The HTTP Protocol

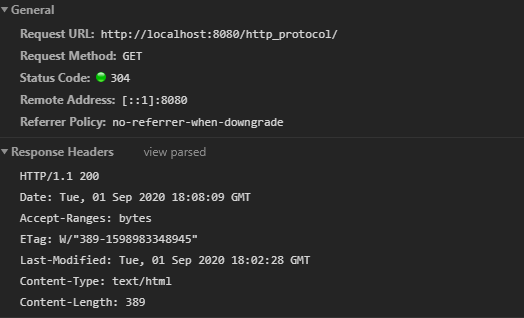
1) Monitoring HTTP Headers 1

*Create a new NetBeans Maven Web-project.*

*For this exercise, we will just use the default index.html generated by NetBeans.*

*Press the run button. When you see the file in the browser (Chrome), open the network tab in the developer window (Windows: Ctrl-shift-j / Mac: Option-Command-i) and press F5*

*Observe and explain each of the values monitored (use view source to see the plain messages).*



Hints: In order to better observe the values related to Caching you might need to:

Go back to NetBeans and rename your file to index1.html

Go back to your browser and (while the developer window is open) change the url to point to the new file.

Observe the values

Press F5 and observe the values again.

Explain what you see.

**General**

**Status code “304”:** This status code means that the requested data has not been modified from the previously downloaded data (stored in cache). I got this status code because I had run the project before taking the screenshots above. Had it been the first time, meaning I wouldn’t have any cached data for the domain, the status code would likely have been 200 – meaning the request was successful (and the response contained the resource that was requested).

**Remote address:** I assume this header just means localhost on port 80 (HTTP protocol port).

**Referrer Policy:** The value “no-referrer-when-downgrade” means navigating to a different URL would include a Referrer HTTP header with the same value as the original URL.

**Response Headers (sent by the server)**

**HTTP/1.1 200:** The first part of this header’s value means that the server is responding using the HTTP protocol version 1.1. The second part, 200, is a status code meaning that the response was successfully sent, and everything is ok.

**Date:** This general header’s value is basically just a timestamp for when the response was created and sent (originated).

**Accept-ranges:** This header indicates that the server supports partial requests. Its value, “bytes”, indicates that bytes can be used to define the range of the partial requests. This is part of a process called byte serving, where only a portion of an HTTP message (response specifically) is sent from the server to the client.

**Etag:** This header is used as an identifier for a specific version of a resource. It helps make caching more efficient as the server does not need to resend a full response if the content has not been changed. The value is the specific version of the resource (that being the index.html file in this case).

**Last-modified:** This header’s value displays the specific time the origin server believes the resource was last modified.

**Content-type:** This entity header tells the client what the content type of the returned content is, which in this case is text and html.

**Content-length:** This entity header indicates the size of the entity body in bytes sent to the recipient.

**Request Headers (sent by the client)**

**Request Method:** This request header specifies which HTTP method to use, and in this case the GET method was used to request a representation of the specified resource (index.html page in the project). The header also contains the project name and HTTP version.

**Host:** This request header specifies the host (localhost) and port number (8080) of the server to which the request was sent.

**Connection:** This general header controls whether the network connection stays open after the current transaction. In this case, the value is “keep-alive”, meaning the connection will remain open to future requests.

**Cache-control:** This header holds directives (instructions) for caching in both requests and responses. The value “max-age=0” indicates that a cache entry is stale and requires re-validation. This does not prevent further caching.

**Upgrade-Insecure-Requests:** This request header sends a signal to the server expressing the client’s preference for an encrypted and authenticated response. This basically means the client is telling the server that it supports the upgrade mechanisms of the *upgrade-insecure-requests* CSP directive.

**User-Agent:** This request header lets the server receiving the request identify the user-agent, acting on behalf of the client. In this case, its value shows that the user-agent is a browser. It does not, however, specify which specific browser (I am using Chrome) but instead lists various commonly used browsers. It does seem to include some system information (Win x64), along with versions of the browsers listed.

**Accept:** This request header specifies which content types the client can understand. Using what is called “content negotiation”, the server chooses one of the proposed content types to use for its response and informs the client of its choice with the **Content-Type** response header. The value in this case shows that the client accepts, among other content types, text and html, which were both listed in the Content-Type response header as seen above.

