Xander Smith
Jonathan Peters
CMPT 371 Mini Project 1

Simple Web Server Requirements

Instructions to Run

(from README.md)

1. Start the Web Server in a new shell tab or window.

```
python3 http_server.py <port> #(optional, default = 8080)
```

2. Send HTTP requests using Curl, Telnet, or your browser

For example:
http://127.0.0.1:8080/test.html
or

```
curl -i http://127.0.0.1:8080/test.html
```

3. Run automated tests with the following command (while the server is running)

```
Python3 http_server_test.py
```

4. Adjust server settings remotely if desired:

Clear cache:

```
curl "http://127.0.0.1:8080/__cache__/clear"
```

Adjust simulated propagation delay (the default amount is 0 seconds):

```
curl "http://127.0.0.1:8080/__cache__/set-miss-delay?seconds=<float>"
```

Adjust base TTL for items in cache (the default is 60 seconds):

```
Curl “http:127.0.0.1:8080/__cache__/set-expiry?<int>”
```

Evict all expired cache items

```
Curl “http:127.0.0.1:8080/__cache__/evict-expired”
```

Request Specifications

200 (Ok)

Requirements

- We currently only support the GET method in the request
- For the response to be 200, the syntax of the request must be correct, and the requested resource must exist and be accessible
- A response payload must always exist unless the Request method is CONNECT. Can be an empty string: ‘’
- The requested resource is cacheable

If the server receives an HTTP request and it satisfies the requirements, a 200 response will be sent.

Example

```
> GET /test.html HTTP/1.1
> Host: localhost (127.0.0.1)
>
```

Testing

```
curl -i http://127.0.0.1:8080/test.html
```

Expected Output (caching-specific fields in red):
Header:

```
HTTP/1.1 200 OK
Date: <when the response was generated>
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/html
Content-Length: 308
Cache-Control: <3600>
ETag: <"abc123">
Last-Modified: <when response was last modified>
Vary: Accept-Encoding
Connection: close
X-Cache: <hit/miss>
```

Body:

```
<!DOCTYPE html>
<rest of body>
```

304 (Not Modified)

Requirements

- The server generates a response to a GET method if the request is sent with a conditional header that evaluates to false
- We check for the header field “If-Modified-Since”, and respond with 304 if the resource has not changed since the specified date (or ETag)
- The server must generate the same header fields as a 200 (Ok) message would.
- The server should not generate additional metadata aside from the listed fields above. The exception to this would be metadata that assists in guiding cache updates. (Last-Modified if the response does not have an ETag)
- The response cannot contain a message body
- If a proxy server forwarded a conditional get on behalf of a user agent with its own cache and also received a 304 response it should forward the 304 response to the client.

Example of Valid Request

```
> GET /test.html HTTP/1.1
> Host: localhost
> If-Modified-Since: Sat, 25 Oct 2025 23:00:00 GMT
>
>
```

Testing

```
curl -v -H "If-Modified-Since: Sat, 25 Oct 2025 23:00:00 GMT"
http://127.0.0.1:8080/test.html
```

Expected Output:
Header:

```
HTTP/1.1 304 Not Modified
Date: <when the response was generated>
Server: Peters-Smith-Web-Server/1.0
Content-Length: 0
Cache-Control: <3600>
ETag: 'abc1'
Last-Modified: <when response was last modified>
Vary: Accept-Encoding
Connection: close
X-Cache: <hit/miss>
```

403 (Forbidden)

Requirements

- Indicates that the server understands the request but refuses to authorize it

- A server may send the 403 code relating to invalid credentials or for something unrelated, like a file without read permissions or outside of the server's root directory
- An origin server may hide a forbidden target resource by responding with a 404 code instead, but we chose to send a 403 code instead for the purposes of testing.
- Reason why the request has been forbidden may be described in the payload

Example

```
> GET /locked.html HTTP/1.1
> Host: localhost
>
>
```

Testing

```
curl -v http://127.0.0.1:8080/locked.html (make a "locked" file)
```

- no read/write/execute permissions

```
printf "GET /../test.html HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc
127.0.0.1 8080
```

- accessing a directory outside of project root (curl doesn't seem to support this?)

