PISA UNIVERSITY

TASK 2

LARGE-SCALE AND MULTI-STRUCTURED DATABASES

"PISAFLIX 2.0" PROJECT DOCUMENTATION

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

Design Document 3
Description 3
Requirements 3
Main Actors 3
Functional 3
Non-Functional 4
Analytics 4
Use Cases 5
Analytics Use Cases And Mockups 5
View Average Rating 5
View Ranking 6
View Engagement Activity 7
Analysis Classes 7
Data Model 8
Architecture 9
Interface Design Pattern 9
Software classes 10
Entities 10
DB-Manager 11
PisaFlix-Services 13
Analytics 18
Average rating 18
Ranking 19
Film engagement 21
Indexes 23
Engage Collection 23
Film Collection 25
User Collection 25
Replicas 27
Setup 27
Read From Secondaries 27
Replica Indexes 27
Hidden Server 28
Priority 28
Write Concern 28
Configuration 29
Sharding 29
User Manual 31
Registration and login 31
Browsing Film 33
Film Details 34
Browsing Users and details 36
Browsing analytics 37
Average Rating 38
Engagement activity 39
Ranking 41

DESIGN DOCUMENT

DESCRIPTION

Have you ever found yourself in a gloomy day? Everyone is at home, no one knows what to do and time seems to slow down. That's the perfect time for a movie!

PisaFlix is a platform in which users can find quality and updated information regarding **movies**. It provides a service to help you to choose what film to watch. *PisaFlix* has a **comment** section that gives at the users the possibility to create a community around their favourite movies, exchanging opinions and news regarding them. It is also possible to add films to a **favourite** list in order to find them quicker. The possibility to see other users favourites it is essential to find new friends with the same cinematic tastes. Lastly it is possible to view interesting **statistics** on films, useful both for normal users and for other people involved in the production of films.

PisaFlix offers services that will change the way users approach the world of the movie, providing them everything they need to enjoy at best their passions.

REQUIREMENTS

MAIN ACTORS

The application will interact only with the **users**, distinguished by their privilege level:

- **Normal User**: a normal user of the application with the possibility of *basic inaction*.
- **Social Moderator**: a trusted user with the possibility to *moderate* the comments.
- **Moderator**: a verified user with the possibility to add and *modify* elements in the application, like films and cinemas.
- **Admin:** an *administrator* of the application, with possibility of a *complete interaction*.

FUNCTIONAL

1. *Users* can **view** the list of **Movies** available on the platform.
2. *Users* can **view** the information about a specific *Movie*.
3. *Users* can **view** the ***statistics*** of *Film* page.
4. *Users* can **view** a set of **analytics** on Movie and another user.
5. *Users* can **register** an account on the platform.
6. *Users* can **log in** as *Normal users* on the platform in order to do some specific operations:
   a. If logged a *Normal user* can **add/remove** to **favourite** a *Movie*.
   b. If logged a *Normal user* can **comment** a *Movie*.
   c. If logged a *Normal user* can **modify** his *Comments*.
   d. A *Normal user* can **modify/delete** his account.
7. *Users* that can **log in** as *Social moderator* can do all operation of a *Normal user* plus:
   a. If logged as *Social moderator* can **delete** other users' comments.
   b. If logged as *Social moderator* can **recruit** others *Social moderator*s.
8. *Users* that can **log in** as M*oderator* can do all operation of a *Social moderator* plus:
   a. If logged a *Moderator* can **add/delete/modify** a *Movie*.
   b. If logged as *Moderator* can **recruit** other *Moderator*s

9. *Users* that can **log in** as *Admins* can do all operation of a M*oderator* plus:
   a. If logged an *Admin* can **delete** another user's account.
   b. If logged as *Admin* can **recruit** other *Admin*s.

NON-FUNCTIONAL

1. The application's focus is the *quality* of the information provided to users.
2. The application needs to be **consistent**, in order to provide correct information to all the users.
3. The application needs to be **tolerant to partitions**, in order to scale the system if needed, preserving the consistency.
4. The application needs to store **replicas** of the data in case of server fault, all the replicas need to be **consistent**.
5. The transactions must be **monotonic:** every user must see the last version of the data and modifications are done in the same order that are committed.
6. The application needs to be *usable* and *enjoyable* for the user, therefore the system needs **limited response times**.
7. The *password* must be protected and stored *encrypted* for privacy issues.

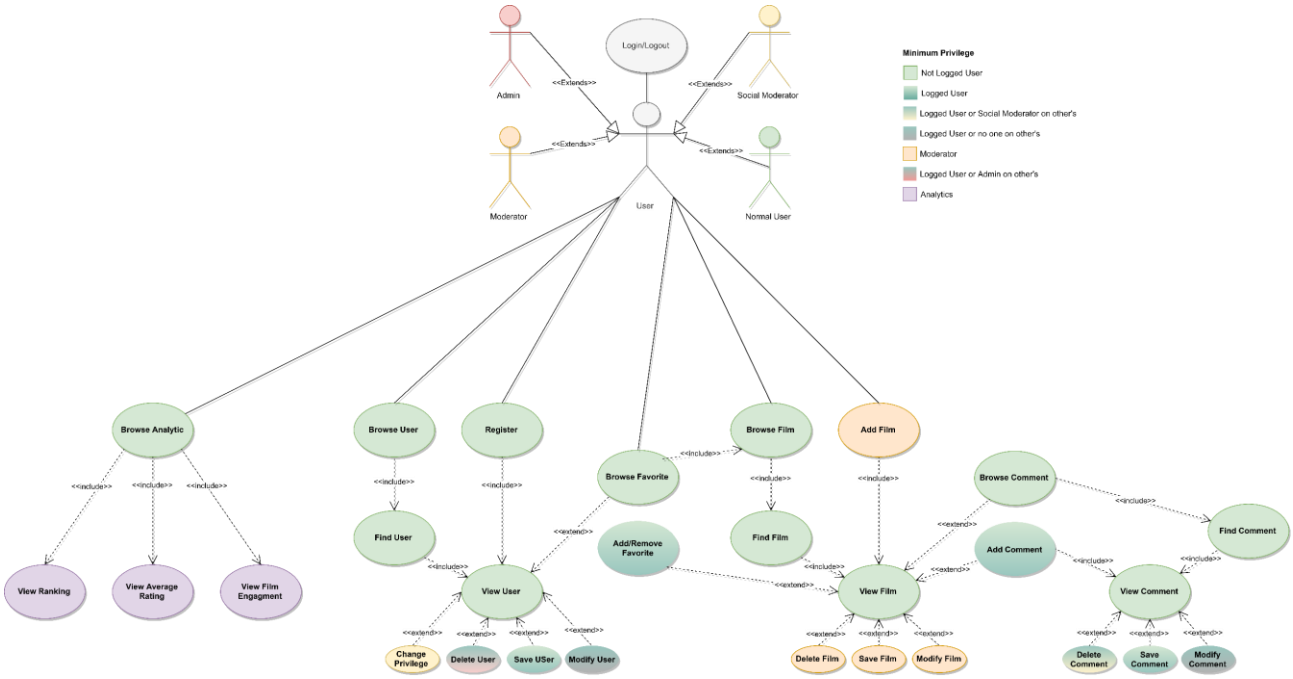
ANALYTICS

In *PisaFlix* there are three main types of **analytics**:

- **Average Rating**: Chosen a *temporal interval*, the application shows the mean value of movie **rating** *grouped by* genre, director or actor.
- **Ranking**: Chosen a *temporal interval*, the application shows either a table with the most **involved** *user* or the most **engaged** *film* of the platform.
  This is done by calculating a weighted sum: $I = (3 \cdot c) + (2 \cdot f) + (1 \cdot v)$.
  - If we are considering **user** activities: *c* stands for the number of *comments* that the user has done, *f* for the number of his *favourites*, and *v* for the number of film page *viewed* by the user.
  - If, instead, we are considering **film** engages: *c* stands for the number of *comments* received by the film, *f* for the number of *favourites*, and *v* for the number of *visits* on the film page.

  The above value is calculated for all films/users and it is used to **rank** them.
- **Film Engagement**: Chosen a *film* and two years (one for the *beginning* and one for the *end* of a period), the application shows the **engagement** of the chosen film *by year*, in the interval of time specified before. Moreover, the application will show also the **composition** of the engagement for a chosen year.
  This is the formula for the engagement of a film $E = (3 \cdot c) + (2 \cdot f) + (1 \cdot v)$. It is a weighted sum where *E* is the *total engagement*, *f* is the *number of favourites* received by the film, *c* is the *number of comments* on that film, and *v* is the number of times that the page of the film has been *visited*.

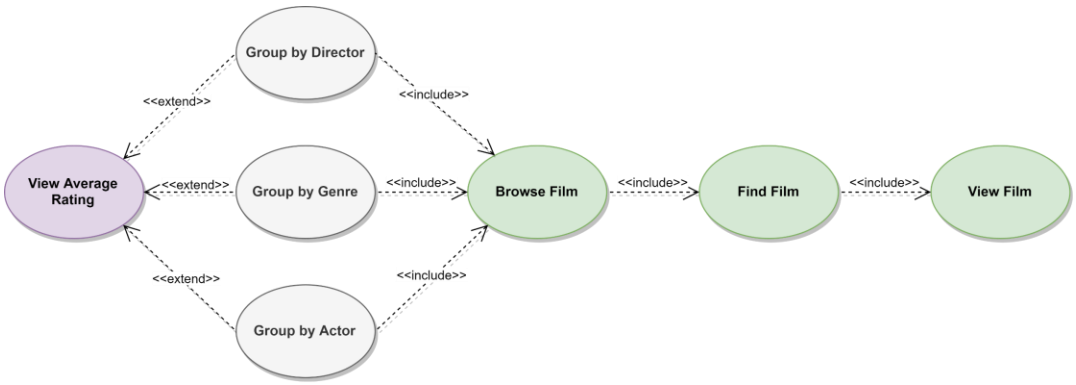
USE CASES



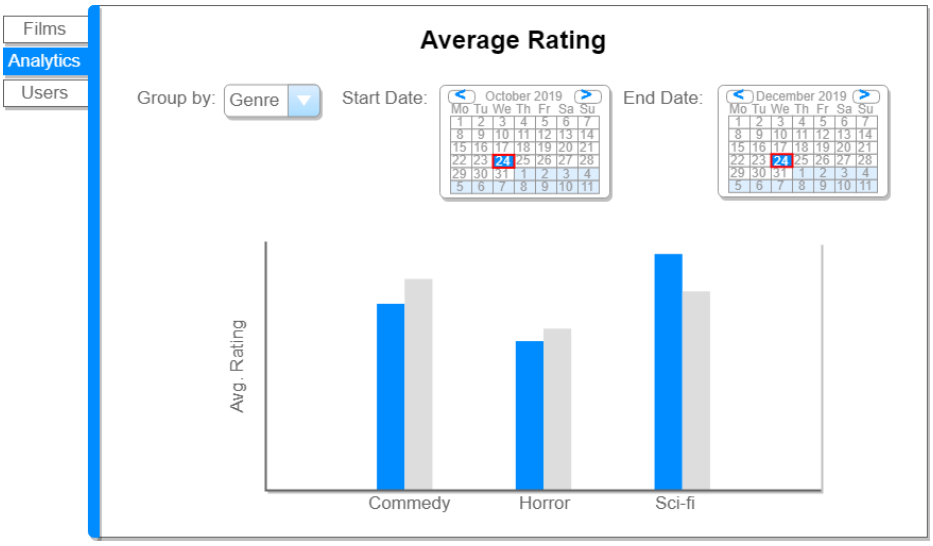
ANALYTICS USE CASES AND MOCKUPS

The use cases of the analytical functions are shown below.

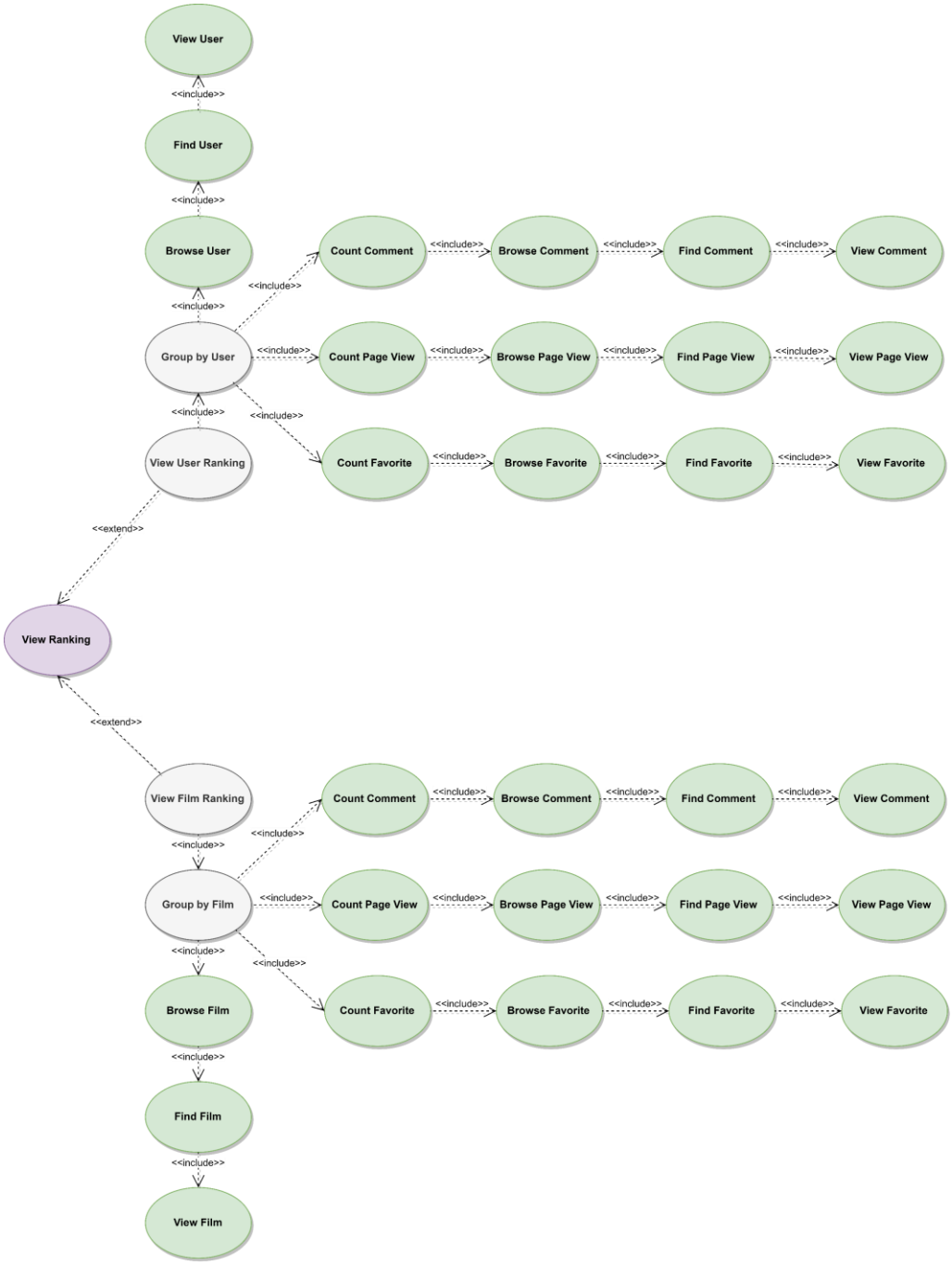
VIEW AVERAGE RATING



MOCKUP



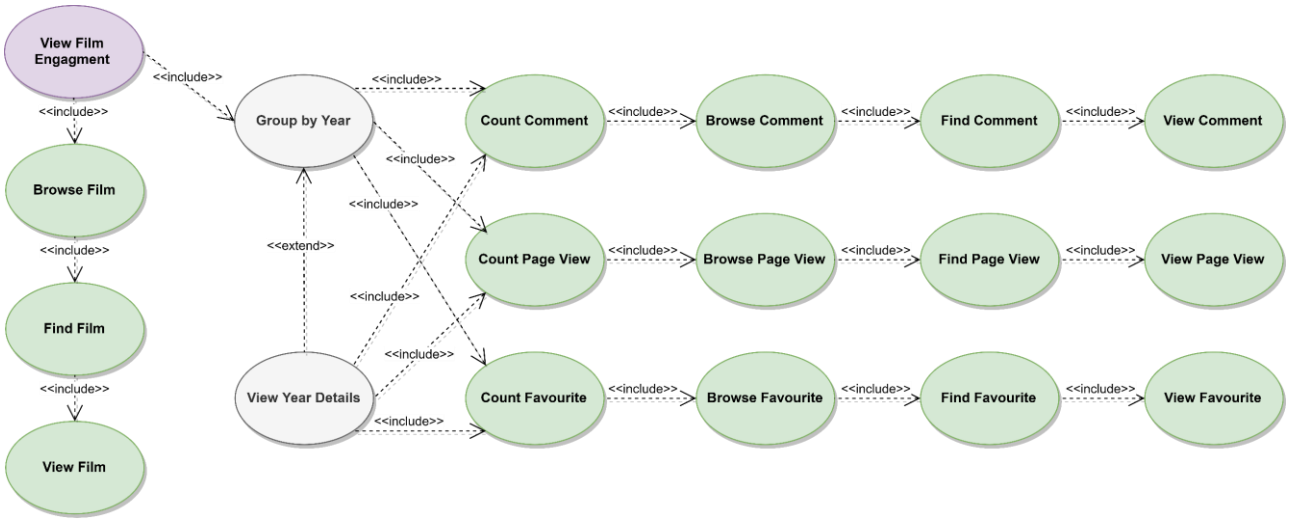
VIEW RANKING



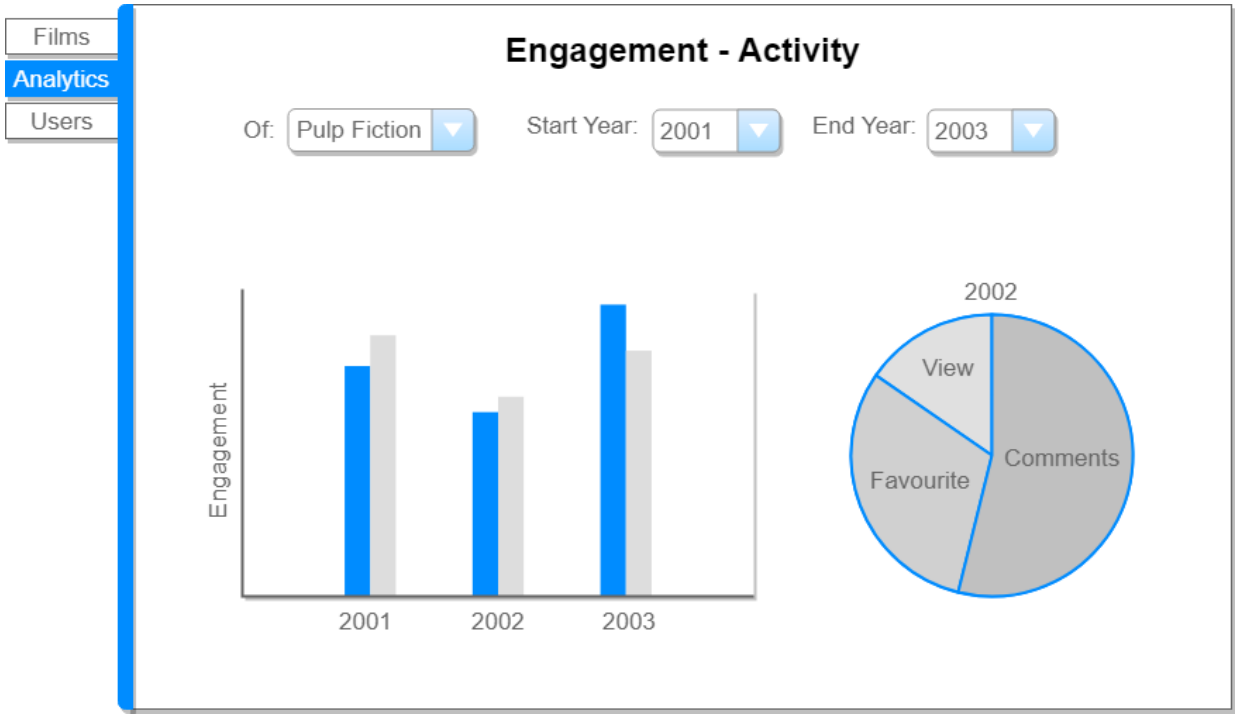
MOCKUP

Rank	Film	Score
1°	2001: A Space Odyssey	12327
2°	Pulp Fiction	7000
3°	Albakiara	3789
4°	Joker	2019
5°	A Pigeon Sat on a Branch Reflecting on Existence	1999
6°	A Clockwork Orange	1070
7°	The Great Beauty	897

