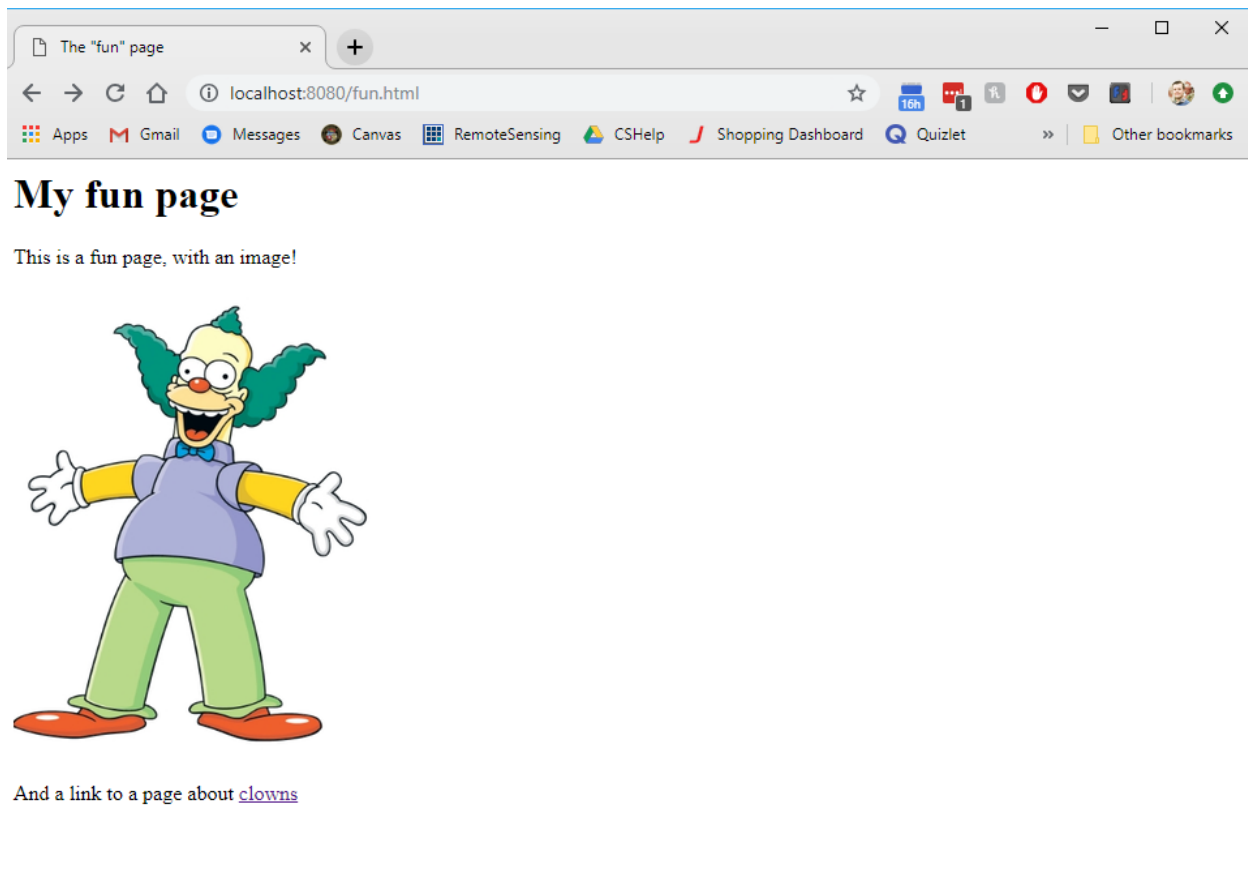


# DV201 (Software Engineering) Assignment 2 (HTTP Server)

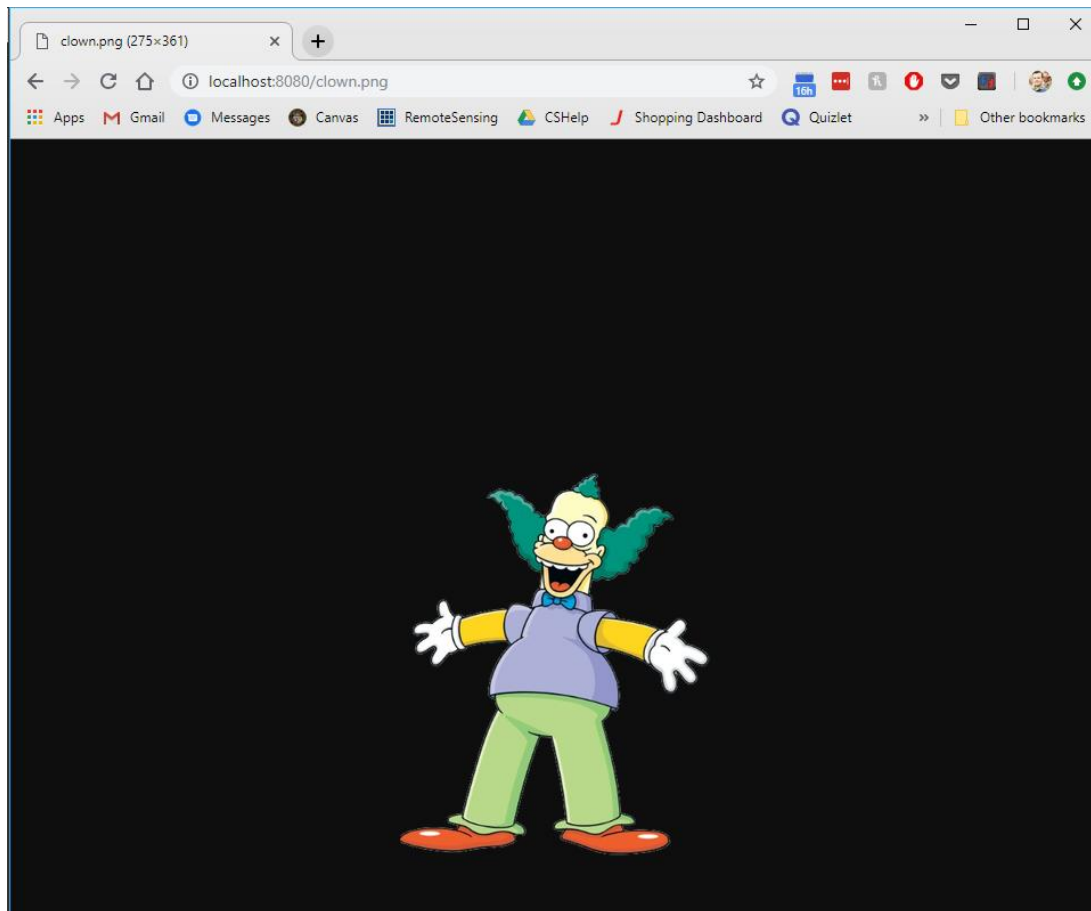
By: Alex and Fabian

## Problem 1:

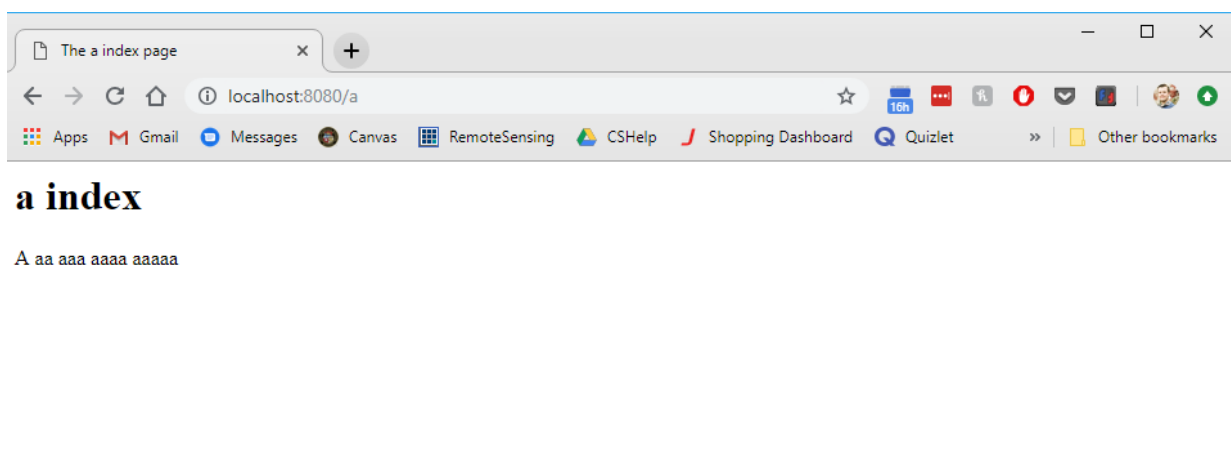
**Named HTML Page:** This test was performed by typing in the URL: "localhost:8080/fun.html" into the browser and taking a screenshot of the response. It retrieves the file fun.html from the root of the server.



