# Movie information management system

**Name/ Pitt-email:**
Ruochi Zhang/ ruz41@pitt.edu
Zhehao Guo/ zhg26@pitt.edu
Jun Jiang/ juj19@pitt.edu

## 1. Introduction

### 1.1 background

Purchasing movie ticket will be easier if customers can buy tickets online. Customers used to buy movie tickets at the theatre when they wanted to watch a movie. Therefore, people who live far away from the theatre may cost extra time and money to get the movie tickets. What's more, once customers buy the tickets but want to cancel the schedule to go to the theatre, it would be a trouble for them to return the tickets. It's inconvenient and wasteful. However, an online movie information management system can solve this problem perfectly. Not only customers can query various movie information they interest, but also they can purchase the ticket of the movie immediately.

### 1.2 overview

In this system, our purpose is to deliver a movie database with user interface(website) where user can get various movie information from the database, system can recommend movies to the user according to the tag chosen by users. Also, users can search for movies by keyword. Users can purchase movie tickets in different theatres. Users can return the ticket before a movie is showed. The system will recommend the nearest theatre to the user, if there is not available seats in that theatre, the system will suggest the users to choose other near theatres with will show that movie.

### 1.3 function description

(1) Browsing movie information.
(2) Searching movies by movie's name or keywords.
(3) User sign in, login and logout.
(4) Operators can add or remove on show movies.
(5) Recommend system.

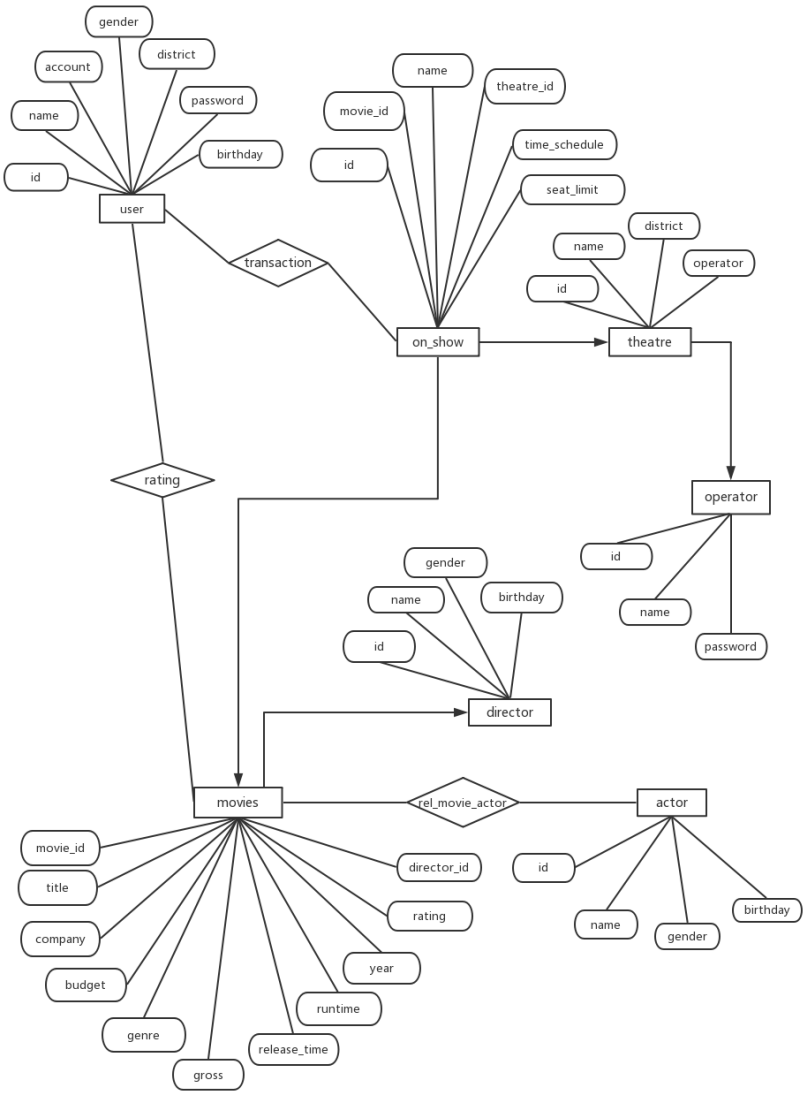
### 1.4 assumption

(1) The system service is in a city.

(2) One operator can manage several theatres, but one theatre can only have one operator.
(3) Users must sign in when using the system.

# 2. Graphical schema

## 2.1 E-R diagram




Description of entities and relationships

**Entity**

Movies: The movie entity record the detailed information of all movies in the database.

Actor:  The actor entity contents  the information of actors who appeared the movies recorded in this database. Actor entity has many to many relationship with movies entity.

Director: The director entity records the information of movie director. Director entity has one to many relationship with movies entity.

User: The user entity has seven attributes which record users' account information. User entity has many to many relationship with On_show entity. This relationship is transaction.

Operator: The operator has the permission to modify the on_show table. Operator entity has one to many relationship with theatre, which means one operator can add and remove the information of several theatres, including the movie information in those theatres.

Theatre: The theatre entity content the information of all theatres in one city. It has one to many relationship with on_show entity, which means a session of movie can only be should on one theatre, although a released movie can have many sessions.

On_show: The on_show entity record the information of each session of movie, such as when and where the movie will be showed and how many available seats in this session.

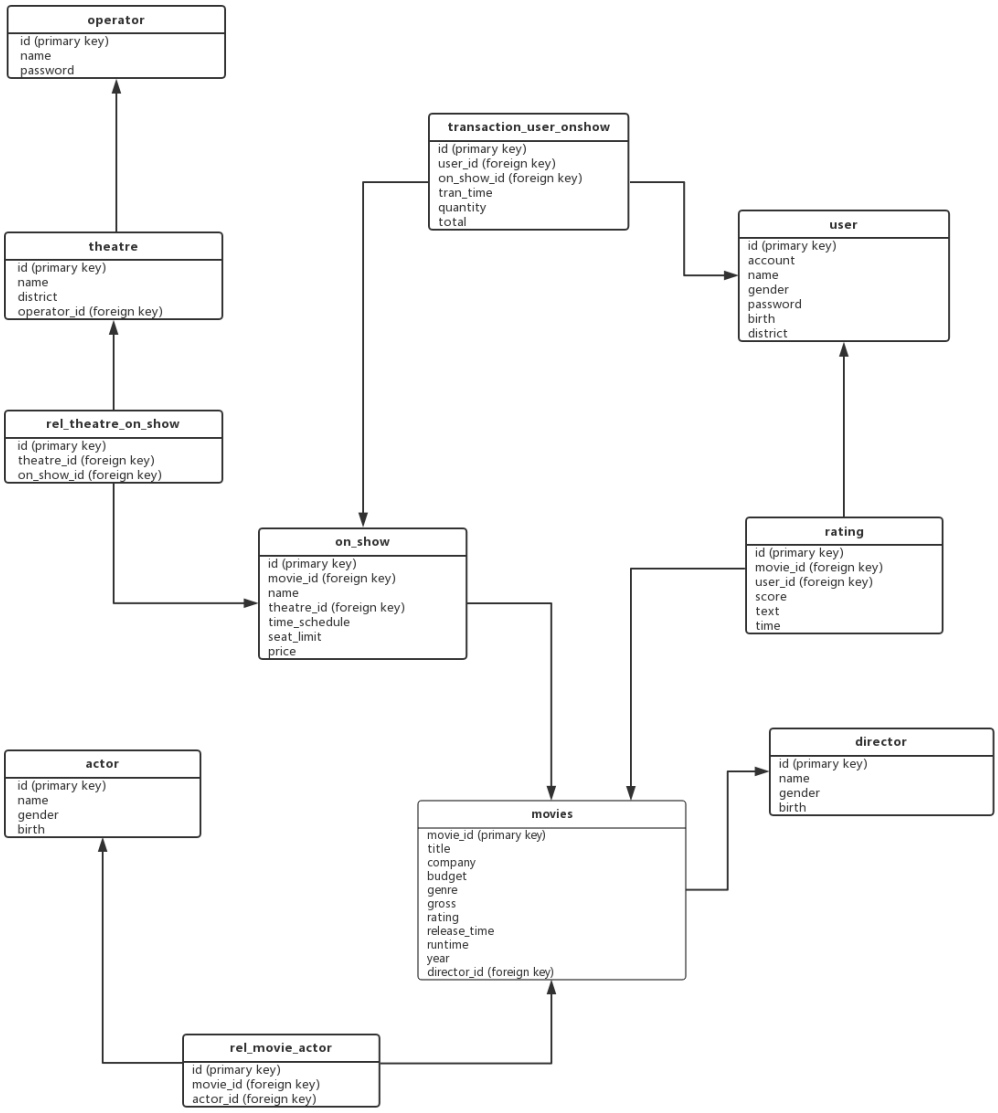
**Relationship**

Transaction: The transaction is a relationship between user table and on_show table and each row record the movie and number of tickets the user purchased.

Rel_movie_actor: The rel_movie_actor is a relationship between movies table and actor table, since an actor can appear in multiple movies and a movie is played by many actors.

Rating: The rating is a relationship between user table and movies table. The rating table record the review and rating for the movies. Each users can write several reviews and rating to movies and each movie can receive multiple reviews.

### 2.2  Relationship schema




Identifications of primary and foreign keys

movies(movie_id, title, company, budget, genre, gross, rating, release_time, runtime, year, director_id )

actor(id, name, gender, birth)

director(id, name, gender, birth)

rating(id, movie_id, user_id, score, text, time)

user(id, account, name, gender, password, birth, district)

on_show(id, movie_id, name, theatre_id, time_schedule, seat_limit, price)

rel_theatre_on_show(id, theatre_id, on_show_id)

rel_movie_actor(id, movie_id,actor_id)

theatre(id, name, district, operator_id)

transaction_user_onshow(id, user_id, on_show_id, tran_time, quantity, total)

operator(id, name, password)

**Data preparing**

The movie data comes from Kaggle. Since we cannot find complete data for the movie management database, we created the user data, transaction data, onshow data, rating data, theatre data and operator data ourselves by using python.

## 3. DDL/Normal Form

**Movies table**

```
CREATE TABLE IF NOT EXISTS `movies` (
  `movie_id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `title` varchar(255) NOT NULL,
  `company` varchar(255) NOT NULL,
  `budget` float NOT NULL,
  `genre` varchar(255) NOT NULL,
  `gross` float NOT NULL,
  `rating` varchar(255) NOT NULL,
  `release_time` date NOT NULL,
  `runtime` int(11) NOT NULL,
  `year` year(4) NOT NULL,
  `director_id` int(11) NOT NULL,
   FOREIGN KEY (`director_id`)
   REFERENCES `director` (`id`)
   ON UPDATE CASCADE
   ON DELETE CASCADE
);
```

**On_show Table**

```
CREATE TABLE IF NOT EXISTS `on_show` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `movie_id` int(11) NOT NULL,
  `name` varchar(255) NOT NULL,
  `theatre` varchar(255) NOT NULL,
  `time_schedule` datetime NOT NULL,
  `seat_limit` int(3) NOT NULL,
   FOREIGN KEY (`movie_id`)
   REFERENCES `movies` (`movie_id`)
   ON UPDATE CASCADE
   ON DELETE CASCADE
);
```

**Rating Table**

```
CREATE TABLE IF NOT EXISTS `rating` (
  `movie_id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  `score` float NOT NULL,
  `text` varchar(255) DEFAULT NULL,
  `time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
FOREIGN KEY (`movie_id`)
REFERENCES `movies` (`movie_id`)
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (`user_id`)
REFERENCES `user` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

**Director Table**

```
CREATE TABLE IF NOT EXISTS `director` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `gender` varchar(255) NOT NULL,
  `birth` datetime NOT NULL
);
```

**User Table**

```
CREATE TABLE IF NOT EXISTS `user` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `gender` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `birth` datetime NOT NULL
);
```

**Actor Table**

```
CREATE TABLE IF NOT EXISTS `actor` (
```

```
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` varchar(200) NOT NULL,
  `gender` varchar(200) NOT NULL,
  `birth` datetime DEFAULT NULL
);
```

**Theatre Table**

```
CREATE TABLE IF NOT EXISTS `theatre` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `district` varchar(255) NOT NULL,
  `operator_id` int(11) NOT NULL,
   FOREIGN KEY (`operator_id`)
   REFERENCES `operator` (`id`)
   ON UPDATE CASCADE
   ON DELETE CASCADE
);
```

Operator Table

```
CREATE TABLE IF NOT EXISTS `operator` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL
);
```

**Rel_theatre_on_show Table**

```
CREATE TABLE IF NOT EXISTS `rel_theatre_on_show`(
`id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
`theatre_id` int(11) NOT NULL,
`on_show_id` int(11) NOT NULL,
FOREIGN KEY (`theatre_id`)
REFERENCES `theatre` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (`on_show_id`)
REFERENCES `on_show` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE

);
```

**Rel_movie_actor Table**

```
CREATE TABLE IF NOT EXISTS `rel_movie_actor` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `movie_id` int(11) NOT NULL,
  `actor_id` int(11) NOT NULL,
FOREIGN KEY (`movie_id`)
REFERENCES `movies` (`movie_id`)
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (`actor_id`)
REFERENCES `actor` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE

);
```

**Transaction_user_onshow Table**

```
CREATE TABLE IF NOT EXISTS `transaction_user_onshow` (
  `id` int(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `user_id` int(11) NOT NULL,
  `on_show_id` int(11) NOT NULL,
  `tran_time` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
FOREIGN KEY (`user_id`)
REFERENCES `user` (`user_id`)
ON DELETE CASCADE
ON UPDATE CASCADE,
FOREIGN KEY (`on_show_id`)
REFERENCES `on_show` (`id`)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

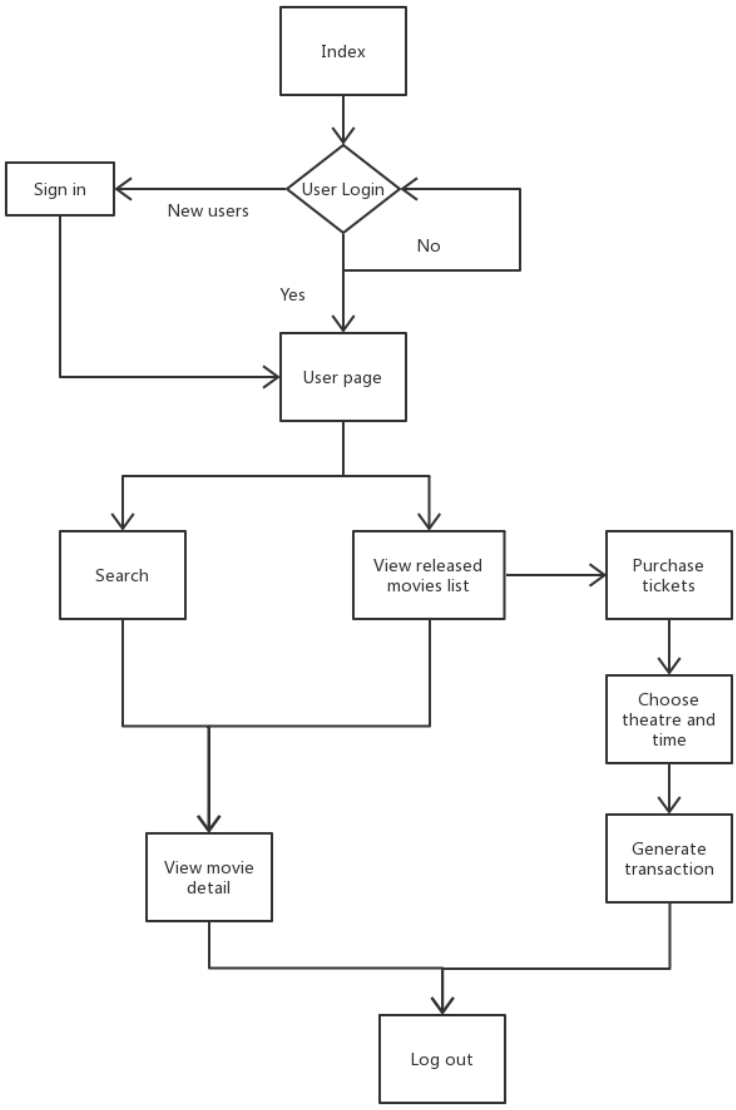
## 4. Front and back end Design

### 4.1 Front-end

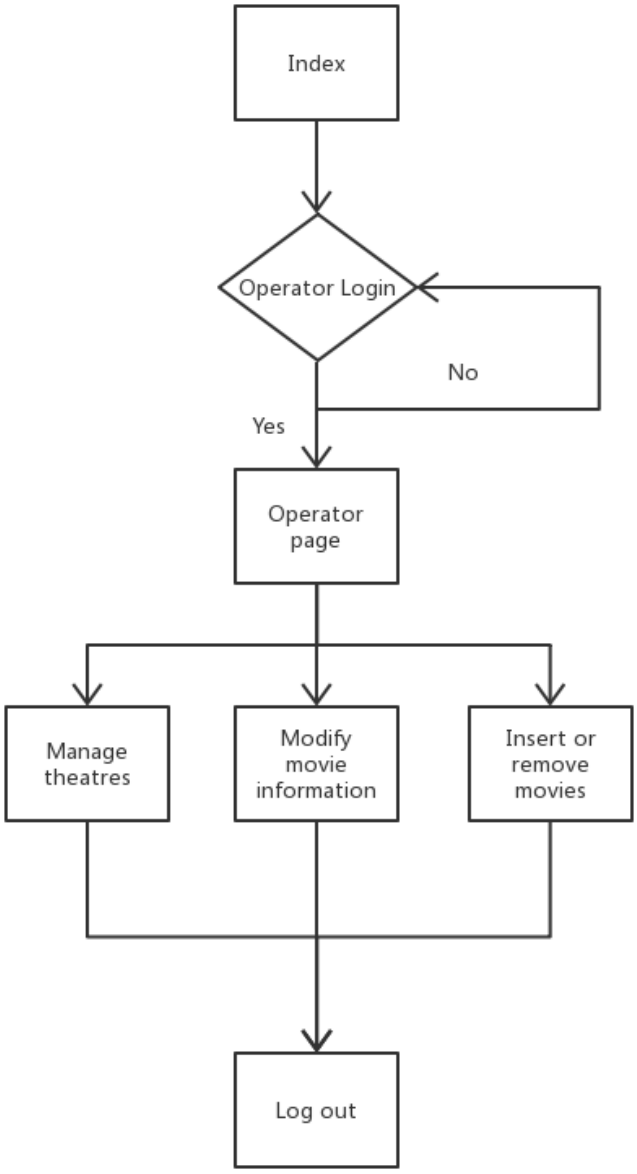
Our website contains 14 web pages. The webpages are written by html. We designed the structure of the website and the skipping logic between webpage as follow. In the html files, we define several parameters to get the results returned by the back-end.

