0. 스프링 MVC 1편 - 백엔드 웹 개발 핵심 기술 v2022-03-07

#인강/4. 스프링 MVC 1/강의#

- **인프런 강의**
- **저자: 김영한**

전체 목차

버전 수정 이력

2022-03-07

- 오타 수정(박훈희님 도움)
- 오타 수정(southbell09님 도움)
- 오타 수정(highjune님 도움)
- 오타 수정(김진욱님 도움)
- 오타 수정(terry9611님 도움)

2021-07-18

- 오타 수정(아카펠라님 도움)
- 오타 수정(sonbbang님 도움)

2021-03-21

- 오타 수정 (PBFT님 도움)

2021-03-21

- `@RequestParam` 애노테이션을 생략한 경우 `required` 적용 설명 보충

2021-03-14

- IntelliJ 무료 버전에서 서버가 정상 실행되지 않는 문제점 해결

2021-03-08

릴리즈

1. 웹 애플리케이션 이해

#인강/4. 스프링 MVC 1/강의#

목차

- 웹 서버, 웹 애플리케이션 서버
- 서블릿
- 동시 요청 - 멀티 쓰레드
- HTML, HTTP API, CSR, SSR
- 자바 백엔드 웹 기술 역사

2. 서블릿

#인강/4. 스프링 MVC 1/강의#

목차

프로젝트 생성

사전 준비물

- Java 11 설치
- IDE: IntelliJ 또는 Eclipse 설치

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: servlet
  - Name: servlet
  - Package name: hello.servlet
  - Packaging: **War (주의!)**
  - Java: 11

- Dependencies: **Spring Web**, **Lombok**

> **주의!**
> Packaging는 Jar가 아니라 War를 선택해주세요. JSP를 실행하기 위해서 필요합니다.

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.4.3'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
  id 'war'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'
```

```
configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}

repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-web'
  compileOnly 'org.projectlombok:lombok'
  annotationProcessor 'org.projectlombok:lombok'
  providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
  useJUnitPlatform()
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`ServletApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

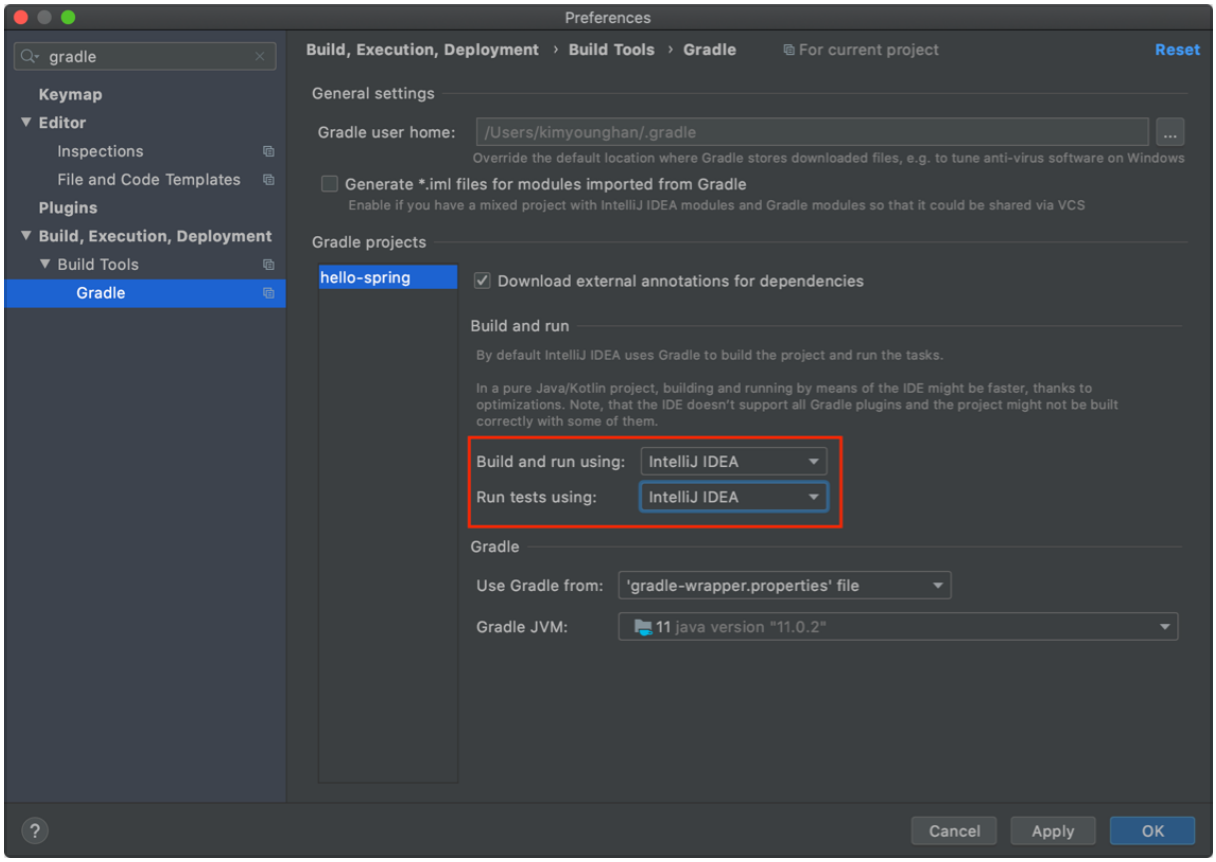
IntelliJ Gradle 대신에 자바 직접 실행

최근 IntelliJ 버전은 Gradle을 통해서 실행 하는 것이 기본 설정이다. 이렇게 하면 실행속도가 느리다. 다음과 같이 변경하면 자바로 바로 실행해서 실행속도가 더 빠르다

- Preferences ➜ Build, Execution, Deployment ➜ Build Tools ➜ Gradle
  - Build and run using: Gradle ➜ IntelliJ IDEA
  - Run tests using: Gradle ➜ IntelliJ IDEA

> **윈도우 사용자**
>
> File ➜ Setting

설정 이미지



> **주의!**
>
> **IntelliJ 무료 버전의 경우 해당 설정을 IntelliJ IDEA가 아니라 Gradle로 설정해야 한다.**
>
> Jar 파일의 경우는 문제가 없는데, War의 경우 톰캣이 정상 시작되지 않는 문제가 발생한다.
>
> 유료 버전은 모두 정상 동작한다.
>
> 또는 `build.gradle` 에 있는 다음 코드를 제거해도 된다.
>
> `providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'`

롬복 적용

1. Preferences ➜ plugin ➜ lombok 검색 실행 (재시작)
2. Preferences ➜ Annotation Processors 검색 ➜ Enable annotation processing 체크 (재시작)
3. 임의의 테스트 클래스를 만들고 @Getter, @Setter 확인

> **윈도우 사용자**
>
> File ➜ Setting

Postman을 설치하자

다음 사이트에서 Postman을 다운로드 받고 설치해두자

- https://www.postman.com/downloads

Hello 서블릿

스프링 부트 환경에서 서블릿 등록하고 사용해보자.

> **참고**
> 서블릿은 톰캣 같은 웹 애플리케이션 서버를 직접 설치하고,그 위에 서블릿 코드를 클래스 파일로 빌드해서 올린 다음, 톰캣 서버를 실행하면 된다. 하지만 이 과정은 매우 번거롭다.
> 스프링 부트는 톰캣 서버를 내장하고 있으므로, 톰캣 서버 설치 없이 편리하게 서블릿 코드를 실행할 수 있다.

스프링 부트 서블릿 환경 구성

`@ServletComponentScan`

스프링 부트는 서블릿을 직접 등록해서 사용할 수 있도록 `@ServletComponentScan` 을 지원한다. 다음과 같이 추가하자.

hello.servlet.ServletApplication

```
package hello.servlet;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;

@ServletComponentScan //서블릿 자동 등록
@SpringBootApplication
public class ServletApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(ServletApplication.class, args);
    }

}
```

서블릿 등록하기

처음으로 실제 동작하는 서블릿 코드를 등록해보자.

hello.servlet.basic.HelloServlet

```
package hello.servlet.basic;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "helloServlet", urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        System.out.println("HelloServlet.service");
        System.out.println("request = " + request);
        System.out.println("response = " + response);

        String username = request.getParameter("username");
        System.out.println("username = " + username);

        response.setContentType("text/plain");
        response.setCharacterEncoding("utf-8");
```

```
        response.getWriter().write("hello " + username);
    }
}
```

- `@WebServlet` 서블릿 애노테이션
  - name: 서블릿 이름
  - urlPatterns: URL 매핑

HTTP 요청을 통해 매핑된 URL이 호출되면 서블릿 컨테이너는 다음 메서드를 실행한다.

`protected void service(HttpServletRequest request, HttpServletResponse response)`

- 웹 브라우저 실행
  - http://localhost:8080/hello?username=world
  - 결과: hello world
- 콘솔 실행결과

```
HelloServlet.service
request = org.apache.catalina.connector.RequestFacade@5e4e72
response = org.apache.catalina.connector.ResponseFacade@37d112b6
username = world
```

> **주의**
> IntelliJ 무료 버전을 사용하는데, 서버가 정상 실행되지 않는다면 **프로젝트 생성 → IntelliJ Gradle 대신에 자바 직접 실행**에 있는 주의 사항을 읽어보자.

HTTP 요청 메시지 로그로 확인하기

다음 설정을 추가하자.

`application.properties`

```
logging.level.org.apache.coyote.http11=debug
```

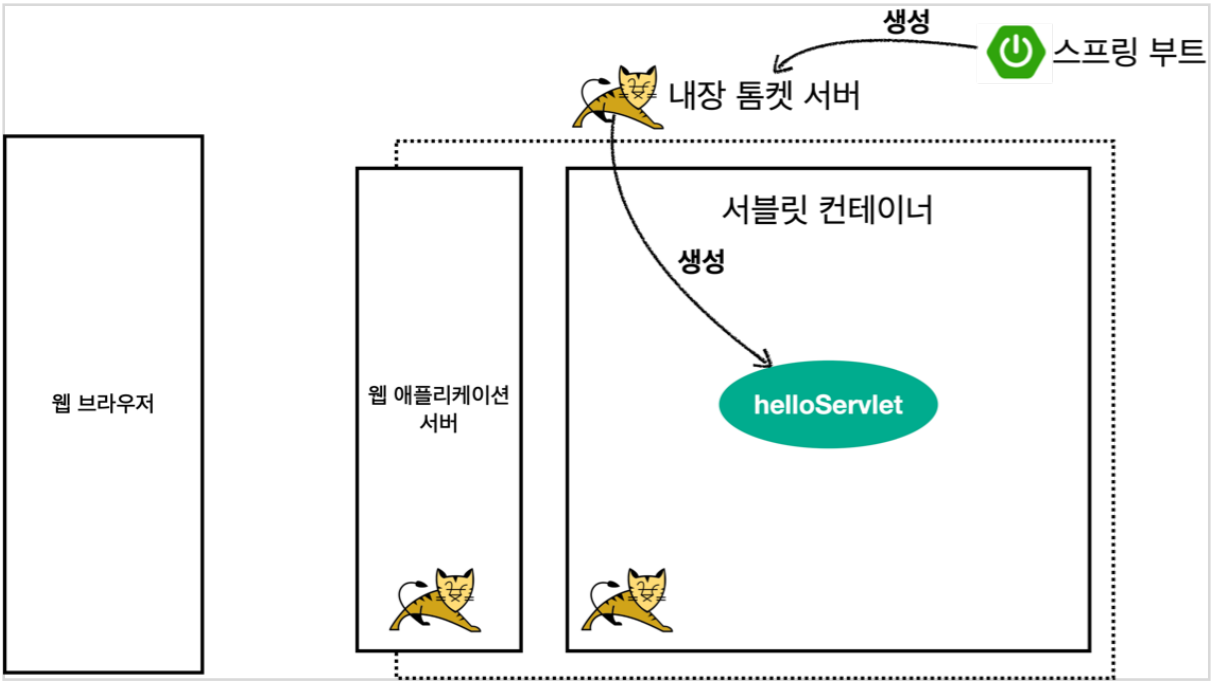
서버를 다시 시작하고, 요청해보면 서버가 받은 HTTP 요청 메시지를 출력하는 것을 확인할 수 있다.

```
...o.a.coyote.http11.Http11InputBuffer: Received [GET /hello?username=servlet
HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
sec-ch-ua: "Chromium";v="88", "Google Chrome";v="88", ";Not A Brand";v="99"
sec-ch-ua-mobile: ?0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 11_2_1) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/88.0.4324.150 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=0.9
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8080/basic.html
Accept-Encoding: gzip, deflate, br
Accept-Language: ko,en-US;q=0.9,en;q=0.8,ko-KR;q=0.7

]
```

> **참고**
> 운영서버에 이렇게 모든 요청 정보를 다 남기면 성능저하가 발생할 수 있다. 개발 단계에서만 적용하자.

서블릿 컨테이너 동작 방식 설명

내장 톰캣 서버 생성



HTTP 요청, HTTP 응답 메시지

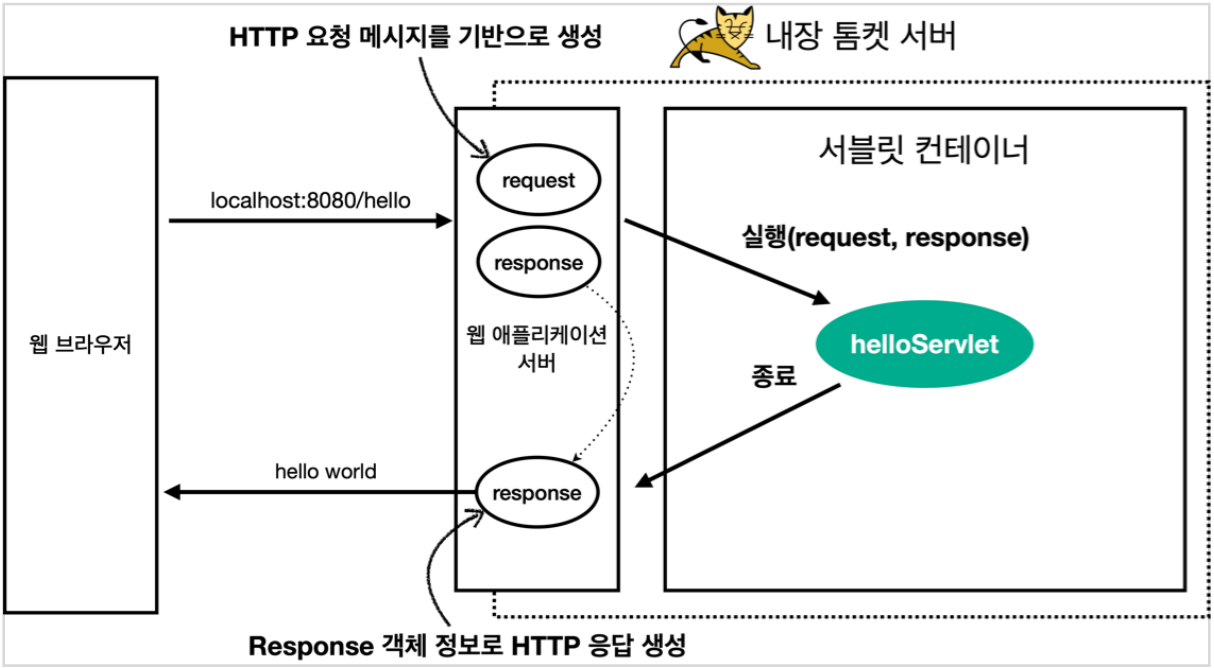
[HTTP 요청]

```
GET /hello?username=world HTTP/1.1
Host: localhost:8080
```

[HTTP 응답]

```
HTTP/1.1 200 OK
Content-Type: text/plain;charset=utf-8
Content-Length: 11

hello world
```

웹 애플리케이션 서버의 요청 응답 구조



> **참고**
> HTTP 응답에서 Content-Length는 웹 애플리케이션 서버가 자동으로 생성해준다.

welcome 페이지 추가

지금부터 개발할 내용을 편리하게 참고할 수 있도록 welcome 페이지를 만들어두자.