Expected Output
Header:

```
HTTP/1.1 403 Forbidden
Date: <when the response was generated>
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 29
Connection: close
```

Body:

```
403 Forbidden: Access denied
```

404 (Not Found)

Requirements

- The server responds with a 404 code when it is unable to find a current representation for a target resource i.e. it does not exist.

Example

```
> GET /fake.html HTTP/1.1
> Host: localhost
>
>
```

Testing

```
curl -v http://127.0.0.1:8080/fake.html
```

- try to get a non-existent file

Expected Output
Header:

```
HTTP/1.1 404 Not Found
Date: <when the response was generated>
Server: Peters-Smith/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 15
```

Body:

```
File Not Found
```

505 (HTTP version not supported)

Requirements

- The server is unable, or unwilling to complete the request using the version of HTTP requested by the client.
- The server should generate a response that describes why the version is not supported, and what other protocols are supported by the user.

Example

```
> GET /test.html HTTP/3.0
> Host: localhost
>
>
```

Testing (curl doesn't support this, pipe to nc)

```
printf "GET /index.html HTTP/3.0\r\nHost: localhost\r\n\r\n" | nc
127.0.0.1 8080
```

Expected Output
Header:

```
HTTP/1.1 505 HTTP Version Not Supported
Date: <when the response was generated>
```

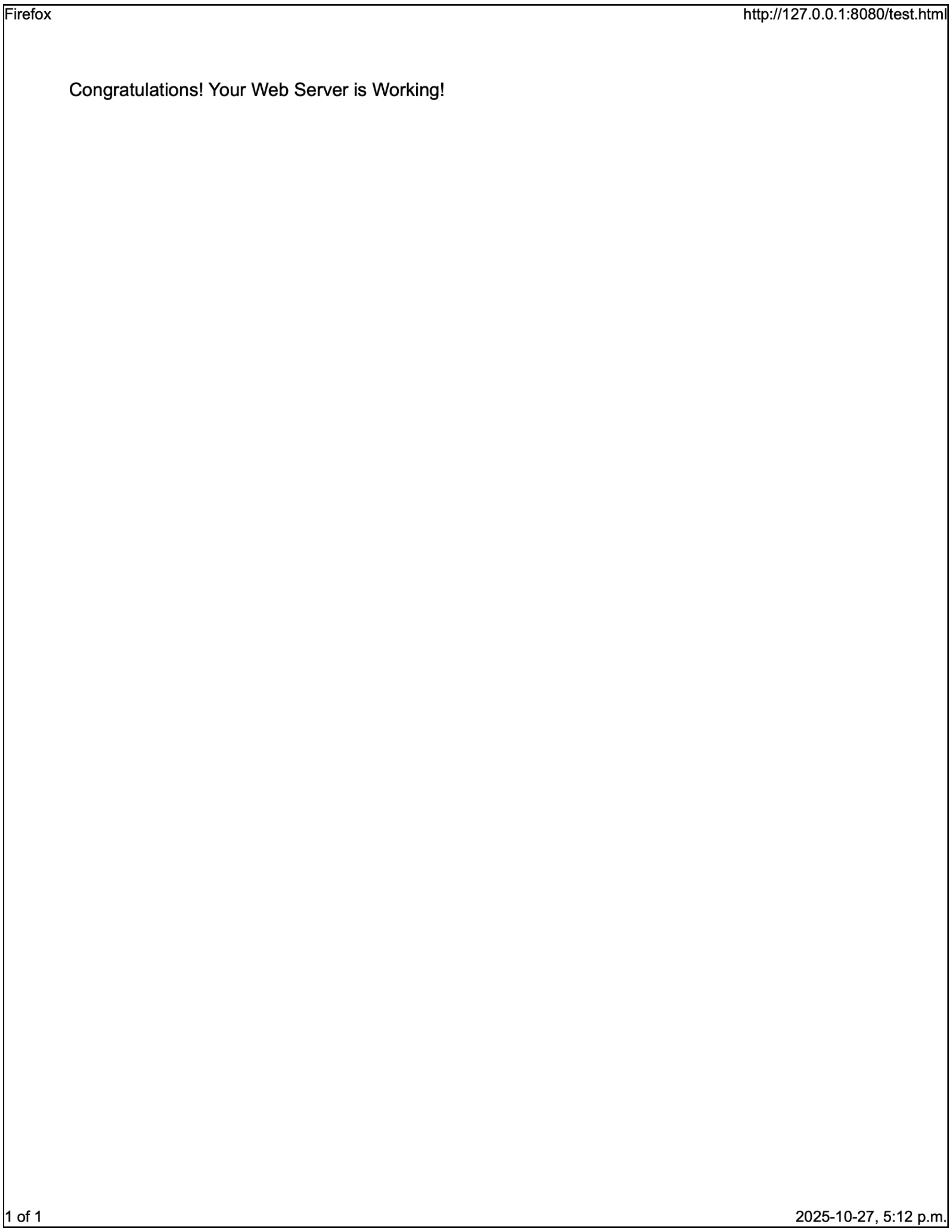
```
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 70
Connection: close
```

