What functionalities were implemented?

Movies part:

1) Search for movies.
2) Get details about a specific movie.

Users part:

1) Sign up.
2) Log in.
3) Log in as a guest.
4) Get me profile.
5) Log out.
6) Add a movie to my list.
7) Remove a movie from my list.
8) Get my list.
9) Get recommendation movies.

What were added that were not planned?

- ✓ We wrote a recommendation function based on the user rating to each movie and the genres of the movies on the user’s list.
- ✓ Log in as a guest.

What functionalities were removed?

- ✗ Nothing.

Models

This section is especially important because it manages all the data logic for our project. This folder contains four files that are the backbone of the project, and those files are:

1- auth file: This file contains a function that acts as a middleware. It authenticates the user to allow him to do some tasks that need authentication.

How it works: This function accepts a token from the user and then take that token and decode it and by using the decoded token, I can get the ID of the user because I made the token using the ID of the user when signing up or logging in. Therefore, we can get everything related to the user by his ID.

2- TMDBApi file: This file contains a class called TMDBApi and that class has some methods that allow us to communicate directly with the TMDB API.

Methods:

- **discoverForRecommendations:** This method will help us discover movies by genres and return a list of movies based on user's list.(more details later)
- **SearchMovies:** This method will help us search for movies by taking a movie name and it will return all the movies that has same name.
- **GetPopular:** This method will get all the popular movies in page 1 in the TMDB API.
- **MovieDetails:** This method gets the primary information about a specific movie.

3- filteringHelper file: This file contains a class called filteringHelper and that class has some methods that do all the needed filtering for the objects we receive from the TMDB API to make them suitable for our needs.

Methods:

- **filterObj:** This method is being called by every method in this class because it filters a movie object based on specific filtering criteria. It takes two arguments the first one is the object that we need to filter, and the second argument is the filtering criteria object that tells the function how and what to filter.
- **FormatMovies:** This method filters every movie object in the movies list based on specific filtering criteria. It takes a list of movies objects, and it filters every object in the list to match our needs and then return the list with the filtered movies objects.
- **FormatMovieToSave:** This method filters a specific movie object to be suitable for saving in the Movies collection based on specific filtering criteria.
- **InjectWatchedToMovies:** This method takes two arguments. The first one is the user's movies list and the second one is the list of searching results from TMDB API after formatting it. This method will go through each movie in both lists and check if each movie is in user's list and if yes, it will add (watched and added) to the object. This is helpful because if I have for example Titanic 1 in my list and I search for Titanic movies

in the home page. I want to know from all the results that Titanic 1 is added to my list and also watched or not.

- **FormatMovie:** This method filters the primary information object about a movie based on specific filtering criteria.

3- userInteractionsLogic file: This file contains a class called userInteractionsLogic and that class has some methods that do all the logic part and data manipulation that relates to all endpoints. (More details will be explained about each endpoint with its method and what each does in the next section).

Routes

Base path of the API is: /myFlex/api/v1/

This folder contains all the project endpoints. It has two files:

1- movies: This file contains all the endpoints related to movies as shown in the table:

HTTP method	**Resource path added to the base**	**Endpoint explanation**	**user needs authentication?**	**Communicates with what Controller file**	**Communicates with what Controller function**
GET	**/search/movie**	Search for movies	YES	MovieController	searchForMovie
GET	**/movie**	Get details about a specific movie.	NO	MovieController	getDetailsAboutSpecificMovie

2- user: This file contains all the endpoints related to users as shown in the table:

HTTP method	**Resource path added to the base**	**Endpoint explanation**	**user needs authentication?**	**Communicates with what Controller file**	**Communicates with what Controller function**
POST	**/signup**	Create a new user - Sign up	NO	UserController	createNewUser
POST	**/login/guest**	Login as Guest	NO	UserController	loginAsGuest
POST	**/login**	Login the user	NO	UserController	login
GET	**/user**	Get my profile	YES	UserController	getMyProfile
POST	**/logout**	Log the user out	YES	UserController	logout
PATCH	**/user/list**	Add a movie to your list	YES	UserController	addMovieToYourList
GET	**/user/list**	Get your movies list	YES	UserController	getMyMoviesList
DELETE	**/user/list**	Remove a movie from your list	YES	UserController	removeMovieFromYourList
GET	**/user/recommendations**	Get recommendations	YES	UserController	getRecomendations

Controllers

This folder contains two files, **MovieController** and **UserController**. They both have functions that are being called by all the endpoints as I mentioned in the tables above. Those functions handle all requests flow, and all those functions communicate with **userInteractionsLogic** file in the model folder that do all the logic part and data manipulation.

