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

SUMMARY

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. You can follow films and users in order to view the posts that concern them on the home page. New users and movies will be continually suggested, based on your friends and their favourite movies. Finally, a mechanism of privileges guarantee the quality of the contents, uploaded only by trusted users, and the possibility of moderate the posts present within the application.

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

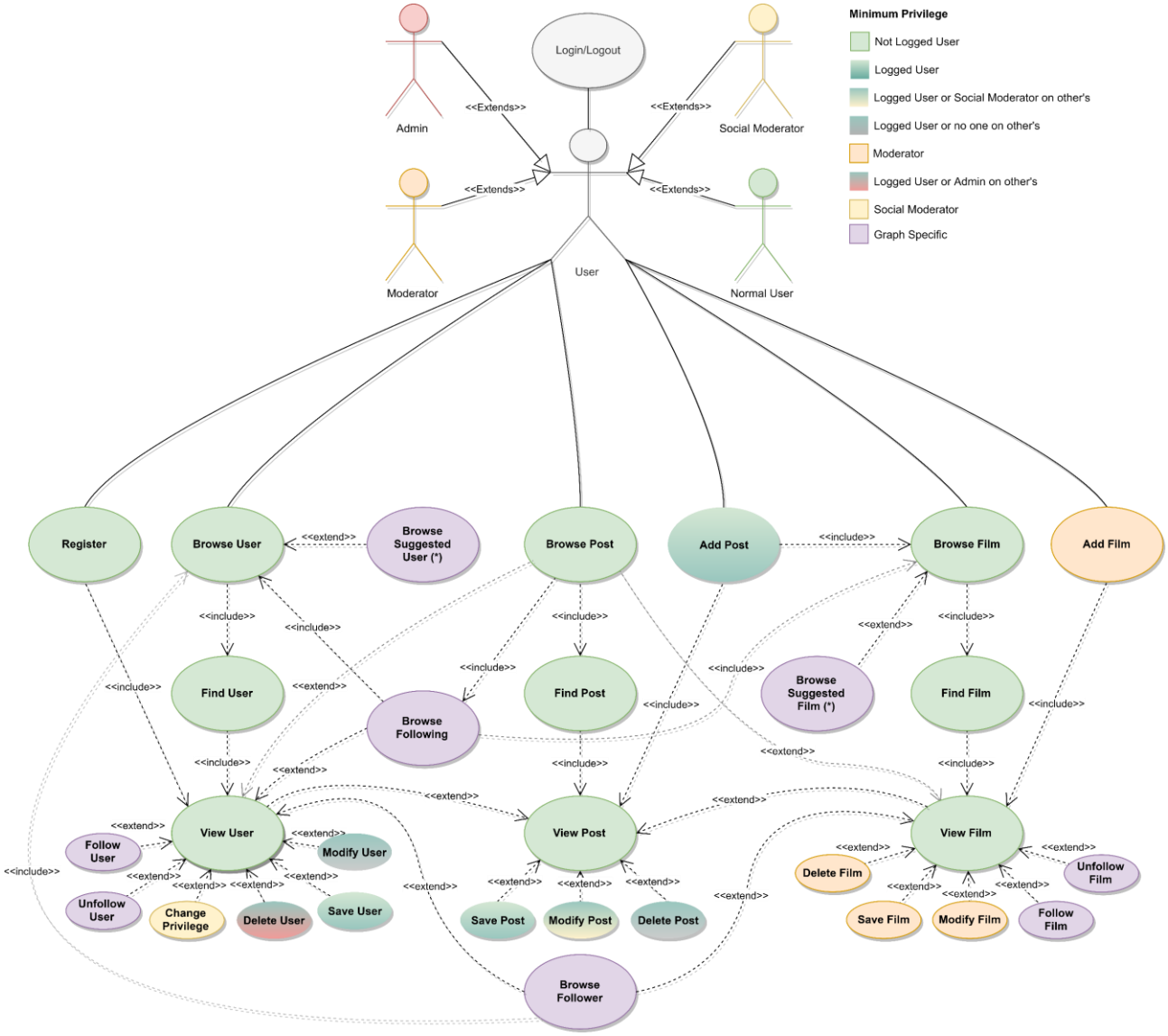
1. *Users* can **view** the list of **Movies** available on the platform.
2. *Users* can **view** the posts about a specific *Movie*.
3. *Users can **view** the list of **Users** on the platform.*
4. *Users can **view** the posts of a specific User.*
   a. *Users* can **register** an account on the platform.
5. *Users* can **log in** as *Normal users* on the platform in order to do some other operations:
   a. If logged a *Normal user* can **follow/unfollow** a *Movie*.
   b. If logged a *Normal user* can **follow/unfollow** a User.
   c. If logged a *Normal user* can **write** a Post on a *Movie*.
   d. If logged a *Normal user* can ***view*** Posts of his following *Movies* and *Users* on the *home page*.
   e. If logged a *Normal user* can ***view*** suggestions on *Movies* to follow.
   f. If logged a *Normal user* can ***view*** suggestions on *Users* to follow.
   g. If logged a *Normal user* can **write** a ***Post.***
   h. If logged a *Normal user* can **modify** his ***Posts***.
   i. 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:
   a. If logged as *Social moderator* can **delete** other users' comments.
   b. 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:
   a. If logged a *Moderator* can **add/delete/modify** a *Movie*.
   b. 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:

a. If logged an *Admin* can **delete** another user's account.
b. If logged as *Admin* can **recruit** other *Admins*.

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**.

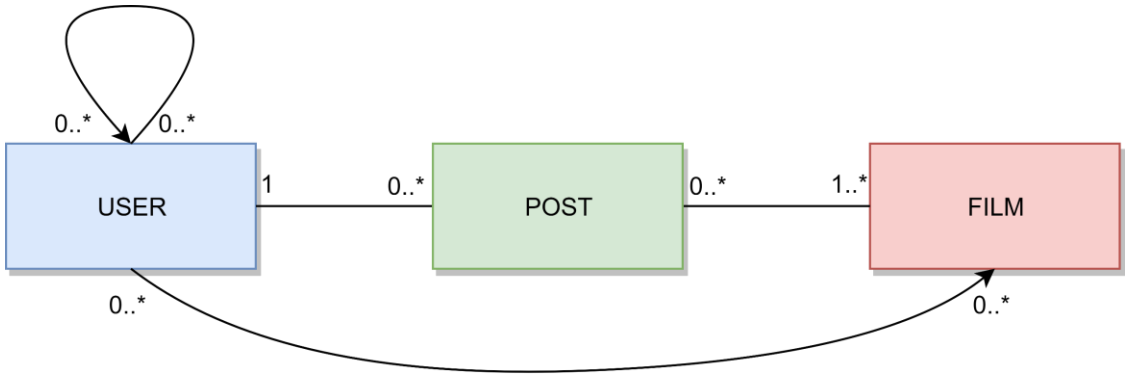
SUGGESTED USERS

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**.

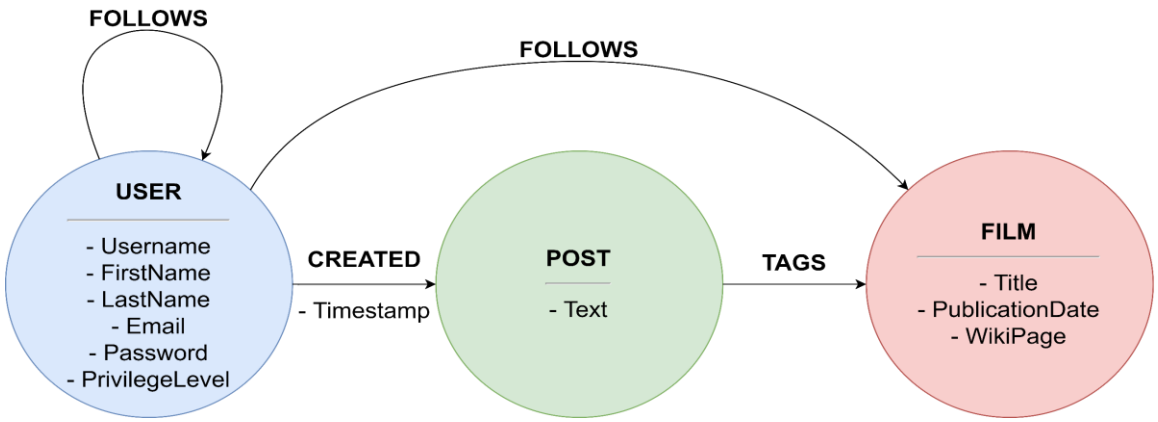
The description of the queries related to these suggestions is present in the Suggested Users and Movies section.

