Computer Networks ENCS3320

Birzeit University, Department of Electrical and Computer Engineering

Report of Project 1, Friday, 22 December 2023

Instructor: Dr. Abdalkarim Awad

Socket Web Server Networking Project

Sondos Aabed 1190652

Contents

Introduction

This report comes as a part of the course work in Computer Networking. It is the initial project about socket programming, and web server functionality. The practical implementations, ranging from the fundamental network commands to the implementation of TCP client and server applications are reported with their source code.

Tools

In developing the solution for this project multiple tool were used:

- **GitHub and Git** were used in version control for the source code. Find the repository in the following link. Please Note if the repository was not found means that it was still private to not give up the solutions.
https://github.com/sondosaabed/Socket-WebServer-Networking-Project

- **Visual Code Studio** was used as IDE for writing and pushing the source code.

- **Python Socket Library** was used to implement socket programming.

Source Code

The following figure shows the Structure of the source code of the solution. The solution was divided into three parts. Each part has it's notebook (Interactive python) solution. Each division also has a folder of screenshots used in the report. And finally the src directory that contains the source code.

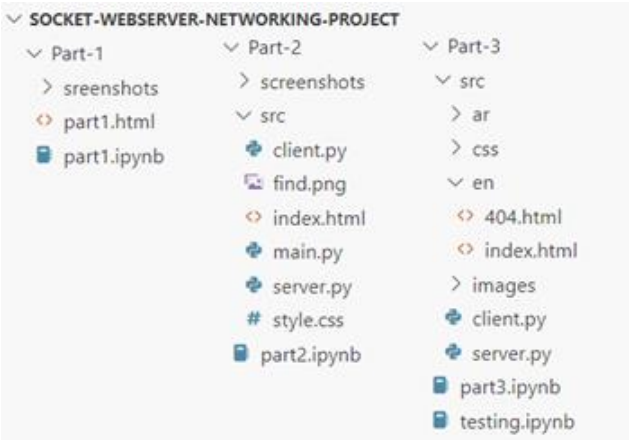

Figure (0): Project Structure

Each part has an html index that is navigable to the other parts. As shown in the next Figure.

Figure (1): Navigation of Solution

Part1: Understanding Network Commands

In this part, the fundamental network commands are defined by the writer's language and are executed, commented upon with screenshots taken.

Definitions

- **Ping**: is the command that is used when there is a desire to check if a hosting server is responsive.
- **Tracert**: it is the tool that is used to trace the route that response (packets) tasks until reaching the destination.
- **Nslookup**: is a command that is used to find the IP address of a specific server.
- **Telnet**: is a tool that is used to remotely connect to a device.

Commands Execution

1) Pinging a Mobile phone

```
ping 192.126.1.oo
```

- This is the command used:
- This is the **output of Pinging** a mobile phone that is in the same network of the laptop.

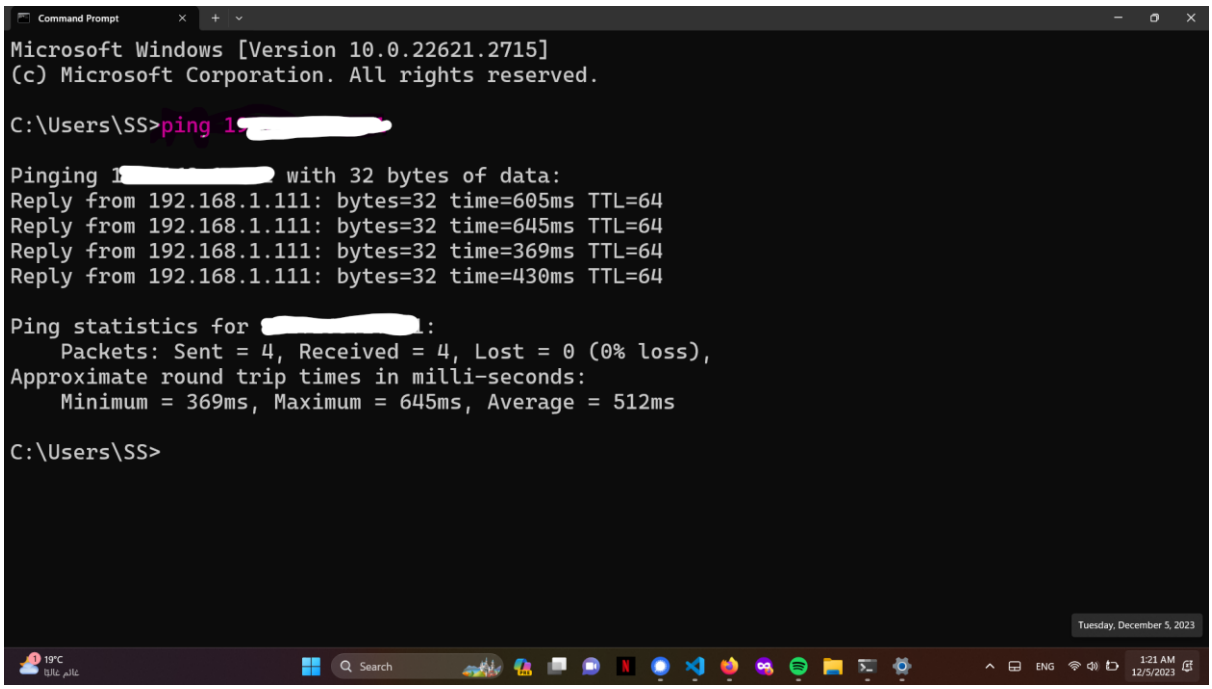

Figure (2): Ping Mobile Phone

Observations on execution: The packet is specified with a size of 32 bytes and each request will contain 32 bytes of data payload.

- The destination device has received the request packet successfully, and a reply has been received from the specified IP address.
- TTL has an initial value of 64.
- There was **no loss,** each packet request had a corresponding packet response.
- Number of packets sent and received is 4.
- Round Trip Time: min = 369ms, max = 645ms, and average value is 512 ms for each packet to travel from source to destination devices and back.

2) Pinging a website

- This the command used:
- This is the **output of Pinging** Cornell website.

```
ping www.cornell.edu
```

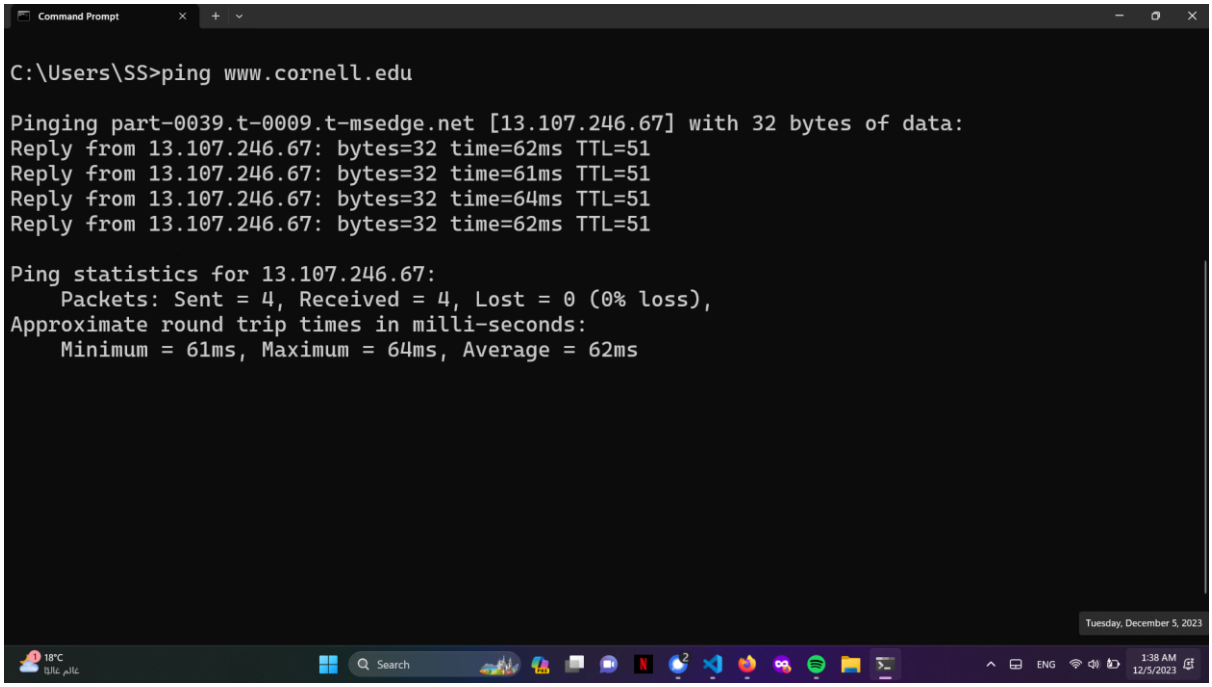

Figure (3): Ping Cornell Output

Observations on execution: The packet is specified with a size of 32 bytes and each request will contain 32 bytes of data payload.

- The line reply indicates that the destination device has received the request packet successfully, and a reply has been received.
- TTL has an initial value of 51.
- There was no loss, meaning that each packet request has had a corresponding packet response.
- Number of packets sent and received is 4.
- RTT maximum = 64ms, minimum = 61ms, and average value is 62 ms, for each packet to travel from source to destination devices

3) Tracer a Website

- This the command used:

```
tracert www.cornell.edu
```

- This is the **output of tracert** Cornell website.

