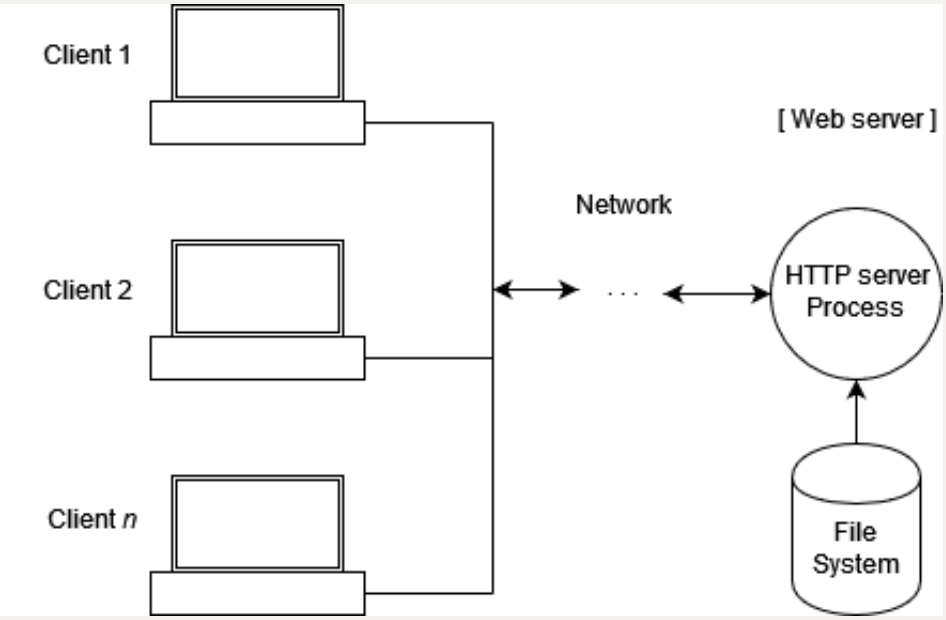
CS305 2022 Fall Assignment 1 - HTTP Server

Introduction

An **HTTP server** is a computer(software) program that plays the role of a server in a C-S model by implementing the server part of the HTTP and/or HTTPS network protocol(s).



The goal of this assignment is to implement an **HTTP/1.1 server** like flask based on the given framework. To achieve that goal, you need to follow the instructions, write codes, and pass the given Python unit tests at last.

HTTP/1.1 RFCs

A **Request for Comments** (**RFC**) is a publication in a series from the principal technical development and standards-setting bodies for the Internet.

In this assignment, you are encouraged to read some chapters of the following RFCs.

- MDN docs
- RFC9110: HTTP Semantics
- RFC7230 : HTTP/1.1 Message Syntax and Routing
    - Chapter 3 : Message Format
    - Chapter 6 : Connection Management
- RFC7231 : HTTP/1.1 Semantics and Content
    - Chapter 4 : Request Methods
    - Chapter 6 : Response Status Codes
        - Only 200, 300, 301, 400, 404
- RFC7232 : HTTP/1.1 Conditional Requests
- RFC7233 : HTTP/1.1 Range Requests
- RFC7234 : HTTP/1.1 Caching

If you are confused about syntax in RFCs, you can try to read the `Syntax Notation` chapter of each RFC. Here are some instances for that notation.

- SP: whitespace
- CRLF: '\r\n'

Tips: About how to read RFC, here is an article about it. How to Read RFC

Tasks

Task 1: HTTP Message encapsulation and de-encapsulation (20 pts)

In task1, you need to complete the code in the framework, and directly manipulate of binary data and sockets to handle HTTP messages.

The methods that need to complete:

- `framework.py` -> `class HTTPResponse` -> `write_all()`
- `class HTTPRequest` -> `read_headers()`.

After finishing this part, you can use curl to test your HTTP server.

Run the server with command `python3 main.py` and run `curl -v http://127.0.0.1:8080` **in a new terminal**. After that, you should get `404 Not Found` as intended.

