The Chinese University of Hong Kong, Shenzhen

CSC 3170

Database System

Group 26 Report: Rotten Potatoes

Authors:	*Student Number:*
郑时飞	119010465
朱伯源	119010485
李易	119010156
施天昊	120090472
汪明杰	119010300

Saturday 14th May, 2022

Contents

1 Introduction

As an art and commercial work, movies have become pursuits of many people and generated tremendous value. Currently, there are two types of online movie platforms. On the one hand, platforms including Rotten Tomatoes and Douban gives people rich information regarding movies. On the other hand, sites like Netflix offers precise, personalized recommendation to users. Such a recommendation strategy allows users to discover more films they enjoy and attracts more users to the platform.

However, currently, no site combines the strengths of the mentioned movie sites. Besides, existing popular movie websites are full of quarrels and controversies from the perspective of the community environment. The movie recommendations are influenced deeply by the advertising, not the quality of films. Therefore, to satisfy the real requirements of movie lovers, we built a movie database called **Rotten Potatoes**, based on which, users can talk about their reviews of the movie freely. It also offers plenty of searching functions and personalized recommendations from the platform.

We use python packages **Requests** and **BeautifulSoup** to obtain the required data from IMDB, store it in a MySQL database. Then we build our searching functions and personalized moive recommendations on those data. We also build a user-friendly web using **amis** as the frontend and **express** as the backend.

2 Design

In this section, we focus on the design of the Entity-Relationship Model, the reduction from ER diagram into relational schemas, constraints and index.

2.1 Entity-Relationship Model

As shown in the ER diagram, there are 5 entities and 3 relationships.

2.1.1 Entities

- **Entity "crew"** The crew participating in shooting movies. Each crew has identifying id, name, photo URL, introduction and birth date. The entity "crew" is a generalization of entity "directors" and "actors".
  - **Entity "directors"** The directors that direct movies. A specification of entity "crew". Each director has the same attributes as the crew.
  - **Entity "actors"** The actors that act in movies. A specification of entity "crew". Each actor has the same attributes as the crew.
- **Entity "users"** The users of our movie website. Each user has identifying id, name, avatar URL and password.
- **Entity "movies"** The movie information. Each movie has identifying id, name, cover URL, introduction, release year and genres, where genres is a multivalued attribute.

2.1.2 Relationships

- **Relationship "direct"** Directors direct movies. A many-to-one relationship between entity "movies" and "directors", which means a director can direct multiple movies while a movie can only be directed by one director, in our assumption.
- **Relationship "act"** Actors act in movies. A many-to-many relationship between entity "movies" and "actors". An actor acts in a movie as a character with an attribute character name.
- **Relationship "comment"** Users comment movies. A many-to-many relationship between entity "movies" and "users". A user comments a movie by a comment with attributes comment date, rate and content.

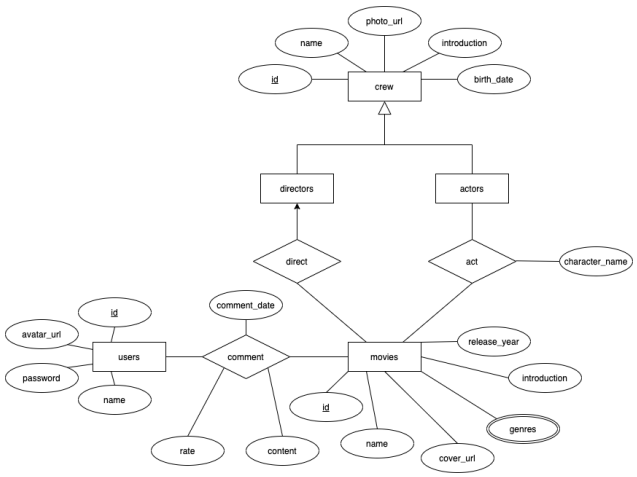

Figure 1: ER Diagram

2.2 Relational Schema

As shown in the relational schema diagram, there are 7 schemas.

The following reductions are made:

- The attribute "genres" in entity "movies" is reduced to schema "genres" with attributes "genres_name" and "movie_id" as a foreign key referencing "id" of "movies". Both of the attributes form a primary key to make sure no redundant genres in a movie.
- The relationship "direct" is reduced to attribute "director_id" as a foreign key referencing "movies" for the many-to-one relationship.
- The relationship "act" is reduced to schema "characters". Except for attribute "character_name", extra attributes "movie_id" as a foreign key referencing "movies" and "actor_id" as a foreign key referencing "actors" are added for the many-to-many relationship. Attribute "id" is also added to allow an actor to act as multiple characters in the same movie.
- The relationship "comment" is reduced to schema "comments". Except for attributes "comment_date", "rate" and "content", extra attributes "movie_id" as a foreign key referencing

"movies" and "user_id" as a foreign key referencing "users" are added for the many-to-many relationship. Attribute "id" is also added to allow a user to make multiple comments on the same movies.

- All identifying "id"s are reduced to the primary keys.

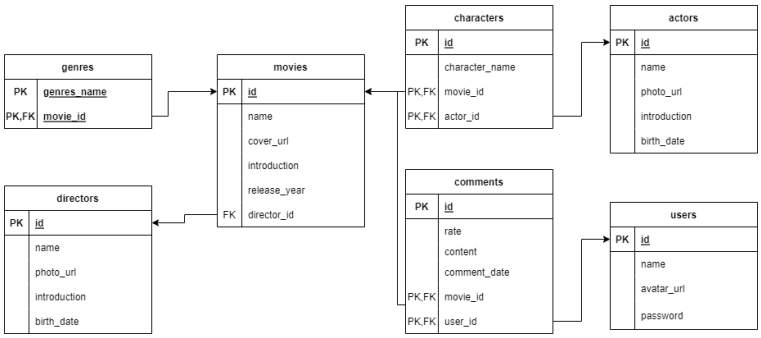

Figure 2: Relational Schema Diagram

2.3 Constraint

3 types of constraints are further added:

Not Null Constraints

- All "id" attributes as they are primary key and thus automatically becoming not null.
- "name" attribute of schema "movies".
- "name" and "director_id" attributes of schema "directors", as a movie is directed by exactly one director.
- "name" attribute of schema "actors".
- "name" and "password" attributes of schema "users".
- "actor_id", "movie_id" and "character_name" attributes of schema "characters", as it is a weak entity of schemas "actors" and "movies".
- "user_id", "movie_id", "rate", "content" and "comment_date" attributes of schema "comments", as it is a weak entity of schemas "users" and "movies".
- "genres_name", "movie_id" attributes of schema "genres", as it is a multivariate attribute of schema "movies".

Unique Constraint A unique constraint is added to the "name" attribute of schema "users" as by our assumption there should be no repeating user names.

Check Constraint A check constraint is added to the "rate" attribute of schema "comments" to make sure the rate is from 0 to 10.

2.4 Index

The following attributes are indexed to make searching faster:

- "name" and "release_year" attributes of schema "movies".
- "name" and "birth_date" attributes of schema "directors".
- "name" and "birth_date" attributes of schema "actors".
- "name" attribute of schema "users".

3 Implementation

In this section, we will introduce the implementation of our movie website. All codes of this project can be accessed from https://github.com/warin2020/rotten-potatoes.

3.1 Frontend

The frontend of our project is constructed using ***amis***, a low-code frontend framework. It can generate a website using JSON configurations, which is suitable for developing a lightweight and agile application like this project. We constructed the following pages:

- Movie: a list of all the movies with their name and posters. Each movie is a "Card" widget linking to the movie detail page.
- Actor: a list of all the actors/actresses, also implemented using the "Card" widget, containing links to the detailed information.
- Director: a list of all the directors implemented similarly to the **Actor** page
- Comment: this page contains the latest comments that users release.
- Me: a portal for users to edit their information. Including updating avatar, name and password. This page also contains a list of movies recommended to the user.
- Search: to search for movies/actors/users, we implemented three different pages. The details will be discussed in the **Sample Queries** section.

Since our application is user-oriented, the UI of our website is very concise and user-friendly. Screenshots of our UI can be found in the appendix.

3.2 Backend

Connect to MySQL in Nodejs The query functions in our database are written in javascript language. They can be found in the corresponding files in the **services** folder. To maximize reusability, we wrote a template query function in **query.js** using the *mysql* module of Node.JS. The template function will first access the **.env** file for database configuration(e.g., the port on which the MySQL server is running, the logging username and password). Then, instead of establishing and closing connections to the database on every execution, the template creates a connection pool. Requests from the frontend will pass the query as a rest parameter to the template function, which will then issue the query and return the results (and errors, if any) back to the frontend.

Provide API by Express Router We use the **express** library to provide APIs for frontend users. All the routers are written in the **router/index.js**. Whenever users access a certain page, it will match a router in the file, which triggers the corresponding handler function. For example, after logging in, the user will see the *Movie* page, which contains a list of movies in the database. When the user clicks on any of the movie poser , the frontend will request for page ./movie/detail/<movie_id>, matching a router in index.js (line 26). The router then calls the corresponding handler function, in this case, the *getMovieDetail()* function in **movie.js**, which directs the user to the corresponding movie detail page.

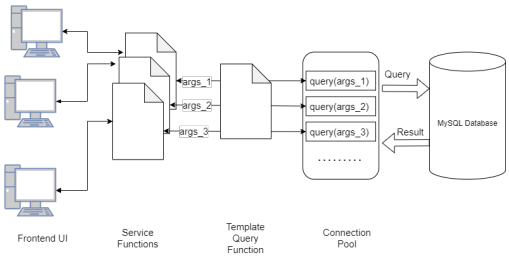

Figure 3: Database connection flowchart

Access Control by jsonwebtoken Base on **jsonwebtoken** package, in **auth/auth.js** using the key in **.env** file **SECRET_KEY** we implemented 2 functions *signToken* and *verifyToken*, by which, in **services/auth.js**, the routing function *login* returns a token to frontend and the middleware function *auth* check whether the token from frontend is correct and not outdated.

3.3 Web Crawler

3.3.1 Basic Workflow

To populate our database with real-world data, we wrote a web crawler to scrap information from IMDB's Top 250 Movie Chart. The crawler is written in Python. It uses the **Requests** library to send https requests. The desired fields are acquired through parsing the page using **BeautifulSoup** library. The detailed process is shown in this flowchart:

1. Access the Top 250 Chart, where all the URLs to the detailed movie pages are located
2. Traverse through the chart. For every movie of the chart:
   (a) Access the detailed movie page, where information regarding the movie can be found.
   (b) From the detailed movie page, we access the casting information.
   For the director:
   - For directors yet to be recorded, access the detailed director page, where information regarding the director can be found.

   For the actors/actresses who casted in this movie:
   - Access the detailed actor/actress page and find detailed information if their details are yet to be recorded.

   (c) The comment section locates at the bottom of the movie page. We collect the rating, comments from this section.
   (d) For each comment, we acquire the user information by accessing their homepage.

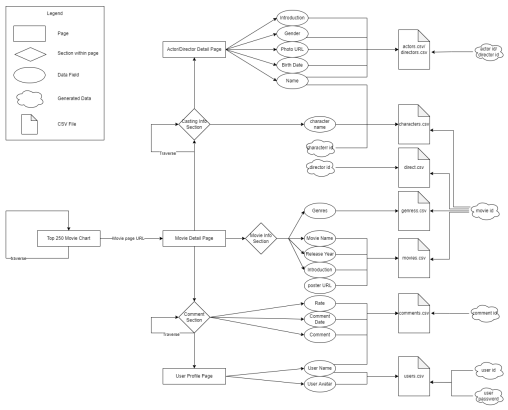

Figure 4: Web crawler flowchart

3.3.2 Encountered Problems and Solutions

During the scrapping process, we encountered some issues.

- The same director or actor/actress can participate in different movies. Therefore, we keep a mapping relationship and check wheter we have record the same person's information. This step is done through hashing the person's name, which takes O(1) complexity.
- We do not have access to each user's password, so we randomly generate a password for each user we scrapped from IMDB. The password is a random combination 5 to 10 numbers and characters.
- Some actors' birth date cannot be achieved from their detailed page. We leave them as NULL.
- The gender of the movie stars is not explicitely listed on their detail page. However, by seeking their role in the movie (i.e, actor/actress), we can acquire their gender.

3.4 Sample Queries

3.4.1 User account & the Portal

User registration When a user try to create his own account,he need to specify his user name. In our system, every user must have a unique account name. Therefore, before granting a registration, we will first query the inputted user name in the **users** table. If the name already exists, the registration will not be accepted and the user should input a new user name. Otherwise, user name as well as user password will be inserted to the user table, with a user ID generated automatically.

User login When a user login, password will be queried from the user table, with user name as the search key. Then, the correct password will be checked using jsonwebtoken, as mentioned in the access control section

The Portal User can modify his avatar, user name and password. This is achieved by updating a record in the **users** table.

Delete user In our system, a user account can be deleted, while the comments on movies he made will be kept even after the account is deleted.This is achieved by associating the comments to a padding user before deleting the user account.

```
SELECT *
FROM   users
WHERE  NAME = register_na
```

Figure 5: User registration verification

```
INSERT INTO users ( name password )
           value
           (
                     inp_name,
                     inp_psw
           )
```

Figure 6: User registration verification insert

```
SELECT *
FROM   users
WHERE  NAME = login_name
```

Figure 7: User login

```
UPDATE users
SET    NAME = new_name
WHERE  id = user_id
```

Figure 8: Update avatar

```
UPDATE users
SET    avatar_url = new_url
WHERE  id = user_id
```

Figure 9: Update user name

```
SELECT password
FROM   users
WHERE  id = user_id;

UPDATE users
SET    password = new_password
WHERE  id = user_id
```

Figure 10: Update user password

```
UPDATE comments
SET    user_id = 0
WHERE  user_id = user_id;

DELETE FROM users
WHERE  id = user_id
```

Figure 11: Delete user

3.4.2 CRUD on movies, actors and directors

Information listing Movies, actors and directors is listed on our web page. Basic information is selected from the corresponding table.

