16308185 Shaun Jose

CS3031 Documentation - Project 1

Introduction

The program I created consists of 5 files and contains the following main features:

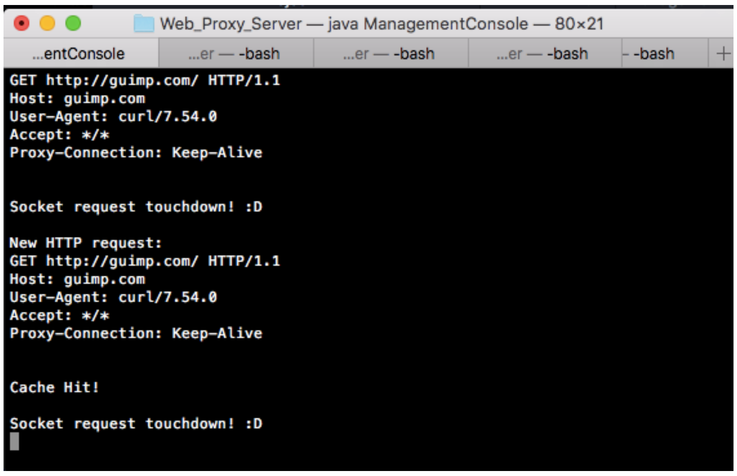
1. Responds to HTTP request and displays the requests on the Management Console
2. Responds to HTTP request and displays the requests on the Management Console
3. Responding to requests sent via the web browser (I used Firefox).
4. Handles Web socket connections (similar to https request handling - stay tuned)
5. URL blocking with saving to and retrieving from a file (don't need to block URLs repeatedly on program startup)
6. Cache with saving to and retrieving from a cache text file
7. Lastly, handling of each of these requests, and blocked URLs, are multithreaded, and can be dealt with simultaneously

All these features are implemented in different parts and files of the program. In this document, I will explain the high level details of my program, file by file, and how the different features are implemented in the relevant files. I will start with ManagementConsole.java, which seems to be the starting point of my entire program, and go down in an organised manner to the other files in a hierarchical manner.
If needed, the code of all the files, along with the README (containing instructions on how to run the program), can be found at the end of this document.
Please move on to the next page to find the explanation of these files and features.

ManagementConsole.java

This is the starting point of the my program, although not the heart of my program. This file is mainly concerned with 3 aspects - file blocking/unblocking, cache managing, and starting off/shutting down the web proxy server.
This file starts with initialising the cache. This means it creates a cache.txt file if such a file doesn't exist. However, if a file cache.txt does exist, it reads the cached URLs from the file, saving it to an ArrayList called cachedURLs, and the corresponding responses for the URLs, saving it in the corresponding indexes of another ArrayList called cachedResponses. Then, it initialises the blockedURLs in a similar manner, i.e. creating the file if it doesn't exist or reading the blocked URLs into the the blockedURLs HashSet if the file does exist. After initialising the blockedURL set and the cache lists, the program spits out instructions to the user, for blocking/unblocking/listing blocked sites or even exiting the web management console, in which case the web proxy server shuts down.

<u>How does the cache work?</u>
As mentioned above, I maintain 2 array lists for the cache - cachedURLs and cachedResponses (for those URLs). Since the cache is always limited, I set a variable CACHE_LIMIT to 2, for easy testing purposes. A limited cache size also implies that we need a replacement policy, Therefore, I have adopted the LRU (Least recently used) policy for my cache, with the oldest url and response in the beginning of the array at index 0, and the most recently used one at the end of the array list.
Every time a URL is accessed, it, along with it's response, is saved at the end of the array lists, while duplicate entries of the url and response are deleted, thus making this URL the most recently used URL. If the cache limit is reached, I discard the oldest URL and response, which are present at index 0 of the array list.
These array lists are saved to a file before the program ends, and are retrieved once again at the start of the program, and that's the simple implementation of my cache :)



```
GET http://guimp.com/ HTTP/1.1
Host: guimp.com
User-Agent: curl/7.54.0
Accept: */*
Proxy-Connection: Keep-Alive


Socket request touchdown! :D

New HTTP request:
GET http://guimp.com/ HTTP/1.1
Host: guimp.com
User-Agent: curl/7.54.0
Accept: */*
Proxy-Connection: Keep-Alive


Cache Hit!

Socket request touchdown! :D
```

Cache Example

How does blocking work?

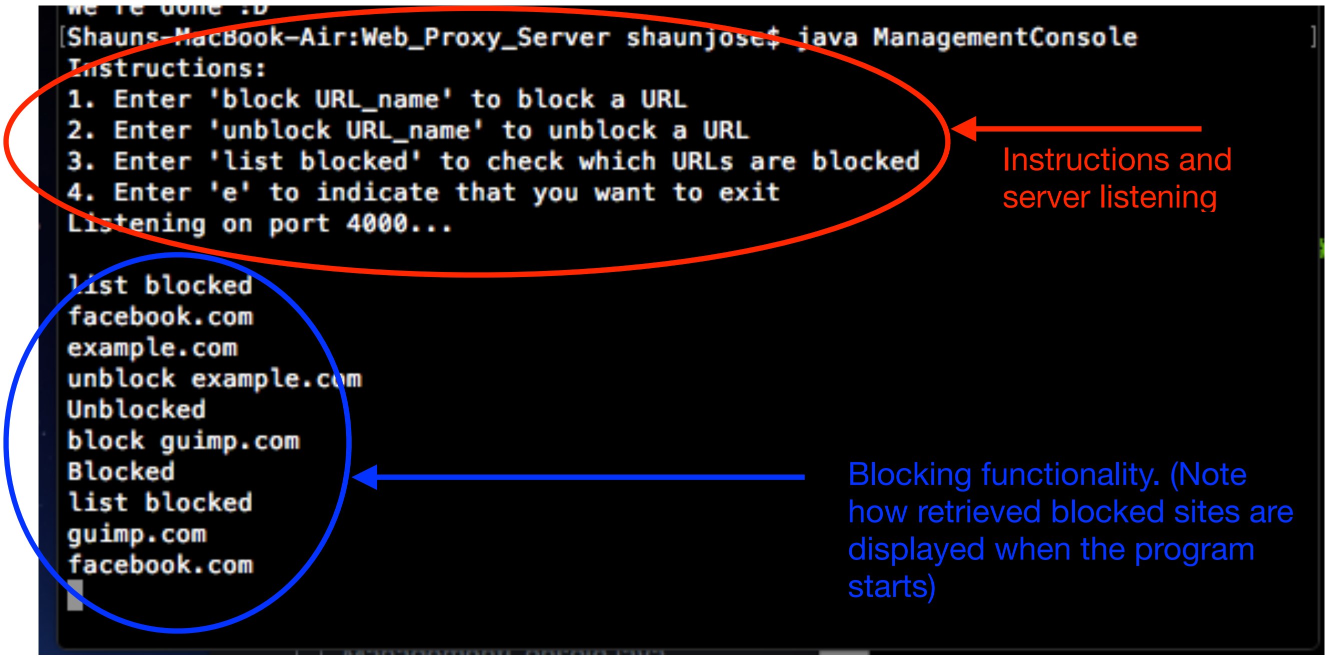
The following 4 lines of instructions are given to the user:

1. Enter 'block URL_name' to block a URL
2. Enter 'unblock URL_name' to unblock a URL
3. Enter 'list blocked' to check which URLs are blocked
4. Enter 'e' to indicate that you want to exit

If the user follows these instructions properly, the program either adds a blocked URL if it's not already blocked, unblocks a URL if it exists in the set of blocked URLs, lists all the blocked URLs or exits the program (with an 'e'). Do note that, the validity of URLs, is checked before the Management program proceeds to block or unblock the specified URL. I use a HashSet for this as it is a very swift data structure, and no sense of ordering is needed as there is no form of replacement policy and no reason for ordering.
Similar to the cache part, right before the program ends, the blocked URLs are saved to a file, to be retrieved another time the program is started up.

How does Web Proxy startup?

Although small and simple enough, the way I handle this is crucial to how the program works. After the cache lists and blocked set are initialised, my program creates a web proxy object, and runs it on a thread. Then only does the while loop start, accepting the user input and following the user's commands if they do make any sense to the program. It is crucial that I start off the Web proxy server on a different thread, as that gives the program the capability to listen on port 4000 (that is the web proxy server listening for requests) as well as block/unblock/listBlockedSites on the same console, simultaneously. So maybe after a request or even during a request, the "blocked sites set" can change.



```
We're done :D
Shauns-MacBook-Air:Web_Proxy_Server shaunjose$ java ManagementConsole
Instructions:
1. Enter 'block URL_name' to block a URL
2. Enter 'unblock URL_name' to unblock a URL
3. Enter 'list blocked' to check which URLs are blocked
4. Enter 'e' to indicate that you want to exit
Listening on port 4000...

list blocked
facebook.com
example.com
unblock example.com
Unblocked
block guimp.com
Blocked
list blocked
guimp.com
facebook.com
```

Instructions and server listening
Blocking functionality. (Note how retrieved blocked sites are displayed when the program starts)

Note that the cache functions are in the ManagementConsole.java file, but are called from another file which handles the client request - (stay tuned).
Also note that I realised that it makes sense to block hosts, instead of URLs, later in the program and changed the functionality to block hosts, as shown in the image above

In summary, this file does the following functions:

1. Initialise cache and blocked URLs from files if they exist, or create files if they don't exist.
2. Start up web proxy server on a different thread.
3. Run a while loop where user's commands for blocking/unblocking/listing sites are accepted (this runs on the main thread)
4. On the user's command to quit the web proxy - the program saves the cache contents to the cache.txt file, blocked URLs to the blocked.txt file, tells the Web proxy to shutDown using a method existing in that file, and then joins the web proxy thread to end multithreading, along with the entire program.

We can now move on to the next file in hierarchy, that is WebProxy.java

WebProxy.java

From the previous section, we know that this file runs on a thread. Note that the moment an object of this class is created, a serverSocket is initialised with the port number 4000, and this socket is what I will be referring to as the “base socket”.
A thread always starts at a function run, in Java, and since this file is run on a thread, it contains the function run which just calls the function start_listening.
start_listening is simple. All it does it accept a socket through which a client makes a request, via the base socket, and sends that socket to another function handleRequest. It does this in a while loop until the user wants to quit or shut down the server
handleRequest takes the clientSocket, creates a RequestHandler object passing in the clientSocket, and starts it off on another thread! It then saves that thread to a HashSet of requestThreads.
Taking a pit stop here… every time a client makes a request, a new request handler object is run on a thread for that corresponding client socket, and the thread is saved to a HashSet. Eventually, this program will have started multiple RequestHandler threads and saved all these threads in the requestThreads set.
The last part of this file deals with shutting down the socket. It is here that I reveal why the request threads are saved at all. In this shutDown function, three things happen. We wait for all the request threads saved in the set to end, by checking if they’re alive or not. Then, a boolean is changed so the base socket can stop listening for more client requests, and finally, the base socket is closed. Recall that this function is called from ManagementConsole.java only after the user enters the quit command (which is ‘e’).

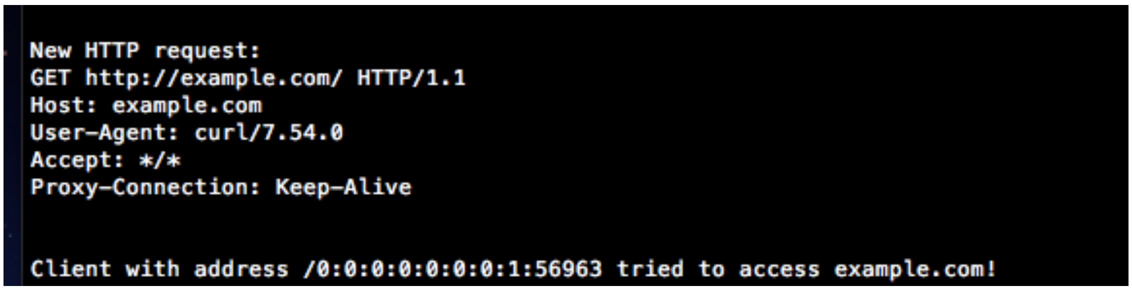
In summary, this file carries on with it’s work in the following way:

1. Open a base socket when object created
2. Start listening for client requests, and retrieve socket connections to those clients
3. Create RequestHandler threads corresponding to each client socket
4. shutDown all the threads, stop listening and close base socket when user wants to quit.

This file might have been simple enough, but as a head’s up, the next file is the largest and most complex file, and I consider that to be the heart of the program, as it is that file which handles all the requests appropriately. So lets move on to RequestHandler.java.