**Image:** This test retrieves a PNG image from the server. It was performed by entering the URL: “localhost:8080/clown.png” where clown.png is a PNG image on the root of the server.



**A directory with index.html file:** This test was performed by entering “localhost:8080/a” a browser. It returns the index.html file present in the “a” directory on the server.

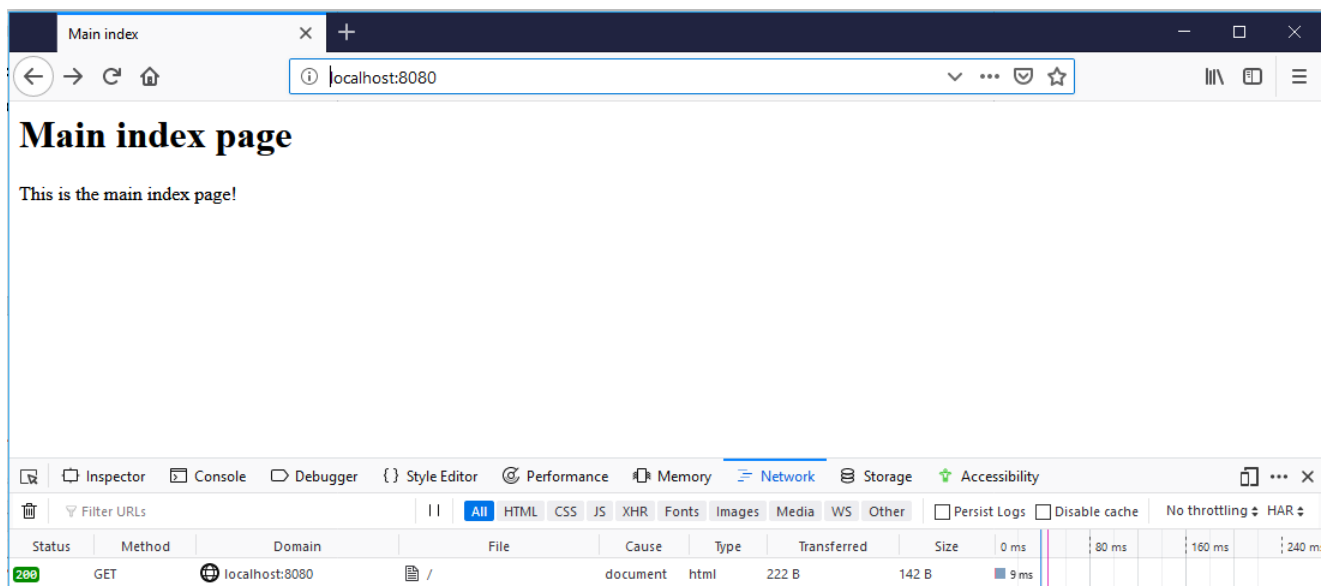


## Problem 2:

Our server has a main class “HTTPServer” that handles starting the server and accepting incoming requests. We check to make sure that the port number to start the server is a valid port number and then each time we receive a new incoming request we create a new thread to handle the inbound request. We chose this because it allows the server to handle multiple simultaneous incoming requests. We chose to use 3 enumerations to represent the various elements needed in generating the HTTP response. This allows for consistent comparisons in switch statements and it allows for a single place for the response strings to be stored.

