HTTP server and RESTful API

INF-2300 - Assignment 1

Deadline: Thursday 14. September 2023

Goals

The goal of this assignment is to familiarise you with the HyperText Transfer Protocol and how it can be used to create a web API. You will be exposed to some of the most important aspects of one of the most widely used protocols in networking which is the top-level backbone of almost all internet communication. You will learn how requests are handled by servers by your own request handler and exposing resources over a network.

Server

You will implement an HTTP server which will expose two endpoints. A server endpoint is simply an address to which we can issue requests and get a response of some kind. A response can contain the data of a file on the server, or it can simply by a confirmation of some task being completed. Responses should contain the `Content-Length` and `Content-Type` headers as a minimum.

HTTP server

This is the root endpoint and should support the following requests:

HTTP methods	URI	Action
GET	/	Return index.html in body of response
GET	/index.html	Return index.html in body of response
POST	/test.txt	Write body to test.txt file

Your server should respond with the correct HTTP response code.

- A **GET** request to a resource that does not exist should return a 404 - Not Found with an optional html body.
- A **GET** request to a forbidden resource such as `server.py` should return a 403 - Forbidden with an optional html body.
- Any successful **GET** request should return a 200 - Ok response code as well as the requested resource.
- A **POST** request to `/test.txt` should create the resource if it does not exist. The content of the request body should be appended to the file and its complete contents should be returned in the response body.
- A **POST** request to any other file should return a 403 - Forbidden with an optional html body.

Any request or response with a non-empty body, MUST contain the *'Content-Length'* header. This field is simply the exact size of the body in bytes.

RESTful API

The second endpoint is `/messages`. This will expose a simple message service where we can create, update and delete (CRUD) JSON-formatted messages. A message object should contain two fields, *text* and *id*. This is an example message object:

```
{
  "id": 1,
  "text": "Example text"
}
```

This end-point should support the following requests:

- A **GET** request to `/messages` should return a JSON-formatted list of all messages currently stored. Example:

```
[{
    "id": 1,
    "text": "Example text"
},
{
    "id": 2,
    "text": "More example text"
}]
```

- A **POST** request to `/messages` should accept a JSON-formatted entity body. It should persist this message so that it can be retrieved later. The response should be 201 - Created and a body containing the newly created message object. Example: Request:

```
{
  "text": "Example text"
}
```

Response:

```
{
    "id": 3,
    "text": "Example text"
}
```

- A **PUT** request to `/messages` should accept a JSON-formatted entity body with an id and text field. The server should find the message with the corresponding id and replace the text-field with the one from the request.
- A **DELETE** request to `/messages` should accept a JSON-formatted entity body, find the message with the corresponding id and remove the message. Response should be 200 - Ok.

Persistence

An important feature of the server is that the `/messages` end point is a persistent service. This means that any messages posted should still be present if you restart the server. You need to implement some way of persisting the state to disk, as well as reading existing state from disk. It's important to understand that the end-point `/messages` is a resource, not necessarily a physical file.

Precode

- `server.py` is an empty `socketserver` handler in python. Simply run this file to start the server. The handler is the method the server will use to process incoming requests. See the precode comments and python documentation for more information. The server is running on:

```
localhost:8080
```

- `test_client.py` is a simple test client to run against your HTTP server solution. It assumes your server is in `server.py`. It will run a series of requests, and check the responses. If you are unsure where to begin, the test client is written to function as a guide to help you get started. The test client is not meant to be a complete requirements test, nor does it present the only way to solve the assignment.

Tips

- Write ONE server! Don't write one server to solve the static HTTP part and one to solve the RESTful part. You will end up copy-pasting a LOT of code. A good solution will write generic code usable in both parts of the assignment.

- When parsing requests, don't assume anything about the configuration of the headers. There might be only one, or there might be 20. Test your code with different browsers and different number of headers, body lengths etc.
- Send manual requests to existing servers, and check the responses.

Special cases?

The specifications presented here are incomplete. This is deliberate. There is no common agreed upon set of specifications for a RESTful Web API. This is because they are very use-case dependent. For the edge cases, make choices that seem logical. Be consistent. Explain your choices in the report.
Here are some examples of edge cases(there are many more!) to consider:

- Non-supported request methods like HEAD, DELETE, or PIZZA for that matter, shouldn't crash the server.
- POST request to `server.py`?
- Non-supported request methods like HEAD, DELETE, or PIZZA shouldn't crash the server.
- DELETE request to non-existent message?
- GET request to empty resource?
- POST request with *id* field? POST request with NO *text* field?
- Empty POST, PUT and DELETE requests?
- PUT request with no *id*? 201 or 404?
- PUT request with existing *id* but no *text* field?

Requirements

Your solution should support the requests outlined above. Whenever you send a non-empty response, it should have a 'Content-Type' and 'Content-Length' headers. To summarise:

- Static HTTP-server supporting basic GET and POST requests.
- RESTful Web API supporting GET, POST, PUT and DELETE to `/messages` using JSON-formatted body.
- The `/messages` resource should persist between runs.
- You use aditional libraries such as json and urlllib, but none that trivialises the assignment such as httpserver or similar. If you are unsure, ask your TA.

Extra credits

- Consider security holes in your solution. Path traversal attacks? Whitelist rather than blacklist.
- Implement redirecting from / to `/index.html`
- Implement *Server* and *Date* headers.
- Implement another level in the `/messages` resource such that: GET, PUT and DELETE requests to `/messages/<id>` is resolved to the id in the url and not the JSON-object in the body. This is how most RESTful API's actually do it.

Resources

- For all available HTTP-response codes:
  https://http.cat/
- For manualy sending requests from the command line, use *httpie*. Multi-platform support. Easy to use. This will save you a lot of time!
  https://httpie.org/
- A wealth of information on RESTful API's!
  https://restapitutorial.com/
- If you prefer a GUI-based interface for testing your server. Insomnia is a good multi-platform tool.
  https://insomnia.rest/
- Hypertext Transfer Protocol – HTTP/1.1:
  https://tools.ietf.org/html/rfc2616
- HTTP - Message Syntax and Routing:
  https://tools.ietf.org/html/rfc7230
- HTTP - Semantics and Content:
  https://tools.ietf.org/html/rfc7231
- HTTP - Conditional Requests:
  https://tools.ietf.org/html/rfc7232
- HTTP - Range Requests:
  https://tools.ietf.org/html/rfc7233
- HTTP - Caching:
  https://tools.ietf.org/html/rfc7234
- HTTP - Authentication:
  https://tools.ietf.org/html/rfc7235

Hand-In

Hand in your solution and the report by uploading it in a compressed file to Canvas. Your report should be in pdf format in the 'doc' folder. Don't forget to update your README!

Good luck!