SEENit!

INTRODUCTION

Aims and Objectives

The primary goal of Seen It! is to simplify the movie-watching experience for users by aiming to make the process of finding and watching films more enjoyable and efficient. The project's objective is to provide recommendations based on user preference and limit the amount of scrolling and indecisiveness by presenting options that align with their interests.
The project aims to offer an effective movie search mechanism, allowing users to find movies based on categories such as genres, decades and keywords.

Film Recommendations

- Develop the functionality to search for films based on different criteria - keyword, genre, decade
- Integrate the film-fetching API for the relevant information
- Provide details on the film for user exploration
- Display the different streaming providers the films can be viewed on

Front-end

- Implement a user-friendly interface
- Easy to navigate interaction for users

Testing

- Input validation to handle different input scenarios
- Unit test functions intended output
- Input error handling for issues identified in the testing stage

BACKGROUND

SEENit is a programme designed to transform the movie-watching experience for users by simplifying the film selection process.It is built to cater to movie enthusiasts, casual viewers, and anyone seeking a convenient way to discover films aligned with their tastes, without expending great effort to review and decide between a large volume of choice.

The application's robust film search functionality allows users to input various criteria to refine their search. Utilising The Movie Database (TMDB) API, the application fetches comprehensive information about films such as the synopsis, rating, visual poster and streaming providers. This comprehensive film information empowers users to make informed choices, enhancing their movie-watching experience, and feeling confident in the choices SEENit aids them in making.

SPECIFICATION AND DESIGN

Functional Requirements

Movie Search:

- Users should be able to search for movies by keyword, genre, decade
- Users can search for movies currently showing in cinemas (GB) using “now playing” functionality
- The search results should display relevant movie details

Movie Details:

- Users should be able to view details about the movie such as synopsis, rating, streaming providers and movie poster

Movie Genres:

- Users will be able to search for movies by genre from a dropdown list that will be populated directly using a TMDB API endpoint to populate genres

API Integration:

- The application should interact with TMDB to return movie data and images

UI:

- The interface offers an interactive user interface, tying together the user selected criteria and the movie recommendation API in the form of a digestible and accessible comprehensive programme

Non-Functional Requirements

Performance:

- The application should respond quickly to user interactions, such as filter selection and searching.
- API requests to TMDB should be optimised for efficient data retrieval and process.

Reliability:

- The application should be available and operational most of the time, with minimal downtime.
- Handle TMDB rate limits gracefully to avoid service disruption.

Usability:

- The user interface should be intuitive and easy to navigate.
- Ensure that users can quickly find the information they need about a movie.

Maintainability:

- Write clean, well-structured code to ease maintenance and future enhancements.
- Use version control to manage changes to the codebase.

Design and architecture

```
group2_project_git/
├── FrontEnd/
|   ├── index.html
|   ├── stylesheet.css
|   └── script.js
├── BackEnd/
|   ├── __init__.py
|   ├── app.py
|   └── utils.py
└── Tests/
    ├── __init__.py
    ├── test_app.py
    └── test_utils.py
```

The directory structure of the project has been divided into separate folders to ensure a tidy and logical structure is in place. The front end code is separated from the back end code, along with the tests, which are in their own folder. The main application code runs from app.py, with the front end using index.html.

The code has been designed to make use of classes and functions so that it is clean and easy to read, whilst

avoiding repetition. We utilise code separation by placing the logic in functions within a utils.py file and instantiating classes from within the app.py.

The front-end is crafted to offer an intuitive and user-friendly interface, facilitating seamless user interaction, employing the use of HTML, Javascript and CSS, which are combined with the use of the external movie API to offer a comprehensive view and experience.

IMPLEMENTATION AND EXECUTION

It was important to us to be able to produce a minimum viable product (MVP). As a group we decided what core functions the system should have and set out to develop it. Developing this type of “bare-bones” system would allow us to deliver our proof of concept, as well as have a MVP that could be expanded and built upon, if time allowed. This theory ensured we would not overstretch ourselves and be able to deliver a working product. In essence, the MVP would entail a landing page with search functionality for keyword, genre, and decades, using the TMDB API to return films that matched that criteria.

