Full Stack Development with MERN Project Documentation

1. Introduction

- **Project Title: Tune Trails** - A Modern Music Streaming Platform
- **Team Members:**

NAME	ROLE
Pari Agarwal	Full Stack Developer
Ojaswini Pradhan	Frontend Developer
Sakshi Chandra	UI/UX Designer
Priyanshi Katiyar	QA Engineer

2. Project Overview

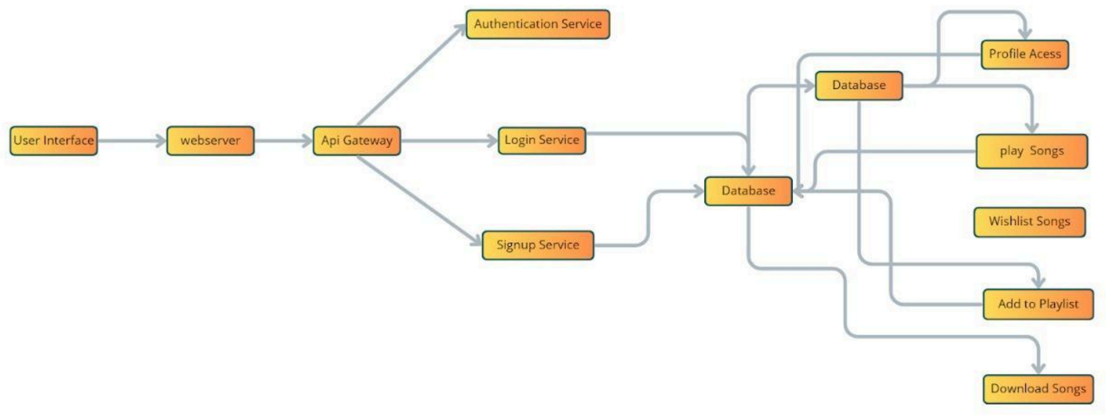
Purpose:

1. To build a Spotify-like music streaming app with:
2. Secure user authentication
3. Smooth audio playback
4. Personalized playlists
5. Cross-platform compatibility

Features:

FEATURE	DESCRIPTION
Social Login	Google/Facebook OAuth + JWT
Music Player	Play/pause/skip with progress bar
Playlists	Create, edit, delete playlists
Search	Find songs/artists instantly
Responsive UI	Mobile-first design with Tailwind CSS

3. Architecture

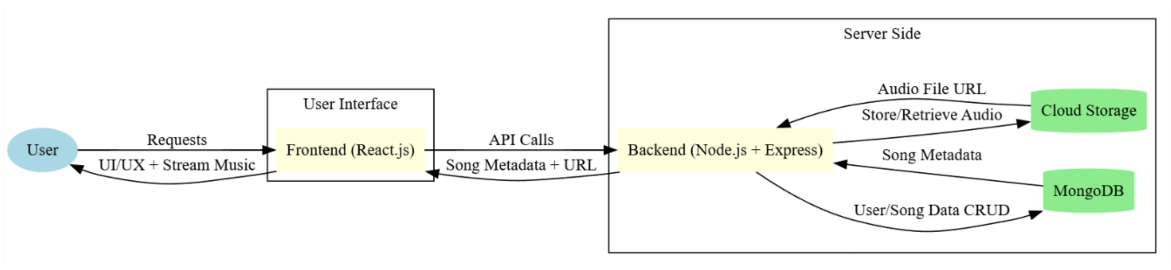


- **Frontend:**
- **Libraries**: react-router-dom, axios, react-icons
- **State Management**: Context API for global state (player, auth)
- **Styling**: Tailwind CSS + custom animations

- **Backend:**
- **RESTful API** with:
- JWT authentication middleware
- Rate limiting (express-rate-limit)
- Error handling wrappers

- **Database:**

```
// User Schema

{

  email: String,

  password: String, // Hashed

  playlists: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Playlist' }]

}



// Song Schema

{
  title: String,

  artist: String,

  duration: Number,

  filePath: String // S3 or local storage

}
```



4. Setup Instructions

- **Prerequisites:**
  1. Node.js v18+
  2. MongoDB Atlas account
  3. FFmpeg(for audio metadata extraction)
- **Installation:**

