Dev-Diary Sports Exercise Battle

Inhalt

Setup

As a first step I wanted to understand the resources we were allowed to use for the project:

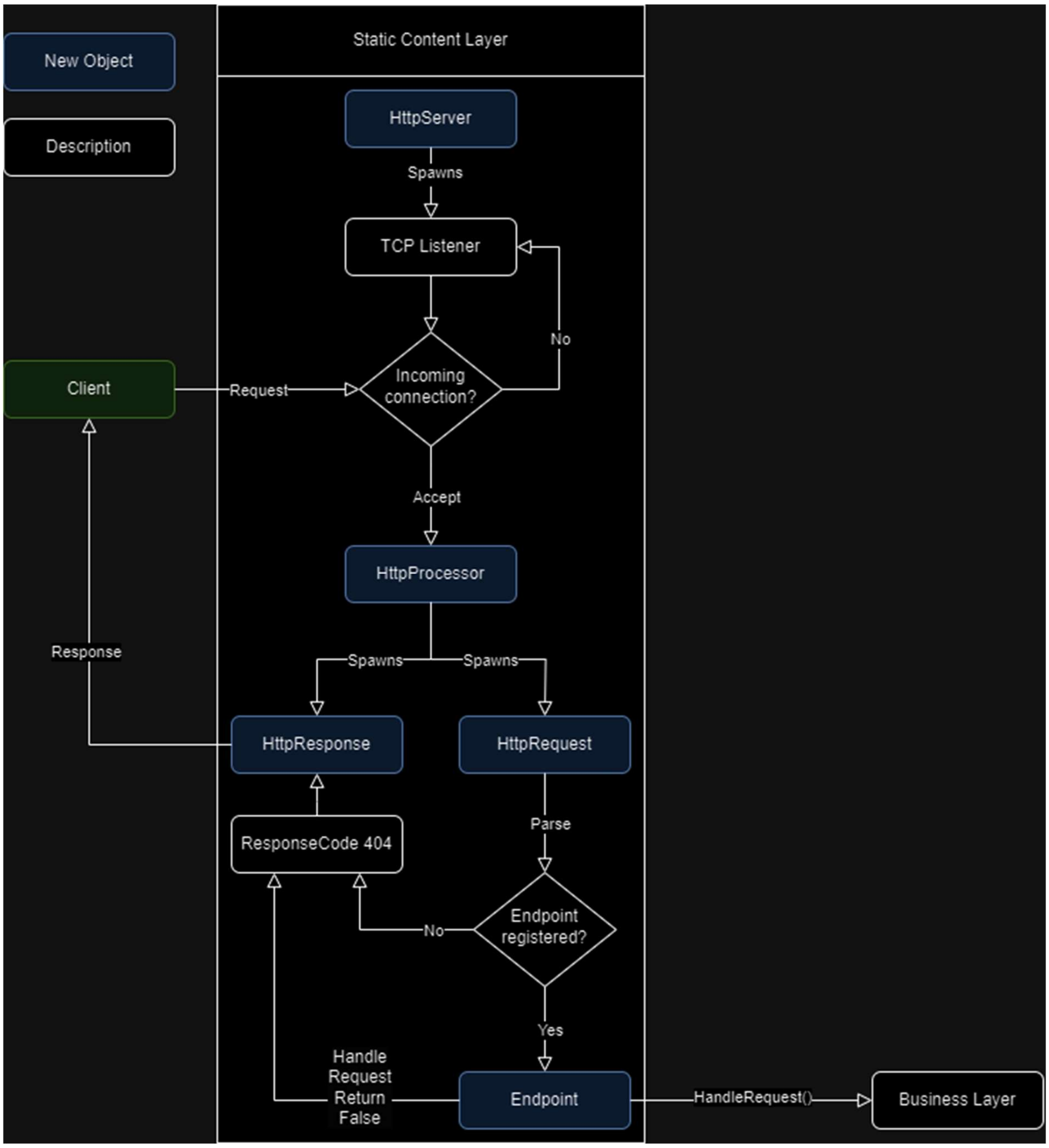
- Lecturer provided code for a functioning HTTP Server Demo (**_08A3A4HttpServerDemo.sln**) https://git.technikum-wien.at/swen/swen1/csharp/httpserverdemo/-/tree/refactorings/_08A3A4HttpServerDemo?ref_type=heads (only accessible FHTW internally or with VPN)
- **PostgresSQL** database run via **Docker**, accessed and modelled with **Datagrip (Jetbrains)**
- **Provided Curl Script** for Integration Tests
- **Swagger** and the MTCG- (previous project) as an API Specification Reference
- **Visual Studio** IDE