ANALYSIS CLASSES

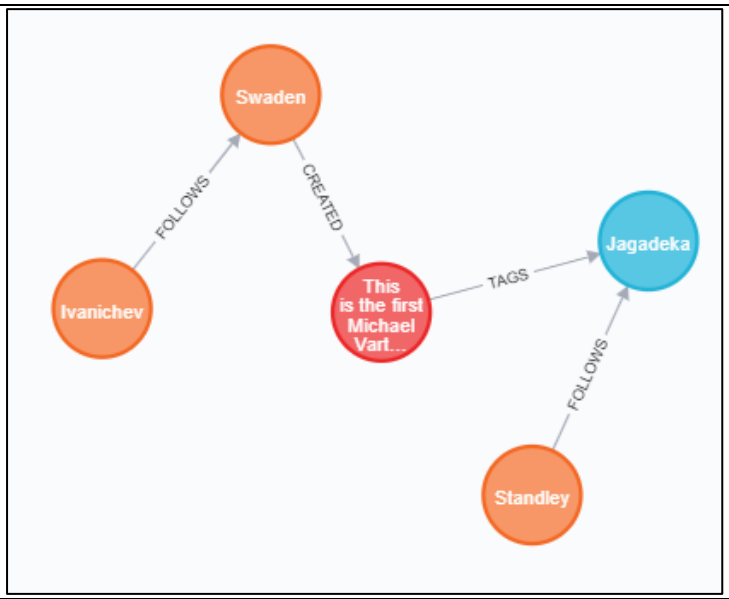


DATA MODEL

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, `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, 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

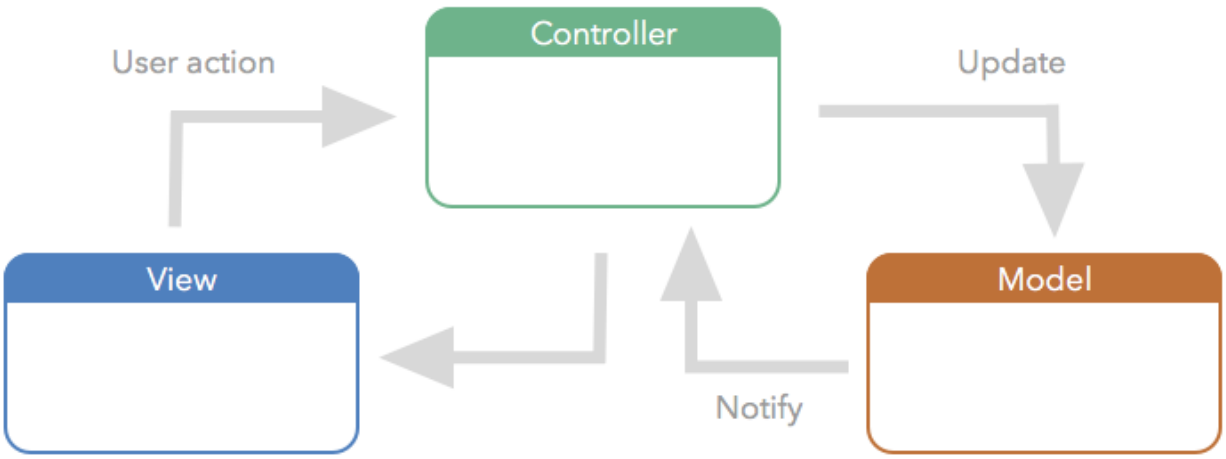
Services module represent the *model* and is the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages logic and rules of the application receiving inputs from the controller. The model is also responsible for managing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

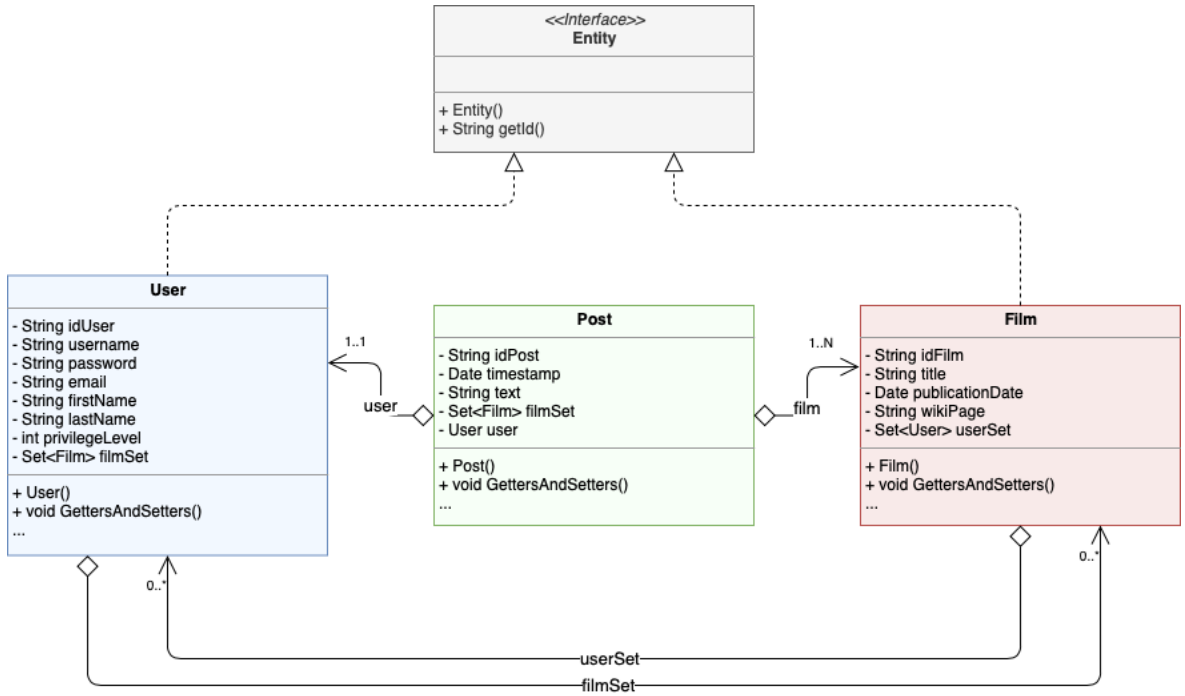
The **page controllers** are the *controller* of the application. They receive inputs from the *view* and converts them into commands for the *model* or *view* itself. Controllers can also validate inputs and data without the intervention of the *model*. Data is exchanged between *model* and *controller*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



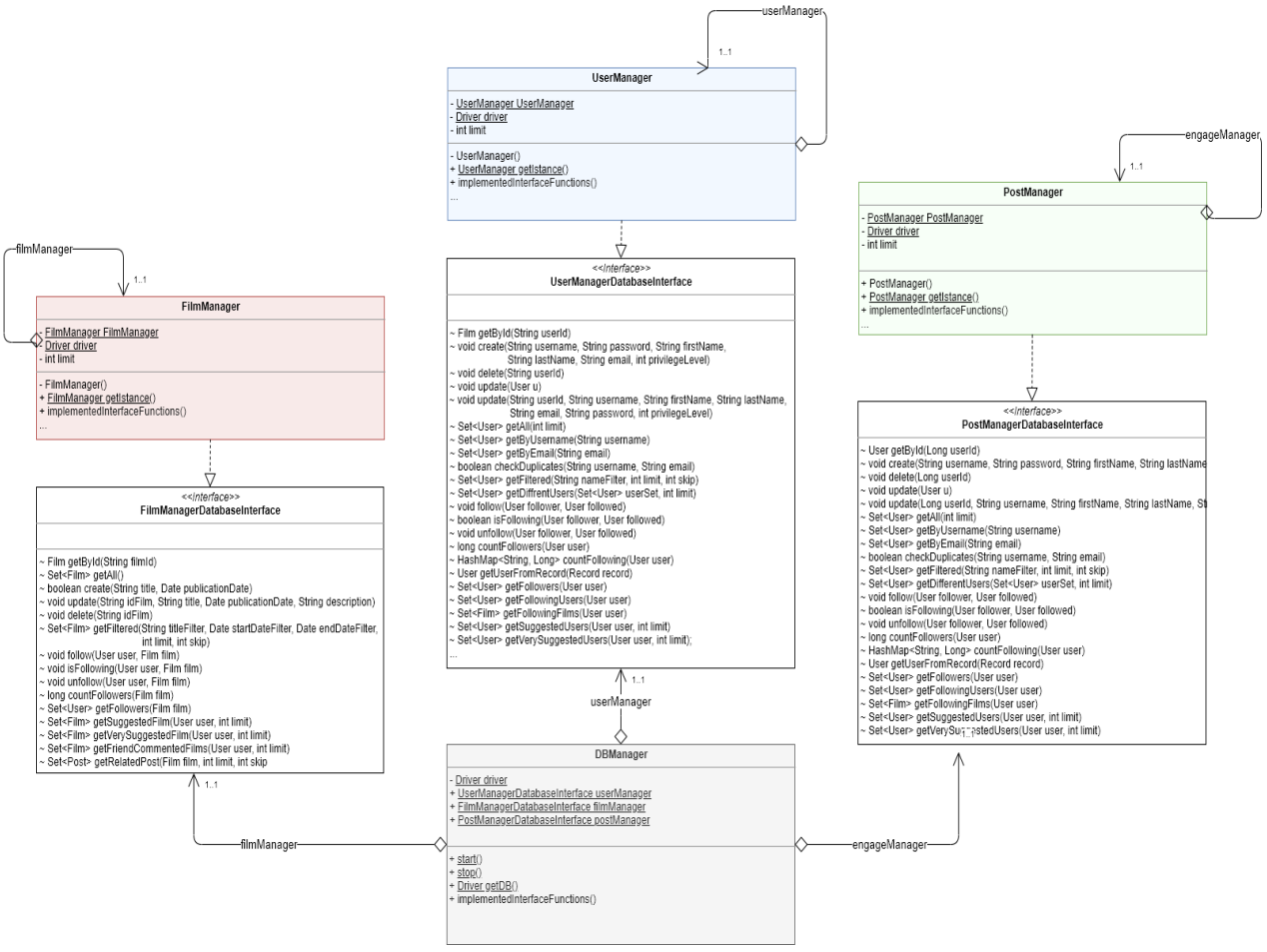
SOFTWARE CLASSES

ENTITIES

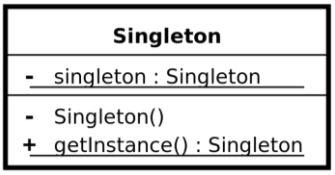
Diagram of the **classes**:



DBMANAGER



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

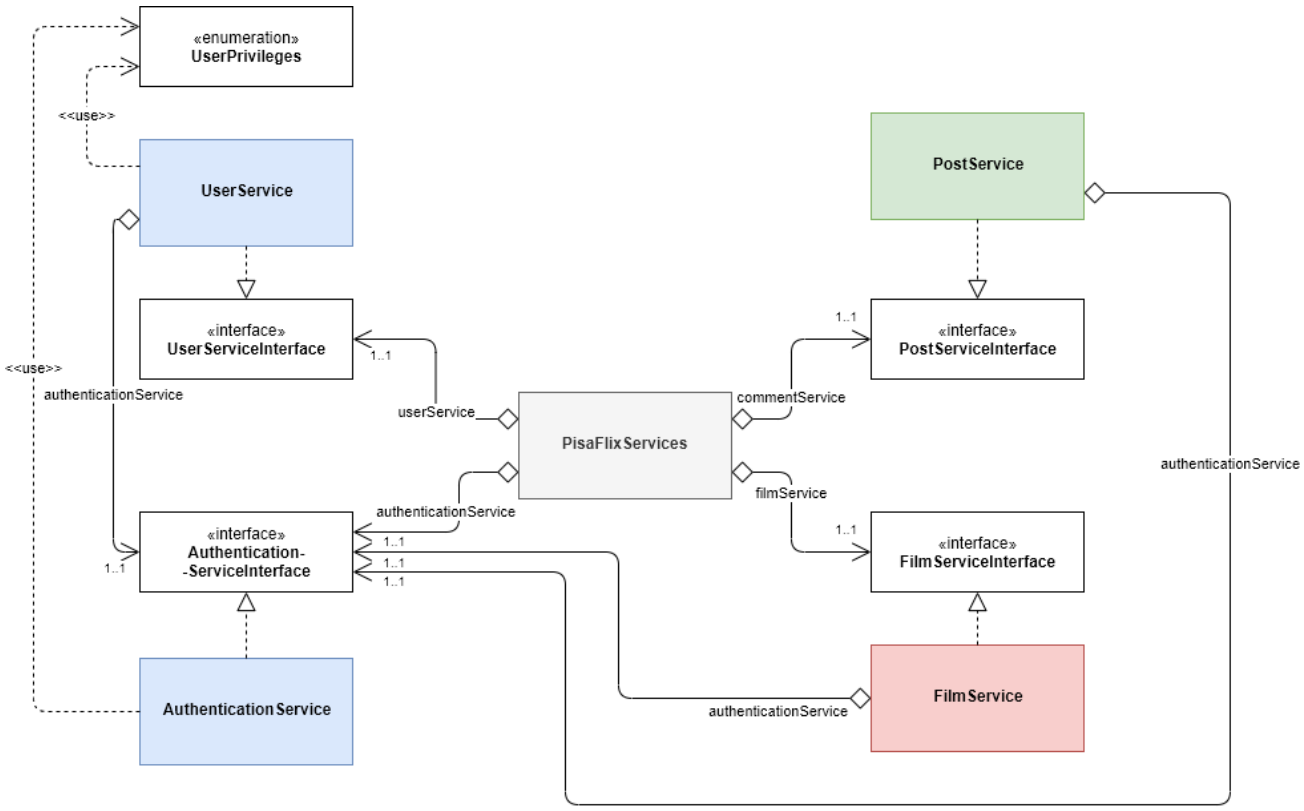


The main classes and functions are described below:

- **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 function implement the retrieval 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 function implement the retrieval 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 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 **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 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 **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 data with 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 count how many users and films are followed by a specific user and is used to fill up the stats of that user when is loaded his profile page:

Domain-specific	Graph-centric
How many users/films are followed by a specific user?	How many outgoing (Follows) edges does the (User) node take into consideration 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
```

For sake of brevity descriptions of further counting queries have been omitted, which have the same structure, including:

- How many Posts have been written by a specific User?
- How many Users follow a specific Film?
- How many Posts tag a specific Film?
- How many followers have a specific User?

These queries are all used to generate the statistics of the pages of the films and users and concern the counting of outgoing and ingoing relations from the node under consideration.

SUGGESTED USERS AND MOVIES

The behavior of this suggestions have been described in the chapters *Suggested Users* and *Suggested Films*.

This are the queries to get the *suggested* and *very suggested* users.

Domain-specific	Graph-centric
What are the *suggested* users for a specific user? (The users that are followed by a followed user of a followed users of the user take into consideration)	What are the nodes that have an exact distance of three (*Follow*) ingoing hopes from the node take into consideration and are of the same type (*User*)?

```
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? (The users that are followed by a followed user of the user take into consideration)	What are the nodes that have an exact distance of two (*Follow*) ingoing hopes from the node take into consideration and are of the same type (*User*)?

```
1.  MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(u:User)
2.  WHERE ID(u1) = $userId
```

```
3.  AND NOT (u1)-[:FOLLOWS]->(u)
4.  RETURN u
```

This are the queries to get the *commented by a friend*, *suggested* and *very suggested* films.

Domain-specific	Graph-centric
What films have been *commented by a friend* of a specific user? (The films that have a post that tags them, created by a followed user of the user taken into consideration)	What are the nodes that have an ingoing (*Tags*) edge from a node that has an ingoing (*Created*) edge from a (*User*) node that has an ingoing (*Follows*) edge from the node taken into consideration and don't have an ingoing (*Follow*) edge from the (*User*) nodes?

```
1.  MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:CREATED]->(:Post)-[:TAGS]->(f:Film)
2.  WHERE ID(u1) = $userId
3.  AND NOT (u1)-[:FOLLOWS]->(f)
4.  AND NOT (u2)-[:FOLLOWS]->(f)
5.  RETURN f
```

Domain-specific	Graph-centric
What are the *suggested* films for a specific user? (The films that are followed by a followed user of the user take into consideration)	What are the (*Film*) nodes that have an exact distance of two (*Follow*) ingoing hopes from the (*User*) node take into consideration?

```
1.  MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(f:Film)
2.  WHERE ID(u1) = $userId
3.  AND NOT (u1)-[:FOLLOWS]->(f)
4.  RETURN f
```

Domain-specific	Graph-centric
What are the *very suggested* films for a specific user? (The films that are followed and have a post that tags them by a followed user of the user take into consideration)	What are the (*Film*) nodes that have two ingoing (*Follow* and *Tags*) edges from a same (*User*) node that has an ingoing (*Follows*) edge from the (*User*) node take into consideration and no ingoing (*Follows*) edges from that node?

```
1.  MATCH (u1:User)-[:FOLLOWS]->(u2:User)-[:FOLLOWS]->(f:Film)
2.  WHERE ID(u1) = $userId
3.  AND NOT (u1)-[:FOLLOWS]->(f)
4.  AND (u2)-[:CREATED]->(:Post)-[:TAGS]->(f)
5.  RETURN f
```

FOLLOW AND UNFOLLOW

These are examples of two queries for movie follow and unfollow, analogous queries for the users have been omitted for brevity:

Domain-specific	Graph-centric
A specific user follows a specific film.	Given two nodes (*User* and *Film*), creates a (*Follows*) edge outgoing from the first (*User*) node and ingoing in the second (*Film*) node.

```
1.  MATCH (u:User),(f:Film)
```

```
2.  WHERE ID(u) = $userId
3.  AND ID(f) = $filmId
4.  CREATE (u)-[:FOLLOWS]->(f)
```

Domain-specific	Graph-centric
A specific user unfollows a specific film.	Given two nodes (*User* and *Film*), delete the (*Follows*) edge outgoing from the first (*User*) node and ingoing in the second (*Film*) node.

```
1.  MATCH (u:User)-[r:FOLLOWS]->(f:Film)
2.  WHERE ID(u) = $userId
3.  AND ID(f) = $filmId
4.  DELETE r
```

HOME PAGE

This is the query that allows to populate the home page with posts related to followed films or written by followed users:

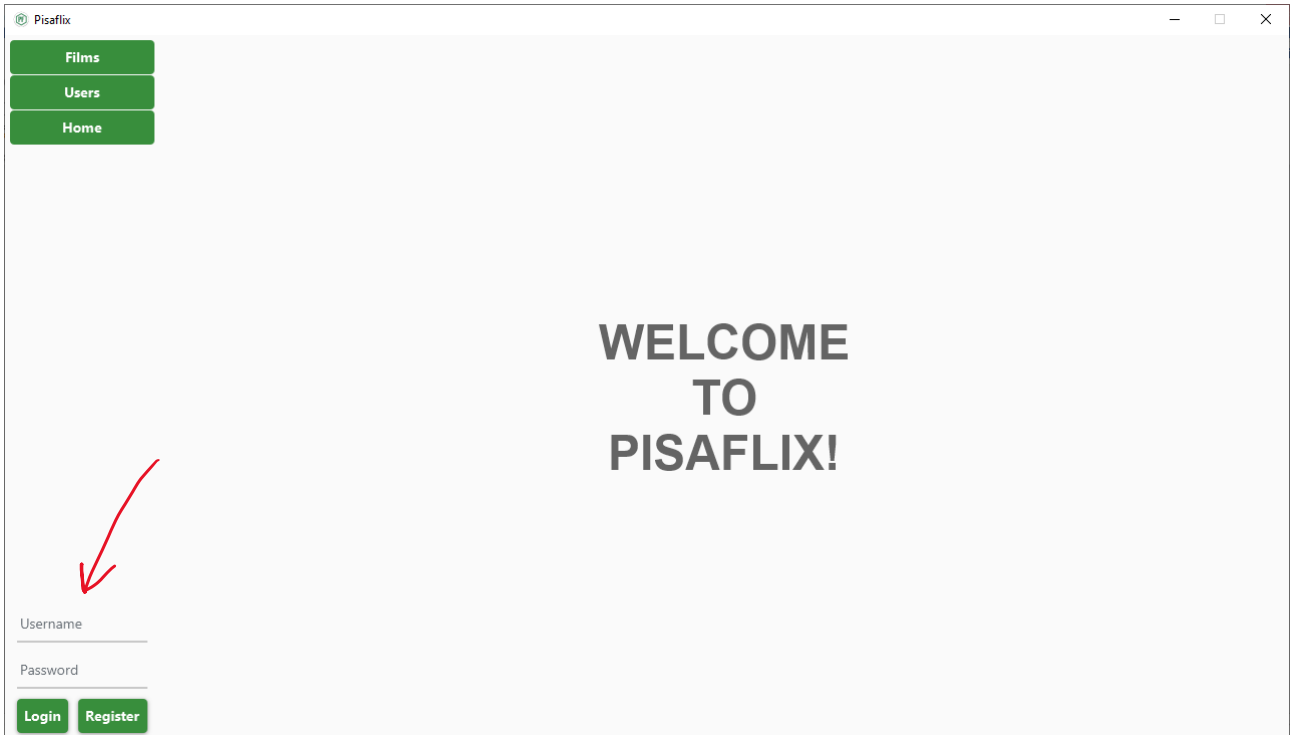
Domain-specific	Graph-centric
What are the posts written by followed users or related to followed films?	What are the (*Post*) nodes that have an ingoing (*Created*) edge from a (*User*) node that has an ingoing (*Follows*) edge from the (*User*) node take into consideration **U** OR an outgoing (*Tags*) edge to a (*Film*) node that has an ingoing (*Follows*) edge from the node **U** AND no ingoing adges from that node?

```
1.  MATCH  (u:User), ()-[r:CREATED]->(p:Post)
2.  WHERE ID(u) = $userId
3.  AND ((u)-[:FOLLOWS]->(:User)-[r]->(p)
4.  OR (u)-[:FOLLOWS]->(:Film)<-[:TAGS]-(p)<-[r]-(:User))
5.  AND NOT (u)-[r]->(p)
6.  RETURN p
```

Note that a check is performed to prevent spurious posts (without creator) from being selected.

