BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS3320 - Computer Networks

Course Project: HTTP Server

Stduent's Name: Mazen Amria
Stduent's ID: 1180897
Instructor: Dr. Abdalkarim Awwad
April 30, 2021

1 Abstract

The project contains a simple HTTP web server that covers the basic functionality and details of the HTTP standard [2]. It supports the most common content types and responds with the minimal required response header fields.

Contents

2 Server Modules

2.1 Core Module

The Algorithm starts by binding the address of the server and the port '9000' to the socket, then starts listening for requests at that socket, whenever it receives an request it starts handling it by trying to match the path of the request with the path of the handlers (handlers are listed by the priority):

```
# IPv4
server_socket = socket(AF_INET, SOCK_STREAM)

# Allow socket reuse
server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# Bind the address to the socket
server_socket.bind((SRVHOST, SRVPORT))

# Listening with maximum connections = QSIZ
server_socket.listen(QSIZ)

while True:
    client_socket, client_address = server_socket.accept()

    # Read the request
    req = recv_request(client_socket)

    # Extract the method, path, HTTP version and Request Heders
    method, path, version, headers = parse_headers(req)

    for handler in HANDLERS:
        # if the request matches the handler
        if handler[0].match(path) and handler[1] == method:
            # Try to handle it
            if handler[2](client_socket, client_address, path, version, headers):
                break

    # Close the connection
    client_socket.close()
```

2.2 Reading and Parsing Requests

Two functions were used for reading and parsing the request. First methods basically reads the bytes from the socket, with maximum size of 8192 bytes (since the server only handles GET requests, and the maximum length of GET request varies according to the browser used, but Mozilla Firefox has the maximum length with 8192 bytes [1]).

```
MAX_REQ = 8192


def recv_request(conn):
    # recieve the request as string
    req = conn.recv(MAX_REQ).decode('utf-8')

    # split it
    req = req.split('\r\n')

    return req
```

The second function is to parse the method of the request (which only will be 'GET' for this project), the path of the request, the HTTP version used by the client and the Request Headers.

```
def parse_headers(req):
    method, path, version = req[0].split()

    headers = {}
    for i in range(1, len(req)):
        if req[i] == '':
            break
        key, value = req[i].split(': ', 1)
        headers[key] = value

    return method, path, version, headers
```

2.3 Handlers

Handlers are used as a list of tuples, where the first element in the tuple is the regex to be matched with the request path, and the next element is the function to be executed.

```
# define list of handlers
HANDLERS = [
    (re.compile(r'^/$'), 'GET', root_handler),
    (re.compile(r'^/Sort'), 'GET', smartphones_handler),
    (re.compile(r''), 'GET', default_handler)
]
```

The handler has the following signature, and the functionality of the handler is to send the appropriate response.

```
def handler(conn, addr, path, version, req_headers):
    """Handler Blueprint
    conn: Client's Socket (socket)
    addr: Client's Address (tuple)
    path: Request Path (str)
    version: Client's HTTP version (str)
    req_headers: Request Headers (map)

    Returns: True on success, False of failure
    """
```

2.3.1 Root Handler

Used to handle request at the root path '/' by sending 'index.html' as a response.

```
def root_handler(conn, addr, path, version, req_headers):
    try:
        file = open('index.html', 'rb')

        # Set the Headers
        status_code = 200
        mod = datetime.fromtimestamp(os.path.getmtime(path), TIMEZONE)
        mod = mod.strftime(DATEFMT)
        res_headers = {
            'Content-Type': 'text/html',
            'Last-Modified': mod,
            'Connection': req_headers['Connection']
        }

        # Set the Data
        data = file.read()

        # Send the response
        send_res(conn, status_code, res_headers, data)
        return True
    except:
        return False
```

2.3.2 Smartphones Handler

Used to handle request at the root path '/SortName' and '/SortPrice' by sending the sorted data from 'smartphones.csv' as 'json' format. It can be extended to more than '/SortName' and '/SortPrice', as it sorts the data according to '⟨col⟩' when received request at '/Sort⟨col⟩'

```
def smartphones_handler(conn, addr, path, version, req_headers):
    try:
        smartphones = pandas.read_csv('smartphones.csv')

        # Sort depending on the desired column
        col = path[len("/Sort"):]
        res = smartphones.sort_values(col)

        # Set Headers
        status_code = 200
        mod = datetime.fromtimestamp(
            os.path.getmtime('smartphones.csv'), TIMEZONE)
        mod = mod.strftime(DATEFMT)
```

```
        res_headers = {
            'Content-Type': 'application/json',
            'Last-Modified': mod,
            'Connection': req_headers['Connection']
        }

        # Set Data
        data = res.to_json(orient='records').encode('utf-8')

        # Send the response
        send_res(conn, status_code, res_headers, data)
        return True
    except:
        return False
```

2.3.3 Default Handler

Used to handle all requests that haven't been handled by the other handlers, it's set to be the lowest priority handler, it basically tries to search for a specified file in the server path '/⟨filename⟩.⟨ext⟩' if it found the file it will send it with the appropriate 'Content-Type' according to the '⟨ext⟩', if not, it will send 'notfound.html' after rendering the IPs and the ports, with '404' as the status code.

```
def default_handler(conn, addr, path, version, req_headers):
    try:
        # remove the starting '/'
        path = path[1:]
        file = open(path, 'rb')

        # Set the Headers
        status_code = 200
        mod = datetime.fromtimestamp(os.path.getmtime(path), TIMEZONE)
        mod = mod.strftime(DATEFMT)
        res_headers = {
            'Last-Modified': mod,
            'Connection': req_headers['Connection']
        }

        for tp in TYPES:
            if tp[0].match(path):
                res_headers['Content-Type'] = tp[1]
                break

        # Set the Data
        data = file.read()

        # Send the response
        send_res(conn, status_code, res_headers, data)
    except:
        # Requested Object Not Found
        file = open('notfound.html', 'r')

        # Set Headers
        status_code = 404
        mod = datetime.now(TIMEZONE)
        mod = mod.strftime(DATEFMT)
        res_headers = {
            'Content-Type': 'text/html',
            'Last-Modified': mod,
            'Connection': req_headers['Connection']
        }

        # Set the Data
        data = file.read()

        # Render the addresses
        data = data.replace('{{ client-ip }}', str(addr[0]))
        data = data.replace('{{ client-port }}', str(addr[1]))
        data = data.replace('{{ server-ip }}', str(conn.getsockname()[0]))
        data = data.replace('{{ server-port }}', str(conn.getsockname()[1]))

        # Encode Data
        data = data.encode('utf-8')
```

```
51
52          # Send the response
53          send_res(conn, status_code, res_headers, data)
54      finally:
55          return True
```

2.4 Sending Response

In this module a function is used to send the response with the appropriate headers. The function sets the 'Date', the 'Content-Length', the 'Server' headers, the status code and the HTTP version used by the server.

```
1  def send_res(conn, status_code, headers, data):
2      # Set the Content-Length header
3      headers['Content-Length'] = len(data)
4
5      # Set the Date header
6      now = datetime.now(TIMEZONE)
7      now = now.strftime(DATEFMT)
8      headers['Date'] = now
9
10     # Set the Server header
11     headers['Server'] = SERVER
12
13     # Send the response status
14     conn.send(
15         f'{VERSION} {status_code} {STATUS[status_code]}\r\n'.encode('utf-8'))
16
17     # Send the headers
18     for (key, value) in headers.items():
19         conn.send(f'{key}: {value}\r\n'.encode('utf-8'))
20
21     # End of headers section
22     conn.send(b'\r\n')
23
24     # Send the payload
25     conn.send(data)
```

3 Server Configuration

The server can be easily configured by changing the following constants in the first lines of the code.

```
1  # Server Interfaces and Port
2  SRVHOST = "0.0.0.0"
3  SRVPORT = 9000
4
5  # Maximum requests in the queue
6  QSIZ = 10
7
8  # Maximum request length
9  MAX_REQ = 8192
10
11 # HTTP response status codes
12 STATUS = {
13     200: 'OK',
14     404: 'Not Found'
15 }
16
17 # HTTP version used by the server
18 VERSION = 'HTTP/1.1'
19
20 # Supported files by the server
21 TYPES = [
22     (re.compile(r''), 'text/plain'),
23     (re.compile(r'.html$'), 'text/html'),
24     (re.compile(r'.js$'), 'application/javascript'),
25     (re.compile(r'.json$'), 'application/json'),
26     (re.compile(r'.png$'), 'image/png'),
27     (re.compile(r'.jpg$'), 'image/jpg'),
28     (re.compile(r'.ico$'), 'image/x-icon'),
```

```
    (re.compile(r'.csv$'), 'text/csv')
]

# Time-zone used by the server
TIMEZONE = timezone('GMT')

# Date foramt used by the server
DATEFMT = '%a, %d %b %Y %H:%M:%S %Z'

# Information about the Server Machine
SERVER = ' '.join(os.uname())
```

4 Procedure: Part I

4.1 Ping a device on LAN

At the local area network, the number of hops between both devices is 1, as it's only the modem that both devices are connected to, starting by the ping command, we can see that the output shows the ttl value of each packet, which is the maximum allowed hops that the packet can travel through, here it's set to 64 means that it can travels through 64 packets only, also we can see the time that the packet took to travel, at the end of the ping command we see some statistics of the network status. We can see after that, specifying the ttl to be 1 and the packets are still travelling since as mentioned the number of hops is only 1.

