Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica Applicata

Distributed Programming
Review Film

Anno Accademico 2021/2022

Dario Civale – 0622701620
d.civale1@studenti.unisa.it
Giuseppe Renzulli – 0622701514
g.renzulli4@studenti.unisa.it
Paolo Mansi – 0622701542
p.mansi5@studenti.unisa.it
Vincenzo Salvati – 0622701550
v.salvati10@studenti.unisa.it

Prof Carmen De Maio

Summary

Introduction

The proposed application is a distributed web app which deals with the review of films.
In particular, there is a first phase of sign-up/login in order to authenticate the user and, in addition to that, the same user has the possibility to change its credentials: username, email and password in a specific section within the application.
Moreover, the application provides for the association of a role for each user:

- Client: after a sign-up this is the default role which is automatically assigned to the user. This kind of user is able to provide the feedback for each film, to modify and delete its previously inserted feedbacks as well as reading all the feedbacks provided by other users;
- Manager: this kind of role could be assigned only by other managers or by the admin and it makes possible. This role is appointee of managing the Actors data and all the films data within the application. In addition, it can provide feedbacks and access all the other features accessible by the clients.
- Admin: this kind of role has the same possibility of the manager, but it can not be managed from other managers. That is why it can’t be deleted from the system and only one of them exists, thus assuring the presence of at least one manager within the application.

So, the management of the users consists in changing their role or banning them; the management of the actors consist in editing name and surname, deleting and adding them; the management of the films consist in adding, updating and deleting all the information regarding the specific film.

Functional Requirements

Requirement for System Administrator:

- Authentication management: it must be possible to update its own credentials;
- Film management: it must be possible adding, updating and deleting a film;
- Actor management: it must be possible adding, updating and deleting an actor;
- Feedback assignment: it must be possible adding, updating, deleting own feedback and own score assigned to a specific film;
- User management: it must be possible banning a user in order to avoid its authentication or just changing its role;

Requirement for System Manager:

- Authentication management: it must be possible updating and deleting own credentials;
- Film management: it must be possible adding, updating and deleting a film;
- Actor management: it must be possible adding, updating and deleting an actor;
- Feedback assignment: it must be possible adding, updating, deleting own feedback and own score to a specific film;
- User management: it must be possible banning a user in order to avoid its authentication or just changing its role;

Requirement for System User:

- Authentication management: it must be possible adding, updating, deleting own credentials;
- Feedback assignment: it must be possible adding, updating, deleting own feedback and own score to a specific film;

Requirement for System Application:

- Discovery procedure; it must be possible visualizing all films and their feedbacks;
- Filtering procedure: it must be possible visualizing filtered films per genre.
- Number of feedback procedure: it must be possible counting and showing comments to a specific film;
- Averaging scores procedure: it must be possible performing and showing the mean of scores to a specific film.

Architecture

As for the chosen architecture, a RESTful microservices based structure has been developed which implement several services. Each microservice manages data of specific database and each of them is independent from the others (low-coupling) in order not to compromise the functioning of one another. Some microservices use a shared database in order to keep the consistency of data. However, each microservice remains completely independent from the others, as it just provides new functionalities to the system.

In this architectural context, there is an API Gateway which accepts all API calls and aggregates the various services' results.

So, the API Gateway is the single access point to the system which acts as an intermediary between client and providers of services. It is handled through a Servlet which receives the requests HTTP from the clients, access the Resources through the REST API, gathers the data from different services if needed and then sends back the response to the clients.

In this way, clients do not need to know where the different services are located since they do not access them directly but it's the API Gateway that provides the mapping of each request of the user to the related services.

