Università di Pisa

Large-Scale and Multi-structured Databases - Project: Rotten Movies

Fabio Piras, Giacomo Volpi, Guillaume Quint

Academic Year: 2022/2023

Contents

Chapter 1

Introduction

Rotten Movies is a web service where users can keep track of the general liking for movies, they can also share their thoughts with the world after signing up. In addition users can follow renowned top critic to be constantly updated with their latest reviews.

Guest users, as well as all others, can browse or search movies based on filters like: title, year of release and personnel who worked in it. They can also view a Hall of Fame for the most positive review genres, production houses and years in which the best movies were released.

For this project, the application has been developed in Java with the Spring framework and Thymeleaf as templating engine to implement a web GUI. The application uses a document database to store the main information about movies, users, reviews and personnel; a graph database is instead used to keep track of the relationship between normal users and top critics in terms of who follows who and between the movies and the users who reviewed it.

Chapter 2

Feasibility Study

The very first step was to perform a look up for what type of data we needed to create the application and how to handle it by performing a feasibility study.

1 Dataset analysis and creation

Initially we searched online for available dataset, we ended up with six different files coming from Kaggle and imdb with a combined storage of 4.18 GB:

- title_principal.tsv
- name_basic.tsv
- title_basic.tsv
- title_crew.tsv
- poster_URL.csv
- rotten_movies.csv
- rotten_reviews.csv

The first three were found on the imdb site containing general data about the title of the entries, type (not all were movies), cast, crew and other useful information. The fourth one came from Kaggle, and contains movies posters urls. The last two also came from Kaggle and they contain data scraped from the rotten tomatoes site regarding movie rating and their reviews. All of these dataset were organized in a relational manner.

Initial steps and creation of the movie collection After a general look it was clear that we needed to process the data to obtain a lighter dataset by deciding what to keep based on our needs and specifications; we decided to use Python as programming language to trim the original files, this was also achieved through the use of Google Colab.

We started by taking all the tsv files and transforming them into a single file, we performed a join operation on all of them based on their id and then discarded the useless information (Appendix **??**). After that we performed the same operation on the csv files coming from Rotten Tomatoes (Appendix **??**). We then proceeded to join the two files based on the title of the movies (Appendix **??**).

Unfortunately we noticed that after the join there were more rows for the same movie, in fact, due to the relational nature of the first dataset, we had a different entry for personnel on each row, so we rollback to before the join, but this time we collapsed all the information in a single row (Appendix **??**).

Due to the variability of the data in the string fields, like the content of the reviews, we decided to perform an escape on various elements to avoid crashes and failures during the import into the DBSM (Appendix **??**). We also had to do a similar process of normalization for the date fields (Appendix **??**)

Finally we achieved our goal of having a JSON file for the movie collection, the file occupied a space of 270MB.

Creation of the user collection The following step was to generate a collection for the user. This was achieved by starting from the movie collection: for each different review author we generated a document, later to be placed in a single JSON file of which the total storage size was 86,6 MB. This step was performed directly on the Mongo shell (Appendix **??**).

2 Analysis result

With the dataset for the document database ready we finally had a better understanding on how to shape the models and the relative functionality in the application code. We will discuss later of the design of the collections and the methods used to interact with them. The same goes for the graph database of which the structure was heavily influenced by the one in the MongoDB

Chapter 3

Design

1 Main actors

As already mentioned in the introduction we have mainly three actors: guest users and registered users. The latter can be divided between normal users and top critics. In addition there is an admin actor whose main role is to oversee the entire service.

