Task 3 – Movie Database
Design Document

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

June 2, 2020

Contents

1 Specifications

1.1 Existing Application Summary

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 users 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 by periodically scraping external websites (e.g., Rotten Tomatoes, Coming Soon, MyMovies).

1.2 Task 3 additions overview

Upon the application described above, which we have already designed and implemented, we will add the management of followed/following relationships between users and the suggestion of movies to the user based on his ratings.

In particular, every logged-in user can follow, or un-follow if previously followed, any other user. Following another user gives the ability to see which movies he has been rating. A dedicated section will allow to see all the ratings coming from followed users in a chronological order. A new section will be built and will be available to logged-in users in their profile page. It will show the list of followers and followings of the user, but will also provide suggestions for new users to follow, based on the current relationship with the other users. User search functionality will be extended to all registered users in order to more quickly find another user given his username.

Furthermore, in his homepage, a logged user will receive suggestions about movies that he might like based on both his ratings and the most recent ratings on the platform. By doing so, suggestions will reflect the current trends in the movie industry.

1.3 Actors

Anonymous user, registered user, administrator and updater “bot”.

1.4 Requirement Analysis

1.4.1 Functional Requirements

In addition to what has already been defined in Task2, a **registered user** can:

- follow another registered user

- un-follow a followed user
- browse the users he is following
- browse the users that are following him
- browse suggestions for new users to follow
- browse a list of suggested movies
- browse the latest ratings of followed users
- view the profile page of any other user
- search a user through a query by username

1.4.2 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**: passwords 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 and green cases are referred to registered user and admin; yellow cases are exclusive of the admin. Green cases highlight the ones that were introduced in task 3 and all refer to actions that a registered user can do.

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.

