Task 2 – Movie Database
Design Document

Federico Fregosi, Mirko Laruina,

Riccardo Mancini, Gianmarco Petrelli

May 18, 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.

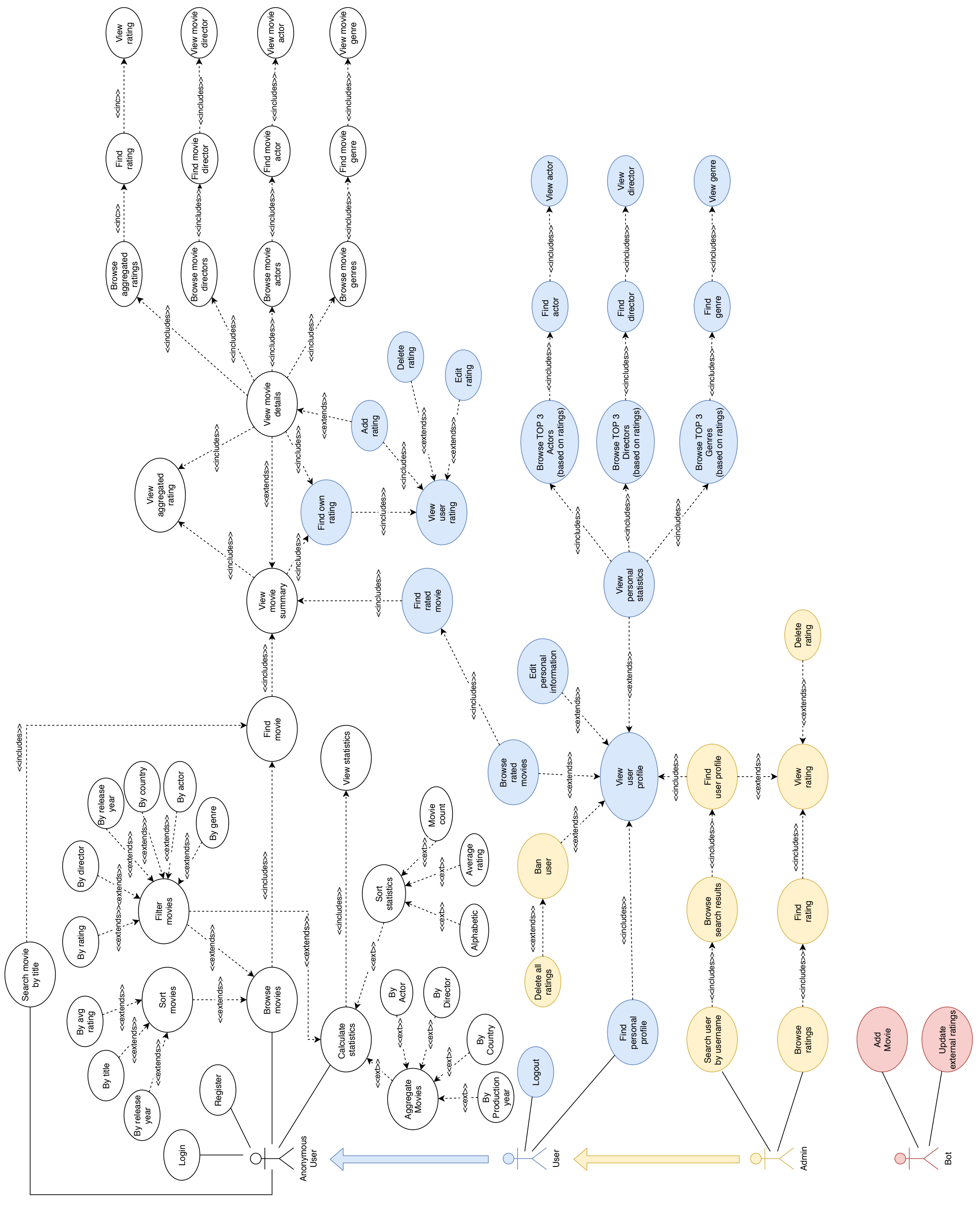

Figure 1: Use-case diagram

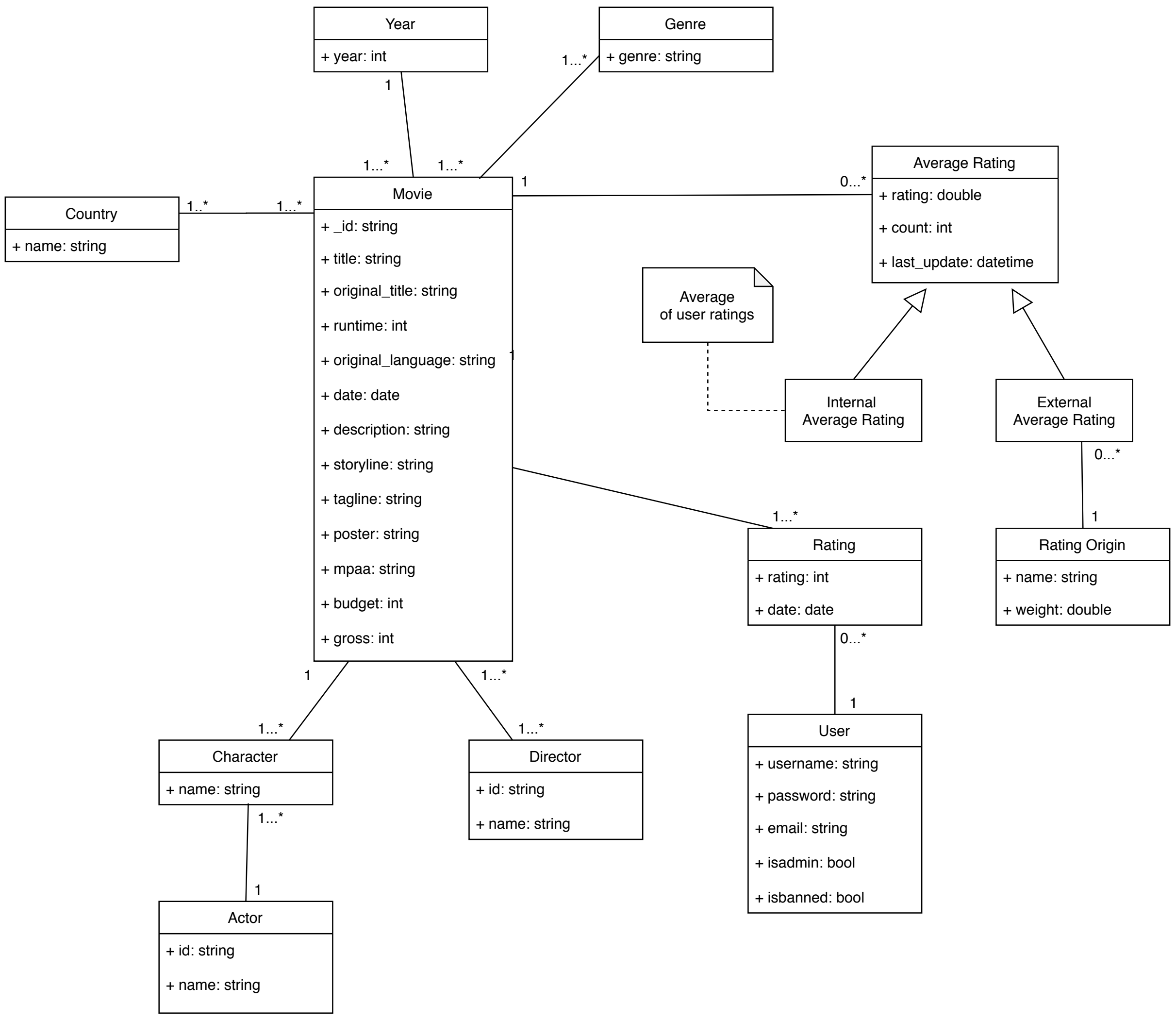

Figure 2: Class diagram for the identified entities

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",
            "sum": 450,
            "count": 100,
            "weight": 1,
            "last_update": "2019-11-28 12:34:56"
        },
        {
            "source": "IMDb",
            "avgrating": 8.6,
            "count": 628981,
            "weight": 0.5,
            "last_update": "2019-11-28 12:34:56"
        },
        ...
    ],
    "total_rating": 4.43,
    "last_scraped": "2019-11-28 12:34:56"
}
```

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`: speed-up sorting by title
- `title`, `original_title` (text index): movie lookup by name
- `countries` (multikey index): filter and aggregation by country
- `genres` (multikey index): filter and aggregation by genre
- `year`: filter and aggregation by year
- `characters.actor_id` (multikey index): filter and aggregation by actor
- `directors.id` (multikey index): filter and aggregation by director
- `total_rating`: filter and sorting by rating
- `last_scraped`: fast lookup for next-to-be-scraped movie (oldest movie).

Among this fields, *total_rating* is the only one that will be updated frequently (all other fields with index are updated with the periodic scraper whose period is of some days). However it is necessary to efficiently fulfill many queries.

2.3.2 Ratings

```
{
    "_id": {
        "user_id": <ObjectId>,
        "movie_id": "tt7286456"
    },
    "date": "2020-01-20",
    "rating": 5
}
```