Once the header is parsed the appropriate handler function for the request type is called. We first check if the requested resource is in the redirect map because if it is and we don't check that first it could generate an error. Then we check if the file is an html, htm, or png file and save the appropriate content type. If the request does not contain a file, we look to see if the specified directory has an index file (either .html or .htm) in it because the server should serve an index file if it can. We verify that the file we are trying to send is readable and that it exists. If not, we save a 404 NOT FOUND response. If the file is found, we check against the forbidden list. If it is forbidden, we save a 403 FORBIDDEN response. After this procedure, we actually send the header and the appropriate body/file.

**200 OK:** For a GET request, the 200 OK response is returned when the requested resource exists and is available (not forbidden, not moved, and readable) to the user. This test was performed by entering: “localhost:8080” into a browser and it returns the index.html page present at the root of the server. As displayed in the browser the index.html page was returned. As displayed in the bottom section of the screenshot the GET request issued to the server returned a 200 OK response.



**302 Found:** For a GET request, the 302 FOUND code indicates that the requested resource is temporarily located under a different URL. On our server we have a Map that contains key value pairs. The key is the requested resource and the value is the new temporary resource location. When the server receives a GET request it checks the map to see if the request resource matches a key in the redirect map. If it does, it returns a 302 Response, that includes the 302 status code and the location field, as shown in the screenshot below when “localhost:8080/alex.html” was requested. Then browsers usually automatically ask for that new location and our server provides it as a usual GET request with 200 OK. In our case this temporary location is “alexTemp.html”. This second request is independent from the first one and our server does not know that they belong together.

The screenshot shows a web browser window with the address bar displaying `localhost:8080/alexTemp.html`. The page content displays the text: **This is a 302 Temp site for the alex.html site**.

Below the browser window, the Chrome DevTools Network tab is open, showing a list of network requests. The first request is a 302 status code, and the second is a 200 status code. The table below summarizes the data from the Network tab:

Status	Method	Domain	File	Cause	Type	Transferred	Size	Time
302	GET	localhost:8080	alex.html	document	html	243 B	196 B	4 ms
200	GET	localhost:8080	alexTemp.html	document	html	276 B	196 B	6 ms

**403 Forbidden:** For a GET request, the 403 FORBIDDEN response code indicates that the server understands the request, but the resource is not allowed to be accessed. Authentication will not help and there is nothing that the client can do to access the resource so they should not reissue the request. Our server contains a list of resources that are forbidden and it includes "localhost:8080/forbidden.html" This test was conducted by requesting "localhost:8080/forbidden.html" from the POSTman API testing tool (Postman). The server returned a 403 FORBIDDEN response as indicated by the Status section of the screenshot and it displayed the 403.html resource from the server to show the client user the resource is forbidden.

The screenshot shows the Postman interface for a GET request to `localhost:8080/forbidden.html`. The status is **403 FORBIDDEN** with a response time of 35 ms and a size of 267 B. The response body is displayed in HTML format, showing a 403 Forbidden page with the title "403 Forbidden" and headers indicating UTF-8 encoding.

GET localhost:8080/forbidden.html

localhost:8080/forbidden.html

GET localhost:8080/forbidden.html

Send Save

Params Authorization Headers (2) Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (2) Test Results Status: 403 FORBIDDEN Time: 35 ms Size: 267 B Download

Pretty Raw Preview

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>403 Forbidden</title>
</head>
<body>
  <h1>403</h1>
  <h2>Forbidden</h2>
</body>
</html>
```

**404 Not Found:** The 404 Not Found response indicates that the requested URI cannot be found on the server. This test was conducted by requesting “unknownpage.html” from the server. Since “unknownpage.html” does not exist on the server it returns a 404 NOT FOUND response as shown in the Status section of the screenshot below and it serves the 404.html resource from the server to show the client user the page cannot be found.

The screenshot shows a web client interface with the following components:

- Top Bar:** A green status bar indicates a **GET** request to `localhost:8080/unknownpage.html`. To the right, there is a dropdown menu set to **No Environment** and icons for eye and settings.
- Address Bar:** Displays `localhost:8080/unknownpage.html`.
- Request Form:** A dropdown menu shows **GET**, followed by a text input containing `localhost:8080/unknownpage.html`. To the right are **Send** (blue button), **Save** (grey button), and a dropdown arrow.
- Navigation Tabs:** A row of tabs includes **Params** (active), **Authorization**, **Headers (2)**, **Body**, **Pre-request Script**, and **Tests**. On the right, there are links for **Cookies**, **Code**, and **Comments (0)**.
- Params Table:** A table with three columns: **KEY**, **VALUE**, and **DESCRIPTION**. It contains one row with **Key** and **Value**. A **Bulk Edit** link is on the right.
- Response Summary:** A bar showing **Status: 404 NOT FOUND**, **Time: 36 ms**, **Size: 277 B**, and a **Download** button.
- Response Body:** A section with tabs for **Pretty** (active), **Raw**, and **Preview**. It displays the following HTML code:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>404 Page Not Found</title>
</head>
<body>
  <h1>404</h1>
  <h2>Page Not Found</h2>
</body>
</html>
```

