11. HTTP 서버 만들기

#0.강의/1.자바로드맵/6.자바-고급2편

HTTP 서버1 - 시작

HTTP 서버를 직접 만들어보자.

웹 브라우저에서 우리 서버에 접속하면 다음과 같은 HTML을 응답하는 것이다.

```
<h1>Hello World</h1>
```

그러면 웹 브라우저가 Hello World를 크게 보여줄 것이다.

참고

- HTML은 `<html>`, `<head>`, `<body>`와 같은 기본 태그를 가진다. 원래는 이런 태그도 함께 포함해서 전달해야 하지만, 예제를 단순하게 만들기 위해, 최소한의 태그만 사용하겠다.
- HTML에 대한 자세한 내용은 강의에서 설명하지 않는다. 참고로 HTML을 잘 몰라도 강의 내용은 충분히 이해할 수 있을 것이다.

```
package was.v1;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
```

```java
import static java.nio.charset.StandardCharsets.*;
import static util.MyLogger.log;

public class HttpServerV1 {

    private final int port;

    public HttpServerV1(int port) {
        this.port = port;
    }

    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        log("서버 시작 port: " + port);

        while (true) {
            Socket socket = serverSocket.accept();
            process(socket);
        }
    }

    private void process(Socket socket) throws IOException {
        try (socket;
             BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream(), UTF_8));
             PrintWriter writer = new PrintWriter(socket.getOutputStream(),
false, UTF_8)) {

            String requestString = requestToString(reader);
            if (requestString.contains("/favicon.ico")) {
                log("favicon 요청");
                return;
            }
            log("HTTP 요청 정보 출력");
            System.out.println(requestString);

            log("HTTP 응답 생성중...");
            sleep(5000); // 서버 처리 시간
            responseToClient(writer);
            log("HTTP 응답 전달 완료");
        }
    }
```

```
    private String requestToString(BufferedReader reader) throws IOException {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            if (line.isEmpty()) {
                break;
            }
            sb.append(line).append("\n");
        }
        return sb.toString();
    }

    private void responseToClient(PrintWriter writer) {
        // 웹 브라우저에 전달하는 내용

        String body = "<h1>Hello World</h1>";
        int length = body.getBytes(UTF_8).length;

        StringBuilder sb = new StringBuilder();
        sb.append("HTTP/1.1 200 OK\r\n");
        sb.append("Content-Type: text/html\r\n");
        sb.append("Content-Length: ").append(length).append("\r\n");
        sb.append("\r\n"); // header, body 구분 라인
        sb.append(body);

        log("HTTP 응답 정보 출력");
        System.out.println(sb);

        writer.println(sb);
        writer.flush();
    }

    private void sleep(int millis) {
        try {
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

}
```

- HTTP 메시지의 주요 내용들은 문자로 읽고 쓰게 된다.

- 따라서 여기서는 `BufferedReader`, `PrintWriter`를 사용했다.
- `Stream`을 `Reader`, `Writer`로 변경할 때는 항상 인코딩을 확인하자.

AutoFlush

`new PrintWriter(socket.getOutputStream(), false, UTF_8)`

- `PrintWriter`의 두 번째 인자는 `autoFlush` 여부이다.
- 이 값을 `true`로 설정하면 `println()`으로 출력할 때 마다 자동으로 플러시 된다.
  - 첫 내용을 빠르게 전송할 수 있지만, 네트워크 전송이 자주 발생한다.
- 이 값을 `false`로 설정하면 `flush()`를 직접 호출해주어야 데이터를 전송한다.
  - 데이터를 모아서 전송하므로 네트워크 전송 횟수를 효과적으로 줄일 수 있다. 한 패킷에 많은 양의 데이터를 담아서 전송할 수 있다.
- 여기서는 `false`로 설정했으므로 마지막에 꼭 `writer.flush()`를 호출해야 한다.

requestToString()

HTTP 요청을 읽어서 `String`으로 반환한다.

HTTP 요청의 시작 라인, 헤더까지 읽는다.

- `line.isEmpty()`이면 HTTP 메시지 헤더의 마지막으로 인식하고 메시지 읽기를 종료한다.
- HTTP 메시지 헤더의 끝은 빈 라인으로 구분할 수 있다. 빈 라인 이후에는 메시지 바디가 나온다.
- 참고로 여기서는 메시지 바디를 전달하지 않으므로 메시지 바디의 정보는 읽지 않는다.

/favicon.ico

웹 브라우저에서 해당 사이트의 작은 아이콘을 추가로 요청할 수 있다. 여기서는 사용하지 않으므로 무시한다.

sleep(5000); // 서버 처리 시간

예제가 단순해서 응답이 너무 빠르다. 서버에서 요청을 처리하는데 약 5초의 시간이 걸린다고 가정하자.

responseToClient()

HTTP 응답 메시지를 생성해서 클라이언트에 전달한다.

시작라인, 헤더, HTTP 메시지 바디를 전달한다.

HTTP 공식 스펙에서 다음 라인은 `\r\n`(캐리지 리턴 + 라인 피드)로 표현한다. 참고로 `\n`만 사용해도 대부분의 웹 브라우저는 문제없이 작동한다.

마지막에 `writer.flush()`를 호출해서 데이터를 전송한다.

```
package was.v1;
```

```java
import java.io.IOException;

public class ServerMainV1 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        HttpServerV1 server = new HttpServerV1(PORT);
        server.start();
    }
}
```

웹 브라우저를 실행하고 다음 사이트에 접속해보자. 5초간 기다려야 한다.

- http://localhost:12345
- http://127.0.0.1:12345

크롬 웹 브라우저에서 마우스 오른쪽 버튼을 눌러서 소스 보기를 하면 다음 결과를 확인할 수 있다.

실행 결과 - 소스 보기

```
<h1>Hello World</h1>
```

실행 결과

```
11:02:07.375 [     main] 서버 시작 port: 12345
11:02:09.671 [     main] HTTP 요청 정보 출력
GET / HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: ko,en;q=0.9,en-US;q=0.8,ko-KR;q=0.7

11:02:09.672 [     main] HTTP 응답 생성중...
11:02:14.678 [     main] HTTP 응답 정보 출력
HTTP/1.1 200 OK
Content-Type: text/html
```

```
Content-Length: 20

<h1>Hello World</h1>
11:02:14.680 [     main] HTTP 응답 전달 완료
14:39:09.761 [     main] favicon 요청
14:39:09.762 [     main] favicon 요청
```

- 실행 결과에서 일부 헤더들은 제외했다.

HTTP 요청 메시지

```
GET / HTTP/1.1
Host: localhost:12345
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/128.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: ko,en;q=0.9,en-US;q=0.8,ko-KR;q=0.7
```

`http://localhost:12345`를 요청하면 웹 브라우저가 HTTP 요청 메시지를 만들어서 서버에 전달한다.

시작 라인

- `GET`: GET 메서드 (조회)
- `/`: 요청 경로, 별도의 요청 경로가 없으면 `/`를 사용한다.
- `HTTP/1.1`: HTTP 버전

헤더

- `Host`: 접속하는 서버 정보
- `User-Agent`: 웹 브라우저의 정보
- `Accept`: 웹 브라우저가 전달 받을 수 있는 HTTP 응답 메시지 바디 형태
- `Accept-Encoding`: 웹 브라우저가 전달 받을 수 인코딩 형태
- `Accept-Language`: 웹 브라우저가 전달 받을 수 있는 언어 형태

참고: 각 헤더에 대한 자세한 내용은 **모든 개발자를 위한 HTTP 기본 웹 지식** 강의를 참고하자.

