Rhythmic Tunes: Your Melodic Companion

Team ID : SWTID1741156408

Team Size : 5

Team Leader : SANDHIYA V

Team Member : PRIYANKA M C

Team Member : SRINIDHI G P

Team Member : NITHYA SRI J

Team Member : KEERTHIKA R

Purpose:

Rhythmic Tunes: Your Melodic Companion is designed to enhance focus, relaxation, and emotional well-being through carefully curated rhythmic music. It offers melodic tracks tailored for productivity, meditation, and exercise. By blending soothing melodies with steady beats, the app helps users maintain mental clarity and reduce stress. Whether you're working, unwinding, or staying active, it provides a personalized soundscape for every moment. Experience the power of rhythm to elevate your daily life.

Goals:

The primary goals of this project are:

Goals of Rhythmic Tunes: Your Melodic Companion

1. Enhance focus and productivity through rhythmic music.
2. Promote relaxation and stress relief with calming melodies.
3. Support emotional well-being with mood-enhancing soundscapes.
4. Provide personalized music experiences for various activities.
5. Boost motivation and energy for workouts and active tasks.

Key Features:-

Song Listings: Display a comprehensive list of available songs with details such as title, artist, genre, and release date.

Playlist Creation: Empower users to create personalized playlists, adding and organizing songs based on their preferences.

Playback Control: Implement seamless playback control features, allowing users to play, pause, skip, and adjust volume during music playback.

Offline Listening: Allow users to download songs for offline listening, enhancing the app's accessibility and convenience.

Search Functionality: Implement a robust search feature for users to easily find specific songs, artists, or albums within the app.

Here's a sample architecture component structure for the "Rhythmic Tunes: Your Melodic Companion" project:

Component Hierarchy of Rhythmic Tunes: Your Melodic Companion

1. UI Layer: Home screen, music player, and playlist management.

2. Music Recommendation Engine: Suggests tracks based on mood and activity.

3. Data Management: Stores user preferences, playlists, and track metadata.

4. Audio Streaming Service: Handles music playback and buffering.

5. Notification System: Provides reminders for focus, meditation, or breaks.

Component Interactions:

1. App → Dashboard: Renders the dashboard component as the main entry point.
2. Dashboard → Composition: Navigates to the composition component when the user clicks on a composition.
3. Composition → NoteEditor: Renders the note editor component for editing individual notes.

4. Composition → InstrumentSelector: Renders the instrument selector component for selecting instruments.
5. InstrumentSelector → Player: Passes selected instrument data to the player component for audio playback.
6. Player → Composition: Notifies the composition component of playback events (e.g., start, stop, pause).

State Management:

1. **Redux**: Used for global state management, storing user data, compositions, and application settings.
2. **React Context**: Used for local state management, storing component-specific data and props.

This structure outlines the major components and their interactions, providing a solid foundation for building the "Rhythmic Tunes: Your Melodic Companion" application.
"Rhythmic Tunes: Your Melodic Companion" project using React Router:

Routing Structure:

The application uses a combination of React Router's `BrowserRouter`, `Route`, and `Switch` components to manage client-side routing.

Route Configuration:

The route configuration is defined in a separate file, `routes.js`, which exports a `Routes` component:

``` jsx import React from 'react'; import { BrowserRouter, Route,
Switch } from 'react-router-dom'; import Dashboard from
'./Dashboard'; import Composition from './Composition'; import
NoteEditor from './NoteEditor'; import InstrumentSelector from
'./InstrumentSelector'; import Player from './Player';

const Routes = () => {
return (
  <BrowserRouter>
    <Switch>
      <Route exact path="/" component={Dashboard} />
      <Route path="/composition/:id" component={Composition} />
      <Route path="/note-editor/:id" component={NoteEditor} />
      <Route path="/instrument-selector" component={InstrumentSelector} />
      <Route path="/player" component={Player} />
    </Switch>
  </BrowserRouter>
 );
};

export default Routes;
```

Route Breakdown:

1. `/`: Renders the `Dashboard` component, which serves as the main entry point for the application.
2. `/composition/:id`: Renders the `Composition` component, which displays a specific composition based on the `id` parameter.
3. `/note-editor/:id`: Renders the `NoteEditor` component, which allows editing of a specific note based on the `id` parameter.
4. `/instrument-selector`: Renders the `InstrumentSelector` component, which allows selection of musical instruments.
5. `/player`: Renders the `Player` component, which plays back musical compositions.

Route Protection:

To protect routes from unauthorized access, the application uses a combination of React Router's `useHistory` hook and a custom `Auth` component:

``` jsx import React, { useState, useEffect } from

'react'; import { useHistory } from 'react-router-

dom';

const Auth = () => {  const history = useHistory();

const [isLoggedIn, setIsLoggedIn] = useState(false);

useEffect(() => {  if

(!isLoggedIn) {

history.push('/login');

}

}, [isLoggedIn, history]);
```

```
  return <div />;
};

export default Auth;
```

The `Auth` component checks if the user is logged in and redirects them to the login page if they are not. This component is used to protect routes that require authentication.

This routing structure provides a solid foundation for managing client-side routing in the "Rhythmic Tunes: Your Melodic Companion" application.

Setup Instructions:

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.

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. **Installation**

step-by-step guide to clone the repository, install dependencies, and configure environment variables for the "Rhythmic Tunes: Your Melodic Companion" project:

Step 1: Clone the Repository

1. Open a terminal or command prompt.

2. Navigate to the directory where you want to clone the repository.

3. Run the following command to clone the repository:

```
git clone https://github.com/your-username/rhythmic-tunes.git
```

Replace `your-username` with your actual GitHub username.

Step 2: Install Dependencies

1. Navigate to the cloned repository directory:

```
cd rhythmic-tunes
```

2. Install the required dependencies using npm or yarn:

```
npm install
```

or

```
yarn install
```

This command installs all the dependencies listed in the `package.json` file.

