Movies Watchlist

(SVVT Documentation)

June, 2025.

Introduction

About the Project

Movies Watchlist is a web-based application designed to help users efficiently manage their personal movie watchlists. It allows users to add, edit, filter, and mark movies as "Watched" while leveraging AI-driven genre suggestions and email notifications with personalized movie recommendations. The application emphasizes easy usage, responsiveness, and scalability.

Core Application's Features are:

- Add, edit, and delete movies
- AI-based genre suggestions via OpenAI API
- Mark movies as "Watched" and receive email notifications with similar movie recommendations
- Filtering and sorting by genre, status, watchlist priority, and category
- Manage custom watchlist categories (e.g., "Watch after exams", "During spring break", etc.)
- Enable or disable email notifications
- Responsive user interface for all device types

Application's Technology Stack:

- Frontend: React
- Backend: Java Spring Boot
- Database: MySQL
- External APIs: OpenAI API (AI recommendations), Infobip API (email delivery)

Live Application Link: https://movie-watchlist-fe-live.onrender.com/

Please note: If the deployed application link appears to be unavailable, this is likely due to Render's free-tier service entering sleep mode after a period of inactivity. If you encounter this issue, notify me and I will immediately restart the deployment.

GitHub Repositories:

- Backend Code, Unit and Integration Tests: Movie-Watchlist-BE-Spring-SE-Project
- Frontend: Movies-Watchlist-FE-React-SE-Project
- Selenium UI tests: SVVT Master

Project Functionalities and Screenshots

This section outlines the core functionalities of the Movies Watchlist application, describing how users interact with its interface to manage and organize their movie watchlists effectively. The application emphasizes simplicity, clarity, and responsiveness across devices.

Landing Page

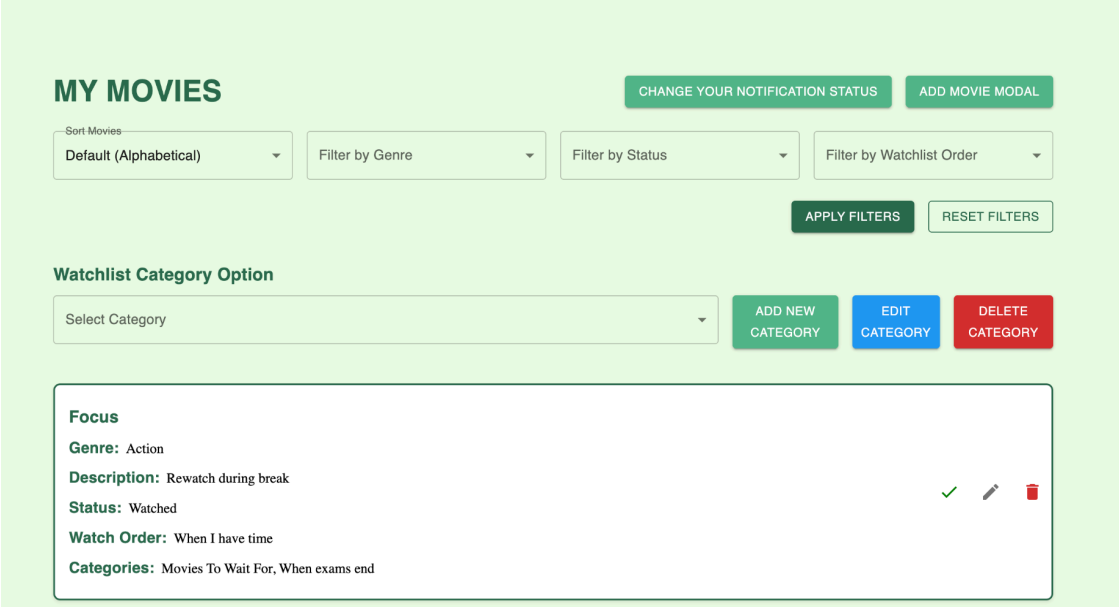
Upon logging in, users are taken to the **My Movies** dashboard — the central hub of the application where all core features are accessible from a single, intuitive interface.

Key elements available on this page include:

- **Sorting dropdown** to organize movies alphabetically or by custom-defined watchlist order.
- **Multiple filters** for narrowing the list by genre, status (e.g., Watched/To Watch), or watchlist order.
- **Watchlist category selector** to display only movies from a specific custom group.
- **Buttons for adding, editing, or deleting categories**, enabling dynamic grouping of movies.
- **Global notification toggle** that allows users to turn email notifications on or off.
- A button to **open the Add Movie modal**, where new entries can be submitted manually or with AI assistance.

This page also displays each movie as a neatly formatted card showing:

- Title, genre, description, and status
- Watchlist order and associated categories
- Action icons for marking as watched, editing, or deleting

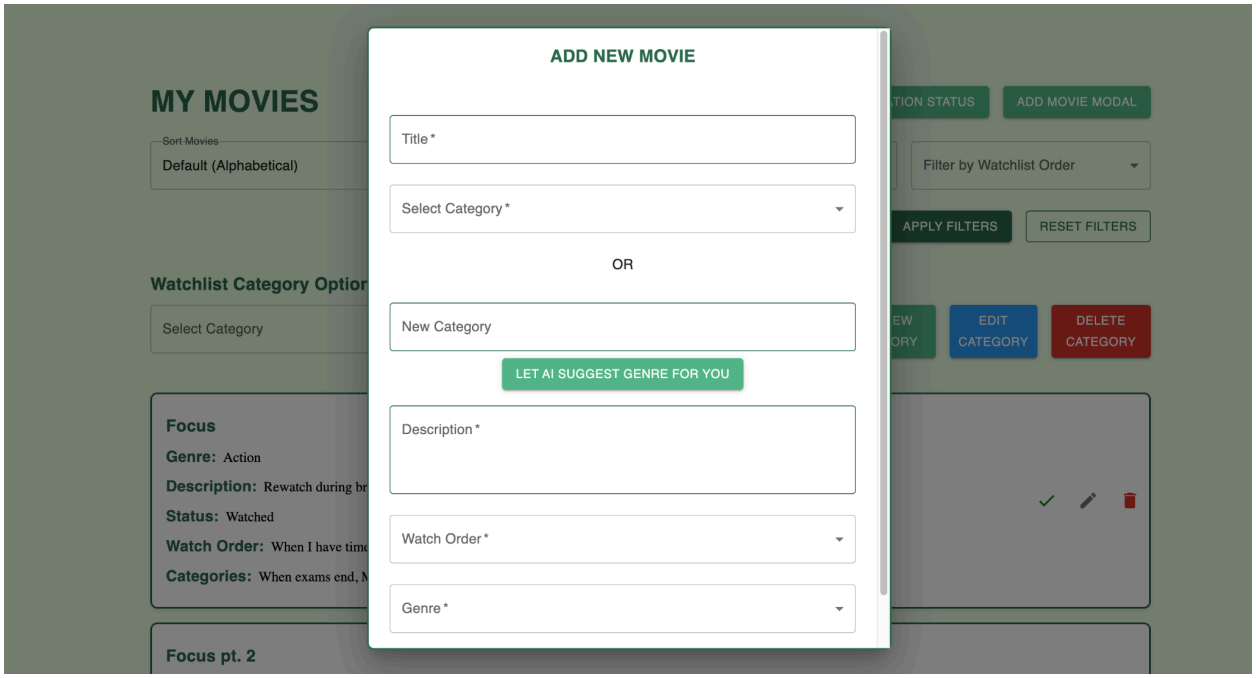


Add Movie Functionality

When a user clicks the **"Add Movie Modal"** button, a dynamic form appears. Key fields include:

- **Title** (required)
- **Description** (required)
- **Category** (select existing or create new)
- **Watchlist Order** (e.g., "Someday", "Next Up")
- **Genre** (manual or AI-suggested)

Clicking the **“Let AI suggest genre for you”** button uses OpenAI’s API to assign the most suitable genre based on the movie’s title and description.

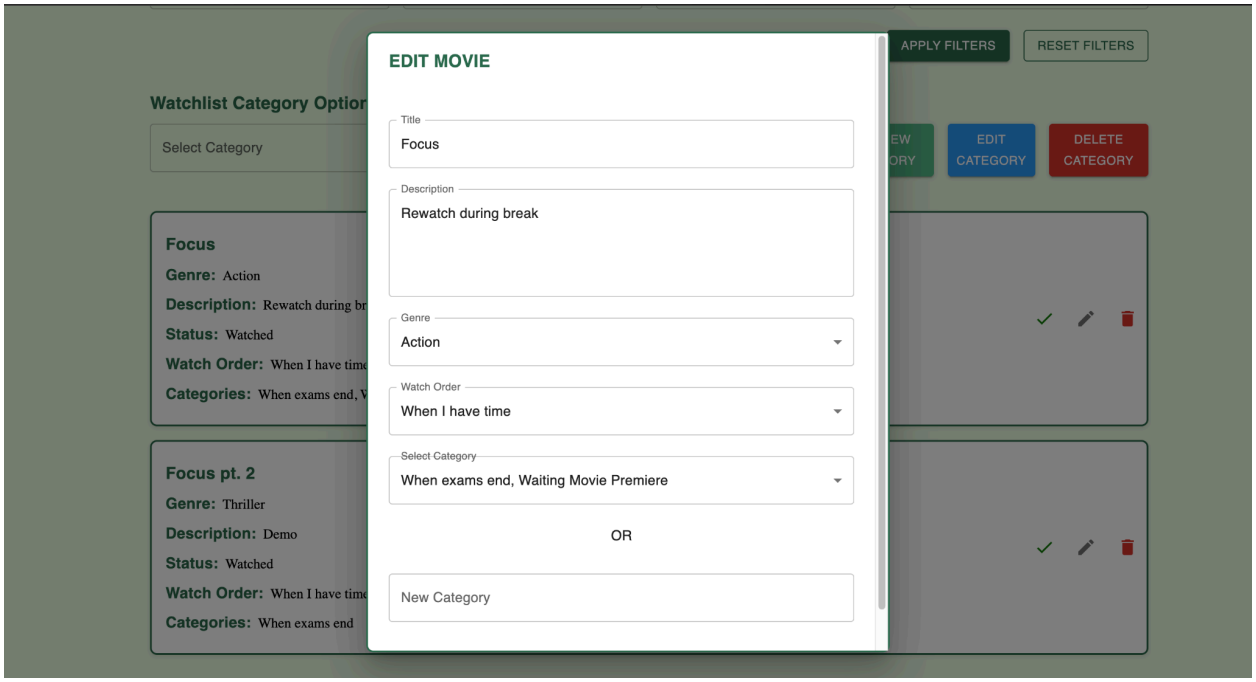


Edit Movie Functionality

Each movie in the user's watchlist can be edited at any time. By clicking the pencil icon on a movie card, a modal titled **"Edit Movie"** appears. It displays all existing information in editable fields:

- **Title** and **Description**
- **Genre**, which can be updated from the predefined list
- **Watchlist Order** (e.g., "Next Up", "When I have time")
- **Watchlist Categories**, where multiple categories can be selected
- An optional field to create and assign a **new category** on the fly

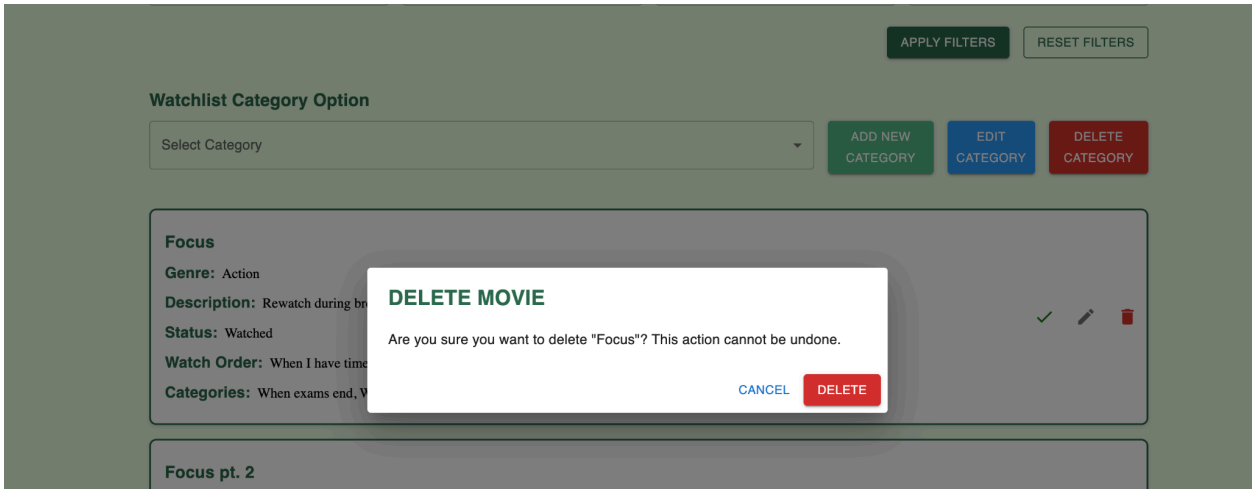
This feature allows users to refine or reorganize their movie entries as their preferences change. Upon saving changes, the updated movie details are instantly reflected on the dashboard.