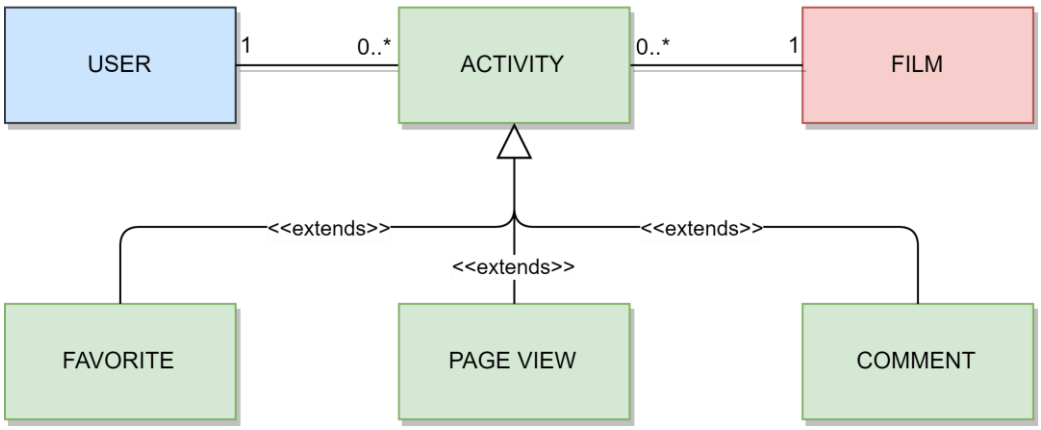
VIEW ENGAGEMENT ACTIVITY



MOCKUP

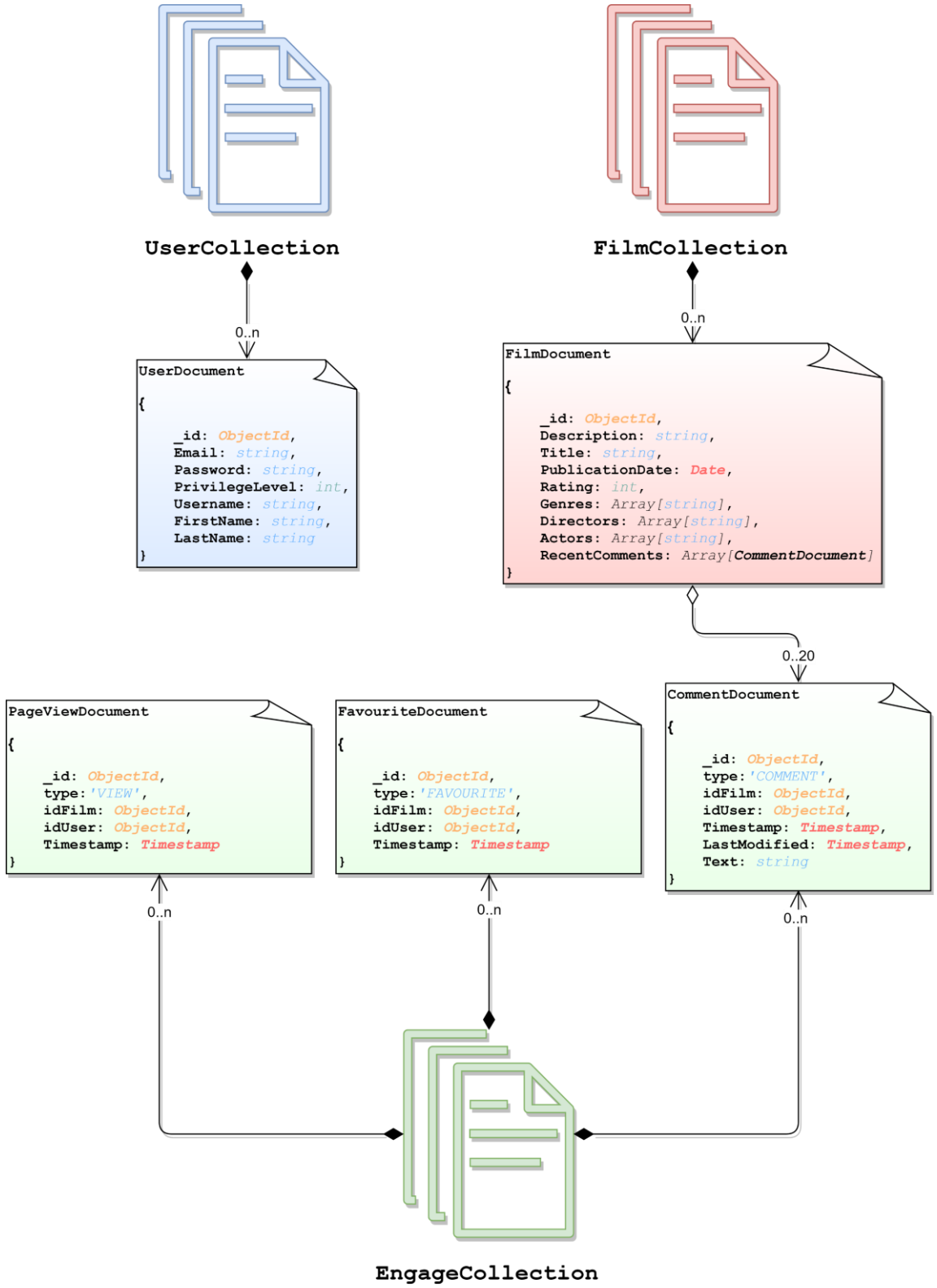


ANALYSIS CLASSES



DATA MODEL

Below is shown the schema of the documents and collections of the database:



The **favorite**, **view** and **comment** documents have been included in the same collection as they represent a relationship between a user and a film, with a timestamp of the time when it took place, only comments can also contain additional fields.

To improve performance in viewing movie pages, the twenty most **recent comments** are embedded within movie documents, so it's possible to retrieve them in one step.

The **favorites**, unlike the recent comments for the films, are generally limited in number for users and must always be shown together, as they have the purpose of shortcuts towards the favourite films. They have not been included embedded in the documents of the users also because all queries involving finding out how many times movies have been preferred in a time frame would always scroll through all users.

ARCHITECTURE

Users can use a java application with a **GUI** to take advantage of all the functionalities of the platform.

The client Application it's made in *Java* using **JavaFX framework** for the *front-end* and the **MongoDB driver** to manage *back-end* functionalities. **Services** and ***JavaBean* objects** compose the *middleware* infrastructure that connect *front-end* and *back-end.*

INTERFACE DESIGN PATTERN

The graphic user interface was build following the software design pattern of **Model-View-Controller**.

MODEL

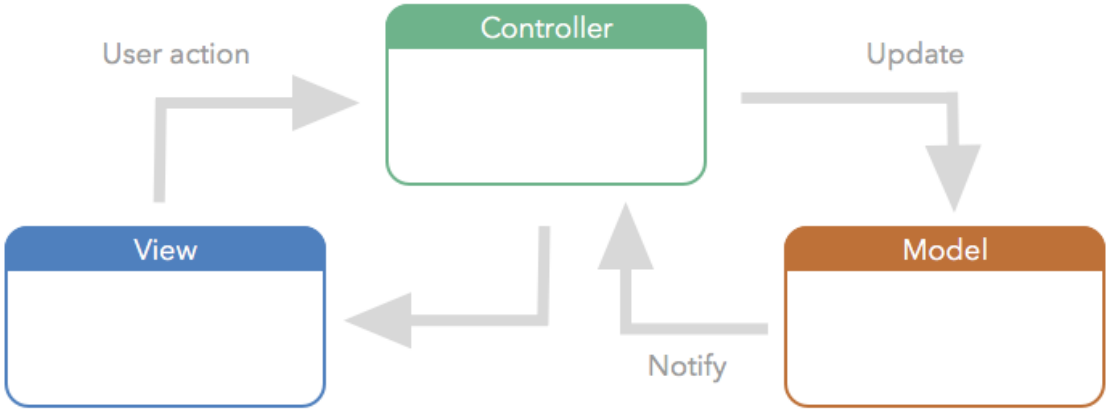
Services module represent the *model* and is the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages logic and rules of the application receiving inputs from the controller. The model is also responsible for managing the application's data in form of JavaBean objects, exchanging them with the controller.

VIEW

The **fxml files** represents the *view* and are responsible for all the components visible in the user's interface.

CONTROLLER

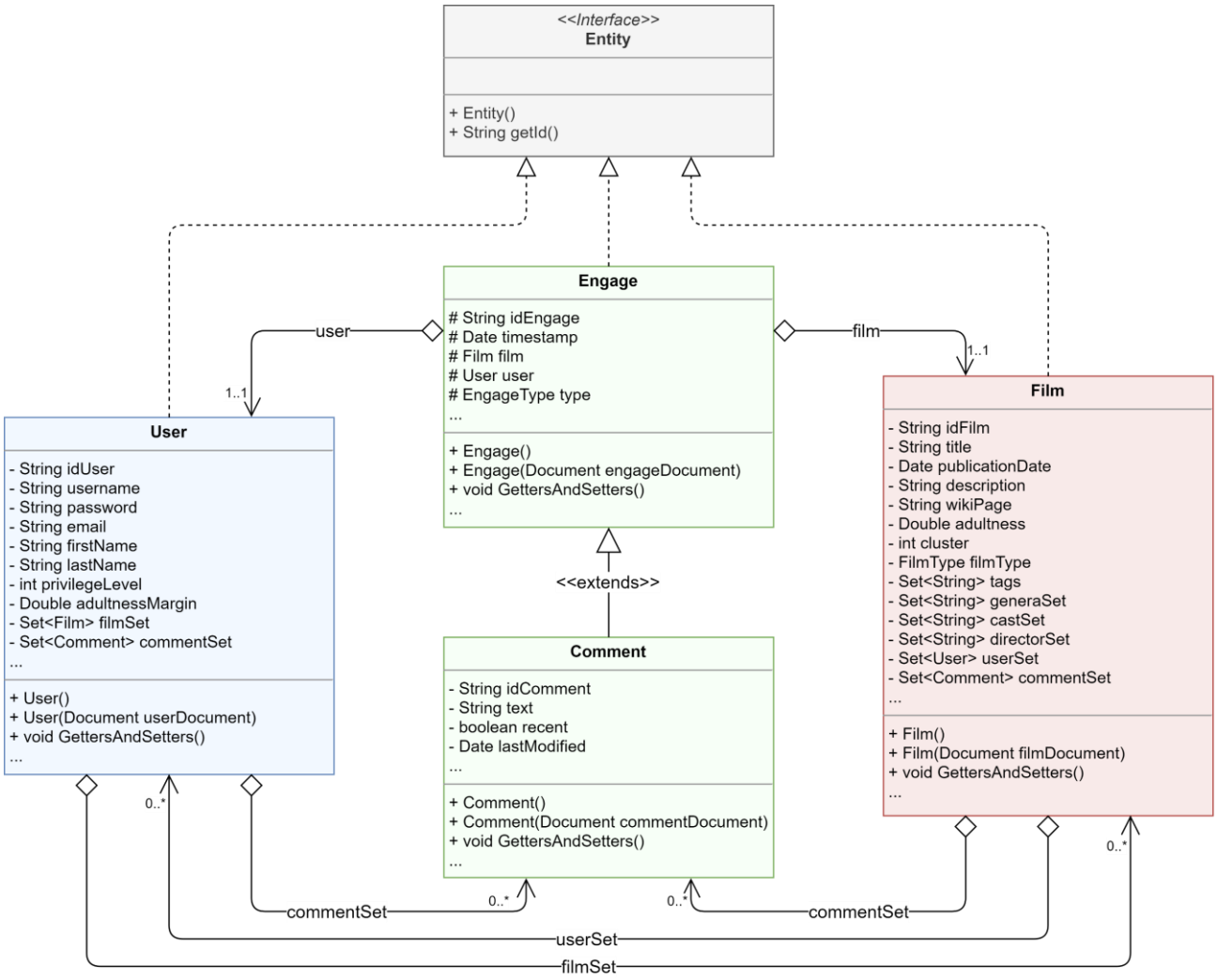
The **page controllers** are the *controller* of the application. They receive inputs from the *view* and converts them into commands for the *model* or *view* itself. Controllers can also validate inputs and data without the intervention of the *model*. Data is exchanged between *model* and *controller* using JavaBean objects.



SOFTWARE CLASSES

ENTITIES

Diagram of the **classes**:



ENTITY

Represent the interface of each entity.

USER

This entity class represents any **user**, in addition to the **personal information** necessary for their display on the application, the user's **privilege level** is present to allow him to perform only the actions allowed by it.

The getters and the class constructor are the only functions present; the various fields can be instantiated using directly a document with the various information.

COMMENT

This entity class represents any **comment**, extends ***Engage*** allowing to save additional information present only in comments.

The getters and the class constructor are the only functions present; the various fields can be instantiated using directly a document with the various information.

ENGAGE

This entity class fully represents **page-views** and **favourites**, is extended by the ***Comment*** class to inherit its common fields.

The getters and the class constructor are the only functions present; the various fields can be instantiated using directly a document with the various information.

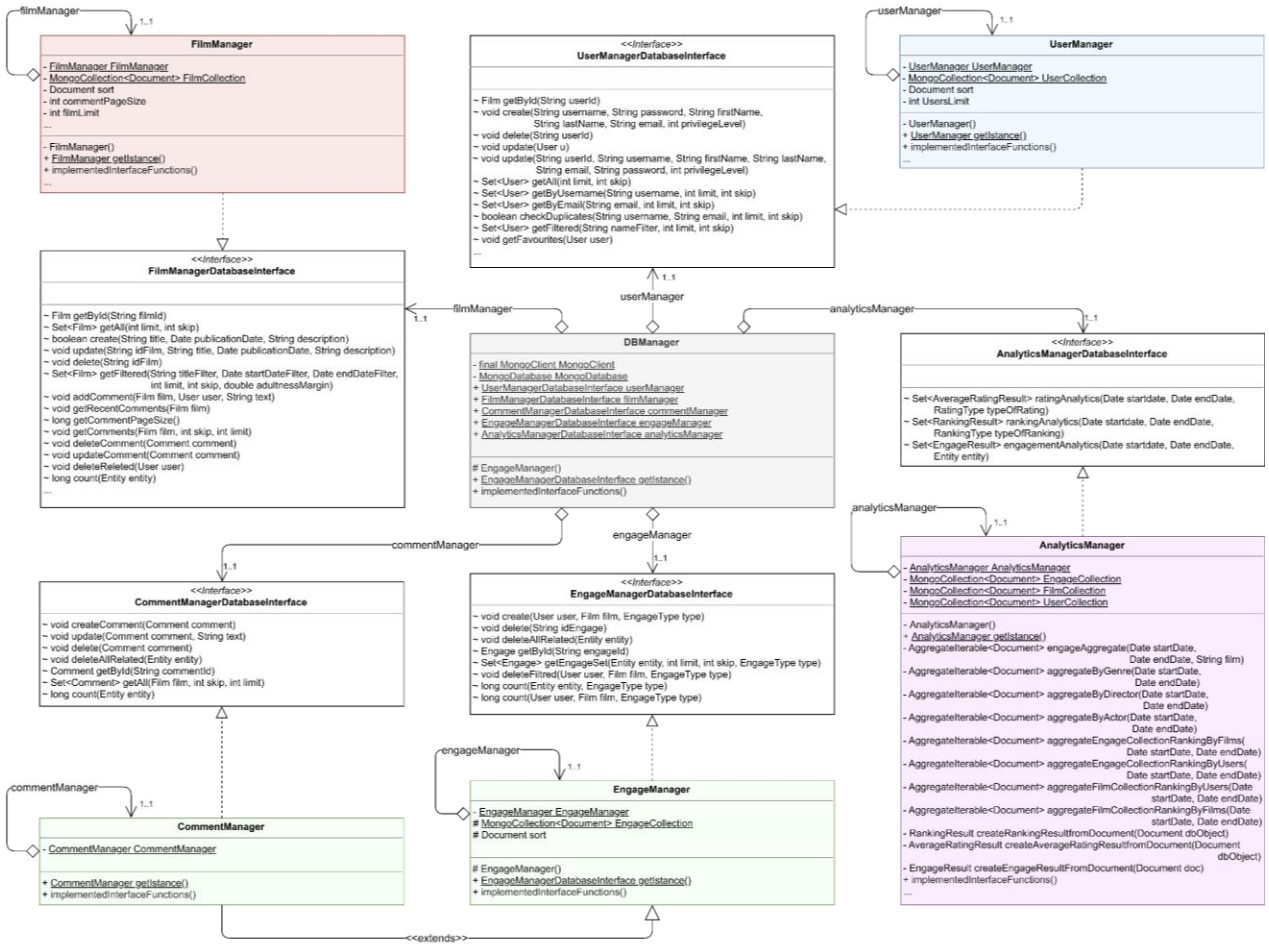
FILM

This entity class represents any **film** and its information.

