**SHUTTLE**

A Basic Web Server

Kyle Nickel

Daman Camara

Matthew Shortall

Zachary Sorenson

University of Maryland University College

Dr. Duchon

CMSC-495

Table of Contents

[1 Overview 6](#_Toc374805053)

[2 Project Plan 7](#_Toc374805054)

[2.1 Introduction 7](#_Toc374805055)

[2.1.1 Purpose 7](#_Toc374805056)

[2.1.2 Scope 7](#_Toc374805057)

[2.2 Definitions 7](#_Toc374805058)

[2.3 Skills Required 7](#_Toc374805059)

[2.4 Environment 8](#_Toc374805060)

[2.5 Milestones 8](#_Toc374805061)

[2.5.1 Overview and Assignments 8](#_Toc374805062)

[2.5.2 Daemon (WebServer Class) 9](#_Toc374805063)

[2.5.3 Logger 9](#_Toc374805064)

[2.5.4 Cacher 10](#_Toc374805065)

[2.5.5 RequestHandler 10](#_Toc374805066)

[2.6 Testing 10](#_Toc374805067)

[2.6.1 Unit Testing 10](#_Toc374805068)

[2.6.2 Application Testing 10](#_Toc374805069)

[2.6.3 Test Site 11](#_Toc374805070)

[3 Requirements Specification 12](#_Toc374805071)

[3.1 Requirements Overview 12](#_Toc374805072)

[3.2 Daemon (WebServer Class) Requirements 12](#_Toc374805073)

[3.2.1 Initialization 12](#_Toc374805074)

[3.2.2 Operational Until Exited 12](#_Toc374805075)

[3.3 RequestHandler Requirements 12](#_Toc374805076)

[3.3.1 Returns Page to User 12](#_Toc374805077)

[3.3.2 Queries the Cacher 13](#_Toc374805078)

[3.3.3 Reads Files on Cacher Fault 13](#_Toc374805079)

[3.3.4 Provides Files to Cacher on Successful Fault 13](#_Toc374805080)

[3.4 Cacher Requirements 13](#_Toc374805081)

[3.4.1 Returns Pages From Cache 13](#_Toc374805082)

[3.4.2 Returns a Fault 14](#_Toc374805083)

[3.4.3 Utilizes a Least-Recently-Used Algorithm 14](#_Toc374805084)

[3.5 Logger Requirements 14](#_Toc374805085)

[3.5.1 Logs to Appropriate File 14](#_Toc374805086)

[3.5.2 Logs Debug Statements 14](#_Toc374805087)

[3.5.3 Suppress Debug Statements 15](#_Toc374805088)

[4 User’s Guide 16](#_Toc374805089)

[4.1 Introduction 16](#_Toc374805090)

[4.1.1 Purpose 16](#_Toc374805091)

[4.1.2 Scope 16](#_Toc374805092)

[4.2 Starting the Daemon (WebServer Class) 16](#_Toc374805093)

[4.3 Command line options 16](#_Toc374805094)

[4.4 Configuration file 16](#_Toc374805095)

[5 Test Plan 18](#_Toc374805096)

[5.1 Introduction 18](#_Toc374805097)

[5.1.1 Purpose 18](#_Toc374805098)

[5.1.2 Scope 18](#_Toc374805099)

[5.2 Application testing 18](#_Toc374805100)

[5.2.1 Manual testing 18](#_Toc374805101)

[5.2.2 Automatic testing 19](#_Toc374805102)

[5.3 Daemon (WebServer Class) testing 20](#_Toc374805103)

[5.4 RequestHandler testing 20](#_Toc374805104)

[5.4.1 Test Case 1 – Returns Page to User 20](#_Toc374805105)

[5.4.2 Test Case 2 – Querying the Cacher 20](#_Toc374805106)

[5.4.3 Test Case 3 – Read Requested File when Cacher Faults 21](#_Toc374805107)

[5.4.4 Test Case 4 – RequestHandler Provides Cacher with Data after Successful Fault 21](#_Toc374805108)

[5.5 Cacher testing 21](#_Toc374805109)

[5.5.1 Test Case 1 – Cacher Returns Cached Data 21](#_Toc374805110)

[5.5.2 Test Case 2 – Cacher Returns a Fault when Data doesn’t Exist 22](#_Toc374805111)

[5.5.3 Test Case 3 – Cacher Implements a Least Recently Used Algorithm for Storage 22](#_Toc374805112)

[5.6 Logger testing 22](#_Toc374805113)

[5.6.1 Test Case 1 – Logger Logs to Appropriate File 22](#_Toc374805114)

[5.6.2 Test Case 2 – Logger Logs Debug Statements 23](#_Toc374805115)

[5.6.3 Test Case 3 – Logger Suppresses Debug Statements when not in Debug Mode 23](#_Toc374805116)

[5.7 Site Generator 23](#_Toc374805117)

[5.7.1 Language and Dependencies 23](#_Toc374805118)

[5.7.2 Design 23](#_Toc374805119)

[5.7.3 Running the Script 24](#_Toc374805120)

[6 Shuttle design 25](#_Toc374805121)

[6.1 Introduction 25](#_Toc374805122)

[6.1.1 Purpose 25](#_Toc374805123)

[6.1.2 Scope 25](#_Toc374805124)

[6.2 Overall architecture 25](#_Toc374805125)

[6.2.1 Description 25](#_Toc374805126)

[6.2.2 Overview diagram 26](#_Toc374805127)

[6.2.3 Flow diagram 27](#_Toc374805128)

[6.3 Daemon (WebServer Class) architecture 28](#_Toc374805129)

[6.3.1 Daemon (WebServer Class) description 28](#_Toc374805130)

[6.3.2 Daemon (WebServer Class) UML Diagram 29](#_Toc374805131)

[6.3.3 Daemon (WebServer Class) flow of processes 30](#_Toc374805132)

[6.4 Cacher architecture 31](#_Toc374805133)

[6.4.1 Cacher description 31](#_Toc374805134)

[6.4.2 Cacher UML Diagram 31](#_Toc374805135)

[6.4.3 Cacher flow of processes 33](#_Toc374805136)

[6.5 RequestHandler architecture 33](#_Toc374805137)

[6.5.1 RequestHandler description 33](#_Toc374805138)

[6.5.2 RequestHandler UML Diagram 34](#_Toc374805139)

[6.5.3 RequestHandler flow of processes 36](#_Toc374805140)

[6.6 Logger architecture 36](#_Toc374805141)

[6.6.1 Logger description 36](#_Toc374805142)

[6.6.2 Logger UML Diagram 37](#_Toc374805143)

[6.6.3 Logger flow of processes 38](#_Toc374805144)

[7 Development History 40](#_Toc374805145)

[7.1 Shuttle Phase I 40](#_Toc374805146)

[7.1.1 Shuttle Phase I Overview 40](#_Toc374805147)

[7.1.2 Shuttle Phase Schedule 40](#_Toc374805148)

[7.1.3 Shuttle Phase Notes 40](#_Toc374805149)

[7.1.4 Completed Tasks 40](#_Toc374805150)

[7.1.5 Upcoming Tasks 41](#_Toc374805151)

[7.1.6 Daemon (WebServer Class) Phase I (Kyle Nickel) 41](#_Toc374805152)

[7.1.7 Cacher Phase I (Matthew Shortall) 42](#_Toc374805153)

[7.1.8 RequestHandler Phase I (Daman Camara) 43](#_Toc374805154)

[7.1.9 Logger Phase I (Zachary Sorenson) 44](#_Toc374805155)

[7.2 Shuttle Phase II 45](#_Toc374805156)

[7.2.1 Shuttle Phase II Overview 45](#_Toc374805157)

[7.2.2 Shuttle Phase Schedule 45](#_Toc374805158)

[7.2.3 Shuttle Phase Notes 46](#_Toc374805159)

[7.2.4 Completed Tasks 46](#_Toc374805160)

[7.2.5 Upcoming Tasks 47](#_Toc374805161)

[7.2.6 Daemon (WebServer Class) Phase II (Kyle Nickel) 47](#_Toc374805162)

[7.2.7 Cacher Phase II (Matthew Shortall) 48](#_Toc374805163)

[7.2.8 RequestHandler Phase II (Daman Camara) 49](#_Toc374805164)

[7.2.9 Logger Phase II (Zachary Sorenson) 50](#_Toc374805165)

[7.3 Shuttle Phase III 51](#_Toc374805166)

[7.3.1 Shuttle Phase III Overview 51](#_Toc374805167)

[7.3.2 Shuttle Phase Schedule 51](#_Toc374805168)

[7.3.3 Shuttle Phase Notes 51](#_Toc374805169)

[7.3.4 Completed Tasks 52](#_Toc374805170)

[7.3.5 Daemon (WebServer Class) Phase III (Kyle Nickel) 53](#_Toc374805171)

[7.3.6 Cacher Phase III (Matthew Shortall) 53](#_Toc374805172)

[7.3.7 RequestHandler Phase III (Daman Camara) 54](#_Toc374805173)

[7.3.8 Logger Phase III (Zachary Sorenson) 55](#_Toc374805174)

[7.4 Shuttle Phase IV 56](#_Toc374805175)

[7.4.1 Shuttle Phase IV Overview 56](#_Toc374805176)

[7.4.2 Shuttle Phase Schedule 57](#_Toc374805177)

[7.4.3 Shuttle Phase Notes 57](#_Toc374805178)

[8 Test Results 58](#_Toc374805179)

[8.1 Unit Testing 58](#_Toc374805180)

[8.2 Application Testing 58](#_Toc374805181)

[8.2.1 Volume Testing 58](#_Toc374805182)

[8.2.2 Traversal Testing 59](#_Toc374805183)

[9 Conclusions 61](#_Toc374805184)

[9.1 Lessons Learned 61](#_Toc374805185)

[9.1.1 Source Control 61](#_Toc374805186)

[9.1.2 APIs 61](#_Toc374805187)

[9.1.3 Communication 62](#_Toc374805188)

[9.1.4 Maven 62](#_Toc374805189)

[9.2 Design Strengths 63](#_Toc374805190)

[9.2.1 Separation of Responsibilities 63](#_Toc374805191)

[9.2.2 Threading 63](#_Toc374805192)

[9.2.3 Caching 63](#_Toc374805193)

[9.3 Limitations 64](#_Toc374805194)

[9.3.1 POST Requests 64](#_Toc374805195)

[9.3.2 Sessions 64](#_Toc374805196)

[9.4 Future Improvements 64](#_Toc374805197)

[9.4.1 POST Requests 64](#_Toc374805198)

[9.4.2 Sessions 64](#_Toc374805199)

[9.4.3 SSL 64](#_Toc374805200)

[9.4.4 Scripting 65](#_Toc374805201)

[9.5 End Notes 65](#_Toc374805202)

# 1 Overview

This project is a group collaboration for the University of Maryland University College CMSC495 course and intends to provide a real-world implementation of a real-world problem within the Computer Science profession. For this project, the intended deliverable is a simple web server. This server shall require concurrency and sockets to effectively and efficiently handle multiple server requests. It is also intended that the web server be able to effectively and efficiently return files upon request.

The Shuttle Web Server project consisted of four individuals: Kyle Nickel, Daman Camara, Zachary Sorenson and Matthew Shortall. Each of the group members were assigned one main functional component within the proposed architecture:

Kyle Nickel (Shuttle Daemon (WebServer Class))

Daman Camara (Shuttle RequestHandler)

Zachary Sorenson (Shuttle Logger)

Matthew Shortall (Shuttle Cacher)

While the group collectively determined pertinent design decisions, each person outlined as the component lead was ultimately responsible for the components design, implementation, integration and testing. Several efforts throughout the project were collaborative.

# 2 Project Plan

## 2.1 Introduction

### 2.1.1 Purpose

The Shuttle web server project is designed to be a basic web server that responds to HTTP GET requests. It will be able to handle multiple concurrent connections utilizing Java threading. Shuttle will also cache the most recent hits to increase the speed of returning requests.

### 2.1.2 Scope

Shuttle is not meant to be on the same scale as Apache HTTPD, but instead a small basic server. It will provide concurrency for multiple connections and respond to HTTP GET requests with parsing of parameters. Additionally, it will log all connections and served pages.

Shuttle will also implement a basic caching system to quickly return recently requested pages.

This document shall outline the components of the Shuttle web server to include the Daemon (WebServer Class), Logger, Cacher and RequestHandler. The document has been written with language intended for use by system architects, designers and developers.

