TP C#13: MOOOORE Rednit

Submission

There is nothing to submit here. It is a bonus to learn more about network. If you do anything from this subject, you can put it in the *bonus* folder of the archive you will submit. Do not forget to tell what you did in the README.

README

In this file, you can write any and all comments you might have about the practical, your work, or more generally about your strengths and weaknesses. You must list and explain all the bonuses you have implemented. An empty `README` file will be considered as an invalid archive (malus).

1 Introduction

Learn:

- how the server works.
- how the database works.
- how to install, create and administrate a database.
- how to launch the server in local.
- how to connect the client to your local server.

You don't want to play in the Rednit folder as it will be graded. So you should copy this *Rednit* folder and paste in the *Bonus* folder.

2 Lesson

2.1 The server

This section will describe how the server works.

2.1.1 Main

Initialize then Start the Server.

2.1.2 Data

Class similar to the Data class in the client (Rednit_Lite). It stores data for the execution of the server.

- DbConnection (NpgsqlConnection) An object that stores a connection to a database. Used to request the database it designates.
- Socket (Socket) The socket of the server.
- Clients (List<ClientData>) The list of connected clients.
- Tasks BlockingCollection<ClientData> The list who sent a message to the server and are waiting to be handled.

2.1.3 ClientData

This class stores the data of a single client.

- Client (TcpClient) The socket to communicate with the client.
- Login (string) The login of the user, if authenticated.
- IsQueued (bool) Used to ensure that HandleRequests is not applied twice on the same client at the same time.

2.1.4 Server

Initialize
Initialize the class Data, then starts the connection with the database. Finally it set the waiting queue for clients connection requests.

Start
Call Server.AcceptClients, Server.HandleTasks, Server.PollClients simultaneously. Accept-Clients and HandleTasks are executed in separate Threads.

AcceptClients
Continuously accept client's connection requests. Add the new clients to the list of Clients for polling.

PollClients
Go through the list of Clients (List<ClientData> Clients). For each client, check if there is data available and it is not currently being treated. So, if the client sent a message, add it to the list of clients to handle (BlockingCollection<ClientData> Tasks).

HandleTasks

Go through the tasks, the clients to handle stored in (BlockingCollection<ClientData> Tasks). For each client, start the treatment of the message they sent in a new Thread. The function called in this Thread is HandleRequests().

HandleRequests

Receive the message from the client. Depending on the type of the Request, calls the corresponding function in class Request. This function returns a new Protocol that is sent to the Client.

2.1.5 Request

The server can receive different kinds of requests. This class has one function per request there is to handle. I will not describe each and everyone of them, they are all documented. What they mostly do is send a request to the database (and receive a response).

2.1.6 UserData

Same as Rednit_Lite.

2.1.7 Protocol

Same as Rednit_Lite.

2.1.8 Formatter

Same as Rednit_Lite.

2.1.9 Npgsql

Npgsql is a package that was added to the project to be able to communicate with a database. Bellow, are described the important functions so that you understand how it works.

- new NpgsqlConnection(string): Create a new NpgsqlConnection used by the following functions to communicate with the base. Here is an example of use on rednit database with role server_sup.

```
NpgsqlConnection connection = new NpgsqlConnection("Host=127.0.0.1 ; " +
    "Port=5432; Database=rednit; Username=server_sup; " +
    "Password=espece de sup");
```

- .Open(): Start the connection with the database
- .Close(): Stop the connection with the database
- new NpgsqlCommand(string, NpgsqlConnection) Create a new command. That command takes in parameter a string that is a SQL request and the NpgsqlConnection object on which it should be executed.

```
NpgsqlCommand command =
    new NpgsqlCommand("SELECT * FROM users;", connection);
```

- .ExecuteNonQuery() and .ExecuteReader() execute the command in the database. If you do not expect an answer (or you don't need it) use ExecuteNonQuery(), it returns nothing. If you want to receive the answer from the database, user ExecuteReader(), it returns a NpgsqlDataReader that allows to read line by line the response. look at the code for more information about the reader.

```
command.ExecuteNonQuery();
NpgsqlDataReader reader =
    command.ExecuteReader(CommandBehavior.SingleResult);
```

Just these functions should be enough for you. If it isn't, ask Google.

2.2 The database

The database stocks the data used by the server. Each user is stored in it and all relations (like / match) between users are also stored in there.

The stored data in a relational database is in a matrix, with each line being a set of data and each column an information.