Further technologies used:

- **Draw.io** for database modeling
- **External C# libraries: Npgsql** for Unit-Tests, BCrypt.Net for Password Hashing

Introduction

To have more oversight, I started with creating a flowchart to better understand the provided code in the solution **_08A3A4HttpServerDemo.sln:**



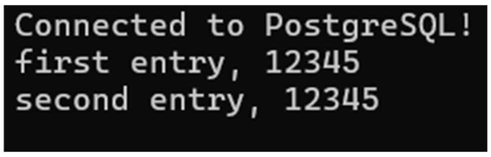
After installing a docker-container with a Postgres Image and created a new Container:

docker run --name SEB_postgres_container -e POSTGRES_PASSWORD=SEB_Admin -d -p 5432:5432 postgres

I downloaded the Application Datagrip and connected it to my Postgres database. I created a new role “seb_connection” which will be the login credentials for the SEB Datalayer. This role was provided with all privileges on the database and all tables and provided with a password. To test the connection via Datagrip, I created a test DML SQL file to create a table. Then I installed the library **Npgsql** via the nuget packet manager in Visual Studio.

Database Access

To connect to the database via the application, I created a database connection object which creates a connection, and executes a parameterized query:



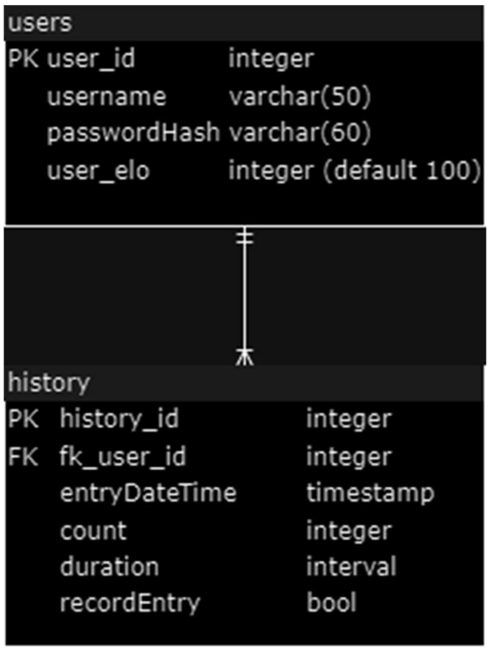
```
Connected to PostgreSQL!
first entry, 12345
second entry, 12345
```

To prevent SQL Injections, all queries are executed as parameterized queries. The BCDatabaseQuery object is the base-class for every data-access object and always provides an open connection to child-classes. All data-access classes inherit this base class. The CRUD-Operations always work in the param-bind-execute schema:

- Establish connection
- Define query including parameter placeholders, i.e. "… WHERE username == @username" these placeholders are required to prevent SQL-Injections
- Bind values to the placeholders query.Parameters.AddWithValue("username", *some variable*);
- Execute the query (variations for different CRUD-Operations
- Close connection

Database Modeling

I modelled the users table in the database, consisting of a primary key that is automatically assigned, a username and a place to store the password. Passwords will be stored encrypted and will be



encrypted in the business logic. A write into the database was also attempted.

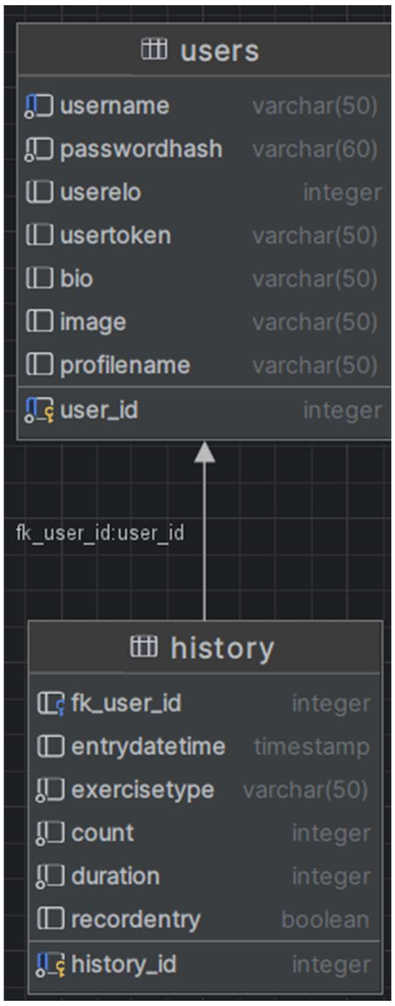
To model the database, I used diagrams.net and constructed the DML Statements accordingly.Endpoint USER

POST - USER CREATION

Next, I used the hashing algorithm provided in the Library BCrypt to hash the password. This will be used to store the passwords encrypted in the library and will also be important for the login functionality.

