DOCUMENTATION

Project Title	**Rhythmic Tune**
Team ID	**SWTID1741164939155819**
Team Size	**4**
Team Leader	**Saranya P**
Team Member	**Desammal P**
Team Member	**Aarthi R**
Team Member	**Bharathi B**

PROJECT OVERVIEW

Purpose:

The goal of this project is to build a modern, responsive, and user-friendly music player frontend using React.js. The application will allow users to play, pause, skip, and control the volume of songs, as well as view a playlist and track progress. The focus will be on creating a seamless user experience with a clean and intuitive design

Key Features:

- **Play/Pause Functionality**: Users can play and pause songs.
- **Skip Controls**: Next and previous buttons to navigate through the playlist.
- **Volume Control**: A slider to adjust the volume.
- **Progress Bar**: Displays the current progress of the song and allows seeking.
- **Playlist Management**: Display a list of songs with the ability to select and play.
- **Responsive Design**: The player should work seamlessly on desktop, tablet, and mobile devices.
- **Repeat and Shuffle**: Options to repeat a song or shuffle the playlist.
- **Song Information**: Display the song title, artist, and album art.

Architecture

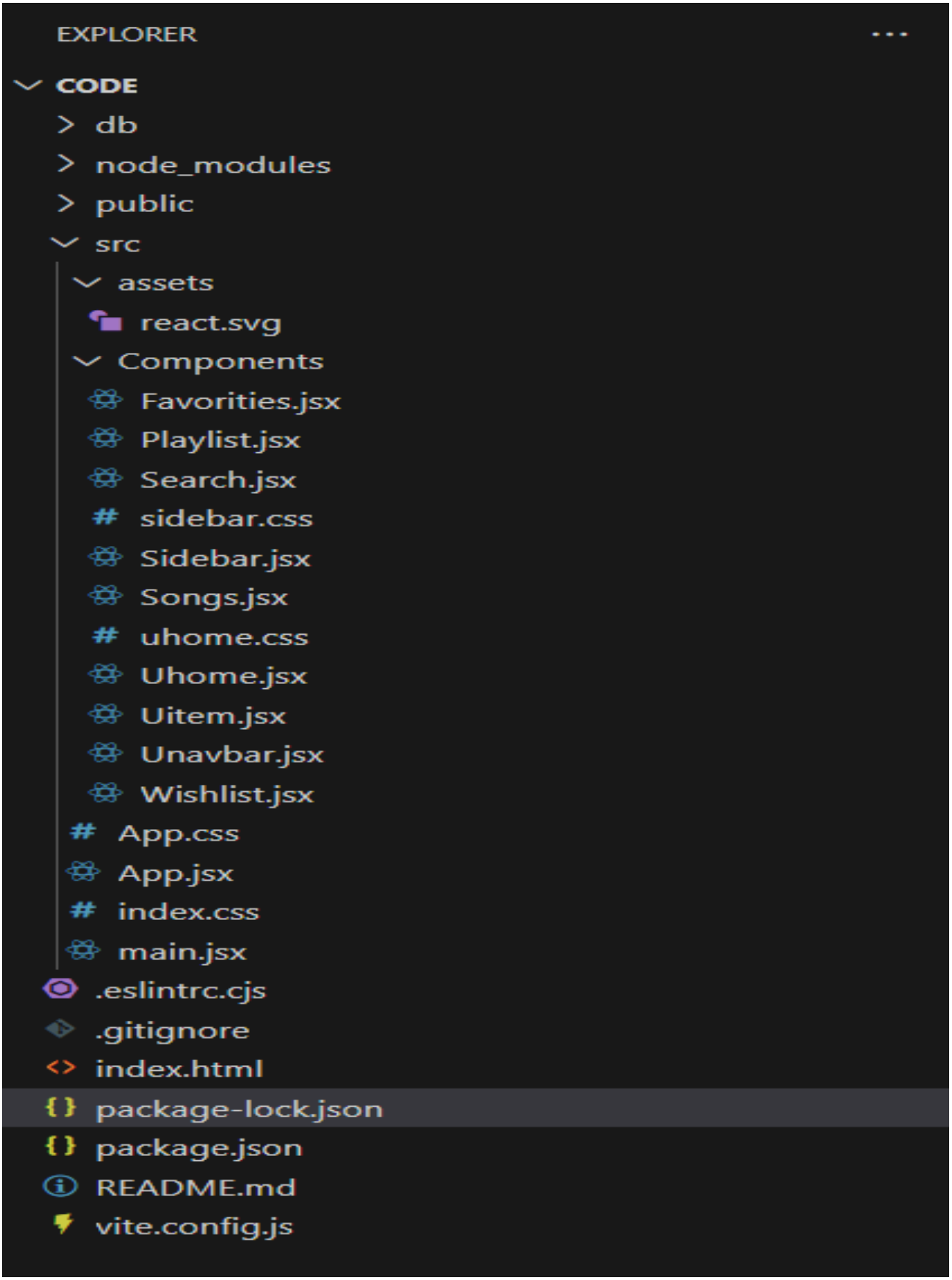
Component Structure:

When building a Rhythmic Tunes **frontend** with React.js, the application should be divided into **modular and reusable components**. Each component should have a **clear responsibility** and interact with other components to create a seamless user experience. Below is a detailed outline

the **major components** and their **interactions**:

```
RhythmicTunes/
|— public/                  # Static assets (icons, images, etc.)
|— src/                     # Main source code
|   |— components/          # Reusable UI components
|   |   |— Header.js        # Top navigation bar
|   |   |— Sidebar.js       # Sidebar navigation
|   |   |— AudioPlayer.js   # Music playback controls
|   |   |— Search.js        # Search bar for filtering songs
|   |   |— SongCard.js      # UI component to display each song
|   |— pages/               # Main pages of the app
|   |   |— Home.js          # Home page displaying featured songs
|   |   |— Songs.js         # Displays all available songs
|   |   |— Playlist.js      # Allows users to manage playlists
|   |   |— Favorites.js     # Displays songs marked as favorites
```

```
|   |— services/          # API handling logic
|   |   |— api.js         # Handles API calls with Axios
|   |— assets/            # Static assets like images, fonts, etc.
|   |— styles/            # CSS and styling files
|   |— App.js             # Main application component with routing
|   |— index.js           # Entry point of the React app
|— db.json                # Mock database (JSON Server)
|— package.json           # Project dependencies
|— README.md              # Project documentation
```



State Management:

The state management for the RhythmicTunes Music Streaming Application primarily relies on React's state management techniques, including:

1. State Handling with useState Hook

useState is used to manage key application states such as:

items: List of all songs fetched from the backend.

wishlist: Stores the user's favorite songs.

playlist: Stores songs added by the user to their playlist.

currentlyPlaying: Tracks the currently playing song.

searchTerm: Stores the search input value.