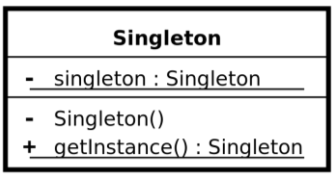
The getters and the class constructor are the only functions present; the various fields can be instantiated using directly a document with the various information.

DB-MANAGER

The structure of **DBManager**:



All the managers are implemented following the software design pattern of ***singleton* pattern** which restricts the instantiation of a manager to *one* instance.



The main classes and functions are described below:

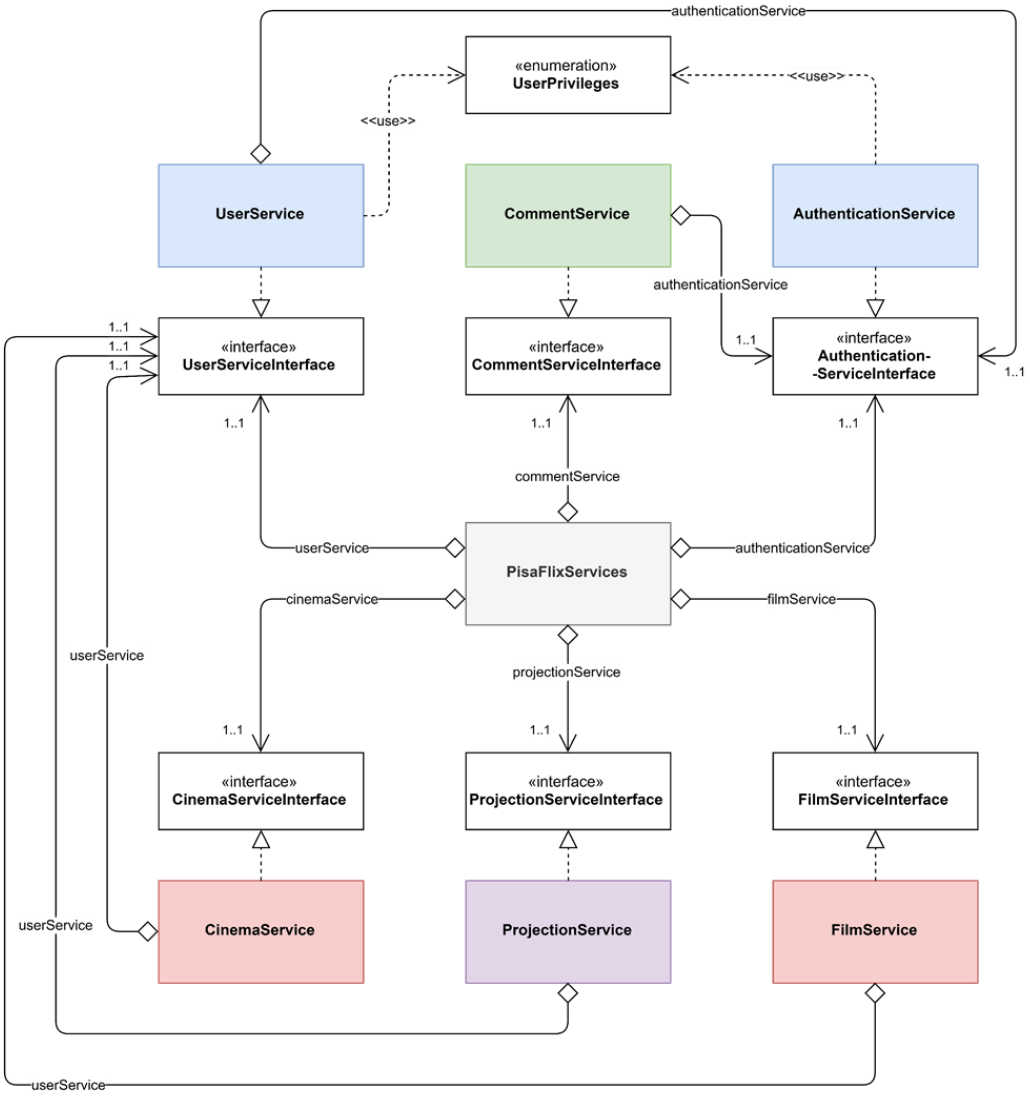
- **DBManager** is an utility class, it's a static class that contains all the other manager specific to certain operations, the other managers are accessible through the public members of the class, it automatically *initialize* all the managers on first call and the method *DBManager.Stop()* must be called at the end of the application in order to close the connection with *MongoDB* servers.
- **UserManagerDatabaseInterface** it's the interface which defines the basic operation that any user manager should have (independent from the technology)
- **UserManager** implements *UserManagerDatabaseInterface* and is in charge of manage all *CRUD* operation with the database for the users.
  The function gets an extra two parameter, that are two integers: limit and skip. These two integers are used to get a pagination, retrieving always "limit" document, and then skipping "skip" document for the next page.
  All functions are self-explanatory by the name except for:
  - **getFiltered**(String user*nameFilter, int limit, int skip*) which search and returns all users who have "user*nameFilter*" in the username, if *usernameFilter* is not set the filter it's not taken into consideration and returns all users. Limit and skip are used for the purpose described before.
- **FilmManagerDatabaseInterface** it's the interface which defines the basic operation that any film manager should have (independent from the technology)
- **FilmManager** implements *FilmManagerDatabaseInterface* and is in charge of manage all *CRUD* operation with the database for the movies.
  Some functions take two additional parameters, limit and skip, for the same reason of UserManager.
  All functions are self-explanatory by the name except for:
  - **getFiltered**(String *titleFilter*, Date *startDateFilter*, Date *endDateFilter, int limit, int skip*) which search and returns all movies which have "*titleFilter*" in the title and the pubblicationDate it's between "*startDateFilter*" and "*endDateFilter*", if some filter is not set the filter it's not taken into consideration, if all filter are not set it returns all movies.
- **EngageManagerDatabaseInterface** it's the interface which defines the basic operation that any engage manager should have (independent from the technology)
- **EngageManager** implements *EngageManagerDatabaseInterface* and is in charge of manage all CRUD *operation* with the database for the engages, that are all possible operations for *favourites* and *views*, but the *CommentManager* extends it to add more.
  Below are the descriptions of the main functions:
  - **create(**User user, Film film, EngageType type**)** which create an engage, that can be of type, View or Favourite (Engages can have a third type that is Comment, but that case is managed by the CommentManager)
  - **deleteAllRelated(**Entity entity**)** which delete all the engages, related to an entity e.g. delete all the views, favourites and comments, done by a User. In this case the function managed too the comments.
- **CommentManagerDatabaseInterface** it's the interface which defines the basic operation that any comment manager should have (independent from the technology)

- **CommentManager** implements *CommentManagerDatabaseInterface* and extends *EngageManager* and is in charge of manage all *CRUD* operation with the database for the comment.
- **AnalyticsManagerDatabaseInterface** it's the interface which defines the basic operation that any analytic manager should have (independent from the technology)
- **AnalyticsManager** implements *AnalyticsManagerDatabaseInterface* and is in charge to perform the analytics described before in the documentation.
  The function that perform the three analytics are:
  - **ratingAnalytics(**Date startDate, Date endDate, RatingType typeOfRating**)** which perform the first analytics.
  - **engagementAnalytics(**Date startDate, Date endDate, Entity entity**)** which perform the seconds.
  - **rankingAnalytics(**Date startDate, Date endDate, RankingType typeOfRanking**)** which perform the third.

  All the other function are for support of these three functions.

PISAFLIX-SERVICES

Due to its complexity, a schematic diagram of the services offered by the application is provided below:



The *PisaFlixServices* follows the same structure of DBManager, all single services follow the singleton software design pattern explained before

- **PisaFlixServices** is a utility class, it's a static class that contains all the other manager specific to certain operations, the other services are accessible through the public members of the class, it automatically initializes all the services on first call.
- **UserPrivileges** it's an enumeration class which maps the user privileges
  - NORMAL_USER -> level 0 of DB
  - SOCIAL_MODERATOR -> level 1 of DB
  - MODERATOR -> level 2 of DB
  - ADMIN -> level 3 of DB
- **AuthenticationServiceInterface** it's the interface which defines the basic operation that any authentication service should have (independent from the technology)
  - we will see the methods in detail in the class which implement it
- **AuthenticationService** implements *AuthenticationServiceInterface* and is in charge of managing the authentication procedure of the application, it uses *UserManagerDatabaseInterface* in order to operate with database and obtain data
  - void **login**(String *username*, String *password*) if called with valid credentials it makes the log in and saves the users information in a local variable opening a kind of session, it may throw *UserAlredyLoggedException* if called with an already open session or *InvalidCredentialsException* if called with invalid credentials
  - void **logout**() it closes the session deleting user information stored in the local variable
  - boolean **isUserLogged**() it checks if the user is logged and gives back the result
  - String **getInfoString**() it provides some text information of the current session (ex. "logged as Example"
  - User **getLoggedUser**() get the information of the logged user
  - void **checkUserPrivilegesForOperation**(UserPrivileges *privilegesToAchieve*, String *operation*) checks if the logged user has the right privileges in order to do an operation, it does do nothing if he has them, otherwise it throws *InvalidPrivilegeLevelException*, it may also throw *UserNotLoggedException* if called without an active session, the field operation it used just to print the operation that we would like to perform in the error message.
  - void **checkUserPrivilegesForOperation**(UserPrivileges *privilegesToAchieve*) it just call **checkUserPrivilegesForOperation**(UserPrivileges *privilegesToAchieve*, String *operation*) with a default text for the "*operation*" field
- **UserServiceInterface** it's the interface which defines the basic operation that any user service should have (independent from the technology)
  - we will see the methods in detail in the class which implement it
- **UserService** implements *UserServiceInterface* and oversees all the operations that are specific for users, in order to work properly it use an *UserManagerDatabaseInterface* to exchange data with the DB and an *AuthenticationServiceInterface* for ensure a correct session status depending by the operation we want to perform
  - Set<User> **getAll**() returns all the users in the DB
  - User **getById**(String *id*) returns a specific user identify by its "*id*"

- Set<User> **getFiltered**(String *nameFilter*) search and returns all users who have "*nameFilter*" in the username, if *nameFilter* is not set the filter it's not taken into consideration and returns all users.
- void **updateUser**(User *user*) updates a user in the database with new information specify by its parameter
- void **register**(String *username*, String *password*, String *email*, String *firstName*, String *lastName*) it register a new user in the database, if some field It's not valid it throws *InvalidFieldException* specify also the reason why it was thrown
- void **changeUserPrivileges**(User *u*, UserPrivileges *newPrivilegeLevel*) allows the logged user to change the privileges of a user (it can also be itself) it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't change the privileges of the target user;
- void **deleteUserAccount**(User *u*) allows the logged user to delete a user (it can also be itself) it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't delete the target user;
- void **deleteLoggedAccount**() it just call **deleteUserAccount**(User *u*) with the user logged as parameter.
- Void getFavourites(User user) it calls the **getFavourites**(user) of the user manager which will update the favourite film set in the user object

- **FilmServiceInterface** it's the interface which defines the basic operation that any film service should have (independent from the technology)
  - we will see the methods in detail in the class which implement it
- **FilmService** implements *FilmServiceInterface* and is in charge of manage all operations that are specific for films, in order to work properly it uses *FilmManagerDatabaseInterface* to exchange data with the DB and *AuthenticationServiceInterface* to ensure that we have the right privileges depending by the operation that we want to perform
  - Set<Film> **getFilmsFiltered**(String *titleFilter*, Date *startDateFilter*, Date *endDateFilter*) search in the DB and returns all movies which have "*titleFilter*" in the title and the pubblicationDate it's between "*startDateFilter"* and "*endDateFilter*", if some filter is not set the filter it's not taken into consideration, if all filter are not set it returns all movies.
  - Set<Film> **getAll**() returns all movies int the DB
  - Film **getById**(int *id*) returns a specific film identify by its "*id*"
  - void **addFilm**(String *title*, Date *publicationDate*, String *description*) allows to insert a new film in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't add a new film
  - void **updateFilm**(Film *film*) allows to modify a film in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't modify a film
  - void **deleteFilm**(String *idFilm*) allows to delete a film in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't delete a film
  - void **getRecentComments**(Film film) it calls the same method of the filmManager so that the Set of recent comments can be updated

    - void **addComment**(Film film, User user, String text) it calls the same method of the filmManager so that it can add a comment associated to the film passed as an argument.
    - long **getCommentPageSize**() it calls the same method of the filmManager so that it can return the number of comments to be displayed in one page.
    - void **getCommentPage**(Film film, int page) it calls the same method of the filmManager so that it can update the set of comments in the object "film". The page integer is used to calculate how many comments to skip in the query to the DB. This number obviously depends on the number of comments per page returned by **getCommentPageSize**()
- **CommentServiceInterface** it's the interface which defines the basic operation that any comment service should have (independent from the technology)
    - we will see the methods in detail in the class which implement it
- **CommentService** implements *CommentServiceInterface* and is in charge of manage all operations that are specific for comments, in order to work properly it use an *CommentManagerDatabaseInterface* to exchange data with the DB, an *AuthenticationServiceInterface* in order to retrieve the current logged user and to ensure that we have the right privileges depending by the operation that we want perform
    - Comment **getById**(int *id*) returns a specific film identify by its "*id*"
    - void **addComment**(String *text*, User *user*, Film *film*) creates a new comment for a "*film*" made by a certain "*user*" and saves it in the *EngageCollection* (Note: this method should be used by the FilmManager if the number of nested comments exceed the limit imposed)
    - void **update**(Comment *comment*) allows to modify a comment in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't modify the comment
    - void **delete**(Comment *comment*) allows to delete a comment in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't delete the comment
    - long **count**(Entity user) it returns the number of comments written by the user passed as an argument.
- **EngageServiceInterface** it's the interface which defines the basic operation that any engage service should have (independent from the technology)
    - we will see the methods in detail in the class which implement it
- **EngageService** implements *EngageServiceInterface* and is in charge of manage all operations that are specific for the engagement activities, in order to work properly it uses a *EngageManagerDatabaseInterface* to exchange data with the DB, an *AuthenticationServiceInterface* in order to retrieve the current logged user and to ensure that we have the right privileges depending by the operation that we want perform
    - void **create**(User *user*, Film *film*, EngageType *type*) it calls a method of the *EngageManager* that saves a document in the *EngageCollection* for an engagement activity done by the "user" on the "film".
    - Engage **getById**(String *engageId*) it returns the engagement object associated to the "engageId" if present, null otherwise.

    - Set<Engage> **getEngageSet**(Entity *entity*, int *limit*, int *skip*, EngageType *type*) it returns a set of Engage entity associated to "entity" of the given "type" (Comment, Favourite, View), with the possibility to specify a limit and a skip.
    - void **deleteFiltred**(User *user*, Film *film*, EngageType *type*) it calls the same method in the *EngageManager* which deletes a document that matches the filters
    - long **count**(Entity *entity*, EngageType *type*) it returns the number of documents in the EngageCollection that matches the type and the entity specified
    - boolean **isAlreadyPresent**(User *userLogged*, Film *film*, EngageType *entityType*) it checks if an equal engage document is already been saved
- **AnalyticsServiceInterface** it's the interface which defines the basic operation that any analytics service should have (independent from the technology)
    - we will see the methods in detail in the class which implement it
- **AnalyticsService** implements *AnalyticsServiceInterface* and is in charge of manage all operations that gather data to be displayed in the analytics tab, in order to work properly it uses an **AnalyticsManagerDatabaseInterface** to exchange data with the DB
    - Set<EngageResult> **engagementAnalytics**(Date *startdate*, Date *endDate*, Entity *entity*) it calls the homonym method of the *AnalyticsManager*
    - Set<AverageRatingResult> **ratingAnalytics**(Date *startdate*, Date *endDate*, RatingType *typeOfRating*) it calls the homonym method of the analyticsManager
    - Set<RankingResult> **rankingAnalytics**(Date *startDate*, Date *endDate*, RankingType *typeOfRanking*) it calls the homonym method of the analyticsManager

ANALYTICS

There are three **analytics entities**, and are associated with the related analytic:

- **AverageRatingResult** contains the result of first Analytic.
- **EngageResult** contains the result of second analytic.
- **RankingResult** contains the result of third analytic.

AVERAGE RATING

This is the code which handle the analytics:

```
1.  @Override
2.  public Set<AverageRatingResult> ratingAnalytics(Date startDate, Date endDate, RatingType typeOfRating) {
3.
4.      Set<AverageRatingResult> res = new LinkedHashSet<>();
5.
6.      AggregateIterable<Document> result;
7.      switch (typeOfRating) {
8.          case GENRE:
9.              result = aggregateByGenre(startDate, endDate);
10.             break;
11.         case DIRECOR:
12.             result = aggregateByDirector(startDate, endDate);
```

```
            break;
        case ACTOR:
            result = aggregateByActor(startDate, endDate);
            break;
        default:
            return res;
    }

    for (Document dbObject : result) {
        res.add(createAverageRatingResultfromDocument(dbObject));
    }
    return res;
}
```

It takes the results from the type of ranking chosen by the user through the function ***aggregateBy<TypeOfRating>(****startDate, endDate****)*** and then it converts the results in the business logic object by cycling on results and apply the function ***createAverageRatingResultfromDocument(****dbObject****)****.*

Here the code of ***aggregateByGenre(****startDate, endDate****)*** (the other aggregations are analogous):

```
private AggregateIterable<Document> aggregateByGenre(Date startDate, Date endDate) {
    return FilmCollection.aggregate(Arrays.asList(
            match(and(and(
                    gte("PublicationDate", startDate),
                    lt("PublicationDate", endDate)),
                    ne("Genres", new BsonNull()),
                    ne("Rating", new BsonNull()))),
            unwind("$Genres",
                    new UnwindOptions().includeArrayIndex("arrayIndex")),
            group("$Genres",
                    avg("avg_rating", "$Rating"),
                    sum("count", 1L)), sort(descending("count"))));
}
```

Which is the translation of this *Mongo* shell code applied on *FilmCollection;* the query is composed by four stages:

```
[{$match: {
  PublicationDate: {
        $gte:ISODate(<StartDate>),
        $lt: ISODate(<EndDate>)
    },
    Genres: {$ne:null},
    Rating: {$ne:null}
}}, {$unwind: {
  path: “$Genres”,
  includeArrayIndex: “arrayIndex”,
}}, {$group: {
  _id: “$Genres”,
  avg_rating: { $avg: “$Rating”},
  count: {$sum: 1}
}}, {$sort: {
  count: -1
}}]
```

- The **Match** stage filters the films that respect the criteria of aggregation.
- The **Unwind** stage is necessary because films may have more genres.
- The **Group** stage aggregates the result grouping by genre and then it counts how may films belongs to a genre and it calculates the average rating.
- The **Sort** stage order the results to be ready to be printed on the screen.