`webapp` 경로에 `index.html`을 두면 http://localhost:8080 호출시 `index.html` 페이지가 열린다.

`main/webapp/index.html`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li><a href="basic.html">서블릿 basic</a></li>
</ul>
</body>
</html>
```

이번 장에서 학습할 내용은 다음 `basic.html`이다.

`main/webapp/basic.html`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li>hello 서블릿
```

```
        <ul>
            <li><a href="/hello?username=servlet">hello 서블릿 호출</a></li>
        </ul>
    </li>
    <li>HttpServletRequest
        <ul>
            <li><a href="/request-header">기본 사용법, Header 조회</a></li>
            <li>HTTP 요청 메시지 바디 조회
                <ul>
                    <li><a href="/request-param?username=hello&age=20">GET -
쿼리 파라미터</a></li>
                    <li><a href="/basic/hello-form.html">POST - HTML Form</a></
li>
                    <li>HTTP API - MessageBody -> Postman 테스트</li>
                </ul>
            </li>
        </ul>
    </li>
    <li>HttpServletResponse
        <ul>
            <li><a href="/response-header">기본 사용법, Header 조회</a></li>
            <li>HTTP 응답 메시지 바디 조회
                <ul>
                    <li><a href="/response-html">HTML 응답</a></li>
                    <li><a href="/response-json">HTTP API JSON 응답</a></li>
                </ul>
            </li>
        </ul>
    </li>
</ul>
</body>
</html>
```

HttpServletRequest - 개요

HttpServletRequest 역할

HTTP 요청 메시지를 개발자가 직접 파싱해서 사용해도 되지만, 매우 불편할 것이다. 서블릿은 개발자가 HTTP 요청 메시지를 편리하게 사용할 수 있도록 개발자 대신에 HTTP 요청 메시지를 파싱한다. 그리고 그 결과를 `HttpServletRequest` 객체에 담아서 제공한다.

HttpServletRequest를 사용하면 다음과 같은 HTTP 요청 메시지를 편리하게 조회할 수 있다.

HTTP 요청 메시지

```
POST /save HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded

username=kim&age=20
```

- START LINE
  - HTTP 메소드
  - URL
  - 쿼리 스트링
  - 스키마, 프로토콜
- 헤더
  - 헤더 조회
- 바디
  - form 파라미터 형식 조회
  - message body 데이터 직접 조회

HttpServletRequest 객체는 추가로 여러가지 부가기능도 함께 제공한다.

임시 저장소 기능

- 해당 HTTP 요청이 시작부터 끝날 때 까지 유지되는 임시 저장소 기능
  - 저장: `request.setAttribute(name, value)`
  - 조회: `request.getAttribute(name)`

세션 관리 기능

- `request.getSession(create: true)`

> **중요**

> HttpServletRequest, HttpServletResponse를 사용할 때 가장 중요한 점은 이 객체들이 HTTP 요청 메시지, HTTP 응답 메시지를 편리하게 사용하도록 도와주는 객체라는 점이다. 따라서 이 기능에 대해서 깊이있는 이해를 하려면 **HTTP 스펙이 제공하는 요청, 응답 메시지 자체를 이해**해야 한다.

HttpServletRequest - 기본 사용법

HttpServletRequest가 제공하는 기본 기능들을 알아보자.

hello.servlet.basic.request.RequestHeaderServlet

```
package hello.servlet.basic.request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import java.io.IOException;

//http://localhost:8080/request-header?username=hello
@WebServlet(name = "requestHeaderServlet", urlPatterns = "/request-header")
public class RequestHeaderServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        printStartLine(request);
        printHeaders(request);
        printHeaderUtils(request);
        printEtc(request);

        response.getWriter().write("ok");
    }
}
```

start-line 정보

```
//start line 정보
private void printStartLine(HttpServletRequest request) {
    System.out.println("--- REQUEST-LINE - start ---");

    System.out.println("request.getMethod() = " + request.getMethod()); //GET
    System.out.println("request.getProtocal() = " + request.getProtocol()); //
HTTP/1.1
    System.out.println("request.getScheme() = " + request.getScheme()); //http
    // http://localhost:8080/request-header
    System.out.println("request.getRequestURL() = " + request.getRequestURL());
     // /request-test
    System.out.println("request.getRequestURI() = " + request.getRequestURI());
    //username=hi
    System.out.println("request.getQueryString() = " +
request.getQueryString());
    System.out.println("request.isSecure() = " + request.isSecure()); //https
사용 유무
    System.out.println("--- REQUEST-LINE - end ---");
    System.out.println();
}
```

결과

```
--- REQUEST-LINE - start ---
request.getMethod() = GET
request.getProtocal() = HTTP/1.1
request.getScheme() = http
request.getRequestURL() = http://localhost:8080/request-header
request.getRequestURI() = /request-header
request.getQueryString() = username=hello
request.isSecure() = false
--- REQUEST-LINE - end ---
```

헤더 정보

```
//Header 모든 정보
private void printHeaders(HttpServletRequest request) {
    System.out.println("--- Headers - start ---");

/*
    Enumeration<String> headerNames = request.getHeaderNames();
    while (headerNames.hasMoreElements()) {
        String headerName = headerNames.nextElement();
        System.out.println(headerName + ": " + request.getHeader(headerName));
    }
*/

    request.getHeaderNames().asIterator()
            .forEachRemaining(headerName -> System.out.println(headerName + ":
" + request.getHeader(headerName)));
    System.out.println("--- Headers - end ---");
    System.out.println();
}
```

결과

```
--- Headers - start ---
host: localhost:8080
connection: keep-alive
cache-control: max-age=0
sec-ch-ua: "Chromium";v="88", "Google Chrome";v="88", ";Not A Brand";v="99"
sec-ch-ua-mobile: ?0
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 11_2_0) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/88.0.4324.150 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=0.9
sec-fetch-site: none
sec-fetch-mode: navigate
```

```
sec-fetch-user: ?1
sec-fetch-dest: document
accept-encoding: gzip, deflate, br
accept-language: ko,en-US;q=0.9,en;q=0.8,ko-KR;q=0.7
--- Headers - end ---
```

Header 편리한 조회

```
//Header 편리한 조회
private void printHeaderUtils(HttpServletRequest request) {
    System.out.println("--- Header 편의 조회 start ---");
    System.out.println("[Host 편의 조회]");
    System.out.println("request.getServerName() = " +
request.getServerName()); //Host 헤더
    System.out.println("request.getServerPort() = " +
request.getServerPort()); //Host 헤더
    System.out.println();

    System.out.println("[Accept-Language 편의 조회]");
    request.getLocales().asIterator()
            .forEachRemaining(locale -> System.out.println("locale = " +
locale));
    System.out.println("request.getLocale() = " + request.getLocale());
    System.out.println();

    System.out.println("[cookie 편의 조회]");
    if (request.getCookies() != null) {
        for (Cookie cookie : request.getCookies()) {
            System.out.println(cookie.getName() + ": " + cookie.getValue());
        }
    }
    System.out.println();

    System.out.println("[Content 편의 조회]");
    System.out.println("request.getContentType() = " +
request.getContentType());
    System.out.println("request.getContentLength() = " +
```

```
request.getContentLength());
    System.out.println("request.getCharacterEncoding() = " +
request.getCharacterEncoding());

    System.out.println("--- Header 편의 조회 end ---");
    System.out.println();
}
```

결과

```
--- Header 편의 조회 start ---
[Host 편의 조회]
request.getServerName() = localhost
request.getServerPort() = 8080

[Accept-Language 편의 조회]
locale = ko
locale = en_US
locale = en
locale = ko_KR
request.getLocale() = ko

[cookie 편의 조회]

[Content 편의 조회]
request.getContentType() = null
request.getContentLength() = -1
request.getCharacterEncoding() = UTF-8
--- Header 편의 조회 end ---
```

기타 정보

기타 정보는 HTTP 메시지의 정보는 아니다.

```
//기타 정보
```

```
private void printEtc(HttpServletRequest request) {
    System.out.println("--- 기타 조회 start ---");

    System.out.println("[Remote 정보]");
    System.out.println("request.getRemoteHost() = " +
request.getRemoteHost()); //
    System.out.println("request.getRemoteAddr() = " +
request.getRemoteAddr()); //
    System.out.println("request.getRemotePort() = " +
request.getRemotePort()); //
    System.out.println();

    System.out.println("[Local 정보]");
    System.out.println("request.getLocalName() = " +
request.getLocalName()); //
    System.out.println("request.getLocalAddr() = " +
request.getLocalAddr()); //
    System.out.println("request.getLocalPort() = " +
request.getLocalPort()); //

    System.out.println("--- 기타 조회 end ---");
    System.out.println();
}
```

결과

```
--- 기타 조회 start ---
[Remote 정보]
request.getRemoteHost() = 0:0:0:0:0:0:0:1
request.getRemoteAddr() = 0:0:0:0:0:0:0:1
request.getRemotePort() = 54305

[Local 정보]
request.getLocalName() = localhost
request.getLocalAddr() = 0:0:0:0:0:0:0:1
request.getLocalPort() = 8080
--- 기타 조회 end ---
```

> **참고**
> 로컬에서 테스트하면 IPv6 정보가 나오는데, IPv4 정보를 보고 싶으면 다음 옵션을 VM options에 넣어주면 된다.
> `-Djava.net.preferIPv4Stack=true`

지금까지 HttpServletRequest를 통해서 HTTP 메시지의 start-line, header 정보 조회 방법을 이해했다. 이제 본격적으로 HTTP 요청 데이터를 어떻게 조회하는지 알아보자.

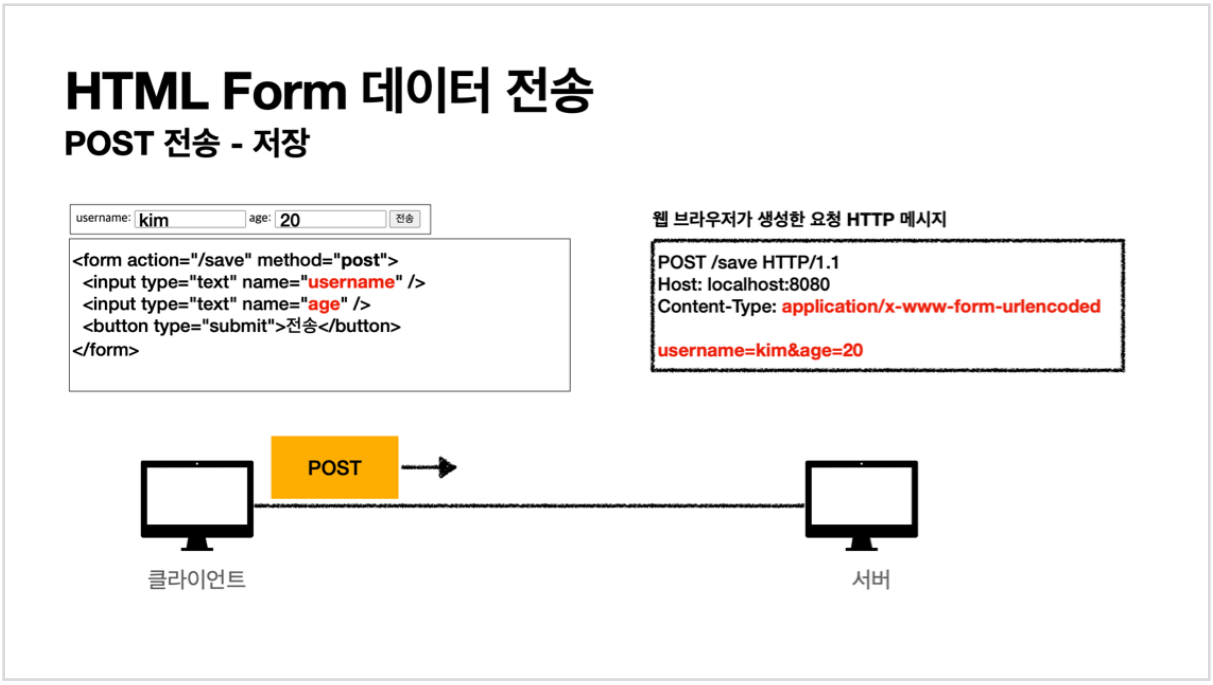
HTTP 요청 데이터 - 개요

HTTP 요청 메시지를 통해 클라이언트에서 서버로 데이터를 전달하는 방법을 알아보자.

주로 다음 3가지 방법을 사용한다.

- **GET - 쿼리 파라미터**
  - /url**?username=hello&age=20**
  - 메시지 바디 없이, URL의 쿼리 파라미터에 데이터를 포함해서 전달
  - 예) 검색, 필터, 페이징등에서 많이 사용하는 방식
- **POST - HTML Form**
  - content-type: application/x-www-form-urlencoded
  - 메시지 바디에 쿼리 파리미터 형식으로 전달 username=hello&age=20
  - 예) 회원 가입, 상품 주문, HTML Form 사용
- **HTTP message body**에 데이터를 직접 담아서 요청
  - HTTP API에서 주로 사용, JSON, XML, TEXT
- 데이터 형식은 주로 JSON 사용
  - POST, PUT, PATCH

POST- HTML Form 예시



하나씩 알아보자.

HTTP 요청 데이터 - GET 쿼리 파라미터

다음 데이터를 클라이언트에서 서버로 전송해보자.

전달 데이터

- username=hello
- age=20

메시지 바디 없이, URL의 **쿼리 파라미터**를 사용해서 데이터를 전달하자.
예) 검색, 필터, 페이징등에서 많이 사용하는 방식

쿼리 파라미터는 URL에 다음과 같이 `?`를 시작으로 보낼 수 있다. 추가 파라미터는 `&`로 구분하면 된다.

- http://localhost:8080/request-param?username=hello&age=20

서버에서는 `HttpServletRequest`가 제공하는 다음 메서드를 통해 쿼리 파라미터를 편리하게 조회할 수 있다.