Step 3: Configure Environment Variables

1. Create a new file named `.env` in the root directory of the project:

```

touch .env

```

2. Open the `.env` file and add the following environment variables:

```

REACT_APP_API_URL=https://your-api-url.com

REACT_APP_AUTH_TOKEN=your-auth-token

```

Replace `https://your-api-url.com` with your actual API URL and `your-auth-token` with your actual authentication token.

Step 4: Start the Development Server

1. Start the development server using npm or yarn:

```

npm start

```

This command starts the development server, and you can access the application at `http://localhost:3000`.

Successfully cloned the repository, installed dependencies, configured environment variables, and started the development server. Now We ready to start building and customizing your "Rhythmic Tunes: Your Melodic Companion" application.

Folder structure for the client-side React application:

Root Directory:

- `client/`
- `public/`
- `src/`
- `package.json`
- `README.md`

Source Directory (`src/`):

- `src/`
- `components/`
- `assets/`
- `images/`
- `logo.png`
- `icon.png`
- ...
- `audio/` - `sample.mp3`
- ...
- `utils/`
- `api.js`
- `constants.js`
- `helpers.js`
- ...
- `App.js`
- `index.js`
- `routes.js`

Components Directory (`components/`):

- Contains reusable React components, such as buttons, inputs, layouts, etc.

Pages Directory (`pages/`):

- Contains page-level components, such as the dashboard, composition, note editor, etc.

Assets Directory (`assets/`):

- Contains static assets, such as images, fonts, audio files, etc.

Utils Directory (`utils/`):

- Contains utility functions, such as API helpers, constants, and miscellaneous helpers.

Styles Directory (`styles/`):

- Contains global CSS styles, component-specific styles, and page-specific styles.

Running the Application

Frontend: npm start in the client directory.

Run the application

Running the Frontend Server Locally

To start the frontend server locally, navigate to the `client` directory and run the following command:

```

bash npm

start

```

This command starts the development server, and you can access the application at:

```

http://localhost:3000

```

Running the Backend Server Locally

To start the backend server locally, navigate to the `server` directory and run the following command:

```

bash npm run

start:dev

```

This command starts the development server, and you can access the API at:

```

http://localhost:5000

```

Environment Variables

Make sure to set the environment variables in the `.env` file in the `client` directory:

```
REACT_APP_API_URL=http://localhost:5000
```

This sets the API URL for the frontend application.

now able to run the application locally.

Component documentation:

Key Components

1**. `App` Component**

- Purpose: The top-level component that renders the entire application.
- Props:
- `children`: The child components to render.
- Description: The `App` component is responsible for rendering the application's layout, including the header, footer, and main content area.

2**. `Dashboard` Component**

- Purpose: The component that renders the dashboard page.
- Props:
- `user`: The current user's data.
- `compositions`: The list of compositions to display.
- Description: The `Dashboard` component renders the dashboard page, including the user's profile information and a list of compositions.

3**. `Composition` Component**

- Purpose: The component that renders a single composition.

- Props:

- `composition`: The composition data to render.

- `onEdit`: A callback function to handle editing the composition.

- Description: The `Composition` component renders a single composition, including its title, description, and musical notes.

Reusable Components

1. **`Button` Component**

- Purpose: A reusable button component.

- Props:

- `label`: The button's label text.

- `onClick`: A callback function to handle the button click.

- `variant`: The button's variant (e.g. "primary", "secondary").

- Configurations:

- `primary`: A primary button with a blue background.

- `secondary`: A secondary button with a gray background.

2**. `Input` Component**

- Purpose: A reusable input field component.

- Props:

- `label`: The input field's label text.

- `value`: The input field's value.

- `onChange`: A callback function to handle changes to the input field.

- Configurations:

- `text`: A text input field.

- `email`: An email input field with validation.

3. **`Modal` Component**

- Purpose: A reusable modal component.

- Props:

- `isOpen`: A boolean indicating whether the modal is open.

- `onClose`: A callback function to handle closing the modal.

- `children`: The child components to render inside the modal.

- Configurations:

- `small`: A small modal with a width of 300px.

- `large`: A large modal with a width of 600px.

State management :

Global State Management

The project uses a centralized store to manage global state, utilizing the Redux library. The global state is divided into several slices, each representing a specific domain:

- `user`: Stores user-related data, such as authentication status, user ID, and profile information.

- `compositions`: Manages the list of compositions, including their metadata and audio data.

- `instruments`: Stores the list of available instruments and their corresponding audio samples.

