Iteration 2
Team Name: GUO

Ahmed Alzubairi (afa2135), Michael Stone (ms5512), Mauricio Antonio Quintanilla (maq2119), Christian Kowalczyk (cjk2159)

part 1:

Revise your user stories, from the first iteration and/or project proposal, to reflect what is now fully implemented, tested and working. **Do not include anything that is not actually evident in your github repository and/or you will not be able to show in your final demo.**

Answer:

User Stories:
<Creating an account> As a movie hobbyist, I want to be able to create an account with my Gmail for a movie related website of my interest.

<Organizing List> As a movie hobbyist, I want a place to store movies on my watch list so that I can have an organized list of my favorite movies. My condition of satisfaction is that I am able to search for any movie and be able to have it into my list.

<Anticipating Top Movies> As a film critic, I want to know what the top movies are. My conditions of satisfaction is that if I click some button or some functionality, I should be able to see some of the top few IMdB movies.

<Movie Information> As a busy person I want to know if a movie has a good plot before watching it so that I don't feel like I wasted my time watching a bad movie. My condition of satisfaction is that I can search for a movie on the website and be given some rating, plot, or other information.

<Meeting for movie> As a solo movie watcher, I want to be linked with others who plan to watch a movie, so that I can watch it virtually with them. My condition of satisfaction is that I am able to get a message or some sort of notification for a Zoom Link where I can watch the movie with others.

Acceptance Testing & Discussion of Acceptance Testing:

<Creating an account> User goes to the home page at the local host (https://127.0.0.1:5000/) and clicks the Google Login in navbar.

a. In a valid input, a user enters a valid Gmail and password, creating an account and logging them in. Users are signaled that they are logged in because they get redirected to the home page with their profile picture and name in the Navbar.
b. In an invalid input, a user inputs an invalid Gmail and password. A MovieLink account is not created, and the user is alerted by Google that the login is not correct. They are not taken to the user homepage.

During our testing:
Valid Input: User went to the home page and logged in with valid credentials.
We inputted ahmedalzubairi1@gmail.com and password=correct_pass into the Google login screen. We were taken to the user home page and saw the name Ahmed Alzubairi displayed in the Navbar. We saw a blank list of movies & a blank list of friends as it is a new account. There were **no bugs** to report when we tested this.

Invalid Input: User went to the home page and clicked login, but had incorrect password.
We inputted ahmedalzubairi1@gmail.com but with the password=incorrect_pass to the login screen. We were is alerted by Google login form that the login was invalid. There were **no bugs** to report when we tested this.

<Organizing List> User logs in, and is taken to the user home page with a blank wish list of movies. They want to add to this list.

a. In a valid input, they can use the search bar to look up a specific movie, click the right movie, and click a button to add it to their wish list.
b. In an invalid input, the user searches for a movie that doesn't exist and is taken to a screen with no movies to choose from. They can go back to the user home page or attempt to search again.

During our testing:
Valid Input: User went to the user homepage after logging in and searched up a valid movie and added to their wish list.
We searched for the movie "Shrek," and saw a list of Shrek movies. We were able to click on the original Shrek, add it to our wish list, which made it appear back on the user home screen under wish list. There was initially a bug during this acceptance testing where we attempted to display more movies than existed in the database after a given search (leading to an error in the terminal), but we were able to **fix** this bug.

Invalid Input: User went to the user home page after logging in, and searched a movie that did not exist.
We searched for a movie "asd!f--jkl", saw a blank list because that movie does not exist. We were able to exit back to the user home page with the navigation bar. One bug to report here is when you search for a (no entry in the text bar) an error occurs. Other than that, there were no bugs on invalid inputs.

<Anticipating Top Movies> User logs in, and is taken to the user home page and sees a random selection of the top 250 movies from IMDb. They can also generate a new random set of movies.

a. In a valid input, the user can click on a "Generate Top Movies" link below the list of top movies and it will refresh this list.
b. There is no invalid input for this user story, because the list is auto generated on the loading of the user home page, and the "Generate Top Movies" simply refreshes this page.

During our testing:
Valid Input: We went to the user homepage after logging in and were able to see a subsection of the top 250 IMDb movies.

