

Pisa University  
  
  
TASK 3  
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

**“*PisaFlix 3.0” project documentation***  
academic year 2019-2020  
  
  
  
  
stefano petrocchi, Andrea Tubak, Francesco Ronchieri, Alessandro Madonna

Summary5

[Design Document 3](#_Toc34222666)

[Description 3](#_Toc34222667)

[Requirements 3](#_Toc34222668)

[Main Actors 3](#_Toc34222669)

[Functional 3](#_Toc34222670)

[Non-Functional 3](#_Toc34222671)

[Use Cases 4](#_Toc34222672)

[Suggestions 4](#_Toc34222673)

[Analysis Classes 5](#_Toc34222674)

[Data Model 5](#_Toc34222675)

[Example 5](#_Toc34222676)

[Architecture 6](#_Toc34222677)

[Interface Design Pattern 6](#_Toc34222678)

[Software Classes 7](#_Toc34222679)

[Entities 7](#_Toc34222680)

[DBManager 7](#_Toc34222681)

# Design Document

## Description

PisaFlix 3.0 is a social network oriented to the discussion of films. A User can visit the profiles of other users and see the pages related to films. In those pages, the User, will find either all the post written by the user or the most recent posts which tag the film. Lastly, it is possible to follow other users or films in order to be informed about their posts and receive suggestions on the browsing pages.

## 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 interaction*.
* **Social Moderator**: a trusted user with the possibility to *moderate* the posts.
* **Moderator**: a verified user with the possibility to add and *modify* elements in the application, like film pages.
* **Admin:** an *administrator* of the application, with possibility of a *complete interaction*.