### 4.1.1 Flow chart of front-end webpage

**User**




**Operator**




**4.1.2 Introduction of webpage**

(1)Index.html

This webpage is the first page presenting to users when they enter the movie information management system.

(2)Login.html

Click login button on the index page, the webpage will jump to login page. By inputting username and password, users can login to the user page. The system will check if the username and the password are correct.. If the username or the password is not valid,

the system will return a error message and ask the user to input again or register a new account. Operators also use this page to login.

(3)Movies.html
On this page, user can select movies by click movie tags. These movie tags classify movies into different types, and users can choose one tag to get movies they may be interested in. There is a search bar. Users can input the title of the movie to check its details. If there are more than one movies share the same name, the movie detail page will print out all movies information. The system will jump to movie detail page to present the movie's information. In each row of the result table, the title is refer to a hyperlink. When users click on the title, the webpage will skip to the moviedetail.html. Users can see the details of the movie.

(4)Moviedetail.html
This page displays the detail information of a movie, including title, company, budget , genre, gross, rating, release time, runtime, year and director. Besides, this page also displays the actors, director and users' reviews and rating. Each actor and director of the movie refer to a hyperlink. Users can click on actor's or director's  name to view the detail of that person.

(5)Movietag.html
When users click on one tag on movie page, the system will select out the movies which meet the description of the tag.

(6)Onshow.html
This page display the sessions of released movies. There is a table on the webpage, and the table contains title, schedule, seat, price, theatre and district. The title in the table refer to a hyperlink. Users can click on the title and skip to the transaction page, trans.html.

(7)Register.html
Register page contains a form. User can input their personal information and become a new user in the system. The same email address is not allowed to be use twice. Thus, if the user's email already exists in the system, the system will return an error message when the user sign in.

(8)User.html
The user page recommend movies to the user.

(9)Operator.html

Operators can login through the login page, and delete the movies from the theatres they manage. Each theatre is display as a button on this page. Operator can click on a button and the system will skip to the management page of this theatre.

(10)Management.html
Management page display the movies on show in a specific theatre. Only operator can see this page. Operators can add and delete the movie information.

(11)Profile.html
Profile page is used to present the user's account information. Users can update their personal information except the account name through this page.

(12)Trans.html
Transaction page will print out the order history of the user. It will provide the order number, username, title of the movie, the movie start time, the quantity of tickets and the total price of the purchasing.

**4.2 Back-end**
For designing back-end, we chose the web.py framework. Web.py is a web framework for python that is as simple as it is powerful. We write the python file as the back-end script. All functions mentioned above can be integrated into one script file. By import pymysql, we can connect Mysql database by python. Then, we use the query function of the pymysql package to execute the sql clause.

**Functions implementation and SQL clauses**

(1)Login
Get the user's email address and password. Passing these two value the Login in class in the movieapp.py. This is the back-end python script. Using SQL to find if the user exists in database. Since we have normal user and operator, we should distinguish them at first.

Pseudocode :

```
If check_user
   return user;
else if check_operator
   return operator(theatre);
else
   return login(error);
```

SQL:

```
Check_user =  db.query('SELECT * FROM user WHERE account = $email AND
password = $passwd', var={email : 'email', password : 'passwd'});

Check_operator = db. query('SELECT * FROM operator WHERE account=$email
AND password =$passwd', var ={email: 'email, password: 'passwd''});

Theaters = db.query("SELECT DISTINCT(name) FROM theater WHERE operator =
{}".format(session.id))
```

(2)Management
Return the theatre list that control by a specific operator.
SQL:

```
Movies = db.query('
SELECT m.title, o.time_schedule, o.seat_limit, o.id, o.price, t.name, t.district,
t.operator
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON
t.id = o.thea_id
WHERE t.operator = $operator and t.name = $name ORDER BY m.movie_id,
o.time_schedule',vars = {"operator":session.id, "name":name})
```

(3)Delete
Operator delete a movie on show by the movie id.
SQL:

```
db.query("DELETE FROM on_show WHERE id = {}".format(id))
```

(4)Register
Pseudocode:

```
if check
        return register(True);
    else
        register new user;
        if res
            return index();
        else
            return "error";
```

“Check” is a SELECT operation which is to verify whether the user already exists. If SELECT returns one or more rows, “check” will equals TRUE.  If there is not the user information, we execute “res”. “Res” is a INSERT operation that insert the user’s input into the user table and save it as a new user.
SQL:

```
Check = db.query('SELECT * FROM user WHERE account = $account and password = $password;', vars = {'account':account, 'password': password});

res = db.insert(‘user’, name = name, gender = gender, password = password, birth = birth, account = account, district = district)
```

(5)Result
Result is used to return a message to tell user if the operation is success.

```
return render.result(True)
```

(6)Onshow
List all on show movies in the on_show table.
SQL:

```
res = db.query('
SELECT m.title,o.time_schedule,o.seat_limit, o.id,o.price,t.name,t.district
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON t.id = o.thea_id
ORDER BY m.movie_id, o.time_schedule')
```

Users can also search for the on show movies by entering title or theater or district keywords.
If the input is title keyword

SQL:

```
res = db.query(‘
SELECT m.title,o.time_schedule,o.seat_limit, o.id,o.price,t.name,t.district
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON
t.id = o.thea_id
WHERE m.title = $title ORDER BY m.movie_id, o.time_schedule’,vars = {"title":title})
```

If the input is theater keyword
SQL:

```
res = db.query(‘
SELECT m.title,o.time_schedule,o.seat_limit, o.id,o.price,t.name,t.district
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON
t.id = o.thea_id
WHERE t.name = $theater ORDER BY m.movie_id, o.time_schedule’,vars =
{"theater":theater})
```

If the input is the district keyword
SQL:

```
res = db.query(‘
SELECT m.title,o.time_schedule,o.seat_limit, o.id,o.price,t.name,t.district
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON
t.id = o.thea_id
WHERE t.district = $district ORDER BY m.movie_id, o.time_schedule’,vars =
{"district": district})
```

(7)Movies
List all the movies in movies table on separating pages. We limit 20 rows on each page. The method of separating page is written as:

```
if not page:
        page = 1

    NavNum = 20
    results = db.query("SELECT COUNT(*) AS c FROM movies")
    count = results[0].c

    if count % NavNum==0:
        pages = count // NavNum
    else:
        pages = count // (NavNum + 1)

    off = (int(page)-1) * NavNum
```

```
    got_movies = db.select('movies',order='movie_id',limit = NavNum,offset = off)

    if got_movies:
        return render.movies(got_movies,int(page))
    else:
        return "Can not find any movie."
```

Select out all movies
SQL:

```
got_movies = db.select('movies',order='movie_id',limit = NavNum,offset = off)
```

Search a movie by title keyword, select out its basic information, the details of actors and director and users' reviews.
SQL:

```
 movies = db.query('SELECT * FROM movies WHERE title = $title',vars =
{"title":title});

 actors = db.query('
SELECT DISTINCT(a.name)
FROM movies m JOIN rel_movie_actor r ON m.movie_id = r.movie_id JOIN actor a
ON r.actor_id = a.id
WHERE title = $title;',vars = {"title": title});

directors = db.query('
SELECT DISTINCT(d.name)
FROM director d JOIN movies ON movies.director_id = d.id
WHERE title = $title;',vars = {"title": title});

ratings = db.query('
SELECT r.score,r.text
FROM rating r JOIN movies m ON  m.movie_id = r.movie_id
WHERE m.title = $title',vars = {"title": title});
```

Forward the results of query to moviedetail.html and present on the front page.

```
    if actors or directors or ratings or movies:
        return render.moviedetail(movies,actors,directors,ratings)
    else:
```

```
        return "no movies"
```

(8)Movietag
Select movies by genre
SQL:

```
got_movies = db.select('movies',order='movie_id', where="genre = $tag",vars = {"tag":tag})
```

Select movies by rating
SQL:

```
got_movies = db.query('SELECT * FROM movies WHERE rating = $tag ORDER BY movie_id',vars = {"tag":tag})
```

Select movies by year
We divide 1990 to 2018 into 8 time periods, and each time period is a tag on the movietag.html.
year tag = {After 2018, 2015-2018, 2010-2015,  2005-2010, 2000-2005,  1995-2000 , 1990-1995,  Before 1990}
SQL:

```
tmp = tag.split("-")
        if len(tmp) == 1 and tmp[0].startswith("After"):
            got_movies = db.query('SELECT * FROM movies WHERE year >= $year ORDER BY movie_id',vars = {"year":tmp[0].lstrip("After")})
        elif len(tmp) == 1 and tmp[0].startswith("Before"):
            got_movies = db.query('SELECT * FROM movies WHERE year < $year ORDER BY movie_id',vars = {"year":tmp[0].lstrip("Before")})
        elif len(tmp) == 2:
            got_movies = db.query('SELECT * FROM movies WHERE year BETWEEN $year_left AND  $year_right ORDER BY movie_id',vars = {"year_left":tmp[0],"year_right": tmp[1]})
```

(9)Actor
Select all information of an actor
SQL:

```
res = db.query('SELECT * FROM actor WHERE name = $name',vars = {"name":name})[0]
```

(10)Director
Select all information of a director
SQL:

```
res = db.query('SELECT * from director WHERE name = $name',vars =
{"name":name})[0]
```

(11)Logout
Response to the logout button.

```
def GET(self):
        if not session.logged_in:
            return render.login()
        else:
            session.logged_in = False
            session.user = None
            raise web.seeother("/")
```

(13)Moviedetail
List the actors, director, users' rating and other information of a specific movie on one page. The movie is selected by the movie title.
SQL:

```
actors = db.query('
SELECT DISTINCT(a.name)
FROM movies m JOIN rel_movie_actor r ON m.movie_id = r.movie_id JOIN actor a
ON r.actor_id = a.id
WHERE title = $title;',vars = {"title": title});

directors = db.query('
SELECT DISTINCT(d.name)
FROM director d JOIN movies ON movies.director_id = d.id
WHERE title = $title;',vars = {"title": title});

ratings = db.query('
SELECT r.score,r.text,r.user_id
FROM rating r JOIN movies m ON  m.movie_id = r.movie_id
WHERE m.title = $title',vars = {"title": title});

movie = db.query('SELECT * FROM movies WHERE title = $title',vars = {"title":title});
```

Insert  new comments into rating table and record the time.
SQL:

```
 movie_id = db.query('S
ELECT movie_id
FROM movies
WHERE title = $title',vars = {"title":session.title})[0].movie_id)

res = db.insert("rating", user_id = user_id, movie_id = movie_id, score = score, text =
comment, time = web.SQLLiteral("NOW()"))
```

(14)Transaction
List the order history of a user by selecting out the transaction of a user.
SQL:

```
res = db.query(‘
SELECT t.id,m.title,t.tran_time,t.quantity,t.total_price
FROM transaction_user_onshow t  JOIN on_show o ON t.on_show_id = o.id JOIN
movies m ON m.movie_id = o.movie_id
WHERE t.user_id = {}’.format(session.id))
```

(15)Order
List the detail information of the movie which the user is going to purchase tickets for.
SQL:

```
res = db.query('
SELECT m.title,o.time_schedule,o.seat_limit, o.id,o.price,t.name,t.district
FROM movies m JOIN on_show o ON m.movie_id = o.movie_id JOIN theater t ON
t.id = o.thea_id
WHERE o.id = $on_show_id ORDER BY m.movie_id, o.time_schedule', vars =
{"on_show_id": on_show_id})[0]
```

Pseudocode:

```
def int purchase_num
        if seat_limit >=purchase_num{
             seat_limit = seat_limit - purchase_num
             add new transaction into transaction table
             return result(True)
        }
        else
            return result(False)
```

If seat quantity is larger than the purchasing quantity, minus purchasing quantity from seat quantity and insert new transaction into the transaction table.
SQL:

```
db.query('UPDATE on_show o SET o.seat_limit = o.seat_limit - $num WHERE o.id = $on_show_id',vars = {"num":num,"on_show_id":on_show_id});

db.query('INSERT INTO transaction_user_onshow (user_id, on_show_id, quantity, total_price) VALUES ({},{},{},{})'.format(session.id, on_show_id, num, num*res.price));
```

(16)Profile
List the user information by user id
SQL:

```
user = db.query('SELECT * FROM user u WHERE u.id = {}'.format(session.id))[0]
```

Update user information. Get user's input from the front-end.
SQL:

```
res = db.query('UPDATE user SET password = \"{}\", birth = \"{}\", district = {}, name = \"{}\", gender = \"{}\"'.format(password,birth,district,name,gender))
```

(17)From validation:
Update user information. Get user's input from the front-end.
SQL:

```
def isVaildDate(date):
    try:
        if ":" in date:
            time.strptime(date, "%Y-%m-%d %H:%M:%S")
        else:
            time.strptime(date, "%Y-%m-%d")
        return True
    except:
        return False

def isVlidScore(number):
    if number.isnumeric()  and  int(number) >= 0 and int(number) <= 10:
        return True
    return False
```

```
def isVlidDistric(number):
    if number.isnumeric() and str(int(number)) == number and int(number) >= 0 and
int(number) < 10:
        return True
    return False


def isVlidGender(gender):
    if gender.lower() in ("male","female"):
        return True
    return False


def isVlidSeatAndPriceaAndTheater(number):
    if number.isnumeric() and str(int(number)) == number and number >= "0":
        return True
    return False
```

## 5. Front-end to back-end connection Design

In the python script, we write multiple classes to execute the requests from the front-end In this classes, we define GET methods and POST methods to transfer parameters from front-end to back-end. These methods also can present the query result on the front pages. Our connection design is introduced in detail as follow.

(1) Import necessary python packages and we will use the web.py integrated functions. Connect database. Session is a global variable to store the parameters. Render can forward the parameter to other pages.

```
import web
import pymysql
from web import form
from web.contrib.template import render_jinja

web.config.debug = False
pymysql.install_as_MySQLdb()
db = web.database(dbn='mysql', user='root', pw='lv23623600', db='movie_infor')
app = web.application(urls, globals())
session = web.session.Session(app, web.session.DiskStore('sessions'),initializer={'logged_in': 0, 'user':None})
render = web.template.render('templates/',globals={'context':session})
```

(2) We create the url set. Url set contains the url pairs which connect the front pages and their responsible function class in the python script. Some of the urls contain “(.*)” means this url can carry a parameter of string type.

```
urls = (
    '/','Index',
    '/index', 'Index',
    '/login','Login',
    '/register','Register',
    '/logout','Logout',
    '/onshow','Onshow',
    '/user',"User",
    '/movies(.*)',"Movies",
    '/moviedetail(.*)',"MovieDetail",
    '/movietag(.*)',"MovieTag",
    '/management(.*)',"Management",
    '/actor(.*)',"Actor",
    '/result',"Result",
    '/order/(.*)/(.*)',"Order",
    '/trans', "Transaction",
    '/profile',"Profile",
    '/delete(.*)',"Delete",
    '/director(.*)',"Director",
    '/statistic',"Statistic"

)
```

(3) POST method

For example, in the class Login, the POST method get the user's input and from the front page , the front page is design as follow. When user click on submit button, their input will be forward to back end. The POST method get "email" and "password" by the name of parameters. After checking the identity of user, POST method will "return render.user()" or "return render.operator(theatre)". That means, the website will forward to user's home page or the operator management page.If the account not exists, method will "return render.login(error)", which is an error page.

