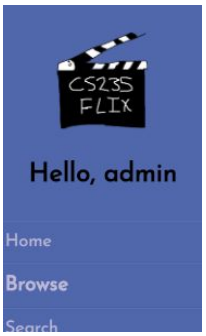
COMPSCI 235 - Assignment 2 Design Report

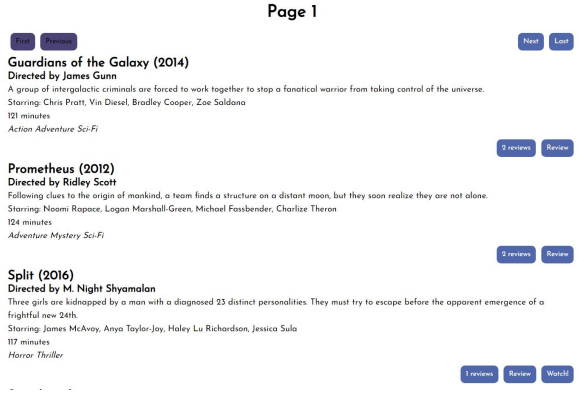
This project involved designing a Web application for the domain model that we built in the last assignment. As well as covering a lot of new ground that I was unfamiliar with, the brief guidelines were also very loose and gave us a laid-back view of what needed to be achieved, which meant it ended up being a relatively daunting task right from the get-go. Fortunately, there was a wealth of online resources available; the COVID news-site template was incredibly helpful in helping me in mentally constructing my path towards a finished app.



The ‘cool new features’ that I ended up implementing were a rough mixture of things. I started off by setting up a profile page which could be accessed by clicking on a greeting message (Hello, …) on the navigation bar. This, of course, could only be accessed when the user was logged in to their account, and the profile page itself was customised to the user currently logged in.

On the profile page, the user is able to see a list of their current friends which they have added on their account. This builds on the functionality of one of my extensions from the previous assignment, where I created a system that allowed users to add others as friends. If the recipient accepted the friend request, the idea was that they would be able to look at each other’s profiles which would contain various stats such as watched movies, total watch-time, reviews etc.. Unfortunately, I ran out of time to fully implement the above as I got side-tracked by my other new feature. The currently implemented functionality still works and is able to display the users friends from the given dataset onto the profile page - however, there is currently no method for adding users as friends outside of that.

The other new feature that I was eager to extend my app with was a movie-watching feature. With this feature, users are able to keep track of what movies they have watched so far. They can add new movies into their ‘watched’ list simply by browsing on any section of the site (i.e. through ‘Search’, ‘Browse’ or ‘Browse by tag’) and clicking a ‘Watch!’ button. In addition, the ‘Watch!’ button only shows up beside movies that have not already been watched, which was purposely integrated into the HTML in order to reduce any unnecessary or potentially confusing clutter within the visual interface. This is true for the existing data from the .csv files as well - the movies that already exist in the user’s watched movies list also do not have ‘Watch!’ buttons.



If I get more time further down the line, I'll probably look to extend the functionality of this feature by allowing users to remove entries from their watched movies, just to allow for more flexibility. The framework that I was able to build in creating this feature means that this kind of system could also easily be replicated for something like a movies-to-watch list as well, which could also go on the user's profile.

In terms of design patterns and principles, the main three that had the biggest influence on my decision-making were (unsurprisingly) the Repository Pattern, Dependency Inversion and Single Responsibility.

- Repository Pattern:
  This is the pattern that shaped my decisions on how to approach situations where I needed to work with data. Applying this pattern meant setting up both an AbstractRepository class as well as a MemoryRepository class which inherits from the AbstractRepository. By utilising this in conjunction with the Python 'abc' module to tag abstract methods of the AbstractRepository, it now becomes a lot easier to swap out the MemoryRepository for a database repository if it is ever needed further down the line. This helps us to develop our minimal viable product without needing to worry (at this stage!) about making it compatible with an actual database.

- Dependency Inversion:
  The Dependency Inversion principle focuses on abstractions. Namely, it outlines the need to follow the rule of avoiding having high-level modules depending on low-level ones. This links in with the Repository Pattern above, which seeks to enforce dependency inversion by creating an abstraction between a high-level and low-level module that both can depend on. I applied this principle to the CS235Flix project in a similar manner, by ensuring that the service layer was dependent on the abstract repository interface rather than the repository itself. As stated in the Repository Pattern, this allows us to have a lot more flexibility in terms of switching out database implementations with new ones.

- Single Responsibility:
  The Single Responsibility principle is all about having classes and/or modules each doing their own specific task without mixing and matching between different areas of the overall code. I feel that this had a big impact on my overall workflow. Throughout the whole process of creating this project, I came across many instances where I needed to go back and adjust the code in a different module. However, because I tried to follow the Single Responsibility principle previously by 'grouping' code into the tasks that they

were used for, it made it a lot easier to maintain it going forwards - any changes needed or made were often localised to one particular region. The use of Flask Blueprints also helped in following this principle, as it acted as a middleman between the view and service layers and maintained a separation of concerns between the two. I feel that the Single Responsibility principle is one of the most crucial software design principles to follow. As I've unfortunately learnt the hard way in my past coding experiences, code that is jumbled up and messy becomes infinitely harder to fix or adjust further down the line.