Delete Movie Functionality

The red trash icon on each movie card allows users to initiate the delete operation. When clicked, a confirmation dialog appears to ensure the user intends to remove the selected movie. It includes the movie's title and a warning that the action is irreversible.

This safety confirmation helps prevent accidental deletions. Once confirmed, the movie is permanently removed from both the database and the visual watchlist.



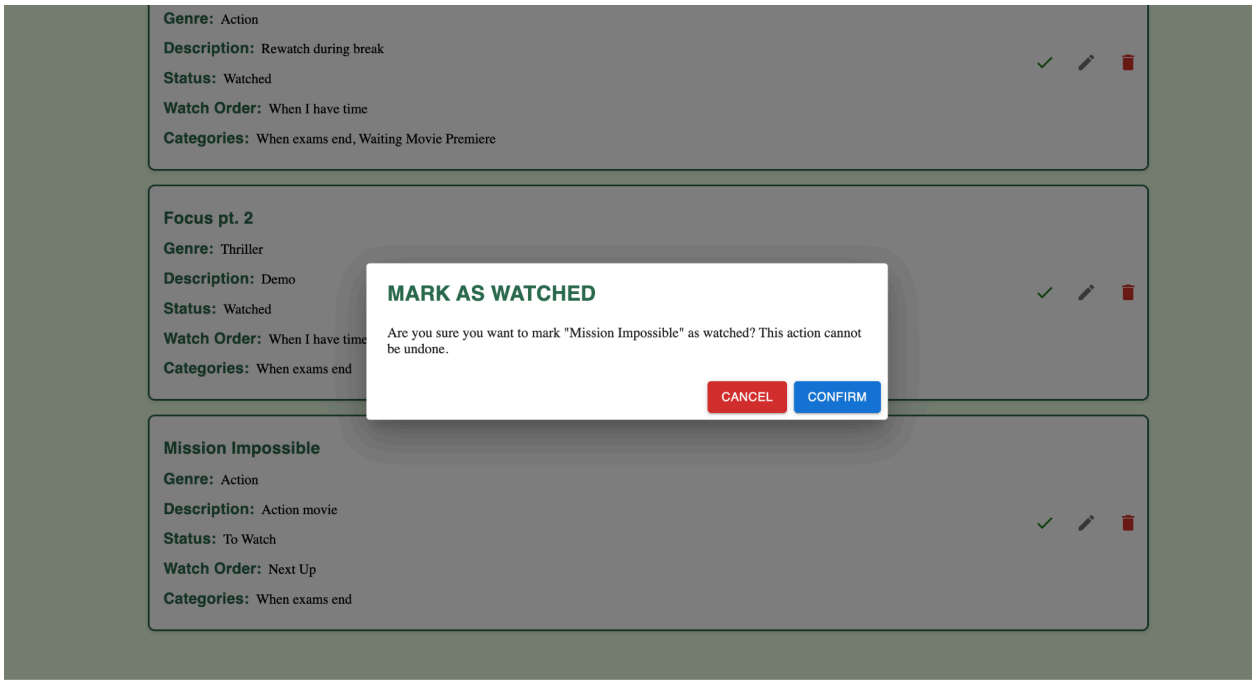
Mark Movie as Watched Functionality

Users can track their viewing progress by marking a movie as **"Watched."** This functionality is accessible directly from the movie card, using a checkmark icon. Once clicked, the movie's status is updated both visually on the frontend and persistently in the backend database.

In addition to updating the status, the system performs the following:

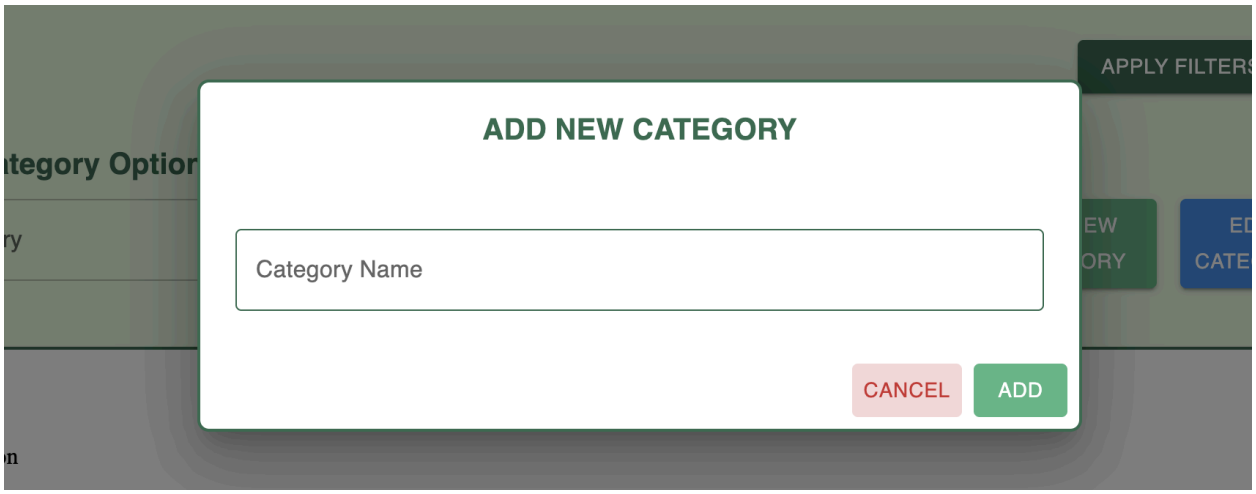
- **Triggers an email notification** (if the user has email notifications enabled)
- The email includes a confirmation message and **five AI-generated recommendations** for similar movies from the same genre, powered by the OpenAI API
- The movie remains visible in the user's watchlist but is now clearly labeled as "Watched," allowing users to revisit or reference it in the future

This feature supports user motivation and organization by providing both functional tracking and intelligent recommendations for continued engagement.



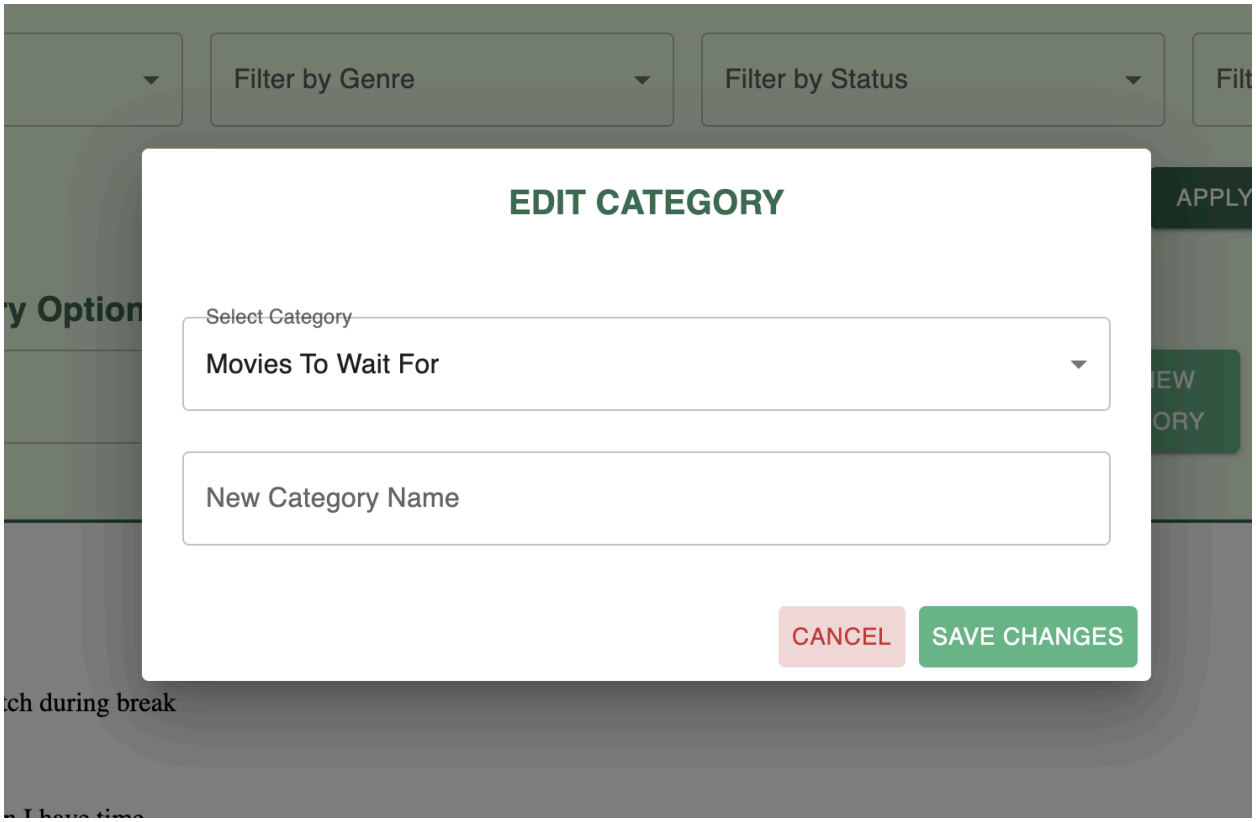
Add New Category Functionality

Users can create custom groupings (e.g., “Watch During Holidays”, “To Watch with Friends”) and assign movies accordingly.



Edit Category Name Functionality

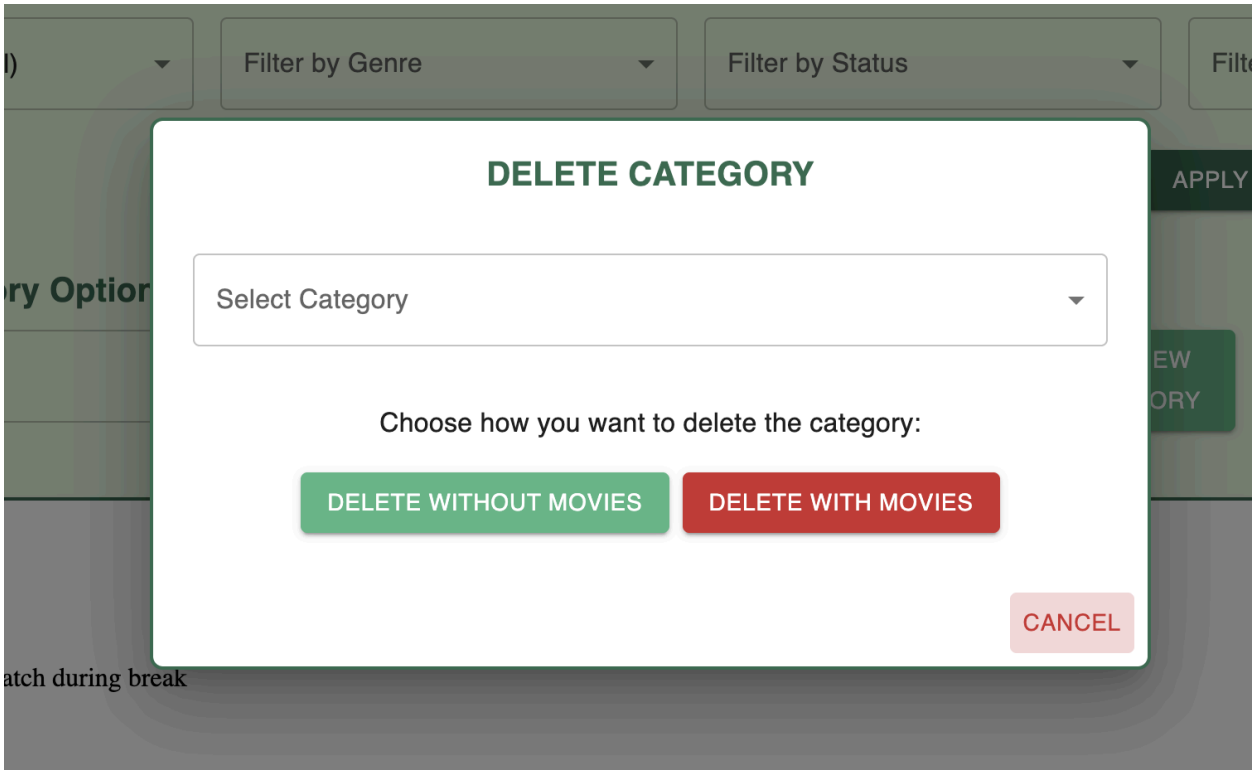
Users can rename existing categories through a clean interface. The new name reflects across all associated movies, preserving organization without data loss.



Delete Category Functionality

To give users users flexible control over their watchlist structure, two options are provided when deleting:

- **Delete without movies** – keeps all associated movies but unassigns their category.
- **Delete with movies** – permanently removes both the category and the movies inside.

