DV200 Progress Milestone Check

Mwape Kurete 231115

Open Window, School of Fundamentals

DV200

Table of Contents

Functional Requirements Checklist

- User Authentication:
    - Database Models and Routes Created
    - Testing endpoints for user authentication
    - Calling endpoints in my frontend
    - Completing login, sign-up cycle with JWT and user sessions in place
- CRUD in Backend
    - Create:
        - **User Registration**: In the `auth.js` route, users can register by creating an account. We also handle **password hashing** with bcrypt to ensure secure storage of passwords.
        - **Reviews**: Users can submit reviews for albums, and those reviews are stored in the MongoDB database. This involves creating a new review document with fields like `content`, `rating`, and `album`.
        - **Favourites**: Users can add albums to their list of favourites, which is stored under their user profile in MongoDB.
        - **Album:** When a new album is searched by users, the results are stored with relevant fields. Making search's happen faster over time
    - Read:
        - **Search Functionality**: The backend retrieves data from the **Spotify API** using the `searchAlbums` function, and it returns the results to the front end, enabling users to search for albums to review.
        - **Recommendations**: Users can receive personalised recommendations based on their favourite albums or listening histories. This involves querying MongoDB for their preferences and then fetching data from external APIs like **Last.fm, Spotify** or **TasteDive** to get similar albums or artists.

- **Discover:** Users can discover similar artists and their top 5 albums through the Discover page. This involves users searching for an album, and rekeni's backend performing content-based filtering where the search results are passed through, **Last.FM, Spotify,** and **TasteDive's API** similarity endpoints. These results are then filtered removing any duplicates and including more unique entries. Users can then interact with the returned results.

    - Put:
        - **Updating Reviews**: Users can update their reviews. This involves sending a PUT request to update the `content` and `rating` fields in the **Review** model.
        - **Updating User Favourites**: If users want to manage their favourite albums, they can update their list of favourites stored in MongoDB.
    - Delete:
        - **Deleting Reviews**: Users can delete their reviews, which removes the review document from MongoDB.
        - **Removing Favorites**: Users can remove albums from their favourites list, ensuring that data is kept up-to-date based on their preferences.

- API integration
    - Setting up the Services Folder in the backend
        - lastfm.js
            - getSimilarArtists: this function calls similar artists based on a user's query input
            - geetSearchedAlbum: this function fetches similar albums based on a user's query input
        - Spotify

                - searchSpotifyAlbums: this function searches Spotify's API for albums based on the user's query input
                - searchSpotifyArtist: this function searches Spotify API returning artists based on the user's query input
                - getTopAlbumsByArtistsSpotify: this function fetches an artist's top albums returning their top album
                - getNewReleases: this function fetches recent releases from Spotify's API
            - Tastedive
                - getSimilarMusicthis This function returns a list of artists similar to the user's initial search. The results of this are then passed to another service file function to build out the recommendation systems (discover and recommendations)
            - RecommendationServcice
                - This service file runs functions to fetch similar artists and recommendations based on either an artist name or an album title. This service is then used within the routes folder to drive my recommendation and discover endpoints
    - Creating routes to handle external API calls
- Frontend Functionality
    - Routing working
    - Basic validation set-up
    - Connections to the backend made
    - Calling endpoints in from the backend and using dynamic loading to load content
- Backend Functionality
    - Set up Models for MongoDB
    - Create routes

    - Create a service folder to handle external API calls
    - Test all routes and endpoints
- Smart Recommendation system
    - Backend set up to handle API iterations
    - User history to inform smart recommendations →User interactions to inform smart recommendations
- Overall frontend
    - Styling on all pages
    - Responsiveness across all pages

System Documentation:

Technical Architecture:

- The system is built using the **MERN stack**:
    - **MongoDB**: NoSQL database is used to store user data, albums, reviews, and user interactions.
    - **Express.js**: Backend framework that manages the server and API routes.
    - **React**: Frontend library for building the user interface and handling dynamic user interactions.
    - **Node.js**: Server-side JavaScript runtime that handles requests and connects the frontend to the backend.
- **Interaction**:
    - The **frontend (React)** sends HTTP requests to the **backend (Express/Node)** through **API endpoints**.
    - The **backend** interacts with **MongoDB** to perform CRUD operations and fetches external data using APIs like Last.fm and iTunes for music-related services.

