

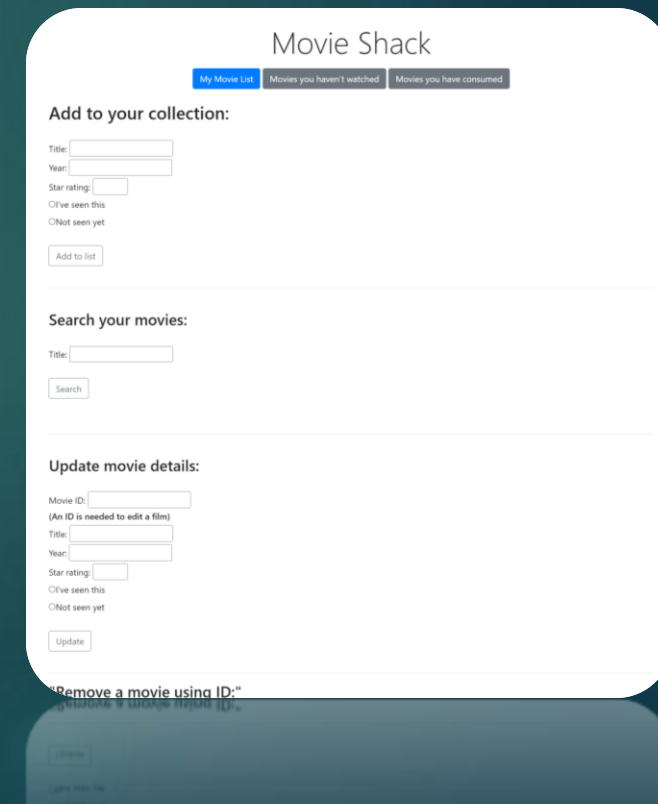


# Movie Shack

FULL STACK PROJECT BY AARON KARIM

# How does my project work?

- ▶ The Movie Shack allows users to add movies, see all of their added movies, sort them by year of release or their rating, update them, remove them, and sort them by whether the user has seen them yet.



The screenshot displays the 'Movie Shack' web application. At the top, the title 'Movie Shack' is centered, with three navigation tabs below it: 'My Movie List' (highlighted in blue), 'Movies you haven't watched', and 'Movies you have consumed'. The main content area is divided into three sections: 1. 'Add to your collection:' featuring input fields for 'Title', 'Year', and 'Star rating', radio buttons for 'I've seen this' and 'Not seen yet', and an 'Add to list' button. 2. 'Search your movies:' featuring a 'Title' input field and a 'Search' button. 3. 'Update movie details:' featuring a 'Movie ID' input field with a note '(An ID is needed to edit a film)', and input fields for 'Title', 'Year', and 'Star rating', along with 'I've seen this' and 'Not seen yet' radio buttons, and an 'Update' button. At the bottom, there is a partially visible section titled 'Remove a movie using ID:'.

# How I approached the project at the beginning

- ▶ About me
- ▶ I approached the project with the MVP in mind. I had ideas about extended features but my main priority was to hit all of the project requirements first.
- ▶ The order of my process went as follows:
  - ▶ 1) I created a database with no tables with MySQL workbench.
  - ▶ 2) I created the spring boot side in its entirety using Eclipse.
  - ▶ 3) And then moved on to the front end, working on the JS and HTML simultaneously in VS Code.

