Billal Ghadie

100888260

COMP 2406B

Final Report

Websites such as IMDb, movies.com, and rotten tomatoes have become the primary sources for cinema lovers looking for information about movies. Indeed, there is a clear and reliable market for a service that allows users to search for, rate, and review movies. Furthermore, building such an application would provide excellent pedagogical experience for learning the fundamentals of web development. Therefore, for my final project, I chose to create and implement the movie database. What follows is a report detailing how I implemented the movie database as well as justification for my implementation and will be structured as follows. First, I will briefly discuss how to run and test the application. Then, I will outline all the movie databases functionality I have implemented and provide justification for my implementations. Limitations and possible extensions of my movie database will be presented. Finally, I will discuss my favourite features of my project.

To access my final project, you first need to find the open stack instant. The instant ID is BillalGhadie, the public IP is 134.117.128.84, the instant username is Student and the associated password is GrandCinema. Running the server is fairly straightforward. That is, everything should be already downloaded and installed in the open stack instance. As such, all that is required to run the program is to go to the "Grand Cinema" directory and, through the terminal, enter *npm start*. If an error occurs while trying to run the project, enter *npm install* in the command line then try *npm start* again. To streamline testing, I have set up two specific users in the database:

- Username: JeganPurushothaman
    - Password: Password123
- Username: BillalGhadie
    - Password: Password123

I suggest logging in under Jegan to test a majority of the functionality (note, Jegan is not a contributing user). That is, search for movies, people or users, add movies or people to the database, and follow or unfollow users. Of key importance, leave a few reviews on some movies and add "Martin Short" to the movie "Napoleon Dynamite" as a director. Once you have tested all the relevant functionality, logout of Jegan's account and login as BillalGhadie, who is actually following Jegan. As such, when you log into Billal's account you will receive an alert for every review Jegan wrote. Finally, to help streamline your testing of adding a movie to the database, you will notice an "auto generate" text field. This text field takes a number, *x*, and can be used to create *x* random movies and add them to the database (I recommend adding about 4-5 movies). Furthermore, I programmed it so the title of these randomly generated movies will be printed to the console upon creation. Therefore, you can simply search those titles to test if the movie has been properly added to the database.

Presently, my final project has incorporated all the requisite functionality outlined by the movie database project document. Specifically, functionality pertaining to users, people[1], and movies. Moreover, in addition to the web-based client, I have implemented public JSON REST API that support a wide variety of routes. What follows is a summary and justification of all of the implementations and a brief conclusion wherein I highlight my favourite aspects of my final project. For pedagogical reasons, I tried to limit my use of modules as much as possible. Therefore, any module used in my final project was included for a specific reason. The express module provides the backbone for my entire server. The pug module dynamically renders webpages served to the client. Finally, the express-session and cookie parser modules were used

[1] A term I will use to refer to celebrities in the present document

in tandem to provide session functionalities in my project as well as authorize specific actions of a logged in user.

First, my final project allows for users to create accounts or log into their account. To improve the robustness of this functionality, regular expressions were used for password validation. Furthermore, upon logging in, a session cookie is created to maintain the integrity of the user's session. Once logged in, users can manage the people and other user's they follow by vising that user or person's page and pressing the follow/unfollow button. They can change from a regular user, the default for new users, to contributing user by pressing a checkbox. I chose a checkbox so anybody visiting the user's page can tell if they are a regular or contributing user. For an example, search for the user Joshua. Contributing users may add a movie or person to the database or add a person to a movie. Finally, the user can view reviews they have made, view a list of recommended movies based on the reviews, and be notified when a user they are following leaves a review.