HTTP 응답 메시지

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 20

<h1>Hello World</h1>
```

시작 라인

- `HTTP/1.1` : HTTP 버전
- `200` : 성공
- `OK` : `200` 에 대한 설명

헤더

`Content-Type`

- HTTP 메시지 바디의 데이터 형태, 여기서는 HTML을 사용

`Content-Length`

- HTTP 메시지 바디의 데이터 길이

바디

`<h1>Hello World</h1>`

문제

서버는 동시에 수 많은 사용자의 요청을 처리할 수 있어야 한다.

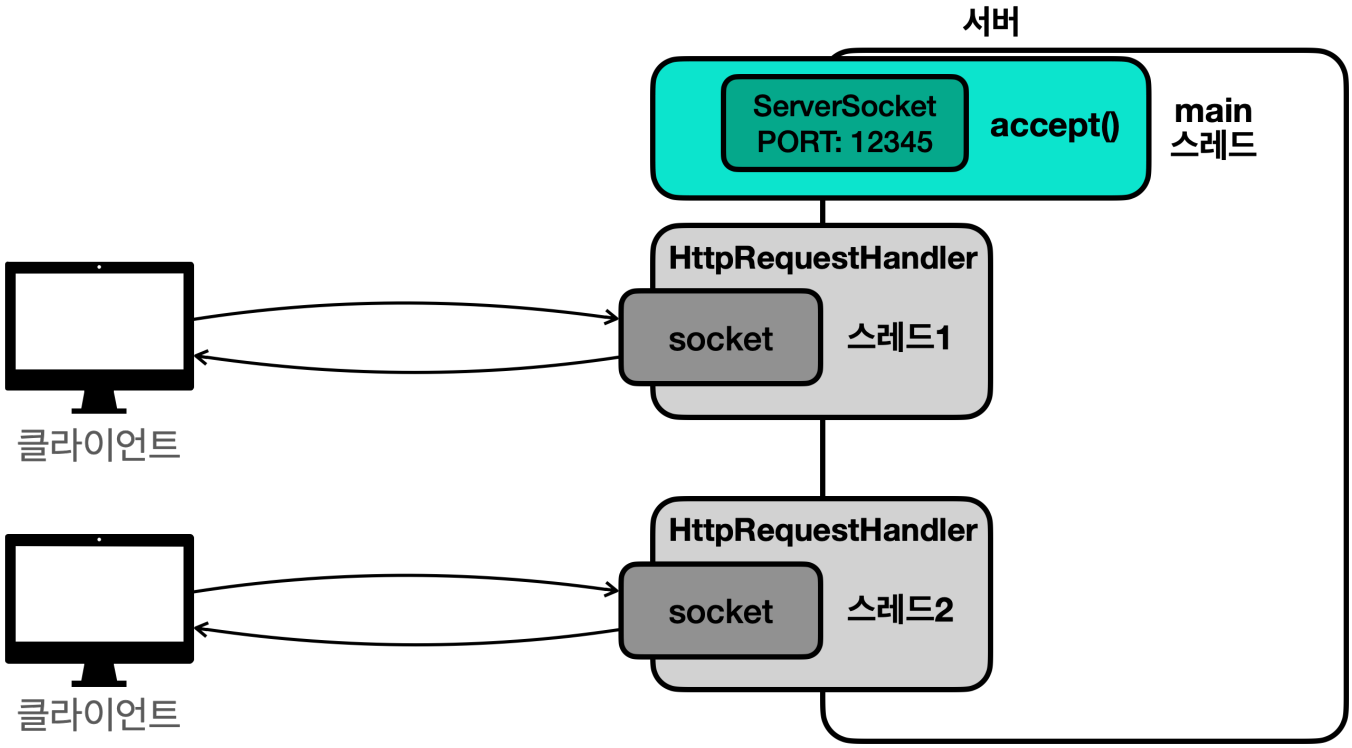
현재 서버는 한 번에 하나의 요청만 처리할 수 있다는 문제가 있다.

다른 웹 브라우저 2개를 동시에 열어서 사이트를 실행해보자. (예를 들어서 크롬, 엣지, 사파리 등 다른 브라우저를 열어야 확실한 테스트가 가능하다)

첫 번째 요청이 모두 처리되고 나서 두 번째 요청이 처리되는 것을 확인할 수 있다.

HTTP 서버2 - 동시 요청

스레드를 사용해서 동시에 여러 요청을 처리할 수 있도록 서버를 개선해보자.




```java
package was.v2;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

import static java.nio.charset.StandardCharsets.UTF_8;
import static util.MyLogger.log;

public class HttpRequestHandlerV2 implements Runnable {
    private final Socket socket;

    public HttpRequestHandlerV2(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            process();
        } catch (Exception e) {
            log(e);
```

```
        }
    }

    private void process() throws IOException {
        try (socket;
             BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream(), UTF_8));
             PrintWriter writer = new PrintWriter(socket.getOutputStream(),
false, UTF_8)) {

            String requestString = requestToString(reader);
            if (requestString.contains("/favicon.ico")) {
                log("favicon 요청");
                return;
            }
            log("HTTP 요청 정보 출력");
            System.out.println(requestString);

            log("HTTP 응답 생성중...");
            sleep(5000);
            responseToClient(writer);
            log("HTTP 응답 전달 완료");
        }
    }

    private String requestToString(BufferedReader reader) throws IOException {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            if (line.isEmpty()) {
                break;
            }
            sb.append(line).append("\n");
        }
        return sb.toString();
    }

    private void responseToClient(PrintWriter writer) {
        // 웹 브라우저에 전달하는 내용
        String body = "<h1>Hello World</h1>";
        int length = body.getBytes(UTF_8).length;

        StringBuilder sb = new StringBuilder();
```

```
            sb.append("HTTP/1.1 200 OK\r\n");
            sb.append("Content-Type: text/html\r\n");
            sb.append("Content-Length: ").append(length).append("\r\n");
            sb.append("\r\n"); // header, body 구분 라인
            sb.append(body); // header, body 구분 라인

            log("HTTP 응답 정보 출력");
            System.out.println(sb);

            writer.println(sb);
            writer.flush();
        }

        private void sleep(int millis) {
            try {
                Thread.sleep(millis);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }

}
```

- 클라이언트의 `HttpRequestHandler`는 이름 그대로 클라이언트가 전달한 HTTP 요청을 처리한다.
- 동시에 요청한 수 만큼 별도의 스레드에서 `HttpRequestHandler`가 수행된다.
- 참고로 `HttpRequestHandlerV1`은 없다.

```
package was.v2;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import static util.MyLogger.log;

public class HttpServerV2 {

    private final ExecutorService es = Executors.newFixedThreadPool(10);
```

```
    private final int port;

    public HttpServerV2(int port) {
        this.port = port;
    }

    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        log("서버 시작 port: " + port);

        while (true) {
            Socket socket = serverSocket.accept();
            es.submit(new HttpRequestHandlerV2(socket));
        }
    }
}
```

- `ExecutorService`: 스레드 풀을 사용한다. 여기서는 `newFixedThreadPool(10)`을 사용해서 최대 동시에 10개의 스레드를 사용할 수 있도록 했다. 결과적으로 10개의 요청을 동시에 처리할 수 있다.
- 참고로 실무에서는 상황에 따라 다르지만 보통 수백 개 정도의 스레드를 사용한다.
- `es.submit(new HttpRequestHandlerV2(socket))`: 스레드 풀에 `HttpRequestHandlerV2` 작업을 요청한다.
- 스레드 풀에 있는 스레드가 `HttpRequestHandlerV2`의 `run()`을 수행한다.

참고 - ExecutorService

- 스레드를 생성하고 관리해준다.
- **김영한의 실전 자바 고급 1편 - 멀티스레드와 동시성** 강의를 참고하자.

```
package was.v2;

import java.io.IOException;

public class ServerMainV2 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        HttpServerV2 server = new HttpServerV2(PORT);
        server.start();
```

```
    }
}
```

이번에는 각 클라이언트의 요청을 별도의 스레드에서 처리한다. 따라서 각 클라이언트의 요청을 동시에 처리할 수 있다. 다른 웹 브라우저 2개를 동시에 열어서 사이트를 실행해보자. (예를 들어서 크롬, 엣지, 사파리 등 다른 브라우저를 열어야 확실한 테스트가 가능하다)

실행 결과

```
14:43:17.517 [     main] 서버 시작 port: 12345
14:43:20.980 [pool-1-thread-1] HTTP 요청 정보 출력
GET / HTTP/1.1
Host: localhost:12345

14:43:20.980 [pool-1-thread-1] HTTP 응답 생성중...
14:43:22.195 [pool-1-thread-3] HTTP 요청 정보 출력
GET / HTTP/1.1
Host: localhost:12345

14:43:22.196 [pool-1-thread-3] HTTP 응답 생성중...
14:43:25.984 [pool-1-thread-1] HTTP 응답 정보 출력
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 20