쿼리 파라미터 조회 메서드

```
String username = request.getParameter("username"); //단일 파라미터 조회
Enumeration<String> parameterNames = request.getParameterNames(); //파라미터 이름들 모두 조회
Map<String, String[]> parameterMap = request.getParameterMap(); //파라미터를 Map 으로 조회
String[] usernames = request.getParameterValues("username"); //복수 파라미터 조회
```

RequestParamServlet

```
package hello.servlet.basic.request;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Enumeration;

/**
 * 1. 파라미터 전송 기능
 * http://localhost:8080/request-param?username=hello&age=20
 * <p>
 * 2. 동일한 파라미터 전송 가능
 * http://localhost:8080/request-param?username=hello&username=kim&age=20
 */
@WebServlet(name = "requestParamServlet", urlPatterns = "/request-param")
public class RequestParamServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse resp) throws ServletException, IOException {

        System.out.println("[전체 파라미터 조회] - start");

        /*
        Enumeration<String> parameterNames = request.getParameterNames();
        while (parameterNames.hasMoreElements()) {
```

```
            String paramName = parameterNames.nextElement();
            System.out.println(paramName + "=" +
request.getParameter(paramName));
        }
        */

        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> System.out.println(paramName +
"=" + request.getParameter(paramName)));
        System.out.println("[전체 파라미터 조회] - end");
        System.out.println();

        System.out.println("[단일 파라미터 조회]");
        String username = request.getParameter("username");
        System.out.println("request.getParameter(username) = " + username);

        String age = request.getParameter("age");
        System.out.println("request.getParameter(age) = " + age);
        System.out.println();

        System.out.println("[이름이 같은 복수 파라미터 조회]");
        System.out.println("request.getParameterValues(username)");
        String[] usernames = request.getParameterValues("username");
        for (String name : usernames) {
            System.out.println("username=" + name);
        }

        resp.getWriter().write("ok");
    }
}
```

실행 - 파라미터 전송

http://localhost:8080/request-param?username=hello&age=20

결과

```
[전체 파라미터 조회] - start
```

```
username=hello
age=20
[전체 파라미터 조회] - end

[단일 파라미터 조회]
request.getParameter(username) = hello
request.getParameter(age) = 20

[이름이 같은 복수 파라미터 조회]
request.getParameterValues(username)
username=hello
```

실행 - 동일 파라미터 전송

http://localhost:8080/request-param?username=hello&username=kim&age=20

결과

```
[전체 파라미터 조회] - start
username=hello
age=20
[전체 파라미터 조회] - end

[단일 파라미터 조회]
request.getParameter(username) = hello
request.getParameter(age) = 20

[이름이 같은 복수 파라미터 조회]
request.getParameterValues(username)
username=hello
username=kim
```

복수 파라미터에서 단일 파라미터 조회

`username=hello&username=kim`과 같이 파라미터 이름은 하나인데, 값이 중복이면 어떻게 될까?
`request.getParameter()`는 하나의 파라미터 이름에 대해서 단 하나의 값만 있을 때 사용해야 한다.
지금처럼 중복일 때는 `request.getParameterValues()`를 사용해야 한다.
참고로 이렇게 중복일 때 `request.getParameter()`를 사용하면 `request.getParameterValues()`의 첫 번째 값을 반환한다.

HTTP 요청 데이터 - POST HTML Form

이번에는 HTML의 Form을 사용해서 클라이언트에서 서버로 데이터를 전송해보자.
주로 회원 가입, 상품 주문 등에서 사용하는 방식이다.

특징

- content-type: `application/x-www-form-urlencoded`
- 메시지 바디에 쿼리 파리미터 형식으로 데이터를 전달한다. `username=hello&age=20`

`src/main/webapp/basic/hello-form.html` 생성

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<form action="/request-param" method="post">
    username: <input type="text" name="username" />
    age:      <input type="text" name="age" />
    <button type="submit">전송</button>
</form>
</body>
</html>
```

실행해보자.

- http://localhost:8080/basic/hello-form.html

> **주의**
> 웹 브라우저가 결과를 캐시하고 있어서, 과거에 작성했던 html 결과가 보이는 경우도 있다. 이때는 웹 브라우저의 새로 고침을 직접 선택해주면 된다. 물론 서버를 재시작 하지 않아서 그럴 수도 있다.

POST의 HTML Form을 전송하면 웹 브라우저는 다음 형식으로 HTTP 메시지를 만든다. (웹 브라우저 개발자 모드 확인)

- **요청 URL**: http://localhost:8080/request-param
- **content-type**: `application/x-www-form-urlencoded`
- **message body**: `username=hello&age=20`

`application/x-www-form-urlencoded` 형식은 앞서 GET에서 살펴본 쿼리 파라미터 형식과 같다.
따라서 **쿼리 파라미터 조회 메서드를 그대로 사용**하면 된다.
클라이언트(웹 브라우저) 입장에서는 두 방식에 차이가 있지만, 서버 입장에서는 둘의 형식이 동일하므로, `request.getParameter()`로 편리하게 구분없이 조회할 수 있다.

정리하면 `request.getParameter()`는 GET URL 쿼리 파라미터 형식도 지원하고, POST HTML Form 형식도 둘 다 지원한다.

> **참고**
> content-type은 HTTP 메시지 바디의 데이터 형식을 지정한다.
> **GET URL 쿼리 파라미터 형식**으로 클라이언트에서 서버로 데이터를 전달할 때는 HTTP 메시지 바디를 사용하지 않기 때문에 content-type이 없다.
> **POST HTML Form 형식**으로 데이터를 전달하면 HTTP 메시지 바디에 해당 데이터를 포함해서 보내기 때문에 바디에 포함된 데이터가 어떤 형식인지 content-type을 꼭 지정해야 한다. 이렇게 폼으로 데이터를 전송하는 형식을 `application/x-www-form-urlencoded`라 한다.

Postman을 사용한 테스트

이런 간단한 테스트에 HTML form을 만들기는 귀찮다. 이때는 Postman을 사용하면 된다.

Postman 테스트 주의사항

- POST 전송시
  - Body ➜ `x-www-form-urlencoded` 선택
  - Headers에서 content-type: `application/x-www-form-urlencoded`로 지정된 부분 꼭 확인

HTTP 요청 데이터 - API 메시지 바디 - 단순 텍스트

- **HTTP message body**에 데이터를 직접 담아서 요청

- HTTP API에서 주로 사용, JSON, XML, TEXT
- 데이터 형식은 주로 JSON 사용
- POST, PUT, PATCH

- 먼저 가장 단순한 텍스트 메시지를 HTTP 메시지 바디에 담아서 전송하고, 읽어보자.
- HTTP 메시지 바디의 데이터를 InputStream을 사용해서 직접 읽을 수 있다.

RequestBodyStringServlet

```
package hello.servlet.basic.request;

import org.springframework.util.StreamUtils;

import javax.servlet.ServletException;
import javax.servlet.ServletInputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

@WebServlet(name = "requestBodyStringServlet", urlPatterns = "/request-body-
string")
public class RequestBodyStringServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        ServletInputStream inputStream = request.getInputStream();
        String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);

        System.out.println("messageBody = " + messageBody);

        response.getWriter().write("ok");
    }
```

```
}
```

Postman을 사용해서 테스트 해보자.

> **참고**
> inputStream은 byte 코드를 반환한다. byte 코드를 우리가 읽을 수 있는 문자(String)로 보려면 문자표(Charset)를 지정해주어야 한다. 여기서는 UTF_8 Charset을 지정해주었다.

문자 전송

- POST http://localhost:8080/request-body-string
- content-type: text/plain
- message body: `hello`
- 결과: `messageBody = hello`

HTTP 요청 데이터 - API 메시지 바디 - JSON

이번에는 HTTP API에서 주로 사용하는 JSON 형식으로 데이터를 전달해보자.

JSON 형식 전송

- POST http://localhost:8080/request-body-json
- content-type: **application/json**
- message body: `{"username": "hello", "age": 20}`
- 결과: `messageBody = {"username": "hello", "age": 20}`

JSON 형식 파싱 추가

JSON 형식으로 파싱할 수 있게 객체를 하나 생성하자

`hello.servlet.basic.HelloData`

```
package hello.servlet.basic;

import lombok.Getter;
import lombok.Setter;
```

```
@Getter @Setter
public class HelloData {

    private String username;
    private int age;
}
```

lombok이 제공하는 `@Getter`, `@Setter` 덕분에 다음 코드가 자동으로 추가된다.(눈에 보이지는 않는다.)

```
package hello.servlet.basic;

public class HelloData {

    private String username;
    private int age;

    //==== lombok 생성 코드 ====//
    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

> **참고**: 만약 잘 동작하지 않는다면 프로젝트 생성에 롬복 부분을 다시 확인하자.

```
package hello.servlet.basic.request;

import com.fasterxml.jackson.databind.ObjectMapper;
import hello.servlet.basic.HelloData;
import org.springframework.util.StreamUtils;

import javax.servlet.ServletException;
import javax.servlet.ServletInputStream;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * http://localhost:8080/request-body-json
 *
 * JSON 형식 전송
 * content-type: application/json
 * message body: {"username": "hello", "age": 20}
 *
 */
@WebServlet(name = "requestBodyJsonServlet", urlPatterns = "/request-body-
json")
public class RequestBodyJsonServlet extends HttpServlet {

    private ObjectMapper objectMapper = new ObjectMapper();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        ServletInputStream inputStream = request.getInputStream();
        String messageBody = StreamUtils.copyToString(inputStream,
```

```
StandardCharsets.UTF_8);

        System.out.println("messageBody = " + messageBody);

        HelloData helloData = objectMapper.readValue(messageBody,
HelloData.class);
        System.out.println("helloData.username = " + helloData.getUsername());
        System.out.println("helloData.age = " + helloData.getAge());

        response.getWriter().write("ok");
    }
}
```

Postman으로 실행해보자.

- POST http://localhost:8080/request-body-json
- content-type: **application/json** (Body ➜ raw, 가장 오른쪽에서 JSON 선택)
- message body: `{"username": "hello", "age": 20}`

출력결과

```
messageBody={"username": "hello", "age": 20}
data.username=hello
data.age=20
```

> **참고**
> JSON 결과를 파싱해서 사용할 수 있는 자바 객체로 변환하려면 Jackson, Gson 같은 JSON 변환 라이브러리를 추가해서 사용해야 한다. 스프링 부트로 Spring MVC를 선택하면 기본으로 Jackson 라이브러리(`ObjectMapper`)를 함께 제공한다.

> **참고**
> HTML form 데이터도 메시지 바디를 통해 전송되므로 직접 읽을 수 있다. 하지만 편리한 파리미터 조회 기능(`request.getParameter(...)`)을 이미 제공하기 때문에 파라미터 조회 기능을 사용하면 된다.

HttpServletResponse - 기본 사용법

HttpServletResponse 역할

HTTP 응답 메시지 생성

- HTTP 응답코드 지정
- 헤더 생성
- 바디 생성

편의 기능 제공

- Content-Type, 쿠키, Redirect

HttpServletResponse - 기본 사용법

hello.servlet.basic.response.ResponseHeaderServlet

```
package hello.servlet.basic.response;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * http://localhost:8080/response-header
 *
 */
@WebServlet(name = "responseHeaderServlet", urlPatterns = "/response-header")
public class ResponseHeaderServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
```

```
response)
            throws ServletException, IOException {

        //[status-line]
        response.setStatus(HttpServletResponse.SC_OK); //200

        //[response-headers]
        response.setHeader("Content-Type", "text/plain;charset=utf-8");
        response.setHeader("Cache-Control", "no-cache, no-store, must-
revalidate");
        response.setHeader("Pragma", "no-cache");
        response.setHeader("my-header","hello");

        //[Header 편의 메서드]
        content(response);
        cookie(response);
        redirect(response);

        //[message body]
        PrintWriter writer = response.getWriter();
        writer.println("ok");
    }
}
```

Content 편의 메서드

```
private void content(HttpServletResponse response) {
    //Content-Type: text/plain;charset=utf-8
    //Content-Length: 2
    //response.setHeader("Content-Type", "text/plain;charset=utf-8");
    response.setContentType("text/plain");
    response.setCharacterEncoding("utf-8");
    //response.setContentLength(2); //(생략시 자동 생성)
}
```

쿠키 편의 메서드

```
private void cookie(HttpServletResponse response) {
    //Set-Cookie: myCookie=good; Max-Age=600;
    //response.setHeader("Set-Cookie", "myCookie=good; Max-Age=600");
    Cookie cookie = new Cookie("myCookie", "good");
    cookie.setMaxAge(600); //600초
    response.addCookie(cookie);
}
```

redirect 편의 메서드

```
private void redirect(HttpServletResponse response) throws IOException {
    //Status Code 302
    //Location: /basic/hello-form.html

    //response.setStatus(HttpServletResponse.SC_FOUND); //302
    //response.setHeader("Location", "/basic/hello-form.html");
    response.sendRedirect("/basic/hello-form.html");
}
```

HTTP 응답 데이터 - 단순 텍스트, HTML

HTTP 응답 메시지는 주로 다음 내용을 담아서 전달한다.

- 단순 텍스트 응답
  - 앞에서 살펴봄 (`writer.println("ok");`)
- HTML 응답
- HTTP API - MessageBody JSON 응답

HttpServletResponse - HTML 응답

hello.servlet.web.response.ResponseHtmlServlet

```
package hello.servlet.basic.response;
```

```java
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(name = "responseHtmlServlet", urlPatterns = "/response-html")
public class ResponseHtmlServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        //Content-Type: text/html;charset=utf-8
        response.setContentType("text/html");
        response.setCharacterEncoding("utf-8");

        PrintWriter writer = response.getWriter();
        writer.println("<html>");
        writer.println("<body>");
        writer.println("  <div>안녕?</div>");
        writer.println("</body>");
        writer.println("</html>");
    }
}
```

HTTP 응답으로 HTML을 반환할 때는 content-type을 `text/html`로 지정해야 한다.

실행

- http://localhost:8080/response-html
- 페이지 소스보기를 사용하면 결과 HTML을 확인할 수 있다.

HTTP 응답 데이터 - API JSON

hello.servlet.web.response. ResponseJsonServlet

```
package hello.servlet.basic.response;

import com.fasterxml.jackson.databind.ObjectMapper;
import hello.servlet.basic.HelloData;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * http://localhost:8080/response-json
 *
 */
@WebServlet(name = "responseJsonServlet", urlPatterns = "/response-json")
public class ResponseJsonServlet extends HttpServlet {

    private ObjectMapper objectMapper = new ObjectMapper();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        //Content-Type: application/json
        response.setHeader("content-type", "application/json");
        response.setCharacterEncoding("utf-8");

        HelloData data = new HelloData();
        data.setUsername("kim");
        data.setAge(20);
```

```
        //{"username":"kim","age":20}
        String result = objectMapper.writeValueAsString(data);

        response.getWriter().write(result);
    }
}
```

HTTP 응답으로 JSON을 반환할 때는 content-type을 `application/json`로 지정해야 한다. Jackson 라이브러리가 제공하는 `objectMapper.writeValueAsString()`를 사용하면 객체를 JSON 문자로 변경할 수 있다.