We clicked on the “Generate Top Movies” button, which refreshed the page and updated the list with a new set of quality movies.

Invalid Input: N/A. You can’t have an invalid input because the button is guaranteed to give you movies.

<Movie Information> User logs in and wants to see information on a valid movie that exists.

a. In a valid input, the user can search for a movie that does exist. They will get a selection of movies with a match name, and be able to click on one to get more information.
b. On an invalid input, the user might (again) search for a movie that does not exist, or search for a movie that has yet to have any information.

During our testing:
Valid Input: User logs in and looks for information on a valid movie that exists.
We searched for American Psycho, were given a list of movies with similar titles, clicked on American Psycho and were able to read the plot summary. There were **no bugs** to report.

Invalid Input: User looking for information on a invalid movie that doesn’t exist or doesn’t exist yet.
We searched for The Matrix 4, and were returned a movie that we could click on that has yet to come out. The **bug** was that it had no information and should probably not be included in the results. We were unable to fix this for this iteration, because many movies in the IMDb database are likely similar and upcoming.

<Meeting for a movie> User wants to create a session with all of his friends to watch a specific movie and notify them in some way.

a. In a valid input, the user can search for a movie that does exist. They will get a selection of movies with a match name, and be able to click on one and then click ‘add session’. This will send an email out to all of their friends.
b. On an invalid input, the user might (again) search for a movie that does not exist, or search for a movie that has yet to have any information. However, in terms of adding the session itself because it is a button click there is likely no invalid input. Another possible “invalid” input is if they were to enter a faulty link or not include the date/time. Our current system just sends the information that is given in the form though, without demanding for valid inputs.

During our testing:
Valid Input: User wants to create a session with all of his friends to watch a specific movie that exists and notify them in some way.
We searched for Psycho, were given a list of movies with similar titles, clicked on Psycho and were able to click “add session”. This sent an email out to everyone on the friends list (which was another account that we had logged into the system from the local database). There were no bugs to report.

Invalid Input: User looking for information on a invalid movie that doesn’t exist or doesn’t exist yet.
We searched for The Matrix 4, and were returned a movie that has yet to come out. The **bug** was that it still had the ability to add a session, even though you would never actually be able to watch that movie with a friend.

---

part 2

Write a test plan that explains the ***equivalence partitions*** and ***boundary conditions*** necessary to unit-test each of the major subroutines in your system (methods or functions, excluding constructors, getters/setters, helpers, etc.) and then implement your plan. Associate the names of your specific test case(s) with the corresponding equivalence partitions and boundaries (if applicable). Your test suite should include test cases from both ***valid*** and ***invalid*** equivalence partitions, and just below, at, and just above each equivalence class boundary (or inside vs. outside the equivalence class when boundary analysis does not apply). Note the same test case might apply to multiple equivalence classes. Say there is a method whose input should be an integer between 1 and 12. Then there is an equivalence class 1-12, an equivalence class <1 (or <=0), and an equivalence class >12 (or >= 13). A test case with input 0 would be just below the lower boundary of the equivalence class 1-12 and also at the high boundary of the equivalence class <1.

Include the link to the folder in your github repository that contains your automated test suite.

Answer:

Testing Equivalence Partitions & Boundary Conditions:

1. Models.py
    a. parseRowFriends(friend_row): Given a row of friends parse them into a list. The function expects to take in a dbs row that has the form [‘name1:email1,name2:email2’] and returns [‘friend1’,’friend2’].
        i. Equivalence: valid partitions of the input would be rows dbs rows in the form [name:email] and invalid partitions would be rows that don't follow such format, or are simply empty.
        ii. Boundary conditions: empty row is passed

iii. Test methods: test_parseRowFriendsExist(), test_parseRowFriendsEmpty(), test_parseRowFriendsInvalidRow()