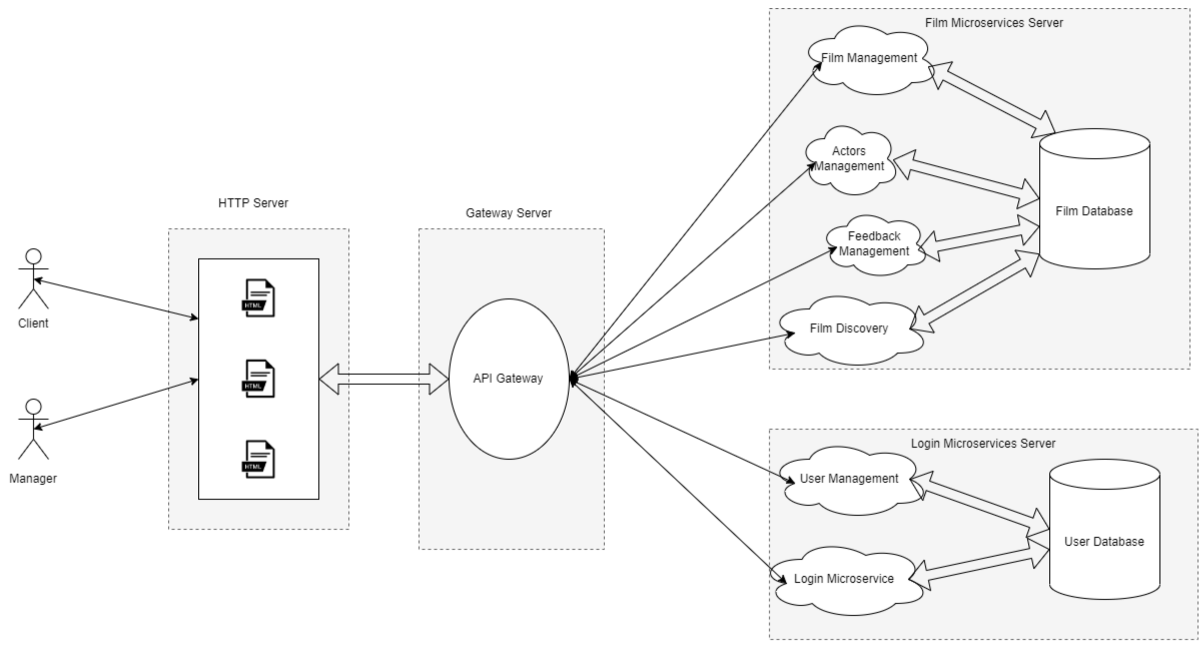


Figure 1 - Architecture

In this setting, the microservices can be distributed on several machines, with the only limitation provided by the access to the database. However, a single server can provide multiple services that access the same database, exploiting the locality of the data. Furthermore, the API Gateway should be located on a different independent machine, always accessible by the clients.

In our implementation, four servers are employed for the communication of the distributed system :

- An HTTP Server hosts the html pages, and all the front-end side of the web application part, with a specific interface for each admitted role;
- An API Server stores the API Gateway in order to deal with different HTTP requests from the clients;

- A Film Microservices Server hosts the Film_db, and thus it is used to store all the microservices that perform queries on this database;
- A Login Microservices Server hosts the User_db and thus it is used to store all the microservices related to the user management and login, that perform queries on this database.

Communication Protocols

As for the communication protocols, the communication between the user and the Gateway happens over HTTP get and post that exchange data in JSON format.

Once the servlet receives the request, it extracts the data and uses them to construct the REST Resource Request, which means making use of the HTTP Protocol for requesting data properly marshalled in JSON format.

The Microservice properly accessed through REST uses the data of object received to perform the query on the database and returns the data to the Gateway as the REST Response. The Gateway then uses these data to fill the response body of the HTTP Request and returns it to the Web Client.

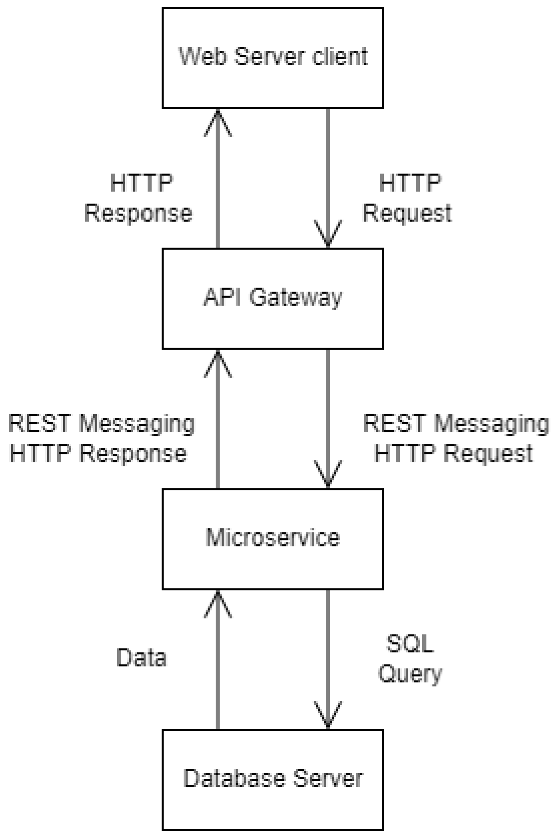


Figure 2 - Comunication protocols

Microservices

The microservices are distributed on two macro services which fetches data from database e passes them to the Gateway:

- a set of microservices which dealing with the management of the films and all the information related to them: ActorResource, FeedbackResource, FilmResource, FilmQueryResource;
- a set of microservices which dealing with the management of the films and all the information related to them: LoginResource, UserResource.

ActorResource

addActor

```
public boolean addActor(Actor actor) throws SQLException
```

Insert a new Actor inside the database

Parameters:

`actor` - - the Actor to be added

Returns:

true if the insertion is completed correctly, false otherwise

editActor

```
public boolean editActor(Actor actor) throws SQLException
```

Edit the Actor data inside the database

Parameters:

`actor` - - the Actor to be edited

Returns:

true if the edit is completed correctly, false otherwise

deleteActor

```
public boolean deleteActor(int id_actor) throws SQLException
```

Delete the Actor associated to the given id

Parameters:

`id_actor` - - the id of the actor to delete

Returns:

true if the deletion is completed correctly, false otherwise

getActor

```
public Actor getActor(int id_actor) throws SQLException
```

Returns the Actor data associated to the passed id

Parameters:

`id_actor` - - the id of the actor to retrieve

Returns:

the Actor object containing all the data of the Actor

getActors

```
public ArrayList<Actor> getActors() throws SQLException
```

Retrieves all data associated to all Actors of the database

Returns:

a list of Actor objects

FeedbackResource

addFeedback

```
public boolean addFeedback(Feedback feedback) throws SQLException
```

Insert a new Feedback inside the database

Parameters:

`feedback` - - the Feedback to be added

Returns:

true if the insertion is completed correctly, false otherwise

editFeedback

```
public boolean editFeedback(Feedback feedback) throws SQLException
```

Edit the Feedback data inside the database

Parameters:

`feedback` - - the Feedback to be edited

Returns:

true if the edit is completed correctly, false otherwise

deleteFeedback

```
public boolean deleteFeedback(int id_film,     int id_user)     throws
SQLException
```

Delete the Feedback of the specified user regarding the specified film

Parameters:

`id_film` - - the id of the film the feedback refers to

`id_user` - - the id of the user who made the feedback

Returns:
true if the deletion is completed correctly, false otherwise

getFeedback

```
public Feedback getFeedback(int id_film,      int id_user)      throws
SQLException
```

Retrieves the Feedback of the specified user regarding the specified film

Parameters:

`id_film` - - the id of the film the feedback refers to

`id_user` - - the id of the user who made the feedback

Returns:
the Feedback object

getFeedbackByFilm

```
public ArrayList<Feedback> getFeedbackByFilm(int id_film)      throws
SQLException
```

Retrieves all data associated to all Feedbacks of the database associated to the specified film

Parameters:

`id_film` - - the id of the film the feedbacks refer to

Returns:
a list of Feedback objects

getFeedbackByUser

```
public ArrayList<Feedback> getFeedbackByUser(int id_user)      throws
SQLException
```

Retrieves all data associated to all Feedbacks of the database associated to the specified user

Parameters:

`id_user` - - the id of the user who made the feedback

Returns:
a list of Feedback objects

FilmResource

addFilm

```
public boolean addFilm(Film film) throws SQLException
```

Insert a new Film inside the database

Parameters:

`film` - - the Film to be added

Returns:
true if the insertion is completed correctly, false otherwise

editFilm

```
public boolean editFilm(Film film) throws SQLException
```

Edit the Film data inside the database

Parameters:

`film` - - the Film to be edited

Returns:
true if the edit is completed correctly, false otherwise

deleteFilm

```
public boolean deleteFilm(int id_film) throws SQLException
```

Delete the Film associated to the given id

Parameters:

`id_film` - - the id of the film to delete

Returns:
true if the deletion is completed correctly, false otherwise

getFilm

```
public Film getFilm(int id_film) throws SQLException
```

Returns the Film data associated to the passed id

Parameters:

`id_film` - - the id of the film to retrieve

Returns:
the Film object containing all the data of the Film

getFilms

```
public List<Film> getFilms() throws SQLException
```

Retrieves all data associated to all Films of the database

Returns:

a list of Film objects

FilmQueryResource

getFilmsHomePage

```
public List<HomePageFilm> getFilmsHomePage() throws SQLException
```

Retrieves a list of HomePageFilms

Returns:

the list of HomePageFilms

getFilmsHomePagePerGenre

```
public List<HomePageFilm> getFilmsHomePagePerGenre(String genre)
throws SQLException
```

Retrieves a list of HomePageFilm associated to the specific genre

Parameters:

`genre` - - the genre of the film to retrieve

Returns:

the list of HomePageFilms

getFilmReviewPage

```
public ReviewPageFilm getFilmReviewPage(int id_film)          throws
SQLException
```

Retrieves the ReviewPageFilm associated to the specified id

Parameters:

`id_film` - - the id of the film to retrieve

Returns:

a ReviewPageFilm object

LoginResource

loginUser

```
public UserCookie loginUser(UserLogin user) throws SQLException
```

Performs the login of the user. It checks for the presence of the user inside the database. If the data are correct, it returns the data representing the cookie

Parameters:

`user` - - the user data to check for

Returns:
an UserCookie object containing the data representing the cookie

UserResource

addUser

```
public boolean addUser(User user) throws SQLException
```

Insert a new User inside the database

Parameters:

`user` - - the User to be added

Returns:
true if the insertion is completed correctly, false otherwise

editUser

```
public boolean editUser(User user) throws SQLException
```

Edit the User data inside the database

Parameters:

`user` - - the User to be edited

Returns:
true if the edit is completed correctly, false otherwise

deleteUser

```
public boolean deleteUser(int id_user) throws SQLException
```

Delete the User associated to the given id

Parameters:

`id_user` - - the id of the user to delete

Returns:
true if the deletion is completed correctly, false otherwise

getUser

```
public User getUser(int id_user) throws SQLException
```

Returns the User data associated to the passed id

Parameters:

`id_user` - - the id of the user to retrieve

Returns:

the User object containing all the data of the user

getUsers

```
public List<UserCookie> getUsers() throws SQLException
```

Retrieves the list of users.

