Web Server

Computer Network – Project #1

2018008904 이성진

2019-10-16

Web Server: Table of Contents

1. Overview

Project #1 is to develop a Web server that can receive HTTP messages from clients and handle responses that match messages based on what you learned in class. A Web server can receive service requests from multiple clients at the same time, and it will have to implement the process of processing requests using multi-threads.

The server code can be described in following two main parts:

1. After creating a socket based on a pre-specified port number, implement **WebServer.java**, which works to detect the client's request coming through the socket. If an HTTP message is received, the file calls a thread that handles the request for that message to be executed, which will allow multiple requests from multiple clients to be processed simultaneously.

2. Implement **HttpRequest.java**, which responds appropriately to each message, invokes the request when the web server receives the request. The code is called as a thread on the web server, and each thread analyzes the HTTP messages it receives using StringTokenizer and others and sends the pre-determined messages and files based on the analysis results.

2. Source Review

2.1. WebServer.java

```java
import java.net.*;

public class WebServer {
    public static final int PORT = 2222;

    public static void main(String[] args){

        try{
            ServerSocket serverSock = new ServerSocket(PORT);
            System.out.println("Server is running...\n");

            while (true){
                Socket connectionSock = serverSock.accept();
                HttpRequest request = new HttpRequest(connectionSock);
                Thread thread = new Thread(request);
                thread.start();
            }
        }
        catch (Exception e){
            System.err.println(e.getMessage());
        }
    }
}
```

This is the source code of the main class **WebServer.java**. First, we import the java.net.* package to use the **Socket** class included in the java.net package. The port number was declared inside the class in the form ***public static final int*** for later modification. Inside the main method, run the try catch syntax to handle the exception.

In the try syntax, create a socket based on a preset port, print out the message that the server is running, and check if a connection request is made to the currently open socket. If a request is received, create a **HttpRequest** class, create a single thread that calls the

processRequest method, and then run the repeat indefinitely. This allows **WebServer.java** to process requests in multithread indefinitely as long as the server is running.

2.2. HttpRequest.java

```
import java.net.Socket;
import java.util.*;
import java.io.*;

enum StatusCode { OK, BAD_REQUEST, FORBIDDEN, NOT_FOUND, HTTP_VERSION_NOT_SUPPORTED, }

final class HttpRequest implements Runnable
{
    final static String CRLF = "\r\n";
    final static String HTTP_VERSION = "1.1";
    final static String DEFAULT_CONTENT_TYPE = "application/octet-stream";
    final static int BUFFER_IN_SIZE = 1024;
    final static int BUFFER_OUT_SIZE = 1024;
    final static Properties CONTENT_TYPES = new Properties();
    final static EnumMap<StatusCode, String> SCODES = new EnumMap<~>(StatusCode.class);
    static
    {
        CONTENT_TYPES.setProperty("html", "text/html");
        CONTENT_TYPES.setProperty("htm", "text/html");
        CONTENT_TYPES.setProperty("txt", "text/plain");
        CONTENT_TYPES.setProperty("jpeg", "image/jpeg");
        CONTENT_TYPES.setProperty("jpg", "image/jpeg");

        SCODES.put(StatusCode.OK, "200");
        SCODES.put(StatusCode.BAD_REQUEST, "400");
        SCODES.put(StatusCode.FORBIDDEN, "403");
        SCODES.put(StatusCode.NOT_FOUND, "404");
        SCODES.put(StatusCode.HTTP_VERSION_NOT_SUPPORTED, "505");
    }
    StatusCode code;
    Socket socket;
    File requestedFile;
```

This is HttpRequest.java, which implements the function of processing an HTTP request when it is sent. Since the HttpRequest class is used as

a factor in the thread's construction, it has to be driven by multithreading, it has to be implemented and used as a Runnable Interface.

And this class definition defined variables such as CRLF, which means the end of this line, the default HTTP version, the Default content type, and CONTENT_TYPES and SCODES that pre-specified format and response code of the requested file. They also specified member variables such as socket for connection, status code to send, and file to save the requested file.

```
public HttpRequest(Socket socket) throws Exception
{
    this.socket = socket;
    this.code = null;
    this.requestedFile = null;
}

public void run()
{
    try
    {
        processRequest();
    } catch (Exception e)
    {
        System.err.println("Exception occurred while processing request: ");
        e.printStackTrace();
    }
}
```