b. parseTopMovies(movie_row): Given a row of movies parse them into a list and return the first 10 movies. Function expects to take in a dbs row in the form [‘movie1,movie2’] and return [‘movie1’, ‘movie2’]
   i. Equivalence: valid partitions are rows in the form [‘movie1,movie2’] and invalid partitions would be rows that don’t follow that format.
   ii. Boundary: an empty row is passed into the function
   iii. Test methods: test_parseTopMovies(), test_parseTopMoviesUnder10(), test_parseTopMoviesOver10(), test_parseTopMoviesEmpty(), test_parseTopMoviesInvalidRow()
c. parseAllMovies(): Returns all movies in the dbs in the form [movie1,movie2,...,movieN]
   i. Equivalence: There is no input this is a get method
   ii. Boundary: Same as equivalence
   iii. Test methods: test_parseAllMovies(), test_parseAllMoviesEmpty()
d. parseSession(session_row): given a row of sessions parse it into a list. Function expects to take in a row in the form [‘movie1:time:link%movie2:time:link’] and returns [‘movie1:time:link’, ‘movie2:time:link’].
   i. Equivalence: valid partitions would be rows of the form [‘movie1:time:link%movie2:time:link’], and invalid partitions would be anything not in that form or an empty row

ii. Boundary: an empty row

iii. Test methods: test_parseSession(), test_parseSessionEmpty(), test_parseSessionInvalidRow()

e. parseFriendsEmails(friends_row): given a row of friends:emails parse it into a list of just emails. Functions expects to take a row in the form [‘friend1:email1,friend2:email2’] and returns [email1,email2].

i. Equivalence: valid partitions would be rows of the form [‘movie1:time:link%movie2:time:link’], invalid partitions would anything in that form or an empty row.

ii. Boundary: an empty row

iii. Test methods: test_parseFriendsEmails(), test_parseFriendsEmailsEmpty(), test_parseFriendsEmailsInvalidRow()

f. constructNewMovielist(current_movies,movie_name): given a list of movies and a new movie, append the new movie to the list and return. Function expects input in the form [‘movie1’,’movie2’], ‘movie3’ and returns [‘movie1’,’movie2, ‘movie3’]

i. Equivalence: valid partitions would be [‘movieName1’, … , ‘movieNameN’], ‘newMovie’. Invalid partitions can be invalid movielist (list is None), invalid movieName (empty string, dont need to worry about None condition since movieName is always an empty string by default), or both can be invalid inputs

ii. Boundary: movieName being empty

iii. Test methods: test_constructNewMovielistOneMovie(), test_constructNewMovielistEmpty(), test_constructNewMovielistInvalidList(), test_constructNewMovielistInvalidName(), test_constructNewMovielistInvalidListAndInvalidName()

g. constructNewSessionList(currsent_sessions, new_session): given a list of sessions and a new session build a new list with the new session included. Function expects input in the form of ['movie1:link:time'],'movie2:link:time' and returns ['movie1:link:time%movie2:link:time'].
   i. Equivalence: Valid partitions are when both current_sessions and new_session are in the mentioned form ['movie1:link:time'], 'movie2:link:time' respectively. Invalid format could be when current_sessions is in invalid (None), and/or if curresen_sessions is empty
   ii. Boundary: new_session being empty
   iii. Test methods: test_constructNewSessionListAddOneSession(), test_constructNewSessionListAddToNone(), test_constructNewSessionListAddEmpty(), test_constructNewSessionListEmptyNewSession(), test_constructNewSessionListEmptyNewSessionAndInvalidList()

h. constructUserList(users_list, searched_name, user_id): given a list of rows of users, a searched_name (String), and a user_id, return all Users whose

names match or have substring **searched_name**. Return a set of 3 lists (names[], emails[], pics[]). Function expects input of the form [(‘id’, 'name', 'email', 'pic')], user_name, user_id and returns (names[], email[], pics[])

   i. Equivalence: Valid input would be in the form [(‘id’, 'name', 'email', 'pic')], user_name (non_empty), user_id (non_empty). Invalid partition would be if the list of users is empty, and/or if the user_name is empty, and/or the user_id is invalid(id doesn’t exist but we can ignore this because we only use the user_id to exclude the user searching)
   ii. Boundary: User searches an empty String, or the User list is empty.
   iii. Test Methods: test_constructUserListDifferentName(), test_constructUserListSameName(), test_constructUserListNoUsers(), test_constructUserListNoneUser(), test_constructUserListNoUsersEmptySearch(), test_constructUserListNoneUserEmptySearch(), test_constructUserListEmptySearch()

i. constructNewFriendList(friend_email_list, new_friend,new_email): given a row of friend_email, and a new_friend string, and a new_email_String construct a new String in the form ‘name1:email1,...,nameN:emailN’ that included the new_friend, and new_email.
   i. Equivalence: Valid partitions are non_empty new_friend and new_email Strings. Both need to be non-empty for them to be valid.