## 2.2 Definitions

|  |  |
| --- | --- |
| **Term** | **Definition** |
| GET | A HTTP request providing parameters in the URL. |
| Caching | Keeping resources in memory for fast access. |
| Unit Testing | Testing on a method level for each class. |
| Git | A source code manager. |
| GitHub | A web service allowing for collaboration with Git. |
| Daemon (WebServer Class) | A master process that runs without user intervention. |

## 2.3 Skills Required

Java

Socket Programming (Java)

Unit Testing

Application Testing

Caching Algorithms

File I/O

Git

## 2.4 Environment

Development of the Shuttle web server shall be performed on a personal laptop, in conjunction with a university supplied Unix server. For this development, the Unix server will likely be UMUC’s Nova. Hardware and Software availability within Nova will be managed by the system administrators. Personal laptops may utilize Windows, Apple OSX, or a Unix/Linux build, with the OS system noted during development and testing documentation.

Shuttle shall utilize Java 7 (currently update 45), or the most recent up to date security release for the main development environment. This will allow Shuttle to run on a wide variety of platforms while mitigating development architecture conflicts. During the development phase, developers shall utilize either Eclipse or Netbeans IDE. The codebase will be stored in a Git repository on GitHub.com at <https://github.com/nickelkr/shuttle>. Each participant should fork this repository and any changes committed back to this master branch.

Shuttle will be configured to utilize Transmission Control Protocol (TCP) network connectivity. TCP will handle Internet Protocol requests between the user’s browser and Shuttle. With that, the NetServer shall be configured to utilize TCP/IP.

For testing purposes, the Shuttle project will be using the JUnit test framework. This framework is well documented and widely used for Java unit tests. For validation during application testing, Firefox v25.0, Internet Explorer v10, and Google Chrome v30 will be used. Additional browser versions shall be utilized when appropriate and documented in follow-on design and testing documentation.

## 2.5 Milestones

### 2.5.1 Overview and Assignments

The Shuttle project is made up of the following components, with the assigned component leads:

|  |  |
| --- | --- |
| **Component** | **Assigned To** |
| Daemon (WebServer Class) | Kyle |
| Logger | Zack |
| Cacher | Matt |
| RequestHandler | Daman |
| Unit Tests | Everyone (Lead – Daman) |
| Sample Web Site | Kyle |
| Application Test | Daman |

Milestone schedule as projected during the Project Plan; modifications may be required as the project progresses. Each of the below Milestones are applicable to each of the web server components and shall be required from the assigned component lead, unless otherwise noted as a “Shuttle” milestone:

|  |  |  |
| --- | --- | --- |
| Period/Phase | Milestone Requirement(s) | Date |
| Project Plan | Requirements Specification & Systems Specification | 11/03/2013 |
| User’s Guide | Documented Usability | Test Site Uses | 11/06/2013 |
| Test Plan | Outlined Component Testing | Detailed Application Test | 11/10/2013 |
| Design | Component Design | Integration Design | Performance Estimates | 11/17/2013 |
| Phase I | Code Development Complete | Code Component Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Component Debug | Component Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

### 2.5.2 Daemon (WebServer Class)

**Note: The Daemon is called WebServer in the Java package.**

The Shuttle Daemon (WebServer Class) is the master process. This part of the application provides the user with the ability to start the application and provide options like port number.

It will also be the class handling the incoming connections. As Shuttle receives a connection request, it will be in charge of creating a new thread and handling the request with a RequestHandler instance.

This Daemon (WebServer Class) will also be in charge of managing all sub-processes for RequestHandler, Logger and Caching instances.

When the user kills the Shuttle Daemon (WebServer Class) he or she is stopping the server entirely.

### 2.5.3 Logger

The Logger is in charge of ensuring all major processes are documented and saved to the file system.

Any connections made to Shuttle should be documented with the corresponding IP address. It should also log any pages that are sent out to such IP addresses.

Changes to the state of the Shuttle Daemon (WebServer Class) should also be logged: starting, stopping.

The Logger should provide an API for each component of Shuttle to provide logging information. The Logger itself should handle time stamping and naming of each log line to differentiate component logging.

### 2.5.4 Cacher

The Cacher is responsible for keeping recently requested pages in memory. As pages are requested by outside systems, the RequestHandler should first look to the Cacher to see if Shuttle currently has the page in memory. If the page is stored in memory, Shuttle can serve the page without performing slower I/O operations. Otherwise, the RequestHandler should retrieve the page and provide the Cacher with the served page.

The Cacher should implement a Least Recently Used algorithm, LRU. It will be responsible for swapping pages in and out of memory, and also responding to page requests from the RequestHandler returning either the page requested or a fault.

### 2.5.5 RequestHandler

The RequestHandler is responsible for handling a single HTTP GET request per instance. The Shuttle Daemon (WebServer Class) will create an instance of the RequestHandler and pass the request in.

The RequestHandler should parse the request, along with any parameters, and then return the requested page from either the cache or the file system. If the page is not found, or another error occurs, the RequestHandler is responsible for conveying the issue with standard HTTP codes.

## 2.6 Testing

### 2.6.1 Unit Testing

Each team member shall write unit tests for each method in the class that they are responsible for developing.

The tests shall be written using the JUnit testing framework. Before committing any work to the production branch, all tests should pass JUnit testing. Additionally, all tests should meet any specific pass/fail criteria specified in separate documentation.

### 2.6.2 Application Testing

A small program is required to test Shuttle as a whole. This program shall use threading to create multiple connections and request many pages in quick succession. This will demonstrate the ability to concurrently handle connections, as well as test the caching system and page retrieval.

### 2.6.3 Test Site

To fully test Shuttle, a test web site will need to be created. This web site shall consist of no less than 1000 synthetically generated web pages. Page templates shall be created and utilized to automatically populate the test web site. The data and/or information on these pages will not be of great significance. Rather, these pages will represent a large web site and help to sufficiently test the capabilities and performance of the Shuttle web server. The java code for the test web site shall not be included in core Shuttle web server, however, it will be documented and saved in a repository.

# 3 Requirements Specification

## 3.1 Requirements Overview

This section of the documentation correlates the projected Shuttle web server project plan, implementations and milestones into component requirements. These requirements shall be completed by the assigned component lead, or Shuttle web server group as necessary.

This specification assumes a working environment consistent with that outlined in section 1.4 of the project plan. At the time of this document, there are no known constraints impacting the component requirements.

## 3.2 Daemon (WebServer Class) Requirements

### 3.2.1 Initialization

The Daemon (WebServer Class) shall be required to be initialized from the command line on an appropriate Operating System, consistent with the specified environments. This document recognizes this operating system in question to be Linux or Unix based.

PASS: The Daemon (WebServer Class) successfully initiates from command line options.

FAIL: The Daemon (WebServer Class) is incapable of being initialized in a passing state or experiences unexpected results.

### 3.2.2 Operational Until Exited

Once initialized, the Daemon (WebServer Class) shall operate until automated and testing is complete. This is imperative to the overall design and structure of the Shuttle web server, and is an integral piece of concurrent operation.

PASS: The Daemon (WebServer Class) operates until exited.

FAIL: The Daemon (WebServer Class) exits prematurely, prior to completed testing, or is inconsistent with passing criteria.

## 3.3 RequestHandler Requirements

### 3.3.1 Returns Page to User

The RequestHandler shall return user requested pages from Cache memory to the user. This requirement assumes the requested page is known and available. Should the page requested not exist, the RequestHandler should react appropriately as defined by additional requirements.

PASS: The RequestHandler successfully returns a known page to the user within an acceptable amount of time.

FAIL: The RequestHandler does not complete passing criteria, completes within an insufficient amount of time, or experiences unexpected results.

### 3.3.2 Queries the Cacher

The RequestHandler shall be integrated within the Shuttle web server to allow communications with the Cacher. The RequestHandler shall successfully submit queries to the running Cacher instance for pages contained within cache memory.

PASS: The RequestHandler initiates a page request to the Cacher, receiving expected results as outlined by all RequestHandler requirements.

FAIL: The page request is not successful, inoperational or returning unexpected results. Full integration with the Cacher cannot be validated or maintained.

### 3.3.3 Reads Files on Cacher Fault

The RequestHandler shall be integrated within the Shuttle web server to allow communications with the Cacher. The RequestHandler shall successfully read files into cache memory when a cache fault is obtained, and return these results to the user.

PASS: The RequestHandler successfully reads a file and returns this to the user when a cache fault occurs.

FAIL: Files are not read on cache fault, or unexpected results are observed.

### 3.3.4 Provides Files to Cacher on Successful Fault

The RequestHandler shall be required to integrate with the Cacher to ensure that pages not contained within cache memory are added to the cache after receiving a successful cache fault. This is required to ensure that pages not in cache are added to cache for increased performance. It is the responsibility of the RequestHandler to initiate this request with the Cacher.

PASS: The RequestHandler successfully calls to the Cacher to add pages to cache memory.

FAIL: The request fails to complete, cannot be made, produces errors or unexpected results. The requested page is not passed to the Cacher.

## 3.4 Cacher Requirements

### 3.4.1 Returns Pages From Cache

The Cacher shall be required to return pages from the cache to the RequestHandler upon request. This requirement assumes the page in question is contained within the cache.

PASS: The Cacher successfully returns the requested page from cache memory to the RequestHandler.

FAIL: A known page in cache is not successfully returned to the RequestHandler, or unexpected results are observed.

### 3.4.2 Returns a Fault

The Cacher shall be required to return a cache fault when a requested page is not contained within cache memory. This assumes integration with the RequestHandler and assumes the requested page is not contained within memory.

PASS: The Cacher successfully returns a fault on a request for a page not contained within cache.

FAIL: The Cacher returns a page, errors or obtains unexpected results on a request for a page not contained within memory.

### 3.4.3 Utilizes a Least-Recently-Used Algorithm

The Cacher implementation within Shuttle Web Server shall utilize a least-recently-used (LRU) algorithm for maintaining the cache. How the LRU algorithm is implemented will be discussed further in the design. The Cacher shall function as expected within the LRU.

PASS: The Cacher successfully utilizes the LRU algorithm for maintaining pages in cache.

FAIL: The Cacher does not utilize LRU, or receives errors or unexpected results following implementation.

## 3.5 Logger Requirements

### 3.5.1 Logs to Appropriate File

The Logger shall maintain a centralized file for write requests, regardless of log level. This requirement is not specific to any one functional component of the Shuttle web server, and should be withheld regardless of which component is making the request.

PASS: Upon successful request, appropriate log data is written to centralized log file.

FAIL: Error in logging observed, or written to file not specified at configuration.

### 3.5.2 Logs Debug Statements

The Logger shall provide debug statements within the centralized logging file when operating in debug mode. These statements are not generated by the Logger, but rather supplied by functional components.

PASS: Debug statements are visible and verifiable within the centralized log when operating within debug mode.

FAIL: Debugging statements are not visible when operating in debug mode, or unexpected results are obtained.

### 3.5.3 Suppress Debug Statements

The Logger shall suppress debugging statements when not operating in debug mode. These statements are provided by functional components as appropriate. This requirements assumes that debugging statements have been included within the Shuttle web server.

PASS: While debugging statements are provided, the logger successfully suppresses these statements from writing to the centralized logging file.

FAIL: Debug statements are available and visible within the log file when operating in a mode that is not debug.

# 4 User’s Guide

## 4.1 Introduction

### 4.1.1 Purpose

The purpose of this section is to provide the users all option available with the Shuttle web server.

### 4.1.2 Scope

This section is written for the users of the Shuttle web server. It does not include the inner workings of options or functions, but provides information about the available options to users and how to utilize them.

It will also cover the configuration file for the Shuttle web server and provide an example of such a configuration file. In this respect, this section also intends to guide the Shuttle server development and the following guides within this document.

## 4.2 Starting the Daemon (WebServer Class)

To start the Shuttle Daemon (WebServer Class), the user should run the following:

java –jar shuttle {options}

This will begin the Shuttle web server and {options} should be the options provided by the user.

## 4.3 Command line options

The following is a list of options. If providing a configuration file, no other options besides the –c option are required. All other provided options will override those in the configuration file.

|  |  |
| --- | --- |
| **Option** | **Description** |
| -c | Accepts a path to a valid configuration file. |
| -m | Mode setting: debug or production |
| -p | Specifies the port Shuttle will listen on |
| -l | Path to where Shuttle should log |

## 4.4 Configuration file

To have a constant configuration used, it is possible to define a key value pair configuration file to Shuttle. Shuttle will read this file and set the appropriate parameters when starting. The following is the structure:

mode: {debug | production (defaults to production)}

port: {port number to listen on}

directory: {top level directory to be served}

log\_file: {file where Shuttle should log}

The following is an example:

mode: debug

port: 80

log\_file: /var/log/shuttle.log

The keys for the configuration file do not need to be in any particular order and are not case-sensitive.

