RYTHMIC TUNES:

1.Introduction

- **Project Title**:

  Rythmic tunes:your melodic companion

- **Team Members**:

  Our team consists of 4 members and they are,

  - ✔ Varshini.M - as the leader
  - ✔ Gladys Nancy.A - member
  - ✔ Gomathi.K - member
  - ✔ Gowri shankari .k - member

2.Project Overview

- **Purpose**:

  The project's purpose is to act as an "augmented companion" for musicians, specifically those experiencing creative blocks, by providing an AI-driven, emotion-based tool to assist in musical composition, potentially using a framework that allows users to interact with notes and gestures to receive rhythmic and melodic recommendations. It aims to help users overcome creative hurdles and facilitate a more intuitive and emotionally resonant music creation process, rather than being a simple music streaming or consumption app.

- **Features**:

  . The "Rhythmic Tunes" frontend offers a responsive and intuitive music streaming experience, featuring core functionalities like a song listing, the ability to browse music, create playlists, and a search feature for finding specific tracks

3.Architecture

- **Component Structure**:

  A typical "RhythmicTunes" project might organize components hierarchically:

  App.js (Root Component):

  This serves as the entry point, often handling global state, routing, and rendering major layout components like Header, Sidebar, and the main content area.

  Layout Components:

  These define the overall structure of the application, such as Header (containing navigation, search bar), Sidebar (for playlists, genres), and Footer (with playback controls).

  Page Components:

  These represent specific views or pages within the application, like HomePage, PlaylistPage, ArtistPage, or SearchPage. They orchestrate the display of multiple smaller components.

  Feature Components:

  These encapsulate specific functionalities, like MusicPlayer (handling audio playback), PlaylistManager (for creating/editing playlists), or UserAuthentication.

  UI Components (Reusable):

  These are small, generic components used across the application for consistent UI, such as Button, Input, Card, SongListItem, or LoadingSpinner.

  Props (Properties):

  Data flows unidirectionally from parent to child components via props. For example, App.js might pass user data to Header, or PlaylistPage might pass a list of songs to a SongListItem component.

  State Management:

  Local State: Components manage their own dynamic data using useState (functional) or this.state (class).

  Global State: For data shared across many components (e.g., current playing song, user authentication status), state management libraries like Redux or

React Context API are often employed, allowing components to subscribe to and update this shared state.

Event Handling:

Child components can communicate back to parent components by invoking functions passed as props. For instance, a Button component might trigger a function in its parent when clicked, leading to a state update or data fetch.

Context API:

This allows data to be passed down the component tree without manually passing props at each level, useful for themes, user preferences, or authentication details.

Hooks (for Functional Components):

useEffect handles side effects (data fetching, subscriptions), useContext accesses context, and useReducer manages complex state logic.

State Management:

Describe the state management approach used (e.g.,Context API, Redux).

Routing: Explain the routing structure if using react-router or another routing library.

4.Setup Instructions

- ## Prerequisite:

1. SourceTree installed on your system.
2. Android Studio installed on your system.
3. A project repository hosted on a Git repository, such as GitHub, Bitbucket, or GitLab.

- ## Installation

- **Node.js and npm:**

Node.js serves as the JavaScript runtime environment, allowing the execution of JavaScript code locally. npm (Node Package Manager) is bundled with Node.js and is essential for managing project dependencies, including the installation, updating, and removal of packages.

- **React.js:**

  This JavaScript library is fundamental for building the user interface components of the frontend application.

- **Package Manager (e.g., npm):**

  While npm is the default for Node.js projects, it is the primary tool for handling the various libraries and frameworks that constitute the project's dependencies.

- **Specific Node.js packages:**
  The project will likely utilize various Node.js packages from the npm registry to implement features such as offline listening, search functionality, and potentially music-related functionalities like MIDI processing or audio manipulation. Examples of such packages might include `react-router-dom` for navigation, `axios` for API calls, and potentially packages for audio playback or music synthesis if advanced features are planned.

5.Folder Structure

Client:

Organizing a React project with a well-planned folder structure is very important for readability, scalability, and maintainability. A clear structure helps in managing code, configurations, modules, and other assets effectively. In this article, we are going to learn the folder structure of the React project. In this article, we'll explore the best practices for organizing the folder structure of a React project.

Steps to Create Folder Structure:

Step 1: Open the terminal, go to the path where you want to create the project and run the command with the project name to create the project.

Eg:npx create-react-app folder-structure

Step 2: After project folder has been created, go inside the folder using the following command.

