RHYTHMICTUNES: YOUR MELODIC COMPANION

(MUSIC STREAMING APP)

NAANMUDHALVAN PROJECT REPORT

Submitted by

TEAM LEADER

R. SWEETHRA (222209389)	**sweethra2005@gmail.com**

TEAM MEMBERS

R. PRIYA (222209376)	**Priyap1856@gmail.com**
G. RUPIKA (222209382)	**rupikagopi@gmail.com**
R. SATHYA (222209387)	**ssathya34912@gmail.com**

DEPARTMENT OF COMPUTER SCIENCE

TAGORE COLLEGE OF ARTS AND SCIENCE

(Affiliated to the University of Madras)

CLC WORKS ROAD, CHROMPET, CHENNAI – 600 044

MARCH - 2025

TABLE OF CONTENT

CHAPTER 1
INTRODUCTION

1.1 Project Overview

Rhythmic Tunes is a music streaming platform designed to provide users with a seamless and engaging experience for discovering, playing, and managing music. The platform will cater to both free and premium users, offering a personalized music experience with curated playlists, social sharing, and offline downloads.

Purpose and Objectives:

- Provide high-quality music streaming with an intuitive interface.
- Offer personalized recommendations and playlist creation.
- Ensure a smooth and responsive user experience across multiple devices.
- Support artists by integrating monetization options for premium features.

Target Users and Expected Benefits:

- **General users**: Enjoy unlimited music streaming, playlist management, and recommendations.
- **Premium users**: Access exclusive content, ad-free streaming, and offline downloads.
- **Admin users**: Manage content, monitor trends, and ensure compliance with platform policies.

1.2 Problem Statement

Current music streaming platforms often suffer from:

- High subscription costs.
- Limited offline features for free users.
- Poor recommendations and lack of personalization.
- Lack of community engagement and social interaction.

Existing Solutions & Limitations

Platforms like Spotify, Apple Music, and YouTube Music dominate the market but have some drawbacks:

- **Expensive premium plans** that restrict essential features.
- **Lack of user control** over recommendations and playlist algorithms.
- **Limited local artist support**, making it difficult for new musicians to gain visibility.

How Rhythmic Tunes Improves User Experience

- Provides affordable subscription plans with better benefits.
- Enhances personalization using AI-based recommendations.
- Encourages user interaction through social and sharing features.
- Supports independent artists with direct uploads and monetization options.

1.3 Scope of the Project

- **Key Features:**
  - Music streaming with a built-in player.
  - Playlist creation and recommendations.
  - Like, comment, and share features.
  - Offline downloads for premium users.
  - Admin dashboard for content management.
- **Platforms Supported:** Web, Android, iOS.
- **Limitations:**
  - Initial version may lack advanced AI-based recommendations.
  - Offline mode may be restricted to premium users.

FEASIBILITY STUDY

CHAPTER 2
FEASIBILITY STUDY

2.1. Technical Feasibility

The **technical feasibility** determines whether the required technology and infrastructure are available to develop and maintain the application.

- **Technology Stack:** The application will be built using **Node.js** with frameworks like **Express.js** for backend development.
- **Database:** NoSQL (**MongoDB, Firebase**) or SQL (**PostgreSQL, MySQL**) for managing user data, playlists, and music metadata.
- **Cloud Storage & CDN: AWS S3, Google Cloud Storage, or Firebase** for storing audio files and **Cloudflare, AWS CloudFront** for fast delivery.
- **Streaming Protocols:** HLS (HTTP Live Streaming) or DASH (Dynamic Adaptive Streaming over HTTP) will be used for **seamless playback** across devices.
- **Authentication: OAuth 2.0, JWT-based authentication, Firebase Auth** for user management.
- **Scalability:** Node.js' **event-driven architecture** enables efficient handling of multiple streaming requests, making it ideal for real-time applications.

2.2. Economic Feasibility

The **economic feasibility** assesses the financial viability of the project.

- **Development Cost:** Estimated **$10,000 - $50,000** depending on features, team size, and third-party services.
- **Infrastructure Cost:** Cloud hosting (AWS, Google Cloud), database storage, and **CDN services** may cost **$500 - $2,000 per month** depending on traffic.
- **Revenue Model:** The app can generate revenue through **subscription plans, advertisements, premium features, and partnerships with artists.**
- **Return on Investment (ROI):** If successfully marketed, it can break even within **12-24 months**, assuming a steady growth of paid subscribers.

