Name, Student Number
- Yun Hye Nam: 101211656
- Parisha Haque: 101118747

Detailed Outline of Data Model

Movie Objects

The movie-data.json object contains all the movie data, like the title, year, rated, released, runtime, genre, director, writer, actors, plot, awards, and poster. The variable mov parses the data from movie-data.json object and stores it. Then we created an empty object called `movies` (which will store array of each individual film), and initiated a variable id to 0. We then used for forEach function to parse each element from mov variable and then added associated Review objects (which stores these keys and their values; reviewUser, reviewRating, reviewText, reviewId, and reviewBrief), the (Average) Rating key and its value, and a movie ID key and its value. The ID is incremented for each element. Then we added the entire film object, which a temporary variable which stores the first json object and increments until every object is added into the movies array.

The format of the object:

```
movies = {
        “t0” :    { Title: “movie_title”,
                    Year: “movie_year”,
                    Rated: “PG”,
                    Released: “released_date”,
                    Runtime: “runtime_minutes”,
                    Genre: [ “genre_1”, “genre_2” ],
                    Director: [ “directorName_1”],
                    Writer: [ “writerName_1” ],
                    Actors: [ “actorName_1”, “actorName_2” ],
                    Plot: “plot”,
                    Awards: “nominated…”,
                    Poster: “poster_url”,
                    Review: [
                        {
                        reviewUser: “reviewer_username”,
                        reviewRating: “#/10”,
                        reviewText: “full_review_text”',
                        reviewBrief: "brief_summary",
                        reviewId: “review_id”
```

```
                }
                , {reviewobject_2}, ...],
            Rating: “#/10”,
            Id: 't0'
        },
“t1” : {……},
….
}
```

Properties and Values:

- Title: string representing the movie title
- Year: string representing the movie year (written in numbers but in string)
- Rated: string representing how the movie is rated “PG”, “G”, etc.
- Runtime: string representing the movie runtime in minutes ex) “120 min”
- Genre: an array of strings that represent each genre of movie
- Director: an array of strings that represent the names of the directors. This is basically referencing the Person object’s Name property.
- Writer: an array of strings that represent the names of the writers. This is basically referencing the Person object’s Name property.
- Actors: an array of strings that represent the names of the actors. This is basically referencing the Person object’s Name property.
- Plot: a string representing the movie’s plot in sentences.
- Awards: a string representing the award information of the movie
- Poster: a string representing the URL to the movie’s poster.
- **Review**: an array of review objects. For each review object, the property and value are
    - reviewUser: a string representing the reviewer’s username
    - reviewRating: string representing the reviewer’s rating for the movie
    - reviewText: string representing the brief summary of review written by reviewer
    - reviewBrief: string representing the full review written by the reviewer
    - The review object will be included in the user object as well.
- Rating: a string representing the average rating in the denominator out of 10. ex) “9/10”
- Id: a string representing the movie’s id. The first character is “t”. As we iterate through the mov variable (array of movies parsed from JSON file), the Id is incremented by 1, and together with the first character, it has the format “t0”, “t1”, etc.

Person Objects

We created an empty object called people, initialised variable v to 0, and created an empty array called allpeople. We then used the forEach function, we check if the person in the selected element's Actors, Director, or Writer array exists in the people object by searching "allpeople" array, if not, then we add it to the object, by running validation.

First, if the person is already in the people object, then we add the person's movie's ID to the pre-existing ActedMovie array, DirectedMovie array, or WroteMovie array depending on how the person appears in the movie. If these arrays don't exist, then we create one with the movie ID added to it.

If the person is not in the people object, then we add the details of the person into the people object: ID, Name. As well we also add the movie ID. for ActedMovie/DirectedMovie/WroteMovie array, Then we added the person's name to the array allpeople to have a list of all the people (without duplicates).

For each procedure, check if the movie element's ID is in the AllWork array. If not, add it to the array.

