Moustapha Isaac Diaby

SQA Course Work

Year – 2020, SCN - 090492276

Advanced higher Computing

Tick Tack Toe With Python Sockets

Contents

[Analysis 3](#_Toc35719707)

[The objective of this (SDD) project 3](#_Toc35719708)

[Scope 3](#_Toc35719709)

[Limitations 4](#_Toc35719710)

[Constraints 4](#_Toc35719711)

[Boundaries 5](#_Toc35719712)

[End-user / Functional Requirements 5](#_Toc35719713)

[Project plan 6](#_Toc35719714)

[UML Diagram 8](#_Toc35719715)

[Gantt Chart 9](#_Toc35719716)

[Design 10](#_Toc35719717)

[Database design 10](#_Toc35719718)

[SQL: Initialize the database table and fields 10](#_Toc35719719)

[SQL: check if a username is in the database 10](#_Toc35719720)

[SQL: creating a new user account in the database 10](#_Toc35719721)

[SQL: updated the user data after playing a game 11](#_Toc35719722)

[Sorting algorithm 11](#_Toc35719723)

[Wireframe of the client’s user interface (UI) 12](#_Toc35719724)

[Login / Registration Page 12](#_Toc35719725)

[Home Page 13](#_Toc35719726)

[Joining Game Loading Page 13](#_Toc35719727)

[Game-board Page 13](#_Toc35719728)

[Leader-Board Page 13](#_Toc35719729)

[UML Classes 16](#_Toc35719730)

[Socket Server class 16](#_Toc35719731)

[Client Socket Server class 16](#_Toc35719732)

[Implementation / design - What are “Actions” going to be? 19](#_Toc35719733)

[Pseudocode: Action 20](#_Toc35719734)

[Action Types 20](#_Toc35719735)

[Example usage: Pseudocode for authenticating users (OOP) 26](#_Toc35719736)

[Implementation 29](#_Toc35719737)

[Researching 29](#_Toc35719738)

[Python Sockets & pickling python objects 29](#_Toc35719739)

[concurrent programming in python 33](#_Toc35719740)

[Packaging and depackaging actions 35](#_Toc35719741)

[Handling unauthenticated clients 36](#_Toc35719742)

[[USER LOGIN] 36](#_Toc35719743)

[[USER REGISTER] 38](#_Toc35719744)

[Issue Log: 39](#_Toc35719745)

[Testing different scenarios: User Interface 40](#_Toc35719746)

[Authentication page 40](#_Toc35719747)

[Incorrect inputs test 41](#_Toc35719748)

[Register button Testing 43](#_Toc35719749)

[Login button 46](#_Toc35719750)

[Successfully providing the right password with a username that exists in the database 47](#_Toc35719751)

[Leader board page 48](#_Toc35719752)

[Joining a game loading page 51](#_Toc35719753)

[Successfully getting 2 players in a game 52](#_Toc35719754)

[Multiple server connections 6 55](#_Toc35719755)

[Evaluation 56](#_Toc35719756)

# Analysis

## The objective of this (SDD) project

To develop a tick tack toe game that uses Sockets to allow players (clients) to play against other clients on different (or the same) computers that are on the same network. This application should be a fun game for students to verse one another in a game of tic tac toe.

I will need to develop a server that will handle authentication, SQL commands to the database and connecting clients with other clients in a game session. This can be achieved by using sockets in python.

I will need to define a suitable database table structure for the application that will store player data like the number of wins, loses, games played, username and password.

Players will also be able to view the leader-board which will be sorted with the (inverse) insertion sort algorithm. This will make the game feel more competitive as students will want to climb the leader-board to become the best. Players will also be able to search for their rank and any other user’s via their username, this provides a quick and easy way to find a player's rank.

Players will be able to create a new user by sending a unique username and a password to the server - which will create and save the new user credentials to the database.

Players need to be authenticated to use the main functionalities of the application. This will be done by passing valid user credentials (username and password) to the server - which will compare the password that the unauthenticated client sent with the correct password for the desired username in the database.

Upon getting authenticated by the server, the server will send the client back the saved user data from the database and recognise any future action request from that client as being the authenticated player.

## Scope

For me to complete this project I will need to do the following:

* Create an intuitive wireframe UI design for the client’s different views. This should include the authentication page, home page, in-game page and leader board page. Other views will also be designed too if I need to.
* I will also need to design a use case UML diagram that will show how the user will interact with the client and then to the server.
* Create a pseudocode representation of an “Action”, this is going to be the object format that will be sent and received from the client. The concept of an “Action” will make sending data to the server or client - structured and much easier to handle the data being sent as it will be predictable.
* Program the designed user interface with python Tkinter and make each button perform the correct function.
* Research on how to use sockets with python and pickling Python objects. Sockets and pickling go beyond the advance higher course; however, it will be required to complete this application. It’s used by both client and server to send pickled python objects to each other in bytes. (then unpickled the bytes back to the python object when the data is received).
* Program the socket server using python. The server must be able to handle any defined “Actions” correctly and pass back the correct data (including errors) to the client that sent the “Action”.
* Program a working authentication/registration system that communicates with the socket server that will be used by the client socket.
* Persists user data via SQLite as the database.
* Program a fully working tic tac toe game, that knows when the game is over (if a player won or the game is a draw, these are the end state of the game board ).
* Create working pseudocode representation of an (inverse) insertion sort and implement the code in python to sort the leader board based on the number of games won.
* I would also want to develop a strong test routine, that will attempt to cover a range of values being sent to the server by a test client and check that all functional requirements are working.
* Lastly write an evaluation of the project (robustness of the code, maintainability and how fit for purpose is the application)

## Limitations

These are the things that this application will not handle or expect players to do:

* The server will not fully expect SQL injections (however it will not make it easy to do so. There will be some level of protection in every SQL command)
* The solution will not make sure that there is only one session per username logged in at any given time. This would stop players from cheating and possibly playing against themselves.
* The server will not automatically close a game session when one player is inactive or disconnected.

## Constraints

There are a few technical, legal and time constraints for this project:

* The client and server will be programmed using python 3.7. I chose this is because I have a lot of experience using python in school and in my own free time.
* I will be using an SQLite database and SQL to interact with the database from the server. SQLite is like a smaller version of MySQL that is saved to a file, I chose this database because it will make setting up the server much easier for the end-users.
* The concurrent tick-tack-toe game sessions that the server should be able to handle is more than 3, so the server must be able to handle at least 6 client’s connections to the socket server concurrently.
* There is no development cost for this project because I am using pythons’ built-in libraries to develop the whole application. Therefore, there are no third-party dependency costs.
* Due to GDPR, I will have to allow clients to see all the data that the data controller has on them. That’s why I'm not storing any personal data about the user such as name and date of birth. All data that will be saved will only be part of the game and will be viewable in the leader board section of the app. (since I'm making the application easily usable, anyone could set up their server and become the data controller.)
* The project should be completed before April – this is when I must hand in my project to the SQA.
* The application must work on a school computer.

## Boundaries

This project will be done when:

* Players can successfully register a new account and authenticate into the application.
* Players can play a full game of tick tack toe against each other and reach an end state (win or draw) on the board.
  + When each player’s data in the databases get updated after a game has finished according to the game result.
* Players can view the top 20 players on a leader board and be able to search for their own or other player’s rank.

## End-user / Functional Requirements

There is a large range of people that can use this application. High school students can use this program to play with friends and families, however, they will need a single computer/server to host the server that will be listening to client action requests. The end-user will be expecting the followings:

A working authentication and registration system that has a clear and easy to use interface for the client to enter their username and password (plus information to connect to the socket server, for example, the Ip and port of the socket server).

Users will authenticate their client by inputting their username and password which will be sent to the socket server where it will get verified if the username is not found or the password in the database is incorrect the socket will reject the client.

* There should be a connection to the socket server which will receive a login action - carries out the followings:
  + Checks that the username exists in the database.
  + Checks if the password sent (encrypted with the MD5 algorithm) matches the password in the database. If so, it will authenticate the client’s socket and send back the user’s data.

The client will be able to register a new user account by providing the client interface with a unique username and a password.

* There should be a connection to the socket server which will receive a register action - carries out the followings:
  + Validate that the username is unique.
  + Encrypt the password with the MD5 algorithm and register the user in the database.

The client’s authentication/registration system must check that a username and password has been provided and is not left empty and password is of length 6 or more, the server will handle the authentication and registration of a client.

The authenticated client will be allowed to perform the following:

Clients will be presented with a user interfacethat will be created with python Tkinter after they have been authenticated. This will allow the authenticated client to do 2 main actins:

* Join a Tic Tac Toe Game.
* View the leader board.

Join and play a fully working and visually appealing game of tick tack toe game with another authenticated player, which will be controlled and kept tracked of by the socket server. The results of the game will be saved in a database. (increment the number of wins or losses and overall games played)

* 2 clients should be connected via a session system to play a game of Tic Tac Toe.
  + This session will be managed by the socket server that is running concurrently.
  + The server needs to run concurrently to support the minimum active games session of 3 games.
* The client's view will change, showing the clients and their opponents' username and the game board.
* Two players will take a turn in selecting a position on the board. This position will be sent to the server with the game id and player id that took the turn.
  + The client will check if the position selected is valid before sending the selected position to the server, which will then check if the board has an end-game condition (winner or draw), if the board isn’t in an end-game condition state, the server will switch player turn and repeat until there is an end-game condition state. Else the results of the game will be saved to the database (players account documents in the database will be updated to reflect the result of the game).
    - If you win there will be a +1 in the column “wins" and “games\_played”.
    - If you lost or the game was a tie there will be a +1 in the column “loses “and “games\_played”.

In the leader-board page, the client will be able to view a correctly sorted leader board by the number of wins descending order (the player with the most games won at the top) maximum of 20 players will be displayed, other users can be searched up to find their rankings.

* Fetch all user data action will be sent out to the server to fetch all player data from the database (minus their encrypted password)
  + The returned data will then be sorted on the client-side by using an (inverse) insertion sort and displayed to the user.

A working player rank search (this will be used to search for the currently authenticated clients rank).

* Clients should also be able to search for a username and get back that users ranking.
* When searching for a username on the leaderboard, if the field is left empty or the username isn’t found the display “Couldn’t Find User” message.

## Project plan

I need to plan the time that each task will take and define what it would include.

1. I will need to design the database, server and client's functions. (6h):

Database Design (1.5h):

* + The table names, and their fields. This will give me a better understanding of how to structure the back end.
  + I could also create some SQL code that will, get all the players, and get one player by username. By listing these commands, it will be useful when it comes to implementing the SQL code on the server.

Server Design (2.5h):

* + Plan the class functions that will be required for the project to meet the specification. This will include inputs and outputs.
  + Come up with a solution to control how the Client will interact with the server and vice-versa.
  + Draw a flow diagram that will illustrate the client's interaction with the server and to the database.

Client Design (2h):

* + Make a wireframe UI design that will demo what the client will see on each view.
  + The wireframe will also include illustrations of actions. E.g. button clicks, page switches etc.

1. Set up the database (0.5h):
   * This will include creating SQL code to create the database, defining each table, and add a test login user. (This code will be executed by the server on an initial start-up. This is because the SQLite database will need to be configured locally every time the server is running on a new SQLite database.)
2. Research (3h)
   * How to use sockets in python.
   * Pickling in python.
3. Set up the server (8h):
   * Allow users to connect to the server, authenticate and perform the requirements in the specification.
   * I will also have to continuously perform some tests during the development of each creation of a function. This may also decrease the time required for testing.
   * Program the session system for players to play a game of tick tack toe with the end-conditions.
   * For testing, the server should print out all the “Actions” it receives followed by which client that sent it (username).
4. Set up the client (8h):
   * Create the client’s UI from the wireframe UI design.
   * Program the required client functions that are specified in the UML Use case diagram.
   * Continuously perform tests during the development of the client’s functions. I might also need to go back to the server program and refined some functions that cause issues or need to be reprogramed.
5. Test (3h):
   * I will need to test the whole application and document the results of the test. This will make sure that the program is in scope/ performs correctly to boundaries and covers the constraints of the specification.
   * During this stage, I will also make the program handle errors. Noting them down as I do.
6. Evaluate the program. Refine the program to meet any missing specification / make the program maintainable and outputs useful errors. (3h)

The estimate total time to finish this application is 31.5 hours. However, this may vary because I will be working on more than one section at the same time. For example, when programming the server, I will be programming the client as well because I will need to test my solutions.

