5COSC022W.2 Client-Server Architectures

Tutorial Week 04: Socket Programming

EXERCISE 1: LOGGING

In this tutorial, you will enhance tutorial Week 03 exercise 2 chat application by adding logging functionalities. Logging is an essential aspect of software development as it helps in tracking the application's behavior, debugging issues, and maintaining records of events. We will use java.util.logging.Logger to log messages and save them to a file for future reference.

STEP 1 – IMPORT THE PROJECT

Import the zip file from the 2nd exercise of tutorial week 3 into your IDE. This will provide you with the base code for the client-server chat application.

STEP 2 - **SET UP LOGGING**

Add Logger to Classes: In each class (e.g., SimpleChatClient and SimpleChatServer), import java.util.logging.Logger and create a Logger instance.

```
import java.util.logging.Logger;
private static final Logger logger =
Logger.getLogger(ClassName.class.getName());
```

STEP 3 - **SET UP FILEHANDLER**

Configure a FileHandler to save logs to a text file. This should be done in a static block to ensure it is set up when the class is loaded.

```
static {
        try {
            FileHandler fileHandler = new FileHandler("logs.txt", true);
            logger.addHandler(fileHandler);
        } catch (IOException e) {
            logger.log(Level.SEVERE, "Failed to set up file handler for
logger", e);
        }
    }
```

STEP 3- **CONFIGURE THE LOGGER**

- Specify Log File Name: Use FileHandler to specify the log file name and set it to append mode.

- Format Log Messages: Use SimpleFormatter to format the log messages for readability.
- Add FileHandler to Logger: Attach the FileHandler to the Logger instance.

STEP 4 - **ADD LOGGING STATEMENTS**

- **Log Informational Messages:** Add logging statements at key points in your code to log informational messages. For example, log when the server starts, when a client connects, and when messages are sent or received.

```
logger.info("Server is running and waiting for a client to connect...");
logger.info("Client connected: " + clientSocket.getInetAddress());
logger.info("Received from client: " + message);
logger.info("Sent response to client: " + responseMessage);
```

- **Log Exceptions:** Use logger.log(Level.SEVERE, "message", exception) to log exceptions.

```
} catch (IOException e) {
    logger.log(Level.SEVERE, "IOException occurred", e);
}
```

STEP 5 - **TEST THE APPLICATION**

- **Run the Server and Client Applications:** Start the server application first, then run the client application to connect to the server.
- **Check the Log Files:** Verify that the log files (logs.txt) are created and contain the expected log messages. This will help you ensure that all relevant events are being logged.

EXERCISE 2 – HTTP SERVER

In this exercise, you will learn how to build a basic HTTP client-server application in Java. The server will handle incoming HTTP GET requests and send back a simple text response. The client will send a GET request to the server and print the server's response.

PROJECT STRUCTURE

Please create a new project and call it as Tutorial-week04-EX2. Then, create two Java files:

1. **MyHttpServer.java:** Contains the code for the HTTP server.

2. **MyHttpClient.java:** Contains the code for the HTTP client.

STEP 1: CREATE THE SERVER CLASS AND MAIN METHOD

```
// Filename: MyHttpServer.java

// Import necessary classes for HttpServer, InetSocketAddress
import ...

CLASS MyHttpServer
    METHOD main(String[] args)
        // Start the server logic here (see next steps)
    END METHOD
END CLASS
```

Task	Description
Import Statements	Import `HttpServer` and `InetSocketAddress`. Use appropriate import statements for these classes.
Class Declaration	Create a public class named `MyHttpServer`.
Main Method Declaration	Create the `public static void main(String[] args)` method. This is where your server application will begin execution.

