

Nicholas Szeto(915451464)

Tommy Tran(916265095)

February 20, 2017

CSC 667-01 John Roberts

GitHub Repository: <https://github.com/sfsu-csc-667-spring-2018/web-server-lookin-like-a-snack>

Assignment One - Java Web Server

Code Quality

- [x] Code is clean, well formatted (appropriate whitespace and indentation)
- [x] Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)
- [x] Methods are small and serve a single purpose
- [x] Code is well organized into a meaningful file structure

Documentation

- [x] A PDF is submitted that contains
 - [x] Full names of team members
 - [x] A link to github repository
 - [x] 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)
 - [x] Brief description of architecture (pictures are handy here, but do not re-submit the pictures I provided)

- [x] Problems you encountered during implementation, and how you solved them
- [x] A discussion of what was difficult, and why
- [x] A thorough description of your test plan (if you can't prove that it works , you shouldn't get 100%)

Functionality - Server

- [x] Starts up and listens on correct port
- [x] Logs in the common log format to stdout and log file
- [x] Multithreading

Functionality - Responses

- [x] 200
- [x] 201
- [x] 204
- [x] 400
- [x] 401
- [x] 403
- [x] 404
- [x] 500
- [x] Required headers present (Server, Date)
- [x] Response specific headers present as needed (Content-Length, Content-Type)

[x] Simple caching (HEAD with If-Modified-Since results in 304 with Last-Modified header, Last-Modified header sent)

[x] Response body correctly sent

Functionality - Mime Types

[x] Appropriate mime type returned based on file extension (defaults to text/text if not found in mime.types)

Functionality - Config

[x] Correct index file used (defaults to index.html)

[x] Correct htaccess file used

[x] Correct document root used

[x] Aliases working (will be mutually exclusive)

[x] Script Aliases working (will be mutually exclusive)

[x] Correct port used (defaults to 8080)

[x] Correct log file used

CGI

[x] Correctly executes and responds

[x] Receives correct environment variables

[x] Connects request body to standard input of cgi process

Introduction

The following project is to write a basic web server written in Java. We were tasked with building a http web server to learn and understand how web servers work and how HTTP is used to serve files/scripts over the internet. There was no skeleton code given to us, but we were provided a UML diagram of the project to give us an idea of how the finished product should look like. There was also some discussion in class about certain functionality.

Overview

As being tasked with writing an Http web server in Java we needed certain functionality and behaviors similar to that of a real Http server. Given a UML of a suggested route for the web server and a brief lecture of how servers work, we were tasked to write one from scratch that is able to respond to five types of http requests and send nine types of http responses. The request types are GET, HEAD, PUT, POST, DELETE. The http responses are 200 “OK”, 201 “Created”, 204 “No Content”, 304 “Not Modified”, 400 “Bad Request”, 401 “Not Authorized”, 403 “Forbidden”, 404 “Not Found”, and 500 “Server Error”.

The Server needs to load two files before it can take request, and those are the configuration file and the mime types file. It then opens a Socket at a port number given by the configuration, waiting for a http request. We read this request through an input stream and parse it to get information and to send the appropriate response. The URI given in a request needs to be checked and resolved for things like aliases and script aliases. After the URI of the request is resolved, the server checks if the file in the path needs authorization. If so, a 401 response is sent until proper authentication is given.

After authentication for the htaccess and htpasswd files, the request is further processed depending on the verb given, and a http response is sent back to the client. Each request received is processed by a thread, that way the server does not hang and process one request at a time. Since there are over nine responses, we used the factory design pattern that we were advised to use which made it easier to write and debug our response classes.

Milestone 1:

The first milestone is create classes to load dependencies for the web server to use and start up such as client information, port to listen, and required information to be sent back. These files would need to be read through and parsed before the server starts to take requests from the client. These dependencies would be read from a folder named conf and it would contain two files named “httpd.conf” and “mime.types”. The http.conf tells information and directories, port to listen to and aliases to file paths. The mime.types will tell what type the file requested would be sent back should be from the extension of the file through the headers returned in the response.

Milestone 2:

Next would be the creation of a class with the ability to parse through to the five different http request: GET, HEAD, POST, PUT, and Delete. There will be slight differences and cues that would distinguish each request apart.

Milestone 3:

After gathering information on the request from the client we would then need to generate different responses for these five http requests. There would be different response codes: 200, 201, 204, 304, 400, 401, 403, 404, 500. There would be different headers in response to each of these codes.

Milestone 4:

The next milestone would be the ability to parse through and respond to multiple requests through the use of threads.

Milestone 5:

Then executing server side scripts from a requested path that was aliased. Our scripts should be able to run perl scripts and be sent back through the browser for the script result to be displayed on the browser.

Milestone 6:

Authentication is next and it would look for a .htaccess file in the same tree directory as the file requested. If that file requested exists a response of 401 would be sent asking for the user to input credentials to be checked by the credentials already loaded onto the server from the .htpassword file. If the correct information matches it would then would allow the user to get access to the requested file and if not a 403 request would be sent back which is forbidden.

Milestone 7:

An update to GET and HEAD to include caching would require us to look for headers of last modified and if the date. If there isn't any change from the date from the header and the file checked then it would send a 304 which means don't send any information through for that file since the client already has the most updated version.

Milestone 8:

The final milestone is appending to a log file with the path from the loaded dependencies from milestone 1.

Development Environment:

The planning process and development was completed on Mac OS High Sierra . The program is written in Java and was developed on JetBrains' IntelliJ 3.4 IDE with Java 8 on update 131.

Command Line Instructions

- 1. `javac WebServer.java`**
- 2. `java WebServer`**

Instructions:

Step 1:

Download the needed files from the repository:<https://github.com/sfsu-csc-667-spring-2018/web-server-lookin-like-a-snack>

Step 2:

Unzip the files with your preferred file extractor

Step 3:

Place directory to a location that easy for you to access

Step 4:

Open your command prompt/ terminal

Step 5:

Navigate through to the /web-server-lookin-like-a-snack-master/ folder in directory in your preferred location using “cd ..” command to change directories to a familiar location and then “cd insertDirectoryHere” until you get to the preferredLocation/web-server-lookin-like-a-snack-master/src/

Step 6:

Run the command line instructions from 1 and 2.

Assumptions

We assumed that completing this assignment would be a tough task, since there were several talks of how students from previous semesters did not complete it on time. Knowing this, we started the project as early as we could, reading up on apache documents and Java libraries to draw out how the design of the server should look like. After some discussions with classmates and guidance from peers and Jrob, we started by following the milestones in the specifications. Some things that were unclear when we started the project were how to connect the server to the client in the first place, so we just reused code from the simple server and simple client shown in class.

In the Request class there was not many assumptions at all as it seemed pretty straight forward. The only thing assumed was the type of the body, which we decided to make it into a byte array as reading an input stream would be in bytes.

In the Resource class we had to assume that it would hold information about the file that was read in. Like the modified date and path and location. This would be attributes of the file that was requested. When there was a file check in the diagram we assumed that it meant a directory as if the file didn't exist it would append the directory index. This would cause problems for a PUT request as creation of a file would be given the wrong path.

In the authentication we had to assume that Jrobs provided htpassword would be easy to understand and use, which it was. Using that we would be able to make htaccess and have it send back responses of 401 and 403. In 401 and 403 we assumed that if anyone of these responses would be sent back then no information about the resource should be sent back in the headers as they do not have access to that information.