Login.html code:

```
  <form class="form-signin" method=POST>
   $if error:
      <div class="alert">
        <span class="closebtn" onclick="this.parentElement.style.display='none';">&times;</span>
        Please Login Again.
      </div>
  <h1 class="h3 mb-3 font-weight-normal" style="color:white">Sign In</h1>
  <label for="inputEmail" class="sr-only" >Email address</label>
  <input type="email" name="email" class="form-control" placeholder="Email address" required autofocus>
  <label for="inputPassword" class="sr-only">Password</label>
  <input type="password" name="passwd" class="form-control" placeholder="Password" required>
  <div class="checkbox mb-3" style="color:white">
    <label>
      <input type="checkbox" value="remember-me"> Remember me
    </label>
  </div>
  <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
</form>
```

POST method code in Login class:

```
def POST(self):
    raw_data = web.input()
    email = raw_data.get('email')
    passwd = raw_data.get('passwd')
    check_user = db.query('select * from user where account = $email and password = $passwd', vars = {'email':email, 'passwd': passwd})

    check_operator = db.query('select * from operator where account = $email and password = $passwd', vars = {'email':email, 'passwd': passwd})


    if check_user:
        error = False
        session.logged_in = 2
        session.user = email
        session.id = check_user[0].id
        return render.user()
    elif check_operator:
        error = False
        session.logged_in = 1
        session.user = email
        session.id = check_operator[0].id
        theaters = db.query("SELECT DISTINCT(name) FROM theater WHERE operator = {}".format(session.id))

        return render.operator(theaters)
    else:
        error = True
        return render.login(error)
```

(4)GET method

GET method will transfer the query result to the front page. When input the url, GET method will make effect.

GET method code in Login class:

```
class Login:
    def GET(self):
        if session.user and session.logged_in == 2:
            return render.user()
        elif session.user and session.logged_in == 1:
            theaters = db.query("SELECT DISTINCT(name) FROM theater WHERE operator = {}".format(session.id))
            return render.operator(theaters)

        error = False
        return render.login(error)
```

(5) Parameters in html

Html file can accept the parameter passed by back-end.

```
$def with (movies,actors,directors,ratings,amb)
```

## 6. Recommend function design

We use NEO4J to implement our recommend function. NEO4J is a native graph database platform specifically optimized to map, store, and traverse networks of highly connected data to reveal invisible contexts and hidden relationships. By intuitively analyzing data points and the connections between them, Neo4j powers intelligent, real-time applications that tackle today's toughest enterprise challenges.

In order to select the movies that user may interest in, we suppose to find out the relation between movie and user. We choose rating given by user as the relation between movies. Analysis the correlation between movies, we can group the the movies which have high correlation into a community. According to the movies that one user

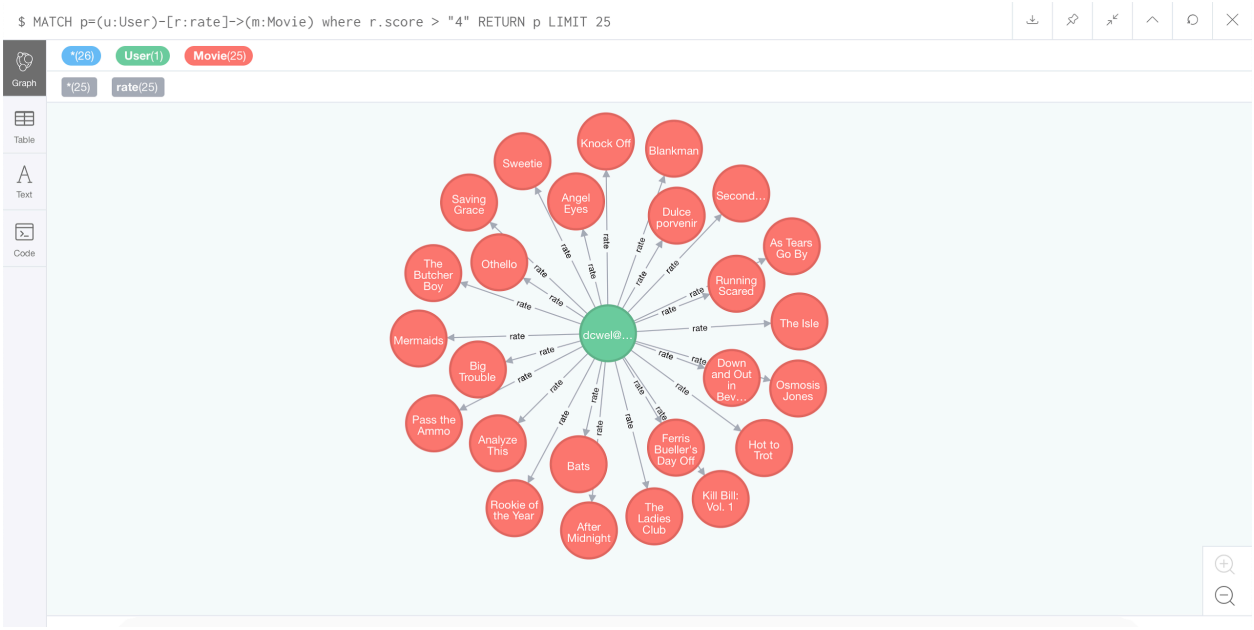
has given reviews. We can find out which movie that he or she may prefer. From the community of this movie, we choose three movies randomly from the community and recommend them to users.

Therefore, our recommendation algorithm is roughly as follows:

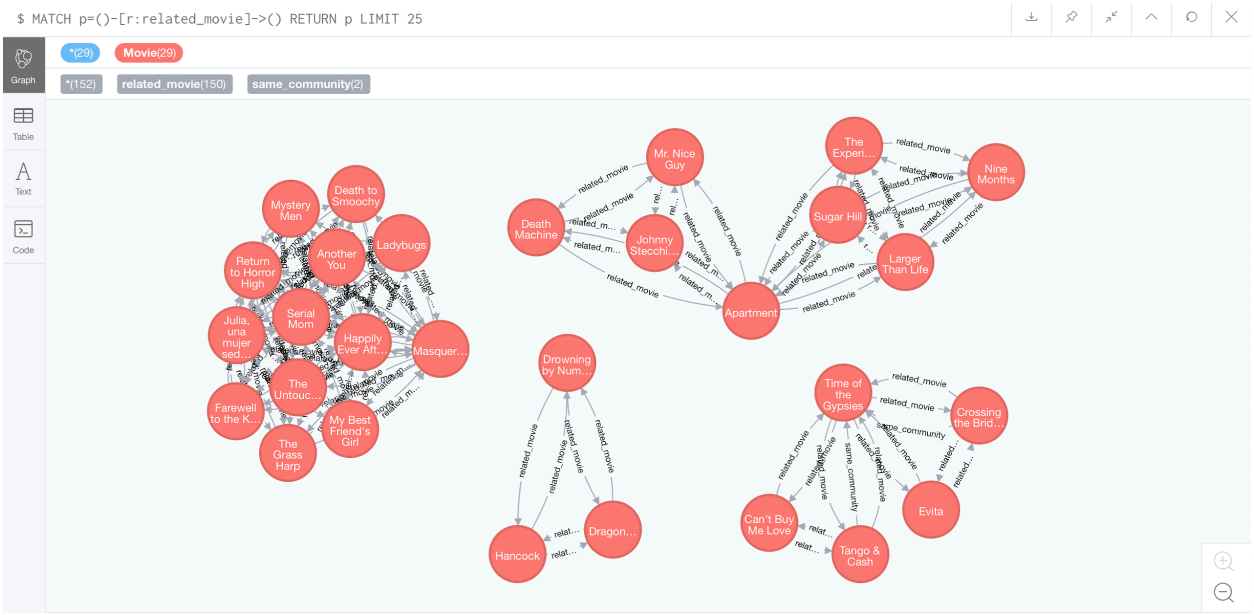
1. We create two kinds of entities in neo4j, which are User and Movie
2. We create the  “Rate” relationship which show the relation between a certain user and a certain movie if a user have rated this movie.
3. We use the item-based collaborative filtering algorithm to calculate the similarity between movies. If the similarity score > 0.7, we add a “Related_movie” relationship to these movies.
4. We use the Louvain algorithm to detect some communities among movies according to their “related_movie” relationships. The movies which are in the same community means that they all have similar feature.
5. Finally, if a user rates a certain movie, we can recommend the user other movies in the same community.

Here are some screenshots in neo4j:

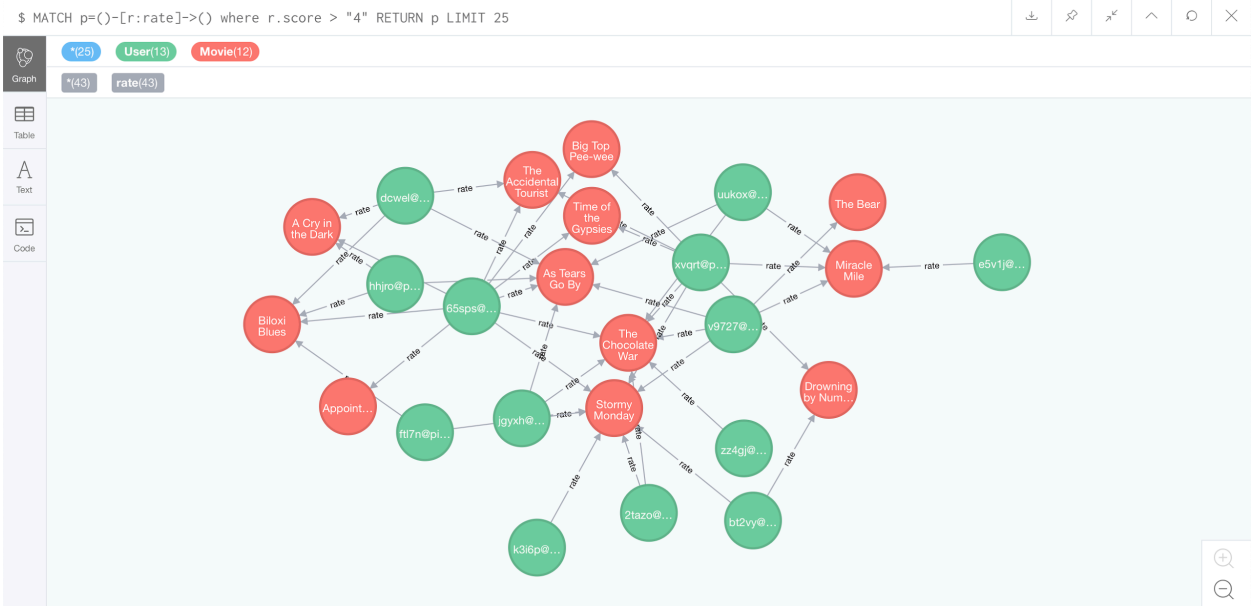
1. The graph shows that part of movies are rated as more than “4” score by a certain user:



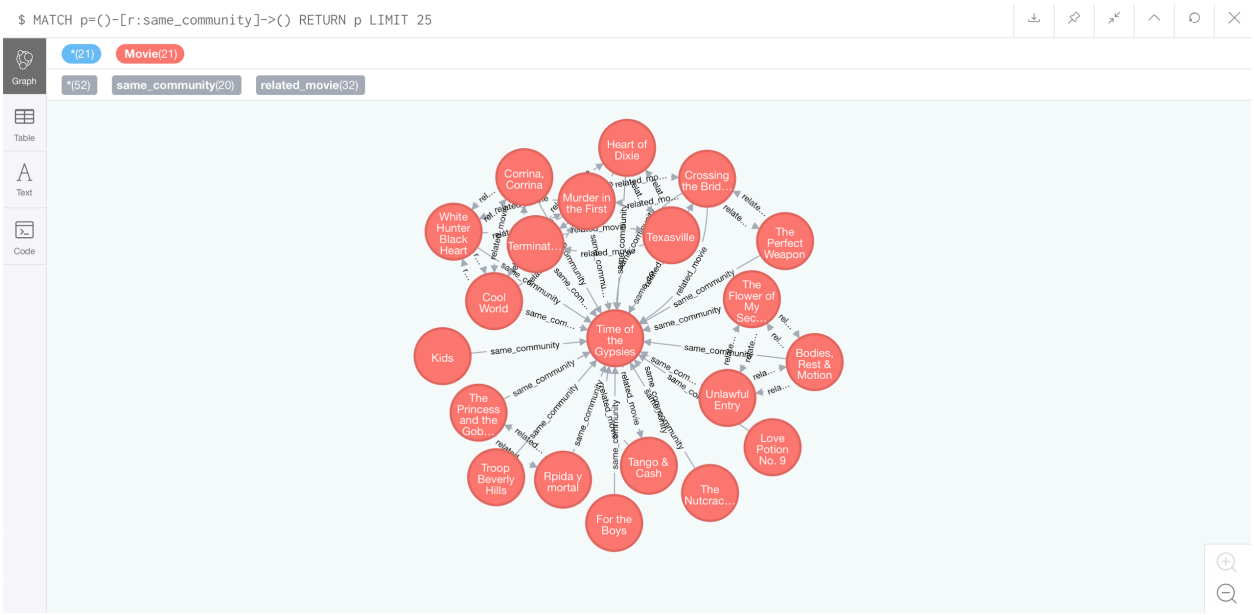
2. The graph shows that part of movies are in the same “related_movie” relationship:



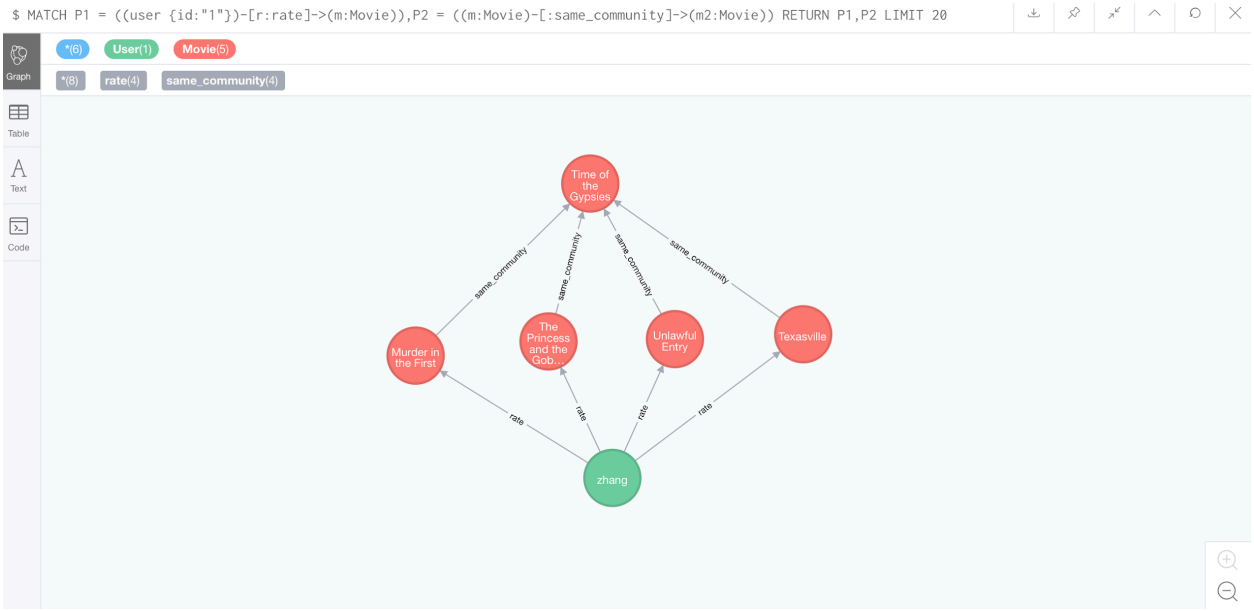

3. The graph partly shows that relation between users and movies and the ratings are more than 4:



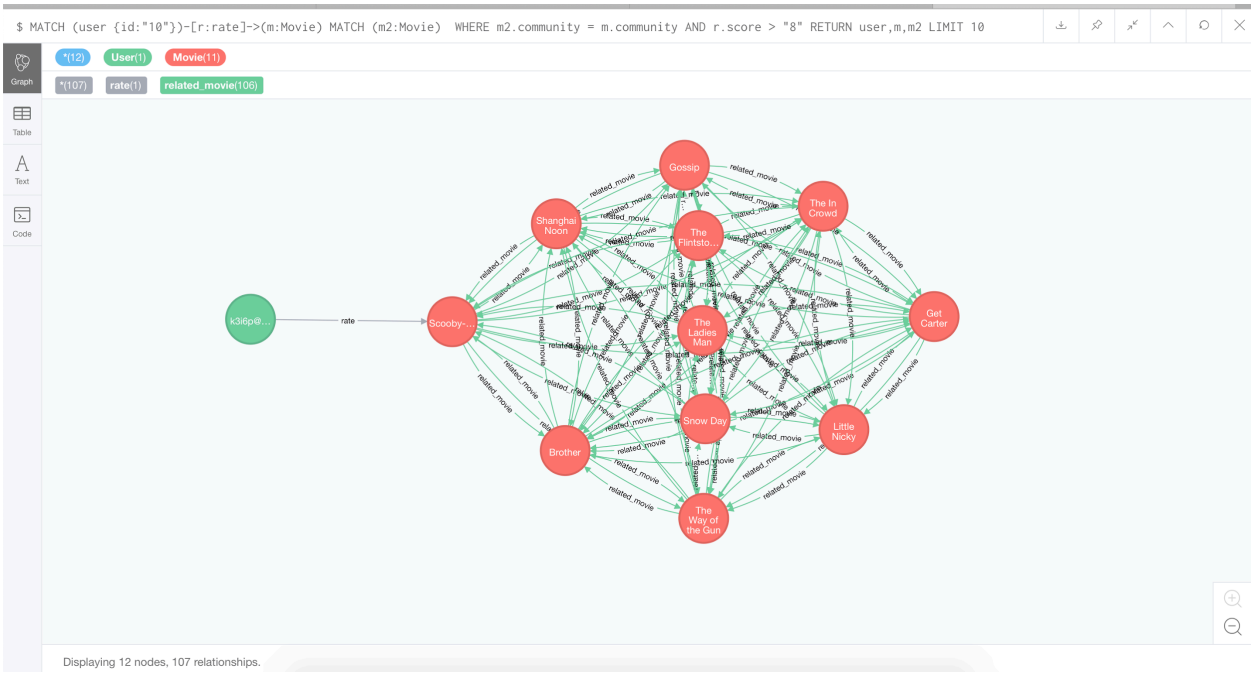
4. The graph shows a community which has lots of same movies:




5. The graph shows that the user whose id is 1 is recommended for the movie named “Time of the Gypsies” and the movies he rated.