Format:

```
people = {
        "p0": {
                Id: "p0",
                Name: "person_name",
                ActedMovie: ["movieID_p0_actedIn", "movieID2_p0_actedIn", ... ],
                DirectedMovie: [ "movieID_p0_Directed", "movieID2_p0_Directed", ... ],
                WroteMovie: [  "movieID_p0_Wrote", ...]
                AllWork: [ "t0", "t1", ...]
            },
        "p1": { …},
        ….
}
```

Properties and Values:

- Id: string representing the Person's id. It is in the format: "p" + #. The number is incremented by iterating through the mov array's Actors array, Director array, and Writer array. Ex) "p0", "'p1", ...

- Name: string representing the Person's full name.
- ActedMovie: the array of strings representing the movie IDs that the Person acted in. So if the movie's Actors array contains the Person, the movie ID is added to the ActedMovie array. If the person has never acted in any movie, the Person object will neither contain the ActedMovie property nor value.
- DirectedMovie: the array of strings representing the movie IDs that the Person directed. So if the movie's Director array contains the Person, the movie ID is added to the Person's DirectedMovie array. If the person has never directed any movie, the Person object will neither contain the DirectedMovie property nor value.
- WroteMovie: the array of strings representing the movie IDs that the Person wrote. So if the movie's Writer array contains the Person, the movie ID is added to the Person's WroteMovie array. If the person has never wrote any movie, the Person object will neither contain the WroteMovie property nor value.
- AllWork: the array of strings representing the movie IDs of all movies that the person directed, wrote, or acted in (since some people both write and direct a movie). There will be no duplicate movie ID for the array. So this array will be used for keeping track of frequent collaborators for each person. We can find how many collaborations each person had with every other person by counting how many elements occur both in one person's AllWork array and another's AllWork array.

User Object

We created a users object to store all the user data. We used their username as the key and added their corresponding values, such as Username, Password, AccountType, Review (which is an array of review data containing information about the reviews such as such as reviewUser, reviewRating, reviewId reviewText, and reviewBrief), followingUsers array, followingPeople array, WatchedMovies array, and Notifications array.

Format:

```
users = {
        "user00_name": {
                Username: "username",
                Password: "password",
                LogIn: "True/False"
```

```
            AccountType: “Contributor/Regular”
            Review: [ {movie_p0_directed}, {movie2_p0_directed}, ... ],


            Review: [
                {
                reviewUser: “reviewer_username”,
                reviewRating: “#/10”,
                reviewText: “full_review_text”',
                reviewBrief: "brief_summary"
                reviewId: “review_id”
                }
                , {reviewobject_2}, ...],
        },
            FollowingUsers: [  “username_following”, ...],
            FollowingPeople: [  “personID_following”, ...],
            WatchedMovies: [“movie_Id”, “movie_id2”,...],
            Notifications: [“notification1”, “notification2”,...]
        },
    “user01_name”: { …},
    ….
}
```

Properties and Values:

- Username: string represents the unique username of a user.
- Password: string represents the password of that user.
- AccountType: string representing this users account type (between Regular and Contributing).
- FollowingUsers: array of strings representing the Username of other users that the object’s user follows.
- FollowingPeople: array of strings representing the ID of Persons that the object’s user follows.
- WatchedMovies: array of strings representing the ID of movies that the object’s user watched.
- Notifications: array of strings each representing the notifications of the object’s user.
- **Review**: an array of strings representing review Ids. In the check-in, it will be a copy of the review object in a Movie object, but later, this will only reference the movie object’s review ID.

RESTful Design

Resources are Movies, Users, People

RESOURCE: MOVIES

/movies

- Request: **GET**
- Use: This route will retrieve all movies in the database and display them as a list of 10 movies per each page. The content-type of request is text/html or application/json and the route will support html or json type. The status code of the response is 200. The data sent to this response is the `movies` object,[1] containing {movie_id}s as object Key and movie information parsed from a JSON object as corresponding Values. Or, the data sent to this response could be an html page compiled by movies.pug file, rendered with the data of `movies` object.
- For the check-in, this page will only display up to 10 movies, and pagination links will not be supported. The route will only support html and the content-type will be text/html.

/movies?

