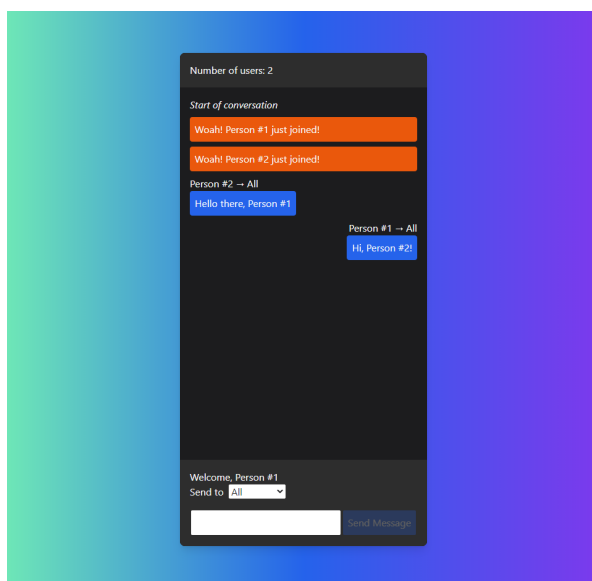
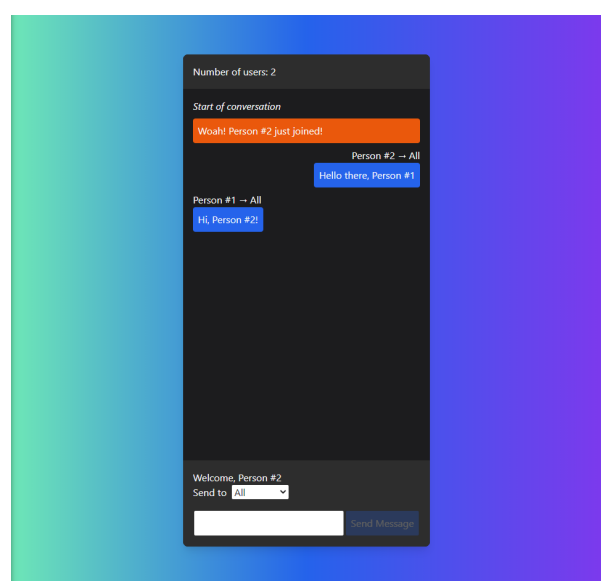


Abstract

The following report outlines the design and development of a chat application which allows multiple clients to share messages. This project aimed to implement a single server - multiple client model for such a purpose. JavaScript (or more specifically, [TypeScript](#)) was used for to create both the client-side and server-side applications. On the client-side, [SvelteKit](#) was used to create the GUI for selecting a username, sending messages, and viewing the current members of the chat. On the server-side, [Deno](#) was used to create the `FileServer` and `WebSocket` handler. The server runs on port `8080` , serving the static client-side files created by Svelte using a simple `FileServer` . Any connections to `https://THE-URL/ws` are directed to the another implementation using the [Deno WebSocket](#) library. The result is a chat app with a modern UI and simple interface for chatting with multiple people, or sending a direct message to a particular user. The usage of Typescript ensured type-safety when sending objects between the client and the server (as these were simply JSON strings representing a variety of JavaScript objects). The following report includes screenshots of the interface with various use cases, along with test cases verifying the correct operation of the program.



A simple back and forth {Person 1}



{Person 2}

Detailed Design

The main project is separated into three components: the `Client`, the `Server`, and shared `Utilities`. The following section will discuss these components in detail.

