Jonathan Shalvi

SE450 – Winter 2012

Final Project

## Summary

The goal of this project was to implemented a basic web server with the following features (incomplete features are highlighted in red):

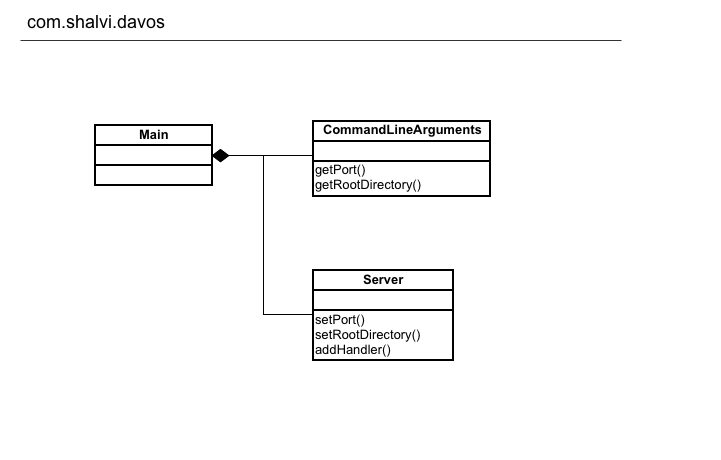
* The server shall listen for requests on a port specified at run time
* The server shall support the following HTTP 1.1 requests: GET, POST, HEAD, and DELETE
  + GET requests will return the desired resource
  + POST requests will return the desired resource, with POST data logged in the console
  + HEAD requests will return headers and be logged to the console
  + DELETE requests will have no effect, but will be logged to the console
* The server shall handle unsupported requests gracefully
* The server shall support plain-text filetypes (html, txt, css, js)
* The server shall support the JPEG filetype
* The server shall support console logging
* The server shall support a generic server configuration mechanism
* The server shall support basic authentication
* The server shall support cgi scripts
* The server shall support cookies
* The server shall support cookie-based sessions

