Overview:

In this practical, we were to write a Proxy Server which could be used to keep requests and data secured. It was to take in requests from the client in HTTP (unencrypted), talk to the server requested using HTTPS (encrypted), and then finally transmit the data it receives from the server back to the client.

Design & Extensions:

I decided to have 2 classes to fulfil the specification: one called "Proxy" which is the runnable class, which runs the proxy on a given port and accepts requests, and one called "Handler", which does the main work of handling the request and displaying the data it receives back to the client. In addition, I decided to attempt 2 extensions, which were to handle caching in my proxy, and also to handle HTTPS requests from the client as well as normal HTTP requests.

Proxy:

This class extends the 'Thread' class, so that it can have multiple executions if there are multiple requests. It has a constructor which assigns the port on which it will run on, and a run method which opens a new server socket on the given port and sends any requests it receives to the Handler class (which it creates a new instance of for each request, and launches its run method). Finally, it has a main runnable method, which takes in 1 program argument which should be the port number that the proxy should run on. It then simply calls the run method with the port number, catching any errors such as invalid number of arguments or invalid port numbers, in which case it gives meaningful output to the console.

Handler:

Throughout this class, I control the output to the console by using an if statement, which was my attempt to only print out output when the URL in the request is something that the user would care about, ignoring all the extra requests made by each webpage. Without these controls, the output to the console is extremely long and therefore not very useful or understandable. Due to this, some URLs are missed out and aren't output to the system, but this has no effect on the proxy or on request handling – just the output to the console. Also, I have left some 'catch' blocks empty on purpose, for a similar reason. Even though all webpages load fine using the proxy, some throw many errors and this therefore clogs the console, hiding any meaningful output. Therefore, since these errors/exceptions don't stop the proxy from working and aren't something I could fix for all websites, I decided to not provide any output for them.

This class also extends the 'Thread' class for the same reason as the Proxy class, and has a number of static variables, such as maximum number of bytes to be stored in the buffer, the maximum size of the cache, the cache itself which is a LinkedHashMap (as I thought it would be sensible to be able to access the bytes of the response of the server as the value, and the URL of the server as the key, while being able to keep track of the order of the elements in which they were added), the client socket and finally an HTTPS Pattern to be able to identify requests. There is a constructor which simply takes in the client socket and assigns it to the static client socket in the class. I then chose to have a number of methods which would be used by the main run method. There is a getRequest method which takes in a bufferedreader, which reads from the input stream from the client socket. It reads the first line of the request using the buffered reader, which will be the request, and returns it. The sendHTTPSRequest takes in the matcher which checks if the request matches the HTTPS pattern, the URL in the request and the same URL with no protocol to be used by