Returns:

a list of UserCookie object containing the data representing the cookie for each user

toggleBan

```
public boolean toggleBan(int id_user, int currentStatus) throws SQLException
```

Changes the status of the user, from banned to unbanned and viceversa

Parameters:

`id_user` - the id of the user target

`currentStatus` - the current status

Returns:

true if the toggle is completed correctly, false otherwise

toggleRole

```
public boolean toggleRole(int id_user, String currentRole) throws SQLException
```

Changes the role of the user, from client to manager and viceversa

Parameters:

`id_user` - the id of the user target

`currentRole` - the current role of the user

Returns:

true if the toggle is completed correctly, false otherwise

getNoBannedUsers

```
public List<UserCookie> getNoBannedUsers() throws SQLException
```

Retrieves the list of Not banned users.

Returns:

a list of UserCookie object containing the data representing the cookie for each target user

Database

As far as the database concerned, it has been implemented two rational databases by MySQL to provide data to the two macro services:

1. Film services: to manage films, their casts and feedbacks.

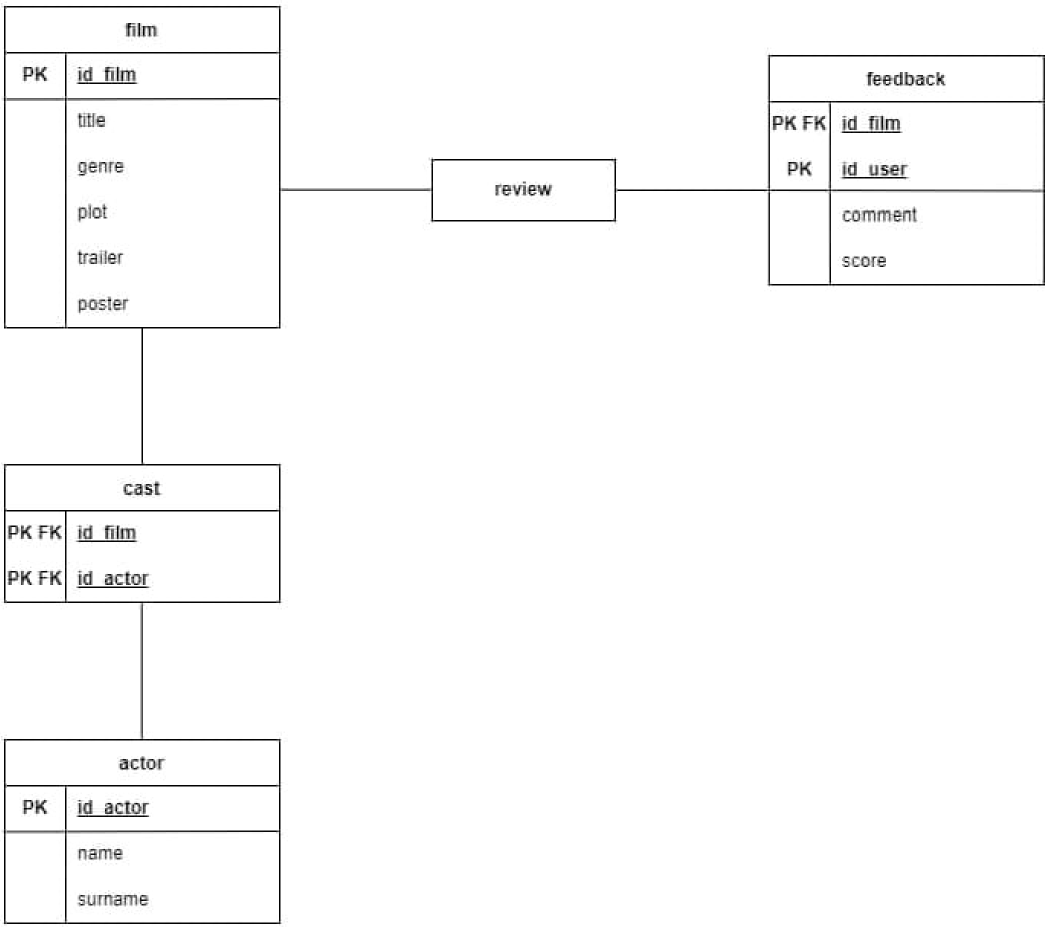


Figure 3 - Film_db

2. User services: to manage the users' account and their login.

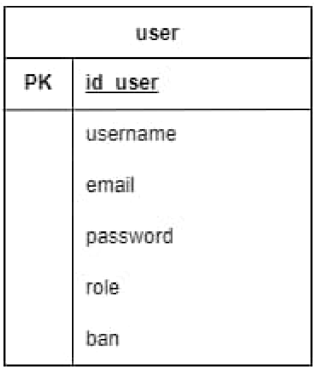


Figure 4 - User_db

Testing

During the development of the web application, it has been performed the test regarding the different functions in microservices' implementation by Jupiter libraries. In that way, it has made sure the basic queries work on different databases:

- a set of microservices' tests which check the proper execution of management of the films and all the information related to them:
  - ActorResourceTest:
    - *void* **testAddActor**();
    - *void* **testEditActor**();
    - *void* **testDeleteActor**();
    - *void* **testGetActor**();
    - *void* **testGetActors**();
  - FilmResourceTest:
    - *void* **testAddFilm**();
    - *void* **testEditFilm**();
    - *void* **testDeleteFilm**();
  - FeedbackResourceTest:
    - *void* **testAddFeedback**();
    - *void* **testEditFeedback**();
    - *void* **testDeleteFeedback**();
    - *void* **testGetFeedback**();
    - *void* **testGetFeedbackByFilm**();
    - *void* **testGetFeedbackByUser**();
- a set of microservices' tests which check the proper execution of the management of the users' account:
  - LoginResourceTest:
    - *void* **testLoginUser**();
    - *void* **testGetNoBannedUsers**();
    - *void* **testGetUsers**();
  - UserResourceTest:
    - *void* **testAddUser**();
    - *void* **testEditUser**();
    - *void* **testDeleteUser**();
    - *void* **testGetUser**().

