CS-4333/6333-Project-3

Overview

HTTP 1.1 [RFC 2616] defines the following methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. The goal of this project is to implement a minimal HTTP 1.1 server supporting and implementing only the GET and HEAD methods. This protocol typically runs on top of TCP on port 80.

The HTTP protocol is a request/response protocol:

1. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possibly body content over a connection with a server.
2. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possibly entity-body content.

Where a URI (Uniform Resource Identifier) is either a URL (Uniform Resource Location) or a URN (Uniform Resource Name). Throughout this document the following notation is used: SP is a blank space and CRLF is a carriage return followed by a line feed character. URIs have been known by many names: WWW addresses, Universal Document Identifiers, Universal Resource Identifiers , and finally the combination of Uniform Resource Locators (URL) and Names (URN). As far as HTTP is concerned, Uniform Resource Identifiers are simply formatted strings which identify a resource via a name, location, or any other characteristic.

Client Request

The general form for an HTTP/1.1 request is:

```
Method SP Request-URI SP HTTP/1.1 CRLF
([general-header line | request-header line | entity-header line] CRLF)*
CRLF
[message body]
```

For this project you are only required to implement the GET and HEAD method. Note that a request may be followed by several other lines (some of them are mandatory as explained below) ended with a CRLF. The end of the header lines is indicated by the client by sending an additional CRLF. The [message body] part will not be used in this project. The message body part is where you would place data being sent to the server, e.g. if we were implementing the PUT method, the message body would contain the data to be sent to the server.

Any HTTP/1.1 compliant client must include a Host request-header field identifying the internet host (the server) they are connecting to and the port number:

```
Host: hostname[:port]
```

The port number must be included unless the server is using the default port 80. For the header lines, the notation `(...)*` means zero or more of the strings enclosed in parenthesis. If the machine where the server is running does not have a DNS name, you may use the IP address for hostname.

Your server must check for the Host request header field before serving the request. If there is no Host request-header, the server must return an error code as specified in the following section.

In summary, a minimal and valid GET request on a server named neo.mcs.utulsa.edu running on port 16405 for a file or resource named index.html will look like:

```
GET /index.html HTTP/1.1 CRLF
Host: neo.mcs.utulsa.edu:16405 CRLF
CRLF
```

The same request as sent by a Chrome browser using Mac OS X may look like:

```
GET /index.html HTTP/1.1 CRLF
Host: neo.mcs.utulsa.edu:16405 CRLF
Connection: keep-alive CRLF
Cache-Control: max-age=0 CRLF
Upgrade-Insecure-Requests: 1 CRLF
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_6)
AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/86.0.4240.80 Safari/537.36 CRLF
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp
,image/apng,*/*;q=0.8,a
pplication/signed-exchange;v=b3;q=0.9 CRLF
Accept-Encoding: gzip, deflate CRLF
Accept-Language: en-US,en;q=0.9 CRLF
CRLF
```

> Note in this case that the client is sending lots of information to the server. It basically notifies the server what kinds of files the browser accepts (Accept lines) and a User-Agent line that notifies the server that the client Chrome.

Remember that there could be more lines following the GET method request. Your server must look among those lines for a Host request-header field before it is served (and return an error code if the line is missing). Your server can safely ignore the lines that are not strictly required by the server in order to process a client's request.

Server Response

After receiving and interpreting a request message, a server responds with an HTTP response message. The general form of a response message consists of several lines: (i) a status line followed by a CRLF, (ii) zero or more header lines followed by a CRLF (also called entity headers), (iii) a CRLF indicating the end of

the header lines and (iv) a message body if necessary containing the requested data. The general form of the response message then is:

```
HTTP/1.1 SP StatusCode SP ReasonPhrase CRLF
([( general-header line | response-header line | entity-header line)] CRLF)
*
CRLF
[ message-body ]
```

> Note: the only difference between a server response to a GET and a server response to a HEAD is that when the server responds to a HEAD request, the message body is empty (only the header lines associated with the request are sent back to the client).

