



Computer networks - 1DV701

Assignment 2



Author: Michael Johansson & Jakob Heyder
Semester: VT 2017

Contents

1	Assignment summary	1
1.1	Small summary of our server	1
1.2	Good to know	1
2	Problem one	2
3	Problem two	4
3.1	VG.1 HTTP status code implementation	5
3.2	VG.2 POST vs PUT	6
4	Problem three	8

1 Assignment summary

- **Michael** did the most of the report and implemented exceptions handling. Wrote some methods in the code and did tests.

Work percentage 45%

- **Jakob** did most of the server code, with the major parts of the functionality. He did the Javascript pages for PUT/POST. He also made sure that we had a good ground to stand on with alot of enumeration types for MIME types etc.

Work percentage 55%

1.1 Small summary of our server

It all starts with that we have our class HTTPReader that reads and parse the the request and then return a HTTPRequest object.

This HTTPRequest we use in our HTTPResponseFactory class that is the heart of our server where we take the request and build our response from that. So in our factory we check if the request was a GET, POST or PUT and respond accordantly.

When the factory then is done building the response we write it out to the outputStream as an byte array stream.

If there is any error we throw an exception and from that exception create different responses that we send to the client.

1.2 Good to know

- We have an out commented code where we can test our HTTP version not implemented exception.
- We choose to close the connection after each request/response but we could alternatively left it open and let the server time out the connection
- Only GET,POST and PUT are implemented HTTP metods.
- Our POST/PUT gives Unsupported Media type for uploading something that is not in our MIME types.
- ../ and any URI with secret are restricted and will give 403.
- Due to the human readability issues of the hash-function generated filenames in our PUT/POST, we display a list of all images in the /images/ folder when trying to PUT an image, so if the client want to update an image he can look up his hash for the image and from that update that file.

2 Problem one

We choose to create an Index file every time an user requests a directory. This way we have index links to the files in that directory to help with the navigation and it looks good. But if the user ask for a file/directory that doesn't exist on the server we send back a 404(File not found).We also link to our secret folder in this main index but if we click on it we get an 403 (access denied).