The path to this file should be provided to Shuttle when starting with ‘-c’ an example follows:

java -cp target/webserver-1.jar org.capstoneproject.classes.WebServer/etc/shuttle.config

To run the webserver with the default options run:

java -cp target/webserver-1.jar org.capstoneproject.classes.WebServer

All other options should be placed at the end.

Below is an example of use with all options except the configuration file:

java -cp target/webserver-1.jar org.capstoneproject.classes.WebServer –m production –p 80 –d /var/www/html –l /var/log/shuttle.log

This will start the server listening on port 80, in production mode, serving all files in /var/www/html, and logging to /var/log/shuttle.log.

To compile the Shuttle webserver from source use Maven by running:

mvn package

This command needs to be run from inside the project directory, shuttle/. This will compile the project and allow it to be run with the previously mentioned commands.

# 5 Test Plan

## 5.1 Introduction

### 5.1.1 Purpose

This section is designed to provide developers and testers required information to ensure that the development and release of the Shuttle web server performs as expected.

### 5.1.2 Scope

This section will cover application testing and the overall testing of Shuttle’s major components. It will provide testing guidelines with expected outcomes for a passing test all other outcomes should be considered failing.

This section will not cover unit testing. Unit tests shall be created by the programmer early and be granular. Unit tests are out of the scope of this section.

## 5.2 Application testing

To test the Shuttle web server, manual testing for browser support and automatic testing utilizing a script will be required.

### 5.2.1 Manual testing

The manual testing included in sections 3.2.1.1 thru 3.2.1.4 shall be performed on each of the three most widely used web browsers: Google Chrome, Mozilla FireFox, and Microsoft Internet Explorer. Specifics about browser versions can be found as stated in section 1.4 of the Project Plan, noted previously in this document

#### 5.2.1.1 Test Case 1 – Index page

Purpose: To test that Shuttle returns the index.html page when no page is specified.  
  
Pre-test requirement: Start the web server listening on port 80.

Operation: Enter 127.0.0.1, or IP address where Shuttle is running.

Expected output: The index.html page is returned to the browser with links to each of the subpages and Shuttle returns a 200 status code.

#### 5.2.1.2 Test Case 2 – Subpages

Purpose: To test that Shuttle returns subpages deeper in the file system.

Pre-test requirement: Start Shuttle listening on port 80.

Operation: Select and follow links from the index.html page and then continue to traverse links into the file system.

Expected output: When each displayed link is clicked, the page title shown in the link shall be returned to the user’s browser and Shuttle returns a 200 status code.

#### 5.2.1.3 Test Case 3 – Missing Page

Purpose: To test that Shuttle returns a not found error, 404, when a web page that doesn’t exist is requested.

Pre-test requirement: Start Shuttle on port 80.

Operation: Enter 127.0.0.1/nopage.html

Expected output: Shuttle should respond with “404 – Page Not Found”.

#### 5.2.1.4 Test Case 4 – Forbidden

Purpose: To ensure Shuttle does not return a page that does not have public access.

Pre-test requirement: Start Shuttle on port 80. Create a directory that should not be publicly visible (ie. /etc/passwd).

Operation: Enter “127.0.0.1/etc/passwd” into the browser.

Expected output: Shuttle should respond with “403 – Forbidden Request”.

### 5.2.2 Automatic testing

The automatic testing shall be performed with a small Java application. This application shall request the index page from the server and then perform a random walk of the file system.

This application shall be threaded to ensure Shuttle can handle multiple concurrent connections.

The testing application shall record results including failures and response times. The statistics shall be printed to the screen in the following format:

Shuttle Test Run:  
 Total Time:

# of Pages Requested:

# of Pages Returned:

# of Failures:

## 5.3 Daemon (WebServer Class) testing

To test the Daemon (WebServer Class), the User should start Shuttle via the command line with (on a Linux / Unix system):

nohup java –jar shuttle –c config.config &

Once Shuttle is started, the automatic testing and manual testing shall be performed. During this testing, the Shuttle will continue to run and be responsive.

## 5.4 RequestHandler testing

The RequestHandler shall be tested for the following basic features: querying the Cacher, reading requested file if faulted, return requested file to user, and provide Cacher with data.

### 5.4.1 Test Case 1 – Returns Page to User

Purpose: To ensure the RequestHandler returns requested pages to the requesting user.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of a web browser.

Expected output: The requested page is returned to the user.

### 5.4.2 Test Case 2 – Querying the Cacher

Purpose: To ensure the RequestHandler queries the Cacher to utilize cached data when a request is processed.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should also exist in cache.

Expected output: The RequestHandler initiates page request to the Cacher and returns the cached data to the user.

### 5.4.3 Test Case 3 – Read Requested File when Cacher Faults

Purpose: To ensure that the RequestHandler appropriately reads and returns the requested data when the Cacher returns a fault.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should not exist in cache.

Expected output: The RequestHandler initiates a page request to the Cacher that returns a fault. The RequestHandler should then read the requested page and return the data to the user.

### 5.4.4 Test Case 4 – RequestHandler Provides Cacher with Data after Successful Fault

Purpose: To ensure that the RequestHandler returns the requested user data to the Cacher for caching after a successful fault.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should not exist in cache.

Expected output: The RequestHandler initiates a page request to the Cacher that returns a fault. The RequestHandler should then read the requested page and return the data to the Cacher for caching.

## 5.5 Cacher testing

The Cacher shall be tested to ensure that when a page is requested, it returns the requested page or a fault. It shall also store any pages provided by a RequestHandler using a least recently used algorithm.

### 5.5.1 Test Case 1 – Cacher Returns Cached Data

Purpose: To ensure that if the Cacher has the requested data cached, it is returned to the RequestHandler.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should exist in cache.

Expected output: The Cacher shall check its cache data structure, find the requested data and return it to the RequestHandler.

### 5.5.2 Test Case 2 – Cacher Returns a Fault when Data doesn’t Exist

Purpose: To ensure that if the Cacher does not have the requested data it returns a fault.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should not exist in cache.

Expected output: The Cacher shall check its cache data structure. When it does not find the requested data it should return a fault to the appropriate RequestHandler.

### 5.5.3 Test Case 3 – Cacher Implements a Least Recently Used Algorithm for Storage

Purpose: To ensure that the Cacher implements the least recently used, LRU, algorithm for caching.

Pre-test requirement: Start Shuttle on port 80.

Operation: Request a known to exist page via the address bar of the web browser, this page should not exist in cache.

Expected output: When the RequestHandler requests a cache of previously faulted data the Cacher shall replace the least recently used data with the provided.

## 5.6 Logger testing

The Logger shall be tested to ensure it is logging appropriate messages.

### 5.6.1 Test Case 1 – Logger Logs to Appropriate File

Purpose: To ensure that the Logger writes all logging statements to the log file specified in the configuration file.

Operation: Start Shuttle on port 80. Select random links from index.html.

Expected output: The logging statements should be written to the file specified in the configuration file associated with the “log\_file” key.

### 5.6.2 Test Case 2 – Logger Logs Debug Statements

Purpose: When Shuttle is in “debug” mode all debug statements are logged to the log file.

Pre-test requirement: Start Shuttle with “mode” set to debug.

Operation: Observe log file.

Expected output: The log file should show more verbose, debugging, statements.

### 5.6.3 Test Case 3 – Logger Suppresses Debug Statements when not in Debug Mode

Purpose: To ensure if Shuttle is not in debug mode it does not log debug statements.

Pre-test requirement: Start Shuttle with “mode” set to production.

Operation: Observe log file.

Expected output: No “debug” statements are logged.

## 5.7 Site Generator

The Site Generator is a script designed to build a large sample web site to be used in testing Shuttle.

### 5.7.1 Language and Dependencies

This script shall be written in Ruby 1.9.2.

To fill out text sections of web pages, the Lorem Ipsum module of the Faker gem shall be used. This will allow random text to be entered into sections of the web page body.

### 5.7.2 Design

The script will take three arguments: number of files to be generated, number of directories to be used, the top-level directory.

The script will then iterate, creating a sample document that will be filled with a random number of paragraphs and links. The links will be filled with page links to pages that already exist.

For ease of design, the script will utilize a class that we will create providing the necessary data structures and randomization.

The index.html file will be created last and filled with links to all of the files that have been generated.

### 5.7.3 Running the Script

The script should be run by using the follow command on a Unix / Linux system:

ruby site\_generator.rb –p 1500 –d 50 –s /var/www/html

-p : the option for setting the number of pages to be generated

-d : the option for setting the number of directories to be created

-s : the starting directory (top level directory)

# 6 Shuttle design

## 6.1 Introduction

### 6.1.1 Purpose

The design section of this document is written for developers and architects. It has been developed to show the inner workings of the Shuttle Web Server.

### 6.1.2 Scope

This section contains detailed information about how the Shuttle Daemon (WebServer Class) works. It will cover the general architecture and also the architecture of each individual component in depth. Developers should be able to understand, extend, and develop the Shuttle Web Server with the contained information.

## 6.2 Overall architecture

### 6.2.1 Description

The Shuttle Web Server is comprised of four major components: Daemon (WebServer Class), RequestHandler, Cacher, and Logger.

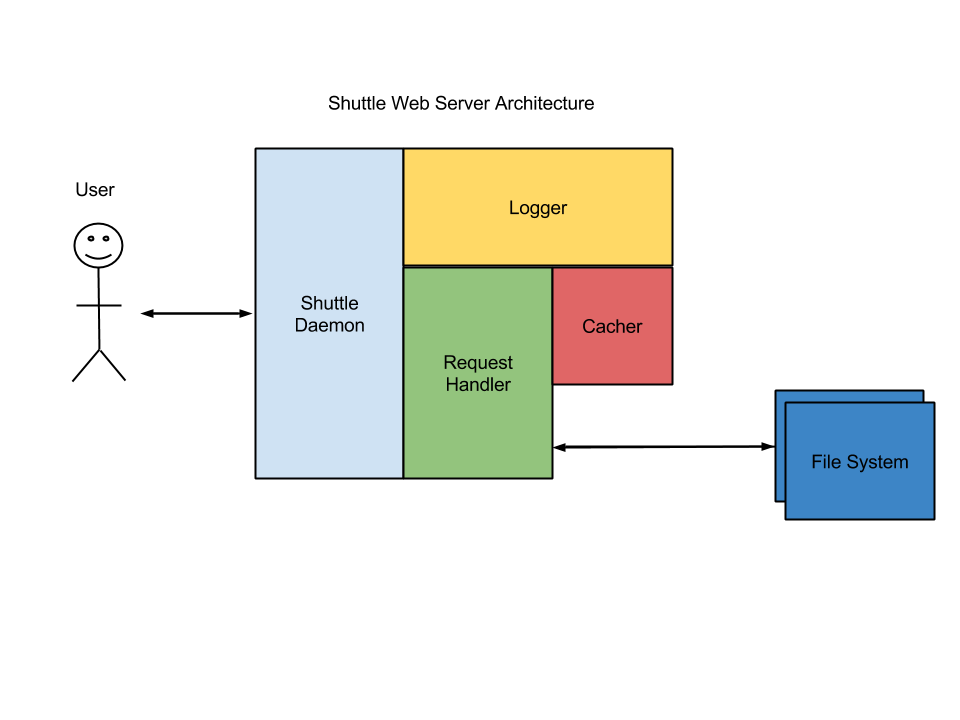
The Daemon (WebServer Class) is responsible for receiving incoming requests, via HTTP, and distributing these requests to individual RequestHandler objects. It is also responsible for creating instances of each of the other components and parsing the configuration file or user given options.

The RequestHandler is responsible for handling one HTTP request per instance of the RequestHandler. It shall parse the given request parameters, check with the Cacher for stored data, and retrieve any uncached requested resource. After the data has been retrieved the RequestHandler then returns the resource to the requestor and supplies the Cacher with the resource if it resulted in a cache miss.

The Cacher is responsible for ensuring any recently used resources are stored for quick access if requested again. It will respond to requests from the RequestHandler and return with the resource data or a miss response. If a cache miss occurs the Cacher is also responsible for storing data that the requesting RequestHandler provides.

