CSC 667 Web Server Project

Team 200: Olivier Chan Sion Moy, Nathan Rennacker

Github repository:
https://github.com/csc-667-fall-2022-sfsu-roberts/webserver-team-200

Index

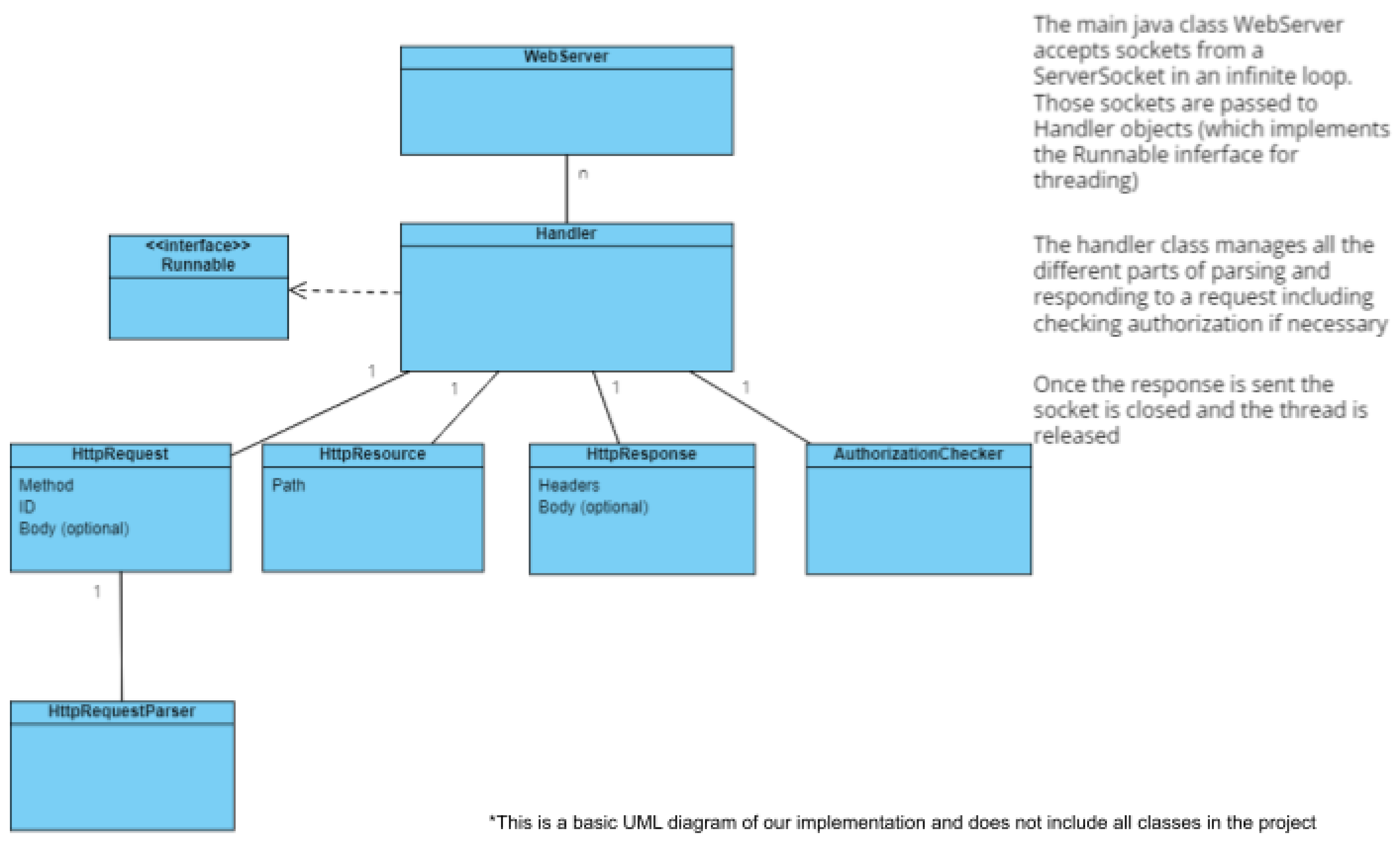
Suggested Grading Rubric

Category	Description	Points Possible	Points Received
Code Quality	Code is clean, well formatted (appropriate white space and indentation)	5	5
	Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	5	5
	Methods are small and serve a single purpose	3	3
	Code is well organized into a meaningful file structure	2	2
Documentation	A PDF is submitted that contains:	3	3
	Full names of team members	3	3
	A link to github repository	3	3
	A copy of this rubric with each item checked off that was completed (feel free to provide a suggested total you deserve based on completion)	1	1
	Brief description of architecture (pictures are handy here, but do not re-submit the pictures I provided)	5	5
	Problems you encountered during implementation, and how you solved them	5	5
	A discussion of what was difficult, and why	5	5
	A thorough description of your test plan (if you can't prove that it works, you shouldn't get 100%)	5	5
Functionality - Server	Starts up and listens on correct port	3	3
	Logs in the common log format to stdout and log file	2	2
	Multithreading	5	5
Functionality - Responses	200	2	2
	201	2	2
	204	2	2
	400	2	2
	401	2	2
	403	2	2
	404	2	2

Category	Description	Points Possible	Points Received
	500	2	2
	Required headers present (Server, Date)	1	1
	Response specific headers present as needed (ContentLength, Content-Type)	2	2
	Simple caching (HEAD with If-Modified-Since results in 304 with Last-Modified header, Last-Modified header sent)	1	1
	Response body correctly sent	3	3
Functionality - Mime Types	Appropriate mime type returned based on file extension (defaults to text/text if not found in mime.types)	2	2
Functionality - Config	Correct index file used (defaults to index.html)	2	2
	Correct document root used	2	2
	Script Aliases working (will be mutually exclusive)	3	3
	Correct port used (defaults to 8080)	2	2
	Correct log file used	2	2
CGI	Correctly executes and responds	4	4
	Receives correct environment variables	3	3
	Connects request body to standard input of cgi process	2	2

Architecture and Design

Web Server UML Diagram



The main java class WebServer accepts sockets from a ServerSocket in an infinite loop. Those sockets are passed to Handler objects (which implements the Runnable inferface for threading)

The handler class manages all the different parts of parsing and responding to a request including checking authorization if necessary

Once the response is sent the socket is closed and the thread is released

*This is a basic UML diagram of our implementation and does not include all classes in the project

Problems and Solutions

Problem 1

We encountered a weird bug with Postman early on into the project where any request sent by Postman would throw an error consistently without reading any data into the request object.

