

**Computer Science Department
San Francisco State University
CSC667
Spring 2022**

Web Server Project

<https://github.com/sfsu-csc-667-spring-2022-roberts/web-server-bug-busters/tree/main>

Joseph Kois
Brian Adams

I. Description of Architecture

When designing the architecture for this server project we wanted a very modular system that was easy to read, and easy to extend. We felt that we could accomplish this by limiting the responsibility of every class, encapsulating their functionality, writing clear interfaces that help reduce coupling, and atomizing each action between the request and response. We feel like we largely accomplished the goals we set.

The largest class in the project is around 100 lines of code, with the average being much less than that. This suggests to us that the majority of the classes are cohesive and focused on solving individual issues. Every action is completely independent and encapsulated in an action object, which allows us to compose them like middleware. This really helps reduce the presence of long difficult to read if else statements. The majority of state that needs to cross public class boundaries and is not owned by the request or response (logging information, action resources and payloads etc) use interfaces. This helps decouple the actions that run in between the request and response.

When planning this project out we separated the server into 3 distinct parts, the request, actions that occur in between the request and response, and the response. The request and response simply serve as vehicles to receive and hold state that comes from or is sent to the client, while the actions in between represent things the server does. We immediately realized that if we were not careful designing the actions in between the request and response that we could introduce a lot of unnecessary complexity and a wall of conditional statements that would be incredibly difficult to maintain. To avoid this we wanted to build a “middleware” system in which actions are composable, atomic and represent a finite state.

To achieve this middleware system we wrote the implementation for each action completely disjointed from the rest of the server. For example, we were not writing classes to read a file based upon request headers, we were simply writing a class that would read a file at a given path. There would be no coupling of headers or understanding of the greater server for each action. After each action implementation was complete, we then wrote a ServerAction that would encapsulate each disjoint action implementation. This Action class was simply responsible for relaying the input from the server, file paths, auth token etc to the implementation. Then determining if the implementation of the Action was successful. The actions status is represented by its status field and includes things like File not found, unauthorized, success etc.

These actions could then be composed into more complex composed actions. If we needed to authenticate a user, then get the file requested we could simply compose

multiple actions together, one after another with the state of each action easily accessible. This substantially reduced the complexity of responding to requests because our server no longer cared about the implementation of each action, simply that an action object existed in its logical path to a response, and that each action object would have a state that determines its next logical path. This “pseudo state machine” approach eliminated the need to trace through complex states. The server can simply look at the state of its current action and either return a status response, or continue on to the next action where it simply looks at that action state.

This not only reduces the complexity but really allows this server to be easily extendable. If we wanted to support more functionality, we would simply need to write an encapsulated action, then either include it in an existing path of composed actions, or write an entirely new action composer, if we wanted an entirely different chain of actions to execute given a specific request. This type of architecture is also incredibly easy to test. Each action requires a minimal amount of data to mock, and functions completely independently. If we wanted to test the auth action or the get file action, we don't need to mock an entire server, we can simply import the action and provide a token or path, then assert a specific state.

Finally the server decides which ActionComposition to execute by using a simple factory pattern. The ComposedActionFactory simply uses the data from the request object to determine which ActionComposition to execute. For example if the request object is requesting a script the factory will execute the ComposedScriptRun action composition. This again makes testing and extending incredibly easy. Keeping each logical path separated and encapsulated rescues the potential complexity and increases the readability, and maintainability of the code base.

The request and response objects shared some of the same architecture. Each request and response object is composed of several smaller objects. The request object though is much simpler in that it is not abstract and does not have any derived classes. It is simply an object that accepts an input stream, and then delegates the responsibility to parse the headers, body and resource path to other classes.

The header is probably the class we are least happy with, simply because there is a lot of state in the root class when the state should probably be split into each derived class as needed. In either case the header class is abstract, and requires the implementation of the generate headers method in its derived classes. This allows requests and responses to generate different headers, while providing the same api. We kept each header in its own string field instead of using a hashmap simply because then we can control which headers are supported to better adhere to any given specification.

To consume the headers we would collect every relevant field determined by the generateHeader methods, then parse out every null field. This allowed the request and response to determine which headers could be sent and received, while providing a very convenient way to generate them from various fields.

