

Step Two - Single-threaded web server

There are two ways to test the single-threaded web server. The first way to test is by using Google Chrome. The second way is by using the Python client program `client_st.py`.

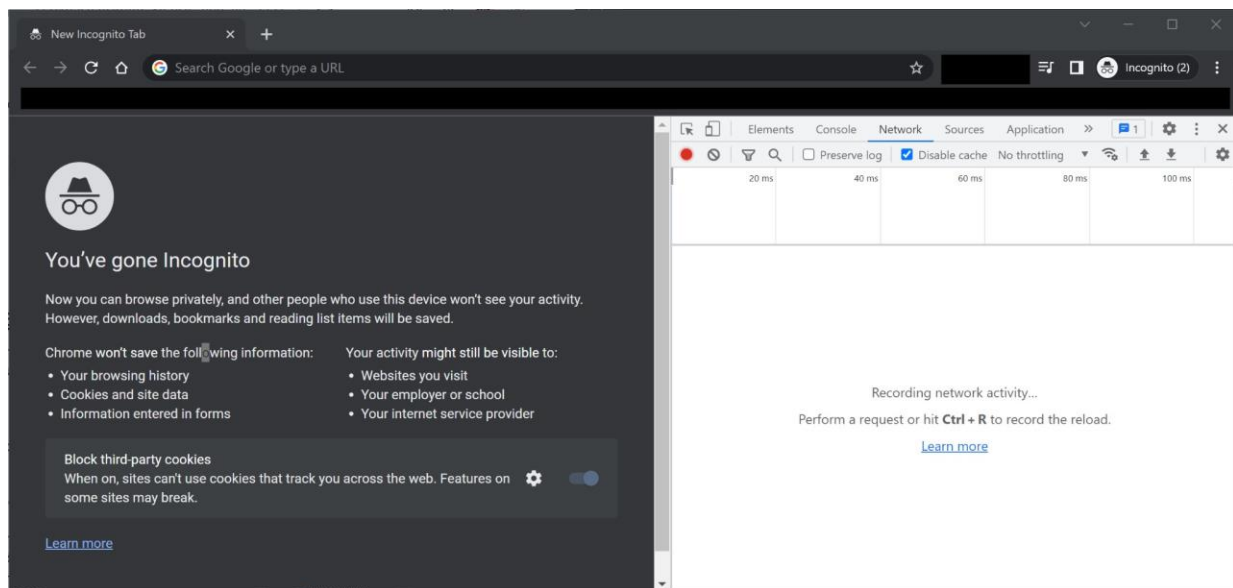
Testing in Google Chrome

We will use the Network tab in the dev tools of a Google Chrome incognito window to see the server response.

Before connecting to the webserver to test any status code, do the following:

1. Go to the `step2` directory, open a terminal, and run the Python server program with the command `python server_st.py`. This will start the single-threaded server.
2. In a Chrome window, press `Ctrl + Shift + N` to open an incognito window.
3. Press `F12` to open the developer tools.
4. Click on the Network tab. If you don't see the Network tab, check the dropdown menu.
5. Click on the Disable cache checkbox to disable caching.

Your Chrome window should look like the image below.

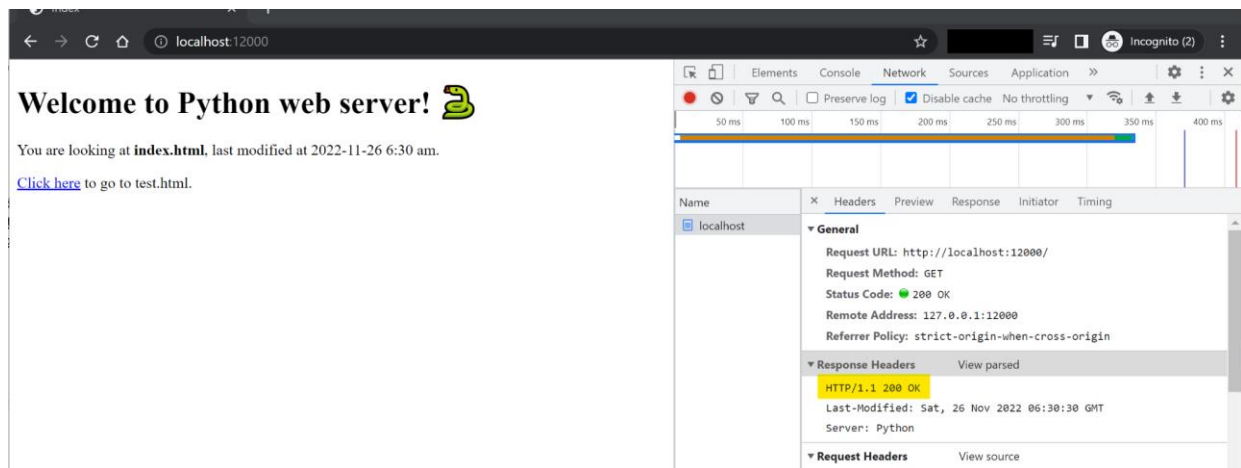


Follow these steps before testing any of the status codes below. It is important to use an incognito window and disable the cache so Chrome doesn't load a cached version of a webpage (note that some tests will have you enable caching). If you want to test a new status code, or re-test a code you've already tested, close the Chrome incognito window first and redo steps 2-5.