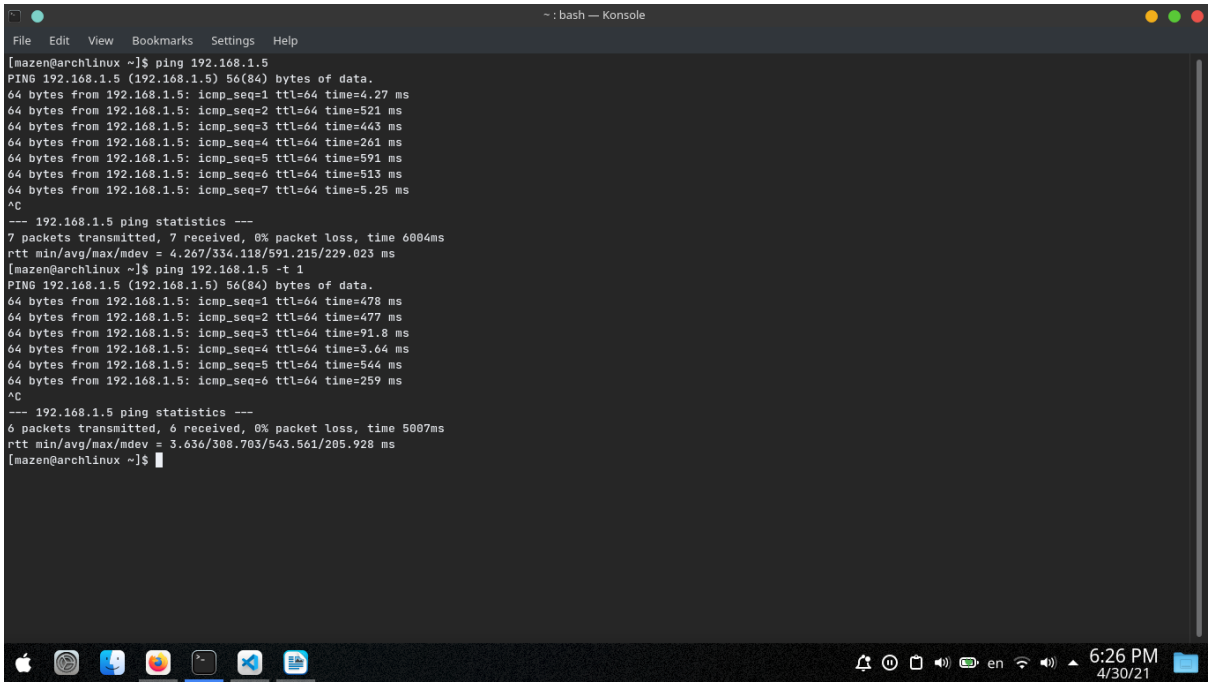

Figure 1: Ping LAN

4.2 Ping stanford.edu

For a remote server, ping will take more hops and ttl will be estimated with a larger number by the ping in order to ensure reaching the destination, we can see that it has been estimated with 243. Next time we specified the ttl to be 10 by using the option '-t' and we can see that the packet didn't reach the destination as the Time to live has been exceeded, it reaches a server called 'SUNet' and then stops. We raised the ttl to be 15 after that, and it seems to work, so we can approximate the number of hops between our device and the server to be between 11 and 15 hops.

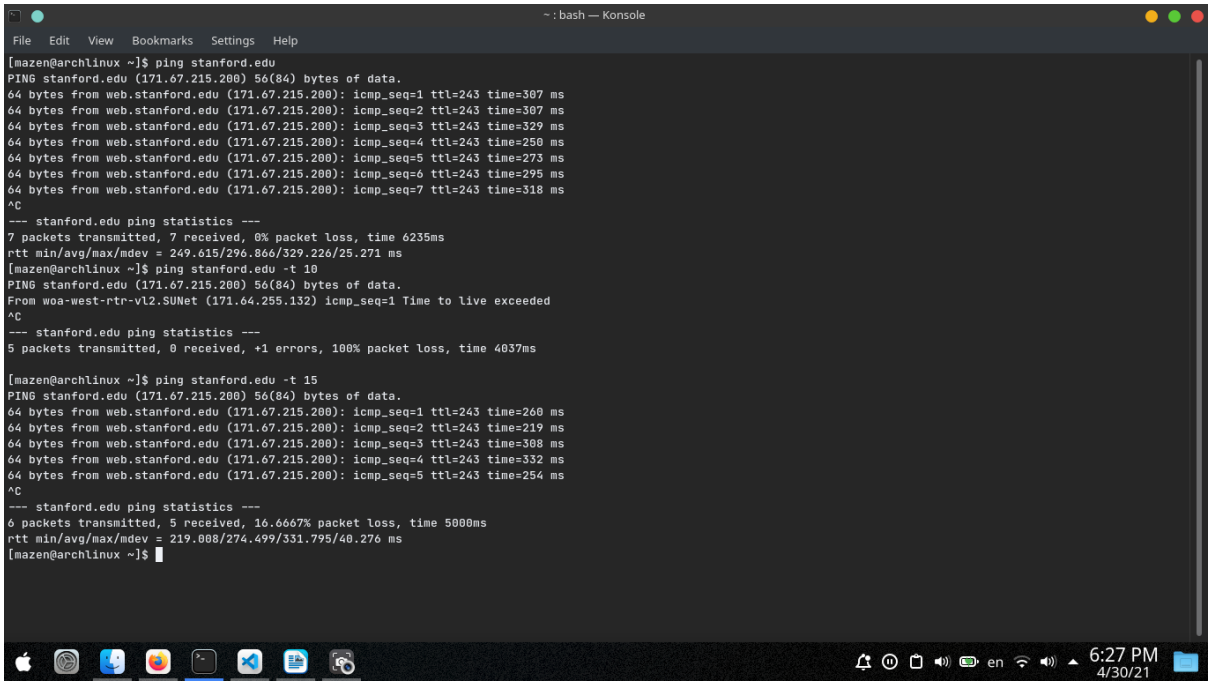

Figure 2: Ping stanford.edu

4.3 Traceroute stanford.edu

In this part we tried to trace the route between the host and stanford.edu server. The result is shown in the following figure, apparently the server didn't respond correctly so we exceeded the maximum number of hops and the traceroute process terminated, we tried to increase the ttl using '-m' option, but still get the same results.

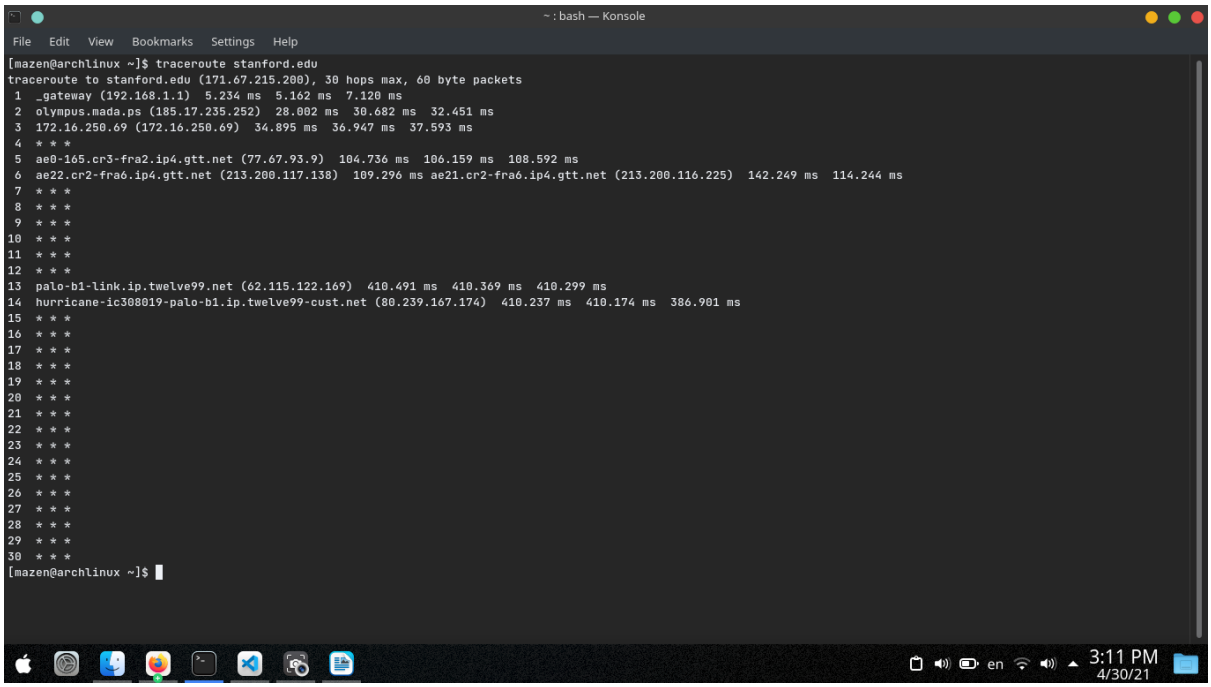

Figure 3: Traceroute stanford.edu

In order to complete this procedure we used google.com as a destination, as we see from the next figure, the distance to google is 10 hops as we see from the traceroute, by excluding the gateway of the host we can specify the ttl of the ping to be 9, as shown in the figure, ping with ttl = 9 is respoding

from the last hop shown in the traceroute ‘zrh04s06’.

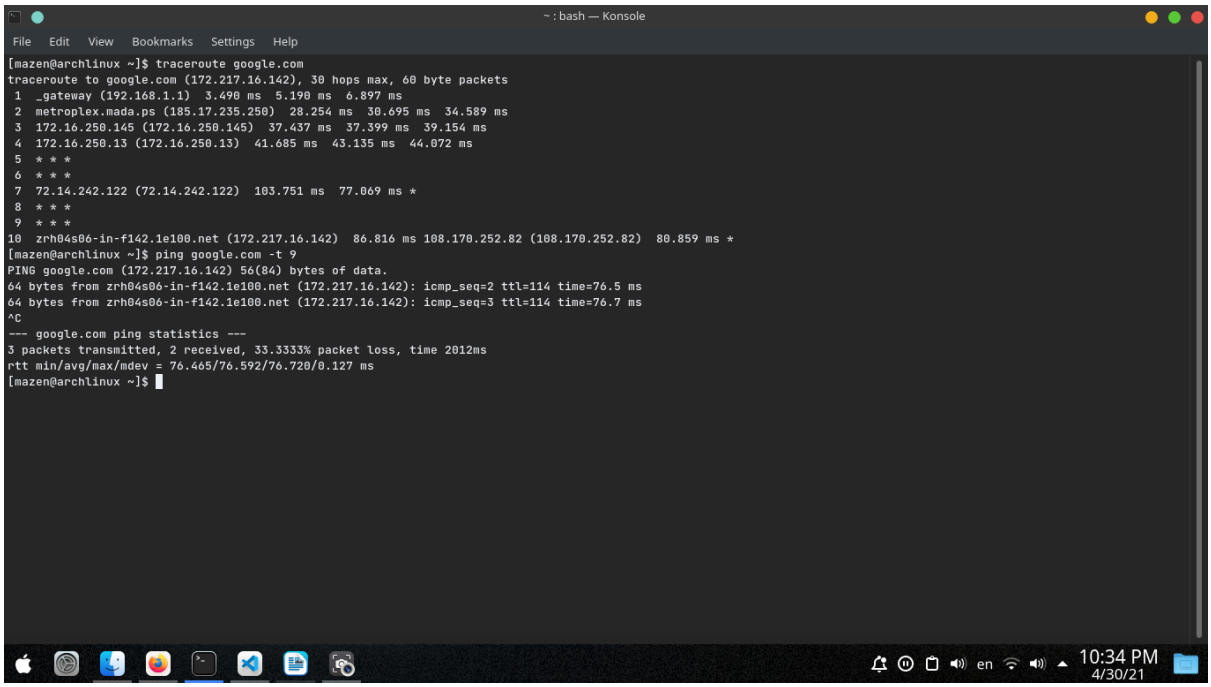

Figure 4: Traceroute google.com

4.4 NSLookUp stanford.edu

In this part we are querying fromt the name server (looking up) for the IP of the canonical name ‘stanford.edu’, as shown in figure the IP is ‘171.67.215.200’ which is exactly the IP we saw when we pinged the same server.

