PISA UNIVERSITY

TASK 1
LARGE-SCALE AND MULTI-STRUCTURED DATABASES

"PISAFLIX" PROJECT DOCUMENTATION

ACADEMIC YEAR 2019-2020

STEFANO PETROCCHI, ANDREA TUBAK, FRANCESCO RONCHIERI, ALESSANDRO MADONNA

SUMMARY

ANALYSIS 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! If you live within the Pisan suburb and you want to enjoy the best experience, PisaFlix is what you need.

PisaFlix is a platform in which you'll find all of the information regarding movies and cinemas in the Pisa area. It gives you the possibility to know which cinema is available, which film you could watch and at what time all of the projections are due. PisaFlix has also a comment section both for cinemas and movies. This allows people to express their opinion, and, by doing so, providing others some really valuable information. Everyone who's still unsure about what to do next will receive a great deal of help by this functionality. We believe PisaFlix offers a complete package of services, that will have a huge impact on the quality of the decisions made by our customers. Proving you everything you need to have a well informed choice is not only our goal, but also a pleasure.

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 the possibility of a complete interaction.

FUNCTIONAL

1. *Users* can **view** the list of **Movies/Cinemas** available on the platform.
2. *Users* can **view** the specific information about a *Movie* (es. category, publish date ecc…).
3. *Users* can **view** the specific information about a *Cinema* (es. Name, Address).
4. *Users* can **view** the *Projections* scheduled in a *Cinema*.
5. *Users* can **view** the *Projections* scheduled for a *Film*.
6. *Users* can **view** the list of ***favorites*** a user.
7. *Users* can **register** an account on the platform.
8. *Users* can **log in** as *Normal users* on the platform in order to do some specific operations:
   a. If logged a *Normal user* can **add/remove** to **favorite** a *Movie/Cinema*.
   b. If logged a *Normal user* can **comment** a *Movie*/*Cinema*.
   c. If logged a *Normal user* can **modify** his *Movie*/*Cinema Comment*.
   d. A *Normal user* can **modify/delete** his account.
9. *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** others users comments.

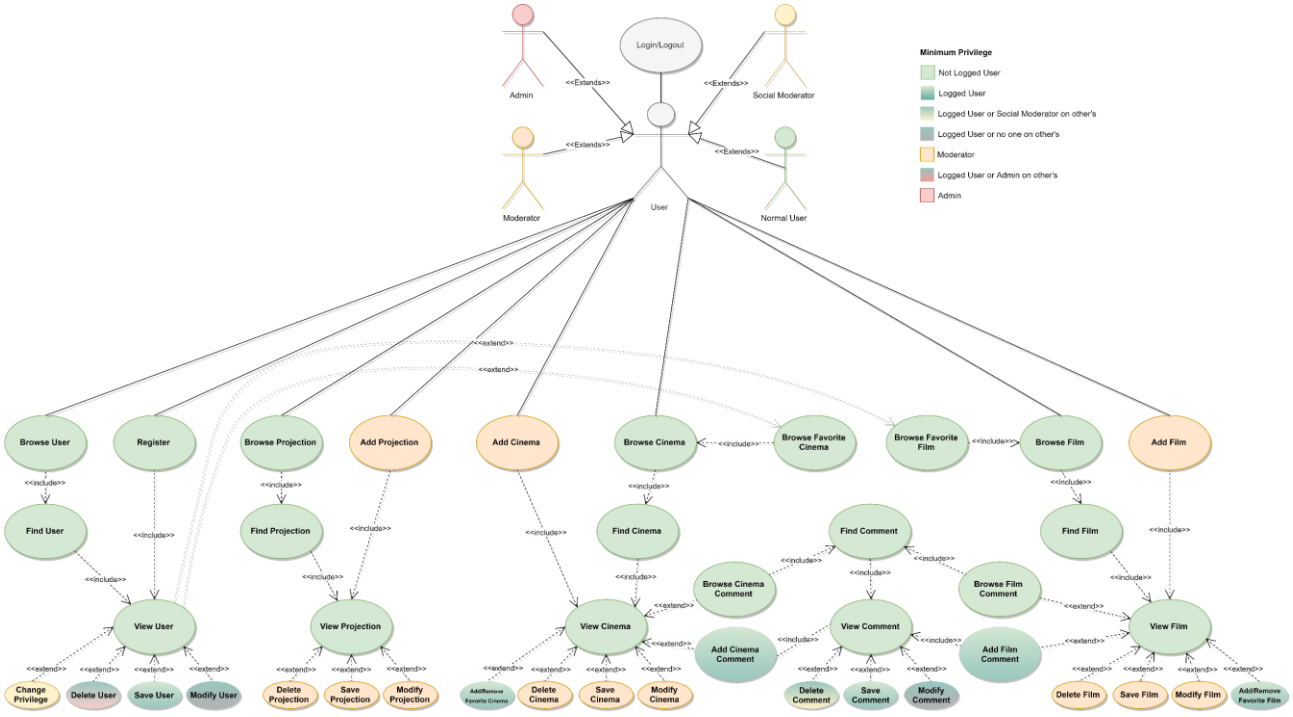
b. If logged as *Social moderator* can **recruit** others *Social moderators*.

10. *Users* that can **log in** as M*oderator* can do all operation of a *Social moderator* plus:
    a. If logged an *Moderator* can **add/delete/modify** a *Movie/Cinema/Projection*.
    b. If logged as *Moderator* can **recruit** other *Moderators*
11. *Users* that can **log in** as *Admins* can do all operation of a M*oderator* plus:
    a. If logged an *Admin* can **delete** other user's account.
    b. If logged as *Admin* can **recruit** other *Admins*.

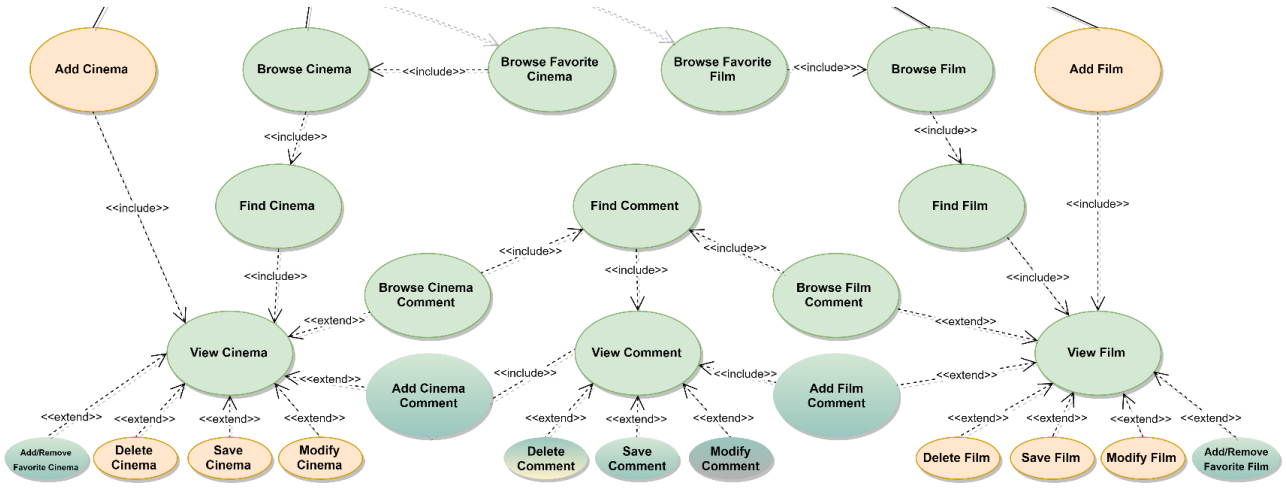
NON-FUNCTIONAL

1. The systems must be on 24/24.
2. The system must support hundred of concurrent access.
3. The response time must be in the order of 1-10 ms.
4. The password must be protected and stored encrypted for privacy issues.

