Task 2 – Movie Database
Design Document

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

May 8, 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 average rating from users and from external sources.

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 profile pages and ban users. In order to do that, he can check the full history of ratings. Once a user is banned, he can no longer log in and his username and email cannot be used by new users.

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

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

1.2 Actors

Anonymous user, registered user, administrator and updater "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: the user can choose on which field to aggregate movies (year, country, actor, director, genre) and additional filters (like the movie browsing feature). E.g. the user might want to see the ranking of the countries with the best movies in the last 10 years.

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 **updater "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: white 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.

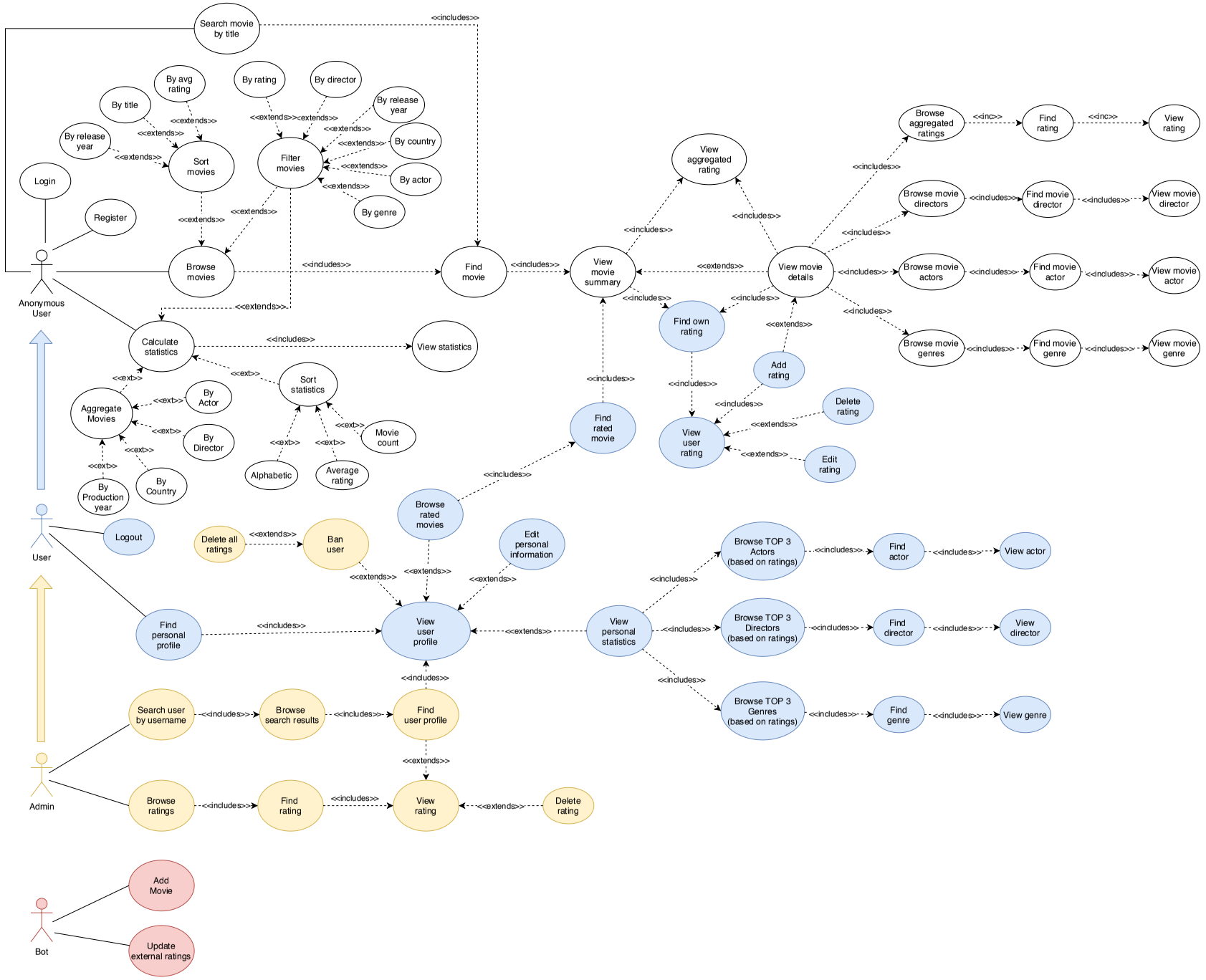

Figure 1: Use-case diagram

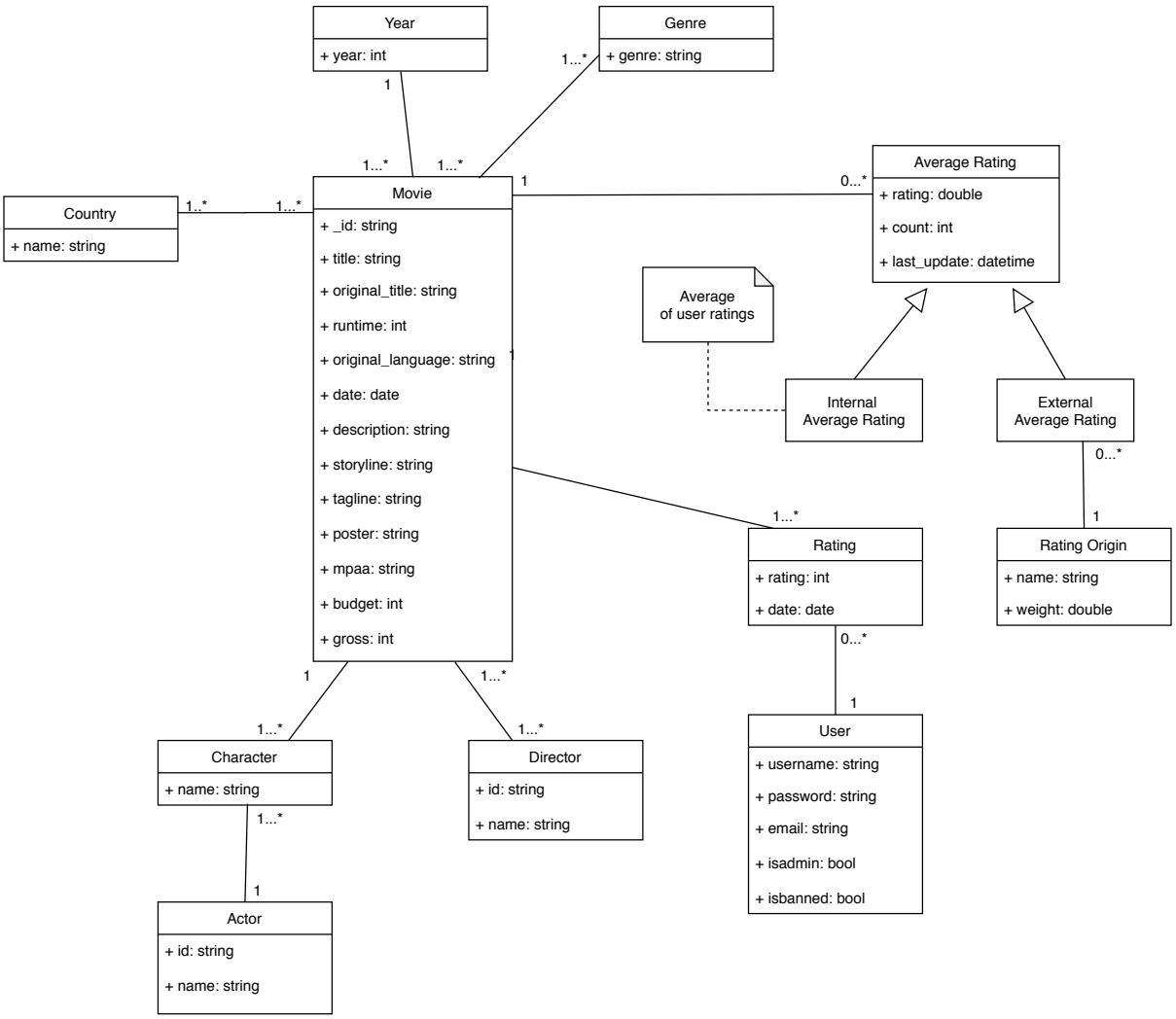

Figure 2: Class diagram for the identified entities

2.3 Data model

The data model is split in 3 different collections:

- Movies (Section 2.3.1)
- Users (Section 2.3.3)
- Ratings (Section 2.3.2)

