**COMP 2406 Project Report**

**Author**

Jason Coil

101003085

**Partner**

Ade Banjo

101109343

April 16th, 2021

Table of Contents

[1. Introduction and Overview 3](#_Toc69292763)

[2. Installation and Initialization Instructions 3](#_Toc69292764)

[2.1 Running the Server 3](#_Toc69292765)

[3. Overall Design 4](#_Toc69292766)

[3.1 Code Quality and Organization 4](#_Toc69292767)

[3.2 RESTful Design Principles 5](#_Toc69292768)

[3.3 Use of Proper HTTP Status Codes 5](#_Toc69292769)

[3.4 Proper Error Handling 6](#_Toc69292770)

[3.5 Use of Asynchronous Operations 7](#_Toc69292771)

[3.6 Minimization of Data Transfer 8](#_Toc69292772)

[4. Movie Recommendation Algorithm 9](#_Toc69292773)

[5. Similar Movies Algorithm 11](#_Toc69292774)

[6. Summary 12](#_Toc69292775)

[Bibliography 12](#_Toc69292776)

[Appendix A 13](#_Toc69292777)

[A1. Required Features 13](#_Toc69292778)

# Introduction and Overview

The goal of this project was to implement a movie web server using NodeJS, Express, and MongoDB that also implemented the required features (required features are listed in Appendix A.1). This report discusses how to properly install, initialize, and run the database and server, the quality and organization of the code, the RESTful design principles, the use proper HTTP status codes, proper error handling, and discusses the two main algorithms. The first algorithm had to recommend movies to a user based on his watched movie list. The second algorithm had to suggest a list of movies similar to the movie the client is currently viewing. The goal of the project was met, but it had areas that it could improve on. The areas for improvement are the code organization, a specific RESTful design principle, and the sophistication of the algorithms.

# Installation and Initialization Instructions

The installation and initialization instructions are detailed in Table 1 below:

**Table 1: Installation and Initialization Instructions**

|  |  |
| --- | --- |
| **Step** | **Description** |
| 1 | Download all files in the package “COMP2406Project” |
| 2 | Ensure you have MongoDB installed (if not, install it) |
| 3 | Open a command prompt and navigate to “\MongoDB\Server\4.4\bin” |
| 4 | Type into the command prompt: “mongod” to start up the database |
| 5 | Open a Node.js command prompt and navigate to the folder containing the “COMP2406Project” files |
| 6 | Type: “npm install” to install all the dependencies needed for this project |
| 7 | Type: “node db-initializer.js” to initialize the database |

The server and database are now all set up to use. The next section will discuss how to run the server.

## 2.1 Running the Server

Since the database is initialized and the dependencies are installed, the next step is to run the server. To run the server, open a Node.js command prompt and navigate to the folder containing all the “COMP2406Project” files. Next, navigate to the folder containing the “server.js” file and type in: “node server.js” to run the server. The server should now be accessible by typing in ‘localhost’ into the URL of the browser.

To test the users logging in and logging out capabilities a few user credentials are given below in Table 2:

**Table 2: User Credentials for Testing**

|  |  |
| --- | --- |
| User’s name | Password |
| Jim | 1234 |
| Bob | 911 |
| Pat | cucumber |

# Overall Design

## 3.1 Code Quality and Organization

For code quality to be satisfactory it must answer yes to the questions defined below:

* Is the code reliable?
* Is the code maintainable?
* Can the code be easily tested?
* Is the code reusable?
* Is the code scalabe?

**Reliable code** is code that has a low probability that it will fail over time and it does sufficient error handling for things such as unexpected input. For our project, the main features are working correctly, and most of the bugs have been worked out. The main features are things such as viewing user/movies/people, creating users/movies/people, and updating user information. The code handles unexpected input by not showing buttons on pages to watch movies or to create a review if they are not logged in. The code will verify users logged in status to ensure that they cannot do anything that is unauthorized such as creating movie or people, and it will give the user an alert to let them know the error. Creating a movie will check if all forms are filled in with the correct type (numbers, or strings) and ensure that if any input is not correct the movie will not be added, and an error will be thrown. However, there is one potential error that occurs and is not handled, which allows users to review the same movie more than once.

**Maintainable code** is code that can be easily maintained to keep it working in the future. The code for this project is maintainable because:

* The code is easy to read and understand
* The variables and functions are given appropriate and meaningful names to aid in the understanding of what the code is doing
* There are helper functions written to help support more complex tasks which breaks the code into more manageable chunks.

However, there are some improvements that could be made. Specifically, routers can be added for each route (movies, users, and people) to split the code into more manageable chunks.