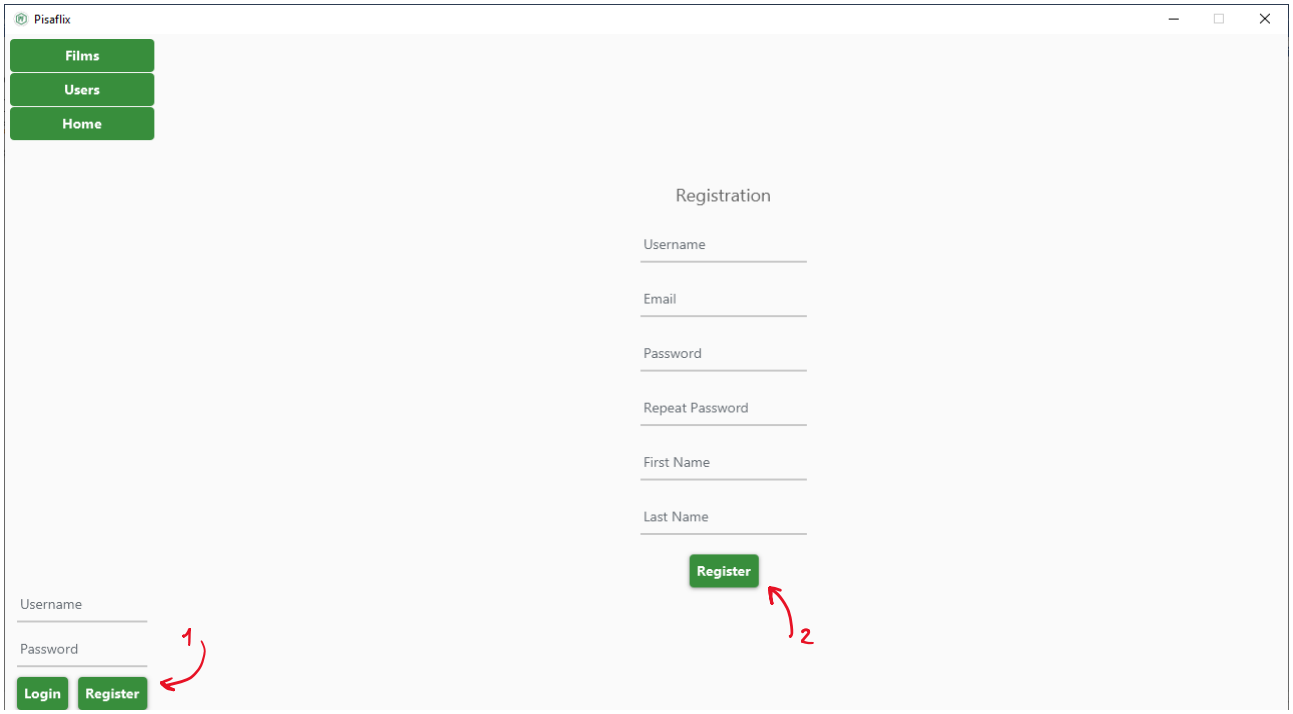
USER MANUAL

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:

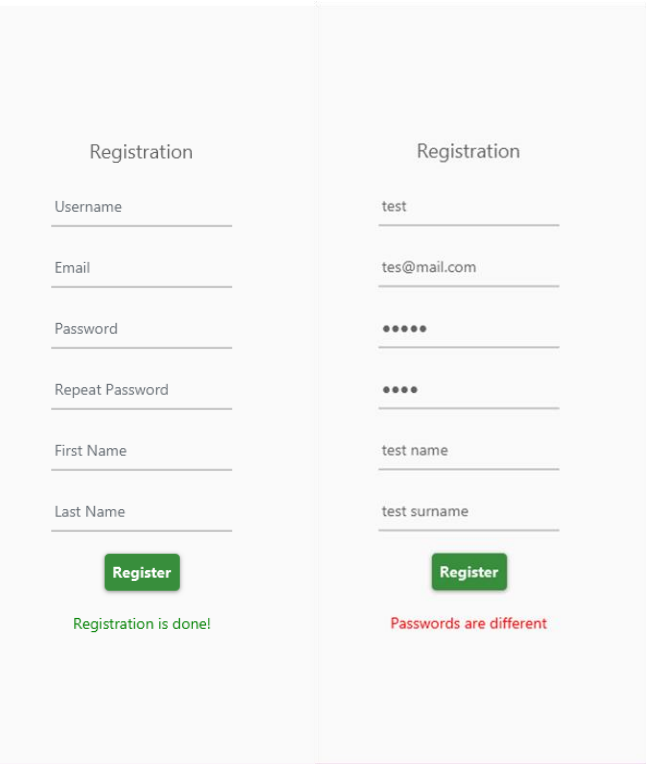


REGISTRATION AND LOGIN

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):



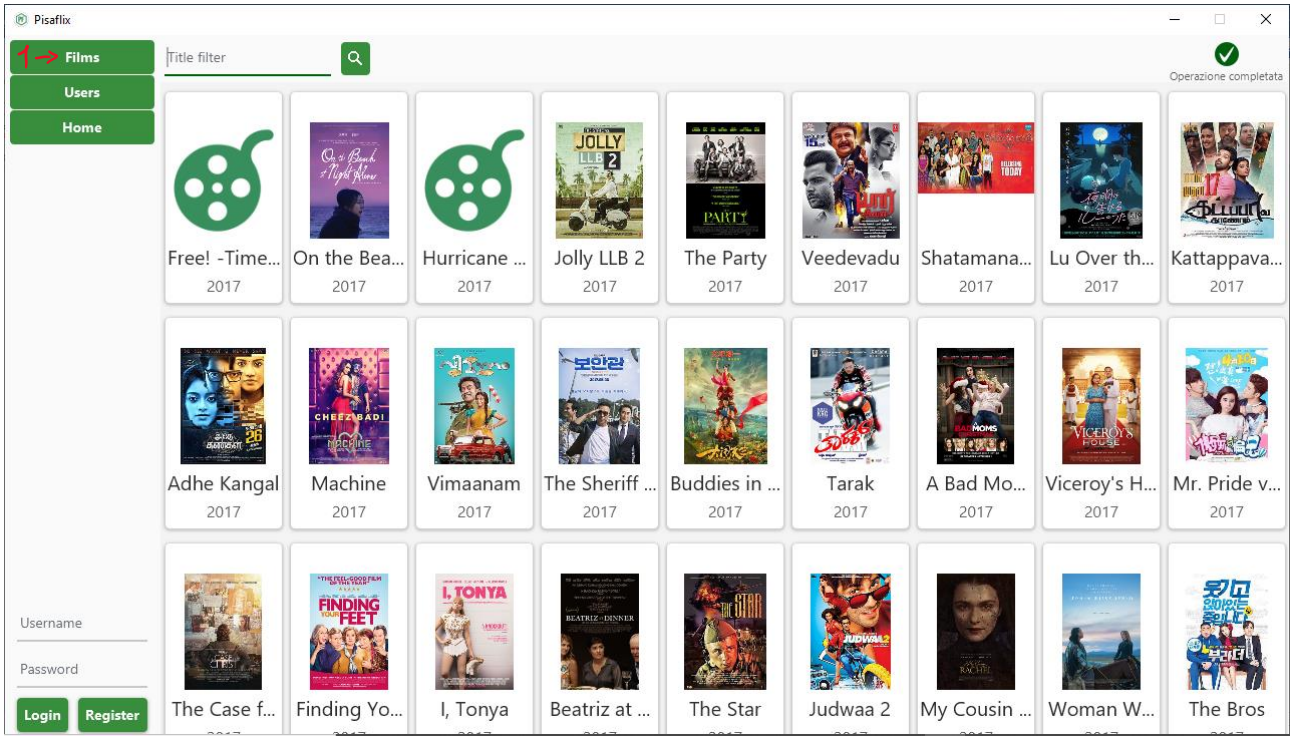
The application will inform the user about any kind of issue after having clicked on the register button. The same is true for a successful registration:



