Team Number:
15

Team Members:
Barrett Brown, Jonathan Gott, Alex Phibbs, Adam Berry, Minh Vu

Project Name:
Web-based Tic-Tac-Toe

Project Synopsis:
A real-time multiplayer tic-tac-toe game hosted through the browser with support for multiple rooms allowing users to play against one another.

Architecture Description:
Our application is a web-based full-stack multiplayer Tic-Tac-Toe game designed to allow people to easily connect, join or create rooms. The project consists of a .Net C# backend using websocket connections to enable real-time gameplay between users on the front end. The frontend was built using HTML/CSS/JS to allow for a lighter-weight delivery of our files/scripts. At the architectural level the project follows a basic client-server model. The frontend will handle user-interactions, board rendering and allow for users to create/join rooms. They will then talk directly to the websocket gateway, at this level they will receive the information from the frontend and pass that event to the “Room Handler” object. This object's responsibility is mainly finding the room that the user is connected to and passing the state to its corresponding game handler. Going into the game handler that will hold the current state of the game but also allow for any state updates to be done on its board. Once those updates are done it will pass the new updated state back to the room handler. The room handler upon receiving this passes it back the state to the websocket layer which will then pass that back to the frontend side. The frontend side upon receiving this state update will update its internal 2d-array for its board and then re-render the board. This allows for the consistent point of source to be maintained on the servers side since if any user clicks instead of optimistically pre-empting the update on their side and having to undo it in the case that it wasn’t their turn or whatever else they don’t result in a collision. When zooming out you can see that the project is very event-driven with the player clicking on the cell making an event sending it back with the backend validating it applying the move and then sending it back to the clients.

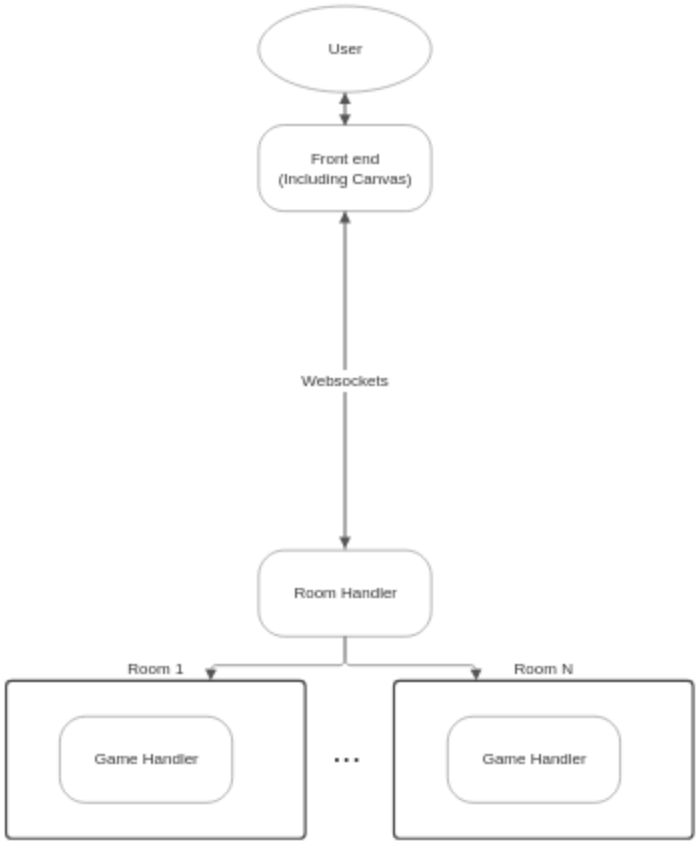
Description of classes/main functions:
Websocket Gateway:
This will be the main point of communication between the frontend and the backend. Any message sent from their side will be sent directly to here. This will check to see if they want to join a room/leave a room, if they did a click/chat option and then also handle the sending back of state updates from the game handler passed to the room handler