# How did I approach the problem?

- ▶ I began with creating a single entity (movies). This contained four attributes: 1) Title, 2) Year of release 3) Rating, 4) and a seen or unseen Boolean.
- ▶ Creating the basic CRUD functionality was done quite quickly before moving on to custom SQL queries for searching the database by year or rating for example. To do this, I connected spring boot to the database via the application properties file.
- ▶ From the outset, I was testing persistence using postman as an indicator of functionality as well as checking the database itself.

```
package com.qa.baetraining.domain;

import java.util.Objects;

@Entity
public class Movie {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "movieTitle", length = 50, nullable = false)
    private String movieTitle;

    @Column
    private int releaseYear;

    @Column
    private int rating; //PRIVATE NO NO

    @Column(nullable = false)
    private boolean seen;

    //-----

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getMovieTitle() {
        return movieTitle;
    }

    public void setMovieTitle(String movieTitle) {
        this.movieTitle = movieTitle;
    }

    public int getReleaseYear() {
        return releaseYear;
    }

    public void setReleaseYear(int releaseYear) {
        this.releaseYear = releaseYear;
    }

    public int getRating() {
        return rating;
    }

    public void setRating(int rating) {
        this.rating = rating;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Movie movie = (Movie) o;
        return id == movie.id;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

# How did I approach the problem (cont.)

To allow users to interact with the database, I needed an easily usable front end that was thorough but not overly convoluted.

For this, I decided to use forms on the main page for features that included searching for a particular attribute (year, rating, title, etc), and buttons at the top for when the user wants to search by a Boolean (seen or unseen movies), and a simple movie list button for viewing all the movies.

Any output would display in raw JSON at the bottom of the page.

# What did I hope to achieve?

- ▶ A pleasant, efficient front end capable of persisting features to the back end for storage and retrieval.
- ▶ Features that were useful for the user and not overly complicated in their layout.
- ▶ Controller, service, and repo classes capable of communicating between each other as well as the front end and database.

# What was needed for this project and what did I learn?

- ▶ For this project, I was required to re-familiarise myself with Spring Boot and learn how to test in a different way to what I had learned in the past.