To simplify the implementation of the HTTP server, we add header `Connection: close` to require clients not to reuse TCP connections.

```
$ curl -v http://127.0.0.1:8080/
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< Connection: close
<
* Closing connection 0
```

At last, we provide the unit-test `TestTask1.py` for you to check your implementation.

Task 2: Basic Static Content Server (20 pts)

In task 2, you are required to write handler to serve local files when clients GET for any files under `/data/`.

For example, if clients GET for `http://127.0.0.1:8080/data/index.html`, the server should respond with the content of file `data/index.html` . If clients GET for a non-existence file, such as `http://127.0.0.1:8080/data/nosuchfile`, the server should respond HTTP 404 Not Found.

You should reject the POST requests of them, and respond properly to GET and HEAD requests. Also, you should set the `Content-Type` and `Content-Length` in Response Headers properly.

Open `http://127.0.0.1:8080/data/index.html` in browser, you should see the rendered HTML, a picture and a javascript alert.

Route: Use different handler for different URL path

This framework contains a very simple Route module, which allows you to register different handlers for different URL path, and the Route module looks for the handler with the **longest** matched URL path.

```
def default_handler(server: HTTPServer, request: HTTPRequest,
response: HTTPResponse):
    # default handler for unmatched requests
    response.status_code, response.reason = 400, 'Bad Request'
    print(f"calling default handler for url
{request.request_target}")

def task2_file_handler(server: HTTPServer, request: HTTPRequest,
response: HTTPResponse):
    # independent handler for URL like '/data/<any file>`
    # This handler will be called when clients are requesting
/data/...
    # TODO: Task 2: Serve local files based on request URL
    pass

# ...
http_server.register_handler("/", default_handler)
# Register your handler here!
http_server.register_handler("/data", task2_file_handler,
allowed_methods=['GET', 'HEAD'])
```

At last, we provide the unit-test `TestTask2.py` for you to check your implementation.

Task 3: Handle POST Request (20 pts)

If you are not so familiar with what POST is, please refer to this document. https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

In task 3, you only need to complete method `read_message_body()` in `class HTTPRequest`. Please be careful about binary boundary!

It is guaranteed that the header `Content-Length` **always exists** in every POST in this assignment if the HTTP message body exists, and the length of HTTP message body equals to its value.

The client will POST a json containing a key named `data` and other junk keys like the json belows. You should store the content of `data` into `server.bucket`. It is guaranteed that the key `data` is present in the json of each test case.

```
{
  "junk1": "...",
  "data": "<random data>",
  "junk2": "..."
}
```

When the client GETs `http://127.0.0.1:8080/post`, you should return the stored data from **last POST**.

```
{
  "data": "<random data>"
}
```

When you implement it, the result will be like what is shown below.

```
$ curl -v http://127.0.0.1:8080/post --data '{"data":"test", "junk":
"ignore"}'
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> POST /post HTTP/1.1
> Host: 127.0.0.1:8080
```

```
> User-Agent: curl/7.85.0
> Accept: */*
> Content-Length: 18
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
<
* Closing connection 0
$ curl -v http://127.0.0.1:8080/post --get
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /post HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
< Content-Type: application/json
< Content-Length: 16
<
* Closing connection 0
{"data": "test"}
```

At last, we provide the unit-test `TestTask3.py` for you to check your implementation.

Task 4: HTTP 302 Found: URL Redirection (10 pts)

If you are not so familiar with what Redirection is, please refer to this document. https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections.

URL Redirection is used to redirect browsers to another URL. The HTTP Server should set status code to `30x` and set the `Location` attribute in Header.

After implementation, when the client GETs/HEADs `http://127.0.0.1:8080/redirect`, the HTTP server generates `302` and redirects to `http://127.0.0.1:8080/data/index.html`.

Curl Result

```
$ curl -v http://127.0.0.1:8080/redirect
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /redirect HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 302 Found
< Connection: close
< Location: http://127.0.0.1:8080/data/index.html
<
* Closing connection 0
```

At last, we provide the unit-test `TestTask4.py` for you to check your implementation.

