INF-2301: Assignment 1

HTTP Web Servers and RESTful applications

Submission Deadline: Monday, September 17th 2018 23:59

1 Introduction

In this assignment, you will explore the Hypertext Transfer Protocol (HTTP) and its usage in RESTful WebAPI applications. The assignment is split up into two parts:

Part-A: The Static Web Server and,
Part-B: The RESTful WebAPI server.

Completion of **BOTH** parts is required to pass this assignment.

The goal is to implement a server compliant with the HTTP protocol specification, as declared in the following documents:

- RFC 7230, HTTP/1.1: Message Syntax and Routing
- RFC 7231, HTTP/1.1: Semantics and Content
- RFC 7232, HTTP/1.1: Conditional Requests
- RFC 7233, HTTP/1.1: Range Requests
- RFC 7234, HTTP/1.1: Caching
- RFC 7235, HTTP/1.1: Authentication

Although t*his assignment only require knowledge of RFC 7230 and RFC 7231*, we do recommend that you at least briefly look over all six documents.

You should try to implement an application that complies with the HTTP/1.1 standard, but hand-ins following HTTP/1.0 will also be accepted.

2 Part-A: The Static Web Server

In the first part of this assignment, you are to design and implement a simple static HTTP Web Server application. Many Web Servers are HTTP server that respond to an HTTP request message by returning the appropriate file that is located on the server's physical file system. In most cases the file returned is written in Hypertext Markup Language (HTML), a language understood by most web browsers to specify graphical and textual content for humans.

2.1 Implementation Details

Your implementation is required to exhibit the following characteristics:

1. An HTTP GET-request message to the root path of the server should be answered with either of the following:
   - An HTTP response message with an entity body containing the contents of the file named `index.html`

    - A redirecting HTTP response message that redirects the request to the `/index.html` resource path.
2. An HTTP GET-request message to any other resource path that can be resolved to a physical file path is to be answered by an HTTP response message that contains the contents of the physical file in the entity body.
3. An HTTP POST-request message should prompt the server to resolve the requested path to a physical file. Then, the server should process the request entity body following the conventions of the `application/x-www-form-urlencoded` content type. The server is to modify (or create) the requested physical file in such a way that it contains the text passed on through the text field of the transmitted web-form. Finally, the server should respond with a response message that contains the contents of the file after the modification in the response entity body.
4. Any GET-request message where the path that cannot be mapped to an existing physical file must be answered with an HTTP response message that carries the status code `404`.
5. Any request that was processed successfully and where the request operation succeeded must be answered with an HTTP response message that carries an appropriate status code that indicates a success (i.e., a status code out of the 2xx category).
6. Although the HTTP RFC defines most HTTP headers as optional, you are required to address the problem of specifying the content length of the entity body. Do at least one of the following:
    - Implement your server in such a way that every HTTP response message that has a non-empty entity body carries an appropriate `Content-Length` header. *(Easy)*
    - Implement chunked transfer encoding *(Hard)*

The requirements listed above are the minimum requirements for the first part of this assignment. Deviations from the requirements above are allowed if properly documented in the hand-in and a legitimate argumentation for the deviation is included.

2.2 Web Root Content

The web server root path is commonly used to refer to the directory that a static web server will serve when the / resource path is requested. The pre-code for this assignment includes an `index.html` file. When opened in a browser, this file will provide the user with an interface of generating POST-request messages to a `test.txt` file. If you examine the HTML text in index.html, note the `form` tag and its `method` attribute. This setup is the primary mechanism with which HTML pages cause browser to issue POST-request messages to web server. A Browser will take the `value` of all `input` tags that are nested within a `form` tag and transmit that information in the entity body of the POST request message. Browsers will normally use the `application/x-www-form-urlencoded` content type to encode the information unambiguously. The provided `index.html` page also includes a section that will display how the information is encoded, which might be of some help when developing this part of the assignment.

3 Part-B: The RESTful WebAPI server

Many applications nowadays consume web services. In most cases, these services are HTTP services that expose a WebAPI to an application. An application can therefore consume the service by issuing simple HTTP request messages to the WebAPI server to access a resource. Note that in the case of

WebAPI servers, the response entity bodies are encoded in data formats such as XML or JSON, which are optimized to transfer data objects, but are not easily readable for humans.