Detail information To show detailed information of movies, actors and directors, we need to join several tables. **Actors** table is joined with the **characters** table to get all characters played by an actor, while **directors** table is joined with **movies** table to select all movies directed by the director. For the detail page of movies, 4 table joins are needed to acquire all necessary data. **Movies** table is joined with **characters** table to get all characters in a specific movie. Joining the **directors** table is also necessary to find the director of a specific movie. All comments and genres of the movie are obtained by joining the **movies** table with the **comments** table and **genres** table, respectively. Figure 16 displays the query used to generate actor details:

Search movies Movies can be searched using their names. Aside from directly searching, we also support filtering movies by their release time, movie rate and movie genres. We also added support for ordering (in ascending or descending order) the result by name, release year and rating.

Search actors Similar to movies, actors are searched by their name, and filtered by their birth date. Also, ordering is supported on the birth date and name.

Search user For user searching, only name search is supported.

```sql
SELECT id,
       name,
       photo_url
FROM   actors
ORDER  BY id
LIMIT  10 offset page_num
```

Figure 12: Listing actor

```sql
SELECT id,
       name,
       photo_url
FROM   directors
ORDER  BY id
LIMIT  10 offset page_num
```

Figure 13: Listing directors

```sql
SELECT id,
       name,
       cover_url
FROM   movies
ORDER  BY id
LIMIT  10 offset page_num
```

Figure 14: Listing movies

```sql
SELECT a.id,
       a.NAME,
       a.photo_url,
       c.character_name
FROM   actors AS a
       INNER JOIN characters AS c
               ON a.id = c.actor_id
WHERE  c.movie_id = movie_id
```

Figure 15: Actor detail

```sql
WITH search_name AS
(
          SELECT     m.NAME,
                     m.id,
                     m.cover_url,
                     m.release_year,
                     Round(Avg(c.rate),1) AS rate
          FROM       movies               AS m
          INNER JOIN comments             AS c
          ON         c.movie_id = m.id
          WHERE      NAME LIKE Concat('%', ?, '%')
          GROUP BY   m.id )
SELECT   NAME,
         id,
         cover_url,
         release_year,
         rate
FROM     search_name
WHERE    release_year BETWEEN ? AND      ?
AND      rate BETWEEN ? AND      ?
ORDER BY
         CASE
                  WHEN ?=1 THEN ??
         end asc,
         CASE
                  WHEN ?=0 THEN ??
         END DESC
```

Figure 16: Movies search

3.4.3 Comments

List comments Join the **comments** table and the **users** table to match the sender of each comment. After joining, all tuples are shown.

```
SELECT m.NAME AS movie_name,
       m.cover_url,
       c.movie_id,
       c.rate,
       c.content,
       c.comment_date,
       c.user_id,
       u.NAME AS user_name,
       u.avatar_url
FROM   movies AS m
       INNER JOIN comments AS c
               ON m.id = c.movie_id
       INNER JOIN users AS u
               ON u.id = c.user_id
ORDER  BY c.id
```

Figure 17: Comments list

Delete comments Comments deletion is achieved by deleting the corresponding tuples comment table.

Add comments Comment addition is achieved by inserting into the comment table.

3.5 Data Analysis

3.5.1 Recommendation System with Restricted Boltzmann Machines

In 2007, Hinton showed that Restricted Boltzmann Machines (RBMs) can be successfully applied to the Netflix dataset and do personalized movie recommendations (Hinton et al., 2007). This paper was given in a DDA course project, but that project requires only the basic RBM without biases, conditional RBM and neighborhood. We reproduce the model proposed by Hinton in this project to do recommendation. We use Contrastive Divergence (CD) to approximate the maximum likelihood of the parameters by Gibbs sampling. The notions for the following procedure can be found in figure 27.

1. Acquire the visible movie ratings $\boldsymbol{V}$ of the user, each $\boldsymbol{v}$ is K binary vector where $\boldsymbol{v}^k = 1$ if the rating is k, otherwise 0.

2. Sample the hidden units $\boldsymbol{h}$ to binary numbers by probability given by $p(h_j = 1|\boldsymbol{V}, \boldsymbol{r}) = \sigma(c_j + \sum_{i=1}^{m}\sum_{k=1}^{K} v_i^k W_{ij}^k + \sum_{i=1}^{M} r_i D_{ij})$, where $\sigma(x) = \frac{1}{1+e^{-x}}$.

3. Reconstruct the visible units by $p(v_i^k = 1|\boldsymbol{h}) = softmax(b_i^k + \sum_{j=1}^{F} h_j W_{ij}^k)$, where $softmax(x_k) = \frac{e^{x_k}}{\sum_{l=1}^{K} e^{x_l}}$. Notice that we only reconstruct the movies the use has rated.

4. Update the parameters by the origin data and reconstructed data. For example, $\Delta W_{ij}^k = \epsilon(< v_i^k h_j >_{data} - < v_i^k h_j >_T)$, where T represents the number of runs we do Gibbs sampling.

5. Prediction is much the same as doing Gibbs sampling except that we can reconstruct (predict) missing ratings.

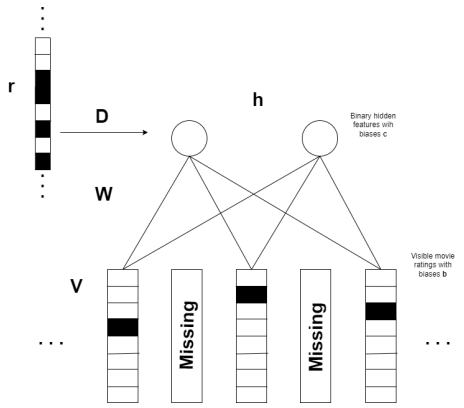

Figure 18: Conditional RBM

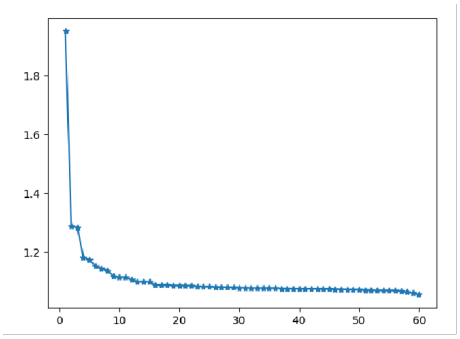

Figure 19: Train RMSE vs. Epochs

Figure 28 shows the evolution of Root Mean Squared Error (RMSE) during training. We achieve an RMSE of 1.04. Every time the user enters the **Me** page on our website, the system will automatically predict the ratings overall movies and recommend the top 8 movies to the user in **Guess you like** module. An example on the webpage is shown in figure 29.

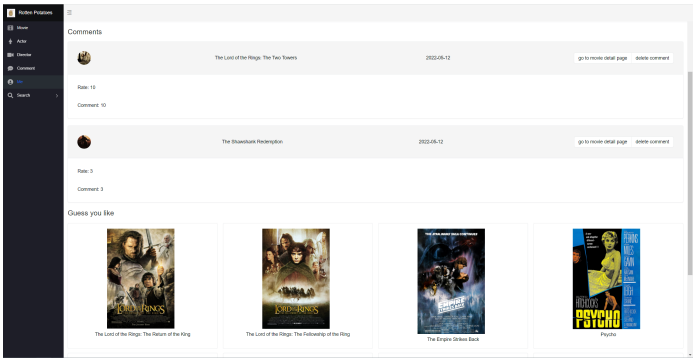

Figure 20: Personalized Movie Recommendation

3.5.2 Neighbourhood Model

Users have their preferences and tend to give similar ratings to movies of the same kind. In other words, movies are correlated and we can utilize this property to improve our predictions $\hat{R}$:

- Define an error matrix by $\widetilde{R} = R - \hat{R}$
- Define similarity between two movies by their column features (user ratings): $d_{AB} = \frac{\widetilde{r}_A^T \widetilde{r}_B}{\|\widetilde{r}_A\|_2 \|\widetilde{r}_B\|_2}$
- Predict the error of one movie through the errors of its neighbours: $\hat{\widetilde{r}}_{iA} = \frac{1}{\sum_{B \in S} |d_{AB}|} \sum_{B \in S} d_{AB} \widetilde{r}_{iB}$, where S is the set of neighbours (with high absolute similarities) of A.
- Improve our RBM predictions by $R^*_{um} = \hat{R}_{um} + \hat{\widetilde{r}}_{um}$ for each $(user, movie)$ pair desired.

We improve the RMSE from 1.04 to 1.02. The improvement is not high because our matrix is sparse. In our data set, many users only give one rating, so it is hard to construct a well defined similarity matrix. The other reason is that we crawl high-rating movies and the biases are not large to give a performance boost. However, we are still able to leverage the data and get some insight into the movie ratings.

For example, *The Godfather: part II* has a high similarity(0.9999) with *The Godfather*(based on the difference between real rating and predicted rating), which is expected. However, the similarity between *The Godfather* and the two sequels of *The Lord of the Rings* diverge. *The Fellowship of the Ring* has a similarity of 0.9999 with *The Godfather*, while its sequel *The Return of the King* has a similarity of -0.9999. This is counterintuitive and we should look into the data. We have three users giving ratings to both *The Godfather* and *The Lord of the Rings: The Return of the King*. The ratings are $(10, 10)$, $(10, 10)$ and $(8, 10)$, respectively. For *The Godfather* and *The Lord of the Rings: The Fellowship of the Ring*, the rating pairs are all $(10, 10)$. A rating of 8 is a relatively low rating in our dataset, hence the algorithm concludes that they are negatively correlated.

4 Result

The website can be accessed through URL http://10.20.9.99:3000. This server can only be accessed within the CUHK(SZ) campus network. In case of access failure, please contact our group member Tianhao SHI (120090472), who is in charge of server maintenance. You can also find the source code on our Github repository. Snapshots of our website can be found in the appendix.

5 Conclusion

In this project, we build a movie database based on large-scale data collected from **IMDB**. Additional attention is paid to assure a **BCNF** form is kept for every schema in our database, thus the redundancy part is eliminated. We also **implement useful functions** for our users to search for the moives, casts or other users using name, time information, rate(only movies) and genres(only movies) as the keyword.

In order to improve user experience and replace the complex database operations with a few simple mouse clicks from users, we build a user-friendly website. Based on that, abundant **personalized interaction behaviors** can be achieved, which serves our users more conveniently. We hold an integral process of account management for users. Users can add comments to a movie and rate it as well as discover their appreciated reviews or users.

It is common for a movie database to recommend movies that the users have a potential interest in. However, how to make a personalized recommendation list of high quality is always a problem. We use a lot of techniques in the data analysis to satisfy the needs of users. We build a **Restricted Boltzmann Machines** with **Neighbourhood model** and get gratifying results.

6 Self Evaluation

In this part, we evaluate the advantages and disadvantages of our project.

6.1 What we have achieved

- Build a database conforming to the low redundancy paradigm
- Implement and optimize many necessary searching functions
- Create a user-friendly website to lower the bar of use
- Feed the database with a suitable amount of data and design a proper data analysis algorithm to satisfy personalized requirements

6.2 Future Improvements

- More information about movies, directors and actors can be added to the database. For example, for actors we may add their nationality and Oscar nominations; for movies, we may add their budget and the classification (PG13, NC17).
- More interaction functions can be added, like "like" a movie or "subscribe" a user. We can also add functions like "want to watch" a movie or "watched" a movie without commenting on it. This information can be utilized in conditional RBM to provide better recommendations (Hinton et al, 2007). Currently, we have the algorithm implemented but the information is unknown.
- In the future, we plan to distinguish the users of our platform. We will allow a common user to upgrade as an administrator, which will reduce the effort to maintain and supervise the comments other users post.
- Currently, the web crawler is implemented using synced requests, which reduced its performance. In the future, we will write the crawler using async requests to speed up the crawling performance and acquire more data for our database.

7 Contribution

Each of us contributed 20% to the whole project.

- 郑时飞
- 朱伯源
- 李易
- 施天昊: Web Crawler, Sorting queries, Server Deployment and Maintenence
- 汪明杰

8 References

Salakhutdinov, R., Mnih, A., & Hinton, G. (2007). Restricted boltzmann machines for collaborative filtering. Proceedings of the 24th International Conference on Machine Learning - ICML '07. https://doi.org/10.1145/1273496.1273596