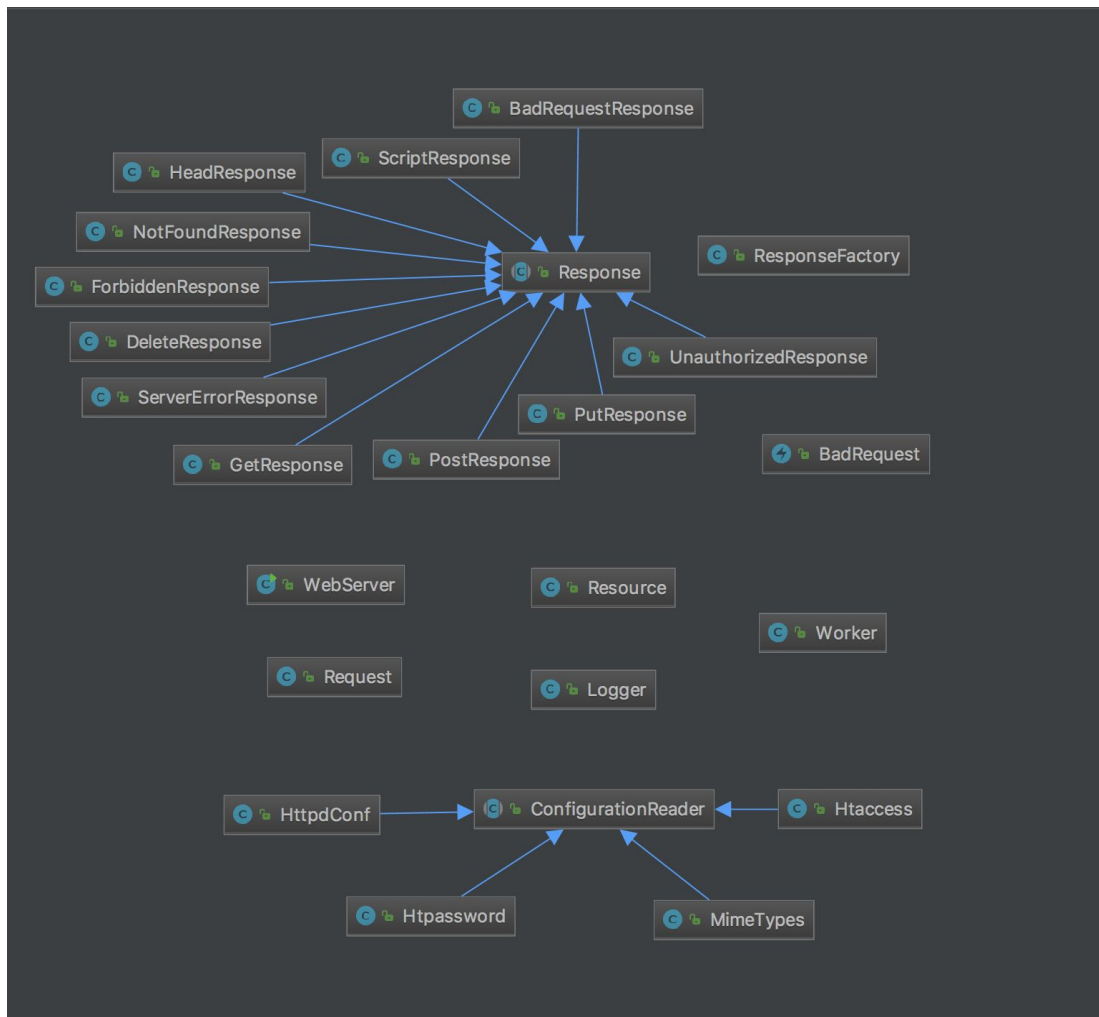
404 would also bring back default information like the date and the server name.

200 and 304 would return the same headers which includes the content length, content type, last modified, date and server name. This was assumed as the web browser would need this information to display the resource and to use the cached version or not.

For PUT and DELETE there are other responses that can be sent back but we assumed that we should just follow the diagram provided for us.

POST request we interpreted it as appending to an existing file.

Implementation and Discussion



(Overall class diagrams for web server)

WebServer		
f	HTTPD_CONF_PATH	String
f	MIME_TYPES_PATH	String
f	DEFAULT_PORT	int
f	socket	ServerSocket
f	configuration	HttpdConf
f	mimeTypes	MimeTypes
WebServer()		
m	loadConfigurationFiles()	void
m	start()	void
m	listenToPort()	void
m	getPortNumber()	int
m	main(String[])	void

The web server class starts and has the path to the dependencies like the conf file and mime types. A constant string has this path hardcoded to have that available. The default port is also there to have the server listen to that number if no port is given. The web server creates all the objects needed when starting up.
































Worker		
f	client	Socket
f	mimes	MimeTypes
f	config	HttpdConf
Worker(Socket, HttpdConf, M		
m	run()	void

The worker class gets the socket connection and starts reading from the stream and starts building the request, resource, response from the response factory and sends back the response back to the stream to be interpreted by the web browser. A logger is recording the response and









request information in common log format. Then the connection is closed to be started again in the while true loop. A copy of the mime types was sent into the factory because there are different responses that can require the mime types in their response.

Request	
InputStream	InputStream
InputStreamReader	BufferedReader
VALID_HTTP_VERBS	HashMap<String, Boolean>
VALID_HTTP_VERSIONS	HashMap<String, Boolean>
Request(InputStream)	
parseRequest()	void
parseFirstLine()	void
isValidHTTPVerb(String)	boolean
isValidHTTPVersion(String)	boolean
isFirstLineOfRequest(String[])	boolean
setFirstLineOfRequest(String, String, String)	void
parseHeaderSection()	void
parseHeader(String)	void
parseBodySection()	void
hasQueryString(String)	boolean
parseQueryString(String)	void
lookup(String)	String
queryString	String
body	byte[]
uri	String
httpVersion	String
firstLineofRequest	String
verb	String
headers	HashMap<String, String>

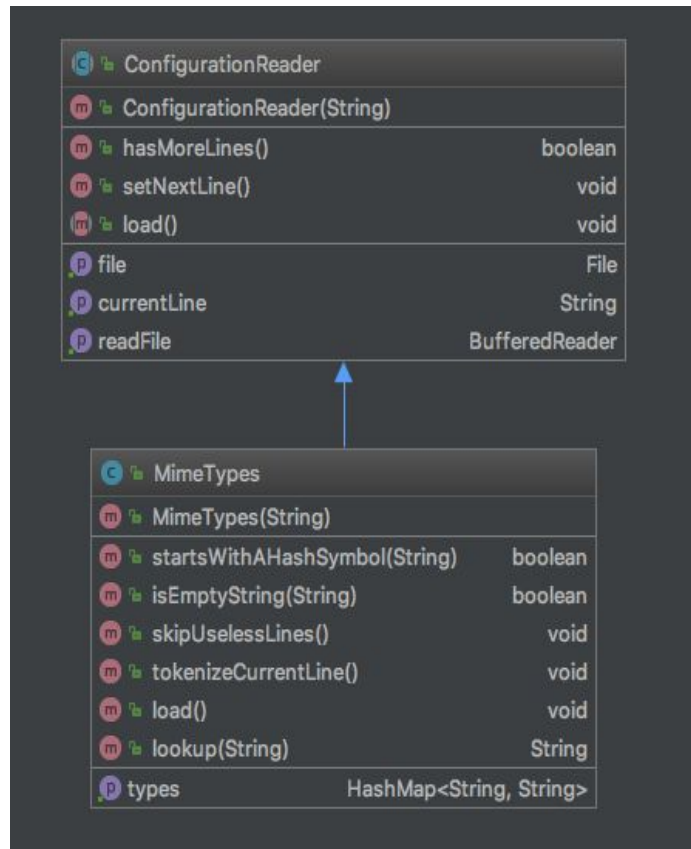
The request would receive the stream that the worker has and read line by line until it got to a carriage return line feed. The stream would be read and parsed through taking in information needed. The stream reading is done by reading the stream by the bufferedreader object. Once the reading is over it will return and the worker will move onto the resource. There would be different conditionals to call other functions like parsing a the body only if there was the verb put or post.

