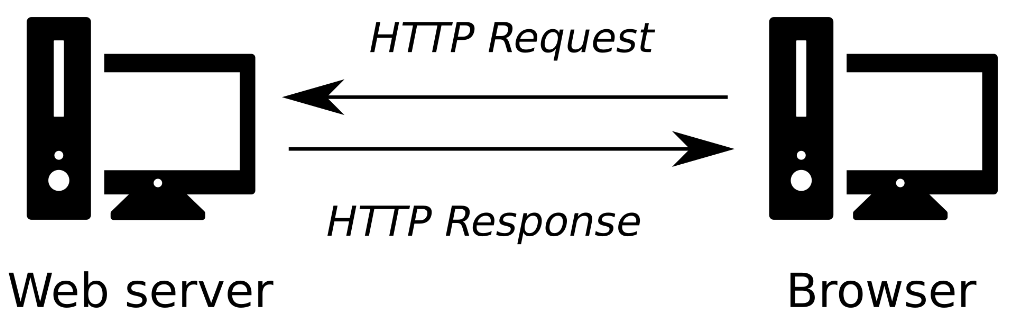

ASSIGNMENT 01

HTTP SERVER

Due Date:
October 4, 2024 23:55:59

`0d 9h 6m 38s left`

ACADEMIC DISHONESTY

Assignments will be run through a similarity checking software to check for code that looks very similar to that of other students. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or reported to the Dean's Office. Plagiarism is a serious offence. Work is to be done **individually.**

If you want to store a PDF version of this assignment, press `Ctrl+p` *on Windows or* `Command+p` *on Mac, the print window will appear. Then, select* `Save As PDF` *from the* ***Destination*** *dropdown. Then click* `Save`

This file was last modified on 2024-09-19 8:06 pm ET

UPDATES and CHANGES

The following are the updates/changes made to this assignment AFTER it was posted:

No changes have been made!

TOP

Anouncements and Important Notes

The following are the announcements and important notes for this assignment AFTER it was posted:

No announcements have been made!

1 Introduction

In this assignment, you will practice the following programming concepts:

- Implementing a basic HTTP server.
- Handling GET and POST requests.
- Managing client sessions.
- File manipulation and string replacement.
- Using sockets for network communication.

In this assignment, you will implement an HTTP server that can handle basic **GET** and **POST** requests. Your task is to create a server that listens for incoming requests, processes them, and responds with the appropriate data. The server will serve static files from a specified directory and dynamically replace placeholders in HTML files based on client information. The server will also manage client sessions by tracking the client's name and updating it via a POST request. The server should be able to handle multiple clients concurrently and gracefully shut down after a period of inactivity.

You are required to implement a single class, `Server`, which will handle all the server operations, including starting the server, handling requests, and managing client sessions.

Notes

- You are required to adhere to the class structure, method descriptions, and naming conventions for both methods and instance variables as outlined in the sections below.
- Feel free to implement your own logic of the methods as long as they fulfill the requirements specified in the assignment.
- You can add additional methods or instance variables to the class if needed for your implementation.

TOP

- Ensure that your server can handle multiple clients concurrently and gracefully shut down after a period of inactivity.
- When developing the methods, it is efficient to use try-except blocks to handle exceptions or errors that may occur in your code.
- Make sure to test your server implementation thoroughly to ensure it works as expected and handles various scenarios correctly.

2 Implementation

For this assignment, you must submit a single file named `server.py`. This file should contain your implementation of the HTTP server. Below is a detailed explanation of the components you need to include:

Server Class

__init__(self, addr, port, timeout)

Instance variables:

- `addr`: The server's IP address or hostname.
- `port`: The port number the server will listen on.
- `timeout`: The time in seconds before the server closes due to inactivity.
- `sessions`: A dictionary to store client sessions, mapping client addresses as keys to their names as values.
- `server_socket`: The server's TCP socket object.