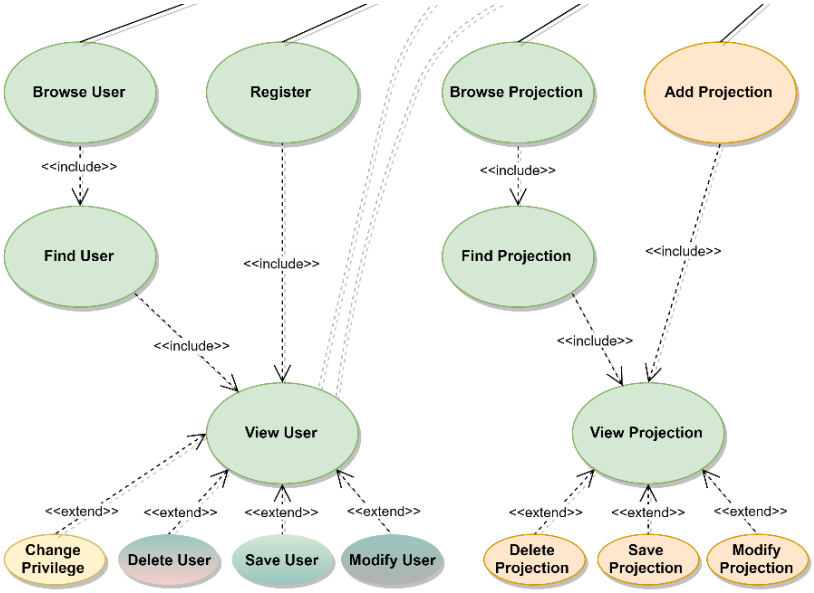
USE CASES



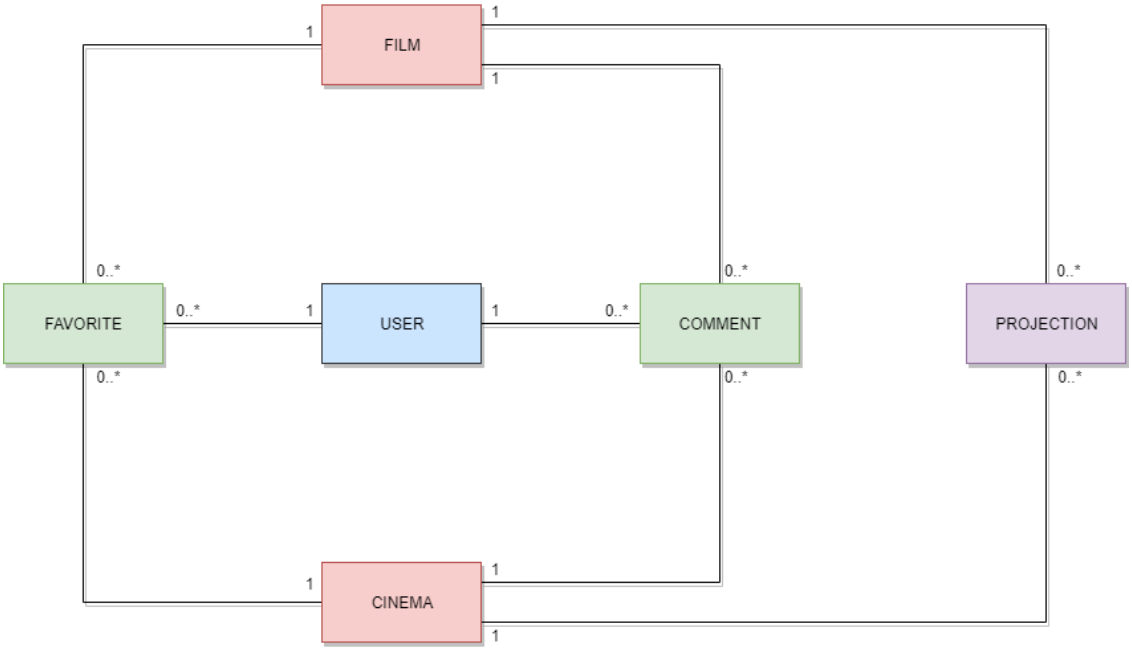
RIGHT DETAIL



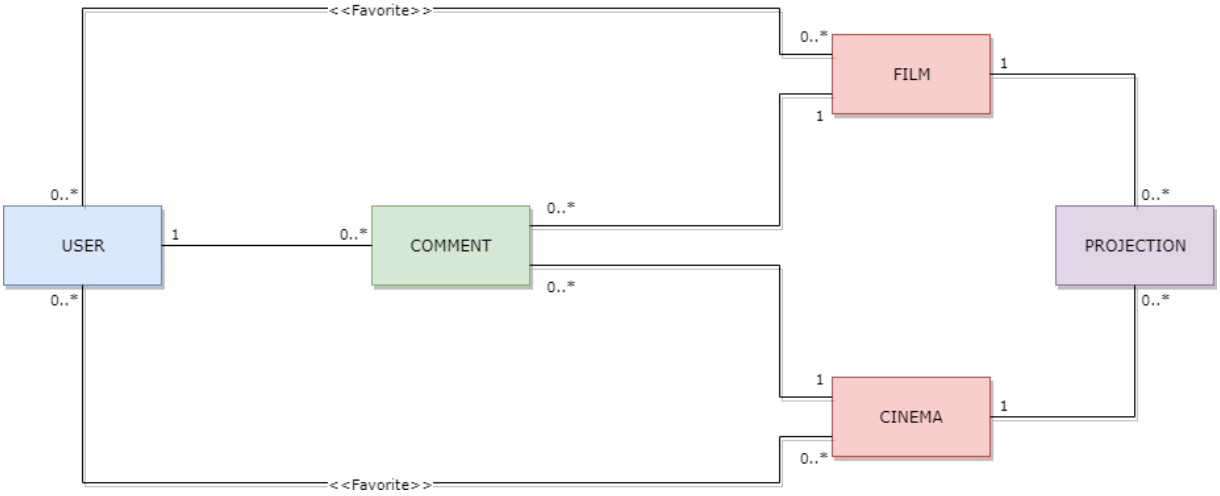
LEFT DETAIL



ANALYSIS CLASSES



DATA MODEL



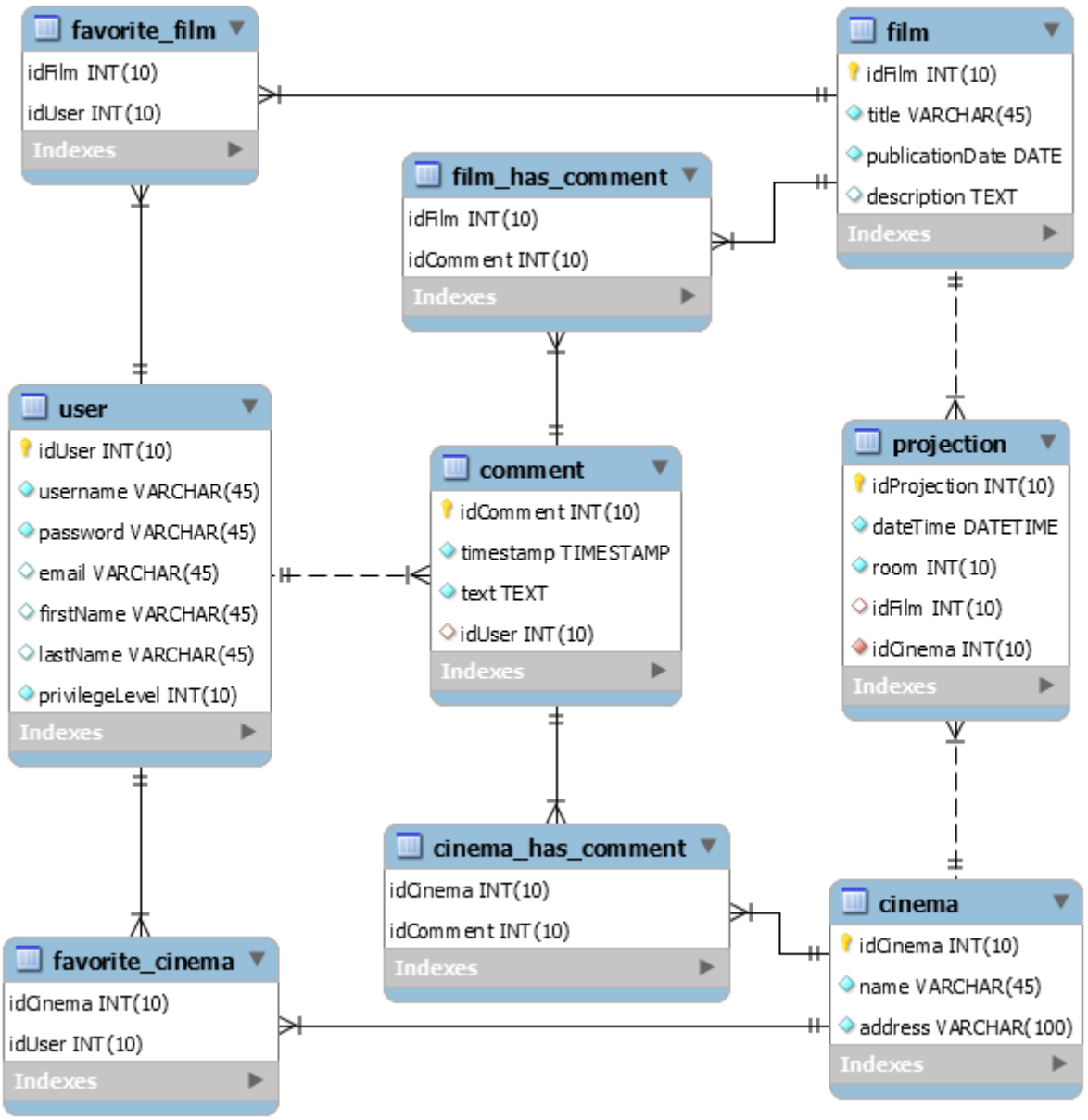
PROJECT DOCUMENT

SOFTWARE ARCHITECTURE

The aim of this project is to build up the platform PisaFlix, a MySQL relational Database was chosen to store all the informations about movies, cinemas, users etc.

The Database has the following structure

E-R DIAGRAM



NOTE: in the table *film_has_comment/cinema_has_comment* the field *idComment* must be UNIQUE, the tables were made in order to make Hibernate work properly.

Users can use a java application with a GUI for using all functionalities of the platform (register, see movies list etc…)

The client Application it's made in Java using JavaFX framework for the GUI and Hibernate JPA for implementing data persistence

GUI – MVC

The graphic user interface was build follow the software design pattern of Model View Controller

Model (PisaFlixServices)

The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

The model is responsible for managing the data of the application. It receives user input from the controller.

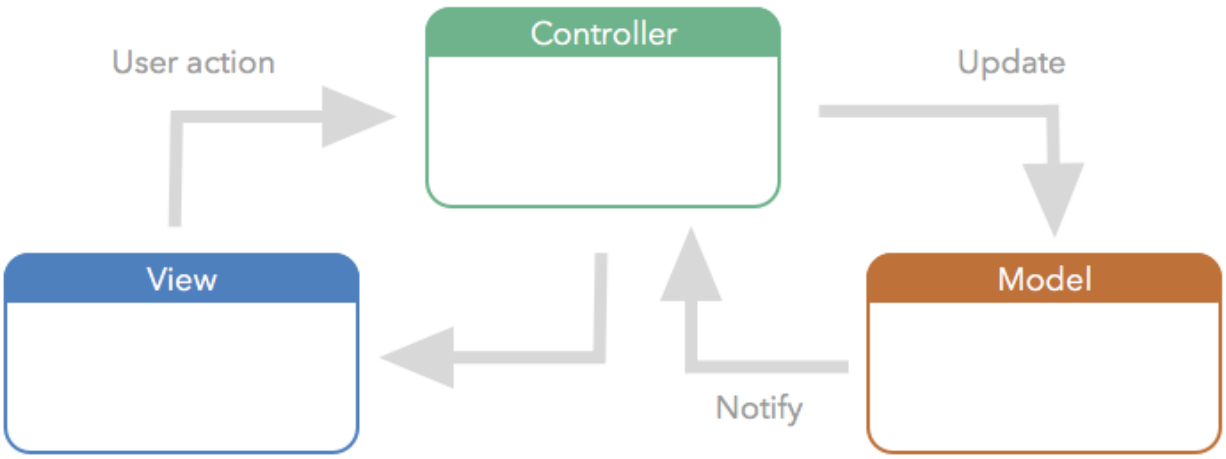
View (FXML files)

All the graphic components (Pages, Buttons).

Controller (Contollers linked to FXML files)

Accepts input and converts it to commands for the model or view.

