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

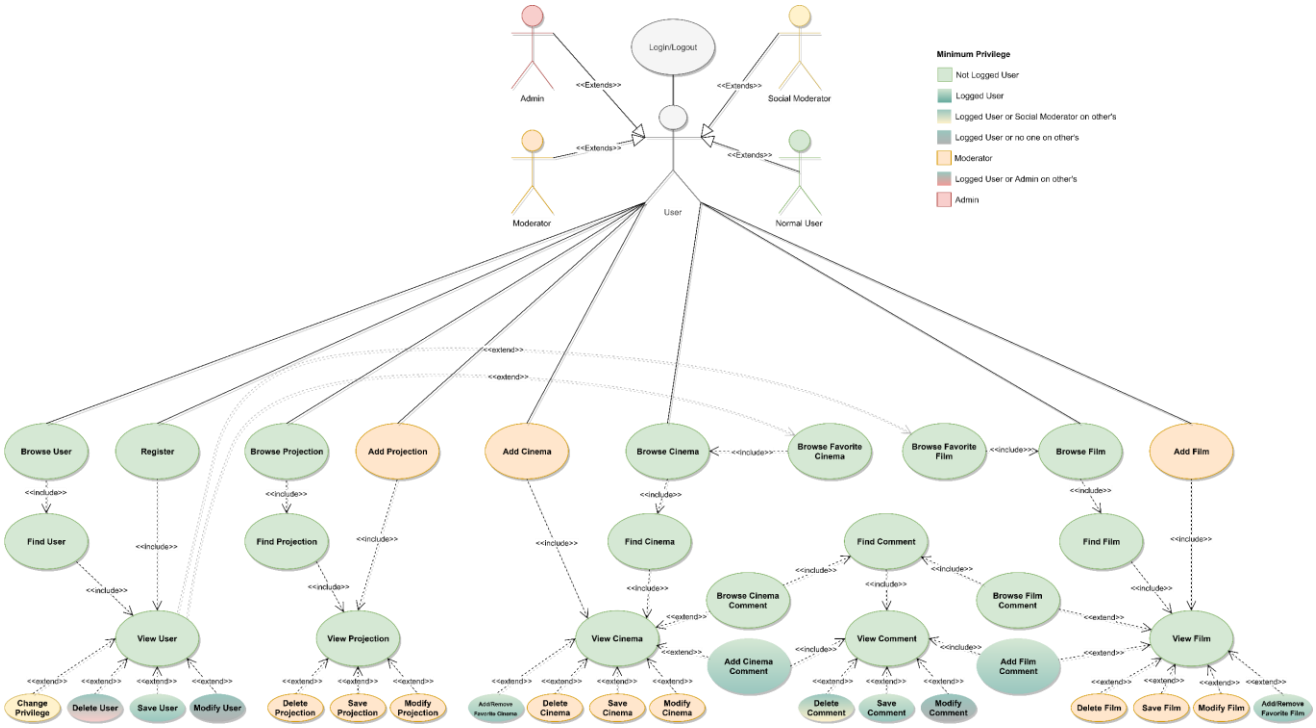
1. *Users* can **view** the list of **Movies/Cinemas** available on the platform.
2. *Users* can **view** the specific information about a *Movie* (es. category, publish date ecc…).
3. *Users* can **view** the specific information about a *Cinema* (es. Name, Address).
4. *Users* can **view** the *Projections* scheduled in a *Cinema*.
5. *Users* can **view** the *Projections* scheduled for a *Film*.
6. *Users* can **view** the list of ***favourites*** of other users (including himself).
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 **favourite** 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** other users' comments.
    b. If logged as *Social moderator* can **recruit** others *Social moderator*s.

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

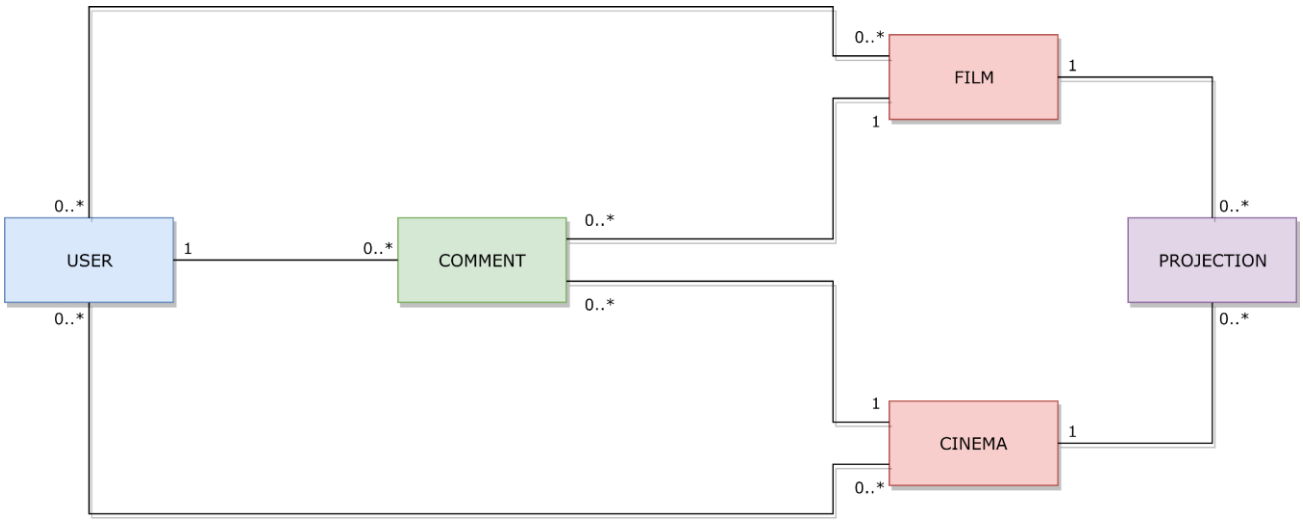
NON-FUNCTIONAL

1. The systems must be on 24/24.
2. The system must support hundreds 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



ANALYSIS CLASSES

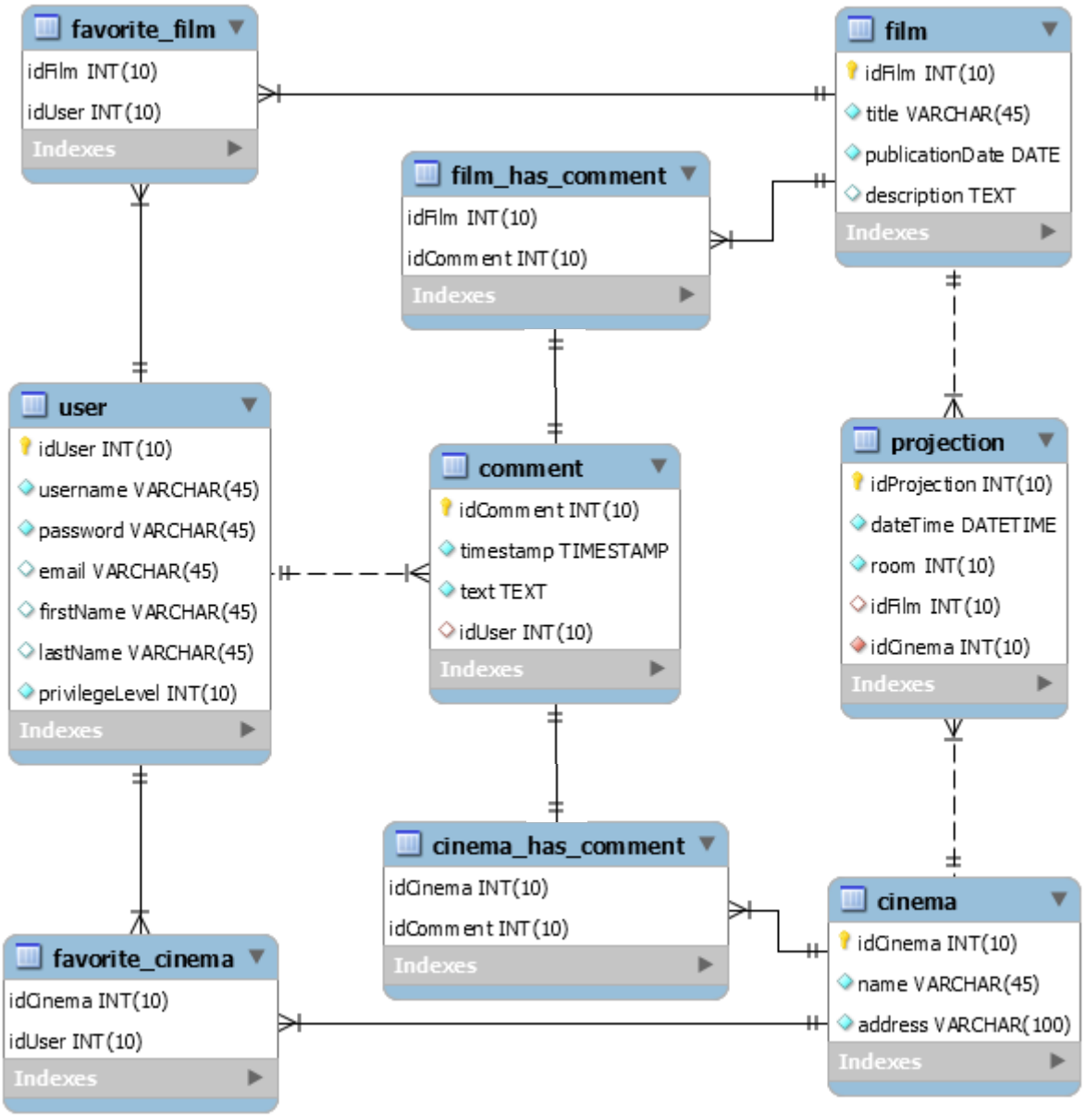


PROJECT DOCUMENT

E-R DIAGRAM

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

The ***Database*** has the following schema:



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.

APPLICATION 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 **MongoDB driver** to manage *back-end* functionalities. **Services** and ***JavaBean* objects** compose the *middleware* infrastructure that connect *front-end* and *back-end.*

INTERFACE DESIGN PATTERN

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

MODEL

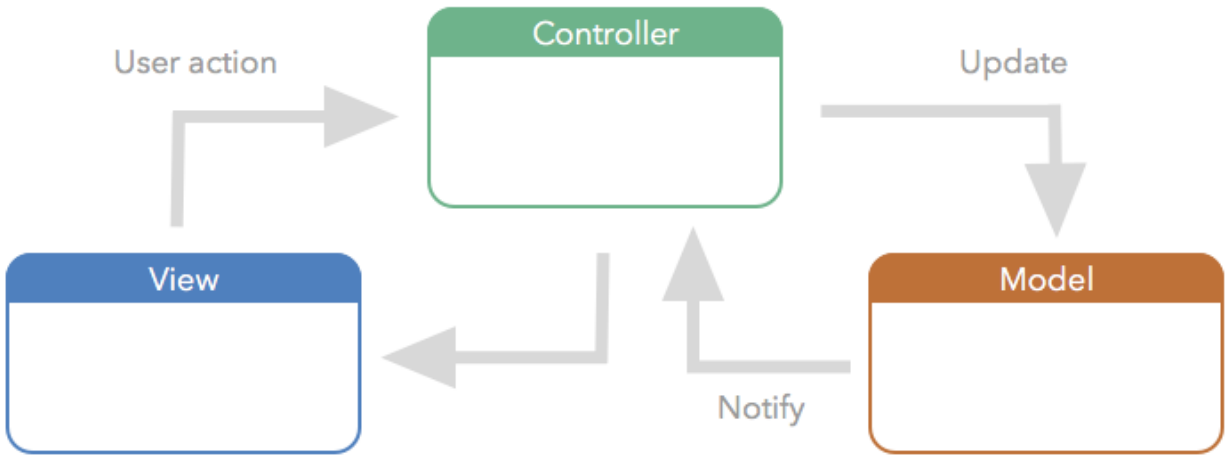
Services module represents the *model* and it's the central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages logic and rules of the application receiving inputs from the controller. The model is also responsible for managing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

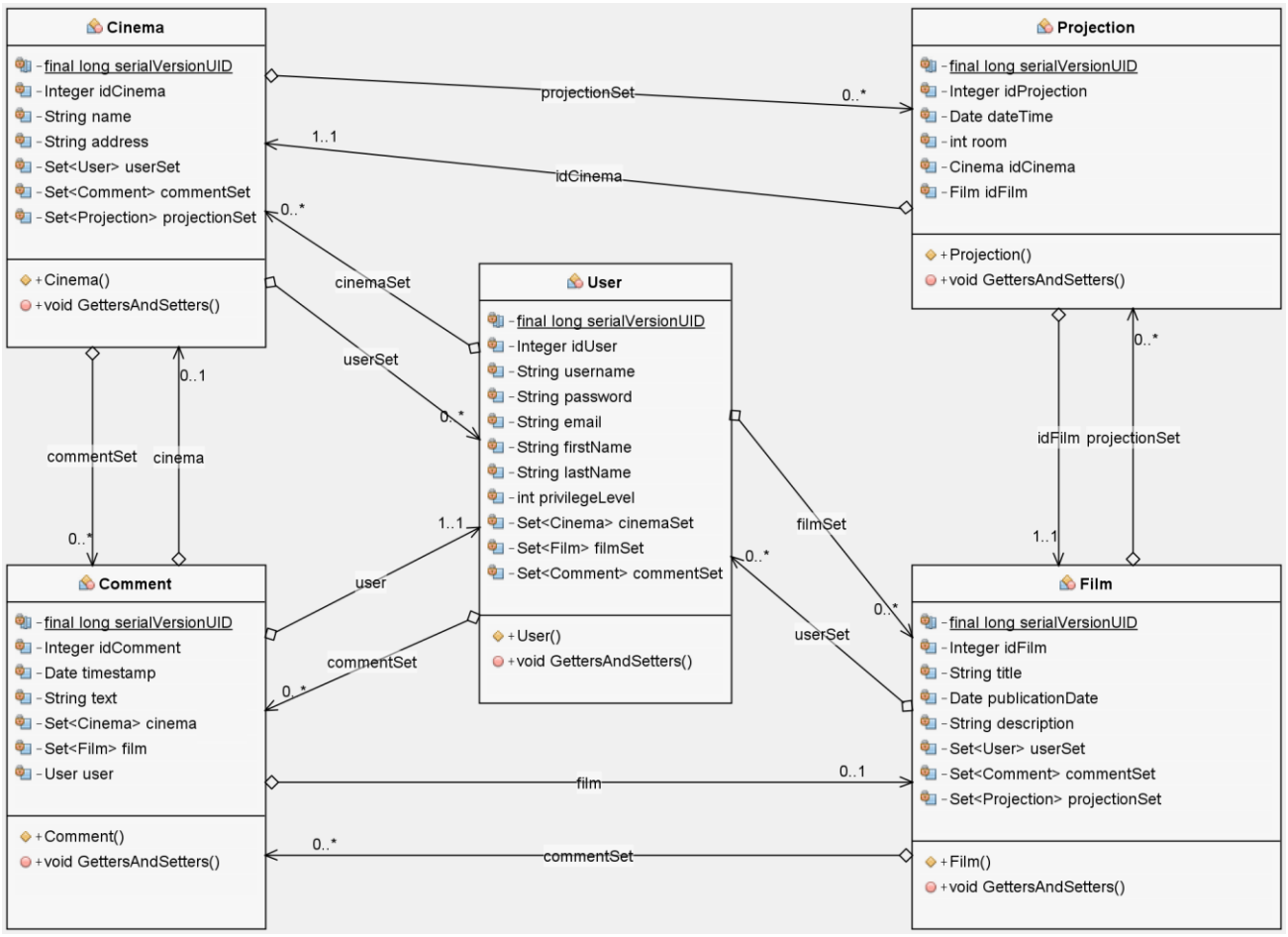
The **page controllers** are the *controller* of the application. They receive inputs from the *view* and convert them into commands for the *model* or *view* itself. Controllers can also validate inputs and data without the intervention of the *model*. Data is exchanged between *model* and *controller*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



