**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.

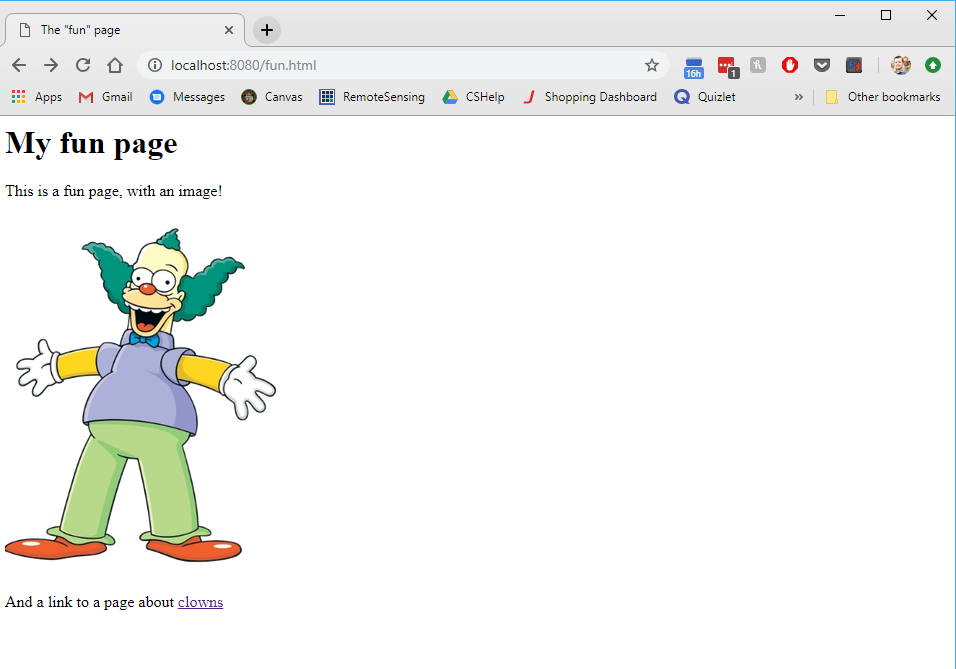
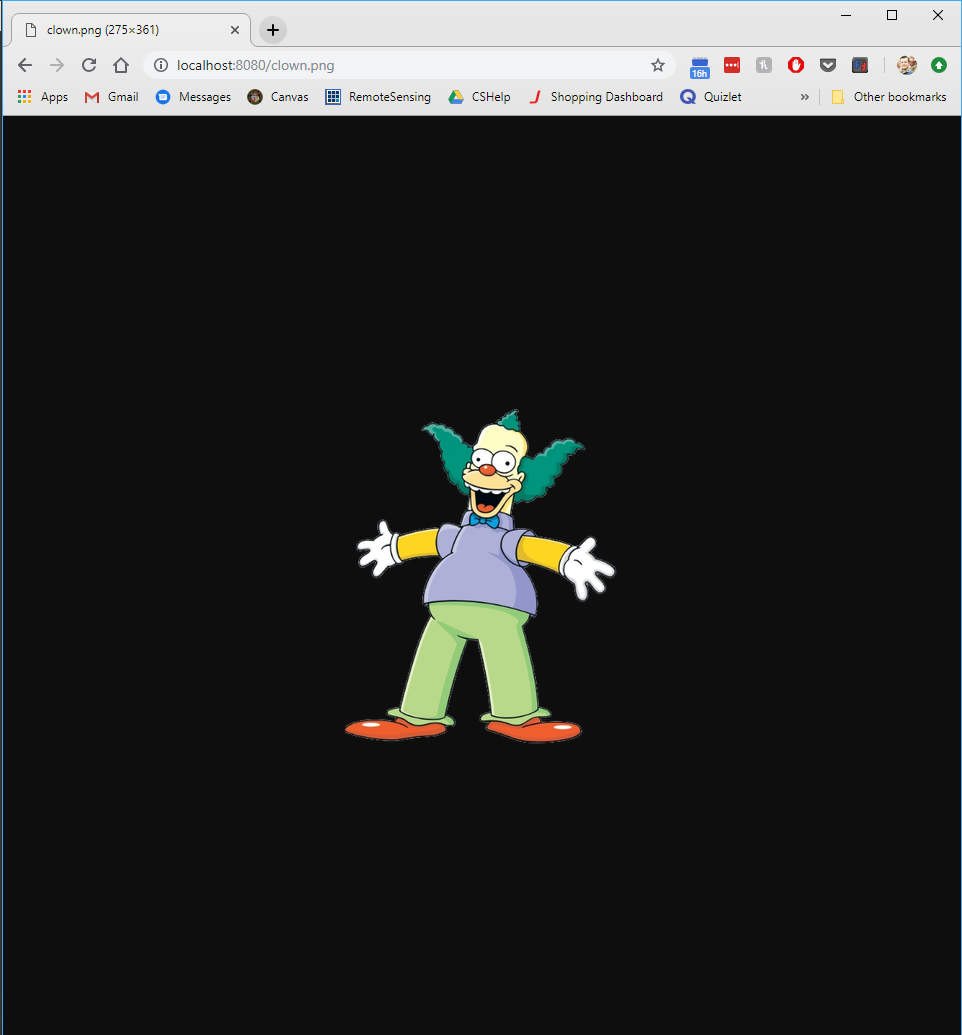
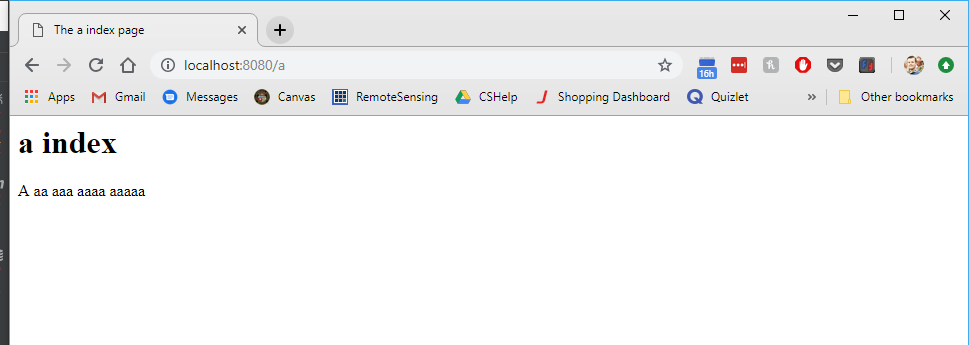


