Movie Database

Project Background

The goal of this project will be to create a web application similar to the Internet Movie Database. Your application will maintain a database of movie information, including things like the movie title, release year, writers, actors, etc. Your site will support regular users, who will be capable of browsing all of the information on the site and adding movie reviews. You will also need to support contributing users, who will be able to add new people/movies and edit existing records. The site will also offer movie recommendations based on a user's past reviews. Within this document, the term 'people' is used to refer to people that are part of a movie (actor, writer, director, etc.), while the term 'user' refers to a user of your application.

The following sections will outline the minimum requirements of the movie database project. You are encouraged to ask questions to clarify the requirements and constraints of the project. The details for this project may be updated slightly throughout the term if necessary. It is the students' responsibility to ensure they are up to date on any changes to the document. A file summarizing the changes to the document will be maintained and shared with the students.

Provided Data and Assumptions

The provided project zip file contains two files that you can use for initial movie data. The *movie-data.json* file contains a JSON array with information for 9125 movies retrieved from The Open Movie Database, while the *movie-data-short.json* file contains an array with the first 10 of those movies. This JSON data should contain all of the required information to initialize your server with movie data, though you will likely need to parse it in some way to convert the data into a more usable format.

For the purposes of this project, you can assume that actor/director/writer names are unique. If movie X lists "Norman Cates" as an actor and movie Y lists "Norman Cates" as a director, you can assume these refer to the same person.

Technology Constraints

The main server code for your project must use Node.js. All client resources required by your project must be served by your server. Your project's data must be stored using MongoDB. Any additional software/modules/frameworks you use must be able to be installed using your NPM install script. The backend of your project must be able to run successfully on the provided course OpenStack image and your client must work within an up-to-date Chrome browser. An updated list of allowed modules/frameworks/etc. will be included with the project documents. If you are unsure if something is allowed or would like to see if something could be allowed, you should ask for clarification before proceeding.

User Accounts

The application must provide a way for users to create new accounts by specifying a username and password. Usernames must be unique. A user should be able to log in and out of the system using their username and password. Within a single browser instance, only a single user should be able to be logged in at one time (i.e., user A and user B cannot log in to the system within the same browser window). All newly created accounts should be considered regular users until the user manually upgrades themselves to a contributing user.

When a user is logged in, they should be able to view and manage information about their account. The application must provide a way for the user to:

1. Change between a 'regular' user account and a 'contributing' user account. If a user changes account types, it should only affect their ability to carry out an action in the future. That is, anything created by a user while they have a contributing user account should remain unaffected if the user switches back to a regular account.
2. View and manage the people they follow. The user should be able to navigate to the personal page of any person they have followed. The user should be able to stop following any person that they have followed.
3. View and manage the other users they follow. The user should be able to navigate to the user page of any user they have followed. The user should be able to stop following any user that they have followed.
4. View recommended movies. These recommendations should be made based on what information your web application knows about the user, such as what movies they have reviewed or what people/users they have followed.
5. Search for movies by title, name, and/or genre keyword, at minimum. Additional types of search can also be included. The user must be able to navigate to the movie page for any of the search results.

Viewing Movies

When viewing a specific movie, a user must be able to:

1. See the basic movie information, including at minimum: the title, release year, average rating, runtime, and plot.
2. See the genre keywords and allow the user to navigate to search results that contain movies with that genre keyword.
3. See the director, writer, and actors the movie has, which should also allow the user to navigate directly to each person's page.
4. See a list of similar movies to this one and allow the user to navigate to the page for any of those movies. How similarity is defined is up to you, though you should be able to support your algorithm design with logical arguments.
5. See movie reviews that have been added for the movie.
6. Add a basic review by specifying a score out of 10.
7. Add a full review by specifying a score out of 10, a brief summary, and a full review text. You can use automatically generated text from an available package to fill these values quickly but must support manual entry as well.

Viewing People (directors, writers, actors)

When viewing the page for a particular person, the user must be able to:

1. See a history of all of this person's work. Each movie entry should allow the user to navigate to that movie's page.
2. See a list of frequent collaborators of this person. That is, a list of people this person has worked with the most, according to your database information.
3. Choose to follow this person. If a user follows a person, the user should receive 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.

Viewing Other Users

When viewing the page for another user, the current user must be able to:

1. See a list of all of the reviews this user has made and be able to read each full review.
2. See a list of all of the people this user has followed and be able to navigate to each person's page.
3. Choose to follow this user. If a user X follows a user Y, user X should receive a notification any time user Y creates a new review.

