

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](#_Toc28614896)

[Description 3](#_Toc28614897)

[Requirements 3](#_Toc28614898)

[Main Actors 3](#_Toc28614899)

[Functional 3](#_Toc28614900)

[Non-Functional 4](#_Toc28614901)

[Analytics 4](#_Toc28614902)

[Use Cases 5](#_Toc28614903)

[ANALYTICS USE CASES And Mockups 6](#_Toc28614904)

[View Average Rating 6](#_Toc28614905)

[View Ranking 7](#_Toc28614906)

[View Engagement Activity 8](#_Toc28614907)

[Analysis Classes 9](#_Toc28614908)

[Data Model 10](#_Toc28614909)

[Architecture 10](#_Toc28614910)

[Interface Design Pattern 10](#_Toc28614911)

# 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 provide a service to help you to choice 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, cinemas or projections.
* **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 *Projections* scheduled for a *Film*.
4. *Users* can **view** the ***statistics*** of a *Cinema* or *Film* page.
5. *Users* can **view** a set of **analytics** on Movie and other user.
6. *Users* can **register** an account on the platform.
7. *Users* can **log in** as *Normal users* on the platform in order to do some specific operations:
   1. If logged a *Normal user* can **add/remove** to **favourite** a *Movie*.
   2. If logged a *Normal user* can **comment** a *Movie*.
   3. If logged a *Normal user* can **modify** his *Movie Comments*.
   4. A *Normal user* can **modify/delete** his account.
8. *Users* that can **log in** as *Social moderator* can do all operation of a *Normal user* plus:
   1. If logged as *Social moderator* can **delete** other users’ comments.
   2. If logged as *Social moderator* can **recruit** others *Social moderator*s.
9. *Users* that can **log in** as M*oderator* can do all operation of a *Social moderator* plus:
   1. If logged a *Moderator* can **add/delete/modify** a *Movie/Projection*.
   2. If logged as *Moderator* can **recruit** other *Moderator*s
10. *Users* that can **log in** as*Admins* can do all operation of a M*oderator* plus:
    1. If logged an *Admin* can **delete** another user’s account.
    2. 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/statistics:

1. Chosen a temporal interval, the application shows the mean value of movie rating grouped by genre/Director/Actor.
2. 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: .

If we are considering user activities: *c* stands for the number of comments that the user does, *f* for the number of favorites given by the user, and *v* for the number of page requested by the user.

