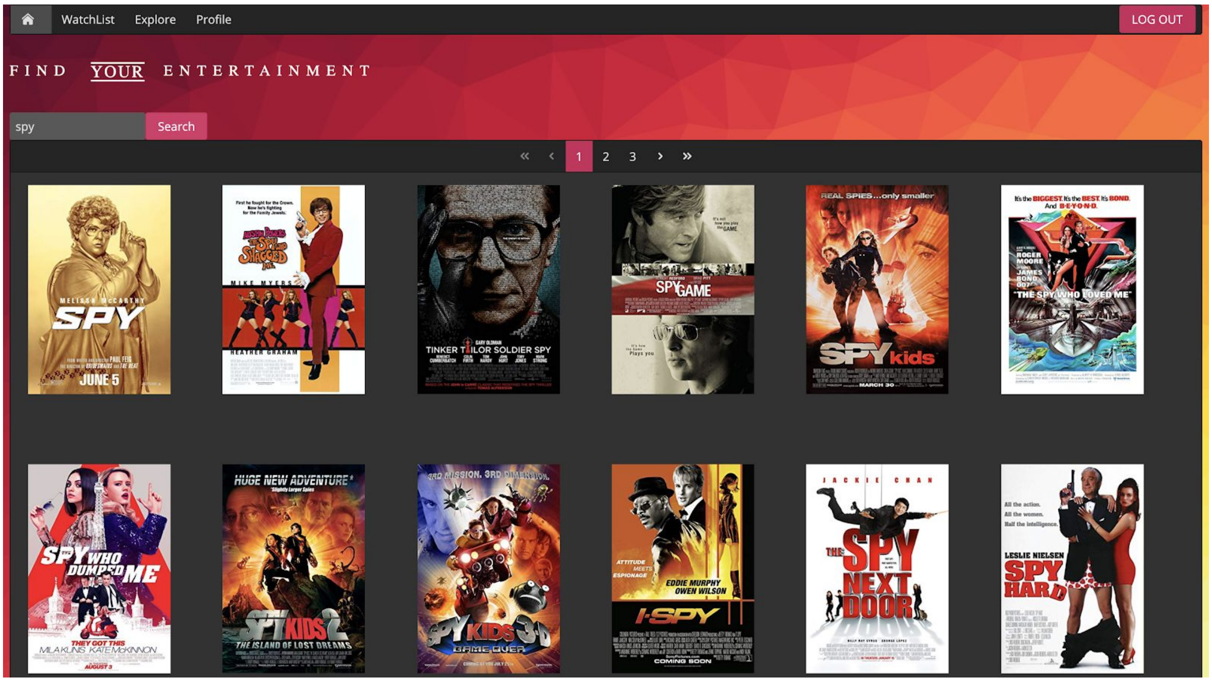
FlickTier

- Save and explore video entertainment



<p:by> Pontus Backman, Hugo Cliffordson, Edwin Eliasson, Benjamin Vinnerholt, Oskar Wallgren </p:by>

1. Introduction

This report is a description of the project conducted in the course *Web Applications (DAT076)* at Chalmers University Of Technology. The project is a web application used for finding and saving video entertainment. The idea of the application is to build a personalized watchlist in which you save video entertainment in the form of movies and series in order to plan future video entertainment. This is done by searching for video entertainment and by exploring friends and other users' profiles. The goal is a social media platform for video entertainment.

Further development of the application could be a more comprehensive tool to save, categorize, rate and explore video entertainment.

Link to git repo

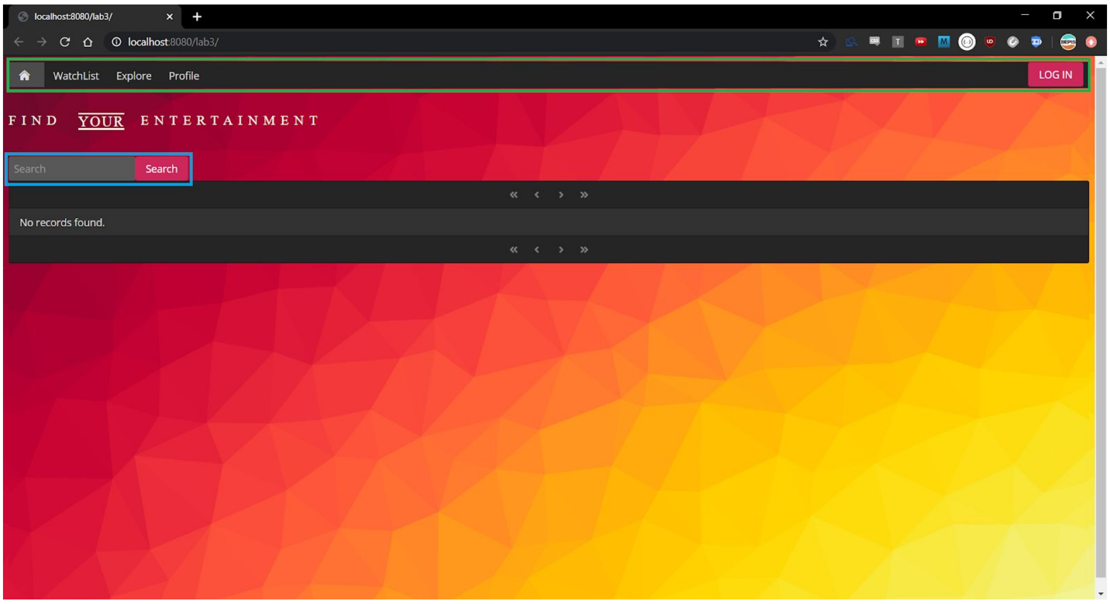
https://github.com/itggot-edwin-eliasson/Webapplikation-boras

1.1. List of Use Cases

- As a user,
    - I want to create an account
    - Log in to an account
    - Log out from an account
    - Search for films
    - Save films to my watchlist
    - View films in my watchlist
    - Remove films watchlist
    - View my profile
    - Edit my profile information
    - Add profile picture
    - Search for other user's watchlist
    - Add films to my watchlist from other user's watchlist
    - Add other user's entire watchlist to my watchlist
    - See other users profile

