SWitCH – Computing systems and networks (SCOMRED)

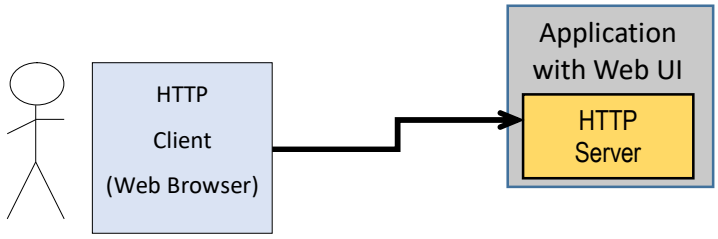
2018/2019

Laboratory Script – Week 07

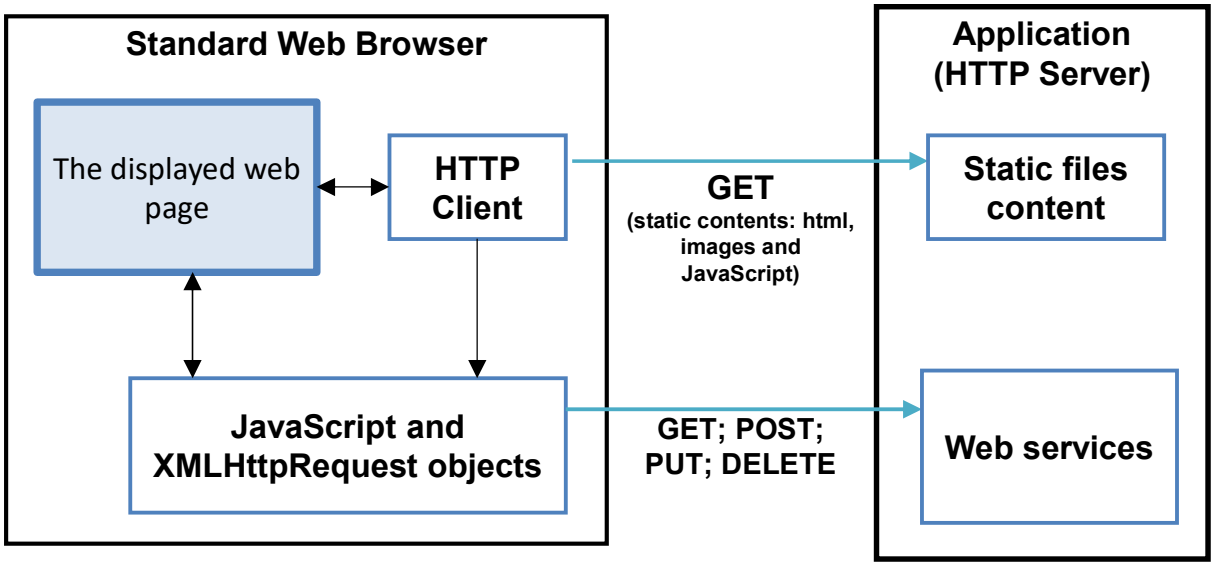
IMPERIOUS: After each practical activity (in yellow), students are supposed to reflect on results, and wait for the teacher's acknowledgment before progressing to the next practical activity in this script.

1. AJAX

On last lab we´ve added an HTTP server to our **TcpChatSrv** project, it's now capable of serving static files contents and also dynamically creating a response content for GET requests on URI **/messages**.



Although in fact most of the hard work is already done, we are now going to make the Web UI do what it's supposed to do, allow user friendly interaction, and for that, AJAX is a must.



The main concept in AJAX is the use if **XMLHttpRequest** objects by JavaScript to create HTTP requests and thus transform the Web Browser in a web services consumer. Despite the XML reference, contents transferred between JavaScript functions and web services can be of any type.

Focusing back on our **TcpChatSrv** project, for its Web UI, main objectives are the following:

- The default root web page (**/index.html**) is loaded only once and yet it will present a constantly updated list of last chat messages exchanged through the server.
- The user must define its nickname and then he can start sending chat messages to the server.

To properly address these objectives we should start by establishing the web services that will be required.

We have already established the URI **/messages**, and that a GET request on it returns a full HTML page with the list of last chat messages.

Nevertheless, for a JavaScript consumer function that can use DOM to replace the content of a single element in the page, it would be more suitable to get just paragraphs with chat messages and not an entire HTML document.

(1) This is one first change on the server side, the **GET /messages** request will now have as response a sequence of **<p>** tags, each with one chat message.

One other web service we must implement is one to allow a JavaScript function to send a chat message to the server. We could use GET because data to be send is just text and can be embedded in the URI, but PUT is a better method for sending data to the server. Because a different method is used, the URI can be the same.

(2) The second change on the server side is adding the **PUT /messages** request processing, it receives a chat message transported in the request's content (body). As with the TCP client (TcpChatCli project), the chat message is and already formatted text line with a nickname prefix enclosed in brackets.