The Logger is responsible for writing critical debugging and production level status updates from all components of Shuttle. The Logger will receive information from each component with log level (debug, info, warn, error), the component name, and the log string. It will then format the data, apply a time stamp, and write the information to disc.

### 6.2.2 Overview diagram



The above diagram shows the associations for all of the components in the Shuttle Web Server.

While the Daemon (WebServer Class) creates all processes it does not communicate with all components after creation. As is shown about the Daemon (WebServer Class) will communicate with RequestHandler and Logger objects after the system has been started.

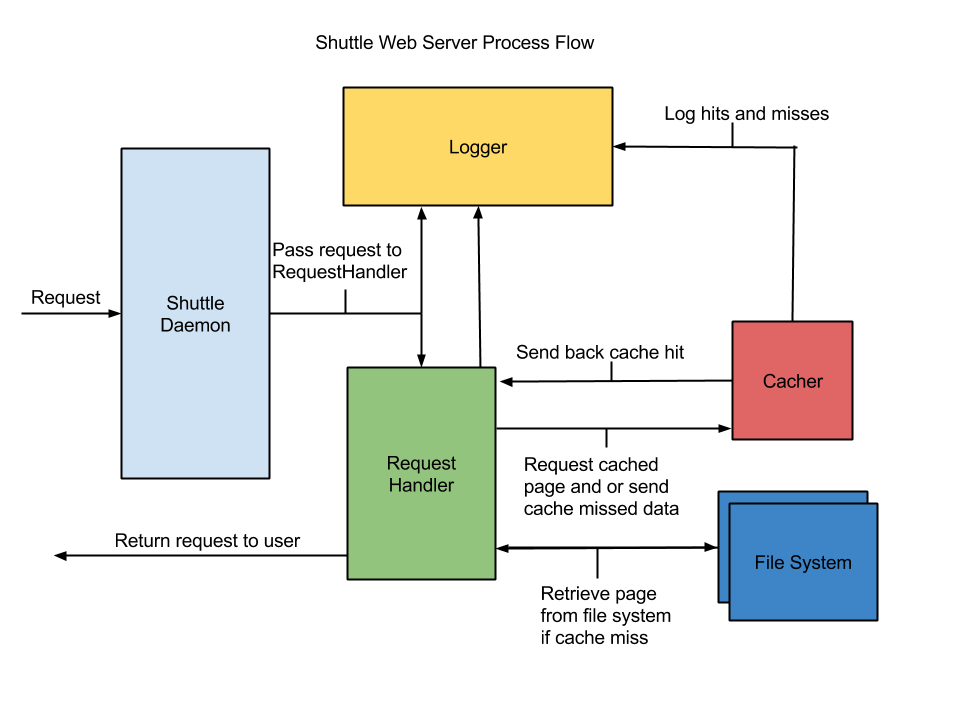
The Logger is required to communicate with all components at all times. It will need to be available to facilitate many concurrent-logging requests by all components.

The RequestHandler will communicate with the Cacher. It will request cached resources and provide resources that it reads from the file system, when a cache miss occurs. It will also provide it’s own logging strings to the Logger.

The Cacher will only communicate with the RequestHandler and the Logger. It will need to search its cache for resources that are requested from the RequestHandler. It will also be required to store any resources that a RequestHandler requests to be cached.

The file system is not apart of the actual web server.

### 6.2.3 Flow diagram



The above diagram demonstrates the process flow of the Shuttle Web Server at a high-level view.

When a user makes a request through their web browser via HTTP on port 80 the Daemon (WebServer Class) receives the connection and then spawns a new RequestHandler object.

The Daemon (WebServer Class) passes the socket and request to the newly created RequestHandler and logs the action.

The RequestHandler then parses the request. After determining the requested resource the RequestHandler then utilizes the Cacher’s API and requests the resource from the Cacher. If the Cacher has the data in cache it returns the data, if not it returns null.

If the data is cached it is immediately returned to the requestor and all transactions are logged.

If the data is not cached the Cacher returns null and then RequestHandler reads the resource from the file system. It then returns the resource to the requestor and asks the Cacher to store the data in its cache.

When a RequestHandler process requests data from the cache the Cacher will check its cache structure. If it finds the requested resource it will return the data. If it does not find the resource it returns null.

If the Cacher previously returned null, a cache miss, then it receives a later request to cache data. The Cacher stores the data in its caching structure. Each cache hit, miss, and storage is logged.

During all of the system interactions logging strings are passed to the Logger. The logger receives the log level which is one of the following: debug, info, warn, and error. It also receives the name of the component that is sending the log request and the string to be logged. It formats this data, applies a time stamp, and writes it to disc.

## 6.3 Daemon (WebServer Class) architecture

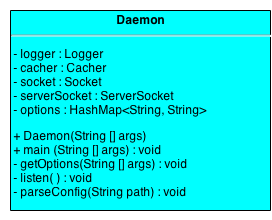
### 6.3.1 Daemon (WebServer Class) description

The Shuttle Daemon (WebServer Class) is in charge of creating all the sub-process components: Logger, Cacher, and RequestHandler. It handles are incoming requests while listening on port 80 for HTTP GET requests and spawns a new RequestHandler object to handle each incoming request.

The Daemon (WebServer Class) also facilitates the passing of static objects for logging and caching to each required component.

When a user specifies a configuration file the Daemon (WebServer Class) parses the file and inserts user specified options into the options data structure. If commands are passed via the command line these options are parsed and inserted into the options data structure.

### 6.3.2 Daemon (WebServer Class) UML Diagram



The Daemon (WebServer Class) is composed of the following data structures:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| logger | Logger | A static instance of the Logger component to be passed to other objects and be used by all objects. |
| cacher | Cacher | A static instance to cache, retrieve, and store resources. |
| socket | Socket | A socket for facilitating communications to the user’s browser. |
| serverSocket | ServerSocket | A socket that listens on port 80 for incoming requests. |
| options | HashMap<String, String> | A HashMap to store user or configuration defined options. |

The Daemon (WebServer Class) has the following methods:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| Daemon (WebServer Class) (String [] args) | Default constructor for the Shuttle Daemon (WebServer Class). |
| main (String [] args) | The main method that initializes the Daemon (WebServer Class) object. |
| getOptions (String [] args) | The method to parse command line options that are passed by the user and stores the values. |
| listen ( ) | The method containing the loop listening for incoming connections on port 80. |
| parseConfig (String path) | When a configuration file is used this method parses the information and stores it. |

### 6.3.3 Daemon (WebServer Class) flow of processes

**Start up:**

The user starts the Shuttle Web Server on the command line providing designated options.

The main method is run and creates the Daemon (WebServer Class) object passing the ‘args’ string array.

The Daemon (WebServer Class) starts the Logger and the Cacher and then calls the getOptions method.

**Configuration**:

The getOptions method is passed a string array that is parsed. When each option is encountered the option is saved as the key and the user provided value as the value.

If the user provides a configuration file it is then passed to the parseConfig method. The parseConfig method reads the file and iterates through each line of key value pairs storing each in the options data structure.

If an option is provided on both the command line and in the configuration file the command line option takes precedence.

**Incoming connections:**

The listen method is then called. Inside the listen method a loop continually runs listening for a request on port 80.

When a connection request is made on port 80 the Daemon (WebServer Class) spawns a new RequestHandler and passes the new object a socket for the client, a static Cacher object, and a Logger object.

The Daemon (WebServer Class) then continues to listen for more requests, repeating the process for each request.

**Logging requirements:**

The Daemon (WebServer Class) is required to log the starting process, each requested connection, and spawning of all RequestHandler processes.

## 6.4 Cacher architecture

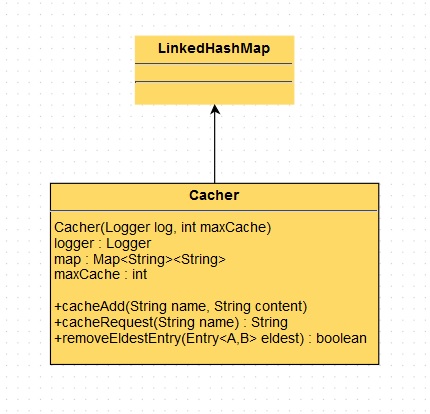
### 6.4.1 Cacher description

The Shuttle Cacher is responsible for adding file system objects to the cache. A LinkedHashMap instance of the Map interface is utilized for the cache. It handles incoming requests from the Daemon (WebServer Class) process to initialize. Once initialized, the Cacher interacts with the RequestHandler to return objects contained within the cache, render null objects when objects are not contained within the cache, and add objects to the cache when appropriate.

The Cacher follows a least recently used (LRU) paging algorithm to maintain the cache. Pages are added to the cache until all physical frames contain a page in memory. Once this has occurred, the algorithm replaces the least recently used page in a physical frame with that of the requested cache page addition.

The Cacher also interacts with the Logger to ensure valuable data is kept while interacting with the RequestHandler.

### 6.4.2 Cacher UML Diagram



The Cacher is composed of the following data structures:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| logger | Logger | A static instance of the Logger component passed by the Daemon (WebServer Class), and used by the Cacher to write cache request hits and misses. |
| map | Map<String><String> | An instance of Map used to facilitate the cache in a key/value relationship. |
| maxCache | int | A variable passed to the constructor by the Daemon (WebServer Class), and specifies the size of the cache. |

The Cacher has the following methods:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| Cacher (Logger log, int maxCache) | Default constructor for the Shuttle Cacher. |
| cacheAdd(String name, String content) | The method to add a file system object to the LinkedHashMap cache for faster retrieval. |
| cacheRequest(String name) | The method call to search the cache for a specific file system object. This method returns the String object if found, or null if the object is not in cache. |
| removeEldestEntry(Entry<A,B> eldest) | This method is called when an object is added to the cache, and removes the eldest entry in the cache following an LRU algorithm. |

### 6.4.3 Cacher flow of processes

**Start up:**

The Cacher is started by the Daemon (WebServer Class) processes. The Daemon (WebServer Class) calls the Default Constructor for the Cacher class and passes the location in memory of the Cacher object to the RequestHandler.

The Cacher does not have a main method, and therefore is reliant upon incoming connections for actions to be taken.

**Incoming connections:**

The Cacher receives incoming connections from the RequestHandler. The RequestHandler calls upon the Cacher to request an object stored in cache. This is achieved by calling upon the cacheRequest method and supplying the name of the object requested. If the name requested correlates to a key contained within the cache, the Cacher shall return the value for the key/name match. This represents a successful Cacher Hit. If the name requested cannot be matched to a key contained within the cache, the Cacher will return a null value to the RequestHandler, signifying that the object is not contained within the cache. This response represents a Cacher Miss.

**Logging requirements:**

The Cacher shall be required to log each Hit and Miss, for each cacheRequest called by each RequestHandler thread. Each log entry shall maintain appropriate formatting to detail the supplied Cacher response. For successful logging, the Cacher requires the Daemon (WebServer Class) supply the spawned Logger object, for outgoing communications.

## 6.5 RequestHandler architecture

### 6.5.1 RequestHandler description

For efficiency reasons, each user request for a web resource will be executed in a different Thread. Therefore the RequestHandler will implement the Runnable interface and override the run() method. To make the design more effective and modular, some of the original functionalities of the RequestHandler class have been extracted to the Request and Response Classes. These classes are used to encapsulate information usually associated with user requests, and server responses to the clients.

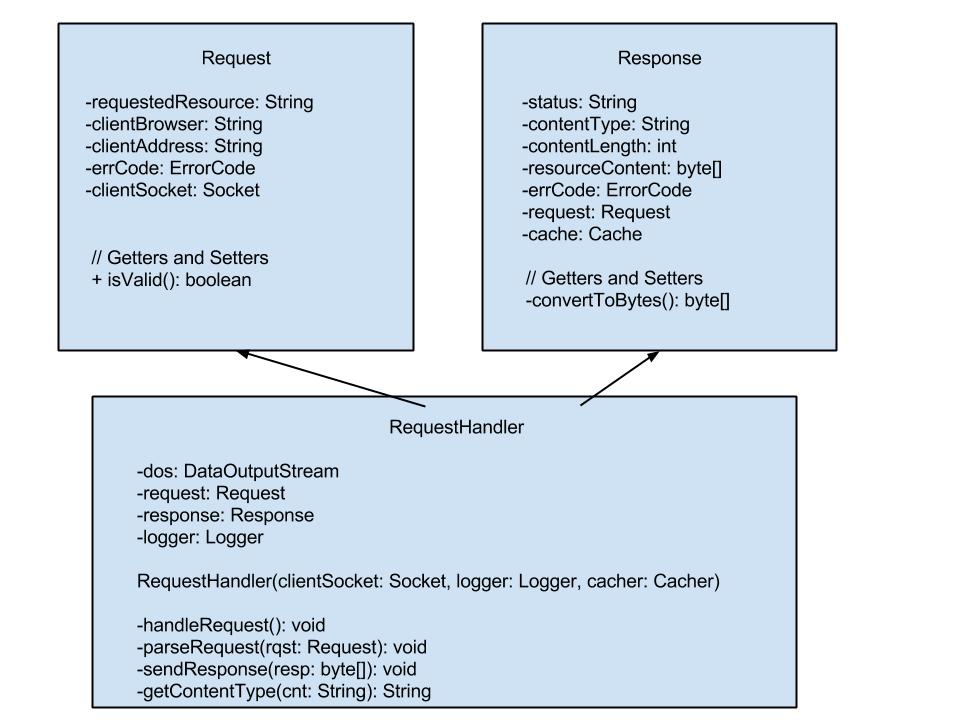
The RequestHandler’s responsibilities will be:

* To parse each user request, consult the cache to determine if there is a cached version of the requested web resource in order to avoid unnecessary file operations.
* Retrieve and send a byte stream form of the requested resource with appropriate headers information.