실행

- http://localhost:8080/response-json

> **참고**
> `application/json`은 스펙상 utf-8 형식을 사용하도록 정의되어 있다. 그래서 스펙에서 charset=utf-8과 같은 추가 파라미터를 지원하지 않는다. 따라서 `application/json` 이라고만 사용해야지 `application/json;charset=utf-8`이라고 전달하는 것은 의미 없는 파라미터를 추가한 것이 된다. response.getWriter()를 사용하면 추가 파라미터를 자동으로 추가해버린다. 이때는 response.getOutputStream()으로 출력하면 그런 문제가 없다.

정리

3. 서블릿, JSP, MVC 패턴

#인강/4. 스프링 MVC 1/강의#

목차

- 3. 서블릿, JSP, MVC 패턴 - MVC 패턴 - 한계
- 3. 서블릿, JSP, MVC 패턴 - 정리

회원 관리 웹 애플리케이션 요구사항

회원 정보

이름: `username`
나이: `age`

기능 요구사항

- 회원 저장
- 회원 목록 조회

회원 도메인 모델

```
package hello.servlet.domain.member;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class Member {

    private Long id;
    private String username;
    private int age;

    public Member() {
    }

    public Member(String username, int age) {
        this.username = username;
        this.age = age;
    }

}
```

- `id`는 `Member`를 회원 저장소에 저장하면 회원 저장소가 할당한다.

회원 저장소

```
package hello.servlet.domain.member;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 동시성 문제가 고려되어 있지 않음, 실무에서는 ConcurrentHashMap, AtomicLong 사용 고려
 */
public class MemberRepository {

    private static Map<Long, Member> store = new HashMap<>(); //static 사용
    private static long sequence = 0L; //static 사용

    private static final MemberRepository instance = new MemberRepository();

    public static MemberRepository getInstance() {
        return instance;
    }

    private MemberRepository() {
    }

    public Member save(Member member) {
        member.setId(++sequence);
        store.put(member.getId(), member);
        return member;
    }

    public Member findById(Long id) {
        return store.get(id);
    }
```

```
    public List<Member> findAll() {
        return new ArrayList<>(store.values());
    }

    public void clearStore() {
        store.clear();
    }
}
```

회원 저장소는 싱글톤 패턴을 적용했다. 스프링을 사용하면 스프링 빈으로 등록하면 되지만, 지금은 최대한 스프링 없이 순수 서블릿 만으로 구현하는 것이 목적이다.
싱글톤 패턴은 객체를 단 하나만 생생해서 공유해야 하므로 생성자를 `private` 접근자로 막아둔다.

회원 저장소 테스트 코드

```
package hello.servlet.domain.member;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;

import java.util.List;

import static org.assertj.core.api.Assertions.*;

class MemberRepositoryTest {

    MemberRepository memberRepository = MemberRepository.getInstance();

    @AfterEach
    void afterEach() {
        memberRepository.clearStore();
    }

    @Test
    void save() {
        //given
```

```
        Member member = new Member("hello", 20);

        //when
        Member savedMember = memberRepository.save(member);

        //then
        Member findMember = memberRepository.findById(savedMember.getId());
        assertThat(findMember).isEqualTo(savedMember);
    }

    @Test
    void findAll() {
        //given
        Member member1 = new Member("member1", 20);
        Member member2 = new Member("member2", 30);

        memberRepository.save(member1);
        memberRepository.save(member2);

        //when
        List<Member> result = memberRepository.findAll();

        //then
        assertThat(result.size()).isEqualTo(2);
        assertThat(result).contains(member1, member2);
    }

}
```

회원을 저장하고, 목록을 조회하는 테스트를 작성했다. 각 테스트가 끝날 때, 다음 테스트에 영향을 주지 않도록 각 테스트의 저장소를 `clearStore()`를 호출해서 초기화했다.

서블릿으로 회원 관리 웹 애플리케이션 만들기

이제 본격적으로 서블릿으로 회원 관리 웹 애플리케이션을 만들어보자.

가장 먼저 서블릿으로 회원 등록 HTML 폼을 제공해보자.

MemberFormServlet - 회원 등록 폼

```
package hello.servlet.web.servlet;

import hello.servlet.domain.member.MemberRepository;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(name = "memberFormServlet", urlPatterns = "/servlet/members/new-
form")
public class MemberFormServlet extends HttpServlet {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        response.setContentType("text/html");
        response.setCharacterEncoding("utf-8");

        PrintWriter w = response.getWriter();
        w.write("<!DOCTYPE html>\n" +
                "<html>\n" +
                "<head>\n" +
                "    <meta charset=\"UTF-8\">\n" +
```

```
                "    <title>Title</title>\n" +
                "</head>\n" +
                "<body>\n" +
                "<form action=\"/servlet/members/save\" method=\"post\">\n" +
                "    username: <input type=\"text\" name=\"username\" />\n" +
                "    age:      <input type=\"text\" name=\"age\" />\n" +
                "    <button type=\"submit\">전송</button>\n" +
                "</form>\n" +
                "</body>\n" +
                "</html>\n");
    }
}
```

`MemberFormServlet`은 단순하게 회원 정보를 입력할 수 있는 HTML Form을 만들어서 응답한다. 자바 코드로 HTML을 제공해야 하므로 쉽지 않은 작업이다.

실행

- http://localhost:8080/servlet/members/new-form
- HTML Form 데이터를 POST로 전송해도, 전달 받는 서블릿을 아직 만들지 않았다. 그래서 오류가 발생하는 것이 정상이다.

이번에는 HTML Form에서 데이터를 입력하고 전송을 누르면 실제 회원 데이터가 저장되도록 해보자. 전송 방식은 POST HTML Form에서 학습한 내용과 같다.

MemberSaveServlet - 회원 저장

```
package hello.servlet.web.servlet;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
import java.io.PrintWriter;

@WebServlet(name = "memberSaveServlet", urlPatterns = "/servlet/members/save")
public class MemberSaveServlet extends HttpServlet {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        System.out.println("MemberSaveServlet.service");
        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));

        Member member = new Member(username, age);
        System.out.println("member = " + member);
        memberRepository.save(member);

        response.setContentType("text/html");
        response.setCharacterEncoding("utf-8");

        PrintWriter w = response.getWriter();
        w.write("<html>\n" +
                "<head>\n" +
                "    <meta charset=\"UTF-8\">\n" +
                "</head>\n" +
                "<body>\n" +
                "성공\n" +
                "<ul>\n" +
                "    <li>id="+member.getId()+"</li>\n" +
                "    <li>username="+member.getUsername()+"</li>\n" +
                "    <li>age="+member.getAge()+"</li>\n" +
                "</ul>\n" +
                "<a href=\"/index.html\">메인</a>\n" +
                "</body>\n" +
                "</html>");
    }
```

```
}
```

`MemberSaveServlet`은 다음 순서로 동작한다.

1. 파라미터를 조회해서 Member 객체를 만든다.
2. Member 객체를 MemberRepository를 통해서 저장한다.
3. Member 객체를 사용해서 결과 화면용 HTML을 동적으로 만들어서 응답한다.

실행

- http://localhost:8080/servlet/members/new-form
- 데이터가 전송되고, 저장 결과를 확인할 수 있다.

이번에는 저장된 모든 회원 목록을 조회하는 기능을 만들어보자.

MemberListServlet - 회원 목록

```
package hello.servlet.web.servlet;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

@WebServlet(name = "memberListServlet", urlPatterns = "/servlet/members")
public class MemberListServlet extends HttpServlet {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
```

```
        response.setContentType("text/html");
        response.setCharacterEncoding("utf-8");

        List<Member> members = memberRepository.findAll();

        PrintWriter w = response.getWriter();
        w.write("<html>");
        w.write("<head>");
        w.write("    <meta charset=\"UTF-8\">");
        w.write("    <title>Title</title>");
        w.write("</head>");
        w.write("<body>");
        w.write("<a href=\"/index.html\">메인</a>");
        w.write("<table>");
        w.write("    <thead>");
        w.write("    <th>id</th>");
        w.write("    <th>username</th>");
        w.write("    <th>age</th>");
        w.write("    </thead>");
        w.write("    <tbody>");

/*
        w.write("    <tr>");
        w.write("        <td>1</td>");
        w.write("        <td>userA</td>");
        w.write("        <td>10</td>");
        w.write("    </tr>");
*/
        for (Member member : members) {
            w.write("    <tr>");
            w.write("        <td>" + member.getId() + "</td>");
            w.write("        <td>" + member.getUsername() + "</td>");
            w.write("        <td>" + member.getAge() + "</td>");
            w.write("    </tr>");
        }

        w.write("    </tbody>");
        w.write("</table>");
```

```
            w.write("</body>");
            w.write("</html>");
        }
    }
```

`MemberListServlet`은 다음 순서로 동작한다.

1. `memberRepository.findAll()`을 통해 모든 회원을 조회한다.
2. 회원 목록 HTML을 for 루프를 통해서 회원 수 만큼 동적으로 생성하고 응답한다.

실행

- http://localhost:8080/servlet/members
- 저장된 회원 목록을 확인할 수 있다.

템플릿 엔진으로

지금까지 서블릿과 자바 코드만으로 HTML을 만들어보았다. 서블릿 덕분에 동적으로 원하는 HTML을 마음껏 만들 수 있다. 정적인 HTML 문서라면 화면이 계속 달라지는 회원의 저장 결과라던가, 회원 목록 같은 동적인 HTML을 만드는 일은 불가능 할 것이다.
그런데, 코드에서 보듯이 이것은 매우 복잡하고 비효율 적이다. 자바 코드로 HTML을 만들어 내는 것 보다 차라리 HTML 문서에 동적으로 변경해야 하는 부분만 자바 코드를 넣을 수 있다면 더 편리할 것이다.
이것이 바로 템플릿 엔진이 나온 이유이다. 템플릿 엔진을 사용하면 HTML 문서에서 필요한 곳만 코드를 적용해서 동적으로 변경할 수 있다.
템플릿 엔진에는 JSP, Thymeleaf, Freemarker, Velocity등이 있다.
다음 시간에는 JSP로 동일한 작업을 진행해보자.

> **참고**
> JSP는 성능과 기능면에서 다른 템플릿 엔진과의 경쟁에서 밀리면서, 점점 사장되어 가는 추세이다. 템플릿 엔진들은 각각 장단점이 있는데, 강의에서는 JSP는 앞부분에서 잠깐 다루고, 스프링과 잘 통합되는 Thymeleaf를 사용한다.

Welcome 페이지 변경

지금부터 서블릿에서 JSP, MVC 패턴, 직접 만드는 MVC 프레임워크, 그리고 스프링까지 긴 여정을 함께할 것이다. 편리하게 참고할 수 있도록 welcome 페이지를 변경하자.

`main/webapp/index.html`

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li><a href="basic.html">서블릿 basic</a></li>
    <li>서블릿
        <ul>
            <li><a href="/servlet/members/new-form">회원가입</a></li>
            <li><a href="/servlet/members">회원목록</a></li>
        </ul>
    </li>
    <li>JSP
        <ul>
            <li><a href="/jsp/members/new-form.jsp">회원가입</a></li>
            <li><a href="/jsp/members.jsp">회원목록</a></li>
        </ul>
    </li>
    <li>서블릿 MVC
        <ul>
            <li><a href="/servlet-mvc/members/new-form">회원가입</a></li>
            <li><a href="/servlet-mvc/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v1
        <ul>
            <li><a href="/front-controller/v1/members/new-form">회원가입</a></li>
            <li><a href="/front-controller/v1/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v2
        <ul>
            <li><a href="/front-controller/v2/members/new-form">회원가입</a></li>
            <li><a href="/front-controller/v2/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v3
        <ul>
```

```
            <li><a href="/front-controller/v3/members/new-form">회원가입</a></li>
            <li><a href="/front-controller/v3/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v4
        <ul>
            <li><a href="/front-controller/v4/members/new-form">회원가입</a></li>
            <li><a href="/front-controller/v4/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v5 - v3
        <ul>
            <li><a href="/front-controller/v5/v3/members/new-form">회원가입</a></
li>
            <li><a href="/front-controller/v5/v3/members">회원목록</a></li>
        </ul>
    </li>
    <li>FrontController - v5 - v4
        <ul>
            <li><a href="/front-controller/v5/v4/members/new-form">회원가입</a></
li>
            <li><a href="/front-controller/v5/v4/members">회원목록</a></li>
        </ul>
    </li>
    <li>SpringMVC - v1
        <ul>
            <li><a href="/springmvc/v1/members/new-form">회원가입</a></li>
            <li><a href="/springmvc/v1/members">회원목록</a></li>
        </ul>
    </li>
    <li>SpringMVC - v2
        <ul>
            <li><a href="/springmvc/v2/members/new-form">회원가입</a></li>
            <li><a href="/springmvc/v2/members">회원목록</a></li>
        </ul>
    </li>
    <li>SpringMVC - v3
        <ul>
            <li><a href="/springmvc/v3/members/new-form">회원가입</a></li>
```

