UNIVERSITÀ DI PISA

Second Cycle Degree in Computer Engineering

Final Project Report
Workgroup Task 2 + Task 3

Large Scale and Multi-Structured Databases

T. Billi, E. Casapieri, A. Di Donato, A. Khalil

Academic Year 2019/2020

Contents

Chapter 1

Design

1.1 Description of the application

A movie review allows viewers to know and understand the whole picture of the movie. In fact, it is an efficient and fast way to obtain information regarding the quality of a movie. Nowadays, with the continuous spreading of the Internet all over the world, the way people connect with each other and share information has changed. The world of film reviews is reforming as well. From traditional critics of experts to review of ordinary audience, the cyber-world offers a new and effective channel for audiences sharing their comments about movies. Additionally, since the number of movies made and released keeps growing every year, it is hard to keep track of every movie ever made. For these reasons, movie databases were born, along with movies review websites/applications.

The aim of this application is to help the users to search a movie, view its details, read reviews from other users and help the user to find where to watch it. The application provides also some statistical analyses on the users preferences and movies data. A system administrator can perform some management operations, like adding or removing movies.

1.2 Requirement analysis

1.2.1 Functional requirements

The main functional requirements provided by this application to the system actors (user and administrator) are the following:

(i) **Registration.** The System provides a registration function to the user that wants to create and maintain a list of watched/favorite movies. Only registered users can access to the application.

(ii) **Login.** User logins to the system by entering valid credentials (username and password) to create and maintain a list of watched movies, rate/review and comments them.

(iii) **Search movies.** The system provides a search function. User can search movies based on their title and/or production year.

(iv) **Display movie details.** After the user searches the movie, the system displays the film details:

- Title
- Synopsis
- Poster
- Year
- Directors
- Countries
- Genres
- Rates
- Box-office
- Comments and their points
- Actors
- Awards
- Platform
- Runtime

(v) **Statistical analyses.** The system provides some statistical analyses, namely:

- top 10 most viewed movies by genre
- most watched movies in the last month
- how the nationalities of the viewers are distributed, for a given movie

(vi) **Movies recommendation** The system provides some functions to suggest movies to the user, each one based on different criteria:

- Selected a movie, the system shows a list of similar movies based on the most number of genres in common with the selected one.
- Analyzing the user's watched list, it is shown a list of suggested movies ordered by the number of actors and directors in common with one or more movies watched by the user.
- Given the user's watched list, the system displays a list of movies that has been highly rated by other users whom rated one or more movies in a similar way with the given user.

(vii) **Logout.** After logging in and using the system's functionalities , the customer may log out.

(viii) **Rate/review a movie.** After searching the film, the user can rate the film and/or write a review for it.

(ix) **Upvote/downvote a comment.** The system provides a feature for the user to like/dislike a film's comment.

(x) **Remove movies.** The system provides a feature for administrator to remove movies.

(xi) **Watched/Favorite movies list.** The system provides a feature for the user to manage their watched and favorite movies lists. In order to add a movie to the watched list, the user is required to give a rating to the movie.

(xii) **Delete users.** The system provides a feature for the administrator to remove users.

(xiii) **Request to add new movies.** The system provides the user the possibility of requesting the addition of a movie, if not present on the system. The request may be either accepted or rejected by the administrator.

(xiv) **Accept/ignore requests to add new movies.** The system provides a feature to the administrator to accept or ignore a request of adding a new movie. When the movie is added by the administrator a notification will be sent to the user who has sent the request.

(xv) **Add movie** The administrator has the possibility to add manually a new movie and its details to the movie's database. However this functionality will be activated only in case that the movie is not present previously in the database.

1.2.2 Non-functional requirements

(i) **Privacy.** The privacy of the users should be guaranteed in the system. The user data (address, nationality, watched list, favorite list) can't be viewed by the other users.

(ii) **Consistency.** The system shall provide consistency of the data between both the database. Each operation performed should update either both of the databases, or neither of them.

(iii) **Scalability.** The system shall have the potential of being scalable horizontally to handle high traffic loads, peak spike of connections and large number of users.

(iv) **Usability.** The user must understand what the system does and feel satisfied with the system. In Addition, the system should be easy to learn and appropriate to use.

(v) **Performance.** The system should promptly display the results of searching queries.

1.3 Use cases

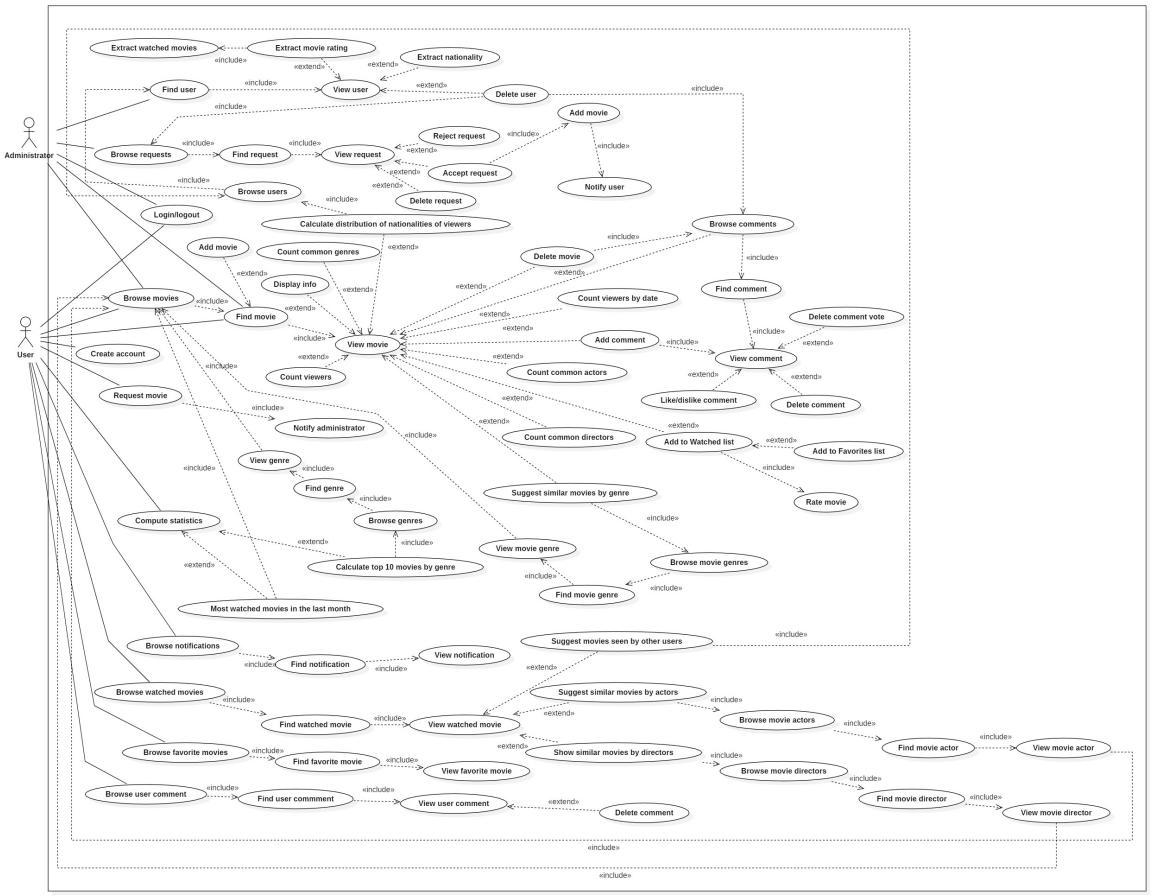

Figure 1.1: Use case diagram of the application.