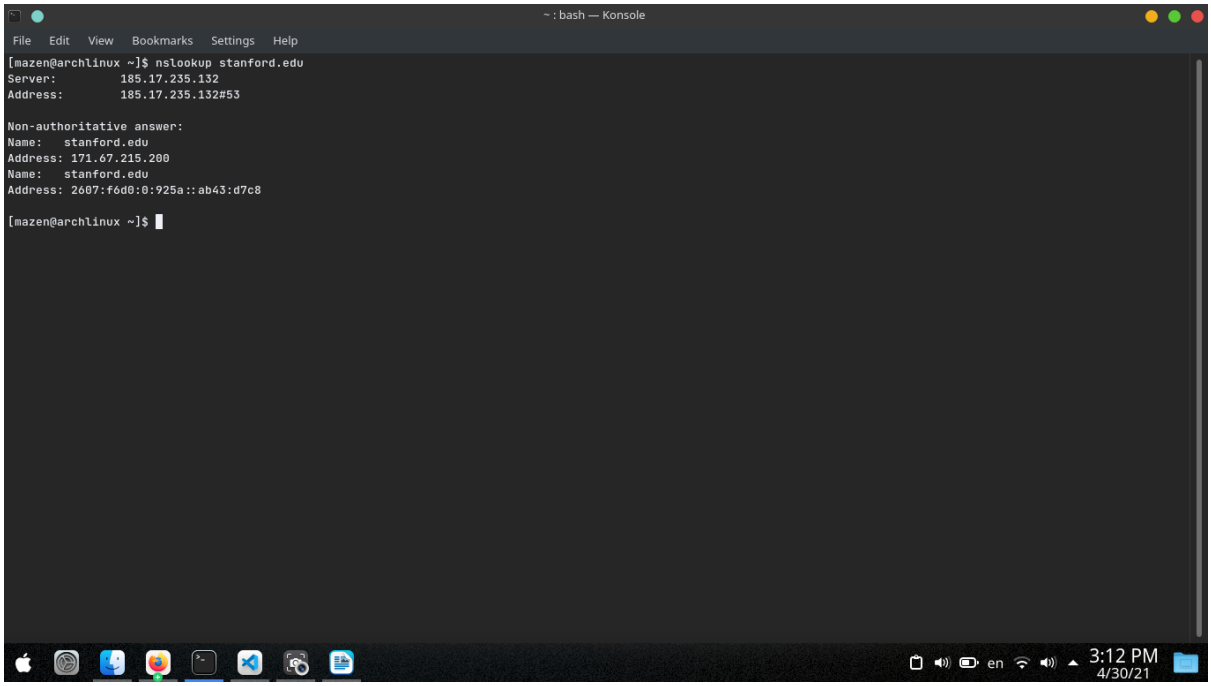

Figure 5: NSLookUp stanford.edu

5 Procedure: Part II

5.1 '/' and '/index.html'

This API has been tested from the same machine and another machine on LAN. We can see that the index.html is returned in both cases and the page is loading perfectly.

