EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS
DEPT. OF Programming Languages
and Compilers

MoodFlicks: Personalized Movie Recommendation System

Supervisor: ***Dr. Alwahab Dhulfiqar Zoltan***
Assistant Professor

Author: ***Deepak Kumar Upadhayay***
B.Sc Computer Science

Budapest , 2024

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:

Student's Name: *Upadhayay Deepak Kumar*
Student's Neptun code: *DD79DY*

Educational Information:

Training programme: *Computer Science BSc*

I have an internal supervisor

Internal Supervisor's Name: ***Dr Alwahab Dhulfiqar Zoltan***
Supervisor's Home Institution: ***Department of Programming Languages and Compilers***
Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***
Supervisor's Position and Degree: ***Assistant Professor***

Thesis Title: *MoodFlicks: Personalized Movie Recommendation System*

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

"MoodFlicks" is a personalized movie recommendation system designed to provide users with tailored movie suggestions based on their mood. This project utilizes Python, Flask, and Large Language Models (LLMs) to create an intuitive and interactive platform for users to discover movies that match their current emotional state.

The system employs natural language processing techniques to understand user inputs regarding their mood. By analyzing the sentiment and tone of the user's input, MoodFlicks categorizes the user's mood into various emotional states such as happy, sad, excited, or nostalgic.

Once the user's mood is identified, MoodFlicks leverages its database of movies along with LLMs to generate movie recommendations that align with the user's emotional state. These recommendations are not solely based on genre or popularity but also on the thematic elements, atmosphere, and emotional resonance of the movies.

The Flask framework is utilized to create a user-friendly web interface where users can input their mood and receive personalized movie recommendations in real-time. Users can also explore additional details about recommended movies such as plot summaries, cast, and trailers.

MoodFlicks aims to enhance the movie-watching experience by providing users with tailored recommendations that resonate with their current feelings, ultimately creating a more engaging and satisfying viewing experience.

Budapest, 2024. 05. 14.

Contents

Chapter 1

1. Introduction

1.1 Motivation

In today's digital age, the entertainment industry is saturated with an overwhelming array of movies, making it increasingly difficult for individuals to find films that resonate with their preferences and moods. Traditional search methods, such as genre filters or user reviews, often fail to provide tailored recommendations, leaving users feeling frustrated and indecisive. This challenge underscores the need for a personalized solution that bridges the gap between user preferences and the vast library of available content.

The motivation for this project stems from the desire to create a seamless movie discovery experience. By leveraging modern technologies such as machine learning, Flask for web development, and PostgreSQL for data management, this system aims to offer a user-centric platform. Additionally, integrating real-time notifications, mood-based suggestions, and advanced filtering options reflects the growing demand for intelligent and adaptive systems in the entertainment domain.

1.2 Goals

The primary goal of this project is to develop a robust **Movie Recommendation System** that simplifies the process of finding suitable movies for users. By combining advanced algorithms with an intuitive interface, this system aspires to:

- Provide personalized movie recommendations based on user preferences, genres, and viewing history.
- Enable mood-based suggestions, allowing users to discover movies aligned with their emotional state.
- Offer features like user authentication, real-time notifications, and dynamic search filters to enhance usability.
- Support analytics and reporting to understand user behavior and improve recommendation accuracy.
- Ensure scalability and reliability by utilizing technologies like Flask, PostgreSQL, and cloud hosting solutions.

1.3 Scope of the Project

The **Movie Recommendation System** is designed to address the growing need for personalized entertainment solutions. Its scope encompasses the following key aspects:

1. **Personalized Recommendations**
   The system leverages advanced algorithms and user data to suggest movies tailored to individual preferences, including genres, ratings, and viewing history.
2. **Mood-Based Suggestions**
   By incorporating mood analysis, users can discover movies that align with their current emotional state, enhancing the entertainment experience.
3. **User Authentication and Management**
   Secure user registration and login functionalities ensure personalized recommendations and a safe platform for users to save their preferences.
4. **Dynamic Search and Filtering**
   Users can explore movies using powerful search and filter options, enabling them to find content that meets their specific criteria, such as release year, genre, or popularity.
5. **Real-Time Notifications**
   The system notifies users about updates, new releases, or personalized suggestions, ensuring they stay engaged with the platform.
6. **Analytics and Reporting**
   Administrators can utilize analytical insights to monitor user behavior, measure system performance, and refine recommendation algorithms for better results.
7. **Scalability and Cloud Hosting**
   By deploying the system on cloud platforms, it ensures scalability to handle a growing user base and supports seamless integration with other services or APIs, such as TMDb.
8. **Interactive and Responsive Interface**
   A user-friendly, responsive web interface is designed to provide an intuitive and engaging experience, accessible across devices.

Chapter 2

2. User Documentation

2.1 Introduction

Welcome to the Movie Recommendation System called as ***Mood Flicks***! This web application is designed to provide users with personalized movie suggestions based on preferences, mood, and genre. The system offers a seamless and interactive experience, ensuring that users can easily discover movies tailored to their tastes.
This movie recommendation system is designed to simplify the process of discovering movies tailored to your mood, genre preferences, or viewing history. By leveraging a vast database of movies, the application provides users with personalized and accurate recommendations.

2.2 Used Methods and Tools

The system integrates a combination of advanced technologies, methodologies, and collaborative tools to provide accurate and user-friendly movie recommendations:

1. **Flask Framework**: A lightweight Python web framework used for backend development, handling HTTP requests, routing, and serving dynamic content.

2. **PostgreSQL**: A robust relational database system to store and manage user data, movie metadata, and recommendation history efficiently.

3. **TMDb API**: Used to fetch movie details, posters, and metadata, providing up-to-date and rich content for recommendations.

4. **OpenAI API**: Utilized for natural language processing and machine learning to enhance mood-based and genre-based movie recommendations by analyzing user preferences and sentiments.

5. **Docker**: Ensures consistent application behavior across environments through containerization, streamlining development, testing, and deployment.

6. **Git**: A version control system employed to manage code changes, enabling collaboration, maintaining history, and ensuring seamless team coordination throughout the development lifecycle.

7. **Frontend Technologies**: HTML, CSS, and JavaScript provide a visually engaging and responsive user interface for seamless interaction.

8. **Recommendation Algorithms**: Incorporates collaborative filtering, content-based filtering, and NLP-based approaches to deliver personalized movie suggestions.

By combining these tools and techniques, the system ensures a user-friendly experience while maintaining scalability, efficiency, and ease of collaboration during development.

2.3 User Guide

Hardware Requirements
The movie recommendation web application can run on systems with the following hardware configurations as show in Table 1 below,

Component	Minimum	Recommended
Processor	2.0 GHz x86- or x64-bit dual-core	3.5 GHz or faster 64-bit quad-core
Memory	4 GB RAM	8 GB RAM or more
Display	Super VGA (1024 x 768 resolution)	Full HD (1920 x 1080 resolution)

Table 1

Running the application on computers that don't meet the minimum requirements may lead to performance issues. Acceptable performance may be achieved on systems with alternative hardware configurations, such as recent quad-core processors with lower clock speeds and more RAM.

Network Requirements
The application is optimized for networks with the following characteristics:

- **Bandwidth**: 50 KBps (400 kbps) or higher.
- **Latency**: Under 150 ms.

Supported Web Browsers
The movie recommendation system supports the following web browsers on specified operating systems:

- **Microsoft Edge** (latest version) on Windows 10, 11, or 8.1.
- **Mozilla Firefox** (latest version) on Windows 10, 11, or 8.1.
- **Google Chrome** (latest version) on Windows 10, 11, and the two latest versions of Mac OS.
- **Apple Safari** (latest version) on the two latest versions of Mac OS.

2.4 User Guide: Setting Up the Mood Flicks Locally

This guide will help you set up and run the Movie Recommendation System on your local machine. Follow these steps carefully to get the project up and running.

1. Prerequisites

Before you begin, ensure you have the following installed on your machine:

- **Git**: To clone the repository.
- **Python (3.9 to 3.11)**: Required to run the application.
- **Docker Desktop**: For running PostgreSQL in a container (optional, but recommended).
- **API Keys**: TMDb and OpenAI API keys to integrate movie data and recommendations.

1.1. Installing Git

If you don't have Git installed, download it from Git's official website. Follow the installation instructions based on your OS.

1.2. Installing Docker

1. Download and install Docker Desktop from Docker's official site.
2. After installation, open Docker Desktop and ensure it's running. This will allow you to use Docker containers for your database (PostgreSQL).

1.3. Installing Python

1. Download and install Python 3.9+ from the official Python website.
2. After installation, verify by opening a terminal/command prompt and running:

bash
Copy code

```
python --version
```

This should return Python 3.9+.

2. Setting Up the Project Locally

2.1. Cloning the Repository

1. Open your terminal (Command Prompt, Git Bash, or Terminal on Mac).
2. Navigate to the directory where you want to clone the project. For example:

bash
Copy code

```
cd ~/projects
```

3. Clone the repository using Git:

bash
Copy code

```
git clone https://github.com/inspironman/Thesis.git
```

4. After cloning, navigate into the project directory:
bash
Copy code
```
cd movie-recommendation-system
```

2.2. Installing Dependencies

1. Inside the project directory, ensure you have Python 3.9+ installed.
2. Create a virtual environment (optional but recommended):

bash
Copy code
```
python -m venv venv
```

3. Activate the virtual environment:
   - **Windows**:

bash
Copy code
```
venv\Scripts\activate
```

   - **Mac/Linux**:

bash
Copy code
```
source venv/bin/activate
```

4. Install the required dependencies by running:

bash
Copy code
```
pip install -r requirements.txt
```

2.3. Obtaining API Keys

To make the recommendation system functional, you'll need API keys for **TMDb** and **OpenAI**.

2.3.1. TMDb API Key

1. Go to TMDb's website.
2. Create an account if you don't have one.
3. Once logged in, go to your **Account Settings** and then **API**.
4. Follow the instructions to create an API key.
5. Copy the generated API key.

2.3.2. OpenAI API Key

1. Go to OpenAI's website.
2. Create an account or log in.
3. Navigate to **API Keys** in your account settings and generate a new API key.
4. Copy the generated API key.

2.4. Configure API Keys in the Project

1. Inside the project folder, create a .env file (if not already present).
2. Add the following lines to the .env file with your actual API keys:

makefile
Copy code

```
TMDB_API_KEY=your_tmdb_api_key_here
OPENAI_API_KEY=your_openai_api_key_here
```

2.5. Docker Setup (Optional for Database)

You can run PostgreSQL using Docker. This will ensure the database is containerized and easily accessible.

1. **Install Docker Desktop** (if not done already).
2. Pull the PostgreSQL Docker image:

bash
Copy code

```
docker pull postgres:latest
```

3. Create and run the PostgreSQL container:

bash
Copy code

```
docker run --name postgres-container -e
POSTGRES_PASSWORD=yourpassword -d postgres
```

This will start PostgreSQL with the default password yourpassword.

4. Connect to the container:

bash
Copy code

```
docker exec -it postgres-container psql -U postgres
```

5. Once inside PostgreSQL, create the necessary database for the project:

sql
Copy code

```
CREATE DATABASE moviedb;
```

6. Update the database connection string in the project configuration with the appropriate details (e.g., host, user, password, and database name).

3. Running the Application

3.1. Starting the Server

1. In your terminal, ensure you are in the project directory.
2. Run the following command to start the server:

bash
Copy code

```
python index.py
```

3. The server should now be running on http://localhost:8080. Open this URL in your web browser to access the Movie Recommendation System.

3.2. Testing the Application

Once the server is running, you can:

- Search for movies.
- Get movie recommendations based on mood, genre, or content.
- View detailed movie information fetched from TMDb and OpenAI.

4. Troubleshooting

4.1. Common Errors

- **Missing Dependencies**: If you get errors like ModuleNotFoundError, ensure that all dependencies are installed correctly using pip install -r requirements.txt.
- **API Key Errors**: If the application doesn't load or returns errors about invalid API keys, make sure your TMDb and OpenAI keys are correctly added in the .env file.

4.2. Docker Issues

- **Docker Container Not Starting**: If the PostgreSQL container doesn't start, try restarting Docker Desktop and re-running the container creation commands.
- **Database Connection**: Double-check that your connection details (host, password) in the project configuration match the settings of your PostgreSQL container.

---

5. Additional Resources

- **TMDb Documentation**: https://www.themoviedb.org/documentation/api
- **OpenAI API Documentation**: https://beta.openai.com/docs/
- **Docker Documentation**: https://docs.docker.com/.
- **Python Documentation**: https://docs.python.org/

2.5 Features of the Movie Recommendation System

The Movie Recommendation System provides several ways to discover movies based on various preferences and moods. The app includes the following key features:

1. Genre-Based Recommendations

- **What It Does**: This feature allows users to discover movies based on selected genres such as Action, Comedy, Drama, etc. As shown in fig. 1 and 2.
- **How to Use It**:
    1. Navigate to the "Genre-Based Recommendations" section.
    2. Select one or more genres.
    3. Type the number of recommendations you want (1-10).
    4. Click the "Get Recommendations" button to receive a list of movies that match your selected genres.
    5. Here the cards are flip able and users can see the movie details by navigating to the particular card and also watch the trailer of the movie.