The biggest design decision I made with respects to user accounts was how to keep track of what user was visiting what page. Specifically, I had to ensure that for any given webpage, content changed dynamically depending on if the user was logged in, viewing their own or someone else's account, and if they were a regular or contributing user. Therefore, I decided that the key for each user object would be the user's username, and I stored the username in the session cookie. As such, keeping track of users became trivial, I only needed to program a few server calls to check the database for the relevant attributes associated with the username stored in the session. For example, if I needed to check if a user was a contributing user or not, the server call could use the stored key to quickly access the user object's data and respond with the appropriate status code. Checking for user information occurs multiple times when loading

pages, thus, this implementation ensured that there was little latency in the load time of each web page.

With respects to people, my final project allows for viewing people, following or unfollowing people, and adding people to a movie or the database proper. Presently, the implementation for viewing people is fairly straightforward. First, people can be searched through by using their name or unique ID. I created the ID's when putting the database together and they consist of integers ranging from 100000 to 1039845, inclusive. If an ID or a name that matches someone in the database is searched, the user will navigate to the person's view page. Alternatively, they can select the name of the person they searched from a list of people who match the search criteria. The person's page consists of a list of frequent collaboration and recent work. To ensure that both lists capture any changes to the database that might alter their content (i.e., the person gets added to a more recent movie than their current work), the lists are created dynamically each time. Specifically, as each person object stores a list of all movies they have worked on, a list of all their work and a frequency count of who they have collaborated can be made quickly, then using a quick sort algorithm, the list is sorted in descending order by either year the movie was released or the number of times the person collaborated with someone. As such, I only need to display the first ten entries in each list. Of key importance, the use of quicksort significantly reduced the latency of loading people's pages while simultaneously ensuring the robustness of the design. Furthermore, each collaborator name and movie title displayed this way is a clickable link that will navigate the user to the person or movies respective page.

In addition, the users will also have the option to follow or unfollow a person by simply pressing a button. Once the button is clicked, calls are made to the server to update the user's

account and, in turn, the button will toggle to now allow users to unfollow that person. Finally, adding a person to a movie or to the database is straightforward. The server first checks if a logged in user is a contributing user, and if they are, the button that allows for adding people to the database and the button that adds people to a movie become enabled. With respects to the former, the user need only specify a person's name, who does not already exist in the database, and their role (i.e., actor, director, or writer). Adding the person to the database involved simply updating the movie and people database. In terms of the later, a person who currently exists in the database can be added to a movie by specifying their name, they must already exist in the database, as well as their role.

In terms of movies, users can search for any movie by specifying any combination of a title, genre, year of release, mini-rating, or ID. ID's are represented as integer values ranging from 0 to 9124, inclusive. If the users enter a valid ID, they will be directed to the specified movies view page. Otherwise, a list of all movies that match the search criteria will be displayed and users can navigate to the desired movie page by pressing the movie's title or poster. Once on the movie's page, users can see details about the movie. For example, the title, summary, cast, director(s), writer(s), and a list of all reviews. Every keyword (i.e., genre, actor, direction, etc.) on this page is a link that will redirect the user to a webpage that matches the keywork they selected. For example, if they select a genre, they will be redirected to a list of all movies that match that genre. Furthermore, if a user is logged in, they can leave a review of the movie. Reviews are displayed on the lower portion of the screen horizontally. I chose this for aesthetic purposes as each review may be long, therefore, I thought it better to display them all in a single, scrollable row. You will notice that I populated each movie with a random number of short reviews from user's already in the database. In previous versions of my project, I had used full

reviews (i.e., including a score out of ten, a summary, and a review) using Lorem Ipsum. However, I found that the additional nonsense text to be an eyesore and distraction from the actual content of the webpage, therefore I decided to only use short reviews to demonstrate the functionality while maintaining the user enjoyment of the page.

In my initial implementation, anybody could leave a review for any movie. However, I decided against this for two reason. First, I believe making users log in or create an account before leaving a review facilitates user enjoyment. That is, it gives the users an opportunity to be an active participator in the website, helping to build the community of reviews. Moreover, my database will keep track of information about a user, such as movies they like and dislike, and will then be able to recommend similar movies the user may enjoy through my recommendation algorithm (discussed subsequently in the extensions section). The second reason for this implementation choice, was to maintain the website security. Currently, each user can enter a single review to each movie. In fact, past reviews will be overwritten if a user attempts to another review to a movie they have already reviewed. Therefore, only authorizing logged in users to add a review completely circumvents the addReview request's vulnerability to a denial-of-service attack in my project.

