RHYTHMIC TUNES

TEAM MEMBERS:

P. THULASI

R. THULASI

J. PALLAVI

M. SRIPRIYA

1. PROJECT OVERVIEW:

Purpose: RHYTHMIC TUNES is a dynamic and engaging music streaming platform designed to provide users with a seamless listening experience. The project aims to deliver high-quality music streaming, personalized recommendations, and interactive features that enhance user engagement. Whether users want to discover new music, create playlists, or enjoy their favorite tunes, RHYTHMIC TUNES ensures a smooth and enjoyable experience across devices.

FEATURES:

- **User-Friendly Interface:** A clean and intuitive UI/UX design for easy navigation.
- **Music Streaming:** High-quality audio playback with a vast library of songs across various genres.
- **Personalized Playlists:** AI-driven recommendations based on listening history and preferences.
- **Search and Discovery:** Advanced search filters to find songs, artists, and albums effortlessly.
- **Offline Mode:** Download music for offline listening.
- **Social Integration:** Share favorite tracks and playlists with friends and followers.
- **Cross-Platform Compatibility:** Responsive design ensuring accessibility across web and mobile devices.
- **Dar**
- **k & Light Modes:** Customizable themes to enhance user experience.

- **Real-Time Lyrics:** Sync lyrics with the song for a karaoke-style experience.
- **Radio & Podcasts:** Access live radio and trending podcasts within the platform.

ARCHITECTURE:

1. Component Structure

The project follows a modular React component structure to ensure scalability and maintainability:

- **App Component**: The root component managing global state and routing.
- **Layout Components**: Common UI wrappers such as `Navbar`, `Sidebar`, and `Footer`.
- **Feature Components**:
    - `Player`: Controls music playback.
    - `Track List`: Displays a list of songs.
    - `Track Item`: Represents individual songs.
    - `Playlist`: Handles user-created playlists.
    - `Search Bar`: Allows users to search for tracks.

- **Auth Components** (if applicable): Handles login, signup, and authentication logic.
- **Pages**: Separate pages like `Home`, `Library`, and `Settings`.

2. State Management

- **Context API**: Used for lightweight state management such as user authentication and theme preferences.
- **Redux (or Zustand, Recoil, etc.)**: If complex state management is needed, Redux can be used to manage global states like the currently playing track, playlist data, and user preferences.

3. Routing

```
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/library" element={<Library />} />
  <Route path="/playlist/:id" element={<Playlist />} />
  <Route path="/track/:id" element={<TrackDetails />} />
  <Route path="/search" element={<SearchResults />} />
  <Route path="/settings" element={<Settings />} />
</Routes>
```

- **React Router** is used to handle navigation between different pages:
- **Dynamic Routing**: Enables track and playlist details pages using URL parameters.
- **Protected Routes**: If authentication is required, routes can be wrapped in a higher-order component to restrict access.

SETUP INSTRUCTION FOR RHYTHMIC TUNE

1. Prerequisites

Before setting up **Rhythmic Tunes**, ensure you have the following installed:

- **Node.js** (v16 or later) – Download Here
- **npm** or **yarn** – Comes with Node.js (Check with `node -v` and `npm -v`)
- **Git** – Download Here
- **Code Editor** (e.g., VS Code) – Download Here

2. Installation Guide

Step 1: Clone the Repository

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

Step 2: Install Dependencies

Using npm:

```
npm install
```

Or using yarn:

```
yarn install
```

Step 3: Set Up Environment Variables

Create a **.env** file in the root directory and add necessary configurations:

```
REACT_APP_API_URL=your_backend_api_url
REACT_APP_FIREBASE_KEY=your_firebase_key
REACT_APP_SPOTIFY_CLIENT_ID=your_spotify_client_id
```

Replace placeholders with actual values.

Step 4: Start the Development Server

```
npm start
```

or

```
yarn start
```

This will start the React app on `http://localhost:3000/`.

Step 5: Build for Production (Optional)

```
npm run build
```

This creates an optimized production-ready build in the `build/` directory.