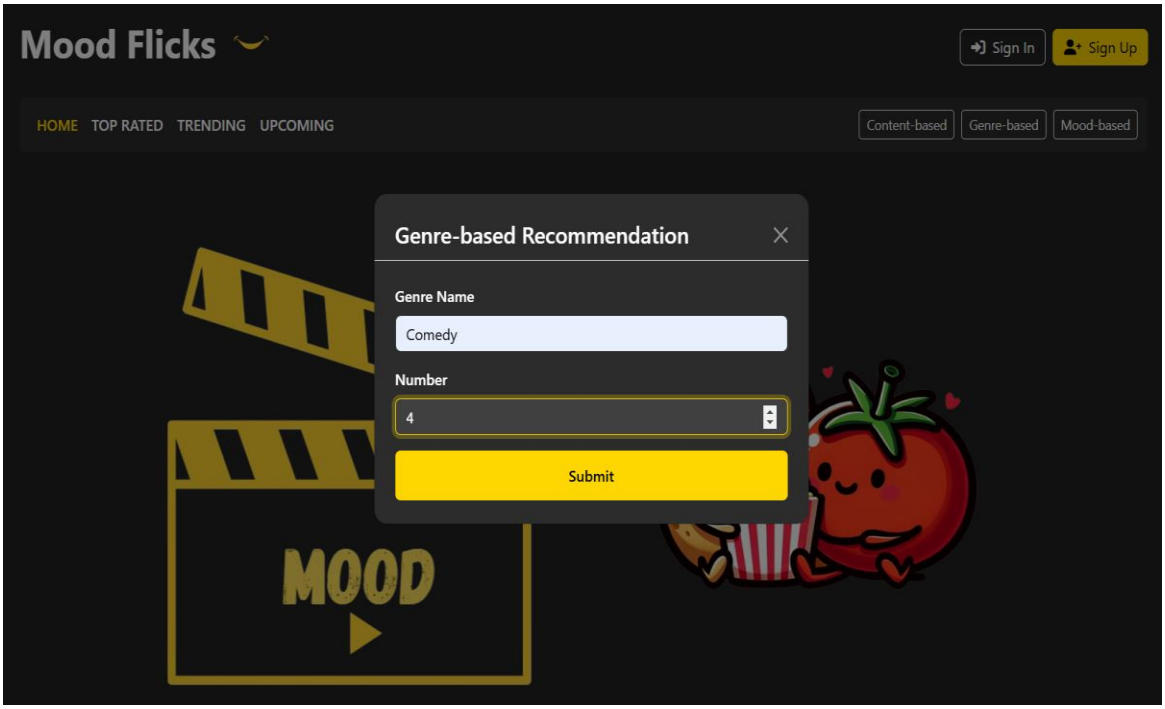

Figure 1 Genre-Based Search

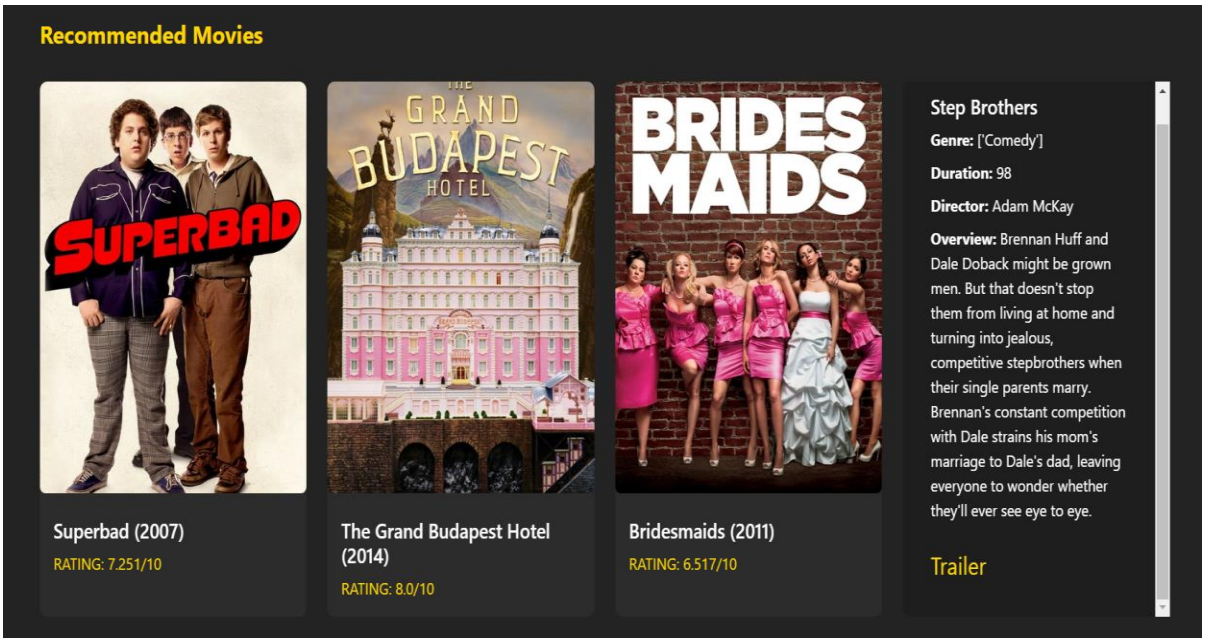

Figure 2 Genre-Based Result

2. Mood-Based Recommendations

- **What It Does**: This feature suggests movies based on the user's current mood (e.g., happy, sad, adventurous). As shown in fig. 3 and 4.
- **How to Use It**:
    1. Go to the "Mood-Based Recommendations" section.
    2. Select your mood from the list (e.g., "Happy," "Romantic," "Action").
    3. Type the number of recommendations you want (1-10).
    4. Click "Get Recommendations" to receive a list of movies that fit your mood.
    5. Here the cards are flip able and users can see the movie details by navigating to the particular card and also watch the trailer of the movie.

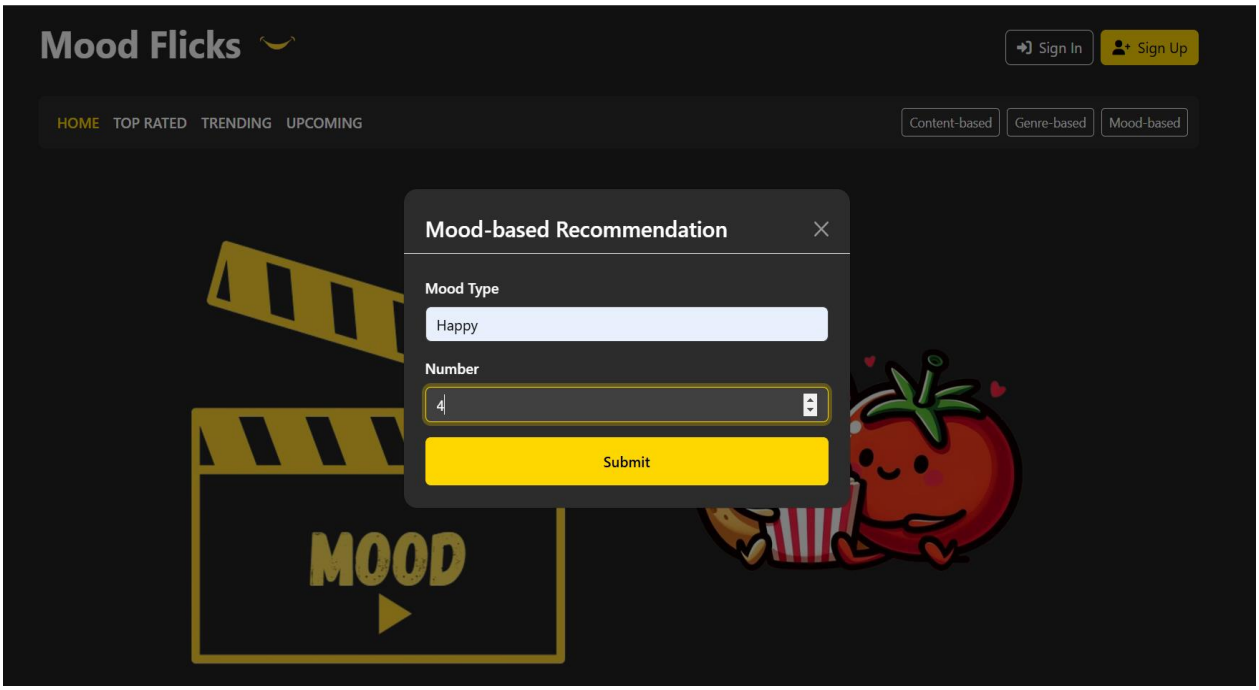

Figure 3 Mood-based Search

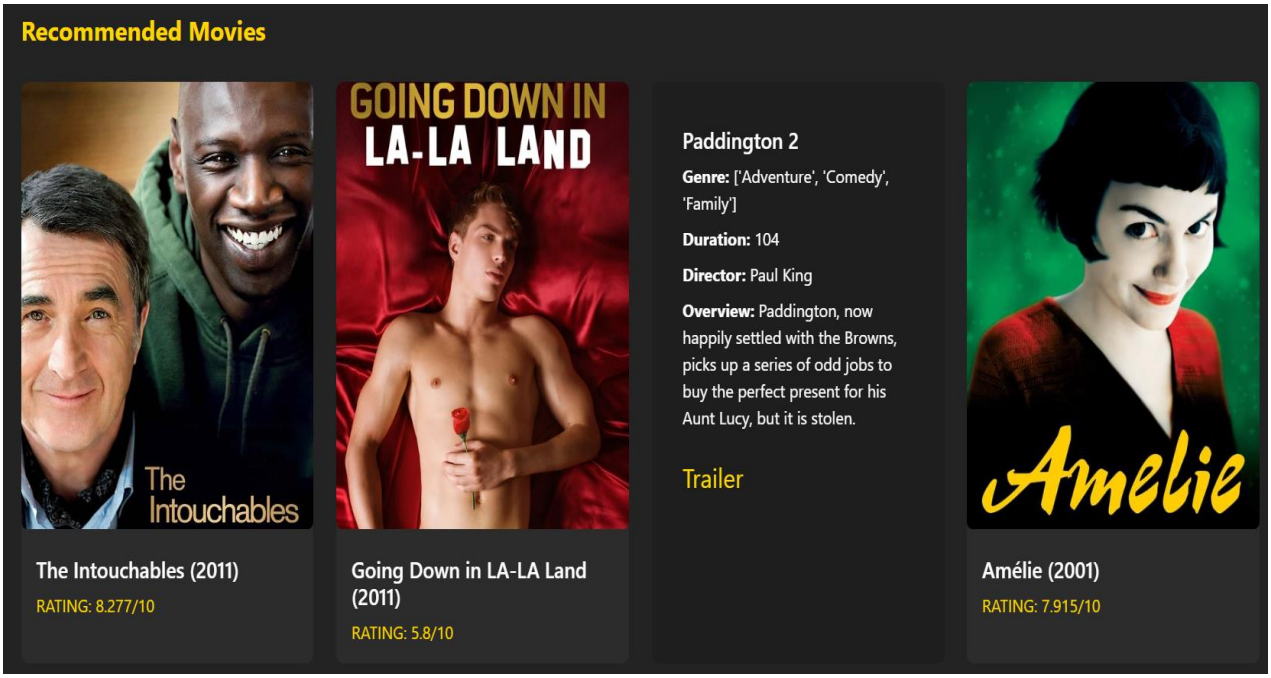

Figure 4 Mood-based Result

3. Content-Based Recommendations

- **What It Does**: Recommends movies based on a given set of content characteristics like actors, directors, and movie plot. As shown in fig. 5 and 6.
- **How to Use It**:
    1. Go to the "Content-Based Recommendations" section.
    2. Enter keywords or titles related to the type of movie you're interested in.
    3. Type the number of recommendations you want (1-10).
    4. Click "Get Recommendations" to see a list of similar movies.
    5. Here the cards are flip able and users can see the movie details by navigating to the particular card and also watch the trailer of the movie.