Utilities

This folder simply holds the TypeScript interfaces that are shared between the client and server. For example:

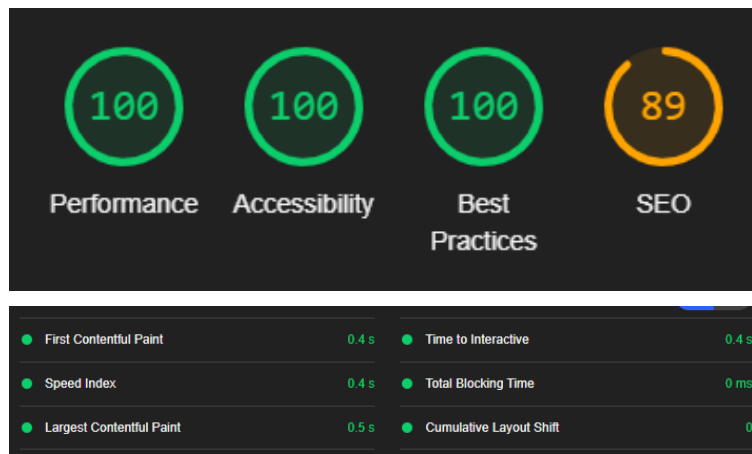
```
export interface Message {
  from: string;
  text: string;
  to: string;
}

export interface Client {
  name: string;
}
```

Client

The client, as mentioned previously, is built with [SvelteKit](#). As opposed to alternatives such as Vue and React, Svelte is a *pre-compiled* library that does **not** use a Virtual-DOM. The benefit is a much smaller package size sent to the client, a pleasant development experience, and blazing-fast apps.

For styling, [WindiCSS](#) is used to merge the benefits of [TailwindCSS](#) and tree-shaking, resulting in an optimized `*.css` file. The entire application, when shipped to the browser, is only `55.1 kB`. The final result? Well, the [Lighthouse](#) score certainly speaks for itself. *Note that the poor Search Engine Optimization score is due to the lack of a `<meta>` tag:*



Server

The server is powered by [Deno](#), a “simple, modern and secure runtime for JavaScript and TypeScript”. The server is really two routes running simultaneously:

1. A `FileServer` to send the static files generated by Svelte
2. A `WebSocket` server to handle users and their messages

The `FileServer` is straightforward, using [Oak](#) to serve the static files (CSS, JS, etc.) and direct users to the `index.html` page.

The `WebSocket` server is a little more complicated, using the [Deno WebSocket](#) library for tracking clients, broadcasting messages, and sending direct messages to particular clients.

```
// Broadcast function
function broadcast(data: string) {
  if (!data) return;

  clients.forEach(function each(client) {
    if (!client.websocket?.isClosed) {
      client.websocket?.send(data);
    }
  });
}

// Direct message (simplified)
function sendPost(post: Post) {
  const clientTo = clients.find((_cl) => _cl.name === message.to);
  if (!clientTo?.websocket?.isClosed) clientTo?.websocket?.send(JSON.str
```

```

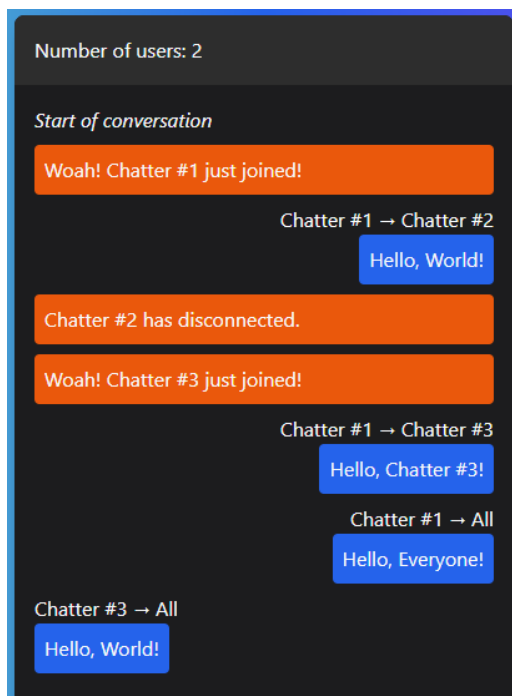
const clientFrom = clients.find((_cl) => _cl.name === message.from);
if (!clientFrom?.websocket?.isClosed) clientFrom?.websocket?.send(JSON
}

```

Testing and Evaluation

Project Specifications

1. Multiple clients



2. No GUI is needed for the server

3. A simple GUI can be implemented for the server

Well, there isn't one.

```

[INFO] Server listening on http://localhost:8080
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #9030 just joined!
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #2143 just joined!
[WARN] Guest #2143 has disconnected.
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #7592 just joined!