The constructor of this class is a structure that receives Socket class object as parameter and assigns it to local variable socket as in WebServer.java. Because all classes that implemented Runnable had to define the run function called when the thread was executed, but since the run method cannot push the Exception through, the method called processRequest was defined separately to handle other logics.

```
private void processRequest() throws Exception {
    InputStream is = null;
    DataOutputStream os = null;
    FileInputStream fis = null;
    BufferedReader br = null;
    try {
        is = socket.getInputStream();
        os = new DataOutputStream(socket.getOutputStream());
        br = new BufferedReader(new InputStreamReader(is), BUFFER_IN_SIZE);
        String requestLine = br.readLine();
        String errorMsg = parseRequestLine(requestLine);
        String headerLine = null;
        while ((headerLine = br.readLine()).length() != 0) {
            System.out.println(headerLine);
        }
        if (errorMsg == null) {
            try {
                fis = new FileInputStream(requestedFile);
            } catch (FileNotFoundException e) {
                System.out.println("FileNotFoundException while opening file input stream.");
                e.printStackTrace();
                code = StatusCode.NOT_FOUND;
            }
        }
        else {
            System.out.println();
            System.out.println(errorMsg);
        }
        sendResponseMessage(fis, os);
    }

    finally {
        if (os != null) os.close();
        if (br != null) br.close();
        if (fis != null) fis.close();
        socket.close();
    }
}
```

This is processRequest method that handles actual HTTP requests. First, you get one BufferedReader object from the inputstream of the socket to process the messages you receive. The first line of the message, the request line portion, is obtained using the readLine method of the BufferedReader class. Afterwards, the HTTP request sent to the server is analyzed and printed to the terminal by calling the parseRequestLine method. If the request line cannot be processed

normally, the return value of the parseRequestLine method is not null but an error message and is stored in the errorMsg variable. In addition, the header line portion is output to the system and not reflected separately.

If errorMsg is null, it means the Status coder is 200 OK and the file can be transferred, so create a FileInputStream guest using the requested file requestFile. If the file is not verified, it will transmit the 404 Not Found code. If there is an error message, when the output is finished, the sendResponseMessage method is called to send the message through os, which is the output of the file and socket using the requested file.

```
private String parseRequestLine(String requestLine)
{
    System.out.println();
    System.out.println("Received HTTP request:");
    System.out.println(requestLine);
    StringTokenizer tokens = new StringTokenizer(requestLine);
    if (tokens.countTokens() != 3) {
        code = StatusCode.BAD_REQUEST;
        return "Request line is malformed. Returning BAD REQUEST.";
    }
    String method = tokens.nextToken().toUpperCase();
    String fileName = tokens.nextToken();
    fileName = "." + fileName;
    File file = new File(fileName);
    if (!file.exists()) {
        code = StatusCode.NOT_FOUND;
        return "Requested file " + fileName + " does not exist. " +
                "Returning NOT FOUND.";
    }
    if (!file.canRead()) {
        code = StatusCode.FORBIDDEN;
        return "Requested file " + fileName + " is not readable. " +
                "Returning with FORBIDDEN.";
    }
```

```
    if (file.isDirectory()) {
        File[] list = file.listFiles(new FilenameFilter() {
            public boolean accept(File dir, String f) {
                if (f.equalsIgnoreCase( anotherString: "index.htm") || f.equalsIgnoreCase
                    return true;
                return false;
            }
        });
        if (list == null || list.length == 0) {
            code = StatusCode.NOT_FOUND;
            return "No index file found at requested location " + fileName +
                    " Returning NOT FOUND";
        }
        file = list[0];
    }
    requestedFile = file;
    String version = tokens.nextToken().toUpperCase();
    if (!version.matches( regex: "HTTP/([1-9][0-9.]*)")) {
        code = StatusCode.BAD_REQUEST;
        return "HTTP version string is malformed. Returning BAD REQUEST.";
    }
    if (version.equals("HTTP/1.0")) {
        code = StatusCode.BAD_REQUEST;
        return "HTTP version 1.0 may cause problems. Returning BAD REQUEST.";
    }
    if (!version.equals("HTTP/1.0") && !version.equals("HTTP/1.1")) {
        code = StatusCode.HTTP_VERSION_NOT_SUPPORTED;
        return version + " not supported. Returning HTTP VERSION NOT SUPPORTED.";
    }
    code = StatusCode.OK;
    return null;
}
```

