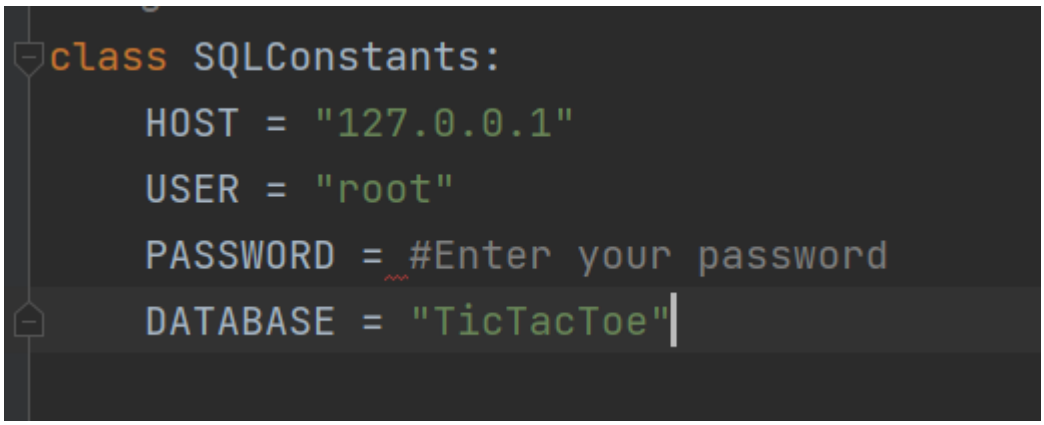
Shimon Ben Ami, ID: 215489949

Operating system: Windows

Code language: Python 3.9

To operate the game, you need to create database with the name of “TicTacToe” and update your PASSWORD constant in sql_constants.py file.



After you do that all that left is to run the server and after that run many clients as you want.

****There is a file of the encryption and decryption that was added to classes. I tried to use without success, so I added this class for you to see that the class of the encryption and decryption indeed implemented. (I couldn’t debug the code because of the problem I told you about in the email).**

Server.py

The **server.py** file defines several functions and a class related to the operation of a Tic-Tac-Toe server. Here's a summary of each:

Functions

- **__init__**: Typically, a constructor in a class. It initializes a new instance of a class. In this context, it's likely part of the **TicTacToeServer** class, setting up initial server configurations, database connections, or encryption settings.
- **load_all_data**: Likely loads game or player data from a database or external source to initialize the server with necessary information at startup.
- **encrypt**: Utilizes encryption algorithms to securely encrypt data before sending it over the network.
- **decrypt**: The counterpart to **encrypt**, this function decrypts received data that was encrypted by the sender.
- **handle_client**: Manages communication with connected clients, handling requests, commands, or data transmission.
- **get_available_games**: Returns a list of games available for joining, possibly filtering by those that are not yet full or have started.
- **get_all_available_games**: Similar to **get_available_games**, but may include all games regardless of status.
- **create_game**: Handles the creation of a new game session, including setting up necessary data structures, selecting initial players, etc.
- **set_num_players**: Configures the number of players for a game, which could be important for games supporting variable player counts.

- **check_win**: Evaluates the game board to determine if a player has met the conditions for winning the game.
- **check_tie**: Checks the game board to determine if the game has ended in a tie, meaning there are no further moves possible that could lead to a win.
- **get_next_player**: Determines which player's turn is next based on the current game state.
- **is_players_turn**: Checks if it is the specified player's turn to make a move in the game.
- **notify_spectators**: Sends updates about game progress to all connected spectators, possibly including moves made, player turns, or game outcomes.
- **remove_player_from_game**: Removes a player from an ongoing game, which could be due to disconnection, forfeit, or other reasons.
- **send_game_board_to_all_clients**: Broadcasts the current state of the game board to all connected clients, ensuring they have the latest view of the game.
- **broadcast_to_all_clients_in_game**: A more general function that sends a given message to all clients connected to a particular game, used for various types of game-related communications.
- **start_server**: Initiates the server, setting it up to accept incoming connections and start handling client requests.