2.3. Operational Feasibility

Operational feasibility examines how well the application meets user needs and integrates into daily operations.

- **User Demand:** The growing demand for **on-demand music streaming** makes this application relevant and competitive.
- **User Accessibility:** The app should be **cross-platform** (web, iOS, Android) to ensure broad usability.
- **Maintenance & Support:** Regular updates, bug fixes, and customer support are essential for retaining users.

SYSTEM DESIGN

CHAPTER 3
SYSTEM DESIGN

3.1. System Architecture

The application follows a **microservices-based architecture** to handle different functionalities like **authentication, music streaming, user management, and recommendations**.

High-Level Architecture

- **Client (Frontend)**
  - Web app (React, Vue.js, Angular)
  - Mobile apps (Flutter, React Native, Swift, Kotlin)
- **Backend (Node.js)**
  - Express.js / Nest.js as the web framework
  - RESTful or Graph QL APIs for communication
- **Database**
  - **SQL (PostgreSQL, MySQL)** for structured data like users, subscriptions, and transactions.
  - **NoSQL (MongoDB, Firebase, Cassandra)** for unstructured data like user preferences, listening history.
- **Cloud Storage & CDN**
  - **AWS S3 / Google Cloud Storage** for storing audio files.
  - **CloudFront / Cloudflare CDN** for fast content delivery.
- **Music Streaming Service**

  - HLS (HTTP Live Streaming) or DASH (Dynamic Adaptive Streaming over HTTP) for adaptive bitrate streaming.
- **Authentication & Authorization**
  - **OAuth 2.0, Firebase Auth, JWT** for secure access.
- **Caching & Optimization**
  - **Redis / Memcached** for caching frequently played songs.
- **Logging & Monitoring**
  - **Prometheus, Grafana, ELK Stack** for tracking performance and logs.

3.2. Existing System vs Proposed System

Feature	Existing Solutions	Rhythmic Tunes
Subscription Cost	Expensive Premium Plans	Affordable with more benefits
User Engagement	Limited Social Features	Share, Comment, Follow Users
Recommendations	Basic Algorithm	AI-Powered Personalized Playlists
Offline Mode	Only for Premium Users	Available with Additional Features

3.3. Performance Optimization

- **Load Balancing**
  - Use **NGINX or AWS ALB** to distribute traffic across multiple backend servers.

- **Database Indexing**
  - Index commonly searched fields like **song title, artist, album** to improve query speed.
- **Caching Popular Songs**
  - Use **Redis** to store frequently played songs, reducing database load.
- **Lazy Loading & Pagination**
  - Implement **infinite scrolling & paginated API responses** to improve performance.
- **Compression & Optimization**
  - Convert music files into **compressed formats (AAC, OGG, Opus)** for faster delivery.

3.4. Security Considerations

- **Data Encryption**
  - Encrypt stored passwords using **bcrypt**.
  - Use **HTTPS (SSL/TLS)** for secure data transfer.
- **Access Control**
  - Restrict **API access** using **JWT authentication**.
  - Role-based permissions for **admin, premium, and free users**.
- **Rate Limiting & DDoS Protection**
  - Use **Express Rate Limit, Cloudflare** to prevent abuse.
- **Secure File Access**
  - Use **presigned URLs** to prevent unauthorized access to music files.

MODULES

CHAPTER 4

MODULES

4.1 User Authentication & Authorization Module

- Manages user **registration, login, and authentication**
- Uses **JWT (JSON Web Token)** or **OAuth 2.0** for secure

  token-based authentication
- Supports **social logins** (Google, Facebook, Spotify API)
- Handles **user role management** (admin, premium user, free user)

4.2 Music Upload & Management Module

- Allows **artists/admins** to upload new songs
- Converts & optimizes **audio formats (MP3, AAC, FLAC, WAV, etc.)**
- Stores **metadata** (title, album, artist, duration, genre, etc.)
- Organizes **playlists, albums, and categories**