**Sec-Fetch-Site:** This metadata header indicates the relationship between the request initiator’s origin and the origin of the resource. Its value, “None”, means the request does not relate to any context like site, origin, or frame. In other words, there is no relationship or connection between the two.

**Sec-Fetch-Mode:** This metadata header indicates the request’s mode. This header is used to distinguish resource loads from navigation requests. For example, the value “navigate” indicates a top-level navigation request, meaning the request involves going somewhere (the webpage, index.html) and not loading a resource.

**Sec-Fetch-User:** This metadata header indicates whether the navigation request was triggered by a user activation. The value “?1” means the navigation request was triggered by something other than user activation – I suppose refreshing the page does not count as user activation?

**Sec-Fetch-Dest:** This metadata header indicates the request’s destination and how the fetched data will be used. The value, “document”, shows that request came from a top-level navigation (refreshing the page would probably count as one, which is what I did to display the header information).

**Accept-Encoding:** This request header indicates which content encoding types the client can understand. Like the Accept header, content negotiation lets the server choose one of the proposed encoding types when creating and sending the response. The selected type can be seen in the **Content-Encoding** response header, however it is not present in this case. The values are a list of the different types of content encoding the client understands.

**Accept-Language:** This request header indicates which (natural, not coding) languages the client is able to understand, and which language the client prefers. The values indicate that Danish and English are acceptable for the client.

**Cookie:** This request header contains stored cookies previously sent by the server with the **Set-Cookie** header. There are two cookies in the Cookie header in this case, separated by a semicolon.

**If-None-Match:** This request header makes the request conditional and makes the server send back the requested resource with a 200 status when using the GET and HEAD methods, but only if it doesn’t have an ETag matching the given ones. If the condition fails, the server returns the status code 304 (Not Modified). Since the value in this header is the same as the Etag header value in the response, it returned the status code 304 as seen above.

**If-Modified-Since:** This request header makes the request conditional by making the server send back the requested resource with a 200 status ONLY IF it was last modified after specified date (the value in the header).

2) Monitoring HTTP Headers 2

*Add an image to the page*

*Add an external style sheet to the page <link rel="stylesheet" type="text/css" href="myStyle.css">*

*Reload the page again, observe the request(s) being made, and explain the purpose of the connection header.*

3 requests were made, and the responses contained: http\_protocol/ (project name and location of the index.html file), smiley.png (image), and myStyle.css (empty CSS file). Only the first of the three mentioned had a connection header, which makes sense since it is the root project containing the other two. This also seems to be the case based on the “Initiator” column, where both “myStyle.css” and “smiley.png” have “index” as their initiator, and “http\_protocol/” has “:8080/http\_protocol” as its initiator.

The purpose of the connection header is to control whether the network connection stays open after the current transaction finishes, as explained above. Like before, the value is “keep-alive”, meaning the connection remains open to more requests.

3) Monitoring HTTP Headers 3 (Response-codes 3xx)

*In the Web-project, created for 1+2, add a new HTML-page called* ***r.html*** *and add this text in an h1-tag “****You got redirected to me****”.*

*Use the Wizard to create a servlet called redirect*

*Remove the processRequest and the doPost method completely from the generated servlet-code.*

*In the doGet(..) method replace the call to processRequest with this line: response.sendRedirect("r.html");*

*While your server is running, open a (Chrome) browser, and Developer Tools and the network tab.*

*Enter the address for the servlet (****http:localhost:8080/redirect)*** *into the browser and explain:*

* *The two HTTP-request you see*

I see the following requests: **redirect** and **r.html.** The first, redirect, has the status code 302, meaning the resource requested has been temporarily moved to the URL given by the **Location header**. The location header’s value is **r.html**, which is the second request made, and this means that the first request required a second request to be made to fetch the r.html page so it could be displayed.

The second request, r.html, has the status code 304, meaning there is no need to retransmit the requested resources. This is probably because I refreshed a couple of times back and forth before writing this down, meaning the file had already been downloaded and cached.

* *How the browser knew where to go in the second request*

Maybe the order of requests is different for me, but it seems to me like the first request is what told the browser where to go, and the second was the resource that was to be displayed upon arrival. Either way, the browser knew where to go because we set up the **redirect** servlet that redirects to r.html when using the GET method and correct URL pattern (“/redirect”), and because of the first request’s Location header specifying that it should go to the r.html page.