**500 Internal Error:** The 500 INTERNAL SERVER ERROR response indicates that the server ran into an unexpected condition that prevented it from fulfilling the request. There are many ways to generate a 500 response both expected and unexpected. In our server we surround the whole code for handling the request with a try-catch block and try to send the 500 Code and 500-page if something goes wrong that we cannot or did not handle. In this test we requested an unsupported HTTP request type as shown in the screenshot below. We issued a PATCH request to the server and since our server does not support PATCH requests it returned a 500 INTERNAL SERVER ERROR. The server served the 500.html resource to show the client user that there was an internal server error.

The screenshot shows a web client interface with a PATCH request to localhost:8080/. The status is 500 INTERNAL SERVER ERROR. The response body is HTML showing the error message.

localhost:8080/

PATCH localhost:8080/ Send Save

Params Authorization Headers (2) Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION
Key	Value	Description

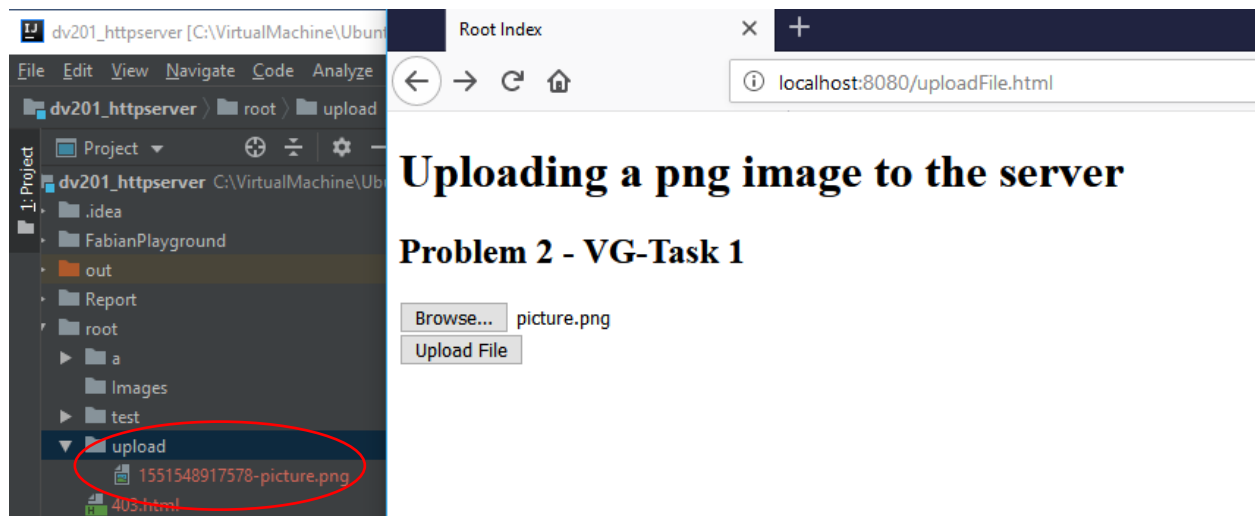
body Cookies Headers (2) Test Results Status: 500 INTERNAL SERVER ERROR Time: 31 ms Size: 301 B Download

