StreamInsights Analyser Applicaton

IIHT

Time To Complete: 3 hrs

Contents

1 Problem Statement 3
2 Business Requirements: 3
3 Implementation/Functional Requirements 3
3.1 Code Quality/Optimizations 3
3.2 Template Code Structure 4
a. Package: com.streaminsightsanalyserapplication 4
b. Package: com.streaminsightsanalyserapplication.model 4
c. Package: com.streaminsightsanalyserapplication.repository 4
4 Execution Steps to Follow 5

1 Problem Statement

The StreamInsights Analyzer Application provides users with the ability to perform not only basic CRUD (Create, Read, Update, Delete) operations and search functionalities in different criterias on movies, movie reviews, and user profiles but also provides the analytical operations like most watched movies, sorting movies as per their genres, getting lowest rated movie any many more for analysis and viewing.

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. User needs to enter into the application. 2. The user should be able to do the particular operations 3. The console should display the menu 1) create a new movie 2) create a new user 3) create a movie review 4) update a movie 5) update a user 6) get details of a movie 7) get details of a user 8) delete a movie 9) delete a user 10) get most watched movies 11) sort movies by genre count 12) get top movies watched by users of age between 25 - 30 13) search movies with minimum rating of 4 by a director 14) get lowest rated movie for an actor 15) search movies with anime category and length under 150 minutes 16) exit

Business Requirements

System Features

The application should provide the following functionalities to the user:

- **Create Movie**: Add a new movie to the system with details such as name, genre, director, star cast, length, and certificate.
- **Create User**: Add a new user to the system with attributes like name, age, and gender.
- **Create Movie Review**: Add a review for a movie by a user, including rating and review text.
- **Update Movie**: Modify the details of an existing movie.
- **Update User**: Modify the details of an existing user.
- **Get Movie by ID**: Retrieve and display details of a movie by its unique identifier.
- **Get User by ID**: Retrieve and display details of a user by their unique identifier.
- **Delete Movie**: Remove a movie from the system by its ID.
- **Delete User**: Remove a user from the system by their ID.
- **Get Most Watched Movies**: Display a list of movies that have been watched the most.
- **Sort Movies by Genre Count**: Display movies sorted by how many times each genre has been represented.
- **Get Top Movies by User Age Range**: Retrieve top-rated movies watched by users within a specified age range.
- **Search Movies by Director and Minimum Rating**: Find movies directed by a specific director that have a minimum user rating.
- **Get Lowest Rated Movie by Actor**: Find the lowest-rated movie in which a given actor has performed.
- **Search Movies by Genre and Length**: Find movies of a specific genre that are under a certain length in minutes.
- **Exit Application**: Terminate the application.

Classes and Method Descriptions

StreamInsightsAnalyserApplication Class

- Entry point for the application, handling the execution flow.

- Manages application flow including database setup, DAO initialization, and user interactions through a console menu.

- Acts as the controller coordinating input, processing, and output using DAO layers for Movies, Users, and Reviews.

- You can define all other methods present in this class as:

showOptions()

- **Task**: Prints the menu (as mentioned above in business requirement section) for the user to choose from.

- **Functionality**: This method prints a list of options for the user to choose from, including operations like creating, updating, deleting movies, users, and movie reviews as mentioned in business requirements above.

- **Return Value**: void

- **Explanation**: The menu options are printed to the console to help the user navigate through the system.

addMovie()

- **Task**: Creates a new movie in the system.

- **Functionality**: This method prompts the user for the movie's details such as name, genre, director, star cast, length, and certificate, then creates a `Movie` object and calls `movieDAO.addMovie()` to save it in the database.

- **Return Value**: void

- **Explanation**: If the movie is created successfully, a message is displayed as "Movie added successfully.".

addUser()

- **Task**: Creates a new user in the system.

- **Functionality**: This method prompts the user for the user's name, age, and gender, then creates a `User` object and calls `userDAO.addUser()` to save it in the database.