Notes The raw ratings are kept separated from the movie they refer for three reasons:

1. single ratings are never shown to the user, except when the user looks his own ratings or when the admin looks all ratings

2. they may be accessed per-user, per-movie (for calculating statistics) and globally (by the administrator). The optimization for one use-case would influence the performance of another.

3. new ratings are expected to be frequently added and, thus, the use of a nested document inside the *movies* collection would be infeasible beacause it would grow indefinitely.

However, this choice mandates the introduction of indices for efficiently obtaining ratings of a user or ratings of a movies are required.

Indices

- `_id.movie_id`: speed up aggregations on movie.
- `_id.user_id`: speed up user ratings browse.
- `date`: speed up browsing user and global ratings.

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 *Username*, *email* and *session_id* must be unique across the whole collection.

Indices

- `username` (text index): find user by username
- `username`: find user by exact username
- `email`: find user by email
- `sessions.session_id` (multikey index): find user by session id

2.4 Software Architecture

The application will be made of the following 4 components:

- **Mongo DB**: a MongoDB cluster will be deployed with 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.

  These scripts will executes asynchronously from the *Java backend*.

3 Database

3.1 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.

Collections are saved into Json files and then upserted in MongoDB database with another simple python script with the use of the **UpdateOne** module. Update requests are inserted in append into an array and served through the **bulk_write()** function.

3.2 Scraping