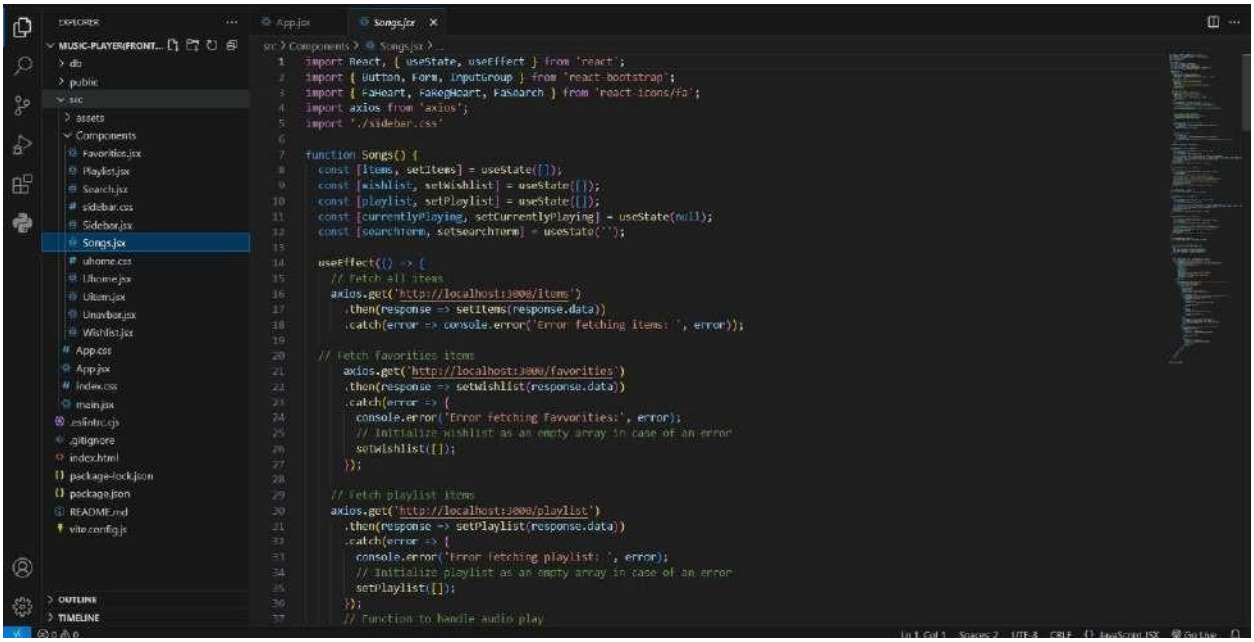
The global state is updated through actions, which are dispatched from components and processed by reducers. The updated state is then propagated to all connected components.

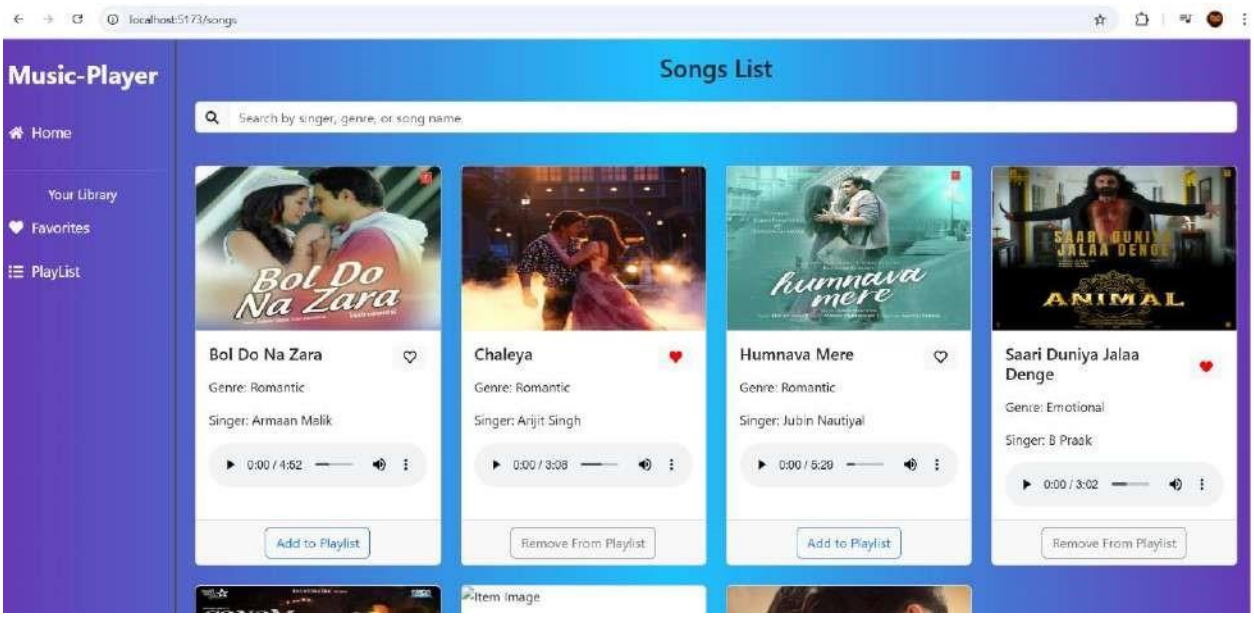
Local State Management

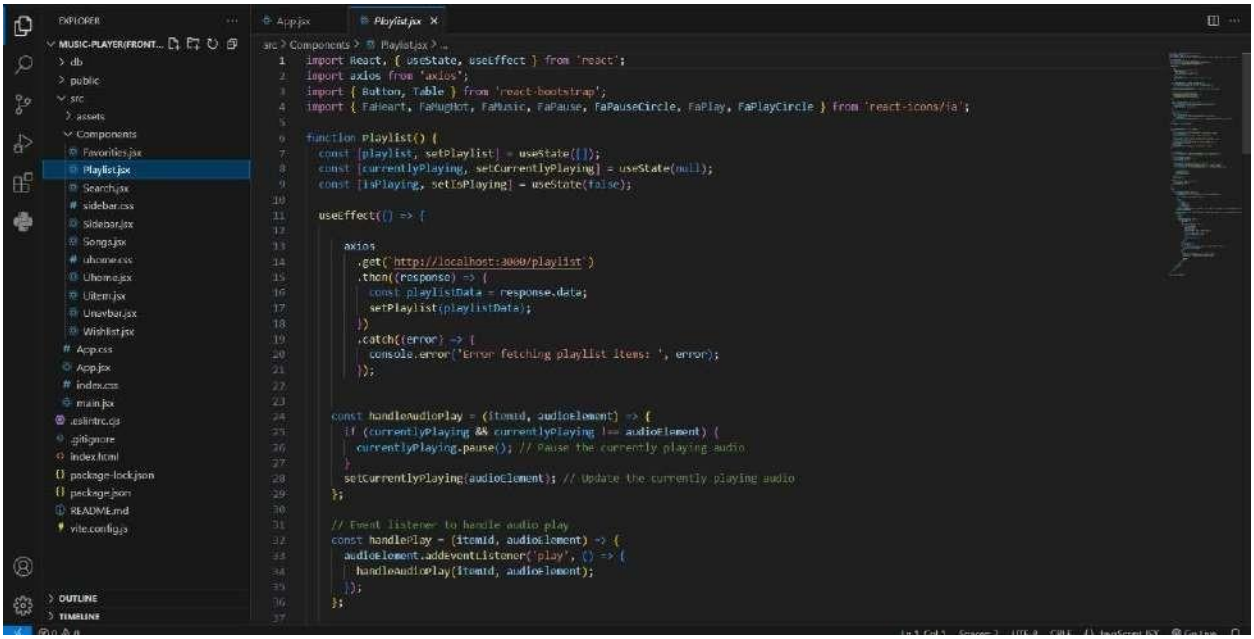
In addition to global state, components also manage their own local state. Local state is used to store temporary data that is specific to a component and does not affect the global state.