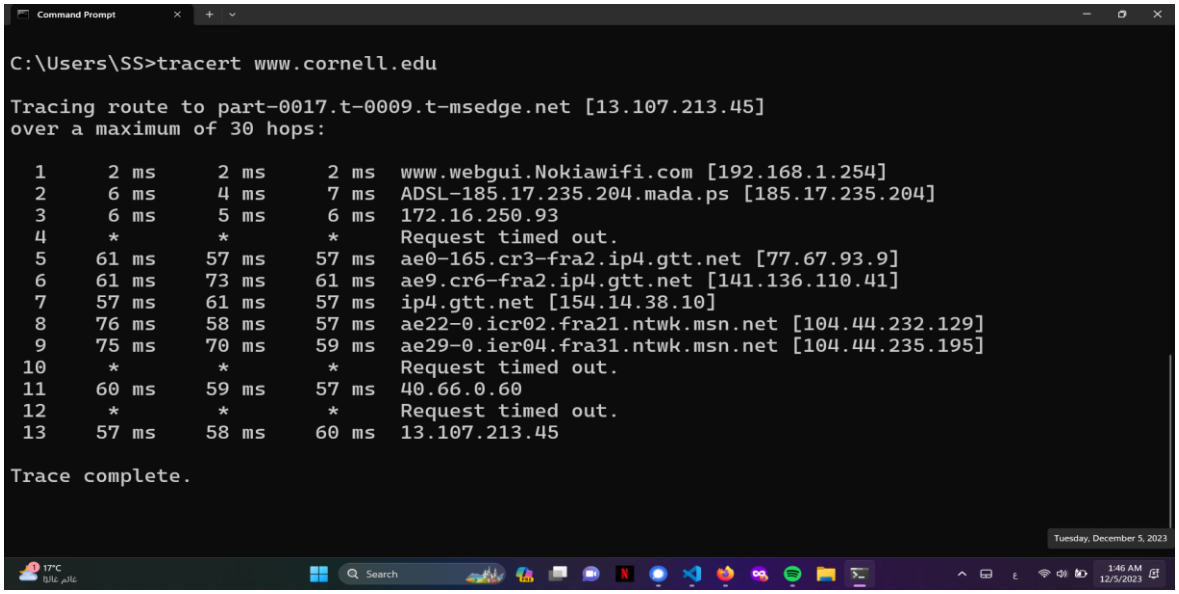

Figure (4): Tracert Cornell Output

Observations on execution: Maximum number of hops or network devices the command will attempt to reach before stopping the trace is 30 hops.

- For each hop, the output indicates the hop number, the round-trip time in ms, which starts at 2ms and ends at 58ms, for the ICMP or UDP packets to reach that hop and return, and the IP address of the router at
- that hop. The first IP Address is my network IP address, and the last IP address is usually associated with the destination IP Address if the trace is successfully completed.
- (Trace complete) has finished tracing the route, and the output provides information about the routers traversed from computer to reach Cornell.

4) Nslookup a Website

```
nslookup www.cornell.edu
```

- This the command used:
- This is the **output of nslookup** Cornell website.

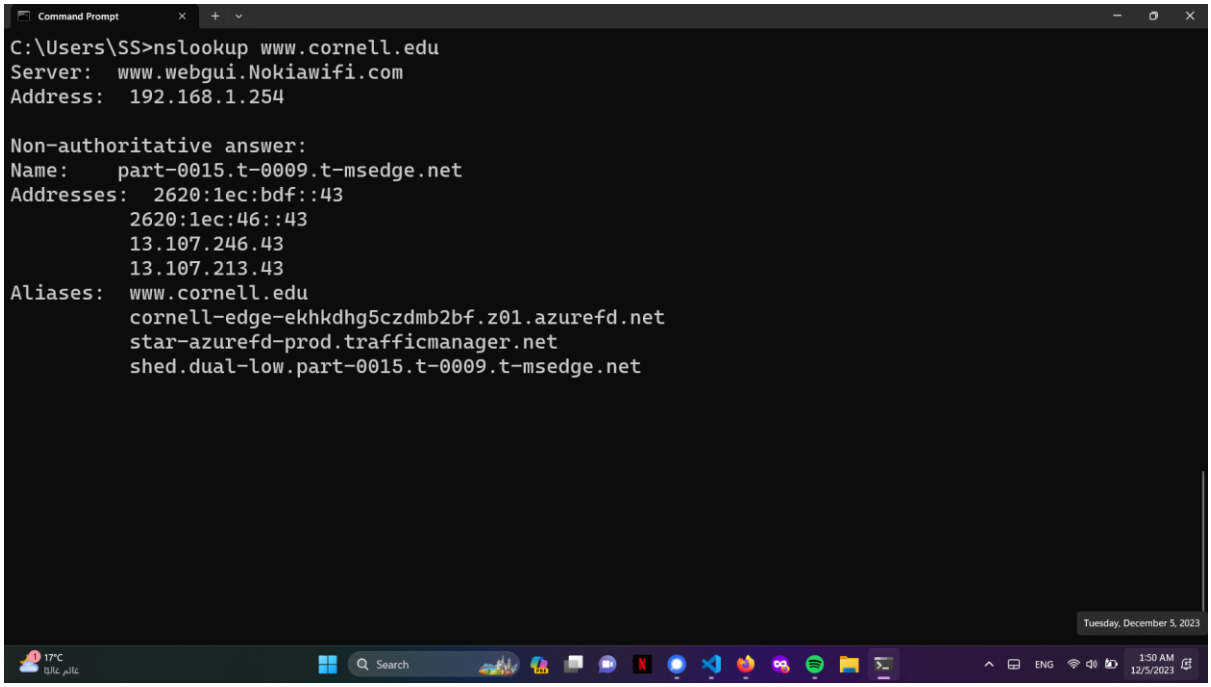

Figure (5): Nslookup Cornell Output

Observations on execution: The output displays that the DNS server used for the query is www.webgui.Nokiawifi.com.

- IP address 192.168.1.254
- It also provides the IP addresses associated with the domain name www.cornell.edu , which are “2640:1ec:bdf::43” and “13.107.246.43”.
- Alias list, www.cornell.edu , which is an alternate name.

5) Wireshark Analysis

- After the installation was done the app was used and the Wifi network was chosen to capture DNS messages.

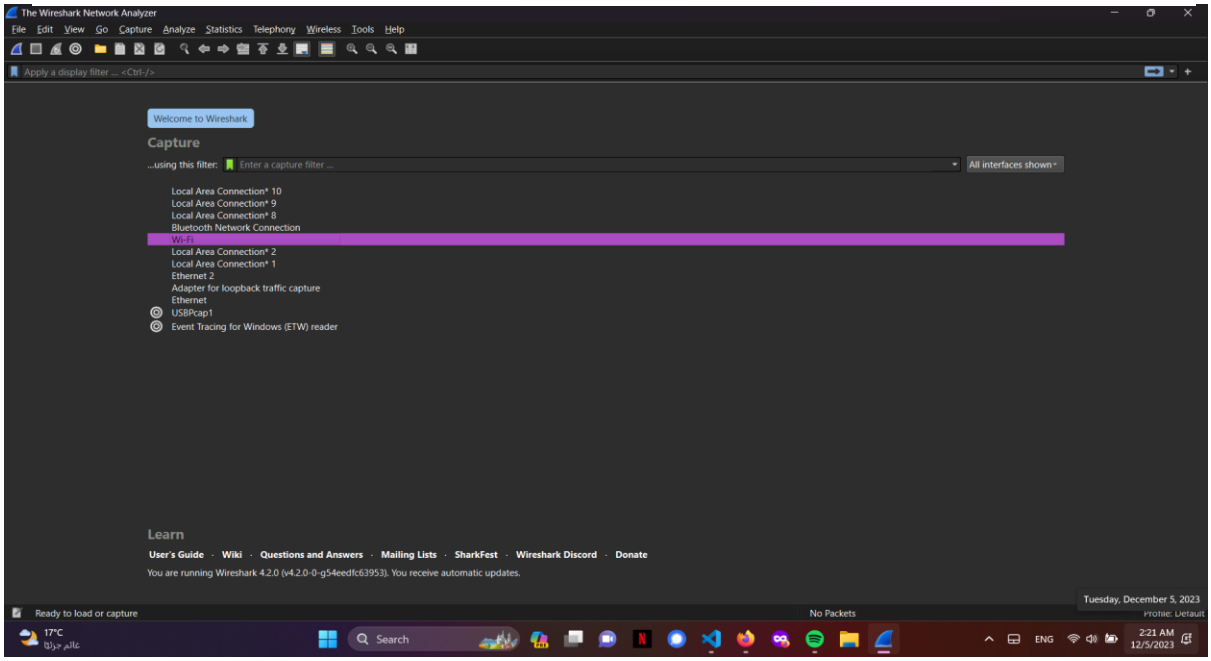

Figure (6): Choosing Wi-Fi Network

- This is the **output of applying the DNS filter**.