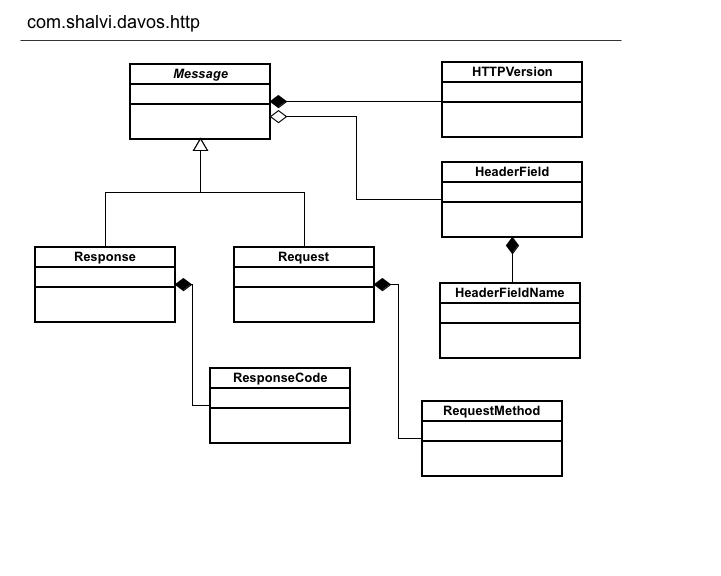
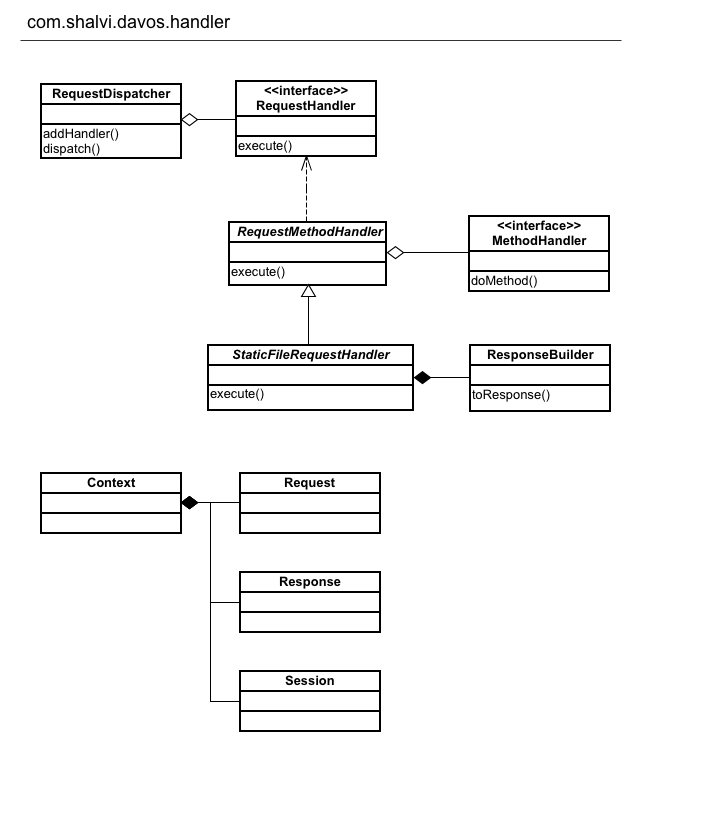
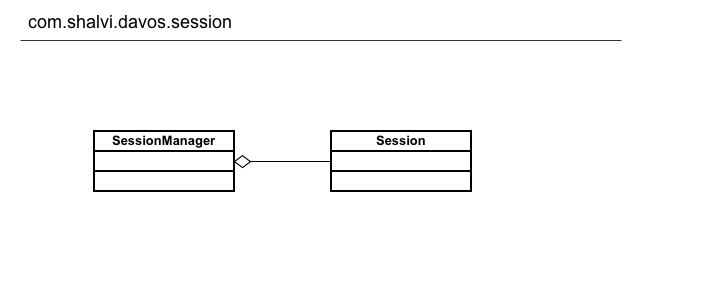
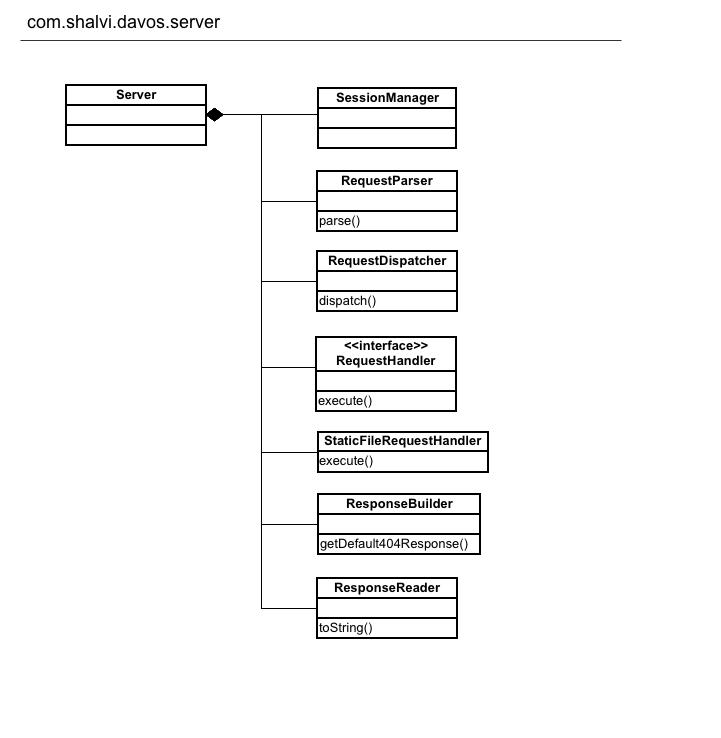
The following features were optional, as time permitted:

* The server shall support configurable URL dispatching to client controllers
* The server shall support client controllers
* The server shall support processing of Mustache templates for views