1.4 Analysis Classes

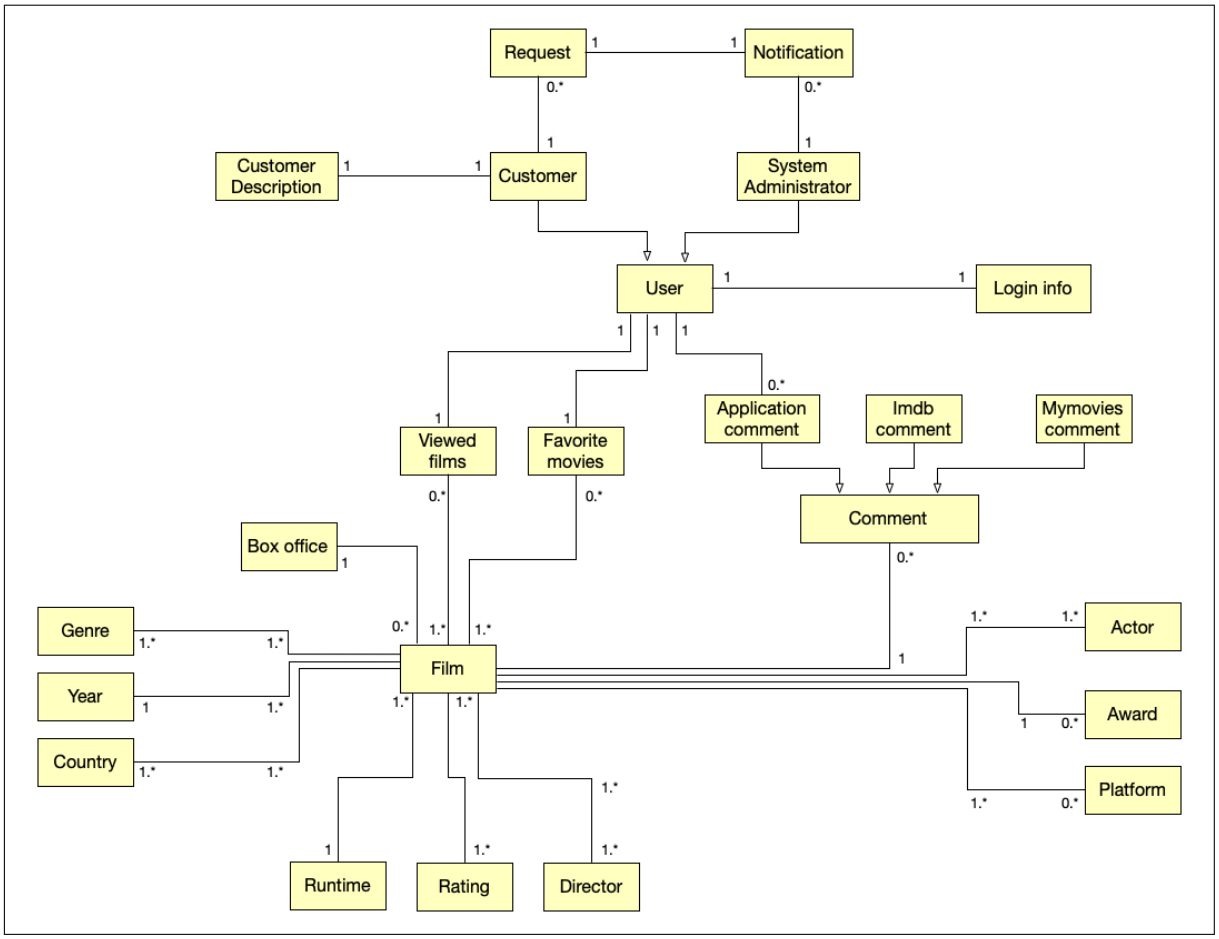

Figure 1.2: Analysis classes of the application.

1.5 Data modeling

1.5.1 MongoDB

The dataset has been organized into three collections. The first one concerns information about the users who are registered on our application. The documents in this collection share a common structure and belong to the same entity; for this reason we thought of putting them into the same collection. The second collection concerns information about film offered to the users by the application. As we have collected the same relevant data for each film, documents have the same attributes and so can be put into the same collection. If next document of a film needs more attributes, they can be added without any problem because everything is schemeless. However, it's convenient to put this kind of documents together because filtering collections is often slower than working directly with multiple collections, each of which contains a single document type. The third collection concerns movie reviews made both by users of our application and by user of real websites like Imdb and MyMovies. In the comment document there is a type field that differentiates the three kinds of comment.

The relationships between entities have been handled as follows:

- The one-to-one relationship between user and watched films list has been handled by adding a new key-value pair into the user documents, where the key is the name of the list and the value is an array of documents. The embedded information are the film Id and other data of the film often needed together like date, needed by the statistics, Italian title, rating and status that points out if that watched film is favorited as well. That array implements the many-to-many relationship between watched film list and film. Indeed, the latter relationship has been modeled so that each collection (user and film) maintains a list of identifiers that reference related entities.

- The same occurs with the one-to-one relationship between user and favorite films list and the many-to-many relationship between favorite films list and film entity. Since a film can be favorite if and only if that film has been already watched by a certain user, we added a boolean field called status inside the document related to the watched film. Moreover, in the film collection we maintain an array of Ids belongs to the users that put the film among their favorite.

- The one-to-many relationship between film and comment has been handled by using linked document. Film entity is the primary document and the instances of comment are represented as an array of comment Ids that reference the comment documents within the comment collection. Since in the application code, when we are retrieving all comments of a certain film, we already know its Id, we query just the comments collection in order to avoid a join operation. Indeed, a film_id field has been added to the comment document to simplify the search of all comments of a film.

- The one-to-many relationship between user and comment has been handled by using linked documents. An array of comment identifiers reference the corresponding comment document in comments collection.

- The one-to-many relationship between user and liked comments has been handled by using linked documents. Indeed, a new key-value pair has been added to the user document where the value is an array of documents. Each document contains the identifier and the action which a user can do on that comment, namely like or dislike.

- The one-to-many relationship between user and messages has been handled by using embedded documents. User entity is the primary document and the instances of message are represented as an array of embedded documents. When a user makes a film request, a new message document will be added to the document of that user and to the document of the administrator. The administrator only can change the status of a request and the modification is visible by the user.

```
collection users:
    {
        UserID:,
        username: "",
        password: "",
        general_info:{ first_name: "",
                       last_name: "",
                       country:"",
                       year_of_birth:""
                     },
        role: ,
        liked_comments: {[  comment_id: ObjectId(""),
                            action:]},
        user_comments:[{comment_id: ObjectId(""}],
        favorite_film: [{ film_id:ObjectId(""),
                          grade:
                       }],
        messages:[{ _id: ObjectId(""),
                    film_title: "",
                    date: ,
                    status:,
                    user_id: ObjectId(""}
                  }],
        watched_films: [{ film_id: ObjectId(""),
                          italian_title:"",
                          date:,
                          favorite:,
                          rating:}]

        //role => 0 -> admin, 1 -> user
        //action => 1 -> like, -1 -> dislike
        //status => 1 -> rejected, 2-> accepted, 0-> waiting
    }


collection films:
    {
        _id: ObjectId(""),
        original_title: "",
        italian_title: "",
        synopsis: "",
        poster: "",
        year: "",
        countries: [{}],
        genres: [{}],
        runtime:,
        ratings: { imdb_rating: ,
                   comingsoon_rating:,
                   mymovies_rating:,
                   application_rating:
                 },
        directors:[{}]
        box_office: ,
        watched_by: [{ user_id: ObjectId(""),
                       date:
                    }],
        favorite_by: [{user_id: ObjectId(""}],
        comments:[comment_id: ObjectId("")],
        cast: [{   name: "",
                   role: ""
               }],
        awards: [{}],
        available_on: [{ platform: "",
                         price:""
                      }],
        film_ratings: [{ user:"",
                         country:"",
                         rating:
                      }]
    }


collection comments:
    {
        _id:ObjectId(""),
        film_id: Object_id(""),
        author:"",
        text:"",
        date:,
        type:"",
        comment_points:,
        user_rating:

        //type => "imdb_comment" or "mymovies_comment" or "comingsoon_comment"
    }
```

1.5.2 Neo4j

A brief description of the graph in terms of entities and relationships follows. The entities corresponding to the nodes of the graph are:

- Registered users. Each user has the following properties:
  - User node ID
  - Username
- Movies. Each movie has the following properties:
  - Movie node ID
  - Italian title
  - Original title
  - Year
  - MongoDB movie document ID, needed to do CRUD operations in both databases referring to the same movie, like addition and deletion.
- Genres. Each genre has the following properties:
  - Genre node ID
  - Name of the genre
- Actors. Each actor has the following properties:
  - Actor node ID
  - Name of the actor
- Directors. Each director has the following properties:
  - Director node ID
  - Name of director

The relationships corresponding to the edges of the graph are:

- Many-to-many relationships between users and movies added to their watched list, that are all the directed edges starting from a user node and ending to a movie node with ADDED label. Each of them has the following properties:
  - Edge Id
  - Date
  - Rating
- Many-to-many relationships between movies and their genres, that are all the directed edge starting from a movie node and ending to a genre node with LABELED label. Each of them does not have any property except the edge Id.
- Many-to-many relationships between movies and the actors, that are all the directed edge starting from an actor node and ending to a movie node with ACTED_IN label. Each of them does not have any property except the edge Id.

- Many-to-many relationships between movies and the directors, that are all the directed edge starting from a director node and ending to a movie node with DIRECTED label. Each of them does not have any property except the edge Id

Above, in Figure 1.3, there's a snapshot of the graph; orange nodes are the users, green the actors, blue the films and in brown the actors.

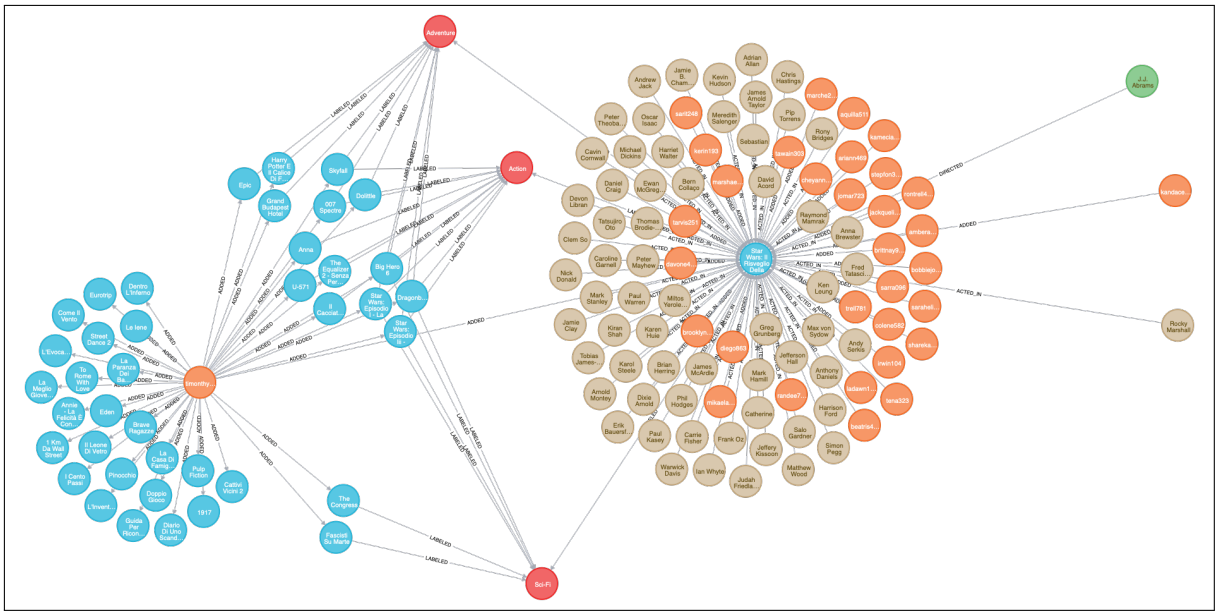

Figure 1.3: Graph's snapshot

1.6 Graph queries defined

The following queries are the typical graph queries implemented inside the application. For each of them it is provided the domain specific version and graph-centric version.