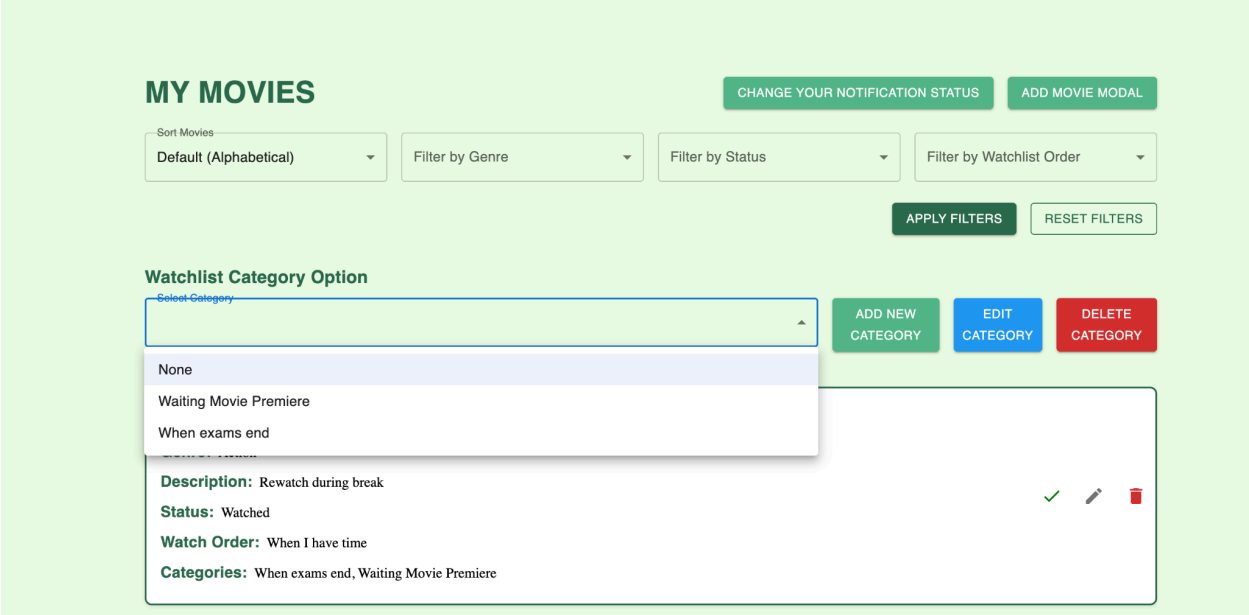


Filtering Functionality

To enhance navigation and movie discovery, the dashboard provides multiple filtering options through dropdowns:

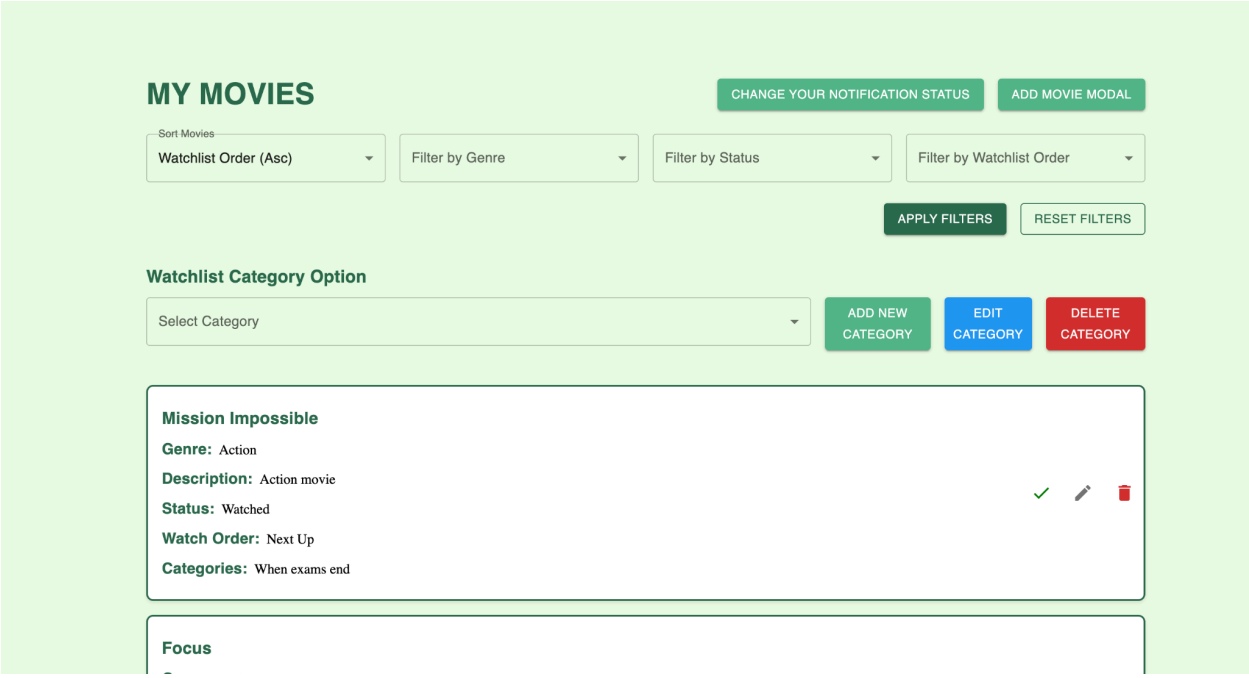
- **Genre** (e.g., Action, Comedy, Thriller)
- **Status** (To Watch / Watched)
  Watchlist Order (e.g., "Next Up", "Someday")
- **Watchlist Category Option** — a custom grouping feature allowing users to narrow results by user-defined categories (e.g., "When exams end", "Ski trip movies")

These filters can be applied in combination using the **"Apply Filters"** button. The **"Reset Filters"** option clears all selections and returns the full watchlist view.



Sorting Functionality

Movies are sorted alphabetically by title by default. Users can also sort their watchlist based on the watch order they assigned (ascending or descending), enabling personalized prioritization of what to watch next.



Change Notifications Status Functionality

The application includes a toggle for enabling or disabling **email notifications**. This functionality is accessible through the **"Change Your Notification Status"** button located in the top right section of the dashboard.

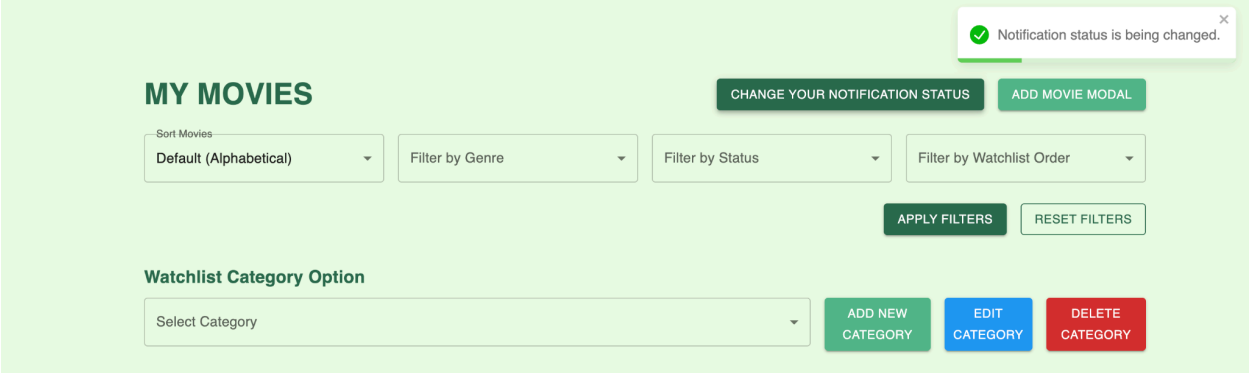
When enabled:

- The system sends an email to the user every time a movie is marked as "Watched"
- The email contains both a confirmation and five AI-recommended movie suggestions from the same genre, powered by OpenAI

When disabled:

- No emails are sent, allowing users to manage their preferences based on need, context, or privacy concerns

Upon clicking the button, a real-time alert appears confirming the notification preference has been updated.



Test Plan

Purpose and Scope

The purpose of this test plan is to define the testing strategy, tools, and activities used to verify and validate the functionality, quality, and reliability of the **Movies Watchlist** application. This includes unit testing, integration testing, system testing (UI), static code analysis, and code coverage reporting. The goal is to ensure the application meets its functional requirements as outlined in the chapter above.

Within the scope of this test plan are all major functionalities provided by the application, including:

- Adding, editing, and deleting movies
- Marking a movie as watched and sending a recommendation email
- Filtering and sorting movies by genre, watchlist order, and status
- Managing user-defined watchlist categories (create/edit/delete)
- Enabling or disabling email notifications

Testing Types and Strategy

The testing approach includes unit testing, integration testing, system (UI) testing, and static analysis of the source code.

Unit Testing (Backend)

Unit testing will be implemented using **JUnit 5**, and it will target key backend layers of the application: the **DAO (repository) layer**, **service layer**, and **controller layer**. These tests are designed to verify that individual components behave as expected in isolation, without relying on external systems.

The repository layer will use mocked return values (not the actual database information) to confirm proper data retrieval and storage operations. Mocked values keep the database integrity intact. The service layer contains the core business logic and it will include tests for operations such as adding movies, marking them as watched, editing attributes, and handling invalid inputs.

Controller tests will validate the behavior of REST endpoints using mocked service responses and confirm proper HTTP status codes and response structures.

All test methods will be isolated from the database using **Mockito library**, which ensures data consistency during testing and allows precise control over test scenarios. The overall goal is to validate internal application behavior at the method level before progressing to integration and system-level tests.

Integration Testing

Integration testing will focus on verifying the correct interaction and data flow between multiple layers and components of the backend application. In the context of the Movie Watchlist system, this primarily involves the coordination between:

- **Controllers and Service Layers**
- **Service Layer and Repository/Data Access Layer**

The goal is to ensure that when components work together, they behave as expected and consistently fulfill the business requirements.

System Testing (Selenium)

System tests simulate real-world usage scenarios through the user interface using **Selenium WebDriver**. These tests are executed on the deployed version of the app, mimicking how a user would interact with the platform in a live environment. Each test scenario covers core features, helping to ensure a stable and user-friendly experience.

Planned tests include:

- Adding a movie with all required fields
- Editing and deleting movie entries
- Filtering movies by genre and status
- Marking a movie as watched and verifying UI update
- Adding a new category
- Editing and deleting movie categories

- Soring the whole watchlist

Static Code Analysis

Static analysis will be conducted using **SonarQube and SonarScanner**, executed locally for backend code. Key metrics that will be collected are:

- Code smells and maintainability index
- Detection of unused code or poor practices
- Security rule violations and duplication

This analysis will tell whether application satisfies general clean code standards and passes all critical SonarQube rules without blocking issues.

Code Coverage Analysis

Code coverage analysis will be conducted using **JaCoCo,** which is integrated within the IntelliJ IDE. The analysis primarily targets **backend unit and integration tests,** with a focus on critical components such as the MovieService, UserService, and REST controller classes. The resulting reports will be generated and reviewed in IntelliJ, providing visual insights into which methods and branches were executed during testing.

Entry and Exit Criteria

The following table outlines the entry and exit criteria for each testing phase used in this project. This table is necessary because it tells us which specific conditions must be met to ensure that the testing environment is ready and that tests can be executed effectively. Similarly, each phase defines exit criteria that must be satisfied to consider the phase successfully completed.

Test Phase	Entry Criteria	Exit Criteria
Unit Testing	Backend builds without compilation errors	All service-level tests pass with ≥ 80% coverage
Integration	Database schema is initialized, endpoints are active	All major API flows return expected status/results
System Testing	Frontend deployed and Selenium scripts configured	All UI flows work without crashes or regressions
Static Analysis	Project is committed to GitHub	No major violations or security flaws remain

Testing Environment and Tools

The Movies Watchlist application was tested using a combination of automated and manual techniques, supported by the following tools and technologies:

Component	Tool / Technology	Purpose
Development IDE	IntelliJ IDEA	Java backend development and test execution

Build Tool	Maven	Dependency management and project builds
Programming Languages	Java, JavaScript (React)	Backend and frontend development
Unit Testing	JUnit (Java Spring Boot)	Automated testing of backend logic
Integration Testing	JUnit with Spring MockMvc	API-level testing of controller-service flow
System Testing	Selenium WebDriver (Java)	Frontend UI testing through user simulation
Static Code Analysis	SonarQube (locally hosted)	Code quality checks, rule enforcement
Browser for Tests	Google Chrome	Running and inspecting Selenium test sessions
Version Control	GitHub	Codebase hosting, versioning, collaboration
Hosting Provider	Render	Live deployment of frontend and backend

Deliverables

Throughout the SVVT project lifecycle, several testing artifacts are going to be produced to document and support the validation process. These deliverables are essential for demonstrating testing coverage, methodology, and outcomes.