9 Appendix

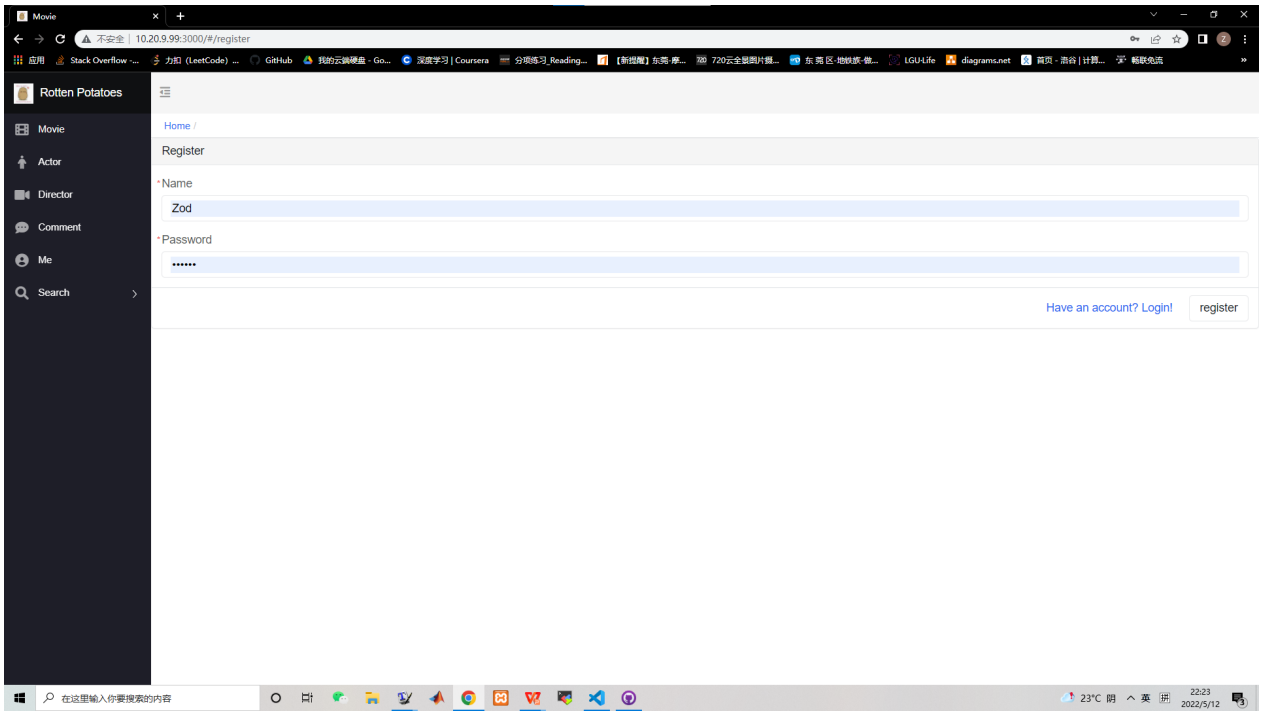

Figure 21: Register page

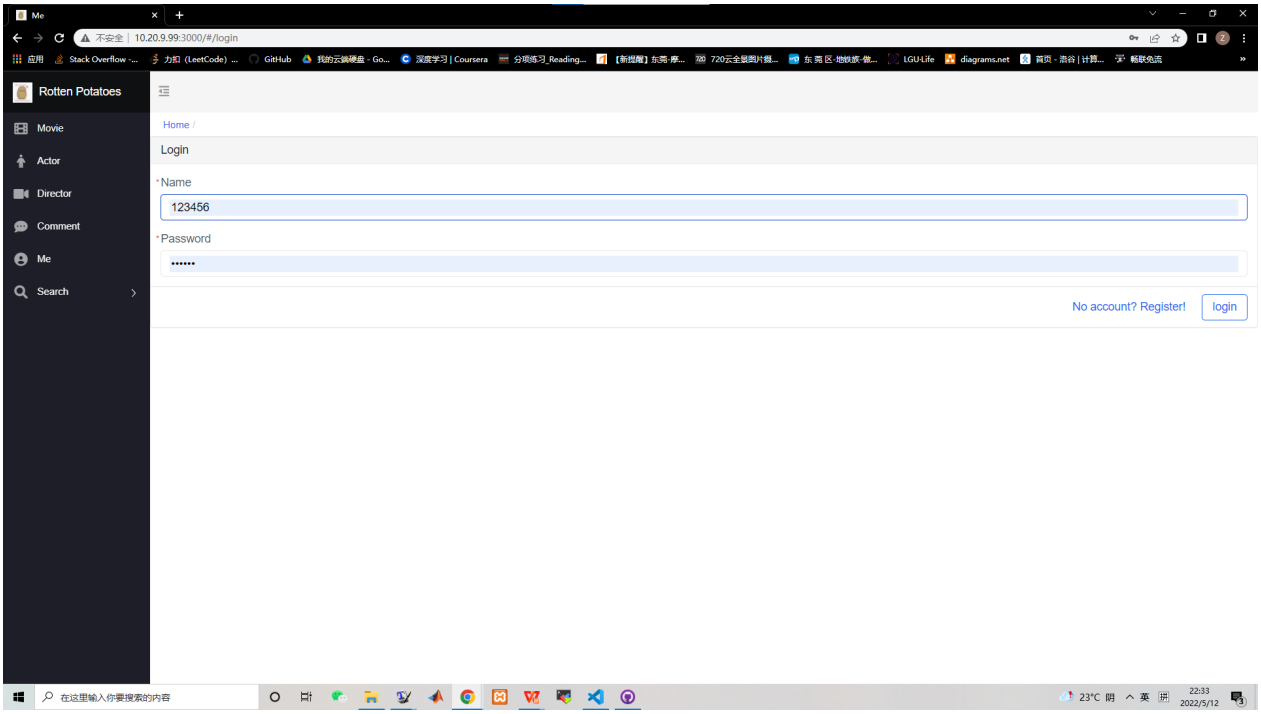

Figure 22: Login page

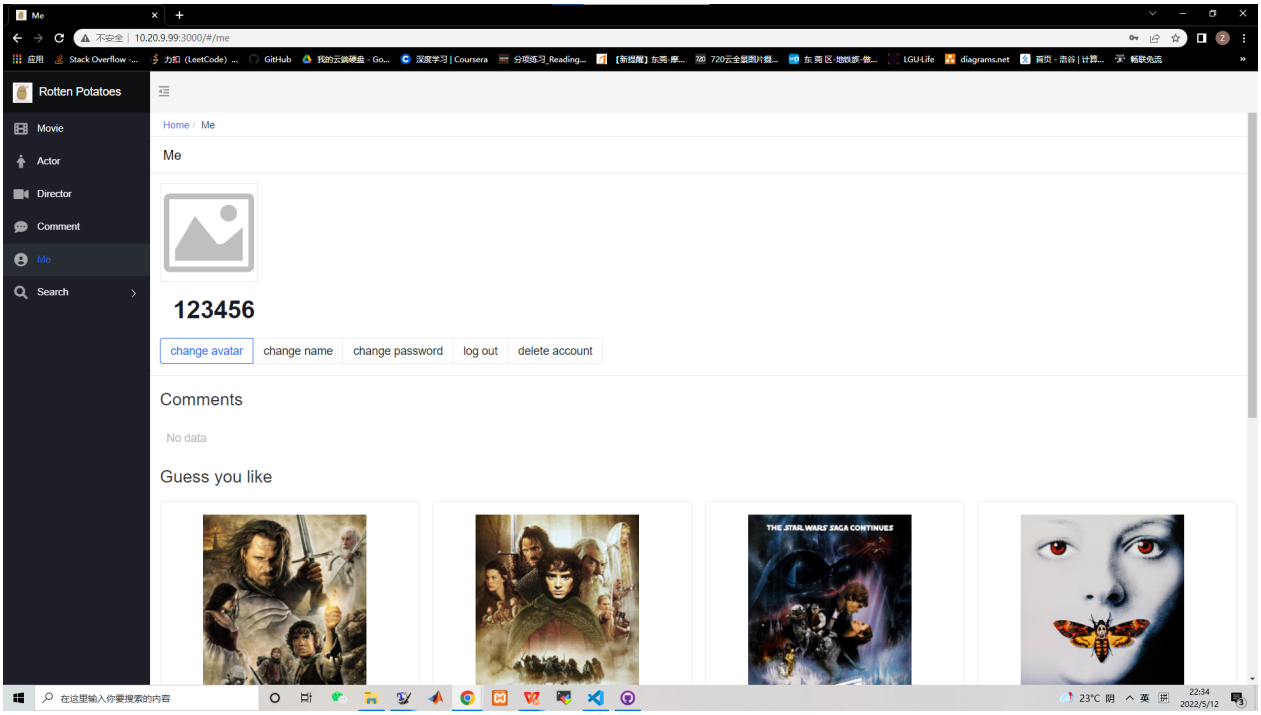

Figure 23: Personal center

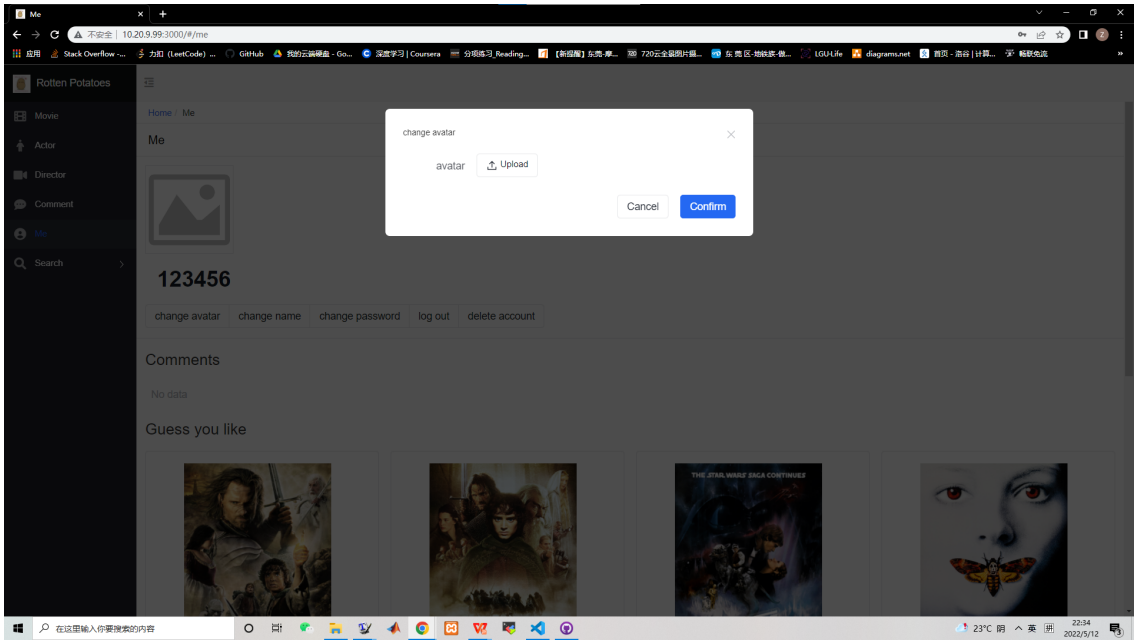

Figure 24: Before chaning avatar

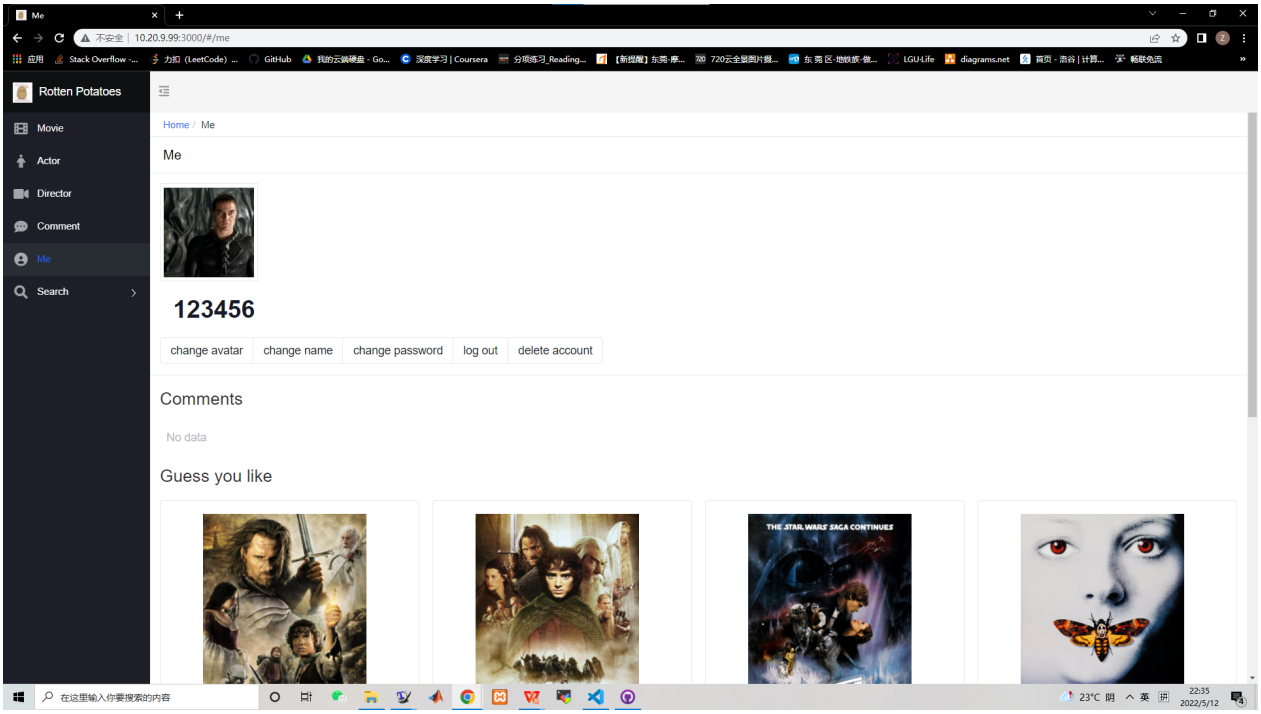

Figure 25: After chaning avatar