How to run

This distributed application has been thought to be executed on different machine by opened port in order to have the access on Tomcat server.

In particular, it has been created five kinds of packages:

- Database: creation and initialization queries of each database;
- Documentation: Doxygen documentation;
- FrontEnd: JavaScript, Html and CSS files;
- Gateway: Java classes;
- LoginMicroservice: Java classes and Db interfaces;
- FilmMicroservices: Java classes and Db interfaces;

To simplify the change of IP addresses, there are only two files which have to be changed:

- "*FrontEnd/src/main/webapp/src/js/GatewayEndPoint.js*": which must contain the Gateway socket.

  ```
  Es.
  ENDPOINT = "http://87.1.89.130:8080";
  ```

- "*C:\Program Files\Apache Software Foundation\Tomcat 9.0\bin\setting.properties*": which must contain server film's microservices socket and server login's microservices socket.

  ```
  Es.
  film_endpoint = http://79.46.58.44:8080
  login_endpoint = http://31.22.51.30:8080
  ```

So, to properly execute the web app program, it is necessary:

1) running MySQL queries for create and initialize the database;
2) setting each Db interfaces;
3) running properly each end point;
4) setting endpoints files;
5) starting "*FrontEnd/src/main/webapp/src/html/index.html*" into a browser.

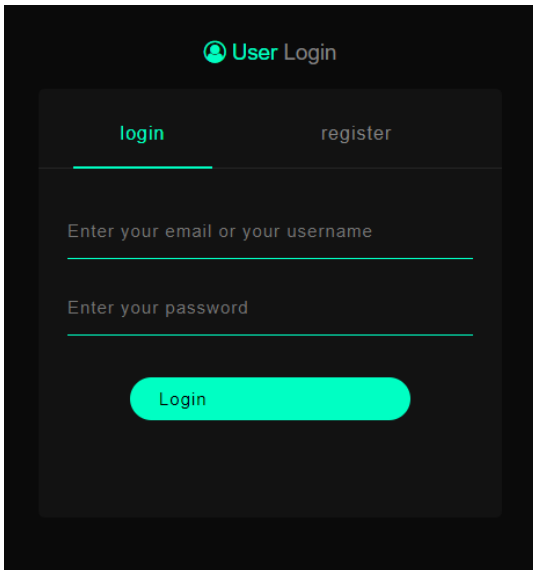
P.S.
The front-end could be run on another server Tomcat to make it accessible from others.

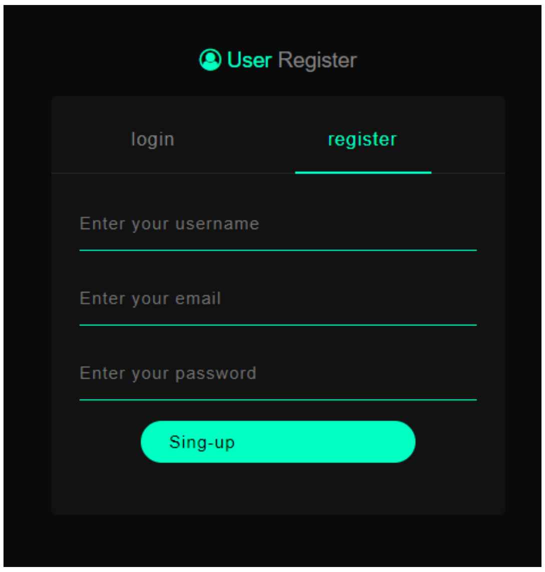
```
Es.
http://21.45.35.60:8084/src/html/index.html
```