GET - USER HANDLING

Every Database access by a client needs to be authenticated first. I added more columns to the users table (token, profilename, bio, image) to represent the new features.



Starting with token, this gets generated at login and maintains a trust connection between the clients and the server by handing the session token to the individual user. As long as the client holds that token it can access and change all user-specific database entries. For that a new class was added (DatabaseAuthenticate) which takes the client provided token and checks it against the database stored token.

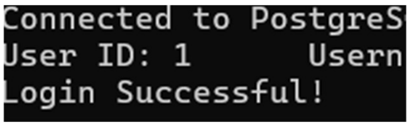
I added Bio, Image and ProfileName to a new helper Class “UserData”. The new class DatabaseGetUserInfo and the method GetUserInfo() was added to allow (authenticated) clients to access their profile info.

PUT – ALTER INFO

For the put method, the client alters the bio of the current user, via the object DatabaseChangeUserInfo().

Endpoint SESSIONS

Using the provided curl-script as a guideline, I started to implement the endpoint “sessions”. I created a new Endpoint object and registered it in the HTTPServer object. Next, I created a Database-access-object that receives the “user”-Object loaded with all credentials as a parameter and builds a query to access the database based on the username. The verifying of the given password was very tricky, because at first, I thought I had to rehash the given password and then compare it to the one stored in the database, however this did not work, since the used method “BCrypt.HashPassword()” required a salting value. If none was provided a random value was generated. This made the compared hashes differ everytime. Reading the documentation of the BCrypt Library further, I found the correct method which was “Bcrypt.Verify()”. Using this the authentification via the given credentials worked immediately:



```
Connected to PostgreS
User ID: 1       Usern
Login Successful!
```

The return value is a server-side generated token that is given back to the client. This way the client can access the personal session via the singleton object BLL_SessionManager which stores all accompanying sessions (individually stored in the session object). Two helper methods provide the username (which is sometimes needed for database queries) when the token is provided and the token when username is provided. (GetSessionByToken and GetSessionByUsername). Later I also added a GetUserID Method to save/get the corresponding user_id, this allowed me to insert history entries with the correct foreign key.

Endpoint STATS

After fully implementing the session functionality, it is now possible to run a query against an previously logged-in client which only provides the token. It accesses a

Endpoint SCORE

Similar to Endpoint STATS, this simply accesses a view (get_score) in the DatabaseGetScore Object and packs it into a UserStats-List. This then gets serialized.

Endpoint History

POST – History Entry

I reused a lot of code from the Endpoint /User/ POST method to create an entry in the history table. The biggest difference was the user_id which I saved in the session data and was used to create a foreign key reference in the history table for the corresponding user (see data modeling). A new object was required to deserialize the JSON-string called UserHistory.

GET – Get a list of all history entries for the user

I reused the UserHistory object to store the query return, which then was stored in a list of returns. This list in turn got serialized and returned to the user.

Endpoint Tournament

Similar to the Endpoint Sessions, to store and track all the tournament data I created a new singleton BLL_TournamentManager which stores a list of all tournaments. Every tournament has a constructor which calls a timer that deactivates the tournament after 2 minutes. Each tournament stores a list tournamententries. By calling AddTournamentEntry, the tournamentEntry is added to either a currently active tournament. If no active tournament is run, a new one gets created.

