

Arvin Bayat Manesh - 301454846

Hieu Cong Tran - 301403714

## Step Two:

### A)

To test if our web server is running correctly, we run the program, and type <http://127.0.0.1:8080/test.html> in our web browser. After requesting the test.html we get following response, “Congratulations! Your Web Server is Working!”, which indicates that the web server is successfully running.

### B)

To test our implementation of the status codes we use curl to send customized HTTP request to determine the behaviour of the web server in response to different types of requests.

## 1. 200 OK

To test 200 OK status code we need to request the test.html webpage using curl.

After running the program we type in the following command line in the terminal: curl -i -X GET <http://127.0.0.1:8080/test.html>, we get the following response:

**HTTP/1.1 200 OK**

Content-Type: text/html

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title></title>

<meta name="author" content="">

<meta name="description" content="">

<meta name="viewport" content="width=device-width, initial-scale=1">

```
</head>
<body>
  <p>Congratulations! Your Web Server is Working!</p>
</body>
</html>
```

which indicates that the 200 OK status code is working.

## 2. 304 Not Modified

To test the 304 Not Modified status code we run the following command line in the terminal: `curl -X GET -I -H "If-Modified-Since: Tue, 04 Apr 2023 15:30:00 GMT" http://127.0.0.1:8080/test.html`, to determine if test.html has been modified since Tue, 04 Apr 2023 15:30:00 GMT.

The output is:

**HTTP/1.1 304 Not Modified,**

which determines that the 304 Not Modified is working correctly.

## 3. 400 Bad Request

To test the 400 Bad Request, we would need to send an HTTP request with an unrecognizable method which differs from the allowed methods such as GET or PUT. Therefore, we would need to run the following command line: `curl -X INVALID_METHOD http://127.0.0.1:8080/invalid\_path`

The output is:

**400 Bad Request,**

which indicates that the 400 Bad Request status code is working correctly.

#### **4. 403 Forbidden**

There might be unauthorized files in the web server which a certain user is not allowed to access. The 403 Forbidden status code could block an access to a certain webpage based on the clients IP address, or it could block access to a certain webpage to all the user for a certain amount of time.

In our example, we created an html file called unauthorized.html, if we request this file we must get a 403 Forbidden status code. To test this, we run the following command line: `curl -X GET http://127.0.0.1:8080/unauthorized.html`

We get the following output:

#### **403 Forbidden**

This means that access to the webpage unauthorized.html is blocked successfully.

#### **5. 404 Not Found**

This status code indicates that the requested webpage does not exist on the web server. To test this status code we use the following command line to request a webpage which does not exist on our web server: `curl -X GET http://127.0.0.1:8080/nonexistent.html`

The output is:

#### **404 Not Found**

This status code is also working correctly.

#### **6. 411 Length Required**

When our HTTP request method is PUT we need to know the length of the body of the request; thus, we need to check for the existence of the Content-Length field in the header of the HTTP request. To test this status code we would use the following command which does not specify the length of its body: `curl -X POST http://127.0.0.1:8080/submit\_form`

The output is:

**411 Length Required,**

which indicates that this status code is working as well.

### **Step Three:**

#### **A)**

The request handling in a proxy server differs significantly from that in a web server. While both are involved in processing HTTP requests, their roles and functionalities are distinct. For a web server, it directly responds to client requests for resources like HTML pages, images, scripts, etc. where it serves content that it hosts, and stores and manages the files and resources that are requested by the clients. While A proxy server acts as an intermediary between a client and other servers, including web servers. It forwards requests from clients to the intended servers and relays responses back to the clients. A proxy server uses web-caching. Meaning that when a resource is requested, the proxy might serve it from its cache if it's available and fresh, reducing the load on the origin server and improving response times.

Web caching: We would implement this in handling “GET” requests in a proxy server. The process begins with the proxy server checking its local cache to determine if the requested URL's content is already stored. If the content is found and deemed fresh according to the server's freshness criteria, the proxy server directly returns this cached content to the client without the need to connect to the original server. However, if the content is found but considered stale, the proxy server sends a conditional GET request to the origin server, including an If-Modified-Since header with the last modified timestamp of the cached content.

In situations where the content is not in the cache, the proxy server makes a regular GET request to the origin server. The server's response then dictates the subsequent action: a “200 OK” response indicates either new or updated content, prompting the proxy server to update its cache with this content and its last modified timestamp, before serving it to the client. While “a 304

Not Modified” response is sent by the origin server if the content has not been modified since the provided timestamp in the If-Modified-Since header. This response allows the proxy server to serve the content directly from its cache, confirming that the cached version remains valid and up to date.

**B)**

For testing it is necessary to be able to configure specific HTTP headers, especially “Last-Modified” because that is what we are using for web caching. We would do this by create a local Flask server where can configure it to send specific HTTP headers, which are essential for testing caching behaviour Basically, what the Flask server doing is that the content served by the local Flask server remains unchanged unless we modify it. This stability is necessary to test for 304 “304 Not Modified” responses, as the server needs to indicate that the content hasn't changed since the last request.