- **Return Value**: void

- **Explanation**: If the user is created successfully, a message is displayed as "User added successfully.".

addMovieReview()
- **Task**: Adds a movie review to the system.

- **Functionality**: This method prompts the user for the movie ID, user ID, review content, and rating, then creates a `MovieReview` object and calls `movieReviewDAO.addMovieReview()` to save it in the database.

- **Return Value**: void

- **Explanation**: If the movie review is added successfully, a message is displayed as "Movie review added successfully.".

updateMovie()
- **Task**: Updates the details of an existing movie.

- **Functionality**: This method prompts the user to enter the movie ID, retrieves the `Movie` object using `movieDAO.getMovieById()`, updates its fields with new input, and calls `movieDAO.updateMovie()` to save the changes.

- **Return Value**: void

- **Explanation**: If the movie exists, it is updated with the new information; if not, a message is displayed as "Movie not found.".

updateUser()
- **Task**: Updates the details of an existing user.

- **Functionality**: This method prompts the user to enter the user ID, retrieves the `User` object using `userDAO.getUserById()`, updates its fields with new input, and calls `userDAO.updateUser()` to save the changes.

- **Return Value**: void

- **Explanation**: If the user exists, it is updated with the new information; if not, a message is displayed as "User not found.".

getMovieDetails()
- **Task**: Retrieves a movie by its unique ID.

- **Functionality**: This method prompts the user to enter the movie ID, retrieves the `Movie` object using `movieDAO.getMovieById()`, and displays its details.

- **Return Value**: void

- **Explanation**: If the movie is found, its details are displayed; if not, an error message is shown as "Movie not found.".

getUserDetails()

- **Task**: Retrieves a user by their unique ID.

- **Functionality**: This method prompts the user to enter the user ID, retrieves the `User` object using `userDAO.getUserById()`, and displays its details.

- **Return Value**: void

- **Explanation**: If the user is found, their details are displayed; if not, an error message is shown as "User not found.".

deleteMovie()

- **Task**: Deletes a movie from the system.

- **Functionality**: This method prompts the user to enter the movie ID and calls `movieDAO.deleteMovie()` to remove the movie from the database.

- **Return Value**: void

- **Explanation**: After deletion, a success message is displayed as "Movie deleted successfully.".

deleteUser()

- **Task**: Deletes a user from the system.

- **Functionality**: This method prompts the user to enter the user ID and calls `userDAO.deleteUser()` to remove the user from the database.

- **Return Value**: void

- **Explanation**: After deletion, a success message is displayed as "User deleted successfully.".

getMostWatchedMovies()

- **Task**: Retrieves and displays the most-watched movies from the system.
    - **Functionality**: This method calls `movieDAO.getMostWatchedMovies()` to fetch a list of the most-watched movies from the database. If the list is not empty, it prints each movie's details to the console. If no movies are found, a message is displayed stating "No movies found."

- **Return Value**: void
- **Explanation**: This method helps in displaying a list of popular movies based on the watch frequency. If no such movies are available, it will notify the user.

sortMoviesByGenreCount()

- **Task**: Sorts and displays movies based on the count of genres.
  - **Functionality**: This method calls `movieDAO.sortMoviesByGenreCount()` to fetch a list of movies sorted by the number of times each genre appears in the database. If the list is not empty, it prints each movie's details to the console in the sorted order. If no movies are found, a message is displayed stating "No movies found."
  - **Return Value**: void
  - **Explanation**: This method helps in displaying the movies in a sorted order based on the frequency of their genres, providing insight into the most popular genres in the system. If no movies are available, it notifies the user.

getTopMoviesByAgeRange()

- **Task**: Retrieves and displays the top movies for a specified age range.
  - **Functionality**: This method prompts the user to input the minimum age, maximum age, and a limit for the number of movies to retrieve. It then calls `movieDAO.getTopMoviesByAgeRange(minAge, maxAge, limit)` to fetch a list of top movies within the specified age range, limited to the specified number. If the list is not empty, it prints each movie's details. If no movies are found, a message is displayed stating "No movies found."
  - **Return Value**: void
  - **Explanation**: This method allows the user to find the most popular movies based on age range criteria, helping to discover content tailored to a specific demographic. If no movies are found for the specified criteria, the user is notified.