```

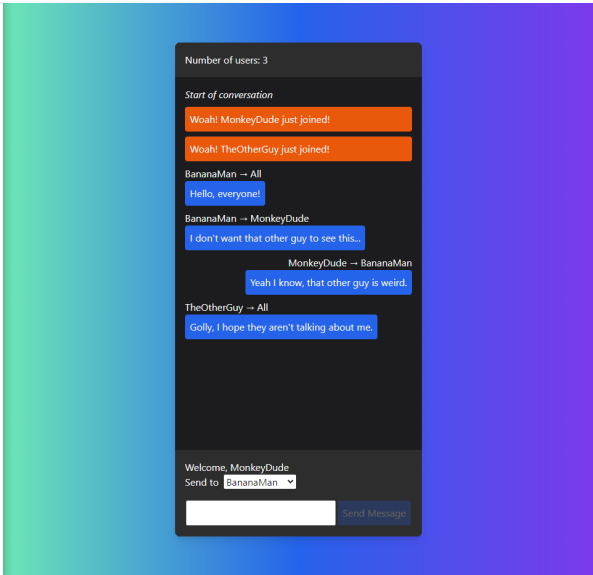
4. Clients must be able to choose a nickname

Username

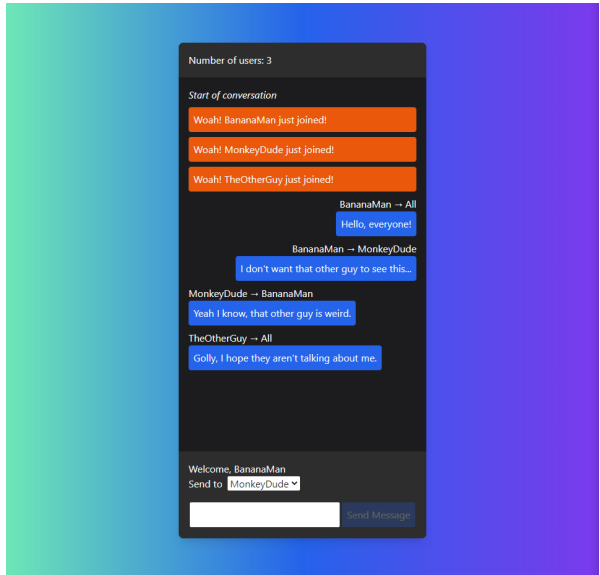
CoolKid123

Request Username

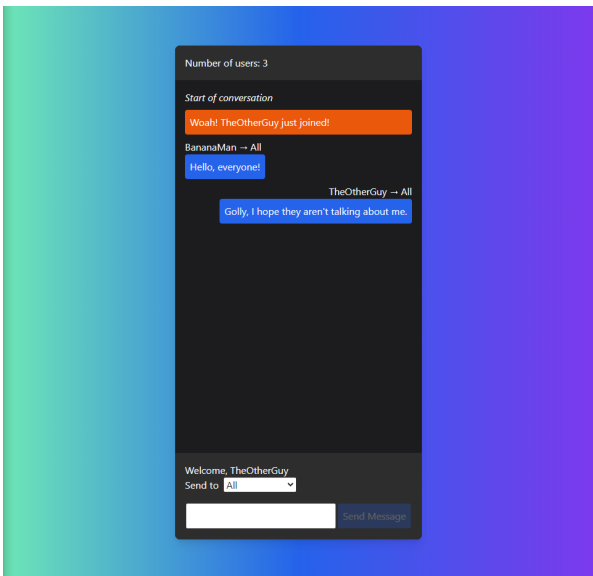
4. Clients must be able to “whisper” to each other without having messages displayed to other users



{MonkeyDude}



{BananaMan}



{TheOtherGuy}

The images above show two people holding a conversation, while *TheOtherGuy* is unable to see any of their messages.