The body object simply consumes the rest of the input stream left over from the headers and stores that in a content field. To read the headers and the body, we created our own reader that would read the bytes individually (while this is much less efficient than buffering we ran into several issues with buffers reading past the headers). In either case we have a method that simulates a read line, by reading bytes until it reaches a line return, then casting those bytes into a string and returning that string, and a method that consumes n number of bytes.

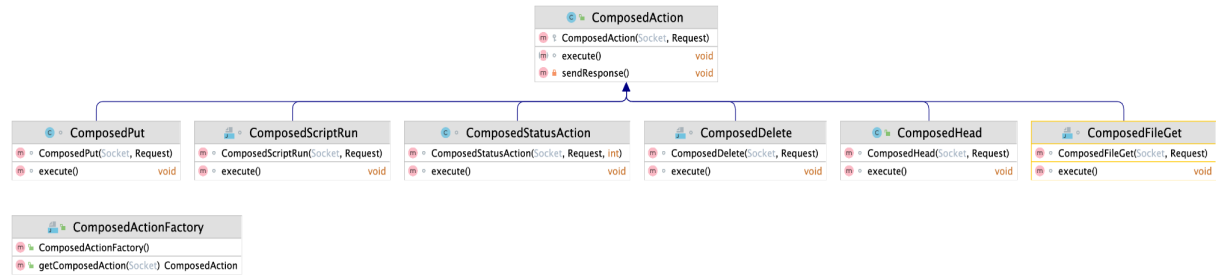
The resource class handles parsing the path provided in the url. The resource class identifies the type of resource requested and modifies the path accordingly. This class is owned by the Request class which provides the url, and can consume the produced resource path.

This server had several config files, each config was read and parsed by a static class that made the configuration available globally. While this certainly introduces tight coupling, we felt that coupling to a configuration object was reasonable. These classes also provide fallbacks with default values in case the configuration was not provided.

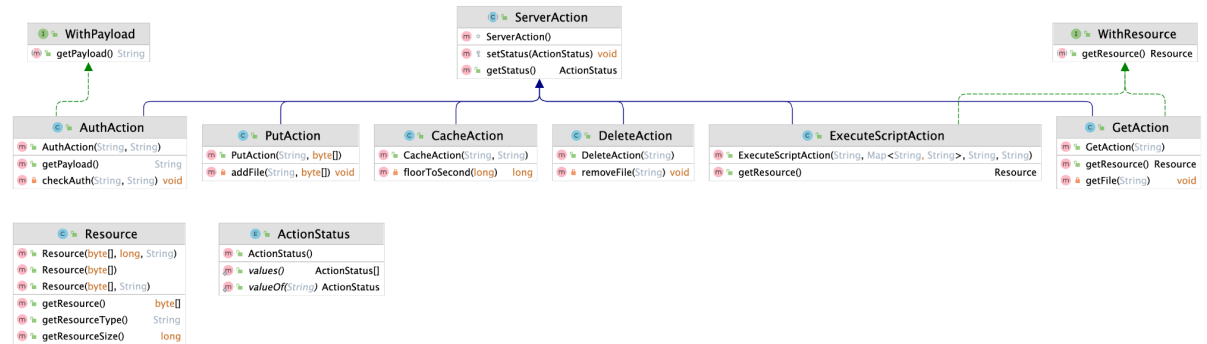
Authentication was handled with composition. We separated out the detection of .htaccess files, the parsing of the .htaccess files and UserAuth files, and the actual authentication between different classes composed into the Authentication class. This provided an organized way to handle the various steps of authentication.

We broke the response up into 3 different types of responses, a status response where just a status code is sent, a file response where a body is sent with headers, and a script response where the scripts response extends the existing headers and provides a body. Each of these responses are derived from an abstract super class. This super class provides apis for adding headers, attaching headers to the body etc.

A. ActionController



B. Actions



C. Authentication

Authorization		
Authorization(String)		
debugTestIsAuth(String, String)	boolean	
getAuthName()	String	
isUserAuthorized(String)	boolean	
debugStripedTestIsAuth(String, String)	boolean	
getAuthType()	String	
getAccessFileFromResourcePath(String)	File	
requiresAuth(String)	boolean	

UserAuth		
UserAuth(String)		
debugStripedTest(String, String)	boolean	
debugPrint()	void	
encryptClearPassword(String)	String	
verifyUser(String, String)	boolean	
isAuthorized(String)	boolean	
debugTestIsAuth(String, String)	boolean	

