CS353 - DATABASE SYSTEMS

PROJECT DESIGN REPORT

08.04.2022

GROUP NO: 19

Ali Emre Aydoğmuş - 21901358

Mustafa Çağrı Durgut - 21801983

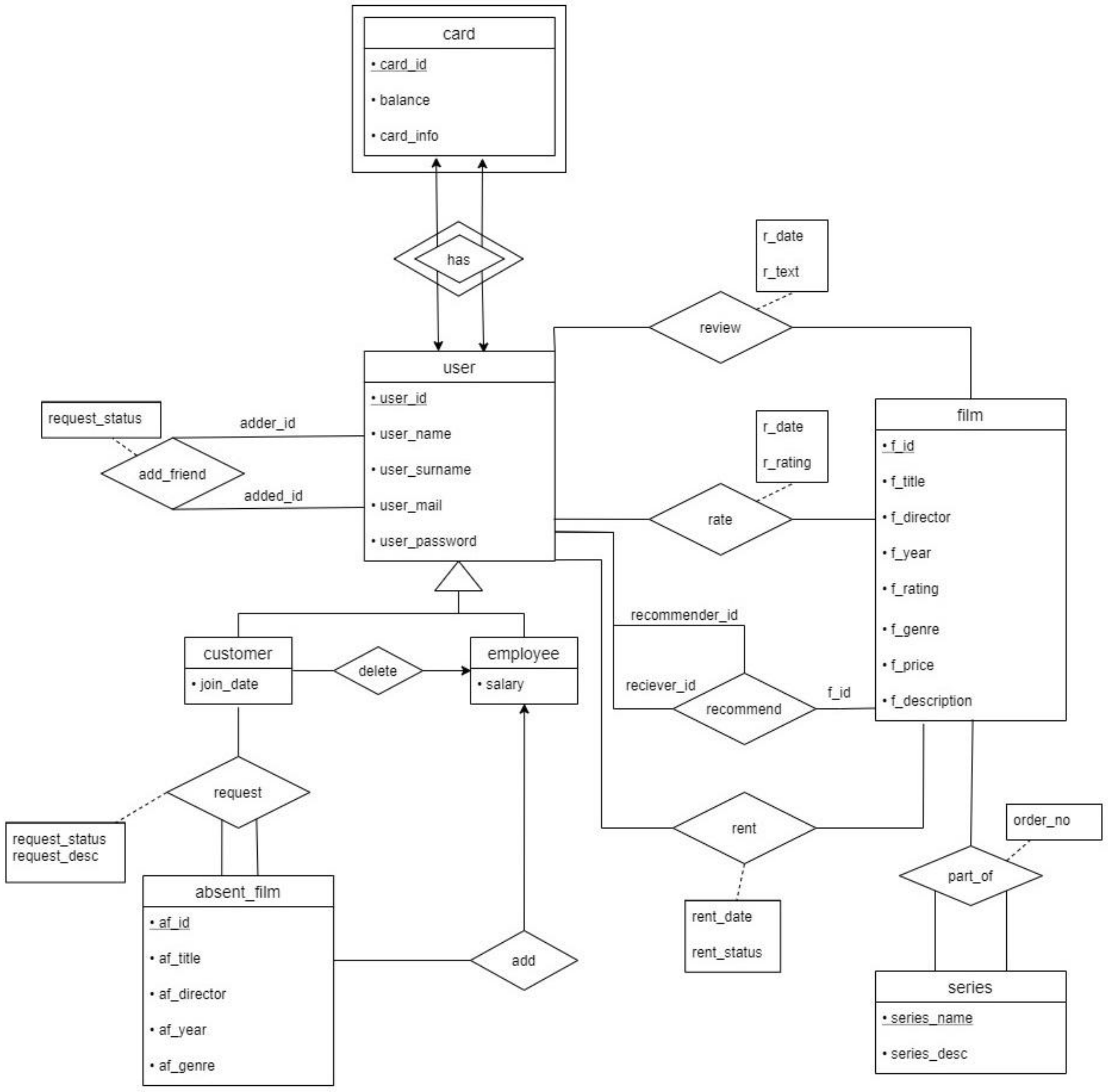
Yekta Seçkin Satır- 21903227

Yusuf Miraç Uyar - 21802626

Instructor: Hamdi Dibekoğlu

TA: Zülal Bingöl

Revised E/R Diagram



Database Relations

1) User

Relational Model:
user(user_id, user_name, user_surname, user_mail, user_password)

Functional Dependencies:
user_id -> all
user_mail -> all

Candidate Keys:
{ (user_id), (user_mail) }

Normal Forms:
3NF
User has dependencies user_id -> all and user_mail -> all. The first one does not disrupt the 3NF as user_id is the primary key. And the latter one does not disrupt the 3NF as user_mail -> user_id, making it another superkey.

Table Definition:
```
     CREATE TABLE IF NOT EXISTS user(user_id char(12) NOT NULL
PRIMARY KEY AUTO_INCREMENT, user_name varchar(30), user_surname
varchar(30), user_mail varchar(60), user_password varchar(30) )
```

2) Customer

Relational Model:
customer(user_id, join_date)

Functional Dependencies:
user_id -> all

Candidate Keys:
{ (user_id) }

Normal Forms:
3NF
Customer has the dependency user_id -> all. This does not disrupt the 3NF as user_id is the primary key.

Table Definition:

```
    CREATE TABLE IF NOT EXISTS customer(user_id char(12) NOT NULL
PRIMARY KEY, join_date date, CONSTRAINT customer_pk FOREIGN KEY
(user_id) REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE
CASCADE )
```

3) Employee

Relational Model:
employee(<u>user_id</u>, salary)

Functional Dependencies:
user_id -> all

Candidate Keys:
{ (user_id) }

Normal Forms:
3NF
Employee has the dependency user_id -> all. This does not disrupt the 3NF as user_id is the primary key.

Table Definition:

```
    CREATE TABLE IF NOT EXISTS employee(user_id char(12) NOT NULL
PRIMARY KEY, salary int, CONSTRAINT employee_pk FOREIGN KEY (user_id)
REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE CASCADE )
```

4) Film

Relational Model:
film(<u>f_id</u>, f_title, f_director, f_year, f_rating, f_genre, f_price, f_desc)

Functional Dependencies:
f_id -> all
director, title, year -> f_id

Candidate Keys:
{ (f_id) }

Normal Forms:
3NF

Film has dependencies f_id -> all and director, title, year -> id. The first one does not disrupt the 3NF as f_id is the primary key. And the latter one does not disrupt the 3NF as director, title, year -> f_id making it another superkey.

Table Definition:

```
CREATE TABLE IF NOT EXISTS film(f_id char(12) NOT NULL PRIMARY KEY AUTO_INCREMENT, f_title varchar(50), f_director varchar(60), f_year char(20), f_rating float(24), f_genre varchar(20), f_price float(24), f_desc varchar(150) )
```

5) Absent Film

Relational Model:
absent_film(af_id, af_title, af_director, af_year, af_genre)

Functional Dependencies:
af_id -> all

Candidate Keys:
{ (af_id) }

Normal Forms:
3NF
Absent Film has the dependency af_id -> all. This does not disrupt the 3NF as af_id is the primary key.

Table Definition:

```
CREATE TABLE IF NOT EXISTS absent_film(af_id char(12) NOT NULL PRIMARY KEY AUTO_INCREMENT, af_title varchar(50), af_director varchar(60), af_year char(20), af_genre varchar(20) )
```

6) Card

Relational Model:
card(card_id, balance, card_info)

Functional Dependencies:
card_id -> all

Candidate Keys:
{ (card_id) }

Normal Forms:
3NF
Card has the dependency card_id -> all. This does not disrupt the 3NF as card_id is the primary key.

Table Definition:

```
CREATE TABLE IF NOT EXISTS card(card_id char(12) NOT NULL
PRIMARY KEY AUTO_INCREMENT, balance float(24), card_info char(20) )
```

7) Has

Relational Model:
has(<u>user_id</u>, <u>card_id</u>)

Functional Dependencies:
card_id -> user_id
user_id -> card_id

Candidate Keys:
{ (user_id), (card_id) }

Normal Forms:
3NF
Has has dependencies card_id -> user_id and user_id -> card_id. These do not disrupt the 3NF as user_id and card_id are the primary keys.

Table Definition:

```
CREATE TABLE IF NOT EXISTS has(user_id char(12) NOT NULL,
card_id char(12) NOT NULL, CONSTRAINT has_pk PRIMARY KEY (user_id,
card_id), CONSTRAINT has_pk1 FOREIGN KEY (user_id) REFERENCES user
(user_id) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT has_pk2
FOREIGN KEY (card_id) REFERENCES card (card_id) ON DELETE CASCADE ON
UPDATE CASCADE )
```

8) Add Friend

Relational Model:
add_friend(<u>adder_id</u>, <u>added_id</u>, request_status)

Functional Dependencies:
adder_id, added_id -> request_status

Candidate Keys:

{ (adder_id), (added_id) }

Normal Forms:
3NF
Add Friend has the dependency adder_id, added_id -> request_status. This does not disrupt the 3NF as { (adder_id), (added_id) } is the primary key.

Table Definition:

```
     CREATE TABLE IF NOT EXISTS add_friend(adder_id char(12) NOT
NULL, added_id char(12) NOT NULL, request_status varchar(20),
CONSTRAINT add_friend_pk PRIMARY KEY (adder_id, added_id), CONSTRAINT
add_friend_fk1 FOREIGN KEY (adder_id) REFERENCES user (user_id) ON
DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT add_friend_fk2 FOREIGN
KEY (added_id) REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE
CASCADE )
```

9) Request

Relational Model:
request(<u>user_id</u>, <u>af_id</u>, request_status, request_desc)

Functional Dependencies:
user_id, af_id -> request_status, request_desc

Candidate Keys:
{ (user_id), (af_id) }

Normal Forms:
3NF
Request has the dependency user_id, af_id -> request_status, request_desc. This does not disrupt the 3NF as {(user_id), (af_id)} is the primary key.

Table Definition:

```
     CREATE TABLE IF NOT EXISTS request(user_id char(12) NOT NULL,
af_id char(12) NOT NULL, request_status varchar(20), request_desc
varchar(150), CONSTRAINT request_pk PRIMARY KEY (user_id, af_id),
CONSTRAINT request_fk1 FOREIGN KEY (user_id) REFERENCES user
(user_id) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT request_fk2
FOREIGN KEY (af_id) REFERENCES absent_film (af_id) ON DELETE CASCADE
ON UPDATE CASCADE )
```

10) Review

Relational Model:
review(user_id, f_id, r_date, r_text)

Functional Dependencies:
user_id, f_id -> r_date, r_text

Candidate Keys:
{ (user_id), (f_id) }

Normal Forms:
3NF
Review has the dependency user_id, f_id -> r_date, r_text. This does not disrupt the 3NF as { (user_id), (f_id) } is the primary key.