Status Line

```
HTTP/1.1 SP StatusCode SP ReasonPhrase CRLF
```

The StatusCode element is a 3-digit integer result code of the attempt to understand and satisfy the request. These codes are fully defined in section 10 of RFC 2616. The ReasonPhrase is intended to give a short textual description of the StatusCode. The StatusCode is intended for use by automata and the ReasonPhrase is intended for the human user. The client is not required to examine or display the ReasonPhrase.

The first digit of the StatusCode defines the class of response. The last two digits do not have any categorization role. There are 5 values for the first digit:

- 1xx: Informational - Request received, continuing process.
- 2xx: Success - The action was successfully received, understood, and accepted.
- 3xx: Redirection - Further action must be taken in order to complete the request.
- 4xx: Client Error - The request contains bad syntax or cannot be fulfilled.
- 5xx: Server Error - The server failed to fulfill an apparently valid request.

The following table summarizes the status codes and reason phrases that your server MUST implement:

StatusCode	**ReasonPhrase**
200	OK
400	Bad Request
404	Not Found
501	Not Implemented

A 200 OK response indicates the request has succeeded and the information returned by the server with the response is dependent on the method used in the request. If the message is in response to a GET method, then the message body contains the data associated with the request resource. If the message is in response to a HEAD method, then only entity header lines are sent without any message body.

A 400 Bad Request message indicates that a request could not be understood by the server due to malformed syntax (the client should not repeat the request without modification).

A 404 Not Found message indicates the server has not found anything matching the requested resource (this is one of the most common responses).

A 501 Not Implemented message indicates that the server does not support the functionality required to fulfill the request. Your server should respond with this message when the request method corresponds to one of the following: OPTIONS, POST, PUT, DELETE, TRACE and CONNECT (the methods you are NOT implementing in this project).

Entity Headers

There are several types of header lines. The intention of these lines is to provide information to the Client when responding to a request. Your server MUST implement (and send back to the client) at least the following three header lines when the request is valid. Otherwise, just the server entity header is fine.

```
Server: ServerName/ServerVersion
Content-Length: lengthOfResource
Content-Type: typeOfResource
```

The Server response-header field contains information about the software used by the origin server (the server you are implementing for this project) to handle the request. An example is:

```
Server: cs4333httpserver/1.0.2
```

The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET. An example for `lengthOfResource=3495` is:

```
Content-Length: 3495
```

The Content-Type entity-header field indicates the media type of the message-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET. Your server must be able to report to the client the following media types:

File Name	File Type	Content Type
*.html, *.htm	html	text/html
*.gif	gif	image/gif
*.jpg, *.jpeg	jpeg	image/jpeg
*.pdf	pdf	application/pdf

An example for typeOfResource when responding to a GET request method for a file named index.html is:

```
Content-Type: text/html
```

As illustrated above, the end of the entity header lines is indicated by sending an additional CRLF.

Message Body

The message-body (if any) of an HTTP message is used to carry the entity-body associated with the request or response (typically the file being requested). I suggest you use the FileInputStream Java class to read the contents of a requested file.

Assignment

Your implementation MUST follow the following guidelines:

1. Use the Java Socket API for this project.
2. You can use as many classes as you want, but the class containing the main starting point should be named HttpServer.java and it should accept a number as a command line option indicating the port number to be used by your server. For instance, a valid invocation to start the server on port 16405 is: `java HttpServer 16405`.
3. Your server should assume the existence of a directory named public_html in the directory where your class and source files are stored. Under no circumstances your server will serve any files not in the public_html directory (or any subdirectory below public_html). We do this for security reasons. If you are not careful, your server would be able to serve any requested files with a valid resource name. The public_html directory is the root directory for any files served by the server. See the Security Considerations of RFC 2616 (Section 15) for additional information. Note: the public_html directory only serves as the base directory for the server and should not be included when forming the URL.
4. Echo back to the screen the entire contents of any requests your server receives and any responses up to, but not including the message body (if there is one to transmit).
5. Your server should be multithreaded and must be able to handle multiple concurrent requests.
6. It is advised that you do not run your server on port 80. There are plenty of Internet worms and hackers scanning machines and looking for port 80.

