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.

All source files can be found in the public repository: `https://gitlab.com/ghi0m/rottenmovies`

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 A .1). After that we performed the same operation on the csv files coming from Rotten Tomatoes (Appendix A .2). We then proceeded to join the two files based on the title of the movies (Appendix A .3).

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 A .4).

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 B .1). We also had to do a similar process of normalization for the date fields (Appendix B .2)

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 B .3).

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.

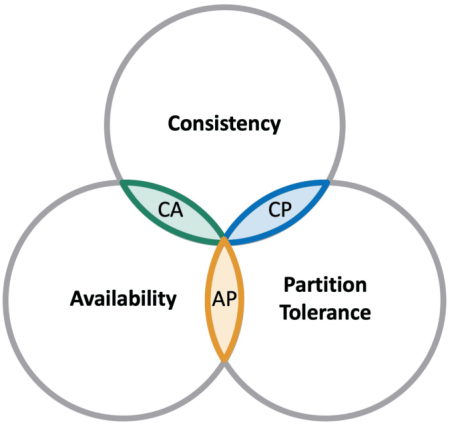
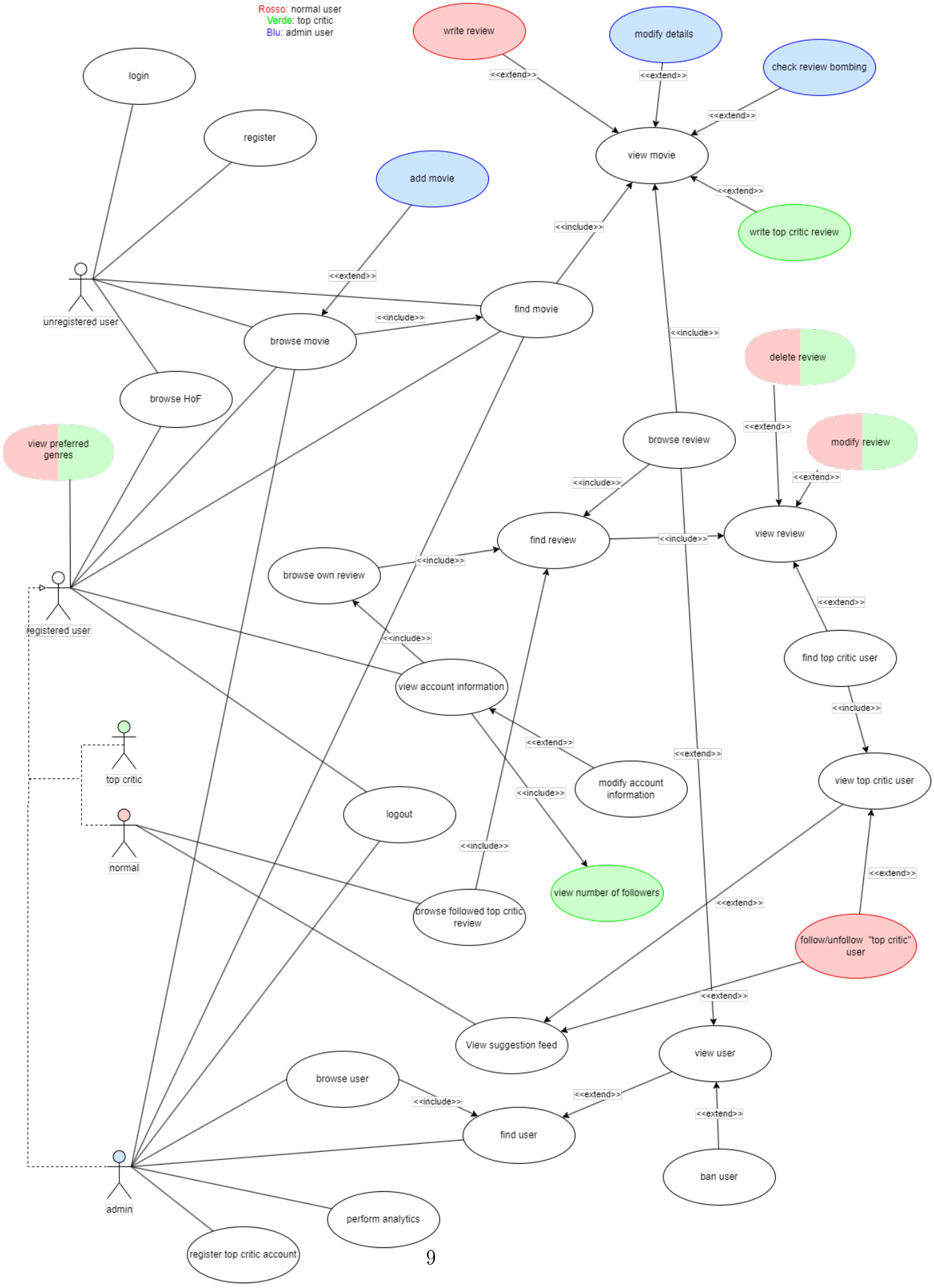


Figure 3.1: CAP theorem diagram

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
write review
modify details
check review bombing
login
register
view movie
add movie
write top critic review
unregistered user
find movie
browse movie
browse HoF
delete review
view preferred genres
browse review
modify review
find review
view review
browse own review
registered user
find top critic user
view account information
top critic
view top critic user
modify account information
logout
normal
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

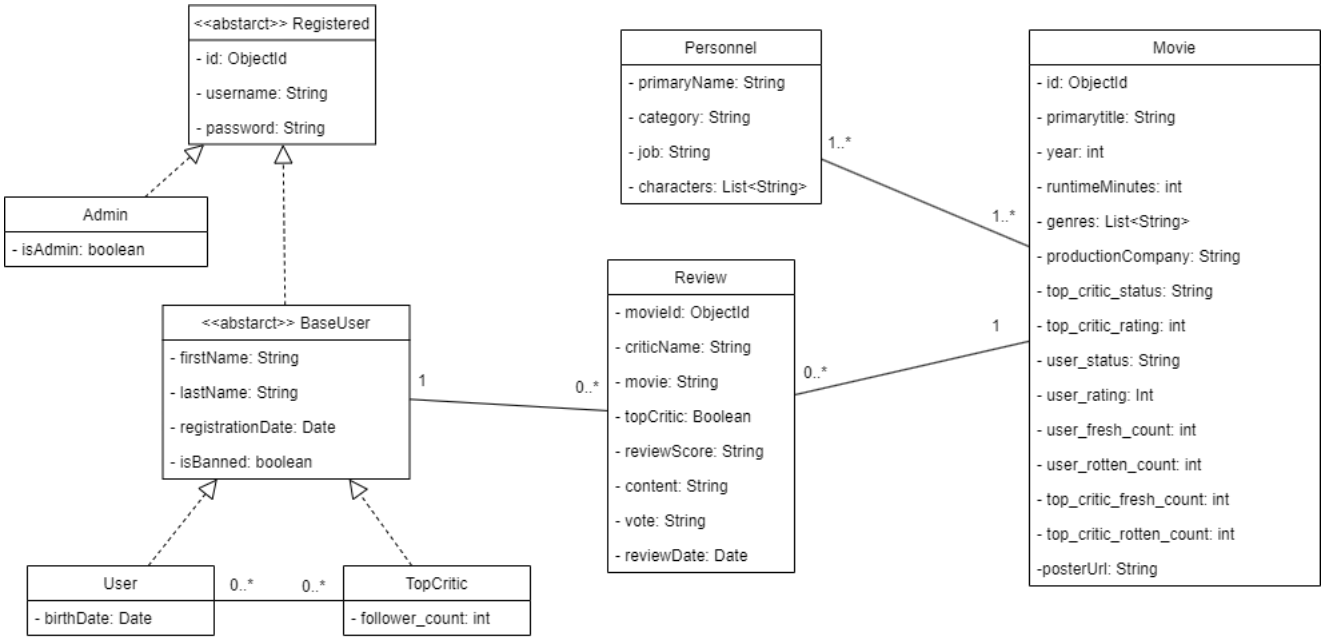


Figure 3.3: Class Diagram

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: 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: 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: 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: 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 C)

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: bestYear.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: MovieMongoDBDAO.java

4 .2 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: bestGenres.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: MovieMongoDBDAO.java

4 .3 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: bestProductionHouses.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: MovieMongoDBDAO.java

4 .4 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: movieCount.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: MovieMongoDBDAO.java

4 .5 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: genreCount.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: BaseUserMongoDBDAO.java

4 .6 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: userPopulation.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: AdminMongoDBDAO.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

Graph database

1 Structure of the graph

The DBSM chosen for the graph database is Neo4j, it is used to manege the social part of the site and also to keep track of the reviews made for the feed and suggestion functionality.

The entities used in the database are:

- User
- TopCritic
- Movie

The nodes themselves do not store a lot of data, we decided to keep the bare minimum by having the following attributes:

- for the User and TopCritic:
  - id
  - name
- for Movie:
  - id
  - title

The existing relationships are:

- User -[:FOLLOWS]-> TopCritic
- User -[:REVIEWED]-> Movie
- TopCritic -[:REVIEWED]-> Movie

In particular the *REVIEWED* relationship contains the following information:

- content excerpt
- date
- freshness:

Below is shown a snapshot of the graph database taken from Neo4j.

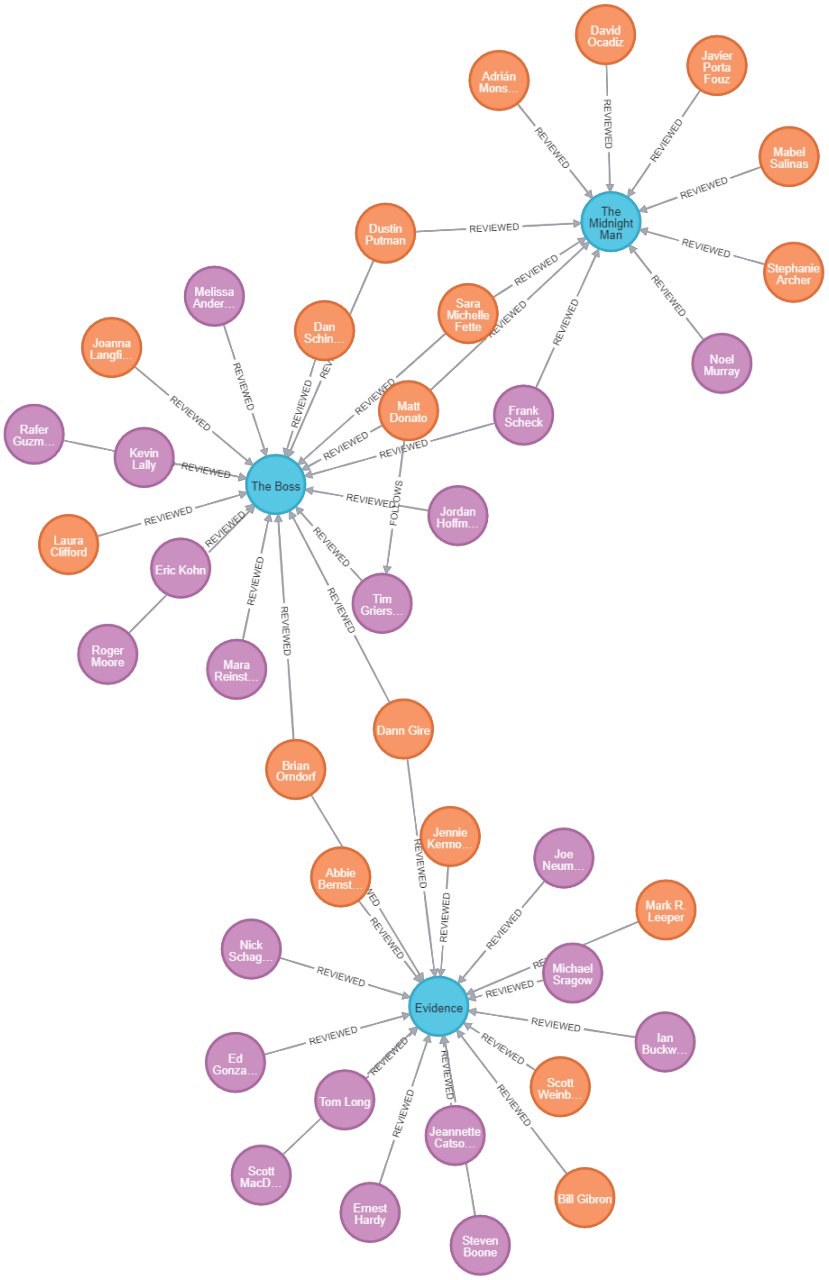


Figure 5.1: graph DB

As previously stated in the Feasibility study section 2 , the graph database design was heavily influenced by the document one, it was in fact generated by a Python script (Appendix A .6) that starts from the movie collection and, after generating the nodes for the *Movie* entity, does the same for the user by distinguishing them between normal users and top critics. It then generates the *REVIEWED* relationship based upon the data stored in the document database and finally it generates the *FOLLOWS* relationship randomly, the total storage size of the database is 89 MB.

2 Index

The *id* attributes are common to all nodes and they store a string which is also used as id by the document database in the *_id* field. This choice was made so that we could have a way to identify the same object across the different databases with only one string from the application. Due to the OOP language used in the application the data was manipulated with classes containing the id of the object so, to improve the performance of the graph database we decided to use the *id* attribute as an index.

Below is the information in a json format from the Neo4j DBMS about the new index (some attributes were omitted for space)

```
[
  {
    "id": 3,
    "name": "movie_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "Movie"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 5,
    "name": "topcritic_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "TopCritic"
    ],
    "properties": [
      "id"
    ]
  },
  {
    "id": 4,
    "name": "user_id_index",
    "type": "RANGE",
    "entityType": "NODE",
    "labelsOrTypes": [
      "User"
    ],
    "properties": [
      "id"
    ]
  }
]
```

Listing 5.1: Neo4jID

We then proceed to profile the queries with and without the above indexes, here we can see the result of this analysis:

Figure 5.2: search by id without and with movie_id _index

Figure 5.3: search by id without and with user_id _index

3 Queries

In the following section we will show the queries that drove the design of the graph database.

