Chatting with Web Sockets
Task: Write a Web Socket Server for Direct Messages (DMs)

In a package named server, write a class named DMServer that:

• When created, sets up a web socket server listening for connections on localhost:8080

• Listens for messages of type "register" containing a username as a String (Use data structures to remember which socket belongs to which username)

• Listens for messages of type "direct_message" containing a JSON string in the format {"to": "username", "message": "text"}. When such a message is received:
  • Send a message of type "dm" to the "to" username containing a JSON string in the format {"from": "username", "message": "text"}

• Example: If 2 different users connect to the server and send:
  • emit("register", "Aesop") and emit("register", "Rob")
  • User "Aesop" sends emit("direct_message", {'to': "Rob", "message": "Happy to be on the food chain at all"})
  • User "Rob" will receive a message from the server of type "dm" containing the string {'from": "Aesop", "message": "Happy to be on the food chain at all"}
Chat Demo

- Let's build a chat app!
- Code is in the repo
- Users can connect to the chat server
- Use a web or desktop front end
- Server doesn't care what type of app a client is using
- All connected users can communicate through text messages
Chat Architecture

- Web Client
- Desktop Client
- Web Sockets
- Chat Server
- SQL Statements
- MySQL Database
Chat App

• Chat server starts up
• Listens for WebSocket connections on port 8080
• Initialize data structures that will store reference to each WebSocket
Chat App

- Server connects to a MySQL database to store the chat history
- Communicates via SQL statements
  - MySQL reacts to the event of receiving a statement
- More details on MySQL in a later lecture
Chat App

- Clients connect to the server using WebSockets
- Client could be web or desktop
- After the connection is established:
  - Client sends a message of type register containing their username
Chat App

• The server receives the register message and reacts to this event

• Adds the new user to the data structures

• Data structure remember the username associated with this socket

• Retrieve the chat history from the database and send it to the client

```sql
SELECT *
```
Chat App

- Client reacts to the chat_history message
- Renders all the content and displays it to the user
Chat App

- Multiple clients can be connected simultaneously
- Each client sends their username in a register message
- Chat servers maps usernames to sockets for all connections
Chat App

• All users can send messages of type chat_message to the server
Chat App

- All users can send messages of type `chat_message` to the server.
- Message is sent when a user sends a message using the GUI.
- This message only contains the message (No username).

```
  | Client --> chat_message --> Chat Server
  |          |                | MySQL Database
  | Client   |                |
  | Client   |
```
Chat App

• When the server receives a chat_message:
  • Lookup the username for the sending socket
  • Store username/message in the database
  • Send username/message to all connected sockets in a message of type new_message
Chat App

- Clients receives the new_message
- Add it to the GUI for the user to read
Chat App

• When a client disconnects the server reacts to the disconnect event

• Remove the user from data structures
To the Code
Clicker Architecture

Client

Chat Server

Game Actor

Actor System
Clicker App

- When the app starts
  - An actor system is created
  - A ClickerServer actor is added to the system
  - UpdateGames message is sent to the server at regular intervals
Clicker App

• When a client connects and chooses a username

• This username is sent to the server in a WebSocket message of type startGame
Clicker App

- In response to receiving the gameStart message, the server:
  - Sends the client the game configuration in a message of type initialize
  - Creates a GameActor in the actor system
  - Updates data structure to remember that this game actor is associated with this web socket
Clicker App

- To create a new Actor
- Use the context variable of any actor
- Use this context the same as the actor system
- Ex. clickerServer.context.actorOf...
Clicker App

- An new game actor is created for each connected client
- Important to update all data structures to associate clients with their actors
Clicker App

- When the server receives click and buy message from a web socket
- Forward the action as an actor message to the appropriate actor
- Game actor will update its state according to the configuration of the game
Clicker App - Update

- Each time the clicker server receives the UpdateGames actor message
- Send an Update message to each game actor
Clicker App - Update

- Each game actor responds with the GameState message (to the sender())
- GameState contains all information of the game in a JSON string
Clicker App - Update

• The clicker server forwards each game state to the appropriate client in a gameState message

• Each client parses the JSON string and updates the GUI for the user to see
Clicker App - Update

- This update process occurs at regular intervals
  - 10 times/second in the handout code
- Notice that all the game logic occurs on the server
- Client only sends user inputs and renders the game state

![Diagram](Image)
Clicker App - Expansion

- Expansion objective - AutoSave
- Send messages to save all games at regular intervals
- Store all game states in a way that will persist
- If a user sends the startGame message with a username that has a saved game, load their game

![Clicker App - Expansion Diagram](image-url)
Task: Write a Web Socket Server for Direct Messages (DMs)

In a package named server, write a class named DMServer that:

• When created, sets up a web socket server listening for connections on localhost:8080

• Listens for messages of type "register" containing a username as a String (Use data structures to remember which socket belongs to which username)

• Listens for messages of type "direct_message" containing a JSON string in the format {"to": "username", "message": "text"}. When such a message is received:
  • Send a message of type "dm" to the "to" username containing a JSON string in the format {"from": "username", "message": "text"}

• Example: If 2 different users connect to the server and send:
  • emit("register", "Aesop") and emit("register", "Rob")
  • User "Aesop" sends emit("direct_message", {"to": "Rob", "message": "Happy to be on the food chain at all"})
  • User "Rob" will receive a message from the server of type "dm" containing the string {"from": "Aesop", "message": "Happy to be on the food chain at all"}