The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

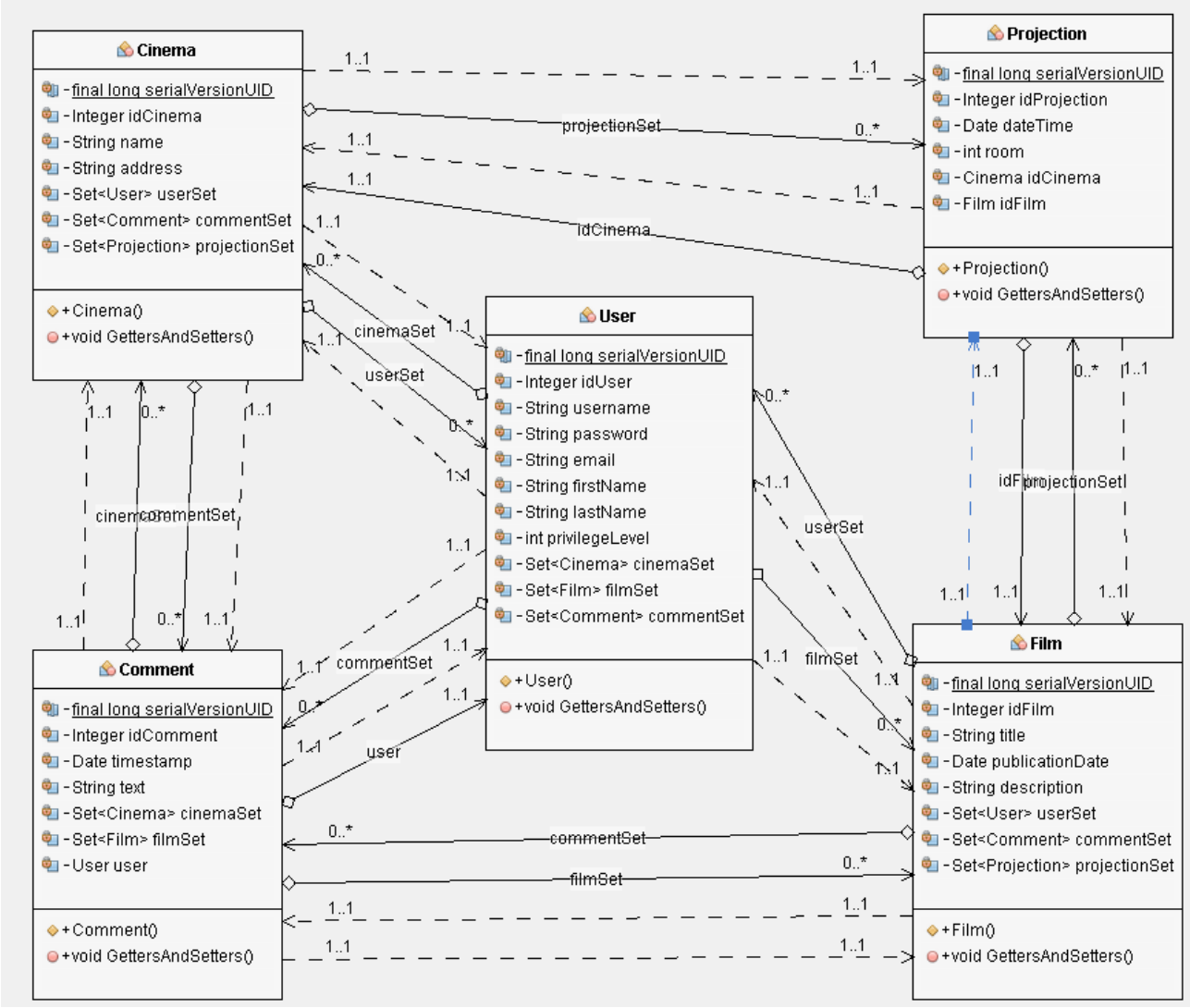


SOFTWARE CLASSES

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.



The only interestring thing is that inside of java file there are directives for Hibernate in order to perform Queries on the database, let's see an example for the film entity.

With `@Entity` we annunce to hibernate our entity film, specify the name of database table `@Table(name = "Film")` after that, we map each class field with the equivalent on the database: let's explain `private Integer idFilm;` the directive `@Id` specify that the field it's part of the primary key, `@GeneratedValue(strategy = GenerationType.IDENTITY)` tells us that if not set will be generate automatically and it will be unique, `@Basic(optional = false)` tells that that field can't be null and at the end with `@Column(name = "idFilm")` we map the field with respectivie field in the database table.

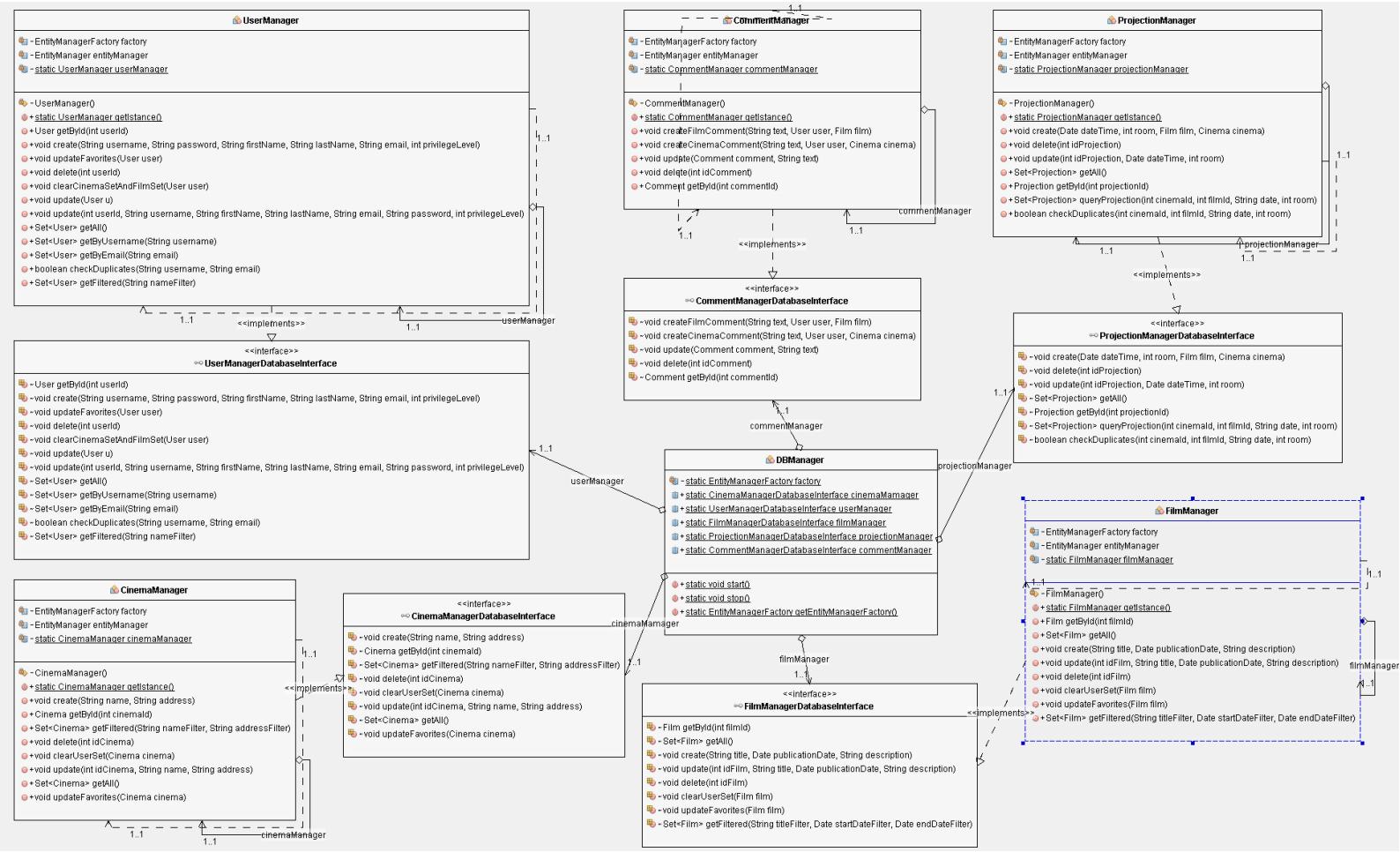
The other fields are used to map relationship with other entities, we will take as example `private Set<User> userSet` which is used to store all users who put as favourite the film.

The directives `@JoinTable` and `@JoinColumn` explain how to make the join with the database table, with `@OneToMany(fetch = FetchType.EAGER)` we specify the type of relationship and setting `fetch = FetchType.EAGER`, we tell to hibernate that when retrive a film automatically retrive all users that put the film into their fauvorite.

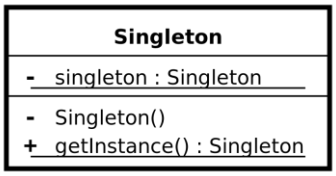
```
1.  //file Film.java
2.  @Entity
3.  @Table(name = "Film")
4.  public class Film implements Serializable {
5.
6.      private static final long serialVersionUID = 1L;
7.
8.      @Id
9.      @GeneratedValue(strategy = GenerationType.IDENTITY)
10.     @Basic(optional = false)
11.     @Column(name = "idFilm")
12.     private Integer idFilm;
13.
14.     @Basic(optional = false)
15.     @Column(name = "title")
16.     private String title;
17.
18.     @Basic(optional = false)
19.     @Column(name = "publicationDate")
20.     @Temporal(TemporalType.DATE)
21.     private Date publicationDate;
22.
23.     @Lob
24.     @Column(name = "description")
25.     private String description;
26.
27.     @JoinTable(name = "Favorite_Film", joinColumns = {
28.         @JoinColumn(name = "idFilm", referencedColumnName = "idFilm")}, inverseJoinColumns
    = {
29.         @JoinColumn(name = "idUser", referencedColumnName = "idUser")})
30.     @ManyToMany(fetch = FetchType.EAGER)
31.     private Set<User> userSet = new LinkedHashSet<>();
32.
33.     @ManyToMany(mappedBy = "filmSet", fetch = FetchType.EAGER, cascade = CascadeType.ALL)

34.     @OrderBy
35.     private Set<Comment> commentSet = new LinkedHashSet<>();
36.
37.     @OneToMany(mappedBy = "idFilm", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
38.     private Set<Projection> projectionSet = new LinkedHashSet<>();
39.
40.     //GETTERS AND SETTERS
41. }
```

DB-MANAGER

Let's se now the structure of DBManager



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.