STEP 2: CREATE THE HTTPSERVER INSTANCE

```
// Inside the main method:
BEGIN TRY
    server = CREATE HttpServer OBJECT, LISTENING ON PORT 8080, DEFAULT BACKLOG
    PRINT "Server starting on port 8080"

    // ... further server setup (see next steps)

EXCEPTION IOException e
    PRINT "Error starting the server: " + e.MESSAGE
    PRINT STACK TRACE OF e
END TRY
```

Task	Description
Create `HttpServer`	Create an instance of `HttpServer` using `HttpServer.create()`.
Specify Port and Backlog	Provide a `new InetSocketAddress(8080, 0)` to the `create()` method, where 8080 is the port and 0 indicates the default backlog value.
Start Server Message	Print "Server starting on port 8080"
Error Handling	Use a `try-catch` block to handle any `IOException` that might occur during server creation. Print an error message and the stack trace if an exception occurs.

STEP 3: CREATE THE HTTPHANDLER (INNER CLASS)

Code snippet

```
// Inside the MyHttpServer class, but outside the main method:

CLASS MyHandler IMPLEMENTS HttpHandler
    METHOD handle(HttpExchange exchange)
        // Request handling logic will go here (see next step)
    END METHOD
END CLASS
```

Task	Description
Create `MyHandler` Class	Create a static inner class named `MyHandler` that implements the `HttpHandler` interface.
Implement `handle` Method	Override the `public void handle(HttpExchange exchange)` method. This is where you will process incoming HTTP requests.

STEP 4: IMPLEMENT REQUEST HANDLING IN A SEPARATE THREAD

```
// Inside the MyHandler class:
METHOD handle(HttpExchange exchange)
        CREATE RequestHandlerThread OBJECT, PASSING exchange
        START requestThread
END METHOD

// (Outside MyHandler, but within MyHttpServer)
CLASS RequestHandlerThread EXTENDS Thread
    DECLARE PRIVATE MEMBER HttpExchange exchange

    CONSTRUCTOR RequestHandlerThread(HttpExchange exchange)
        ASSIGN exchange TO this.exchange
    END CONSTRUCTOR

    METHOD run()
        // Request processing and response logic will go here
    END METHOD
END CLASS
```

Task	Description
Create and Start `RequestHandlerThread`	Inside the `handle` method, create a new instance of `RequestHandlerThread` (which you'll define next) and pass the `HttpExchange` to it. Then, start the thread.
Create `RequestHandlerThread` Class	Create another static inner class `RequestHandlerThread` that extends `Thread`.
`RequestHandlerThread` Member	Declare a private member variable of type `HttpExchange` to store the exchange object within the thread.
`RequestHandlerThread` Constructor	Create a constructor for `RequestHandlerThread` that takes an `HttpExchange` object as an argument and assigns it to the member variable.
Implement `run` Method	Override the `public void run()` method. This is where the actual request processing logic will reside.

STEP 5: IMPLEMENT REQUEST PROCESSING AND RESPONSE

Please include the following codes insdie *RequestHandlerThread* class:

```
        @Override
        public void run() {
            try {
                // Get request information
                String requestMethod = exchange.getRequestMethod();
                URI requestURI = exchange.getRequestURI();
                Headers requestHeaders = exchange.getRequestHeaders();

                // Display request information
                System.out.println("\n=== INCOMING REQUEST ===");
                System.out.println("Method: " + requestMethod);
                System.out.println("URI: " + requestURI);
                System.out.println("Headers:");
                printHeaders(requestHeaders);

                // Prepare response with request information
                StringBuilder responseContent = new StringBuilder();
                responseContent.append("Request Details:\n");
                responseContent.append("Method:
").append(requestMethod).append("\n");
                responseContent.append("URI:
").append(requestURI).append("\n");
                responseContent.append("Headers:\n");
                requestHeaders.forEach((key, value) ->
                    responseContent.append(key).append(":
").append(value).append("\n"));
                responseContent.append("\nHandled by thread:
").append(Thread.currentThread().getName());

                String response = responseContent.toString();

                // Get response headers
                Headers responseHeaders = exchange.getResponseHeaders();

                // Send response
                exchange.sendResponseHeaders(200,
response.getBytes().length);

                // Display response information
                System.out.println("\n=== OUTGOING RESPONSE ===");
                System.out.println("Status: 200 OK");
                System.out.println("Headers:");
                printHeaders(responseHeaders);
                System.out.println("Body length: " +
response.getBytes().length + " bytes");

                // Send response body
                try (OutputStream os = exchange.getResponseBody()) {
                    os.write(response.getBytes());
                }

            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                exchange.close();
            }
        }
```

