# COMP2322 Computer Networking Project

# Multi-thread Web Server

Jyotsna Venkatesan 22108825D

## Introduction

This paper describes a simple HTTP server built using Python, capable of handling basic web requests like GET and HEAD. The server uses Python's standard libraries to manage network connections, threads, and file interactions. Key functionalities include evaluating if files have been modified since the last request, supporting persistent connections with 'keep-alive' for efficiency, and detailed logging of each client request for monitoring purposes. The server is flexible, allowing the base directory for files to be set at startup through command-line arguments. This setup is ideal for educational purposes, providing a clear example of how a basic web server operates and how it can be implemented using Python.

## Design

This program is built using Python and utilizes several of its built-in libraries to handle network communications, threading, file operations, and date/time processing. Below are some of the key features of the program:

**Server Configuration and Multi-threading**

The server starts by creating a socket using Python's socket library, which listens for incoming TCP connections on a specified host and port. When a client connects, the server accepts the connection and spawns a new thread to handle the client's request. This multi-threaded approach allows the server to handle multiple clients simultaneously, improving the server's ability to scale with increasing client requests.

**Handling Client Requests**

A respective thread is assigned to handle each client connection, through the handle_client_connection function. This function continuously listens for incoming data from the client, interpreting it as HTTP requests. The primary tasks involved in handling the requests are as follows:

1) Request Parsing: The server reads the incoming data and divides it into lines. The first line, known as the request line, contains the HTTP method, the requested resource, and the HTTP version. Subsequent lines are processed to extract HTTP headers using the parse_headers function, which stores header values in a dictionary.
2) Conditional GET Processing: The server examines the If-Modified-Since header to determine whether the requested file has been modified since the date specified in the header. This functionality minimizes unnecessary data transfer by enabling the server to issue a 304 Not Modified response if the file remains unchanged.

**Generating Responses**

The server generates responses based on the analysis of the request:

1) Checking File Status and Details: The generate_response_headers function verifies the existence of the requested file and gathers necessary metadata like file size and last modification date. It then decides the proper HTTP status code and headers based on the file's modified status relative to the If-Modified-Since header.
2) Delivering Content: For GET requests that require a file transfer, either because the file has been modified or no modification date was specified, the server loads and appends the file content to the prepared headers before dispatching the complete response to the client.

**Managing Connections and Logging Activities**

The server maintains or closes connections based on the client's Connection header. Persistent connections are supported if the client specifies Connection: keep-alive. Each transaction is also logged through the log_request function, aiding in operational monitoring and troubleshooting.

## Implementation

Below is a description with the working of the functions included in the program:

1) **parse_headers(request_lines)**

Parameters:

- request_lines: A list of request lines containing the HTTP headers from an HTTP request.

Description:

The parse_headers function extracts HTTP headers from the provided request lines. It iterates through the lines, splitting them at ": " to separate the header key and value. The resulting key-value pairs are stored in a dictionary, which is then returned.

2) **generate_response_headers(file_path, file_exists, if_modified_since=None, keep_alive=False)**

Parameter:

- file_path: Path to the file for which the HTTP response headers are generated.
- file_exists: Boolean indicating whether the file exists.
- if_modified_since: Datetime representing the value of the "If-Modified-Since" header.
- keep_alive: Boolean indicating whether the connection should be kept alive.

Description:

The generate_response_headers function generates HTTP response headers based on file existence, modification time, and connection type. It determines the appropriate response code and headers, including content type, length, and last modified timestamp. The generated response headers are returned as a string.

3) **log_request(client_ip, timestamp, request_line, response_type)**

Parameters:

- client_ip: IP address of the client making the request.
- timestamp: Timestamp of the request.
- request_line: Request line containing the HTTP method, requested resource, and HTTP version.
- response_type: Type of response generated for the request.

Description:

The log_request function logs the details of an HTTP request and its corresponding response. It creates a log entry string by formatting the client IP, timestamp, request line, and response type. The log entry is then appended to the "server.log" file using the "a" mode for appending.

4) **def handle_client_connection(client_socket, client_address)**

Parameters:

- client_socket: Socket object representing the client connection.
- client_address: Tuple containing the client IP address and port number.

Description:

The handle_client_connection function handles the communication with a client. It receives the client request, parses the headers, generates the appropriate response, sends it back to the client, and logs the request. The function supports keeping the connection alive based on the "Connection" header.

5) **start_server(host='localhost', port=8080, base_dir=None)**

Parameters:

- host: Hostname or IP address where the server should bind. Default is 'localhost'.

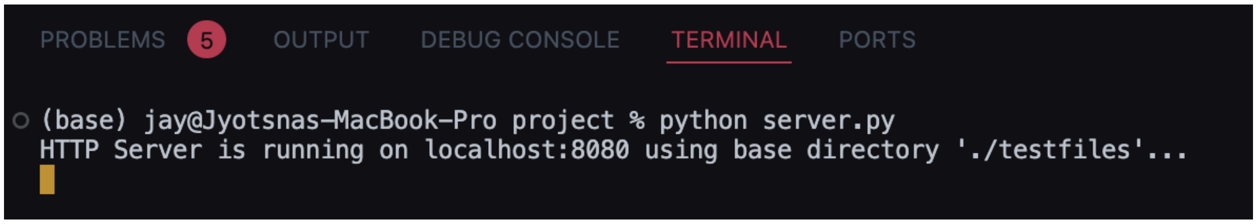
- port: Port number on which the server should listen. Default is 8080.
- base_dir: Base directory from which the server should serve files. Default is None.

