Connecting to the server

In this section we will

- Run our app
- Connect to the chat server
- Send the chat server an 'auth' command
- View chat server responses on screen

Getting started with your application

1. From the command line navigate to the root chat-client directory. Inside this folder should be a ***package.***json file. Execute ***'npm install'*** followed by '***npm run serve'*** in this directory.

    - This will compile your application for local development.
    - Open the localhost link in your browser to see your application.

    Any code changes you make until you terminate this command will cause a hot reload. This means when you save changes to files, your application will re-compile and you can see those changes immediately in your browser!

    Ctrl+C will terminate this command if you need it.

Viewing your application

2. In your browser you will see the login screen, enter a username and hit 'Go!'

    When navigating between the login screen and the chat screen Vue is rendering different html. Even though the URL changes, we aren't navigating away and rendering a new page. In the background, Vue is destroying and created new pages on the same screen. This is a Single Page Application and made possible by the Vue router.

Connecting to the server

3. Navigate to ***src/views/chat/chat.component.ts***
    - You'll see there are already some imported modules, and an empty class.
    - In your class create a private method called ***mounted***.
    - Inside this method, create an instance of ***WebSocketServerConnection*** using the chat server URI as a ***string parameter***
    - Create an instance of ***Server*** using the instance of ***WebSocketServerConnection*** you just created.
    - Save your changes and view your chat screen in your browser.
    - Check your console for any errors. If there aren't any, we're good to go!

    mounted is a 'lifecycle hook'. It's one of many events Vue will fire when creating a component. You can view these on your Vue cheat sheet. You can hook into different lifecycle events to ensure code is executed at the right time. We prefer to use the correct lifecycle event rather than using constructors.

4. Navigate to ***src/chat-server/chat-server.ts***
   - You'll see an empty stubbed out class. We are going to extend the ***Server*** class so we can do something!
   - Create a public property called ***responses$***. Its type is ***Subject<unknown>***
   - In your constructor, initialise it to a ***new Subject()***
   - Create a private method called ***broadcastMessage***. It will take a single parameter called ***message***. Its type is ***unknown***. The method returns ***void***
   - Inside ***broadcastMessage*** call the ***next*** method on your ***responses$*** property, passing in your ***message***
   - In your ***constructor*** add the following line
     - ***this.connection.stream$.subscribe(this.broadcastMessage.bind(this));***
   - If you'd like to understand '***this binding***' grab and ask me. Because I am not typing out here!

   responses$ is a rxjs Subject. You can go online and read more about this. Simply, it is an event stream. You push messages into it, and subscribers at the other end are alerted when messages come through. It may seem a bit silly right now to forward on all messages from the WebSocketServerConnection through the Server, and we will see in future how we can handle this better. It is standard in JavaScript to denote streams with a $ at the end. Much in the same way it is standard to denote jQuery objects with a $ at the front!

5. Navigate back to ***src/views/chat/chat/chat.component.ts***
   - Replace your instance of ***Server*** with your instance of ***ChatServer.*** Make sure you store your ***ChatServer*** in a ***variable inside mounted*** named ***chatServer***.
   - Create a private class property called ***allResponses*** of type ***Array<string>.***
   - Initialise your ***allResponses*** property inside ***mounted*** by setting it to an empty array

   You'll see TypeScript complain that allResponses has not definitely been initialised. This is because we're not using constructors, and we haven't initialised allResponses on the same line it was declared. There is a stylistic choice to make here.

   1. Initialise allResponses on the same line it is declared, and remove the initialisation from mounted

   2. We know we have initialised it inside of mounted, therefore we can add '!' to the end of allResponses so it becomes ***allResponses!*** *This tells TypeScript to trust us, and we are explicitly saying "Don't worry, I know this is definitely initialised correctly"*

   In Vue if we use method number 2, we will get an exception because Vue will try to access allResponses to render HTML before it is initialised in your component's mounted function. There are happy paths around this, but for now. Go with option 1.

6. Navigate back to ***chat.component.ts***
   - Create a ***private method*** called ***handle***. It has a single parameter ***message*** of type ***unknown***
   - In here, call ***JSON.Stringify***, passing in your message, and store the result in a variable.

- Push your variable into your ***allResponses*** array
- Add the following code to your mounted function

```
chatServer.responses$.subscribe(this.handle);

chatServer.connect().then(x => {

  chatServer.send(new AuthCommand("yourusername"));

});
```

7. Inside ***chat.html***
    - Below the ***p tag*** add the following

```
<textarea v-model="allResponses"></textarea>
```

- In your web browser, when you load the chat page you should now see messages from the chat server inside your ***textarea*** element.