3.3 Updating the database

4 Backend Implementation

The API specification can be found in the `docs/api.md` file. This section will go through the implementation of the main APIs.

4.1 Java classes overview

TODO - Class diagram - Some words - Each class has an Adapter to conveniently convert from/to Mongo Document

4.2 Authentication

Register In order to register a user, it must first be checked that no user with same username and password exists (snippet 1) and then, if no duplicate entry is found, the user can be added to the database (snippet 2). If creation was successful, a new session for the new user is then added (snippet 5). Any subsequent API call must be accompanied by the session identifier in order to identify the user (snippet 6).

Change Password This query (snippet 3) is simply done by matching the user by username and setting the new password.

Login In order to authenticate the user, a matching user with same username and password is found in the database (snippet 4). If a user matches, a new session for the given user is then added (snippet 5).

Logout The session is removed from the DB (snippet 7).

```java
// 1 - check user uniqueness
usersCollection.find(
        or(
            eq("username", u.getUsername()),
            eq("email", u.getEmail())
        )
).first();

// 2 - insert user
usersCollection.insertOne(User.Adapter.toDBObject(u));

// 3 - edit password
usersCollection.updateOne(
    eq("username", u.getUsername()),
    set("password", u.getPassword())
);

// 4 - authenticate user
usersCollection.find(
        and(eq("username", u.getUsername()),
            eq("password", u.getPassword())
    ))
    .first();

// 5 - add session
usersCollection.updateOne(
    eq("_id", u.getId()),
    push("sessions", Session.Adapter.toDBObject(s))
);