Class

- **TicTacToeServer**: This class likely encapsulates all the server functionality, including starting the server, managing client connections, handling game logic, and communicating with the database. The functions listed are probably methods within this class, contributing to its overall operation.

Each function is designed to handle a specific aspect of server operation, from managing game sessions and player interactions to ensuring secure communication. The **TicTacToeServer** class serves as the central hub for all these activities, orchestrating the flow of data and interactions between clients and the server.

client.py

The **client.py** file contains several functions and classes that seem to be related to the client-side operation of a Tic-Tac-Toe game. Below is an explanation of each function and class found in the file:

Functions

- **__init__**: A constructor function, likely part of the **TicTacToeClient** class, initializing client instance variables, setting up the connection to the server, or preparing the GUI.
- **create_authentication_window**: Sets up a GUI window for user authentication, including fields for login and registration.
- **on_closing**: A handler function for window close events, ensuring proper disconnection and cleanup when the user closes the application.
- **register**: Handles the registration of a new user, sending user details to the server for account creation.
- **login**: Manages user login, authenticating with the server using provided credentials.
- **show_game_options**: Displays a menu or interface for game options, such as creating a new game, joining an existing game, or observing a game.
- **create_game**: Sends a request to the server to create a new game session, including any specified game settings.
- **show_available_games**: Displays a list of games available for joining, fetched from the server.
- **get_available_games**: Requests a list of available games from the server for the user to join.
- **join_game**: Handles the process of joining an existing game as a player.

- **join_game_observer**: Similar to **join_game**, but for joining a game as an observer, allowing the user to watch the game without participating.
- **join_observer**: Possibly an alternative or helper function for **join_game_observer**, specifically focused on the observer role.
- **exit_game**: Manages the process of leaving a game session, notifying the server and updating the UI accordingly.
- **close_gui_windows**: Closes all open GUI windows, typically used during application shutdown or logout.
- **send_data**: Sends data to the server, such as game moves or chat messages, using the established connection.
- **receive_data**: Receives data from the server, blocking or asynchronous based on implementation, and processes the received data.
- **periodic_update**: Might be a periodic polling or update function to refresh game state, check for messages, or update UI elements.
- **show_game_window**: Displays the main game window, with the game board and controls for making moves.
- **make_move**: Sends a move made by the player to the server for processing and updates the local game state/UI accordingly.
- **listen_for_server_messages**: A continuous loop or asynchronous task that listens for messages from the server and handles them.
- **process_server_message**: Processes messages received from the server, updating the game state, UI, or other elements based on the message content.
- **update_game_board_ui**: Updates the game board's UI to reflect the current game state, including player moves.
- **display_game_state**: Displays the current state of the game, including player turns, scores, and win/tie notifications.
- **update_game_state_after_join**: Updates the client's game state after joining a game, syncing with the server's state.

Classes

- **Move**: Likely represents a game move, including details such as the player who made the move and the move's location on the board.
- **Player**: Represents a player in the game, potentially including attributes like the player's name, score, and current game state.
- **TicTacToeClient**: This class encapsulates all client functionality, including connecting to the server, handling user input, displaying the game and authentication GUIs, and managing the game state.

These functions and classes collectively form the client-side application for a Tic-Tac-Toe game, handling everything from user authentication and game session management to displaying the game UI and communicating with the server.

EncryptionManager.py

The `EncryptionManager` class provides a simplified interface for encrypting and decrypting messages, utilizing the `cryptography.fernet` module for secure cryptographic operations. This class is particularly suited for applications requiring data confidentiality, such as messaging apps or data storage systems. Below is an overview of each function within the `EncryptionManager` class, designed to give a quick understanding of its capabilities and processes.

Functions

`__init__(self)`

- **Purpose**: Sets up an **`EncryptionManager`** instance, initializing the encryption key and cipher for later use. This step is critical for preparing the class to perform encryption and decryption tasks.
- **Process**: Assigns a predefined key to **`self.key`** and initializes the Fernet cipher with this key, storing the cipher in **`self.cipher`**.

