**Score: \_\_\_\_\_**

**Lab10 – A Web Server**

COMP256 – Computing Abstractions

Dickinson College

Spring 2023

Prof. Grant Braught

**Name:**

**Introduction:**

A web server is a piece of back-end software that receives all requests that come to a site. When a request is received, the web server generates a response to the request and returns that response to whomever requested it (typically a web browser). In this lab you will begin with a super simple web server that just returns a status message in response to all requests (not very useful). You will enhance this server so that it is able to serve static HTML web pages. Then you’ll then also be able to additional features that enable it to handle requests for missing pages and pages that contain jpeg images.

**Setup:**

1. This lab will used the Comp256Silli container that you created and used in Lab05. If you still have that container in your Docker Desktop, start it and connect to it using the Tiger VNC viewer.

If you do not have that container, you can create it from a command prompt on your machine (MacOS or WSL in Windows) using the following command:

docker create --name Comp256Silli --publish 5901:5901 --publish 6901:6901 braughtg/comp256-silli:1.0.0

There is no answer required here, but you must be able to start and connect to the Comp256Silli container.

2. Next you will get the starter code for this lab using git. Open a Terminal in the Comp256Silli container and use the commands:

cd /home/student

git clone https://github.com/dickinson-comp256/WebServer.git

There is no answer required here. But the above commands create a WebServer directory in your home directory. You need to ensure that you have the WebServer directory because it contains the starter code for the lab and because you will compress and submit this directory to Moodle at the end of the lab.

3. Run VSCodium and open the WebServer folder.

There is no answer required here, but you should see the file tree for the project in the left pane of VSCodium and be able to open individual files using that file tree.

**Getting Familiar with the Starter Code:**

4. Included in the zip are a few simple HTML files that make up a little web site that will be useful for testing your server. This site contains three pages: index.html, page2.html and sports/soccer.html. Open and read these files as to answer the questions below.

a. Give the anchor (a) tag that is contained in the index.html page.

b. Give the anchor (a) tag that is contained in the page2.html page.

c. Give the image (img) tags that are contained in the soccer.html page.

**Running the Starter Code:**

5. Using a Terminal in the Comp256Silli container, change into the WebServer directory and then compile and run the program in the WebServer.java file.

When you successfully run the server, it will print two startup messages that indicate that it is ready.

Copy and paste the messages that the server printed to the console as your answer for this question. Do not go on until you are able to compile and run the WebServer program.

6. Open the Firefox web browser in the Comp256Silli container and enter the URL:

http://localhost:8085/index.html

a. What is rendered in the web browser? (Note: even though index.html is typically the default home page… you will not see that because you haven’t written that part of the WebServer yet!)

b. When you visited the URL above in your browser the web server should have also generated some additional output to the console about the request that your browser made. Copy and paste that output here.

7. If your web server is running correctly, your answer to #6a should be a line of text indicating that things are working. If you don’t see that line rendered in your browser, something is not working correctly. Revisit questions #1-5 until the issue is corrected.

There is no answer required here, but you will need to be sure your server is running correctly before proceeding with the lab.

**Understanding the Server:**

Now that you have confirmed that the web server is running correctly, let’s dig into its source code to understand how it works. The text below explains some of the key lines in WebServer.java. This code is explained at a high level of abstraction, for context and perspective. You are expected to understand the idea of what is happening and why, but you do not need to understand every detail. Nor would you be expected to produce this code from scratch.

Read the text carefully and follow along by looking at the code. Some of the questions will help to connect the code in the server to what you’ve seen in prior questions. Some later questions will help to focus you in on the parts of the code that will need to be changed to make the web server more functional.

In WebServer.java:

* Line 17 creates a ServerSocket object connected to port 8085. The ServerSocket object in Java and the port number are transport layer features that make it possible for the OS to deliver messages that are addressed to our web server. On any given machine there may be many different servers running (e-mail, web, api, ftp, etc...). Each of those servers is identified with a port number. If the machine were an office building, then the port number would be a particular office in the building. In our case, this means that our web server is listening for requests that arrive on port 8085. The messages that will be delivered to port 8085 will be application layer messages from browsers requesting web pages.

8. In question #6, how did the browser ensure that its request for the page went to port 8085?

* Lines 27-28 call the accept() method on the ServerSocket object.   
    
  That method makes a system call to the OS that causes the OS to move the WebServer process to the Waiting state until a message is received on port 8085. When a message is received on port 8085, the OS will move the WebServer back to the Ready state and the accept() method will run, returning a Socket object.   
    