Pretty Raw Preview

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>500 Internal Server Error</title>
</head>
<body>
  <h1>500</h1>
  <h2>Internal Server Error</h2>
</body>
</html>
```

## VG Task 1:

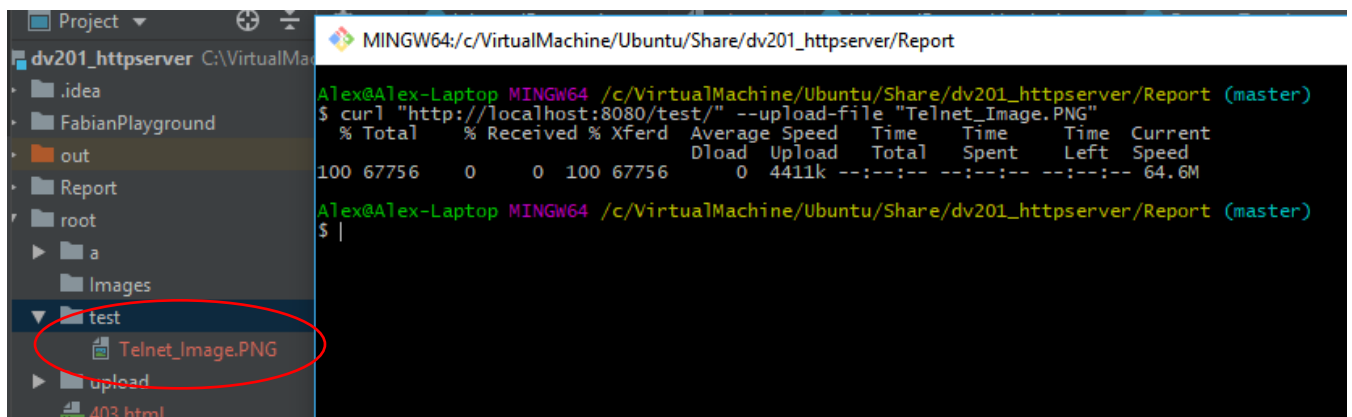
For this task we have an html form that uses the enc-type “multipart/form-data.” This allows us to submit the filename and the binary file data to the server in the body of the POST request. The test was performed by accessing the “uploadFile.html” page and selecting a PNG image from my file system. Then the Upload File button was pressed. When the server receives the POST request it will always create a new resource in the upload folder on the server. Regardless of the filename there will always be a new resource because a date stamp is appended to each file name. The user is only able to upload a file; the server handles the creation of the resource and where it will be placed on the server. This is due to the way POST should work (in difference to PUT). If the request header contains expect continue, we will send a 100 continue response before handling the rest of the request.





## VG Task 2:

For the PUT request the user specifies the desired resource URL in the request. The requested directory must exist on the server since our server does not handle directory creation. If the directory exists and is available, then the resource will be created in the directory with its original filename. If the filename is a directory name and not a file name, we create a index.html file in this directory, because this is the way a GET request to a directory works (if you ask for a folder, you get the index.html inside it) This differs from the POST request since the PUT request specifies the URL where the resource will be created on the server whereas the POST request will create the resource with a URL that the server chooses. Each time a PUT request is issued it creates or replaces a resource in its entirety. This means multiple duplicate requests will always produce the same result unlike POST. CURL also expects a 100 continue response before it will transmit the body of the PUT data. We parsed the header and if the header contains expect continue, we will send a 100 continue response before handling the rest of the request. We also wait to send the 201 or 204 status header until the file has been written to the system to make sure that we don't prematurely send a successful response.



```
Project
└─ dv201_httpserver C:\VirtualMa
   ├── .idea
   ├── FabianPlayground
   ├── out
   ├── Report
   ├── root
   ├── a
   ├── Images
   └── test
       ├── Telnet_Image.PNG
       ├── upload
       └── 403.html