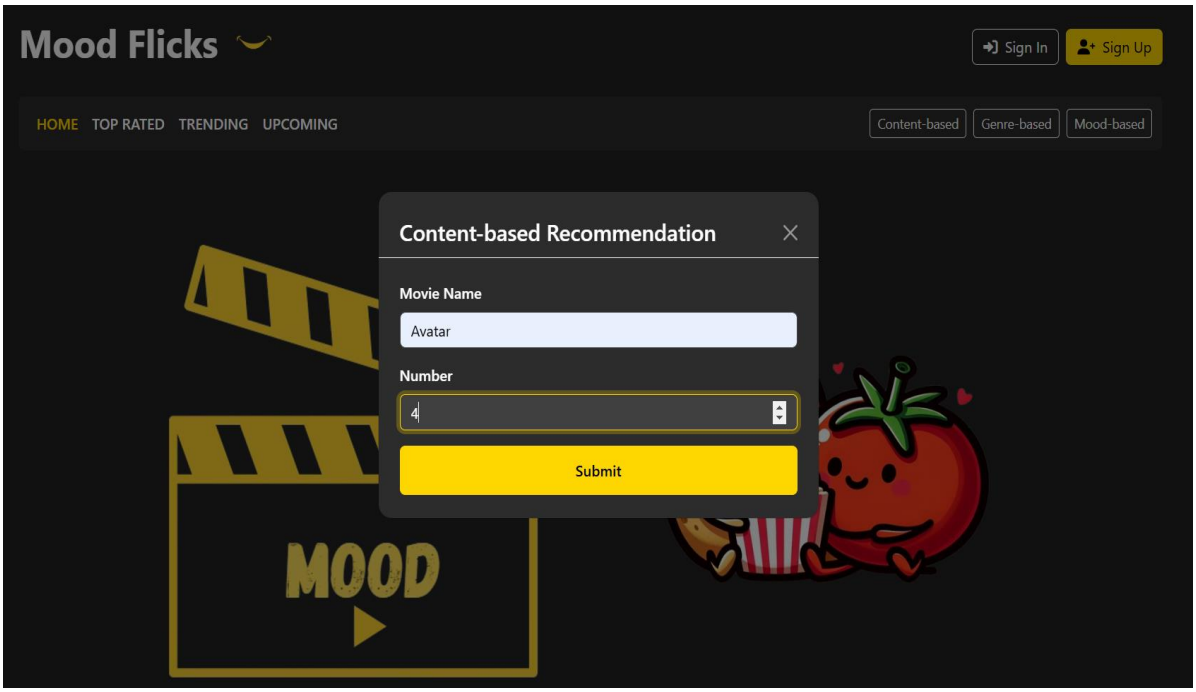

Figure 5 Content Based Search

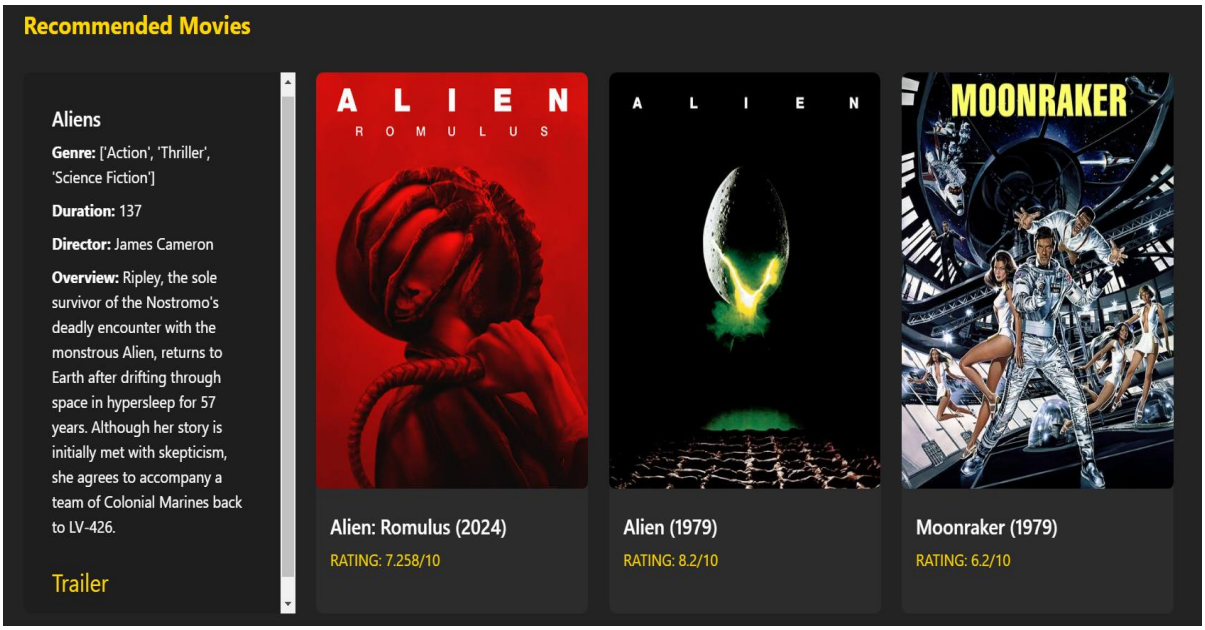

Figure 6 Content Based Result

4. Top Rated Movies

- **What It Does**: Displays a list of the highest-rated movies across all genres. This is a great way to find critically acclaimed films. As shown in fig. 7 and 8
- **How to Use It**:
    1. Navigate to the "Top Rated Movies" section.
    2. The system will show a list of the top-rated movies, including their ratings and basic information.
    3. It comes up with load more features so that users can load more movies from the API.

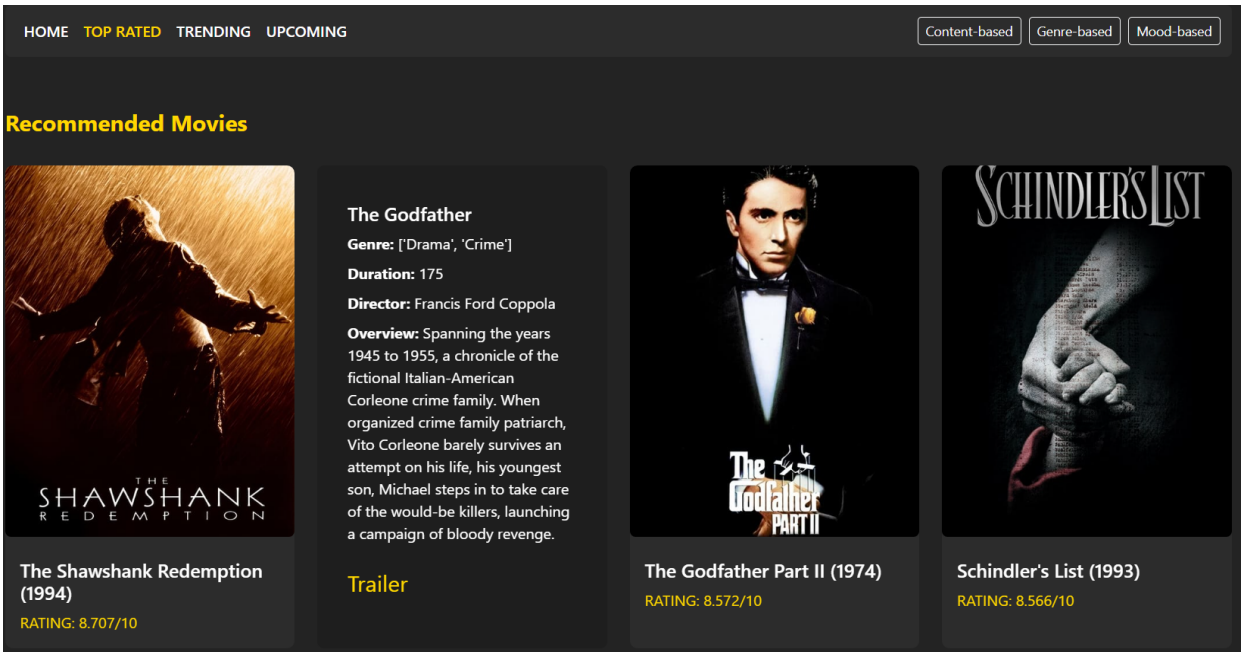

Figure 7 Top Rated Search

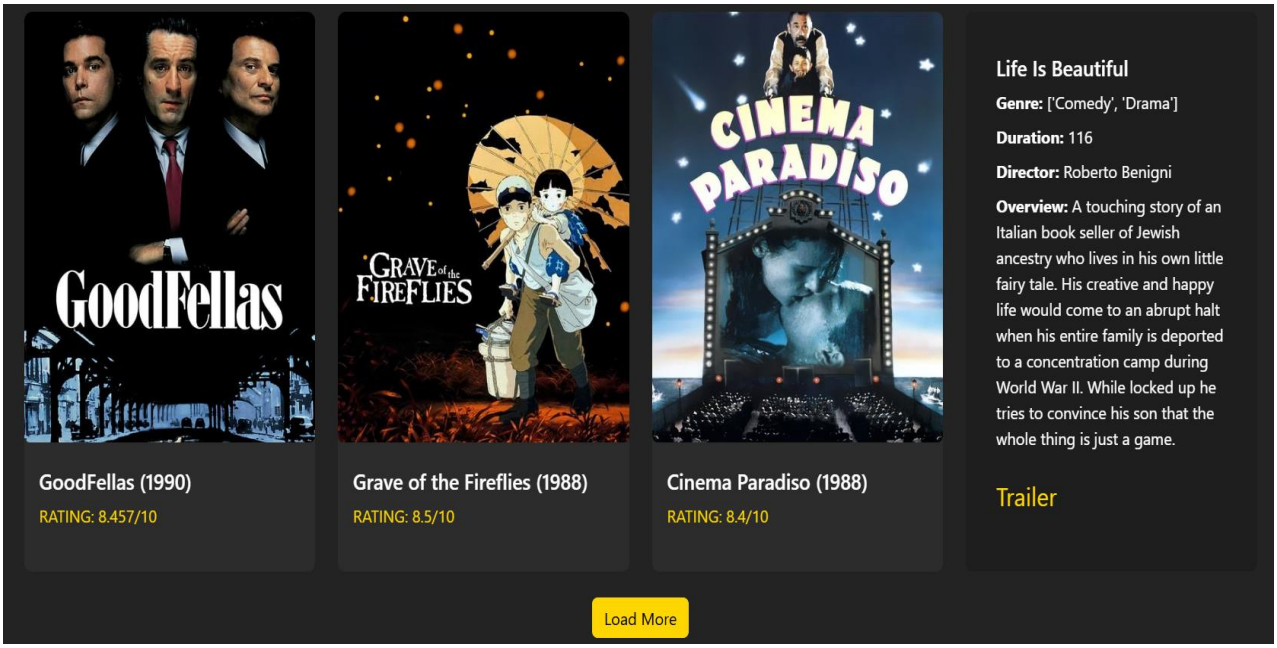

Figure 8 Top Rated Search with load more features

5. Trending Movies

- **What It Does**: Shows a list of movies that are currently trending based on popularity. This includes movies that are being talked about the most or have recently gained attention. As shown in fig. 9 and 10.
- **How to Use It**:
    1. Go to the "Trending Movies" section.
    2. The system will display a list of movies that are trending, based on factors such as box office performance and social media buzz.
    3. It comes up with load more features so that users can load more movies from the API.