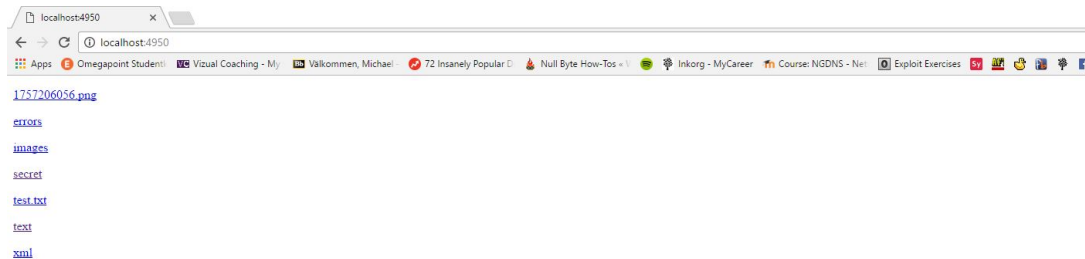


Figure 2.1: GET request on a directory, responds with index.html

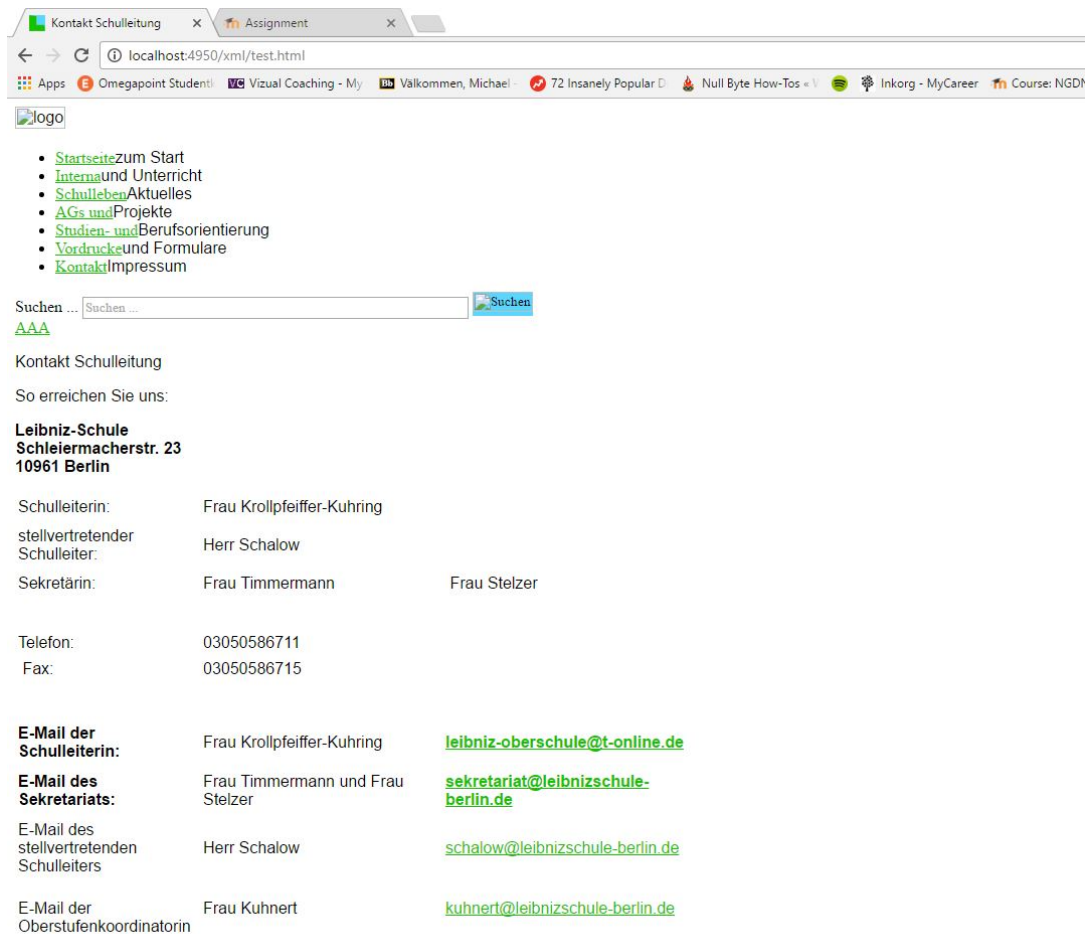


Figure 2.2: GET request on a HTML file, this one is a copy without css of a German site

