Carnegie Mellon University in Qatar

Database Applications

15-415 - Spring 2020

Project 2

Out: February 27, 2020

Due: March 22, 2020

1 Project Objectives

You are to create a movie recommendation website, CMUQFlix, which allows users to **(1)** login, **(2)** rate movies, **(3)** form parties with other users, and **(4)** get personalized and party-based movie recommendations. The goal is to learn how to set up a website with a database back-end. In particular, you are to develop two alternative versions of the back-end logic, one using SQL and the other using object-relational mapping (ORM).

2 Data Storage

CMUQFlix uses PostgreSQL to store different types of data as described below:

1. **User data**: For every user, we want to record their *username* (e.g., `coolguy2016`), *password*, and *email address*. Usernames should be unique and passwords cannot be empty.

2. **Movie data**: For the movies, we will use the MovieLens database (from P1), which maintains the following fields for each movie: a unique *id*, its *title*, the *year* of its release, a global average *rating* between 1.0 and 5.0 (or 0.0 if it is unrated), and the total *number of ratings* it has received.

3. **Personalized ratings**: Any user can award any movie a personalized *rating* out of five stars (i.e., an integer between 1 and 5). Therefore, for each user, the system should keep a record of their personalized ratings.

4. **Movie parties**: Any group of users that enjoy watching movies together can form a *movie party*, identified simply by an auto-incrementing integer. The system should record the memberships of users in parties.

3 System Functionalities

The CMUQFlix website consists of different webpages that serve different purposes as described below:

1. **Sign-Up Page**: Before a user can start using your system, they need to sign up by providing a username, password, and email address. None of the fields can be empty. If a username chosen by a user during sign-up already exists, the system should give an error message and prompt for a different username. You may store the password as plain text or using an encrypted hash, and you need not verify the validity of the email address.

2. **Login Page**: A user should be able to log in with the username and password they provided during sign-up. For security reasons, no one should be able to view any page of your website without logging in. Also, once logged in, the user should not need to log in to view other pages. A login session is valid until the user explicitly logs out, unless the user remains inactive for 30 minutes, in which case the session should automatically expire. Users should be able to log out manually at any time. *Hint: In this project we use Django, which provides easy session management. You will simply need to modify the default session timeout period.*

3. **Profile Page**: When a user logs in, they should see their profile information, which includes their username and email address.

4. **Movies Page**: A user should be able to navigate to a webpage that lists all the movies in the database divided into two sets: **(a)** *rated movies* (i.e., movies that the user has already rated) and **(b)** *unrated movies* (i.e., movies that the user has not yet rated). Both sets should be sorted alphabetically by title. Moreover, this page should allow the user to award any movie a personalized rating between 1 and 5 stars (where 1: *Awful!*, 2: *Boring*, 3: *Nice*, 4: *Good*, 5: *Epic!*). More specifically:

   (a) Each entry in the set of rated movies should be accompanied by both its global rating and the user's personalized rating, along with an option for the user to modify or delete their personalized rating.

   (b) Each entry in the set of unrated movies should be accompanied by the global rating of that movie, along with an option for the user to award it a personalized rating.

   Whenever a user adds, modifies, or deletes their personalized rating for some movie, the system should reflect this change in the movies table by updating that movie's global average rating and total number of ratings. For example, say we have a movie titled *Alien* with a global average rating of 3.25 over 20 total ratings. Now, say a user named Alice modifies their (existing) personalized rating for *Alien* from 3 to 4 stars. Consequently, the global average rating of *Alien* should be updated to $(3.25 \times 20 + (4 - 3)) \div 20 = 3.3$, while its total number of ratings stays unchanged since this counts as a modification and not a new rating.

5. **Personalized Recommendations Page**: A user should be able to navigate to a webpage that generates up to five personalized movie recommendations based on the following criteria:

   (a) We say that a user *liked* a movie if they awarded it 3 stars or higher.

   (b) Personalized recommendations should not include any movies that the user has already rated.