2. Database Schema:

- **MongoDB** stores user data, album metadata, reviews, and user interactions (like favourites and listening history).
- **Database Structure**:
    - **User Model**: Stores user profiles with fields for username, email, password (hashed), and relationships to reviews and favourites.
    - **Album Model**: Stores metadata for albums (title, artist, release date, artwork, etc.) retrieved from the iTunes API.
    - **Review Model**: Stores user reviews, which are linked to the specific albums reviewed.

- **Relationships**:
    - A **User** can have many **favourites** (many-to-many relationship with albums).
    - A **User** can submit many **reviews**, and each **Album** can have many **reviews** (one-to-many relationship).

3. API Endpoints:

- **Authentication** (`/api/auth`):
    - **POST** `/register`: Registers a new user.
    - **POST** `/login`: Authenticates a user and returns a token.
- **Albums and Search** (`/api/search`):
    - **GET** `/search`: Searches for albums using the iTunes API.
- **Favourites** (`/api/favourites`):
    - **POST** `/favourites/:albumId`: Adds an album to the user's favourites.
    - **DELETE** `/favourites/:albumId`: Removes an album from the user's favourites.
- **Reviews** (`/api/reviews`):
    - **POST** `/reviews`: Submits a review for an album.
    - **PUT** `/reviews/:reviewId`: Updates an existing review.
    - **DELETE** `/reviews/:reviewId`: Deletes a review.
    - **READ** `/reviews/:reviewId:` fetches all reviews.
- **Recommendations** (`/api/recommendations`):
    - **GET** `/recommendations/:userId`: Fetches personalised album recommendations for the user, using data from external APIs and user interactions.
- **Discover** (`/api/recommendations`):

- **GET** `/discover:` Searches for an artist by the query passed and then returns a list of similar artists and their top 5 albums

4. User Interface (UI) Design:

- The user interface is designed to be responsive across devices (desktop, tablet, mobile).
- **Screens/Features**:
  - **Sign-Up/Login Pages**: Allow users to register and sign in securely.
  - **Search Page**: Users can search for albums to review.
  - **New Page**: Displays recommended albums based on user preferences.
  - **Review Section**: Allows users to read, submit, update, or delete reviews for specific albums.
  - **Profile Page**: Displays albums the user has marked as favourites, reviews they have left, and their username and bio.
- The UI is built using **React** and styled with **React-Bootstrap** for a clean, modern, and mobile-friendly layout.

5. Instructions for Running the Application:

- **Prerequisites**: Ensure Node.js, npm (Node package manager), and MongoDB are installed on your system.
- **Steps to Set Up**:
  - Clone the repository: `git clone <repo_url>`.
  - Navigate into the project directory: `cd <project_folder>`.
  - Install backend dependencies: `npm install` (in the root folder).
  - Install frontend dependencies: `cd client` then `npm install`.
  - Create a `.env` file in the root folder with environment variables like `MONGO_URI`, API keys for Last.fm, iTunes, and TasteDive.

- **Running the Application**:
    - Start MongoDB server: `mongod`.
    - Start the backend server: `npm run dev` (in the root folder).
    - Start the frontend server: `npm start` (inside the `client` folder).
- **Accessing the Application**:
    - Visit `http://localhost:3000` to view the frontend, and `http://localhost:5000` for backend operations.

Login Credentials:

As it currently stands Rekeni does not have an admin login. Users are free to create accounts and login as they please.

However, for testing my application functionality I made use of the following credentials:

Username: Tom

Email: tom@mail.com

Password: berryTom@12

Problem Statement

Over the past year, music streaming platforms have leaned heavily on AI-driven recommendations to help users discover new music and shape their listening habits. While this has brought efficiency, many users feel the recommendations lack accuracy and don't align with their personal tastes. Instead of introducing them to fresh, diverse music, they're often served the same mainstream artists and tracks they've already encountered.

This issue affects both music lovers and emerging artists. When recommendation algorithms are limited or biased, they contribute to a homogenised music landscape, limiting exposure for lesser-known talent. For many, music is deeply personal, so discovering new and exciting sounds that match individual tastes is essential.

Rekeni steps in to fill this gap by providing a more personalised, user-driven recommendation experience. By combining data from users' listening habits, favourites, and reviews, Rekeni offers recommendations that aren't only based on algorithms but are shaped by user interaction and preferences. Integrating APIs like Last.fm and TasteDive, Rekeni delivers fresh music suggestions that capture both popular trends and new genres.