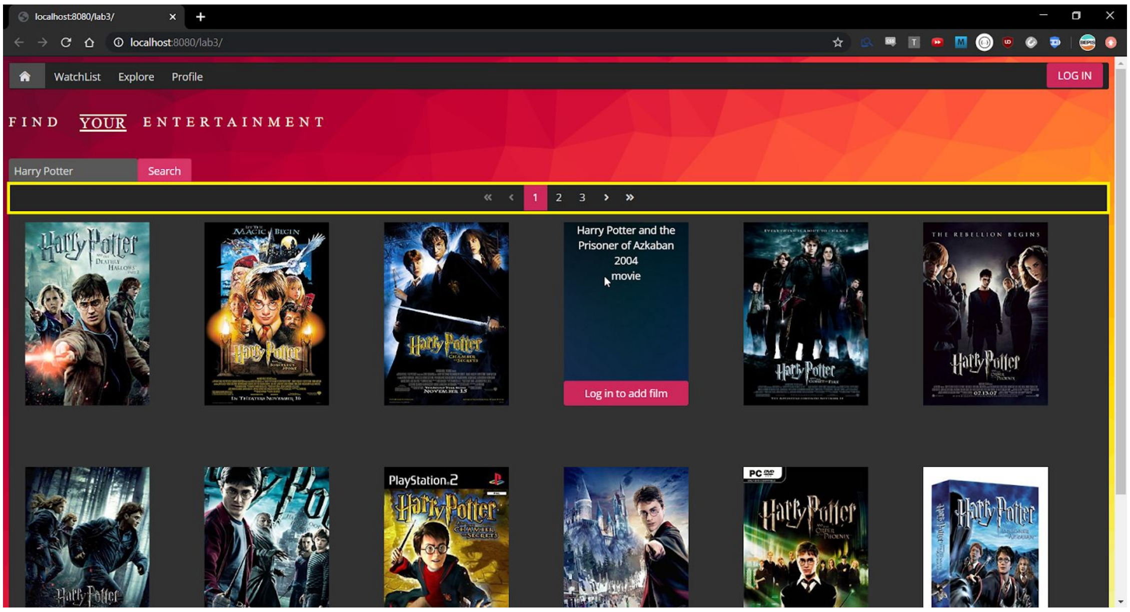
2. User Manual

The first page the user sees when entering the website is the home page. The main contents of the home page are the navigation bar (marked in green), which lets you navigate to the different pages of the site, and the search box (marked in blue) with corresponding search results to be found below. The home page can always be reached by pressing the home icon to the very left in the navigation bar.



No records found.

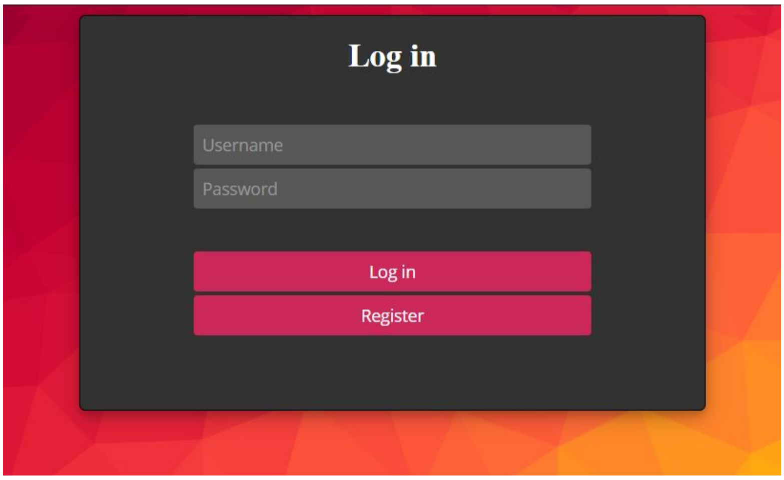
To start looking for films to add to your watchlist, simply enter the title of your desired film into the search box and press the “Search” button (or enter). Let’s search for some Harry Potter films.
All films matching your search will now be displayed in the grid below the search box.



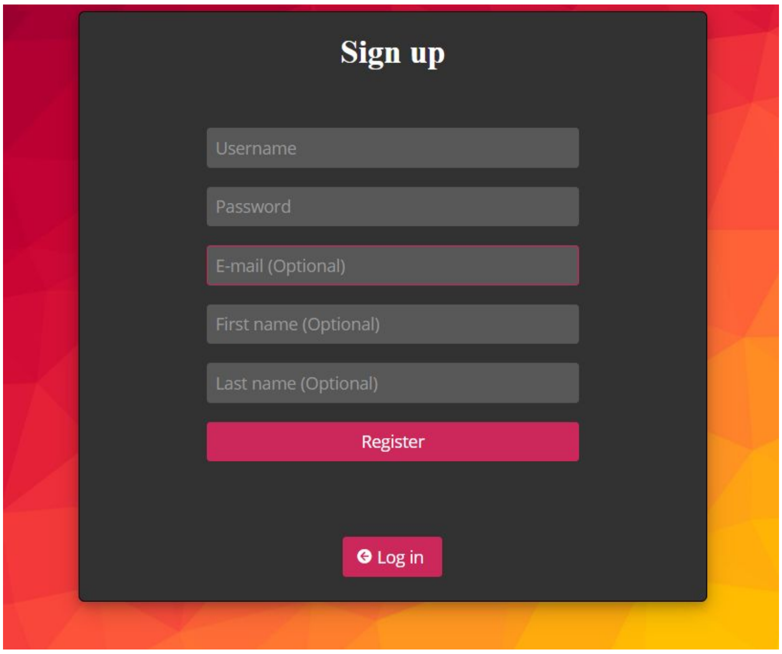
Harry Potter and the Prisoner of Azkaban
2004
movie

The paginator (marked in yellow) can be used to display more films matching your search.

When you hover over the films in the grid, the cards will flip and display some basic information about the film. To add a film to your watchlist you first have to log in, so let’s do that. Click either the “Log in to add film” button on the flipped card or the “LOG IN” button in the top right of the page.



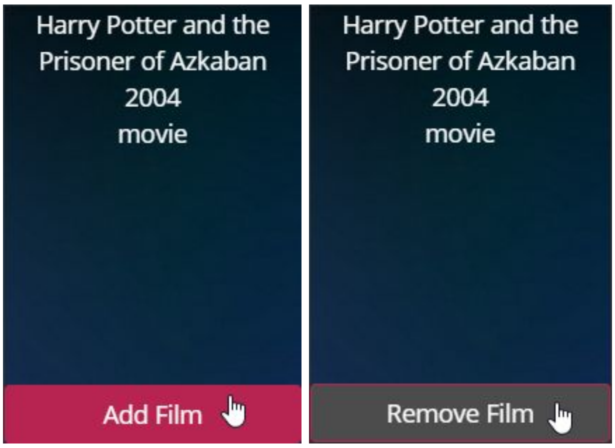
On the login page you can choose between logging in with an existing account or registering a new one. If you have already registered an account, log in using that, otherwise click the “Register” button to create your account.



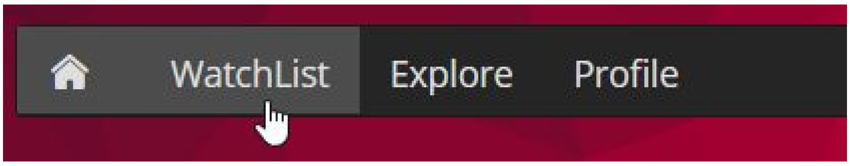
To register, enter your desired username and password and click the “Register” button.