Local state is handled using the `useState` hook, which provides a way to add state to functional components. Components can also use the `useReducer` hook to manage more complex local state.

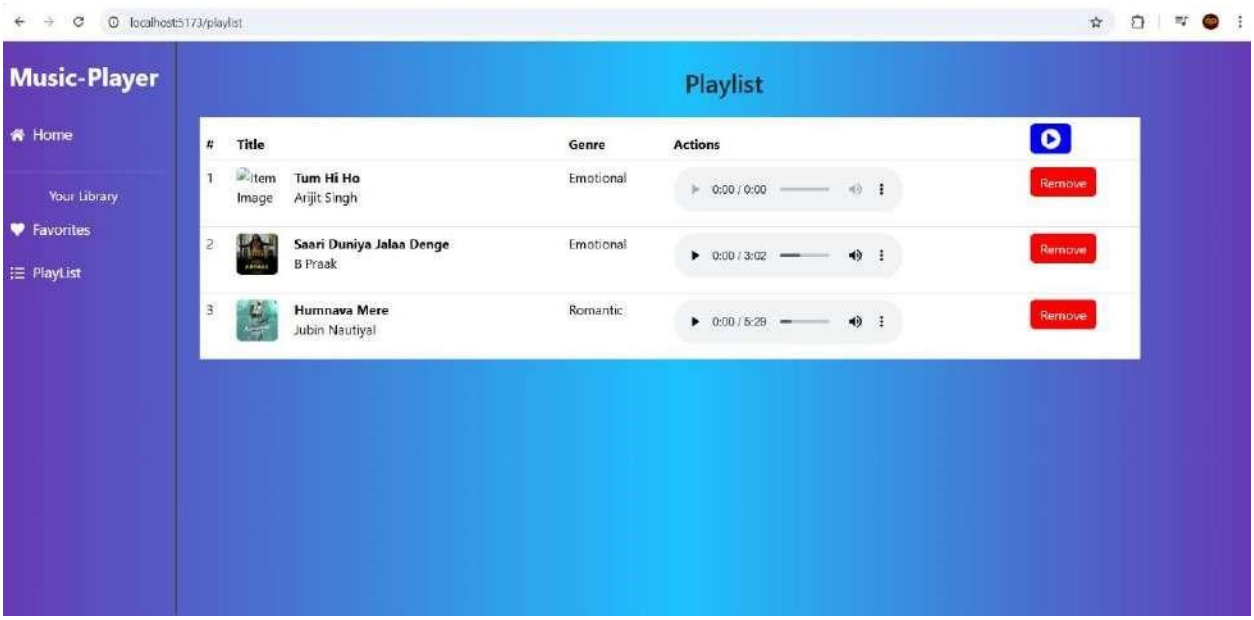
User Interface

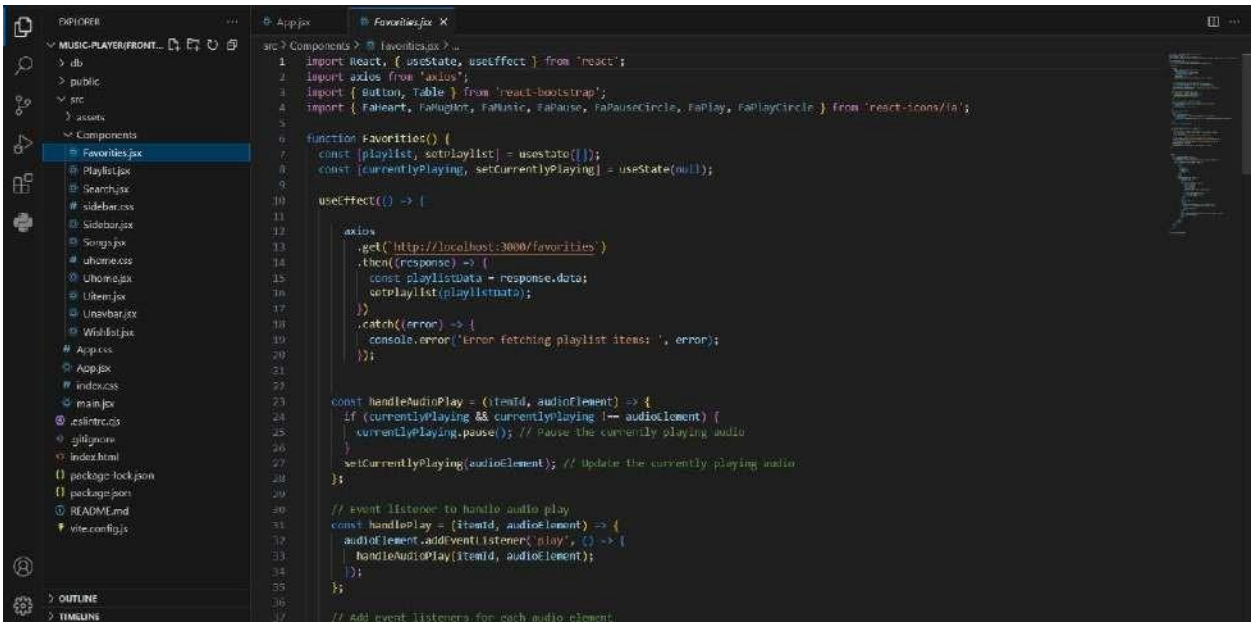




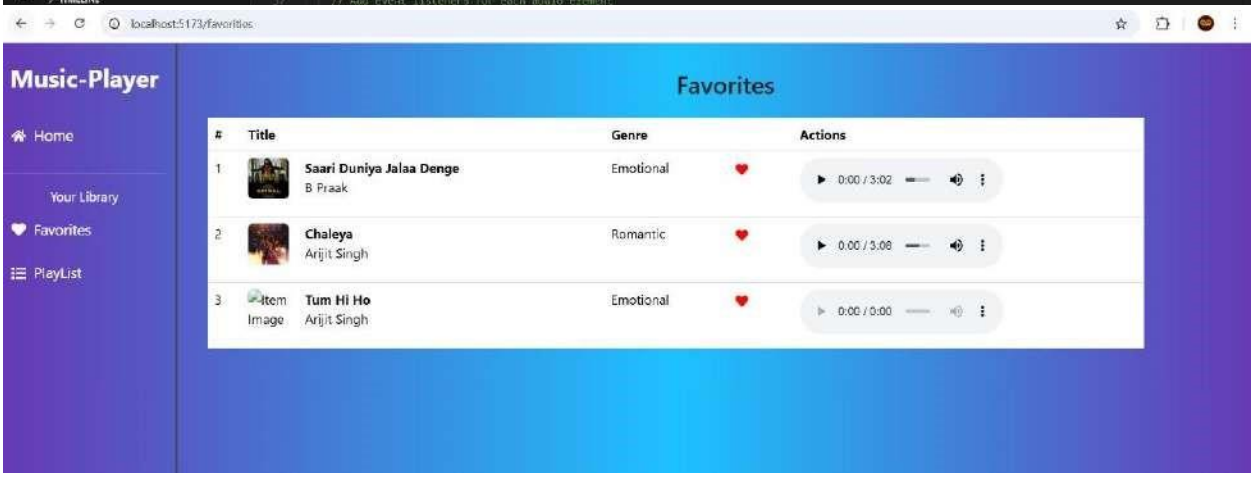


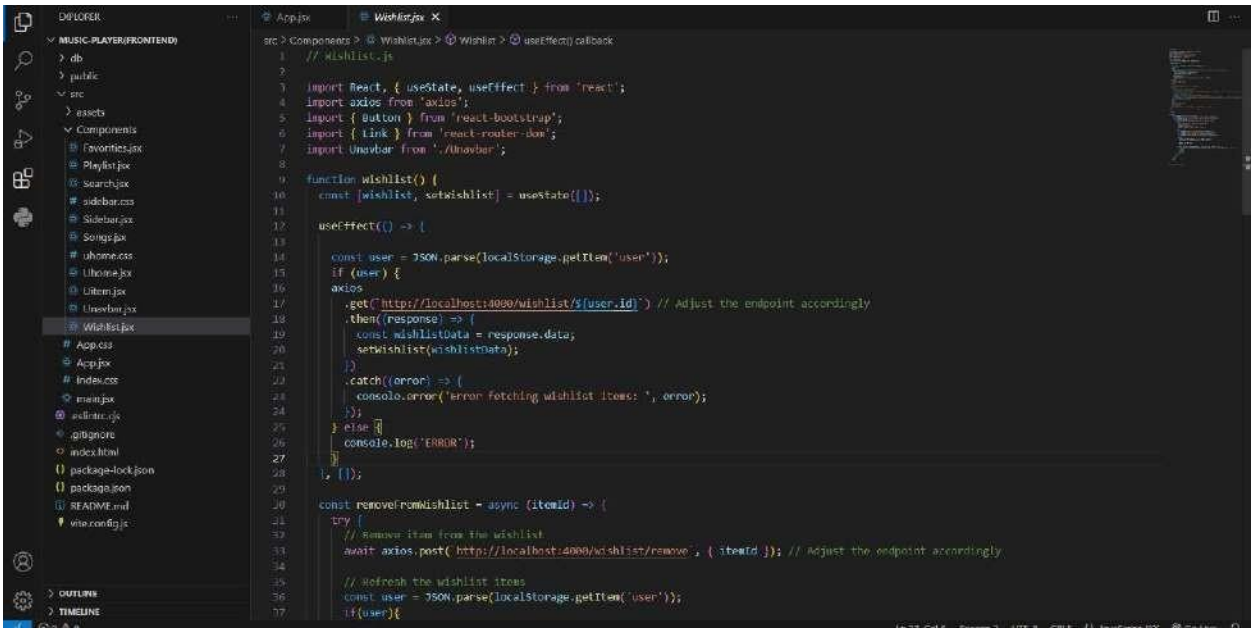
```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { Button, Table } from 'react-bootstrap';
import { FaHeart, FaMugHot, FaMusic, FaPause, FaPauseCircle, FaPlay, FaPlayCircle } from 'react-icons/fa';

function Playlist() {
  const [playlist, setPlaylist] = useState([]);
  const [currentlyPlaying, setCurrentlyPlaying] = useState(null);
  const [isPlaying, setIsPlaying] = useState(false);

  useEffect(() => {

    axios
      .get('http://localhost:3000/playlist')
      .then((response) => {
        const playlistData = response.data;
        setPlaylist(playlistData);
      })
      .catch((error) => {
        console.error('Error fetching playlist items: ', error);
      });


    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);
      });
    };
```