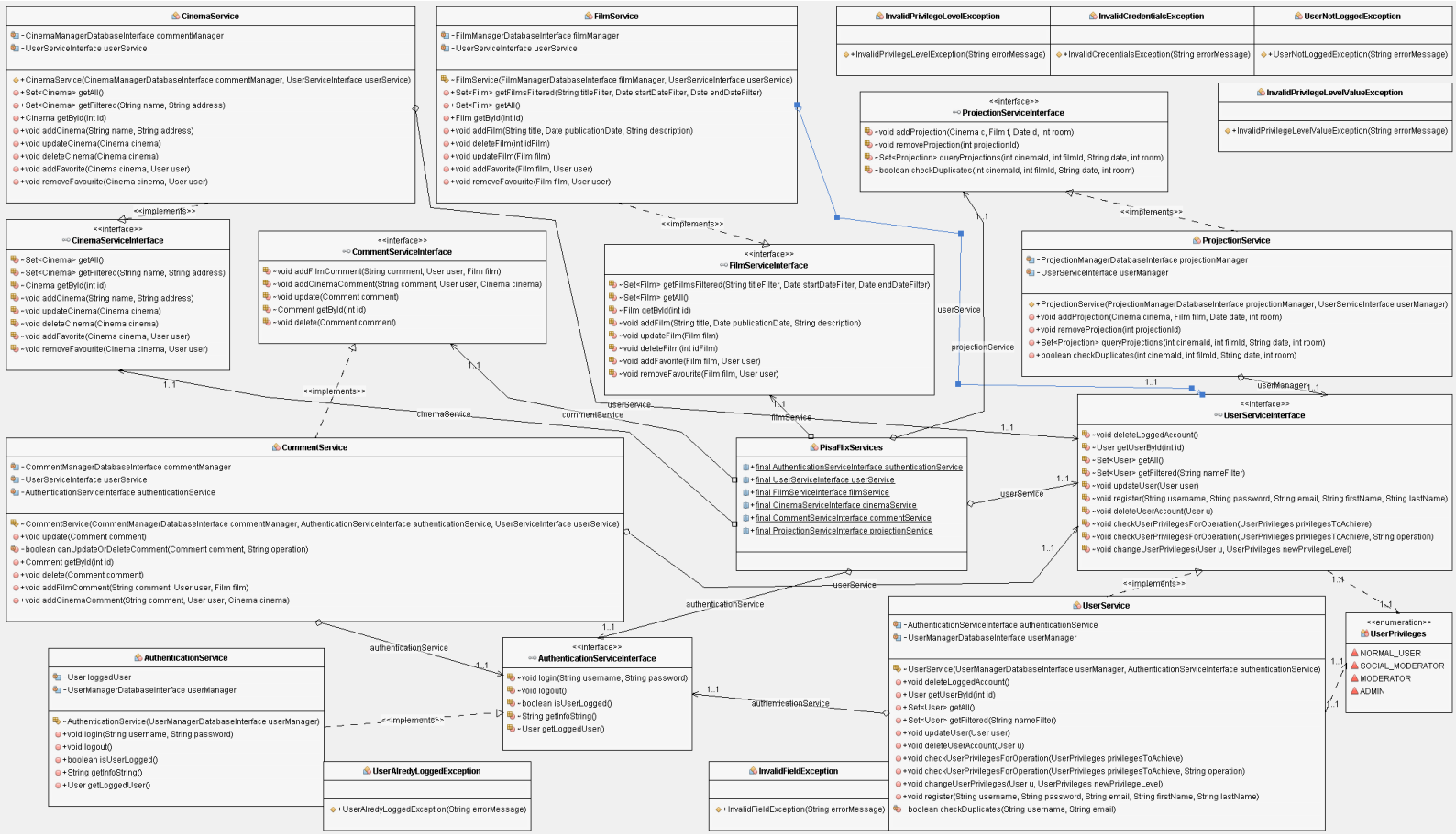
- **DBManager** is an utility class, it's a static class that contains all the other manager specific to certain operations, the other managers are accessible trought the public members of the class, it automatically inizialize all the managers on first call and the method DBManager.Stop() must be called at the end of the application in order to close the factorty manager of hibernate.
- **UserManagerDatabaseInterface** it's the interface which defines the basic operation that any user manager should have (independent from the technology)
  - User **getById**(int *userId*);
  - void **create**(String *username*, String *password*, String *firstName*, String *lastName*, String *email*, int *privilegeLevel*);
  - void **updateFavorites**(User *user*);
  - void **delete**(int *userId*);
  - void **clearCinemaSetAndFilmSet**(User *user*);
  - void **update**(User *u*);

  - void **update**(int *userId*, String *username*, String *firstName*, String *lastName*, String *email*, String *password*, int *privilegeLevel*);
  - Set<User> **getAll**();
  - Set<User> **getByUsername**(String *username*);
  - Set<User> **getByEmail**(String *email*);
  - boolean **checkDuplicates**(String *username*, String *email*);
  - Set<User> **getFiltered**(String *nameFilter*);
- **UserManager** implements **UserManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the users, all function are self-explanatory by the name except for:
  - **getFiltered**(String *nameFilter*) which 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.
- **FilmManagerDatabaseInterface** it's the interface which defines the basic operation that any film manager should have (independent from the technology)
  - Film **getById**(int *filmId*);
  - Set<Film> **getAll**();
  - void **create**(String *title*, Date *publicationDate*, String *description*);
  - void **update**(int *idFilm*, String *title*, Date *publicationDate*, String *description*);
  - void **delete**(int *idFilm*);
  - void **clearUserSet**(Film *film*);
  - void **updateFavorites**(Film *film*);
  - Set<Film> **getFiltered**(String *titleFilter*, Date *startDateFilter*, Date *endDateFilter*);
- **FilmManager** implements **FilmManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the movies, all function are self-explanatory by the name except for:
  - **getFiltered**(String *titleFilter*, Date *startDateFilter*, Date *endDateFilter*) 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.
- **CinemaManagerDatabaseInterface** it's the interface which defines the basic operation that any cinema manager should have (independent from the technology)
  - void **create**(String *name*, String *address*);
  - Cinema **getById**(int *cinemaId*);
  - Set<Cinema> **getFiltered**(String *nameFilter*, String *addressFilter*);
  - void **delete**(int *idCinema*);
  - void **clearUserSet**(Cinema *cinema*);
  - void **update**(int *idCinema*, String *name*, String *address*);
  - Set<Cinema> **getAll**();
  - void **updateFavorites**(Cinema *cinema*);
- **CinemaManager** implements **CinemaManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the cinemas, all function are self-explanatory by the name except for:

  - **getFiltered**(String *nameFilter*, String *addressFilter*) which search and returns all cinemas which have "*nameFilter*" in the name and the "*addressFilter"* in the address, if some filter is not set the filter it's not taken into consideration, if all filter are not set it returns all cinemas.
- **ProjectionManagerDatabaseInterface** it's the interface which defines the basic operation that any projection manager should have (independent from the technology)
  - void **create**(Date *dateTime*, int *room*, Film *film*, Cinema *cinema*);
  - void **delete**(int *idProjection*);
  - void **update**(int *idProjection*, Date *dateTime*, int *room*);
  - Set<Projection> **getAll**();
  - Projection **getById**(int *projectionId*);
  - Set<Projection> **queryProjection**(int *cinemaId*, int *filmId*, String *date*, int *room*);
  - boolean **checkDuplicates**(int *cinemaId*, int *filmId*, String *date*, int *room*);
- **ProjectionManager** implements **ProjectionManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the projections, all function are self-explanatory by the name except for:
  - **queryProjection**(int *cinemaId*, int *filmId*, *String date*, int *room*) which search and returns all projections for cinema specidied by "*cinemaId*" and the film specified by "*filmId"* it also take in consideration the date specidied by "*date"* and the room specified by "*room"* , if some field is not set the field it's not taken into consideration, if all fields are not set it returns all projections.
- **CommentManagerDatabaseInterface** it's the interface which defines the basic operation that any comment manager should have (independent from the technology)
  - void **createFilmComment**(String *text*, User *user*, Film *film*);
  - void **createCinemaComment**(String *text*, User *user*, Cinema *cinema*);
  - void **update**(Comment *comment*, String *text*);
  - void **delete**(int *idComment*);
  - Comment **getById**(int *commentId*);
- **CommentManager** implements **CommentManagerDatabaseInterface** and is in charge of manage all CRUD operation with the database for the comments, all function are self-explanatory so we will not see them in details.

PISAFLIX-SERVICES

Let's se now the structure of PisaFlixServices



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

- **PisaFlixServices** is an utility class, it's a static class that contains all the other manager specific to certain operations, the other services are accessible trought the public members of the class, it automatically inizialize all the services on first call.
- **UserPrivileges** it's an enumeration class which map 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 manage the authentication procedure of the application, it use **UserManagerDatabaseInterface** in order to operate with database and obtain datas
  - 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 trhow *UserAlredyLoggedException* if called with an already open session or *InvalidCredentialsException* if called with invalid credentials

  - void **logout**() it close the session deleting user information stored in the local variable
  - boolean **isUserLogged**() it checks if the user is logged and give back the results
  - String **getInfoString**() it provides some text information of the current session (ex. "logged as Example"
  - User **getLoggedUser**() get the information of the loggedUser
- **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 is in charge of manage all 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 dempending by the operation that we want perform
  - Set<User> **getAll**() returns all the users in the DB
  - User **getUserById**(int *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 specidy 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 **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
  - 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.

- **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 use an **FilmManagerDatabaseInterface** to exchange data with the DB and a **UserServiceInterface** for ensure that we have the right privileges dempending 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**(int *idFilm*) allows to delte 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 **addFavorite**(Film *film*, User *user*) allows to add a specific "*film*" as favourite of a specific "*user*"
    - void **removeFavourite**(Film *film*, User *user*) allows to remove a specific "*film*" as favourite of a specific "*user*"
- **CinemaServiceInterface** it's the interface which defines the basic operation that any cinema service should have (independent from the technology)
    - we will see the methods in detail in the class which implement it
- **CinemaService** implements **CinemaServiceInterface** and is in charge of manage all operations that are specific for cinemas, in order to work properly it use an **FilmManagerDatabaseInterface** to exchange data with the DB and an **UserServiceInterface** for ensure that we have the right privileges dempending by the operation that we want perform
    - Set<Cinema> **getAll**() returns all cinemas int the DB
    - Set<Cinema> **getFiltered**(String *name*, String *address*) search int the DB and returns all cinemas which have "*nameFilter*" in the name and the "*addressFilter*" in the address, if some filter is not set the filter it's not taken into consideration, if all filter are not set it returns all cinemas.
    - Cinema **getById**(int *id*) returns a specific film identify by his "*id*"