searchMoviesByDirectorAndRating()

- **Task**: Retrieves and displays movies based on the director's name and minimum rating.
  - **Functionality**: This method prompts the user to enter a director's name and a minimum rating. It then calls `movieDAO.searchMoviesByDirectorAndRating(director, rating)` to fetch a list of movies directed by the specified director with a rating

greater than or equal to the given minimum rating. If the list is not empty, it prints each movie's details. If no movies are found, a message is displayed stating "No movies found."

- **Return Value**: void

- **Explanation**: This method helps users find movies from a specific director that meet a certain rating threshold, making it easy to discover highly-rated works by particular filmmakers. If no matching movies are found, the user is notified.

getLowestRatedMovieByActor()

- **Task**: Retrieves and displays the lowest-rated movie for a specific actor.

  - **Functionality**: This method prompts the user to enter an actor's name and then calls `movieDAO.getLowestRatedMovieByActor(actor)` to fetch the movie with the lowest rating that the specified actor has starred in. If a movie is found, it prints the movie's details. If no movie is found, a message is displayed stating "No movie found."

  - **Return Value**: void

  - **Explanation**: This method allows users to find the lowest-rated movie for a specific actor, providing insight into their less successful roles. If no movie is found, the user is notified.

searchMoviesByGenreAndLength()

- **Task**: Retrieves and displays movies based on a specific genre and maximum length.

  - **Functionality**: This method prompts the user to enter a genre and the maximum length of the movie. It then calls `movieDAO.searchMoviesByGenreAndLength(genre, maxLength)` to fetch a list of movies that match the specified genre and have a length less than or equal to the given maximum length. If the list is not empty, it prints each movie's details. If no movies are found, a message is displayed stating "No movies found."

  - **Return Value**: void

  - **Explanation**: This method helps users find movies that match a specific genre and fit within a certain length, allowing for more tailored recommendations. If no matching movies are found, the user is notified.

Entity Class Field Annotations

Movie Class

Make sure to use proper annotation to make this class as Entity and have the table name as “movie”.

Field Annotations for Movie

1. **`id` field (Primary Key)**:
 - The **`id`** field is expected to be the primary key of the `movie` table.
 - **Auto-Generated**: The ID is auto-generated by the database using the **`GenerationType.IDENTITY`** strategy.

2. **`name` field**:
 - **Not Null**: While not explicitly annotated with `@NotNull`, it is expected to be non-null for valid movie records.
 - **Database Column**: Mapped to the column `name` in the `movie` table.

3. **`genre` field**:
 - **Not Null**: Expected to be provided by the user and should not be null.
 - **Database Column**: Mapped to the column `genre`.

4. **`director` field**:
 - **Not Null**: The director's name should be provided and not be null.
 - **Database Column**: Mapped to the column `director`.

5. **`starCast` field**:
 - **Not Null**: This field is expected to contain the lead actors’ names and should not be null.
 - **Database Column**: Mapped to the column `star_cast`.

6. **`length` field**:
 - **Not Null**: Should always have a valid integer value representing the movie’s duration in minutes.
 - **Database Column**: Mapped to the column `length`.

7. **`certificate` field**:
 - **Optional**: Not enforced as required in code, but usually holds information like "U", "PG", "A", etc.
 - **Database Column**: Mapped to the column `certificate`.

MovieReview Class

Make sure to use proper annotation to make this class as Entity and have the table name as "movie_review".

Field Annotations for MovieReview

1. **`id` field (Primary Key)**:
 - The **`id`** field is expected to be the primary key of the `movie_review` table.
 - **Auto-Generated**: The ID is auto-generated using the **`GenerationType.IDENTITY`** strategy.