2 Functional requirements

This section describes the functional requirements that need to be provided by the application in regards of the actors:

- Guest (Unregistered) Users can:
  - login/register into the service
  - search movies via the search bar and other filters
  - view movies, their details and relative reviews
  - view the personal page of the author of a selected *top critic* review
  - view the different halls of fame
- Normal users can:
  - logout from the service
  - search movies via the search bar and other filters
  - view movies, their details and relative reviews
  - write a review for a selected film
  - view the personal page of the author of a selected *top critic* review
  - view the different halls of fame
  - follow/unfollow a *top critic* user
  - view the feed of latest reviews from the followed *top critics*
  - view a suggestion feed for *top critics* to follow
  - view the history of its own reviews
  - modify its own reviews
  - delete its own reviews

  - change its account information

- Top Critics can:
  - logout from the service
  - search movies via the search bar and other filters
  - view movies, their details and relative reviews
  - write a top critic review for a selected film
  - view the different halls of fame
  - view the history of its own top critic reviews
  - modify its own top critic reviews
  - delete its own top critic reviews
  - change its account information
  - see the number of its followers

- Admin users can:
  - logout from the service
  - search movies via the search bar and other filters
  - view movies, their details and relative reviews
  - view the different halls of fame
  - modify film details
  - add/remove films
  - browse *users* and *top critics*
  - ban *users* and *top critics*
  - register new *top critics*
  - perform analytics on the user base

3 Non functional requirements

The following section lists non functional requirements for the application.

- The system must encrypt users' passwords
- The service must be built with the OOP paradigm
- The service must be implemented through a responsive website
- Avoidance of single point of failure in data storage
- High availability, accepting data displayed temporarily in an older version

4 Implementation regarding the CAP theorem

We will now discuss on how we decided to tackle the CAP theorem issue. We determined that the application is a read-heavy one where we expect that the number of read transactions are by far more frequent than write operations, so we decided that our application should prioritize high availability, low latency and should be capable of withstanding network partitions. In reference of figure 3.1 it is clear that we moved towards a AP approach in spite of temporarily data inconsistency.

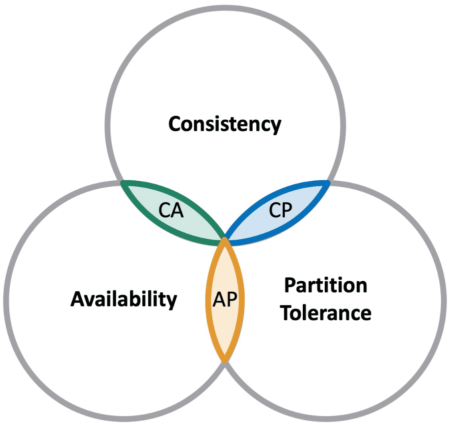
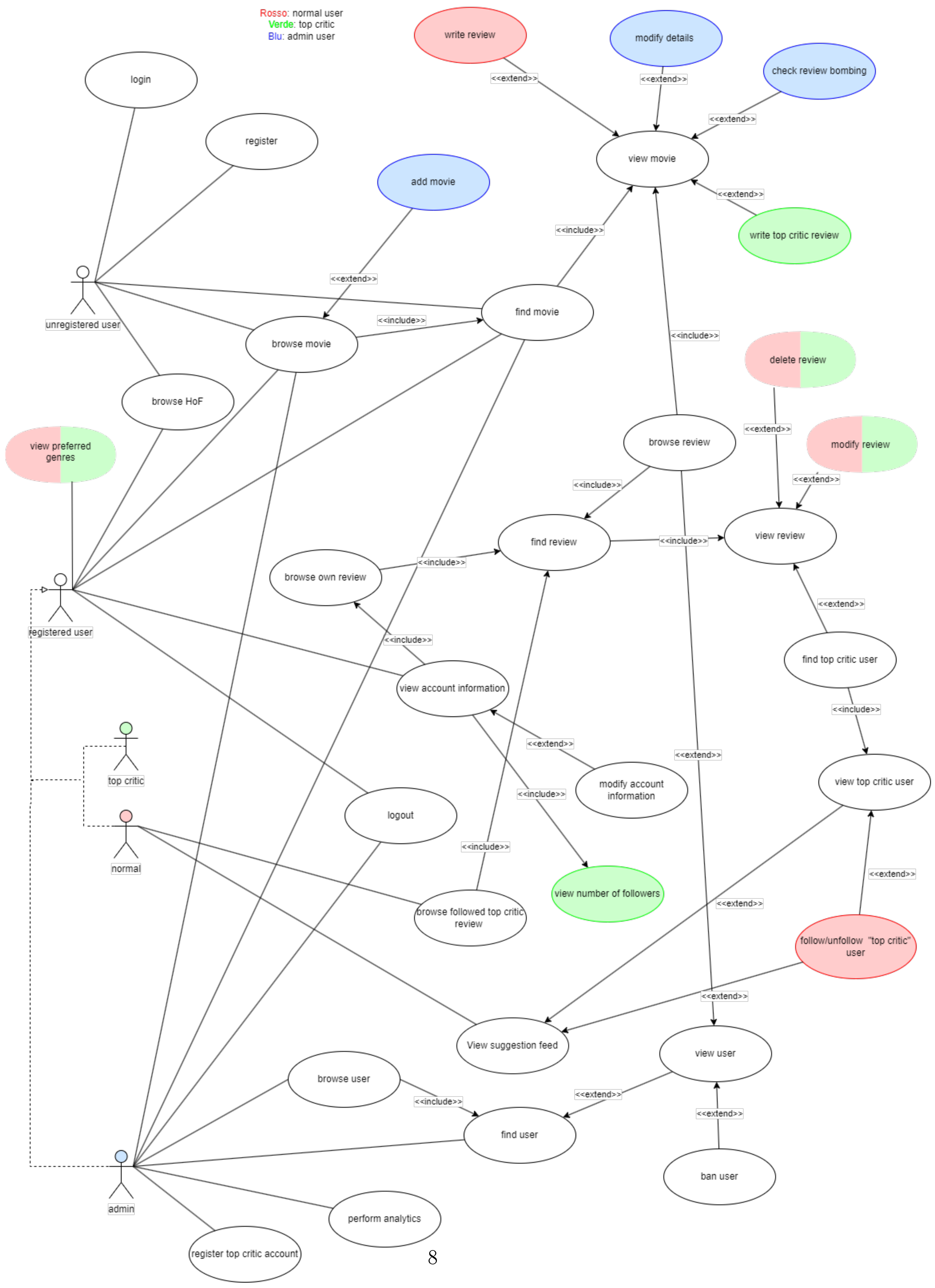