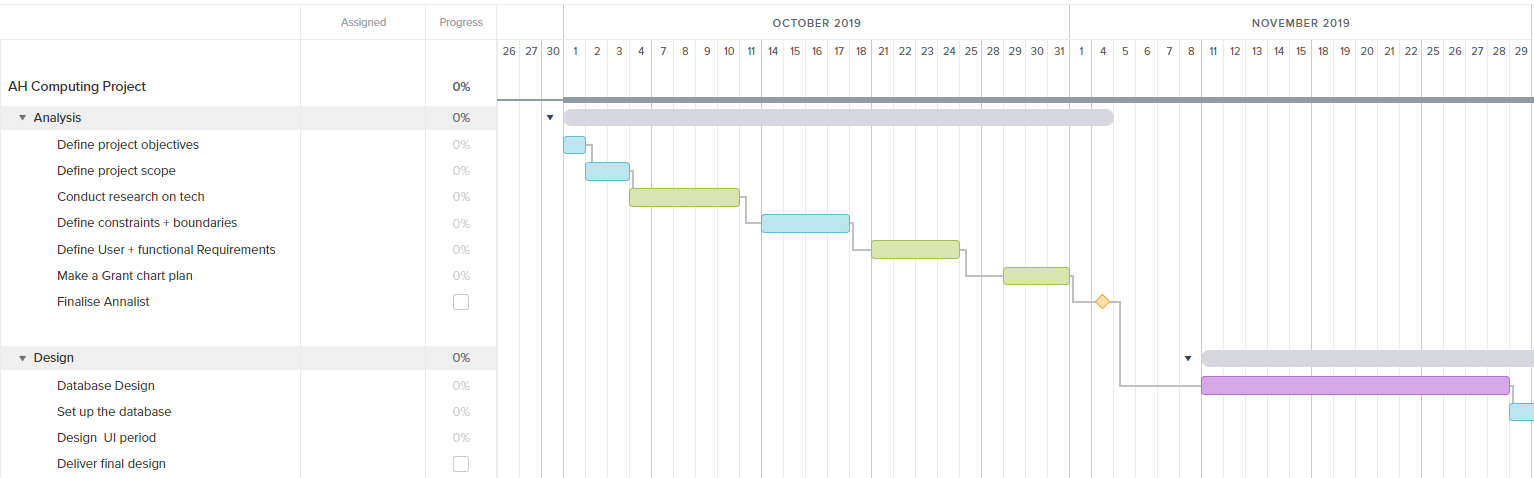
## UML Diagram

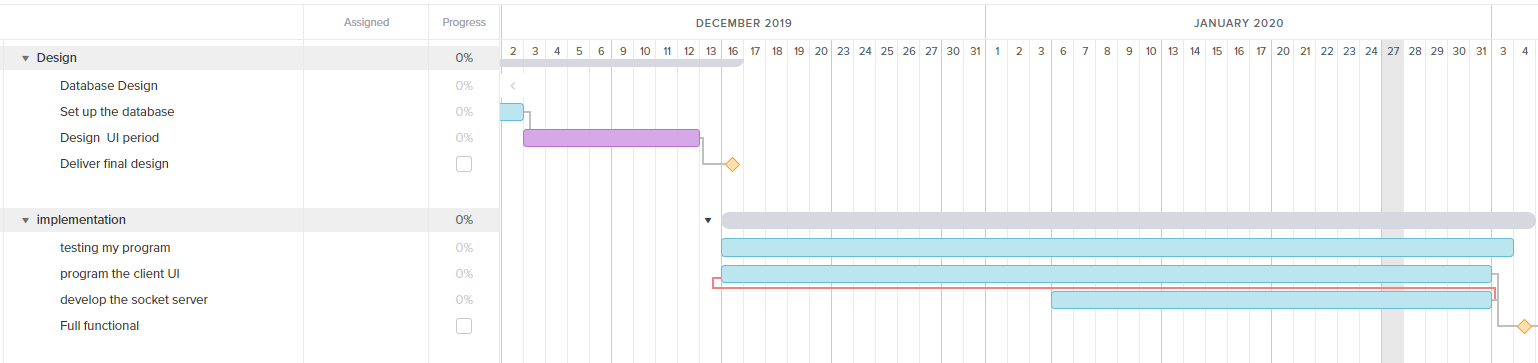
UML use cases section 1

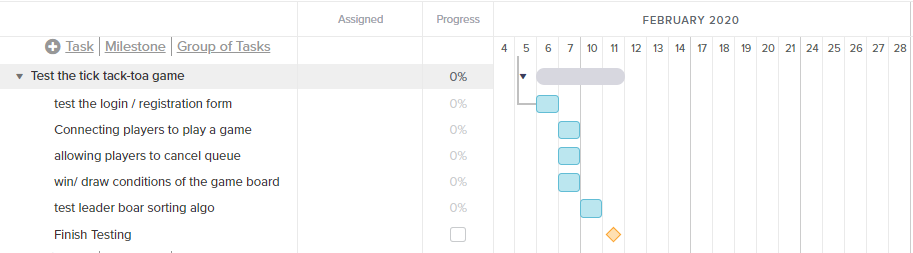

UML use cases section 2


UML use cases section 3


## Gantt Chart







# Design

In this section, I will be creating a plan of what I will be programming like wireframes, data structures and some concepts like actions and sockets.

## Database design

The database will be SQLite, which is easy to start-up (it’s just a file) and requires SQL commands to interact with it. All SQL queries will be executed on the server socket, which adds a layer of security.

### SQL: Initialize the database table and fields

When starting the server for the first time, I will need to Initialize the database.

CREATE TABLE IF NOT EXISTS users (

username VARCHAR (25) NOT NULL UNIQUE,

password VARCHAR (500) NOT NULL,

wins INT UNSIGNED DEFAULT 0,

loses INT UNSIGNED DEFAULT 0,

games\_played int UNSIGNED DEFAULT 0

);

* The “IF NOT EXISTS” option makes will stop the server from successfully creating a second users table when the program restarts.
* Fields like “wins”, “loses” and “games\_played” are unsigned integers because you can’t have negative games won, lost and played. It makes the database more robust.

### SQL: check if a username is in the database

“: username” is a variable for the username

SELECT username FROM users WHERE username = :username;

* I do not get the password column now because I want to a layer of security.
* I will only return all columns if the (encrypted) password matches the saved password in the database for that username.
* If this query returns null that will mean that there is no user in the user’s database that has that username. If the client wanted to register a new account, they will now be able to do so.

### SQL: creating a new user account in the database

INSERT INTO users (username, password) VALUES (? ,?);

* The question marks (?) are replaced with the username and password value, with SQL injection protection. Using them makes the program more robust and safer to use.
* We don’t need to provide the “wins”, “loses” and “games\_played” columns because we already set up default values of 0 when we created the table.

### SQL: updated the user data after playing a game

“: username” is a variable for the username

If the player won:

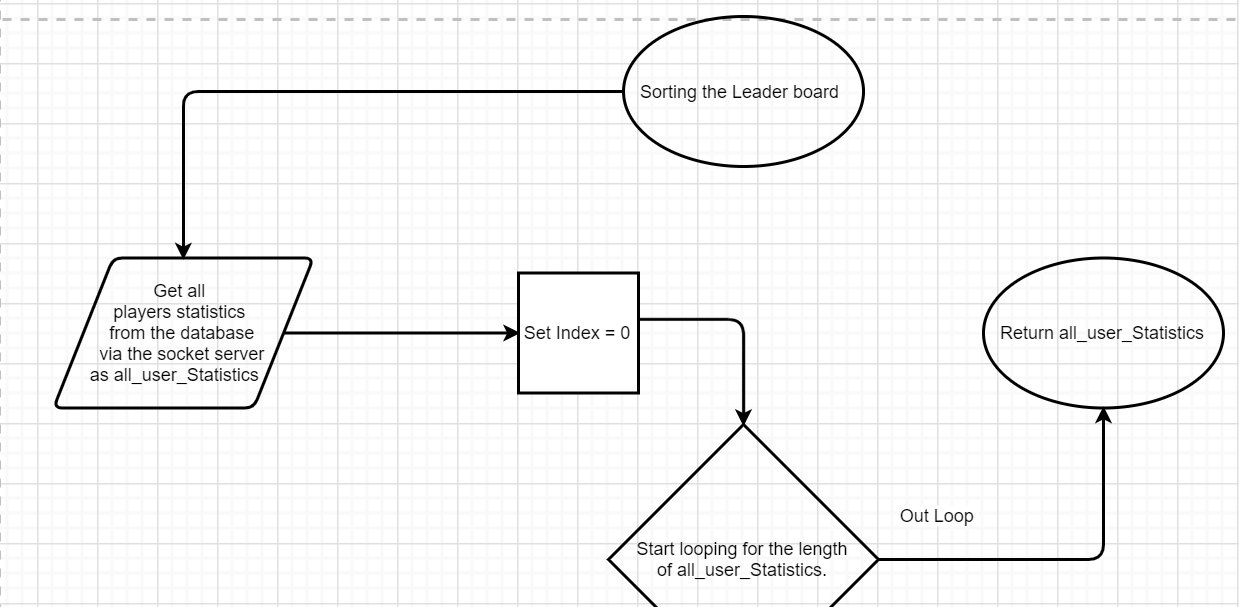
UPDATE users SET wins = wins+1, games\_played=games\_played+1 WHERE username = :username

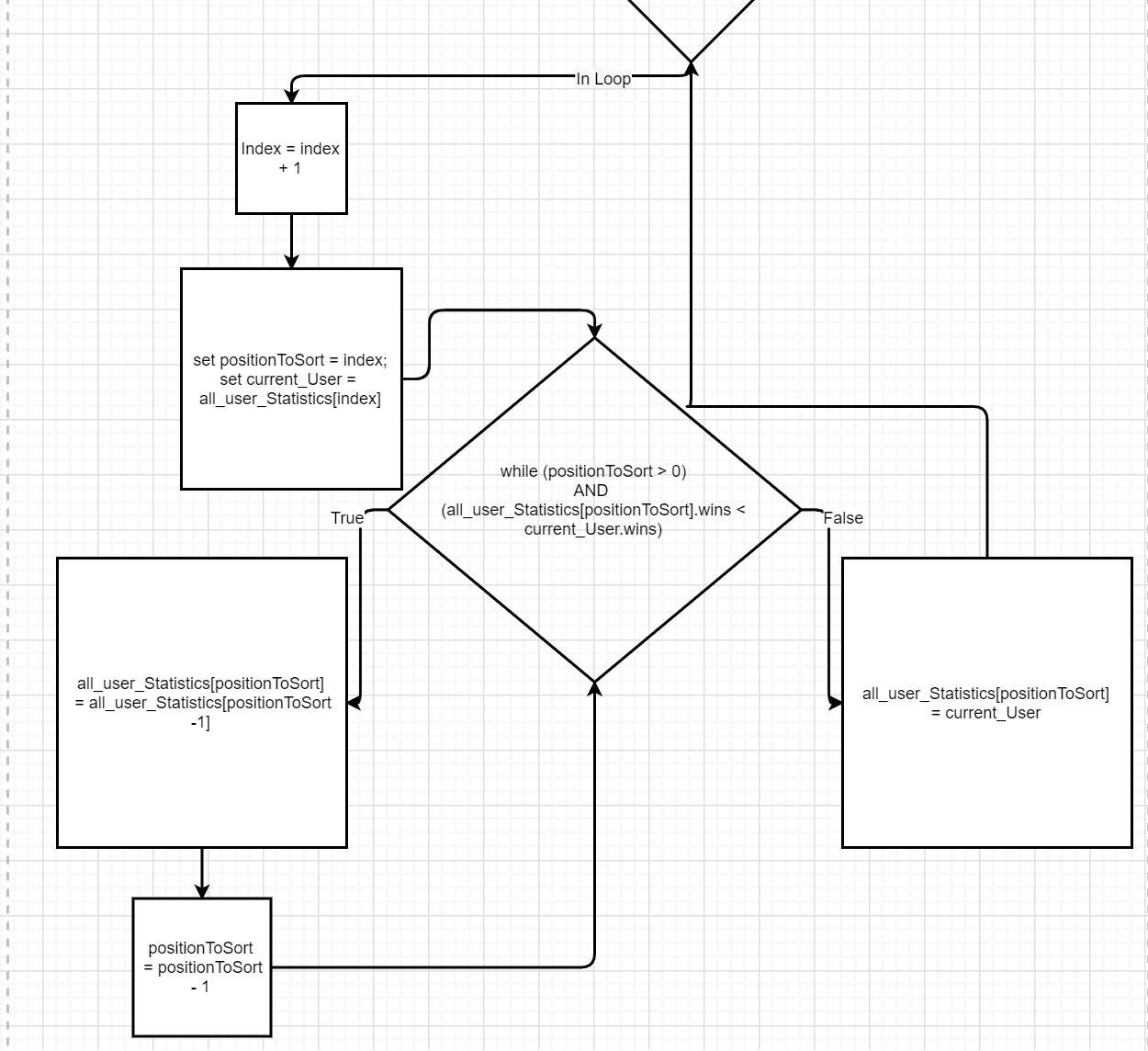
Else:

UPDATE users SET loses=loses+1, games\_played=games\_played+1 WHERE username = :username

* Both commands will increment the games\_played by 1
* Depending on if the user won the game or lost/drew it will increment wins or loses respectively.

## Sorting algorithm

Since I'm using an (inverse) insertion sort which is an advance higher concept, I will be writing up a detailed flow chart on how I will be using it in this application. The inverse of an insertion sort will simply just have the greatest number of wins first (descending).



## Wireframe of the client’s user interface (UI)

The following client user interfaces will be created using python Tkinter. Each design aims to:

* Making each label clear, predictable and easy to follow.
* Considers the user's experience, by including a dedicated section to inform the client as to what’s happening (errors and action messages.)
* The designs illustrate navigational buttons or events in a clear way.

### Login / Registration Page

The authentication section - the client will be able to provide their username and password (both type String) that they want to authenticate themselves with.

The server info section – the user must be able to provide the client with the location of the socket server (type String) and its port (type Integer). If the client is not properly directed to the address of the socket server and its opened port, the client will not be able to connect to the server when attempting to authenticate/register. There will be an error message displayed saying “Can’t reach the socket server”.

If the user tries to register a new account, there will be a '[USER REGISTER]' action sent to the server. if the username has already been registered the client will be told that the provided username is already in use. Else If the username provided hasn’t been registered in the database, the account will be made, and the client will be sent a confirmation message that their account has been created and they can attempt to login with their credentials.

If the user tries to log in to their account, there will be a ''[USER LOGIN]' action sent to the server. If the username isn't found in the database or the password is incorrect, the client will be told that there are no users with that username, or the password is incorrect respectively. Else if the passed credentials are both correct the client will be authenticated by the server and the client will be authenticated and sent to the Home Page.

### Home Page

This page is only accessible to authenticated clients. There will be a welcome message at the top of the page followed by the currently logged in player username. A “join game "and “view leader-board” button followed by a section at the bottom of the page that allows me to display any information/ errors messages.

### Joining Game Loading Page

When a player wants to join a game, they will be presented with a page that contains a message as to what is happening (“Waiting for another player to join”) and a cancel button that the client can use to leave the “waiting for a game queue” on the server.