**Easily testable code** is code that can be easily separated and tested in chunks (i.e. independent from other factors). Since express breaks the code down into steps based on what request is coming in, the code is easily manageable. The only parts that may be more difficult to test is the recommended movie algorithm and similar movie algorithm as they have more connected parts. Overall, most of the code can be easily tested.

**Reusable code** is code that can be easily reused or repurposed for other tasks. Most of the code for the project can be repurposed or reused for other tasks, but there are a few reusability issues. There may be portions of the code that could be rewritten. For example, there are a few verification functions that do similar things and could be rewritten to reduce any repeated code.

**Scalable code** is when the code is capable of handling larger amounts of data. Since, we are using the mongo database we are no longer limited on the amount of data we could store based on if we had just kept all our data on the server. Therefore, since we are using mongoDB our project is scalable with large amounts of data.

## 3.2 RESTful Design Principles

A good RESTful design follows a handful of specific principles [1]. The principles and whether the project implements them is outlined below in Table 3:

**Table 3: RESTful Design Principles Used**

|  |  |  |  |
| --- | --- | --- | --- |
| **Principle** | **Not implemented** | **Somewhat Implemented** | **Fully implemented** |
| Accept and respond with JSON |  |  |  |
| Use nouns instead of verbs in endpoint paths |  |  |  |
| Name collections with plural nouns |  |  |  |
| Nesting resources for hierarchical objects |  |  |  |
| Handling errors correctly |  |  |  |
| Allow filtering sort and pagination |  |  |  |
| Maintain good security practices |  |  |  |
| Cache data to improve performance |  |  |  |

It is important to note that caching and security practices are out of the scope for this project. The “nouns at endpoint paths” principle is broken by the route that create users, people, or movies. Although despite these routes, the project overall still covers most of the RESTful design principles (disregarding the last two).

## 3.3 Use of Proper HTTP Status Codes

Status codes are important to send the proper responses back to understand if a request was successful or not and if it was unsuccessful it gives more detail into the error. The status codes used in the project our outlined below, in Table 4, as well as in what situation they are used:

**Table 4: HTTP Status Codes Used**

|  |  |
| --- | --- |
| **HTTP Status Codes** | **Description of when the status code is used** |
| **2xx Success** | |
| **200** | * Information from database is successfully retrieved * User tries to login after already being logged in * User successfully logs in * Logging out or trying to logout but not logged in * Page is rendered successfully * User successfully updates information * Person information is updated * Movie information is updated |
| **201** | * New user is created successfully * New movie is created successfully * New person is created successfully * New review is created successfully |
| **4xx Client Error** | |
| **401** | * Username and/or password is entered incorrectly to try and login * User tries to follow user but is not logged in * User tries to follow person but is not logged in |
| **403** | * Creating a new user but passwords do not match * Creating a new user but username already exists * Changing user type without anything selected * Removing movies from watched movie list without selecting anything * Creating a new person but name already exists * Creating a movie but missing form elements |
| **404** | * Handled by express when client tries to view a URL that does not exist or post to a URL that does not exist |
| **5xx Server Error** | |
| **500** | * When an error occurs when the server tries to communicate with the database to find/update movies or people or users |

## 3.4 Proper Error Handling

When errors occur, the function should return the proper error status and print out the error that occurred. The function should then stop before any other helper functions can occur. For example: when creating a movie, if there was an error, then the people in the movie should not be updated if the movie creation had an error, which our project handles.

Error handling should also check for unexpected inputs to handle them correctly, such as inputting a string into a field that requires numbers. The create movie function requires users to input a release year and a runtime, which are expected to be numbers. If the user inputs strings into these fields the error is caught before the movie is created and it sends an alert back to the user telling them to input a number instead of a string.

Another issue could be trying to do unauthorized events such as creating a movie or a person if you are not currently logged in or if you are not currently a contributing user. The verification step before the creation step makes sure to handle these errors before any creation happens.

A potential issue that can occur is when users try to follow or watch a movie when they are not logged in. These are handled in two different ways (either approach works). When the client tries to follow other users, it checks if the client is logged in first, giving an unauthorized error if the client is not logged in. However, the client is not informed/alerted that he is unauthorized. The other approach used on the ‘watch movie’ button is that the page is not rendered with the ‘watch movie’ button if the user is not logged in. Perhaps these errors could be handled with only one approach to make things more consistent.

The errors are handled when necessary, preventing any updating of the database if any errors arise. The errors also inform the client (most of the time) about the error and why it occurred. Overall, the errors are handled well, but some improvements can be made to make some of the error handling more consistent (i.e. use one type of approach rather than multiple) and to alert the user as to why the error occurred.