// 6 - finds the user whose session is s
usersCollection.find(
        and(
            eq("sessions._id", s.getId())
        ))
    .first();

// 7 - remove session
usersCollection.updateOne(
    eq("_id", u.getId()),
    pull("sessions", eq("_id", s.getId()))
);
```

4.3 Movies

4.3.1 Browse

Returns a list of filtered movies with the given sorting. The result is paged.

1. build a list of filters based on user specifications. People names are matched if all words in the query sring are substrings of the full name (this is done using a regex).
2. Make the filter to pass to the *find* function by and-ing all conditions.
3. Make the query:

   find all movies matching the conditions

   sort by the user-defined sort order

   project only movie important details

   skip the first *n* pages

   limit the results to the page size

If user is logged in, a separate query will be done to fetch the user's ratings of the returned movies (4.4.2).

```java
// 1 - define filters
List<Bson> conditions = new ArrayList<>();
if (minRating != -1)
    conditions.add(gte("total_rating", minRating));

if (maxRating != -1)
    conditions.add(lte("total_rating", maxRating));

if (director != null && !director.isEmpty()){
    List<Bson> directorConditions = new ArrayList<>();
    for (String s:director.split(" ")){
        directorConditions.add(regex("directors.name", s, "i"));
    }
    conditions.add(and(directorConditions.toArray(new Bson[]{})));
}

if (actor  != null && !actor.isEmpty()){
    List<Bson> actorConditions = new ArrayList<>();
    for (String s:actor.split(" ")){
        actorConditions.add(regex("characters.actor_name", s, "i"));
    }
    conditions.add(and(actorConditions.toArray(new Bson[]{})));
}

if (country != null && !country.isEmpty())
    conditions.add(eq("countries", country));

if (fromYear != -1){
    conditions.add(gte("year", fromYear));
}

if (toYear != -1)
    conditions.add(lte("year", toYear));

if (genre != null && !genre.isEmpty())
    conditions.add(eq("genres", genre));

// 2 - and conditions
```

```
Bson filters;

if (!conditions.isEmpty())
    filters = and(conditions.toArray(new Bson[]{}));
else
    filters = new BsonDocument();

// 3 - make the query
FindIterable<Document> movieIterable = moviesCollection
    .find(filters)
    .sort(sorting)
    .projection(include("title", "year", "poster", "genres", "total_rating", "
    description"))
    .skip(n*(page-1))
    .limit(n);
```

4.3.2 Search

During a search, movies that (fuzzy) match the entire query are returned sorted by “likelyhood” (meta text score, in MongoDB). Results are paged as before.

If user is logged in, a separate query will be done to fetch the user’s ratings of the returned movies (refer to 4.4.2).

```
FindIterable<Document> movieIterable = moviesCollection
    .find(text("\""+ query + "\""))
    .projection(Projections.metaTextScore("score"))
    .sort(Sorts.metaTextScore("score"))
    .skip(n*(page-1))
    .limit(n);
```

4.3.3 Get Details

Given a movie id, return all movie details. Code snippet not reported for brevity’s sake.

4.3.4 Browse Statistics

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.

4.4 Ratings

4.4.1 CRUD operations

A rating can be added, read, updated or deleted. Related code snippets are not shown for brevity's sake.

After a rating is updated, the corresponding movie rating information are asynchronously updated (refer to 4.4.5).

4.4.2 Get all ratings for a user and list of movies

This particular operation is done to add the user's ratings to a query result set. First the list of movie ids of the movies in the query result is built and then all ratings of the given user to any of those movies is returned.

By building the array of identifiers, just one query to the database is necessary.

NB: the number of movies will always be small due to paging (20).

