CSC667 Web Server Project

Team: Jacky Qiu Xu(920068055), Lucas Carvajal (922296242)

https://github.com/sfsu-csc-667-fall-2021/webserver-csc667-multithread-server-project

CATEGORY	DESCRIPTION	POINTS
Code Quality	Code is clean, well formatted (appropriate white space and indentation)	5
	Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	5
	Methods are small and serve a single purpose	3
	Code is well organized into a meaningful file structure	2 = 15
Documentation	A PDF is submitted that contains:	3
	Full names of team members	3
	A link to github repository	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
	Brief description of architecture (pictures are handy here, but do not re-submit the pictures I provided)	5
	Problems you encountered during implementation, and how	5

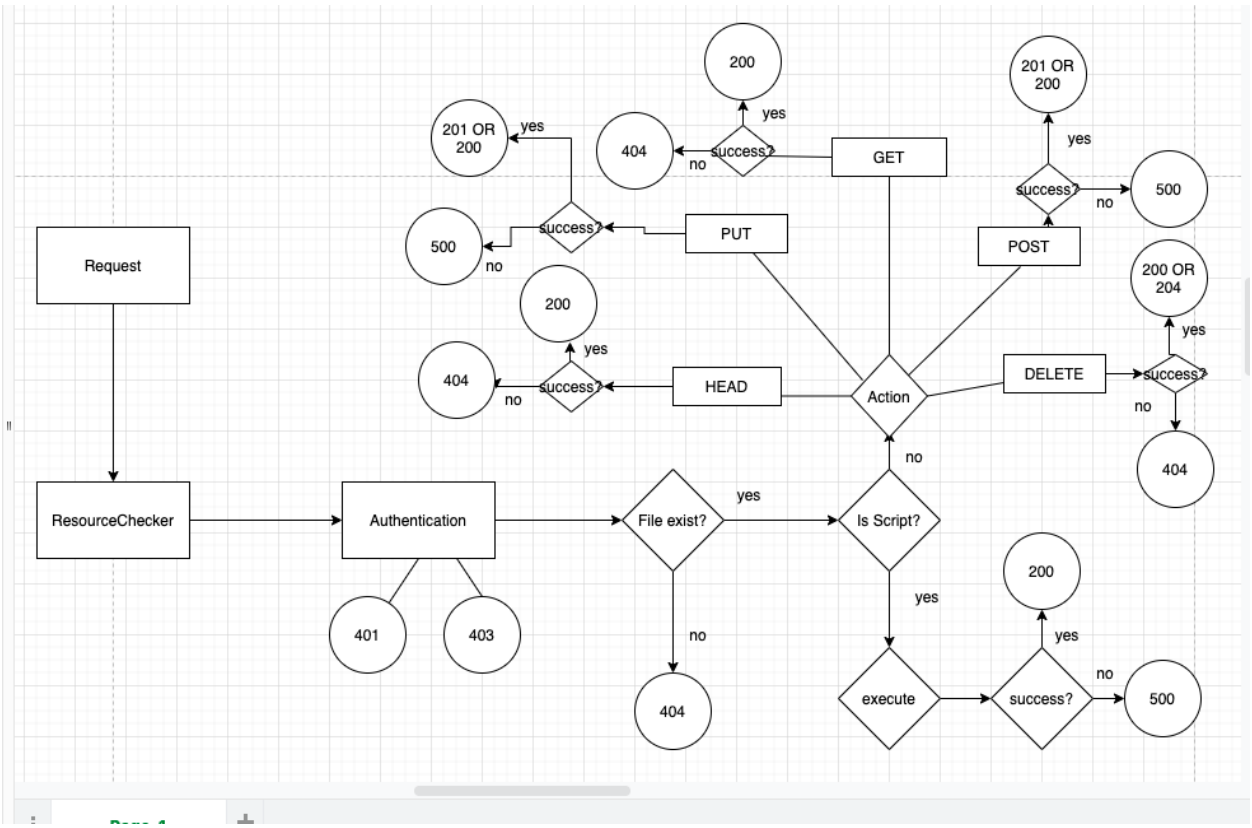
	you solved them	
	A discussion of what was difficult, and why	5
	A thorough description of your test plan (if you can't prove that it works, you shouldn't get 100%)	5 = 30
Functionality- Server	Starts up and listens on correct port	3
	Logs in the common log format to stdout and log file	2
	Multithreading	5
	200	2
	201	2
	204	2
	400	2

	401	2
	403	2
	404	2
	500	2
	Required headers present (Server, Date)	1
	Response specific headers present as needed (ContentLength,	2

	Content-Type)	
	Simple caching (HEAD with If-Modified-Since results in 304 with Last-Modified header, Last-Modified header sent)	1
	Response body correctly sent	3 = 23
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)	1

	Correct htaccess file used	1
	Correct document root used	1
	Aliases working (will be mutually exclusive)	3
	Script Aliases working (will be mutually exclusive)	3

	Correct port used (defaults to 8080)	1
	Correct log file used	1 = 11
CGI	Correctly executes and responds	4
	Receives correct environment variables	3
	Connects request body to standard input of cgi process	2 = 9



Brief Description:

The architecture includes the Web Server, the client/request and the responses. The web server will have a worker that would process all the requests that come in. The server also processes the config file and mime type file so that it can match with the file extension. The server and client are connected together through the worker class and it is also in charge of checking our resources such as the uri and performing accordingly depending on whether the uri is a folder or a file. It is also responsible for sending back responses accordingly after a request is made as well as request authentications.

Problems Encountered:

There were many bugs that were encountered when trying to run the web server which caused compile errors. The problems were resolved through analyzing the exceptions as well as the code that could have possibly led to them. We also had trouble getting the expected output from using Postman and resorted to curl which worked well. We also had difficulties in testing and trying to make the responses work as they should for certain requests and basically went back to try to change some code here and there to see if it works and sometimes it works and sometimes not and the cycle continues.

What was difficult:

The difficulties included testing the request(GET, POST, PUT, etc) services to acquire the expected correct response because it required a lot of analyzing and remodifying of codes. We also had difficulty in trying to render the image of the index.html page because

initially for the GET request, the body was sent back as a string which we thought was the reason why the image didn't render because images are represented as bytes. So we tried writing the body as bytes and it still didn't work.

Test plan description:

Testing started after all the responses were implemented and integrated with the web server. This was considered one of the final steps after setting up the request, server, worker, as well as the authentication. As for testing the application, the GET request with the user authentication works. GET request would send back the headers and body while the HEAD request will only send back the header. More importantly, localhost:8096 renders the body of the file(excluding png files for now) after prompted authentication. DELETE request successfully removes the file from the project folder. PUT request will create a file if the file does not exist and populate the data in there. If the file does exist, all the existing information on that file will be replaced with the PUT request data. As for POST, if the file already exists, it will add on the new information instead of replacing the existing ones and if it doesn't exist it'll create a new file.