3a) Redirecting to HTTPs instead of HTTP

*In Chrome enter this address (with the developer window + the network-tab open), and exactly as it is spelt:* [***http://studypoints.info***](http://studypoints.info)

*Explain the first two requests monitored (notice where you requested to go, and where you actually ended).*

The first two requests (based on the waterfall column) are both called “studypoints.info”. The first has the status code 301, indicating that the requested resource has been moved to the URL given by the Location headers (<https://studypoints.info/>), and the second has the status code 200, meaning the request was successful and everything is ok.

The interesting part about the first request is that it redirected me from <http://studypoints.info> to <https://studypoints.info/> (as indicated by the Request URLs of both requests), meaning that the server’s response indicated that the website was no longer an HTTP website but instead an HTTPS website (which is more secure due to encryption of requests and responses). The server recognized the URL, but automatically redirected me to the HTTPS version instead.

The second request was the actual webpage I was redirected to, as is indicated by the Remote Address header which contains the value: 157.230.21.145:**443**. The port number 443 is specifically for HTTPS, thus confirming that this request was indeed made to download the HTTPS version of the studypoints website. This is also corroborated by the fact that the value of the Remote Address header for the first request is: 157.230.21.145:**80**.

*Important: Later this week, you will learn how to set up your own server to use https,* ***and ONLY https****.*

4a) Status Codes (5xx)

*Use the Wizard to create a servlet called* ***Ups***

*In the processRequest(..) method,  just before the try-statement add this code:*

*int result = 100/0;*

*While your server is running, open Chrome developer tools and the network tab and then call the servlet.*

*Write down the response code generated by the server as for the previous exercises*

I can see that one request was made: “ups”. The request has the status code 500, which is an Internal Server Error response code sent when the server encounters an unexpected condition that prevents it from fulfilling a request. In this case, it’s because dividing by 0 is impossible, so when the servlet tried to use the *processRequest()* method, an ArithmeticException occurred. As for the response code:

**Connection:** The value is “close” here, meaning the connection to the server was closed when the exception occurred.

As for the actual response code generated as an HTML page:

# HTTP Status 500 – Internal Server Error

**Type** Exception Report

**Message** / by zero

**Description** The server encountered an unexpected condition that prevented it from fulfilling the request.

**Exception**

java.lang.ArithmeticException: / by zero

servlets.Ups.processRequest(Ups.java:35)

servlets.Ups.doGet(Ups.java:62)

javax.servlet.http.HttpServlet.service(HttpServlet.java:626)

javax.servlet.http.HttpServlet.service(HttpServlet.java:733)

org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53)

**Note** The full stack trace of the root cause is available in the server logs.

As can be seen, it basically describes what I mentioned above, with specific line numbers provided for error correction. The rest of the response header information is not really anything out of the ordinary.

4b) Status Codes (4xx)

*While your server is running, open Chrome developer tools and the network tab, and call this address:* ***http://localhost:8080/i\_dont\_exist***

*Write down the response code generated by the server as for the previous exercises*

# HTTP Status 404 – Not Found

**Type** Status Report

**Message** The requested resource [/http\_protocol/i\_dont\_exist] is not available

**Description** The origin server did not find a current representation for the target resource or is not willing to disclose that one exists.

### Apache Tomcat/9.0.37

As can be seen, this request resulted in the status code 404, meaning the requested resource was not found on the server.

4c) Status Codes - Ranges

*Your document, containing the Status Codes for all the exercises done so far, should now contain codes like 2xx, 3xx, 4xx and 5xx.*

*Explain (write down your answer so you won’t forget) the meaning of the first digit in the 3-digit Status Codes you have seen so far.*

HTTP status codes are split into five categories:

1. **Information responses (100-199)**
2. **Successful responses (200-299)**
3. **Redirects (300-399)**
4. **Client errors (400-499)**
5. **Server errors (500-599)**

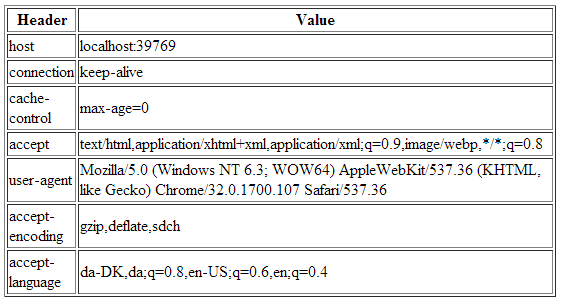
I have not encountered any **information response** status codes yet, but they seem to be used for interim processes (a request was received, processing continues). **Successful response** status codes are used when requests are successfully received, understood, and accepted. **Redirect** status codes are used when further action needs to be taken to complete a request (like making another request to get an additional file, like in exercise 3a).