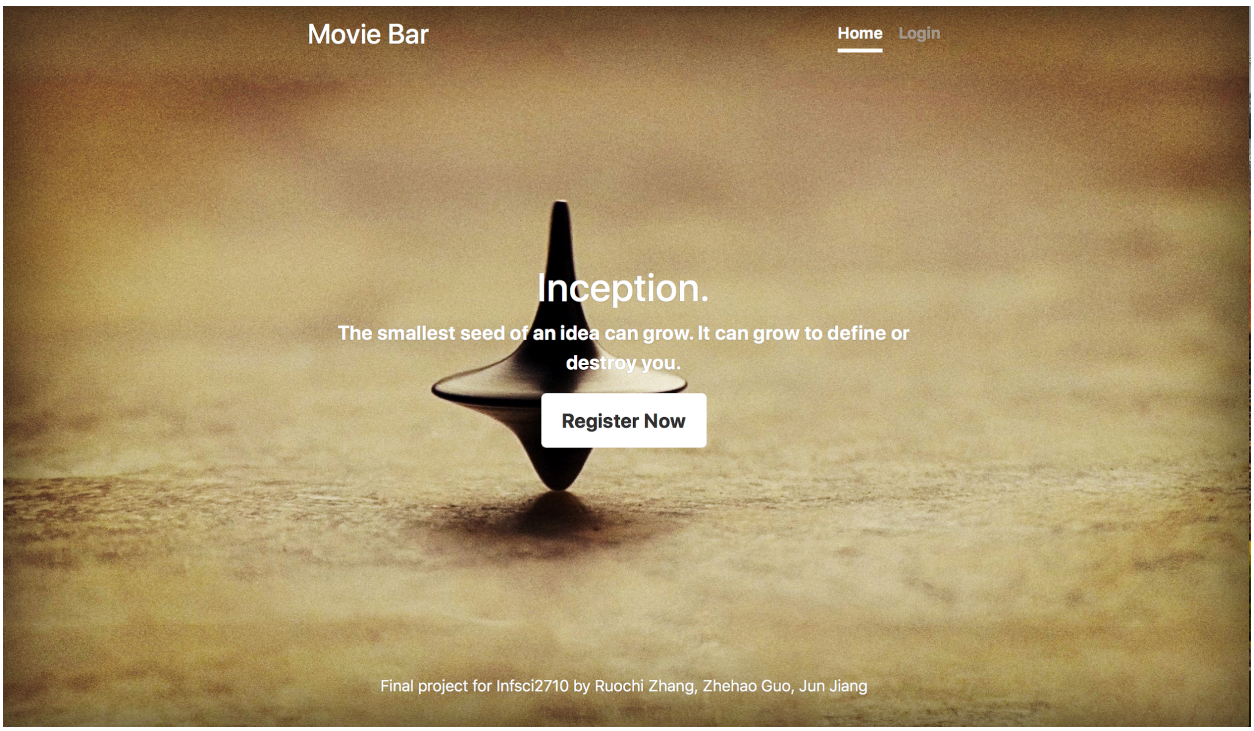
6. The graph shows that the user whose id is 10 is recommended by the system for several movies due to a movie he rates:



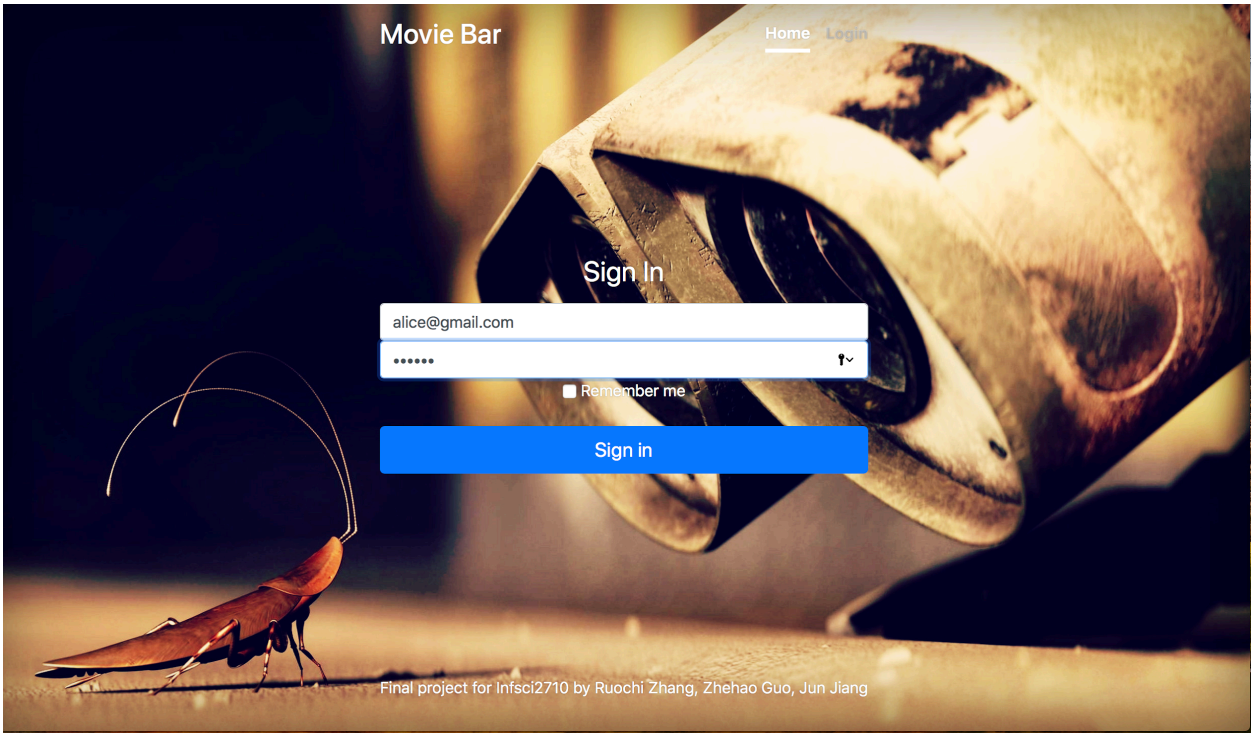

We implement this recommend function in Statistic class.