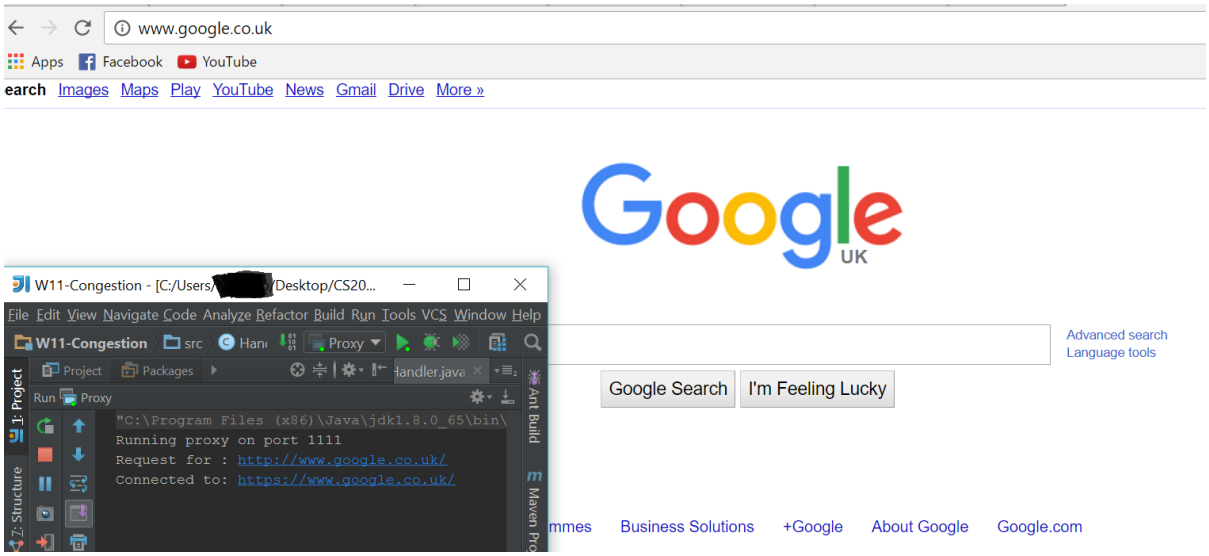
the cache, and the output stream in which it should write the response. It firstly opens a socket using the information in the matcher, which will be used to send the client's request to the server it is trying to access. If this fails, the sendHTTPRequest method is called to try creating an HTTP request instead, rather than failing completely. Otherwise, it writes a response to show that this was successful, and then opens a new Thread to start sending the client's requests to the server in parallel (as there may be many), by calling the sendData method. It then finally closes all the resources. Similarly, the sendHTTPRequest takes in the same arguments but without the matcher (as we know this will need to be an HTTP request). It changes the URL to resemble an HTTP request in case it doesn't, opens a new HttpURLConnection using the URL, and calls the sendData method to display the response to the client. If anything fails, meaningful output is provided to the console. The sendDataFromCache method takes in the URL and the output stream where it should write the response, and simply creates a ByteArrayInputStream, the contents of which are taken from the cache using the URL, and calls the sendData method with these streams. The sendData method which has been used by all the other methods takes in the URL and the protocol (used for output) as strings, the input stream which it should read from and the output stream to which it should write. After checking that the input stream isn't null, it checks whether or not the URL is in the cache. It then uses a buffer to read the response in the form of an array of bytes from the input stream, keeping track of the index of the buffer. If the URL is not in the cache, it writes the array of bytes to a ByteArrayOutputStream, as well as the output stream it took as argument. After providing meaningful output to the console, it adds the bytes of data and the URL to the cache if they aren't already there. This involves checking if the cache has reached maximum capacity, and if so, it removes the oldest element, which can easily be done as linked hashmaps store elements in order in which they were added, and so keeps an index of the last position where something was removed. Finally, the run method is the main one which is used by the Proxy, and which calls all these other methods. This method creates a BufferedReader to read from the client socket's input stream, and a DataOutputStream to write to the client's output. It gets the URL from the request to be used later, and provides output to the console. If the URL is in the cache, the sendDataFromCache method is called, otherwise it checks if the request matches the HTTPS pattern using a matcher. If it does, it calls the sendHTTPSRequest method. Otherwise, we have an HTTP request, which it tries to change to an HTTPS request, which is what the specification required, by opening an HttpsURLConnection. If the response code from this is 200, the switch to HTTPS worked, and so sendHTTPSRequest is called. Otherwise, sendHTTPRequest is called, and the request will have to remain unencrypted, rather than the request failing. If a ConnectException is encountered, I call sendHTTPRequest again to try one more time, which more often than not, is the difference between a request being correctly handled or not.

Running:

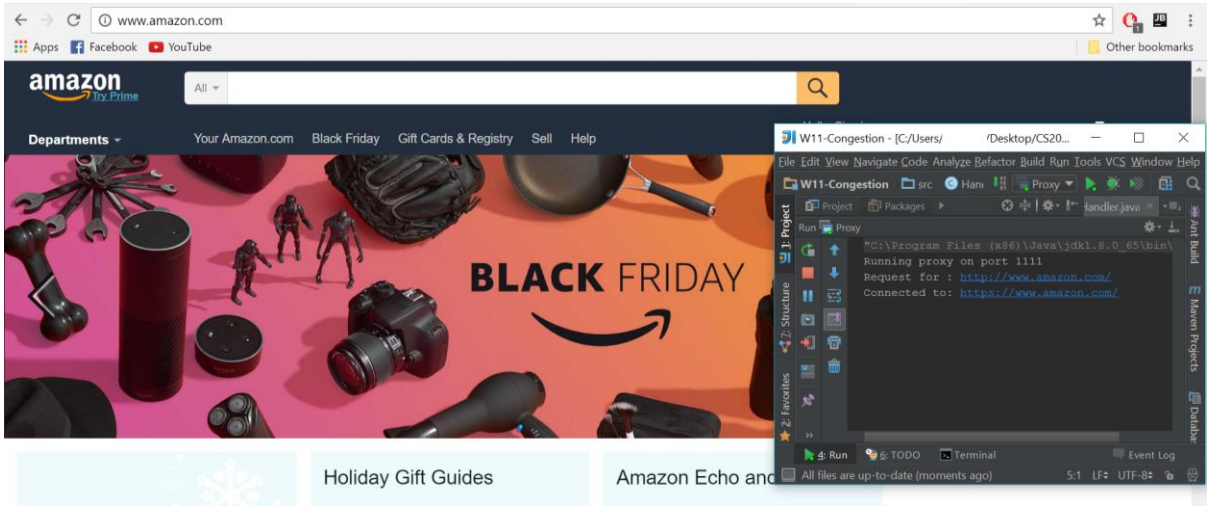
To run my Proxy, first of all configure the browser you will use to go through "localhost" or "127.0.0.1", and supply a port number. If you are using terminal and 'curl', this isn't necessary. Then, run the Proxy class, with the same port number as before as argument.