If there is a second player also waiting to play a game, the server will start a game against the two clients, then the view will switch to the Game-board Page

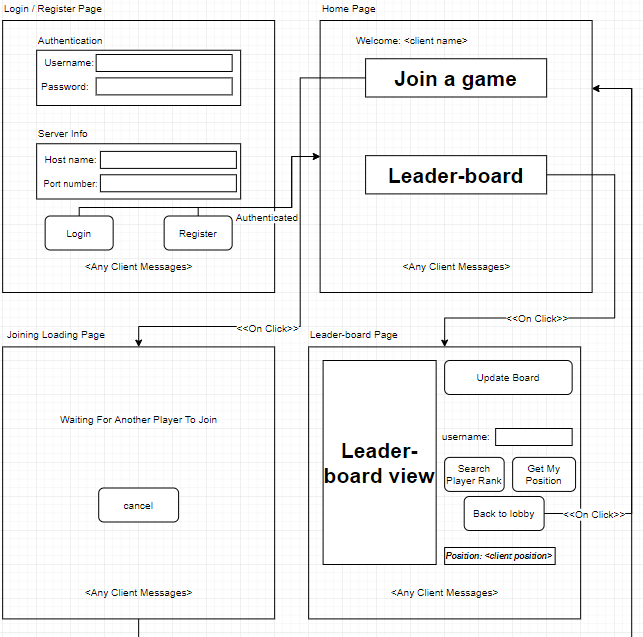
### Game-board Page

There will be a 3 by 3 game-board that allows the current player to take a turn by selecting a square they want to place their colour. If it isn't the client’s turn, selecting the squares will do nothing. Else the selected square will be validated and send to the server to update the board. Attempting to select already selected squares will also do nothing. At the bottom of the page, there will be a section for the player's information followed by the current player turn and any errors or information.

When the game ends, their page will display the end condition of the game and a button that will say “go back to Home Page”

### Leader-Board Page

On the left-hand side, there will be a sorted list of all account names with their statistics in descending order of the number of wins. On the right-hand side, there will be a button to update the leader board data, search for a specific username’s rank via a text field and button and a quick way to find your rank. There will also be a button to allow the player to go back to the Home Page. At the bottom of the page, there will be a section for the user to receive any information/ error.



Client Wireframe section 2


## UML Classes

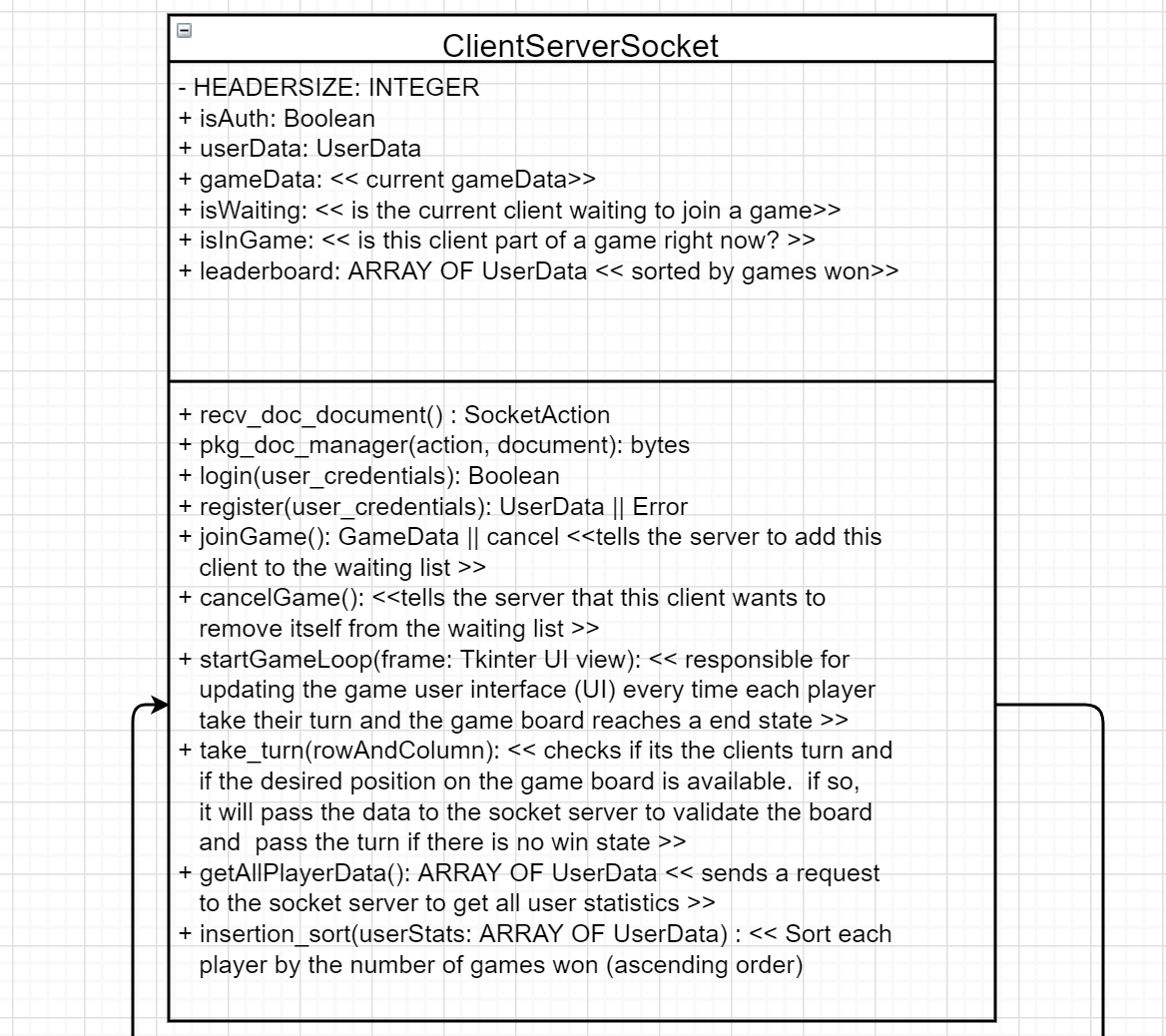
I created a class UML diagram for the client-server-socket and socket-server. Both client-server-socket and socket-server will be receiving/sending objects that are in the form of the **SocketActions** class. An example of how Actions will be used is going to be in the next section.

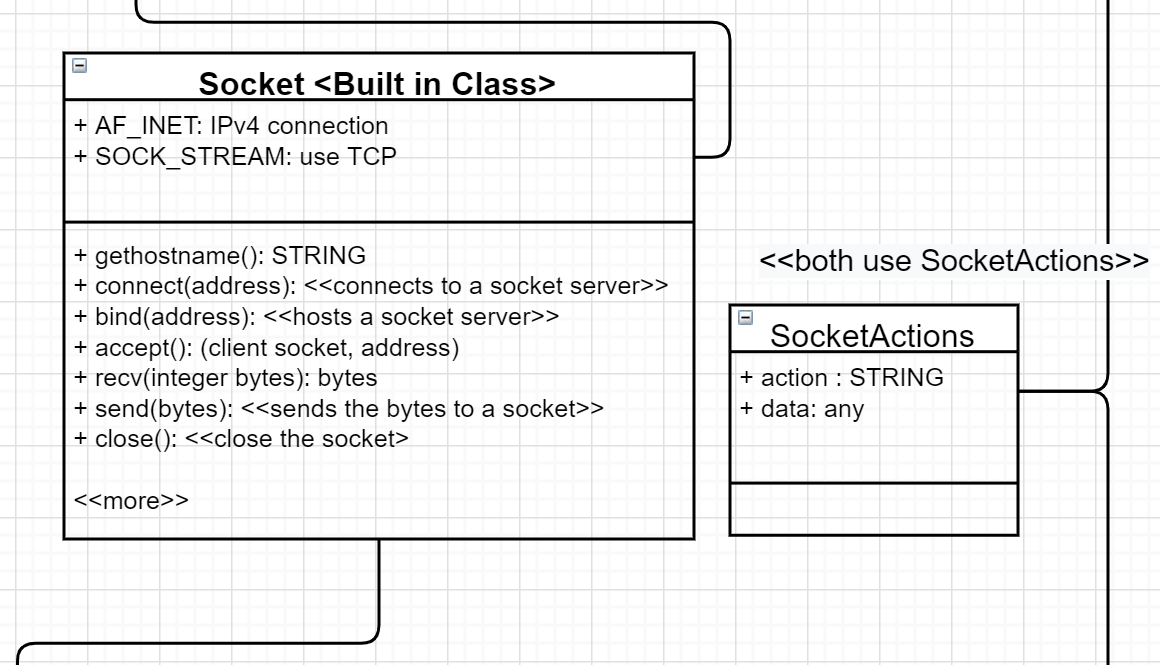
### Socket Server class

The server should not be accessible manually because its purpose is to authenticate clients, perform CRUD (Create, read, update and delete) operations on the SQLite database and handle game logic, that is why all its properties and functions are private and it inherits the Socket class in python. It also uses the SQL-Server-Connection class which will just connect to the database with valid credentials, set up the database’s table and allow the Socket-Server class to perform CRUD operations on the database.

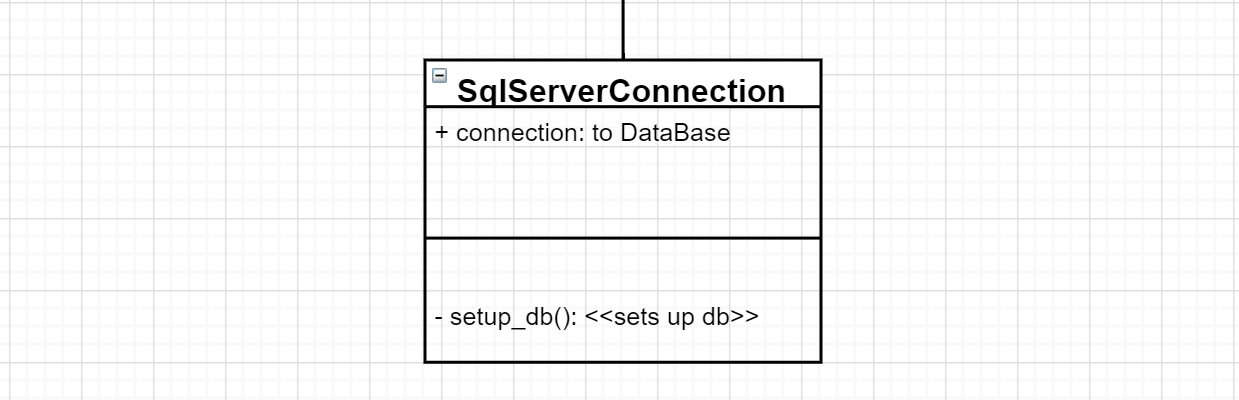
### Client Socket Server class

The client socket will be responsible for providing the Tkinter application data, verification/authentication, connection with other clients (to play the tick tack toe game with anyone else that is connected to the server). Since the Client Socket server class will be heavily utilized by the front-end application, I made all (except HEADERSIZE because it needs to the same as the server) of the Client Socket server-class properties and functions public.





UML Class Section 3

## Implementation / design - What are “Actions” going to be?

To answer the question, I must explain what pickling an object in Python means. Pickling is simply a way to turn python objects into bytes. Since I can only send bytes between sockets, pickling data is an easy way to send action objects. Once the data is received by a socket, I will need to unpickle the data to receive the python object.

A pickled “action” record will be the only thing that will be sent to and received from the socket server. By defining the structure of the record, will make data being sent and received from socket server, predictable and easier to maintain when using the unpickled data. I was inspired by the “Redux/Ngrx” pattern, which I am personally fond of using as a data manager for my web applications. The client will be the “Action Dispatcher” (which means, it will be the source of all actions). The server will act like the reducer which will be responsible for controlling the state and reacting to actions being sent to it and responding with its action that reflects the result of the received action.

### Pseudocode: Action

Record **SocketActions** IS {STRING action, ANY data}

The action value will be a string the briefly describes the type of action that is being sent. Each action should be unique so that no two action types should be used and is expected to do different things. For example, “[LOGIN USER]” should be used to handle authentication of an already existing user and “[USER REGISTER]” only handle clients that are trying to create a new user account.

The data value can vary depending on the action that is being sent, that is why it has a type of any. I will have to make the client send the correct data for each action type otherwise the server will send back an error action with a message string as the data.

### Action Types

Here are all the possible actions that can be sent between the client and the server. The headers are the action types that a client (player) socket can send to the server because they are “Action Dispatchers”, each action type also has its expected “Data” that needs to be passed to it. The server will respond with the action type followed by “SUCCESS” or “FAIL” reflecting the result of the request.

#### [USER LOGIN]

Attempt to login to an account. If successful, the server will allow the client socket to connect to the server and recognise future actions from the sender socket to be that user until the connection is closed. Else the server will not accept the client’s connection and respond with an error message. If there was an error connecting to the database to look up the username and compare the hashed passwords or an unexpected error occurs, the server will respond with a failed action with the error message as its data.

Data: (username: STRING, password: STRING)

Responses:

* [USER LOGIN - SUCCESS] – Data: RECORD {result: True, data: **UserData**}
* [USER LOGIN - FAIL] – Data: RECORD {result: False, data" "Incorrect password"} OR

RECORD {result: False, data: "Error when authenticating client's account."} OR

RECORD {result: False, data: "No user found with the username: (username)”}

**UserData** = RECORD {username: STRING, wins: INTEGER, losses: INTEGER, games\_played: INTEGER}

#### [USER REGISTER]

Attempt to register a new user account. If successful, the server will create a new user entity in the user's table with the desired username and hashed password in the SQLite database, the client can use the same credentials to authenticate when it sends the login action to the server. If the username has already been registered or the server was not able to connect to the database, the server will respond with a failed action with the error message as its data.