RequestHandler.java

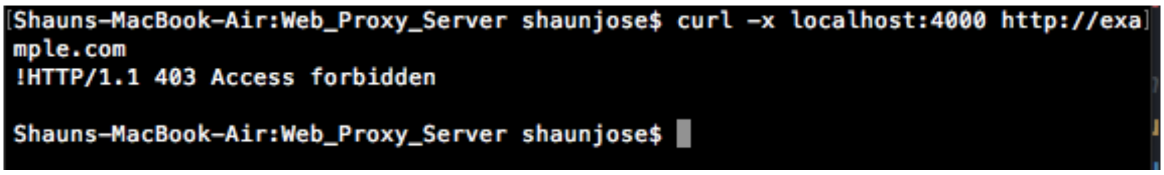
When the WebProxy creates an object of this class, three variables are initialised - the clientSocket (passed as parameter), and the request type (set to HTTP by default) and port number (set to 80 by default). The last two variables remain unchanged if the request is an HTTP request (explained later in this section)
Since this file is run on a thread for every request, the run method is called, which in this case, calls the processRequest method. This method calls another method (getMethodHostReq), to get the method, hostName and entire request from the client via the input stream of the client socket, by receiving a String array from this function. Note that this function handles both, HTTP and HTTPS requests. If it's an HTTPS request, the method would always be replaced by the word "CONNECT", and so if that's the method, we know that we've received an HTTPS request. In case of an HTTP request, we just acquire the method, hostName, and entire request, whereas for the HTTPS request, we not only acquire all of these, but receive and change the port number as well (which is right after the hostname, in the request) and change the type of the request from HTTP (the default) to HTTPS.
The entire request is received in two parts - the header (which ends with a \r\n\r\n), and the body (which, if exists, ends with an </html> tag), as the body is in html format.
The method processRequest then displays the HTTP or HTTPS request, and checks if the host is blocked.
This method then goes on to check if the host is blocked, and in that case, sends an HTTP 403 forbidden response. It also informs the user at the management console, which client (with address and port number) tried to access a forbidden site, as shown below:



```
New HTTP request:
GET http://example.com/ HTTP/1.1
Host: example.com
User-Agent: curl/7.54.0
Accept: */*
Proxy-Connection: Keep-Alive


Client with address /0:0:0:0:0:0:0:1:56963 tried to access example.com!
```

Management Console



```
Shauns-MacBook-Air:Web_Proxy_Server shaunjose$ curl -x localhost:4000 http://exa
mple.com
!HTTP/1.1 403 Access forbidden

Shauns-MacBook-Air:Web_Proxy_Server shaunjose$
```

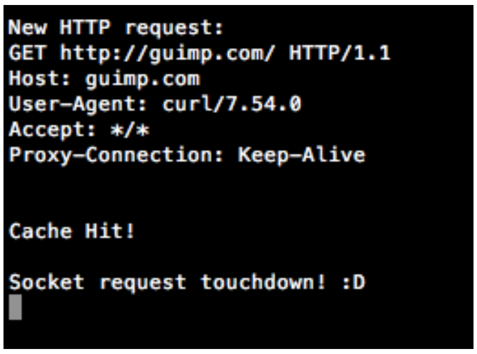
Client Side

If not blocked, the method checks if the request was cached, and this can only happen for an HTTP request in my program (as an HTTPS request is a fully duplexed two way connection - explained later in this section). If received, the method outputs “Cache Hit”, as shown in the very first image of this document, and retrieves the response from the cache, using one of the methods in ManagementConsole.java.

Recall that the hostName was received and the port number was set to default in case of HTTP, or actually changed to the real port number in case of HTTPS. In both cases, these variables contain appropriate values and so the serverSocket gets initialised with these two pieces of information. Again the program checks whether the request was an HTTP request and if no response was received from the cache, the method forwards the HTTP request to the server via the server socket’s output stream, in order to receive the response from the server. In case of HTTPS, there is no need to forward anything to the server, and just the server socket is opened.

Now, this is where the program diverges, to handle HTTP and HTTPS differently, as the program now has to deal with establishing connections, and getting appropriate responses.

In case of an HTTP request, the client gets a response from the input stream of the server, by getting header first and then the body (if the response is not already received from the cache). The method then opens an output stream to the client socket and fires off the response to it. A working example of an HTTP request-response is shown below

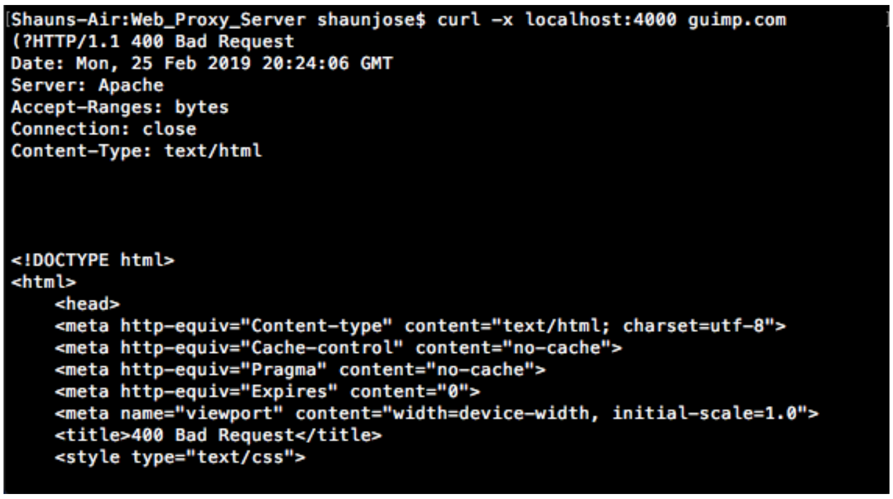


```
New HTTP request:
GET http://guimp.com/ HTTP/1.1
Host: guimp.com
User-Agent: curl/7.54.0
Accept: */*
Proxy-Connection: Keep-Alive


Cache Hit!

Socket request touchdown! :D
```

Management Console
(request happens to lead to
a cache hit)



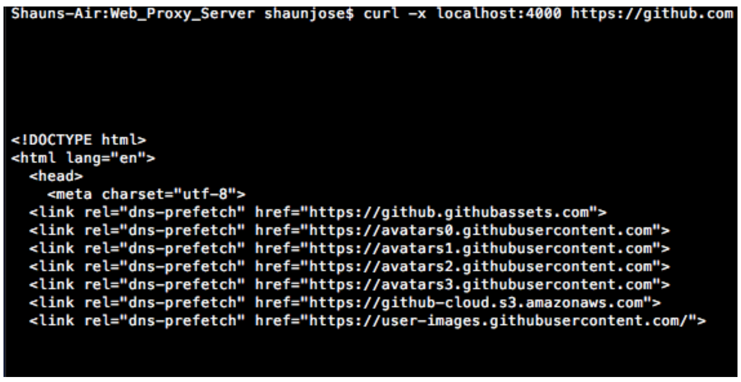
```
Shauns-Air:Web_Proxy_Server shaunjose$ curl -x localhost:4000 guimp.com
(?HTTP/1.1 400 Bad Request
Date: Mon, 25 Feb 2019 20:24:06 GMT
Server: Apache
Accept-Ranges: bytes
Connection: close
Content-Type: text/html



<!DOCTYPE html>
<html>
    <head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <meta http-equiv="Cache-control" content="no-cache">
    <meta http-equiv="Pragma" content="no-cache">
    <meta http-equiv="Expires" content="0">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>400 Bad Request</title>
    <style type="text/css">
```

Client Side
(with just part of the response
shown)