Graph-Centric Query	Domain-Specific Query
Which are the User nodes with the most outgoing edges of type REVIEWED	Which are the non top critic users with most reviews made
Which are the TopCritic nodes with the most incoming edges of type FOLLOWS	Which are the most followed top critics
Which are the Movies nodes that have been most recently connected with TopCritic nodes by a REVIEWED relationship meanwhile the latter are connected to a User node with a FOLLOWS relationship	Which are the latest reviewed movies by followed top critics given a starting user
Which are the User and TopCritic nodes that have outgoing REVIEWED edges to the same Movie node without having a FOLLOWS edge between them, which of these TopCritic nodes have more similar attribute on the REVIEWED relationship with the one on the same type going towards the same Movie node given a User node	Which are the non-followed top critics with the most affinity towards a given user
Check if a Movie node has different ratio of freshness in the incoming REVIEWED edges split based on their date attribute	Check if a movie has been targeted by review bombing in the last x months

In the following pages are presented the implementation in Java of the queries

3 .1 Users with most reviews

```
1  ArrayList<UserLeaderboardDTO> userList = session.readTransaction((
       TransactionWork<ArrayList<UserLeaderboardDTO>>) tx ->{
2              String query = "MATCH (b)-[:REVIEWED]->(m:Movie) " +
3                      "RETURN b.id as Id, b.name AS Name, count(*) as NumMovies
       , labels(b) as Type " +
4                      "ORDER BY NumMovies DESC " +
5                      "LIMIT 10";
6              Result result = tx.run(query);
7              ArrayList<UserLeaderboardDTO> list = new ArrayList<>();
8              while(result.hasNext()){
9                  Record r = result.next();
10                 UserLeaderboardDTO user = new UserLeaderboardDTO();
11                 user.setId(new ObjectId(r.get("Id").asString()));
12                 user.setUsername(r.get("Name").asString());
13                 user.setCounter(r.get("NumMovies").asInt());
14                 user.setType(r.get("Type").get(0).asString());
15                 list.add(user);
16             }
17             return list;
18         });
```

Listing 5.2: mostActiveUser

3 .2 Most followed top critics

```
ArrayList<UserLeaderboardDTO> userList = session.readTransaction((
    TransactionWork<ArrayList<UserLeaderboardDTO>>) tx ->{
            String query = "MATCH (t:TopCritic)<-[:FOLLOWS]-(u:User) " +
                    "RETURN t.id as Id, t.name AS Name, count(*) as NumMovies
     " +
                    "ORDER BY NumMovies DESC " +
                    "LIMIT 10";
            Result result = tx.run(query);
            ArrayList<UserLeaderboardDTO> list = new ArrayList<>();
            while(result.hasNext()){
                Record r = result.next();
                UserLeaderboardDTO user = new UserLeaderboardDTO();
                user.setId(new ObjectId(r.get("Id").asString()));
                user.setUsername(r.get("Name").asString());
                user.setCounter(r.get("NumMovies").asInt());
                list.add(user);
            }
            return list;


        });
```

Listing 5.3: mostFollowTopCritic

3 .3 Latest review movies by followed top critics

```
ArrayList<ReviewFeedDTO> reviewFeed = session.readTransaction((
    TransactionWork<ArrayList<ReviewFeedDTO>>)(tx -> {
            String query = "MATCH(u:User{id:$userId})-[f:FOLLOWS]->(t:
    TopCritic)-[r:REVIEWED]->(m:Movie) "+
                    "RETURN t.id as Id, m.title AS movieTitle, m.id AS
    movieId, t.name AS criticName, r.date AS reviewDate, "+
                    "r.content AS content, r.freshness AS freshness " +
                    "ORDER BY r.date DESC SKIP $skip LIMIT $limit ";
            Result result = tx.run(query, parameters("userId", usr.getId().
    toString(),
                    "skip", skip, "limit", REVIEWS_IN_FEED));
            ArrayList<ReviewFeedDTO> feed = new ArrayList<>();
            while(result.hasNext()){
                Record r = result.next();
                Date date;
                try {
                    date = new SimpleDateFormat("yyyy-MM-dd").parse(String.
    valueOf(r.get("reviewDate").asLocalDate()));
                } catch (ParseException e) {
                    throw new RuntimeException(e);
                }
                feed.add(new ReviewFeedDTO(
                        r.get("Id").asString(),
                        r.get("movieTitle").asString(),
                        r.get("movieId").asString(),
                        r.get("criticName").asString(),
                        r.get("content").asString(),
                        r.get("freshness").asBoolean(),
                        date
                ));
            }
```

```
27              return feed;
28          }));
```

Listing 5.4: Latest reviews

3 .4 Non-followed top critics with the most affinity

```
1 ArrayList<TopCriticSuggestionDTO> suggestionFeed = session.readTransaction((
      TransactionWork<ArrayList<TopCriticSuggestionDTO>>)(tx -> {
2             String query = "MATCH(u:User{id:$userId})-[r:REVIEWED]->(m:Movie)
      <-[r2:REVIEWED]-(t:TopCritic) "+
3                     "WHERE NOT (u)-[:FOLLOWS]->(t) " +
4                     "RETURN 100*(toFloat(sum(case when r.freshness = r2.
      freshness then 1 else 0 end)+1)/(count(m.title)+2)) as Rate, "+
5                     "t.name as Name, t.id as Id ORDER BY Rate DESC SKIP $skip
       LIMIT $limit";
6             Result result = tx.run(query, parameters("userId", usr.getId().
      toString(),
7                     "skip", skip, "limit", SUGGESTIONS_IN_FEED));
8             ArrayList<TopCriticSuggestionDTO> feed = new ArrayList<>();
9             while(result.hasNext()){
10                Record r = result.next();
11
12                feed.add(new TopCriticSuggestionDTO(
13                        r.get("Id").asString(),
14                        r.get("Name").asString(),
15                        (int) r.get("Rate").asDouble()
16                ));
17            }
18            return feed;
19        }));
```

Listing 5.5: suggestion feed

3 .5 Check for review bombing

```
1 reviewBombingList = session.readTransaction((TransactionWork<
      MovieReviewBombingDTO>)(tx -> {
2             String query = "MATCH (m:Movie{id:$movieId})<-[r:REVIEWED]-() " +
3                     "WITH SUM(CASE WHEN r.date<date(\""+strDate+"\") THEN 1
      ELSE 0 END) as StoricCount, " +
4                     "100*toFloat(SUM(CASE WHEN r.date<date(\""+strDate+"\")
      AND r.freshness = true THEN 1 ELSE 0 END))" +
5                     "/SUM(CASE WHEN r.date<date(\""+strDate+"\") THEN 1 ELSE
      0 END) as StoricRate, " +
6                     "SUM(CASE WHEN r.date>=date(\""+strDate+"\") AND r.date<
      date(\""+todayString+"\") THEN 1 ELSE 0 END) as TargetCount, " +
7                     "100*toFloat(SUM(CASE WHEN r.date>=date(\""+strDate+"\")
      AND r.date<date(\""+todayString+"\") AND r.freshness = true THEN 1 ELSE 0
       END))" +
8                     "/SUM(CASE WHEN r.date>=date(\""+strDate+"\") AND r.date<
      date(\""+todayString+"\") THEN 1 ELSE 0 END) as TargetRate, m.title as
      Title " +
9                     "RETURN Title, StoricCount, StoricRate, TargetCount,
      TargetRate";
10            Result result=null;
11            try{
```

```
12                  result=tx.run(query, parameters("movieId", movie.getId().
    toString(),
13                          "date", strDate));
14              }
15              catch (org.neo4j.driver.exceptions.ClientException e){
16                  return null;
17              }
18              MovieReviewBombingDTO feed = new MovieReviewBombingDTO(
19                      result.peek().get("Title").asString(),
20                      result.peek().get("StoricCount").asInt(),
21                      (int)result.peek().get("StoricRate").asDouble(),
22                      result.peek().get("TargetCount").asInt(),
23                      (int)result.peek().get("TargetRate").asDouble(),
24                      LocalDate.now().minusMonths(month)
25              );
26              return feed;
27          }));
```

Listing 5.6: review bombing

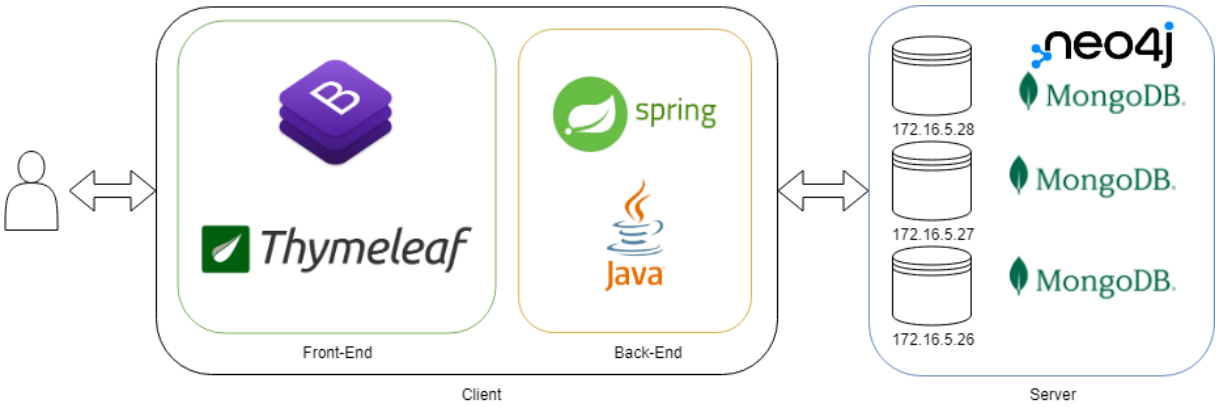
Chapter 6

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 7

Application Structure

1 Modules and code organization

The picture below represents the packages in which the application is organized.

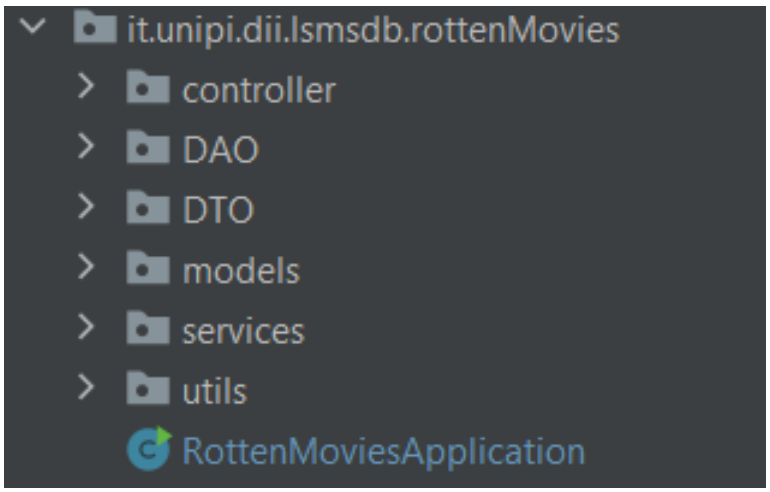


Figure 7.1: module organization

First of all we follow the reverse-domain convention for the root package, before passing to analyzing the source code, we put in the Appendix the pom.xml file for the dependency (Appendix E .1).

- *controller* is responsible for handling the request to the various endpoints with the use of Spring

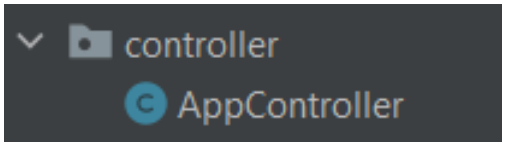


Figure 7.2: controller

- *DAO* (Data Access Object) is responsible for accessing the databases and retrieving the necessary objects

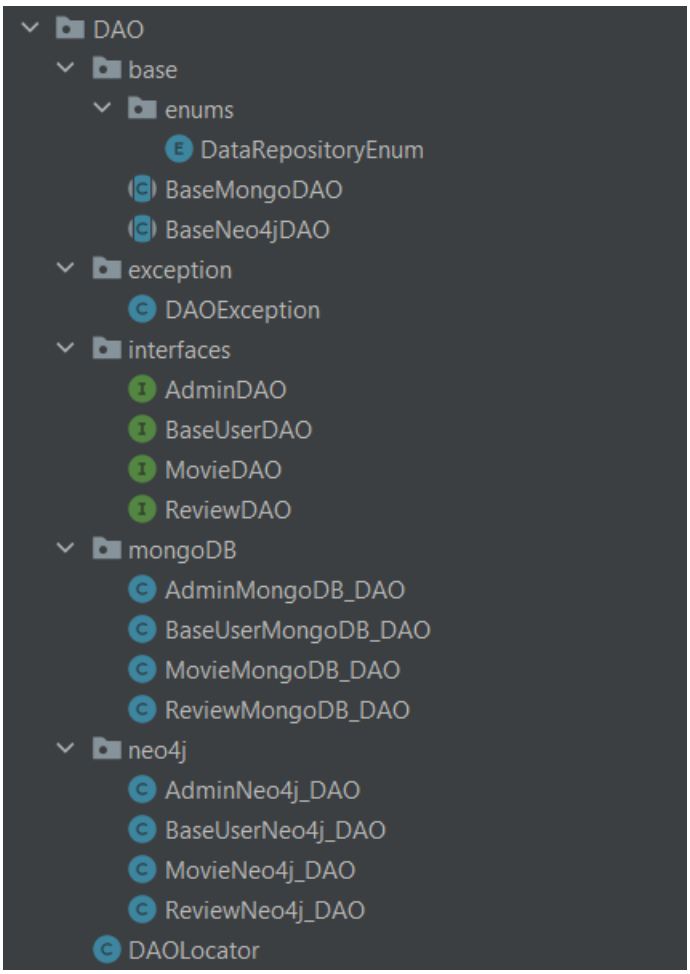


Figure 7.3: DAO

  - *base* contains the classes responsible for handling the base connection to the DBs, *enums* is packet used to differentiate the connection type based on an enum from above calls
  - *exception* is responsible for generating and handling a custom exception invoked when trying to access the wrong database
  - *interfaces* contains the various interfaces that map all the methods for accessing the databases differentiated based on the general field for the operation, they extend the *AutoClosable* interface
  - *mongoDB* handles the operation on the MongoDB for the different entities
  - *neo4j* is the same as *mongoDB* but for the Neo4j database

- *DTO* (Data Transfer Object) presents all the classes that are used as container of data passed between the service layer and the presentation layer

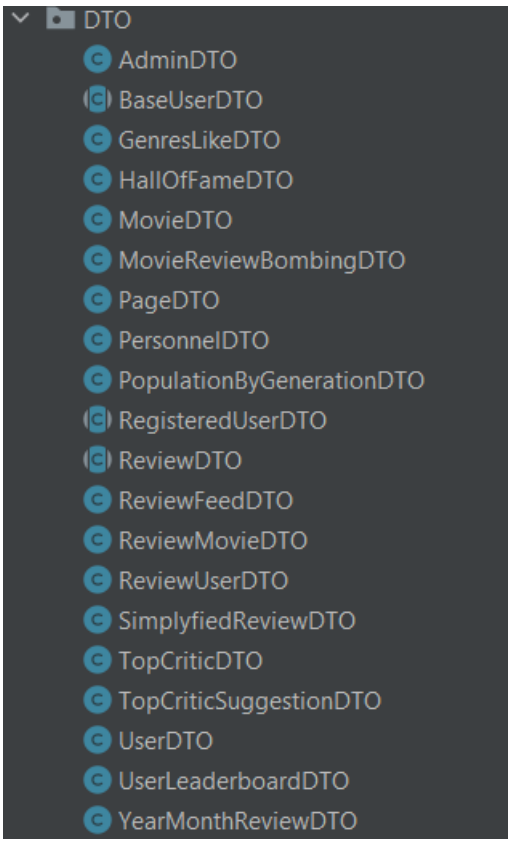


Figure 7.4: DTO

- *Models* presents all the classes that are mapped with the database organization. They all contains private fields and have getters/setters.

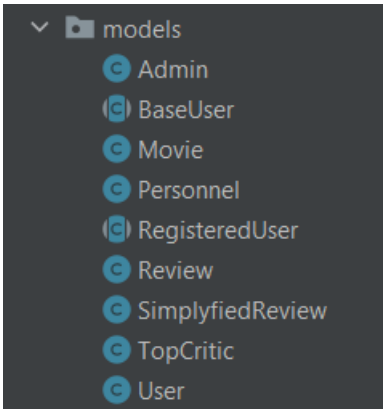


Figure 7.5: Models

- *Services* contains the middleware of our application. It is called in AppController methods and interfaces with DAO classes.

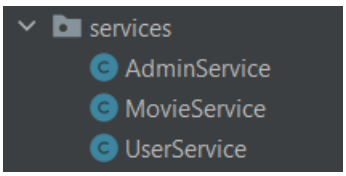


Figure 7.6: Services

- *Utils* provides utility methods to all packages. It includes password hashing, different possibility to sort/project results in Neo4j and MongoDB

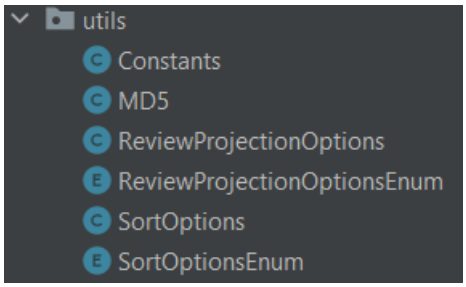


Figure 7.7: Utils

2 Managing consistency between MongoDB and Neo4j

Because we use two databases we need to manage consistency among them. An example on how it is managed is the addMovie method in MovieService. Here first we try to add a movie in MongoDB, if the Mongo operation is successfull we try to add the movie in Neo4j. If Neo4j fails we roll-back the insert on MongoDB, deleting the movie added previously. This strategy is also adopted in every other add/update/delete operation.

