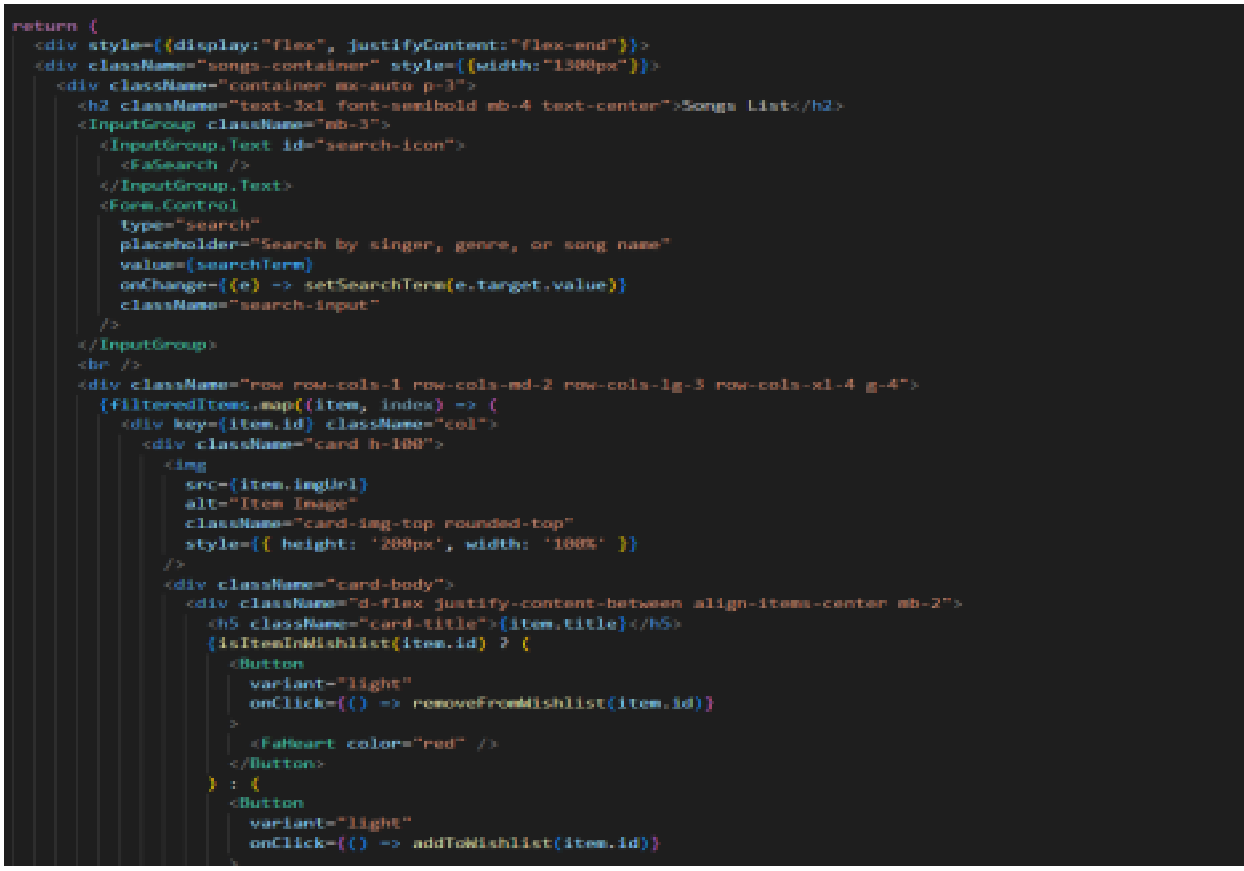
Project Execution:

After completing the code, run the react application by using the command "npm start" or