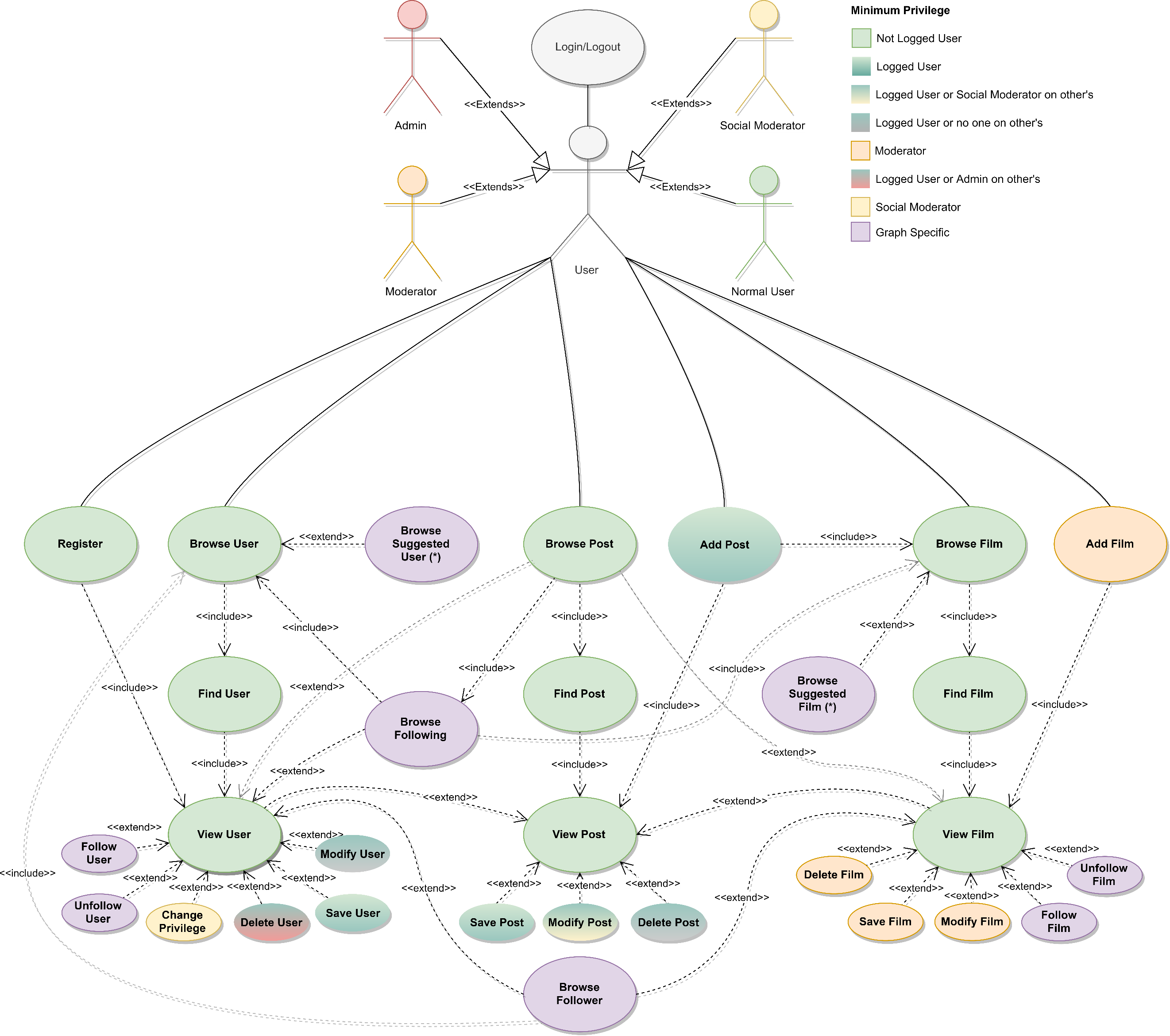
Functional

1. *Users* can **view** the list of **Movies** available on the platform.
2. *Users* can **view** the posts about a specific *Movie*.
3. *Users can* ***view*** *the list of* ***Users*** *on the platform.*
4. *Users can* ***view*** *the posts of a specific User.*
   1. *Users* can **register** an account on the platform.
5. *Users* can **log in** as *Normal users* on the platform in order to do some other operations:
   1. If logged a *Normal user* can **follow/unfollow** a *Movie*.
   2. If logged a *Normal user* can **follow/unfollow** a User.
   3. If logged a *Normal user* can **write** a Post on a *Movie*.
   4. If logged a *Normal user* can ***view***Post of his following *Movies* and *Users*.
   5. If logged a *Normal user* can ***view***suggestions on *Movies* to follow.
   6. If logged a *Normal user* can ***view***suggestions on *Users* to follow.
   7. If logged a *Normal user* can **write** a ***Post.***
   8. If logged a *Normal user* can **modify** his ***Posts***.
   9. A *Normal user* can **modify/delete** his account.
6. *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.
7. *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*.
   2. If logged as *Moderator* can **recruit** other *Moderator*s
8. *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 focus of the application is the *quality* of the information provided to the users.
2. The application needs to be **consistent**, in order to provide correct information to all the users.
3. The transactions must be **monotonic:** every user must see the last version of the data and modifications are done in the same order in which they are committed.
4. The application needs to be *usable* and *enjoyable* for the user, therefore the system needs **limited response times**.
5. The *password* must be protected and stored *encrypted* for privacy issues.

## Use Cases



### Suggestions

The suggestions are shown only if the user is logged in. The suggestions can be found in the initial pages of the *browsers*, the page is filled with the suggestions from the highest priority to the lowest until exhaustion. If the suggestions are not enough to fill the page, the most recent films\users, that have not been already suggested, are chosen to complete it.

#### Suggested Films

There are three levels of suggestions with different priorities:

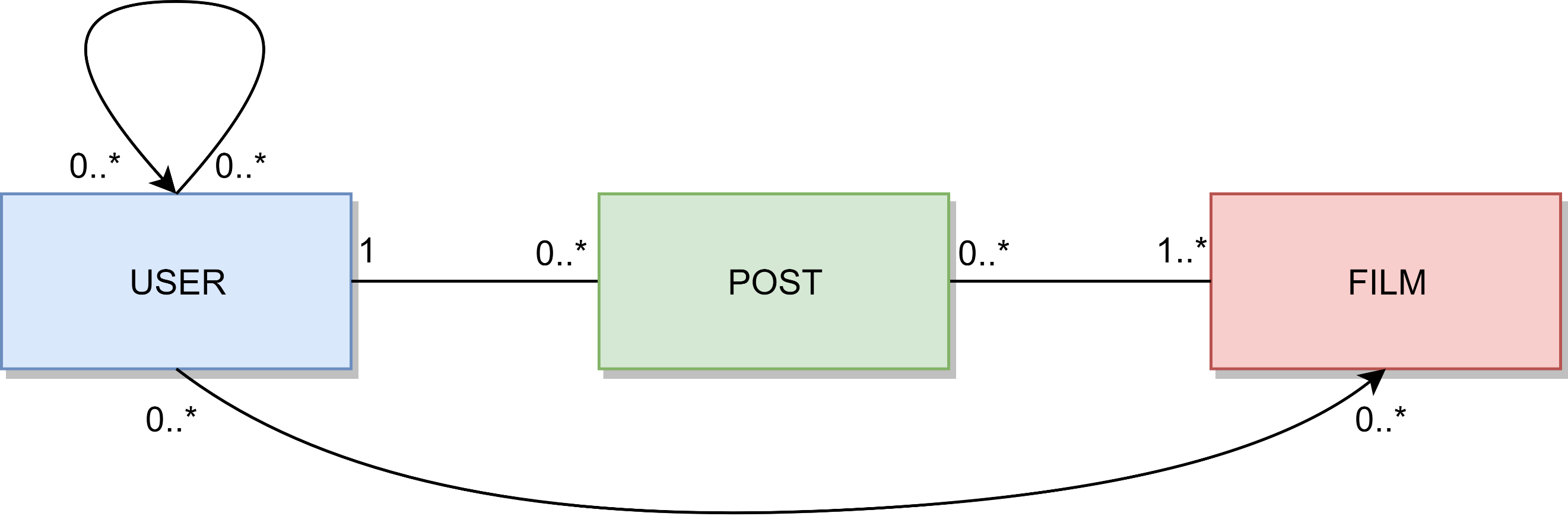
* **Very Suggested:** They have the highest priority, given a user **U1** if **U1** is following user **U2** and **U2** is following a movie **F** and has also posted on **F**, then **F** is *very suggested* to **U1**.
* **Suggested:** They have the second priority level, if a user **U1** is following user **U2** and **U2** is following a film **F**, then **F** is *suggested* to **U1**.
* **Commented by Friend:** They have the lowest priority level, if a user **U1** follows a user **U2** who posted on a movie **F**, then **F** is suggested as "*commented by a friend*" at **U1**.

#### Suggested Users

There are two levels of suggestions with different priorities:

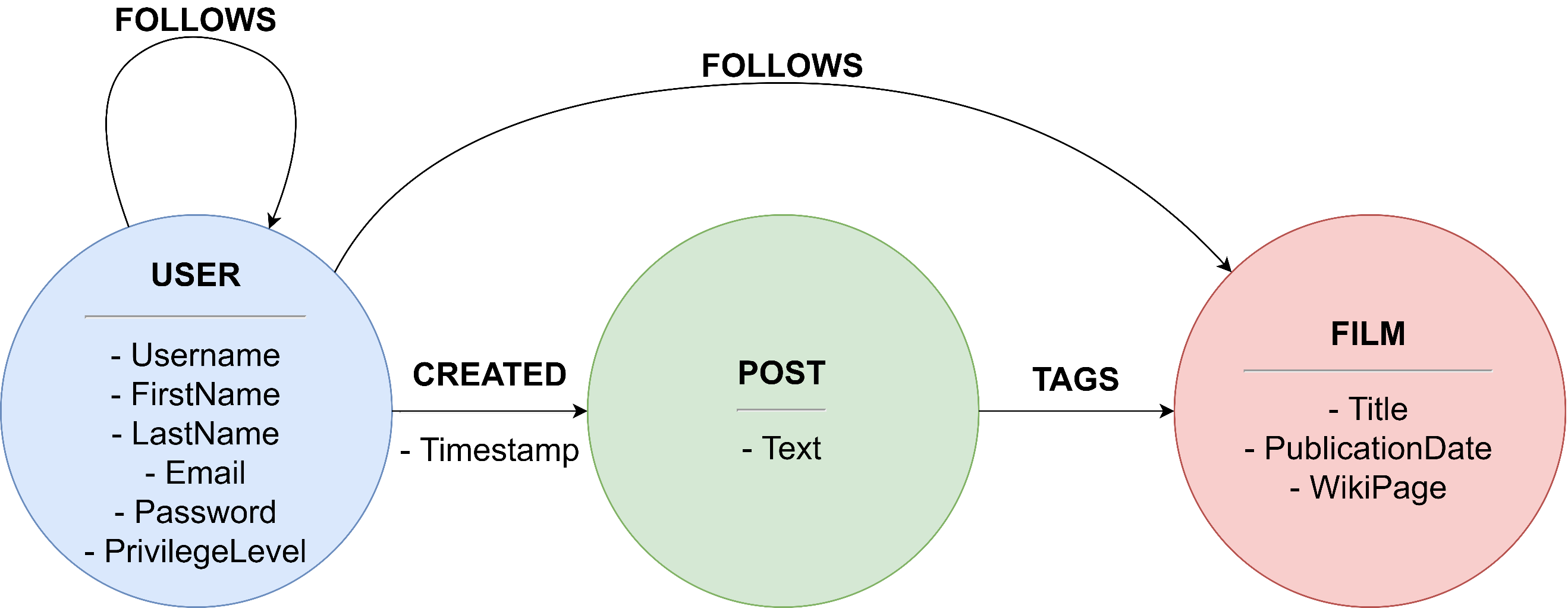
* **Very Suggested:** They have the highest priority, given a user **U1** if **U1** is following user **U2** and **U2** is following user **U3**, then **U3** is *very suggested* to **U1**.
* **Suggested:** They have the lowest priority level, if a user **U1** is following user **U2** and **U2** is following a user **U3** and **U3** is following a user **U4**, then **U4** is *suggested* to **U1**.