To view the server's response message after connecting to it (typing in a URL and pressing Enter), do the following:

1. After connecting to the server, there should be an entry in the Network tab. Click on the name column of the entry.
2. A few more tabs will pop up inside the Network tab. Go to the Headers tab and click on "View source" beside "Response Headers". This will let you see the server's response message.

Your Chrome window should look like the image below. The [status line](#) of the response, what we'll be looking at mostly, is highlighted in yellow.

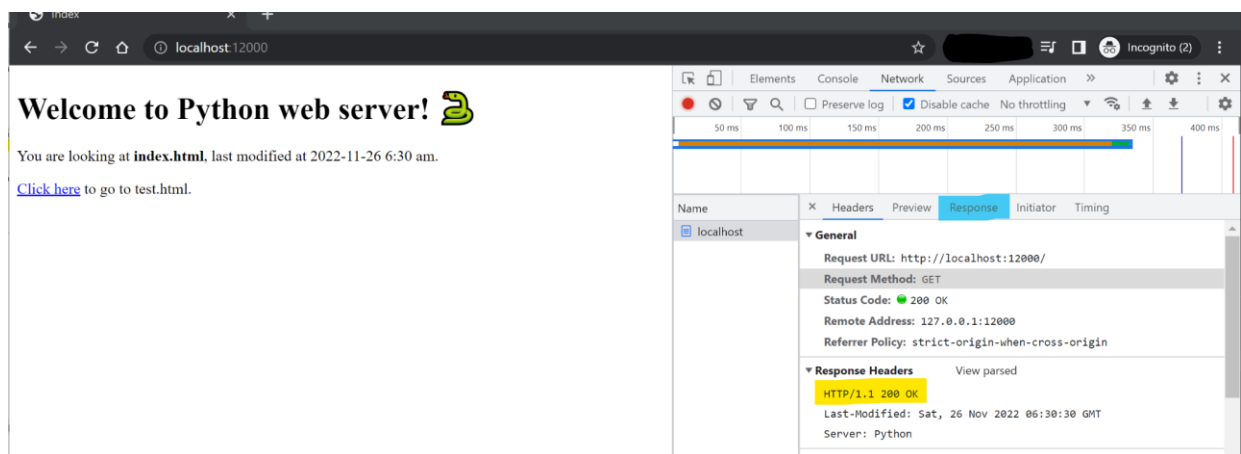


Test status code 200 OK

To test 200 OK,

1. Connect to localhost:12000.
2. Check the response message. It will have the status line HTTP/1.1 200 OK, highlighted in yellow in the image below.

If you want to see the HTML file that the server returned, click on the Response tab, highlighted in blue below. Note that you can go to localhost:12000/test.html instead for the same result.

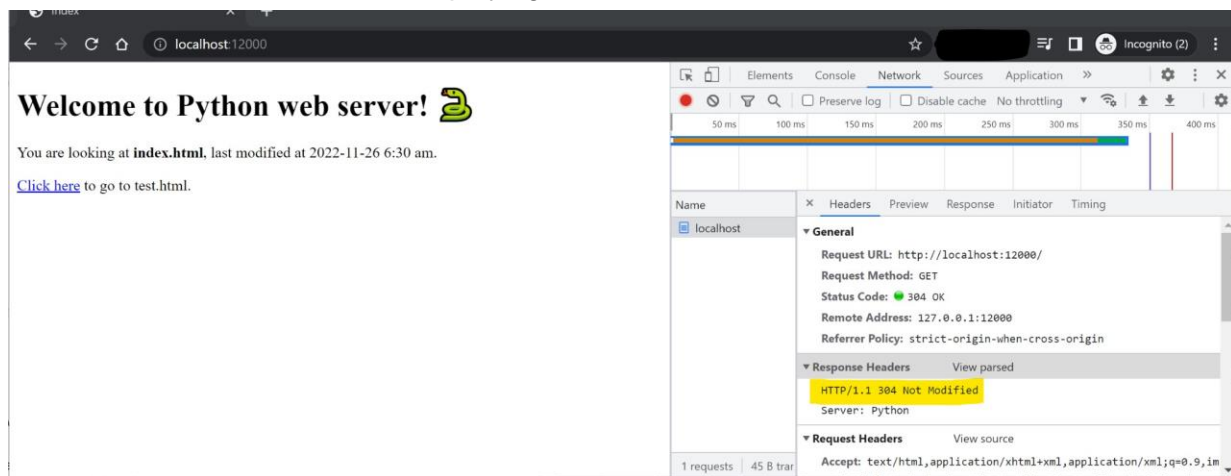


Test status code 304 Not Modified

To test 304 Not Modified,

1. Go to `localhost:12000`. Notice that the server's response message has the status line `HTTP/1.1 200 OK`, which isn't the code we are looking for.
2. Enable caching by clicking the "Disable cache" checkbox again before connecting to the server.
3. Refresh the webpage. The response message will now have `HTTP/1.1 304 Not Modified` status line. This is highlighted in yellow in the image below.