- ▶ I had to understand how HTML and JavaScript work as well as how they interact with each other.
- ▶ I learned how to test using a H2 database via 'active profile' selection.
- ▶ I learned how to diagnose problems using 'inspect' in the browser.
- ▶ Axios was a great tool due to its ability to persist data in JSON format which allowed for simple code usable across the application.



# How I handled version control

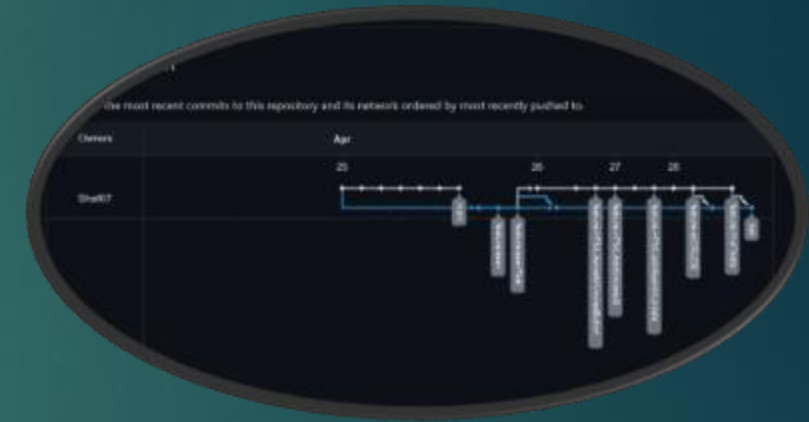
Using Github, I originally pushed (via Git Bash) the basic CRUD functions of my spring boot application to the dev branch of my repository.

From there, I added features to independent branches and then conducted some tests before merging to the dev branch.

When I began the front end, I stored these files within my repo directory. These files would be pushed using the feature branch model but each new feature would not merge into dev.

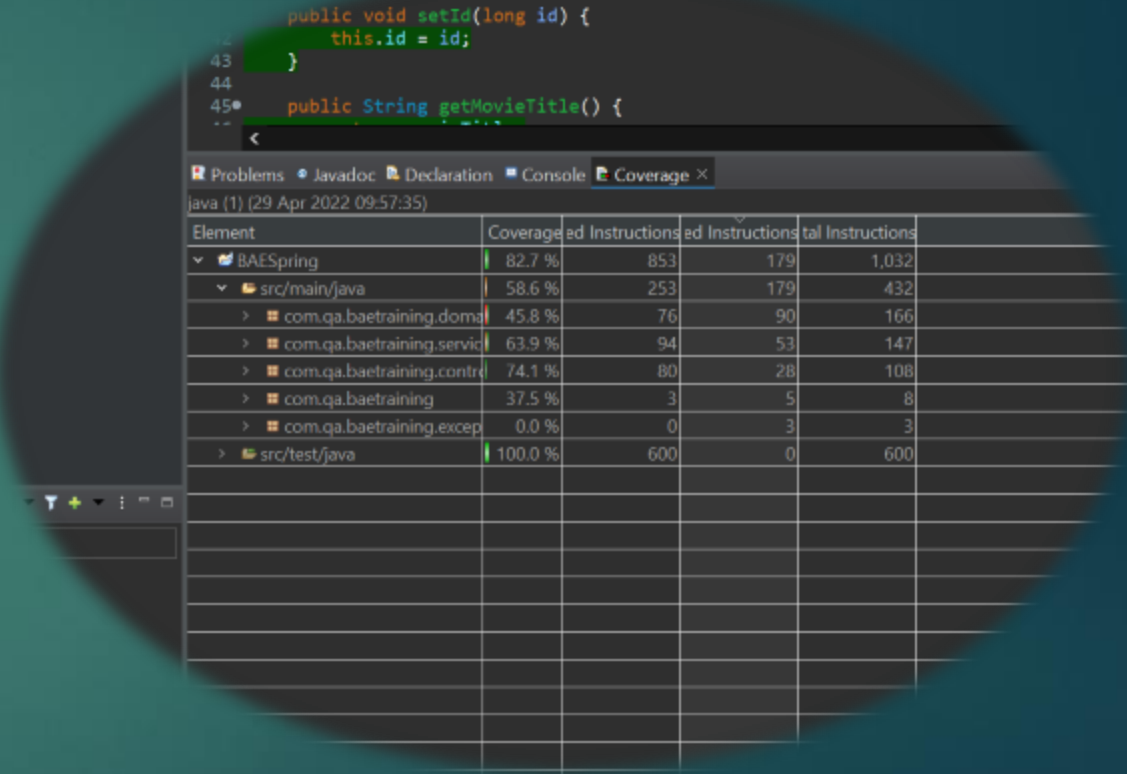
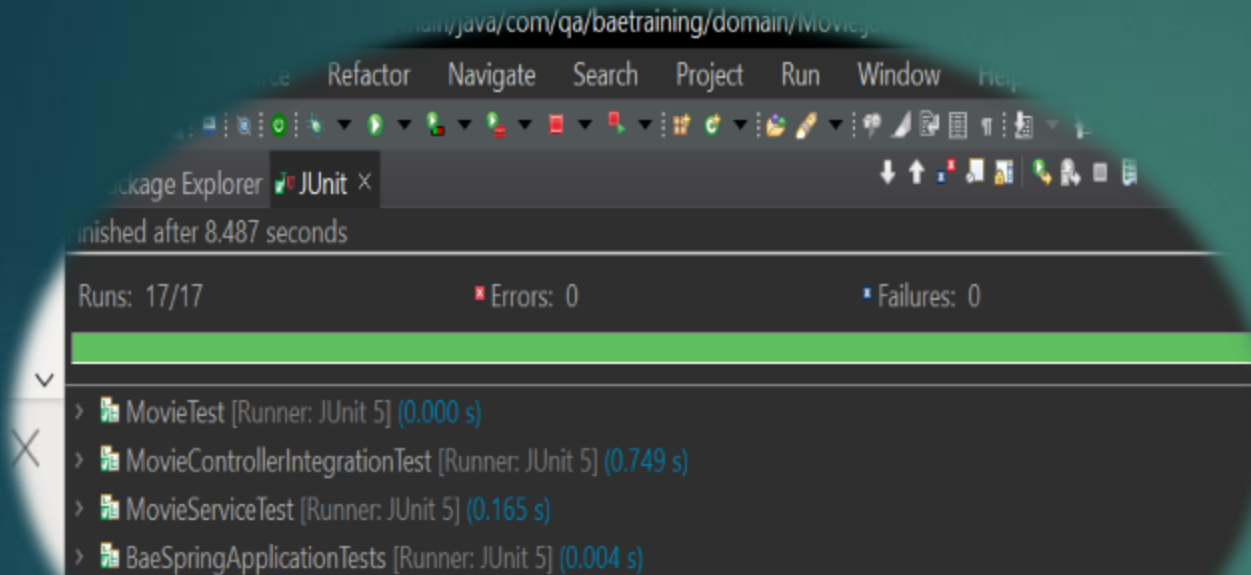
Once I achieved full functionality from my front end, I merged this to the dev branch.

I created one final feature that was to be used for testing. My previous tests only covered 49% of the main file but my final feature would be merged back into the dev branch at 58.6% coverage.





# Coverage screenshots



# Testing

- ▶ I began with testing my controller class using the H2 database. To manufacture the build structure required to support the information being persisted, I created a separate sql file containing a query that would build the table needed within the H2 database.
- ▶ This integration test would test all CRUD functionality using the MockMvc import. This would allow me to start an 'in-memory' container to check if the methods were functional.
- ▶ I then used Mockito to test whether the service methods were integrating with the repo methods to achieve the right results.
- ▶ I then conducted an 'Assert equals' test on the domain class to ensure that the constructor that I used was also functional.
- ▶ 58.6% test coverage achieved using JUnit test at time of this presentation.

# What was completed

- ▶ I managed to complete all of the intended outcomes which pertained to the MVP.
- ▶ I did not plan for extra features as I did not anticipate having extra time once I accounted for setbacks in time.

# What is not completed?

- ▶ I had a feature in the backend to update a movie's viewing status as I believed that this could be done via the update movie form, and thought it would be better suited for movie output that can be interacted with and not just a raw text format.