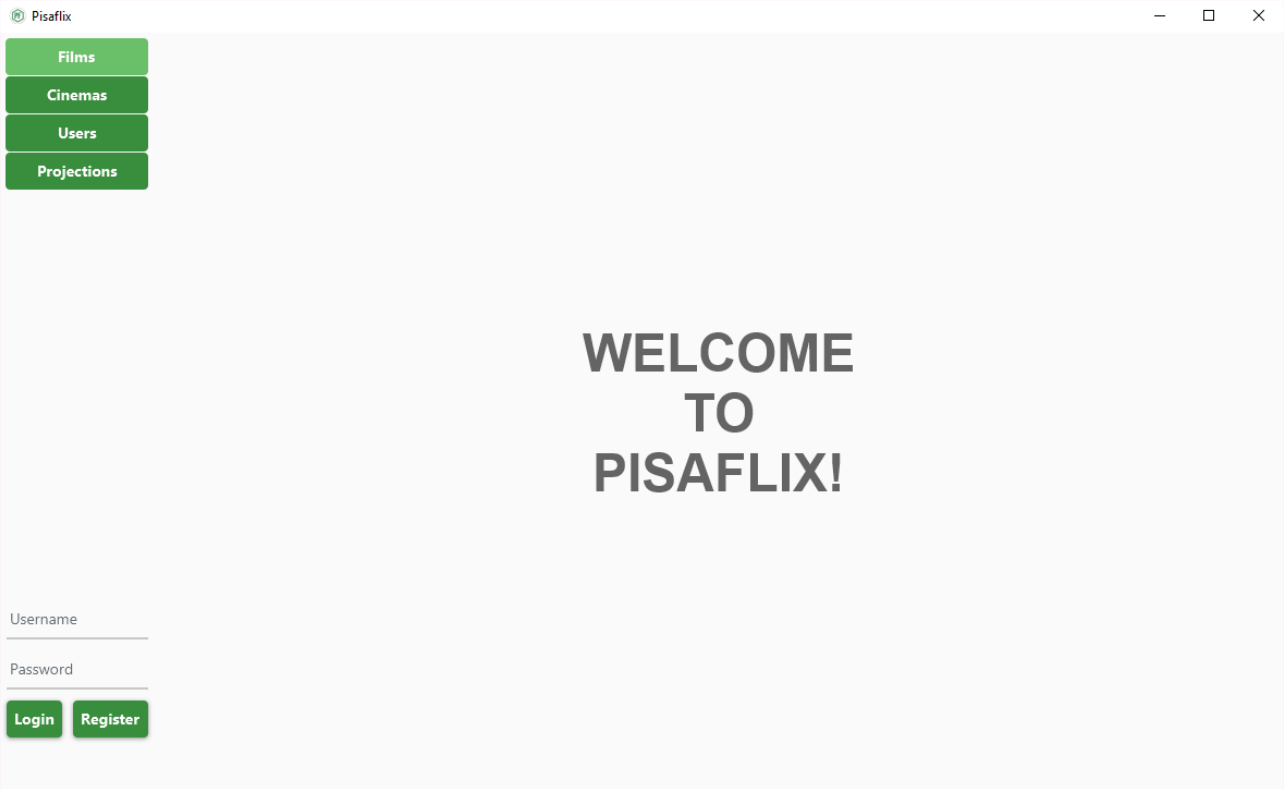
  - void **addCinema**(String *name*, String *address*) allows to insert a new cinema in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't add a new cineam
  - void **updateCinema**(Cinema *cinema*) allows to modify a cinema in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't modify a cinema
  - void **deleteCinema**(Cinema *cinema*) allows to delete a cinema in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't delete a cinema
  - void **addFavorite**(Cinema *cinema*, User *user*) allows to add a specific "*film"* as favourite of a specific "*user"*
  - void **removeFavourite**(Cinema *cinema*, User *user*) allows to remove a specific "*film"* as favourite of a specific "*user"*
- **CommentServiceInterface** it's the interface which defines the basic operation that any comment service should have (independent from the technology)
  - we will see the methods in detail in the class which implement it
- **CommentService** implements **CommentServiceInterface** and is in charge of manage all operations that are specific for comments, in order to work properly it use an **CommentManagerDatabaseInterface** to exchange data with the DB, an **AuthenticationService** in order to retrieve the current logged user and an **UserServiceInterface** for ensure that we have the right privileges dempending by the operation that we want perform
  - Comment **getById**(int *id*) returns a specific film identify by its "*id"*
  - void **addFilmComment**(String *comment*, User *user*, Film *film*) creates a new comment for a "*film*" made by a certain "*user*"
  - void **addCinemaComment**(String *comment*, User user, Cinema *cinema*) creates a new comment for a "*cinema*" made by a certain "*user*"
  - 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
- **ProjectionServiceInterface** it's the interface which defines the basic operation that any projection service should have (independent from the technology)
  - we will see the methods in detail in the class which implement it
- **ProjectionService** implements **ProjectionServiceInterface** and is in charge of manage all operations that are specific for projections, in order to work properly it use an **CommentManagerDatabaseInterface** to exchange data with the DB and an **UserServiceInterface** for ensure that we have the right privileges dempending by the operation that we want perform
  - void addProjection(Cinema c, Film f, Date d, int room) allows to insert a new projection in the DB, it throws *UserNotLoggedException* if called with no user

logged, or *InvalidPrivilegeLevelException* if the logged user can't add a new projection

- void removeProjection(int projectionId) allows to delete a projection in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't delete a projection
- Set<Projection> queryProjections(int cinemaId, int filmId, String date, int room) search int the DB and returns all projections for cinema specidied by "*cinemaId*" and the film specified by "*filmId*" it also take in consideration the date specidied by "*date*" and the room specified by "*room*" , if some field is not set the field it's not taken into consideration, if all fields are not set it returns all projections.

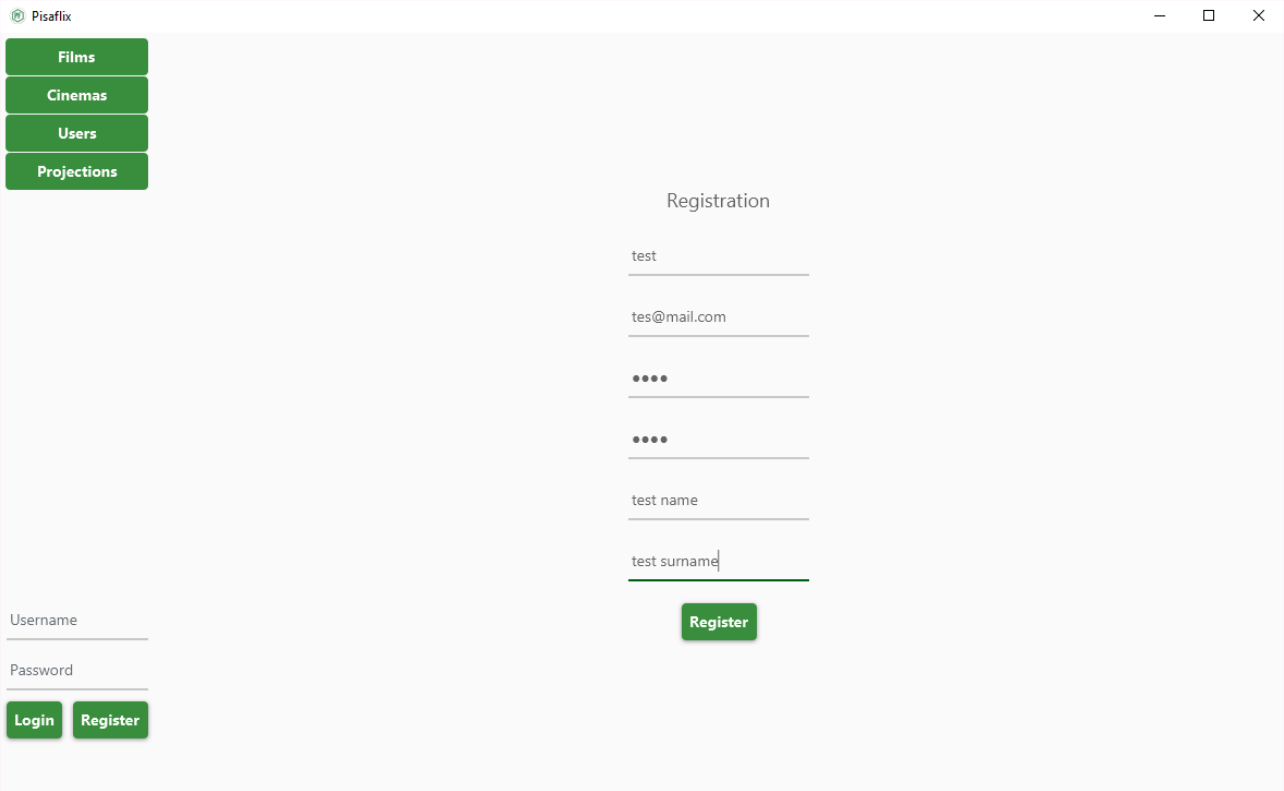
USER MANUAL

The graphic interface is based on a left side menu and a space on the right where the application pages are displayed, at the bottom of the menu it is possible to log in