For trying: http://79.46.58.44:8084/src/html/index.html

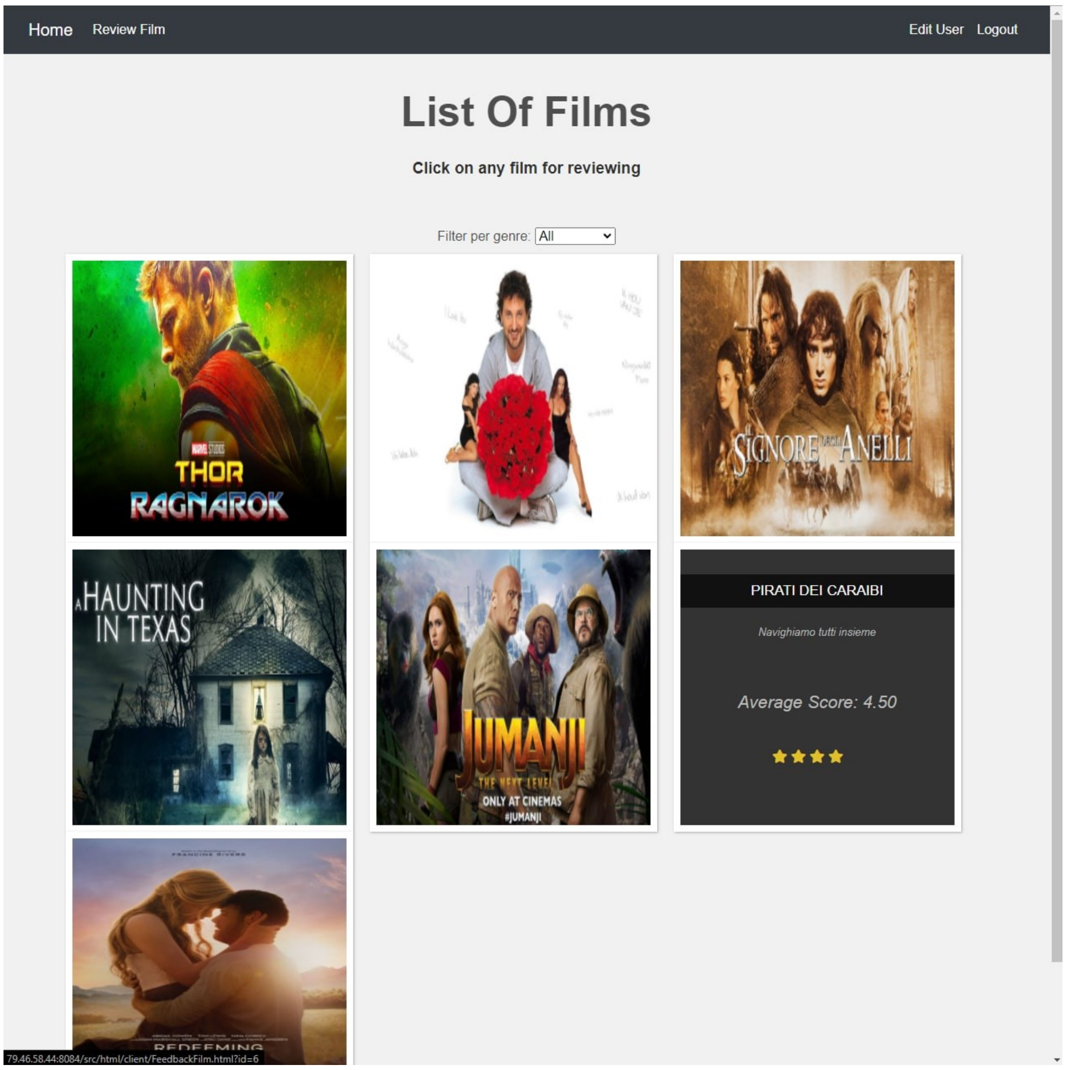
Interface Web Application

Login and Sign-up

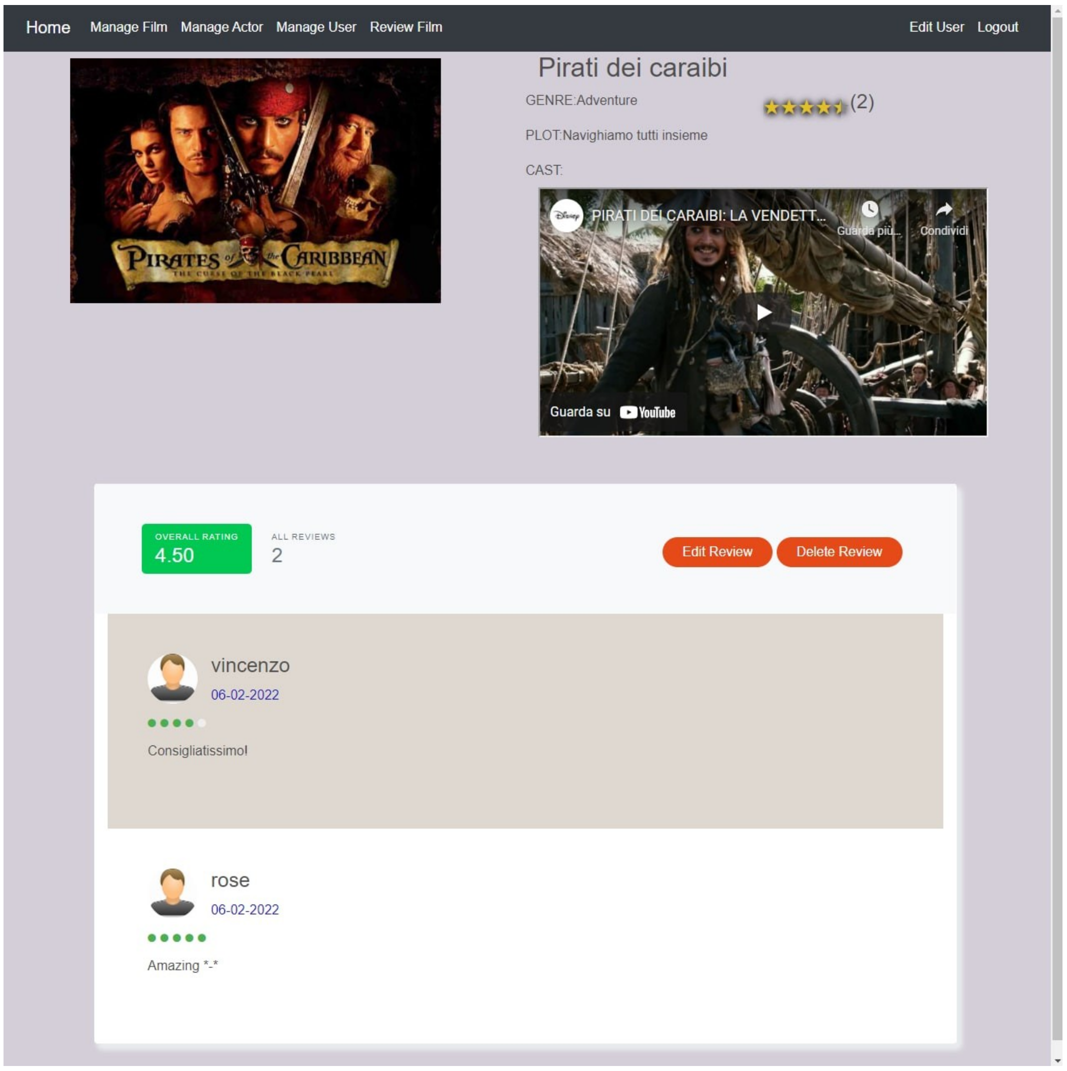




Review films by client



Assignment feedbacks



Pirati dei caraibi

GENRE:Adventure ★★★★½ (2)

PLOT:Navighiamo tutti insieme

CAST:

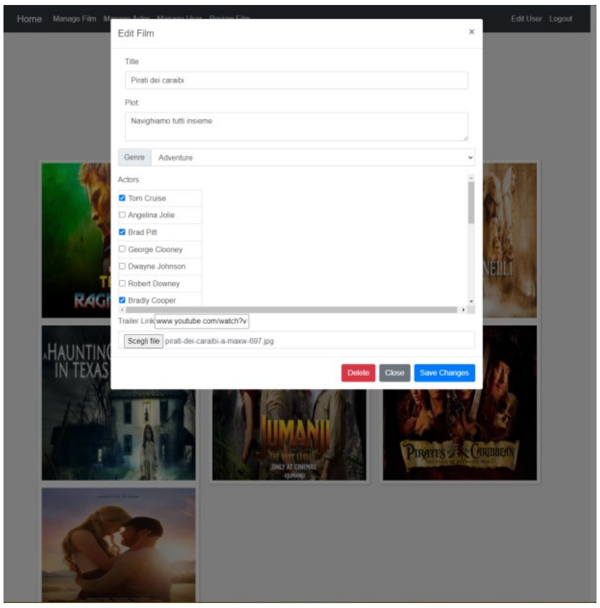
OVERALL RATING 4.50