## 3.5 Use of Asynchronous Operations

Some of the asynchronous functions used are the creating movie function, the adding of directors/writers/actors to a new movie, and the removing notification’s function. The functions are described in more detail below.



**Figure 1: Create Movie Function (createmovie.js)**

The create movie function is asynchronous as it waits for a response from the server before completing its task. A user starts by filling in the forms required to create a new movie and then clicks the “create” button. This calls the ‘createMovie’ function which sends an asynchronous request to the server with the movie data along. The function then waits for a response back from the server to see if the movie was created or not. Once it receives a response from the server, the function sends an alert back to the user specifying if the movie was created, and if not, it will give a reason why.



**Figure 2: Add Director Function (createmovie.js)**

The director, actor, and writer functions are all similar, so explaining just the add director function explains all of them. The user can search for people dynamical to add them to the movie they are trying to create. The search button calls the ‘addDirector’ function which will send a request to the server. It sends along the text typed in the search field so the server can then contact the database to query for people’s names that match. The function waits for a response back, but the user is free to search for other people to add or to continue filling in the form. Once the function receives a response back from the server it will update the DOM elements on the page to give the user a list of people that match their query as well as a button that the user can click to add the person to the movie as a director.

**Removing notifications from user’s page (removeNotification.js):**



**Figure 3: Remove User Notification Function (removeNotification.js)**

There are two functions used to remove notifications (user or person notifications), but they are very similar, so without loss of generality explaining just one is sufficient. When the user clicks the delete button to remove a notification from their page it sends an asynchronous HTTP request to the server to update the user’s notifications. If the update was successful, then the DOM elements can be changed to remove the notification from the page. Otherwise, the notification will stay and an alert will be sent to the user specifying what went wrong.

It is also worth noting that most of the express functions and mongoose operations are asynchronous as they allow the server to continue operating as the express function or mongoose operation attempts to complete. When the express/mongoose function has completed it will send a callback to where it was called and return either an error or a result if successful.

## Minimization of Data Transfer

The main way to minimize data transfer is through pagination as it limits the amount of data needed to render a page. The limit is set to 10 for this project, which results in a lot less data transfer as opposed to no pagination and rendering a page with a list of 2500+ movies.

In addition to pagination, data transfer is minimized even further by using the mongoose ‘select’ feature. The ‘select’ feature specifies which properties from the object we care about and want to transfer over. This feature is used when searching for users or movies or people. The properties it selects for movies, people, and users is shown below in Table 5:

**Table 5: Selected Properties for Searching Movies/Users/People**

|  |  |
| --- | --- |
| **Searching for \_\_\_** | **Properties Selected** |
| Movies | ID, title, directors, writers, actors, releaseYear, runtime |
| Users | ID, name, followers |
| People | ID, name |

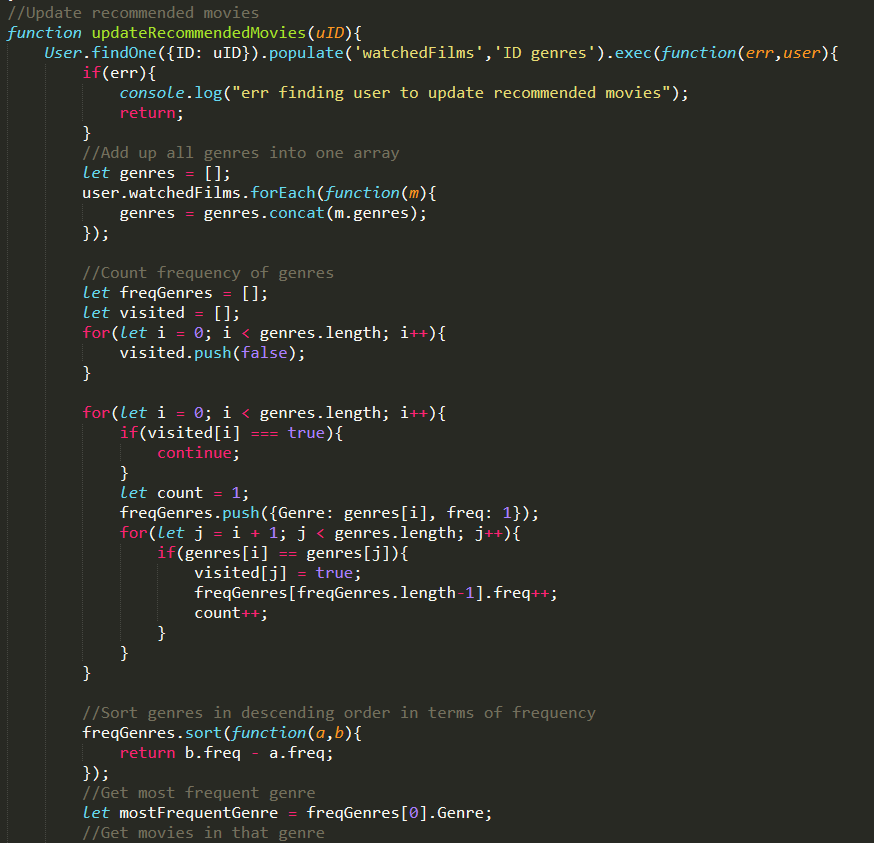
The get movies function requires the directors, writers, and actors to filter the matching movies based on the query for people names. However, this data is minimized to only requiring each of the directors/writers/actor’s names rather than all their properties. The ID is needed for linking to the movie page, and the title, release year and runtime are shown as information about each movie.