Testing:

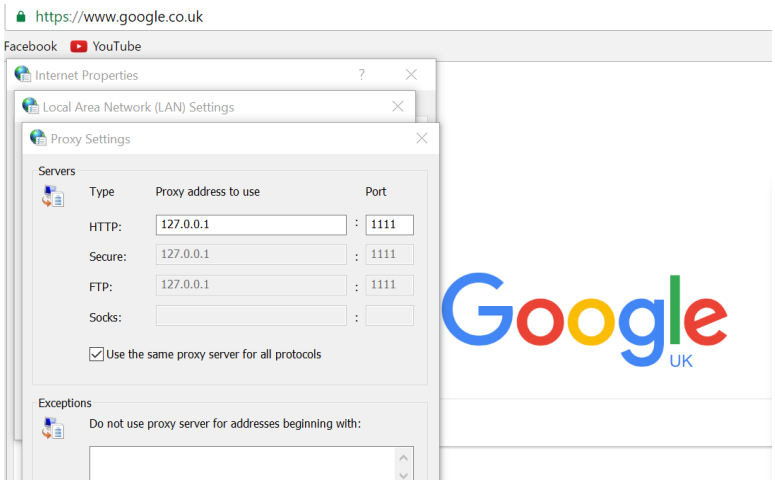
Normal HTTP request works, and is converted to HTTPS, as we can see on the console:



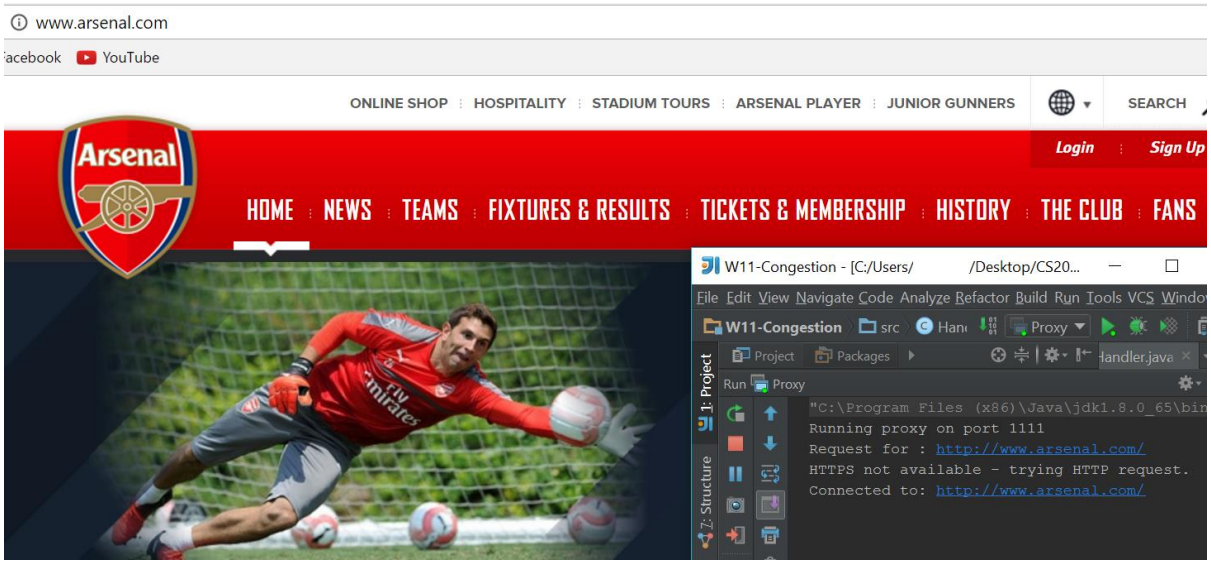
Same as above with images, but the Amazon website doesn't support HTTP, so this proves my proxy changes the request to HTTPS (the green lock isn't there on the top left):