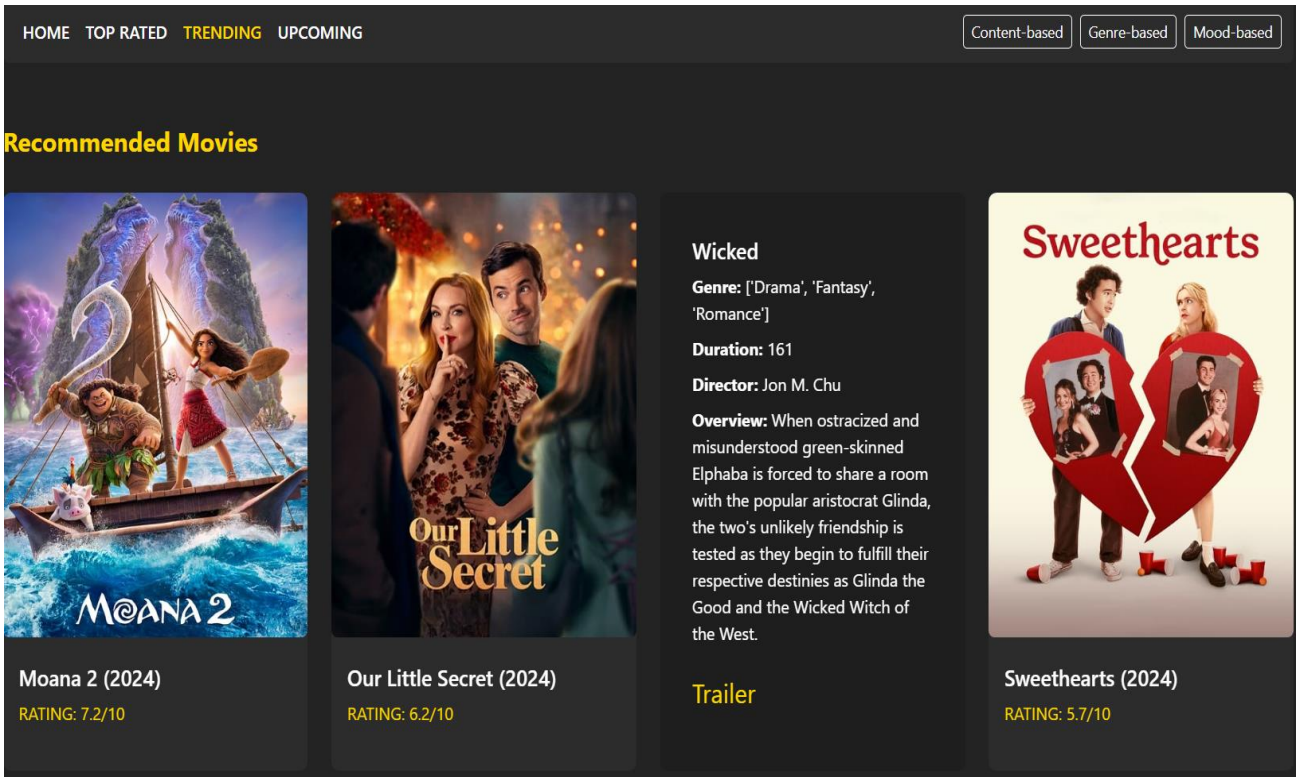

Figure 9 Trending Search

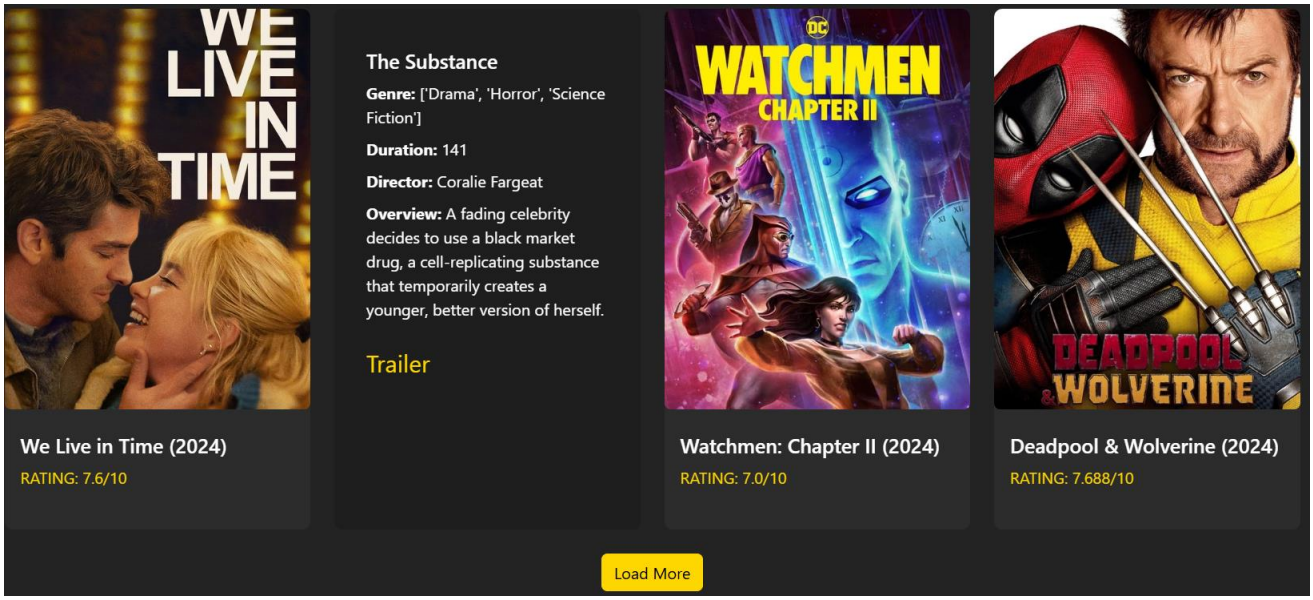

Figure 10 Trending Search with load more features

6. Movie Details and Ratings

- **What It Does**: You can view detailed information about a movie, including ratings, plot summaries, and trailers. As shown in fig. 11 and 12.
- **How to Use It**:
    1. After getting the recommendations or viewing top-rated/trending movies, navigate on any movie card to view more detailed information.

2. The page will display a summary, ratings, and other relevant information like release year, cast, and trailer links.

Figure 11 Movie card front

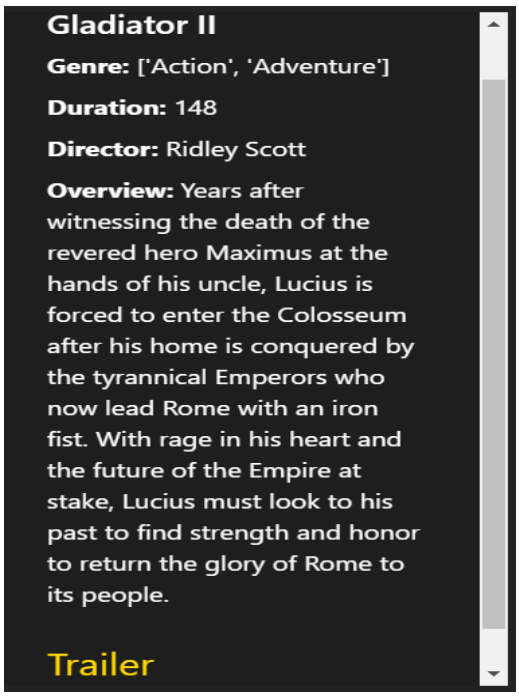

Figure 12 Movie card back

7. User Authentication (With future implementations)

- **What It Does**: Allows users to sign up, log in, and personalize their recommendations. As shown in fig. 13 and 14
- **How to Use It**:
    1. Click on the "Login" or "Sign Up" button.
    2. Enter your credentials to log into the system or create a new account.
    3. Once logged in, your recommendations will be tailored to your preferences and history.

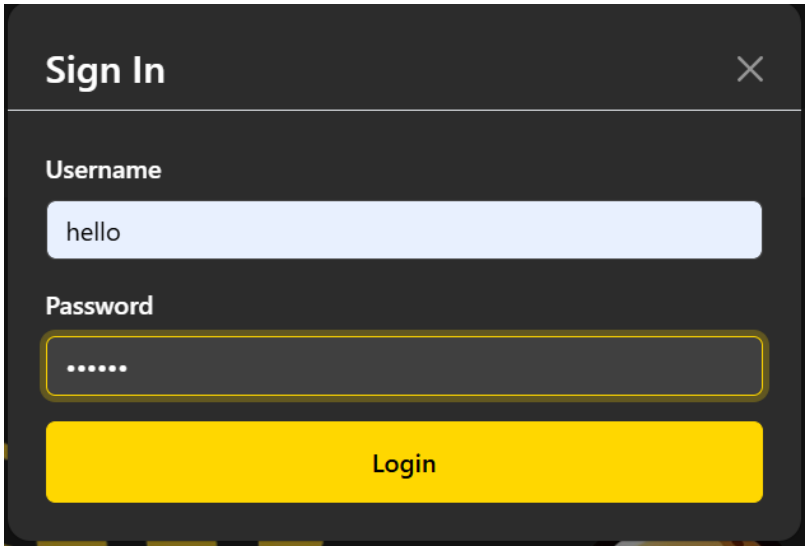

Figure 13 Sign in Box

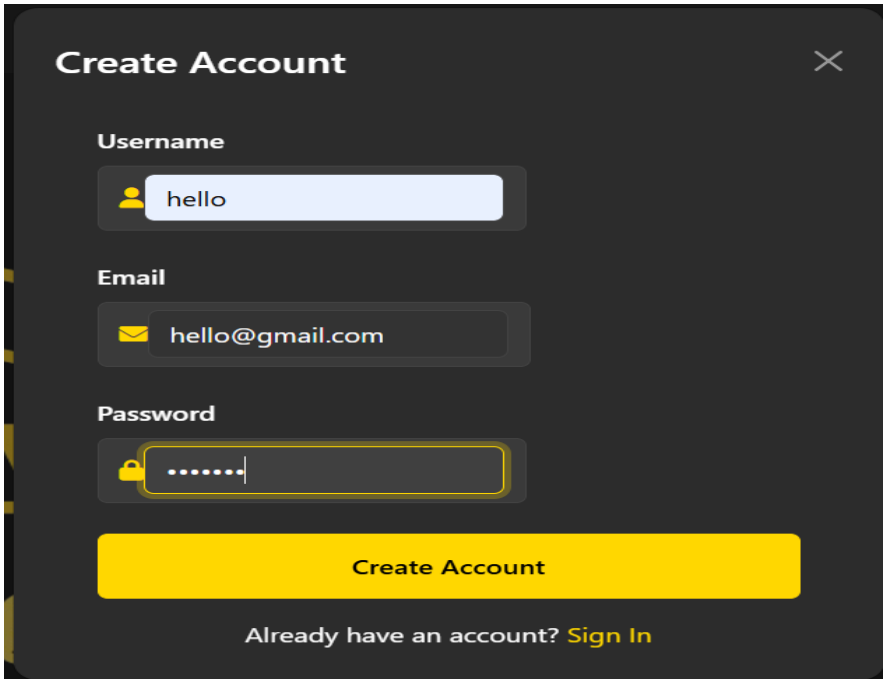

Figure 14 Sign up Box

8. Real-Time Notifications (With future implementations)

- **What It Does**: Notifies users when new movie recommendations are available or when a new feature is added.
- **How to Use It**:
    - Enable notifications on your browser settings.
    - You will receive notifications about the latest movie recommendations or updates related to the app.

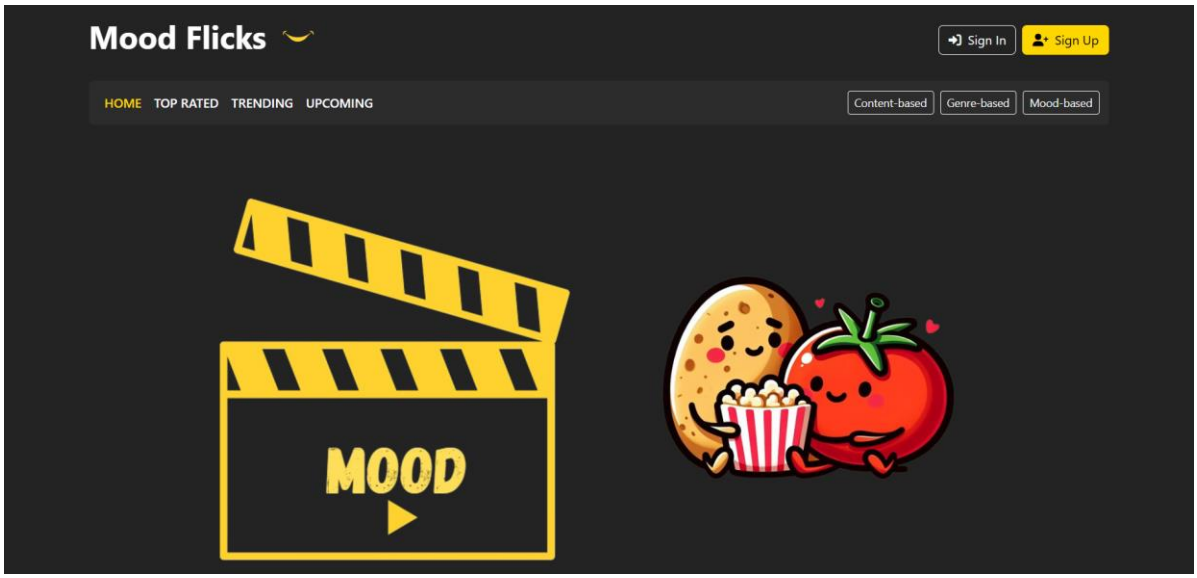

Figure 15 Main page

Troubleshooting and FAQs

- **Q: I can't see any movie recommendations. What should I do?**
    - Ensure that you have selected at least one genre, mood, or keyword before clicking "Get Recommendations."

- **Q: How can I filter my movie recommendations?**
    - The system currently suggests movies based on genre, mood, or content. You can modify your preferences and try again.

- **Q: Why am I not getting any mood-based recommendations?**
    - Ensure that your mood is selected correctly. If the recommendations still don’t show, try refreshing the page.

Chapter 3

3. Developer Documentation

3.1 Problem Specification

The Movie Recommendation System is a web application designed to provide personalized movie recommendations based on user preferences such as genres, moods, and keywords. The system integrates external APIs from **TMDb** for movie details and **OpenAI** for mood-based recommendations. It aims to provide users with the following features:

- **Genre-Based Recommendations**: Movie suggestions based on selected genres.
- **Mood-Based Recommendations**: Movie suggestions based on the user's mood, powered by OpenAI.
- **Content-Based Recommendations**: Movie suggestions based on a specified keyword or title.
- **Trending and Top-Rated Movies**: The system fetches trending and top-rated movies from TMDb.

3.2 Tools and Methods Used

The following methods and approaches were used to build the Movie Recommendation System:

2.1 APIs Used

1. **TMDb API**:
    - Provides movie details (e.g., title, genre, rating, etc.), trending and top-rated movies.
    - Key methods used:
        - get_movie_details_by_title(title): Fetches detailed information of a movie by its title.
        - get_movie_details_by_id(id): Fetches detailed information of a movie by its ID.
        - get_trending_movies(): Fetches the list of currently trending movies.
        - get_top_rated_movies(): Fetches the list of top-rated movies.
2. **OpenAI API**:
    - Provides mood-based recommendations.
    - Key methods used:
        - generate_mood_based_prompt(number, category): Generates movie suggestions based on a given mood category, by using GPT-based text generation.
        - generate_genre_based_prompt(number, category): Generates

movie suggestions based on a given genre category, by using GPT-based text generation.