2 .1 Insert movie

```
1  public ObjectId addMovie(String title) {
2          if (title == null || title.isEmpty()) {
3              return null;
4          }
5          Movie newMovie = new Movie();
6          newMovie.setPrimaryTitle(title);
7          ObjectId id = null;
8          try (MovieDAO moviedao = DAOLocator.getMovieDAO(DataRepositoryEnum.
       MONGO)) {
9              id = moviedao.insert(newMovie);
10         } catch (Exception e) {
11             System.err.println(e);
12         }
13         if (id == null) {
14             return null;
15         }
16         newMovie.setId(id);
17         try (MovieDAO moviedao = DAOLocator.getMovieDAO(DataRepositoryEnum.
       NEO4j)) {
18             id = moviedao.insert(newMovie);
```

```
19          } catch (Exception e) {
20              System.err.println(e);
21          }
22          if (id == null){ // roll back di mongo
23              try (MovieDAO moviedao = DAOLocator.getMovieDAO(
        DataRepositoryEnum.MONGO)) {
24                  moviedao.delete(newMovie);
25              } catch (Exception e) {
26                  System.err.println(e);
27              }
28              return null;
29          }
30          return id;
31      }
```

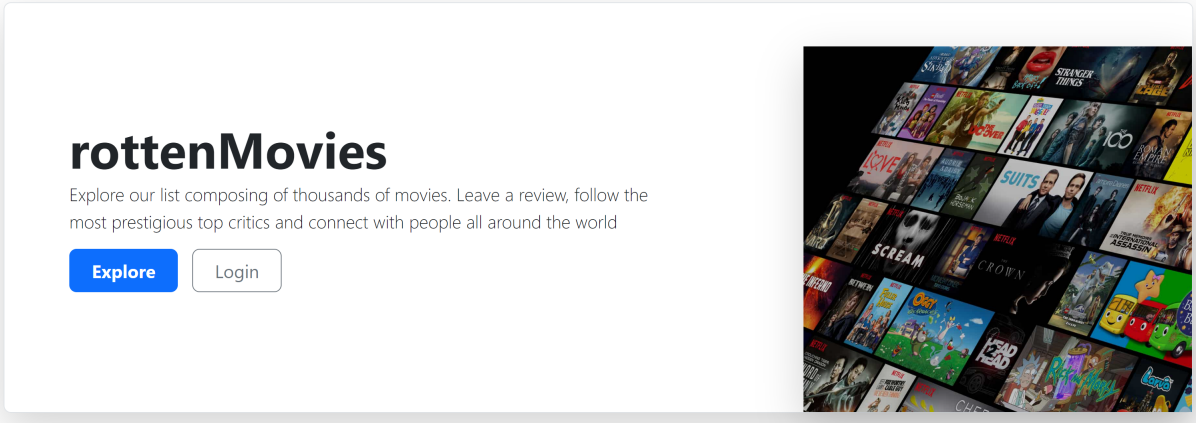
Listing 7.1: Movie insertion

Chapter 8

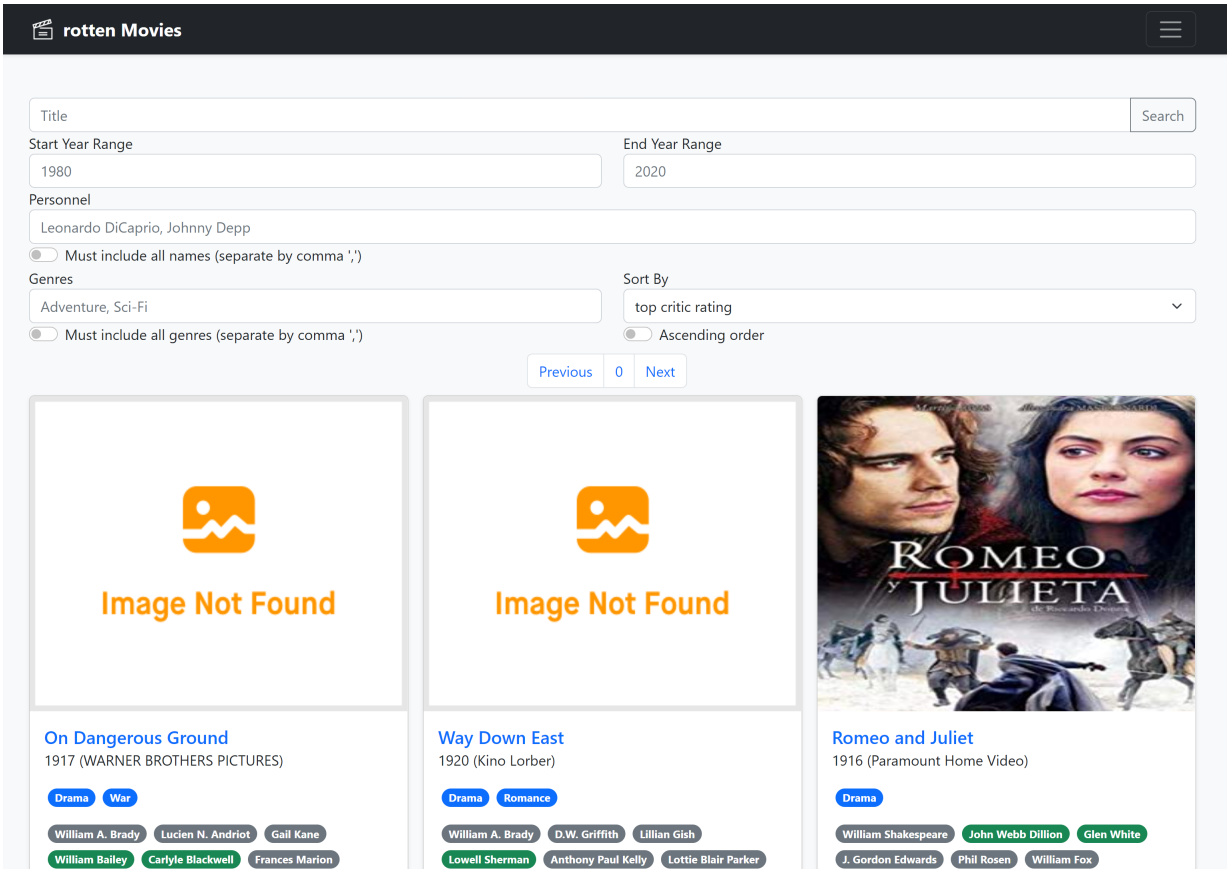
Instruction Manual

The application is accessible through a web browser and doesn't require an authentication to perform some basic navigation and scrolling of movies and reviews. To leave a review under a movie, follow top critics and receive personalized information, a user must log-in or sign-up to the service. Some notable and certified users may be registered by an administrator as a *Top Critic* which can be followed by other users and whose reviews are highlighted under a movie

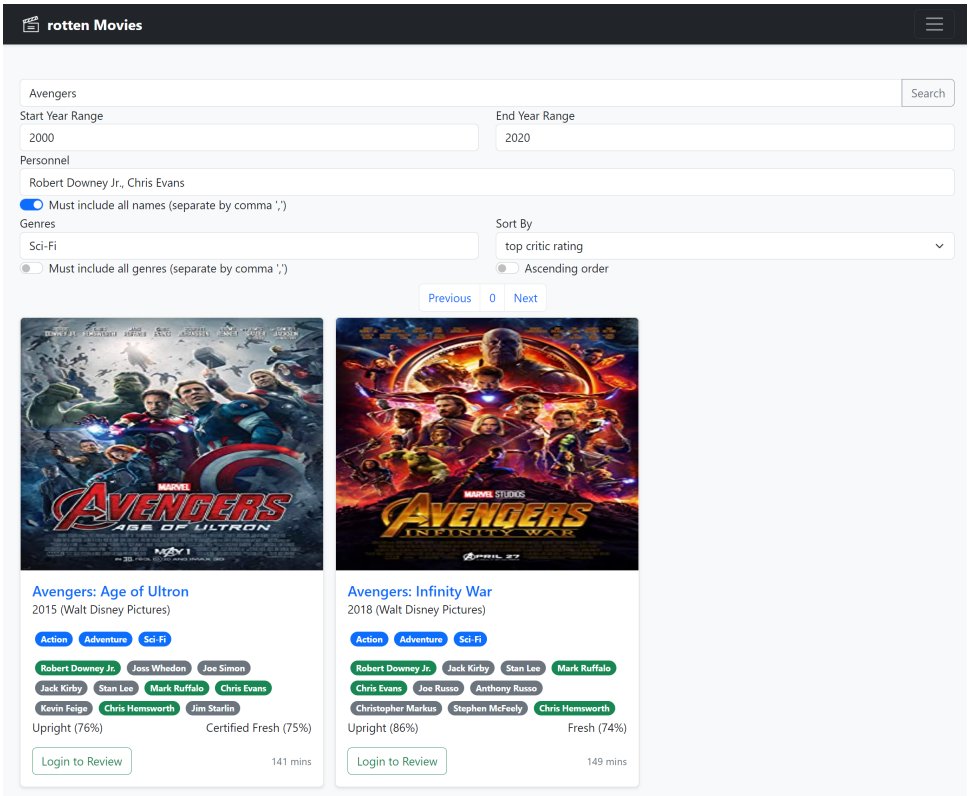
1 Greeting Page



The landing page presents the option to jump directly to the main page for exploring movies and incites the new user to login or sign-up to the application.



The exploring page offers multiple options to filter between all movies inside the database



It is possible to search by

- a string contained in the title
- a range of year in which the movie has been produced

- people that worked on the film. It is possible to build a list of names by employing a comma (,) and stating if the movie must include all names or at least one of them
- genres. It is possible to build a list of genres by employing a comma (,) and stating if the movie must include all genres or at least one of them

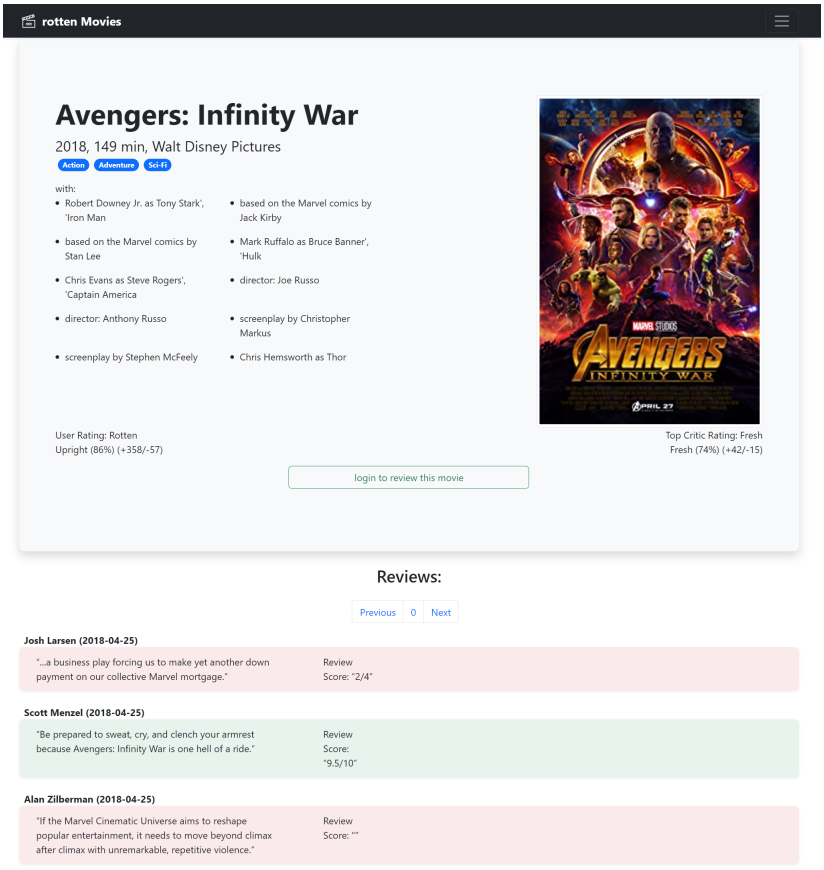
The available sorting options are

- by top critic rating (the default)
- by user critic rating
- alphabetical order
- by date of production

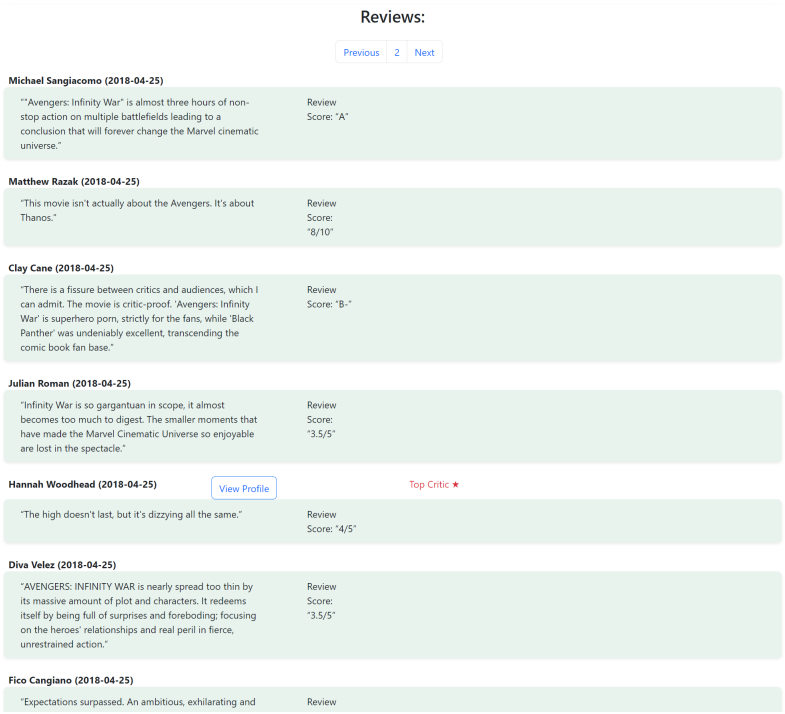
and the user can opt for an ascending or descending sort.

At any point the user can click on the pills of genres or crew members to perform a search with that value.

If a user isn't logged-in, the button under a movie says 'Login to Review' and brings to the login page. Otherwise it becomes 'Review' and clicking on it allows the quick creation or editing of the review for that movie.



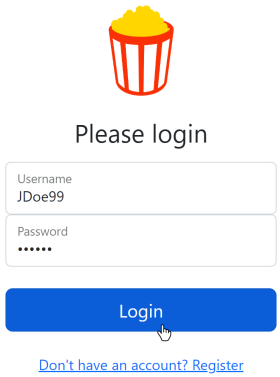
Selecting a movie allows to see a full description of it



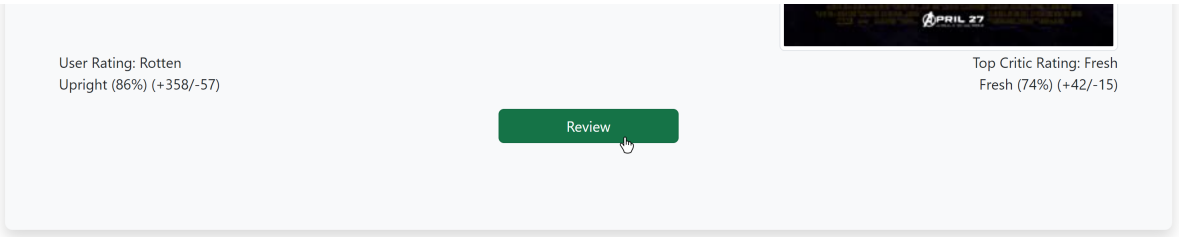
A selected movie allows also to read a paginated list of all its reviews

If a user isn't logged-in, the button under the movie says 'Login to Review this movie' and brings to the login page. Otherwise it becomes 'Review' and clicking on it allows the creation or editing of the review for that movie.