<h1>Hello World</h1>
14:43:25.988 [pool-1-thread-1] HTTP 응답 전달 완료
14:43:26.111 [pool-1-thread-2] favicon 요청
14:43:26.112 [pool-1-thread-4] favicon 요청
14:43:27.201 [pool-1-thread-3] HTTP 응답 정보 출력
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 20

<h1>Hello World</h1>
14:43:27.202 [pool-1-thread-3] HTTP 응답 전달 완료
```

- 실행 결과에서 대부분의 헤더들은 제외했다.
- 실행 결과를 보면 `pool-1-thread-1`과 `pool-1-thread-3`이 동시에 실행된 것을 확인할 수 있다.

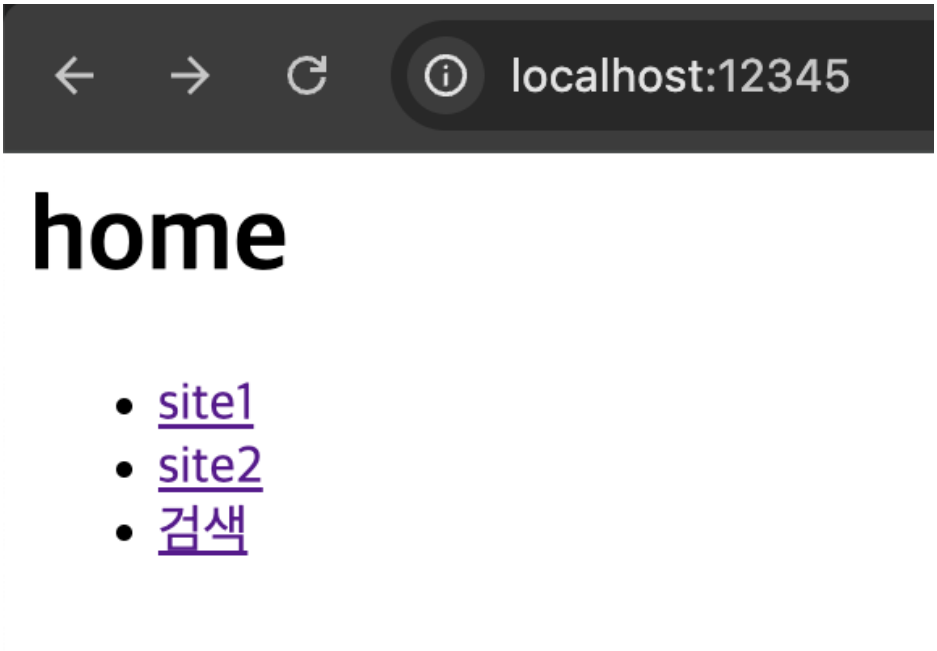
첫 번째 요청과 두 번째 요청이 동시에 처리되는 것을 확인할 수 있다.

HTTP 서버가 어떻게 작동하는지 이해했다면, 이제 기능을 추가해보자.

HTTP 서버3 - 기능 추가

HTTP 서버들은 URL 경로를 사용해서 각각의 기능을 제공한다.

이제 URL에 따른 다양한 기능을 제공해보자.



- home: `/` 첫 화면
- site1: `/site1` 페이지 화면1
- site2: `/site2` 페이지 화면2
- search: `/search` 기능 검색 화면, 클라이언트에서 서버로 검색어를 전달할 수 있다.
- notFound: 잘못된 URL을 호출했을 때 전달하는 화면

```
package was.v3;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.URLDecoder;
```

```
import static java.nio.charset.StandardCharsets.UTF_8;
import static util.MyLogger.log;

public class HttpRequestHandlerV3 implements Runnable {
    private final Socket socket;

    public HttpRequestHandlerV3(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            process(socket);
        } catch (Exception e) {
            log(e);
        }
    }

    private void process(Socket socket) throws IOException {
        try (socket;
             BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream(), UTF_8));
             PrintWriter writer = new PrintWriter(socket.getOutputStream(),
false, UTF_8)) {

            String requestString = requestToString(reader);
            if (requestString.contains("/favicon.ico")) {
                log("favicon 요청");
                return;
            }
            log("HTTP 요청 정보 출력");
            System.out.println(requestString);

            log("HTTP 응답 생성중...");
            if (requestString.startsWith("GET /site1")) {
                site1(writer);
            } else if (requestString.startsWith("GET /site2")) {
                site2(writer);
            } else if (requestString.startsWith("GET /search")) {
                search(writer, requestString);
            } else if (requestString.startsWith("GET / ")){ // '/' 다음에 space
```

필수!

```
                home(writer);
            } else {
                notFound(writer);
            }
            log("HTTP 응답 전달 완료");
        }
    }

    private String requestToString(BufferedReader reader) throws IOException {
        StringBuilder sb = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null) {
            if (line.isEmpty()) {
                break;
            }
            sb.append(line).append("\n");
        }
        return sb.toString();
    }

    private static void home(PrintWriter writer) {
        // 원칙적으로 Content-Length를 계산해서 전달해야 하지만, 예제를 단순하게 설명하기 위해 생략하겠다.
        writer.println("HTTP/1.1 200 OK");
        writer.println("Content-Type: text/html; charset=UTF-8");
        writer.println();
        writer.println("<h1>home</h1>");
        writer.println("<ul>");
        writer.println("<li><a href='/site1'>site1</a></li>");
        writer.println("<li><a href='/site2'>site2</a></li>");
        writer.println("<li><a href='/search?q=hello'>검색</a></li>");
        writer.println("</ul>");
        writer.flush();
    }

    private static void site1(PrintWriter writer) {
        writer.println("HTTP/1.1 200 OK");
        writer.println("Content-Type: text/html; charset=UTF-8");
        writer.println();
        writer.println("<h1>site1</h1>");
        writer.flush();
    }
```

```
    private static void site2(PrintWriter writer) {
        writer.println("HTTP/1.1 200 OK");
        writer.println("Content-Type: text/html; charset=UTF-8");
        writer.println();
        writer.println("<h1>site2</h1>");
        writer.flush();
    }

    private static void notFound(PrintWriter writer) {
        writer.println("HTTP/1.1 404 Not Found");
        writer.println("Content-Type: text/html; charset=UTF-8");
        writer.println();
        writer.println("<h1>404 페이지를 찾을 수 없습니다.</h1>");
        writer.flush();
    }

    private static void search(PrintWriter writer, String requestString) {
        int startIndex = requestString.indexOf("q=");
        int endIndex = requestString.indexOf(" ", startIndex+2);
        String query = requestString.substring(startIndex+2, endIndex);
        String decode = URLDecoder.decode(query, UTF_8);

        writer.println("HTTP/1.1 200 OK");
        writer.println("Content-Type: text/html; charset=UTF-8");
        writer.println();
        writer.println("<h1>Search</h1>");
        writer.println("<ul>");
        writer.println("<li>query: " + query + "</li>");
        writer.println("<li>decode: " + decode + "</li>");
        writer.println("</ul>");
        writer.flush();
    }