RANKING

This is the code which handle the analytics:

```
@Override
public Set<RankingResult> rankingAnalytics(Date startDate, Date endDate, RankingType typeOfRanking) {

```

```
4.      List<RankingResult> res = new ArrayList<>();
5.
6.      AggregateIterable<Document> fromEngage;
7.      AggregateIterable<Document> fromFilm;
8.
9.      switch (typeOfRanking) {
10.         case FILM:
11.             fromEngage = aggregateEngageCollectionRankingByFilms(startDate, endDate);
12.             fromFilm = aggregateFilmCollectionRankingByFilms(startDate, endDate);
13.             break;
14.         case USER:
15.             fromEngage = aggregateEngageCollectionRankingByUsers(startDate, endDate);
16.             fromFilm = aggregateFilmCollectionRankingByUsers(startDate, endDate);
17.             break;
18.         default:
19.             return new LinkedHashSet<>();
20.     }
21.
22.     for (Document dbObject : fromEngage) {
23.         RankingResult rr = createRankingResultfromDocument(dbObject);
24.         res.add(rr);
25.     }
26.     for (Document dbObject : fromFilm) {
27.         RankingResult rr = createRankingResultfromDocument(dbObject);
28.         RankingResult.addOrMerge(res, rr);
29.     }
30.
31.     Collections.sort(res);
32.
33.     return new LinkedHashSet<>(res);
34. }
```

In order to get the ranking results is needed to get all the comments, view and favourite of target films (or users it depends by the type of rating), since recent comments are nested in films, two queries are made for archiving the result, one on ***FilmCollection*** and the other on ***EngageCollection***.

Once the two partial results are windrowed, they are converted to business logic class, merged together and sorted getting the rankings ready to be printed on the screen.

This is the Mongo shell query for the film ranking in *FilmCollection* composed by five stages:

```
1.  [{$match: {
2.    Timestamp: {
3.      $gte: ISODate(<StartDate>),
4.      $lt: ISODate(<EndDate>)
5.    }
6.  }}, {$group: {
7.    _id: {
8.      $toObjectId: '$Film'
9.    },
10.   commentCount: {
11.     $sum: {$cond: {
12. if: { $eq: [ '$Type', 'COMMENT' ] },
13.       then: 1,  else: 0}}
14.   },
15.   viewCount: {
16.     $sum: {$cond: {
17. if: { $eq: [ '$Type', 'VIEW' ] },
18.       then: 1, else: 0}}
19.   },
20.   favouriteCount: {
21.     $sum: {$cond: {
22. if: { $eq: [ '$Type', 'FAVOURITE' ] },
23.       then: 1, else: 0}}
24.   }
```

- The **Match** stage filters the actions that respect the criteria of aggregation.
- The **Group** stage aggregates the result grouping by film and counting the number of comments, view and favourite.

```
}}, {$lookup: {
  from: 'FilmCollection',
  localField: '_id',
  foreignField: '_id',
  as: 'FilmD'
}}, {$unwind: {
  path: '$FilmD',
  includeArrayIndex: 'ArrayPosition',
  preserveNullAndEmptyArrays: false
}}, {$project: {
  _id: 1,
  title_username: '$FilmD.Title',
  commentCount: 1,
  favouriteCount: 1,
  viewCount: 1
}}]
```

- The **Lookup** stage is used to get the name of the film by the ID.
- The **Unwind** stage is used to extract the film fields.
- The **Project** stage removes the unused fields.

Then are needed the comments nested in the films, they are windrowed by this three stages query:

```
[{$unwind: {
  path: "$RecentComments",
  includeArrayIndex: 'CommentNumber',
  preserveNullAndEmptyArrays: false
}}, {$match: {"RecentComments.Timestamp": {
    $gte: ISODate(<StartDate>),
    $lt: ISODate(<EndDate>)
  }
}}, {$group: {
  _id: "$_id",
  commentCount: {$sum: 1} ,
  title_username: {$first: '$Title'}
}}]
```

- The **Unwid** stage unwinds all recent comment.
- The **Match** stage filters the comments that respect the criteria of aggregation.
- The **Group** stage groups by film and count the comments.

FILM ENGAGEMENT

This is the code which handle the third analytics.

```
@Override
public Set<EngageResult> engagementAnalytics(Date startDate, Date endDate, Entity entity){
    AggregateIterable<Document> result;
    String idFilm = entity.getId();
    Film film = DBManager.filmManager.getById(idFilm);
    DBManager.filmManager.getRecentComments(film);

    HashMap<Integer, Integer> commentYear = new HashMap<>();

    film.getCommentSet().stream()
                        .map((comment) ->(Date) comment.getTimestamp())
                        .filter((date) -> !(date.getTime() < startDate.getTime()
                                            || date.getTime() > endDate.getTime()))
                        .map((date) -> {
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        return calendar;
    }).map((calendar) -> calendar.get(Calendar.YEAR)).forEachOrdered((year) -> {
        if (commentYear.containsKey(year)) {
            int currentCount = commentYear.get(year);
            commentYear.put(year, currentCount + 1);
        } else {
            commentYear.put(year, 1);
        }
    });

    result = engageAggregate(startDate, endDate, idFilm);

```

```
    Set<EngageResult> res = new LinkedHashSet<>();

    for (Document dbObject : result) {
        EngageResult engageResult = createEngageResultFromDocument(dbObject);
        Integer commentCount = commentYear.get(engageResult.getYear());

        if (commentCount != null) {
            commentCount += film.getCommentSet().size();
            engageResult.setCommentCount(commentCount.longValue());
        }

        res.add(engageResult);
    }
    return res;
}
```

To obtain the engage by year of a specific film, we need to get all the views, comments, and favourite relative to that film. Views and favourite are situated in the ***EngageCollection***, comments too, but not all. How explained before, there are some recent comments, that are nested on the film document.

Now are described how the code of the pipeline that perform the aggregation work, it is a four-stage pipeline:

```
[{$match: {
  Film: <idFilm>,
  Timestamp: {
    $gt: ISODate(<StartDate>),
    $lt: ISODate(<EndDate>)
  }
}}, {$project: {
  Film: 1,
  Timestamp: 1,
  View: {
    $cond: {
      if: {$eq: ["$Type", "VIEW"]},
      then: 1,
      else: 0
    }
  },
  Favourite: {
    $cond: {
      if: {$eq: ["$Type", "FAVOURITE"]},
      then: 1,
      else: 0
    }
  },
  Comment: {
    $cond: {
      if: {$eq: ["$Type", "COMMENT"]},
      then: 1,
      else: 0
    }
  }
}}, {$group: {
  _id: {$year: "$Timestamp"},
  ViewCount: {
    $sum: "$View"
  },
  FavouriteCount: {
    $sum: "$Favourite"
  },
  CommentCount: {
    $sum: "$Comment"
  }
}}, {$sort: { _id: 1 }}]
```

- **Match stage:** in this stage the documents are filtered, to get only the relevant document
- **Project stage:** basically it prepare the documents to be processed, by the group.
- **Group stage:** Grouping by year, it count the number of views, favourite and comments (in the engage collection) for that year.
- **Sort stage:** It sort in ascending order, in base on the year.

INDEXES

The following are the indexes deemed necessary for an effective retrieval of documents from the database:

ENGAGE COLLECTION

- **Film (asc) Timestamp(desc):** For the analytic that have been defined, film and timestamp, always go together in the match stage. An index like that can increase of a huge amount the performance of the db. Below is shown a comparison of query performance with and without indexes to demonstrate their effectiveness:

```
{Film:"5e2025be8472fbdeee121311",Timestamp:{$gt:ISODate('2012-12-17T00:00:00Z'),$lt:ISODate('2020-12-17T00:00:00Z')}
```

Query Performance Summary

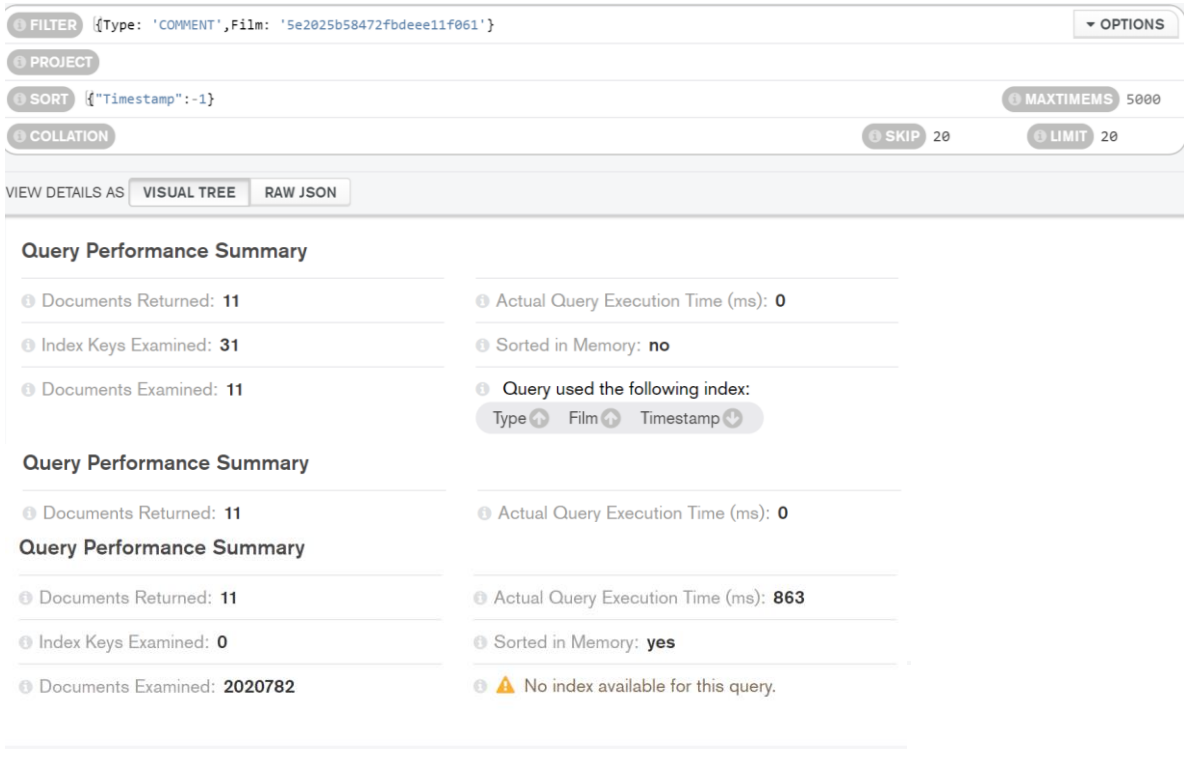
Documents Returned: **8**	Actual Query Execution Time (ms): **0**
Index Keys Examined: **8**	Sorted in Memory: **no**
Documents Examined: **8**	Query used the following index: Film ↑ Timestamp ↓

Query Performance Summary

Documents Returned: **0**	Actual Query Execution Time (ms): **125**
Index Keys Examined: **0**	Sorted in Memory: **no**
Documents Examined: **291716**	⚠ No index available for this query.

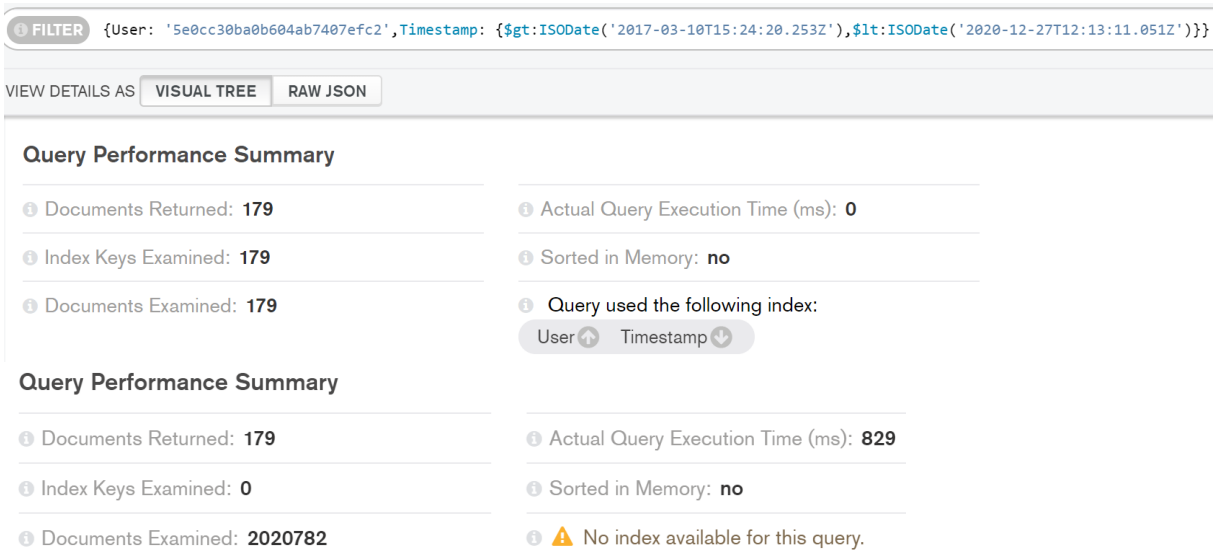
Without the index, the query must range through all the collection.

The same index is also essential for the query that provides older comment pages on a particular film, as shown below:



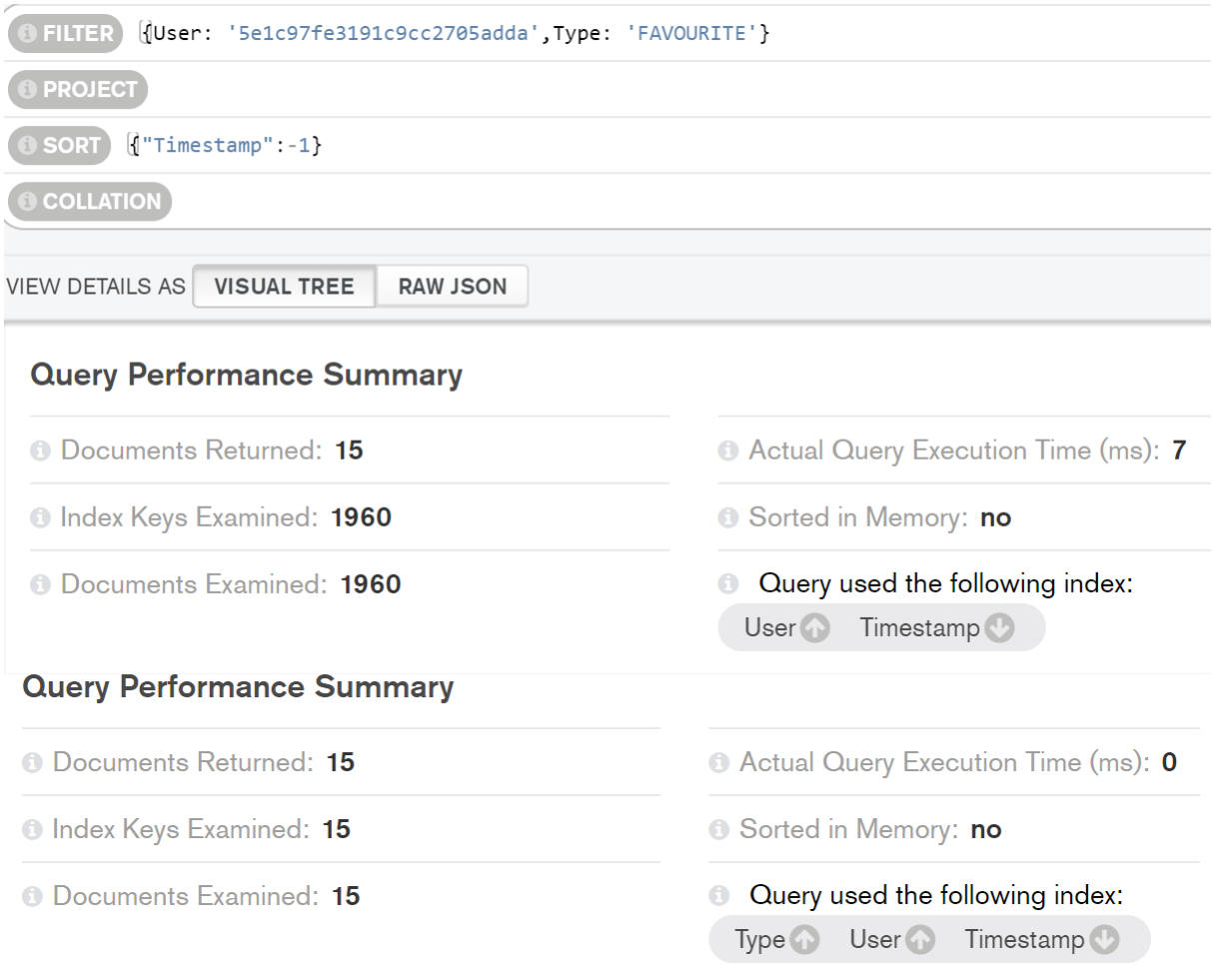
Using a triple index (Type, Film, Timestamp) does not increase performance significantly, therefore it has not been added, in order to not weigh down the writing in the collection and avoid the memory consumption necessary to maintain it.