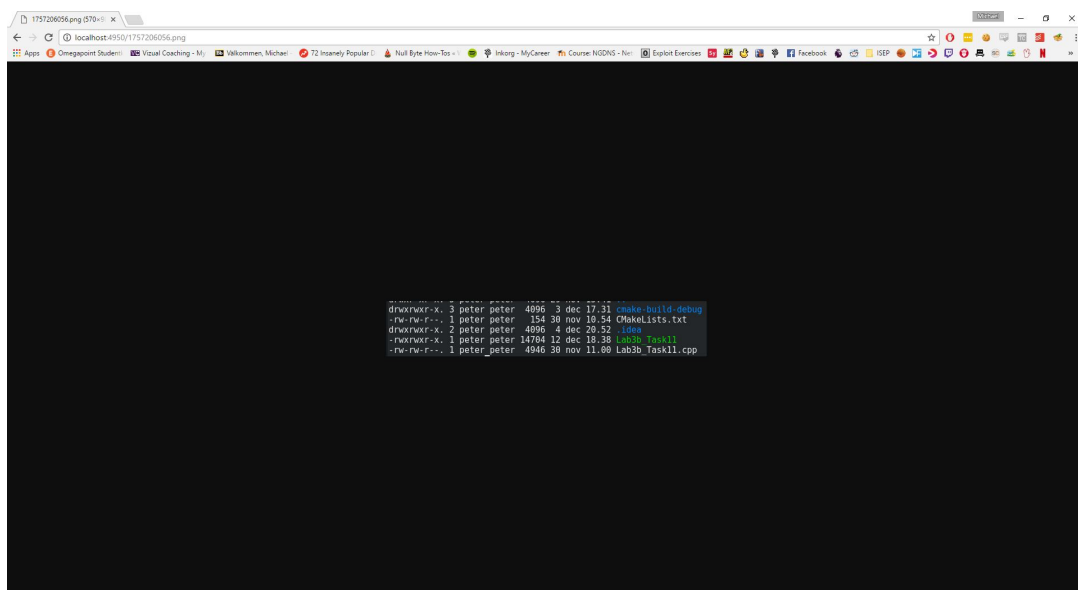
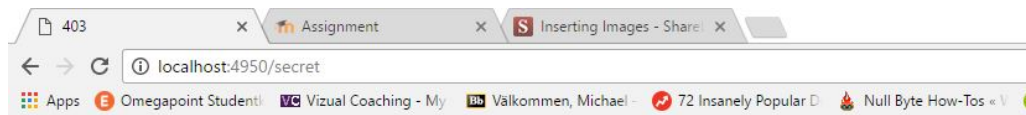


Figure 2.3: GET request on an image

3 Problem two

List of all implemented server responses

- 200: OK
- 201: Created
- 204: No Content
- 302: Found
- 400: Bad request
- 403: Access Denied
- 404: Not Found
- 411: Length Required
- 414: URI Too Long
- 415: Unsupported Media Type
- 500: Internal Server Error
- 501: Not implemented
- 505: HTTP Version Not Supported



403

Access Denied

The requested resource requires an authentication.

Figure 3.1: 403 Access denied when trying to access our secret folder

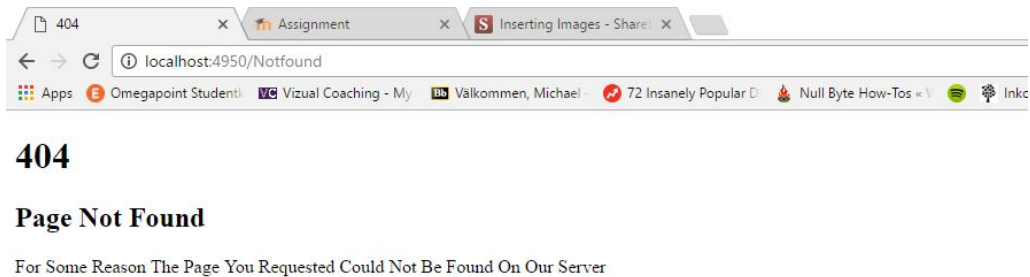


Figure 3.2: 404 Not Found when trying to GET a resource that don't exist on the server

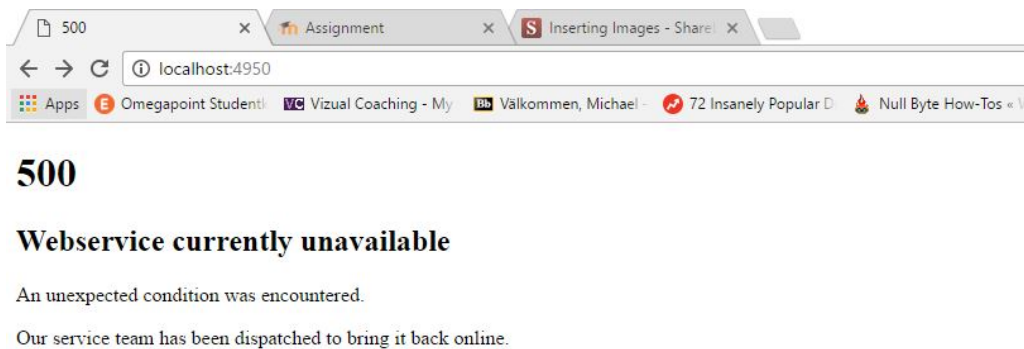


Figure 3.3: 500 for any internal server errors, socket error etc.