}
```

- HTTP 요청 메시지의 시작 라인을 파싱하고, 요청 URL에 맞추어 응답을 전달한다.
  - `GET /` ➜ `home()` 호출
  - `GET /site1` ➜ `site1()` 호출
- 응답시 원칙적으로 헤더에 메시지 바디의 크기를 계산해서 `Content-Length`를 전달해야 하지만, 예제를 단순화하기 위해 생략했다.

검색

검색시 다음과 같은 형식으로 요청이 온다.

- `GET /search?q=hello`
- URL에서 `?` 이후의 부분에 `key1=value1&key2=value2` 포멧으로 서버에 데이터를 전달할 수 있다.
- 이 부분을 파싱하면 요청하는 검색어를 알 수 있다.
- 예제에서는 실제 검색을 하는 것은 아니고, 요청하는 검색어를 간단히 출력한다.
- `URLDecoder`는 바로 뒤에서 설명한다.

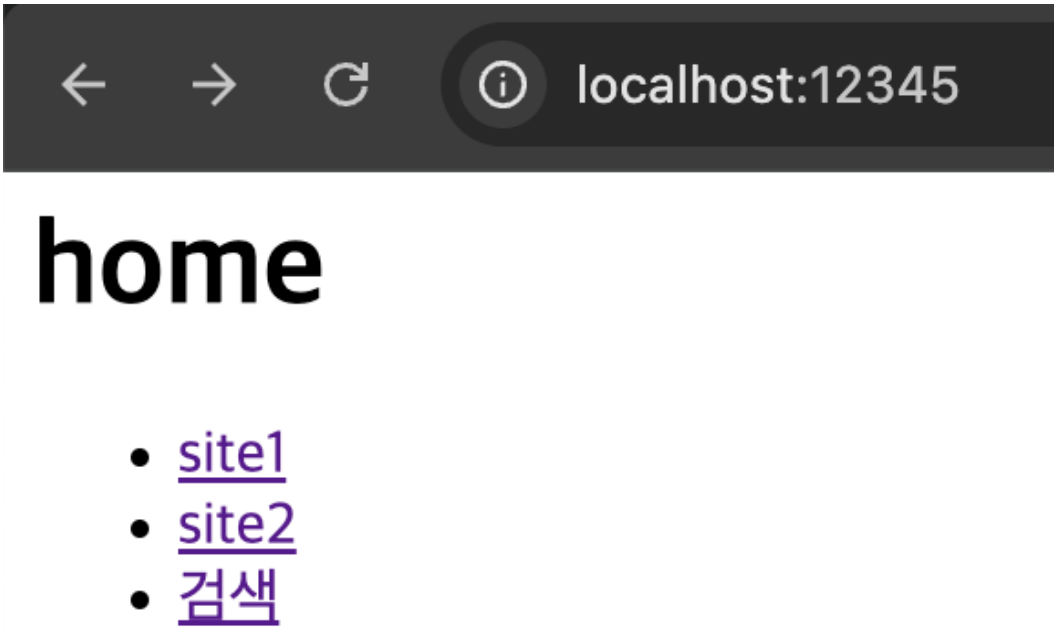
```
package was.v3;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import static util.MyLogger.log;

public class HttpServerV3 {

    private final ExecutorService es = Executors.newFixedThreadPool(10);
    private final int port;

    public HttpServerV3(int port) {
        this.port = port;
    }

    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        log("서버 시작 port: " + port);

        while (true) {
            Socket socket = serverSocket.accept();
            es.submit(new HttpRequestHandlerV3(socket));
        }
    }
}
```

- 기존 코드와 같다.

```
package was.v3;

import java.io.IOException;

public class ServerMainV3 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        HttpServerV3 server = new HttpServerV3(PORT);
        server.start();
    }
}
```

- 기존 코드와 같다.

실행 결과




```
14:56:08.365 [     main] 서버 시작 port: 12345
14:56:09.978 [pool-1-thread-1] HTTP 요청 정보 출력
GET / HTTP/1.1
Host: localhost:12345
```

```
14:56:09.979 [pool-1-thread-1] HTTP 응답 생성중...
14:56:09.981 [pool-1-thread-1] HTTP 응답 전달 완료
14:56:12.837 [pool-1-thread-4] HTTP 요청 정보 출력
GET /site1 HTTP/1.1
Host: localhost:12345

14:56:12.837 [pool-1-thread-4] HTTP 응답 생성중...
14:56:12.837 [pool-1-thread-4] HTTP 응답 전달 완료
14:56:13.685 [pool-1-thread-8] HTTP 요청 정보 출력
GET /site2 HTTP/1.1
Host: localhost:12345

14:56:13.687 [pool-1-thread-8] HTTP 응답 생성중...
14:56:13.687 [pool-1-thread-8] HTTP 응답 전달 완료
14:56:14.425 [pool-1-thread-2] HTTP 요청 정보 출력
GET /search?q=hello HTTP/1.1
Host: localhost:12345

14:56:14.425 [pool-1-thread-2] HTTP 응답 생성중...
14:56:14.432 [pool-1-thread-2] HTTP 응답 전달 완료
```

- 주요 로그만 남겼다.

URL 인코딩

URL이 ASCII를 사용하는 이유

HTTP 메시지에서 시작 라인(URL을 포함)과 HTTP 헤더의 이름은 항상 ASCII를 사용해야 한다.

HTTP 메시지 바디는 UTF-8과 같은 다른 인코딩을 사용할 수 있다.

지금처럼 UTF-8이 표준화된 시대에 왜 URL은 ASCII만 사용할 수 있을까?

HTTP에서 URL이 ASCII 문자를 사용하는 이유

- 인터넷이 처음 설계되던 시기(1980~1990년대)에, 대부분의 컴퓨터 시스템은 ASCII 문자 집합을 사용했다.
- 전 세계에서 사용하는 다양한 컴퓨터 시스템과 네트워크 장비 간의 호환성을 보장하기 위해, URL은 단일한 문자 인코딩 체계를 사용해야 했다. 그 당시 모든 시스템이 비-ASCII 문자를 처리할 수 없었기 때문에, ASCII는 가장 보편적이고 일관된 선택이었다.
- HTTP URL이 ASCII만을 지원하는 이유는 초기 인터넷의 기술적 제약과 전 세계적인 호환성을 유지하기 위한 선

택이다.

- 순수한 UTF-8로 URL을 표현하려면, 전 세계 모든 네트워크 장비, 서버, 클라이언트 소프트웨어가 이를 지원해야 한다. 그러나, 여전히 많은 시스템에서 ASCII 기반 표준에 의존하고 있기 때문에 순수한 UTF-8 URL을 사용하면 호환성 문제가 발생할 수 있다.
- HTTP 스펙은 매우 보수적이고, 호환성을 가장 우선시 한다.

그렇다면 검색어로 사용하는 `/search?q=hello`를 사용할 때 `q=가나다`과 같이 URL에 한글을 전달하려면 어떻게 해야할까?
우선 웹 브라우저 URL 입력 창에 다음 내용을 입력해보자.

`http://localhost:12345/search?q=가나다`

실행 결과 - 웹 브라우저 화면

```
Search
    • query: %EA%B0%80%EB%82%98%EB%8B%A4
    • decode: 가나다
```

퍼센트(%) 인코딩

한글을 UTF-8 인코딩으로 표현하면 한 글자에 3byte의 데이터를 사용한다.
가, 나, 다를 UTF-8 인코딩의 16진수로 표현하면 다음과 같다.
참고: 2진수는 (0, 1), 10진수는 (0 ~ 9), 16진수는(0~9, A, B, C, D, E, F 총 16개로 표현)

- 가: EA,B0,80 (3byte)
- 나: EB,82,98 (3byte)
- 다: EB,8B,A4 (3byte)

URL은 ASCII 문자만 표현할 수 있으므로, UTF-8 문자를 표현할 수 없다.
그래서 한글 "가"를 예를 들면 "가"를 UTF-8 16진수로 표현한 각각의 바이트 문자 앞에 %(퍼센트)를 붙이는 것이다.

- `q=가`
- `q=%EA%B0%80`

이렇게 하면 약간 억지스럽기는 하지만 ASCII 문자를 사용해서 16진수로 표현된 UTF-8을 표현할 수 있다. 그리고 `%EA%B0%80`는 모두 ASCII에 포함되는 문자이다.
이렇게 각각의 16 진수 byte를 문자로 표현하고, 해당 문자 앞에 `%`를 붙이는 것을 퍼센트(`%`) 인코딩이라 한다.

`%` 인코딩 후에 클라이언트에서 서버로 데이터를 전달하면 서버는 각각의 `%`를 제거하고, `EA`, `B0`, `80`이라는 각 문자

를 얻는다. 그리고 이렇게 얻은 문자를 16진수 byte로 변경한다. 이 3개의 byte를 모아서 UTF-8로 디코딩 하면 "가"라는 글자를 얻을 수 있다.

% 인코딩, 디코딩 진행 과정

1. 클라이언트: `가` 전송 희망
2. 클라이언트 % 인코딩: `%EA%B0%80`
    1. "가"를 UTF-8로 인코딩
    2. `EA`, `B0`, `80` 3byte 획득
    3. 각 byte를 16진수 문자로 표현하고 각각의 앞에 `%`를 붙임
3. 클라이언트 ➜ 서버 전송: `q=%EA%B0%80`
4. 서버: `%EA%B0%80` ASCII 문자를 전달 받음
    1. `%`가 붙은 경우 디코딩해야 하는 문자로 인식
    2. `EA`, `B0`, `80`을 byte로 변환, 3byte를 획득
    3. `EA`, `B0`, `80` (3byte)를 UTF-8로 디코딩 ➜ 문자 "가" 획득

% 인코딩

자바가 제공하는 `URLEncoder.encode()`, `URLDecoder.decode`를 사용하면 `%` 인코딩, 디코딩을 처리할 수 있다.

