**University of Pisa**

SCUOLA DI INGEGNERIA

Corso di Laurea in Artificial Intelligence and Data Engineering


# Moviegram documentation

**Anno Accademico 2019–2020**

# 1 Introduction

The **Moviegram** application offers a search and information service in the field of cinema. When the application starts, the system requires authentication to use the service. The logged-in user can perform a search by entering the first characters of a movie title in the search bar, obtaining a list of 10 movies in the database. After that you can select one of the proposed titles or carry out a more in-depth search, adding characters. Selecting a row, the system allows you to view more information in the right section, including cover, title, director and ratings. The user can leave a mark from 1 to 5 for the selected movie. There is also a module for the system administrator: when he logs in, he's redirected to a different activity than the user's one. In this page, he can view some statistics linked to the movies and the searches carried out by the users of the application, such as the ranking of 10 most voted films or the 10 most sought after films. Moreover, the system admin can add new movies to the application database or delete those already present, by searching by title.

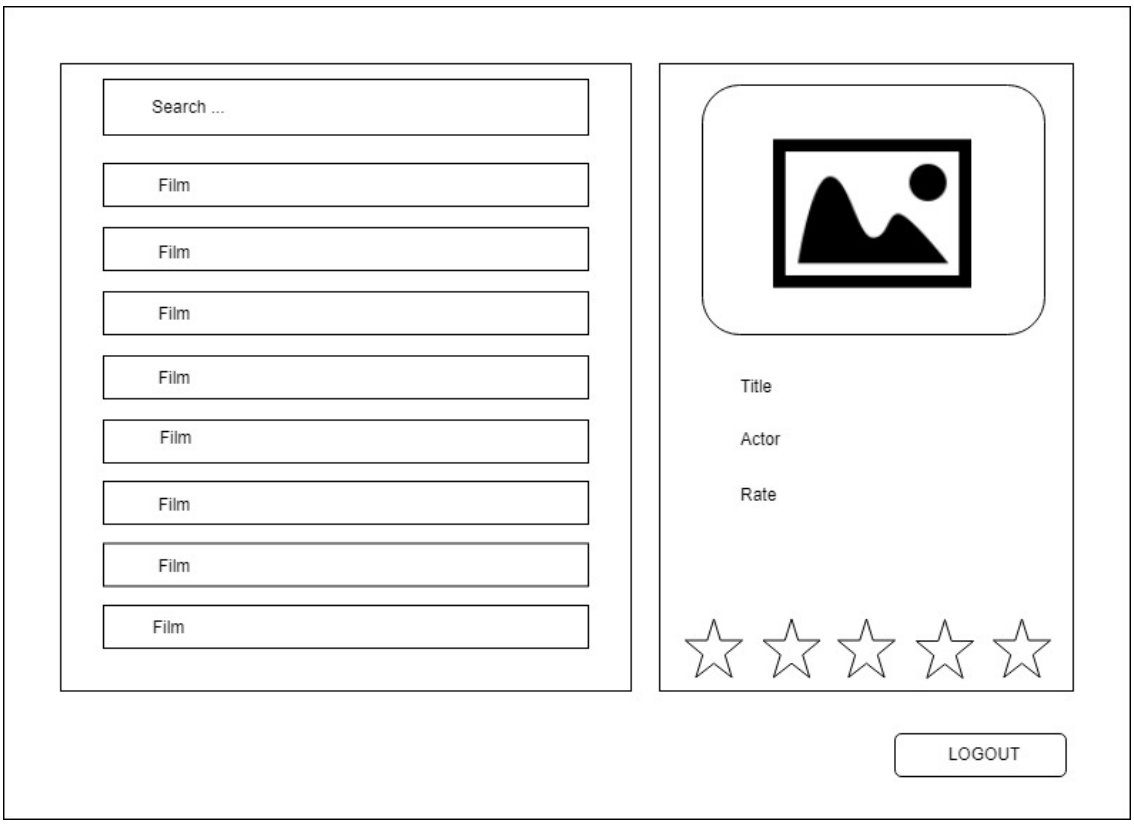


**Figure 1:** User Mockup

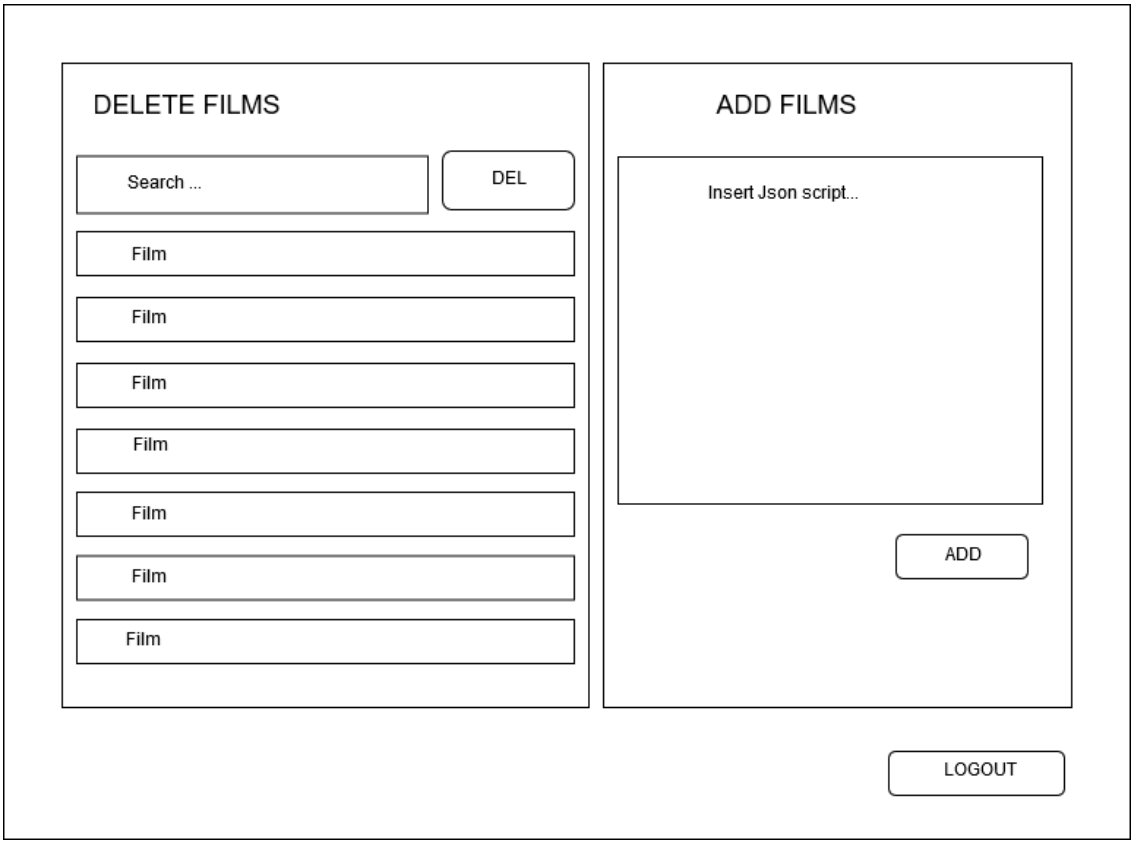


**Figure 2:** Admin Mockup

# 2 Analysis and workflow

## 2.1 Requirements

### 2.1.1 Functional requirement

The system has to allow the user to carry out basic functions such as:

- To sign up into the system.
- To login to the system.
- To search for a movie.
- To vote a movie.
- To view a list of the top rated movies, both in general and filtered by a selected country.
- To view a list of the top production houses.
- To view the top countries in making movies, filtered by a selected year.
- To view the most active users in the application, i.e. the users that have given most votes.

The system has to allow the administrator to carry out basic functions such as:

- To login to the system.
- To add a movie.
- To delete a movie.

### 2.1.2 Non-functional requirements

- Usability, ease of use and intuitiveness of the application by the user.
- Availability, with the service guaranteed h24, using replicas.
- The system should provide access to the database with a few seconds of latency.
- Eventual consistency.