Table Definition:

```
     CREATE TABLE IF NOT EXISTS review(user_id char(12) NOT NULL,
f_id char(12) NOT NULL, r_date date, r_text varchar(150), CONSTRAINT
review_pk PRIMARY KEY (user_id, f_id), CONSTRAINT review_fk1 FOREIGN
KEY (user_id) REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE
CASCADE, CONSTRAINT review_fk2 FOREIGN KEY (f_id) REFERENCES film
(f_id) ON DELETE CASCADE ON UPDATE CASCADE )
```

11) Rate

Relational Model:
rate(user_id, f_id, r_date, r_rating)

Functional Dependencies:
user_id, f_id -> r_date, r_rating

Candidate Keys:
{ (user_id), (f_id) }

Normal Forms:
3NF
Rate has the dependency user_id, f_id -> r_date, r_rating. This does not disrupt the 3NF as { (user_id), (f_id) } is the primary key.

Table Definition:

```
     CREATE TABLE IF NOT EXISTS rate(user_id char(12) NOT NULL, f_id
char(12) NOT NULL, r_date date, r_rating float(24), CONSTRAINT
rate_pk PRIMARY KEY (user_id, f_id), CONSTRAINT rate_fk1 FOREIGN KEY
(user_id) REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE
CASCADE, CONSTRAINT rate_fk2 FOREIGN KEY (f_id) REFERENCES film
(f_id) ON DELETE CASCADE ON UPDATE CASCADE )
```

12) Recommend

Relational Model:
recommend(recommender_id, reciever_id, f_id)

Functional Dependencies:
-

Candidate Keys:
{ (recommender_id), (receiver_id), (f_id) }

Normal Forms:
3NF
Recommend has no functional dependency, all of its columns make up the primary key. Therefore it is in 3NF.

Table Definition:
```
     CREATE TABLE IF NOT EXISTS recommend(recommender_id char(12)
NOT NULL, receiver_id char(12) NOT NULL, f_id char(12) NOT NULL,
CONSTRAINT recommend_pk PRIMARY KEY (recommender_id, receiver_id,
f_id), CONSTRAINT recommend_fk1 FOREIGN KEY (recommender_id)
REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT recommend_fk2 FOREIGN KEY (receiver_id) REFERENCES user
(user_id) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT
recommend_fk3 FOREIGN KEY (f_id) REFERENCES film (f_id) ON DELETE
CASCADE ON UPDATE CASCADE )
```

13) Rent

Relational Model:
rent(user_id, f_id, rent_date, rent_status)

Functional Dependencies:
user_id, f_id -> rent_date, rent_status

Candidate Keys:
{ (user_id), (f_id) }

Normal Forms:

3NF

Rent has the dependency user_id, f_id -> rent_date, rent_status. This does not disrupt the 3NF as { (user_id), (f_id) } is the primary key.

Table Definition:

```
CREATE TABLE IF NOT EXISTS rent(user_id char(12) NOT NULL, f_id char(12) NOT NULL, rent_date date, rent_status varchar(20), CONSTRAINT rent_pk PRIMARY KEY (user_id, f_id), CONSTRAINT rent_fk1 FOREIGN KEY (user_id) REFERENCES user (user_id) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT rent_fk2 FOREIGN KEY (f_id) REFERENCES film (f_id) ON DELETE CASCADE ON UPDATE CASCADE )
```

14) Series

Relational Model:

series(<u>series_name</u>, series_desc)

Functional Dependencies:

series_name -> all

Candidate Keys:

{ (series_name) }

Normal Forms:

3NF

Series has the dependency series_name -> all. This does not disrupt the 3NF as series_name is the primary key.

Table Definition:

```
CREATE TABLE IF NOT EXISTS series(series_name char(50) NOT NULL PRIMARY KEY, series_desc char(150) )
```

15) Part_of

Relational Model:

part_of(<u>f_id</u>, <u>series_name</u>, order_no)

Functional Dependencies:

f_id, series_name -> order_no

Candidate Keys:

{ (f_id), (series_name) }

Normal Forms:

3NF

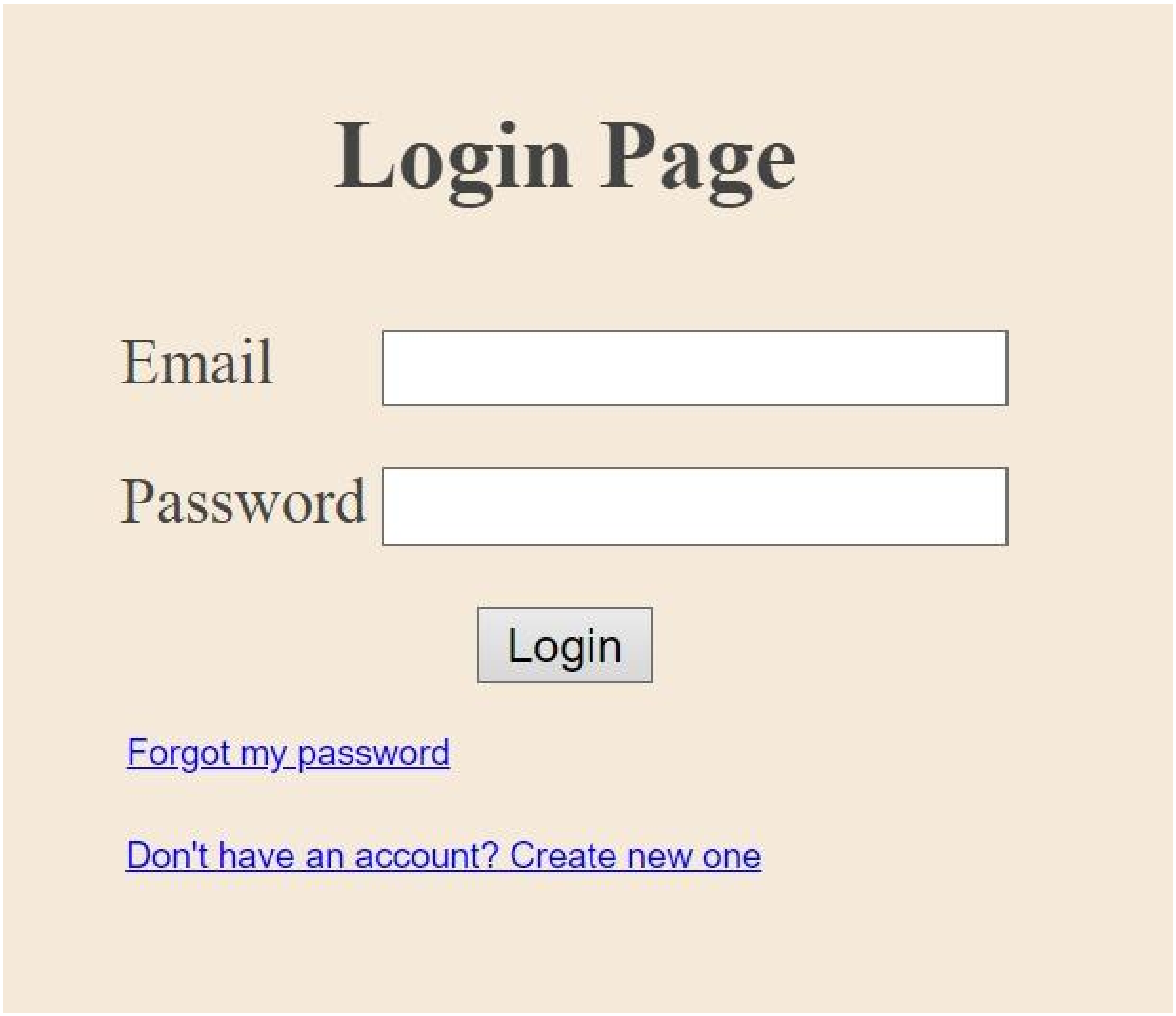
Part_of has the dependency f_id, series_name -> order_no. This does not disrupt the 3NF as { (f_id), (series_name) } is the primary key.

Table Definition:

```
    CREATE TABLE IF NOT EXISTS part_of(f_id char(12) NOT NULL,
series_name char(50) NOT NULL, order_no int, CONSTRAINT part_of_pk
PRIMARY KEY (f_id, series_name), CONSTRAINT part_of_fk1 FOREIGN KEY
(f_id) REFERENCES film (f_id) ON DELETE CASCADE ON UPDATE CASCADE,
CONSTRAINT part_of_fk2 FOREIGN KEY (series_name) REFERENCES series
(series_name) ON DELETE CASCADE ON UPDATE CASCADE )
```

GUI Design and Corresponding SQL Statements

1) Login Page



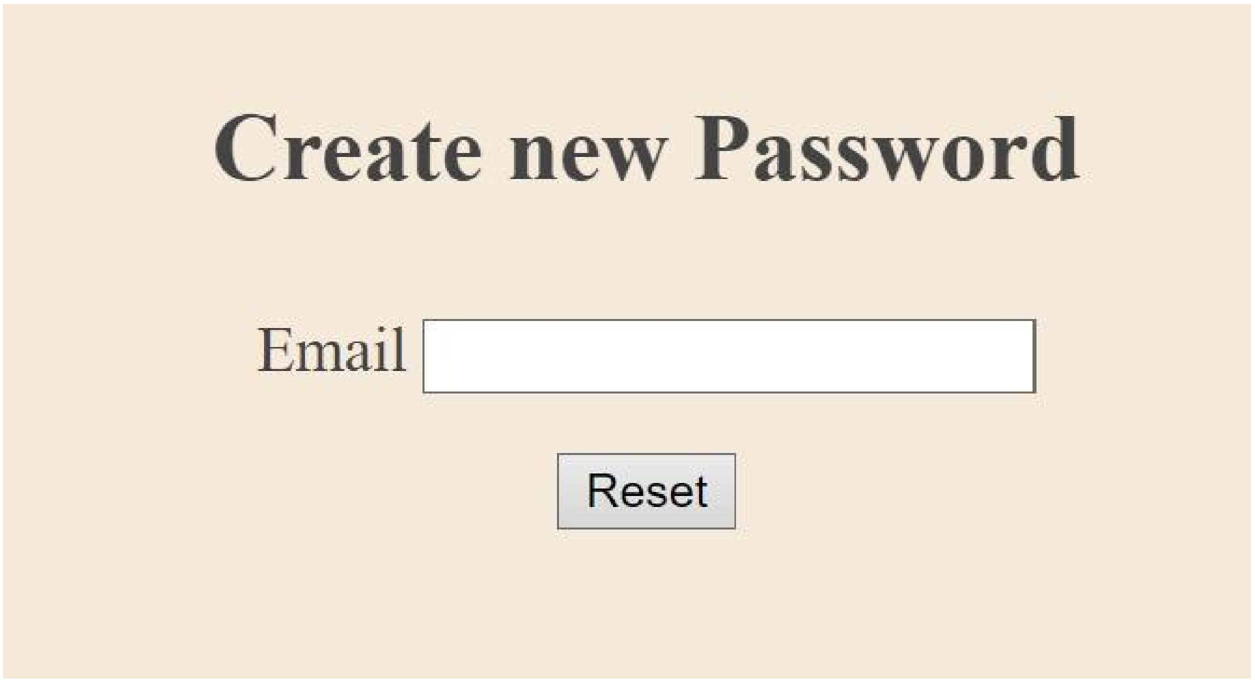
The system will require users to login by having their email and password entered as credentials. The page will have links to Create New Password and Create Account routes. Clicking Login will execute a function that:

- Hashes the password
- Executes the query:

```
SELECT user_id
FROM user
WHERE user_mail = '"+req_user_mail+"' AND user_password =
'"+req_user_password+"'
```

- Checks if the result is null, and grants access accordingly.