For a better understanding, here is the users table for Rednit.

```
SELECT * FROM users;
```

id	login	password	firstname	lastname	age	description	picture	hobbies
1	hack0	1234	hacker	man	69	Hack me		ffftt
2	hack1	1234	Michel	Dupont	42	Hack me		ttttt
3	hack2	0123	Robert	Dupont	666	Hack me		ftftt
4	hack3	0123	hacker	man	1	Hack me		ttfff

Each line is a user profile (randomly created). Hobbies should be divided in 5 columns, but the page is too small (f = false, t = true). So each user is stored as such in the database. They can be identified with their id.

Matches are stored like this:

```
SELECT * FROM users;
```

id	source	target	matches
1	1	2	f
2	1	3	t
3	2	3	f

Each line is one like or one match.
Source and target use the user's id, so 1 is hack0, 2 hack1, 3 hack2, 4 hack3.
If user1 never liked user2 and user2 never like user1 either, there will be no line added to the table.
If user1 liked user2 but user2 has yet to answer: a new line is added {source=user1, target=user2, matches=false}.
If user1 liked user2 but then user2 disliked user1, then the associated line is deleted.
If user1 liked user2 and user2 liked user1 then in the line with {source=user1, target=user2}

matches is set to true.

These two tables are all that is needed for Rednit. However there is still more there are users created in SQL to handle permissions and SQL functions too.

Look at the file init.sql in the *Rednit_Server* folder. There are the creation of tables, roles and functions.

The permissions give read/write access to user server_sup with password 'espece de sup'. Look at the config.json file in *Redni_Server* the fields username and password should be set to that. If you modify init.sql to change the password and execute it, you have edit config.json too.

Functions are called by the server. They were created to have easier requests in the server code.

2.2.1 Get postgreSQL

Install postgres, you should easly find how to do it with your best friend Google.

2.2.2 Start a postgreSQL server

After installing postgreSQL, you want to start it.

Windows

Press ⊞ . Type Services. Find PotgreSQl X.X Server. Click Start.

Linux

```
echo 'export PGDATA="$HOME/postgres_data"' >> ~/.bashrc
source ~/.bashrc
initdb --locale "\$LANG" -E UTF8
postgres -D "\$PGDATA" -k /tmp
```

2.2.3 Administrate database

psql

Start psql:

```
psql -h localhost postgres
```

It doesn't work well on Windows. A solution is to put Ubuntu on your windows.
`https://docs.microsoft.com/en-us/windows/wsl/install-win10`

psql allows you to administrate your database. It gives access to a number of commands. You can also write in SQL directly in it.

- \c <database>: connect to the database (rednit for example). By default you are in database postgres.
- \dt: print the tables in the base.
- \du: print the users.

- \i <file>: execute a .sql file.

With this, you should be able to do anything you want. Look at the example bellow that creates a new database and connect to it.

```
CREATE DATABASE test;
\c test
/* coder en sql */
```

Look at database.sh in Rednit_Server folder. It set up the database automatically. Execute this script to set the database and print the content of table users.

```
chmod 744 database.sh
./database.sh
psql -h localhost -U postgres
\c rednit
\dt
SELECT * FROM users;
\q
```

psql
pgadmin is the same as psql but with a graphical interface. On windows, you can use it instead of psql if you want. To download it ask your best friend.

2.3 Launch the server in local.

If the database is properly set up, all you have to do is run **Rednit_Server** in rider.

2.4 Connect the client to your local server.

You will need to edit two files: *Rednit_Server/config.json* and *Rednit_Lite/config.json*. In these two files, address and port in the server dictionary must be the same. You should set the address to "127.0.0.1" You should set the port to a random number between 1025 and 65535.

With this, when you start your client it will try to connect to your server. Your server must obviously be started.

3 Exercices

This part is only to give you ideas, you can do anything you want. Since you have the server you can modify Protocol, Server, Request and/or init.sql to create new functionalities.

3.1 Idea 1 - Sell data to the CIA

Modify the handling of the Chat on the server's side to keep all the messages exchanged between two users, so that you can sell their data to the CIA.

3.1.1 Idea 2 - Improve the Chat

Modify the handling of the Chat on the server's and client's side. Improve the chat by putting colors, dates (reception or sending), emotes, different kind of messages...

If you manage to add a voice call to it, GG.
If it is still not enough add a video call :).
C# should have some packages that can help you do that.

3.1.2 Idea 3 - SuperLike

Implement a SuperLike functionality, like Tinder. You will have to edit the windows form FormMatch, use Visual Studio for that.

3.1.3 Idea 4 - Add data

We want more data about the user for the CIA, so add something like sex, address, bank card... If you add the bank card data it has to be securely stored (HF).

These violent deadlines have violent ends.