SEB – Protocol: Christoph Leopold if22b234

Database Structure:

The SportsExerciseBattle system is designed to facilitate and record push-up tournaments among registered users. The system aims to provide a platform where users can register, participate in tournaments, and track their progress over time. The database structure supports the core functionalities.

The database consists of two primary tables:

- **users**: This table is designed to manage user-related information. It includes the following columns:
  - **username** (Primary Key): A unique identifier for each user, facilitating user login and identification.
  - **password**: Stores passwords for user authentication, ensuring security.
  - **elo**: Represents the Elo rating of a user
  - **bio**: A short biography allowing users to share a brief personal description.
  - **image**: can be used to store location of image, but in the integration tests it is just a string
  - **name**: The real name of the user, used for personalization and display purposes.
- **push_up_records**: This table captures the details of each push-up session by the users:
  - **record_id** (Primary Key): A unique identifier for each record, ensuring each session can be uniquely identified and referenced.
  - **fk_user_id** (Foreign Key): Links to the username in the users table, associating each record with a specific user (should have named it fk_username but did not want to change it afterwards)
  - **count**: The number of push-ups completed in a session
  - **duration**: The time taken to complete the push-ups,
  - **date_time**: The date and time when the session was recorded

Design:

A HTTP Server listens for connections and opens a new thread for each client.

There several requests that a client can make:

- register user
    - register with username and password
- login user
    - login user via username and password
- view profile
    - view name, bio, image
- edit profile
    - edit name, bio, image
- view stats of user
    - view elo and count of push-ups overall
- view scoreboard
    - ranking all users according to elo descending
- view history
    - lists all records of user
- view tournament info
    - views if tournament is active or not
    - views log of current tournament or of the last one that ended(marked as ended)
- submit record/take part in tournament
    - saves the record in the database
    - starts tournament if not active, or takes part in current tournament

Tournament Logic

- If the tournament is inactive, the next entry starts a new tournament that lasts 2 minutes.
- If the tournament is active, the user of each following entry in the 2 minute timeframe takes part in the tournament.
    - For each entry the username and the count of push-ups is saved in a dictionary. If a user submits multiple entries, the counts are added together.
- After the two minutes are over, the leading participant/s get +1/+2 Elo which is saved to the database. All losers get -1 Elo.

Approach to the problem:

The first step was to set up the HTTP Server. Mr. Berger worked out a message server with the class in the SWEN1 lectures. Since the message server also included a multithreaded HTTP server, I integrated that part into my project and familiarized myself with the server. Afterwards I looked at the curl script, tried to understand what was tested in each case and how it made sense with the requirements written in the specification of the project. Then I started to implement the use cases from top to bottom. Register and Login user was done quite fast, as it almost worked out of the box with the example from the message server. For Getting and Updating the user data took me a bit longer and was probably the part where I was stuck the longest, since I missed some parts about how the routing worked. But after I understood what was missing the other use cases, such as getting stats, scoreboard, history, were done quite fast, as the problems were similar. For the tournament, I first needed to think about how it all should work for some time, since it was essential for the implementation. Since only one tournament was running at the same time always, I made the choice to use a singleton. First I made a basic tournament manager that just returned me a string. Then I introduced a timer so I can run the tournament for 2 minutes. After this I added a dictionary with users and push up counts and introduced a function to add entries, including database operations. Finally I added the elo calculation, also including database operations.

Unit Tests:

First thought was to test register and login user. But after trying for a little while I realised that all those functions require the database and are therefore unfit to unit test. Integration tests like the ones in the curl script are better suited for that.

Therefore I struggled a bit to find sufficient use cases and made all of them for the Tournament.

These are the my Test cases (all comments from my code):

1. Test that the TournamentManager.GetInstance method returns the same instance of TournamentManager
2. Test that GetTournamentInfo returns "No active tournament" when the tournament is not active
3. Test that the AddEntry method starts the tournament when the tournament is not active
4. Test that log is not empty after the AddEntry method is called
5. Test that the AddEntry method throws an exception when the Entry object is invalid
6. Test that the entry is added to the user pushups after the AddEntry method is called
7. Test that the AddEntry returns "Entry added successfully" when an entry is added
8. Test that count is added to the user pushups when the AddEntry method is called multiple times for the same user
9. Test that GetPushUps returns -1 when the user does not exist
10. Test that the Tournament is inactive after 20 seconds(should be 2 minutes, but for testing purposes its set to 20 seconds)

In my opinion the most critical ones are cases are 1 and 10.

- 1 ensures that all threads interact with the same tournament manager; else it would be possible for different tournaments to run simultaneously, which it should not.
- 10 ensures that the tournament ends after the selected timeframe, and is not endless.

Of course the other uses cases are also important, such as checking if the Tournament changes to active after the first entry, or that count are added up if a user submits multiple records, but since it would break encapsulation for some states directly with the unit tests, the possibilities are limited and would be better test some other way.

Lessons Learned:

The biggest gains for me were understanding how routing works, or should work, how to use lock and some aspects of unit testing. The parts that I probably still don't understand fully and is the one thing (that I noticed) that is not implemented as it should, are the tokens for authentication. At least I think that is not fully correct and I just used what was provided by the message server and the token are not saved in the database.

Time invested(approx.):

- Design: 10h
- HTTP Server: 10h
- Routing: 20h
- Database Operations: 7h
- Tournament: 10h
- Unit Tests: 4h

Total: 63h

Git link in Readme file