  1. Clone the repo:
     git clone https://github.com/agl724/MERN-Project
     cd MERN-Project
  2. Backend setup:
     cd backend
     npm install
     cp .env.example .env # Add your MongoDB_URI, JWT_SECRET

  3. Frontend setup:
     cd frontend
     npm install

5. Folder Structure

- **Client:**

```
frontend/
├── public/            # Static assets
│   ├── index.html
│   ├── favicon.ico
│   └── robots.txt
│
```

```
├── src/
│   ├── assets/            # Media files
│   │   ├── images/
│   │   └── fonts/
│   │
│   ├── components/          # Reusable UI components
│   │   ├── auth/
│   │   │   ├── LoginForm.jsx
│   │   │   └── RegisterForm.jsx
│   │   ├── common/
│   │   │   ├── Navbar.jsx
│   │   │   └── Footer.jsx
│   │   └── items/
│   │       ├── ItemCard.jsx
│   │       └── ItemList.jsx
│   │
│   ├── contexts/           # Global state management
│   │   ├── AuthContext.jsx
│   │   └── ItemContext.jsx
│   │
│   ├── pages/             # Route-based components
│   │   ├── HomePage.jsx
│   │   ├── Dashboard.jsx
│   │   └── NotFound.jsx
│   │
│   ├── services/           # API service calls
│   │   ├── authService.js
│   │   └── itemService.js
│   │
│   ├── styles/            # CSS/Tailwind files
│   │   ├── App.css
│   │   └── tailwind.config.js
│   │
│   ├── App.jsx            # Main component
│   ├── main.jsx           # React entry point
│   └── routes.jsx         # React Router setup
│
├── .eslintrc.json          # Linting rules
├── package.json           # Frontend dependencies
└── README.md              # Frontend docs
```

- **Server:**

```
backend/

├── config/              # Configuration files

│   ├── db.js            # MongoDB connection
│   └── jwtConfig.js        # JWT secrets
│
├── controllers/            # Business logic
│   ├── authController.js   # handleLogin, handleRegister
│   └── itemController.js   # CRUD operations
│
├── middleware/             # Custom middleware
│   ├── authMiddleware.js   # JWT verification
│   └── errorHandler.js     # Centralized error handling
│
├── models/                 # MongoDB schemas
│   ├── User.js             # User schema
│   └── Item.js             # Item schema
│
├── routes/                 # API endpoints
│   ├── authRoutes.js       # POST /api/auth/login
│   └── itemRoutes.js       # GET /api/items
│
├── uploads/                # File storage (if applicable)
│
├── utils/                  # Helper functions
│   ├── validation.js       # Input sanitization
│   └── logger.js           # Request logging
│
```

```
├── .env              # Environment variables
├── app.js            # Express server setup
├── package.json      # Backend dependencies
└── server.js         # Server entry point
```

1. **Client-Side Organization**:
   - Logical separation of components (UI), contexts (state), and services (API calls).
   - Route-based pages for better scalability.
2. **Server-Side Modularity**:
   - MVC pattern (models, controllers, routes).
   - Dedicated middleware for auth and error handling.
3. **Scalability**:
   - Easy to add new features (e.g., /services/paymentService.js).
   - Clear separation of concerns (e.g., validation.js for input checks).

6. Running the Application

- **Frontend:** cd frontend && npm start # Port 3000

Backend: cd frontend && npm start # Port 3000

7. API Documentation

Endpoint	Method	Body (Example)	Response (200)
/api/auth/login	POST	{ email: "user@demo.com" }	{ token: "jwt_token" }
/api/songs	GET	-	[{ id: 1, title: "Song 1"}]

Example Request:

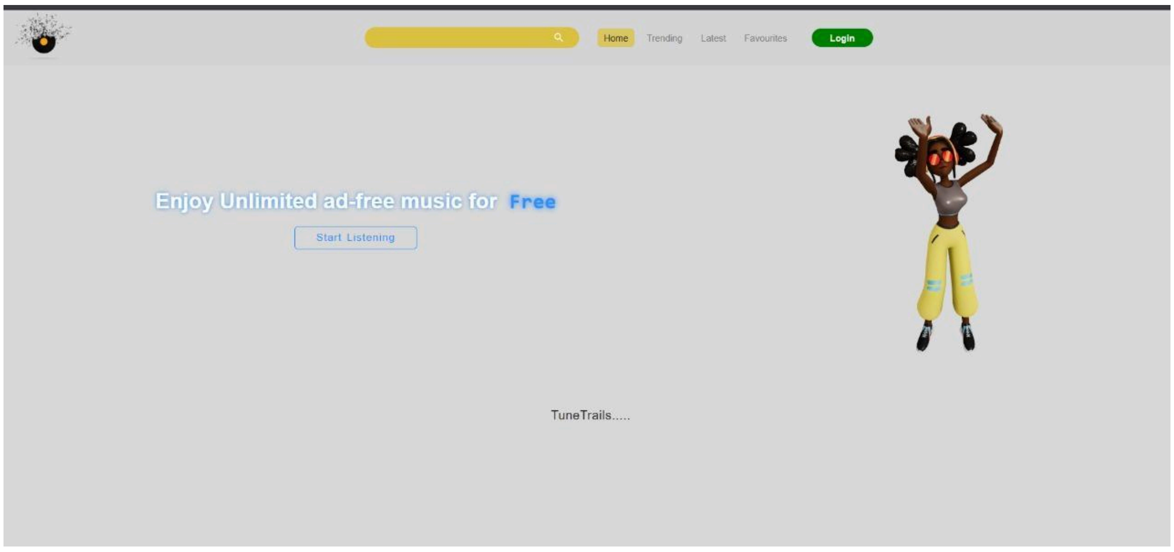
POST /api/users/login

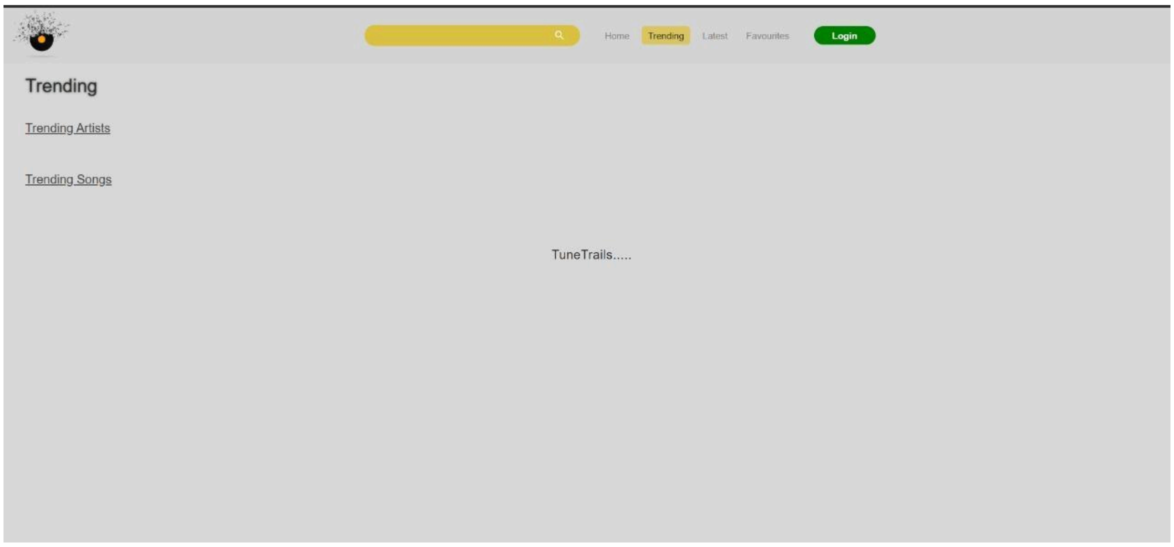
{ "username": "test", "password": "123" }

8. Authentication

- **Flow**:
    1. User logs in → Server validates credentials → Returns JWT
    2. Token stored in localStorage → Attached to API requests via axios interceptor
- **Security**:
    - Passwords hashed with bcryptjs
    - JWT expires in **24h**