1.1. The server side (web services)

The only class we need to modify is **HttpServerConn** that handles HTTP requests from clients, more precisely its **run()** method:

```
@Override
public void run() {
    try {
        sIn = new DataInputStream(cli.getInputStream());
        sOut = new DataOutputStream(cli.getOutputStream());
        HTTPmessage request = new HTTPmessage(sIn);
        HTTPmessage response = new HTTPmessage();
        response.setResponseStatus("200 OK");

        if(request.getMethod().equals("GET")) {
            if (request.getURI().equals("/messages")) {
                String content = ChatMessages.getLastHTML();
                response.setContent(content, "text/html");
            }
            else { // NOT GET /messages , THEN IT MUST BE A FILE
                String fullname = webRootFolder + "/";
                if (request.getURI().equals("/")) fullname = fullname + "index.html";
                else fullname = fullname + request.getURI();
                if (!response.setContentFromFile(fullname)) {
                    response.setContentFromString(
                            "<html><body><h1>404 File not found</h1></body></html>",
                            "text/html");
                    response.setResponseStatus("404 Not Found");
                }
            }
        }
        else {
            if (request.getMethod().equals("PUT")) {
                if (request.getURI().equals("/messages")) {
                    ChatMessages.push(request.getContentAsString());
                    TcpServerConn.sendToAll(request.getContent().length,
                             request.getContent());
                } else {
                    response.setContentFromString(
                            "<html><body><h1>404 File not found</h1></body></html>",
                            "text/html");
                    response.setResponseStatus("404 Not Found");
                }
            } else {
                response.setContentFromString(
                  "<html><body><h1>ERROR: 405 Method Not Allowed</h1></body></html>",
                  "text/html");
                response.setResponseStatus("405 Method Not Allowed");
            }
        }
        response.send(sOut);
    }
    catch(Exception ex) { System.out.println("HTTP I/O error"); }
    try { cli.close(); } catch(Exception ex) { System.out.println("HTTP I/O error
closing client connection"); }
}
```

If the client's request is using the GET method, then we check if the URI is **/messages**, in that case, the response will have the content returned by **ChatMessages.getLastHTML()**, this will be a sequence of <p> tags. **In the previous version this web service was returning a full HTML page**.

As before for any other **GET** request we assume it must be a static file.

Now the server will also processes **PUT** requests, but only for URI **/messages**, as we have settled, this stands for a chat message being sent to the server. The received chat message was transported in the request's body (content), so we use it to add the new chat message to recently exchanged messages (**ChatMessages.push(request.getContentAsString())**) and send it to all TCP connected clients (**TcpServerConn.sendToAll(request.getContent().length, request.getContent())**).

And that's all, most work was already done when we implemented the **HTTPmessage** class.

If the GET request is not for **/messages** and no matching filename is found, the **404 Not Found** response is sent. If the PUT request is not for **/messages**, the **404 Not Found** response is sent.

If the request method is not GET neither PUT, the **405 Method Not Allowed** response is sent.

PRACTICE:

- Implement the described changes to the **HttpServerConn** class in your **TcpChatSrv** project, there is an updated file in Moodle.

1.2. The client side (HTML, JavaScript and AJAX)

The most significant effort will be on the client side (the Web Browser), all contents are provided by the HTTP server itself and should be stored in files placed in the appropriate folder (web root folder).

1.2.1. The /index.html file

This file is provided by the server when a client sends a **GET /** request, the request web browsers send when the user types a URL.

This page is supposed to present two input text fields, one for the nickname and another for the chat message to be sent. Yet, the text field for the chat message to be sent should be available only once the nickname is settled, so it will be initially disabled.

The page must also present an always updated view of the last chat exchanged messages, this will have to be periodically fetched from the server.

Regarding the use of the page, using a button to send each chat message is not practical, instead they will be sent when the user presses the keyboard's ENTER key. To achieve that with JavaScript a function will be settled to check every typed key and if it's ENTER, then the message is sent.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>TCP Chat Web UI</title>
    <script src="tcpchat.js"></script>
</head>
<body bgcolor="#d3d3d3" onLoad="refreshMessages()">
<img src="SWitCH.png" />
<img src="isep.png" />
<h1>TCP Chat Server Web UI</h1>
<p>Your Nickname: <input id="nickname" type="text" />
    <input id="setnick" type="button" value="Set Nick" onclick="setNick()"/></p>
<p>Send message: <input id="message" type="text" size="80" disabled="true"
onkeydown="enterKey()"/></p>
<hr />
<div id="messages">
    <p>Loading messages from server ...</p>
</div>
<hr />
</body></html>
```

All required JavaScript function will be stored in file named **tcpchat.js**. When the page is loaded, the **refreshMessages()** function is executed, it will be up to it fetching the last chat messages from the server, putting then on the tag with id **messages**, and then keep it updated.

The text field **message** is initially disabled, it will be enabled by the **setNick()** function when the user clicks the corresponding button.

Once enabled, for every key typed in the **message** field, the **enterKey()** function is called to check if it's the ENTER key, if so the field content will be sent to the server.

1.2.2. The /tcpchat.js file

This file is provided by the server and contains the JavaScript function to be used by the **/index.html** page's elements.

The **refreshMessages()** function is executed for the first time when the page is loaded:

```
function refreshMessages() {
        var request = new XMLHttpRequest();

        request.onload= function upDate() {
        if(this.responseText == "") document.getElementById("messages").innerHTML=
                "<b>Sorry, no messages yet</b>";
        else document.getElementById("messages").innerHTML= this.responseText;
        setTimeout(refreshMessages, 1500);
        };

        function errorResult() {
                document.getElementById("messages").innerHTML=
                        "Server unreachable, still trying ...";
                setTimeout(refreshMessages, 1000);
        };

        request.ontimeout=errorResult;
        request.onerror=errorResult;

        request.open("GET", "/messages", true);
        request.timeout= 5000;
        request.send();
}
```

It starts by creating an **XMLHttpRequest** object and then establishing call-back functions for properties regarding several result events: onload; ontimeout; onerror.

The request is settled (GET /messages), also a response timeout (5 seconds) and the request is finally sent.

Once there is a result for the request, the corresponding call-back function is executed.

In case of success, the **upDate()** function is executed, the response's content is available on property **responseText**, if it's empty then there are no messages yet, then the **messages** element in the page is replaced with a suitable text in bold. Otherwise, the received content is used to replace the **messages** element's content.

Either way, a new execution of the **refreshMessages()** function is scheduled to happen one and a half seconds after.

In the event of timeout or another error the **errorResult()** function is executed, it will replace the messages being displayed by an adequate message, and then a new execution of the **refreshMessages()** function is scheduled to happen one seconds after.

The **enterKey()** function is executed every time the user type a key on the **message** text field. By checking the **keyCode** attribute of the **event** object we can tell if it was the ENTER key (code 13), if not so, the function's execution ends.

```
function enterKey() {
    if(event.keyCode != 13) return;
    if(document.getElementById("message").value=="") return;
    var request = new XMLHttpRequest();
    request.open("PUT", "/messages", true);
    request.setRequestHeader("Content-type", "text/plain")
    request.send("(" + document.getElementById("nickname").value + ") " +
        document.getElementById("message").value);
    document.getElementById("message").value="";
}
```

If the user has pressed ENTER then an additional check is made to see if the **message** element is empty, if so the function also ends.

If there is something to be sent, then a new **XMLHttpRequest** object is created, the HTTP request to be sent is settled (**PUT /messages**).

Because it will be a PUT request with a text content (body), the **Content-type** header line is settled to **text/plain**.

Finally the request message is sent, the content is text, containing the nickname enclosed in brackets followed by the chat message itself.

To give a feedback to the user that the message was sent and he can now type another one, the text field **message** is cleared.

In **tcpchat.js** file there's yes another function called **setNick()**, it has nothing to do with AJAX. This function is used to settle a nickname, disable the chance to change it, and enable the **message** input text field so the user can then send messages.

PRACTICE:

Download the new **index.html** and **tcpchat.js** files from Moodle into you **TcpChatSrv** project's **www** folder.

Testing locally:

1st Start your server (**TcpChatSrv** project) and leave it running.

2nd Use your personal Web Browser to open URL **http://localhost:8080**. You should now see the new **index.html** file content. Define a nickname and test it.

3rd Open another browser's window/tab and open the same URL, use a different nickname. Test it. Check that chat messages sent by a client are presented to the other client.

4th Run the **TcpChatCli** project and use a yet different nickname. Test it, check that all clients (Web UI clients and TCP clients) are able to use the server.

5th Use a **postman** application to send a **PUT /messages** request to your server with the appropriate content.

6th Use a **postman** application to send a **GET /messages** request to your server, in fact, because it's a GET request, you could also use your browser directly.

Testing remotely:

1st Settle a team of nearby partners to share the same **TcpChatSrv** server, one team member will run the server and share with others the IP address his laptop is using (SERVER-IP-ADDRESS).

Notes regarding wireless restrictions apply here again. To remove this possible obstacle all team members can connect to a DEI VPN service. Also the team member providing the service should allow incoming traffic for the server application he is running.

2nd All team members are now going to connect to the same server by HTTP, so they can type on their personal Web Browser the URL: **http://SERVER-IP-ADDRESS:8080**. As before each team member should use a different nickname to make things clearer, unique nicknames are not enforced by these applications.

3rd Some team members should also try the **TcpChatCli**. On the **TcpChatCli** project's runtime definitions they should settle the first command line argument of be the IP address of the team member providing the service (SERVER-IP-ADDRESS).