CSE 389 Web Systems Architecture
Fall Semester 2017 Term Project

Option 1: **Web Server using Java**

Arianna Lee, Marcus Robinson, Nigel Heeralall, Sean Dewey

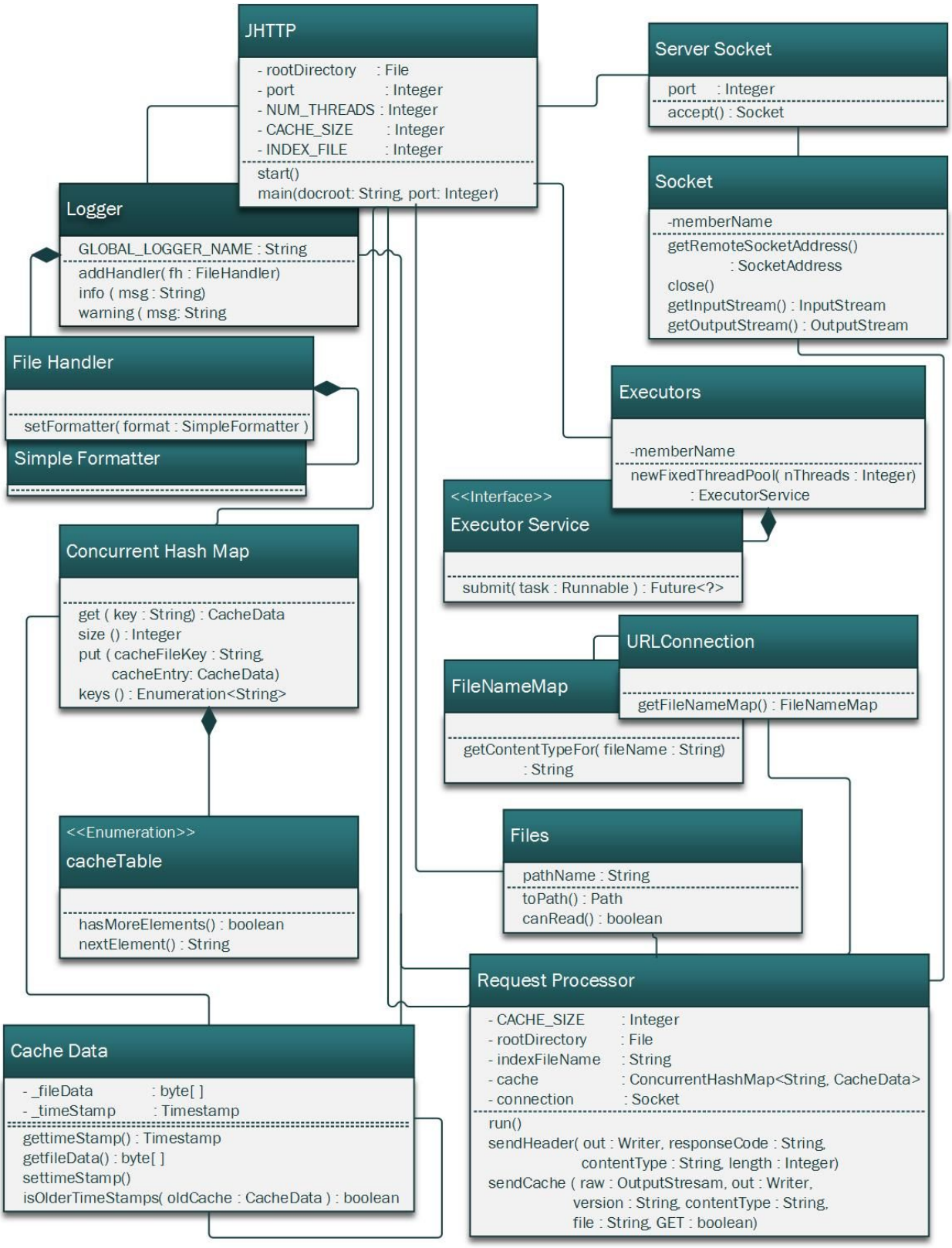
Table of Contents

Introduction

The main objective of this project was to implement a web server using Java that could host a very simple HTML website. The web server used GET and HEAD requests to provide the user with the appropriate web pages. POST requests were implemented to allow the user to enter information into a form to be processed by the server. The information entered is used for Authentication and Authorization. This processes created three situations: unauthenticated, authenticated but not authorised, and authorised. Each situation provided the user with feedback web pages. Also, the use of multithreading allows more than one request to be fulfilled at a single time to the server. As well, the web server uses caching which stores previously requested web pages for quicker accessing. Finally, all actions are recorded server-side using logging.