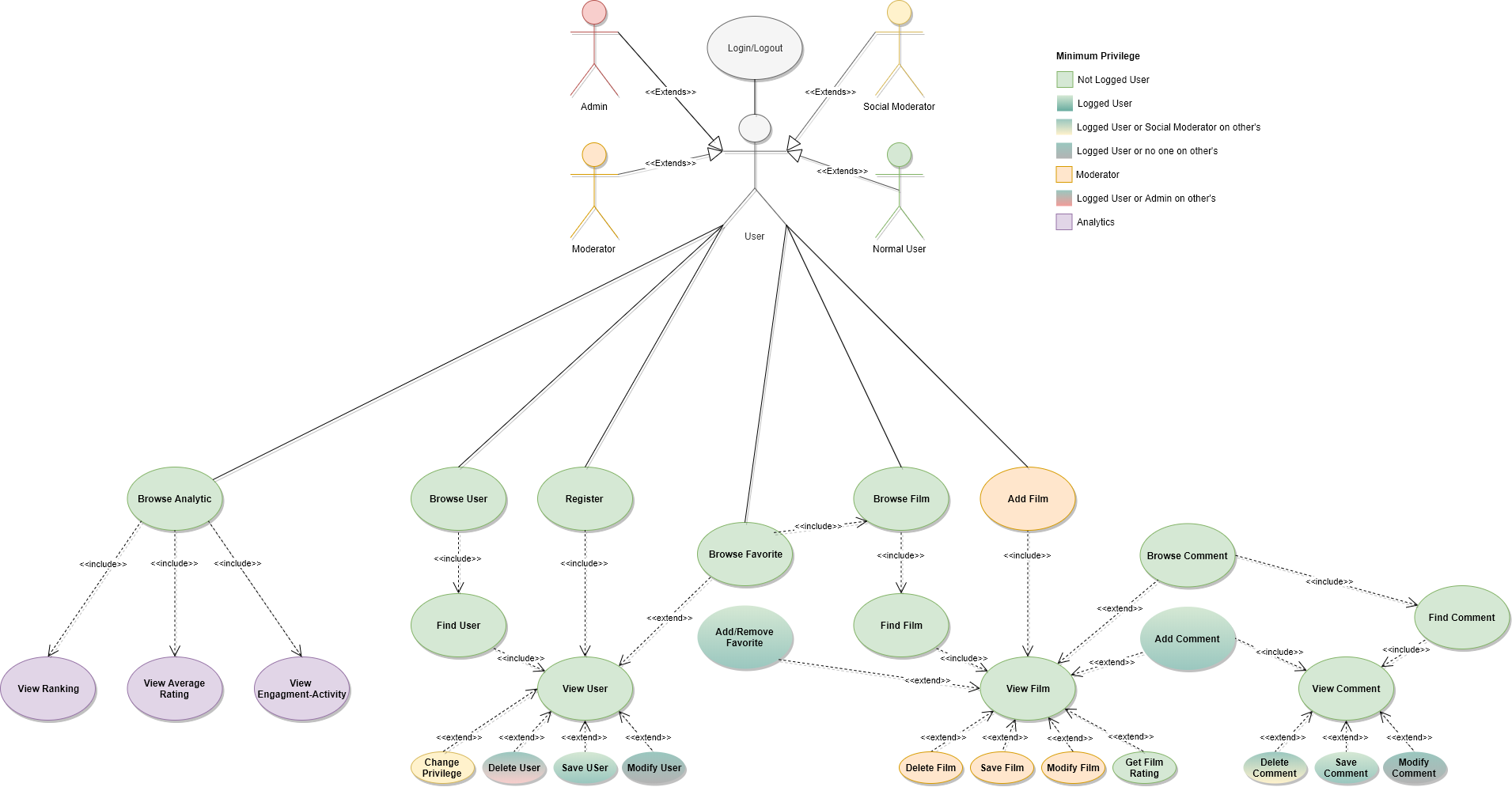
If, insead, we are considering film activities: *c* stands for the number of comments received by the film, *f* for the number of favorites, and *v* for the number of visits on the film page.

The above value il calculated for all films/users and it is used to rank them.

1. Chosen a film (or a user) and two years (one for the beginning and one for the end of a period), the application shows the engagement of the chosen film/user by year, in the interval of time specified before. Moreover, the application will show also the composition of the value of the engagement for a chosen year. The formula use to calculate the engagement changes depending on the subject.  
   This is the formula for the engagement of a film . It is basically a weighted sum where E is the 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.

The formula for the engagement of a user is . It is basically a weighted sum where E is the engagement, *n* is the number of favourites given by the user, *c* is the number of comments written, and *v* is the number of pages visited.