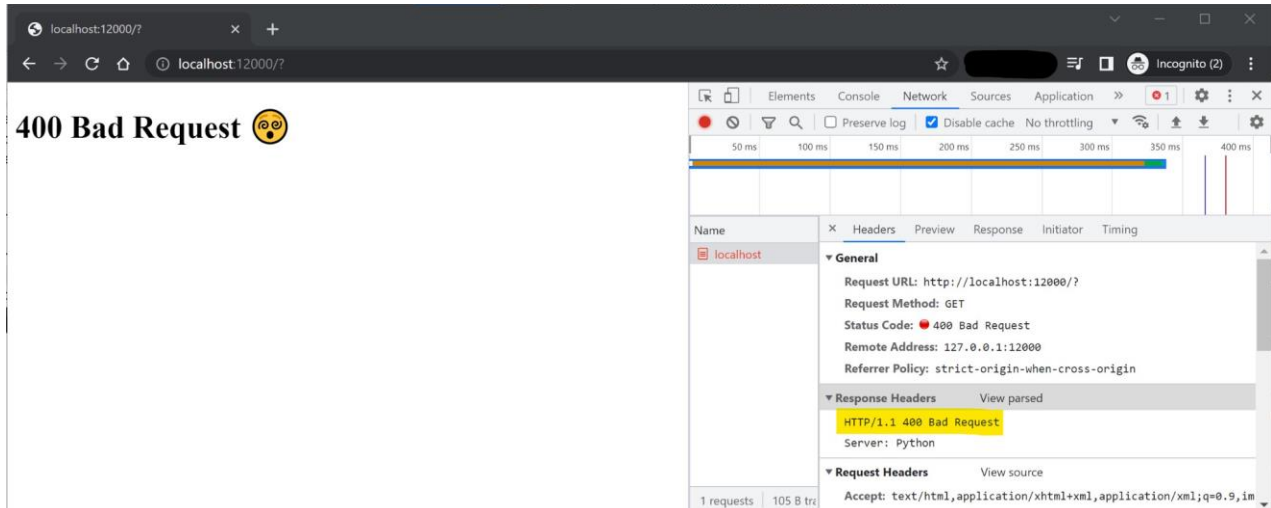
Note that the Response tab will still have an HTML file in it, despite the response message having nobody (you can check `send_HTTP()` in the server code to verify). The server didn't send the file; rather, Chrome is displaying what it loaded from the cache.



Test status code 400 Bad Request

Make sure you disable caching again if you have just tested 304 Not Modified. To test 400 Bad Request,

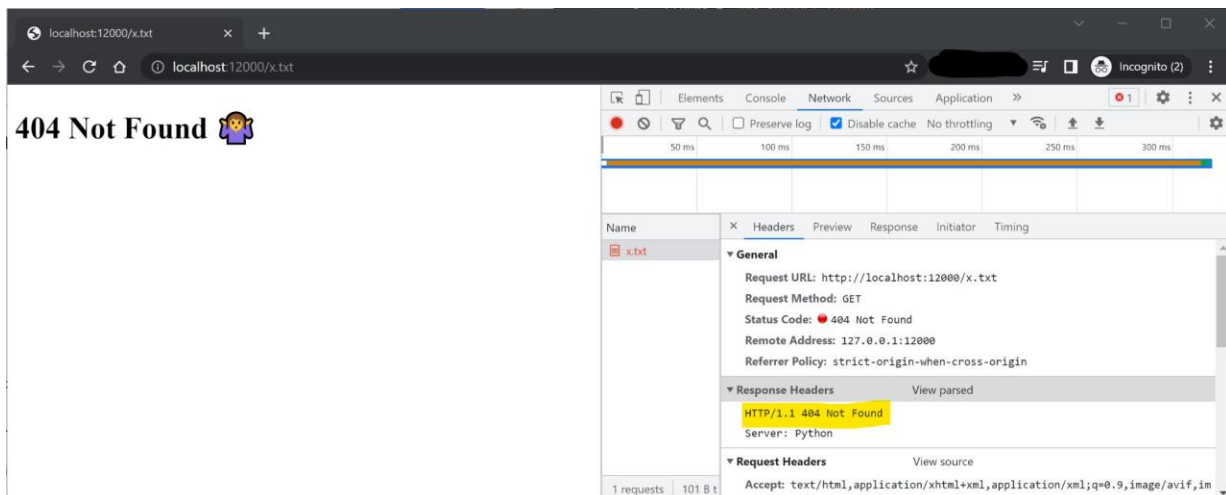
1. Go to `localhost:12000/?`. File names cannot have '?' in them, so the server interprets the request as badly.
2. Check the response message. It will have the status line `HTTP/1.1 400 Bad Request`. This is highlighted in yellow in the image below.



Test status code 404 Not Found

To test 404 Not Found,

1. Go to `localhost:12000/x.txt`. The file `x.txt` does not exist on the server, so it will return an erroneous response.
2. Check the response message. It will have the status line `HTTP/1.1 404 Not Found`. This is highlighted in yellow in the image below.