2) Forgot Password Page



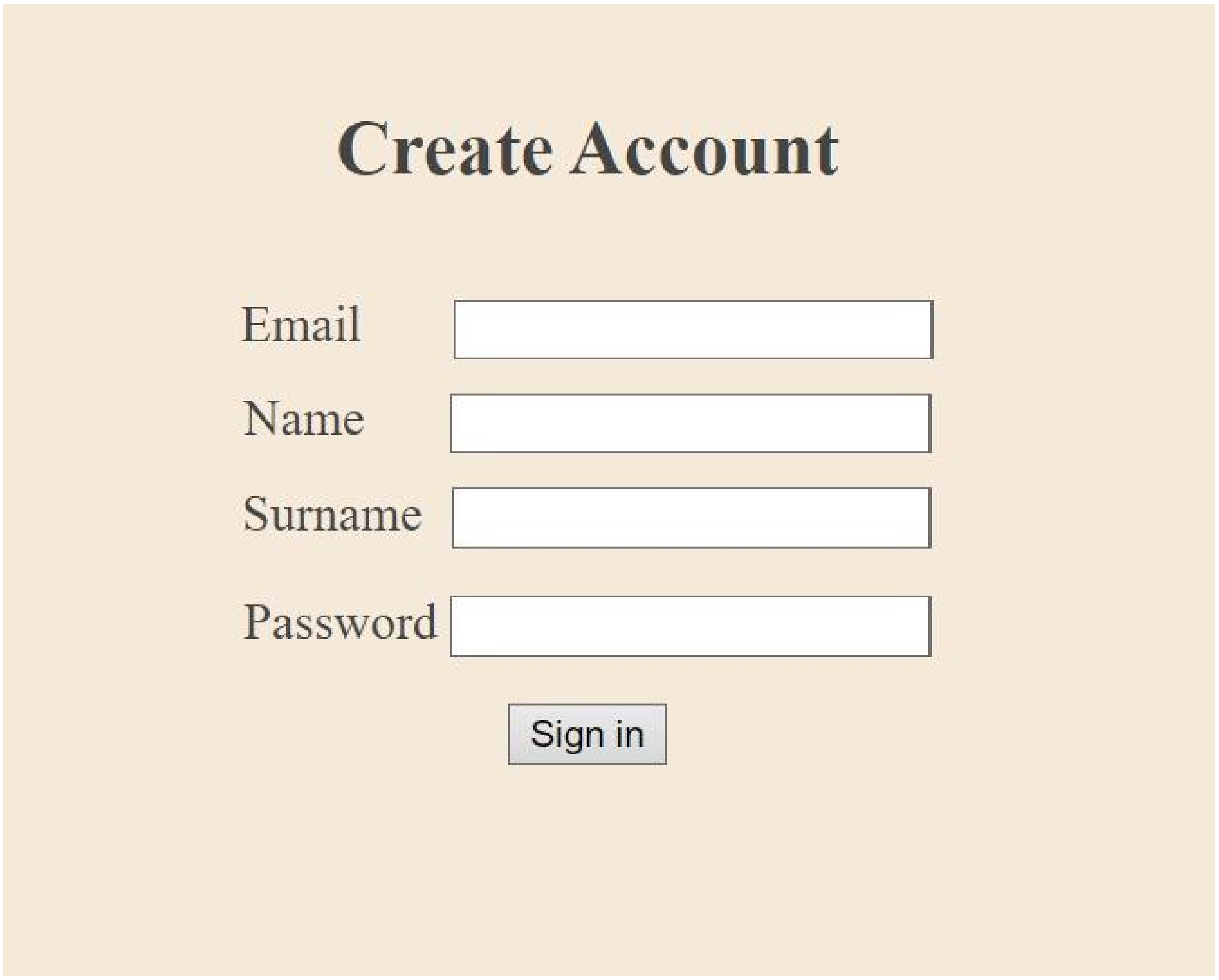
If a user forgets their password, the system lets the user change the password by sending password resetting instructions to the authorized email. The button will:

- Execute the query:

```
SELECT * FROM USERS
WHERE user_mail = '"+req_user_mail+"'
```

- If the result is not null, send the password resetting email.

3) Create Account Page



New users will be able to create their accounts by entering an email, their names, and a password. The Sign in button will:

- Check if the values in input fields are valid.
- Execute the query:

```
SELECT * FROM USERS
WHERE user_mail = '"+req_user_mail+"'
```

- If the result is not null, display an error message, else
  - Hash the password
  - Execute the query

```
    INSERT INTO user (user_name, user_surname, user_mail,
    user_password ) VALUES ('"+user_name+"', '"+user_surname+"',
    '"+user_mail+"', '"+user_password+"' )
```

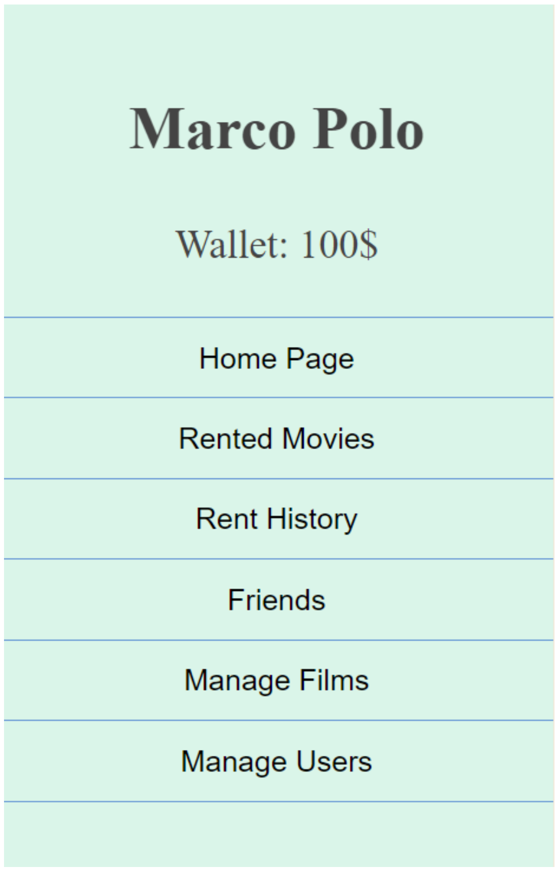
  - Check whether the user is an employee or a customer by email. If they are an employee, execute:

```
    INSERT INTO employee (user_id, salary)
    VALUES ('"+user_id+"', '"+salary+"')
```

- Else Execute:

```
INSERT INTO customer (user_id, join_date)
VALUES ('"+user_id+"', '"+join_date+"')
```

4) Left Menu



A menu will be displayed on the left side of the screen as a column.
The name of the user to be displayed will be obtained and kept from the query previously executed during login.

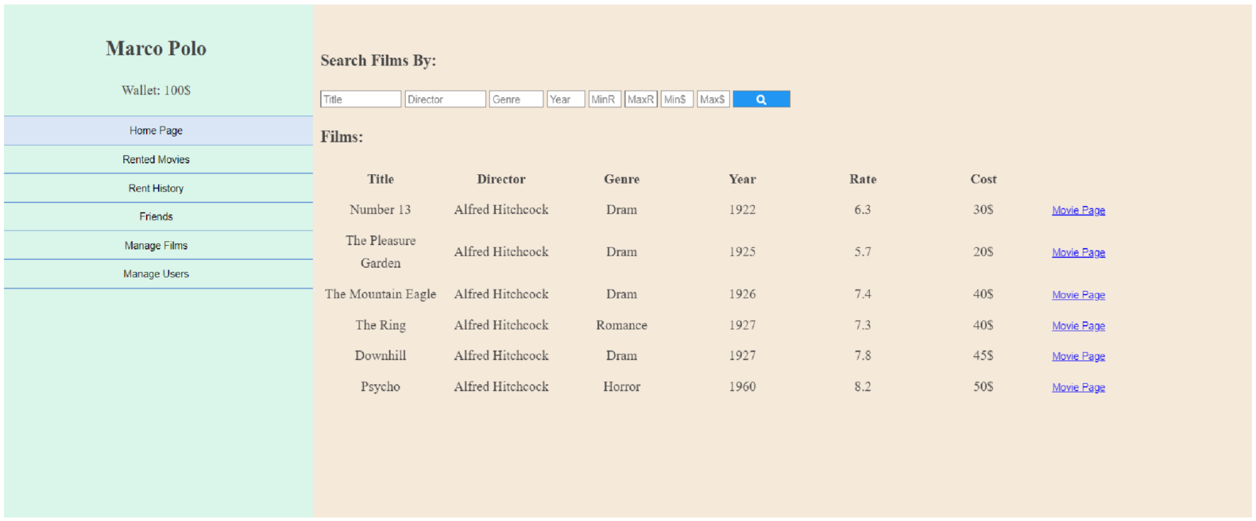
The menu will also display the balance on the user's wallet by executing the following query:

```
SELECT C1.balance
FROM card as C1, customer as C2, has as C3
WHERE C1.card_id = C3.card_id AND C2.user_id = '"+user_id+"'
AND C3.user_id = C2.user_id;
```