To successfully complete this project, we had to refresh our memory on the JHTTP.java and RequestProcessor.java classes as this was the base of our project that we build off of. We also needed an understanding of servers, sockets, simple HTML and basic Java. Overall, this project resulted in a deeper understand of what happens behind the scenes of internet services.

System Architecture



Features and Components

GET

HTTPs 'GET' method requests a visual representation of the desired source. GET requests are safe HTTP methods, meaning they don't alter the state of the server in any way. It is also an idempotent method, because any number of the same requests may be made and they will all have the same result, again without having any impact on the server. Lastly, the GET method is cacheable, meaning that GET requests are cached to be retrieved later, so that your computer doesn't have to send another request to the server. For our Project, the GET request was already implemented in the RequestProcessor java file that we were given. The request first verifies that the desired file exists and can be accessed, then retrieves the data on it to a byte array. The data retrieved is then displayed to the user, and the request is over. The GET method in this java file also returns error pages for when the request does not work. If the file cannot be found, then a 404 error is printed to the screen by the request processor.

HEAD

HTTPs 'HEAD' Method is actually identical to the GET method, but it prevents the server from returning a message body. Instead, only the meta information about the desired file is returned to the client. Retrieving data about the file before returning the body is useful because it can be used to determine the size of the file, and whether it exists, before attempting a GET request. Like GET requests, HEAD requests are safe and idempotent. In our implementation, all we had to do to create a HEAD request was remove two lines of code.

POST

The HTTP POST method requests that a web server accept the data enclosed in the body of the request message. The data enclosed in the body is typically stored. Some examples of POST are: signing into a website, submitting a query on Google, and filling out a survey.

For our project, we had to implement the Post method and to test it, we created a simple HTML login form to see if the user can send credentials to the server. The way that posts work in our server is that the user enters credentials in our HTML form and when they hit submit, the data is sent in a request which is handled by the server. Then the server parses the request for the credentials and decides what to do with the data based on the HTML content.

To implement this design, we had to first create an HTML form that sends data to our server. In our previous experience, we have created HTML forms which sends data to php files using the action attribute so we didn't know how to send data to the server at first. To overcome this, we researched html forms and realized that we didn't need the action attribute. By omitting the action attribute the data would be sent directly to the server. To test that we were actually able to send the credentials, we used a Google chrome ad-on that observes HTTP traffic and shows us what was sent in the request. After we verified that, we worked on reading in the POST request to parse it for credentials. We had noticed the given jhttp get code just read in one line of the request so we tried reading more. By using a for loop that ran 200 times, we appeneded the character being read in to a string and displayed the string. The displayed string showed more of the post request so we had to figure out a way to read till the end of the request. We realized that we need to read in the content length data and advance the cursor reading in the file to the credentials line. We developed a solution that would increment a counter if the

character read in was '\n' '\r' and if the counter equaled four then we knew that we were at the credentials. Next, we used the content length to read in all the credential data and used a string split method to get the credentials. Once we got the data, we chose what to do based on the HTML content and that's how POST works on a server.

Multithreading

Multithreading allows for the server to handle more than one request at a time through the use of threads. This allows for numerous users or a single user to make requests concurrently without stalling or corrupting data. In our implementation, all of the threads are managed by an Executors Object. This object manages and schedules a fix number of threads contained in a thread pool. If there are more requests than there are unused threads the Executor will hold them in a queue until a thread becomes available. As the server is only running on one CPU, the requests are not actually running in parallel but they appear to be because of context switching. This is where the balance between having enough threads to support requests and limiting context switching comes into play. If there are too many threads then the CPU is consumed more with switching between them than actually fulfilling the request. For our purposes, JHTTP.java is the main program which allows for the creation of the threads and contains the Executors object. Each threads then runs the RequestProcessor program which actually fulfills the request. Overall, multithreading is an important implementation in allowing web servers to handle the mass amounts of internet traffic.