Body:

```
HTTP Version Not Supported.
Only HTTP/1.0 and HTTP/1.1 are supported.
```

Testing the Web Server

Browser Screenshot



Firefox http://127.0.0.1:8080/test.html

Congratulations! Your Web Server is Working!

1 of 1 2025-10-27, 5:12 p.m.

Automated Tests

Steps to test:

1. Navigate to project root directory
2. Run `python3 html_server_test.py`
3. See results.md for test output

Test Results:

200 OK

Command:

```
curl -i http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 200 OK
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:24 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/html
Content-Length: 308
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: MISS # didn't implement caching yet
```

Body:

```
<!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>
```

304 Not Modified

Command:

```
curl -i -H 'If-Modified-Since: Mon, 20 Oct 2025 19:08:33 GMT'
http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 304 Not Modified
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:24 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Length: 0
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: MISS
```

403 Forbidden: Locked File

Command:

```
curl -i http://127.0.0.1:8080/locked.html
```

Status Line:

```
HTTP/1.1 403 Forbidden
```

Headers:

```
Date: Mon, 27 Oct 2025 23:50:24 GMT
Server: Smith-Peters-Web-Server/1.0
```

```
Content-Type: text/plain; charset=utf-8
Content-Length: 29
Connection: close
```

Body:

```
403 Forbidden: Access Denied
```

403 Forbidden: File outside server root path

Command:

Send using Python Socket module

```
GET /../ HTTP/1.1\r\nHost: localhost\r\n\r\n
```

Status Line:

```
HTTP/1.1 403 Forbidden
```

Headers:

```
Date: Tue, 28 Oct 2025 02:34:48 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 29
Connection: close
```

Body:

```
403 Forbidden: Access Denied
```

404 Not Found

Command:

```
curl -i http://127.0.0.1:8080/no_such_file.html
```

Status Line:

```
HTTP/1.1 404 Not Found
```

Headers:

```
Date: Mon, 27 Oct 2025 23:50:24 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 15
Connection: close
```

Body:

```
File Not Found
```

405 Method Not Allowed

Command:

```
'Socket send: POST /test.html HTTP/1.1\r\nHost: localhost\r\n\r\n'
```

Status Line:

```
HTTP/1.1 405 Method Not Allowed
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:24 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 19
Connection: close
```

Body:

```
Method Not Allowed
```

505 Version Not Supported

Command:

Send using Python Socket module

```
GET /test.html HTTP/3.0\r\nHost: localhost\r\n\r\n
```

Status Line:

```
HTTP/1.1 505 HTTP Version Not Supported
```

Headers:

```
Date: Mon, 27 Oct 2025 23:50:24 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/plain; charset=utf-8
Content-Length: 70
Connection: close
```

Body:

```
HTTP Version Not Supported.

Only HTTP/1.0 and HTTP/1.1 are supported.
```

Proxy Server Specifications

The primary difference between a Proxy Server and a Web Server is that the Proxy Server acts as both a client and server depending on the situation. A Proxy Server is intended to reduce access link utilization to Web Servers by storing client requests in a local cache. If a request is already in the cache then the Proxy Server may send the data in its cache instead of requesting data from a distant web server. Installing a Proxy Server is a relatively cheap method of reducing response time for clients when requesting a resource over the web.