This constructor initializes the `server` class with the specified `addr`, `port`, and `timeout` values. It initializes the `sessions` dictionary to store client sessions. it also intializes the `server_socket` object and bind it to the given `addr` and `port` to listen on. You can add any additional instance variables you need for the server's operation.

start_server(self)

The method should run the server indefinitely, accepting incoming connections and handling them in separate threads. The method should also track the last time a connection was made, if no new connections are made within the specified `timeout` period, the loop stops and the server should close by calling the `stop_server()` method.

stop_server(self)

This method should close the server's socket and terminate the server's operation.

parse_request(self, request_data)

TOP

This method is responsible for parsing the raw HTTP request data `request_data` into request line, headers, and body by splitting it using the delimiter (`\r\n`). The first line, known as the request line, contains the HTTP method, the requested path, and the HTTP version. The method then iterates through the subsequent lines to capture the headers, storing each header and its corresponding value in a dictionary. If a blank line is encountered, it indicates the end of the headers, and the remaining part of the request is treated as the body, which may contain data like form submissions in the case of POST requests. The method returns the `request line`, `headers`, and `body`.

handle_request(self, client_socket)

This method is responsible for processing incoming HTTP requests from clients. It manages the reception of request data, parsing of the request, and invoking the appropriate handler methods based on the HTTP method (GET, POST, etc.). Below is a step-by-step description of its operation:

- **Receiving Request Data:** The method starts receiving the HTTP request data from the client socket. Make sure that the data includes the entire request.
- **Extracting Request Details:** The method calls `parse_request()` to break down the request data into its components: the request line, headers, and body. The path to the requested resource, the HTTP method, and the HTTP version are then extracted from the request line. If no path is specified, the server defaults to serving `index.html`.

  If the method is GET, the method calls `handle_get_request()` to serve the requested file. If the method is POST, the method calls `handle_post_request()` to process the form data or other POST content.

  If the method is neither GET nor POST, the method calls `handle_unsupported_method()`. After processing the request, the method closes the client socket to terminate the connection.

handle_get_request(self, client_socket, file_path)

This method processes incoming HTTP GET requests. It first retrieves the client's address and checks if the requested file is present. If the file is found, it reads the file content, replaces the placeholder `{{name}}` with the client's name, and prepares a successful HTTP response with a **"200 OK"** status. If the file is not found, it prepares a **"404 Not Found"** error page as the response. The method then constructs the necessary HTTP headers, including the content type and length, and finally by using the socket connection, sends the full HTTP response back to the client.

handle_post_request(self, client_socket, path, headers, body)

This method processes incoming HTTP POST requests. It begins by retrieving the client's address and parsing the POST request's body to get the form data. If the given `path` of the request is `change_name`, it updates the client's name in the `sessions` dictionary with the provided value from the form data. The method then prepares a successful HTTP response by responding with a **"200 OK"** status and a message like **"Name updated"** within the response body. For any other paths, the method returns a **"404 Not Found"** status with an appropriate error message. Finally, the method constructs and sends the complete HTTP response, including necessary headers like content type and length, back to the client.

handle_unsupported_method(self, client_socket, method)

TOP

This method is responsible for handling HTTP requests that use a method not supported by the server. When an unsupported method is detected, the method generates a **"405 Method Not Allowed"** response. The response body contains an HTML message informing the client that the method used is not allowed. Finally, the method constructs and sends the complete HTTP response, including necessary headers like content type and length, back to the client.

3 Attachments

1. Create a directory named `assets` in the same directory as your `server.py` file.
2. `index.html`: This file is placed in the `assets` directory and serves as the default page for the server. It contains a simple HTML form that allows the client to enter their name. The form submits a POST request to the server to update the client's name and reload the page with the updated name.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Welcome Page</title>
</head>
<body>
    <h1>Hello {{name}}</h1>
    <form action="/change_name" method="post">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