Now we have Flask server ready to go; we would then start with our test.

First, we issue the “curl” command:

```
“curl --http0.9 -v -x http://127.0.0.1:8080 http://127.0.0.1:8000/test.html --output  
first_request.html”
```

We would get:

```

C:\Users\thang>curl --http0.9 -v -x http://127.0.0.1:8080 http://127.0.0.1:8080/test.html --output first_request.html
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left     Speed

  0     0    0     0    0     0      0      0      0  0*    Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
> GET http://127.0.0.1:8080/test.html HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.4.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
{ [16 bytes data]
100 308  0 308  0  7823  0 --:--:-- --:--:-- --:--:-- 7897
* Closing connection

C:\Users\thang>curl --http0.9 -v -x http://127.0.0.1:8080 http://127.0.0.1:8080/test.html --output second_request.html
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left     Speed

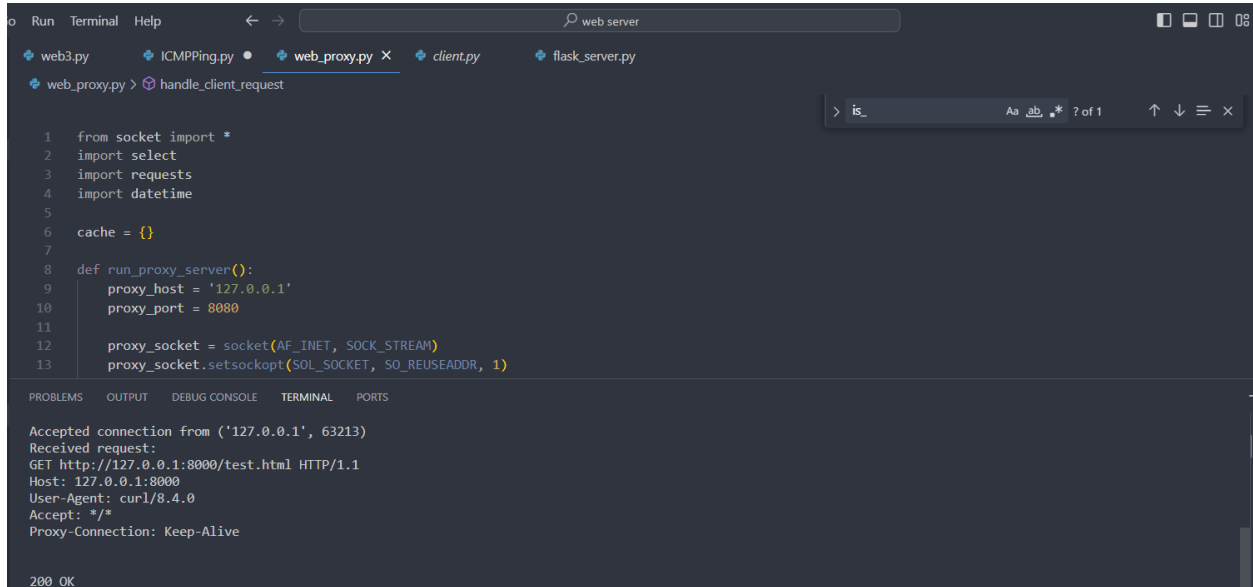
  0     0    0     0    0     0      0      0      0  0*    Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
> GET http://127.0.0.1:8080/test.html HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.4.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
{ [16 bytes data]
100 308  0 308  0 12377  0 --:--:-- --:--:-- --:--:-- 12833
* Closing connection

C:\Users\thang>

```

This tells us that: The output from the curl command using the --http0.9 flag shows that the request was successfully sent through your proxy server to the Flask server, and a response of 308 bytes was received and written to first\_request.html.

Now if we check on the terminal of our web proxy server; we get the output:



The image shows a web browser window at the top with the address bar displaying 'web server'. Below it is a terminal window with a dark background. The terminal shows the output of a curl command and the proxy server's response. The curl command is: `curl --http0.9 -v -x http://127.0.0.1:8080 http://127.0.0.1:8000/test.html --output second_request.html`. The output shows the request being sent to the proxy server, which then forwards it to the original server. The response is a 200 OK status with a content length of 308 bytes.

```
1 from socket import *
2 import select
3 import requests
4 import datetime
5
6 cache = {}
7
8 def run_proxy_server():
9     proxy_host = '127.0.0.1'
10    proxy_port = 8080
11
12    proxy_socket = socket(AF_INET, SOCK_STREAM)
13    proxy_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Accepted connection from ('127.0.0.1', 63213)
Received request:
GET http://127.0.0.1:8000/test.html HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: curl/8.4.0
Accept: */*
Proxy-Connection: Keep-Alive

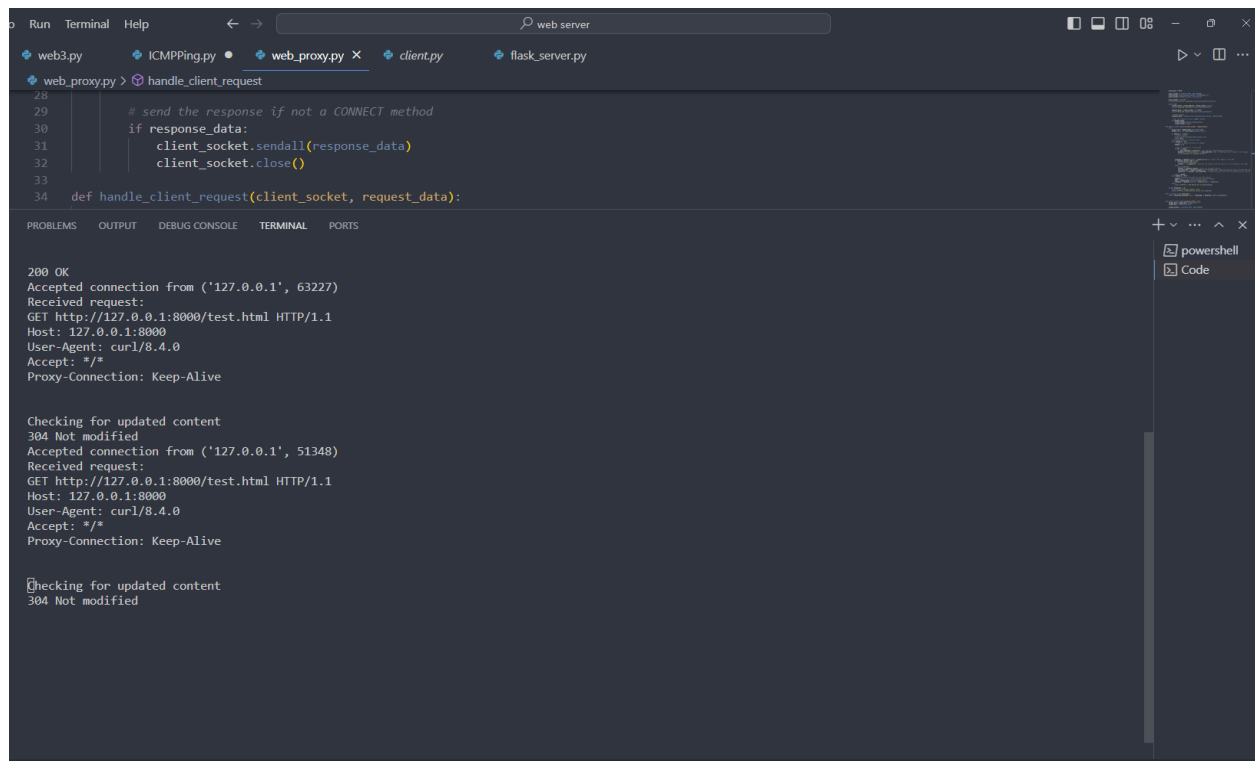
200 OK
```

This result is to be expected since there is no content to be found on the cache first so the proxy would send the request to the original server. After that the proxy server updates the cache with this new content and its last modified timestamp.

We will wait for 1 minute or longer since the freshness period defined in our proxy server is roughly around 1 minute. This is to ensure the cached content becomes stale. After 1 minute, we can issue the same command above:

“curl --http0.9 -v -x http://127.0.0.1:8080 http://127.0.0.1:8000/test.html --output second\_request.html”

Now in the terminal of our web proxy server; we can see that:



```
28
29     # send the response if not a CONNECT method
30     if response_data:
31         client_socket.sendall(response_data)
32         client_socket.close()
33
34 def handle_client_request(client_socket, request_data):
```

```
200 OK
Accepted connection from ('127.0.0.1', 63227)
Received request:
GET http://127.0.0.1:8000/test.html HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: curl/8.4.0
Accept: */*
Proxy-Connection: Keep-Alive

Checking for updated content
304 Not modified
Accepted connection from ('127.0.0.1', 51348)
Received request:
GET http://127.0.0.1:8000/test.html HTTP/1.1
Host: 127.0.0.1:8000
User-Agent: curl/8.4.0
Accept: */*
Proxy-Connection: Keep-Alive

Checking for updated content
304 Not modified
```

Because the content for the URL is found in the cache so the proxy server is checking if the cached content is still fresh or needs to be updated. It does this by sending a conditional GET request to the Flask server with the If-Modified-Since header. The Flask server responded with a 304 Not Modified status, indicating that the requested content hasn't changed since the last time it was fetched and cached by the proxy server. This response is used by the proxy server to understand that the cached version of the content is still valid and can be served to the client without fetching it again from the Flask server.

### Bonus Point: Step Four

The current implementation of our servers do have Head-of-Line (HOL) blocking problem, since the server processes requests sequentially and synchronously. Meaning that it would have to process the first request then move on to the next request where if a particular request takes a long time to process, the users would experience delays and slow-connection. A solution for this problem would be the implementation of multithreading where each request would be handled

by different threads. Meaning that the server can operate multiple requests simultaneously without one blocking the others. We have introduced threading in our web proxy server to address the HOL problem.