General Cache behaviour

Data inside the cache is stored in a list object that is bounded by an attribute limiting the maximum cache size. Once a worker thread has validated that the request has no formatting errors (such as unsupported methods or http versions) the request is passed into the cache to see if there is a record in the cache that matches the request specifications. The URL of the request is used as the primary key for comparisons. If a matching record is found then the web server determines whether the client possesses an up to date copy or not. This determines whether the web server responds with a 304 or 200 status code.

If there is a cache miss then the Web Server requests the resource externally. If the request is fulfilled successfully then the record is inserted into the cache.

Only one thread may have access to the Cache object at a time. This is achieved using a shared lock among all threads.

Record Class

Defined in the cache_utils.py module, this class provides the underlying infrastructure for the cache class. Each record is distinguished by 10 attributes.

- `_etag`*:* This attribute assists clients with acquiring the correct version of the resource. It is calculated as the hash of a tuple that contains the Content and the Vary values of the associated record.
- `_last_modified`: A string formatted into “Day, Month etc ”. Assists with helping the Proxy Server send only necessary requests to a distant web-server.
- `_vary`: A value that allows further identification of cached resources for the client. Our implementation has the Vary baked into the ETag for faster invalidation.
- `_expires`: tells the cache when the Record should be evicted from cache resulting from age. Minimum value is 2 seconds but can be longer.
- `_content_type`: Stores the content type for the Content-Type header field.
- `_content`: Stores the content for the body of a packet. Is used in ETag for the hash.
- `_url`*:* The primary key of the record. A matching URL in the cache does not always result in returning the record.
- `_version`: Represents the version of the stored request
- `_req_headers`: A dictionary that represents other headers that is important for representation

Alongside getter methods for each attribute, the record class has 4 additional methods

```
update_expiry_date(self, offset:float=0):
```

When called on a record updates the _expires attribute to be the time at which the method was called + (DEFAULT_TTL_SECONDS + offset) seconds. This method is called upon initialization of the record to provide it with a minimum life span before being evicted from the cache. DEFAULT_TTL_SECONDS is 60 by default and can be modified to any integer value remotely.

```
@staticmethod
_extract_request_line(key: dict):
```

A helper method used in *is_match* to assist with finding and transforming values used to identify records uniquely. Returns assorted keys as a tuple.

```
is_match(self, key) -> bool:
```

Helper function that performs a series of checks on the key to determine if the calling record is a match to the passed to the key. Identifies commonality by a url, method, and version of HTTP. Afterwords performs key match on for modification date and etag values among others to confirm that the request has validated this request.

```
is_newer_than(self, header_str : str):
```

Header_str is a string formatted by the formatdate function from the email.utils module. This parameter as well as the calling records _last_modified value is converted to a datetime object and compared. Returns a boolean value, True if the record was more recently modified than the request. This method is used on cache hit within the handle request function. If it evaluates true then the resource held by the client is old and needs to be updated -> 200 Ok. Otherwise the resource is still valid -> 304 Not Modified.

Cache Class

The cache object is instantiated up running http_server.py and is initially empty. The cache is shared between each worker thread. In a typical scenario, a worker thread interfaces with the cache with either the find_records() method or the insert_request() method. Other methods are available to the thread, but are only used in testing to verify cache behaviour. Each Cache instance is defined with a lock object, so that the object handles its own lock behaviour instead of the threads that may access the cache.

A object cache has 3 attributes.

