CSCI 4131 – Spring 2023
Internet Programming Assignment 4 (Version 1)

Due Date: March 24th at 11:59pm
Late Submission Deadline: March 27th at 11:59pm

1. Description

The objective of this assignment is for you to learn the Hyper-Text Transfer Protocol (HTTP) and build a small subset of an HTTP server in Python 3 (If possible, update to the newest version of Python 3). In this assignment, using Python 3 and TCP sockets, you will program some of the basic functionality of an HTTP (Web) server. You are required to use Python 3. You may not use the Python server.py for this assignment. You will need to go through RFC 2616 for HTTP 1.1 protocol details. This assignment specification is 8 pages long.

When a web client (such as Google Chrome) connects to a web server (such as https://www.google.com) the data that is exchanged between them (e.g., HTTP messages, HTML, CSS, JavaScript, pictures, audio, video, etc.) is transmitted using the Hyper-Text Transfer Protocol.

An outline of how a basic HTTP server works for an HTTP request message is specified below.

- An HTTP client connects to the HTTP-server and sends an HTTP request message requesting a resource to the HTTP server.
- The request can be of type HEAD, GET, POST, etc.
- The server parses the request header fields.
- For a GET request, the server identifies the requested resource (e.g., an HTML file) and checks if the resource exists and if it can access it. If this is the case, the server proceeds to step 5 below. Otherwise it proceeds to step 6 below.
- The HTTP server then generates an appropriate HTTP response message. If the requested resource is found and is accessible, the HTTP server reads in the resource and builds a response message. The response includes successful (2xx) status code in the response headers along with other metadata such as the Content Type and Content Length, and includes the resource data as the message body. The server then sends the message to the HTTP client (e.g., a browser, such as Chrome) which sends the GET request.
- Otherwise, if the resource requested by the HTML client is not found, then the HTTP Server composes and sends a response message with an error response status code to the HTTP client (e.g., a browser). See section 4.3 below for a discussion about the errors your server must recognize and respond to (i.e., compose a proper response message and send to the HTTP client).

2. Required Functionality

Download additional files

You will need to download the following files for this assignment:

- 403.html - this file should be sent to the client if permissions do not permit its access (Provided).
- 404.html - this file should be sent to client if the server cannot find the requested file (Provided)
- private.html - this file is the private file that triggers 403 forbidden code, you can use it to test – ***but you have to do it on a computer running Unix or Linux so you can set the permissions!***
- Coffman.html – html file containing an image to use for testing your server's capability to respond to a request for an image (Provided).
- OuttaSpace.html – html file containing an audio controls element to use for testing your server's capability to respond to a request for an audio file (Provided).
- Coffman_N_OuttaSpace.html– contains both an image and audio control element to use for testing your server's capability to respond to a request for an image and an audio file (Provided).
- coffman.png – the image file (a .png file) used by Coffman..html and Coffman_N_OuttaSpace.html (Provided).
- OuttaSpace.mp3 – the audio file (a .mp3 file) used by OuttaSpace.html and Coffman_N_OuttaSpace.html (Provided).

We have provided files listed above in the file named: Hw4Resources.zip

Additionally, you will need all of the files from your previous homeworks. Include all CSS you've written, any scripts, and your HTML pages from Homework 3. You'll then need to restructure your project structure as required below.

We have provided the executable Python 3 files EchoClient.py and EchoServer.py which you are free to use and refactor in order to construct your server. If you do so, your final server file should be named `myServer.py`.

Finally, we provide a skeleton of the python code in myServer.py. This gives a starting point for your server and describes much of the requirements within comments. Do not rely on the skeleton to provide prescriptive requirements.

Starting the server

When you start your server, it will establish a socket and bind to a port, listening for connections. You can send a request to the server to get your Schedule from your web browser by typing:

http://<host>:<port>/MySchedule.html

For this assignment, when developing and testing your server, you will run the server on your local machine, so <host> will be localhost and <port> should default to 9001.

Also, you are required to use python 3 to develop and test your server – use of frameworks that generate Python, like Flask, are not permitted. Use of any Python module that provides an HTTP server is not permitted.

When you run your server on Unix and Linux Operating Systems your server should bind to port 9001 and serve requests. Note, if you are developing and testing on Windows OS, and you run into errors, bind to another port (for example, 8050). Regardless, ensure it binds to port 9001 upon submission. An example call to start your server on the CSELabs computers is: python3 myServer.py

Updated file structure and file references

Instead of any number of possible arbitrary directory layouts, we'll be asking you to standardize your assets. This change should be pretty simple but will require you to make updates to your anchor links, scripts, and image paths in your HTML files.

