2406 Project Report

Simon Han & Musa Taha

TABLE OF CONTENTS

Object Models

```
Movie {
        Title: string,
        Year: string,
        Rated: string,
        Released: string,
        Runtime: string,
        Genre: [string],
        Director: [string],
        Writer: [string],
        Actors: [string],
        Plot: string,
        Poster: string,
        reviews:  {Review obj},
        id: string
}

User {
        username : string,
        Password: string,
        movieList: {Movie obj},
        followedPeople: {People obj},
        followedUsers: {User obj},
        reviews : {Review obj},
        isContributor: boolean,
        id ; string,
        notifications: [notification Obj],
        recMovieList: {Movie obj}
}

Person {
        name: string,
        id: string,
        Acted: [movie Obj],
        Directed: [movie Obj],
        Wrote: [movie Obj],
```

}

Review {
    movieId: string,
    movieTitle: string,
    id: string,
    uid: string,
    username: string,
    text: string,
    summary: string,
    rating: integer,
}

notification {
    type: string,
    name: string,
    id: string,
    movieId: string
}

RESTful Design Outline

/movies
- GET /movies/:id
- GET /movies/?
- GET /movies/genre/?
- POST /movies
- POST /movies/:id/reviews

/people
- GET /people/?
- GET /people/:id
- POST /people

/user
- GET /user
- GET /user/:id
- POST /user/:id/movieList
- POST /user
- POST /user/:id/followedPeople
- POST /user/:id/followedUsers
- DELETE /user/:id/followedPeople/:pid
- DELETE /user/:id/movieList/:mId
- DELETE /user/:id/followedUsers/:pid

Other

GET /
GET /profile
GET /account
GET /logout
GET /contribute
GET /search
POST /login
POST /signup
POST /account/regAccount
POST /account/contributor

Specifications

/movies

GET /movies/:id

- HTTP Method: GET
- Description: Render a pug file displaying information for the movie with the specified id
- URL parameter:
  - id: a movie's unique id, which will be used to search the collection of movies for the requested movie
- Response Data Types:
  - application/json
  - text/html
- Expected Responses
  - 404 if the movie matching the URL parameter id could not be found
  - 200 if the movie matching the URL parameter id was found, renders a page displaying information about that movie

GET /movies/?

- HTTP Method: GET
- Description: Search for movies that match the query parameters
- Query parameters:
  - actors: a string containing one or more actors, separated by a comma
  - genres: a string containing one or more genres, separated by a comma
  - movieTitle: a string containing the movie title
- Response Data Types:
  - application/json
  - text/html
- Expected Responses
  - 404 if no movies match the query parameters
  - 200 if the query was successful, displays movies contained in the search results

GET /movies/genre/?

- HTTP Method: GET
- Description: Search for a movies that match the specified genre
- Query parameters:

        - genres: a string containing the name of a specific genre
    - Response Data Types:
        - application/json
        - text/html
    - Expected Responses
        - 404 if no searches match the genre
        - 200 if the query was successful, displays a page containing movies of the specified genre

POST /movies
- HTTP Method: POST
- Description: Create a new movie and add it to the collection of movies
- Body information:
    - Title: the new movie's title
    - Year: the new movie's release year
    - Runtime: the new movie's runtime
    - Plot: the new movie's plot
    - Genre: the new movie's genre(s)
    - Actors: the new movie's actor(s)
    - Director: the new movie's director(s)
    - Writer: the new movie's writer(s)
- Response Data Types:
    - none
- Expected Responses
    - 404 if any of the specified people were not found (they are not part of the known people)
    - 201 if the query was successful

POST /movies/:id/reviews
- HTTP Method: POST
- Description: Create a new review for a movie
- URL parameters
    - id: a movie's unique id
- Body information:
    - reviewText: the written review
    - summary: the review's summary
    - rating: the rating given to the movie
- Response Data Types:
    - none
- Expected Responses
    - 404 if any of the specified people were not found
    - 201 if the query was successful

/people

GET /people/?
- HTTP Method: GET
- Description: To find people with names containing the string and send the list back to represent "predictive search"
- Query parameters:
    - name: a name or part of a name the user typed to search for a specific person
- Response Data Types:

        - application/json
    - Expected Responses
        - 200 if the query was successful

GET /people/:id
- HTTP Method: GET
- Description: To display a page containing the information for the person corresponding to the parameter id
- URLparameters:
    - id: a person's id
- Response Data Types:
    - text/html
    - application/json
- Expected Responses
    - 200 if the request was successful

POST /people
- HTTP Method: POST
- Description: To create a new person and add them to the collection on people
- Body information
    - name: the name of the person that will be created and added to the collection of people
- Response Data Types:
    - none
- Expected Responses
    - 201 if the creation was successful
    - 409 if a person with the specified name already exists

