Computer Networks
Fall 2023/24

Lab – Web Server

Lab must be submitted by **14-2-2024** >>> **No late submissions will be accepted.**
Submission is done via Moodle
Submission file name must be lab-StudentName1-StudentName2.zip
For example, lab-AnatTal-YuvalGal.zip (and nothing else)

Building a Multi-threaded Web Server over TCP

Goals:

- Multi-threaded web server over TCP.

Major Milestones (Web Server):

- A config.ini file that contains the following values for the server:
  - port – the port that the server listens to. **We will use port 8080.**
  - root – the root directory of the files. **We will use ~/www/lab/html/**
  - defaultPage – The default page to return the client in case no page is requested in the HTTP request. **We will use index.html**
  - maxThreads – maximum responding threads allowed. **We will use 10.**

- Return the following HTTP Response code:
  - 200 OK
  - 404 Not Found
  - 501 Not Implemented
  - 400 Bad Request
  - 500 Internal Server Error

- Use “content-type: text/html”, “content-type: image”, “content-type: icon” for HTML, image files and icons respectively, in the response. “content-type: application/octet-stream” should be used for anything else.

- Image files supported: .bmp, .gif, .png, .jpg.

- Support the HTTP methods GET and POST, and support getting parameters for both methods (store in a data structure). Also, support HTTP HEAD and TRACE.
  Print to the console window (System.out.println()) the HTTP request header received from the client and HTTP response header that is being sent back by the server to the client.

- Support “transfer-encoding: chunked”. Return as chunked only if the client contains the HTTP header “chunked: yes”, any other header or value means that the response should be with the usual content-length.

- The server's default page should return the HTML page you have written in exercise 1 with a new HTML form that contains a textarea, checkbox and a submit button. Clicking the submit button causes the client to request params_info.html using POST HTTP request. The file should return to the client an HTML page that contains details about the parameters that has been submitted in the form (parameter name and parameter value).

- The server **must not** crash! (Exceptions must be handled and server should be able to recover). For example, if an exception/problem occurs while loading the server (for instance, reading the config.ini), the server should print to the console a user-friendly message and shutdown gracefully.

- Limit the number of threads that can be spawned to maxThreads.

- Client must not be able to surf "outside" the server's root directory

- Add favicon.ico to your website

Goals in Detail:

1. The server is multi-threaded:

In case of a single threaded server, when Alice connects to the server, the server is busy responding to Alice instead of listening to the welcome socket (ServerSocket), therefore if Bob performs a request to the server **while** the server is responding to Alice, the server will not respond to Bob until it finishes handling Alice's request.

In a multi-threaded server, as soon as Alice connects to the server, a new thread is being created to respond a-synchronically, thus allowing the main thread of the server get back and listen on the welcome socket. This way, the server is able to get also Bob's request.

2. A config.ini file that contains the following values for the server:

"port": The port that the server listens to **(we will use port 8080).**
If the port value is "8080", than the welcome socket (ServerSocket) will listen to port 8080.

"root": The root directory of the files **(we will use ~/www/lab/html/)**
Example:
Assume root= ~/www/lab/html/

Assuming the client's browser requests the page: http://localhost/**index.html**,
Then the request can be:
GET /index.html HTTP/1.0[CRLF]
[CRLF]

So the file the server should return is:
[root]\index.html → ~/www/lab/html/index.html

"defaultPage": The default page to surf to, in case no page found in the HTTP request.
(**We will use index.html)**

Example:
Assume defaultPage=index.html and root=c:\wwwroot\

Assuming the client's browser requests the page: http://localhost/,
The request can be:
GET / HTTP/1.0[CRLF]
[CRLF]

So the page the server would look for without defaultPage is:
c:\wwwroot\
But that is a directory – not a file! So, the default page value tells the server which page (=file) to return if the client is not requesting a specific page.

The page the server is looking for is: c:\wwwroot\index.html.

config.ini file format:
port=[port number]
root=[root direrctory]
defaultPage=[default page]

"maxThreads": Maximum responding threads. (**We will use 10)**

Example:
Each connection is being handled by a different thread. That means that if there are "maxThreads" connections, the server has used all its resources to respond to a new connection. The new connection will not be responded until one of the threads finished its job.