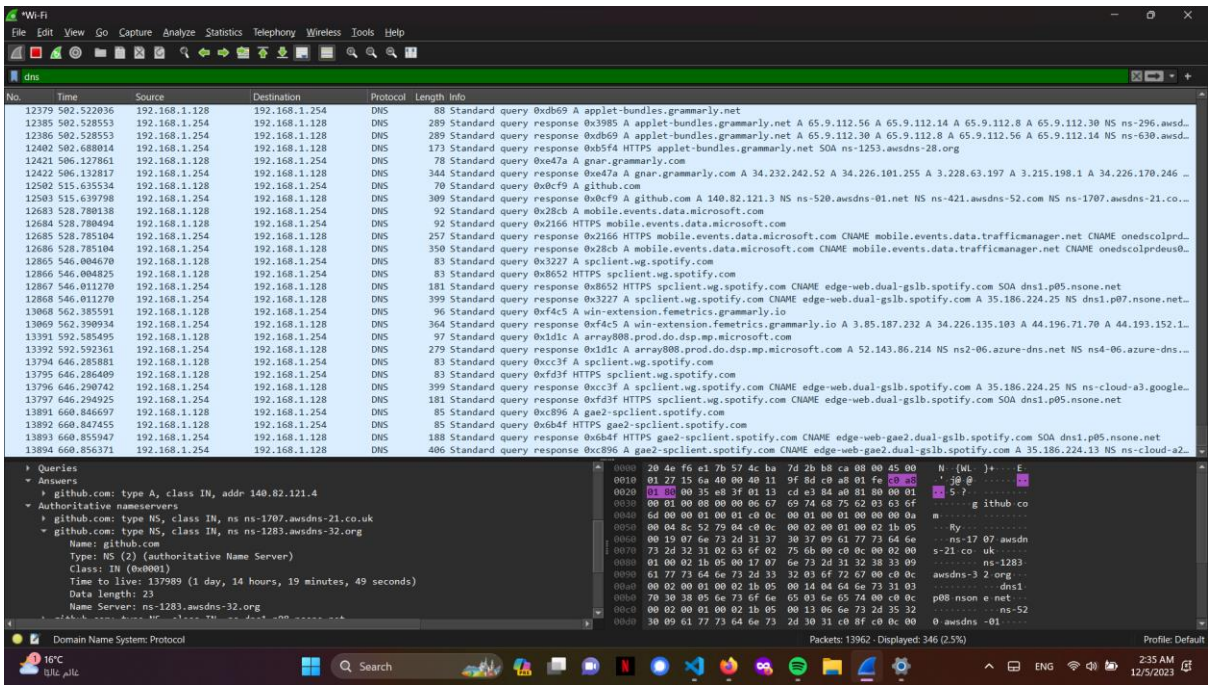

Figure (7): Results of DNS Filtering

Part2: Socket Programming Implementation

In this part, a TCP client, Server is implemented using Socket programming. The idea is to check an ID and lock the screen if not listed. Since this is not a group work, four valid IDs were provided, the writer's (1190652), (1234567), (0101010) and (2211221).

Logic behind ID check & screen Lock

The following Figures shows the logic used to validate the student ID's and to lock the screen. These functions were defined in a script called utils.py and called in the server side.

```python
def validate_student_id(student_id):
    """
        Args:
            student_id: this value is tajen from the Form
        Returns boolean:
            if entered value is in valid_id's returns True else False
    """
    valid_ids = ["1190652", "1234567", "0101010", "2211221"]
    return student_id in valid_ids
```

Python

Figure (8): Function Validating student ID

```python
import platform
import ctypes
def lock_screen():
    """
    Locks the screen based on the operating system.
    """
    system_platform = platform.system()

    if system_platform == 'Windows':
        # Implement locking for Windows using ctypes
        ctypes.windll.user32.LockWorkStation()
```

Python

Figure (9): function to lock the Screen

```python
import time
def check_id(student_id, client_socket ):
    """ To perform lock screen and send messages to client and display on server
        Args:
            student_id: the entered id
            client_socket: the created client socket
        Return:
            Nothing
    """
    if student_id:
        print(f"Received valid student ID: {student_id}")
        # Send a message to the client that the server will lock the screen
        # Display a message on the server side that the OS will lock screen after 10 seconds
        client_socket.send("Locking screen in 10 seconds...".encode("utf-8"))
        print("OS will lock screen after 10 seconds")

        time.sleep(10) # Wait for 10 seconds
        lock_screen()

        # Send a response to the client after locking the screen
        client_socket.send("Screen locked.".encode("utf-8"))
    else:
        # If the received message is not a valid student ID, display an error message
        print("Error: Invalid student ID or message")
```

Python

Figure (9): function to lock the Screen

This is what was done:

- Send a message to the client that the sever will lock screen after 10 seconds
- Display a message on the server side that the OS will lock screen after 10 seconds.
- Then waited 10 seconds

TCP Client and Server

Socket Programming was used to implemnet a TCP connection.

```
import socket
```

```
def run_client(server_ip = "127.0.0.1", server_port = 9955 ):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create a socket object
    client.connect((server_ip, server_port))# establish connection with server

    while True:
        msg = input("Enter Student ID: ") # input message and send it to the server
        client.send(msg.encode("utf-8")[:1024])

        response = client.recv(1024) # receive message from the server
        response = response.decode("utf-8")

        if response.lower() == "closed":
            # if server sent us "closed" in the payload, we break out of the loop and close our socket
            break

        print(f"Received: {response}")

    # close client socket (connection to the server)
    client.close()
    print("Connection to server closed")
```

Figure (10): function to run the Client

```
def run_server(server_ip = "127.0.0.1", port = 9955):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create a socket object

    server.bind((server_ip, port))  # bind the socket to a specific address and port
    server.listen(0)    # listen for incoming connections
    print(f"Listening on {server_ip}:{port}")

    client_socket, client_address = server.accept()     # accept incoming connections
    print(f"Accepted connection from {client_address[0]}:{client_address[1]}")

    # receive data from the client
    while True:
        request = client_socket.recv(1024)
        request = request.decode("utf-8") # convert bytes to string

        # if we receive "close" from the client, then we break out of the loop and close the connection
        if request.lower() == "close":
            # send response to the client which acknowledges that the
            # connection should be closed and break out of the loop
            client_socket.send("closed".encode("utf-8"))
            break

        # Check if the received message is a valid student ID
        student_id = validate_student_id(request)
        check_id(student_id, client_socket)

        # convert string to bytes and send accept response to the client
        response = "accepted".encode("utf-8")
        client_socket.send(response)

    client_socket.close()
    print("Connection to the client closed")
    server.close()
```

Figure (11): function to run the Server

Error Handling

Error handling was implemented using try and except upon calling the functions. As shown in the following figures.

```python
try:
    run_client()
except Exception as e:
    print(f"Error in : {e}")
```

```python
try:
    run_server()
except Exception as e:
    print(f"Error in : {e}")
```

Figure (12): Error Catching when calling scripts

Running Example

The following Figures shows the logic used to validate the student ID's and to lock the screen. These functions were defined in a script called utils.py and called in the server side. These two command were run in different shells.

```
python server.py
```

```
python client.py
```

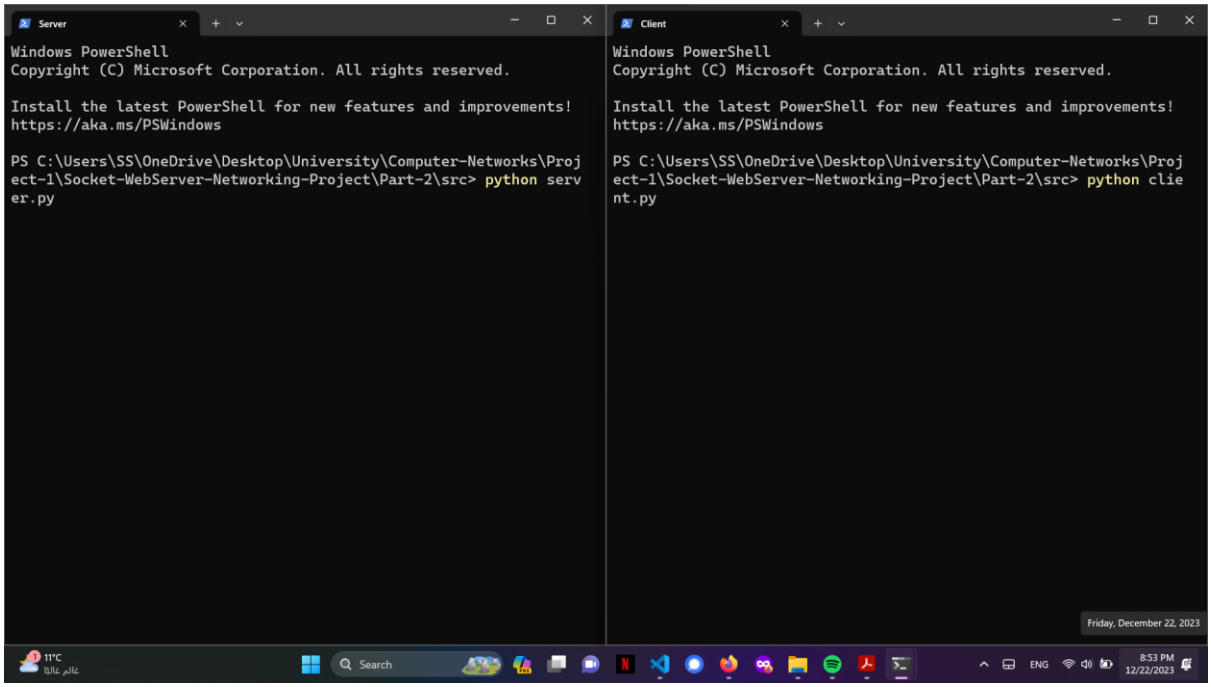

Figure (13): Shell Script Run python

In this figure, the output of running the server and then client is shown. Where the server is listening and the client received a message to enter a student ID.