UserController file functions:

UserController file functions	**What it does?**	**What it does?**	**UserInteractionsLogic file functions**
createNewUser	1) Receives a request from a user with a body that contains all needed sign-up information (username – email – password). 2) call createNewUser in UserInteractionsLogic and give it that body. 3) hash the password of the user. 4) generating a token for the user. 5) send back a response that contains (all user information – token – success message). 6) error handling.	1) checks if the entered username and email is in the database or not. If not, it creates new user in the DB and return it.	createNewUser
loginAsGuest	1) call loginAsGuest. in UserInteractionsLogic. 2) hash the password of the guest user. 3) generating a token for the guest user. 4) send back a response that contains (all user information – token). 5) error handling.	1) generating automatically all the needed information for the Guest user such as (random id – username – email – password). 2) creates new user with this information in the DB and return it.	loginAsGuest
login	6) call login in UserInteractionsLogic and give it that body. 7) send back a response that contains (all user information – token – success message). 8) error handling.	1) figuring out how the user wants to log in (email or username). 2) get the user from the database. 3) generating a token for the user. 4) return the user and the token.	login
GetMyProfile	1) receives the user profile from the authentication middleware.		No function for getMyProfile.

Refer to what auth function does at the top of the document. Because this and all the following functions will use it as a middleware.	2) send back a response that contains (all user information – success message).		
logout	1) receives the user profile from the authentication middleware. 2) call logout in UserInteractionsLogic and give it the user and the request body. 3) send back a response that contains (all user information – success message). 4) error handling.	1) looping through the user's tokens list and remove the token it receives in the request body. 2) save the updates in the database.	logout
addMovieToYourList	1) receives the user profile from the authentication middleware. 2) call addMovieToYourList in UserInteractionsLogic and give it the user and the request body. 3) send back a response that contains (success message). 4) error handling.	1) doing filtering to the request body. 2) if the user has that movie, overwrite it to be updated if there is any changes. 3) Otherwise, check if the Movies collection doesn't has that movie, add it there and then add it in my list. Otherwise, just add it in my list. 4) save any changes.	addMovieToYourList
getMyMoviesList	1) receives the user profile from the authentication middleware. 2) call getMyMoviesList in UserInteractionsLogic and give it the user. 3) send back a response that contains (user's movies list, success message). 4) error handling.	1) loop through the user's movies list and get all the movies Ids and save them into a list. 2) giving that Ids list to the movies collection in the DB to get another list of movies with all details. 3) doing some filtering to the last it receives from the movies collection and then return it.	getMyMoviesList
removeMovieFromYourList	1) receives the user profile from the authentication middleware. 2) call removeMovieFromYourList in UserInteractionsLogic and give it the user and the request body.	1) getting the movie id that the user wants to delete from the request body. 2) searching for the index of that movie in user's movies list and remove it. 3) save any changes.	removeMovieFromYourList

	3) send back a response that contains (success message). 4) error handling.		
getRecomendations	1) receives the user profile from the authentication middleware. 2) call getRecomendations in UserInteractionsLogic and give it the user and the request body. 3) send back a response that contains (recommendation or popular list). 4) error handling.	1) if my list of movies is empty, this function will get some popular movies from the TMDB API and return it. 2) otherwise, create an empty genre preferences list. Also, every movie has a user rating. ➢ Loop through the genre list of each user’s movie. If genres Pref list contains this genre, add the user rating of that movie to this genre and if not just put the genre equals the user rating. ➢ Then get the genre that has the maximum and minimum rating and get the range. ➢ Create two empty lists called with_genres and without_genres. ➢ Finally, loop through the genre preferences list and if the rating of each genre is > range/2, add it to with_genres list. Otherwise, add it to without _genres list. ➢ call discoverForRecommendations function in the TMDBApi file and giving it these with_genres and without_genres. ➢ do some filtering on the response it gets and then return the recommendation list.	getRecomendations

MovieController file functions: this file has two functions:

1- searchForMovie: this function Search for a movie and return a list of movies with the same name.

2- getDetailsAboutSpecificMovie: this function gets all the details about specific movie.

Data Model collections: We designed our database which it will have two collections:

Collection name	Schema	Schema fields description
Movie	id	The id of the movie.
	title	The name of the movie.
	overview	The description of the movie.
	poster_path	The link we will get the movie's poster from.
	vote_average	The IMDb rating.
	genres	The list of genres of a movie.
User	username	The username of the user.
	Email (validation)	The email of the user.
	password	The password of the user.
	tokens	The list of tokens that relates to the user's profile.
	movies	The list of movies.