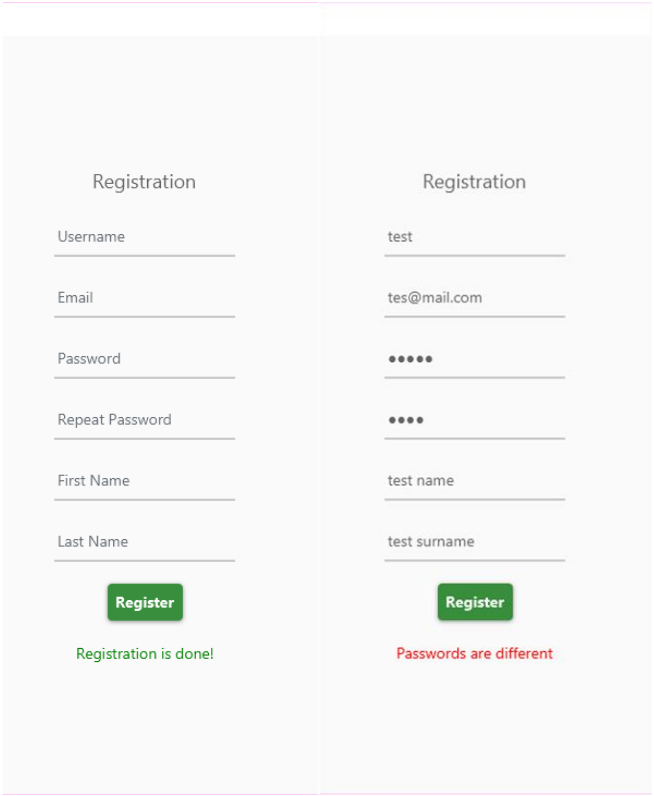


REGISTRATION AND LOGIN

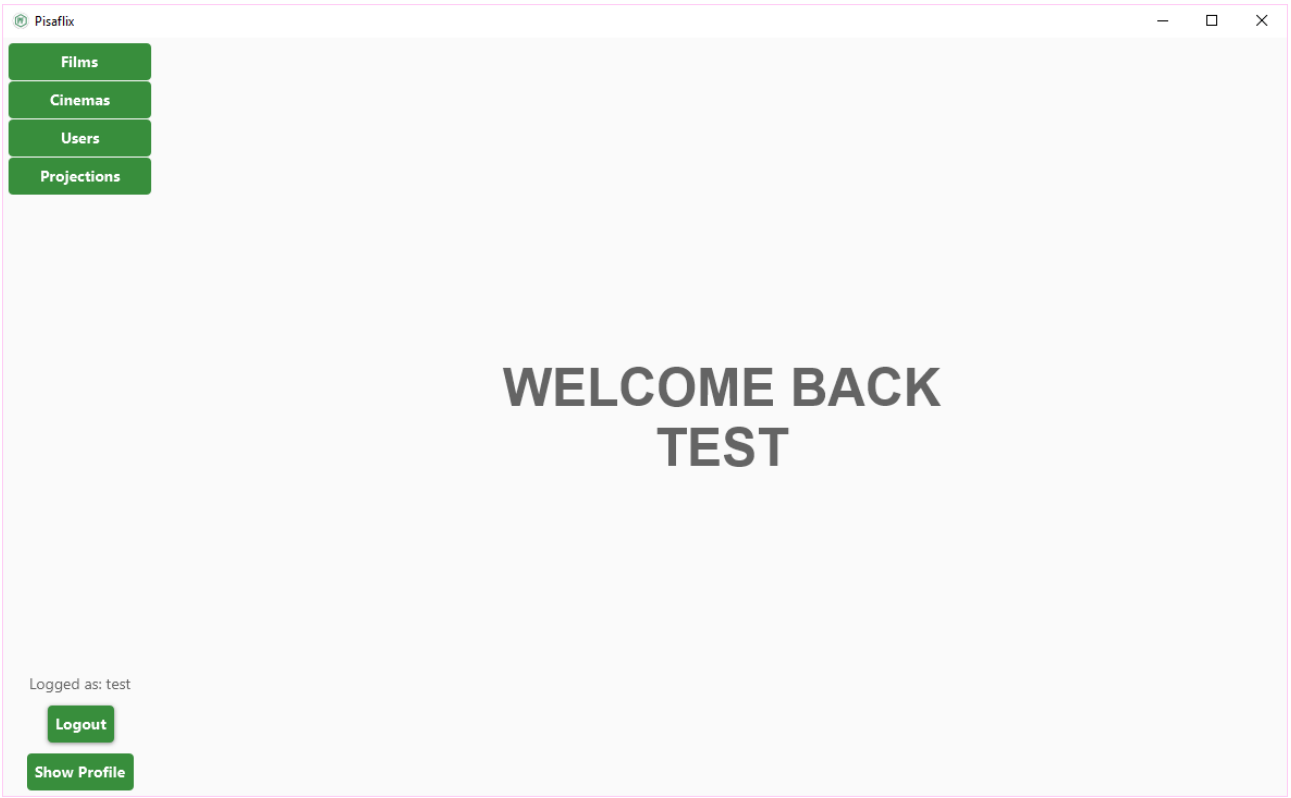
A new user can register using the specific button on the log in part in the buttom left corner, after clicking, the registration page will appear which a user can fill out with his own information and then register.



Both in case of errors or success the application shows the result with some text information

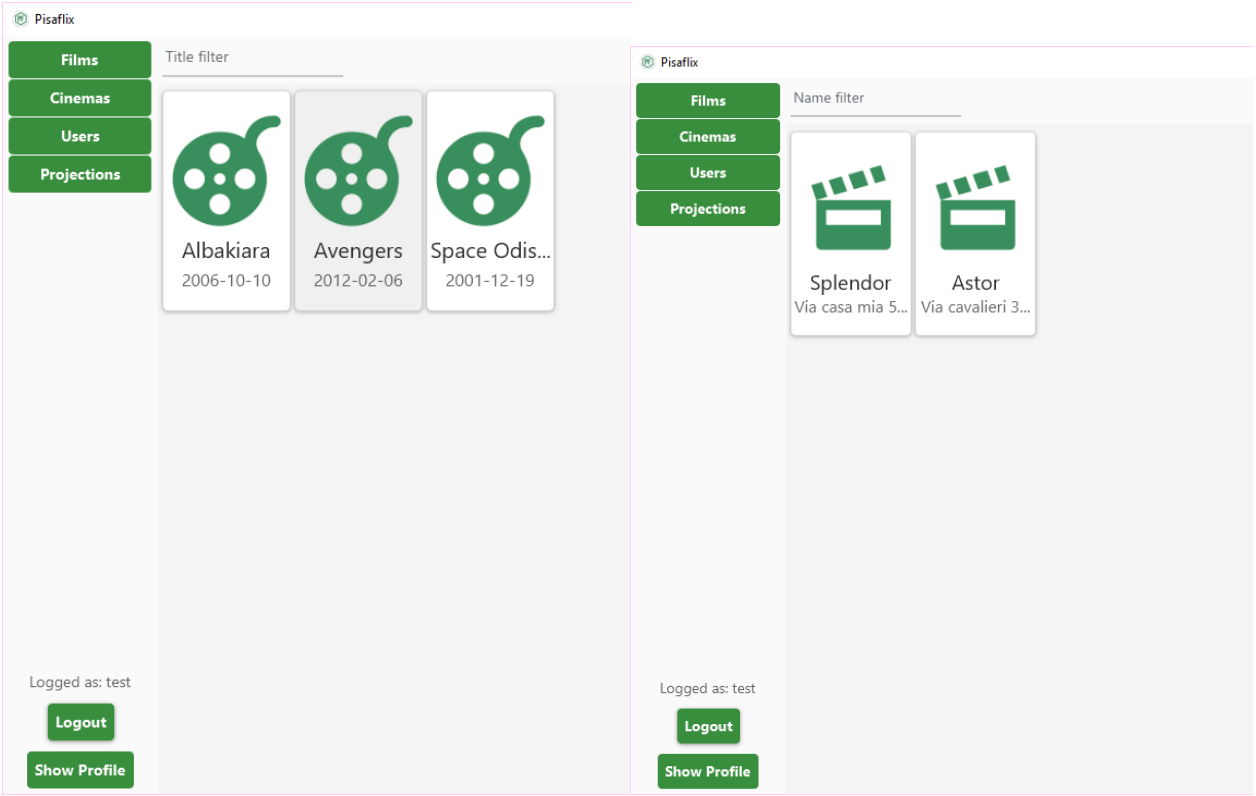


Once registrated the user can log in by the apposite fields in the button left corner, the the user can comments movies/cinemas, add them to favourite and do all other specific operations based on his privileges.

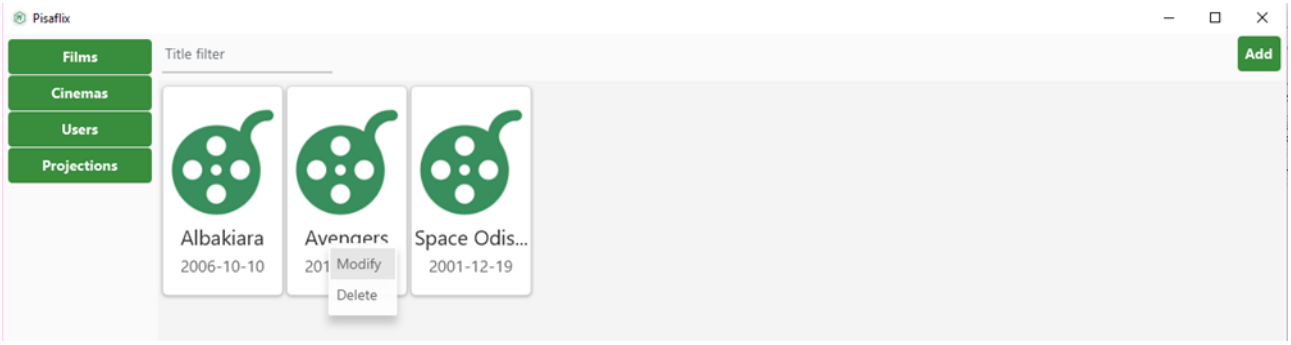


BROWSING FILM/CINEMAS

Once open the application a user can browse films and cinemas by clicking the apposite bottons in the top left corner.



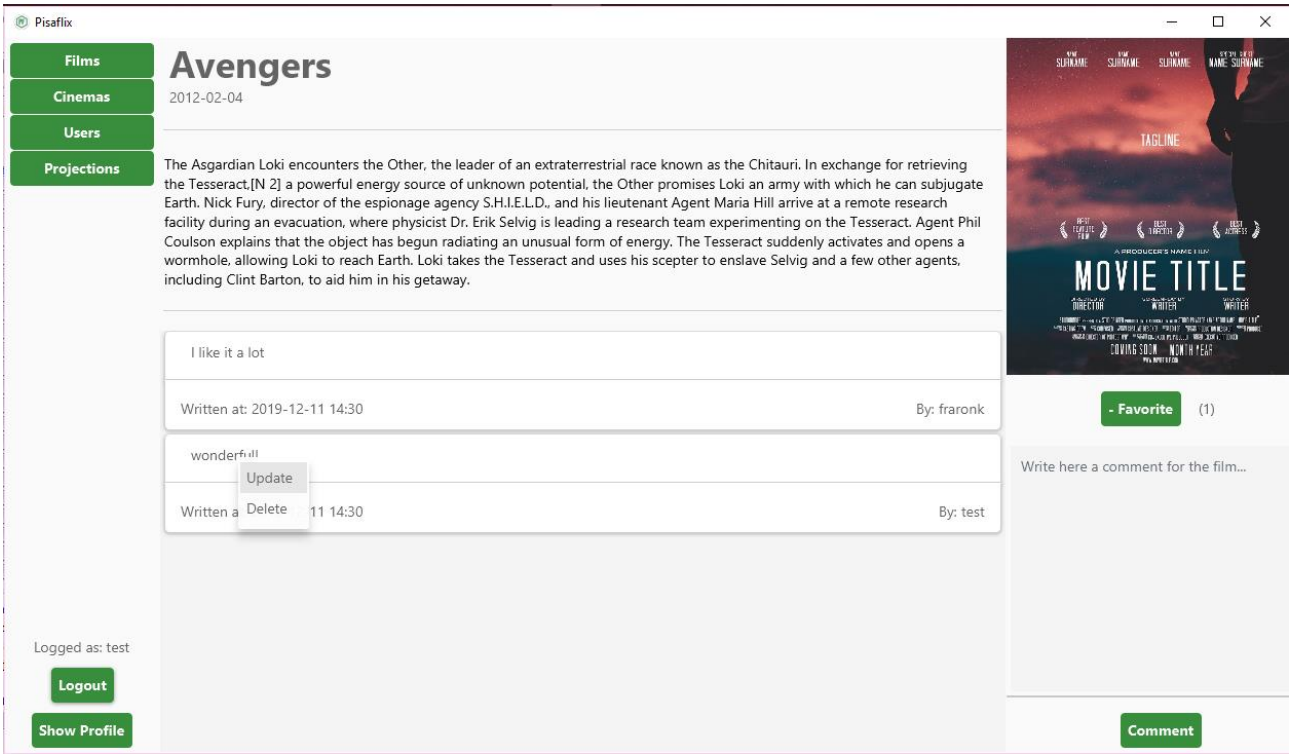
In the browse films/cinemas the user can search for a specific item filtering by title/name, if the user has the right privileges it can also add a new film/cinema (by clicking the “add” button in the top right corner) or modify/delete an existing one by right clicking on it ad select the wanted operation



