# Task 2 – Movie Database
Design Document

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

January 28, 2020

## Contents

# 1 Specifications

## 1.1 Application Overview

The application is an aggregator of movies and movie ratings with the purpose of providing logged users statistics and informations about a large set of movies. Logged-in user can also rate movies they have watched while not logged-in users may still use the service to browse movie rankings and statistics but they are not able to give their rate. Only movies released in Italy are considered.

All users can search a movie and view its details (e.g., title, original title, duration, cast, ...) along with its aggregated statistics about ratings.

In addition, all users can browse the list of movies sorting and filtering it by many parameters (e.g. year, genre, country, actors, ...).

System administrators can view all user pages and ban users. In order to do that, he can check the full history of ratings.

The movie database will be built upon the publicly available IMDb dataset.

The ratings will be gathered also by periodically scraping other websites (e.g., Rotten Tomatoes, Coming Soon, MyMovies).

## 1.2 Actors

Anonymous user, registered user, administrator and bot.

## 1.3 Requirement Analysis

### 1.3.1 Functional Requirements

An **anonymous user** must be able to register in order to become a *registered user*. Login is carried out using username and password selected by the user when registering. Username must be unique. A valid email address is also required in order to register. An email cannot be used more than once.

Both **anonymous user** and **registered user** must be able to:

- view details and average rating of a specific movie
- view a list of movies and filter it by many parameters. Combined filters are also allowed
- view aggregated statistics about movies grouped by year, country, actor, director, genre

A **registered user** must be able to rate a movie, in addition to what anonymous user can do. A registered user must also be able to manage his profile. In the profile a registered user can:

- check, add and modify his personal data
- browse the history of his rates
- view aggregated statistics about his profile (i.e. most viewed genre, most recurrent actor, etc...) based on his rated movies

- delete the account

Finally, a registered user can logout in any moment.

An **administrator** is a special registered user who must be able to ban users. In order to do that, an administrator can check a global rating history to retrieve information about all the application's activity, and to check every user's profile. Banned user's rating are automatically removed from the database. Email and username of banned users cannot be used again.

The **bot** is not a real user but an entity used to periodically update the database in order to add new movies and update external ratings.

#### 1.3.2 Non-Functional Requirements

- Availability: the Database must be replicated in order to be always available. Write operations on the Database can be eventually consistent.
- Scalability: the application must be able to scale to an arbitrary number of servers.
- Security: passwards must be stored in a secure way.
- Responsive UI: Client-side application must provide a responsive view both for pc, laptops and mobile devices.

## 2 Design

### 2.1 Use-case diagram

The use-case diagram is shown in Figure 1. Different colors are used to highlight cases that are exclusive of some actors: blank cases are referred to all users; blue cases are referred to registered user and admin; yellow cases are exclusive of the admin.

### 2.2 Class diagram

The class diagram is shown in Figure 2. It was decided to show *Country*, *Year* and *Genre* as separate entities as they are some of the fields over which aggregate statistics are calculated. In the database implementation, this will reflect in having a separate collection for each of them in which pre-computed results of the aggregations will be stored.

