Computer Networks

Phase 1

Web Server

N. 49423 Rafael Pegacho
N. 49431 Tiago Neto

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2021/2022

10/04/2022

1 Introduction

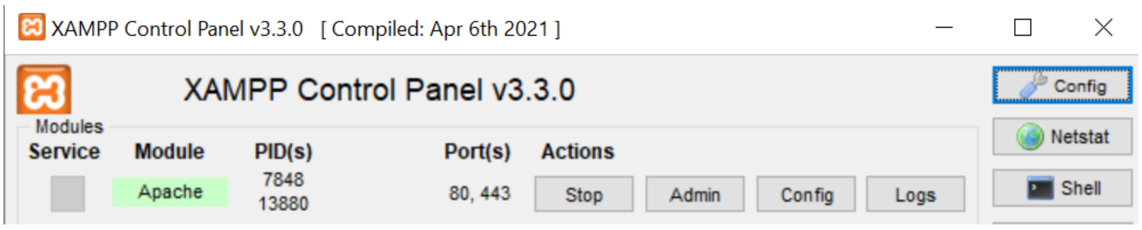
This project consists of analyzing and understanding how the HTTP requests and replies happen and taking a look at the correlation between TCP and HTTP. While the project is being explained we are gonna take some conclusion about what we learnt in the process.

To get to the objective we need a Web Server and a Web Client, we will use our machine as host and a free Web Server named "apache", then we Will use a Web Browser as a client and code a Web Client to later analyze and compare how both of them communicate with the server. To analyze this "conversation" we will be using a Network Protocol Analyzer tool named "WireShark"

2 Starting Web Server

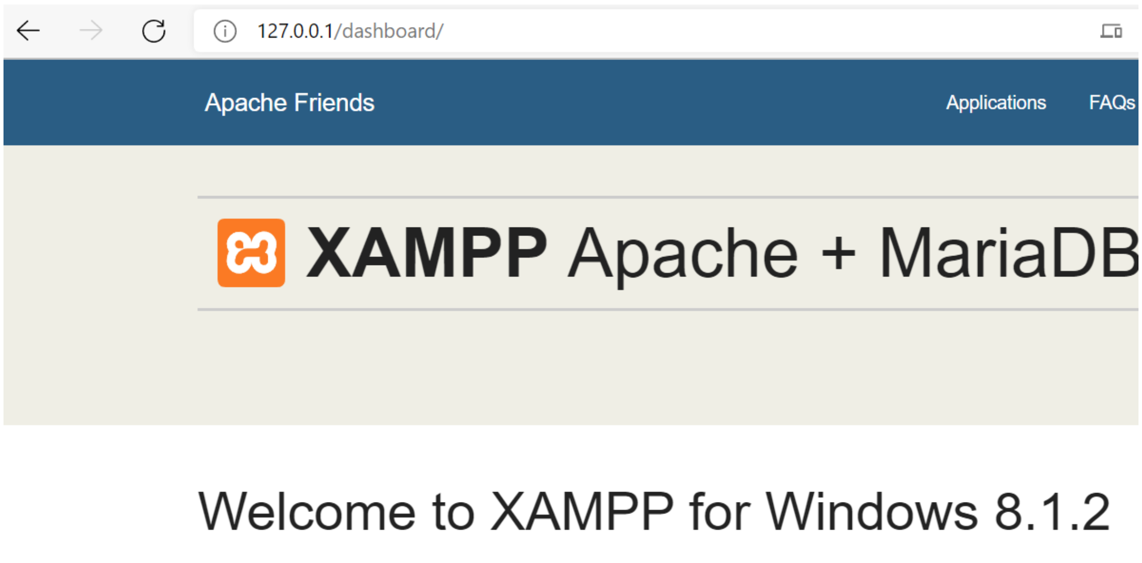
2.1 Starting XAMPP/Apache

To begin with, we need to download and install XAMP and after having that installed we need started the Apache web server.



2.2 LoopBack Address

Now we needed to test if the Web Server was actually working. At first we tested it on the machine that is hosting the web server, so we used the LoobBack Address for it (127.0.0.1).

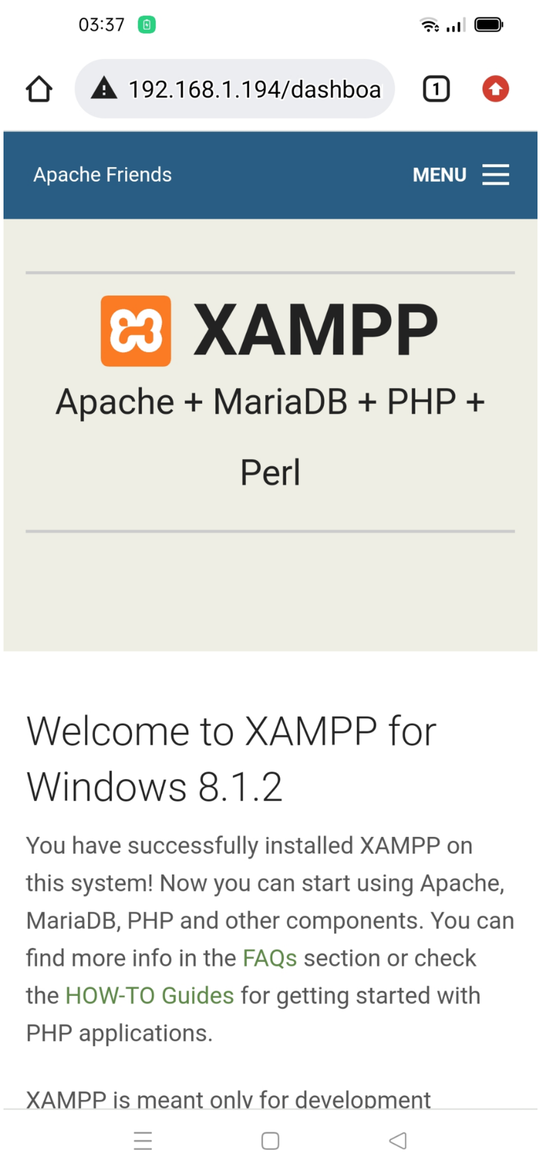


As the printscreen shows, everything is working great (packet analysis will later be shown).

2.3 Other devices

Taking advantage of the "ipconfig" command we saw that our LAN IPv4 was 192.168.1.194 and then tried to connect to it using a mobile device.

The result was the same so everything was fine.



Apache + MariaDB + PHP + Perl

Welcome to XAMPP for Windows 8.1.2

You have successfully installed XAMPP on this system! Now you can start using Apache, MariaDB, PHP and other components. You can find more info in the FAQs section or check the HOW-TO Guides for getting started with PHP applications.

XAMPP is meant only for development

3 Learning Phase

3.1 HTTP vs HTTPS

After testing the web server and before going directly to code, we decide to learn a bit more about HTTP and HTTPS.

3.1.1 TLS/SSL Encryption

As we were taught in RC classes, HTTPS isn't much more than an encrypted HTTP, using TLS/SSL, but how is it decrypted by the Client?

3.1.2 TLS/SSL HandShake

As we were taught as well, there is a protocol called "TLS handshake". When the TCP three-way handshake is done and as soon as the Client Acknowledges it, the Client sends a "hello" message that is replied by the server with a certificate that tells us that it is in fact the Server we want to access. After that there is a KeyExchange, finally that key is used to decrypt the encrypted data.

3.2 How to code the HTTP Client

The knowledge acquired about HTTP/HTTPS, TCP, TLS/SSL made us comfortable about learning about how to code the Client.

3.2.1 With HTTP libraries

We first went to Youtube/Github/StackOverflow and saw some HTTP Client coding videos, nearly all of them using HTTP libraries, but it was still a good way to understand the steps taken.

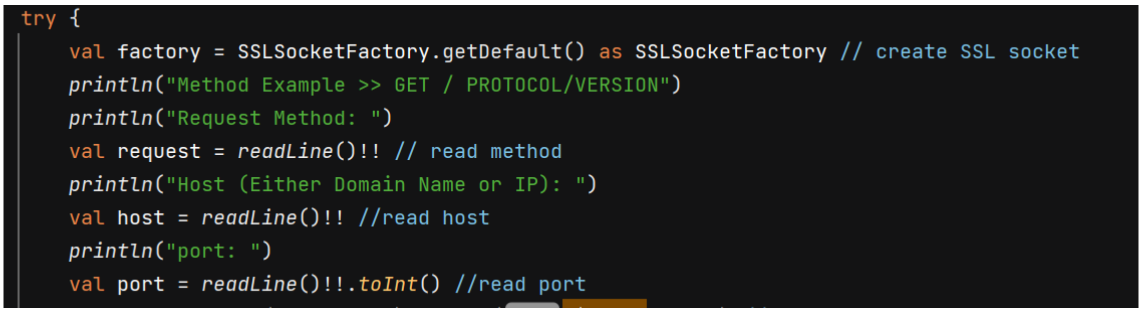
3.2.2 Without HTTP libraries

After understanding the steps needed to code an HTTP Client we went looking out on how to do it without HTTP libraries, after searching for a long time we found several Clients in different languages and with different functionalities and then we felt ready to start coding.

4 Coding the HTTP Client

4.1 Create the Inputs

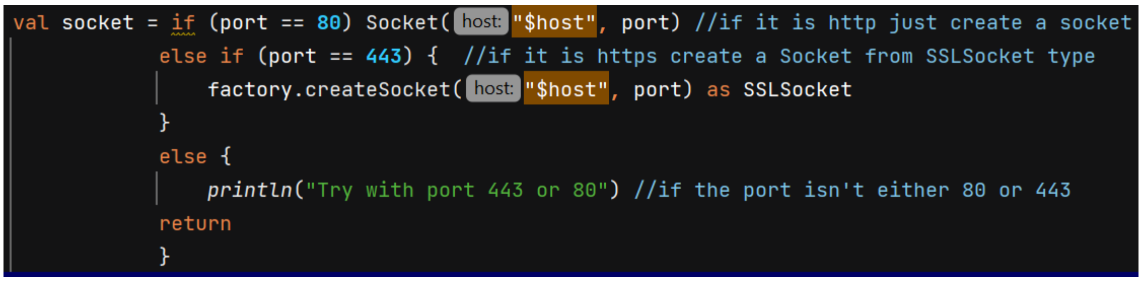
We started coding by asking the "host", "request method" and "port".



```
try {
    val factory = SSLSocketFactory.getDefault() as SSLSocketFactory // create SSL socket
    println("Method Example >> GET / PROTOCOL/VERSION")
    println("Request Method: ")
    val request = readLine()!! // read method
    println("Host (Either Domain Name or IP): ")
    val host = readLine()!! //read host
    println("port: ")
    val port = readLine()!!.toInt() //read port
```

4.2 Socket connection to a listening Port

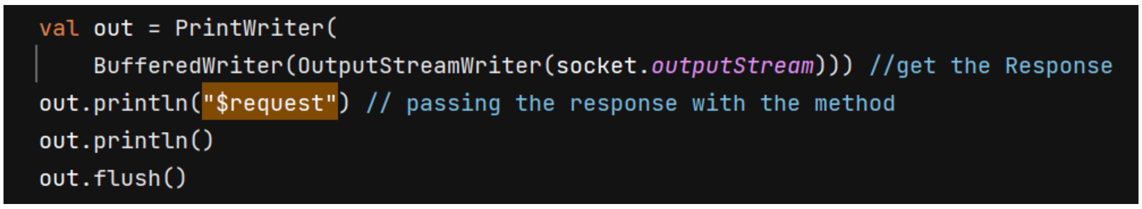
It's time to connect the socket to the port the server is listening to and decide if it's gonna be a Socket for HTTP or HTTPS.



```
val socket = if (port == 80) Socket( host: "$host", port) //if it is http just create a socket
            else if (port == 443) {  //if it is https create a Socket from SSLSocket type
                factory.createSocket( host: "$host", port) as SSLSocket
            }
            else {
                println("Try with port 443 or 80") //if the port isn't either 80 or 443
            return
            }
```

4.3 Getting the response

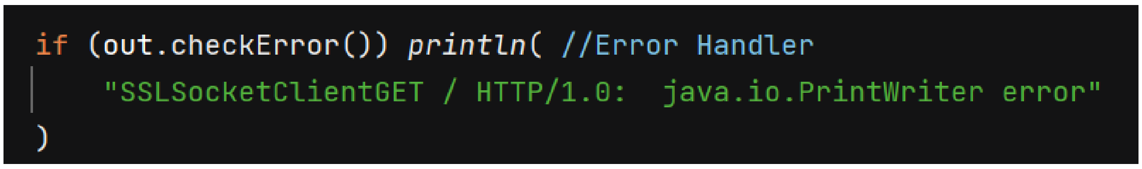
Using the method passed before we get the response message and we pass it to an "out".



```
val out = PrintWriter(
    BufferedWriter(OutputStreamWriter(socket.outputStream))) //get the Response
out.println("$request") // passing the response with the method
out.println()
out.flush()
```

4.4 Error handling

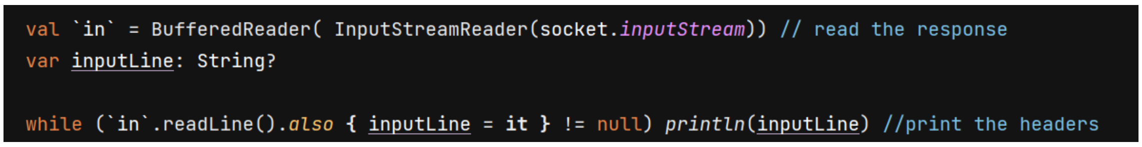
Just in case there is any error, we decided to print an error message to avoid errors in the execution.



```
if (out.checkError()) println( //Error Handler
    "SSLSocketClientGET / HTTP/1.0:  java.io.PrintWriter error"
)
```

4.5 Printing the GET Request Headers

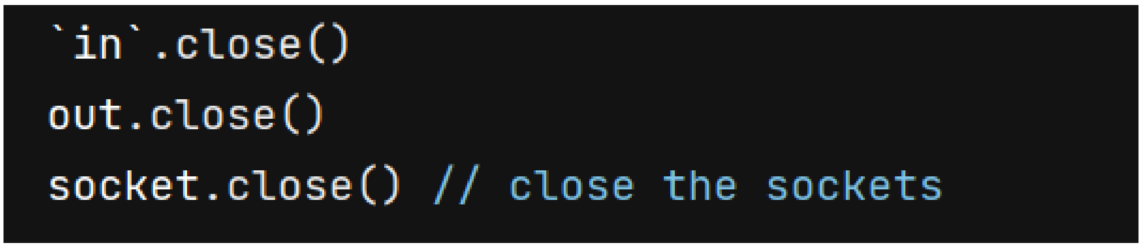
Having the error cases covered and having the responses, we read it from the socket and then printed the headers.



```
val `in` = BufferedReader( InputStreamReader(socket.inputStream)) // read the response
var inputLine: String?

while (`in`.readLine().also { inputLine = it } != null) println(inputLine) //print the headers
```

4.6 Closing the Socket

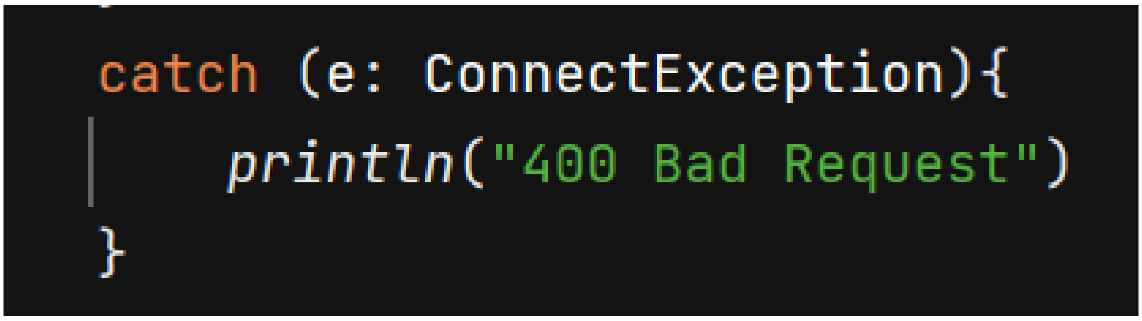
Everything is done now and we just need to close the connection, simply closing the writers and the socket.



```
`in`.close()
out.close()
socket.close() // close the sockets
```

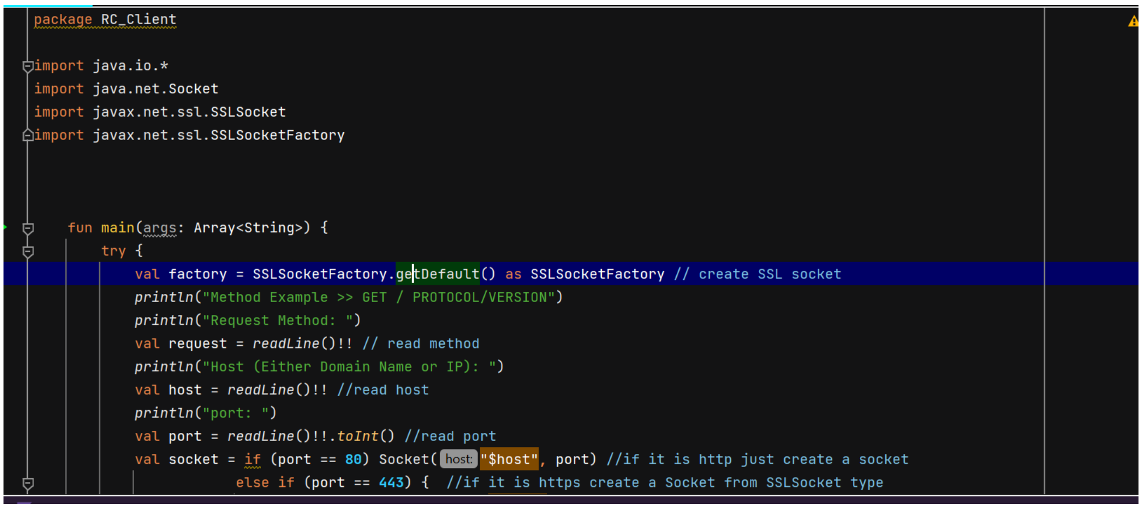
4.7 Connect Exception (400 code)

In case the host or port doesn't exist the result would be an "400 Bad request", being this case a ConnectException.

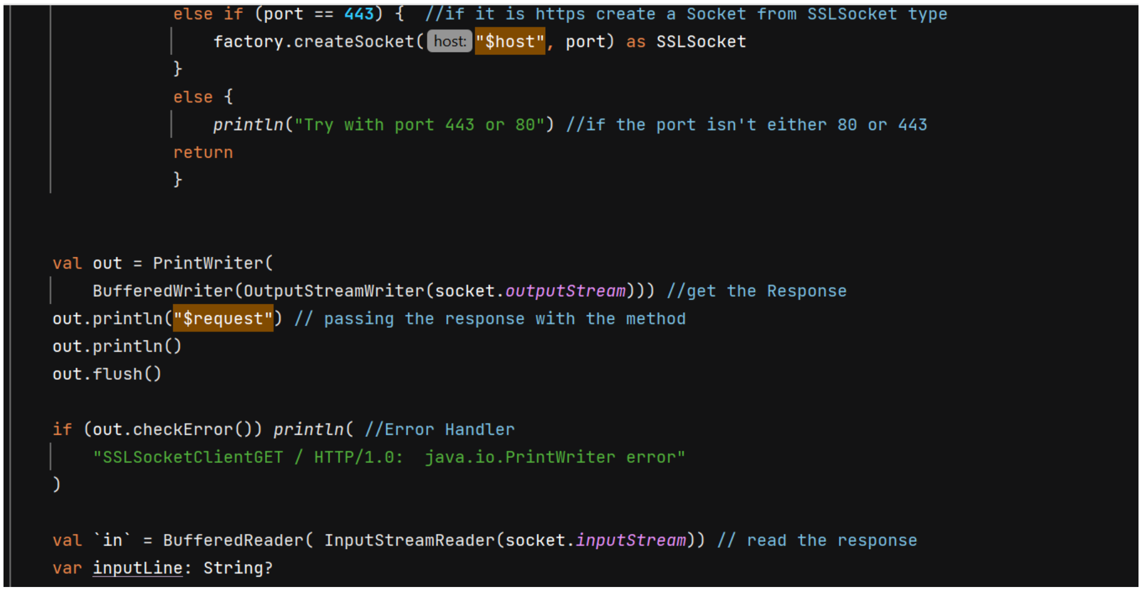


```
catch (e: ConnectException){
    println("400 Bad Request")
}
```

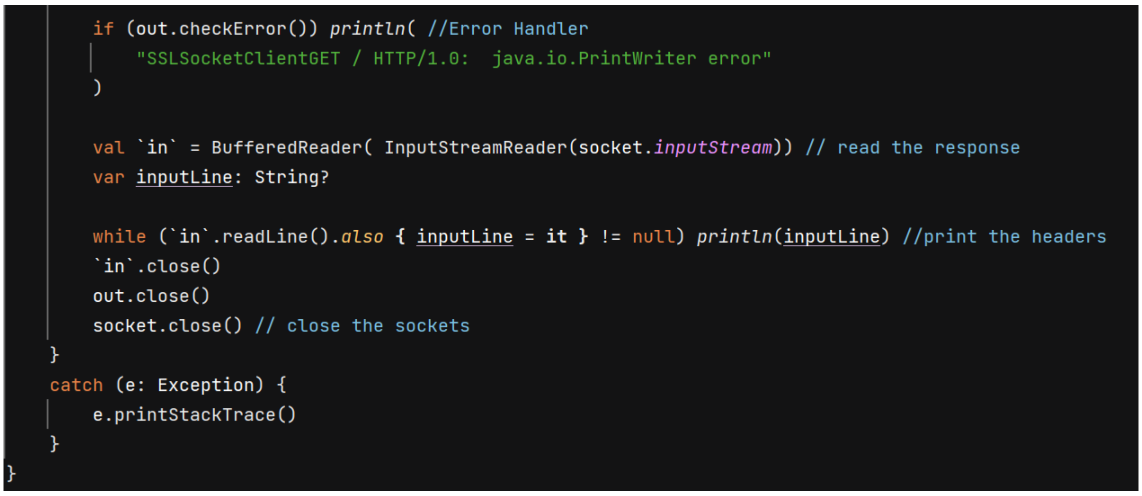
4.8 General code view



```
package RC_Client

import java.io.*
import java.net.Socket
import javax.net.ssl.SSLSocket
import javax.net.ssl.SSLSocketFactory


fun main(args: Array<String>) {
    try {
        val factory = SSLSocketFactory.getDefault() as SSLSocketFactory // create SSL socket
        println("Method Example >> GET / PROTOCOL/VERSION")
        println("Request Method: ")
        val request = readLine()!! // read method
        println("Host (Either Domain Name or IP): ")
        val host = readLine()!! //read host
        println("port: ")
        val port = readLine()!!.toInt() //read port
        val socket = if (port == 80) Socket( host: "$host", port) //if it is http just create a socket
                else if (port == 443) {  //if it is https create a Socket from SSLSocket type
```



```
            else if (port == 443) {  //if it is https create a Socket from SSLSocket type
                factory.createSocket( host: "$host", port) as SSLSocket
            }
            else {
                println("Try with port 443 or 80") //if the port isn't either 80 or 443
            return
            }


val out = PrintWriter(
    BufferedWriter(OutputStreamWriter(socket.outputStream))) //get the Response
out.println("$request") // passing the response with the method
out.println()
out.flush()

if (out.checkError()) println( //Error Handler
    "SSLSocketClientGET / HTTP/1.0:  java.io.PrintWriter error"
)

val `in` = BufferedReader( InputStreamReader(socket.inputStream)) // read the response
var inputLine: String?
```

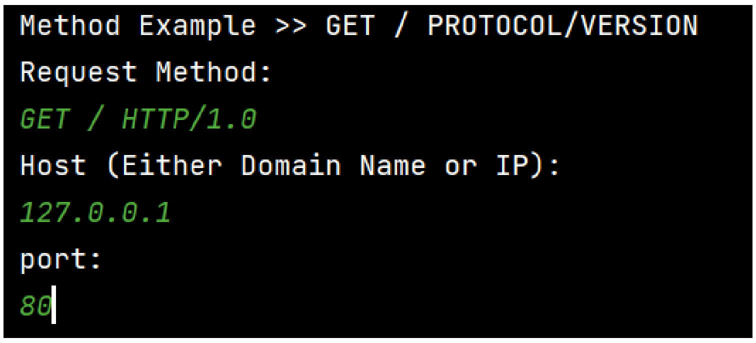


```
        if (out.checkError()) println( //Error Handler
            "SSLSocketClientGET / HTTP/1.0:  java.io.PrintWriter error"
        )

        val `in` = BufferedReader( InputStreamReader(socket.inputStream)) // read the response
        var inputLine: String?

        while (`in`.readLine().also { inputLine = it } != null) println(inputLine) //print the headers
        `in`.close()
        out.close()
        socket.close() // close the sockets
    }
    catch (e: Exception) {
        e.printStackTrace()
    }
}
```

5 Testing the Code

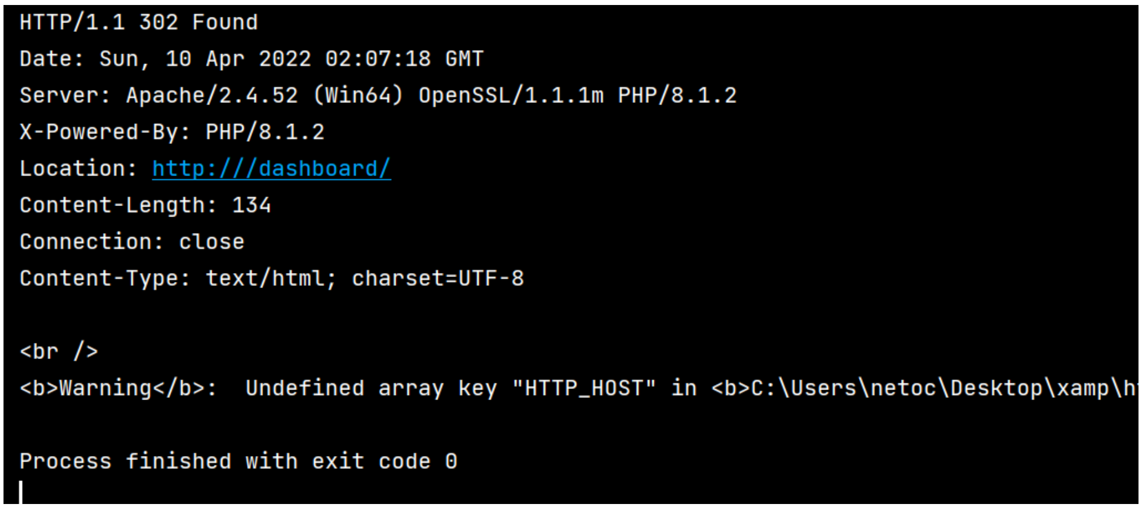
5.1 HTTP Requests

Next step was running the code, we first tried with HTTP, using port 80 the result is shown below.
Input:



```
Method Example >> GET / PROTOCOL/VERSION
Request Method:
GET / HTTP/1.0
Host (Either Domain Name or IP):
127.0.0.1
port:
80
```

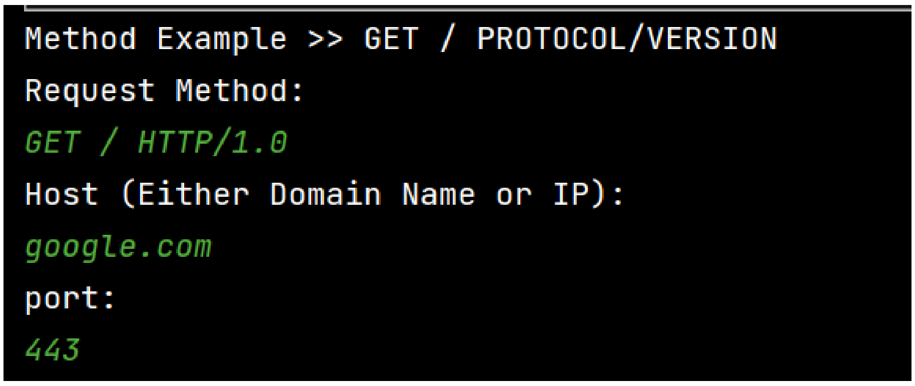
Output:



```
HTTP/1.1 302 Found
Date: Sun, 10 Apr 2022 02:07:18 GMT
Server: Apache/2.4.52 (Win64) OpenSSL/1.1.1m PHP/8.1.2
X-Powered-By: PHP/8.1.2
Location: http:///dashboard/
Content-Length: 134
Connection: close
Content-Type: text/html; charset=UTF-8

<br />
<b>Warning</b>:  Undefined array key "HTTP_HOST" in <b>C:\Users\netoc\Desktop\xamp\h

Process finished with exit code 0
```

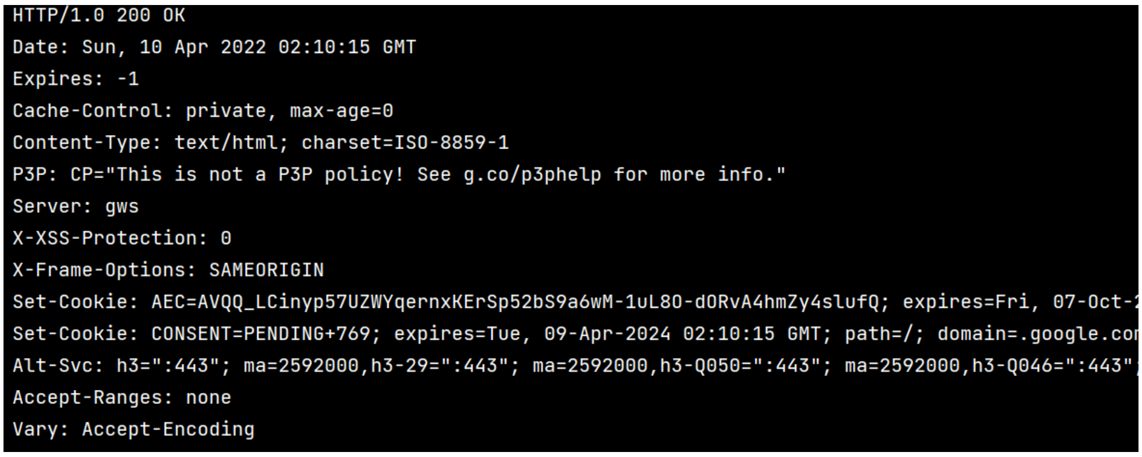
5.2 HTTPS Requests

Now it was time to test the HTTPS, moving to port 443.
Input:



```
Method Example >> GET / PROTOCOL/VERSION
Request Method:
GET / HTTP/1.0
Host (Either Domain Name or IP):
google.com
port:
443
```

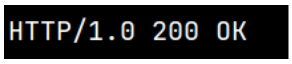
Output:



```
HTTP/1.0 200 OK
Date: Sun, 10 Apr 2022 02:10:15 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: AEC=AVQQ_LCinyp57UZWYqernxKErSp52bS9a6wM-1uL80-dORvA4hmZy4slufQ; expires=Fri, 07-Oct-
Set-Cookie: CONSENT=PENDING+769; expires=Tue, 09-Apr-2024 02:10:15 GMT; path=/; domain=.google.co
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000,h3-Q050=":443"; ma=2592000,h3-Q046=":443"
Accept-Ranges: none
Vary: Accept-Encoding
```

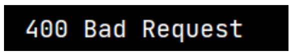
5.3 Different Codes

Types of different codes shown:



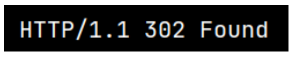
```
HTTP/1.0 200 OK
```

200 OK: It means success.



```
400 Bad Request
```

400 Bad Request: It means the destination doesn't exist, either port or host is wrong.



```
HTTP/1.1 302 Found
```

302 Found: It means you will be redirected to another url.

6 Packet Analysis

Everything is done on the coding side, it's time to take some conclusions and analyze what's happening in the process of requests and responses.

6.1 Setting up Wireshark

We followed the steps shown below to setup our Wireshark:

1. Download and Install WireShark from your browser. (Wireshark · Download)
2. Open WireShark, go to "Capture" in the toolbar and then "Options".
3. As we made it using the same machine for client and server, we used the LoopBack address, so choose the "Adapter for LoopBack traffic capture". (We could use Wi-Fi, but since we want to take a detailed view on the packets, LoopBack Address has way less traffic).

6.2 Web Browser as a Client

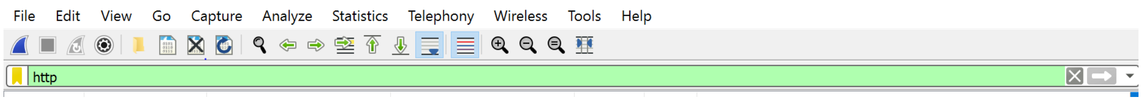
6.2.1 Capture

Assuming that we had wireshark already open and Apache running, we launched a Web Browser and followed the steps:

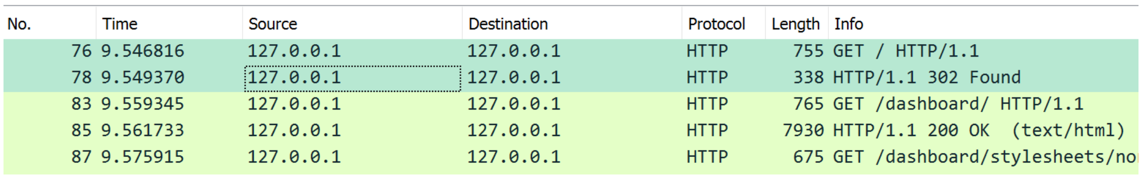
1. Start to capture packets on wireshark (Ctrl + E).
2. Write "127.0.0.1" on the Browser address bar and search it.
3. After waiting for the web page that we have seen before to open, stop capturing packets on wireshark (CTRL + E).

6.2.2 Analyze

Going to the wireshark now we see a lot of different information after the capture part, but for now we just wanna take a look at "HTTP protocol" packets. To filter it write "http" on the line between toolbar and the packets, as it's shown below:

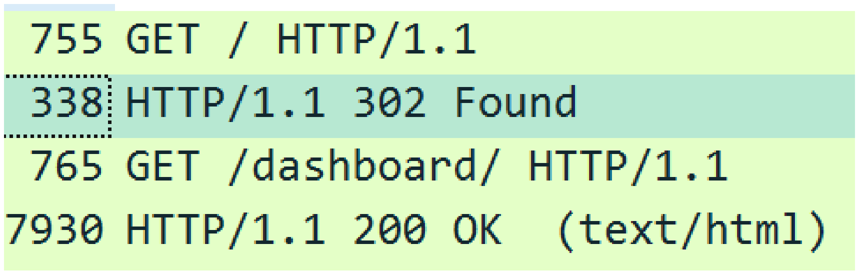


We can only see the HTTP packets by this time, we can gather some information just by looking at the screen. On figure 100 we can see 5 packets, but what can we see besides that?



No.	Time	Source	Destination	Protocol	Length	Info
76	9.546816	127.0.0.1	127.0.0.1	HTTP	755	GET / HTTP/1.1
78	9.549370	127.0.0.1	127.0.0.1	HTTP	338	HTTP/1.1 302 Found
83	9.559345	127.0.0.1	127.0.0.1	HTTP	765	GET /dashboard/ HTTP/1.1
85	9.561733	127.0.0.1	127.0.0.1	HTTP	7930	HTTP/1.1 200 OK (text/html)
87	9.575915	127.0.0.1	127.0.0.1	HTTP	675	GET /dashboard/stylesheets/no

First of all, we can see 2 GET Requests being replied with 2 different answers, one receives an "302 Found" code in the answer and the other receives an "200 OK".

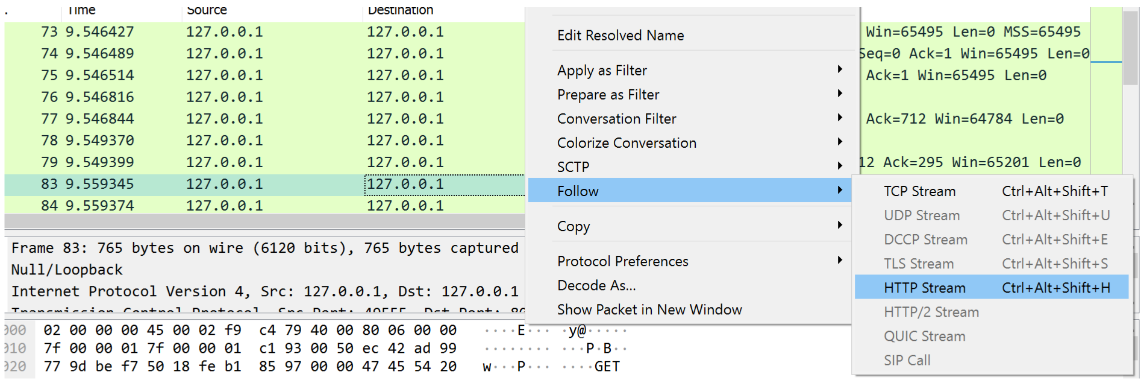


```
 755 GET / HTTP/1.1
 338 HTTP/1.1 302 Found
 765 GET /dashboard/ HTTP/1.1
7930 HTTP/1.1 200 OK  (text/html)
```

As we have seen before the message "302 Found" tell us that we have been redirected, in this case we typed the "127.0.0.1/" so the GET Request try that address and we can see that it sent us to the "/dashboard", because after that it repeats the GET Request but now for the "127.0.0.1/dashboard" and the reply is now "200 OK", so it was a successful connection.

If we take a close look at it, we can see something else in there, the Reply is not immediately after the Request on the column called "No.", there is something in the middle, but is it related to the HTTP?

We won't worry about this for now, let's keep this in mind, but we first need to Analyze these HTTP packets, to make this easier we did something called "follow HTTP stream", right click on the first packet then "Follow" -¿ "HTTP Stream".



```
.   Time          Source          Destination
 73 9.546427      127.0.0.1       127.0.0.1       Win=65495 Len=0 MSS=65495
 74 9.546489      127.0.0.1       127.0.0.1       Seq=0 Ack=1 Win=65495 Len=0
 75 9.546514      127.0.0.1       127.0.0.1       Ack=1 Win=65495 Len=0
 76 9.546816      127.0.0.1       127.0.0.1
 77 9.546844      127.0.0.1       127.0.0.1       Ack=712 Win=64784 Len=0
 78 9.549370      127.0.0.1       127.0.0.1
 79 9.549399      127.0.0.1       127.0.0.1       12 Ack=295 Win=65201 Len=0
 83 9.559345      127.0.0.1       127.0.0.1
 84 9.559374      127.0.0.1       127.0.0.1

Frame 83: 765 bytes on wire (6120 bits), 765 bytes captured
Null/Loopback
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1

000  02 00 00 00 45 00 02 f9  c4 79 40 00 80 06 00 00   ····E···  ·y@·····
010  7f 00 00 01 7f 00 00 01  c1 93 00 50 ec 42 ad 99   ········  ···P·B··
020  77 9d be f7 50 18 fe b1  85 97 00 00 47 45 54 20   w···P···  ····GET
```

Now we have the "conversation" between our client and server, the RED messages are from the Client and the BLUE ones are from the server, let's first analyze the first message of the Client.

```
GET / HTTP/1.1
Host: 127.0.0.1
Connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="100", "Microsoft Edge";v="100"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/100.0.4896.75 Safari/537.36 Edg/100.0.1185.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: pt-PT,pt;q=0.9,pt-BR;q=0.8,en;q=0.7,en-US;q=0.6,en-GB;q=0.5
```

On the first line we can see the method (GET) , the path (/), the protocol (HTTP) and its version (1.1)(Everything below first line has the name of "headers").

On the second line we can see the Host.

On the third line, we have this "keep-alive" message, this means that once the connection is made, until the conversation it's over, we will no longer need more TCP three-way handshakes, because we won't break the connection, it seems obvious that we don't need either "syn" or "syn-ack", but will we need "ack"? Of course we will, and it will be shown soon, we won't get into the other information by now.

Scrolling down past the first Client message we get to the Server reply.

```
HTTP/1.1 302 Found
Date: Sun, 10 Apr 2022 05:12:11 GMT
Server: Apache/2.4.52 (Win64) OpenSSL/1.1.1m PHP/8.1.2
X-Powered-By: PHP/8.1.2
Location: http://127.0.0.1/dashboard/
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

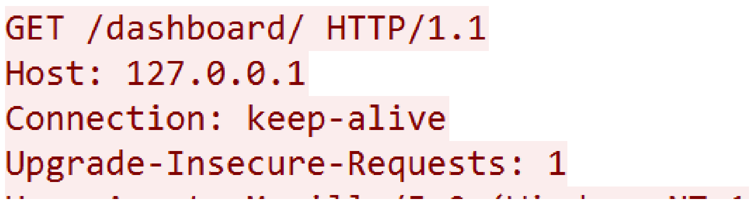
On the first line, we already saw that it is the Protocol and the version, and we can see the reply code too.

On the second line you can see information about the date of the reply.

On the third line you can see the Web Server name, version, coding language and even more.

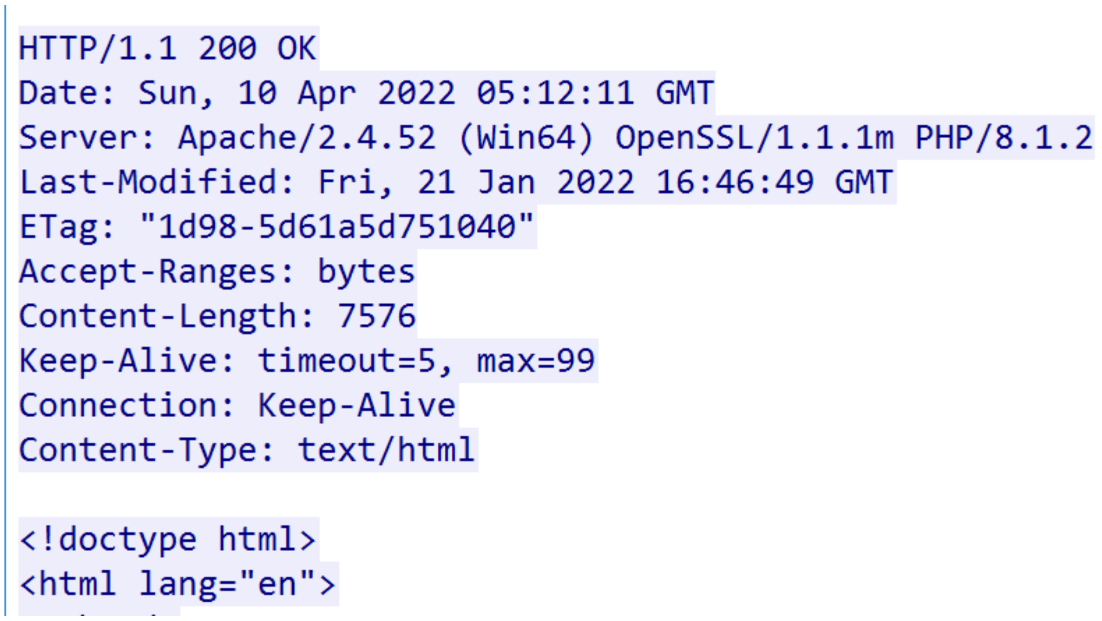
Passing to the fifth line, here you have the address that you have been redirected to, the cause of the "302 Found".

We are just gonna analyze one more message from each one, Client and Browser, we kept scrolling down and the next message was the Get Request.



```
GET /dashboard/ HTTP/1.1
Host: 127.0.0.1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

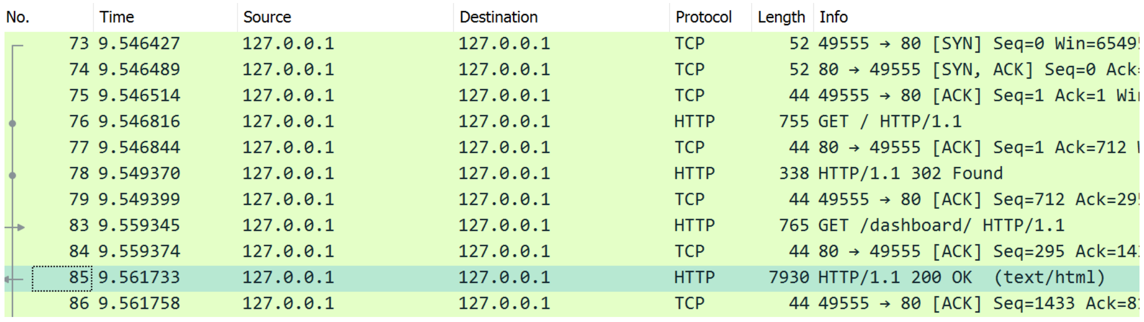
As we saw in the early analyses, the GET Request is done to /dashboard/ now and despite that, everything remains equal, let's see what changed in the next server message.



```
HTTP/1.1 200 OK
Date: Sun, 10 Apr 2022 05:12:11 GMT
Server: Apache/2.4.52 (Win64) OpenSSL/1.1.1m PHP/8.1.2
Last-Modified: Fri, 21 Jan 2022 16:46:49 GMT
ETag: "1d98-5d61a5d751040"
Accept-Ranges: bytes
Content-Length: 7576
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: text/html

<!doctype html>
<html lang="en">
```

In the very first lines, as expected, only the Reply Code changed, but let's take a look at line ten "Content-Type: text/html". It's telling us that the page content we've just Requested the GET is HTML, and after that you will see the HTML Content of the page, we can see the start of it in the printscreen but it's not of our interest to see the rest, just to understand why is it there and how.

Close the HTTP Stream Window, there is something different now.



```
No.     Time        Source      Destination   Protocol  Length  Info
     73 9.546427    127.0.0.1   127.0.0.1     TCP           52 49555 → 80 [SYN] Seq=0 Win=6549
     74 9.546489    127.0.0.1   127.0.0.1     TCP           52 80 → 49555 [SYN, ACK] Seq=0 Ack
     75 9.546514    127.0.0.1   127.0.0.1     TCP           44 49555 → 80 [ACK] Seq=1 Ack=1 Wi
     76 9.546816    127.0.0.1   127.0.0.1     HTTP         755 GET / HTTP/1.1
     77 9.546844    127.0.0.1   127.0.0.1     TCP           44 80 → 49555 [ACK] Seq=1 Ack=712
     78 9.549370    127.0.0.1   127.0.0.1     HTTP         338 HTTP/1.1 302 Found
     79 9.549399    127.0.0.1   127.0.0.1     TCP           44 49555 → 80 [ACK] Seq=712 Ack=29
     83 9.559345    127.0.0.1   127.0.0.1     HTTP         765 GET /dashboard/ HTTP/1.1
     84 9.559374    127.0.0.1   127.0.0.1     TCP           44 80 → 49555 [ACK] Seq=295 Ack=14
     85 9.561733    127.0.0.1   127.0.0.1     HTTP        7930 HTTP/1.1 200 OK  (text/html)
     86 9.561758    127.0.0.1   127.0.0.1     TCP           44 49555 → 80 [ACK] Seq=1433 Ack=8
```

We now have the HTTP and the TCP packets that are responsible for this conversation, first 3 packets are the beginning of the connection, this is when the three-way handshake happens (SYN-SYN-ACK-ACK) and it's never done again because, as we saw, the connection type is "Keep-alive".

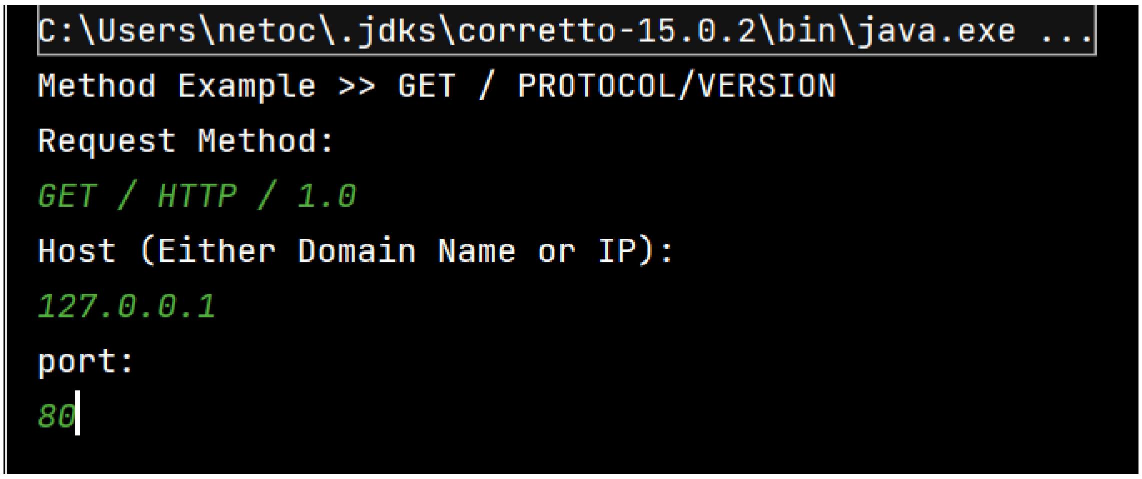
If we take a look at the GETs Requests No.76 && 83, we see that there is in fact something between them and their replies and yes it is related. We can see that this middle packet is a "TCP ACK", what does it do? As the word says, it sends acknowledgement messages between the Client and the Server, even though we don't need any "SYN" synchronization anymore, because the connection will be kept, we need to make sure that the receiver is aware that it will receive something.

6.3 Web Client in Kotlin

6.3.1 Capture

Assuming that we had wireshark already open and Apache running, we opened the Web Client Project and then followed the steps:

1. Start capturing (Ctrl + E).
2. Run the Http_Client.kt.
3. Write the asked inputs

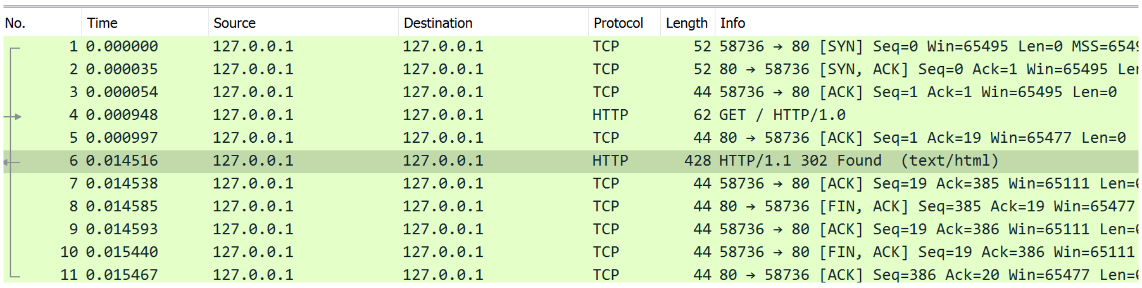


```
C:\Users\netoc\.jdks\corretto-15.0.2\bin\java.exe ...
Method Example >> GET / PROTOCOL/VERSION
Request Method:
GET / HTTP / 1.0
Host (Either Domain Name or IP):
127.0.0.1
port:
80
```

4. Stop Capturing on WireShark (Ctrl + E).

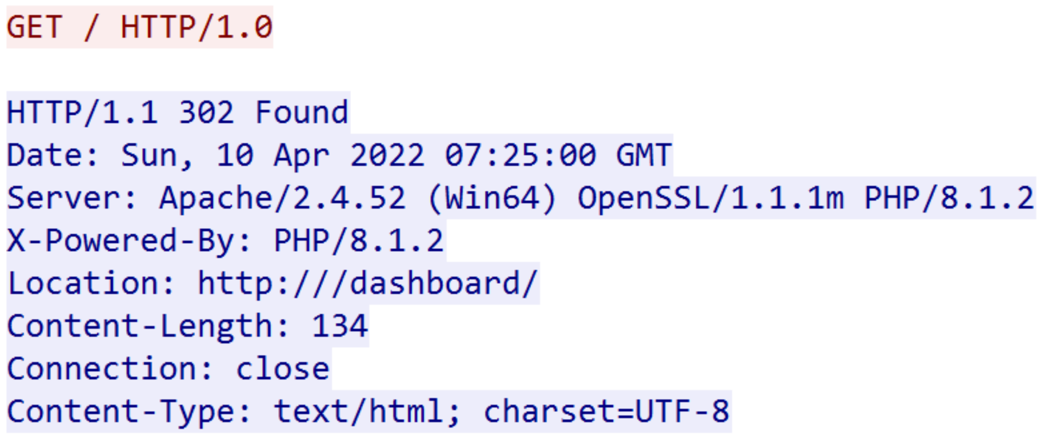
6.3.2 Analyzes

This time we don't have a lot of packets and we already know how TCP and HTTP are related so we didn't filter the HTTP packets.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	127.0.0.1	127.0.0.1	TCP	52	58736 → 80 [SYN] Seq=0 Win=65495 Len=0 MSS=654
2	0.000035	127.0.0.1	127.0.0.1	TCP	52	80 → 58736 [SYN, ACK] Seq=0 Ack=1 Win=65495 Le
3	0.000054	127.0.0.1	127.0.0.1	TCP	44	58736 → 80 [ACK] Seq=1 Ack=1 Win=65495 Len=0
4	0.000948	127.0.0.1	127.0.0.1	HTTP	62	GET / HTTP/1.0
5	0.000997	127.0.0.1	127.0.0.1	TCP	44	80 → 58736 [ACK] Seq=1 Ack=19 Win=65477 Len=0
6	0.014516	127.0.0.1	127.0.0.1	HTTP	428	HTTP/1.1 302 Found (text/html)
7	0.014538	127.0.0.1	127.0.0.1	TCP	44	58736 → 80 [ACK] Seq=19 Ack=385 Win=65111 Len=
8	0.014585	127.0.0.1	127.0.0.1	TCP	44	80 → 58736 [FIN, ACK] Seq=385 Ack=19 Win=65477
9	0.014593	127.0.0.1	127.0.0.1	TCP	44	58736 → 80 [ACK] Seq=19 Ack=386 Win=65111 Len=
10	0.015440	127.0.0.1	127.0.0.1	TCP	44	58736 → 80 [FIN, ACK] Seq=19 Ack=386 Win=65111
11	0.015467	127.0.0.1	127.0.0.1	TCP	44	80 → 58736 [ACK] Seq=386 Ack=20 Win=65477 Len=

As we can see, our Web Client only has one GET Request and its reply is "302 Found", but this time we already know why. It is redirecting us to the /dashboard, but even knowing it, we can follow the HTTP Stream, expecting it to be way smaller now.

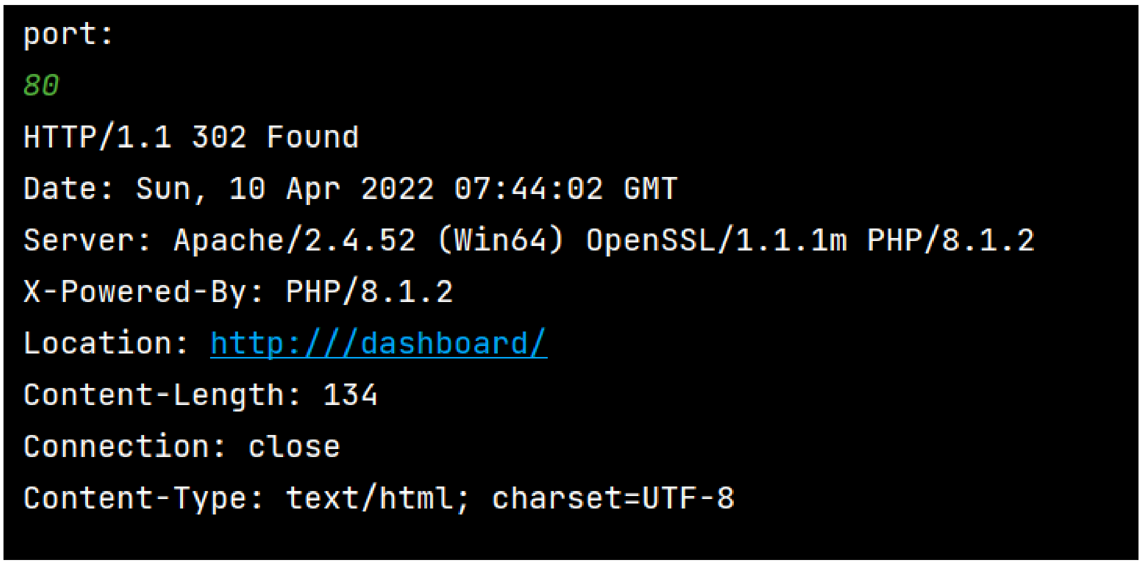


```
GET / HTTP/1.0

HTTP/1.1 302 Found
Date: Sun, 10 Apr 2022 07:25:00 GMT
Server: Apache/2.4.52 (Win64) OpenSSL/1.1.1m PHP/8.1.2
X-Powered-By: PHP/8.1.2
Location: http:///dashboard/
Content-Length: 134
Connection: close
Content-Type: text/html; charset=UTF-8
```

Our Client request is simply a GET Request to the host given directly in the input of the Web Client with the Protocol and its version as well as the path.

On the other side the Server reply has the same code as with the web browser ("302 Found") trying to redirect us to the /dashboard/, the Web Server is the same so its version keeps the same too. What changed was the Connection, it's not Keep-Alive anymore, so we have to do each Request separately, taking us more time.

Looking again to the Web Client you can see that everything we are seeing in the Server Reply is being shown in the terminal.



```
port:
80
HTTP/1.1 302 Found
Date: Sun, 10 Apr 2022 07:44:02 GMT
Server: Apache/2.4.52 (Win64) OpenSSL/1.1.1m PHP/8.1.2
X-Powered-By: PHP/8.1.2
Location: http:///dashboard/
Content-Length: 134
Connection: close
Content-Type: text/html; charset=UTF-8
```

7 Conclusion

To conclude, we learnt more about HTTP vs HTTPS, socket and Encryption concepts.

We noted too that Browser based Clients can Keep-Alive a connection, this enables multiple HTTP GET requests without repeating the three way handshake, only using TCP Protocol to transport Acknowledgement messages between the Server and the Client. Reading the packet content and HTTP stream we saw the behavior of the Client and the Server through the different types of Response Codes.