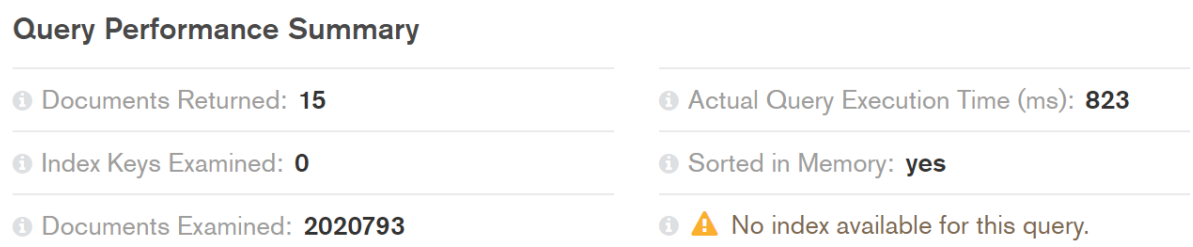
- **User(asc) Timestamp(desc):** For the same type of analytics, this index is also required, as shown below:



Without the index, the query must range through all the collection.

the same index is also essential to efficiently retrieve user's favorites, as shown below:

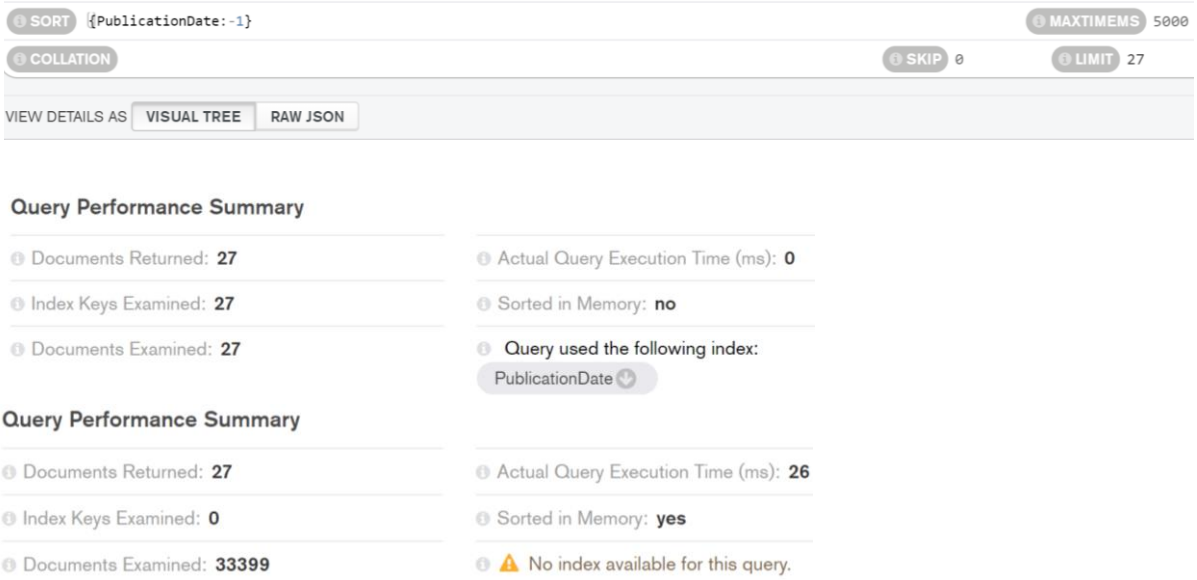




Using a triple index (Type, User, Timestamp) does not increase performance significantly, therefore it has not been added, in order to not weigh down the writing in the collection and avoid the memory consumption necessary to maintain it.

FILM COLLECTION

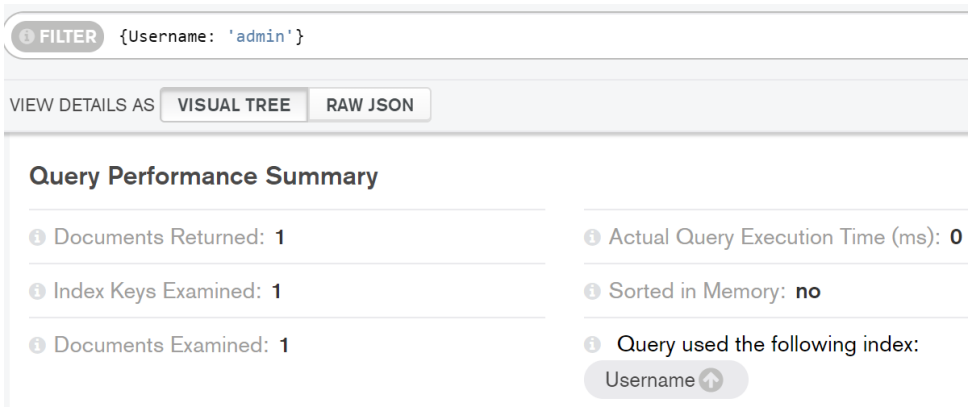
- **PublicationDate(desc):** This index is necessary every time is opened the page for searching films, as both the first opening and after a search, the films are sorted by publication date and limited in number so as to fit in one screen:

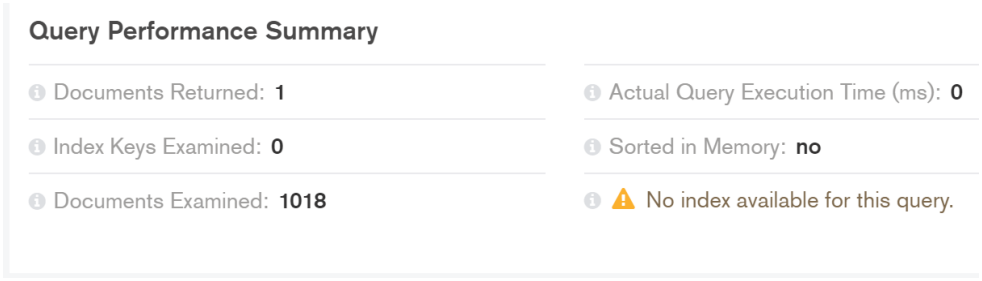


Without the index, the query must range through all the collection and re-order the documents in memory.

USER COLLECTION

- **Username(asc):** This index is necessary every time the login is made, in order to verify the existence of the username and the correctness of the password:





Without the index, the query must range through all the collection.

In conclusion, with the addition of these indexes almost all of the queries made on the database are covered, both by the analytics and by the normal functioning of the app, as the rest of the operations have been designed in order to use the index on the documents id directly.

REPLICAS

SETUP

To activate the replicas, the steps below must be followed:

1. Access one of the servers in the cluster.
2. In the mongo shell, create a configuration document, in our case:

```
… > rsconf = {
                _id: "mdbDefGuide",
                members:[
                    {_id: 0, host: "lsmsdcluster-shard-00-00-yeauu.mongodb.net:27017"},
                    {_id: 1, host: "lsmsdcluster-shard-00-01-yeauu.mongodb.net:27017"},
                    {_id: 2, host: "lsmsdcluster-shard-00-02-yeauu.mongodb.net:27017"}
                ]
            }
```

3. Pass the configuration to the function rs.initiate():
   `MongoDB Enterprise LsmsdCluster-shard-0:PRIMARY> rs.initiate(rsconf)`

One of the servers will be elected randomly as primary and the replicas will be created with default settings.

READ FROM SECONDARIES

Applications that require strongly consistent reads should not read from secondaries replicas. They are usually within a few milliseconds of the primary, however there is no guarantee of this.

Since one of the requirements of this application expects great data accuracy, as long as a single server is able to manage all the reading requests, the use of secondary servers is not foreseen for this purpose. If the traffic will exceed the capacity of the primary server, the readings of the copies contained in the secondary servers can be activated, as explained below, to improve the availability:

1. Access the secondary server (*00* or *01* in our case):
   `mongo "mongodb://root:root@lsmsdcluster-shard-00-01-yeauu.mongodb.net:27017" –tls`
2. Enter the following command which enables the reading of the documents contained in secondary server:
   `MongoDB Enterprise LsmsdCluster-shard-0:SECONDARY> rs.slaveOk()`

In any case, it is not possible to write to a secondary server directly from a client, limiting the scalability given by the replicas to reads only.

REPLICA INDEXES

Preventing one of the replicas from creating indexes can be useful to limit the memory occupied, assuming that at most one server at a time may have a failure (in any case due to the method of election of the primary server by majority, the application would in case stop in presence of a failure of two on three servers), one of the replicas indexes can be disabled as it would never be chosen as primary in any case.

IN order to do it we have to specify `"buildIndexes" : false` in the member's configuration, this is a permanent setting, the server can never be configurated to build index again.

HIDDEN SERVER

If, as explained in the previous sections, is enabled one of the two secondary servers for reading and prevented the creation of the indexes to the third server, this can be set as hidden, in order to give a higher priority to the other server for the creation of replicas. Furthermore, setting a server as hidden prevents any type of access by a client to it in order to inhibit the reading of data out-of-date and without indexes.

To do it is necessary to specify `"hidden" : true` in the member's configuration, to make the server visible again set the field to `false`.

PRIORITY

Changing the priority of servers can be useful in the following cases:

- To elect a specific server as primary, its priority can be set higher than the priority of the other servers, making it elected as primary whenever possible.
- If one of the servers is prevented from building indexes or set as hidden, as indicated in the previous section, it must take priority 0 in order to prevent its choice as primary.

The following commands, on the shell of one of the servers, can be used to change the priority of server *00*:

```
1.  … > var config = rs.config()
2.  … > config.members[0].priority = 0
3.  … > rs.reconfig(config)
```

WRITE CONCERN

Since the quality of information within the application is one of the requirements, a two-phase-commit mechanism is provided for the collections of films and users, so that readings from replica servers provide updated data. On the other hand, it is not expected for writings in the engage collection to wait for the replications, in order to improve the performance.

To ensure this, the following lines of code have been included within the managers of the collections:

- **FilmManager:**

```
1.  public FilmManager() {
2.          FilmCollection = DBManager.getMongoDatabase()
                                      .getCollection("FilmCollection")
                                      .withWriteConcern(WriteConcern.MAJORITY);
3.  }
```

- **UserManager:**

```
1.  public UserManager() {
2.          UserCollection = DBManager.getMongoDatabase()
                                      .getCollection("UserCollection")
                                      .withWriteConcern(WriteConcern.MAJORITY);
```

```
3.  }
```

- **EngageManager:**

```
1.  public EngageManager() {
2.          EngageCollection = DBManager.getMongoDatabase()
                                        .getCollection("EngageCollection")
                                        .withWriteConcern(WriteConcern.ACKNOWLEDGED);
3.  }
```

CONFIGURATION

Unfortunately in the servers provided by *MongoDB Atlas* is not possible to change the priority and consequently to do all the operations that are required.

This would be the configuration which we would recommend for a cluster of three servers:

Primary Server (priory *2*)**:** it is the only one authorized to receive write operations.

Secondary Server (priory *1*)**:** it contains indexed and updated replicas thanks to the write concerns, it replaces the primary in case of failure and provides read access to the replicas it contains.

Hidden Server (priory *0*)**:** it contains replicas that are not indexed in order to save space, cannot be elected as primary and cannot be accessed for reading.

NOTE: it must be considered that the choice of the configuration of the replicas also depends on the type of hardware available and the specific problems encountered, the one shown above is a solution that can be applied using three servers of which the hidden one can also have limited hardware features and be used only to avoid data loss.

SHARDING

Given two or more clusters of replicas with a configuration that can be the one above, it is possible to distribute the load of the database on multiple clusters by means of ***sharding***.

A recommended setup is to use one ***mongos*** process *per* application server, on the same machine; every *mongos* keeps a “table of contents” that tells it which shard contains which data. Through *mongos* is possible to add the replica sets as shards, it is also important to enable the ***balancer*** that will ensure that data is *evenly* distributed across all shards.

Firstly, it is needed to enable sharding on the database *via* the command:

```
… > sh.enableSharding("PisaFlix")
```

The only other things that needs to be chosen are the ***sharding keys***, which must be chosen from one of the collection *indexes*. In our case, the chosen *sharding keys* are:

- **Film** + **Timestamp** by *range* for **EngageCollection**, using the command:
  ```
  … > sh.shardCollection("PisaFlix.EngageCollection", {"Film" : 1, "Timestamp": 1})
  ```
- **_id** by *hash* for **FilmCollection**, using the command:
  ```
  … > sh.shardCollection("PisaFlix.FilmCollection", {"_id" : "hashed"})
  ```
- **_id** by *hash* for **UserCollection**, using the command:
  ```
  … > sh.shardCollection("PisaFlix.UserCollection", {"_id" : "hashed"})
  ```

The choice of **Film** + **Timestamp** by range for *EngageCollection* is justified by the following reasons:

- This index covers as many queries as possible, as indicated in the index section. A similar choice could have been made for *User + Timestamp* but would have covered fewer queries. This allows to reduce the number of *shards* to visit each time a query uses that index.
- Thanks to the automatic division of the chunks carried out by the *mongos*, using the film and the timestamp of the engages as a sharding key allows to distribute the load evenly on all the shards, as the engagements are carried out randomly on the various films in the database.
- Hashing the sharding key, albeit increasing load balancing, would not have allowed for range-based queries for that index, which are essential for analytics.

The choice of **_id** by hash for *FimCollection* and *UserCollection* is justified by the following reasons:

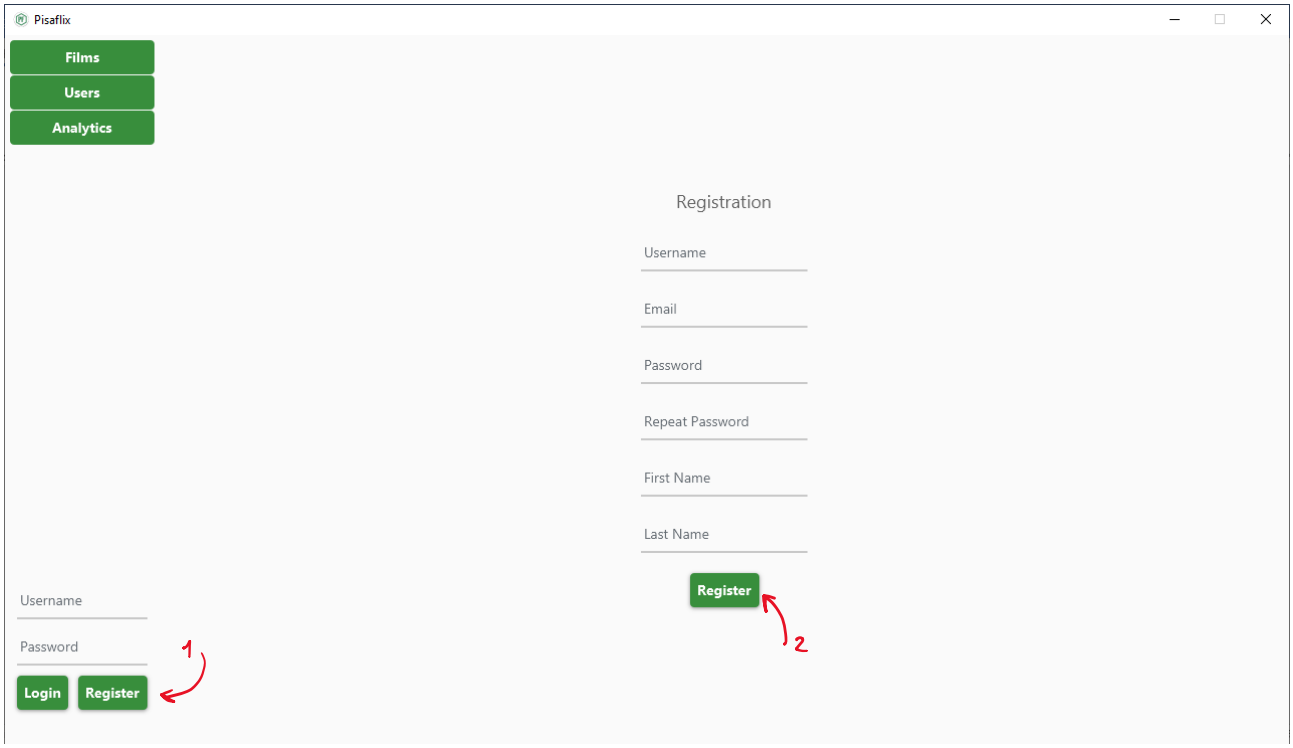
- For the *FimCollection*, the *PublicationDate* index could have been chosen for range queries as *sharding key*, but an ascending shards, as films with a more recent release date are usually added, would cause balance problems, as operations would almost always focus on last chunks, then on a single shard. therefore the *_id* was preferred as the sharding key.
- For the *UserCollection* it would have been possible to use the *Username* as a hash, but most of the operations foresee the recovery of the users through the id, therefore the *_id* was preferred as the sharding key.