Data: (username: STRING, password: STRING)

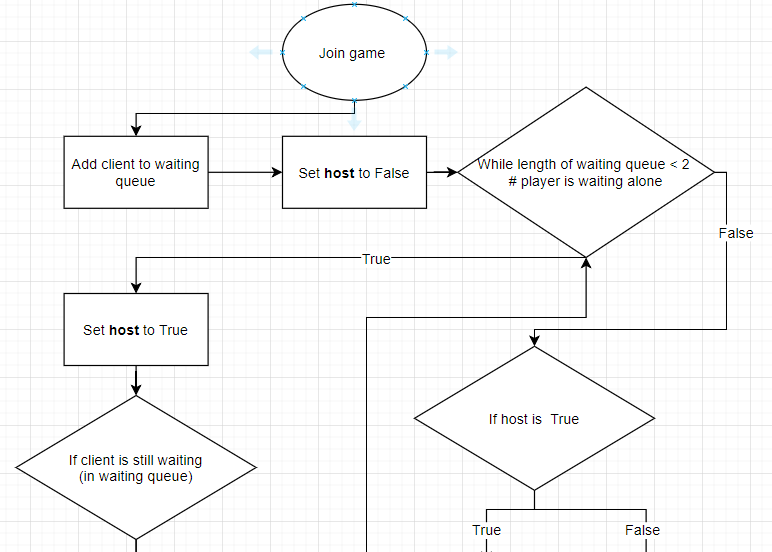
Responses:

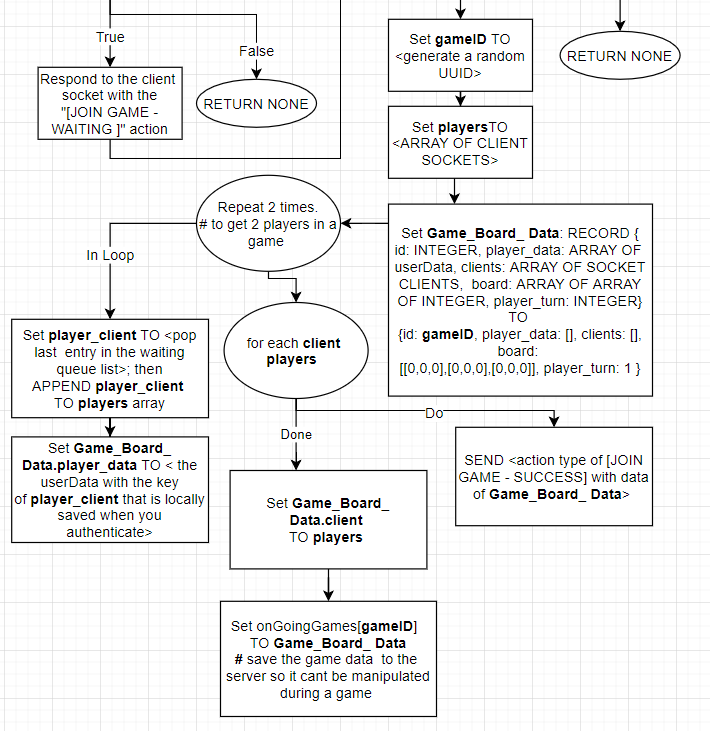
* [USER REGISTER - SUCCESS] – Data: RECORD {result: True, message: "Account was created successfully."}
* [USER REGISTER - FAIL] – Data: RECORD {result: False, message: "Username already exists. "} OR

RECORD {result: False, message: "Error when authenticating client's account."}

#### [JOIN GAME]

The client is attempting to join a game. If successful, the server will append a new game data in the **onGoingGames** record with a randomly generated **gameID** as the key and the **Game\_Board\_Data** as its value. Both players will get sent a success action when they are both in a game with the **Game\_Board\_Data** as its data. If the player is waiting in the queue alone, their request will become a host ( this is so I don’t get duplication of games being started by both players in the queue when they are getting paired up), only the host request can create a new game session and notify both players that they are in a game. Whilst waiting the server will check if the host request is still waiting in the queue and will wait for a second player to join the waiting queue on a new thread so the program is not blocking. (allowing other player’s action to be processed on the main thread without needing the previous code to finish). There is a flow diagram below that illustrates how the server will handle the “[join Game]” action.





**Game\_Board\_Data** = RECORD {id: INTEGER, player\_data: ARRAY OF userData, clients: ARRAY OF SOCKET CLIENTS, board: ARRAY OF ARRAY OF INTEGER, player\_turn: INTEGER}

Data: (username: STRING) OR ANY

Responses:

* [JOIN GAME - SUCCESS] - Data: **Game\_Board\_Data <without the client's property>**
* [JOIN GAME - WAITING] - Data: (username: STRING) OR ANY

#### [CANCEL GAME]

This will tell the server that the player no longer wants to join a game. If the client is in the Waiting Queue, the server will remove the client from the list and respond with a success action. Otherwise, the server will respond with a failed action.

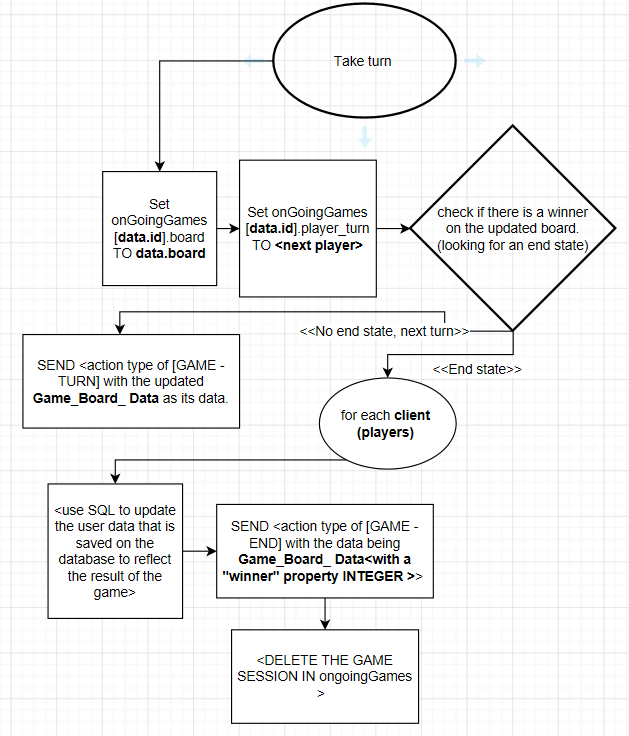
Data: (username: STRING) OR ANY

Responses:

* [CANCEL GAME - SUCCESS] - Data: "Cancelled"
* [CANCEL GAME - FAIL] - Data: "Not waiting for a game"

#### [TAKE TURN]

This action will be sent to the server every time a player selects a valid square on the gameboard. The server will then check if there is a win state on the board (3 of the same colours adjacent of one another or in a diagonal line) or a draw state (there is no winner and no actions can be played). if any of these states are true then the server will update each user data accordingly and end the game session with the end game action type. However, if the game hasn’t ended and is still playable, the server will respond with a turn action type with the updated game board and next player turn (e.g. if player one just played, now player two can take a turn) to both players so they can update their boards. A flow diagram is provided below that illustrates how this would work in code.



Data: **Game\_Board\_Data <updated game board, without the client property>**

Responses:

* [GAME - END] - Data: RECORD {winner: INTEGER, updated\_userData: **UserData**}
* [GAME - TURN] - Data: **Game\_Board\_Data <with the updated game board and player turn. Without the client's property >**

#### [GET ALL PLAYER STATS]

The client will want all player statistics to populate the leader board. The server will query the database for the User table to get all the player statistics (removing the hashed password). If successful, the server will respond with a successful action type with the data being an ARRAY of **UserData**. If there was an error, the server will respond with a failed action type with an error message as its data.

Data: (username: STRING) OR ANY

Responses:

* [GET ALL PLAYER STATS - SUCCESS] - Data: ARRAY of **UserData**
* [GET ALL PLAYER STATS - FAIL] - Data: "Error whilst getting all player statistics"

### Example usage: Pseudocode for authenticating users (OOP)

This will be how the client attempts to log in (authentication) by using a client socket and sending the byte representation of the “[LOGIN USER]” action with the client’s username and password to the socket server. The server will respond with its action in bytes which need to be unpickled, this will allow the client to know if their request was successful or ran into an error.

#### Client-side login Pseudocode code

FUNCTION **login\_client** (ARRAY OF STRINGS **user\_authentication\_data**) RETURNS BOOLEAN

# format for user\_Authentication\_Data will be a python tuple which is

# accessed like an array hence the reason I cast the data type as ARRAY

# OF STRINGS cause the only differences are that tuple values are

# immutable (can’t be changed). example (“username”, “password”)

# DECLARE VARIABLES

DECLARE **userAuthData** AS SocketActions INITIALLY NULL

DECLARE **packagedUserAuthData** AS STRING INITIALLY ‘’

DECLARE **serverResponseData** AS SocketActions INITIALLY NULL

IF **user\_authentication\_data** IS NOT VALID THEN

SEND “INVALID” & **user\_authentication\_data** TO DISPLAY

RETURN False

SET **userAuthData** TO **SocketActions** (“[LOGIN USER]”, **user\_authentication\_data**)

SET **packagedUserAuthData** TO <parse (pickle) **userAuthData** TO BYTES so it can be sent to server>

SEND **packagedUserAuthData** TO <socket server>

RECEIVE **serverResponseData** FROM <wait for server to respond, then unparsed (unpickle) the response BYTES to a **SocketActions** (data) > <socket server>

IF **serverResponseData**.action == “[LOGIN USER SUCCESSFUL]” THEN

SET THIS.**userData** TO **serverResponseData.data**

RETURN True

ELSE

RETURN False

END IF

END FUNCTION

#### Server-side login Pseudocode code

SET **client\_action\_document** TO < Received the pickled “[LOGIN USER]” action from an unauthenticated client socket and successfully unpickled the bytes into a python “Action” object>

IF **client\_action\_document.action** == “[LOGIN USER]” THEN

SET **user** TO **login\_manager(client\_action\_document.data)**

IF **user.result** == False then

# there was an error or incorrect username/password. Send the error message to the client

< respond to the client socket with an action type of "[USER LOGIN - FAIL]" and data being the value of **user.data**>

Else

# successfully authenticated credentials

<Respond to the client socket with an action type of "[USER LOGIN - SUCCESS]" and the data being the value **of user.data**>

<add the client socket to a list of authenticated clients as a RECORD with the userData from the database being the value of **user.data**>

# display a message of saying a newly authenticated user

SEND “Accepted new connection from: ” & user.data.username TO DISPLAY

END IF

END IF

FUNCTION **login\_manager**(ARRAY OF STRINGS **userCredentials**

) RETURNS RECORD {result: BOOLEAN, data: STRUNG or userData}

TRY:

SET **userCredentials\_fromDB** TO < use the SqlServerConnetion class to query the database for the username and hashed password of where username = **userCredentials[0]** (thuis is the username that is trying to authenticated)>

IF **userCredentials\_fromDB** IS NONE THEN

# there is no user with that username in the database

RETURN RECORD {result: FALSE, data: "No user found with the username: “ & (**userCredentials[0]**)}

END IF

SET **hashPassword** TO <MD5 hash userCredentials[1] (this will the the password provided to login)>

IF **hashPassword** == **userCredentials\_fromDB.password** THEN

# the client socket can be accepted and authenticated

SET **userData** TO < use the SqlServerConnetion class to query the database for the username, wins, losses and number of games played where username = userCredentials[0]>

RETURN RECORD {result: TRUE, data: **userData**}

END IF

CATCH:

# THERE WAS AN ERROR WHEN PERFORMING THE CODE ABOVE because the program couldn’t query the database or something unexpected happened.

RETURN RECORD {result: FALSE, data: " Error when authenticating client's account.”}

END TRY

END FUNCTION

# Implementation

In this section of the document, I will be going over different sections of the program I developed, concepts I had to learn, integrate and some reasonings as to why I programmed sections the way I did. This application is all object orientated, so I will try to keep all the indentations to keep things easy to follow.

## Researching

### Python Sockets & pickling python objects

Sockets allowed me to connect 2 nodes on a network. I need to use sockets to create a connection between the players and the server. This will then allow me to update both players that are in the same game session and allow them to play against each other on different clients ( and computers).

The websites I used to learn everything that I needed to know was from ( <https://pythonprogramming.net/sockets-tutorial-python-3/> ) which provided video and written tutorials on how to build your first application (chat app) with python sockets. I used the information and the examples from this website to create this online tick tack toe game.

#### Creating a socket server and client

The tutorial doesn’t take advantage of object-oriented programming as I did, however, the concept is still the same.

##### Server socket set up



A close up of a screen

Description automatically generatedA screen shot of a social media post

Description automatically generated

Ps, when hosting on a server you need to set **socketHostData** TO (““, PORT). This tells the server to connect to any available IP (the server IP). This allows connection from different computers that are on different networks to connect to the server.



The server binds itself to host address and listens for any incoming data from its stream. When the server receives any data or socket connection request they go through the **\_action\_handler** function.

A screenshot of a cell phone

Description automatically generated

The example uses the **select** library which is just an efficient way to iterate and monitor many connection and requests to the server at scale. I also used the **select** library because it gave me a clear structure and flow to handling incoming requests to the server. I create an indefinite loop that will handle any incoming data to the server socket. The incoming data will be sent by either an authenticated client or unauthenticated client (I talk about this process in the “handling unauthenticated clients” section.) that’s why I check if the **user\_socket**  is equal to the server socket if so the client is unauthenticated and will be limited to attempt to log in or create a new user account.

A screenshot of a cell phone

Description automatically generated

If the **user\_socket** is in **self.sockets\_list** that means that they are an authenticated client and is allowed to use the other functionalities. When an action is sent by an authenticated client socket the server will do the following:

* Check if it’s a closed connection request (so I can remove that player’s connection from the server). This will free up space and makes the application robust and scalable.
* Depackage the incoming data into an action object (I talk about this in the “Packaging and Depackaging actions” section)
* Checks if the action type that has been sent is defined in the constructer. This just makes the code more robust as the server will not process any unknown action types. A screen shot of a social media post

  Description automatically generated
* Log the action that has been sent and the user that sent the action. This comes in useful when debugging what's being sent by the client.
* If the action type sent is defined then the server will process the action on a new thread (I talk about “\_thread” later in the research section)

##### Client socket set up



A screenshot of a computer screen

Description automatically generated

The client then needs to connect to the socket server. The client socket doesn’t get accepted straight away when they connect, they will need to send the “[USER LOGIN]” action as I spoke about in the design section.

#### Sending data between sockets

I used the buffering method that was shown in the examples <https://pythonprogramming.net/pickle-objects-sockets-tutorial-python-3/?completed=/buffering-streaming-data-sockets-tutorial-python-3/>.This allowed me to send dynamic stream lengths. Usually, you would have a fixed buffer size which is the number of bytes that will be read, since I will be sending dynamic actions objects this was a perfect solution for doing that.