The Server

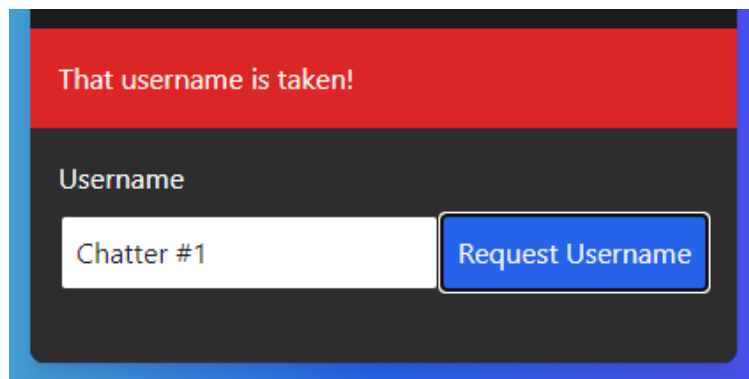
1. Server operations (such as connect requests and disconnect requests) should be printed out by the server.
2. The server must handle connections / disconnections without disruption of other services.

As show below, the server logs connections and disconnections. Clients are disconnected when they close their browser window, and the server is not disrupted by clients joining and leaving. Note that a client *joins* when they first open the app, but only receive messages once they have claimed a username (shown by the User connected, awaiting username message).

```
[INFO] Server listening on http://localhost:8080
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #9030 just joined!
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #2143 just joined!
[WARN] Guest #2143 has disconnected.
[WAIT] User connected, awaiting username.
[JOIN] Woah! Guest #7592 just joined!
```

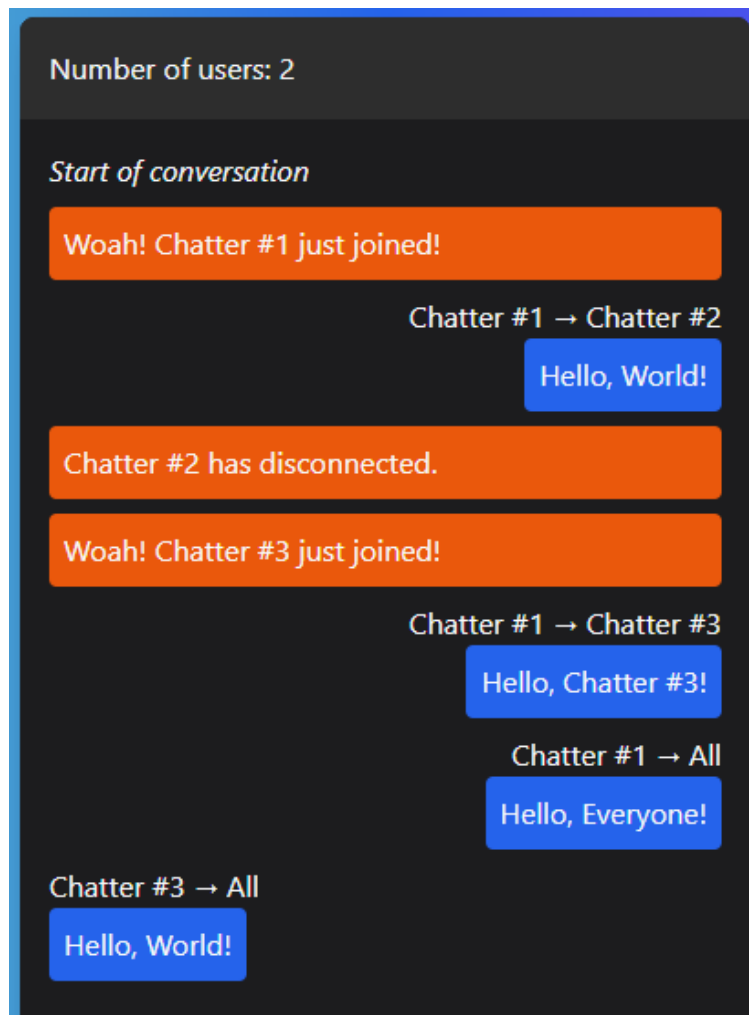
3. Clients must have unique nicknames, duplicates must be resolved before allowing a client to be connected.