 Resource	
  configuration	HttpdConf
  fileURI	String
  isAlias	boolean
  Resource(String, HttpdConf)	
  absolutePath()	String
  isAliased()	boolean
  addDocumentRootToTheStartOfURI()	void
  appendDirectoryIndexToURI()	String
  isDirectory()	boolean
  uriContains(HashMap<String, String>)	boolean
  modifyURI(HashMap<String, String>)	void
 file	File
 script	boolean
 lastModified	long
 protected	boolean
 URIDirectoryTree	String
 lastModifiedDate	Date
 absolutePath	String
 htAccessLocation	String

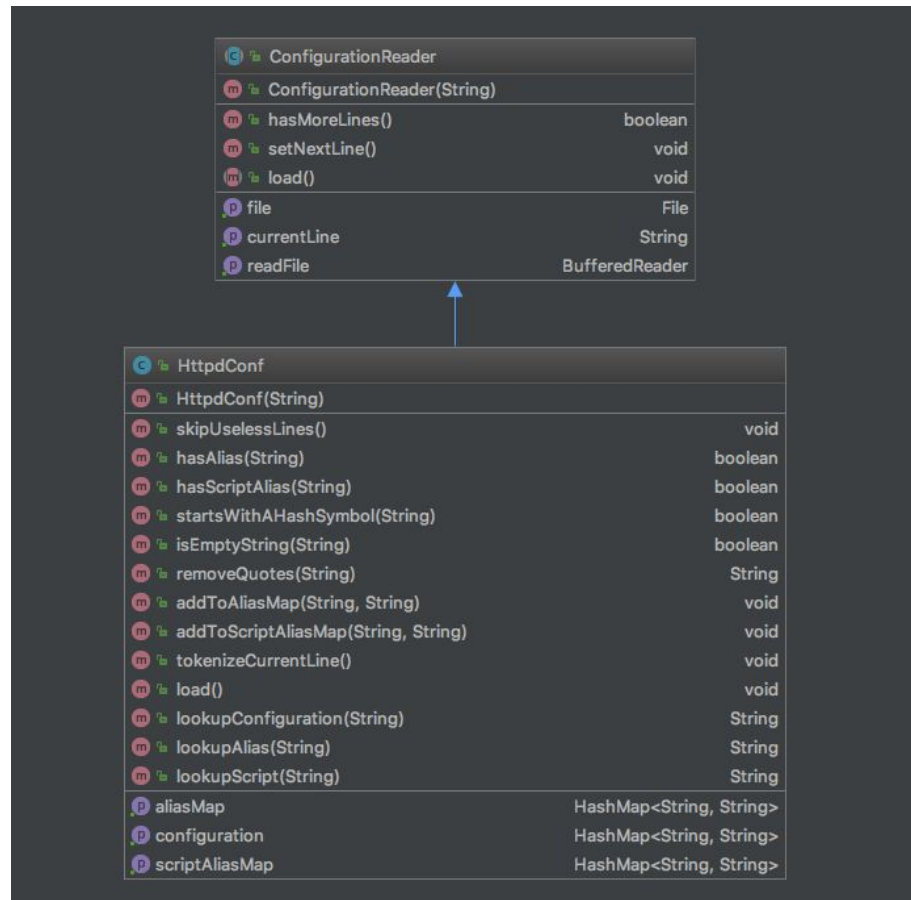
The Resource class is responsible for resolving the URI received from the Request class. The unmodified uri goes through several checks to make sure that it finishes as an absolute path on the file system. This class takes note if the script is aliased and will tell the response factory to send a script response.

	ConfigurationReader	
	ConfigurationReader(String)	
	hasMoreLines()	boolean
	setNextLine()	void
	load()	void
	file	File
	currentLine	String
	readFile	BufferedReader

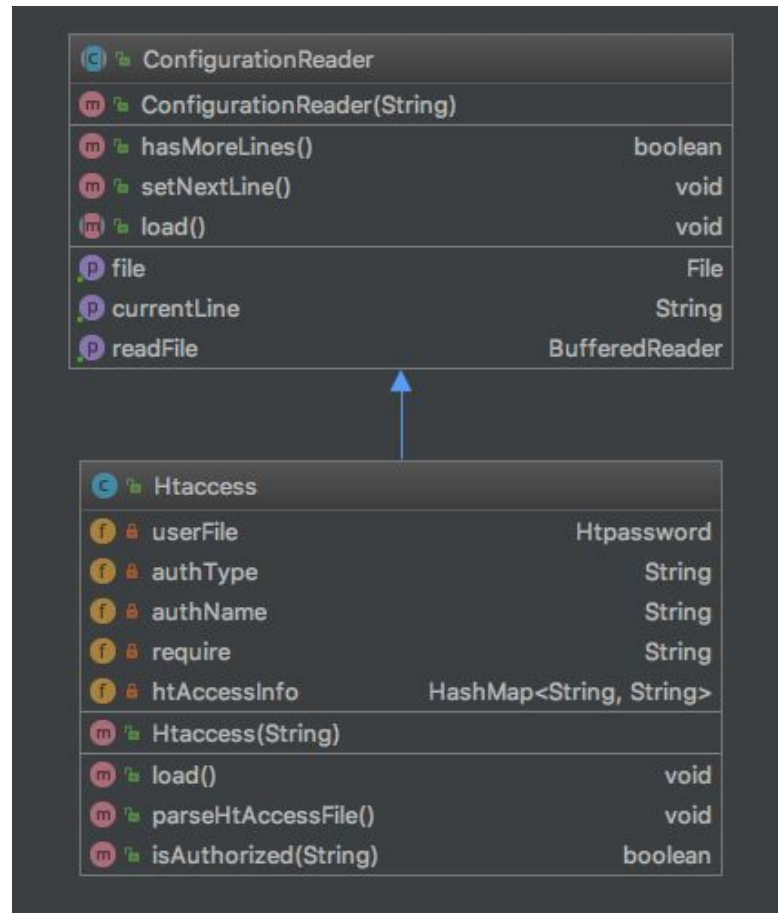
The configuration reader is an abstract class to be extended to read in a file and load information like the different dependencies needed for that specific file type. There are methods that would read each line and set the new line until the file is empty. Lines would continue reading until there is a line read that was null.