AccessFile		
AccessFile(String)		
getRequire()	String	
getAuthUserFile()	String	
debugPrint()	void	
getAuthName()	String	
getAuthType()	String	

D. CGI

ProcessRunner		
m	ProcessRunner(String, Map<String, String>, String, String)	
m	readOut()	byte[]
m	end()	void
m	formatEnvironmentVariables(Map<String, String>, String, String)	Map<String, String>
m	getErrorStream()	InputStream
m	debugPrintOut()	void
m	writeln(byte[])	void

E. Config











ServerConfig		
m	ServerConfig()	
m	debugPrint()	void
m	getDocumentRoot()	String
m	getAccessFileName()	String
m	getDirectoryIndex()	String
m	getServerRoot()	String
m	getListen()	int
m	assignDefaultValues()	void
m	getScriptAlias()	ArrayList<Alias>
m	getAlias()	ArrayList<Alias>
m	getLogFile()	String

MimeTypes		
m	MimeTypes()	
m	debugPrint()	void
m	getMimeTypeFromFile(String)	String

Alias		
m	Alias(String, String)	
m	getAbsolutePath()	String
m	getSymbolicPath()	String

F. Logger

LoggableRequest	
getTimeOfRequest()	String
getIdentity()	String
getProtocol()	String
getUserId()	String
getHost()	String
getPath()	String
getMethodVerb()	String

Logger		
  logFile		File
  Logger()		
  writeLog(LoggableRequest, LoggableResponse)		void
  generateLog(LoggableRequest, LoggableResponse)		String
  wrapLog(String)		String

```
LoggableResponse
  (m) getStatusCode() String
  (m)  getReasonPhrase() String
  (m)  getSizeOfResponse() long
```

G. Messages

1. Headers

Headers	
🔍	Headers()
🔍	getHeadersWithEnd()
🔍	getReasonPhrase()
🔍	getHeadersWithoutEnd()
🔍	toStringMap()
🔍	getAuthorization()
🔍	generateHeaders()
🔍	getPath()
🔍	setContentType(String)
🔍	setCharset(String)
🔍	getStatusCode()
🔍	setStatus(int, String)
🔍	getProtocol()
🔍	getContentType()
🔍	setWWWAuthenticate(String)
🔍	formatHeader(String)
🔍	debugPrint()
🔍	getHost()
🔍	getIfModifiedSince()
🔍	setContentLength(long)
🔍	getContentLength()
🔍	getMethodVerb()

GivesHeaders

getHeaders() `Map<String, String>`

```

1  RequestHeaders
2  RequestHeaders(ByteStreamReader)
3  toStringMap() Map<String, String>
4  generateHeaders() String

```

```

ResponseHeaders
ResponseHeaders()
generateHeaders() String
toStringMap() Map<String, String>

```

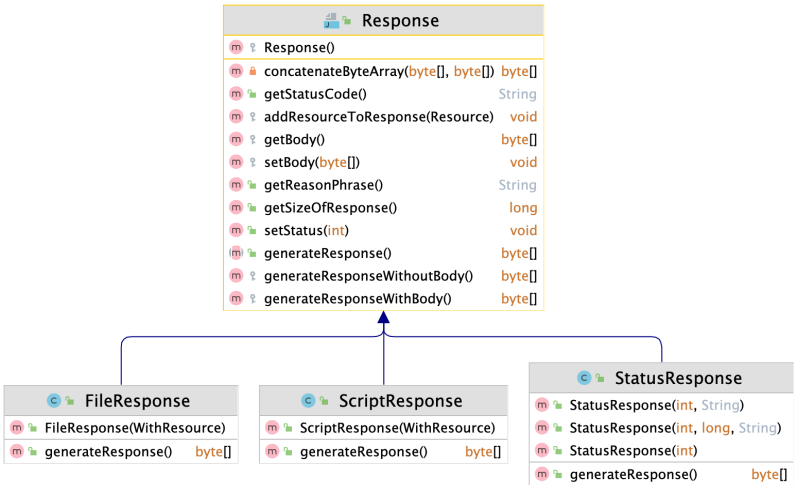
2. Request

Request	
Request(InputStream)	
getIfModifiedSince()	String
getIsScript()	boolean
getAuthHeader()	String
getTimeOfRequest()	String
getUserId()	String
debugPrint()	void
getResourcePath()	String
getMethodVerb()	String
getIdentity()	String
getPath()	String
getProtocol()	String
getContentType()	String
getRawQueryString()	String
getHost()	String
getBody()	byte[]
getHeaders()	Map<String, String>
getQueryParams()	Map<String, String>