Image: This test retrieves a PNG image from the server. It was performed by entering the URL: “localhost:8080/clown.png” where clown.png is a PNG image on the root of the server.

****

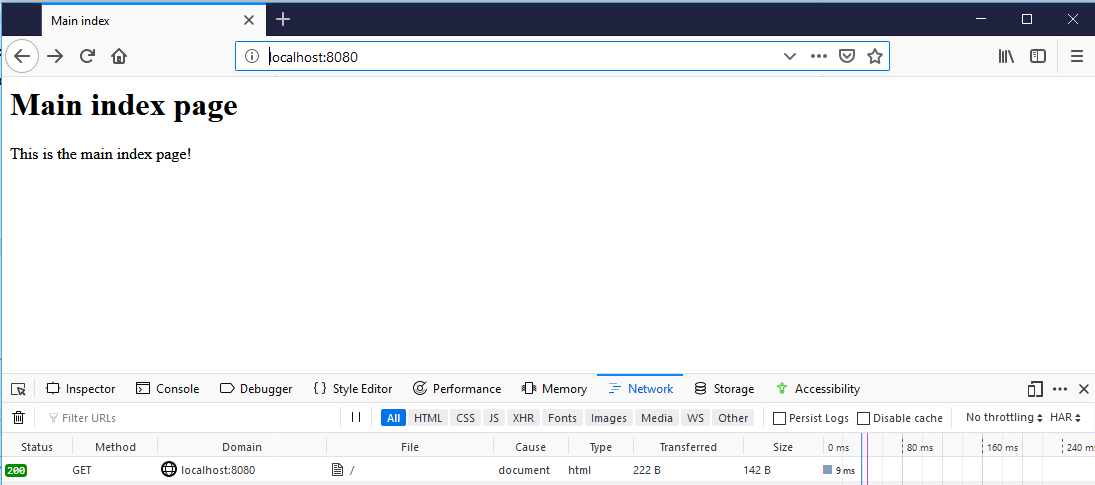
A directory with index.html file: This test was preformed by entering “localhost:8080/a” a brower. It returns the index.html file present in the a directory on the server.