A screenshot of a social media post

Description automatically generated

I then send over the **packaged\_action** variable over the sockets. After retrieving the action bytes from the stream. I will load the data by doing the following

A screenshot of a cell phone

Description automatically generated

As you can see, the action object is fully usable by the receiving node. I will be using this method to send data between sockets.

### concurrent programming in python

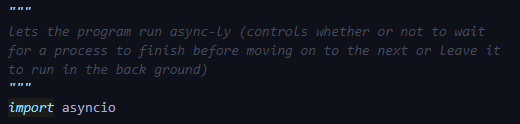
Concurrent programming is essentially the ability to perform more than one command at the same time. I already have some experience in writing concurrent so I will just be jumping in use cases examples.

#### asyncio

When the player sends an action to the server it may take some time for the server to respond. The client may need to wait until it receives a response before moving on to the next line of code. This is when we can use **asyncio** to tell the program to wait (**await**) for a function to complete before mobbing on to the next line of code. This allows us to control the flow of command execution. Whenever a client socket sends an action to the server it will need to **await** the server's response.

A screenshot of a cell phone

Description automatically generated

We can only use the **await** keyword inside an Asynchronous (**Async**) function, which we can do by adding **async** before the **def** keyword when defining a function. As you can see, I **await** the **rec\_doc\_manager** function because the server takes time to respond. By doing so there will always be some data in the result variable as it is relied upon by the next lines. 

A picture containing bottle

Description automatically generated

When rendering the leader board I need to get all users statistics data before placing the widget to the screen. That is why I call an **async** function called **get\_all\_userData\_sorted** that will do this before rendering the widgets. You can only call **async** functions in a coroutine which can be created with the **async.run** function as I did above.

A screenshot of a cell phone

Description automatically generated

Inside the **get\_all\_userData\_sorted** function, we await on the results of the **getAllPlayerData** function which we looked at above. We **await** it to finish because the next line then checks if we have successfully retrieved the data from the server. If we didn’t await the **getAllPlayerData** function, it would complete in the background, but the program would have already done the check and concluded False by the time the server responded and the data has been sorted.

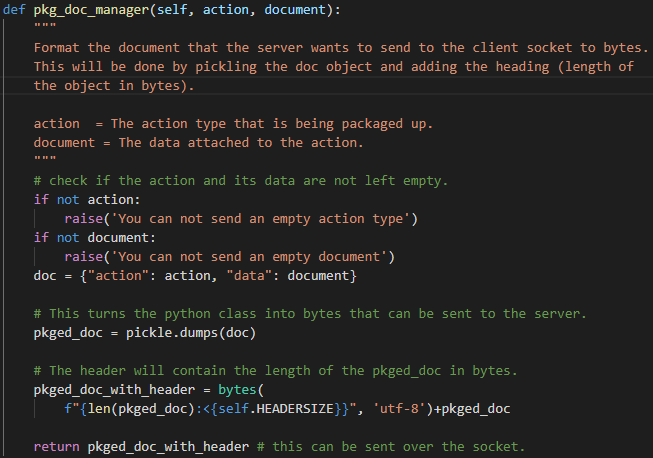
#### \_thread

Threading allows me to create another process to carry out a task concurrently. I used **\_thread** on the server when processing a valid action. This is because I don’t want a single action blocking other requests being carried out. For example, without processing request off the main thread - if a player is waiting to join a game, the server's main thread will be blocked as it will be stuck in the waiting loop. Other players action won't be processed, this is what I call “blocking” and was an issue that I encountered. Hence why I create a new thread whenever an action is being processed on the server. This method doesn’t block the main thread.A close up of a screen

Description automatically generated

## Packaging and depackaging actions

A difficult section was creating a function that would pickle and unpickle the Action object (into bytes and from bytes). Since you can only send bytes over the socket connection this would allow me to send and receive the action object over the sockets to both clients and server.

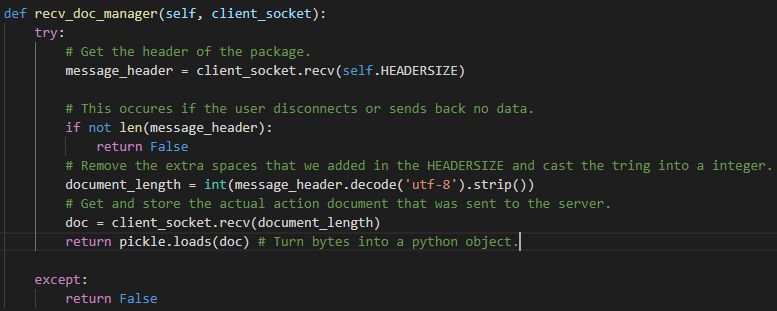


I used the **pickle** library to achieve the parsing of the python object into bytes. Then I got the length of the bytes as an integer and prepended the length with the size of **HEADER SIZE** (eg. “512\*\*\*\*\*\*\*”, eg2. “12345\*\*\*\*\*”. The size of the header will always be fixed) to the action object (in bytes.).



I included a **HEADER SIZE** (of 10 bytes) because that would contain the length of the action. The receiving node will be able to read the first 10 bytes to get the full length of the action, This will tell the receiving node how much more bytes are part of the action that is being sent,

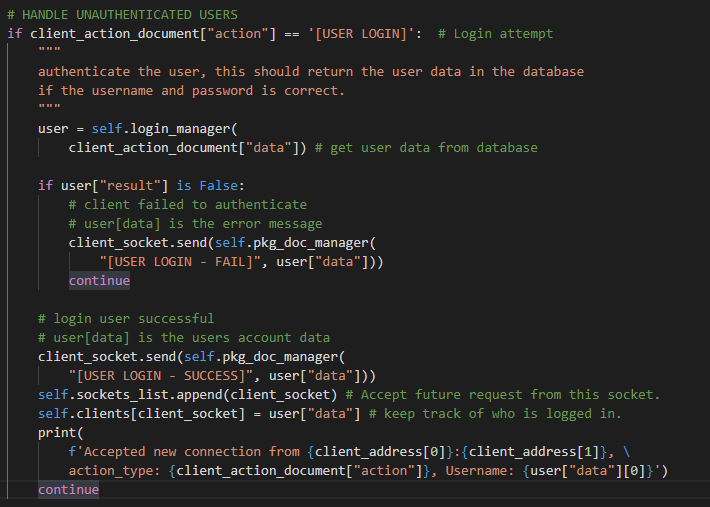
then read in the next bytes to get the full action document in bytes.



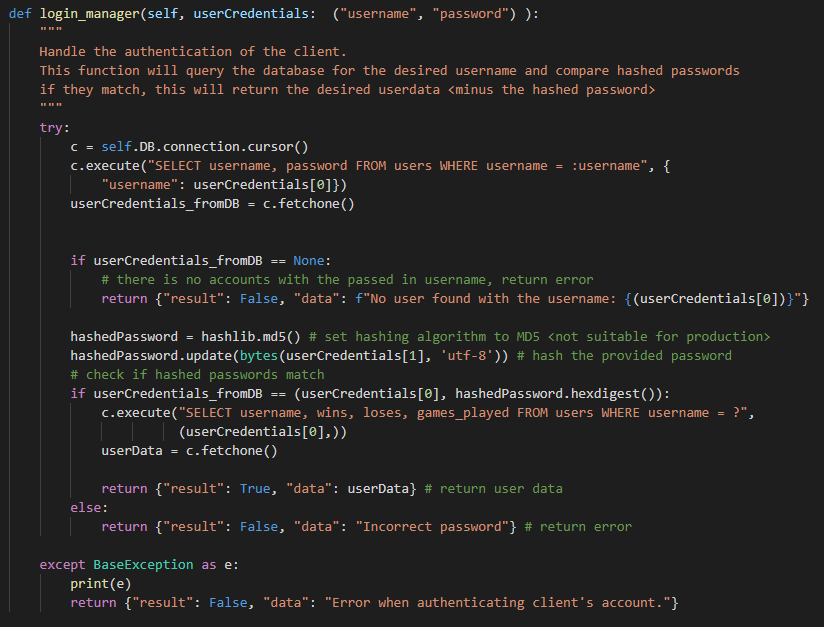
## Handling unauthenticated clients

When an unauthenticated client sends a request to the server, it will get turned into an action object (**client\_action\_document** variable) and be filtered one of 2 action types: “[USER LOGIN]” or “[USER REGISTER]”.

### [USER LOGIN]

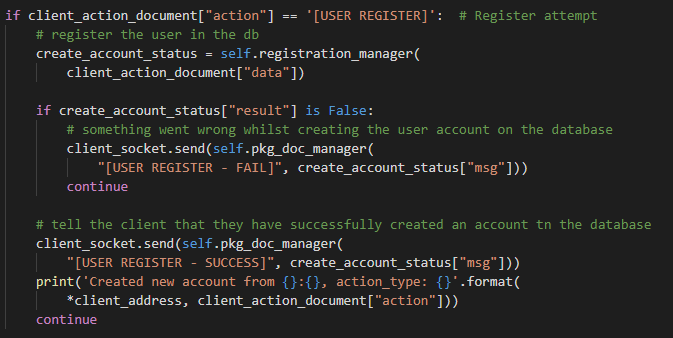


The **login\_manager** function is responsible for querying the database and checking if the password sent matches the hashed password saved in the database for that username.

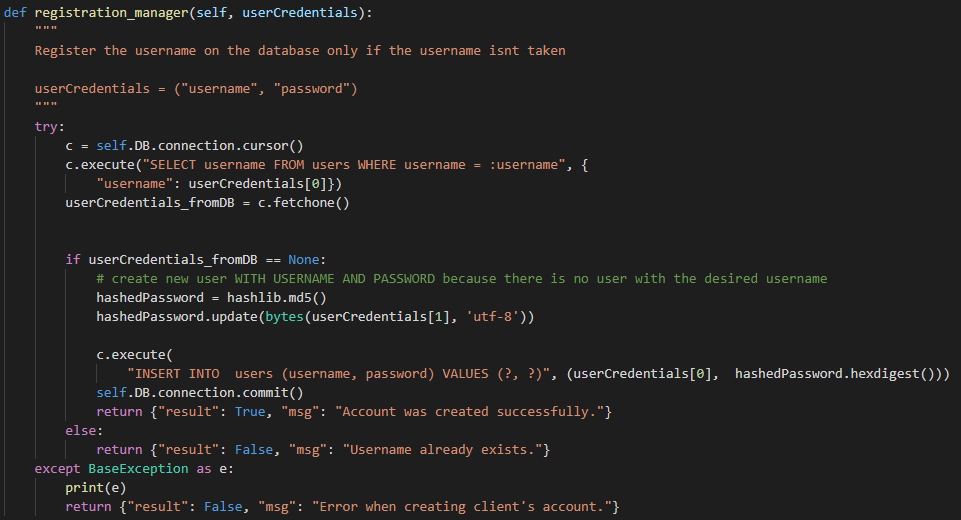


### [USER REGISTER]

This sequence works as I mentioned in the “design – action types - user register” section of this document.



The **registration\_manager** function is responsible for creating the new user data in the SQLite database.



## Issue Log:

* I decided to tackle the client and server communication section first, so, I began researching how to set up a sockets server that will listen to requests. However, I ran into a problem where the server didn’t know how to handle the incoming data. That was when I implemented the “Action” object and learnt about pickling Python objects. Now sending data between the client sockets and server socket was predictable and makes handling received/sending data much easier.
* When I was creating the front-end I was having difficulty switching the screens, that is when I implemented the **switch\_to\_frame** function that would bring the frame that you are switching to, to the front then removing the previous frame. By doing this I was able to easily switch views if a button was clicked or a condition state (eg. log in view to home view).
* Another issue was presented when I was working with dynamic views (for example: updating the leader board after getting the data from the server). I needed a way to update the whole frame. My solution was to add a **render** function to each view frame. This allowed me to re-render the view every time I wanted to rebuild the view. I also made each view re-render every time the view is switched to (**switch\_to\_frame**).A screenshot of a cell phone

  Description automatically generated

<https://pythonprogramming.net/change-show-new-frame-tkinter/>: I used this write up to come to my solution.

## Testing different scenarios: User Interface

My UI closely matches my designs.

Commands used to start up the client and the server. The programs need python 3.7 or higher to run. If you use python 2.7 the programs will run into errors when using string manipulations via f strings. This is a python 3.7 feature.

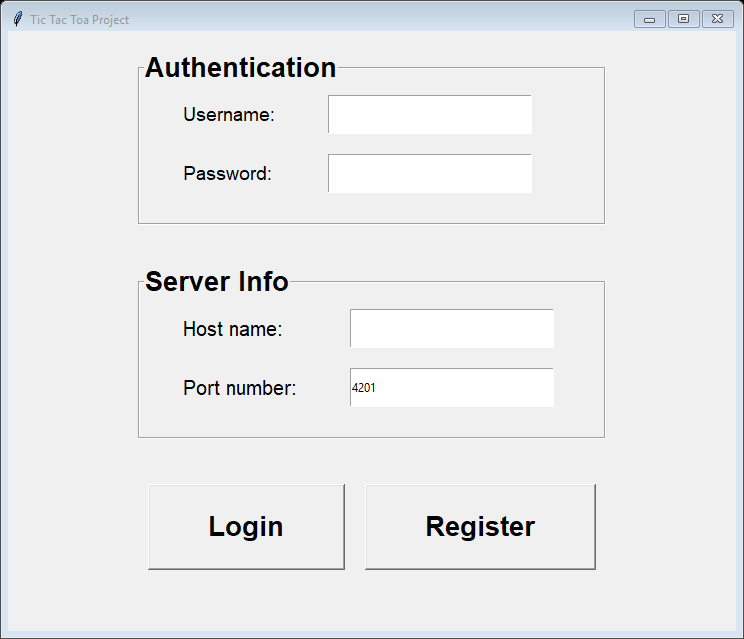




I ran this application on a school computer, the hostname of the server is EDU-D004872 and it is running on port 4201

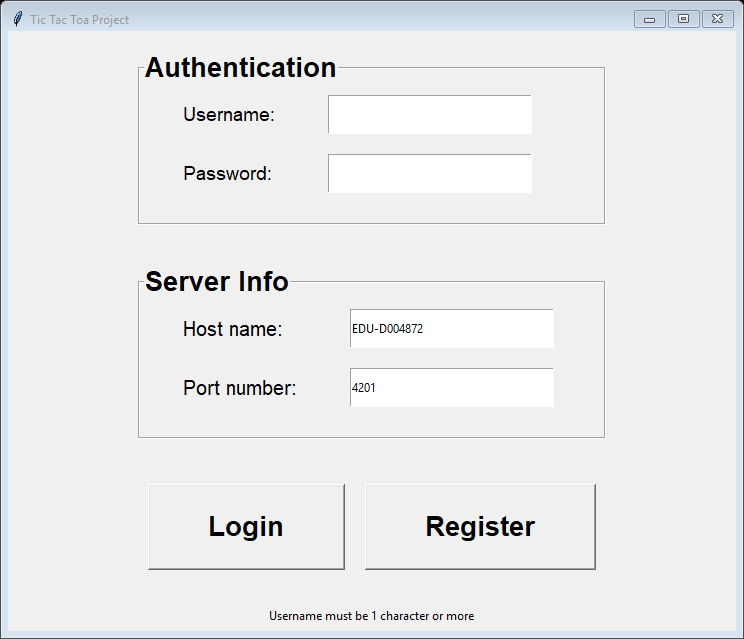
### Authentication page

This is the first page that every player will see. They are required to provide their account username, password, the address of the server and the port the server is running on to log in or register a new account. This closely matches my UI design.