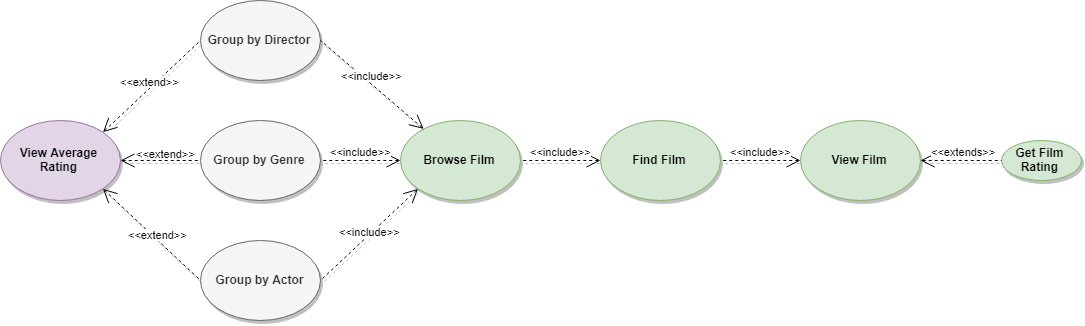
## Use Cases



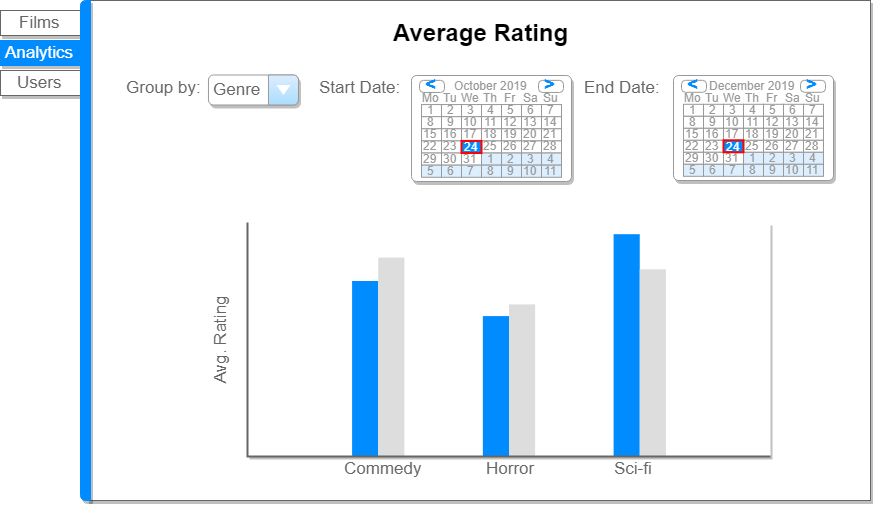
## ANALYTICS USE CASES And Mockups

Let us see the use cases of the analytics reported in the diagram above.