STEP 6: IMPLEMENT PRINTHEADERS METHOD

```
// Inside the MyHttpServer class (this can be outside the main method,
//  but inside the class, potentially inside the RequestHandlerThread class,
//  or at the top level of MyHttpServer)

METHOD printHeaders(Headers headers)  // Note: Headers, not Map
    FOR EACH (key, value) PAIR IN headers DO
        PRINT key + ": " + JOIN THE STRINGS IN value WITH ", " AS SEPARATOR
    END FOR
END METHOD
```

Task	Description
Method Signature	Create a method named `printHeaders` that takes a `Headers` object (from `com.sun.net.httpserver.Headers`) as input. This is different from the client-side version.
`forEach` with Lambda	Use the `forEach` method of the `Headers` object. This method takes a lambda expression.
Lambda Parameters	The lambda expression takes two parameters: `key` (the header name) and `value` (the list of header values). These are automatically typed.
Print Key-Value Pair	Inside the lambda, print the key followed by a colon and a space. Then, use `String.join(", ", value)` to concatenate the strings in the `value` list, separated by commas, and print the result.

STEP 7: CREATE A CONTEXT AND START THE SERVER

```
// Back in the main method, inside the try block:

CREATE CONTEXT "/hello" ASSOCIATED WITH NEW MyHandler OBJECT
START server
```

Task	Description
Create Context	Use `server.createContext()` to associate the path "/hello" with a new instance of your `MyHandler` class.
Start Server	Start the server using `server.start()` .

STEP 8: CREATE THE CLIENT CLASS AND MAIN METHOD

```
// Filename: MyHttpClient.java

// Import necessary classes: BufferedReader, IOException, InputStreamReader, HttpUR
import ...

CLASS MyHttpClient
    METHOD main(String[] args)
        // Client logic will go here (see next steps)
    END METHOD
END CLASS
```

Task	Description
Import Statements	Import `BufferedReader`, `IOException`, `InputStreamReader`, `HttpURLConnection`, and `URL`. Use appropriate import statements.
Class Declaration	Create a public class named `MyHttpClient`.
Main Method Declaration	Create the `public static void main(String[] args)` method. This is where your client application will begin execution.

STEP 9: CREATE THE URL AND OPEN A CONNECTION

```
// Inside the main method:
BEGIN TRY
    url = CREATE URL OBJECT WITH "http://localhost:8080/hello"
    con = OPEN CONNECTION ON url, CAST TO HttpURLConnection

    // ... rest of the client logic

EXCEPTION IOException e
    PRINT "Error during connection: " + e.getMessage()
    PRINT STACK TRACE of e
END TRY
```

Task	Description
Create `URL` Object	Create a `URL` object with the server's address and the path: `"http://localhost:8080/hello"`.
Open Connection	Use `url.openConnection()` to open a connection to the URL. Cast the result to `HttpURLConnection`.
Error Handling	Use a `try-catch` block to handle any `IOException` that might occur during the connection process. Print an error message and the stack trace if an exception occurs.

STEP 10: IMPLEMENTING PRINT HEADER METHOD

```
// Inside the MyHttpClient class, replacing the previous printHeaders method

METHOD printHeaders(Map<String, List<String>> headers)
    FOR EACH (key, value) PAIR IN headers DO
        IF key IS NOT NULL THEN
            PRINT key + ": " + JOIN THE STRINGS IN value WITH ", " AS SEPARATOR
        END IF
    END FOR
END METHOD
```

Task	Description
Method Signature	The method signature remains the same: `private static void printHeaders(Map<String, List<String>> headers)`.
`forEach` with Lambda	Use the `forEach` method of the `Map` interface. This method takes a lambda expression as an argument.
Lambda Parameters	The lambda expression takes two parameters: `key` (the header name) and `value` (the list of header values).
Null Key Check	Inside the lambda, check if the `key` is not null. This is a good practice because some HTTP headers might have a null key (though it's uncommon).
Print Key-Value Pair	If the key is not null, print the key followed by a colon and a space. Then, use `String.join(", ", value)` to concatenate the strings in the `value` list, separated by commas. Print the result.