USER MANUAL

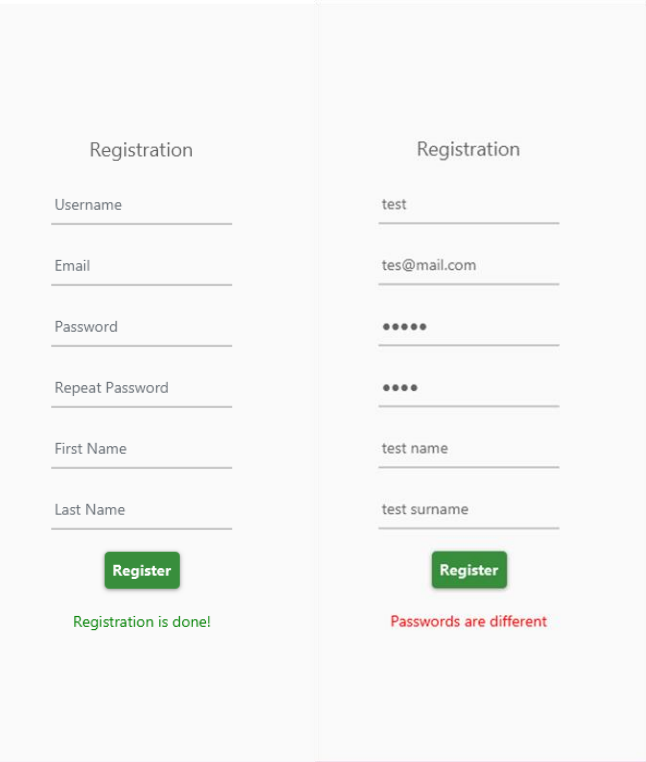
The graphic interface is divided in two sides; a menu on the left side and a space on the right side where the application pages will be displayed. Below the menu it is possible to log in by filling the apposite form:

REGISTRATION AND LOGIN

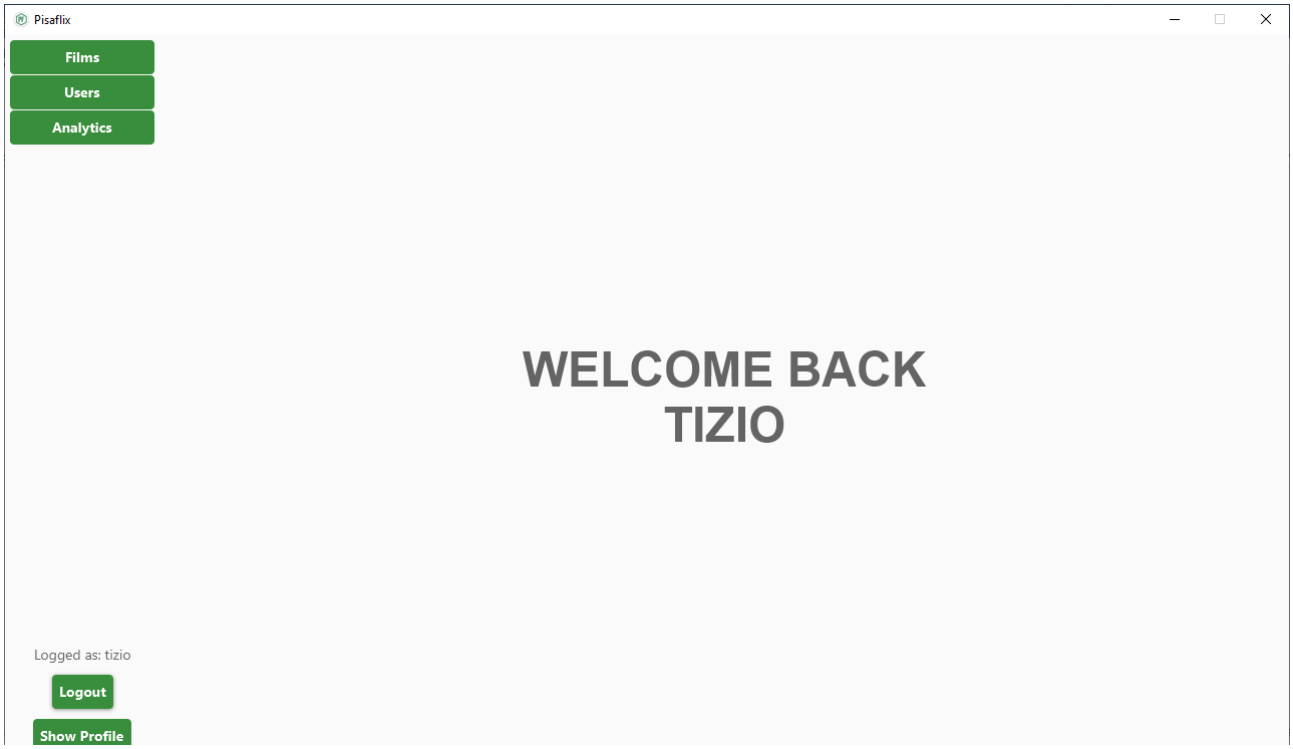
A new user can register by clicking the specific button (1) located in the bottom left corner. This will request the registration page which the user can fill up with his own information and then register (2):



The application will inform the user about any kind of issue after having clicked on the register button. The same is true for a successful registration:

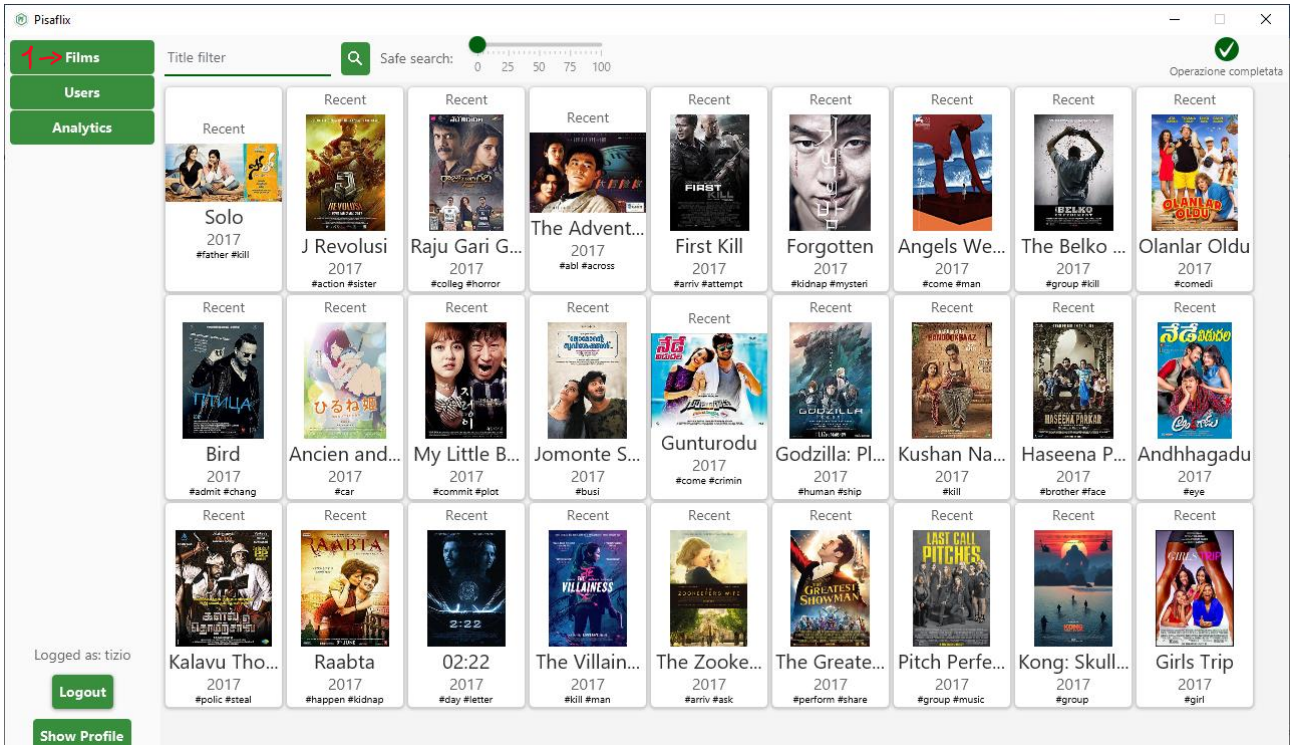


Once registered, the user can log in with the credentials chosen by filling up the form in the bottom left corner. This is the welcoming page:

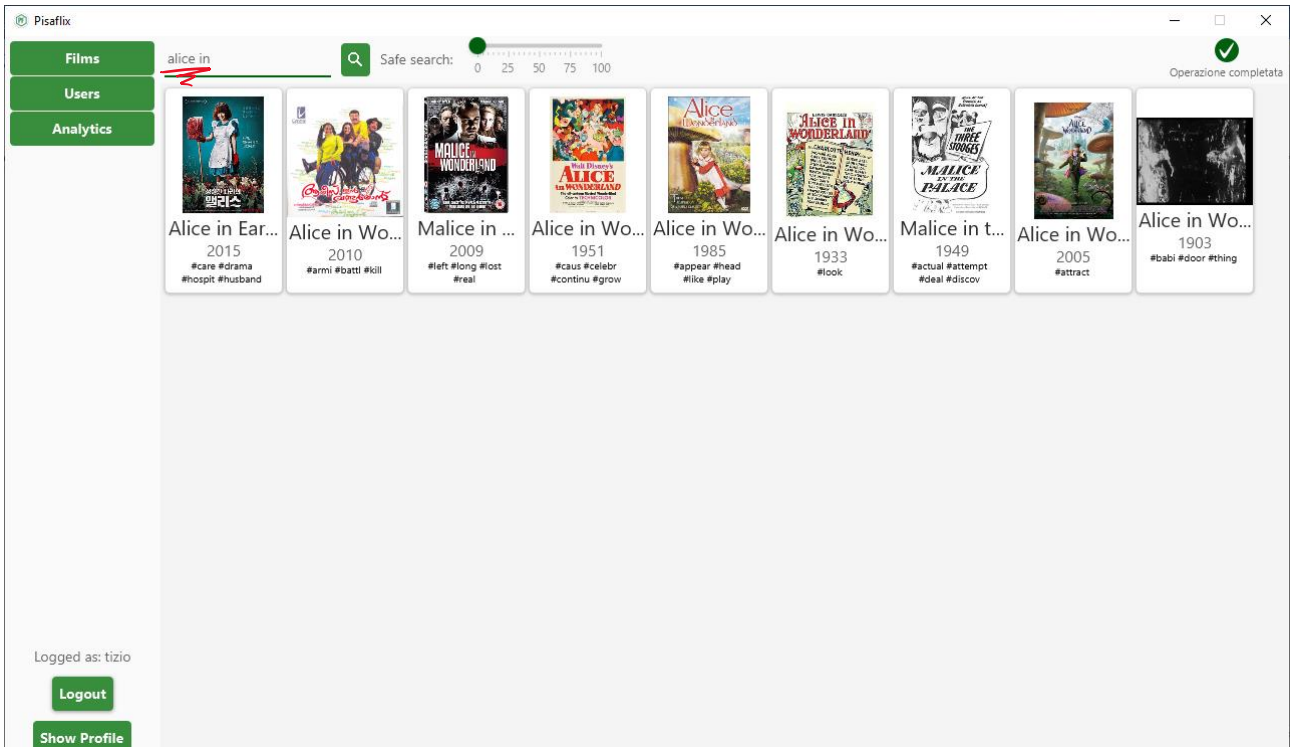


BROWSING FILM

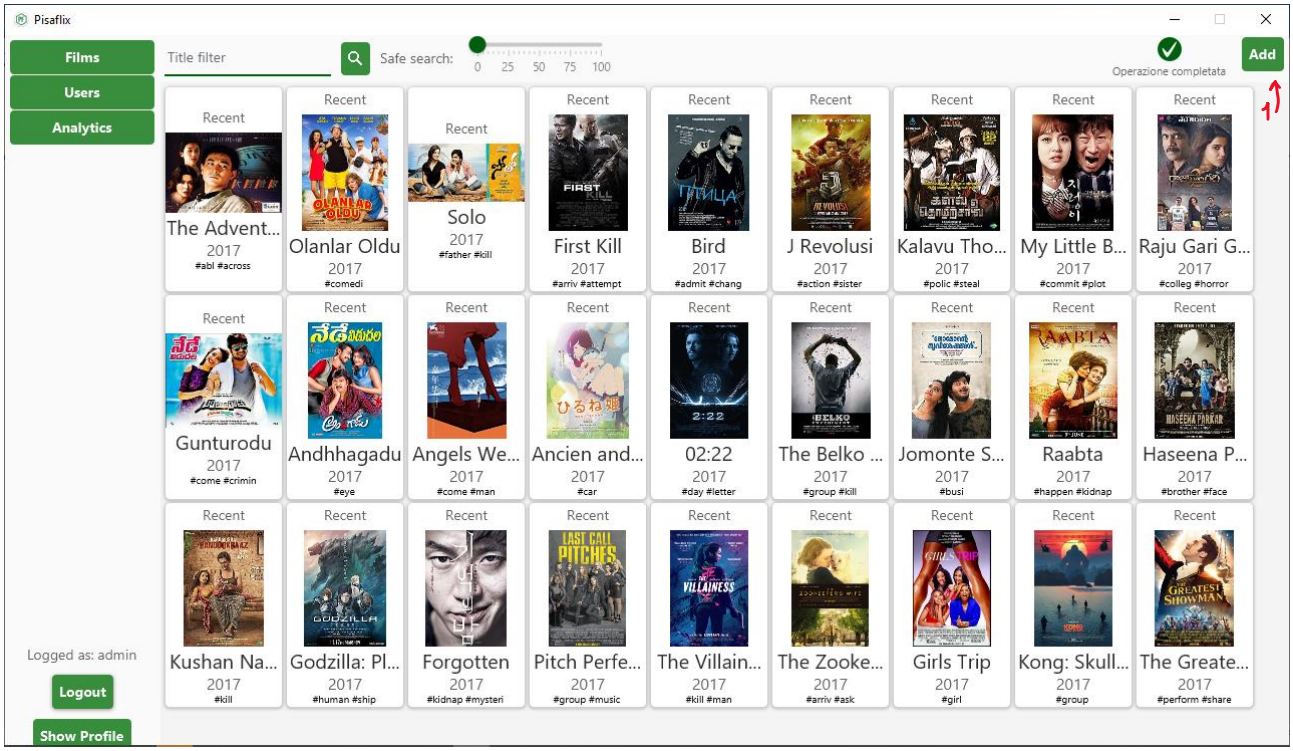
A user can browse (even without being registered) films by clicking the apposite bottom (1) in the top left corner:



In the browse films the user can search for a specific item filtering by title:



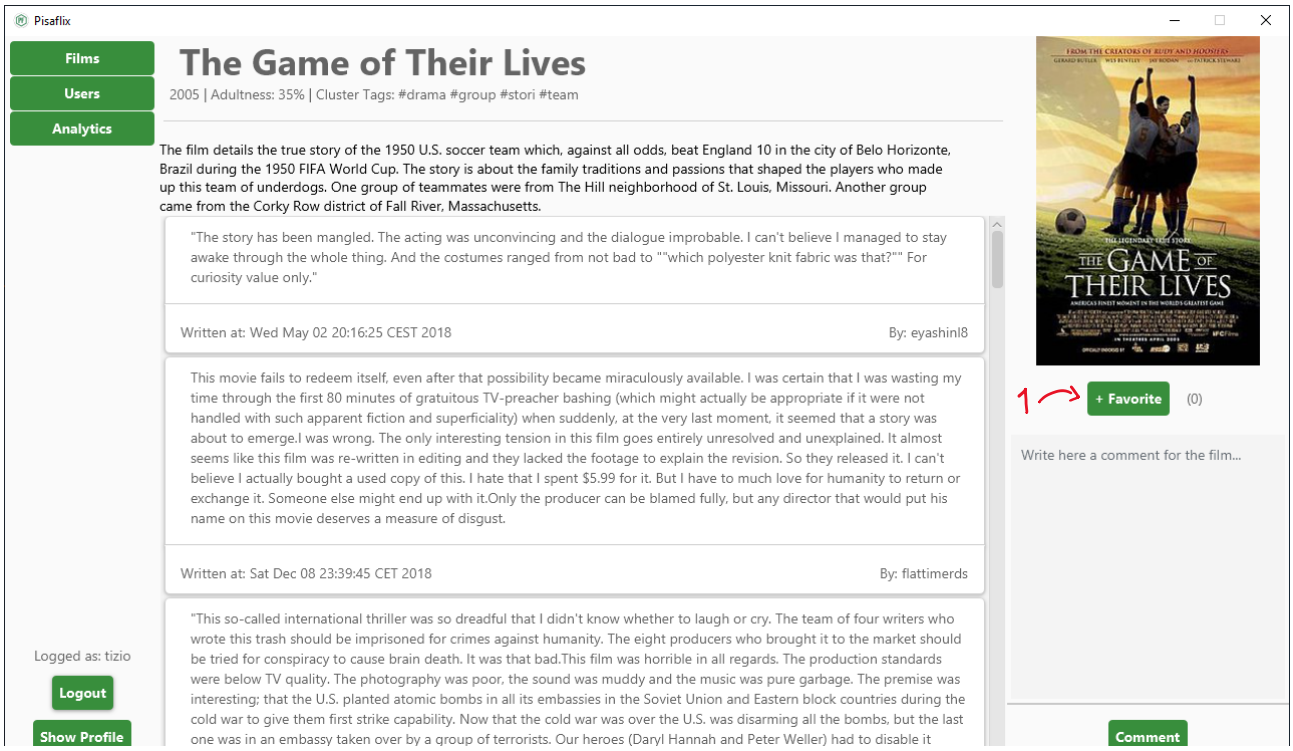
A user with the right privileges can also add a new film by clicking on the add button (1) of the top right corner:



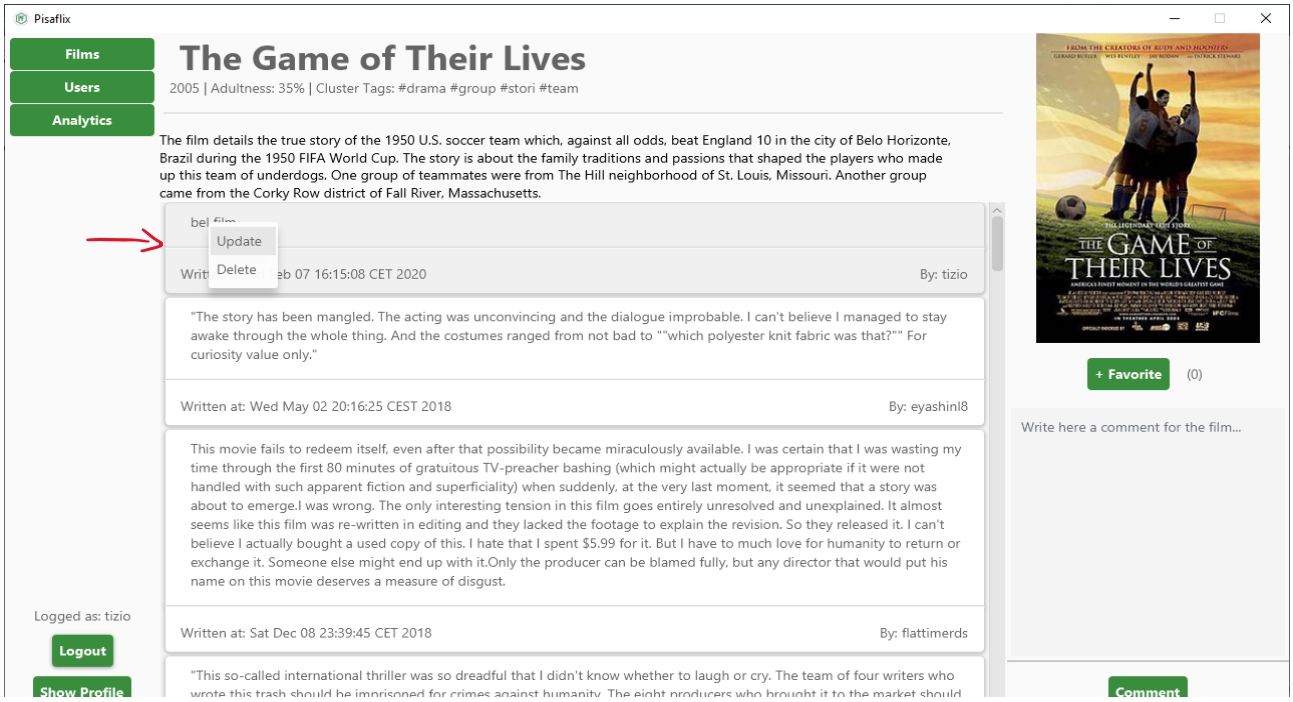
FILM DETAILS

After clicking on a film during browsing, the application will show the film detail page which contains all the information about it and also all the recent comments made by users.