/user

GET /user
- HTTP Method: GET
- Description: Render a page displaying all users in the collection of users
- Response Data Types:
    - text/html
    - application/json
- Expected Responses
    - 200 if the request was successful

GET /user/:id
- HTTP Method: GET
- Description: Render a page displaying the user's profile
- Response Data Types:
    - text/html
    - application/json

- URL parameter
    - id: the user's unique id
- Expected Responses
    - 200 if the request was successful
    - 404 if a user with the id was not found

POST /user/:id/movieList

- HTTP Method: POST
- Description: Add a new movie to the user's watchlist
- URL parameters:
    - id: the user's unique id
- Body information
    - name: the movie name
    - id: the movie's unique id
- Response Data Types:
    - none
- Expected Responses
    - 200 if the request was successful
    - 409 if the movie is already in the list

POST /user/:id/followedPeople
- HTTP Method: POST
- Description: Follow a person
- URL parameters:
    - id: the user's unique id
- Body information
    - pid: the person's unique id
- Response Data Types:
    - none
- Expected Responses
    - 200 if the request was successful
    - 409 if the person is already in the list

POST /user/:id/followedUsers
- HTTP Method: POST
- Description: Follow a user
- URL parameters:
    - id: the user's unique id
- Body information
    - pid: the user's unique id
    - username: the user's username
- Response Data Types:
    - none
- Expected Responses
    - 200 if the request was successful
    - 409 if the user is already in the list

DELETE /user/:id/followedPeople/:pid
- HTTP Method: DELETE
- Description: unfollow a person
- URL parameters:
    - id: the user's unique id
    - pid: the person's unique id
- Response Data Types:
    - none
- Expected Responses
    - 204 if the person was removed successfully
    - 404 if the person is not in the follow list

DELETE /user/:id/movieList/:mId

- HTTP Method: DELETE
- Description: Remove a movie from the watch list
- URL parameters:
    - id: the user's unique id
    - mid: the movie's unique id
- Response Data Types:
    - none
- Expected Responses
    - 204 if the movie was removed successfully
    - 404 if the movie is not in the follow list

DELETE /user/:id/followedUsers/:pid

- HTTP Method: DELETE
- Description: unfollow a user
- URL parameters:
    - id: the user's unique id
    - pid: the user's unique id
- Response Data Types:
    - none
- Expected Responses
    - 204 if the user was removed successfully
    - 404 if the user is not in the follow list

Other

GET /

- HTTP Method: GET
- Description: Render the home page
- Response Data Types:
    - text/html
    - application/json
- Expected Responses
    - 200 if the request was successful
    - 404 if the resources were not found

GET /profile

- HTTP Method: GET
- Description: Render the profile page
- Response Data Types:
    - text/html
    - application/json
- Expected Responses
    - 200 if the request was successful
    - 404 if the account related to the session id was not found

GET /account

- HTTP Method: GET
- Description: Render the logged in page, where you can change your account between contributing and regular, and logout
- Response Data Types:
    - text/html
    - application/json

- Expected Responses
    - 200 if the request was successful
    - 404 if the account related to the session id was not found

GET /logout

- HTTP Method: GET
- Description: Logout of the account
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the request was successful
    - 404 if the account related to the session id was not found

GET /contribute

- HTTP Method: GET
- Description: Render the contribute page
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the request was successful
    - 401 if the user is not a contributor

GET /search

- HTTP Method: GET
- Description: Render the advanced search page
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the request was successful
    - 404 if the resource was not found

POST /login

- HTTP Method: POST
- Description: Log a user in
- Body information:
    - username: the entered username
    - password: the entered password
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the login was successful - redirected to the home page
    - 401 if the credentials do not match any account

POST /signup

- HTTP Method: POST
- Description: Sign up for an account
- Body information:
    - username: the entered username
    - password: the entered password
- Response Data Types:
    - text/html
- Expected Responses
    - 201 if the account creation was successful - redirected to the home page
    - 409 if the username is taken

POST /account/regAccount

- HTTP Method: POST
- Description: Switch to regular account type
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the request was successful
    - 409 if the the account is already a regular account

POST /account/contributor
- HTTP Method: POST
- Description: Switch to contribor account type
- Response Data Types:
    - text/html
- Expected Responses
    - 200 if the request was successful
    - 409 if the the account is already a contributor account

Steps on how to initialize the program

1. Navigate to the project folder on your terminal
2. Run command "npm install" to install all dependencies
3. Run the server.js file
4. open up a browser to localhost:3000
5. Login with username: "simon" and password: "password" - Contributing account (can change to regular on account tab) or username: "musa" and password: "pass" - Regular account (also can change to contributor on account tab)

