Final Documentation

Movie Database

COMP 2406

Malcolm Yeh

Table of Contents

Setup Instructions

OpenStack

IP Address	134.117.128.136
username	student
password	test123

1. Tunneling server

```
ssh -L 9000:localhost:5000 student@134.117.128.136
```

2. Tunneling client

```
ssh -L 9001:localhost:3000 student@134.117.128.136
```

3. Initializing database (should already be done)

```
sudo ./init.sh
```

Note: This step should not be required as the database should already be populated. It can be used to clear any added users, reviews and movies. The initialization script installs all required modules for the server and loads each of the ~10,000 movies into MongoDB, creating many-to-many relationships for each movie. This process will take several minutes to finish in OpenStack. Please be patient!

4. Run server (should already be done)

```
./runserver.sh
```

5. Run client (should already be done)

```
./runclient.sh
```

Note: if an error occurs with either run script because an instance is already running on either port 3000 or 5000, kill all node processes using

```
pkill node
```

Server should be tunneled to http://localhost:9000
Client should be tunneled to http://localhost:9001

Running React App Locally

1. Navigate to application folder

```
cd MovieDatabase/client
```

2. Install required modules

```
npm i
```

3. Start React app

```
npm start
```

Note: If the app cannot connect to the server, check inside the .env file that

```
REACT_APP_API_URL=http://localhost:9000
```

Summary of Functionality

User Accounts

All required functionality is implemented.

1. Creates new account by specifying a (unique) username and password

    - Go to **Sign Up** in navigation bar
    - Enter username and password

2. Limits number of users logged in at one time in a single browser instance to one
3. Changes between regular and contributor user account

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Go to **Profile** in navigation bar
    - Click on **Become Contributing User / Become Regular User** button below username to toggle account types

4. View and manage people they follow

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Go to **Profile** in navigation bar
    - On the right (bottom if using mobile interface) is a list of following People
    - Click on the **name** to navigate to following Person's page
    - Click on the **X** next to the name to unfollow Person

5. View and manage users they follow

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Go to **Profile** in navigation bar
    - On the right (bottom if using mobile interface) is a list of following Users
    - Click on the **name** to navigate to following User's page
    - Click on the **X** next to the name to unfollow User

6. View recommended movies

    - Recommended movies are shown in the carousel in the **Home** page
    - If not logged in or user has not marked any movies as watched/reviewed, the 5 most recent movies are shown

Viewing Movies

All required functionality is implemented.

1. Displays basic movie information (title, release year, average rating, runtime, plot)

    Navigate to a movie by the following methods:
    - Click on the **movie title** in the carousel of movie posters and trailers in the **Home** page
    - Type movie title or keyword in **Search bar**, click **Search** or press **Enter**, and click on a **Movie Card**
    - Go to **Genres** in navigation bar, select desired genre, click on a **Movie Card**
    - Search for user or people in **Search bar**, click on a **Movie Card** in Movies Watched (user) or Movies (people)

2. Displays genre keywords and allow the user to navigate to search results that contain movies with that genre keyword
3. Displays the director, writer and actor the movie has, which allows the user to navigate directly to each person's page
4. Displays a list of similar movies and allows the user to navigate to the page for any of those movies
5. Displays movie reviews that have been added for the movie
6. Adds basic review (score out of 10)

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Scroll to bottom and click **Rate this title**
    - Enter a score from 0-10 and click **Submit** OR click **Generate**

7. Adds full review (score out of 10, brief summary, full review text) and supports both automatically generated text and manual entry

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Scroll to bottom and click **Review this title**
    - Fill in required fields and click **Submit** OR click **Generate**

Viewing People

All required functionality is implemented

1. Displays a person's work and allows the user to navigate to that movie's page

- Navigate to a person's page by searching for them in the **Search Bar**, through a **Movie** Page, or through a **User** Page who is following the person

2. Displays a list of frequent collaborators
3. Allows user to follow the person and displays notification to user when a new movie is added to the database involving the person or if the person is added to an existing movie

    - Sign in through **Log In** or **Sign Up** in navigation bar
    - Navigate to Person's page
    - Click follow under the person's name
    - Add a new movie through API or client as a contributing user

Contributing Users

All required functionality is implemented

1. Adds a new person to the database by specifying their name. Does not allow duplicates.

    - Become contributor (outlined in User Accounts)
    - Navigate to **Add Name** in the navigation bar
    - Enter a name and click **Add**

2. Adds a new movie by specifying all of the minimum information required by the system and supports both automatically generated text and manual entry