## 7. System implementation and screenshots

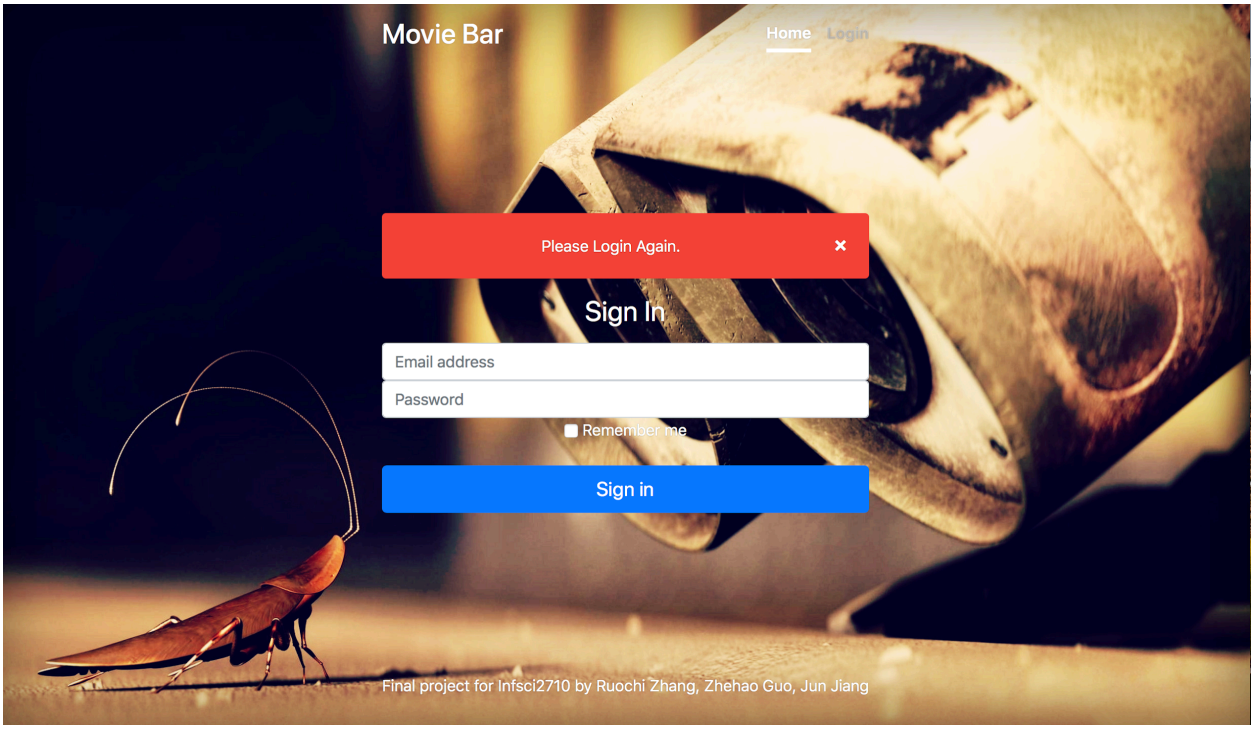
(1)Index



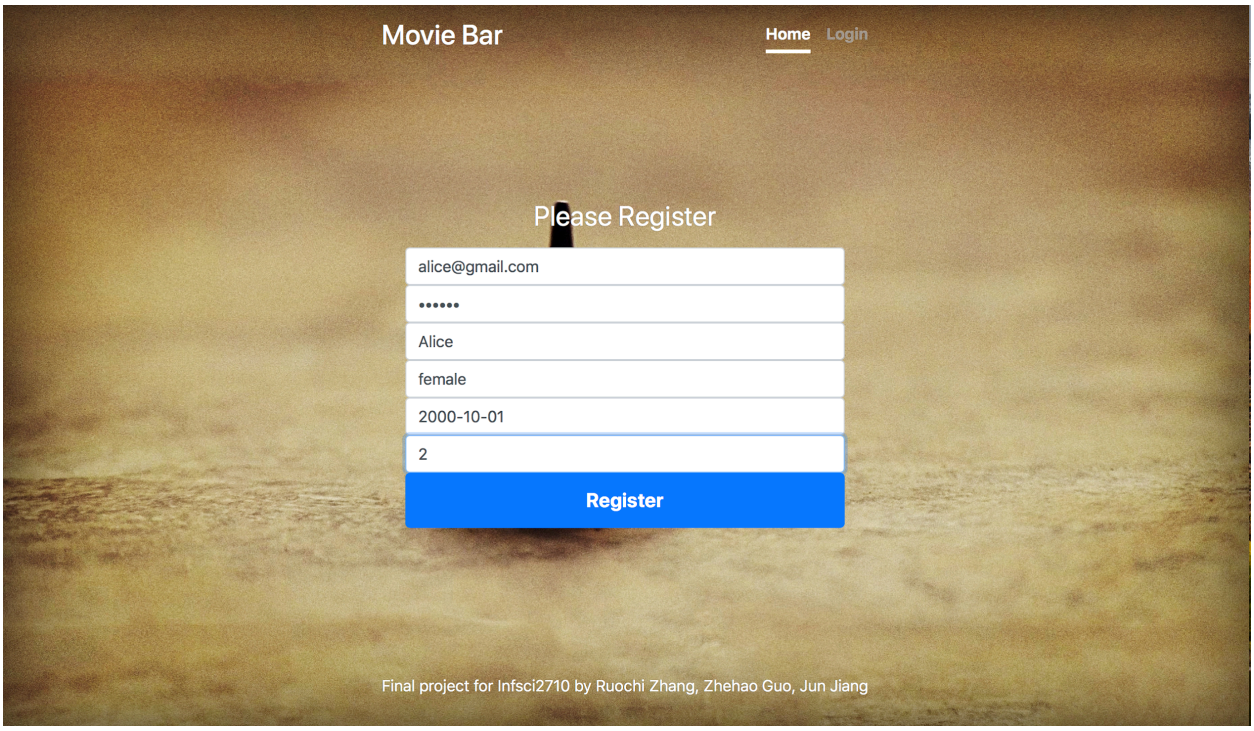
(2)Login



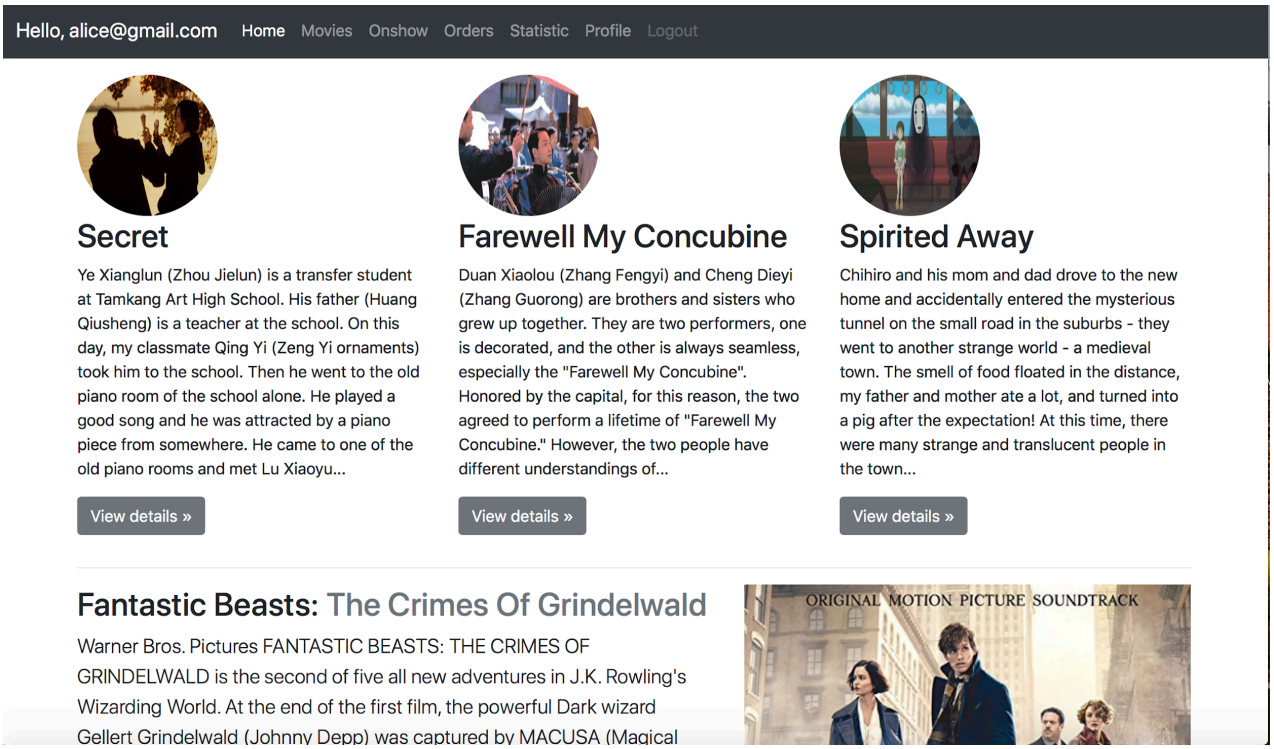

(3)Login fail



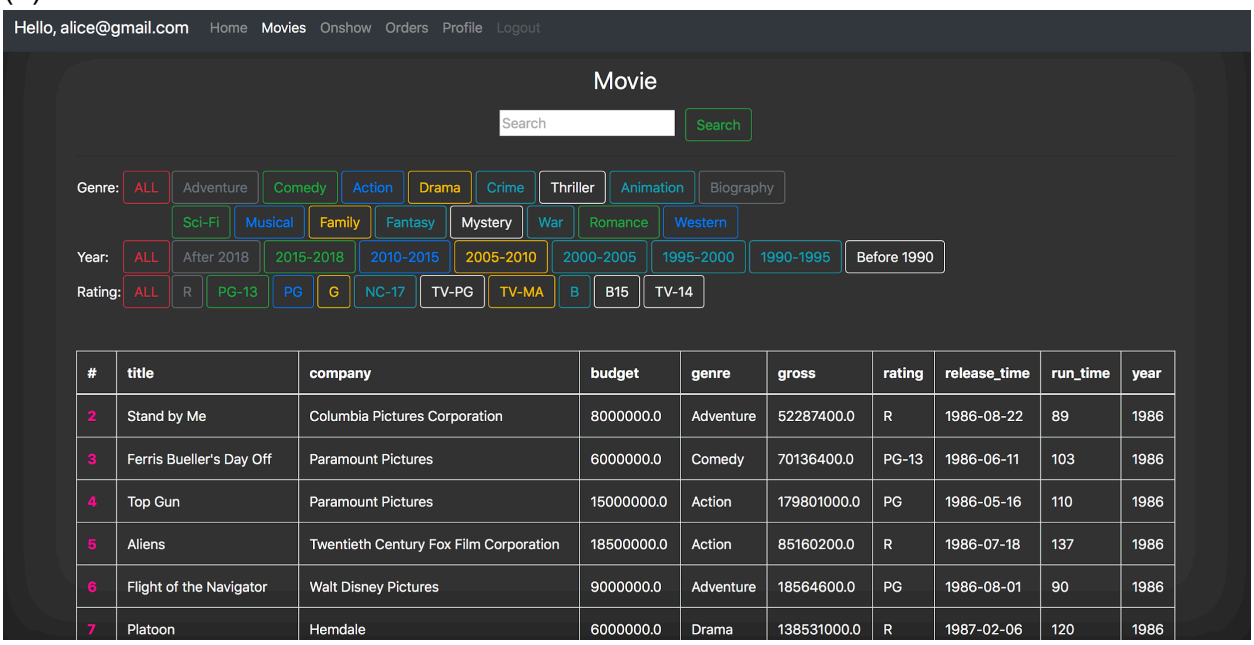

(4)Register



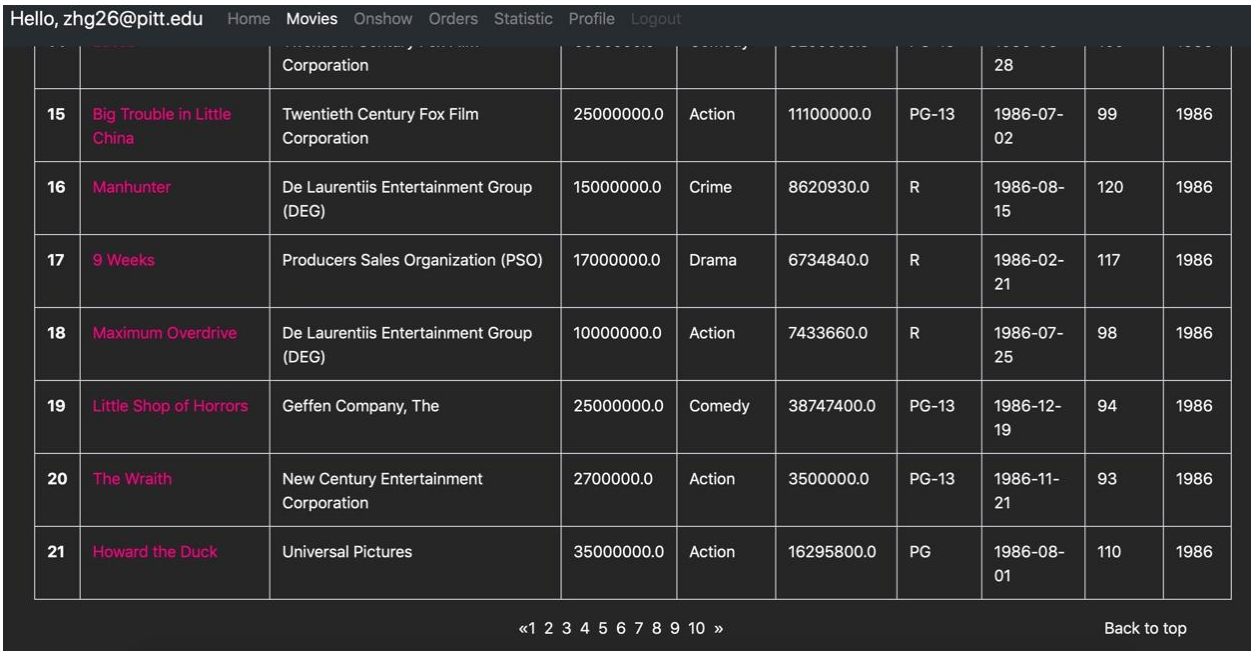

(5)User Homepage




(6)Movies



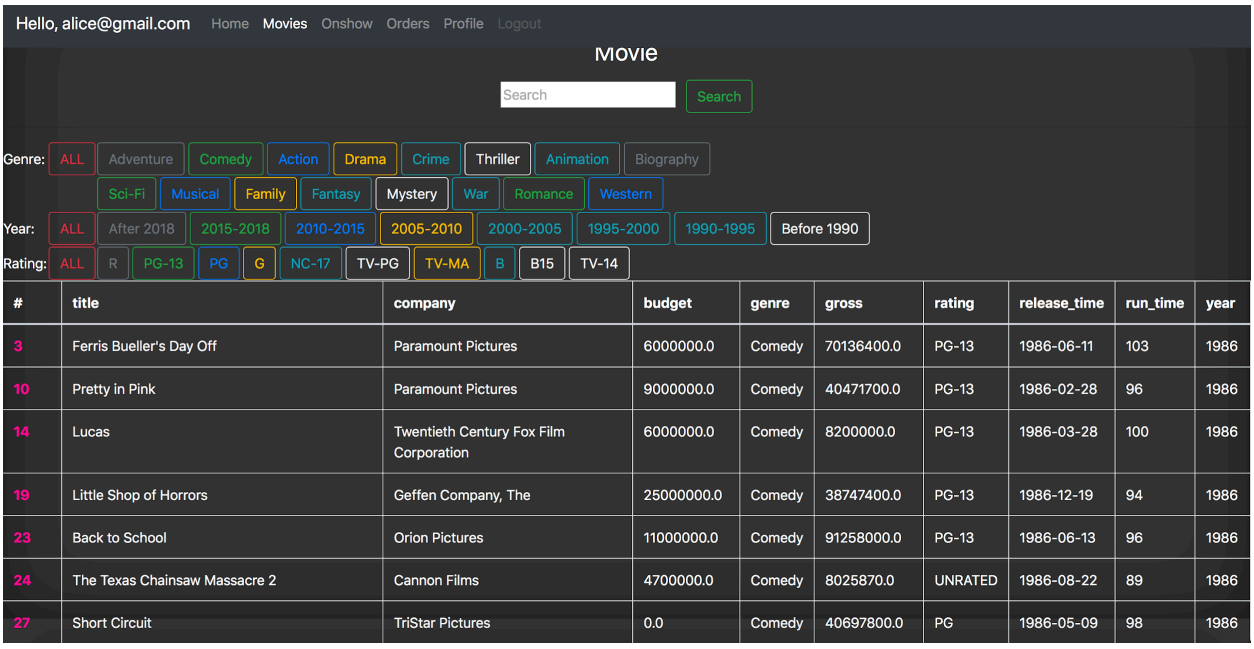

One page can only contain 20 items




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Corporation | | | | | 28 | | |
| 15 | Big Trouble in Little China | Twentieth Century Fox Film Corporation | 25000000.0 | Action | 11100000.0 | PG-13 | 1986-07-02 | 99 | 1986 |
| 16 | Manhunter | De Laurentiis Entertainment Group (DEG) | 15000000.0 | Crime | 8620930.0 | R | 1986-08-15 | 120 | 1986 |
| 17 | 9 Weeks | Producers Sales Organization (PSO) | 17000000.0 | Drama | 6734840.0 | R | 1986-02-21 | 117 | 1986 |
| 18 | Maximum Overdrive | De Laurentiis Entertainment Group (DEG) | 10000000.0 | Action | 7433660.0 | R | 1986-07-25 | 98 | 1986 |
| 19 | Little Shop of Horrors | Geffen Company, The | 25000000.0 | Comedy | 38747400.0 | PG-13 | 1986-12-19 | 94 | 1986 |
| 20 | The Wraith | New Century Entertainment Corporation | 2700000.0 | Action | 3500000.0 | PG-13 | 1986-11-21 | 93 | 1986 |
| 21 | Howard the Duck | Universal Pictures | 35000000.0 | Action | 16295800.0 | PG | 1986-08-01 | 110 | 1986 |

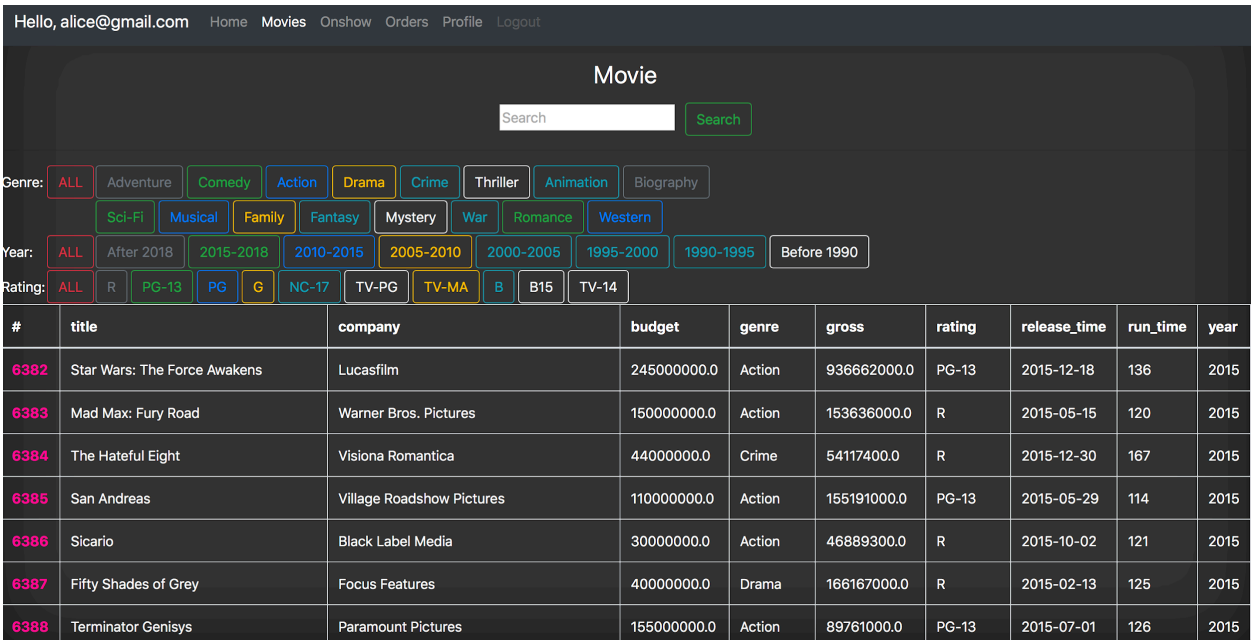


(7)Click on comedy tag of genre




| # | title | company | budget | genre | gross | rating | release_time | run_time | year |
|---|---|---|---|---|---|---|---|---|---|
| 3 | Ferris Bueller's Day Off | Paramount Pictures | 6000000.0 | Comedy | 70136400.0 | PG-13 | 1986-06-11 | 103 | 1986 |
| 10 | Pretty in Pink | Paramount Pictures | 9000000.0 | Comedy | 40471700.0 | PG-13 | 1986-02-28 | 96 | 1986 |
| 14 | Lucas | Twentieth Century Fox Film Corporation | 6000000.0 | Comedy | 8200000.0 | PG-13 | 1986-03-28 | 100 | 1986 |
| 19 | Little Shop of Horrors | Geffen Company, The | 25000000.0 | Comedy | 38747400.0 | PG-13 | 1986-12-19 | 94 | 1986 |
| 23 | Back to School | Orion Pictures | 11000000.0 | Comedy | 91258000.0 | PG-13 | 1986-06-13 | 96 | 1986 |
| 24 | The Texas Chainsaw Massacre 2 | Cannon Films | 4700000.0 | Comedy | 8025870.0 | UNRATED | 1986-08-22 | 89 | 1986 |
| 27 | Short Circuit | TriStar Pictures | 0.0 | Comedy | 40697800.0 | PG | 1986-05-09 | 98 | 1986 |

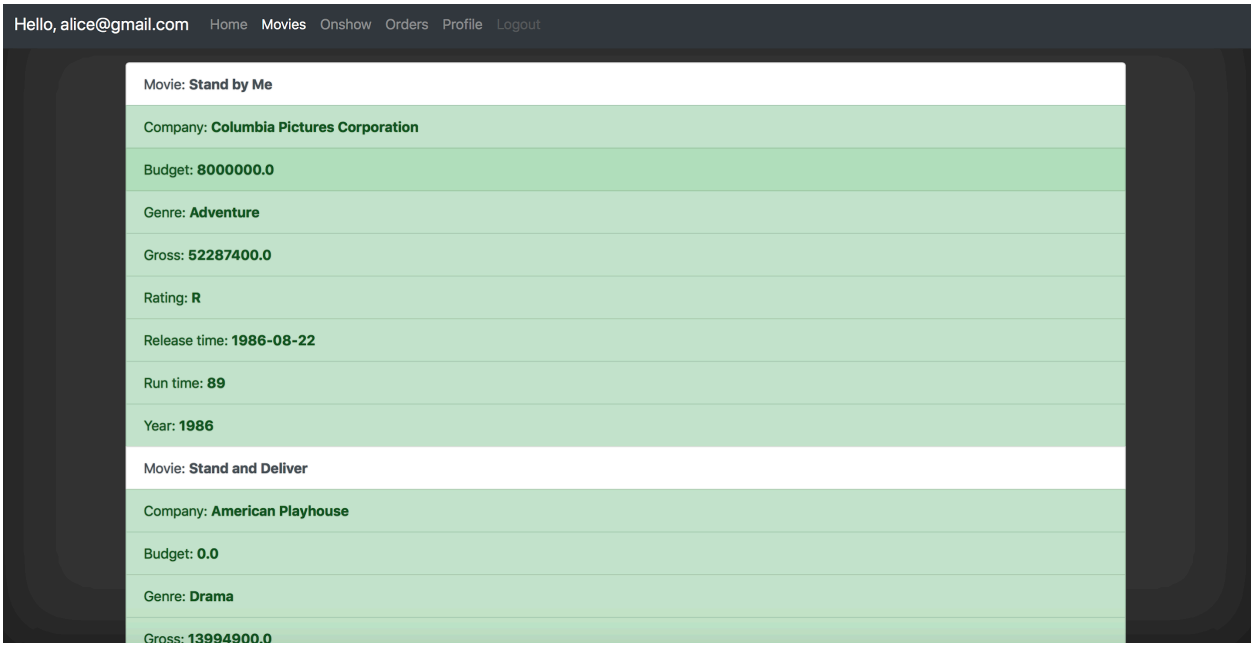
(8)Click 2015-2018 tag of year

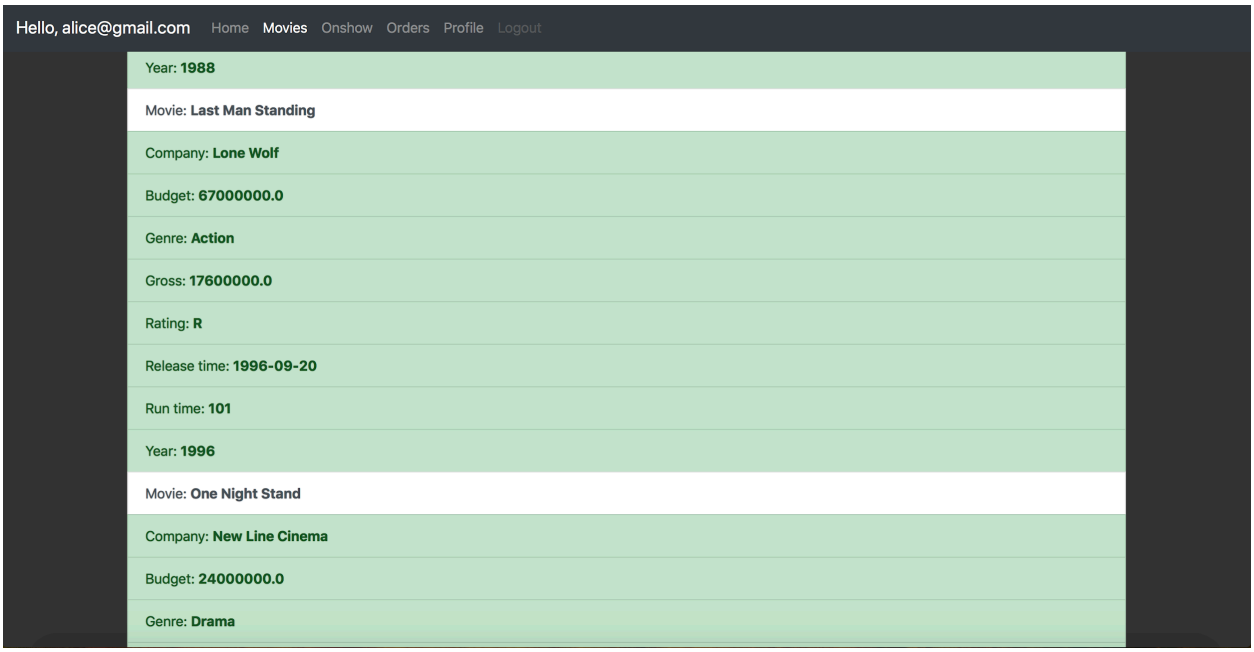



| # | title | company | budget | genre | gross | rating | release_time | run_time | year |
|---|---|---|---|---|---|---|---|---|---|
| 6382 | Star Wars: The Force Awakens | Lucasfilm | 245000000.0 | Action | 936662000.0 | PG-13 | 2015-12-18 | 136 | 2015 |
| 6383 | Mad Max: Fury Road | Warner Bros. Pictures | 150000000.0 | Action | 153636000.0 | R | 2015-05-15 | 120 | 2015 |
| 6384 | The Hateful Eight | Visiona Romantica | 44000000.0 | Crime | 54117400.0 | R | 2015-12-30 | 167 | 2015 |
| 6385 | San Andreas | Village Roadshow Pictures | 110000000.0 | Action | 155191000.0 | PG-13 | 2015-05-29 | 114 | 2015 |
| 6386 | Sicario | Black Label Media | 30000000.0 | Action | 46889300.0 | R | 2015-10-02 | 121 | 2015 |
| 6387 | Fifty Shades of Grey | Focus Features | 40000000.0 | Drama | 166167000.0 | R | 2015-02-13 | 125 | 2015 |
| 6388 | Terminator Genisys | Paramount Pictures | 155000000.0 | Action | 89761000.0 | PG-13 | 2015-07-01 | 126 | 2015 |