## Analysis Classes



## Data Model

We have basically three entities, User, Film, and Post. The relation between Users is of type “follows”, such as the relation between User and Film. The relation between User and Post is of type “create” and contains a property Timestamp. The relation between Post and Film, is of type “Tags”.



### Example

|  |
| --- |
|  |
| User Ivanichev:  {Email: [eivanichevcb@intel.com](mailto:eivanichevcb@intel.com), FirstName: Elicia, LastName: Ivanichev, Username: eivanichevcb, PrivilegeLevel: 0, Password: 23847207fb18f5d4c7f12a1dd8c6938b1254217ed695183a65a2ebd5c602477e}  User Swaden being followed by Ivanichev  {Email: mswaden3e@people.com.cn, FirstName: Melly, LastName: Swaden, Username: mswaden3e, PrivilegeLevel: 0, Password: 1639b647d3274638a489902e2b5de5f607000d3b285e22196152f18b7baec446}  The CREATED relation has a property Timestamp:  {Timestamp:"2020-02-25T16:11:26.099000000Z"}  The Post created by Swaden:  {Text: This is the first Michael Vartan movie i've seen…}  The movie tagged by the post above:  {Title: Jagadeka Veerudu Athiloka Sundari, PublicationDate: 1990, WikiPage: https://en.wikipedia.org/wiki/Jagadeka\_Veerudu\_Athiloka\_Sundari}  The user who FOLLOWS the movie above:  {Email: [gstandley7v@cafepress.com](mailto:gstandley7v@cafepress.com), LastName: Standley, Username: gstandley7v, PrivilegeLevel: 0, FirstName: Gert, Password: 43cfb25c46e3f319c4b1c81e4bccc9d5668251fad732e744a8a087cab152a3fc} |

## Architecture

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

The client Application is made in *Java* using **JavaFX framework** for the *front-end* and the **Neo4j 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.