The get users function requires the ID for linking to the user page and the name is shown as text information to show who the user is. The follower’s array is used as an extra feature to show how many followers each user has.

The get people function requires the ID for linking to the person page and the name is shown as text information to let the client know which person they are viewing.

# Movie Recommendation Algorithm

The recommendation algorithm is called whenever the user watches a new movie. When a user watches a new movie his list of recommended movies is recalculated, and the user’s recommended movie list is updated accordingly. The algorithm is shown below in Figures 4 and 5:



**Figure 4: Movie Recommendation Algorithm (first half)**



**Figure 5: Movie Recommendation Algorithm (second half)**

The algorithm finds the user’s watched movie information and counts the number of times a genre occurs. It then sorts the genres in descending order to get the most frequent genre this user has watched. It then grabs all movies that contain that genre making sure not to include any movies in the users watched movie list (this is done using the *$nin* mongoose feature). It then sorts the movies based on their average rating. If the number of movies in that genre is more than 5 it will only display up to 5. It then updates the users recommended movie list. The result is a new list of the top-rated movies in the genre that is most frequently watched by that user. The algorithm efficiency is seen below (disregarding the time complexity of the mongoose find and update):

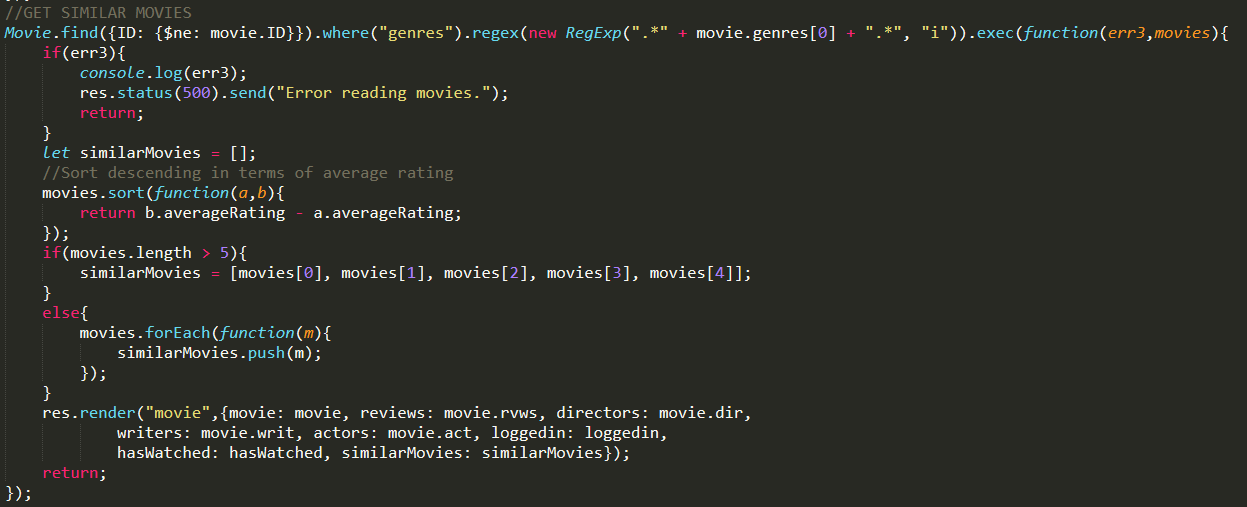
Let g = number of genres (it counts duplicates). Let n = number of movies containing that genre.

Therefore, worst case is (just denoting everything as n now as is standard practice): O(n2)

O(n2) works, but it is not ideal, and could possibly be improved to O(nlogn) by improving the counting frequency algorithm.