4.3 Music Streaming & Playback Module

- Implements **on-demand streaming** using HLS (HTTP Live Streaming)
- Supports **adaptive bitrate streaming** for different network speeds
- Uses **WebSockets for real-time playback synchronization**
- Manages **buffering & caching** for seamless music playback

4.4 Search & Discovery Module

- Enables **searching for songs, albums, artists, and playlists**
- Uses **full-text search** for better recommendations
- Implements **filters (genre, language, artist, release year, etc.)**
- Supports **autocomplete suggestions**

4.5 Personalized Playlist & Recommendation Module

- Generates **automated & user-curated playlists**
- Uses **AI/ML-based recommendations** based on listening history
- Suggests **similar songs & trending tracks**
- Supports **user-generated playlists**

4.6 User Profile & Social Features Module

- Allows **users to edit profiles (name, avatar, preferences)**
- Implements **friend lists & following system**
- Enables **likes, comments, and shares**
- Supports **user activity tracking (recently played, top tracks, etc.)**

4.7 Notification & Messaging Module

- Sends **push notifications** for new releases, updates, or offers
- Supports **in-app messaging (for community features)**
- Uses **email notifications** for account updates & promotions

4.8 Admin Dashboard Module

- Provides **analytics & reports** (users, streams, revenue, etc.)

- Manages **music uploads, artists, and payments**
- Controls **user access & bans suspicious activities**

4.9 Offline & Download (Optional)

- **Premium users can download songs** for offline listening.
- **Downloaded songs managed locally.**

4.10 Notifications & Updates

- **Real-time push notifications** for new releases.
- **Subscription renewal reminders.**

IMPLEMENTATION & TECHNOLOGIES

CHAPTER 5

IMPLEMENTATION & TECHNOLOGIES

5.1 Frontend Technologies

The frontend of Rhythmic Tunes is responsible for user interaction, UI design, and handling API calls to the backend.

Web Application (React.js)

- **React.js** is chosen for its component-based architecture, fast performance, and scalability.
- **Key Features:**
  - Reusable components for UI elements like music player, playlists, and song cards.
  - React Router for smooth navigation.
  - State management using Redux or React Context API.
  - API calls using Axios or Fetch API for data retrieval.
  - Responsive design with Tailwind CSS or Material UI.

Mobile Application (Flutter)

- **Flutter** is used for mobile app development due to its cross-platform capabilities.
- **Key Features:**
  - Single codebase for Android and iOS.
  - Beautiful UI with widgets for seamless user experience.
  - Integration with backend APIs using HTTP or Dio package.
  - Secure authentication via Firebase or JWT.

5.2 Backend Technologies

The backend handles user authentication, music storage, data retrieval, and API endpoints.

Backend Framework Choices

- **Option 1: Node.js with Express.js** (JavaScript-based backend)
  - Event-driven and non-blocking I/O for fast performance.
  - RESTful APIs for communication with frontend.
  - Handles user authentication, music metadata, and playlist management.
- **Option 2: Django with Python**
  - High-level Python framework with built-in security.
  - Django REST Framework (DRF) for API development.
  - SQLite/PostgreSQL integration for scalable data handling.

5.3 Database & Storage

Data storage includes user information, song metadata, and playlists.

Database Choices

- **MongoDB (NoSQL)**
  - Flexible document-based structure, ideal for storing user preferences and playlists.
  - Scalable and easy to integrate with Node.js.
- **Firebase Firestore (NoSQL)**
  - Real-time database with fast syncing.
  - Ideal for mobile-first applications.

Storage Solutions

- **AWS S3 (Amazon Simple Storage Service)**
  - Scalable, secure, and widely used for media files.
  - Stores songs, album art, and user-uploaded content.
- **Firebase Storage**
  - Direct integration with Firebase Authentication.
  - Ideal for mobile users due to fast media access.

TESTING & EVALUATION

CHAPTER 6
TESTING & EVALUATION

6.1 Testing Strategies

A well-tested system ensures a smooth user experience with minimal bugs.

1. Unit Testing

- **Frontend:**
  - Testing UI components using Jest and React Testing Library.
  - Ensuring the music player works as expected.
- **Backend:**
  - Testing API endpoints using Postman or Jest (for Node.js).
  - Checking authentication and data retrieval.