Immagine che contiene screenshot

Descrizione generata automaticamente

## Software Classes

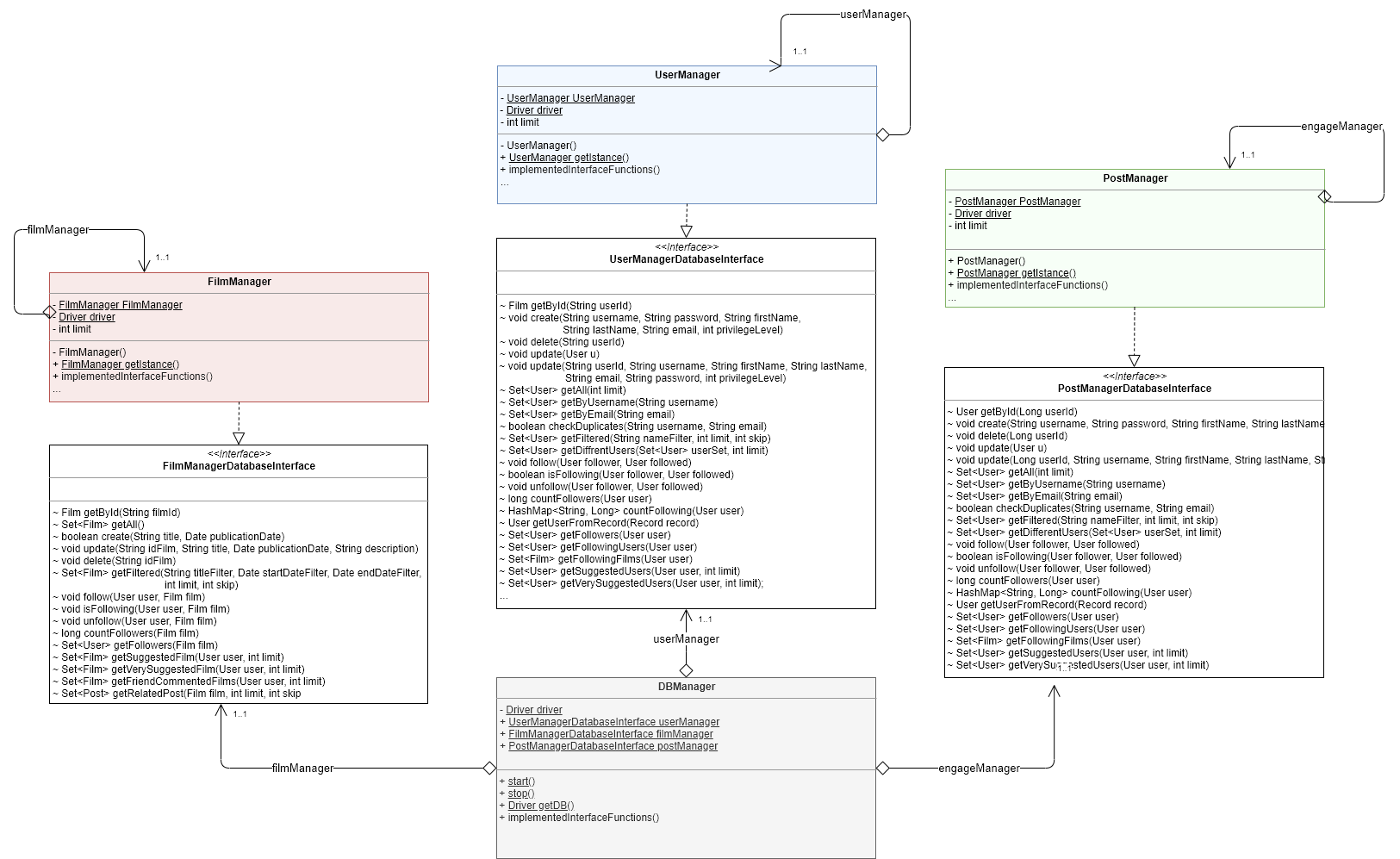
### Entities

Diagram of the **classes**:

Immagine che contiene screenshot, mappa

Descrizione generata automaticamente

### 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 our Graph database. Each of the following Managers have their own utility method called **getEntityFromRecord**(Record record) where Entity should be replaced with the actual name of the entity managed. This method is used to convert the records retrieved by the Graph database into an Entity object.
* **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 managing all *CRUD* operation with the database for the users. Some functions get an extra two parameter, that are two integers: limit and skip. These two integers are used to realize pagination, retrieving always “limit” document, and then skipping “skip” document for the next page. All functions are self-explanatory by the name except for:

* + **getDifferentUsers**(Set<User> userSet, int limit) which searches for users that aren’t already present in the userSet passed as an argument. The int limit specifies how many users the function should retrieve.
  + **getSuggestedUsers**(User user, int limit) and **getVerySuggestedUsers**(User user, int limit) this functions implement the retreival of the users that should be “suggested” or “very suggested” to the user passed as an argument. The criteria has been explained in the paragraph “suggested Users”.
* **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 managing all *CRUD* operation with the database for the movies.