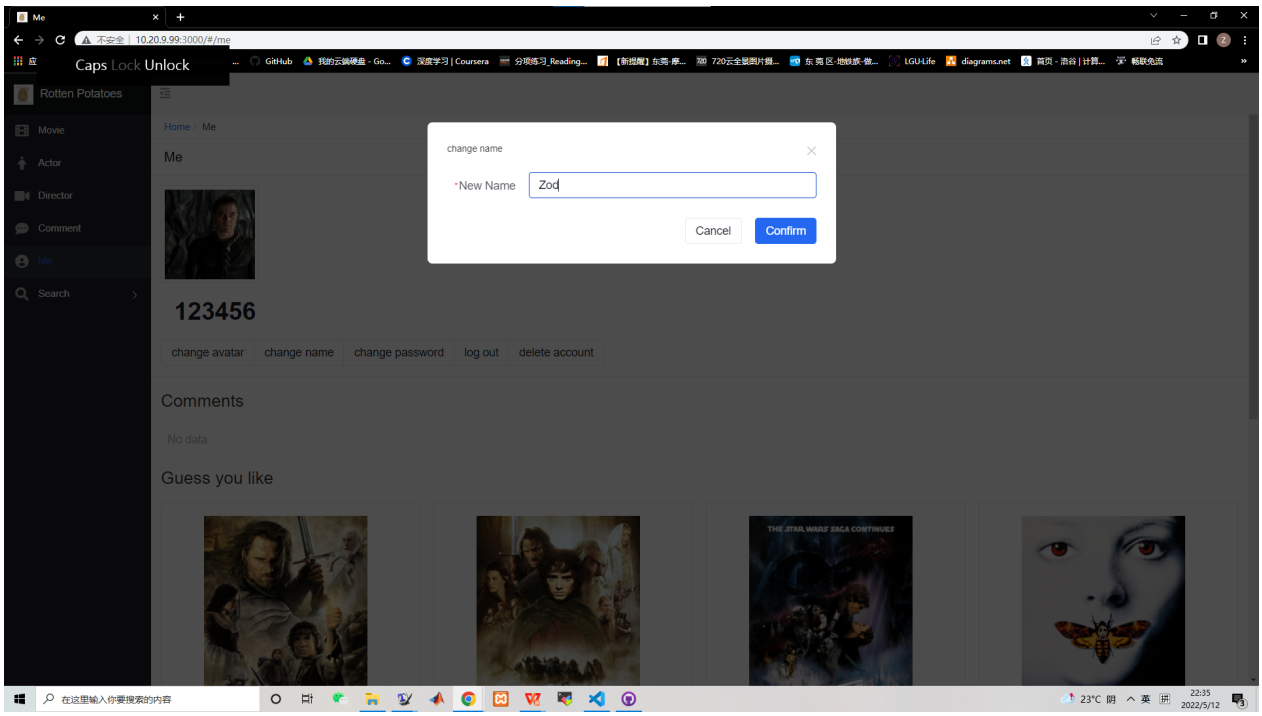

Figure 26: Before chaning name

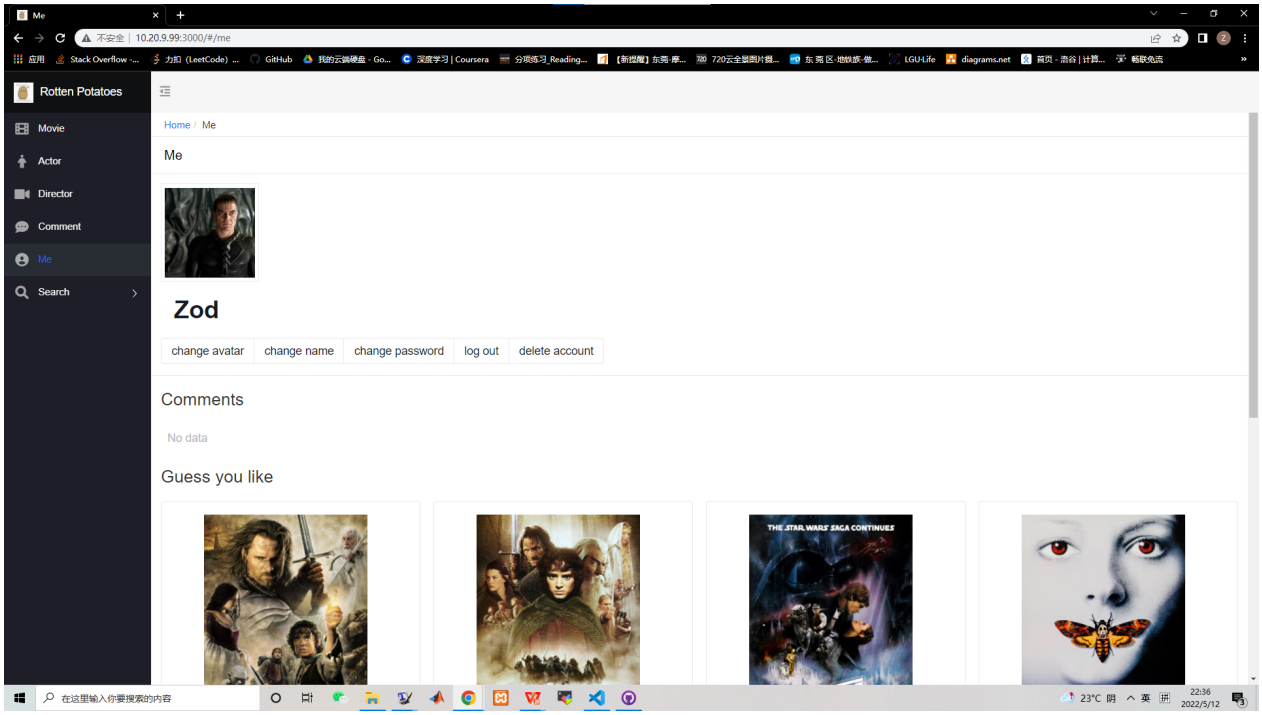

Figure 27: After chaning name

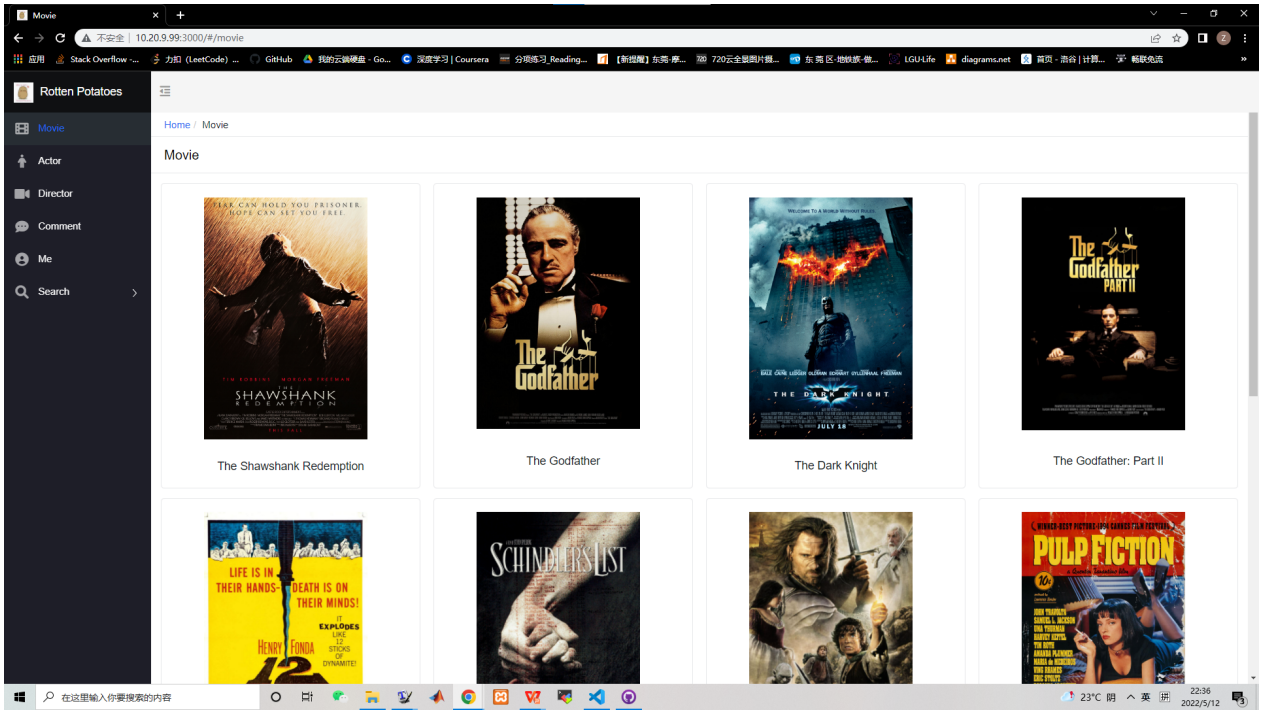

Figure 28: Movie list page

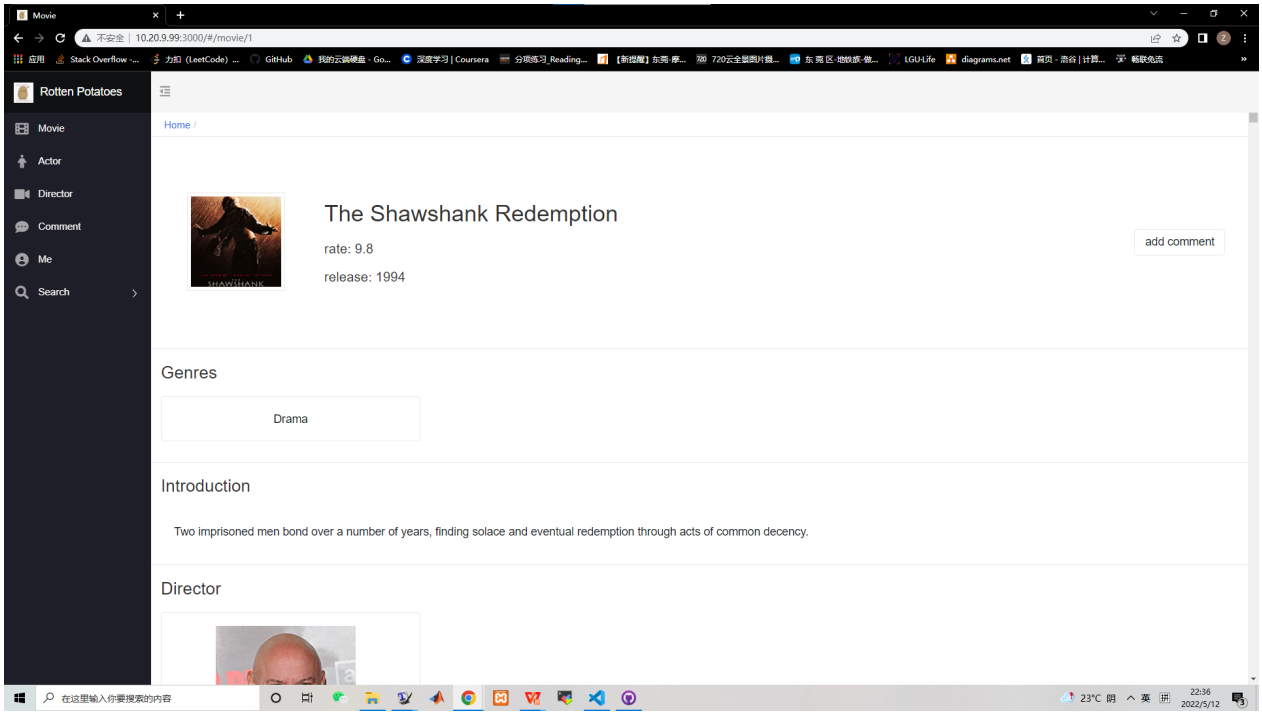

Figure 29: Movie detail page

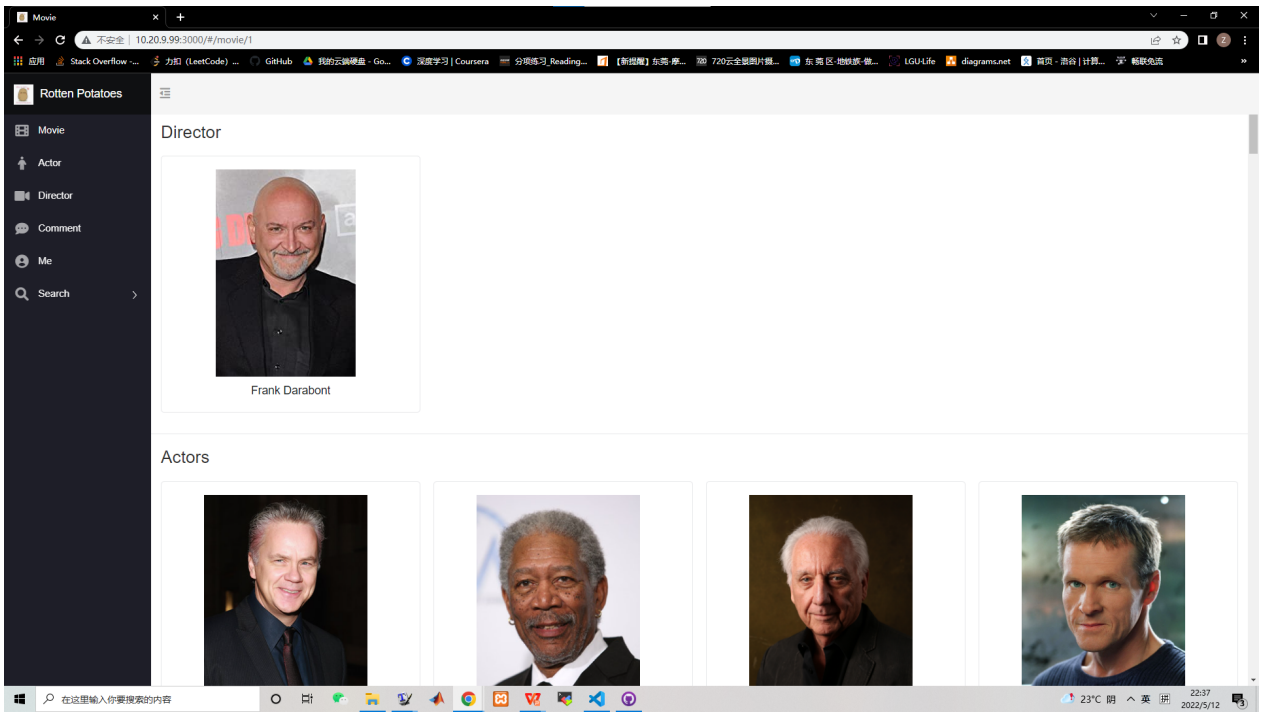

Figure 30: Movie detail page