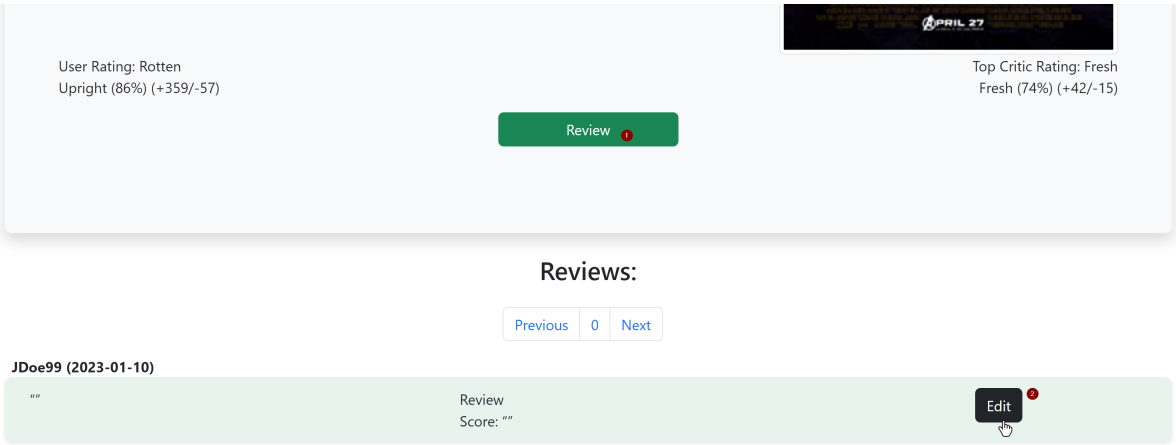
User registration requires some basic information about the new user



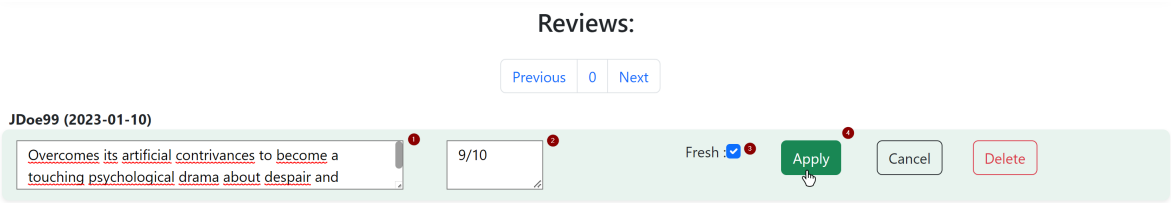
If a user has already registered to the application, he/she can login with its credentials



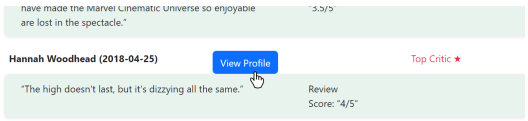
After logging-in, a user can leave a review under a movie



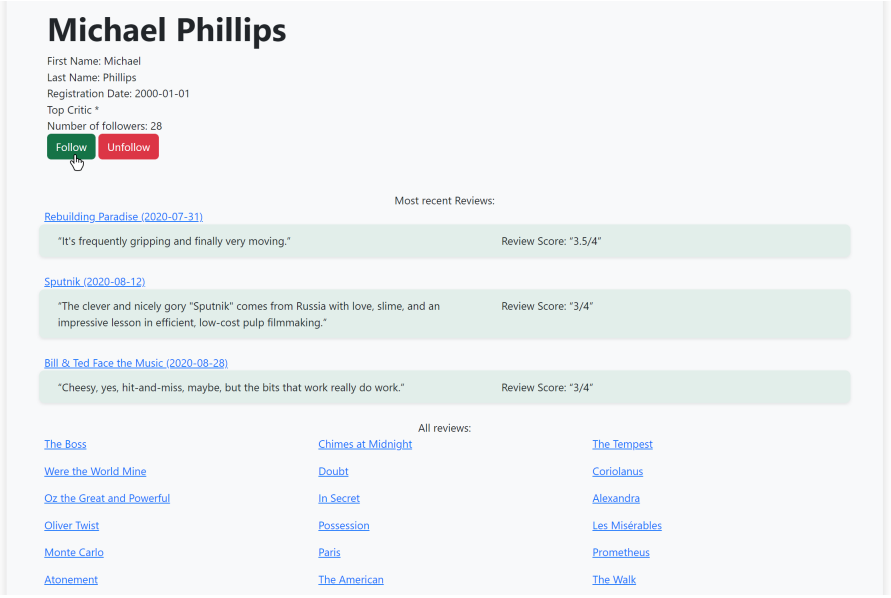
If a user has never reviewed a movie, a new one is created and can be immediately modified



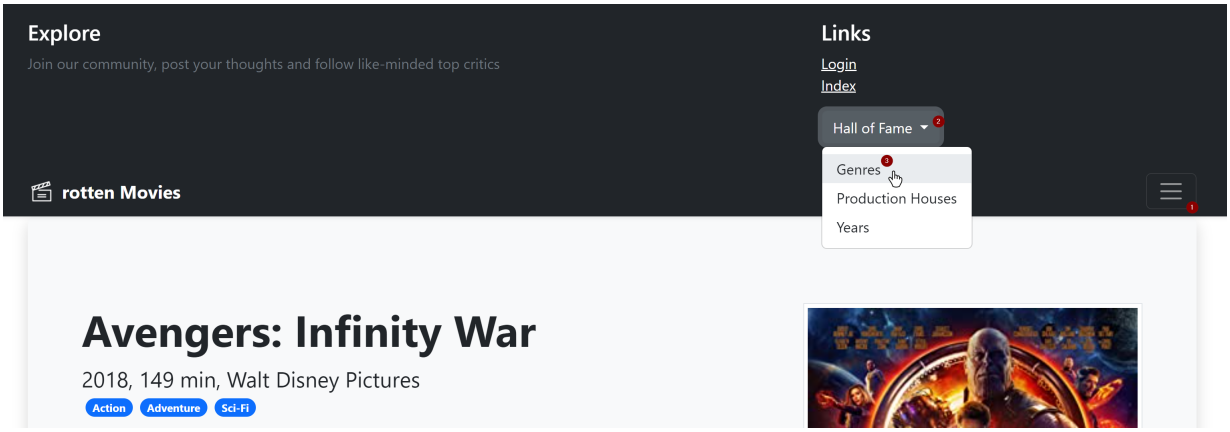
Writing a review, a user can write its content, leave a summarizing score and determine if it's a *Fresh* (positive) or *Rotten* (negative) review. Here the user can also choose to delete its review



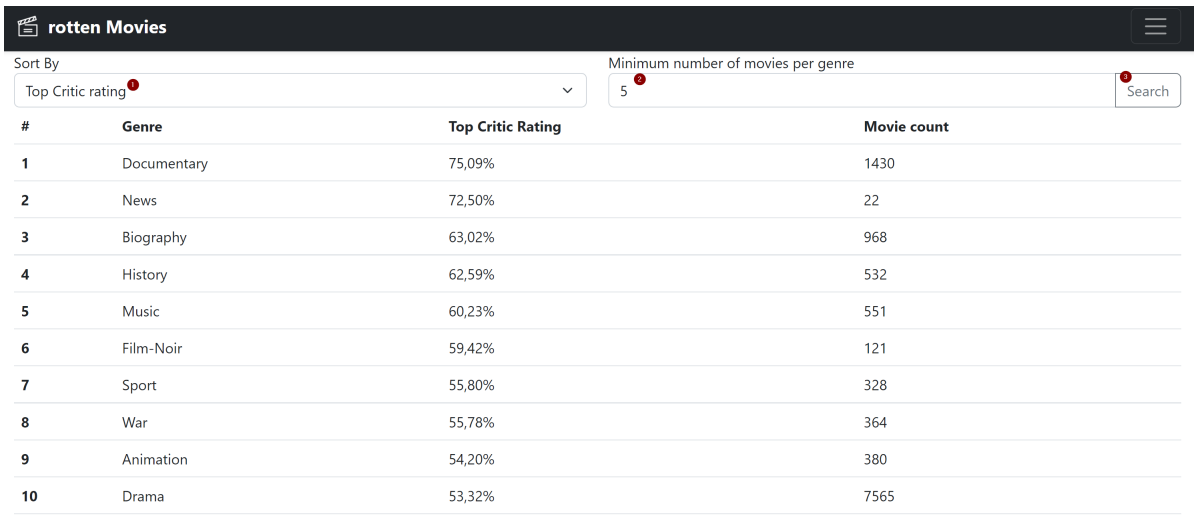
Top Critic reviews are highlighted by a little label with a star and a link that brings to its user page



Viewing the user page of a Top Critic, a user can choose to follow or, if its already a follower, unfollow that critic

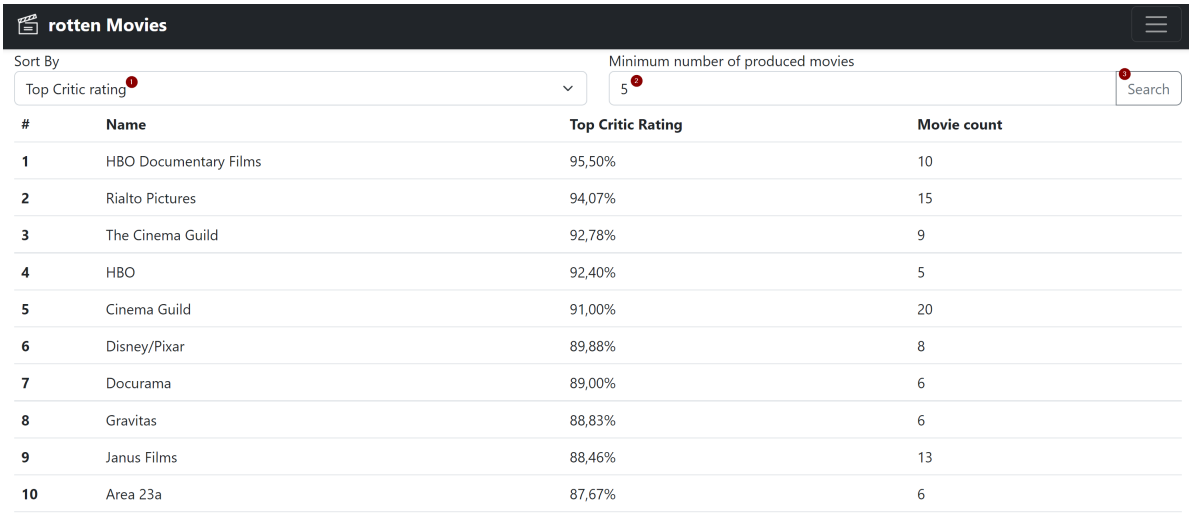


The navbar for a user that hasn't logged-in has some basic features



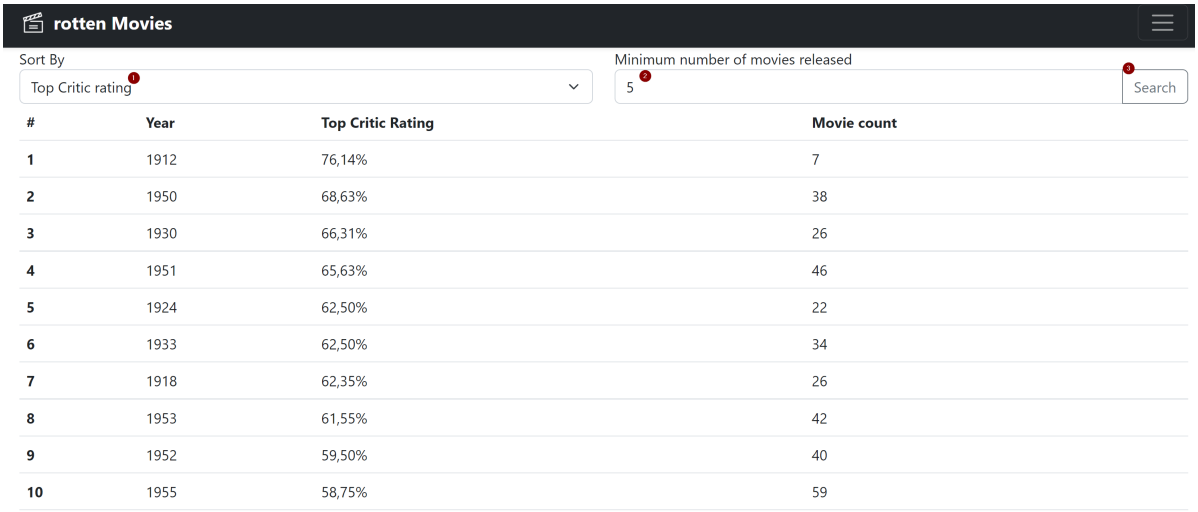
#	Genre	Top Critic Rating	Movie count
1	Documentary	75,09%	1430
2	News	72,50%	22
3	Biography	63,02%	968
4	History	62,59%	532
5	Music	60,23%	551
6	Film-Noir	59,42%	121
7	Sport	55,80%	328
8	War	55,78%	364
9	Animation	54,20%	380
10	Drama	53,32%	7565

This page shows the best genres across all movies, sorted by top critic rating or user rating. It is also possible to specify the minimum number of movies that a genre must have to appear in the list



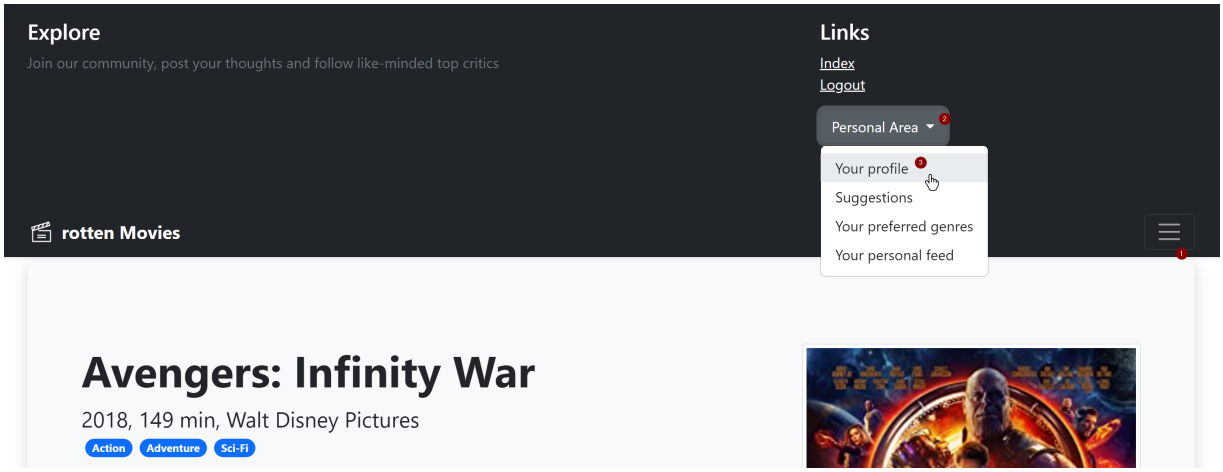
#	Name	Top Critic Rating	Movie count
1	HBO Documentary Films	95,50%	10
2	Rialto Pictures	94,07%	15
3	The Cinema Guild	92,78%	9
4	HBO	92,40%	5
5	Cinema Guild	91,00%	20
6	Disney/Pixar	89,88%	8
7	Docurama	89,00%	6
8	Gravitas	88,83%	6
9	Janus Films	88,46%	13
10	Area 23a	87,67%	6

This page shows the best production houses based on the reviews left by users on the platform. They can be sorted by top critic rating or user rating and the user can specify the minimum number of produced movies to appear on the list

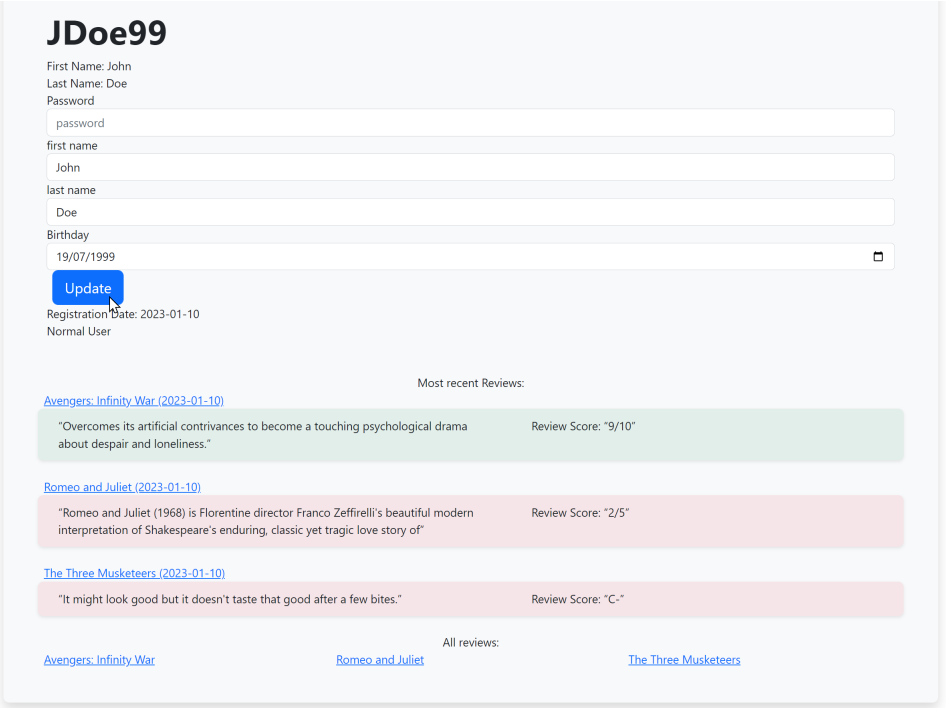


#	Year	Top Critic Rating	Movie count
1	1912	76,14%	7
2	1950	68,63%	38
3	1930	66,31%	26
4	1951	65,63%	46
5	1924	62,50%	22
6	1933	62,50%	34
7	1918	62,35%	26
8	1953	61,55%	42
9	1952	59,50%	40
10	1955	58,75%	59

This page shows the best years in terms of produced movies with positive reviews left by users on the platform. They can be sorted by top critic rating or user rating and the user can specify the minimum number of released movies in that year to appear on the list

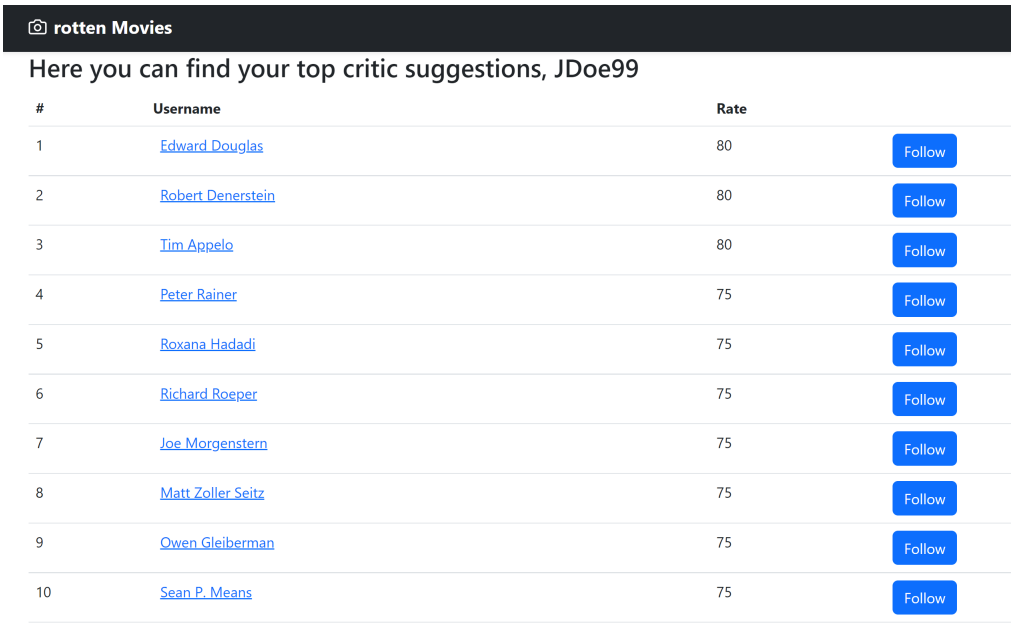


The navbar of an authenticated user shows some more personalized pages

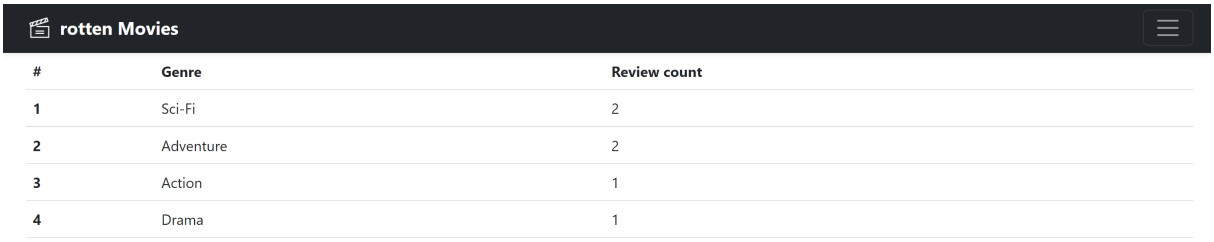


In the personal page, a user can change its credentials, see its three most recent reviews and the list of all its reviewed movies.

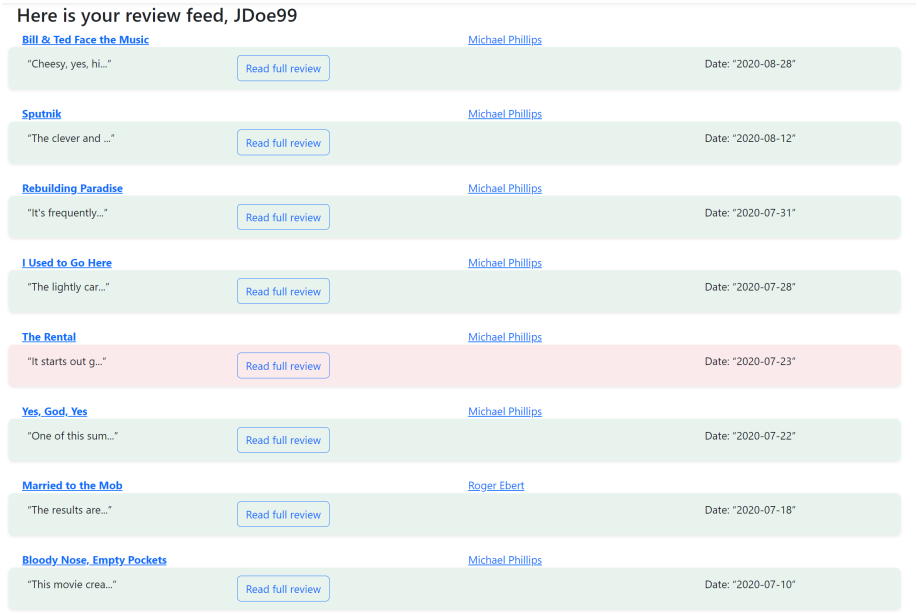
Leaving the password field blank won't change it.



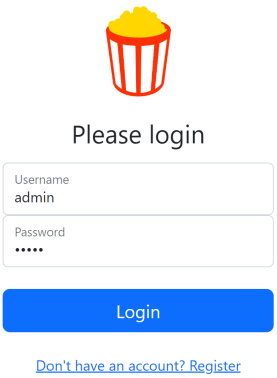
The suggestion page shows a list of possible top critics that the user could follow. These top critics are chosen based on common reviewed movies (both positively or both negatively) between the two. The Rate index shows the estimated probability that the user might align with the suggested top critic views.



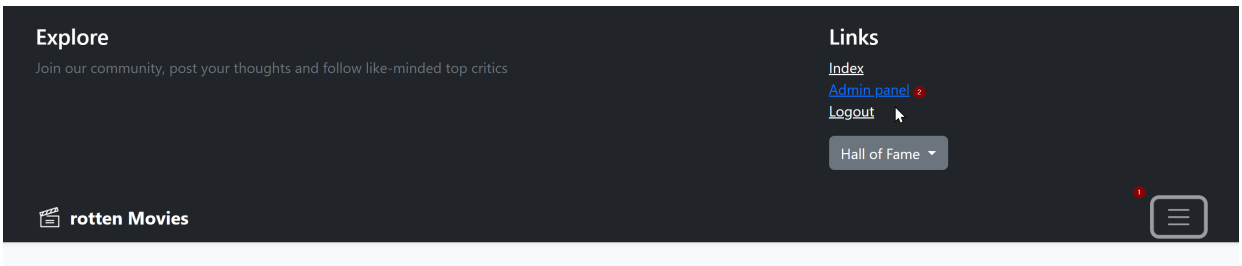
The preferred genres page shows an aggregated overview of genres reviewed by the user. This highlights possible preferences towards certain genres more than others



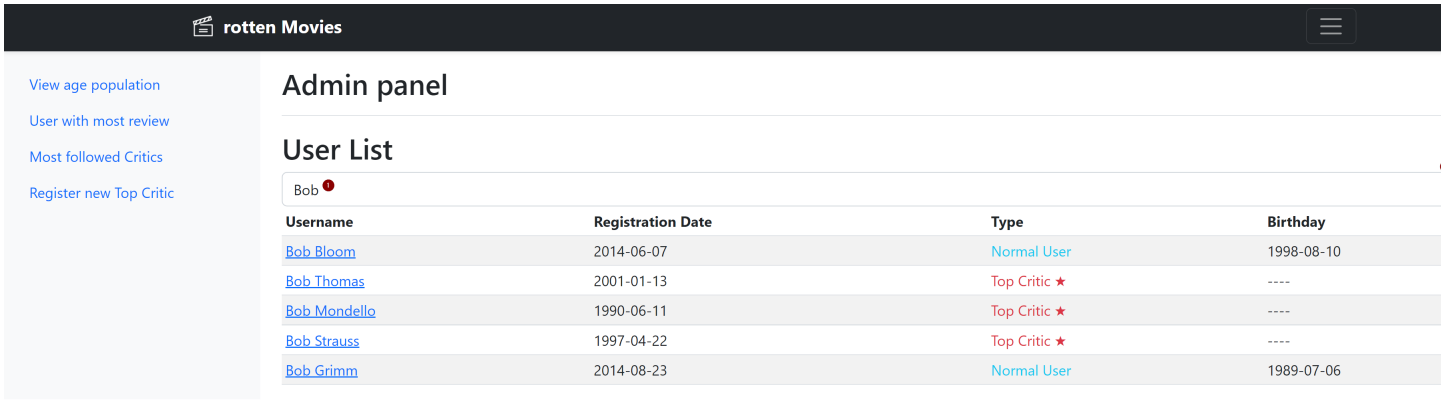
The feed shows a list of recent reviews written by followed top critics. The review contains an excerpt of the full content, which can be seen by clicking on the button *Read full review*



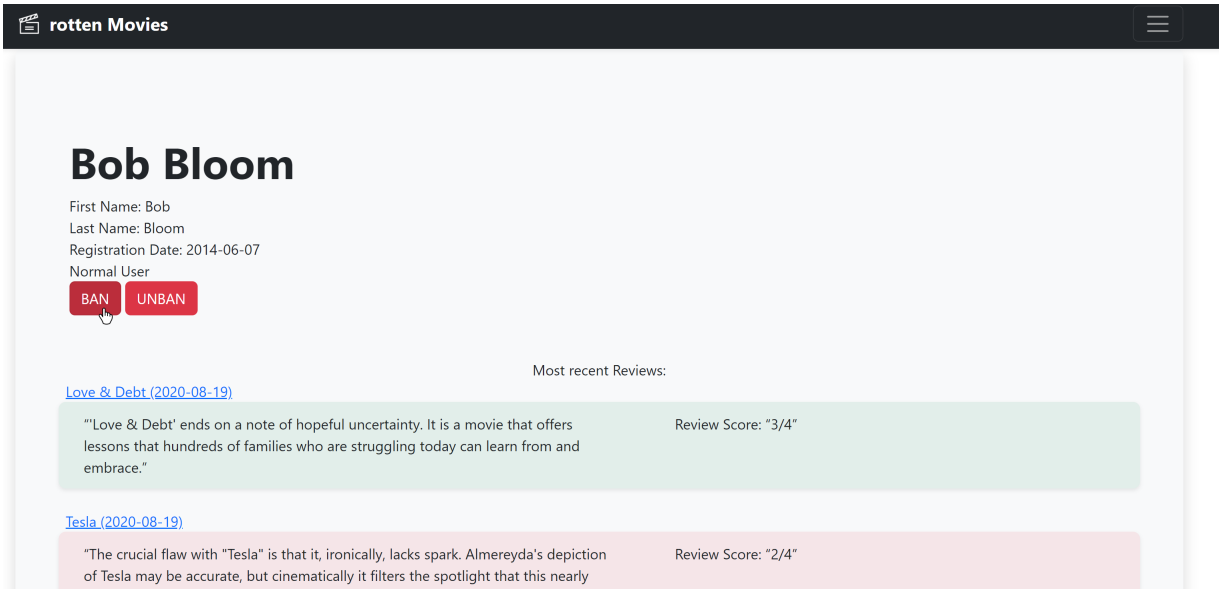
A special kind of user can log-in to perform administration tasks on the application



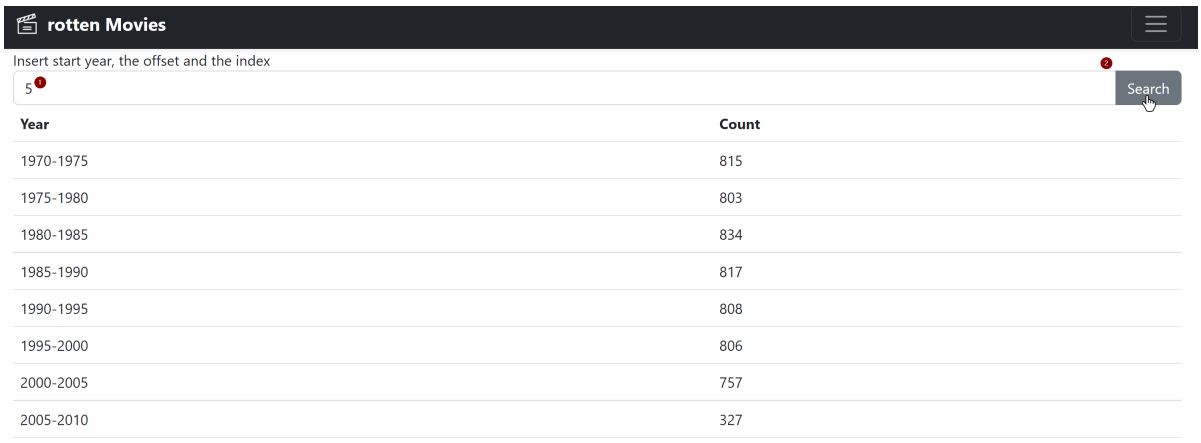
The admin navbar allows to reach the administrator panel



In the administrator panel the admin can search for a specific user. The side panel links to different analytic pages about the status of the application



After selecting a user, the admin can ban or unban it



This analytic shows the number of registered users for each year range, passed as input

rotten Movies

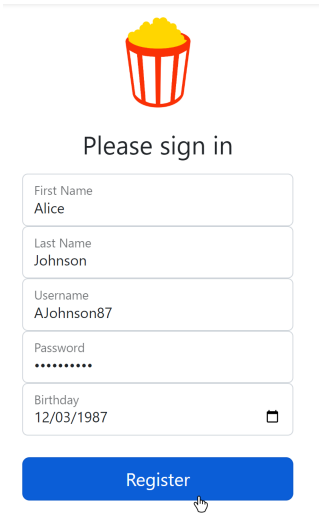
#	Username	Type	Number of Reviews
1	Emanuel Levy	TopCritic	5850
2	Roger Ebert	TopCritic	4850
3	Dennis Schwartz	User	4813
4	Brian Orndorf	User	4546
5	Frank Swietek	User	4528
6	Jeffrey M. Anderson	User	4153
7	Roger Moore	TopCritic	4143
8	James Berardinelli	TopCritic	3995
9	David Nusair	User	3908
10	Frederic and Mary Ann Brussat	User	3677

This page shows a ranking of users based on their number of reviews. This can be useful to verify a possible automated user, or promote a certified active user as a top critic