- `_records`: A list that stores items of type Record for retrieval.
- `_max_capacity`: Unchanging attribute that limits the number of elements that the *_records* attribute can contain. Default value is 2.
- `_lock`: A lock as defined by `threading.lock`.

The Cache class has 8 methods, not including the constructor.

```
_change_base_TTL(self, val):
```

Testing function. Within the cache_utils module there is a global variable called DEFAULT_TTL_SECONDS. By default is set to 60 but can be changed by using this function. It is not recommended to use the function outside of a testing context. Is accessed by the client side command:

```
Curl "http:127.0.0.1:8080/__cache__/set-expiry?<int>"
```

Where the value of DEFAULT_TTL_SECONDS is updated to some int.

```
_is_expired(self, record):
```

A helper function that checks if the records _expiry attribute is less than or equal to the current time. Returns a boolean value.

```
_remove_records(self, array):
```

A helper function that receives an array of records and when called removes them from the calling cache objects _records attribute. The records in *array* should be in _records.

```
find_records(self, key):
```

One of two non-test methods called by *handle_requests()*. Method Iterates through each record in _*records* and performs a series of checks on them. Either returns a Record or None. When called tries to acquire the cache lock.

While searching through each record, before checking if it is a match first checks if the record has expired using _*is_expired()*. If it has then the record is appended to a list for future removal and the iteration is incremented.
If the record is not expired then it is compared against *key* with the *is_match* Record method. This is done until the first match or all records have been checked.
If a match is found the corresponding is moved to the front_*records* and returned.
Before the function exits completely _*remove_records()* is called with the list containing the expired record entries.

```
print_cache(self):
```

Prints each record in the cache to the console

```
insert_response(self, record):
```

One of two non-test methods called by *handle_requests()*. This method takes a record and inserts it into the front of the cache. When called tries to acquire the cache lock.
If insertion is attempted while the cache is at capacity then the method first removes all expired items from the cache. If the cache has no expired records then the back most record is popped opening up room.
The record is always inserted at the front of _*records.*

```
clear_cache(self):
```

Test method. Empties the cache. When called tries to acquire the cache lock. Is accessed by the client-side command:

```
curl "http://127.0.0.1:8080/__cache__/clear"
```

```
evict_expired(self):
```

Test method. Iterates through every element in _*records* and removed them from the cache if they have expired. When called tries to acquire the cache lock. Is accessed by the client-side command:

```
Curl “http:127.0.0.1:8080/__cache__/evict-expired”
```

Testing the Cache

We also run automated tests to check the correctness and performance of our cache implementation. We add a 1.2 second delay to a response if it is not in the cache, to simulate the real-world propagation delay of retrieving the resource from a distant web server. The caching tests are in the same module as the requirements tests above.

Steps to test:

4. Navigate to project root directory
5. Run `python3 html_server_test.py`
6. See results.md for test output

304 Not Modified with ETag from Cache

Command:

```
curl -i -H 'If-None-Match: "2189052537601941603"' -H 'Accept-Encoding: 
identity' http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 304 Not Modified
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:25 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Length: 0
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: HIT
```

304 Not Modified with If-Modified-Since from Cache

Command:

```
curl -i -H 'If-Modified-Since: Mon, 20 Oct 2025 19:08:33 GMT' -H 
'Accept-Encoding: identity' http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 304 Not Modified
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:25 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Length: 0
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: HIT
```

Cache MISS on First Request:

Command:

```
curl -i -H 'Accept-Encoding: identity' http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 200 OK
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:27 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/html
Content-Length: 308
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: MISS
```

Cache HIT on Repeat Requests: Request 2

Command:

```
curl -i -H 'Accept-Encoding: identity' http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 200 OK
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:27 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/html
Content-Length: 308
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: HIT
```

Cache Expiry Eviction

Commands:

```
curl -sS -X POST http://127.0.0.1:8080/__cache__/set-expiry?0
```

- Ensures that any records will be expired as they enter the Cache

```
curl http://127.0.0.1:8080/test.html
```

- Inserts a record into cache

```
curl -sS -X POST http://127.0.0.1:8080/__cache__/evict-expired
```

- Tells cache to evict all expired records. Sends the number of records in cache as body.