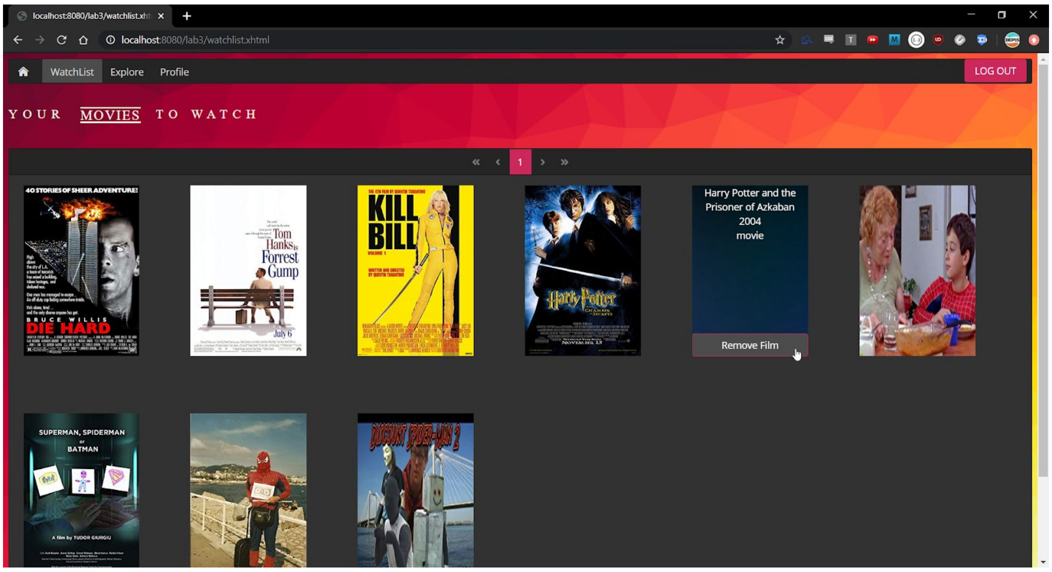
If you would like for your friends to be able to search for your watchlist using your name or e-mail in the future, enter those as well. If you’re not sure, you can leave them blank and edit them later.

You are now logged in and back at the home page. If you search for some films you are now able to add them to your watchlist. Simply hover over the card for it to flip, and then press the “Add film” button. Once you have pressed the button it will change color and will now say “Remove film”. The film is now added to your watchlist.


Harry Potter and the Prisoner of Azkaban
2004
movie
Add Film
Harry Potter and the Prisoner of Azkaban
2004
movie
Remove Film

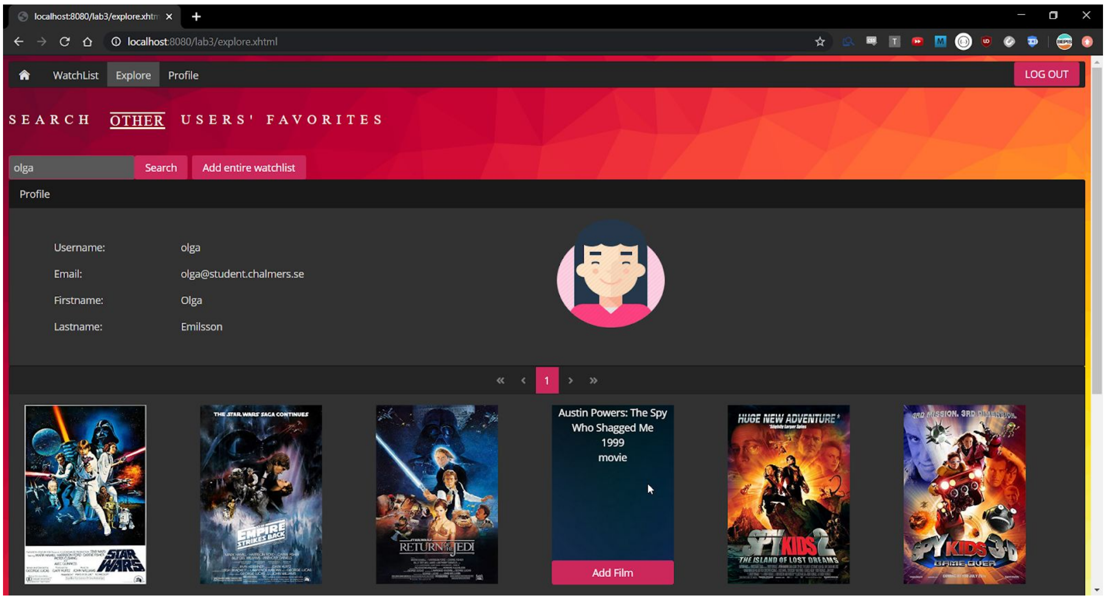
To view your watchlist simply press “WatchList” in the navigation bar at the top of the page.




localhost:8080/lab3/watchlist.xhtml
WatchList
Explore
Profile
LOG OUT
YOUR MOVIES TO WATCH
1
Harry Potter and the Prisoner of Azkaban
2004
movie
Remove Film

Your watchlist is displayed similarly to the home page, and you can remove films from your watchlist in the same manner - by hovering over the film and pressing “Remove Film”.

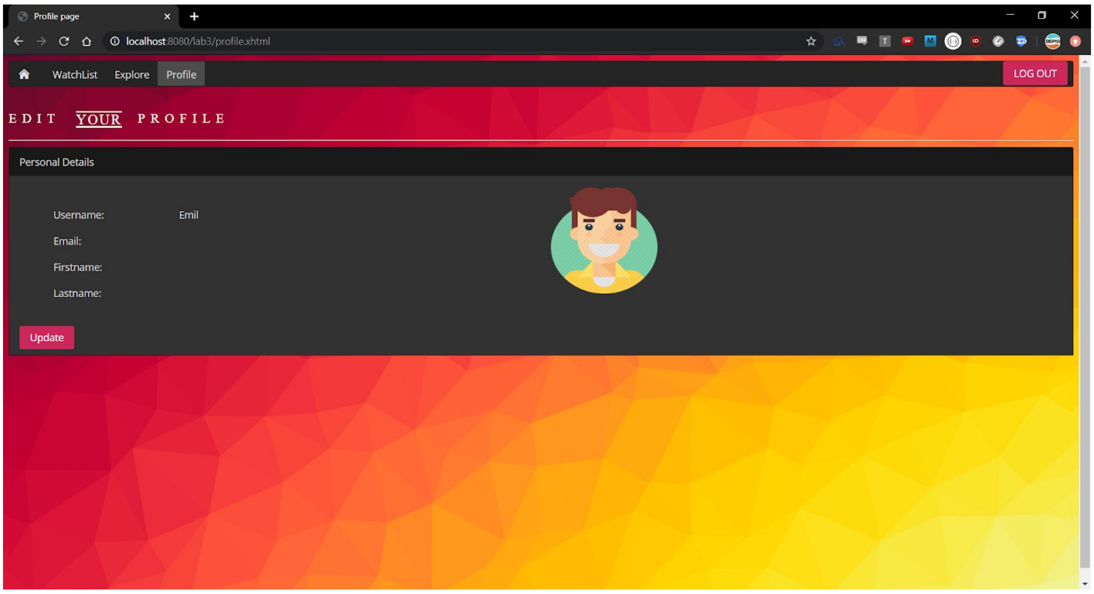
Now let’s continue through the navigation bar. The next page is “Explore”. On the Explore page you can search for other users and view their watchlists.
Enter your friend’s username into the search box on the Explore page and hit “Search”.