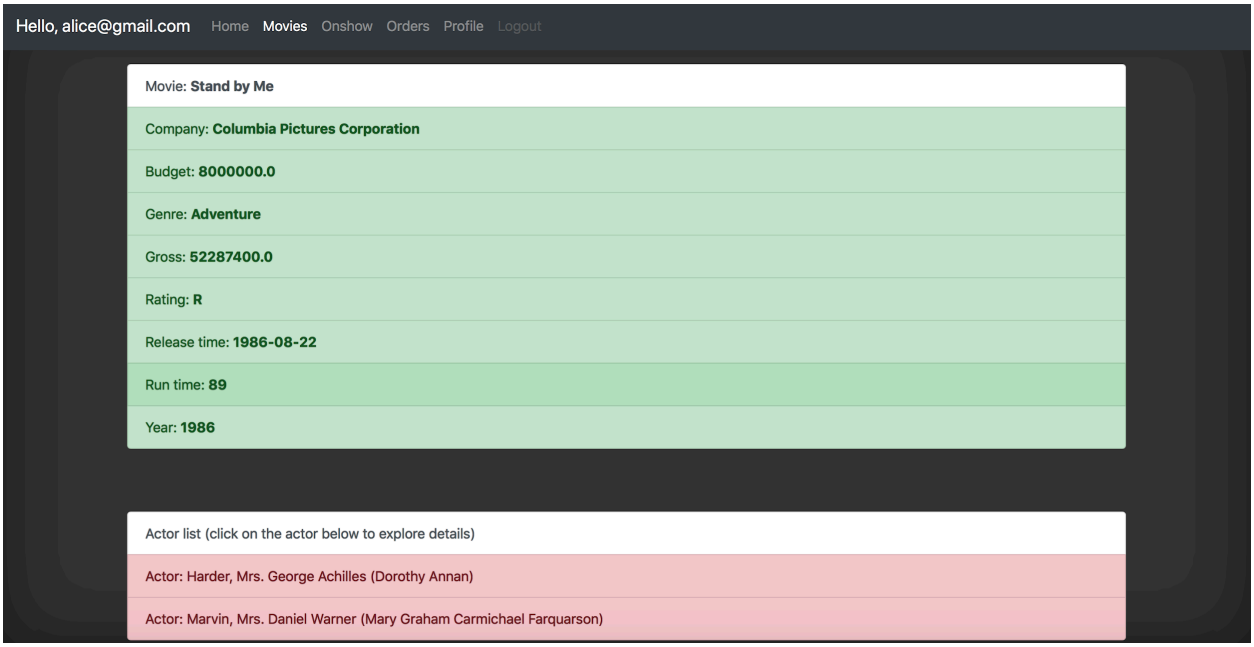
(9) Click PG tag of rating

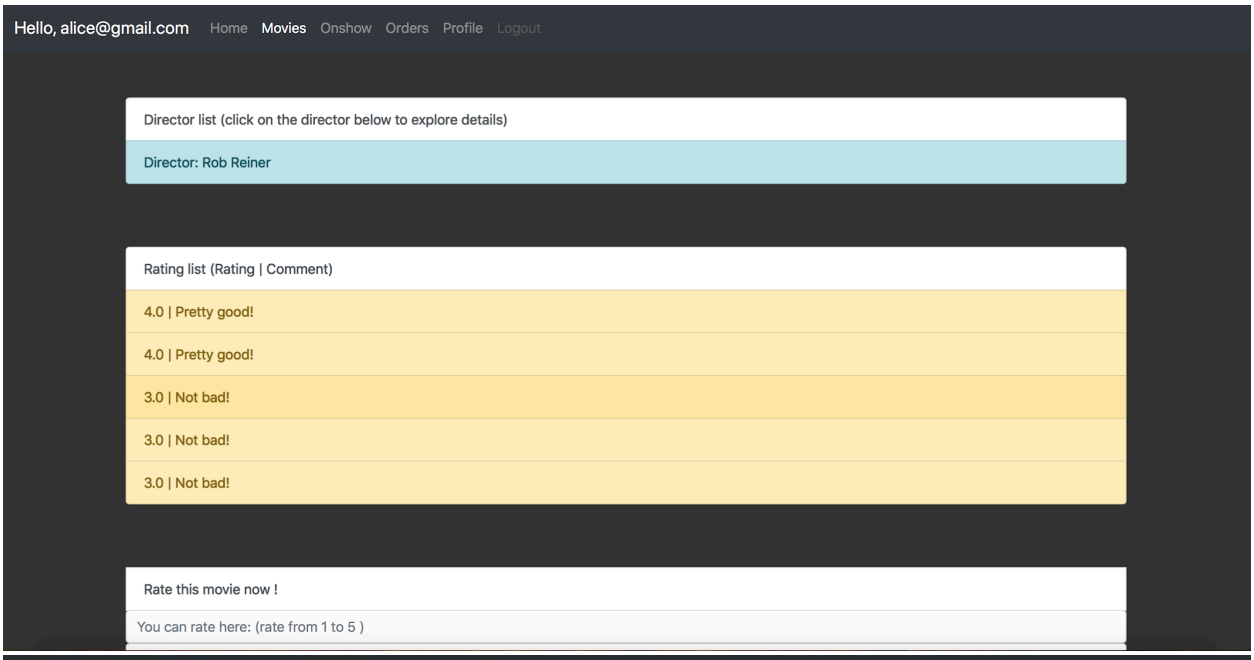
(10) Input “Stand” keyword in the search bar, the website return the movie whose title contains “stand”.

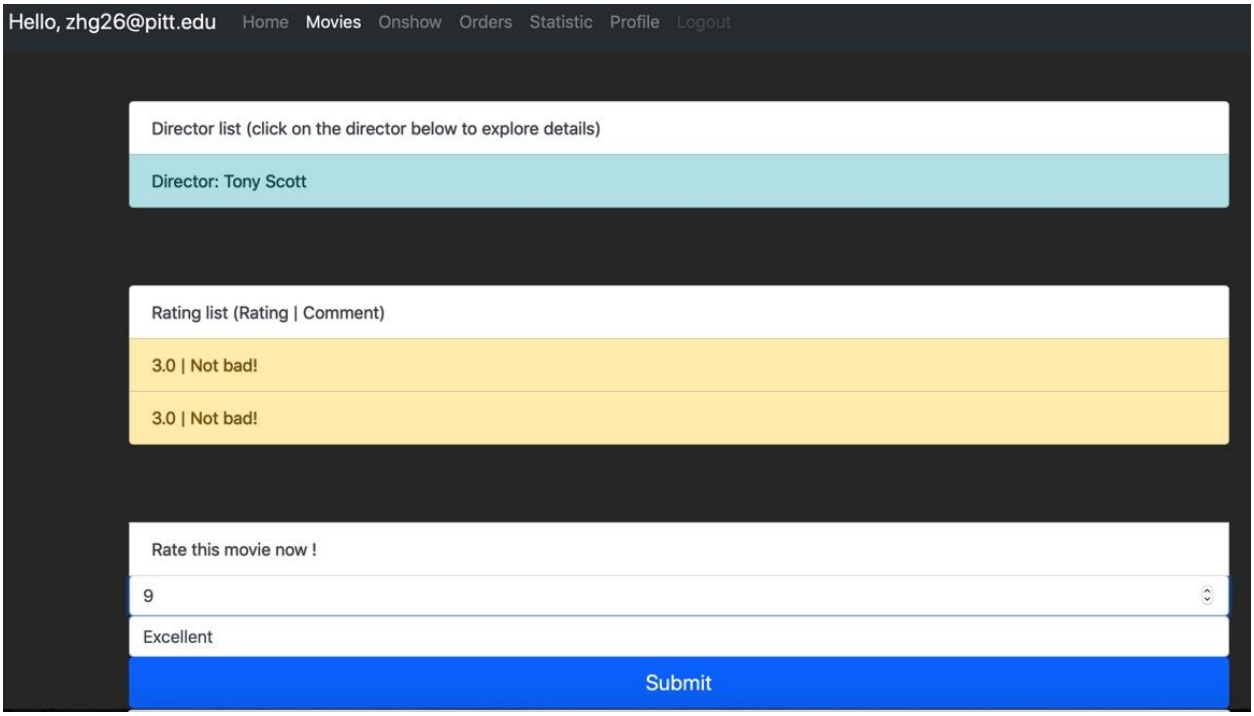




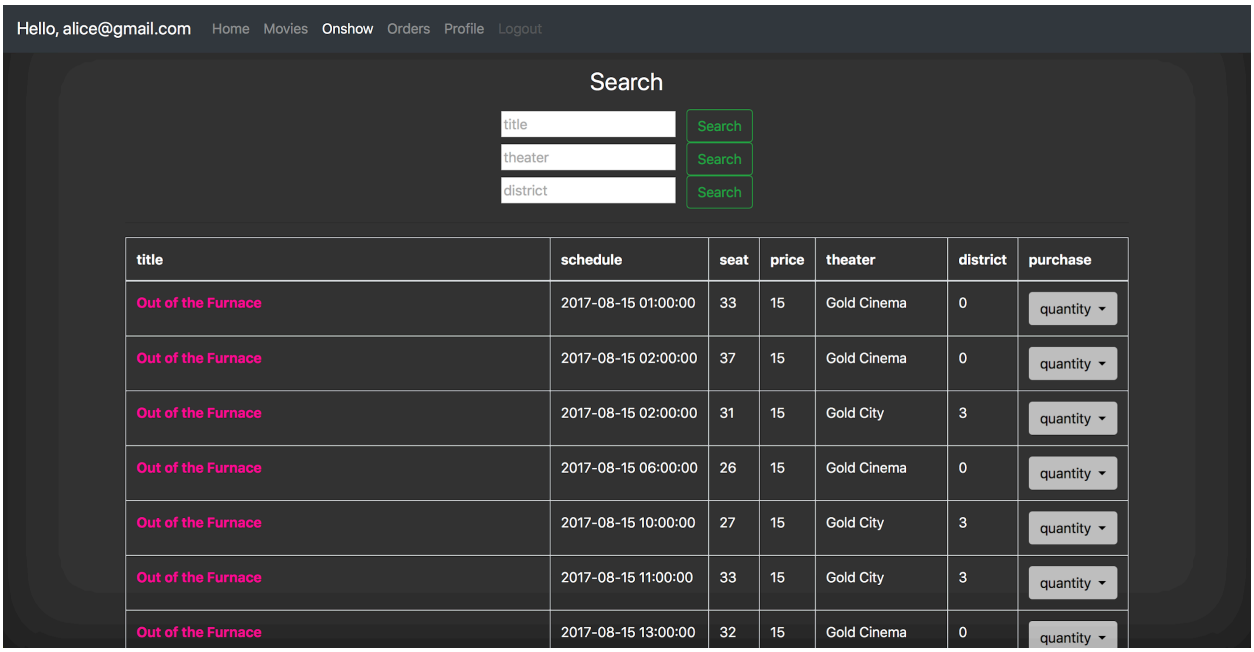

## (11) Movie detail page



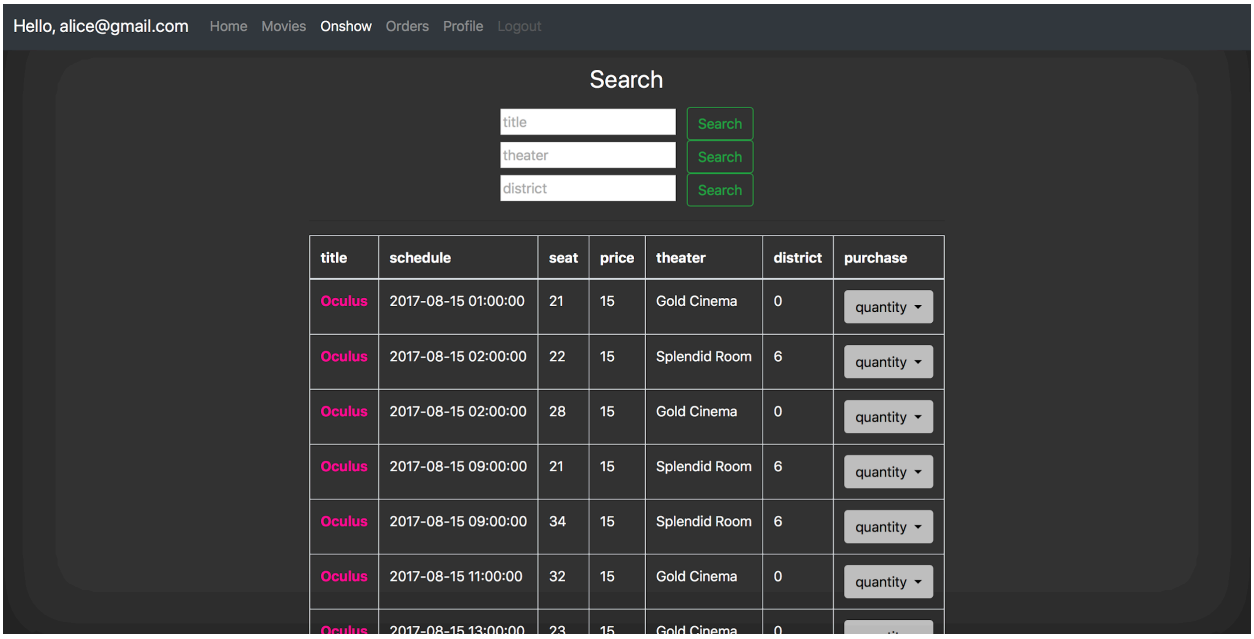


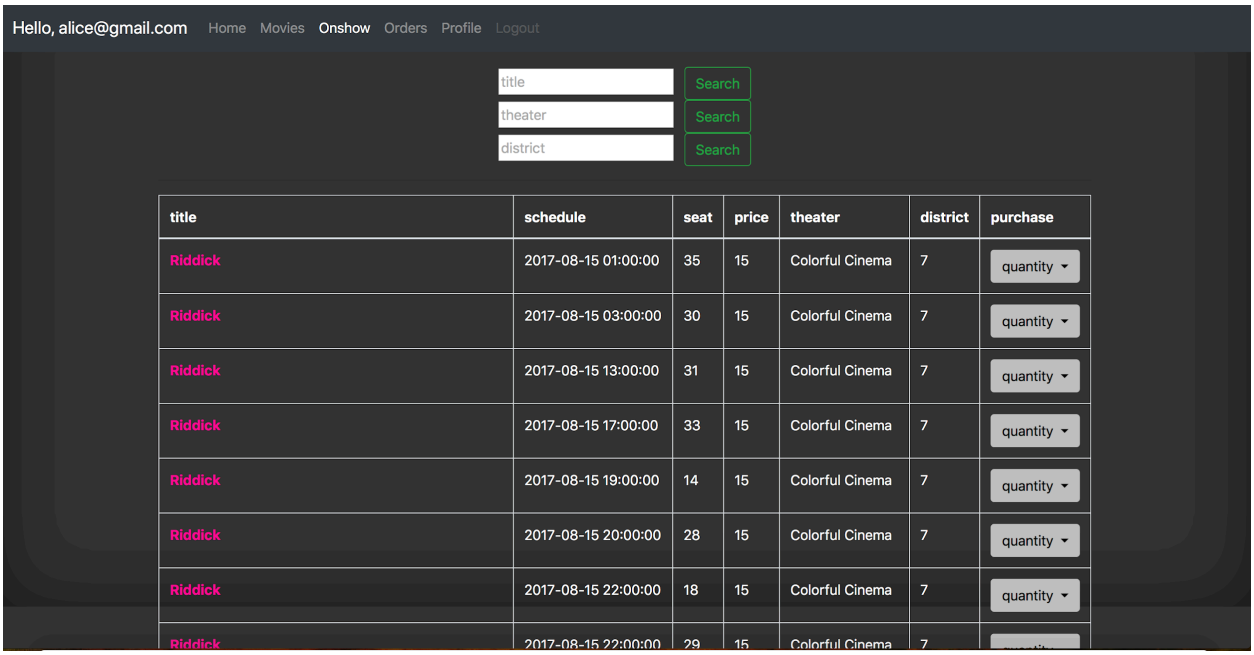

(12) Check on show movies




(13) Search for “Oculus”

