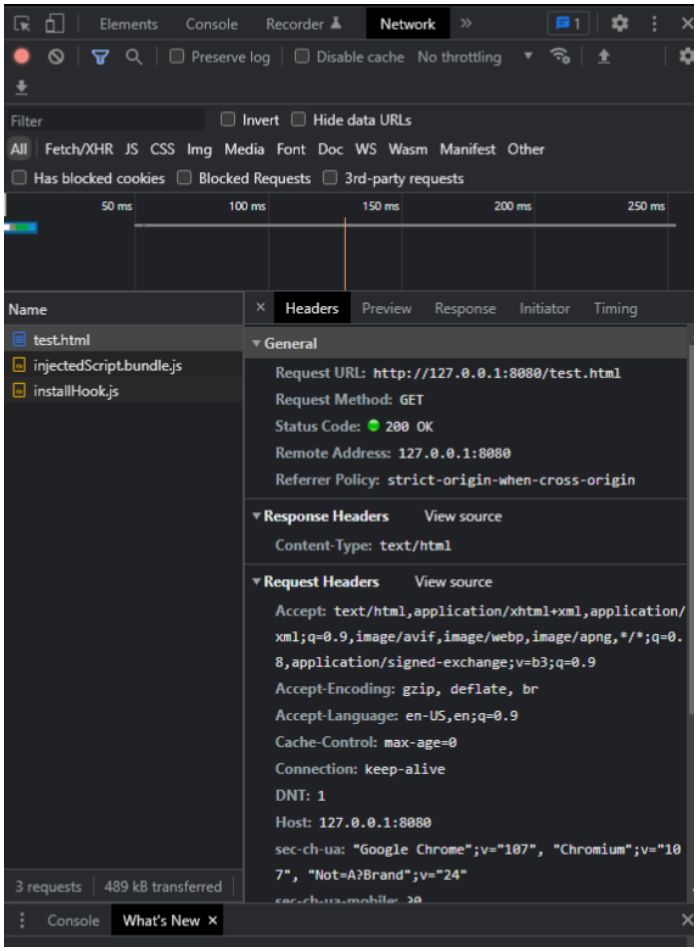
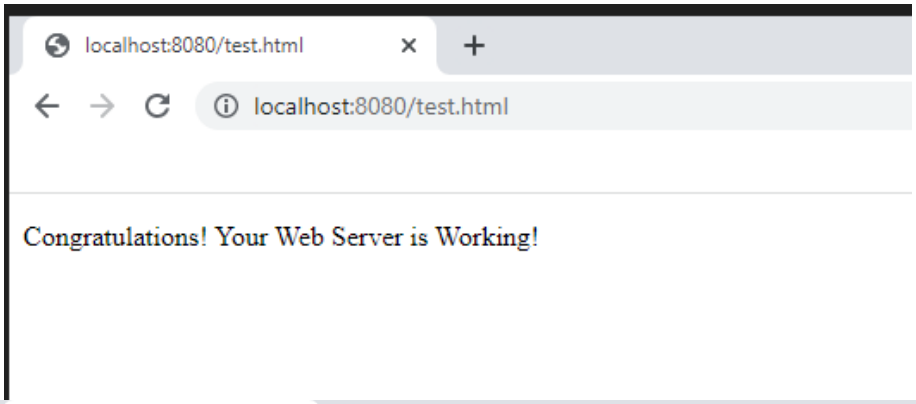
Step Two

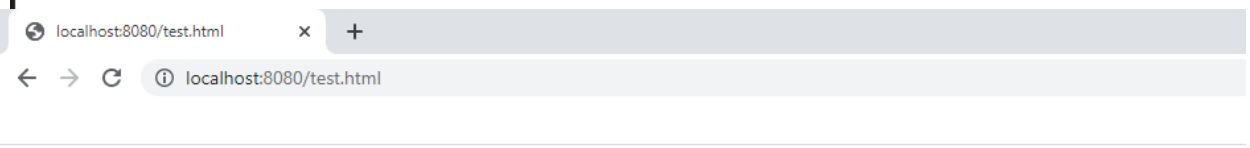
(B)

Test procedures included removing the file from the server (404), we slept the connection and kept track of a start time (408), let the file specified be on the server (200 OK), and replicated a bad request (400). We could not replicate a not modified, as refreshing in chrome clears the cache of that webpage. To replicate, we would have to use Postman or some other API for custom requests, so we deemed the error handling for 308 to be correct based on examples we found online. Below is the inspect element of our server where upon refresh, there is only one GET request, which means cache was cleared upon refresh of the browser.