```
            <li><a href="/springmvc/v3/members">회원목록</a></li>
        </ul>
    </li>
</ul>
</body>
</html>
```

JSP로 회원 관리 웹 애플리케이션 만들기

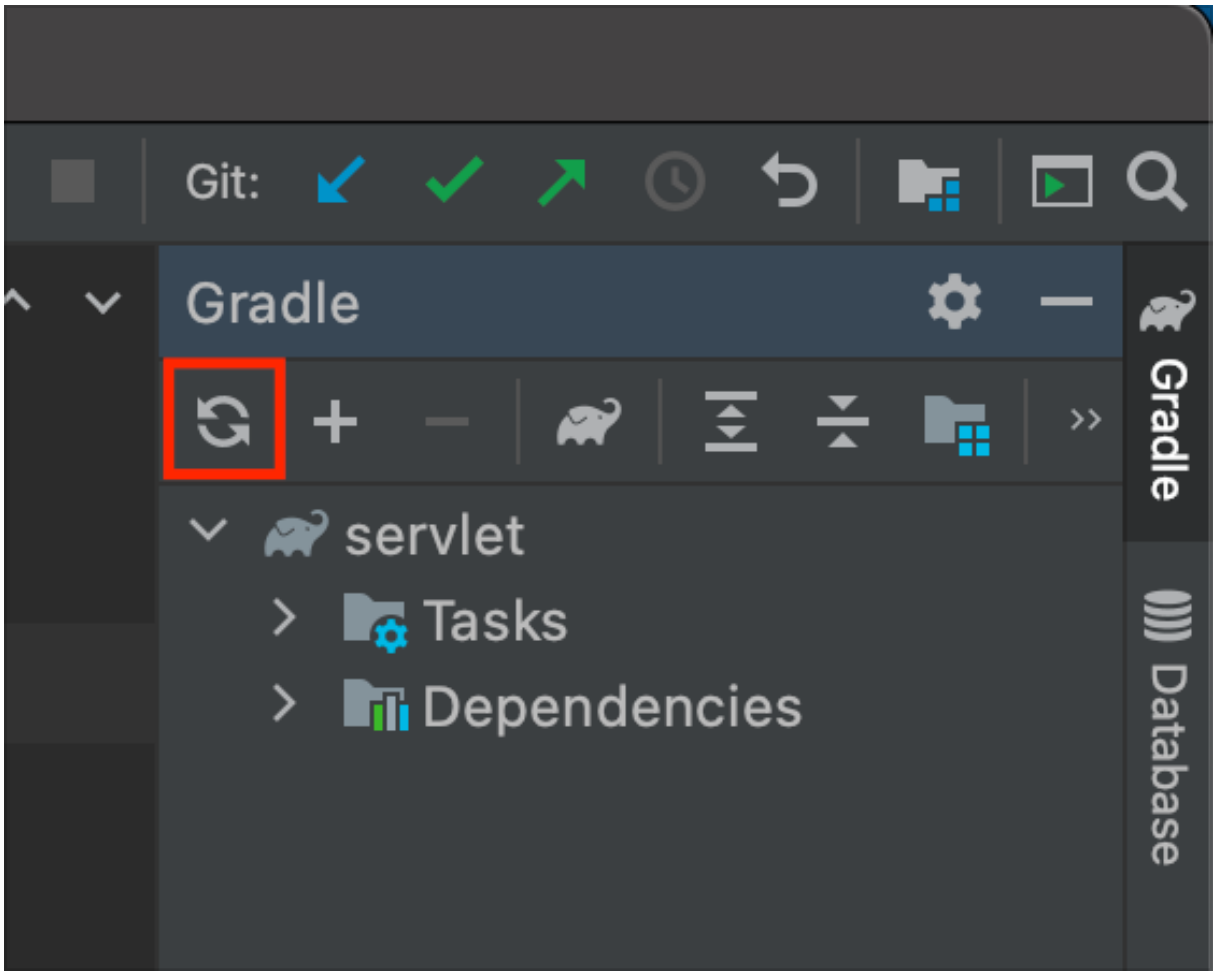
JSP 라이브러리 추가

JSP를 사용하려면 먼저 다음 라이브러리를 추가해야 한다.

`build.gradle` 에 추가

```
//JSP 추가 시작
implementation 'org.apache.tomcat.embed:tomcat-embed-jasper'
implementation 'javax.servlet:jstl'
//JSP 추가 끝
```

라이브러리를 추가하면 다음 버튼을 클릭해서 Gradle을 refresh 해주자.



회원 등록 폼 JSP

main/webapp/jsp/members/new-form.jsp

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>

<form action="/jsp/members/save.jsp" method="post">
    username: <input type="text" name="username" />
    age:      <input type="text" name="age" />
    <button type="submit">전송</button>
</form>

</body>
</html>
```

- `<%@ page contentType="text/html;charset=UTF-8" language="java" %>`
  - 첫 줄은 JSP문서라는 뜻이다. JSP 문서는 이렇게 시작해야 한다.

회원 등록 폼 JSP를 보면 첫 줄을 제외하고는 완전히 HTML와 똑같다. JSP는 서버 내부에서 서블릿으로 변환되는데, 우리가 만들었던 MemberFormServlet과 거의 비슷한 모습으로 변환된다.

실행

- http://localhost:8080/jsp/members/new-form.jsp
  - 실행시 `.jsp` 까지 함께 적어주어야 한다.

회원 저장 JSP

`main/webapp/jsp/members/save.jsp`

```
<%@ page import="hello.servlet.domain.member.MemberRepository" %>
<%@ page import="hello.servlet.domain.member.Member" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%

//    request, response 사용 가능

    MemberRepository memberRepository = MemberRepository.getInstance();

    System.out.println("save.jsp");
    String username = request.getParameter("username");
    int age = Integer.parseInt(request.getParameter("age"));

    Member member = new Member(username, age);
    System.out.println("member = " + member);
    memberRepository.save(member);

%>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
```

```
성공
<ul>
    <li>id=<%=member.getId()%></li>
    <li>username=<%=member.getUsername()%></li>
    <li>age=<%=member.getAge()%></li>
</ul>
<a href="/index.html">메인</a>
</body>
</html>
```

JSP는 자바 코드를 그대로 다 사용할 수 있다.

- `<%@ page import="hello.servlet.domain.member.MemberRepository" %>`
  - 자바의 import 문과 같다.
- `<% ~~ %>`
  - 이 부분에는 자바 코드를 입력할 수 있다.
- `<%= ~~ %>`
  - 이 부분에는 자바 코드를 출력할 수 있다.

회원 저장 JSP를 보면, 회원 저장 서블릿 코드와 같다. 다른 점이 있다면, HTML을 중심으로 하고, 자바 코드를 부분부분 입력해주었다. `<% ~ %>`를 사용해서 HTML 중간에 자바 코드를 출력하고 있다.

회원 목록 JSP

`main/webapp/jsp/members.jsp`

```
<%@ page import="java.util.List" %>
<%@ page import="hello.servlet.domain.member.MemberRepository" %>
<%@ page import="hello.servlet.domain.member.Member" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
    MemberRepository memberRepository = MemberRepository.getInstance();
    List<Member> members = memberRepository.findAll();
%>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
```

```
<body>
<a href="/index.html">메인</a>
<table>
    <thead>
    <th>id</th>
    <th>username</th>
    <th>age</th>
    </thead>
    <tbody>
<%
    for (Member member : members) {
        out.write("    <tr>");
        out.write("        <td>" + member.getId() + "</td>");
        out.write("        <td>" + member.getUsername() + "</td>");
        out.write("        <td>" + member.getAge() + "</td>");
        out.write("    </tr>");
    }
%>
    </tbody>
</table>

</body>
</html>
```

회원 리포지토리를 먼저 조회하고, 결과 List를 사용해서 중간에 `<tr><td>` HTML 태그를 반복해서 출력하고 있다.

서블릿과 JSP의 한계

서블릿으로 개발할 때는 뷰(View)화면을 위한 HTML을 만드는 작업이 자바 코드에 섞여서 지저분하고 복잡했다.
JSP를 사용한 덕분에 뷰를 생성하는 HTML 작업을 깔끔하게 가져가고, 중간중간 동적으로 변경이 필요한 부분에만 자바 코드를 적용했다. 그런데 이렇게 해도 해결되지 않는 몇가지 고민이 남는다.

회원 저장 JSP를 보자. 코드의 상위 절반은 회원을 저장하기 위한 비즈니스 로직이고, 나머지 하위 절반만 결과를 HTML로 보여주기 위한 뷰 영역이다. 회원 목록의 경우에도 마찬가지다.

코드를 잘 보면, JAVA 코드, 데이터를 조회하는 리포지토리 등등 다양한 코드가 모두 JSP에 노출되어 있다. JSP가 너무 많은 역할을 한다. 이렇게 작은 프로젝트도 벌써 머리가 아파오는데, 수백 수천줄이 넘어가는 JSP를 떠올려보면 정말 지옥과 같을 것이다. (유지보수 지옥 썰)

MVC 패턴의 등장

비즈니스 로직은 서블릿 처럼 다른곳에서 처리하고, JSP는 목적에 맞게 HTML로 화면(View)을 그리는 일에 집중하도록 하자. 과거 개발자들도 모두 비슷한 고민이 있었고, 그래서 MVC 패턴이 등장했다. 우리도 직접 MVC 패턴을 적용해서 프로젝트를 리팩터링 해보자.

MVC 패턴 - 개요

너무 많은 역할

하나의 서블릿이나 JSP만으로 비즈니스 로직과 뷰 렌더링까지 모두 처리하게 되면, 너무 많은 역할을 하게되고, 결과적으로 유지보수가 어려워진다. 비즈니스 로직을 호출하는 부분에 변경이 발생해도 해당 코드를 손대야 하고, UI를 변경할 일이 있어도 비즈니스 로직이 함께 있는 해당 파일을 수정해야 한다. HTML 코드 하나 수정해야 하는데, 수백줄의 자바 코드가 함께 있다고 상상해보라! 또는 비즈니스 로직을 하나 수정해야 하는데 수백 수천줄의 HTML 코드가 함께 있다고 상상해보라.

변경의 라이프 사이클

사실 이게 정말 중요한데, 진짜 문제는 둘 사이에 변경의 라이프 사이클이 다르다는 점이다. 예를 들어서 UI를 일부 수정하는 일과 비즈니스 로직을 수정하는 일은 각각 다르게 발생할 가능성이 매우 높고 대부분 서로에게 영향을 주지 않는다. 이렇게 변경의 라이프 사이클이 다른 부분을 하나의 코드로 관리하는 것은 유지보수하기 좋지 않다. (물론 UI가 많이 변하면 함께 변경될 가능성도 있다.)

기능 특화

특히 JSP 같은 뷰 템플릿은 화면을 렌더링 하는데 최적화 되어 있기 때문에 이 부분의 업무만 담당하는 것이 가장 효과적이다.

Model View Controller

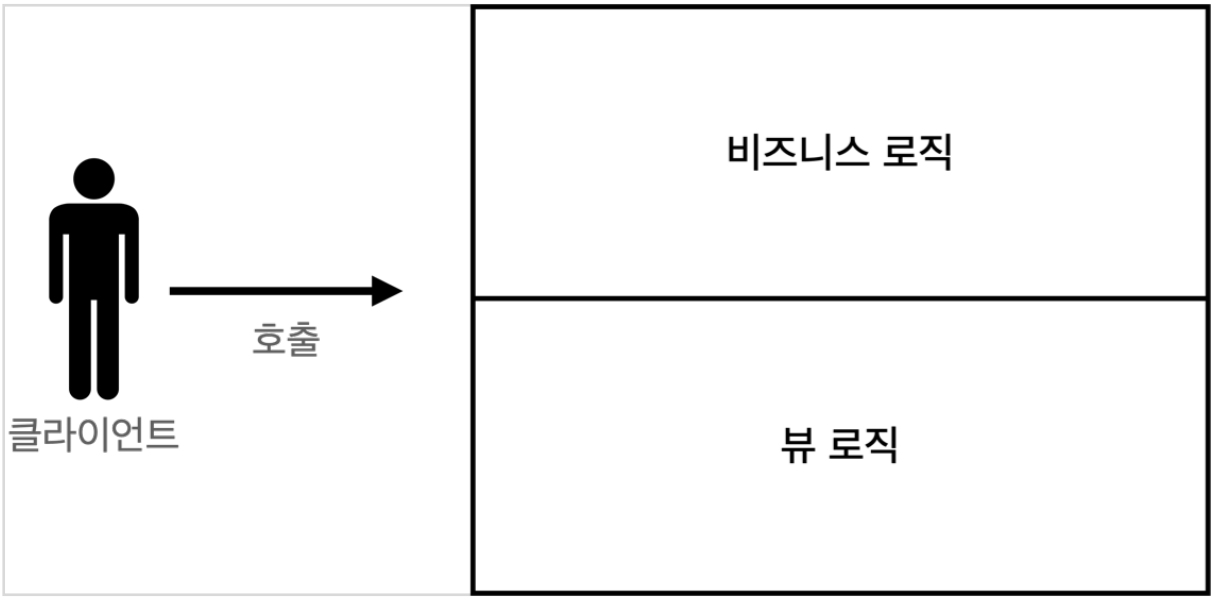
MVC 패턴은 지금까지 학습한 것 처럼 하나의 서블릿이나, JSP로 처리하던 것을 컨트롤러(Controller)와 뷰(View)라는 영역으로 서로 역할을 나눈 것을 말한다. 웹 애플리케이션은 보통 이 MVC 패턴을 사용한다.

컨트롤러: HTTP 요청을 받아서 파라미터를 검증하고, 비즈니스 로직을 실행한다. 그리고 뷰에 전달할 결과 데이터를 조회해서 모델에 담는다.

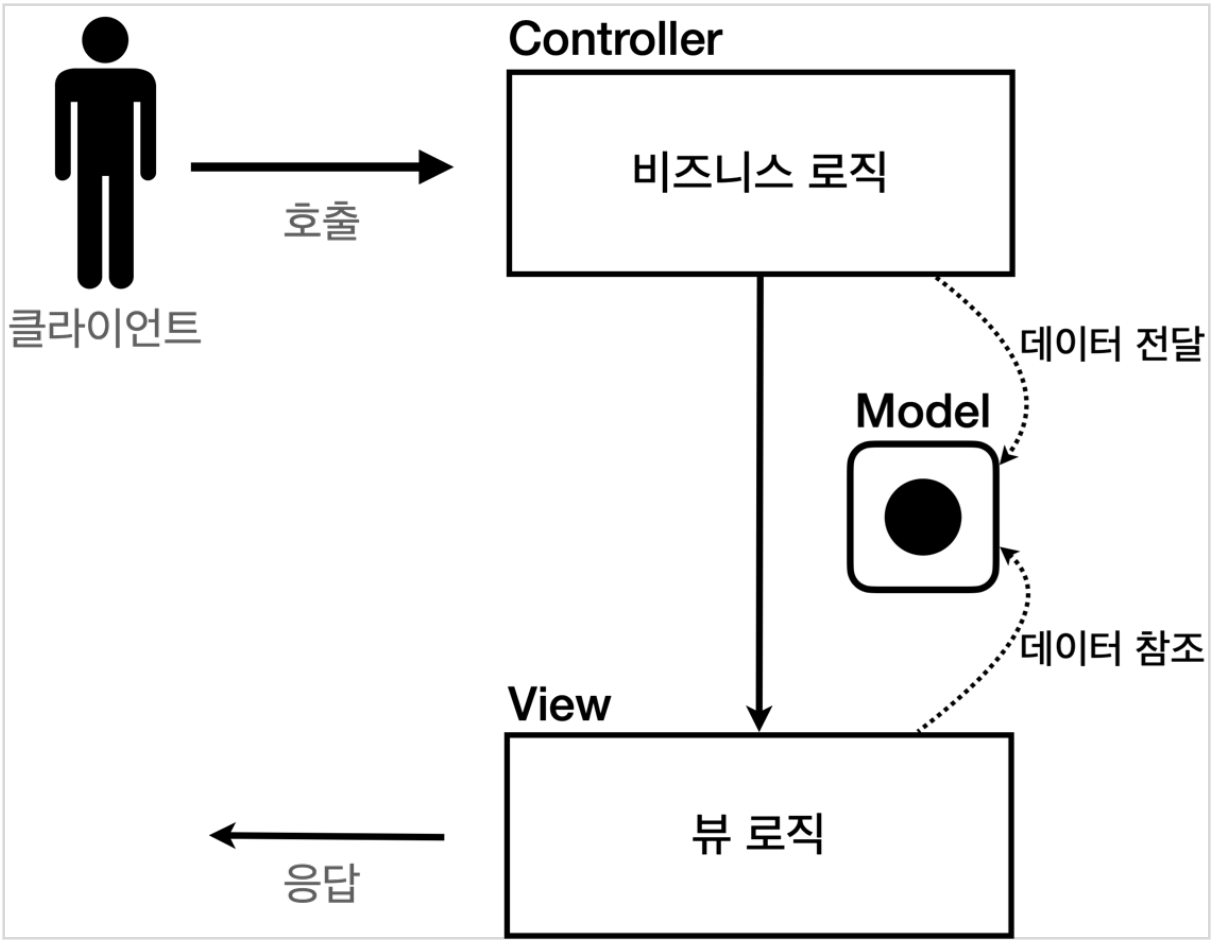
모델: 뷰에 출력할 데이터를 담아둔다. 뷰가 필요한 데이터를 모두 모델에 담아서 전달해주는 덕분에 뷰는 비즈니스 로직이나 데이터 접근을 몰라도 되고, 화면을 렌더링 하는 일에 집중할 수 있다.
뷰: 모델에 담겨있는 데이터를 사용해서 화면을 그리는 일에 집중한다. 여기서는 HTML을 생성하는 부분을 말한다.