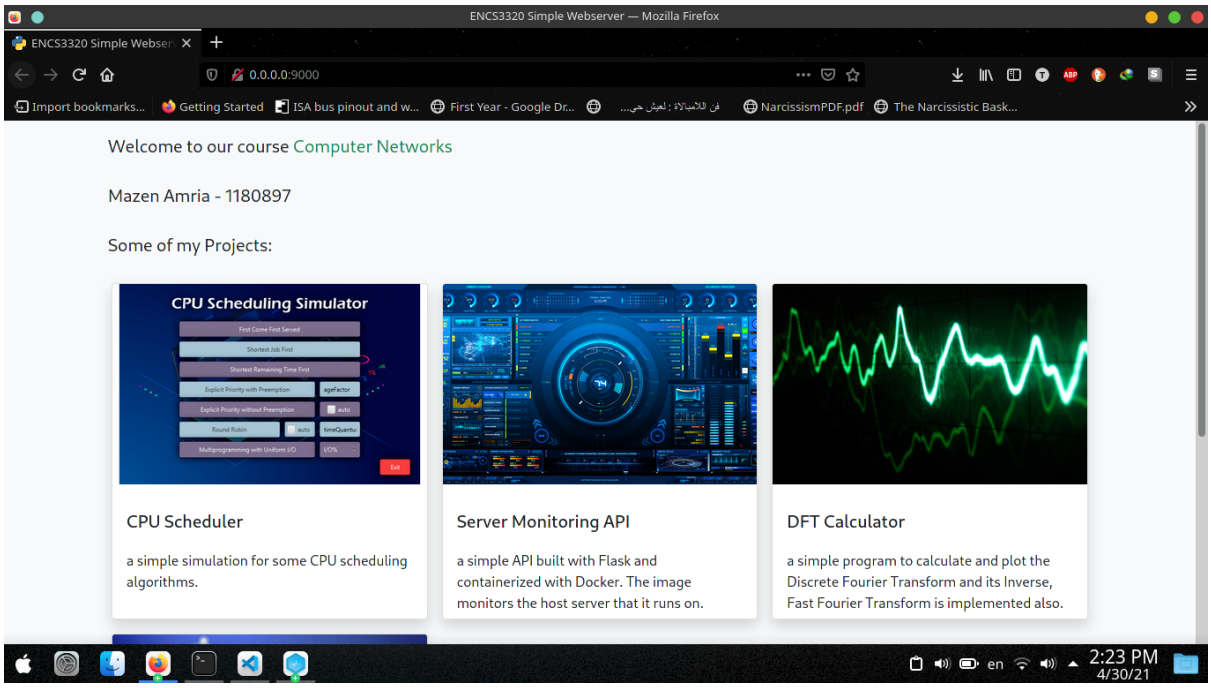

Figure 6: Accessing '/' from localhost

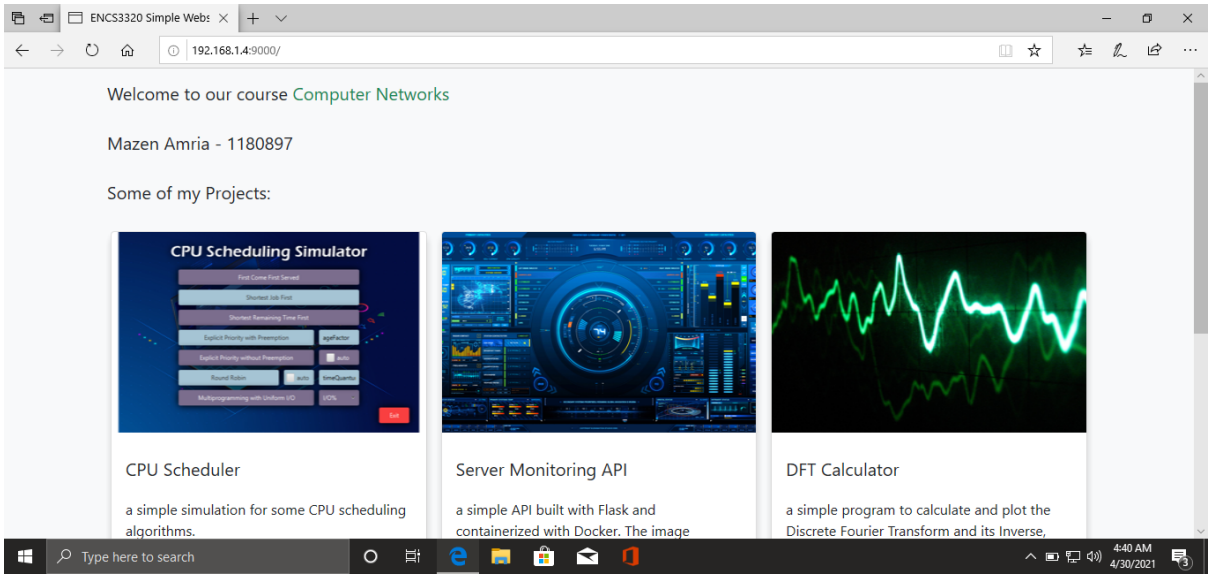

Figure 7: Accessing '/' from LAN

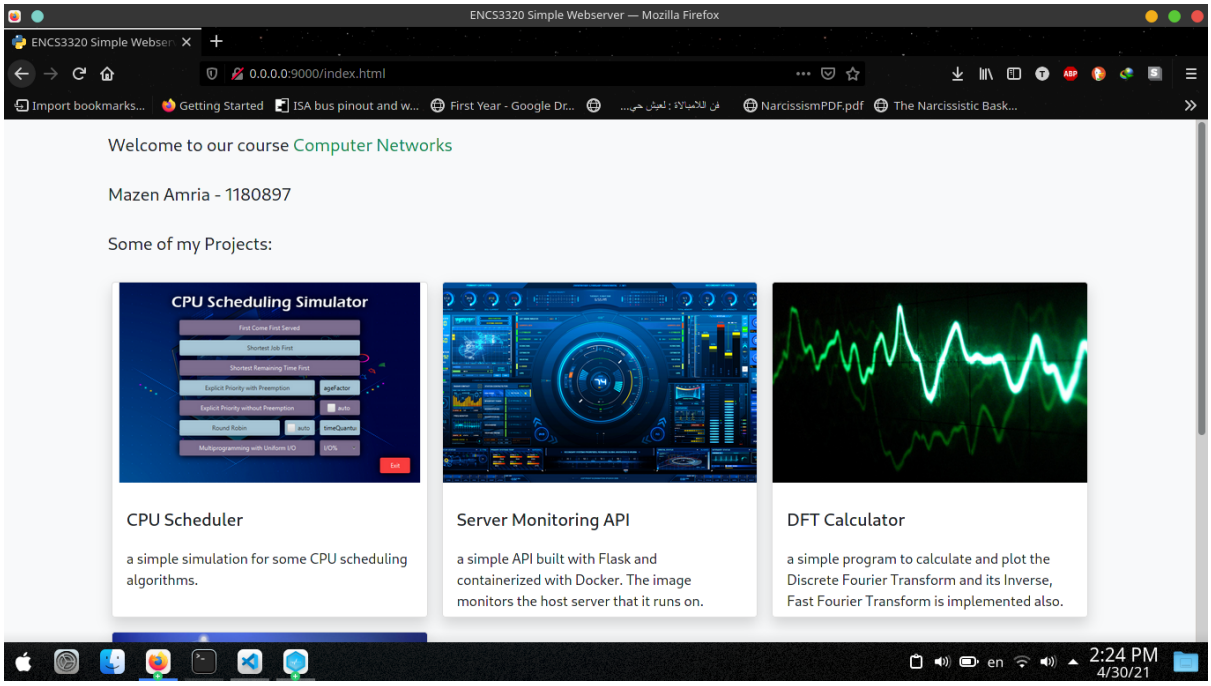

Figure 8: Accessing '/index.html' from localhost

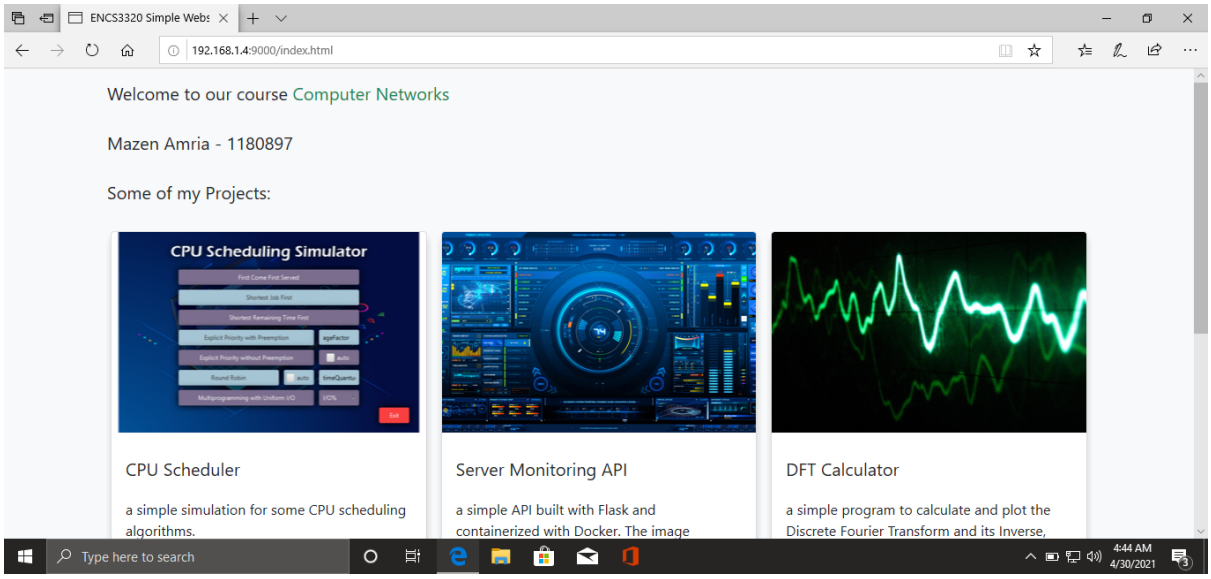

Figure 9: Accessing '/index.html' from LAN

Also we can see the output of the server process, which prints the request and then prints some logging lines indicating the request and the response status. (request is printed inside the red rectangle, and logging inside the green rectangle)

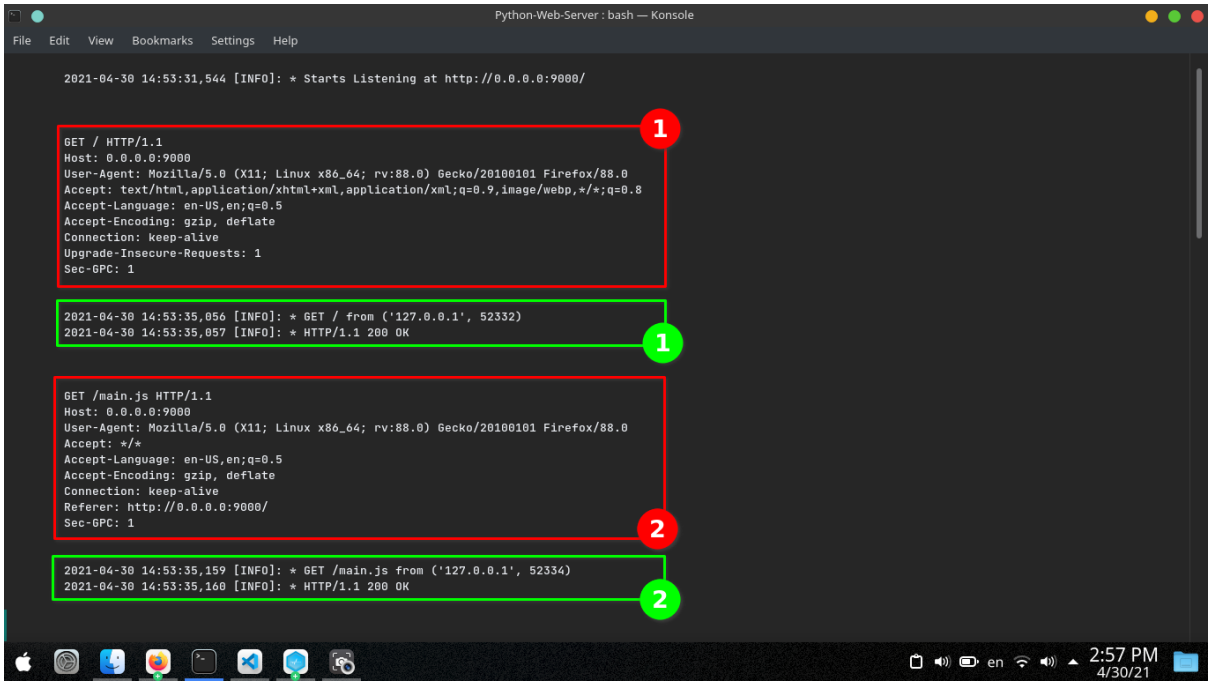

Figure 10: Output of the server process

We can see that the server responded with 200 as status code, which means that the response status is success. We can see both html and javascript files has been transferred, this means that 'Content-Type': 'application/javascript' and 'text/html' is working fine.

5.2 '/⟨filename⟩.⟨ext⟩'

In this part we tested the default API that sends files with the appropriate 'Content-Type', for the 'application/javascript' and 'text/html' we've tested them in the previous part, for the other types, first we tested the images, we can see from the following figures that the API is able to send images with both 'jpg' and 'png' format (i.e 'Content-Type': 'image/png', 'image/jpg').

