

Study Point Exercise-1 - HTTP servers and the HTTP protocol

- You can earn a maximum of **5** points for this exercise, one for each of the steps 1-5.
- If you hand in via mail you can get the additional "attendance point" if your score is **three** or above. If it's not, you should probably have attended the class to get help ;-)
- Step-6– we are awaiting a solution to the Azure problem.

How to hand in: All exercises must be included in ONE single NetBeans project, which must be committed and pushed to your Git-hub account no later than Friday **14.30**. If you attend the class, you can demo your solution up until **15.45**.

Send a mail to gangof5ive@gmail.com including the following:

Topic: Study Point Exercise-1

Content: First line should be your full name, second line, the link to your Git-hub repository.

The exercises below use the FirstHttpServer, which we did together in the class, as a starting point.

Put your project under version control and commit it as "starting point for exercise".

You can make a clone from here, if you have problems with your own solution: <https://github.com/Lars-m/firstHttp.git>

1) Content-type with the response

Does your server provide the browser with a content type (use Google Developer Tools to get the answer)?

Press Ctrl-u in your browser to see what actually is sent.

It seems like your browser is forgiving and "guesses" that the context is text ("txt/plain").

Let's see what happens if we provide the browser with some html. Add the following code (use cut-and paste) after the declaration of the response string:

```
StringBuilder sb = new StringBuilder();
sb.append("<!DOCTYPE html>\n");
sb.append("<html>\n");
sb.append("<head>\n");
sb.append("<title>My fancy Web Site</title>\n");
sb.append("<meta charset='UTF-8'>\n");
sb.append("</head>\n");
sb.append("<body>\n");
sb.append("<h2>Welcome to my very first home made Web Server :-)</h2>\n");
sb.append("</body>\n");
sb.append("</html>\n");
response = sb.toString();
```

What do you see in the browser? Press **Ctrl-U** in your browser to see what actually is sent.

It seems like the browser was able to guess that this was **html**, solely from the content. Browsers are allowed to guess the type of data they receive. But a well written server will "provide" clients with this information via the **Content-Type** header.

For everything you do forward you are expected to set the content-type for the response.

Let's set the content type to the html-mime type (even if the browser could guess this from the content).

Add the following line before the call to `sendResponseHeaders(..)`:

```
Headers h = he.getResponseHeaders();  
h.add("Content-Type", "text/html");
```

Compile and start the server. Verify, using Chrome Developer Tools, that the response includes the Content-Type header.

2) Obtain the Request Headers (Create a dynamic response)

Add a new route `"/headers"` to your server which should serve requests like: <http://localhost:8080/headers>

The response should be an html page with a table that shows all the HTTP-headers as sketched below:

Header	Value
Cache-control	[max-age=0]
Host	[localhost:8080]
Accept-encoding	[gzip,deflate,sdch]
Connection	[keep-alive]
Accept-language	[da,en-US;q=0.8,en;q=0.6,en-GB;q=0.4]
User-agent	[Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.143 Safari/537.36]
Accept	[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8]

Hints:

The RequestHandlers `HttpExchange` object contains a method `getRequestHeaders()` which return a Map with the HTTP headers (key) and values (values).

Iterate over this Map (the keys) and build the table rows dynamically. Use the basic html code given in the previous step, as the starting point for the page you generate.

When completed, verify via Google Developer Tools that all headers sent from the browser can be reached by the server.

Hints for you CA: In the CA you have to build two dynamic pages. One that show the number of online users on the chat, and one that shows the content of the Chat servers log file. This exercise has illustrated one way to create a dynamic page.

Commit this as "Added page that shows Request Headers"

3) The worlds simplest file server ☺

In this step we will create a route that shows a file from the servers file system. In a later step we will expand this exercise to allow users to read all files in a public folder, but here we will return the same file always.

Add a new route to your server: `"/pages/"` and a corresponding RequestHandler.

Provide the class with a field `String contentFolder= "public/"`. You could let users change this value (select another folder) via the command line arguments as for *port* an *ip*. This value indicates the folder that contains public content.

Create the folder in the root of your project, and copy the html file with the Form (ex-9) from the exercises day-2 into the folder.

You can use the code below for the RequestHandler `handle(..)` method, but make sure you understand the code, you have to expand this method in a later exercise:

```
public void handle(HttpExchange he) throws IOException {
    File file = new File(contentFolder+"index.html");
    byte[] bytesToSend = new byte[(int) file.length()];
    try {
        BufferedInputStream bis = new BufferedInputStream(new FileInputStream(file));
        bis.read(bytesToSend, 0, bytesToSend.length);
    } catch (IOException ie) {
        ie.printStackTrace();
    }
    he.sendResponseHeaders(200, bytesToSend.length);
    try (OutputStream os = he.getResponseBody()) {
        os.write(bytesToSend, 0, bytesToSend.length);
    }
}
```

Commit this as "Added Worlds simplest Web-Server"

4) Obtaining GET and Post parameters.