localhost:8080/lab3/explore.xhtml
localhost:8080/lab3/explore.xhtml
WatchList
Explore
Profile
LOG OUT
SEARCH OTHER USERS' FAVORITES
olga
Search
Add entire watchlist
Profile
1
Austin Powers: The Spy Who Shagged Me
1999
movie
Add Film

Username: olga
Email: olga@student.chalmers.se
Firstname: Olga
Lastname: Emilsson

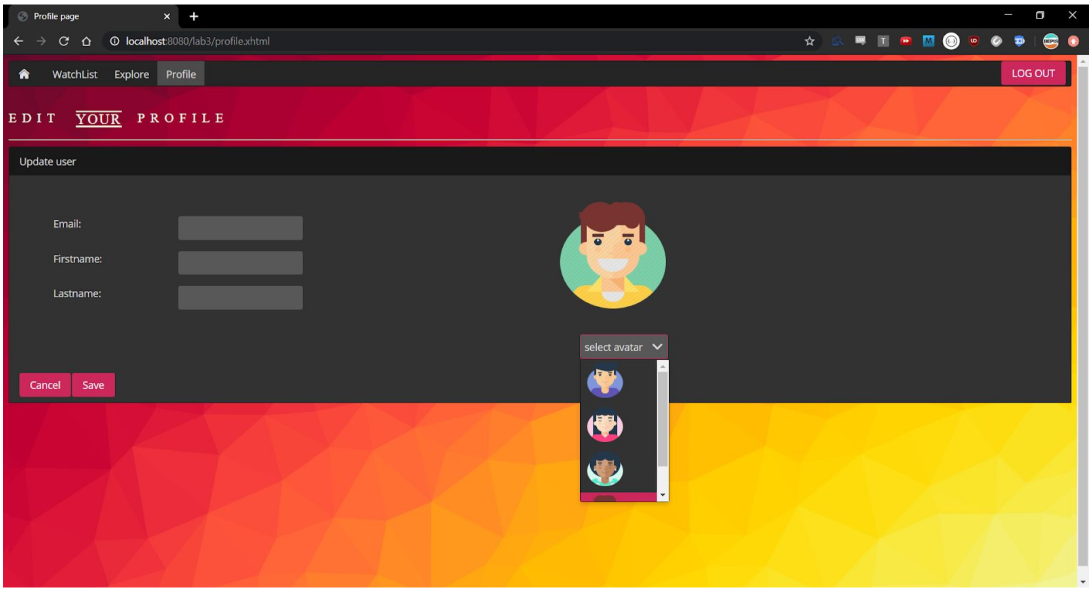
If your search matched any existing user, their profile and watchlist will be presented. As usual, you can hover over the film and add it to your own watchlist. You also have the choice to add their entire watchlist to your watchlist by pressing the “Add entire watchlist” button next to the search button.

Finally, the profile page.



Username: Emil
Email:
Firstname:
Lastname:

The profile page displays your personal information as well as your avatar. All details can be edited except for your username. To edit your information, press “Update”.



Now you can edit your information and choose between four preset avatars. To save your edits press the “Save” button.

3. Design

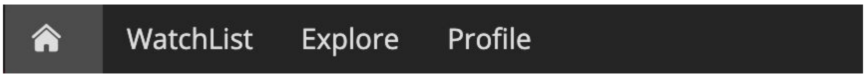
This section will describe the technical construction of the application. This includes all libraries, frameworks and other technologies used to complete the project. A number of technologies used were mandatory in the course and will therefore not be described in detail. However, the project specific technologies will be explained more thoroughly.

3.1. JSF

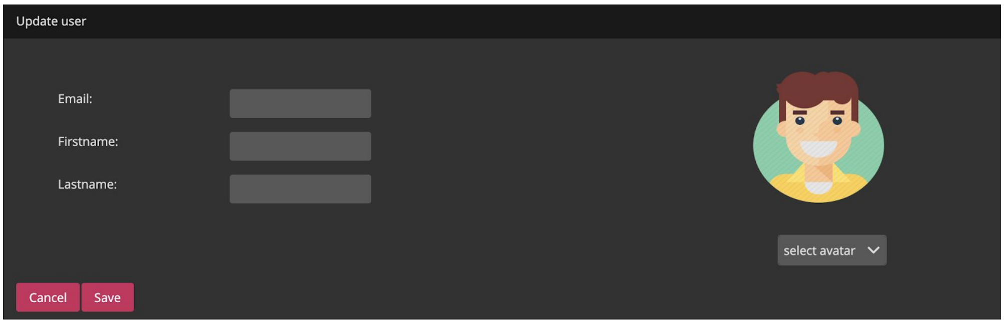
The group have used and successfully applied the software architectural pattern Model-View-Controller (MVC) at previous occasions. With the familiarity of the pattern and the knowledge of its benefits, the group chose to build the user interface to the application with the help of JavaServer Faces. JavaServer Faces (JSF) is the Java EE standard component-based web application framework. JSF’s core idea is to encapsulate functionality into reusable components. The architecture of JSF is built with a design focus on MVC. JSF components are backed by Java objects, which are independent of the HTML and have the full range of Java abilities, including accessing remote APIs and databases. A result of using JSF is that developers are allowed to build web interfaces without much interaction with technologies as: HTML, JavaScript and CSS. This was something that was noticeable while developing the application.

3.2. PrimeFaces

JavaServer Faces can easily be extended with component libraries. The component library used in this project, and the most notable in combination with JSF, is PrimeFaces. PrimeFaces is an open source UI component library. It features 100+ JSF UI components. PrimeFaces was used to rapidly build a “good looking” user interface. Below are some examples of PrimeFaces components used in the application.



Navbar From PrimeFaces



Panel from PrimeFaces

3.3. Lombok

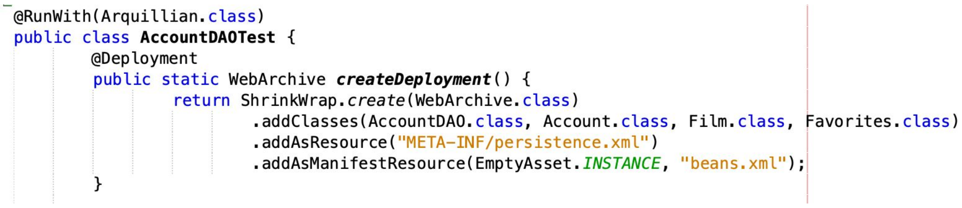
With the goal of writing code faster and better looking, the group used lombok in the development. Lombok is a Java library that plugs into the editor and build-tools, making the development more convenient. This is because Lombok can, as an example, remove the need to write setters and getters. By using an annotation your Class has a featured builder automating these processes.