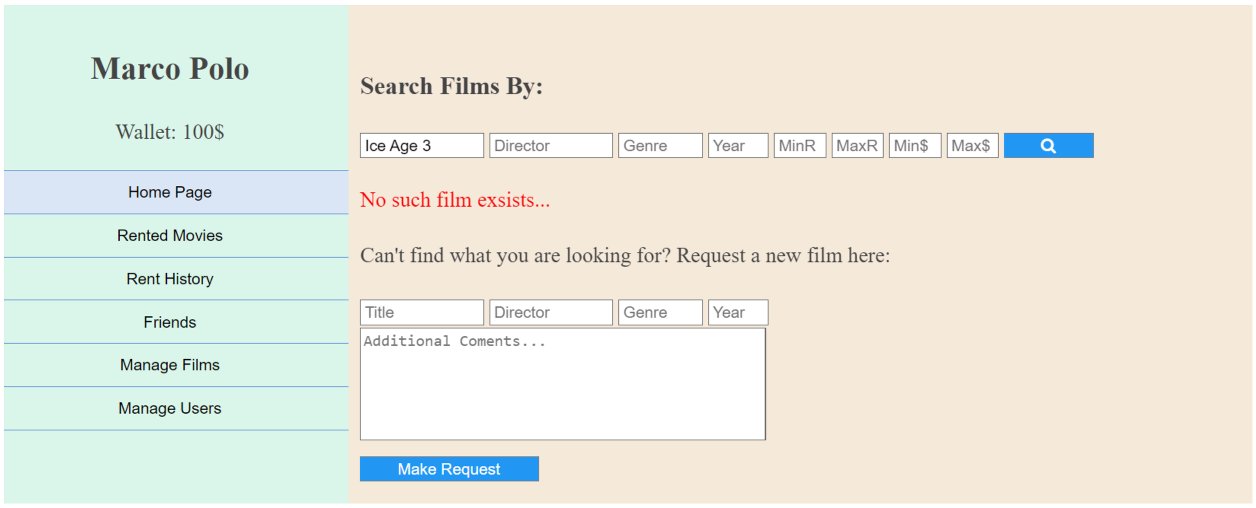
The buttons that read "Manage Films" and "Manage Users" are shown only for employee users. The following query will be executed to determine whether they will be displayed if the following query is not null:

```
SELECT * FROM employee WHERE user_id = '"+user_id+"'
```

5) Home Page



Title	Director	Genre	Year	Rate	Cost	
Number 13	Alfred Hitchcock	Dram	1922	6.3	30$	Movie Page
The Pleasure Garden	Alfred Hitchcock	Dram	1925	5.7	20$	Movie Page
The Mountain Eagle	Alfred Hitchcock	Dram	1926	7.4	40$	Movie Page
The Ring	Alfred Hitchcock	Romance	1927	7.3	40$	Movie Page
Downhill	Alfred Hitchcock	Dram	1927	7.8	45$	Movie Page
Psycho	Alfred Hitchcock	Horror	1960	8.2	50$	Movie Page



No such film exsists...

Can't find what you are looking for? Request a new film here:

The homepage will display the films. The films to be displayed can be filtered by search criteria of the user's choice. Those criteria include minimum rating, maximum rating, minimum price, maximum price, and search key strings for the title, director, and genre. Fields that display the films will include a link to the page of that movie. If the search criteria return no films, the request film menu will be displayed. Initially, on the page, the following will be executed:

- Execute the query:

```
SELECT f_title, f_director, f_year, f_rating, f_genre, f_price
FROM film
```

- Display the results of the query

When search criteria are updated, the following will be executed.

- Execute the query:

```
SELECT f_title, f_director, f_year, f_rating, f_genre, f_price
FROM film
WHERE ( ("+f_title+" IS NULL) OR (f_title = "+f_title+") ) AND (
("+f_director+" IS NULL) OR (f_director = "+f_director+") ) AND (
("+f_year+" IS NULL) OR (f_year = "+f_year+") ) AND ( ("+f_genre+" IS
NULL) OR (f_genre = "+f_genre+") ) AND ( ("+minr+" IS NULL) OR
(f_rating > "+minr+") ) AND ( ("+maxr+" IS NULL) OR (f_rating <
"+maxr+") ) AND ( ("+minp+" IS NULL) OR (f_price > "+minp+") ) AND (
("+maxp+" IS NULL) OR (f_price < "+maxp+") )
```

- Display the results of the query

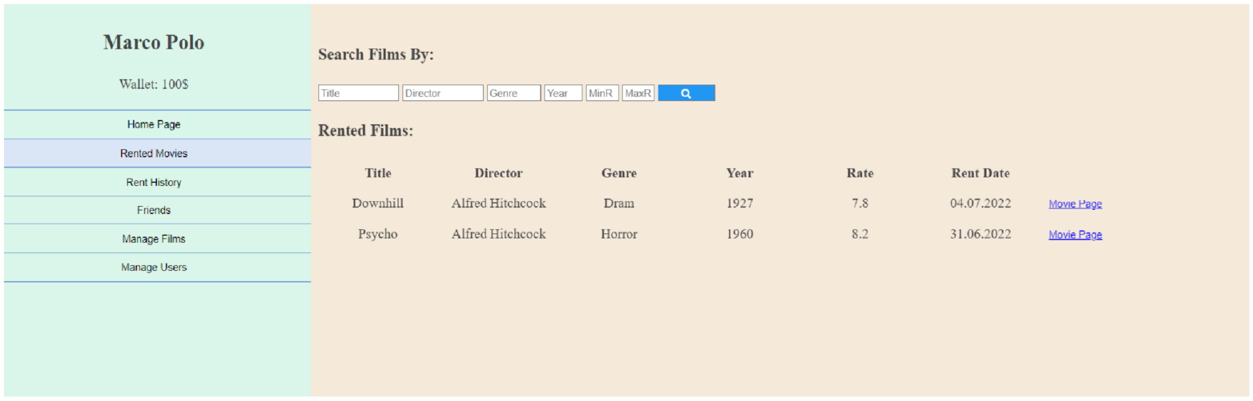
The reason for this long query string is that search boxes might be empty. However, before execution, if the values are not null, the character (‘) has to be added to any value string of +VALUE+. This can be done in PHP.

- If the query returns null, a small menu to let users request films will be displayed.
- A user, by filling the according fields with the title, director, genre, year of the film and with optional comments about the request, can execute the following queries subsequently:

```
INSERT INTO absent_film ( af_title, af_director, af_year, af_genre )
VALUES ( '"+af_title+"', '"+af_director+"', '"+user_year+"',
'"+af_genre+"' )

INSERT INTO request ( user_id, af_id, request_status, request_desc )
VALUES ( '"+user_id+"', '"+af_id+"', "Pending", '"+request_desc+"' )
```

6) Rented Movies Page


Marco Polo

Wallet: 100$

Rented Films:

Title	Director	Genre	Year	Rate	Rent Date	
Downhill	Alfred Hitchcock	Dram	1927	7.8	04.07.2022	Movie Page
Psycho	Alfred Hitchcock	Horror	1960	8.2	31.06.2022	Movie Page

The rented movies page will display the films that are rented by the user currently, in a similar way to the home page. The films to be displayed can be filtered by search criteria of the user's choice. Those criteria include the same ones from the home page.

- Execute the query:

```
SELECT F.f_title, F.f_director, F.f_year, F.f_rating, F.f_genre,
F.f_price
```

```
FROM film as F, rent as R
WHERE R.rent_status = "Ongoing" AND R.user_id = '"+user_id+"' AND
R.f_id = F.f_id
```

- Display the results of the query

When search criteria are updated, the following will be executed.

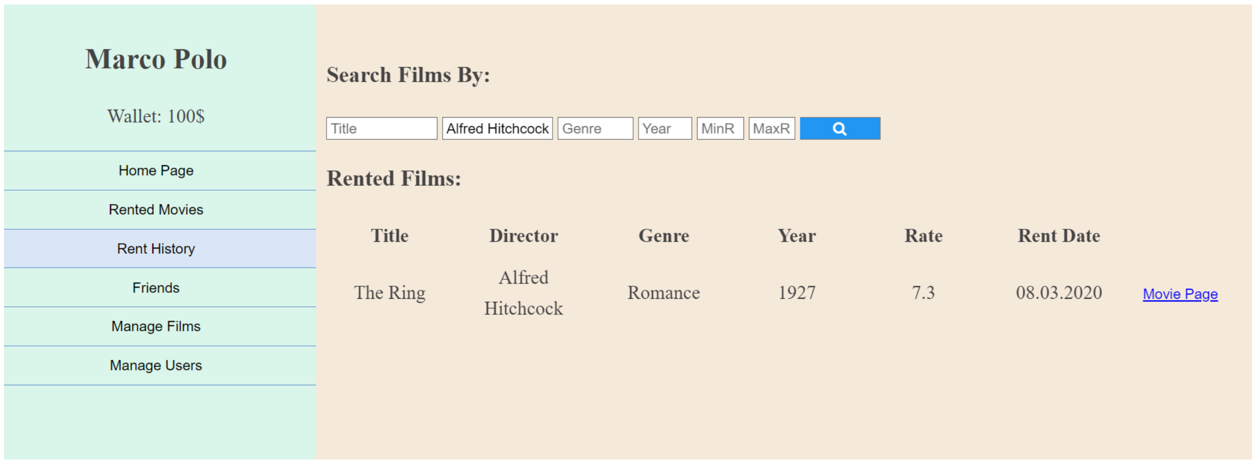
- Execute the query:

```
SELECT F.f_title, F.f_director, F.f_year, F.f_rating, F.f_genre,
F.f_price
FROM film as F, rent as R
WHERE R.rent_status = \"Ongoing\" AND R.user_id = '"+user_id+"' AND
R.f_id = F.f_id AND ( ("+f_title+" IS NULL) OR (f_title =
"+f_title+") ) AND ( ("+f_director+" IS NULL) OR (f_director =
"+f_director+") ) AND ( ("+f_year+" IS NULL) OR (f_year = "+f_year+")
) AND ( ("+f_genre+" IS NULL) OR (f_genre = "+f_genre+") ) AND (
("+minr+" IS NULL) OR (f_rating > "+minr+") ) AND ( ("+maxr+" IS
NULL) OR (f_rating < "+maxr+") )
```

- Display the results of the query

The reason for this long query string is that search boxes might be empty, as on the home page. The same thing can be done with several queries using if clauses in PHP before execution.

7) Rent History Page



Title	Director	Genre	Year	Rate	Rent Date	
The Ring	Alfred Hitchcock	Romance	1927	7.3	08.03.2020	Movie Page

The rent history page will display the films that were rented by the user and expired. The display will be similar to the way it is on the home page. The films to be displayed can be filtered by search criteria of the user's choice. Those criteria include the same ones from the home page.

- Execute the query:

```
SELECT F.f_title, F.f_director, F.f_year, F.f_rating, F.f_genre,
F.f_price
FROM film as F, rent as R
```

```
WHERE R.rent_status = "Expired" AND R.user_id = '"+user_id+"' AND
R.f_id = F.f_id
```

- Display the results of the query

When search criteria are updated, the following will be executed.