Test status code 408 Request Timeout

Note that this way to test may not work on your computer. If it does not, use the Python client program instead.

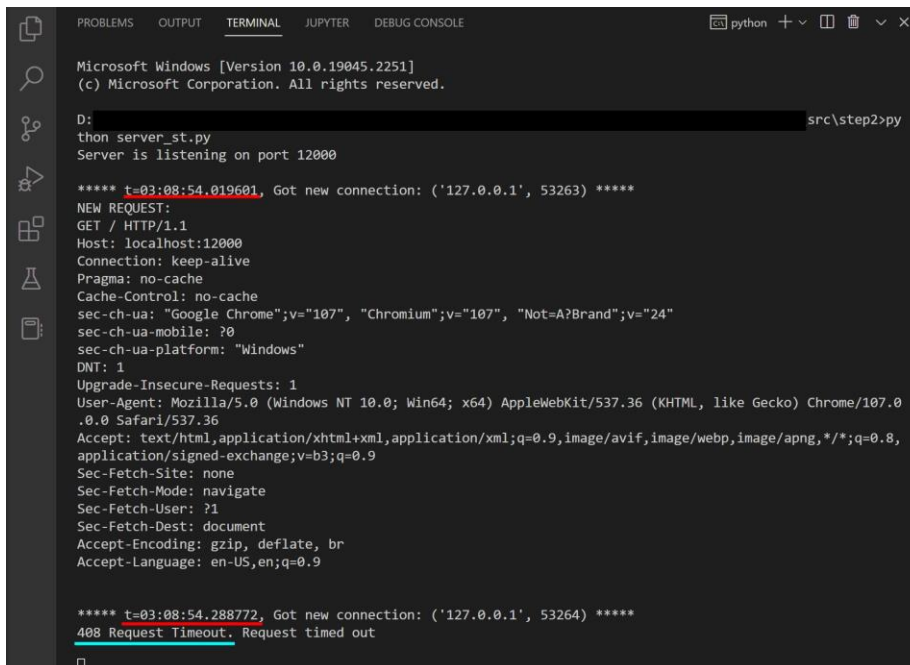
Google Chrome* can open two TCP connections when it connects to a webpage. Other people have seen this happen; [see here](#). One connection will function normally and be responded to, but the other connection will sit idle; the server will eventually timeout the connection and respond with 408 Request Timeout. Chrome, the client, can't display the timeout response, so we will look at the server's output instead.

*This behavior is also observed in Microsoft Edge, but not in Firefox.

Consider restarting the Python server program to clear its output. To test 408 Request Timeout,

1. Go to `localhost:12000`.
2. Quickly look at the server's output in the terminal you're running it in. You will see two connections being opened very quickly (the time when connections are opened are underlined in **red**).
 - a. The 1st connection has a request with a [start line](#) of `GET / HTTP/1.1`. This is the client's request for `index.html`; the server handles this by responding with the requested file.
 - b. The 2nd connection is of interest; it does not have a request, so it will just sit idle until the server shuts the connection down. Wait a few seconds, and the server will output `408 Request Timeout` (underlined in **blue**), indicating the shutdown connection.

Note that there may be another connection before the timed-out one that has a request with a start line of `GET /favicon.ico HTTP/1.1`. This is Chrome requesting `favicon.ico`. Ignore this connection.



```
Microsoft Windows [Version 10.0.19045.2251]
(c) Microsoft Corporation. All rights reserved.

D:
thon server_st.py
Server is listening on port 12000

**** t=03:08:54.019601, Got new connection: ('127.0.0.1', 53263) ****
NEW REQUEST:
GET / HTTP/1.1
Host: localhost:12000
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: "Google Chrome";v="107", "Chromium";v="107", "Not=A?Brand";v="24"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
DNT: 1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0
.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,
application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9

**** t=03:08:54.288772, Got new connection: ('127.0.0.1', 53264) ****
408 Request Timeout. Request timed out
```

Testing with the Python client program

Alternatively, you can test the server using the Python client program, `client_st.py`. The client program will connect to the server and send a request based on the argument you give it (unless you pass in 408 then it sends no request). It then outputs the response message it receives from the server.

Go to the `step2` directory and run the client program in a terminal using the command `python client_st.py <code>`, where `<code>` is the status code you want to test. The valid values for `<code>` are:

- 200- tests 200 OK.
- 304- tests 304 Not Modified.
- 400nohost, 400fewpar, 400post, 400http2, 400invalid- tests 400 Bad Request.
- 404- tests 404 Not Found.
- 408- tests 408 Request Timeout.

