**Project Example #1: Building a multi-threaded web server**

**Name**: Kwon DoHyun

**Student ID**: 2023065350

**Organization**: Department of computer science and engineering, Hanyang Univ. (Seoul, Republic of Korea)

**Introduction**

This assignment aims to teach you how to handle HTTP requests and responses. The assignment builds a server step by step. Connect to a specific port through socket programming and process HTTP requests and responses sent by clients through it. It allows multiple requests to be processed quickly by utilizing threads.

**Contents of the attachment**

**WebServer.java**

**/\* Extension 1: \*/**

|  |
| --- |
| import java.io.\* ;  import java.net.\* ;  import java.util.\* ;  /\*\*  \* WebServer class implements a simple multi-threaded web server.  \*  \* This class serves as the main entry point for the web server application.  \* Its primary responsibilities include:  \* 1. Initializing the server on a specified port  \* 2. Listening for incoming client connections  \* 3. Creating new threads to handle each client request  \*  \* The server runs indefinitely, continuously accepting new connections  \* and spawning threads to process HTTP requests.  \*/  public final class WebServer {  public static void main(String argv[]) throws Exception {  // Check if command line argument for port number is provided  if(argv.length < 1){  System.out.println(“Usage: java WebServer <port number : >”);  return ;  }  // Get the port number from the command line.  int port = (new Integer(argv[0])).intValue();    // Mission 1(Handle Connection): create and bind a socket (Fill #1 ~ #2)  // Fill #1 Create the Serversocket and wait for the TCP Connection  // Establish the Serversocket wait for the TCP Connection  ServerSocket socket = new ServerSocket(port);    // Process HTTP service requests in an infinite loop.  while (true) {  // Fill #2 Listen for a TCP connection request.  Socket ClientSocket = socket.accept();  //Mission 1-B Fill #3 Construct an object to process the HTTP request message.  HttpRequest request = new HttpRequest(ClientSocket);  // Mission 1-BFill #4, Create a new thread to process the request and start the thread  Thread thread = new Thread(request);  thread.start();  }  }  } |
|  |