Resource	
Resource(String)	
debugPrint()	void
getPathToResource()	String
parsePathToResource(String)	void
isInAlias(String)	Alias?
getIsScript()	boolean
getQueryParameters()	HashMap<String, String>
formatNonAlias(String)	String
getRawQueryString()	String
parseParams(String)	void

ByteStreamReader	
ByteStreamReader(InputStream)	
getBytes(int)	byte[]
getLine()	String
changeCharSet(Charset)	void

3. Response



II. Problems During Implementation

1. Issue number 1

Problem: The first issue we ran into was the buffered input reader reading past the headers and into the body. Our initial implementation for parsing the headers used a buffered input reader, which was incredibly convenient because we were able to read lines from the buffer. Unfortunately, the buffer read past the end of the header and upon researching the issue we found some compelling arguments suggesting that using a reader to dump bytes is a bad practice, (with that said, we think casting from byte to char to byte in Java could certainly be considered technically wrong, we cannot imagine a situation in which it would cause an actual error assuming the encoding is known).

Solution: We decided to not use a reader and build our own “reader” class. Our `ByteStreamReader` simply wrapped a `DataInputStream`, offering a more convenient way to work with bytes. One method simulated a read line by collecting bytes in an array list, looking for a byte that represented a new line character, then casting to a string and returning it, and another simply exposed the read n bytes method from the `DataInputStream`. Since writing this solution we have learned more about input and output streams in Java and we think that this solution might have been a little over engineered.

2. Issue number 2

Problem: The second issue we had was the execution of scripts without a qualifier like `python test.py`, or `perl -T perl_env`. We spent a fair amount of time trying out different configurations and file permissions in order to resolve this problem.

Solution: After looking into this issue and testing on various operating systems we have determined that this is most likely an issue with our permissions. We cannot say that with complete certainty but after running the server on Brian’s Manjaro laptop, executing shell scripts from the server that were located in the `bin` directory without any qualifiers, we can confidently say that the server will execute scripts, but only if they have specific permissions and are located within the system path. This very specific behavior suggests that if the server is provided with a file that can be treated like an executable, by whatever means it takes the owner to do so, it will execute that file.

3. Issue number 3

Problem: The final issue we struggled with was getting clarity on very specific topics. The specification for HTTP 1.1 is large, and while the documentation provided a really great description highlighting exactly which parts we needed to implement, we were still not entirely clear which subset of the implementation we needed to implement. We were unsure which of the conditional headers we needed to support, how the Authorization header would look if we were just supporting one Auth type, if the file extensions were going to be provided in the put request etc.

Solution: As we worked on various parts of the project, we kept a word document open of various things we were unclear about or thought we might need clarity on to proceed. We fortunately started this project as early as possible so we were prepared when the instructor offered class time to answer questions. We tried to maximize this time and wrote down the answers to the questions we had asked so we could then quickly reference them, as well as documented the questions and answers to some other questions students had asked that we had not even considered ourselves. We found that both of these question and answer sections were incredibly helpful and just about covered every question we had.

III. Discussion of What Was Difficult

We find that being able to accurately anticipate how long specific parts of projects will take and to feel comfortable in our assessment is one of the most difficult parts of programming, and this project was no exception. Knowing when you need to slow down and spend some time either rethinking an implementation before it becomes much harder to do so, or even familiarizing yourself with concepts that you might not have a complete understanding of is incredibly important when writing maintainable code. We found that we were inclined to move quicker—even when we should not, when we do not feel confident in our time budgeting estimations. This can sometimes lead to less than optimal decisions, for example the request header using a combination of explicitly defined fields, then later realizing that the scripts might need some headers that were not included in the fields we defined, so we dumped the rest of request headers in a map. Had we moved slower, we would have probably been able to anticipate that it would not be reasonable for us to define every field we would need to consume for the request headers.

We found the concept of I/O streams initially difficult to work with. We were very used to receiving data in a way that is abstracted away from streams, an ajax request returns a complete response, a db query returns the requested fields etc. The idea of the data not coming in as an already composed object, but instead we must pull the data out byte by byte or line by line feels like it is substantially more prone to error. We think, conceptually, this is pretty easy to reason about. But the complexity comes from managing potential issues like deadlocks, where your server is waiting to read more, but the client has finished sending the request. Checking for EOF, or empty strings is certainly a straightforward solution, but we cannot help but feel as if there are unchecked edge cases where malformed requests might potentially lock the server in an unexpected way.