SOFTWARE CLASSES

ENTITIES

Diagram of the **classes**:



USER

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

In ***cinemaSet*** the set of user favourite cinemas is saved, it allows to map the *many-to-many* relationship between entities *Cinema* and *User*, the same for ***filmSet***.
commentSet allows to map the *many-to-one* relationship between entities *Comment* and *User.*

The getters and the class constructor are the only functions present; the class code is shown below:

```java
//file User.java
@Entity
@Table(name = "User")
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "idUser")
    private Integer idUser;
```

```
11.
12.     @Basic(optional = false)
13.     @Column(name = "username")
14.     private String username;
15.
16.     @Basic(optional = false)
17.     @Column(name = "password")
18.     private String password;
19.
20.     @Column(name = "email")
21.     private String email;
22.
23.     @Column(name = "firstName")
24.     private String firstName;
25.
26.     @Column(name = "lastName")
27.     private String lastName;
28.
29.     @Basic(optional = false)
30.     @Column(name = "privilegeLevel")
31.     private int privilegeLevel;
32.
33.     @ManyToMany(mappedBy = "userSet", fetch = FetchType.EAGER,
                    cascade = {CascadeType.MERGE, CascadeType.PERSIST})
34.     private Set<Cinema> cinemaSet = new LinkedHashSet<>();
35.
36.     @ManyToMany(mappedBy = "userSet", fetch = FetchType.EAGER,
                    cascade = {CascadeType.MERGE, CascadeType.PERSIST})
37.     private Set<Film> filmSet = new LinkedHashSet<>();
38.
39.     @OneToMany(mappedBy = "user", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
40.     private Set<Comment> commentSet = new LinkedHashSet<>();
41.
42.     //GETTERS AND SETTERS
43. }
```

Explanation of the various *Hibernate* directives:

- `@Entity` and `@Table(name = "User")` indicate that the class represents the entity, and therefore also the table, User.
- The directive `@Id`, before `private Integer idUser`, specify that the field it's part of the *primary key.*
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` indicate that, if not set, the value for that field will be generate automatically and it will be unique.
- `@Basic(optional = false)` indicate that that field can't be null.
- `@Column(name = "idUser")` map the field "idUser" with respective field in the database table.
- (Line 33, 36) `@ManyToMany(…)` indicate a *many-to-many* association between Cinema (or Film) entities and User entities, mapped by the *usersSet* field in Cinema (or Film) class:
  - The *eager* fetch type indicate that the cinemas associated with the user are withdrawn immediately upon withdrawal of the user.
  - The *merge* cascade type indicate that *merge()* operations cascade to related entities.
  - The *persist* cascade type indicate that *save() or persist() operations cascade to related entities.*
- (Line 39) `@OneToMany(…)` indicate a *one-to-many* association between a User entity and Comment entities, mapped by the *user* field in Comment class:
  - The *eager* fetch type indicate that the comments associated with the user are withdrawn immediately upon withdrawal of the user.

- The *cascade all* type indicate that all operations cascade to related entities.

COMMENT

This entity class represents any **comment**, the comment *text*, *id* and *creation date* are saved inside it.

The sets ***cinema*** and ***film*** contain the class of the entity to which the comment refers, only *one* of the two sets contains a *single* entity at a time, it is necessary to use sets and map the relationship between comments and cinema or films as *many-to-many*, instead of *many-to-one*, due to the particular type of association that exists between the *comment* table, the *has_comment* support tables and the *film* and *cinema* tables. This allows to normalize the relationship as much as possible and to avoid that an unused field always exists within the comment table.

The getters and the class constructor are the only functions present; the class code is shown below:

```
1.  //file Comment.java
2.  @Entity
3.  @Table(name = "Comment")
4.  public class Comment implements Serializable {
5.
6.      @Id
7.      @GeneratedValue(strategy = GenerationType.IDENTITY)
8.      @Basic(optional = false)
9.      @Column(name = "idComment")
10.     private Integer idComment;
11.
12.     @Basic(optional = false)
13.     @Column(name = "timestamp")
14.     @Temporal(TemporalType.TIMESTAMP)
15.     private Date timestamp;
16.
17.     @Basic(optional = false)
18.     @Lob
19.     @Column(name = "text")
20.     private String text;
21.
22.     @JoinTable(name = "cinema_has_comment", joinColumns = {
23.                 @JoinColumn(name = "idComment", referencedColumnName = "idComment")},
                     inverseJoinColumns = { @JoinColumn(name = "idCinema",
                                                        referencedColumnName = "idCinema")})
24.     @ManyToMany(fetch = FetchType.EAGER)
25.     private Set<Cinema> idCinema = new LinkedHashSet<>();
26.     // it is necessary to use a Set to allow Hibernate to map the “cinema_has_comment”
           table, but it always contains at most one value (none if the comment refers to
           a film)
27.
28.     @JoinTable(name = "film_has_comment", joinColumns = {
29.                 @JoinColumn(name = "idComment", referencedColumnName = "idComment")},
                    inverseJoinColumns = { @JoinColumn(name = "idFilm",
                                                       referencedColumnName = "idFilm")})
30.     @ManyToMany(fetch = FetchType.EAGER)
31.     private Set<Film> idFilm = new LinkedHashSet<>();
32.     // it is necessary to use a Set to allow Hibernate to map the “film_has_comment”
           table, but it always contains at most one value (none if the comment refers to
           a cinema)
33.
34.     @JoinColumn(name = "idUser", referencedColumnName = "idUser")
35.     @ManyToOne(fetch = FetchType.EAGER)
36.     private User user;
37.
38.    //GETTERS AND SETTERS
39. }
```

Explanation of the various *Hibernate* directives:

- `@Entity` and `@Table(name = "Comment")` indicate that the class represents the entity, and therefore also the table, Comment.
- The directive `@Id`, before `private Integer idComment`, specify that the field it's part of the *primary key.*
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` indicate that, if not set, the value for that field will be generate automatically and it will be unique.
- `@Basic(optional = false)` indicate that that field can't be null.
- `@Column(name = "idComment")` map the field "idComment" with respective field in the database table.
- (Line 14) `@Temporal(TemporalType.TIMESTAMP)` indicate that the field is a timestamp.
- (Line 18) `@Lob` signifies that the annotated field should be represented as BLOB (binary data) in the DB.
- **(Line 22, 28)** `@JoinTable(…)` map a *many-to-many* association between Cinema (or Film) entities and Comment entities:
  - `name = "…_has_comment"` indicate that the association is done using "cinema_has_comment" (or "film_has_comment") intermediate table, a *many-to-many* relationship is used instead of *one-to-many* relationship to allow the use of this table, necessary to differentiate film comments from cinema comments.
  - `@JoinColumn` indicate the attributes on which to join the tables.
- (Line 24, 30) `@ManyToMany(fetch = FetchType.EAGER)` indicate a *many-to-many* association between Comment entities and Cinema (or Film) entities:
  - The *eager* fetch type indicate that the cinemas (or Films) associated with the comment are withdrawn immediately upon withdrawal of the comment.
- (Line 35) `@ManyToOne(fetch = FetchType.EAGER)` indicate a *many-to-one* association between Comment entities and a User entity:
  - The *eager* fetch type indicate that the user associated with the comment are withdrawn immediately upon withdrawal of the comment.
- (Line 34) `@JoinColumn(name = "idUser", referencedColumnName = "idUser")` indicate the attributes on which to join the tables.

PROJECTION

This entity class represents a **projection** of a specific movie (***idFilm***) scheduled in a specific cinema (***idCinema***) and contains the information about it.

The getters and the class constructor are the only functions present; the class code is shown below:

```
1.  //file Projection.java
2.  @Entity
3.  @Table(name = "Projection")
4.  public class Projection implements Serializable {
5.
6.      @Id
7.      @GeneratedValue(strategy = GenerationType.IDENTITY)
8.      @Basic(optional = false)
9.      @Column(name = "idProjection")
10.     private Integer idProjection;
```

```
11.
12.     @Basic(optional = false)
13.     @Column(name = "dateTime")
14.     @Temporal(TemporalType.TIMESTAMP)
15.     private Date dateTime;
16.
17.     @Basic(optional = false)
18.     @Column(name = "room")
19.     private int room;
20.
21.     @JoinColumn(name = "idCinema", referencedColumnName = "idCinema")
22.     @ManyToOne(optional = false, fetch = FetchType.EAGER)
23.     private Cinema idCinema;
24.
25.     @JoinColumn(name = "idFilm", referencedColumnName = "idFilm")
26.     @ManyToOne(fetch = FetchType.EAGER)
27.     private Film idFilm;
28.
29.     //GETTERS AND SETTERS
30. }
```

Explanation of the various *Hibernate* directives:

- `@Entity` and `@Table(name = "Projection")` indicate that the class represents the entity, and therefore also the table, Projection.
- The directive `@Id`, before `private Integer idProjection`, specify that the field it's part of the *primary key.*
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` indicate that, if not set, the value for that field will be generate automatically and it will be unique.
- `@Basic(optional = false)` indicate that that field can't be null.
- `@Column(name = "idProjection")` map the field "idProjection" with respective field in the database table.
- (Line 22, 26) `@ManyToOne(fetch = FetchType.EAGER)` indicate a *many-to-one* association between Projection entities and a Cinema (or Film) entity:
  - The *eager* fetch type indicate that the user associated with the comment are withdrawn immediately upon withdrawal of the comment.
- (Line 21, 25) `@JoinColumn(name = "idCinema(Film)", referencedColumnName = "idCinema(Film)")` indicate the attributes on which to join the tables.

CINEMA

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

The set ***userSet*** is used to map the relationship *many-to-many* with the users that have that cinema in their favourites. The sets ***commentSet*** and ***projectionSet*** are used to map the *many-to-one* relationship between the cinema and the comments referred to it and the projections scheduled on it.

The getters and the class constructor are the only functions present; the class code is shown below:

```
1.  //file Cinema.java
2.  @Entity
3.  @Table(name = "Cinema")
4.  public class Cinema implements Serializable {
5.
6.      @Id
```

```
7.      @GeneratedValue(strategy = GenerationType.IDENTITY)
8.      @Basic(optional = false)
9.      @Column(name = "idCinema")
10.     private Integer idCinema;
11.
12.     @Basic(optional = false)
13.     @Column(name = "name")
14.     private String name;
15.
16.     @Basic(optional = false)
17.     @Column(name = "address")
18.     private String address;
19.
20.     @JoinTable(name = "Favorite_Cinema", joinColumns = {
                    @JoinColumn(name = "idCinema", referencedColumnName = "idCinema")},
                    inverseJoinColumns = { @JoinColumn(name = "idUser",
                                                       referencedColumnName = "idUser")})
21.     @ManyToMany(fetch = FetchType.EAGER)
22.     private Set<User> userSet = new LinkedHashSet<>();
23.
24.     @ManyToMany(mappedBy = "idCinema", fetch = FetchType.EAGER,
                     cascade = CascadeType.ALL)
25.     @OrderBy
26.     private Set<Comment> commentSet = new LinkedHashSet<>();
27.
28.     @OneToMany(cascade = CascadeType.ALL, mappedBy = "idCinema", fetch = FetchType.EAGER)
29.     private Set<Projection> projectionSet = new LinkedHashSet<>();
30.
31.     //GETTERS AND SETTERS
32. }
```

Explanation of the various *Hibernate* directives:

- `@Entity` and `@Table(name = "Cinema")` indicate that the class represents the entity, and therefore also the table, Cinema.
- The directive `@Id`, before `private Integer idCinema`, specify that the field it's part of the *primary key.*
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` indicate that, if not set, the value for that field will be generate automatically and it will be unique.
- `@Basic(optional = false)` indicate that that field can't be null.
- `@Column(name = "idCinema")` map the field "idCinema" with respective field in the database table.
- (Line 20) `@JoinTable(…)` map a *many-to-many* association between Cinema entities and User entities:
  - `name = "Favorite_Cinema "` indicate that the association is done using that intermediate table.
  - `@JoinColumn` indicate the attributes on which to join the tables.
- (Line 21) `@ManyToMany(fetch = FetchType.EAGER)` indicate a *many-to-many* association between Cinema entities and User entities:
  - The *eager* fetch type indicate that the users associated with the cinema are withdrawn immediately upon withdrawal of the cinema.
- (Line 24) `@ManyToMany(…)` indicate a *many-to-many* association between Cinema entities and Comment entities, mapped by the *cinemaSet* field in Comment class:
  - The *eager* fetch type indicate that the comments associated with the cinema are withdrawn immediately upon withdrawal of the cinema.
  - The *cascade all* type indicate that all operations cascade to related entities.

- (Line 25) `@OrderBy` sorts comments from most recent to least recent within the set.
- (Line 28) `@OneToMany(…)` indicate a *one-to-many* association between Cinema entity and Projection entities, mapped by the *idCinema* field in Comment class:
  - The *eager* fetch type indicate that the projection associated with the cinema are withdrawn immediately upon withdrawal of the cinema.
  - The *cascade all* type indicate that all operations cascade to related entities.