2. Data Fetching with useEffect and Axios

useEffect fetches data when the component mounts:

Fetches songs, favorites, and playlists from the backend.Uses axios.get() for API calls to retrieve data from a JSON server running on http://localhost:3000.

3. Audio Playback Management

Controls song playback to ensure only one song plays at a time.

Uses event listeners to manage play/pause actions.

4. Managing User Interactions

Functions to handle adding/removing songs from wishlist and playlist:

addToWishlist(itemId), removeFromWishlist(itemId)

addToPlaylist(itemId), removeFromPlaylist(itemId)

Uses axios.post() and axios.delete() to update the backend.

5. Routing and Navigation (React Router)

Uses React Router to navigate between different sections:

/ → Displays the list of songs.

/favorities → Shows the user's favorite songs.

/playlist → Displays user-created playlists.

6. State Filtering (Search Functionality)

Filters the song list based on searchTerm by matching it against:

Song title

Singer

Genre

7. Local JSON Server as Backend

Runs using json-server --watch ./db/db.json

Stores song data, favorites, and playlists in a local database.

Possible Improvements

Context API / Redux: If the app scales, a centralized state management system like Redux or React Context API can be introduced.

Global State for Audio Playback: Instead of managing audio playback state per component, a global state can ensure seamless playback control.

Routing:

1. Importing React Router

In the App.js file (or equivalent), the application imports the necessary components from react-router-dom:

javascript Copy Edit

```javascript
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Sidebar from './components/Sidebar';
import Songs from './components/Songs';
import Favorites from './components/Favorites';
import Playlist from './components/Playlist';
```

2. Setting Up the Router

The main routing structure is wrapped inside the <BrowserRouter> component, which enables navigation without page reloads.

```javascript
function App() {
  return (
    <BrowserRouter>
      <div className="app-container">
        <Sidebar />
        <div className="content">
          <Routes>
            <Route path="/" element={<Songs />} />
            <Route path="/favorites" element={<Favorites />} />
            <Route path="/playlist" element={<Playlist />} />
          </Routes>
        </div>
      </div>
    </BrowserRouter>
  );
}

export default App;
```

3. Navigation with Links

The **Sidebar** component (or any navigation menu) likely contains **React Router's <Link> or <NavLink>** for navigation.

```javascript
import { Link } from 'react-router-dom';

function Sidebar() {
  return (
    <nav>
      <ul>
        <li><Link to="/">Songs</Link></li>
        <li><Link to="/favorites">Favorites</Link></li>
        <li><Link to="/playlist">Playlist</Link></li>
      </ul>
    </nav>
  );
}

export default Sidebar;
```

4. Handling Dynamic Routes (if applicable)

If the app allows viewing specific songs, albums, or artists dynamically, it might use **route parameters** like:

```javascript
<Route path="/song/:id" element={<SongDetails />} />
```

```javascript
import { useParams } from 'react-router-dom';

function SongDetails() {
  let { id } = useParams();
  return <div>Now playing song {id}</div>;
}
```

5. Redirecting (Optional)

If an invalid route is entered, a **404 Not Found** page can be handled using a wildcard route:

javascript

CopyEdit

This ensures that any undefined route leads to a NotFound component.

```javascript
<Route path="*" element={<NotFound />} />
```

Setup Instructions:

PRE-REQUISITES:- **Installation**:

Here are the key prerequisites for developing a frontend application using React.js:

 Node.js and npm:

Node.js is a powerful JavaScript runtime environment that allows you to run JavaScript code on the local environment. It provides a scalable and efficient platform for building network applications.

Install Node.js and npm on your development machine, as they are required to run JavaScript on the server-side.

- Download: https://nodejs.org/en/download/
- Installation instructions: https://nodejs.org/en/download/package-manager/

 React.js:

React.js is a popular JavaScript library for building user interfaces. It enables developers to create interactive and reusable UI components, making it easier to build dynamic and responsive web applications.

Install React.js, a JavaScript library for building user interfaces.

- Create a new React app:

npm create vite@latest

Enter and then type project-name and select preferred frameworks and then enter

- Navigate to the project directory:

cd project-name

npm install

- Running the React App:

With the React app created, you can now start the development server and see your React application in action.

- Start the development server:

npm run dev

This command launches the development server, and you can access your React app at http://localhost:5173 in your web browser.

 HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

 Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

• Git: Download and installation instructions can be found at: https://git-scm.com/downloads

 Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

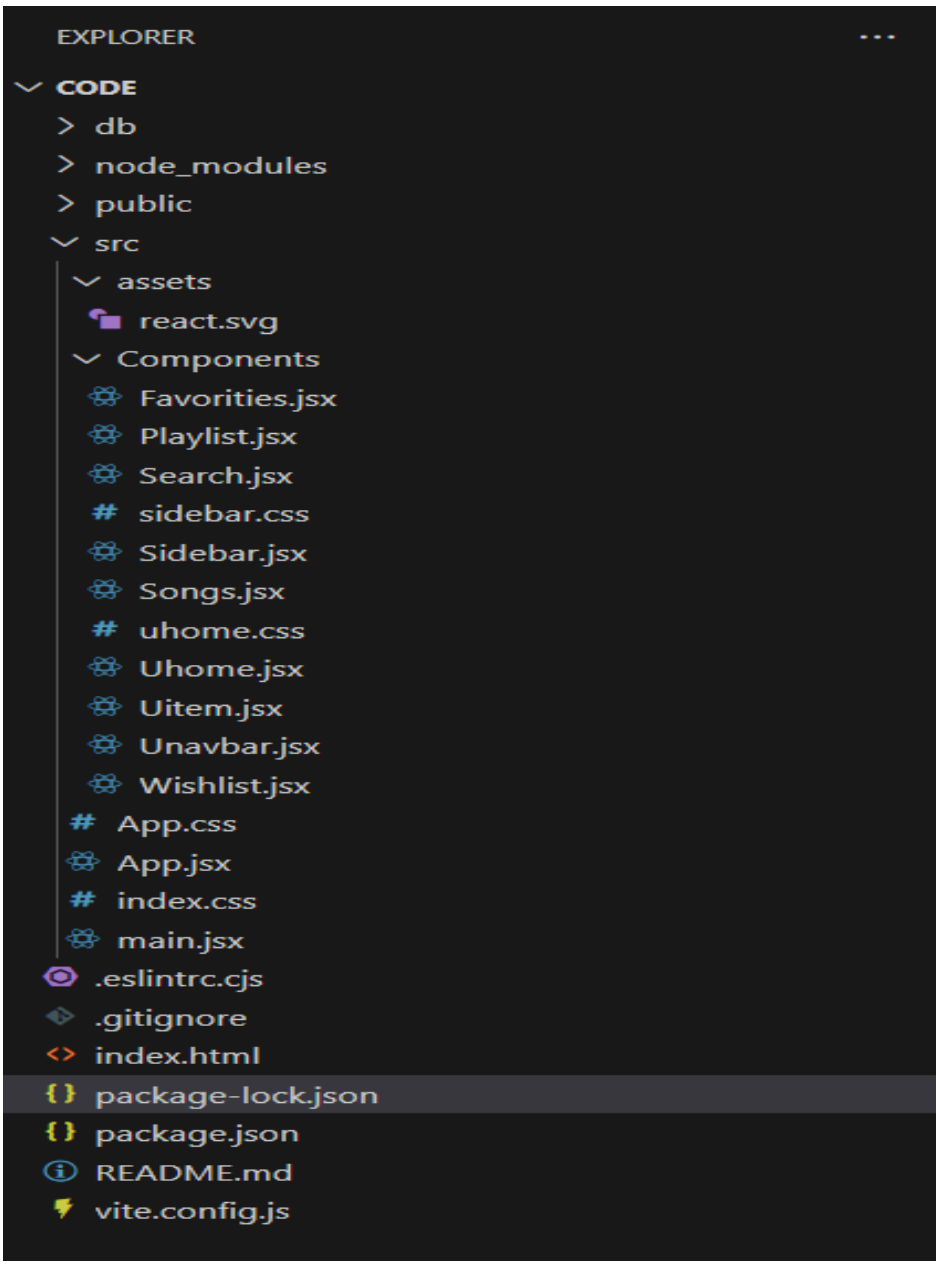
• Visual Studio Code: Download from https://code.visualstudio.com/download

• Sublime Text: Download from https://www.sublimetext.com/download

•WebStorm:Download from https://www.jetbrains.com/webstorm/download

Folder Structure

The React application follows a well-organized structure to ensure maintainability and scalability. Below is the typical folder layout:



Utilities & Custom Hooks

1. useFetch.js (Fetching API Data)

A custom React Hook to handle API calls efficiently using useState and useEffect.

```
import { useState, useEffect } from "react";
import axios from "axios";

const useFetch = (url) => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    axios.get(url)
      .then((response) => {
        setData(response.data);
        setLoading(false);
      })
      .catch((err) => {
        setError(err);
        setLoading(false);
      });
  }, [url]);

  return { data, loading, error };
};

export default useFetch;
```

USAGE:

```
const { data: songs, loading, error } = useFetch("http://localhost:3000/items");
```

2. useAudioPlayer.js (Managing Audio Playback)

A helper Hook to ensure only one song plays at a time.

```js
import { useState } from "react";

const useAudioPlayer = () => {
  const [currentAudio, setCurrentAudio] = useState(null);

  const playAudio = (audioElement) => {
    if (currentAudio && currentAudio !== audioElement) {
      currentAudio.pause();
    }
    setCurrentAudio(audioElement);
    audioElement.play();
  };

  return { playAudio };
};

export default useAudioPlayer;
```

```js
const { playAudio } = useAudioPlayer();

<button onClick={() => playAudio(document.getElementById(`audio-${song.id}`))}>
  Play
</button>
```

3. api.js (Centralized API Calls)

Handles all API interactions using Axios.

```js
import axios from "axios";

const API_URL = "http://localhost:3000";

export const getSongs = () => axios.get(`${API_URL}/items`);
export const getFavorites = () => axios.get(`${API_URL}/favorities`);
export const addToFavorites = (song) => axios.post(`${API_URL}/favorities`, song);
export const removeFromFavorites = (id) => axios.delete(`${API_URL}/favorities/${id}`);
```

```js
import { getSongs } from "../services/api";

useEffect(() => {
  getSongs().then((res) => setSongs(res.data));
}, []);
```

Running the Application

1. **Running the Application**

```sh
cd project-name
```

2. **Install dependencies:**

```sh
npm install
```

3. **Start the frontend server:**

   If using Vite.js:

```sh
npm run dev
```

This will start the development server, and you can access it at http://localhost:5173.

If using Create React App:

```sh
npm start
```

4. Start the JSON server (if required for mock data):

```sh
json-server --watch ./db/db.json
```

Once these commands are executed, you should be able to launch and interact with Rythimic Tunes on your local machine. Let me know if you need further clarification!

Component Documentation

Component Documentation for RhythmicTunes (React Music Streaming Application)

Key Components

Below are the major components of the RhythmicTunes application, their purpose, and the props they receive:

1. App Component
    - Purpose: Root component that sets up routing and layout.

- Props: None explicitly, but manages navigation via react-router-dom.