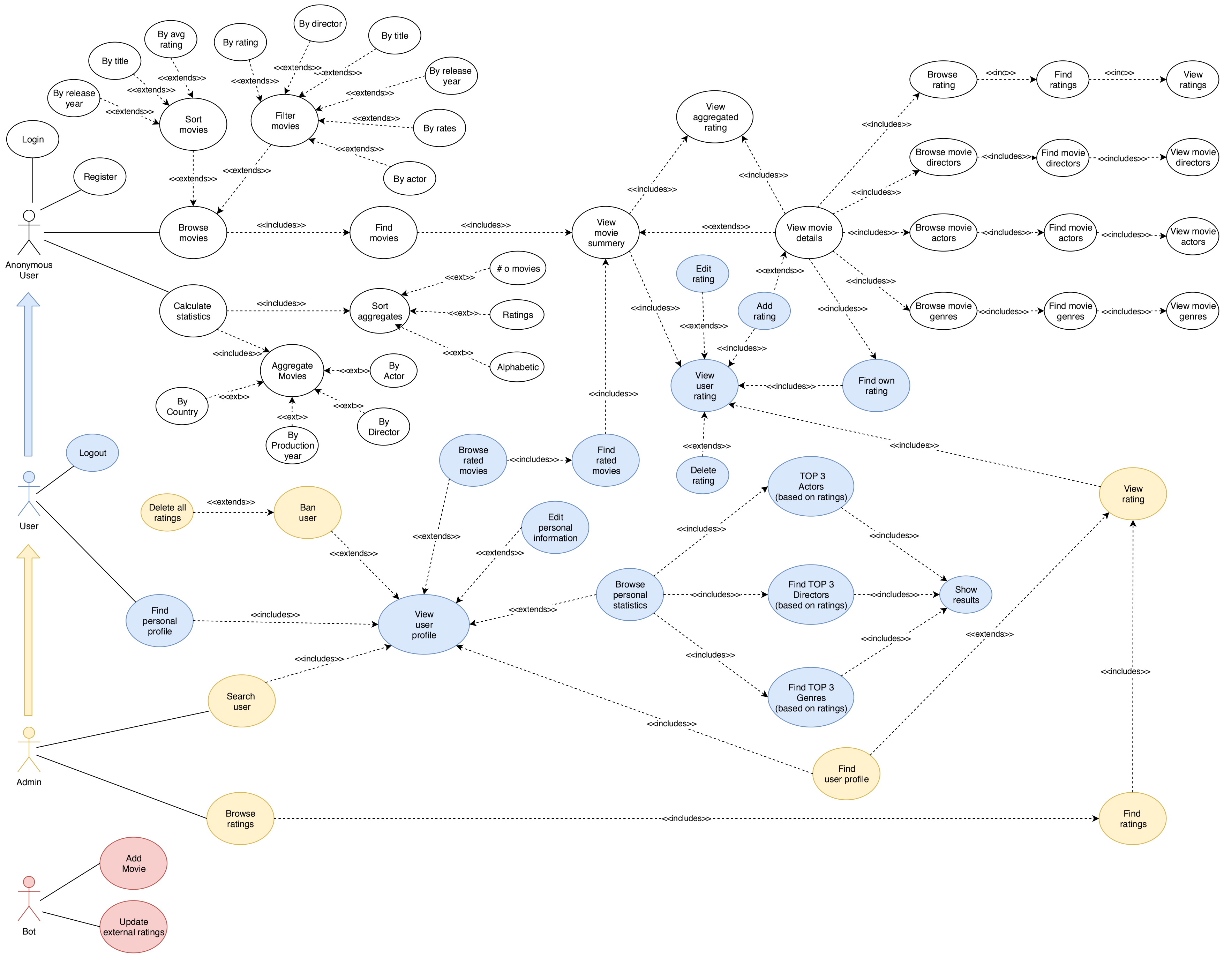

Figure 1: Use-case diagram

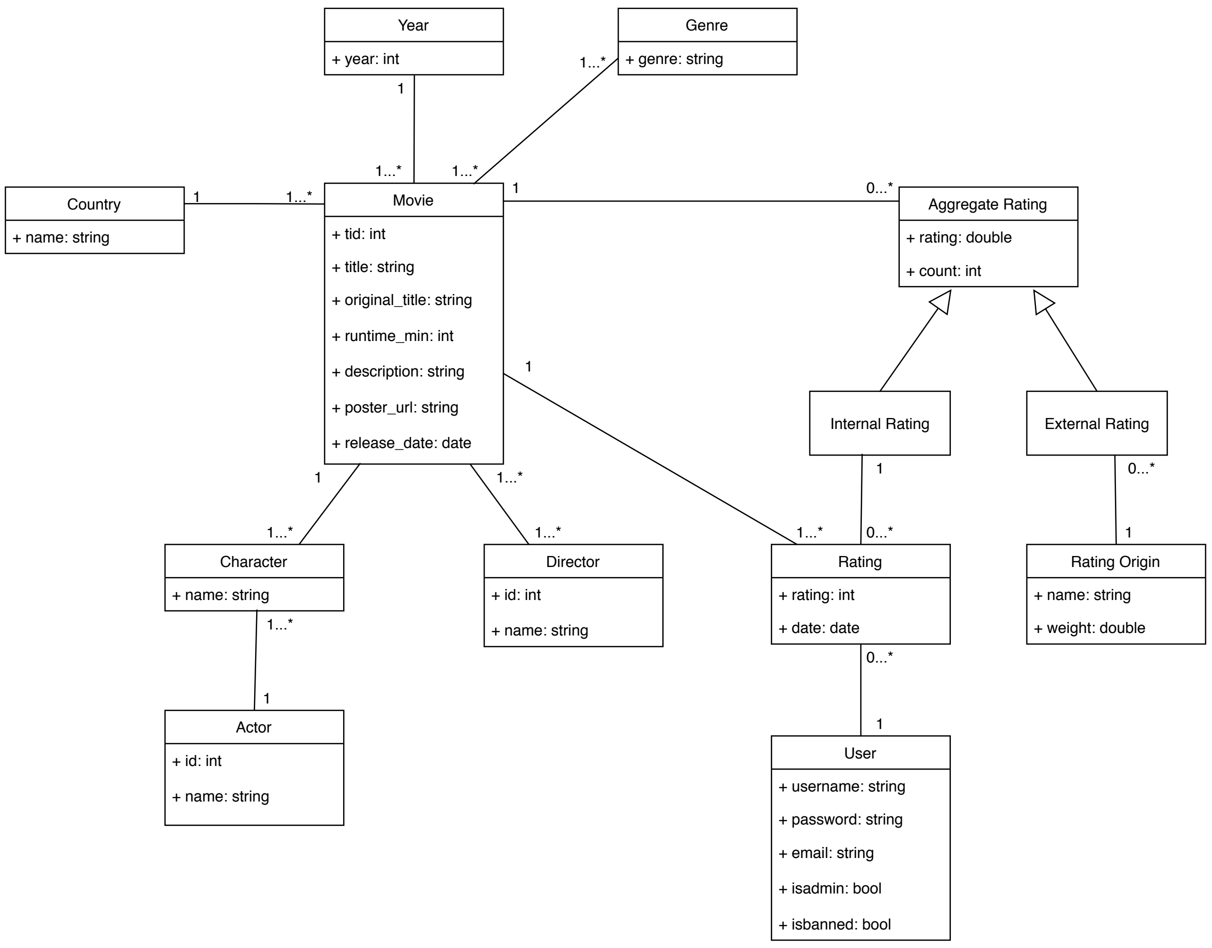

Figure 2: Class diagram for the identified entities

## 2.3 Data model

The data model is split in 9 different collections:

- Movies (Section 2.3.1)
- Users (Section 2.3.8)
- Ratings (Section 2.3.2)
- Actors (Section 2.3.3)
- Directors (Section 2.3.4)
- Years (Section 2.3.5)
- Genres (Section 2.3.6)
- Countries (Section 2.3.7)
- BannedUsers (Section 2.3.9)

In each following subsection, an example document is shown for every collection.

### 2.3.1 Movies

```
{
    _id: 7286456,
    title: "Joker",
    original_title: "Joker",
    runtime: 122,
    region: "US",
    full_region: "USA",
    year: 2019,
    date: "2019-10-04",
    description: "In Gotham City, mentally troubled comedian [...]",
    poster: "https://m.media-amazon.com/images/M/[...].jpg",
    characters: [
        {
            name: "Joker",
            actor_name: "Joaquin Phoenix",
            actor_id: 1618
        },
        ...
    ],
    directors: [
        {
            id: 680846,
            name: "Todd Phillips",
        }
    ],
    genres: ["Crime", "Drama", "Thriller"],
    ratings: [
        {
            source: "internal",
            avgrating: 9,
            count: 100,
            weight: 2
        },
        {
            source: "IMDb",
            avgrating: 8.6,
            count: 628981,
            weight: 1
        },
        ...
    ],
    total_rating: 8.87
}
```

**Notes** Character and director nested documents contain redoundant data (person's name) that is introduced to reduce the number of seeks necessary to return the movie details to the user.

This collection is mainly read-heavy since most fields will be written only once. The only fields subject to change are the ones related to ratings. For these reasons, we can affort many indices to speed-up the aggregations.

**Indices** Indices on the following fields will be set for this collection:

- `title`
- `region`
- `year`
- `characters.actor_id` (multikey index)
- `directors.id` (multikey index)
- `total_rating`

An index on the *total rating* is necessary for efficiently returning a subset of movies based on their rating. However, frequent writes on it would negatively impact the performance on the system. Therefore, this field will not contain the real-time total rating but will be updated periodically.

