WEB SERVER

TEAM MEMBERS:

Sr No.	Name	SFSU ID
1.	Umang Mathur	917927301
2.	Vipul Karanjkar	917924532

Git Repository Link:

https://github.com/sfsu-csc-667-fall-2017/web-server-umangmathur_vipulkaranjkar

Rubric:

Category	Description	**Score (According to us)**	Comments
1. Code Quality	Code is clean, well formatted (appropriate whitespace and indentation)	5	- Code is clean (Proper whitespaces, and tabs used)
	Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	5	- Meaningful class names, and methods used
	Methods/Classes are small and serve a single purpose	3	- Yes

	Code is well organized into a meaningful file structure	2	- Yes, referred to the file structure sent by you on Slack.
Total on Code Quality		15	
2. Documentation	A PDF is submitted that contains:	3	
	Full names of team members	3	- Yes
	A link to github repository	3	- Yes
	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	- Yes
	Brief description of architecture (pictures are handy here, but do not re-submit the pictures I provided)	3	- We followed most of the your workflow. So there are minor changes to our architecture.
	Problems you encountered during implementation, and how you solved them	3	- We've described only two problems, as we were short in time to complete the pdf.
	A discussion of what was difficult, and why	3	- Again the above reason mentioned.
	A description of your test plan (if you can't prove that it works, you shouldn't get 100%)	4	- It explains how to verify various features implemented in our Web Server.

Total on Documentation		23	
3. Functionality - Server	Starts up and listens on correct port	3	- Yes, we used port 7000.
	Logs in the common log format to stdout and log file	2	- Yes, we've saved the logs to a txt file.
	Multithreading	5	- Works correctly.
Total on Functionality Server		10	
4. Functionality - Responses	200	2	- Works
	201	2	- Works
	204	2	- Works
	400	2	- Works
	401	2	- Works
	403	2	- Works
	404	2	- Works
	500	2	- Works
	Required headers present (Server, Date)	1	- Yes
	Response specific headers present as needed (Content-Length, Content-Type)	2	- Yes
	Simple caching (HEAD results in 200 with Last-Modified header)	1	- Yes
	Response body correctly sent	3	- Yes

Total on Functionality Responses		23	
5. Functionality - Mime Types	Appropriate mime type returned based on file extension (defaults to text/text if not found in mime.types)	2	- Yes
6. Functionality - Config	Correct index file used (defaults to index.html)	1	- Yes
	Correct htaccess file used	1	- Yes
	Correct document root used	1	- Yes
	Aliases working (will be mutually exclusive)	3	- Yes
	Script Aliases working (will be mutually exclusive)	3	- Yes
	Correct port used (defaults to 8080)	0	- No
	Correct log file used	1	- Yes
Total on Functionality Config		10	
7. CGI	Correctly executes and responds	4	- Yes
	Receives correct environment variables	3	- Yes
	Connects request body to standard input of cgi process	2	- Yes
Total on CGI		9	

GRAND TOTAL		92	

Test Plan:

i) Testing GET, PUT, POST, DELETE, HEAD request

- Use Postman/Advanced Rest Client chrome apps to send a request to the server for resources such as HTML pages. For example, a GET request to fetch a resource such as index.html can be sent. If the resource is found the appropriate HTML content should be returned in the body along with status code of 200. If the resource in question is not found, the appropriate HTML page should be rendered and the header should contain http status such as 404.
- Send a PUT request using postman to test file creation. Try creating text files and send the content in body. A file with the name matching URL endpoint should be created and it's contents should match the content of the body.
  In the current implementation however, multipart data is not supported and boundary separators will be written to the file unless a 'Raw' body is sent. Ensure that you send a 'Raw' body with every request and not a multipart data request. Similarly, single images/audio or any other file can be sent using the Postman REST client and the appropriate file will be created using the PUT request. This was tested by both of us and all files were reproduced perfectly.
- SImilarly, a DELETE request can be used to check deletion of files created in the previous point.

ii) Testing Caching

- For caching purposes, the 'If-modified' header is sent. If the date sent in this header is lesser than the actual resource file's modification date, then a 200 response should be sent. Else, a 304 response will be received. This can be verified for both GET and HEAD requests. The response of both will be identical except for the fact that the body should always be empty in case of the HEAD request which can be verified by checking the content of the Response body.

iii) Testing Authentication

- For testing authentication, the following steps need to be followed
  - Set name of AccessFileName in the Httpd.conf file.
  - Place access file of this name in any directory which you want to protect.
  - Ensure that you set the AuthUserFile path in the AccessFile.
  - Try to access any file that directory via your browser : You should see an Authorization challenge.
  - If correct username-password is entered, the file contents ill be displayed on the browser or else a 403 status Forbidden page will be shown.