- URL format: /movies?title={movie_Id}&genre={genre_name}&actor={actor_name}&director={director_name}&writer={writer_name}&year={year_name}
- Request: **GET**
- Use: This route will enable searching movies in the database by specified query parameter values. It will retrieve a list of 10 movies per page, filtered to contain the specified parameter value in the request. The response will contain a html page (through movies.pug template engine) rendering the array of Movie objects,

[1] `movies` Object is an object acting as a collection of all movie objects. Key: {movie_id}, Value: {movie object}. The Movie object is an object representing an individual movie.

filtered from the `movies` object, so that for each Key (Title, Genre etc.), the corresponding Value contains or corresponds to specified parameter values. The content-type of request is text/html and the route will allow HTML. If successful, the status code of the response is 200.

- The supported query parameters are:
    - Title: a string that limits the movies to any movie whose title contains the string. Both lower/upper cases are supported. If no value is specified, all movies will meet this title restriction.
    - Genre: a string that limits the movies to movies whose genre array has a genre equal to the string. Both lower/upper cases are supported. If no value is specified, all movies will meet this genre restriction.
    - Actor: a string that limits the movies to movies whose actors list has an actor equal to the string. The letter cases should be exact, so that the first characters of first and last names are capitalized. If no value is specified, all movies will meet the actor restriction.
    - Director: a string that limits the movies to movies whose director list has an director equal to the string. The first characters of the first and last names should be capitalized. If no value is specified, all movies will meet this restriction.
    - Writer: a string that limits the movies to movies whose writer list has an writer equal to the string. The first characters of the first and last names should be capitalized. If no value is specified, all movies will meet this restriction.
    - Year: string that limits the movies to those whose released year is exactly the same as the parameter value. If no value is specified, all movies will meet this restriction.

  If no parameter value is specified, this route will act like “/movies”.

/contribute

- This part will only explain for adding movies. To add person, go to **PERSON: /contribute**)
- (button) Request: **POST**
- Use: This route will allow a contributing user to create a new movie to be added to the movie database. The “Add Movie" submit button allows us to perform the POST request. The data sent to this POST request is a JSON stringified new Movie object containing the required text inputs for Title, Released Year, Runtime, Genre, Actors, Directors, Writers as the object’s Values. Also in the request, the new Movie object will be added as a Value to the `movies` object, with its movie ID as the Key. The status code of the response is 201. The

content-type of request could be application/json, for which the route will respectively support JSON.

- For the Actors, Directors, Writers text bars, it should allow GET requests similar to the "/movies?". This is to ensure that when the user types their names, the space beside the text bars should display the search result list of person names to be added to the movie.
- But to display this URL route, use GET request. The contribute.pug file will be rendered into html in the response. Status code of the response is 200. The route will support HTML, and the content-type of request will be text/html. However, the server must first check whether the URL-requesting user is stored as a user object with contributor account type.

/movie/{movie_id}

- URL Request: **GET**
- Use: This route allows for viewing movie information. It retrieves the movie object with the ID, the parameterized route *{movie_id},* in the GET request, to render the page with the movie's Poster, Title, Released Year, Runtime, Average Rating, Released date, Plot, Genres, Directors, Actors, Writers, Reviews, Similar Movies (8 movies with the same genre). The API will first check whether the movie_id is accurate. The Content-type of the request is text/html, the status code of response is 200. The route will support HTML. The response will contain an html page compiled through movie.pug file. This pug file is rendered with data which includes `movies` object, people object, and a single movie object specified by the parameterized route.
    - The single movie object will display information about the requested movie with the id {movie_id}.
    - The `movies` object will be used to render similar movies
    - The people object will be used to render directors, actors, writers.
- <u>The Add to Watch List Button</u>
    - Request: **POST**
    - Use: This button will add the movie to the list of movies that the logged-in user watched using the POST request. The data sent to the request is the movie object, whose id is {movie_id}. This object will be added in the `users` object, more specifically, the logged-in User object's[2] WatchedMovies array. The status code of the response is 200. The content-type is text/html, and the request will support HTML.
    - For the check-in, it is linked to "/profile" page.
- <u>Submit Basic Review / Submit Full Review Buttons:</u>

[2] `users` Objects: object which is basically a collection of all user objects. User object is basically representing a single user.

    - Request: **POST**
    - Use: This button will add the brief review and/or the full review to the Movie object's Review array and the User object's Review array. The data sent in the POST request will contain the brief review, rating, the user's id, and the full review. The status code of the response is 200. The content-type is text/html, and the request will support HTML.
    - For this particular check-in, it performs a GET request in form(submit) to go to the route "/review/reviewpage".