FILM

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

The set ***userSet*** is used to map the relationship *many-to-many* with the users that have that movie in their favourites. The sets ***commentSet*** and ***projectionSet*** are used to map the *many-to-one* relationship between the movie and the comments referred to it and the projections of it.

The getters and the class constructor are the only functions present; the class code is shown below:

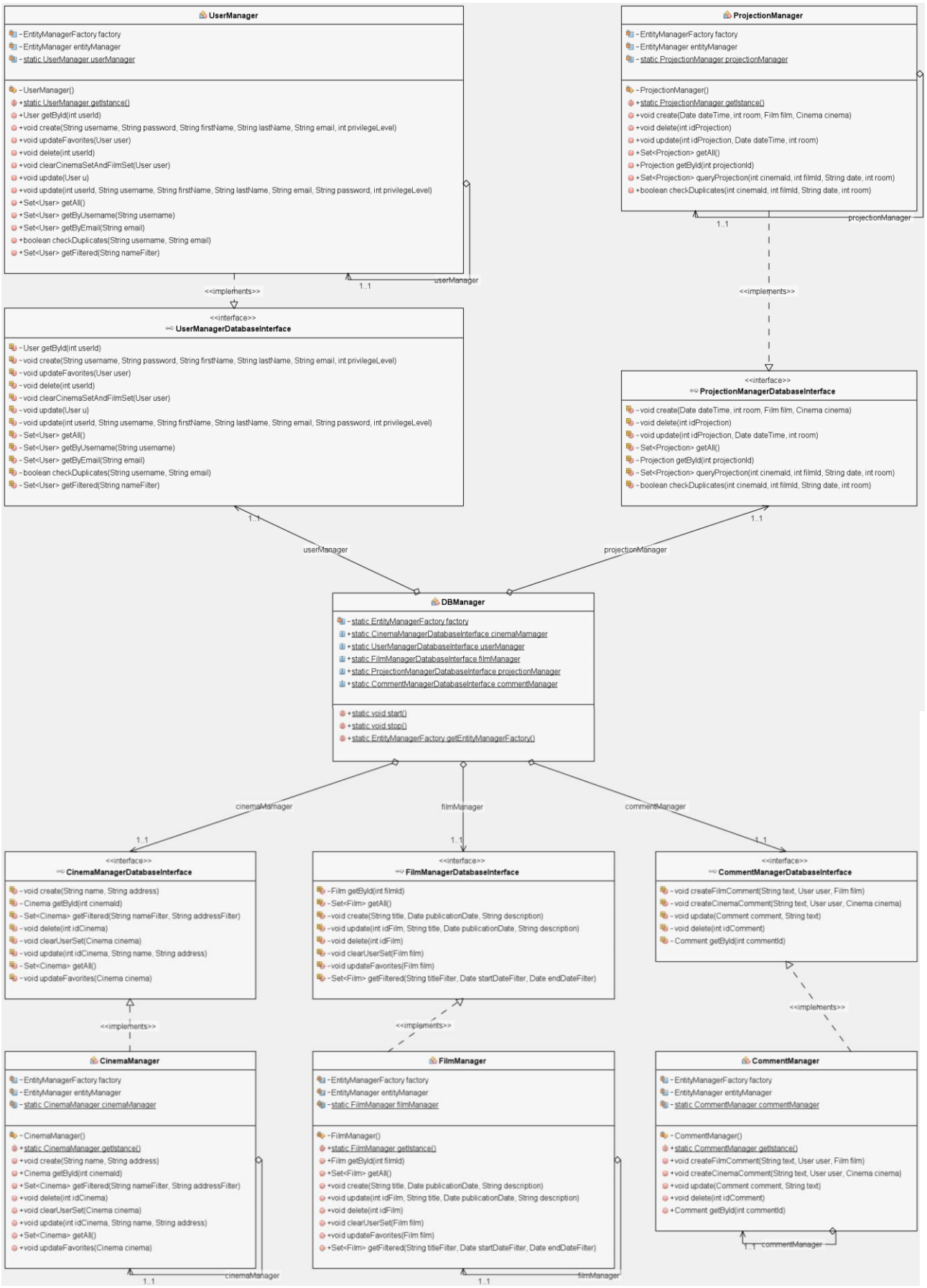
```
1.  //file Film.java
2.  @Entity
3.  @Table(name = "Film")
4.  public class Film implements Serializable {
5.
6.      @Id
7.      @GeneratedValue(strategy = GenerationType.IDENTITY)
8.      @Basic(optional = false)
9.      @Column(name = "idFilm")
10.     private Integer idFilm;
11.
12.     @Basic(optional = false)
13.     @Column(name = "title")
14.     private String title;
15.
16.     @Basic(optional = false)
17.     @Column(name = "publicationDate")
18.     @Temporal(TemporalType.DATE)
19.     private Date publicationDate;
20.
21.     @Lob
22.     @Column(name = "description")
23.     private String description;
24.
25.     @JoinTable(name = "Favorite_Film", joinColumns = {
26.                 @JoinColumn(name = "idFilm", referencedColumnName = "idFilm")},
                    inverseJoinColumns { @JoinColumn(name = "idUser",
                                                     referencedColumnName = "idUser")})
27.     @ManyToMany(fetch = FetchType.EAGER)
28.     private Set<User> userSet = new LinkedHashSet<>();
29.
30.     @ManyToMany(mappedBy = "idFilm", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
31.     @OrderBy
32.     private Set<Comment> commentSet = new LinkedHashSet<>();
33.
34.     @OneToMany(mappedBy = "idFilm", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
35.     private Set<Projection> projectionSet = new LinkedHashSet<>();
36.
37.     //GETTERS AND SETTERS
38. }
```

Explanation of the various *Hibernate* directives:

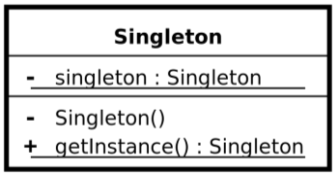
- `@Entity` and `@Table(name = "Film")` indicate that the class represents the entity, and therefore also the table, Film.
- The directive `@Id`, before `private Integer idFilm`, specify that the field it's part of the *primary key.*
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` indicate that, if not set, the value for that field will be generate automatically and it will be unique.
- `@Basic(optional = false)` indicate that that field can't be null.
- `@Column(name = "idFilm")` map the field "idFilm" with respective field in the database table.
- (Line 18) `@Temporal(TemporalType.DATE)` indicate that the field is a date.
- (Line 21) `@Lob` signifies that the annotated field should be represented as BLOB (binary data) in the DB.
- (Line 25) `@JoinTable(…)` map a *many-to-many* association between Film entities and User entities:
  - `name = "Favorite_Film "` indicate that the association is done using that intermediate table.
  - `@JoinColumn` indicate the attributes on which to join the tables.
- (Line 27) `@ManyToMany(fetch = FetchType.EAGER)` indicate a *many-to-many* association between Film entities and User entities:
  - The *eager* fetch type indicate that the users associated with the Film are withdrawn immediately upon withdrawal of the Film.
- (Line 30) `@ManyToMany(…)` indicate a *many-to-many* association between Film entities and Comment entities, mapped by the *filmSet* field in Comment class:
  - The *eager* fetch type indicate that the comments associated with the film are withdrawn immediately upon withdrawal of the film.
  - The *cascade all* type indicate that all operations cascade to related entities.
- (Line 31) `@OrderBy` sorts comments from most recent to least recent within the set.
- (Line 34) `@OneToMany(…)` indicate a *one-to-many* association between Film entity and Projection entities, mapped by the *idFilm* field in Comment class:
  - The *eager* fetch type indicate that the projection associated with the film are withdrawn immediately upon withdrawal of the film.
  - The *cascade all* type indicate that all operations cascade to related entities.

DB-MANAGER

The structure of **DBManager**:



All the managers are implemented following the software design pattern of ***singleton* pattern** which restricts the instantiation of a manager to one instance, also the *EntityFactoryManager* used by *Hibernate* and managed in the *DBManager* class follows this design pattern.



The main classes and functions are described below:

DB MANAGER

DBManager is an utility class, it's a *static* class that contains all the other manager specific for 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 *factory manager* of *Hibernate*.

USER MANAGER

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 user entities; the manager code is shown below:

```
1.  //file UserManager.java
2.  public class UserManager implements UserManagerDatabaseInterface {
3.
4.      //…
5.
6.      @Override
7.      public User getById(int userId) {
8.
9.          User user = null;
10.         try {
11.             entityManager = factory.createEntityManager();
12.             entityManager.getTransaction().begin();
13.             user = entityManager.find(User.class, userId);
14.
15.             if (user == null) {
16.                 System.out.println("User not found!");
```

```
            }

        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            ex.printStackTrace(System.out);
            System.out.println("A problem occurred in retriving a user!");
        } finally {
            entityManager.close();
        }

        return user;
    }

    @Override
    public void create(String username, String password, String firstName,
                       String lastName, String email, int privilegeLevel) {

        User user = new User();
        user.setUsername(username);
        user.setPassword(password);
        user.setPrivilegeLevel(privilegeLevel);
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setEmail(email);

        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();
            entityManager.persist(user);
            entityManager.getTransaction().commit();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            System.out.println("A problem occurred in creating the user!");
        } finally {
            entityManager.close();
        }

    }

    @Override
    public void updateFavorites(User user) {

        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();
            entityManager.merge(user);
            entityManager.getTransaction().commit();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            System.out.println("A problem occurred in updating favorites!");
        } finally {
            entityManager.close();
        }

    }

    @Override
    public void delete(int userId) {

        clearCinemaSetAndFilmSet(getById(userId));

        try {
            entityManager = factory.createEntityManager();
            entityManager.getTransaction().begin();
            User reference = entityManager.getReference(User.class, userId);
            entityManager.remove(reference);
            entityManager.getTransaction().commit();
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            System.out.println("A problem occurred in removing a User!");
        } finally {
```

```
87.             entityManager.close();
88.         }
89.
90.     }
91.
92.     @Override
93.     public void clearCinemaSetAndFilmSet(User user) {
94.
95.         user.setCinemaSet(new LinkedHashSet<>());
96.         user.setFilmSet(new LinkedHashSet<>());
97.
98.         try {
99.             entityManager = factory.createEntityManager();
100.            entityManager.getTransaction().begin();
101.            entityManager.merge(user);
102.            entityManager.getTransaction().commit();
103.        } catch (Exception ex) {
104.            System.out.println(ex.getMessage());
105.            System.out.println("A problem occurred clearing the user's cinemaset and
                                    filmset!");
106.         } finally {
107.            entityManager.close();
108.         }
109.
110.    }
111.
112.    @Override
113.    public void update(User u) {
114.        update(u.getIdUser(), u.getUsername(), u.getFirstName(), u.getLastName(),
                   u.getEmail(), u.getPassword(), u.getPrivilegeLevel());
115.    }
116.
117.    @Override
118.    public void update(int userId, String username, String firstName, String lastName,
                           tring email, String password, int privilegeLevel) {
119.
120.        User user = new User(userId);
121.        user.setUsername(username);
122.        user.setPassword(password);
123.        user.setFirstName(firstName);
124.        user.setLastName(lastName);
125.        user.setEmail(email);
126.        user.setPrivilegeLevel(privilegeLevel);
127.
128.        try {
129.            entityManager = factory.createEntityManager();
130.            entityManager.getTransaction().begin();
131.            entityManager.merge(user);
132.            entityManager.getTransaction().commit();
133.        } catch (Exception ex) {
134.            System.out.println(ex.getMessage());
135.            System.out.println("A problem occurred in updating a user!");
136.        } finally {
137.            entityManager.close();
138.        }
139.
140.    }
141.
142.    @Override
143.    public Set<User> getAll() {
144.
145.        System.out.println("Retrieving users");
146.        Set<User> users = null;
147.
148.        try {
149.            entityManager = factory.createEntityManager();
150.            entityManager.getTransaction().begin();
151.            users = new LinkedHashSet<>(entityManager.createQuery("FROM User")
                                                         .getResultList());
152.        if (users == null) {
153.            System.out.println("User is empty!");
```

```
154.         }
155.         } catch (Exception ex) {
156.             System.out.println(ex.getMessage());
157.             ex.printStackTrace(System.out);
158.             System.out.println("A problem occurred in retriving a user!");
159.         } finally {
160.             entityManager.close();
161.         }
162.         return users;
163.     }
164.
165.     @Override
166.     public Set<User> getByUsername(String username) {
167.
168.         Set<User> users = null;
169.
170.         try {
171.             entityManager = factory.createEntityManager();
172.             entityManager.getTransaction().begin();
173.             users = new LinkedHashSet<>(entityManager.createQuery("SELECT u FROM
                                                    User u WHERE u.username = '"
                                                    + username + "'").getResultList());
174.             if (users == null) {
175.                 System.out.println("Users is empty!");
176.             }
177.         } catch (Exception ex) {
178.             System.out.println(ex.getMessage());
179.             ex.printStackTrace(System.out);
180.             System.out.println("A problem occurred in retriving a user!");
181.         } finally {
182.             entityManager.close();
183.         }
184.         return users;
185.     }
186.
187.     @Override
188.     public Set<User> getByEmail(String email) {
189.         Set<User> users = null;
190.         try {
191.             entityManager = factory.createEntityManager();
192.             entityManager.getTransaction().begin();
193.             users = new LinkedHashSet<>(entityManager.createQuery("SELECT u FROM
                                                    User u WHERE u.email = '"
                                                    + email + "'").getResultList());
194.
195.             if (users == null) {
196.                 System.out.println("Not Found!");
197.             }
198.
199.         } catch (Exception ex) {
200.             System.out.println(ex.getMessage());
201.             ex.printStackTrace(System.out);
202.             System.out.println("A problem occurred in retriving a user!");
203.         } finally {
204.             entityManager.close();
205.         }
206.         return users;
207.     }
208.
209.     @Override
210.     public boolean checkDuplicates(String username, String email) {
211.         return !(getByUsername(username).isEmpty() && getByEmail(email).isEmpty());
212.     }
213.
214.     @Override
215.     public Set<User> getFiltered(String nameFilter) {
216.         Set<User> users = null;
217.         String name = "";
218.
219.         if (nameFilter != null) {
220.             name = nameFilter;
```

```
221.         }
222.
223.         String query = "SELECT u "
224.                 + "FROM User u "
225.                 + "WHERE ('" + name + "'='' OR u.username LIKE '%" + name + "%') ";
226.
227.         try {
228.             entityManager = factory.createEntityManager();
229.             entityManager.getTransaction().begin();
230.             users = new LinkedHashSet<>(entityManager.createQuery(query)
231.                                                      .getResultList());
232.             if (users == null) {
233.                 System.out.println("Users are empty!");
234.             }
235.         } catch (Exception ex) {
236.             System.out.println(ex.getMessage());
237.             System.out.println("A problem occurred in retrieve users filtered!");
238.         } finally {
239.             entityManager.close();
240.         }
241.
242.         return users;
243.     }
244. }
```

Explanation of the various Functions:

- User **getById**(int *userId*): get the user, with the id passed to the function, from the database.
- void **create**(String *username*, String *password*, String *firstName*, String *lastName*, String *email*, int *privilegeLevel*): inserts the user, with the fields passed to the function, into the database.
- void **updateFavorites**(User *user*): update the user's favourites by performing a merge on the passed entity, which contains an updated list of favourites.
- void **delete**(int *userId*): delete the user, with the id passed to the function, from the database.
- void **clearCinemaSetAndFilmSet**(User *user*): removes all cinemas and films from the user's favourites list, this operation is done "by hand" and not by using a cascade on the *remove()*, as this caused the cancellation also of the films and cinemas as well as the relationship with users.
- void **update**(User *u*): updates the user information through the fields inside the entity passed to the function.
- void **update**(int *userId*, String *username*, String *firstName*, String *lastName*, String *email*, String *password*, int *privilegeLevel*): updates the user information through the fields passed to the function.
- Set<User> **getAll**(): fetches all users from the database and returns them in a set.
- Set<User> **getByUsername**(String *username*): get the user, with the username passed to the function, from the database.
- Set<User> **getByEmail**(String *email*): get the user, with the email passed to the function, from the database.
- boolean **checkDuplicates**(String *username*, String *email*): check if there are already users in the database with the same username or the same email.

