Nicholas Mohan

Our project was a simple HTTP server written in java, utilizing java sockets classes. The design of the server is overall non complex and has been designed for a user to take the base code and twist it to their needs. The basic code of this server supports HTTP methods GET, POST, HEAD, OPTIONS, PUT, and DELETE. We have SSL integration and a blacklist for IPs that spam the server.

The server is written in two different files, a Server.java file and a HttpServer.java file. The server is run from the Server.java file. When the server runs, the first thing that happens is that two logger files are initialized. There is one file for general server activity and another for errors and error logging. Handlers are added for external logging and we set the level of everything to FINER. The next thing that is done is that the server sockets are initialized. Using the SSLServerSocketClass, first a factory is created, and then from that the server a server socket is created. This is the server socket that is used for connecting with the client sockets. The next thing that is initialized is the executor. This is used for the multithreading of the server. Our server uses the cached thread pool option to optimize the number of threads for the computer that this server is running on. The next thing that happens with the server is the connection with the client is made. The program then continues with a while loop that runs forever. Inside this loop, first the server socket accepts a new client, and that is stored in a new client socket. We set a timeout for this socket just so the client doesn't hang onto the server and waste server resources so that we can serve more clients. Next the client socket is passed into a new self made class HttpServer. This will be explained in detail in a bit. Next, this class is run with the executor class for multithreading and serving multiple clients at once. Finally, the last thing in the Server.java file is the server catches for exceptions. We catch both a SocketTimeoutException and a SSLHandshakeException just making sure that the program does not crash. The server exceptions are then logged in the error log file.

The next big part of the program is the HttpServer class. This class is responsible for the connection, parsing, and response to the client. HttpServer implements the runnable class, this means we implement the abstract run method for the executor multithreading purposes. In this class we start with some static global variables. We define a default file, index.html, a 404.html file, and the root folder relative path of where the server should look for the files requested by the client. We then have some class variables. They are the client socket, the two loggers, and a boolean variable that is responsible for keeping track of the clients use of the keep-alive header. We then have a constructor that initializes the client and the two logger files. The constructor takes these three things as arguments. The final method in this class is the httpserver method. It takes a socket as the argument, and it is called by the overridden run method. The first thing that this method does is log the client connection into the activity

log file. Then a new BufferedReader, PrintWriter, and BufferedOutPutStream is declared outside of the try block so that they can be closed in a finally after any exceptions are caught. Next, the method connects to the input and output streams of the client socket. Using this connect, the server then parses out the request line from the client request. This contains two important pieces of information. It holds the HTTP method requested, and the file requested. The server parses these out of the request line using a string tokenizer class. Then, the rest of the request headers and parsed. Below is the code that is responsible for this. It can easily be modified for parsing out different and new things in the headers:

```
while(!(temp.equals(""))){
    temp = in.readLine();
    StringTokenizer headParse;
    if(temp.startsWith("Connection:")){
        headParse = new StringTokenizer(temp);
        headParse.nextToken();
        connectKeepAlive = headParse.nextToken();
    }
    if(temp.startsWith("Content-Type:")){
        headParse = new StringTokenizer(temp);
        headParse.nextToken();
        contentType = headParse.nextToken();
    }
    if(temp.startsWith("Content-Length:")){
        headParse = new StringTokenizer(temp);
        headParse.nextToken();
        contentLength = Integer.parseInt(headParse.nextToken());
    }
}
```

The most important headers are the connection header, the content type, and the content length. The connection header is important because it tells us if the client is going to request more things and to keep the connection open if it does. Then the content type and content length are important for any HTTP requests that send information in the body of the request. The type and length allow the server to know how many bytes of information to expect in the body.

