Java Web Server Project Documentation

Mariam Elwirish (202421646)

Elie Jbara (202304537)

1 Overview

- The web server was implemented in Java using TCP sockets (through `java.net` package) and multithreading.
- It listens on port **6789**, accepts client connections, and processes each request in a separate thread.
- The server:
  - Supports receiving, processing, and returning proper HTTP status lines and headers.
  - Has a landing page `index.html` such that when you open `http://localhost:6789/`, it redirects you to this landing page.
  - Supports handling HTTP GET requests and serving static files such as HTML, images, and other file types. We created multiple test files to verify this behavior (`test1.txt`, `test2.html`, and a missing file), all accessible from the server's landing page.
    - `test1.txt`**:** This file is intentionally left without a MIME mapping. Since our server does not define a content type for the `.txt` extension, it falls back to the default `application/octet-stream` type, causing the browser to download the file. This demonstrates the server's correct fallback behavior.
    - `test2.html`**:** This file loads normally in the browser because the server correctly recognizes the `.html` extension and responds with the proper `text/html` MIME type.
    - **Missing file test:** We also added a link to a non-existent file to verify error handling. When requested, the server returns an HTTP `404 Not Found` status and serves our custom `error404.html` page, demonstrating correct handling of invalid or missing resources.
  - Supports handling 404 Not Found when files are missing (we created an HTML page – called `error404.html` – to show when such an error occurs for a better experience).
- The implementation consists of two classes:
  - `WebServer.java` – creates the listening socket and spawns threads
  - `HttpRequest.java` – handles each client request independently

---

2 `WebServer.java` – Main Server Logic.

2.1 Creating the Listening Socket.

- The main method creates a `ServerSocket` bound to port 6789:

```
ServerSocket socket = new ServerSocket(port);
```

- This initializes the server and allows it to accept incoming TCP connections.

2.2 Infinite Accept Loop (Server Always On).

- The server then runs an infinite loop:

```
while (true) {
    Socket client = socket.accept();
     ...
}
```

- `accept()` blocks until a client (browser) connects, making server always listening and waiting for connections.
- For every connection, a new `HttpRequest` object is created.

2.3 Multithreading.

- To allow multiple clients to be served in parallel, each request is handled in a separate thread:

```
HttpRequest request = new HttpRequest(client);
Thread thread = new Thread(request);
thread.start();
```

- This prevents slow or hanging clients from blockingthe entire server.

---

3 `HttpRequest.java` – Handling HTTP Requests.

3.1 `run()` method – Thread Entry Point.

3.1.1 Purpose

- The `HttpRequest` class implements `Runnable` for each instance runs in its own thread.
- So, the `run()` method is the entry point for handling a single client connection.

3.1.2 Logic.

- `run()` simply delegates the real work to `processRequest()` and catches `Exceptions`, if any.

```
public void run() {
    try {
        processRequest();
    } catch (Exception e) {
        System.out.println("Error processing request: " + e);
    }
}
```

3.2 `processRequest()` – Core HTTP Request Handling.

3.2.1 Purpose.

It performs all steps required to process the HTTP request and generate the HTTP response.

3.2.2 Setting Up the Input and Output Streams.

```
InputStream is = socket.getInputStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);

OutputStream os = socket.getOutputStream();
```

- `InputStream` reads raw bytes from the socket.
- `InputStreamReader` converts bytes → characters.
- `BufferedReader` allows reading lines (GET /index.html HTTP/1.1).
- `OutputStream` for sending raw bytes back.

3.2.3 Reading the HTTP Request Line and Headers.

- Now, we need to extract the browser's request, such as:

```
GET / HTTP/1.1
Host: localhost:6789
User-Agent: Chrome
...
```

- We read the first line = request line.
- Then read header lines until a blank line is reached.
- *Note:* Synchronized block prevents threads from mixing output, just for nicer console printing.

```
synchronized(System.out) {
    String requestLine = br.readLine();
    System.out.println("\n----------------- HTTP Request -----------------");
    System.out.println(requestLine);

    String headerLine;
    while ((headerLine = br.readLine()).length() != 0) {
        System.out.println(headerLine);
    }
    ...
}
```

3.2.4 Extracting the Requested Filename

- Now we need to figure out which file the browser is asking for.
  - **Example:** `GET /index.html HTTP/1.1` → the file is `/index.html`
  - The server serves it from its local directory, so it becomes `./index.html`.
- `StringTokenizer` splits the request line.
  - Skip the "GET".
  - Extract the second token (the path).
  - Prepend "." to force local directory access.

```
StringTokenizer tokens = new StringTokenizer(requestLine);
tokens.nextToken();
String fileName = tokens.nextToken();
fileName = "." + fileName;
```