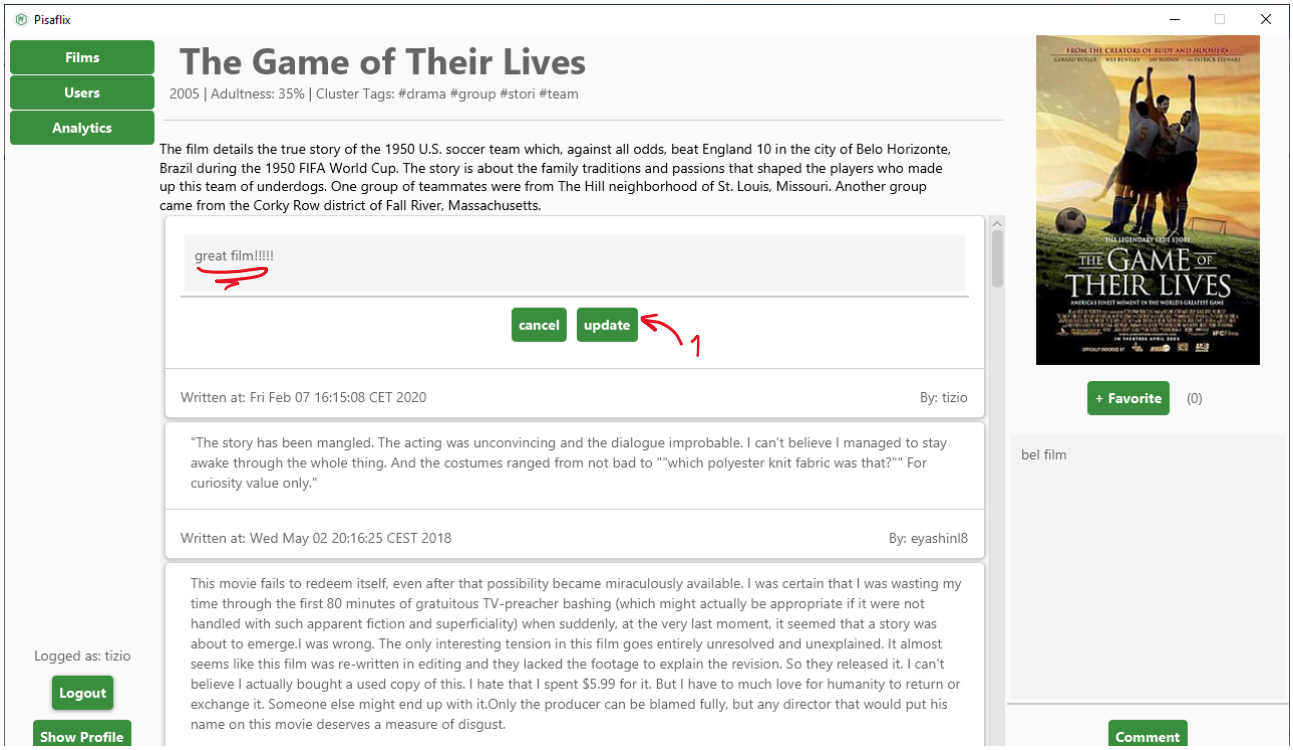
In that page a user, if logged, can add the film to its favourite (1) (by clicking the apposite button in the right side of the application) or comment it:



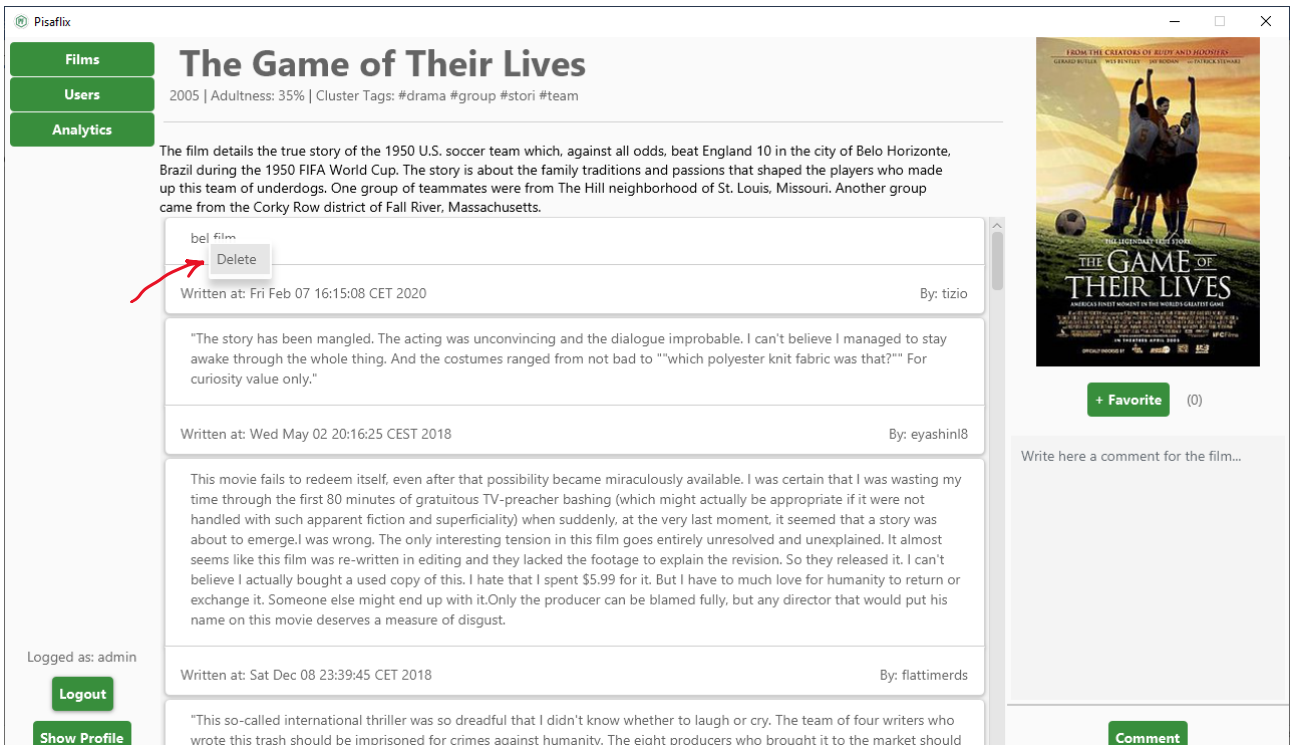
Then the user can also modify/delete its own comments by right clicking on them:



To perform the update, click on update (1) once the change has been made:



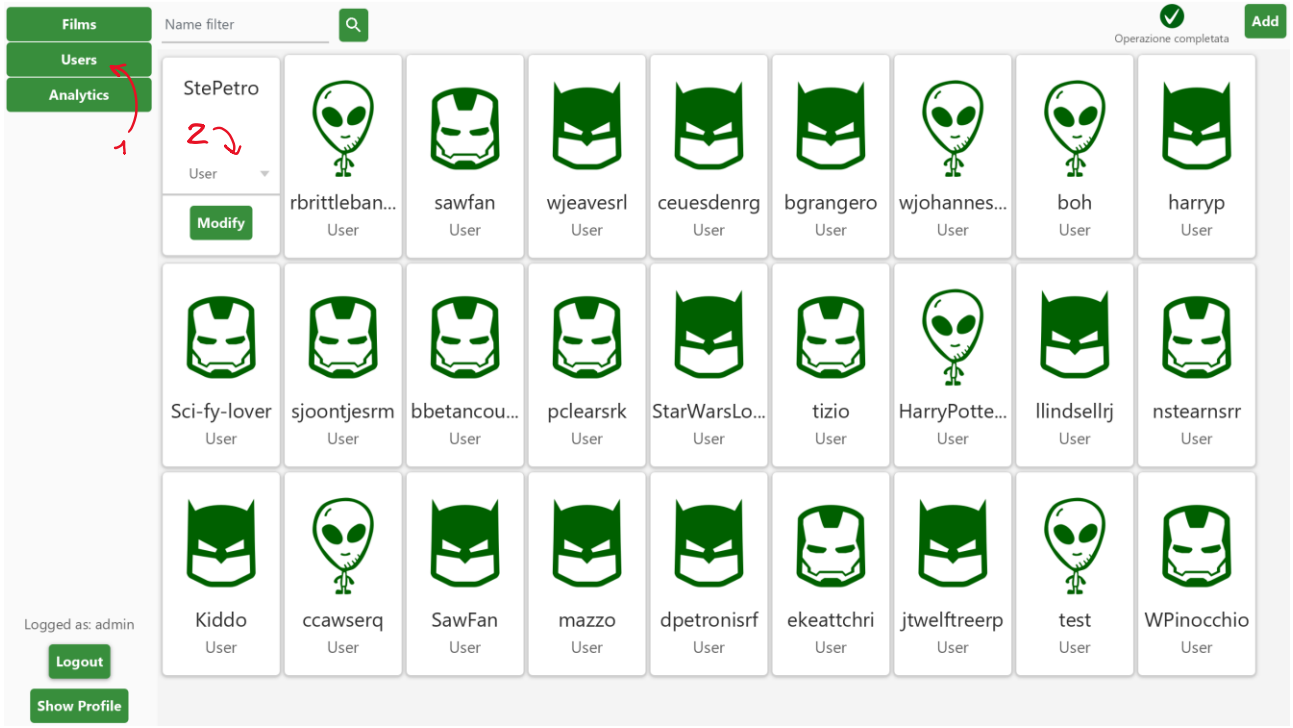
With the right privileges a user can also delete other users' comments, in the same way:



BROWSING USERS AND DETAILS

Similarly to films, a user can also navigate through other users by clicking the apposite button (1) in the top left corner, there it can see all usernames and privileges.

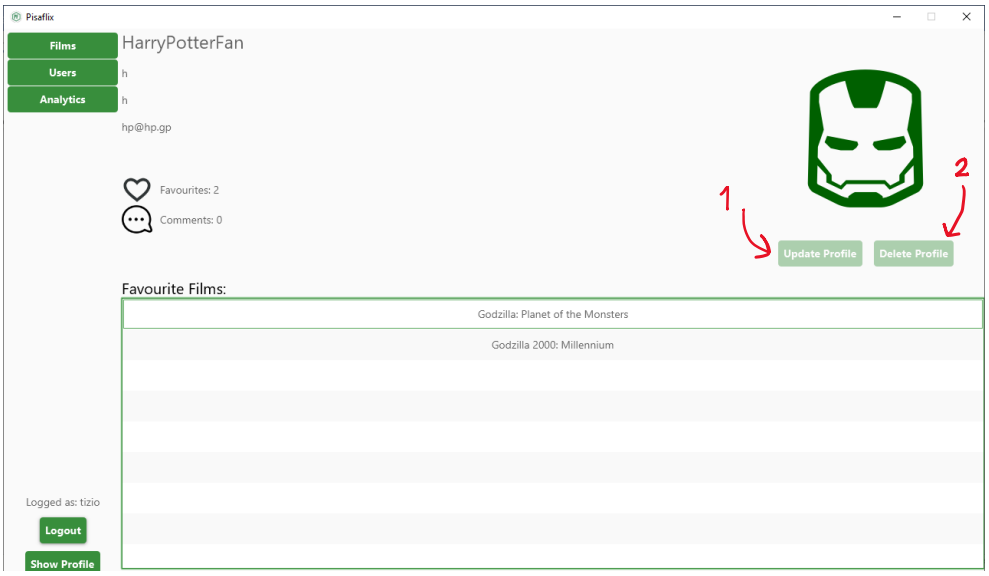
With the right privileges a user can modify other user's privileges by right clicking on them ad using the apposite menu (2):



Once the user clicks on a user while browsing it will open its detail page, there it's visible how many favourite/comment a user did and the list of his favourite films.

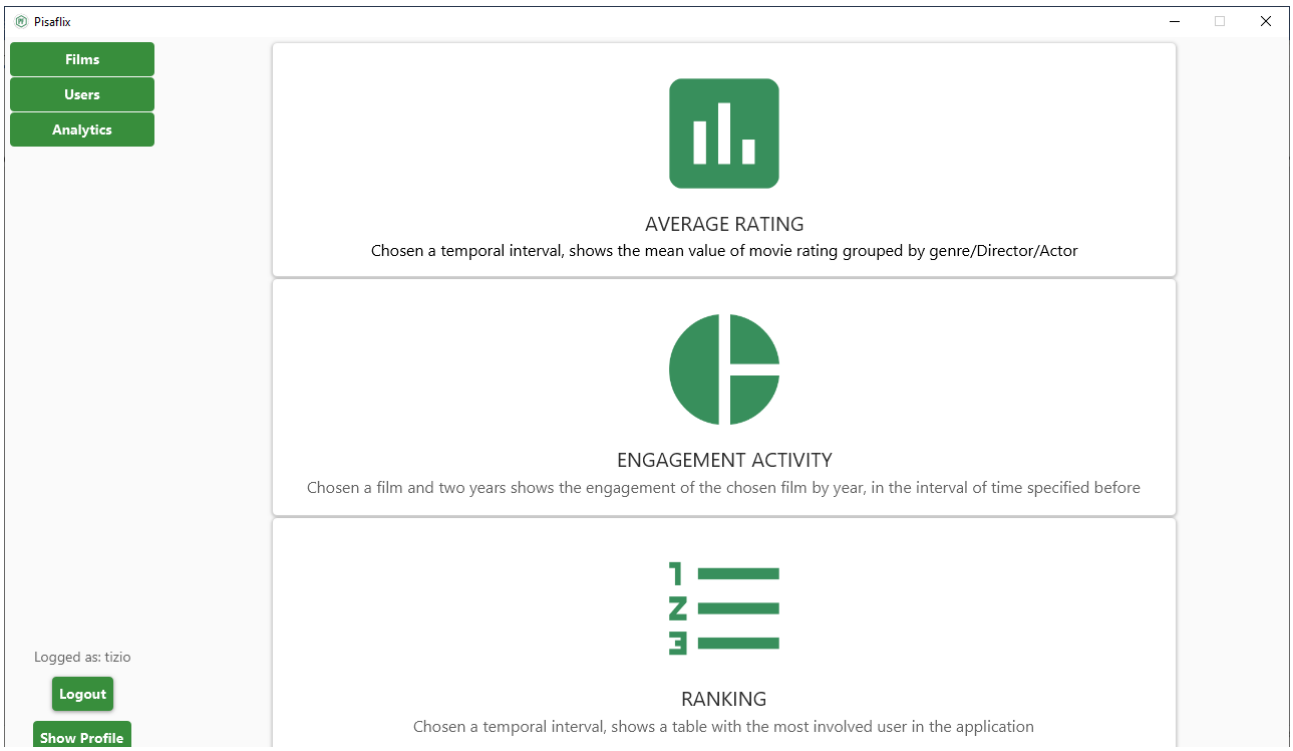
When browsing the user can also click on his own detail page, then he can modify (1) his information or delete (2) his account (the same page is accessible by the apposite button in the bottom left corner after the login).

With the right privileges, once a user detail page is opened, the user can have the possibility to delete (2) another user account.