Obtaining parameters from a request is surprisingly cumbersome compared to “real” web servers.

This is however OK, for our purpose, since we are mainly using this server to demonstrate the HTTP protocol.

Remember, GET parameters are encoded in the URL as in: <http://myserver.dk?name=lars&job=teacher> whereas POST parameters are sent with the request body.

So in order to read the get Parameters we need access to the requestUri, and to read Post parameters we need to read the requests InputStream.

Create a new route “Parameters/” and a corresponding RequestHandler.

Copy or refactor the basic code for an html-page from one of the other handlers.

Add code so the page prints “Method is: xx” where xx is obtained from the HttpExchange objects `getMethod()` method.

For a GET-request add this to the page “Get-Paremters: xxx” where xxx are the unparsed GET- parameters (if any) for the request obtained via `HttpExchanges getRequestURI().getQuery()`.

For a POST-request add this to the page: “Request body, with Post-parameters: xxxx” where xxxx is content read from the request’s input stream (Use the code below).

```
Scanner scan = new Scanner(he.getRequestBody());
while(scan.hasNext()){
    sb.append("Request body, with Post-parameters: "+scan.nextLine());
    sb.append("<br>");
}
```

Test this page using the Form you added in the part “*The worlds simplest file server*”, by changing the Forms action into `action="../parameters"` and set the method attribute to GET. [Understand what you did here.](#)

Test and verify that we can access GET-parameters sent from a client in our server.

Change the method attribute in your Form into POST and test and verify that we can access POST-parameters sent from a client in our server.

Commit this as “Reading Get and Post parameters”

5) Expanding the worlds simplest file server

Having created the worlds simplest file server is not a record worth holding ;-) so let's expand it to allow us to fetch "all" content placed in the "public" folder.

Add an image, a pdf document, a jar-file (just use the one generated by the current project) to the public folder.

Hints:

Instead of the hardcoded file name, we need to fetch the name of the requested file from the request URL.

In the previous exercise we saw how we could obtain the request URI via `HttpExchange's getRequestId()` method. For a request like <http://localhost:8080/pages/myHtml.html> this method will return a URI which when we call `toString()` on it, will return `"/pages/myHtml.html"`.

Parse this string to get the file name, and use it instead of the hardcoded file name.

Test and verify that you can fetch the files added to the public folder (image, pdf and jar), even if we haven't set the Content-type on the response.

Set the Content-Type for the Request

Now, parse the input string to get the extension (.html, .pdf etc) and set the Content-Type according to this (find the correct mime type for a given file type here: http://en.wikipedia.org/wiki/Internet_media_type)

Example:

Pdf → `application/pdf`

Test and verify via Chrome Developer Tools that the correct Content-Type is set.

Add a new html file to the public folder, including an image and an external css-file, and verify that our server easily servers files with external content (verify that this requires three http requests via Google Developer Tools).

Error handling

Type `localhost:8080/bla` and explain the result. Implement a similar behaviour if users are requesting a non-existing file.

Commit this as "File server handles several filetypes and are no longer hardcoded to one file"

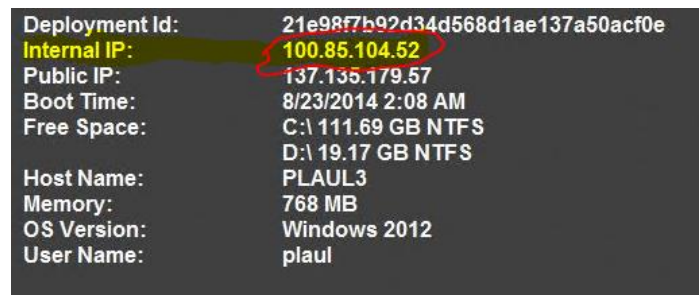
6) Deploy the Server to Azure

Before you complete this step, you should have completed the exercise **JavaOnAzure.pdf**.

This exercise will guide you through the steps of setting up a Virtual Machine with Java installed on Azure.

When the VM is ready you can install new applications the same way as you did in the section "To install a Java application server on your virtual machine".

- Replicate the steps from this section, but this time upload the **dist** folder generated by your NetBeans http server project.
- Your life will a lot easier if you have generated a *.bat file.
- Remember to change the bind address (preferably in the .bat-file) to the internal IP of your Virtual Machine (see below)



Deployment Id:	21e98f7b92d34d568d1ae137a50acf0e
Internal IP:	100.85.104.52
Public IP:	137.135.179.57
Boot Time:	8/23/2014 2:08 AM
Free Space:	C:\ 111.69 GB NTFS
	D:\ 19.17 GB NTFS
Host Name:	PLAUL3
Memory:	768 MB
OS Version:	Windows 2012
User Name:	plaul