WebSocket

Sistemas Distribuídos 2018/19

1 Introduction

Given that HTTP requests are always initiated by clients, how can a Web server update the content shown in a Web browser?

One of the first approaches was to use *auto-refreshing* iframes on the webpage (using the `meta-refresh` header). This would update part of the webpage to have new content. This is an example of polling that wasted CPU on both the Client and the Server.

More recent techniques explored *long-polling requests*, in which the server doesn't output the response immediately, but keeps the connection open until something of interest is to be sent to the client. Comet is a well known example of these techniques, considered workarounds to push content to browsers.

Proposed for standardization in 2011, *WebSocket* is an API that solves this problem by allowing notifications to be *pushed* to the browser. WebSockets are a full duplex channel with a JavaScript API that can be initiated from any webpage. WebSockets are designed to be used by browsers and webservers, but can be used by any client/server. The aim is to expose an API similar to TCP Sockets.

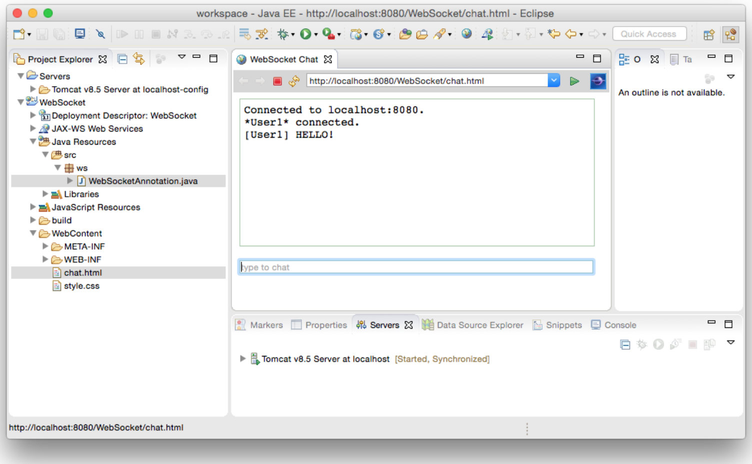
2 WebSocket Handshake

A WebSocket connection is initiated by the browser, like any HTTP request, with headers preceding the content. Notice, in the code below, the `Connection: Upgrade` and `Upgrade: websocket` headers that specify that the browser requests to upgrade this connection from HTTP to WebSocket. If the Server supports it, it will reply with the same headers and a two-way communication channel will be created. After that, Client and Server will no longer communicate via HTTP, as the result is a WebSocket connection.

```
GET /mychat HTTP/1.1
Host: server.mychat.org
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://mychat.org
```

3 Chat using WebSockets

In this demonstration example you will implement a chat application using WebSockets. You are supplied with an incomplete example, in which each client writes something to the server and receives a response. You shall complete the code in order for all clients to receive all messages (one chat window for everyone to talk). Load the project's files into your IDE, as shown in the figure below:



```
Connected to localhost:8080.
*User1* connected.
[User1] HELLO!
```

Tomcat v8.5 Server at localhost [Started, Synchronized]

3.1 HTML

The user interface of the WebSocket Chat is minimal, and consists fundamentally of the two lines below. The first line contains an inner div with the chat history, and the second line has an input box to write new text. All the behavior is specified in JavaScript.

```
<div id="container"><div id="history"></div></div>
<p><input type="text" placeholder="type to chat" id="chat"></p>
```

3.2 JavaScript code

The JavaScript code, which must be supported in order to use WebSockets, sets up a connection and a few event handlers (key presses and received messages).

```
window.onload = function() { // URI = ws://10.16.0.165:8080/WebSocket/ws
    connect('ws://' + window.location.host + '/WebSocket/ws');
    document.getElementById("chat").focus();
}
```

In this snippet, we create a custom handler for the `onload` event of the webpage. Once the page loads, the `connect()` function will be called, and the chat input field will be focused (similar to being selected or clicked by the user).

Note: The connection is established to the host that served the web page (`window.location.host`) and the project **must** run within `.../WebSocket` so make sure you configure your web project correctly (e.g., name the project "WebSocket").

```
function connect(host) { // connect to the host websocket
    if ('WebSocket' in window)
        websocket = new WebSocket(host);
    else if ('MozWebSocket' in window)
        websocket = new MozWebSocket(host);
    else {
        writeToHistory('Get a real browser which supports WebSocket.');
        return;
    }

    websocket.onopen    = onOpen; // set the event listeners below
    websocket.onclose   = onClose;
    websocket.onmessage = onMessage;
    websocket.onerror   = onError;
}
```

The connect function receives the URI (Uniform Resource Identifier) of the WebSocket server endpoint. Notice that the IP and port are being dynamically populated from the `window.location.host` variable. The used protocol is `ws://` instead of the traditional `http://` because of the different nature of the connection.