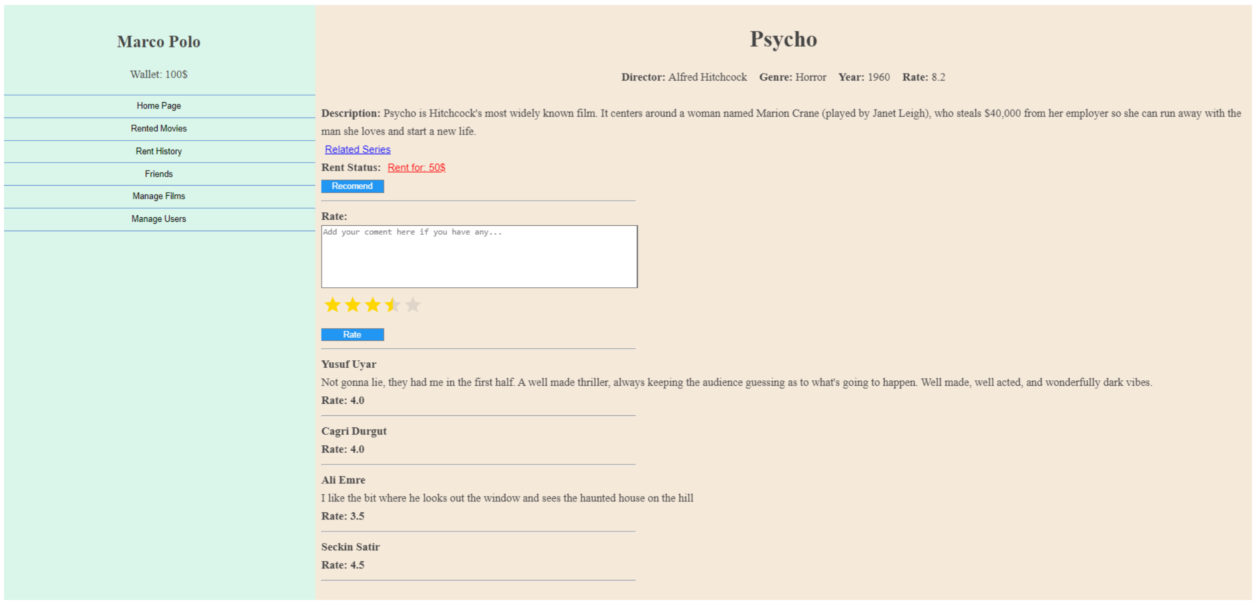
- Execute the query:

```
SELECT F.f_title, F.f_director, F.f_year, F.f_rating, F.f_genre,
F.f_price
FROM film as F, rent as R
WHERE R.rent_status = \"Expired\" AND R.user_id = '"+user_id+"' AND
R.f_id = F.f_id AND ( ("+f_title+" IS NULL) OR (f_title =
"+f_title+") ) AND ( ("+f_director+" IS NULL) OR (f_director =
"+f_director+") ) AND ( ("+f_year+" IS NULL) OR (f_year = "+f_year+")
) AND ( ("+f_genre+" IS NULL) OR (f_genre = "+f_genre+") ) AND (
("+minr+" IS NULL) OR (f_rating > "+minr+") ) AND ( ("+maxr+" IS
NULL) OR (f_rating < "+maxr+") )
```

- Display the results of the query

The reason for this long query string is that search boxes might be empty, as on the home page. The same thing can be done with several queries using if clauses in PHP before execution.

8) Movie Viewing Page



Psycho

Director: Alfred Hitchcock **Genre:** Horror **Year:** 1960 **Rate:** 8.2

Description: Psycho is Hitchcock's most widely known film. It centers around a woman named Marion Crane (played by Janet Leigh), who steals $40,000 from her employer so she can run away with the man she loves and start a new life.

Rent Status: Rent for: 50$

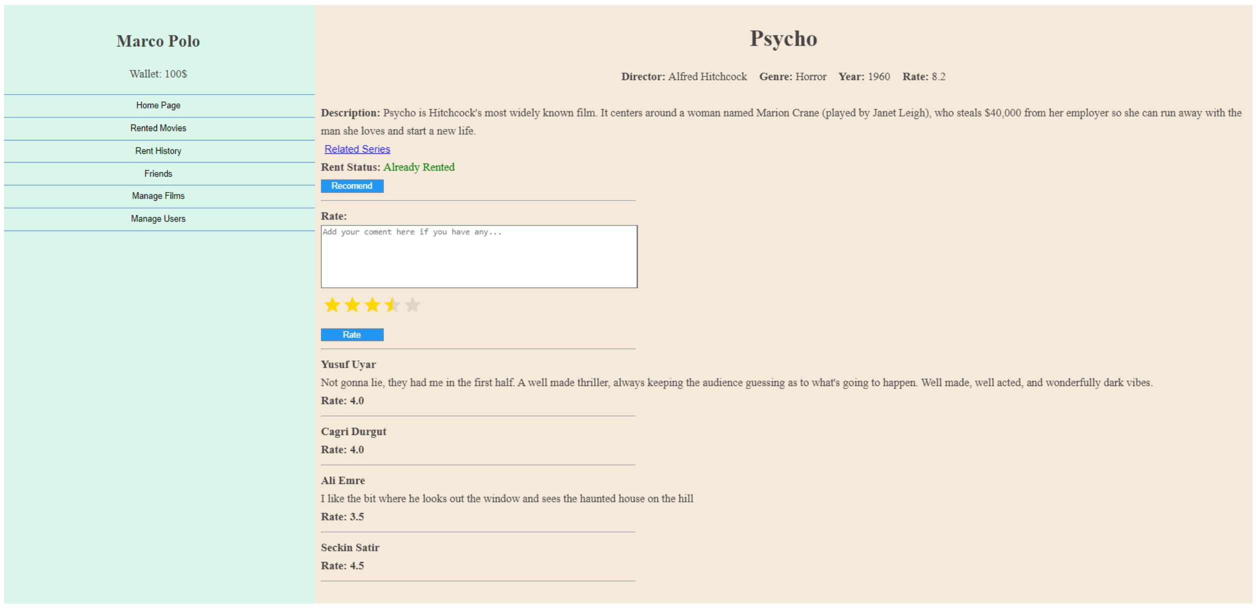
Rate:

Yusuf Uyar
Not gonna lie, they had me in the first half. A well made thriller, always keeping the audience guessing as to what's going to happen. Well made, well acted, and wonderfully dark vibes.
Rate: 4.0

Cagri Durgut
Rate: 4.0

Ali Emre
I like the bit where he looks out the window and sees the haunted house on the hill
Rate: 3.5

Seckin Satir
Rate: 4.5



Psycho

Director: Alfred Hitchcock **Genre:** Horror **Year:** 1960 **Rate:** 8.2

Description: Psycho is Hitchcock's most widely known film. It centers around a woman named Marion Crane (played by Janet Leigh), who steals $40,000 from her employer so she can run away with the man she loves and start a new life.

Rent Status: Already Rented

Rate:

Yusuf Uyar
Not gonna lie, they had me in the first half. A well made thriller, always keeping the audience guessing as to what's going to happen. Well made, well acted, and wonderfully dark vibes.
Rate: 4.0

Cagri Durgut
Rate: 4.0

Ali Emre
I like the bit where he looks out the window and sees the haunted house on the hill
Rate: 3.5

Seckin Satir
Rate: 4.5

Movies will have Viewing pages assigned for them. On this page; basic information about the movie, comments made by users about the movie, and information about if the user has rented the film will be displayed. If not, a rent button with the price of the film will be displayed. There also will be a button that displays related series of the film, if the movie is related to any series. There will be a menu for the user to make their own comments and ratings about the film. With the recommend button, the user can recommend the film to their friends.

General information about the film will be obtained by the following query:

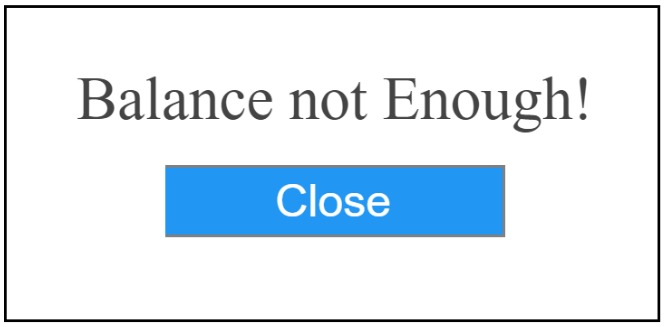
```
SELECT f_title, f_director, f_year, f_rating, f_genre, f_desc
FROM film
WHERE f_id = '"+f_id+"'
```

The rental status will be checked by whether the following query returns null:

```
SELECT * FROM rent
WHERE user_id = '"+user_id+"' AND rent_status = "Ongoing" AND f_id =
'"+f_id+"'
```

For the user to rent the movie, first if the balance is enough will be checked. If the following query is null, the user will not be able to rent the movie:

```
SELECT C1.balance FROM card as C1, user as C2, has as C3 WHERE
C1.card_id = C3.card_id AND C2.user_id = C3.user_id AND C2.user_id =
'"+user_id+"' AND C1.balance > '"+rent_cost+"'
```



Then, if the query is not null, the balance of the user will be updated:

```
UPDATE card, has SET card.balance = card.balance - '"+rent_cost+"'
WHERE card.card_id = has.card_id AND has.user_id = '"+user_id+"'
```

Reviews of the movie will be gathered with the following query:

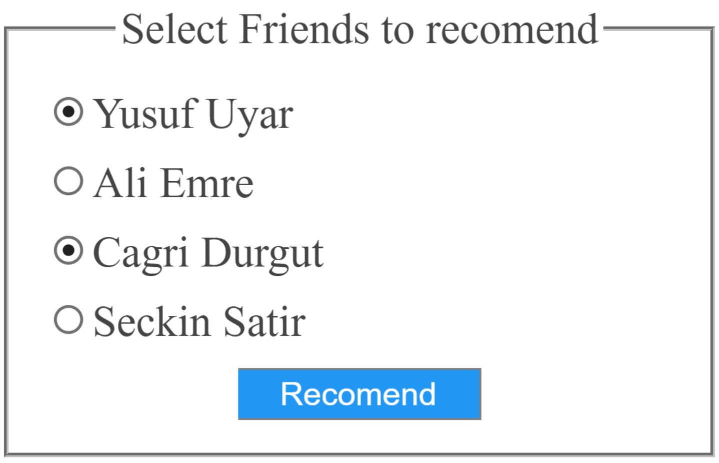
```
SELECT C1.user_name, C1.user_surname, C2.r_text, C2.r_rating
FROM customer as C3, review as C2, user as C1
WHERE C1.user_id = C2.user_id AND C2.f_id = '"+f_id+"' AND C3.user_id
= C1.user_id
```