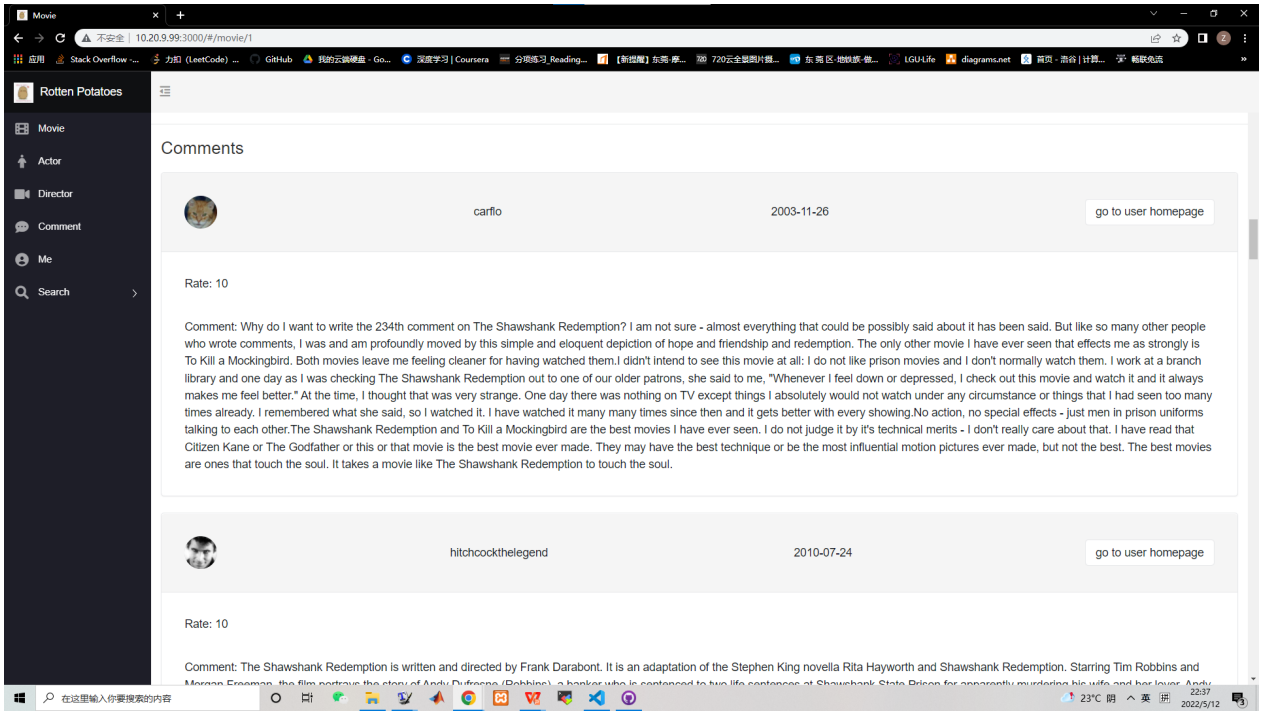

Figure 31: Movie detail page

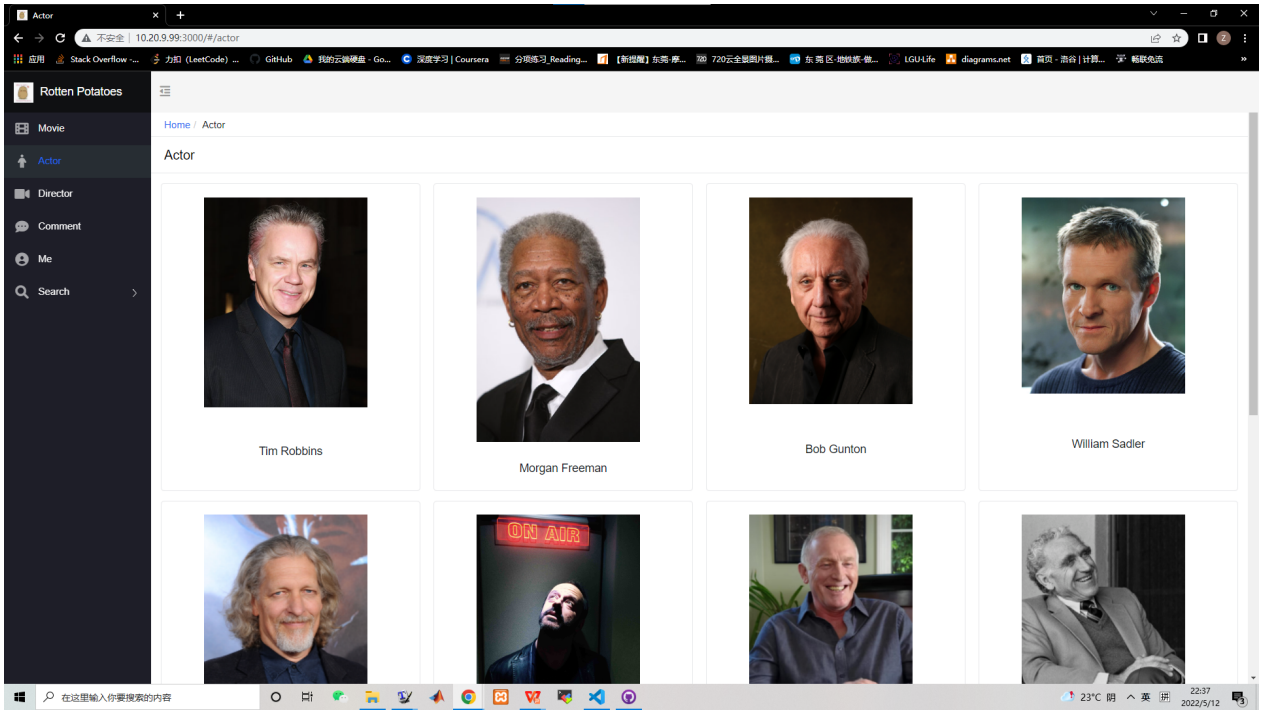

Figure 32: Actor list page

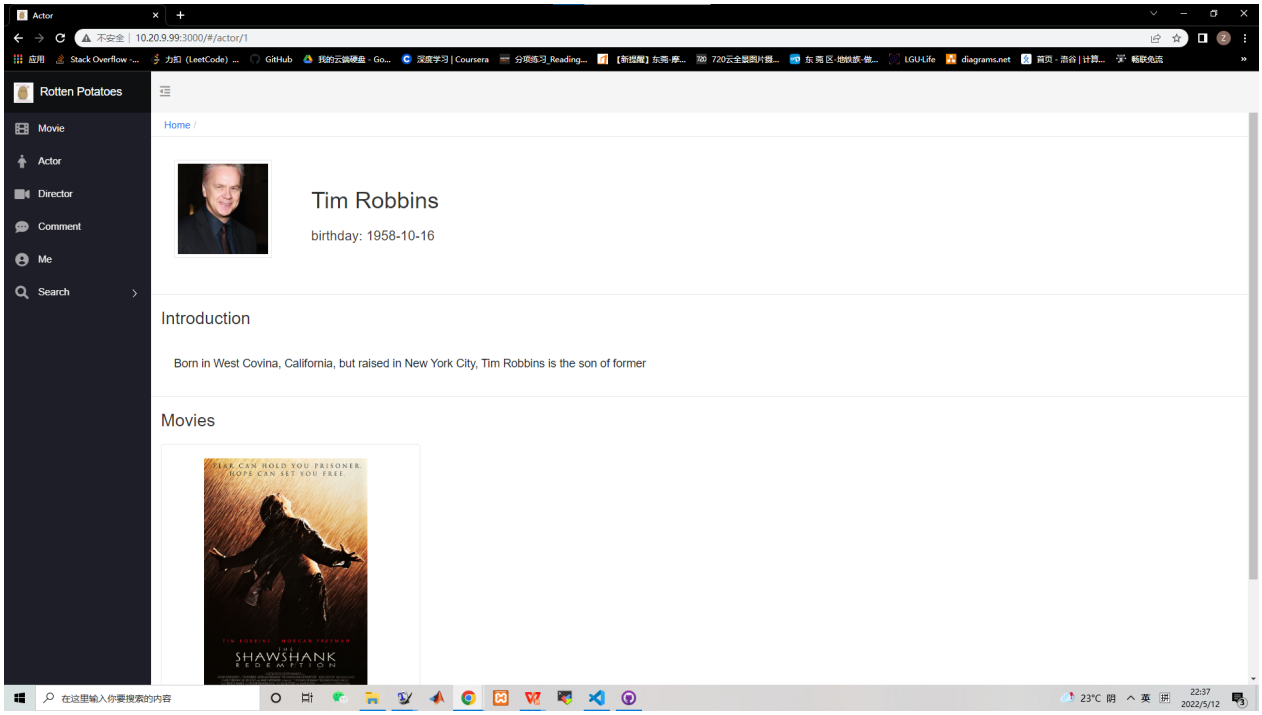

Figure 33: Actor detail page

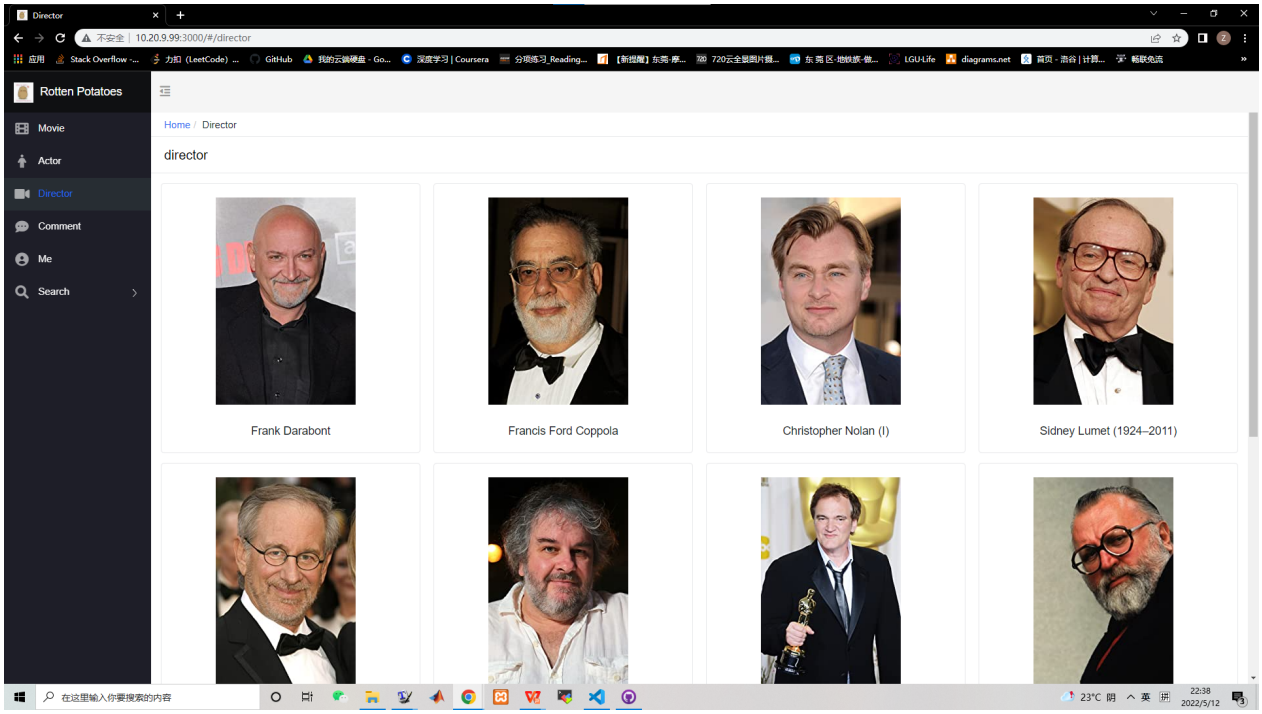

Figure 34: Director list page

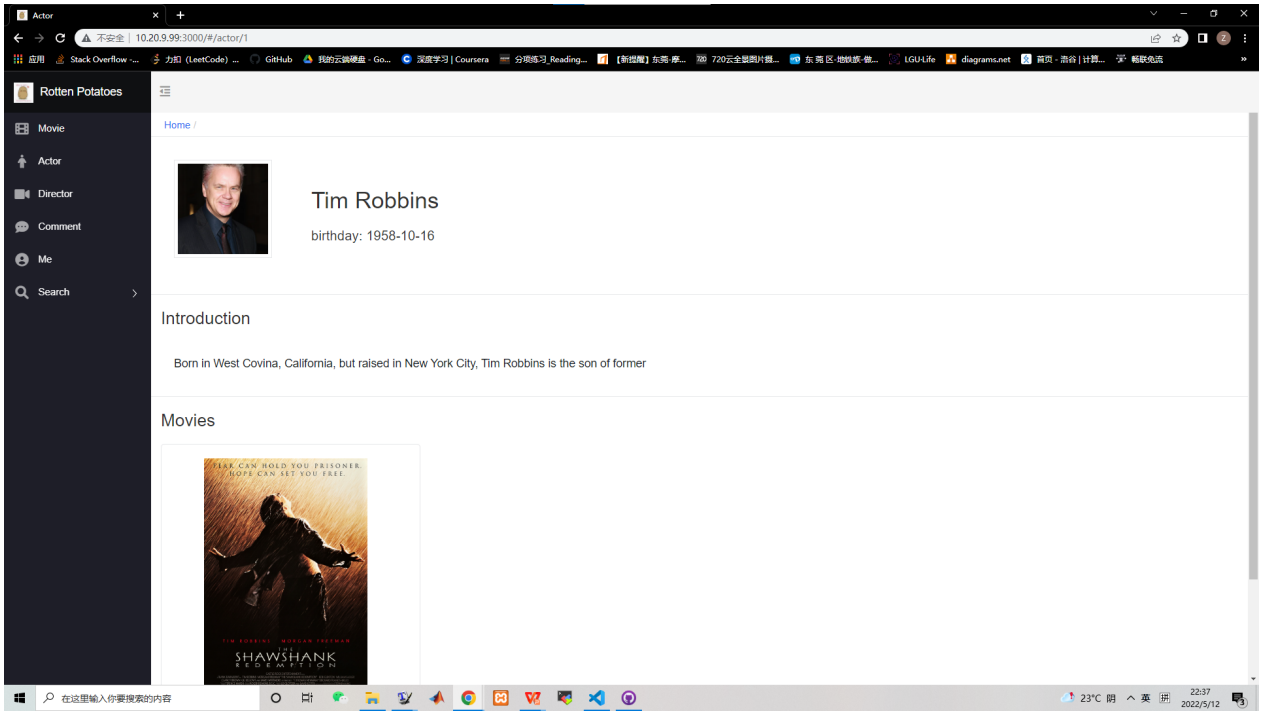

Figure 35: Director detail page