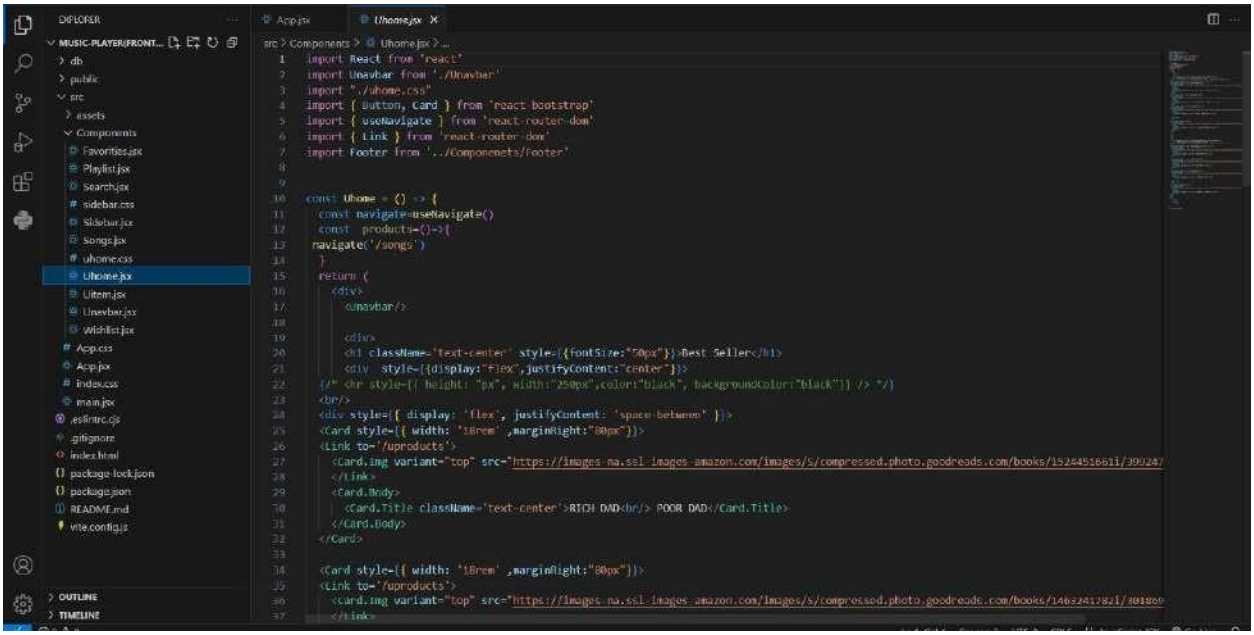


```
import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { Button, Table } from 'react-bootstrap';
import { FaHeart, FaMusic, FaPause, FaPauseCircle, FaPlay, FaPlayCircle } from 'react-icons/fa';

function Favorities() {
  const [playlist, setPlaylist] = useState([]);
  const [currentlyPlaying, setCurrentlyPlaying] = useState(null);

  useEffect(() => {

    axios
      .get('http://localhost:3000/favorities')
      .then((response) => {
        const playlistData = response.data;
        setPlaylist(playlistData);
      })
      .catch((error) => {
        console.error('Error fetching playlist items: ', error);
      });


  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);
    });
  };
```





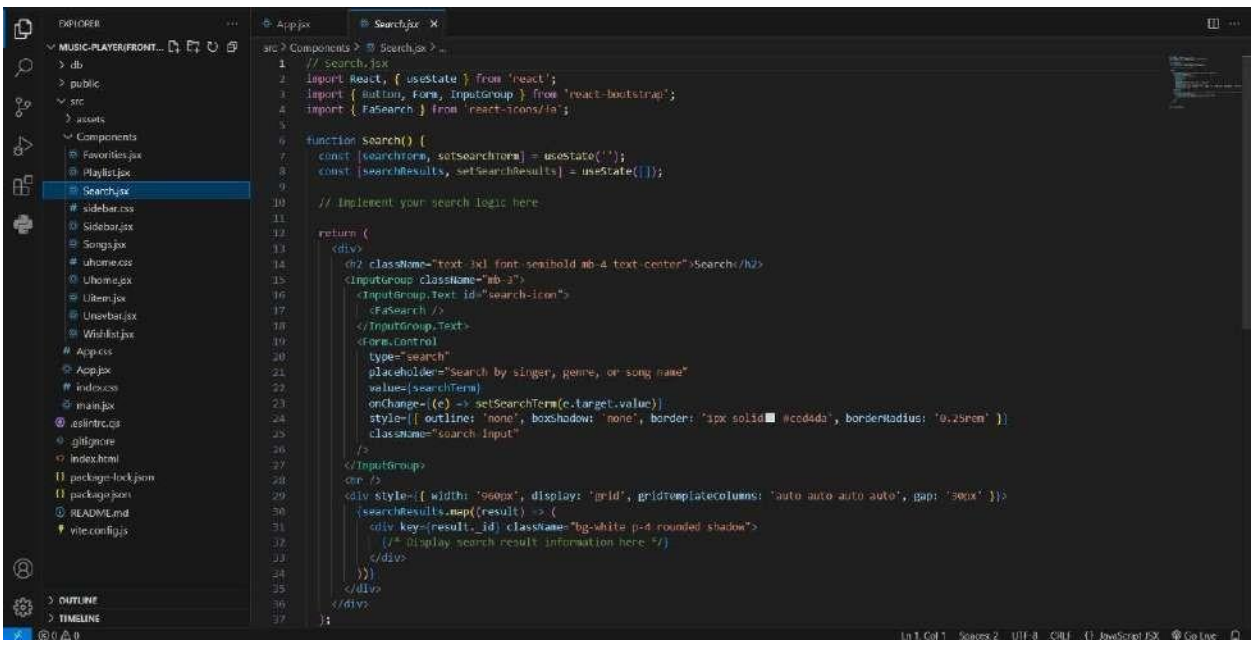
```
// Wishlist.js

import React, { useState, useEffect } from 'react';
import axios from 'axios';
import { Button } from 'react-bootstrap';
import { Link } from 'react-router-dom';
import Unavbar from './Unavbar';

function Wishlist() {
  const [wishlist, setWishlist] = useState([]);

  useEffect(() => {

    const user = JSON.parse(localStorage.getItem('user'));
    if (user) {
    axios
      .get(`http://localhost:4000/wishlist/${user.id}`) // Adjust the endpoint accordingly
      .then((response) => {
        const wishlistData = response.data;
        setWishlist(wishlistData);
      })
      .catch((error) => {
        console.error('Error fetching wishlist items: ', error);
      });
    } else {
      console.log('ERROR');
    }
  }, []);

  const removeFromWishlist = async (itemId) => {
    try {
      // Remove item from the wishlist
      await axios.post(`http://localhost:4000/wishlist/remove`, { itemId }); // Adjust the endpoint accordingly

      // Refresh the wishlist items
      const user = JSON.parse(localStorage.getItem('user'));
      if(user){
```