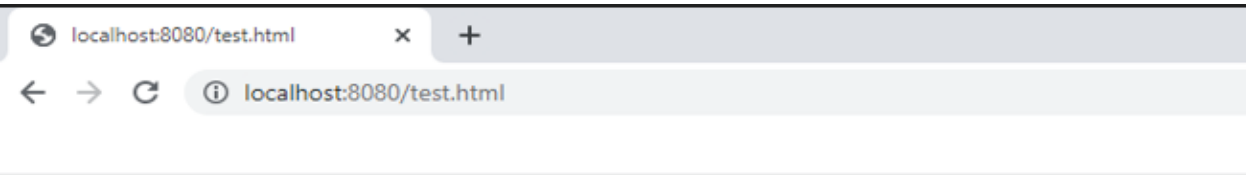


Screenshots of messages:

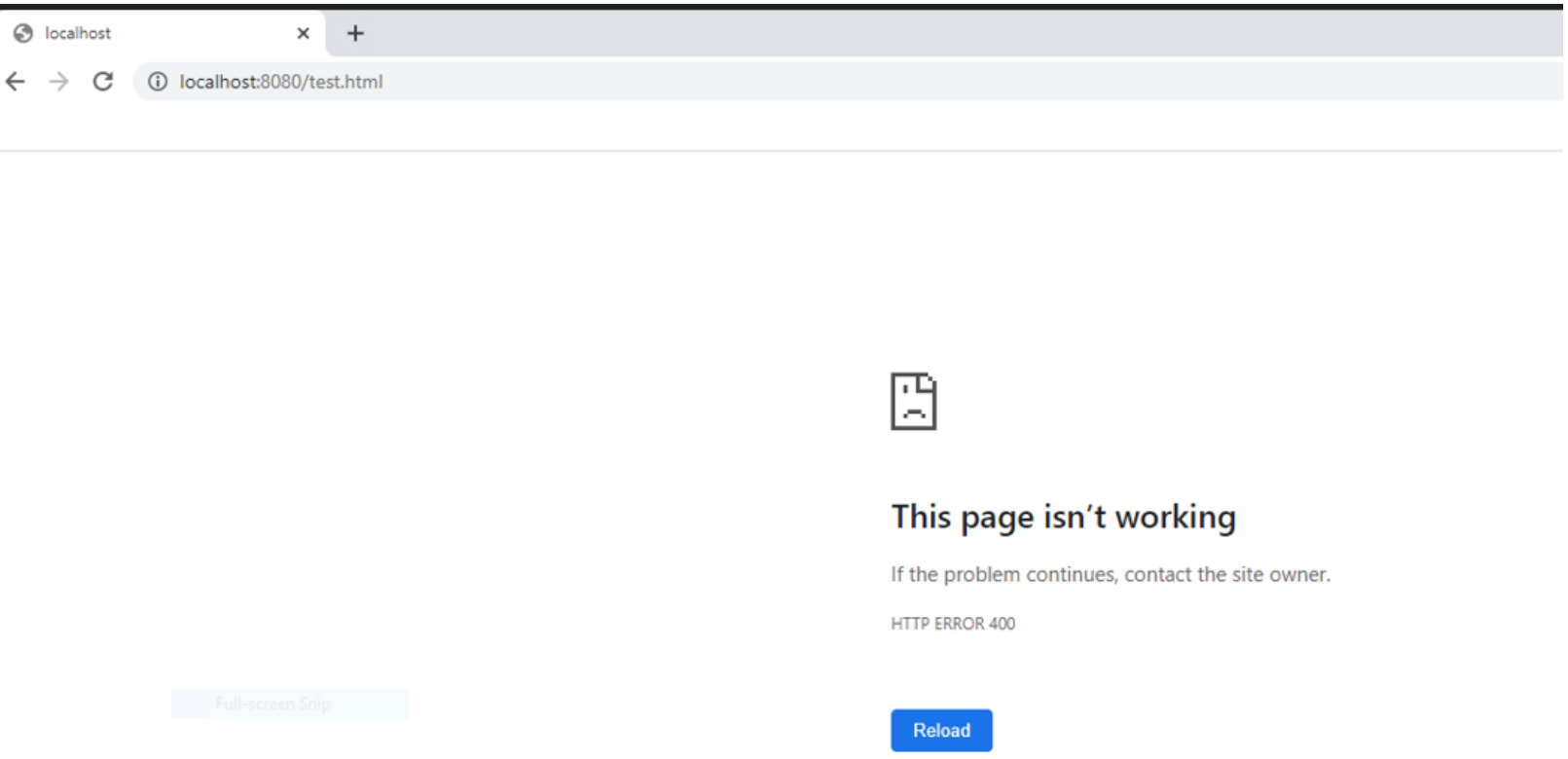




408 Request Timeout



404 Not Found


localhost
localhost:8080/test.html
This page isn't working
If the problem continues, contact the site owner.
HTTP ERROR 400
Reload

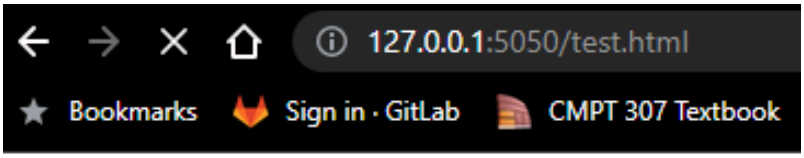
For the second part of your mini project, think about a web proxy server. What is different in request handling in a proxy server and a web server that hosts your files? Write down the detailed specifications you come up with for a minimal proxy server only using the knowledge you have from module (2) slides 29-34 (2 points),

The proxy server acts as a middleman between a client and the server providing the client with a gateway. Instead of requests having to go all the way from the client to the server, it only has to go to the proxy. It acts as both client and server. To implement a basic proxy server, it should be able to handle requests from clients and take those requests from the server as well as the opposite, which is taking a response from the server and passing it on to the client. Ideally, it should be able to cache objects as well as update them when able too, and pass these cached objects back and forth. In our implementation, we decided to forego the caching as we felt demonstrating a simple proxy server was mostly about the stream of data, and wanted to showcase how our proxy was both a client and a server.

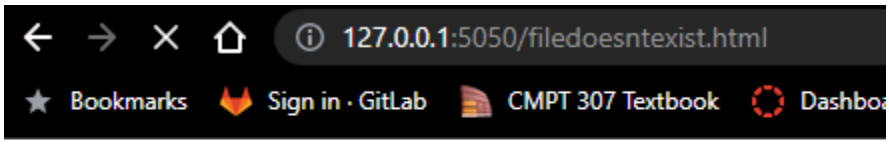
(b) Decide the test procedure to show the working of your proxy server. Does this need possible changes at the client side? If your answer is yes, you are not required to implement them, but need to describe and find alternative ways to test your server side functionality. Print screen, or cut and paste output and document your test procedures to show. (2 points)

To test the proxy server, we first ran MiniProject.py, which was the server from step 2. From here, we ran our proxy server on a different port using it as a python script in the terminal. Then, in our browser, we used the ip address and port for the proxy server, and requested the test.html document that was on the server from step 2. Since this is effectively a get request, and the file existed on the server, we were shown the file we requested. Note that the port number in the url in the screenshots below is the port we connected to in our proxy server, and that the main server was on port 8080. We also tried to request files that didn't exist, and make requests purposefully timing out by sleeping the request to the main server.