```java
// 1 - build list of ids of the movies I'm interested in
List<String> ids = new ArrayList<>();
for (Movie m:movies){
    ids.add(m.getId());
}

// 2 - fetch corresponding ratings from database
FindIterable<Document> ratingIterable = ratingsCollection.find(
    and(
        eq("_id.user_id", u.getId()),
        in("_id.movie_id", ids)
    )
);
```

4.4.3 Browse All

Admins can browse all ratings in the collection in descendig date order. Results are paged as seen in previous sections. Related code snippet is not shown for brevity's sake.

4.4.4 Browse User Ratings

Users can browse all their ratings in descendig date order. Results are paged as seen in previous sections. Related code snippet is not shown for brevity's sake.

4.4.5 Update Internal Rating

After a rating is changed, the average internal rating and the total rating of the movie should be updated. This is done asynchronously after the update in order not to make the client wait for this additional operation.

The update takes in input the old and the new rating (one of them can be null in case of insertion or delition) and consists in three parts:

1. fetch the movie from the DB
2. build the update request

(a) Update *internal rating*. The *internal rating* nested document may be missing, in which case we need to push it. Otherwise, addition, deletion or modification must be correctly handled by incrementing `sum` and `count` appropriately.

(b) Update the *total rating* by calculating the new average. If no rating is present any longer, the *total rating* is unset.

NB: the internal rating update is protected from concurrent modifications since it uses the atomic update operation `$inc` to update the sum and the count. Furthermore, note also that all other aggregate ratings do not have the sum attribute but drectly the average attribute since they are scraped. On the other hand, the update of the total rating is not consistent since another asynchronous entity (e.g. scraper) may overwrite it with stale information. However, this is not a big issue since at maximum just a few ratings will be lost among a big pool (thousands and more) so the average does not change much.

```java
// 1 - Fetch movie from DB
Movie m = dba.getMovieDetails(movieId);

// 2- Build the update request
Bson match;
List<Bson> updates = new ArrayList<>();

// 2a - update internal rating

if (internalRating != null) { // there are previous ratings
    // match the related nested document
    match = and(
        eq("_id", movieId),
        elemMatch("ratings", new Document() // always present
            .append("source", "internal"))
    );

    if (oldRating == null) { // adding
        updates.add(inc("ratings.$.sum", newRating.getRating()));
        updates.add(inc("ratings.$.count", 1));
    } else if (newRating == null) { // deleting
        updates.add(inc("ratings.$.sum", -oldRating.getRating()));
        updates.add(inc("ratings.$.count", -1));
    } else { // changing
        updates.add(inc("ratings.$.sum", newRating.getRating() - oldRating.
    getRating()));
        updates.add(set("ratings.$.last_update", new Date()));
    }
    // [...] update internalRating
} else { // internal rating is missing and must be created
    match = eq("_id", movieId);
    // [...] create new internalRating
    updates.add(
        push("ratings", AggregatedRating.Adapter.toDBObject(internalRating))
    );
}

// 2b - update total rating
// [...] calculate total rating
if (ar_count > 0) {
    updates.add(set("total_rating", m.getTotalRating()));
} else { // if no rating source found
    updates.add(unset("total_rating"));
}

// 3 - execute query on DB
dba.getMoviesCollection().updateOne(match, combine(updates));
```

4.5 Users

4.5.1 Get Profile

The user profile is composed by his anagraphic information and his rating statistics. The first part is just a *find* operation. The second part uses an aggregation pipeline similar to the one used to find movie statistics with th difference that in the end results are sorted by descending rating and only first 3 results are shown to the user.

Note that this aggregation pipeline is, in a real scenario, very fast since a user will not have more than a few hundreds of ratings and ratings are indexed also by `user_id` (so the match stage is extremely fast).

```
// 1 - fetch User from DB
User u = User.Adapter.fromDBObject(
        usersCollection.find(
                eq("username", username)
        ).first()
);

// 2 - calculate statistics

for (String[] field:new String[][]{actors, directors, genres}) {
    // for each aggregation (actors, directors, genres) th pipeline is the
    // same so I just set the specific fields and reuse the same code in a
    // loop
    AggregateIterable<Document> iterable = ratingsCollection
        .aggregate(Arrays.asList(
            Aggregates.match(eq("_id.user_id", u.getId())),
            // [...] lookup, project, unwind, unwind, group
            // sort by rating
            Aggregates.sort(descending("avg_rating", "movie_count")),
            // get three most liked
            Aggregates.limit(3)
    ));

