LAB: USING WEBSOCKETS

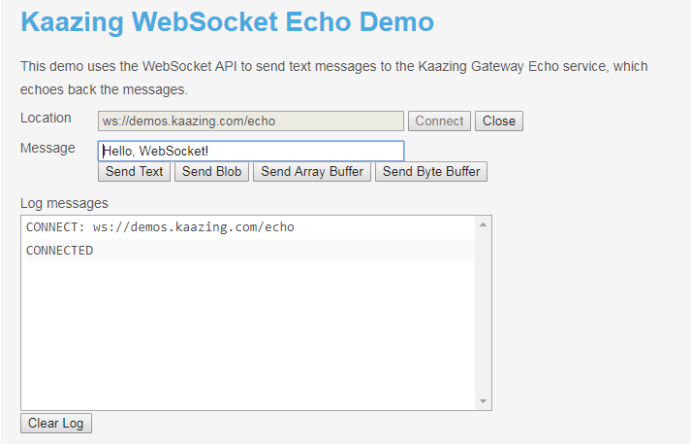
In this lab you will create client-side code to implement simple communication with WebSocket servers. You will send messages to a WebSocket echo server, and you will observe the difference between polling and WebSocket implementations of server-push.

Task 1. Using an echo server to test communication

The Kaazing Websocket Echo Demo (**http://demos.kaazing.com/echo/index.html**) is useful for testing WebSocket communication. The webpage at the above URL provides a web interface that allows your browser to connect to a publicly reachable WebSocket server that simply echoes any message it receives back to the sender. It is useful for testing whether your browser supports WebSockets.

1. Open the above URL in Chrome. Open the Chrome Developer tools and select the Network tab. In the Network tab, select WS from the available network traffic types.

2. Click Connect in the web page. You should see a log message showing that your browser has connected to the WebSocket server. Note that the URL of the server is shown in the Location box.



3. You should see a single request in the Network tab. Click on this and choose to inspect Headers. Answer the following questions:

   - **What is the URL that the request was sent to?**
   - **What is the URL scheme (e.g. http://)?**
   - **What is the value of the Connection request header?**
   - **What header in both request and response has a value that indicates websocket is being used?**

4. Click the Send Text button (you can change the value in the Message box before doing so if you want, it doesn't really matter what the actual message is). Switch to the Frames tab instead of Headers. You should see two messages containing

the text you sent, one outgoing, sent by your browser, one incoming, returned by the Echo server (as that is all the server does with a message).

- **How do you know which messages are incoming and which are outgoing?**

5. Click the other Send… buttons. These send binary data. You should be able to see the frames in the Frames tab, but you won't see the content of them.

- **You may also see some other incoming and outgoing text frames that don't match up to any message you sent – what do you think the purpose of these is?**

Task 2. Comparing polling and WebSocket

The Echo server is useful for testing, but the real-time web can do a lot more than just echo a message. To do something more interesting you'll need to implement a server. You can run the server on your local machine. For this you will use **node.js**. You will be given the server code – since node.js is just server-side JavaScript you should be able to understand what it does fairly easily, but you will focus on creating client-side code.

Open a command window and type:

node –v

This checks that node.js is installed on your computer. If not, download it from the following URL and install it:

https://nodejs.org/en/download/

Once node is installed you will need to add WebSocket support. In your command window enter

npm install websocket

You will also need a local web server on your machine, e.g XAMPP or Chrome Web Server (the lab PCs have XAMPP installed on the host machine).

Polling

1. Download the file *http.js* from GCULearn. This is a **node.js** file that creates an HTTP server that listens on port 3000 and responds to requests by returning a singe numeric value. The value changes every so often, so repeated requests may receive the same or different values depending on the timing. This value could represent some real-world information that changes frequently, for example a company's stock price.

2. Create a web page called *http_test.html*. This page should:

- Contain a DIV to display the numeric value, with a suitable label
- Run a script that uses *setInterval* function to poll the server every 5 seconds – at the end of each interval an XHR should sent and the value returned is appended to the DIV.

3. Deploy this page on your local web server.

4. Run your node server by opening a command window in the folder where http.js is located and entering:

   node http.js

   You should see a message in the command window every time the value on the server changes.

5. Access your web page on your local web server. Watch what happens. You can also observe the network traffic in the Chrome developer tools.

   - **Do you ever see the same value repeated?**
   - **Does the polling interval match the interval that the server uses to change its value?**

6. Try adjusting the polling interval so that you never see the same value returned more than once.

   - **Do you ever have to wait for a significant time after the value changes before you see it in the web page? You should see the value change on the node server by observing messages in the command window where it was run from.**

WebSockets

1. Download the file *ws.js* from GCULearn. This is a **node.js** file that creates a WebSocket server that listens on port 1337 and once a connection has been made sends messages to the client containing a singe numeric value. The value changes every so often and every time it does a message is sent. Note that this is a very simple implementation that only works sensibly when a single client is connected.

2. Create a web page called *ws_test.html*. This page should:
   - Contain a DIV to display a numeric value, with a suitable label
   - Run a script that connects to the WebSocket server and then responds to messages by appending the value received to the DIV. It never actually needs to send a message to the server, as the communication in this example is driven by the server.

   Note that you can use the WebSockets chat example to help with creating the above files. You can download the code for that example from the URL

mentioned in the lecture, or from GCU Learn (chat-server.js, chat.html, frontend.js)

3. Deploy this page on your local web server.

4. Run your node server by opening a command window in the folder where http.js is located and entering:

   node ws.js

   You should see a message in the command window every time the value on the server changes.

5. Access your web page on your local web server. Watch what happens. You can also observe the network traffic in the Chrome developer tools.

   - **Do you ever see the same value repeated?**
   - **Does the change in the web page content match the interval that the server uses to change its value?**
   - **Which solution (polling or WebSocket) will give the best user experience and minimise network traffic?**
   - **Are there any disadvantages to that solution?**

Task 3 (extra). Using Socket.io

Usually when creating a node-based WebSocket application you would use frameworks such as **Express** and **Socket.io**. These will allow you to create a web server and WebSocket server within the same node app, and make the design of more complex applications much more straightforward than with "raw" WebSocket code.

There is a nice demo/tutorial of a Socket.io chat application at:

https://socket.io/get-started/chat/

Follow though this tutorial and try to get the example working on your computer. You can compare this application with the WebSocket chat application in the lecture