When a user who has signed up joins the lobby, a websocket connection is established to the “lobby socket” endpoint. When that happens, their user data is sent to the server so that it updates the lobby browser accordingly and renders a button with their name on it that they won’t see but other players will. Once another player joins, the same process happens, and then either player can click on each other.

```
function Lobby() {
    const [ items, setItems ] = useState( initialState: []);
    const { user } = useUser();
    const navigate = useNavigate();
    const location = useLocation();

    const { sendMessage } = useWebSocket( url: "ws://localhost:8080/lobby_socket/",  options: {
```

```
useEffect( effect: () => {
    //console.log(user);
    sendMessage(JSON.stringify(user));
}, deps: []);
```

Now, these lobby buttons all have click events binded to them, so when clicked, a method that sends a POST request to the “connection” endpoint on the server with the id of both the user who clicked on the button and the id of the user that button represents are sent as query parameters of the POST.

```
{
    items.filter(x => x.id != user.id).map(({id, name}) => {
        return <button key={id} onClick={handleJoinGame(id)} className="button-enabled">{name}</button>
    })
}
```

```
const handleJoinGame = id => async () => {
    const response = await fetch( input: "http://localhost:8080/connection?" + new URLSearchParams({
        thisId: user.id,
        otherId: id
    }),  init: {
        method: "POST"
    });
```

On POST to that endpoint, a method called startGame() is invoked on the server that queries the users from the database, establishes a new GameConnection for both players and sets the data of the GameConnections correctly, then inserts those 2

records into the database. Lastly, the GameConnection data for both players is then sent over to the “lobby socket” endpoint through a method called sendGameStart(), which then sends that data, stringified, to each corresponding player.

```
Niklas Sheth
@Path("/connection/")
public class GameConnectionResource {

    2 usages
    @Inject
    LobbyResource lobbyResource;

    Niklas Sheth
    @POST
    @Transactional
    public Void startGame(@QueryParam("thisId") long thisId, @QueryParam("otherId") long otherId) {
        Player thisPlayer = Player.findById(thisId);
        Player otherPlayer = Player.findById(otherId);
        Game game = new Game();
        game.persist();
        GameConnection a = new GameConnection();
        GameConnection b = new GameConnection();
        a.game = game;
        a.player = thisPlayer;
        a.moveOrder = 0;
        b.game = game;
        b.player = otherPlayer;
        b.moveOrder = 1;
        a.persist();
        b.persist();
        lobbyResource.sendGameStart(a);
        lobbyResource.sendGameStart(b);
        return null;
    }
}
```

```
2 usages  Niklas Sheth
public void sendGameStart(GameConnection connection) {
    JsonMapper mapper = new JsonMapper();
    try {
        sessions.inverse().get(connection.player).getAsyncRemote().sendText(mapper.writeValueAsString(connection));
    } catch (JsonProcessingException e) {
        throw new RuntimeException(e);
    }
}
```

Finally, once they receive that data, both clients are redirected over to the “multiplayer” route, which just re renders the page to the Connect 4 multiplayer board with all the

data from the server transferred over to that component through the {state: data} argument to the navigate() method. They end up in the same game because the GameConnections “game” were set to the same Game.

```
const { sendMessage } = useWebSocket( url: "ws://localhost:8080/lobby_socket/", options: {

    onMessage: (event : MessageEvent<any> ) => {
        //console.log(event.data);
        const data = JSON.parse(event.data);
        if (data instanceof Array) {
            //lobby data
            setItems(data);
        }
        else {
            navigate( to: "/multiplayer", options: {state: data})
        }
    }
```

```
// this component is mainly for setting up all of our routes
// All of these routes have access to the user state because they are children of the UserProvider context
const App = () => (
    <UserProvider>
        <Routes>
            <Route index element={<Landing />} />
            <Route path="local" element={<Connect4Local />} />
            <Route path="signup" element={<Signup />} />
            <Route path="lobby" element={<RequireUser><Lobby /></RequireUser>} />
            <Route path="multiplayer" element={<Connect4Multiplayer />}/>
        </Routes>
    </UserProvider>
);
```

Removing the buttons from the lobby also occurred throughout this entire process.

We also used a library called Jackson for generating all the JSON messages as strings to the clients and parsing the stringified JSON messages sent from the clients to the server.

```
    JsonMapper mapper = new JsonMapper();
    Game game = sessions.get(session).game;
    try {
        Move move = mapper.readValue(message, Move.class);
```