> **참고**
> 컨트롤러에 비즈니스 로직을 둘 수도 있지만, 이렇게 되면 컨트롤러가 너무 많은 역할을 담당한다. 그래서 일반적으로 비즈니스 로직은 서비스(Service)라는 계층을 별도로 만들어서 처리한다. 그리고 컨트롤러는 비즈니스 로직이 있는 서비스를 호출하는 역할을 담당한다. 참고로 비즈니스 로직을 변경하면 비즈니스 로직을 호출하는 컨트롤러의 코드도 변경될 수 있다. 앞에서는 이해를 돕기 위해 비즈니스 로직을 호출한다는 표현 보다는, 비즈니스 로직이라 설명했다.

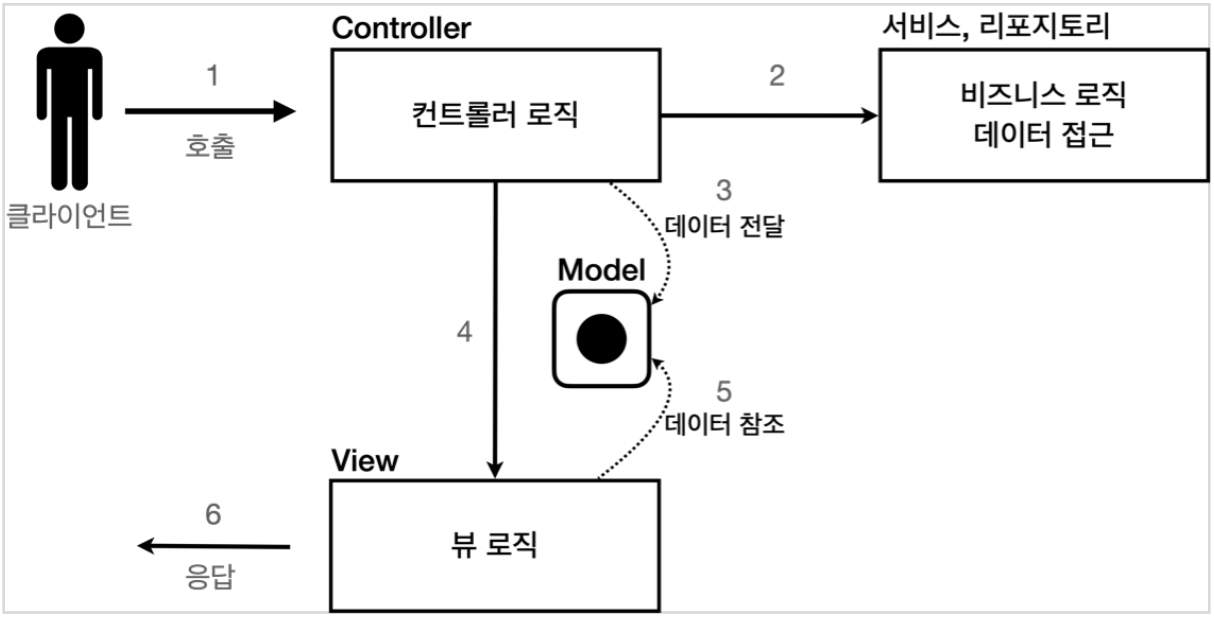
MVC 패턴 이전



MVC 패턴1


Controller
비즈니스 로직
호출
클라이언트
데이터 전달
Model
데이터 참조
View
뷰 로직
응답

MVC 패턴2


Controller
서비스, 리포지토리
1
호출
컨트롤러 로직
2
비즈니스 로직
데이터 접근
클라이언트
3
데이터 전달
Model
4
5
데이터 참조
View
6
응답
뷰 로직

MVC 패턴 - 적용

서블릿을 컨트롤러로 사용하고, JSP를 뷰로 사용해서 MVC 패턴을 적용해보자.
Model은 HttpServletRequest 객체를 사용한다. request는 내부에 데이터 저장소를 가지고 있는데, `request.setAttribute()`, `request.getAttribute()`를 사용하면 데이터를 보관하고, 조회할 수 있다.

회원 등록

회원 등록 폼 - 컨트롤러

`hello.servlet.web.servletmvc.MvcMemberFormServlet`

```
package hello.servlet.web.servletmvc;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "mvcMemberFormServlet", urlPatterns = "/servlet-mvc/members/
new-form")
public class MvcMemberFormServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        String viewPath = "/WEB-INF/views/new-form.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);

    }
}
```

`dispatcher.forward()` : 다른 서블릿이나 JSP로 이동할 수 있는 기능이다. 서버 내부에서 다시 호출이

발생한다.

> `/WEB-INF`
> 이 경로안에 JSP가 있으면 외부에서 직접 JSP를 호출할 수 없다. 우리가 기대하는 것은 항상 컨트롤러를 통해서 JSP를 호출하는 것이다.

> **redirect vs forward**
> 리다이렉트는 실제 클라이언트(웹 브라우저)에 응답이 나갔다가, 클라이언트가 redirect 경로로 다시 요청한다. 따라서 클라이언트가 인지할 수 있고, URL 경로도 실제로 변경된다. 반면에 포워드는 서버 내부에서 일어나는 호출이기 때문에 클라이언트가 전혀 인지하지 못한다.

회원 등록 폼 - 뷰

`main/webapp/WEB-INF/views/new-form.jsp`

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

<!-- 상대경로 사용, [현재 URL이 속한 계층 경로 + /save] -->
<form action="save" method="post">
    username: <input type="text" name="username" />
    age:      <input type="text" name="age" />
    <button type="submit">전송</button>
</form>

</body>
</html>
```

여기서 form의 action을 보면 절대 경로(로 시작)이 아니라 상대경로(로 시작X)하는 것을 확인할 수 있다.
이렇게 상대경로를 사용하면 폼 전송시 현재 URL이 속한 계층 경로 + save가 호출된다.
현재 계층 경로: `/servlet-mvc/members/`
결과: `/servlet-mvc/members/save`

실행

- http://localhost:8080/servlet-mvc/members/new-form
- HTML Form이 잘 나오는 것을 확인할 수 있다.

> **주의!**
> 이후 코드에서 해당 jsp를 계속 사용하기 때문에 상대경로를 사용한 부분을 그대로 유지해야 한다.

회원 저장

회원 저장 - 컨트롤러

MvcMemberSaveServlet

```
package hello.servlet.web.servletmvc;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "mvcMemberSaveServlet", urlPatterns = "/servlet-mvc/members/
save")
public class MvcMemberSaveServlet extends HttpServlet {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));
```

```
        Member member = new Member(username, age);
        System.out.println("member = " + member);
        memberRepository.save(member);

        //Model에 데이터를 보관한다.
        request.setAttribute("member", member);

        String viewPath = "/WEB-INF/views/save-result.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }
}
```

HttpServletRequest를 Model로 사용한다.
request가 제공하는 `setAttribute()`를 사용하면 request 객체에 데이터를 보관해서 뷰에 전달할 수 있다.
뷰는 `request.getAttribute()`를 사용해서 데이터를 꺼내면 된다.

회원 저장 - 뷰

`main/webapp/WEB-INF/views/save-result.jsp`

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
성공
<ul>
    <li>id=${member.id}</li>
    <li>username=${member.username}</li>
    <li>age=${member.age}</li>
</ul>
<a href="/index.html">메인</a>
</body>
</html>
```

`<%= request.getAttribute("member")%>`로 모델에 저장한 member 객체를 꺼낼 수 있지만, 너무 복잡해진다.
JSP는 `${}` 문법을 제공하는데, 이 문법을 사용하면 request의 attribute에 담긴 데이터를 편리하게 조회할 수 있다.

실행

- http://localhost:8080/servlet-mvc/members/new-form
- HTML Form에 데이터를 입력하고 전송을 누르면 저장 결과를 확인할 수 있다.

MVC 덕분에 컨트롤러 로직과 뷰 로직을 확실하게 분리한 것을 확인할 수 있다. 향후 화면에 수정이 발생하면 뷰 로직만 변경하면 된다.

회원 목록 조회

회원 목록 조회 - 컨트롤러

MvcMemberListServlet

```
package hello.servlet.web.servletmvc;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

@WebServlet(name = "mvcMemberListServlet", urlPatterns = "/servlet-mvc/
members")
public class MvcMemberListServlet extends HttpServlet {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
```

```
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        System.out.println("MvcMemberListServlet.service");
        List<Member> members = memberRepository.findAll();

        request.setAttribute("members", members);

        String viewPath = "/WEB-INF/views/members.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }
}
```

request 객체를 사용해서 `List<Member> members` 를 모델에 보관했다.

회원 목록 조회 - 뷰

`main/webapp/WEB-INF/views/members.jsp`

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<a href="/index.html">메인</a>
<table>
    <thead>
    <th>id</th>
    <th>username</th>
    <th>age</th>
    </thead>
    <tbody>
    <c:forEach var="item" items="${members}">
        <tr>
```

```
            <td>${item.id}</td>
            <td>${item.username}</td>
            <td>${item.age}</td>
        </tr>
    </c:forEach>
    </tbody>
</table>

</body>
</html>
```

모델에 담아둔 members를 JSP가 제공하는 taglib기능을 사용해서 반복하면서 출력했다.
`members` 리스트에서 `member`를 순서대로 꺼내서 `item` 변수에 담고, 출력하는 과정을 반복한다.

`<c:forEach>` 이 기능을 사용하려면 다음과 같이 선언해야 한다.
`<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>`

해당 기능을 사용하지 않고, 다음과 같이 출력해도 되지만, 매우 지저분하다.

```
<%
    for (Member member : members) {
        out.write("    <tr>");
        out.write("        <td>" + member.getId() + "</td>");
        out.write("        <td>" + member.getUsername() + "</td>");
        out.write("        <td>" + member.getAge() + "</td>");
        out.write("    </tr>");
    }
%>
```

JSP와 같은 뷰 템플릿은 이렇게 화면을 렌더링 하는데 특화된 다양한 기능을 제공한다.

실행

- http://localhost:8080/servlet-mvc/members
- 저장된 결과 목록을 확인할 수 있다.

> **참고**
> 앞서 설명했듯이 JSP를 학습하는 것이 이 강의의 주 목적이 아니다. JSP가 더 궁금한 분들은 이미 수 많은

> 자료들이 있으므로 JSP로 검색하거나 관련된 책을 참고하길 바란다. (반나절이면 대부분의 기능을 학습할 수 있다.)

MVC 패턴 - 한계

MVC 패턴을 적용한 덕분에 컨트롤러의 역할과 뷰를 렌더링 하는 역할을 명확하게 구분할 수 있다.
특히 뷰는 화면을 그리는 역할에 충실한 덕분에, 코드가 깔끔하고 직관적이다. 단순하게 모델에서 필요한 데이터를 꺼내고, 화면을 만들면 된다.
그런데 컨트롤러는 딱 봐도 중복이 많고, 필요하지 않는 코드들도 많이 보인다.

MVC 컨트롤러의 단점

포워드 중복
View로 이동하는 코드가 항상 중복 호출되어야 한다. 물론 이 부분을 메서드로 공통화해도 되지만, 해당 메서드도 항상 직접 호출해야 한다.

```
RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
dispatcher.forward(request, response);
```

ViewPath에 중복

```
String viewPath = "/WEB-INF/views/new-form.jsp";
```

- prefix: `/WEB-INF/views/`
- suffix: `.jsp`

그리고 만약 jsp가 아닌 thymeleaf 같은 다른 뷰로 변경한다면 전체 코드를 다 변경해야 한다.

사용하지 않는 코드
다음 코드를 사용할 때도 있고, 사용하지 않을 때도 있다. 특히 response는 현재 코드에서 사용되지 않는다.