2. Integration Testing

- Verifying interactions between frontend and backend.
- Testing API requests and responses.

3. UI/UX Testing

- Conducting user surveys and A/B testing.
- Checking responsiveness across different devices.

6.2 Performance Testing

1. Load Testing

- Simulating thousands of users streaming music simultaneously.
- Tools: Apache JMeter, K6.

2. Response Time Optimization

- Implementing caching with Redis for faster data retrieval.
- Optimizing database queries to reduce load time.

- Using Content Delivery Networks (CDN) to speed up music streaming.

DEPLOYMENT

CHAPTER 7

DEPLOYMENT

7.1 Hosting the Application

After development and testing, Rhythmic Tunes will be deployed on production servers.

Frontend Deployment

- Vercel / Netlify for hosting the React.js frontend.
- Google Play Store & Apple App Store for Flutter mobile app.

Backend Deployment

- AWS EC2 / Heroku / DigitalOcean for hosting the backend.
- NGINX or Apache as a web server.

Database & Storage Deployment

- MongoDB Atlas for managed database hosting.
- Firebase Firestore for real-time data storage.
- AWS S3 for scalable song storage.

7.2 Maintenance & Future Enhancements

1. AI-Based Recommendations

- Implementing machine learning models to suggest songs based on user preferences.
- Using collaborative filtering and deep learning algorithms.

2. Expanding to More Platforms

- Developing a desktop app (Electron.js or native Windows/macOS).
- Adding support for smart TVs and IoT devices.

3. Live Radio Streaming Support

- Partnering with online radio stations.
- Allowing users to stream live music channels.

FUTURE SCOPE

CHAPTER 8

FUTURE SCOPE

8.1 Summary of Project Implementation

- Rhythmic Tunes is a full-featured music streaming platform with authentication, playlists, music streaming, admin controls, and social features.
- Technologies Used:
  - Frontend: React.js (Web), Flutter (Mobile).
  - Backend: Node.js (Express) or Django (Python).
  - Database: MongoDB/Firebase Firestore.
  - Storage: AWS S3 / Firebase Storage.
- Key Features:
  - Streaming music with a media player.
  - Creating and managing playlists.
  - AI-based recommendations (future enhancement).
  - Social sharing and offline mode (premium users).

8.2 Challenges Faced During Development

1. Handling Large-Scale Streaming

- Optimizing music delivery using CDNs and caching techniques.

2. Securing User Data

- Implementing OAuth2.0 authentication and JWT tokens.
- Encrypting sensitive user data.

3. Payment Gateway Integration

- Managing Stripe/PayPal transactions securely.
- Ensuring smooth subscription management.

8.3 Future Enhancements & Scalability

- AI-Powered Recommendations: Improving user experience with personalized music suggestions.
- Music Licensing & Partnerships: Collaborating with artists and record labels.
- Integration with Smart Assistants: Enabling voice commands via Alexa, Google Assistant.
- Live Concert Streaming: Adding real-time concert broadcasts.

REQUIREMENTS

CHAPTER 9
REQUIREMENTS

9.1 Functional Requirements

- **User Authentication:** Sign up/login with email or social accounts.
- **Music Streaming:** Play, pause, and browse songs seamlessly.
- **Search & Filters:** Find songs by title, artist, or genre.
- **Playlist Management:** Create, edit, and delete custom playlists.
- **Favorites & Likes:** Users can like and save favorite songs.
- **Admin Panel:** Manage users, upload songs, and track trends.
- **Music Upload & Storage:** Securely store songs in Firebase/AWS S3.
- **Subscription & Payment (Optional):** Premium features like offline mode and ad-free streaming.
- **Social Features (Optional):** Share playlists, follow users, and interact.
- **Notifications:** Get updates on new releases and recommendations.

9.2 Non-Functional Requirements

- **Performance:** Fast and smooth streaming experience.
- **Security:** User authentication, encrypted data, and role-based access.
- **Scalability:** Support for a growing number of users and songs.
- **Usability:** Intuitive interface for web and mobile users.
- **Compatibility:** Works on different devices and operating systems.

