Semester Project Final Submission

CSCI 5448

Title: Movie Tracker

Team Members: Xinyu Jiang, Qiuyang Wang, Vladimir Zhdanov

Final State of System

The final state of the system is essentially what we expected to build when planning this project. We implemented all of the features which we defined in Project 4. In Project 4, we defined the following requirements for this project:

- This system must be a web application which can respond to user input.
- The system must be able to keep track of user accounts, along with associated movie profile data for each user.
- The system must have access to specific data about movies such as titles, runtime, and rating.
- The user must be able to add a movie to their list.
- The user must be able to search for a movie to add to their list.
- The user must be able rate and review movies added to their list.
- The user is able to go to their profile to check the movies that they have added and view statistics about the movies on their list.
- The user can view others' profiles.

All of these requirements are met by our final application.

There were some features which changed from our original design in Project 4. Our website UI changed a bit from the initial UI mockups we designed. We added a navigation bar, and changed the layout of some of the pages of the application. In planning, we also weren't sure how users would be able to view others' profiles. In our final application, we decided to design a user leaderboard page which shows the users with the most movies and watchtime. From this page, a user is able to view others' profiles easily and effectively. We also had some design pattern changes, which will be discussed in the next section.

Final Class Diagram and Comparison

The final class diagram and the class diagram from Project 4 are appended to the end of this PDF. In the final design of our system, we included the following design patterns:

1) *Adapter* -

   The adapter pattern is used to connect the external OMDb API into something that can be used in our system. The MovieAdapter class acts as an adapter between the external API and the MovieController in our system. By implementing this in this way, if the OMDb API ever changes, we will be able to easily update the system without making major code changes throughout the application.

2) *Data Access Object (DAO)* -

The DAO pattern is used to decouple the controllers from the database. Whenever an API call is made and passed to the controller, the controller will call on a DAO to manipulate the data in the database. This allows the controllers to be loosely coupled from the database. And, if the database architecture ever changes, it will be a much smaller maintenance effort to update the system.

3) *Model-View-Controller (MVC)* -

The MVC pattern is used to connect the user interface (the view), the various API controllers in the system, and the database (the model). The controllers act as a translator between the database and the UI. When data is changed on the frontend, the frontend will call the API, which will call the controller to use a DAO to update the information in the database. Likewise, each time a page is loaded on the frontend, the latest information is pulled from the database by the DAO, passed to the controller and reflected on the frontend. From this, our data is kept in sync between all parts of the application.

Comparing the final class diagram to that of Project 4, we see that there are quite a few changes. First, we removed the observer pattern which we planned to use between the MovieProfileController and UserController classes. During development, we realized that the relationship between these two classes was not really that of an observer pattern, as the UserController just needed to call some of the methods in the MovieProfileController. Instead, we decided it would be easier to use dependency injection to inject an instance of the MovieProfileController, and call it in this way

We also added a DAO design pattern, which was not present in our original design. The purpose of this pattern is to decouple the controller classes from the database. In our original design, the controllers were tightly coupled with the database, which is not a good practice to have. So, we implemented a DAO layer to access the database, and to allow for loose coupling of our system.

On the frontend of the system, we also decoupled the API from the components. We built a separate service layer for all of the logic related to API calls (in the MovieService.js, MovieProfileService.js, and UserService.js files). So, if the API ever changes, only the service files will need to be updated, rather than every component which uses them.

Since we are using Flask to route API calls, we also implemented the use of the Flask Injector. With this, we are able to directly inject singleton instances of each of our dependencies into the classes that need to use them. This also helps decouple the system, and makes sure there is only one instance of each of the controllers and DAOs over the runtime of the application.

Third-Party Code vs. Original Code

For this project, we used several external sources:

- The base React app was bootstrapped from Facebook's Create React App repository. This allows for a user to create a simple React app with no build configuration.

- Using this base app, we followed this tutorial by Miguel Grinberg to get a Flask server running and interacting with a React frontend. A lot of the build files and configurations in our repository are from this tutorial.
- We used the OMDb API as an external API to fetch movie details and movie search information.