Conclusions from the SWOT analysis naturally allowed us to fall into roles we all felt comfortable with and matched our strengths. We have quite a large team (6) and so there was a broad depth, and breadth, of skills on offer. These skills ranged from front-end technologies (HTML, CSS, Javascript), back-end technologies (Python & API integration), database and SQL development, project management skills and agile working.

Although the front end is not considered in the marking scheme, we felt it would benefit the system to dedicate some time towards its development and ensuring the user experience (UX) was visually appealing and easy to use. We created wireframes and developed a front end using HTML, CSS & Javascript.

We made sure each member of the team contributed:

- 2 team members took on building the back end logic to return the MVP, creating functions and API calls to return the data
- 1 team member reviewed the back end code and refactored it to make use of classes and better design principles
- 1 team member spent time working on the front end, ensuring the design and functionality worked as we required, as well as assisting with code refactoring and class design
- 2 team members worked on building unit tests, giving us the confidence that the functions worked as we expected
- All team members contributed to writing the documentation required for the project

To adhere to good software practices, we implemented classes, refactored code and made use of a utils file to store the logic in functions.

Tools and Libraries

We used the following tools and libraries in our project:

- Version Control: Git and GitHub for collaborative version control and code sharing
- Backend: Python with Flask framework for building the backend API. Requests library for making HTTP requests to TMDB API
- Frontend: HTML, CSS, JavaScript for creating the user interface
- External APIs: TMDB (The Movie Database)
- Testing: Python Unittest module for writing and running unit tests
- Project coordination: Slack, Jitsi, Jira was leveraged for task tracking and collaborative documentation, though it was not used as much as we hoped

Agile development

While our team didn't adopt a strict Agile methodology, we used its most important features (collaboration, feedback, adaptability, iteration) and adapted it to fit the needs of our group and make sure the development of the project went smoothly.

We divided the project into smaller parts, reviewed and made sure we had a feature working before trying to add more functionality. We used feedback loops allowing us to change our strategy if needed. We revisited and refactored our code ensuring continuous improvement. We used tools like Jira to keep track of progress & tasks which helped with coordination and organising within our group. Slack was our main tool for communication and troubleshooting.

We would meet regularly via Jitsi, either as a group, or in smaller teams when paired programming. We met at least twice a week to share updates, address issues, and adjust our priorities. During our regular weekly Thursday meeting we would demo the work we had done and perform code reviews. When we encountered challenges, like the showtimes API, we showed flexibility and adjusted the functionality to focus more on movie data from TMDB.

Adaptability is the Agile feature that our team excelled at. Whether it was challenges with GitHub postings, scheduling meetings, or the unavailability of a usable API for showtimes, we learned from each challenge and adapted our approach.

Challenges

One of the biggest challenges was integrating the front and back end components, merging pieces of code from different team members and making sure they worked well together. As each of us worked on our respective areas, communication within the team and coordinating the development was very important.

Several other challenges emerged during the process, including difficulties with Git and GitHub. Instances of uncommitted and overwritten code, working with incorrect code versions, confusion regarding code selection, and the creation of numerous branches.

We encountered a scope change when the cinema listing API, initially promising free trials, only provided them to businesses intending to purchase their packages rather than for project usage. This was not stated on their website and only became apparent once contacted by an account

manager. Finding another free or budget external API for the showtimes feature that we initially wanted to integrate was difficult. As an alternative solution, we opted to expand our usage of the TMDB API. We identified a method to connect streaming providers with the retrieved movies which enabled us to showcase images of each streaming platform where the movies were accessible for viewing

As a team with varying schedules and commitments, like family, work, holidays, the CFG course and homework, finding common time to meet and coordinate tasks was not easy. We made sure that we updated each other via Slack on our progress and any changes or decisions that had been made.

Regarding technical challenges, we noticed some of our unit tests linked with external APIs were failing when the development was in the final stages. These errors required us to spend some time debugging in the last few days, but we discovered they were linked more to the format returned vs the format expected which we managed to resolve.

Another area of improvement for the future would be to optimise our application's response time when showing the streaming providers as well. Although it works, it's slower than hoped for due to making multiple API requests to update each movie.