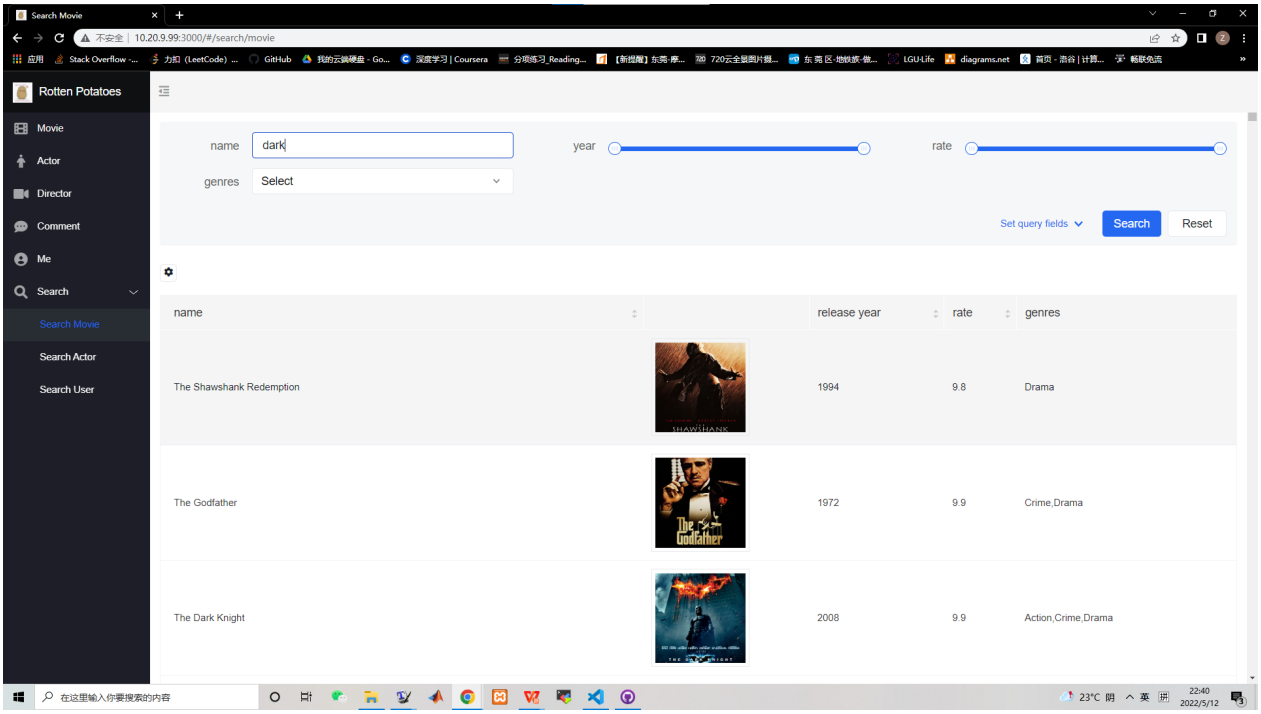

Figure 36: Movie search page

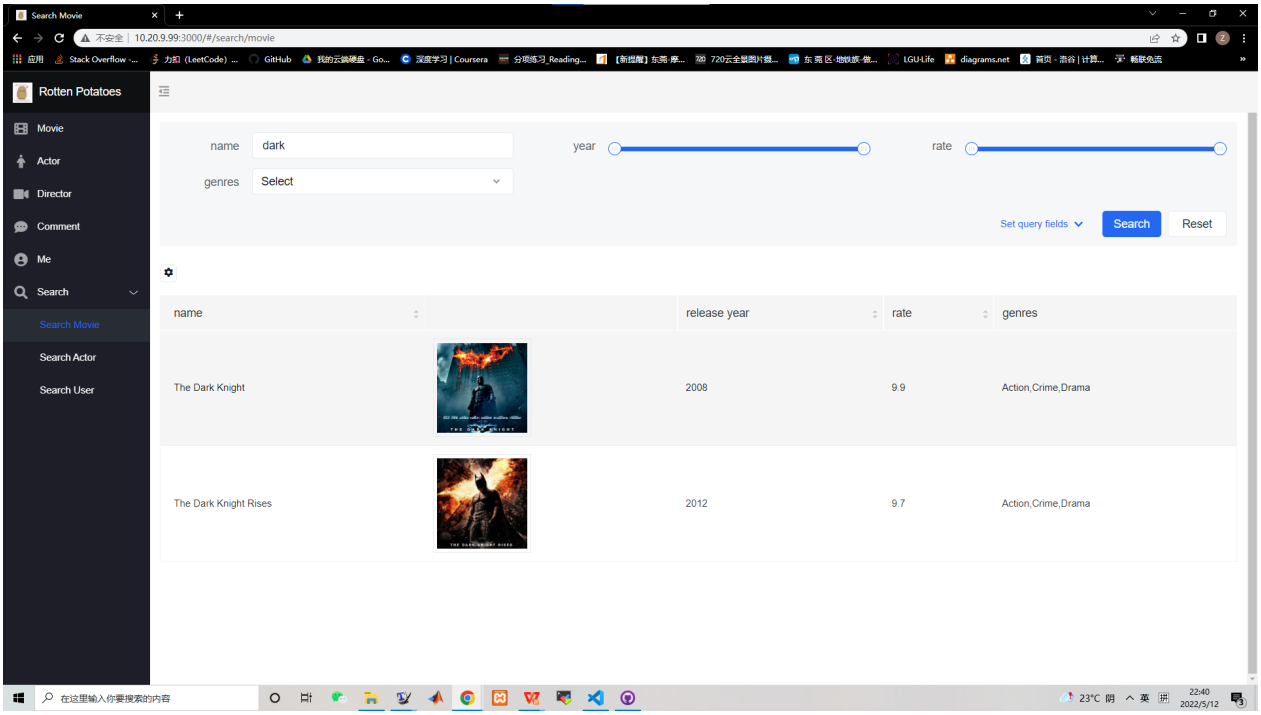

Figure 37: Search movie by name

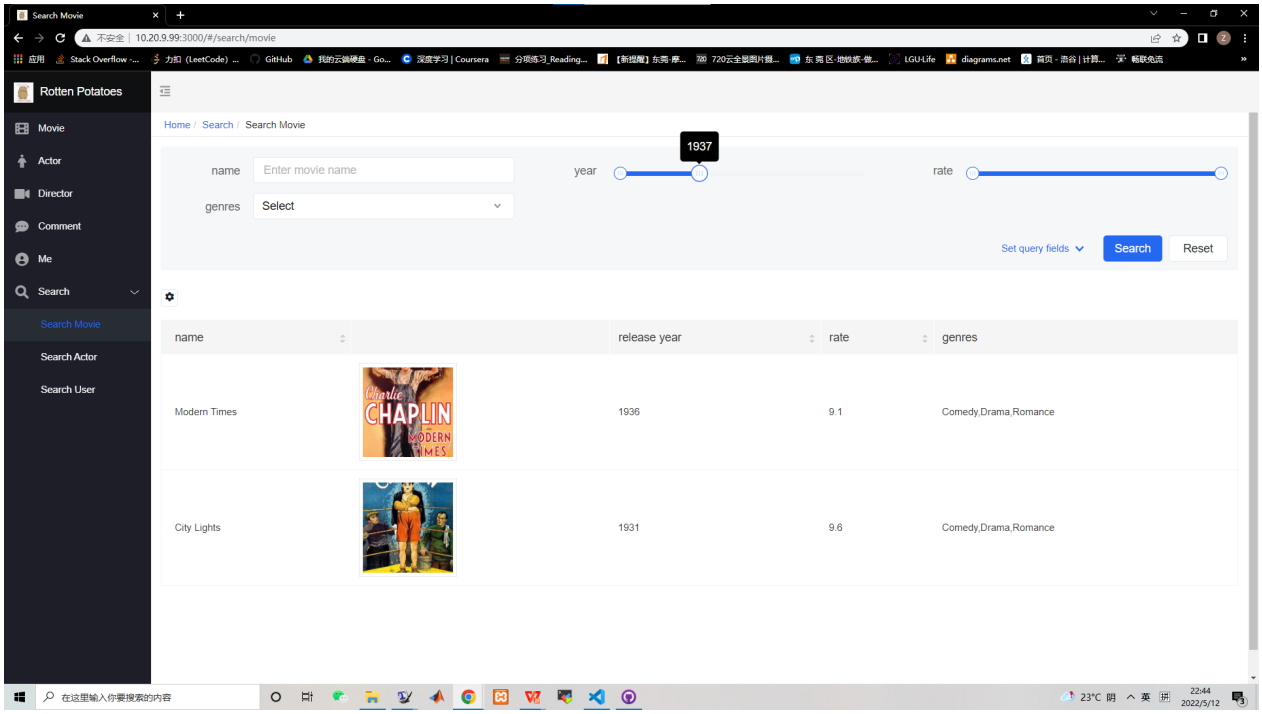

Figure 38: Filter movie by release date

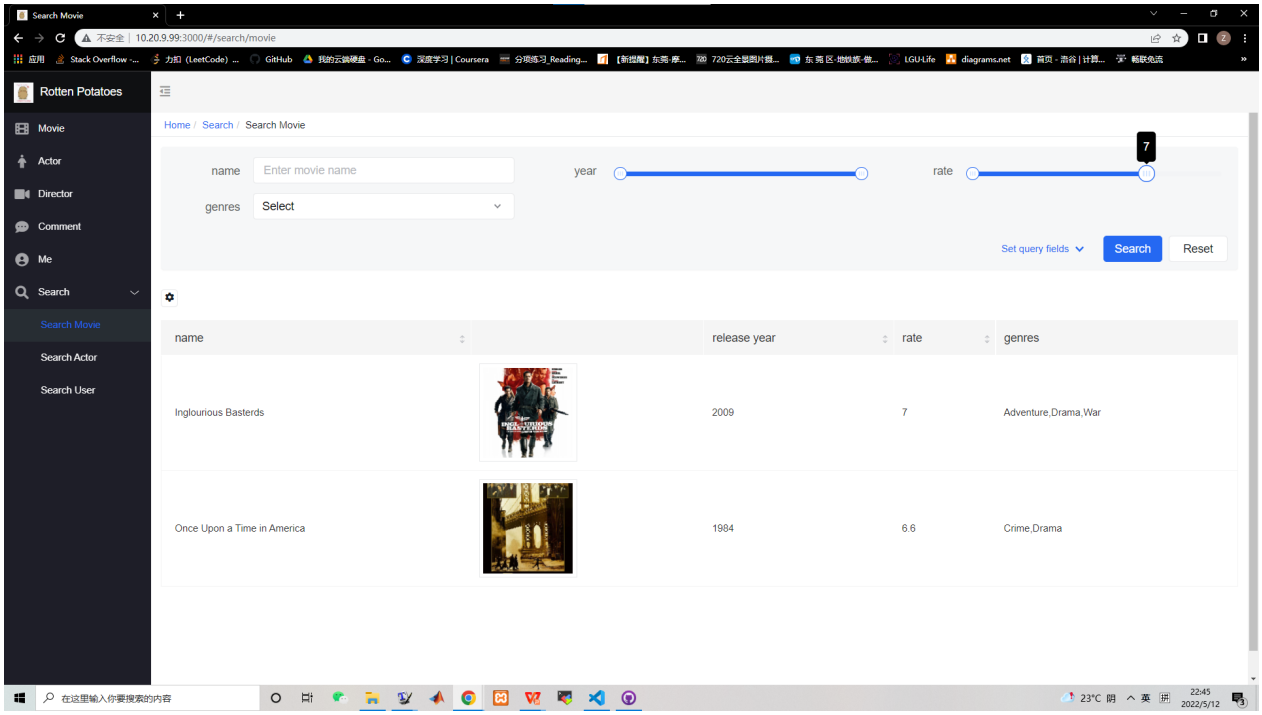

Figure 39: Filter movie by rate

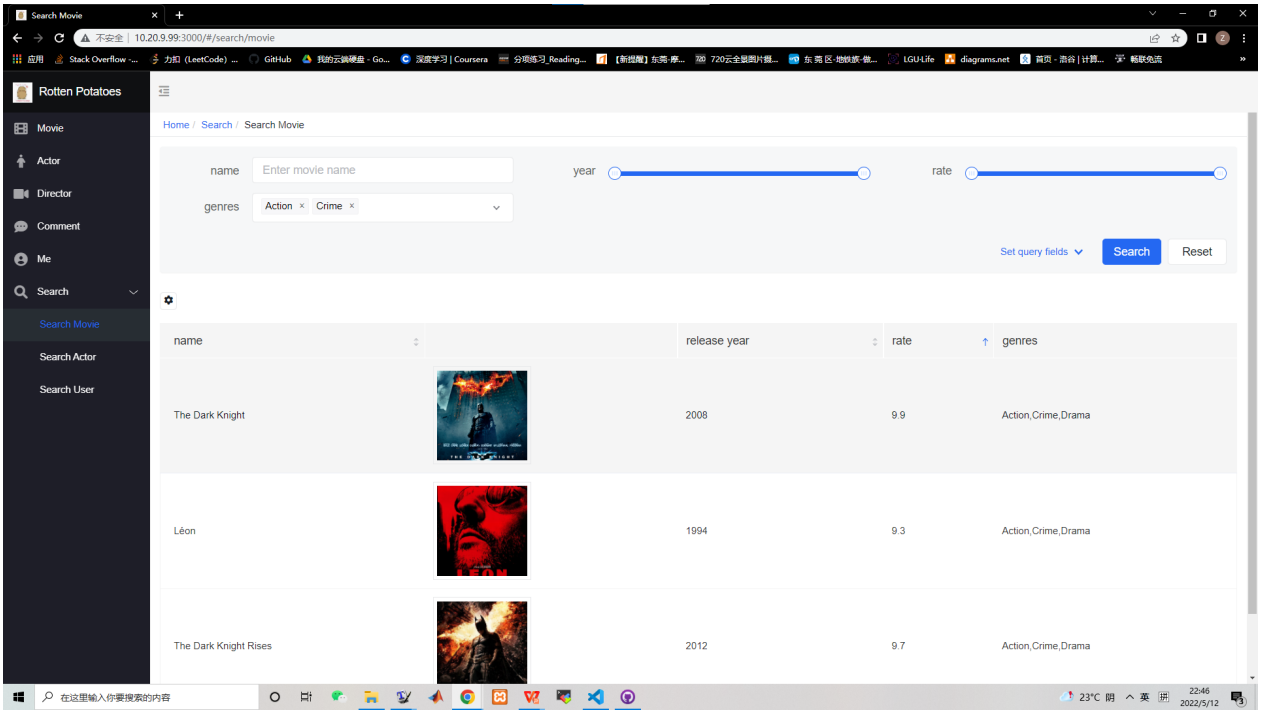