2.2 Web App Framework

- **Frontend**: HTML, CSS, JavaScript.
- **Backend**: Python (Flask).
- **Database**: PostgreSQL (hosted in Docker) to manage user data, preferences, and movie history.
- **Docker**: Used to containerize the database and backend services for easy deployment.

2.3 Containerization and Virtualization

- **Docker:** Used to containerize the application and database for consistent and portable deployment.
- **Docker Compose:** To manage multi-container applications (Flask app + PostgreSQL + Adminer).

2.4 Development Environment

- **Visual Studio Code (VS Code)**: Used as the primary code editor for writing and debugging Python, HTML, CSS, and JavaScript. It offers extensions for linting, formatting, and integrating with Docker and Git.

2.5 Testing and Debugging

- **Postman**: API testing tool to test endpoints and debug issues during development.
- **Pytest**: Python testing framework to write and execute unit tests.
- **Browser Developer Tools**: Used for debugging and testing the frontend

2.6 Other Utilities

- **VS Code Extensions:**
  - **Python Extension**: For syntax highlighting, debugging, and code intelligence.
  - **Docker Extension:** To manage Docker containers directly from VS Code.
  - **Prettier:** For formatting HTML, CSS, and JavaScript files.

2.7 Version Control

- **Git:** For version control and collaboration.
- **GitHub:** Used as a remote repository to store and manage the project codebase.

2.8 Key Functionalities

- **User Authentication**: Users can sign up, log in .
- **Movie Recommendation**: Based on user input, the system returns a set of recommended movies.
- **Data Caching**: To improve performance, the system caches frequent API requests (e.g., top-rated, trending movies) to reduce external API calls.
- **Real-time Updates**: The app can dynamically display trending and top-rated movies using asynchronous calls.

I chose **OpenAI** for this **Movie Recommendation System** over other LLMs because

1. **Advanced Text Generation**: OpenAI's GPT models, particularly **GPT-3** and **GPT-4**, are known for their ability to generate contextually rich, personalized, and human-like responses, making them ideal for generating mood-based movie recommendations. They can process complex inputs (like mood descriptions) and output nuanced recommendations, a key requirement for your system.
2. **Ease of Integration**: OpenAI provides a highly accessible API, which is developer-friendly and requires minimal configuration. For a project focused on delivering quick and effective recommendations, OpenAI's easy-to-use interface and the extensive documentation saved significant time and effort.
3. **Strong Support for Personalization**: OpenAI models excel at understanding subtle differences in language, which is essential for mood-based recommendation systems. The ability to refine recommendations based on nuanced user input (e.g., "feeling adventurous" or "looking for a feel-good movie") was a key deciding factor.
4. **Established Track Record**: OpenAI has been successfully deployed across numerous industries for various use cases, including personalized recommendations, customer support, and content generation. This made it a reliable choice for a production-level recommendation system.
5. **Scalability**: OpenAI's infrastructure is designed to handle large-scale requests, which is important as the system expands to support more users and more complex recommendation tasks.
6. **Low Latency**: OpenAI's API is optimized for low-latency responses, meaning it can quickly process requests and generate recommendations or answers. This is especially important for real-time applications like a movie recommendation system where users expect immediate feedback.

I chose the **GPT-4o-mini** model for my movie recommendation system due to its **balance of performance and efficiency**. It delivers high-quality recommendations with nuanced understanding, thanks to its advanced language capabilities, while being computationally efficient and cost-effective. Additionally, its **speed**, **reliability**, and seamless API integration make it ideal for real-time user interactions, outperforming many other models in both usability and scalability.

TMDb API

The **TMDb API** was chosen for this project as it provides a **comprehensive and reliable dataset of movies and TV shows**. It offers detailed information, including genres, keywords, ratings, popularity metrics, and posters, which are crucial for building a robust recommendation system. TMDb's rich and constantly updated database ensures that users get accurate and up-to-date information about trending and top-rated movies. Additionally, its API is easy to integrate, making it an efficient choice for developing movie recommendation systems.

5000 Movie Dataset and Content-Based Recommendation
For the **content-based recommendation system**, I used a **5000-movie dataset** sourced from TMDb. This dataset was ideal for applying machine learning techniques to generate personalized recommendations based on movie content.

TF-IDF Technique

The **TF-IDF (Term Frequency-Inverse Document Frequency)** vectorization technique was employed to analyze and represent movie descriptions. Here's why TF-IDF was chosen:

1. **Focus on Key Features**: TF-IDF highlights the most important keywords in movie overviews by emphasizing terms that are frequent within a single description but rare across others.
2. **Efficient Text Vectorization**: It converts textual data into numerical vectors, enabling the application of machine learning algorithms.
3. **Scalability**: TF-IDF is computationally efficient, making it suitable for a dataset of this size and allowing for fast similarity calculations.

Machine Learning Algorithm for Content-Based Filtering
Using **cosine similarity** with TF-IDF vectors, I implemented a **content-based recommendation system**. This approach measures the similarity between movies based on their TF-IDF vectors and recommends movies with the highest similarity scores to a given input movie.

Why This Algorithm is Good:

1. **Personalized Recommendations**: The model works well for users looking for movies like their preferences by analysing descriptive content.
2. **Interpretability**: TF-IDF combined with cosine similarity provides interpretable results, as each recommended movie has a clear textual basis.
3. **No Dependency on User Data**: Unlike collaborative filtering, this system doesn't rely on user interaction data, making it robust for new users or items.
4. **Fast and Lightweight**: The algorithm is computationally efficient, ensuring quick recommendations even with a sizable dataset.

Summary
The combination of TMDb API, a 5000-movie dataset, TF-IDF, and cosine similarity creates a powerful, scalable, and interpretable recommendation system. TMDb's robust data source, paired with the efficiency of TF-IDF and cosine similarity, ensures high-quality, relevant, and personalized movie recommendations for users.

3.3 UML Diagram

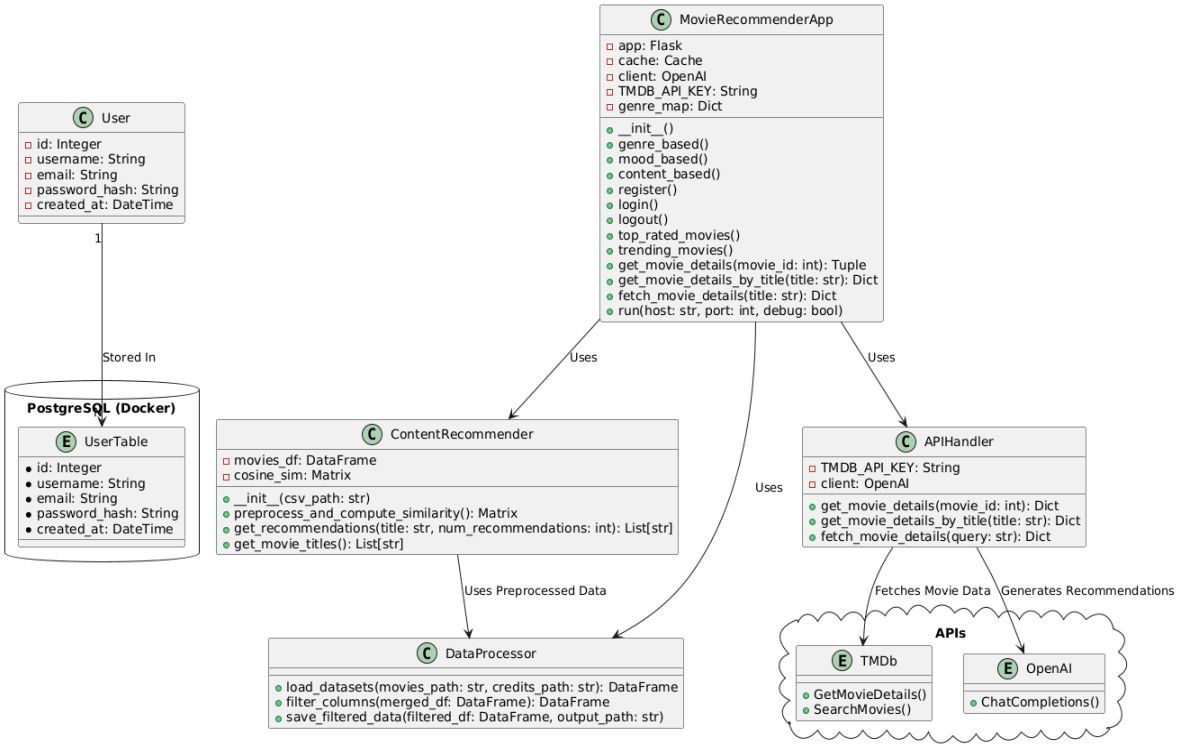

Figure 16 UML Diagram

OOP Relationships :

1. **Composition**
   - MovieRecommenderApp is composed of ContentRecommender and APIHandler.
   - ContentRecommender integrates the DataProcessor for data preparation.
2. **Dependency**
   - MovieRecommenderApp depends on external libraries (Flask, Cache) and APIs (TMDb, OpenAI).
   - APIHandler depends on TMDb and OpenAI APIs for data retrieval and advanced recommendations.
   - ContentRecommender depends on Pandas for handling datasets.
3. **Aggregation**
   - User interacts with the PostgreSQL database for authentication and data management.
4. **Association**
   - User is associated with the MovieRecommenderApp for actions like registration and login.
   - DataProcessor interacts with external CSV datasets for data loading and filtering.