Eg:cd folder-structure

Step 3: Install the dependencies required for your project (if any) using the following command.

Eg:npm i package-name

Step 4: Run the command git-init to initialized the git in the project. This will add .gitignore file.

Eg:eggit init

Step 5: Create a file named Readme.md which will contain all the info of the project.

Eg:touch Readme.md

Step 6: Create a file with extension .env which will contain the sensitive information and credentials of the project.

Eg:touch process.env

Step 7: Create a file named .gitignore so that all the un-necessary files and folders should not uploaded to github.

Eg:touch .gitignore

Step 8: Create folder like components (contains main components) and pages (contains files which combines components to make a page).

Eg:mkdir components
touch Navbar.jsx

Improved Files and Folders Structure:

For managing the project in more concise way we can create these folder for more manageable code.

- **Components Folder**
- **Context Folder**
- **Hooks Folder**
- **Services Folder**
- **Utils Folder**

- **Assets Folder**
- **Config Folder**
- **Styles Folder**

    - **Utilities**:

React Custom Hooks are JavaScript functions which give you the ability to reuse stateful logic throughout your React codebase. When using Custom Hooks we tap into the React Hooks API that lets us manage our state and its side effects in a way that follows the React functional component process.

One of the unique characteristics of Custom Hooks is being able to leverage state management which means that we can access the React built-in hooks like `useState`, and `useEffect`. Another unique identifier is the fact that we have to follow the named conventions for React Custom Hooks as we must prefix the start of the hook with the word `use` followed by its name, for example, `useFetch`.

When we use Custom Hooks they can access and change a component state, plus lifecycle methods as they are deeply interconnected with the logic for React component

This Custom Hook is called useFetch and has reusable logic for fetching data from an API. It can manage the loading and error states and can be imported into multiple components.

Now that we have a fundamental understanding of React Custom Hooks let's see how they compare to Helper Functions.

6.Running the Application

Provide commands to start the frontend server locally.

- **Frontend**: npm start in the client directory.

7.Component Documentation

- ## Key Components:

- **Melody:**

  The main tune or a sequence of notes forming a memorable pattern. It's the central theme that often carries the emotional weight of the piece.

- **Rhythm:**

  The pattern of sounds and silences over time, organizing music into a structured flow. It's the backbone of the music, giving it a beat and flow.

- **Harmony:**

  The combination of different notes played or sung at the same time to create chords that support the melody. Harmony adds depth and richness to the music.

- **Pitch:**

  The highness or lowness of a sound, determined by its frequency of vibration. It forms the individual notes of the melody.

- **Tempo:**

  The speed of the music, measured in beats per minute (BPM). It influences the mood and feel, from fast and energetic to slow and calm.

- **Timbre:**

  The unique "tone color" of a sound, which allows listeners to distinguish between different voices or instruments.

- **Dynamics:**

  The variation in volume within a piece, using terms like crescendo (gradual increase) and decrescendo (gradual decrease).