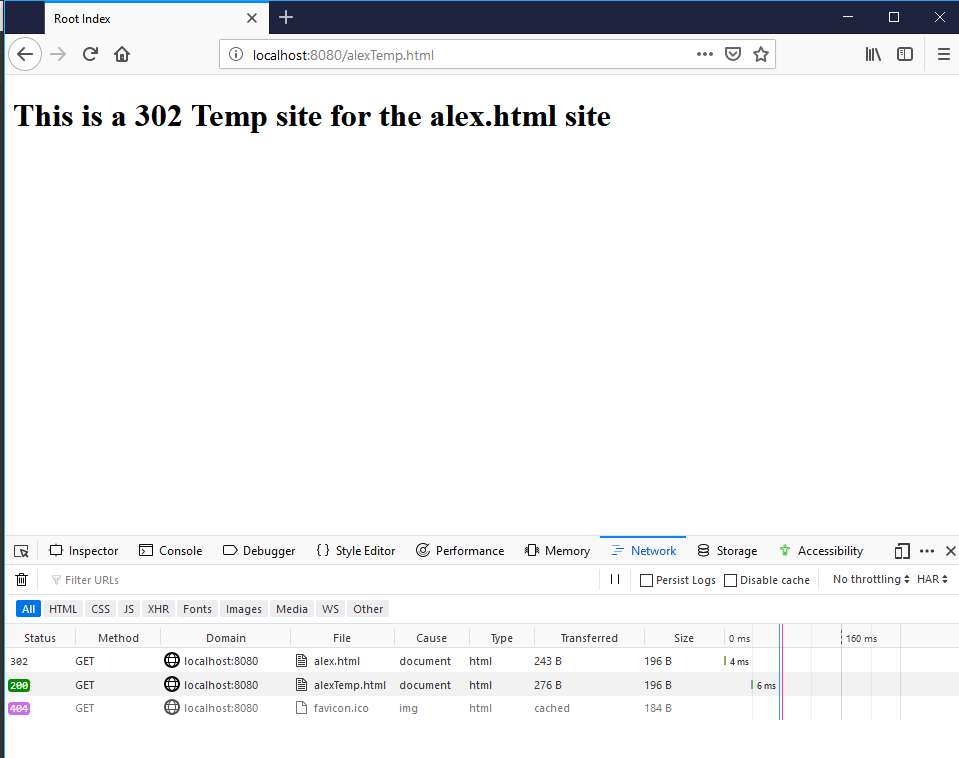
**Problem 2:** Our server has a main class “HTTPServer” that handles starting the server and accepting incoming requests. We check to make sure that the port number to start the server is a valid port number and then each time we receive a new incoming request we create a new thread to handle the inbound 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 send the appropriate content type back. If the request does not contain a file, we look to see if the specified directory has an index file 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 and if we cannot find it, we return a 404 response because the file is not there. If the file is found, we check against the forbidden list and if it is forbidden, we return a 403 Forbidden response.

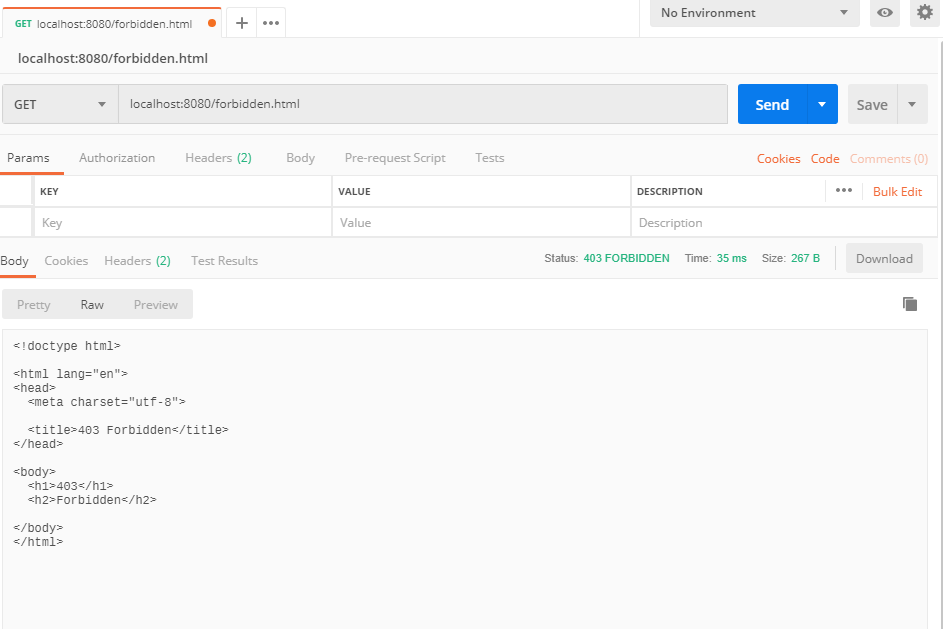
**200 OK:** For a GET request, the 200 OK response is returned when the requested resource exists and is available (not forbidden or moved) to the user. This test was performed by entering: “localhost:8080” into a browser and it returns the index.html page present at the root of the server. As displayed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