3. Return the following HTTP Response code and use "content-type: text/html", "content-type: image" and "content-type: icon" in the response. "content-type: application/octet-stream" should be used for anything else:

- 200 OK – in case everything is okay.
- 404 Not Found – if the file was not found.
- 501 Not Implemented – if the method used is unknown (a Method is like "GET").
- 400 Bad Request – if the request's format is invalid.
- 500 Internal Server Error – some kind of an error.

Reminder: [CR] == "\r" == 0x0d ; [LF] == "\n" == 0x0a

Response example:
HTTP/1.1 200 OK[CRLF]
content-type: text/html[CRLF]
content-length: <page/file size>[CRLF]
[CRLF]
<content of page/file>

4. Support the HTTP methods GET and POST, and getting parameters with both methods. Also, print to the console window (System.out.println()) the HTTP request header from the client and HTTP response header that is sent back to the client:

The server must be able to parse and respond to HTTP requests which use both the GET and the POST methods. Moreover, there must be a (java) method that will be able to parse the parameters from the requests (or even better, an object that will hold the parameters).
In this lab we will do nothing with the parsed parameters, only in Lab 2, so the server must be able to parse the parameters but it will do nothing with them.

Examples:

Requesting "C:\wwwroot\Index.html" is a page.
So the "root" value in the config file is "c:\wwwroot\".

(1) Request:
GET /index.html?x=1&y=2 HTTP/1.0[CRLF]
[CRLF]

Server's actions:

- Print using System.out.println() the request.
- Check which HTTP method is being used → in this case it is "GET".
  If unknown method - return "501 Not Implemented".
- Check if the file c:\wwwroot\index.html exists:
  If not, return error "404 Not Found".
- Read the file's content.
- Create HTTP response header:
  HTTP/1.1 200 OK[CRLF]
  content-type: text/html[CRLF]
  content-length: <the length of index.html>[CRLF]
  [CRLF]
- Print the header.
- Send full response to client (including the page content).

(2) Request:
POST /index.html?x=1&y=2 HTTP/1.0[CRLF]
Content-length: 3
[CRLF]
z=8

Server's actions:

- Print, using System.out.println(), the request header (without z=8).
- Check which HTTP method is being used → in this case it is "POST".
  If unknown method, then return "501 Not Implemented".
- Check if the file c:\wwwroot\index.html exists:
  If not, return error "404 Not Found".
- Read the file's content.
- Create HTTP response header:
  HTTP/1.1 200 OK[CRLF]
  content-type: text/html[CRLF]
  content-length: <the length of index.html>[CRLF]
  [CRLF]
- Print the header.
- Send full response to client (including the page content).

5. Do not allow users to surf "outside" the server's root directory.

Assume the following situation:
Root in config.ini is:
root=c:\websiteroot\

And I have this important file:
C:\passwords\mypasses.txt

So surfing to http://localhost:8080/../passwords/mypasses.txt
Will return the page:
c:\websiteroot\..\passwords\mypasses.txt → c:\passwords\mypasses.txt

In other words, ignore the "/../" part of the request.

6. Add Favicon.ico to your website.

Favicon is the icon of a website, and can be seen by supported browsers (most up-to-date browsers). It is the icon you can see at the address bar in the browser.
When your browser surfs to a specific page in a website it requests the favicon.ico icon, and if there is one, the browser displays it.
By default the browser will look for favicon at the root directory of the website (for example: www.google.com/favicon.ico).

Make your own favicon.ico at: <LINK>

How to start testing:

First, run the server. The port that is mentioned in the config.ini should open, and the server should start listen to it (the port that should be used is 8080).

Now, open your browser and surf to http://localhost:8080/ (or http://127.0.0.1:8080/).
At this point the browser will send HTTP GET request to the server, and the server should parse the request and return the right HTTP response.

Another way to test HTTP POST request is by using HTML Form like the one added to the HTML you wrote in Exercise 1 with method="POST".

```
<form name="input" action="index.html" method="get">
```

You can check the webpage http://www.w3schools.com/html/html_forms.asp, where HTML forms is explained.

Do not forget to test the cases that the requested page does not exist, such as http://localhost:8080/pageDoesntExist.html, and other cases that was not written here specifically (just QA your server as good as you can).

In case you are testing using "localhost" server, using Wireshark will not help.
To get Wireshark to work, you have to run the server on your friend's computer and connect through a real network.
Another way is to use Firefox extension "Live HTTP Headers" that is written in the recitation. It will show only the Headers of the requests and the response, even if working with the localhost.