   (c) If the user has not yet *liked* any movie that at least one other user has also *liked*, then display the top five movie titles with the highest global average ratings, along with their respective ratings.

   (d) If the user has *liked* at least one movie in common with one or more other users, then these users implicitly form the user's *movie clan*. As such, display the top five movies with the highest average ***clan ratings*** (derived from the ratings awarded by the clan's members only), along with their respective clan ratings.

   (e) The recommendations should be sorted in descending order by their ratings. Ties should be broken by sorting the titles alphabetically.

Figure 1 demonstrates this logic with an example. To make recommendations for the user Alice, we first find her *movie clan*, which is the set of users who *liked* at least one movie in common with her. Thus, since *Alien* is the only movie that Alice *liked*, her clan includes Bilal and Dana, who also *liked Alien*. We then report the five most popular movies *among Alice's clan* based on their average ratings (i.e., as awarded by the members of this clan, not globally). Thus, the movies to recommend to Alice would be *Logan* and *Titanic* (in that order). Of course, we do not return *Alien* or *Dunkirk*, since Alice has rated them already.

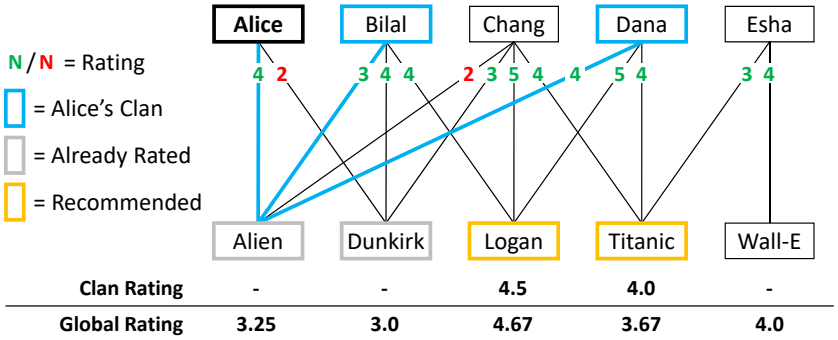

Figure 1: Example of clan-based movie recommendations for user 'Alice'.

6. **Movie Party Membership Page**: This page should allow the user to participate in movie parties with other users by creating, joining, or leaving parties. To facilitate this:

   - The page should list all the parties in the database divided into two sets: **(a)** *joined parties* and **(b)** *other parties* (based on the user's current memberships).
   - Each listed entry should link to its corresponding *Movie Party Recommendations* page (see below) and be accompanied by its size (i.e., the number of members in that party) and an option for the user to leave or join that party (depending on their current membership in that party).
   - A party can have a maximum of **10** members at any given time (its size may change over time as members join or leave). As such, the user should not be allowed to join a party that is already full.
   - The user should be able to create a new party. Upon creating a party, the user should join it automatically, thus becoming its first member.

7. **Movie Party Recommendations Page**: The user should be able to navigate to this page from the Movie Party Membership page (see above) in order to see the list of members and movie recommendations associated with a particular movie party. The page should show:

   - The members of the party listed using their usernames sorted in ascending order.
   - The top five most popular movies among the party's members (i.e., chosen from the set of movies that previously received a rating from one or more of the party's members). Sort the movies by their average ratings *among the party's members* (i.e., derived from the ratings awarded by the party's members only) in descending order, with ties broken by sorting the titles alphabetically. Each entry in this list should be accompanied by its average rating among the party's members and its global average rating.

Figure 2 demonstrates this logic with an example. To make recommendations for a movie party that has two members, Alice and Bilal, we first narrow down the pool of movies to the ones that either (or both) of them have previously rated, namely, *Alien*, *Dunkirk*, and *Logan*. We then report the most highly rated movies (at most five) in this list based on their average ratings calculated from Alice's and Bilal's ratings only. Thus, the movies to recommend to this party would be *Logan*, *Alien*, and *Dunkirk* (in that order).

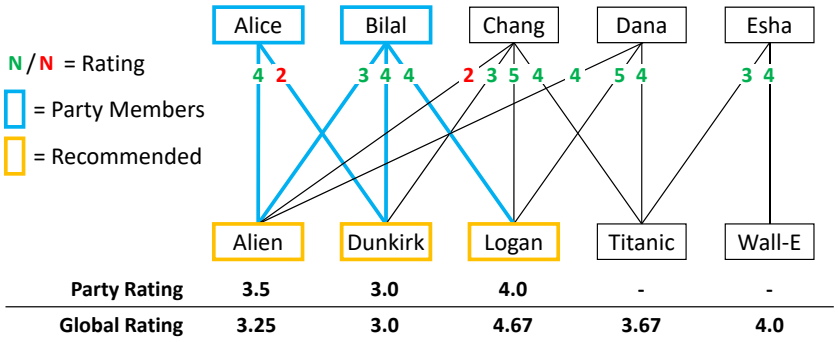

Figure 2: Example of party-based movie recommendations for a movie party comprising users 'Alice' and 'Bilal'.

8. **Reporting Page**: The system should print a summary of user activity in a report page. The page should report the following system-wide statistics:

   - The total number of movies.
   - The total number of registered users.
   - The total number of ratings.
   - The average number of ratings per movie.
   - The average number of ratings per user.
   - The list of *avid users*, namely, the top 10 users with the most number of ratings. The users should be listed using their usernames and sorted by their number of ratings in descending order, with ties broken by sorting the usernames alphabetically.
   - The total number of parties.
   - The average number of users per party.
   - The list of *popular users*, namely, the top 10 users with the most number of party memberships. The users should be listed using their usernames and sorted by their number of memberships in descending order, with ties broken by sorting the usernames alphabetically.

4 System Architecture

CMUQFlix adopts a three-tier architecture with a front-end, middle tier, and back-end as shown in Figure 3.

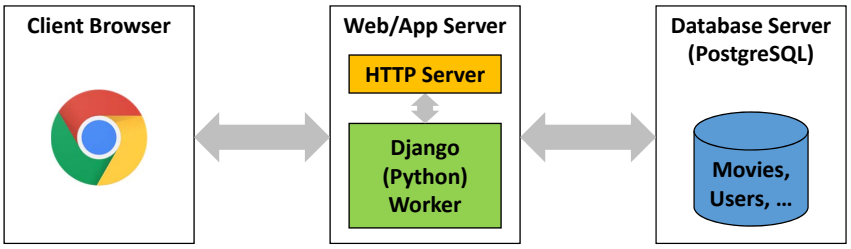

Figure 3: Three-tier architecture of CMUQFlix.

The *front-end* is a client browser that can display the various webpages of CMUQFlix. The *back-end* is a PostgreSQL database housing the movies and user relations (among others). The middle tier is the middle-man that connects the front-end with the back-end to make a whole system.

In particular, the middle tier consists of a *web server* (e.g., Apache) that, as the name implies, serves clients' HTTP requests for HTML webpages. An HTML webpage served by the web server is either a *static* page whose HTML content has been predefined at the time of website creation, or a *dynamic* page whose HTML content is produced by a *Django worker* process running on the web server. Django is a Python-based web application framework that facilitates the encoding of our application logic (including any interactions with a back-end database) and dynamically generates webpages based on this logic. Therefore, the web server houses a combination of our application's static HTML structure/content and our application's dynamic Python code.

5 Project Milestones

You are to develop **two** alternative implementations of the CMUQFlix web application using Django:

1. **SQL.** All your application logic that deals with querying or updating the database must be written & executed as *raw* SQL. In other words, you may not use Django's ORM features (see below). For more information on how to program using raw SQL in Django see [1].
2. **ORM.** The structure and implementation of your application should follow Django's *Model-View-Template (MVT)* development pattern and object-relational mapping (ORM) data model. This approach precludes the need for writing raw SQL by providing you with a purely object-oriented data model. Django will automatically generate and execute the appropriate SQL commands as and when required. For more information on how to program using Django's MVT pattern and ORM model see [2].

An important objective of the project is to appreciate the kind of tedious complexities handled for us transparently by any ORM framework. This should become evident upon contrasting the two aforementioned implementations. Note that we shall cover both of the approaches over the course of multiple recitations.

[1] `https://docs.djangoproject.com/en/2.0/topics/db/sql/#executing-custom-sql-directly`
[2] `https://docs.djangoproject.com/en/2.0/intro/tutorial01/`

You need to achieve the following milestones in order to complete the project:

1. Install PostgreSQL [3] and Python [4] on your machine.
2. Additionally, we highly recommend that you use PyCharm Professional IDE [5] (free for academics) for Django/Python development. You can refer to the recitation notes provided by your TA for further details on the installation procedure.
3. The Django framework includes a lightweight web server for initial prototyping and testing. If you prefer, however, you may install and use any other production web server, such as NGINX [6].
4. For your **SQL** implementation:
   (a) Download the `cmuqflix_sql` folder from the course website [7]. This will be your working directory.
   (b) Create a new database called `cmuqflix_sql` in PostgreSQL.
   (c) The `cmuqflix_sql` folder contains an SQL script called `movies.sql`, which creates a `movies` table in the database and populates it with the MovieLens dataset. Run this script using the PostgreSQL command-line client: `psql -U postgres -f movies.sql`
   (d) Using the Django management console (`manage.py`), run `migrate` to push Django's internal schema to the database.
   (e) Manually create all the necessary tables in the database for implementing the functionalities listed in Section 3.
   (f) The `cmuqflix_sql` folder provides a skeleton Django web application that you can build upon. Implement the necessary Django *views* (business logic) and *templates* (user interface) for achieving the required functionalities.
5. For your **ORM** implementation:
   (a) Download the `cmuqflix_orm` folder from the course website. This will be your working directory.
   (b) Create a new database called `cmuqflix_orm` in PostgreSQL.
   (c) Using the Django management console (`manage.py`), run `migrate` to push Django's internal schema to the database.
   (d) The `cmuqflix_orm` folder provides a skeleton Django web application that you can build upon. This includes a `Movies` *model* (or class). Using the Django management console (`manage.py`), run `migrate` to push this model to the database (this creates the movies table) and then run `loaddata` to load the provided movie fixtures (i.e., records) in `fixtures/movies.json` (this populates the movies table).
   (e) Develop all the models, views, and templates necessary for implementing the functionalities listed in Section 3. Note that every time you create or modify a model, you must run `makemigrations` followed by `migrate` to push it to the database (using `manage.py`).

[3] `https://www.postgresql.org/`
[4] `https://www.python.org/`
[5] `https://www.jetbrains.com/pycharm/download/`
[6] `https://www.nginx.com/`
[7] `https://web2.qatar.cmu.edu/~mhhammou/15415-s18/projects/P2_Archive.tgz`

6 Final Deliverable

In an archive (named `P2_<AndrewID>.zip`) add your modified `cmuqflix_sql` and `cmuqflix_orm` folders. In `cmuqflix_sql/scripts`, also include a script (named `create.sql`) containing all the SQL statements required to create the tables, views, and indexes used by your SQL implementation. In addition, if you wrote any SQL scripts or JSON fixtures for populating your database with test data, include them in `cmuqflix_sql/scripts` and `cmuqflix_orm/fixtures`, respectively.

7 Getting Help

You can get help by visiting the professor and the TA during their office hours or by appointment. You can also post your questions on Piazza [8].

8 Late Policy

- If you hand in on time, there is no penalty.
- 0-24 hours late = 25% penalty.
- 24-48 hours late = 50% penalty.
- More than 48 hours late = you lose all the points for this project.

Note: You can use your grace-days quota. For details about the quota, please refer to the course syllabus.

[8] `http://piazza.com/qatar.cmu/spring2018/15415/home`