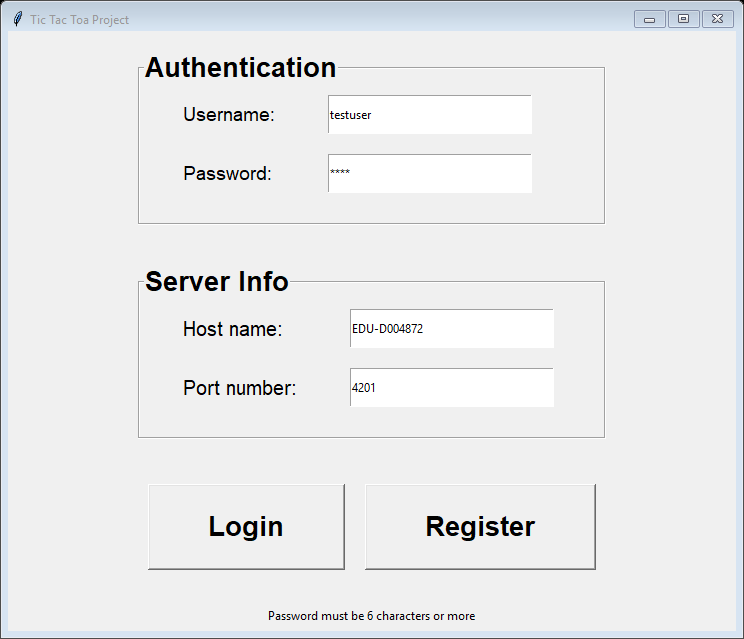
### Incorrect inputs test

I tried registering and logging in an account with no username input, this should result in an error.



The client program successfully picks this up and tell the user that there must be at least 1 character in the username input field.

Next, I tested registering and logging in an account with less than 6 characters in your password, this should also result in an error.

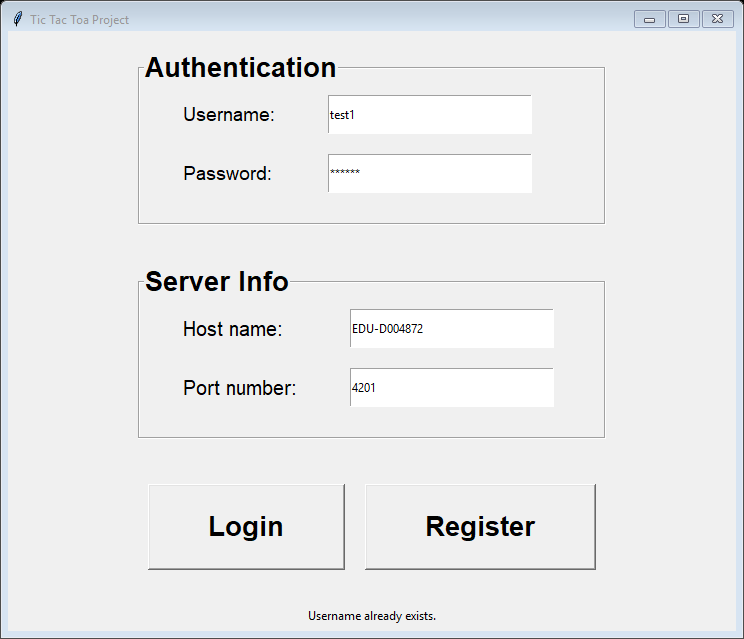


The client application also catches this error and tell the user that their password must be 6 or more characters long.

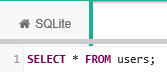
### Register button Testing

#### Attempting to register a username that already exists in the database

This should result in an error as usernames are unique to each player.



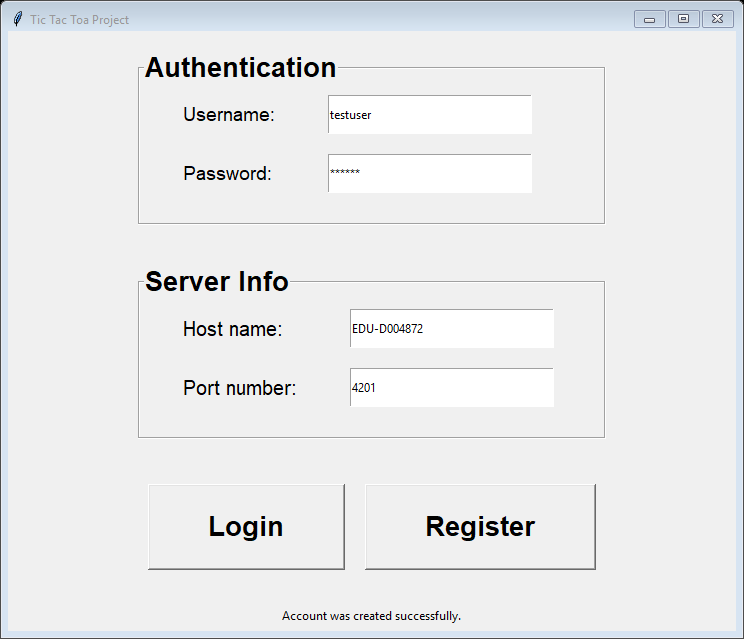
The server responds with the “username already exists” error. If we take a look at the SQLite file, we can see that there is already a username of “test1” in the database. (I used <https://sqliteonline.com/> to view and debug my SQLite file )





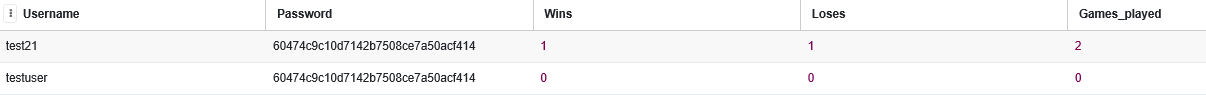
#### Successfully registering a user with a unique username.

This should result in a new entity created in the user table with my provided username, hashed password and default player statistics.



In this test, I created the new user account with the username of “testuser” and password of “test12”. As you can see the new user is saved in the SQLite database with the default statistics and hashed password.

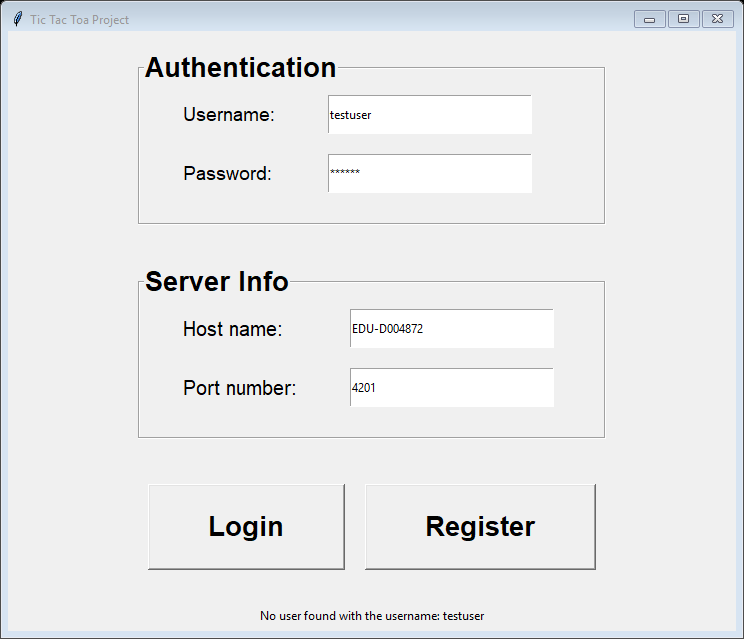




### Login button

#### Attempting to login with a username that doesn’t exists

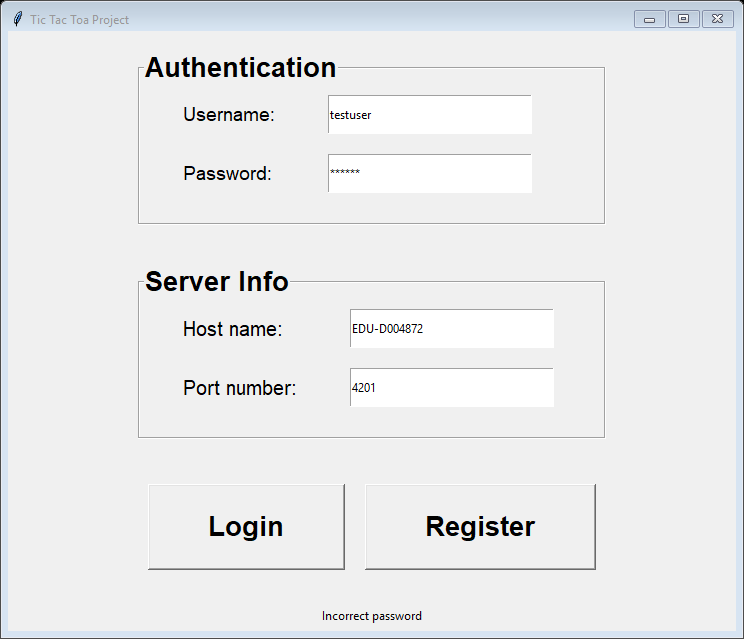
This should result in an error as there is no user with the provided username in the database.



Before creating the account I attempted to login to the “testuser” account which didn’t exist. The server responded with the “No user found with the username: testuser” error which gets displayed to the user.

#### Attempting to login with the wrong password for a username that exists.

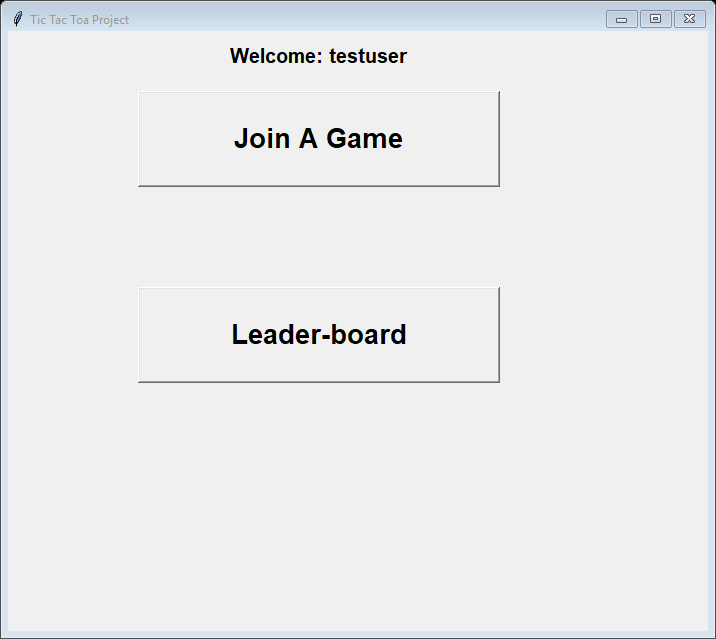
This should result in an error because to be authenticated, the password that you send to the server must match the hashed password in the database that is linked to the username when your provided password is hashed.



The server successfully identifies that the password provided hashed doesn’t match the hashed password associated with the username. The server responds with the “Incorrect password” message which gets displayed to the user.

### Successfully providing the right password with a username that exists in the database

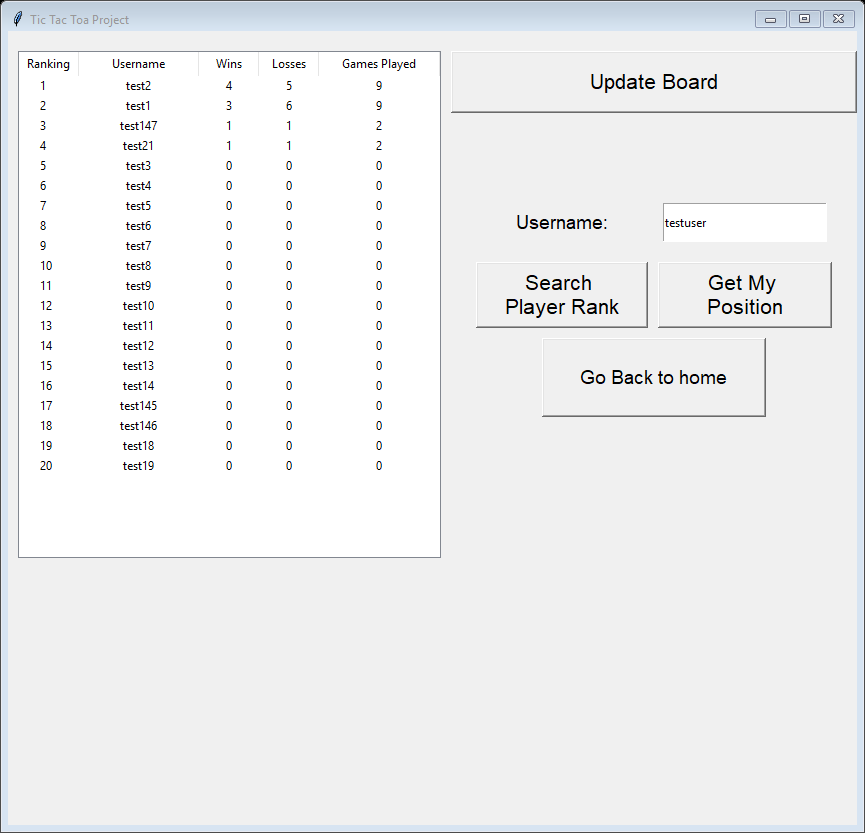
This should successfully authenticate the player and switch the screen to the home page.