    // 3 - add result to user profile
    if (field == actors){
        u.setFavouriteActors(
            Statistics.Adapter.fromDBObjectIterable(iterable, Person.class));
    } else if (field == directors){
        u.setFavouriteDirectors(
            Statistics.Adapter.fromDBObjectIterable(iterable, Person.class));
    } else if (field == genres){
        u.setFavouriteGenres(
            Statistics.Adapter.fromDBObjectIterable(iterable, Genre.class));
    }
}
```

4.5.2 Ban

When a user is banned, a flag is set on its document and all his ratings are removed from the database (triggering the internal rating update). Code snippet is not reported for brevity's sake.

4.5.3 Search

User search works alike the movie search, taking advantage of a *text index*, sorting results by best match and paging the result. Code snippet is not reported for brevity's sake.

5 Authentication and replication

5.1 Configuration of the replica set

Replication has been achieved through the set up of a MongoDB replica set in a remote cluster of 3 virtual machines provided by the University of Pisa.

A configuration document has been written, specifying the id of the replica set and the addresses of the machines. Additionally, a location tag was included. This could be useful if, in the future, the application is deployed to multiple datacenters for a more fine-grained control over the database requests (e.g. writes could be applied to at least one machine in every datacenter before returning).

```
rsconf = {
    _id: "replicaset0",
    members: [
    {_id: 0, host: "172.16.1.251:27017", tags: { location: 'unipi'} },
    {_id: 0, host: "172.16.1.162:27017", tags: { location: 'unipi'} },
    {_id: 0, host: "172.16.1.195:27017", tags: { location: 'unipi'} }
    ]
}
```

By default MongoDB binds only to localhost, to allow connection through the network the `mongod` daemon was started with the `--bind_ip` flag.

```
mongod  --replSet replicaset0
        --dbpath ~/data/rs_db
        --oplogSize 200
        --bind_ip 172.16.1.162
```

5.2 Authentication

In order to keep our system secure, authentication must be setup. A user responsible of handling our database, *moviedb*, was created from the mongo shell.

```
db.createUser(
    {
        user: 'lsmsdb',
        pwd: '******',
        roles: [ { role: "readWrite", db: "moviedb" } ]
    }
)
```

This is not enough since particular operations, like adding or removing new nodes to the replica set, require special privileges. The solution we choose was to create a superuser:

```
db.createUser(
    {
        user: "lsmsdb_admin",
        pwd: "******",
        roles: [ {role: "root", db: 'admin' }],
        passwordDigestor : "server"
    }
)
```

The `passwordDigestor` is an optional parameter which allows us to specify who has to digest the password (client or server). Since we use the SCRAM-SHA-256 mechanism, which is not compatible with the client, the server choice was forced.

Additionally, to let the MongoDB instances authenticate with each other, a keyfile was created and shared among the replica set.

```
openssl rand -base64 756 > ~/keyfile
chmod 400 ~/keyfile
```

After the users have been created and the keyfile shared, we run the MongoDB daemon with the authentication enabled, adding the `--auth` and `--keyFile` flags:

```
mongod  --replSet replicaset0
        --dbpath ~/data/rs_db
        --oplogSize 200
        --bind_ip 172.16.1.162
        --auth
        --keyFile ~/keyfile
```

All the connections from our Spring Boot server will now have to be authenticated with the *lsmsdb* user we just created, while the administration of the cluster or of its node is to be done with *lsmsdb_admin*.

5.3 Consistency, Availability and Partition tolerance

The CAP theorem states that only two between consistency, availability and partition tolerance could be achieved in a database: our main service, providning movies infos and ratings, could work well even if with outdated data. Movie's description or title rarely chages while ratings should have a low variability: it is therefore the most convenient choise to increase our availability and partition tolerance. This is not true for what concerns logins and accounts in general: e.g a password change should be set in stone or the user creation should grant login consistency. A bad user experience is the consequence of not handling correctly this aspects.

Different handling of the write and read requests depending on the data we want could be a reasonable solution. The drivers connecting to the database can specify a WriteConcern and a ReadPreference: the first determines the number of members of the replica set that should have written the new data before a positive response is sent to the driver, the second option is about the possibility of reading from nodes other than the master. The default configuration of MongoDB is a WriteConcern equal to 1 (only the primary server is updated) and a ReadPreference of 'primary' (we can read only from the primary). This means that, in case of a network partition excluding a number of nodes less than the majority and including the master, thus with the election of a new primary, consistency is not guaranteed since a write could have been made just before the partitioning (and the read would work just fine from the newly elected master).

What has been chosen, as already anticipated, is a differentiated approach depending on the request. To access the `users` collection, the driver establishes three different connections with the database.