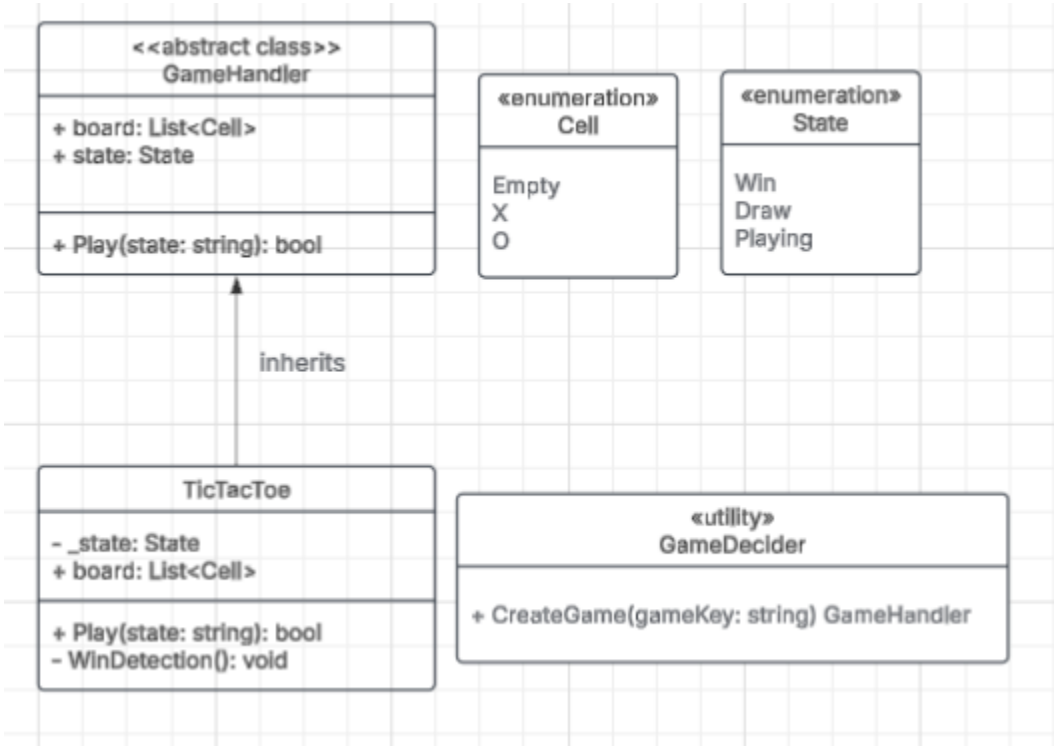
Room Handler:
The room handler will be the funnel that allows a player to be matched to their room. To do this we have implemented the singleton design pattern to ensure that only one global room registry exists at a time. This allows us to make sure that the room object itself wont be duplicated but we also made sure that the room dictionary of {roomid: clients} was thread-safe otherwise we could result in a situation where it would crash/be inaccurate. As the room handler itself doesn't need to be a big class we just needed a couple of main operations. For this we only needed something to create a room, find a room/send room update, and delete a room if the clients = 0. Additionally we also have a function in here for our quick play feature by looping over the rooms that have a single player in them to allow for quick joining. Finally we have to ensure that the joining of rooms supports >2 players by putting others into a "spectator" box.