3.4 User Case Diagram

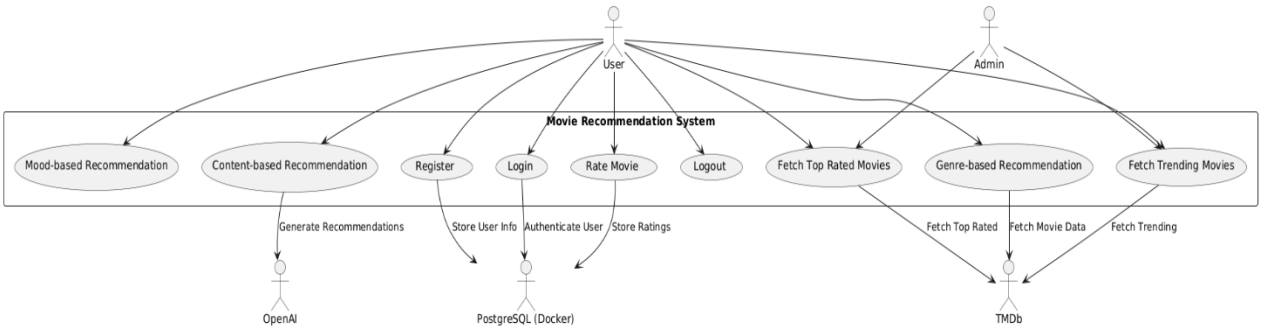

Figure 17 User Case Diagram

3.5 User Sequence Diagram

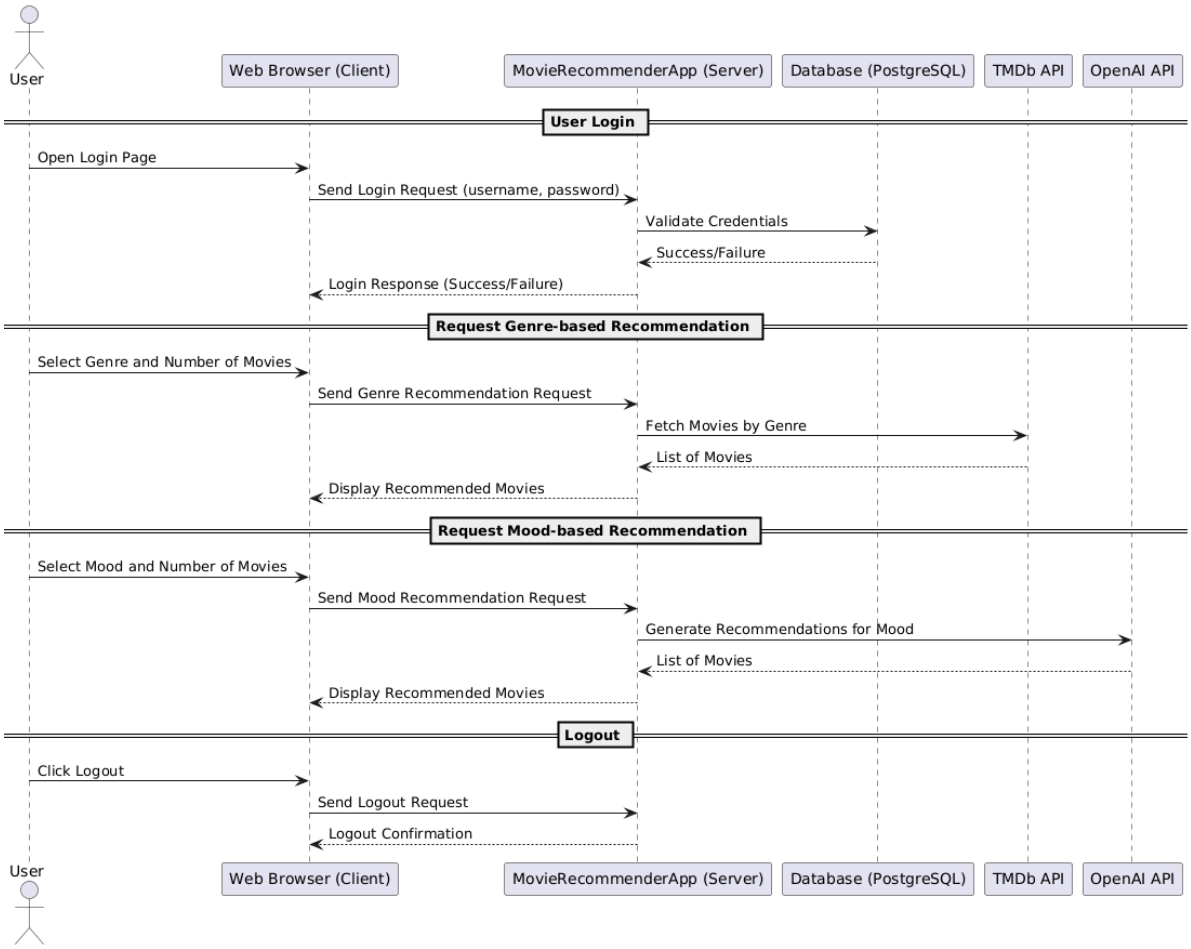

Figure 18 User Sequence Diagram

Interactions and Workflow

1. **User Registration & Login:**
    - User information is stored in the PostgreSQL database using the User class.
    - The MovieRecommenderApp handles authentication and session management.
2. **Recommendation Generation:**
    - **Content-Based Recommendations:**
        - ContentRecommender uses preprocessed movie data and cosine similarity to recommend movies based on the content.
    - **Genre-Based Recommendations:**
        - MovieRecommenderApp filters movies by genre using data retrieved from TMDb.
    - **Mood-Based Recommendations:**
        - APIHandler interacts with OpenAI APIs to provide recommendations tailored to the user's mood.
3. **Data Preprocessing:**
    - The DataProcessor class processes raw datasets for creating the similarity matrix used by ContentRecommender.
4. **API Integration:**
    - The APIHandler class fetches movie data and details from TMDb and enhances recommendations using OpenAI's capabilities.

The MovieRecommenderApp acts as the central application that orchestrates all operations. It uses Flask as its web framework (app attribute) to handle HTTP requests, and integrates a cache mechanism to optimize data retrieval. The client attribute connects to OpenAI's API for advanced recommendations, while the TMDb_API_KEY (String) is used to access the TMDb database for movie data. The genre_map (Dictionary) supports genre-based filtering. The app provides multiple functionalities through methods like register(), login(), and logout() for user management, as well as genre_based(), mood_based(), and content_based() for different types of recommendations. Additional methods like top_rated_movies() and trending_movies() retrieve movie trends, while fetch_movie_details() gets detailed data for a given title.

3.6 Developer Setup

6.1. Prerequisites
Ensure the following are installed on your system:

- **Python 3.8+**
- **Docker** (for running PostgreSQL in a container)
- **pip** (Python package manager)
- **Git** (version control)
- **Postman** (optional, for API testing)

6.2. Clone the Repository

1. Open a terminal or command prompt.
2. Create Project
3. Clone the project repository:

bash
Copy code

```
git init
git clone https://github.com/inspironman/Thesis.git
```

6.3. Setup Python Environment

1. Create a Virtual Environment:

bash
Copy code

```
python -m venv venv
```

6.4. Activate the Virtual Environment:

- On **Linux/macOS**:

bash
Copy code

```
source venv/bin/activate
```

- On **Windows**:

bash
Copy code

```
venv\Scripts\activate
```

6.5. Install Dependencies
bash
Copy code

```
pip install -r requirements.txt
```

6.6. Configure Environment Variables

Update .env with appropriate values:

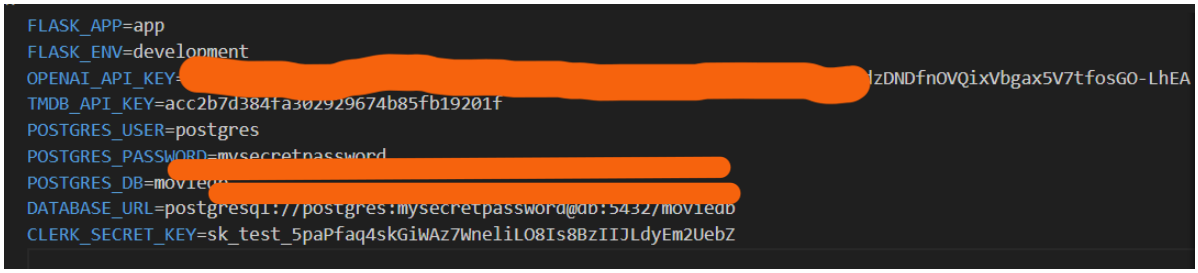

Figure 19 env structure

6.7. Setup Docker for PostgreSQL

Create a docker-compose.yml file for PostgresSQL:

docker-compose.yml

```yaml
version: '3.8'

services:
  web:
    build: .
    ports:
      - "8080:8080"
    environment:
      - DATABASE_URL=postgresql://postgres:mysecretpassword@db:5432/moviedb
      - FLASK_ENV=development
      - FLASK_APP=/api/api/index2.py
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8080/ || exit 1"]
      interval: 10s
      retries: 3
    networks:
      - app-network

  db:
    image: postgres:13
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=mysecretpassword
      - POSTGRES_DB=moviedb
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "postgres", "-d", "moviedb"]
      interval: 10s
      retries: 5
    networks:
      - app-network
```

Figure 20 docker_code

```
  adminer:
    image: adminer
    restart: always
    ports:
      - "8081:8080"
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  postgres_data:
```

Dockerfile > ...

```
FROM python:3.9

# Set the working directory inside the container
WORKDIR /api

# Copy requirements file and install dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the entire project into the container
COPY . .

# Expose port 8080
EXPOSE 8080

# Set environment variable to force Flask to run in production mode
ENV FLASK_ENV=development
ENV FLASK_APP=/api/api/index.py

# Command to run the application
CMD ["python", "/api/api/index2.py"]
```

Run Docker Compose :
bash
Copy code

```
docker-compose up -d
```

Verify PostgreSQL is running
bash
Copy Code

```
docker ps
```

```
PS D:\Thesis> docker ps
CONTAINER ID   IMAGE         COMMAND                  CREATED        STATUS                  PORTS                    NAMES
7cae5c6f6952   thesis-web    "python /api/api/ind…"   2 hours ago    Up 2 hours (healthy)    0.0.0.0:8080->8080/tcp   thesis-web-1
3d784718e31a   postgres:13   "docker-entrypoint.s…"   25 hours ago   Up 2 hours (healthy)    0.0.0.0:5432->5432/tcp   thesis-db-1
7b3738742fa3   adminer       "entrypoint.sh php -…"   25 hours ago   Up 2 hours              0.0.0.0:8081->8080/tcp   thesis-adminer-1
PS D:\Thesis>
```

Figure 21 docker ps

6.8. Directory Structure

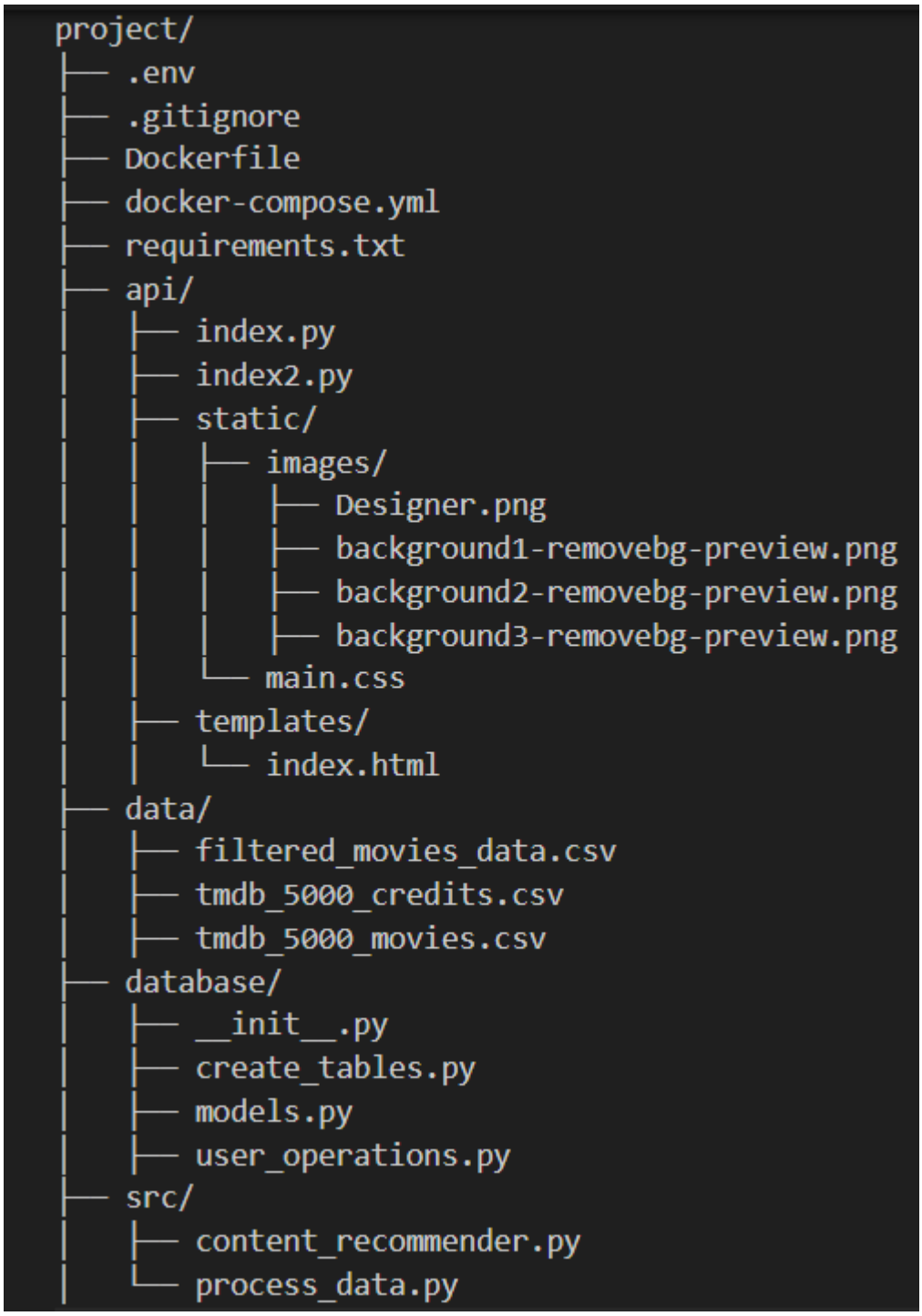

Figure 22 tree

3.7 Core Components

7.1 Content Based Recommender : The **ContentRecommender** class leverages **pandas** for reading and manipulating the movie dataset, and **sklearn.feature_extraction.text.TfidfVectorizer** to convert textual features into **TF-IDF vectors**. These vectors are used with **sklearn.metrics.pairwise.cosine_similarity** to compute the similarity between movies based on their textual content. The class's methods include **preprocess_and_compute_similarity**, which combines key movie features (like **title**, **overview**, **genres**, **keywords**, **cast**, and **crew**) into a single string for similarity analysis, and **get_recommendations**, which returns the most similar movies. This **content-based filtering** approach efficiently identifies movie similarities using **TF-IDF** and **cosine similarity**.

```python
class ContentRecommender:
    def __init__(self, csv_path):
        self.movies_df = pd.read_csv(csv_path)
        self.cosine_sim = self.preprocess_and_compute_similarity()

    def preprocess_and_compute_similarity(self):
        self.movies_df['combined_features'] = (
            self.movies_df['title'].fillna('') + ' ' +
            self.movies_df['overview'].fillna('') + ' ' +
            self.movies_df['genres'].fillna('') + ' ' +
            self.movies_df['keywords'].fillna('') + ' ' +
            self.movies_df['cast'].fillna('') + ' ' +
            self.movies_df['crew'].fillna('')
        )

        # Create TF-IDF vectors
        tfidf = TfidfVectorizer(stop_words='english')
        tfidf_matrix = tfidf.fit_transform(self.movies_df['combined_features'])

        return cosine_similarity(tfidf_matrix, tfidf_matrix)

    def get_recommendations(self, title, num_recommendations=10):
        idx = self.movies_df[self.movies_df['title'] == title].index[0]

        sim_scores = list(enumerate(self.cosine_sim[idx]))
        sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)
        sim_scores = sim_scores[1:num_recommendations+1]

        movie_indices = [i[0] for i in sim_scores]

        return self.movies_df['title'].iloc[movie_indices].tolist()

    def get_movie_titles(self):
        return self.movies_df['title'].tolist()
```