    - Become contributor (outlined in User Accounts)
    - Navigate to **Add Movie** in the navigation bar
    - Enter required fields and click **Add** or click **Generate**

3. Allows editing of movies by adding actors, writers and/or directors

    - Become contributor (outlined in User Accounts)
    - Navigate to a movie (outlined in Viewing Movies)
    - Click **Edit Movie** underneath the movie's description
    - Enter new people (separate multiple people with commas) and click **Submit**

REST API

All required functionality is implemented

Details covered in Full REST API Documentations

1. GET /movies
    - Supports title, genre, year, migrating query parameters
2. GET /movies/:movie
3. POST/movies
4. GET /people
    - Supports optional name query parameter
5. GET /people/:person
6. GET /users
    - Supports optional name query parameter
7. GET /users/:user

Full REST API Documentation

Note the /api/ before each endpoint.

GET /api/people

Query parameters:

- name: String

Sample request:

```
/api/people?name=alex
```

GET /api/people/:person

:person is a People ObjectId

GET /api/users

Query parameters:

- name: String

Sample request:

```
/api/users?name=test-user-1
```

Get /api/users/:user

:user is a User ObjectId

GET /api/movies

Query parameters:

- title: String
- genre: String
- year: Number
- minrating: Number

Sample request:

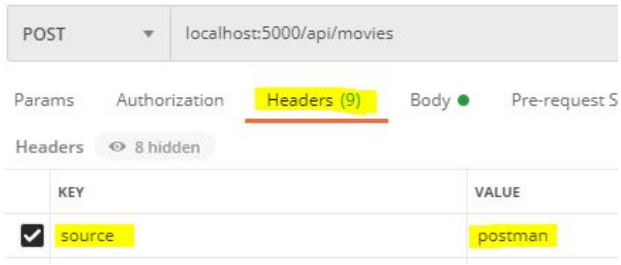
```
/api/movies?title=toy&genre=Family
```

GET /api/movies/:movie

:movie is an Movie ObjectId

POST /api/movies

If testing from Postman, include {source: “postman”} in Headers so that the server bypasses the check for user authentication and authorization.



Sample data (paste in Body > x-www-form-urlencoded):

```
{
   "Title":"Cyberpunk 2077",
   "Year":"2020",
   "Rated":"R",
   "Released":"10 Dec 2020",
   "Runtime":"999 min",
   "Genre":"Adventure, Crime, Thriller",
   "Director":"Richard Borzymowski",
   "Writer":"Marcin Blacha, Jakub Szamałek, Stanisław Święcicki",
   "Actors":"Keanu Reeves, Michael-Leon Wooley, Samuel Barnett",
   "Plot":"Cyberpunk 2077 is an open-world, action-adventure story set in
Night City, a megalopolis obsessed with power, glamour and body
modification. You play as V, a mercenary outlaw going after a one-of-a-kind
implant that is the key to immortality. You can customize your character's
cyberware, skillset and playstyle, and explore a vast city where the
choices you make shape the story and the world around you.",
   "Language":"English",
   "Country":"USA",
   "Awards":"Nominated for 9999 Oscars.",
   "Poster":"https://i.imgur.com/s6vO46h.png",
   "Metascore":"99",
   "imdbRating":"9.9",
   "imdbID":"11111111",
   "Type":"movie",
   "DVD":"10 Dec 2020",
   "BoxOffice":"N/A",
   "Production":"CD Projekt"
}
```


POST /api/movies/addPeople/:movie

:movie is an Movie ObjectId

If testing from Postman, include {source: “postman”} in Headers so that the server bypasses the check for user authentication and authorization.

Sample data:


```
{
      "Director": "New Director",
      "Writers":"New Writer1, New Writer2",
      "Actors":"New Actor1, New Actor2, New Actor3"
}
```

GET /api/featuredmovies

Returns array of 5 most recently released movies in the database.

GET /api/genres

Returns array of movie genres.
Note: currently the array is hard coded for the sake of speed, but above the response is a function that returns unique genres from every movie in the database, in the case where movies can be added that have a new genre(s);

GET /api/trailer/:searchterm

:searchterm is in the format

```
${movie.Title}+${movie.Year}+trailer
```

Returns link to embedded YouTube video for the first result of the searchterm.

GET /api/addfromimdb

Adds movie by imdbID or by searching IMDB for a movie or series (adds the first search result).
Query parameters:

- id: String (must be imdbId)
- title: String
- type: String (must be either movie or series)

Either id or both title and type are required.