- ✓ We made our data model normalized. We are saving our movies with all needed information in the movies collection, and we just save ids in user's movies list with some additions relates to the user. Now if a user wants his list of movies, we can easily pass the list of movies ids and MongoDB will retrieve all the movies from the movies collection like we did in getMyMoviesList function. Also, we made it that way because TMDB API does not allow us to send a list of movies ids and get all the movies with all the information in a one HTTP request but instead we need to loop through each movie id and make an HTTP request and that is so slow overtime and not efficient.

Tests

first of all, I tested the user schema with two successes cases:

1- Test creation of a valid user with parameters matching. (S)
2- Test that the DB creates an empty movies and tokens list. (S)

secondly, I tested the API calls:

Test sign up endpoint with 1 success and 3 fails:

1- Test passing valid user (signing up first time). (S)
2- Test sign up with invalid email. (F)
3- Test sign up with an email that exists in the database. (F)
4- Test sign up with an username that exists in the database. (F)

Test log in endpoint with 2 success and 3 fails:

1- Test log in with incorrect email. (F)
2- Test log in with wrong username. (F)
3- Test log in with wrong password. (F)
4- Test log in with correct email and password. (S)
5- Test log in with correct username and password. (S)

Test get your profile endpoint with 1 success and 1 fail:

1- Test passing wrong token in the object. (F)
2- Test passing correct token in the object. (S)

Test Add a movie to your list with 2 success and 1 fail:

1- Test adding nothing in the correct user's list. (F)
2- Test adding valid movie id in the correct user's list. (S)
3- Test editing valid movie id existing in the correct user's list. (S)

Test Get my movies list with 1 success:

1- Test passing correct token and get user's movies list. (S)

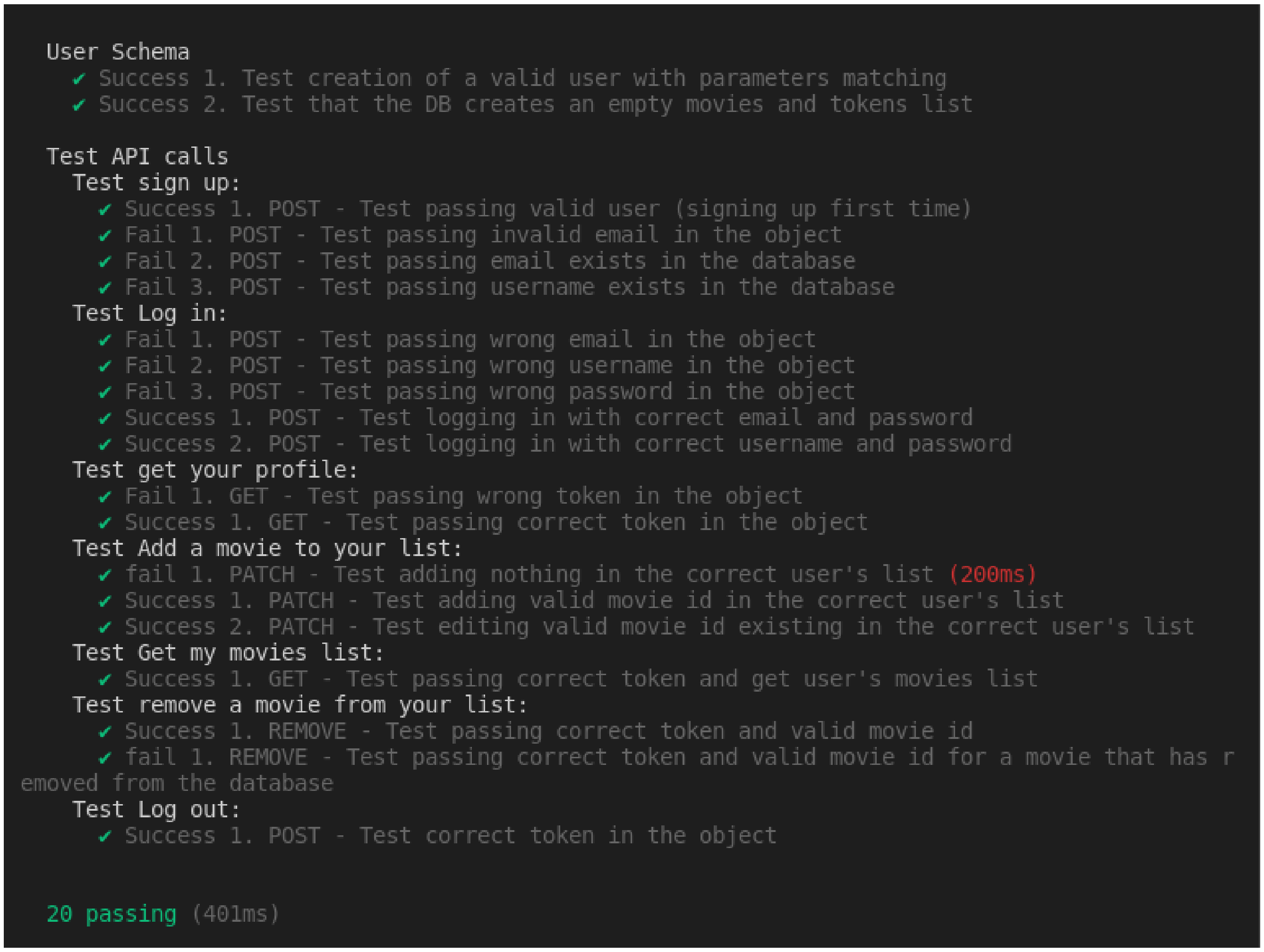
Test remove a movie from your list with 1 success and 1 fail:

1- Test passing correct token and valid movie id. (S)
2- Test passing correct token and valid movie id for a movie that has removed from the database. (F)

Test Log out with 1 success:

1- Test correct token in the object. (S)

From my point of view these are the most essential functions in my project. This a screenshot of the first test:

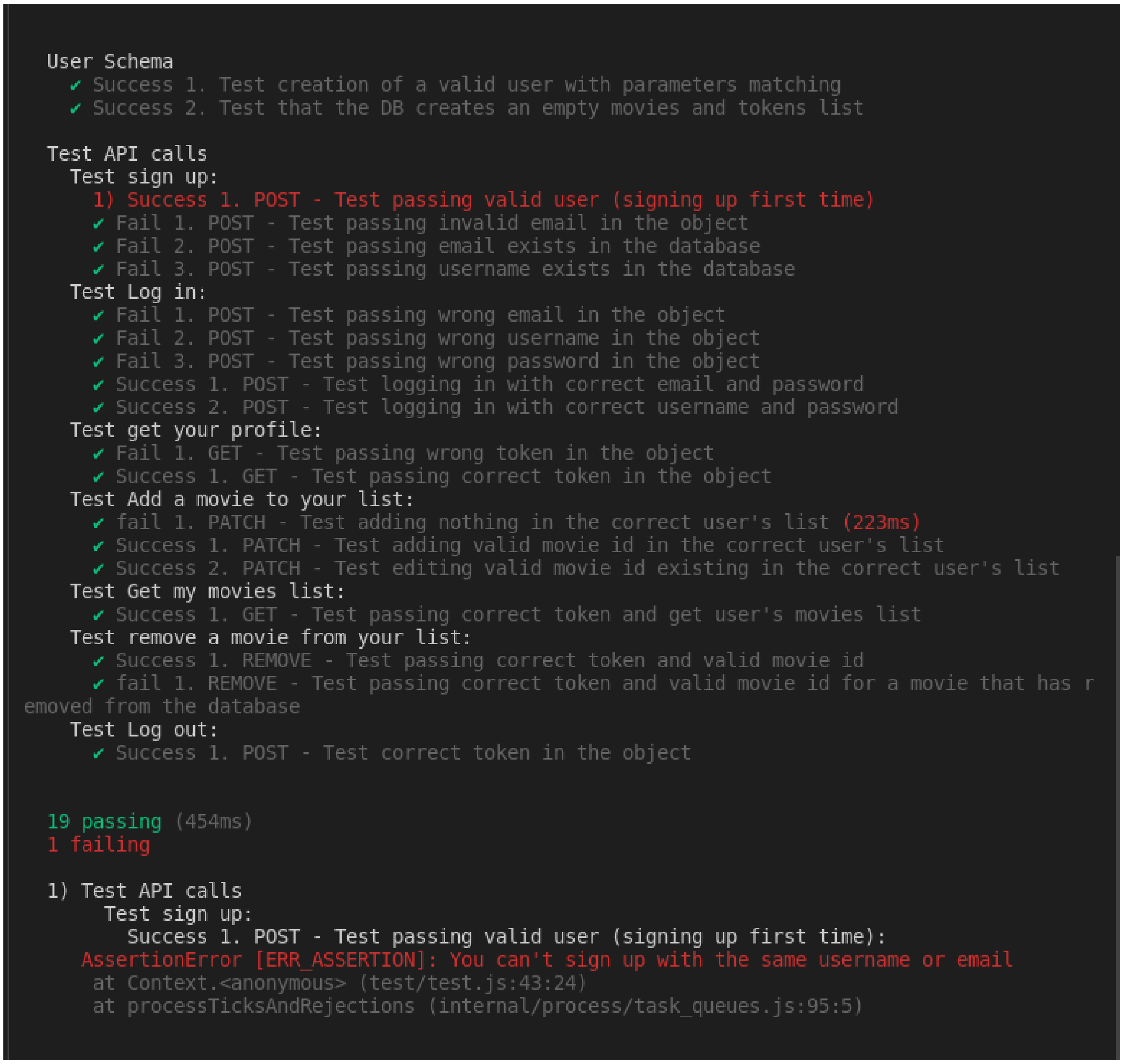


```
  User Schema
    ✔ Success 1. Test creation of a valid user with parameters matching
    ✔ Success 2. Test that the DB creates an empty movies and tokens list

  Test API calls
    Test sign up:
      ✔ Success 1. POST - Test passing valid user (signing up first time)
      ✔ Fail 1. POST - Test passing invalid email in the object
      ✔ Fail 2. POST - Test passing email exists in the database
      ✔ Fail 3. POST - Test passing username exists in the database
    Test Log in:
      ✔ Fail 1. POST - Test passing wrong email in the object
      ✔ Fail 2. POST - Test passing wrong username in the object
      ✔ Fail 3. POST - Test passing wrong password in the object
      ✔ Success 1. POST - Test logging in with correct email and password
      ✔ Success 2. POST - Test logging in with correct username and password
    Test get your profile:
      ✔ Fail 1. GET - Test passing wrong token in the object
      ✔ Success 1. GET - Test passing correct token in the object
    Test Add a movie to your list:
      ✔ fail 1. PATCH - Test adding nothing in the correct user's list (200ms)
      ✔ Success 1. PATCH - Test adding valid movie id in the correct user's list
      ✔ Success 2. PATCH - Test editing valid movie id existing in the correct user's list
    Test Get my movies list:
      ✔ Success 1. GET - Test passing correct token and get user's movies list
    Test remove a movie from your list:
      ✔ Success 1. REMOVE - Test passing correct token and valid movie id
      ✔ fail 1. REMOVE - Test passing correct token and valid movie id for a movie that has r
emoved from the database
    Test Log out:
      ✔ Success 1. POST - Test correct token in the object


  20 passing (401ms)
```