The player’s client socket is now successfully authenticated (accepted) by the server socket. Any further request from that client socket will be registered as “testuser”

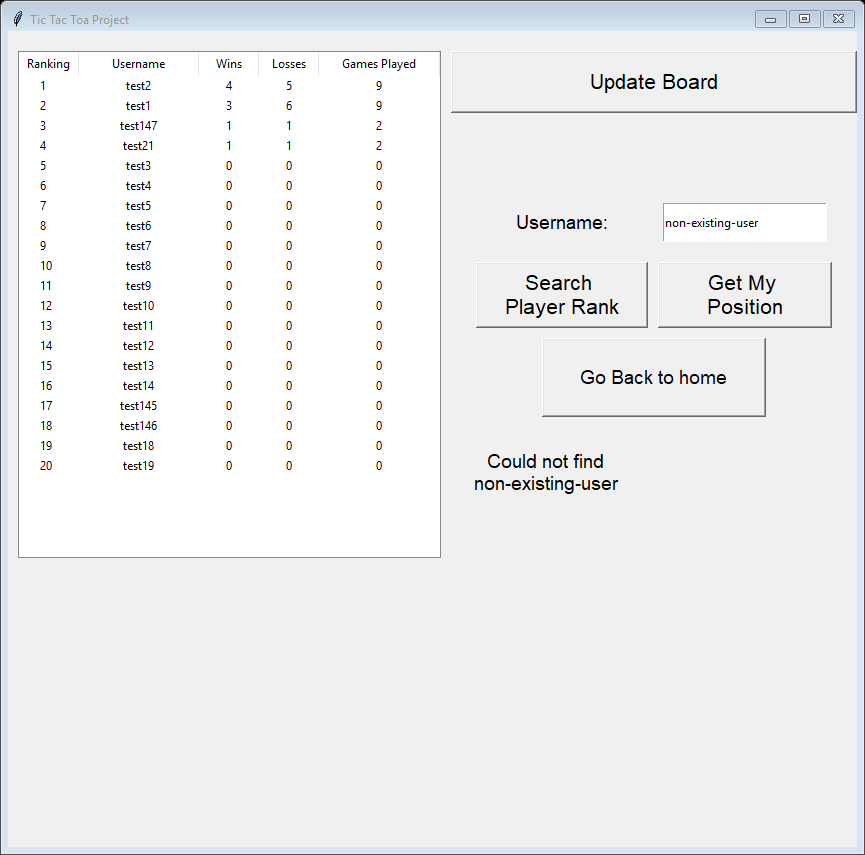
### Leader board page

When clicking on the leader-board button the player is presented with the leaderboard page. This closely matches my design as well. As required, only the top 20 players ( with the highest wins) are displayed on the leader board.



#### Searching for the rank of a player that doesn’t exists

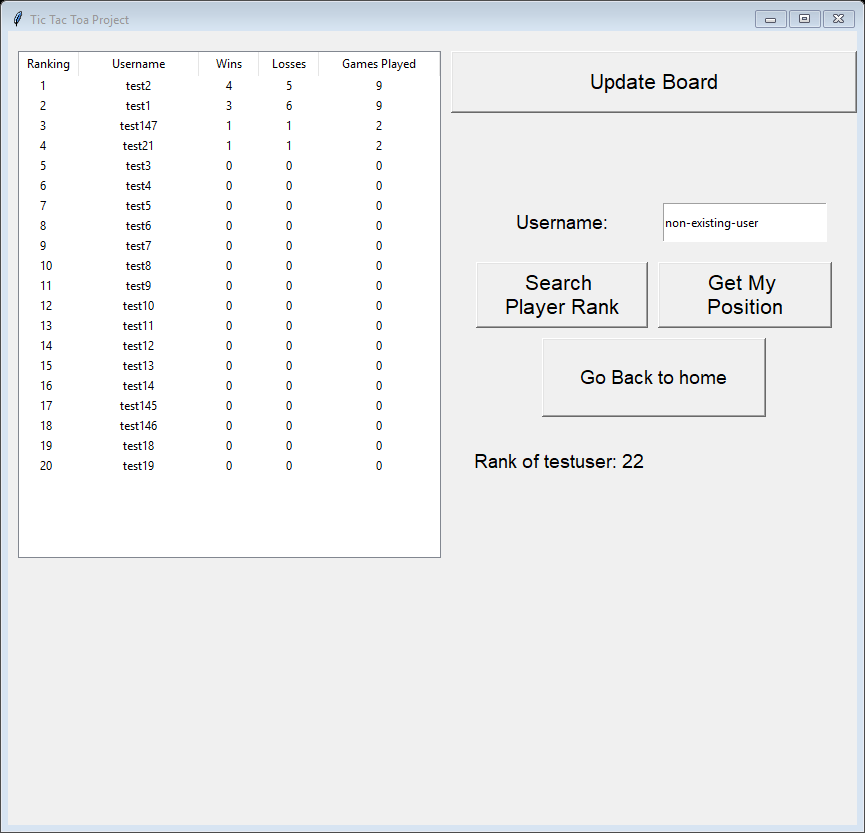
This should result in an error as the program won't be able to find the desired username in the sorted array of all players statistics.



The program displayed the message “Could not find non-existing-user” to the player. This means that it couldn’t find the player’s and told the player.

#### Searching for my rank

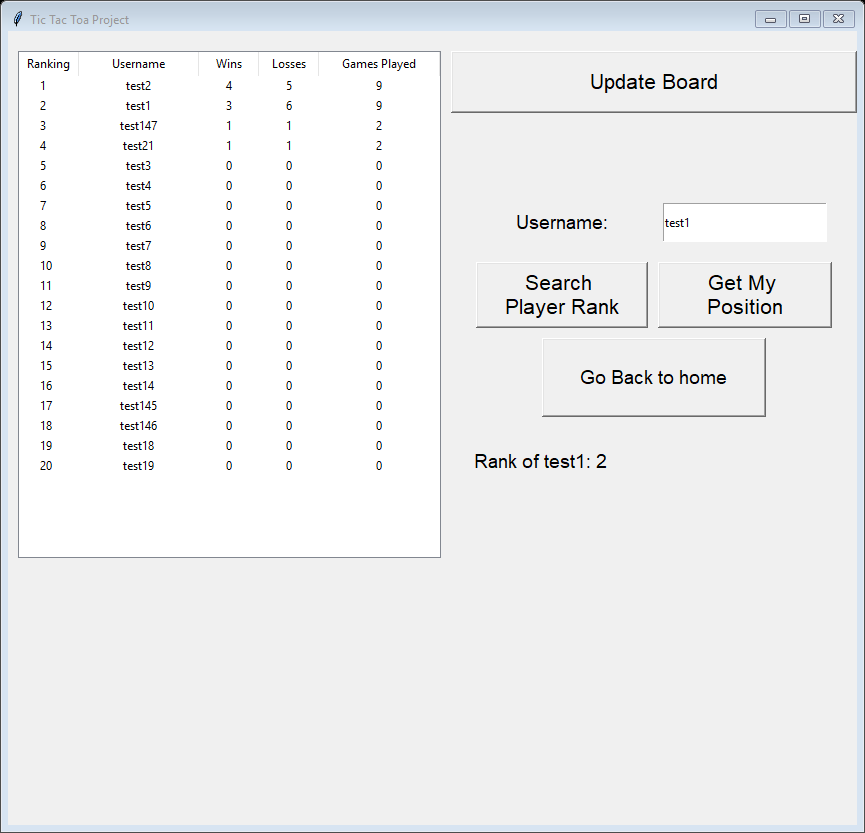
This should result in the player receiving their rank amongst all existing players.



The program successfully displayed the current players rank.

#### Searching for another players rank

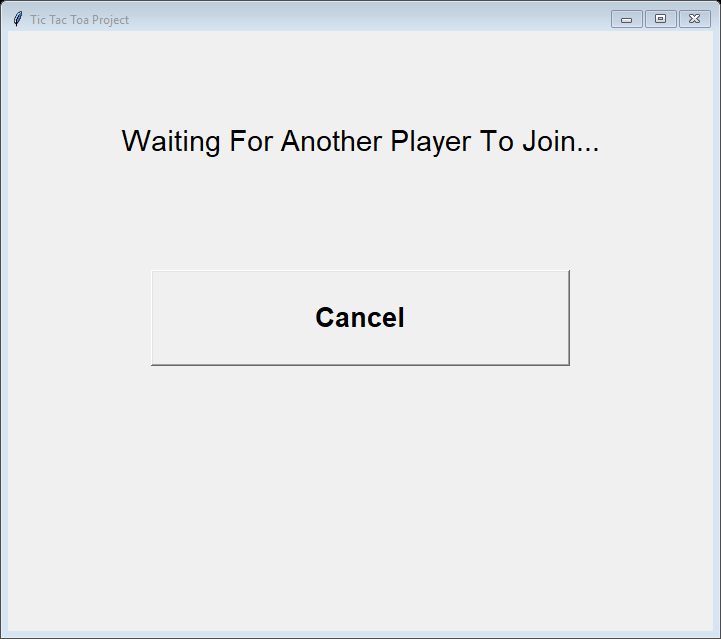
This should result in the player receiving the rank of the desired username amongst all existing players if the username exists.



In this test, I want to get the rank of the username “test1” who is ranked 2 with 3 total wins. The program successfully returns “Rank of test1: 2”

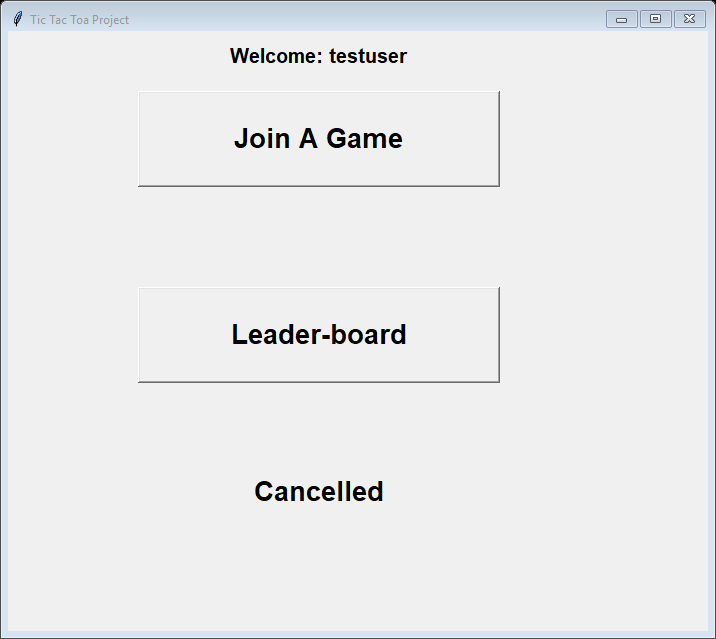
### Joining a game loading page

When the player clicks on the “play a game” button they get presented with the “joining a game” page. The player is now in the waiting queue on the server. The logic is the same in my design. Waiting for the server to respond with an opponent to play against is processed on another thread, this allows the player to still cancel while waiting in the queue.



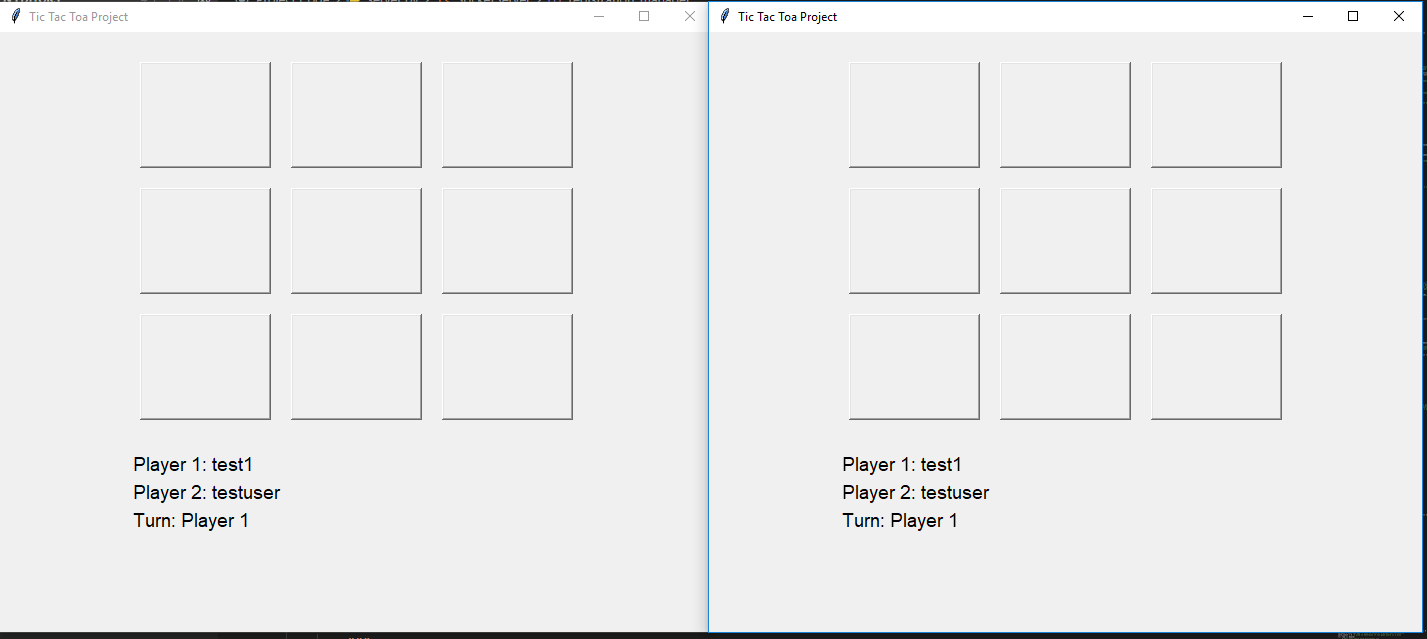
#### Clicking on the cancel button.