```
HttpServletRequest request, HttpServletResponse response
```

그리고 이런 `HttpServletRequest`, `HttpServletResponse`를 사용하는 코드는 테스트 케이스를 작성하기도 어렵다.

공통 처리가 어렵다.
기능이 복잡해질 수 록 컨트롤러에서 공통으로 처리해야 하는 부분이 점점 더 많이 증가할 것이다. 단순히 공통 기능을 메서드로 뽑으면 될 것 같지만, 결과적으로 해당 메서드를 항상 호출해야 하고, 실수로 호출하지 않으면 문제가 될 것이다. 그리고 호출하는 것 자체도 중복이다.

정리하면 공통 처리가 어렵다는 문제가 있다.
이 문제를 해결하려면 컨트롤러 호출 전에 먼저 공통 기능을 처리해야 한다. 소위 **수문장 역할**을 하는 기능이 필요하다. **프론트 컨트롤러(Front Controller) 패턴**을 도입하면 이런 문제를 깔끔하게 해결할 수 있다.
(입구를 하나로!)
스프링 MVC의 핵심도 바로 이 프론트 컨트롤러에 있다.

정리

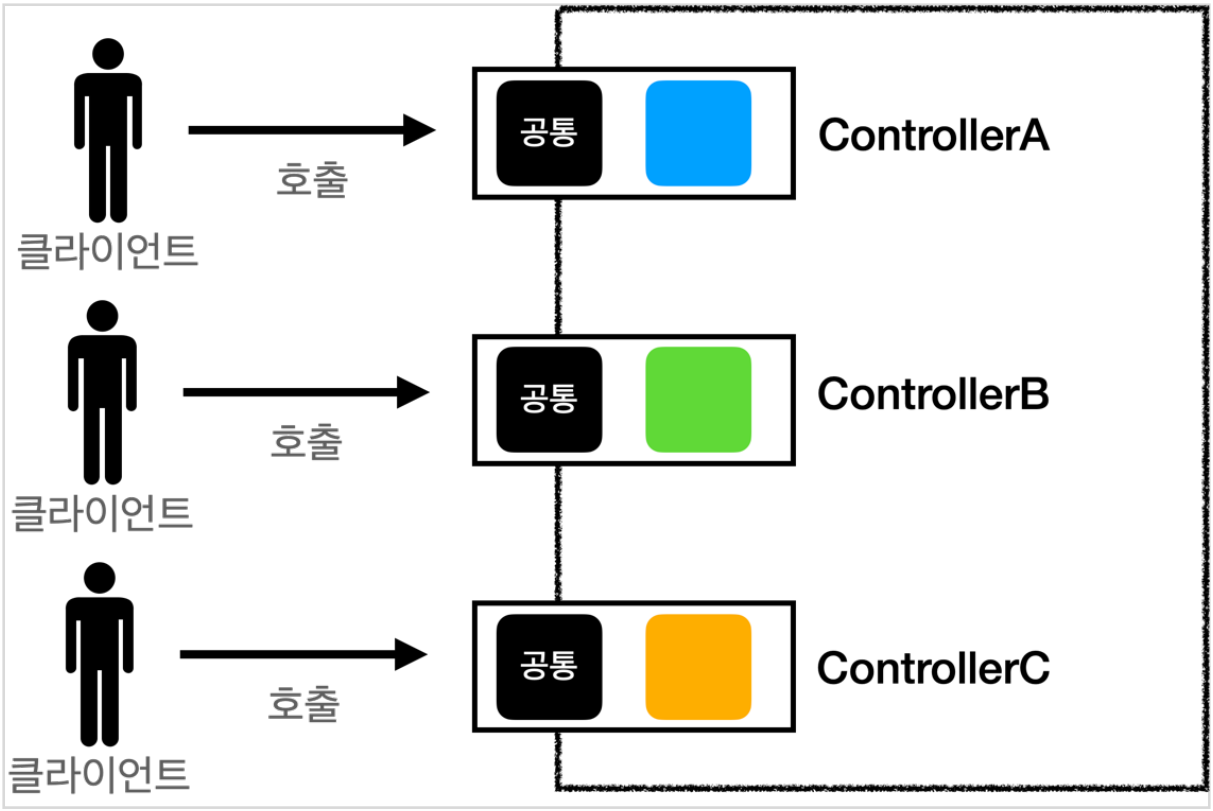
4. MVC 프레임워크 만들기

#인강/4. 스프링 MVC 1/강의#

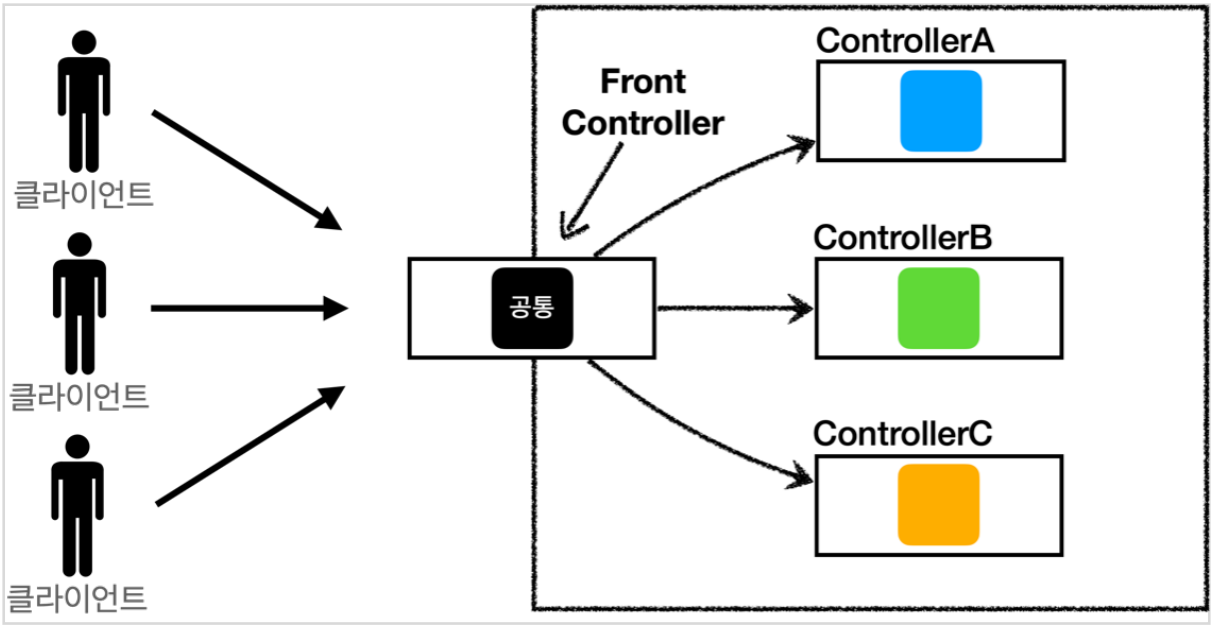
목차

프론트 컨트롤러 패턴 소개

프론트 컨트롤러 도입 전



프론트 컨트롤러 도입 후



FrontController 패턴 특징

- 프론트 컨트롤러 서블릿 하나로 클라이언트의 요청을 받음
- 프론트 컨트롤러가 요청에 맞는 컨트롤러를 찾아서 호출
- 입구를 하나로!

- 공통 처리 가능
- 프론트 컨트롤러를 제외한 나머지 컨트롤러는 서블릿을 사용하지 않아도 됨

스프링 웹 MVC와 프론트 컨트롤러

스프링 웹 MVC의 핵심도 바로 **FrontController**

스프링 웹 MVC의 **DispatcherServlet**이 FrontController 패턴으로 구현되어 있음

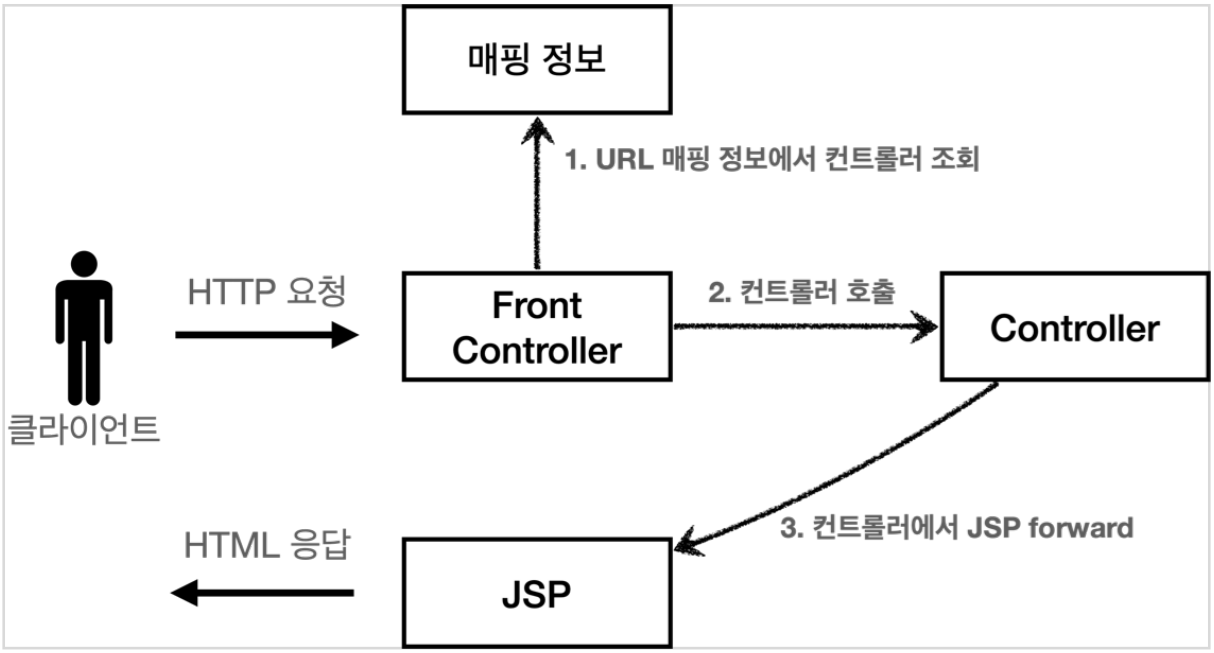
프론트 컨트롤러 도입 - v1

프론트 컨트롤러를 단계적으로 도입해보자.

이번 목표는 기존 코드를 최대한 유지하면서, 프론트 컨트롤러를 도입하는 것이다.

먼저 구조를 맞추어두고 점진적으로 리펙터링 해보자.

V1 구조



ControllerV1

```
package hello.servlet.web.frontcontroller.v1;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface ControllerV1 {

    void process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException;
}
```

서블릿과 비슷한 모양의 컨트롤러 인터페이스를 도입한다. 각 컨트롤러들은 이 인터페이스를 구현하면 된다. 프론트 컨트롤러는 이 인터페이스를 호출해서 구현과 관계없이 로직의 일관성을 가져갈 수 있다.

이제 이 인터페이스를 구현한 컨트롤러를 만들어보자. 지금 단계에서는 기존 로직을 최대한 유지하는게 핵심이다.

MemberFormControllerV1 - 회원 등록 컨트롤러

```
package hello.servlet.web.frontcontroller.v1.controller;

import hello.servlet.web.frontcontroller.v1.ControllerV1;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class MemberFormControllerV1 implements ControllerV1 {

    @Override
    public void process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String viewPath = "/WEB-INF/views/new-form.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }
}
```

MemberSaveControllerV1 - 회원 저장 컨트롤러

```
package hello.servlet.web.frontcontroller.v1.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.v1.ControllerV1;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class MemberSaveControllerV1 implements ControllerV1 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public void process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);

        request.setAttribute("member", member);

        String viewPath = "/WEB-INF/views/save-result.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }
}
```

MemberListControllerV1 - 회원 목록 컨트롤러

```
package hello.servlet.web.frontcontroller.v1.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.v1.ControllerV1;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

public class MemberListControllerV1 implements ControllerV1 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public void process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        List<Member> members = memberRepository.findAll();
        request.setAttribute("members", members);

        String viewPath = "/WEB-INF/views/members.jsp";
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }
}
```

내부 로직은 기존 서블릿과 거의 같다.
이제 프론트 컨트롤러를 만들어보자.

FrontControllerServletV1 - 프론트 컨트롤러

```
package hello.servlet.web.frontcontroller.v1;
```

```
import hello.servlet.web.frontcontroller.v1.controller.MemberFormControllerV1;
import hello.servlet.web.frontcontroller.v1.controller.MemberListControllerV1;
import hello.servlet.web.frontcontroller.v1.controller.MemberSaveControllerV1;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

@WebServlet(name = "frontControllerServletV1", urlPatterns = "/front-
controller/v1/*")
public class FrontControllerServletV1 extends HttpServlet {

    private Map<String, ControllerV1> controllerMap = new HashMap<>();

    public FrontControllerServletV1() {
        controllerMap.put("/front-controller/v1/members/new-form", new
MemberFormControllerV1());
        controllerMap.put("/front-controller/v1/members/save", new
MemberSaveControllerV1());
        controllerMap.put("/front-controller/v1/members", new
MemberListControllerV1());
    }

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        System.out.println("FrontControllerServletV1.service");
        String requestURI = request.getRequestURI();

        ControllerV1 controller = controllerMap.get(requestURI);
        if (controller == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
```

```
            return;
        }

        controller.process(request, response);
    }
}
```

프론트 컨트롤러 분석

urlPatterns

- `urlPatterns = "/front-controller/v1/*"` : `/front-controller/v1` 를 포함한 하위 모든 요청은 이 서블릿에서 받아들인다.
- 예) `/front-controller/v1` , `/front-controller/v1/a` , `/front-controller/v1/a/b`

controllerMap

- key: 매핑 URL
- value: 호출될 컨트롤러

service()

먼저 `requestURI` 를 조회해서 실제 호출할 컨트롤러를 `controllerMap` 에서 찾는다. 만약 없다면 404(SC_NOT_FOUND) 상태 코드를 반환한다.
컨트롤러를 찾고 `controller.process(request, response);` 을 호출해서 해당 컨트롤러를 실행한다.

JSP

JSP는 이전 MVC에서 사용했던 것을 그대로 사용한다.

실행

- 등록: http://localhost:8080/front-controller/v1/members/new-form
- 목록: http://localhost:8080/front-controller/v1/members

기존 서블릿, JSP로 만든 MVC와 동일하게 실행 되는 것을 확인할 수 있다.

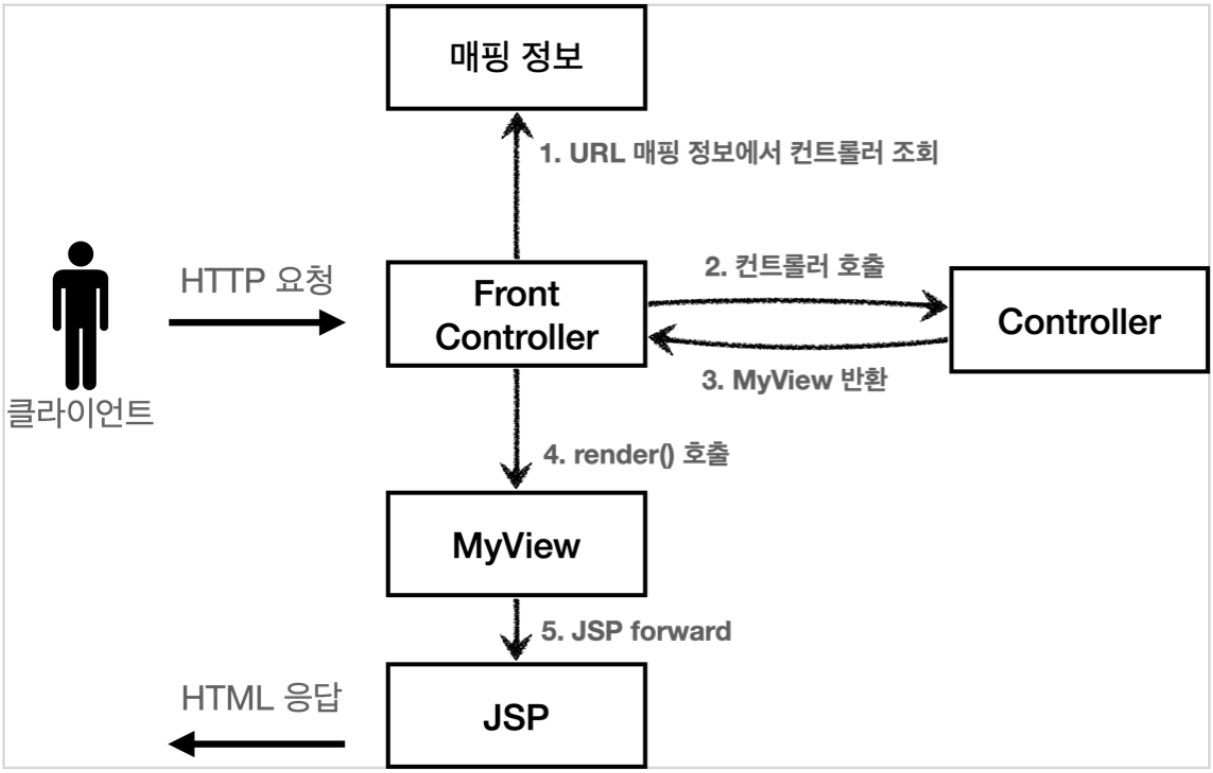
View 분리 - v2

모든 컨트롤러에서 뷰로 이동하는 부분에 중복이 있고, 깔끔하지 않다.