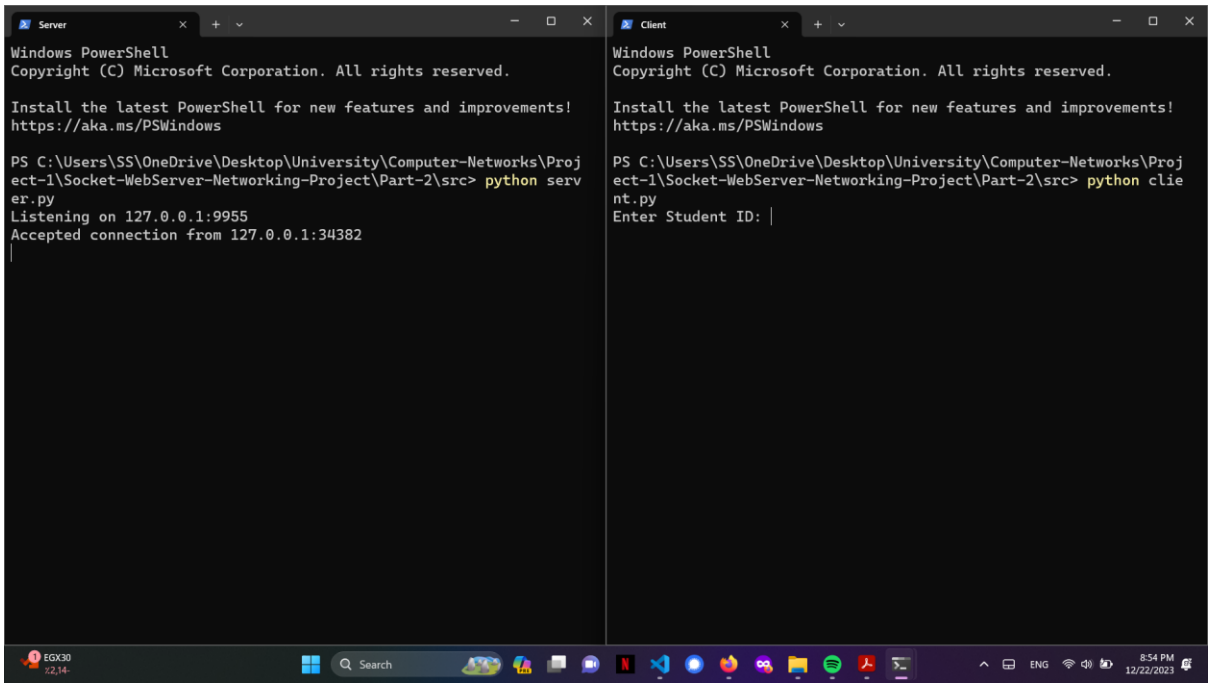

Figure (14): Shell Script Run python

Following will show different input and responses from the server:

Non-Valid (by ids list) ID: 1201201

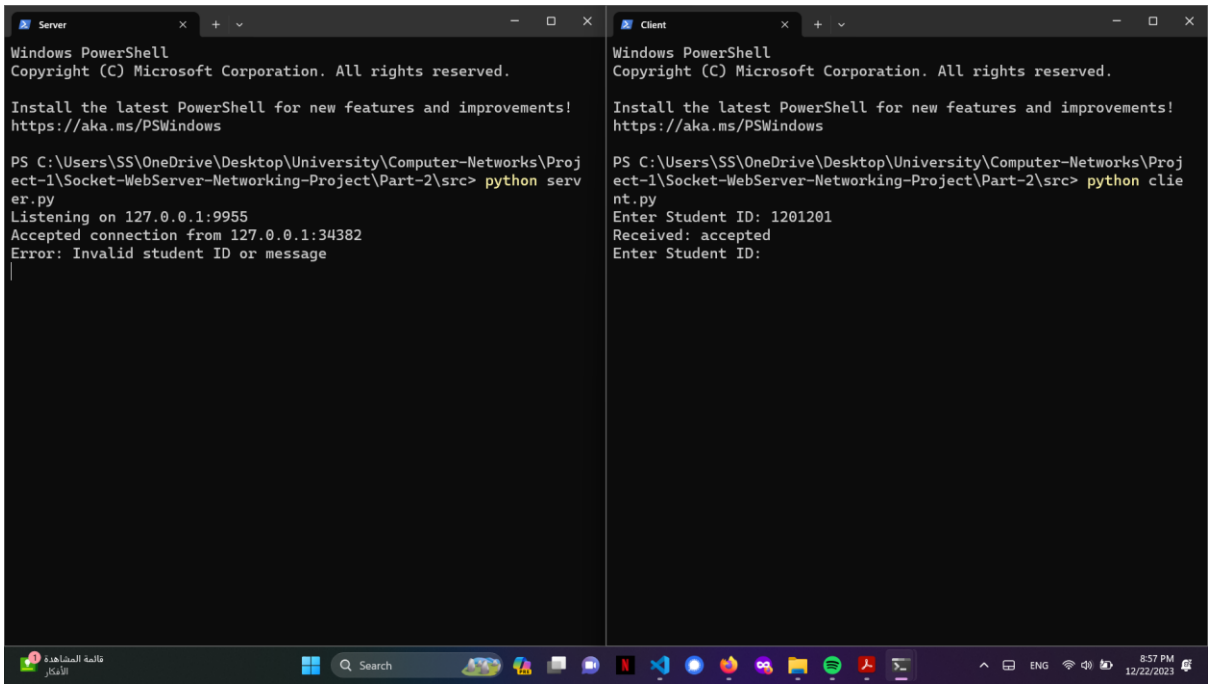

Figure (15): Error Invalid ID

Non-Valid ID (as string).

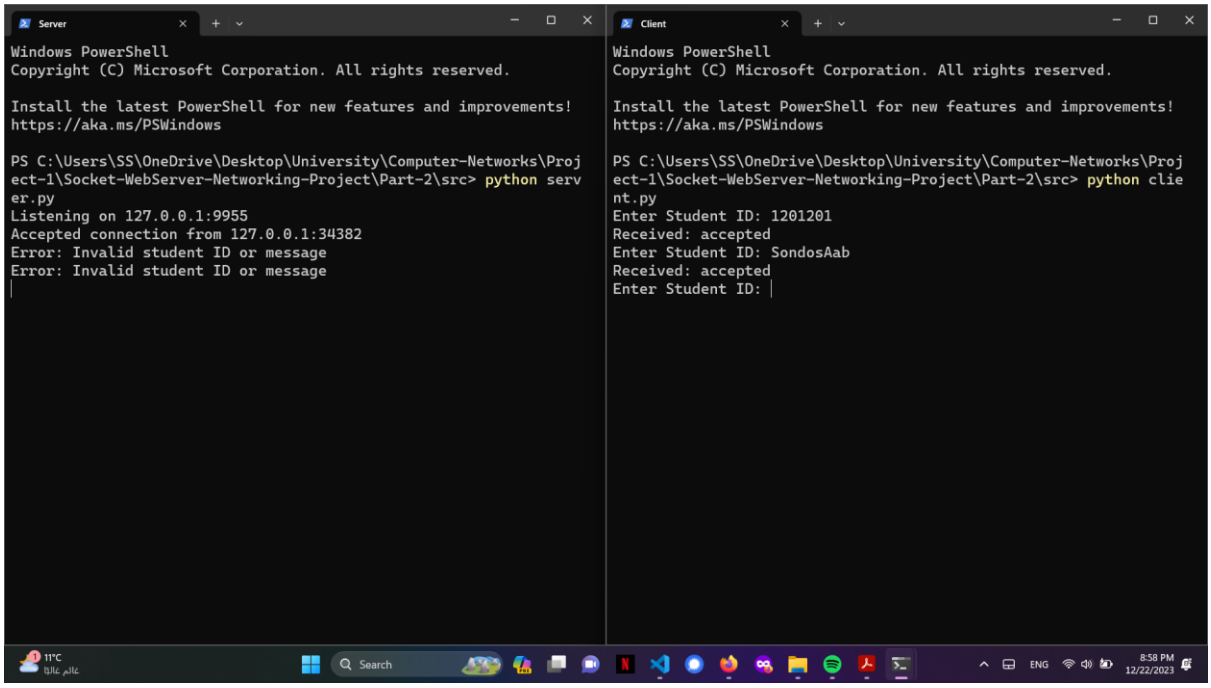

Figure (16): Error Invalid ID

Valid ID (1190652) the screen was locked after 10 seconds.