And Invalid partitions would be if either new_friend or new_email are empty.

ii. Boundary: in this cases there wouldn't really be a boundary, A person either passes in 2 strings or they don't

iii. Test methods: test_constructNewFriendListFriendsExist(), test_constructNewFriendListNoFriend(), test_constructNewFriendListNoFriends(), test_constructNewFriendListEmptyNewName(), test_constructNewFriendListEmptyNewEmail(), test_constructNewFriendListEmptyNewEmailAndNewFriend()

j. test_getTopMovies(): gets top movies from API. No real partitions to analyze here

   i. Test methods: test_getTopMovies()

k. renderUserHome(user_movies, user_friends): this is just a helper that calls parseAllMovies() and parseRowFriends(). It retrieves the data from these two method calls and returns a set of (movies, wishlists,friends_list, images)

   i. Test methods: test_renderUserHomeWithUsers(), test_renderUserHomeWithoutUsers()

l. findMovies(movie_name): given a movie name, look it up in the api to see if it exists. Return the movie data if it does, otherwise return an empty list.

i. Equivalence: Valid partition, a non_empty movie name that exists. Invalid Partition would be a movie name that does not exist. Simply put, the movie searched either exists or it doesn't
ii. Boundary: Search string is empty.
iii. Test Method: test_findMoviesActualMovie(), test_findMoviesDoesntExist(), test_findMoviesEmptyRequest()

m. renderMovies(movieID): given a movie ID, return movie information. This is a helper function managed by the API, and unless we have the exact movie ID testing is hard.
   i. Test Methods: test_renderMovie()

n. sendEmail(friend_list, session_info): given a firiends_list and a session_info send an email with session info to friends in friend_list. Returns 'valid' if email was successfully sent and 'invalid' otherwise
   i. Equivalence: Error handling is mainly handled by email API. Valid partition is a valid email address, and an Invalid partition is anything not a valid email since the API returns invalid. Also no need to test the friend_list, and session_list since those are handled by other helper functions tested earlier.
   ii. Boundary: none since the API accepts only valid emails strictly
   iii. Test methods: test_sendEmailWithFriends(), test_sendEmailWithNoFriends()

2. user.py
   a. getAll(): retrieve all users from database

i. Equivalence: This is a getter there isn't any input that we can partition
ii. Test methods: est_get_all_exists(), test_get_all_empty()

b. get(user_id): given a user_id return the user information row if user exists:
   i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
   ii. Boundary: an empty user_id
   iii. Test methods: test_get_exists(), test_get_invalid_id(), test_get_no_users(), test_get_none(), test_get_none_no_users(), test_get_empty(), test_get_empty_no_users()
c. getFriends(user_id): given a user_id return users friends only.
   i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
   ii. Boundary: an empty user_id
   iii. Test methods: test_get_friends_friend_list(), test_get_friends_invalid_id(), test_get_friends_no_users(), test_get_friends_empty(), test_get_friends_no_users_empty(), test_get_friends_no_friends()
d. getMovieWishlist(user_id):given a user_id return users wishlist.
   i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
   ii. Boundary: an empty user_id

iii. Test methods: test_get_movie_wish_list_movie_list_exists(), test_get_movie_wish_list_movie_list_invalid_id(), test_get_movie_wish_list_movie_no_user(), test_get_movie_wish_list_movie_list_empty(), test_get_movie_wish_list_movie_list_no_id(), test_get_movie_wish_list_movie_no_id_and_no_user()

e. getSessions(user_id): given a user_id return users session list.
   i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
   ii. Boundary: an empty user_id
   iii. Test methods: test_get_sessions(), test_get_sessions_no_users(), test_get_sessions_invalid_id(), test_get_sessions_empty_list(), test_get_sessions_empty_input()