Tips:

Notice, the following tips are merely recommendations.
You can write the web server however you like, as long as it works correctly!

- Initially Write a basic multi-threaded webserver.
  The main thread is the one listening to the welcome socket (ServerSocket) in the infinite loop.
  Every new connection opens a thread that returns HTTP response that is hard-coded (just like in the code from the recitation).
- Test that server using the browser to see that you're getting the right response.

- Create the config.ini, and read its content. Make sure that your server listens on the port that is written in the file.

- Create a new class called "HTTPRequest".
  The new class represents an HTTP request.
  It receives in its constructor the HTTP request header and parses:
  - Type (GET/POST…)
  - Requested Page (/ or /index.html etc.)
  - Is Image – if the requested page has an extension of an image (jpg, bmp, gif…)
  - Content Length that is written in the request
  - Referer – The referer header
  - User Agent – the user agent header
  - Parameters – the parameters in the request (I used java.util.HashMap<String,String> to hold the parameters).

- Use that class to parse the request.
- After parsing the request – read the requested file (if it exists), and generate the proper response.
- Remember, if the HTML contains embedded objects, the browser will request them right after it receives the HTML.
- Code to read from File to byte[]

```
private byte[] readFile(File file)
{
	try
	{
		FileInputStream fis =  new FileInputStream(file);
		byte[] bFile = new byte[(int)file.length()];

		// read until the end of the stream.
		while(fis.available() != 0)
		{
			fis.read(bFile, 0, bFile.length);
		}

		return bFile;
	}
	catch(FileNotFoundException e)
	{
		// do something
	}
	catch(IOException e)
	{
		// do something
	}
}
```

- Java File class API: http://java.sun.com/j2se/1.3/docs/api/java/io/File.html
- Java HashMap: http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashMap.html

Exception Handling:

The program **must** not crash!

Use exception handling to recover. If you cannot recover, print what happened to the console and close the application gracefully.

Notice that crashing will drop **many** points!

Traces (a.k.a. print to console):

The following traces are **required**:

- Listening port (on startup).
- The HTTP requests arriving to the server.

- The HTTP response header returning to the browser.

You are **encouraged** to use additional traces in your code!
Traces will help you debug your application and perform a better QA.
Make sure your traces actually mean something that will help to understand better what is going on during runtime. **That can and will save you a lot of time!**

Bonus:

Feel free to add more functionality to the application as you see fit!
Good ideas will be **rewarded** with points and world-wide fame (score may be up to a 100)!!!

Please write and explain all the implemented bonuses in a file named bonus.txt.
Notice that bonuses that will not be mentioned in bonus.txt will not be checked, hence they will be ignored!

Important: We don't guarantee in advance that for extra-functionality, extra-points will be rewarded. Also, the idea is to ***add*** a new functionality ***not replace a requested functionality*** – that will lead to dropping of points! In other words, do not ignore things we require, and implement other things instead and call it a bonus! We cannot make it any clearer than that!

Discussion Board:

Please, use the forum of the course to ask questions when something is unclear.

You must make sure that you check the discussion board. If there are unclear matters about the lab that were raised in the discussion board, **our answers are mandatory**, and apply to **everyone**!

You can register to receive e-mail notification from the forum.

A word from your checker:

- **Do not** create your own packages, all your classes **must be in the same package**, and compile while all sources in the same directory
- If you are coding using a mac, make sure you remove all hidden directories generated by your IDE
- You can implement the lab using up to JDK1.7
- Your code will be tested **without an IDE**. Test your code without an IDE and make sure it works!
- Compilation error will drop a huge amount of points

Remember - A happy checker is a merciful checker!

Submission:

- You must submit a bash file called “compile.sh” - The bash file compiles your code successfully on a Linux operating system. If the bash script fails to compile, points will be dropped.
- You must submit a bash file called “run.sh” - The bash file needs to be able to execute your server on a Linux operating system. If the bash scripts fails to launch the server, points will be dropped.

Submit the lab using a zip that contains the following files:

- Sources
- config.ini
- compile.sh
- run.sh
- server root directory
- bonus.txt with the bonuses you implemented – if applicable.
- readme.txt that explains what exactly you have implemented. Explain each class with a few words, and its role in the program. Also write a paragraph on the design you have chosen to implement your server.

Notice: not submitting one of the files needed requested (exclude bonus.txt) will lead to dropping of points!

That’s it ☺ !
Good Luck!