```java
mongoClient = MongoClients.create(connectionURI);
database = mongoClient.getDatabase(dbName);

moviesCollection = database.getCollection("movies").withReadPreference(
    ReadPreference.nearest());

usersCollection = database.getCollection("users").withReadPreference(ReadPreference
    .nearest());

usersCollectionMajorityWrite = database.getCollection("users").withWriteConcern(
    WriteConcern.MAJORITY);

usersCollectionPrimaryRead = database.getCollection("users").withReadPreference(
    ReadPreference.primary());

ratingsCollection = database.getCollection("ratings").withReadPreference(
    ReadPreference.nearest());
```

`usersCollection` will be used for all the requests where we want high availabillity, like showing up the profile page or finding the user from the admin control page.
`usersCollectionMajorityWrite` handles all the requests for which we want the updates to be permanent (e.g. password changes, new session id etc.): a write concern of 'majority' (which is computed against the total nodes in the replica set) protects us against network partition since no matter what happens to the nodes, as long as a majority is reached, the election system will make sure the successive primary node will have our write. Since the secondaries could or could not have it, when we want consistency we have to read only from the primary. Therefore, the `usersCollectionPrimaryRead` will set the correct ReadPreference and will be used accordingly.

For what concerns the other requests, consistency is not strictly necessary and we would like our system to be highly available. For this reason, we keep the default write concern (equal to 1) and set the read preference to 'nearest', thus allowing reads from secondaries (which could have outdated informations).

5.4 Test and evaluation

5.4.1 Election

A test of the election mechanism has been done,

```
2020–05–16T19:55:36.242+0100 I  ELECTION [replexec–0] Starting an election, since we've seen no PRIMARY in the past 10000ms
2020–05–16T19:55:36.242+0100 I  ELECTION [replexec–0] conducting a dry run election to see if we could be elected. current term: 41
2020–05–16T19:55:36.242+0100 I  REPL     [replexec–0] Scheduling remote command request for vote request: RemoteCommand 305 –– target:172.16.1.162:27017 db: admin cmd:{ replSetRequestVotes: 1, setName: "replicase
t0", dryRun: true, term: 41, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp(1589655321, 1), t: 41 } }
2020–05–16T19:55:36.242+0100 I  REPL     [replexec–0] Scheduling remote command request for vote request: RemoteCommand 306 –– target:172.16.1.195:27017 db: admin cmd:{ replSetRequestVotes: 1, setName: "replicase
t0", dryRun: true, term: 41, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp(1589655321, 1), t: 41 } }
2020–05–16T19:55:36.242+0100 I  ELECTION [replexec–5] VoteRequester(term 41 dry run ) failed to receive response from 172.16.1.162:27017: HostUnreachable: Error connecting to 172.16.1.162:27017 :: caused by :: C$nnection refused
2020–05–16T19:55:36.244+0100 I  ELECTION [replexec–2] VoteRequester(term 41 dry run ) received a yes vote from 172.16.1.195:27017; response message: { term: 41, voteGranted: true, reason: "", ok: 1.0, $clusterTi$e: { clusterTime: Timestamp(1589655321, 1), signature: { hash: BinData(0, 0000000000000000000000000000000000000000), keyId: 0 } }, operationTime: Timestamp(1589655321, 1) }
2020–05–16T19:55:36.244+0100 I  ELECTION [replexec–2] dry election run succeeded, running for election in term 42
2020–05–16T19:55:36.250+0100 I  REPL     [replexec–2] Scheduling remote command request for vote request: RemoteCommand 307 –– target:172.16.1.162:27017 db: admin cmd:{ replSetRequestVotes: 1, setName: "replicas$t0", dryRun: false, term: 42, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp(1589655321, 1), t: 41 } }
2020–05–16T19:55:36.251+0100 I  REPL     [replexec–2] Scheduling remote command request for vote request: RemoteCommand 308 –– target:172.16.1.195:27017 db: admin cmd:{ replSetRequestVotes: 1, setName: "replicas$t0", dryRun: false, term: 42, candidateIndex: 0, configVersion: 6, lastCommittedOp: { ts: Timestamp(1589655321, 1), t: 41 } }
2020–05–16T19:55:36.251+0100 I  ELECTION [replexec–5] VoteRequester(term 42) failed to receive response from 172.16.1.162:27017: HostUnreachable: Error connecting to 172.16.1.162:27017 :: caused by :: Connectio$ refused
2020–05–16T19:55:36.265+0100 I  ELECTION [replexec–3] VoteRequester(term 42) received a yes vote from 172.16.1.195:27017; response message: { term: 42, voteGranted: true, reason: "", ok: 1.0, $clusterTime: { cl$sterTime: Timestamp(1589655321, 1), signature: { hash: BinData(0, 0000000000000000000000000000000000000000), keyId: 0 } }, operationTime: Timestamp(1589655321, 1) }
2020–05–16T19:55:36.265+0100 I  ELECTION [replexec–3] election succeeded, assuming primary role in term 42
2020–05–16T19:55:36.265+0100 I  REPL     [replexec–3] transition to PRIMARY from SECONDARY
2020–05–16T19:55:36.265+0100 I  REPL     [replexec–3] Resetting sync source to empty, which was :27017
2020–05–16T19:55:36.266+0100 I  REPL     [replexec–3] Entering primary catch–up mode.
2020–05–16T19:55:36.268+0100 I  REPL_HB  [replexec–5] Heartbeat to 172.16.1.162:27017 failed after 2 retries, response status: HostUnreachable: Error connecting to 172.16.1.162:27017 :: caused by :: Connection $efused
```