```
package was.v3;

import java.net.URLDecoder;
import java.net.URLEncoder;

import static java.nio.charset.StandardCharsets.*;

public class PercentEncodingMain {

    public static void main(String[] args) {
        String encode = URLEncoder.encode("가", UTF_8);
        System.out.println("encode = " + encode);

        String decode = URLDecoder.decode(encode, UTF_8);
        System.out.println("decode = " + decode);
    }
}
```

```
encode = %EA%B0%80
decode = 가
```

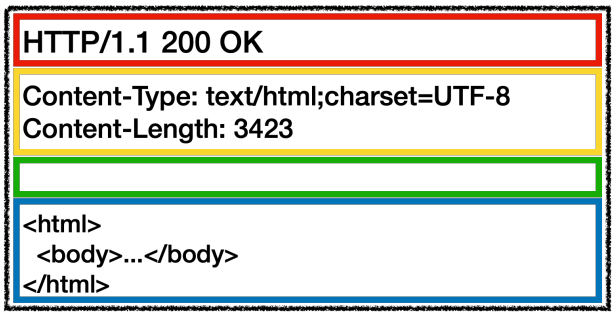
% 인코딩 정리

- `%` 인코딩은 데이터 크기에서 보면 효율이 떨어진다. 문자 "가"는 단지 3byte만 필요하다. 그런데 % 인코딩을 사용하면 `%EA%B0%80` 무려 9byte가 사용된다.
- HTTP는 매우 보수적이다. 호환성을 최우선에 둔다.

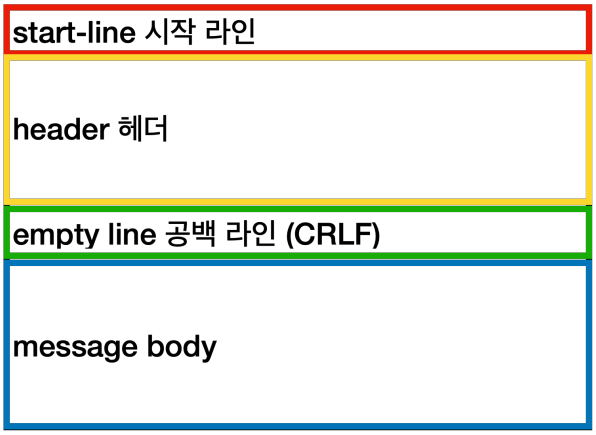
HTTP 서버4 - 요청, 응답

예) HTTP 요청 메시지

요청 메시지도 body 본문을 가질 수 있음



예) HTTP 응답 메시지



HTTP 메시지 구조

HTTP 요청 메시지와 응답 메시지는 각각 정해진 규칙이 있다.

- GET, POST 같은 메서드
- URL
- 헤더
- HTTP 버전, Content-Type, Content-Length

HTTP 요청 메시지와 응답 메시지는 규칙이 있으므로, 각 규칙에 맞추어 객체로 만들면, 단순히 `String` 문자로 다루

는 것 보다 훨씬 더 구조적이고 객체지향적인 편리한 코드를 만들 수 있다.

HTTP 요청 메시지와 응답 메시지를 객체로 만들고, 이전에 작성한 코드도 리펙토링 해보자.

HTTP 요청 메시지

```
package was.httpserver;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.URLDecoder;
import java.util.HashMap;
import java.util.Map;

import static java.nio.charset.StandardCharsets.*;
import static util.MyLogger.log;

public class HttpRequest {

    private String method;
    private String path;
    private final Map<String, String> queryParameters = new HashMap<>();
    private final Map<String, String> headers = new HashMap<>();

    public HttpRequest(BufferedReader reader) throws IOException {
        parseRequestLine(reader);
        parseHeaders(reader);
        // 메시지 바디는 이후에 처리
    }

    private void parseRequestLine(BufferedReader reader) throws IOException {
        String requestLine = reader.readLine();
        if (requestLine == null) {
            throw new IOException("EOF: No request line received");
        }

        String[] parts = requestLine.split(" ");
        if (parts.length != 3) {
            throw new IOException("Invalid request line: " + requestLine);
        }

        method = parts[0];
```

```
        String[] pathParts = parts[1].split("\\?");
        path = pathParts[0];

        if (pathParts.length > 1) {
            parseQueryParameters(pathParts[1]);
        }
    }

    private void parseQueryParameters(String queryString) {
        for (String param : queryString.split("&")) {
            String[] keyValue = param.split("=");
            String key = URLDecoder.decode(keyValue[0], UTF_8);
            String value = keyValue.length > 1 ?
URLDecoder.decode(keyValue[1], UTF_8) : "";
            queryParameters.put(key, value);
        }
    }

    private void parseHeaders(BufferedReader reader) throws IOException {
        String line;
        while (!(line = reader.readLine()).isEmpty()) {
            String[] headerParts = line.split(":");
            // trim() 앞 뒤에 공백 제거
            headers.put(headerParts[0].trim(), headerParts[1].trim());
        }
    }

    public String getMethod() {
        return method;
    }

    public String getPath() {
        return path;
    }

    public String getParameter(String name) {
        return queryParameters.get(name);
    }

    public String getHeader(String name) {
        return headers.get(name);
    }
```

```
    @Override
    public String toString() {
        return "HttpRequest{" +
                "method='" + method + '\'' +
                ", path='" + path + '\'' +
                ", queryParameters=" + queryParameters +
                ", headers=" + headers +
                '}';
    }

}
```

- 패키지 위치에 주의하자. `was.httpserver`는 앞으로 여러 곳에서 사용할 예정이다.
- `reader.readLine()` : 클라이언트가 연결만 하고 데이터 전송 없이 연결을 종료하는 경우 `null`이 반환된다. 이 경우 간단히 `throw new IOException("EOF")` 예외를 던지겠다.
  - 일부 브라우저의 경우 성능 최적화를 위해 TCP 연결을 추가로 하나 더 하는 경우가 있다.
  - 이때 추가 연결을 사용하지 않고, 그대로 종료하면, TCP 연결은 하지만 데이터는 전송하지 않고, 연결을 끊게 된다. (크게 중요한 내용은 아니니 참고만 하자)

HTTP 요청 메시지는 다음과 같다.

```
GET /search?q=hello HTTP/1.1
Host: localhost:12345
```

시작 라인을 통해 `method`, `path`, `queryParameters`를 구할 수 있다.

- method: `GET`
- path: `/search`
- queryParameters: `[q=hello]`

`query`, `header`의 경우 `key=value` 형식이기 때문에 `Map`을 사용하면 이후에 편리하게 데이터를 조회할 수 있다.

만약 다음과 같은 내용이 있다면 `queryParameters`의 `Map`에 저장되는 내용은 다음과 같다.

- `/search?q=hello&type=text`
- `queryParameters` : `[q=hello, type=text]`

퍼센트 디코딩도 `URLDecoder.decode()`를 사용해서 처리한 다음에 `Map`에 보관한다. 따라서 `HttpRequest` 객

체를 사용하는 쪽에서는 퍼센트 디코딩을 고민하지 않아도 된다.

- `/search?q=%EA%B0%80`
- `queryParameters: [q=가]`

HTTP 명세에서 헤더가 끝나는 부분은 빈 라인으로 구분한다.

- `while (!(line = reader.readLine()).isEmpty())`

이렇게 하면 시작 라인의 다양한 정보와 헤더를 객체로 구조화할 수 있다.
참고로 메시지 바디 부분은 아직 파싱하지 않았는데, 뒤에서 설명한다.

HTTP 응답 메시지