If a client attempts to claim a username that is already taken, they are not allowed to enter the chat and are prompted with the error below. When the requested username leaves, a new client can claim that username. Note that the client is *suggested* a name such as “Guest #1241” when they join, but this can be changed by simply deleting the text and entering a custom username.



4. All clients must be informed of changes in the list of connected users.

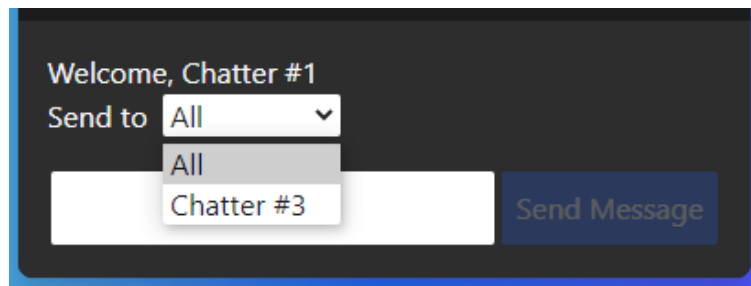
When a client claims a username and joins the chat, the server broadcasts this to all clients in the chat. When a client disconnects, this is also broadcasted to all clients. On the client side, these messages are highlighted in orange:



The Client(s)

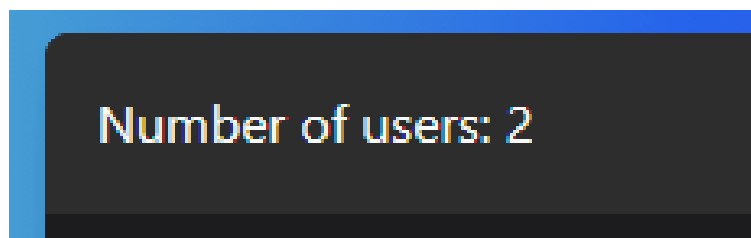
1. A list of online users must be displayed (via GUI or command) to all users.

A list of connected users is displayed in the “Send To” dropdown. This dropdown is also used to target specific users for a direct message. These direct messages aren’t just hidden from other clients, the server really only sends the message to that *one* particular client.



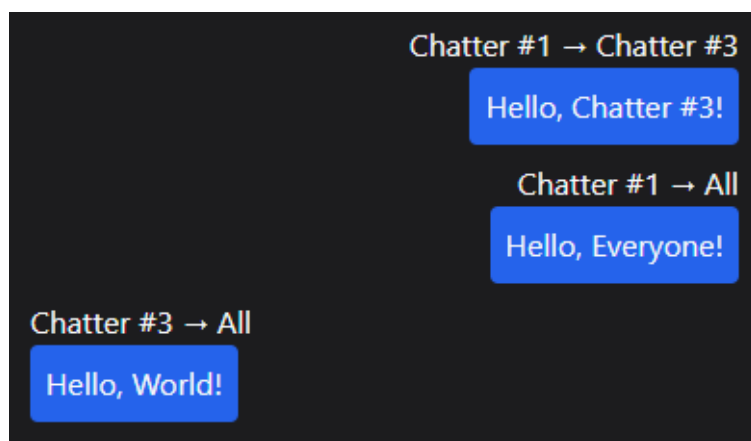
2. Connection / disconnection actions of users must be displayed to all users.

This was already shown above (with the orange boxes). Note that there is also a “Number of users” counter to show how many active users there are in the chat. When users leave and join, this number updates simultaneously.



3. Messages from the originating user and other users must be displayed (in other words the messages you send must also be displayed).