STEP 11: GET AND DISPLAY DEFAULT REQUEST HEADERS

```
PRINT "DEFAULT REQUEST HEADERS:"
defaultHeaders = GET REQUEST PROPERTIES FROM con  // Returns a Map
CALL printHeaders(defaultHeaders)  // Calls the *client-side* printHeaders
```

Task	Description
Print Header Label	Print the string "DEFAULT REQUEST HEADERS:" to the console.
Get Default Headers	Use `con.getRequestProperties()` to get a `Map<String, List<String>>` containing the default request headers. Store this map in `defaultHeaders`.
Call `printHeaders`	Call the `printHeaders` method that is defined *within the* `MyHttpClient` *class* (the one that takes a `Map<String, List<String>>` as an argument), passing `defaultHeaders`.

STEP 12: SET THE REQUEST METHOD AND GET THE RESPONSE CODE

```
// Inside the main method, inside the try block:

SET REQUEST METHOD OF con TO "GET"
responseCode = GET RESPONSE CODE FROM con
PRINT "Response Code: " + responseCode
```

Task	Description
Set Request Method	Use `con.setRequestMethod("GET")` to set the HTTP request method to GET.
Get Response Code	Use `con.getResponseCode()` to get the HTTP response code from the server. Store it in an integer variable named `responseCode`.
Print Response Code	Print the `responseCode` to the console.

STEP 13:

```
// Inside the main method, inside the try block:

IF responseCode EQUALS HTTP_OK THEN
    in = CREATE BufferedReader, WRAPPING AN InputStreamReader, READING FROM con INP

    DECLARE String inputLine
    response = CREATE StringBuilder OBJECT

    WHILE (READ LINE FROM in INTO inputLine) IS NOT NULL
        APPEND inputLine TO response
    END WHILE

    CLOSE in
    PRINT "Response: " + CONVERT response TO STRING
ELSE
    PRINT "GET request not worked"
END IF
```

Task	Description
Check Response Code	Check if the `responseCode` is equal to `HttpURLConnection.HTTP_OK` (which represents the HTTP 200 OK status code).
Create `BufferedReader`	If the response code is OK, create a `BufferedReader` to read from the connection's input stream. Wrap an `InputStreamReader` around `con.getInputStream()` to convert bytes to characters.
Read Response Line by Line	Use a `while` loop to read the response from the `BufferedReader` line by line. Use a `StringBuilder` to efficiently build the complete response string.
Close `BufferedReader`	Close the `BufferedReader` using `in.close()`.
Print Response	Print the complete response string (obtained from the `StringBuilder`) to the console.
Handle Non-OK Response	If the response code is not OK, print an error message to the console.

STEP 14: RUN AND TEST THE PROGRAM

RUNNING THE SERVER

1. Right-click on MyHttpServer.java in the Projects window.
2. Select "Run File."
3. The server will start, and you should see the "Server starting on port 8080" message in the Output window (usually at the bottom).

RUNNING THE CLIENT

1. Right-click on MyHttpClient.java in the Projects window.
2. Select "Run File."
3. A new Output window will appear, showing the client's output, including the response code and the server's message.

STEP 15: TEST THE PROGRAM IN A WEB BROWSER

Open a browser and go to http://localhost:8080/hello). The browser will interact with your server running from within NetBeans.

STEP 16: OBSERVING OUTPUT

- **Server Output:** When the server receives a request (either from MyHttpClient or from the web browser), you will likely see some additional output in the server's terminal or NetBeans Output window, indicating that a request has been handled. You might also see the thread name printed as part of the response string.
- **Client Output:** The MyHttpClient program will print the response code and the response message to its console (either the terminal or the NetBeans Output window).
- **Web Browser:** The web browser will display the server's response message in the browser window.