### 2.3.2 Ratings

```
{
    _id: <ObjectId>,
    user_id: <ObjectId>,
    movie_id: 7286456,
    date: "2020-01-20",
    rating: 10
}
```

**Notes** The raw ratings are kept separated from the movie they refer to because the single ratings are never shown to the user. Furthermore, new ratings are expected to be frequently added. By using a separate collection, the problem of a long array of nested documents is avoided. However, the introduction of an index is necessary.

**Indices** An index will be introduced on the *movie_id* field in order to speed up aggregations.

### 2.3.3 Actors

```
{
    _id: 1618,
    name: "Joaquin Phoenix",
    avg_rating: 8.75,
    movie_count: 17
}
```

**Notes** This collection will store the pre-computed results of the aggregation.

### 2.3.4 Directors

```
{
    _id: 680846,
    name: "Todd Phillips",
    avg_rating: 7.45,
    movie_count: 4
}
```

**Notes** This collection will store the pre-computed results of the aggregation.

### 2.3.5 Years

```
{
    _id: 2019,
    avg_rating: 6.45,
    movie_count: 867
}
```

**Notes** This collection will store the pre-computed results of the aggregation.

### 2.3.6 Genres

```
{
    _id: "Crime",
    avg_rating: 4.78,
    movie_count: 751
}
```

**Notes** This collection will store the pre-computed results of the aggregation.

#### 2.3.7 Countries

```
{
    _id: "US",
    full_name: "USA",
    avg_rating: 4.78,
    movie_count: 1597
}
```

**Notes** This collection will store the pre-computed results of the aggregation.

#### 2.3.8 Users

```
{
    _id: <ObjectId>,
    username: "joker",
    password: "<HASHED PASSWORD>",
    email: "joker@dccomics.com",
    isadmin: 1 (optional),
    rated_actors: [
        {
            actor_id: 1618,
            avg_rating: 7.5,
            rating_count: 2
        },
        ...
    ],
    rated_directors: [
        {
            director_id: 680846,
            avg_rating: 7.7,
            rating_count: 4
        },
        ...
    ],
    rated_genres: [
        {
            genre: "Crime",
            avg_rating: 7.1,
            rating_count: 4
        },
        ...
    ]
}
```

**Notes** In order to provide the user with statistics about his favourite actors, directors and genres, partial statistics need to be stored within the *Users* collection. In fact, calculating them is very expensive since it involves more than one collection.

**Indices** An index will be introduced on the *username* and *email* fields. Furthermore, these fields must also be unique.

### 2.3.9 BannedUsers

```
{
    _id: <ObjectId>,
    username: "topolino.hackerino",
    email: "hackerino@topolino.it"
}
```

**Notes** This collection stores the username and passwords of banned users. When a user is banned, its record in the *Users* collection is moved to this one. When a new user registers, this collection is checked for any match and, if any is found, the registration will fail.

## 2.4 Software Architecture

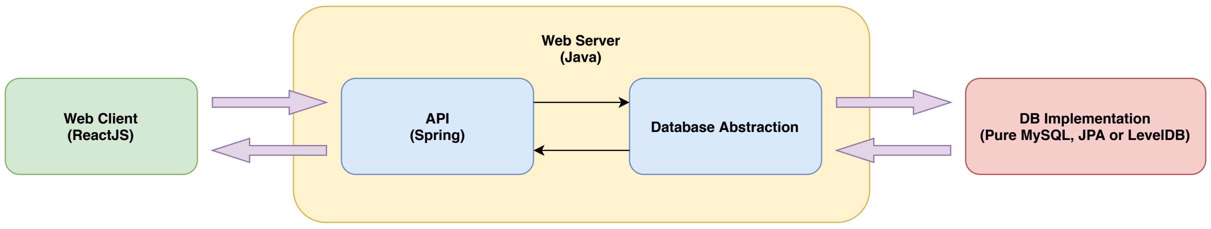

Figure 3: Software architecture

As it can be seen, a Java server back-end is the core of the proposed architecture, providing both the APIs – through the open source Spring Framework – and the database interface – which has to allow a high flexibility in terms of database implementation.

Client-side, a ReactJS front-end has been chosen for the implementation of the web app the user will interact with.