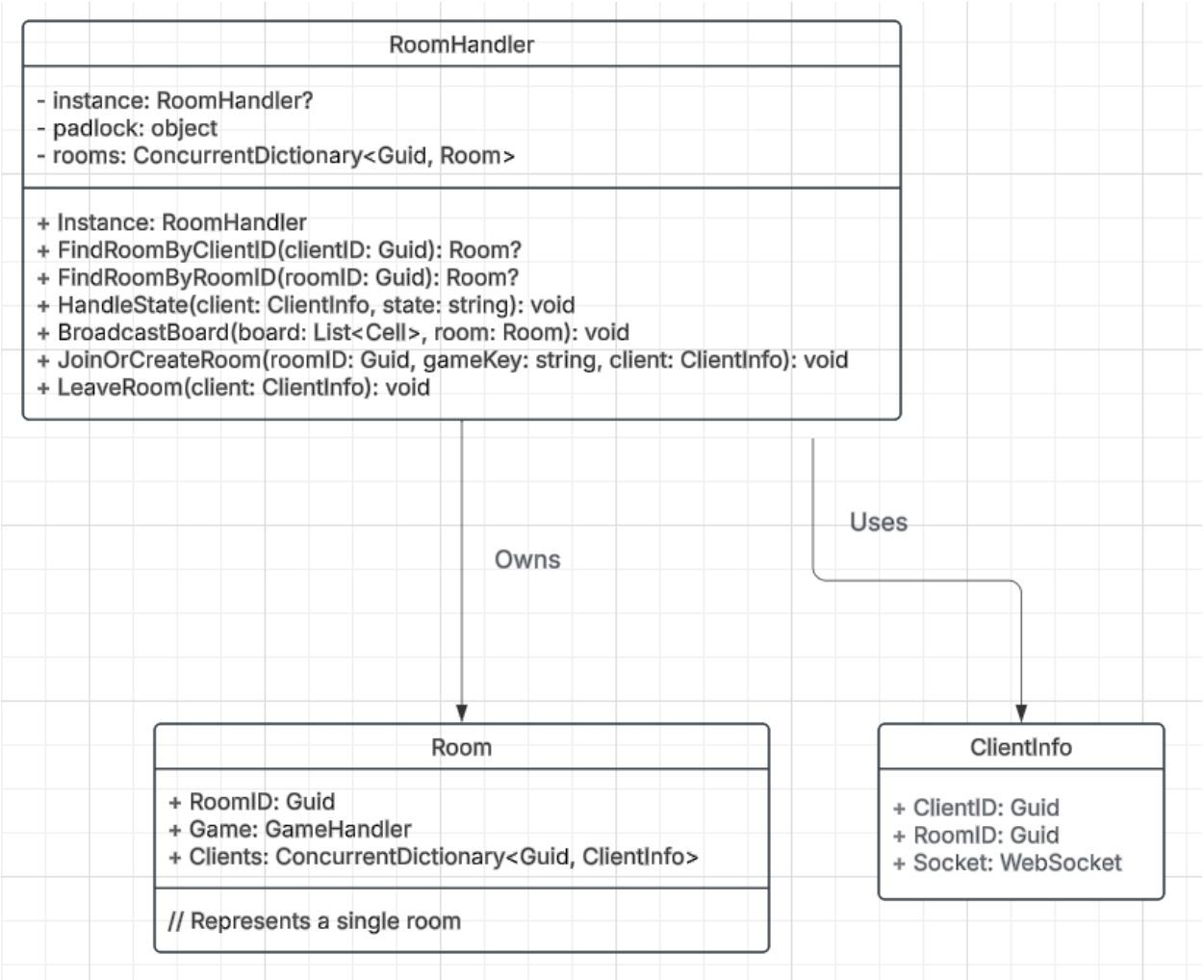
Game Handler:
Each game handler initiates its own game of tic-tac-toe stored in a 2d-array. The functions supported by this would be to "apply a move", detect a win/draw and finally to restart a game. Upon receiving a move from the room handler it will apply this to it's 2d array and then send back that new state to the room handler to allow for broadcasting to the clients. This class needs to handle a lot of base game logic like validating moves, ensuring event types match expected and concurrency between this and other game handler objects.

Frontend:
This will be the main point the users will interact with. This will have a couple of main functions. The first is it will handle the opening of a websocket connection, afterwards it will handle rendering the UI + re-rending upon a state update. After this it will also handle turning the users clicks into "click/play" events and passing that back to the backend. This allows for the frontend to not have a lot of actions bloating their side while allowing for easy adoption of accessibility and extensibility.

UML Diagrams:


User
Front end
(Including Canvas)
Websockets
Room Handler
Room 1
Room N
Game Handler
...
Game Handler


<<abstract class>>
GameHandler
+ board: List<Cell>
+ state: State
+ Play(state: string): bool
«enumeration»
Cell
Empty
X
O
«enumeration»
State
Win
Draw
Playing
inherits
TicTacToe
- _state: State
+ board: List<Cell>
+ Play(state: string): bool
- WinDetection(): void
«utility»
GameDecider
+ CreateGame(gameKey: string) GameHandler


RoomHandler
- instance: RoomHandler?
- padlock: object
- rooms: ConcurrentDictionary<Guid, Room>
+ Instance: RoomHandler
+ FindRoomByClientID(clientID: Guid): Room?
+ FindRoomByRoomID(roomID: Guid): Room?
+ HandleState(client: ClientInfo, state: string): void
+ BroadcastBoard(board: List<Cell>, room: Room): void
+ JoinOrCreateRoom(roomID: Guid, gameKey: string, client: ClientInfo): void
+ LeaveRoom(client: ClientInfo): void
Owns
Uses
Room
+ RoomID: Guid
+ Game: GameHandler
+ Clients: ConcurrentDictionary<Guid, ClientInfo>
// Represents a single room
ClientInfo
+ ClientID: Guid
+ RoomID: Guid
+ Socket: WebSocket