Assignment 1

INF-2300

Erling Heimstad Willassen

1. Introduction

HyperText Transfer Protocol (HTTP) is a world wide web application-layer protocol based on the communication model between a client and server for information transfer[1] .Clients can request different CRUD (create, read, update, delete) operations to change the state of server objects. In this assignment, the students are tasked to create a server to handle HTTP CRUD operations on different server objects from external HTTP requests.

2. Technical Background

HTTP is the protocol which the world wide web mainly uses to transfer information between client and servers[2]. The protocol dictates how request and responses should be structured to maintain a good communication line. HTTP consists of CURL commands that dictate which information is of interest from the client to the server. For instance, internet browsers use HTTP protocol with GET command to acquire the layout of the webpage of interest in form of a HTML file, where the server responds with the context of said HTML file[2]. HTTP can be used with a TCP connection where the server awaits connection establishment from clients in the form of a request. The client identifies the server by an address in the HTTP protocol that is either an IP-address or an ASCII address with a port number which is unique to the server[2]. If the request is valid (allowed by the server for clients to ask and receive), the server responds with the information requested with headers including a status code to confirm the success of both requests and respond such as the "200 OK"[2]. If the request is invalid in the form of a forbidden or non-understandable request, an error status code such as "404 Not Found" is sent instead[2]. This way, the server can communicate to clients about the request was valid or not. HTTP is also a stateless protocol in that it does not store any previous information on requests by clients[3].

REST is an architectural style for HTTP users to transfer information in an orderly fashion between web-APIs[4]. A web-API (Application Programming Interface) which follows the architect of REST is often called RESTful, indicating that it follows the structure of REST in HTTP communication. These RESTful web-APIs often use address encoded parameters such as file-names and using JSON or other structure to convey data through HTTP communications[4].

3. Design

In the construction of a server, the coding needs to handle requests from clients to correctly respond to said request. When the server is live, the incoming request needs to be understood by the server to decipher the server to respond. The different requests a client can make, such as CURL, needs to be understood by the server so the client can acquire or alter the state of correct data on the server. By designing the server in such way to always listening after request by clients and then deciphering the request on the server, the client can achieve the requested data operation. The server will therefore handle the request in the form of a string that will be divided into several parts in which the important key information regarding the request can be accessed. After dissecting the request and accessing key details, the server can then respond correctly to said request. Since the assignment asks for common CURL commands such as GET (request data), POST (adding data), PUT (changing data) and DELETE (deleting data), different "if" statements for each command will be implemented to divide the operations to its own coding to ensure correct response. From there, the server can check if the request is allowed or prohibited to alter the state of said files in the request and contain odd cases of possible server crashes when accessing server files or "hacking" possibilities.

4. Implementation

4.1 Acquiring and deciphering request

By using python, one can import "socketserver" as a python library to handle requests from clients. With the functionality that comes with socketserver library, it is possible to create a server that is running continuously and waiting on requests from a client (innlevering kilde). When a request is sent to the server, the handle() method of the socketserver library will be used to handle the incoming request. Here, the request needs to be deciphered and divided into segments to fully understand the request in its entirety. To get the request, the inbuilt method of rfile.read() will acquire the request in a string format in a encoded version, which can be decoded by python-inbuilt function of decode().

By acquiring the request as a string, the request can easily be segmented by the split() function that is integrated in the python language. By splitting each line, one can easily acquire the key details of the request into a list format. By doing this, the CRUD operation and data target of said operation can be acquired in a string format for further processing. To avoid copying a lot of code, an own function to respond to client requests are made that responds according to the status code from the handle() function, where the status code acts as a key to a dictionary containing the words accompanying the status codes.

4.2 Processing request

When the request is finally segmented, the CRUD operation in the request can be understood. By having if-statements for each CRUD operation that is asked for in the assignment (GET, POST, PUT and DELETE), the CRUD operation in the request can be correctly assessed. For each operation, one must make sure that the correct file is processed in the operation and no other files that are forbidden to be accessed. Even if the assignment were divided into two parts of HTTP and REST requests for the student to solve, were the coding for both requests in the same part where the operation is the same, such as GET and POST for both the .txt and .json file.