3.4. jUnit

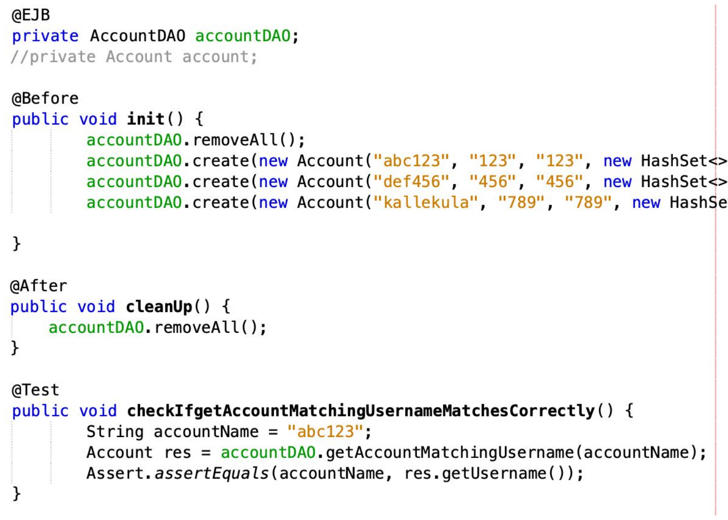
jUnit is a framework used to write unit tests in applications for the Java programming language. In the project, jUnit was used in combination with Arquillian, to write tests for the application.

3.5. Arquillian

Arquillian is a test suite that can be used together with testing frameworks such as TestNG and JUnit, allowing developers to run a complete test environment that supports all the features of an application server straight inside a unit test. In this course, we used Arquillian in the projects to allow us to write fully EE-aware JUnit and integration tests. Below is an example of how defining and testing with Arquillian looked like in our project.



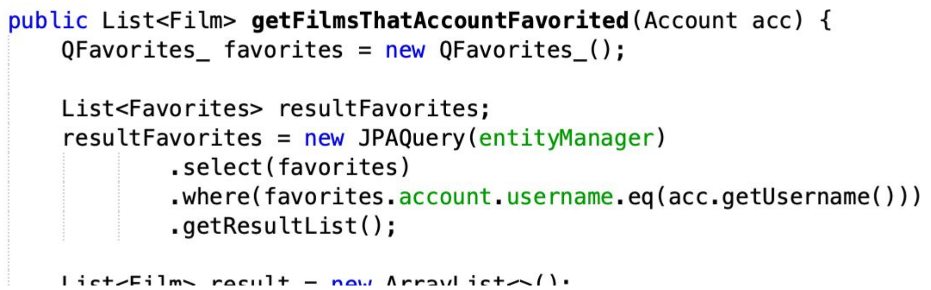
```
@RunWith(Arquillian.class)
public class AccountDAOTest {
        @Deployment
        public static WebArchive createDeployment() {
                return ShrinkWrap.create(WebArchive.class)
                        .addClasses(AccountDAO.class, Account.class, Film.class, Favorites.class)
                        .addAsResource("META-INF/persistence.xml")
                        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
        }
```



```
@EJB
private AccountDAO accountDAO;
//private Account account;

@Before
public void init() {
        accountDAO.removeAll();
        accountDAO.create(new Account("abc123", "123", "123", new HashSet<>()));
        accountDAO.create(new Account("def456", "456", "456", new HashSet<>()));
        accountDAO.create(new Account("kallekula", "789", "789", new HashSet<>()));

}

@After
public void cleanUp() {
    accountDAO.removeAll();
}

@Test
public void checkIfgetAccountMatchingUsernameMatchesCorrectly() {
        String accountName = "abc123";
        Account res = accountDAO.getAccountMatchingUsername(accountName);
        Assert.assertEquals(accountName, res.getUsername());
}
```

3.6. Easy Criteria

In order to define queries for entities we used JPA - Criteria API. And in order to make querying easier and more convenient we used Easy Criteria. This is a convenience layer on top of JPA Criteria API which allows to write queries in the following manner:

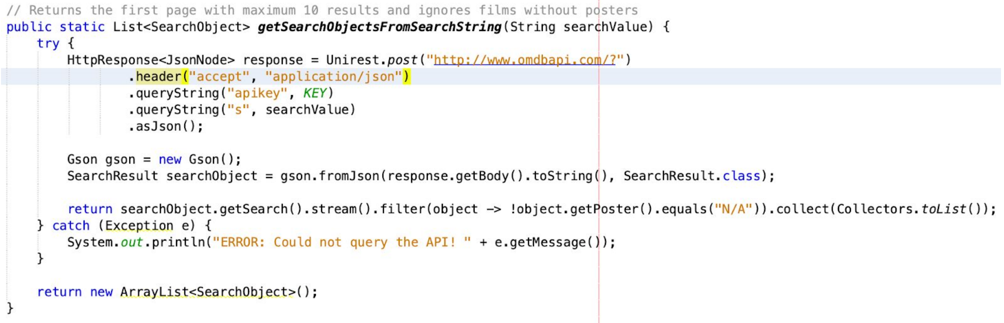


```
public List<Film> getFilmsThatAccountFavorited(Account acc) {
    QFavorites_ favorites = new QFavorites_();

    List<Favorites> resultFavorites;
    resultFavorites = new JPAQuery(entityManager)
            .select(favorites)
            .where(favorites.account.username.eq(acc.getUsername()))
            .getResultList();

    List<Film> result = new ArrayList<>();
```

Query with Easy Criteria

3.7. OMDb

OMDb API is a RESTful web service to obtain movie information. Since the application is about finding and saving movies you plan to watch, and finding other users' movies, using an api for movies was the best choice. The movie objects fetched from a search against the API were then stored in the application database. This way we could use the movie object as if they were our own objects, making it easier to build the rest of the project. Accessing the data from the database was also a faster option to get the information, instead of going through the OMDb api every time we needed it. Below is an example of the way information was retrieved from the api.



```
// Returns the first page with maximum 10 results and ignores films without posters
public static List<SearchObject> getSearchObjectsFromSearchString(String searchValue) {
    try {
        HttpResponse<JsonNode> response = Unirest.post("http://www.omdbapi.com/?")
                .header("accept", "application/json")
                .queryString("apikey", KEY)
                .queryString("s", searchValue)
                .asJson();

        Gson gson = new Gson();
        SearchResult searchObject = gson.fromJson(response.getBody().toString(), SearchResult.class);

        return searchObject.getSearch().stream().filter(object -> !object.getPoster().equals("N/A")).collect(Collectors.toList());
    } catch (Exception e) {
        System.out.println("ERROR: Could not query the API! " + e.getMessage());
    }

    return new ArrayList<SearchObject>();
}
```

3.8. Gson

Gson is a Java library used to serialize and deserialize Java objects from JSON. In the project, Gson was used to transform JSON data to Gson when fetching movies from the OMDb API. The Gson search objects were then transformed with java Stream to Java objects.