This removes the player from the waiting queue on the server. The server handles for this by always checking if the player is still waiting to join a game when they are waiting.



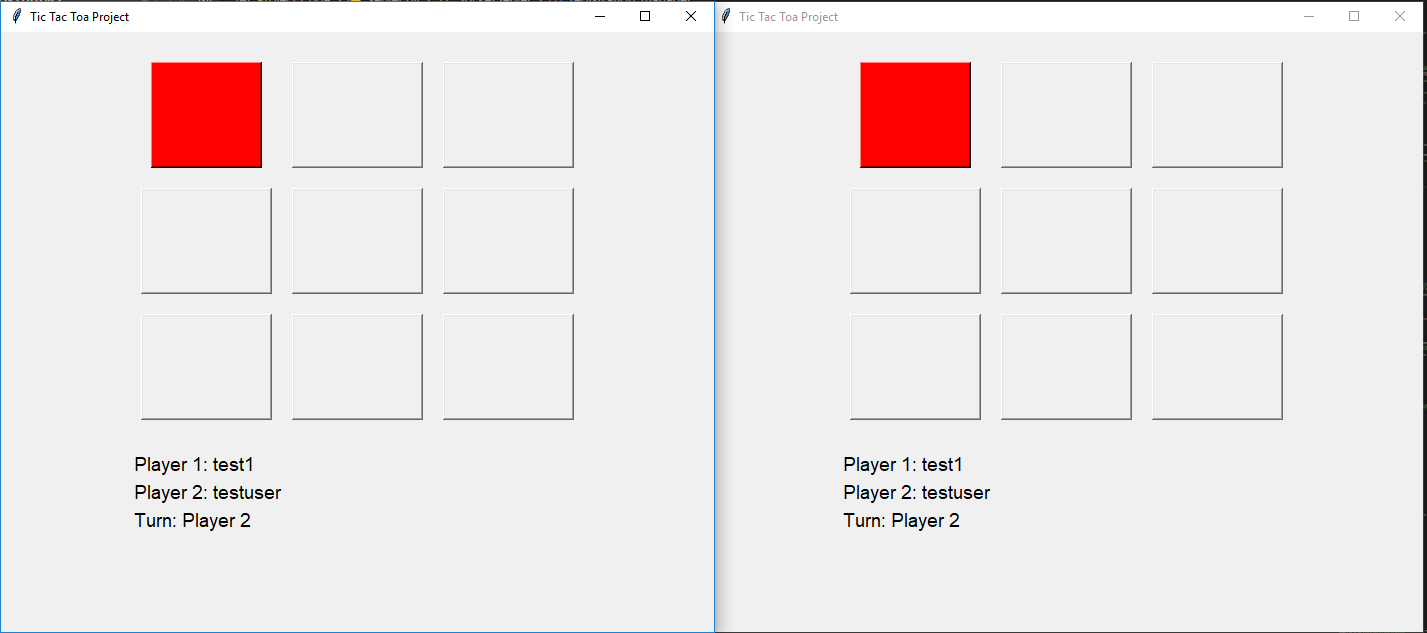
### Successfully getting 2 players in a game

When 2 players are waiting in the queue the host request will create the game data and notify both players that the game has started . with this data the client program creates the UI below that matches my design

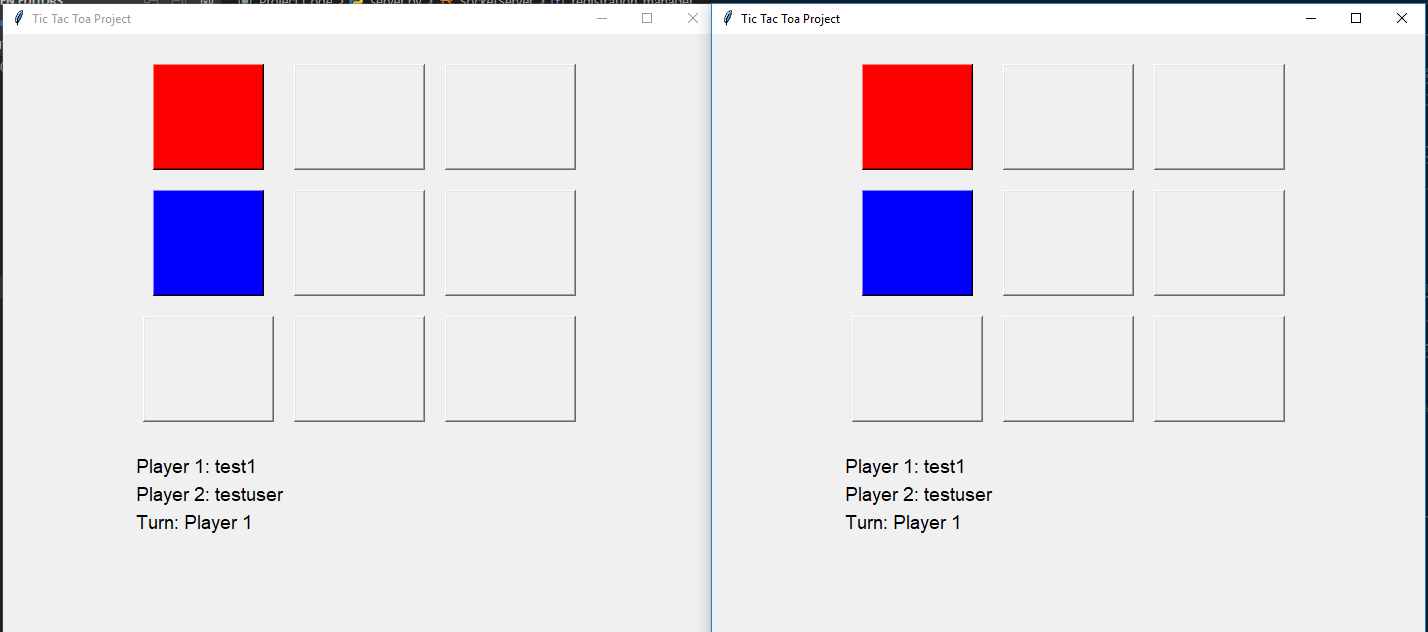


#### Players taking their turns

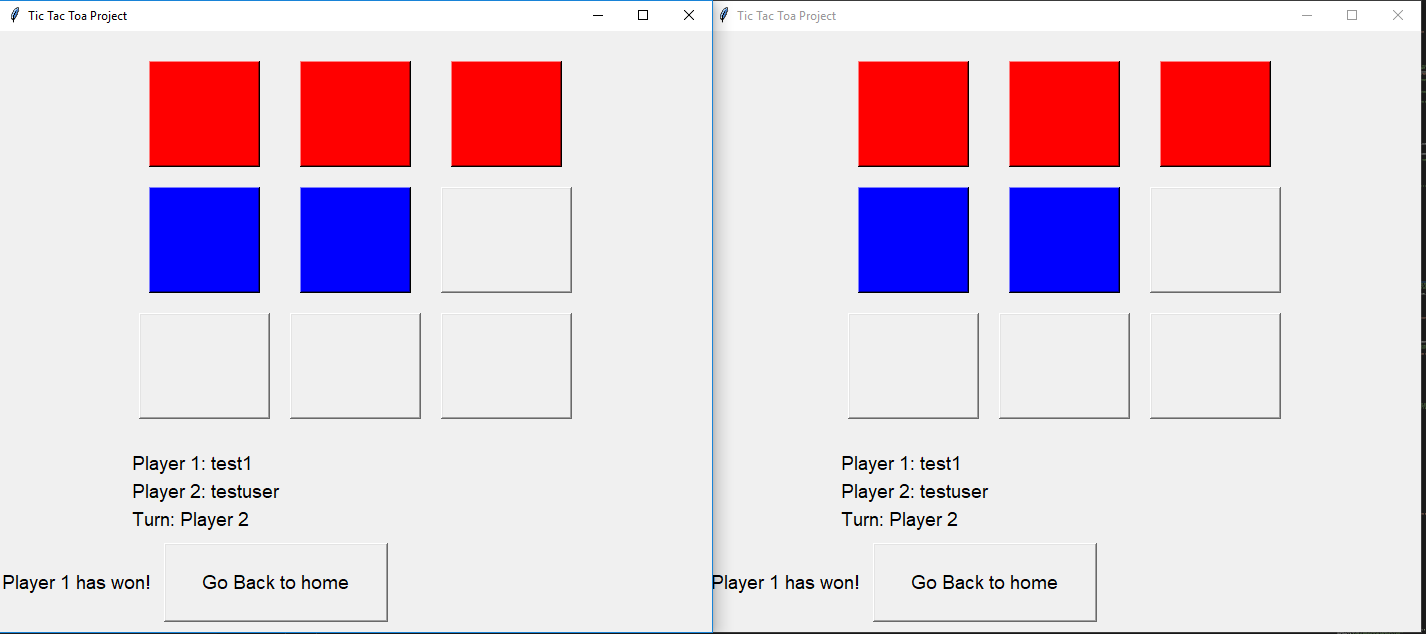
Player 1 takes a turn.



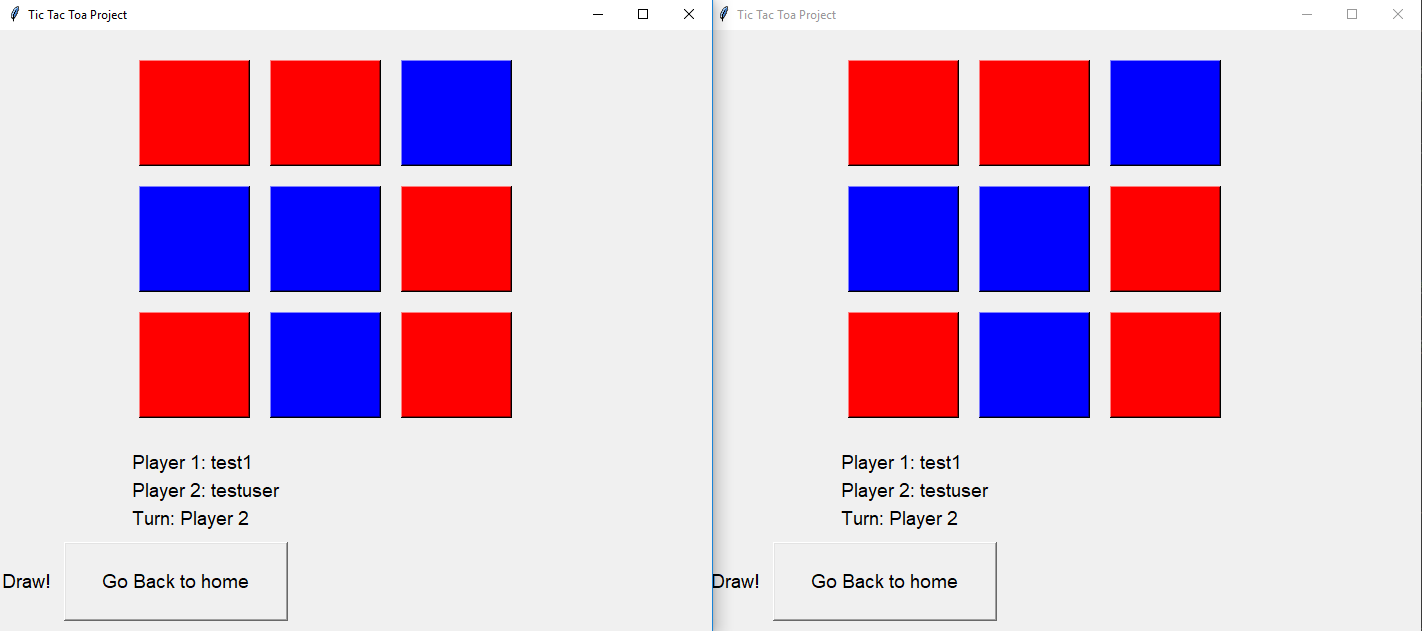
Player 2 takes its turn.



Player 1 wins.



The game comes to a draw.



Each player account statistic gets updated after the game reflecting the result of the game. For example, since the player “testuser” lost a game and drew the next game its statistics in the database are now: 0 wins, 2 loses 2 games played.

A screenshot of a social media post

Description automatically generated

### Multiple server connections 6

I tested if the server was able to handle 6 connected clients playing 3 different games.

A screenshot of a social media post

Description automatically generated

The server successfully matched each player into a game and can keep track of all 3 games at the same time.



The server was also able to determine the end state for all 3 game boards successfully.

A screenshot of a social media post

Description automatically generated

Lastly, the server was able to accurately update each player data in the database successfully.

# Evaluation

Since all my tests were all successful, I believe that the application that I wrote fully satisfy the specification requirements:

* Players interact with a clear and easy to use user interface with input validation.
* Players can register an account and authenticate their client. This provides a level of security for the players and the database.
* Players passwords are hashed in the database.
* The server and client utilize python sockets to communicate.
* Additionally, the server makes use of concurrent programming and threading to handle the minimum required clients of 6, this also allows the server to handle more players.
* Players can play a game of tick-tack toa and get their statistics updated at the end of the game. This makes playing the game fun and has a stake attached to each game.
* Players can view the leader board, which shows the top 20 players in a list. This makes use of the insertion sort to sort players by numbers of games the won descending order.
* Layers can also search their ranks and other player ranks.
* Starting up the server is easy and is easily portable.

I fully documented my code and added small descriptions to each function. The code that I wrote is object-oriented and modular. This makes updating or changing sections of my code clear and easy. I also provide a README.md file and some bash scripts to easily get end-users started. By doing all of this, makes my code more maintainable.

When writing my SQL queries, I was also thinking about possible SQL injections. I used built-in character escapes making my SQL statements safer. Additionally, I established strong input validation on the client-side and the programs account for and handle errors when they occur. These make my code more robust.