- **Domain-specific**: Rank, in descending order, movies that have the highest possible number of genres in common with a given movie. Of these resulting movies, only the ones that have a number of genres closest to the number of common genres are picked.
  Graph-centric: Count the number of outgoing edges from a given Movie node. For each Genre node connected to the starting Movie node, other Movie nodes are detected by analyzing the incoming edges. For each of the Movie nodes found this way, we count how many Genre nodes that are also connected to the starting Movie node they are connected to (number of genres in common). Furthermore, for each of the newly found Movie nodes, we count how many of the outgoing edges are incident to Genre nodes (this is the total number of genres for each Movie). The Movies are then ordered based on the difference between the number of total genres of the target Movie node and the number of genres in common with this, and also on the total number of genres of the returned Movie nodes.

- **Domain-specific**: Return a list of Movies seen by other users that have similarly rated Movies seen by both them and the starting user, that were given a positive

rating.
Graph-centric: Find all the Movie nodes the starting User node is connected to. For each of these nodes, find the other User nodes they are connected to, and filter out the ones where the difference between the rating property of the outgoing edge from the newly found User nodes and the starting node is greater than 2. For each of the User nodes satisfying this condition we find which Movie nodes they are connected to. This Movies nodes are then ranked based on the average of the rating property of all the incoming edges for each Movie node.

- **Domain-specific**: Rank, in descending order, movies that have the highest possible number of actors in common with the movies already seen by the user.
  Graph-centric: Find all the Movie nodes connected to the starting User node. For each of these we find which Actor nodes they are connected to. From there, other Movie nodes are found by considering the outgoing edges from the Actor nodes. For each Movie node found this way, we count the number of the previously found Actor nodes connected to them (this is the number of common actors with the starting Movie node). The Movie nodes are then sorted in descending order based on the number of common actors.

- **Domain-specific**: Rank, in descending order, movies that have the highest possible number of directors in common with the movies already seen by the user.
  Graph-centric: Find all the Movie nodes connected to the starting User node. For each of these we find which Director nodes they are connected to. From there, other Movie nodes are found by considering the outgoing edges from the Director nodes. For each Movie node found this way, we count the number of the previously found Director nodes connected to them (this is the number of common directors with the starting Movie node). The Movie nodes are then sorted in descending order based on the number of common directors.

1.7 System architecture

The architecture of the application is a client-server. The application uses a no-SQL database (MongoDB) to store and retrieve the big amount of movies and users data. In addition, in order to improve the execution performance of some specific functionalities like movie recommendations, the application interacts with a graph database (neo4j) in which a portion of MongoDB data is stored and represented as entities (nodes) and relationships between them (edges). Indeed, data stored and structured this way are suitable for relationships-based queries and network traversal. The CRUD operations done on MongoDB server on those portion of data, are also done on Neo4j server in order to maintain data consistency.

The reason that MongoDB is used instead of a relational database management system like MySQL, is that can store and maintain large volumes of data without a fixed structure (schema). Since, MongoDB has a flexible document-oriented data model, it allows you to represent hierarchical and complex structures in an easy way and to obtain faster query response to access your documents by indexing them.

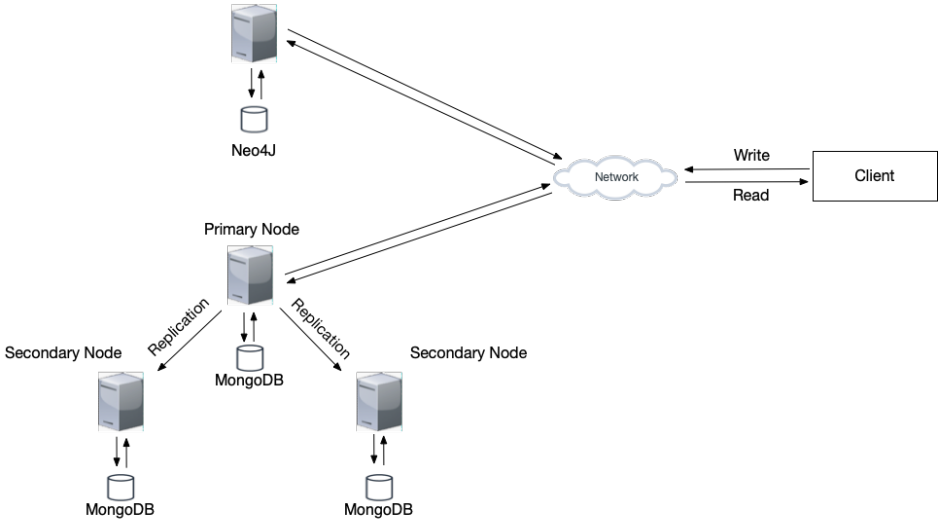

Figure 1.4: System's Architecture

Moreover, MongoDB offers the possibility to scale horizontally (dividing the application dataset and distribute them over multiple servers) your system resources by using a cluster and a replica set, instead of scaling it vertically (increasing the capacity of a single server), which may be costly and limited by the technology used. Replication increase the level of system's availability, keeps your application running and your data safe, even if something happens to one or more of your servers. For this reason, our application uses a replica set composed of three nodes (servers), one of them is set as the primary node, while the two others nodes as secondary, the replica set architecture is showed in Figure 1.4.

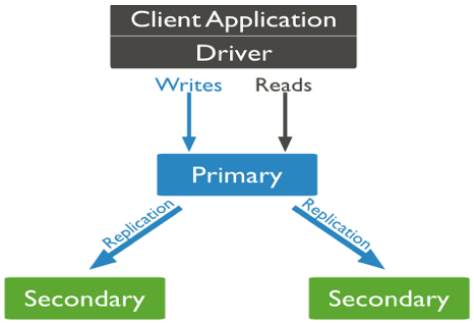

Figure 1.5: The MongoDB Replication diagram

Java programming language is used to implement the client-server application, the Java MongoDB driver was used to interconnect and communicate with the database. While ,the dataset of the database is populated with data obtained from the websites IMDB,MyMovies and Coomingsoon using a scraping script written in Python.

Chapter 2

Implementation and test of the application

2.1 Replication

One of the key features of our application is the possibility for users to keep their movie lists always up to date, and the ability to publicly comment on every movie. For this reason, we decided to have our application stand on the *CP* edge of the *CAP* triangle.

The system is deployed on a three-node-cluster replica set; one of them is elected to be the primary, while the others are secondary. The read/write setting is set to always issue both operation to the primary server, and the *writeConcern* setting is set to "W2", meaning that the write operations are only acknowledged when the data has been written to at least two of the three replicas. If the primary node goes down, one of the remaining to nodes is elected as new primary. At least two servers need to be up and running in order to have a node that can be elected to be a primary. This way, whenever a write operation is issued, the new data is written to the primary and immediately replicated to one of the secondary nodes, and consistency is ensured.

On the other hand, when two out of the three servers are unavailable, we find ourselves in the situation where there is only one node, that node being a secondary. In this scenario, the application can't work properly, as neither read nor write operations can be completed. After a timeout period has elapsed without the application being able to connect to a primary node, the application itself is closed, displaying a message to the user and informing them that the service is temporarily unavailable.

This behavior allows the applications to be consistent, but not available, for the following reasons:

- If the primary node disconnects from the cluster, it takes a few seconds to elect a new one. Hence, the service is unavailable for writes and reads.
- A client can always disconnect from the primary server due to network partition even if both client and leader node is running fine, hence making it unavailable.

2.2 Description of the main modules of the software

A brief description of the main modules of the application follows:

- A DaoMongo class has been defined so that the other modules of the application can connect and interact with mongoDB/Neo4j cluster in order to do CRUD operations or calculate statistics. Just one MongoClient instance (private member of this class) must be created for a given MongoDB deployment (replica set in our case) and used across the application; the same followed for Neo4j to connect and interact with its server. To guarantee the creation of a unique instance, we made use of Singleton pattern. At the end of the application MongoClient.close() and driver.close() are called to clean up resources.

- BasicFilm class represents a film with the essential information to be displayed on the film table. This table is split into pages and every time a page is visualized, each film belonging to that page is mapped to the corresponding BasicFilm object. This class contains much less information than the Film class and it is used to not make the loading of the film table too slower.

- Film class represents a film with all the information retrieved from the database. A Film object is created whenever the user select a certain film on the film table because wants to know all the details of it.

- Actor class represents an actor and includes data about the name and the role. A list of Actor object is put inside the Film object in the same way that an actor document is embedded into a film document in the data model.

- Comment class represents a comment of a certain film. A list of Comment object is put inside the Film object in the same way that the one-to-many relationship is implemented in the data model by using a comment ids list.

- The two actors of the application have been mapped into two different classes. User class represents a normal user that makes use of the services of the application. A User object is created after login. This class has also a list of Watched_film, a list of Comment and a list of Request. These fields implement in the class diagram the one-to-many relationships between user and these other entities. Admin class represents the administrator of the service containing methods that correspond to the special functionalities of this actor. Unlike the User class, this class has a list of RequestAdmin object that are the requests sent to the admin, instead of the same one-to-many relationships of the User class.

- WatchedFilm class contains all the information needed from the time when a film is added to the watched film list of a certain user. These data are: the data when the film has been watched, the assigned rating and the status to distinguish if the film is favorite as well. These class is linked by a one-to-many relationship with user class.

- Request class represents a request that a user can send to the admin in order to ask for the addition of a new film to the database. A user can scan and check the status of all his previous requests. RequestAdmin is derived from the previous class and represents the request arrived to the admin. This class adds a new field

that stores the Id of the user who made the request; this field is useful to reply properly to the user.