Authentication and Authorization

According to owasp.com, authentication is "the process of verification that an individual, entity or website is who it claims to be. Authentication in the context of web applications is commonly performed by submitting a user name or ID and one or more items of private information that only a given user should know."

For our project we needed to implement authentication. To do so, we created a simple HTML form that was previously used for posting data to the server. If the client inputted the correct username and password we let them access a special database but if they entered the wrong credentials they were directed to an HTML page that said they had invalid credentials.

To implement our authentication we had to mostly write java code since our server had access to POST data. We simply got the username and password from the post data and stored in variables. We then had to create a text file to serve as an access control list of sorts; it contains username and passwords that would be authenticated. To use the text file we used scanner to read in the strings and stored the data in an ArrayList. We then searched if the user inputted credentials was in the array list and if it was they were sent the database HTML, but if there were no matches on both username and password, they were sent an invalid credentials HTML. Our final product followed authentication protocols.

According to wikipedia, "Authorization is the function of specifying access rights/privileges to resources related to information security and computer security in general and to access control in particular ." For our project we interpreted authorization as restricting someone's access to certain data. As an example, the HR department at a company may have access to employee personal information whereas an employee of that company may not have that information about his coworkers.

To do authorization we took a similar approach as authentication. We created a text file with a list of users who were authorized. In our website, if the user inputted credentials that were authorized, they had edit and delete privileges in our database, whereas if a user didn't have authorized credentials they could only view the database. This meant that we had to create another HTML file that showed edit and delete as an option when viewing the database in contrast of our original file which did not show those options.

To implement this on our server, we had to write the HTML page with edit and delete, and write java to authorize users. Writing the HTML was fairly simple; we added another column to the database with links named edit and delete. For the sake of the requirements, we did not allow the user to actually make changes, we just resolved the links to google and bing.
For the java portion, we read in the list of authorized users and stored them in an ArrayList. If the client's inputted credentials matched to credentials in the ArrayList, they were sent the database with edit privileges. If the credentials did not match, they were sent a view only database.

Cache

Cache is implemented through the use of a ConcurrentHashMap Object. This object implements the abstract map class which lays the groundwork for storing key and value pairings. With the addition of hashing, the speed of the lookup is significantly faster and therefore helps the user experience. Hashing is implemented by assigning a hash value to each entry. In case of multiple matching hash values, all of the values entries will be stored at the single value and must be iterated over. In this implementation, the hashed Map is limited in the number of entries due to minimal amount of web pages that are available to the user. If there reaches a point when all of the web pages are stored in the cache, then it defeats the purpose of implementing it. It is there as a quick access point but should be significantly smaller than the main server memory.

Now, the cache also needed to be concurrent as there are multiple threads that can be accessing it at the same time. The ConcurrentHashMap does not block on retrieving information from the object. However, it may not always provide up-to-date information as 'put' or 'remove' can be used at the same time as 'get'. On the other hand, both 'put' and 'remove' work concurrently and are mostly able to work without contention. Seeing as both of those methods are implemented in this project, it is a key aspect to the overall success.

In our implementation, GET and HEAD requests will first look to see if the requested file is stored in cache. If the program gets a hit, it will simply send the data from there, update the timestamp and finish out the request. If there is a miss, the file will be accessed from file as it was implemented in the given JHTTP.java program. At the end of completing a new access, the cache system will check, using an enumeration of keys (filenames), to see what data in its system has the oldest Timestamp. This timestamp is located in the value portion of the Map entry along with the data. Both of these values are encapsulated by a CacheData Object that we created. Once located, the program will be removed the oldest data from the Map to make room for the new web page data. Again, the request will finish out as normal.

Logging

Logging is the process of keeping a record of all data input, processes, data output and final results in a program. It allowed us to monitor the server activity and log the errors that

happened. It helps us determine where certain problems, such as server responses and client requests, actually occurred. We were are able to log for the following features:

- Authentication
- Authorization
- GET request
- POST request
- Initiation of Server
- Caching

Within the implementation, JHTTP.java initialized a logger. From then on, whenever an action occurred, we used 'logger.info' to write to a log file. For example, when the server starts, the program sends a message to the logger to say that the server started at a specific time. Not only the program sends the message, it also tells the severity of it. In logging, there are seven standard logging levels to control logging output. For example, if the server couldn't start, this would be classified as the highest level, SEVERE. In RequestProcessor.java, this is where logging for authentication and authorization occurred. For these two features, this would be classified as the WARNING level. The levels of logging helps track the simple common problems in order to track and fix them locally.