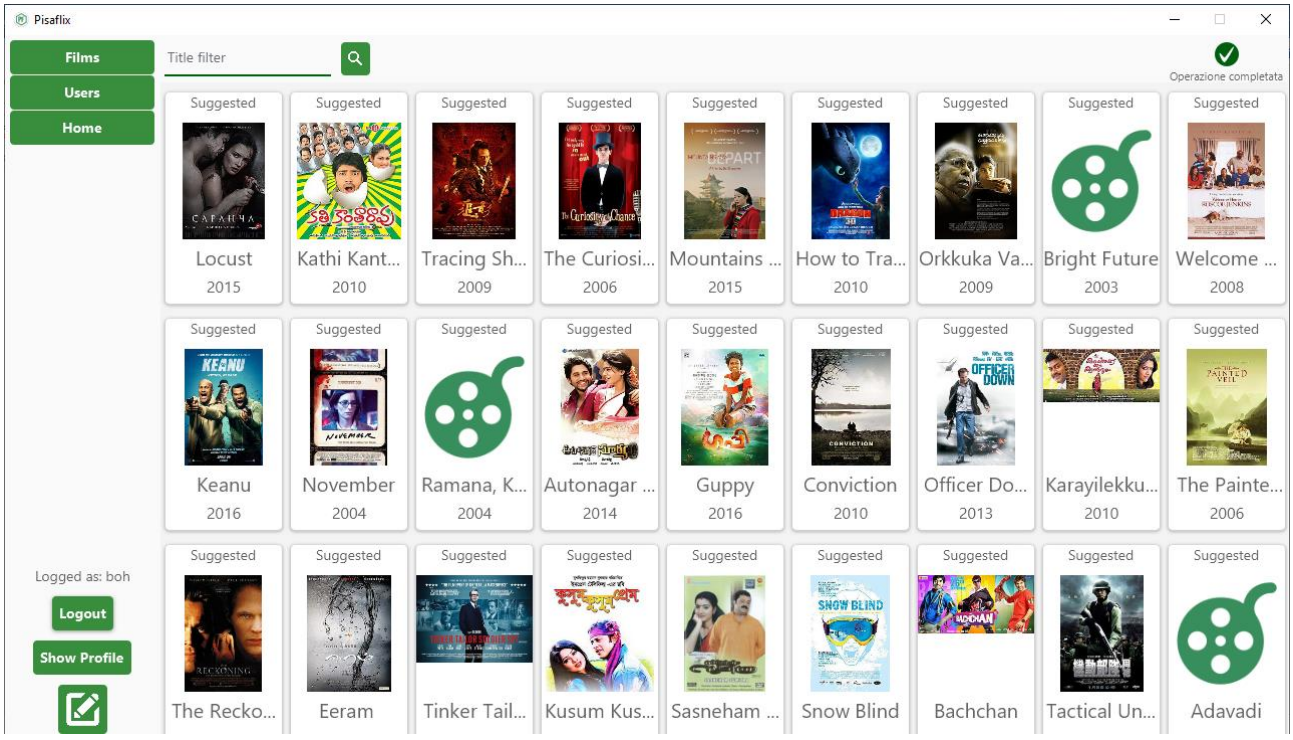
Once registered, the user can log in with the credentials chosen by filling up the form in the bottom left corner. This is the welcoming page:

BROWSING FILM

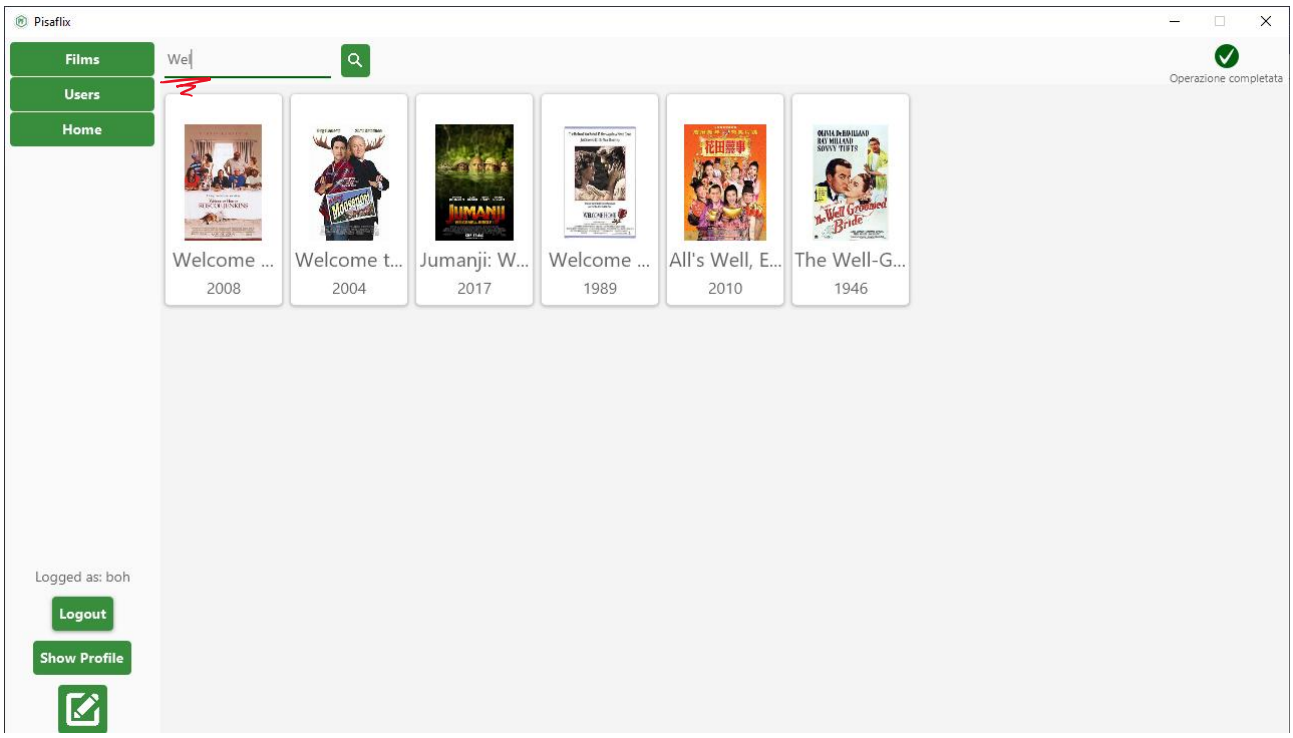
A user can browse (even without being registered) films by clicking the apposite bottom (1) in the top left corner:



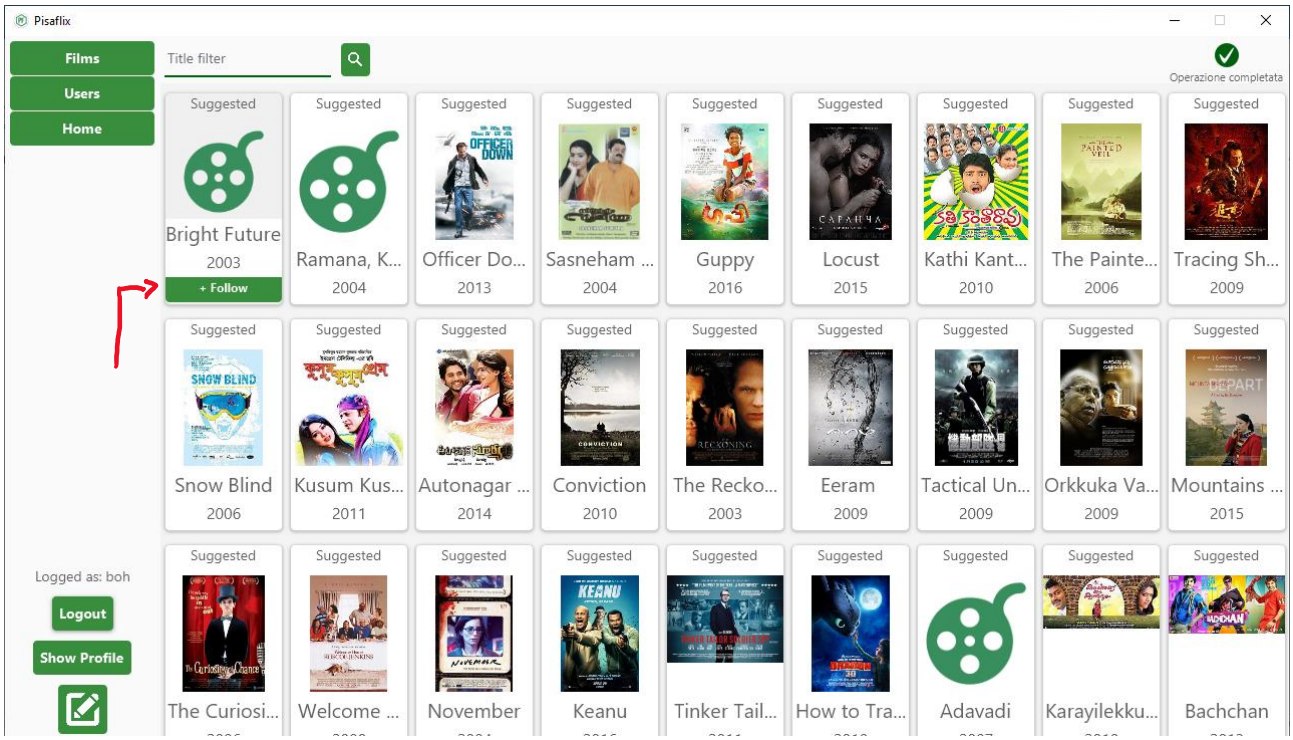
The appearance of this page will change if the user is logged due to the mechanism for suggesting films explained previously.