2. Sidebar Component
   - Purpose: Navigation menu for different sections like songs, favorites, and playlists.
   - Props: None explicitly.
3. Songs Component
   - Purpose: Displays a list of songs with details such as title, artist, genre, and release date.
   - Props:
     - items (Array): List of songs fetched from the backend.
     - wishlist (Array): List of favorite songs.
     - playlist (Array): List of songs added to the user's playlist.
     - searchTerm (String): Current search term for filtering songs.
4. Favorites Component
   - Purpose: Displays the user's favorite songs.
   - Props:
     - wishlist (Array): List of favorite songs.
     - removeFromWishlist (Function): Removes a song from favorites.
5. Playlist Component

- Purpose: Displays and manages user-created playlists.
- Props:
  - playlist (Array): List of songs added to the user's playlist.
  - removeFromPlaylist (Function): Removes a song from the playlist.

6. SearchBar Component
   - Purpose: Allows users to search for songs by title, artist, or genre.
   - Props:
     - searchTerm (String): Current search term.
     - setSearchTerm (Function): Updates the search term.
7. SongCard Component
   - Purpose: Displays individual song details, audio player, and buttons for adding/removing from playlists and favorites.
   - Props:
     - item (Object): Song details (title, artist, genre, URL).
     - isItemInWishlist (Function): Checks if a song is in the wishlist.
     - addToWishlist (Function): Adds a song to the wishlist.

- removeFromWishlist (Function): Removes a song from the wishlist.
- isItemInPlaylist (Function): Checks if a song is in the playlist.
- addToPlaylist (Function): Adds a song to the playlist.
- removeFromPlaylist (Function): Removes a song from the playlist.

Reusable Components & Configurations

1. Button Component
    - Used for actions like "Add to Favorites", "Remove from Playlist", etc.
    - Props:
        - onClick (Function): Handles button click event.
        - label (String): Text displayed on the button.
2. AudioPlayer Component
    - Manages audio playback within song cards.
    - Props:
        - src (String): Audio file URL.
        - onPlay (Function): Handles play events.
3. Modal Component
    - Used for displaying pop-ups (e.g., song details).
    - Props:

        - isOpen (Boolean): Controls modal visibility.
        - onClose (Function): Handles modal close action.

4. Input Component
    - Used in search bars and forms.
    - Props:
        - value (String): Input field value.
        - onChange (Function): Handles input changes.

State Management

State Management in Applications

State management is a crucial aspect of application development, ensuring that data flows efficiently between

components and remains consistent. It helps maintain the application's behavior, user interactions, and UI updates.

Global State Management

What is Global State?

Global state refers to the data that is shared across multiple components in an application. It is stored in a central location and can be accessed by different parts of the application, ensuring consistency.

How Global State Flows Across an Application

1. Centralized Storage: The global state is usually managed in a centralized store (e.g., Redux store in React, Vuex in Vue, or Context API in React).
2. State Providers: A provider component makes the global state accessible to all child components.
3. Actions and Dispatchers: Components can update the global state using predefined actions or events.
4. Selectors and Subscribers: Components subscribe to specific parts of the state and re-render when the state changes.

Examples of Global State Management

- Redux (React): Uses a centralized store, reducers, and actions to manage global state.
- Context API (React): Provides a lightweight way to share state across components.

- Vuex (Vue.js): Centralized state management system for Vue applications.
- MobX (React/Vue): A reactive state management library.

Local State Management

What is Local State?

Local state refers to data that is confined within a single component. It is used for temporary states such as UI interactions, form inputs, and toggles.

Handling Local State Within Components

1. Use of Component State:
    - In React, the useState hook manages local state.
    - In Vue, data() manages local state.
2. Updating Local State:
    - In React:

jsx

CopyEdit

const [count, setCount] = useState(0);

const increment = () => setCount(count + 1);

    - In Vue:

vue

CopyEdit

<script>

```
export default {
  data() {
    return { count: 0 };
  },
  methods: {
    increment() {
      this.count++;
    }
  }
};
</script>
```

3. Lifting State Up: If multiple components need the same data, the local state can be moved to a common parent component and passed as props.
4. Effect Hooks for Side Effects: In React, useEffect can manage side effects related to state changes.

Examples of Local State Usage

- UI components toggling between states (e.g., show/hide modal).
- Managing form input fields and validation.
- Handling user interactions within a single component.