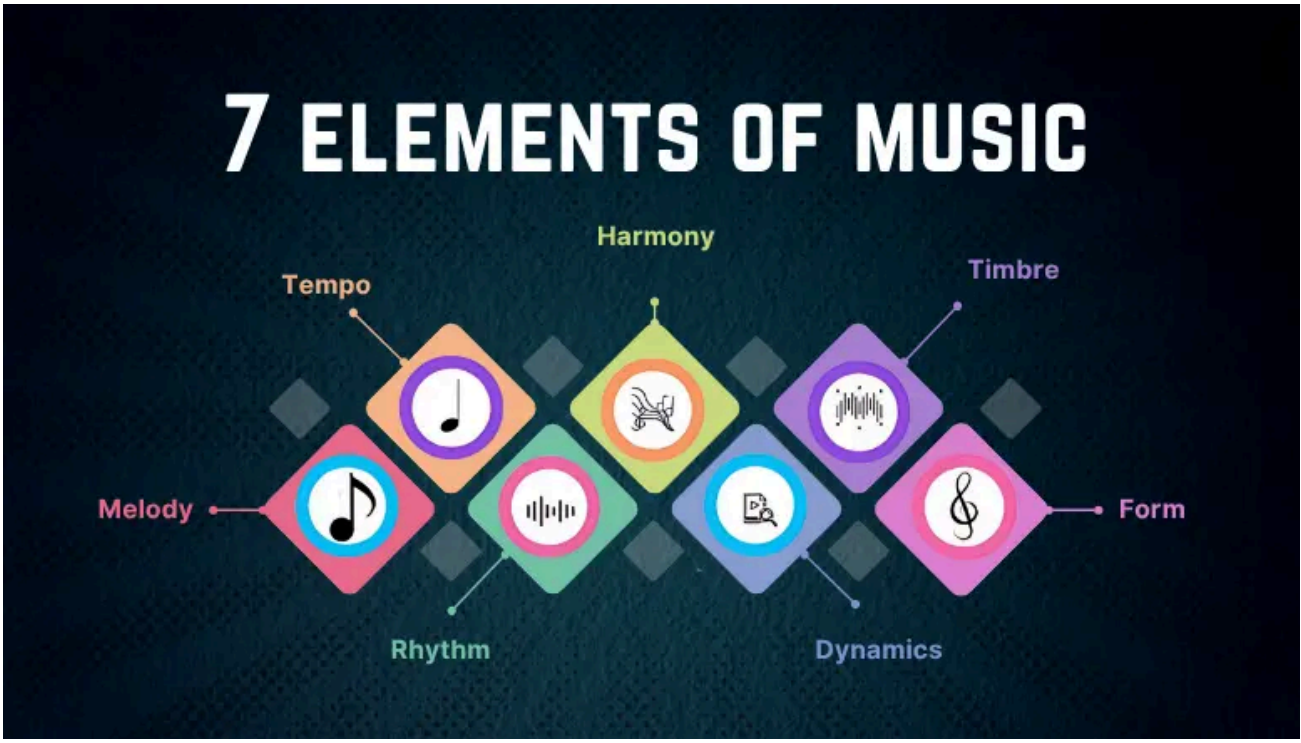
- **Texture:**

  How melody, harmony, and rhythm are combined, creating the overall quality and density of the sound.

  Props (What They Receive)

- **Melody**: receives a specific rhythm and is often accompanied by supporting harmony.
- **Rhythm**: provides the temporal framework for notes and silences, dictating their duration and emphasis.

- **Harmony**: provides the chords and harmonic support for the melody.
- **Pitch**: is a characteristic of individual sounds, while rhythm gives it length.
- **Tempo**: is the underlying speed that all elements are played to.
- **Dynamics**: are variations in volume that add expressive layers to a piece.
- **Texture**: describes how all the elements—melody, harmony, rhythm, and timbre—are woven together.



- **Reusable Components**:

- **User Interface (UI) Elements:**

  A "Modal Component" that displays song details or playlist forms is an example of a reusable UI component in the "RhythmicTunes" application, as it can be used on different screens or sections of the app.

- **Backend Modules:**

  A component that handles music data processing, user authentication, or streaming logic would also be a reusable component.

- **Data Structures:**

Common data formats for songs, artists, or playlists could be considered reusable components.

Configuration.

Configurations refer to the specific settings, parameters, or styles that define how a reusable component functions or appears in a particular instance. For the "RhythmicTunes" Modal Component, configurations might include:

- **Title and Content:**

  The specific text displayed in the modal's header and the data it presents.

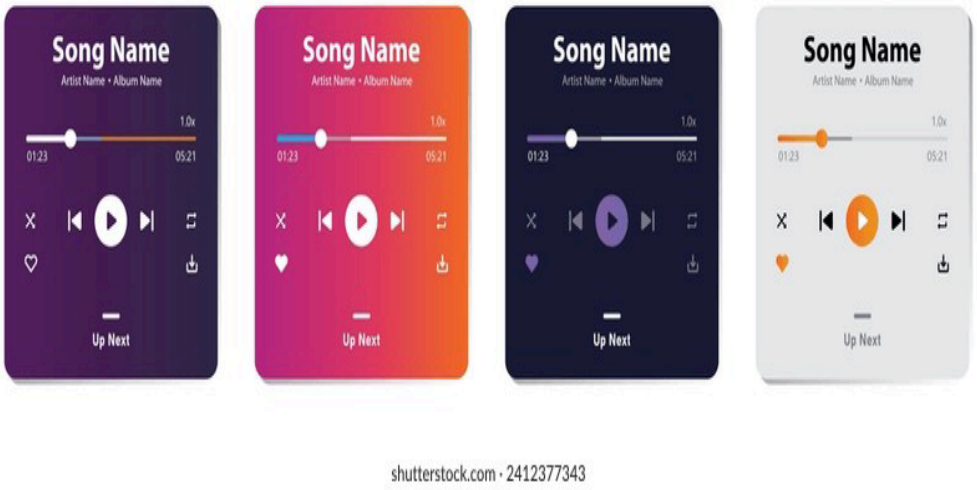
- **Buttons and Actions:**

  Which buttons appear (e.g., "Play," "Add to Playlist") and what actions they trigger.