The following items represent the primary deliverables of the verification and validation process:

- Unit test source files (JUnit)
- Selenium scripts
- Code coverage with JaCoCo
- Static code analysis summary from SonarQube
- Screenshots of test execution and passed cases
- Final summary report

Test Execution

Unit Testing

Unit testing in this project was conducted using the **JUnit 5** framework, with extensive use of **Mockito** for mocking dependencies. These tests validated the internal behavior of the backend logic and ensured that each method behaves correctly in isolation. The objective was to confirm that:

- Repositories correctly return mock data
- Services correctly process logic based on mocked inputs
- Controllers return proper HTTP responses based on mocked services

Scope of Unit Tests

The test suite includes classes that cover all core backend components:

- **Repository Layer** – basic mocking coverage to validate DAO methods
- **Service Layer** – full coverage with all CRUD operations, business logic, and input validation

- **Controller Layer** – endpoint-level behavior and HTTP response validation using MockMvc

Unit test methods are completely isolated from the live database. All repository interactions are mocked to preserve data integrity and prevent side effects during testing.

Code Coverage

As observed from the code coverage summary (see attached screenshot), backend unit testing achieved:

- **~85% method coverage**
- **Over 80% branch and line coverage** for key services
- Most tested classes: MovieService, UserService, WatchlistGroupService, and MovieController

Tests such as:

- testMarkAsWatched_Success() and testMarkAsWatched_AlreadyWatched()
- testAddMovie(), testEditMovie_Simple()
- testFilterMoviesByStatus(), testSortMoviesByWatchlistOrderAsc()

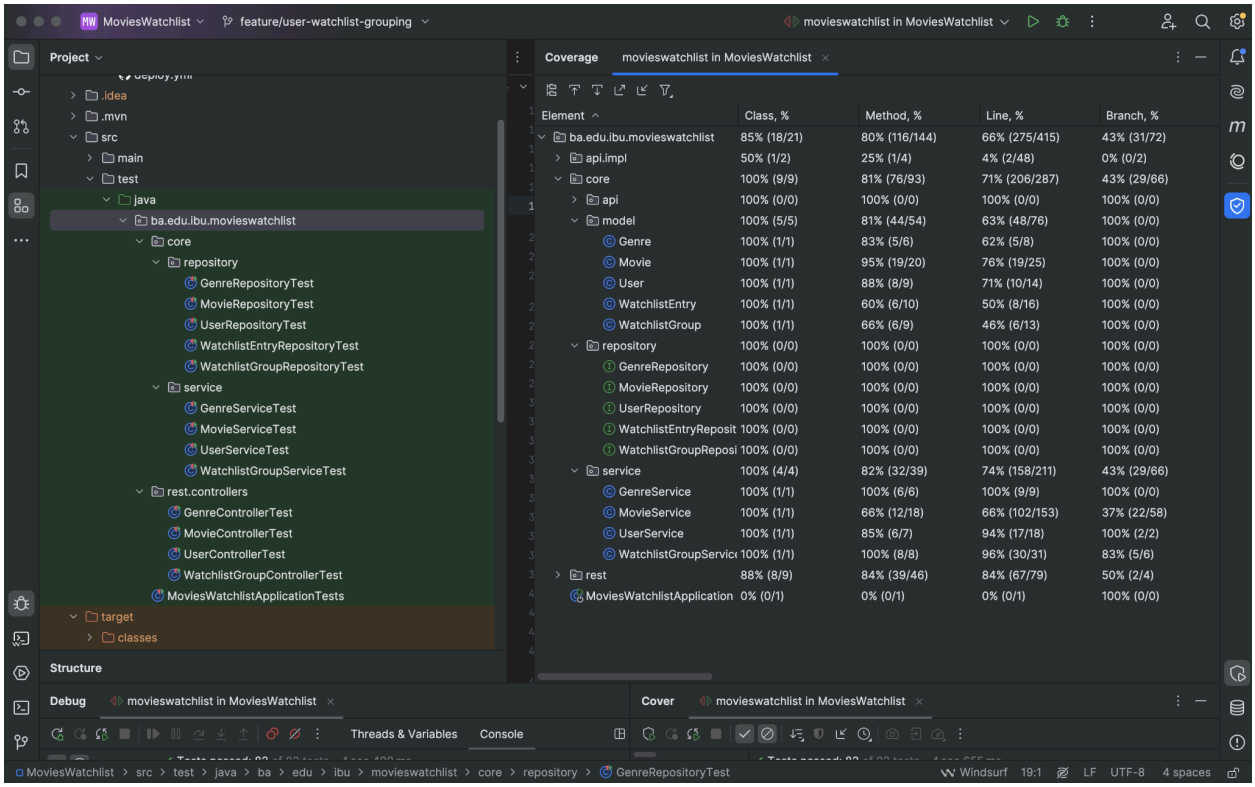
Are examples that the small parts of the code (unit logic) work correctly in different situations, including unusual cases and when errors happen.

To conclude, all unit tests passed successfully with expected assertions. The codebase is highly stable and internally consistent based on the tested business rules.

Artifacts produced in this part are:

- JUnit test source code (organized by layer)
- Screenshot of IntelliJ code coverage summary

- Test logs from console output



Integration Testing

Integration testing was conducted to verify that various backend components—such as controllers, services, and repositories—communicate correctly and produce the expected results when working together. These tests are written using JUnit and Spring's testing support, including @SpringBootTest and MockMvc.

Scope of Integration Tests

The tests focus on validating the behavior of complete request flows for:

- Creating and editing movies
- Marking movies as watched
- Filtering and sorting via endpoint parameters
- Managing watchlist categories
- Toggling notification settings

Rather than mocking all dependencies, integration tests rely on real service and controller layers while mocking the database (using @DataJpaTest), in order not to change actual data.

Test Objectives

- Ensure that HTTP requests made to the REST API result in the expected logic being executed
- Confirm that service methods and validation are triggered properly when requests pass through controllers
  Validate edge cases and negative scenarios (e.g., missing parameters, invalid input, or nonexistent IDs)

Example Test Flows

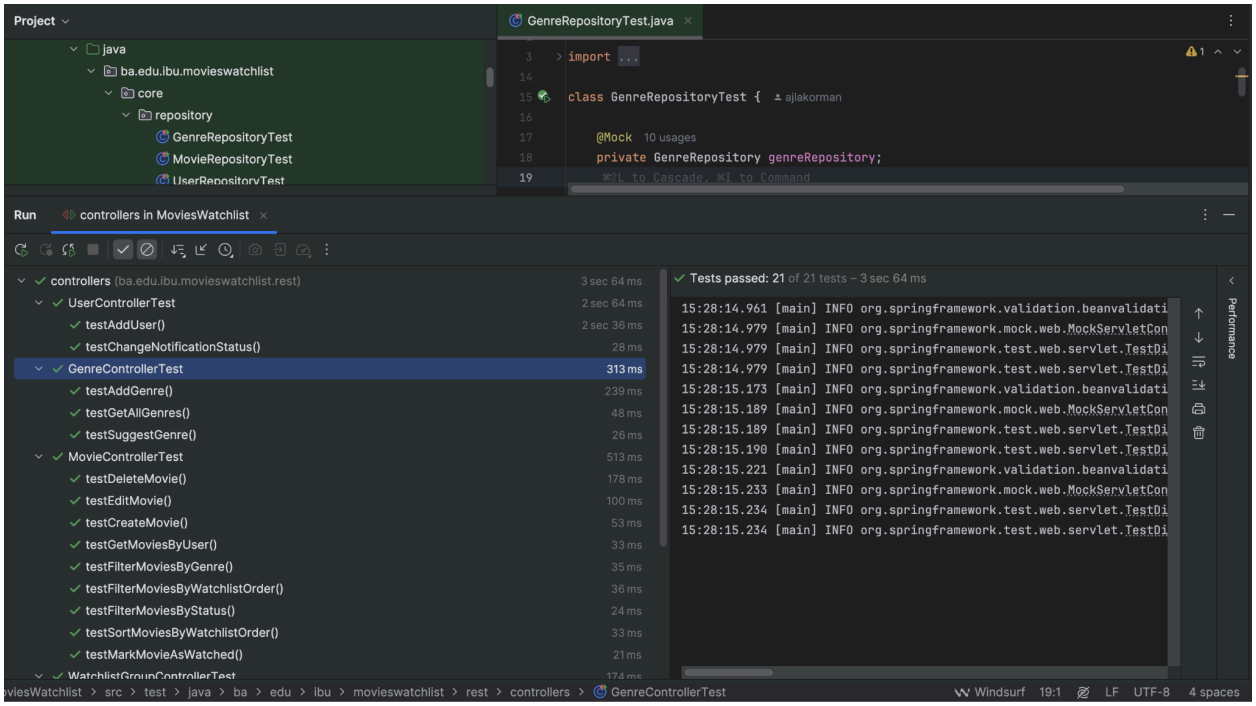
- Sending a POST /api/movies request with valid payload and checking that the movie is persisted and a 201 status is returned
- Updating an existing movie using PUT /api/movies/{id} and asserting that changes are saved correctly
- Using GET /api/movies/filter?status=Watched to ensure the filtering logic works end-to-end

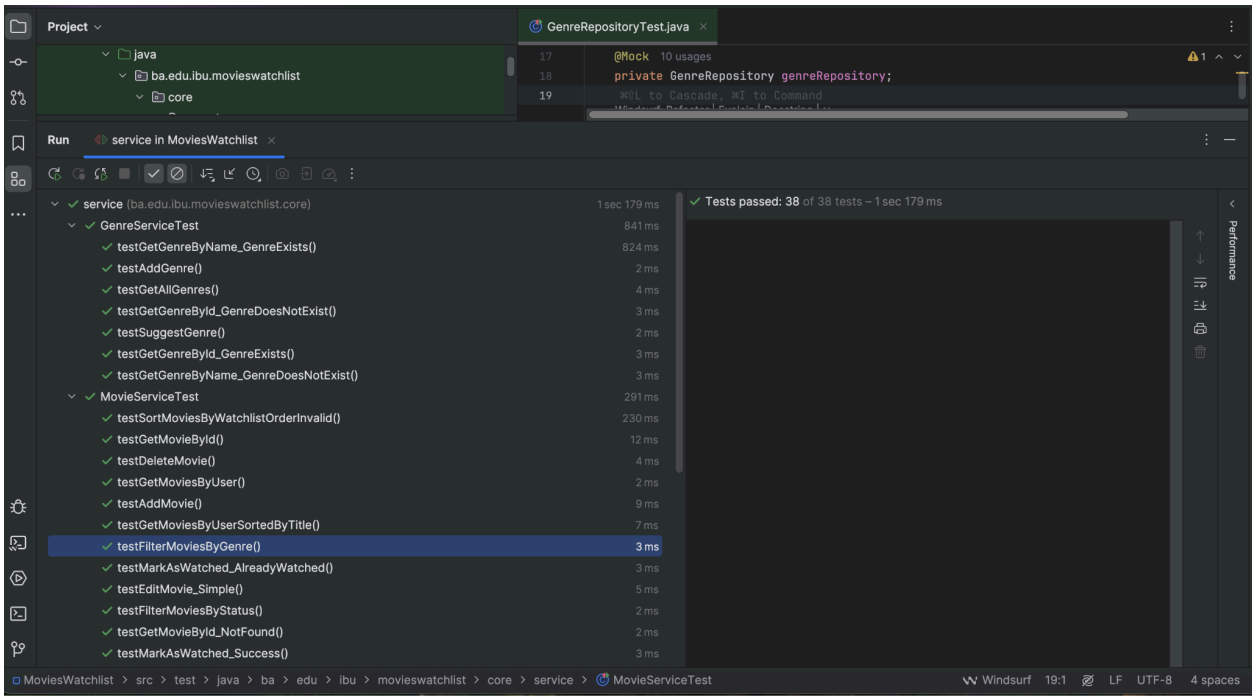
In the end, all integration tests passed successfully and confirmed that the system correctly handled expected and edge-case user actions. Test cases validated status codes, returned content structures, and method execution paths across all major backend workflows.

Artifacts produced in this part are:

- Integration test source files (JUnit with Spring Boot)
- Console output showing successful end-to-end test execution

38 tests that I had in service layer, alongside 21 test in the conntrollers layer have successfully passed. The screenshots attached below serve as an evidence for that. Additionally, here in the GitHub repository for backend in folders **core/services and rest/controllers,** you can analyze the code for these tests in detail.