4.3 GET request

The file that can be requested from the GET operation is the index.html file, which constructs a website for a POST operation to the same server. For the GET operation in the code, other if-statements can be seen. Here, the code will look after the substring "/" or "/index.html" in the request, which indicate the client wants to access the index.html. Since this is a request that is allowed by the client, the .html file is accessed by the inbuilt read() function which is sent back to the client with the appropriate headers such as the status code '200 OK', data type of 'text/html' and length of data.

Another GET request that is allowed is the REST API of 'message.json'. Here the client can request the whole .json file of messages with id number and text field. In this if-statement, the substring of "/messages" or "/messages.json" are looked after in the request. If containing the substring, the server will open the .json file of messages in the server home folder with the help of the function json.dump() from the json import. Here the file will be fetched as a string and the correct headers such as type of "text/json" and length are replied to the client, giving the client all the messages on the file as in the terms of a .json structured file.

To avoid that the client is accessing files that are forbidden or off-limits, one if-statement looks after the word "server" or "README" in the request that are forbidden files to access for clients. Here, no data is replied to the client other than the status code "403" for "Forbidden".

All other GET requests that are not any of these if statements will be replied to as the "404", "Not Found".

4.4 POST request

Same as the GET request, the server tries to find the substring of "POST" in the segmented operation string of the request. Here, the same as with the GET request part of the server code, the are additional if-statements for different POST requests to different files. The first if-statement checks after the substring of "/test.txt" or "test.txt" in the request string. This file allows clients to change of state too. Since this is a POST request, additional text is required to post into the file of interest. Here, this text (hereby called as body) is accessed from the request string. Then, the .txt file is then opened, and the body is written into a new line in the file. As part of the assignment, the body is then sent back to the client as confirmation of a complete POST request with additional headers as further confirmation by the HTTP rules.

An additional POST request to the server that is allowed for clients to access is changing the state of a "messages.json" file. This file contains messages in form a .json file, containing an id number and a text field for each message. Looking at the substring "messages" in the request will allow the client to add messages to the .json file. To avoid problems downstream such as server crash, the request string is checked to be in the correct format to be posted to the .json file, such as including "{", "}", and " "text" ". Further on, if the request body is correct (exampe: { "text": "test"}) then using the json.dump() function will list all the messages in the .json file as a list. Here, by counting the length of the list (number of messages) we will acquire a new id number for the new POST request by counting the next length after the new message is put in the file. By converting the substring of the request as a python dictionary with the help of "ast" import (converts string into dictionary), updating the body with a id number is easily done with built-in python dictionary function of update(). Then, by appending the new message to the list of all messages, the list is converted into a .json structure with the dump() function and written over the old .json file. The body of the request substring of text and id number is then replied to the client with the additional headers. Several try/exception functions are present throughout in the code to avoid cases of bad requests such as no text or missing symbols that could crash the server.

Same as with the GET request part of the code, POST requests trying to access server.py, README.md or other bad requests will lead to no altering of states and responding with conditional negative status codes.