2. **`movieId` field**:
 - **Not Null**: This field must not be null and links the review to a specific movie.
 - **Database Column**: Mapped to the column `movie_id` in the database.

3. **`userId` field**:
 - **Not Null**: This field must not be null and links the review to a specific user.
 - **Database Column**: Mapped to the column `user_id`.

4. **`review` field**:
 - **Optional**: This field contains the text of the review and is not strictly validated for null.
 - **Database Column**: Mapped to the column `review`.

5. **`rating` field**:
 - **Not Null**: Must have a valid integer value representing the rating (e.g., 1–5).
 - **Database Column**: Mapped to the column `rating`.

User Class

Make sure to use proper annotation to make this class as Entity and have the table name as "user".

Field Annotations for User

1. **`id` field (Primary Key)**:
 - The **`id`** field is the primary key for the `user` table.
 - **Auto-Generated**: The ID is auto-generated using the **`GenerationType.IDENTITY`** strategy.

2. **`name` field**:
 - **Not Null**: The user's name must be provided and must not be null.
 - **Database Column**: Mapped to the column `name`.

3. **`age` field**:
 - **Not Null**: The age must be provided as a valid integer and should not be null.

- **Database Column**: Mapped to the column `age`.

4. **`gender` field**:
- **Not Null**: This field indicates the user's gender and must not be null.
- **Database Column**: Mapped to the column `gender`.

DAO Classes and Method Descriptions

MovieDAOImpl Class

addMovie(Movie movie)
- **Task**: Inserts a new movie into the database.

- **Functionality**: Executes an SQL `INSERT` query to add the movie's data into the `movie` table.

- **Return Value**: void

- **Explanation**: This method inserts the provided `Movie` object into the database using a prepared statement.

updateMovie(Movie movie)
- **Task**: Updates an existing movie's details in the database.

- **Functionality**: Executes an SQL `UPDATE` query using the movie ID to modify the corresponding record in the database.

- **Return Value**: void

- **Explanation**: This method updates the fields of the existing movie in the database.

getMovieById(int id)
- **Task**: Retrieves a movie by its ID.

- **Functionality**: Executes a `SELECT` query using the given movie ID to fetch its details.

- **Return Value**: `Movie` object representing the movie.

- **Explanation**: This method fetches a movie record by its ID from the database and maps it to a `Movie` object.

deleteMovie(int id)
- **Task**: Deletes a movie by its ID.

- **Functionality**: Executes an SQL `DELETE` query to remove the movie from the database based on its ID.

- **Return Value**: void

- **Explanation**: This method deletes the movie record from the database for the given ID.

getMostWatchedMovies()
- **Task**: Retrieves the top 10 most watched movies.

  - **Functionality**: Executes a `JOIN` query on `movie` and `movie_review` tables and groups by movie ID, then orders by watch count.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method returns the most frequently reviewed (watched) movies in descending order of review count.

sortMoviesByGenreCount()
- **Task**: Sorts movies by their genre count.

  - **Functionality**: Executes a `GROUP BY` and `ORDER BY` query on the `movie` table to count genre occurrences.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method returns movies sorted by how often each genre appears.

getTopMoviesByAgeRange(int minAge, int maxAge, int limit)
- **Task**: Retrieves top movies watched by users within a specific age range.

  - **Functionality**: Executes a `JOIN` query between `movie`, `movie_review`, and `user` tables filtering users by age, and limits the result.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method filters reviews by user age and returns the top movies watched in that age group.

searchMoviesByDirectorAndRating(String director, int rating)
- **Task**: Searches for movies by a specific director with a minimum rating.

  - **Functionality**: Executes a `JOIN` query between `movie` and `movie_review` filtering by director name and minimum rating.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method returns movies directed by the specified director that have a rating equal to or greater than the provided value.

getLowestRatedMovieByActor(String actor)
- **Task**: Retrieves the lowest rated movie in which the specified actor appears.