In case of HTTPS requests, an HTTP 200 Connection established response is sent over to the client, indicating that both the server and client sockets are opened and connected. This method then gets interesting here. It creates a ClientListenHTTPS object (class explained later), which basically listens to the client for messages and sends it to the server. It also creates a ServerClientListenHTTPS object (class explained later), which listens to the server socket input stream and sends any incoming messages to the client socket output stream. The ClientListenHTTPS object is run on another thread, and the current thread is used to listen to the server (which the ServerListenHTTPS object is run on). Therefore, these two threads are working simultaneously in order to forward messages from one side to the other, whenever they need to, and the connection is always open until they both are done. This fully duplex two way connection establishment is how I deal with HTTPS requests, but this is the principle of Web Sockets, and so, I have tackled HTTPS requests using the idea of Web Sockets in my program. A working example of an HTTPS request and response is shown below:

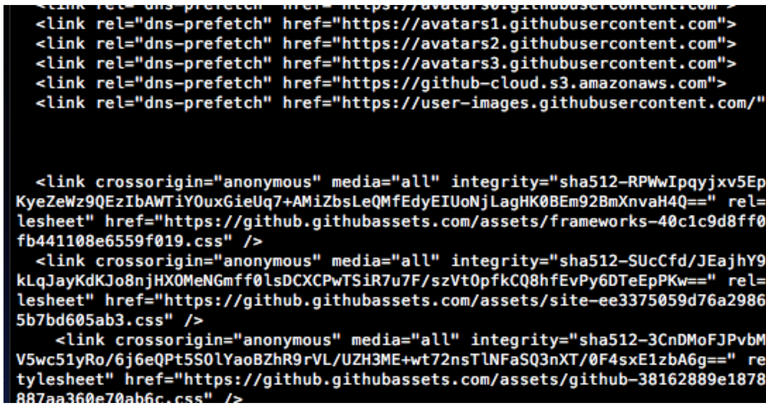


```
Shauns-Air:Web_Proxy_Server shaunjose$ curl -x localhost:4000 https://github.com



<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
  <link rel="dns-prefetch" href="https://github.githubassets.com">
  <link rel="dns-prefetch" href="https://avatars0.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars1.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars2.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars3.githubusercontent.com">
  <link rel="dns-prefetch" href="https://github-cloud.s3.amazonaws.com">
  <link rel="dns-prefetch" href="https://user-images.githubusercontent.com/">
```

Response part 1

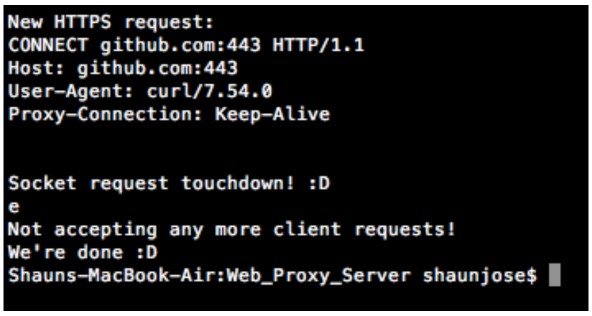


```
  <link rel="dns-prefetch" href="https://avatars1.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars2.githubusercontent.com">
  <link rel="dns-prefetch" href="https://avatars3.githubusercontent.com">
  <link rel="dns-prefetch" href="https://github-cloud.s3.amazonaws.com">
  <link rel="dns-prefetch" href="https://user-images.githubusercontent.com/"



  <link crossorigin="anonymous" media="all" integrity="sha512-RPWwIpqyjxv5Ep
KyeZeWz9QEzIbAWTiYOuxGieUq7+AMiZbsLeQMfEdyEIUoNjLagHK0BEm92BmXnvaH4Q==" rel=
lesheet" href="https://github.githubassets.com/assets/frameworks-40c1c9d8ff0
fb441108e6559f019.css" />
  <link crossorigin="anonymous" media="all" integrity="sha512-SUcCfd/JEajhY9
kLqJayKdKJo8njHXOMeNGmff0lsDCXCPwTSiR7u7F/szVtOpfkCQ8hfEvPy6DTeEpPKw==" rel=
lesheet" href="https://github.githubassets.com/assets/site-ee3375059d76a2986
5b7bd605ab3.css" />
    <link crossorigin="anonymous" media="all" integrity="sha512-3CnDMoFJPvbM
V5wc51yRo/6j6eQPt5SOlYaoBZhR9rVL/UZH3ME+wt72nsTlNFaSQ3nXT/0F4sxE1zbA6g==" re
tylesheet" href="https://github.githubassets.com/assets/github-38162889e1878
887aa360e70ab6c.css" />
```

Continuation of response



```
New HTTPS request:
CONNECT github.com:443 HTTP/1.1
Host: github.com:443
User-Agent: curl/7.54.0
Proxy-Connection: Keep-Alive


Socket request touchdown! :D
e
Not accepting any more client requests!
We're done :D
Shauns-MacBook-Air:Web_Proxy_Server shaunjose$
```

Request shown on management console

Finally, the program shuts down the client and server sockets, and prints out a “Socket request touchdown :D” message (as you might have noticed), to inform that the request has been completely dealt with and that the request thread has ended.

That’s it from RequestHandler,java! We can now move on to the last two simplest files of the program

ServerListenHTTPS.java

This file simply deals with listening to the server and sending responses to the client, in case of an HTTPS request. The moment an object is created, 3 things are initialised -> the server socket input stream, the client socket output stream and the client thread (which is running ClientListenHTTPS).
The function listenAndSend is called from the requestHandler class, with the same thread, and all this function does is takes the response from the server and dumps it onto the client until the server has nothing more to send. The function then uses the clientThread, and waits in the function for as long as this thread is alive, to ensure that the connection is open until the client is done sending messages as well.

ClientListenHTTPS.java

This file is just like the one above, but instead, it listens to the client and sends requests to the server continuously, segment by segment. Thus, when the object is created, an input stream to the server and an output stream from the client socket are initialised.
When the thread for this program starts, run is called, which calls the listenAndSend method, which listens for client messages and fires it off to the server, until the client is done sending messages.
Notice how there is no server thread here! This thread ends when the client is done sending messages, and doesn't wait for the server thread to finish. There are two reasons for this:

1. It's not needed! When this thread dies out or ends, the request handler doesn't just close the sockets if the server is running, as the server is using the Request handler's thread after all, and cant return to request handler to close the sockets until the server is done sending messages!
2. Secondly, imagine a scenario where the server thread waits for the client thread to finish, and vice versa. This is a classic example of non-termination, as both threads will be waiting for each other to end first, before it can allow itself to end, which will never happen.

This is everything about my program, and I shall follow up with a few limitations, followed by the code.