2.3 Description of the performed tests

2.3.1 Indexes analysis

Following there is the analysis of some queries performance on MongoDB:

- **Index to sort the movies by italian title in ascending order.** We first issue the following query, used inside the application to display 40 movies every time the user click on previous or next button; doing that it's avoided the load of all movies documents inside the respective collection every time the user log in.

```
$ db.films.explain('executionStats').aggregate(
    [
        {$sort: {"italian_title": 1}},
        {$skip: 0*40},
        {$limit: 40},
        {$project : { "italian_title" : 1 , "year" : 1, "runtime" : 1} }
    ]
)
```

  The output of the *executionStats*, without using the index on italian title is:

```
...
    "nReturned" : 5477,
    "executionTimeMillis" : 12,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 5477
...
```

  where:

  - *nReturned* is the number of documents that match the query condition.
  - *executionTimeMillis* is the total time in milliseconds required for query plan selection and query execution
  - *totalKeysExamined* is the number of index entries scanned
  - *totalDocsExamined* is the number of documents examined during query execution

  After the introduction of the index on the italian title , the output is the following:

```
...
    "nReturned" : 40,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 40,
    "totalDocsExamined" : 40
...
```

  From the second output it comes out that with the index above to answer the query the system need to scan only 40 documents against the 5477 scanned before without the index; so it is clear why the introduction of the index on italian title. It should be noted that the same index improve the performance of the following query used to count the users, divided by their nationality, that watched a movie; this query it is used to compute the statistics available to the admin to see how are distributed, in percentage, the users for a certain movie in term of countries.

```
$ db.films.explain('executionStats').aggregate(
    [
        {$match: {"italian_title": "Pulp Fiction"}},
        {$unwind: "$film_ratings"},
        {$group: {_id : {"country": "$film_ratings.country"}, viewers: { "$sum":
            1}}},
    ]
)
```

Without the index this is the output:

```
...
    "nReturned" : 1,
    "executionTimeMillis" : 4,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 5477
...
```

With the introduction of the index the output is the following:

```
...
    "nReturned" : 1,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1
...
```

- **Index to sort the users by username in ascending order**. At the login all the information about the user that want to log in have to be retrieved. The query to do that is:

```
$ db.users.explain('executionStats').find({"username": "ernesto831"})
```

which return the following document:

```
...
    "nReturned" : 1,
    "executionTimeMillis" : 5,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 10009
...
```

instead, with the introduction of the index on user's username:

```
...
   "nReturned" : 1,
   "executionTimeMillis" : 0,
   "totalKeysExamined" : 1,
   "totalDocsExamined" : 1
...
```

From that, given the fact that from the second output the system take 0ms to execute against 5ms, it's been introduced the use of an index on the user's username.

- **Index to sort the movies by the year of production in ascending order**. The System, how said before, give the possibility to both user and admin to search all the movie with a certain year of production. So every time one of the actors of the system ask to do that, this query is issued:

```
$ db.films.explain('executionStats').find({ "year": 2020})
```

without an index on the year of production, the output is the following:

```
...
    "nReturned" : 13,
    "executionTimeMillis" : 5,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 5477
...
```

with the index:

```
...
    "nReturned" : 13,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 13,
    "totalDocsExamined" : 13
...
```

As can be seen, with and index on year of production the performance are improved. That's why it's been used this index.

- **Index to sort the comments by the author in ascending order**. From his profile panel, after he logged in, the user can scroll through the comments he has done on the database's movie; from there e can also delete them. To ensure that the system have to load from the database the user's comment issuing this query:

  ```
  $ db.comments.explain('executionStats').find({"author": "sarastro7"})
  ```

  If there is not the index on the author it is necessary to scan all the documents of comments collection, as we can see from the output.

  ```
  ...
      "nReturned" : 7,
      "executionTimeMillis" : 53,
      "totalKeysExamined" : 0,
      "totalDocsExamined" : 82058
  ...
  ```

  After the introduction of the index the performance are definitely better, as we can see:

  ```
  ...
      "nReturned" : 7,
      "executionTimeMillis" : 0,
      "totalKeysExamined" : 7,
      "totalDocsExamined" : 7
  ...
  ```

- **Compound index to sort the comment by the film Id in ascending order and by the date of creation in descending order**. Every time a user or the admin click on a movie, the system will display only maximum 10 comments of all the comments related to it. To do that it is issued this query:

  ```
  $ db.comments.explain('executionStats').aggregate(
      [
        {"$match": {"film_id": ObjectId("5df7957791050c2492fd7d1c")}},
        {$sort: {"date": -1}},
        {$limit: 10},
      ]
    )
  ```

Without the compound index the output is:

```
...
    "nReturned" : 23,
    "executionTimeMillis" : 43,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 85964
...
```

introducing an index only on the film Id the result is the following:

```
...
    "nReturned" : 23,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 23,
    "totalDocsExamined" : 23
...
```

using instead the compound index, the output is better:

```
...
    "nReturned" : 10,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 10,
    "totalDocsExamined" : 10,
...
```

It should also be noticed that to implement the functionality of searching a movie using the original title or the italian title it is been used a text index on both type of title; text index, indeed, is and index to support text search queries on string content.
It has been also carried out a brief analysis of queries performance on Neo4j:

- **Index to sort the user by username in ascending order**. One of the functionality introduced thanks to the graph database is the suggestion to the user of movies seen by other users. The list of suggested movies it is displayed in the panel *Seen by others*; the query issued by the system is:

```
EXPLAIN MATCH (me:User {username:'timonthy198'})-[my:ADDED]->(m:Film)
MATCH (other:User)-[their:ADDED]->(m)
WHERE me <> other
AND abs(my.rating - their.rating) < 2
WITH other,m
MATCH (other)-[otherRating:ADDED]->(movie:Film)
WHERE movie <> m
WITH avg(otherRating.rating) AS avgRating, movie
RETURN movie
ORDER BY avgRating desc
LIMIT 25
```

  Without the using of the index on username the system to answer to the query have to perform the NodeByLabelScan namely the scan of all the user nodes. Using the *EXPLAIN* clause the output indeed is: *10007 estimated rows*. With the index the scan is done using NodeIndexSeek namely it use only the index on the username. In this way the output of the query using *EXPLAIN* clause is *1 estimated rows*; this means that is done only the scan of one node. It is clear the the improvement of the performance of the query after the introduction of the index on the username.

- **Index to sort the movies by the film id in MongoDB in ascending order**. For every movies of the database the system suggest to the user similar movies, issuing the following query every time the user click on a certain movie.

```
EXPLAIN CALL { MATCH (m1:Film {_id :'5df9dd483e51ac2360d5f777'})
               MATCH (m1)-[r1:LABELED]->(g1:Genre)
               RETURN count(r1) as NumGenresTarget
             }
MATCH (m:Film {_id :'5df9dd483e51ac2360d5f777'})
MATCH (m)-[:LABELED]->(g:Genre)
MATCH (movie:Film)-[r:LABELED]->(g)
MATCH (movie)-[r2:LABELED]->(g1: Genre)
WHERE m <> movie
WITH movie, count(DISTINCT r) AS commonGenres, count(DISTINCT r2) AS
    totalGenresAdvice, NumGenresTarget
RETURN movie.italian_title as Title
ORDER BY (NumGenresTarget - commonGenres), totalGenresAdvice
```

Without the index on the film id in MongoDB the system does the scan of all film nodes, instead with the index it need the scan of only one node to answer to the query.

2.4 Performed tests to verify the consistency between MongoDB and Neo4j

Some of the operations within the application involve both MongoDB and Neo4j. In order to ensure consistency when performing these write operations, they are performed within a MongoDB transaction function. First, the data is written to the MongoDB database. Then, the write operation is also attempted on the Neo4j database. Write operations on Neo4j are set to be retried for 30 seconds. When this timeout has elapsed, if an exception is thrown, the transaction is also rolled back from MongoDB, and the application is closed (login is disabled until both MongoDB and Neo4j are available again).

The following test have been performed on each of the critical operations involving both databases, as they all have a similar *modus operandi*.

In order to login into the application, at first, both Neo4j and MongoDB need to be up and running. If MongoDB is unavailable when the transaction starts (two out of the three replicas are down), the transaction is never executed. If, on the other hand, Neo4j becomes unavailable before or during this operation, a flag is set and the whole operation is also rolled back on MongoDB.

2.5 Manual of usage of the application

Once the application is launched, the user is prompted with a login window (figure 2.1). An already registered user can enter his credentials and then press the "login" button to access the user panel (see figure 2.3), while a new user accesses the registration window and can create a new profile by clicking on the "sign up" button as showed in figure 2.2, in order to complete the registration the user must enter correct and complete data, some of these will be used for the statistics and analytics. If the user inserts duplicated usernames, invalid year of birth or empty textboxes an error message will appear.