4.5 PUT request

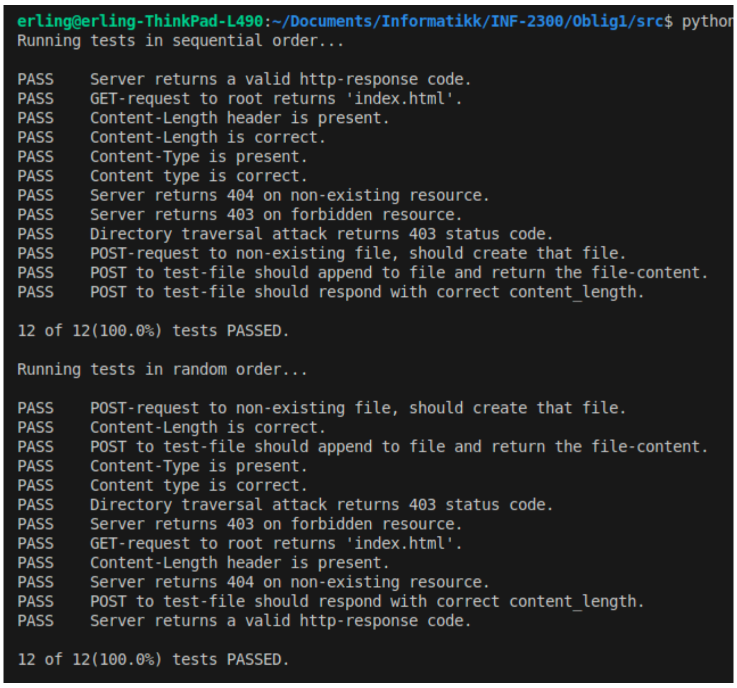
The only service the server can provide with PUT requests is accessing the messages.json file. Here, the request can alter an existing message if provided with the correct id number and an additional text to replace the older one with. As with the POST request to messages, the substring is checked for correct syntax for a .json file. Then after an error checking for the .json file and creating a dictionary of the request substring, the message with the corresponding id number to the substring is found and the old text is changed to the new from the request. After that, the message.json file is rewritten containing the new altered text and the new message is replied as the body by the server with additional headers confirming the action. All other PUT requests other than to the message file are replied with a negative error status code.

4.6 DELETE request

Since PUT and DELETE requests are different operations, they operate almost identical in code, except for processing the text string as in a PUT request the DELETE request will have no text string to process. The id number in the DELETE request will be the only information in the request, and by accessing the .json file in the same way as in the PUT request the old text will be replaced by a blank space and thereby deleting the old message. As there are no other files that can be accessed by a DELETE request, all other requests are responded by a negative status code.

5. Results

By using the test_client.py file that came with the pre-code, one can check if the CRUD operations are working somewhat as intended. The test file will test the coding of the server file, by mimicking various client requests and situations such as missing server files or bad client requests in general. As seen in the picture below, all tests from the test file were passed accordingly and correct according to HTTP standards.



```
erling@erling-ThinkPad-L490:~/Documents/Informatikk/INF-2300/Oblig1/src$ python
Running tests in sequential order...

PASS    Server returns a valid http-response code.
PASS    GET-request to root returns 'index.html'.
PASS    Content-Length header is present.
PASS    Content-Length is correct.
PASS    Content-Type is present.
PASS    Content type is correct.
PASS    Server returns 404 on non-existing resource.
PASS    Server returns 403 on forbidden resource.
PASS    Directory traversal attack returns 403 status code.
PASS    POST-request to non-existing file, should create that file.
PASS    POST to test-file should append to file and return the file-content.
PASS    POST to test-file should respond with correct content_length.

12 of 12(100.0%) tests PASSED.

Running tests in random order...

PASS    POST-request to non-existing file, should create that file.
PASS    Content-Length is correct.
PASS    POST to test-file should append to file and return the file-content.
PASS    Content-Type is present.
PASS    Content type is correct.
PASS    Directory traversal attack returns 403 status code.
PASS    Server returns 403 on forbidden resource.
PASS    GET-request to root returns 'index.html'.
PASS    Content-Length header is present.
PASS    Server returns 404 on non-existing resource.
PASS    POST to test-file should respond with correct content_length.
PASS    Server returns a valid http-response code.

12 of 12(100.0%) tests PASSED.
```

Figure 1: Test result from the test_client.py. All test were passed against the server code.

As for the REST API operations, no test files were given so one had to test themselves if the correct responses were acquired. For the assignment, some scenarios were constructed to test the server is behaving accordingly without crashing or being non-responsive. First scenario, the messages.json file will be present on the server to GET, POST, PUT and DELETE. Then, the messages.json will be removed to see if it can be constructed automatically and then proceed with the same operation. In the last scenario, different bad requests will be tried to the server to see if the server can handle the incoming messages with appropriate responses.