Limitations

Although my program works for the most part, there are a few holes which can break my program, and a lot of space for improvement. Here are some of the limitations:

1. While handling the URL blocked list, I only check if a URL is valid, not verified. For example, bhbhsbkfsef.com will be considered as a URL, even though it's not, but something like "hello world" will not be accepted.
2. The first limitation might seem small, but this point is a huge limitation. In my program, I cannot differentiate between the sites example.com and www.example.com . I thought of adding "www." at the start of every url if it doesn't contain the substring, but not all URLs start with "www." (for example scss.tcd.ie). This means, if one blocks facebook.com , people will be allowed to access facebook via www.facebook.com and people who try to access facebook via facebook.com will be blocked. Therefore, both versions of the url need to be added to the blocked list, in order to completely block off a site
3. The previous point leads to a bigger problem, caching of responses. Imagine caching a response for github.com and then caching another response for www.github.com . In the worst case, twice as much cache space will be used than in normal circumstances, and I consider this to be the most significant limitation in my program.
4. The cache could be improved, by using last modified dates and saving images instead of just text. Because of no use of time and date or bandwidth, my cache implementation is somewhat inefficient and could even lead to displaying an incorrect response to the user if the page has been modified since the response has been cached.
5. Once a URL which is in the cache is also blocked, I do not remove it from the cache, but I leave it in just in case the URL gets unblocked. If it remains to be blocked, I would expect that response to eventually get trashed from the cache by the LRU replacement policy
6. Lastly, my program doesn't allow clients to post data with GET requests, as when the method is "GET", I don't even check if the request has a body. Although I am stating this in the limitations section, I consider this to be a good act, as it strictly makes the client use POST to post stuff and not GET, so as to ensure encryption of data while posting

Please refer to the next pages for the code

“Just Show Me The Code”

ManagementConsole.java

```
/* author: Shaun Jose
   github: github.com/ShaunJose
   Class Description: Maintains a blocked list, cache and listens to the manager's requests
to block urls, list blocked urls or shut down the proxy
*/

//imports
import java.net.URL;
import java.util.HashSet;
import java.util.Scanner;
import java.lang.Thread;
import java.util.ArrayList;
import java.io.File;
import java.io.PrintWriter;
import java.util.Iterator;

class ManagementConsole
{
  //constants
  private static final int DEFAULT_PORT = 4000;
  private static final int CACHE_LIMIT = 2;
  private static final String CACHE_FILE = "cache.txt";
  private static final String BLOCKED_FILE = "blocked.txt";
  private static final String FILE_DELIMITER = "---***^***^^^^^^***^***---";

  //static variables for the proxy (makes sense to have only 1 proxy)
  private static HashSet<String> blockedURLs = new HashSet<String>();
  private static ArrayList<String> cachedURLs = new ArrayList<String>();
  private static ArrayList<String> cachedResponses = new ArrayList<String>();
```

```
 /**
  * Program initiation method. Accepts requests to block certain servers, and runs the
web proxy server on another thread
  *
  * @param args: This program doesn't care about args, no input needed
  *
  * @return: None
  */
 public static void main(String[] args) // keeping it nice and small :)
 {
   //initialise the cache
   initCache();

   //initialises the set of blocked URLs
   initBlockedSet();

   //start managing the server and blocked lists
   start_managing();

   System.out.println("We're done :D");
 }


 /**
  * Create the cache file if it doesnt exist. If exists, call readFromcache to read cache
contents into the cache variables
  *
  * @return: None
  */
 private static void initCache()
 {
   try
   {
     File cache = new File(CACHE_FILE); //cache path
     if(cache.exists()) //if cache exists, read from file
```

```
    {
      readFromCache(cache);
    }
    else //if it doesn't exist create new empty cache file
    {
      cache.createNewFile();
    }
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }

}



/**
 * Reads from filepath into cache variables, to initialise cache vars
 *
 * @param cache: Filepath to cache
 *
 * @return: None
 */
private static void readFromCache(File cache)
{
  //initialise vars used for this task in loop
  Scanner sc;
  try
  {sc = new Scanner(cache);} catch(Exception e){ e.printStackTrace();return; }
  String line = "";

  //read urls and responses into arrays
  while(sc.hasNext())
  {
    //read url into url list
```

```
      line = sc.nextLine();
      cachedURLs.add(line);

      //read response into response list
      String response = "";
      line = ""; //reset line
      do
      {
        response += line;
        line = sc.nextLine() + "\r\n"; //nextLine removes the much needed \r\n
      } while (!line.equals(FILE_DELIMITER + "\r\n"));
      cachedResponses.add(response);

    }
  }



  /**
   * Writes all cache contents into the file, follwing the format rules
   *
   * @return: None
   */
  private static void saveCache()
  {
    //initialise variables used to write cache
    PrintWriter cacheWriter = null;
    try
    { cacheWriter = new PrintWriter(CACHE_FILE, "UTF-8"); }
    catch(Exception e) { e.printStackTrace(); return; }

    //save url's and their cahced responses into the cache file
    for(int i = 0; i < cachedURLs.size(); i++)
    {
      //save url
      String url = cachedURLs.get(i);
      cacheWriter.println(url);

      //save response
      String response = cachedResponses.get(i);
      cacheWriter.print(response);

      //delimiter!
      cacheWriter.println(FILE_DELIMITER);
    }

    cacheWriter.close();

  }



  /**
   * Create the bockedURL-keeper file if it doesnt exist. If exists, call readFromBlocked to
read blockedURLS into the HashSet
   *
   * @return: None
   */
  private static void initBlockedSet()
  {
    try
    {
      File blocked = new File(BLOCKED_FILE); //file path
      if(blocked.exists()) //if file exists, read from file
      {
        readFromBlocked(blocked);
      }
      else //if it doesn't exist create new empty blocked file
      {
        blocked.createNewFile();
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }



  /**
   * Reads from filepath into blockedURL HashSet, line by line
   *
   * @param blocked: Filepath to the blockURLs-keeper file
   *
   * @return: None
   */
  private static void readFromBlocked(File blocked)
  {
    //initialise vars used for this task in loop
    Scanner sc;
    try
    { sc = new Scanner(blocked);} catch(Exception e){ e.printStackTrace();return; }

    //read urls line by line
    while(sc.hasNextLine())
    {
      String blockedURL = sc.nextLine();
      blockedURLs.add(blockedURL);
    }

  }



  /**
   * Save all the blockedURLs in the relevant file, line by line
   *
   * @return: None
```

```
   */
  private static void saveBlockedSet()
  {
    //initialise variables used to write cache
    PrintWriter blockedWriter = null;
    try
    { blockedWriter = new PrintWriter(BLOCKED_FILE, "UTF-8"); }
    catch(Exception e) { e.printStackTrace(); return; }
    Iterator blockedIter = blockedURLs.iterator();

    //save blocked urls to file
    while(blockedIter.hasNext())
    {
      String url = (String) blockedIter.next();
      blockedWriter.println(url);
    }

    blockedWriter.close();
  }


  /**
   * Manages blockedUrls set, stores it in a file, and calls the        start_listening() method
to start running a server
   *
   * @return: None
   */
  private static void start_managing()
  {
    //Instructions for managing urls
    System.out.println("Instructions:");
    System.out.println("1. Enter 'block URL_name' to block a URL");
    System.out.println("2. Enter 'unblock URL_name' to unblock a URL");
    System.out.println("3. Enter 'list blocked' to check which URLs are blocked");
    System.out.println("4. Enter 'e' to indicate that you want to exit");
```

```
//make Web proxy server run on another thread
WebProxy proxy = new WebProxy(DEFAULT_PORT);
Thread proxyThread = new Thread(proxy);
proxyThread.start();

//create variables needed for block url processing
Scanner sc = new Scanner(System.in);
boolean addingBlockedSites = true;

//Accepting blocked lists
while(addingBlockedSites)
{
  //get input in lower case, so as to not add duplicates just because of case difference
  String input = sc.nextLine().toLowerCase();

  //check input and act accordingly
  //Case 1: exit
  if(input.equals("e"))
    addingBlockedSites = false;

  //Case 2: list the blocked urls
  else if(input.equals("list blocked"))
    displayHashSet(blockedURLs);

  //Case 3: Blocking a url
  else if(input.length() > 6 && input.substring(0, 6).equals("block "))
  {
    //get url part of input
    input = input.substring(6);

    input = formatURL(input);

    //if URL is invalid
    if(!isValidUrl(input))
```

```
      System.out.println("This URL is invalid");
    //if URL already blocked
    else if(blocked(input))
      System.out.println("This URL has already been blocked.");
    //block the URL
    else
    { //hostName and not entire url
      blockedURLs.add(input.substring(input.indexOf("//") + 2));
      System.out.println("Blocked");
    }
  }

  //Case 4: Unblocking a url
  else if(input.length() > 8 && input.substring(0, 8).equals("unblock "))
  {
    //get url part of input
    input = formatURL(input.substring(8));

    //if URL is invalid
    if(!isValidUrl(input))
      System.out.println("This URL is invalid");
    //if URL is not blocked
    else if(!blocked(input))
      System.out.println("This URL is not blocked.");
    //block the URL
    else
    { //hostName and not entire url
      blockedURLs.remove(input.substring(input.indexOf("//") + 2));
      System.out.println("Unblocked");
    }
  }
  //Case 5: Invalid input
  else
    System.out.println("Sorry, I didn't get that");
    }

  //save cache, blockedURLs and shut down the web proxy completely
  saveCache();
  saveBlockedSet();
  proxy.shutDown();

  //end the thread
  try
  {
    proxyThread.join();
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }

}



/**
 * Displays a String type HashSet
 *
 * @param set: The set who's elements need to be displayed
 *
 * @return: None
 */
public static void displayHashSet(HashSet<String> set)
{
  //standard null check
  if(set == null)
    return;

  //Iterate over String elements and print them
  for(String element : set)
```

```
    System.out.println(element);
  }



  /**
   * Checks if the given url string is valid
   *
   * @param url: The String url whose validity has to be checked
   *
   * @return: true indicating URL is valid, false otherwise
   */
  private static boolean isValidUrl(String url)
  {

    try
    {
      new URL(url).toURI(); //exception occurs here = BAD url
      return true;
    }
    catch(Exception e)
    {
      return false;
    }
  }

  /**
   * Adds http or https to the String passed if it doesnt exist
   *
   * @param url: The url that has to be formatted (type String)
   *
   * @return: The formatted url
   */
  private static String formatURL(String url)
  {
    //add http:// if needed
    if(!url.contains("http://") && !url.contains("https://"))
      url = "http://" + url; //might not be secure, so only http

    return url;
  }

  /**
   * Checks if a url has been cached and return true or false indicating whether it exists or
not
   *
   * @param url: The url that you want to check has been cached
   *
   * @return: true if url is cached, else false
   */
  public static boolean isCached(String url)
  {
    url = formatURL(url);

    return cachedURLs.contains(url);
  }

  /**
   * Gets a response for a cached url from the cache. Null if not cached
   *
   * @param url: The url whose response is needed
   *
   * @return: Cached response fro url, or null if doesnt exist in cache
   */
  public static String getFromCache(String url)
  {
    url = formatURL(url);

    if(!isCached(url)) //if not in cache, abort and return null
      return null;
    int index = cachedURLs.indexOf(url);
    return cachedResponses.get(index);
  }



  /**
   * Saves url and response to cache, as most recently added cache. Recursive function
   *
   * @param url: The url whose repsonse needs to be cached
   * @param response: The response which needs to be cached
   *
   * @return: None
   */
  public static void saveToCache(String url, String response)
  {
    url = formatURL(url);

    if(isCached(url)) //if already cached, replace as Newest elements!
    {
      int index = cachedURLs.indexOf(url);
      removeFromCache(index);
      saveToCache(url, response); //recursively save it
      return; //end
    }

    if(cachedURLs.size() >= CACHE_LIMIT) //if cache is full, remove LRUs
    {
      removeFromCache(0); //remove least recently used element
      saveToCache(url, response); //recursively add
      return; //end
    }

    //reaches here if not duplicate and cache not full
    cachedURLs.add(url);
    cachedResponses.add(response);
   }

  }

   /**
    * Delete cahced element at index from cached urls and cahced responses
    *
    * @param index: index at which elements need to be deleted
    *
    * @return: None
    */
   private static void removeFromCache(int index)
   {
    if(index < CACHE_LIMIT) //private arr, so trusting that limit isn't crossed
    {
     cachedURLs.remove(index);
     cachedResponses.remove(index);
    }
   }


   /**
    * Checks if a site is blocked or not
    *
    * @param url: The url whose blockage being checked
    *
    * @return: True if site is blocked, else false
    */
   public static boolean blocked(String url)
   {
    url = formatURL(url); //to ensure that is has http:// or https://

    //concentrate on hostname, not URL
    return blockedURLs.contains(url.substring(url.indexOf("//") + 2));
   }
  }
```