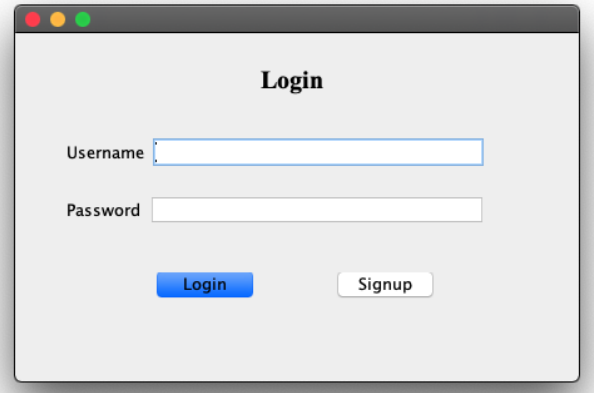

Figure 2.1: Login Window

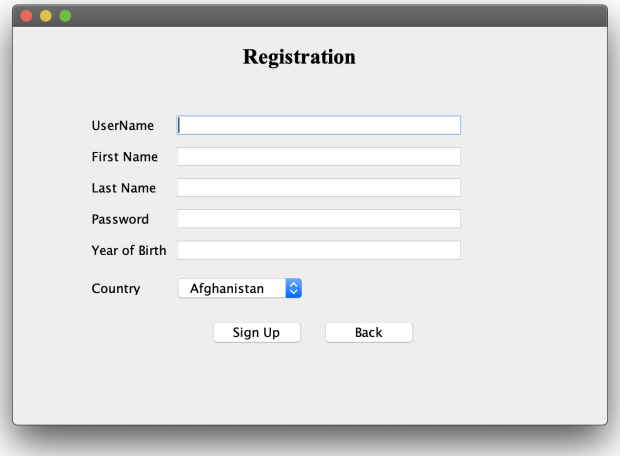

Figure 2.2: Registration Window

2.5.1 User

In case the user complete the registration process successfully, or it logins with correct credentials, the user window will appear, which from the left side of it he can access a number of panels with different functionalities:

(i) profile panel (figure 2.3): this is the main panel of the user window, in this the user can review his information, change his password by inserts the old, the new password and clicks on "change password" button. Furthermore, he can browse his own comments.

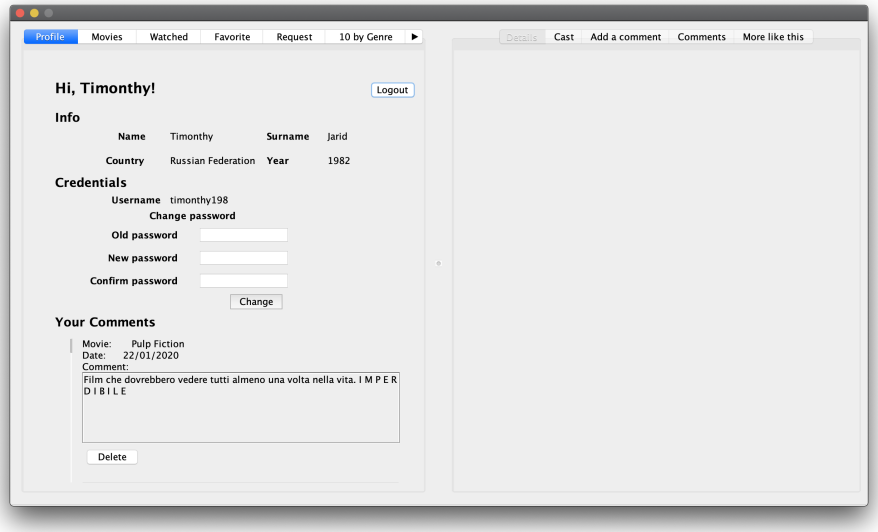

Figure 2.3: User's Panel

(ii) Movies panel: in this panel the user can view and browse the list of all available movies(Figure 2.4); to search a movie the user must specify its title and/or its production year and click on "Search" button(Figure 2.5 - Figure 2.6). When the user clicks on a movie from the list, the right panels will be activated and populated. From these panels the user can view the movie's details(Figure 2.7), the cast(Figure 2.8), user comments and like/dislike their comments(Figure 2.9 - Figure 2.10). In addition, the user can add a comment to the movie by using the "add a comment" panel(Figure 2.11). He can rate the movie and inserts it in his watched list by clicking on the add button. In the right panel it is also shown a list of similar movies to the current selected. (Figure 2.12)