Data

The data that is being transmitted to our server is username and passwords that clients try to authenticate with. The data that is stored is the list of username and passwords that are authenticated and/or authorized. Other than that, no more data is being stored.

**Note: the cache system in place stores web pages in memory which can be data our server handles.

Input files

To complete this project, we had the following input files:

authenticationList- this file served as a list of users that were authenticated to our website which displayed a database. The file has usernames on one line and passwords on the next line and this pattern repeated for 5 test users. The server reads in the file and compares its content to client's submitted credentials.

authorizationList- this file served as a list of users that were authorized on our database. The file has usernames on one line and passwords on the next line and this pattern repeated for 3 test users. The server reads in the file and compares its contents to client's submitted credentials and if they match, they can make changes to the database.

HTML webpages

The following are our web pages that we used to test our server:

Index.html- This was used to prove that we could manage Get and Post requests. This webpage displayed a simple login form that asks for username and password. We got the

code for the web page from codepen.io and altered it to suit our needs. We changed the HTML form to send data to our server instead of a php page.

authorizedHTML- This was used to prove that our server could perform authorization. This webpage included HTML that would allow the user to edit or delete content in the database if they entered the correct credentials.

notAuthorizedHTML- This HTML file was used to prove that our server could perform correct authorization, If the user did not have edit or delete privileges they had read only rights on the database.

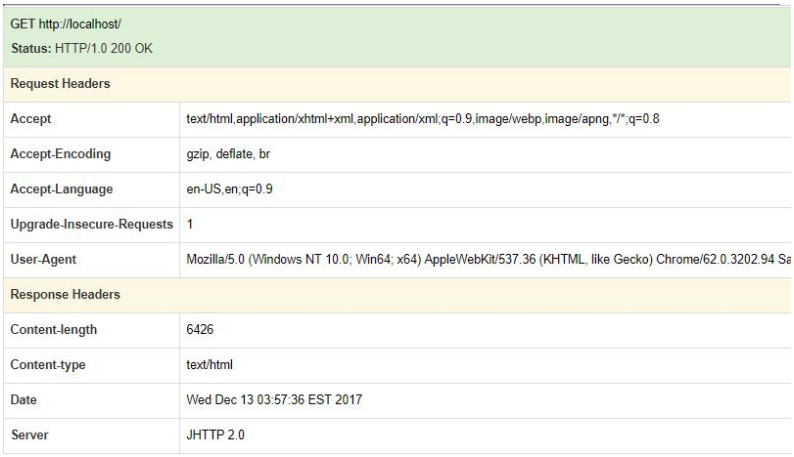
Authenticated-This HTML page was used to prove that our server could perform authentication. The file is displayed to the user once the server verifies that the client's credentials are correct.

Output files

LogFile.log- This file is used to keep track of events on our server. For example, once a client authenticates to our webpage, we log that for information purposes. We also log if a client makes an incorrect login attempt and we log all the HTTP requests being made. Generally, we log when a client interacts with the server or if code within the server breaks.

Experimental Results

GET Request



GET http://localhost/ **Status:** HTTP/1.0 200 OK	
Request Headers	
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding	gzip, deflate, br
Accept-Language	en-US,en;q=0.9
Upgrade-Insecure-Requests	1
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Sa
Response Headers	
Content-length	6426
Content-type	text/html
Date	Wed Dec 13 03:57:36 EST 2017
Server	JHTTP 2.0

Figure depicts the GET request made by the client observed by HTTP Headers (Google Chrome ad-on)

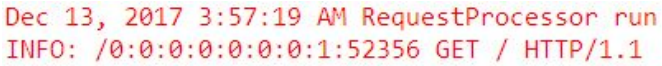


Figure depicts the server receiving the GET request

HEAD Request

```
> testHEAD = function(url) {
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open("HEAD", url, true);
    xmlhttp.onreadystatechange=function() {
       if(xmlhttp.readyState==4) {
         console.log(xmlhttp.getAllResponseHeaders());
       }
    }
    xmlhttp.send(null);
  }
< ƒ (url) {
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open("HEAD", url, true);
    xmlhttp.onreadystatechange=function() {
       if(xmlhttp.readyState==4) {
         console.log(xmlhttp.getAllResponse…
> testHEAD('')
```