Project
java
ba.edu.ibu.movieswatchlist
core
repository
GenreRepositoryTest
MovieRepositoryTest
GenreRepositoryTest.java
import ...
class GenreRepositoryTest {
@Mock 10 usages
private GenreRepository genreRepository;
Run
controllers in MoviesWatchlist
controllers (ba.edu.ibu.movieswatchlist.rest) 3 sec 64 ms
UserControllerTest 2 sec 64 ms
testAddUser() 2 sec 36 ms
testChangeNotificationStatus() 28 ms
GenreControllerTest 313 ms
testAddGenre() 239 ms
testGetAllGenres() 48 ms
testSuggestGenre() 26 ms
MovieControllerTest 513 ms
testDeleteMovie() 178 ms
testEditMovie() 100 ms
testCreateMovie() 53 ms
testGetMoviesByUser() 33 ms
testFilterMoviesByGenre() 35 ms
testFilterMoviesByWatchlistOrder() 36 ms
testFilterMoviesByStatus() 24 ms
testSortMoviesByWatchlistOrder() 33 ms
testMarkMovieAsWatched() 21 ms
Tests passed: 21 of 21 tests – 3 sec 64 ms
15:28:14.961 [main] INFO org.springframework.validation.beanvalidati
15:28:14.979 [main] INFO org.springframework.mock.web.MockServletCon
15:28:14.979 [main] INFO org.springframework.test.web.servlet.TestDi
15:28:14.979 [main] INFO org.springframework.test.web.servlet.TestDi
15:28:15.173 [main] INFO org.springframework.validation.beanvalidati
15:28:15.189 [main] INFO org.springframework.mock.web.MockServletCon
15:28:15.189 [main] INFO org.springframework.test.web.servlet.TestDi
15:28:15.190 [main] INFO org.springframework.test.web.servlet.TestDi
15:28:15.221 [main] INFO org.springframework.validation.beanvalidati
15:28:15.233 [main] INFO org.springframework.mock.web.MockServletCon
15:28:15.234 [main] INFO org.springframework.test.web.servlet.TestDi
15:28:15.234 [main] INFO org.springframework.test.web.servlet.TestDi
Performance
GenreControllerTest
Windsurf 19:1 LF UTF-8 4 spaces

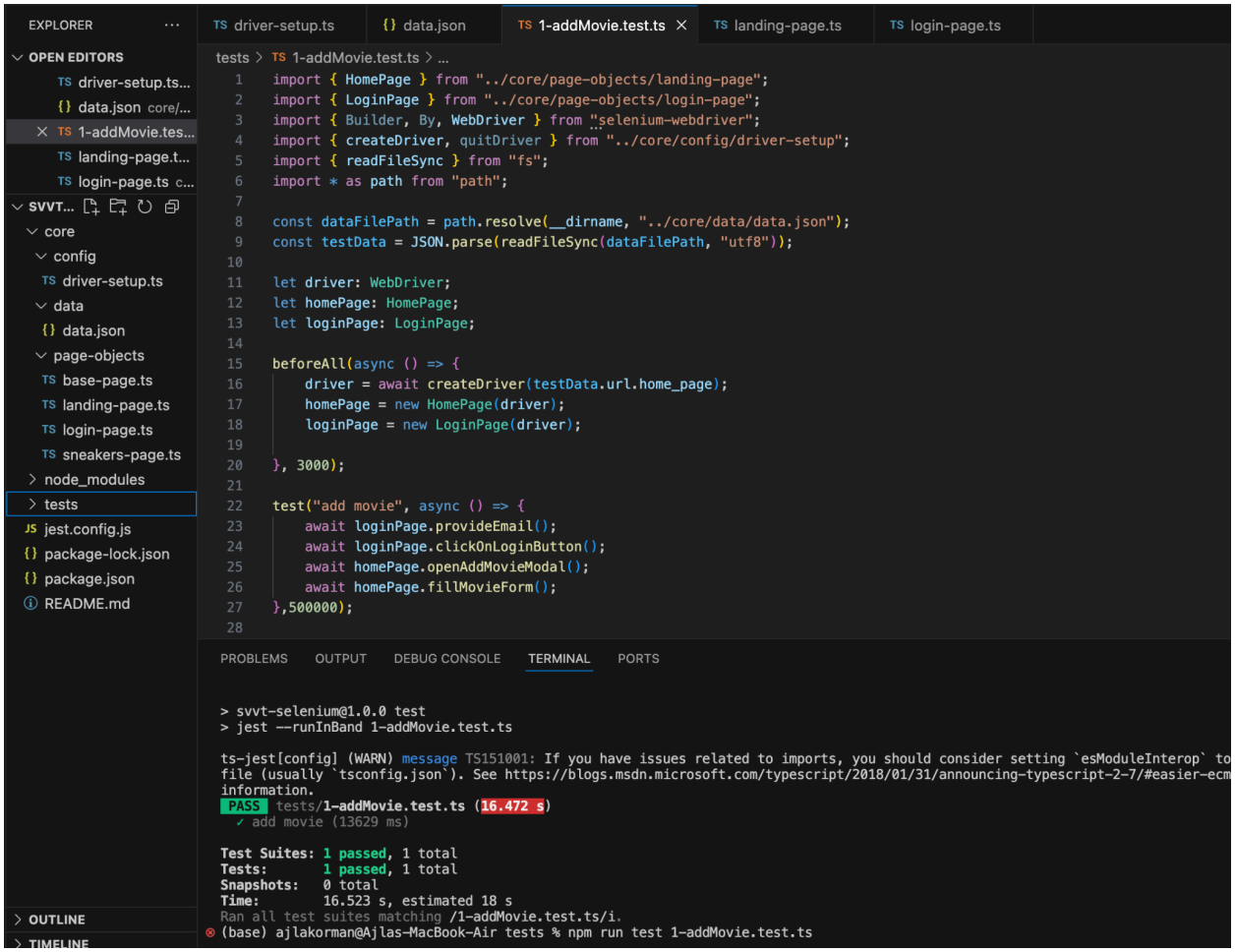

Project
java
ba.edu.ibu.movieswatchlist
core
GenreRepositoryTest.java
@Mock 10 usages
private GenreRepository genreRepository;
Run
service in MoviesWatchlist
service (ba.edu.ibu.movieswatchlist.core) 1 sec 179 ms
GenreServiceTest 841 ms
testGetGenreByName_GenreExists() 824 ms
testAddGenre() 2 ms
testGetAllGenres() 4 ms
testGetGenreById_GenreDoesNotExist() 3 ms
testSuggestGenre() 2 ms
testGetGenreById_GenreExists() 3 ms
testGetGenreByName_GenreDoesNotExist() 3 ms
MovieServiceTest 291 ms
testSortMoviesByWatchlistOrderInvalid() 230 ms
testGetMovieById() 12 ms
testDeleteMovie() 4 ms
testGetMoviesByUser() 2 ms
testAddMovie() 9 ms
testGetMoviesByUserSortedByTitle() 7 ms
testFilterMoviesByGenre() 3 ms
testMarkAsWatched_AlreadyWatched() 3 ms
testEditMovie_Simple() 5 ms
testFilterMoviesByStatus() 2 ms
testGetMovieById_NotFound() 2 ms
testMarkAsWatched_Success() 3 ms
Tests passed: 38 of 38 tests – 1 sec 179 ms
Performance
MovieServiceTest
Windsurf 19:1 LF UTF-8 4 spaces

System Testing (Selenium)

System testing was implemented using **Selenium WebDriver** to automate real user interactions with the deployed application. Code for the described tests below is available in this GitHub Repository here.

Test Case 1 - Add a movie with valid data

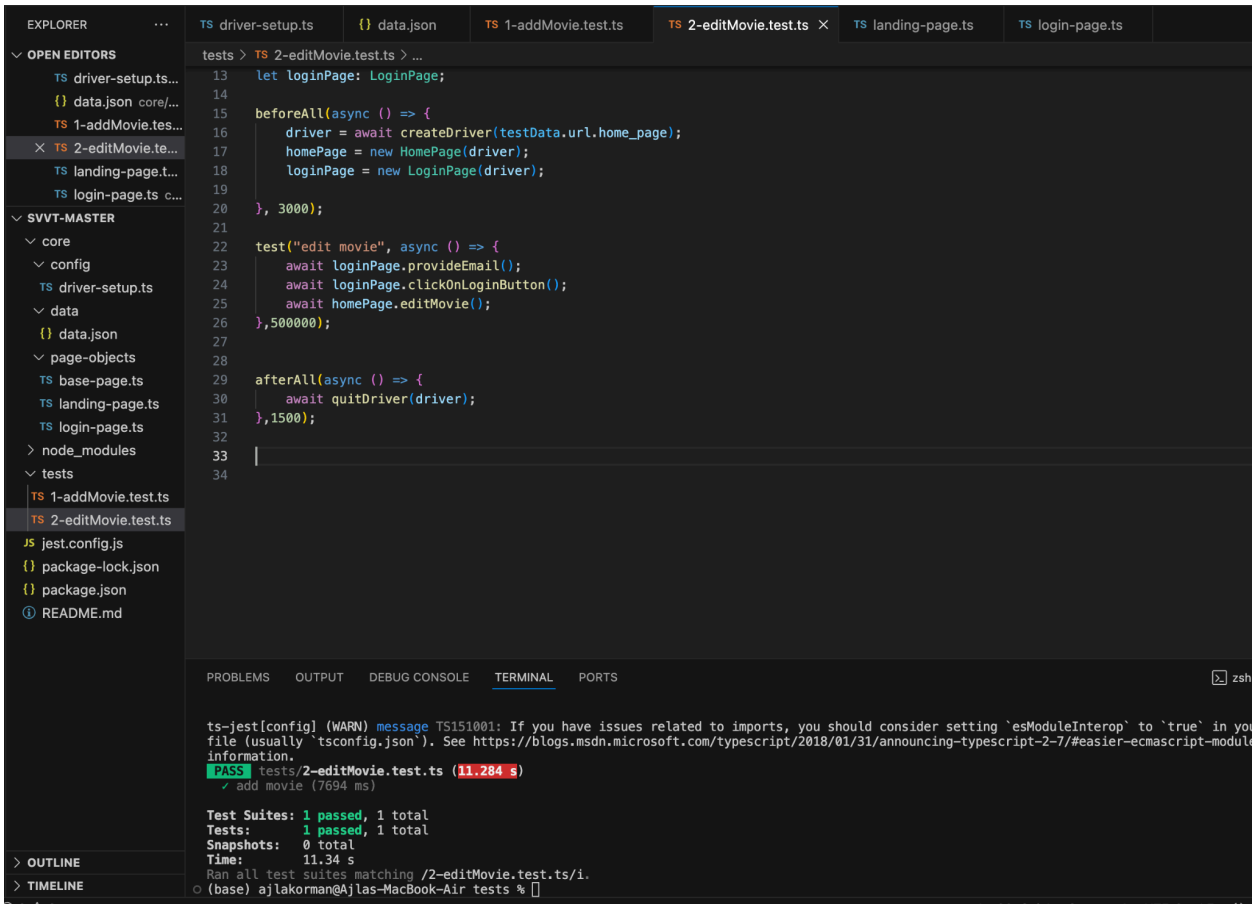
Test Name: Add Movie Modal – Positive Flow Test				
Description: This test verifies that a user can successfully open the Add Movie Modal, enter valid data into the movie form fields, and submit the form to add a new movie to the list.				
Pre-condition(s): - User is logged in - Homepage is fully loaded				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click on the "Add Movie Modal" button 2. Fill in all required fields: – Title – Description – Category (new + dropdown) – Watchlist Order – Genre 3. Click on "Add Movie" button 4. Verify that the movie is added to the list and appears with correct data	Title: Interstellar Description: A sci-fi movie about space travel Category: Sci-Fi Watchlist Order: Next Up Genre: Adventure Dropdown Category: Waiting Movie Premiere	The movie Interstellar is successfully added and displayed in the user's watchlist with all the entered details	The movie Interstellar appears in the list with correct title, description, genre, and watchlist information	**PASS**