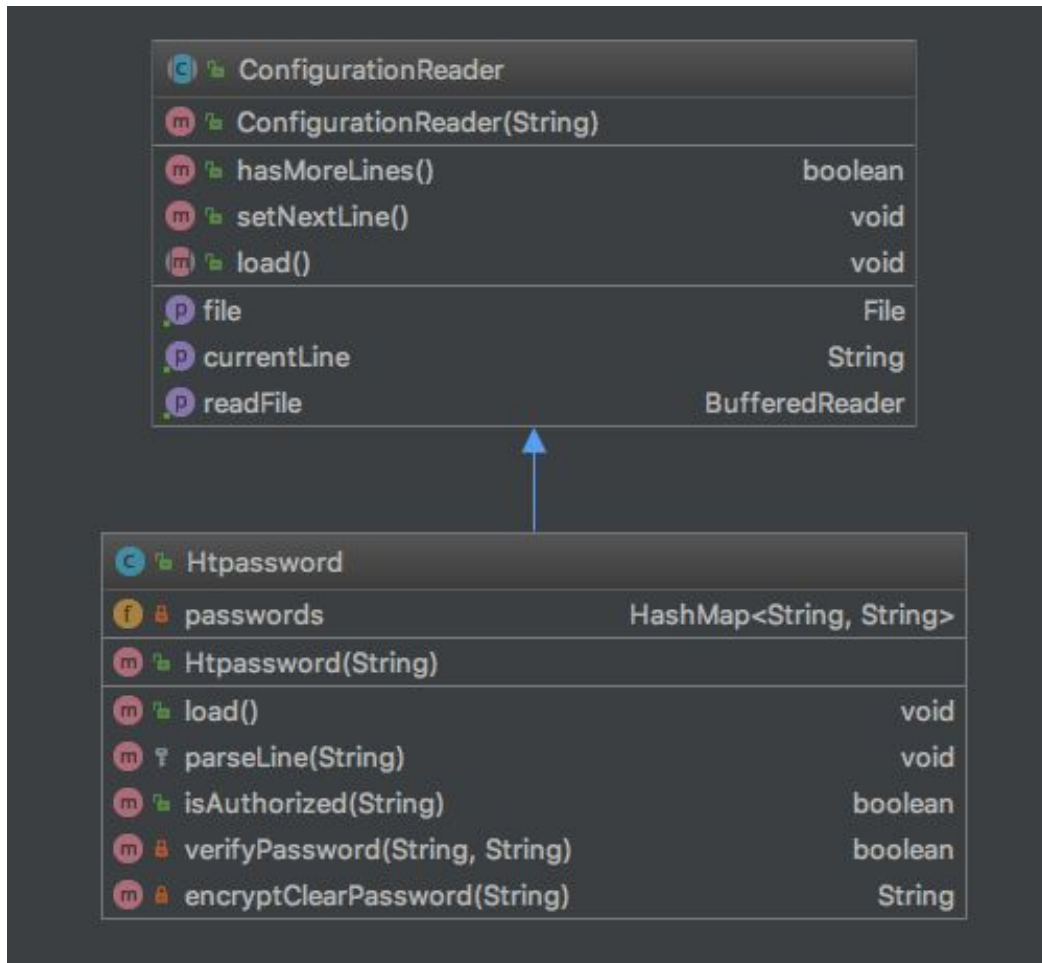
The mime types class extends the configuration reader to have the ability to read and load dependencies. These dependencies would be loaded into a hashmap which would be keyed by the end extension of the file requested. The value is the needed format for a header that a browser needs to interpret a file. The mime types file would have useless lines to skip over like empty spaces and hash marks. This class would be used for headers to send back in a response.













The **HttpdConf** class is used to read the servers configuration file. This configuration file holds key server information like the port number, server aliases, directory index, etc. The class stores all of the information of the configuration file into a hashmap, this allows for other parts of the server to access information needed from the file. The alias map and script alias map are separate from the configuration map so that information could be more discrete amongst other parts of the server.



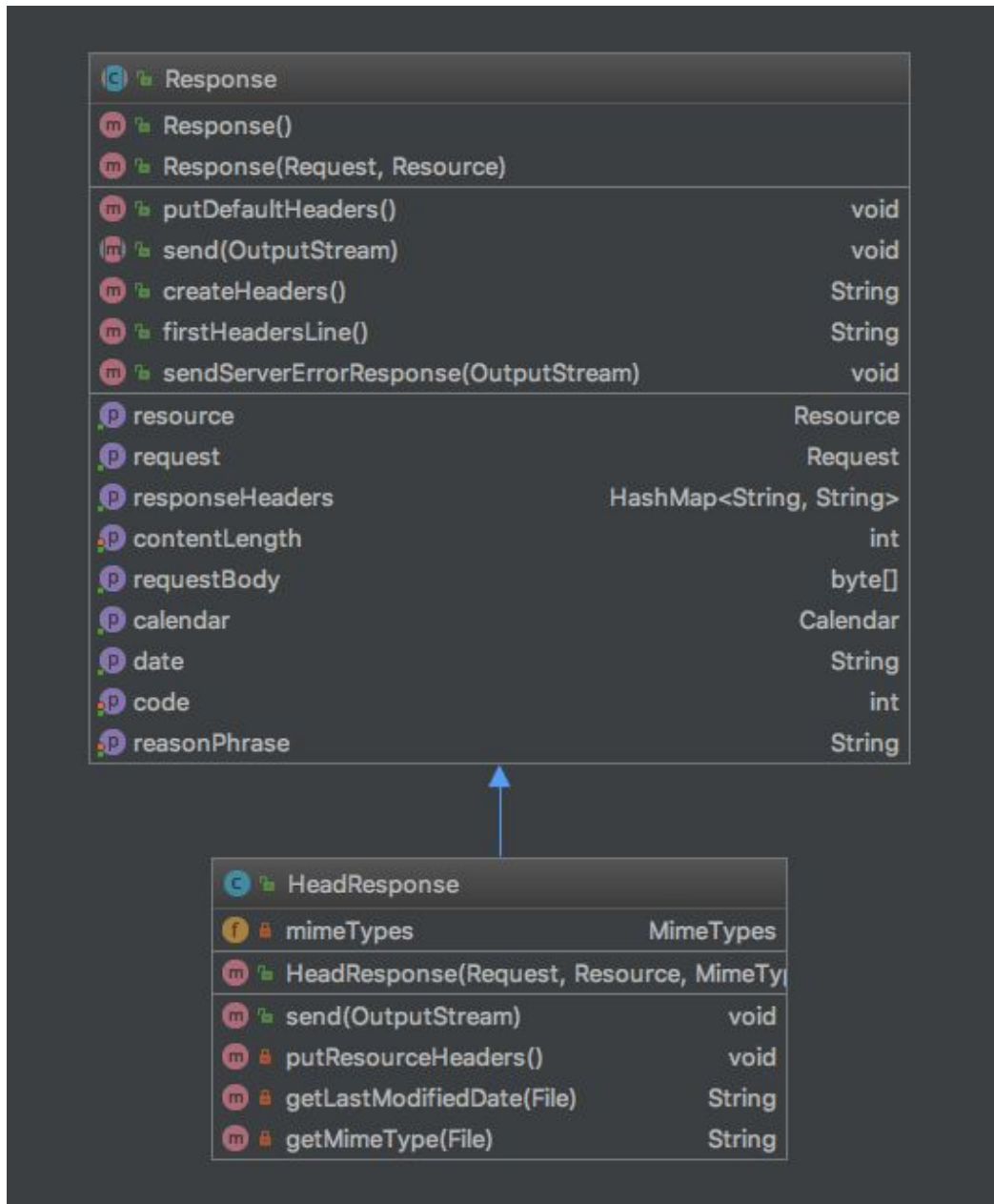
The Htaccess class would extend the configuration reader to load in a file if a certain directory tree is protected by a certain file read. This class is needed to signify a return response of 401 and 403 to ask the user for information and authentication information and then if the provided information is incorrect then give a forbidden to the browser.