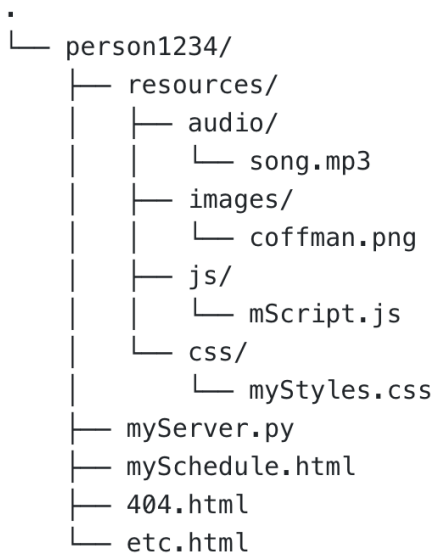

Figure 1: An example of the requested directory structure

Specifically, you should have a directory named `resources` which has up to 5 different subdirectories. The following subdirectories are allowed: `audio`, `css`, `fonts`, `images`, and `js`. Each of these subdirectories can only contain field that match its category. For example, CSS files shouldn't be found in your `images` directory. In the root directory, you should include your `myServer.py` file, as well as any and all HTML files.

With this new structure, you should update your HTML to properly point to the new location of each image, stylesheet, script, mp3 file, or font (if you're using a custom font). Once you get your server implemented, you'll need to update these links again to make new GET requests for each resource.

For example, instead of having a link to a local stylesheet "./mystyles.css", you should instead try to get the stylesheet from "http://localhost:9001/resources/mystyles.css". When a user

navigates to your page that's served by the web server, any additional resources should also be requested from the web server.

GET Request implementation

GET requests are the most commonly used HTTP requests. For example, if you enter the following address in your browser's address bar:

http://localhost:9001/MySchedule.html

the browser will issue a GET request to the server to fetch MySchedule.html file from the directory in which the server code resides. Below is an example get request received by your server:

```
GET /MySchedule.html HTTP/1.1
Host: localhost:9001
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:33.0) Gecko/20100101 Firefox/33.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Since GET requests are common and easy to test (just open your browser to http://localhost:9001/MySchedule.html), it's worthwhile to start from implementing GET requests first before moving to POST or HEAD requests.

A high level overview of how to implement GET requests using the `MyServer.py` template code is provided below:

This part requires a lot of reading the existing code template provided. Open up `MyServer.py` and take a look at the code. One of the first things you'll notice is that there's a helper class named `Request` which handles and holds the data a client will send the web server. It's not extremely important to fully understand the class or the `recv_until_crlfs` function, but you should look at the documentation for the Request class as you'll be using it.

You should however jump forward to line 118 which has some information about the binary_type_files variable. You should finish defining this set with any other file extensions that your application has that need to be returned as binary instead of text. Right afterwards, you should finish defining the `mime_types` dict with any file extension to MIME type mapping. For information about MIME types, check out the MDN documentation on them: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

From there, you'll want to figure out how to return information to the client. We've provided a helper class named `ResponseBuilder`. This class holds information necessary to create a response, and has some of its functionality already defined. The `build` function is not implemented. If you want to use the ResponseBuilder, you need to implement the build function which generates an HTTP Response. You don't have to use the ResponseBuilder, but it will reduce the amount of work required for this assignment. Implementing the `build` function (or its equivalent) is one of the most important parts of this assignment. Make sure you're properly

returning a valid HTTP response. See https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html for more information on how an HTTP response should be structured.

Once you have the `build` function implemented, you can take a look at the `method_not_allowed` function for an example of how the ResponseBuilder is used to return content. If you want to

Providing Redirect Support

The provided HTML page `MyServer.html` should be updated to have a new form that sends a GET request to `localhost:9001/redirect`. This redirection should be handled by the server.

When a user submits the form, the form's content should be used to create a redirect response. This redirect response should send the user to a YouTube search results page for their query. When testing on a browser this should lead to the browser automatically going to the YouTube results.

For extra credit, you can allow the user to specify from a selection of possible websites which one they want to search. The client can then choose if they want to search Google, or search for YouTube videos.

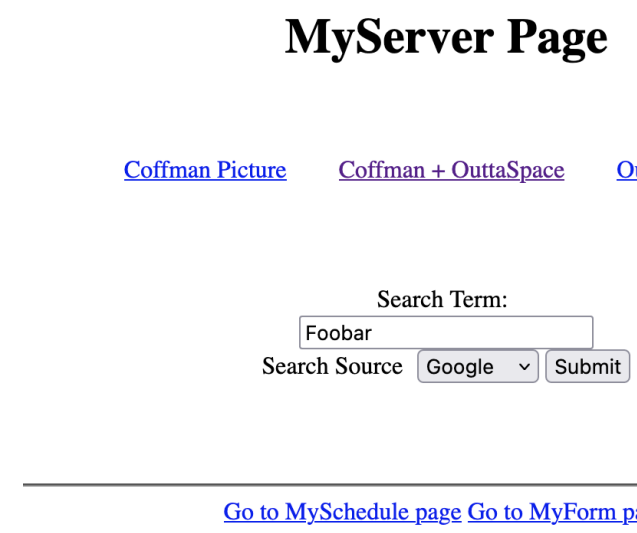

Figure 2: An example of MyServer.html updated to include a search form that then redirects the user to their intended search results

Your server should handle these requests and return a response with HTTP 307 (Temporary Redirect) status code. You can read more about this status code in section 10.3.8 of RFC 2616.

This response will have an empty body. However, in the response headers you should specify a 'Location' header with the value equal to a YouTube URL. On receiving the response, the browser reads this Location header and navigates to the given YouTube URL.

Note – doing this will involve parsing the query-string part of the URL to extract what the user searched for. You are expected to handle searches with spaces and other "special characters".

The function (provided via import at the top of the file) unquote plus can be used to parse the URL-encoding used by GET requests.

HEAD Requests

Your server should support head requests. The head request type is expected to return all the same headers as a GET request type for the same URL, without returning a body. This can be done very easily if you use good software engineering principles. You could, for example, run the same internal code for a GET response, but modify the response so it doesn't have a file attached.

In order to test a HEAD request, try doing a GET request to a resource and make note of the headers received back. Then do a HEAD request and see if the response headers and status match.

404 Responses

If a file doesn't exist, a GET request should return a 404 Response (with the 404.html file's content as the body of the response).

To do this, implement the `resource_not_found` function at the bottom of `MyServer.py`. Once this function is properly implemented, have it be called by the `get_request` function when handling a file that doesn't exist.

Figure 3: An example of a 404 NOT FOUND response

Implement Permissions

You are required to test your solution on a Linux or Unix machine such as the Department of Computer Science Computers running the Ubuntu OS to ensure you test your solution on files with permissions set.

To give the files above proper permissions, execute following chmod commands to correctly set permissions on your files after downloading them to your folder:

```
chmod 640 private.html

chmod 644 403.html

chmod 644 404.html
```

```
chmod 644 MyContacts.html

chmod 644 MyWidgets.html

chmod 644 MyServer.html

chmod 644 Coffman.html

chmod 644 OuttaSpace.html

chmod 644 Coffman_N_OuttaSpace.html

chmod 644 coffman.png

chmod 644 OuttaSpace.mp3

chmod 644 MyForm.html
```

If the web server does not have the permission to access requested resources (e.g., private.html), your server should create a response message with a 403 error response code and 403.html (provided) and send it to the requesting client.

POST requests

You will use the form that you developed for first homework. In the first three homework assignments, your form simply submitted the data to our server which returned an HTML for the browser to display. For this homework, you will instead submit your form to your python HTTP server. You achieve this by first changing the method of the form to "post" and action to http://localhost:9001/MyForm.html

Upon successful submission of the form, your server should return an HTML page with all the information submitted on the form. So your server must construct and return an HTML document with the data values that was submitted to your server via the POST request message sent by the client. One way to do this is to use Python to build a string with HTML tags and the data embedded in it (concatenate HTML strings & the data values to form an HTML document).

Recall that your form's data will be sent form encoded. To decode this data, we've included the unquote plus function in the provided starter code. Take care in decoding this data to extract key value pairs, as prematurely decoding may make extracting them impossible.

It's possible that there are special characters in the request body. If so, your implementation may crash if it doesn't utilize unquote properly. If so, that's okay, but if you want to be able to handle arbitrary special characters in either the key or value of a given entry in your POST request, that will get you some extra credit for this homework.

Grading Criteria

- **a.** All files should be organized in the directory structure requested above **5 points**
- **b.** All HTML files and scripts should be updated to make GET requests for images, styling, scripts etc. A web page with 4 images should make 5 total GET requests (ignoring styling / scripts) to the web server. **5 points**
- **c.** The web server properly starts on port 9001 **5 points**
- **d.** GET requests work correctly if a file exists. **30 points**
    a. Images can be requested properly
    b. JavaScript, CSS, and HTML all can be requested and returned properly
    c. HTTP status codes and response headers are all correct
    d. MP3 files are properly returned
- **e.** 404 Responses are sent if a requested file doesn't exist and 405 Responses are sent if an invalid request arrives (this should already be implemented for you) **5 points**
- **f.** HEAD requests are properly implemented for both resources that do and don't exist **5 points**
    a. If a file is locked behind authorization, the HEAD request's response should reflect that
- **g.** Permission related requests are handled properly with a 403 response if the file isn't properly accessible **5 points**
- **h.** A redirect request properly routes the user to YouTube **15 points**
    a. If the user can choose from 2+ sites, and is dynamically redirected to the right site, **+10 Extra Credit**
- **i.** POST requests properly echo back user input from a form **25 points**
    a. If the POST request properly handles arbitrary special characters **+5 Extra Credit**

SUBMISSION INSTRUCTIONS:

Your submission should be packaged in a tar file or zip file. When opened, it must create a directory named: '<Your UMN x.500 ID>' containing all of your files (HTML, CSS, Pictures, and any additional external JavaScript files. Submit your homework via the class Canvas item named Homework 4 Submission Link in the week X module on the class Canvas site.