Test Case 2 - Edit an existing movie

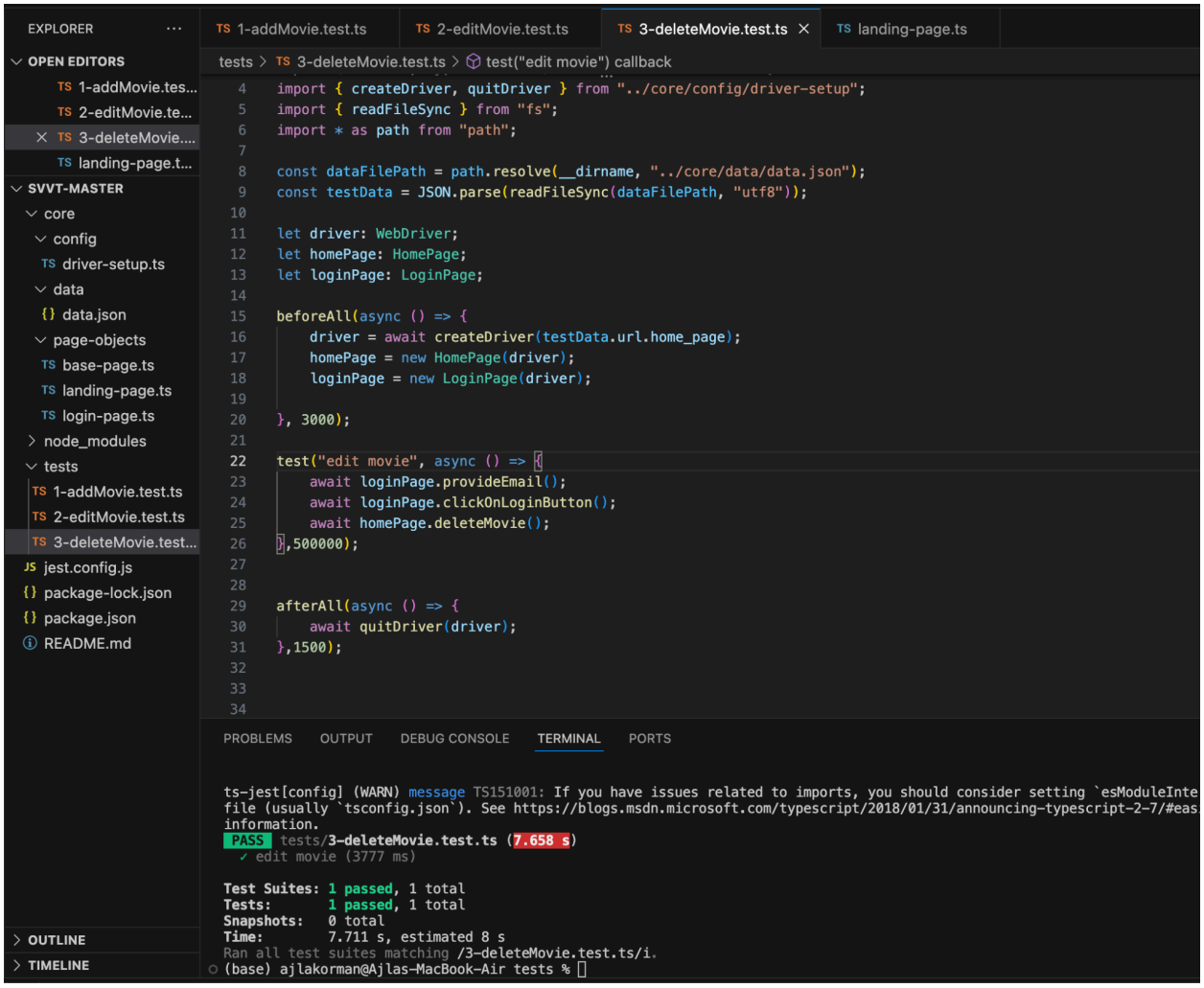
Test Name: Edit Movie – Update Description and Genre				
Description: This test verifies that a user can successfully edit an existing movie's description and genre using the UI and save the updated data.				
Pre-condition(s): - A movie must exist in the list - User is logged in and on the homepage				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click the edit icon on a movie card 2. Clear and enter a new	New Description: Rewatch during break – now with friends and	The movie is updated with the new description and genre, and changes are	Description and genre were successfully updated and are visible in the	**PASS**

description 3. Open genre dropdown and select “Comedy” 4. Click “Save Changes” 5. Verify updated values appear on the movie card	snacks. New Genre: Comedy	reflected in the UI	watchlist	



Test Case 3 - Delete a Movie

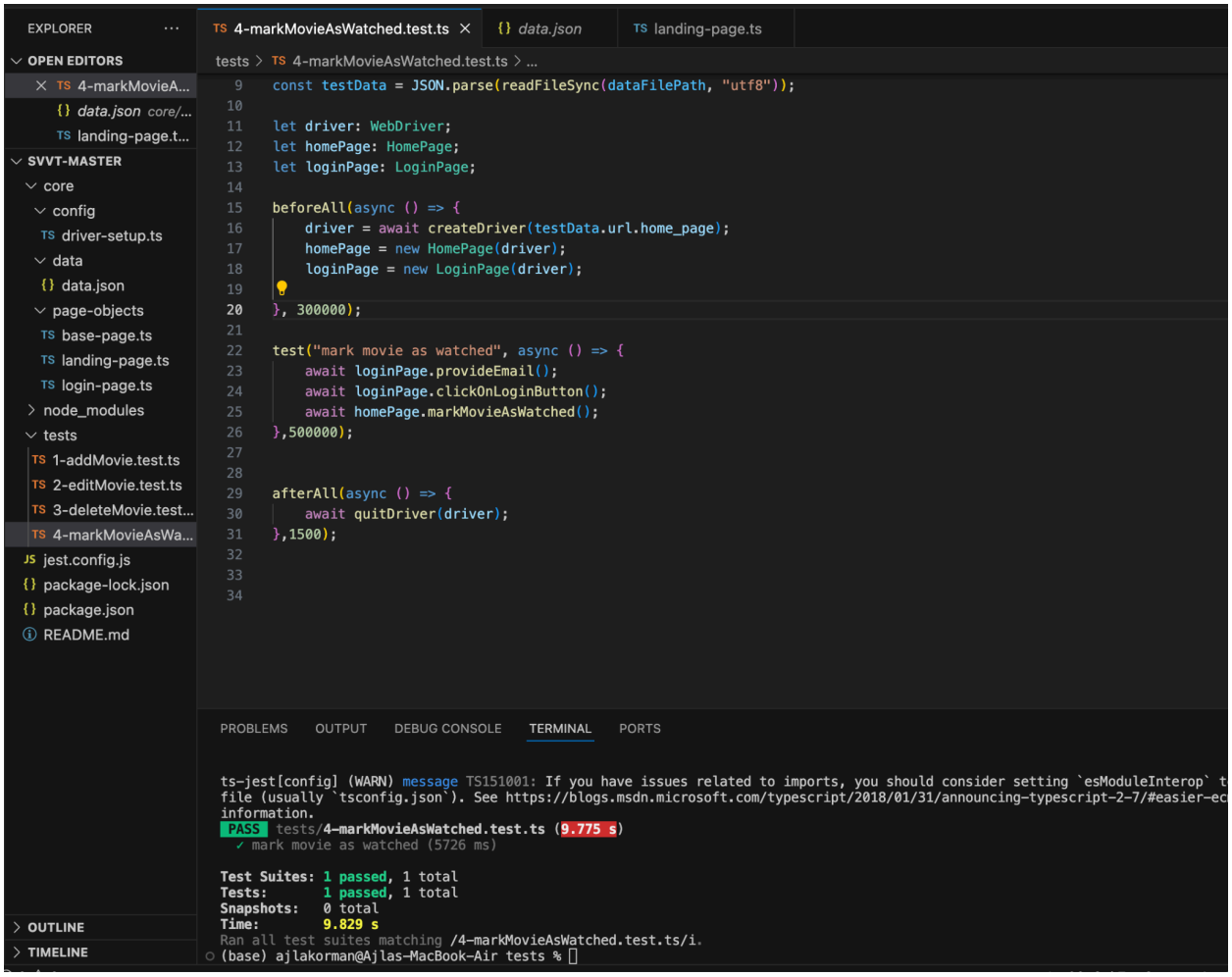
Test Name: Delete Movie – Confirmation Modal Flow				
Description: This test ensures that a movie can be deleted using the trash icon, and that the confirmation modal functions correctly.				
Pre-condition(s): - A movie exists in the watchlist - User is logged in and on the homepage				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click the delete icon on the movie card 2. Wait for the confirmation modal 3. Click the "Delete" button in the modal 4. Confirm that the movie is removed from the list	Any existing movie in the user's watchlist	The movie is removed from the interface and no longer appears in the list	The movie is successfully deleted after confirmation	**PASS**



Test Case 4 - Mark Movie as Watched

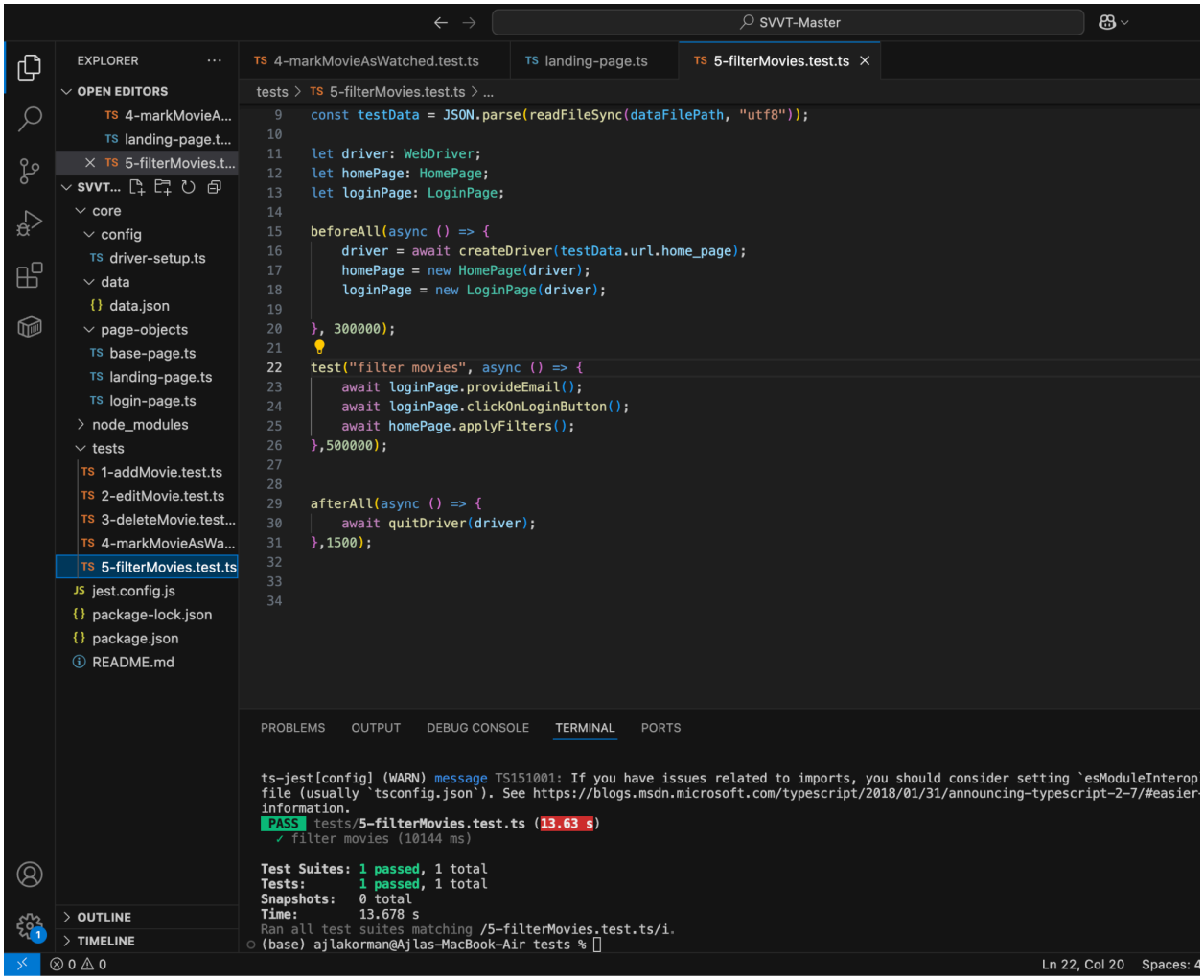
Test Name: Mark Movie as Watched – Status Change Verification				
Description: This test confirms that a movie marked as "To Watch" can be updated to "Watched" through the UI, and that the change is reflected correctly.				
Pre-condition(s): - Movie with status "To Watch" exists - User is logged in and on homepage				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Read current movie status from UI 2. Click on the	Any movie card where status = "To Watch"	Status value updates from "To Watch" to "Watched" after	Movie status changed successfully and reflected on UI	**PASS**

check icon (“mark as watched”) 3. Confirm action in modal 4. Wait for status to update 5. Verify the status now reads “Watched”		confirmation		