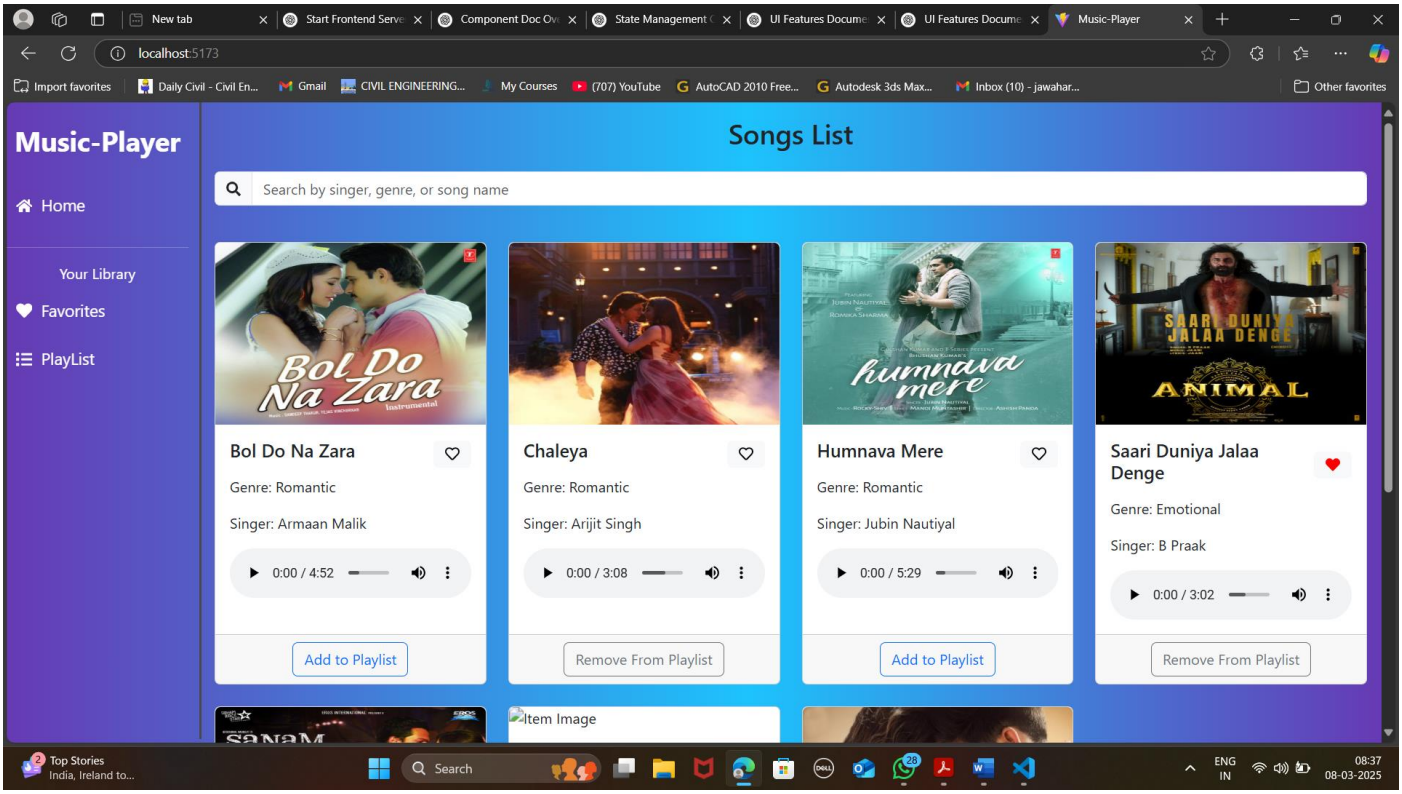
Feature	Global State	Local State
Scope	Available across the application	Limited to a single component
Storage	Stored in a central store	Stored in the component itself
Use Case	Shared data (e.g., user authentication, theme settings)	Temporary state (e.g., form input, toggles)
Management Tools	Redux, Vuex, Context API, MobX	useState (React), data() (Vue)