MINGW64:/c/VirtualMachine/Ubuntu/Share/dv201_httpserver/Report
Alex@Alex-Laptop MINGW64 /c/VirtualMachine/Ubuntu/Share/dv201_httpserver/Report (master)
$ curl "http://localhost:8080/test/" --upload-file "Telnet_Image.PNG"
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %                               Dload  Upload   Total   Spent    Left   Speed
100 67756    0     0  100 67756      0  4411k  --:--:-- --:--:-- --:--:-- 64.6M
Alex@Alex-Laptop MINGW64 /c/VirtualMachine/Ubuntu/Share/dv201_httpserver/Report (master)
$ |
```

## PUT vs. POST:

The idea of a POST request is to send data to the server inside the body of the request. The server can do whatever it wants with that data. If the data is a file the server can, for example, save it in its file system. So, a POST request can modify, update, or create a resource but the resource URI is not specified in the request. If you send the same POST request multiple times, the client must assume that the server state is changed every time.

On our server when a user issues a POST request using “uploadFile.html” the filename and the PNG binary image data is sent to the server. This follows the factory design pattern and when the server receives the request, it creates a new resource in the “upload” directory with a date stamp appended to the filename. We chose to append a date stamp because a POST request is not idempotent, and each new request should be handled independent to older ones, so each uploadFile POST request creates a new resource in the upload/ folder

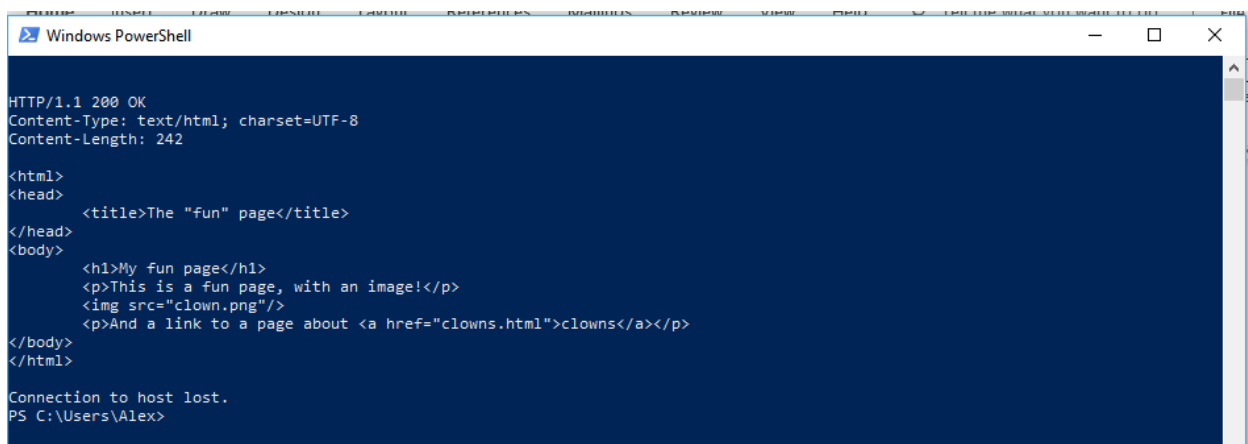
The idea of a PUT request is to save a file to a specific location respectively URL on the server. With a PUT request you specify the resource URI in the request which makes it idempotent. This means that regardless of how many times you execute the request and regardless of whether it previously

existed the result from the request will be the same. PUT updates or creates resources by replacing them in their entirety.

**Problem 3 (Telnet):** The following tests were all performed using telnet through windows PowerShell. Each session was initiated by typing “telnet localhost 8080”. Telnet enables you to connect with a server and send and receive any kind of text-based messages. That is why we can issue HTTP requests from the command line instead of from a web browser.

**Named HTML Page:** This test retrieved the fun.html page from the server. It shows the 200 OK HTTP response from the server indicating that the request was successful, it shows the HTTP version used, which is 1.1. The reply header also shows the Content-Type of the file in the body for letting the client know how he should handle it. The header also included the Content-length which can be checked against the body as a simple data-loss check or used for different ways. Finally, after the blank line, which indicates the end of the header, it shows the desired HTML page. The HTML page is shown in plain text because telnet is a terminal and does not render the page like a browser. After the request is issued and the page is returned the server closes the connection.

**Command used:** GET /fun.html HTTP/1.1



```
Windows PowerShell

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 242

<html>
<head>
  <title>The "fun" page</title>
</head>
<body>
  <h1>My fun page</h1>
  <p>This is a fun page, with an image!</p>
  
  <p>And a link to a page about <a href="clowns.html">clowns</a></p>
</body>
</html>