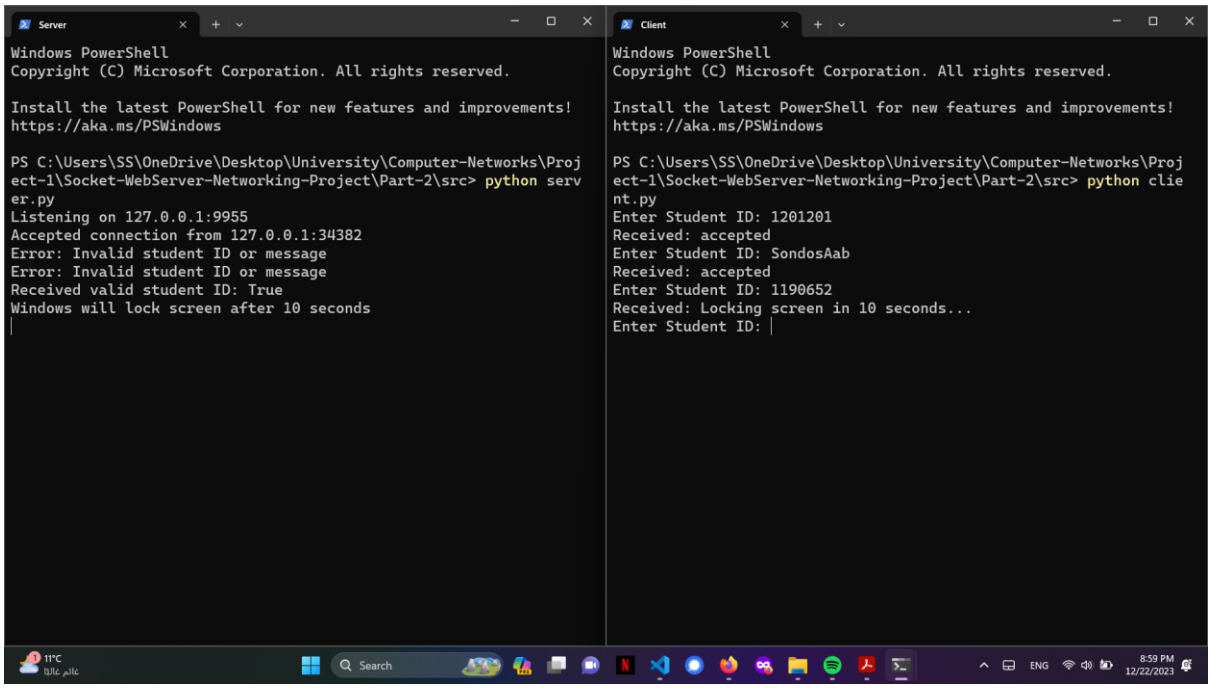

Figure (17): Error Invalid ID

Additional Idea

The following Figure shows a form (index.html) that takes the id as an input field and then send it to the server and then validate it. Given the time it was not used.

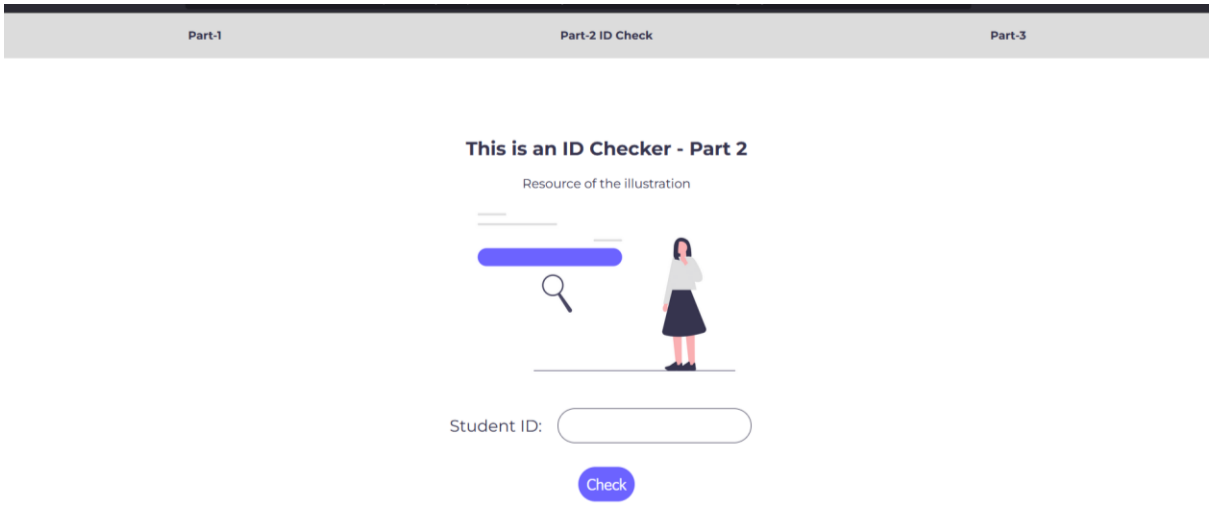

Figure (18): ID submission form

Part3: Tiny Web Server Implementation

In this part, a tiny web server is built on interacting with a portfolio of the writer's. Different content types are handled, 404 not found is also handled. Finally, testing is performed.

Content Type Definition

3.7 ***Media Types***

HTTP uses Internet Media Types [17] in the Content-Type (section 14.17) and Accept (section 14.1) header fields in order to provide open and extensible data typing and type negotiation.

- media-type = type "/" subtype *(";" parameter)
- type = token
- subtype = token

The type, subtype, and parameter attribute names are case- insensitive. Parameter values might or might not be case-sensitive, depending on the semantics of the parameter name. Linear white space (LWS) MUST NOT be used between the type and subtype, nor between an attribute and its value. The presence or absence of a parameter might be significant to the processing of a media-type, depending on its definition within the media type registry.

14.17 ***Content-Type***

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

Content-Type = "Content-Type" ":" media-type

Resource: RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1

Content Types

The web server is designed to dynamically respond to different file requests, ensuring proper content delivery based on file types. The following Function is used to ensure that the server adeptly determines the appropriate content type and serves the requested content accordingly.

- If the request is an `.html` file then the server should send the requested html file with Content-Type: text/html, any html file.
- If the request is a `.css` file then the server should send the requested css file with Content-Type: text/css, any CSS file
- If the request is a `.png` then the server should send the png image with Content-Type: image/png, any jpeg image.
- If the request is a `.jpg` then the server should send the jpg image with Content-Type: image/jpeg, jpeg any image.

```python
def get_content_type(file_path):
    """
        Args:
            Takes file_path of the desired file as a string
        Returns:
            The content Type of the file based on the extenstion
    """
    if file_path.endswith('.html'):
        return 'text/html; charset=utf-8'
    elif file_path.endswith('.css'):
        return 'text/css; charset=utf-8'
    elif file_path.endswith('.png'):
        return 'image/png'
    elif file_path.endswith('.jpg'):
        return 'image/jpeg'
    else:
        return 'text/html; charset=utf-8'
```

Figure (21): function to get the content type

HTML Pages

Four HTML pages were used in this section. Contains the requirements.

- 404.html In Arabic and In English
- Index.html contains requirements of html codes and portfolio.