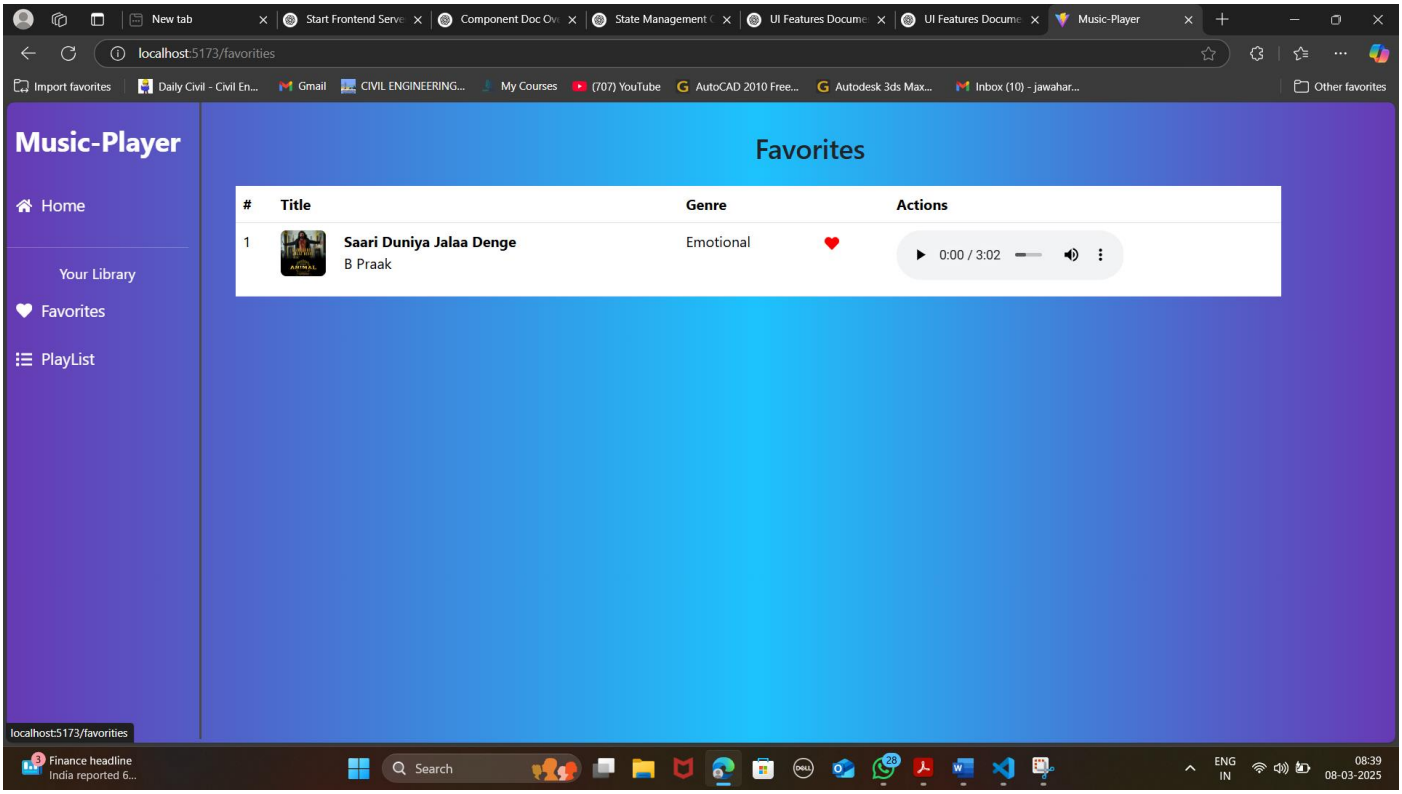
When to Use Global vs. Local State

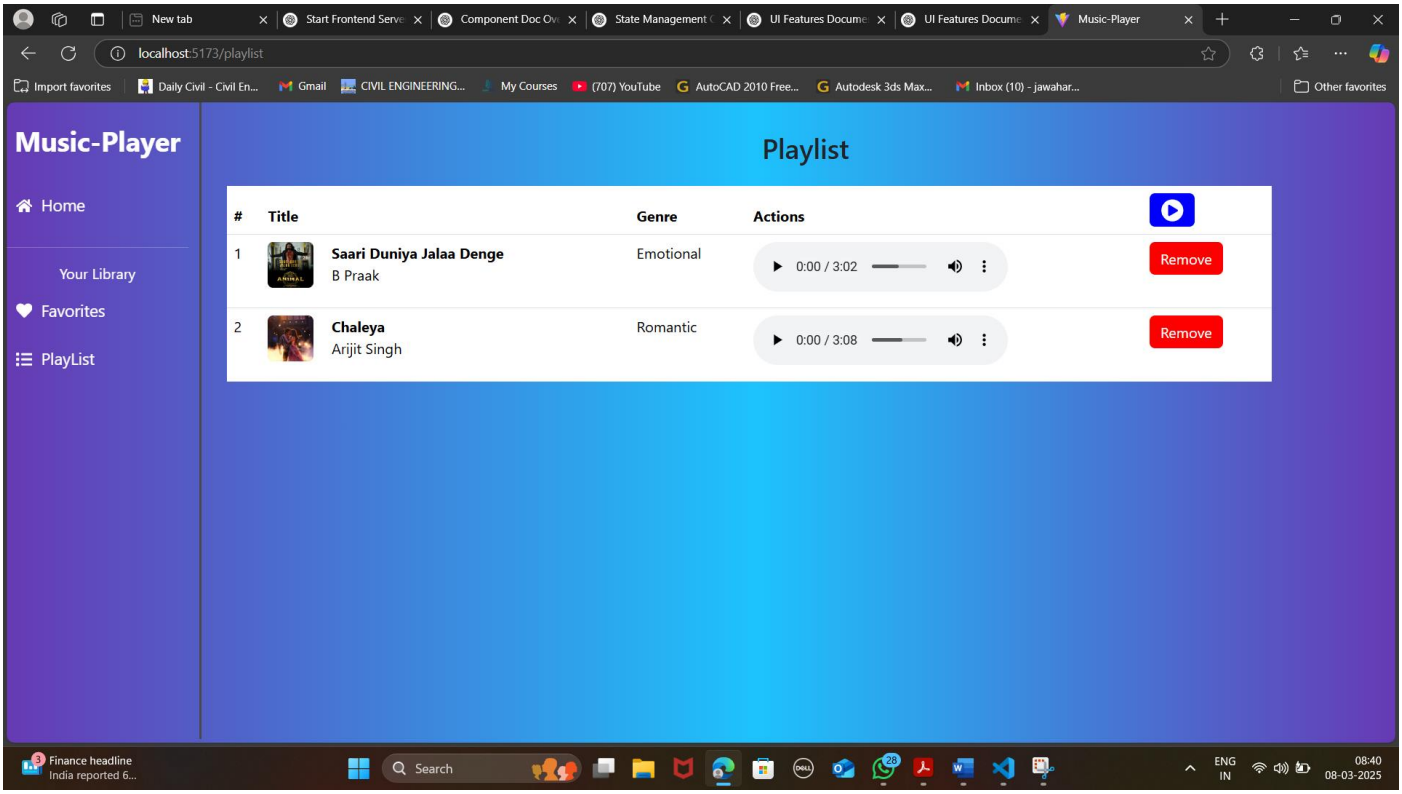
- Use global state when data needs to be accessed by multiple components, such as authentication, user preferences, or application-wide notifications.
- Use local state when managing temporary data within a component, like input values, UI visibility, or dropdown selection.

Understanding and properly managing both global and local states improves the efficiency, maintainability, and scalability of an application.

User Interface







Styling

Styling Documentation for RhythmicTunes Music Streaming Application

CSS Frameworks/Libraries

The RhythmicTunes frontend is built using **React.js** and leverages multiple CSS frameworks and libraries for styling and UI enhancement:

- **Bootstrap/Tailwind CSS**: Used for pre-styled UI components, grid layouts, and responsive design.
- **React Bootstrap**: Provides ready-to-use Bootstrap components with seamless integration in React.
    - Imported using:

```js
import 'bootstrap/dist/css/bootstrap.min.css';
```

- **React Icons**: Used for adding scalable vector icons to enhance UI elements like buttons and navigation.

Theming and Custom Design System

- **Custom CSS**: The application includes a dedicated **App.css** file to implement custom styling beyond the frameworks.
- **Component-Based Styling**: Each component can have its own CSS file, ensuring modular and reusable styles.
- **Dark/Light Mode (Potential)**: If theming is required, a state-based toggle system can be implemented using CSS variables or React Context API.
- **Responsive Design**: Ensured using Bootstrap/Tailwind utility classes and media queries.