  - **Functionality**: Executes a query joining `movie` and `movie_review` filtering by actor name in `star_cast` and ordering by rating ascending.

  - **Return Value**: `Movie` object.

  - **Explanation**: This method returns the movie with the lowest rating among those that include the given actor in the cast.

searchMoviesByGenreAndLength(String genre, int maxLength)
- **Task**: Searches for movies of a specific genre with length under a given limit.

  - **Functionality**: Executes a `SELECT` query filtering by genre and movie length.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method fetches movies that match the specified genre and do not exceed the given duration.

deleteAllMovies()
- **Task**: Deletes all movies from the database.

  - **Functionality**: Executes a SQL `DELETE` query to remove all records from the `movie` table.

  - **Return Value**: void

  - **Explanation**: This method clears all movie records from the database.

getAllMovies()
- **Task**: Retrieves all movies from the database.

  - **Functionality**: Executes a `SELECT *` query on the `movie` table to fetch all movie records.

  - **Return Value**: List of `Movie` objects.

  - **Explanation**: This method retrieves all existing movies from the database.

MovieReviewDAOImpl Class

addMovieReview(MovieReview movieReview)
- **Task**: Inserts a new movie review into the database.

  - **Functionality**: Executes an SQL `INSERT` query to add the review data into the `movie_review` table.

  - **Return Value**: void

  - **Explanation**: This method inserts the provided `MovieReview` object into the database.

updateMovieReview(MovieReview movieReview)
- **Task**: Updates an existing movie review in the database.

  - **Functionality**: Executes an SQL `UPDATE` query using the review ID to modify the corresponding record.

  - **Return Value**: void

  - **Explanation**: This method updates the details of an existing movie review using the provided object.

getMovieReviewById(int id)
- **Task**: Retrieves a movie review by its ID.

  - **Functionality**: Executes a `SELECT` query using the provided review ID to fetch its details.

  - **Return Value**: `MovieReview` object representing the review.

  - **Explanation**: This method fetches a movie review by ID and maps the result to a `MovieReview` object.

deleteMovieReview(int id)
- **Task**: Deletes a movie review from the database.

  - **Functionality**: Executes an SQL `DELETE` query to remove the review by its ID.

  - **Return Value**: void

  - **Explanation**: This method deletes the specified movie review from the database.

deleteAllMovieReviews()
- **Task**: Deletes all movie reviews from the database.

  - **Functionality**: Executes an SQL `DELETE` query to remove all records from the `movie_review` table.

  - **Return Value**: void

  - **Explanation**: This method clears the entire `movie_review` table.

getAllMovieReviews()
- **Task**: Retrieves all movie reviews from the database.

  - **Functionality**: Executes a `SELECT *` query on the `movie_review` table to fetch all reviews.

  - **Return Value**: List of `MovieReview` objects.

  - **Explanation**: This method retrieves all movie reviews stored in the database.

UserDAOImpl Class

addUser(User user)
- **Task**: Inserts a new user into the database.

  - **Functionality**: Executes an SQL `INSERT` query to add the user data into the `user` table.

  - **Return Value**: void

  - **Explanation**: This method inserts the provided `User` object into the database.

updateUser(User user)
- **Task**: Updates an existing user in the database.

- **Functionality**: Executes an SQL `UPDATE` query using the user ID to modify the corresponding record.

- **Return Value**: void

- **Explanation**: This method updates the details of an existing user using the provided object.

getUserById(int id)
- **Task**: Retrieves a user by their ID.

- **Functionality**: Executes a `SELECT` query using the provided user ID to fetch their details.

- **Return Value**: `User` object representing the user.

- **Explanation**: This method fetches a user by ID and maps the result to a `User` object.

deleteUser(int id)
- **Task**: Deletes a user from the database.

- **Functionality**: Executes an SQL `DELETE` query to remove the user by their ID.

- **Return Value**: void

- **Explanation**: This method deletes the specified user from the database.

deleteAllUsers()
- **Task**: Deletes all users from the database.

