Final Project

Secured WebServer in Java

Developed and Reported by: Farhan Sabbir Siddique

ID: 1331106

Course: CNC509

Date: 19-09-2020



**Abstract**

The final project required the following:

* A web server, running on top of SSL (Secured Socket Layer)
* Receive and parse Http request
* Create a response and send the requested file if exists
* Else send Http not found status.
* If the server is unable is open listening port, it will failover to a secondary port.

The project also requires couple of criteria:

* Chained filter streams
* Multithreaded Client/Server
* Avoiding race condition and deadlocks
* Encrypted communication between client and server
* Client authentication for server program
* Cookie support

**Assumptions**

The task requires couple of assumptions regarding the environment the software will be run on:

* Being Java based, the software is generally Operating System agnostic. However, it is found to have worked well under the below Operating environments:
  + MacOS 10.15.6, 32GB memory, 8 core Intel i9. JVM 1.8.206
  + Fedora Linux 32, kernel 5.8.7, 16GB memory, 2 core Intel i5. JVM 1.8.218
  + Windows 10, 8GB memory, 2 core Intel i7. JVM 1.8.190
* Software needs elevated privilege to run on port 80, although such exception is handled from inside the application itself where it falls back to port 8080.
* All parameters required for application functionality is hard coded in an interface called Defs. Application uses no command-line parameters.
* The www and keystore.jks requires to be present in the working directory of the application.

**Known limitations**

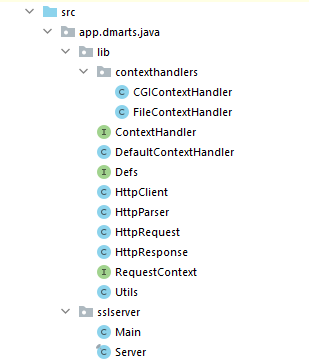
Below are some of the key known limitations of the software:

* Does not handle cookies (deliberately left out).
* Does not handle client authentication (final version may include this functionality).
* Uses self-signed keys (created using java’s built in keytool).
* Chrome browser does not work.
* Application is not profiled for CPU and memory for workloads.
* All configurations are hard-coded and mostly placed in Defs interface as final.
* Does not scale very well with more than 1000 files in www folder.
* Does not handle /favicon.ico request.
* Only support below mime-types:
  + HTML, TXT, CSS, JS, PNG, JPEG
* Does not support connection and additional request headers.
* Does not use NIO2 non-blocking IO channel for server-client communication.
* Does not follow MVC pattern, although minimal structure is kept for future extensions.
* Without the SSL handshake, the application performs well as expected, providing above 90% responses within first 10ms.
* Although capability is kept internally, but currently there’s no way to stop the server apart from killing the JVM.

**Architecture**

The application is divided into three packages in total:

* Main application from app.dmarts.java.sslserver.
* All dependent components are kept in app.dmarts.java.lib
* While all context handlers are kept in app.dmarts.java.lib.contexthandlers.



Below image depicts work-flow of the application:

Main

Thread

Server

Thread

Server Thread tasks

* List All files in www directory.
* Adds files only to a ConcurrentHashMap (CONTEXTHANDLERS) with context paths as keys and maps corresponding handlers to the paths as values.
* Creates a pool of client threads with pool size as defined in Defs.HTTP\_CLIENT\_BACKLOG parameter.
* Opens the keystore.jks file and creates a SSLServerSocket and starts listening on port 80 by default. On failure, the exception is handled from Main to start another instance of Server on port 8080.
* Server thread enters infinite loop.
* The server immediately hands over the client socket to HttpClient thread. The socket is returned from ServerSocket’s accept blocking method.
* The task from threadpool is kept for future capability enhancement, but no implementation is done on it as of now.
* The loop continues from the top indefinitely.

Figure 1 Main program execution flow

HttpClient

Thread

HttpParser

Client Socket

Http Request

HttpClient Thread Tasks

* HttpClient is barebones 11 lines of code.
* Immediately begins to parse the request:
* Creates a HttpParser object and passes the client socket
* HttpParser object’s parseHttpRequest method is called to get a HttpRequest object.
* The HttpRequest object is used to identify if ContextHandler has request’s context in its keys.
* If the key is available, ContextHandler’s handle method is called.
* Else DefaultContextHandler is created and it handles the context.
* The HttpClient exits after handle returns.

HttpParser Object Tasks

* The HttpParser reads from the client socket passed in its constructor and parseHttpRequest method is used to construct and return a HttpRequest object.

Figure 2 HttpClient to HttpRequest via HttpParser flow

ContextHandlers

handle(HttpRequest):void

ContextHandler

handle(HttpRequest):void

CGIContextHandler

FILE: String

handle(HttpRequest):void

FileContextHandler

handle(HttpRequest):void

DefaultContextHandler

Figure 3 UML of ContextHandlers

These are special purpose classes, built on top of interface ContextHandler. The interface provides only one method (handle) that takes an HttpRequest object. The implementing ContextHandlers are expected to write full code or business logic on the handle method. In partial implementation schematics, these ContextHandlers provide, in small context, a Controller part of the MVC framework. While handle actually is expected to call in an HttpResponse and allow the View part, but unlike proper segregation of View from Controller in MVC, current implementation does not distinguish this segregation.

The ContextHandlers are inherent part of the server application, being created at the very beginning of the Server thread runtime. This creates an early on attachment of ContextHandlers with the context paths, which can later be replaced with better or adequate handlers by simply updating the CONTEXTHANDLERS.

A note worth mentioning in this regard that, a class implementing ContextHandler interface can be extended to implement full range templating engine, and hence the proper View part of MVC.

The UMLs below depict class structure of HttpRequest and HttpResponse. While there are many other implementation details missing in this schematic diagram, but it does provide a high level overview of what to expect from the classes. The current implementation does not cover many exception handling due to lack of adequate time and limitation in the scope of final project.

METHOD, PATH, HTTP\_VER, QUERY: String

HEADERS: HashMap<String, String>

BODY: StringBuilder

CLIENTSOCKET: Socket

requestHasQuery(): Boolean

getQuery(): String

getRequestMethod(): String

getRequestBody(): String

getRequestHeaders(): HashMap<String, String>

getContextPath(): String

getClientSocket(): Socket

setClientSocket(Socket): void

HttpRequest

Figure 4 UML of HttpRequest

FIRSTLINE, BODY: String

HEADERS: HashMap<String, String>

HttpResponseBuilder: HttpResponse

toString(): String

getNotFoundHttpResponse(): HttpResponse

getOKHttpResponse(): HttpResponse

HttpResonse

Figure 5 UML of HttpResponse