- ▶ Although I did not plan to experiment with front end designs, I expect I would have done given enough time at the end however this isn't / wasn't feasible due to time constraints.
- ▶ I am yet to finish testing to achieve 80% coverage (currently at 60%), and I am yet to compile the application into a jar file.

# What went well?

- ▶ I was able to compile a functional back end within the first day which left me plenty of time to work on the front end as this was completely new to me.
- ▶ I was able to implement all the features that I had initially planned.
- ▶ The webpage design is easy to navigate and use.

# What did not go well

- ▶ When I was creating the JavaScript functions, I had failed to recognise that tuning the HTML file to go to a specific URL when a button is clicked will override the JavaScript functions about what to do when the button was clicked. This set me back a few hours as I continued to get errors.
- ▶ After I had completed the project (minus complete testing), I turned off my computer for a break. When I came back, Eclipse would no longer load my project. The folders within the project directory had become jumbled and files displaced. I reorganised all the folders but to no avail. It took me two hours to learn that the metadata had been corrupted and that deleting it would solve the problem.
- ▶ It took me a couple of hours to figure out how to direct to a URL based on user input, which I found out was due to me appending the input to the URL in html format. Appending .value stopped this problem, but it took me a couple of hours to sort this out.
- ▶ At the beginning, it became a recurring theme where my HTML file would not detect the JS file however this resolved on its own.

# My reflections

- ▶ This project was based on a derivative idea I have which involves displaying all Netflix content and being able to sort it by IMDb ratings (hence the ratings and viewed options).
- ▶ This has been done on other sites however many movies/content have high ratings simply because they have very few ratings. The higher the quantity of ratings, the more accurate the rating is itself.
- ▶ My website would eliminate any content that had less than 5000 reviews as this would be more accurate and content sub 5000 ratings tend to have less production value.
- ▶ The CRUD functions I have implemented are fully functional and a few can be implemented into this future project.



# Future projects

- ▶ To do this, I will have to learn how to extract data from other sites and automate them to update a database semi regularly. This would not be done on demand in the user-end due to it being slow.
- ▶ I will need to improve my use of JavaScript, HTML and make use of CSS which I haven't done in this project.