Solution 1

By shoring up our error catching and error response creation we managed to work out that Postman actually sends an empty socket sometimes before it sends the actual request (we posited that this is *maybe* used to establish an initial connection but still don't fully understand why this is necessary)

Problem 2

In our initial first steps into the project we did not realize that saving an HTTP body as a string would create a huge issue when trying to convert that string back into a file.

Solution 2

Switching to byte arrays for saving HTTP body data–and tinkering with the parsing as a whole–finally made it so we could save the file in the correct format without it being corrupted.

Problem 3

Using a while loop when reading from the socket inputStream was problematic since sometimes the stream would have data that wouldn't be read.

Solution 3

Switching to a do {} while{} loop ensured that the data would be read regardless.

Problem 4

For logging HTTP responses in common log format, you need the amount of bytes sent to the client and there were troubles initially when trying to find that data.

Solution 4

Extending OutputStream solved the issue as it allowed for a total byte count to be present which could be read from when writing to the log.

Problem 5

For simple authentication in the browser, the spec specified that users should receive a popup within the browser window that asks for username and password but attempts to find a way to manifest that prompt were fruitless at first.

Solution 5

In order for the browser authentication popup to show, you need to pass specific headers to the browser

Testing

Different parts of the project required different types of tests to ensure that they were working as intended. A couple different testing methods were used to verify functionality including **Postman**, the **browser and supplied website files**, **standard out printing,** and **Telnet.**

Postman

Postman requests were the most useful testing strategy out of everything that we did, (besides the small issue of it sending a blank request at the start–which actually turned out to be useful for error checking and 400 responses). This program allows for testing all the different request methods that we were supposed to implement–GET, HEAD, POST, PUT, DELETE–while including any authentication and extra headers.

Browser and Supplied Website Files

The browser was the most useful in keeping track of project progress, as you could plainly see what step you were on in the implementation of different server functionalities. It was also generally useful in basic testing of those functionalities. In particular, the threading test was handy as you could plainly see whether threading was implemented correctly. Two other important testing features from the website were the scripting test and the authentication test, though we tested both of those through Postman as well.