ALL REVIEWS 2

vincenzo
06-02-2022
●●●●○
Consigliatissimo!

rose
06-02-2022
●●●●●
Amazing *-*

Management films and actors



Edit Film

Title: Pirati dei caraibi

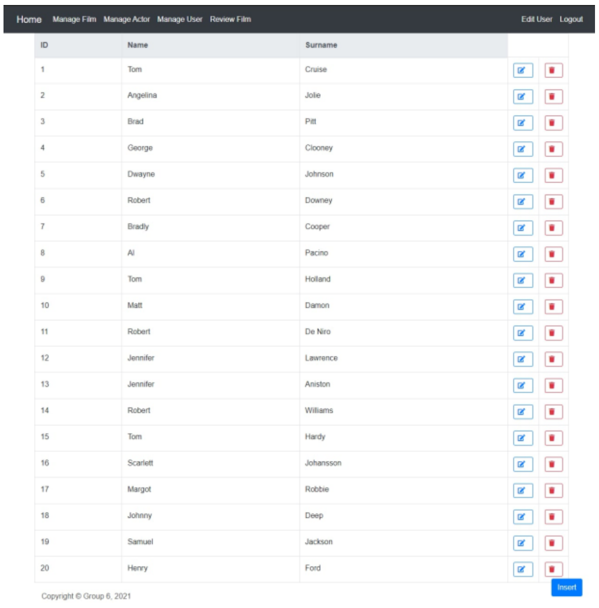
Plot: Navighiamo tutti insieme

Genre: Adventure

Actors:
- [✓] Tom Cruise
- [ ] Angelina Jolie
- [✓] Brad Pitt
- [ ] George Clooney
- [ ] Dwayne Johnson
- [ ] Robert Downey
- [✓] Bradly Cooper