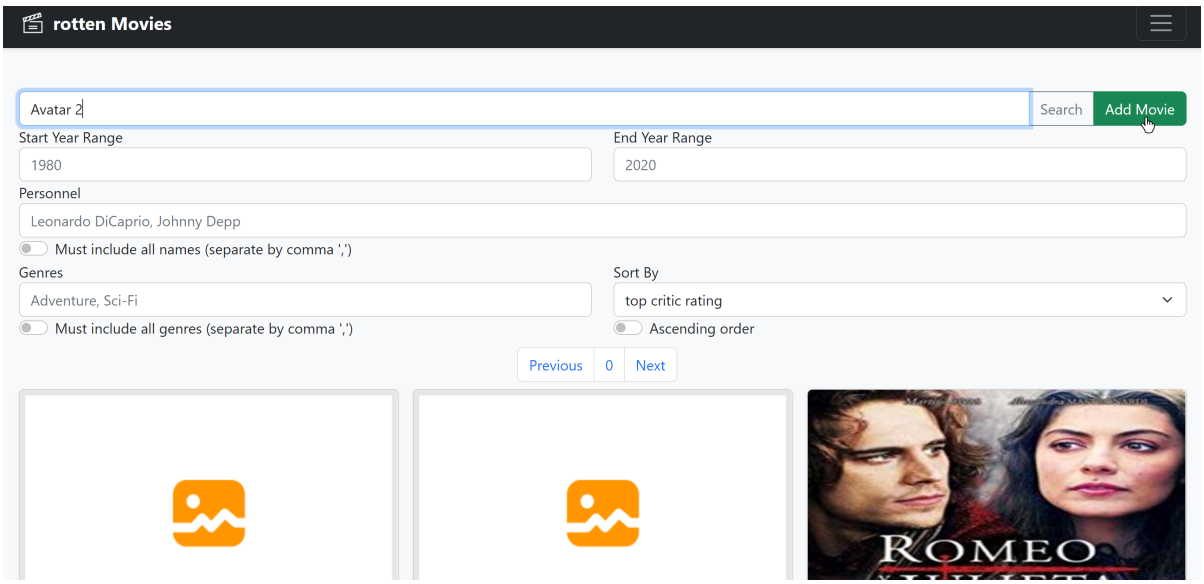
rotten Movies

#	Username	Type	Number of Reviews
1	Elissa Suh		44
2	Gene Stout		41
3	Harry Carr		41
4	Lynne Heffley		41
5	Gemma Files		41
6	Chris Harvey		40
7	Suzanne S. Brown		40
8	Julia Irion Martins		40
9	Manohla Dargis		40
10	Drew Toal		40

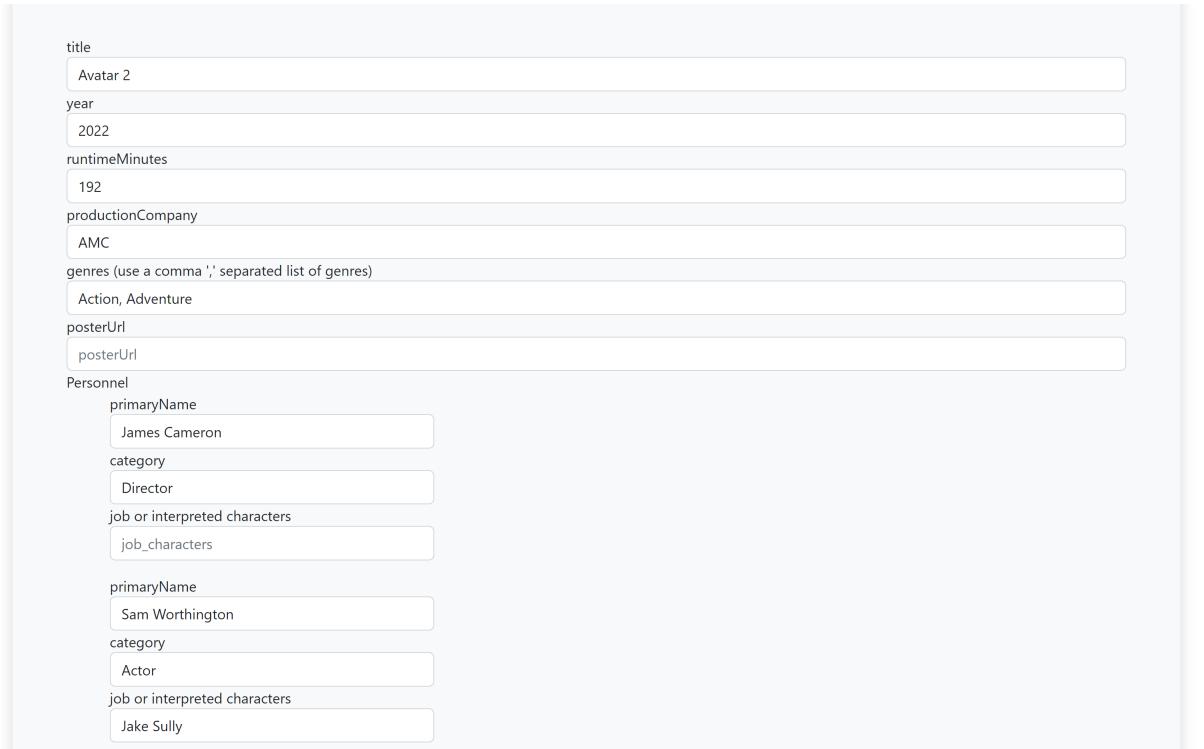
With this page, the administrator can see a ranking of most active top critics, based on the number of reviews



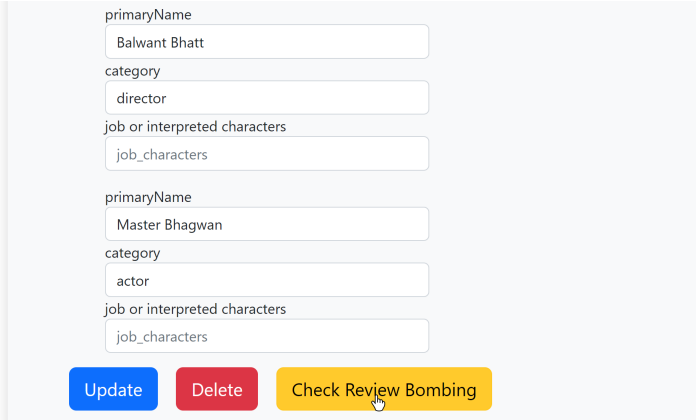
The top critic creation page isn't accessible to the public, only to the registered administrator. It is very similar to the user registration page, but it doesn't redirect to the explore movie page after a successful sign-up



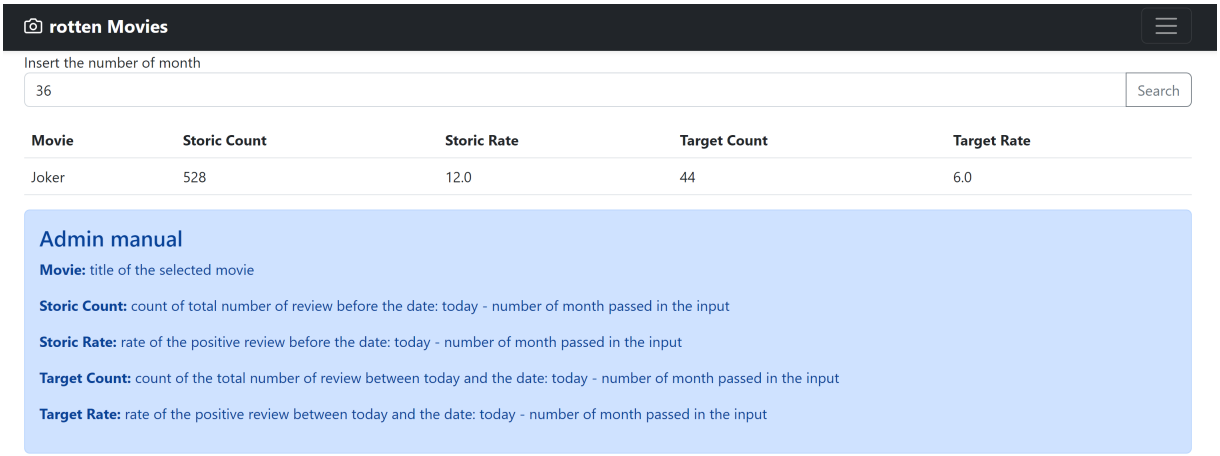
Adding a new movie is done through the explore movie page. A new button labeled *Add Movie* appears near the *Search* one and will create a new movie, temporarily named as the string currently present in the search bar



After adding a movie, or when selecting an existing one, the administrator can change its descriptive information



When viewing a movie, the administrator can check for possible review bombing for this movie, by clicking on the specific button



The review bombing page shows some statistics about recent reviews made on a specific movie. An info box explains the meaning of each value

A Python Code

A .1 Creation of one single dataset from the tsv imdb files

```
1
2 import pandas as pd
3 from google.colab import drive
4 drive.mount('/content/drive')
5 numberofrows=None #100000
6 title_basics = pd.read_csv("/content/drive/MyDrive/Dataset/Original/
      title_basics.tsv",sep='\t',nrows=numberofrows,header=0)
7 title_principals = pd.read_csv("/content/drive/MyDrive/Dataset/Original/
      title_principals.tsv",nrows=numberofrows,sep='\t',header=0)
8
9 keep_col = ["tconst","titleType","primaryTitle","originalTitle","startYear","
      runtimeMinutes","genres"]
10 title_basics = title_basics[keep_col]
11 title_basics = title_basics[title_basics["titleType"].str.contains("movie")
      == True]
12
13 print(title_basics.head(3))
14
15 merged1=pd.merge(title_basics,title_principals,how='inner',on='tconst')
16 del title_basics,title_principals
17 print(merged1)
18
19 name_basics=pd.read_csv("/content/drive/MyDrive/Dataset/Original/name_basics.
      tsv",sep='\t',nrows=numberofrows,header=0)
20
21 merged2=pd.merge(merged1,name_basics,how='inner',on='nconst')
22 del merged1
23 merged2=merged2.drop(columns=["ordering","nconst","birthYear","deathYear","
      knownForTitles","primaryProfession"])
24 del name_basics
25 print(merged2)
26
27 category=merged2.groupby('tconst')['category'].apply(list).reset_index(name='
      category')
28 job=merged2.groupby('tconst')['job'].apply(list).reset_index(name='job')
29 characters=merged2.groupby('tconst')['characters'].apply(list).reset_index(
      name='characters')
30 primaryName=merged2.groupby('tconst')['primaryName'].apply(list).reset_index(
      name='primaryName')
31 result=merged2.drop_duplicates(subset=['tconst'])
32 result=result.drop(['category'], axis=1).drop(['job'], axis=1).drop(['
      characters'], axis=1).drop(['primaryName'], axis=1)
33 result=result.merge(category,on='tconst').merge(job,on='tconst').merge(
      characters,on='tconst').merge(primaryName,on='tconst')
34 print(result)
35
36 result.to_csv("/content/drive/MyDrive/Dataset/resultSetFinale.csv",index=
      False)
```

Listing 8.1: Test

A .2 Creation of one single dataset from the csv Kaggle files

```
1
```

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
numberofrows=None #100000
movies = pd.read_csv("/content/drive/MyDrive/Dataset/Original/rotten_movies.csv",nrows=numberofrows,header=0)
reviews = pd.read_csv("/content/drive/MyDrive/Dataset/Original/rotten_reviews.csv",nrows=numberofrows,header=0)
to_keep = ["rotten_tomatoes_link", "movie_title", "production_company","critics_consensus",
           "tomatometer_status", "tomatometer_rating", "tomatometer_count",
           "audience_status", "audience_rating", "audience_count",
           "tomatometer_top_critics_count", "tomatometer_fresh_critics_count",
           "tomatometer_rotten_critics_count"]

movies = movies[to_keep]
to_drop = ["publisher_name"]
reviews=reviews.drop(columns=to_drop)

merged=pd.merge(movies,reviews,how='inner',on="rotten_tomatoes_link")
print(merged)

categories = {}
arr = ["critic_name", "top_critic", "review_type", "review_score", "review_date", "review_content"]
for x in arr:
  categories[x]=merged.groupby('rotten_tomatoes_link')[x].apply(list).reset_index(name=x)

result=merged.drop_duplicates(subset=['rotten_tomatoes_link'])
for x in arr:
  result=result.drop([x], axis=1)
for x in arr:
  result=result.merge(categories[x],on='rotten_tomatoes_link')

print(result)

result.to_csv("/content/drive/MyDrive/Dataset/resultSetRotten.csv",index=False)
```

Listing 8.2: Test

A .3 Merging of the files generated in the previous scripts

```

import pandas as pd
from google.colab import drive
drive.mount('/content/drive')
numberofrows=None
imdb = pd.read_csv("/content/drive/MyDrive/Dataset/resultSetFinale.csv",nrows=numberofrows,header=0)
rotten = pd.read_csv("/content/drive/MyDrive/Dataset/resultSetRotten.csv",nrows=numberofrows,header=0)

merged = {}
choose = ['primaryTitle', 'originalTitle']
rowHeadDataset = 20
for x in choose:
```

```
13    merged[x]=pd.merge(imdb,rotten,how='inner', left_on=x, right_on='
       movie_title')
14    print(x)
15    merged[x]=merged[x].drop_duplicates(subset=[x])
16    merged[x]=merged[x].drop(columns=['titleType','tomatometer_count','
       tomatometer_top_critics_count'])
17    merged[x]=merged[x].rename(columns={'startYear':'year'})
18    merged[x]=merged[x].drop(columns=['rotten_tomatoes_link', 'movie_title']+[j
       for j in choose if j!=x])
19    print(len(pd.unique(merged[x][x])))
20    print(list(merged[x]))
21    print("==========================")
22    merged[x].to_csv(f"/content/drive/MyDrive/Dataset/ImdbJoinRotten{x}.csv",
       index=False)
23    merged[x]=merged[x].head(rowHeadDataset)
24    merged[x].to_csv(f"/content/drive/MyDrive/Dataset/headDataset{x}.csv",index
       =False)
```

Listing 8.3: Test

A .4 Collapsing different rows into a single one by generating an array for personnel field

```
1
2  import pandas as pd
3  from ast import literal_eval
4  from google.colab import drive
5  drive.mount('/content/drive')
6
7  numberofrows=None
8  df = pd.read_csv("/content/drive/MyDrive/Dataset/ImdbJoinRottenprimaryTitle.
       csv",nrows=numberofrows,header=0)
9
10 #print([x.split(',') for x in df['genres']])
11 print(df)
12
13 col = ["primaryName","category","job","characters"]
14 col1 = ["critic_name","top_critic","review_type","review_score","review_date"
       , "review_content"]
15
16 df['personnel'] = ""
17 df['review'] = ""
18
19 for row in range(df[col[0]].size):
20   it = df['genres'][row]
21   df['genres'][row] = ['"' + x + '"' for x in it.split(',')] if it != '\\N'
       else []
22   tmp = []
23   for c in col:
24     tmp.append({c:eval(df[c][row])})
25   res = []
26   for c in range(len(tmp[0][col[0]])):
27     res.append({})
28   for i, j in zip(col, tmp):
29     for idx, x in enumerate(j[i]):
30       #print(i, idx, x)
31       if x != '\\N':
32         if i == 'characters':
33           x = eval(x)
```

```
34          res[idx]["'" + i + "'"] = "'" + str(x).replace("'", "##single-quote##
    ").replace('"', "##double-quote##") + "'"
35      df['personnel'][row] = list(res)
36      #print(res)
37      ###
38      tmp = []
39      for c in col1:
40        to_eval = df[c][row].replace('nan', 'None')
41        arr = eval(to_eval)
42        if c == "review_date":
43          for i, elem in enumerate(arr):
44            arr[i] = elem + "T00:00:00.000+00:00"
45        tmp.append({c:arr})
46        #print(tmp)
47      res = []
48      for c in range(len(tmp[0][col1[0]])):
49        res.append({})
50      for i, j in zip(col1, tmp):
51        for idx, x in enumerate(j[i]):
52          #print(i, idx, x)
53          if x != '\\N':
54            res[idx]["'" + i + "'"] = "'" + str(x).replace("True", "true").
    replace("False", "false").replace("'", "##single-quote##").replace('"', "
    ##double-quote##") + "'"
55      df['review'][row] = list(res)
56      #df['review'][row] = eval(str(res))
57      #print(res)
58      #print()
59
60  df=df.drop(columns=col)
61  df=df.drop(columns=col1)
62  df=df.drop(columns=['tconst'])
63
64  print(df["review"][0])
65
66  it = df['personnel'][0]#[4]['review_content']
67  print(type(it))
68  print(it)
69
70  df.to_csv("/content/drive/MyDrive/Dataset/
    movieCollectionEmbeddedReviewPersonnel.csv",index=False)
71  df = df.head(20)
72  df.to_csv("/content/drive/MyDrive/Dataset/
    headmovieCollectionEmbeddedReviewPersonnel.csv",index=False)
```

Listing 8.4: Test

A .5 Generates a hashed password for all the users