For the second time the first test for signing up failed because the user can't sign up with the same email or user. So, I did fail assertion.



```
  User Schema
    ✔ Success 1. Test creation of a valid user with parameters matching
    ✔ Success 2. Test that the DB creates an empty movies and tokens list

  Test API calls
    Test sign up:
      1) Success 1. POST - Test passing valid user (signing up first time)
      ✔ Fail 1. POST - Test passing invalid email in the object
      ✔ Fail 2. POST - Test passing email exists in the database
      ✔ Fail 3. POST - Test passing username exists in the database
    Test Log in:
      ✔ Fail 1. POST - Test passing wrong email in the object
      ✔ Fail 2. POST - Test passing wrong username in the object
      ✔ Fail 3. POST - Test passing wrong password in the object
      ✔ Success 1. POST - Test logging in with correct email and password
      ✔ Success 2. POST - Test logging in with correct username and password
    Test get your profile:
      ✔ Fail 1. GET - Test passing wrong token in the object
      ✔ Success 1. GET - Test passing correct token in the object
    Test Add a movie to your list:
      ✔ fail 1. PATCH - Test adding nothing in the correct user's list (223ms)
      ✔ Success 1. PATCH - Test adding valid movie id in the correct user's list
      ✔ Success 2. PATCH - Test editing valid movie id existing in the correct user's list
    Test Get my movies list:
      ✔ Success 1. GET - Test passing correct token and get user's movies list
    Test remove a movie from your list:
      ✔ Success 1. REMOVE - Test passing correct token and valid movie id
      ✔ fail 1. REMOVE - Test passing correct token and valid movie id for a movie that has r
emoved from the database
    Test Log out:
      ✔ Success 1. POST - Test correct token in the object


  19 passing (454ms)
  1 failing

  1) Test API calls
       Test sign up:
         Success 1. POST - Test passing valid user (signing up first time):
     AssertionError [ERR_ASSERTION]: You can't sign up with the same username or email
      at Context.<anonymous> (test/test.js:43:24)
      at processTicksAndRejections (internal/process/task_queues.js:95:5)
```