9. User Interface



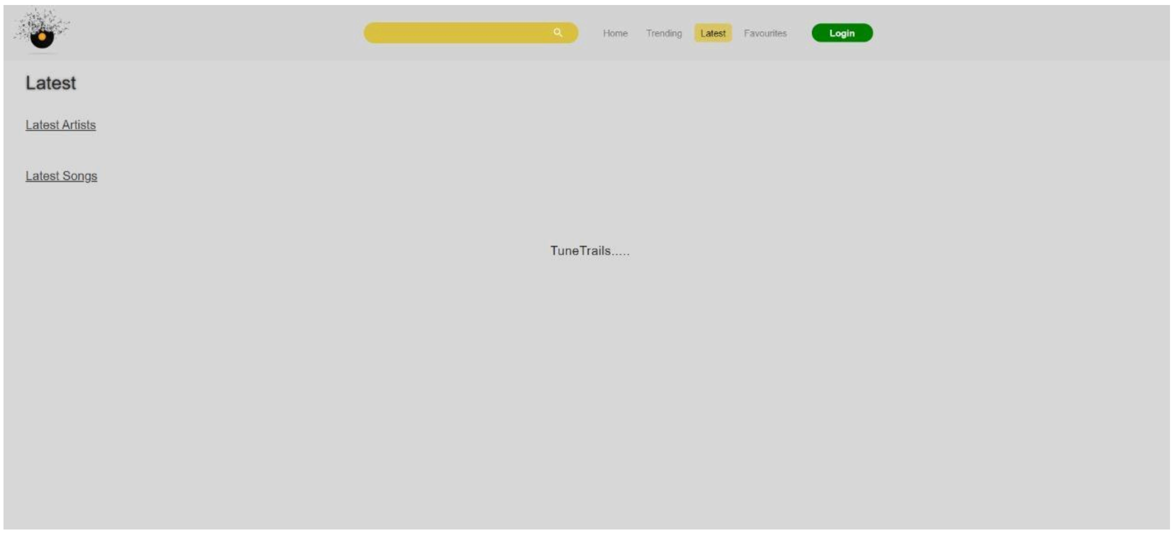


Trending

Trending Artists

Trending Songs

TuneTrails.....