# Similar Movies Algorithm

The similar movie algorithm is put inside of the function that gets the movie the client is requesting at “/movies/movieID”. Once the function has retrieved the correct movie the client has wanted to view the algorithm is then called to retrieve a list of similar movies (max length of 5). The algorithm can be seen below in Figure 5:



**Figure 6: Similar Movie Algorithm**

The algorithm finds all movies related to the first genre in the current movies genre array, making sure not to include the current movie itself. It then sorts those movies in descending order according to their average rating. If the length is greater than 5 it will only take the first 5 movies. The result is a list (max length of 5) of movies that contain that genre that also have high average ratings (respective to the other movies). An improvement that could be made is to not use the first genre, but rather use all genres associated with the film and grab the top rated out of all movies containing those genres. This would increase the amount of data transferred and increase the computing time, but it would make the similar movies more meaningful.

# Summary

The main goal of this project was to create a web server that implemented the required features, which was completed. The overall design of the project focused on code quality, RESTful design principles, proper HTTP status codes and error handling, use of asynchronous operations, and minimization of data transfer. The code quality overall was well organized, the function and variable names were well defined, and the code was easy to understand. However, the improvements to be made to increase the quality would be to create routers for users, movies, and people to break the code down even further into more manageable chunks. The RESTful design implemented most of the principles well, but it needs some improvement on the principle of “nouns at endpoint paths”. The HTTP status codes were done correctly. The errors were handled properly, but some improvements can be made to make the error handling more consistent and to inform the client as to why the error occurred. Asynchronous operations were used when required, and their callback functions were meaningful. The data was minimized well using pagination and the mongoose ‘select’ feature, although the get movies function could possibly be improved to minimize the data transfer even further. The two algorithms work as designed without any errors. However, they could be improved further to reduce the time complexity and/or increase the meaningfulness of the recommended/similar movies. Altogether the project implemented all required features with a strong overall design, and algorithms that produce expected outputs, but also leaving some areas that need improvements.

# Bibliography

|  |  |
| --- | --- |
| [1] | J. Au-Yeung, "Best practices for REST API design," 2 March 2020. [Online]. Available: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/. [Accessed April 2020]. |

# Appendix A

## A1. Required Features

|  |  |
| --- | --- |
| **USER ACCOUNTS** | |
| **Feature #** | **Description** |
| 1 | Create a user (username and password) |
| 2 | Users can login |
| 3 | Users can logout |
| 4 | Users can switch between regular and contributing |
| 5 | Users can view and mange the people they follow. They must be able to navigate to any person they follow and be able to stop following that person. |
| 6 | Users can view and manage users they follow. They must be able to navigate to any user they follow and be able to stop following that user. |
| 7 | View a list of movies they have watched |
| 8 | Remove any movies from their watched list |
| 9 | View recommended movies |
| 10 | View notifications about people |
| 11 | View notifications about users |
| 12 | Search for movies by title, person name, and/or genre |
| 13 | Search should only show 10 movies (supports pagination) |
| **VIEWING MOVIES** | |
| 14 | User must see basic info about the movie |
| 15 | See each genre keyword and navigate to search results for that genre |
| 16 | See directors, writers, and actors and navigate to the people’s pages |
| 17 | See list of similar movies to this one and navigate to them |
| 18 | Add movie to watched list or remove movie from watched list |
| 19 | See movie reviews that have been added for the movie |
| 20 | Add basic review by specifying score out of 10 |
| 21 | Add full review by specifying score out of 10, summary, and full review text |
| **VIEWING PEOPLE** | |
| 22 | See history of persons work with navigable links |
| 23 | Separate into writing/directing/acting work (if not directed/written/acted then category should not show!) |
| 24 | See list of frequent collaborators (top 5 people this person has worked with the most) |
| 25 | Choose to follow or unfollow this person |
| 26 | If user follows this person, user should receive a notification whenever a new movie is added that contains this person |
| **VIEWING OTHER USERS** | |
| 27 | See list of all reviews this user has made and be able to read each review |
| 28 | See list of all people this user follows and navigate to them |
| 29 | See users watched movies list and navigate to those movies |
| 30 | Choose to follow or unfollow this user |
| 31 | If X follows Y, X should receive a notification whenever Y creates a new review |
| **CONTRIBUTING USERS** | |
| 32 | Navigate to add person page |
| 33 | Add person to database (if name exists already then don’t add the person) |
| 34 | Navigate to the add movie page |
| 35 | Create a new movie by specifying the title, release year, runtime, plot, genres, and at least one writer, director, and actor. |
| 36 | Page must give user a way to dynamically search for people within database to add to the movie |
| 37 | User should not be required to type full name but instead should be able to add person directly from search results |