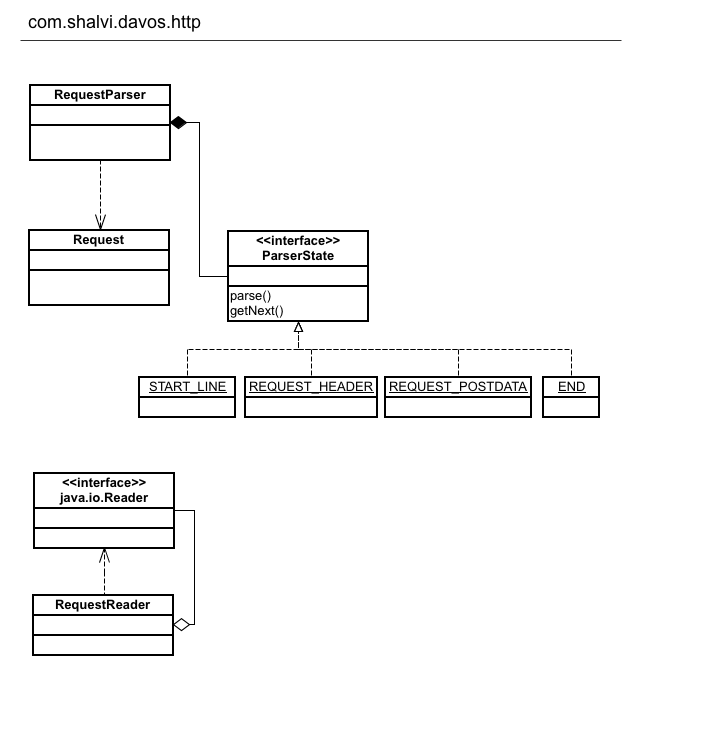
Unfortunately I was unable to address basic authentication, cgi scripts, or server configuration. I’m confident, however, that the server can be easily extended to accommodate these features.

## Class diagrams



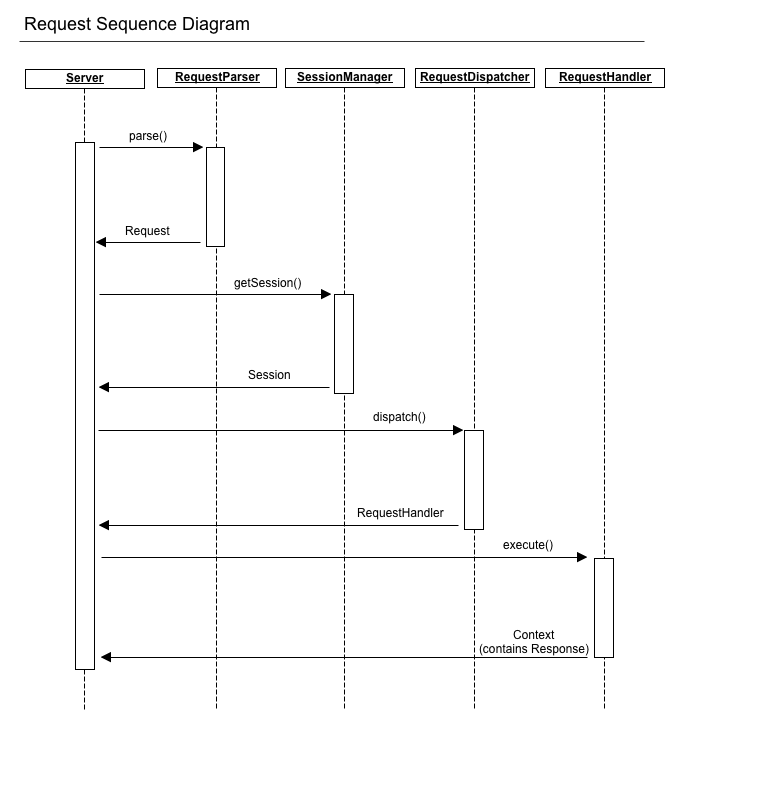


Continued on next page…



## Sequence diagram

The following sequence diagram describes the process of handling a typical HTTP Request:



## Time Summary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Week | 1 | 2 | 3 | 4 | Total |
| Design | 0 | 1 | 2 | 6.5 | 9.5 |
| Code | 3 | 7.5 | 10 | 17 | 37.5 |
| BigBug | 0 | 1.5 | .5 | 4.5 | 6.5 |

## Patterns

### State

The State pattern was used in the RequestParser to assist in the parsing of Requests. An HTTP Request contains three main sections: the start line, the headers, and the body. Each section requires particular validation rules, and are dependent upon eachother in certain cases. For instance, parsing the request body is dependent on the presence of a Content-Length header.

In order to simplify the representation of this process, I created a ParserState enum with the following entries:

* START\_LINE
* REQUEST\_HEADERS
* REQUEST\_POSTDATA
* END

Each enum entry implements a parse() and getNext() method. The parse method reads the request, line-by-line from the RequestReader, until it decides it has completed its section of the request. The parser then requests the next ParserState. The process continues until the END state is reached and the request has been parsed.

### Decorator

A request comes out of a Socket via a Reader, which normally would allow a stream to be read line-by-line. Lines in an HTTP request, however, are delimited by a carriage-return and line-feed (CRLF), whereas a Reader will consider just a carriage-return to be the end of a line. Without a proper reader, the server would simply sit in a loop waiting for the stream to end. This only occurs when the client closes the connection.