Description:

The start_server function initializes and starts an HTTP server, binding it to the specified host and port. It accepts incoming client connections, creates threads to handle each connection, and starts them.

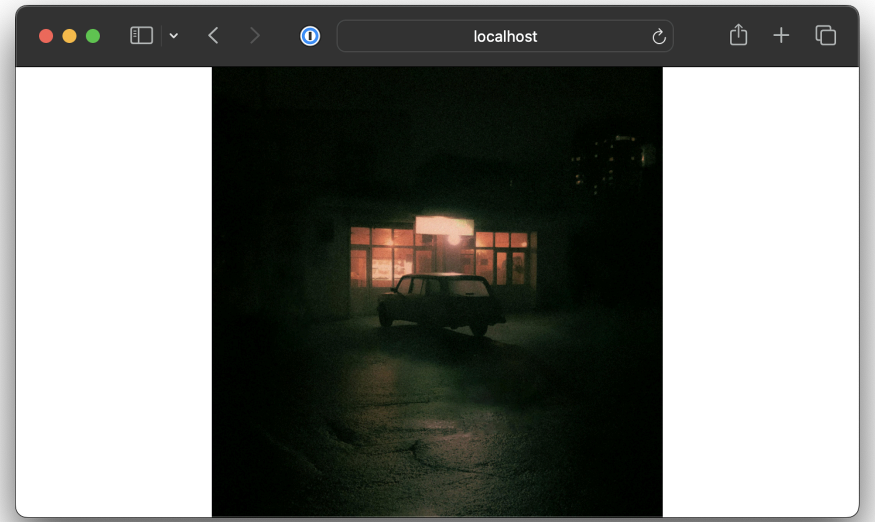
## Demonstration

**Running server.py:**

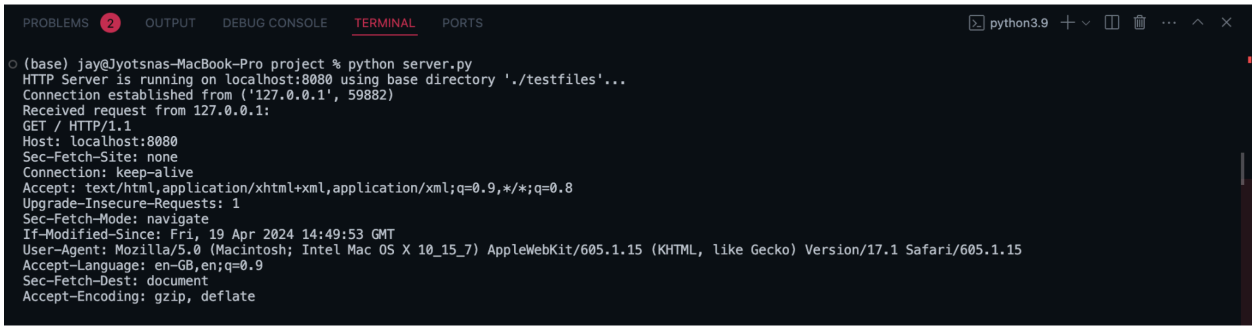


```
(base) jay@Jyotsnas-MacBook-Pro project % python server.py
HTTP Server is running on localhost:8080 using base directory './testfiles'...
```

**Open the website on a browser:**

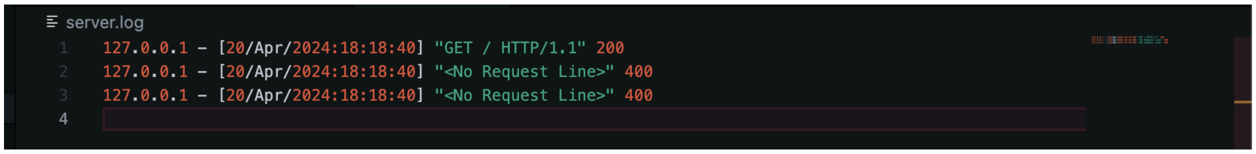


## Output by the program:



```
(base) jay@Jyotsnas-MacBook-Pro project % python server.py
HTTP Server is running on localhost:8080 using base directory './testfiles'...
Connection established from ('127.0.0.1', 59882)
Received request from 127.0.0.1:
GET / HTTP/1.1
Host: localhost:8080
Sec-Fetch-Site: none
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Upgrade-Insecure-Requests: 1
Sec-Fetch-Mode: navigate
If-Modified-Since: Fri, 19 Apr 2024 14:49:53 GMT
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/17.1 Safari/605.1.15
Accept-Language: en-GB,en;q=0.9
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
```

## Log File:



```
127.0.0.1 - [20/Apr/2024:18:18:40] "GET / HTTP/1.1" 200
127.0.0.1 - [20/Apr/2024:18:18:40] "<No Request Line>" 400
127.0.0.1 - [20/Apr/2024:18:18:40] "<No Request Line>" 400
```