Stdout Printing

Printing to the console was useful throughout the entire project albeit less so than the other two methods listed because really what matters is that the requests receive proper responses and those need to be checked for accuracy from the client side. However, obviously for logging, ensuring that the correct format was present and *every* request was being logged, it was vital.

Telnet

Used to send incomplete requests to test 400 responses.

Test Plan

Functionalities

Functionality	Test Plan
Starts up and listens on correct port	Change the “Listen” directive and issue GET requests to the specified port. Remove the “Listen” directive and issue GET requests to the default port.
Logs in the common log format to stdout and log file	Issue requests. Verify correct log format in console (stdout) and configured logfile.
Multithreading	Use the Javascript “thread test” button in the example webpage. Verify that requests finish in a non-deterministic order, and that loading the “really big image” does not block other requests from completing.
Required headers present (Server, Date)	Issue requests. Verify required headers present with correct format.
Response specific headers present as needed (Content- Length, Content-Type)	Issue requests. Verify response specific headers present as needed.
Simple caching (HEAD with If-Modified-Since results in 304 with Last-Modified header, Last-Modified header sent)	Issue GET and HEAD requests with “If-Modified-Since” header. Verify “If-Modified-Since” dates both before and after last file modification return 200 and 304 respectively.
Response body correctly sent	Load example webpage. Verify HTML (text) and images (binary data) load properly.
Appropriate mime type returned based on file extension (defaults to text/text if not found in mime.types)	Issue GET requests to files. Verify correct mime type. Issue GET requests to files without a registered extension, verify mime type is “text/text”.
Correct index file used (defaults to index.html)	Issue GET requests to directory. Verify that “DirIndex” directive is respected by changing “DirIndex” directive, and by changing the index file’s filename. Remove “DirIndex” directive and verify that default index file is “index.html”.
Correct document root used	Issue GET requests, expecting to retrieve files from document root as parent. Change “DocumentRoot” directive and verify that requests use the new root.
Script Aliases working (will be mutually exclusive)	Similar methodology as “Correct document root used”.

Functionality	Test Plan
Correct port used (defaults to 8080)	Same methodology as “Starts up and listens on correct port”.
Correct log file used	Issue requests. Verify loglines are appended to configured logfile path. Change “LogFile” directive and verify newly configured logfile path is used.
CGI Correctly executes and responds	Use “perl_env” button on example webpage works, or try to execute “perl_env” using a batch file if on Windows.
CGI Receives correct environment variables	Execute “perl_env”, verify returned data contains correct environment variables including those specified in spec “SERVER_PROTOCOL”, “QUERY_STRING”, “HTTP_*”
CGI Connects request body to standard input of cgi process	Modify perl script to print out STDIN. Issue CGI request with request body, verify response contains request body data.

Response Codes

Response Code Functionality	Test Plan
200	Tested by issuing GET requests to existing files.
201	Tested by issuing PUT requests to non-existent filepaths, reasoning that issuing PUT requests to existing filepaths should respond with 200 instead.
204	Tested by issuing DELETE requests to existing files.
304	Tested by issuing GET and HEAD requests with an “If-Modified-Header”, using both an If-Modified-Header that returns 200 (because the file has been modified since) or returns 304 (because the file has not been modified since).
400	Tested by using Telnet to issue an incomplete request.
401	Tested by issuing requests to folders with .htaccess, without authorization.
403	Tested by issuing requests to folders with .htaccess, with invalid authorization.
404	Tested by issuing requests to non-existent files.
500	This one was probably the most difficult to test because ideally you don’t want to respond with this, but we managed to see it a couple times while testing the functionality of other parts of the project. (Can also be tested by breaking code so that the server fails)

Project Reflection and Struggles

As with most group projects, the beginning phases of developing a workflow are always difficult, trying not to offend while offering important feedback typically comes slowly at first until you build trust that the other person actually knows/understands the material. Surprisingly, while we had a little bit of a slow start when the project was first assigned, we managed to pick up the pace within the next week and started quickly plugging away, implementing all the requirements listed in the spec as we went.

There were definitely some hiccups along the way, however. One thing we struggled with was trying to decipher exactly what the spec was requiring in certain areas. We also had to manage our expectations a bit and realize that working on certain features would be way outside the scope of the project and frankly, would be a waste of time for everyone involved. That being said, we implemented *a lot* of edge case protection throughout the whole project, which *could* be seen as necessary considering the server runs on an infinite loop which would be seriously hampered by too many errors.