```
19 2020−05−16T19:55:36.268+0100 I  REPL     [replexec−5] Caught up to the latest
    optime known via heartbeats after becoming primary. Target optime: { ts:
    Timestamp(1589655321, 1), t: 41 }. My Last Applied: { ts: T$mestamp(1589655321,
    1), t: 41 }
20 2020−05−16T19:55:36.268+0100 I  REPL     [replexec−5] Exited primary catch−up mode.
21 2020−05−16T19:55:36.268+0100 I  REPL     [replexec−5] Stopping replication producer
22 2020−05−16T19:55:36.268+0100 I  REPL     [ReplBatcher] Oplog buffer has been
    drained in term 42
23 2020−05−16T19:55:36.268+0100 I  REPL     [ReplBatcher] Oplog buffer has been
    drained in term 42
24 2020−05−16T19:55:36.268+0100 I  REPL     [RstlKillOpThread] Starting to kill user
    operations
25 2020−05−16T19:55:36.269+0100 I  REPL     [RstlKillOpThread] Stopped killing user
    operations
26 2020−05−16T19:55:36.269+0100 I  REPL     [RstlKillOpThread] State transition ops
    metrics: { lastStateTransition: "stepUp", userOpsKilled: 0, userOpsRunning: 0 }
27 2020−05−16T19:55:36.269+0100 I  REPL     [rsSync−0] transition to primary complete;
     database writes are now permitted
```

As we can see, it worked flawlessly: no primary was seen for 10 seconds, this triggered an election (or, to be precise, a dry run followed by a *real* run). The candidate node was `172.16.1.251` and received a positive vote from `172.16.1.195`, while `172.16.1.162` was unreachable (since it was down). A majority (2 out of 3) was reached and the secondary node transitioned to primary. It then caught up with the latest optime (row 19), cleaned the oplog and killed all the current user operations (row 25).

From the time of the dry run for election (19:55:36 and 242ms), which is 10 seconds after the last heartbeat in our case, to the new primary being up and running (19:55:36 and 269ms) only 27ms were elapsed: this is possible since the three nodes are connected to the same LAN, but certifies how the election mechanism is fast and efficient.