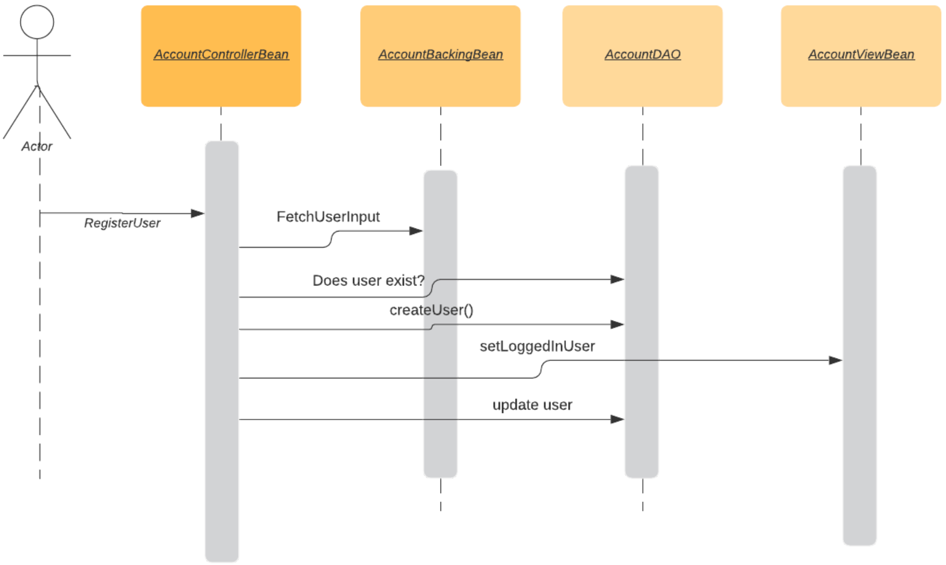
3.9. Unirest

Unirest is a lightweight HTTP client library used in the project. It provided chunks of code that helped do the basic things an application needs to do in order to interact with an API. In the figure under the section *OMDb,* you can see how Unirest was used.

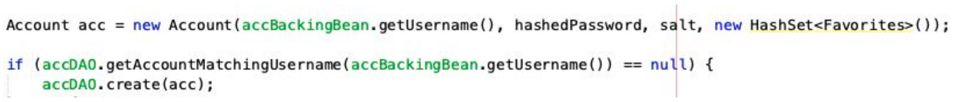
4. Application Flow

As described in previous chapters, this application follows a standard MVC design pattern that uses clearly separated responsibilities. To visualize the flow of the application, a few examples of how use cases are executed are presented.

4.1. Create an Account



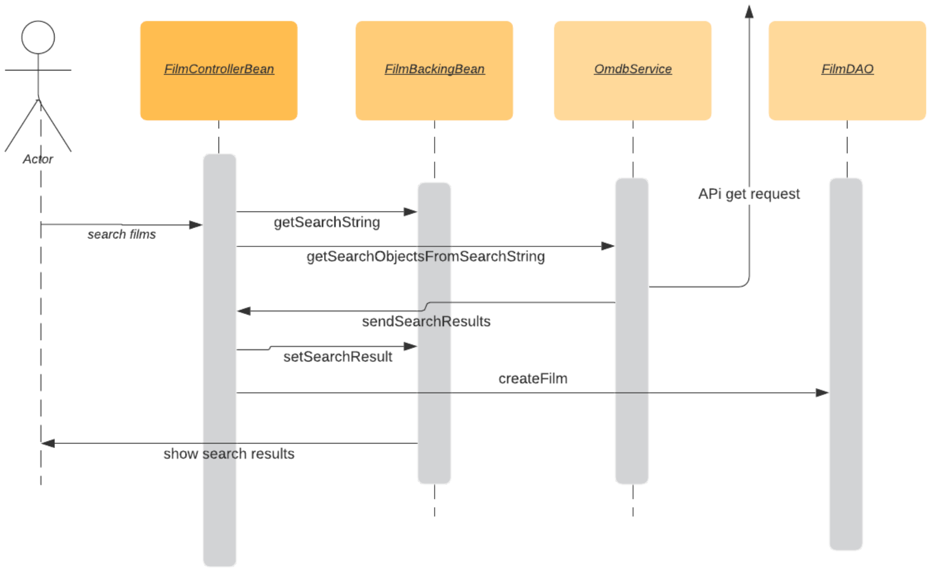
When a user wants to create an account, the view presents relevant fields for the user. These fields are variables in the model that can be set, some of them required. Pressing the register button, the AccountControllerBean method onRegister() is executed. The controller gathers all fields from the view and creates a new account.



```
Account acc = new Account(accBackingBean.getUsername(), hashedPassword, salt, new HashSet<Favorites>());

if (accDAO.getAccountMatchingUsername(accBackingBean.getUsername()) == null) {
    accDAO.create(acc);
```

Before the account is created, the controller checks for a few criteria. The username cannot already exist in the database and the password has to be hashed. This happens in the AccountDAO which uses JPAQuery to fetch information from the database. The AccountController also fetches the fields and calls each update method in the AccountDAO which updates the model. When the user is successfully created, the user is sent to the browse page.

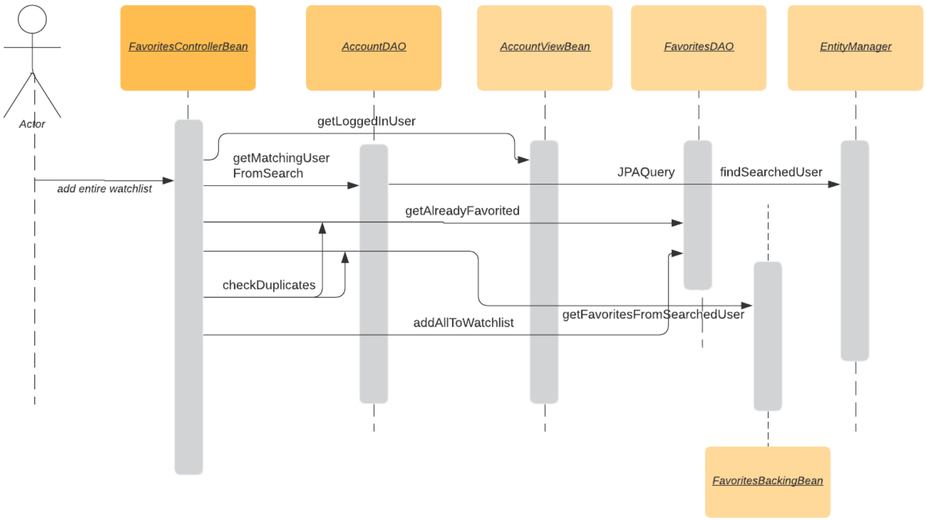
4.2. Search Films



Searching for films is done in the browse window. This is the landing page of the application and the user is given the possibility to use the search bar to search for films. Upon a search, the method onSearchFilms() in FilmControllerBean is called. This method fetches the searchString from the FilmBackingBean and uses it when calling on the Omdb API service. The service posts a get request with a query containing the searchString. This method returns a list with the search result. The controller then iterates through the search result and creates new Film entities with accompanying parameter, title, release year, type and film poster. Each film is created through the entity manager and added to the database. As happens, each film is added to a list in the FilmBackingBean which is sent to browse which updates the film grid.