Figure 40: Filter movie by genres

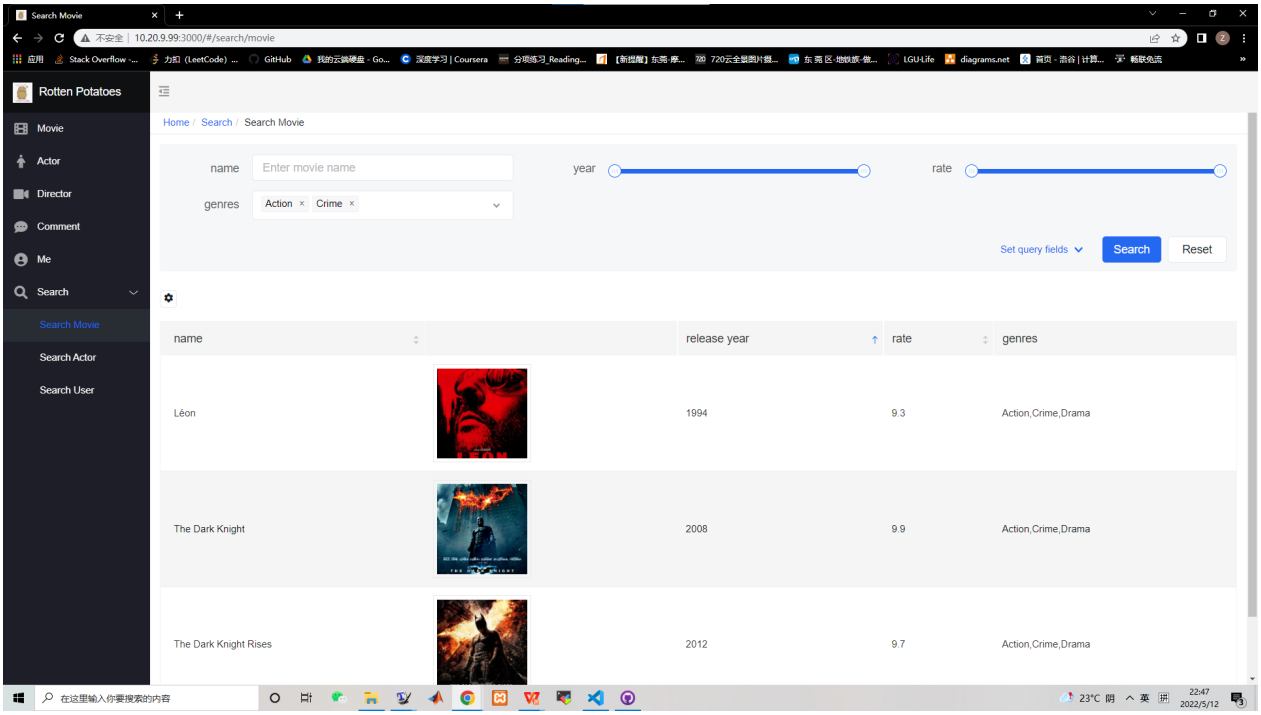

Figure 41: Order movie by release date

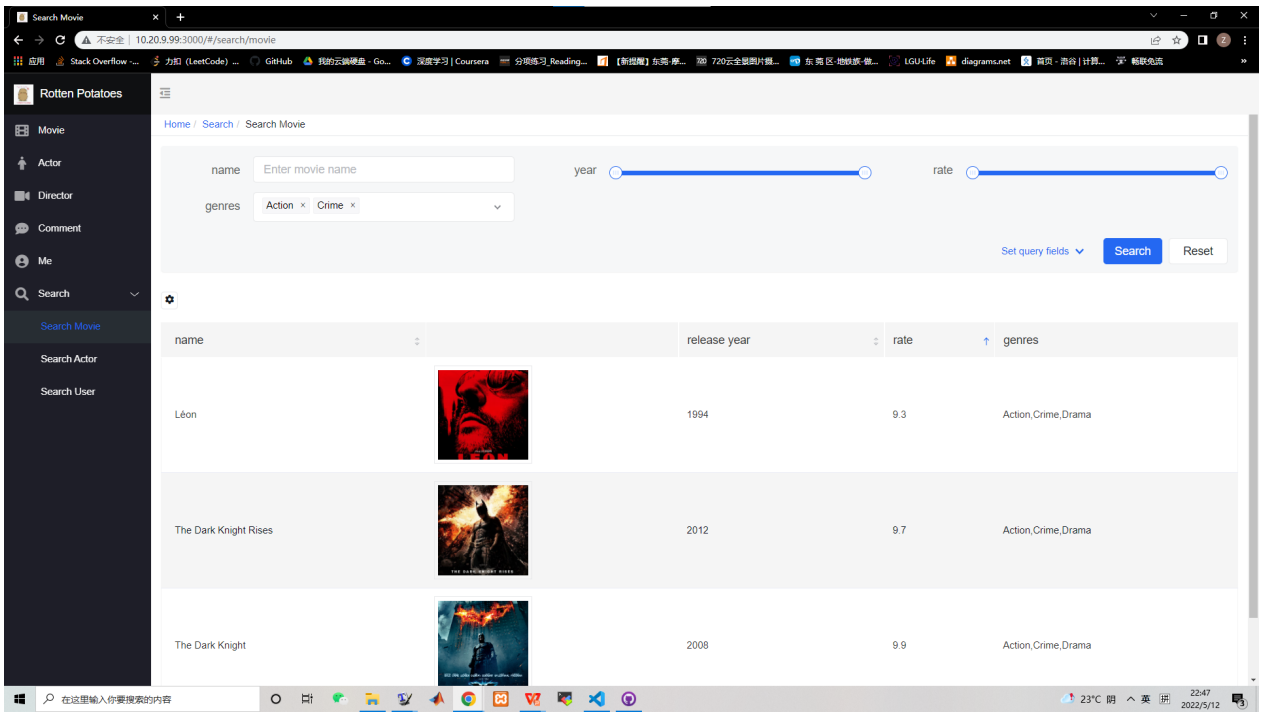

Figure 42: Order movie by rate

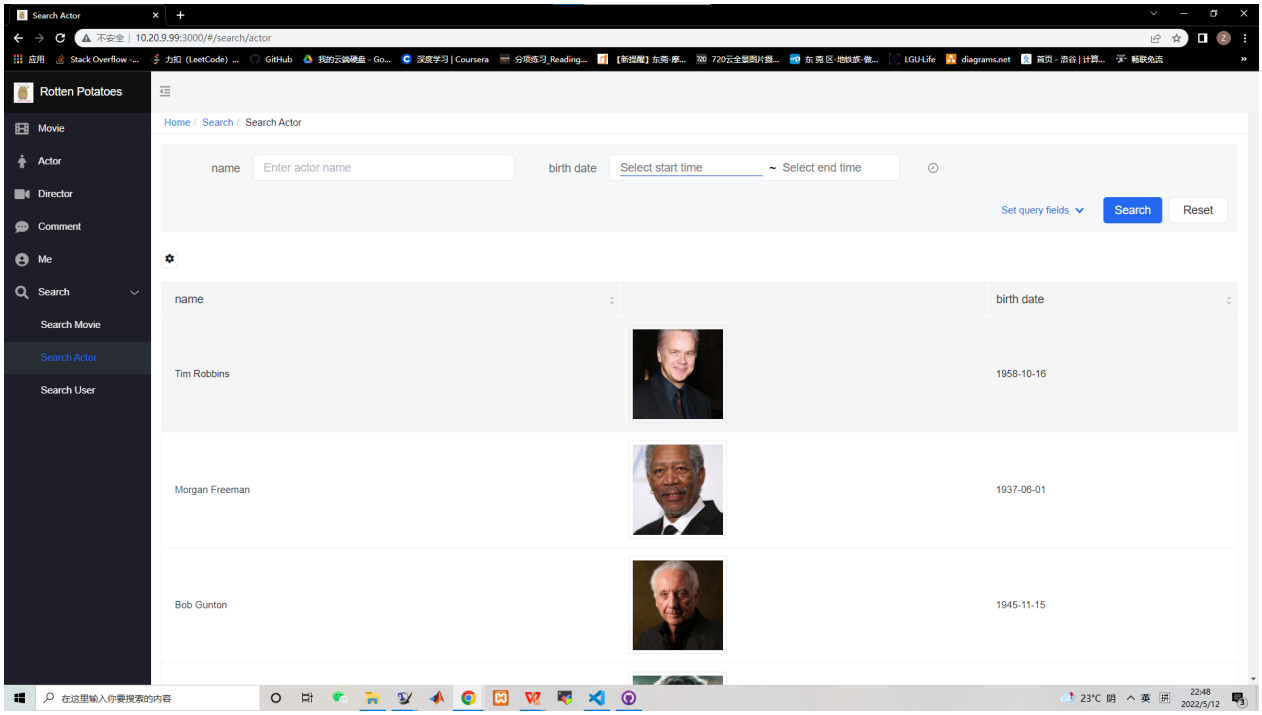

Figure 43: Actor search page

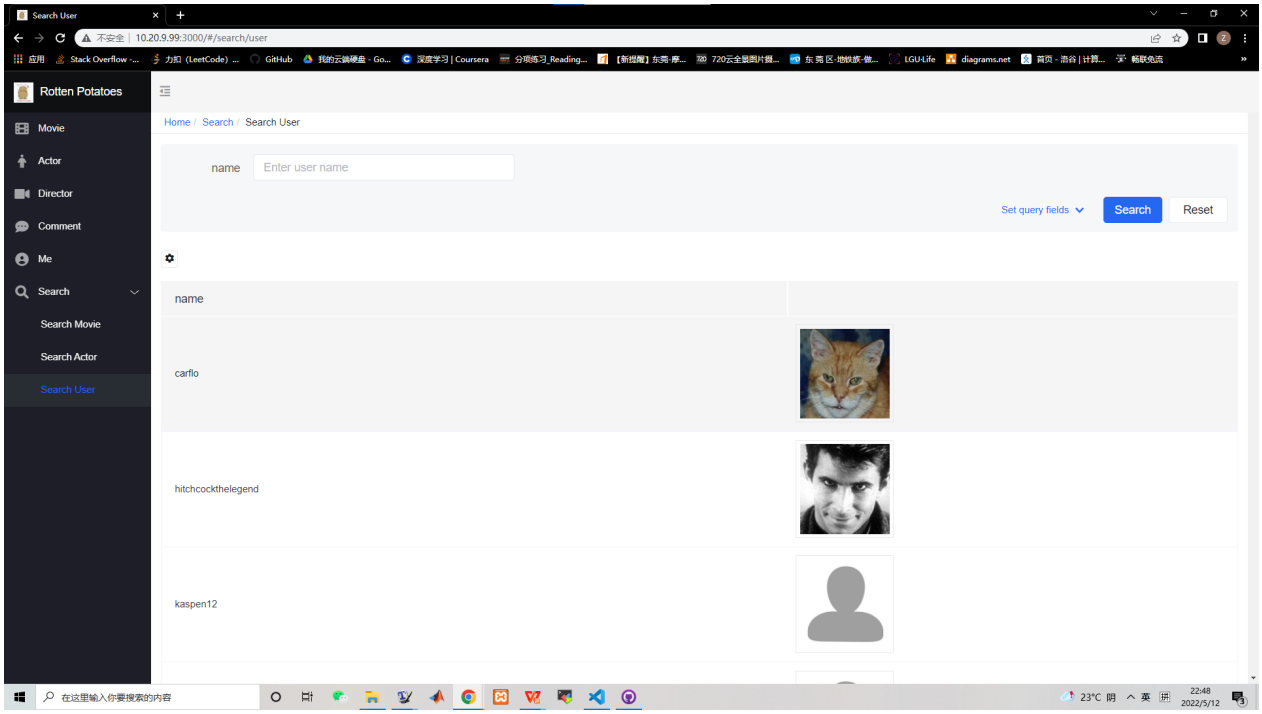

Figure 44: User search page

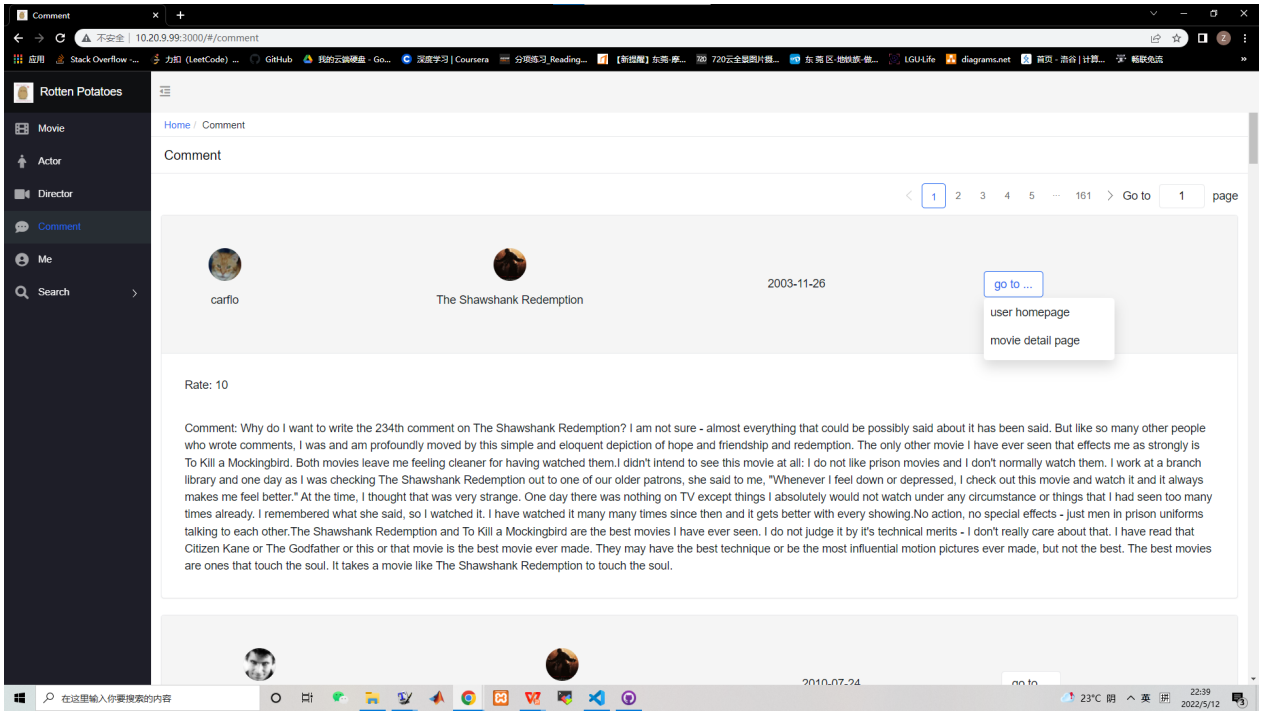

