Web Server Project

CSC 667/867
Fall 2016

Github: https://github.com/SFSU-CSC-667/web-server-paul-broestl-elaine-phuong.git

Paul Broestl
Elaine Phuong

Server Architecture

- Server uses worker threads to support multithreading
- Factory design pattern is used to generate the different type of responses
- Uses Java 8 to support Base64 encoding for authentication
- Uses ProcessBuilder to execute server-side CGI scripts

Problems encountered during implementation

One difficulty we encountered during implementation was the inconsistencies we found between our server specifications and how Apache designed their server. For instance, in the httpd.conf provided, the values of the Aliases and ScriptAliases contained the DocumentRoot. However, in the httpd that's packaged with Apache, the values only contains the respective folders to be replaced. Furthermore, this conflicted with the server workflow chart as it told us to append the DocumentRoot to the uri that should already contain the DocumentRoot. We solved this issue by walking through our code multiple times and discussing theoretical server setups to help us visualize how the server would traverse through the directory.

Another difficulty we encountered during implementation was executing CGI scripts. Whenever we ran the script, we got “Create Process” errors stating that “%1 is not a valid Win32 application.” We hypothesized that because the shebang operator is not supported on Windows, ProcessBuilder cannot successfully execute the script. We were able to get around this issue by running the server on a Linux virtual machine.

We also encountered several difficulties implementing authentication. One of which was figuring out the correct steps for encoding and decoding the password to and from Base64 and determining the correct hashing algorithm. We initially tried different online SHA-1 generators to get a hashed password that was familiar to the one given in the .htpasswd file. However, we were getting inconsistent passwords until we found a site that supported Base64 encoding. We were eventually able to correctly authenticate the password after some extensive research and trial and error. Another difficulty we faced during implementing authentication was finding the correct .htaccess file in a given absolute path for a resource. After a lot of trial and error and tracing through our code, we were able to narrow down our problem to incorrect splitting of the absolute path and the .htaccess file being incorrectly named.

We also had trouble understanding how sockets were used to communicate between the server and the browser. It was difficult to grasp how a socket is able to grab the uri that is typed from the browser. With the help of our term project members - Alex and Brian - we were able to understand that sockets are constantly receiving and sending out streams that contain the request and response information.

Discussion

Our initial unfamiliarity with how a web server processes responses and requests was our most notable difficulty. Coming from a limited understand of how servers work, it was hard to imagine how each class specifically worked together to produce the functionality of the server. Additionally, the high dependency between each class makes it difficult to test the server when it is only partially complete. Wrapping our brains around how all the various pieces in the server work together was a challenge in of itself. Extensive mapping out of function of paper was how we approached unifying the functions in the server.

Our biggest mistake was trying to complete each class as fully as possible before starting on a new class. This slower approach is fine in theory, but as a result, we were unable to practically test our server until late in the development process. When we did get the server running, numerous bugs we were previously unable to detect appeared. If we were to restart this project, we would have prioritized getting a basic hardcoded server running. Having a base to build off and test from would have helped our development process immensely.

Honestly, procrastination was a significant issue we dealt. We underestimated the amount work necessary to create and test each class. As our server architecture developed, many of our original classes needed to be adjusted. We underestimated how time consuming that iterative process would be, and it left us rushed to finish. Starting earlier would have made a world of difference.

Test Plan

Compiling and running the server

To run our server:

- `javac *.java` in the ServerRoot to compile the files
- `java Server` to execute the server
- Ctrl-C to terminate the server.

Testing GET using the test website

On a browser, enter the uri of any resource that exists on the server. Example: localhost:8096/index.html. For the specified example, the index page and the resources it uses, such as CSS files and images will display.

Testing PUT, DELETE, and HEAD methods using Postman
The Postman app for Chrome was used to test PUT, DELETE, and HEAD responses as these cannot be directly tested using the browser. First start the server, then open Postman.

PUT:

- Select PUT on the dropdown menu
- Enter the uri of the new resource you wish to create. Example: localhost:8096/newpage.html to put a resource named newpage.html in the DocumentRoot.
- Send the request, and you should receive the appropriate response headers.

DELETE:

- Select DELETE on the dropdown menu.
- Enter the uri of the new resource you wish to delete from the server. Example: localhost:8096/newpage.html to delete the resource named newpage.html in the DocumentRoot.
- Send the request and receive the appropriate response headers.

HEAD

- Select HEAD on the dropdown menu.
- Enter the uri of the resource you wish to see. Example: localhost:8096/index.html to view the headers of that resource
- Send the request and receive the appropriate response headers.

Testing multithreading using the test website
Both Firefox and Chrome have automatic caching that allows the large image and threads to load immediately. Caching also causes the threads to execute sequentially, interfering with the thread test. To get around this issue:

Chrome:

- Inspect element, and under the Network tab select “Disable caching”
- Reload the page and select the thread test. The threads should now load asynchronously.

Firefox:

- Inspect element, and under the Network tab, click the gear on the upper right and select “Disable caching” under the advance settings.
- Reload the page and select the thread test..

Testing authentication and 401, 403 responses using the test website
Before starting the server, create a protected folder within public_html and place an HTML file named index.html. This is because the 401/403 test directs you to

localhost:8096/protected/ which doesn’t exist. And because the given httpd.conf uses the default DirectoryIndex, the uri is resolved to localhost:8096/protected/index.html.

Because our server looks for the first .htaccess file in the absolute path - which is the root directory in the test site - the index page of the DocumentRoot will require authentication to view. To properly test authentication on the protected page, move the .htaccess file to the protected/ folder.

Note: Both Chrome and Firefox remember the authentication after the first time. For repeat tests, load the page in incognito windows.

Testing Aliases

On the browser, enter a uri that contains an alias defined in the httpd.conf file. If the uri ends in /ab/ or /ab/index.html, it will take you to the proper page. If the uri ends in a resource that doesn’t exist on the server, it will appropriately bring you to a 404 Not Found page.

Grading Rubric

Grading Rubric

Category	Description		
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 you solved them	5	✓
	A discussion of what was difficult, and why	5	✓
	A 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	10
Functionality - Responses	200	2	✓
	201	2	✓
	204	2	✓
	400	2	✓
	401	2	✓
	403	2	✓
	404	2	✓
	500	2	

Category	Description		
	Required headers present (Server, Date)	1	/
	Response specific headers present as needed (Content-Length, Content-Type)	2	/
	Simple caching (HEAD results in 200 with Last-Modified header)	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