Next, the program splits the into if and if else statements. They all differentiate between the different HTTP requests sent to the user. The first response is both HEAD and GET because they are very similar. The first thing the server checks is if the the file requested is a folder or not. If it is a folder we concatenate the default file (index.html)

onto the file requested. We then make a file object out the requested file, while doing that we also get the length and the MIME type for the file. We get the MIME type from a method called getContentType. It takes a file object as an arguments and returns a string of the MIME type of the file. After we get the file information the last thing to do before sending a response is to convert the file into a byte array for sending. The server now takes the PrintWriter and prints lines out with all the response headers. These headers include the length and type of the returned file. It is very important to both flush out the output buffer once after the headers and again after the response body. We send the headers with the PrintWriter and the body containing the requested file back with the BufferedOutputStream. To differentiate between the GET and HEAD responses, they both are responded to with the headers, but HEAD responses do not contain any information in the body. The next request we respond to is the POST request. The response starts by taking the response body and putting it into a string. Then the server returns a response to the POST request returning whatever was requested by the client. Another example is the OPTIONS request. It is very similar to the HEAD request, except in the header it responds all of the options that the particular server is capable to responding to. Below is code that all of the response methods use to respond to the client with minor modifications. This is the OPTIONS response used:

```
out.println("HTTP/1.1 200 OK");
out.println("Allow: GET, HEAD, OPTIONS, POST, PUT, DELETE");
out.println("Server: TEST");
out.println("Date: "+new Date());
out.println("Content-length: 0");
out.print("\r\n\r\n");
out.flush();
actLog.finer("OPTIONS Request Returned");
```

There are other requests supports, but these are the interesting and noteworthy differences that can be seen in the between all of the requests supported. If the file requested could not be found in any of these HTTP requests, it calls a method called fileNotFound, which returns with a 404.html page. Then finally, any exception that is thrown is caught and logged in the error log file. After any exception is caught and logged, if the connection type was to close in the initial header, the connection is closed. If not the connection stays open and waits for another request from the client. This has a timeout value that will close if the client doesn't request any more data in a set amount of time.

Raymond Zhu

In the the process of completing implementations for the server, we ran into many problems. I was tasked with implementing SSL into the server so that we can implement authentication/authorization. I looked on Oracle.com for examples of how to get started with implementing SSL on both the server and the client. I noticed that on the server side there was no SSL socket but for the client there was. I took a client example to test it on our server with just a regular socket and it did not work.

```
SSLSocketFactory factory = null;
try {
    SSLContext ctx;
    KeyManagerFactory kmf;
    KeyStore ks;
    char[] passphrase = "passphrase".toCharArray();

    ctx = SSLContext.getInstance("TLS");
    kmf = KeyManagerFactory.getInstance("SunX509");
    ks = KeyStore.getInstance("JKS");

    ks.load(new FileInputStream("testkeys"), passphrase);

    kmf.init(ks, passphrase);
    ctx.init(kmf.getKeyManagers(), null, null);

    factory = ctx.getSocketFactory();
} catch (Exception e) {
    throw new IOException(e.getMessage());
}

SSLSocket socket = (SSLSocket)factory.createSocket(host, port);

/*
 * send http request
 *
 * See SSLSocketClient.java for more information about why
 * there is a forced handshake here when using PrintWriters.
 */
socket.startHandshake();
```

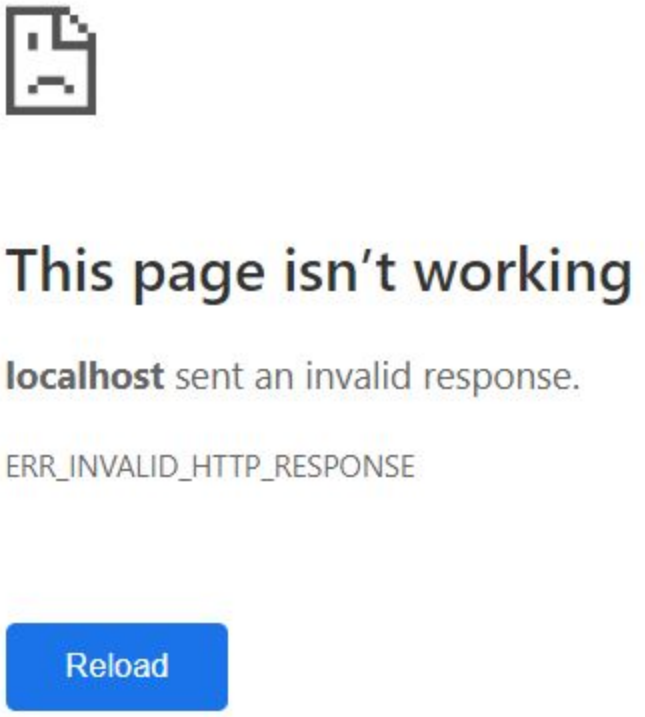
I then went on to trying just to get it working on the server so that localhost:port would post and run on the browser. It did not work after countless attempts of trial and error. The error we kept getting was SSL handshake error where it'd either not have the correct certificate or some other error. I made my own certificate in my root folder thinking this would fix it however it did not. I imported all the common CA certificates into the server and printed them on the screen to make sure they were all there.

```
Certificate cert = keystore.getCertificate(alias);
System.out.println(cert);
```