3.1 VG.1 HTTP status code implementation

For some images below we just show the error.html response but we send the status code as well for every error.

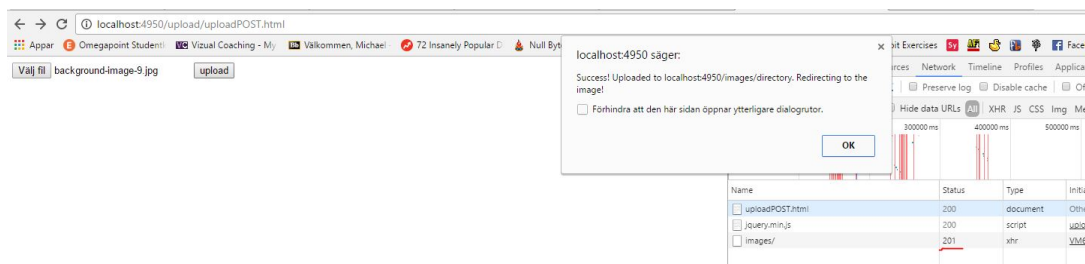


Figure 3.4: 201 Created, when the user creates a new resource with POST/PUT.

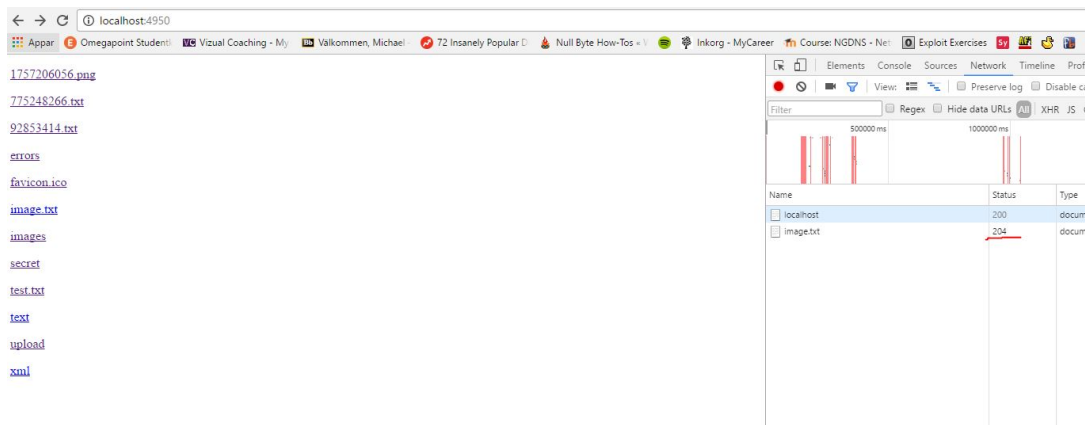


Figure 3.5: 204 Not Found, if the user tries to GET a empty file

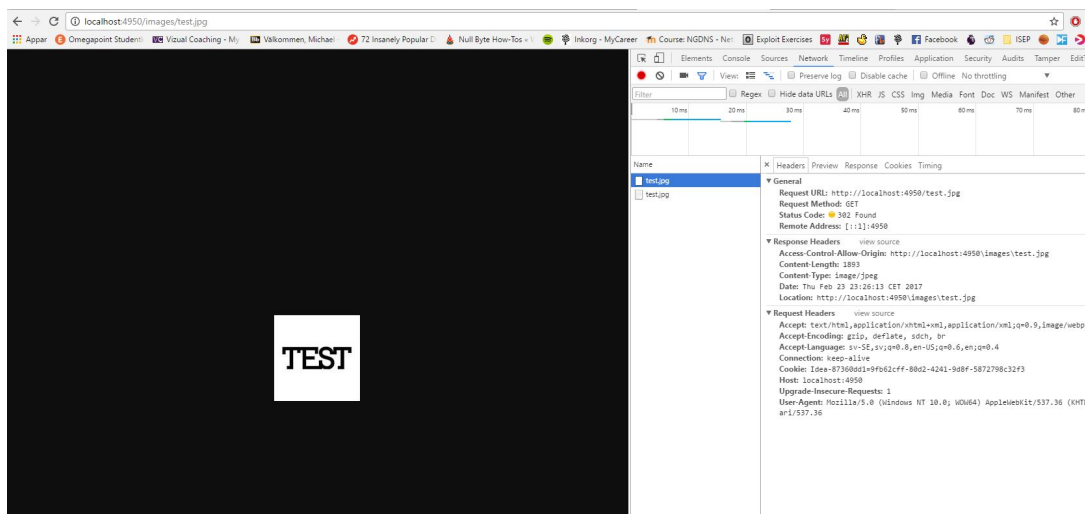


Figure 3.6: 302 if we have specified that a specific file has changed location on the server, redirect the browser to the new location with the location header.