EXPLORER
MUSIC-PLAYER(FRONT...
db
public
src
assets
Components
Favorites.jsx
Playlist.jsx
Search.jsx
sidebar.css
Sidebar.jsx
Songs.jsx
uhome.css
Uhome.jsx
Uitem.jsx
Unevbar.jsx
Wishlist.jsx
App.css
App.jsx
index.css
main.jsx
.eslintrc.cjs
.gitignore
index.html
package-lock.json
package.json
README.md
vite.config.js
OUTLINE
TIMELINE
App.jsx
Uhome.jsx
src > Components > Uhome.jsx > ...
import React from 'react'
import Unavbar from './Unavbar'
import "./uhome.css"
import { Button, Card } from 'react-bootstrap'
import { useNavigate } from 'react-router-dom'
import { Link } from 'react-router-dom'
import Footer from '../Componenets/Footer'
const Uhome = () => {
const navigate=useNavigate()
const products=()=>{
navigate('/songs')
}
return (

<Unavbar/>

<h1 className='text-center' style={{fontSize:"50px"}}>Best Seller</h1>
<div style={{display:"flex",justifyContent:"center"}}>
{/* <hr style={{ height: "px", width:"250px",color:"black", backgroundColor:"black"}} /> */}

<div style={{ display: 'flex', justifyContent: 'space-between' }}>
<Card style={{ width: '18rem' ,marginRight:"80px"}}>
<Link to='/uproducts'>
<Card.Img variant="top" src="https://images-na.ssl-images-amazon.com/images/s/compressed.photo.goodreads.com/books/1524516611/39924
</Link>
<Card.Body>
<Card.Title className='text-center'>RICH DAD
 POOR DAD</Card.Title>
</Card.Body>
</Card>
<Card style={{ width: '18rem' ,marginRight:"80px"}}>
<Link to='/uproducts'>
<Card.Img variant="top" src="https://images-na.ssl-images-amazon.com/images/s/compressed.photo.goodreads.com/books/1465241982i/801869
</Link>


EXPLORER
MUSIC-PLAYER(FRONT...
db
public
src
assets
Components
Favorites.jsx
Playlist.jsx
Search.jsx
sidebar.css
Sidebar.jsx
Songs.jsx
uhome.css
Uhome.jsx
Uitem.jsx
Unevbar.jsx
Wishlist.jsx
App.css
App.jsx
index.css
main.jsx
.eslintrc.cjs
.gitignore
index.html
package-lock.json
package.json
README.md
vite.config.js
OUTLINE
TIMELINE
App.jsx
Search.jsx
src > Components > Search.jsx > ...
// Search.jsx
import React, { useState } from 'react';
import { Button, Form, InputGroup } from 'react-bootstrap';
import { FaSearch } from 'react-icons/fa';
function Search() {
const [searchTerm, setSearchTerm] = useState('');
const [searchResults, setSearchResults] = useState([]);
// Implement your search logic here
return (