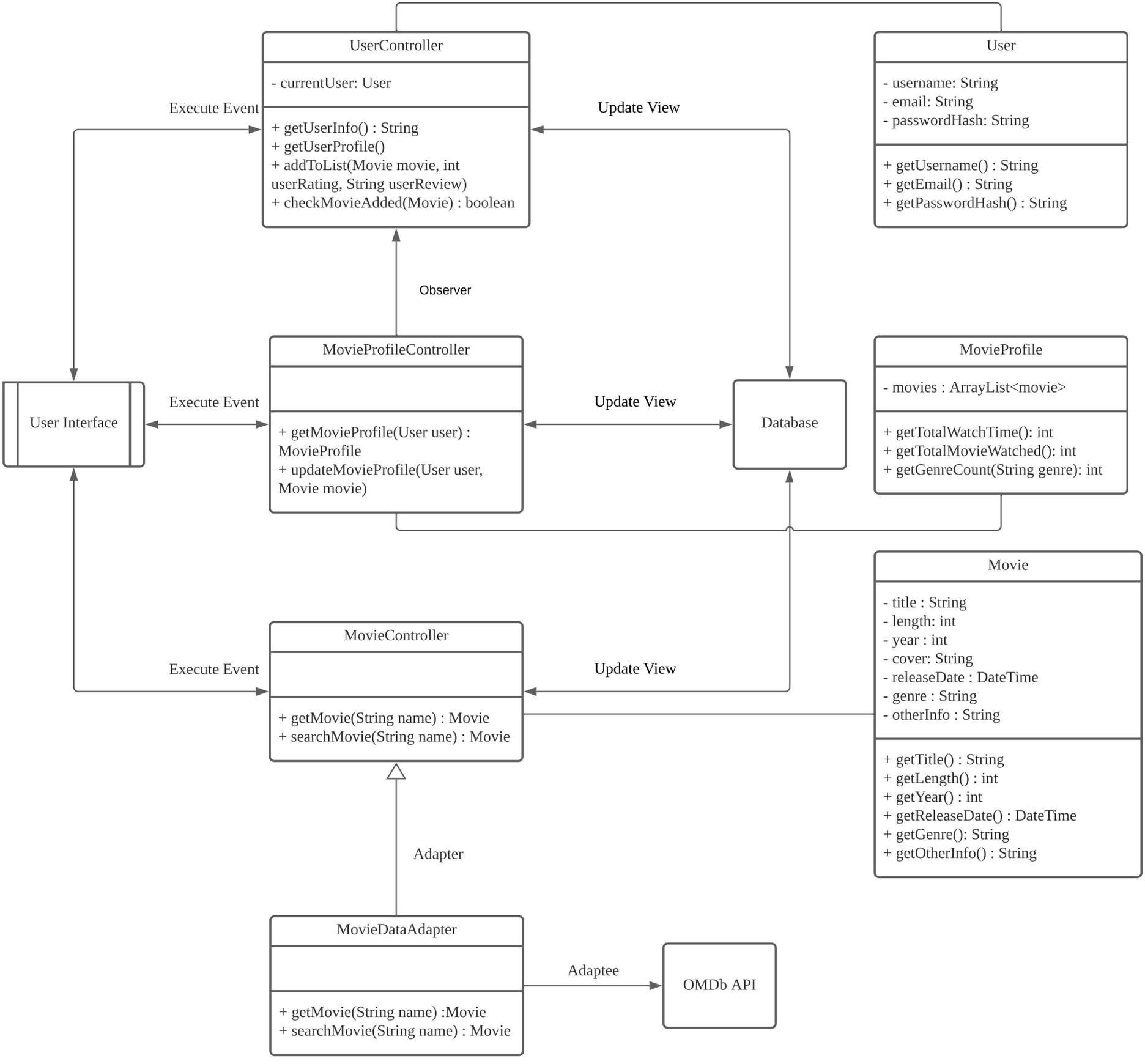
All of the remaining code in the project is original code. All of the backend code (DAOs, controllers, the movie API adapter, etc), all of the Flask API logic, and all of the frontend pages and components were created by us for the purpose of this project.

OOAD Process for Overall Semester Project

The following are three key design process elements that our team experienced over the course of analysis and design of the OO semester project:

1) We learned the importance of loose coupling in our system. Initially, we did not plan to use DAOs to decouple the database from our controllers. We figured it would be fine to leave the database logic in the controllers since this isn't a very large, enterprise-level, project. But, in our initial code review, it was suggested that our database would be tightly coupled with the entire application, so we took the steps to implement the DAO pattern in our project.
2) It was nice to see how OO principles and patterns helped clean up our code. The patterns we used made our code easier to read, have better overall structure, and made it easier to make changes to large portions of the system.
3) We appreciated how dependency injection helped simplify our code. We used the flask injector to implement singleton dependency injection. By doing this, during the implementation period, we were able to easily inject an object into a class or method knowing that the total number of instances of each object would be controlled.

Project 4 Class Diagram



Movie Tracker Final UML Class Diagram