This part is the parseRequestLine method that analyzes the request line received from the client. Using StringTokenizer, the request line is split by a toolken to determine if the appropriate number of token contains the appropriate content, if a file is required, what format or exist and accessible, and then the error message is returned when an error occurs. It also assigns the appropriate Status code to the code variable to enable the server to send the appropriate response.

```
private void sendResponseMessage(FileInputStream fis, DataOutputStream os) throws Exception
{
    String statusLine = "HTTP/" + HTTP_VERSION + " " + SCODES.get(code) + " ";
    String entityBody = "<HTML>" + CRLF +
            " <HEAD><TITLE>?</TITLE></HEAD>" + CRLF +
            " <BODY>?</BODY>" + CRLF +
            "</HTML>";
    String message;
    switch (code) {...} // Set appropriate status message string to message.
    statusLine += message;
    if (code != StatusCode.OK)
        entityBody = entityBody.replaceAll( regex: "\\?",  replacement: message
                + " - sent by Girinman's WebServer");

    System.out.println("message: " + message);
    System.out.println("statusLine: "+ statusLine);
    System.out.println("entityBody:" + CRLF + entityBody);
    os.writeBytes( s: statusLine + CRLF);
    sendHeaderLines(os);
    os.writeBytes(CRLF);
    if (code == StatusCode.OK) {
        System.out.println("Sending requested file to client...");
        sendBytes(fis, os);
    }
    else {
        System.out.println("Sending error message to client...");
        os.writeBytes(entityBody);
    }
}
```

Request line for analysis after the response message sendresponsemessage that act as methods to send. A response to http request line make appropriate response based on the content of messages. If status is not 200 OK, response message must conatain text values that shows the status code.

Also a challenge to meet the specifications, sendheaderlines method was called - content type, content length like headers are also configured. After sending status line and header line header, there will be one more CRLF that indicates header line was finished. After that, the requested file data will be sent, if it exists.

```
private void sendHeaderLines(DataOutputStream os) throws Exception {
    StringBuffer headerLines = new StringBuffer();
    String contentTypeLine = "Content-type: ";
    switch (code) {
        case OK:
            String contentLength = "1024";
            contentTypeLine += contentType(requestedFile.getName()) + CRLF;
            contentTypeLine += "Content-length: " + contentLength;
            break;
        default:
            contentTypeLine += "text/html";
    }
    headerLines.append(contentTypeLine + CRLF);
    os.writeBytes(headerLines.toString());
}

private void sendBytes(FileInputStream fis, OutputStream os) throws Exception {
    byte[] buffer = new byte[BUFFER_OUT_SIZE];
    int bytes = 0;
    while((bytes = fis.read(buffer)) != -1) {
        os.write(buffer,  off: 0, bytes);
    }
}

private String contentType(String fileName) {
    String lowFileName = fileName.toLowerCase();
    int lastdot = lowFileName.lastIndexOf( str: ".");
    if ((lastdot != -1) && (lastdot != lowFileName.length() - 1))
        return CONTENT_TYPES.getProperty(lowFileName.substring(lastdot+1), DEFAULT_CONTENT_TYPE);
    else return DEFAULT_CONTENT_TYPE;
}
```

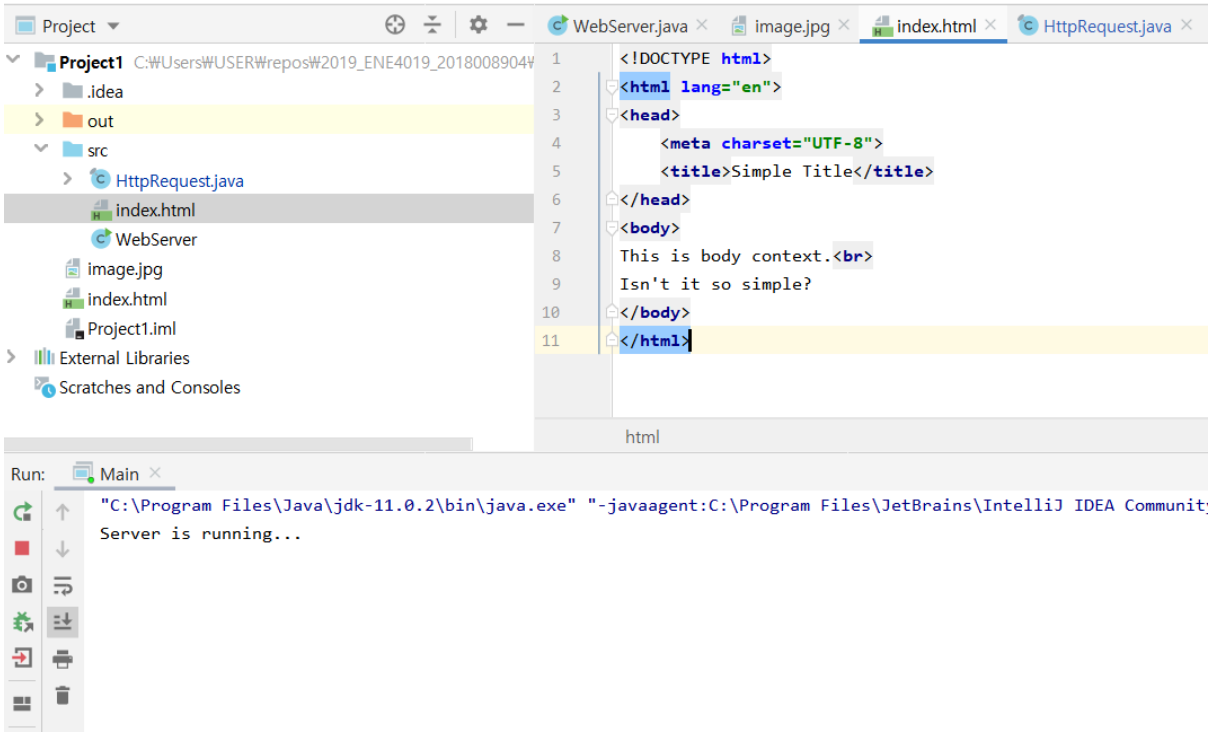
Finally, the functions that are executed in the middle when 함수 is called are sendHeaderLines, sendBytes, and contentType. Based on the request message analyzed, sendHeaderLines is responsible for writing the appropriate header line and outputting it to the socket.