3. The `server.py` file should contain the following code structure:

```
import socket
import threading
#import more modules if needed

class Server:
    def __init__(self, addr, port, timeout):
        pass

    def start_server(self):
        pass

    def stop_server(self):
        pass

    def parse_request(self, request_data):
        pass

    def handle_request(self, client_socket):
        pass

    def handle_unsupported_method(self, client_socket, method):
        pass

    def handle_get_request(self, client_socket, file_path):
        pass

    def handle_post_request(self, client_socket, path, headers, body):
        pass
```

4. To assist you in testing your implementation, you can run `test_server.py` to simulate client requests and check the server's responses. Ensure the file is in the same directory as your `server.py` file.

```
import socket, threading, time
from server import Server
addr = '127.0.0.1'
port = 8080

def test_1():# Test 1: Check if the server is running by connecting to it
    with socket.create_connection((addr, port)) as client_socket:
        cond = client_socket is not None
    if cond:print("Test 1 passed")
    else:print("Test 1 failed")

def test_2():# Test 2: Test a simple GET request
    with socket.create_connection((addr, port)) as client_socket:
        client_socket.sendall(b"GET / HTTP/1.1\r\nHost: localhost\r\n\r\n")
        response = client_socket.recv(4096).decode()
        cond = "200 OK" in response and "Hello" in response
    if cond:print("Test 2 passed")
    else:print("Test 2 failed")

def test_3():# Test 3: Test a request for a nonexistent file
    with socket.create_connection((addr, port)) as client_socket:
                client_socket.sendall(b"GET  /nonexistent.html  HTTP/1.1\r\nHost:
localhost\r\n\r\n")
        response = client_socket.recv(4096).decode()
        cond = "404 Not Found" in response
    if cond:print("Test 3 passed")
    else:print("Test 3 failed")

def test_4():# Test 4: Test a simple POST request
    with socket.create_connection((addr, port)) as client_socket:
                  client_socket.sendall(b"POST  /change_name  HTTP/1.1\r\nHost:
localhost\r\nContent-Length: 9\r\n\r\nname=Alice")
        response = client_socket.recv(4096).decode()
    with socket.create_connection((addr, port)) as client_socket:
        client_socket.sendall(b"GET / HTTP/1.1\r\nHost: localhost\r\n\r\n")
        response = client_socket.recv(4096).decode()
        cond = "200 OK" in response and "Alice" in response
    if cond:print("Test 4 passed")
    else:print("Test 4 failed")

if __name__ == "__main__":
```

TOP

```
    try:
        server = Server(addr, port, 5)
        server_thread = threading.Thread(target=server.start_server)
        server_thread.start()
        time.sleep(1)
    except Exception as e:
        print(e)
    test_1()
    test_2()
    test_3()
    test_4()
    try:
        server.stop_server()
        server_thread.join()
    except Exception:pass
```

5. The following video demonstrates testing the server implementation for updating the client's name using a POST request and serving static files using GET requests.

– Submission

REMEMBER!

- You have 4 coupons (for the entire semester) that will be automatically applied when you submit late.

TOP

> - It is the student's responsibility to ensure the work was submitted and posted in GradeScope.
> - Any assignment not submitted correctly will not be graded.
> - Submissions through the email will not be accepted under any circumstances.
> - Please check this page back whenever an announcement is posted regarding this assignment.
> - Marks will be deducted if you submit anything other than the required Python files.
> - Submit the assignment on time. Late submissions will penalty unless you have enough coupons to be applied (automatically).
> - You may re-submit your code as many times as you like. Gradescope uses your last submission for grading by default. There are no penalties for re-submitting. However, re-submissions that come in after the due date will be considered late.
> - Again, assignments will be run through similarity checking software to check for code similarity. Sharing or copying code in any way is considered plagiarism (Academic dishonesty) and may result in a mark of 0 on the assignment and/or be reported to the Dean's Office. Plagiarism is a serious offense. Work is to be done individually.

1. You must submit one file only to the Assignment submission page on Gradescope. The required file is as follows:

   `server.py`

2. Make sure that you get the confirmation email after submission.

3. The submission will be using Gradescope Assignment-01 page .