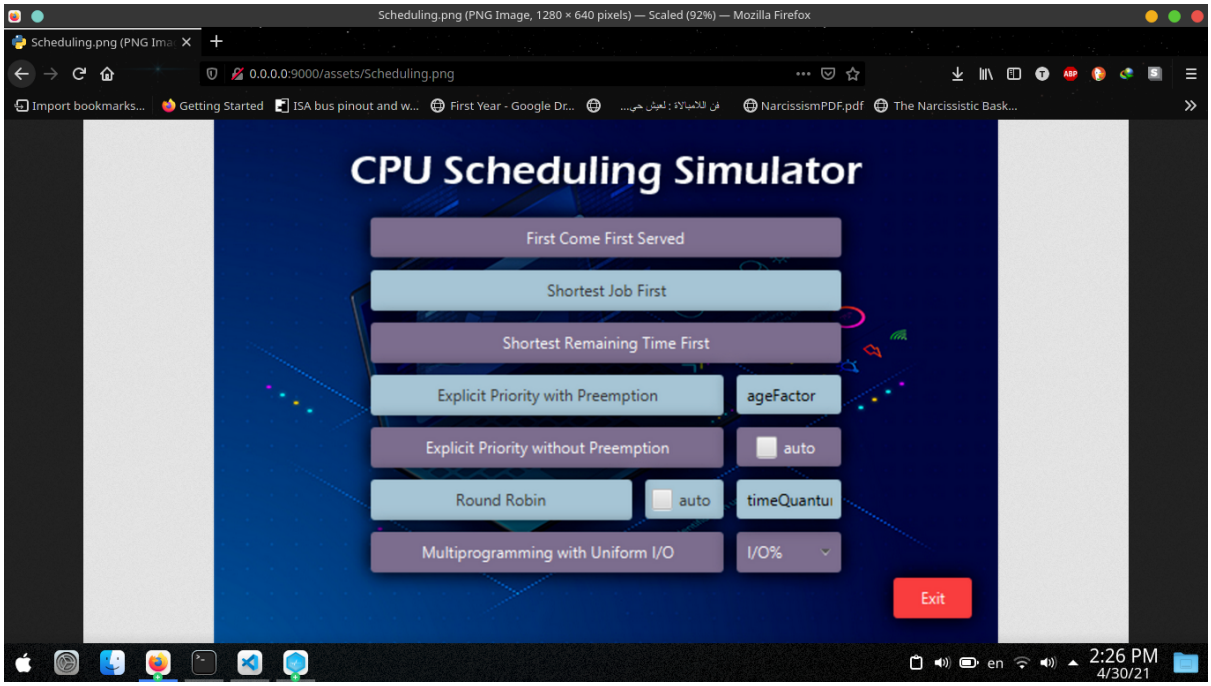

Figure 11: Accessing '/assets/Scheduling.png' from localhost

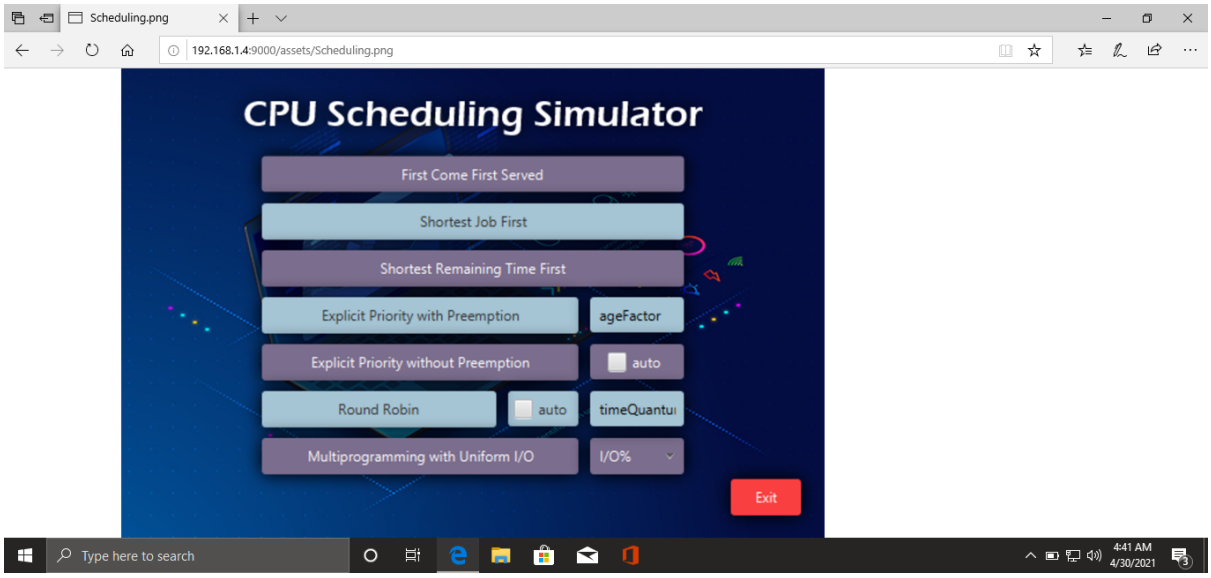

Figure 12: Accessing '/assets/Scheduling.png' from LAN

Figure 13: Accessing '/assets/Server-Monitor.jpg' from localhost

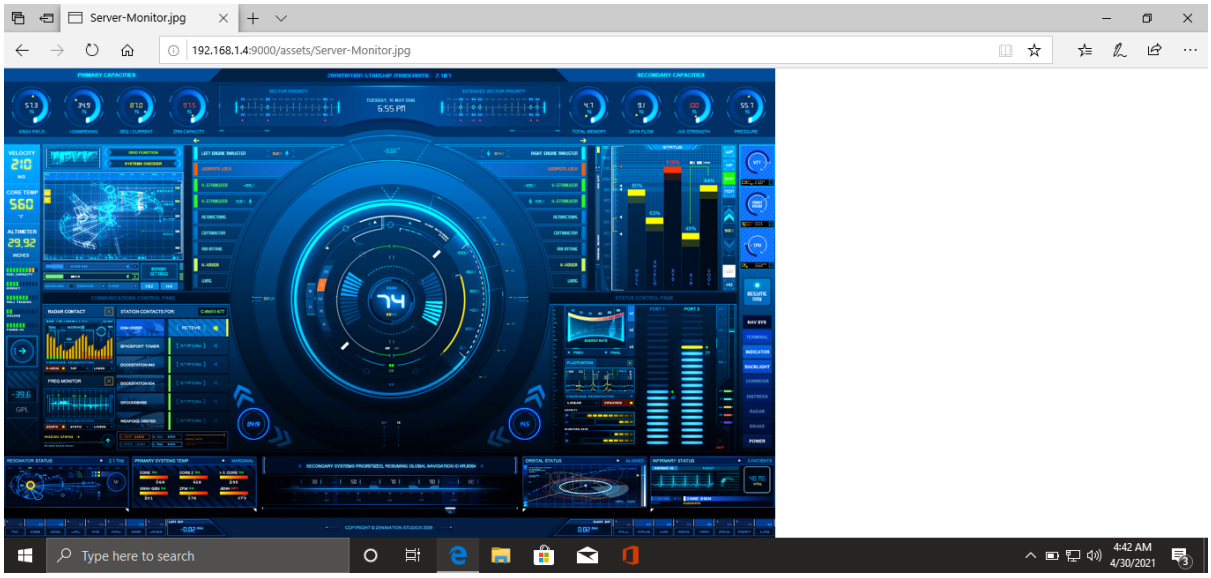

Figure 14: Accessing '/assets/Server-Monitor.jpg' from LAN

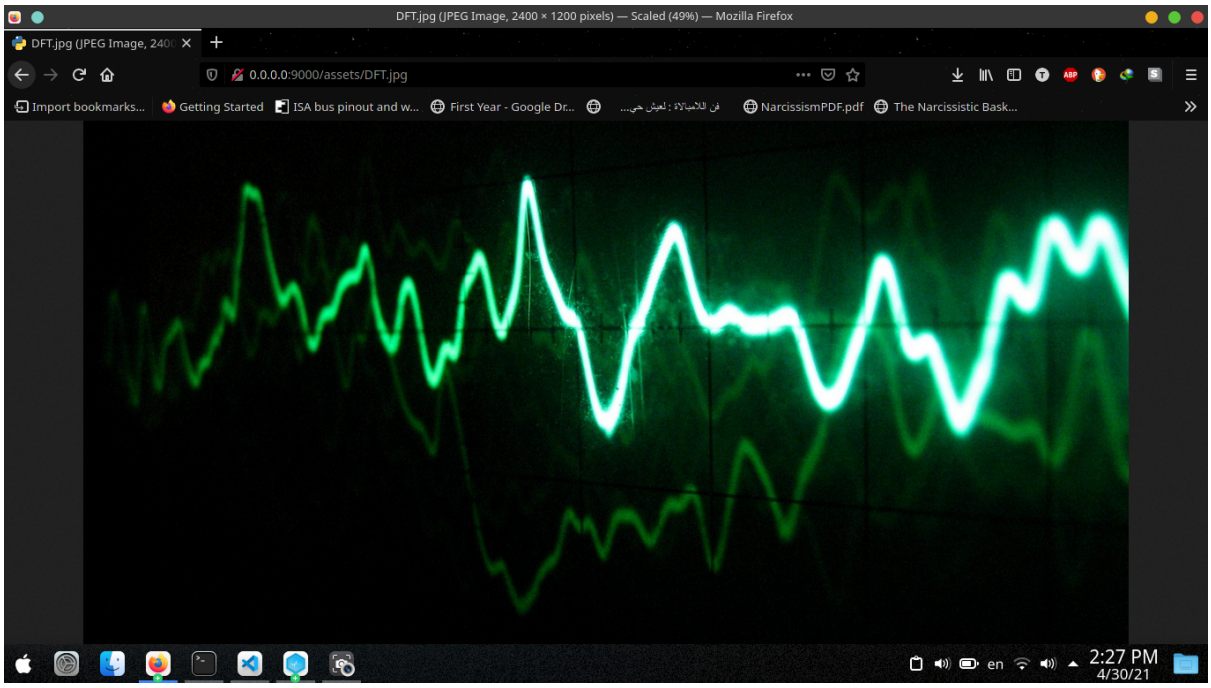

Figure 15: Accessing '/assets/DFT.jpg' from localhost

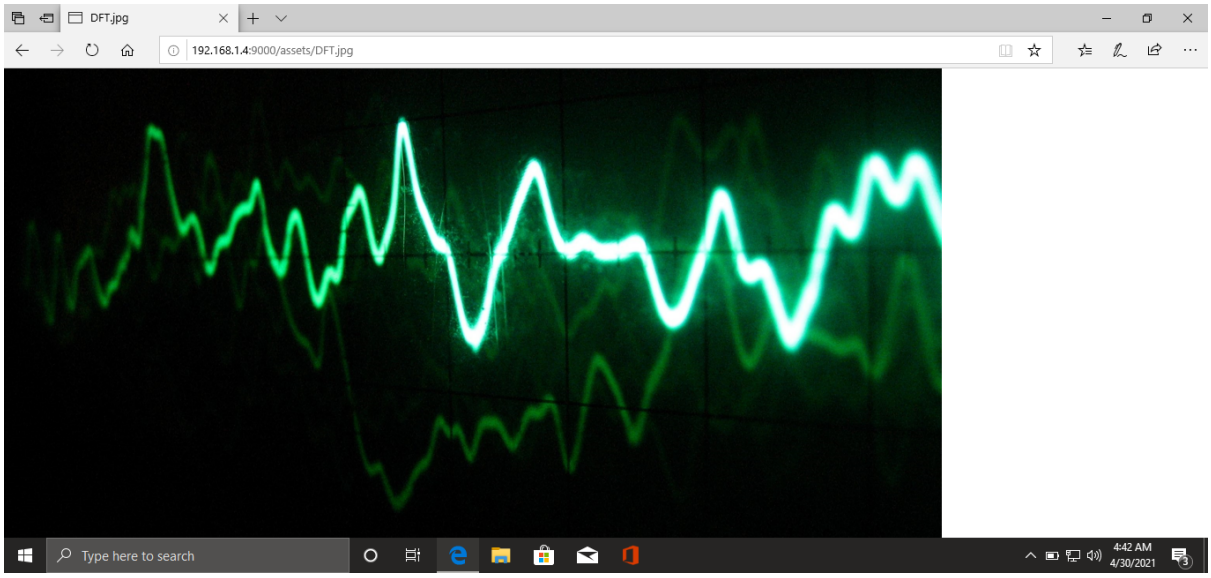

Figure 16: Accessing '/assets/DFT.jpg' from LAN

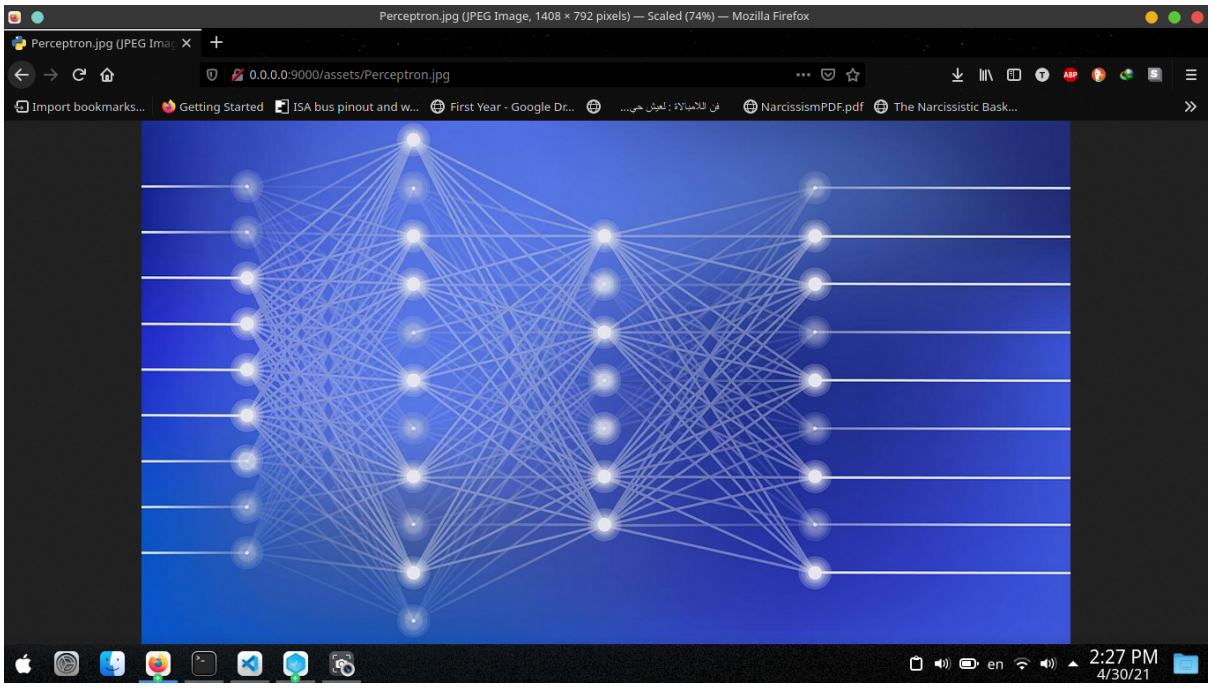

Figure 17: Accessing '/assets/Perceptron.jpg' from localhost

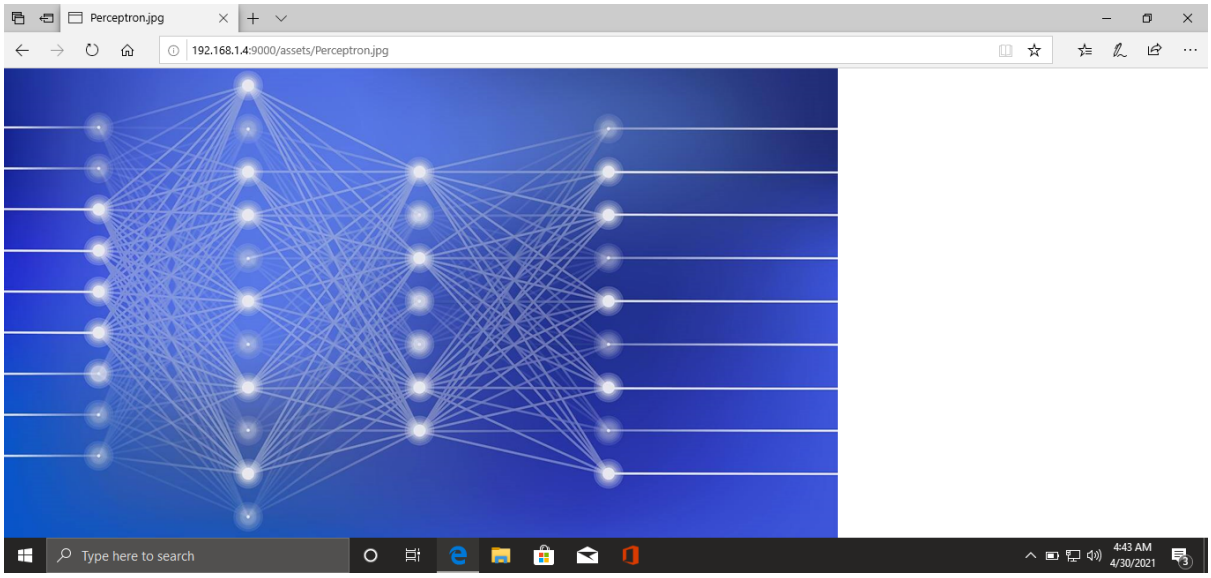

Figure 18: Accessing '/assets/Perceptron.jpg' from LAN

Then we tested 'application.json' and 'text/csv', for the 'application/json' we can see that Mozilla Firefox supports viewing 'application/json' in pretty format, as shown in the following figure