9.3 Hardware & Software Requirements

- **Frontend:** React.js (Web), Flutter (Mobile).
- **Backend:** Node.js (Express.js) / Django (Python).
- **Database:** MongoDB / Firebase Firestore.
- **Storage:** AWS S3 / Firebase Storage.
- **Hosting:** Vercel/Netlify (Frontend), AWS/Heroku (Backend).

CONCLUSION

CONCLUSION

In conclusion the development of a **music streaming application** using **Node.js** has proven to be a robust and efficient approach, providing a scalable, high-performance, and feature-rich platform for delivering seamless audio streaming experiences. With Node.js' event-driven and non-blocking I/O model, the application efficiently handles multiple simultaneous requests, ensuring smooth playback and real-time interactions for users. The integration of on-demand streaming, high-quality audio playback, personalized playlists, search and discovery features, and user authentication mechanisms enhances the overall user experience, making it intuitive and engaging. Additionally, the use of **cloud storage, database management, and caching mechanisms** optimizes performance, reducing latency and ensuring uninterrupted streaming. Security considerations, such as **data encryption, secure payment gateways, and authentication protocols**, have been implemented to protect user data and transactions.

The deployment of the application on cloud-based infrastructure ensures high availability and scalability, allowing the system to accommodate a growing user base efficiently. Moreover, continuous maintenance and monitoring play a crucial role in identifying and resolving potential issues, ensuring that the platform remains stable and up-to-date.

The flexibility of **Node.js, along with modern front-end technologies, APIs, and third-party integrations**, provides ample opportunities for future enhancements, such as **AI-driven recommendations, social features, offline playback, and live streaming capabilities**. In conclusion, this **Node.js-based music streaming application** serves as a powerful and versatile solution, delivering high-quality music content to users while maintaining **scalability, performance, and security**. With evolving technology and market trends, the application has the potential to expand further, offering a more immersive and personalized listening experience for music enthusiasts worldwide.

APPENDIX

SOURCE CODE:

```
import React, { useEffect, useState } from "react";
import Home from "./page/home/Home";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import SearchResult from "./page/searchResult/SearchResult";
import PlaylistSongs from "./page/playlistSongs/PlaylistSongs";
import ScrollToTop from "./utils/ScrollToUp";
import PageNotFound from "./page/pageNotFound/PageNotFound";
import Trending from "./page/trending/Trending";
import Player from "./components/player/Player";
import { useSelector } from "react-redux";
import Explore from "./page/explore/Explore";
import Header from "./components/header/Header";
import Feedback from "./page/feedback/Feedback";
import About from "./page/about/About";
import OfflineBanner from "./components/offlineBanner/OfflineBanner";
import Footer from "./components/footer/Footer";
import RedirectToOrigin from "./utils/RedirectToOrigin";
import ImportedPlaylist from "./page/importedPlaylist/ImportedPlaylist";

function App() {
  const currentSong = useSelector(
    (state) => state.currentSongSlice.currentSongInfo
  );
  const { id } = currentSong;
```

```
// offline status
const [isOffline, setIsOffline] = useState(!navigator.onLine);

useEffect(() => {
  window.addEventListener("offline", (e) => {
    setIsOffline(true);
  });
  window.addEventListener("online", (e) => {
    setIsOffline(false);
  });
}, []);

const OnlineRoute = (PageRoute) => {
  return !isOffline ? <PageRoute /> : <OfflineBanner />;
};

return (
  <BrowserRouter>
    <RedirectToOrigin />
    <ScrollToTop />
    <Header />
    <Routes>
      <Route path="/" element={OnlineRoute(Home)} />
      <Route
```

```
          path="/:urlTitle/:playlistId"
          element={OnlineRoute(PlaylistSongs)}
        />
        <Route path="/search/:q" element={OnlineRoute(SearchResult)} />
        <Route path="/trending" element={OnlineRoute(Trending)} />
        <Route path="/explore" element={OnlineRoute(Explore)} />
        <Route
          path="/imported-playlist"
          element={OnlineRoute(ImportedPlaylist)}
        />
        <Route path="/feedback" element={<Feedback />} />
        <Route path="/about" element={<About />} />

        <Route path="*" element={<PageNotFound />} />
      </Routes>
      <Footer />
      {id ? <Player /> : null}
    </BrowserRouter>
  );
}

export default App;
```