Web Sockets

Overview

In this lab, you'll implement a Chat Room web application by using HTML5 Web Sockets. The application will incorporate server code and client code.

When the client sends a message to the server, the server will receive the message and broadcast it back to all clients (you can have multiple browser windows, representing different clients). Each client will display a transcript of all the messages broadcast from the server, from every user.

We'll provide all the server code, and you'll implement the client code ☺.

Roadmap

There are 3 exercises in this lab, of which the last is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Familiarise with the project
2. Implementing the client
3. Additional suggestions (if time permits)

Exercise 1: Familiarise with the project

The project contains two servers:

- A socket server, located in `\start\socketserver`
- A web server that serves the socket client code, located in `\start\socketclient`

The project itself is a node-project. In the package.json file you can see what the dependencies are and how to run the servers.

Install

To install the dependencies, open the command line in the folder where package.json is located. Then type:

```
npm install
```

This will create a node_modules folder with all dependencies.

Run

To run both servers, open the command line in the folder where package.json is located. Then type:

```
npm start
```

This will start both servers in parallel and will open de browser. If you want to open multiple chat windows, just open more browser tabs with the same URL.

Stop running

Execute ctrl+c in the command line twice or do ctrl+c once and answer the question with Y(es).

Exercise 2: Implementing the client

You're now ready to implement the client. This will be a Web page that contains a mixture of HTML and JavaScript, to establish a Web Sockets connection with the server and to have an on-going Web Sockets conversation with the server.

Follow these steps to get started:

- Locate `index.html` in the `\start\socketclient` folder.
- Take a look at the code in `index.html`. At the bottom of the file, note the following HTML:
    - There's a simple `<form>` that contains a text box. This is where the user will be able to enter text, ready to send to the server (the form will automatically submit the text when the user presses ENTER).
    - There's also a `<div>` named `messages`. This is where messages will be displayed, when they are broadcast from the server.

  Now skim to the top of the file and note the following points:
    - We've included the jQuery script files, to simplify client-side processing.
    - The `init()` JavaScript function contains start-up code. The function initially prompts the user to enter his or her name, and displays the name on the web page. The function then contains a series of TODO comments, asking you to implement the client-side logic for a Web Sockets conversation with the server. You'll implement all these steps in this exercise...

Add code to the `init()` function, to implement a Web Sockets conversation. Each of the following steps corresponds to a TODO comment in the `init()` function:

a) Set the `uri` variable to hold the URL for Web Sockets communication. This will be a URL such as `ws://localhost:8888/?username=andy`, but rather than hard-coding it like this, build it up bit-by-bit as follows:

   - The URL definitely starts with `ws://`, because this is how you indicate you want to establish a Web Sockets conversation with the server.
   - The host name might not be `localhost` – it could be a real domain name such as `www.acme.com`! How do you know what to use? The simple answer is, the host name of the Web Sockets server is the same as the host name for the current Web page (they come from the same server). You can get the host name of the current web page as follows:

     ```
     window.location.hostname
     ```

   - You can hard-code the next bit of the URL:

     ```
     ':8888/?username='
     ```

     The `?username=` part sets up an HTTP parameter for the request.
   - To finish off the URL, simply append the `username` variable.

b) Create a new `WebSocket` object, passing `uri` as a parameter. Assign the object to the `websocket` variable. This will set up the Web Sockets connection.

c) Handle the "open" event on the `websocket` object, which signifies a successful connection with the server. In the handler function, do the following:

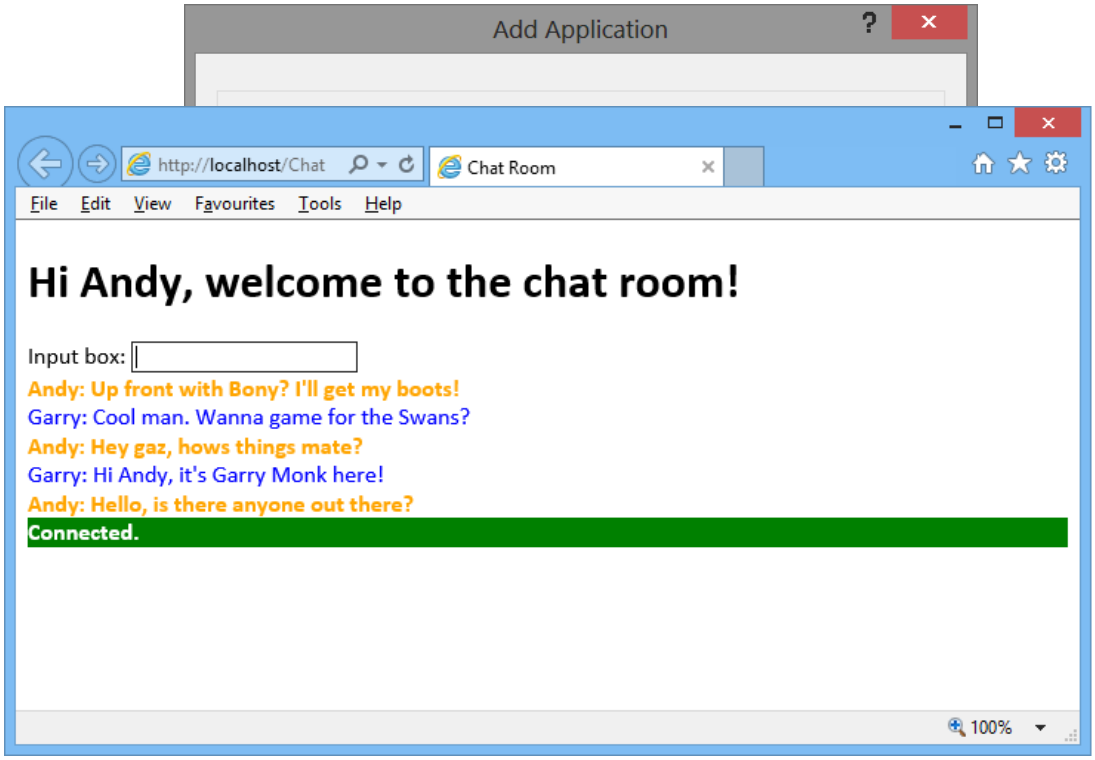
   - Display a message on the web page, to indicate that we've successfully established a Web Sockets connection with the server.
   - Set up a handler the "submit" event on the form. Get the text from the `inputbox` text box, and send it to the server via `websocket.send()`.

d) Handle the "error" event on the `websocket` object, which signifies a problem occurred somewhere. Display a simple error message.

e) Handle the "message" event on the `websocket` object, which signifies we've received data from the server. Display the message data in the `messages` element (hint: the event parameter has a `data` property, which contains the data sent from the server).

Exercise 3 (If Time Permits): Additional Suggestions

- Define some styles to display messages in different colours (e.g. orange if it's your



  message being echoed back to you, or blue if it's someone else). Also define some styles to make the "connected" or "error" messages more prominent. For example:

- Introduce multithreading at the client, do that you can have a separate stream on conversations with particular people.
- Send more "interesting" data between the client and server. Hint: use JSON.