Figure 45: Comment list

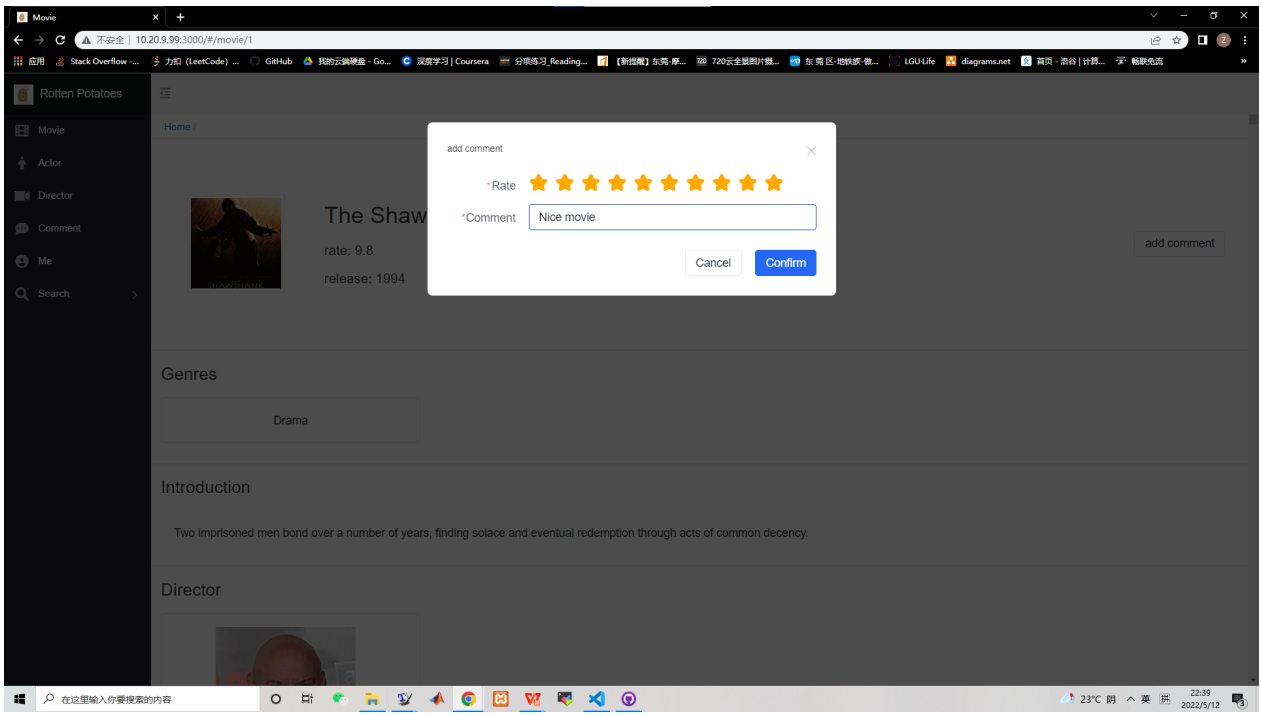

Figure 46: Add comment

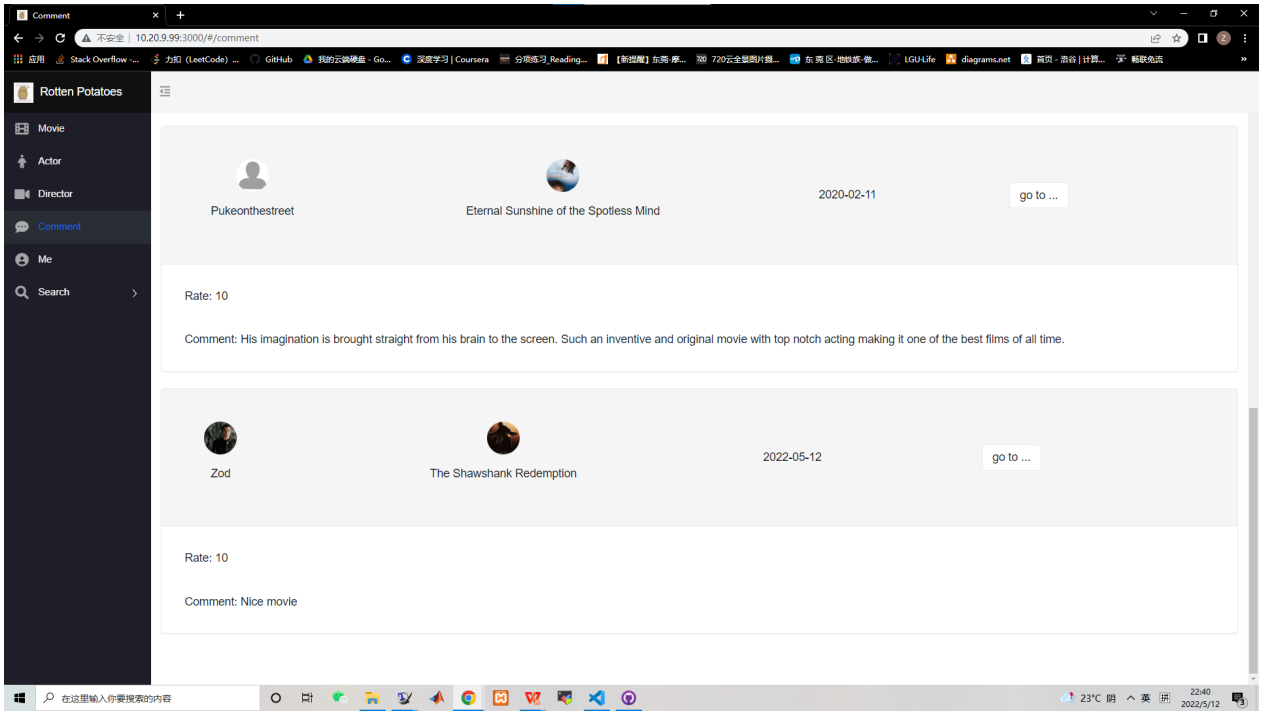

Figure 47: Comment before deleting user account

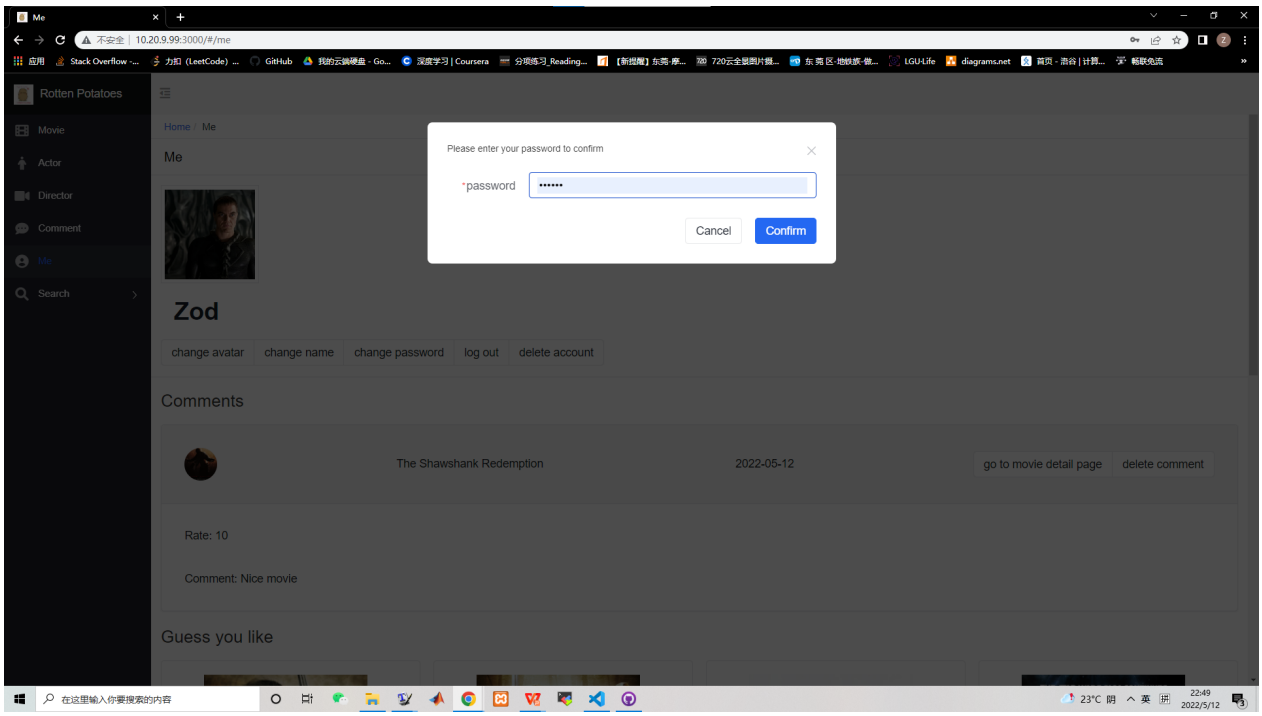

Figure 48: Delete user account

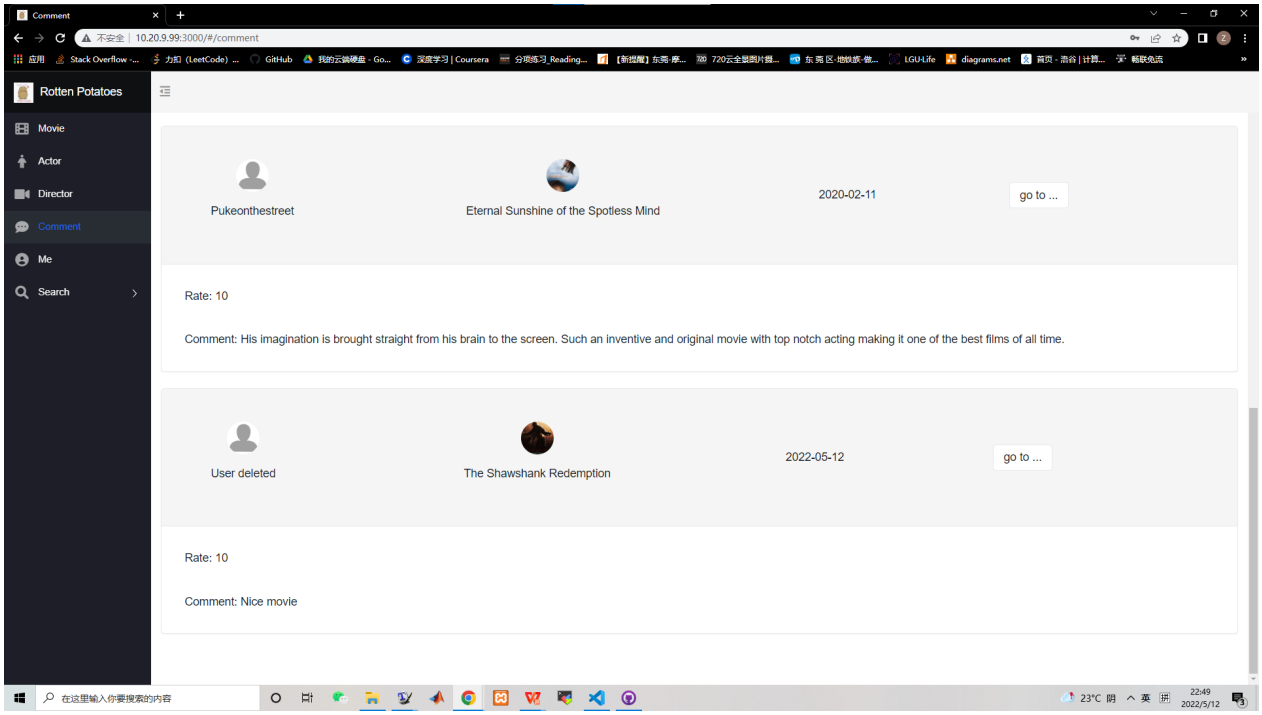

Figure 49: Comment after deleting user account