WebProxy.java

```
/* author: Shaun Jose
   github: github.com/ShaunJose
   Class Description: Creates a new thread for all client requests, and handles the base
socket
*/

//imports
import java.net.ServerSocket;
import java.net.Socket;
import java.util.HashSet;

public class WebProxy implements Runnable
{
  //class variables
  private int port;
  private HashSet<Thread> requestThreads;
  private boolean open;
  private ServerSocket serverSocket;

  /**
   * Creates a web proxy object. Intiialises the class variables
   *
   * @param port: The port at which you want the WebProxy  server to listen
   */
  WebProxy(int port)
  {
    this.port = port;
    requestThreads = new HashSet<Thread>();
    this.open = true; //server is open

    try
    { serverSocket = new ServerSocket(this.port); }
    catch(Exception e)
  { System.out.println("Couldn't create server socket for proxy"); }
 }

 /**
  * Open a Socket and start listening on port 4000
  *
  * @return: None
  */
 private void start_listening()
 {
   try
   {
     System.out.println("Listening on port " + this.port + "...\n");

     do
     {
       handleRequest(serverSocket.accept()); //send in sockets to be handled
     } while(this.open);
   }

   catch(Exception e)
   {
     System.out.println("Not accepting any more client requests!");
   }

 }


 /**
  * Handles a request by making reqHandler object run as another thread
  *
  * @param clientSocket: The socket for the relevant request
  *
  * @return: None
  */
```

```
private void handleRequest(Socket clientSocket)
{
  //Create request handler object and make it run on a thread
  RequestHandler reqHandler = new RequestHandler(clientSocket);
  Thread reqThread = new Thread(reqHandler);
  reqThread.start(); //start the thread

  //save the thread here
  requestThreads.add(reqThread);
}



 /**
  * Waits for all request threads to end, then closes the base proxy socket
  *
  * @return: None
  */
 public void shutDown()
 {
   //waiting for all the request threads to end
   for(Thread reqThread: requestThreads)
     while(reqThread.isAlive()) {}

   this.open = false; //quit the request or socket-accepting loop

   //close main server (base) socket
   try
   { serverSocket.close(); }
   catch(Exception e)
   { System.out.println("Failed to close proxy server socket :("); }

 }



/**
```

```
   * Thread of WebProxy object starts here. Just makes the proxy start listening
   *
   * @return: None
   */
  @Override
  public void run()
  {
    start_listening();
  }

}
```