To properly read an HTTP request stream, I created a RequestReader which decorates any Reader instance. In practice, the RequestReader decorates a BufferedReader, such that calling readLine() will return successive lines in the request. This allows for much simpler parsing within RequestParser.

### Strategy

Handling a request has two aspects: the URI and the Method. A URI may be a static file, or a dynamically generated response. Likewise, the action and response required by a requested URI can change based on the requested Method. For instance, a GET request for index.html will return that file, whereas a DELETE request for index.html should delete it. In practice, there isn’t much expectation that a server implement more than GET and POST, but it should still respond properly.

To address handling such a variety of actions, I used the Strategy method in two stages. In the first stage, a client programmer will register any number of RequestHandler objects to the server. Each handler is registered by a pattern which is used to match an incoming request’s URI. The handler is handed off to RequestDispatcher which will fetch the proper handler at run-time.

The second stage occurs within an abstract class called RequestMethodHandler. Strictly speaking, a client programmer may implement the RequestHandler interface directly without using RequestMethodHandler. However, this abstract class forces its subclasses to define a method for each RequestMethod (i.e. doPost, doGet, doDelete). At run-time, RequestMethodHandler will search for the correct method handler (defined in a static EnumMap), and will invoke the correct handler. Any methods which are not overridden will invoke a default method handler.

This allows the server to transparently handle requests with no knowledge of the response’s source or method of construction.

### Builders

In order to simplify the construction of objects, while keeping the immutable outside of their packages, I created various builders. Most notable are the Session Manager, which creates new session objects that are automatically registered with a unique id in the session cache. The other notable builder is the ResponseBuilder, which allows an immutable Response object to be created step-by-step within a RequestHandler. The ResponseBuilder also has aspects of the Factory pattern with the getDefault404Response() and getDefault501Response() methods.

## Successes and Failures

The main success of this project is the lessons learned through Test-Driven Development. Nearly each class has an associated unit test, and about 40% of the entire codebase is unit tests. The overall impact is only 12% of my time was spent on big bugs. Personally, this is quite an accomplishment. While development was slower, it resulted in less time spent on bugs and allowed me to more easily estimate upcoming features.

Another benefit of Test-Driven Development was how easily it allowed me to perform significant refactoring’s with very few bugs. Towards the end of the project, I realized I need to complete change the class used by request handlers and the dispatcher. This required changing many classes in many files. I used Eclipse to perform about half of the refactoring, and manually did the rest. To my surprise, all unit tests passed once everything compiled.

Certain parts of the system were quite successful. The use of the State pattern felt like a lot of overhead at first, but it paid off in its flexibility down the line. Initially I was unsure of how a message body was supposed to be parsed, and was unaware of its dependency on the presence of certain headers. Once I understood the standard, however, it was very easy to add another state to the parser.

Other parts of the system were less successful. My initial concept of the request handler did not include the idea of passing a session which could mutate during the execution of the handler. I had to create a Context class which necessitated the refactoring mentioned earlier. Although the refactor was successful, I was unhappy that I didn’t foresee this issue, and that my design wasn’t flexible enough to adapt to the change.

I’m generally unhappy with the Server class. Since it is the integration point of all functionality, it has too many dependencies and not enough tests. Most of my time spent on bugs was spent in the Server class. The CommandLineArguments class was also hastily written, and could have benefited from the Strategy pattern rather than a series of if-else statements.

The method of building responses is very poorly done. My hope was to set up the ResponseReader so the response could be streamed to the socket’s printwriter. I couldn’t think of a way to do this well and resorted to a simple toString() method which essentially puts the entire response into memory.

A related failure is content-handling. I’m currently not setting the Content-Type header, which is required for images and other non-text media and could cause problems in certain browsers. Even worse, I’m determining the Content-Length on every static file request, rather than for types that need it. This is bad because I’m creating a second file reader for each response in order to determine its length. I imagine there is a better way to do this, perhaps by resetting the already existing stream, but I could not find a way given my limited time. This has the effect of putting the entire file’s contents into memory. Between calculating content length, and outputting content via toString(), static files are places into memory twice. My only conciliation is that I noticed many servers include Content-Length even for dynamic content, which is unnecessary.

Overall I feel the structure of the server is sound, and will allow for further expansion and improvement.