Trying to identify when the appropriate time to implement specific design patterns was something we found to be difficult. It's incredibly easy to get carried away and implement “best practice” design patterns everywhere just because it's “the best practice”. In our experience, this will often lead to overly complex code whose reliance on inappropriately applied design patterns

becomes a liability. We wanted to make sure we were not using a singleton or factory pattern just because we didn't have a better idea, but because it was the best solution for the given problem.

Throughout this entire project we wanted to be very careful to not introduce unnecessary complexity. While it might feel great to code golf, or implement elaborate solutions, it generally does not feel great to maintain or debug. We wanted to make sure we were not using terse code or incredibly complex solutions without a clear tangible benefit to the project. Trying to ensure that the code we wrote was simple, maintainable, and extendable was difficult. It required us to take a step back before every commit and consider whether or not we would want to maintain the code we just wrote. This project was not just about writing code that functioned, but writing code that made sense.

IV. Description of Test Plan

The application used for testing was Postman. By using Postman we were able to create separate collections based on the type of testing we wanted to perform. For example, we created separate collections for different aspects of the web server like authentication, delete, get, head, post, put, and miscellaneous tests. Within each subsection, we had numerous tests that will all output to a folder in our web server which contains test details. The tests involved making a request to a specific endpoint, and asserting a specific status code. These tests involved intentionally failing requests (requesting resources that were not available, unauthenticated requests etc), requests that succeeded, and testing edges (if-modified-since headers being sent on the same second last modified, a second before, and a second after etc).

Postman was used as an end to end test, testing out a specific path given a request. Postman also served as a generalized form of regression testing, used as new features were added. Early on we decided to focus on end to end testing instead of unit testing for simplicity and speed. We can guarantee that we have nowhere near complete coverage using postman alone, but we feel like due to the manner in which this application is graded we would rather focus on more complete end to end tests, rather than splitting our tests up including individual unit tests. If this was a project that we intended to scale up further than the subset of HTTP 1.1 that is covered in the documentation, then unit testing would be a hard requirement and something that should have been implemented from the start.

V. Grading Rubric

Category	Description		
Code Quality	Code is clean, well formatted (appropriate white	5	5

	space and indentation)		
	Classes, methods, and variables are meaningfully named (no comments exist to explain functionality - the identifiers serve that purpose)	5	5
	Methods are small and serve a single purpose	3	3
	Code is well organized into a meaningful file structure	2	2 15/ 15
Documentation	A PDF is submitted that contains:	3	3
	Full names of team members	3	3
	A link to github repository	3	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	1
	Brief description of architecture (pictures are handy here, but do not re-submit the pictures I provided)	5	5
	Problems you encountered during implementation, and how you solved them	5	5
	A discussion of what was difficult, and why	5	5
	A thorough description of your test plan (if you can't prove that it works, you shouldn't get 100%)	5	5 30/ 30
Functionality - Server	Starts up and listens on correct port	3	3
	Logs in the common log format to stdout and log file	2	2
	Multithreading	5	5 10/ 10
Functionality - Responses	200	2	2
	201	2	2

	204	2	2
	400	2	2
	401	2	2
	403	2	2
	404	2	2
	500	2	2
	Required headers present (Server, Date)	1	1
	Response specific headers present as needed (ContentLength, Content-Type)	2	2
	Simple caching (HEAD with If-Modified-Since results in 304 with Last-Modified header, Last-Modified header sent)	1	1
	Response body correctly sent	3	3 23/ 23
Functionality - Mime Types	Appropriate mime type returned based on file extension (defaults to text/text if not found in mime.types)	2	2 2/2
Functionality - Config	Correct index file used (defaults to index.html)	1	1
	Correct htaccess file used	1	1
	Correct document root used	1	1
	Aliases working (will be mutually exclusive)	1	1
	Script Aliases working (will be mutually exclusive)	3	3
	Correct port used (defaults to 8080)	1	1
	Correct log file used	1	1 11/ 11

CGI	Correctly executes and responds	4	4
	Receives correct environment variables	3	3
	Connects request body to standard input of cgi process	2	2 9/9

Total: ~~110/100~~
95/100