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



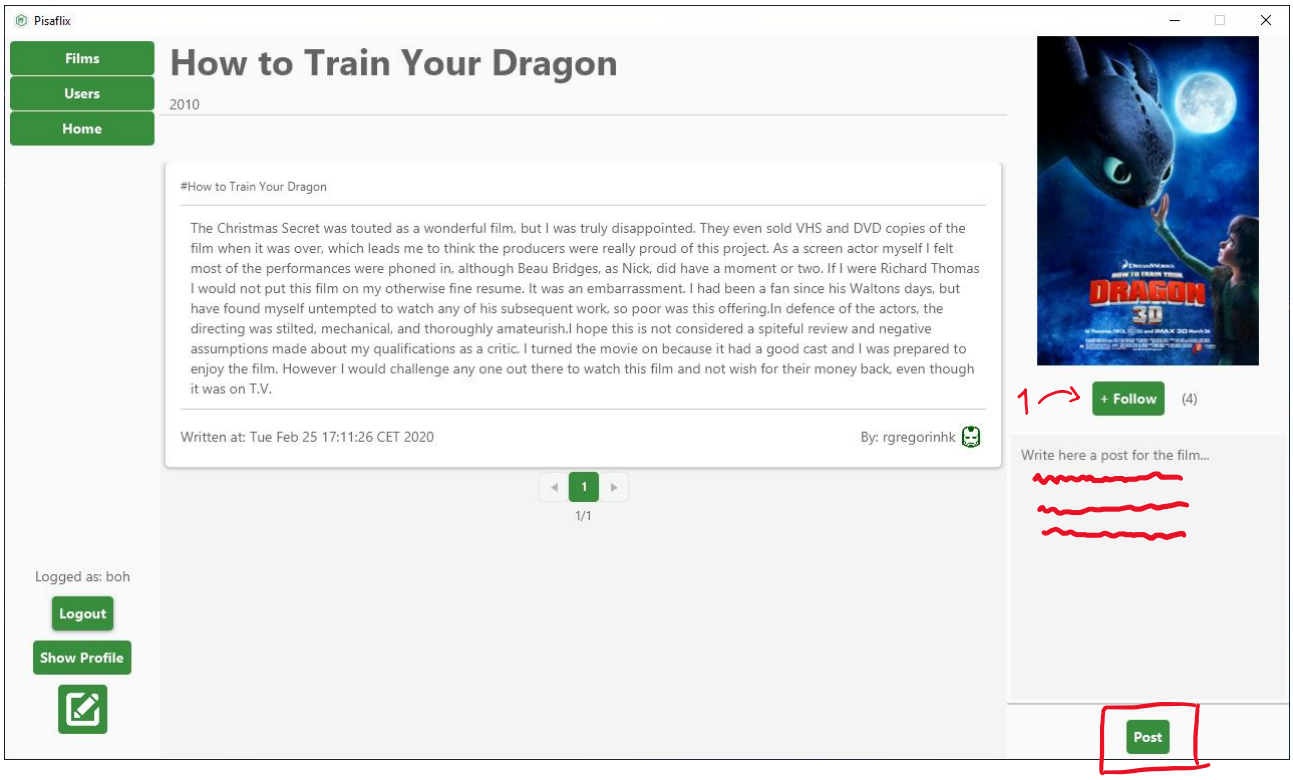
A logged user can also follow a film by simply passing the mouse over the card of a film card (which will reveal a button), and click the follow button:



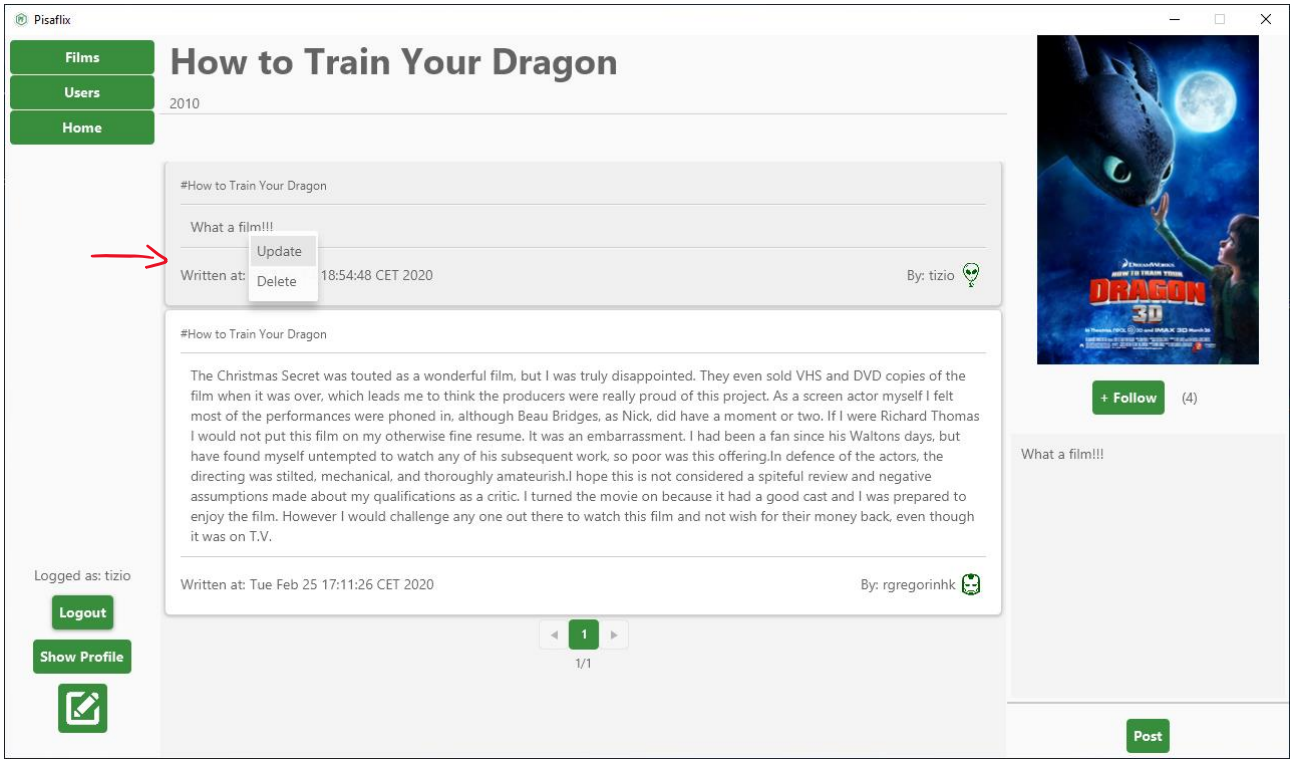
FILM DETAILS

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 posts made by users.

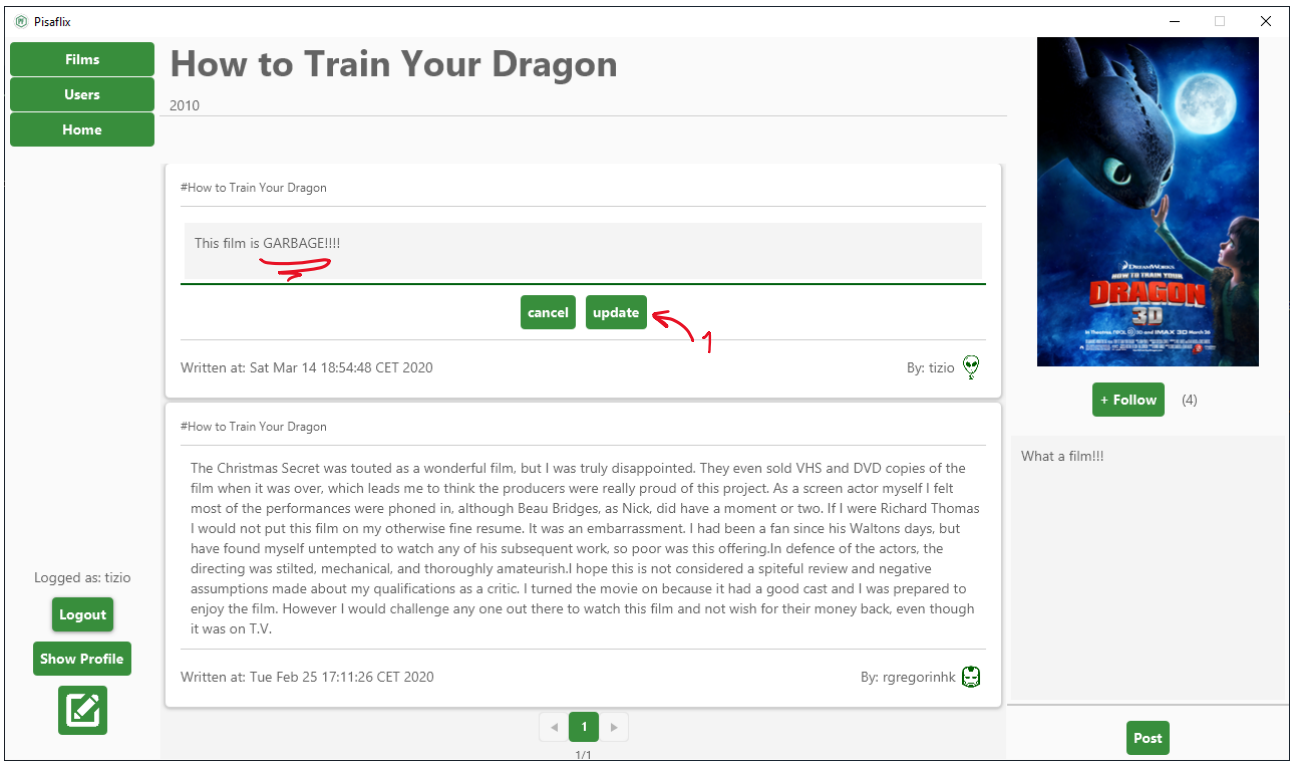
In that page a user, if logged, can follow the film (1) (by clicking the apposite button in the right side of the application) or write a post about it:



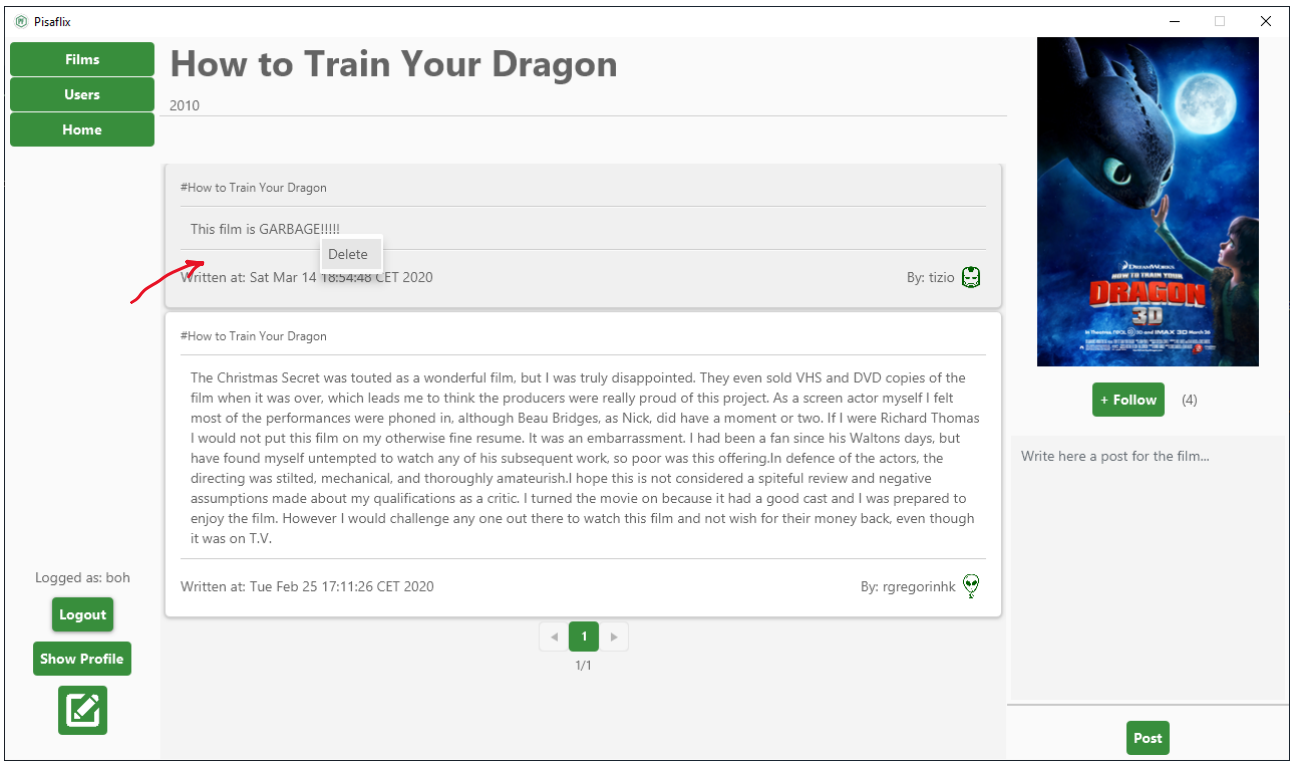
Then the user can also modify/delete its own comments by right clicking on them:



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



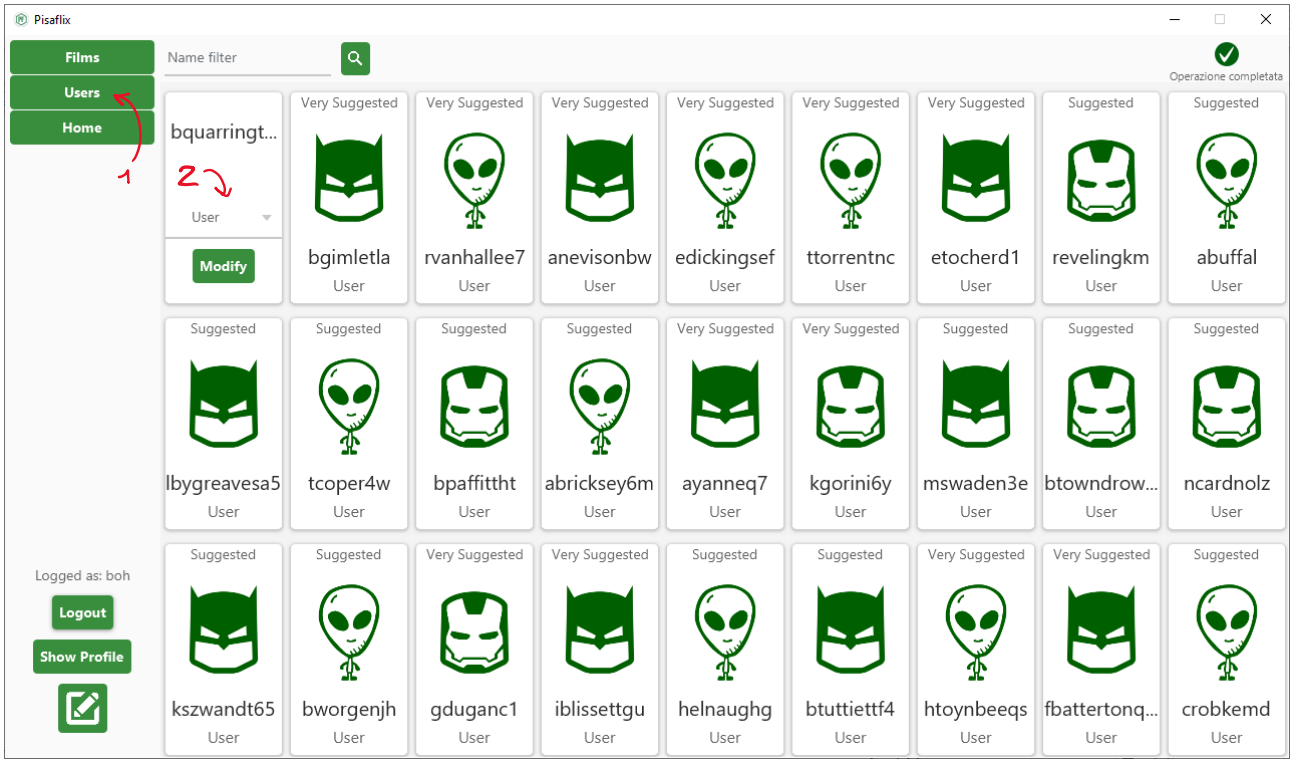
With the right privileges a user can also delete other users' comments, in the same way:



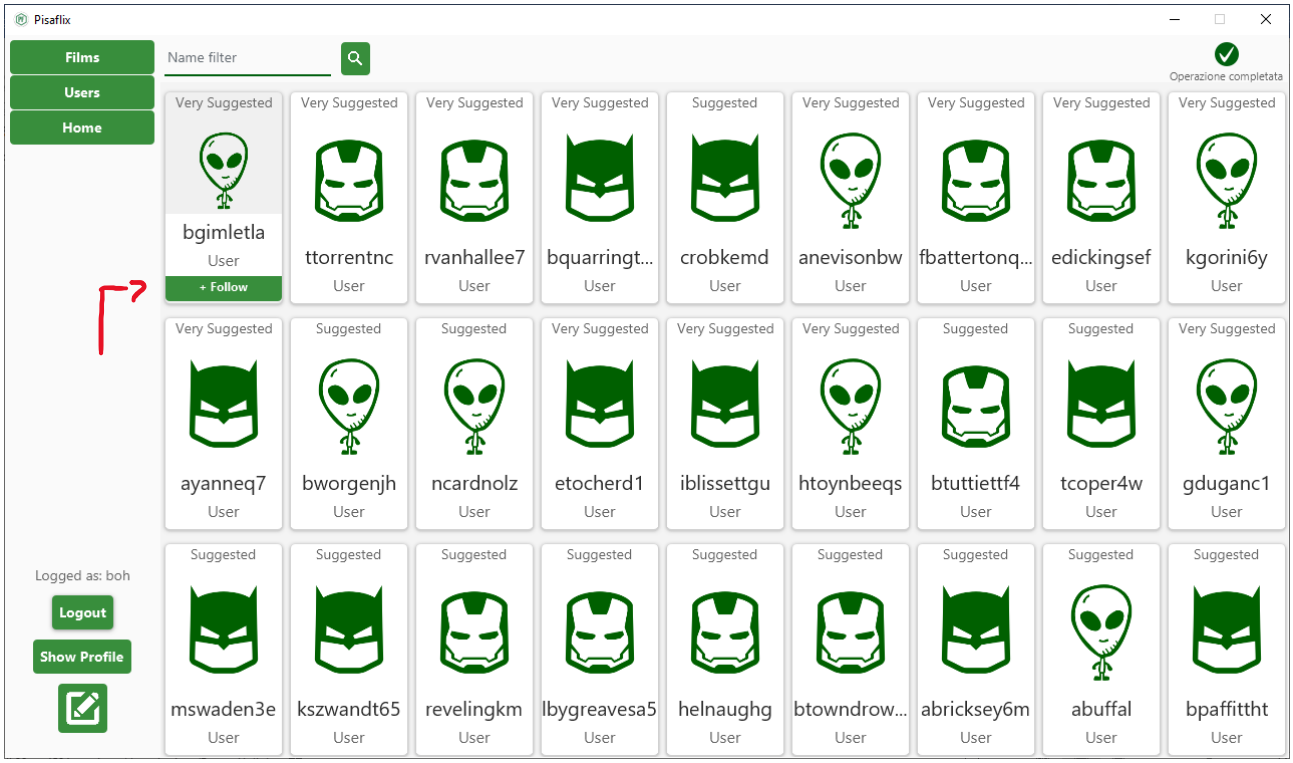
BROWSING USERS AND DETAIL PAGES

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.