Figure above shows the javascript code used to make a Head request

```
Nov 30, 2017 7:33:07 PM RequestProcessor run
INFO: /0:0:0:0:0:0:0:1:59708 HEAD / HTTP/1.1
```

Image shows the server receiving a Head request

```
date: Thu Nov 30 19:33:07 EST 2017
server: JHTTP 2.0
content-length: 130
content-type: text/html
```

Figure shows the server's response to the client's request

POST Request

POST http://localhost/ **Status:** HTTP/1.0 200 Good	
Request Headers	
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding	gzip, deflate, br
Accept-Language	en-US,en;q=0.9
Content-Type	application/x-www-form-urlencoded
Origin	http://localhost
Referer	http://localhost/
Upgrade-Insecure-Requests	1
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Saf
Response Headers	
Content-length	110
Content-type	text/html; charset=utf-8
Date	Sun Dec 03 14:57:39 EST 2017
Server	JHTTP 2.0

Figure depicts the Post request made by the client observed by HTTP Headers

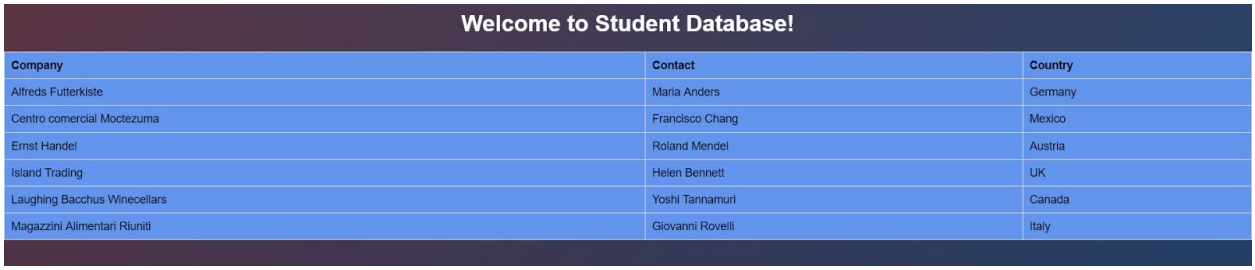
```
Dec 03, 2017 2:52:28 PM RequestProcessor run
INFO: /0:0:0:0:0:0:0:1:56108 POST / HTTP/1.1
```

Figure depicts the server receiving the GET request

Multithreading

The fixed number of threads allowed during run time is 30.

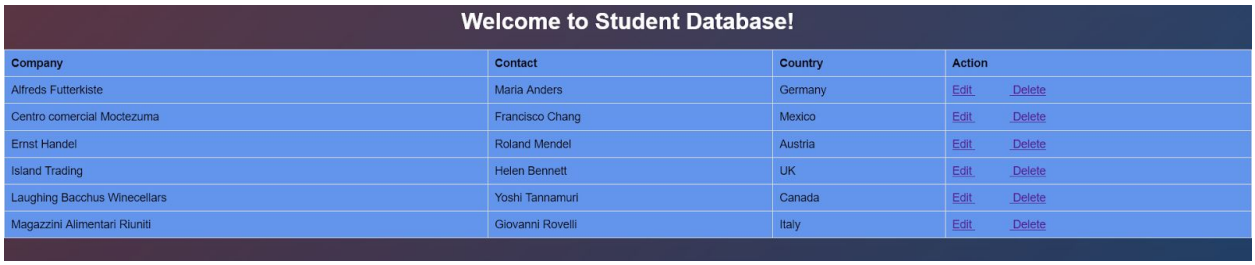
Authentication and Authorization



Welcome to Student Database!

Company	Contact	Country
Alfreds Futterkiste	Maria Anders	Germany
Centro comercial Moctezuma	Francisco Chang	Mexico
Ernst Handel	Roland Mendel	Austria
Island Trading	Helen Bennett	UK
Laughing Bacchus Winecellars	Yoshi Tannamuri	Canada
Magazzini Alimentari Riuniti	Giovanni Rovelli	Italy

Resulting web page for someone Authenticated but not Authorized



Welcome to Student Database!