`encrypt_message(self, message)`

- **Purpose**: Encrypts a plain text message, turning it into a secure, unreadable format. This function is essential for protecting sensitive information from unauthorized access.
- **Process**: Converts the input message to bytes (if it's a string), encrypts it using the Fernet cipher, and then decodes the encrypted data back into a UTF-8 string for easy handling and storage.

`decrypt_message(self, encrypted_message)`

- **Purpose**: Decrypts an encrypted message back into its original plain text form. This process is the reverse of encryption, allowing secure messages to be read by authorized parties.
- **Process**: Encodes the encrypted message to bytes (if necessary), decrypts it using the Fernet cipher, and decodes the byte string back to a UTF-8 plain text message.

SQLClient.py

The **SQLClient.py** file defines a class **SQLClient** and several functions related to database operations, specifically tailored for managing game data, user authentication, and leaderboards. Here's a breakdown of each function within the **SQLClient** class:

Functions

- **__init__**: Initializes the **SQLClient** instance, potentially setting up database connection parameters or initializing the connection to the database itself.
- **create_connection**: Establishes a connection to the SQL database using predefined connection parameters.
- **close_connection**: Closes the database connection cleanly to ensure data integrity and release resources.
- **create_tables**: A general function that may call other functions to create necessary tables in the database if they do not already exist.
- **create_games_table**: Specifically creates the table for storing game sessions data, including game state, player information, and outcomes.
- **create_users_table**: Creates the table for storing user account information, such as usernames, passwords (hopefully hashed), and other relevant user details.
- **create_leaderboard_table**: Sets up a table for the game's leaderboard, tracking user scores, win/loss records, and possibly rankings.
- **generate_token**: This function likely generates a unique token or identifier, possibly for session management or to link game sessions with user accounts.
- **insert_user**: Adds a new user to the users table, handling user registration processes including storing usernames, hashed passwords, and any other initial user data.
- **authenticate_user**: Verifies a user's login credentials against the stored information in the users table, facilitating user authentication.
- **get_user_id**: Retrieves the unique identifier (ID) for a user based on provided credentials or user information, useful for linking game sessions or leaderboard entries to specific users.

- **update_leaderboard**: Updates the leaderboard table with new game results, adjusting scores and rankings based on the outcome of a game.
- **load_leaderboard_data**: Retrieves leaderboard information, potentially for display to the user, including rankings, scores, and user identifiers.
- **load_user_data**: Fetches data associated with a specific user, such as personal scores, game history, or account settings.
- **load_games_data**: Loads data related to game sessions, including ongoing games, available slots for joining, or historical game outcomes.

Class

- **SQLClient**: This class encapsulates all the database operations necessary for the game's functioning, including connecting to the database, managing user accounts, handling game session data, and updating the leaderboard. The methods defined provide a comprehensive interface for interacting with the game's underlying SQL database, ensuring that data is consistently managed and accessible for game operations.

  Each function within the **SQLClient** class plays a critical role in managing the data layer of the game application, ensuring seamless operation, data integrity, and a good user experience by efficiently handling database interactions

sql_queries.py

The **sql_queries.py** file contains SQL query strings used by the application, likely to create tables, insert records, and perform other database operations. Based on the initial content, here's an explanation of the queries provided:

Queries

- **CREATE_USER_SQL**: This query creates a table named **users** if it does not already exist. The table is structured to hold user account information, including:
  - **username_id**: A unique identifier for each user, set to auto-increment.
  - **username**: The user's chosen username, which must be unique.
  - **password**: The user's password (hopefully stored in a hashed format for security).
  - **token**: A token associated with the user, possibly for session management or authentication purposes.
- **CREATE_LEADERBOARD_TABLE_SQL**: Begins a query to create a **leaderboard** table, structured to track game performance statistics for users, including:
  - **leaderboard_id**: A unique identifier for each entry, set to auto-increment.
  - **username_id**: Links the leaderboard entry to a specific user in the **users** table.
  - **wins**, **losses**, **draws**: Integer columns to track the number of wins, losses, and draws for the user.
  - A **FOREIGN KEY** constraint ensures **username_id** references an existing user in the **users** table.