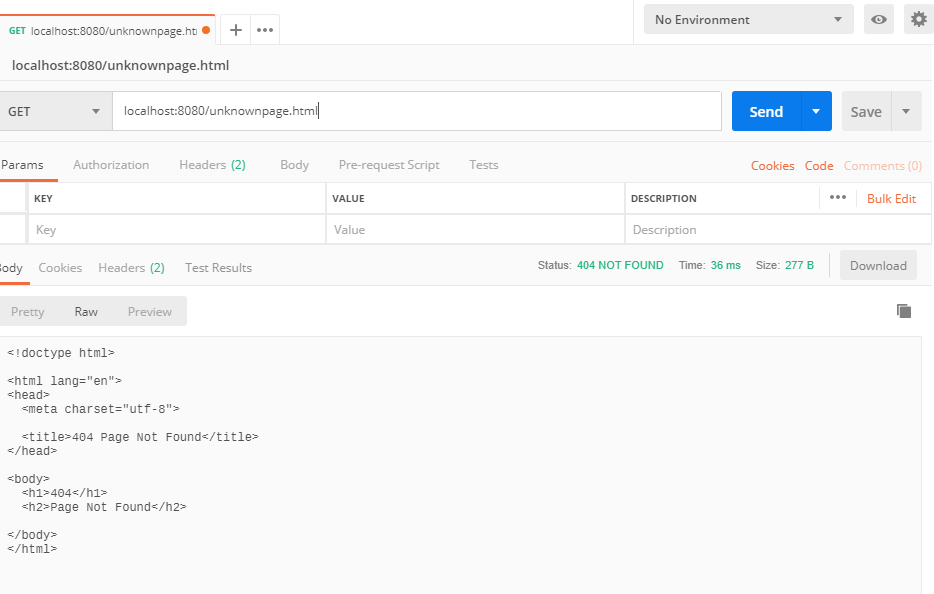
**302 Found:** For a GET request, the 302 Found code indicates that the requested resource is temporarily located under a different URI. On our server we have a Map that contains key value pairs. The key is the requested resource URI and the value is the new temporary resource URI. When the server receives 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 as shown in the screenshot below when “localhost:8080/alex.html” was requested. Then the server provides the new temporary resource (in this case alexTemp.html) and returns a 200 OK status afterwards to indicate that the resource is available and ok.