3. Additional Notes

- If you're using **Docker**, a `Dockerfile or docker-compose.yml` should be configured.

- **If there are** database connections**, ensure the backend is running before launching the app.**

FOLDER STRUCTURE:

A well-organized project structure helps with maintainability and scalability. Below is an ideal folder structure for **Rhythmic Tunes**:

rhythmic-tunes/

```
rhythmic-tunes/

|— public/            # Static files (index.html, icons, etc.)

|— src/               # Main source code

|   ├— assets/        # Static assets like images, fonts, icons

|   ├— components/    # Reusable UI components

|   |   ├— Player/    # Music player components

|   |   ├— Playlist/  # Playlist-related components

|   |   ├— Track/     # Individual track components

|   |   ├— UI/        # Generic UI components (Button, Modal, etc.)

|   ├— pages/         # Route-specific page components

|   |   ├— Home.jsx    # Home page

|   |   ├— Library.jsx  # User's saved songs

|   |   ├— Search.jsx   # Search page

|   |   ├— Playlist.jsx # Playlist details page
|   ├── hooks/          # Custom React hooks
|   ├── context/        # Context API providers
|   ├── store/          # Redux or Zustand state management (if used)
|   ├── utils/          # Utility functions/helpers
|   ├── routes/         # React Router setup
|   ├── services/       # API calls and integrations (e.g., Spotify, Firebase)
|   ├── styles/         # Global styles (CSS, SCSS, Tailwind)
|   ├── App.jsx         # Main App component
|   ├── index.js        # Entry point
|── .env                # Environment variables
|── package.json        # Dependencies and scripts
|── README.md           # Project documentation
```

1. Client: React Application Organization

- **`assets/`**: Contains images, icons, and fonts used in the project.
- **`components/`**: Houses reusable UI components such as buttons, modals, and dropdowns.
  - `Player/`: Components for controlling playback (Play, Pause, Seekbar).
  - `Playlist/`: Components for displaying and managing playlists.
  - `Track/`: Individual track-related UI components.
  - `UI/`: Generic components like `Button.jsx`, `Modal.jsx`.
- **`pages/`**: Defines different page views that correspond to React Router routes.
- **`hooks/`**: Custom React hooks for handling app-specific logic.
- **`context/`**: Context API providers for global state management.
- **`store/`**: Redux/Zustand slices if a centralized state management library is used.

- **`services/`**: Handles API calls and external service integrations.
- **`styles/`**: Global styles, CSS modules, or Tailwind configurations.

2. Utilities: Helper Functions, Custom Hooks, and Utilities

- **`utils/`**: Contains helper functions for formatting dates, handling API responses, and managing local storage.
  - `formatTime.js` – Converts seconds to `mm:ss` format.
  - `fetchWithCache.js` – Caches API requests for better performance.
  - `debounce.js` – Helps optimize search input performance.
- **`hooks/`**: Contains custom React hooks to abstract logic.
  - `useAuth.js` – Manages authentication logic.
  - `usePlayerControls.js` – Controls playback state.
  - `useFetch.js` – Handles API data fetching.
- **`services/`**: API integration services.
  - `spotifyService.js` – Fetches music data from Spotify API.
  - `firebaseAuth.js` – Handles user authentication via Firebase.

RUNNING THE APPLICATION:

1. Start the Frontend Server

Navigate to the project directory and run:

cd rhythmic-tunes

npm start

or, if using Yarn:

yarn start

- This will launch the frontend React application on **`http://localhost:3000/`**.
- The server will automatically reload on file changes.

Running with a Backend (If Applicable)

If Rhythmic Tunes depends on a backend API, ensure the backend server is running before starting the frontend.

For example:

cd backend

npm start

Then, start the frontend as described above.