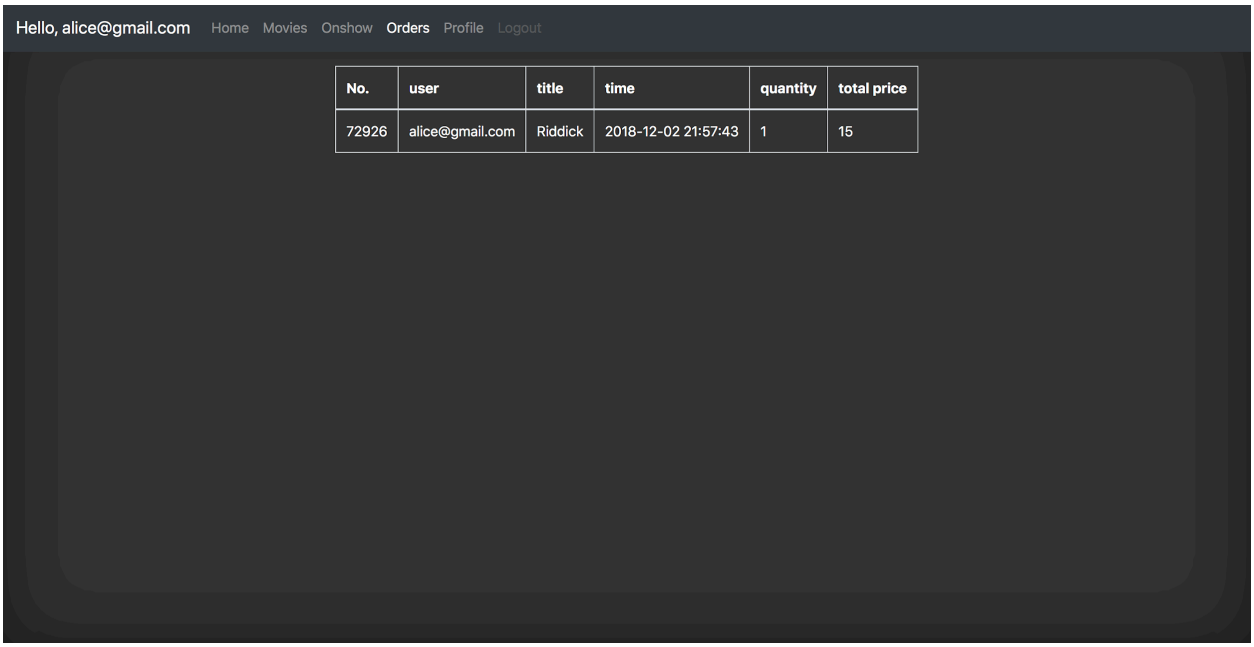


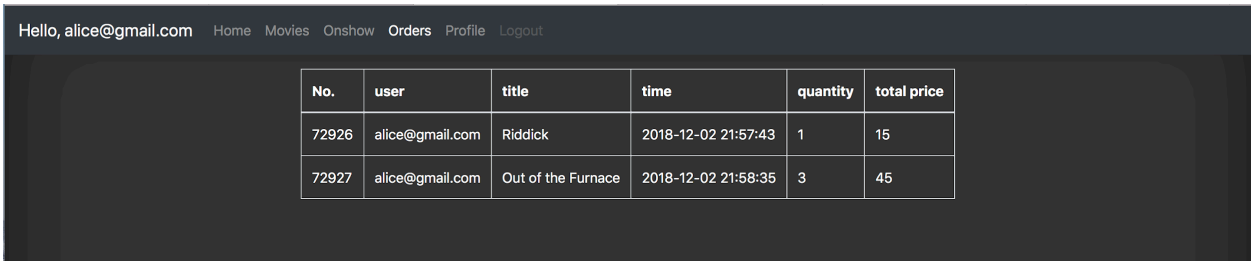

(14) Search the movies in district 7



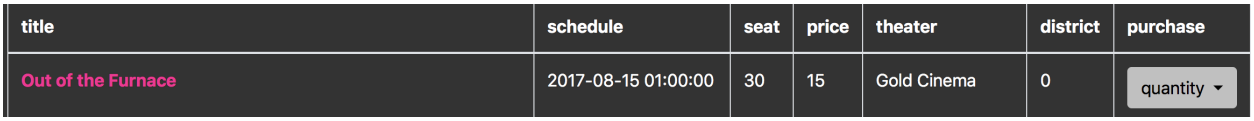

(15) When users purchase tickets, they can click on quantity button and choose the number of tickets they want. If they purchase successfully, website will return a success message.


(16) The order information

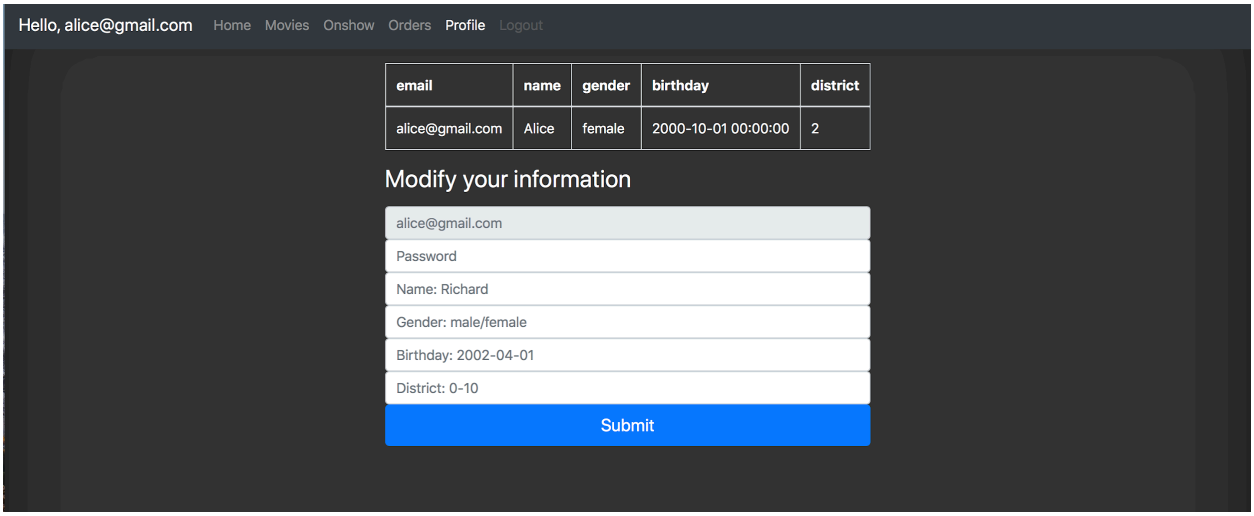



The number of seat decrease.



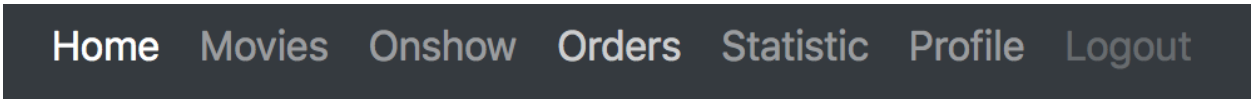

(17) User's profile

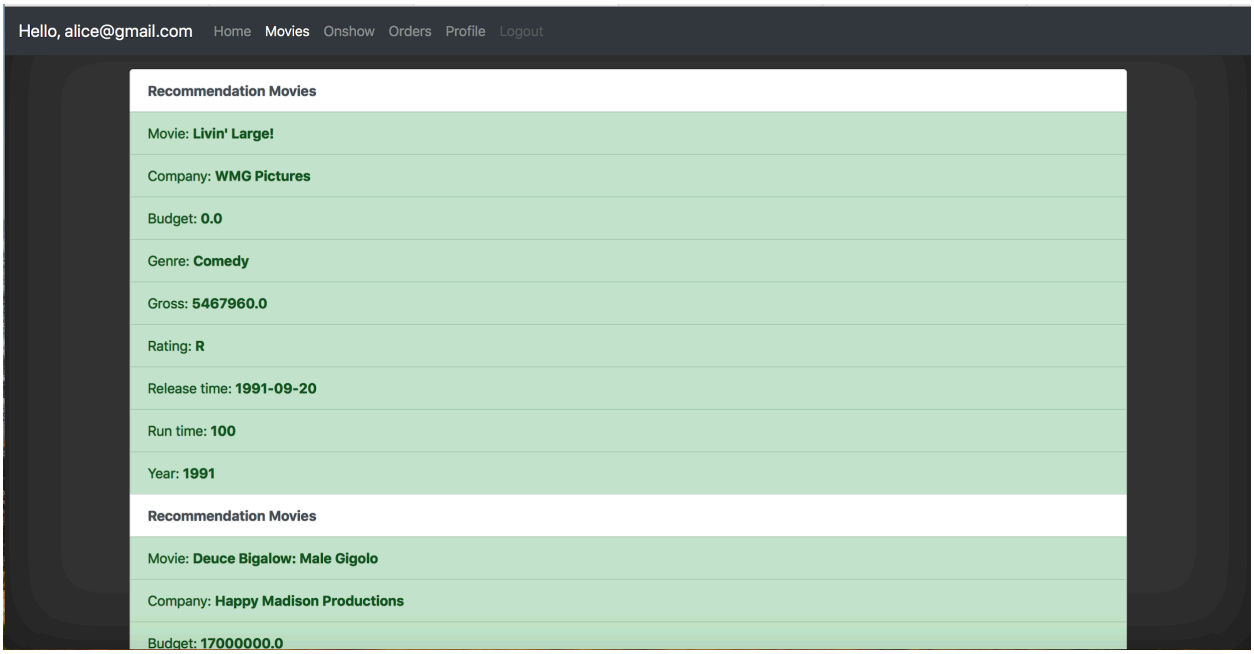


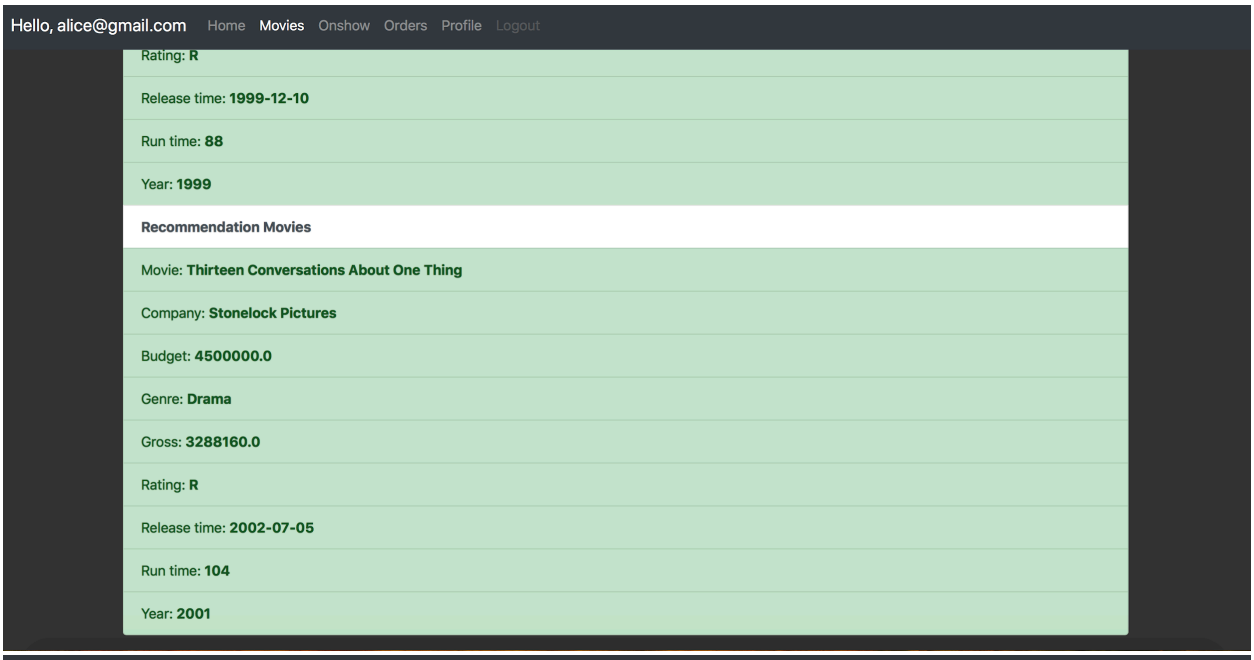

(18) Update user's information

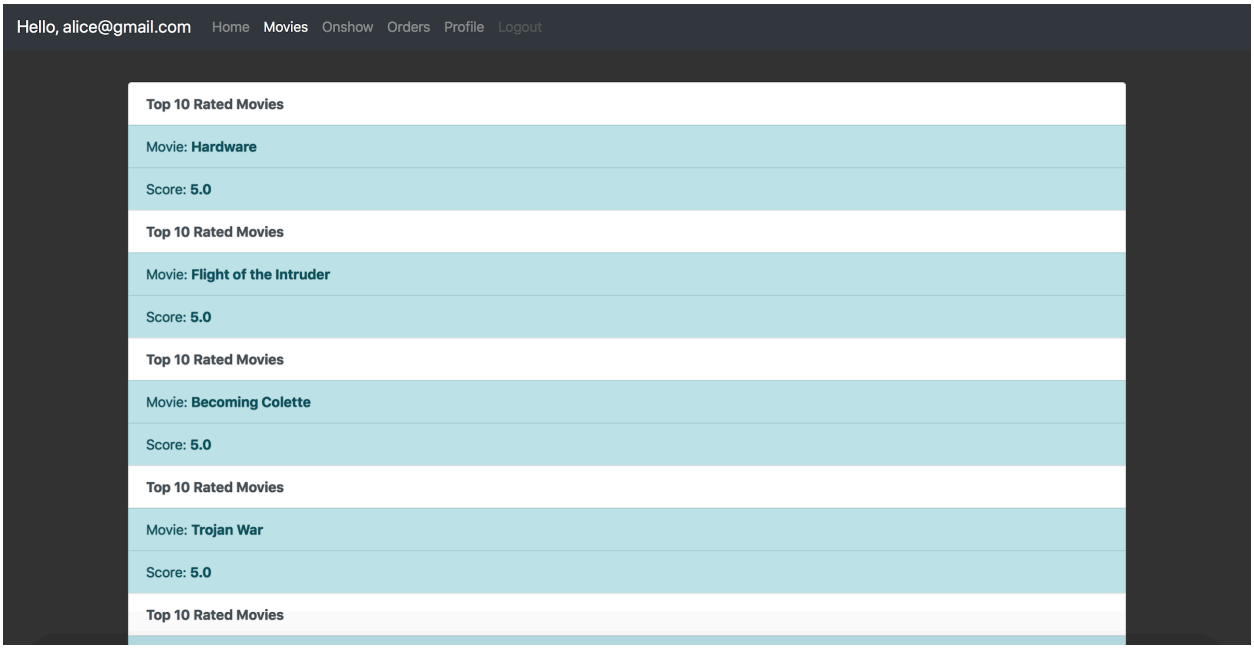
| email | name | gender | birthday | district |
| --- | --- | --- | --- | --- |
| alice@gmail.com | Alice Joe | female | 2000-10-01 00:00:00 | 2 |

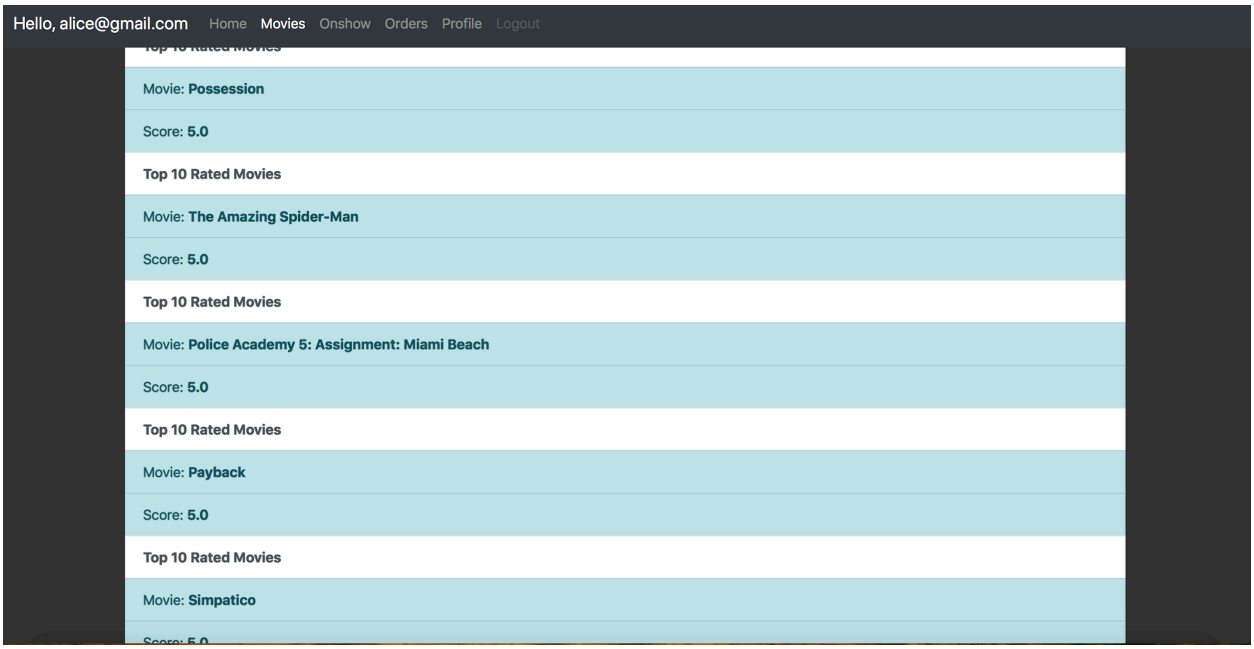
(19)Click the Statistic button on user home page, the system will recommend movies to the user. Neo4J will recommend three movies that the user may interest in. Besides, system will also select out the top 10 movies to user.