Figure 3.1: CAP theorem diagram

In order to guarantee the requirements of high availability, we decided to accept the cases in witch data shown to the user could be not up to date to the latest version in the database.

5 Use cases


Rosso: normal user
Verde: top critic
Blu: admin user
login
register
add movie
write review
modify details
check review bombing
view movie
write top critic review
unregistered user
find movie
browse movie
browse HoF
view preferred genres
delete review
modify review
browse review
find review
view review
browse own review
registered user
view account information
find top critic user
view top critic user
top critic
normal
modify account information
logout
view number of followers
browse followed top critic review
follow/unfollow "top critic" user
View suggestion feed
view user
browse user
find user
ban user
admin
perform analytics
register top critic account
<<extend>>
<<include>>

The *perform analytics* use case groups all of the following direct use cases:

- Group user base by age
- Most active users and top critics
- Most followed top critics

6 Class analysis

In this section we shall discuss the design of the various classes and how they relate to each other

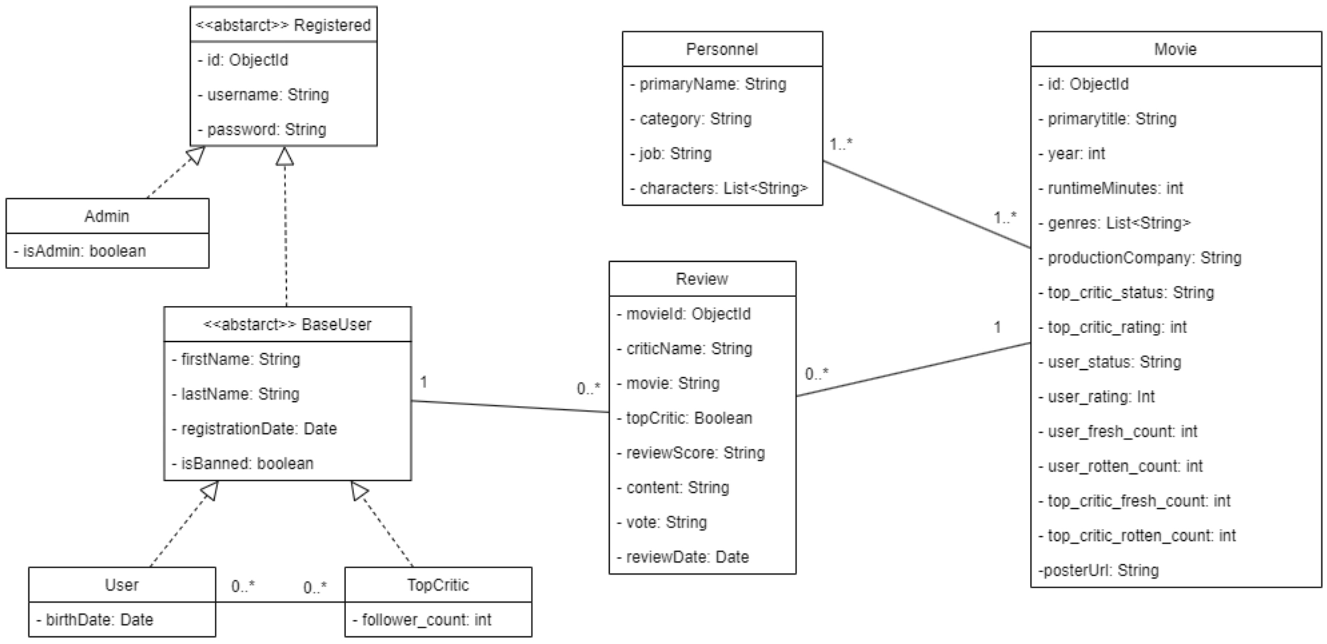


Figure 3.3: Class Diagram

The diagram expresses the following relationships between the entities.

- a BaseUser can write from zero to many reviews
- a Review can be written by a single BaseUser for a single Movie
- a Movie can have zero to many reviews and can have from 1 to many Personnel working in it
- a Personnel can work in 1 to many movies

Chapter 4

Document database

In this chapter we will discuss the organization of the document database and how we handled the replicas. We decided to use MongoDB as DBMS for the document database for the purpose of storing the main information for movies, users, reviews and personnel. In MongoDB we created the following two collections:

- movie
- user

Inside the movie collection are embedded the documents for the reviews and personnel

1 Collection composition

1 .1 Movie collection

The following is the composition of a movie document in the collection.

```
{
    "_id" : ObjectId(<<id_field>>),
    "primaryTitle": "The first ever movie",
    "year": 1989,
    "runtimeMinutes": 70,
    "genres": ["Crime", "Drama", "Romantic"],
    "productionCompany": "Dingo Picture Production" ,
    "personnel": [
        {
            "name": "John Doe",
            "category": "producer",
            "job": "writer"
        },
        {
            "name": "Christopher Lee",
            "category": "actor",
            "character": ["The one"]
        },
        ...
    ],
    "top_critic_fresh_count": "4",
    "top_critic_rotten_count": 0,
    "user_fresh_count": 14,
    "user_rotten_count": 1,
    "top_critic_rating": 100,
    "top_critic_status": "Fresh",
    "user_rating": 93,
    "user_status": "Fresh",
    "review":
    [
        {
            "critic_name":    "AntonyE",
            "review_date": "2018-12-07",
            "review_type": "Rotten",
            "top_critic":  false,
            "review_content":"I really didn t liked it!",
            "review_score":   "1/10"
        },
        {
            "critic_name":    "AntonyE",
            "review_date": "2015-12-07",
            "review_type": "Fresh",
            "top_critic":  true,
            "review_content":"I really liked it!",
            "review_score":   "A-"
        },
        ...
    ]
}
```

Listing 4.1: Test

Movie document

As previously stated the field personnel and review are arrays of embedded documents.