- **Functionality**: Executes an SQL `DELETE` query to remove all records from the `user` table.

- **Return Value**: void

- **Explanation**: This method clears the entire `user` table.

getAllUsers()
- **Task**: Retrieves all users from the database.

- **Functionality**: Executes a `SELECT *` query on the `user` table to fetch all user records.

- **Return Value**: List of `User` objects.

- **Explanation**: This method retrieves all users stored in the database.

3 Implementation/Functional Requirements

3.1 Code Quality/Optimizations

1. Associates should have written clean code that is readable.
2. Associates need to follow SOLID programming principles.

3.2 Template Code Structure

a. Package: com.streaminsightsanalyserapplication

Resources

Class/Interface	Description	Status
StreamInsightsAnalyserApplication.java(class)	This represents bootstrap class i.e class with Main method, that shall contain all console interaction with the user.	Partially implemented

b. Package: com.streaminsightsanalyserapplication.model

Resources

Class/Interface	Description	Status
Movie.java(class)	This represents entity class for Movie	Partially Implemented
MovieReview.java(class)	This represents entity class for MovieReview	Partially Implemented
User.java(class)	This represents entity class for User	Partially Implemented

c. Package: com.streaminsightsanalyserapplication.repository

Resources

Class/Interface	Description	Status
MovieDao.java(interface)	This is an interface containing declaration of DAO method	Already Implemented
MovieDaoImpl.java(class)	This is an implementation class for DAO methods. Contains empty method bodies, where logic needs to written by test taker	Partially Implemented

MovieReviewDao.java(interface)	This is an interface containing declaration of DAO method	Already Implemented
MovieReviewDaoImpl.java (class)	This is an implementation class for DAO methods. Contains empty method bodies, where logic needs to written by test taker	Partially Implemented
UserDao.java(interface)	This is an interface containing declaration of DAO method	Already Implemented
UserDaoImpl.java(class)	This is an implementation class for DAO methods. Contains empty method bodies, where logic needs to written by test taker	Partially Implemented

4 Execution Steps to Follow

1. All actions like build, compile, running application, running test cases will be through Command Terminal.
2. To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.
3. To build your project use command:

   sudo JAVA_HOME=$JAVA_HOME /usr/share/maven/bin/mvn clean package -Dmaven.test.skip

*If it asks for the password, provide password : pass@word1

4. This editor Auto Saves the code.
5. These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

6. Default credentials for MySQL:
   a. Username: root
   b. Password: pass@word1

7. To login to mysql instance: Open new terminal and use following command:
   a. sudo systemctl enable mysql
   b. sudo systemctl start mysql

NOTE: After typing any of the above commands you might encounter any warnings.

>> Please note that this warning is expected and can be disregarded. Proceed to the next step.

c. mysql -u root -p

The last command will ask for password which is 'pass@word1'

8. These are time bound assessments. The timer would stop if you logout (Save & Exit) and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

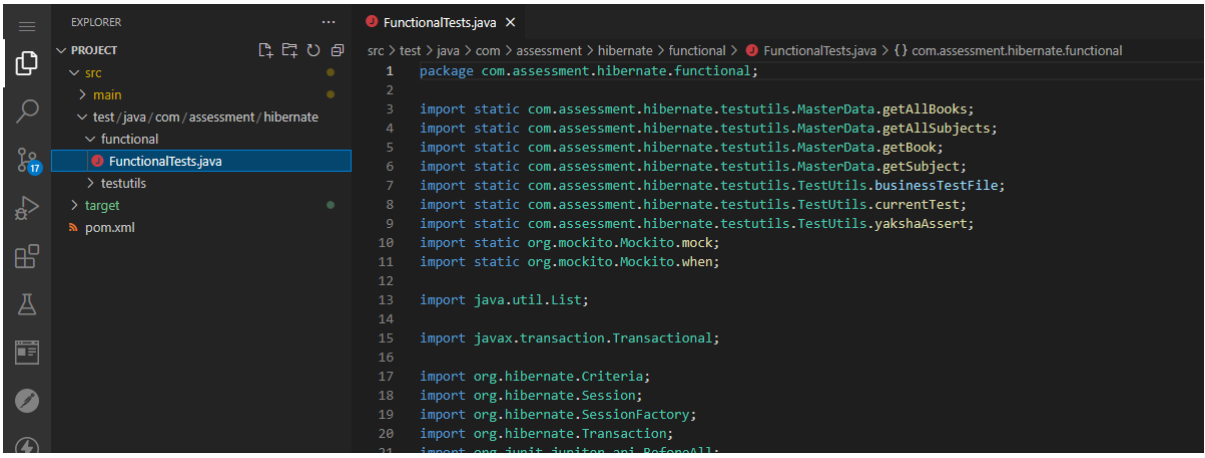
9. To run your project use command:

sudo JAVA_HOME=$JAVA_HOME /usr/share/maven/bin/mvn clean install exec:java -Dexec.mainClass="com.streaminsightsanalyserapplication.StreamInsightsAnalyserApplication"

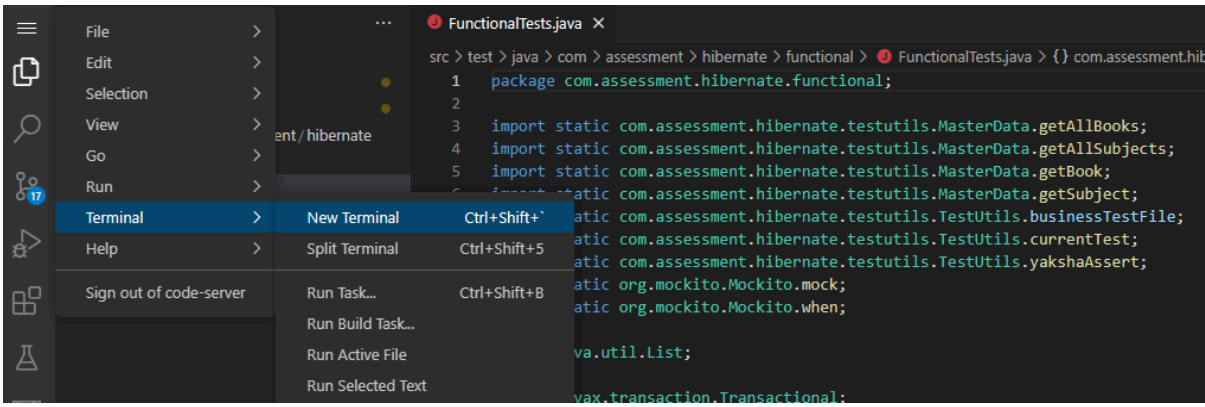
*If it asks for the password, provide password : pass@word1

10. To test your project, use the command

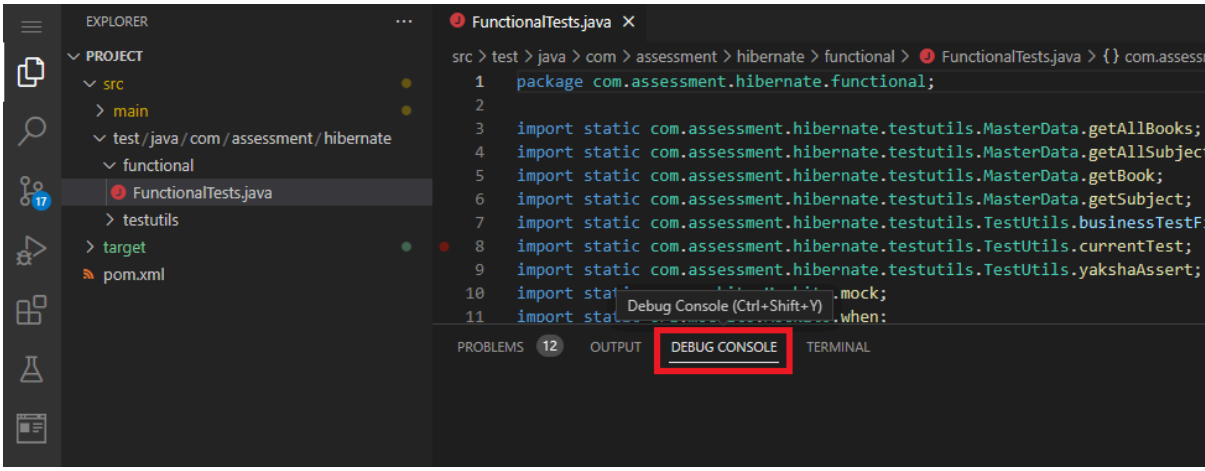
a. Open FunctionalTests.java file in editor