## *UML Use Cases Diagrams*

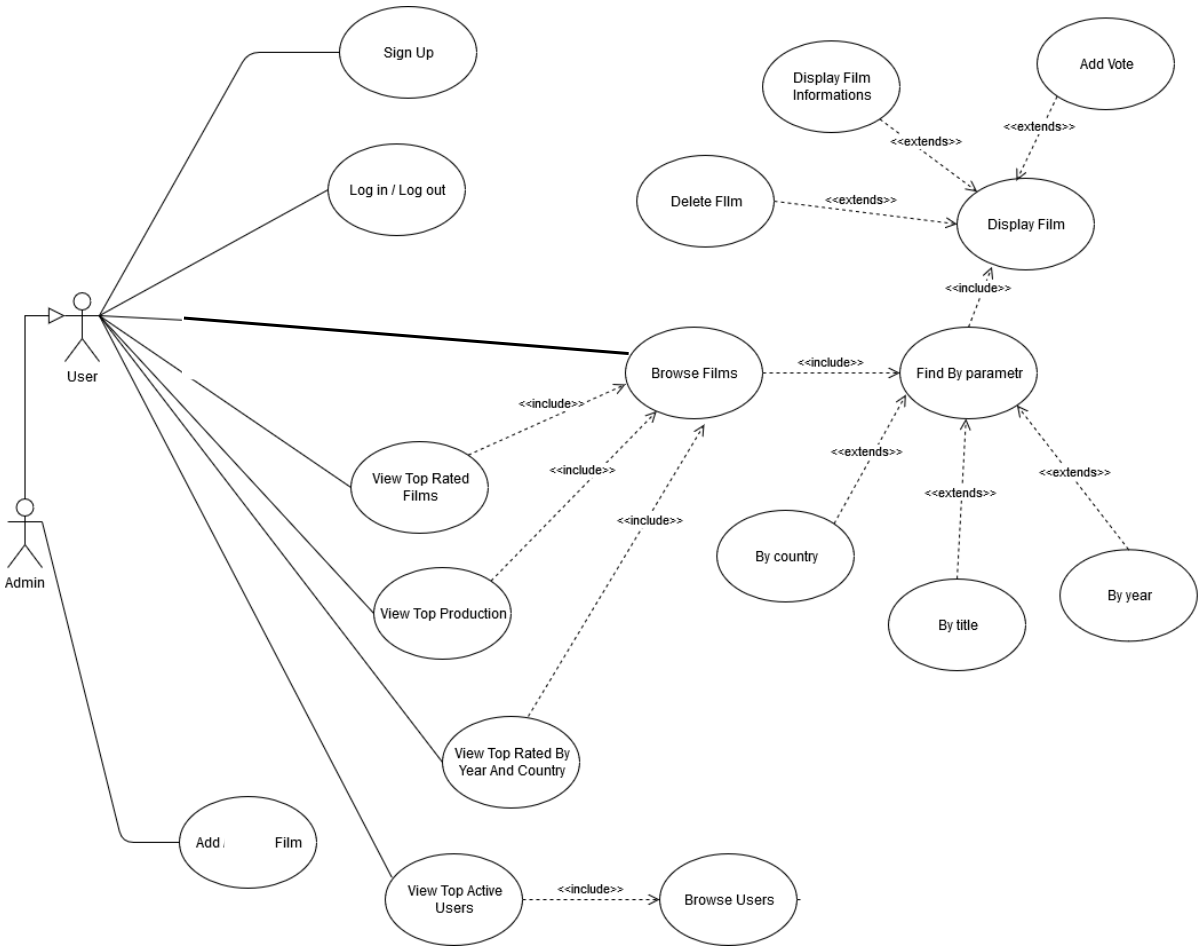


**Figure 3:** Use cases diagram

## 2.3 Analysis of entities

This diagram represents the main entities of the application and the relations between them.

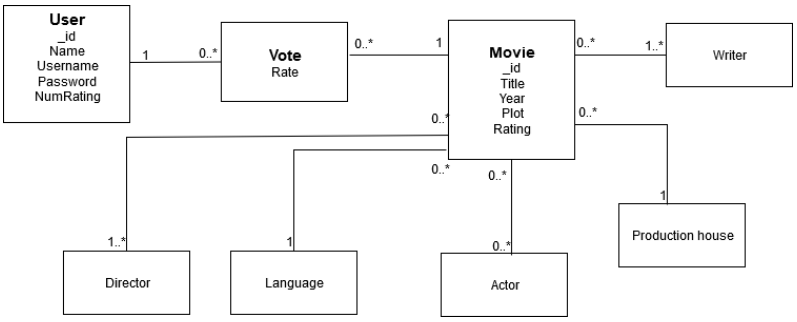


**Figure 4:** UML analysis diagram

### 3.3 Prepare connection to guarantee Availability

In order to guarantee a high level of availability, we give back the control to application after the write on the primary server, without waiting the update of all replicas. As said previous, this choice guarantee a good perfomance in term of speed, but decrease the part related to consistency, it could happen that a user reads some "old" data from a replica, in case of primary server failure.

```
mongoClient = MongoClients.create(
   MongoClientSettings.builder()
           .applyToClusterSettings(builder ->
                   builder.hosts(Arrays.asList(new ServerAddress(ip0, port0),
                                               new ServerAddress(ip1, port1),
                                               new ServerAddress(ip2, port2))))
                    .writeConcern(new WriteConcern().ACKNOWLEDGED)
           .build());
```

### 3.4 Populating the database

The dataset used in the **Moviegram** application was created by scraping the Open Movie Database, using their API (`http://www.omdbapi.com/`). Given a movie title, this API returns all the information in the OMDb about that movie. Before started building up the dataset using the API, it was necessary to obtain a list of movie titles, to be passed to the API; the list of titles was built up using different web pages as sources:

- a page of Wikipedia was scraped
- the pages referring to each year from 2007 to 2021 on the site `https://www.wildaboutmovies.com/` were scraped
- a csv file in the GitHub repository `https://github.com/fivethirtyeight/data` was used

Once the titles' list was ready, the only thing left to do was to obtain the key requested by the OMDb API to download the films' information. Each key provided by the database allows a user to download 1000 film's information per day. Once the movies' list and the keys were ready, we could start to use our scraping application.

```java
public class OmdbScraper {
    final static String key = "587a7b27";
    //final static String key = "84a209f9";
    public static void main(String[] args) throws IOException{
        int i = 0;
        String url = "http://www.omdbapi.com/?apikey="+key+"&t=";
        String path = "C:\\Users\\usr\\Desktop\\Unipi\\Large scale and multi-structured database\\progetto\\";
        String source = "source.txt";
        String destination = "movies_DB.json";

        try(BufferedReader bufferedReader = new BufferedReader(new FileReader(path+source))) {
            while(i < 1) {
                String movie = bufferedReader.readLine();
                System.out.println(i+": "+movie);
                Document doc = Jsoup.connect(url+movie).ignoreContentType(true).get();
                //Document doc = Jsoup.connect(url).ignoreContentType(true).get();
                Elements body = doc.getElementsByTag("body");//getElementsByClass("data");
                //System.out.println(body.text()+"\n");
                try(BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(path+destination,true))) {
                    String fileContent = body.text()+"\n";
                    bufferedWriter.write(fileContent);
                }
            i++;
            }
        }
    }
}
```

**Figure 6:** Java class of OMDBScraper

Using our "OmdbScraper" application, we managed to obtain the collection of movies' information in a JSON format.
The format of our JSON elements are the following:


```
_id: ObjectId("5e05144f847f991da90d42c1")
Title: "Howl's Moving Castle"
Year: "2004"
Rated: "PG"
Released: "17 Jun 2005"
Runtime: "119 min"
Genre: "Animation, Adventure, Family, Fantasy"
Director: "Hayao Miyazaki"
Writer: "Hayao Miyazaki (screenplay), Diana Wynne Jones (novel)"
Actors: "Chieko Baishô, Takuya Kimura, Akihiro Miwa, Tatsuya Gashûin"
Plot: "When an unconfident young woman is cursed with an old body by a spitef..."
Language: "Japanese"
Country: "Japan"
Awards: "Nominated for 1 Oscar. Another 14 wins & 19 nominations."
Poster: "https://m.media-amazon.com/images/M/MV5BZTRhY2QwM2UtNWRlNy00ZWQwLTg3Mj..."
Ratings: Array
  0: Object
     Source: "Internet Movie Database"
     Value: "8.2/10"
  1: Object
     Source: "Rotten Tomatoes"
     Value: "87%"
  2: Object
     Source: "Metacritic"
     Value: "80/100"
Metascore: "80"
imdbRating: "8.2"
imdbVotes: "292,947"
imdbID: "tt0347149"
Type: "movie"
DVD: "07 Mar 2006"
BoxOffice: "$4,520,887"
Production: "Buena Vista"
Website: "N/A"
Response: "True"
```


*A better solution should include an embedded array UserRatings, rather than the current Ratings array. Each element of this array should be composed as follows:*
*{*
*{userName: Pietro*
*Rating: 7.3}*

*In this way, using MongoDB features, it is possible to quickly calculate the average rating of a movie and counting the number of votes that the movie received.*
*In the solutions proposed by the students that developed this project, the imbRating and the imbVotes have been used for the statistics and updated each time a new votes was added. The Ratings array was in practice never used!*
*Remember to store in your DB only the fields that you plan to actually use in your application.*

**Figure 7:** Example of JSON document

We integrated our movies dataset with a plot dataset, found on `kaggle.com` (wiki_plot). In order to do so, we searched for matching between the plot's movie title and the movie title in our collection; once the matching was found, the plot field in the original element was replaced with the data presents in the plot dataset.

### 3.4.1 Collections

At the end we modeled all the entities in two collections: *moviesCollection* and *usersCollection*. The first one store all the information related to each movie, also the following entities: actors, production houses, writers and directors. An example is shown in figure 7.
The second collection stores all information related to the users (name, username, password, and country), in this case we store also the entity Vote, inserting each Vote in an array. We decide to store the votes that a user give to a movies in an array in usersCollections instead to store the votes that a movie received in the moviesCollection, because in the application we are interested to retrieve the votes given by the logged user.

*Pay attention to the highlighted text! Read the comments written in the previous page!*


**Figure 8:** Example of document of collection of user.