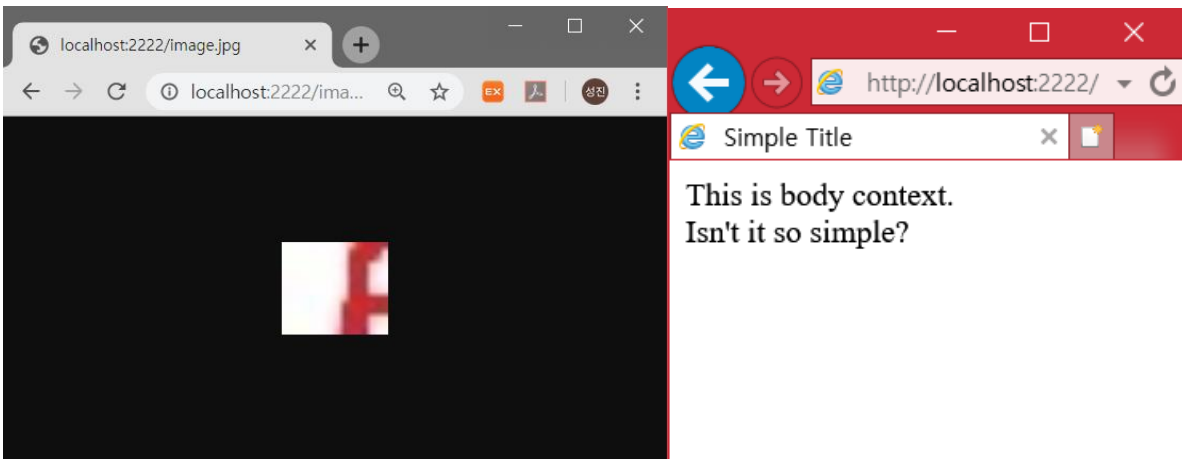
sendBytes reads the data from the requested file to the buffer via FileInputStream and outputs it to the socket.

contentType is a role that tells you what extension the file ends with and what data format it is based on a predefined enum based on the file's name value.

3. Results

To verify that the index.html and image.jpg files are normally sent to the client, place them in the server's directory and run the server. Afterwards, the HTTP request was sent to the web browser and the server responded to the request.





```
Received HTTP request:
GET / HTTP/1.1
Accept: text/html, application/xhtml+xml, image/jxr, */*
Accept-Language: ko,en-US;q=0.7,en;q=0.3
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
Accept-Encoding: gzip, deflate
Host: localhost:2222
Connection: Keep-Alive
message: OK
statusLine: HTTP/1.1 200 OK
entityBody:
<HTML>
 <HEAD><TITLE>?</TITLE></HEAD>
 <BODY>?</BODY>
</HTML>
Sending requested file to client...
```

In addition, checking the web server through an automatic scoring server showed that the server was implemented that met all of the following goals:

Your Test Result

Student INFO

Student Number	Student Name	WebServer IP	WebServer PORT	ACCESS TIME	SCORE
2018008904	Lee Seong Jin	121.190.137.208	2222	2019-45-17 02:45:55	100

List of Test Items

WEB SERVER SOCKET :	TRUE
Multi Thread :	TRUE
STATUS CODE : 200 OK	TRUE
STATUS CODE : 404 NOT FOUND(EXCEPTION HANDLING)	TRUE
STATUS CODE : 400 BAD REQUEST(HTTP PROTOCOL VERSION)	TRUE
CONTENT LENGTH	TRUE
CONTENT TYPE TEXT/HTML	TRUE
CONTENT TYPE IMAGE/JPEG	TRUE