

CMPT 371 MP1

Noah Hikichi, 301465527

Joseph Zhang, 301564229

1.

Status Code 200, OK:

Specifications of the logic for the generation of status code: Returned when a valid “GET” request is received and the requested file exists and can be read.

HTTP request message to test it:

GET /blah.html HTTP/1.1

Host: localhost:1234

Connection: close

Status Code 304, Not Modified:

Specifications of the logic for the generation of status code: Returned when the request includes “If-Modified-Since” and the file has not been changed since that date.

HTTP request message to test it:

GET /blah.html HTTP/1.1

Host: localhost:1234

If-Modified-Since: Mon, 1 Oct 2000 01:00:00 GMT

Status Code 403, Forbidden:

Specifications of the logic for the generation of status code: Returned when the requested file exists but access is denied like hitting a restricted directory or no read permission.

HTTP request message to test it:

GET /private/blah.html HTTP/1.1

Host: localhost:1234

Status Code 404, Not Found:

Specifications of the logic for the generation of status code: Returned when the requested file does not exist on the server.

HTTP request message to test it:

GET /doesntexist.html HTTP/1.1

Host: localhost:1234

Status Code 505, HTTP Version Not Supported:

Specifications of the logic for the generation of status code: Returned when the client uses an unsupported HTTP version.

HTTP request message to test it:

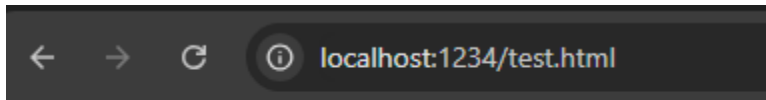
GET /blah.html HTTP/2.0

Host: localhost:8080

2.

a) Code is in web_server.py

b) Browser test:



Congratulations! Your Web Server is Working!

c) Curl tests for generation and responding of status codes:

200 OK:

```
~ $ curl -i http://localhost:1234/test.html
HTTP/1.1 200 OK
```

304 Not Modified:

```
~ $ curl -i -H "If-Modified-Since: Mon, 27 Oct 2024 22:00:00 GMT" http://localhost:1234/test.html
HTTP/1.1 200 OK

~ $ curl -i -H "If-Modified-Since: Mon, 27 Oct 2025 22:00:00 GMT" http://localhost:1234/test.html
HTTP/1.1 304 Not Modified
```

403 Forbidden:

```
~ $ curl -i http://localhost:1234/private/NoReadAccess.html
HTTP/1.1 403 Forbidden
```

Note: tested with an .html with no read access to get the 403, but could not submit it since we couldn't zip a file with no read access. Decided to add a check if the path starts with private/

directory to respond with status code 403 and submitted a zip with a private path and .html we tested as well.

404 Not Found:

```
~ $ curl -i http://localhost:1234/doesntexist.html
HTTP/1.1 404 Not Found
```

505 HTTP Version Not Supported:

```
~ $ curl -i --http1.0 http://localhost:1234/test.html
HTTP/1.1 505 HTTP Version Not Supported
```

3.

Note: We used a separate file to implement the proxy server, as we felt that this was more modularized and better from an organizational perspective.

a) Differences in request handling in a proxy server and a web server:

The web server acts as a file storage system that receives requests, and handles the requests by sending the corresponding files. In the original implementation, the web server is the final destination for all requests, so all relevant data and objects should be directly returned by the web server. Note that since we are using a simplistic proxy server without handling 304, we will always return the object in 304 responses.

The proxy server acts like an intermediate temporary storage system in between the client and web server, and also receives and handles requests from clients. For most requests, the proxy server will return the object if it is temporarily stored, or send a request to the web server if the object is not currently stored. After receiving a response from the web server, the proxy server will then update its temporary storage (evict on LRU) and return the object to the client.

If the client uses an If-modified-Since header, we will always forward these requests to the web-server. If the response to the client's request is 403/404, we will always forward these requests directly to the client without storing the response. If the response is 304, we will update the cache and forward the web-server's response. 505 is handled by proxy, as explained below.

Protocol when proxy receives requests from client:

Conditional GET	These requests are always immediately forwarded to the web-server
Bad version	Requests that result in 505 responses are immediately returned by proxy

Other	If the request is in cache, return the response in the cache. Otherwise, forward the request to the web server.
-------	---

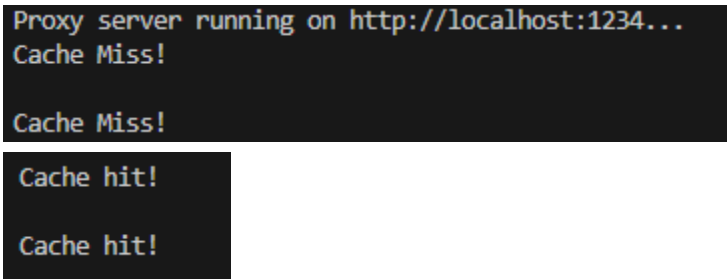
Protocol when proxy receives response from web-server:

200	Client sends GET, Proxy receives request. Proxy checks if the object is in cache. If the object is in cache, the proxy returns with 200. If the object is not in cache, the proxy sends a request to the web server, caches the object (evicting if necessary) and sends the response back to the client.
304	Because we are using the simplistic proxy server, the 304 from the web server will always return the object. For the web-server, we will store the object in the cache, then forward the web-server's response to the client.
403	Not stored in cache, the request is always forwarded to the web server, and the response from the web server is always forwarded to the client.
404	Not stored in cache, the request is always forwarded to the web server, and the response from the web server is always forwarded to the client.
505	Not stored in cache, the request is always forwarded to the web server, and the response from the web server is always forwarded to the client.

Finally, we also note that the implementation of the proxy server uses a simplistic model, and does not keep track of a user's history. In normal production standards, the proxy server should maintain a log of the user's past interaction. The consequence of this is that

b) Test procedure to show the working proxy server:

To test the proxy server, we used the terminal running the proxy server with print statements to show when the cache would hit and miss. We used a cache size of 2 objects for simplicity. We've outlined the test cases we tried in the table below.

Test case	Result
GET test.html GET test1.html Expected: 2 cache misses GET test.html GET test1.html Expected: 2 cache hits	

GET test2.html GET test1.html Expected: cache miss, cache hit	<div>Cache Miss!</div> <div>Cache hit!</div>
GET test.html Expected: cache miss	<div>Cache Miss!</div>
GET NonExistant.html GET NonExistant2.html Expected: 2 cache miss GET test.html GET test1.html Expected: 2 cache hits (This is to test 404s are not stored)	<div>Cache Miss!</div> <div>Cache Miss!</div> <div>Cache hit!</div> <div>Cache hit!</div>
GET private/NoReadAccess.html Expected: cache miss GET test.html GET test1.html Expected: 2 cache hits (This is to test 403s are not stored)	<div>Cache Miss!</div> <div>Cache hit!</div> <div>Cache hit!</div>
GET test.html (HTTP1.0) Expected: Don't check cache, no hit or misses GET test.html GET test1.html Expected: 2 cache hits (This is to test 505s are not stored)	<p>The location for the first test is highlighted in full history below.</p> <div>Cache hit!</div> <div>Cache hit!</div>
GET test3.html GET test3.html (If-Modified-Since 10/27/2024) GET test3.html (If-Modified-Since 10/27/2025) GET test3.html GET test1.html Expected: cache miss, cache hit, cache hit	<p>304s work similarly to cache misses, however it is not printed out, as we will never search the cache before forwarding the request to the web server. The location is again marked below. Additionally, the 304 should only update the LRU value for the existing entry.</p> <div>Cache Miss!</div> <div>Cache hit!</div> <div>Cache hit!</div>

Full History:

```
Proxy server running on http://localhost:1234...
Cache Miss!

Cache Miss!
Cache hit!

Cache hit!
Cache Miss!

Cache hit!
Cache Miss!
Cache Miss!

Cache Miss!
Cache hit!

Cache hit!
Cache Miss!
Cache hit!

Cache hit!
Cache hit!
Cache hit!
Cache Miss!
Cache hit!
Cache hit!
```

Handwritten red annotations on the terminal output:

- A red line under "Cache hit!" is crossed out, with an arrow pointing to "505".
- A red line under "Cache hit!" is crossed out, with an arrow pointing to "304".

c) Why our web server is a multi-thread server and how that impacts performance:

We made our web / proxy server multithreaded by creating a new separate thread to handle requests for each client connection. Each thread handles all aspects of a client request which allows the server to handle multiple clients at once rather than processing them one at a time. This helps reduce latency before it doesn't block requests all waiting on one thread. It also better utilizes resources since IO operations like reading files and network communication can overlap

which lets the CPU work on other requests while waiting for IO in another thread. Overall it has better concurrency leading to lower latency and better throughput compared to the single threaded server.

4. (Bonus)

Changes to avoid the HOL problem using frames:

To reduce HOL blocking, the web server now sends responses in small frames of size 512 instead of sending the entire file at once. This allows each thread to send data gradually, so a slow client doesn't block the server from handling other requests. Files are read and sent in chunks/frames, which also improves memory efficiency and reduces latency for the client. This chunked sending along with the multithreading ensures the server can keep handling when given multiple requests at a time.

Code Instructions:

To run the full project, run each of the files (`web_server.py`, `proxy_server.py`), and use `localhost:1234` as the host/port for the proxy server. To directly access the web server without going through the proxy server, use `localhost:3652`.