Safwan Wadud

Project Report

Table of Contents

1. Start-up Instructions

1.1. Database setup

1. Open terminal and navigate to the directory containing all the source code and resources
2. If a folder named “database” is not there, create one
3. Start up the mongo daemon by entering the command:

mongod --dbpath=”/path-to-database-folder-from-step-1.1.2”

1.2. Database initialization

1. Open another terminal window and avigate to the directory containing all the source code and resources
2. Initialize the database by entering the command: *node db_init_script.js*
    a. Note: this might take a few minutes

1.3. Instructions to run server

1. In the same terminal window as used in step 1.2, enter the command: *node server_db.js*
2. Incase of server crash: repeat steps 1.2.2 - 1.3.1

2. Functionality Implemented

2.1. User Account

- Users can **create new accounts** by specifying a username and password on the login page and then clicking the sign-up button. On the client side, I ensured that no fields were left blank. Once the button is pressed and all fields are filled up, an ajax response is sent to the server. On the server, I make sure that the username is unique by checking if it exists within the database. If the username is unique, then a new user is created and saved

to the database. A username property is then added to the session object and is set to the username entered by the client. This ensures that only a single user is able to be logged in at one time within a single browser instance. Users are also able to log in with an existing account by entering their credentials and clicking the sign-in button.

- Users can **change between a 'regular' user account and a 'contributing' user account**. This can be done so by navigating to their profile page and switching between two radio buttons. If a user changes account types, it only affects their ability to carry out an action in the future. That is, anything created by a user while they have a contributing user account remains unaffected if the user switches back to a regular account.

- Users can **view the people and users they follow** on their profile page. They can also navigate to the personal page of any person or user they have followed by clicking on their names. Users can decide to stop following any person or user by clicking the unfollow button near the top-right of the person's or other user's page.

- Users can **view recommended movies** on their profile page. The recommendations are made based on the movies that they have reviewed. The algorithm that I used to determine recommended movies is to select a random review made by the user and then pick out 3 movies similar to the one they reviewed. Similarity was determined by selecting 2 random genres from the movie that they reviewed and finding movies that contain those 2 genres. I also ensured that the user does not get recommended a movie that they already reviewed. If a user has not made a review yet, then 3 random movies will be displayed. In addition, users can view reviews made by themselves on their profile page.

- On the search page, users can **search for movies, people, or users** by specifying the search type in the dropdown menu. One can search for movies by typing in a movie title and/or clicking a genre and then clicking the search icon button. One can search for people or other users in a similar manner. Search results show a max of 50 items on a single page. Users can use the next/prev buttons found near the top and bottom of the page to show the next/prev 50 results. Users are also able to navigate to the movie/person/user page for any of the search results by clicking on the result's picture or name.

- Note: the database has 2 initial users with predefined data. The first user has the username *Safwan* and password *123* and the second user has the username *Wadud* and password *456*. One can use these accounts to test the application if they wish.

2.2. Viewing Movies

- When viewing a particular movie's page, a user is able to **see movie information**, including the title, release year, average rating, runtime, plot, poster, language, country, awards, genre keywords, directors, writers, and actors.

- A user can click on the **genre keywords** which will bring them to the search page, displaying search results that contain movies with that genre keyword. In addition, a user can click on the **director, writer, and actors** of the movie which brings them directly to each person's page.

- A user can see a list of **similar movies** to the one they are viewing and can navigate to the page for any of those movies. The algorithm that I used to determine similar movies was to select 2 random genres from the current movie and find 3 other movies that contain those 2 genres. I also ensured that the 3 new movies do not include the current movie.

- A user can **view reviews** that have been added for the movie near the bottom of the movie's page. A user can also **make a review** themselves by either adding a **basic review** by specifying a score out of 10 on the dropdown list or by adding a **full review** by specifying a score out of 10, a brief summary, and a full review text.

2.3. Viewing People

- When viewing a particular person's page, a user is able to **see basic personal information**, including their name, roles, picture, and bio.

- A user can see a list of **frequent collaborators** of any person. The algorithm used to determine this was to first gather a list of all the movies that the person has been a part of. Then, the number of times that the person has worked with every other person within the domain of the list of movies initially gathered was stored using a map/dictionary. Finally, the frequency dictionary was sorted and then the top 5 results were returned.