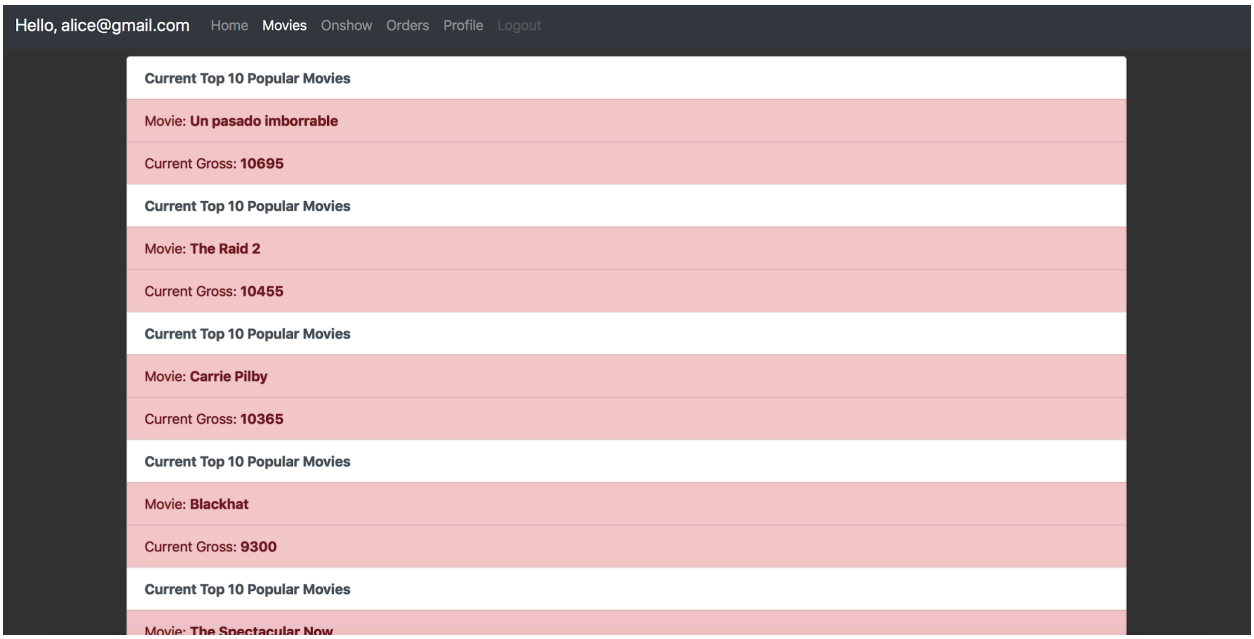


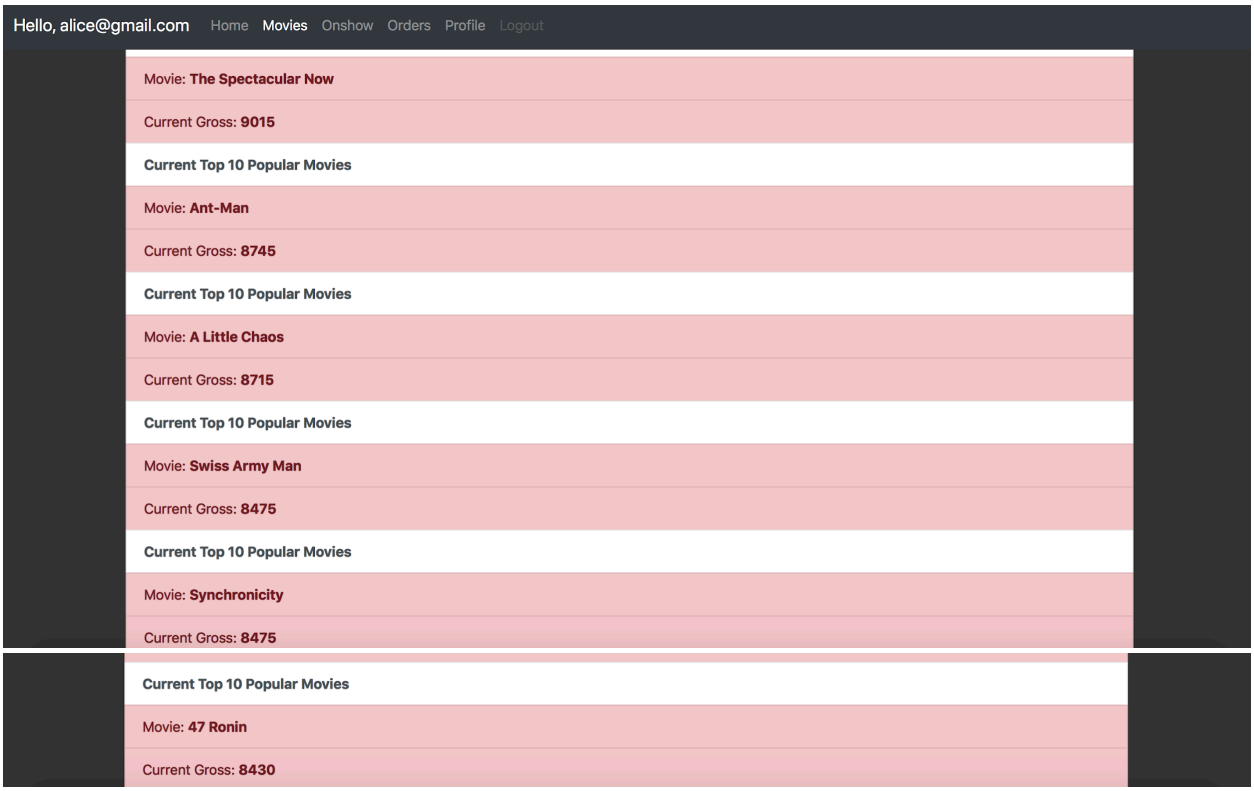




Hello, alice@gmail.com
Home
Movies
Onshow
Orders
Profile
Logout
Rating: R
Release time: 1999-12-10
Run time: 88
Year: 1999
Recommendation Movies
Movie: Thirteen Conversations About One Thing
Company: Stonelock Pictures
Budget: 4500000.0
Genre: Drama
Gross: 3288160.0
Rating: R
Release time: 2002-07-05
Run time: 104
Year: 2001

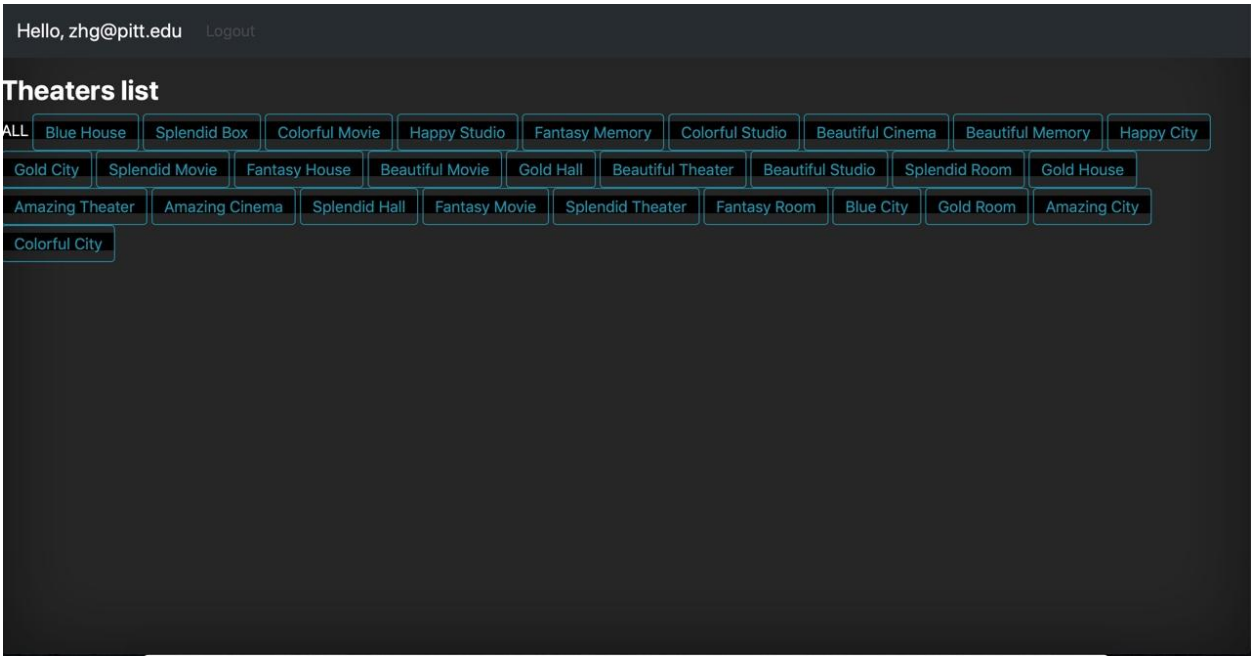

Hello, alice@gmail.com
Home
Movies
Onshow
Orders
Profile
Logout
Top 10 Rated Movies
Movie: Hardware
Score: 5.0
Top 10 Rated Movies
Movie: Flight of the Intruder
Score: 5.0
Top 10 Rated Movies
Movie: Becoming Colette
Score: 5.0
Top 10 Rated Movies
Movie: Trojan War
Score: 5.0
Top 10 Rated Movies


Hello, alice@gmail.com
Home
Movies
Onshow
Orders
Profile
Logout
Movie: Possession
Score: 5.0
Top 10 Rated Movies
Movie: The Amazing Spider-Man
Score: 5.0
Top 10 Rated Movies
Movie: Police Academy 5: Assignment: Miami Beach
Score: 5.0
Top 10 Rated Movies
Movie: Payback
Score: 5.0
Top 10 Rated Movies
Movie: Simpatico

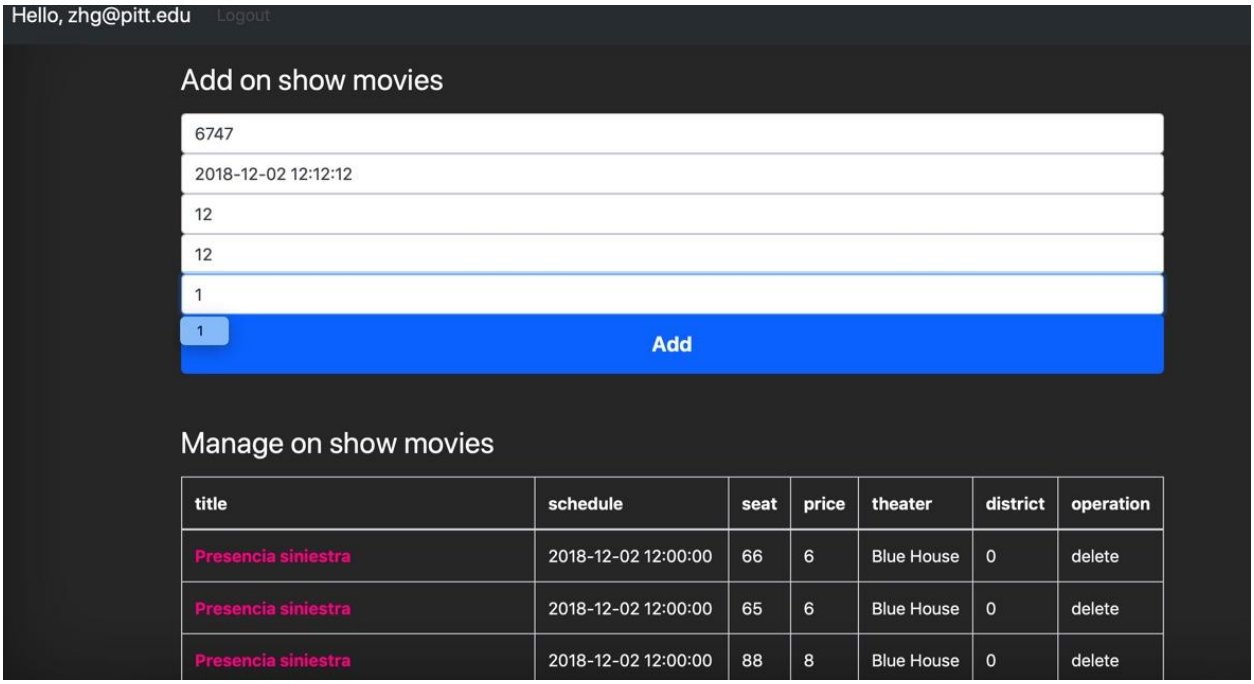

Hello, alice@gmail.com
Home
Movies
Onshow
Orders
Profile
Logout
Current Top 10 Popular Movies
Movie: Un pasado imborrable
Current Gross: 10695
Current Top 10 Popular Movies
Movie: The Raid 2
Current Gross: 10455
Current Top 10 Popular Movies
Movie: Carrie Pilby
Current Gross: 10365
Current Top 10 Popular Movies
Movie: Blackhat
Current Gross: 9300
Current Top 10 Popular Movies



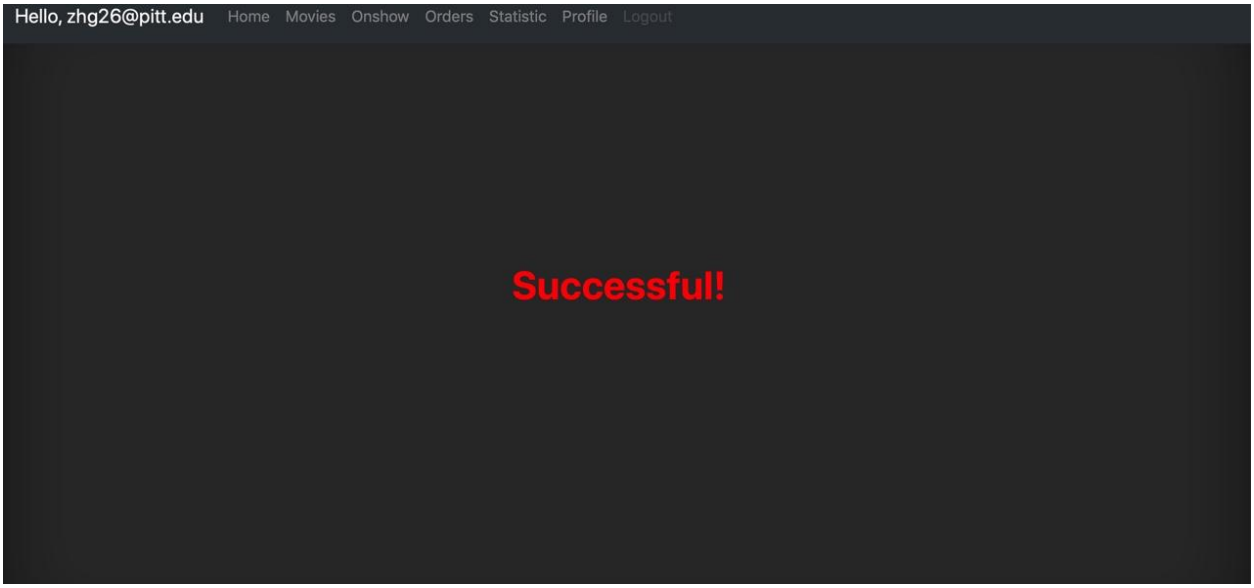

(20) Operator page




(21) Add and delete movies




(22) Operation success




## 8. Testing and Error cases

### Black box testing

The black-box test is to test whether each function can be used normally. In the test, the system is regarded as a black box that cannot be opened. It is tested in the system interface without considering the internal structure and internal characteristics of the

system. It only checks whether the function of the system is normally used according to the requirements specifications. Whether the system can properly receive input data and produce correct output information. The black box test focuses on the external structure of the system and does not consider the internal logical structure. It mainly tests the  interfaces and the functions.

| Number | Name | Operation | Result |
|---|---|---|---|
| 1 | User login | Enter user's email and password | If the user exist, skip to the user's home page. Otherwise, return error message. |
| 2 | User sign in | Enter user's information on sign in page | If the user is not in the database, return success message. If the user exist, return a unsuccess message. |
| 3 | Browsing information | Click on movie tag | Return the list of movie information. |
| 4 | Search | Enter a word and enter a movie's name | If the input is a single word, system return a list of movie titles that contain the word.<br>If the input is the title of a movie, system returns the detail information of the movie.<br>If system cannot find out the movie, return error message. |
| 5 | Purchase tickets | Click on show button and click quantity button. Choose the number of tickets. | Purchase success, if the number of seat is more than the purchasement |

| | | | |
|---|---|---|---|
| | | | quantity. Else return unsuccess message. |
| 6 | Checking order history | Click on order button | Return a list of all orders |
| 7 | Operator management | Click on one theatre<br>Add new movies | Return the list of on show movies in the theatre, adding success. |
| 8 | Recommend system | Click on statistic button | Return 3 recommend movies and top 10 movies. |

We use the website we create to check whether all parts are able to run well. Every button, every link, and every input box is used and checked in the front-end by us for several time and nearly every function runs well.

We check every function in the back-end carefully, especially for the parameters which are used for links from one page to another.

## 9.System limitations and possible improvements

To avoid sql injection, we have tried some methods such as validation in Front-end, which the users were limited to enter the email format. We also deal with the input string in the back-end and make the string validated. Additionally, we compile the parameters of sql query by format of pymysql rule. However, there are some errors when we use the methods above. So, the only way is to use the default format.

The tables in our database which are created by us obey the rule of normal, but some issues occur when we create sql query in practice, for example, the run-time of sql function is too slow when we join many tables. Thus, for the speed, it is necessary to discard the normal roles when it is in practice.