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

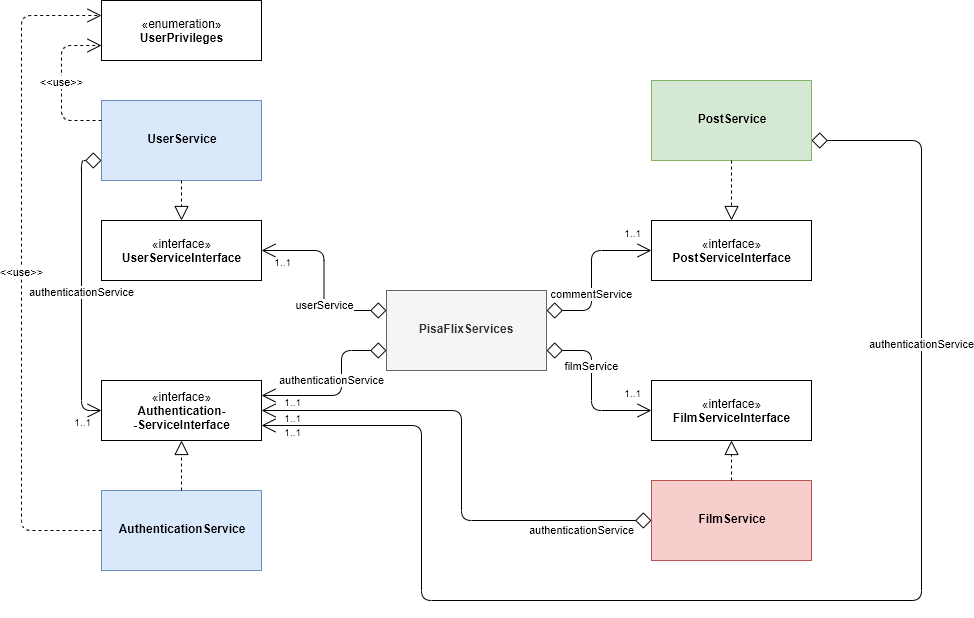
All functions are self-explanatory by the name except for:

* + **getSuggestedFilms**(User user, int limit) and **getVerySuggestedFilms**(User user, int limit) this functions implement the retreival of the films that should be “suggested” or “very suggested” to the user passed as an argument. The criteria has been explained in the paragraph “suggested Films”.
* **PostManagerDatabaseInterface** it’s the interface which defines the basic operation that any post manager should have (independent from the technology)

**PostManager** implements *PostManagerDatabaseInterface* and is in charge of manage all CRUD *operation* with the database for the posts.  
All functions are self-explanatory by the name except for:

* + **getPostFollowed**(User user, int currentPageIndex) It searches for all the posts coming from two sources:
    - the ones which has been written by a user who is followed by the user passed as an argument
    - the ones which tagged a film followed by the user passed as an argument