1 .2 User collection

The following is the composition of a user document in the collection.

```
{
    "_id"                   : ObjectId(<<id_field>>),
    "username"              :"AntonyE",
    "password"              :"hashed_password",
    "firstName"             :"Anton",
    "lastName  "            :"Ego",
    "registration_date"      :"2019-06-29",
    "date_of_birth"         :"2002-07-16",
    "last3Reviews":
        [
            {
                "_id"                   : ObjectId(<<id_field>>),
                "primaryTitle"          : "Star Wars: A new Hope",
                "review_date"           : "2018-12-07",
                "review_type"           : "fresh",
                "top_critic"            : false,
                    "review_content"     : "I really liked it!",
                    "vote"               : "8/10"
                },
                {
                    "_id"                : ObjectId(<<id_field>>),
                    "primaryTitle"       : "Ratatouille",
                    "review_date"        : "2019-02-17",
                    "review_type"        : "rotten",
                    "top_critic"         : false,
                    "review_content"     : "The director was also
    ↪ controlled by a rat",
                    "vote"               : "D+"
                },
                {
                    "_id"                : ObjectId(<<id_field>>),
                    "primaryTitle"       : "300",
                    "review_date"        : "2020-02-15",
                    "review_type"        : "fresh",
                    "top_critic"         : false,
                    "review_content"     : "Too much slow motion",
                    "vote"               : "3.5/5"
                }
            ],
        "reviews":[
            {
                "movie_id" : ObjectId(<<id_field>>),
                "primaryTitle" : "Evidence",
                "review_index": 11
            },
            ...
        ]

}
```

Listing 4.2: Test

User document

It is important to notice that in this collection we have two fields for the reviews, *last3Reviews* and *reviews* which present different ways of storing data for the same underling entity. As suggested by the name itself *last3Reviews* contains the last three review in the form of an embedded document meanwhile *reviews* contains all the reviews made by a user in the form of document linking towards the movie for which they were written; in particular, the link is formed by the id of the movie, the title and the position in which the review can be found in the array *review* of the movie document. This structure was chosen so that we could avoid a full replica of the reviews in both collections and, at the same time, avoiding to perform join operation for those reviews that are more frequently checked, which are the most recent ones. The idea of creating a separated collection for reviews was immediately discarded because it would have implied a design for the document database that resembles a third normal form.

2 Indexes

In order to provide the best execution speed in search queries we use indexes. In particular we focus on the application part that is available also without registering to the site, like Hall of

Fame and search movies functionalities. Without indexes we need a collection scan in user and movie collections to find the right document.

2 .1 Movie collection

- primaryTitle
- year
- genres
- top_critic _rating
- user_rating

```
// Dimension is expressed in Kb
{
  _id_: 228,
  primaryTitle_1: 292,
  year_1: 92,
  genres_1: 180,
  'personnel.primaryName_1': 1680,
  top_critic_rating_1: 80,
  user_rating_1: 80
}
```

Listing 4.3: Test

Movie collection indexes

2 .2 User collection

- username
- date_of _birth

```
// Dimension is expressed in Kb

{ _id_: 180, username_1: 168, date_of_birth_1: 100 }
```

Listing 4.4: Test

User collection indexes

We decided to use these indexes because they provide a jump in term of speed in search queries without occupying much space. We accept the fact that writes are slower because our application is read-intesive. We consider the idea of using an index on personnel.primaryName but the cost in term of space was higher than the potential benefit, so we discarded it. For a full analytic report for the indexes, see appendix. (Appendix ??)

3 Partition and replicas

To ensure high availability we deploy a cluster of replicas (3). We install and configure MongoDB on all machines, with these priorities

1. 172.16.5.26 (primary)
2. 172.16.5.27 (secondary)
3. 172.16.5.28 (secondary)

We decided to use **nearest read preference** and **W2 write preference**. In fact we can tolerate that users see temporarily an old version of data with a 33% chance.

4 Aggregations

In this section we shall discuss the different aggregations that the application will implement, the values between «» represent a value passed by an above level.