Even though they were all there, the localhost:port was still not working. I tried connecting with different browsers and computers but no luck.

```
try {
    ServerSocket serverConnect = new ServerSocket(PORT);
    //ServerSocket serverConnect = ((SSLServerSocketFactory)SSLServerSocketFactory.getDefault()).createServerSocket(PORT);
```

This is the line of code I tried to create a SSL socket for the server side.



This is what happens when I try to connect to localhost with the SSL used.

```
Connecton opened. (Fri Dec 14 13:40:19 EST 2018)
Server error : javax.net.ssl.SSLException: Unrecognized SSL message, plaintext connection?
Connection closed.
```

The terminal prints this out and I tried looking up solutions to fix this error but nothing fixed it.

Login Form

Username

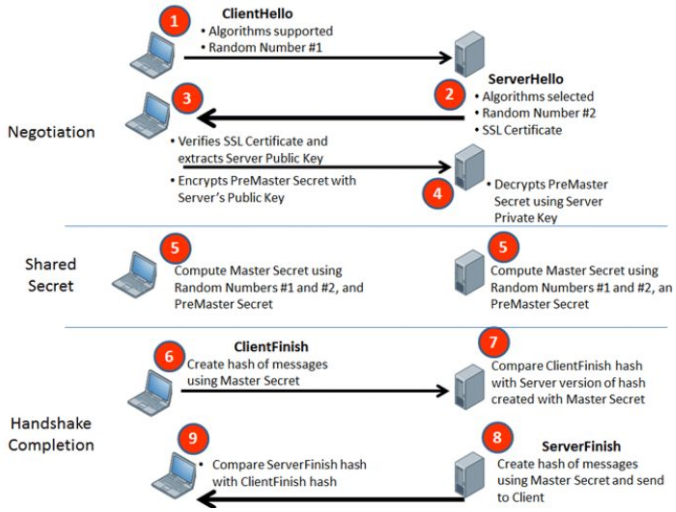
Enter Username

Password

Enter Password

Login

Here is a screenshot of the login form. Hoping that SSL would work I worked on an HTML login page. This doesn't work as SSL didn't cooperate and I did not know how to connect the information given by the user through the html to the server.
We all tried to implement SSL into our server and it ended up taking most of our time away.



(Google images)

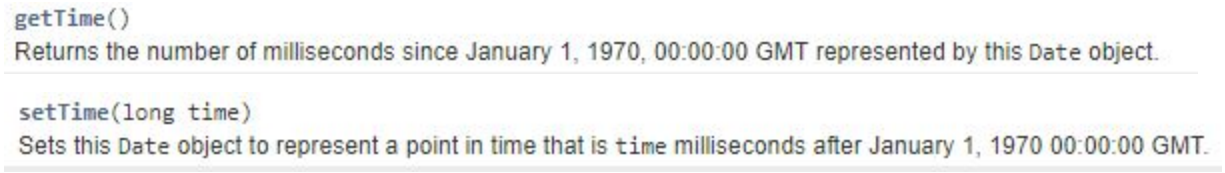
This is how SSL should work, the server sends the correct certificates to the client and the client sends back an encrypted key to let the server know it's connected. Then everything else is just communication. I still do not know why our implementation never worked. I spent a lot of time into trying to get SSL but overall it was not successful.

Nick Landry

Another part of the project was to implement an authorization and/or a authentication system for our web server. Our first thought was to create a whitelist or a blacklist for the server. A whitelist is a list of IP addresses that are allowed to access the web server. If a client's IP address does not match with the ones in the list, them the connection closes for that client. It acts as a VIP list for the web server. On the other hand, A blacklist doesn't restrict normal users from accessing the server. It does, however, block IP addresses of clients that try to do malicious things to it, such as a DDoS (distributed denial of service) attack. If the server detects that a client is spamming connections to it, the server will add the clients IP address to a blacklist, which will restrict it from accessing the server in the future.