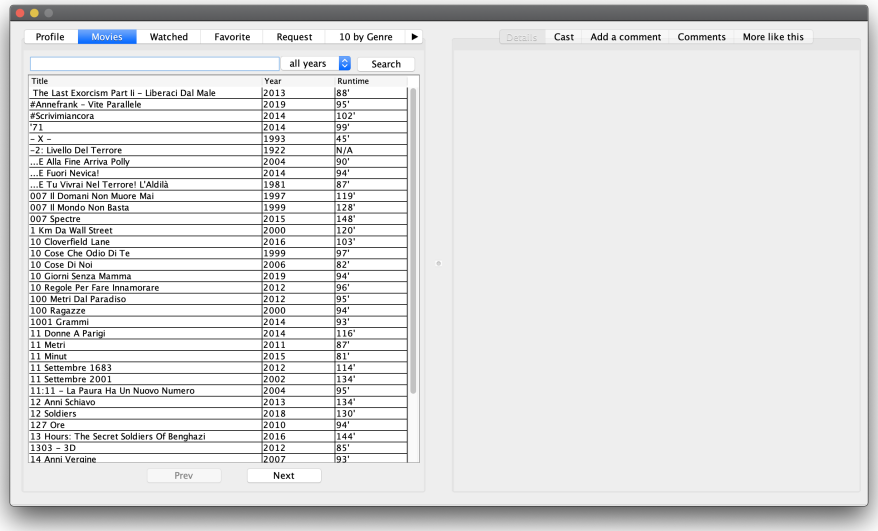

Figure 2.4

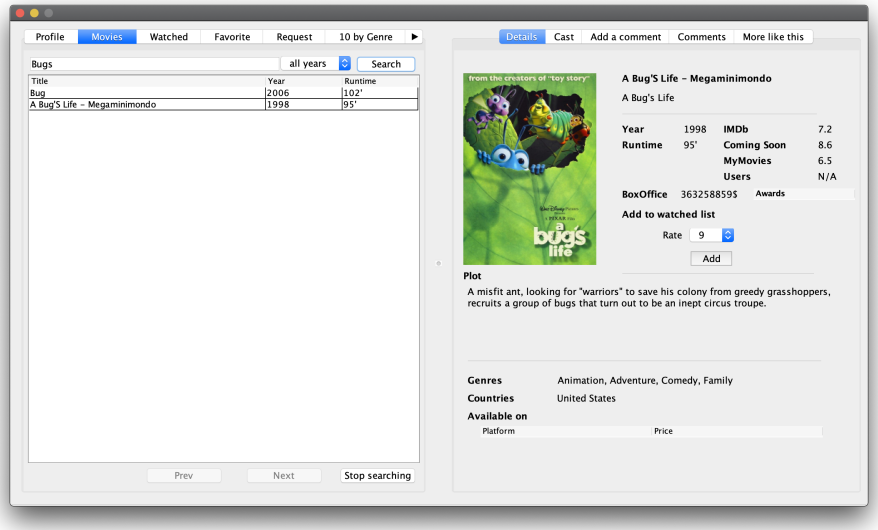

Figure 2.5

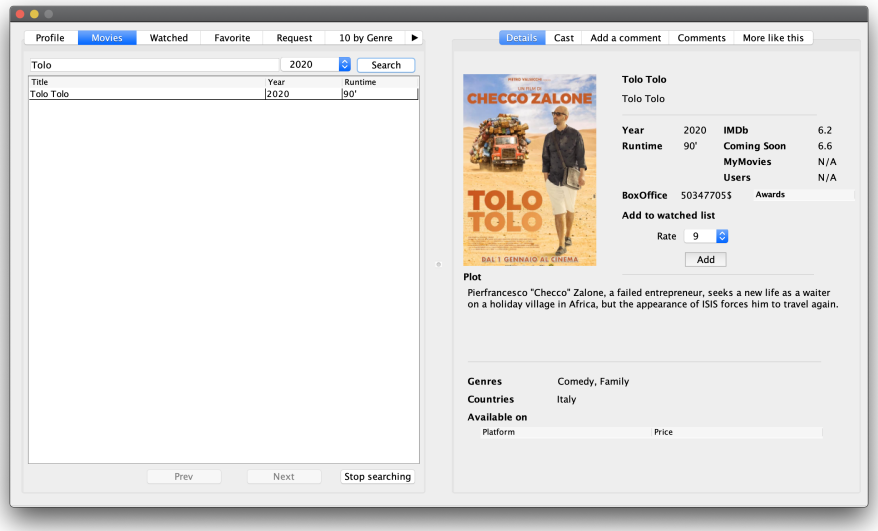

Figure 2.6

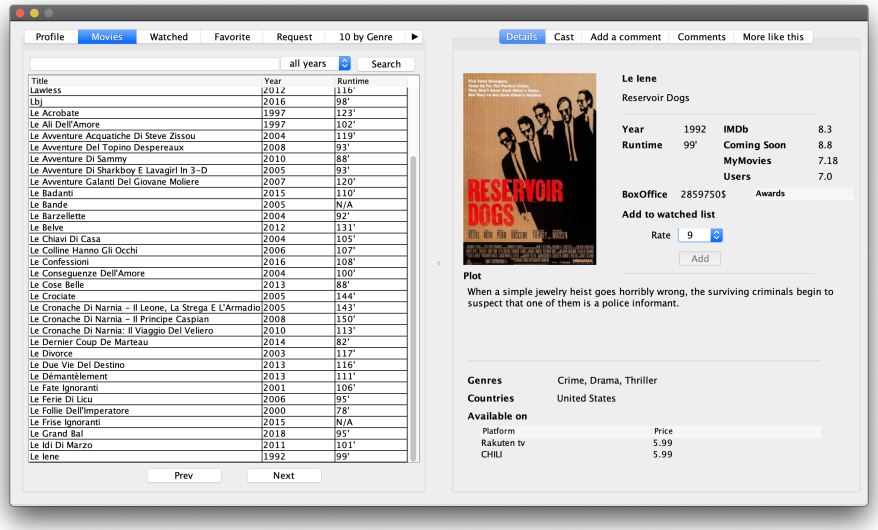

Figure 2.7

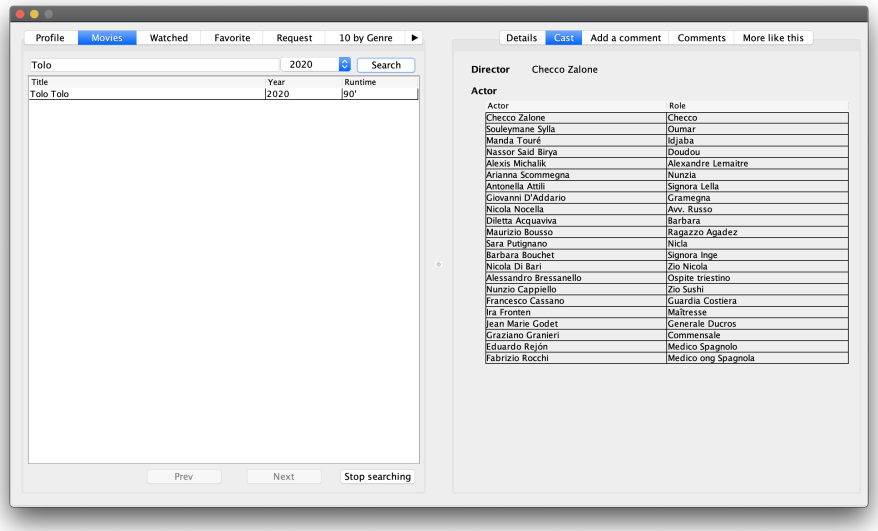

Figure 2.8

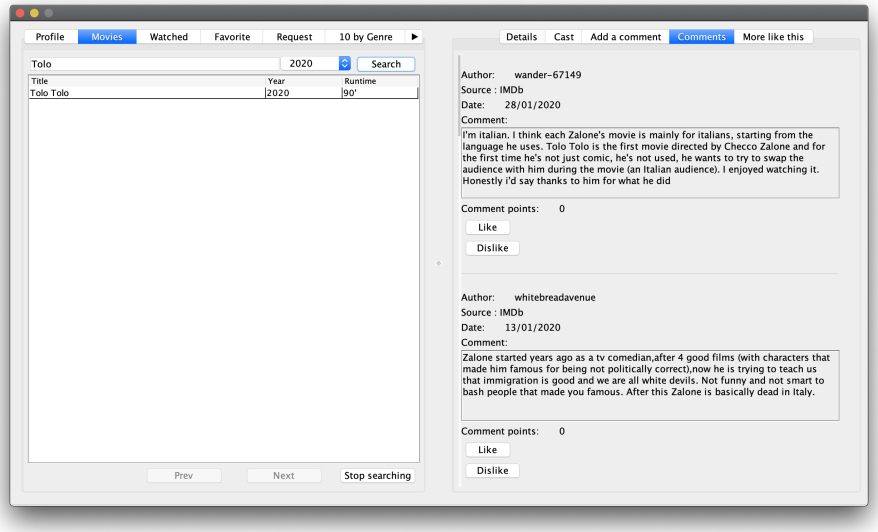

Figure 2.9

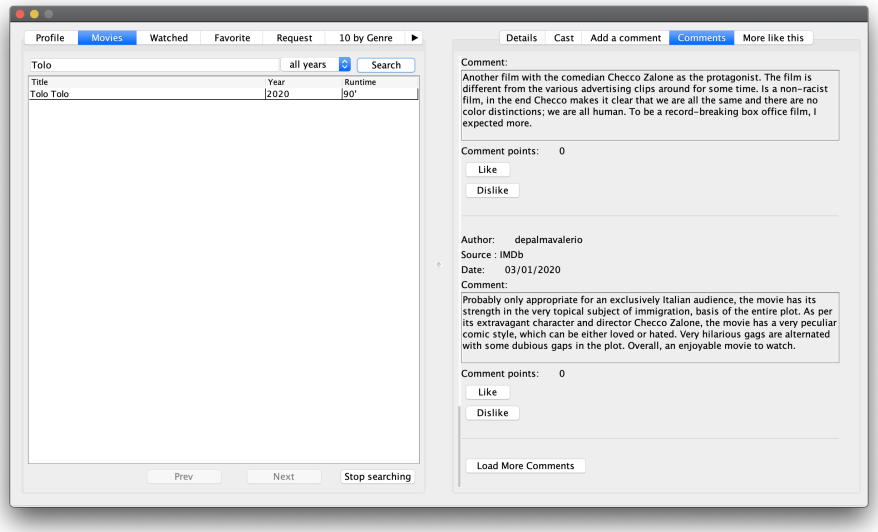

Figure 2.10

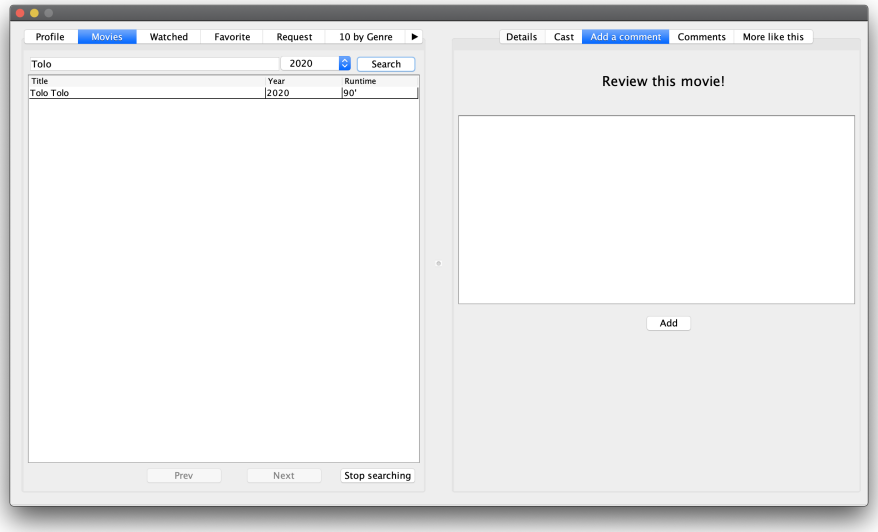

Figure 2.11

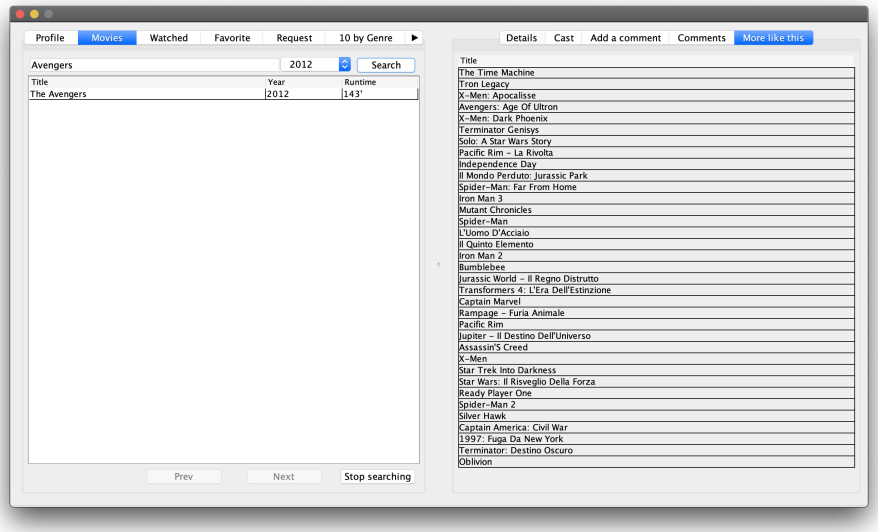

Figure 2.12

(iii) Watched panel: lets the user to browse his watched movies list and to choose which of them are his favourites ones by checking the corresponding check box. (Figure 2.13)

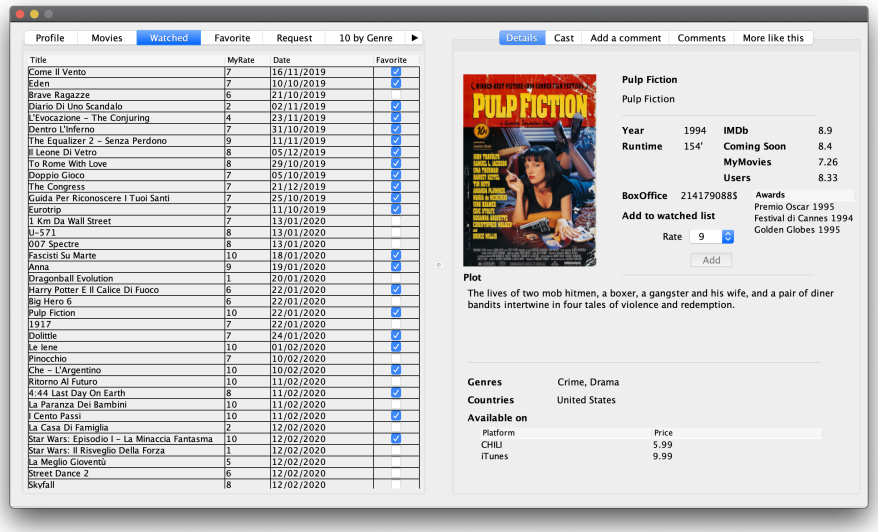

Figure 2.13

(iv) In the comments panel the user can scan the comments related to the selected film starting from those that are the most recent. It's important that just 10 comments are loaded at the beginning and whenever a user wants to visualize other 10 comments can click the appropriate button. This button will be disabled when there are no more comments. (Figure 2.9 - Figure 2.10)

(v) Favourite movies panel : in this panel the user can review the list of his favourite movies, however in order to remove a movie from his favourite list he has to return to the watched panel and uncheck this movie.