Company	Contact	Country	Action
Alfreds Futterkiste	Maria Anders	Germany	Edit Delete
Centro comercial Moctezuma	Francisco Chang	Mexico	Edit Delete
Ernst Handel	Roland Mendel	Austria	Edit Delete
Island Trading	Helen Bennett	UK	Edit Delete
Laughing Bacchus Winecellars	Yoshi Tannamuri	Canada	Edit Delete
Magazzini Alimentari Riuniti	Giovanni Rovelli	Italy	Edit Delete

Resulting web page for someone Authorized

You have invalid credentials

Resulting web page text for incorrect login information (not Authenticated)

Cache

The number of cache elements allowed in the cache map is two. Once the map contains two web pages and someone wishes to access one that is not inside the map, the one with the oldest timestamp is replaced.

```
INFO: /0:0:0:0:0:0:0:1:7914 GET / HTTP/1.1
Dec 13, 2017 11:41:20 AM CacheData <init>
INFO: Timestamp is: 2017-12-13 11:41:20.598
Dec 13, 2017 11:41:20 AM RequestProcessor run
INFO: Caching /index.html
```

Figure depicts server caching a file after a GET request

```
INFO: /0:0:0:0:0:0:0:1:7918 GET / HTTP/1.1
Dec 13, 2017 11:41:40 AM RequestProcessor sendCache
INFO: Loading /index.html from cache
Dec 13, 2017 11:41:40 AM CacheData settimeStamp
INFO: New timestamp is: 2017-12-13 11:41:40.211
```

Figure depicts server using data from cache to fulfill GET request

Logging

```
SEVERE: Server could not start
java.net.BindException: Permission denied (Bind failed)
  at java.net.PlainSocketImpl.socketBind(Native Method)
  at java.net.AbstractPlainSocketImpl.bind(AbstractPlainSocket
  at java.net.ServerSocket.bind(ServerSocket.java:375)
  at java.net.ServerSocket.<init>(ServerSocket.java:237)
  at java.net.ServerSocket.<init>(ServerSocket.java:128)
  at JHTTP.start(JHTTP.java:27)
  at JHTTP.main(JHTTP.java:83)
```

An error being logged

```
Dec 03, 2017 4:21:02 PM JHTTP start
INFO: Accepting connections on port 1024
Dec 03, 2017 4:21:02 PM JHTTP start
INFO: Document Root: ./www
```

Logged information from web server startup

Analysis

GET

Completing GET was simple since JHTTP.java was given to us and it already implemented the GET functionality. The only thing we needed to to figure out is how to start the server.

HEAD

In order to complete HEAD, we had to first understand what the head request asked for and what we should send to the client. After we learned that, we got the data we needed from the GET request which was given to us. We noticed we only needed to get rid of two lines of code, the code that sent the HTML page to the user. We needed to send the content length so we still had to read in the file but we don't send it in the HEAD response.

A challenge part of this was figuring out how to prove that our server can handle HEAD requests. We knew that web browsers by default just send out GET requests. We were thinking about potential options like using curl on linux and set some parameters. But then we realized that we can use javascript to make an HTTP request and hopefully make it a head request. Luckily, we found javascript code on stackoverflow that sent a head request and he needed to alter it to point to our localhost. After sending the request, we saw the response which is what we set the server to send. We also saw that the server got a head request. These two pieces proved to us that we correctly implemented HEAD.

POST

By far, this HTTP method was the hardest to implement. It required us to know about the format of the request and to understand how the given JHTTP code was reading requests. Also, this method required us to understand how an enter (new line) is represented with '\r' and '\n'. Once we analyzed the JHTTP code we hypothesized that if we kept reading more from the request data, we could find the entire request; prior to this only the first line was read in. After we proved that we could read all of it we needed to figure out how get the content length and get the content data. This was hard to figure out at first but luckily we came up with an idea to count the number of '\n' and '\r' we encountered. If we hit four in a row then we know we hit the content. After overcoming this, we were quickly able to use string methods to get the content data.

One thing our group thought about was whether or not the server should handle post data or if it should send it to another java class that would handle authentication for example. We figured for the scope of this project that leaving it in the server would suffice but in the real world, there is probably another program that handles post data.