And the ratings of the movie will be gathered with the following query:

```
SELECT C1.user_name, C1.user_surname, C2.r_date, C2.r_rating
FROM customer as C3, rate as C2, user as C1
WHERE C1.user_id = C2.user_id AND C2.f_id = 4 AND C3.user_id =
C1.user_id
```

Upon clicking recommend, get friends list by:

```
SELECT C2.user_name, C2.user_surname
FROM add_friend as C1, user as C2
WHERE ( C1.adder_id = '"+user_id+"' OR C1.added_id = '"+user_id+"' )
AND request_status = "Accepted" AND ( C2.user_id = C1.adder_id OR
C2.user_id = C1.added_id ) AND C2.user_id <> '"+user_id+"'
```



After selecting a friend and clicking recommend, the following query will be executed to make the recommendation:

```
INSERT INTO recommend (recommender_id, receiver_id, f_id) VALUES
('"+user_id+"', '"+selected_friend_id+"', '"+f_id+"' )
```

After clicking the Rate button;
If the text box where a review should be written is not null:

```
INSERT INTO review (user_id, f_id, r_date, r_text)
VALUES('"+user_id+"', '"+f_id+"', '"+current_date+"', '"+r_text+"'
```

Rating where r_rating value is derived from the stars in php:

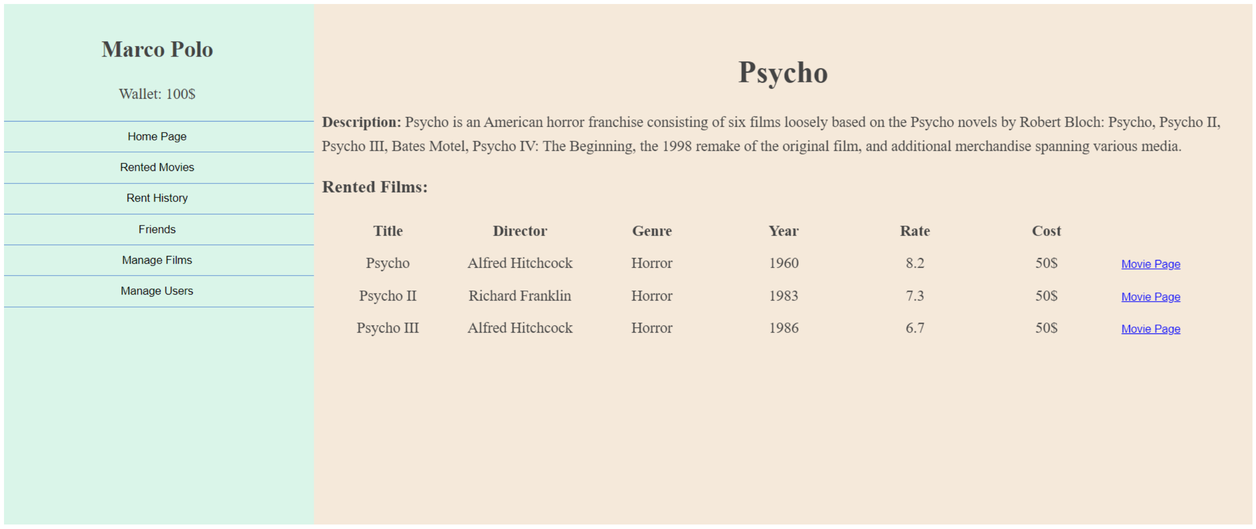
```
INSERT INTO rate (user_id, f_id, r_date, r_rating)VALUES
('"+user_id+"', '"+f_id+"', '"+current_date+"', '"+r_rating+"' )
```

```
CREATE TRIGGER film_rating
AFTER INSERT ON rate
FOR EACH ROW
BEGIN
UPDATE film SET f_rating = (SELECT avg(f_rating) FROM rate where
rate.f_id = NEW.f_id)END
```

Query to check if the film is a part of any series (null result means it's not):

```
SELECT * FROM part_of
WHERE f_id= '"+f_id+"'
```

9) Series


Marco Polo

Wallet: 100$

- Home Page
- Rented Movies
- Rent History
- Friends
- Manage Films
- Manage Users

Psycho

Description: Psycho is an American horror franchise consisting of six films loosely based on the Psycho novels by Robert Bloch: Psycho, Psycho II, Psycho III, Bates Motel, Psycho IV: The Beginning, the 1998 remake of the original film, and additional merchandise spanning various media.

Rented Films:

Title	Director	Genre	Year	Rate	Cost	
Psycho	Alfred Hitchcock	Horror	1960	8.2	50$	Movie Page
Psycho II	Richard Franklin	Horror	1983	7.3	50$	Movie Page
Psycho III	Alfred Hitchcock	Horror	1986	6.7	50$	Movie Page

The series page will display a description of the series and a list of the movies it is related to. As it will be linked from a movie page, that specific movie's id will be the parameter for the following query to get that information;

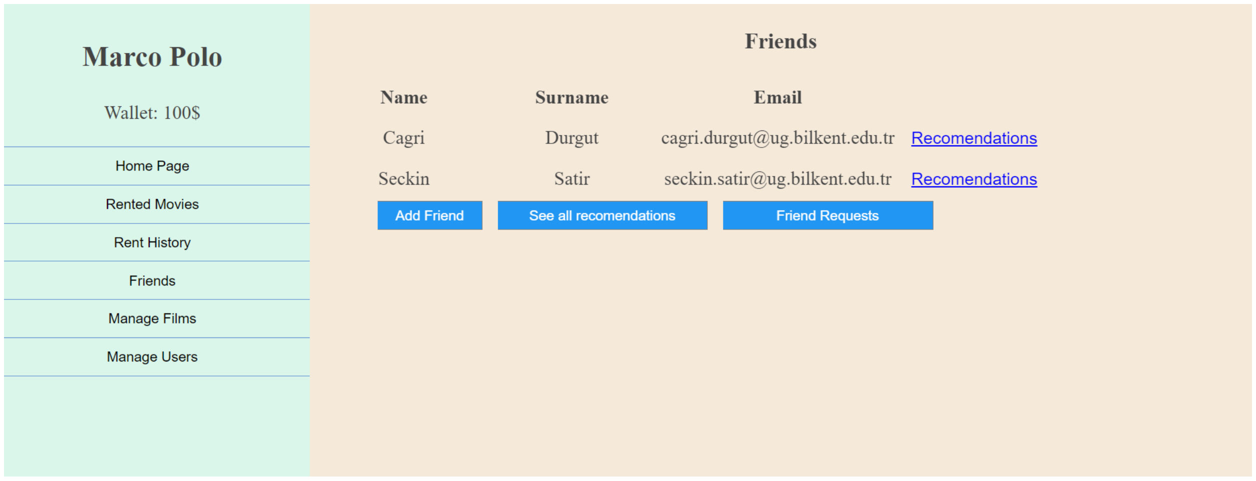
Getting series description:

```
SELECT series_desc FROM series WHERE series_name = '"+series_name+"'
```

Getting movie list:

```
SELECT C1.f_title, C1.f_director, C1.f_genre, C1.f_year, C1.f_rating,
C1.f_price FROM film as C1, part_of as C2 WHERE C1.f_id = C2.f_id AND
C2.series_name = '"+series_name+"'
```

10) Friends


Marco Polo

Wallet: 100$

- Home Page
- Rented Movies
- Rent History
- Friends
- Manage Films
- Manage Users

Friends

Name	Surname	Email	
Cagri	Durgut	cagri.durgut@ug.bilkent.edu.tr	Recomendations
Seckin	Satir	seckin.satir@ug.bilkent.edu.tr	Recomendations

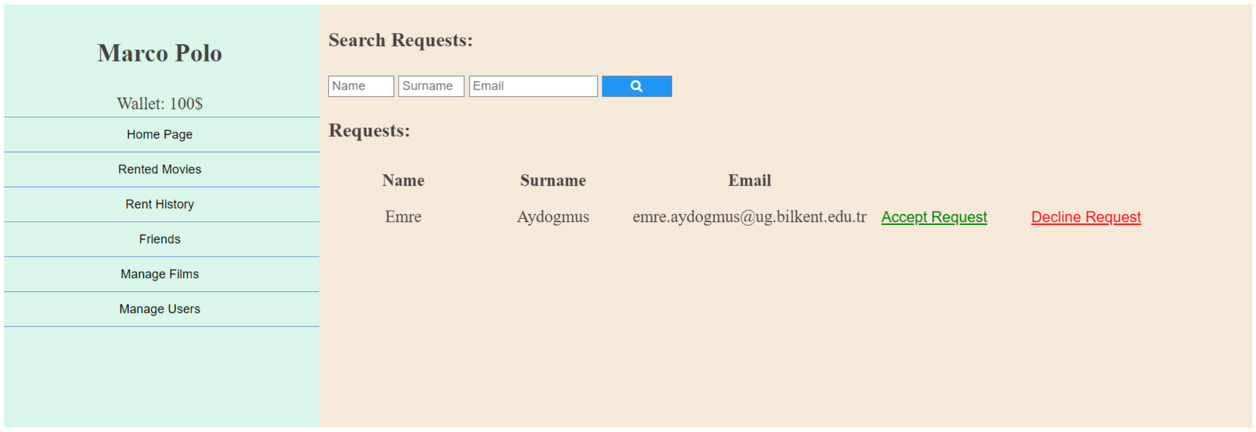
In the friends page, a user's friends will be listed. The information to be displayed will be obtained by the following query:

```
SELECT C2.user_name, C2.user_surname, C2.user_mail
FROM add_friend as C1, user as C2
WHERE ( C1.adder_id = '"+user_id+"' OR C1.added_id = '"+user_id+"' )
AND request_status = "Accepted" AND ( C2.user_id = C1.adder_id OR
C2.user_id = C1.added_id ) AND C2.user_id <> '"+user_id+"'
```

There will be buttons next to that information that links to the recommendations that a friend made. The button that reads “See all recommendations” will link to the list of the recommendations made by all of the user’s friends.

There will be two buttons that read “Add Friend” and “Friend Requests”. Those buttons will link to those pages accordingly.

11) Friend Requests



Name	Surname	Email
Emre	Aydogmus	emre.aydogmus@ug.bilkent.edu.tr

The friend requests page will display a list of people who want to add the user as a friend. The user can filter by using name, surname, or email as a search key.