```
package was.httpserver;

import java.io.PrintWriter;

import static java.nio.charset.StandardCharsets.*;

public class HttpResponse {
    private final PrintWriter writer;
    private int statusCode = 200;
    private final StringBuilder bodyBuilder = new StringBuilder();
    private String contentType = "text/html; charset=UTF-8";

    public HttpResponse(PrintWriter writer) {
        this.writer = writer;
    }

    public void setStatusCode(int statusCode) {
        this.statusCode = statusCode;
    }

    public void setContentType(String contentType) {
        this.contentType = contentType;
    }

    public void writeBody(String body) {
        bodyBuilder.append(body);
    }
```

```
    public void flush() {
        int contentLength = bodyBuilder.toString().getBytes(UTF_8).length;
        writer.println("HTTP/1.1 " + statusCode + " " +
getReasonPhrase(statusCode));
        writer.println("Content-Type: " + contentType);
        writer.println("Content-Length: " + contentLength);
        writer.println();
        writer.println(bodyBuilder);
        writer.flush();
    }

    private String getReasonPhrase(int statusCode) {
        switch (statusCode) {
            case 200:
                return "OK";
            case 404:
                return "Not Found";
            case 500:
                return "Internal Server Error";
            default:
                return "Unknown Status";
        }
    }

}
```

- 패키지 위치에 주의하자. `was.httpserver`는 앞으로 여러 곳에서 사용할 예정이다.

HTTP 응답 메시지는 다음과 같다.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 20

<h1>Hello World</h1>
```

시작 라인

- HTTP 버전: `HTTP/1.1`
- 응답 코드: `200`
- 응답 코드의 간단한 설명: `OK`

응답 헤더

- `Content-Type`: HTTP 메시지 바디에 들어있는 내용의 종류
- `Content-Length`: HTTP 메시지 바디의 길이

HTTP 응답을 객체로 만들면 시작 라인, 응답 헤더를 구성하는 내용을 반복하지 않고 편리하게 사용할 수 있다.

```
package was.v4;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

import static java.nio.charset.StandardCharsets.UTF_8;
import static util.MyLogger.log;

public class HttpRequestHandlerV4 implements Runnable {
    private final Socket socket;

    public HttpRequestHandlerV4(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        try {
            process(socket);
        } catch (Exception e) {
            log(e);
        }
    }

    private void process(Socket socket) throws IOException {
        try (socket;
             BufferedReader reader = new BufferedReader(new
```

```
InputStreamReader(socket.getInputStream(), UTF_8));
             PrintWriter writer = new PrintWriter(socket.getOutputStream(),
false, UTF_8)) {

            HttpRequest request = new HttpRequest(reader);
            HttpResponse response = new HttpResponse(writer);

            if (request.getPath().equals("/favicon.ico")) {
                log("favicon 요청");
                return;
            }

            log("HTTP 요청 정보 출력");
            System.out.println(request);

            if (request.getPath().equals("/site1")) {
                site1(response);
            } else if (request.getPath().equals("/site2")) {
                site2(response);
            } else if (request.getPath().equals("/search")) {
                search(request, response);
            } else if (request.getPath().equals("/")){
                home(response);
            } else {
                notFound(response);
            }
            response.flush();
            log("HTTP 응답 전달 완료");
        }
    }

    private static void home(HttpResponse response) {
        response.writeBody("<h1>home</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li><a href='/site1'>site1</a></li>");
        response.writeBody("<li><a href='/site2'>site2</a></li>");
        response.writeBody("<li><a href='/search?q=hello'>검색</a></li>");
        response.writeBody("</ul>");
    }

    private static void site1(HttpResponse response) {
        response.writeBody("<h1>site1</h1>");
    }
```

```
    private static void site2(HttpResponse response) {
        response.writeBody("<h1>site2</h1>");
    }

    private static void search(HttpRequest request, HttpResponse response) {
        String query = request.getParameter("q");
        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
        response.writeBody("</ul>");
    }

    private static void notFound(HttpResponse response) {
        response.setStatusCode(404);
        response.writeBody("<h1>404 페이지를 찾을 수 없습니다.</h1>");
    }

}
```

- 클라이언트의 요청이 오면 요청 정보를 기반으로 `HttpRequest` 객체를 만들어둔다. 이때 `HttpResponse`도 함께 만든다.
- `HttpRequest`를 통해서 필요한 정보를 편리하게 찾을 수 있다.
- `/search`의 경우 퍼센트 디코딩을 고민하지 않아도 된다. 이미 `HttpRequest`에서 다 처리해두었다.
- 응답의 경우 `HttpResponse`를 사용하고, HTTP 메시지 바디에 출력할 부분만 적어주면 된다. 나머지는 `HttpResponse` 객체가 대신 처리해준다.
- `response.flush()`는 꼭 호출해주어야 한다. 그래야 실제 응답이 클라이언트에 전달된다.

```
package was.v4;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import static util.MyLogger.log;

public class HttpServerV4 {
```

```
    private final ExecutorService es = Executors.newFixedThreadPool(10);
    private final int port;

    public HttpServerV4(int port) {
        this.port = port;
    }

    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        log("서버 시작 port: " + port);

        while (true) {
            Socket socket = serverSocket.accept();
            es.submit(new HttpRequestHandlerV4(socket));
        }
    }
}
```

- 기존 코드와 같다.
- `HttpRequestHandlerV4` 사용에 주의하자

```
package was.v4;

import java.io.IOException;

public class ServerMainV4 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        HttpServerV4 server = new HttpServerV4(PORT);
        server.start();
    }
}
```

- 기존 코드와 같다.
- `HttpServerV4`에 주의하자

실행 결과

- 실행 결과는 기존과 같다.

- `/search`는 디코딩이 완료된 query를 확인할 수 있다.

정리

`HttpRequest`, `HttpResponse` 객체가 HTTP 요청과 응답을 구조화한 덕분에 많은 중복을 제거하고, 또 코드도 매우 효과적으로 리팩토링 할 수 있었다.
지금까지 학습한 내용을 잘 생각해보면, 전체적인 코드가 크게 2가지로 분류되는 것을 확인할 수 있다.

- HTTP 서버와 관련된 부분
  - `HttpServer`, `HttpRequestHandler`, `HttpRequest`, `HttpResponse`
- 서비스 개발을 위한 로직
  - `home()`, `site1()`, `site2()`, `search()`, `notFound()`

만약 웹을 통한 회원 관리 프로그램 같은 서비스를 새로 만들어야 한다면, 기존 코드에서 HTTP 서버와 관련된 부분은 거의 재사용하고, 서비스 개발을 위한 로직만 추가하면 될 것 같다.
그리고 HTTP 서버와 관련된 부분을 정말 잘 만든다면 HTTP 서버와 관련된 부분은 다른 개발자들이 사용할 수 있도록 오픈소스로 만들거나 따로 판매를 해도 될 것이다.

HTTP 서버5 - 커맨드 패턴

HTTP 서버와 관련된 부분을 본격적으로 구조화해보자. 그래서 서비스 개발을 위한 로직과 명확하게 분리해보자.
여기서 핵심은 HTTP 서버와 관련된 부분은 코드 변경 없이 재사용 가능해야 한다는 점이다.
HTTP 서버와 관련된 부분은 `was.httpserver` 패키지에 넣어두자.
현재 `HttpRequest`, `HttpResponse`가 들어가있다.
이후에 `HttpServer`, `HttpRequestHandler`도 잘 정리해서 추가해보자.

커맨드 패턴 도입

앞에서 다음 코드를 보고 아마도 커맨드 패턴을 도입하면 좋을 것이라 생각했을 것이다.

```
if (request.getPath().equals("/site1")) {
    site1(response);
} else if (request.getPath().equals("/site2")) {
    site2(response);
} else if (request.getPath().equals("/search")) {
    search(request, response);
```

```
} else if (request.getPath().equals("/")){
    home(response);
} else {
    notFound(response);
}
```

커맨드 패턴을 사용하면 확장성이라는 장점도 있지만, HTTP 서버와 관련된 부분과 서비스 개발을 위한 로직을 분리하는데도 도움이 된다.

커맨드 패턴을 도입해보자.

```
package was.httpserver;

import java.io.IOException;

public interface HttpServlet {
    void service(HttpRequest request, HttpResponse response) throws
IOException;
}
```

- `HttpServlet`이라는 이름의 인터페이스를 만들었다.
  - HTTP, Server, Applet의 줄임말이다. (HTTP 서버에서 실행되는 작은 자바 프로그램(애플릿))
- 이 인터페이스의 `service()` 메서드가 있는데, 여기에 서비스 개발과 관련된 부분을 구현하면 된다.
- 매개변수로 `HttpRequest`, `HttpResponse`가 전달된다.
  - `HttpRequest`를 통해서 HTTP 요청 정보를 꺼내고, `HttpResponse`를 통해서 필요한 응답을 할 수 있기 때문에 이 정도면 충분하다.