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

2.3.1 Movies

```
{
    "_id": "tt7286456",
    "title": "Joker",
    "original_title": "Joker",
    "runtime": 122,
    "countries": ["USA", "Canada"],
    "original_language": "English",
    "year": 2019,
    "date": "2019-10-04",
    "description": "In Gotham City, mentally troubled comedian [...]",
    "storyline": "Joker centers around an origin of the iconic arch [...]",
    "tagline": "Put on a happy face.",
    "poster": "https://m.media-amazon.com/images/M/[...].jpg",
    "mpaa": "Rated R for strong bloody violence, disturbing behavior, [...]"
   ,
    "budget": 55000000,
    "gross": 1074251311,
    "characters": [
        {
            "name": "Joker",
            "actor_name": "Joaquin Phoenix",
            "actor_id": "nm0001618"
        },
        ...
    ],
    "directors": [
        {
            "id": "nm0680846",
            "name": "Todd Phillips",
        }
    ],
    "genres": ["Crime", "Drama", "Thriller"],
    "ratings": [
        {
            "source": "internal",
            "avgrating": 9,
            "count": 100,
            "weight": 2,
```

```
            "last_update": "2019-11-28"
        },
        {
            "source": "IMDb",
            "avgrating": 8.6,
            "count": 628981,
            "weight": 1,
            "last_update": "2019-11-28"
        },
        ...
    ],
    "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` (text index)
- `country`
- `year`
- `characters.actor_id` (multikey index)
- `directors.id` (multikey index)
- `total_rating`
- `ratings.last_update`

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": {
        "user_id": <ObjectId>,
        "movie_id": "tt7286456"
    },
    "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.

Another reason for keeping it separated is that they may be accessed per-user, per-movie (for calculating statistics) and globally (by the administrator).

Indices

- `_id.movie_id`: speed up aggregations.
- `(_id.user_id, _id.movie_id`: quickly find user's ratings (compound multi-key index).

2.3.3 Users

```
{
    "_id": <ObjectId>,
    "username": "joker",
    "password": "<HASHED PASSWORD>",
    "email": "joker@dccomics.com",
    "isadmin": True (optional),
    "isbanned": False (optional),
    "sessions": [
        {
            "session_id": "<session_id>",
            "expiry": "2020-01-28"
        },
        ...
    ]
}
```

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* (text index) and *email* fields. Furthermore, these fields must also be unique.

2.4 Software Architecture

The application will be made of the following 4 components:

- **Mongo DB**: a MongoDB cluster will be deployed with sharding and replication.
- **React Frontend**: web-based UI.
- **Java Backend**: using *Spring*, the *Java backend* will provide REST APIs to the *React frontent*.

- **Updater "bot"**: two Python scripts are needed in order to nightly update the DB with the latest movies and ratings:
  1. the **IMDB parser** periodically parses the IMDb dataset to add the latest movies.
  2. the **scraper** continuously parses the rating sources to update the ratings. Ratings of newer movies will be updated more often than older.

3 Implementation

3.1 API

3.1.1 GetStatistics

This function is used to return aggregated statistics on movies based on one of the following information attributes:

- genre
- year
- country
- director
- character

Before performing the aggregation, movies can be filtered as shown in the *GetMovieList* API. The statistics returned show the following informations about the aggregation attribute:

- aggregated attribute
- average rating calculated on the aggregation
- number of movies present in the aggregation (calculated after filtering)

Results can also be sorted in ascending or descending order based on each of the previous attributes.

The aggregation pipeline is performed with the following code:

```
AggregateIterable<Document> iterable = moviesCollection.aggregate(
        Arrays.asList(
                Aggregates.match(filters),
                Aggregates.unwind("$" + realUnwindBy),
                Aggregates.group(
                        "$" + realGroupBy,
                        Accumulators.first("name", "$" + realGroupName),
                        Accumulators.avg("avg_rating", "$total_rating"),
                        Accumulators.sum("movie_count", 1)
                ),
                Aggregates.sort(sorting),
                Aggregates.skip(n*(page-1)),
                Aggregates.limit(n)
        )
);
```

The **match** function uses the array of conditions "*filters*" to perform all the filters in a single step. **Unwind** deconstructs the array field and returns a document for each array

element (this is necessary for country, director and character fields). Then, the **group** function groups documents by the value expressed in the "*realGroupBy*" and then applies accumulator expression to each group:

- **Accumulators.first** sets "*realGroupName*" as name of the group
- **Accumulators.avg** calculates the average rating of all the "*total_rating*" and saves them in the "*avg_rating*" attribute
- **Accumulators.sum** calculates the number of movies in the aggregation, by adding 1 to the variable "*movie_count*" for each movie

After that, the results are sorted with the **sort** function, where the argument "*sorting*" is calculated as follow:

```
Bson sorting;
if (sortOrder == 1) {
    sorting = ascending(realSortBy);
} else if (sortOrder == -1) {
    sorting = descending(realSortBy);
} else {
    throw new RuntimeException("sortOrder must be 1 or -1.");
}
```

Here, "*realSortBy*" is the value to sort.

Finally, the **skip** and the **limit** functions are used to manage the display of the results.

4 Database

4.1 Creation of the database

The database is created from scratch starting from the public IMDb dataset available at the link https://www.imdb.com/interfaces/.

We developed a Python script, in order to obtain the data in the form described in the Data Model. However, some information where missing. So we integrated missing information scraping them from other movie's sites while we were scraping for ratings.

4.2 Scraping

4.3 Updating the database