  The content shown is a snippet, suggesting there are more SQL statements in the file for various database operations related to game management, user authentication, and leaderboard updates. These queries form the backbone of the database interactions within the application, ensuring data is structured and manipulated correctly to support the game's functionality.

sql_constants.py

This file contains a class **SQLConstants** with constants for configuring SQL database connections. These include:

- **HOST**: The database server address.
- **USER**: The username for database access.
- **PASSWORD**: The password for database access.
- **DATABASE**: The name of the database to use, specifically for a TicTacToe game.
  Each file serves a distinct purpose within the application:

- **constants.py** provides basic configuration for networking and encryption.
- **game.py** defines the structure and state management for game sessions.
- **sql_constants.py** configures database connection parameters, separating these specifics from other constants and settings to modularize database interactions.

constants.py

This file defines global constants for the application, likely used across multiple components for consistent configuration. Some of the constants include:

- **HOST** and **PORT**: Network address and port number for the server.
- **FORMAT**: Character encoding format, likely for data transmission.
- **ADDR**: A tuple combining **HOST** and **PORT**.
- **MOVE_TIMEOUT**: Timeout for a player's move, possibly in milliseconds.
- **DATABASE_FILE**: The filename for a database file, suggesting SQLite might be used.
- **key** and **iv**: Variables for encryption, likely used to secure data.
- **SQL_PATH**: Possibly a path to SQL script files or a directory for database schema initialization.

game.py

This file defines a **Game** class, which seems to encapsulate constants related to game state management. The attributes include:

- **PLAYERS_AND_SPECTATORS_CONNECTIONS**: Likely a key for tracking connections to players and spectators.
- **GAME_ID**: A unique identifier for each game session.
- **NUM_PLAYERS**: The number of players in a game.
- **PLAYERS**: Information about the players participating in the game.
- **BOARD**: The game board's current state.
- **CURRENT_PLAYER**: The player whose turn it is.
- **SPECTATORS**: Users watching the game but not actively playing.

5.1 Security

- **Enhancing Application Security**:
  - **Vulnerabilities**: Potential vulnerabilities include SQL injection, cross-site scripting (XSS), and man-in-the-middle (MITM) attacks.
  - **Solutions**: Use prepared statements for database queries, sanitize user inputs, employ HTTPS for encrypted communication, and implement content security policies.
- **Risks of Plain Text Transmission**:
  - **Concerns**: Transmitting messages in plain text exposes data to interception and tampering.
  - **Encryption Implementation**: Use TLS/SSL protocols for encrypted communication channels, ensuring confidentiality and integrity of messages.

5.2 Scalability

- **Handling Concurrent Users**:
  - **Bottlenecks**: Potential bottlenecks include database access, network I/O, and CPU limitations.
  - **Design Considerations**: Implement caching, use a load balancer, optimize database queries, and consider using a microservices architecture to enhance scalability.
- **Load Distribution Strategies**:
  - Employ horizontal scaling by adding more servers, use a load balancer to distribute traffic, and consider using a content delivery network (CDN) for static resources.

5.3 Reliability and Fault Tolerance

- **Ensuring Reliability**:
  - Implement health checks, use redundant systems and failover mechanisms, and regularly backup data to handle server failures gracefully.
- **Message Persistence**:
  - Use a database or a message queue with durability guarantees to ensure messages are stored persistently and can survive server restarts.

5.4 User Authentication

- **Importance of Authentication**:
  - Essential for identifying users, protecting sensitive information, and ensuring that users can only access resources they are authorized to.
  - **Methods**: Use JWT tokens, OAuth, or other secure authentication mechanisms that provide both identification and verification.
- **User Registration and Password Management**:
  - Implement strong password policies, use salted password hashing for storage, and consider multi-factor authentication (MFA) for enhanced security.

5.5 Concurrency and Synchronization

- **Handling Multiple Connections**:
  - Challenges include data consistency, race conditions, and resource contention.
  - **Solutions**: Use locks, semaphores, or other synchronization mechanisms to ensure thread safety and data integrity.
- **Implementing Thread Safety**:

- Use thread-safe collections, atomic operations, and synchronized blocks/methods where necessary to prevent race conditions and ensure safe access to shared resources.