Testing

expected. This includes:

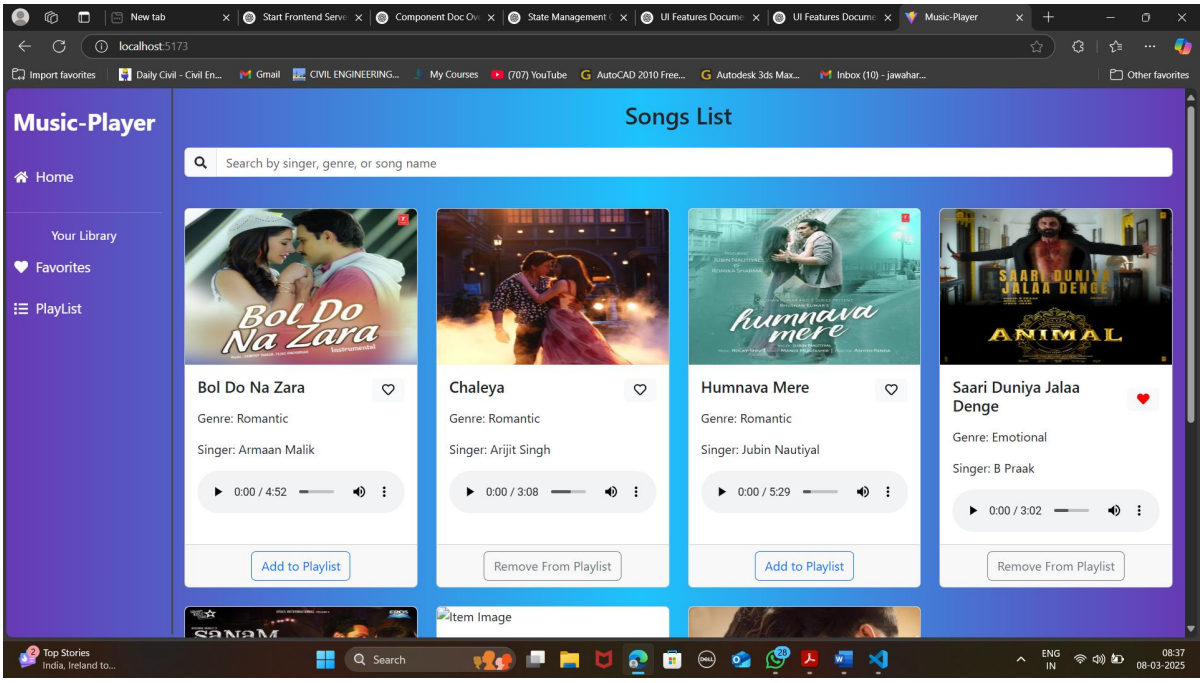
- **Unit Testing**: Individual React components and utility functions are tested in isolation using **Jest** and **React Testing Library**.
- **Integration Testing**: Verifying interactions between components and API endpoints (e.g., song fetching, playlist management) using **React Testing Library** and **Mock Service Worker (MSW)**.
- **End-to-End (E2E) Testing**: Ensuring complete user workflows, such as searching for a song, adding to a playlist, and playing a track, using **Cypress**.