Task 5: HTTP Cookie and Session (30 pts)

> *An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests. Typically, an HTTP cookie is used to tell if two requests come from the same browser—keeping a user logged in, for example. It remembers stateful information for the stateless HTTP protocol.*

If you are not so familiar with what Cookie is, please refer to this document.https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

In task 5, you are required to protect a image file that only authenticated users can access.

In the first step, the client will POST their credential to a login API, you should verify it and mark them as authenticated or not. Since HTTP protocol is stateless, Cookie is used to keep the state of clients.

In the second step, the client will request the protected image with Cookie set by server in the first step, you should only respond the protected image to authenticated client.

In this part, you don't need to care CORS problem, and you can just ignore `Domain`, `Path` attributes in Cookie.

Cookie (15 pts)

After receiving an HTTP request, a server can send one or more Set-Cookie headers with the response. After receiving the cookie, browser usually stores the cookie and sends it with requests made to the same server inside a Cookie HTTP header.

Step 1. Login Authentication

The client will POST `http://127.0.0.1:8080/api/login` with the following json structure, the server should verify whether the `username` and `password` are equal to admin, admin.

```
{"username":"admin", "password":"admin"}
```

If they are equal, the server should return HTTP 200 OK and set cookie `Authenticated=yes`. Otherwise, it should return `HTTP 403 Forbidden`.

To simplify code, we could ignore any other attributes of Cookie.

Step 2. Access Protected Resources

The client will GET/HEAD `http://127.0.0.1:8080/api/getimage`.

If the client authorizes in Step1, it requests this URL with Cookies specified in the last Response, i.e., the `Authenticated` field in Request Cookie should be `yes`.

If the `Authenticated` field is `yes`, the server should return the image `data/test.jpg` and set `Content-Length` and `Content-Type` properly. Otherwise, if the `Authenticated` Cookie doesn't exist or isn't `yes`, it should return `HTTP 403 Forbidden`.

You should pass unittest `TestTask5Cookie.py` if you implement this part properly. You can also open `http://127.0.0.1:8080/api/test` in browser to test.

Curl Result

```
$ curl -v http://127.0.0.1:8080/api/login --data
'{"username":"admin", "password":"admin"}'
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> POST /api/login HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
> Content-Length: 40
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
< Set-Cookie: Authenticated=yes
<
* Closing connection 0

$ curl -v http://127.0.0.1:8080/api/getimage
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /api/getimage HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 403 Forbidden
< Connection: close
```

```
<
* Closing connection 0

$ curl -v http://127.0.0.1:8080/api/getimage --header "Cookie:
Authenticated=yes"
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /api/getimage HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
> Cookie: Authenticated=yes
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
< Content-Length: 44438
< Content-Type: image/jpeg
<
Warning: Binary output can mess up your terminal. Use "--output -" to
tell
Warning: curl to output it to your terminal anyway, or consider "--
output
Warning: <FILE>" to save to a file.
* Failure writing output to destination
* Closing connection 0
```

After that, you should pass unit-test `TestTask5Cookie.py`. You can open `http://127.0.0.1:8080/api/test` in browser to play with your server.

Session (15 pts)

The authentication method in the above method is not secure, a malicious client can access the protected resources by just setting the cookie `Authenticated=yes` without knowing the real username and password credential.

To implement a session, the server initializes a dictionary, which uses a random string as a key, and stores data in the dictionary that is not accessible to clients.

The server will set this random string as session key in response's `Set-Cookie`, the client will bring the given cookie in future requests, and the server will find the stored data via the key in Cookie.

Step 1. Login Authentication

The client POSTs `http://127.0.0.1:8080/apiv2/login` with the following json structure, The server should verify whether both the `username` and `password` are equal to `admin`.

```
{"username":"admin", "password":"admin"}
```

If they are equal, the server should generate a random string as `SESSION_KEY`, and make sure it's unique in stored session keys. The server should return `HTTP 200 OK` and set cookie `SESSION_KEY=<random string>`. Otherwise, it should return `HTTP 403 Forbidden`.