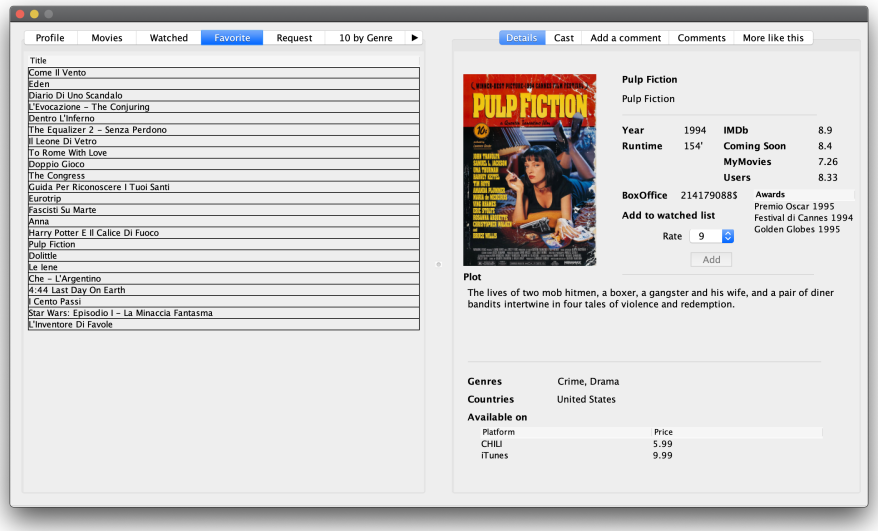

Figure 2.14

(vi) Request panel : in case the user search a specific movies and he didn't find it , he can make a add movie request to the admin by specifying the movie title in this panel and clinking on send request.

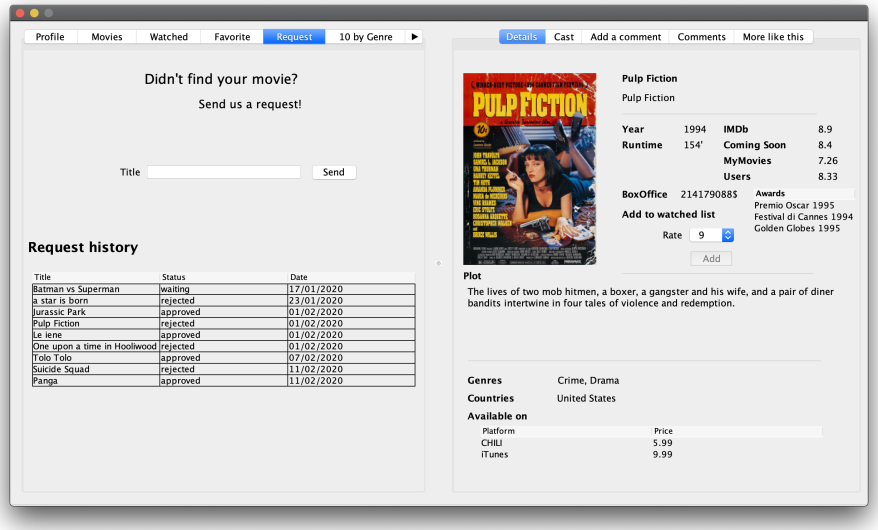

Figure 2.15

(vii) 10 by genre panel: displays the top ten movies of each movie category which are sorted by the number of views.

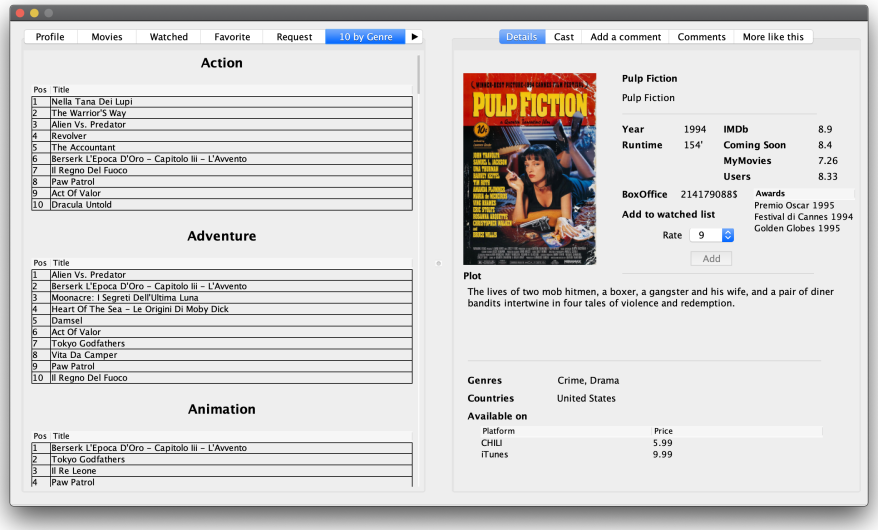

Figure 2.16

(viii) Popular panel: from this panel the user can see a list of the most watched movies in the last month.

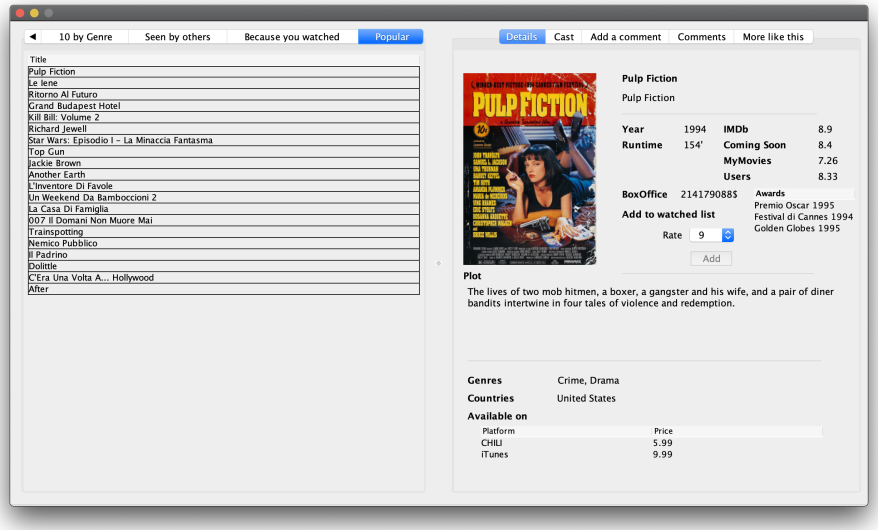

Figure 2.17

(ix) Seen by others panel: from this panel the user visualize a list of similar movie based on the most number of genres in common with the selected one.

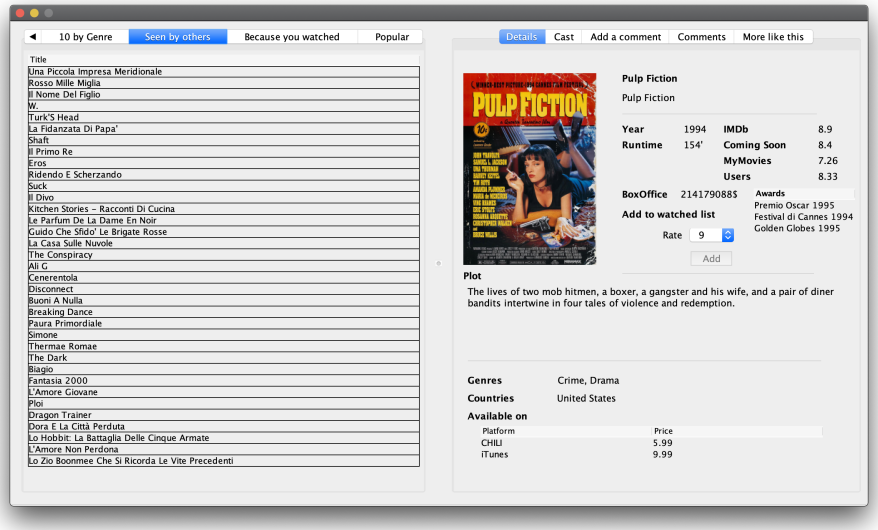

Figure 2.18

(x) Because you watched panel: in this panel it is shown a list of suggested movies oredered by the number of actors and directors in common with one or more movies watched by the users.

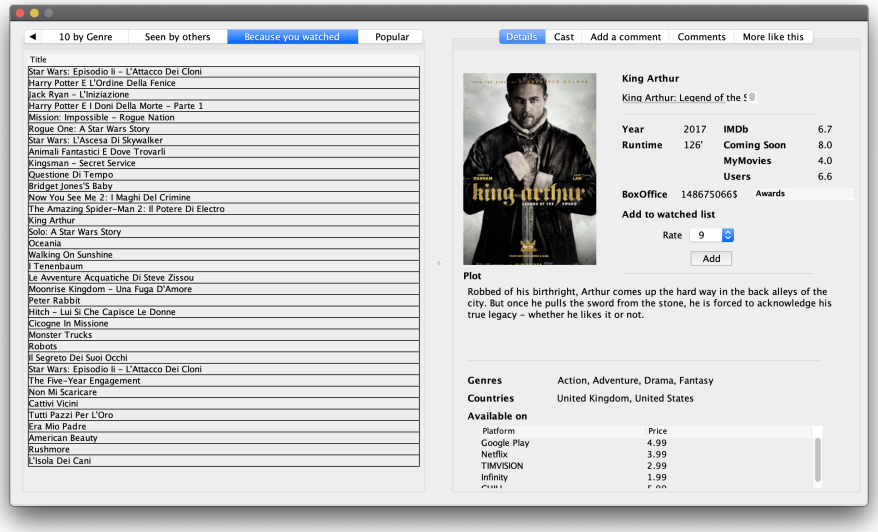

Figure 2.19

2.5.2 Admin

This window can be only accessed by an authorized administrator of the system using valid credentials at the login window. The admin window like the user window is dived into two sections (left and right), each part has a set of panels. The left panels are:

(i) Admin profile: it has the same functionalities of the user profile (display admin info and change password), however the comment section will not appear, because the admin cant comments the movies.