4 .1 Return the best years by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {$group:
        {
        _id: "$year",
        topCriticRating:{$avg:"$top_critic_rating"},
        userRating:{$avg:"$user_rating"},
        count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{[topCriticRating/userRating]:-1}},
    {$limit: <<j>>}
])
```

Listing 4.5: Test

best$_year.js$

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.group("$year",
                                avg("top_critic_rating", "$top_critic_rating"
    ),
                                avg("user_rating", "$user_rating"),
                                sum("count",1)),
                        Aggregates.match(gte("count",numberOfMovies)),
                        Aggregates.sort(opt.getBsonAggregationSort()),
                        Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                )
        );
```

Listing 4.6: Test

MovieMongoDB$_DAO.java$

Return the best genres by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {
        $unwind:"$genres"
    },
    {$group:
        {
            _id: "$genres",
            topCriticRating:{$avg:"$top_critic_rating"},
            userRating:{$avg:"$user_rating"},
            count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{[topCriticRating/userRating]:-1}},
    {$limit:<<j>>}
])
```

Listing 4.7: Test

best$_g$*enres.js*

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.unwind("$genres"),
                        Aggregates.group("$genres",
                                avg("top_critic_rating", "$top_critic_rating"
    ),
                                avg("user_rating", "$user_rating"),
                                sum("count",1)),
                        Aggregates.match(gte("count",numberOfMovies)),
                        Aggregates.sort(opt.getBsonAggregationSort()),
                        Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                )
        );
```

Listing 4.8: Test

MovieMongoDB$_D$*AO.java*

Return the best production houses by top critic and user ratings

Mongo shell

```
db.movie.aggregate([
    {$group:
        {
            _id: "$production_company",
            topCriticRating:{$avg:"$top_critic_rating"},
            userRating:{$avg:"$user_rating"},
            count:{$sum:1}
        }
    },
    {$match:{count:{$gte:<<i>>}}},
    {$sort:{[topCriticRating/userRating]:-1},
    {$limit:<<j>>}
])
```

Listing 4.9: Test

best$_production_houses.js$

Java implementation

```
AggregateIterable<Document> aggregateResult = collection.aggregate(
                    Arrays.asList(
                              Aggregates.group("$production_company",
                                        avg("top_critic_rating", "$top_critic_rating"
    ),
                                        avg("user_rating", "$user_rating"),
                                        sum("count",1)),
                              Aggregates.match(gte("count",numberOfMovies)),
                              Aggregates.sort(opt.getBsonAggregationSort()),
                              Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                    )
          );
```

Listing 4.10: Test

MovieMongoDB$_DAO.java$

Given a movie count the review it has received by each month

Mongo shell

```
db.movie.aggregate([
    {
        $match:{id:<<"id">>}
    },
    {$unwind:"$review"},
    {
        $group:
        {
            _id:{year:{$year:"$review.review_date"}, month:{$month:"$review.
    review_date"}},
            count:{$sum:1}
        }
    },
    {$sort:{_id:1}}

])
```

Listing 4.11: Test

movie$_count.js$

Java implementation

```
Document yearDoc = new Document("year",new Document("$year","$review.
    review_date"));
```

```
        Document monthDoc = new Document("month",new Document("$month","
    $review.review_date"));
        ArrayList<Document> test=new ArrayList<>();
        test.add(yearDoc);
        test.add(monthDoc);
        AggregateIterable<Document> aggregateResult = collection.aggregate(
                Arrays.asList(
                        Aggregates.match(eq("_id",id)),
                        Aggregates.unwind("$review"),
                        Aggregates.group(test,
                                sum("count",1)),
                        Aggregates.sort(Sorts.ascending("_id"))
                )
        );
```

Listing 4.12: Test

MovieMongoDB$_DAO.java$

Given a user count the review he/she has made divided by genres

Mongo shell

```
db.movie.aggregate([
    {$match:
        {"review.critic_name":{$eq:<<"name">>}}
    },
    {$unwind:"$genres"},
    {$group:{_id:"$genres", count:{$sum:1}}},
    {$sort:{count:-1}},
    {$limit: <<n>>}
])
```

Listing 4.13: Test

genre$_count.js$

Java implementation

```
AggregateIterable<Document> aggregateResult = collectionMovie.aggregate(
                Arrays.asList(
                        Aggregates.match(eq("review.critic_name",username)),
                        Aggregates.unwind("$genres"),
                        Aggregates.group("$genres",
                                sum("count",1)),
                        Aggregates.sort(Sorts.descending("count")),
                        Aggregates.limit(Constants.
    HALL_OF_FAME_ELEMENT_NUMBERS)
                )
        );
```