- 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.

FILM MANAGER

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 movie entities; the manager code is shown below:

```
1.  //file FilmManager.java
2.  public class FilmManager implements FilmManagerDatabaseInterface {
3.
4.      //…
5.
6.      @Override
7.      public Film getById(int filmId) {
8.
9.          Film film = null;
10.
11.         try {
12.             entityManager = factory.createEntityManager();
13.             entityManager.getTransaction().begin();
14.             film = entityManager.find(Film.class, filmId);
15.         } catch (Exception ex) {
16.             System.out.println(ex.getMessage());
17.             System.out.println("A problem occurred in retriving a film!");
18.         } finally {
19.             entityManager.close();
20.         }
21.
22.         return film;
23.     }
24.
25.     @Override
26.     public Set<Film> getAll() {
27.
28.         Set<Film> films = null;
29.
30.         try {
31.             entityManager = factory.createEntityManager();
32.             entityManager.getTransaction().begin();
33.             films = new LinkedHashSet<>(entityManager.createQuery("FROM Film")
                                                      .getResultList());
34.             if (films == null) {
35.                 System.out.println("Film is empty!");
36.             }
37.         } catch (Exception ex) {
38.             System.out.println(ex.getMessage());
39.             System.out.println("A problem occurred in retrieve all films!");
40.         } finally {
```

```
41.             entityManager.close();
42.         }
43.
44.         return films;
45.     }
46.
47.     @Override
48.     public void create(String title, Date publicationDate, String description) {
49.
50.         Film film = new Film();
51.         film.setTitle(title);
52.         film.setDescription(description);
53.         film.setPublicationDate(publicationDate);
54.
55.         try {
56.             entityManager = factory.createEntityManager();
57.             entityManager.getTransaction().begin();
58.             entityManager.persist(film);
59.             entityManager.getTransaction().commit();
60.         } catch (Exception ex) {
61.             System.out.println(ex.getMessage());
62.             System.out.println("A problem occurred in creating the film!");
63.         } finally {
64.             entityManager.close();
65.         }
66.
67.     }
68.
69.     @Override
70.     public void update(int idFilm, String title, Date publicationDate,
                           String description) {
71.
72.         Film film = new Film(idFilm);
73.         film.setTitle(title);
74.         film.setDescription(description);
75.         film.setPublicationDate(publicationDate);
76.
77.         try {
78.             entityManager = factory.createEntityManager();
79.             entityManager.getTransaction().begin();
80.             entityManager.merge(film);
81.             entityManager.getTransaction().commit();
82.         } catch (Exception ex) {
83.             System.out.println(ex.getMessage());
84.             System.out.println("A problem occurred in updating the film!");
85.         } finally {
86.             entityManager.close();
87.         }
88.
89.     }
90.
91.     @Override
92.     public void delete(int idFilm) {
93.
94.         clearUserSet(getById(idFilm));
95.
96.         try {
97.             entityManager = factory.createEntityManager();
98.             entityManager.getTransaction().begin();
99.             Film reference = entityManager.getReference(Film.class, idFilm);
100.            entityManager.remove(reference);
101.            entityManager.getTransaction().commit();
102.        } catch (Exception ex) {
103.            System.out.println(ex.getMessage());
104.            System.out.println("A problem occurred in deleting the film!");
105.        } finally {
106.            entityManager.close();
107.        }
108.
109.    }
110.
```

```
111.    @Override
112.    public void clearUserSet(Film film) {
113.
114.        film.setUserSet(new LinkedHashSet<>());
115.
116.        try {
117.            entityManager = factory.createEntityManager();
118.            entityManager.getTransaction().begin();
119.            entityManager.merge(film);
120.            entityManager.getTransaction().commit();
121.        } catch (Exception ex) {
122.            System.out.println(ex.getMessage());
123.            System.out.println("A problem occurred clearing the film's userset!");
124.        } finally {
125.            entityManager.close();
126.        }
127.
128.    }
129.
130.    @Override
131.    public void updateFavorites(Film film) {
132.
133.        try {
134.            entityManager = factory.createEntityManager();
135.            entityManager.getTransaction().begin();
136.            entityManager.merge(film);
137.            entityManager.getTransaction().commit();
138.        } catch (Exception ex) {
139.            System.out.println(ex.getMessage());
140.            System.out.println("A problem occurred updating favorite films!");
141.        } finally {
142.            entityManager.close();
143.        }
144.
145.    }
146.
147.    @Override
148.    public Set<Film> getFiltered(String titleFilter, Date startDateFilter,
                                    Date endDateFilter) {
149.
150.        Set<Film> films = null;
151.        String title = "";
152.        String startDate = "0000-01-01";
153.        String endDate = "9999-12-31";
154.
155.        if (titleFilter != null) {
156.            title = titleFilter;
157.        }
158.        if (startDateFilter != null) {
159.            startDate = startDateFilter.toString();
160.        }
161.        if (endDateFilter != null) {
162.            endDate = endDateFilter.toString();
163.        }
164.
165.        String query = "SELECT f "
166.                + "FROM Film f "
167.                + "WHERE ('" + title + "'='' OR f.title LIKE '%" + title + "%') "
168.                + "AND (publicationDate between '" + startDate + " 00:00:00' and '"
                    + endDate + " 23:59:59') ";
169.
170.        try {
171.            entityManager = factory.createEntityManager();
172.            entityManager.getTransaction().begin();
173.            films = new LinkedHashSet<>(entityManager.createQuery(query).getResultList());

174.            if (films == null) {
175.                System.out.println("Films are empty!");
176.            }
177.
178.        } catch (Exception ex) {
```

```
179.             System.out.println(ex.getMessage());
180.             ex.printStackTrace(System.out);
181.             System.out.println("A problem occurred in retrieve films filtered!");
182.         } finally {
183.             entityManager.close();
184.         }
185.
186.         return films;
187.
188.     }
189. }
```

Explanation of the various Functions:

- Film **getById**(int *filmId*): get the film, with the id passed to the function, from the database.
- Set<Film> **getAll**(): fetches all movies from the database and returns them in a set.
- void **create**(String *title*, Date *publicationDate*, String *description*): inserts the film, with the fields passed to the function, into the database.
- void **update**(int *idFilm*, String *title*, Date *publicationDate*, String *description*): updates the film information through the fields passed to the function.
- void **delete**(int *idFilm*): delete the film, with the id passed to the function, from the database.
- void **clearUserSet**(Film *film*): removes all users favourite relationship with that film, this operation is done "by hand" and not by using a cascade on the *remove()*, as this caused the cancellation also of the users as well as the relationship with films.
- void **updateFavorites**(Film *film*): update the users who have that film in their favourites by performing a merge on the passed entity, which contains an updated list of favourites.
- Set<Film> **getFiltered**(String titleFilter, Date startDateFilter, Date endDateFilter): search and returns all movies which have “titleFilter” in the title and with publication date between “startDateFilter” and “endDateFilter”. If some filters are not set, are not taken into consideration by the function, if all filter are not set it returns all movies.

CINEMA MANAGER

CinemaManagerDatabaseInterface is 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 cinema entities; the manager code is shown below:

```
1.  //file CinemaManager.java
2.  public class CinemaManager implements CinemaManagerDatabaseInterface {
3.
4.      //…
```

```
5.
6.      @Override
7.      public void create(String name, String address) {
8.          Cinema cinema = new Cinema();
9.          cinema.setName(name);
10.         cinema.setAddress(address);
11.         try {
12.             entityManager = factory.createEntityManager();
13.             entityManager.getTransaction().begin();
14.             entityManager.persist(cinema);
15.             entityManager.getTransaction().commit();
16.         } catch (Exception ex) {
17.             System.out.println(ex.getMessage());
18.             System.out.println("A problem occurred in creating the cinema!");
19.         } finally {
20.             entityManager.close();
21.         }
22.     }
23.
24.     @Override
25.     public Cinema getById(int cinemaId) {
26.         Cinema cinema = null;
27.         try {
28.             entityManager = factory.createEntityManager();
29.             entityManager.getTransaction().begin();
30.             cinema = entityManager.find(Cinema.class, cinemaId);
31.         } catch (Exception ex) {
32.             System.out.println(ex.getMessage());
33.             ex.printStackTrace(System.out);
34.             System.out.println("A problem occurred in retriving the cinema!");
35.         } finally {
36.             entityManager.close();
37.         }
38.         return cinema;
39.     }
40.
41.     @Override
42.     public Set<Cinema> getFiltered(String nameFilter, String addressFilter) {
43.         Set<Cinema> cinemas = null;
44.         String name = "";
45.         String address = "";
46.         if (nameFilter != null) {
47.             name = nameFilter;
48.         }
49.         if (addressFilter != null) {
50.             address = addressFilter;
51.         }
52.
53.         String query = "SELECT c "
54.                 + "FROM Cinema c "
55.                 + "WHERE ('" + name + "'='' OR c.name LIKE '%" + name + "%') "
56.                 + "AND ('" + address + "'='' OR c.address LIKE '%" + address + "%') ";
57.
58.         try {
59.             entityManager = factory.createEntityManager();
60.             entityManager.getTransaction().begin();
61.             cinemas = new LinkedHashSet<>(entityManager.createQuery(query)
                                                        .getResultList());
62.             if (cinemas == null) {
63.                 System.out.println("No cinema found!");
64.             }
65.         } catch (Exception ex) {
66.             System.out.println(ex.getMessage());
67.             ex.printStackTrace(System.out);
68.             System.out.println("A problem occurred in retrieve cinemas filtered!");
69.         } finally {
70.             entityManager.close();
71.         }
72.         return cinemas;
73.     }
74.
```

```
75.     @Override
76.     public void delete(int idCinema) {
77.         clearUserSet(getById(idCinema));
78.         try {
79.             entityManager = factory.createEntityManager();
80.             entityManager.getTransaction().begin();
81.             Cinema reference = entityManager.getReference(Cinema.class, idCinema);
82.             entityManager.remove(reference);
83.             entityManager.getTransaction().commit();
84.         } catch (Exception ex) {
85.             System.out.println(ex.getMessage());
86.             System.out.println("A problem occurred in removing a Cinema!");
87.         } finally {
88.             entityManager.close();
89.         }
90.     }
91.
92.     @Override
93.     public void clearUserSet(Cinema cinema) {
94.         cinema.setUserSet(new LinkedHashSet<>());
95.         try {
96.             entityManager = factory.createEntityManager();
97.             entityManager.getTransaction().begin();
98.             entityManager.merge(cinema);
99.             entityManager.getTransaction().commit();
100.        } catch (Exception ex) {
101.            System.out.println(ex.getMessage());
102.            System.out.println("A problem occurred clearing the
                                        cinema's userset!");
103.        } finally {
104.            entityManager.close();
105.        }
106.    }
107.
108.    @Override
109.    public void update(int idCinema, String name, String address) {
110.        Cinema cinema = new Cinema(idCinema);
111.        cinema.setName(name);
112.        cinema.setAddress(address);
113.        try {
114.            entityManager = factory.createEntityManager();
115.            entityManager.getTransaction().begin();
116.            entityManager.merge(cinema);
117.            entityManager.getTransaction().commit();
118.        } catch (Exception ex) {
119.            System.out.println(ex.getMessage());
120.            System.out.println("A problem occurred in updating the film!");
121.        } finally {
122.            entityManager.close();
123.        }
124.    }
125.
126.    @Override
127.    public Set<Cinema> getAll() {
128.        Set<Cinema> cinemas = null;
129.        try {
130.            entityManager = factory.createEntityManager();
131.            entityManager.getTransaction().begin();
132.            cinemas = new LinkedHashSet<>(entityManager.createQuery(
                                                    "FROM Cinema").getResultList());
133.            if (cinemas == null) {
134.                System.out.println("Cinema is empty!");
135.            }
136.        } catch (Exception ex) {
137.            System.out.println(ex.getMessage());
138.            ex.printStackTrace(System.out);
139.            System.out.println("A problem occurred in retrieve all cinemas!");
140.        } finally {
141.            entityManager.close();
142.        }
143.        return cinemas;
```

```
144.     }
145.
146.     @Override
147.     public void updateFavorites(Cinema cinema) {
148.         try {
149.             entityManager = factory.createEntityManager();
150.             entityManager.getTransaction().begin();
151.             entityManager.merge(cinema);
152.             entityManager.getTransaction().commit();
153.         } catch (Exception ex) {
154.             System.out.println(ex.getMessage());
155.             System.out.println("A problem occurred updating favorite cinemas!");
156.         } finally {
157.             entityManager.close();
158.         }
159.     }
160.
161.}
```

Explanation of the various Functions:

- void **create**(String *name*, String *address*): inserts the cinema, with the fields passed to the function, into the database.
- Cinema **getById**(int *cinemaId*): get the cinema, with the id passed to the function, from the database.
- Set<Cinema> **getFiltered**(String *nameFilter*, String *addressFilter*): search and returns all cinemas which have "*nameFilter*" in the name and the "*addressFilter*" in the address. If some filters are not set, are not taken into consideration by the function, if all filter are not set it returns all movies.
- void **delete**(int *idCinema*): delete the cinema, with the id passed to the function, from the database.
- void **clearUserSet**(Cinema *cinema*): removes all users favourite relationship with that cinema, this operation is done "by hand" and not by using a cascade on the *remove()*, as this caused the cancellation also of the users as well as the relationship with cinemas.
- void **update**(int *idCinema*, String *name*, String *address*): updates the cinema information through the fields passed to the function.
- Set<Cinema> **getAll**(): fetches all cinemas from the database and returns them in a set.
- void **updateFavorites**(Cinema *cinema*): update the users who have that cinema in their favourites by performing a merge on the passed entity, which contains an updated list of favourites.