Trailer Link: www.youtube.com/watch?v

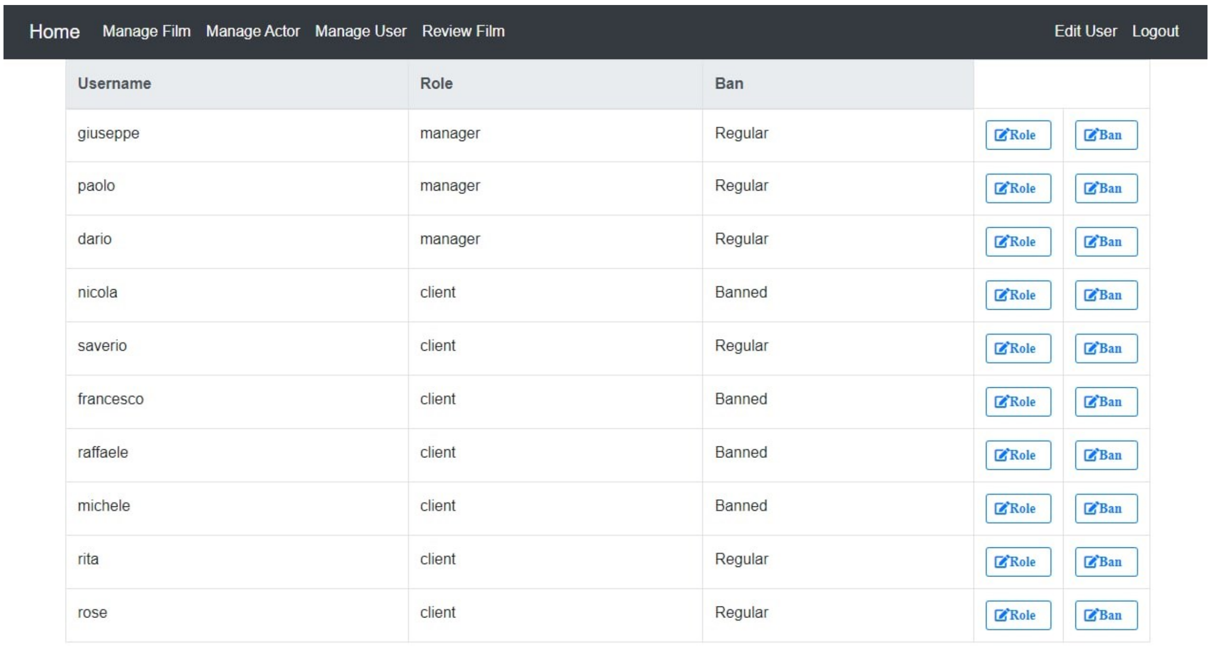
Scegli file: pirati-dei-caraibi-a-maxw-697.jpg



ID	Name	Surname
1	Tom	Cruise
2	Angelina	Jolie
3	Brad	Pitt
4	George	Clooney
5	Dwayne	Johnson
6	Robert	Downey
7	Bradly	Cooper
8	Al	Pacino
9	Tom	Holland
10	Matt	Damon
11	Robert	De Niro
12	Jennifer	Lawrence
13	Jennifer	Aniston
14	Robert	Williams
15	Tom	Hardy
16	Scarlett	Johansson
17	Margot	Robbie
18	Johnny	Deep
19	Samuel	Jackson
20	Henry	Ford

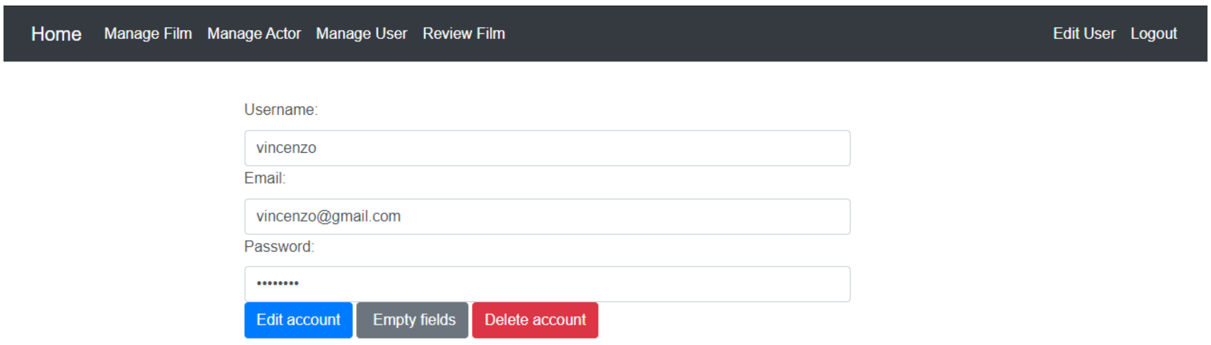
Copyright © Group 6, 2021

Management users



Username	Role	Ban		
giuseppe	manager	Regular	Role	Ban
paolo	manager	Regular	Role	Ban
dario	manager	Regular	Role	Ban
nicola	client	Banned	Role	Ban
saverio	client	Regular	Role	Ban
francesco	client	Banned	Role	Ban
raffaele	client	Banned	Role	Ban
michele	client	Banned	Role	Ban
rita	client	Regular	Role	Ban
rose	client	Regular	Role	Ban

Management own account



Username: vincenzo

Email: vincenzo@gmail.com

Password: ••••••••