```
1  import hashlib
2  #from pprint import pprint as print
3  from pymongo import MongoClient
4
5  def get_database():
6      CONNECTION_STRING = "mongodb://localhost:27017"
7      client = MongoClient(CONNECTION_STRING)
8      return client['rottenMovies']
9
10 if __name__ == "__main__":
```

```
    dbname = get_database()
    collection = dbname['user']
    total = collection.count_documents({})
    for i, user in enumerate(collection.find()):
        all_reviews = user['last_3_reviews']
        sorted_list = sorted(all_reviews, key=lambda t: t['review_date'])
    [-3:]

        hashed = hashlib.md5(user["username"].encode()).hexdigest()

        newvalues = { "$set": { 'password': hashed, 'last_3_reviews':
    sorted_list } }
        filter = { 'username': user['username']}
        collection.update_one(filter, newvalues)
        print(f"{i/total:%}\r", end='')
    print()
```

Listing 8.5: Test

A .6 Generates the graph database

```
from pymongo import MongoClient
from neo4j import GraphDatabase
from random import randint, shuffle

def get_database():
    CONNECTION_STRING = "mongodb://localhost:27017"
    client = MongoClient(CONNECTION_STRING)
    return client['rottenMovies']

class Neo4jGraph:

    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password),
    database="rottenmoviesgraphdb")

    def close(self):
        self.driver.close()

    def addUser(self, uid, name, isTop):
        with self.driver.session() as session:
            if isTop:
                result = session.execute_write(self._addTopCritic, uid, name)
            else:
                result = session.execute_write(self._addUser, uid, name)

    def addMovie(self, mid, title):
        with self.driver.session() as session:
            result = session.execute_write(self._addMovie, mid, title)

    def addReview(self, name, mid, freshness, content, date):
        with self.driver.session() as session:
            result = session.execute_write(self._addReview, name, mid,
    freshness, content, date)

    def addFollow(self, uid, cid):
        with self.driver.session() as session:
            result = session.execute_write(self._addFollow, uid, cid)

```

```
37      @staticmethod
38      def _addUser(tx, uid, name):
39          query = "CREATE (n:User{id:\"" + str(uid) + "\", name:\"" + name.
        replace('"', '\\"') + "\"})"
40          #print(query)
41          result = tx.run(query)
42
43      @staticmethod
44      def _addTopCritic(tx, cid, name):
45          query = "CREATE(m:TopCritic{id:\"" + str(cid) + "\", name:\"" + name.
        replace('"', '\\"') + "\"})"
46          #print(query)
47          result = tx.run(query)
48
49      @staticmethod
50      def _addMovie(tx, mid, title):
51          query = "CREATE(o:Movie{id:\"" + str(mid) + "\", title:\"" + title.
        replace('"', '\\"') + "\"})"
52          #print(query)
53          result = tx.run(query)
54
55      @staticmethod
56      def _addReview(tx, name, mid, freshness, content, date): # date in format
        YYYY-mm-dd, freshness in [TRUE, FALSE]
57          query = "MATCH(n{name:\"" + str(name).replace('"', '\\"') + "\"}), (m
        :Movie{id:\"" + str(mid) + "\"}) CREATE (n)-[r:REVIEWED{freshness:" +
        freshness + ", date:date('" + date + "'), content:\"" + content.replace('
        "', '\\"') + "\"}]->(m)"
58          #print(query)
59          result = tx.run(query)
60
61      @staticmethod
62      def _addFollow(tx, uid, cid):
63          query = "MATCH(n:User{id:\"" + str(uid) + "\"}), (m:TopCritic{id:\""
        + str(cid) + "\"}) CREATE (n)-[r:FOLLOWS]->(m)"
64          #print(query)
65          result = tx.run(query)
66
67  if __name__ == "__main__":
68      # dbs initialization
69      dbname = get_database()
70      graphDB = Neo4jGraph("bolt://localhost:7687", "neo4j", "password")
71
72      # user creation
73      collection = dbname['user']
74      total = collection.count_documents({})
75      print(f"user {total = }")
76      for i, user in enumerate(list(collection.find({}, {"_id":1, "username":1,
        "date_of_birth":1}))):
77          graphDB.addUser(user['_id'], user['username'], 'date_of_birth' not in
        user)
78          if not i%100:
79              print(f"{(i+1)/total:%}\r", end='')
80
81      # movie creation and review linking
82      collection = dbname['movie']
83      total = collection.count_documents({})
84      print(f"\nmovie {total = }")
85      for i, movie in enumerate(list(collection.find({}, {"_id":1, "
        primaryTitle":1, "review":1}))):
86          graphDB.addMovie(movie['_id'], movie['primaryTitle'])
```

```
87          movie['review'] = list({v['critic_name']:v for v in movie['review']}.
        values()) # make unique reviews per critic
88          for rev in movie['review']:
89              graphDB.addReview(rev['critic_name'], movie['_id'], {"Fresh":"
        TRUE", "Rotten":"FALSE"}[rev['review_type']], str(rev['review_content'])
        [:15], str(rev['review_date'])[:10])
90          print(f"{(i+1)/total:%}\r", end='')
91
92      # follow linking
93      collection = dbname['user']
94      uids = [x['_id'] for x in list(collection.find({"date_of_birth":{"$exists
        ":True}}, {"_id":1}))]
95      cids = [x['_id'] for x in list(collection.find({"date_of_birth":{"$exists
        ":False}}, {"_id":1}))]
96      total = len(uids)
97      print(f"\nfollow {total = }")
98      for i, user in enumerate(uids):
99          shuffle(cids)
100         for j in range(randint(0, 20)):
101             graphDB.addFollow(user, cids[j])
102         print(f"{i/total:%}\r", end='')
103
104     graphDB.close()
```

Listing 8.6: Test

B Mongosh scripts

B .1 Perform the escape on the string fields

```
1
2 db.movie.find().forEach(
3     x => {
4         print(x.primaryTitle);
5         x.review = JSON.parse(
6             x.review.replaceAll('"\'', '"')
7                 .replaceAll('\'"', '"')
8                 .replaceAll('"false"', 'false')
9                 .replaceAll('"true"', 'true')
10                .replaceAll('"None"', 'null')
11                .replaceAll(/\\x\d{2}/g, "")
12                .replaceAll("##single-quote##", "\'")
13                .replaceAll("##double-quote##", '\\"')
14                .replaceAll("\\x", "x")
15        );
16        x.personnel = JSON.parse(
17            x.personnel.replaceAll('"\'', '"')
18                .replaceAll('\'"', '"')
19                .replaceAll('"None"', 'null')
20                .replaceAll("##single-quote##", '\'')
21                .replaceAll("##double-quote##", '\\"')
22                .replaceAll('"[\'', '["')
23                .replaceAll('"[\\"', '["')
24                .replaceAll('\']"', '"]')
25                .replaceAll('\\"]"', '"]')
26                .replaceAll(/(\[[^[:]*)\\", \\"([^]:]*\])/g, '$1", "$2')
27                .replaceAll(/(\[[^[:]*)\', \\"([^]:]*\])/g, '$1", "$2')
28                .replaceAll(/(\[[^[:]*)\\", \'([^]:]*\])/g, '$1", "$2')
29        );
30        x.genres = JSON.parse(
31            x.genres = x.genres.replaceAll('"\'', '"')
32                    .replaceAll('\'"', '"')
33                    .replaceAll('"None"', 'null')
34                    .replaceAll("##single-quote##", "\'")
35                    .replaceAll("##double-quote##", '\\"')
36        );
37        db.movie.updateOne(
38            {"_id": x._id},
39            {$set:
40                {
41                    "review": x.review,
42                    "personnel": x.personnel,
43                    "genres": x.genres,
44                    "runtimeMinutes":parseInt(x.runtimeMinutes),
45                    "year":parseInt(x.year),
46                    "tomatometer_rating":parseFloat(x.tomatometer_rating),
47                    "audience_rating":parseFloat(x.audience_rating),
48                    "audience_count":parseFloat(x.audience_count),
49                    "tomatometer_fresh_critics_count":parseInt(x.
    tomatometer_fresh_critics_count),
50                    "tomatometer_rotten_critics_count":parseInt(x.
    tomatometer_rotten_critics_count)
51                }
52            }
53        );
```

```
    }
);
```

Listing 8.7: Test

B .2 Normalize the date fields in the DB

```
total = db.movie.find().count();
i = 0;
db.movie.find().forEach(
    x => {
        print(x.primaryTitle);
        x.review.forEach(rev =>{
            if(typeof (rev.review_date) === "string" ){
                db.movie.updateOne(
                    {primaryTitle: x.primaryTitle },
                    { $set: { "review.$[elem].review_date" : new Date(rev.review_date) } },
                    { arrayFilters: [ { "elem.critic_name": rev.critic_name } ] }
                )
            }
        })
        print(100*i++/total);
});
```

Listing 8.8: Test

B .3 Create a new collection for the user based on the data present in the movie collection

```
total = db.runCommand({ distinct: "movie", key: "review.critic_name", query: {"review.critic_name":{$ne:null}}}).values.length
i = 0;
db.runCommand(
{ distinct: "movie", key: "review.critic_name", query: {"review.critic_name":{$ne:null}}}).values.forEach(
    (x) => {
        review_arr = []
        movie_arr = []
        is_top = false
        db.movie.aggregate(
            [
                { $project:
                    {
                        index: { $indexOfArray: ["$review.critic_name", x]},
                        primaryTitle: 1
                    }},
                {$match:{index:{$gt:-1}}}
            ]
        ).forEach(
            y => {
                tmp = db.movie.aggregate([
                    {
                        $project:
                        {
                            top_critic: {
```

```
                                    $arrayElemAt: ["$review.top_critic", y.index]
                                },
                                primaryTitle: y.primaryTitle,
                                review_type: {
                                    $arrayElemAt: ["$review.review_type", y.index
    ]
                                },
                                review_score: {
                                    $arrayElemAt: ["$review.review_score", y.
    index]
                                },
                                review_date: {
                                    $arrayElemAt: ["$review.review_date", y.index
    ]
                                },
                                review_content: {
                                    $arrayElemAt: ["$review.review_content", y.
    index]
                                }
                            }
                        },
                        {
                            $match:{_id:{$eq:y._id}}
                        }
                    ]).toArray()[0];
                    is_top |= tmp.top_critic;
                    review_arr.push(tmp)
                    //movie_arr.push(tmp._id)
                    movie_arr.push({"movie_id": tmp._id, "primaryTitle": y.
    primaryTitle, "review_index": y.index})
                })

        name_parts = x.split(/\s/)
        first_name = name_parts.splice(0, 1)[0]
        last_name = name_parts.join(' ')

        print(100*i++/total, x, is_top)
        //print(first_name, ':', last_name)
        //print(review_arr)
        //print(movie_arr)
        db.user.insertOne(
            {
                "username": x,
                "password": "",
                "first_name": first_name,
                "last_name": last_name,
                "registration_date": new Date("2000-01-01"),
                "last_3_reviews": review_arr,
                "reviews" : movie_arr
            }
        );
        if (!is_top){
            db.user.updateOne(
                {"username": x},
                {$set:
                    {"date_of_birth": new Date("1970-07-20")}
                }
            )
        }
        print("================================")
    }
```

```
)
```

Listing 8.9: Test

C MongoDB indexes:Movie collection

C .1 primaryTitle

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { primaryTitle: { '$eq': 'Evidence' } },
    queryHash: '9839850C',
    planCacheKey: '9839850C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { primaryTitle: { '$eq': 'Evidence' } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 275,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { primaryTitle: { '$eq': 'Evidence' } },
      nReturned: 1,
      executionTimeMillisEstimate: 245,
      works: 14106,
      advanced: 1,
      needTime: 14104,
      needYield: 0,
      saveState: 18,
      restoreState: 18,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
  command: {
    find: 'movie',
    filter: { primaryTitle: 'Evidence' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
```

```
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280853, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280853, i: 1 })
}
```

Listing 8.10: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { primaryTitle: { '$eq': 'Evidence' } },
    queryHash: '9839850C',
    planCacheKey: 'B734708E',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { primaryTitle: 1 },
        indexName: 'primaryTitle_1',
        isMultiKey: false,
        multiKeyPaths: { primaryTitle: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { primaryTitle: [ '["Evidence", "Evidence"]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 1,
    totalKeysExamined: 1,
```

```
    totalDocsExamined: 1,
    executionStages: {
      stage: 'FETCH',
      nReturned: 1,
      executionTimeMillisEstimate: 0,
      works: 2,
      advanced: 1,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 1,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 1,
        executionTimeMillisEstimate: 0,
        works: 2,
        advanced: 1,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { primaryTitle: 1 },
        indexName: 'primaryTitle_1',
        isMultiKey: false,
        multiKeyPaths: { primaryTitle: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { primaryTitle: [ '["Evidence", "Evidence"]' ] },
        keysExamined: 1,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: { primaryTitle: 'Evidence' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
```

```
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285103, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285103, i: 1 })
}
```

Listing 8.11: Test

C .2 year

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { year: { '$eq': 2012 } },
    queryHash: '412E8B51',
    planCacheKey: '412E8B51',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { year: { '$eq': 2012 } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 480,
    executionTimeMillis: 13,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { year: { '$eq': 2012 } },
      nReturned: 480,
      executionTimeMillisEstimate: 1,
      works: 14106,
      advanced: 480,
      needTime: 13625,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
```

```
  command: { find: 'movie', filter: { year: 2012 }, '$db': 'rottenMovies' },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280923, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280923, i: 1 })
}
```

Listing 8.12: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { year: { '$eq': 2012 } },
    queryHash: '412E8B51',
    planCacheKey: '62915BA3',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { year: 1 },
        indexName: 'year_1',
        isMultiKey: false,
        multiKeyPaths: { year: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { year: [ '[2012, 2012]' ] }
      }
    },
    rejectedPlans: []
  },
```

```
  executionStats: {
    executionSuccess: true,
    nReturned: 480,
    executionTimeMillis: 2,
    totalKeysExamined: 480,
    totalDocsExamined: 480,
    executionStages: {
      stage: 'FETCH',
      nReturned: 480,
      executionTimeMillisEstimate: 0,
      works: 481,
      advanced: 480,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 480,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 480,
        executionTimeMillisEstimate: 0,
        works: 481,
        advanced: 480,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { year: 1 },
        indexName: 'year_1',
        isMultiKey: false,
        multiKeyPaths: { year: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { year: [ '[2012, 2012]' ] },
        keysExamined: 480,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: { find: 'movie', filter: { year: 2012 }, '$db': 'rottenMovies' },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
```

```
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285143, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285143, i: 1 })
}
```

Listing 8.13: Test

C .3 top critic rating

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '33018E32',
    planCacheKey: '33018E32',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'SORT',
      sortPattern: { top_critic_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: { stage: 'COLLSCAN', direction: 'forward' }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 1818,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'SORT',
      nReturned: 14104,
      executionTimeMillisEstimate: 1740,
      works: 28211,
      advanced: 14104,
      needTime: 14106,
      needYield: 0,
      saveState: 47,
      restoreState: 47,
      isEOF: 1,
      sortPattern: { top_critic_rating: 1 },
      memLimit: 104857600,
```

```
      type: 'simple',
      totalDataSizeSorted: 262640385,
      usedDisk: true,
      spills: 3,
      inputStage: {
        stage: 'COLLSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 0,
        works: 14106,
        advanced: 14104,
        needTime: 1,
        needYield: 0,
        saveState: 47,
        restoreState: 47,
        isEOF: 1,
        direction: 'forward',
        docsExamined: 14104
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { top_critic_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287293, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287293, i: 1 })
}
```