Query to obtain friend requests:

```
SELECT C2.user_name, C2.user_surname, C2.user_mail
FROM add_friend as C1, user as C2
WHERE C1.added_id = '"+user_id+"' AND C1.adder_id = C2.user_id AND
request_status = "Pending"
```

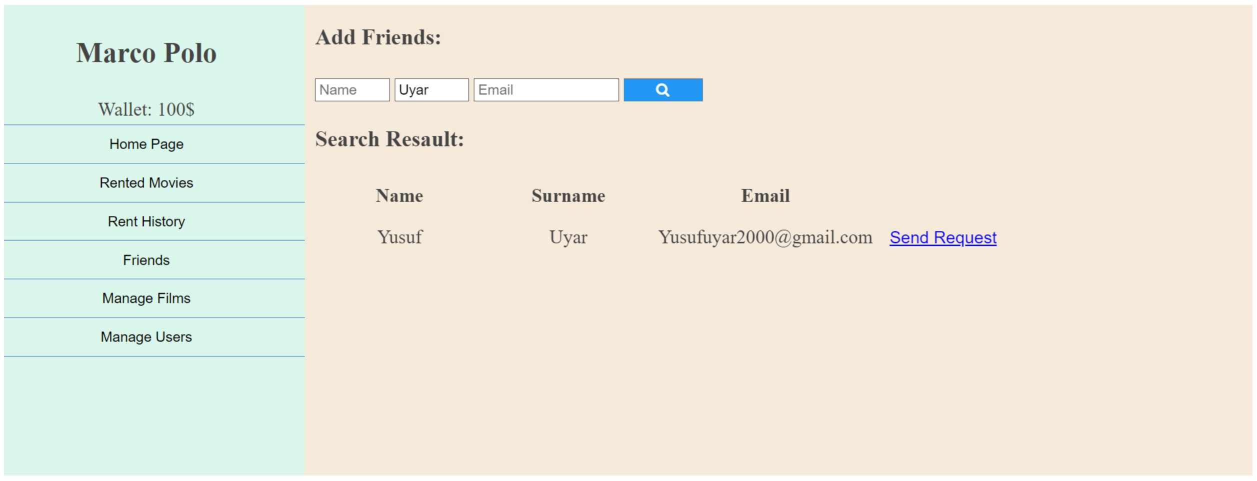
The button that says “Accept Request” will execute the following SQL statement to accept the friend request:

```
UPDATE add_friend, user SET add_friend.request_status = "Accepted"
WHERE add_friend.added_id = '"+user_id+"' AND add_friend.adder_id =
'"+requester_id+"'
```

The button that says “Decline Request” will execute the following SQL statement to decline the friend request:

```
DELETE FROM add_friend WHERE add_friend.added_id = '"+user_id+"' AND
add_friend.adder_id = '"+requester_id+"'
```

12) Add Friend



Name	Surname	Email	
Yusuf	Uyar	Yusufuyar2000@gmail.com	Send Request

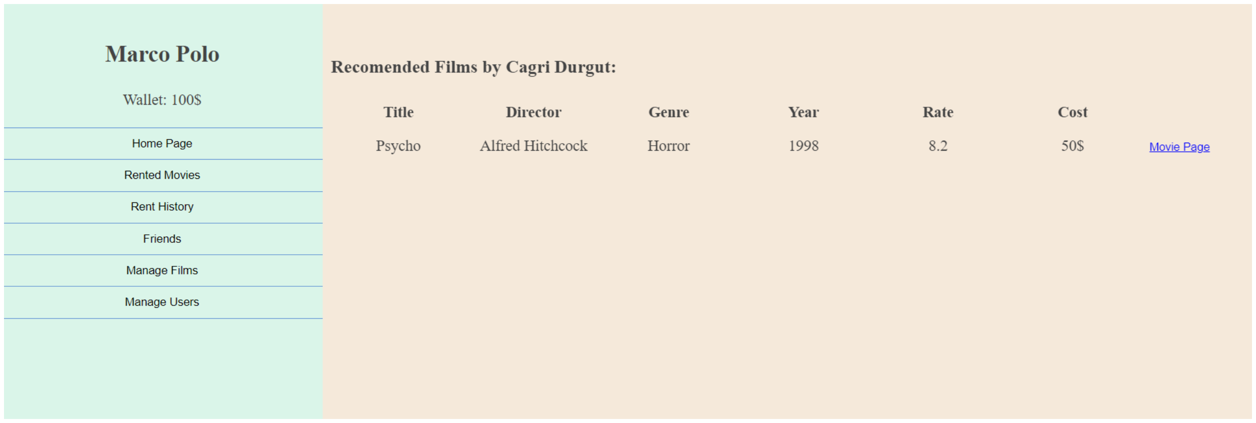
On the Add Friend Page, a user will be able to search for a friend using name, surname, or email by executing the following query:

```
SELECT user_name, user_surname, user_mail
FROM user
WHERE ( ("+search_user_name+" IS NULL) OR (user_name =
"+search_user_name+") ) AND ( ("+search_user_surname+" IS NULL) OR
(user_surname = "+search_user_surname+") ) AND (
("+search_user_mail+" IS NULL) OR (user_mail = "+search_user_mail+")
) AND user_id <> '"+user_id+"'
```

The results of the query will be displayed as a list. From the button to the right, the user will be able to send a friendship request with the following query:

```
INSERT INTO add_friend (adder_id, added_id, request_status) VALUES
('"+user_id+"', '"+selected_user_id+"', "Pending" )
```

13) Recommendation From a Friend



Recomended Films by Cagri Durgut:

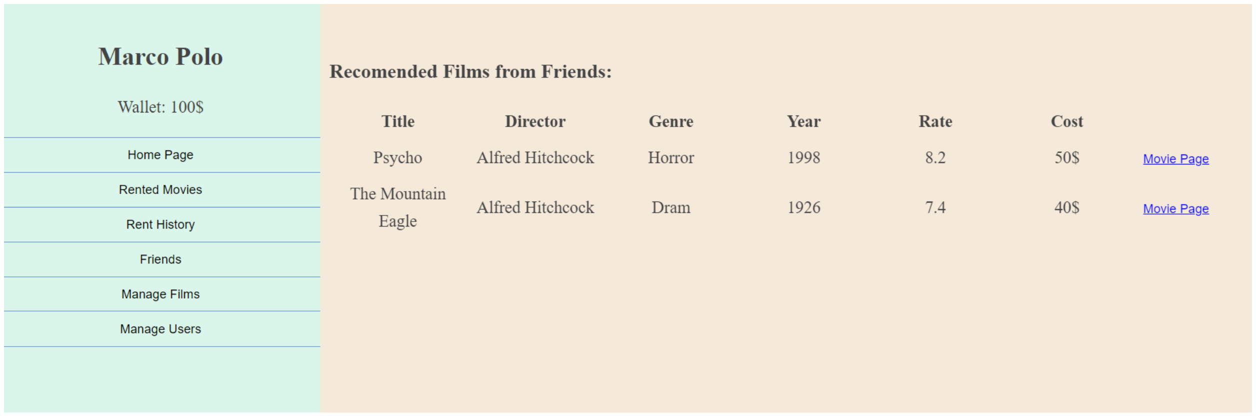
Title	Director	Genre	Year	Rate	Cost	
Psycho	Alfred Hitchcock	Horror	1998	8.2	50$	Movie Page

In the Recommendation page from a specific friend, the following query will be executed to get a list of recommendations that the friend made:

```
SELECT C1.f_title, C1.f_director, C1.f_genre, C1.f_year, C1.f_rating,
C1.f_price
FROM film as C1, recommend as C2
WHERE C2.recommender_id = '"+friend_id+"' AND C2.receiver_id =
'"+user_id+"' AND C1.f_id = C2.f_id
```

As it will be linked from a friend, that specific friend's name will be taken to be displayed at the header.

14) Recommendation From Friends



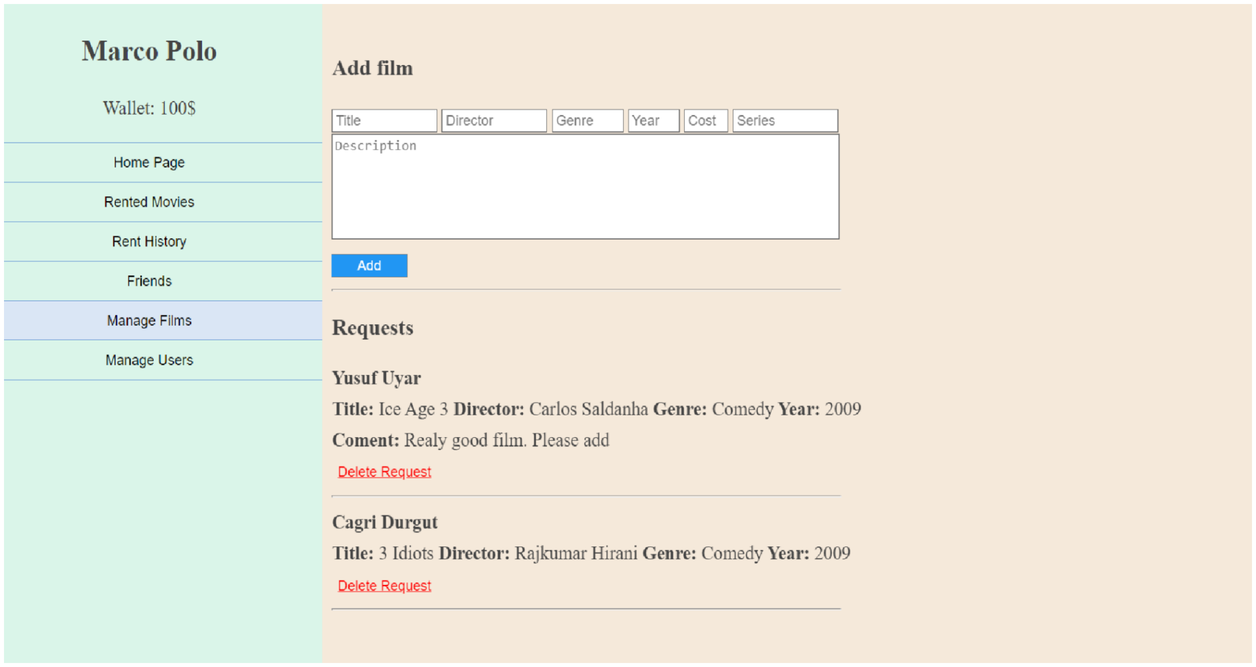
Recomended Films from Friends:

Title	Director	Genre	Year	Rate	Cost	
Psycho	Alfred Hitchcock	Horror	1998	8.2	50$	Movie Page
The Mountain Eagle	Alfred Hitchcock	Dram	1926	7.4	40$	Movie Page