5.6 Protocol Design

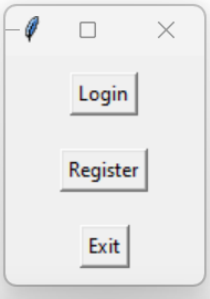
- **Communication Protocol Choice**:
  - TCP is typically chosen for reliability, order, and error-checking, ideal for a game requiring consistent state. UDP might be chosen for faster, but less reliable, message delivery where speed is critical.
  - Factors include reliability, performance, and the specific needs of the application.
- **Socket Handshake and Message Protocol**:
  - Utilize TCP for a reliable connection, ensuring messages are sent and received in order. Design a simple and efficient message format (e.g., JSON) for easy parsing and extendibility.

5.7 Message Ordering and Delivery

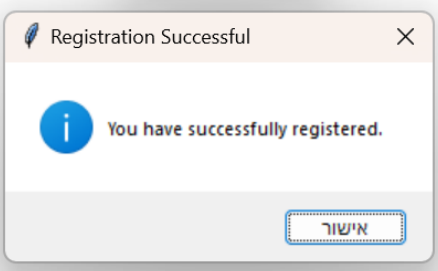
- **Addressing Message Ordering**:
  - Implement sequence numbers or timestamps to ensure messages are processed in the correct order.
- **Ensuring Timely Delivery**:
  - Employ acknowledgments and retries for critical messages, and use TCP's built-in mechanisms for reliable delivery.

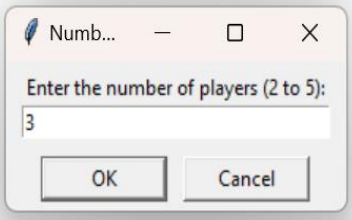
5.8 Persistent Storage

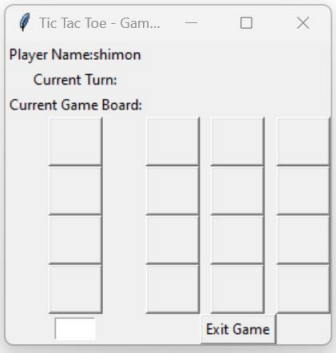
- **Importance and Implementation**:
  - Persistent storage is crucial for saving user data, game states, and ensuring data is not lost. Implemented using databases (SQL or NoSQL) tailored to the application's needs, ensuring scalability and efficiency.
- **Storage Solution Trade-offs**:
  - SQL databases offer structured data and strong ACID properties but might not scale horizontally as well as NoSQL databases, which offer flexibility and scalability but might lack in transactional integrity. These answers offer a starting point for addressing the theoretical aspects of your project. Depending on the specific requirements and constraints of your application, you may need to adapt or expand upon these suggestions.

