

Mammad Mammadov

Principles of Distributed Systems

CRN 10212

Fall 2024

## The Unreliable Server Assignment

### Introduction

For this assignment, we must set up an unreliable server with two routes – **HTTP GET /balance** route and **HTTP GET /getlogs** route.

For **/getbalance**, the server is supposed to return different responses with their respective probability distributions – 20% chance for HTTP status code 408 (timeout), 20% chance for HTTP status code 403 (forbidden), 10% for chance HTTP status code 500 (internal server error), and 50% chance for HTTP status code 200 (OK) with HTML/CSS content should be returned.

For **/getlogs**, logs are supposed to be returned to the client in JSON format.

For this assignment, I used **Java** programming language as I had more experience with it in my other courses. This report will first discuss the server solution and then the client solution. It will also reflect that the server is assumed to be flexible, capable of running on different IP addresses and ports.

### Server Solution

Starting with the *main* method in *Server.java*, by default I have assigned port 8080 and IP address 0.0.0.0 for my server in case no arguments are provided. However, in a usual case, I take the port number and IP address from the arguments provided by the user. I have an *isValidPort()* method to check whether the entered port number is between 0 and 65535 (valid port range) and an *isValidIP()* method to check whether the entered IP address is technically valid. After taking the arguments, I initiate a HTTP Server on the specified IP address and port, then add two different routes - */getbalance* and */getlogs*. I have *BalanceHandler* and *LogsHandler* classes for managing HTTP requests directed to those paths.

To set the custom logic for handling requests and providing responses, *BalanceHandler* and *LogsHandler* classes both implement the *HttpHandler* interface.

Looking at the overridden *handle()* method in *BalanceHandler* class, I first get the combination of an IP address and port number of the client as an *InetSocketAddress* object, then from there get the IP address of the client in human-readable string format and assign it to variable *clientIP*. I also get the timestamp from the *LocalDateTime* class. To satisfy the probability distributions given in the assignment instructions, I generate a random number between 0 and 99, then with *if-else*

statements, check whether it is between 0 and 19 (20%), between 21 and 39 (20%), between 40 and 49 (10%), or between 50 and 99 (50%).

In order to send a HTTP response back to a client, I have a *sendResponse()* method. Inside that method, I send the appropriate HTTP headers and convert the response into a byte array, and write it to an output stream. For different probability distributions, I set different values for the *response* argument of the *sendResponse()* method inside *if-else* blocks.

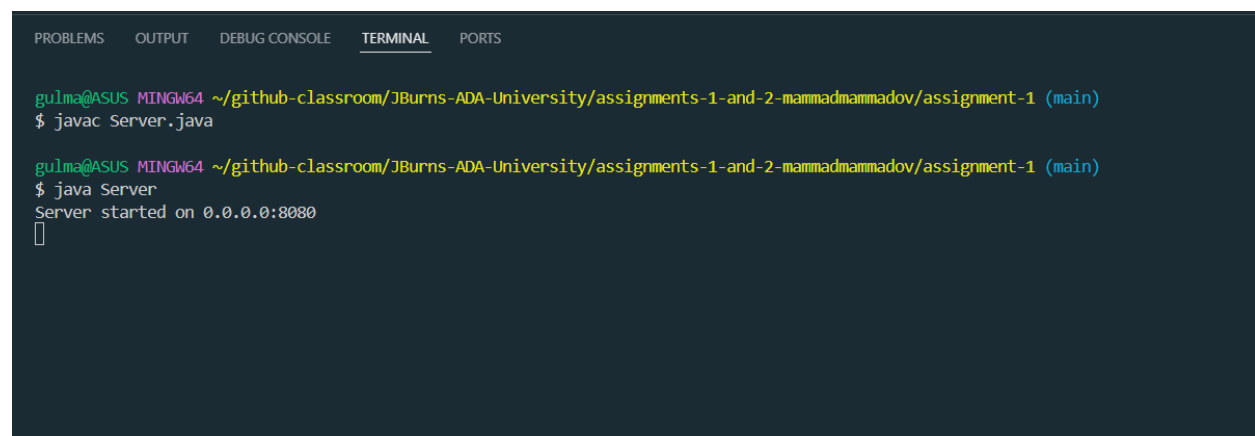
When the HTTP status code turns out to be 200 by chance, I return HTML content, which is stored inside the *response.html* file.

At the end of the *handle()* method inside *BalanceHandler* class, you may see a code line - *logRequest(time, clientIP, resultCode)*; that logs each operation and gives me the chance to talk a little bit about the *logRequest()* method. In this method, I take time, client IP address, and status code as arguments, construct a string from them, and assign it to *String* variable *log*. Then I add that log at the end of my *logs.log* file. For a rare case when the *logs.log* file does not exist, I create it first, then add the log.

Finally discussing the *LogsHandler* class, I aim to present the logs in JSON format to the client when he/she sends a request to */getlogs* endpoint. I read all the logs inside the *logs.log* file and store them in a list one by one. Then I use *StringBuilder* to construct JSON string from those logs. I traverse through the list, split them in 3 components – timestamp, IP address, outcome, and also add necessary punctuation for JSON format.

After converting my *StringBuilder* object to *String*, I finally present my response body to the user.

For the evidence that my Server works successfully, below I provide several pictures. Figure 1 is for the case when I stick to the default values and start my server on IP 0.0.0.0 (since it supports all IP addresses, I show this demonstration via localhost) and on port 8080. For this case, I sent several requests (Figure 2, Figure 3, Figure 4, Figure 5) from the browser to show all the possible pages. Figure 5 is the case for a timeout. Figure 6 shows the respective logs generated.

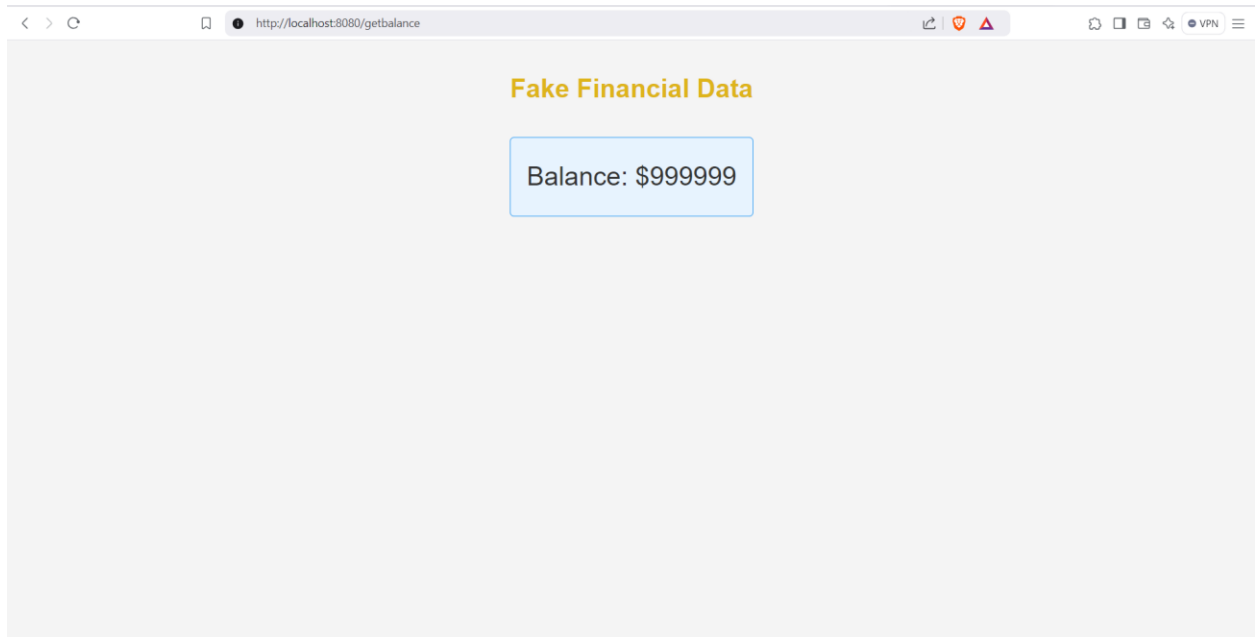


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

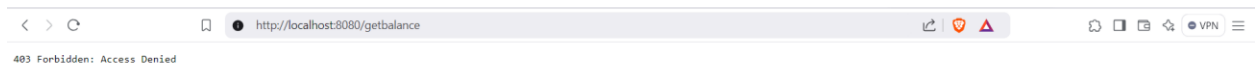
gulma@ASUS MINGW64 ~/github-classroom/JBurns-ADA-University/assignments-1-and-2-mammadmammadov/assignment-1 (main)
$ javac Server.java

gulma@ASUS MINGW64 ~/github-classroom/JBurns-ADA-University/assignments-1-and-2-mammadmammadov/assignment-1 (main)
$ java Server
Server started on 0.0.0.0:8080
[]
```

Figure 1



*Figure 2*



*Figure 3*

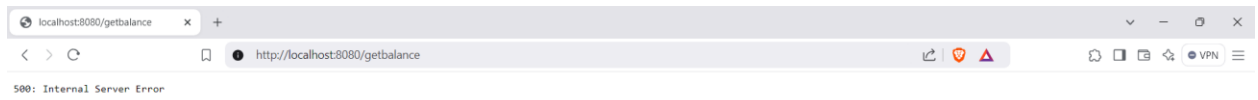


Figure 4



Figure 5

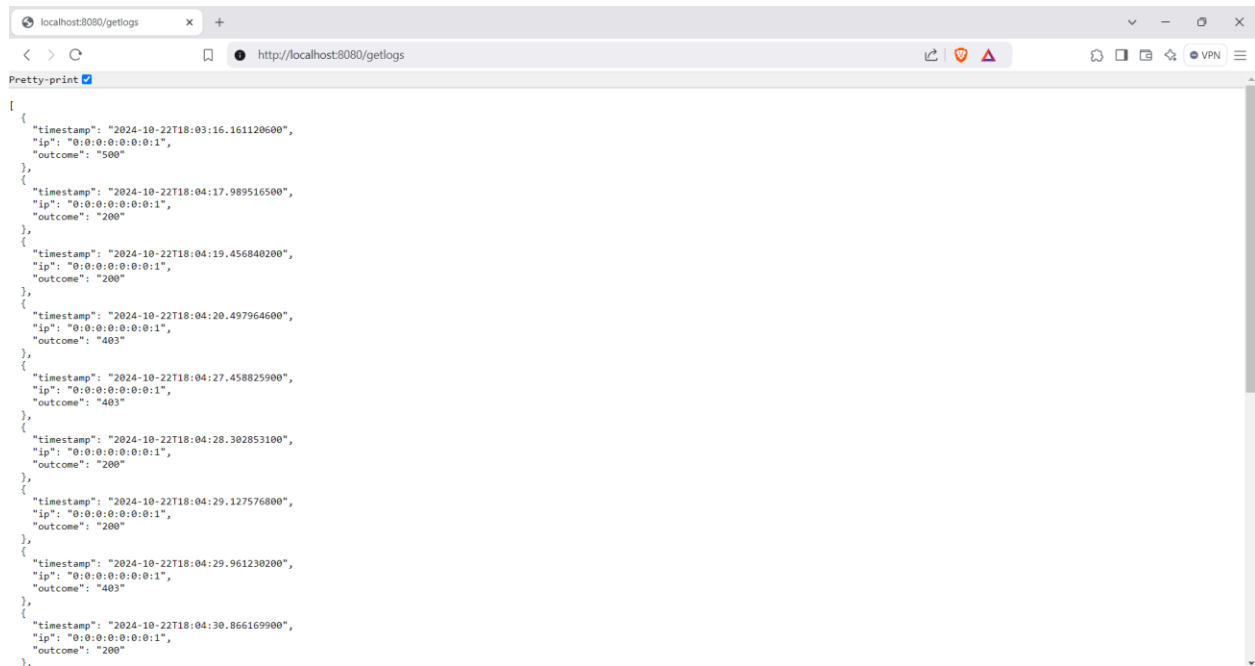


Figure 6

Now, I start my server with a different IP address and port (Figure 7), then again send several requests (Figure 8, Figure 9, Figure 10 – Timeout, and Figure 11) and provide their logs (Figure 12).

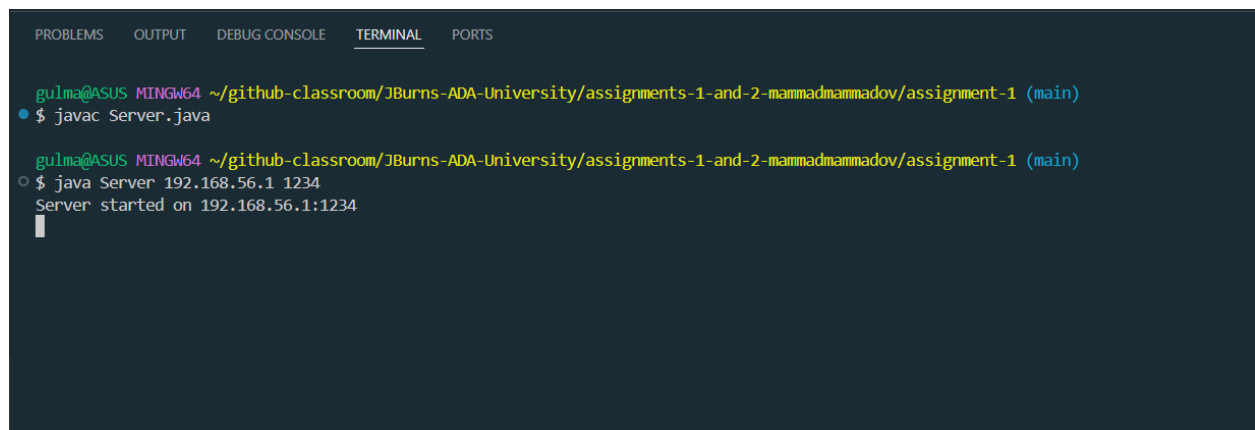


Figure 7



Figure 8

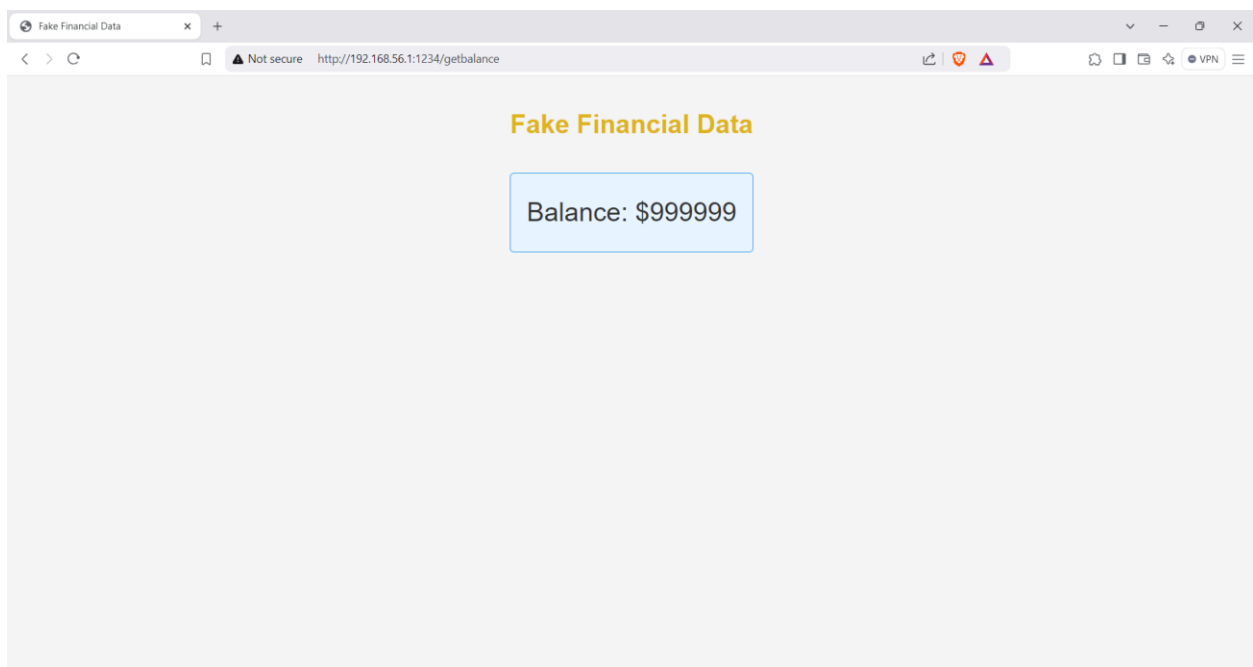


Figure 9

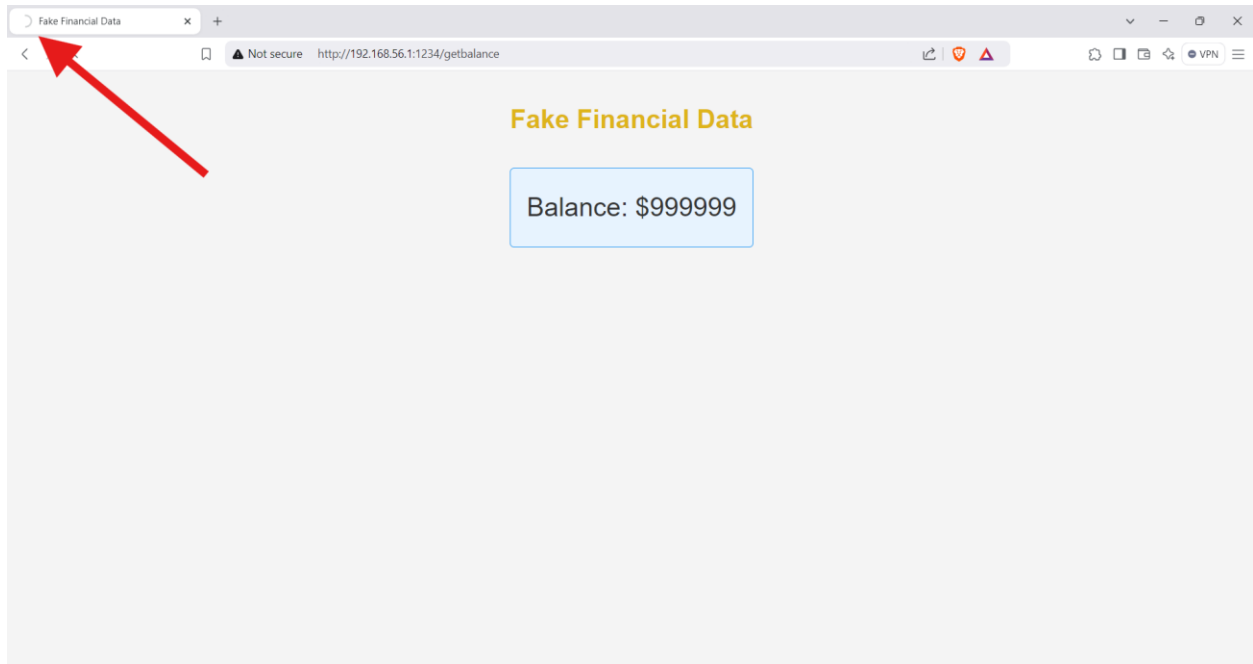


Figure 10

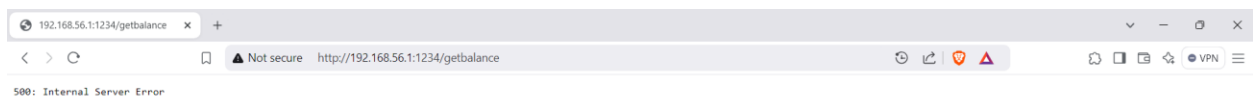
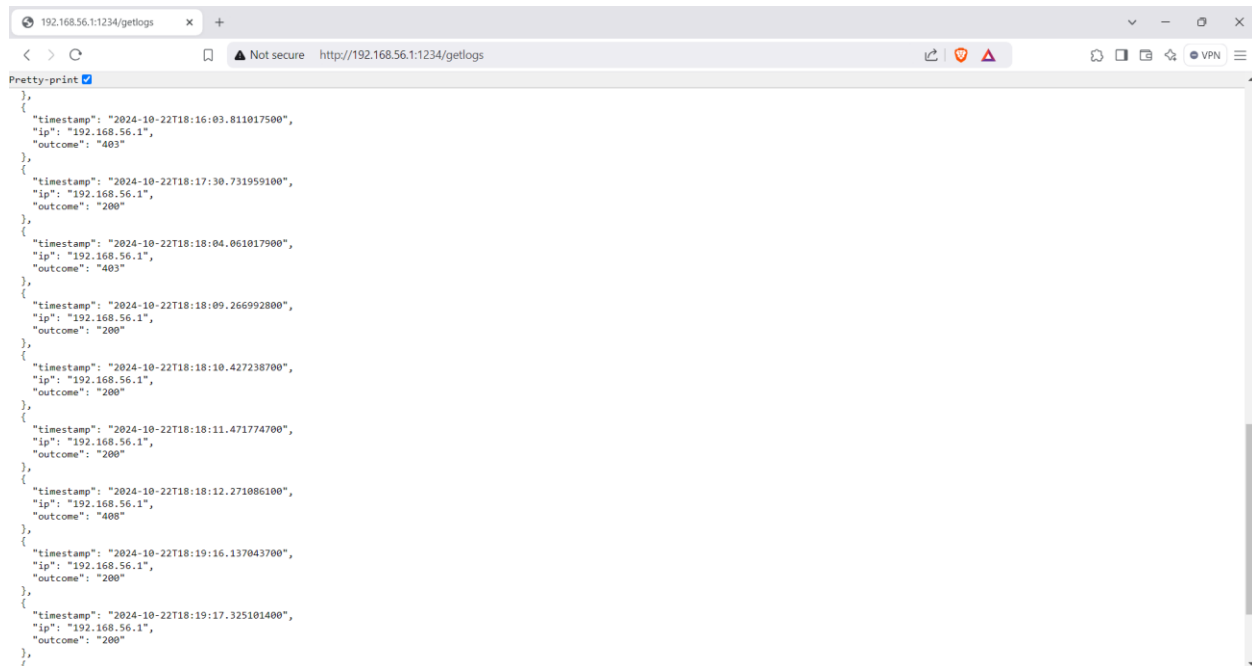


Figure 11

A screenshot of a web browser window. The address bar shows the URL 'http://192.168.56.1:1234/getlogs'. The page content displays a JSON array of log entries, each containing a timestamp, IP address, and outcome. The outcomes are 403, 200, 403, 200, 200, 200, 200, 408, and 200. The browser's developer tools are open, showing the 'Pretty-print' view of the JSON data.

```
},
{
  "timestamp": "2024-10-22T18:16:03.811017500",
  "ip": "192.168.56.1",
  "outcome": "403"
},
{
  "timestamp": "2024-10-22T18:17:30.731959100",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
  "timestamp": "2024-10-22T18:18:04.061017900",
  "ip": "192.168.56.1",
  "outcome": "403"
},
{
  "timestamp": "2024-10-22T18:18:09.266992800",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
  "timestamp": "2024-10-22T18:18:10.427238700",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
  "timestamp": "2024-10-22T18:18:11.471774700",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
  "timestamp": "2024-10-22T18:18:12.271086100",
  "ip": "192.168.56.1",
  "outcome": "408"
},
{
  "timestamp": "2024-10-22T18:19:16.137043700",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
  "timestamp": "2024-10-22T18:19:17.325101400",
  "ip": "192.168.56.1",
  "outcome": "200"
},
{
}
```

Figure 12

## Client Solution

As previously mentioned, we expect the Server to be able to run with different IP addresses and on different ports. To send a request to the server, the client should know the server's IP address and the port it is running on. I ask the user to enter them from the console. If it is not successful, I provide a user-friendly error message, otherwise, send 20 requests to the server at **/getbalance** endpoint. While sending requests, in each iteration, I call *performGetBalance()* and *callGetLogs()* – methods which I am about to explain, but before that, I want to discuss about *checkServerConnection()* method.

*checkServerConnection()* is the method to check whether the client successfully connected to the server for the first time. It does this by sending one request to the **/getbalance** endpoint. In the method body, I first generate a URL based on the server's address, initiate a connection, and configure time settings while and after establishing the connection. After that, I get the response code and check if it is included in a set of numbers – 200, 403, 500, and 408. This is because those are the response codes asked in the assignment instructions. If it does not satisfy this requirement, the connection is considered unsuccessful.

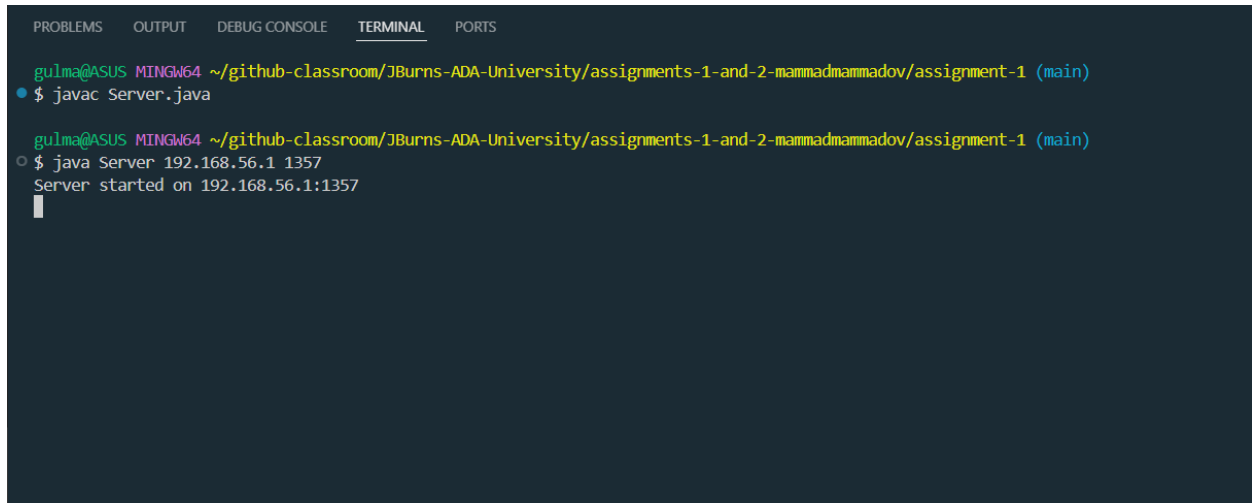
Now coming back to the *performGetBalance()* method, in the method body, I send a request to **/getbalance** endpoint and handle the response code. For each case, I print the response code in the console of the client with a message.

For the *callGetLogs()* method, using the similar flow as above, I generate a URL for the **/getlogs** endpoint and open a connection. I send a request to that endpoint, and if it is successful, I read the response from the server with *BufferedReader*. I do it line by line and append it to my variable



*jsonResponse*, which is a *StringBuilder* object. Since it is mentioned that one line per event should be returned for */getlogs* route, I traverse through all logs, try to disassemble them, and print them in a console, where each of them is presented in one line. To achieve that, I have used regex operations with the built-in *replaceAll()* method.

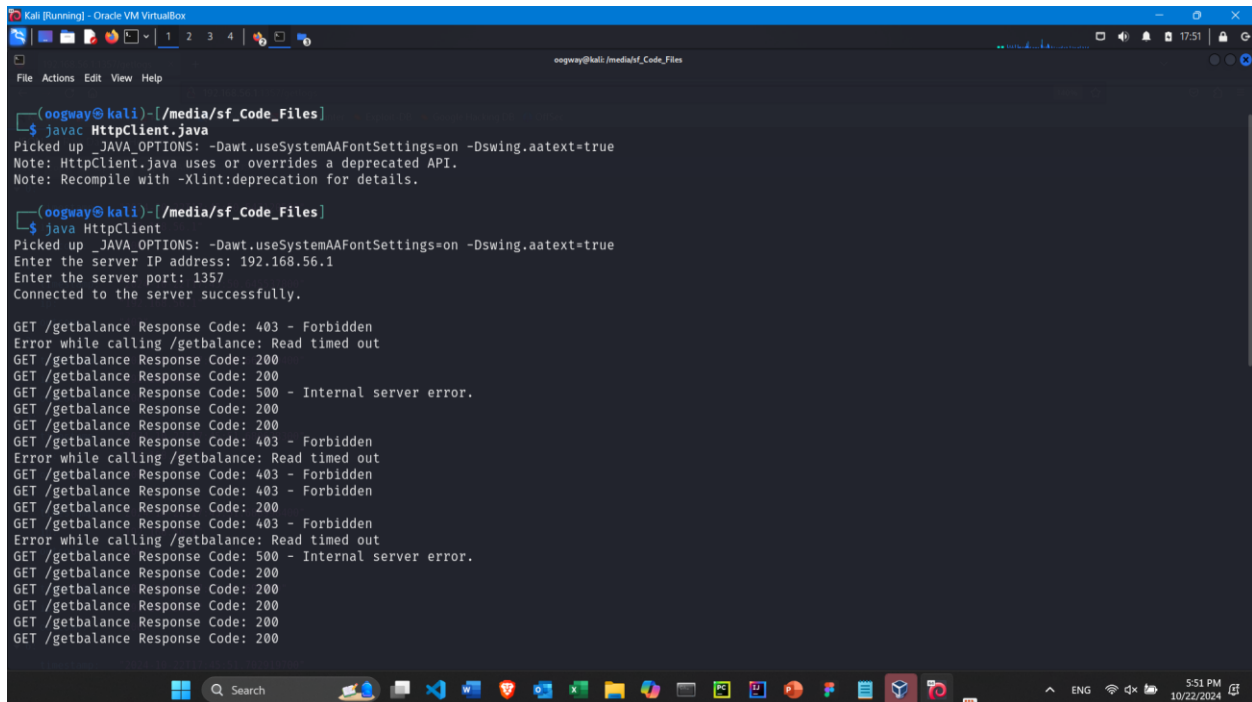
For the evidence that my client works, I will start the server on my machine with port number 1357 (Figure 13), but the client on virtual machine (Figure 14.1, Figure 14.2) that is running on Linux environment – I used Kali distribution for this one. (Note: I deleted some of the previous logs in the Server solution part from my file).



```
gulma@ASUS MINGW64 ~/github-classroom/JBurns-ADA-University/assignments-1-and-2-mammadmammadov/assignment-1 (main)
$ javac Server.java

gulma@ASUS MINGW64 ~/github-classroom/JBurns-ADA-University/assignments-1-and-2-mammadmammadov/assignment-1 (main)
$ java Server 192.168.56.1 1357
Server started on 192.168.56.1:1357
```

Figure 13

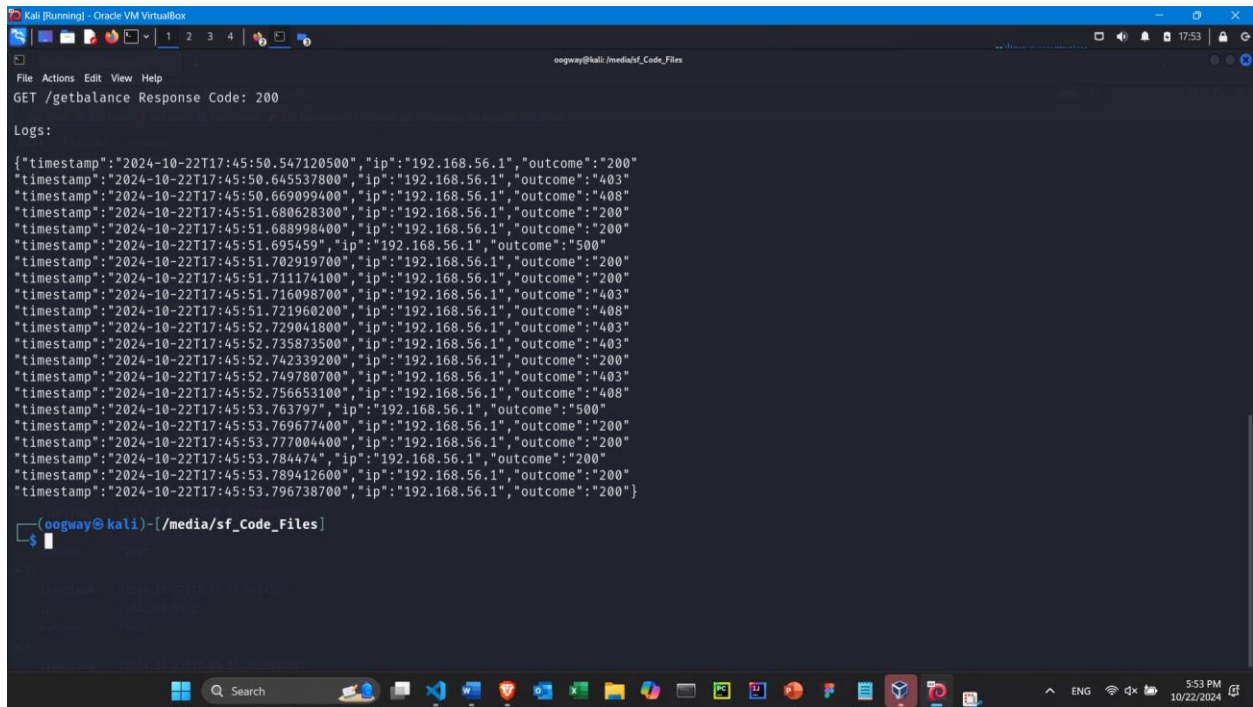


```
(oogway@kali) - [/media/sf_Code_Files]
$ javac HttpClient.java
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Note: HttpClient.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

(oogway@kali) - [/media/sf_Code_Files]
$ java HttpClient
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Enter the server IP address: 192.168.56.1
Enter the server port: 1357
Connected to the server successfully.

GET /getbalance Response Code: 403 - Forbidden
Error while calling /getbalance: Read timed out
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
GET /getbalance Response Code: 500 - Internal server error.
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
GET /getbalance Response Code: 403 - Forbidden
Error while calling /getbalance: Read timed out
GET /getbalance Response Code: 403 - Forbidden
GET /getbalance Response Code: 403 - Forbidden
GET /getbalance Response Code: 200
GET /getbalance Response Code: 403 - Forbidden
Error while calling /getbalance: Read timed out
GET /getbalance Response Code: 500 - Internal server error.
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
GET /getbalance Response Code: 200
```

Figure 14.1



```
Kali [Running] - Oracle VM VirtualBox
oogway@kali: /media/sf_Code_Files
File Actions Edit View Help
GET /getbalance Response Code: 200

Logs:
{"timestamp":"2024-10-22T17:45:50.547120500","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:50.645537800","ip":"192.168.56.1","outcome":"403"}
{"timestamp":"2024-10-22T17:45:50.669099400","ip":"192.168.56.1","outcome":"408"}
{"timestamp":"2024-10-22T17:45:51.680628300","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:51.688998400","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:51.695459","ip":"192.168.56.1","outcome":"500"}
{"timestamp":"2024-10-22T17:45:51.702919700","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:51.711174100","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:51.716098700","ip":"192.168.56.1","outcome":"403"}
{"timestamp":"2024-10-22T17:45:51.721960200","ip":"192.168.56.1","outcome":"408"}
{"timestamp":"2024-10-22T17:45:52.729041800","ip":"192.168.56.1","outcome":"403"}
{"timestamp":"2024-10-22T17:45:52.735873500","ip":"192.168.56.1","outcome":"403"}
{"timestamp":"2024-10-22T17:45:52.742339200","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:52.749780700","ip":"192.168.56.1","outcome":"403"}
{"timestamp":"2024-10-22T17:45:52.756653100","ip":"192.168.56.1","outcome":"408"}
{"timestamp":"2024-10-22T17:45:53.763797","ip":"192.168.56.1","outcome":"500"}
{"timestamp":"2024-10-22T17:45:53.769677400","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:53.777004400","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:53.784474","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:53.789412600","ip":"192.168.56.1","outcome":"200"}
{"timestamp":"2024-10-22T17:45:53.796738700","ip":"192.168.56.1","outcome":"200"}

(oogway@kali)-[/media/sf_Code_Files]
$
```

Figure 14.2

I also checked the endpoints from the browser of my client machine. Figure 15, Figure 16 (timeout), Figure 17, and Figure 18 show the ones for the `/getbalance` endpoint. Figure 19 shows the one for the `/getlogs` endpoint.

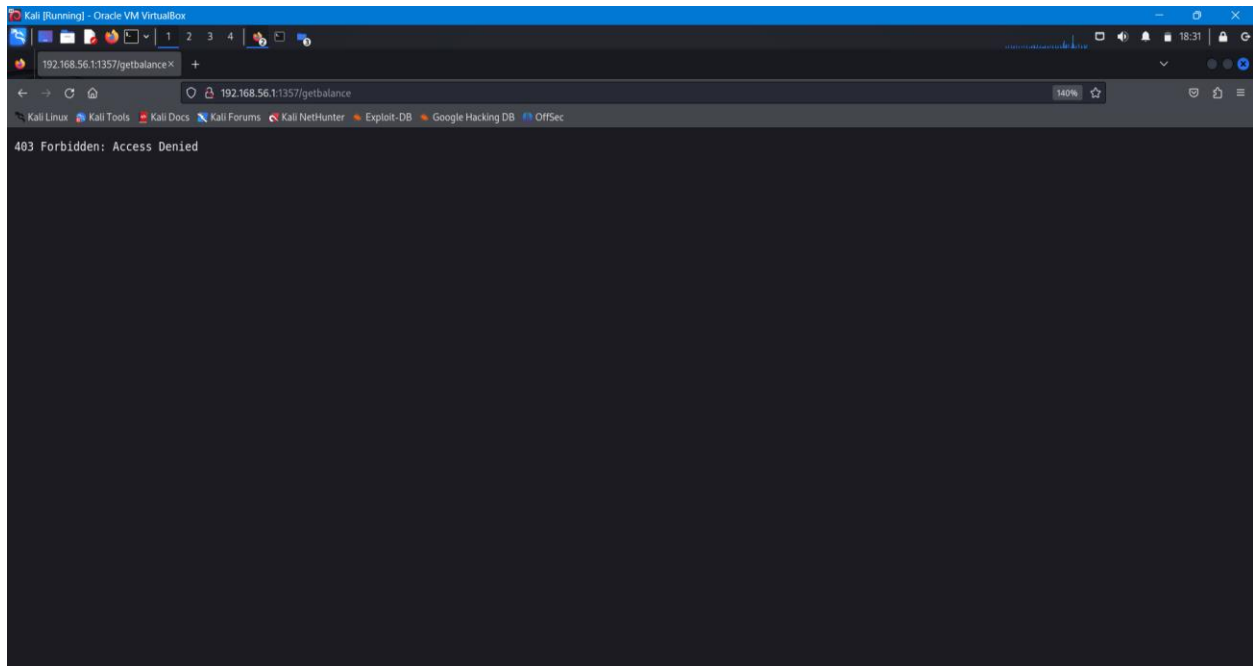


Figure 15

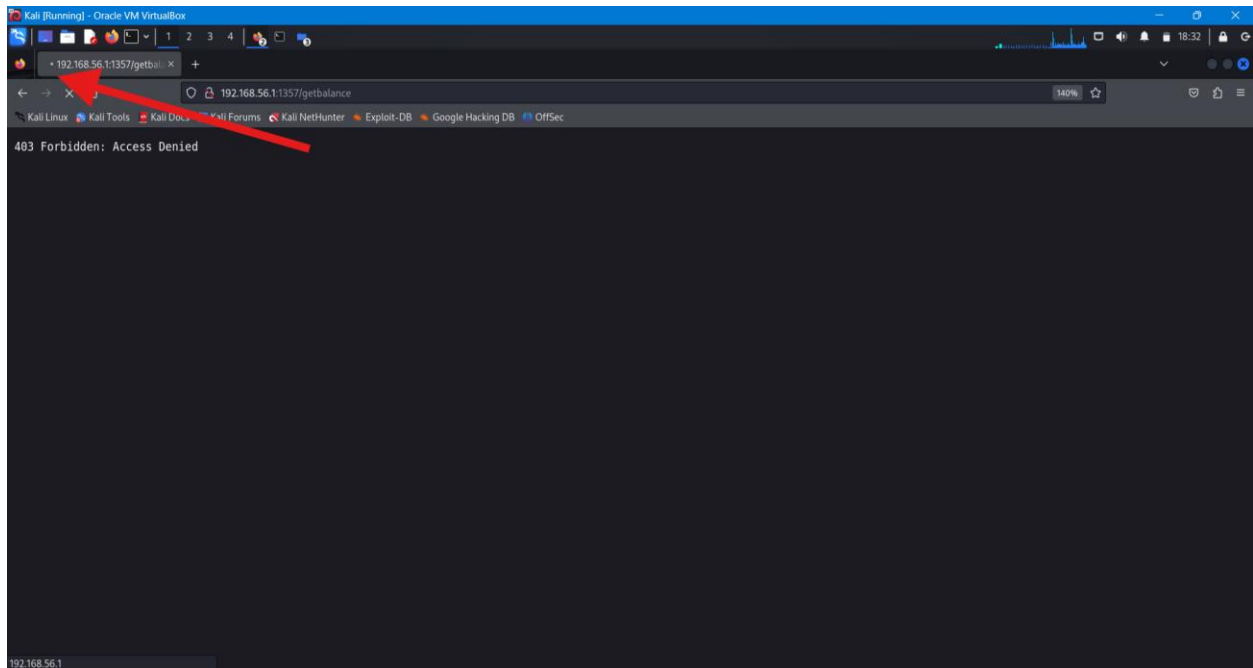


Figure 16

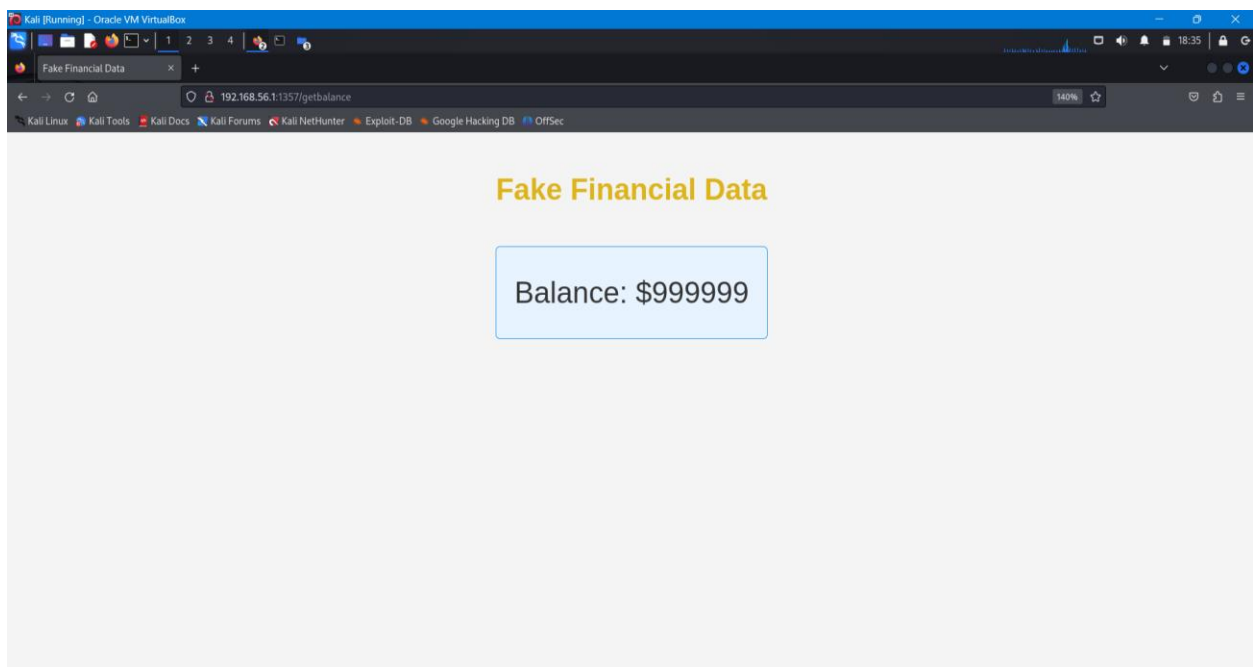


Figure 17

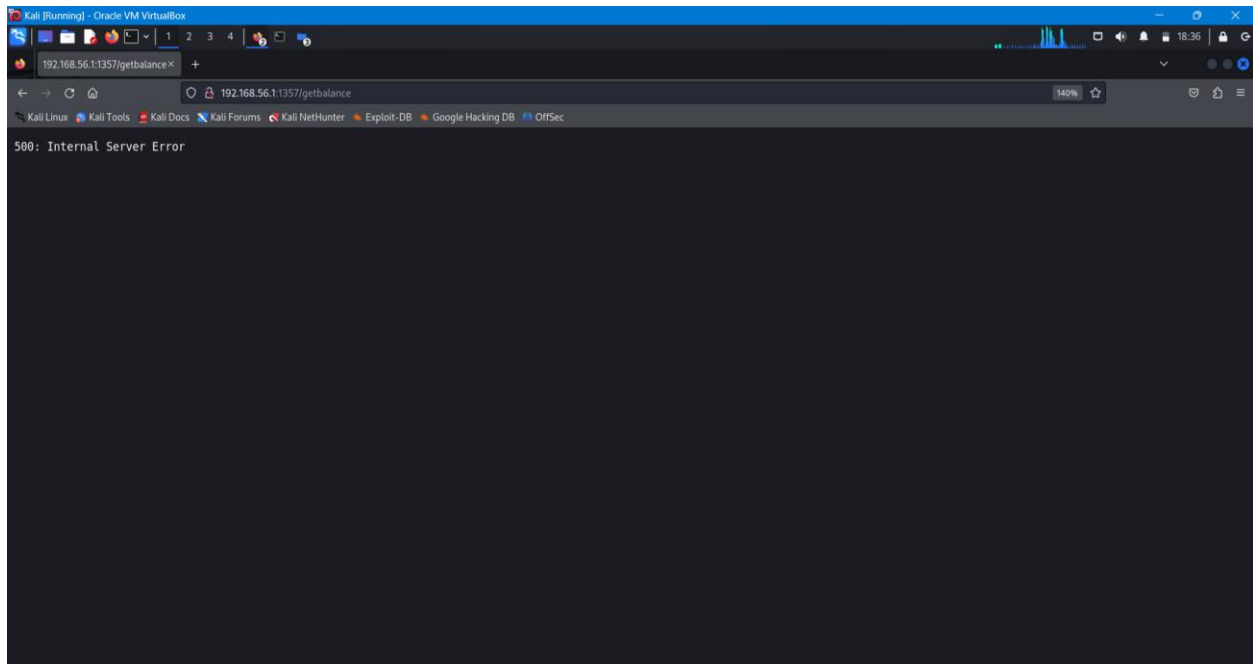


Figure 18

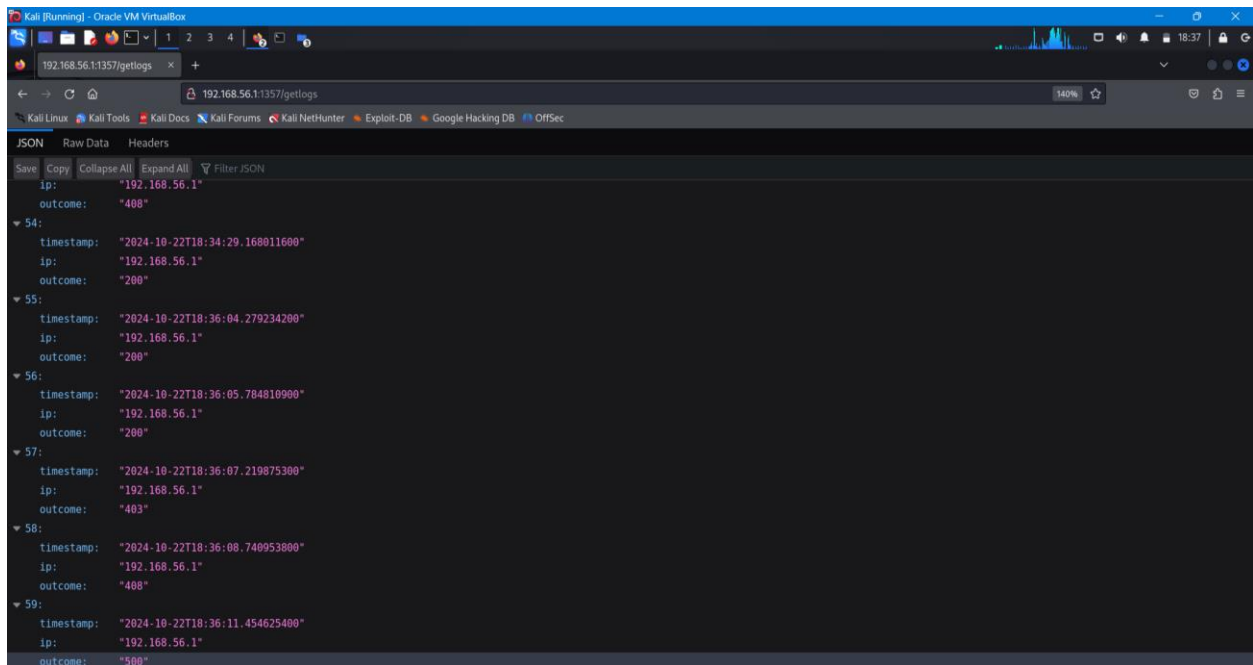


Figure 19