The final notable feature when viewing a movie allows users to add a person to the movie object. The implementation of which, was trivial as I wrote code in a manner that facilitated reusability. That is, all the relevant functions are in a separate source file and when the server makes a call to, for example, add a person to a movie, helper functions are subsequently called to validate the input, create the person object, and updated all databases. This bolsters the scalability of my databases, which is an acknowledged weakness in my final project (discussed subsequently).

The final design implication for the final project was to create a public REST API that supports a number of different parameters. As such, my API supports requests that gets a list of movies based on any combination of the following query parameters: title, genre, year, and mini-rating. Alternatively, a request can be made with a specific movie id (i.e., "/movies:id") that will return specified movie JSON object that contains all the relevant movie information. Furthermore, a post request can be made that will accept a JSON representation of a movie, will validate the movie information, then add it to the movie database. As per professor Dave McKenny's instructions, anybody can go to the "/addMovie" page and attempt to add a movie to the database. However, to maintain the integrity of my database, only logged in contributing user's will be able to actually add a movie to the database. If anybody else attempts to, an error message will be displayed asking the user to upgrade to a contributing user to use this feature.

Beyond movies, my API also supports getting or posting people objects. With respects to the former, users can search for a person by using their name or ID (i.e., "/people:id") parametric routes. If a valid ID is entered, a person object containing the details of the person who matched the provided ID is returned. Or, a list of people who match the provided name parameter will be provided and the user can select which person to view. If the user would like to add a person to a movie or the database, a simple post request can be made, provided they are a logged in contributing user. As with movies, a series of function and helper function calls are made following an authorized post request to perform the adding operation which ensures that the database is as scalable as possible. Finally, similar parametric routes are available that get a user or list of users that match a search ID or criteria, respectively. Furthermore, users can only be "logged in" following validation from the server. This includes checking that no user exists with the given username, provided the user is creating the account. Or, that a password matches an

existing account in the database. Furthermore, a regular expression was used to validate the password, this ensured that a password met a minimum level of security. Moreover, creating a user was done through a post request to try and prevent malicious intenders attempting to hijack an account.

For my final project, I did not want to simply implement the minimum requirements outlined in the movie project document. Rather, I thought it would be an excellent pedagogical experience to try an apply an extension to the project, that supplement my skillset. Specifically, I decided to implement the Content-Based recommender outlined by Le (2018) that relies on the similarly of items being recommended. My implementation centers primarily around concepts of Term Frequency (TF) and Inverse Document Frequency (IDF) which count the frequency of terms in a piece of datum while controlling for high frequency words such as “the” and “of” respectively. A TF-IDF score can be conceptualized as a vector in an n-dimensional space. The distance between two vectors can be viewed as a measure of the degree of similarity between two movies. Therefore, taking the cosine of the two vectors, termed cosine similarity, can be used to determine which movies are close together and thus, similar (Le, 2018). As such, the algorithm uses similarity scores to determine how similar a given movie is to all other movies in the database. Consider lines 57-58 in movieRecommender.py. Note, that an “amalgamate” column is created that contains a simple string consisting of the director names, actor names, writer names, and genre key words. Furthermore, I gave directors more weight in affecting similarity scores by adding it three times to said string. For each movie, this amalgamate vector was used to compute a TF-IDF score, which as a reminder, is a frequency count of each word in the string that simultaneously controls for high frequency words. Then, in the movieRecommender function, a cosine similarly matrix is computed using these scores as

vectors. Entries in the matrix are then turned into a list and sorted in descending order, and the top ten most similar movies are retrieved.