For our web server, we decided to implement a blacklist. We initially thought to create a class called "Blacklist" and have our Server.java call methods from that class. It started out initializing an Array List which would store the blacklisted IP address'. We soon realized that we only needed one method from our created class, which was the method that checked if an IP address is in the blacklist, called "isBlacklisted". Instead of having a seperate blacklist class, we moved the method we needed over to Server.java and created an array list called "blacklist".

We also needed a way for the server to detect if it is being spammed by an IP, so it needed to be able to check the time of connection and which IP was the most recent to connect. We then needed a way to track the amount of time between connections. For this we implemented java.util.Date, which allows you to track the milliseconds that have passed since the Date was initialized. We only needed to use two methods from it:



We initialized a date at the beginning of the main method and set its time to 0 ms since the actual date didn't matter, we only cared about the amount of time that had passed. Once the date was set we could use getTime() to the server to get the milliseconds that have passed between connections once a client connects, and every time a client connected, the server would check the amount of time that has passed between connections. If the IP that connected was the same as the previous connection and connected within 10 milliseconds after the first connection, the server would close the socket for the client. It would then store the IP address and the connection time of the current client as the previous connection and add the IP to the blacklist.

```
1   import java.io.*;
2   import java.net.*;
3   import java.util.*;
4   import java.util.Date;
5   import java.util.concurrent.Executor;
6   import java.util.concurrent.Executors;
7   import java.util.logging.*;
8   import javax.net.ssl.*;
9   import javax.net.*;
10  import java.security.cert.*;
11  import java.security.KeyStore;
12  import java.security.KeyStoreException;
13  import java.security.NoSuchAlgorithmException;
14  import java.security.UnrecoverableKeyException;
15  import java.security.KeyManagementException;
16
17  public class Server{
18          private static ArrayList<InetAddress> blacklist = new ArrayList<InetAddress>()
19          private static long timeLast;
20          private static InetAddress prev;
21          static final int PORT = 8080;
22
23          public static boolean isBlackListed(InetAddress check){
24                  if(!blacklist.isEmpty()){
25                          if (blacklist.contains(check)){
26                                  return true;
27                          }
28                  }
29                  return false;
30          }
31
32          public static void main(String[] args) throws IOException{
33                  blacklist.add(InetAddress.getByName("www.myspace.com"));
34                  Date time = new Date();
35                  time.setTime(0);
36
```

In the screenshot you can see where we initialized the blacklist, and the long that will store the previous connection time and the InetAddress that will store the previous client IP address. The screenshot also includes the method that checks the blacklist if an IP already exists within it, and the beginning of the main method which initializes the date and sets the time to 0, which begins to count the milliseconds once it is set.

A problem that we encountered was that when the server went to check the blacklist for the very first time, it ran into a NullPointerException, meaning that the array

list was empty. To fix that, we entered an obscure InetAddress that will probably never access the server.

The rest of the code for our blacklist was included in our while loop that accepted connections from clients. Once the server socket accepts the client, the server immediately checks if the current client is in the blacklist. If it is, the socket closes, if it isn't, the server moves on to check if the client was the previous connection. If the client was the previous connection, it will only add the client to the blacklist if the client is spamming the server. We decided that if the same client tried to connect in under 10 ms that it was trying to spam the server. If the conditions apply, the current client will be added to the blacklist and the socket will close.

```
while(true){
        try{
                Socket client = serverSock.accept();
                System.out.println("HELLO");
                if (isBlackListed(client.getLocalAddress())){
                        client.close();
                }
                long past = timeLast;
                timeLast = time.getTime();
                long currentTime = time.getTime();
                if ((currentTime < (past+10) ) && ((client.getLocalAddress()) == prev)) {
                        blacklist.add(client.getLocalAddress());
                        prev = client.getLocalAddress();
                        client.close();
                }
                else{
                        prev = client.getLocalAddress();
                }

                        client.setSoTimeout(10000);
                        //client.setEnabledCipherSuites(factory.getSupportedCipherSuites());
                        HttpServer temp = new HttpServer(client,actLog,errLog);
                        service.execute(temp);


        }
        catch(SocketTimeoutException x){
                errLog.finer("Socket timed out: "+x);
        }
        catch(SSLHandshakeException z){
                errLog.finer("SSL Handshake: "+z);
        }
        catch(NullPointerException a){
                errLog.finer("NullPointerException: "+a);
        }
}
```

Here is a screenshot of the while loop. We added another catch just to catch the NullPointerException, just as a safeguard. We also added a print statement just to verify that the server executed the try catch block. In the screenshot you can see where the server uses the blacklist method to check the array list. Right after it, the server starts to compare the times of connections, which are used as longs.