Contributing Users

If a user's account type is set to be a contributing user, the user should be able to do everything a regular user can do and also:

1. Add a new person to the database by specifying their name. If the name already exists, the user should not be able to add the new person.
2. Add a new movie by specifying all of the minimum information required by your system, including at least one writer, director, and actor. The user should only be allowed to add people to a movie if that person exists in the database already. You can include an 'auto-fill' feature to generate a random movie quickly but must also provide a way for the user to manually create a movie entry.
3. When viewing a movie, be able to edit the movie by adding actors, writers, and/or directors.

REST API

In addition to the web-based client that most users will use, your movie database application must also provide a public JSON REST API that supports the following routes and parameters, at minimum:

1. **GET /movies** – Allows searching for movies in the database. Returns an array of movies that match the query constraints. Must support at least the following query parameters:
    a. *title* – A string that should be considered a match for any movie that has a title containing the given string *title* (character case should be ignored). If no value is given for this parameter, all movies should match the title constraint.
    b. *genre* – A string that should be considered a match if the movie's list of genre keywords contains the given keyword. If no value is given for this parameter, all movies should match the genre constraint.
    c. *year* – A number that should be considered a match if the movie's release year matches. If no value is given for this parameter, all movies should match the year constraint.
    d. *minrating* – A number that should be considered a match if the movie's overall average review rating on your site is greater than or equal to the given value. If a movie does not have any reviews, the number value of its rating should be considered 0.
2. **GET /movies/*:movie*** – Allows retrieving information about a specific movie with the unique ID *movie*, assuming it is a valid ID. The returned JSON should contain the basic movie information (title, year, actors, etc.) and also information about the reviews of the movie.
3. **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 the

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.

4. **GET /people** – Allows searching for people within the movie database. At minimum, this must support an optional *name* query parameter. If the query parameter is included, it should return any person in the database whose name contains the given *name* parameter. The search should also be done in a case-insensitive nature. If no parameter is given, all people should match the query.
5. **GET /people/*:person*** – Retrieves the person with the given unique ID, if they exist. This should include, at minimum, the name of the person and the movies they have been a part of.
6. **GET /users** – Allows searching the users of the application. At minimum, this must support an optional *name* query parameter. If the query parameter is included, it should return any user in the database whose name contains the given *name* parameter. The search should also be done in a case-insensitive nature. If the parameter is not specified, any user should match the search constraint.
7. **GET /users/*:user* –** Get information about the user with the given unique ID, if they exist. This should contain their username, their account status (regular user or contributing user) and the reviews the user has made.

Project Report

You will also be required to submit a project report that must explain the overall design of the system, outline any assumptions that were made, provide supporting arguments for the design decisions that were made, discuss technologies used within the application, document your server's API, describe any testing completed as part of the project, highlight key features of the system, and provide documentation on how to use/test the system. You must be able to justify your design and implementation choices from a perspective that considers scalability, robustness, extensibility, and maintainability, as well as the overall user experience. It is important to document the design decisions you make, as well as your reasoning for making those decisions and any alternative approaches you tried or considered as the term progresses. This will make writing a quality report near the end of the term significantly easier. We will discuss potential sections and other outline details for the project reports throughout the term.

Pair Project Extensions

Pair projects will be held to higher standards than individual projects in general. If you are completing a pair project, it is recommended that you include at least 3 of the below extensions into your project, or similar extensions you come up with on your own. For those looking for an A grade on the project, pairs are encouraged to include more than the minimum recommended extensions and individuals are encouraged to include some extensions as well.

1. Use React, Angular, or another supported frontend framework.
2. Implement a GraphQL API in addition to the REST API.
3. Implement your own load balancing solution and simulate a distributed system by instantiating multiple instances of your server. This load balancer should be able to detect any of the servers going offline (e.g., simulating a machine crash). This load balancer could also handle the addition of more servers at runtime.
4. Support a full-sized/desktop interface, as well as a smaller, mobile-friendly interface.
5. Provide in-depth automated testing of your API and system, including documenting specific test cases and their expected results.
6. Implement your own caching mechanism for your server/API. Test your caching system under various circumstances and compare its performance to a version that does not use caching.
7. Implement a web-scraper to retrieve additional movie data from IMDB and incorporate it into your system. You could do this in an automated way (e.g., like a web crawler) or by allowing a user to provide an IMDB URL to have a specific movie parsed. If your parser is error-prone, you could also use this feature to initially populate a form on the page and have the user double-check the information manually before it is submitted to the database.