/review/{review_id}

- URL Request: **GET**
- Use: The route will display a review (both brief and full) for a movie, the reviewer username, and the reviewer's rating by retrieving the review object specific to the parametrized route, from a Movie object's Review array. This is done by GET request. The content-type is text/html and the request will support HTML. The status code of the response is 200. The data sent to this response is an html page compiled by reviewpage.pug file, rendered with a Movie object data.
- For the check-in, it will be just linked to "/review/reviewpage".

/search

- Request: GET
- Use:
    - Contains a form to perform an advanced search by searching the movie data for keywords that are inputted by the user. The user can search based on the title, released year, genre, actor, director, writer. The response will contain a html page compiled through advsearch.pug file. The pug file compiles to a html page, but is not rendered with a particular movie data. However, when pressing the search button, the request should contain search query parameter values for "/movies?". The status code is 200. The content-type is text/html, and this route supports html.
- Search Button
    - Contains a submit button which performs a **GET** request to route "/movies?" (refer to /movies? above). The data sent in the GET request is the user's inputs for title, released year, genre etc. In the request, inputs will be added into parameterized routes "/movies?{parameter1}={value1}&...." to retrieve the search result from an array of movie objects. For more, read the section "/movies?".
        - For the check-in, it will only link to a default search page

- The search bar performs a nearly identical function using GET request, however, the user can only write one of Title, Actor, Genre as inputs to be sent in the response. The HTML page rendered with searchbar.pug in the partial folder. The requested route will be the same as /movies?Title=...&..

RESOURCE: USERS

/profile

- Request: GET/PUT
- Use:
- URL Request: **GET**
  - This route displays the profile of the profile holder, containing information like their username, whether they are a regular user or a contributing user, which people they are following, which users they are following, which movies they watched, reviews written by them, some movie recommendations based on what they have already watched, and their notifications. All of this is done by checking whether the user is logged-in (by checking the boolean in the user object's "LoggedIn"), and if so, by retrieving the User object's Username, Account Type, FollowingUsers, FollowingPeople, WatchedMovies (watched list), Notifications, Reviews using a GET request.
  - The content-type of the request is text/html. The route will support HTML. The data sent to the response contains html compiled through profile.pug, rendered with the User object, people object, `movies` object, the `users` object. The status code of the response is 200.
    - The user object will be used to render logged-in user infos
    - `movies` object will be used to render watched list movies and recommended movies.
    - People object will be used to render people you are following.
    - `users` object will be used to render users you are following.
  - The notification section only displays upto 25 characters. If notification is longer than 25 characters, it displays the first 25 characters followed by ellipses ("...").
- Unfollow Button
  - User: The button allows the user for unfollowing other users and persons.
  - **PUT** request for Unfollow. If the user wants to unfollow the other user, the data sent in the request is the other user's username, so that it is removed from the logged-in user's FollowingUsers array. If the user wants to unfollow the person, the data sent in the request is JSON stringified

person's ID, so that it is removed from the logged-in user's FollowingPeople array. The status code is 200. The Content-type supports application/json and the server will support JSON.

- Remove Buttons (For movies in the watched list)
    - **PUT** request:
    - This page contains "remove" buttons that are used to remove movies from the watched list. This is done by PUT request. The request should send the JSON stringified, specified Movie object to the server for it to be removed from the User object's WatchedMovies array. The status code is 200. The Content-type supports application/json and the server will support JSON.
- Delete Buttons (For reviews and notifications)
    - **PUT** request:
    - This page contains "delete" buttons that are used to delete reviews posted by the user and notifications. This is done by PUT request.
        - For the review, the request should send the JSON stringified, specified user's review object to the server for it to be removed from the User object's Review array and the Movie object's Review array.
        - For the notifications, the request should send the specified notification to be removed from the User's Notifications array.
    - The status code is 200. The Content-type supports application/json and the server will support JSON.
- Save Button (for account type change)
    - **PUT** request
    - A "save" button that is used to change the account type of the profile holder between regular and contributing. This is all done using PUT requests. The request contains the AccountType "Contributing" or "Regular", and the server must change the User object's AccountType to Contributing or Regular based on the radio item specification. The status code is 200. The Content-type supports application/json and the server will support JSON.
- The names of the followed people are hyperlinked to "/people/{personId}". The names followed users are hyperlinked to "/user/{user_id}". The names of the watched movies and recommended movies are hyperlinked to "/movie/{movie_id}". "View" button beside the notification will link to "/notification". This is all done using GET requests (For more, read individual sections).
- For the check-in, save, unfollow, remove, and delete buttons do not function, however the view button does. Also, the default profile holder is "harry01" who has a regular account type.