3. Additional Commands

- **Run the app in development mode:**

  npm run dev

- **Build for production:**

  ```
  npm run build
  ```

- **Lint and fix issues:**

  ```
  npm run lint --fix
  ```

- **Run tests (if implemented):**

  ```
  npm test
  ```

COMPONENT DOCUMENTATION:

1. Key Components

1.1 Player Component:

Purpose: Controls music playback, including play, pause, seek, and volume control.
Props:

<Player

  track={track}   // Object containing track details (title, artist, duration)

  is Playing={true}   // Boolean: Whether a track is playing

  on Play Pause={() => {}}   // Function: Handles play/pause toggle

  on Seek={(time) => {}}   // Function: Seeks to a specific time in the track

/>

Track List Component

Purpose: Displays a list of tracks, either from a playlist or search results.
Props:

<Track List

  tracks={track Array}   // Array of track objects

  on Track Select={(id) => {}} // Function: Handles track selection

/>

`Playlist` Component:

Purpose: Shows user-created playlists and allows adding/removing tracks.

Props:

<Playlist

  name="Chill Vibes"   // String: Playlist name

  tracks={play list Tracks} // Array: List of track objects in the playlist

on Add Track={(track) => {}} // Function: Adds a track to the playlist

on Remove Track={(track) => {}} // Function: Removes a track from the playlist

/>

Search Bar Component

Purpose: Enables users to search for songs.
Props:

```
        <Search Bar

          placeholder="Search for songs..." // String: Input placeholder text

          on Search={(query) => {}}        // Function: Handles search input changes

        />
```

2. Reusable Components

2.1 Button Component

Purpose**: Standard button used across the app.**
Props**:**

```
<Button

  text="Play"              // String: Button label

  on Click={() => {}}       // Function: Handles button click

  variant="primary"        // String: Defines button style (primary, secondary)

  disabled={false}         // Boolean: Disables button if true

/>
```

`Modal` Component:

Purpose: Displays pop-up dialogs for user actions.
Props:

```
<Modal
  title="Add to Playlist"  // String: Modal title
  is Open={true}          // Boolean: Controls modal visibility
  on Close={() => {}}      // Function: Handles closing modal
>
  <p>Content goes here...</p>  // Children: Inner content
</Modal>
```

`Loader` Component

Purpose: Displays a loading animation while data is being fetched.
Props:

```
<Loader size="large" />  // "small" | "medium" | "large"
```

STATE MANAGEMENT:

State management in **Rhythmic Tunes** ensures a smooth and interactive user experience by efficiently handling music playback, playlists, authentication, and UI state.

1. Global State Management

◈ **Approach:** The application uses **Context API** for lightweight global state management. If the app scales, **Redux Toolkit** or **Zustand** can be introduced.

1.1 Global State with Context API

- **Why?**

- Manages **user authentication**, **current playing track**, and **playlist state** across components.
- Prevents unnecessary prop drilling.

Implementation:

```
import { create Context, use Context, use State } from "react";


const Player Context = create Context();


export const Player Provider = ({ children }) => {
  const [current Track, set Current Track] = use State(null);
  const [is Playing, set Is Playing] = use State(false);


  return (
    <Player Context .Provider value={{ current Track, set Current Track,
is Playing, set Is Playing }}>
      {children}
    </Player Context. Provider>
  );
};


export const use Player = () => use Context (Player Context);
```

Usage in Components:

```
import { use Player } from "../context/Player Context";
```

```
const Player Controls = () => {
  const { is Playing, set Is Playing } = use Player();

  return (
    <button on Click={() => set Is Playing(!is Playing)}>
      {is Playing ? "Pause" : "Play"}
    </button>
  );
};
```

1.2 Global State with Redux (Alternative):

```
npm install @reduxjs/toolkit react-redux
```

Redux Store (`store.js`)

```
import { configure Store, create Slice } from "@reduxjs/toolkit";

const player Slice =  create Slice({
  name: "player",
  initial State: { current Track: null, is Playing: false },
  reducers: {
    set Track: (state, action) => { state .current Track = action .payload;
},
    toggle Play: (state) => { state .is Playing = !state . is Playing; },
  },
});
```

export const { set Track, toggle Play } = player Slice. actions;

export const store = configure Store({ reducer: { player: player Slice. reducer } });

3. State Flow Across the Application

- **Playback state (`current Track`, `is Playing`)** is managed **globally** via Context API or Redux.
- **Search input, form states, and UI toggles** are handled **locally** using `use State`.
- **Authentication state** is stored globally in Context or Redux to persist user sessions.