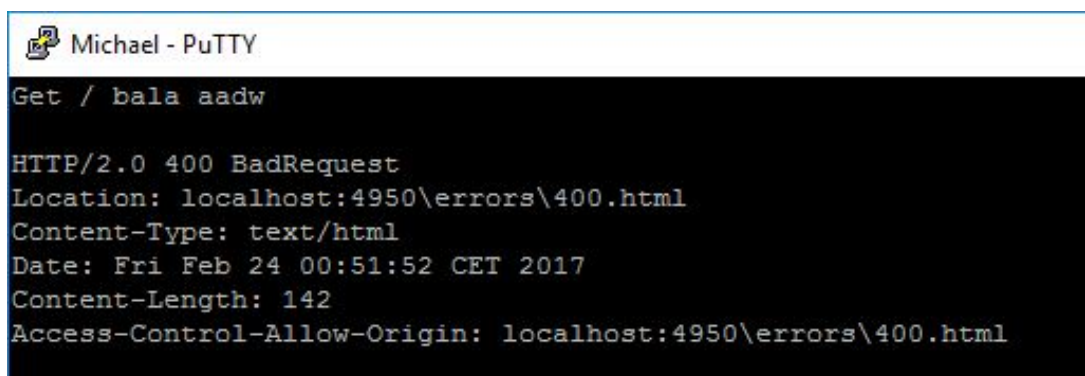


Figure 3.7: 400 if request has malformed syntax

3.2 VG.2 POST vs PUT

We use POST when the user uploads a new resource that our sever handles where it should be placed. In our POST implementation the user can upload an image through our upload page. The server then creates that images with a hash as the file name and under the images folder.

If instead the user has the exact request-URI then we can use PUT to create or over-