**Client error** status codes are used when the request contains poor syntax or cannot be fulfilled. **Server error** status codes are used when the server failed to fulfil an ostensibly valid request.

5) Get HTTP Request Headers on the Server

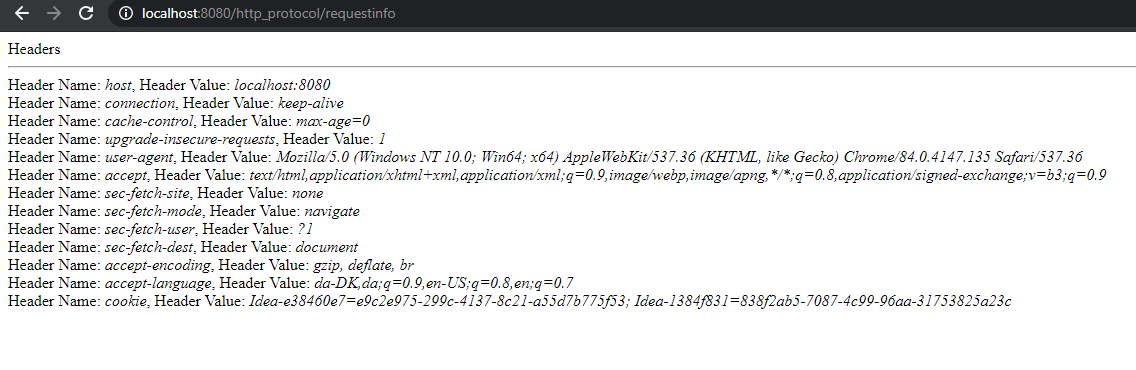
*We have seen that an HTTP request from a Browser typically includes a lot of headers with information related to the client.*



*This information is available to a servlet (actually to any web-server technology)  via the request object. Create a Servlet, which should output this information in a table as sketched in this figure (or in any way you like, but don’t focus on presentation).*

*Hints: Use the request objects getHeaderXXX methods.*

Here is a picture of how I chose to display the header information:



6) Get/Post-parameters

*Create a new HTML-file in the web-project made in exercise 1.*

*Add a form to the file, including two text input boxes and a submit button as sketched below:*

*Add an extra input field to the form with type=”hidden”, name=”hidden” and value=12345678.*

*Add the value “#” for the forms action attribute.*

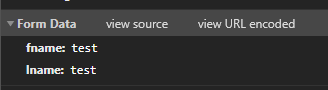
*Set the forms method-attribute to the value “GET” (actually the default value) and test the form. Observe what happens in your browser's address field.*



As can be seen, both form parameters and the values given can be seen in the address field. I believe the # symbol at the end is from the form’s action attribute, and would normally be something like “/newdestination.html”.

*Change the forms method-attribute to the value “POST” and test the form. Observe the change in your browsers address field. Figure out (using Chrome Developer Tools), how parameters are passed in, for a POST request.*





As can be seen, the address field no longer contains the form parameters or their values and instead just has the “#” symbol, indicating that the form was successfully sent (the form’s action parameter has the value “#”). Additionally, the request body now contains the form parameter values in a segment called “Form Data”, as can be seen in the second screenshot.