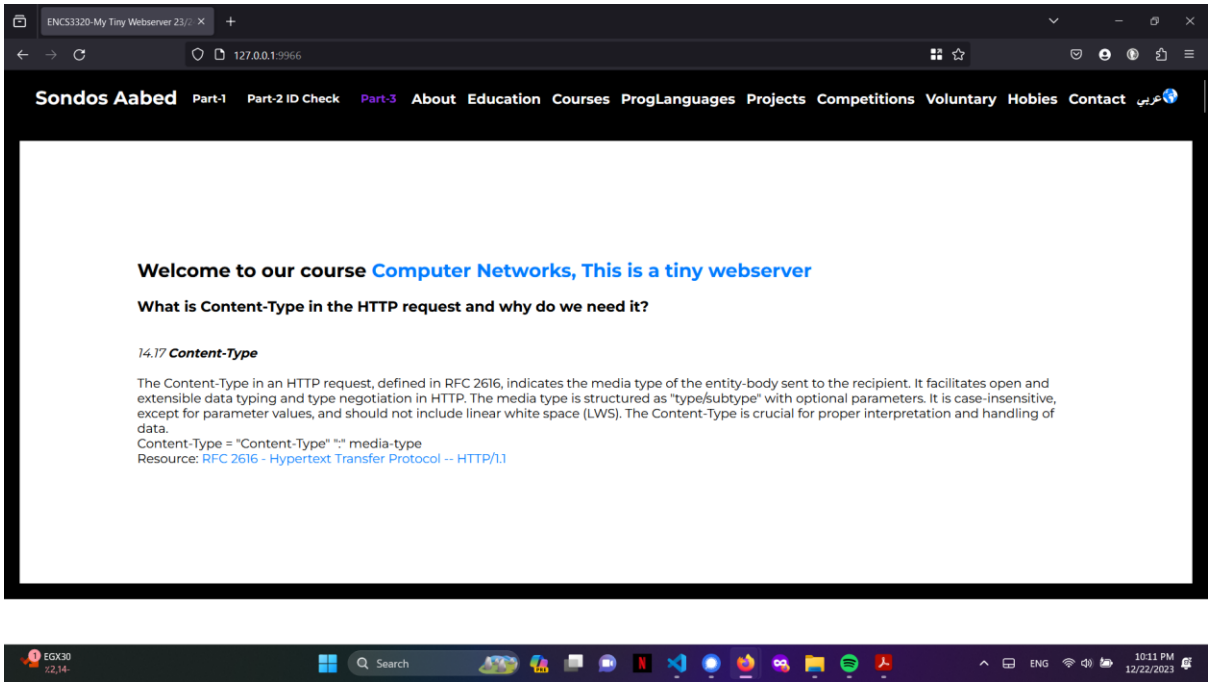

Figure (19): index.html pages ar&en

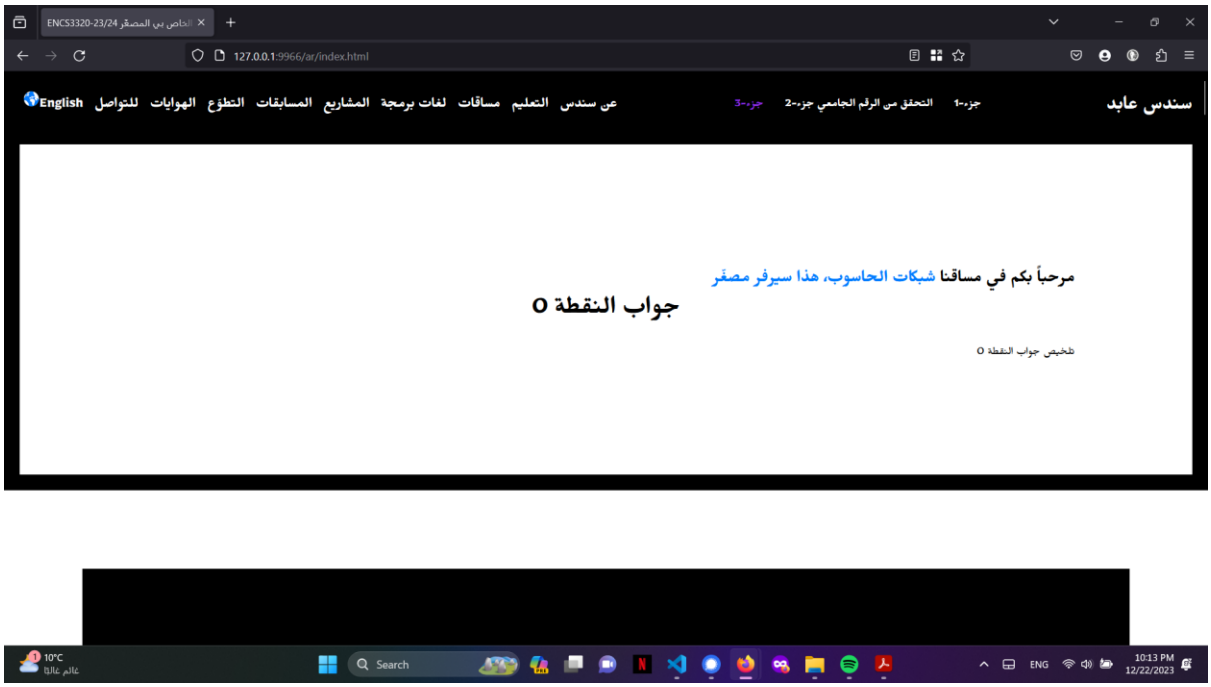

Figure (20): index.html pages ar

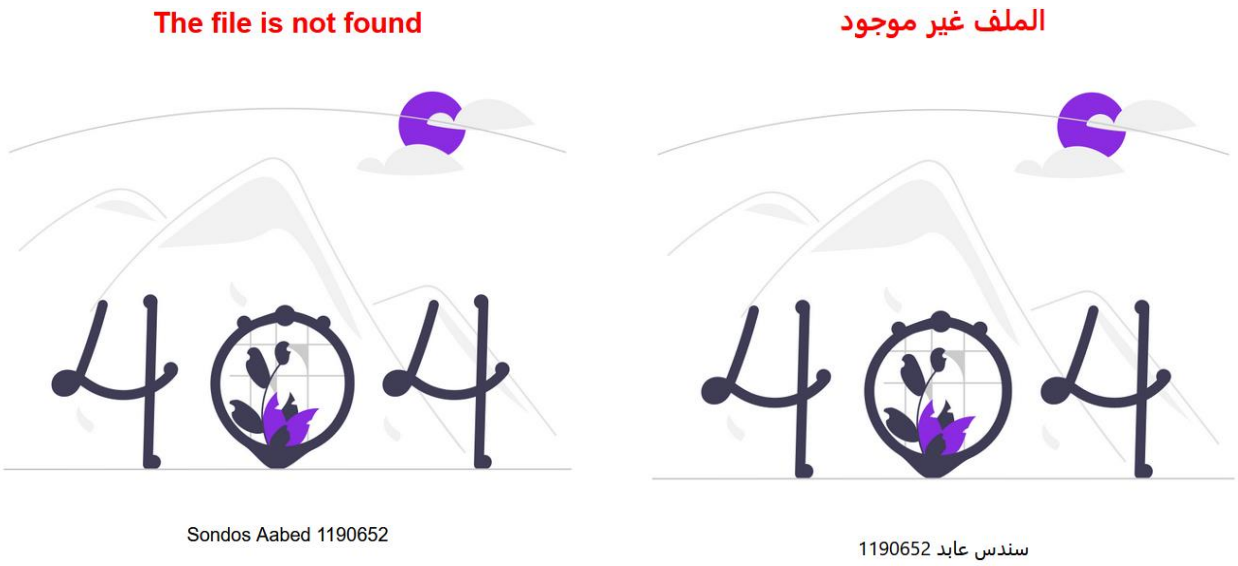

Figure (21): 404.html pages Arabic & English

Server & Client Functionality

Once the main.py script is run by python script the start_server function is called and it will enter a loop of calling and handling the client until they leave. As shown below:

```
from server import start_server

if __name__ == "__main__":
    """
        Checks if this is the main and calls the start_server funtion on it's deafult args
    """
    start_server()
```

Figure (22): main.py script

```
def start_server(localhost='127.0.0.1', port=9966):
    """
        Handles the client until they leave
        Args:
            takes the local host ip
            takes the port number as required
    """
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((localhost, port))
    server_socket.listen(1)
    print("Server listening on port 9966...")

    while True:
        handle_client(server_socket)
```

Figure (23): Start the Server Function

The program prints the HTTP requests on the terminal window once handled. The following function handles the client socket.

```
def handle_client(server_socket):
    """
        To take requests from the users
        Args:
            Server_socket
    """
    client_socket, addr = server_socket.accept()
    request = client_socket.recv(1024).decode('utf-8')

    print(f"Received request from {addr}:\n{request}")# Print HTTP request on the terminal

    request_parts = request.split(' ')# Extract the requested path
    if len(request_parts) >= 2:
        request_path = request_parts[1]
        handle_request(client_socket, request_path) # Handle the request
```

Figure (24): Handle the Client Function

Responses & Requests Handling

To send responses of the server a function is created to handle the responses to be send to a client socket.

```
def send_response(client_socket, status_code, content_type, data):
    """
        To sent a header & data to a client
        Args:
            client_socket, status_code, content_type, data
    """
    response_headers = f"HTTP/1.1 {status_code}\r\nContent-Type: {content_type}\r\n\r\n"
    response_data = response_headers.encode('utf-8') + data
    client_socket.send(response_data)
```

Figure (25): Send Response Function

As for requests requirements, if the user types in the browser something like http://localhost:9966/ar or http://localhost:9966/en The program checks and response.

If the request is / or /index.html or /main_en.html or /en (for example localhost:9966/ or localhost:9966/en) then the server send main_en.html file with Content-Type: text/html.

If the request is /ar then the server response with main_ar.html which is an Arabic version of main_en.html

```
def handle_request(client_socket, request_path):
    """
        Args:
            client_socket, request_path takes and client socket to be send to and a request path to render
            handles all cases of temrory redirection & file doesn't exists
    """
    if request_path in ['/', '/index.html', '/main_en.html', '/en']:
        file_path = 'en/index.html'
        lang = 'en'
    elif request_path in ['/main_ar.html', '/ar']:
        file_path = 'ar/index.html'
        lang = 'ar'
    elif request_path in ['/cr', '/so', '/rt']:
        status_code= temp_redirection(request_path, client_socket)
        return
    else:
        file_path = request_path.lstrip('/')
        lang = 'en' if file_path.startswith('en/') else 'ar'
    try:
        with open(file_path, 'rb') as file:
            data = file.read()
            content_type = get_content_type(file_path)
            status_code = '200 OK'
            send_response(client_socket, status_code, content_type, data)
    except FileNotFoundError:
        handle_404(client_socket, lang)
```

Figure (26): Handle Response Function

Temporary Redirection

The web server is designed to return a temporary redirection 307 when these conditions occur in a request path:

- If the request is /cr then redirect to cornell.edu website
- If the request is /rt then redirect to ritaj website
- If the request is /so then redirect to stackoverflow.com website

```
def temp_redirection(request_path, client_socket):
    """
        Args:
            Request_path, Client_socket
        Returns:
            Status Code as Temor. Redirection as 307
    """
    redirect_to = {'/cr': 'http://cornell.edu',
                   '/so': 'http://stackoverflow.com',
                   '/rt': 'http://ritaj'}
    response_data = f"Redirecting to {redirect_to[request_path]}".encode('utf-8')
    client_socket.send(response_data)
    client_socket.close()
    return '307 Temporary Redirect'
```

Figure (27): function to Temporary Redirection

File Doesn’t Exist

If the request is wrong or the file doesn’t exist the server should return a simple HTML webpage that contains (Content-Type: text/html)

- “HTTP/1.1 404 Not Found” in the response status
- The IP and port number of the client

```
def handle_404(client_socket, lang='en'):
    """
        Args:
            Client socket to send the response
            lang: if ar will sent the ar/404.html
                or else will sent the english and the en is default
    """
    custom_404_path = f'{lang}/404.html'
    with open(custom_404_path, 'r', encoding='utf-8') as custom_404_file:
        data = custom_404_file.read().encode('utf-8')
        content_type = get_content_type(custom_404_path)
        status_code = '404 Not Found'
        send_response(client_socket, status_code, content_type, data)
```

Figure (28): function to handle Not Found

Testing

In this section, test cases are generated and run using the library Unittest in python.

```python
def test_existing_html_file():
    request = "GET /index.html HTTP/1.1\r\nHost: localhost\r\n\r\n"
    expected_output = b"HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=utf-8\r\n\r\nContent of 'en/index.html'"
    run_test(request, expected_output)
```

Pythor

```python
def test_nonexistent_file():
    request = "GET /nonexistent.html HTTP/1.1\r\nHost: localhost\r\n\r\n"
    expected_output = b"HTTP/1.1 404 Not Found\r\nContent-Type: text/html; charset=utf-8\r\n\r\nContent of 'en/404.html'"
    run_test(request, expected_output)
```