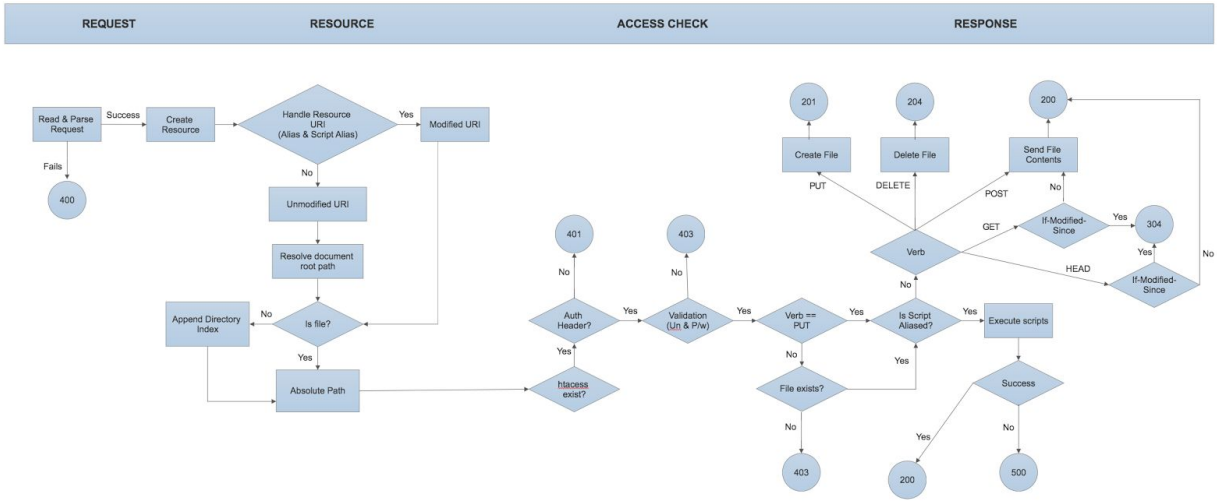
iv) Testing Script Aliasing and CGI

- For testing script aliasing, follow these steps :
  - set appropriate key-value pairs in the config file, ensure alias points to a folder that contains all the scripts for your project.
  - Hit the appropriate URL, the aliasing code will search for the script in the appropriate directory and execute it.

v) Testing Logging

- For testing logging, the following steps need to be followed
  - Set appropriate path for log file
  - Execute requests via Postman or browser and see the results in the text file.

Architecture:



Problems(P) encountered & respective Solutions(S) to it:

P: Reading and parsing without using 3rd party libraries and inbuilt utility classes in Java.
S: Both of us have prior experience of building projects in Java, however we were extremely dependent on using 3rd party libraries for reading and manipulating data. Those libraries are excellent no doubt but tend to dumb things down for programmers by creating a high level of abstraction. In this case, we had to write a lot of boilerplate code and were forced to go through the HTTP specs. This has greatly improved our understanding of the HTTP protocol and how things work under the hood.

P: Favicon Issue
S: This was a really an interesting problem which we faced. We never had the favicon.ico, but the browser always made a request for it. Initially, that was confusing and we thought of removing it, but later we found that it is actually a good thing. Favicon plays an important role in building the brand on their website. Also, it makes it easier to recognize your website and display your brand as a sign of trust.

What was Difficult?

Authentication:
- The example .htpassword file proved to be our guide to the authentication part. The actual implementation was not that difficult as most of the process was explained in the example file. The real difficulty we faced was actually understanding that workflow, i.e how the password in plaintext was encrypted using SHA-1, and later to Base64.

Cgi Scripts:
- We felt this was one of the most difficult parts of the project as we had a very little idea about CGI scripts. So we had to actually study what CGI Scripts are, and how do they work? We understood that CGI scripts will write out HTML content to be viewed. This typically has the structure of the "head" which contains non-viewable content and "body" which provides the viewable content.