Test status code 200 OK

In a terminal, run the command `python client_st.py 200`. This sends a normal request to the server for `index.html`. The client will receive a response with a full HTML page, and the status line will say `HTTP/1.1 200 OK`. The output is shown below.

```
D:\ ... \step2>python client_st.py 200
Request is: GET / HTTP/1.1
Host: localhost:12000

Response from server is:
HTTP/1.1 200 OK
Last-Modified: Sat, 26 Nov 2022 06:30:30 GMT
Server: Python

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Index</title>
</head>
<body>
  <main>
    <h1>
      Welcome to Python web server! &#x1F40D
    </h1>
    <p>
      You are looking at <b>index.html</b>,</p>
```

```
        last modified at 2022-11-26 6:30 am.  
</p>  
<p><a href="test.html">Click here</a> to go to test.html.</p>  
</main>  
</body>  
</html>
```

Test status code 304 Not Modified

In a terminal, run the command `python client_st.py 304`. This sends a request for `index.html` with the header `If-Modified-Since`. The value of this header equals the current time, which simulates the client having an up-to-date cached version of `index.html` on disk. The client does not receive an HTML page back as the server didn't send a response with a payload. Note that the response's status line says `HTTP/1.1 304 Not Modified`. The output is shown below.

```
D:\ ... \step2>python client_st.py 304  
Request is: GET / HTTP/1.1  
Host: localhost:12000  
If-Modified-Since: Mon, 05 Dec 2022 03:33:25 GMT  
  
Response from server is:  
HTTP/1.1 304 Not Modified  
Server: Python
```

Test status code 400 Bad Request

There are many methods to test 400 Bad Request. Each method below tests 400 Bad Request in a different way. One way to test it would be to run the command `python client_st.py 304` in a terminal.

- `400nohost`: This sends a request that doesn't have the Host header. HTTP/1.1 requires a request to have the Host header, so a request without one is bad.
- `400fewpar` - This sends a request that has a malformed start line, making it a bad request.
- `400post` - This sends a request that uses the POST method instead of GET. The server doesn't handle POST, so it's a bad request.
- `400http2` - This sends a request that uses HTTP/2 instead of HTTP/1.1. The server doesn't handle HTTP/2, so it's a bad request.
- `400invalid` - This sends a request for a filename that contains an illegal character, making it a bad request.

The client will receive a response with an HTML page that says 400 Bad Request, no matter which method you use; the status line will say HTTP/1.1 400 Bad Request. The output from using 400nohostis shown below.

```
D:\ ... \step2>python client_st.py 400nohost
Request is: GET / HTTP/1.1

Response from server is:
HTTP/1.1 400 Bad Request
Server: Python

<!DOCTYPE html><html><h1>400 Bad Request &#x1F635</h1></html>
```

Test status code 404 Not Found

In a terminal, run the command `python client_st.py 404`. This sends a request for `x.txt`, a file that doesn't exist on the server. The client will receive a response with an HTML page that says 404 Not Found; the status line will say HTTP/1.1 404 Not Found. The output is shown below.

```
D:\ ... \step2>python client_st.py 404
Request is: GET /x.txt HTTP/1.1
Host: localhost:12000

Response from server is:
HTTP/1.1 404 Not Found
Server: Python

<!DOCTYPE html><html><h1>404 Not Found &#x1F937</h1></html>
```

Test status code 408 Request Timeout

In a terminal, run the command `python client_st.py 408`. The client will connect to the server, but it will not send a request. The connection will sit idle until the server decides to shut it down a few seconds later. The client will receive a response with an HTML page that says 408 Request Timeout; the status line will say HTTP/1.1 408 Request Timeout. The output is shown below.

```
D:\ ... \step2>python client_st.py 408
Testing timeout. Doing nothing while waiting for server response...
Response from server is:
HTTP/1.1 408 Request Timeout
```

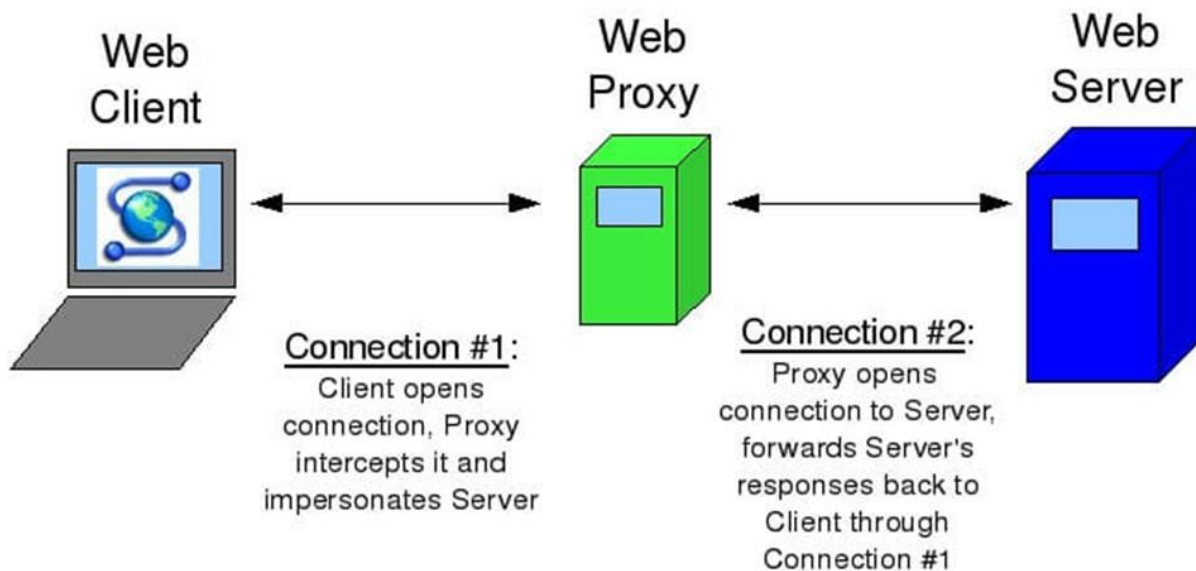

Server: Python

<!DOCTYPE html><html><h1>408 Request Timeout ⏱</h1></html>

Step Three - Web Proxy Server

The difference in request handling and detailed specifications for implementation