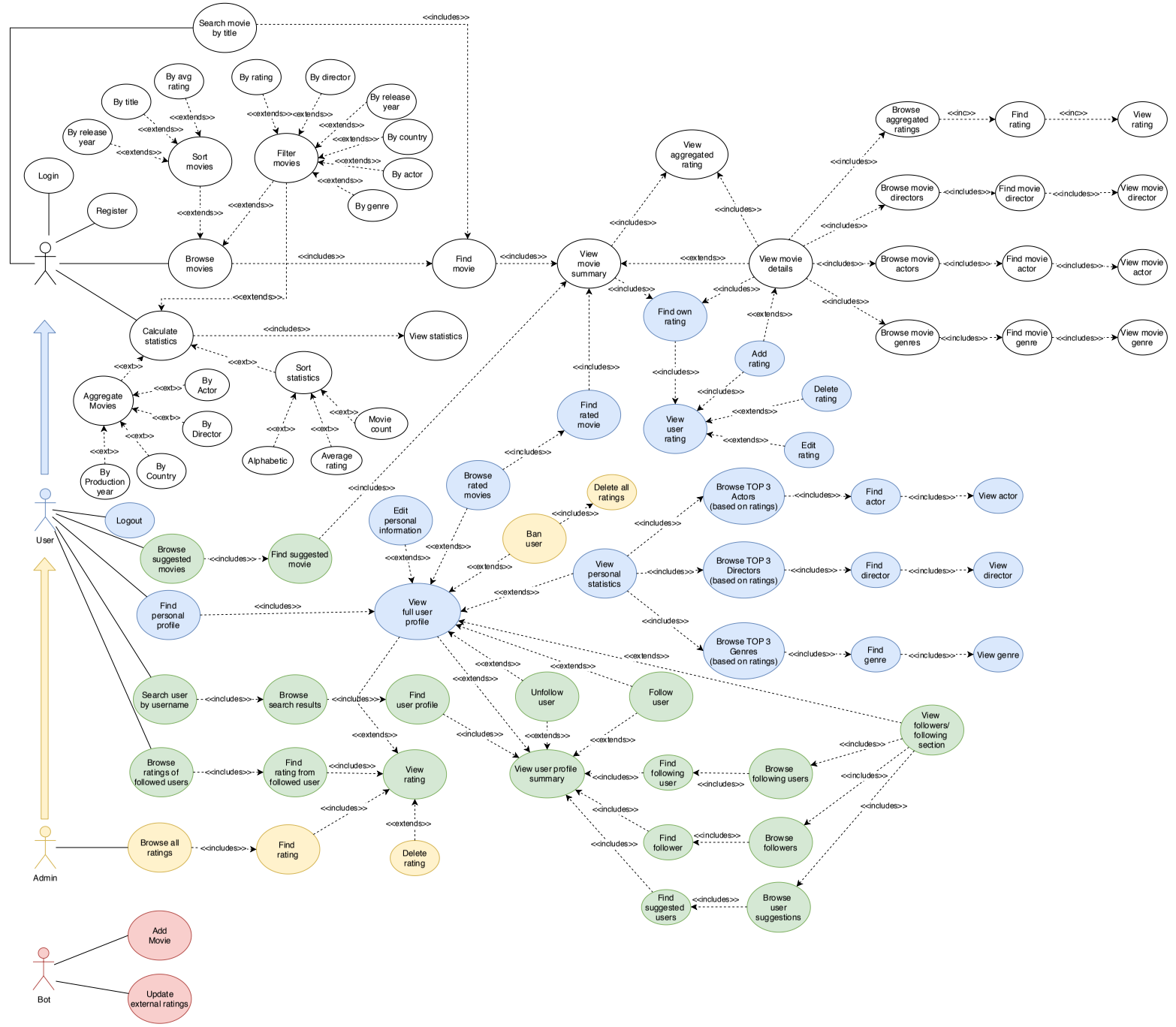

Figure 1: Use-case diagram

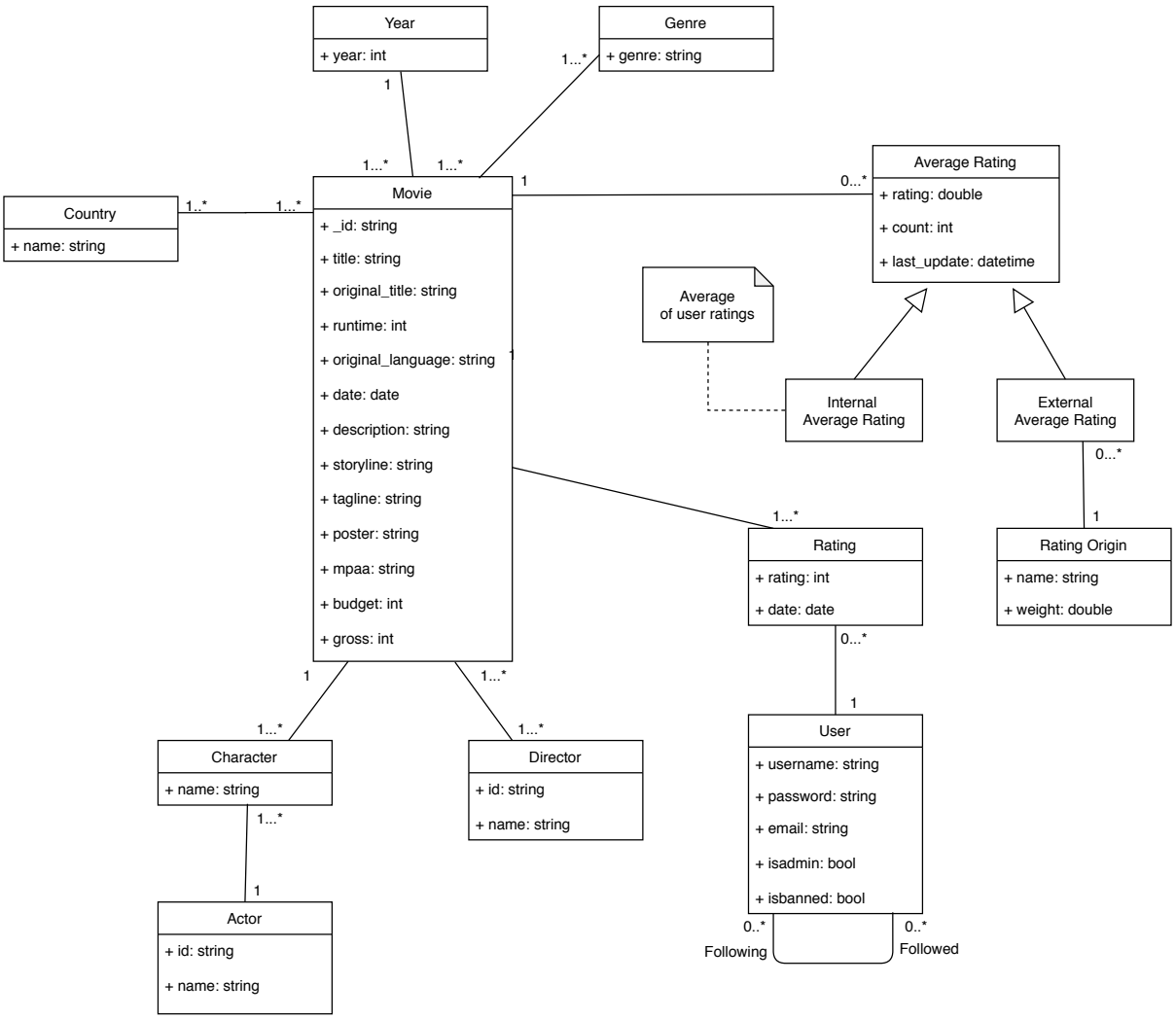

Figure 2: Class diagram for the identified entities

2.3 Data model

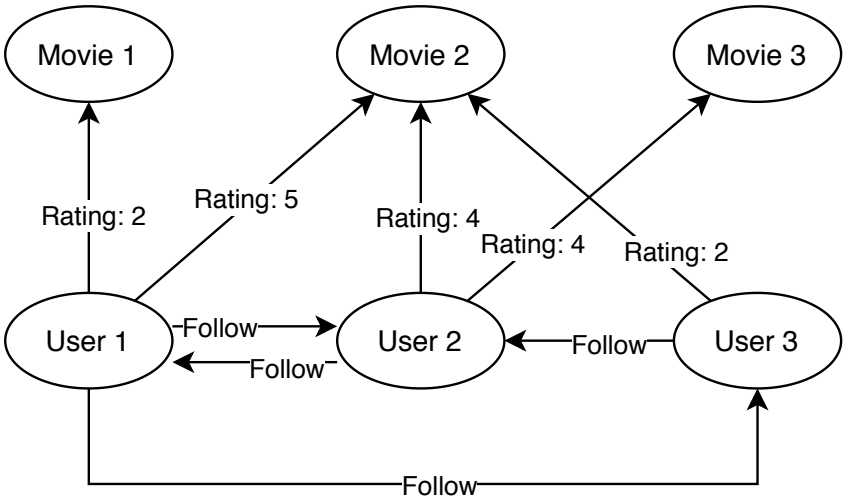

Figure 3: Example graph

The graph database will have 2 different types of nodes and 2 types of edges:

Node types

- **Movie**: it will contain all information needed to show a list of suggested movies: *id, title, year, poster*.
- **User**: it will contain only the *username* and the *_id* since no other information is required.

Edge types

- **Rating** (User->Movie): it will contain the *rating* as attribute.
- **Follow** (User->User): no attribute is required.

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

2.4 Graph-centric operations

This section will explain how we intend to execute the queries in the database in terms of graph operations. Simple CRUD operations will not be reported for brevity's sake.

Movie suggestion Movie will be suggested based on the ratings of the users that liked similar movies as the user we are suggesting to. We consider as liked movies, the ones the user has given a rating greater or equal to 3. For example, in figure 3, *Movie 3* may be suggested to *User 1* since *User 2* liked *Movie 2* as *User 1* and he also liked *Movie 3*. In practice, starting from a user we will go to each movie he liked, then to the users who recently liked that movie, then to other movies they liked. We will then count the number of paths like the one described for each movie (the last one in the path) and show the user the top-N movies given this metric.

User suggestion Users to follow will be suggested based on the number of common follow relationships. For example, in figure 3, *User 3* may be suggested to *User 2* since *User 2* follows *User 1* that follows *User 3*. In practice, starting from the user, we will go to each user he follows and from here to each user that the latter user follows. We will then count the number of paths from the user to the users with common followers and show the user the top-N users given this metric.

2.5 Software Architecture

The application will be made of the following 4 components:

- **Mongo DB**: a MongoDB cluster will be deployed with replication.
- **Neo4j**: an auxiliary graph database will be used to calculate more efficiently the movie suggestions and to store information about follow relationships since this kind of operations are more suited to this kind of DB.
