INF-2300: Computer Communication
Assignment 1 - HTTP server and RESTful API

Sera Madeleine Elstad
UiT id: sel063@uit.no
GitHub user: SeraMadeleine

September 12, 2023

Introduction

The main goal of this project is to create a robust HTTP server equipped with a RESTful API. This server is pivotal in efficiently managing HTTP requests, facilitating the creation, retrieval, updating, and deletion of resources. Operating within the context of the HyperText Transfer Protocol (HTTP), the server will encompass crucial functionalities, including handling diverse HTTP methods, ensuring accurate error responses, and integrating data persistence. The project's core emphasis lies in developing a versatile server that can adeptly process a wide range of HTTP requests while effectively managing resources. This server seamlessly operates within the HTTP framework, employing numerous HTTP methods for resource manipulation.

Technical background

HyperText Transfer Protocol(HTTP)

HTTP is the foundation of web communication, a client-server protocol that enables data exchange via the Internet. Clients, typically web browsers, request data from servers using methods like GET or POST. HTTP protocols like GET, POST, PUT, and Delete enable various interactions, enhancing internet accessibility and involvement [1].

An HTTP server is a software application that handles incoming HTTP requests from clients, acting as a mediator between the client and the requested resource. It retrieves static content like HTML files and sends them back as a response, while dynamic content like RESTful API responses is processed by the server [2].

HTTP headers are crucial for communication between clients and servers, providing additional information about the request or response. Content-Type and Content-Length are common headers, indicating the type of data sent and the size of the content in bytes, respectively, helping the recipient understand how to process and display the received information.

HTTP response codes are crucial for online interactions, debugging, troubleshooting, and user experience. They range from 200 OK to 404 Not Found or 500 Internal Server Error [3]. Proper handling of these codes impacts automation, security, and SEO in web development.

RESTful API

A REST API manages communication between applications and devices, adhering to REST principles. It emphasizes a standardized interface for reliable resource access using HTTP, promoting statelessness and reducing server-side sessions. It uses caching for efficiency and scalability, a tiered system for intermediates, and code-on-demand for variable execution. REST APIs use JSON format and common HTTP methods for CRUD operations, with request headers and parameters carrying data and HTTP status codes improving functionality [2].

JSON (JavaScript Object Notation)

JSON is a popular data format for APIs due to its simplicity and compatibility with programming languages. It's lightweight and easy to interchange between systems. JSON objects consist of key-value pairs, allowing for easy organization and representation of complex data in a hierarchical manner [4].

Design and implementation

The code for this project includes an HTTP server that handles HTTP methods (GET, POST, PUT, and DELETE) to deliver both static and dynamic content, as well as handling error and data storing capabilities. *Error_handling.py*, *HTTP_handler.py*, and *server.py* are the three main classes that make up the code. The HTTP server, which is located on a specific port, manages static and dynamic content while forwarding incoming requests to the right handlers.

Error handling

The error handling system is designed to provide customized error messages to users based on both the error code and the corresponding message. This design choice ensures that users receive accurate and meaningful information when an errors occur. Additionally, the system is designed to handle headers and error message contents for a variety of HTTP error codes. This design encourages code reuse and simplifies the process of generating consistent and informative error responses.

To implement the error handling system, two essential functions have been created. The first function is responsible for constructing error message bodies. It takes two parameters: an HTTP error code and an associated error message. This function generates HTML error bodies tailored to display the error message in a user-friendly format.

The second function handles the complete HTTP error responses. It evaluates the provided error code and constructs an HTTP response that includes headers and the error message body. This function effectively manages a range of common HTTP error codes, such as 204 (No Content), 400 (Bad Request), 403 (Forbidden), 404 (Not Found), 405 (Method Not Allowed), and 500 (Internal Server Error).

HTTP handling

The HTTPHandler is a class that manages HTTP-related tasks within a server application, primarily focusing on creating HTTP response headers and identifying content types based on file extensions.

One fundamental design aspect involves establishing a mapping between file extensions and their corresponding content types. With the help of the get content dictionary and the find content type function, files are precisely categorized so that the server can identify the correct content type for each file based on its extension. The function extracts the file type from the given filename and looks up the content type in the dictionary. This implementation enables the server to determine content types dynamically, enhancing versatility in handling various file types.

The design ensures that all response headers are built in the same way and makes it simple to generate an HTTP response header. The function takes three parameters: the HTTP response status, the content type of the response, and the length of the response content, and puts together the whole HTTP response header as bytes.

A dictionary containing all the content types is used to implement the mapping, which links each file extension (such as ".html") to the appropriate content type (such as "text/html"). The dictionary, which is declared as a class attribute, is a key part of the functionality of the class's content type determination. It has key-value pairs where content types are the corresponding keys and file extensions are the corresponding values. This design decision improves the adaptability and accessibility of content type mapping inside the class.