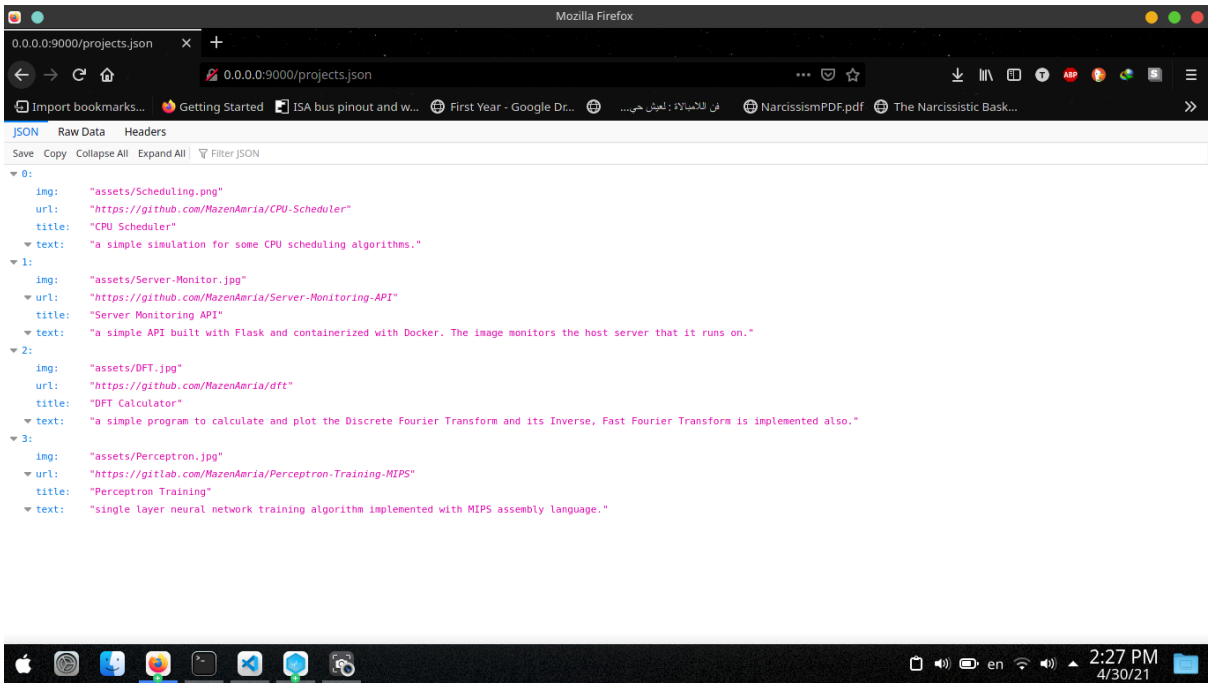

Figure 19: Accessing '/projects.json' from localhost

and for the 'text/csv' it looks like Mozilla Firefox doesn't recognized this type so it tried to download it or open it through 'LibreOffice Calc' (which is MS Excel alternative for Linux machines).

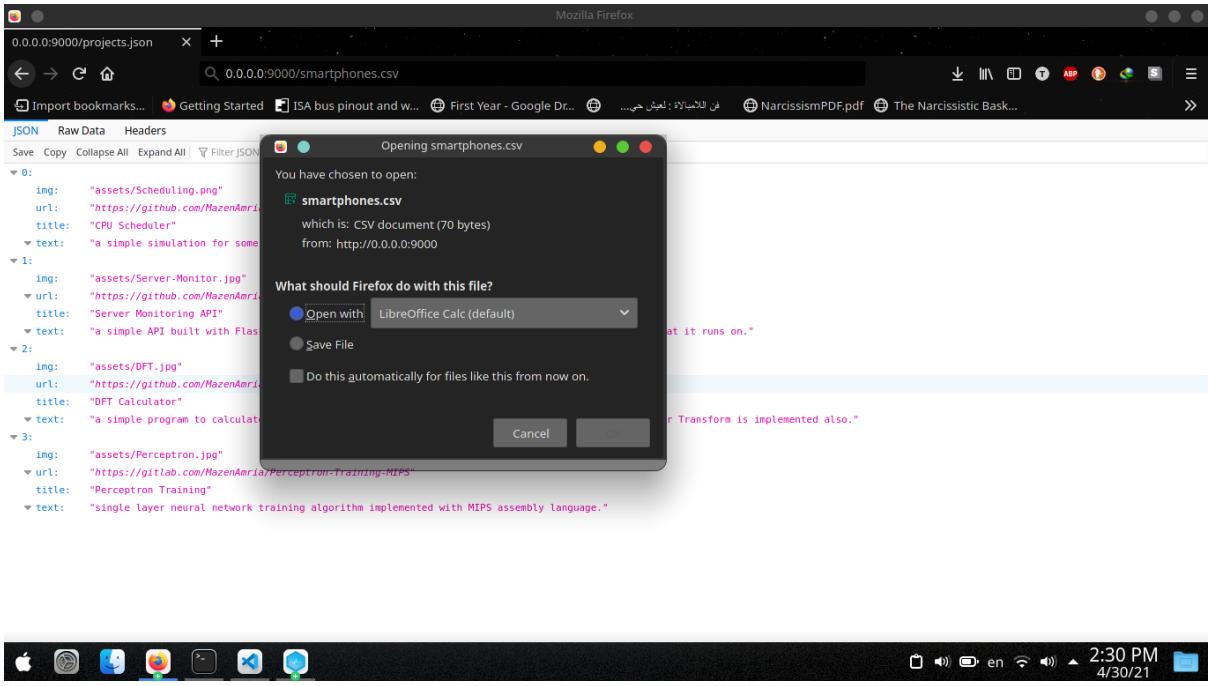

Figure 20: Accessing '/smartphones.csv' from localhost

For the windows machine, we've used Microsoft Edge as a client, it seems that it treats 'application/json' as a plain text, and also it isn't recognizing 'text/csv' so it tried to download it. In the following figure we see the '/projects.json' on the browser and '/smartphones.csv' on the download bar bellow.

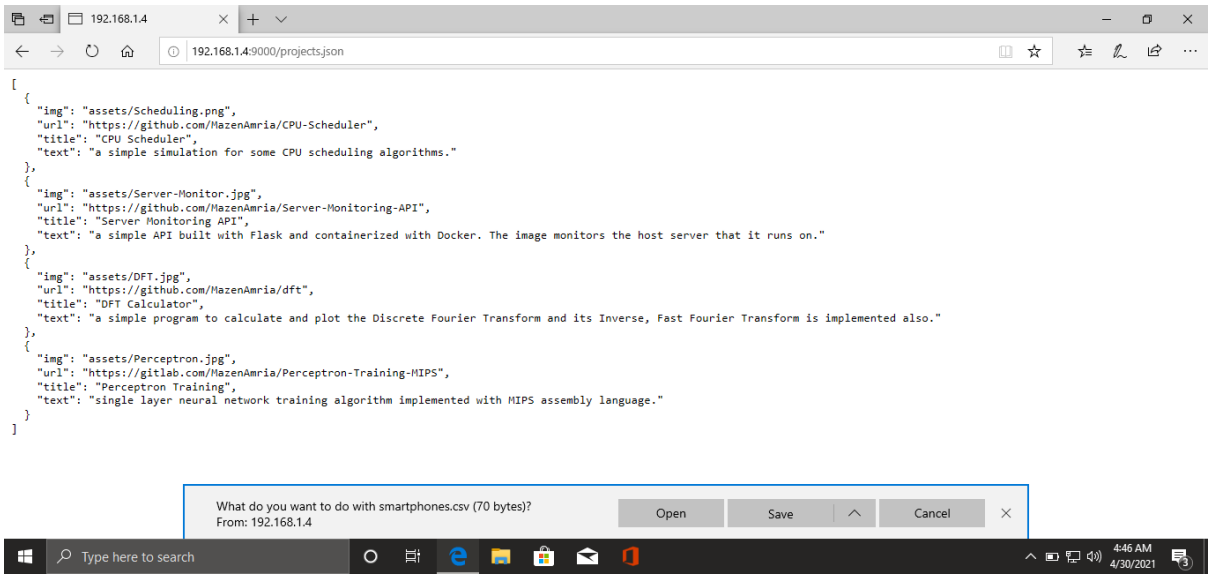

Figure 21: Accessing '/projects.json' and '/smartphones.csv' from LAN

Finally, we tried to access '/requirements.txt' to test the 'text/plain' type which is set by default. We see that it shows the text on the browser as expected.