Listing 8.14: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
```

```
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '33018E32',
    planCacheKey: '33018E32',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { top_critic_rating: 1 },
        indexName: 'top_critic_rating_1',
        isMultiKey: false,
        multiKeyPaths: { top_critic_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { top_critic_rating: [ '[MinKey, MaxKey]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 24,
    totalKeysExamined: 14104,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'FETCH',
      nReturned: 14104,
      executionTimeMillisEstimate: 5,
      works: 14105,
      advanced: 14104,
      needTime: 0,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      docsExamined: 14104,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 1,
        works: 14105,
        advanced: 14104,
        needTime: 0,
        needYield: 0,
        saveState: 14,
        restoreState: 14,
        isEOF: 1,
        keyPattern: { top_critic_rating: 1 },
        indexName: 'top_critic_rating_1',
        isMultiKey: false,
        multiKeyPaths: { top_critic_rating: [] },
        isUnique: false,
        isSparse: false,
```

```
66          isPartial: false,
67          indexVersion: 2,
68          direction: 'forward',
69          indexBounds: { top_critic_rating: [ '[MinKey, MaxKey]' ] },
70          keysExamined: 14104,
71          seeks: 1,
72          dupsTested: 0,
73          dupsDropped: 0
74        }
75      }
76    },
77    command: {
78      find: 'movie',
79      filter: {},
80      sort: { top_critic_rating: 1 },
81      '$db': 'rottenMovies'
82    },
83    serverInfo: {
84      host: 'Profile2022LARGE10',
85      port: 27017,
86      version: '6.0.3',
87      gitVersion: 'f803681c3ae19817d31958965850193de067c516'
88    },
89    serverParameters: {
90      internalQueryFacetBufferSizeBytes: 104857600,
91      internalQueryFacetMaxOutputDocSizeBytes: 104857600,
92      internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
93      internalDocumentSourceGroupMaxMemoryBytes: 104857600,
94      internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
95      internalQueryProhibitBlockingMergeOnMongoS: 0,
96      internalQueryMaxAddToSetBytes: 104857600,
97      internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
98    },
99    ok: 1,
100   '$clusterTime': {
101     clusterTime: Timestamp({ t: 1673285423, i: 1 }),
102     signature: {
103       hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
      hex"), 0),
104       keyId: Long("0")
105     }
106   },
107   operationTime: Timestamp({ t: 1673285423, i: 1 })
108 }
```

Listing 8.15: Test

C .4 user rating

Before the index

```
1 {
2   explainVersion: '1',
3   queryPlanner: {
4     namespace: 'rottenMovies.movie',
5     indexFilterSet: false,
6     parsedQuery: {},
7     queryHash: '3E9B1E6C',
8     planCacheKey: '3E9B1E6C',
9     maxIndexedOrSolutionsReached: false,
```

```
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'SORT',
      sortPattern: { user_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      inputStage: { stage: 'COLLSCAN', direction: 'forward' }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 1779,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'SORT',
      nReturned: 14104,
      executionTimeMillisEstimate: 1698,
      works: 28211,
      advanced: 14104,
      needTime: 14106,
      needYield: 0,
      saveState: 45,
      restoreState: 45,
      isEOF: 1,
      sortPattern: { user_rating: 1 },
      memLimit: 104857600,
      type: 'simple',
      totalDataSizeSorted: 262640385,
      usedDisk: true,
      spills: 3,
      inputStage: {
        stage: 'COLLSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 0,
        works: 14106,
        advanced: 14104,
        needTime: 1,
        needYield: 0,
        saveState: 45,
        restoreState: 45,
        isEOF: 1,
        direction: 'forward',
        docsExamined: 14104
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { user_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287353, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287353, i: 1 })
}
```

Listing 8.16: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: {},
    queryHash: '3E9B1E6C',
    planCacheKey: '3E9B1E6C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { user_rating: 1 },
        indexName: 'user_rating_1',
        isMultiKey: false,
        multiKeyPaths: { user_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { user_rating: [ '[MinKey, MaxKey]' ] }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 14104,
    executionTimeMillis: 25,
    totalKeysExamined: 14104,
    totalDocsExamined: 14104,
```

```
    executionStages: {
      stage: 'FETCH',
      nReturned: 14104,
      executionTimeMillisEstimate: 5,
      works: 14105,
      advanced: 14104,
      needTime: 0,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      docsExamined: 14104,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 14104,
        executionTimeMillisEstimate: 2,
        works: 14105,
        advanced: 14104,
        needTime: 0,
        needYield: 0,
        saveState: 14,
        restoreState: 14,
        isEOF: 1,
        keyPattern: { user_rating: 1 },
        indexName: 'user_rating_1',
        isMultiKey: false,
        multiKeyPaths: { user_rating: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { user_rating: [ '[MinKey, MaxKey]' ] },
        keysExamined: 14104,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: {},
    sort: { user_rating: 1 },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
```

```
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285383, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285383, i: 1 })
}
```

Listing 8.17: Test

C .5 personnel.primaryName

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { 'personnel.primaryName': { '$eq': '' } },
    queryHash: 'E212F03B',
    planCacheKey: 'E212F03B',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'COLLSCAN',
      filter: { 'personnel.primaryName': { '$eq': '' } },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 47,
    totalKeysExamined: 0,
    totalDocsExamined: 14104,
    executionStages: {
      stage: 'COLLSCAN',
      filter: { 'personnel.primaryName': { '$eq': '' } },
      nReturned: 0,
      executionTimeMillisEstimate: 9,
      works: 14106,
      advanced: 0,
      needTime: 14105,
      needYield: 0,
      saveState: 14,
      restoreState: 14,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 14104
    }
  },
```

```
  command: {
    find: 'movie',
    filter: { 'personnel.primaryName': '' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673287593, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673287593, i: 1 })
}
```

Listing 8.18: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.movie',
    indexFilterSet: false,
    parsedQuery: { 'personnel.primaryName': { '$eq': '' } },
    queryHash: 'E212F03B',
    planCacheKey: '9D4A6814',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { 'personnel.primaryName': 1 },
        indexName: 'personnel.primaryName_1',
        isMultiKey: true,
        multiKeyPaths: { 'personnel.primaryName': [ 'personnel' ] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { 'personnel.primaryName': [ '["", ""]' ] }
```

```
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 0,
    totalKeysExamined: 0,
    totalDocsExamined: 0,
    executionStages: {
      stage: 'FETCH',
      nReturned: 0,
      executionTimeMillisEstimate: 0,
      works: 1,
      advanced: 0,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 0,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 0,
        executionTimeMillisEstimate: 0,
        works: 1,
        advanced: 0,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { 'personnel.primaryName': 1 },
        indexName: 'personnel.primaryName_1',
        isMultiKey: true,
        multiKeyPaths: { 'personnel.primaryName': [ 'personnel' ] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { 'personnel.primaryName': [ '["", ""]' ] },
        keysExamined: 0,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'movie',
    filter: { 'personnel.primaryName': '' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
87    },
88    serverParameters: {
89      internalQueryFacetBufferSizeBytes: 104857600,
90      internalQueryFacetMaxOutputDocSizeBytes: 104857600,
91      internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
92      internalDocumentSourceGroupMaxMemoryBytes: 104857600,
93      internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
94      internalQueryProhibitBlockingMergeOnMongoS: 0,
95      internalQueryMaxAddToSetBytes: 104857600,
96      internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
97    },
98    ok: 1,
99    '$clusterTime': {
100     clusterTime: Timestamp({ t: 1673287763, i: 1 }),
101     signature: {
102       hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
      hex"), 0),
103       keyId: Long("0")
104     }
105   },
106   operationTime: Timestamp({ t: 1673287763, i: 1 })
107 }
```

Listing 8.19: Test

D MongoDB indexes:User collection

D .1 username

Before the index

```
1  {
2    explainVersion: '1',
3    queryPlanner: {
4      namespace: 'rottenMovies.user',
5      indexFilterSet: false,
6      parsedQuery: { username: { '$eq': 'Abbie Bernstein' } },
7      queryHash: '7D9BB680',
8      planCacheKey: '7D9BB680',
9      maxIndexedOrSolutionsReached: false,
10     maxIndexedAndSolutionsReached: false,
11     maxScansToExplodeReached: false,
12     winningPlan: {
13       stage: 'COLLSCAN',
14       filter: { username: { '$eq': 'Abbie Bernstein' } },
15       direction: 'forward'
16     },
17     rejectedPlans: []
18   },
19   executionStats: {
20     executionSuccess: true,
21     nReturned: 1,
22     executionTimeMillis: 6,
23     totalKeysExamined: 0,
24     totalDocsExamined: 8339,
25     executionStages: {
26       stage: 'COLLSCAN',
27       filter: { username: { '$eq': 'Abbie Bernstein' } },
28       nReturned: 1,
```

```
      executionTimeMillisEstimate: 0,
      works: 8341,
      advanced: 1,
      needTime: 8339,
      needYield: 0,
      saveState: 8,
      restoreState: 8,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 8339
    }
  },
  command: {
    find: 'user',
    filter: { username: 'Abbie Bernstein' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673280753, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673280753, i: 1 })
}
```

Listing 8.20: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: { username: { '$eq': 'Abbie Bernstein' } },
    queryHash: '7D9BB680',
    planCacheKey: '24069050',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
```

```
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { username: 1 },
        indexName: 'username_1',
        isMultiKey: false,
        multiKeyPaths: { username: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { username: [ '["Abbie Bernstein", "Abbie Bernstein"]' ]
    }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 1,
    executionTimeMillis: 1,
    totalKeysExamined: 1,
    totalDocsExamined: 1,
    executionStages: {
      stage: 'FETCH',
      nReturned: 1,
      executionTimeMillisEstimate: 1,
      works: 2,
      advanced: 1,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 1,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 1,
        executionTimeMillisEstimate: 1,
        works: 2,
        advanced: 1,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { username: 1 },
        indexName: 'username_1',
        isMultiKey: false,
        multiKeyPaths: { username: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: { username: [ '["Abbie Bernstein", "Abbie Bernstein"]' ]
    },
        keysExamined: 1,
        seeks: 1,
        dupsTested: 0,
```

```
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'user',
    filter: { username: 'Abbie Bernstein' },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285013, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285013, i: 1 })
}
```

Listing 8.21: Test

D .2 date of birth

Before the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: {
      date_of_birth: {
        '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard
    Time)'
      }
    },
    queryHash: 'D7A0117C',
    planCacheKey: 'D7A0117C',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
```

```
      stage: 'COLLSCAN',
      filter: {
        date_of_birth: {
          '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
        }
      },
      direction: 'forward'
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 32,
    totalKeysExamined: 0,
    totalDocsExamined: 8339,
    executionStages: {
      stage: 'COLLSCAN',
      filter: {
        date_of_birth: {
          '$eq': 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
        }
      },
      nReturned: 0,
      executionTimeMillisEstimate: 23,
      works: 8341,
      advanced: 0,
      needTime: 8340,
      needYield: 0,
      saveState: 8,
      restoreState: 8,
      isEOF: 1,
      direction: 'forward',
      docsExamined: 8339
    }
  },
  command: {
    find: 'user',
    filter: {
      date_of_birth: 'Mon Jan 09 2023 17:05:07 GMT+0000 (Western European Standard Time)'
    },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
```

```
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673283903, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673283903, i: 1 })
}
```

Listing 8.22: Test

After the index

```
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'rottenMovies.user',
    indexFilterSet: false,
    parsedQuery: {
      date_of_birth: {
        '$eq': 'Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)'
      }
    },
    queryHash: 'D7A0117C',
    planCacheKey: '90F68BB6',
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    winningPlan: {
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { date_of_birth: 1 },
        indexName: 'date_of_birth_1',
        isMultiKey: false,
        multiKeyPaths: { date_of_birth: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: {
          date_of_birth: [
            '["Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)", "Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)"]'
          ]
        }
      }
    },
    rejectedPlans: []
  },
  executionStats: {
    executionSuccess: true,
    nReturned: 0,
    executionTimeMillis: 1,
    totalKeysExamined: 0,
```

```
    totalDocsExamined: 0,
    executionStages: {
      stage: 'FETCH',
      nReturned: 0,
      executionTimeMillisEstimate: 0,
      works: 1,
      advanced: 0,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      docsExamined: 0,
      alreadyHasObj: 0,
      inputStage: {
        stage: 'IXSCAN',
        nReturned: 0,
        executionTimeMillisEstimate: 0,
        works: 1,
        advanced: 0,
        needTime: 0,
        needYield: 0,
        saveState: 0,
        restoreState: 0,
        isEOF: 1,
        keyPattern: { date_of_birth: 1 },
        indexName: 'date_of_birth_1',
        isMultiKey: false,
        multiKeyPaths: { date_of_birth: [] },
        isUnique: false,
        isSparse: false,
        isPartial: false,
        indexVersion: 2,
        direction: 'forward',
        indexBounds: {
          date_of_birth: [
            '["Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)", "Mon Jan 09 2023 17:24:42 GMT+0000 (Western European Standard
    Time)"]'
          ]
        },
        keysExamined: 0,
        seeks: 1,
        dupsTested: 0,
        dupsDropped: 0
      }
    }
  },
  command: {
    find: 'user',
    filter: {
      date_of_birth: 'Mon Jan 09 2023 17:24:42 GMT+0000 (Western European
    Standard Time)'
    },
    '$db': 'rottenMovies'
  },
  serverInfo: {
    host: 'Profile2022LARGE10',
    port: 27017,
    version: '6.0.3',
    gitVersion: 'f803681c3ae19817d31958965850193de067c516'
```

```
  },
  serverParameters: {
    internalQueryFacetBufferSizeBytes: 104857600,
    internalQueryFacetMaxOutputDocSizeBytes: 104857600,
    internalLookupStageIntermediateDocumentMaxSizeBytes: 104857600,
    internalDocumentSourceGroupMaxMemoryBytes: 104857600,
    internalQueryMaxBlockingSortMemoryUsageBytes: 104857600,
    internalQueryProhibitBlockingMergeOnMongoS: 0,
    internalQueryMaxAddToSetBytes: 104857600,
    internalDocumentSourceSetWindowFieldsMaxMemoryBytes: 104857600
  },
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673285073, i: 1 }),
    signature: {
      hash: Binary(Buffer.from("0000000000000000000000000000000000000000", "
    hex"), 0),
      keyId: Long("0")
    }
  },
  operationTime: Timestamp({ t: 1673285073, i: 1 })
}
```

Listing 8.23: Test

E Application code

E .1 Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.
    org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache
    .org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.0</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>it.unipi.dii.lsmsdb</groupId>
    <artifactId>rottenMovies</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>rottenMovies</name>
    <description>Project for the rotten movies service</description>
    <properties>
        <java.version>19</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
```

```
28          <dependency>
29              <groupId>org.springframework.boot</groupId>
30              <artifactId>spring-boot-starter-data-mongodb</artifactId>
31          </dependency>
32          <dependency>
33              <groupId>org.springframework.boot</groupId>
34              <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
35          </dependency>
36          <dependency>
37              <groupId>org.springframework.boot</groupId>
38              <artifactId>spring-boot-starter-data-neo4j</artifactId>
39          </dependency>
40
41          <dependency>
42              <groupId>org.springframework.boot</groupId>
43              <artifactId>spring-boot-starter-test</artifactId>
44              <scope>test</scope>
45          </dependency>
46          <dependency>
47              <groupId>io.projectreactor</groupId>
48              <artifactId>reactor-test</artifactId>
49              <scope>test</scope>
50          </dependency>
51          <dependency>
52              <groupId>com.google.code.gson</groupId>
53              <artifactId>gson</artifactId>
54              <version>2.10</version>
55          </dependency>
56          <dependency>
57              <groupId>com.fasterxml.jackson.core</groupId>
58              <artifactId>jackson-annotations</artifactId>
59              <version>2.14.1</version>
60          </dependency>
61          <dependency>
62              <groupId>com.fasterxml.jackson.core</groupId>
63              <artifactId>jackson-databind</artifactId>
64              <version>2.14.0</version>
65          </dependency>
66          <dependency>
67              <groupId>org.neo4j.driver</groupId>
68              <artifactId>neo4j-java-driver</artifactId>
69              <version>5.3.0</version>
70          </dependency>
71      </dependencies>
72
73      <build>
74          <plugins>
75              <plugin>
76                  <groupId>org.springframework.boot</groupId>
77                  <artifactId>spring-boot-maven-plugin</artifactId>
78              </plugin>
79          </plugins>
80      </build>
81
82  </project>
```

Listing 8.24: Test