  That Socket object represents the transport layer network service and provides a two-way communication channel between the browser that sent the request and our server. It will be used to read the Application layer message that was sent from the browser to make the request and to send back the Application layer message that is the server’s response (i.e. a web page).

9. The call to accept() is inside of a while loop. Why do you think that this call is inside of a while loop? What would happen if it was not?

* Lines 33-34 use methods of the Socket object to get its InputStream and OutputStream objects. Your WebServer will use these objects to read the request that was received (input) and return the response (output).
* Lines 38-39 wrap the InputStream in a Scanner object, which just makes it easier to read the request that was sent. It then uses the hasNextLine() method to check that the request that was received actually contains a line to be read
* Lines 41-42 use the Scanner to read the contents of the request into a String referred to by the variable requestText and then print it to the console. You should recognize this as the output from the server that you observed earlier in question #6b.

10. Enter each of the following URLs into your browser to make requests to the server.

a. Record the value that is in the requestText variable for each request. Note: You can infer this by knowing what the code does and looking at the output that is printed.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Request URL** | **Printed requestText** |  |
|  | http://localhost:8085/index.html |  |  |
|  | http://localhost:8085/page2.html |  |  |
|  | http://localhost:8085/sports/soccer.html |  |  |
|  |  |  |  |

b. You’ll notice that no matter what request you make the response that is rendered in the browser is the same. Of course, this isn’t how we want our final web server to behave, but it is what the starter code is supposed to do. What response is rendered in the browser window for every request you made in part a? Why?

11. How is the full text of the request that is received by the server related to the URL that was entered in the browser?

* Lines 45-46 define the response that is returned to the browser. Currently, the response is just a String, but we’ll see shortly that that string can become a complete web page or any other kind of content. The call to responseText.getBytes(“UTF-8”) encodes the String in responseText into the format that it needs to be in to be transferred over the internet to the browser that requested it. We can treat that as an abstraction and not worry about the details.

12. Where have you seen the String contained in the responseText before? What happens if you change the responseText and then make a new request from your browser? **NOTE: To see the effects of any changes you make to the WebServer source code you will need to stop the server (Use CTRL+C in the terminal), recompile it and run it again.**

* Line 51 calls a helper method named sendHTTPResponse that uses the OutputStream to send the response back to the browser. We’ll be using this function as an abstraction that just lets us send the response. But there are a few little things you should know.   
    
  The “200 OK” is an *HTTP status code* that tells the browser that the request was successful. There are different status codes for other events (e.g. 404 for page not found). The “text/html” is called a *MIME Typ*e and it tells the browser what type of content is in the response. There are also other MIME Types for other types of content (e.g. “image/jpeg”) for Jpeg images.
* Lines 55-56 close the Socket that the server was using to communicate with the browser. The request has been received and the response has been sent, so the Socket is no longer needed because the while loop we saw earlier ensures that new one is created for every request.

**Handling Requests for HTML Pages:**

The answers to questions #10, #11 and #12 point to the way that the web server needs to be modified so that the response returned contains the requested page. Specifically, you will need process the request to get the path to file containing the page that is being requested (e.g. index.html, or sports/soccer.html). You will then need to read the contents of that file (i.e. the HTML source code), convert it to an array of bytes and return it as the response.

🔑 13. To get started, let’s identify the key parts of the server that are involved. Fill in the “Variable Name” column below with the name of the Java variable in WebServer.java that contains the indicated content:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | **Variable Name** | **Contents** |  |
|  |  | The string that containing the request that was received and thus also the path to the page that has been requested. |  |
|  |  | The string that contains the response that is converted to an array of bytes and sent back to the browser. |  |
|  |  |  |  |

14. Next, let’s extract the path to the file containing the HTML source code for the requested page. Add some code to the main method so that in addition to printing the complete request, it also prints just the path to the file containing the page being requested. For example, if the request is “GET sports/soccer.html HTTP/1.1” then the server should output:

Request: GET /sports/soccer.html HTTP/1.1

Filename: sports/soccer.html

**It is important that the leading / is omitted from the path.**

Hint: Use some methods in the String class to extract just the part that we want.

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

15. Now that you have the path to the file that contains the HTML source code for the page that is being requested, you need to open that file, read its contents into a String, have that String converted to an array of bytes and send it back to the browser as the response.

Though you may not recall it from memory, you learned **how to read a text file in Java** in COMP 132. You can refer back to those assignments. Or alternatively you could search for something like “Java read text file into string” and try to adapt what you find to this application.

You’ll know you have this working when you can make a request for index.html or page2.html and see the page that you requested being rendered in your browser.