Use **TUGrader.java** to ensure that all unit tests in the provided test file(s) pass.

Submission

The assignment is due by the end-of-day on the last day of classes. You must include a well typed report explaining your design and implementation. You MUST implement the required features as described in this document and you MAY include any other additional features as long as they are HTTP/1.1 compliant.

*To submit your work, please review the **Commits** section below.*

Grading

- [ ] (140pts) Can parse and validate HTTP requests

- [ ] (100pts) Can serve HTTP response
- [ ] (60pts) Written report

Compiling and Testing Code

Your IDE should provide tools to compile your code. If you're unfamiliar with that process, you can research it online or ask. Most developers compile their code from command line using a shell script, such as a **Makefile** or build script (**build.sh**). I've provided build scripts for you in both *Powershell* and *Bash*. Refer to the following directions on how to use these scripts based on the terminal that you're using. If you're on Windows, please use Windows Subsystem for Linux (WSL), Git Bash, or Powershell, not Command Prompt.

Windows Users (WSL, Git Bash), Mac and Linux Users

- To compile your code: `./build.sh`
- To compile and run your code: `./build.sh run` (forwards clargs to program)
- To compile and test your code: `./build.sh test` (forwards clargs to TUGrader)
- To format your code: `./build.sh fmt`
- To sync your code: `./build.sh sync`
- To submit your code: `./build.sh submit`
- To remove class files: `./build.sh clean`

These scripts use the following commands. Note that Windows users need to replace the colon with a semicolon in the Java classpath.

- To compile a Java file: `javac -d target -cp lib/*:src <filepath>.java`
- To execute a Java file: `java -cp lib/*:target <package-path>.<filename>`
- To format a Java file: `java -jar lib/google-java-format.jar --replace --skip-javadoc-formatting <filepath>.java`
- To remove class files: `rm -r target/*`

Code Style

All code should follow the Google Java style guidelines. If you find anything in the code that does not follow the style guidelines, feel free to fix it, but you are not required to do so. Only your handwritten code will be evaluated for its style. You do not need to follow the style guidelines to the letter but egregious deviations from the style guidelines will be penalized. A submission that passes all test cases but does not use an appropriate style will not receive an A for the assignment.

For those using an IDE, such as Eclipse or VS Code, the IDE should provide a formatting tool. I've included the XML specification of the Google Java Style Guidelines at `.vscode/java-google-style.xml`. You can configure your IDE to use the provided XML as its formatting rules to format your code to the Google Java Style Guidelines, which are the industry standard.

If you're working from command-line, google-java-format is an open-source formatting tool that you can use to format your files. You can use the following commands to format your code depending on your terminal.

- `./build.sh fmt`

Commits

Commits should be made incrementally. Many commits are always better than few, and commits can always be squashed together later if there are too many. You should try to make a commit every time you've made tangible progress in the development of your code.

Every commit should have a commit message. The standard format for commit messages is to start with a verb in present-tense followed by a concise message (typically less than 50 characters) that summarizes the change that the commit is introducing to the repo. For example, "Updates README", "Implements Array", "Passes testGet".

Popular IDEs, such as Eclipse and VS Code, provide integrated Git tools. If you're on Windows, you can install Git Bash or Windows Subsystem for Linux (WSL). If you're on Mac or Linux, you already have git installed.

If you've just installed git, it will need to be configured. The easy way to configure git is from a terminal. Use the following commands.

- `git config --global user.name "<github-username-goes-here>"`
- `git config --global user.email "<github-email-goes-here>"`
- `git config --global pull.rebase true` (optional)
- `git config --global fetch.prune true` (optional)
- `git config --global diff.colorMoved zebra` (optional)

To sync changes made from another device, use the following command.

- `git fetch origin main`
- `git pull origin main`

To push commits from command line, use the following commands.

- `git add -A`
- `git commit -m "<your message goes here>"`
- `git push origin main`

You can also sync all changes and submit with the following commands depending on your terminal.

- `./build.sh submit`