The TCPHandler and server

The MyTCPHandler class controls incoming HTTP requests and handles GET, POST, DELETE, and PUT requests as well as static content like HTML, CSS, images, and text files. It saves messages in a storage file and obtains the Content-Length from HTTP headers. The code sets up messages as dictionaries with "ID" and "Text" values to maintain data accessibility and permanence even after server restarts.

The handle method is in charge of responding to HTTP requests in an organized way. It reads the request line, parses it for correct formatting, and extracts the HTTP method and URI. To maintain consistency, the method enforces the HTTP method to be in uppercase and the URI to be in lowercase. If a request is malformed, it sends a 400 Bad Request response, and if the URI ends with ".py" or ".md," it issues a 403 Forbidden response. The requested method is based on the used HTTP method (GET, POST, PUT, DELETE). If the HTTP method is unsupported, it dispatches a 405 Method Not Allowed response.

GET requests are handled using a technique that distinguishes between URIs like "/", "/index.html," "/favicon.ico," and "/test.txt." Requests for the root URI ("/") are sent to "/index.html." It calls another method that retrieves and sends JSON data when it encounters URIs that begin with "/message." The inbound URI returns a 404 Not Found response if none of these criteria match, indicating that the requested resource could not be found.

The POST function differentiates between various URIs in order to process POST requests efficiently. It activates the post-request method when it comes across URIs like "/test.txt," which enables data appending to the specified file. It uses the post-json function to produce new messages and transmit updated JSON data for URIs that start with "/message.". When a new message is created, it is assigned a unique ID specific to that message. This ID generation process involves searching through the existing IDs within the message file and assigning the new message the lowest available ID. To maintain strict access control and stop unwanted actions, it swiftly responds to other POST requests with a 403 Forbidden status.

PUT requests for URIs that begin with "/message" are primarily designed to let clients effectively modify existing messages by using their ID. It starts by extracting the message ID and the modified content from the JSON request body. The method instantly responds with a 404 Not Found response to indicate the absence of the requested message if the message ID cannot be found in the currently delivered messages. If the update process is successful, it provides a 200 OK response to recognize the completion and makes sure that the updated messages are regularly preserved.

DELETE removes a message from the targeted URI (message) if its unique ID is present. By processing the JSON request body, the message ID is obtained for deletion. The request is promptly answered with a 404 Not Found response to let the user know that the requested message is not currently accessible if the message ID is not included in the collection of messages. A 200 OK response is returned, however, if the deletion process is completed and the requested message has been successfully erased.

The process of saving messages to a file serves a crucial role in maintaining server continuity, as it stores updated message data in a dedicated storage file, such as "message.json." When it comes to retrieving the appropriate filename, the method eliminates the leading slash from the URI. Additionally, the "find length" method plays a vital role in accurately processing both POST and PUT requests by adeptly extracting the Content-Length from the HTTP request headers, ensuring seamless handling of data within these requests.

Testing

The test client passes 9 out of 12 tests, demonstrating a rather solid performance. Furthermore, the code has undergone extensive testing against the web application, and the results align with the expected outcomes. In particular, GET requests for various endpoints, including the website root, "text," and "message," all display the expected content. Additionally, POST requests are sent to the "text" endpoint function correctly.

The code efficiently addresses security issues by giving a 403 Forbidden response when trying to access prohibited file formats like ".py" or ".md." In addition, it gently manages situations in which non-existent endpoints are requested, reliably returning 404 Not Found replies.

The code works as intended when testing JSON capabilities, returning the expected response codes and the text message with its matching ID, as instructed in the assignment. Overall, the code performs well in some scenarios, but the problem in test case 10 underlines that the code is far from perfect.

Discussion

It is important to be aware that test case 10 in the "test_client.py" file presents a specific issue while testing the code. When test cases 11 and 12 are attempted to be executed in reverse order, an unexpected behavior appears where it seems to enter an infinite loop. The actual cause of this phenomenon is still unknown. It is also important to underline that while the code generally functions well, there are a number of edge cases that are not considered and may cause unexpected behavior or program failures.

Conclusion

Although the code performs satisfactorily in many ways, there is obviously opportunity for development and refinement, especially in resolving edge situations and assuring robustness.

References

[1] An overview of HTTP - HTTP | MDN, September 2023. [Online; accessed 8. Sep. 2023].

[2] What is a REST API? | IBM, September 2023. [Online; accessed 11. Sep. 2023].

[3] Rogério Vicente. HTTP Cats, July 2023. [Online; accessed 11. Sep. 2023].

[4] JSON, April 2023. [Online; accessed 11. Sep. 2023].