Congratulations! You've connected to the server. We have subscribed to our previously created Subject to receive any message from the ChatServer, and we are now pumping that out to the screen. Using promises, we are connecting to the server, and once we successfully connect, are immediately sending an auth command. The server does not respond to anything until an auth command has been issued. As such, you can see some response messages on your screen.

Creating a debug component

8. Create a new ***debug*** folder inside ***src/components***
9. Inside the folder create the following
    - Create ***debug.component.ts***
    - Create ***debug.html***
    - Create ***debug.scss***
    - Create ***debug.vue***
    - Create ***index.ts***

    Index.ts acts as a barrel export file. A barrel file is a way to rollup exports from several modules into a single convenient module. The barrel itself is a module file that re-exports selected exports of other modules. In our Vue application, without an index.ts file our imports in other files would need the fully qualified path of our .vue files. When looking at import paths Vue will use by default an index.ts file if it exists.

10. Inside ***debug.vue***
    - Copy the contents from your ***chat.vue*** component. Update your references to your ***debug*** files.
11. Inside ***index.ts***
    - Copy the contents from your ***chat's index.ts***. Update your references to your ***debug.vue*** file
12. Inside ***debug.html***
    - Copy your ***<textarea>*** element from your chat component into here. Place it inside a ***div*** container.

   - Style your ***div*** container to have a class named ***margin-all***

13. Inside ***debug.scss***
    - Style ***divs*** to have a ***width*** of 100% and a ***height*** of 15vh
    - Style ***textareas*** to have a ***width*** of 65% and a ***height*** of 100%
    - You can style the raw HTML elements here. No need to worry about creating classes.

    Usually when writing CSS if you style the raw element, it will apply to every element in the DOM. In Vue, we can choose to have our component's CSS only apply for that component. If you look inside debug.vue you will see when we are importing our CSS we have also declared 'scoped'. Scoped means the CSS will apply only to our component.

14. Inside ***debug.component.ts***
    - Using what you have already seen, scaffold an empty class called ***Debug***.
    - Make sure it ***extends Vue***
    - Create a ***public property*** called ***chatServer*** of type ***ChatServer***. We will be using dependency injection to inject our server from the chat component into our new debug component. Add an '!' to the end of chatServer to tell TypeScript we are definitely going to initialise it. ***chatServer!***
    - Add a ***@Prop()*** decorator to your ***chatServer***
       i. You will need to: ***import { Prop } from "vue-property-decorator";***
       ii. ***Prop() public chatServer: ChatServer***
    - Move all logic relating to ***allResponses*** from your chat component into your new debug component. This includes the subscription, which you can now use your ***chatServer*** property for. **Do not include connecting to the server and sending an Auth command.**

Using your new component

15. ***Inside chat.component.ts***
    - Above your class you can see a ***@Component*** decorator. Inside it specifies the name of your chat component. In Vue you need to declare the child components you wish to use.
    - Import your new ***debug*** component into your file
    - Add a '***components'*** key inside the ***decorator***.
    - Set the value of '***components'*** equal to ***{ debug: DebugComponent }***
    - Delete all code from inside your chat component that has now been moved to your new debug component.
    - Store a reference to your ***chatServer*** as a ***private property*** on your Chat class. Leaves the initialisation inside ***mounted*** and ensure you are using ***'!'*** to create ***chatServer!***
16. Inside chat.html
    - Change your ***<textarea>*** element to now be a ***<debug>*** element
    - Remove the v-model to ***allResponses*** that no longer exists.
    - Bind the chat component's ***chatServer*** to the debug component's ***chatServer.***
17. Make sure all your files are saved and up to date. View your application in the browser. Check if everything works.

Lifecycle hooks revisited

Checking your browsers console, you will notice that things aren't running smoothly and that there are a bunch of errors. Don't worry, that's expected. What's happening?

The Chat component is created and mounted. As a part of this, its html is also mounted, which includes all of its child components. After which, we are then initialising our chatServer property and trying to connect to the chat server!

Simply, before out chatServer property has been initialised, it has become bound to our child debug component. That component is then trying to execute methods as a part of its mounted lifecycle hook. All the code should work, it's just happening in the wrong order!

18. Using your Vue cheat sheet, change the chat component's lifecycle hook from mounted to something else. The best hook is the one furthest along in the lifecycle that still works.

Congratulations. You've finished part one!