RequestHandler.java

```
/* author: Shaun Jose
   github: github.com/ShaunJose
   Class Description: Handles all types of requests appropriately (http or https)
*/

//imports
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.util.Scanner;

public class RequestHandler implements Runnable
{
  //define type enum
  enum ReqType { HTTP, HTTPS; }

  //class variables
  private Socket clientSocket;
  private Socket serverSocket;
  private ReqType type;
  private int port;

  //constants
  public static final String SUCCESS_STATUS = "HTTP/1.1 200 Connection
established\r\n\r\n";
  public static final String FORBIDDEN_STATUS = "HTTP/1.1 403 Access
forbidden\r\n\r\n";
  public static final int HTTP_PORT = 80;
  public static final int MAX_BYTES = 4096;

  /**
   * Constructor. Creates object and initialises clientSocket, type to HTTP (i.e. not secure)
by default and port set to HTTP_PORT by default as well
```

```
  *
  * @param clientSocket: The socket through which the client sent a request
  */
  RequestHandler(Socket clientSocket)
  {
    this.clientSocket = clientSocket;
    this.type = ReqType.HTTP;
    this.port = HTTP_PORT;
  }


  /**
   * Handles a request appropriately, by connecting to the right server and getting info
from it
   *
   * @return: None
   */
  private void processRequest()
  {
    //Get method, host, and full request. Also change port and type depending on if the
request is an https or http one
    String[] reqStuff = getMethodHostReq();
    String method = reqStuff[0];
    String hostName = reqStuff[1];
    String request = reqStuff[2];
    System.out.println("\nNew " + type + " request:\n" + request);

    //check if host is blocked
    if(ManagementConsole.blocked(hostName))
    {
      try { //send FORBIDDEN_STATUS to client
        DataOutputStream outputStream = new
DataOutputStream(clientSocket.getOutputStream());
        outputStream.writeUTF(RequestHandler.FORBIDDEN_STATUS);
        outputStream.flush();
      }
      catch(Exception e) { e.printStackTrace(); }
      //inform manager about this naughty business
      System.out.println("Client with address " + clientSocket.getRemoteSocketAddress() +
" tried to access " + hostName + "!"); //let manager know who tried to access what
      this.shutDown(); //close client socket connection
      return; //end request thread
    }

    //check if it's cached (only http can be cached)
    String response = null;
    if(type == ReqType.HTTP && ManagementConsole.isCached(hostName))
    {
      response = ManagementConsole.getFromCache(hostName);
      System.out.println("Cache Hit!\n");
    }

    //Connect to appropriate server and start req-response transmission
    try
    {
      //open connection to server
      serverSocket = new Socket(hostName, this.port);

      //send client HTTP request to server if not retrieved from cache
      DataOutputStream outputStream;
      if(type == ReqType.HTTP && response == null)
      {
        outputStream = new DataOutputStream(this.serverSocket.getOutputStream());
        outputStream.writeUTF(request);
        outputStream.flush();
      }

      /* --- response part --- */

      // set up client output stream
      outputStream = new DataOutputStream(clientSocket.getOutputStream());

      //if req is https, listen to client and server until they're both done
      if(type == ReqType.HTTPS)
      {
        //send success message to client!
        outputStream = new DataOutputStream(clientSocket.getOutputStream());
        outputStream.writeBytes(RequestHandler.SUCCESS_STATUS);
        outputStream.flush();

        //listen to client on another thread
        ClientListenHTTPS clientListener = new ClientListenHTTPS(clientSocket, 
serverSocket);
        Thread clientThread = new Thread(clientListener);
        clientThread.start();
        //listen to server using the current thread
        ServerListenHTTPS serverListener = new ServerListenHTTPS(clientSocket, 
serverSocket, clientThread);

        serverListener.listenAndSend();
      }
      else //if req is http, then send response once and relax :)
      {
        if(response == null) //if not retrieved from cache
          response = getHTTPResponse();//get response from server in response to query

        //save response to cache or update it's position for LRU policy
        ManagementConsole.saveToCache(hostName, response);

        //send response to client
        outputStream.writeUTF(response);
        outputStream.flush();
      }

    }
    catch(Exception e)
    {
      System.out.println("Could not connect to actual server :( \n" +
      "Check if you entered the url correctly!!");
      e.printStackTrace();
    }

    this.shutDown(); //close client and server sockets

    System.out.println("Socket request touchdown! :D");
  }



  /**
   * Finds out the method, host and request from the input stream of the clientSocket, for
HTTP/HTTPS requests. Also manages class variables depending on the type of request
(http/s)
   *
   * @return: String array of the method, host name and the request
   */
  private String[] getMethodHostReq()
  {
    String[] reqStuff = new String[3]; //will contain method, host name, req
    String requestMessage = ""; //will contain entire request message

    //get method and hostName from HTTP request
    try
    {
      DataInputStream inputStream = new DataInputStream(clientSocket.getInputStream());
//open input stream

      //get method
      requestMessage = inputStream.readLine() + "\r\n"; //get first line of req
      int firstSpace = requestMessage.indexOf(' ');
      reqStuff[0] = requestMessage.substring(0, firstSpace);

      //get rest of message
      String restOfMessage = getRequest(reqStuff[0]);
      //get hostname depending of req being HTTPS/HTTP
      if(reqStuff[0].equals("CONNECT"))
        reqStuff[1] = requestMessage.substring(8, requestMessage.lastIndexOf(' ')); //first
line = CONNECT hostName HTTPversion
      else
        reqStuff[1] = restOfMessage.substring(6, restOfMessage.indexOf("\r\n")); //get it
from the Host header line

      requestMessage += restOfMessage; //add restOfMessage to requestMessage
    }

    catch(Exception e)
    {
      System.out.println("Could not read input request :(");
      e.printStackTrace();
    }

    //change variables if it's an https request, by checking what method is
    if(reqStuff[0].equals("CONNECT"))
    {
      String[] hostAndPort = reqStuff[1].split(":"); //split host and port
      reqStuff[1] = hostAndPort[0]; //first part is host
      this.port = Integer.parseInt(hostAndPort[1]); //second part is port num
      this.type = ReqType.HTTPS; //type is an HTTPS request
    }

    reqStuff[2] = requestMessage; // add req message into the return array

    return reqStuff;
  }
```

```
/**
 * Gets the HTTP/S request message from the client
 *
 * @param method: HTTP method being used
 *
 * @return: String with the HTTP/HTTPS request from the client side
 */
private String getRequest(String method)
{
  String requestLine = "";
  String requestMessage = "";
  Scanner sc = null;

  //get client input stream
  try
  { sc = new Scanner(clientSocket.getInputStream()); }
  catch(Exception e)
  { e.printStackTrace(); }

  //get header of client request
  do
  {
    requestLine = sc.nextLine() + "\r\n";
    requestMessage += requestLine;
  } while(!requestLine.equals("\r\n"));

  //get body if it exists (CONNECT HTTPS requests dont have a body)
  if(method.equals("POST") || method.equals("PUT"))
  {
    requestLine = "";
    String body = "";

    //get body of client request
    do
```

```
      {
        requestLine = sc.nextLine() + "\r\n";
        requestMessage += requestLine;
      } while(!requestLine.equals("</html>") && sc.hasNext());
    }

    return requestMessage;
  }



  /**
   * Gets the response message from the server
   *
   * @return: String with the HTTP request from the client side
   */
  private String getHTTPResponse()
  {
    String responseLine = "";
    String responseMessage = "";
    Scanner sc = null;

    //get server input stream
    try
    { sc = new Scanner(serverSocket.getInputStream()); }
    catch(Exception e)
    { e.printStackTrace(); }

    //get header of server response
    do
    {
      responseLine = sc.nextLine() + "\r\n";
      responseMessage += responseLine;
    } while(!responseLine.equals("\r\n"));

    //get body
    responseLine = "";
    String body = "";

    //get body of server response
    do
    {
      responseLine = sc.nextLine() + "\r\n";
      responseMessage += responseLine;
    } while(!responseLine.contains("</html>") && sc.hasNext());

    return responseMessage; //return entire response
  }



  /**
   * Closes the client and server socket connections
   *
   * @return: None
   */
  private void shutDown()
  {
    try
    {
      if(clientSocket != null && !clientSocket.isClosed())
        clientSocket.close();

      if(serverSocket != null && !serverSocket.isClosed())
        serverSocket.close();
    }

    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
  /**
   * Thread starts from here. This function calls processRequest
   *
   * @return: None
   */
  @Override
  public void run()
  {
    processRequest();
  }

}
```