3.2.5 Checking if the Requested File Exists.

- Now we need to determine whether the file exists or not and we send:
  - a **200 OK** response if the file is found.
  - a **404 Not Found** response if the file is missing.
- We try opening the file with FileInputStream.
  - If it succeeds, then the file exists.
  - if it fails, then the file doesn't exist. In this case, we open the file `error404.html` so that an error page shows (for nice view).

```
        FileInputStream fis = null;
        boolean fileExists = true;
        try {
            fis = new FileInputStream(fileName);
        } catch (FileNotFoundException e) {
            fis = new FileInputStream("./error404.html");
            fileExists = false;
    }
```

3.2.6 Building the HTTP Response

- Now we need to prepare the response, which will consist of:
  - The status line (200 OK or 404 Not Found).
  - Header line(s), in our case we are only using `Content-Type` line, which is either the requested file's type if the files exists, or `text/html` if not found, since we are gonna display `./error404.html` in such a case.
  - Blank Line (`CRLF`).
  - Body (the requested file if found, or `./error404.html` if not.
- Then, we convert the response into bytes and send it through the output stream `os`.
- Here, we used `contentType()` and `sendBytes helpers` helpers, which are explained below.

```
String statusLine;
String contentTypeLine;
if (fileExists) {
    statusLine = "HTTP/1.1 200 OK" + CRLF;
    contentTypeLine = "Content-type: " + contentType(fileName) + CRLF;
} else {
    statusLine = "HTTP/1.0 404 Not Found" + CRLF;
    contentTypeLine = "Content-type: text/html" + CRLF;
}
os.write(statusLine.getBytes());
os.write(contentTypeLine.getBytes());
os.write(CRLF.getBytes());
sendBytes(fis, os);
```

---

4 contentType() — Determining the MIME Type

- Because web browsers rely on the Content-Type header to understand how to render the returned file, the server determines the correct MIME type based on the file extension.
  - .html → text/html
  - .jpg → image/jpeg
  - .png → image/png
  - .css → text/css ...
- However, if there is a file that doesn't have MIME mapping, it falls back to `octet-stream`, which represents generic binary data and downloads it on the user's device.

```
if (fileName.endsWith(".html")) return "text/html";
if (fileName.endsWith(".jpg"))  return "image/jpeg";
...
return "application/octet-stream";
```

---

5 sendBytes() — Sending File Contents.

- This method handles transmitting the actual file content to the browser.
- It reads the file in 1 KB chunks and writes them into the output stream.

```
byte[] buffer = new byte[1024];
int bytes = 0;

while ((bytes = fis.read(buffer)) != -1) {
    os.write(buffer, 0, bytes);
}
```

---

6 Lessons Learned & Challenges.

1. **Default MIME type (`application/octet-stream`).** Understanding what an `octet-stream` is was initially confusing. However, we figured this out by adding a file that is not mapped and trying to access it. This made us get what it actually does. This is why we kept `test1.txt` as it really helped us get a sense of this concept. Also, we searched for the concept to understand. The resources are attached in the last section.
2. **Thread-safe logging.** Because we are trying to log what happens on the server and display it on the console, we faced an issue when multiple clients connected, header lines overlapped in the console. Using `synchronized(System.out)` helped us solve that and provide a clear output and logging, making everything understandable and helping us understand what actually happens behind the scenes.
3. **Handling "/" requests.** Browsers automatically request "/" when no file is specified. Because we chose to build a real-like web page, mapping it to `/index.html` wasn't challenging in a sense that it is hard but required some attention and extra work from us. Doing such a thing provided a proper landing page and helped us navigate everything easily. Same thing applies for the other pages on the site.
4. **Multiple parallel connections from the browser.** When we first accessed the server from Chrome, we observed multiple 'New connection received' messages appearing in ur console logs. We thought this was an error at the beginning, but after some search, we found out modern browsers open several parallel TCP connections (typically up to 6 per domain in Chrome) to improve performance when fetching resources from a server. So this behavior is normal.
5. **Browser caching during testing.** Chrome cached old files sometimes and kept showing old cached files after we modified our files. This made us confused at the beginning about why the modification are not showing. However, we realized that later and cleared cache.
6. **Correct HTTP formatting and CRLF ordering.** The order "status line → headers → blank line → body" must be exact. Any missing or incorrect `\r\n` caused the browser to reject the response entirely.

---

7 Resources.

1. What are Max Parallel HTTP Connections in a Browser?
2. Common media types
3. We used ChatGPT's assistance to explain some concepts, debug some errors, and to assist with writing the HTML code for the server pages.