- **Styling:**

  Colors, fonts, and layout that are specific to the context where the modal is used.

8.User Interface



9.Styling

- **CSS Frameworks/Libraries**:

CSS Frameworks:

- **Bootstrap:**

A popular, comprehensive framework providing pre-styled components (buttons, forms, navigation) and a responsive grid system. It offers a quick way to build visually consistent and responsive interfaces.

- **Tailwind CSS:**

A utility-first framework that provides low-level utility classes (e.g., flex, pt-4, text-lg) to build custom designs directly in HTML. It emphasizes flexibility and customization over pre-built components.

- **Material-UI (MUI):**

A React component library implementing Google's Material Design. It offers a rich set of pre-built, customizable UI components that follow Material Design guidelines.

- **Bulma:**

A modern, open-source CSS framework based on Flexbox. It is purely CSS, meaning no JavaScript dependencies, and focuses on modularity and ease of use.

CSS Libraries:

- **Styled-Components:**

A CSS-in-JS library for React that allows writing CSS directly within JavaScript components using tagged template literals. This approach promotes component-scoped styling and dynamic styling based on component props.

- **Emotion:**

Another performant and flexible CSS-in-JS library, similar to Styled-Components, offering features like critical CSS extraction and server-side rendering support.

CSS Pre-processors:

- **Sass (Syntactically Awesome Style Sheets):**

A powerful pre-processor that extends CSS with features like variables, nesting, mixins, functions, and inheritance. It improves code organization, reusability, and maintainability of large stylesheets.

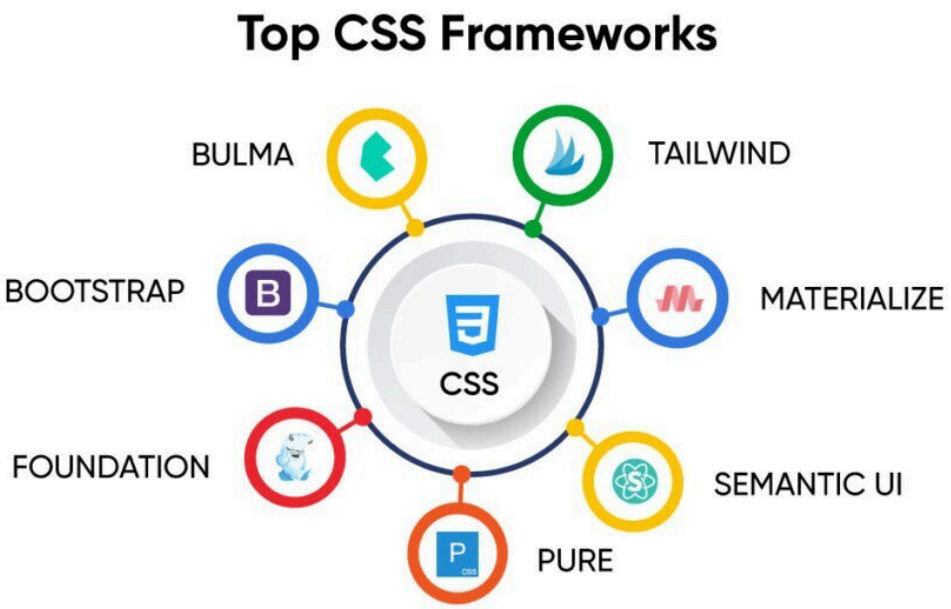
- **Less (Leaner CSS):**

Similar to Sass, Less offers dynamic programming capabilities like functions and operations, making it useful for complex styling tasks.

- **Stylus:**

A minimalist pre-processor with a flexible syntax that supports both indentation-based and CSS-like syntax, along with inline JavaScript expressions for advanced logic.

The choice of framework, library, or pre-processor depends on project requirements, team familiarity, and desired level of customization and control over the styling.



10.Testing

- **Testing Strategy**: Describe the testing approach for components, including unit, integration, and end-to-end testing (e.g., using Jest, React Testing Library).
- **Code Coverage**: Explain any tools or techniques used for ensuring adequate test coverage.

11.Screenshots or Demo

- Provide screenshots or a link to a demo showcasing the application's features and design.

12.Known Issues

- Document any known bugs or issues that users or developers should be aware of.

13.Future Enhancements

- Outline potential future features or improvements, such as new components, animations, or enhanced styling.