f. addToWishList(user_id, movie_list): given a user_id, store movie_list [‘movie1,movie2,...,movieN] into the users wishlist.
   i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
   ii. Boundary: Empty user_id and/or empty movie_list
   iii. Test methods: test_addToWishlist_non_empty_list(), test_add_to_wishlist__empty_list(), test_addToWishlist_no_users(), test_addToWishlist_empty_input(), test_addToWishlist_invalid_id()

g. addSessionToWishList(user_id, session_list): given a user_id, store session_list [‘movie:time:link,...,movieN:timeN:linkN] into the users sessions.
    i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
    ii. Empty user_id and/or empty session_list
    iii. Test methods: test_addSessionToWishlist(), test_add_Session_to_wishlist_empty(), test_add_Session_to_wishlist_no_users(), test_add_Session_to_wishlist_empty_input(), test_add_Session_to_wishlist_invalid_id()
h. addFriend(user_id, friend_name_email): given a user_id, store friend_name_email name:email into the user’s friend_list.
    i. Equivalence: Valid partition would be existing users, invalid partition would be user_ids that dont exist.
    ii. Empty user_id and/or empty friend_name_email
    iii. Test methods: test_add_friend_existing(), test_add_friend_empty(), test_add_friend_no_users(), test_add_friend_empty_input(), test_add_friend_invalid_id()

3. sessions.js
    a. movieSearch ()
        i. Equivalence: User searching with **no movie** in input (or some other “invalid” input) which will lead to 0 results, or searching with user inputted **valid movie** in search bar. Note, there is no input that is not accepted, it would just take you to a page with 0 results like

most search bars, so I tested that the redirect to the right search page was performed properly no matter whether or not the input was logically “valid”.

ii. Boundary: Empty/blank search.

iii. Test methods: test(Blank movie search test', () => { })

test(Movie search test with movie', () => { })

b. friendSearch ()

i. Equivalence: Searching with no friend in input (or some other invalid input) which will lead to 0 results, searching with a user inputted friend in the search bar (which should take the user to /searchUser/[name)]. Similar to (a).

ii. Boundary: Empty/blank search.

iii. Test methods: test(Blank movie search test', () => { })

test('Friend search test with friend', () => { })

c. quickAdd (movieName)

i. Equivalence: This involves clicking a button to add a movie from the top 10 to the wish list. There is no potential for an invalid input, it is either a click that executes the function or a misclick that does not. The two parts that were tested were (1) the immediate HTML change to remove the need to refresh the page (2) the ajax call that adds the movie to the database so it is saved to the user’s wish list.

ii. Boundary: N/A (clicked vs. not clicked)

iii. Test methods: (1) test('Quick add test', () => { }),

(2) it('Quick add ajax test', () => { })

d. refreshTopMovies ()

i. Equivalence: Again, this involves clicking a single button to generate new movies. There is no potential for an invalid input, it is either a click that executes the function or a misclick that does not. The validity of the ajax call was tested.

ii. Boundary: N/A (clicked/not clicked)
iii. Test methods: it('Refresh top test', () => { })

e. updateHTML() This was a helper method for refresh top movies to update the HTML with the retrieved data
i. Equivalence: The two equivalence classes here would be a blank string from the IMDb API if some error occurred during the generation of 10 movies, or a valid string of *x* movies.
ii. Boundaries: receiving a blank list from the IMDb API, or the expected string of 10 movies from the API.
iii. Test methods: test('Update HTML test valid', () => { })

test('Update HTML test invalid', () => { })

4. friendSearch.js
a. addFriend (fNum), this method involves clicking from a series of buttons associated with potential friends in the database. There is no invalid input, just clicking a button to execute the function or a misclick. I am assuming that because the back-end has been unit tested, they are valid friend1, friend2, etc. entries.
i. Equivalence: The different equivalence classes would be the various potential friends that could be added (first one on the list, nth one on the list) to make sure that they can all be added depending on the value of n. It also checks that the ajax call is made with the expected values.
ii. Boundary: addFriend(fNum = 1) is the smallest possible input, from there fNum could be as large as the number of names in the database.
iii. Test methods:

it('Friend 1 add test', () => { })

it('Friend 5 add test', () => { })