Isaiah Plummer

Besides the basic functionality required of a web server compliant with Option 1's guidelines, such as basic HTTP functions, logging and multi-threading, our server also has several optional features included within it. The first optional function that was implemented into our server is a IP blacklist for handling unwanted connections. It is designed to prevent spam and is updated automatically and autonomously by the server. Although it can be manually changed, this requires hard coding at the moment. The second feature we added to the project was the very finicky Secure Sockets Layer which guarantees that the data transmitted between the server and client remains securely encrypted. In addition to these two features, our server has also been built with a Socket Timeout so that clients don't waste the web server's resources. Finally, our server also has additional HTTP functions aside from the required GET, POST, and HEAD methods. These include a TRACE function, which echos the input of the client, which is fairly useful for debugging, and a OPTIONS function. This returns the HTTP functions accepted by the web server.

The web server is written using only two files, the Server.java file which runs the web server and the httpServer.java file both holding classes of the same name. When the Server.java class is ran, two logger files are created for archiving up to date information on the servers activities. The first file serves to hold the general activity of the server while the other logs any errors the server may throw during its run time.

```
37                  //Set up the loggers
38                  Logger actLog = Logger.getLogger("activity");
39                  Logger errLog = Logger.getLogger("errors");
40                  actLog.setLevel(Level.FINER);
41                  errLog.setLevel(Level.FINER);
42
43                  //Handler files for log messages
44                  Handler act = new FileHandler("activity.log");
45                  Handler err = new FileHandler("error.log");
46                  act.setLevel(Level.FINER);
47                  err.setLevel(Level.FINER);
48
49                  //Join loggers and handler
50                  actLog.addHandler(act);
51                  errLog.addHandler(err);
```

In order to use SSL in our program a factory used to create them is first initialized and from that the server socket is created. Because our server is made to run on personal computers, we use a thread pool allowing the user to adjust the number of threads created for optimal performance. After all of its basic functions are initialized, the server runs passively waiting for a client to attempt to make a connect. Once a connection is made that client is set to a client socket. An important aspect to this socket is the timeout set to it. This ends the connection after a predetermined amount of time if it is not told to extend the length of that time. This is to prevent clients from

wasting the finite resources of the server by maintaining a inactive connection. From this point the client socket is sent to the httpServer class to handle the clients request.

```
169                 while(true){
170                         try{
171                                 Socket client = serverSock.accept();
172                                 System.out.println("HELLO");
173                                 if (isBlackListed(client.getLocalAddress())){
174                                         client.close();
175                                 }
176                                 long past = timeLast;
177                                 timeLast = time.getTime();
178                                 long currentTime = time.getTime();
179                                 if ((currentTime < (past+10) ) && ((client.getLocalAddress()) == prev)) {
180                                         blacklist.add(client.getLocalAddress());
181                                         prev = client.getLocalAddress();
182                                         client.close();
183                                 }
184                                 else{
185                                         prev = client.getLocalAddress();
186                                 }
187
```

The httpServer class in designed to handle the clients requests by maintaining a connection and reading the information sent from the client and sending a relevant response back to the client. The httpServer is able to achieve this by implementing the runnable class. This allows the class to be run in multiple instances, enabling more than one client to be served at a time. The first thing done by the class after being run from the server is to set static variables for some basic html files and a ROOT file where the server searches for the clients' requests. Other variables created are the two loggers and a boolean used to tell the server if the client's requested to maintain the connection.

```
24    public class HttpServer implements Runnable{
25          static final String DEFAULT_FILE = "index.html";
26          static final String FILE_404 = "404.html";
27          static final File ROOT = new File("root/");
28
29          private Socket client;
30          private static Logger actLog;
31          private static Logger errLog;
32          private static boolean connection = false;
34         public HttpServer(Socket cl, Logger act, Logger err){
35                 client = cl;
36                 actLog = act;
37                 errLog = err;
38          }
```