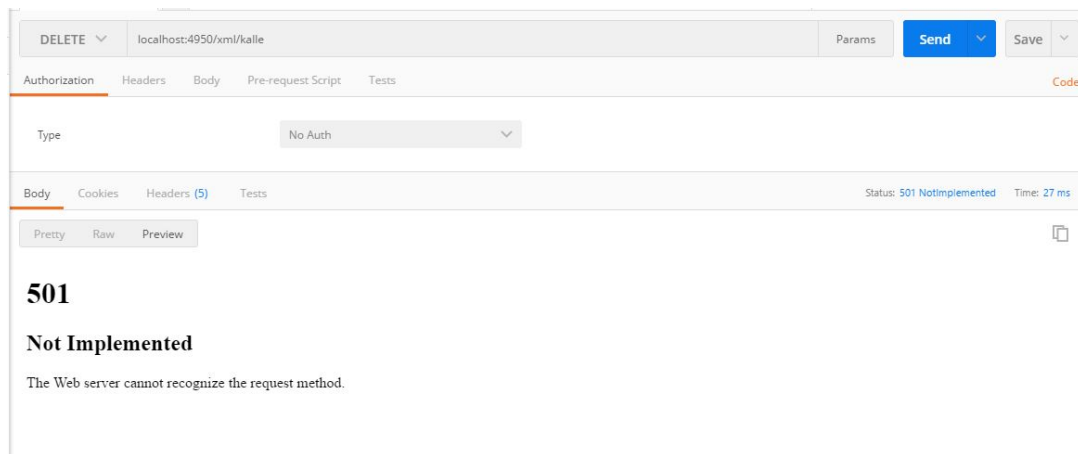


Figure 3.11: 501 for any HTTP methods that we haven't implemented, like in this case the DELETE method.

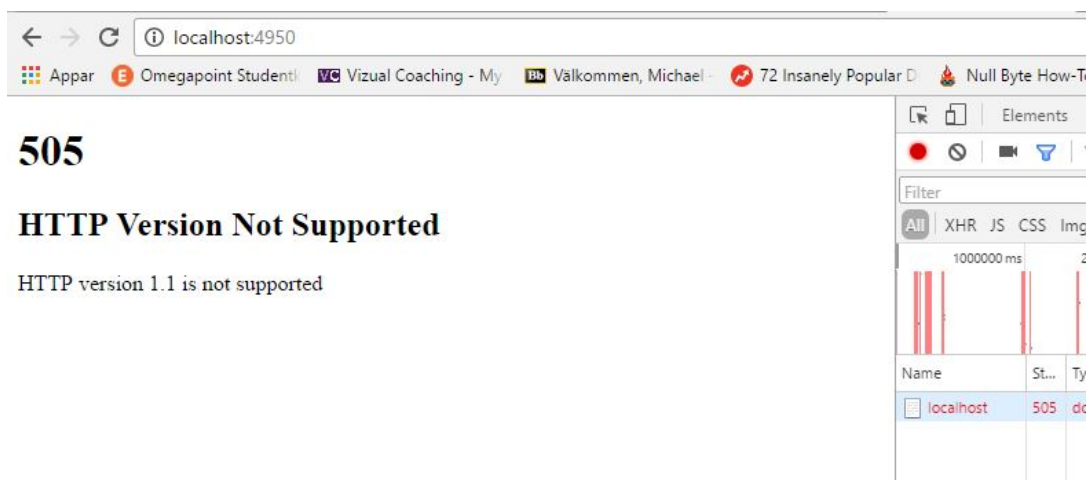


Figure 3.12: 505 for HTTP version 1.1. We support all HTTP versions when we run the server, but we can set this exception if we don't want to support a special HTTP version later on.

4 Problem three

Using Telnet we are creating a raw TCP connection that sends and receive plain text. So as we se in figure 4.1 for our request for a text file we get the text as is. But if we then ask for an image like in figure 4.2 we get the image information as text as well. So the Telnet interprets the image binary as plain text and that's why we get those weird symbols.

```

kaikun@kaikun-ThinkPad-T450s:~$ telnet localhost 4950
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /text/example.txt HTTP/1.1
Host: localhost:4950

HTTP/2.0 200 OK
Access-Control-Allow-Origin: localhost:4950/text/example.txt
Content-Length: 11
Location: localhost:4950/text/example.txt
Content-Type: text/plain
Date: Fri Feb 24 13:15:47 CET 2017

teststresraConnection closed by foreign host.
kaikun@kaikun-ThinkPad-T450s:~$

```

Figure 4.1: Telnet client asking for a .txt file, reads it in plain text

[illegible]

Figure 4.2: Telnet client asking for a .PNG file, also reads it as plain text