One common architectural principle of how to design and structure a WebAPI is the RESTful application principle, which suggest using appropriate HTTP verbs (or methods)[1] to perform CRUD[2] operations on HTTP resources.

In this part of the assignment, you will implement a RESTful WebAPI that serves a messages repository over HTTP. The API will allow an application to read, create, modify and delete messages using HTTP methods.

3.1 Implementation Details

As with the first part of the assignment, in this part you are also to implement an HTTP server that complies with the HTTP/1.0 or higher standards. For the implementation, you may choose to use JSON or XML as you data transfer format.

Your web service must expose an end-point named `/messages` implementing the following:

1. A GET request to `/messages` should return all stored messages. In JSON, this should be done by returning a single JSON array with the message objects as elements. In XML, this should be done by returning a root element named `messages` that contains all message objects as child elements.
2. A POST request to `/messages` should accept a JSON or XML formatted entity body that contains a partial message object. The server should create a new message object in the repository, assign a unique `id` to that message, and set the `text` property to same text as received from the request entity body. The server must respond with the correct status code and the newly created object.
3. A PUT request to `/messages` should accept a JSON or XML formatted entity body that contains a complete message object. The server should find the message object in the repository whose `id` property matches the `id` of the received message object, if no message with the given `id` exists, one should be created. It should then update all other properties to the same value as the corresponding property of the received object. Finally, the server should respond with a response message that contains the updated message object. Note that this request method differs from POST method as it here is the client's responsibility to specify a value for the `id`, whereas the POST method lets the server decide one.
4. A DELETE request to `/messages` should accept a JSON or XML formatted entity body that contains a partial message object. The server should find the message object in the repository whose `id` property matches the `id` of the received message object and remove it

[1] In HTTP, the use of the term verb is most often interchangeable with the term method. Examples are GET, HEAD, POST, DELETE, PUT and describe what operation a web server should perform on a resource.

[2] CRUD: Create, Update, Delete. The three usual modifying operations that can be applied to an object. Together with the Select operation, these form the common data operations.

from the repository. The server should then reply with response message that has an empty, zero-length entity body.

3.2 Data Format Specification

The messages web API declares one type of data object, the message object. It is a data object that has two properties:

- A property named `id` that has a signed 32-bit integer value as its data type.
- A property named `text` that has string value as its data type.

3.2.1 XML Message object representation

In XML, the element name of a message object should be `message`, the `id` property should be used as an attribute and the `text` should be the element text as shown in the example below.

```
<message id="1">This is a sample message object.</message>
```

In a POST request, where the id is not yet known, the following would also be a valid partial message object: (In cases where the id attribute is specified in a POST, the server should simply ignore the value.)

```
<message>This is a sample message object.</message>
```

In DELETE messages, where only the id needs to be specified, the following would also be a valid partial message object:

```
<message id="1" />
```

3.2.2 JSON Message object representation

In JSON, a message object should be represented as a JSON object, with the respective properties:

```
{
  id: 1,
  text: id="This is a sample message object."
}
```

4 Requirements and Expectations

This assignment requires you to hand in both code and a report. If you fail to implement every aspect of this assignment in code, you may compensate in the report. The purpose of the report is to convince us that you have completed all required tasks and acquired the needed skills. We will run and test your code, but the report is our primary source for checking your work.

4.1 Expected Code Abstraction Level

For this assignment, you may use any programming language of your choice. However, you should keep in mind that we can only offer assistance on common programming languages like Java, C, or Python. If you are uncertain whether or not a programming language is suitable, please ask us.

Various HTTP server frameworks are readily available. Many programming languages even comes equipped with complete HTTP servers, right out of the box. These make building complex web sites and APIs a simple task, requiring little knowledge of the underlying protocols. Solving this assignment using such tools defeats its purpose**. The use of HTTP servers and frameworks are therefore not allowed.** You are, however, allowed to use libraries that simplify I/O handling by, for instance, providing `ReadLine` and `WriteLine` type of functions. For instance, in Python the `SocketServer.TCPServer` class in combination with `SocketServer.StreamRequest-Handler` is an acceptable (and recommended) abstraction level for solving this assignment. If you are unsure what is acceptable, please ask.