Figure 23 code example 1

7.2 Process Data : The script uses pandas to load and merge the movies and credits datasets, then processes the data to extract relevant features. The **convert** function parses genre data, the **convert3** function limits the cast to the top 3 members, and **fetch_director** extracts the director's name from the crew list. The **filter_columns** function filters and cleans the merged Data Frame, retaining essential columns like genres, keywords, cast, and crew. Finally, **save_filtered_data** saves the cleaned Data Frame to a CSV file for further use in the **ContentRecommender** system.

```python
import pandas as pd
import ast

def load_datasets(movies_path, credits_path):
    """
    Load the movies and credits datasets from CSV files.
    :param movies_path: Path to the movies dataset
    :param credits_path: Path to the credits dataset
    :return: Merged DataFrame
    """
    movies_df = pd.read_csv(movies_path)
    credits_df = pd.read_csv(credits_path)

    merged_df = movies_df.merge(credits_df, on='title')

    print("Columns in merged DataFrame:", merged_df.columns.tolist())

    return merged_df


def convert(text):
    L = []
    for i in ast.literal_eval(text):
        L.append(i['name'])
    L = ' '.join(L)
    return L


def convert3(text):
    L = []
    counter = 0
    for i in ast.literal_eval(text):
        if counter < 3:
            L.append(i['name'])
        counter+=1
    return L
```

```python
def fetch_director(text):
    L = []
    for i in ast.literal_eval(text):
        if i['job'] == 'Director':
            L.append(i['name'])
    return L


def filter_columns(merged_df):
    """
    Filter the merged DataFrame to keep only the necessary columns.
    :param merged_df: Merged DataFrame from movies and credits datasets
    :return: Filtered DataFrame
    """
    print("Columns in merged DataFrame:", merged_df.columns.tolist())

    filtered_df = merged_df[['movie_id', 'title', 'overview','genres', 'keywords',
                             'vote_average', 'vote_count', 'popularity',
                             'release_date', 'cast', 'crew']]
    filtered_df.dropna(inplace=True)
    filtered_df['genres'] = filtered_df['genres'].apply(convert)
    filtered_df['keywords'] = filtered_df['keywords'].apply(convert)
    filtered_df['cast'] = filtered_df['cast'].apply(convert3)
    filtered_df['crew'] = filtered_df['crew'].apply(fetch_director)

    return filtered_df

def save_filtered_data(filtered_df, output_path):
    """
    Save the filtered DataFrame to a CSV file.
    :param filtered_df: DataFrame with filtered columns
    :param output_path: Path where the filtered data will be saved
    """
    filtered_df.to_csv(output_path, index=False)
    print(f"Filtered data saved to {output_path}")
```

Figure 24 code example 2

7.2.1 Processed Data Sample

```
movie_id,title,overview,genres,keywords,vote_average,vote_count,popularity,release_date,cast,crew
19995,Avatar,"In the 22nd century, a paraplegic Marine is dispatched to the moon Pandora on a unique mission, but becomes torn between following orders and protecting an
285,Pirates of the Caribbean: At World's End,"Captain Barbossa, long believed to be dead, has come back to life and is headed to the edge of the Earth with Will Turner a
206647,Spectre,"A cryptic message from Bond's past sends him on a trail to uncover a sinister organization. While M battles political forces to keep the secret service a
49026,The Dark Knight Rises,"Following the death of District Attorney Harvey Dent, Batman assumes responsibility for Dent's crimes to protect the late attorney's reputat
49529,John Carter,"John Carter is a war-weary, former military captain who's inexplicably transported to the mysterious and exotic planet of Barsoom (Mars) and reluctant
```

Figure 25 processed data

7.3 Databse Model : The **User** table in the database is created using **SQLAlchemy** with **PostgreSQL** as the database backend. The model defines the necessary fields for user authentication and management: id, username, email, password_hash, and created_at. The **id** field is the primary key, while **username** and **email** are unique and non-nullable. The **password_hash** stores the hashed password for secure authentication, and **created_at** records the timestamp when the user is created.

```python
from sqlalchemy import Column, Integer, String, DateTime, ForeignKey, CheckConstraint
from sqlalchemy.orm import declarative_base
from sqlalchemy.sql import func

Base = declarative_base()
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    username = Column(String(50), unique=True, nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    password_hash = Column(String(250), nullable=False)
    created_at = Column(DateTime(timezone=True), server_default=func.now())
```

Figure 26 data base code

After creating User model I have also made some user operation functions to handle **user registration** and **authentication** using **PostgreSQL** and **Werkzeug** for password hashing. The **get_db_connection** function establishes a connection to the PostgreSQL database using environment variables. In **register_user**, the code checks if the username or email already exists, hashes the password, and inserts the new user into the database. **authenticate_user** retrieves the user from the database and verifies the password using **check_password_hash**. Both functions manage database connections and handle errors with appropriate rollback mechanisms.

7.4 Functions and Methods for Main Application

1. **index()**:
    - **Purpose**: Displays the homepage with the option to select recommendation type (genre, mood, or content-based).
    - **Route**: /
    - **Methods**: GET
    - **Parameters**: recommendation_type (passed as query parameter)
    - **Returns**: Renders the main page with movie recommendations and user details.

2. **genre_based()**:
    - **Purpose**: Handles genre-based movie recommendations.
    - **Route**: /genre_based
    - **Methods**: GET, POST
    - **Parameters**: category (genre), number (number of recommendations)
    - **Returns**: Renders the page with movie recommendations based on the selected genre.

3. **mood_based()**:
    - **Purpose**: Handles mood-based movie recommendations using **OpenAI**'s GPT model.
    - **Route**: /mood_based
    - **Methods**: GET, POST
    - **Parameters**: mood (mood type), number (number of recommendations)
    - **Returns**: Renders the page with movie recommendations based on the selected mood.

4. **content_based()**:
    - **Purpose**: Handles content-based movie recommendations based on a user's input movie.
    - **Route**: /content_based
    - **Methods**: GET, POST
    - **Parameters**: movie (movie title), number (number of recommendations)
    - **Returns**: Renders the page with movie recommendations based on the content similarity of the selected movie.

5. **register()**:
    - **Purpose**: Handles user registration.
    - **Route**: /register
    - **Methods**: GET, POST
    - **Parameters**: username, email, password
    - **Returns**: Renders the registration page, either with a success message or an error message.

6. **login()**:
    - **Purpose**: Handles user login.
    - **Route**: /login
    - **Methods**: GET, POST
    - **Parameters**: username, password
    - **Returns**: Renders the login page with appropriate success or failure messages.

7. **logout()**:
    - **Purpose**: Logs the user out by clearing the session.
    - **Route**: /logout
    - **Methods**: POST
    - **Returns**: Redirects to the homepage with a success message.

8. **top_rated_movies()**:
    - **Purpose**: Fetches and displays the top-rated movies from **TMDb**.
    - **Route**: /top_rated
    - **Methods**: GET
    - **Parameters**: page (optional, for pagination)
    - **Returns**: Renders the page with top-rated movie details.

9. **trending_movies()**:
    - **Purpose**: Fetches and displays the trending movies from **TMDb**.
    - **Route**: /trending
    - **Methods**: GET
    - **Parameters**: page (optional, for pagination)
    - **Returns**: Renders the page with trending movie details.

3.8 Testing

To ensure the reliability and correctness of the **Movie Recommendation System**, two primary tools were utilized for testing: I used **Pytest** for backend logic and functionality testing, and **Postman** for API testing.

Pytest is a powerful testing framework for Python, widely used for unit testing, functional testing, and integration testing. It was employed in this project to validate backend functionality, including user authentication, recommendation logic, and database operations.

Postman is a widely used API testing tool that simplifies the process of sending requests to APIs and validating their responses. It was leveraged in this project to test the RESTful APIs exposed by the **Movie Recommendation System**.

8.1 Testing using Pytest

8.1.1 test_user_operations.py

- **test_register_user(db_connection):**
  - Purpose: Verifies that a user can be successfully registered in the system.
  - Setup: Registers a new user in the database and checks if the user is present in the users table.
  - Assertions: Checks if the registration is successful and the user appears in the database.

- **test_register_user_duplicate(db_connection):**
  - Purpose: Ensures that the system rejects duplicate usernames or emails during registration.
  - Setup: Attempts to register a user with an existing username or email.
  - Assertions: Verifies that a relevant error message is returned when trying to register a duplicate user.

- **test_authenticate_user(db_connection):**
  - Purpose: Verifies successful user authentication with valid credentials.
  - Setup: Registers a user and attempts to log in with correct credentials.
  - Assertions: Verifies that the user is authenticated and the returned user details match the expected values.

- **test_authenticate_user_invalid(db_connection):**
  - Purpose: Verifies that invalid user credentials (incorrect password) result in authentication failure.
  - Setup: Attempts to authenticate using incorrect credentials.
  - Assertions: Checks that the authentication fails and None is returned.

```
(venv) PS D:\Thesis> pytest .\tests\test_user_operations.py
================================== test session starts ==================================
platform win32 -- Python 3.12.4, pytest-8.3.3, pluggy-1.5.0
rootdir: D:\Thesis
plugins: anyio-4.6.2.post1, mock-3.14.0, time-machine-2.16.0
collected 6 items

tests\test_user_operations.py ......                                            [100%]

=================================== 6 passed in 1.39s ===================================
```

Figure 27 Pytest Resullt

8.1.2 test_recommender.py

- **test_get_movie_titles(recommender):**
  - Purpose: Verifies that the movie titles are correctly loaded from the dataset.
  - Assertions: Ensures that the movie list contains the expected movie titles like "Avatar."
- **test_get_recommendations_content_based(recommender):**
  - Purpose: Tests content-based movie recommendations for a specific movie.
  - Setup: Tests the get_recommendations method by passing in the title "Avatar."
  - Assertions: Checks if the returned recommendations are similar to "Avatar," such as "The Matrix."
- **test_get_recommendations_genre_based(recommender):**
  - Purpose: Verifies genre-based recommendations for a movie (e.g., "Inception").
  - Setup: Calls the get_recommendations method with a genre like "Action."
  - Assertions: Ensures that the recommendations include relevant genre movies, such as "Mad Max."

8.1.3 test_api.py

- **test_get_movie_details(api_handler, mocker):**
  - Purpose: Mocks the TMDb API response and verifies that movie details are correctly fetched.
  - Setup: Mocks the requests.get method to return a predefined movie detail response.
  - Assertions: Ensures that the get_movie_details function returns the correct title and overview.
- **test_generate_mood_recommendation(api_handler, mocker):**
  - Purpose: Mocks the OpenAI API to generate mood-based recommendations.
  - Setup: Mocks the openai.Completion.create method to return a predefined list of movie recommendations.
  - Assertions: Verifies that the returned recommendations match the expected output (e.g., "The Matrix").

8.2 API Testing and Routes Testing Documentation for Postman

This document outlines the steps and details for testing the **Movie Recommendation System** API endpoints using **Postman**. It covers user-related endpoints (registration, login), recommendation routes, and API responses. Each route is listed with its HTTP method, required headers, body, and expected responses.

Prerequisites

1. **Postman Installed**:
   - Download and install Postman from https://www.postman.com/downloads/.
2. **API Running Locally**:
   - Ensure the Flask application is running on your local machine (default: http://127.0.0.1:8080/)
   - If using Docker, ensure the container is running and accessible.
3. **Database Setup**:
   - Confirm the database is initialized, tables are created, and the API can connect to the database.