Some of the other tasks performed by the RequestHandler class are to log messages relevant to the Server’s operational status, and determine the type and set the type of the resource being requested based on the file extension.

The RequestHandler class is dependent on the Daemon (WebServer Class) class which listens for incoming user requests, and spawns a RequestHandler instance to service each request.

### 6.5.2 RequestHandler UML Diagram



The RequestHandler is composed of the following data structures:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| logger | Logger | A static instance of the Logger component passed by the Daemon (WebServer Class), and used by the RequestHandler to log relevant events during the Server’s operation. |
| Dos | DataOutputStream | An instance of the DataOutputStream class used to send a byte stream form of the response to the client. |
| Request | Request | Abstract Data Type encapsulating information relevant to a Request such as the client address and name of the resource being requested. |
| Response | Response | An ADT encapsulating information relevant to a Response such as the HTML status code, a byte array representation of the requested resource. |

The RequestHandler has the following methods:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| RequestHandler (Socket sock, Logger log, Cacher cache) | Constructor for the Shuttle RequestHandler. |
| handleRequest() | The method used to handle user requests. It parses the request, then takes actions depending on the status of the request which is indicated by an error code. |
| parseRequest(Request rqst) | The method used to parse each incoming request. It extracts information from the user request such as the client address and requested resource name. |
| sendResponse(byte[] resp) | The method used to send a response back to the client. It relies on the DataOutputStream instance of the class to carry out this feature. |
| getContentType(String cnt) | The method determines the content type being requested by examining the file extension. It returns a String indicating the type of resource being requested. |

### 6.5.3 RequestHandler flow of processes

**Start up:**

The RequestHandler is started by the Daemon (WebServer Class) process. The Daemon (WebServer Class) calls the Constructor for the RequestHandler class and passes the client socket, and static instances of the Logger and Cacher classes.

The RequestHandler does not have a main method, and instead depends on the Daemon (WebServer Class) class to spawn new instances as user requests arrive.

**Incoming connections:**

As users continue to request resources on the web server, a new instance of the RequestHandler class is created to serve the requests. The Daemon (WebServer Class) process will also be required to call the start() method on each RequestHandler instance created in order for the Thread to run to completion.

**Logging requirements:**

The RequestHandler class will contain an instance of the Logger class, and therefore will be able to log relevant events during the course of the Server’s operation.

## 6.6 Logger architecture

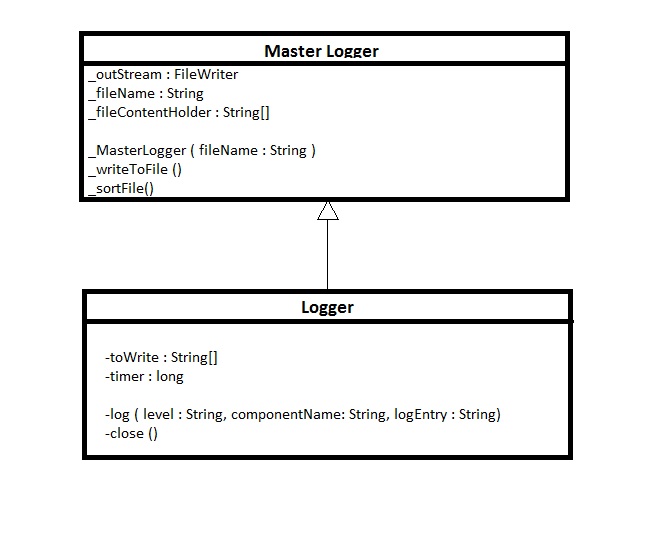
### 6.6.1 Logger description

The Logger exists to record the activities of the various components of the Shuttle webserver. Regardless of subcomponent, there will be a unified format for log information.

The Logger needs to handle concurrent log requests, without bottlenecking the other components. To accomplish this, the Logger is derived from a static super class which actually records data to file. A Logger subclass records data concurrently into temporary data structures before waiting in line to record to file. This wait is independent of the subcomponents of Shuttle that use Logger.

The super class will periodically, and prior to close, sort the log file in order to maintain a chronological record in spite of race conditions.

### 6.6.2 Logger UML Diagram



+

+

+Logger()

The MasterLogger is composed of the following data structures:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| outStream | FileWriter | A basic file stream to write data to file. |
| fileName | String | A file name to be specified by whichever process initializes the logger. |
| fileContentHolder | String[] | An array that holds each log entry, or line, for file sorting purposes. |

The MasterLogger has the following methods:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| MasterLogger (String fileName) | Default constructor for the Shuttle MasterLogger, static called by main method, passed output file name. |
| sortFile() | This method sorts the file chronologically by line. |

The Logger is composed of the following data structures:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| toWrite | String[] | Records log information line by line to be passed to writeToFile(String[]) method. |
| timer | long | Records system time at instance of log request in order to mark the time of that requrest. |

The Logger has the following methods:

|  |  |
| --- | --- |
| **Signature** | **Description** |
| Logger() | This constructs an instance of Logger, which starts a new thread. |
| log(string level, string subcomponent, string entry) | This is passed the debug level, the component of shuttle making the log request, and the entry to be recorded in the log. This data, along with time, is recorded in a temporary data structure owned by the Logger instance. |
| close() | This method tells this Logger instance to wait in line to record its data to the log file. |

### 6.6.3 Logger flow of processes

**Start up:**

The MasterLogger is first initialized by the main() method in the Daemon (WebServer Class). This creates a FileWriter.

Each subcomponent that would like to use the Logger then initializes an instance of Logger each.

**Log Process:**

The Logger then logs data with each log() method called for each instance. Each instance is its own thread, so it can log data without bottlenecking other processes.

When the subcomponent thread shuts down, it should call the close() method on its Logger instance. This tells the Logger to start waiting in line to write its data to the log file.

Every 5 seconds without write activity, the MasterLogger will sort its file. When the last thread shuts down, it will do the same before closing. This way the file is always sorted.

**Error Handling:**

If the Logger is passed log information deemed incorrect format, it will still make a log entry but not the discrepancy.

# 7 Development History

## 7.1 Shuttle Phase I

### 7.1.1 Shuttle Phase I Overview

At the Phase I milestone the Shuttle Web Server is on schedule. Overall basic functionality has been completed and a working prototype developed.

### 7.1.2 Shuttle Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Component Debug | Component Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

### 7.1.3 Shuttle Phase Notes

With the majority of the codebase having been developed and functional, Phase I is complete. In Phase 2 unit testing will commence and any required refactoring will be performed.

### 7.1.4 Completed Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Responsibility** | **Status** |
| Daemon (WebServer Class) – Listening | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Creation of required objects | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Basic testing | Kyle Nickel | Complete |
| Cacher – Allow for cache requests | Matthew Shortall | Complete |
| Cacher – Basic testing | Matthew Shortall | Complete |
| Cacher – Implement LRU | Matthew Shortall | Complete |
| RequestHandler – Parse user Requests | Daman Camara | Complete |
| RequestHandler – Determine Request status (200, 403, 404) | Daman Camara | Complete |
| RequestHandler – Send Response to Client | Daman Camara | Complete |
| RequestHandler – Consult Cache for cached pages | Daman Camara | Complete |
| RequestHandler – Provide default Web Page if Unspecified in URL | Daman Camara | Complete |
| Logger – Log strings | Zachary Sorenson | Complete |
| Logger – Log using debug level | Zachary Sorenson | Complete |
| Logger – Concurrent writes | Zachary Sorenson | Complete |

### 7.1.5 Upcoming Tasks

|  |  |  |
| --- | --- | --- |
| **Task** |  | **Status** |
| Daemon (WebServer Class) – Unit testing | Kyle Nickel | Incomplete |
| Daemon (WebServer Class) – Configuration modification | Kyle Nickel | Incomplete |
| Web Generation Script – Development | Kyle Nickel | Incomplete |
| Web Generateion Script – Testing | Kyle Nickel | Incomplete |
| Cacher – Unit testing | Matthew Shortall | Incomplete |
| Shuttle – Integration | Team Effort | Continuing |
| Shuttle – Application testing | Team Effort | Incomplete |
| RequestHandler – Log pages served | Daman Camara | Incomplete |
| RequestHandler – Ensure case-insensitivity in URL | Daman Camara | Incomplete |
| Shuttle – Provide global access to configuration Object (which contains Information such as the location of the default Web Root) | Daman Camara / Kyle Nickel | Continuing |
| Logger – File selection | Zachary Sorenson | Incomplete |
| Logger – Debugging features | Zachary Sorenson | Incomplete |
| Logger – Statement sorting | Zachary Sorenson | Incomplete |

### 7.1.6 Daemon (WebServer Class) Phase I (Kyle Nickel)

#### 7.1.6.1 Daemon (WebServer Class) Phase I Overview

The Shuttle Daemon (WebServer Class)’s basic functionality has been completed. It is currently creating all necessary objects and passing the correct arguments. The Daemon (WebServer Class) currently conforms to section 1.5.2.

The Daemon (WebServer Class) is in the same package as the RequestHandler and related classes. This is due to their close relationship and inter-workings. The Daemon (WebServer Class) is handling multiple incoming connections and spinning up new RequestHandler threads.

#### 7.1.6.2 Daemon (WebServer Class) Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Daemon (WebServer Class) Debug | Daemon (WebServer Class) Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.1.6.3 Daemon (WebServer Class) Phase Notes

With the Daemon (WebServer Class) currently in a functioning state Unit testing will begin during the next phase cycle. The Daemon (WebServer Class) also will require some refactoring and minor adjustments.

The Daemon (WebServer Class) is following object-oriented practices and is instantiated as a single object that then acts as the master process to the other sub processes. This code needs to be refactored to exclude any older non object-oriented code.

The configuration options still need to be implemented to provide better user functionality.

### 7.1.7 Cacher Phase I (Matthew Shortall)

#### 7.1.7.1 Cacher Phase I Overview

During Phase I of the Shuttle Web Server project plan, Casher code development begun. Initial elements of the development were tied to specific requirements as noted in section 1.5 & 1.6 of the Project Plan. Additional element decisions were correlated to testing requirements as noted in section 3.5, and design requirements specified in section 4.4 of the Design Document. Once all specified requirements could be mapped, initial coding of Phase I began.

A Cacher package was created to allow for ease of portability and allow for additional classes to be included, should follow-on requirements dictate such. A Cacher class was also created to house the cacher in its entirety. A default constructor was created for use by the controlling Daemon (WebServer Class) which specifies the size of the cache. Additionally, two methods were implemented for interaction with the RequestHandler; cacheRequest and cacheAdd. The cacheRequest method call returns the value in a key/value pair if the requested page was found to be contained within the cache. If a match cannot be made, the method call was developed to return a null array. Each of these responses are intended for handling by the RequestHandler. The cacheAdd method adds a key/value element to the synchronized LinkedHashMap.

During Phase I, the Least-Recently-Used algorithm was also implemented by overriding the removeEldestEntry method of the LinkedHashMap class. This method will identify the least recently used element of the map and remove it if the map size is greater than the allowed cache size specified by the Daemon (WebServer Class). Initial testing of the LRU algorithm implementation was conducted, with additional follow-on testing required.

Following Phase I of the Shuttle Web Server Project Plan, the Cacher implementation compiles correctly with initial testing efforts showing successful results.