4.2 THE REPORT

The report should document your work, and what you learned. You must write in English. Although we do not emphasis grammar, structure, and style, it is important that your report is comprehensible and convey the correct information. If we cannot understand what you are writing, we cannot pass your report; and a well written exposition on lambda calculus or the inner workings of the EXT3 filesystem is of little value in an assignment on HTTP servers.

You should write with an expert reader in mind. Keep it short, to the point, and technical. Use the language, concepts, and knowledge you have learned from this and previous CS classes to state interesting observations and insights. Do not be afraid to talk about complexity, data structures, access patterns, memory layout, and other similar classical CS topics that you already master. Think about why you did what you did, and what the alternatives are.

Your report should at least contain the following sections: introduction, design/implementation, and conclusion. Other sections can be added as needed. You could include a description of how HTTP request messages are processed and how responses are formulated. Perhaps explain in what order you process the information in an HTTP request message and what effect the different elements of the request have. You could also explain what happens if a client transmits a message with a header-line stating: `Connection: keep-alive.` Explain how you extract information from the request message and what you do with the components of the request-line, or how do you process the information contained in the entity body.

5 HAND-IN INSTRUCTIONS

Submit your hand-in via Canvas before the specified deadline. You hand-in should consist of a separate pdf-file, as well as a SINGLE archive file such as a Zip-compressed Archive of your choice(.7z, .tar, .tar.gz, .tar.bz, .tgz, .tbz, etc.) or a WinRAR-archive (.rar). The name of the hand-in file should be in the following format: *inf2301-2018.2-1-abc123.tar.gz*, where *abc123* is your student name.

Inside the archive folder, there should only be a single folder with the same name as the archive filename. This step is to ensure that multiple hand-ins can be extracted safely within the same folder by the grading TA.

Underneath that root folder, there should be a subfolder called "src" containing the source code. The handin should also include a README file in the root folder where you give source-code-specific instruction on how to compile and execute your source code.

Your report is handed in separately in Canvas as a pdf-file.

Reasonable use of external non-standard libraries is okay, but only if their use is properly documented in the hand-in and you give instructions on how to obtain them.

6 Extra Credits

In additional to the requirements listed above, you may consider the following for extra tasks

6.1 For Part-A:

- Implement the `Last-Modified`, `Server` and `Date` HTTP response headers.
- Implement the `Content-Type` HTTP response header. The simplest way here, is to make a list of known file extensions and how they map to appropriate content types. (e.g.: `.html` maps to `text/html`, `.txt` maps to `text/plain` and `.json` to `application/json`, all else to `application/octet-stream`). The implementation should affect how your browser displays the entity body of the received response.
- Implement recognition of the `If-Unmodified-Since` HTTP request header and respond accordingly. You may further explore other `If-*` headers to perform conditional requests. Read RFC 7232 for more information on this topic.
- Implement recognition of the `Expect: 100-continue` HTTP request header, pre-allocate space for the incoming request entity body, respond with an appropriate intermediate HTTP response message, and continue to process the request normally.
- Implement additional HTTP methods to perform file system operations. E.g., support PUT and DELETE operations for files. Also, provide a directory listing for GET requests that have a request path that resolves to a directory rather than a file.
- Implement persistent connections by appropriately reacting to the `Connection` request header and respond with an appropriate `Connection` header.

6.2 For Part-B

- Implement the possibility to use the resource path `/messages/{id}` where `{id}` is the value of the `id` property of the message you want to operate on. A GET message with the path `/messages/4` should return only the message object with an `id` value of `4`. A DELETE request to `/messages/4` should delete the message object with an `id` value of `4`.
- Implement the POST method in such a way that it responds with a `201` response message and a `Content-Location` header that points to the appropriate `/messages/{id}` path.
- Implement that DELETE request are answered with `204` response messages