- A user can also choose to **follow any person** by navigating to the person's page and clicking the follow button found at the top-right corner of the page. If a user follows a person, the user **receives a notification** any time a new movie is added to the database that involves this person, or any time this person is added to an existing movie. The notification can be viewed on the home page under the News section.

2.4. Viewing Other Users

- A user can see a **list of all of the reviews other users have made** near the bottom of other users' pages. This includes being able to **read each full review** made by the other users.

- A user can see a **list of all the people and users that other users have followed** on the left side bar within the other user's page. One can also navigate to each person's or user 's page that other users have followed by clicking on their names.

- A user can also choose to **follow any other user** by navigating to the other user's page and clicking the follow button found at the top-right corner of the page. If a user follows another user, the user **receives a notification any time the other user creates a new review**. The notification can be viewed on the home page under the News section.

2.5. Contributing Users

- Contributing users can add a new person to the database by specifying their name and bio by navigating to the contribute page and filling in the form found at the bottom of that page. If the name of the person that the user is trying to add already exists, the person is not created.

- Similarly, contributing users can add a new movie by specifying at minimum, a title, runtime, year, plot, 2 genre keywords, director(s), actor(s), and writer(s). If a person is trying to add a new person to the movie, the person will be created and then added to the new movie. When specifying actors, directors, or writers, I ensured that users are not able to enter the same name more than once for each category. In addition, the person names are not case-sensitive which makes it easier for the user to add existing users.

- Note: if a regular user tries to access the contribute page, it will simply redirect them to their profile page to then allow them to decide if they want to switch to being a contributing user.

2.6. REST API

- **GET /movies** – Allows searching for movies in the database. Returns an array of movies that match the query constraints. Supports the following query parameters:

    - **title** - A string that is considered a match for any movie that has a title containing the given string title (ignores case). If no value is given for this parameter, all movies match the title constraint.
    - **genre** - A string that is considered a match if the movie's list of genre keywords contains the given keyword (ignores case). If no value is given for this parameter, all movies match the genre constraint.
    - **year** - A number that is considered a match if the movie's release year matches. If no value is given for this parameter, all movies match the year constraint.
    - **minrating** - A number that is considered a match if the movie's overall average review rating on my site is greater than or equal to the given value. If no value is given for this parameter, all movies match the year constraint.
    - **page** - A number that represents the index at which results should be matched. For example, if page = 1, then the first 50 results will match, if page = 2, then the next 50 results will match, etc. If no value is given for this parameter, only the first 50 results will be matched.
- **GET /movies/:movie** - Allows retrieving information about a specific movie with the unique ID movie, assuming it is a valid ID.
- **POST /movies** - Allows a new movie to be added into the database. It will accept a JSON representation of a movie and is responsible for checking data is valid before adding it to the database. Your documentation should specify the required data and expected format. If a person is specified as a writer/director/actor within the movie but does not currently exist in the database, a new person should be created and added to the database.
    - Required data:

```
{
   "title": String,
   "releaseYear": Number,
   "runtime": Number,
   "genre": [String(list of genres can be found in the genre.JSON file)],
   "directors": [String],
   "writers": [String],
   "actors": [String],
   "plot": String
}
```

    - Expected output:

```
{
   "averageRating": 0,
   “rated”: “Not Rated”
   "genre": [String],
   "directors": [{“_id”: Schema.Types.ObjectId, “name”: String}],
   "writers": [{“_id”: Schema.Types.ObjectId, “name”: String}],
   "actors": [{“_id”: Schema.Types.ObjectId, “name”: String}],
   “language”: “”,
   “country”: “”,
   awards: “”,
   “Poster”:
"https://icon-library.com/images/no-photo-icon/no-photo-icon-28.jpg",
   reviews: [],
   similar: [{ poster:String,  _id: Schema.Types.ObjectId, “title”:String}],
   “_id”: Schema.Types.ObjectId,
   "title": String,
   "releaseYear": Number,
   "runtime": Number,
   "plot": String,
   “__v”: Number
}
```

- **GET /people** - Allows searching for people within the movie database. This supports an optional name query parameter. If the query parameter is included, it returns any person in the database whose name contains the given name parameter, in a case-insensitive nature.
- **GET /people/:person** – Retrieves the person with the given unique ID, if they exist. This includes the name of the person and the movies they have been a part of.
- **GET /users** - Allows searching the users of the application. This supports an optional name query parameter. If the query parameter is included, it returns any user in the database whose name contains the given name parameter, in a case-insensitive nature. If the parameter is not specified, any user will match the search constraint.
- **GET /users/:user** - Get information about the user with the given unique ID, if they exist. This contains their username, their account type (regular or contributor) and the reviews the user has made