Sample request:

```
/api/addfromimdb?id=tt13143964
```

```
/api/addfromimdb?title=chhalaang&type=movie
```

Note: This route should be tested with newer movies or series so it would not run into duplication errors with the database (2017+)

Extensions

React

- Used React for front-end client
- React hooks to manage state, form fields and app context

Interface Adapts to Varying Screen Sizes

- Used Bootstrap's responsive grids to set breakpoints for elements
- Rearrange display for smaller screen sizes
- Event listener on window to modify carousel (show only movie poster if screen size is too small)

PassportJS

- See Design Decisions
- Local strategy for authentication

SocketIO

- See Design Decisions
- Notification for when followed user or person adds a review/movie

MongoDB/Mongoose

- Mongoose Schemas for Movie, People, Review
- Field validation, duplication checking
- Initialization script that inserts movies, sample users and sample reviews from json

```
/server/init
```

Automated Backend Tests

```
npm test
```

- Test Mongoose schemas

```
/server/models/tests
```

    - Check that documents are created and saved properly
    - Ignores fields not defined in schemas
    - Field validation (required + valid inputs)
    - Duplication checking

- Integration tests

```
/server/tests/integration.test.js
```

    - Check that correct mongoose functions are executed when calling particular API endpoints

IMDB Web Scraper

```
/server/scraper/imdbscraper.js
```

- Searches for movies or series or episodes with given search term and returns list of imdbIds
- Searches for movie or series or episode with imdbId and returns data that can be used directly with server's POST /movies
- Used Axios to make http request to IMDB page and Cheerio to parse the HTML
- GET /api/addfromimdb/:imdbid endpoint to add to database

YouTube Trailers

- Carousel in home page that has embedded Trailer video
- Video will automatically stop when changing Carousel slides to prevent audio overlap
- Originally started off with **YouTube Data API**
    - Search for movie trailer and return VideoId of first result, then set iframe src to be video link
    - Unfortunately, YouTube API limits to 100 searches per day, which is not sufficient for this project as a single user can reach 100 searches in a few minutes (especially since Home page carousel loads multiple trailers at once). Quota was quickly exhausted while testing the implementation.
- Switched to **Youtube Iframe API**
    - Using query parameter listtype=search, was able to directly embed the first youtube video as part of a playlist that contains search term
    - Unfortunately, Youtube Iframe API deprecated the feature on **November 15 2020** https://developers.google.com/youtube/iframe_api_reference#Queueing_Functions
- Switched to **manual web scraping** (using the same technique as IMDB scraper)
    - Added endpoint /api/trailer/:searchterm
    - React app will request trailer video link from server, passing in the query param

```
${movie.Title}+${movie.Year}+trailer
```

    - Server makes request to youtube search
    - Using Cheerio, data from the response is extracted and the videoId of the first result is sent back to the client
    - Client sets the src of iframe player to response data
- The major drawback to this approach is that it is very slow, especially compared to using the Youtube Iframe API.
    - Client →Server →YouTube (Search) →Server→Client→YouTube (Embed) →Client

vs
Client →YouTube (Embed) →Client

- Another approach is to find the trailer link during database initialization and adding movies, storing it as part of the document, but this will take an extremely long time as it adds several seconds to each movie and is likely not worth it as most movies will not be visited by the Pareto distribution

Design Decisions

React instead of Template Engine

- Employ MVC pattern instead of using Pug or EJS and having the server render pages
  - Server is only responsible for receiving and sending JSON data while the frontend will parse received data for a view and handle user input
  - leads to clearer distinction between the responsibility of client and server
- The movie database has a lot of dynamic pages (~10,000 movies, ~26,000,000 people and potentially many users) and within those pages, components can update frequently (follow/unfollow, add/remove review, add/edit movie, etc.), which makes client side rendering significantly faster and more maintainable as it scales
- Reusability of functional components leads to less redundant code, especially if there are only minor differences in specific use cases
- Client-side validation can act as a second safety layer on top of Mongoose Schema validation and prevent requests that are erroneous, lessening the load on the server
- Loading individual components as some data requires more server calls and/or more time to fetch. Furthermore, some component data needs to change and it does not make sense to reload unaffected sections. A loading spinner was added to provide the user a visual indicator. Since a 2-second delay results in website abandonment rates of up to 87%[1], it is crucial to provide visual feedback as soon as possible.