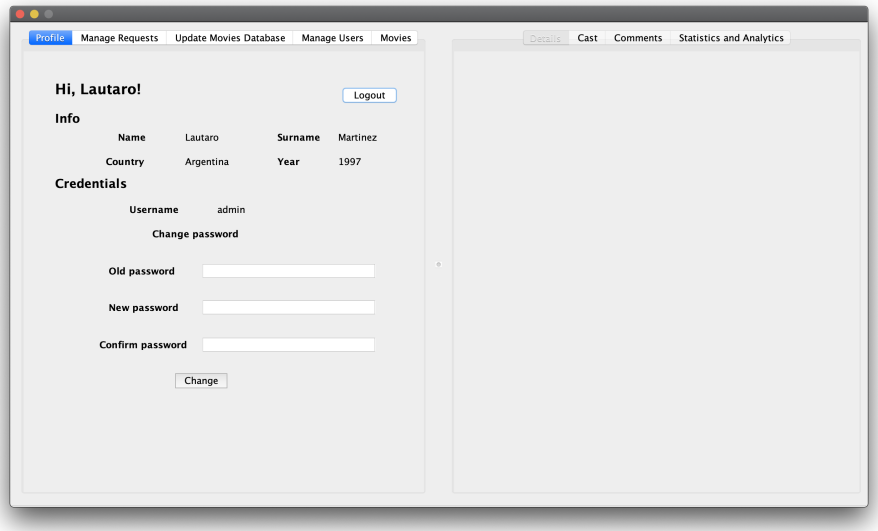

Figure 2.20

(ii) Manage requests: lets the admin to check the list of requests of adding new movies to the database with some details (movie title, date of request, username, status), moreover, if the admin clicks on one of the requests, a dialog box will box appear with two choices “approve” or “reject” the request.

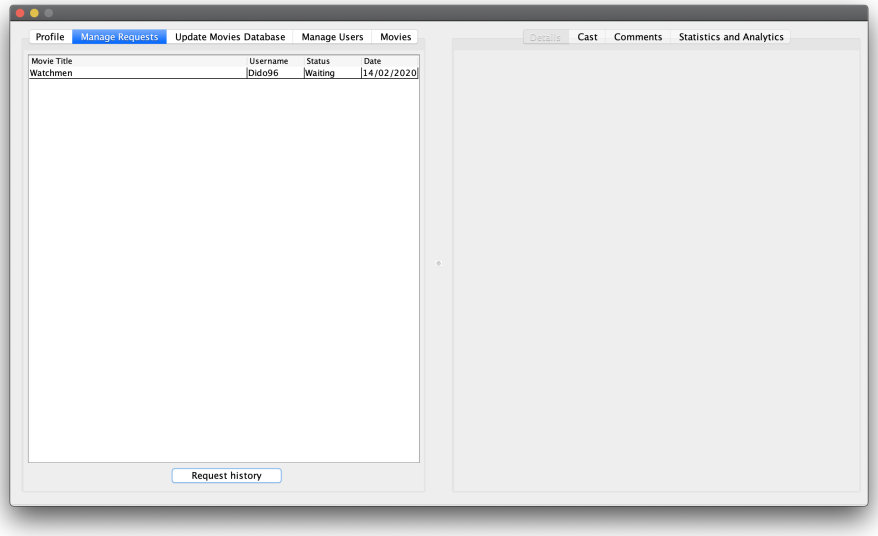

Figure 2.21

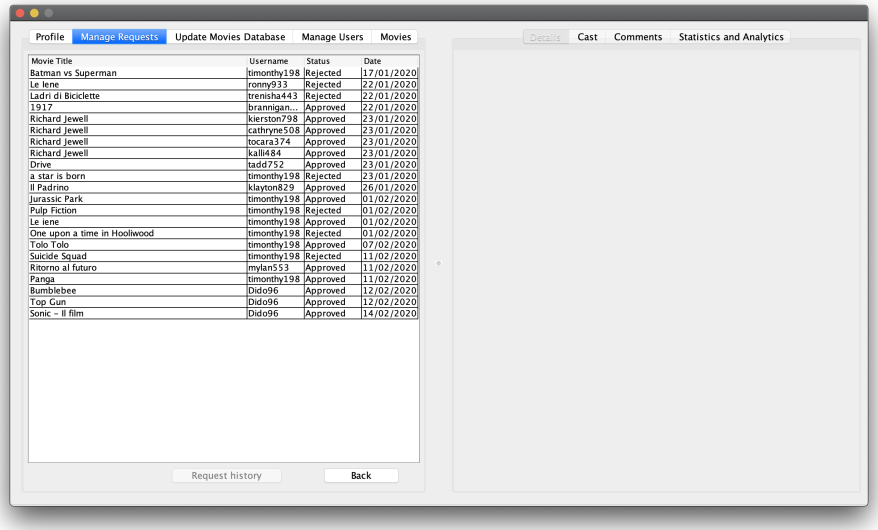

Figure 2.22

(iii) Update movies database panel: from this panel the admin can search a movie by its title and its production year, add a new one and delete movies. It’s important to note, that clicking on the button “add movie” will start a scrapping process to search the movie on web. The delete button will be enabled only if the film with those features is found and is unique. Whereas the add button is enabled only when a film with those features is not found in the database and can be added to it.

(iv) Manage users panel: lets the admin to search a user by his username in order to display his basic info and have the possibility of deleting him.

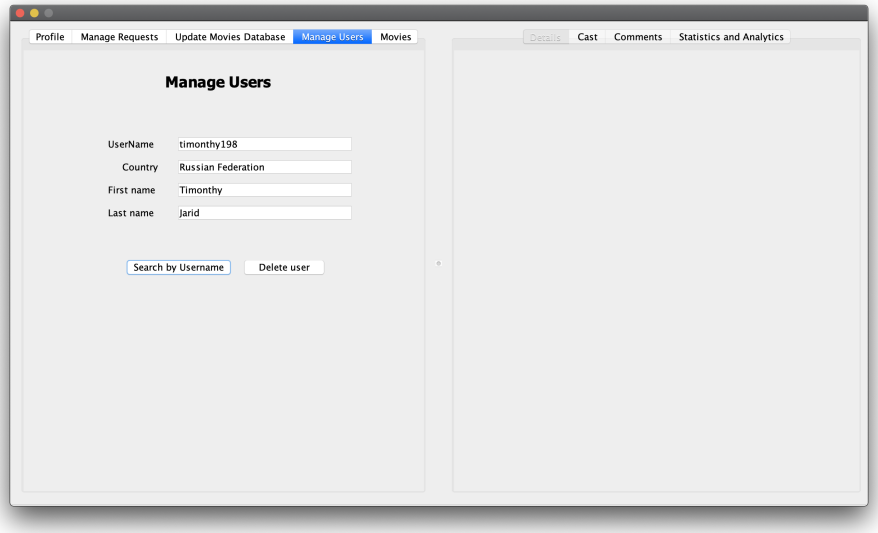

Figure 2.23

(v) Movies panel: this panel is very similar to the movies panel in the user window, with some differences such as the admin can delete the comments of all user's comments. As in the case of the user version of this panel, the click on of the movies in the list will activate and fill the right panels with the selected movies data.Note that the admin has a right panel dedicated to show the audience distribution per nation of selected movie as shown in the following figure:

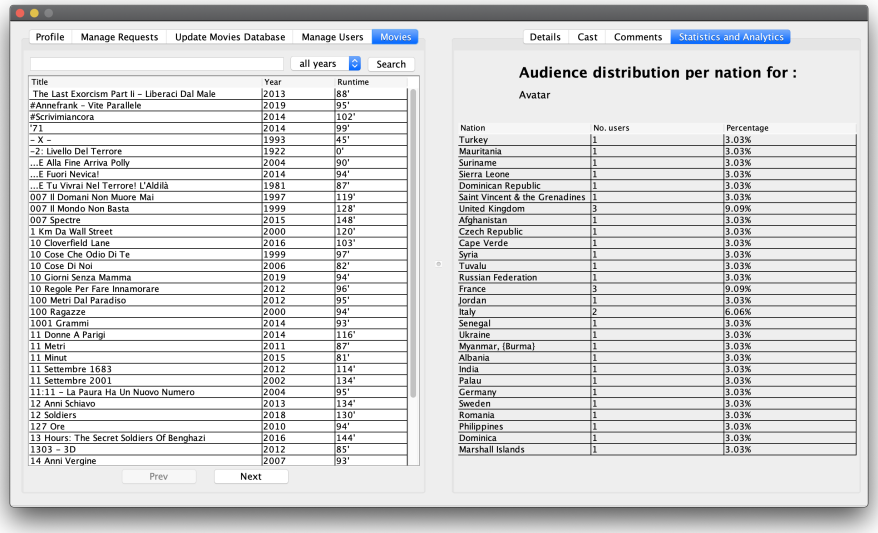

Figure 2.24

2.6 Possible future improvements of the application

The dataset of the application is divided in three collections: users, films, and comments. While both users and films collections are relatively small in size, the comments collection is the most vastly populated, with comments and reviews coming both from the users within the application itself, and by scraping movie data. Currently, there is an average of about 15 comments for every movie present in the database. With a growing number of movies, and more and more users joining the application and commenting on each movie, the comments collection could potentially grow out of control, making it really inefficient for a single machine to handle the many requests.

For all the reasons listed above, we could consider to shard the comments collection, to distribute the load over multiple machines. This would ensure not only to have more storage space available for comments, but also better response times, when the application is required to load either a user or a movie that have many different comments.

In order to maintain a load balance of the collection between the machines, we could consider using a hashed-based sharding on the _id value of each comment document. Having a good cardinality, this could be considered an ideal key to perform this method of sharding.

This enhancement would improve the overall performance of the system, and at the same time also greatly improving the horizontal scalability of the database.