PROJECTION MANAGER

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 projection entities; the manager code is shown below:

```
1.  //file ProjectionManager.java
2.  public class ProjectionManager implements ProjectionManagerDatabaseInterface {
3.
4.      //…
5.
6.      @Override
7.      public void create(Date dateTime, int room, Film film, Cinema cinema) {
8.
9.          Projection projection = new Projection();
10.         projection.setDateTime(dateTime);
11.         projection.setRoom(room);
12.         projection.setIdCinema(cinema);
13.         projection.setIdFilm(film);
14.
15.         cinema.getProjectionSet().add(projection);
16.         film.getProjectionSet().add(projection);
17.
18.         try {
19.             entityManager = factory.createEntityManager();
20.             entityManager.getTransaction().begin();
21.             entityManager.persist(projection);
22.             entityManager.merge(cinema);
23.             entityManager.merge(film);
24.             entityManager.getTransaction().commit();
25.         } catch (Exception ex) {
26.             System.out.println(ex.getMessage());
27.             System.out.println("A problem occurred in creating the projection!");
28.         } finally {
29.             entityManager.close();
30.         }
31.     }
32.
33.     @Override
34.     public void delete(int idProjection) {
35.
36.         try {
37.             entityManager = factory.createEntityManager();
38.             entityManager.getTransaction().begin();
39.             Projection reference = entityManager.getReference(Projection.class,
                                                                  idProjection);
40.             reference.getIdCinema().getProjectionSet().remove(reference);
41.             reference.getIdFilm().getProjectionSet().remove(reference);
42.             entityManager.remove(reference);
43.             entityManager.getTransaction().commit();
44.         } catch (Exception ex) {
45.             System.out.println(ex.getMessage());
46.             System.out.println("A problem occurred in removing the Projection!");
47.         } finally {
48.             entityManager.close();
49.         }
50.
51.     }
52.
53.     @Override
54.     public void update(int idProjection, Date dateTime, int room) {
55.
56.         Projection projection = new Projection(idProjection);
57.         projection.setDateTime(dateTime);
58.         projection.setRoom(room);
59.
60.         try {
61.             entityManager = factory.createEntityManager();
62.             entityManager.getTransaction().begin();
63.             entityManager.merge(projection);
64.             entityManager.getTransaction().commit();
```

```
65.         } catch (Exception ex) {
66.             System.out.println(ex.getMessage());
67.             System.out.println("A problem occurred in updating the projection!");
68.         } finally {
69.             entityManager.close();
70.         }
71.
72.     }
73.
74.     @Override
75.     public Set<Projection> getAll() {
76.         Set<Projection> projections = null;
77.         try {
78.             entityManager = factory.createEntityManager();
79.             entityManager.getTransaction().begin();
80.             projections = new LinkedHashSet<>(entityManager.createQuery("FROM projection")
                                                                .getResultList());
81.             if (projections == null) {
82.                 System.out.println("Projection is empty!");
83.             }
84.         } catch (Exception ex) {
85.             System.out.println(ex.getMessage());
86.             System.out.println("A problem occurred in retrieve all projections!");
87.         } finally {
88.             entityManager.close();
89.         }
90.         return projections;
91.     }
92.
93.     @Override
94.     public Projection getById(int projectionId) {
95.         Projection projection = null;
96.         try {
97.             entityManager = factory.createEntityManager();
98.             entityManager.getTransaction().begin();
99.             projection = entityManager.find(Projection.class, projectionId);
100.        } catch (Exception ex) {
101.            System.out.println(ex.getMessage());
102.            ex.printStackTrace(System.out);
103.            System.out.println("A problem occurred in retriving a projection!");
104.        } finally {
105.            entityManager.close();
106.        }
107.        return projection;
108.    }
109.
110.    @Override
111.    public Set<Projection> queryProjection(int cinemaId, int filmId, String date,
                                               int room) {

112.        Set<Projection> projections = null;
113.
114.        String query = "SELECT p "
115.                + "FROM Projection p "
116.                + "WHERE (("
                           + cinemaId + " = -1) OR ( " + cinemaId + " = p.idCinema)) "
117.                + "AND ((" + filmId + " = -1) OR ( " + filmId + " = p.idFilm)) "
118.                + "AND (('" + date + "' = 'all') OR dateTime between '"
                           + date + " 00:00:00' and '" + date + " 23:59:59') "
119.                + "AND ((" + room + " = -1) OR ( " + room + " = p.room)) ";
120.
121.        try {
122.            entityManager = factory.createEntityManager();
123.            entityManager.getTransaction().begin();
124.            projections = new LinkedHashSet<>(entityManager.createQuery(query)
                                                                .getResultList());
125.        } catch (Exception ex) {
126.            System.out.println(ex.getMessage());
127.            System.out.println("A problem occurred in retriving the projections!");
128.        } finally {
129.            entityManager.close();
```

```
130.         }
131.
132.         return projections;
133.     }
134.
135.     @Override
136.     public boolean checkDuplicates(int cinemaId, int filmId, String date, int room) {
137.
138.         Set<Projection> projections = null;
139.
140.         String query = "SELECT p "
141.                 + "FROM Projection p "
142.                 + "WHERE ((" + cinemaId
                            + " = -1) OR ( " + cinemaId + " = p.idCinema)) "
143.                 + "AND ((" + filmId + " = -1) OR ( " + filmId + " = p.idFilm)) "
144.                 + "AND (('" + date + "' = 'all') OR dateTime = '" + date + "') "
145.                 + "AND ((" + room + " = -1) OR ( " + room + " = p.room)) ";
146.
147.         try {
148.             entityManager = factory.createEntityManager();
149.             entityManager.getTransaction().begin();
150.             projections = new LinkedHashSet<>(entityManager.createQuery(query)
                                                                    .getResultList());
151.         } catch (Exception ex) {
152.             System.out.println(ex.getMessage());
153.             System.out.println("A problem occurred in checking duplicates!");
154.         } finally {
155.             entityManager.close();
156.         }
157.
158.         return !projections.isEmpty();
159.
160.     }
161. }
```

Explanation of the various Functions:

- void **create**(Date *dateTime*, int *room*, Film *film*, Cinema *cinema*): inserts the projection, with the fields passed to the function, into the database.
- void **delete**(int *idProjection*): delete the projection, with the id passed to the function, from the database.
- void **update**(int *idProjection*, Date *dateTime*, int *room*): updates the projection information through the fields passed to the function.
- Set<Projection> **getAll**(): fetches all projections from the database and returns them in a set.
- Projection **getById**(int *projectionId*): get the projection, with the id passed to the function, from the database.
- Set<Projection> **queryProjection**(int *cinemaId*, int *filmId*, String *date*, int *room*): search and returns all projections for cinema specified by "*cinemaId*" and the film specified by "*filmId",* it also take in consideration the date specified by "*date"* and the room specified by "*room"*. If some filters are not set, are not taken into consideration by the function, if all filter are not set it returns all movies.
- boolean **checkDuplicates**(int *cinemaId*, int *filmId*, String *date*, int *room*): checks the presence of a projection with the same information already present in the database
- *cinemaId*, int *filmId*, *String date*, int *room*) which.

COMMENT MANAGER

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; the manager code is shown below:

```
1.  //file CommentManager.java
2.  public class CommentManager implements CommentManagerDatabaseInterface {
3.
4.  //..
5.
6.      @Override
7.      public void createFilmComment(String text, User user, Film film) {
8.
9.          Comment comment = new Comment();
10.         comment.setText(text);
11.         comment.setTimestamp(new Date());
12.
13.         user.getCommentSet().add(comment);
14.         film.getCommentSet().add(comment);
15.         Set<Film> filmSet = new LinkedHashSet<>();
16.         filmSet.add(film);
17.         comment.setIdFilm(filmSet);
18.         comment.setUser(user);
19.
20.         try {
21.             entityManager = factory.createEntityManager();
22.             entityManager.getTransaction().begin();
23.             entityManager.persist(comment);
24.             entityManager.getTransaction().commit();
25.         } catch (Exception ex) {
26.             System.out.println(ex.getMessage());
27.             System.out.println("A problem occurred in creating the comment!");
28.         } finally {
29.             entityManager.close();
30.         }
31.
32.     }
33.
34.     @Override
35.     public void createCinemaComment(String text, User user, Cinema cinema) {
36.
37.         Comment comment = new Comment();
38.         comment.setText(text);
39.         comment.setTimestamp(new Date());
40.
41.         user.getCommentSet().add(comment);
42.         cinema.getCommentSet().add(comment);
43.         Set<Cinema> cinemaSet = new LinkedHashSet<>();
44.         cinemaSet.add(cinema);
45.         comment.setIdCinema(cinemaSet);
46.         comment.setUser(user);
47.
48.         try {
49.             entityManager = factory.createEntityManager();
50.             entityManager.getTransaction().begin();
51.             entityManager.persist(comment);
```

```
52.             entityManager.getTransaction().commit();
53.         } catch (Exception ex) {
54.             System.out.println(ex.getMessage());
55.             System.out.println("A problem occurred in creating the comment!");
56.         } finally {
57.             entityManager.close();
58.         }
59.
60.     }
61.
62.     @Override
63.     public void update(Comment comment, String text) {
64.
65.         comment.setText(text);
66.
67.         try {
68.             entityManager = factory.createEntityManager();
69.             entityManager.getTransaction().begin();
70.             entityManager.merge(comment);
71.             entityManager.getTransaction().commit();
72.         } catch (Exception ex) {
73.             System.out.println(ex.getMessage());
74.             System.out.println("A problem occurred in updating the Comment!");
75.         } finally {
76.             entityManager.close();
77.         }
78.     }
79.
80.     @Override
81.     public void delete(int idComment) {
82.
83.         try {
84.             entityManager = factory.createEntityManager();
85.             entityManager.getTransaction().begin();
86.             Comment reference = entityManager.getReference(Comment.class, idComment);
87.             if (!reference.getIdCinema().isEmpty()) {
88.                 reference.getIdCinema().iterator().next().getCommentSet()
                                                            .remove(reference);
89.             }
90.             if (!reference.getIdFilm().isEmpty()) {
91.                 reference.getIdFilm().iterator().next().getCommentSet().remove(reference);

92.             }
93.             reference.getUser().getCommentSet().remove(reference);
94.             entityManager.remove(reference);
95.             entityManager.getTransaction().commit();
96.         } catch (Exception ex) {
97.             System.out.println(ex.getMessage());
98.             System.out.println("A problem occurred in removing the Comment!");
99.         } finally {
100.            entityManager.close();
101.        }
102.
103.    }
104.
105.    @Override
106.    public Comment getById(int commentId) {
107.
108.        Comment comment = null;
109.
110.        try {
111.            entityManager = factory.createEntityManager();
112.            entityManager.getTransaction().begin();
113.            comment = entityManager.find(Comment.class, commentId);
114.        } catch (Exception ex) {
115.            System.out.println(ex.getMessage());
116.            System.out.println("A problem occurred in retriving a comment!");
117.        } finally {
118.            entityManager.close();
119.        }
120.
```

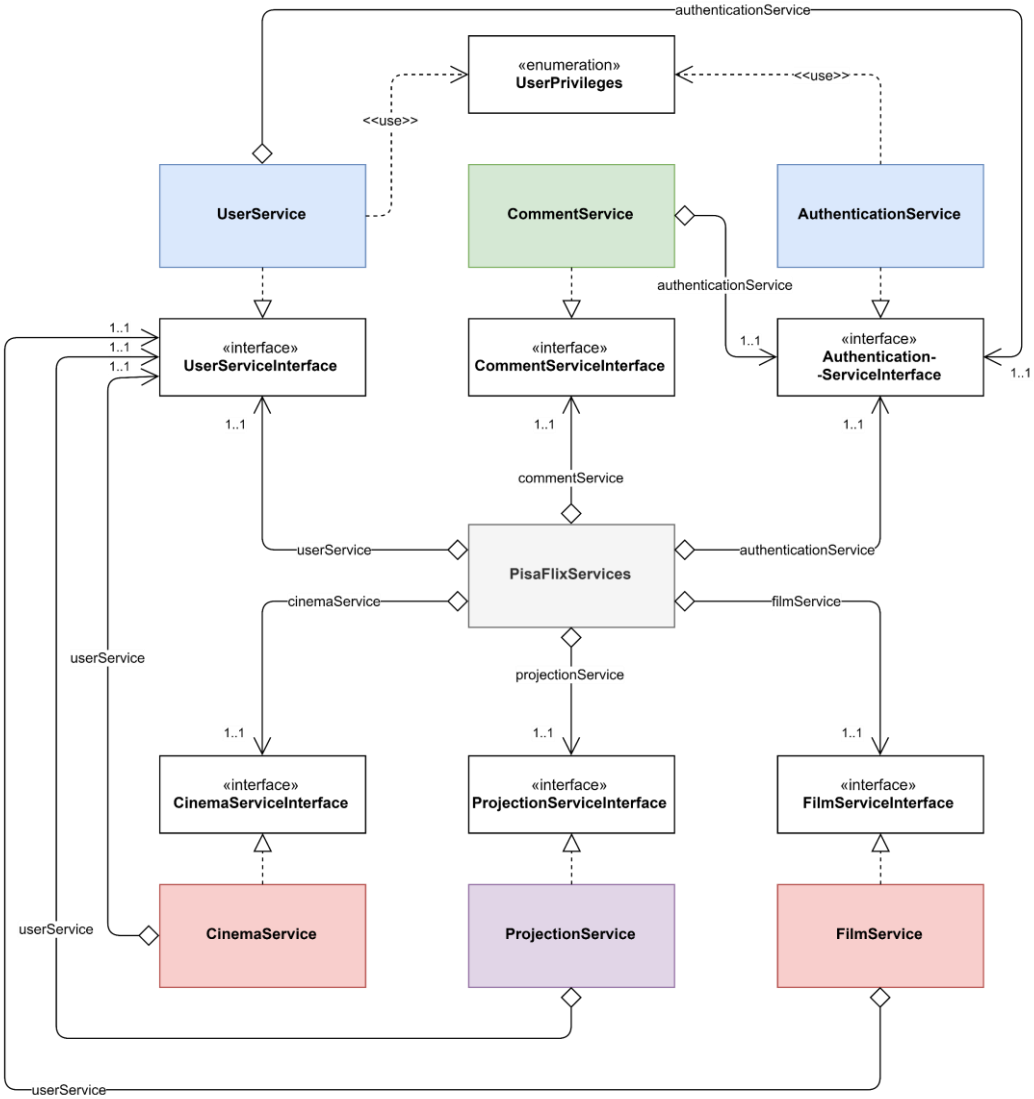
```
121.         return comment;
122.
123.     }
124.
125. }
```

Explanation of the various Functions:

- void **createFilmComment**(String *text*, User *user*, Film *film*): inserts the comment, with the fields passed to the function, into the database.
- void **createCinemaComment**(String *text*, User *user*, Cinema *cinema*): inserts the comment, with the fields passed to the function, into the database.
- void **update**(Comment *comment*, String *text*): updates the comment information through the fields passed to the function.
- void **delete**(int *idComment*): delete the comment, with the id passed to the function, from the database.
- Comment **getById**(int *commentId*): get the comment, with the id passed to the function, from the database.