Projectinformation

Git Repositories

Sports Exercise Battle: https://github.com/LaurinGob/Sports_Exercise_Battle.git

Unit Tests: https://github.com/LaurinGob/Sports_Exercise_Battle_UnitTests.git

Working Hours

Date	Time Spent
29.03.2024	8h
30.03.2024	8h
31.03.2024	5h
02.04.2024	6h
04.04.2024	6h
06.04.2024	8h
07.04.2024	10h
08.04.2024	5h
09.04.2024	2h
10.04.2024	6h
11.04.2024	7h
13.04.2024	8h

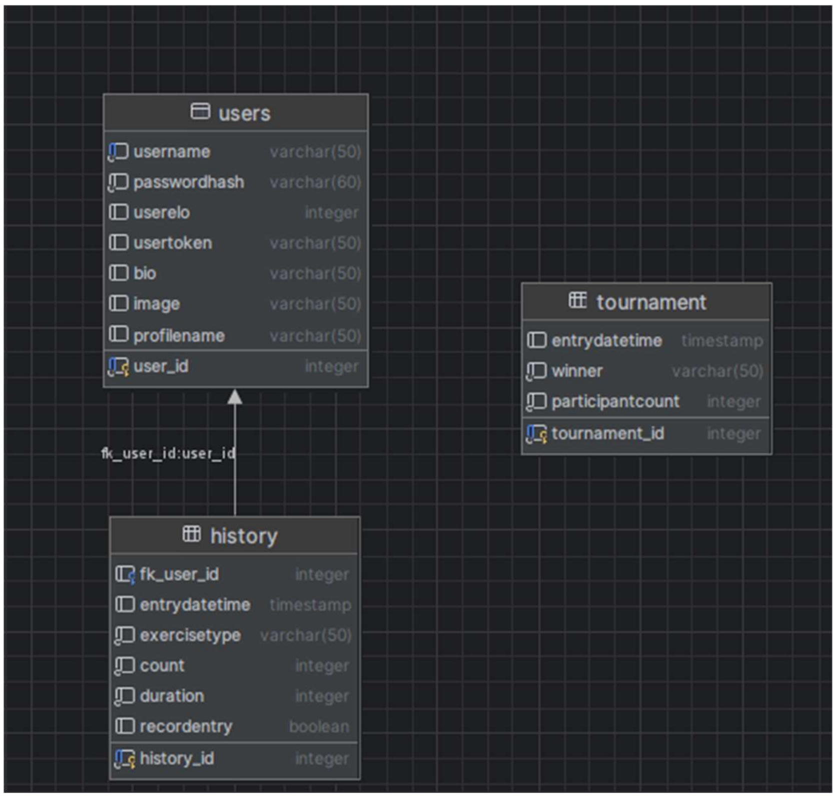
Change to Integration Tests (CURL)

The "wait"-Period between starting and checking the conclusion to the Tournament has been increased from 120 to 122 seconds to account for the asynchronicity between the curl-script-based integration test and the server. (sometimes tournaments wouldn't conclude in time for the next call)

To test the unique feature a new entry has been added to the curl script.

Unique Feature

As a unique feature I decided to build a history of all concluded tournaments. I added a new table "tournaments" in the database:



This will hold all past tournaments. I created an object “PastTournament” that holds all the datapoints so they can be handed to the DAO. It will be filled in the “ConcludeTournament” Method. The DAO writes the objects content into the database.

Endpoint TournamentHistory

To get the tournament entries, a new endpoint has been created. It returns all past tournaments. The endpoint has been added to the curl script. It returns a serialized list of PastTournament-objects.

Unit Tests

For the unit tests, I used the unit test framework NUnit.

TournamentPlacementTest

To test if the placement of the user correctly changes if a new (higher) entry is added. Main part of the battlelogic.

TournamentMultipleEntryTest

To test if multiple entries of the same person add together and the placement correctly changes. Main part of the battlelogic.

TournamentManagerAccessibleTest

To test if the singleton instance BLL_TournamentManager is accessable from everywhere in the code. This is very important since if it is inaccessible it can cause problems in many different parts of the code.

TournamentManagerNewTournament

Checks if a adding a new TournamentEntry by a test person to the BLL_TournamentManager-instance correctly creates a new Tournament.

SessionManagerAccessibleTest

Same as the BLL_TournamentManager, the BLL_SessionManager object is a singleton instance which provides a lot of functionalities in different part of the code, therefore it always needs to be accessible.

SessionManagerCreateSessionTest

Check if the SessionManager allows to create a Session for a newly logged in user. This is important because without an existing session, no further functionalities are available to the client.

SessionManagerUpdateSessionTest

Check if the session correctly receives updates. This is required because it allows up-to-date information to be accessible everywhere in the code.

SessionManagerUpdateElo

Same as SessionManagerUpdateSessionTest correctly updated ELO needs to be accessible everywhere in the code.

SessionManagerIsolateToken

The isolate token method is a helper function that allows the token provided by the client to be correctly parsed and checked against the session stored token. This is critical for all token based endpoint methods.