Base URL

For local testing, use the following **Base URL** for all endpoints:

http://127.0.0.1:8080

8.2.1. User Management Endpoints

1.1 User Registration

- **Endpoint**: /register
- **Method**: POST
- **Headers**:
  - Content-Type: application/json
  -

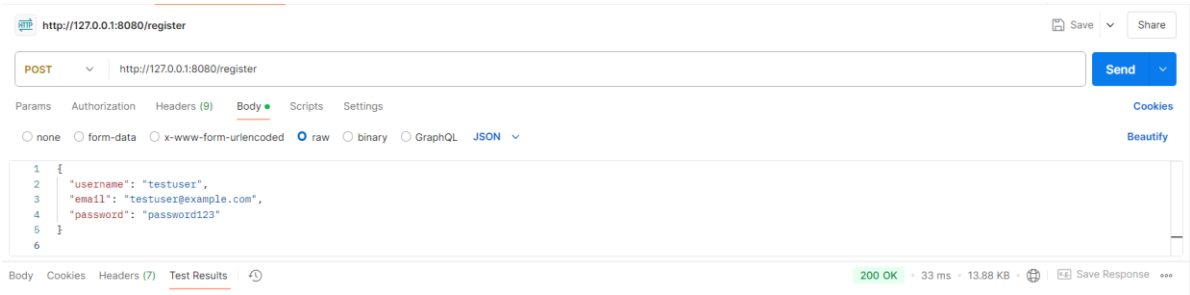

Figure 28 register_test

1.2 User Login

- **Endpoint**: /login
- **Method**: POST
- **Headers**:
    - Content-Type: application/json

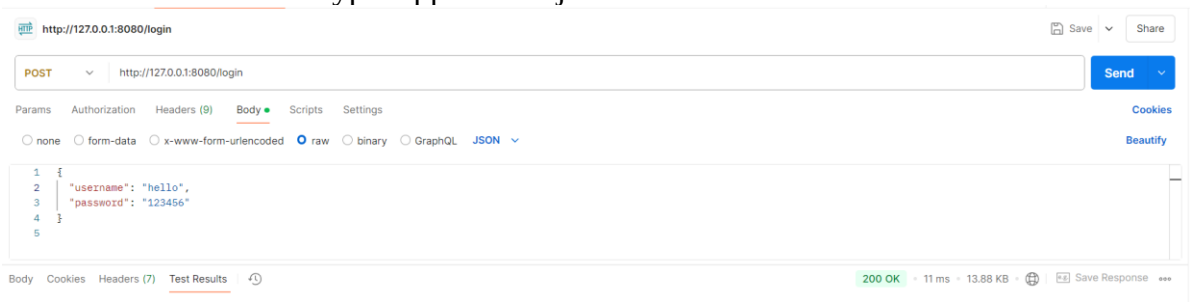

Figure 29 login_test

1.3 User Logout

- **Endpoint**: /logout
- **Method**: POST
- **Headers**:
    - Authorization: Bearer <JWT Token>

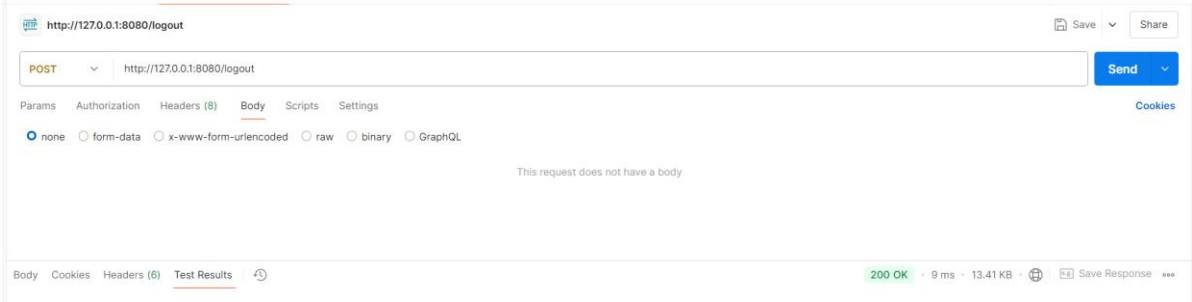

Figure 30 logout_test

8.2.2. Recommendation Endpoints

2.1 Genre-Based Recommendations

- **Endpoint:** /genre_based
- **Method:** POST
- **Headers:**
    - **Content-Type:** application/json

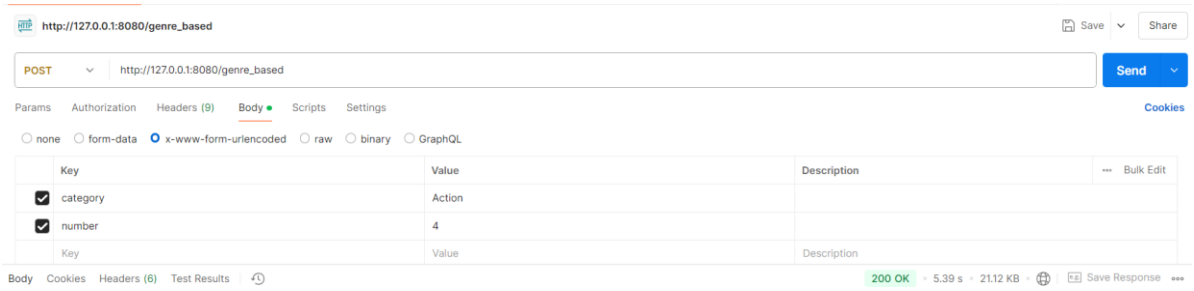

Figure 31 genre_based test

2.2 Mood-Based Recommendations

- **Endpoint:** /mood_based
- **Method:** POST
- **Headers:**
    - **Content-Type:** application/json

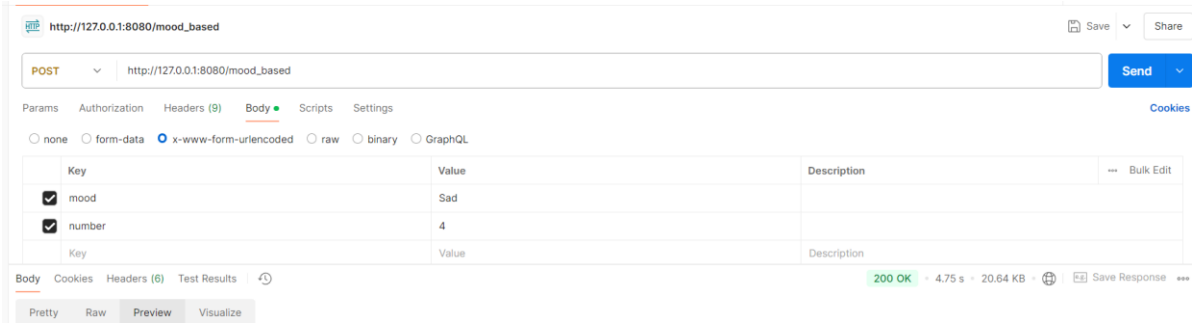

Figure 32 mood_based test

2.3 Content-Based Recommendations

- **Endpoint:** /content_based
- **Method:** POST
- **Headers:**
    - **Content-Type:** application/json

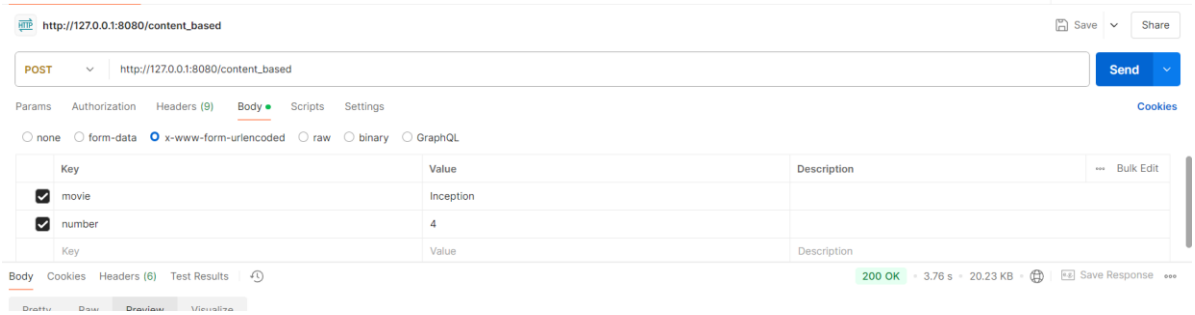

Figure 33 content based test

8.2.3 Movie Details Endpoints

3.1 Fetch Trending Movies

- **Endpoint:** /trending
- **Method:** GET

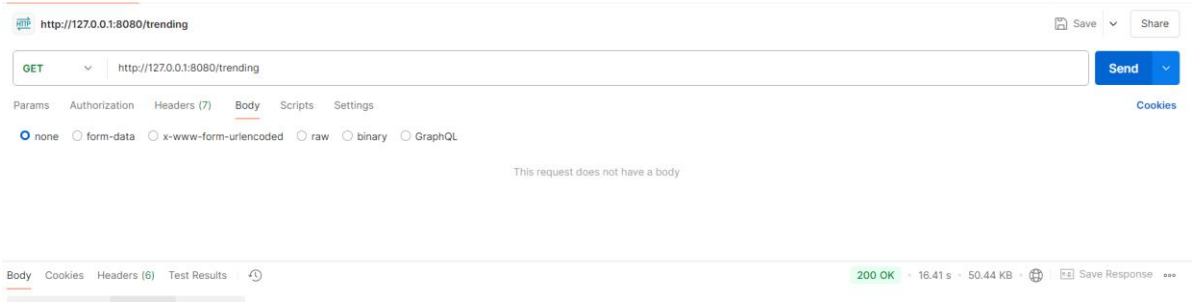

Figure 34 trending test

3.2 Fetch Top-Rated Movies

- **Endpoint:** /top_rated
- **Method:** GET

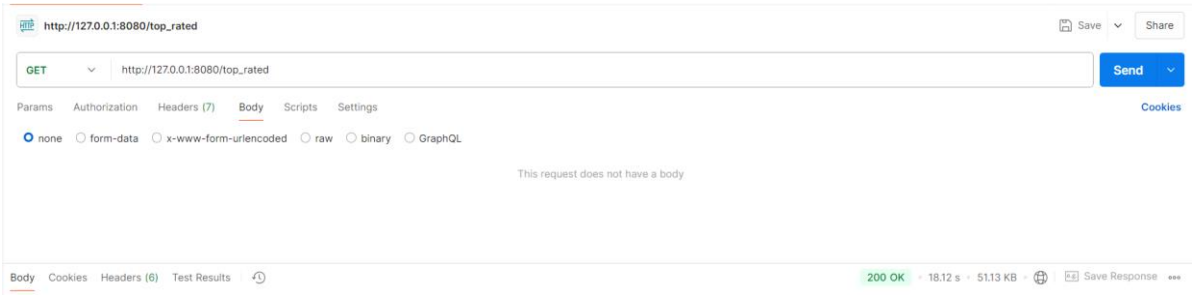

Figure 35 top_rated test

Chapter 4

4. Conclusion and future work

4.1 Conclusion

The Movie Recommendation System successfully provides users with personalized movie recommendations based on genre, mood, and content. It offers seamless user registration, login, and authentication functionalities, along with genre-based and mood-based recommendation capabilities. Users can also explore trending and top-rated movies, making the platform engaging and informative. By integrating APIs like TMDb and OpenAI, the system delivers accurate and data-driven recommendations, ensuring a smooth and user-friendly experience.

4.2 Future Work

To enhance the platform further, the following features can be implemented in the future:

1. **Watch Later Functionality**:
    - Allow users to add movies to a "Watch Later" list for future viewing.
2. **Blog and Comment System**:
    - Enable users to write blogs about movies and share their thoughts.
    - Allow users to comment on blogs or movie pages to foster community engagement.
3. **Movie Ratings and Reviews**:
    - Let users rate movies and write reviews to help others make informed decisions.
4. **Password Reset via Email**:
    - Implement a secure password reset mechanism using email verification.
5. **Follow Movie News and Updates**:
    - Integrate a news feed or notification system to keep users informed about the latest in the movie industry.
6. **Social Features**:
    - Allow users to follow other users, share recommendations, and build a network of movie enthusiasts.
7. **Advanced Recommendation Algorithms**:
    - Incorporate collaborative filtering and hybrid recommendation methods for more accurate suggestions.

By implementing these features, the Movie Recommendation System can evolve into a comprehensive movie engagement platform, offering users a rich and interactive experience.

Bibliography

1. Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval: The Concepts and Technology Behind Search*. Pearson Education.
2. Aggarwal, C. C. (2016). *Recommender Systems: The Textbook*. Springer.
3. The Movie Database (TMDb). (2023). *TMDb API*. Retrieved from https://developers.themoviedb.org/
4. Kaggle. (2023). *Movie Recommendation Dataset*. Retrieved from https://www.kaggle.com
5. Python Software Foundation. (2023). *Python Programming Language*. Retrieved from https://www.python.org/
6. scikit-learn developers. (2023). *scikit-learn: Machine Learning in Python*. Retrieved from https://scikit-learn.org/
7. Flask developers. (2023). *Flask: Web Framework Documentation*. Retrieved from https://flask.palletsprojects.com/
8. Medium. (2021). How to Build a Movie Recommendation System using Python and Machine Learning. *Medium*. Retrieved from https://medium.com/analytics-vidhya/building-movie-recommendation-system-python-823abe51a6fa
9. Machine Learning Mastery. (2020). A Gentle Introduction to Collaborative Filtering for Recommender Systems. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/collaborative-filtering-for-recommender-systems/
10. Pandas Documentation. (2023). *Pandas: Python Data Analysis Library*. Retrieved from https://pandas.pydata.org/pandas-docs/stable/
11. Docker. (2023). *Docker Documentation*. Retrieved from https://docs.docker.com/
12. PostgreSQL. (2023). *PostgreSQL Documentation*. Retrieved from https://www.postgresql.org/docs/
13. Flask-SQLAlchemy. (2023). *Flask-SQLAlchemy Documentation*. Retrieved from https://flask-sqlalchemy.palletsprojects.com/
14. GitHub. (2023). *GitHub: Code Hosting and Version Control*. Retrieved from https://github.com/

List of Figures

Acknowledgment

First and foremost, I would like to express my heartfelt gratitude to my supervisor, **Dr. Alwahab Dhulfiqar Zoltan**, for his unwavering support and guidance throughout this project. His expertise and insightful feedback have been instrumental in shaping this thesis and ensuring its successful completion. I deeply appreciate his encouragement and dedication, which have motivated me to strive for excellence.

I would also like to extend my sincere gratitude to my family for their unconditional love, patience, and constant encouragement throughout this journey. Their belief in me and their support during challenging times have been my greatest source of strength. I am immensely thankful to them for always being there, cheering me on, and providing the foundation upon which I have built my academic endeavours.

This work is a testament to the collective support and inspiration I have received from those around me.