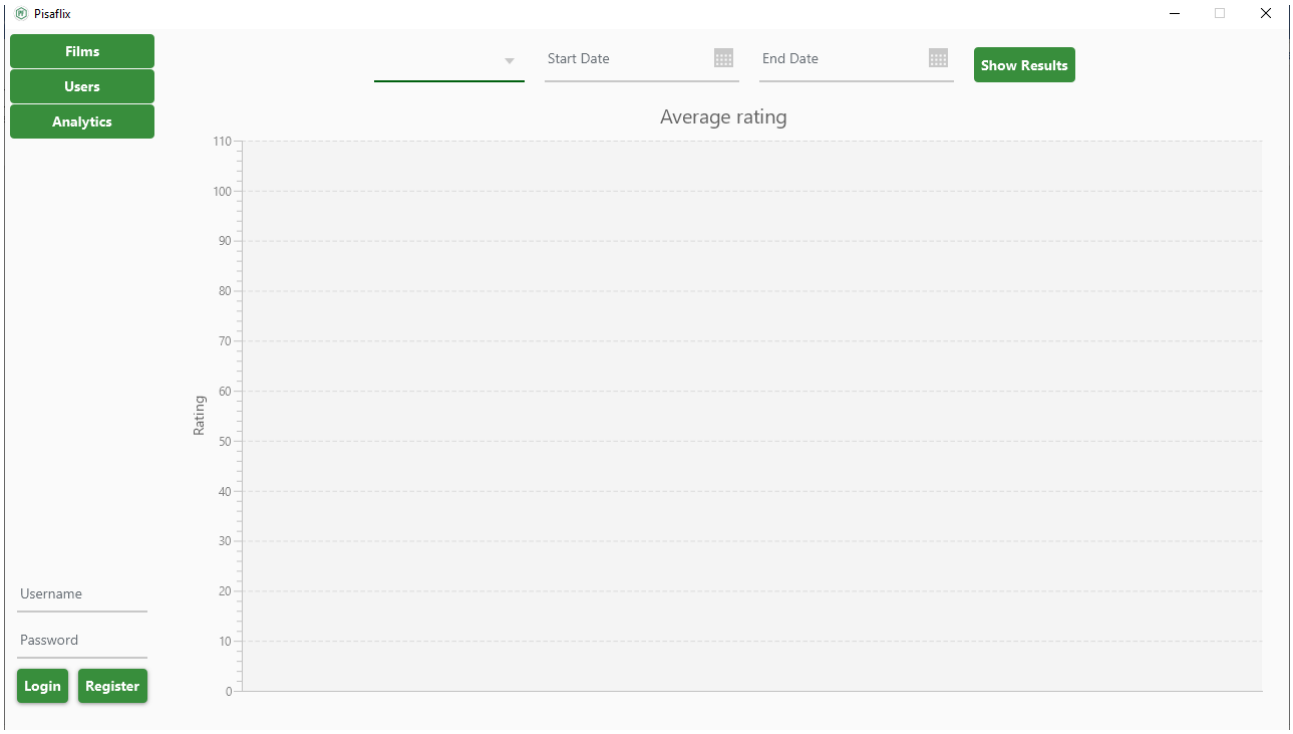
BROWSING ANALYTICS

By clicking on the Analytics button you can access the following page:

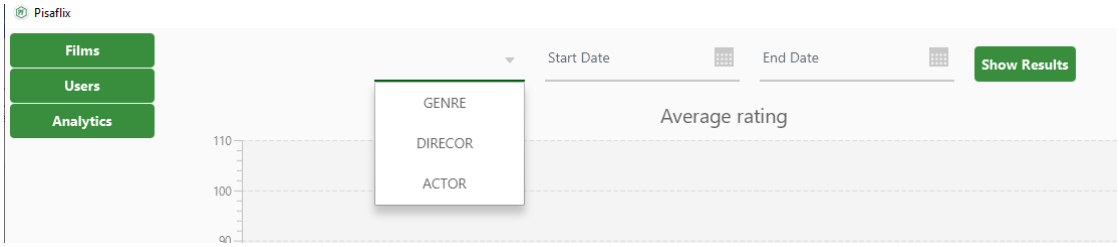


AVERAGE RATING

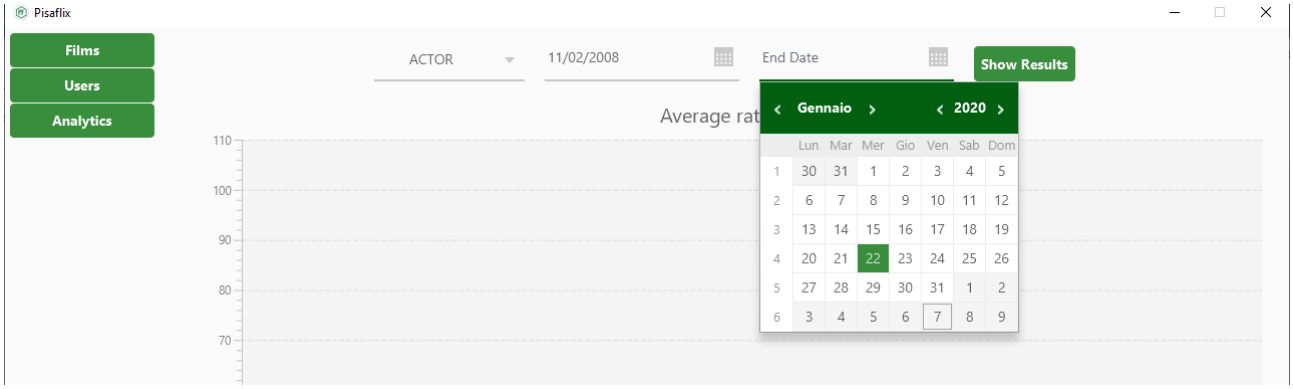
This is the main page:



It is possible to select the subject of the analytics from a drop menu:



Then is necessary to insert Start Date and End Date by either writing them or selecting them from a calendar menu:

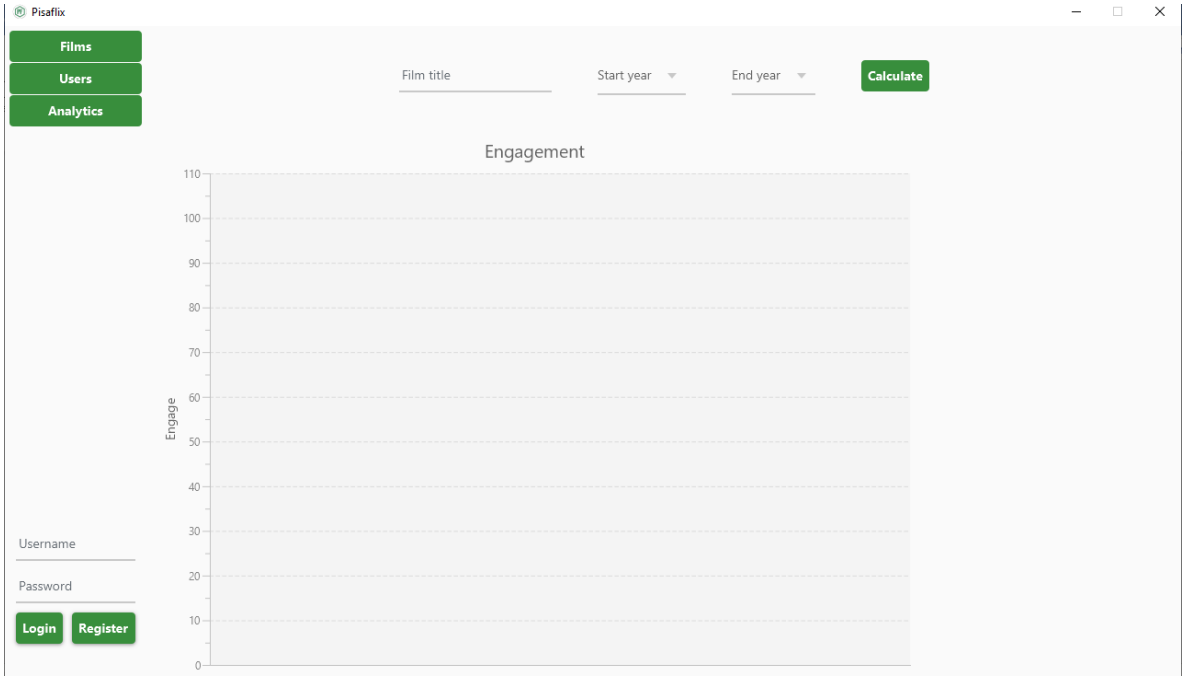


After clicking the “Show Results” button the bar chart will be populated:

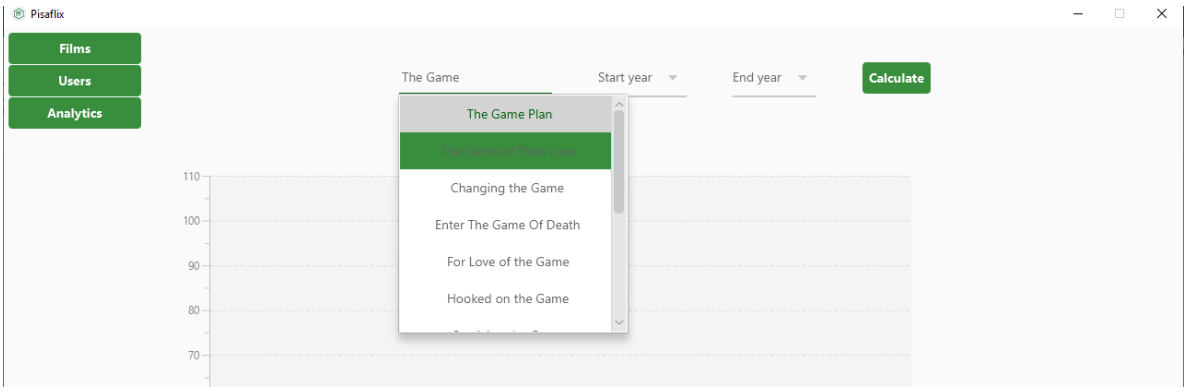
On the Y axis is present the average rating, on the X axis the name of the actor followed by the number of films in which he/she has acted in the specified time interval, the same chart is provided with the other selectable options.

ENGAGEMENT ACTIVITY

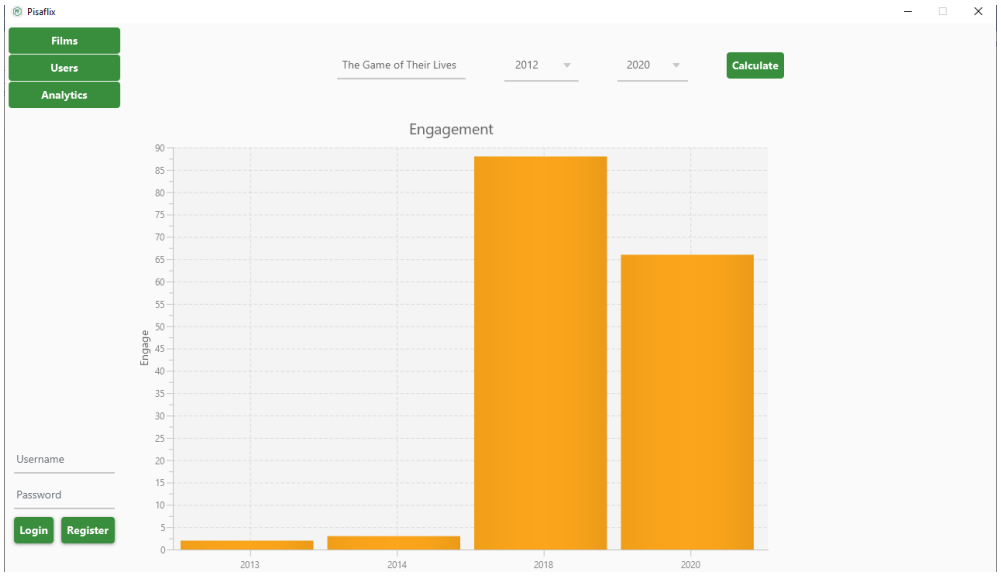
This is the main page:



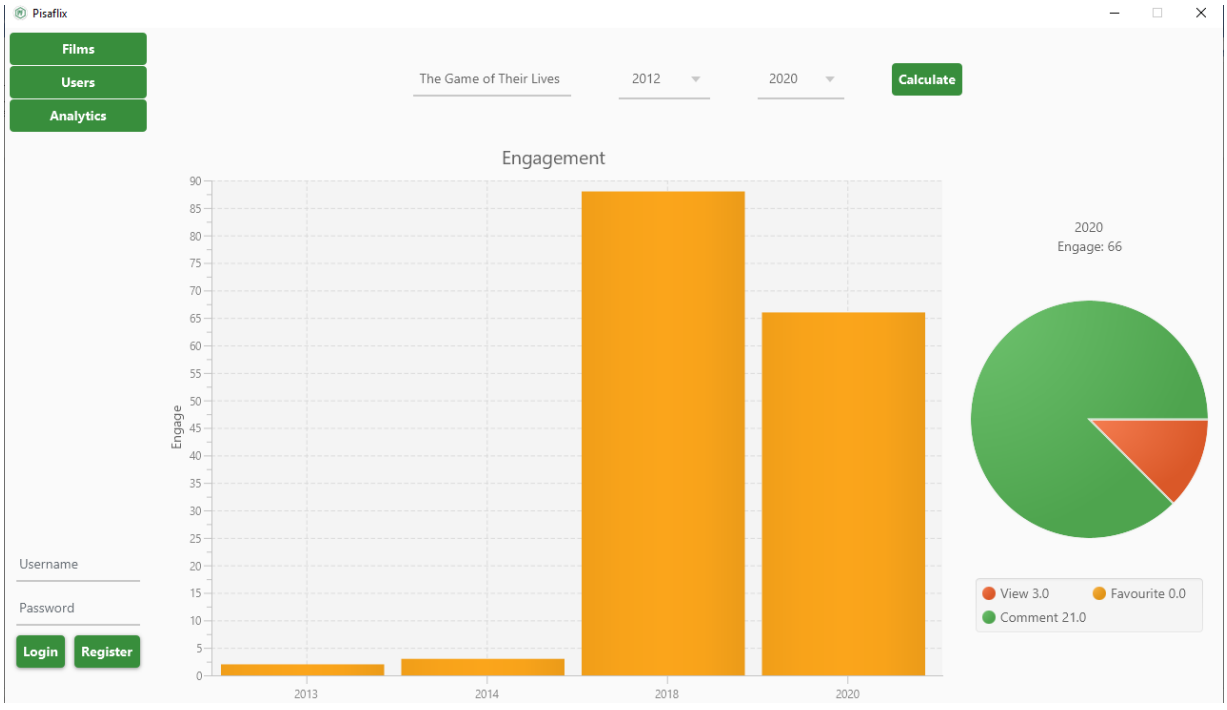
A user can insert a film by searching for it:



After the selection of the start year and the end year and by clicking the “Calculate” button, the bar chart will be populated:

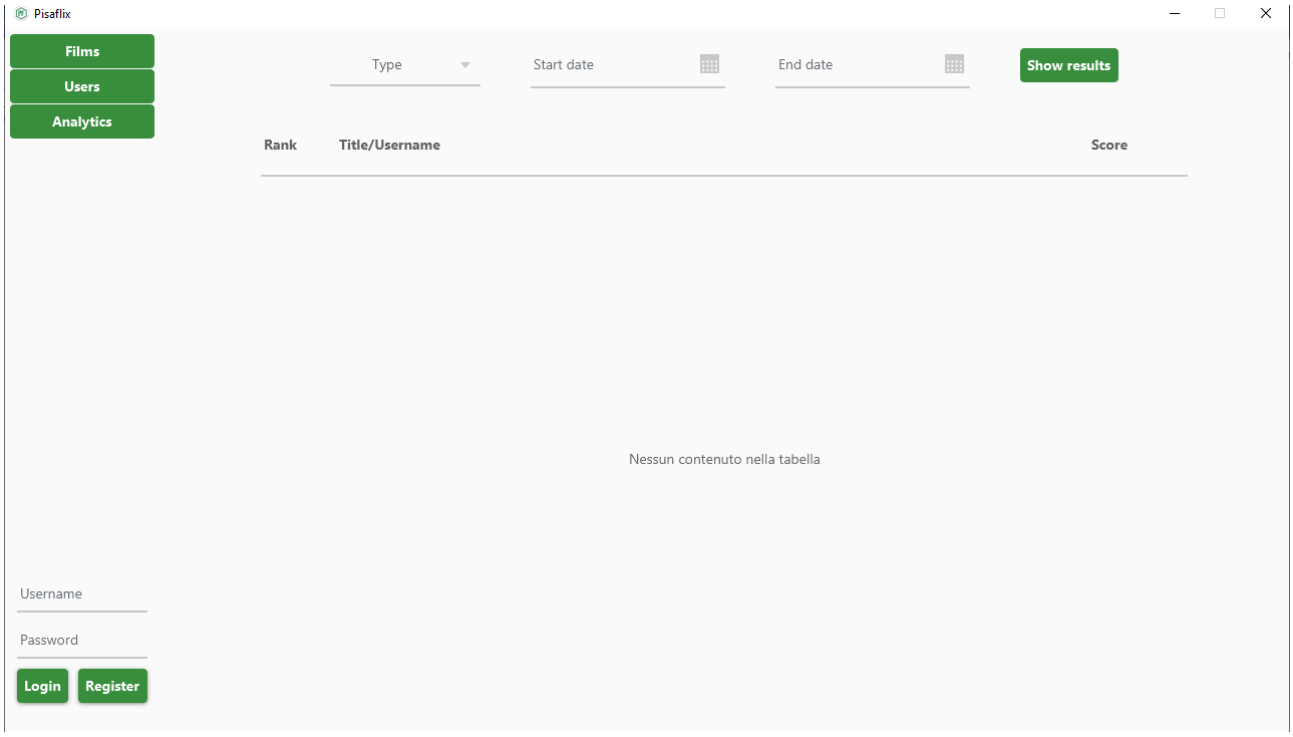


By clicking on one column is it also possible to see a pie chart that explains the proportions of all the different components that contributed to the engagement value of that year:

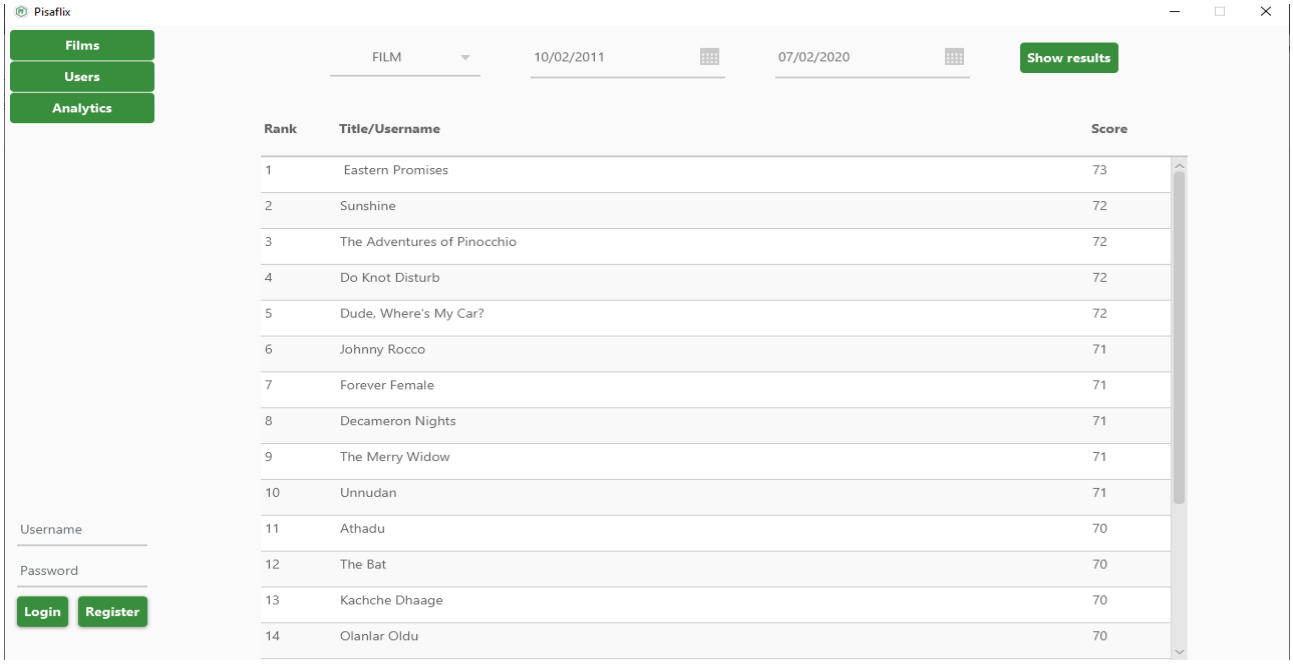


RANKING

This is the main page:



After selecting either “FILM” or “USER” from the drop-down menu, and inserting the two required dates, by clicking on the “Show results” button the table will be populated:



The various results will be arranged by their score on the table showing the top twenty entities, it I also possible to double click on a film/user to visit its detail page.