/notification

- Request: GET/PUT
- URL (Request: **GET**)
    - This route displays a list of all the notifications of the profile holder from the `users` object by performing GET requests. The content-type is text/html and the request will support HTML. The status code of the response is 200. The data sent to this response is an html page compiled by notification.pug file, rendered with the logged-in User object data.
- Delete Button (Request: **PUT**)
    - This page contains a delete button which deletes the specific notification from the list using PUT requests. To read more, go to the section **/profile.**
    - For the check-in, delete buttons remain nonfunctional/

/user/{user_id}

- Request: GET/PUT
- Use:
    - This route displays information about users, including their username, list of people they are following, movies they have watched, their reviews if they have any, and their account type. This is done by retrieving the parameterized route (user_id)'s User object in the GET request. The content-type of the request is text/html, the route will support HTML. The status code of the response is 200. The data in the response will contain html compiled through otheruser.pug, rendered with the User object specified by the parameterized route, `movies` object, people object, and `users` object.
        - The user object will be used to render the request user's infos.
        - `movies` object will be used to render watched list movies and recommended movies by referencing the specified User object's properties.
        - People object will be used to render people you are following by referencing specified User object's properties
        - `users` object will be used to render users you are following by specified User object's properties.
- Follow/Unfollow Button
    - User: The button allows the user for following and unfollowing other users. The value of the button will be updated to become "follow" upon clicking the "unfollow" button (and vice versa), allowing for different requests to be performed using the same button.

    - **POST** request for Follow. If the user wants to follow the other user, the data sent in the request is the other user's username, so that it is added to the logged-in user's FollowingUsers array. The status code is 201. The Content-type supports application/json and the server will support JSON.
    - **PUT** request for Unfollow. If the user wants to unfollow the other user, the data sent in the request is the other user's username, so that it is removed from the logged-in user's FollowingUsers array. The status code is 200. The Content-type supports application/json and the server will support JSON.
        - For the check-in, this button is set to "follow" with no changes on click.
    - The names of the people following are hyperlinked to "/people/{personId}". The names of the watched movies are hyperlinked to "/movie/{movie_id}". This is all done using GET requests.

/signin

- URL Request: **GET**
- Use:
    - This route allows us to sign into the website, as long as the user ID and password are correctly inputted. ID and Password information is found in users object. The Content-type of request is text/html, the response status code should be 200 if successful. The data in the response is the signin.pug file.
- Sign-In Button
    - **POST** request
    - It contains the Sign In button which will be used to access the profile holder's profile once validation is run on the ID and Password. The data sent in the request is the username and password. The status code of the successful response is 201. The content-type of the request could be application/json, for which the route will support JSON. The JSON string of new object will be created.
- Create Account Button
    - **POST** Request.
    - This create account button will be used for adding a new user to the database. The Username should not be the same as another user.
    - For the check-in, the Sign Up page has not been implemented yet. It will use POST requests to add a new user object to the database. The new object will contain the user-specified Username, Password, Regular Account Type, set LoggedIn as True and will have default empty arrays for other properties. This new object will be sent in the request and added in

the `users` object. Validation checks will be implemented to ensure username is not already taken. The status code of the successful response is 201. The content-type of the request could be application/json, for which the route will support JSON. The JSON string of a new object will be in the response.
    - For the check-in, Create Account button which will take us to a Profile page. This button will also use GET requests.
        - For the check-in, both the Sign In button, and the Create Account button takes us to user harry01's profile by default, and the validation checks have not been implemented yet.