HTTP 서블릿을 구현해서 서비스의 각 기능을 구현해보자.

서비스 서블릿

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;
```

```
public class HomeServlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        response.writeBody("<h1>home</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li><a href='/site1'>site1</a></li>");
        response.writeBody("<li><a href='/site2'>site2</a></li>");
        response.writeBody("<li><a href='/search?q=hello'>검색</a></li>");
        response.writeBody("</ul>");
    }
}
```

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class Site1Servlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        response.writeBody("<h1>site1</h1>");
    }
}
```

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class Site2Servlet implements HttpServlet {
    @Override
```

```
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        response.writeBody("<h1>site2</h1>");
    }
}
```

```
package was.v5.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class SearchServlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        String query = request.getParameter("q");

        response.writeBody("<h1>Search</h1>");
        response.writeBody("<ul>");
        response.writeBody("<li>query: " + query + "</li>");
        response.writeBody("</ul>");
    }
}
```

- `HomeServlet`, `Site1Servlet`, `Site2Servlet`, `SearchServlet`는 현재 프로젝트에서만 사용하는 개별 서비스를 위한 로직이다. 따라서 `was.v5.servlet` 패키지를 사용했다.

공용 서블릿

`NotFoundServlet`, `InternalErrorServlet`, `DiscardServlet`은 여러 프로젝트에서 공용으로 사용하는 서블릿이다. 따라서 `was.httpserver.servlet` 패키지를 사용했다.

```
package was.httpserver.servlet;

import was.httpserver.HttpRequest;
```

```
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

public class NotFoundServlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) {
        response.setStatusCode(404);
        response.writeBody("<h1>404 페이지를 찾을 수 없습니다.</h1>");
    }
}
```

- 페이지를 찾을 수 없을 때 사용하는 서블릿이다.

```
package was.httpserver.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

public class InternalErrorServlet implements HttpServlet {
    @Override
    public void service(HttpRequest request, HttpResponse response) {
        response.setStatusCode(500);
        response.writeBody("<h1>Internal Error</h1>");
    }
}
```

- HTTP에서 500 응답은 서버 내부에 오류가 있다는 뜻이다.
- 이 부분은 새로 추가했다.

```
package was.httpserver.servlet;

import was.httpserver.HttpRequest;
import was.httpserver.HttpResponse;
import was.httpserver.HttpServlet;

import java.io.IOException;

public class DiscardServlet implements HttpServlet {
```

```
    @Override
    public void service(HttpRequest request, HttpResponse response) throws
IOException {
        // empty
    }
}
```

- `/favicon.ico`의 경우 아무일도 하지 않고 요청을 무시하는 `DiscardServlet`을 사용할 예정이다.

```
package was.httpserver;

public class PageNotFoundException extends RuntimeException {

    public PageNotFoundException(String message) {
        super(message);
    }
}
```

- 페이지를 찾지 못했을 때 사용하는 예외이다.
- `was.httpserver` 패키지를 사용했다.

`HttpServlet`을 관리하고 실행하는 `ServletManager` 클래스도 만들자.

```
package was.httpserver;

import was.httpserver.servlet.InternalErrorServlet;
import was.httpserver.servlet.NotFoundServlet;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class ServletManager {
    private final Map<String, HttpServlet> servletMap = new HashMap<>();
    private HttpServlet defaultServlet;
    private HttpServlet notFoundErrorServlet = new NotFoundServlet();
    private HttpServlet internalErrorServlet = new InternalErrorServlet();
```

```
    public ServletManager() {
    }

    public void add(String path, HttpServlet servlet) {
        servletMap.put(path, servlet);
    }

    public void setDefaultServlet(HttpServlet defaultServlet) {
        this.defaultServlet = defaultServlet;
    }

    public void setNotFoundErrorServlet(HttpServlet notFoundErrorServlet) {
        this.notFoundErrorServlet = notFoundErrorServlet;
    }

    public void setInternalErrorServlet(HttpServlet internalErrorServlet) {
        this.internalErrorServlet = internalErrorServlet;
    }

    public void execute(HttpRequest request, HttpResponse response) throws
IOException {
        try {

            HttpServlet servlet = servletMap.getOrDefault(request.getPath(),
defaultServlet);
            if (servlet == null) {
                throw new PageNotFoundException("request url= " +
request.getPath());
            }
            servlet.service(request, response);

        } catch (PageNotFoundException e) {
            e.printStackTrace();
            notFoundErrorServlet.service(request, response);
        } catch (Exception e) {
            e.printStackTrace();
            internalErrorServlet.service(request, response);
        }
    }
}
```

- `ServletManager`는 설정을 쉽게 변경할 수 있도록, 유연하게 설계되어 있다.
- `was.httpserver` 패키지에서 공용으로 사용한다.

servletMap

`["/"= HomeServlet, "/site1"= Site1Servlet ...]`

`key=value` 형식으로 구성되어 있다. URL의 요청 경로가 Key이다.

- **defaultServlet**: `HttpServlet`을 찾지 못할 때 기본으로 실행된다.
- **notFoundErrorServlet**: `PageNotFoundException`이 발생할 때 실행된다.
  - URL 요청 경로를 `servletMap`에서 찾을 수 없고, `defaultServlet`도 없는 경우 `PageNotFoundException`을 던진다.
- **internalErrorServlet**: 처리할 수 없는 예외가 발생하는 경우 실행된다.

```
package was.httpserver;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

import static java.nio.charset.StandardCharsets.*;
import static util.MyLogger.log;

public class HttpRequestHandler implements Runnable {
    private final Socket socket;
    private final ServletManager servletManager;

    public HttpRequestHandler(Socket socket, ServletManager servletManager) {
        this.socket = socket;
        this.servletManager = servletManager;
    }

    @Override
    public void run() {
        try {
            process(socket);
        } catch (Exception e) {
            log(e);
            e.printStackTrace();
        }
```

```
    }

    private void process(Socket socket) throws IOException {
        try (socket;
             BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream(), UTF_8));
             PrintWriter writer = new PrintWriter(socket.getOutputStream(),
false, UTF_8)) {

            HttpRequest request = new HttpRequest(reader);
            HttpResponse response = new HttpResponse(writer);

            log("HTTP 요청: " + request);
            servletManager.execute(request, response);
            response.flush();
            log("HTTP 응답 완료");
        }
    }

}
```

- `was.httpserver` 패키지에서 공용으로 사용한다.
- `HttpRequestHandler`의 역할이 단순해졌다.
- `HttpRequest`, `HttpResponse`를 만들고, `servletManager`에 전달하면 된다.

```
package was.httpserver;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import static util.MyLogger.log;

public class HttpServer {

    private final ExecutorService es = Executors.newFixedThreadPool(10);
    private final int port;
    private final ServletManager servletManager;
```

```
    public HttpServer(int port, ServletManager servletManager) {
        this.port = port;
        this.servletManager = servletManager;
    }

    public void start() throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        log("서버 시작 port: " + port);

        while (true) {
            Socket socket = serverSocket.accept();
            es.submit(new HttpRequestHandler(socket, servletManager));
        }
    }
}
```

- `was.httpserver` 패키지에서 공용으로 사용한다.
- 기존과 거의 같다.