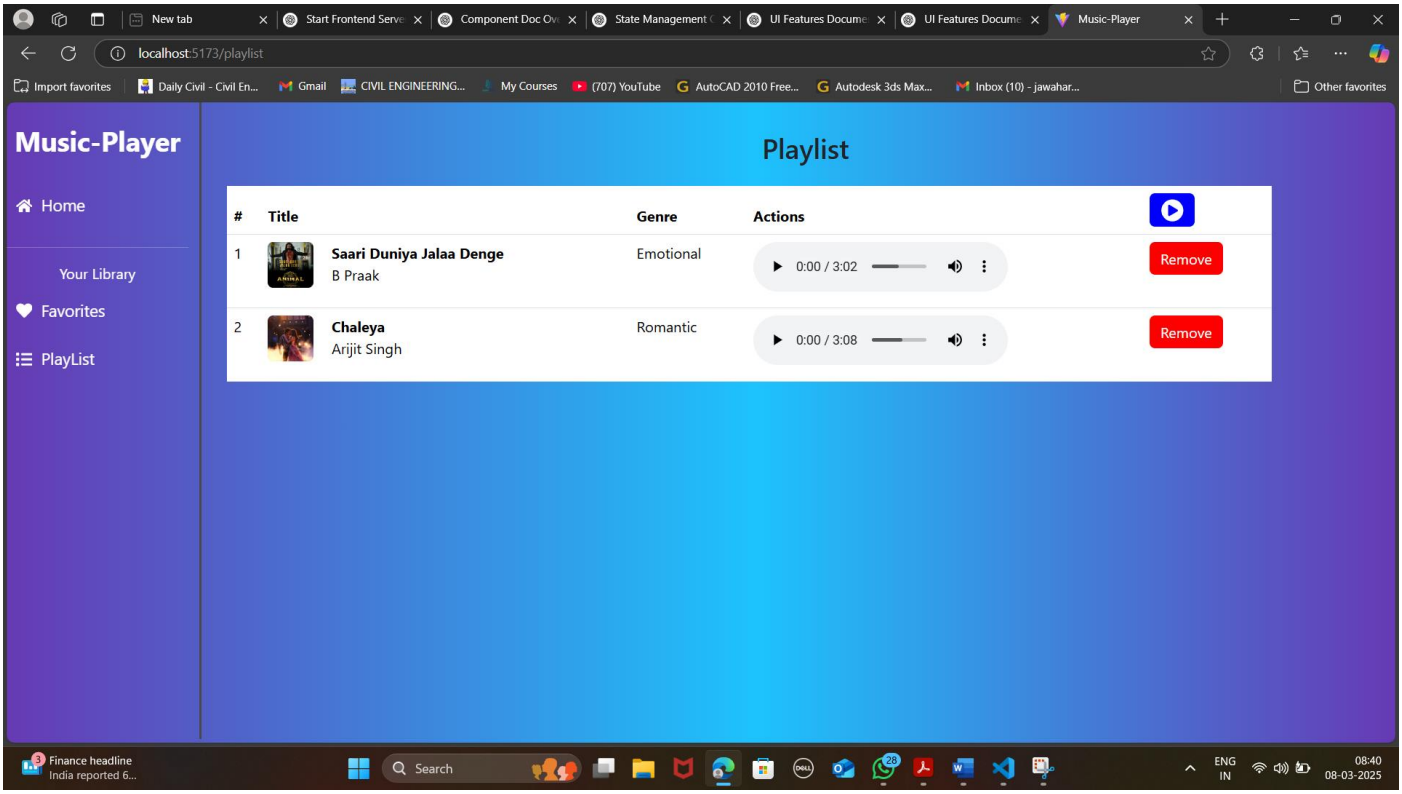
2. Code Coverage

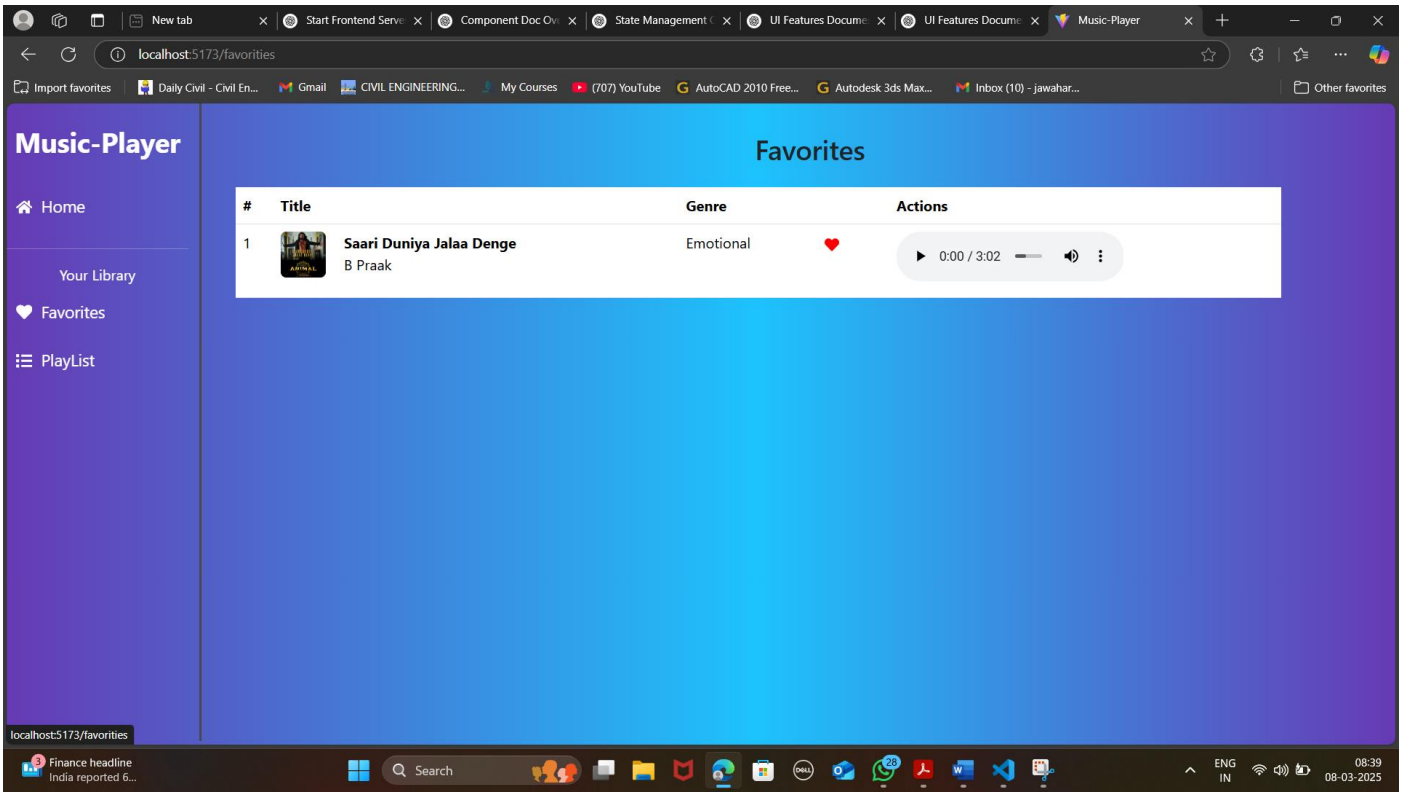
To measure and ensure adequate test coverage:

- **Jest with Istanbul (nyc)**: Provides insights into function, statement, and branch coverage.
- **React Testing Library**: Ensures real-world user interactions are covered.
- **Cypress Code Coverage Plugin**: Tracks untested parts of E2E workflows.

Screenshots or Demo







Demo video

https://drive.google.com/file/d/1zZuq62lyYNV_k5uu0SFjoWa35UgQ4LA9/view?usp=driv

The document does not contain a dedicated section for "Known Issues," but based on the provided content, potential issues users or developers should be aware of include:

1. **Audio Playback Management**
    - Only one audio element can play at a time, which may cause unexpected behavior if users attempt to play multiple tracks simultaneously.
2. **Data Fetching & API Calls**
    - The app relies on a JSON server (http://localhost:3000), meaning it requires the server to be running for proper functionality.
    - Errors in fetching, adding, or removing items from the wishlist or playlist could occur due to API request failures.
3. **Search Functionality Limitations**
    - The search function only matches lowercase versions of song titles, singers, or genres, which may lead to inconsistent results.
4. **Frontend UI Issues**
    - Card layout depends on Bootstrap grid settings, so styling inconsistencies might arise on different screen sizes.
5. **Dependency on External Libraries**
    - The application uses React Router Dom, React Icons, Bootstrap/Tailwind CSS, and Axios, meaning updates or

deprecations in these libraries could impact functionality.

6. **Local Development Environment Issues**
    - Requires running both npm run dev and json-server --watch ./db/db.json in separate terminals, which may cause issues for developers unfamiliar with this setup.

Future Enhancements

1. **AI-Powered Recommendations** – Implement a machine learning-based recommendation system to suggest personalized playlists and new music based on user preferences and listening history.
2. **Collaborative Playlists** – Allow multiple users to contribute to shared playlists, making it perfect for group activities or parties.
3. **Social Media Integration** – Enable users to share their favorite songs or playlists on platforms like Instagram, Twitter, and Facebook.
4. **Lyrics Display** – Integrate real-time lyrics display while playing songs.
5. **Podcast & Audiobook Support** – Expand content offerings beyond music to include podcasts and audiobooks.
6. **Live Streaming & Radio Stations** – Allow users to tune into live performances, concerts, or curated radio stations.

UI/UX Enhancements:

1. **Dark & Light Mode** – Provide users with theme-switching options for a better