Connection to host lost.
PS C:\Users\Alex>
```

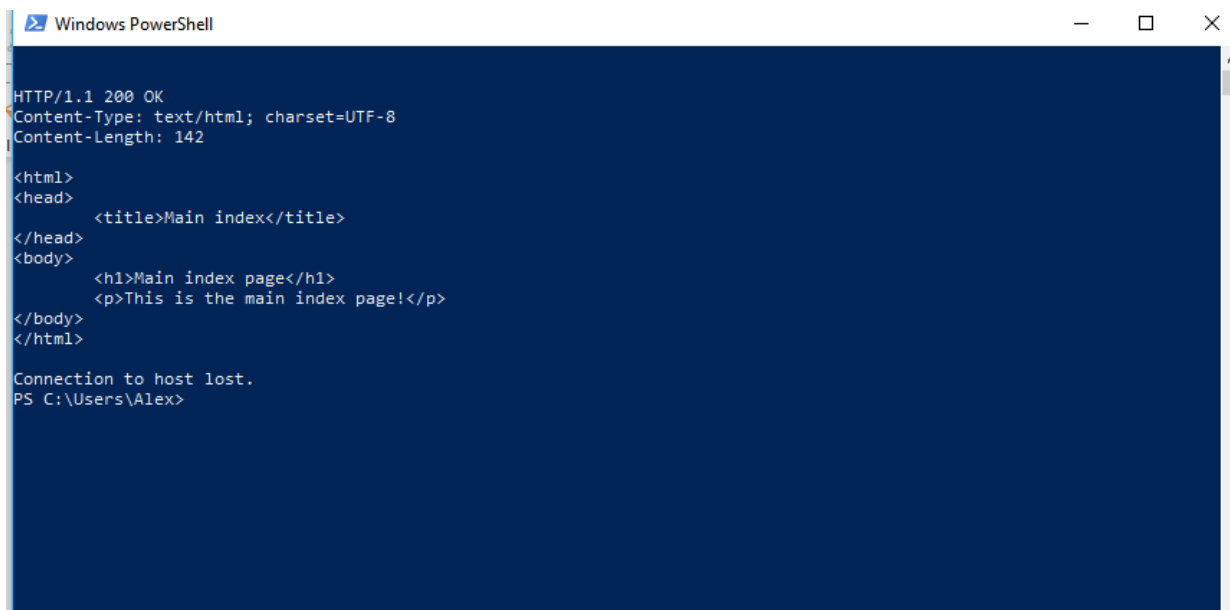
Finally, we see the desired PNG image. The PNG image is binary data and since this terminal is not able to interpret PNG images, it tries to convert it to characters which explains the strange streams of seemingly random characters. After sending the response, the server closes the connection.

**Command used:** GET /clown.png HTTP/1.1

[illegible]

**Directory:** This test retrieved the index.html file from the root of the server, because “/” means the root directory. It shows the 200 OK HTTP response from the server indicating that the request was successful and the HTTP version that the server used. The Content-Type is text/html because the file in the body is a text file that should be interpreted as a html file. The Content-Length is 142, which means that the body of the response, so the html file, is 142 byte long. After the header, there is a blank line which indicates the end of the header which means that after that blank line the body starts which finally shows the desired HTML file.

**Command used:** GET / HTTP/1.1

A screenshot of a Windows PowerShell window with a dark blue background. The window title is "Windows PowerShell". The output shows an HTTP response: "HTTP/1.1 200 OK", "Content-Type: text/html; charset=UTF-8", and "Content-Length: 142". This is followed by a blank line and then the HTML content: "<html>", "<head>", " <title>Main index</title>", "</head>", "<body>", " <h1>Main index page</h1>", " <p>This is the main index page!</p>", "</body>", and "</html>". Below the HTML content, it says "Connection to host lost." and "PS C:\Users\Alex>".

```
Windows PowerShell
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 142

<html>
<head>
    <title>Main index</title>
</head>
<body>
    <h1>Main index page</h1>
    <p>This is the main index page!</p>
</body>
</html>

Connection to host lost.
PS C:\Users\Alex>
```

**Summary of Project:**

Fabian (50%) Alex (50%):