- **React Front-end**: web-based UI.
- **Java Back-end**: using *Spring*, the *Java back-end* will provide REST APIs to the *React front-end*.
- **Updater "bot"**: three 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.
  3. the **synchronizer** periodically synchronizes movies from MongoDB to Neo4j.

  These scripts will executes asynchronously from the *Java back-end*.

3 Implementation

3.1 Neo4j database

Neo4j database is created by a python script that takes data directly from the document database: useful data are first withdrawn from mongoDB, then they are post-processed in order to filter only the necessaries attributes, and then they are merged into neo4j. This can be done thanks to the *pymongo* and *py2neo* modules.

The script manage the consistency of 3 out of the 4 elements present in our graph database:

- vertex **Movie**;
- vertex **User**;
- edge **Rating**.

Edge **Follows** is not managed in this script because it only exists in the graph database.

The work is done by 3 functions (one for each managed element) that simply scan the array of documents that they receives as input in order to keep only the attributes we are interested in. Then they create the relative node and finally they merge it with the current Neo4j database.

The sync function for the **Movie** vertex in shown below:

```
def sync_movie(m):
    global graph

    movie = {k:v for k,v in m.items() if k in keep_movie_attrs}
    n = Node("Movie", **movie)
    graph.merge(n, "Movie", "_id")
```

The sync function for the **User** vertex in shown below:

```
def sync_user(u):
    global graph

    u['_id'] = u['_id'].__str__()
    user = {k:v for k,v in u.items() if k in keep_user_attrs}
    m = Node("User", **user)
    graph.merge(m, "User", "_id")
```

The sync function for the **Rating** edge in shown below:

```
def sync_rating(r):
    global graph

    RATED = Relationship.type("RATED")

    u = graph.nodes.match("User", _id=r["_id"]["user_id"].__str__()).first()
    m = graph.nodes.match("Movie", _id=r["_id"]["movie_id"]).first()
    graph.merge(RATED(u,m, rating=r['rating'], date=r['date']))
```

The consistency is ensured by this script in an asynchronous way by periodically scanning mongoDB in order to find missing or not updated informations to upload on neo4j. Note that asynchronous consistency is sufficient because there are not features that necessarily requires the strict one.