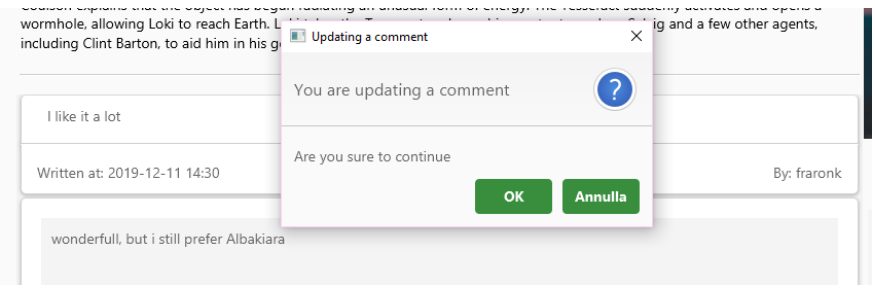
FILM/CINEMAS DITTAILS

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

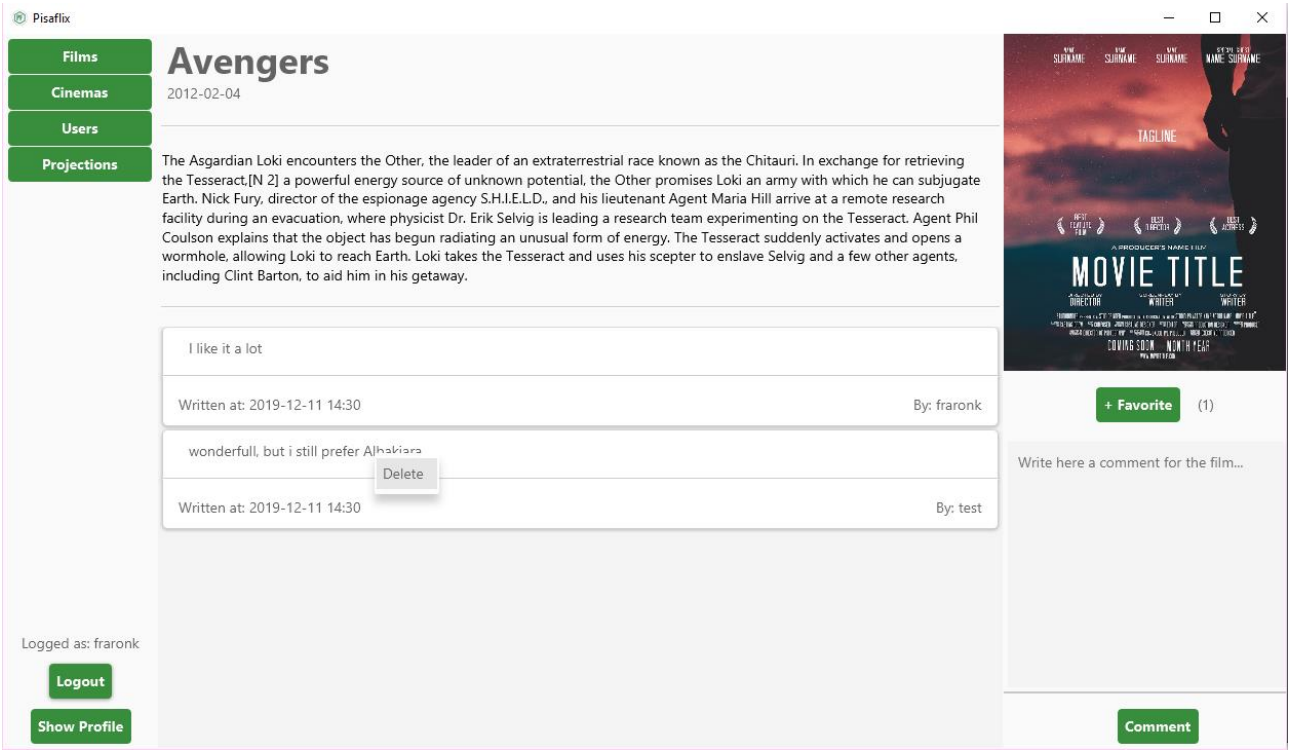
In that page an user, if logged, can add the film/cinema to its favourite (by clicking the apposite botton in the right side of the application) or comment it.



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

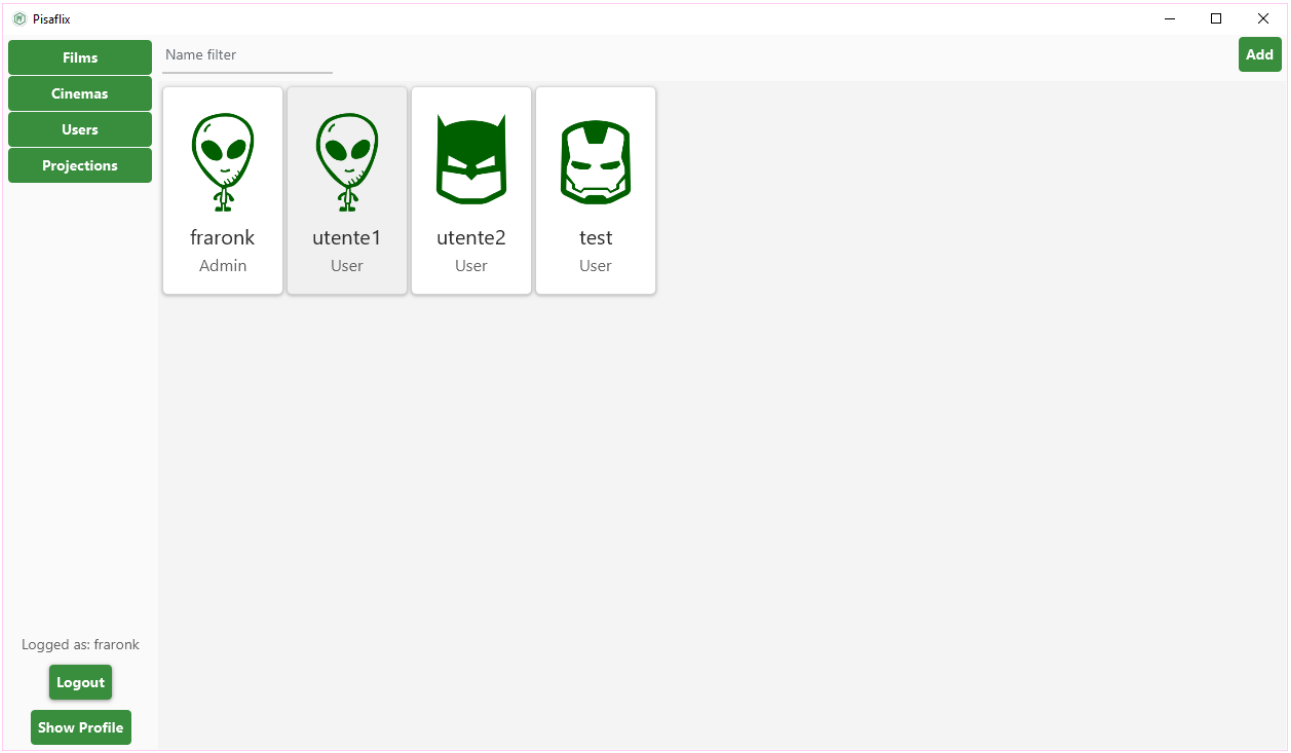


With the right privileges a user can also delete other users comments, in the same way of its

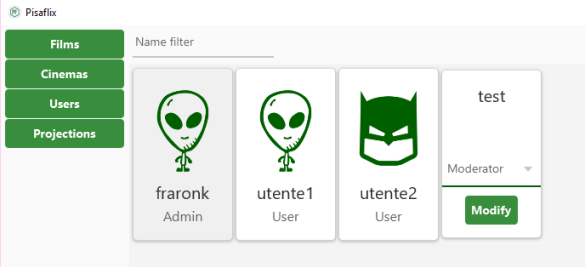


BROWSING USERS AND DETAILS

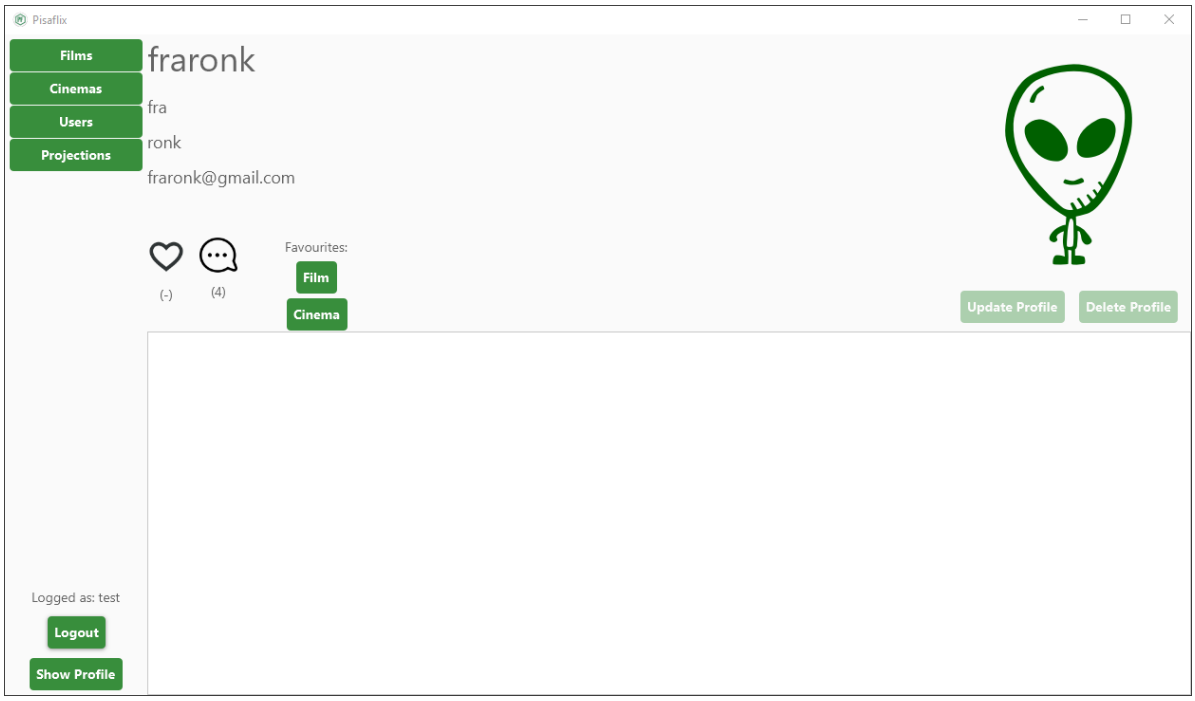
Similar to browse films/cinemas and user can also navigate through users by the apposite button in the top left corner, there it can see all usernames and privileges.