With a movie recommender algorithm made, I simply needed to wait for information from the users. Specifically, for a user to review a movie. Once a movie has been reviewed, my algorithm checks if the user enjoyed the movie (i.e., they rated a movie above a 7), and if they did, it recommends movies similar to the reviewed on to the user. Seven was chosen as an arbitrary, conservative, estimate of user enjoyment, as no qualitative work was done to validate that seven is the threshold for "enjoyed". Eventually, as more reviews are made, the top five most highly rated movies will have their most similar movies recommended to the user. If a user does not have any reviews or a have not enjoyed a movie highly enough, my database will simply recommend the highest rated movies in the database to the user. Taken together, this feature maximizes the user experience as a personalized selection of recommended movie is created and updated dynamically as more information is collected in the user profile stored in the database.

Despite this, my final project is not without limitations. Specifically, my project lacks in scalability as all the databases are stored in RAM and thus, changes affected by user's are not permanent. Furthermore, my recommendation algorithm is strictly Content-Based and thus cannot recommend items to user's outside of their content profile (Le, 2018). With respects to the former, after implementing all the functionality highlighted in this document, I did not have adequate time to transition into using an actual NoSQL database such as MongoDB. Using an actual database would improve the scalability of my final project as all my databases would be stored in long term storage and thus, adding, removing, and updating entries in my database would become seamless. In terms of my movie recommendation algorithm, although functional,

it fails to capture specific variations in movie preferences across users. For example, consider two users A and B. User A enjoys movies 1, 2, 3, and 4 while user B enjoys movies 1, 2, and 3. A more sophisticated algorithm that incorporates collaborative filtering would be able process this information and recommend movie 4 to user B, as they share the same interests as user A (Le, 2018). Alternatively, additional quantitative data, such as average IMDB scores, could be used to teach the algorithm which movies to recommend (i.e., recommend movies with higher scores over lower scores). Once again, the time constraint prevented me from improving my recommendation algorithm in these manners. However, I fully intend to incorporate these two features over the extended holiday break this winter as it would dramatically improve the scalability, robustness, and user experience of my project.

Without a doubt, the feature that I am proudest of is my movie recommendation algorithm. While completing my masters in experimental psychology, I gained a lot of experience in data science, however, this experience was limited to an in-lab context[2]. In fact, my actual master's thesis outlined a similar concept to cosine similarity, called Euclidean distance which determines how similar two vectors are by measuring the actual distance between them in nth vector space. Notably, Euclidian distance is analogous to cosine similarity, except, it can be used to account for quantitative data (for full review see; Qian, Sural, Gu, & Pramanik, 2004). Completing this project gave me the opportunity to actually apply my knowledge as in a real world setting as, like everyone else, I enjoy movies. Therefore, researching and implementing a functioning recommendation algorithm that I have actually used to some success was extremely fulfilling[3]. The other thing I am most proud of is the general design of my

[2] Typically, this involved testing complex (and in my opinion, bogus) models that describe specific human behaviour.

[3] 12 Monkey's was a fantastic movie

webpages. Although not perfect aesthetically speaking, functionally it was designed in a way that simplifies the user interface tremendously. For example, on any page the user can press a link to return to the home page or press a button to login/create an account or return to their account. Furthermore, the location of both the link and button is consistent throughout all my webpages. Taken together, I am extensively proud of my work and what I was able to do alone, in such a short amount of time.

Reference

Le, J. (2018, June 11). The 4 Recommendation Engines That Can Predict Your Movie Tastes. Retrieved from https://towardsdatascience.com/the-4-recommendation-engines-that-can-predict-your-movie-tastes-109dc4e10c52

Qian, G., Sural, S., Gu, Y., & Pramanik, S. (2004, March). Similarity between Euclidean and cosine angle distance for nearest neighbor queries. In *Proceedings of the 2004 ACM symposium on Applied computing* (pp. 1232-1237).