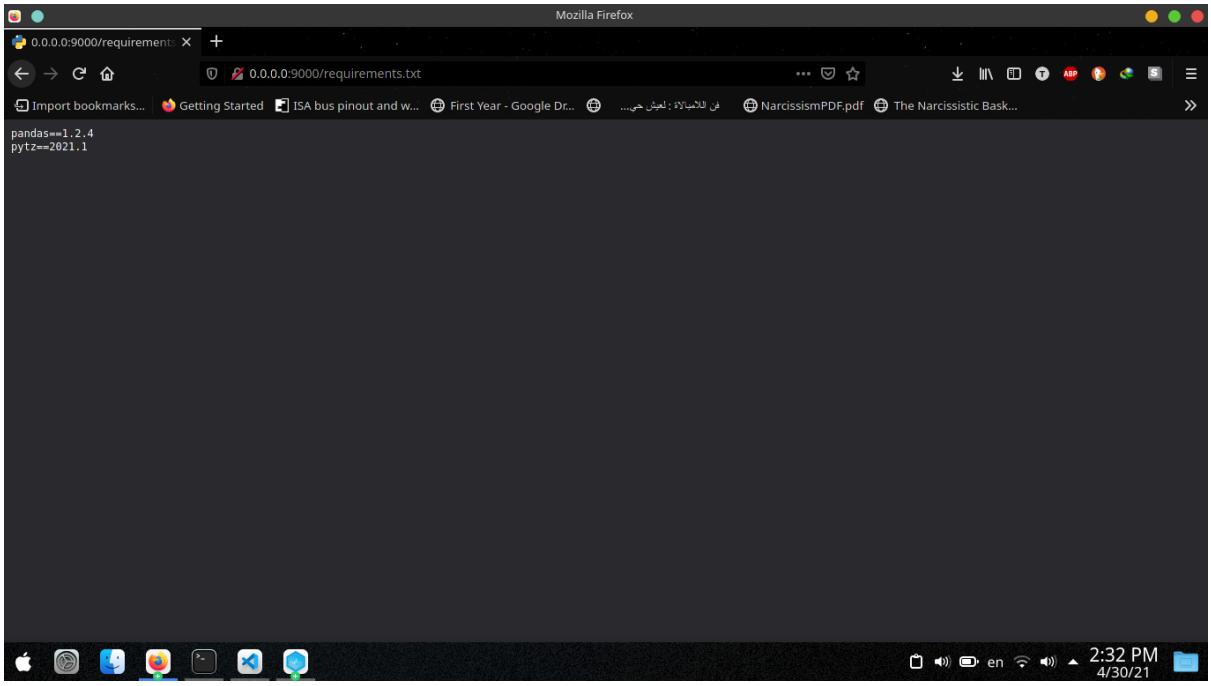

Figure 22: Accessing '/requirements.txt' from localhost

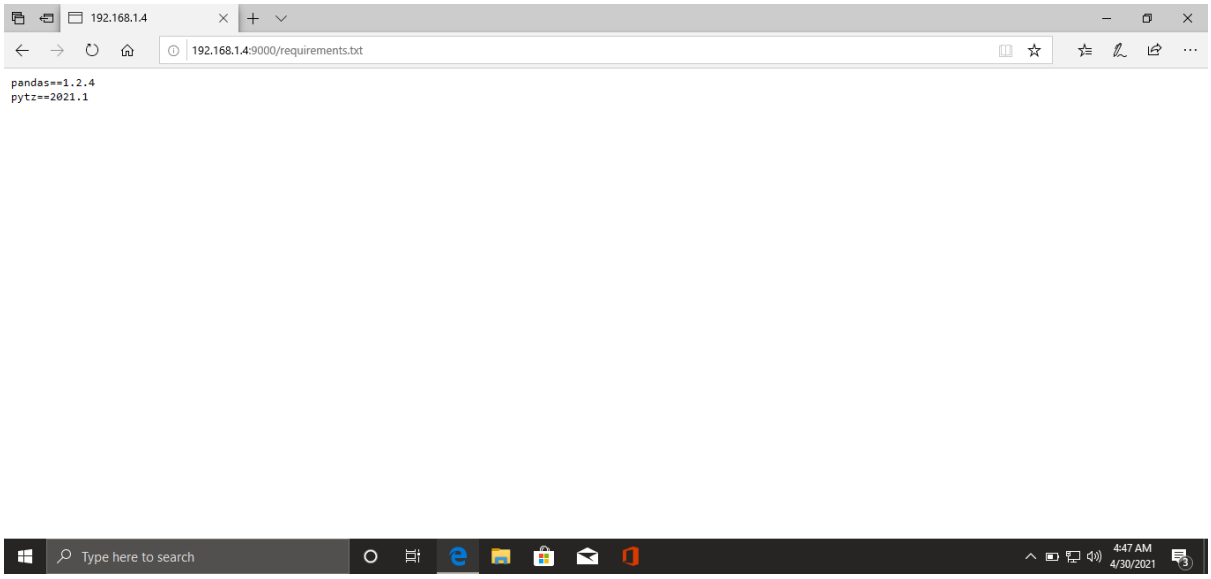

Figure 23: Accessing '/requirements.txt' from LAN

5.3 '/SortName' and '/SortPrice'

In this API we sorted the content of 'smartphones.csv' based on the required column and then sent the result as 'application/json' format, we know that the API supposed to work in 'text/plain' or 'text/html' types but we think that 'application/json' is the leading format for data transfer and APIs, hence it would be better to build all out APIs on that format. Trying to sort by name, we see that 'Phone 1' is the first element and 'Phone 5' is the last element.

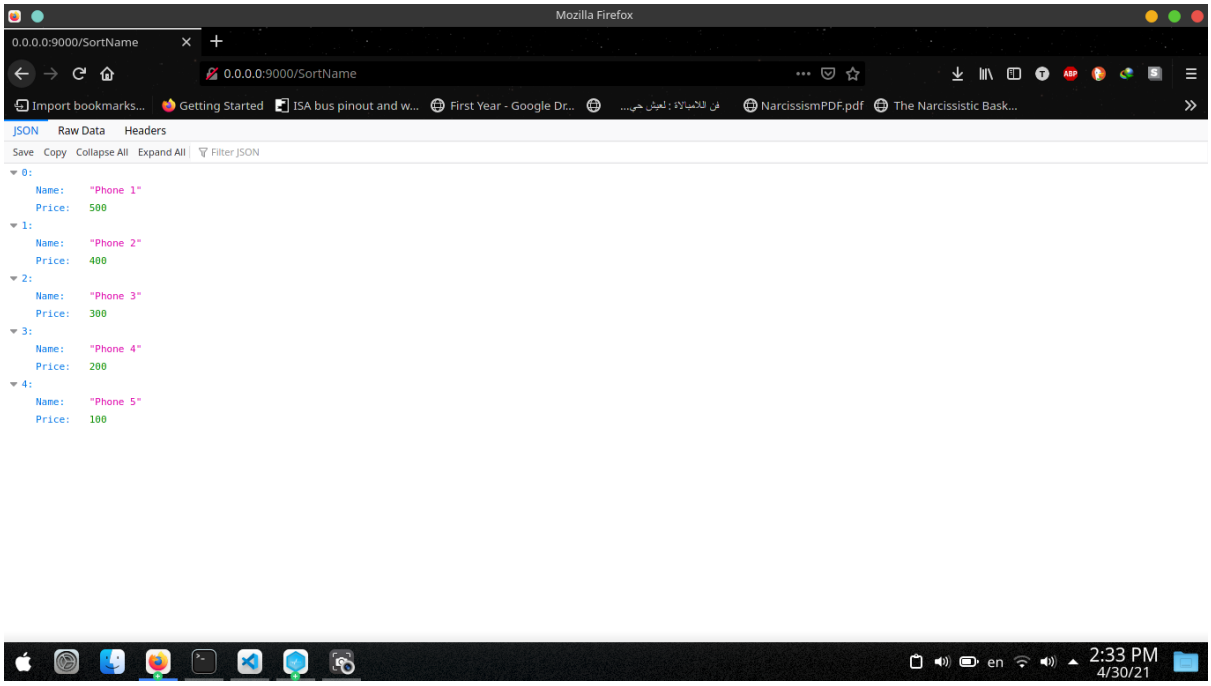

Figure 24: Accessing '/SortName' from localhost

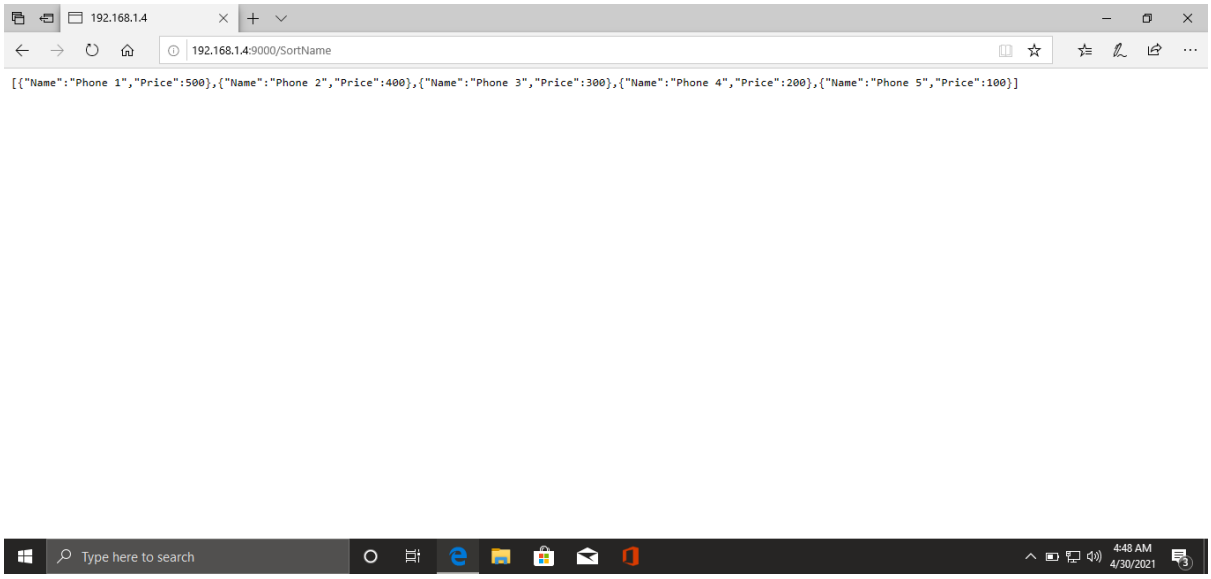

Figure 25: Accessing '/SortName' from LAN

For sorting the price we see that 'Phone 5' with 100 is the first element and 'Phone 1' with 500 is the last element.