Step 2. Access Protected Resources

The client will GET/HEAD `http://127.0.0.1:8080/apiv2/getimage` with or without `SESSION_KEY` in Cookies.

If the client authorizes successfully in Step1, it requests this URL with Cookies included in the last response.

If the `SESSION_KEY` field exists and server recognizes it as valid, the server should return the image `data/test.jpg` and set `Content-Length` and `Content-Type` properly. Otherwise, the server should return `HTTP 403 Forbidden`.

You should pass unit-test `TestTask5Session.py` if you implement this part properly. You can also open `http://127.0.0.1:8080/apiv2/test` in browser to test.

Curl Result

```
```

```
$ curl -v http://127.0.0.1:8080/apiv2/getimage
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /apiv2/getimage HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 403 Forbidden
< Connection: close
<
* Closing connection 0
$ curl -v http://127.0.0.1:8080/apiv2/login --data
'{"username":"admin", "password":"admin"}'
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> POST /apiv2/login HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
> Content-Length: 40
> Content-Type: application/x-www-form-urlencoded
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
< Set-Cookie: SESSION_KEY=DJK5LAFTY8NEQN6JNRIA
<
* Closing connection 0
$ curl -v http://127.0.0.1:8080/apiv2/getimage --header "Cookie:
SESSION_KEY=DJK5LAFTY8NEQN6JNRIA"
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /apiv2/getimage HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.85.0
> Accept: */*
> Cookie: SESSION_KEY=DJK5LAFTY8NEQN6JNRIA
```

```
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Connection: close
< Content-Length: 44438
< Content-Type: image/jpeg
<
Warning: Binary output can mess up your terminal. Use "--output -" to
tell
Warning: curl to output it to your terminal anyway, or consider "--
output
Warning: <FILE>" to save to a file.
* Failure writing output to destination
* Closing connection 0
```

At last, we also provide the unit-test `TestTask5Session.py` for you to check your implementation in this part. Please notice that, the test files in the `/data` directory will be different in our testing environment.

You can open `http://127.0.0.1:8080/apiv2/test` in browser to play with your server.

How to run the code

Server

Run `python main.py` in the root directory of the project, then run `curl -v http://127.0.0.1:8080` **in new terminal**.

Unit Test

VSCode

Run `python -m unittest tests.TestTask1` in the root directory of the project to execute `TestTask1.py` .

For other tests, the execution commands are similar.

PyCharm

Go to the root directory of the assignment, then execute tests.

What to submit

You must provide **a zip file** of your implementation, including `main.py`, `framework.py` and other files if needed.

A PDF file that includes some necessary screenshots to show your code works and some brief explanations.

Score

The score of your assignment will be determined by an **extended** set of given unittests. You can use the given unittests to check the correctness of your implementation. But you still might not get 100 points, even if you pass all the given unittests.

ATTENTION!!!

Please modify `YOUR_STUDENT_ID = 12010000` to your SID in `main.py`, this field is used in automated testing. Otherwise, SAs will be painful to handle your submission.

Score Environment

Please notice that your code will eventually be tested and scored on a platform based on the following environments.

- Python 3.9
- **LIB LIMITATION**: You can only use The Python Standard Library, **excluding** the http module.

Dockerfile to build the test environment (You don't need to build this environment locally):

```
FROM python:3.9-bullseye
RUN apt update && apt install -y curl
RUN pip3 install requests
VOLUME /score
WORKDIR /score
CMD ["/bin/bash", "-c", "python3 -m tests.TestAll >result.txt
2>stderr.txt"]
# Build : docker build . -t ass1
# Run   : docker run -v $PWD:/score -it ass1:latest
```

Q&A Link

If you have any question about this assignment, please go to the link above to raise your question. And we will check and reply in time.

https://github.com/yuk1i/cs305-2022fall-homework1-student/issues

Enjoy yourselves!