Test Case 5 - Apply Filters

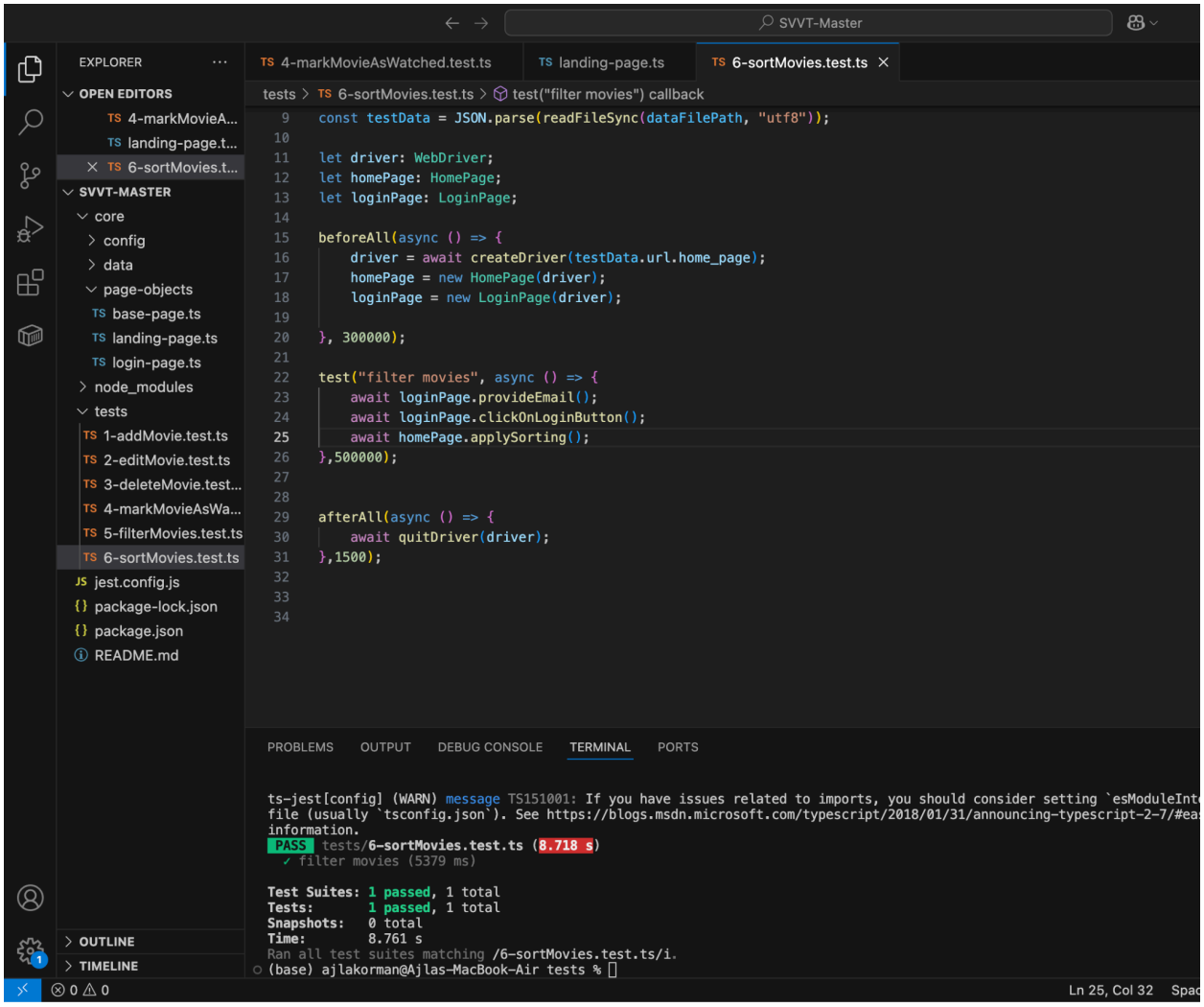
Test Name: Filter Movies – Genre, Status, and Category				
Description: This test ensures that the user can apply filters based on genre, status, and watchlist category using the filter dropdowns and see a filtered list.				
Pre-condition(s): - User is logged in - Movies exist in the list covering various genres, statuses, and categories				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Open genre dropdown and select "Action" 2. Open status dropdown and select "Watched" 3. Open watchlist category dropdown and select "Waiting Movie Premiere" 4. Click "Apply Filters" 5. Confirm that the filtered results reflect the selected criteria	Genre: Action Status: Watched Category: Waiting Movie Premiere	The movie list updates to only show items that match all selected filters	Movie list was successfully filtered based on genre, status, and category	**PASS**



Test Case 6 - Apply Sorting

Test Name: Sort Movies – Watchlist Order Ascending				
Description: This test checks if movies can be sorted by watchlist order in ascending sequence using the sorting dropdown and Apply Filters button.				
Pre-condition(s): - User is logged in - Watchlist contains multiple movies with varying watchlist positions				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Open the sort dropdown 2. Select “Watchlist Order	Sort option: Watchlist Order (Asc)	Movies are reordered in ascending watchlist order	Movies appeared in the expected order, starting from "Next Up"	**PASS**

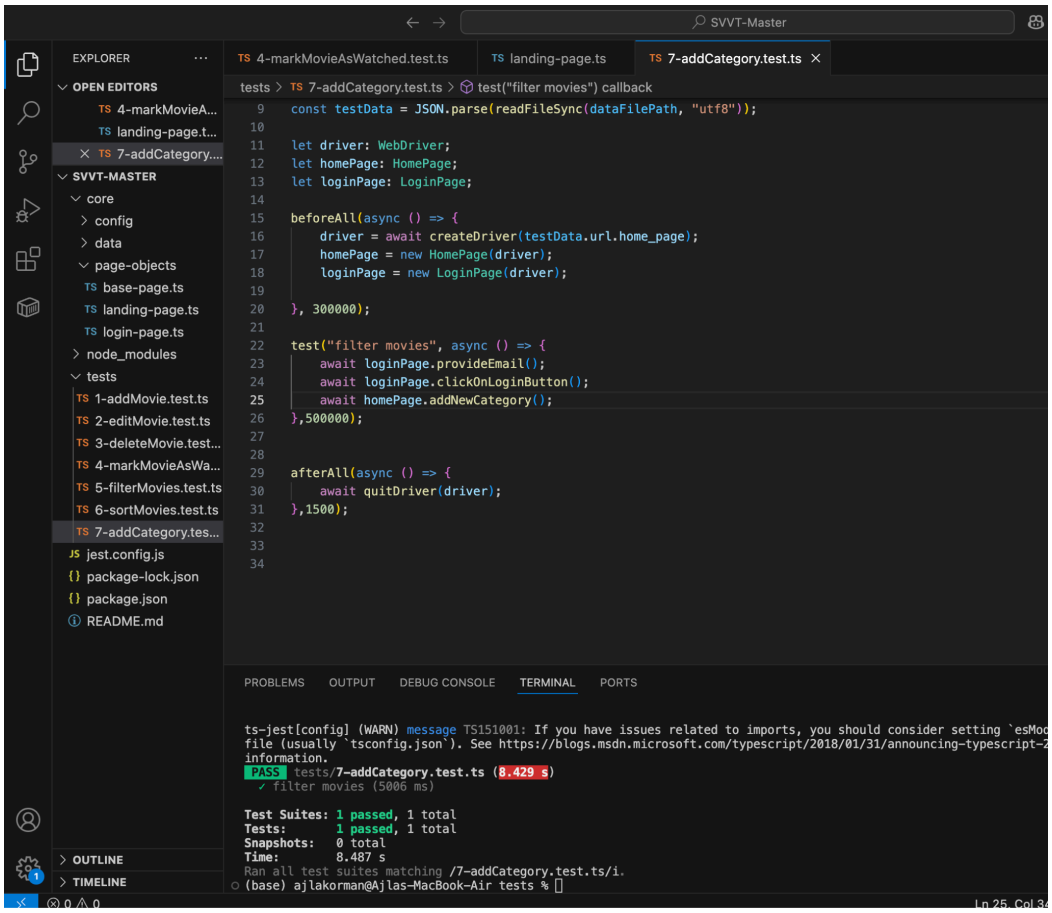
(Asc)" 3. Click "Apply Filters" 4. Verify the movie list is reordered accordingly		on the UI	or lowest rank	



Test case 7 - Add New Category

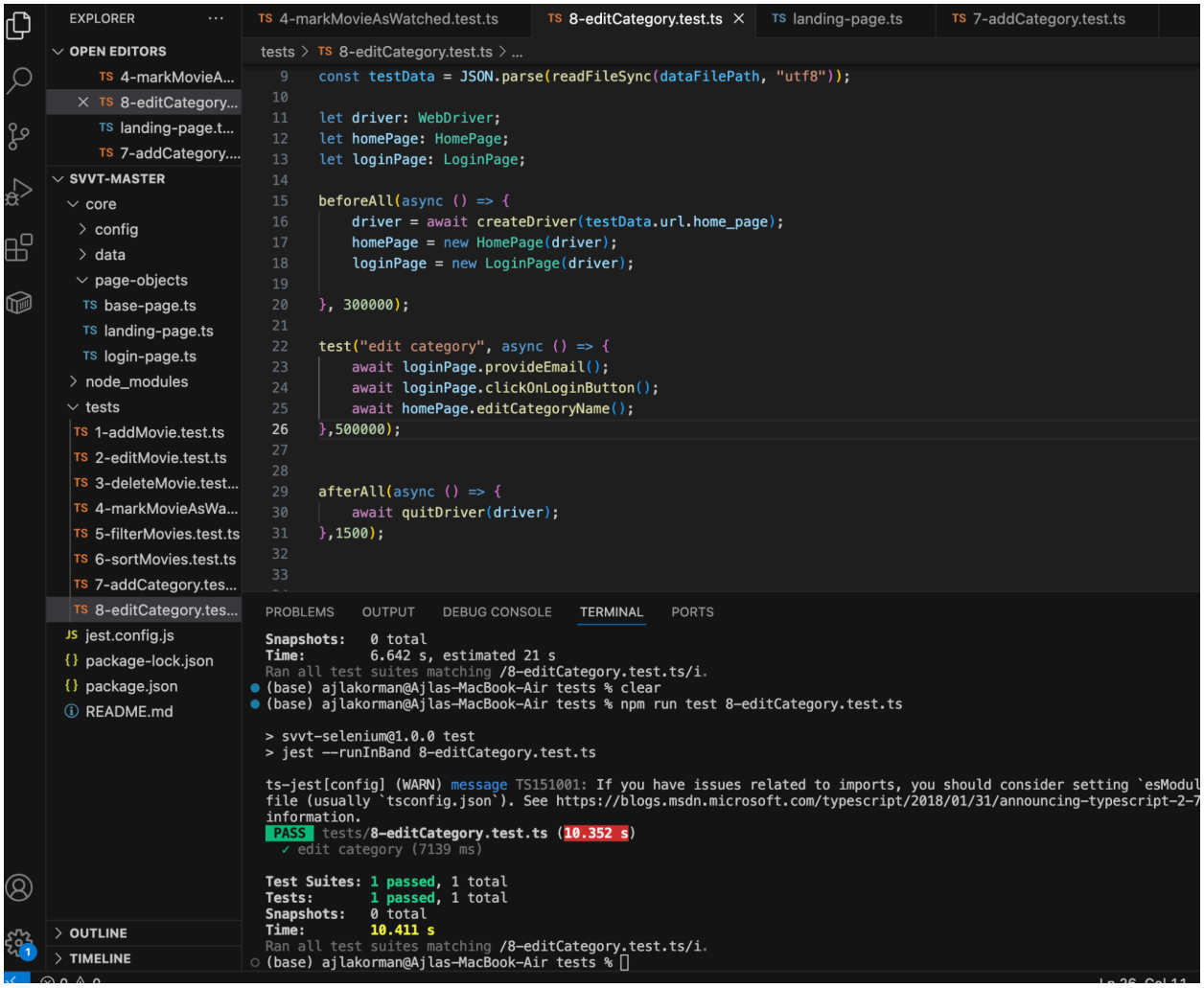
Test Name: Add New Watchlist Category
Description: This test verifies that a user can create a new watchlist category using the "Add New Category" modal.