5. view.js

a. addWishList ()
    i. Equivalence: This method involves clicking a button to add a movie to the wish list. The user does not input a string, it gets the string from the valid list of movies provided from the IMDb API. However, I still accounted for the equivalence case of a valid movie string name and an invalid blank string. The ajax call is tested here.
    ii. Boundary: N/A (click/not clicked)
    iii. Test Methods: it('Add wish list valid', () => { })

    it('Add wish list invalid, no movie', () => { })

b. addSession ()
    i. Equivalence: Leaving the Zoom link field blank (or some other invalid input) or one of the other fields blank. The same process should still execute, but these are the varied options depending on user input. The ajax call is also tested.
    ii. Boundary: Lower boundary is trying to create session with no link/movie.
    iii. Test methods: it('Add sessions valid', () => { })

    it('Add sessions blank movie')

    it('Add sessions blank link')

---

part 3:

Measure the ***branch coverage*** achieved by your automated test suite. This requires using a coverage tool appropriate for your programming language and platform. Branch coverage should strive to achieve 100%, but may not reach 100%. Add more test cases until you reach at least 90%. Each additional test case should try to force a particular branch that was not previously covered. Tell us what branch coverage you finally did achieve. If the coverage tool reports less than 100% (that is, between 90% and 99%), discuss one example of a branch that your test cases did not cover and

explain why it is difficult to test this branch. If you were unable to reach 90%, explain why not.

Include the link to the folder in your repository that contains the coverage test reports. Note this means you need to configure your coverage tool to produce reports that can be saved as files in your repository.

Answer:

Coverage report is found here:
https://github.com/AhmedAlzubairi1/GUO/tree/main/htmlcov

Open up the index.html file to navigate through it.

Things to Note:

- Backend coverage:
    - App.py wasn't tested rigorously because all of the logic of app.py is inside model.py. App.py is just calling stuff from model.py. The whole point of model.py is to host all the logic code of app.py to form a MVC flow. Thus even though app.py has a low coverage, it is in actuality a high coverage because model.py is at 100%.
    - Db.py wasn't tested because it is just getters/setters for database.
    - Thus, if you factor in the reasons for db.py & app.py not being explicitly covered, the actual coverage is actually around **100%.**

Coverage for the front-end Javascript can be seen (link below) at index.html for Flask_Project/static. Or, you can run **npm run test** to see the coverage in the terminal.

Branch coverage was **100%** for each of the JS files.

For sessions.js, while I still got to 100% branch coverage, I was not able to cover the specific success and error statements within the ajax call-which is why "Statements" is 94% for that file. I was sure to write a test for the function that occured upon success (updateHTML(data)), and can guarantee that it is being called, because of the new generated list of movies on click and the verification of the rest of statements in ajax. However, my line of "success: expect.any(Function)" is not specific enough to the function that occurs, and I could not achieve 100% line coverage in this instance.

Coverage report for JS unit tests:

https://github.com/AhmedAlzubairi1/GUO/tree/main/coverage/lcov-report

---

part 4:

As part of the release effort, you need to institute ***continuous integration*** for your codebase. The idea is to integrate automated build and test with your version control repository using Travis CI or a similar tool, so that build and test is automatically initiated whenever there is a new commit to the main branch of your github repository. Make sure to ask for help far in advance of the assignment deadline if you have trouble getting CI to work.

Include links to the files in your repository that configure the CI tool(s) and a folder in your repository that includes the CI reports. Note this means you need to configure your CI tool to produce reports that can be saved as files in your repository.

Answer:

We used Travis CI for our continuous integration.
Since we used Travis CI for CI, we don't have our CI reports in a folder, they are hosted through travis CI. Please click the building badge on our github repo to be redirected to the continuous integration report.
Our travis CI config file is located here:
https://github.com/AhmedAlzubairi1/GUO/blob/main/.travis.yml

Note: It is split into the frontend js CI and the backend Python CI, just click the one you want to check and it will tell you the status.