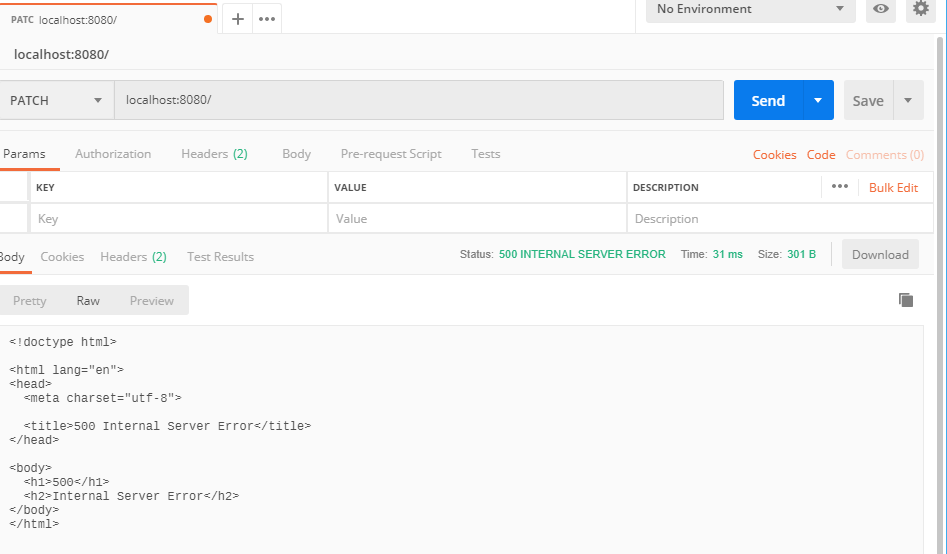
**403 Forbidden:** For a GET request, the 403 Forbidden response code indicates that the server understands the request, but the resource is not allowed to be accessed. Authentication will not help and there is nothing that the client can do to access the resource so they should not reissue the request. 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 the resource is forbidden.



**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 the page cannot be found.

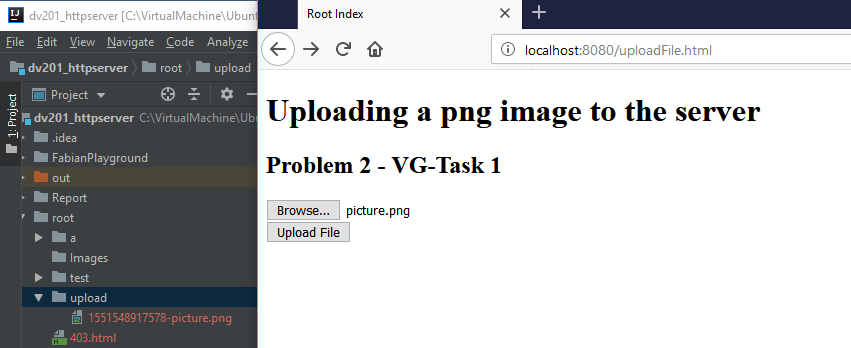


**500 Internal Error:** The 500 internal server error response indicates that the server ran into an unexpected condition that prevented it from fulfilling the request. There are many ways to generate a 500 response both expected and unexpected. In this test we requested an unsupported HTTP request type 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 that there was an internal server error.



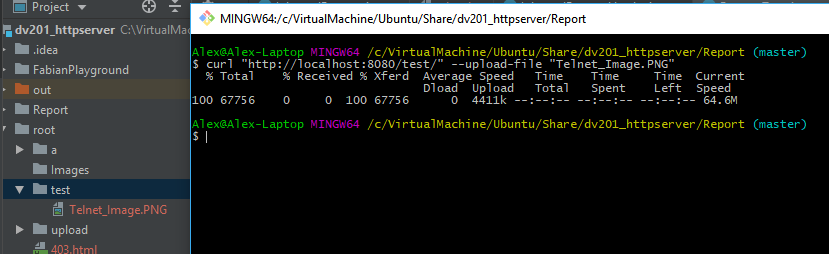
**VG Task 1:**

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.



**VG Task 2:**

For the PUT request the user specifies the desired resource URI 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. This differs from the POST request since the request specifies the URI where the resource will be created on the server whereas the POST request will create the resource with a URI 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 POST 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 status header until the file has been written to the system to make sure that we don’t prematurely send a successful response.



**PUT vs. POST:**

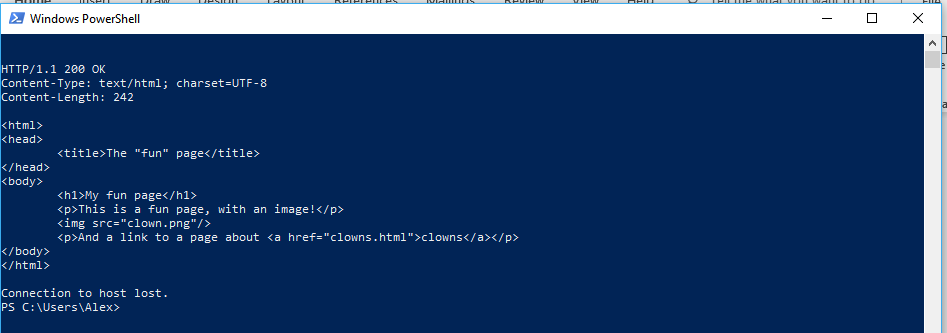
A POST request can modify, update, or create a resource but the resource URI is not specified in the request. A POST request can update or create a child resource at a predefined URL on the server. For example, if you generate a POST request to /items then a new resource will be created on the server that lives under /items for example /items/1. If you send the request multiple times, multiple resources will be generated. 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 create a new resource.