Pre-condition(s): - User is logged in - Category with the same name does not already exist				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click on “Add New Category” button 2. Enter a new category name 3. Click the “Add” button 4. Wait for UI to update and confirm category appears in dropdown	New Category: "Holiday Picks"	The new category “Holiday Picks” is added and becomes available for selection in the category dropdown	New category successfully appeared and was selectable from the list	**PASS**



Test Case 8 - Edit Category

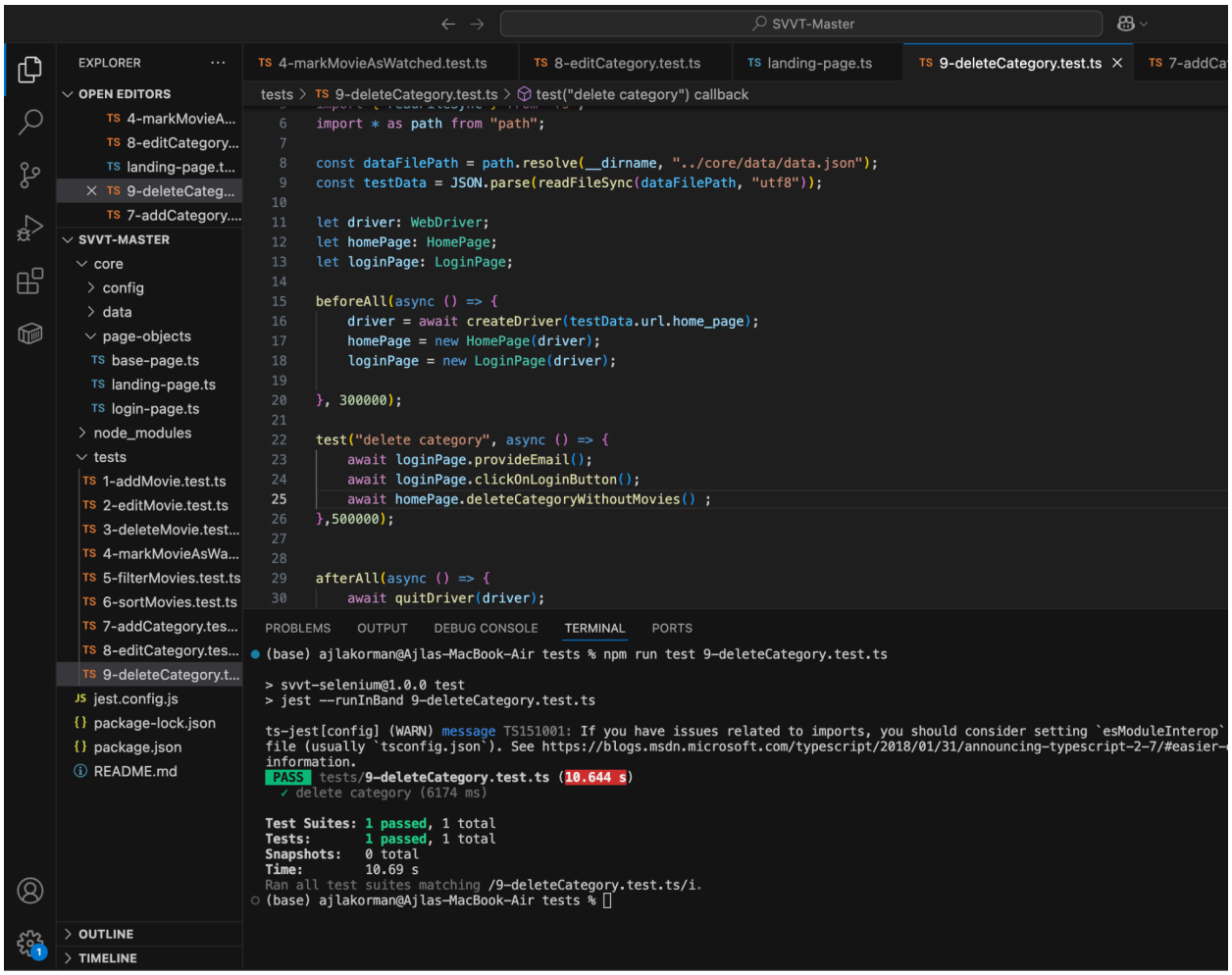
Test Name: Edit Watchlist Category – Rename Existing Category				
Description: This test validates that an existing watchlist category name can be updated through the edit category modal.				
Pre-condition(s): - Category "Watch during Winter Holidays" exists - User is logged in				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click on "Edit Category" button 2. Open the dropdown and select **"Watch during Winter Holidays"** 3. Enter a new name: "Holiday Picks" 4. Click on "Save Changes" 5. Verify the name is updated and shown in dropdown	Old Name: Watch during Winter Holidays New Name: Holiday Picks	Category name is successfully updated and reflected in the UI	The selected category was renamed to "Holiday Picks" and appeared updated in the dropdown	**PASS**



Test Case 9 - Delete Category

Test Name: Delete Watchlist Category – Without Associated Movies				
Description: This test ensures that a watchlist category can be deleted even if it contains movies, by choosing to delete the category without its associated movies.				
Pre-condition(s): - Category “Holiday Best Picks” exists - User is logged in				
Test Steps	**Test Data**	**Expected Result**	**Actual Result**	**Status**
1. Click on “Delete Category” button	Category to delete: Holiday Best Picks	Category is deleted while keeping the movies in the	Category removed successfully and UI reflects	**PASS**

2. Select “Holiday Best Picks” from dropdown 3. Click on “Delete Without Movies” button 4. Verify that the category no longer appears in the category list		watchlist under no category	change	



Static Code Analysis

SonarQube was used for static code analysis in this project. It is a widely used static code analysis tool that automatically reviews code without executing it. It is used in professional software development teams to enforce clean coding standards and ensure long-term code quality and consistency. Its primary purpose is to detect:

- Code smells (inefficient coding practices)
- Security vulnerabilities
- Reliability issues (bugs that could cause runtime failures)
- Code duplications
- Overall maintainability issues

SonarScanner is the command-line tool that analyzes your code and sends the results to a SonarQube server for processing and visualization. It reads configuration from the sonar-project.properties file placed in the project root directory, and it includes:

- Project name and version
- Source and test code locations
- Compilation output path (target/classes)

In this project, SonarQube Community Edition was installed and run locally, and was used to analyze the Java Spring Boot backend of the Movie Watchlist application.

The process was as follows:

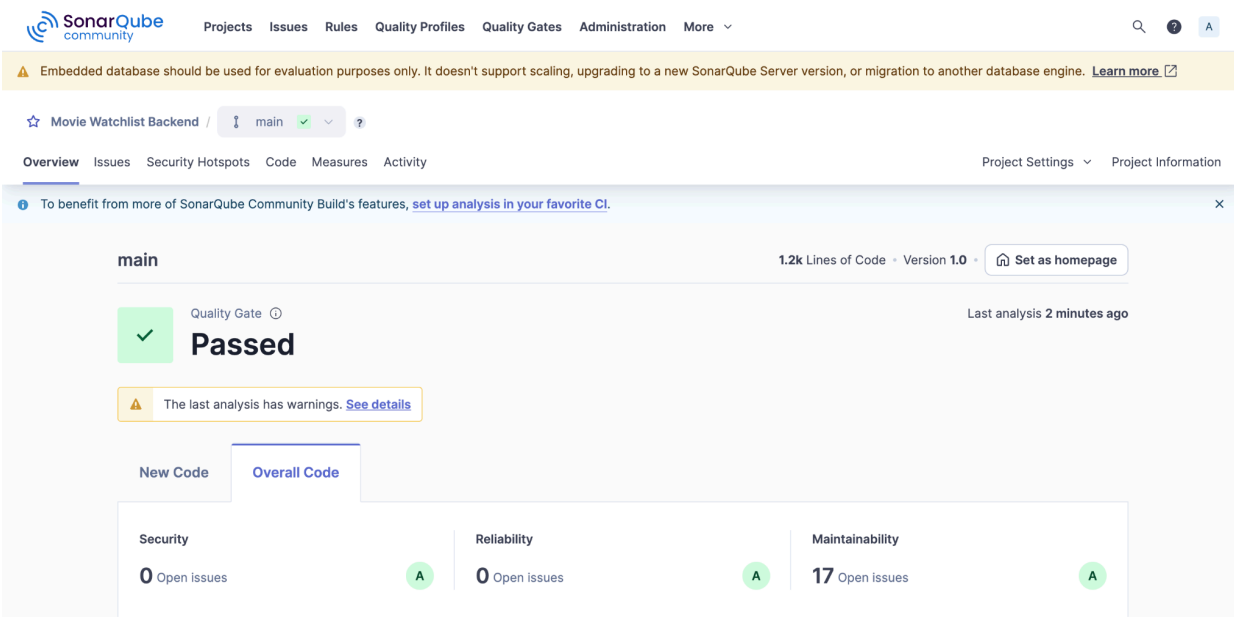
1. SonarQube server was started locally and accessed via http://localhost:9000
2. A configuration file named sonar-project.properties was created in the project root, specifying:
    - The source directory: src/main/java
    - The test directory: src/test/java
    - Project metadata
3. The project was compiled using Maven (mvn clean install)
4. SonarScanner was then executed to start the analysis

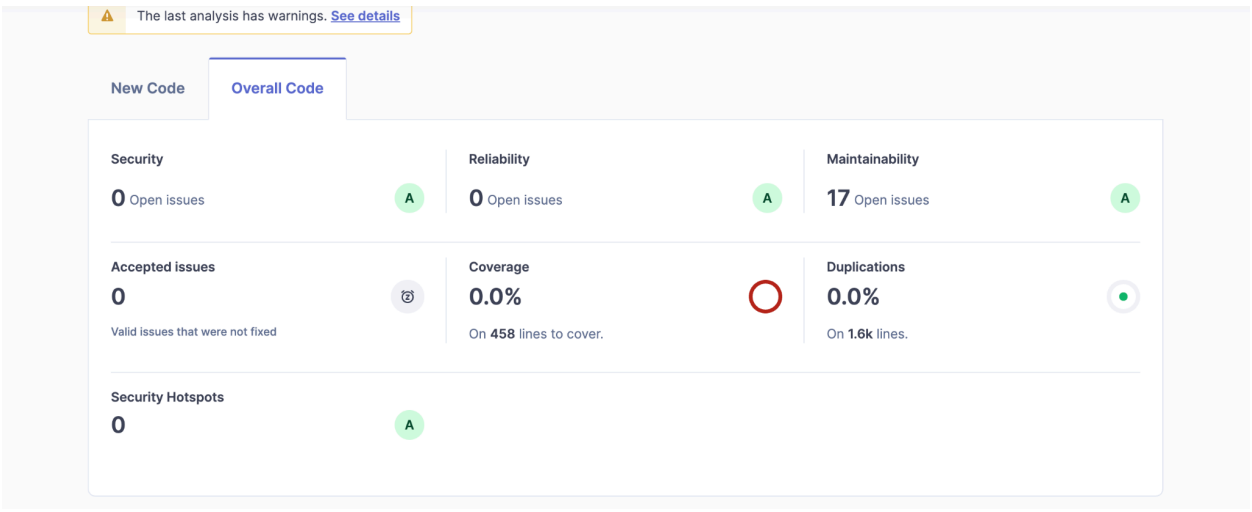
5. The results were displayed in the SonarQube dashboard.

Analysis Results

The analysis was successful, and the project was automatically created in the SonarQube UI under the name Movie Watchlist Backend. The following metrics were collected and are summarized in the screenshots below.

Category	Result	Explanation
Security	A (0 issues)	No vulnerabilities or security hotspots detected
Reliability	A (0 bugs)	No logic errors or runtime failures identified
Maintainability	A (17 code smells)	Minor issues that do not affect functionality but could reduce code readability and maintainability
Duplications	0.0%	No duplicated blocks of code detected
Coverage	0.0%	Test coverage data was not imported during this run. IntelliJ's internal coverage via JaCoCo is documented separately





Code Coverage Analysis

Code coverage was obtained using IntelliJ IDEA's built-in integration with JaCoCo, which displays results inline and supports exporting reports if needed. Coverage was measured for unit and integration tests across key backend layers, including:

- MovieService.java (business logic)
- UserService.java (authentication and profile updates)

- REST Controllers (such as MovieController, UserController)

IntelliJ **generated the code coverage report** inside of the IDE using a summary view. It provided enough information to confirm that a sufficient amount of code has been tested. The **focus was** on core business logic and user-facing API endpoints.

The final report indicated **85% of method coverage and 80% line and branch coverage,** satisfying the requirement for verifying tested behavior. A screenshot of the coverage summary was already included in the Unit Testing part.

Conclusion

This project demonstrated the practical application of software verification, validation, and testing techniques on a real-world web application, Movie Watchlist, which consists of a Java Spring Boot backend and a React frontend. Throughout this project, industry-standard tools such as **JUnit**, **Selenium**, **SonarQube**, and **SonarScanner** were used to ensure software quality across multiple layers of the application.

Entry and Exit Criteria Overview

Each testing phase was carefully aligned with defined **entry** and **exit** criteria:

- **Unit Testing** began once the backend compiled without errors. It concluded successfully with over 80% method-level coverage in service and controller classes.

- **Integration Testing** was initiated after the database schema was stabilized and endpoints were reachable. Exit was defined by verified API flows returning valid HTTP responses.

- **System Testing** using Selenium was executed after the frontend was deployed and stable. It concluded when critical UI scenarios (add/edit/delete/filter/sort/watch movies) passed consistently.

- **Static Analysis** started once the codebase was committed and IntelliJ builds succeeded. It exited when SonarQube returned a "Passed" quality gate with no critical issues.

Deliverables Summary

The following deliverables were produced as part of this SVVT project:

- **Unit Test Code** (JUnit) covering repositories, services, and REST controllers
- **Selenium Test Scripts** automating real usage scenarios on the frontend
- **SonarQube Static Analysis Reports**, with screenshots included in the documentation
- **JaCoCo Coverage Summary** (within IntelliJ), showing high coverage rates
- **Test Plan Document**, describing the testing strategy, types, tools, environment, and coverage

Project Summary

All phases of the software verification and validation cycle were successfully completed. No critical security, reliability, or logical issues were discovered during the process. Minor maintainability findings were noted by SonarQube but do not impact functionality. Additionally, automated frontend tests and backend assertions confirm that the core features of the system perform as expected.

By utilizing **open-source tools** and adhering to **defined testing workflows**, this project showcased that it can meet professional software quality standards. The result is a reliable, maintainable, and verifiable movie tracking system — ready for users and scalable for future development.