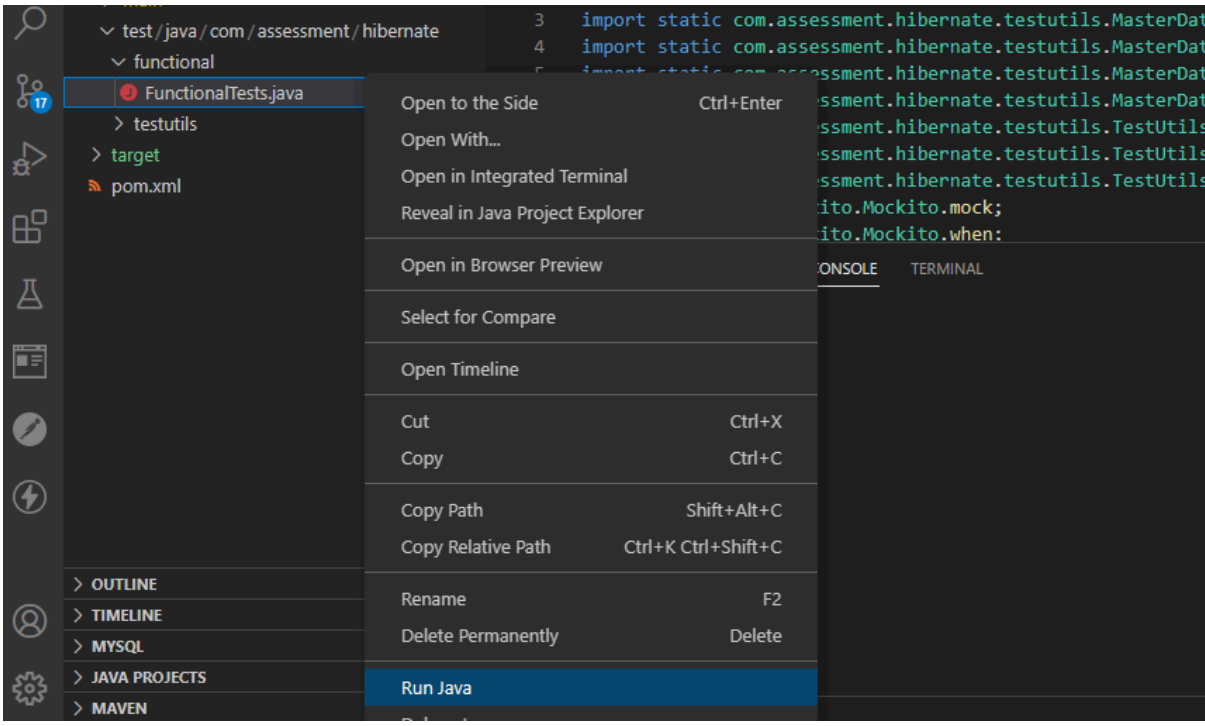
b. Open a new Terminal



c. Go to Debug Console Tab



d. Right click on FunctionalTests.java file and select option Run Java



e. This will launch the test cases and status of the same can be viewed in Debug Console