This test confirms the eviction of the expired record in conjunction with other tests passing where the record is confirmed inserted.

Cache Miss then Hit with Vary on Accept-Encoding

Command:

```
curl -i --compressed http://127.0.0.1:8080/test.html
```

Status Line:

```
HTTP/1.1 200 OK
```

Headers:

```
Date: Wed, 29 Oct 2025 06:40:29 GMT
Server: Smith-Peters-Web-Server/1.0
Content-Type: text/html
Content-Length: 308
Cache-Control: max-age=3600
ETag: "2189052537601941603"
Last-Modified: Mon, 20 Oct 2025 19:08:33 GMT
Vary: Accept-Encoding
Connection: close
X-Cache: HIT
```

High Volume Requests Performance

Command:

```
ab -n 50 -c 10 -H 'Accept-Encoding: identity'
http://127.0.0.1:8080/test.html
```

Status Line:

```
ab completed
```

Headers:

```
Elapsed: 1.24s
```

This test runs a concurrent load of requests using the ApacheBench (ab) tool. It checks that all requests receive correct responses and that the benchmark finishes in under a few seconds. This test would fail if caching is not implemented, since each request would take 1.2 seconds, resulting in a lengthy wait for the benchmark to finish. By implementing caching, we see that the test only takes 1.24 seconds to finish, since the first test is delayed, but every request after returns the correct response from the web proxy cache.

Multi-threading Performance

We are again using ApacheBench, a command-line tool to test the performance of the server. To simulate a processing delay, sleep for 0.01 seconds before handling the request.

We sent 1000 requests per test, using concurrency levels 1 (single-threaded), 10, 20, and 100.

Note that the number of worker threads used for each test must be greater than the level of concurrency. Otherwise, the tool overwhelms the server and results in some connections being refused since all threads are busy. There are two ways to solve this issue:

1. Add more worker threads.
2. Implement a queue for requests to wait in before they are processed.

While the latter option is a more robust solution, this increases the level of complexity of our simple http server. Therefore, for each test we have a few more worker threads to handle a high volume of traffic.

Single-threaded: 1000 requests (no concurrency)

```
ab -n 1000 http://127.0.0.1:8080/test.html
```

Example of results:

```
Server Software:        Smith-Peters-Web-Server/1.0
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /test.html
Document Length:        308 bytes

Concurrency Level:      1
Time taken for tests:   14.970 seconds
Complete requests:      1000
Failed requests:        0
Total transferred:      512000 bytes
HTML transferred:       308000 bytes
Requests per second:    66.80 [#/sec] (mean)
Time per request:       14.970 [ms] (mean)
Time per request:       14.970 [ms] (mean, across all concurrent
requests)
Transfer rate:          33.40 [Kbytes/sec] received
```

Multi-threaded: 1000 requests, 12 threads (level 10 concurrency)

```
ab -n 1000 -c 10 http://127.0.0.1:8080/test.html
```

Multi-threaded: 1000 requests, 25 threads (level 20 concurrency)

```
ab -n 1000 -c 20 http://127.0.0.1:8080/test.html
```

Multi-threaded: 1000 requests, 105 threads (level 100 concurrency)

```
ab -n 1000 -c 100 http://127.0.0.1:8080/test.html
```