ServerListenHTTPS.java

```
/* author: Shaun Jose
   github: github.com/ShaunJose
   Class Description: Handles HTTPS requests from the Server side, listens to server and
sends to client
*/

//imports
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;

class ServerListenHTTPS
{
  //class variables
  private DataInputStream serverIn;
  private DataOutputStream clientOut;
  private Thread clientThread;

  /**
   * Constructor. initialises class variables (in/out streams) using sockets
   *
   * @param clientSock: Socket through which the client is connected to proxy
   * @param serverSock: Socket through which the proxy is connected to server
   */
  ServerListenHTTPS(Socket clientSock, Socket serverSock, Thread clientThread)
  {
    try
    {
      //initialise IO streams and client thread
      this.serverIn = new DataInputStream(serverSock.getInputStream());
      this.clientOut = new DataOutputStream(clientSock.getOutputStream());
      this.clientThread = clientThread;
    }
  catch(Exception e)
  {
    e.printStackTrace();
  }

}



/**
 * Listens to the server and sends to the client until server is done
 *
 * @return: None
 */
public void listenAndSend()
{
  //set up array where you storing the bytes
  byte[] messageBytes = new byte[RequestHandler.MAX_BYTES];
  int retVal; // return value from the read function
  boolean serverSending = true;

  try
  {
    //get bytes from server and send to client
    while(serverSending)
    {
      retVal = serverIn.read(messageBytes); //get message and return value
      serverSending = retVal != -1; //update the serverSending boolean
      //send the client the bytes if there are bytes to send
      if(serverSending)
      {
        clientOut.write(messageBytes, 0, retVal);
        clientOut.flush(); //flush it out
      }
    }
```

```
      //wait until client is done
      while(clientThread.isAlive())
      {}
    }

    catch(Exception e)
    {
      e.printStackTrace();
    }

  }
}
```

ClientListenHTTPS.java

```
/* author: Shaun Jose
   github: github.com/ShaunJose
   Class Description: Handles HTTPS requests from the Client side, listens to client and
sends to server
*/

//imports
import java.net.Socket;
import java.io.DataInputStream;
import java.io.DataOutputStream;

class ClientListenHTTPS implements Runnable
{
  //class variables
  private DataInputStream clientIn;
  private DataOutputStream serverOut;

  /**
   * Constructor. initialises class variables (in/out streams) using sockets
   *
   * @param clientSock: Socket through which the client is connected to proxy
   * @param serverSock: Socket through which the proxy is connected to server
   */
  ClientListenHTTPS(Socket clientSock, Socket serverSock)
  {
    try
    {
      //initialise IO streams using the sockets passed
      this.clientIn = new DataInputStream(clientSock.getInputStream());
      this.serverOut = new DataOutputStream(serverSock.getOutputStream());
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }

  }



  /**
   * Listens to the client and sends to the server until client is done
   *
   * @return: None
   */
  private void listenAndSend()
  {
    //set up array where you storing the bytes
    byte[] messageBytes = new byte[RequestHandler.MAX_BYTES];
    int retVal; // return value from the read function
    boolean clientSending = true;

    try
    {
      //get bytes from client and send to server
      while(clientSending)
      {
        retVal = clientIn.read(messageBytes); //get message and return value
        clientSending = retVal != -1; //update the clientSending boolean
        //send the server the bytes if there are bytes to send
        if(clientSending)
        {
          serverOut.write(messageBytes, 0, retVal);
          serverOut.flush(); //flush it out
        }
      }
    }

    catch(Exception e)
    {
      e.printStackTrace();
    }

  }



  /**
   * Thread starts from here. This function calls listenAndSend
   *
   * @return: None
   */
  @Override
  public void run()
  {
    listenAndSend();
  }

}
```

README.md

# Web Proxy Server

### Description
An implementation of a web proxy server (acting as an intermediary between a client and a host) in Java.

### Tools used
1. JVM (for Java code)

2. curl

### Instructions:
1. Compile ManagementConsole.java
```
javac ManagementConsole.java
```

2. Run ManagementConsole.java
```
java ManagementConsole
```

3. Block sites you want to by following instructions which will we displayed

4. Open new tab on terminal and use curl to send a request to the proxy
```
curl -x localhost:4000 <InterestedSiteUrl>
```