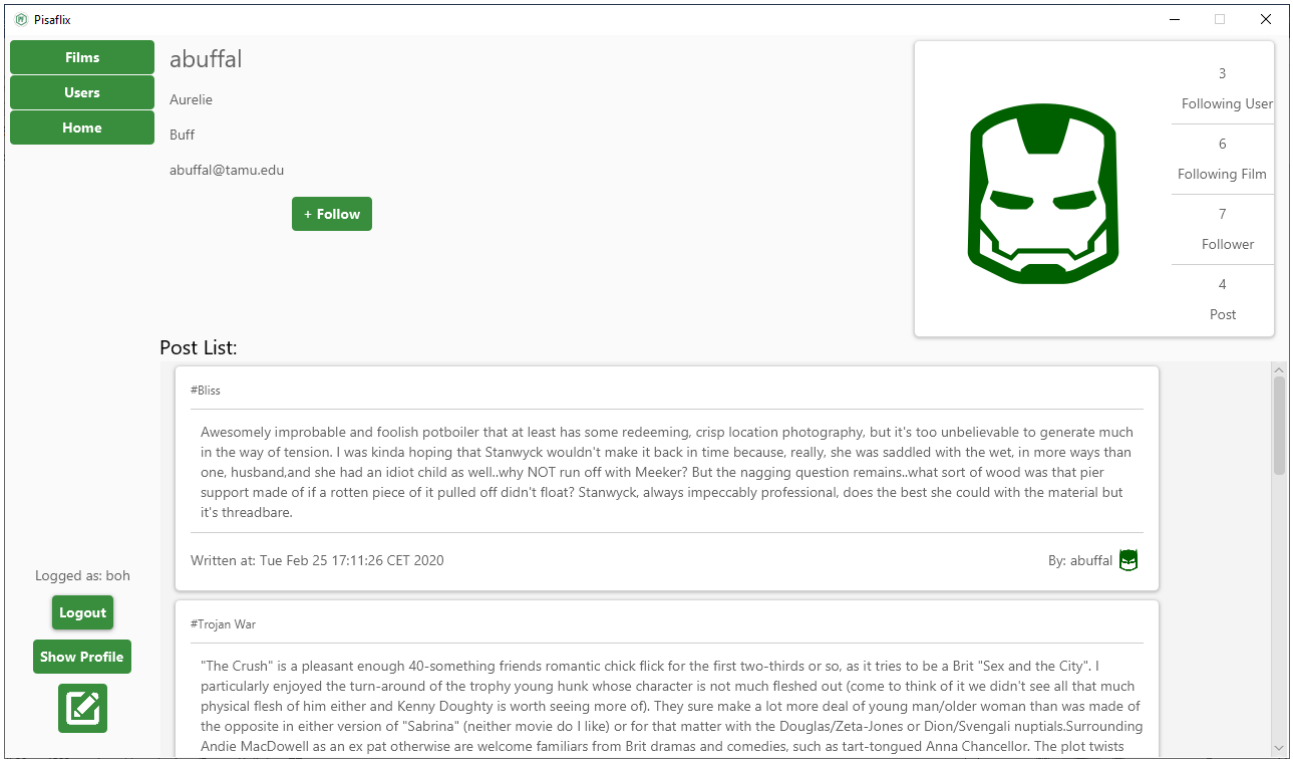
With the right privileges a user can modify other user's privileges by right clicking on them and using the apposite menu (2):



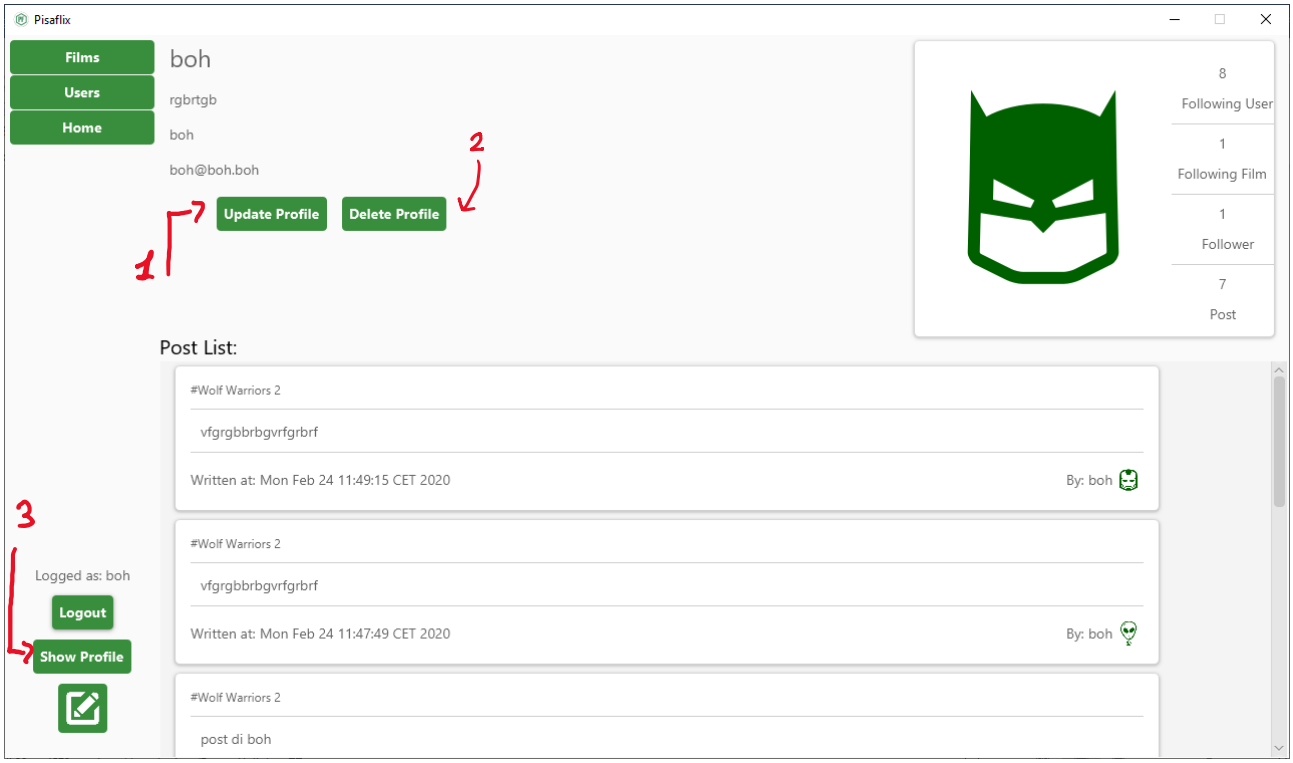
A logged user can follow another user by simply passing the mouse over the card of a user card (which will reveal a button), and click the follow button:



Once the user clicks on a user card while browsing, it will open his/her detail page. There you will find all the information related to the activities done by the specific user:

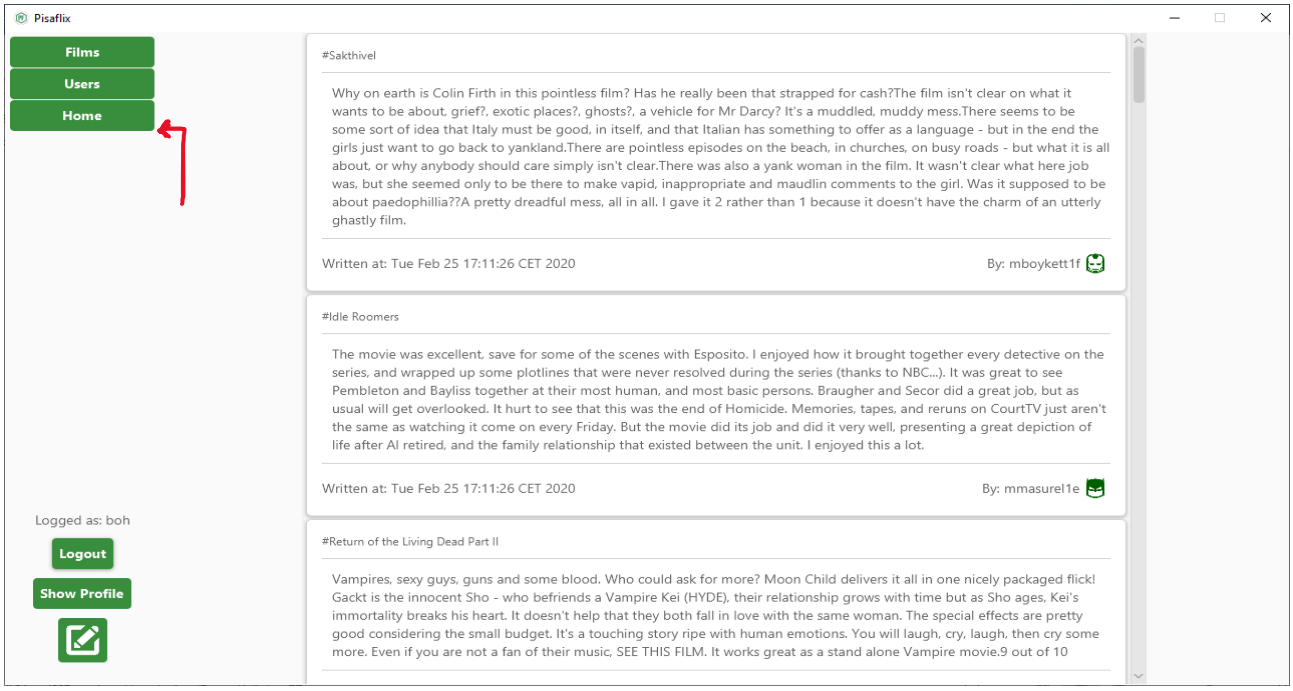


When browsing the user can also click on his own detail page, then he can modify (1) his information or delete (2) his account (the same page is accessible by the apposite button in the bottom left corner after the login (3)).



BROWSING POSTS (HOME PAGE)

A logged user can see his/hers home page by clicking the apposite button on the top left corner. This page will contain a selection of the posts wrote by the users that he/she follows.



By clicking on the name of the user who wrote the post, or the name of the film tagged, we can reach their detail page directly.

WRITE POST

A logged user can write a post by clicking on the apposite button on the bottom left of the screen (1), write the post itself (2), search for films to tag (3), tag them by clicking the cards below (4), erase all tags selected so far (5) and finally publish the post (6):