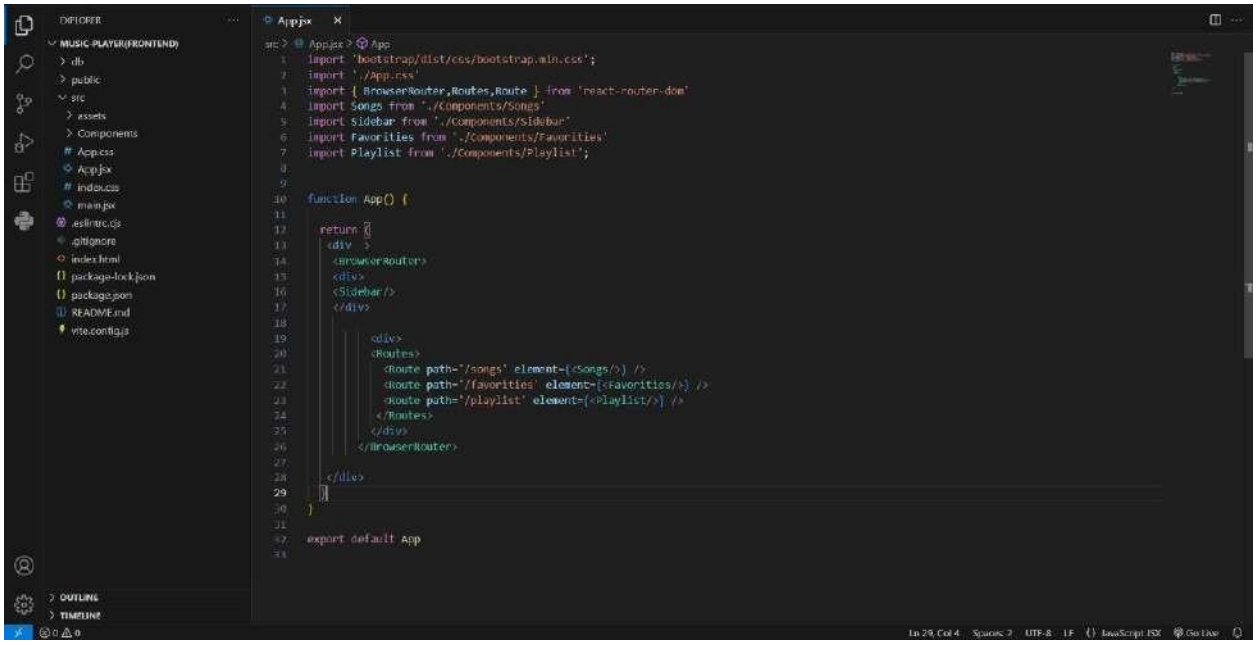
<h2 className="text-3xl font-semibold mb-4 text-center">Search</h2>
<InputGroup className="mb-3">
<InputGroup.Text id="search-icon">
<FaSearch />
</InputGroup.Text>
<Form.Control
type="search"
placeholder="Search by singer, genre, or song name"
value={searchTerm}
onChange={(e) => setSearchTerm(e.target.value)}
style={{ outline: 'none', boxShadow: 'none', border: '1px solid #ced4da', borderRadius: '0.25rem' }}
className="search-input"
/>
</InputGroup>

<div style={{ width: '960px', display: 'grid', gridTemplateColumns: 'auto auto auto auto', gap: '30px' }}>
{searchResults.map((result) => (

{/* Display search result information here */}

))}

);



Styling approach used in the "Rhythmic Tunes: Your Melodic Companion" application:

CSS Frameworks/Libraries:

The application uses a combination of CSS frameworks and libraries to achieve a consistent and responsive design:

- **Tailwind CSS:** A utility-first CSS framework for building custom user interfaces.

- **Styled Components:** A library for styling React components using CSS-in-JS.

Pre-processors:

- **Sass:** A CSS pre-processor used for writing more efficient and modular CSS code.

Theming:

The application implements a custom design system with a focus on accessibility and consistency:

- **Color Scheme**: A custom color scheme with a primary color (#3498db), secondary color (#f1c40f), and accent color (#2ecc71).
- **Typography**: A custom typography system using Open Sans as the primary font.
- **Spacing**: A consistent spacing system using a combination of margin and padding.

Benefits:

The use of CSS frameworks, libraries, and pre-processors provides several benefits:

- **Consistency:** A consistent design language throughout the application.
- **Efficiency:** Faster development and maintenance times using pre-built components and utility classes.
- **Customizability**: Easy customization of the design system using Sass variables and Styled Components.

Testing

Testing strategy for the "Rhythmic Tunes: Your Melodic Companion" application:

Testing Strategy:

The testing approach for components includes:

Unit Testing:

- **Jest:** Used for unit testing individual components and functions.
- **React Testing Library:** Used for testing React components in isolation.

Integration Testing:

- **Jest:** Used for integration testing multiple components and their interactions.
- **React Testing Library**: Used for testing the integration of React components.

End-to-End Testing:

- **Cypress:** Used for end-to-end testing the entire application, simulating user interactions.

Code Coverage:

- **Jest Coverage:** Generates code coverage reports, highlighting areas of the codebase that need more testing.
- **Codecov:** Used for tracking code coverage over time, providing insights into areas of improvement.
- **Test-Driven Development (TDD):** Encourages writing tests before writing code, ensuring that all functionality is thoroughly tested.

Testing Tools:

The following testing tools are used:

- **Jest:** A popular testing framework for JavaScript.
- **React Testing Library:** A testing library for React components.
- **Cypress:** An end-to-end testing framework.
- **Codecov:** A code coverage tracking tool.

Known Issues:

1. **Audio playback issue:** Audio playback may not work on older browsers.
2. **Composition saving bug:** Compositions may not save correctly if the user's internet connection is slow.
3. **Navigation menu issue**: The navigation menu may not display correctly on smaller screens.
4. **Search functionality bug:** The search functionality may not work correctly for compositions with special characters.
5. **User profile update issue:** User profiles may not update correctly if the user's email address is changed.

Future Enhancement

1. **Collaboration Tool**: Allow multiple users to collaborate on compositions in real-time.
2. **Audio Effects**: Add audio effects such as reverb, delay, and distortion to enhance the audio experience.
3. **MIDI Support**: Add support for MIDI devices to allow users to create compositions using external instruments.
4. **Community Feed:** Create a community feed where users can share their compositions and interact with others.

Improvements:

1. **Enhanced User Interface**: Improve the user interface to make it more intuitive and userfriendly.
2. **Animation and Transitions:** Add animations and transitions to enhance the user experience.
3. **Responsive Design:** Improve the responsive design to ensure that the application works well on all devices.
4. **Accessibility Features:** Add accessibility features such as screen reader support and keyboard navigation.

Enhanced Styling:

1. **Customizable Themes:** Allow users to customize the theme of the application.
2. **Advanced Typography:** Improve the typography of the application to make it more visually appealing.
3. **Icon Library:** Add an icon library to provide more visual options for users.

Technical Enhancements:

1. **Optimize Performance**: Optimize the performance of the application to improve load times and responsiveness.
2. **Security Enhancements:** Enhance the security of the application to protect user data.

3. **Integration with Other Services:** Integrate the application with other services such as music streaming platforms and social media.