Multithreading

For multithreading, the majority of the code was already given to us from Java Network Programming, 4th Edition. As mentioned in the Feature section, we needed to choose a value for the number of threads that we wanted to allow. The default value that was given was fifty. After some research, we found that for a single CPU that is implementing context switching, it is best to limit it to thirty or forty threads. For that reason, we chose to allow for a maximum of 30 threads. Overall, this project was on a small enough scale that context switching vs. thread/request ratio was not a main factor in our implementation.

The biggest issues that we saw arising from multithreading was when other sections such as POST, logging, and cache needed to edit information that all of the threads could see. Thankfully, the logger class already implements multi-thread safety. Therefore, there should never be a conflict when writing to the the console or to a file. As well, we chose to use the ConcurrentHashMap for this exact reason. There were various other Map implementations available but this particular class accounts for multi-thread safety. It does not block on reading but it does block on adding and removing elements from the map.

When it came to POST requests, we chose not to handle deep multi-threading issues. The typical issue that we would see with POST requests and multiple threads is when more than one threads tries to update a single document at the same time. Simultaneous accesses would cause the corrupted data. However, in our implementation of the web server, we do not allow users to edit the same information. Our POST request lets the user have access to the various web pages by comparing information that they entered with a database. They can all read the database concurrently, but as long as no one is editing it, then there isn't a way for users to corrupt it. Therefore, our project had no need for the additional feature. However, this is somewhere that we can improve upon at a later date. Once we implement the ability to use the edit buttons available to authorized users, we would need this type of multi-threading safety.

Authorization and Authentication

When starting out the project we were confident in our algorithm for doing authentication; we planned on storing usernames in a text file and checking against client input

data. When it came to implement authentication, the hardest part was actually completing POST. Thinking of that algorithm for that was harder than reading in a text file and comparing it to user inputted data. But once post was done, we stored a text file's contents in an ArrayList and check against the data we got from the client.

The easiest way we could have tested authorization was making an admin interface for CRUD (create, read, update, delete) operations on our fake database. As a result, we created another HTML page that allow authorized users to have edit and delete privilege.

Cache

As mentioned above in Features, the purpose of having cache is to allow for quick access to the most common pages. Seeing as our website only contains about seven or eight different web pages, we did not think it was appropriate to store more than two items at a time in the cache. Just as a computer's RAM is smaller than their main memory, our cache is smaller than the amount of pages that we have to access.

In our implementation, we chose to only save pages that were requested by the user. This means that cache is only used for GET and HEAD requests. We felt as though the authorised and authenticated pages should not be caches as they were private pages. As well, these were just a response to sending data. For the same reasoning, we did not cache the error pages either. All of them are simply created on the web server and do not need to be accessed from somewhere else anyways. Therefore, the time efficiency would be about the same between caching errors and producing the error page.

Overall, the implementation of cache was not extremely difficult. The biggest learning portion was figuring out how to implement a Map object. We knew the basics of key and value pairing but the syntax between coding languages is always slightly different.

Logging

We mainly had issues based upon logging for everything but the initiation of the server. We placed the logging code for the initiation of the server in the JHTTP.java file. In there, we use 'logger.info' to write to the file. However when we use the same syntax in the RequestProcessor.java file, it didn't log the errors to the file at all. All of the loggers were placed in each 'else' statement to show if the user was not authorized/authenticated, can't find a file, etc.. Therefore, we switched to the statement 'logger.warning.' This was done because we wanted the messages to be on a warning level. Once done, the computer was able to recognize the warning and wrote it to the log file.

Team member contributions:

Nigel

Nigel created the index.html page, authentication/authorization web pages, implementing post request, and authorization/authentication. Wrote README file.

Sean

Sean implemented HEAD request. Assisted in developing HTML pages and designed 5 more web pages for cache purposes.

Marcus

Marcus handled the tracking of common problems. Logging was used to write the errors that occured to a log file.

Arianna

Arianna handled the implementation of both multithreading and cache. These two items went hand in hand because for the cache accessing and updating, thread conflicts had to be taken into consideration. Also helped with post requests. authentication/authorization and HTML forms. As well, she created the UML seen under System Architecture in this document.