Overall Design

Critique

This application is based on data initialized and stored in RAM, and makes requests to a server hosted locally by the pc on which the application is run. In terms of quality and design, there could have been a lot of improvement. Though the features were responding as we were hoping for, some of them could have been implemented in a more secure fashion. For example, instead of sending entire objects over when making a request, we could have been more selective about what specific properties we would need to send over in order to execute the task at hand, as it is not a great idea to provide too much information for free to the end-users. In terms of RESTful system constraints, we could not implement cacheable content, which would mean that the resources would have to be requested every time we want to access it, leading to slower performance. In order to create meaningful properties, we found ourselves having to use many loops which slowed down the initial loading of information into collections, as well as accessing specific resources. In this aspect, we could have improved by instead using objects as opposed to arrays since accessing values with keys is faster than iterating over an array to find what we need. Input fields could have also been more strict by limiting what the user can enter, and value processing could have been improved with small details like capitalizing the first letter and converting the rest to lower case. The overall quality and cleanliness of our code could have been improved with better planning, such as which functions we were going to use and how we were going to execute it. For example, it would have potentially been a better idea to implement forms instead of creating a function to send an xmlhttp request to the server since xmlhttp requests do not support url redirection too well.

Successes

Though we were unable to make use of MongoDB in this application, I believe we've done a good job of replicating its object creation process using a randomId generator as opposed to using something more prone to duplication poor security, such as Title or an incrementing count. Using a randomId generator, we can limit what the end-user knows

about our data (such as number of users). Furthermore, we have thought of which codes to send in response to request errors, such as 404 whenever things like movies were not found when requesting for their page, or 401 for authorization and authentication errors. Generally, we have used the code 409 when an error related to lists occurred since these errors often involve data conflicting with a request, such as when you want to follow a person you are already following. In terms of following RESTful system constraints, we have been able to implement a couple of them. We do have a client/server side working properly, and each request is indeed independent from each other, we have uniquely identifiable resources.In addition, I believe our separation of request routes were categorized nicely. We have designed it based on general routes, (e.g. users, movies, people) so that with each request to a specific route, it would always be related to that general thing. By doing this, we found it was a lot easier to work with routes, as the requests were quite self-descriptive. The client and the server also interact well using JSON representations of resources in order to check for property values, change properties, etc. Our application makes it so that you have to login to access anything, which was cleaner for the implementation because we can avoid having to pinpoint where the user needed to be authenticated, and instead we were able to implement the authentication and authorization system in a single file.

Scalability, Latency, Experience

This project was built with scalability in mind, as all functions are made in such a way that if more movies were to be added, our project would be able to handle all processes. Each page created was separated logically so that the user is able to navigate easily throughout the webpage. However since MongoDB was not implemented, there is some latency when loading the original home page since everything is stored in RAM. The UI of this application is very attractive and user friendly as it is very simply laid out.

3. Description of Movie Recommendation Algorithm

The movie recommendation algorithm is situated in the User-router.js file. It learns about the user and recommends a single movie each time they add a movie to their list. For this to occur its computation is done in the router when a user clicks on any movie to add to their watch list. A variable named "`curMovieGenres`" is set equal to the current movies' genre that was just clicked. This is an array assigned to the variable above which makes the following calculations possible. It then traverses the movies database and compares the current movie's genre array to the databases' movie genre array and sees how many genres they have in common in order to provide the user with the best recommended movie. This is done with a count variable and whenever a movie has 2 or more similar genres, it creates an object called "`recObj`" which stores the movie title, movie id, and movie poster. This movie is then added the user "`recMovieList`" which is an object property containing all their recommended movies. However due to the fact that the user has the possibility to add an endless amount of movies, we have implemented a "`numMovies`" variable which is a counter and limits the amount of recommended movies to 5 at a time per user. This works by deleting the oldest recommended movie object out of their `recMovieList` so that they are always presented with newer more accurate recommendations.

4. Description of Similar Movie Algorithm

The similar movie algorithm is situated in the Movie-router.js file. It learns about the current movie displayed on the single movie page and compares its genres and actors to the ones in the movie database. This is computed by first storing the single movie page genre array into "`curMovieGenres`" and storing the single movie page actor array into "`curMovieActors`". It then iterates through the movies in the database and compares each value within the current genre and actors array with the movie databases' current genre and actors array. Once it finds a movie where there are three or more similar genres, it adds that movie to the similar movie list using the "`recObj`" which stores the movie title and movie id. It also checks whether that database movie has an actor that is also in the current single movie page, If so, that movie will be added to the "`recObj`" and into the similar movie list. It checks for duplicate movie

objects by filtering out objects that contain the same values. Since there can be several similar movies, it limits the program to only display a max of five similar movies on the single movie page.