Sign Out

- URL: /{any url}. It is in the partial HTML page generated by header.pug.
- Request: **PUT**
- Use:
    - This Sign Out tab present on the header will use **PUT** requests to sign the profile holder out of the website. The data sent in the request will contain the logged-in user object. The user who is logged in will have its object's LoggedIn property set to False. The status code of the successful response is 200. The content-type of the request could be application/json, for which the route will support JSON. The response will have a JSON string of the changed user object.
        - For the check-in, it will only redirect to the home page ("/").

<u>Resources: PERSON</u>

/people

- Request: **GET**
- Use: This URL will retrieve all persons in the database, listing 10 people per each page. The content-type of this request is text/html or application/json. The route will support either html or json. For the response, the status code will be 200. The data in the response may be the people object, whose Keys will be each {person_id} and Values will be person's information. Or else, the data sent to this response could be an html page compiled by people.pug file that is rendered with the people data.
- For the check-in, this page will only display up to 10 people, and pagination links will not be supported. The route will only support html and the content-type will be text/html.

/people?name={people_name}

- Request: **GET**
- Use: This route will enable searching persons in the database by specified query parameter value (person's name) in the GET request. It will retrieve a list of 10 people per page, filtered to contain the specified parameter value. The content-type of the request is text/html and the route will allow HTML. The status code of the response is 200. The response will contain a html page (compiled through people.pug) rendering the array of person objects, filtered from the People object, so that for the person's name contains the parameter value string.
- The supported query parameter is the person's "name":
    - Name: a string that limits the persons to any person whose name contains the string. Both lower/upper cases are supported. If no value is specified, all persons meet this constraint.

/people/{person_id}

- (URL) request: **GET**
- This route displays information about a person: the list of movies they acted in, directed, and/or wrote, and 5 other persons they most frequently collaborated with. This is done by retrieving the parameterized route (person_id)'s Person object in the GET request. The content-type is text/html. The status code of the response is 200. The route will support HTML. The data in the response is a HTML generated by exampleperson.pug file, rendered with the Person object specified by the parameterized route, frequent collaborators object, and `movies` object. In the API, we will generate frequent collaborators object based on how many AllWork elements the Person and other people share.
    - A specific Person object for displaying their information
    - A "frequent" (collaborators) object displaying a list of 5 people the person most worked with
    - `movies` object used for displaying the movies, referencing the movie-Ids in Person's ActedMovie, DirectedMovie, and WroteMovie arrays.
- Follow/Unfollow Button
    - Use: The button allows the user for following and unfollowing a Person. The value of the button will be updated to become "follow" upon clicking the "unfollow" button (and vice versa), allowing for different requests to be performed using the same button.
    - **POST** request for Follow. If the user wants to follow a Person, the data sent in the request is the Person's ID, so that it is added to the logged-in user's FollowingPeople array. If the user wants to follow the person, the data sent in the request is the person's ID, so that it is added to the

logged-in user's FollowingPeople array. The status code is 201. The Content-type supports application/json and the server will support JSON.

- **PUT** request for Unfollow. If the user wants to unfollow the person, the data sent in the request is the other person's ID, so that it is removed from the logged-in user's FollowingPeople array. The status code is 200. The Content-type supports application/json and the server will support JSON.

/contribute

- This part will only explain adding a person. To add a movie, go to **MOVIES: /contribute**)
- (button) Request: **POST**
- Use: This route enables contributors to add a person to the database by using POST requests. The "Add Person" submit button allows us to perform the POST request. The data sent to this POST request is JSON stringified new Person object containing the user's text inputs for Name, Movies one acted in and/or directed in and/or wrote. In the request, the Person should be added to a people object as one of its values (with the new person's id being the Key). Also in the request, the person will be added to an existing Movie object's Actor, Director, and/or Writer array. It will be added in the people object, with person_id as key and the full object as values. However, the API must check whether the pre-existing people object contains the person's name to disable duplicates. The status code of the response is 201. The content-type of the request could be application/json, for which the route will support JSON.
- But to display this URL route, use GET request. The contribute.pug file will be rendered into html in the response. Status code is 200. The route will support HTML, and the content-type will be text/html. However, the server must first check whether the URL-requesting user is stored as a user object with contributor account type.

For all wrong URLs or unsuccessful methods:

- The status code will be 404 for client side error.