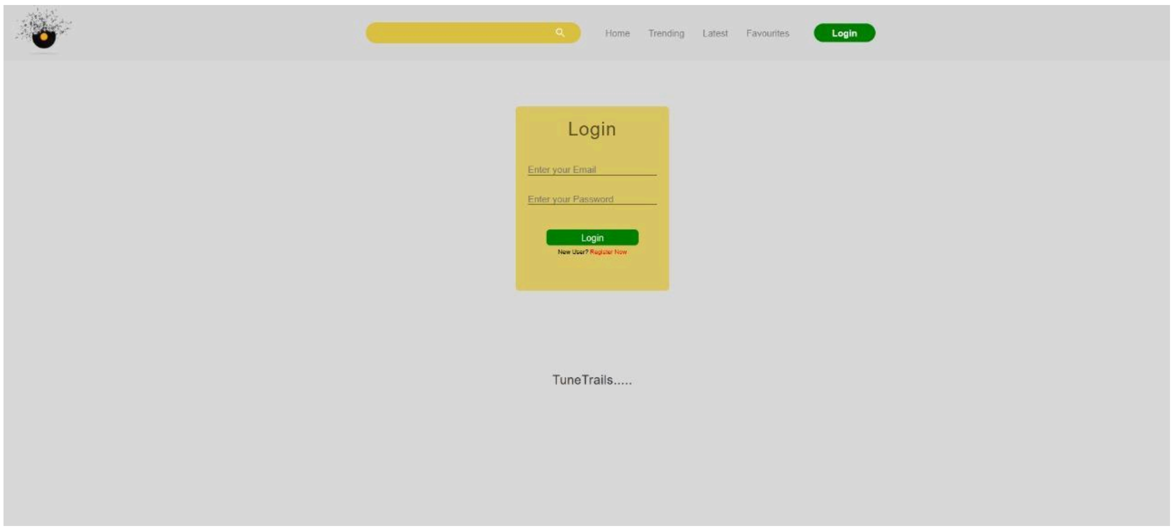


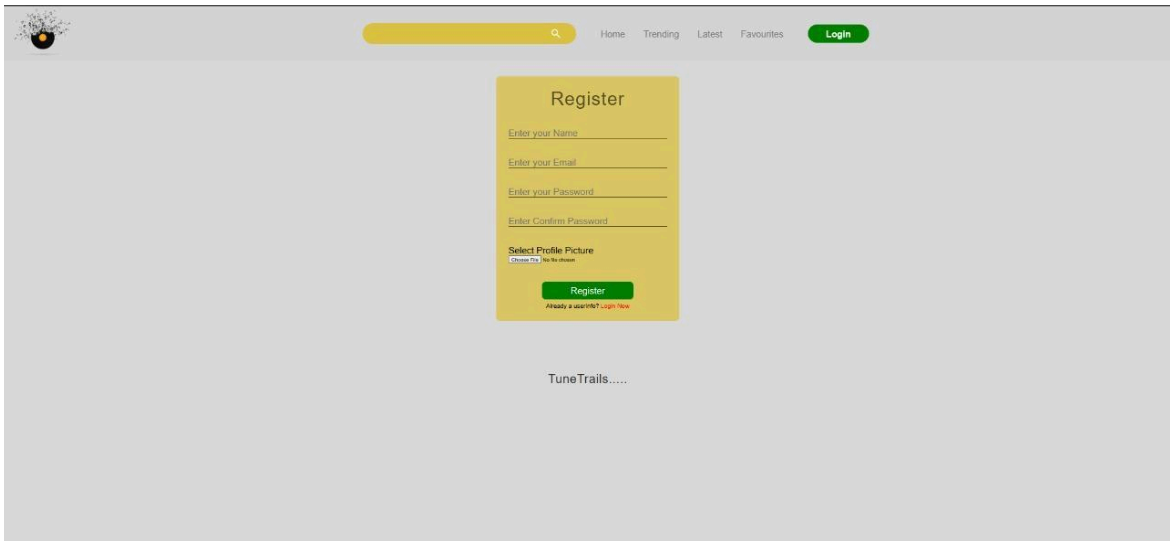
Latest

Latest Artists

Latest Songs

TuneTrails.....





10. Testing

A robust testing strategy is essential to ensure the reliability, security, and performance of the MERN stack application. Below is a detailed breakdown of the testing approach:

Testing Pyramid

We follow the **Testing Pyramid** methodology to ensure comprehensive coverage:

1. **Unit Testing** (70%)
    - Tests individual functions/components in isolation.
    - **Tools**:
        - **Jest** (JavaScript testing framework)

        - **React Testing Library** (for React components)
        - **Mocha/Chai** (for Node.js backend)

2. Intergration Testing(20%):
   Tests interactions between modules
   Tools:
      a. Supertest
      b. Jest (with mocking)
3. End to End Testing(10%):
   Simulates real user workflows
   Tools:
   a. Cypress
   b. Selenium

11. Demo :

https://drive.google.com/file/d/1AVa81YmacTps0M1Samr-6JEFk4KMrg8v/view?usp=drive_link

12. Known Issues

Issue	Workaround
Skipping songs too fast crashes player	Throttle API calls
Google OAuth fails on Safari	Use Firefox/Chrome

13. Future Enhancements

- Outline potential future features or improvements that could be made to the project.

To ensure **Tune Trails** remains competitive and feature-rich, here are detailed future enhancements:

---

1. Advanced State Management with Redux

Problem: As the app scales, managing state with **Context API** becomes complex.
Solution:

- Migrate to **Redux Toolkit** (RTK) for predictable state management.

- **Key Benefits**:
  - Centralized state for **player controls**, **user preferences**, and **playlists**.
  - Middleware support (e.g., **Redux Thunk** for async API calls).
  - Time-travel debugging with **Redux DevTools**.

Implementation Plan:

javascript

Copy

```
// Example: Redux slice for player state
const playerSlice = createSlice({
  name: 'player',
  initialState: { currentSong: null, isPlaying: false },
  reducers: {
    playSong: (state, action) => {
      state.currentSong = action.payload;
      state.isPlaying = true;
    },
    pauseSong: (state) => {
      state.isPlaying = false;
    },
  },
});
```