Listing 4.14: Test

BaseUserMongoDB$_DAO.java$

Return the number of user divided by an age bucket

Mongo shell

```
db.user.aggregate([
    {
        $match:{"date_of_birth":{$exists:true}}
    },

    {
        $bucket:
        {
            groupBy: {$year:"$date_of_birth"},
            boundaries: [<<values>>],
            output:
            {
                "population": {$sum:1}
            }
        }
    }
])
```

Listing 4.15: Test

user$_p$*opulation.js*

Java implementation

```
BucketOptions opt = new BucketOptions();
        ArrayList<Integer> buck=new ArrayList<>();
        opt.output(new BsonField("population",new Document("$sum",1)));
        int bucketYear=1970;
        buck.add(bucketYear);
        while(bucketYear<=2010){
            bucketYear=(bucketYear+offset);
            buck.add(bucketYear);
        }
        AggregateIterable<Document> aggregateResult = collectionUser.
    aggregate(
                Arrays.asList(
                        Aggregates.match(exists("date_of_birth")),
                        Aggregates.bucket(new Document("$year","
    $date_of_birth"),buck,opt)
                )
        );
```

Listing 4.16: Test

AdminMongoDB$_D$*AO.java*

5 Sharding considerations

We consider the possibility of sharding the movie collection, with the benefit of much less space occupation, using id as a possible sharding key. However we realized that we search movies not only by id but also with their primary titles, their years, etc. With these type of queries we end up in a query flooding scenario, where every replica must be consulted in order to find the appropriate document.

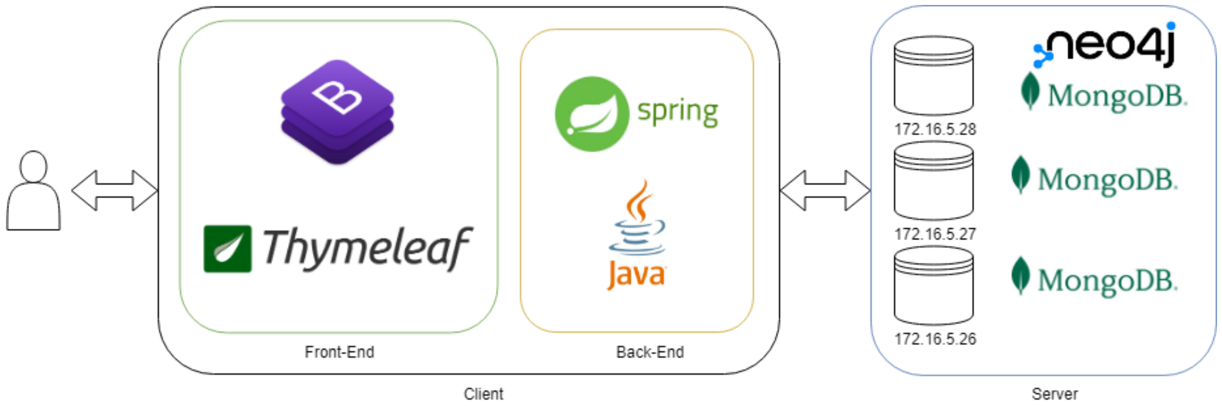
Chapter 5

Software Architecture

Rotten Movies is a web application developed in *Java* that implements the Model-View-Controller (MVC) paradigm through the use of the *Spring Framework*.

Specifically, the *view* layer is handled by the templating engine *Thymeleaf*, which enables the creation of custom web pages by interpolating data provided by the middle-ware with predefined html templates. Interface styling has been eased by the use of some predefined CSS classes defined in *Bootstrap 5.0*[1]

The back-end side of the application is supported by the document database *MongoDB* and the graph database *Neo4J*, which are accessed via the official mongo driver[2] and the neo4j driver[3] for Java, respectively



[1]https://getbootstrap.com/
[2]https://www.mongodb.com/docs/drivers/java/sync/current/
[3]https://neo4j.com/developer/java/

Chapter 6

Repository Structure