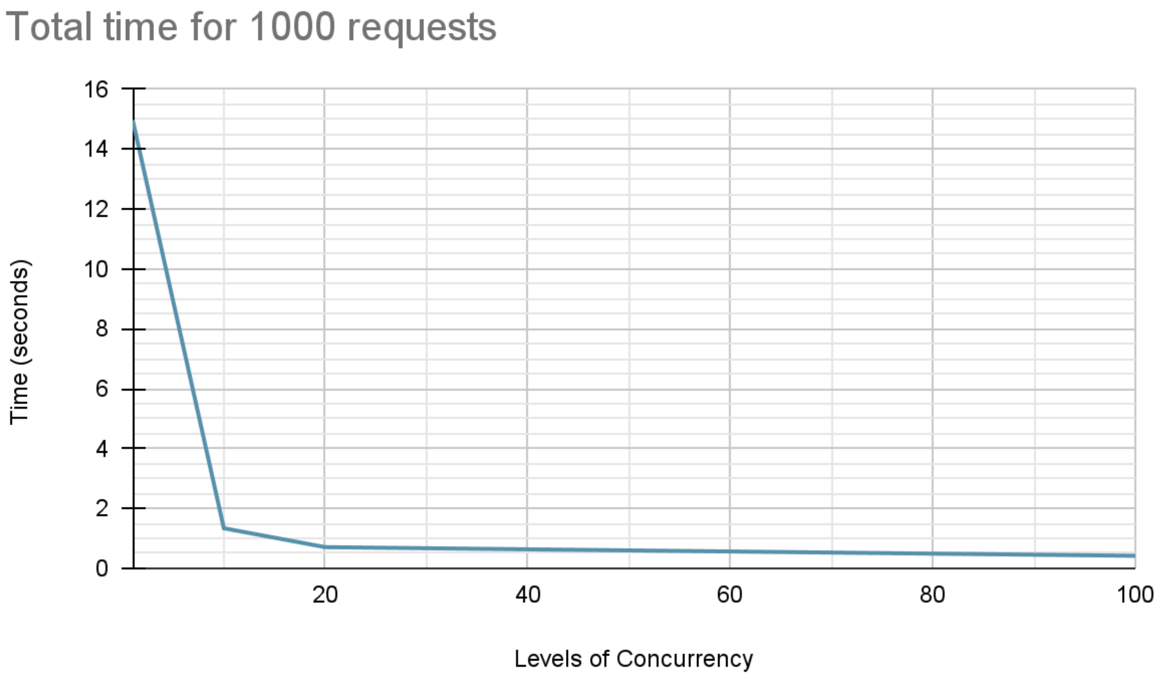
	No concurrency (Level 1)	Level 10	Level 20	Level 100
Total Time	14.970 seconds	1.339 seconds	0.708 seconds	0.419 seconds
Requests (mean)	66.80 [#/sec]	746.80 [#/sec]	1412.61 [#/sec]	2388.01 [#/sec]
Time / request (mean)	14.970 [ms]	13.391 [ms]	14.158 [ms]	41.876 [ms]
Time / request (mean, across all concurrent requests)	14.970 [ms]	1.339 [ms]	0.708 [ms]	0.419 [ms]
Transfer rate	33.40 [Kbytes/sec]	373.40 [Kbytes/sec]	706.30 [Kbytes/sec]	1194.00 [Kbytes/sec]

Results and Discussion:

The benefits of multi-threading over single-threading is clear from the test results. Receiving requests concurrently and then handling them and sending responses over multiple threads reduces the wait time during a high volume of traffic. Our server's multi-threading functionality is implemented in thread_utils.py. The system works by:

- Attempt to initialize a thread upon accepting a connection
- If the maximum number of threads are in use, send a 503 Service Unavailable response
- Otherwise, dispatch a worker thread to receive and parse the http request, and send back the appropriate response.
- To ensure the robustness of our multithreading, we implemented a timeout on the connection to prevent hangs under heavy volume
- We also try to catch exceptions for a broken pipe error, a reset connection, or other errors that could cause our server to crash
- We ensure that the worker thread uses locks to access shared resources and that we properly handle tracking of worker threads to ensure we don't exceed the maximum amount.

Before introducing a simulated delay, the benefits of multi-threading were unclear, since our responses are quite short. After adding a delay of 0.01 seconds, the reduced time to handle 1000 http requests was substantial.



Examining the results, it seems like there are diminishing returns from increasing the number of threads and levels of concurrency. We will leave the number of worker threads at 16 to strike a balance between performance and the amount of overhead needed to manage the worker threads. To handle higher levels of concurrency with fewer threads, we could implement a queue to store requests until a worker thread is available. We will leave this as a future upgrade.