#### 7.1.7.2 Cacher Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Cacher Debug | Cacher Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

**Status**: Phase I Complete and On Schedule.

#### 7.1.7.3 Cacher Phase I Notes

During Phase I of the Shuttle Web Server Project Plan, initial implementations of the Cacher class utilized an array of String values within the LinkedHashMap. Following initial group testing and review, it was found that better performance and I/O could be achieved if byte[] array values were utilized. This change was implemented during Phase I and slated for additional unit testing.

Additional changes were implemented in the Cacher class during Phase I to correct run-time errors encountered while calling the default constructor. It was found that the declaration of the LinkedHashMap as wrapped by a Collection of syncrhonizedMap needed to be declared within the default constructor. These changes were also implemented and successfully tested during Phase I.

### 7.1.8 RequestHandler Phase I (Daman Camara)

#### 7.1.8.1 RequestHandler Phase I Overview

The Shuttle RequestHandler’s basic functionality has been completed. As it stands, the request handler parses each user request, and reacts accordingly if it is able to successfully parse the request. While parsing, the RequestHandler checks to see if the User’s request contains the name of a default webpage such as index.html, if not then index.html will be selected as the default web page to display.

The RequestHandler also is able to react to the different types of Erroneous conditions. For instance, if the user tries to access a directory for which he or she does not have the proper permission, the user will be greeted with a 403 – Request Forbidden Web Page. There is also a similar behavior for a Bad Request that could be the result of a forged HTTP request.

#### 7.1.8.2 RequestHandler Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | RequestHandler Debug | RequestHandler Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.1.8.3 RequestHandler Phase Notes

Currently the RequestHandler class has most of the necessary functionalities required. However, more work needs to be done to refactor the code, and thoroughly test the Class, as the algorithm in the parseRequest() method does not appear to be very convincing. Another area that needs to be addressed relates to the default page displayed to users if they do not specify one. For instance, if the user types the following address: <http://localhost:3000/> on their browser, the user will be greeted with an index.html page. This behavior is likely to be altered to accommodate other types of files such as index.htm.

The RequestHandler will most likely require the Cacher class to provide a getter to its Logger instance.

### 7.1.9 Logger Phase I (Zachary Sorenson)

#### 7.1.9.1 Logger Phase I Overview

The Shuttle Logger is currently able to handle multiple concurrent log statement requests. It works first by instantiating a MasterLogger object when the Daemon (WebServer Class) starts. The other components then use a Logger object that provides them with methods write and close.

The component then calls write with a string and close to ensure that the MasterLogger object writes the log statements. The MasterLogger object handles the write requests and queues them accordingly allowing the components to continue processing without having to block for naturally slow write calls.

#### 7.1.9.2 Logger Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Logger Added Features | Logger Debug | Logger Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.1.9.3 Logger Phase Notes

The core Logger development has been split between Phase I and Phase II. Phase II will introduce more features to the Logger. These features include the ability to handle file selection by the call component.

Phase II development will also support the sort functionality. This is required to ensure that all log lines are representing according to their chronological order that could be skewed due to the concurrent nature.

Concurrency tests will be run during Phase II by all of the calling components as well as unit tests. The Logger is currently on schedule.

## 7.2 Shuttle Phase II

### 7.2.1 Shuttle Phase II Overview

Phase 2 focused on unit testing the individual components and implementing finer grain features.

The unit tests for the Daemon (WebServer Class), RequestHandler, and Cacher are complete and passing. The JUnit4 framework was utilized for testing the Daemon (WebServer Class) and RequestHandler. This allowed for easy integration of the tests and the ability to run them through the Eclipse IDE.

The RequestHandler is utilizing the Selenium browser test framework. This was decided when unit testing using the JUnit framework was not successful due to certain states that the RequestHandler requires to run properly.

All tests are currently passing.

The Logger’s API was finalized and implemented in all classes this week. With the

Logger now allowing for log level modes and passed in a passed in log file path it is ready to undergo extensive testing in Phase 3.

The Daemon (WebServer Class) now accepts both command line and configuration file options. This allows the environment to be customized by the user at run time and also for static or default configuration settings.

All components have been refactored and unnecessary or complex code removed.

The Shuttle project is on time and ready for full integration and application testing in Phase 3.

### 7.2.2 Shuttle Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Component Debug | Component Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

### 7.2.3 Shuttle Phase Notes

Due to the nature of the RequestHandler it was required to use another testing framework besides JUnit. The Selenium testing framework allows for browser testing and is required to running as well as the Shuttle application for the testing to work. This will allow the simulation of user clicks in the browser.

All components are working well with each other with some minor changes need in this phase for proper operation. These changes were not critical but instead necessary for some of the finer grain features.

### 7.2.4 Completed Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Responsibility** | **Status** |
| Daemon (WebServer Class) – Listening | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Creation of required objects | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Basic testing | Kyle Nickel | Complete |
| Cacher – Allow for cache requests | Matthew Shortall | Complete |
| Cacher – Basic testing | Matthew Shortall | Complete |
| Cacher – Implement LRU | Matthew Shortall | Complete |
| RequestHandler – Parse user Requests | Daman Camara | Complete |
| RequestHandler – Determine Request status (200, 403, 404) | Daman Camara | Complete |
| RequestHandler – Send Response to Client | Daman Camara | Complete |
| RequestHandler – Consult Cache for cached pages | Daman Camara | Complete |
| RequestHandler – Provide default Web Page if Unspecified in URL | Daman Camara | Complete |
| Logger – Log strings | Zachary Sorenson | Complete |
| Logger – Log using debug level | Zachary Sorenson | Complete |
| Logger – Concurrent writes | Zachary Sorenson | Complete |
| Daemon (WebServer Class) – Unit testing | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Configuration modification | Kyle Nickel | Complete |
| Cacher – Unit testing | Matthew Shortall | Complete |
| RequestHandler – Log pages served | Daman Camara | Complete |
| RequestHandler – Ensure case-insensitivity in URL | Daman Camara | Complete |
| Logger – File selection | Zachary Sorenson | Complete |
| Logger – Debugging features | Zachary Sorenson | Complete |
| Logger – Statement sorting | Zachary Sorenson | Complete |

### 7.2.5 Upcoming Tasks

|  |  |  |
| --- | --- | --- |
| **Task** |  | **Status** |
| Web Generation Script – Development | Kyle Nickel | Incomplete |
| Web Generateion Script – Testing | Kyle Nickel | Incomplete |
| Shuttle – Integration | Team Effort | Continuing |
| Shuttle – Application testing | Team Effort | Incomplete |
| Shuttle – Provide global access to configuration Object (which contains Information such as the location of the default Web Root) | Daman Camara / Kyle Nickel | Continuing |
| Logger – Unit Testing | Zachary Sorenson | Incomplete |

### 7.2.6 Daemon (WebServer Class) Phase II (Kyle Nickel)

#### 7.2.6.1 Daemon (WebServer Class) Phase II Overview

At the end of Phase 2 the Daemon (WebServer Class) now accepts command line options and can parse them from a configuration file. The command line options passed by the starting entity take precedence over the configuration file settings.

The unit tests for the Daemon (WebServer Class) have been completed and are currently passing. These tests ensure that all passed values, on the command line, are set accordingly and can be used both long and short hand. Unit tests also ensure that the configuration file settings are parsed and set.

#### 7.2.6.2 Daemon (WebServer Class) Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Daemon (WebServer Class) Debug | Daemon (WebServer Class) Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.2.6.3 Daemon (WebServer Class) Phase Notes

Command line options and the configuration file are optional and default values are assigned if none are provided by the starting entity.

The configuration file follows a key value pair structure:  
key:value

It reads line-by-line and only sets the value if it hasn’t already been set from a command line option.

With Phase 2 complete the Daemon (WebServer Class) has been refactored and all major features implemented. The unit tests ensure that the Daemon (WebServer Class)’s methods are working according to the plan.

In the upcoming Phase 3 integration testing will begin. During Phase 3, any changes that are required for the Daemon (WebServer Class) to properly work with the other components will be changed and tested.

### 7.2.7 Cacher Phase II (Matthew Shortall)

#### 7.2.7.1 Cacher Phase II Overview

During Phase 2 of the project, the Cacher milestone completed was unit testing. To facilitate this level of testing, the JUnit 4 framework was utilized to compile unit tests. For the Cacher, it was important to test the two available methods within the class; cacheAdd and cacheRequest. To test the cacheAdd method, it needed to be combined with the cache Request because the response to the assertation cannot be void. For this reason, unit testing included a unit test with expected pass results for a known object, as well as a test for expected fail results for an object known to be missing from cache. To date, all JUnit testing has concluded with passing results.

Additional work was performed during Phase 2 to implement Logging requirements. Initial logging was written within the source code to note required Cacher logging. With the Logger API now finalized, initial component integrations were performed in preparation for Phase 3 of the project. Initial tests show successful results as well.

#### 7.2.7.2 Cacher Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Cacher Debug | Cacher Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.2.7.3 Cacher Phase Notes

During Phase 2, it was observed that the Cacher contained methods that were outside of scope for some classes not contained within the Cacher package. To rectify these scope issues, some methods within the Cacher package were modified to be “public”, opening the methods up to additional class files within the Shuttle project. The alternative approach was to import the Cacher package to other components within the project, but the team felt that this would not be the best resolution.

Initial attempts to create JUnit tests for the cacheAdd method were not successful. These tests failed to run at compile time with Void errors. It was observed that because the cacheAdd method was Void, it could not be tested with the implemented assertation. This JUnit test was modified to run appropriately.

JUnit 3 framework tests were also created during this phase. They will not be utilized for this project, but were utilized during the creation of JUnit tests.

At this time, the Cacher component of the Shuttle project remains on-schedule with no integration issues expected during Phase 3.

### 7.2.8 RequestHandler Phase II (Daman Camara)

#### 7.2.8.1 RequestHandler Phase II Overview

The bulk of the code written for the RequestHandler during phase II dealt with Unit Testing. The Unit Testing is performed using Selenium which is browser automation software that facilitates the testing of web-based software. It does this by allowing the user to automatically interact with the web by opening a web browser, click on links, fetch content, etc… In order to run the tests, both the Selenium server and the shuttle Server need to be running as a pre-requisite.

The tests are implemented by opening the web browser and fetching a web page several times, and comparing the duration (in milliseconds) of the fetch times. Since file operations are time-consuming, web page retrievals from the cache are expected to take much less time than web page retrievals using file Input/Output.

At the end of phase II, the RequestHandler class behaves as expected. It is able to serve pages nested in directories, provide various messages as necessary based on the validity of the request.

#### 7.2.8.2 RequestHandler Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | RequestHandler Debug | RequestHandler Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.2.8.3 RequestHandler Phase Notes

The various classes associated with RequestHandler functionalities (most notably the RequestHandler class) have been refactored by removing redundant or unnecessary code.

Changes have been made to the Cacher class to provide the Response object with public access to its members, and allow the RequestHandler object access to the Logger instance.

The Unit Testing methodology used is Black-box testing, meaning that it is ignorant of the internal workings of the application. White-box testing was attempted with little to no success because of problems with keeping track of important program counters.

The parseRequest() method in the RequestHandler class has been modified to not address the scenario where the user omits the name of the page in the URL, until a better solution is found.

### 7.2.9 Logger Phase II (Zachary Sorenson)

#### 7.2.9.1 Logger Phase II Overview

The Shuttle Logger now has all planned features implemented. The format of the log entries has been formalized with the time stamp at the front for sorting purposes.

Debug is now highlighted in the log entries when subcomponents call for it. Debug log entries will stand out compared to standard entries. A standard entry must be passed “NORM” or “norm” in order to not display a debug status.

The file sorts automatically by chronological order of log entries. Depending on testing status, additional sort considerations may be implemented.

#### 7.2.9.2 Logger Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Logger Added Features | Logger Debug | Logger Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.2.9.3 Logger Phase Notes

During Phase II additional features were added to the Logger. Primarily, a sort was added so that the log entries in the output file would be sorted chronologically despite concurrency issues. Additionally, the format of the log entries was formalized, and a debug feature was added.

Phase II was going to implement a file select feature, but it was decided that this was something that should be left transparent to the user. The main method of the WebServer passes a file name to the Logger master static object.

Other components of the server have already integrated the core Logger functionality, so the Phase II features should integrate seamlessly. The Logger is now ready for extensive testing, which is the purpose of Phase III. The Logger is currently on schedule.

## 7.3 Shuttle Phase III

### 7.3.1 Shuttle Phase III Overview

Phase 3 focused on integration and application testing. Using curl, Daman, constructed a testing application that allowed the user to specify the number of requests to be made and also traverse directories.