Messages that are sent from the user are right-side oriented, while all other messages are left-side oriented (like most chat apps). Above each message is further information about who sent the message (on the left side of the arrow) and who the message was sent to (on the right side of the arrow). “All” represents everyone in the chat.



4. Must still be able to receive messages / actions while typing a message.

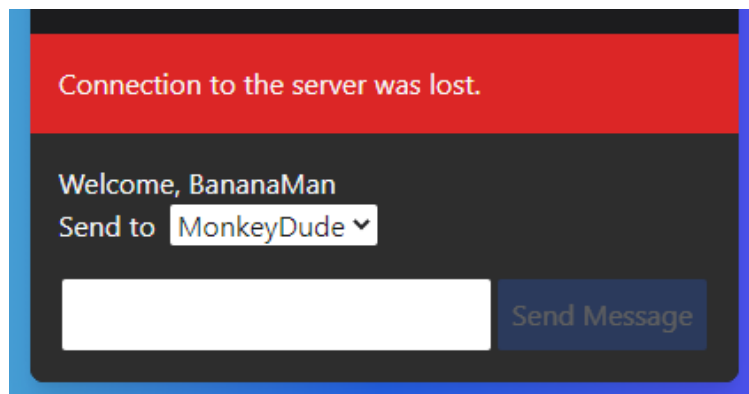
Unsure how to demonstrate this, but rest assured that the main thread is never blocked to prevent the user from typing. This is mainly a JavaScript thing, and would be pretty hard to do incorrectly anyways.

5.\ Clients must be able to disconnect without disrupting the server.

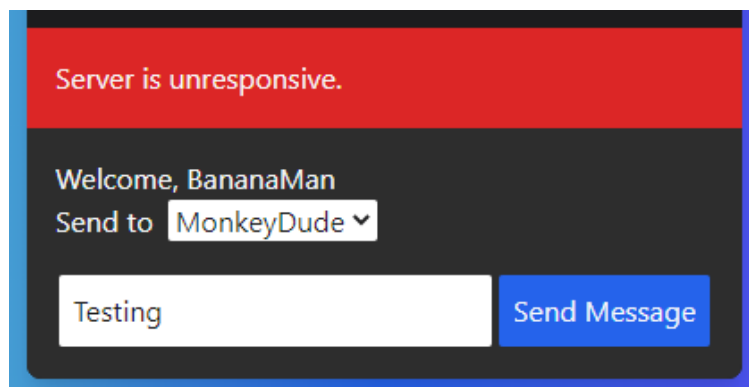
This was also shown earlier in the server log, but clients can connect and disconnect to their hearts' content without disrupting other clients.

Stability and Error Reporting

The client-side program will report errors to the user about the status of the server. For example, if the server was online but went offline:



If the WebSocket becomes closed (for any reason), the user is prompted with the following when attempting to send a message or request their username:



Due to the reactive nature of Svelte, it is impossible to whisper a non-existing client. If the user has a client selected that leaves, it will auto-magically select "All" for them (and the client is removed from the list of clients). That happens with this line:

```
message.to = clients.findIndex((_cl) => _cl.name === message.to) !== -1 ?  
message.to : 'All';
```

I'm not quite sure what "Client Stable after Server Termination" means, but I will say that the client will need to reload the page if the server terminates and then comes back online.

For "All Opened Sockets and Streams Closed on Client after Termination", the WebSocket implementation in JavaScript seems to handle this automatically. When the client closes their browser window, it emits one last "close" event to the server and then closes the connection.

Conclusions

This project presents a modern chat application built with web technologies, capable of running on almost any device with an internet connection. Users can message to everyone, or send private messages directly to another user. These private messages aren't simply broadcasted and then hidden client-side, they really are only sent to the target user.

I was going to try and host the program, but they asked for my credit card so I got scared. I usually use NodeJS for this kind of thing, but wanted to try the bleeding edge (which would be almost everything I used... Svelte, Deno, WindiCSS...)

Appendices

The code is available here: <https://codeload.github.com/rmanky/chat-app/zip/refs/heads/master>