Write down your observations

Session and Cookies

*For the next two exercises/demos you should create a new Maven web-project. Both the demos use a Servlet.*

7)      Sessions (Session Cookies)

*In your web project use the wizard to generate a  new Servlet*

1. *Enter* ***SessionDemo*** *as the name of the Servlet and servlets as package name*
2. *Right-click the file and select Run to see “what is does”*
3. *Change the generated processRequest(..) method as sketched below.*

*protected void processRequest****(****HttpServletRequest request****,***

*HttpServletResponse response****)***

***throws*** *ServletException****,*** *IOException* ***{***

*String name* ***=*** *request****.****getParameter****(****"name"****);***

***if******(****name* ***!=******null)******{***

*request****.****getSession****().****setAttribute****(****"name"****,*** *name****);***

***}******else******{***

*name* ***=******(****String****)*** *request****.****getSession****().****getAttribute****(****"name"****);***

***}***

*response****.****setContentType****(****"text/html;charset=UTF-8"****);***

***try******(****PrintWriter out* ***=*** *response****.****getWriter****())******{***

*out****.****println****(****"<!DOCTYPE html>"****);***

*out****.****println****(****"<html>"****);***

*out****.****println****(****"<head>"****);***

*out****.****println****(****"<title>Servlet SessionDemo</title>"****);***

*out****.****println****(****"</head>"****);***

*out****.****println****(****"<body>"****);***

***if******(****name* ***!=******null)******{***

*name* ***=******(****String****)****request****.****getSession****().****getAttribute****(****"name"****);***

*out****.****println****(****"<p> Welcome "* ***+*** *name* ***+*** *" !</p>"****);***

***}******else******{***

*out****.****println****(****"<h2>Please enter your name, and submit</h2>"****);***

*out****.****println****(****"<form action='SessionDemo'>"****);***

*out****.****println****(****"<input type='input' name='name'>"****);***

*out****.****println****(****"<input type='submit'></form>"****);***

***}***

*out****.****println****(****"</body>"****);***

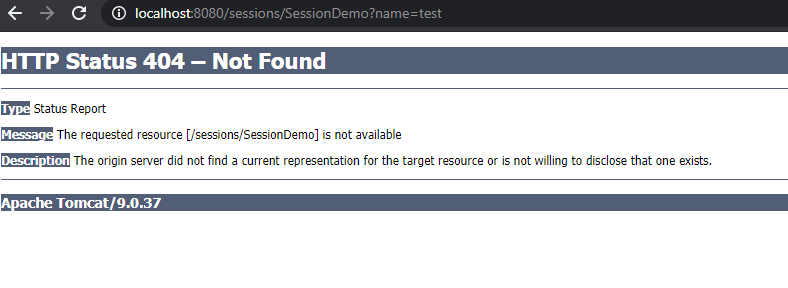
*out****.****println****(****"</html>"****);***

***}***

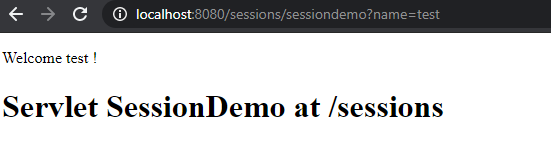
***}***

1. *Enter your name and press submit, copy the URL in the browser into your clipboard, close the tab (but not the browser) and load the page again in a new tab using the URL in the clipboard.*
2. *While doing the things in step d, you should monitor the content of your local cookies and the HTTP requests being sent, using the development tools in Chrome.*
3. ***Most import part of this exercise:***

*Explain (on paper) using both words and images how the Server can maintain state between subsequent calls even when the state is not transmitted from the client to server.*



As can be seen, after inputting the name “test” in the text field and clicking “send”, I get the status code 404 meaning the requested resource was not found. It seems that the servlet or browser by default changes “sessiondemo” to “SessionDemo”. I cannot seem to find a way to fix this in NetBeans, but it is easy to fix in the browser by just manually changing the word back to all lowercase letters:



A new cookie has also been sent by the server after successfully getting to the above screen with the name “test”:



There are two other cookies which seem to be the same ones seen in previous exercises. This new cookie has the name “JSESSIONID”, which tells me that a cookie was created on the session (request.getSession()) when I created a user named “test” (not literally, but in the name “test” received a unique ID (the value after the cookie name) that the server can identify it with).

As for how the server can maintain state despite the client not transmitting the state, I believe it is able to through the Connection header:



Since its value is “keep-alive”, the network connection between the client and server remains open regardless of the number of requests. The session generated when “logging in” as “test” is stored on the server, and whenever the client tries to go to the page with the name “test”, the request cookie is compared to the session ID stored on the server. If it matches, the user gets through to the page shown above (the one with “Welcome test!”).