4.3. Add entire watchlist to my watchlist

When exploring what other users have added to their watchlist, it is possible to add another user’s entire watchlist to your watchlist.



From explore, The FavoritesControllerBean calls the method addEntireWatchlistFromUser(). This method fetches the currently logged in user from the ViewBean and the searched user from the AccountDAO. When affected accounts have been set, FavoritesDAO is called to get films from the searched user and oneself. This fetches all films from the database with a JPAQuery and sends it back to the controller. The controller then compares films and updates the currently logged in users watchlist into a combined version of the two. As this has been executed, the data grid is updated accordingly.

5. Package Diagram

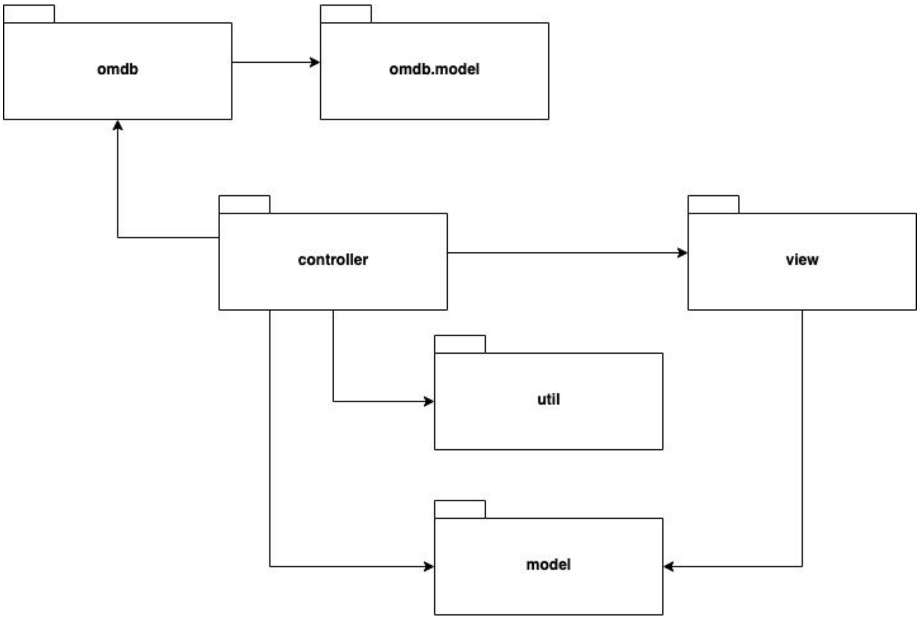
Our packages are mostly divided according to the pattern Model-View-Controller, MVC, with respective classes in the related package. The view package contains our backing beans that are responsible for holding and fetching data relevant to its name. For example, AccountBackingBean is responsible for holding temporary information like the username and password, as well as optional information of the user.

The controller package is where all the business logic is contained, where any modification to the data is executed. An example would be the AccountControllerBean that is used to log in, register and log out the user.

In the model package is where the data is managed. Our entity classes, Data Access Objects (DAOs) as well as a session bean are contained here.

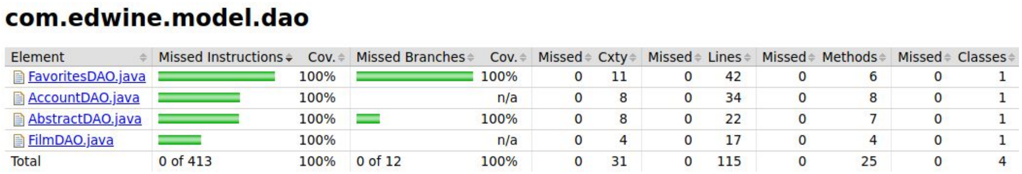
We also have a util package with a password hasher and salt generator that is used by AccountControllerBean when registering and logging in a user as to not have their passwords available as a string in case of a data breach.

One of the irregularities in our structures would be the omdb and omdb.model packages. These contain classes used when contacting the OMDb API we have implemented in our project. In the omdb package is the class that is used to contact the API, where as the omdb.model contains classes used to break down the information fetched into a more manageable way with classes like, searchObject, searchResult and filmObject



UML diagram for the packages of the project.

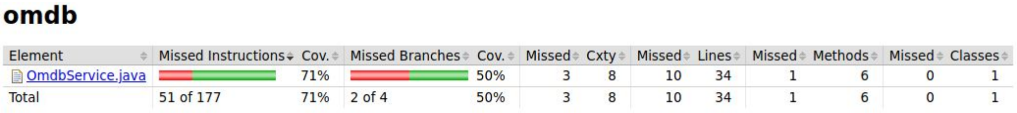
6. Code Coverage Report



com.edwine.model.dao

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
FavoritesDAO.java	🟩	100%	🟩	100%	0	11	0	42	0	6	0	1
AccountDAO.java	🟩	100%		n/a	0	8	0	34	0	8	0	1
AbstractDAO.java	🟩	100%	🟩	100%	0	8	0	22	0	7	0	1
FilmDAO.java	🟩	100%		n/a	0	4	0	17	0	4	0	1
Total	0 of 413	100%	0 of 12	100%	0	31	0	115	0	25	0	4

This is the code coverage report for the DAO-package.



omdb

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
OmdbService.java	🟥🟩	71%	🟥🟩	50%	3	8	10	34	1	6	0	1
Total	51 of 177	71%	2 of 4	50%	3	8	10	34	1	6	0	1

This is the code coverage report for the OMDB-package.

These reports were both generated by JaCoCo with default settings.

The lines that are not covered in OmdbService.java are only reachable when the OMDB-API does not respond, therefore they are hard to cover in a test.

These are the only two packages we test since the entity-package does not contain anything worth testing on its own. Failing entity classes should be caught in the DAO-tests. The DAO-classes are being tested with the help of Arquillian.

Everything bean, and frontend related is not tested automatically because of our inability to properly test it. We agreed that learning how to properly mock sessions and everything else needed to create these tests are to be done in the future. Instead we have chosen to test them by manually going through our use cases while checking that everything works correctly.