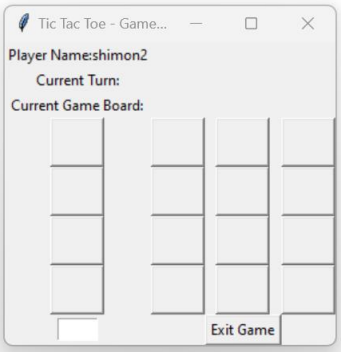

Login
Register
Exit

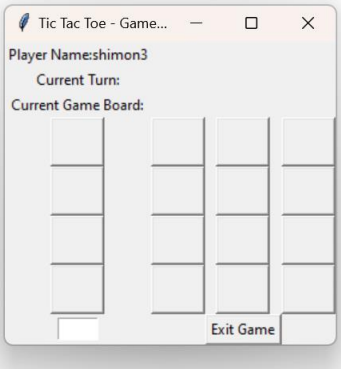

G
Join Available Game
Create New Game
Join Existing Game As Observer
Exit

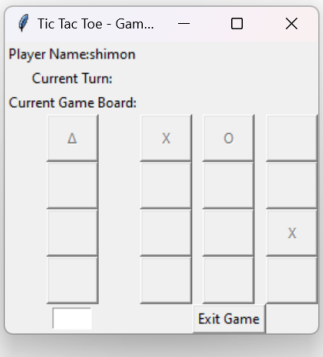

Registration Successful
You have successfully registered.
אישור

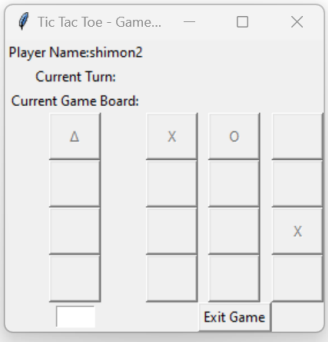

Numb...
Enter the number of players (2 to 5):
3
OK
Cancel

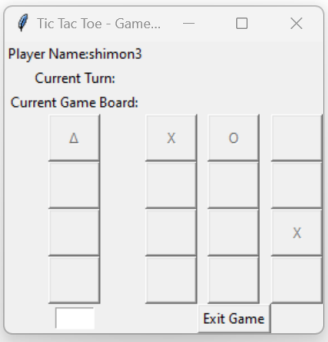

Tic Tac Toe - Gam...
Player Name:shimon
Current Turn:
Current Game Board:
Exit Game

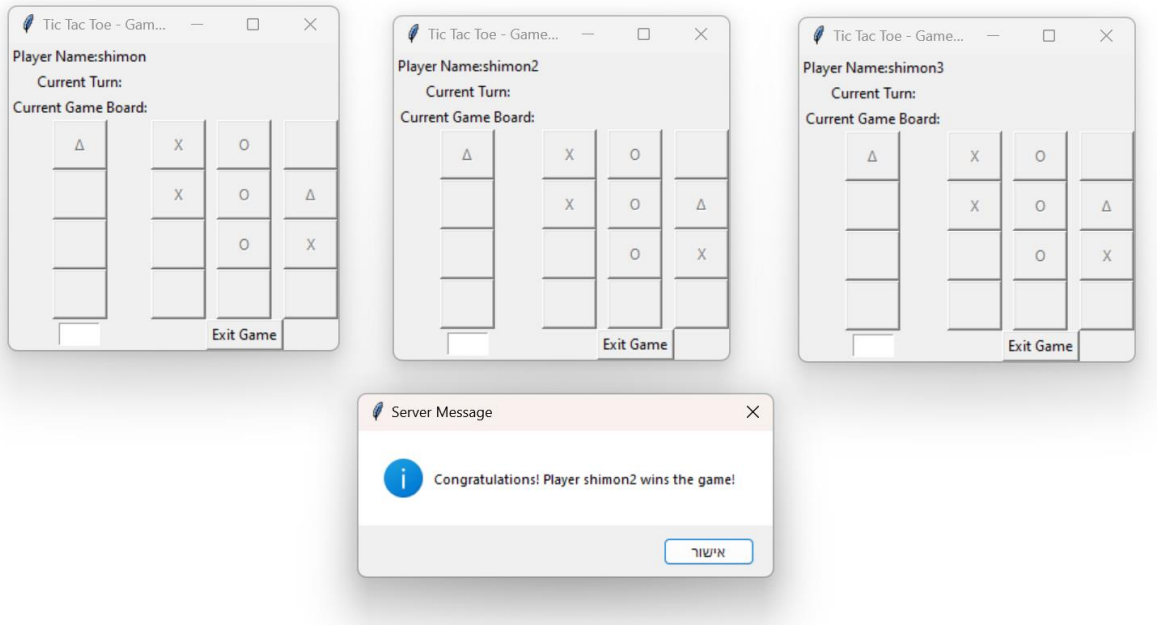

Tic Tac Toe - Game...
Player Name:shimon2
Current Turn:
Current Game Board:
Exit Game