We know that a web request is a communication message transmitted between the client or web browsers to the servers. Generally, when the client makes a request, the request is sent to the web server. While using a proxy server, when a web request is initiated to a specific website, the request first makes a pit stop at the proxy server. Further, the request is then sent to the website, which finally sends the web page data to the web proxy server. We can say that both the request message sent by the client and the response message delivered by the web server passes through the proxy server.



The addition of a proxy server between the client and the web server improves the performance and gives a good security baseline since they act as additional data boundaries.

The main goal of a web proxy server is to satisfy client requests without involving the origin server. For this, we implement a web cache that checks if the requested information is available to it in local storage. If it is available, the response is sent back to the client with the requested information, otherwise, the cache forwards the request on behalf of the user to the origin server, retrieving the information from there and finally sending it to the client. In a way, the cache acts as both the client (to the origin server) and the server (for the original requesting client). Moreover, we also implement the conditional GET which supports not sending the object/ information if the cache has an up-to-date cached version.

Test procedures for the proxy server

To test our implementation of a minimal proxy server, we are testing the server-side functionality in the following ways:

First, run localhost (or 127.0.0.1) with port 12000, and request a web page (www.google.com) from the browser. The site could not be reached (as shown in Figure 1 below) since the proxy server isn't running (refer to the next steps).

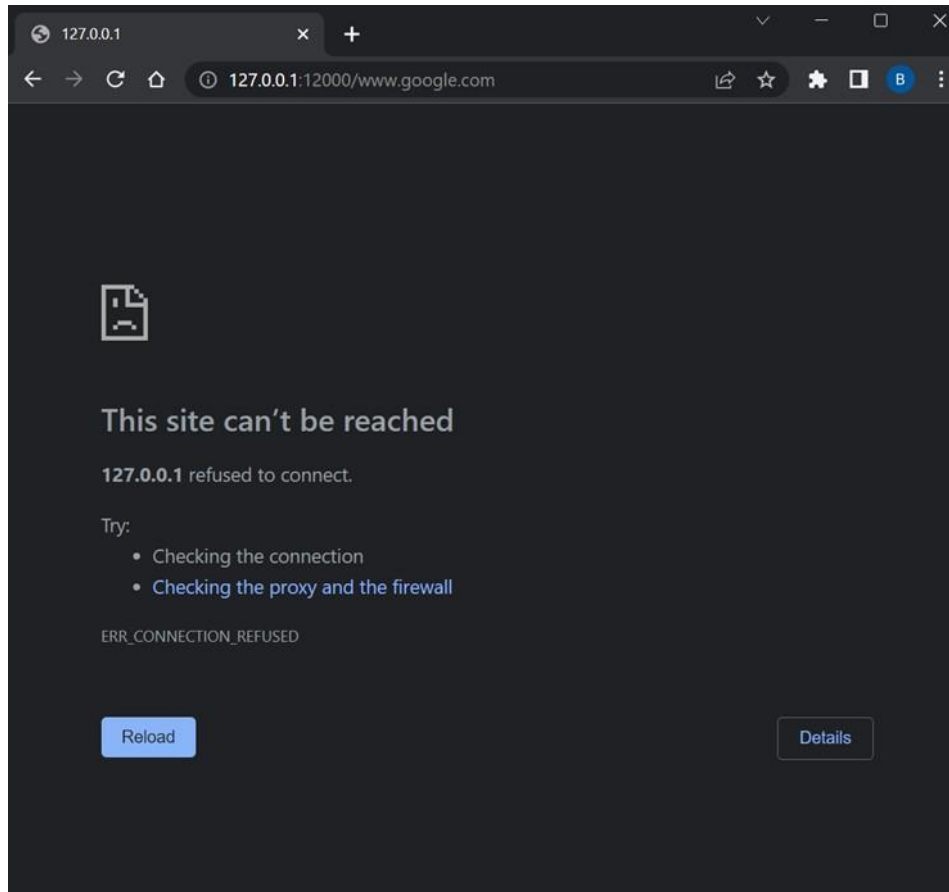


Figure 1

Now, run the given Python script `proxy_server.py` (Figure 2) in your terminal (here, we used VSCode's in-built terminal). Make sure to run it in the correct directory. Also, note that we give the port number in addition to running the python script.

```
py -3.11 proxy_server.py 12000
```