User Interface (UI) in Rhythmic Tunes

The **Rhythmic Tunes UI** is designed for a smooth, visually appealing, and user-friendly music streaming experience. It follows a **modern, responsive layout** with a dark theme and intuitive navigation.

1. Key UI Features

Home Page

Purpose: Displays featured playlists, trending songs, and user recommendations.
Components:

- **Hero Section**: Showcases the latest trending playlist.
- **Playlist Grid**: Displays user-generated and featured playlists.
- **Recently Played**: Shows the last played tracks for quick access.

User Interaction: Click on any track or playlist to start playing.

Music Player

Purpose: Provides full music playback controls.
Components:

- **Play/Pause Button**: Controls music playback.
- **Seek Bar**: Allows users to scrub through the track.
- **Volume Control**: Adjusts playback volume.
- **Track Info**: Displays song title and artist name.
- **Next/Previous Buttons**: Skip between tracks.

User Interaction: Hover animations, dynamic progress bar updates, and keyboard shortcuts.

Search Page

Purpose: Lets users search for songs, artists, or albums.
Components:

- **Search Input**: Type to filter tracks dynamically.
- **Search Results**: Displays results in real-time with album covers and play buttons.
- **Filter Options**: Sort by artist, genre, or popularity.

User Interaction: Live search with debounce for performance optimization.

Playlist Page

Purpose: Allows users to create, manage, and play custom playlists.
Component

- **Playlist Cover & Name**
- **Track List with Play Buttons**
- **"Add to Playlist" Functionality**

User Interaction: Drag and drop to reorder tracks in the playlist

Styling in Rhythmic Tunes

Rhythmic Tunes uses **Tailwind CSS** for a modern, responsive, and highly customizable design. The UI follows a **dark-themed aesthetic** with vibrant accent colors for highlights, ensuring a sleek and immersive music experience. Components are styled with **flexbox and grid layouts** for a clean structure, while **Framer Motion** enhances user interactions with smooth animations. Custom utility classes provide consistency across buttons, modals, and cards. The design prioritizes **mobile-first responsiveness**, ensuring seamless playback and navigation across all devices.

Testing in Rhythmic Tunes

Rhythmic Tunes follows a **comprehensive testing approach** to ensure reliability and performance. **Jest** and **React Testing Library** are used for unit and integration tests, covering components like the **Music Player, Search, and Playlist**. End-to-end (E2E) testing is performed with **Cypress** to validate user flows, such as authentication and track playback. ESLint and Prettier enforce code quality, while manual testing ensures smooth UI interactions. Continuous testing is integrated into the CI/CD pipeline for consistent performance across updates.

Screenshots or Demo

https://drive.google.com/file/d/1qDIjZ-wWNbcnhXDLSmLvY-L0veCSykIM/view?usp=sharing

Known Issues:

Rhythmic Tunes currently has a few known issues, including **playback delay** on the first track play, **search lag** with large datasets, and **playlist sync issues** where updates don't reflect immediately. Some users experience **UI overlaps on mobile**, affecting elements like volume controls. Additionally, **authentication session expiry** may log users out unexpectedly. Planned fixes include **optimizing state updates, improving responsiveness, and implementing token refresh mechanisms** for a smoother experience.

Future Enhancements in Rhythmic Tunes

Planned improvements for Rhythmic Tunes include **AI-powered music recommendations**, **offline playback support**, and **real-time lyrics display**. The UI will be enhanced with **dark/light mode toggling** and **theme customization**. Performance optimizations, such as **faster search algorithms and improved caching**, will enhance responsiveness. Additionally, **social features** like playlist sharing, collaborative playlists, and in-app messaging are in development to create a more interactive experience.