*Write down your observations*

*8)     Persistent Cookies*

1. *In your web project, use the wizard to generate a new servlet*
2. *Enter CookieDemo as the name of the Servlet and servlets as package name*
3. *Change the generated processRequest(..) method as sketched below.*

*protected void processRequest****(****HttpServletRequest request****,*** *HttpServletResponse response****)***

***throws*** *ServletException****,*** *IOException* ***{***

*String name* ***=*** *request****.****getParameter****(****"name"****);***

***if******(****name* ***!=******null)******{***

*Cookie cookie* ***=******new*** *Cookie****(****"username"****,*** *name****);***

*cookie****.****setMaxAge****(****60* ***\**** *60* ***\**** *24* ***\**** *365****);***

*response****.****addCookie****(****cookie****);***

***}***

*Cookie****[]*** *cookies* ***=*** *request****.****getCookies****();***

***if******(****cookies* ***!=******null)******{***

***for******(****Cookie cookie* ***:*** *request****.****getCookies****())******{***

***if******(****cookie****.****getName****().****equals****(****"username"****))******{***

*name* ***=*** *cookie****.****getValue****();***

***}***

***}***

***}***

*response****.****setContentType****(****"text/html;charset=UTF-8"****);***

***try******(****PrintWriter out* ***=*** *response****.****getWriter****())******{***

*/\* TODO output your page here. You may use following sample code. \*/*

*out****.****println****(****"<!DOCTYPE html>"****);***

*out****.****println****(****"<html>"****);***

*out****.****println****(****"<head>"****);***

*out****.****println****(****"<title>Servlet CookieDemo</title>"****);***

*out****.****println****(****"</head>"****);***

*out****.****println****(****"<body>"****);***

***if******(****name* ***!=******null)******{***

*out****.****println****(****"<p> Welcome "* ***+*** *name* ***+*** *" !</p>"****);***

***}******else******{***

*out****.****println****(****"<h2>Please enter your name, and submit</h2>"****);***

*out****.****println****(****"<form action='CookieDemo'>"****);***

*out****.****println****(****"<input type='input' name='name'>"****);***

*out****.****println****(****"<input type='submit'></form>"****);***

***}***

*out****.****println****(****"</body>"****);***

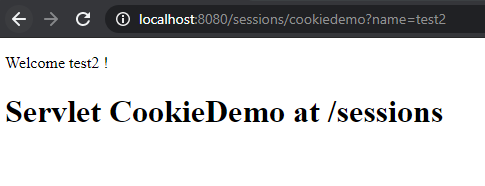
*out****.****println****(****"</html>"****);***

***}***

***}***

1. *Enter your name and press submit, copy the URL in the browser into your clipboard, close the tab (but not the browser) and load the page again in a new tab using the URL in the clipboard.*
2. *Now close your browser (you could even close your laptop, but don’t ;-) , open it again and load the page again using the URL in the clipboard*
3. *While doing the things in step e, you should monitor the content of your local cookies and the HTTP requests being sent, using the development tools in Chrome.*
4. ***The most import part of this exercise:***

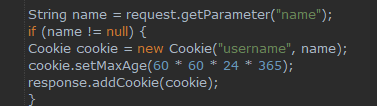
*Explain (on paper) how Cookies can be used to maintain “state” on the client between subsequent calls to a server, even when a browser has been closed down.*



As can be seen, the “welcome” screen for the CookieDemo servlet is the same as the one for the SessionDemo servlet. The new cookie provided upon login, however, is different from the previous one:



Instead of its name being “JSESSIONID”, it is now simply “username” and its value is the name “test2” rather than a long ID string. This is because of the pasted code, which generates a Cookie object:



Another interesting thing here is that the Cookie’s max age has been set to 1 year (in seconds). Because of this, I can close the browser or even turn off my pc completely and get back to the welcome page, provided I have the correct URL. I could even wait several months (though the server would have to be running, so I would have to start it to get in). This is what allows the client to maintain state between subsequent calls to the server, since the Cookie’s lifespan means it can be used to “log in” as the “test2” user for an entire year.

A cookie is stored locally (typically somewhere in the browser’s folders) once sent from the server to the client. Therefore, as long as it is on my machine, I will be able to log in as “test2” provided I have the correct URL and the server is running.