**HttpRequest.java**

|  |
| --- |
| import java.io.\* ;  import java.net.\* ;  import java.util.\* ;  final class HttpRequest implements Runnable {  final static String CRLF = "\r\n";  Socket socket;    // Constructor  public HttpRequest(Socket socket) throws Exception {  this.socket = socket;  }    // Implement the run() method of the Runnable interface.  public void run() {  try {  processRequest();  } catch (Exception e) {  System.out.println(e);  }  }  /\*\*  \* ProcessRequest Method class handles HTTP Request Messages  \* 1. Receive and send HTTP Request and HTTP Response  \* 2. Parse HTTP request line and save  \* 3. Parse HTTP Header line and save  \*  \* The server runs indefinitely, continuously accepting new connections  \* and spawning threads to process HTTP requests.  \*/  private void processRequest() throws Exception {  //Mission 2: parse the HTTP request (Fill #5 ~ #7) // Fill#5 Create input stream from socket to receive data from client  // Fill#6 Create output stream via socket to send data to client  // Fill#7, bufferedReader filter around the input stream to parse HTTP Request  //to Get a reference to the socket's input and output streams.  // <https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html>  InputStream is = socket.getInputStream(); //Mission 2: Get input stream from the socket  DataOutputStream os = new DataOutputStream(socket.getOutputStream()); //Mission 2: Get Output stream from the socket    // Set up input stream filters.  BufferedReader br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream())); //Mission 2: wrap InputStreamReader and BufferedReader filters around the input stream  // Mission 2(2-A, 2-B, 2-C): parse the HTTP request (Fill #8 ~ #9)  String requestLine = br.readLine(); // Mission 2: Get the request line of the HTTP request message  // Fill #9: Use StringTokenizer to HTTP request  // StringTokenizer from Java.util  // <https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html>  StringTokenizer tokenizer = **new** StringTokenizer(requestLine);  String method = tokenizer.nextToken(); // Mission 2-A: Get method information, Optional Exercises  String fileName = tokenizer.nextToken(); // Mission 2-B: Get URI information  String version = tokenizer.nextToken(); // Mission 2-C: Get HTTP Version information    // Prepend a "." so that file request is within the current directory.  fileName = "." + fileName ;    // Open the requested file.  FileInputStream fis = null ;  boolean fileExists = true ;  try {  fis = new FileInputStream(fileName);  } catch (FileNotFoundException e) {  fileExists = false ;  }  // Debug info for private use  System.out.println("Incoming!!!");  System.out.println(requestLine);  String headerLine = null;  while ((headerLine = br.readLine()).length() != 0) {  System.out.println(headerLine);  }  // Construct the response message.  String statusLine = null;  String contentTypeLine = null;  String contentLengthLine = null;  String entityBody = null;  /\*\*  \* Mission 3. Analyze the request and send an appropriate response  \* Mission 3. If HTTP response message consisting of the requested file, make the code with 200 OK  \* If the requested file is not present in the server, the server should send an HTTP “404 Not Found” message back to the client.  \* If the request message is not proper, the server should send an HTTP “400 BAD REQUEST” message back to the client.  \* and make more response codes for your HTTP web server  \* Optional Projects. Not only for the Method “GET”, you also have to consider handling other Methods.  \*/  if (fileExists) {  //Fill#10. When requested file exists, Status Code 200 OK  statusLine = "HTTP/1.1 200 OK" + CRLF; // Mission 3-A: Status Code 200 OK  contentTypeLine = "Content-Type: " + contentType(fileName) + CRLF;  contentLengthLine = "Content-Length: " + getFileSizeBytes(fileName) + CRLF;  } else {  #Fill#11. When requested file doesn’t exist, Status Code 404 NOT FOUND  statusLine = "HTTP/1.1 404 NOT FOUND" + CRLF; // Mission 3-B: Status Code 404 Not found  contentTypeLine = "Content-Type: text/html" + CRLF;  entityBody = "<HTML>" +  "<HEAD><TITLE>Not Found</TITLE></HEAD>" +  "<BODY>Not Found</BODY></HTML>";  }  //  / Mission 3-C(option): Status Code 400 Bad Request  statusLine = "HTTP/1.1 400 BAD REQUEST" + ***CRLF***;  contentTypeLine = "Content-Type: text/html" + ***CRLF***;  entityBody = "<HTML>" +  "<HEAD><TITLE>Bad Request</TITLE></HEAD>" +  "<BODY>400 Bad Request</BODY></HTML>";  // Send the status line.  os.writeBytes(statusLine);  // Send the content type line.  os.writeBytes(contentTypeLine);  // Send the content length line.  os.writeBytes(contentLengthLine);  // Send a blank line to indicate the end of the header lines.  os.writeBytes(CRLF);  // Send the entity body.  if (fileExists) {  sendBytes(fis, os);  fis.close();  } else {  os.writeBytes(entityBody) ; // Mission 3: Send appropriate entity body  }  // Close streams and socket.  os.close();  br.close();  socket.close();  }  **/\*\***  **\*** Method which sends the context  **\* @param fis** FileInputStream to transfer  **\* @param os** outputstream to client  **\*/**  private static void sendBytes(FileInputStream fis,  OutputStream os) throws Exception {  // Construct a 1K buffer to hold bytes on their way to the socket.  byte[] buffer = new byte[1024];  int bytes = 0;    // Copy requested file into the socket's output stream.  while ((bytes = fis.read(buffer)) != -1) {  os.write(buffer, 0, bytes);  }  }  /\*\*  \* Method to return appropriate  \* @param fileName  \*/  private static String contentType(String fileName) {  if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {  return "text/html";  }  /\*\*  \* Mission 4, create an HTTP response message consisting of the requested file preceded by header lines  \* Now, you are just handling text/html, is there any more context-types? Find and make codes for it.  \*/  #Fill 12 Detect appropriate file extensions and return appropriate response type(audio)  if (fileName.endsWith(".mp3")) {  return "audio";  }    #Fill 13 Detect appropriate file extensions and return appropriate response type(image)  if (fileName.endsWith(".jpeg") || fileName.endsWith(".jpg")) {  return "image";  }  }  /\*\*  \* Get the File name, and through the file name, get the size of the file.  \*.@param fileName  \*/  private static long getFileSizeBytes(String fileName) throws IOException {  File file = new File(fileName);  return file.length();  }  // This method returns the size of the specified file in bytes.  } |
|  |

**Result**

****

**Flow chart of Web Server and Web client**

텍스트, 도표, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

**Block diagram of Web Server**

텍스트, 폰트, 영수증, 흑백이(가) 표시된 사진

자동 생성된 설명

**Instructions: How to run the program**

1. Run Web Server.
2. Run Web Client.
3. HttpRequest.java will create HTTP response message.
4. Web Server sends response message to client.

**Wireshark**

172.16.162.201: Client IP, 104.103.68.40: Server IP

Server HTTP Request



If you look at packet number 18 in the picture above, the server sends an HTTP response called HTTP/1.1 200 OK to the client.



If you look at packet number 47 in the picture above, the server sends an HTTP response called HTTP/1.1 200 OK to the client.

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

**How the program works**

1. WebServer.java

1. Check the port number through the command line

2. Create a socket and wait until it is connected to the client

3. If connected, send an HTTP response message through the HttpRequest class

4. Start a thread

2. HttpRequest.java

1. Open input and output streams on the Socket so that it can read HTTP requests.

2. Analyze the HTTP request to determine the requested file path.

3. Check if the file actually exists and include a status message based on its existence.

4. Send the content.

5. Close the socket.