PISAFLIX-SERVICES

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



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

The main classes and functions are described below:

- **PisaFlixServices** is a utility class, it's a static class that contains all the other service 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 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 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 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 depending by the operation that we want to 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 a user in the database with new information specify by its parameter

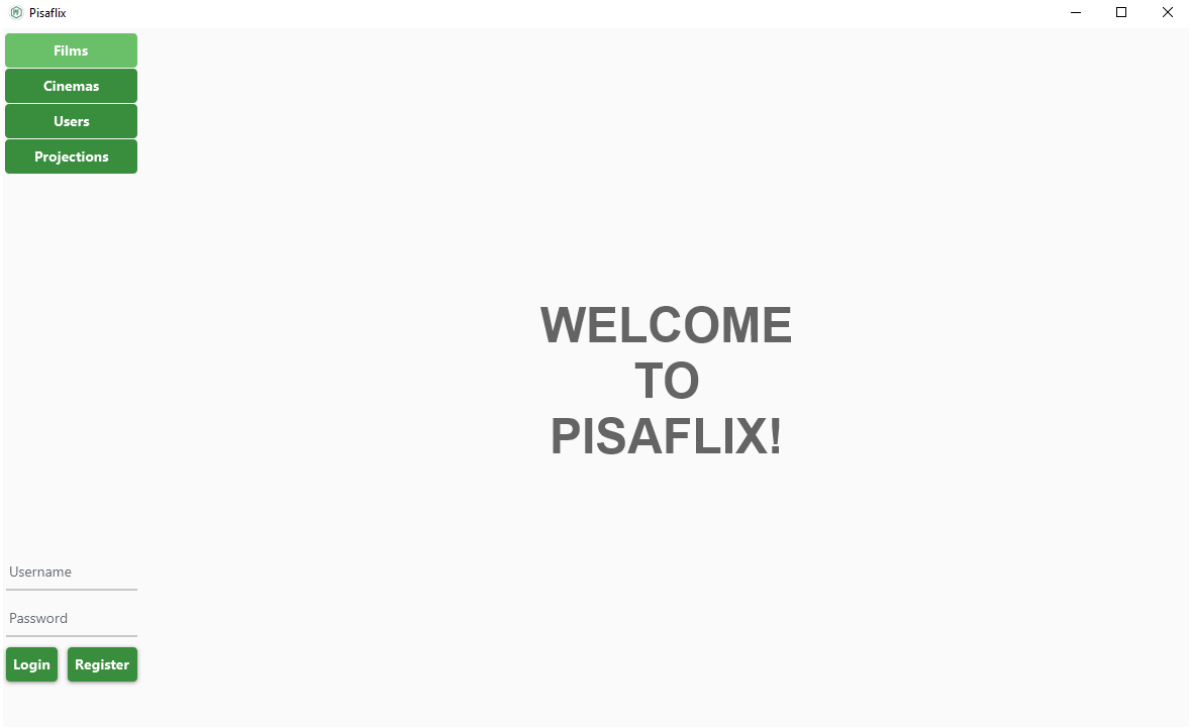
  - void **register**(String *username*, String *password*, String *email*, String *firstName*, String *lastName*) it registers a new user in the database, if some field It's not valid it throws *InvalidFieldException* specify also the reason why it was thrown
  - void **checkUserPrivilegesForOperation**(UserPrivileges *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 *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 calls **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 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 calls **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 depending by the operation that we want perform:
  - Set<Film> **getFilmsFiltered**(String *titleFilter*, Date *startDateFilter*, Date *endDateFilter*) search in the DB and returns all movies which have "*titleFilter*" in the title and the *pubblicationDate* it's between "*startDateFilter*" and "*endDateFilter*", if some filter is not set the filter it's not taken into consideration, if all filter are not set it returns all movies.
  - Set<Film> **getAll**() returns all movies int the DB
  - Film **getById**(int *id*) returns a specific film identify by its "*id*"
  - void **addFilm**(String *title*, Date *publicationDate*, String *description*) allows to insert a new film in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't add a new film
  - void **updateFilm**(Film *film*) allows to modify a film in the DB, it throws *UserNotLoggedException* if called with no user logged, or *InvalidPrivilegeLevelException* if the logged user can't modify a film

  - void **deleteFilm**(int *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 **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 depending 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 cinema
  - 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 depending 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 depending 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 specified by "*cinemaId*" and the film specified by "*filmId*" it also take in consideration the date specified 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.

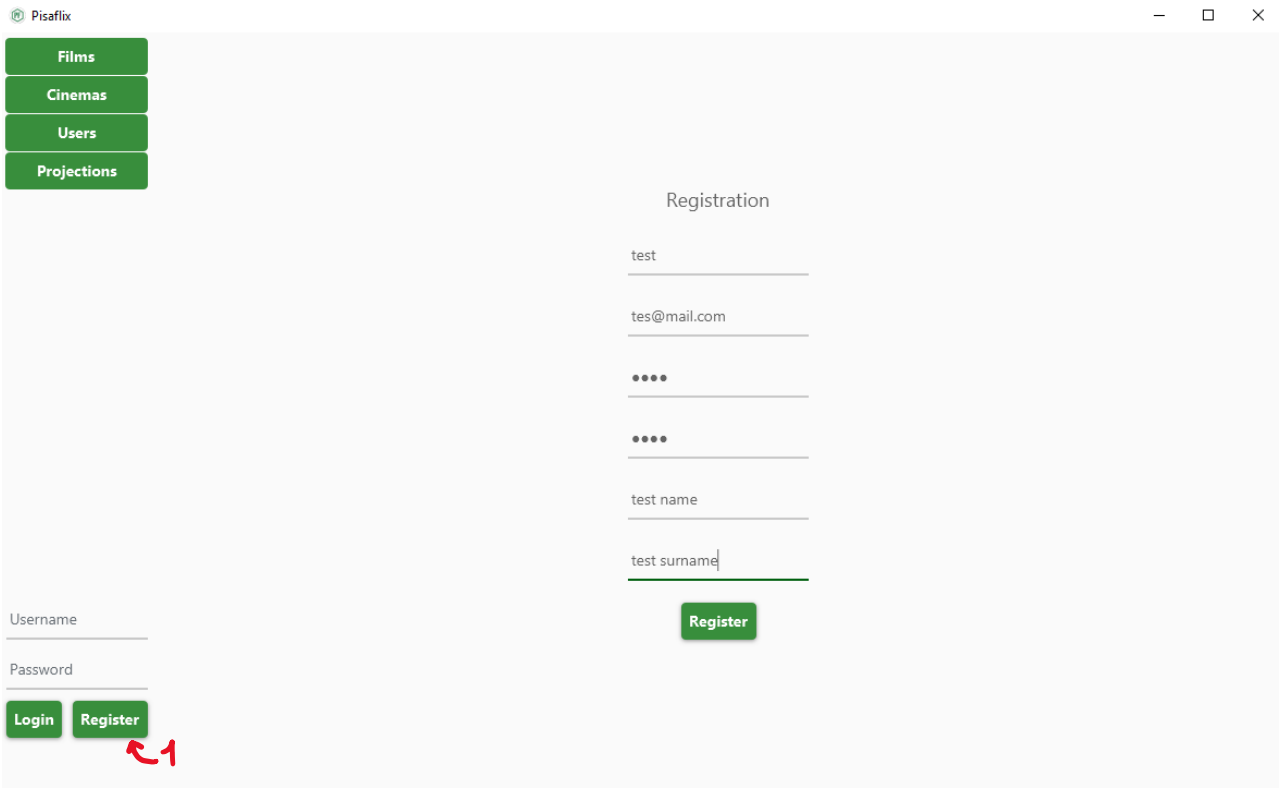
USER MANUAL

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:

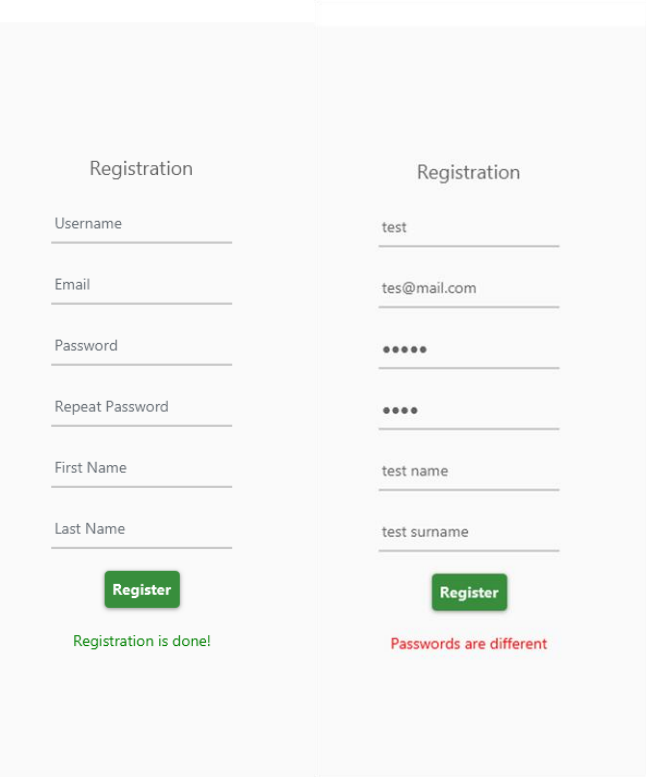


REGISTRATION AND LOGIN

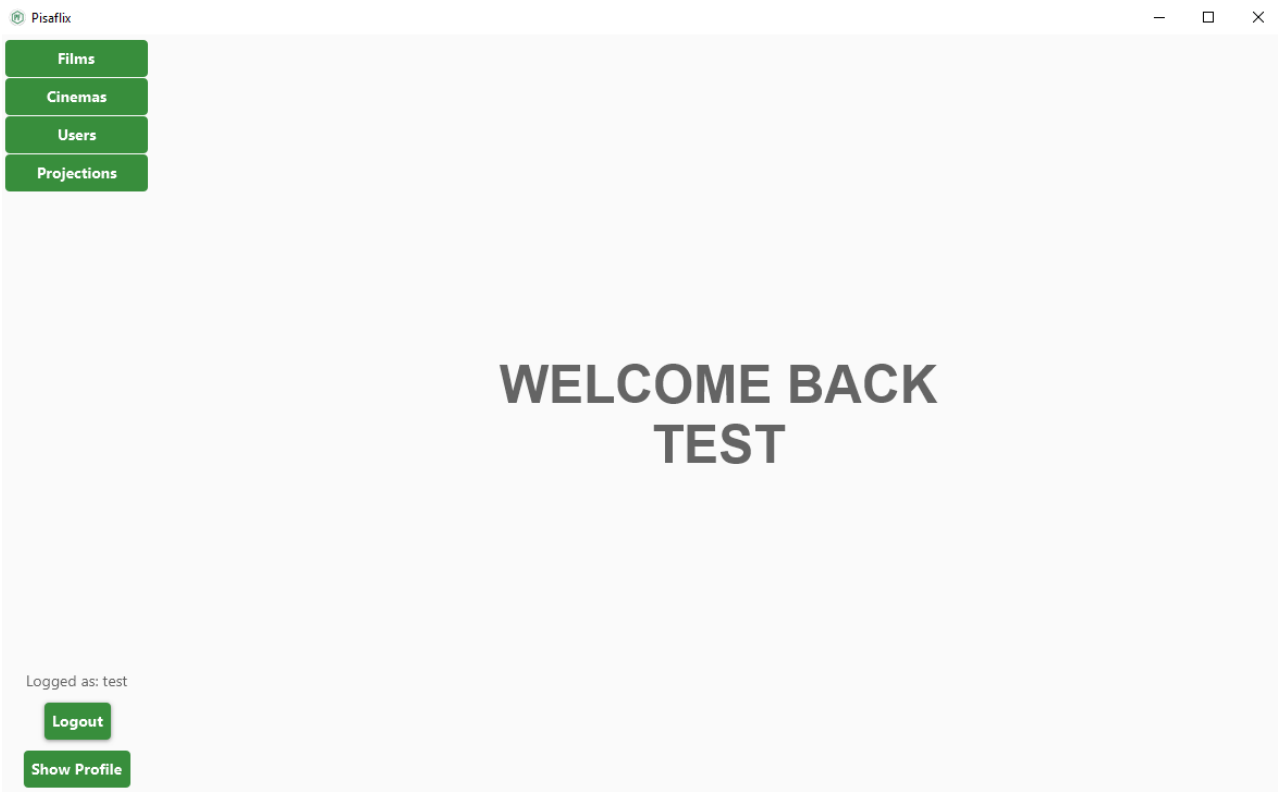
A new user can register using the specific button (1), after clicking, the registration page will appear which a user can fill out with his own information and then register:



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

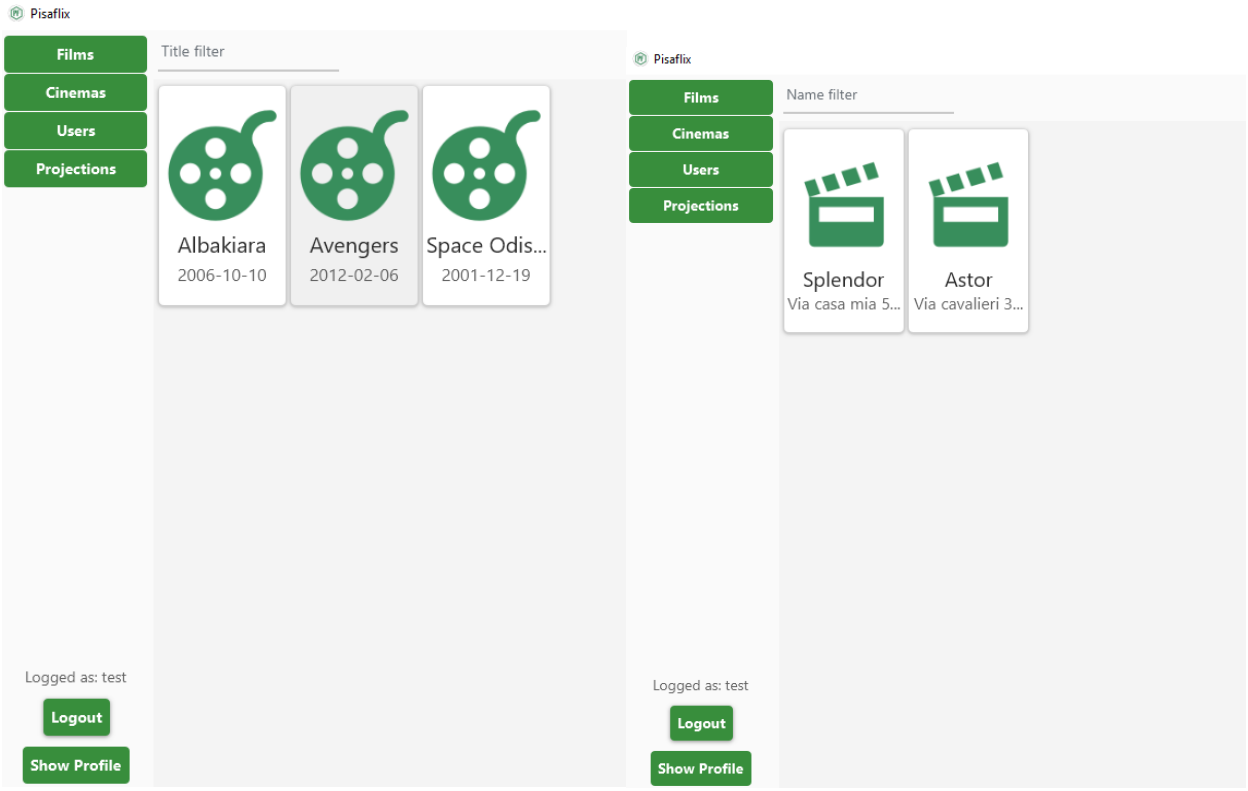


Once signed-in the user can log-in by the fields in the button left corner, if logged a user can comments movies/cinemas, add them to favourite and do all other specific operations based on his privileges:

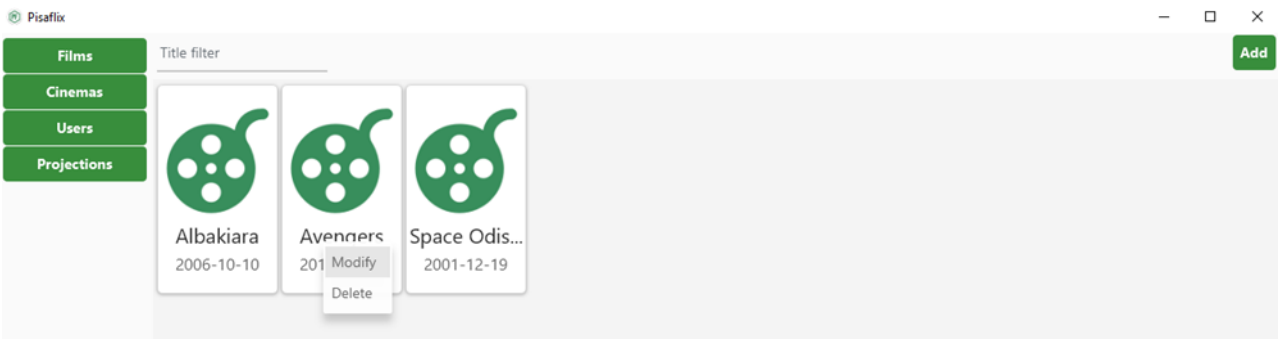


BROWSING FILM/CINEMAS

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



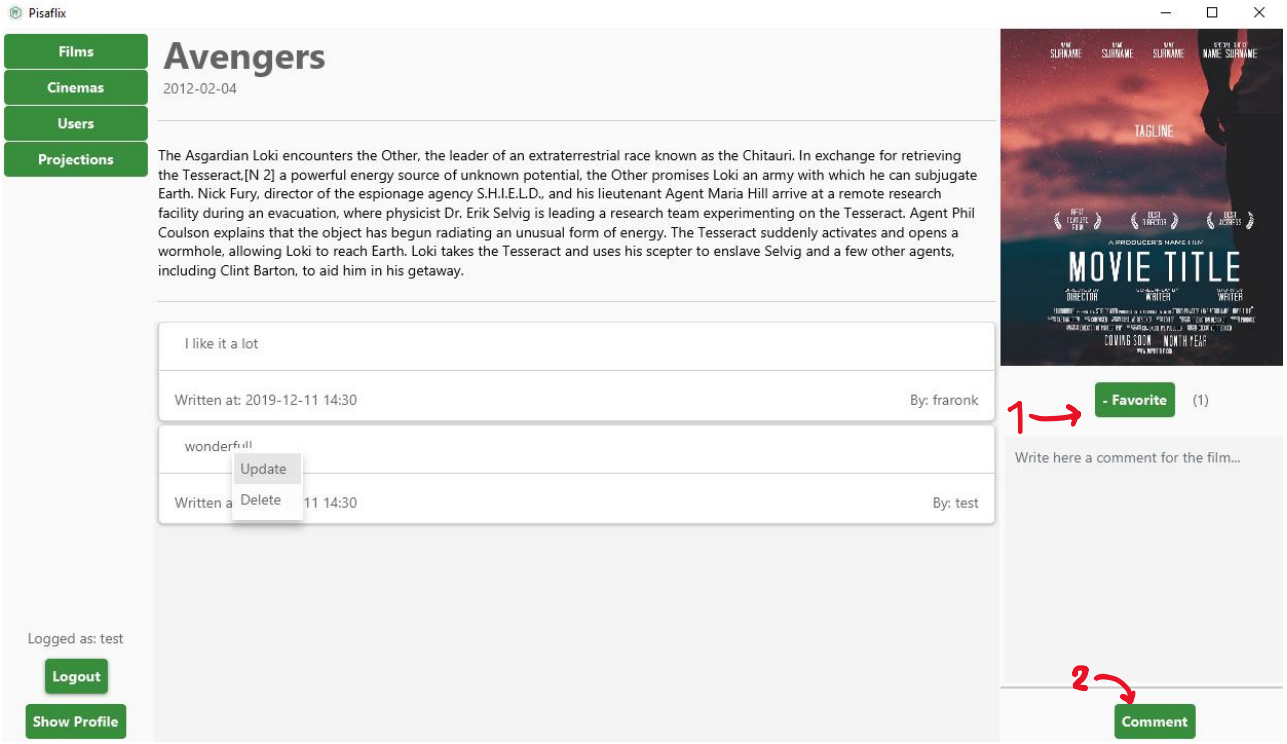
In the browse films/cinemas the user can search for a specific item filtering by title/name, if the user has the right privileges it can also add a new film/cinema (by clicking the “add” button in the top right corner) or modify/delete an existing one by right clicking on it and select the wanted operation:



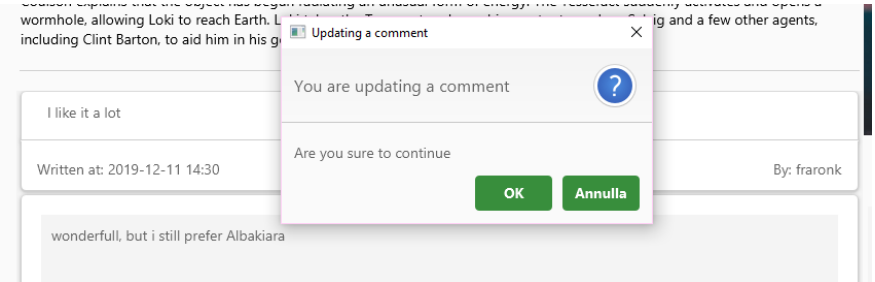
FILM/CINEMAS DETAILS

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 the users.

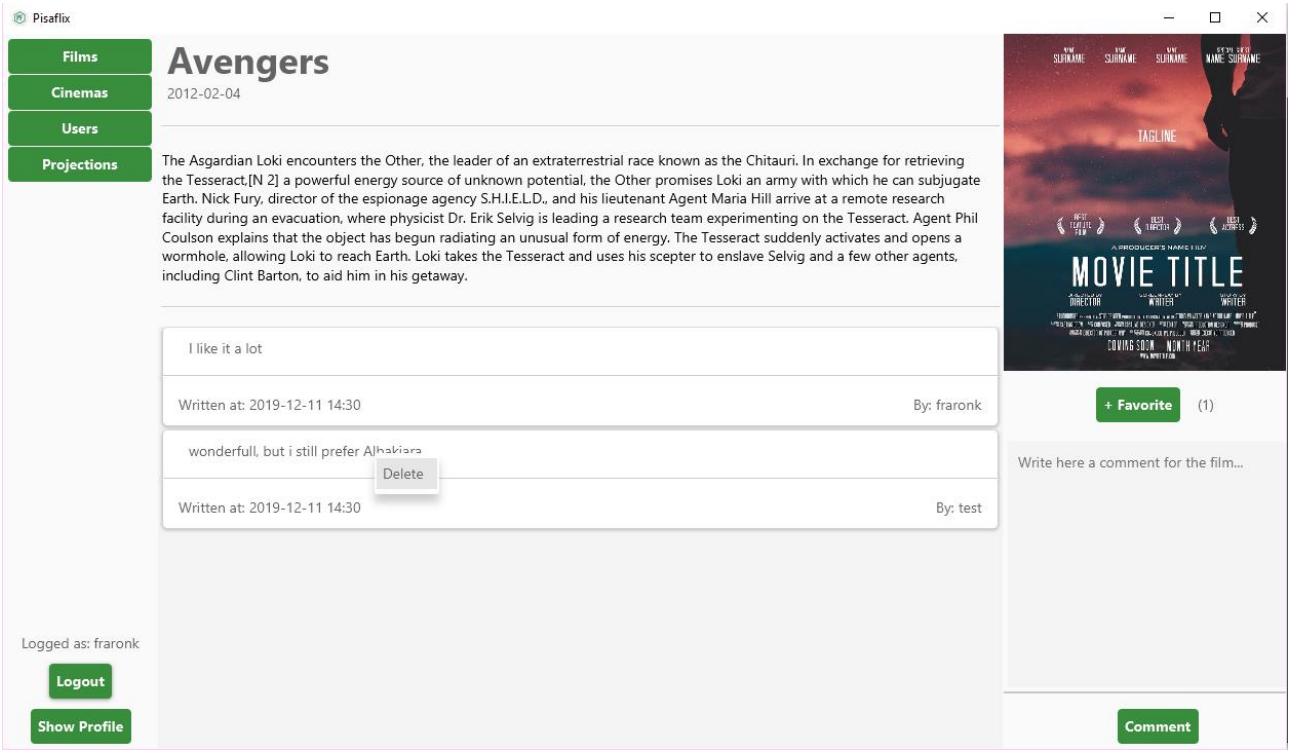
In that page a logged user can add the film/cinema to its favourite (1) or comment it (2).



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

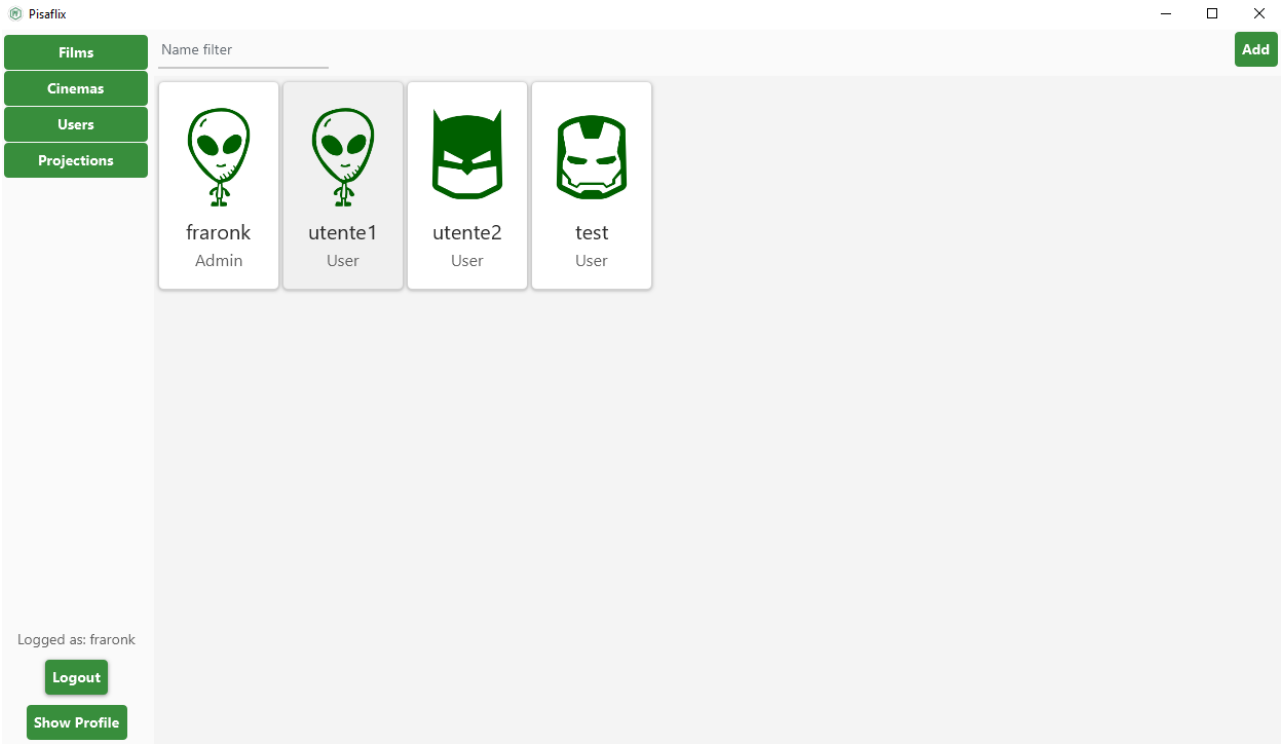


With the right privileges a user can also delete other users' comments:

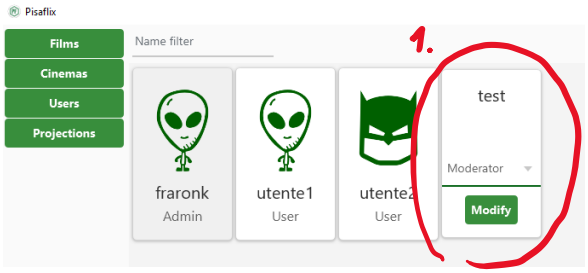


BROWSING USERS AND DETAILS

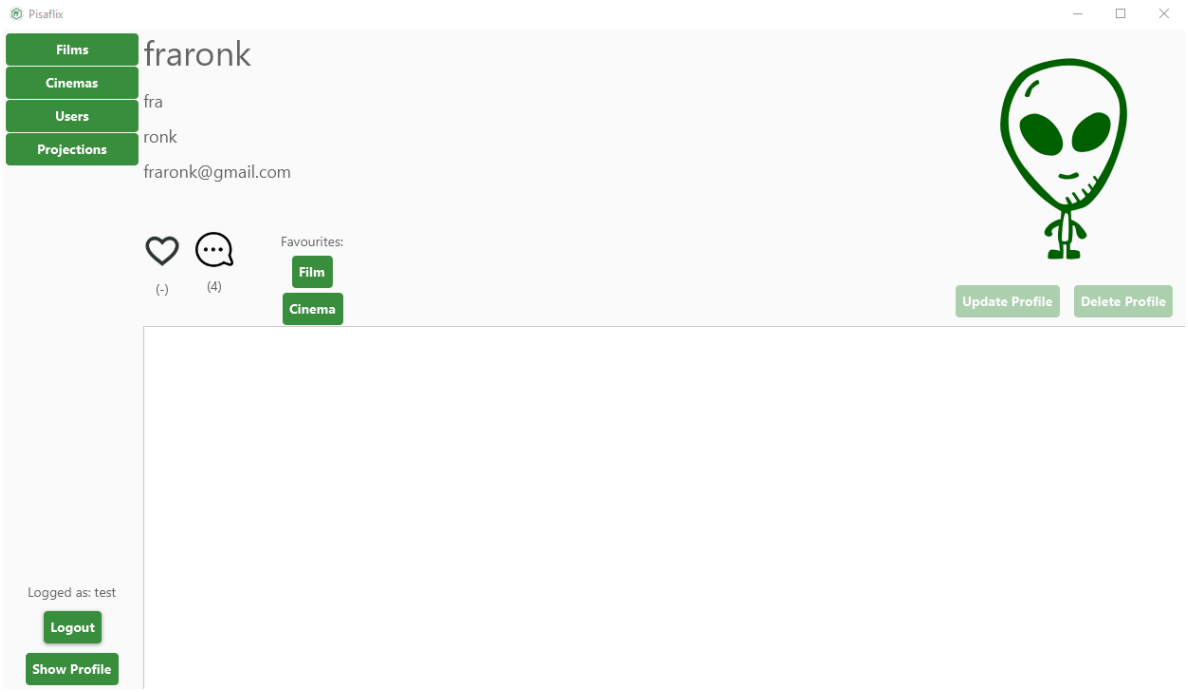
Similar to browse films/cinemas a user can also navigate through other users by the apposite button in the top left corner, the page shows all usernames and privileges.



With the right privileges a user can modify others user privileges by right clicking on them ad using the apposite menu (1).

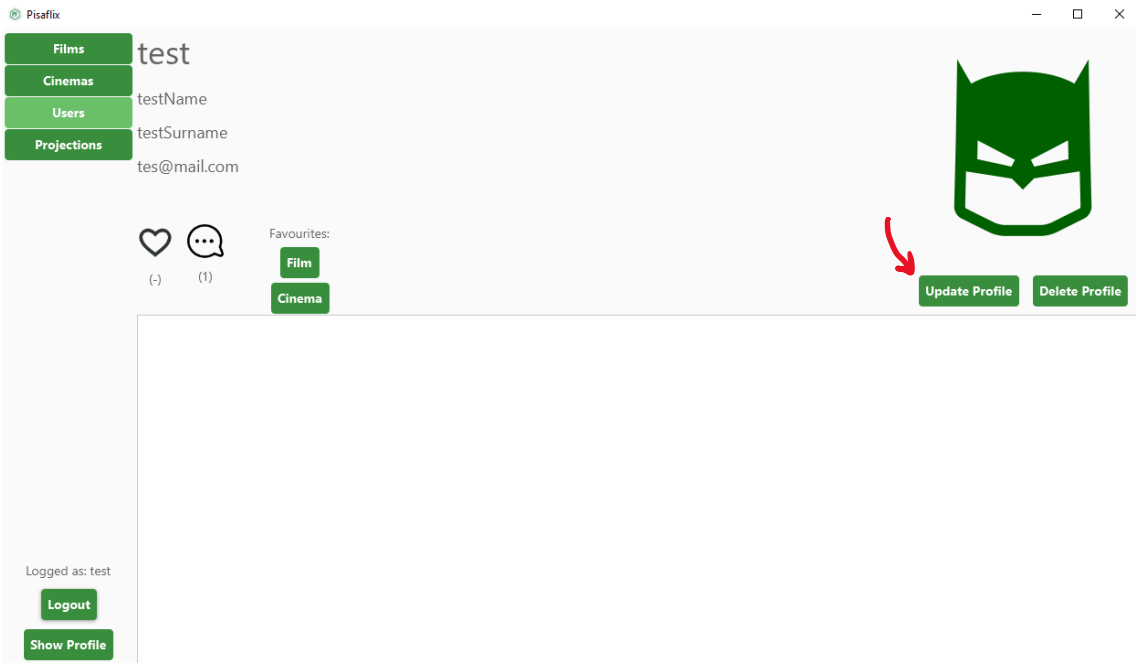


Once the user clicks on a user while browsing it will open its page detail:

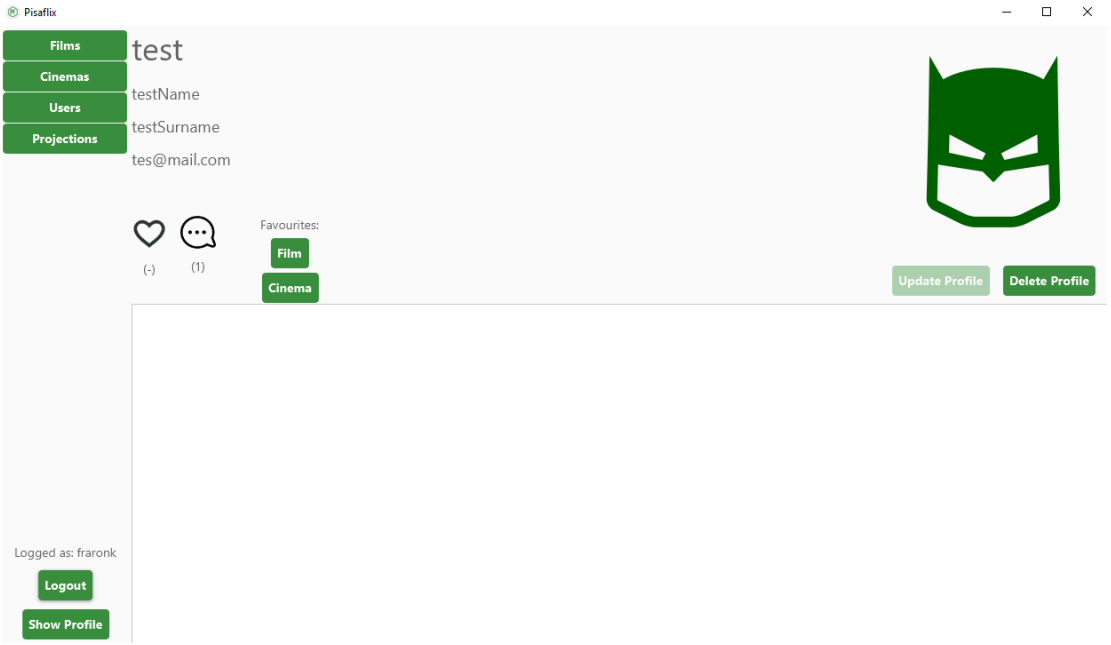


In the detail page is visible how many favourite/comment a user did and a list of his favourite films and cinemas by clicking on the respective buttons.

On his personal page, a user can modify its information or delete its account:

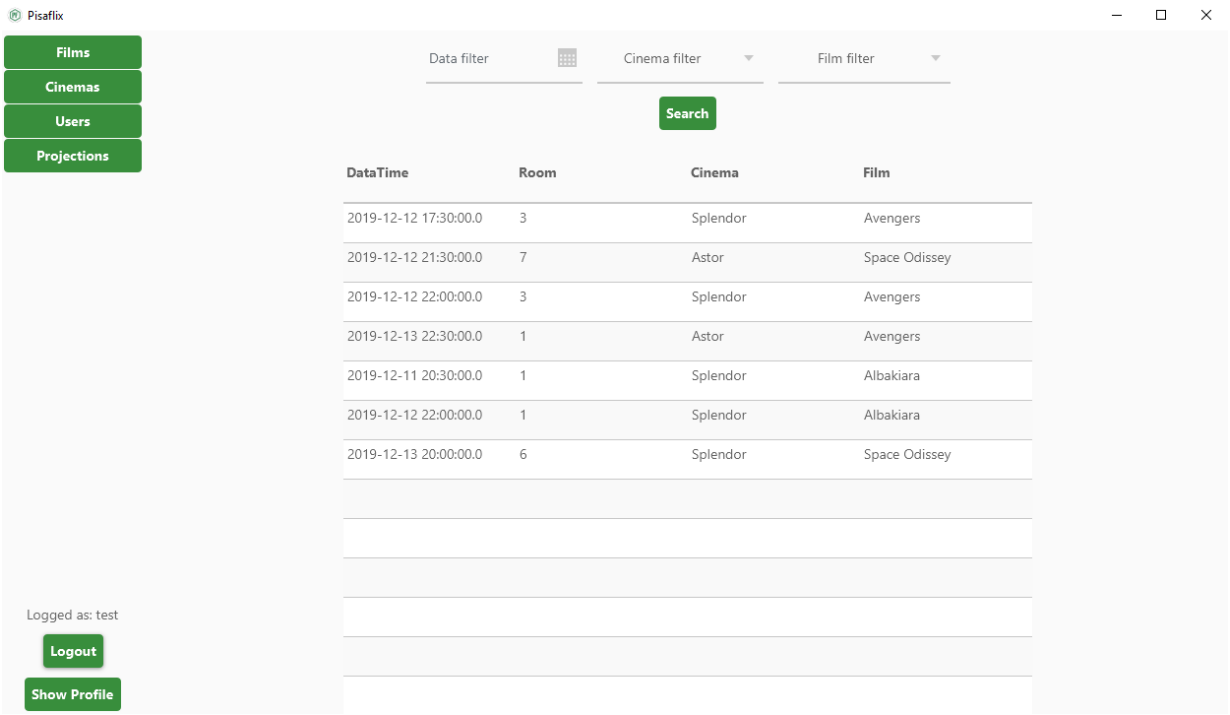


With the right privileges an administrator can have the possibility to delete another user account:



PROJECTION

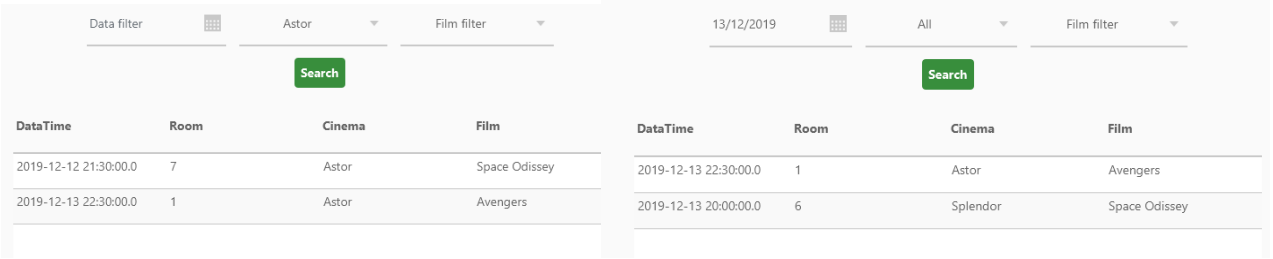
By clicking the apposite button in top left corner, the application will show the projection page on which the user can see the all the projections available:

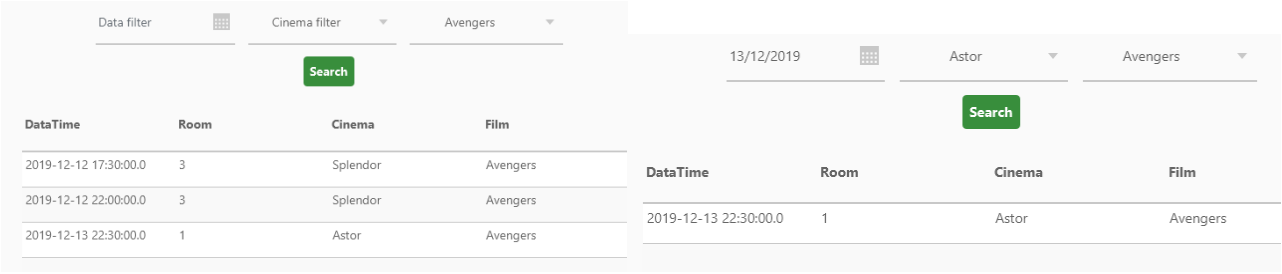


On the top of the page there are three filters that users can use to search for the projections:

- By Date
- By Cinema
- By Film

All or no filters can be used simultaneously:





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: