Assignment 2
Upper Layers of the OSI-Model

Version: August 21, 2022

Prerequisites

1. The assigned readings in module 2 on Canvas
2. Lecture videos from Canvas
3. Running and understanding the following examples from the GitHub repo (Network/SimpleGrabHttpURL, Network/SimpleGrabURL, Network/HTTP-JSON)
4. Setup of a second device (second computer, AWS EC2, Raspberry PI) – see Canvas for details
5. Videos, tutorials about Wireshark
6. Understanding about lower levels of Networking (module 1)

Learning outcomes of this assignment are:

1. Understanding the upper layer Network protocols
2. Understanding how to use the upper layer protocols, e.g. HTTP, SMTP, FTP
3. Compare the different traffic for different protocols when running a client/server application on two different systems
4. Understanding how to use the command line

1 Understanding HTTP (20 points)

For this you will only need your web browser and Wireshark (Wireshark only for taking a look at things yourself). To understand GET HTTP requests a little better we want to do a couple of GET requests through an API. In this case, GitHub. The API basically works by nesting topics, so sometimes you need to do a base call to get data and you will use the result to do the next call, etc.

Do the following:

Go to the GitHub Api documentation page. This will give you some information about the API. It might be overwhelming at first, that is ok. The interesting part here is you can make requests directly on your browser and not just from Java, JavaScript etc. For instance we use the List repositories for a user API to get all the public repos from a specific user. In your browser use this URL:

```
https :// api . github .com/users/amehlhase316/repos
```

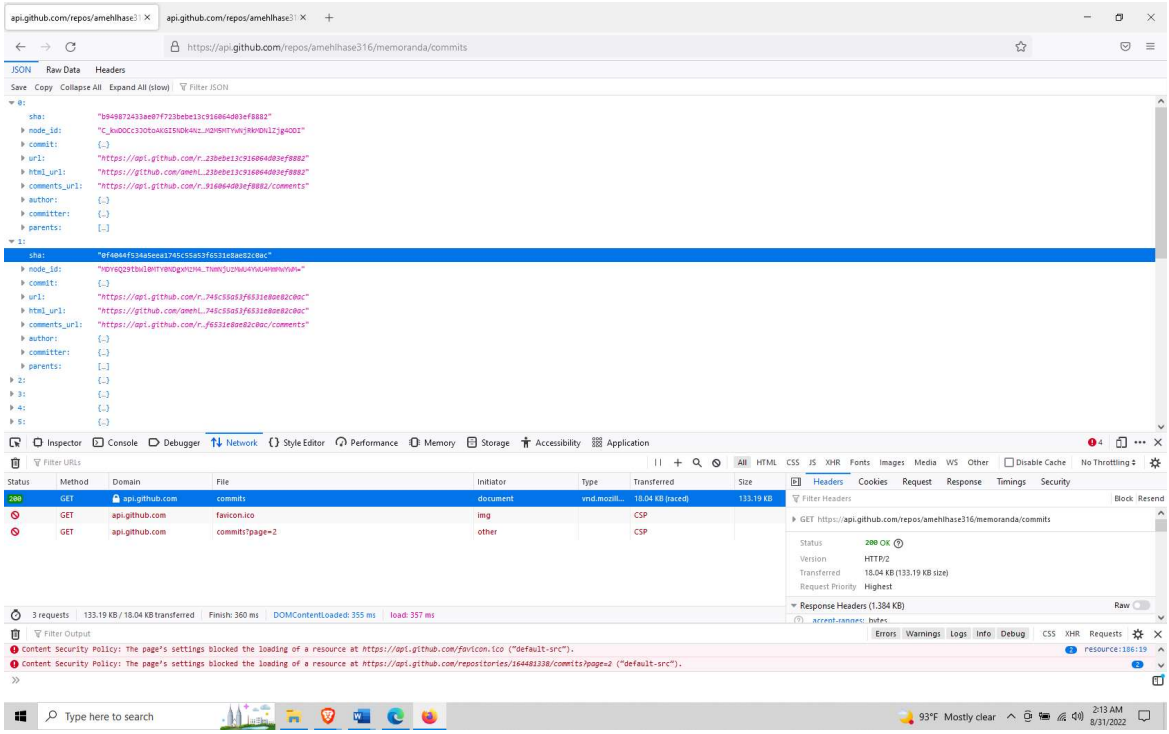
This should show you a JSON of all my public repos (you can of course use a different username if you like, my public profile is boring).
Next, we want to look at one specific repo, we will use the Get a repository API method for that, go to:

```
https :// api . github .com/repos/amehlhase316/memoranda
```

This will give us a JSON about the memoranda project where I am the owner (or another project you want to use of course).
Now, find and run another call that gets all the commits on the default branch for one of the repositories you chose. Any public repo that has some branches and some commits on their branches is fine.

Deliverable (5 points): Paste the url you used into your document and take a screenshot and add it to your document (the screenshot should show the call you made and the JSON result – if the JSON is too big, partially showing it is ok).

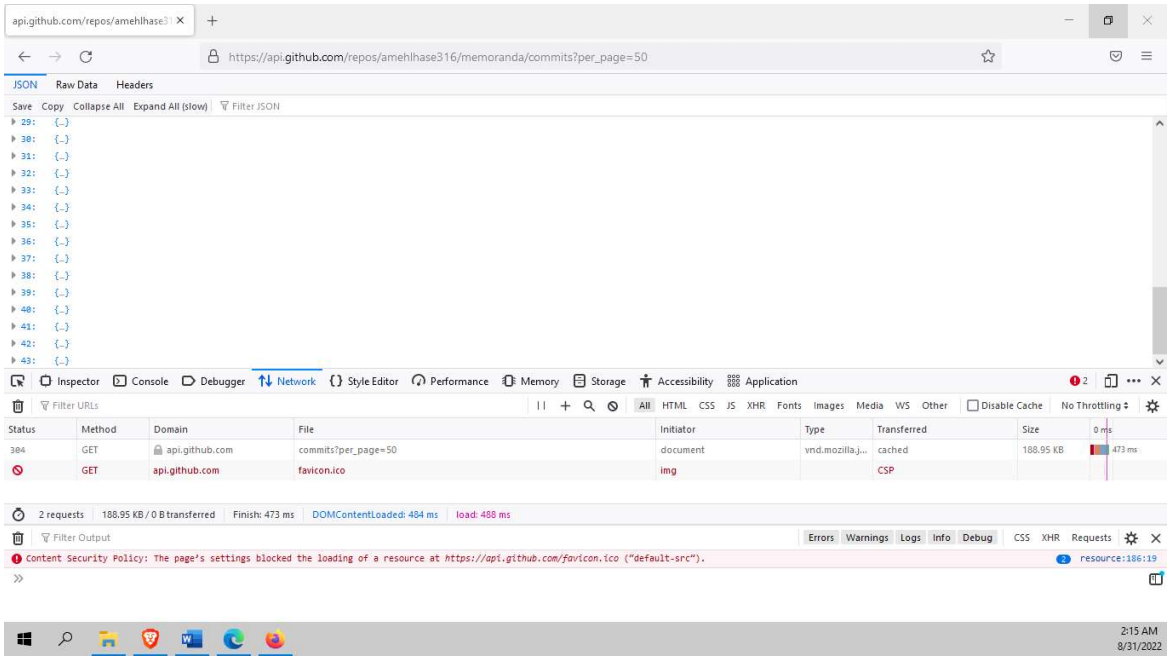
https://api.github.com/repos/amehlhase316/memoranda/commits



Do another call. In this call, add GET parameters to the call from above that specifies a specific branch (so not the default) and set the per_page limit to 50 (so that now 50 commits instead of just 30 are shown). Usually APIs limit the response size and will return one page and you can call the next one if you need more data or you increase the page limit as we do here for simplicity. If you are stuck on this, go to the documentation and try to understand how to do it.

Deliverable (5 points): Paste the url you used into your document and take a screenshot of your browser and add it to the document.

https://api.github.com/repos/amehlhase316/memoranda/commits?per_page=50



Deliverable: Answer the following in your document (10 points):

1. Explain the specific API calls you used.

   The first API call was a simple get request filtering for only the commits in the memoranda repo.

   The second command was the same as the first but this time we overrode the default 30 results and instead requested the browser show 50 results per page. Although we requested 50, we only saw 43 total because this is the total number of available commits on the page.

2. Explain the difference between stateless and a stateful communication.

   Stateless communication is when there is no data stored where in stateful communication, data is stored. The storage of data can be useful because it can help create a more tailored user experience upon launching a page.

Now you should take a look at Wireshark and check the communication that was going on with your calls. You do not have to document anything for me but I advise you to take a detailed look and do your best to understand all the traffic you generated.

2 Setup your second system and run Server on it (65 points)

For this part you will need to setup your second machine (which you should have already done). See setup Page on Canvas. We will call your local machine "first machine" and your second one (AWS, PI, extra computer) "second machine" in this document.

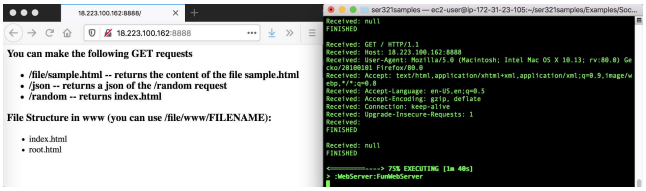
2.1 Getting the sample code onto your systems (should be done already)

This is something that should be done already but in case you did not do this yet. So now that you have your second machine setup you should make sure the GitHub repo with all of the examples is available on that second machine and first machine. I would advise you to fork the given repository and then clone that repo on all the machines you want to work on, so you can make changes and still commit, push and pull. This fork will be public, thus it is not for your assignment changes but just for "playing" with the examples.
You can of course also download the zip and add it to both systems (but then you won't be able to pull updates easily). If you have not worked with GitHub before I advise you to go to the GitHub review page on Canvas.

2.2 Running a simple Java WebServer (10 points)

You now need to work with the *Socket/WebServer/*. Copy this folder into YOUR assignment 2 folder of your private repo before you make any major changes.
We want to run the *FunWebServer* task from Gradle in this example.
The Server should run on your second machine (AWS). You should also start Wireshark on your first machine again if it does not run anymore.
When your Server runs on your second machine go to a Web Browser on your first machine and go to: *ipOfSecondMachine:9000* (you can also change the port if you like of course, the port you use must be opened up for TCP traffic on AWS). If everything works as intended this should bring up a web page.

Delivearble: 10 points: Take a screen shot of your web browser showing the *ipOfSecondMachine:9000* and the web page. You should take a screen shot of your second machine (can be two separate screen shots or just one) and add it to your document under Task 2. The screen shot should look something like this:



You can make the following GET requests

- **/file/sample.html -- returns the content of the file sample.html**
- **/json -- returns a json of the /random request**
- **/random -- returns index.html**

File Structure in www (you can use /file/www/FILENAME):

- index.html
- root.html

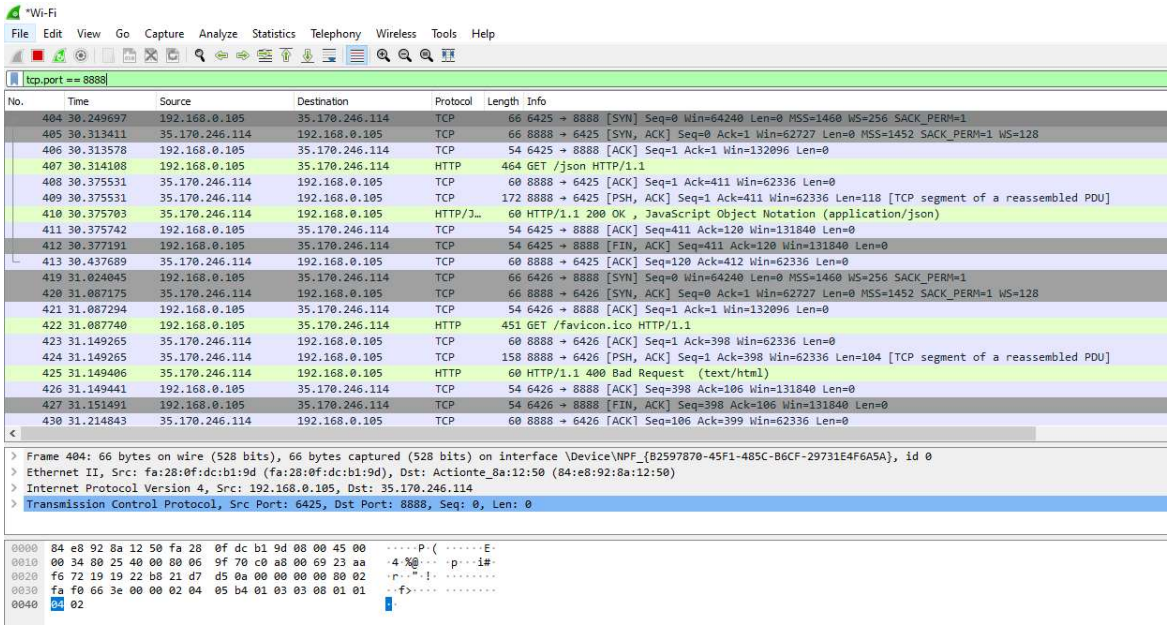
```
[ec2-user@ip-172-31-95-53 WebServer]$ gradle FunWebServer

> Task :FunWebServer
Received: GET / HTTP/1.1
Received: Host: 35.170.246.114:8888
Received: Connection: keep-alive
Received: Upgrade-Insecure-Requests: 1
Received: User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.102 Safari/537.36
Received: Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
Received: Sec-GPC: 1
Received: Accept-Language: en-US,en
Received: Accept-Encoding: gzip, deflate
Received:
FINISHED PARSING HEADER

<=========----> 75% EXECUTING [3m 12s]
> :FunWebServer
```

2.3 Analyze what happens (10 points)

Wireshark should still be running in the background. Go to Wireshark and create a filter so that it shows your Network traffic to and from your WebServer. Take a screenshot of your Wireshark capture and add it to your document.



Delivearble: Now in your document answer the following (1-2 points each):

1. What filter did you use? Explain why you chose that filter.

   I chose tcp.port==8888

   I chose this because I modified the program to run on port 8888 because it was a free port I had in AWS. Filtering for tcp port 8888 allowed me to remove traffic that was not caused by the web server.

2. What happens when you are on /random and click the "Random" button compared to the browser refresh (you can also use the command line output that the WebServer generates to answer this)?

   Clicking random:

```
<==========----> 75% EXECUTING [45m 8s]
Received: GET /json HTTP/1.1
Received: Host: 35.170.246.114:8888
Received: Connection: keep-alive
Received: User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.102 Safari/537.36
Received: Accept: */*
Received: Sec-GPC: 1
Received: Accept-Language: en-US,en
Received: Referer: http://35.170.246.114:8888/random
Received: Accept-Encoding: gzip, deflate
Received:
FINISHED PARSING HEADER

<==========----> 75% EXECUTING [45m 30s]
> :FunWebServer
```

   Browser refresh:

```
Received: GET /random HTTP/1.1
Received: Host: 35.170.246.114:8888
Received: Connection: keep-alive
Received: Cache-Control: max-age=0
Received: Upgrade-Insecure-Requests: 1
Received: User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.102 Safari/537.36
Received: Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
Received: Sec-GPC: 1
Received: Accept-Language: en-US,en
Received: Accept-Encoding: gzip, deflate
Received:
FINISHED PARSING HEADER

Received: GET /json HTTP/1.1
Received: Host: 35.170.246.114:8888
Received: Connection: keep-alive
Received: User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.5112.102 Safari/537.36
Received: Accept: */*
Received: Sec-GPC: 1
Received: Accept-Language: en-US,en
Received: Referer: http://35.170.246.114:8888/random
Received: Accept-Encoding: gzip, deflate
```

When I refresh the page, it reloads the page using get/random whereas when I click random, it does not need to reload the page, only get a new image.

3. What kinds of response codes are you able to get through different requests to your server?

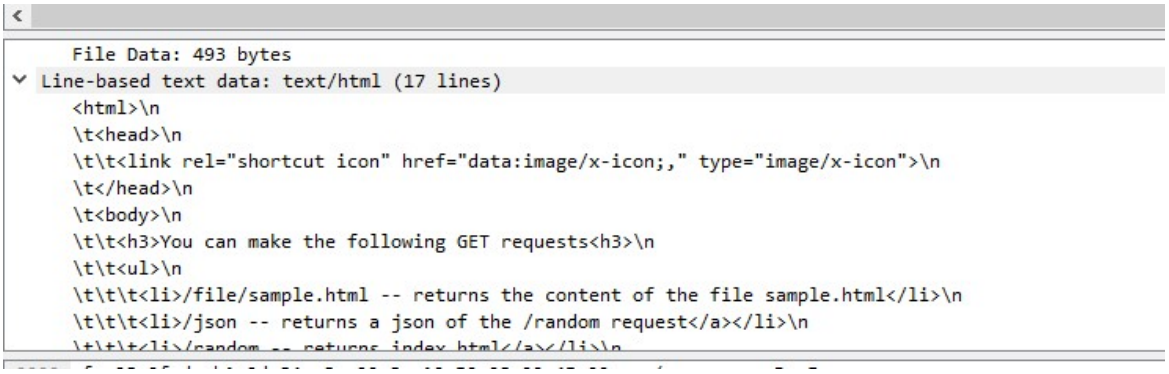
   I filtered for http in wireshark and got codes 200 and 304.

4. Explain the response codes you get and why you get them.

   The 200 codes were because the request was successful.

   The 304 code is a redirection code.

5. When you do a *ipOfSecondMachine:9000* take a look what Wireshark generates as a server response. Are you able to find the data that the server sends back to you?
   Yes, here is a screen shot of it.



6. Based on the above question explain why HTTPs is now more common than HTTP.

   Https is more common because it is more secure. It uses encryption to ensure that the data is secured and not plain text.

7. What port does the server listen to for HTTP requests in our case and is that the most common port for HTTP?

In our case, I used port 8888. I had this port free on AWS. The most common port for http is port 80 and https is 443.

8. What local port is used when sending different requests to the WebServer? How does it differ to the traffic to your SMTP server from part 1?

   If we had used SMTP we should have used port 25 that is the most common but here we are using HTTP which I ran on port 8888 but often HTTP is ran on port 80.

2.4 Setting up a "real" Web server (10 points)

Stop your Java Web server and do the following while on your second machine (it might need to be apt-get depending on your system setup).
Lets setup nginx so you have a real Web server running: Run the following:

```
sudo amazon−linux−extras install nginx1 −−> when prompted , sudo nginx −−> to start    type y
the web server
```

Then change the config file for the server. Note: All changes will be in the server ... section. (On the Pi the config file looked different sometimes, just a warning).

```
sudo vim /etc/nginx/nginx . conf
```

You need to enter the server_name and a location block into your config file. It should look something like this (with the correct ip from your host and port you used in the Java file) – this is only a snippet but shows the part that you need to change.

```
....
server {
        server_name          18.223.100.162;                          #CHANGE IP HERE
       root                    /usr/share/nginx/html ;
          location / {                                                  #Add location          block with
                    proxy_pass http :// localhost :9000/; }
     # Load configuration f i l e s for the default server block . include /etc/nginx/ default .d/∗. conf
      ;

        # redirect      server     error      pages to the      static      page /40x . html
     #
      error_page 404 /404.html ; location = /40x . html
           {
     } .
...
```

correct port

Now reload nginx with the configuration updates by using:

```
  sudo  nginx  −s  reload
```

Your web server is running and your port 80 traffic will be re-directed to the port you specified in *location block* above.
Start your Java funWebServer again.

Make sure Wireshark is running again. Now go to your browswer again. The url you need to use should change slightly now with the real webserver. Make sure that it is different and that you understand why.

Delivearble: Now in your document answer the following:

1. What is the URL that you can use now to reach the main page?
   We can now use the IP address without the port number.

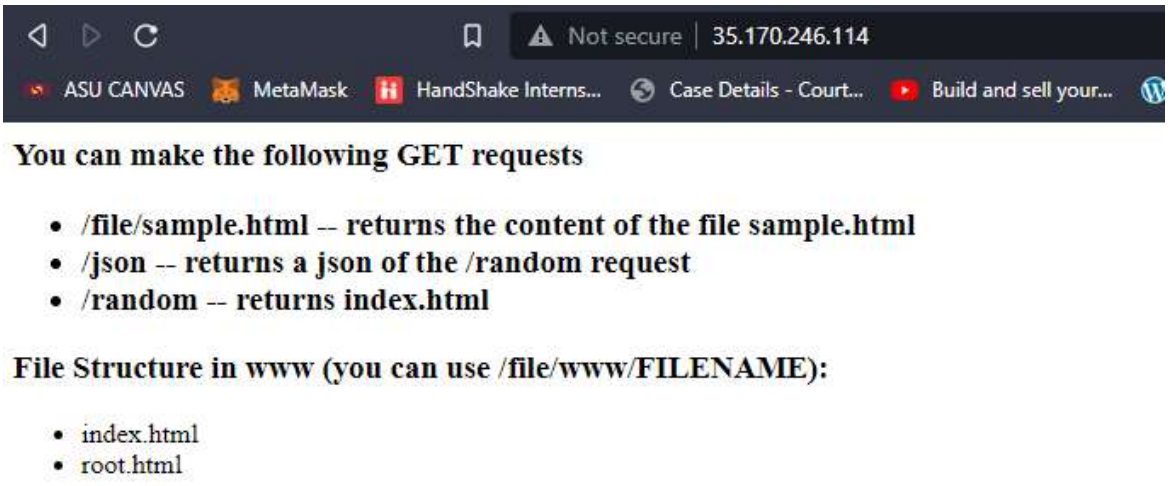
2. Check your traffic to your Webserver. What port is the traffic going to now? Is it the same as previously used or is it and should it be different?

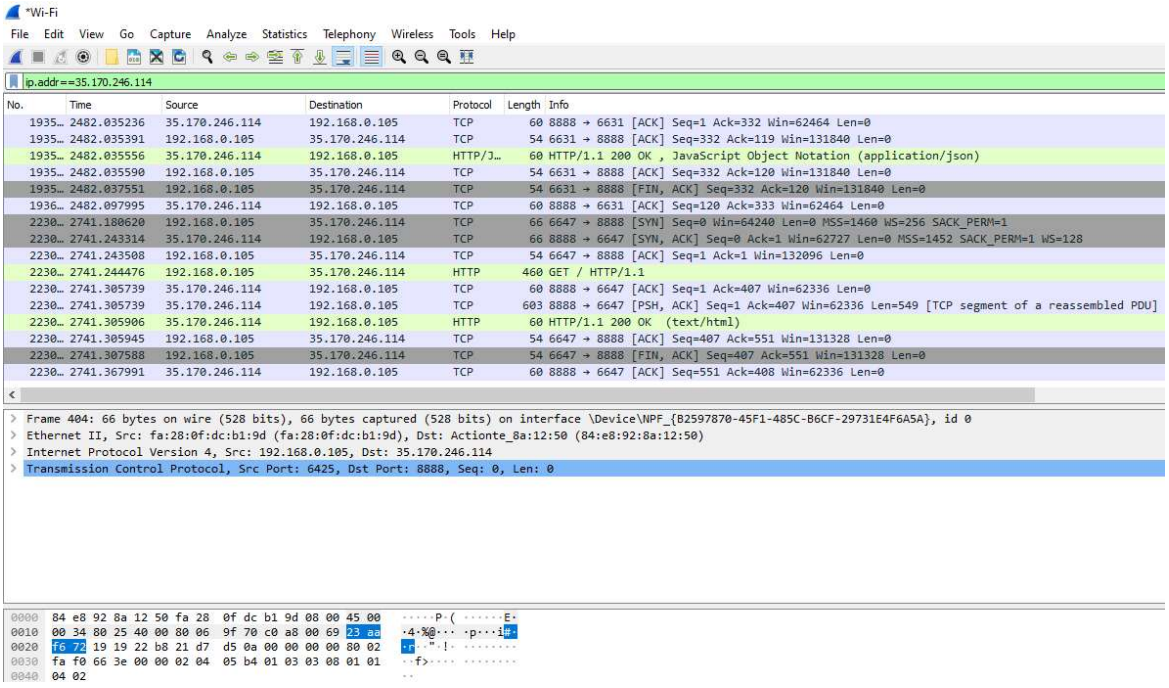
   Before we were sending our traffic to port 6425, now we are sending it to port 6647.

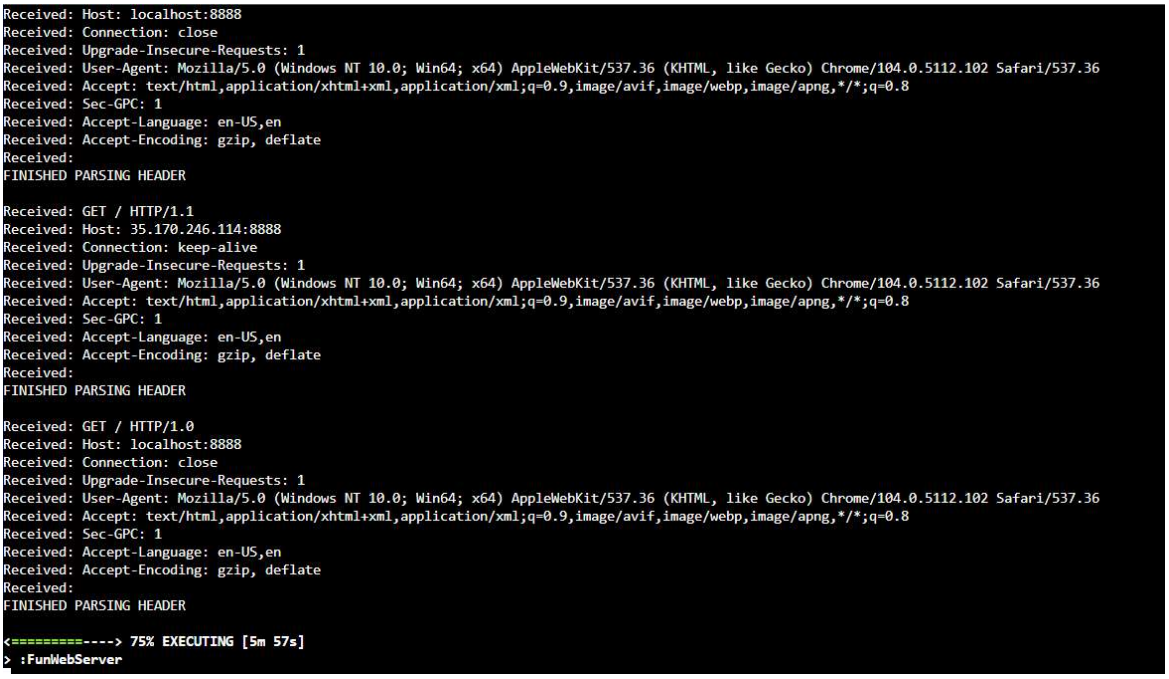
3. Is it still HTTP or is it now HTTPs? Why?

   We are still using HTTP, we have not added an SSL certificate to secure the transaction and make it HTTPs.

4. Take a screen shot of your Web browser, your second machine and also showing the port on Wireshark, similar to the screen shot you took before (but also with Wireshark) and add it to your document for this task. If we do not see that you can now get to the webserver with the "different URL" we will not see that you actually setup the server correctly so make sure that shows up if you want points for this task.







If you like you can stop your nginx again:

```
sudo nginx -s stop
```

Should bring you back to the stage you had without your nginx WebServer.

2.5 Brownie points/Extra Credit (5 points extra) – just this one section: Setting up HTTPs

This did not work on every instance for some reason, thus extra credit if you can make it work. Only attempt if you are ready for some extra steps and some research!! You can think about creating a copy of the current nginx config file in case you want to go back and this would make things easier later on. Start your nginx again

```
sudo   nginx
```

Now, to make our traffic secure we want our communication to go through HTTPs. This will need some more work and some additional steps but worth it to understand what is actually going on.
Work on your second machine for this.
Get a Domain, here we use DuckDNS - you can use something else if you like. In the end we just get a Domain Name which will get resolved to your IP address (remember DNS from the lecture):
In a Broswer visit DuckDNS (your main machine), then

1. login with whatever method you prefer
2. enter a sub domain –> click "add domain" 3. select "install" from the

   top navigation bar

4. Duck DNS/Install:
   - Choose "linux cron"
   - Select the domain you created
   - Go to your second maching & follow the linux cron instructions Note: When saving the crontab, use (esc –> :wq! –> Enter) instead of the instruction's (CTRL+o –> CTRL+x).

Your Domain is setup, now we can continue with the certificate setup. On your second machine run the command to change your nginx config info:

```
sudo vim /etc/nginx/nginx . conf
```

Change the server name from the IP you had to your newly created DuckDNS domain name. E.g., ser321.duckdns.org. Save the changes and reload nginx. When you go to DuckDNS.org now you should see your domain name listed and it should now show the IP from your second machine as the IP address. So your domain "perfectname.duckdns.org" gets resolved to your machines IP address.
Now, you should already be able to use *yourdomain.duckdns.org* to reach your server (still as HTTP and of course after starting your Java program again). Stop the Java program before continuing.
We need to setup a certificate, we can use Certbot. Run

```
sudo yum install −y certbot python2−certbot−nginx sudo certbot
```

Press 2 for nginx. This should bring you to some setup questions.
Lets finish setting up the certificate, run:

```
sudo /usr/ local /bin/certbot −−nginx −−debug
```

When prompted –> type y
When prompted –> enter an email address
When prompted –> type a to agree
When prompted –> type n to decline emails
When prompted –> type the number associated with your domain (it is listed above this prompt)
When prompted –> type 2 to configure a redirect

Your certificate should be all setup.
Now, you should be able to go to *perfectname.duckdns.org* and get to your website through HTTPs. Check on Wireshark and on the Browser you should see that your traffic is going through HTTPs now.
Take a screen shot of your Web browser, your second machine and also showing the port on Wireshark, similar to the screen shot you took before (but also with Wireshark) and add it to your document. We need to see that it now shows HTTPS in your broswer to receive the extra credit points.

Delivearble: Now in your document answer the following:

1. What port is your traffic going through now?

2. Can you still find the plain text responses as before with HTTP?

If you want to stop the nginx just call

```
sudo nginx –s stop
```

Your domain should not work anymore and you should only be able to use the ip address to access your Java server again.
OPTIONAL, in case you want to remove the certificate (which you can do but when nginx is stopped you will only have the "normal traffic" again anyway).
Removing the certificate so only the nginx WebServer works, step one is to remove the certificate, step two to update the nginx config, which is easy if you copied the file back before you started with HTTPS and you can just replace it now. If you did not copy it then open it and remove everything related to the certificate and the port 443. Also if you do not want to use the Domain anymore the server name do:

```
/usr/ local /bin/certbot−auto revoke −−cert−name YOUR_DOMAIN sudo vim /etc/nginx/nginx
. conf
```

DONE.
This ends the "Brownie points/Extra Credit" section and the rest are normal points.

2.6 Some programming on your WebServer (50 points)

This should be done no matter if you have nginx running or not! Working without certificate might be better so you can read the traffic on Wireshark if you need to. I advise you to spend some time on the Webserver code and play around so you understand what happens and how things work.
In the WebServer code you will see a couple todos, which are the things you need to implement and some further details will be explained here. These are the changes that you need to

implement on your private repo not the example repo. You can definitely use a JSON library, look at the JSON examples in the repo mentioned at the beginning of the document how to include a JSON library into your Java and Gradle files.

2.6.1 Multiply (8 points)

Check out the *if* for the *multiply* case. This is a normal GET request that gets two parameters. Your task is now to add some error handling into it in case the user does not provide the correct inputs, or just one input etc. You can handle it as you see fit, e.g. set default values, just print a message but your server should not crash and it should respond in a good way (true for the whole assignment, DO NOT have your server crash).
Important is that you create an appropriate header response with a good error code for that case. You can find explanations about error codes here: https://developer.mozilla.org/enUS/docs/Web/HTTP/Status.
Explain in your document what you decided to do and which error code you decided to use and why. DO NOT just catch any error and send out just one error message. The error message should be specific to the error you caught.

I chose to use 400 error codes for when a user enters incorrect information and 200 codes for when the request was processed successfully per the guidelines set forth on Mozilla.org.

Example protocol, you can change the responses to what you want to send back and you should add on to the status codes.

multiply request

request host :PORT/multiply?num1=1&num2=2

Path:

```
multiply
```

Body Parameter (you can make the parameters optional as well and use default values if they are not provided):

```
num1 ( int ) – required num2 ( int ) –
required
```

response

This is just an example and not complete, since the error handling will be your job! OK Response (200) - Example

```
"Result is: " + result
```

HTTP response status codes

```
200 –– OK, multiply happened correctly xxx –– Error1 xxx
–– Error2
```

2.6.2 GitHub (10 points)

You see another *if* statement waiting for *github*?. In there you see a fetch that will pull information from GitHub. This part is about parsing a JSON response and understanding this response.
Implement your webserver so that when calling
host : *PORT/github*?*query* = *users/amehlhase*316*/repos* you should get a response with all my public repos (not a lot). You should parse that JSON in your code and respond with some data. The data you should return is the full_name of the repos, the ids of the repos, login of the owner of each repo. Go by what this assignment wants displayed, the given code in the example repo says something else.
The webpage should display this information in some way (does not have to be pretty). Make sure you include good error handling and that we cannot crash your server with wrong request (which we will try). You should include good error codes, so not just one big try and catch!
Example protocol, you can change the responses to what you want to send back and you should add on to the status codes.

github request

request host :PORT/github?query=users/amehlhase316/repos

Path:

```
github
```

Body Parameter:

```
query ( String ) – required // This Parameter is the
one that will be given to the real GitHub API
```

response This is just an example and not complete, since the error handling will be your job!
OK Response (200)
Example (you can of course format this differently to send back to the HTML page)

```
[
  {" fullname ":  "repoName" ,
"  id  ":  1  ,  "
loginname ":  "ownerName"
} ,
  {" fullname ":  "repoName2" ,
 " id ":  2 ,  "ownerName2"
  " loginname ":
}
]
```

HTTP response status codes 200 – OK,
the data could be fetched xxx – Error1
xxx – Error2

2.6.3 Make your own requests (15 points - 7.5 for each request)

In your webserver add two more request types that the webserver can handle. This should be similar to the multiply or github request. You can do any request you like but they should fulfill the following requirements:

1. the request should be a bit more than just a new version of the multiply (e.g. do not just do an add/subtract/divide)
2. the two requests you come up with should not be almost the same thing
3. the request should get at least two argument and use these arguments for something
4. the request should have proper error handling, we should not be able to crash your server with wrong input or wrong calls and the error message should be appropriate (good error codes)
5. You should explain your request on the main page that opens when going to your server and give an example how we can use your request, provide the exact url with example data that we can just copy and paste

2.6.4 Explain the protocol you designed for your requests (10 points - 5 for each request)

For the requests you created in 2.6.3, you should have come up with some kind of protocol how the request is called and what it returns and which error codes it returns. Write down this protocol and explain it in detail, you can write it in a similar way as I did above.

I chose to use 400 error codes when a user enters incorrect data and 200 codes when a request was processed successfully. I also added conditional statements to provide tailored feedback to the user as to what they did incorrectly.

2.6.5 Webserver for everyone (5 points)

Now, figure out how you can keep your webserver running even when closing the terminal window to AWS. Then post the link to your server with your public IP address and port to Slack #servers.

2.6.6 Test other webservers (2 points)

Test 2 other servers that are provided in the #servers channel and make a valuable comment on each one you test (you can do this up to 2 days after the due date). No valuable comment, no points.

3 Submission

Submit your link to your GitHub repo Assignment2 folder on Canvas.
Throw your PDF *UpperLayers_asurite.pdf* with all your answers, explanations, and screenshort into your Assignment 2 folder. No Wireshark captures needed.
Now, add the webserver directory into the Assignment2 folder as well. Call the directory "Webserver" and it should have all necessary files in that folder for us to run your code and to see your implementation. You should leave the port at 9000. We will run the code through going to your Webserver directory and call *gradle FunWebServer*. If this does not work you will not receive points for the server!!! So things should look as follows:

- Assignment2
- UpperLayers_asurite.pdf
- /Webserver
- /src
- /www
- build.gradle
- README.md

Zip this whole Assigment 2 folder and put it on Canvas as well!