On the Recommendation page from a specific friend. The following query will be executed to get a list of recommendations that were made by all of the user's friends. The information will be obtained from the result of the following query:

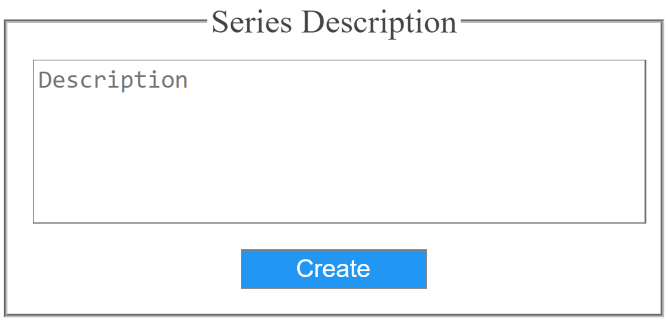
```
SELECT C1.f_title, C1.f_director, C1.f_genre, C1.f_year, C1.f_rating,
C1.f_price
FROM film as C1, recommend as C2
WHERE C2.receiver_id = '"+user_id+"' AND C1.f_id = C2.f_id
```

15) Manage Films



Yusuf Uyar
Title: Ice Age 3 **Director:** Carlos Saldanha **Genre:** Comedy **Year:** 2009
Coment: Realy good film. Please add

Cagri Durgut
Title: 3 Idiots **Director:** Rajkumar Hirani **Genre:** Comedy **Year:** 2009



In the manage films page, a list of film requests made by users are shown. The list is obtained by the following query:

```
SELECT C1.user_name, C1.user_surname, C2.af_title, C2.af_director,
C2.af_genre, C2.af_year, C3.request_desc
FROM user as C1, absent_film as C2, request as C3
WHERE C3.user_id = C1.user_id AND C3.af_id = C2.af_id
```

Employees can delete requests by the Delete Request button, which executes the query:

```
DELETE FROM absent_film WHERE af_id = '"+af_id+"'
```

Employees are authorized to add films by entering required fields: Title, Director, Genre, and Year, and optionally non-required fields: Series, Description. If the inserted film is a part of a series, then also the relation of the inserted movie will be entered into the database system. If the description of the film is null, then a default description will be displayed. When the Add button is pressed, the following query is executed:

```
INSERT INTO film (f_title, f_director, f_year, f_rating, f_genre,
f_price, f_desc) VALUES ('"+f_title+"', '"+f_director+"',
'"+f_year+"', NULL, '"+f_genre+"', '"+f_price+"', '"+f_desc+"' )
```

If the series textbox is not null:

When a movie is added whose series does not already exist in the database, a pop-up will prompt the employee to enter a description for the series. The Following query will be executed to check if the series already exists in the database:

```
     SELECT series_name FROM series where series_name =
'“entered_series”'
```

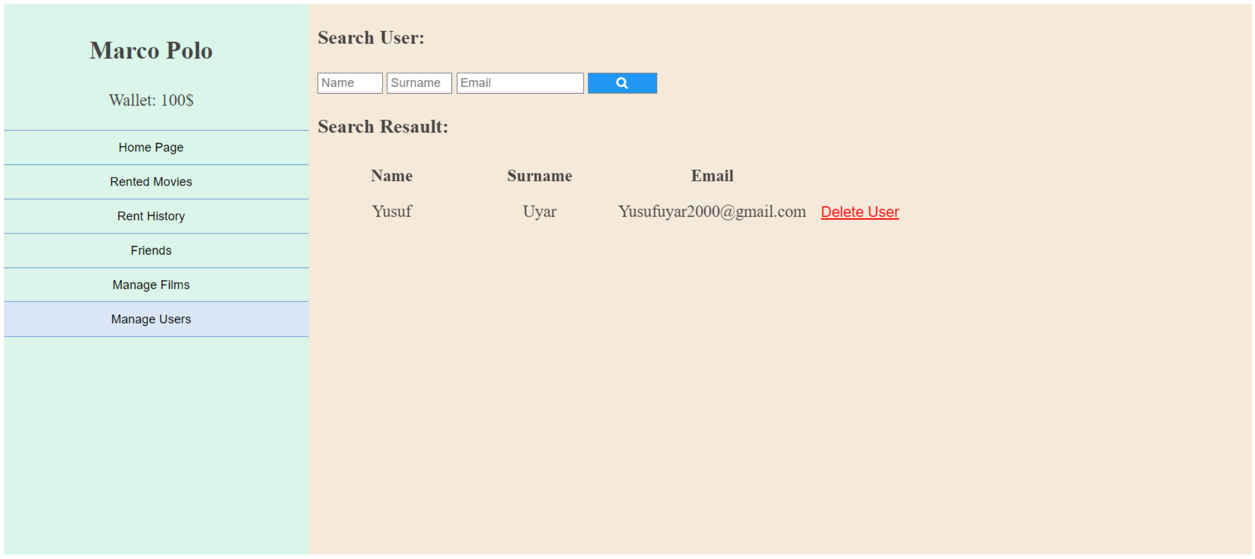
If the query returns null, the following query will be executed afterwards, if there is no series with entered name:

```
     INSERT INTO series VALUES('“series_name”', 1)
     INSERT INTO ( f_id, series_name, order_no ) VALUES (
'"+next_f_id+"', '"+series_name+"', '"+next_order_no+"' )
```

If there is a series with the entered name we will directly insert the movie in that series.

```
INSERT INTO part_of ( f_id, series_name, order_no ) VALUES (
'"+next_f_id+"', '"+series_name+"', '"+next_order_no+"' )
UPDATE series SET series_desc = series_desc + 1 WHERE series_name =
'“next_series_name”'
```

16) Manage Users



Name	Surname	Email
Yusuf	Uyar	Yusufuyar2000@gmail.com

In the Manage Users Page, an employee will be able to search for a user using name, surname or email by executing the following query:

```
SELECT user_name, user_surname, user_mail
FROM user as C1, customer as C2
WHERE ( ("+search_user_name+" IS NULL) OR (C1.user_name =
"+search_user_name+") ) AND ( ("+search_user_surname+" IS NULL) OR
(C1.user_surname = "+search_user_surname+") ) AND (
("+search_user_mail+" IS NULL) OR (C1.user_mail =
"+search_user_mail+") ) AND C1.user_id <> '"+user_id+"' AND
C1.user_id = C2.user_id
```

The results of the query will be displayed as a list. From the button to the right, the employee will be able to delete a user by the Delete User button which executes the following query:

```
DELETE FROM customer
WHERE user_id = '"+selected_user_id+"' AND user_id <> '"+user_id+"'
```

17) Additional Queries

In Addition to those queries, we might need a report query to see weekly and monthly reports for new customers.

```
SELECT count(*) as totalNewCustomersWeekly FROM customer
GROUP BY week(join_date)
SELECT count(*) as totalNewCustomersMonthly FROM customer
GROUP BY month(join_date)
```

Those queries will generate weekly and monthly reports for new customer statistics.

The views below will ease our job when we list movies,ratings, and absent movies. We don’t need to fetch the full tables.

```
CREATE View movie_ratings_view AS
SELECT f_id, f_rating, f_name
From film

CREATE View series_ratings AS
SELECT series_name, avg(f_rating)
FROM film natural join part_of
GROUP BY series_name

CREATE View absent_names AS
SELECT af_id, af_title
FROM absent_film
```

We used the following stored procedures to quickly access logged in user ID.

```
CREATE PROCEDURE login(mail, password)
BEGIN
     SELECT user_id FROM user WHERE user_mail = 'mail' AND
user_password = 'password';
END
```

We used the following stored procedures to quickly access the rater count of a film.

```
CREATE PROCEDURE rater_count(ID)
BEGIN
     SELECT count(*)as count FROM rate WHERE f_id = ID
END
```

The assertion constraint below forbids a movie to be in two different series.

```
CREATE ASSERTION movies_in_series_constraint
CHECK ( NOT EXISTS
(SELECT *
FROM part_of P1, part_of P2
WHERE P1.f_id = P2.f_id AND P1.series_name <> P2.series_name)
```

The assertion constraint below forbids two requests for the same movie.

```
CREATE ASSERTION absent_movies_constraint
CHECK ( NOT EXISTS
(SELECT *
FROM absent_film as M1, absent_film as M2
WHERE M1.title = M2.title AND M1.director = M2.director AND
     M1.year = M2.year AND M1.genre = M2.genre)
```

Use-cases

1- Customer Creates Account and Logins

1- From the index page, clicks create an account.
2- Fill out the form, click Sign in.
3- Logins to the system.

2- Customer Adds Friend

1.1- From the home page, the first user clicks on the friends tab on the left.
1.2- Clicks on add friend button.
1.3- Searches for his friend using the search bar.
1.4- Clicks on Send Request.
2.1- Second user goes to the friends tab.
2.2- Clicks on Friend Requests.
2.3- Accepts the friend request of the first user.

3- Search for a movie, rent it, rate it, and write a review, and recommend it

1- From the home page, search for the desired movie using the search bar.

2- From the list, click on the movie page corresponding to the desired film.
3- On the movie page, click on rent for: $ link to rent the movie.
4- On the movie page, write your review using the text area on the Rate section
5- Rate the film using the stars and click on the Rate button.
6- On movie page, click on recommend button
7- On popup, select the desired friends to recommend the film.
8- Click on the recommend button.

4- Customer requests a film, Employee adds it

1.1- From the home page, customer search for the desired film using the search bar.
1.2- Upon failure, the customer requests the film by filling the form.
2.1- Employee goes to Manage Films tab from the left menu.
2.2- Employee reads the request of the customer and adds the film accordingly.

5- Employee deletes a customer

1- Employee goes to Manage Users tab from the left menu.
2- Type the information of the desired customers information to the search bar.
3- Deletes the customer by clicking the delete user button

6- Customer checks rented films, rented films history, and recommended films

1- Customer clicks on the Rented Movies tab using the left menu.
2- Customer clicks on the Rent History tab using the left menu.
3- Customer clicks on the Friends tab using the left menu.
4- From the Friends tab, click on the see all recommendations button.

Implementation Plan

We decided to make a web based application for the online movie rental system. For databases, we are planning to use MySQL database management system For the software and user interface part, we are planning to use PHP.

Currently, we have prepared all of the HTML pages. For the database part, almost every necessary query is written in java and ready to use. Database will be connected to the PHP code and dummy data will be replaced with the database queries.

To make things easier, we made a work division between members. Yusuf Uyar will be responsible from the GUI aspects of the project. Ali Emre Aydoğmuş will be responsible for the database preparation part. Mustafa Çağrı Durgut and Yekta Seçkin Saraç will be responsible for the connectıon of the database to the GUI part.

Project Web Page

OnlineMovieRentalSystem.github.io