That URI will be used to create the `WebSocket` object (or `MozWebSocket` in the case of Firefox Browser). This object creates and represents the websocket connection to the server endpoint. Afterwards, the handlers for the four main events are bound: `onOpen`, `onClose`, `onError` and `onMessage`. While the first three are self-explanatory, the `onMessage` event is raised when the browser receives a message sent by the WebSocket server – the server may generate such messages *at any time*.

```
function onOpen(event) {
    writeToHistory('Connected to ' + window.location.host + '.');
    document.getElementById('chat').onkeydown = function(key) {
        if (key.keyCode == 13)
            doSend(); // call doSend() on enter key
    };
}
```

When opening the WebSocket connection, a message will be printed in the `history` HTML div. Furthermore, the `onKeyDown` event of the input field will be hooked to a function that will check to see if was the enter/return key (code 13). If it was, it will call the `doSend()` function to send it to the server.

```
function onMessage(message) { // print the received message
    writeToHistory(message.data);
}
```

When the browser receives a new message from the server, it will simply display it on the `history` div.

```
function doSend() {
    var message = document.getElementById('chat').value;
    if (message != '')
        websocket.send(message); // send the message
    document.getElementById('chat').value = '';
}
```

The `doSend` function captures the content of the input field, clears it and finally sends the old content through the websocket.

By using the `websocket.send()` function and the `websocket.onMessage` event, one can perform a two-way communication with the server, and receive push events from the server without having to waste resources constantly pooling for updates. They are immediately sent to the browser.

3.3 Java code

The relevant part on the Java side is that the handler of a websocket connection is annotated using `ServerEndpoint`, from package `javax.websocket.server`.

Import the supplied files to your favorite Java IDE, by setting up a new project, open `WebSocketAnnotation.java`, and examine the code. Observe that you get the same *events* at the server side as you did on the client side (on-open, on-close, on-message, on-error). This time, on the server side, we handle such events using plain Java.

Assignment: After running the example, as it is supplied, transform it into a chat application. All clients connected to the application should see everyone's messages, and should be able to send messages to everyone.

1. You need to store all connections in a set, to be able to iterate through it and broadcast messages to all clients. You can try using a

   ```
   private static final Set<WebSocketAnnotation> users =
           new CopyOnWriteArraySet<>();
   ```

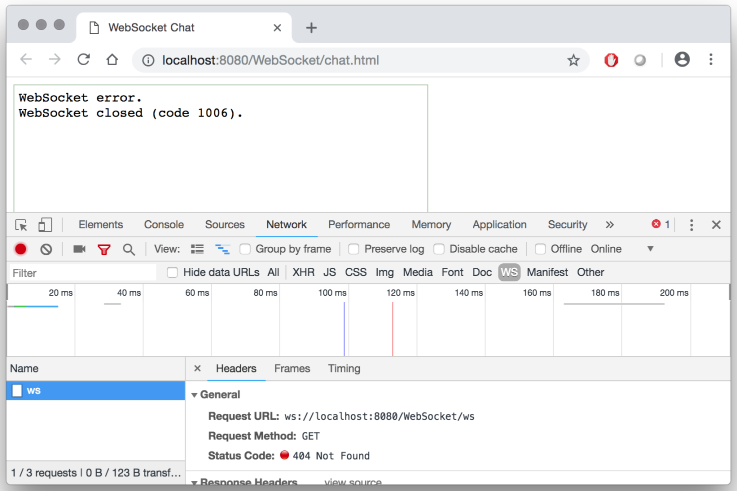
2. With the `users` object, you may execute `users.add(this)` whenever there is an `OnOpen` event (a new user connected), and execute `users.remove(this)` whenever you get an `OnClose` event or some exception related to that user.
3. Finally, upgrade the `sendMessage()` method to iterate through all connections (all items in `users`) and send any message to all of them.

As a suggestion, open your WebSocket chat application on your mobile phone.

4. Add support for multiple chatrooms which users may join and leave.

4 Debugging

The *developer tools* provided by most Web Browsers are very useful to identify problems that affect clients. The example below shows a WebSocket session closed (with status code 1006) due to some error. Opening the developer tools clarifies that the requested URL was not found and this should help solving the problem.



localhost:8080/WebSocket/chat.html

```
WebSocket error.
WebSocket closed (code 1006).
```

ws

General

Request URL: `ws://localhost:8080/WebSocket/ws`

Request Method: `GET`

Status Code: ● `404 Not Found`

1 / 3 requests | 0 B / 123 B transf…

5 Integration with Struts2

When you integrate WebSockets with Struts, the global interceptor will disable the mapping to the server endpoint that you create to handle WebSockets. In order to exclude that interceptor for all WebSocket traffic, add the following line to the `struts.xml` file, together with the other constants:

```
<constant name="struts.action.excludePattern" value="/ws"/>
```