Figure 2

Figure 3 below shows the terminal response we get after running the script.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\barun\OneDrive\Desktop\MP\proxy> py -3.11 proxy_server.py 12000
12000
Ready to serve...
Received a connection from ('127.0.0.1', 63607)
message: GET /www.google.com HTTP/1.1
Host: localhost:12000
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Not?A_Brand";v="8", "Chromium";v="108", "Google Chrome";v="108"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
```

Figure 3

Finally, check your browser and refresh the webpage, or go to localhost:12000/www.google.com to access Google's web page. We receive the web page as requested thanks to the proxy server (Figure 4).

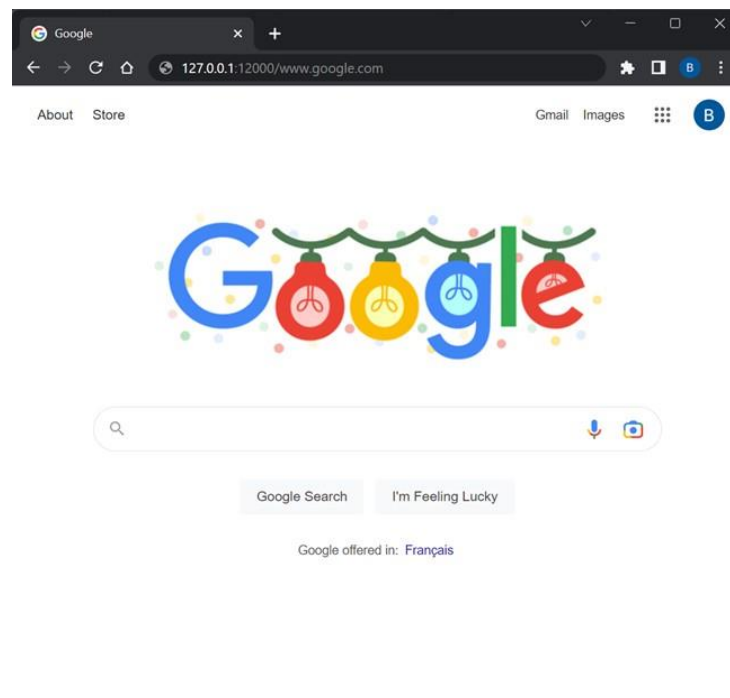


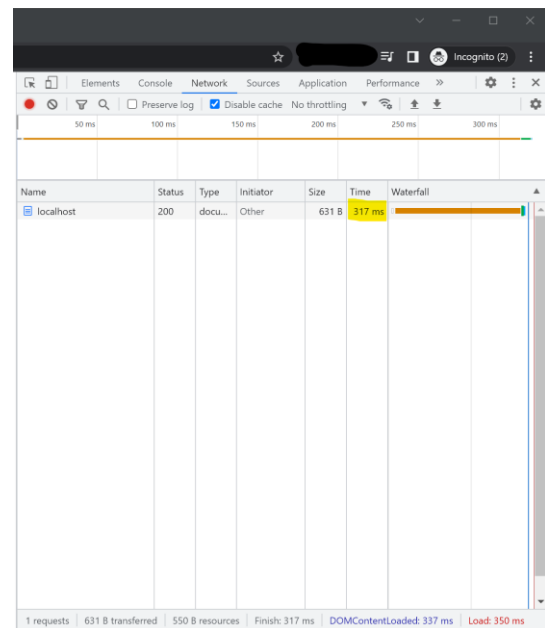
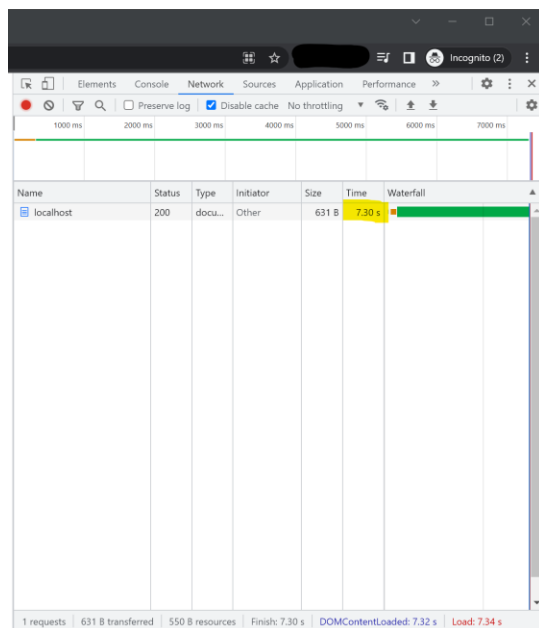
Figure 4

Step Four - Multi-threaded web server

To show off the multi-threaded functionality of step 4's multi-threaded web server, we will first show off step 2's single-threaded (ST) server limitations. The ST server can only handle one connection at a time. If the server is handling one connection and another connection comes in, the second connection must wait until the first one is finished before it can start being handled. You can see this waiting in action by following these steps:

1. Go to the step2 directory, open a terminal, and run the Python server program with the command `python server_st.py`. This will start the **single-threaded** server.
2. In a Chrome window, press Ctrl + Shift + N to open an incognito window.
3. Press F12 to open the developer tools.
4. Click on the Network tab. If you don't see the Network tab, check the dropdown menu.
5. Click on the Disable cache checkbox to disable caching.
6. Type in `localhost:12000` into the URL box, but don't press Enter yet.
7. Open a second terminal and run the command `python client_st.py 408`, then press Enter. This will open a connection with the server, using up its one connection slot.
8. Quickly switch to your Chrome incognito window, then click on the URL box and press Enter to connect to the server.
9. Watch how long Chrome must wait for its request to be handled. It will finally be handled when the client program outputs its 408 Request Timeout response from the server.