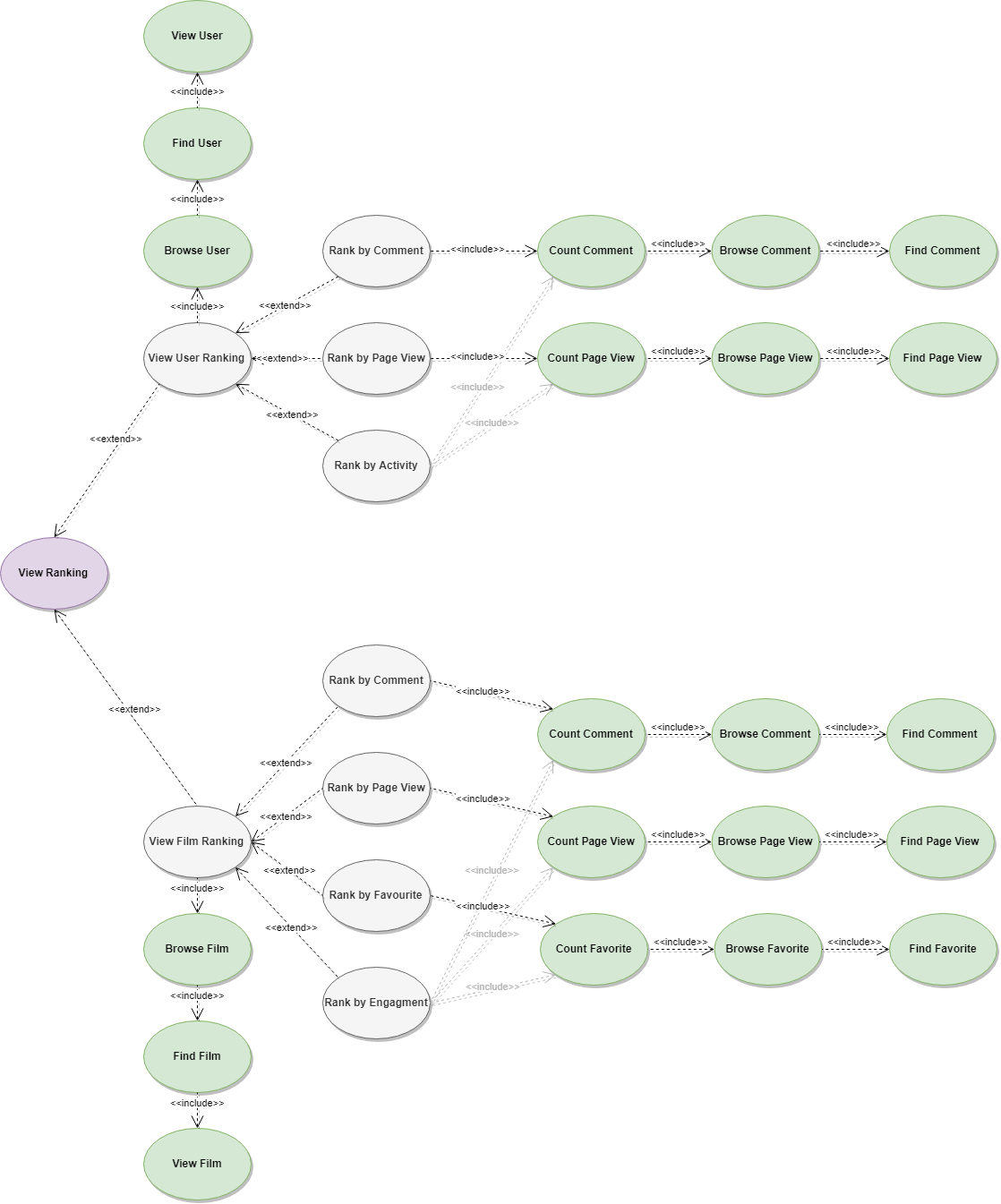
### View Average Rating



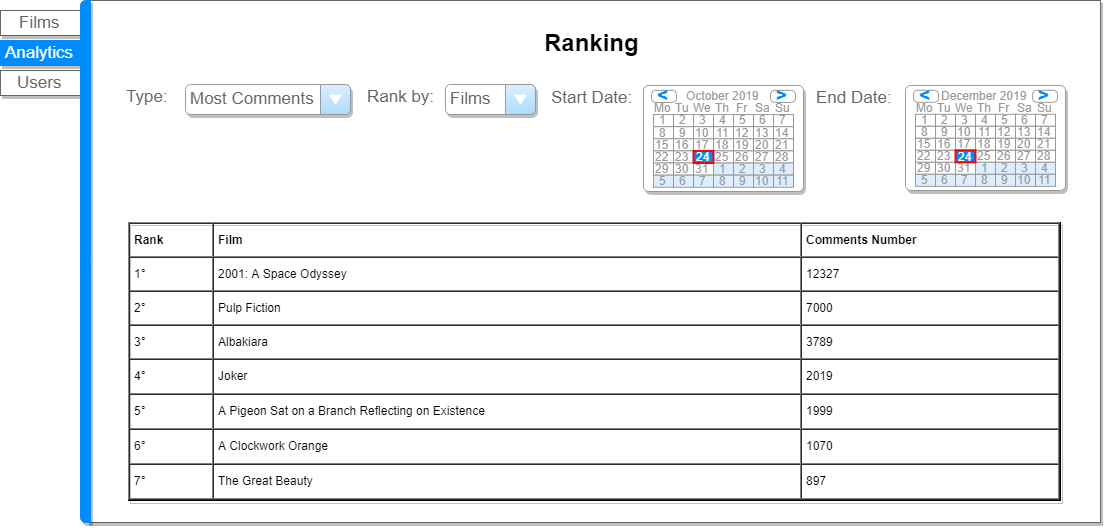
#### Mockup



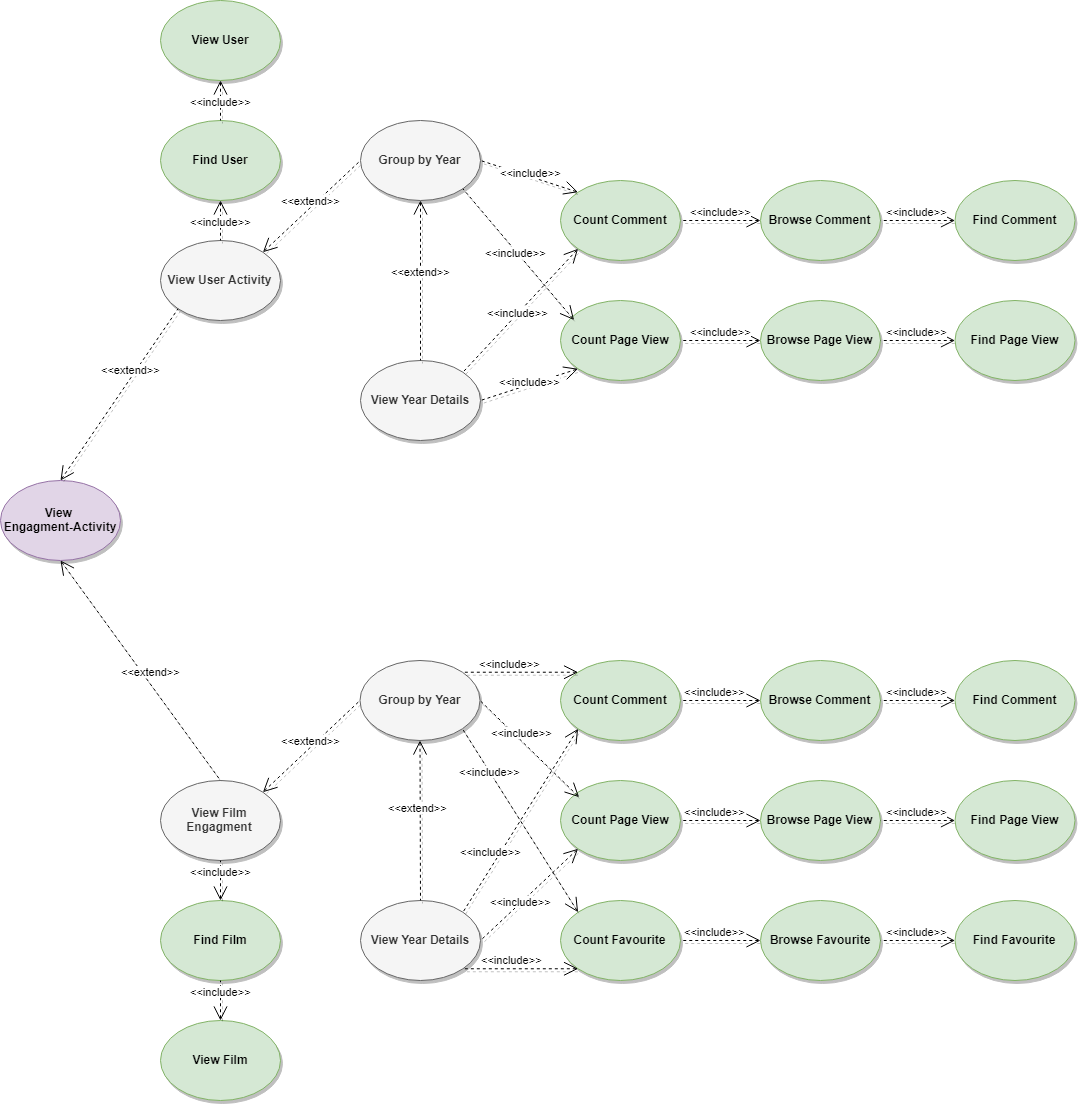
### View Ranking



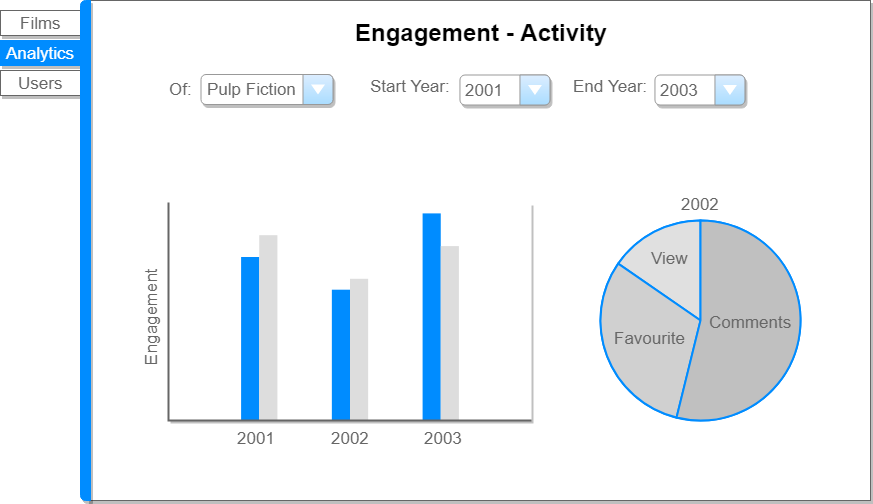
#### Mockup



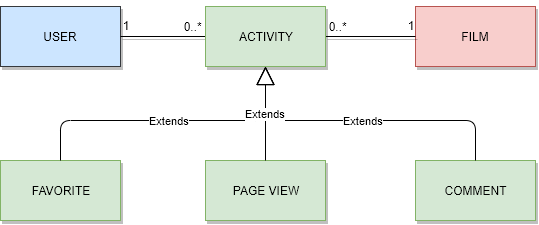
### View Engagement Activity



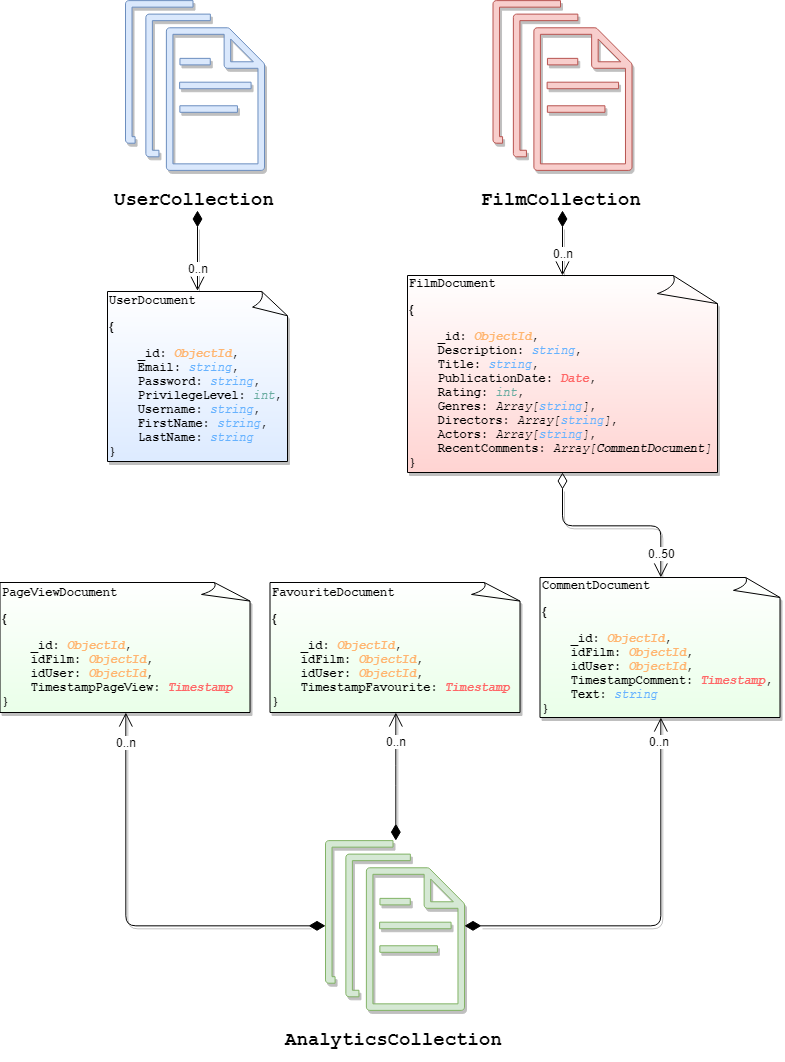
#### Mockup



## Analysis Classes



## Data Model



## 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 fosr 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.

Immagine che contiene screenshot

Descrizione generata automaticamente

## Software classes

### Entities

In the next pages we will describe all classes presents in the application.

Let’s start with the main entities, but since they are self explanatory we will not see them in details.

Immagine che contiene testo, mappa

Descrizione generata automaticamente

These classes have all the get and the setter for all private attribute. Moreover, they have a constructor that get the document retrieved from mongodb, and build the object starting from that.