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

**The 404!**

You have likely seen a 404 error before. This happens when the web server is unable to find the page that you have requested.

16. Try to request the page http://localhost:8085/noSuchPage.html from your server. Describe what happens and explain some detail why it happens.

17. When a web server does not find the page that is requested, it should return a response with the HTML status code “404 Not Found” (instead of “200 OK”). This is where the 404 comes from! Modify your web server so that if it cannot find the requested page, instead of crashing, it returns a response with the “404 Not Found” status.

You’ll know this is working when your browser tells you that there was a 404 error when you request a non-existent page.

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

18. **Optional**: Sometimes a 404 response just the browser telling you the page cannot be found, as you just saw. Other times a site will provide a custom 404 page. For example:

* <https://www.dickinson.edu/nosuchpage.html>
* <http://gogole.com/nosuchpage.html> (not very interesting)

Some sites have fun with their 404 pages, providing a clever, witty or snarky page telling you that you’ve requested a page that doesn’t exit. For example, check out the following examples:

* <https://www.marvel.com/nosuchpage.html> (different each time)
* <https://slack.com/nosuchpage.html>
* [And lots more...](https://www.canva.com/learn/404-page-design/)

Create a special 404 page for the site hosted by your web server and modify your WebServer.java so that instead of just returning an empty 404 page it returns your special 404 page as the response when it sends back the “404 Not Found” status.

No answer is required here. The code for this question will be contained in your WebServer.java file and in the .html page that it serves for 404 errors.

**Handling Requests for Image Files:**

19. The page sports/soccer.html is supposed to display a soccer ball and a soccer goal.

a. Give value html elements from the page that display the those images.

b. Now request the soccer.html page using the URL:

http://localhost:8085/sports/soccer.html

Examine the output generated by the server in the console when you make the above request. What do you see that is related to the images?

You should have seen three requests. The first one is for the page itself. Then as your browser attempts to render the page it encounters the img elements that you identified in part a. Each of those elements has a src attribute that gives the name of the file containing the image. Thus, to actually render the images, the browser must request the files containing those images. The two additional requests that you saw were the browser making the requests for the image files to be rendered in the page.

c. What does your web server do when it receives the request for these images? Are the images requested images rendered in your page?

20. **Optional**: Despite the browser requesting the images that are in the page, you will notice that the browser does not actually render them. This is because images are **binary data** and the way we have written the server thus far it tries to read them as text data - because that’s how we wrote the code in question #15.

To fix this, modify your web browser so that it reads any file with an extension of .jpg as a binary file. Be sure that when you return the response containing the bytes in that file that you set the MIME type to be "image/jpeg" so that the browser knows what type of image it is. Hint: Check out the Files.*readAllBytes* method in Java for an easy way to do this.

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

**The Default Page:**

The default home page for many web sites is saved in the file index.html. When this is the case, and someone makes a request with just the name of the server, the contents of the index.html file are returned.

21. Make the request http://localhost:8085 to your web server. What happens? Why?

22. Modify your WebServer.java file so that if a request like the one in #19 is made the response will contain the default home page (index.html).

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

**Optional Challenge: Multithreading the Server**

As you have seen, when a page contains images, each image generates a new request to the server. Because our server processes requests one by one, if you were to request a page with lots of images it could take it could take a long time to load all of them. Further, if one of the images happens to be very large then all of the others will be forced to wait. This gets even worse if many people are making requests of the server simultaneously. To fix this web servers use multiple threads. A new thread is created for each request and runs independently. That provides two advantages. First the server can accept more requests while previous ones are being processed (i.e. main is its own thread). Second, if one thread is forced to wait or takes a long time, others will still be able to make progress.

23. Modify your WebServer.java so that each request is handled in its own thread. Hint: When a request is received, create a new thread and run it. The code in that thread should generate and send the response.

No answer is required here. The code for this question will be contained in your WebServer.java file when you submit the lab.

**Submitting:**

24. Submit this lab as follows:

a. Use the following commands in a terminal window in the container:

cd /home/student

tar -zcvf Lab10.tar.gz WebServer

b. Use the FireFox browser in the container to go to the course Moodle site and submit the Lab10.tar.gz file to the Lab10 Code assignment.

c. Convert this activity sheet to a PDF and submit it to the Lab10 assignment on Moodle.

Optional: To help me improve and scope these activities for future semesters please consider providing the following feedback.

a. Approximately how much time did you spend on this activity outside of class time?

b. Please comment on any particular challenges you faced in completing this activity.