A Ruby script was developed to generate a fake website. This script will generate the provided number of pages and spread them across the provided number of directories.

All components completed integration and application testing has been conducted successfully.

### 7.3.2 Shuttle Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Component Debug | Component Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

### 7.3.3 Shuttle Phase Notes

Utilizing the application testing script testing was completed. Multiple runs were made to test the speed and results of the fully integrated Shuttle web server. These results will be presented in the final presentation.

The application testing scripts uses a required application, curl and the website generator requires Ruby and the Faker gem.

All components have been finalized and integrated, duplicate files removed, and classes moved to better directory locations.

### 7.3.4 Completed Tasks

|  |  |  |
| --- | --- | --- |
| **Task** | **Responsibility** | **Status** |
| Daemon (WebServer Class) – Listening | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Creation of required objects | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Basic testing | Kyle Nickel | Complete |
| Cacher – Allow for cache requests | Matthew Shortall | Complete |
| Cacher – Basic testing | Matthew Shortall | Complete |
| Cacher – Implement LRU | Matthew Shortall | Complete |
| RequestHandler – Parse user Requests | Daman Camara | Complete |
| RequestHandler – Determine Request status (200, 403, 404) | Daman Camara | Complete |
| RequestHandler – Send Response to Client | Daman Camara | Complete |
| RequestHandler – Consult Cache for cached pages | Daman Camara | Complete |
| RequestHandler – Provide default Web Page if Unspecified in URL | Daman Camara | Complete |
| Logger – Log strings | Zachary Sorenson | Complete |
| Logger – Log using debug level | Zachary Sorenson | Complete |
| Logger – Concurrent writes | Zachary Sorenson | Complete |
| Daemon (WebServer Class) – Unit testing | Kyle Nickel | Complete |
| Daemon (WebServer Class) – Configuration modification | Kyle Nickel | Complete |
| Cacher – Unit testing | Matthew Shortall | Complete |
| RequestHandler – Log pages served | Daman Camara | Complete |
| RequestHandler – Ensure case-insensitivity in URL | Daman Camara | Complete |
| Logger – File selection | Zachary Sorenson | Complete |
| Logger – Debugging features | Zachary Sorenson | Complete |
| Logger – Statement sorting | Zachary Sorenson | Complete |
| Web Generation Script – Development | Kyle Nickel | Complete |
| Web Generateion Script – Testing | Kyle Nickel | Complete |
| Shuttle – Integration | Team Effort | Complete |
| Shuttle – Application testing | Team Effort | Complete |
| Shuttle – Provide global access to configuration Object (which contains Information such as the location of the default Web Root) | Daman Camara / Kyle Nickel | Complete |
| Logger – Unit Testing | Zachary Sorenson | Complete |

### 7.3.5 Daemon (WebServer Class) Phase III (Kyle Nickel)

#### 7.3.5.1 Daemon (WebServer Class) Phase III Overview

The Daemon (WebServer Class) has been completed. During the third phase only minor changes occurred in the Daemon (WebServer Class) this week.

Some changes to the logger use in the Daemon (WebServer Class) occurred. There was a missed close() statement which initiates the command to the MasterLogger to write the log statements to disk.

#### 7.3.5.2 Daemon (WebServer Class) Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Daemon (WebServer Class) Debug | Daemon (WebServer Class) Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.3.5.3 Daemon (WebServer Class) Phase Notes

The Daemon (WebServer Class)’s functionality has remained the same between the second and third phases. Only minor code changes have been made, none of which effect the operation of the class itself and are not visible to the other components or users.

The Daemon (WebServer Class) is complete and application testing complete. It has been fully integrated into the system and correctly uses all other system classes.

### 7.3.6 Cacher Phase III (Matthew Shortall)

#### 7.3.6.1 Cacher Phase III Overview

Phase III for the Cacher brought about additional unit testing and integration testing within the Shuttle web server. The Cacher has been fully integrated within Shuttle. Initial unit testing was successful during Phase II of the project, however, follow-on unit testing showed room for improvement. Phase III also validated application testing efforts for the Cacher’s integration within Shuttle. These tests were successful, and following a code review, will be marked complete and entering hardening.

#### 7.3.6.2 Cacher Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Cacher Debug | Cacher Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.3.6.3 Cacher Phase Notes

During this Phase of the project, not many changes were required for the Cacher. As noted in Section 7.3.1, integration testing identified necessary improvements for unit testing of the Cacher class. Unit tests were found to be compiling and returning, though results were not as expected. These tests were re-written and will be validated prior to hardening.

Phase III also identified an issue with the Cacher integrating with the Logger class. Initial attempts to integrate with the Logger API were successful, however, follow-on changes to the API required changes within the Cacher class. These changes were made and validated during follow-on testing.

No additional modifications were found to be required for the Cacher.

### 7.3.7 RequestHandler Phase III (Daman Camara)

#### 7.3.7.1 RequestHandler Phase III Overview

Since the RequestHandler met the requirements set forth in the beginning of the project, it did not require many changes during Phase 3. Most of the changes made during phase three involved integrating the various components of the application, and ensuring that they are interacting properly. For instance, the RequestHandler class was modified to switch from using a Stub for the Logger class to an actual Thread-safe and secure implementation.

#### 7.3.7.2 RequestHandler Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | RequestHandler Debug | RequestHandler Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.3.7.3 RequestHandler Phase Notes

Only one significant change was made to the RequestHandler during phase 3, which focused on testing.

There was an attempt to convert the current Cacher Test to a Unit Test.

While running the Application Testing class, it is assumed that the webroot is a folder called website.

### 7.3.8 Logger Phase III (Zachary Sorenson)

#### 7.3.8.1 Logger Phase III Overview

The main purpose of Phase III was testing. A shell program was written to test the logger with concurrency. Delays were built in so that the sorting function would also be tested.

Also, a formal API was finalized so other components can properly implement the logger with deterministic results.

Finally, logger was integrated with the other components and testing of the overall application has begun.

#### 7.3.8.2 Logger Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Logger Added Features | Logger Debug | Logger Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

#### 7.3.8.3 Logger Phase Notes

The following API guide was developed to help team members in use of the logger as they integrated it into their respective subcomponents:

Logger Classes:

MasterLogger  
Logger (child of MasterLogger)

General Description:

MasterLogger contains static variables which manage file operations. Any program using logger must build a MasterLogger object in order to initialize these static variables, to include giving the desired file name for the output file. MasterLogger sorts the data in the file continuously as it is accessed by child objects.

Logger maintains a dynamic array of log entries accessed directly from program subcomponents. A close method must be called when the subcomponent is finished logging so that the Logger object can access the MasterLogger methods to write its data to file. This process is thread-safe due to the existence of a static semaphore in the MasterLogger class.

Usage and Methods:

* To use the logger, first a MasterLogger object must be initialized, preferably in the main method, and before any logging is to take place. The object must be passed a String object containing the desired file name for the output file.

MasterLogger *name* = new MasterLogger(*String object: file name*);

* Each individual sub component of the program that is to log actual entries must create a Logger object.

Logger *name* = new Logger();

* In order to log an entry, the log() method must be used. It must be passed three String objects: the debug status, the name of the sub component using the logger, and the log entry text.

*loggerObject*.log(*String: debug status*, *String: subcomponent, String: log entry*);

* The debug status will be displayed directly at the end of each log entry unless the status is normal. “NORM” or “norm” must be passed as the debug status for this to occur.
* When the subcomponent is finished logging, or after the thread using the logger is closed, the specific Logger object must be closed in order for the data to be written to file.

*loggerObject*.close();

* This writes data to file, and sorts the file in chronological order

Output:

Normal –

YYYYMMDD\_HHMMSS : *subcomponent* : *logEntry*

Debug status-

YYYYMMDD\_HHMMSS : *subcomponent* : *logEntry* - [Debug Status - *debugText* ]

Example -

20131130\_175746 : requestHandler : Request Made By 12345

The test program successfully generated a log file with both concurrency and delays. Below is a sample of the output of that test file:

…

20131130\_175746 : testLog : 6part1 - [Debug Status - Test ]  
20131130\_175746 : testLog : 6part2 - [Debug Status - Test ]  
20131130\_175747 : testLog : 10part0 - [Debug Status - Test ]  
20131130\_175747 : testLog : 10part1 - [Debug Status - Test ]  
20131130\_175747 : testLog : 10part2 - [Debug Status - Test ]

…

## 7.4 Shuttle Phase IV

### 7.4.1 Shuttle Phase IV Overview

Phase IV focused on hardening the Shuttle webserver and packaging the final product.

Hardening included final application testing. This testing was completed utilizing curl and curls related GUI. It was tested using two different approaches. The first approach tested volume and return times. The second approach tested directory traversal.

Final packaging was completed using Maven. Maven was used for the entire project to ensure that all dependencies were resolved easily and requiring as little human downloading / interaction as possible.

### 7.4.2 Shuttle Phase Schedule

|  |  |  |
| --- | --- | --- |
| Phase | Requirement(s) | Date |
| Phase I | Code Dev Complete | Compilation | High-Level Testing | 11/24/2013 |
| Phase II | Component Debug | Component Unit Testing | Shuttle Compilation | Refactoring | 12/01/2013 |
| Phase III | Shuttle Integration and Application Testing | 12/08/2013 |
| Phase IV | Shuttle Hardening Phase | Final Packaging | 12/15/2013 |

### 7.4.3 Shuttle Phase Notes

During the testing portion of Phase IV the server was started as normal. With the server running the testing application is started. This application provides a GUI with the options to run a volume test or a traversal test.

With the volume test a number of requests is put in and then the application begins making requests to the server. Our server was tested with 1000 requests for each iteration. The requests are made and the final time it takes for the server to respond recorded. These results are shown in the test results sections.

Packaging of the application was fairly easy using Maven. The project has been using Maven from the start to handle application dependencies. By using Maven we were able to complete the packaging using the command:

mvn package

Running this command on the command line compiles and packages the source. The application is now compiled, packaged, and ready to run.

This also compiles and packages the tests as well.

# 8 Test Results

## 8.1 Unit Testing

Unit testing is key to ensuring that each component is working accordingly and run during the develop phases of the project.

As each component is modified the unit tests are run. All tests described in section three are currently passing.

## 8.2 Application Testing

### 8.2.1 Volume Testing

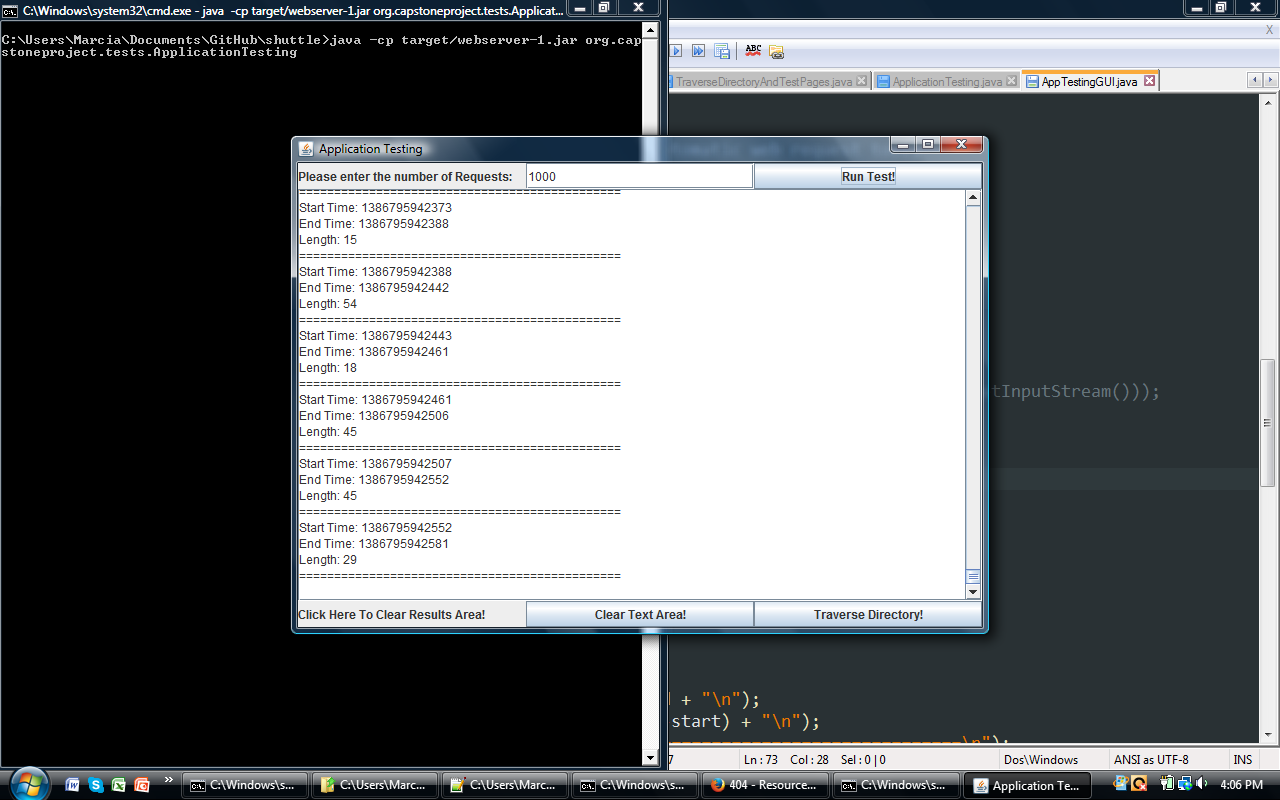
Volume testing was completed using the testing application that provides a GUI for curl. The test was configured to make 1000 requests against the server and it records the time for each set of 1000.