With the right privileges an user can modify others user privileges by right clicking on them ad use the apposite menu

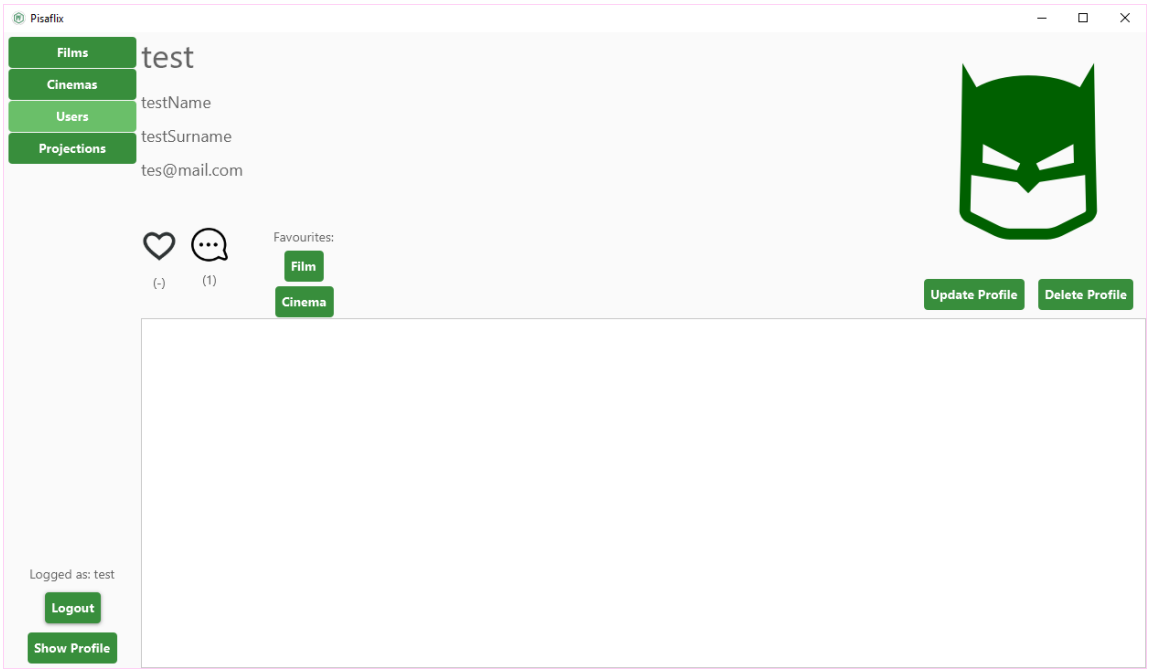


Once the user click on a user while browsing it will open its page detail

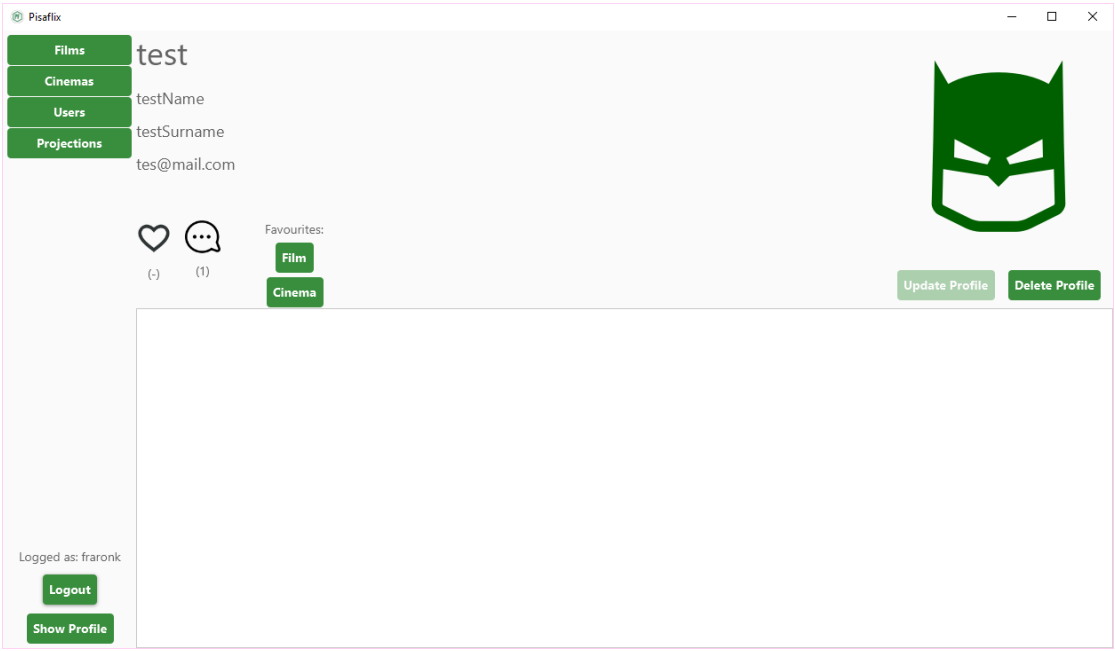


In the detail page it's visible how many favourite/comment a user did, refear to a film/cinema, and a list of favourite films/cinemas its avaible in that page.

When browsing the user can also click on its own detail page, then I can modify its information or delete its account (the same page it's accessible by the apposite button in the buttom left corner after the login)

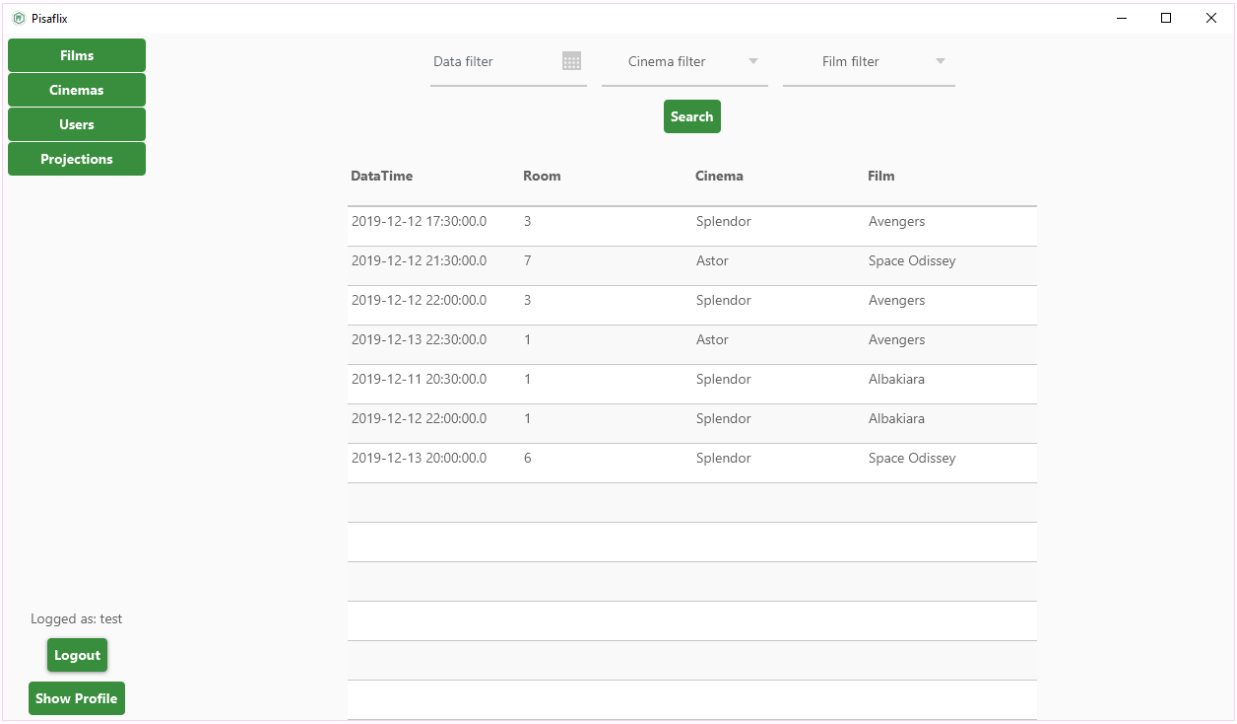


With the right privileges oce it open an other user detail page, the user can have the possibility to delete other user account



PROJECTION

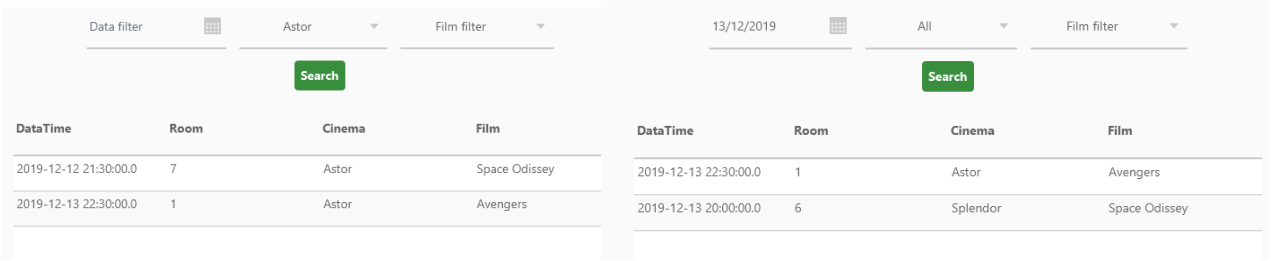
By clicking the apposite button in top left corner, the application will show the projection page on which the user can se the all the projections avaible.

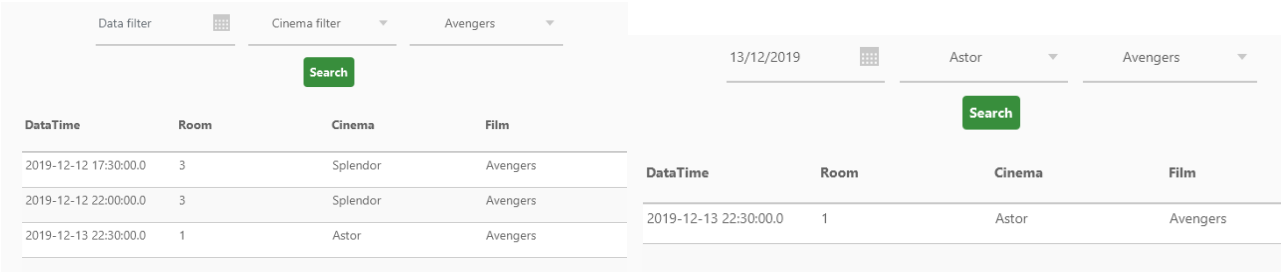


On the top of the page there are 3 filters that user can use to filters the projections

- By Date
- By Cinema
- By Film

The user can use a composition of above in order to make a more specific search





With the right privileges the user can also remove a projection or add a new one, with the apposite buttons that will appear next to the search button