TESTING AND EVALUATION

Testing strategy

We used the '**unittest'** testing framework to write and run our tests and the **'unittest.mock.patch'** decorator to mock API responses. We used PyCharm, an integrated development environment (IDE), to write and run our unit tests as it's a user-friendly interface for viewing test results and analysing code. We are aware of other tests that could have been useful such as integration tests. All tests can be found on GitHub.

Utils.py part of the code contains the core logic of our application. It has several classes and functions responsible for interacting with the TMDB API.

The test covers various aspects of the code functionality, including testing the behaviour of various classes. We included scenarios such as successful API responses, missing keys in API responses, exceptions during API requests, valid and invalid movies based on criteria, and handling multiple pages of API results. Additionally, the strategy accounts for edge cases like scenarios where the API returns no movies and when an invalid keyword is used for searching.

- Test Genres Class (TestGetGenres)
- Test NowPlaying Class (TestNowPlaying)
- Test valid_movie Function (TestValidMovie)
- Test MoviesByDecadeGenreKeyword Class (TestMoviesByDecadeGenreKeyword)
- Edge Case Tests for MoviesByDecadeGenreKeyword Class

App.py part of the code contains the Flask web application. It uses HTTP requests to interact with movie data through endpoints. The objective is to verify the behaviour of our Flask endpoints and make sure that the endpoints behave as expected and handle different input scenarios correctly.

1. **Unit Tests** for Classes and Functions in **app.py file**, write unit tests to directly test those functions for correctness.

- TestGetGenres
- TestDiscoverMoviesByGenre
- TestGetNowPlayingMovies

2. **Edge case tests** in **app.py file** that cover different scenarios and potential issues for the classes and functions in our code:

- Test_empty_response
- Test_valid_decade_no_movies
- Test_empty_response
- Test_missing_results_key
- Test_api_unavailable

3. **Error handling tests** are included in the **test_utils.py** file. These tests focus on ensuring that the code handles exceptions and error scenarios gracefully and correctly. The types of errors being tested for are as follows:

- TestMissingGenresKey
- TestRequestException
- TestNoMoviesReturnedByAPI
- TestInvalidKeyword

4. **Usability tests** test it from the customer's perspective. This is to ensure user friendliness and effectiveness of our application. The tests scenarios are following, more detail in **user_test.txt** file:

- Testing Genre Dropdown
- Searching by Keyword
- Filtering by Genre and Decade
- Error handling
- Checking Movie Details

System Limitations

1. Reliable on external API: this means if the structure is to change then that could affect the functionality of our application
2. API key security: this may be not secure if the code is shared publicly

3. Test coverage is limited: we are aware that the existing unit tests could be expanded upon to cover more scenarios, edge cases and errors
4. Performance and scalability: the application performance may be impacted depending on the number of requests made to the external API as well as gains in user and data volume
5. Compatibility with different browsers: need to test the application on different web browsers and mobile devices
6. User experience: need to conduct further usability tests to identify areas where users might find it hard to navigate the site

CONCLUSION

Collaborating as a team of six offered both advantages and disadvantages. On the positive side, we had a broad range of ideas and skills we could utilise and it helped having more people when tackling errors and more complex tasks.

The downside of working in a larger group involved the need for heightened coordination and communication. However, our teamwork remained harmonious, characterised by flexibility and mutual respect for each other's ideas. Importantly, our collaborative efforts on GitHub were conflict-free, highlighting our good communication and version control.

We felt like we could have benefitted from external support at the initial stages of the project, especially when it came to validating our ideas. We recognise that establishing firm requirements would be beneficial. The final product differs slightly from the initial idea as we encountered a blocker along the way with an external API we intended on using. This was flagged in a timely manner and we quickly came up with and agreed upon an alternative solution.

Despite the demanding nature of managing the project alongside coursework and daily obligations, we successfully delivered a functional MVP that brought us a sense of accomplishment. We are aware that further improvements and additional functionalities could be added in the future. We also recognize the value of incorporating better coding practices like the SOLID principles.

Given our beginner level experience in software development, we are content with the product we have developed and happy with the progress we have made since starting the course. We are pleased with how we worked collaboratively as a team and the shared experience that comes with it.