This test was run multiple times and the final results averaged out:

Testing with 1000 requests:

|  |  |
| --- | --- |
| **Run** | **Time** |
| Run 1 | 15 seconds |
| Run 2 | 54 seconds |
| Run 3 | 18 seconds |
| Run 4 | 45 seconds |
| Run 5 | 45 seconds |
| Run 6 | 29 seconds |
| **Average Time** | 34.33 seconds |

This averages out to 0.0343 seconds per page request. This is a very acceptable response time from the current version of the Shuttle webserver.



### 8.2.2 Traversal Testing

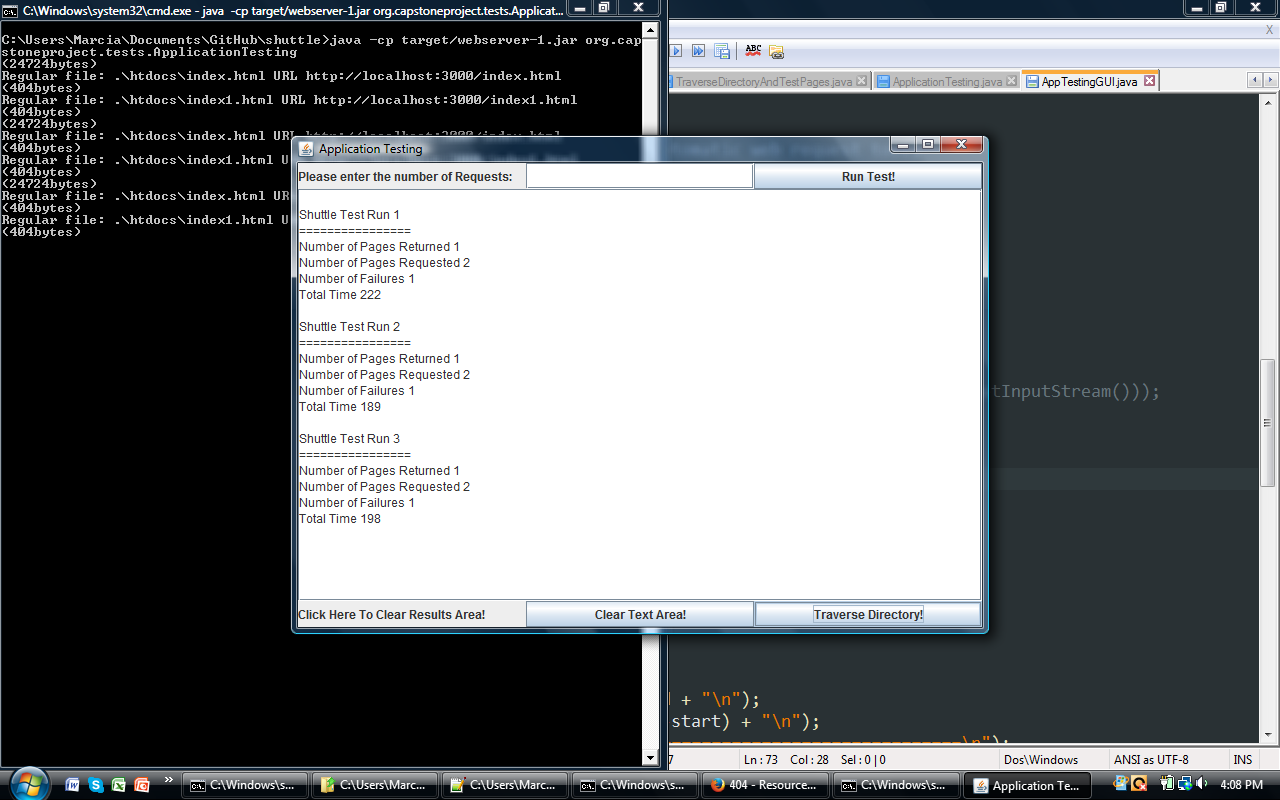
Traversal testing was accomplished using the same application as the volume test that runs using curl. To provide enough web pages for a good traversal test a Ruby script was developed to generate a fake website which is run by passing the arguments for path to create the files in, the number of pages to be created, and the number of directories for them to be spread across. The following is an example:

Ruby web\_site\_generator.rb /tmp/site/ 1000 5

Running the above command will generate a website consisting of 1000 pages across 5 directories in the /tmp/site directory. A index page with links to every page created is generated in placed at the top level of the directory as well.

After the site has been generated the testing application is run in traversal mode. This mode request one page from each page in the website.

This test was successful and demonstrated the ability to serve all pages the user has access to and not the ones that are restricted.



# 9 Conclusions

## 9.1 Lessons Learned

### 9.1.1 Source Control

When working on a team project and especially with each member not being co-located keeping the changes and source code inline is a difficult task. This proved to be a challenge but using Git and GitHub made the task mush less daunting.

With the source code kept under Git version control resolving and development conflicts that come about naturally. Git provided a very well documented and easy to use conflict management system.

GitHub works great with Git and allowed our team to host our source code online. By hosting the source code online each team member was able to have their own version of the code to make changes to. When changes were complete and the individual project set was functional the team member requested a merge with the master repository.

GitHub also records statistics such as a change log and contributor statistics. These can be helpful to see who has committed what code and what has been changed. It can be used to determine causes of issue or just to check on the status and history of the project.

The team did well to learn and use both Git and GitHub. At the end of the project we had to start the repository fresh due to major changes that were made to the structure and file removal. It is possible to accomplish this using some more advanced techniques but due to time constraints we chose this route.

Source control and hosting applications are critical in today’s development environment. Learning how to use Git will help when working on many projects as corporations utilize Git or similar source control applications.

### 9.1.2 APIs

APIs are critical to both the development and the overall running of the application. The team learned this early on when the components were divvied up between team members. A different team member developed each component, but each component interacted with at least one other. To ensure that development wasn’t blocked and each component was able to communicate with the others APIs needed to be finalized early.

Early finalizing of the APIs allowed everyone to focus on the development of their component and not worrying about the other components implementations. This was required to ensure that development continued moving concurrently.

If the finalization was not completed early the other components would have to wait for each component to be completed individually in order of dependence which is not acceptable as develop cycles would be wasted waiting.

How the APIs are designed was found to be critical as well. The entire team should be involved in determining the requirements for component or system APIs. The more concrete the API is the less that will be required to be changed later during development.

### 9.1.3 Communication

Communication is known to be a major key point with any team projects and the Shuttle project was no different.

Early in the project the team developed a habit of continually making email communication with all other team members. This allowed everyone to be on the same page when code was changed or opinions were needed.

Email was an effective form of communication during this project. It allowed for separate threads to be created and communication records to be referred to later.

By establishing email as the main form of communication early the team members responded to all project emails promptly. It was used to confirm work and finalize documentation.

Determining communication standards and habits is critical to an effective project. It ensures that everyone is on the same page and no one is developing blindly. This allowed smooth changes and frequent team participation.

### 9.1.4 Maven

No projects set out to reinvent the wheel so most will utilize third party packages. Shuttle did as well. Most of the time these packages require manual downloading and tracking by the developers. This can become tedious and actually cause quite a few issues during development.

Maven is what Shuttle used to handle third party packages, tracking, and compiling. Maven proved to be a great asset to the team. As each person is committing different code that sometimes included new third party packages the team was able to run the Maven command. Maven would then reach out to the Internet and download the new package.

Maven not only downloaded the packages but also tracked the packages allowing Maven to be used to compile and package the project itself.

Maven allowed the team to not worry about third party packages and build files. It managed this for the project and was detrimental to moving the development along on time.

## 9.2 Design Strengths

### 9.2.1 Separation of Responsibilities

Shuttle was designed to take full advantage of encapsulation. Each component was developed as its own class or classes.

Separating the core system responsibilities into different classes each component developer had to worry only about the component he was developing and not the implementation of the other components.

This proved to be very effective. It allowed changes to be made to a class while nothing or only minor changes were made to the other classes. This made the development move smoother and the application to be more stable.

### 9.2.2 Threading

Threading is key to a webserver. Without threading each HTTP request would have to wait until all other before it were finished. This would make for a very slow and unacceptable webserver.

To alleviate this issue all requests generated a new thread. The new thread handled only one request and then died once it was done. This kind of threading architecture allowed for fast response times and concurrent users to access the webserver’s resources.

Threading wasn’t only crucial to handling HTTP requests but also in the logging system. The Logger was required to handling logging for many different threads and yet keeps time continuity. By creating a master Logger thread and the child threads for each component the system is able to log effectively without blocking any of the component processes.

Threading was key to not only the successful completion of the webserver but it also is the reason it is able to run with acceptable speeds.

### 9.2.3 Caching

Requests are not usually unique. Many of the same pages are requested many times. This can cause a significant load on the server if handled like a new request every time.

To reduce the load and increase response times on the server a caching system was developed. As new requests came in the RequestHandler queries the Cacher to see if that resource is currently stored in memory. If it is the resource is handed back but if it isn’t then it is handled like any new request.

This design played a major role in the speed of the webserver. Hard disk I/O is the slowest operation on a computer and the Shuttle webserver is able to reduce the amount of hard disk I/O buy keeping recently requested data in memory which in turn increased overall system speeds.

## 9.3 Limitations

### 9.3.1 POST Requests

The current version of the Shuttle webserver only handles GET requests. While these requests are very common, probably the most common, HTTP requests POST still exist and have a commonly used protocol for forms and “hiding” data.

### 9.3.2 Sessions

Sessions are a major part of subscription-based websites. It allows users to stay logged in after provided the credentials one time.

The Shuttle webserver does not currently support sessions.

## 9.4 Future Improvements

### 9.4.1 POST Requests

Handling POST requests would greatly expand the Shuttle webserver and allow for larger forms and data passing. All major webserver handle POST requests and future development should begin by focusing on this implementation.

### 9.4.2 Sessions

Allowing a user to continue their session after logging in once and keeping these sessions active for a period of time is a great feature. This would provide a very common feature to the Shuttle webserver.

### 9.4.3 SSL

SSL allows for the verification of websites and systems. This helps to ensure users that they are on the correct website and not on a faked site that may be looking to exploit them.

Future versions of the Shuttle webserver should implement SSL support to provide the common security feature.

### 9.4.4 Scripting

Many websites provide dynamic data and formatting to users. Using scripting languages creates this dynamic content. The most common server side scripting language for websites is PHP.

When a user requests a web resource that includes a PHP script the webserver would run the information through a PHP interpreter and then generate the resulting page from the interpreter’s results.

By implementing the ability to use scripting languages Shuttle will be able to provide a system to handle very dynamic websites.

## 9.5 End Notes

The Shuttle webserver project allowed the team to not only utilize the information that was learned over the course of the computer science degree program but also learn to use tools and develop methods for team programming.

The computer science program taught many concepts that could be implemented in the Shuttle project. Concurrency, object-oriented paradigm, and software engineering are all skill sets that were required during development. Without this knowledge the Shuttle webserver would not have been successful.

Learning to use tools is critical in and outside of academic settings. It allows teams to work effectively together even if they do not work in the same location and to reduce the amount of work required to keep all the packages in line and source code up to date.

Team coordination and communication were two of the biggest hurdles for the project. These also represent common issues in the workplace. Learning to use these tools and developing good teamwork habits was crucial to the success of the Shuttle webserver.