Aaditya Chopra, Erica Li

COMP 2406 FINAL PROJECT

Project Report

TABLE OF CONTENTS

1. How to install, initialize database, and run server

a. Option 1:

i. Make sure your mongo daemon is running and listening on port 27017.

ii. To install the dependencies and start the server, simply enter the *npm run clean-start* command. This command will install the node modules, run the database initialization script, and then start the server.

iii. Once the ‘server is listening at http://localhost:3000’ message appears, then the login page can be found by entering http://localhost:3000 in the browser of choice.

iv. To only run the server (once the dependencies are installed), simply enter the *npm start* command.

v. Use Ctrl-C to stop the server.

b. Option 2:

i. To install the dependencies, enter the *npm install* command.

ii. Make sure your mongo daemon is running and listening on port 27017.

iii. Enter ‘*node database_initializer.js*’ into your command line to initialize the mongoose database.

iv. Once the node modules are installed, enter the *node app.js* command to start the server. Once the ‘Server listening at http://localhost:3000’ message appears, then the login page can be found by entering http://localhost:3000 in the browser of choice.

v. Use Ctrl-C to stop the server.

c. Login Credentials:

i. To login as a **non-contributing** user, use the following credentials:
   1. Username: userPogging1
   2. Password: password123

ii. To login as a **contributing** user,use the following credentials:
   1. Username: poggerUser20
   2. Password: verysecure

iii. The sidebar allows you to navigate to the different routes on our website.

2. Additional features beyond the project requirements

a. Feed posts widget on the logged-in user's home page

i. It differs from notifications because it displays the information in a timeline layout, and it shows the exact day a movie was released/review was added. Meanwhile, the timeline simply displays how long ago the event happened. This is part of our effort to make an intuitive and easy to use UI.

b. Marking notifications as read or unread

i. We added a “read” boolean attribute to the notification objects, defined in the notificationModel schema. This would allow users to see which notifications are read and unread. if we had more time we would have fully implemented the UI and added the red dot beside unread notifications.

c. Some additional steps we could take are:

i. Deploy the project on heroku. This would allow anybody to access our website from any computer. This would also allow for scalability as more users sign up and more movies/persons/reviews are added.

ii. Divide the app.js server code up into routers and a main script that calls these routers. This would improve the code design and allow for an easier time adding new routes because of better organization.

3. Critique of the overall design

a. What we did well:

i. Our **RESTful design** was good. We have a server that responds to a variety of GET, PUT, and POST requests with appropriate status codes and responses. The client side scripts make appropriate AJAX requests to the server for things like adding movies, adding reviews to movies, adding persons, adding users, etc.

ii. Our **data model** is something else we did well. We slightly modified it from the project checed model because we implemented mongoose databases, so the objects now have built-in mongoose ID objects as attributes rather than simple integers (from before). We also made our data model more dynamic by adding similar movie and recommended movie algorithms that allowed the respective fields of movie and user objects to be populated appropriately.

iii. Our **database implementation** is good. We added mongoose schemas for movies, persons, users, feed posts, notifications, and reviews. We made a database initializer script. This allows for flexibility because we can easily update the objects (i.e. add a similarMovies field to the movie objects, add a recommendedMovies field to user objects), and add/remove objects more robustly.

iv. Our website also has a good **UI design**. We carefully crafted the colour scheme and made mockups of each page (i.e movie page, person page, home page, etc). We took the user experience into consideration and implemented responsive design, such as flexible window sizing using media tags or changing colours of clickable elements on hover. We added a nice navigation bar that also uses responsive design. The design aims to be friendly for mobile-sized devices. The add movie and add person pages have dropdown menus that display results dynamically, and in real time, when the user types in a director, genre or actor name (similar to the way Google displays search results).

v. Proper **error handling**: for invalid requests (i.e the user queries for a movie ID that doesn't exist), the server sends a response that says "Sorry, we couldn't find that resource!" We also make sure to throw the errors if they occur, in the mongoose query callback functions.

vi. Use of **asynchronous operations**: We effectively used callback functions, async and promises keywords to handle AJAX requests. This allows us to perform network requests without blocking the main thread, which is very important because javascript is single threaded.

vii. **Scalability**: we implemented login, logout, and signup functionality. This means that new users can be added and saved into the users database by accessing and saving session data. We also have separate models for each type of object, so when we add new objects there's less dependency between them.

viii. **Efficiency**: The top 5 collaborators for a person are displayed in random order. This was done on purpose into order to improve efficiency and runtime, since sorting them again would be an O(n) operation.

b. What we can improve:
   i. Our algorithms for movie recommendation and similar movies are rather simple.
   ii. Our app.js file is rather long and convoluted, because it contains ALL of the route handlers.
   iii. Latency isn't something we particularly paid attention to while working on this project. For the current scope of the project, this is not much of an issue, however if the app and databases expand then latency is something for us to consider.

c. How we can achieve these improvements:
   i. We can improve them by looking at multiple factors, such as common actors or directors, to create a more accurate result.
   ii. We can divide up our server into server routers, such as movie-router.js, person-router, user-router.js, etc and then export the router objects so we can access it in the base app. This would create a more organized server structure.
   iii. There is an *express-ping* module we can add to the node dependencies, that will add a /ping endpoint to our routes. This can be used to test for latency of our database queries.

4. Description of movie recommendation algorithm

a. Iterate through all reviews of the logged-user:
   i. If the rating is greater than 6: push the first genre of the movie that the review belongs to into a genres array
b. Iterate through the watchlist of the logged-in user:
   i. Push the first genre of each movie into the same genres array
c. Use the mongoose query Movie.find() to find all movies with genres belonging to the genres array
   i. If the results array length is greater than 10:
      1. Set the currentUser.recommendedMovies field equal to the the first ten results by array slicing
   ii. If the results array length is lesser than or equal to 10:
      1. Simply set the currentUser.recommendedMovies field equal to it

5. Description of similar movie algorithm

a. When we query for a specific movie, we take the first element in the movies's genres, which is stored as a string array in the database.
   i. Then we query the database using Movie.find() for that specific genre

1. If the results array length is longer than 5:
    a. We slice the first five elements and set the similarMovies field equal to that.
2. If the results array length is 5 or less:
    a. Then we simply set the movie.similarMovies field equal to it.

6. Description of top 5 collaborators algorithm

a. Define an object to be used as a hashmap, where the key is a person ID and the value is the frequency
b. Query for a specific person
    i. For each movie in the Person.directors array:
        1. Iterate through the actors array and put the frequencies into the object
        2. Iterate through the directors array and put the frequencies into the object
        3. Iterate through the writers array and put the frequencies into the object
    ii. For each movie in the Person.actors array:
        1. Iterate through the actors array and put the frequencies into the object
        2. Iterate through the directors array and put the frequencies into the object
        3. Iterate through the writers array and put the frequencies into the object
    iii. For each movie in the Person.writers array:
        1. Iterate through the actors array and put the frequencies into the object
        2. Iterate through the directors array and put the frequencies into the object
        3. Iterate through the writers array and put the frequencies into the object
    iv. Set the Person.top5 field equal to the five IDs with the highest frequencies