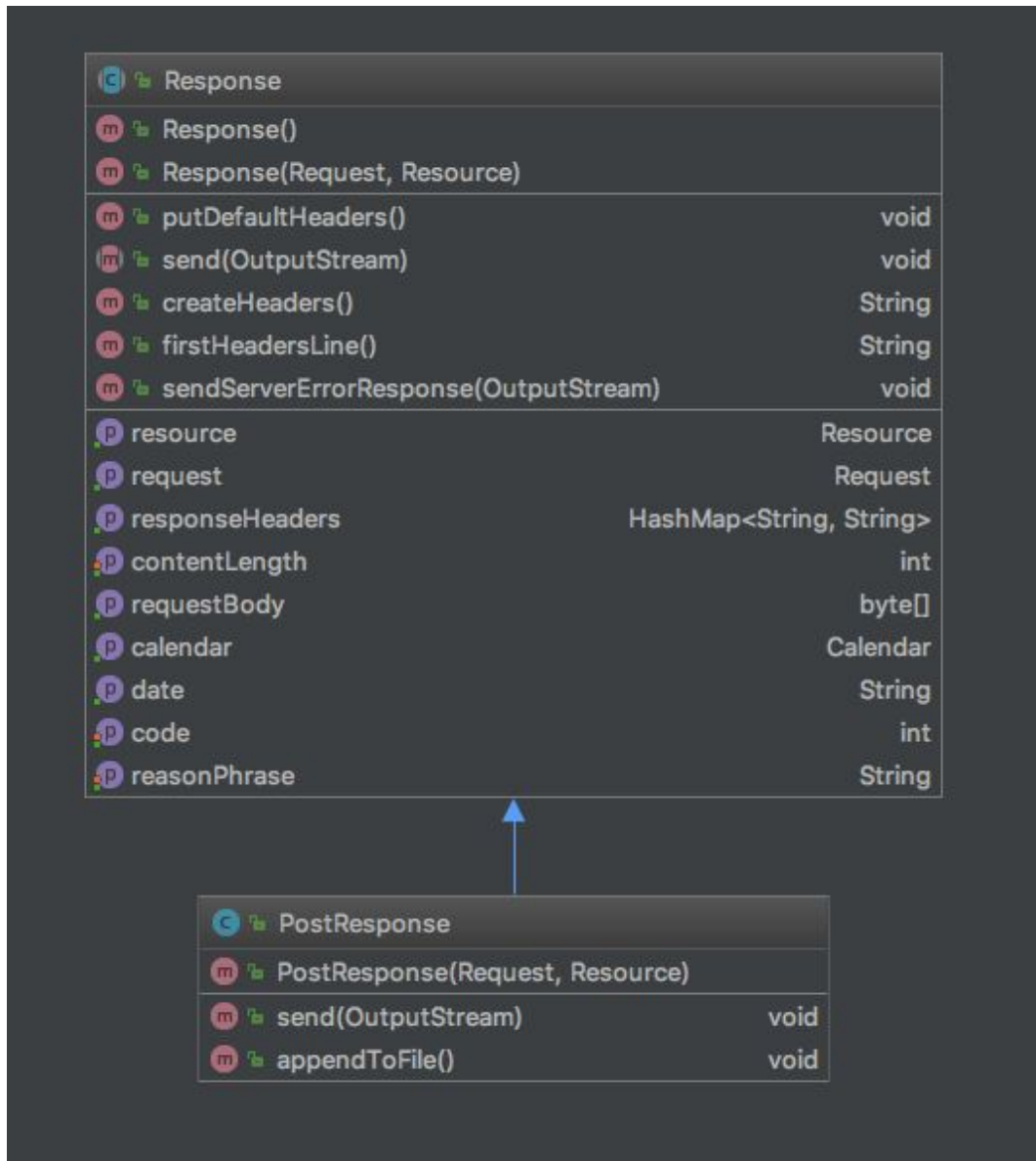
The Htpassword class would also extend configuration reader but is an attribute inside of an Htaccess. This would provide a way to check the header returned of “Authorization” after a 401 response is sent to the client’s browser. The information would then be parsed and compared to the file that was loaded in that holds the username and encrypted password. If the information provided does not match the ones inside the loaded hashmap a 403 response would be send back.

	Logger	
	logFile	File
	fileWriter	FileWriter
	bufferedWriter	BufferedWriter
	Logger(String)	
	write(Request, Response, String)	void
	getLogMessage(Request, Response, Stri	
	formatDate(Calendar)	String
	getUser(Request)	String
	getContentLength(Response)	String

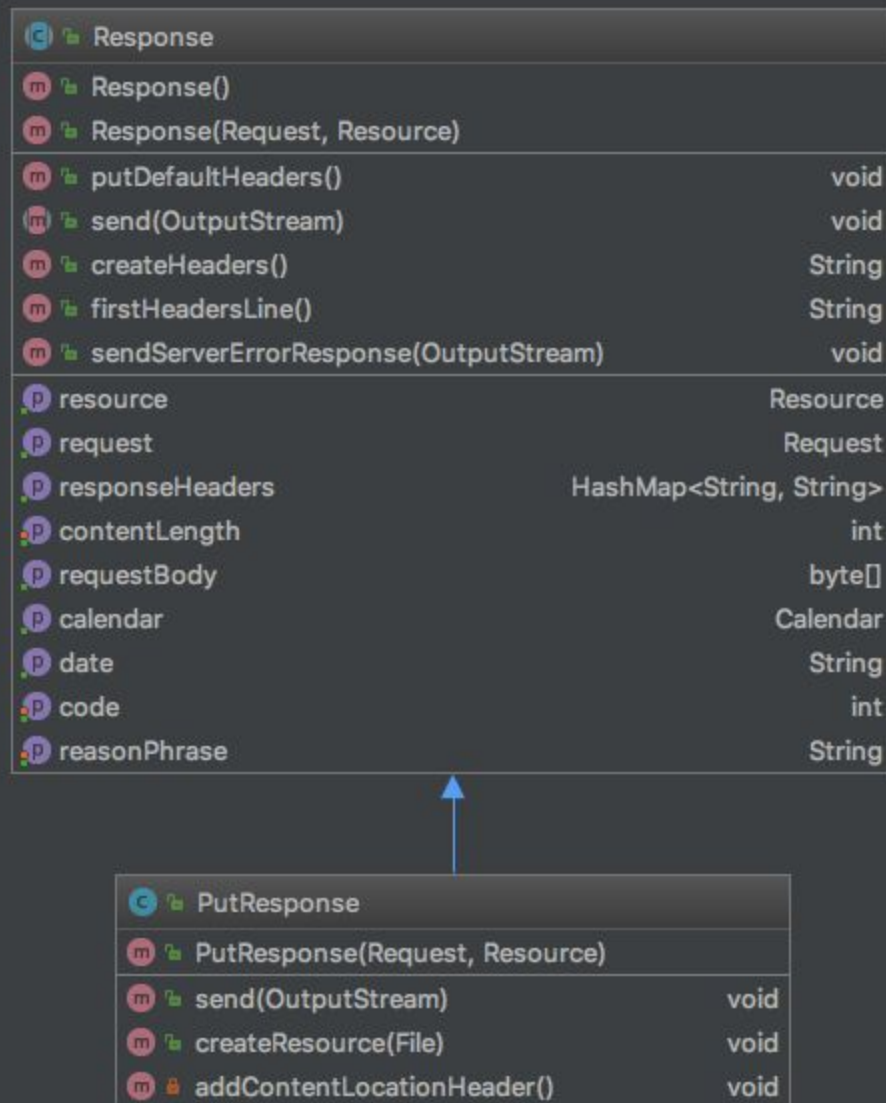
The Logger class creates and writes logs of server activity. When a response is successfully sent to the client, the Logger writes to a text file in common log format, saving the date, request header, and more. If a log file is not present in the server directory, one is created to write to.