Table 1: CRUD operations against two scenarios with and without .json file, and one with bad handling.

Scenario	**GET**	**POST**	**PUT**	**DELETE**
With .json file	200 OK	201 OK	200 OK	200 OK
Without .json file	200 OK	201 OK	200 OK	200 OK
Bad request/handling	404 Bad Request	400 Bad Request/ 404 Not found	400 Bad Request/ 404 Not found	400 Bad Request/ 404 Not found

As seen with the table above, the server handles all CRUD requests with and without the .json file as intended. When .json file is present, the handling of the file works as intended as mentioned in the implementation chapter. If not present, the code will detect this and recreate the file as intended. The file would be empty, but all the CRUD operations can still be performed. The test was performed both from both directions of operations, to see if deleting a non-existing entry would crash the server where the correct bad handling response was sent back to client. For the bad request or handling to the server would contain .json structured body which are not complete or non-existing. In all cases, the server could handle these requests in a reasonable fashion where a bad request or handling response would be sent back to the client. All in all, the server could handle all good and bad requests in an accordingly fashion.

6. Discussion

In the end, the server handled all the requests in a good fashion. However, it took a lot of trial and error to find out the correct way to open files and handle incoming requests. When finally succeeding with said problems, the remainder of the time usage would be converting request information to strings, alter the server state with said information strings, and converting strings back to encoded bits again for response.

One major problem with opening files was that of files not existing or handled incorrectly. If one file did not exist, the opening function would crash the whole server. Depending on where one would initiate the server, the server's "home address" would also change and by consequence change where to find the .json file. However, by importing the "os" import to the code, the functions accompanying the os import could appropriately see if the .json file exists or not without crashing the server. This way, one could open a .json file without being afraid of a crash or create a new one if one is not existing. Another helpful function was the inbuilt methods of try/except in python. These methods would "try" to do the code inside its loop, and if an error exception would occur, the method would send the problems to the exception loop. By using try/except when altering or converting the state of files or datatypes, one can avoid server crash if something unexpected happens. These methods made it much easier to handle error codes that could potentially crash servers unexpectedly and handle them correctly.

Another annoying feature of the code is the massive copy and pasting of responses to each CRUD operation and its odd cases. Each status code and text would be present many times throughout each CRUD operation and make the code hard to read. Some coding was improving by having a dictionary for status codes and its description to fetch; however, this was a negligible improvement of code. This could be much cleaner done and not an easy method was seen how to prevent this. Unfortunately, a bit too late in the coding, a way was perhaps thought out by having try/except functions for each crud operation or for the whole code where the exception string would act as a key to the correct status code and reply in a python dictionary. This way, only one would be needed and therefore save space and make the code more readable.

Another thing is when being a novice in data computing, it can be difficult to know what other back-end problems that can occur to a server and how to prevent it. Without knowing what problems could occur, it is difficult to be hindsight to prevent unknown problems. Another problem is with hacking and what tricks hackers can do to hack a server. Without knowing explicably what hackers do, it is also hard to build good defenses around the server.

7. Conclusion

The student was tasked to create a server to handle CRUD operations on internal file types such as .html, .txt and .json. The server that was constructed handled these operations well and could handle odd cases of bad requests and handling without crashing the server unexpectedly. The student managed to complete the assignment by creating a server that could handle CRUD operations on said internal files in a well manner.

8. Sources

[1] C. Biordin, ‘Chapter 2 Application Layer’, Universitet i Tromsø, Norges Arktiske Universitet.

[2] ‘HTTP’, *Wikipedia*. Jul. 23, 2025. Accessed: Sep. 02, 2025. [Online]. Available: https://no.wikipedia.org/w/index.php?title=HTTP&oldid=25264018

[3] K. Ross and J. Kurose, *Computer Networking: A Top-Down Approach*, 8th ed. Pearson Education Limited, 2021.

[4] ‘Representational state transfer’, *Wikipedia*. Mar. 04, 2024. Accessed: Sep. 02, 2025. [Online]. Available: https://no.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=24311255