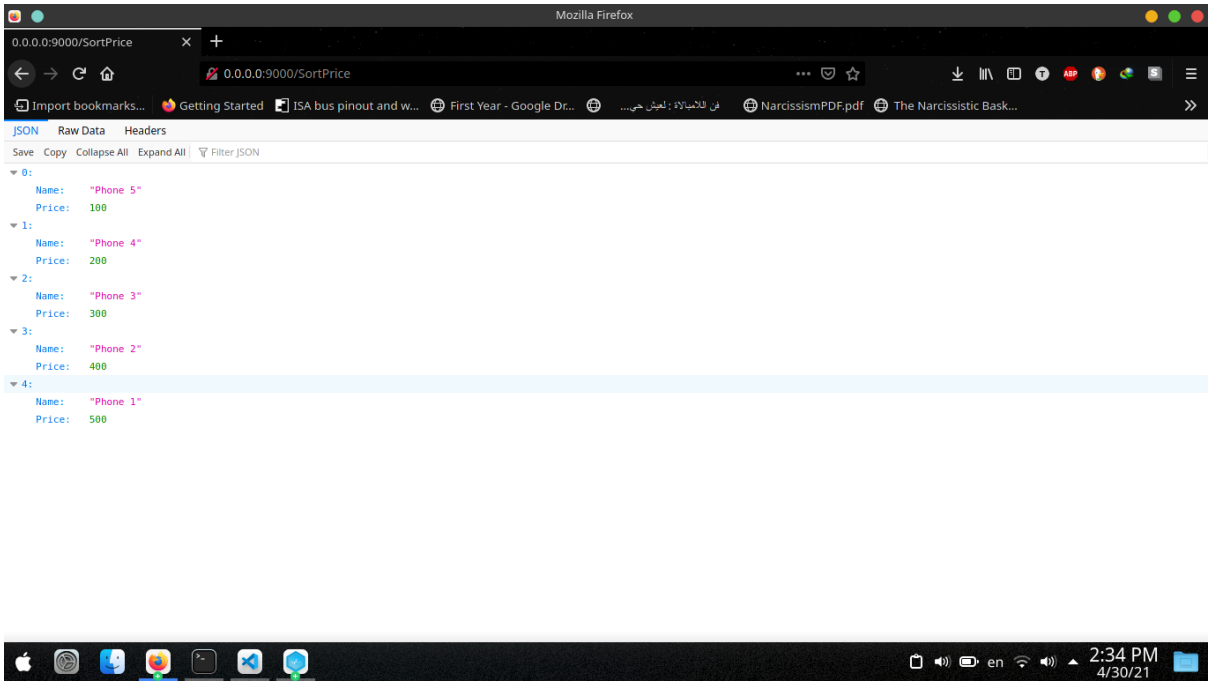

Figure 26: Accessing '/SortPrice' from localhost

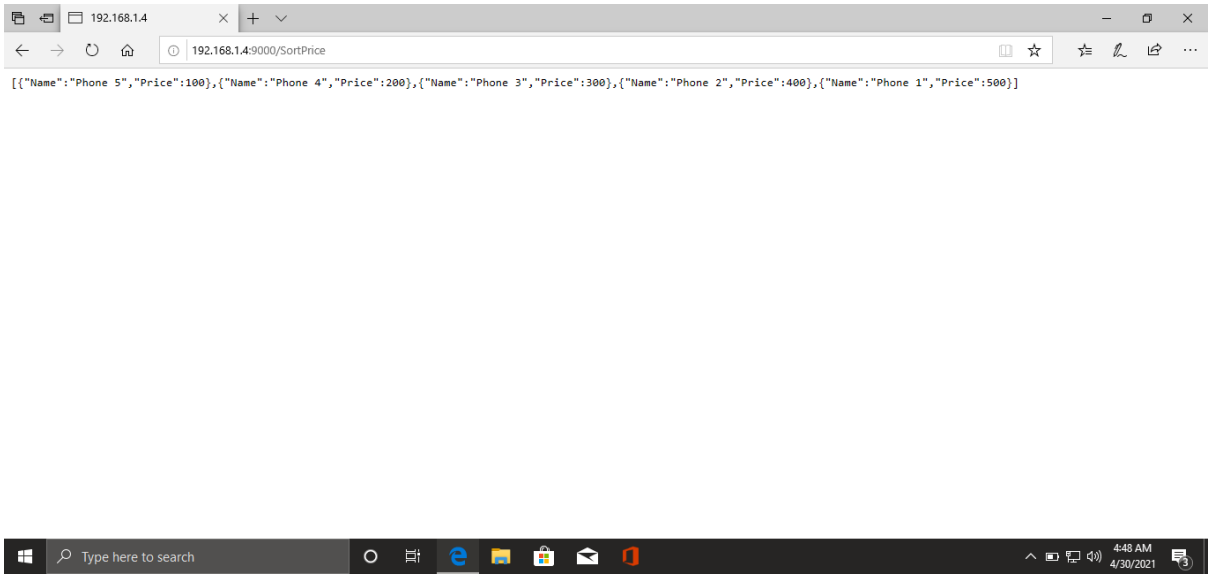

Figure 27: Accessing '/SortPrice' from LAN

This means that the '/Sort*' API is working as expected.

5.4 404: Not Found

If all the handlers that matches the request path failed to send response then an 404 message will be sent, the output of the page includes the client's IP and port, and the server's IP and port, we know that the server has static IP and port, but accessing the server from different places leads to accessing it from different interfaces, each has it's IP, for example if the server's IP is 127.0.0.1 the client must be on the same machine as the host, since he's able to see the loopback interface.

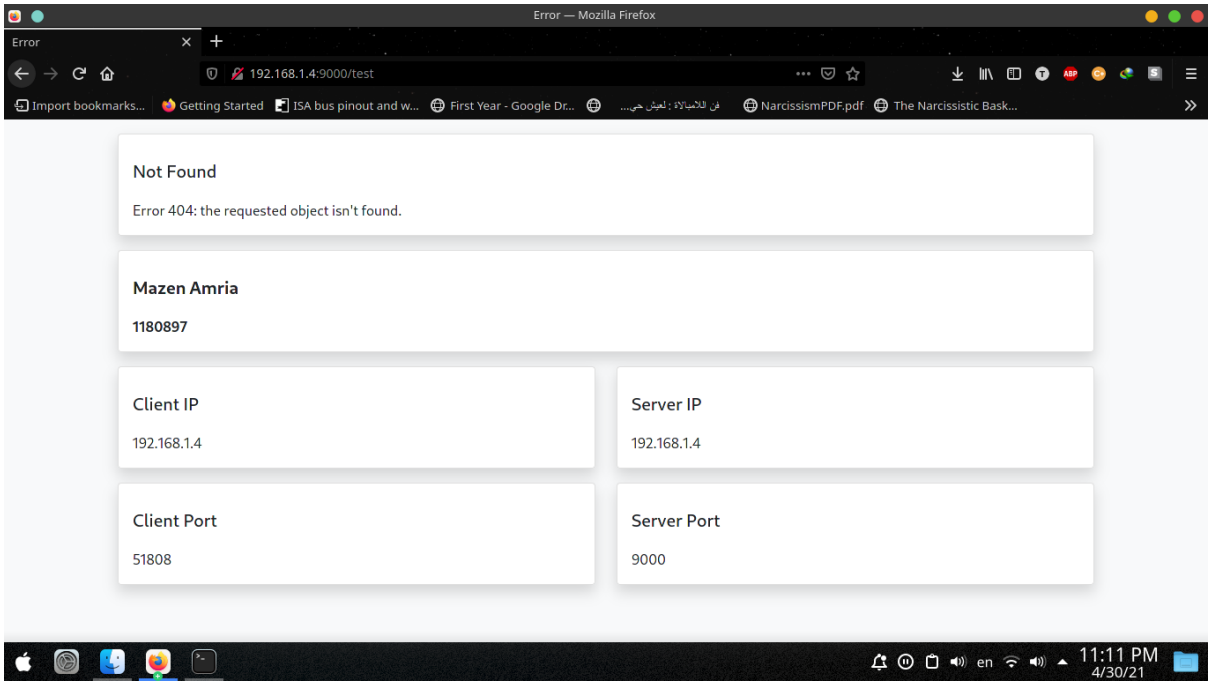

Figure 28: 404: Not Found

See, even if the client and the server are both at 192.168.1.4 but still not showing 127.0.0.1 that because we've accessed the 192.168.1.4 which is the LAN interface not the loopback, and the output of the server indicates the error.

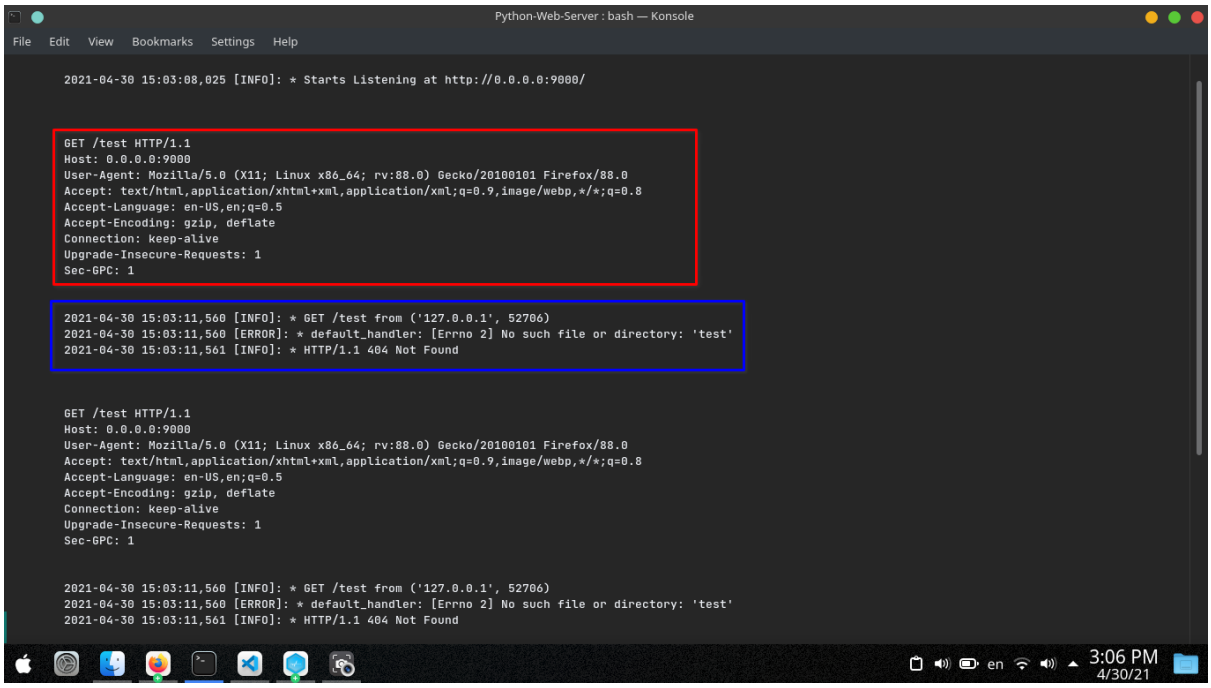

Figure 29: Server Output

6 Conclusion

Web Servers have to support multiple types of content, and It's important to specify as many HTTP headers as possible, since it enables other software (receiving the request or the response) to decide how to deal with data and whether it's valid or not.

7 Future Work

For a Web Server, we believe that multi-threading or multi-processing is one of the most essential parts of it, as it allows multiple connections at the same time, but for the purposes of this project this is not required. Also many 'HTTP' methods exists, but in this project only the 'GET' method is implemented. For a professional web server it must consider other methods. Note that allowing more methods require to deal with the request headers also and make use of them carefully, so for a professional web server it may also consider to cover as many parts as it can from the 'rfc2616' standard.

References

[1] BalusC. *Maximum length of HTTP GET request.* URL: `https://stackoverflow.com/a/2659995`.

[2] *Hypertext Transfer Protocol – HTTP/1.1.* URL: `https://tools.ietf.org/html/rfc2616`.