[1] https://www.hobo-web.co.uk/your-website-design-should-load-in-4-seconds/

PassportJS Middleware for Authentication

- Encapsulates authentication flow which increases maintainability for rest of the server
- Although only the local strategy has been used in the project, it is easy to integrate new strategies with OAuth providers such as Google and Facebook, which have become an increasingly popular method of user authentication for web applications
- Session support is built into PassportJS and integrates seamlessly with Mongoose and MongoDB (where temporary sessions are stored)

Combining Log In/Sign up

- Apart from password confirmation (entering the same password twice), which can be handled very easily on the frontend, there is very little difference between the sign in flow

of Log In and Sign Up
- Thus, I chose to combine the functionality into a single express route and a single PassportJS authenticate function.
- The drawback of this approach is that there is no way to distinguish a user signing up with an existing username and a user logging in with the incorrect username/password combination, which may lead to some confusion for the front-end user

Setting Many-to-Many Relationships in Database

- Importing movie data from JSON to MongoDB can be done with a single call to insertMany(), which is very fast. Instead, the initialization script loops through each movie, checking if each person already exists, creating them if they don't exist, and linking everything together via ObjectId (People have frequent collaborators and movies, while a Movie will have actors, directors, writers). This causes the initialization process to take several minutes as opposed to a few seconds.
- The ObjectID is the primary key which is indexed. Since MongoDB uses self-balancing tree indexes, searching for a particular ObjectID will be O(log n). Thus, it is faster to search for a document by its ObjectId rather than another field like its Name or Title (which can also lead to issues if two Documents have the same field).
    - The application will process potentially hundreds of searches for every interacting user, so it is important that querying the database is as fast as possible
- The increased initialization cost is negligible as it should only be run once when the server is deployed into production (OpenStack)

Movie Recommendation

- For single movie:
    - Match the rating
    - Match as many genres as possible (starting with all of the genres and if initial query doesn't find at least 1 movie, keep trying with less and less genres)
- For user
    - Match movies that following People star in
    - Get most popular genres and try to match movies (starting with all of the genres and if initial query doesn't find at least 1 movie, keep trying with less and less genres)

Notification

- For the React frontend, if the user is authenticated, it opens a socket listening to the room userId
- On the server side, whenever a followed person/user adds a movie/review, it loops through their followers and broadcasts to the room of the follower's userId
- Using butter-toast (quick and cleaner alternative to window.alert()), displays notification broadcast from server

Possible Improvements

Full CRUD Functionality for API

- A contributor should be able to edit Movies in more ways than just adding new People
- As it stands, there is no way to fix any typo without accessing the database directly
- Not only adds more functionality to end user but could have potentially sped up development as Documents could be edited/removed rapidly through the terminal instead of through the MongoShell or GUI like Robo3T

React Component Testing

- As the project gets more complex and more functionality is added, it is crucial any changes doesn't break previously working components, so it would be beneficial to add automated tests to the front end
- Test driven approach to development can reduce the defect rate and might save time spent debugging big problems

Custom Middleware to Log More Information

- Custom middleware could be implemented in the Express server for telemetry and logging more information, which can be useful in debugging and analyzing user trends

Using more Mongoose functionality

- Could have used instance methods and statics for each Schema (in particular Movie and User), which can abstract away basic Mongoose calls, which leads to more readable code

Refactoring

- Although many React functions were reused, there still remains a lot of similar code
- The same applies to mongoose functions for the server

Proper Custom Alerts

- I used window.alert and butter-toast for the sake of time but I could have written a more flexible alert modal component that could be reused for multiple use cases

Additional Modules

Client

axios	Http client to interact with the server that is simple, lightweight and promise-based. Prefer syntax over fetch.
bootstrap	Front-end framework
butter-toast	Notification display
lorem-ipsum	Random text generation for Reviews and Movies
react	**[Explained in Design Decisions]**
socket.io-client	Receiving notifications from server
dotenv	Set server endpoint as environment variable to facilitate test on local server and deployment to OpenStack

Server

bcryptjs	Hash passwords before storing in database
lorem-ipsum	Random text generation for sample reviews (database initialization)
socket.io	Sending notifications to client
axios	Get data from a webpage to be used with cheerio.
cheerio	Implementation of core jQuery for web scraping
jest	Testing framework (Mongoose Schemas and API integration)
supertest	Test framework for endpoint and integration testing
passportjs	**[Explained in Design Decisions]**

Best Features

I am content with my database organization as it is optimized for querying (as mentioned in Design Decisions), my React implementation where I was able to learn about hooks and step away from React.Component, and the addition of YouTube movie trailers. Although the solution isn't elegant, I'm glad that I was able to find a fix when an API I was relying on was deprecated.