### Services



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 managers 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 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 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 **follow**(User follower, User followed) stores the follows relation between “follower” and “followed”
  + boolean **isFollowing**(User follower, User followed) returns true or false whether the “follower” is following the “followed”
  + void **unfollow**(User follower, User followed) deletes the follows relation between the two users passed
  + long **countFollowers**(User user) returns the number of followers of the user passed
  + long **countFollowingUsers**(User user) returns the number of users followed by the user passed
  + long **countFollowingFilms**(User user) returns the number of films followed by the user passed
  + long **countTotalFollowing**(User user) returns the number of users and films followed by the user passed
  + Set<User> **getFollowers**(User user) returns the set of users who follow the user passed
  + Set<User> **getFollowingUsers**(User user) returns the set of users followed by the user passed
  + Set<Film> **getFollowingFilms**(User user) returns the set of films followed by the user passed
  + Set<User> **getSuggestedUsers**(User user, int limit) returns the set of users suggested to the user passed; limit specifies how many users we want
  + Set<User> **getVerySuggestedUsers**(User user, int limit) returns the set of very suggested users to the user passed; limit specifies how many users we want
  + Set<User> **getMixedUsers**(User user) returns a set of users that is composed by very suggested users, suggested users, and then a selection of other users. The set is populated up to a certain level, and then returned; so whenever the set is full it will be returned without including users of the subsequent categories.
  + Set<User> **getDifferentUsers**(Set<User> userSet, int limit) returns a set of users who are not present in the userSet passed
* **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 managing 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 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 **follow**(Film film, User user) stores the follow relation between user and film
  + boolean **isFollowing**(Film film, User user) return true of false whether the user is following the film or not
  + void **unfollow**(Film film, User user) removes the follow relation between user and film
  + long **countFollowers**(Film film) returns the number of followers of film
  + Set<User> **getFollowers**(Film film) returns the set o users who follow the film
  + Set<Film> **getSuggestedFilms**(User user, int limit) returns the set of films suggested to the user passed
  + Set<Film> **getVerySuggestedFilms**(User user, int limit) returns the set of very suggested films to the user passed
  + Set<Film> **getFriendCommentedFilms**(User user, int limit) returns the set of films on which at least one of the users followed by “user” has commented on, provided that that film isn’t already followed by us or by the user who commented on in.
  + Set<Film> **getMixSuggestedRecent**(User user) returns a set of films that is composed by very suggested films, suggested films, films commented by a friend, and then a selection of other films. The set is populated up to a certain level, and then returned; so whenever the set is full it will be returned without including films of the subsequent categories.
  + Set<Post> **getRelatedPosts**(Film film, int page) returns the set of posts which tag the film passed
  + int **getPostPageSize**() returns the number of posts per page to be displayed
  + Set<Film> **getDifferentFilms**(Set<Film> filmSet, int limit) returns a set of films which are not present on the filmset passed
* **PostServiceInterface** it’s the interface which defines the basic operation that any post service should have (independent from the technology)
  + we will see the methods in detail in the class which implement it
* **PostService** implements *PostServiceInterface* and is in charge of manage all operations that are specific for the posts, in order to work properly it uses the *PostManagerDatabaseInterface*to exchange datawith the DB, and *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
  + Post **getById**(Long idPost) returns the post with id equal to the one passed
  + void **create**(String text, User user, Set<Film> films) Stores a post with the field passed
  + void **delete**(Long idPost) deletes the post with id equal to the one passed
  + void **update**(Long idPost, String text) updates the text of the post with id equal to the one passed
  + int **count**(Entity entity) the entity passed can either be a film or a user. Depending on the case this function returns the number of posts which tag a film, or the number of posts created by a user
  + Set<Post> **getPostFollowed**(User user, int currentPageIndex) returns the set of posts written on a film followed by the user passed or written by a user followed by the user passed
  + int **countPostFollowed**(User user) returns the number of posts retrieved in the same way as **getPostFollowed**
  + Set<Post> **getUserPosts**(User user, int currentPageIndex) returns the set of posts written by the user passed
  + int **countUserPosts**(User user) returns the number of posts written by the user passed
  + int **getHomePostPerPageLimit**() returns the number of posts to be displayed in a page

## Relevant Queries

### Count Following

This query is similar both in the case of counting the FOLLOWS relations of a film and a user. The same goes for counting the number of follower of a film or user.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Domain-specific | Graph-centric | | How many users/films are followed by specific user? | How many outgoing edges does a vertex has? |  1. MATCH (u1:User)-[:FOLLOWS]->(u2:User), (u1:User)-[:FOLLOWS]->(f:Film) 2. WHERE ID(u1) = **$userId** 3. RETURN count(DISTINCT u2) AS followingUsers, count(DISTINCT f) AS followingFilms |

### Suggested Film or User

The types of suggestion have been described before.

It’s been reported the query to get the suggestion for the user.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Domain-specific | Graph-centric | | What users are suggested for a specific user? | What are the nodes, that have a distance of 2 hopes from a specific node? |  1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(:User)-[:FOLLOWS]->(u:User) 2. WHERE ID(u1) = **$userId** 3. AND NOT (u1)-[:FOLLOWS]->(u) 4. AND NOT (u2)-[:FOLLOWS]->(u) 5. RETURN u |
| |  |  | | --- | --- | | Domain-specific | Graph-centric | | What are the very suggested users for a specific user? | What are the nodes, that have a distance of 1 hopes from a specific node? |  1. MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u:User) 2. WHERE ID(u1) = **$userId** 3. AND NOT (u1)-[:FOLLOWS]->(u) 4. RETURN u |