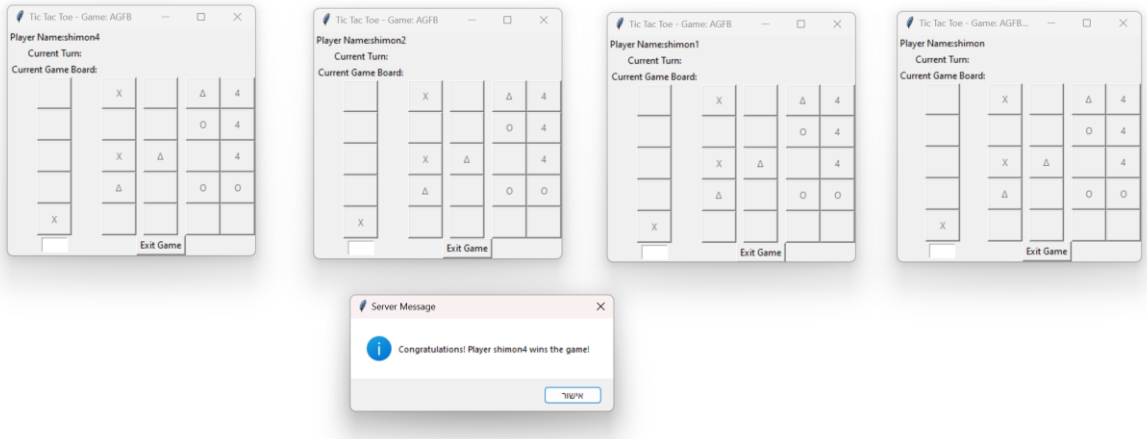

Tic Tac Toe - Game...
Player Name:shimon3
Current Turn:
Current Game Board:
Exit Game


Tic Tac Toe - Gam...
Player Name:shimon
Current Turn:
Current Game Board:
Δ
X
O
X
Exit Game

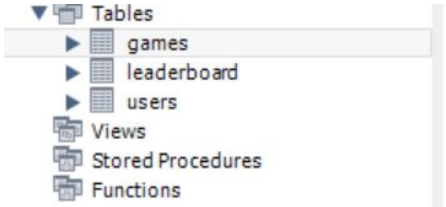

Tic Tac Toe - Game...
Player Name:shimon2
Current Turn:
Current Game Board:
Δ
X
O
X
Exit Game


Tic Tac Toe - Game...
Player Name:shimon3
Current Turn:
Current Game Board:
Δ
X
O
X
Exit Game


Tic Tac Toe - Gam...
Player Name:shimon
Current Turn:
Current Game Board:
Exit Game
Tic Tac Toe - Game...
Player Name:shimon2
Current Turn:
Current Game Board:
Exit Game
Tic Tac Toe - Game...
Player Name:shimon3
Current Turn:
Current Game Board:
Exit Game
Server Message
Congratulations! Player shimon2 wins the game!
אישור


Tic Tac Toe - Game: AGFB
Player Name:shimon4
Current Turn:
Current Game Board:
Exit Game
Tic Tac Toe - Game: AGFB
Player Name:shimon2
Current Turn:
Current Game Board:
Exit Game
Tic Tac Toe - Game: AGFB
Player Name:shimon1
Current Turn:
Current Game Board:
Exit Game
Tic Tac Toe - Game: AGFB...
Player Name:shimon
Current Turn:
Current Game Board:
Exit Game
Server Message
Congratulations! Player shimon4 wins the game!
אישור

SQL Tables:



Result Grid | Filter Rows: | Edit:

leaderboard_id	username_id	wins	losses	draws
1	1	4	0	0
2	3	3	0	0
3	6	1	0	0
5	8	1	0	0
NULL	NULL	NULL	NULL	NULL

username_id	username	password	token
4	shimon	d74ff0ee8da3b9806b18c877dbf29bbde50b5bd...	48cafb20c7a54497440a5a2ca290f401
5	shimon1	d74ff0ee8da3b9806b18c877dbf29bbde50b5bd...	6877ca2985148da7feda0f25e26f3c01
6	shimon2	d74ff0ee8da3b9806b18c877dbf29bbde50b5bd...	768ba5ff20df31ae4bcbf07aa22c38d4
7	shimon3	d74ff0ee8da3b9806b18c877dbf29bbde50b5bd...	c7b9ecd9421da8f1c0c32d01444e0b28
8	shimon4	d74ff0ee8da3b9806b18c877dbf29bbde50b5bd...	cc351f939ba37f11ff727dccb493c6c3
NULL	NULL	NULL	NULL