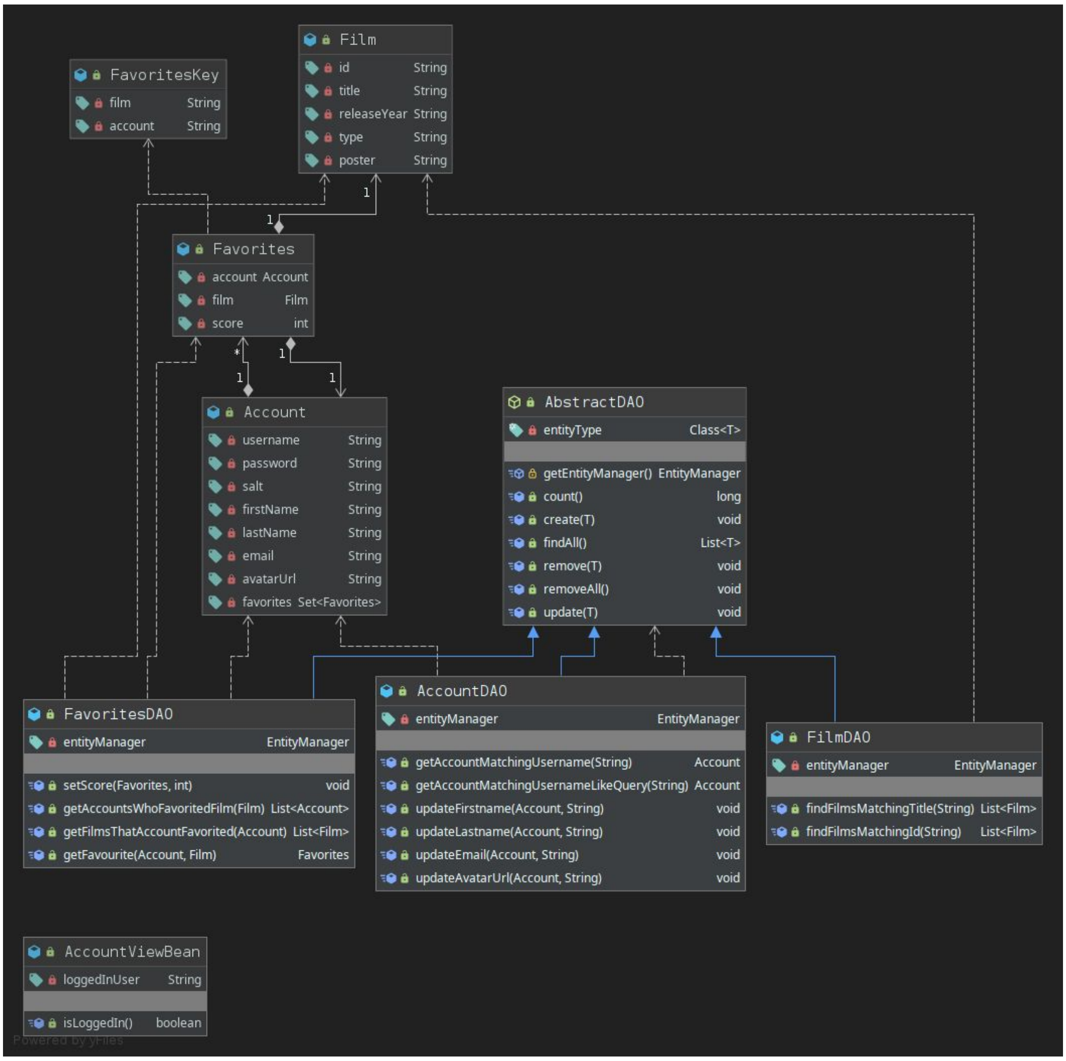
A few examples of manual tests we used:

1. Register user with all information
2. Manually check that the correct data is now in the database

1. Enter site without being logged in
2. Click on the login-button
3. Enter a correct username and password and click login
4. Check that the user now is properly logged in

1. Enter site while being logged in on an account without any favorites
2. Search for “Spider-man”
3. Favorite the first result
4. Click on Watchlist-button and check that the movie you favorited is there

7. Model Diagram


Film
id String
title String
releaseYear String
type String
poster String
FavoritesKey
film String
account String
Favorites
account Account
film Film
score int
1
1
*
1
1
1
Account
username String
password String
salt String
firstName String
lastName String
email String
avatarUrl String
favorites Set<Favorites>
AbstractDAO
entityType Class<T>
getEntityManager() EntityManager
count() long
create(T) void
findAll() List<T>
remove(T) void
removeAll() void
update(T) void
FavoritesDAO
entityManager EntityManager
setScore(Favorites, int) void
getAccountsWhoFavoritedFilm(Film) List<Account>
getFilmsThatAccountFavorited(Account) List<Film>
getFavourite(Account, Film) Favorites
AccountDAO
entityManager EntityManager
getAccountMatchingUsername(String) Account
getAccountMatchingUsernameLikeQuery(String) Account
updateFirstname(Account, String) void
updateLastname(Account, String) void
updateEmail(Account, String) void
updateAvatarUrl(Account, String) void
FilmDAO
entityManager EntityManager
findFilmsMatchingTitle(String) List<Film>
findFilmsMatchingId(String) List<Film>
AccountViewBean
loggedInUser String
isLoggedIn() boolean
Powered by yFiles

Pretty self-explanatory diagram with our three entities: Favorites, Account and Film. FavoritesKey is the composite key of a film and an account used for Favorites. It also contains the respective DAO:s and AbstractDAO which contains more general database operations. AccountViewBean keeps track of logged in users.

8. Responsibilities

The base of the project was made by Benjamin, Edwin and Pontus as the group chose their application idea after the four laborations in the course. We then split the project into parts such that everyone could work on something.

Benjamin was along with Hugo mostly in charge of the front end and design. Together they created the main browse page and the watchlist page. They mostly worked together on one computer, being able to actively discuss every implementation. Benjamin also worked with Edwin in creating and designing the login and register pages. At the later stages of the project when the general structure was set up it was easy to implement new features in a full stack fashion and therefore he had some amount of interaction with the backend in almost every feature.

Edwin mostly worked on the backend of things, setting up interactions between frontend and model, as well as structuring the project according to MVC. He was also trying to help others in the group when they needed it. He also worked together with Benjamin to create and style the Login and register page.

Hugo worked together with Benjamin and was in charge of the design and main user functionality of the application. This includes building and designing the front end for the home page and the watchlist page. Hugo also built the explore page in which you can find other users profiles and watchlist. Here you can also add titles from users profiles as well as add the entire list. At this point the responsibilities were more “fullstack like”, as Hugo built both front end, design, parts of backend and some testing.

Oskar started the project exploring the possibilities of using React as a frontend library. Using React proved to be tedious as we encountered many issues using the npm server together with maven and JavaEE. Along with this, Oskar also took part in designing the film cards and finding css for the spin. He was also in charge of the profile section, developing the entire frontend and large parts of the backend.

Pontus integrated the functionality with the OMDb API and helped with the backend. He also wrote the tests for the data access objects and OmdbService. Made an initial html-file and the backend components needed to search for objects in the API, display them in a table, add all results to the local database, favoriting and displaying login status. This html-file was later deleted and the functionality was moved to new files.

In addition much development was done together, discussing and helping each other develop the entire application.