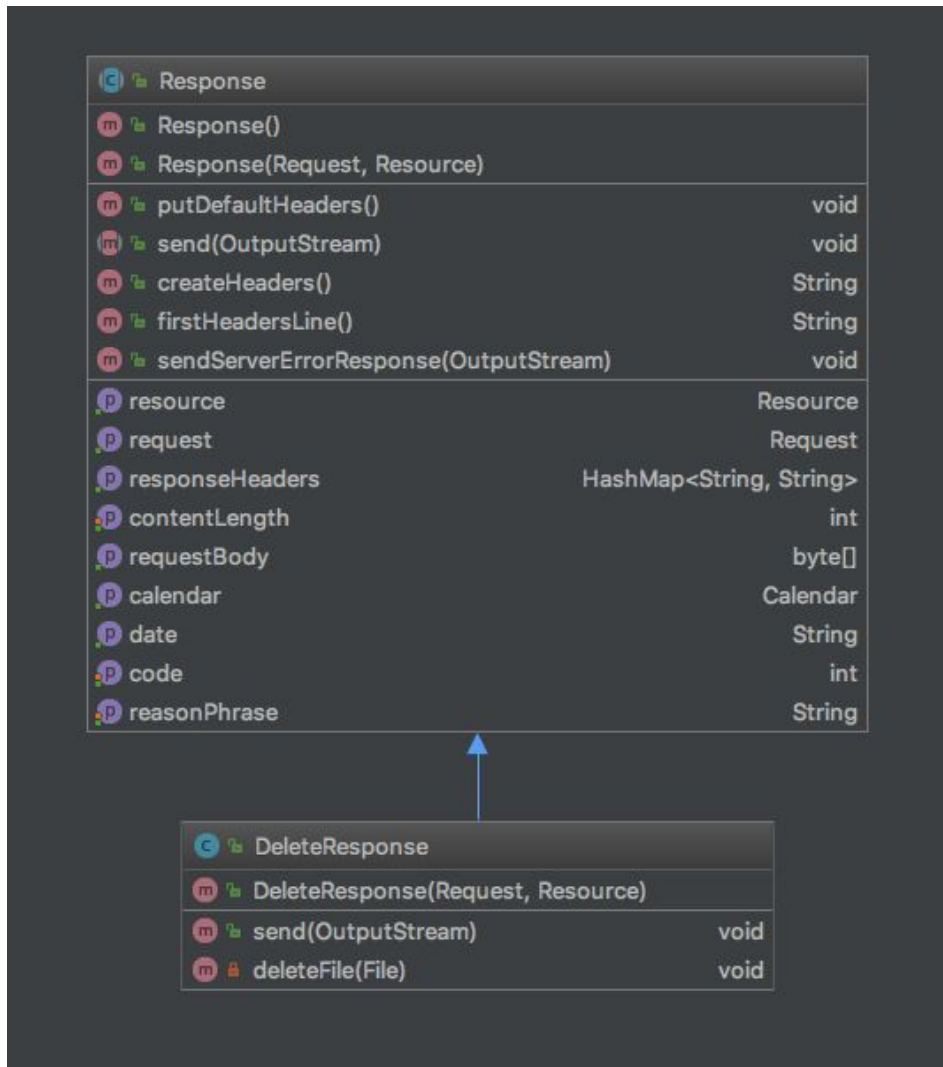
The `HeadResponse` extends the `Response` class which holds the general things responses would have like the code and the reason phrase and default headers. The extended head response would have mime types to have the headers sent back for the possible files sent by a get request.



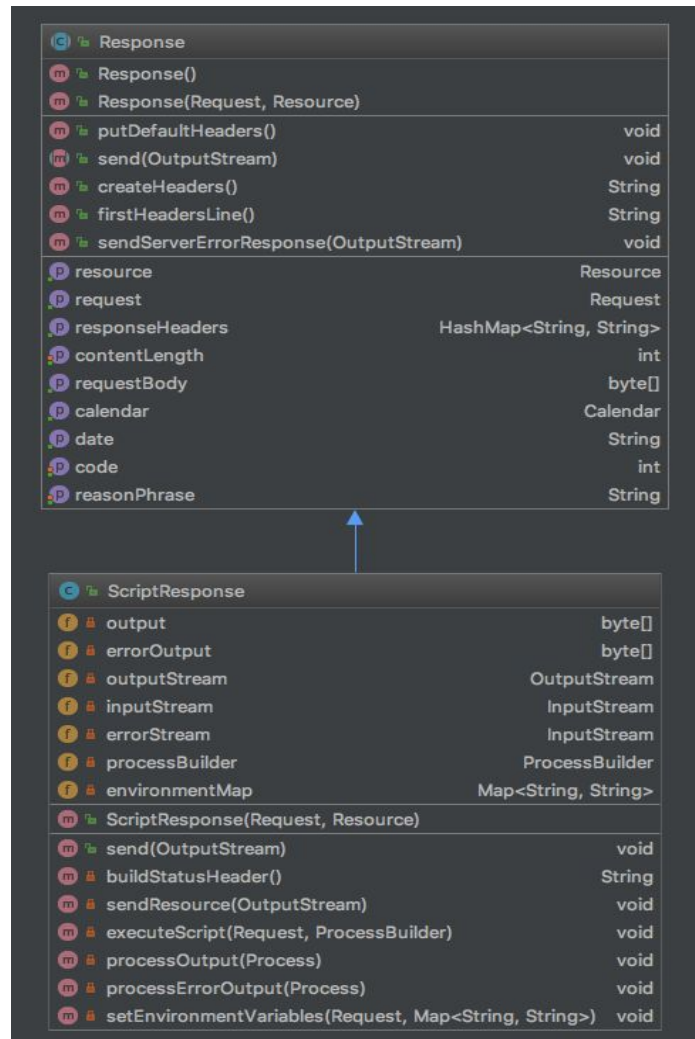
The Post Response class is responsible for handling a post request. The post response would append to the file given and then the response is sent back notifying that a change was made to the file.