### DB-Manager

All the menagers are implementented following the software design pattern of **singleton pattern** which restricts the instantiation of a manager to one "single" instance, Also the EntityFactoryManager used by Hibernate and managed in the DBManager class it follows this design pattern.

Now it is explained how the structure of the DBManager work.

* **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 trought the public members of the class, it automatically inizialize 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 the mongo Database.
* **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 get an extra two parameter, that are two integer: limit and skip. These two integer are used to get a pagination, retrieving always “limit” document, and then skipping “skip” document for the next page.

All function 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 user*nameFilter* 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.

Here, too, some functions take two additional parameters, limit and skip, for the same reason of UserManager.

All function 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 wich defines the basic operation that any engage manager should have (independent from the techonology)
* **EngageManager** implements **EngageManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the engages.

Some function may need an explaination so:

* + **create(**User user, Film film, EngageType type**)** wich 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**)** wich delete all the engages, releted to an entity e.g. delete all the views, favourites and comments, done by an User. In this case the function managed too the comments.
* **CommentManagerDatabaseInterface** it’s the interface wich defines the basic operation that any comment manager should have (independent from the techonology)
* **CommentManager** implements **CommentManagerDatabaseInterface** and extends **EngageManager** and is in charge of manage all CRUD operation with the database for the comment. It extend EngageManager because a comment is a type of engage.
* **AnalyticsManagerDatabaseInterface** it’s the interface wich defines the basic operation that any analytic manager should have (independent from the techonology)
* **AnalyticsManager** implements **AnalyticsManagerDatabaseInterface** and is in charge to perform the analytics described before in the documentation. It is better described in the next section.

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**)** wich perform the seconds.
  + **rankingAnalytics(**Date startDate, Date endDate, RankingType typeOfRanking**)** which perform the third.

All the other funtion are to support these three function.

Immagine che contiene testo, screenshot

Descrizione generata automaticamente

### Analytics entities

Now are described the entities used in **AnalyticsManager** to support the analytics.

There are three **AnalyticsEntities**, and are associated with the related analytic, so:

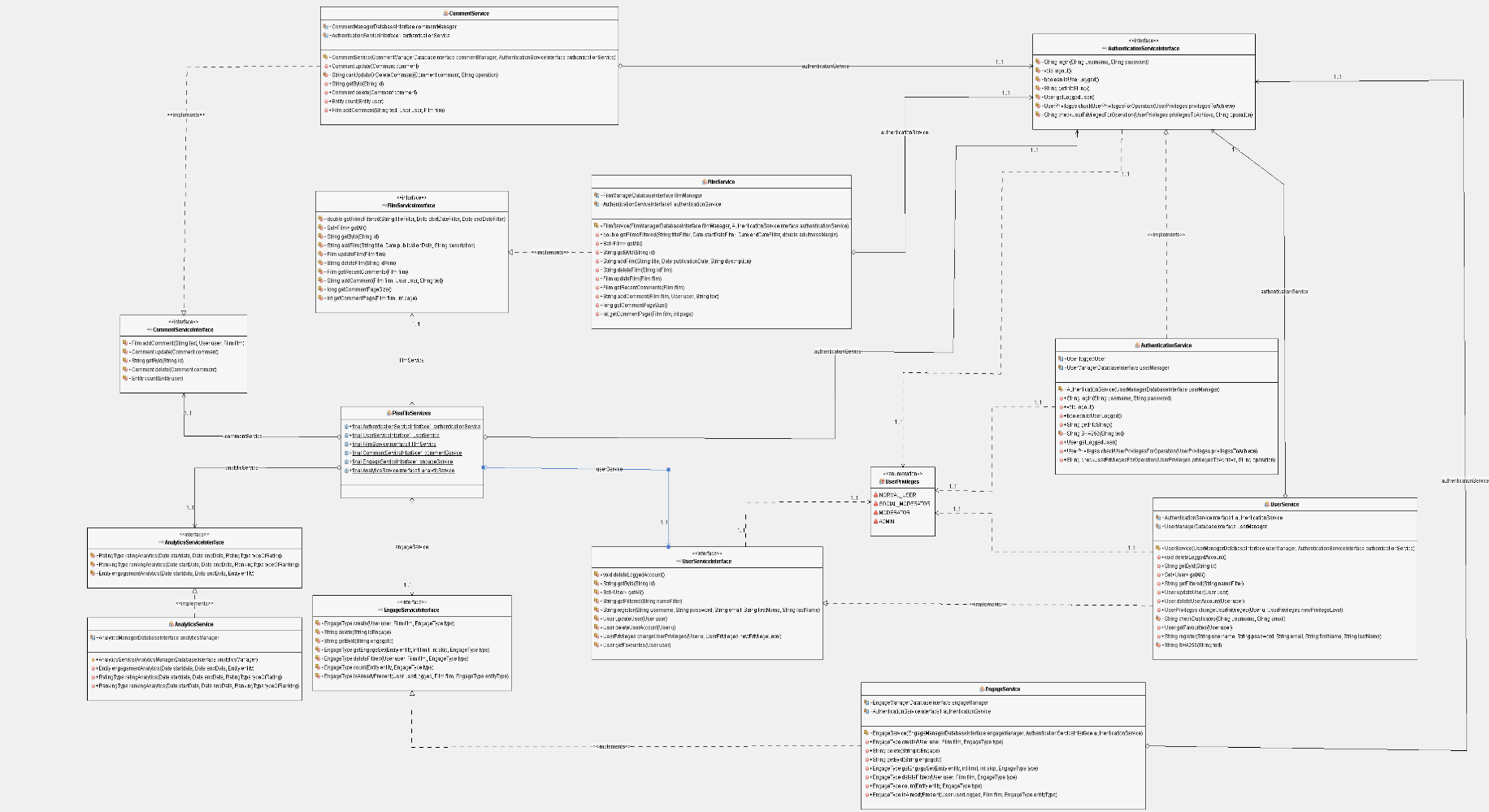
* **AverageRatingResul** are associated with the first Analytic.
* **EngageResult** with the second analytic.
* **RankingResult** with the third analytic.

Immagine che contiene screenshot, testo

Descrizione generata automaticamente

### PISAFLIX-Services

Let’s see now the structure of PisaFlixServices



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 loggedUser
  + 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 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 datawith the DB andan **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 an 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 an 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 an 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 datawith the DB and **AuthenticationServiceInterface** to ensure that we have the right privileges depending by the operation that we want 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 datawith 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 datawith 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 engagementManager 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 a **AnalyticsManagerDatabaseInterface** to exchange datawith 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