3. Extensions Included

3.1. Database

One of the extensions I worked on was to incorporate a database within my server using mongoose. Interacting with the mongo daemon through mongoose was done for all functionalities of the application with the exception of sessions (which was done using the express-session module). Incorporating Mongoose into my project involved storing 4 collections: movies, users, people, and reviews. The movies and people collections store documents about each individual movie and person provided in the movie json file as well as additional movies and people created through my app. Similarly, the users and reviews collections store documents for every user and review in my app. Within each document, specific information relating to an individual movie/user/person/review is contained. I also created schemas for each collection that contain properties about each collection as well as having some instance and static methods. Using a database also required refactoring a lot of my code which resulted in having routers for the /movies, /users, and /people paths as a way to modularize my code.

A database initialization script is also included. The script takes the data contained in JSON files and stores them into the database. In doing so, it provides the application with initial data as well as providing a way to reset the database in the event where the data might become corrupt.

3.2. Responsive web app

Another extension I worked on was to support a full-sized/desktop interface, as well as a smaller, mobile-friendly interface. How this was done was primarily using flexbox css properties. I tried to incorporate some form of flexbox usage on all the pages. The layout of some pages responsively change by growing/shrinking elements or by wrapping them. Also, most text is responsive by using the word-wrap:break-word css property. However, this extension was not worked on too extensively so there are areas where the page does not respond well to changes in size.

4. Design Decisions

4.1. Database

Deciding to change my server from one that stores all the data in ram to one that stores them in a database was a key design decision I made that I feel like drastically improved my program. Incorporating Mongoose into my project provided several advantages. One

advantage is that it can handle concurrent access. This allows for my server to access the same or different info within the database at the same time without causing issues. For example, when updating the state of multiple users, the asynchronous nature of mongoose operations, such as save, allow for my server to update all the users at the same time.

Another reason why using a database improved the overall quality of my program was that it significantly increased its robustness. The key to this was by making use of mongoose schemas. Automatic validation is handled through Schemas instead of having to manually handle leaves less room for error. For example, mongoose gracefully handles automatic type conversions so that situations where comparisons like 1 vs "1" occur, can be done so without handling these edge cases manually. In addition, it cuts down the amount of code required for validation as it would be handled automatically.

4.2. Pagination

Incorporating pagination into my program increased its scalability as well as improving user experience. By only sending a limited number of resources each time, it makes it fast and easy to work with a wide range of data sizes, since only a constant number of resources is transferred for every GET request for /movies, /people, and /users. On the web application, having pagination improved user experience by not overwhelming the client with large-sized data.

4.3. Referencing Objects by ID

I made sure that every object that had references to other objects, only stored IDs of those references and not whole objects. In doing so, it makes it more scalable as large amounts of data can be processed much quicker. In addition, when rendering things such as recommended movies and similar movies, I would only populate select fields of each objectID before sending them back as a response as json or through a template engine instead of sending all of the values which again makes it more scalable since only as much data that is needed is transferred over.

5. Improvements

5.1. Status codes

Although I used appropriate status codes for every response (200 for success, 401 for unauthorized, 404 for unknown sources, 500 for server error), I feel as though they were not precise enough for the various responses I needed to send within my program. In the future, I would take a look at the documentation for status codes in order to be more familiar with when to

use the appropriate status codes. In doing so, it would improve the interaction between the server and the client which would then lead to a higher quality application.

5.2. Handling Errors

Another area I feel I could work on is the way that my program handled errors. In most cases, in the event of an error, I would send the cause of the error as a json to the client. However, for my web app, this would display a new blank page with the error message which isn't ideal. The reason why this happens is because I used forms on my website instead of AJAX requests to send POST requests. If I had used AJAX requests, I would have been able to handle error messages better by displaying the message as an alert on the current page. By changing these forms into AJAX requests, it can improve the user experience of my program.

5.3. Modularity

A major improvement that can be made to my system would be to increase its modularity. As it is right now, there are places in my code where I can see can be separated into other functions and even separate files. For example, In all of my routers, there is no separation between the server code and the business logic. This resulted in having middleware functions that would be longer in length than necessary. By separating my business logic from my server code, it would drastically clean up my code, making it more readable and modular.