The entry for the connection in the Network tab will have a large number of seconds in the Time column (left image), showing how long Chrome had to wait. If you now refresh the incognito window, where Chrome's connection is immediately ready to be served, you'll see a much smaller time value (right image).

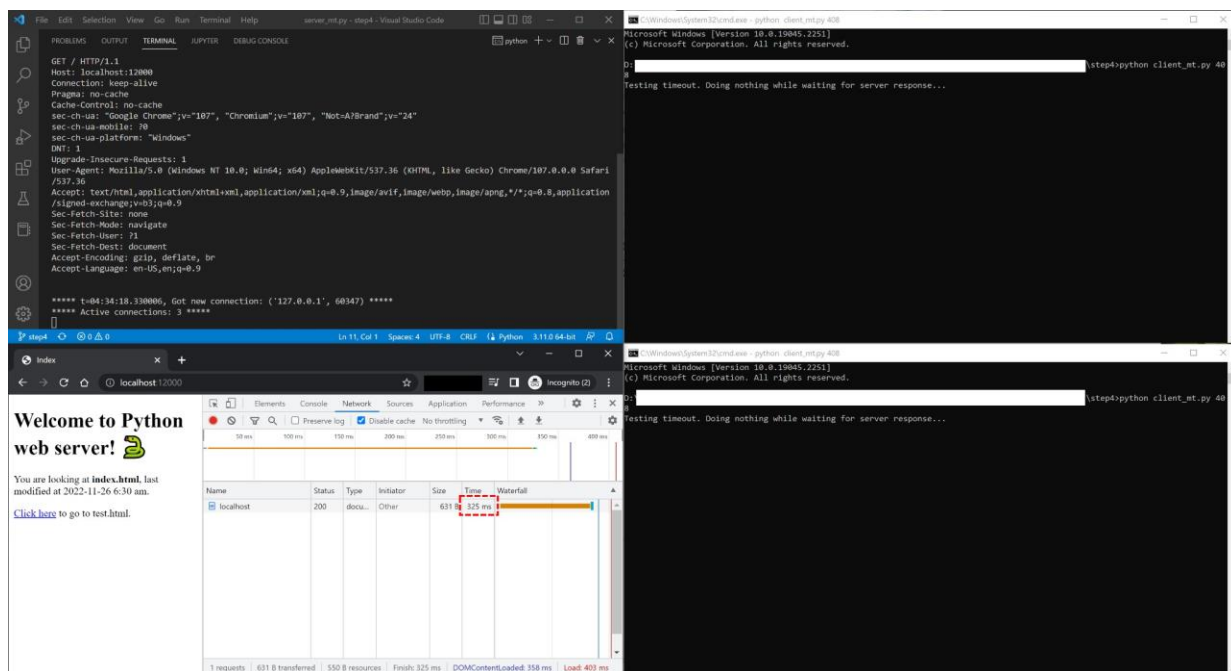


Test the multi-threaded server

Now we will show how the multi-threaded (MT) web server lack of limits. The MT server creates a new thread for each connection it receives. These threads can run in parallel, letting the server handle multiple connections at once. Follow these steps to see this in action:

1. Before starting, ensure you kill the terminal running the single-threaded server.
2. Switch the step4 directory, open a terminal, and run the Python server program with the command `python server_mt.py`. This will start the **multi-threaded** server.
3. In a Chrome window, press Ctrl + Shift + N to open an incognito window.
4. Press F12 to open the developer tools.
5. Click on the Network tab. If you don't see the Network tab, check the dropdown menu.
6. Click on the Disable cache checkbox to disable caching.
7. Type in `localhost:12000` into the URL box, but don't press Enter yet.
8. Open two terminals and type in the command `python client_mt.py 408` into both. Press Enter in one, then quickly press Enter in the other so they both open connections with the server.
9. Quickly switch to your Chrome incognito window, then click on the URL box and press Enter to connect to the server.
10. Watch how quickly Chrome's connection to the server is handled and closed. The MT server was able to do this while handling the client terminals' connections. This is the power of the MT server.

The image below shows both client terminals waiting for the server to close their connections while Chrome has already closed its connection. Chrome's response time is boxed in **red**.



Additionally, notice the output in the Visual Studio Code window (top left). The server received Chrome's request and outputted it, then closed the connection. After Chrome's request has finished, there are still 3 connections open. Two of these connections are from the client terminals, whose idle connections haven't been shut down yet. The last connection is from Chrome's second TCP connection it opens.