---

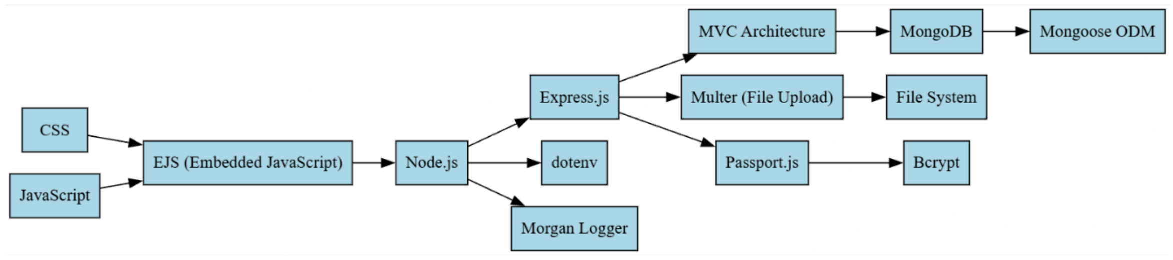
2. Payment Gateway Integration

Problem: Monetization requires secure payment processing.
Solution:

- Integrate **Stripe** or **Razorpay** for subscriptions/one-time purchases.

- **Key Features**:
    - **Monthly/Annual Plans** (e.g., "Premium Tier: $9.99/month").
    - **Trial Periods** (e.g., 30-day free trial).
    - **Webhooks** to handle payment failures/subscription renewals.

Tech Stack:



- **Frontend**: Stripe Elements (PCI-compliant UI components).
- **Backend**:

javascript

Copy

```
// Node.js route to create a payment intent
app.post('/api/payment/create-intent', async (req, res) => {
  const paymentIntent = await stripe.paymentIntents.create({
    amount: 999, // $9.99
    currency: 'usd',
  });
  res.send({ clientSecret: paymentIntent.client_secret });
});
```

---

3. Real-Time Features with WebSockets

Problem: Users expect live interactions (e.g., collaborative playlists).
Solution:

- Use **Socket.io** to enable:

    - **Live Lyrics Sync**: Display timed lyrics for songs.
    - **Group Listening**: Friends can listen to the same song simultaneously.

Implementation:

javascript

Copy

```
// Socket.io server setup
io.on('connection', (socket) => {
  socket.on('join-room', (roomId) => {
    socket.join(roomId);
    socket.to(roomId).emit('user-joined', socket.id);
  });
});
```

---

4. Offline Mode with Service Workers

Problem: Users lose access to music without internet.
Solution:

- Cache songs/playlists using **Workbox** (Google's PWA library).
- **Steps**:
    1. Cache API responses (e.g., GET /api/songs).
    2. Store audio files in **IndexedDB** for playback offline.

---

5. Advanced Analytics Dashboard

Problem: Lack of insights into user behavior.
Solution:

- **Tools**:
    - **Mixpanel**/**Amplitude** for tracking:
        - Most-played songs.

        - User retention rates.
    - **Custom Admin Panel** (React + Chart.js) to visualize data.

---

6. AI-Powered Recommendations

Problem: Static playlists reduce engagement.
Solution:

- Use **TensorFlow.js** or a third-party API (Spotify's recommendation algorithm) to:
    - Suggest songs based on listening history.
    - Generate dynamic playlists (e.g., "Your Morning Coffee Mix").

---

7. Cross-Platform Expansion

Problem: Mobile users need dedicated apps.
Solution:

- **React Native** for iOS/Android apps (reuse 80% of React code).
- **Electron** for desktop apps (Windows/macOS).

---

8. Accessibility Improvements

Problem: App isn't fully accessible.
Solution:

- **WCAG Compliance**:
    - Keyboard navigation for player controls.
    - Screen reader support (ARIA labels).

---

9. Microservices Architecture

Problem: Monolithic backend slows down feature development.
Solution:

- Split into microservices:

    - **User Service** (Auth/profile).
    - **Payment Service** (Stripe integration).
    - **Recommendation Service** (AI/ML).
- **Tools**: Docker, Kubernetes, gRPC.

---

10. Community Features

Problem: Lack of social engagement.
Solution:

- **User Profiles**: Bios, follower counts.
- **Shared Playlists**: Users can collaborate on playlists.
- **Live Chat**: Discuss songs in real-time.

Prioritization:

1. **Redux Migration** (High impact, low effort).
2. **Payment Gateway** (Revenue-critical).
3. **Offline Mode** (User retention).