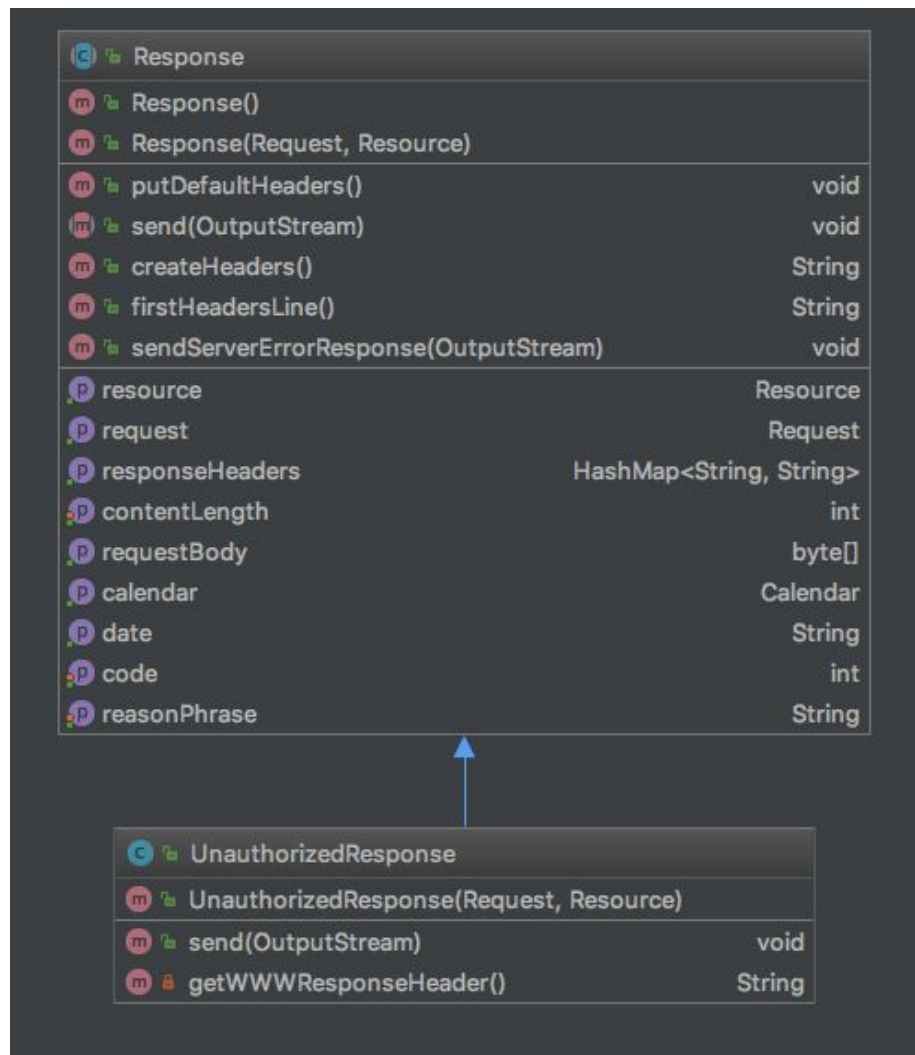
The Put Response handles a client's PUT request. This class will check if the resource exists on the server. If the file exists, then it is overwritten with the contents that are in the body of the request. If the file does not exist, then a new file is created with the same name as the one given by the request URI. A response with status code 200 and reason phrase "OK" is sent back if the file is overwritten, but a response with status code 201 and reason phrase "Created" is sent if the file did not exist before the request was made.



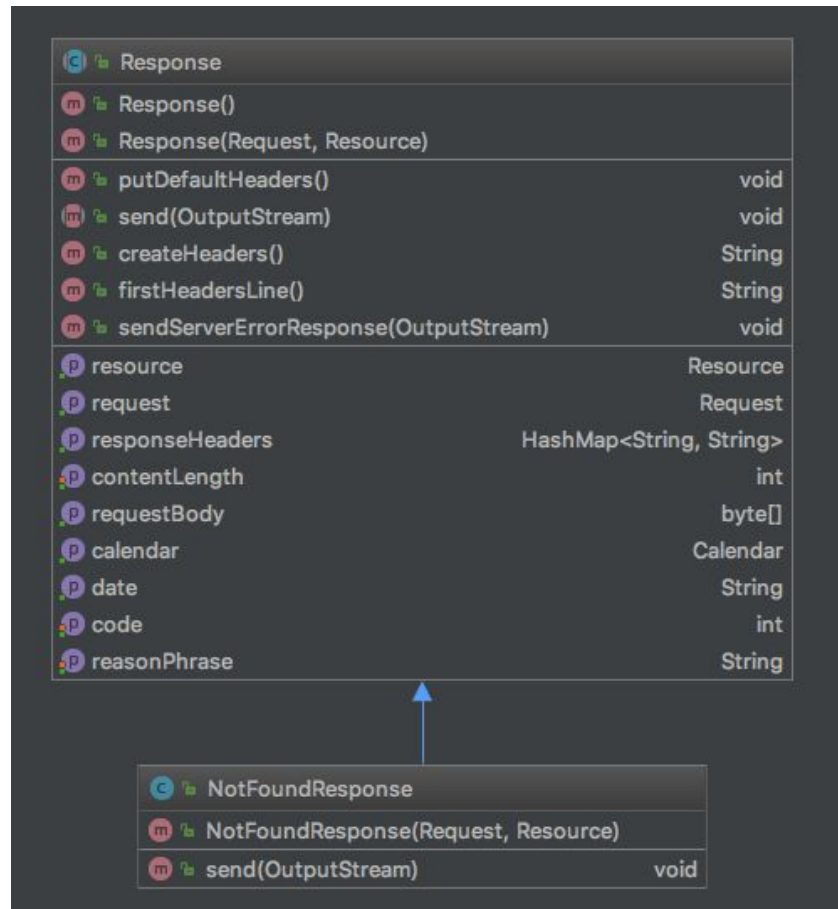
Delete Response class is created when a delete request is sent to the server. It checks if the file exists on the server before deleting the file. Once the file is deleted, a response is sent back to the client with status code 204 with reason phrase "No Content".



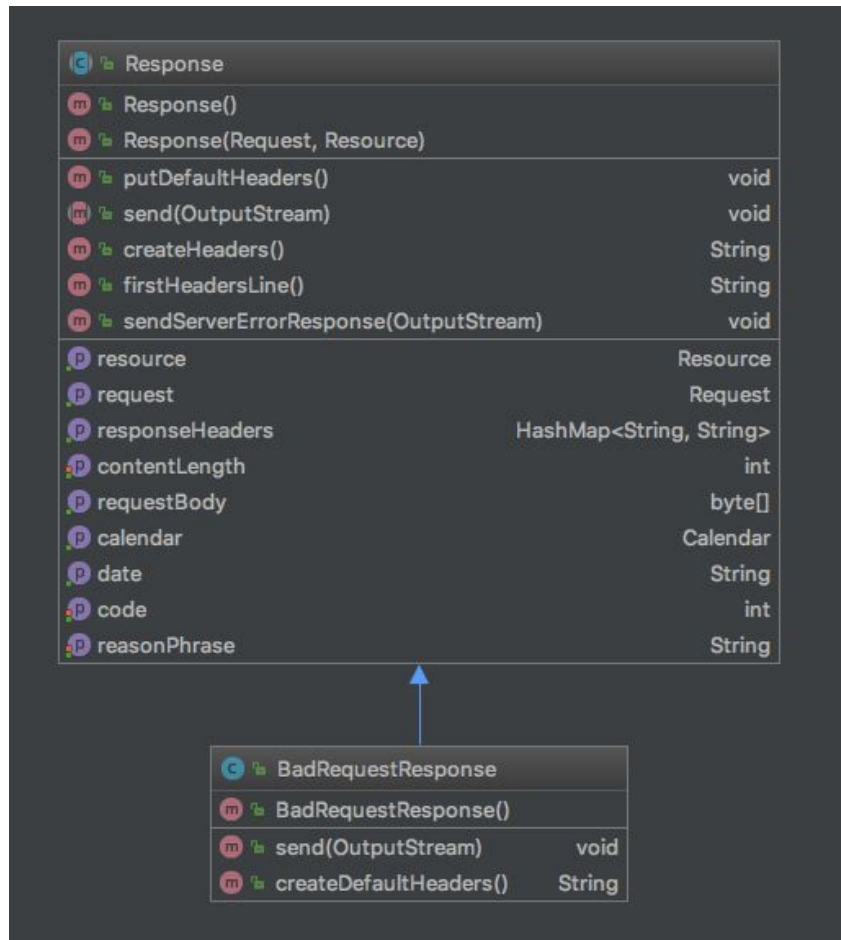
A Script Response is created when the server knows that the file requested is a script in the directory. This class creates a process builder at the given directory, and puts environment variables like “HTTP_VERSION”, “QUERY_STRING”, and “HTTP_”, into the script before executing it. After using a process object to execute the script, the information is pulled through the processes inputstream and error stream. A simple response is then made with basic http headers and the output of the script is appended to the body of the response. The response is then sent back to the client.