```
package was.v5;

import was.httpserver.HttpServer;
import was.httpserver.ServletManager;
import was.httpserver.servlet.DiscardServlet;
import was.v5.servlet.HomeServlet;
import was.v5.servlet.SearchServlet;
import was.v5.servlet.Site1Servlet;
import was.v5.servlet.Site2Servlet;

import java.io.IOException;

public class ServerMainV5 {

    private static final int PORT = 12345;

    public static void main(String[] args) throws IOException {
        ServletManager servletManager = new ServletManager();
        servletManager.add("/", new HomeServlet());
        servletManager.add("/site1", new Site1Servlet());
        servletManager.add("/site2", new Site2Servlet());
        servletManager.add("/search", new SearchServlet());
        servletManager.add("/favicon.ico", new DiscardServlet());
```

```
        HttpServer server = new HttpServer(PORT, servletManager);
        server.start();
    }
}
```

먼저 필요한 서블릿(`HttpServlet`)들을 서블릿 매니저에 등록하자. 이 부분이 바로 서비스 개발을 위한 로직들이다.
그리고 `HttpServer`를 생성하면서 서블릿 매니저를 전달하면 된다.

`/favicon.ico`의 경우 아무일도 하지 않고 요청을 무시하는 `DiscardServlet`을 사용했다.

실행 결과

- 기존과 실행 결과는 같다.

정리

이제 HTTP 서버와 서비스 개발을 위한 로직이 명확하게 분리되어 있다.

- HTTP 서버와 관련된 부분 - `was.httpserver` 패키지
  - `HttpServer`, `HttpRequestHandler`, `HttpRequest`, `HttpResponse`
  - `HttpServlet`, `HttpServletManager`
  - `was.httpserver.servlet` 패키지
    - `InternalErrorServlet`, `NotFoundServlet`, `DiscardServlet`
- 서비스 개발을 위한 로직 - `v5.servlet` 패키지
  - `HomeServlet`
  - `Site1Servlet`
  - `Site2Servlet`
  - `SearchServlet`

이후에 다른 HTTP 기반의 프로젝트를 시작해야 한다면, HTTP 서버와 관련된 `was.httpserver` 패키지의 코드를 그대로 재사용하면 된다. 그리고 해당 서비스에 필요한 서블릿을 구현하고, 서블릿 매니저에 등록한 다음에 서버를 실행하면 된다.
여기서 중요한 부분은 새로운 HTTP 서비스(프로젝트)를 만들어도 `was.httpserver` 부분의 코드를 그대로 재사용할 수 있고, 또 전혀 변경하지 않아도 된다는 점이다.

웹 애플리케이션 서버의 역사

우리가 만든 `was.httpserver` 패키지를 사용하면 누구나 손쉽게 HTTP 서비스를 개발할 수 있다. 복잡한 네트워크, 멀티스레드, HTTP 메시지 파싱에 대한 부분을 모두 여기서 해결해준다. `was.httpserver` 패키지를 사용하는 개발자들은 단순히 `HttpServlet`의 구현체만 만들면, 필요한 기능을 손쉽게 구현할 수 있다.

`was.httpserver` 패키지의 코드를 다른 사람들이 사용할 수 있게 오픈소스로 공개한다면, 많은 사람들이 HTTP 기반의 프로젝트를 손쉽게 개발할 수 있을 것이다.
HTTP 서버와 관련된 코드를 정말 잘 만들어서 이 부분을 상업용으로 판매할 수 도 있을 것이다.

웹 애플리케이션 서버
실무 개발자가 목표라면, 웹 애플리케이션 서버(Web Application Server), 줄여서 WAS라는 단어를 많이 듣게 될 것이다.
Web Server가 이나라 중간에 Application이 들어가는 이유는, 웹 서버의 역할을 하면서 추가로 애플리케이션, 그러니까 프로그램 코드도 수행할 수 있는 서버라는 뜻이다.
정리하면 웹(HTTP)를 기반으로 작동하는 서버인데, 이 서버를 통해서 프로그램의 코드도 실행할 수 있는 서버라는 뜻이다. 여기서 말하는 프로그램의 코드는 바로 앞서 우리가 작성한 서블릿 구현체들이다.
우리가 작성한 서버는 HTTP 요청을 처리하는데, 이때 프로그램의 코드를 실행해서 HTTP 요청을 처리한다.
이것이 바로 웹 애플리케이션 서버(WAS)이다.
우리가 만든 `was.httpserver` 패키지도 서블릿 구현체들을 실행해서 프로그램의 코드를 작동할 수 있으므로 웹 애플리케이션 서버라 할 수 있다.

HTTP와 웹이 처음 등장하면서, 많은 회사에서 직접 HTTP 서버와 비슷한 기능을 개발하기 시작했다. 그런데 문제는 각각의 서버간에 호환성이 전혀 없는 것이다. 예를 들어서 A 회사의 HTTP 서버를 사용하다가. B 회사의 HTTP 서버로 변경하려면 인터페이스가 다 다르기 때문에 코드를 너무 많이 수정해야 했다.

예를 들면 A사와 B사는 기능 개발을 위한 인터페이스를 다음과 같이 제공한다.
A사의 기능을 위한 인터페이스

```
package acompany.server;

import java.io.IOException;

public interface HttpProcess {
    void process(Request request, Response response);
}
```

B사의 기능을 위한 인터페이스

```
package bcompany.server;

import java.io.IOException;

public interface HttpCall {
    void call(HttpRequest request, HttpResponse response);
}
```

클래스도 다르고 인터페이스도 모두 다르다. 결과적으로 A사의 HTTP 서버를 사용하다가 B사의 HTTP 서버를 사용하려면 코드를 완전히 다 변경해야 한다.

서블릿과 웹 애플리케이션 서버

이런 문제를 해결하기 위해 1990년대 자바 진영에서는 서블릿(Servlet)이라는 표준이 등장하게 된다.

참고로 우리가 앞서 만든 바로 그 서블릿이다.

```
package jakarta.servlet;

import java.io.IOException;

public interface Servlet {

    void service(ServletRequest var1, ServletResponse var2)
                                                    throws ServletException,
IOException;

    ...
}
```

- 서블릿은 `Servlet`, `HttpServlet`, `ServletRequest`, `ServletResponse`를 포함한 많은 표준을 제공한다.
- HTTP 서버를 만드는 회사들은 모두 서블릿을 기반으로 기능을 제공한다.
- 처음에는 `javax.servlet` 패키지를 사용했는데, 이후에 `jakarta.servlet`으로 변경된다.

서블릿을 제공하는 주요 자바 웹 애플리케이션 서버(WAS)는 다음과 같다.

- 오픈소스
  - Apache Tomcat
  - Jetty
  - GlassFish

    - Undertow
- 상용
    - IBM WebSphere
    - Oracle WebLogic

참고: 보통 자바 진영에서 웹 애플리케이션 서버라고 하면 서블릿 기능을 포함하는 서버를 뜻한다. 하지만 서블릿 기능을 포함하지 않아도 프로그램 코드를 수행할 수 있다면 웹 애플리케이션 서버라 할 수 있다.

표준화의 장점
HTTP 서버를 만드는 회사들이 서블릿을 기반으로 기능을 제공한 덕분에, 개발자는 `jakarta.servlet.Servlet` 인터페이스를 구현하면 된다. 그리고 Apache Tomcat 같은 애플리케이션 서버에서 작성한 `Servlet` 구현체를 실행할 수 있다.
그러다가 만약 성능이나 부가 기능이 더 필요해서 상용 WAS로 변경하거나, 또는 다른 오픈소스로 WAS로 변경해도 기능 변경없이 구현한 서블릿들을 그대로 사용할 수 있다.

이것이 바로 표준화의 큰 장점이다. 개발자는 코드의 변경이 거의 없이 다른 애플리케이션 서버를 선택할 수 있고, 애플리케이션 서버를 만드는 입장에서도 사용자를 잃지 않으면서 더 나은 기능을 제공하는 데 집중할 수 있다. 즉, 표준화된 서블릿 스펙 덕분에 애플리케이션 서버를 제공하는 회사들은 각자의 경쟁력을 키우기 위해 성능 최적화나 부가 기능, 관리 도구 등의 차별화 요소에 집중할 수 있고, 개발자들은 서버에 종속되지 않는 코드를 작성할 수 있는 자유를 얻게 된다.
이와 같은 표준화의 이점은 개발 생태계 전반에 걸쳐 효율성과 생산성을 높여준다. 애플리케이션 서버의 선택에 따른 리스크가 줄어들고, 서버 교체나 환경 변화를 쉽게 받아들일 수 있게 되며, 이는 곧 유지 보수 비용 감소와 장기적인 안정성 확보로 이어진다. 특히 대규모 시스템을 운영하는 기업들에게는 이러한 표준화된 기술 스택이 비용 절감과 더불어 운영의 유연성을 크게 높여준다.
결국, 서블릿 표준은 다양한 벤더들이 상호 운용 가능한 환경을 제공할 수 있게 만들어 주며, 이는 개발자와 기업 모두에게 큰 이점을 제공한다.
이런 표준화 덕분에 자바 웹 애플리케이션 생태계는 크게 발전할 수 있었다.

참고: 서블릿과 WAS에 대한 자세한 내용은 스프링 MVC 강의에서 자세히 다룬다.

정리