Chatting with each other

In this section we will

- Send messages to everyone on the chat server
- View messages in a chat window

Creating a chat input component

1. Navigate to ***src/components*** and create a ***chat-input*** component. We will use this later to send messages to the server.
   - Using what you have learned about your debug component, scaffold an empty ***chat-input*** component with ***component.ts, html, scss, vue, and index.ts*** files.
   - Import your ***chat-input*** component into your ***chat*** component and render your ***chat-input*** component in your ***chat*** component's html file. ***Don't forget to register your chat-input component in your chat component.***
   - Bind the ***chat*** component's ***chatServer*** to your ***chat-input*** component in the same way you did the debug component.
   - In your ***chat-input*** component, create an ***input*** field of type ***text*** and a clickable ***button***.
   - Bind your ***chat-input's input field*** to a property on your ***chat-input*** component.
   - Bind the click of your ***chat-input's button*** to a method on your ***chat-input*** component.
   - Code your ***button click*** so that when it is clicked, your input field's bound property is reset to an ***empty string***.

   We have scaffolded a new component, complete with some functionality. With this in place, we can now type a message, and clear that message once our button is clicked. All that's left is to send your message to the server.

   Does this new chat-input component seem familiar to you? It's incredibly similar to the login component, isn't it? There is also an input field and a clickable button there too.

   When creating components it's best practice to abstract away from your use case and instead create an encapsulated, reusable component where possible. If we had a single text-input component we could have used that for both login and chat!

   If you feel up to the challenge, try to refactor your existing code to use a single, reusable component. You could create a component that binds a text variable and function callback for the button click, for example.

Creating your own command

1. Inside ***chat.component.ts***
   - Look at how we are using the auth command to send a message to the server.
2. Inside ***chat-server/outgoing***
   - Create a new file named ***message.ts***

- Using ***AuthCommand*** as a template, create a new class that implements the ***ICommand interface*** called ***MessageCommand***
- This ***MessageCommand*** will need to have the following ***public*** properties
    1. ***'room'*** of type ***string***
    2. ***'message'*** of type ***string***
    3. ***'command'*** value equal to ***'message'***
- Export ***MessageCommand*** from its file

3. Inside ***components/chat-input***
    - Import your newly created ***MessageCommand***
    - On a button click, create a new ***MessageCommand***
        1. The room is ***'general'***
        2. The message is the contents of your input field.
    - ***send*** the server your ***MessageCommand***
    - View your application in the browser. You should now be able to log in, type a message, send that message, and view your message in your debug component.

You can see how easily we can create new commands to send to our server. By keeping the message payloads as simple JSON, we can communicate commands such as Auth and Message, and receive responses from the server, such as Auth, Message, and Memberlist.

By default, every person that connects to the server is subscribed to the room 'general'.

At the server end, we are handling messages through the web socket the same way we are in the chat client. By creating an event stream, we can subscribe to specific messages, or create unique event streams per message. This way we can decouple our application logic. We'll now look at how we can create a new event stream for viewing messages others have sent the server.

You will have noticed by now every command you send to the server is echo'd back to your client with a corresponding success message. As we develop our client further, try to think about how you could handle error messages from the server when commands fail. For example, an auth command without a valid username will fail. A message without a valid room or message will fail too. These include a helpful description why.

When sending a message to the server, the message you receive in response is the source of truth. Never trust the client! You'll notice even though you did not append your username, or a timestamp, to your message command that they appear on the response. This helps ensure any clients that consume the chat server can trust the origin, and ordering of messages. Let's display some friendly formatted messages.

Viewing messages in a chat window

4. Navigate to ***src/components*** and create a new empty component named ***chat-window***
5. Import this component into your ***chat*** component and render it to the screen
    - Remember to bind your ***chatServer*** to from your ***chat*** component to your ***chat-window*** component
6. In your ***chat-window*** component, create an array named ***messageList*** of type ***string*** and initialise it to an empty array. This will contain all the friendly messages received from the server.
7. In your ***chat-window.html***, create a ***div*** with a ***ul*** element inside it.

8. Inside your ***ul*** element, use ***list rendering*** (this can be found on your Vue cheat sheet) to iterate over your ***messageList*** array. This will display friendly messages on screen.
9. Inside ***chat-window.scss***, style your ***div*** container to have a ***width of 50%*** and ***height of 45vh*** and set ***overflow-y*** to ***scroll*** and set ***text-align*** to ***left***
10. Sanity check your work by initialising your ***messageList*** array with some simple strings, and make sure they output to the screen as you would expect.

Filtering responses from the chat server

This bit is really cool. I wanted us all to be able to develop this ourselves, but the logic is quite complicated. Especially if you aren't used to Vue, TypeScript, and RxJS. The code has been provided for you, but I urge you all to research what we are doing in this file.

To understand what is happening, search for RxJs pipe, filter, and map. Hopefully as a group many of us have reached this part of the code. If there is time, we can look at this together as it is easier to explain in words than it is in text.

What we are going to do is create a new Observable stream. We will use this to create an event stream of only one particular chat command.

11. Navigate to ***src/chat-server/chat-server.ts***
    - Create a public property called ***message$***. Its type is ***Observable<Message>.*** We will route every message command from the server through this Observable.
    - Inside the ChatServer's constructor initialise your property using the following code:
        i. this.message$ = Message.toStream(this.connection.stream$);
12. Navigate to ***src/chat-server/incoming/message.ts***
    - Explore this file. This contains a lot of complex information
    - Using ***rxjs operators*** for every message the chat server sends your client, this class will filter those messages to those it is interested in. Namely, ***message*** commands. It will take only these commands, and using a new, separate Observable, push message commands onto their own stream.
    - We now have a way to access only the message commands sent by the server. Subscribing to ***message$*** inside your chat-window component, you will not receive events for auth or memberlist commands.
13. Navigate to ***src/components/chat-window/chat-window.component.ts***
    - Update your code to subscribe to the newly available ***message$*** stream on your ChatServer.
    - In your subscription handler your message parameter can now be strongly typed to Message. You will need to import this from ***src/chat-server-incoming/message***
    - In your subscription handler, using the available fields, concatenate a string together composed of the time, username, and message. Push this string onto your ***messageList*** array.
14. View your application in the browser. You should now be able to send and receive formatted messages. Yay!

15. You may notice when you receive lots of messages, they are hidden, and you need to scroll down to access it. Wouldn't it be nice if it did that automatically?
    - In your chat-window component's subscription handler add the following code at the end:

```
var elem = this.$el;

Vue.nextTick().then(function() {
  elem.scrollTop = elem.clientHeight;
});
```

Consider that Vue renders on every frame in your browser, or a 'tick'. What we are saying here is "Please update my messageList array with this latest value. On the next tick, after you have processed and rendered the dom, run the following code to ensure the container that my messages are in is scrolled to the bottom."

Without waiting for the next tick, when your messages exceeded the length of your container, the bottom most recent message would be the only one not visible on screen.