The constructor for the httpServer takes in three arguments: the client socket, and the two loggers. When executed the the httpserver method creates a new BufferedReader, PrintWriter and BufferedOutputStream. All further actions are handled in a try to block to catch exceptions that may occur. These I/O streams are then set to read and write to and from the client. Some basic variables are created to assist the method and the instance of the httpserver finally gets to work handling the client's requests. It begins parsing the data sent to it from the client by reading individual request lines for a HTTP request type and what file is requested.

```
49              try{
50
51                      //set up all IO connections for the server
52                      in = new BufferedReader(new InputStreamReader(connect.getInputStream()));
53                      out = new PrintWriter(connect.getOutputStream());
54                      fileOut = new BufferedOutputStream(connect.getOutputStream());
55
56                      //Initialize some variables for later use
57                      String fileRequested = null;
58                      String httpRequestType = null;
59                      int contentLength = -1;
60                      String contentType = "";
61                      String connectKeepAlive = "";
62
63                      actLog.finer("Finish I/O Connections");
64
65                      //Request Line Parsing
66                      String requestLine = in.readLine();
67                      System.out.println("Request Line: "+ requestLine);
68                      StringTokenizer requestLineTokenizer = new StringTokenizer(requestLine);
69                      httpRequestType = requestLineTokenizer.nextToken().toUpperCase();
70                      fileRequested = requestLineTokenizer.nextToken().toLowerCase();
71                      actLog.finer("Request Parsed");
```

From there the client loops through all the headers extracting key information from inside it. This includes the content type and length and also whether or not the client wishes to maintain its connection. Depending on what HTTP request type is sent by the client, different actions are carried out. GET and HEAD requests are handled under the same case because of their similarities. The only difference between the the two requests is that for a GET request the information stored in the server that is requested by the client is also sent to the client along with the standard HEAD return. In the case of the POST request, the data sent by the client is stored onto the server. The request is only properly handled if the data sent is in a multipart form and the client is sent a bad request for any information not in such a form. The TRACE request simply echos whatever is sent to it from the client and the OPTIONS request sends the client the allowed HTTP requests that can be handled by the server.

```
if(httpRequestType.equalsIgnoreCase("GET") || httpRequestType.equalsIgnoreCase("HEAD")){

        if(fileRequested.endsWith("/")){
                fileRequested += DEFAULT_FILE;
        }

        actLog.finer("File Requested Path:" + fileRequested);

        File file = new File(ROOT, fileRequested);
        int fileLength = (int) file.length();
        String fileType = getContentType(fileRequested);

        System.out.println(fileLength+"\t"+fileType);

        byte[] fileData = fileDataToBytes(file,fileLength);

        out.println("HTTP/1.1 200 OK");
        out.println("Server: TEST");
        out.println("Date: "+new Date());
        out.println("Content-type:" + fileType);
        out.println("Content-length: "+fileLength);
        out.print("\r\n\r\n");
        //out.println();
        out.flush();
```

The last thing done by the server is catching any exceptions thrown by the server during its run time and handling and logging them to the best of its ability. After this the client checks to see if the client still wants to maintain a connection as determined by reading the data sent. If not, the server will close all connects to the server.

```
catch(FileNotFoundException z){
        errLog.finer("File Not Found Exception: "+z);
        try{fileNotFound(out,fileOut);}
        catch(IOException a){}
}
catch(IOException x){errLog.finer("IOException: " + x);}
catch(NullPointerException y){errLog.finer("NullPointerException: "+y);}
catch(Exception e){errLog.finer("Exception: "+ e);}
finally{
        try {
                if(connection == false) {
                        in.close();
                        out.close();
                        fileOut.close();
                        connect.close();
                        System.out.println("Closed connection");
                }
                else if(connection == true) {
                        connect.setKeepAlive(true);
                        System.out.println("Client keep alive: " + connect.getKeepAlive());
                        connect.setSoTimeout(5000);
                        System.out.println("So_TimeOut set: " + connect.getSoTimeout());
                }
        }
        catch(IOException x){errLog.finer("IOException: "+x);}
}
```