Pythor

```python
def test_temporary_redirection():
    request = "GET /cr HTTP/1.1\r\nHost: localhost\r\n\r\n"
    expected_output = b"HTTP/1.1 307 Temporary Redirect\r\nContent-Type: text/html; charset=utf-8\r\n\r\nRedirecting to http:
    run_test(request, expected_output)
```

Pythor

```python
def test_css_file():
    request = "GET /style.css HTTP/1.1\r\nHost: localhost\r\n\r\n"
    expected_output = b"HTTP/1.1 200 OK\r\nContent-Type: text/css; charset=utf-8\r\n\r\nContent of 'style.css'"
    run_test(request, expected_output)
```

Figure (29): Test Cases

When running the test cases the output was Ran 3 test and returned OK that indicates all test cases has been asserted to be true. The following figure shows the output:

```
PS C:\Users\SS\OneDrive\Desktop\University\Computer-Networks\Project-1\Socket-WebServer-Networking-Project> & C:/Users/SS/AppData/Local/Programs/Python/Python311/python.exe c:/Users/SS/OneDrive/Desktop/University/Computer-Networks/Project-1/Socket-WebServer-Networking-Project/Part-3/src/test_server.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.068s

OK
```

Figure (30): Test Cases

The following are examples of running the main.py script:

First let’s Test Path en/ main_en.html, the following figure is it’s output:

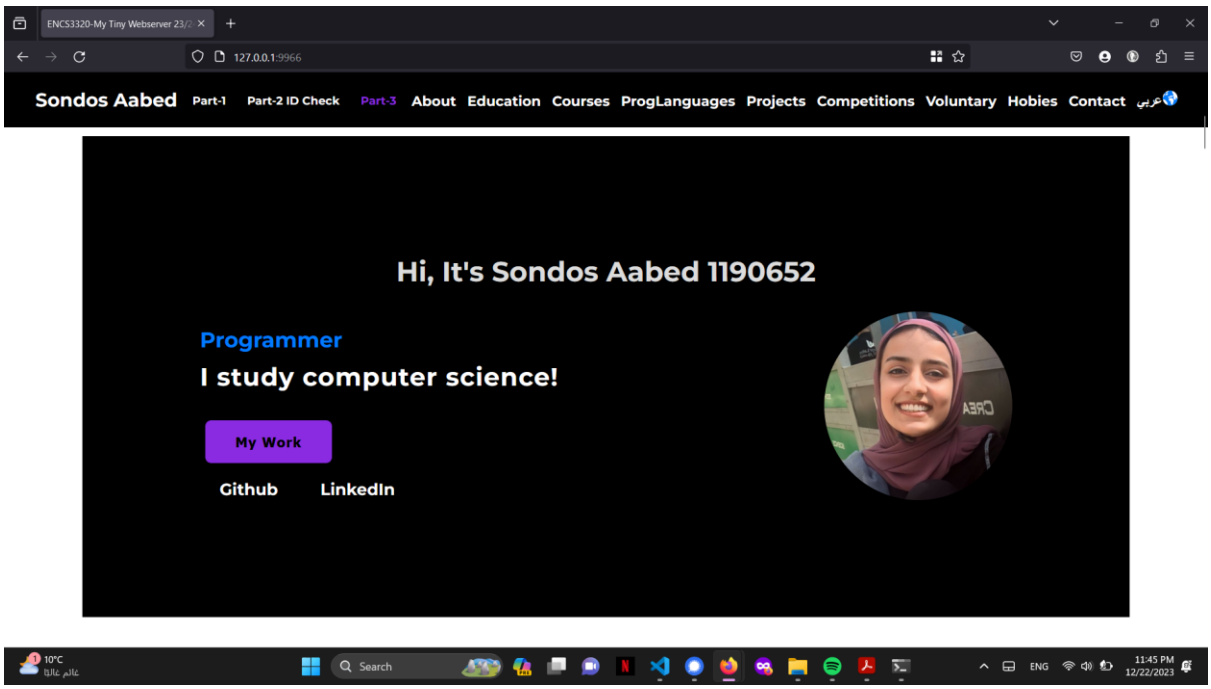

Figure (30): main_en.html

Now the ar/inedx.html route is redirecting to the arabic page:

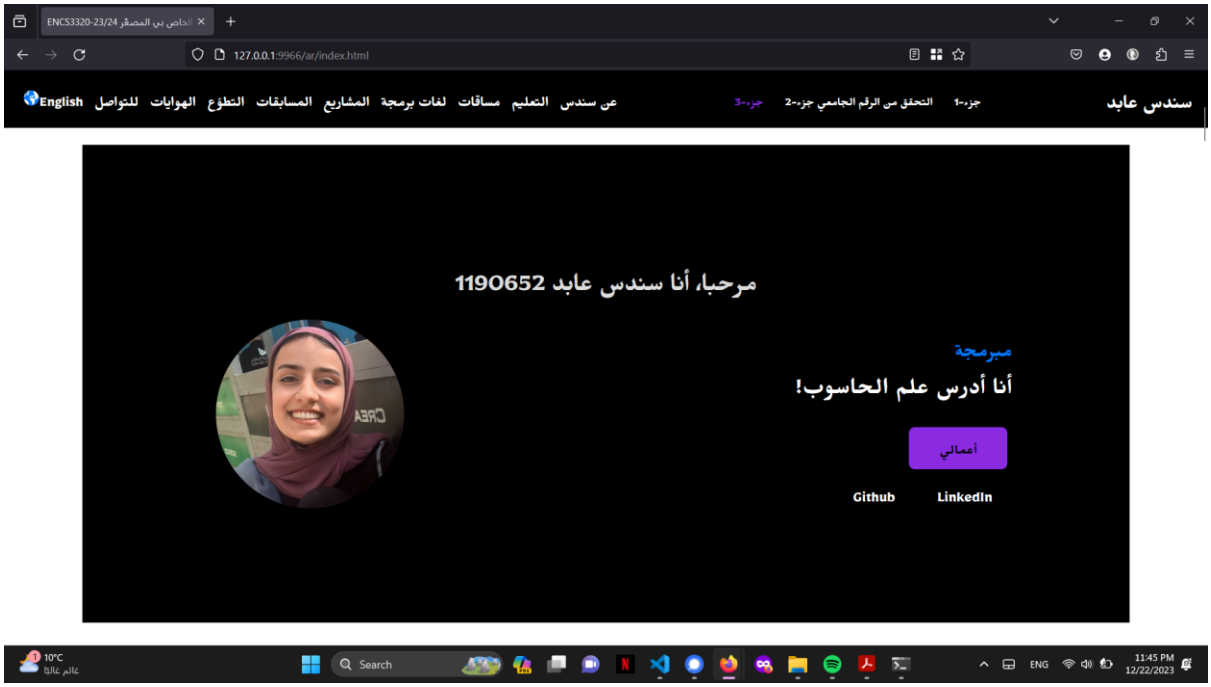

Figure (31): index.html

Now to test the redirecting to a non existing file, the language is set by default to english but if it was an arabic one it is redirecting to the arabic:

Test File Doesn’t exist in Arabic:

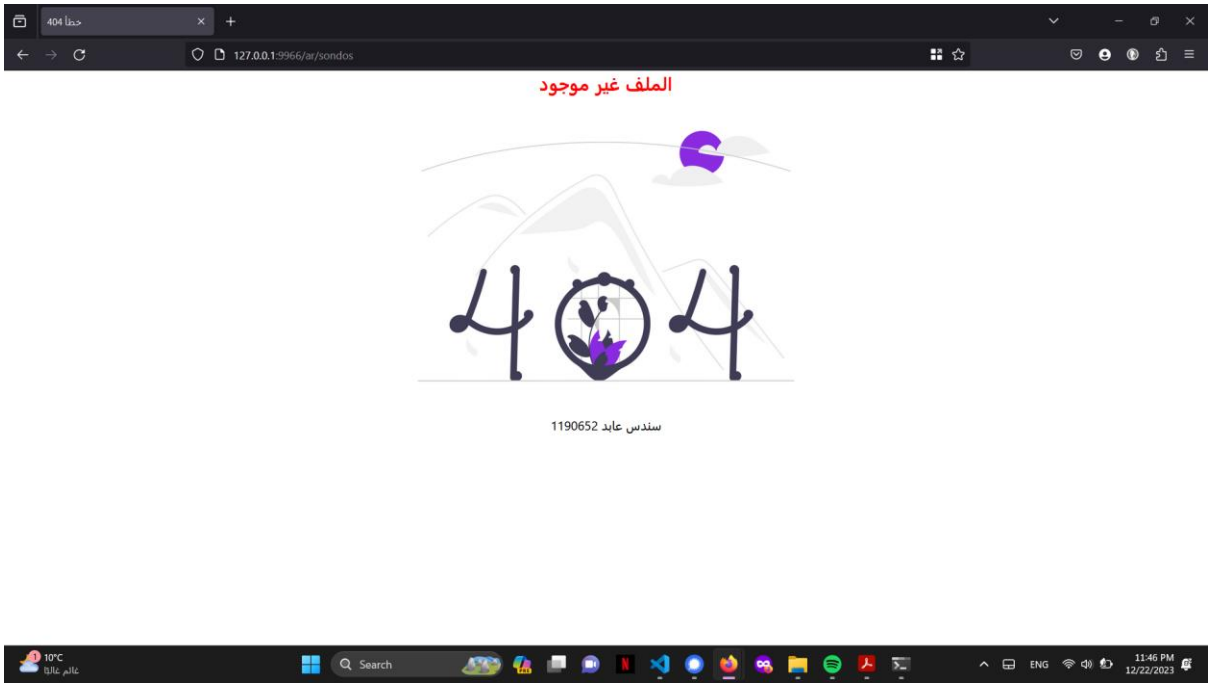

Figure (31): 404 arabic response

Test File doesn’t exist in English:

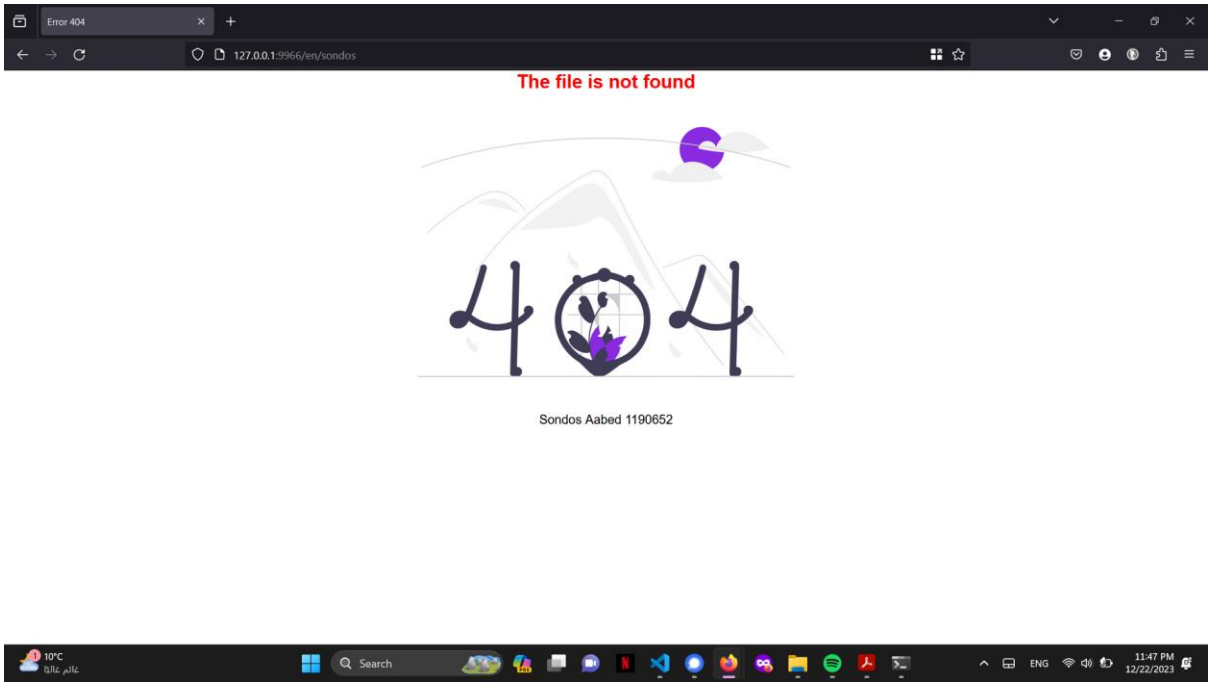

Figure (32): 404 english response

While running the the server, the following figures shows how it’s printed on the terminal:

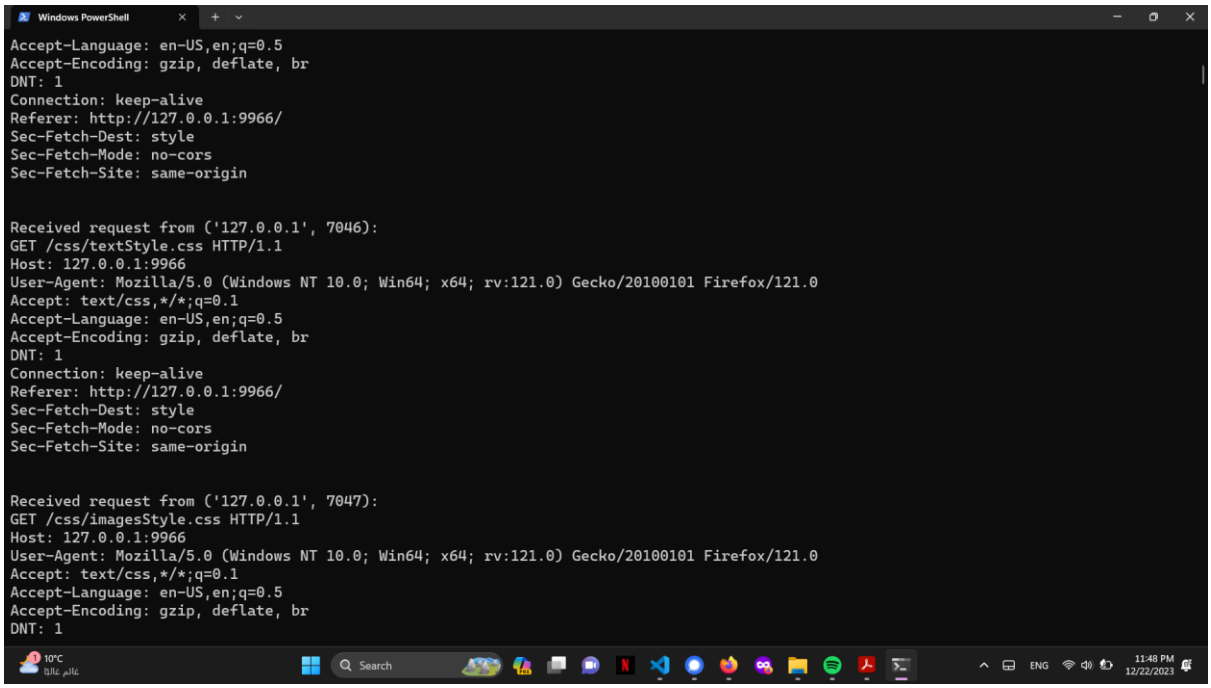

Figure (33): Outputs on terminal

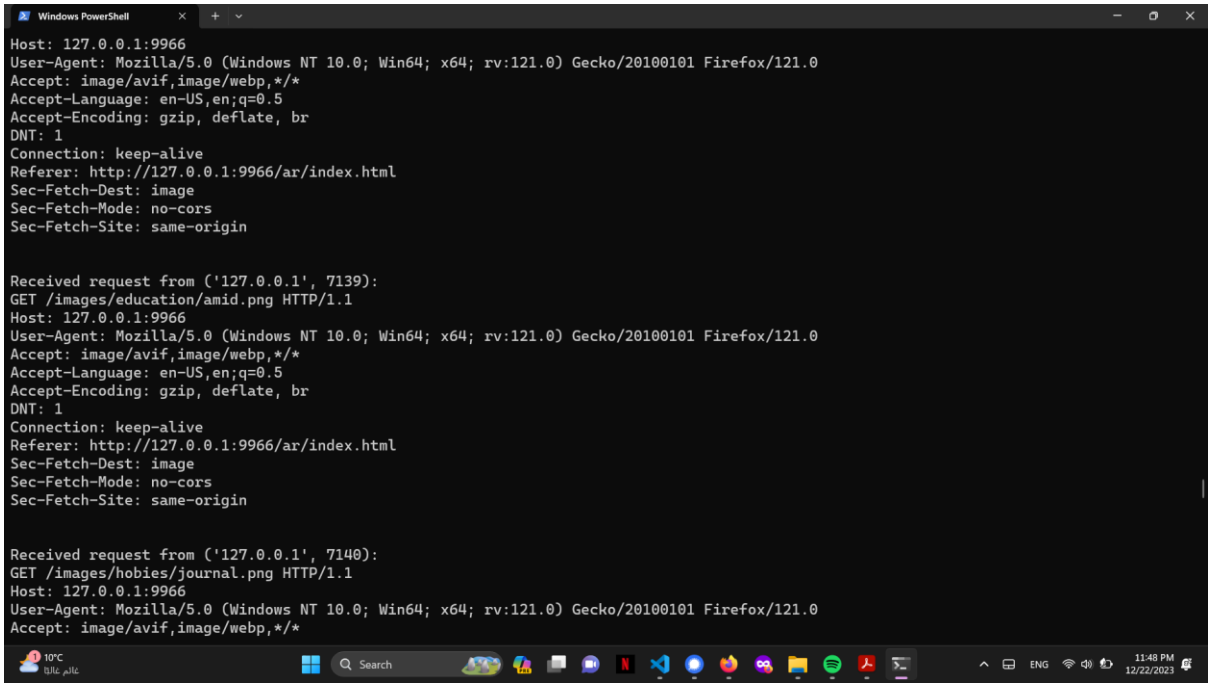

Figure (34): Outputs on terminal

Summary

This report has presented an exploration project of the computer networking principles, focusing on socket programming and web servers. Beginning with fundamental network commands and proceeded to practically implement TCP client and server application with ID validator utility and lock screen functionality were used.

The development of a tiny web server, adhering to RFC2616 standards, demonstrated dynamic responses to diverse HTTP requests. Throughout the report, the organization of the source code was represented, utilizing tools like Git and Visual Studio Code and Python Socket Library, for efficient development.

Resources

Used for illustrations Illustrations | unDraw

Used ro learn content types RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1

Used to learn socket programming: A Complete Guide to Socket Programming in Python | DataCamp

Appendices

Find the code in this Repository:

GitHub - sondosaabed/Socket-WebServer-Networking-Project: TCP and a simple web server. Implementations and documentation with socket programming concepts