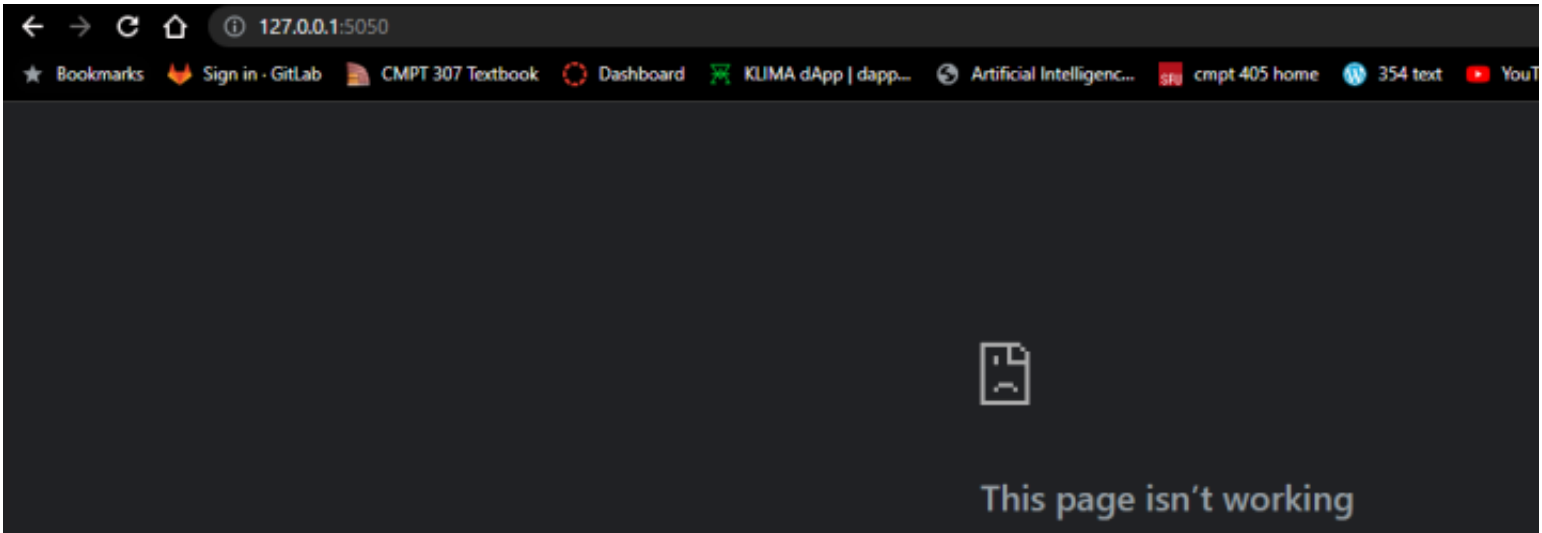
```
proxy listening on 127.0.0.1 5050
Client connected to proxy on 127.0.0.1, port:  5050
sending data to server at 127.0.0.1, 8080
proxy recieved response from the server!
proxy sent data to client
Client connected to proxy on 127.0.0.1, port:  5050
sending data to server at 127.0.0.1, 8080
proxy recieved response from the server!
proxy sent data to client
```



Congratulations! Your Web Server is Working!



404 Not Found



Bonus: Step Four (2 points):

(b) Decide the test procedure to show the multi-threaded nature of your server. Print screen, or cut and past output and document your test procedures to show. (1 point)

After a thread connected, we slept the thread before closing it and then opened a new connection, and the port number changed as there was still a prior active connection. The port numbers range from 8095 to 8081, and a thread starts at 8095 and pops it off the stack. When it's done, it pops it back on the stack. This means that we can have 14 ports active at once and when a thread is done the port goes back to the top of the stack. This can help with server load by making sure ports are re-used when a connection is closed. Since a thread now executes handle_client(), the error handling which is called in the function is unchanged, ensuring our tests of our "single-thread" server for http codes would hold up for our multi-thread server.

```
PS C:\Users\Todor\Desktop\CMPT_371> &
Listening on IP 127.0.0.1, port 8080.
8095
client on IP 127.0.0.1, Port: 8095
8094
client on IP 127.0.0.1, Port: 8094
8095
client on IP 127.0.0.1, Port: 8095
8094
client on IP 127.0.0.1, Port: 8094
8093
client on IP 127.0.0.1, Port: 8093
```

```
time.sleep(5)

client_socket.close()
prt_arr.append(newport)
return
```