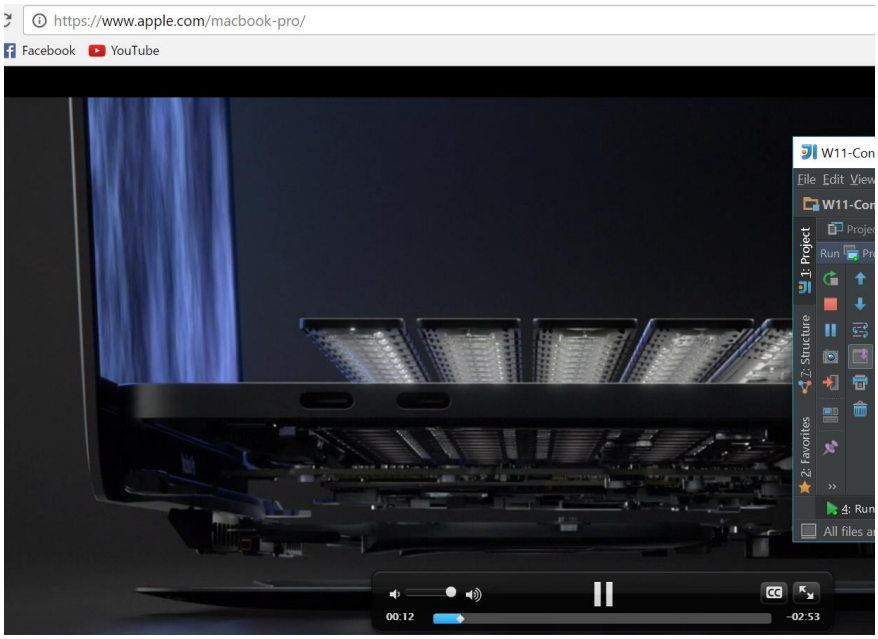
HTTPS request for google works through proxy, as you can see because I have instructed the browser to send HTTPS requests through the proxy too:



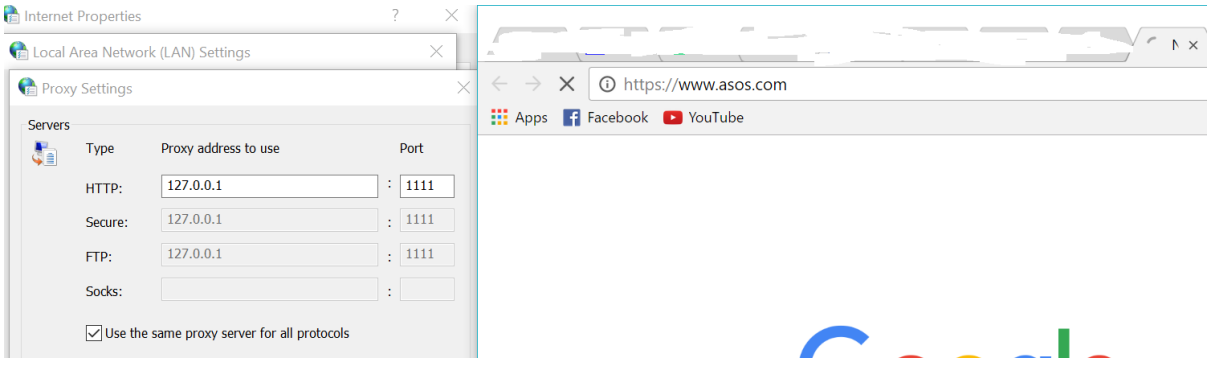
If an HTTPS request isn't possible, an HTTP request is made, with meaningful output to the console:

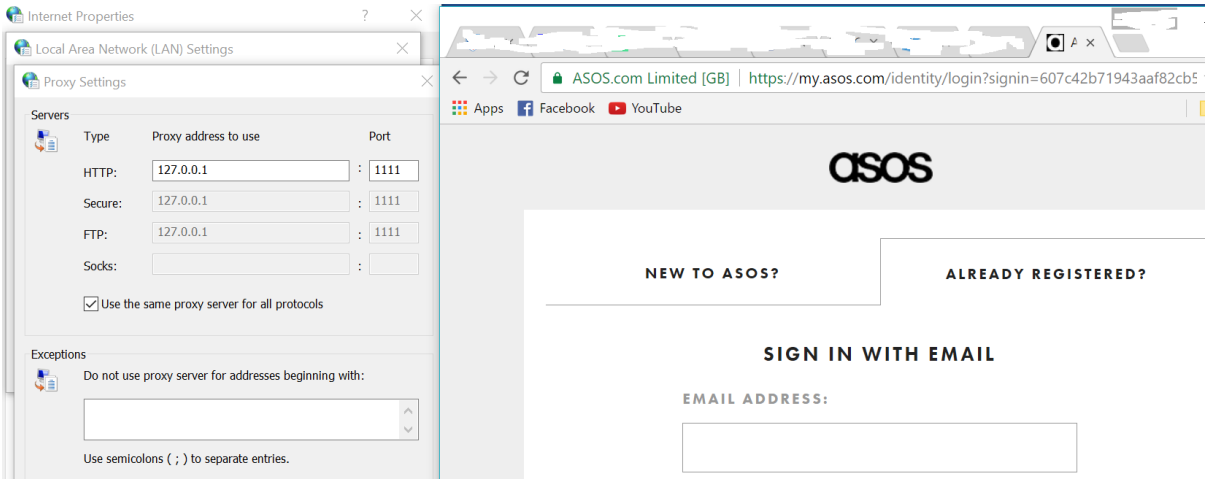


Videos and images work through the proxy:

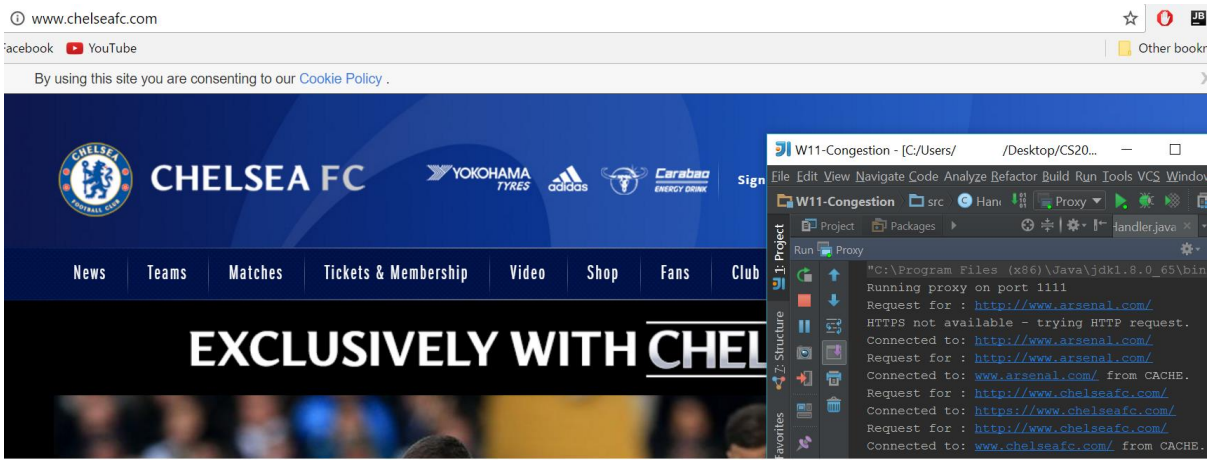


Redirecting works, settings included to prove that HTTPS requests go through proxy:

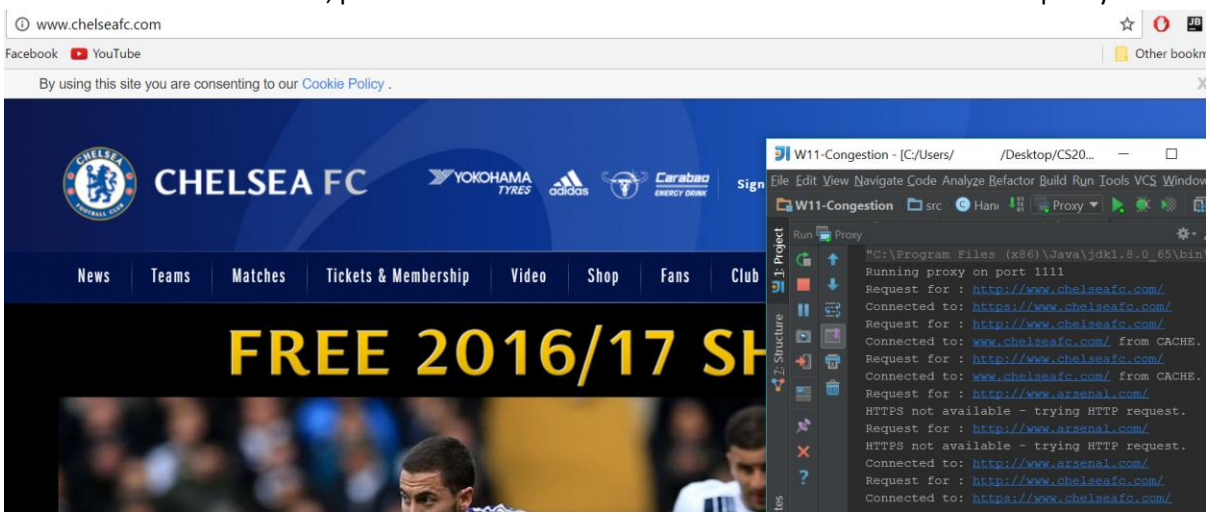




Caching works for multiple websites, as shown by the output to the console:

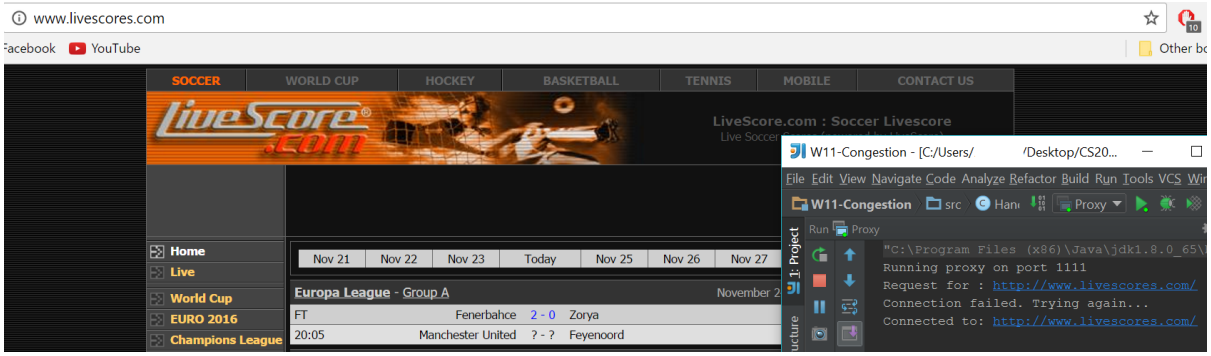


Reduced size limit of cache, proof that it deletes some URLs when it has reached full capacity:



"www.chelseafc.com" was connected to twice through the cache, then "www.arsenal.com" was connected to, and then when the request for "www.chelseafc.com" was made again after arsenal, it wasn't loaded through the cache, which means it wasn't there.

Proof that trying to send an HTTP request again when a request has failed is worth it, as the website ended up loading. This is because HTTPS was attempted, but threw an exception.



Evaluation & Conclusion:

Overall, I am very satisfied with my Proxy, as I have succeeded to fulfil the basic requirements and more, which can be seen by my tests. They prove that my Proxy successfully gets HTTP requests from the client and turns them into HTTPS requests to the server, displaying the response back to the client. Images and videos work too, and this is further proven as my Proxy manages to load Amazon without the green lock on the top left, which would only be possible if this was done with HTTP, which Amazon doesn't support, but which can also be seen through the console. In addition to all this, I have done the extensions of caching websites, and handling HTTPS requests directly from the client. Handling HTTPS is proven in my tests as well as caching, which I also prove with the tests. Caching also successfully deletes the oldest requests when it reaches full capacity. Although I found some difficulty in getting started with this practical, after looking at some examples online, I managed to write a basic solution, which I managed to perfect and extend by myself.