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 form the request will be the same. PUT updates or creates resources by replacing them in their entirety. On our server the location for the resource is specified in the PUT request. When the request is issued the new resource will be created at the URI specified in the request.

**Problem 3 (Telnet):** The following tests were all preformed using telnet through windows PowerShell. Each session was initiated by typing “telnet localhost 8080”. Telnet enables you to issue HTTP requests from the command line instead of from a web browser.

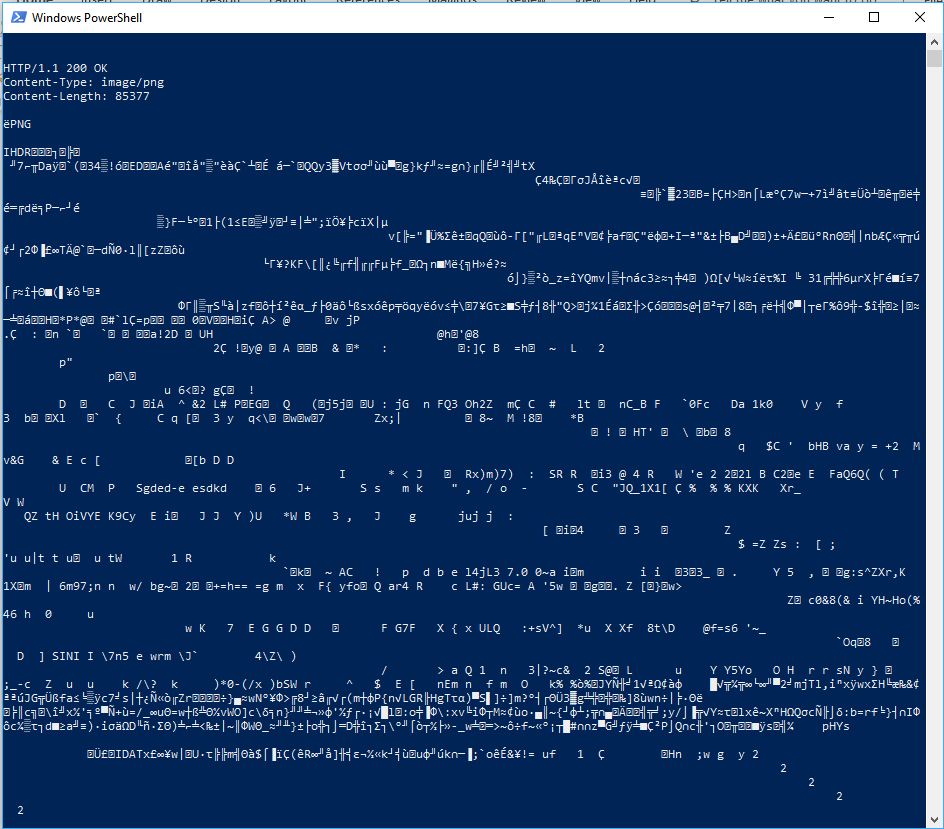
**Named HTML Page:** This test retrieved the fun.html page from the server. It shows the 200 OK HTTP response from the server indicating that the request was successful, and it shows the desired HTML page. The replay header shows the content-length which can be checked against the sent header as a simple data-loss check. The HTML page is shown in plain text because telnet is not 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 session.

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



**Image:** This test retrieved the clown.png image from the server. It shows the 200 OK HTTP response from the server indicating that the request was successful, and it shows the desired PNG image. The PNG image is a binary data stream 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.

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



**Directory:** This test retrieved the index.html file from the root of the server. It shows the 200 OK HTTP response from the server indicating that the request was successful, and it shows the desired HTML page.

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