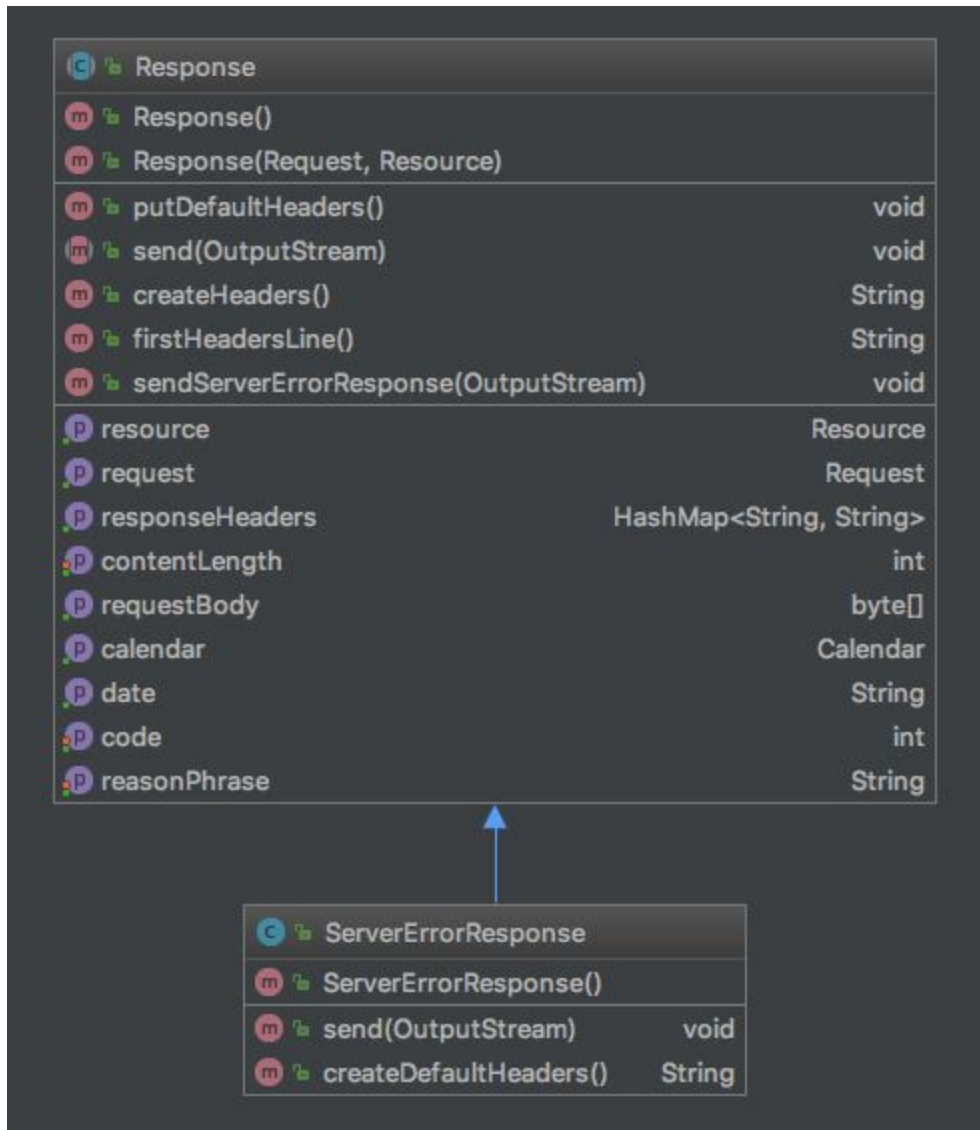
The Unauthorized response is a response that is sent when the client has not given credentials to a secured file. It prompts the client for a username and password, and checks these credentials with the `htaccess` and `htpassword` class. If the given username and password are in these files, then the request is further processed.



The Not Found Response is a basic response that sends with a status code 404 with reason phrase “Not Found”. It is sent back to the client if the file in the URI of the request does not exist on the server. The client would have to pass authentication before receiving this response in the case that the file is not there.



The Bad Request Response is sent if the Request class is unable to parse the client's request. It sends a response with status code 400 to let the client know it sent a request that the server was unable to handle. The Bad Request Response extends Response because it is a type of response. It must send itself back to the client in the case of a bad http request.



The Server Error Response class is used as a catch all in the case that no other response was sent. This fulfills the 500 status code, meaning that the server had an error in processing the request.

Implementation Decisions

The major implementation decision was how the responses was done. Each response would hold and perform their actions and change codes and reason phrase depending on the action performed from the verb. So a get response would check if there was if modified since header and if its present check the date then change its reason phrase to modified and return a 304 rather than a 300 and ok. If not it would then send the file and a 200 response instead. This is done so it would remove the responsibility from the factory onto the specific responses.

Code Organization

The classes are organized very similar to that of the UML diagram provided by Jrob. There wasn't much deviation besides the different specific responses that encompass a specific action and changes reason phrase and response codes depending on what was performed. Responses was an abstract class and would be extended for other responses. Similarly the configuration reader would be extended for other files that need to read and load in dependencies.

Results and Conclusions

After completing this project, we learned exactly how a web server works, and how to make one ourselves. This was one of the largest projects that we have worked on. Since we knew so little coming in, we had to teach each other certain about aspects of the server to save time. The most difficult parts of the project included understanding what each class is supposed to do, and how to test each feature along the way.

There were also quite a few times in which we ran into bugs that were quite difficult to recreate and debug. One the the most interesting realizations we made after reflecting on the project was that the bugs that took the longest to fix were incredibly small. For example, we had a bug in which the index.html file would successfully load on firefox, but not on chrome. After a day or two of debugging, we found out that this was due to the lack of a carriage return line feed between our response headers and response body. Just a lot of time looking into the problem and thinking about the problem at hand and talking to others about problems encountered helped speed up and give solutions to problems encountered.

Because of the web server project that was worked on and completed we learned about the workings and functions of a web server and how files are send and changed. This would help us understand how the web works and if we do become web developers give us the knowledge to be a more effective web developer in the long run.

Testing our Server

We used Chrome and Postman to test our web server. After executing the WebServer.java class, we would use chrome to send a get request to the server by typing in localhost:8080 into the search bar. A 401 response pops onto chrome, this is how we test for authentication. If the authentication fails it would send a 403 response. Once the index pages loads, we use the buttons on the web page to test for other features. We clicked on the thread test button and made sure that threads were not completing in chronological order and load the very big image. We also clicked on the perl script link to test our script response. Any link that leads

to a missing file would return a 404 response. Postman was used to test the rest of the verbs that we needed to implement.

The PUT verb was tested by sending a PUT request with a body and filename through postman. If a file with the contents of the body shows up in the server directory, then we know that the PUT was successful. If any changes are performed and saved and the very same put request was performed the file should be replaced with the body in the response. A HEAD request is successfully handled if a 200 is given back as a response and if the header returned a if modified since compared to the file modified date is before then a 304 would be sent saying that the file has not changed and you can get the file version that was cached. A DELETE request is successful if the file in the URI is properly deleted from the path. We tested a POST request by writing to an existing file through postman and the body in the request is then appended to the specified file.