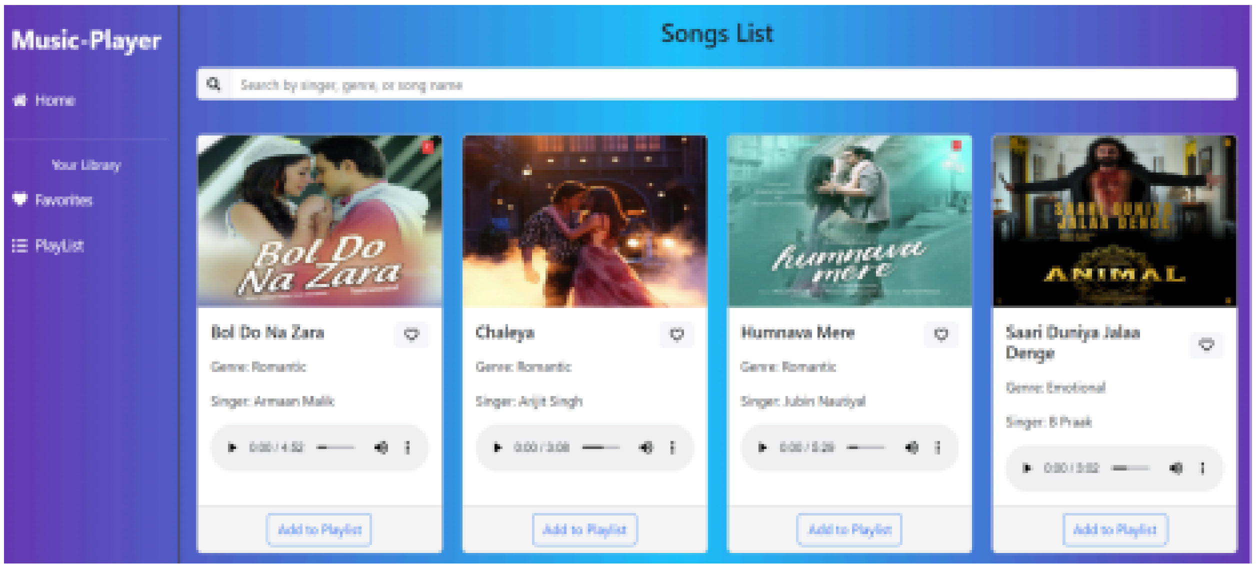
"npm run dev" if you are using vite.js

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

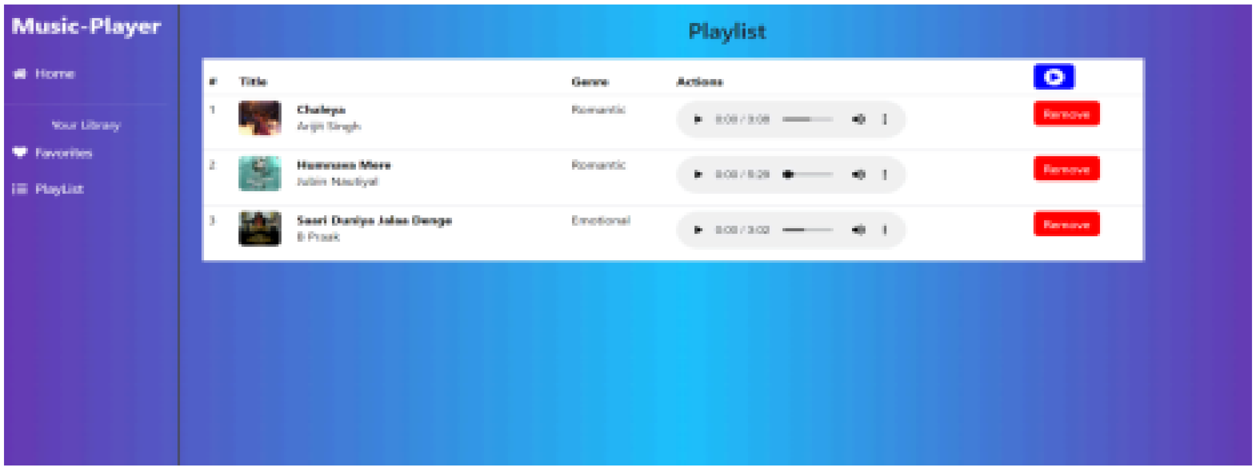
the json server too.
After that launch the Rythimic Tunes.
Here are some of the screenshots of the application.



```
return (
  <div style={{display:"flex", justifyContent:"flex-end"}}>
  <div className="songs-container" style={{width:"1300px"}}>
    <div className="container mx-auto p-3">
      <h2 className="text-3xl font-semibold mb-4 text-center">Songs List</h2>
      <InputGroup className="mb-3">
        <InputGroup.Text id="search-icon">
          <FaSearch />
        </InputGroup.Text>
        <Form.Control
          type="search"
          placeholder="Search by singer, genre, or song name"
          value={searchTerm}
          onChange={(e) => setSearchTerm(e.target.value)}
          className="search-input"
        />
      </InputGroup>
      <br />
      <div className="row row-cols-1 row-cols-md-2 row-cols-lg-3 row-cols-xl-4 g-4">
        {filteredItems.map((item, index) => (
          <div key={item.id} className="col">
            <div className="card h-100">
              <img
                src={item.imgUrl}
                alt="Item Image"
                className="card-img-top rounded-top"
                style={{ height: '200px', width: '100%' }}
              />
              <div className="card-body">
                <div className="d-flex justify-content-between align-items-center mb-2">
                  <h5 className="card-title">{item.title}</h5>
                  {isItemInWishlist(item.id) ? (
                    <Button
                      variant="light"
                      onClick={() => removeFromWishlist(item.id)}
                    >
                      <FaHeart color="red" />
                    </Button>
                  ) : (
                    <Button
                      variant="light"
                      onClick={() => addToWishlist(item.id)}
```



Bol Do Na Zara	Chaleya	Humnava Mere	Saari Duniya Jalaa Denge
Genre: Romantic	Genre: Romantic	Genre: Romantic	Genre: Emotional
Singer: Armaan Malik	Singer: Arijit Singh	Singer: Jubin Nautiyal	Singer: B Praak
0:00 / 4:32	0:00 / 3:08	0:00 / 5:26	0:00 / 3:02



Playlist

#	Title	Genre	Actions	
1	Chaleya Arijit Singh	Romantic	0:00 / 3:08	Remove
2	Humnava Mere Jubin Nautiyal	Romantic	0:00 / 5:28	Remove
3	Saari Duniya Jalaa Denge B Praak	Emotional	0:00 / 3:02	Remove