```
String viewPath = "/WEB-INF/views/new-form.jsp";
RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
dispatcher.forward(request, response);
```

이 부분을 깔끔하게 분리하기 위해 별도로 뷰를 처리하는 객체를 만들자.

V2 구조



MyView

뷰 객체는 이후 다른 버전에서도 함께 사용하므로 패키지 위치를 `frontcontroller`에 두었다.

```
package hello.servlet.web.frontcontroller;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
public class MyView {

    private String viewPath;

    public MyView(String viewPath) {
        this.viewPath = viewPath;
    }

    public void render(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }

}
```

이 코드만 봐서는 어떻게 활용하는지 아직 감이 안올 것이다. 다음 버전의 컨트롤러 인터페이스를 만들어보자.
컨트롤러가 뷰를 반환하는 특징이 있다.

ControllerV2

```
package hello.servlet.web.frontcontroller.v2;

import hello.servlet.web.frontcontroller.MyView;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface ControllerV2 {

    MyView process(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException;
}
```

MemberFormControllerV2 - 회원 등록 폼

```
package hello.servlet.web.frontcontroller.v2.controller;

import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v2.ControllerV2;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class MemberFormControllerV2 implements ControllerV2 {

    @Override
    public MyView process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        return new MyView("/WEB-INF/views/new-form.jsp");
    }
}
```

이제 각 컨트롤러는 복잡한 `dispatcher.forward()`를 직접 생성해서 호출하지 않아도 된다. 단순히 MyView 객체를 생성하고 거기에 뷰 이름만 넣고 반환하면 된다.

`ControllerV1`을 구현한 클래스와 `ControllerV2`를 구현한 클래스를 비교해보면, 이 부분의 중복이 확실하게 제거된 것을 확인할 수 있다.

MemberSaveControllerV2 - 회원 저장

```
package hello.servlet.web.frontcontroller.v2.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.MyView;
```

```
import hello.servlet.web.frontcontroller.v2.ControllerV2;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class MemberSaveControllerV2 implements ControllerV2 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public MyView process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);

        request.setAttribute("member", member);

        return new MyView("/WEB-INF/views/save-result.jsp");
    }
}
```

MemberListControllerV2 - 회원 목록

```
package hello.servlet.web.frontcontroller.v2.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v2.ControllerV2;

import javax.servlet.ServletException;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

public class MemberListControllerV2 implements ControllerV2 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public MyView process(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        List<Member> members = memberRepository.findAll();
        request.setAttribute("members", members);

        return new MyView("/WEB-INF/views/members.jsp");
    }
}
```

프론트 컨트롤러 V2

```
package hello.servlet.web.frontcontroller.v2;

import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v2.controller.MemberFormControllerV2;
import hello.servlet.web.frontcontroller.v2.controller.MemberListControllerV2;
import hello.servlet.web.frontcontroller.v2.controller.MemberSaveControllerV2;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```
@WebServlet(name = "frontControllerServletV2", urlPatterns = "/front-
controller/v2/*")
public class FrontControllerServletV2 extends HttpServlet {

    private Map<String, ControllerV2> controllerMap = new HashMap<>();

    public FrontControllerServletV2() {
        controllerMap.put("/front-controller/v2/members/new-form", new
MemberFormControllerV2());
        controllerMap.put("/front-controller/v2/members/save", new
MemberSaveControllerV2());
        controllerMap.put("/front-controller/v2/members", new
MemberListControllerV2());
    }

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        String requestURI = request.getRequestURI();

        ControllerV2 controller = controllerMap.get(requestURI);
        if (controller == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        MyView view = controller.process(request, response);
        view.render(request, response);
    }
}
```

ControllerV2의 반환 타입이 `MyView` 이므로 프론트 컨트롤러는 컨트롤러의 호출 결과로 `MyView` 를 반환받는다. 그리고 `view.render()` 를 호출하면 `forward` 로직을 수행해서 JSP가 실행된다.

`MyView.render()`

```
public void render(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
    dispatcher.forward(request, response);
}
```

프론트 컨트롤러의 도입으로 `MyView` 객체의 `render()`를 호출하는 부분을 모두 일관되게 처리할 수 있다. 각각의 컨트롤러는 `MyView` 객체를 생성만 해서 반환하면 된다.

실행

- 등록: http://localhost:8080/front-controller/v2/members/new-form
- 목록: http://localhost:8080/front-controller/v2/members

Model 추가 - v3

서블릿 종속성 제거

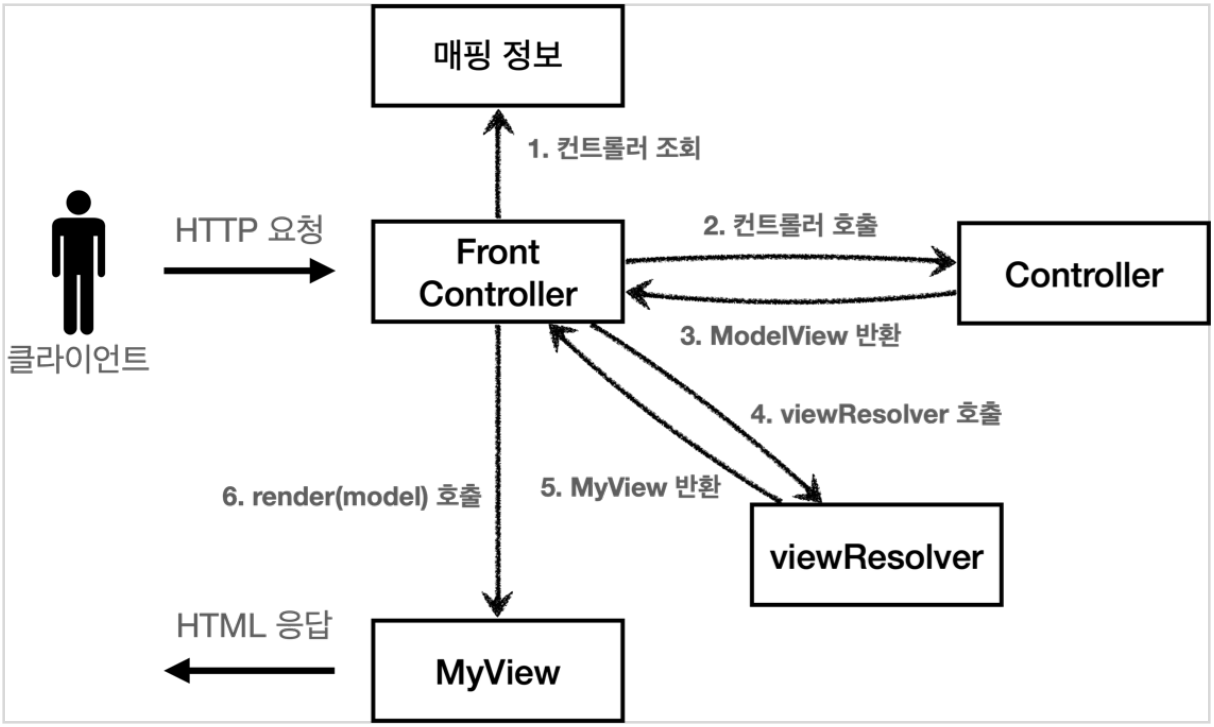
컨트롤러 입장에서 HttpServletRequest, HttpServletResponse이 꼭 필요할까?
요청 파라미터 정보는 자바의 Map으로 대신 넘기도록 하면 지금 구조에서는 컨트롤러가 서블릿 기술을 몰라도 동작할 수 있다.
그리고 request 객체를 Model로 사용하는 대신에 별도의 Model 객체를 만들어서 반환하면 된다.
우리가 구현하는 컨트롤러가 서블릿 기술을 전혀 사용하지 않도록 변경해보자.
이렇게 하면 구현 코드도 매우 단순해지고, 테스트 코드 작성이 쉽다.

뷰 이름 중복 제거

컨트롤러에서 지정하는 뷰 이름에 중복이 있는 것을 확인할 수 있다.
컨트롤러는 **뷰의 논리 이름**을 반환하고, 실제 물리 위치의 이름은 프론트 컨트롤러에서 처리하도록 단순화하자.
이렇게 해두면 향후 뷰의 폴더 위치가 함께 이동해도 프론트 컨트롤러만 고치면 된다.

- `/WEB-INF/views/new-form.jsp` ➜ **new-form**
- `/WEB-INF/views/save-result.jsp` ➜ **save-result**
- `/WEB-INF/views/members.jsp` ➜ **members**

V3 구조



ModelView

지금까지 컨트롤러에서 서블릿에 종속적인 HttpServletRequest를 사용했다. 그리고 Model도 `request.setAttribute()`를 통해 데이터를 저장하고 뷰에 전달했다.
서블릿의 종속성을 제거하기 위해 Model을 직접 만들고, 추가로 View 이름까지 전달하는 객체를 만들어보자.
(이번 버전에서는 컨트롤러에서 HttpServletRequest를 사용할 수 없다. 따라서 직접 `request.setAttribute()`를 호출할 수 도 없다. 따라서 Model이 별도로 필요하다.)

참고로 `ModelView` 객체는 다른 버전에서도 사용하므로 패키지를 `frontcontroller`에 둔다.

ModelView

```
package hello.servlet.web.frontcontroller;

import java.util.HashMap;
import java.util.Map;

public class ModelView {
    private String viewName;
```

```
    private Map<String, Object> model = new HashMap<>();

    public ModelView(String viewName) {
        this.viewName = viewName;
    }

    public String getViewName() {
        return viewName;
    }

    public void setViewName(String viewName) {
        this.viewName = viewName;
    }

    public Map<String, Object> getModel() {
        return model;
    }

    public void setModel(Map<String, Object> model) {
        this.model = model;
    }

}
```

뷰의 이름과 뷰를 렌더링할 때 필요한 model 객체를 가지고 있다. model은 단순히 map으로 되어 있으므로 컨트롤러에서 뷰에 필요한 데이터를 key, value로 넣어주면 된다.

ControllerV3

```
package hello.servlet.web.frontcontroller.v3;

import hello.servlet.web.frontcontroller.ModelView;

import java.util.Map;

public interface ControllerV3 {

    ModelView process(Map<String, String> paramMap);
```

```
}
```

이 컨트롤러는 서블릿 기술을 전혀 사용하지 않는다. 따라서 구현이 매우 단순해지고, 테스트 코드 작성시 테스트 하기 쉽다.
HttpServletRequest가 제공하는 파라미터는 프론트 컨트롤러가 paramMap에 담아서 호출해주면 된다.
응답 결과로 뷰 이름과 뷰에 전달할 Model 데이터를 포함하는 ModelView 객체를 반환하면 된다.

MemberFormControllerV3 - 회원 등록 폼

```
package hello.servlet.web.frontcontroller.v3.controller;

import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.v3.ControllerV3;

import java.util.Map;

public class MemberFormControllerV3 implements ControllerV3 {

    @Override
    public ModelView process(Map<String, String> paramMap) {
        return new ModelView("new-form");
    }
}
```

`ModelView`를 생성할 때 `new-form`이라는 view의 논리적인 이름을 지정한다. 실제 물리적인 이름은 프론트 컨트롤러에서 처리한다.

MemberSaveControllerV3 - 회원 저장

```
package hello.servlet.web.frontcontroller.v3.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.v3.ControllerV3;
```

```
import java.util.Map;

public class MemberSaveControllerV3 implements ControllerV3 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public ModelView process(Map<String, String> paramMap) {
        String username = paramMap.get("username");
        int age = Integer.parseInt(paramMap.get("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);

        ModelView mv = new ModelView("save-result");
        mv.getModel().put("member", member);
        return mv;
    }

}
```

`paramMap.get("username");`
파라미터 정보는 map에 담겨있다. map에서 필요한 요청 파라미터를 조회하면 된다.

`mv.getModel().put("member", member);`
모델은 단순한 map이므로 모델에 뷰에서 필요한 `member` 객체를 담고 반환한다.

MemberListControllerV3 - 회원 목록

```
package hello.servlet.web.frontcontroller.v3.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.v3.ControllerV3;

import java.util.List;
```

```
import java.util.Map;

public class MemberListControllerV3 implements ControllerV3 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public ModelView process(Map<String, String> paramMap) {
        List<Member> members = memberRepository.findAll();

        ModelView mv = new ModelView("members");
        mv.getModel().put("members", members);

        return mv;
    }

}
```

FrontControllerServletV3

```
package hello.servlet.web.frontcontroller.v3;

import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v3.controller.MemberFormControllerV3;
import hello.servlet.web.frontcontroller.v3.controller.MemberListControllerV3;
import hello.servlet.web.frontcontroller.v3.controller.MemberSaveControllerV3;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
```

```
@WebServlet(name = "frontControllerServletV3", urlPatterns = "/front-controller/v3/*")
public class FrontControllerServletV3 extends HttpServlet {

    private Map<String, ControllerV3> controllerMap = new HashMap<>();

    public FrontControllerServletV3() {
        controllerMap.put("/front-controller/v3/members/new-form", new MemberFormControllerV3());
        controllerMap.put("/front-controller/v3/members/save", new MemberSaveControllerV3());
        controllerMap.put("/front-controller/v3/members", new MemberListControllerV3());
    }

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String requestURI = request.getRequestURI();

        ControllerV3 controller = controllerMap.get(requestURI);
        if (controller == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        Map<String, String> paramMap = createParamMap(request);
        ModelView mv = controller.process(paramMap);

        String viewName = mv.getViewName();
        MyView view = viewResolver(viewName);
        view.render(mv.getModel(), request, response);
    }

    private Map<String, String> createParamMap(HttpServletRequest request) {
        Map<String, String> paramMap = new HashMap<>();
```

```
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
        return paramMap;
    }

    private MyView viewResolver(String viewName) {
        return new MyView("/WEB-INF/views/" + viewName + ".jsp");
    }

}
```

`view.render(mv.getModel(), request, response)` 코드에서 컴파일 오류가 발생할 것이다. 다음 코드를 참고해서 MyView 객체에 필요한 메서드를 추가하자.

`createParamMap()`
HttpServletRequest에서 파라미터 정보를 꺼내서 Map으로 변환한다. 그리고 해당 Map(`paramMap`)을 컨트롤러에 전달하면서 호출한다.

뷰 리졸버

`MyView view = viewResolver(viewName)`
컨트롤러가 반환한 논리 뷰 이름을 실제 물리 뷰 경로로 변경한다. 그리고 실제 물리 경로가 있는 MyView 객체를 반환한다.

- 논리 뷰 이름: `members`
- 물리 뷰 경로: `/WEB-INF/views/members.jsp`

`view.render(mv.getModel(), request, response)`
- 뷰 객체를 통해서 HTML 화면을 렌더링 한다.
- 뷰 객체의 `render()` 는 모델 정보도 함께 받는다.
- JSP는 `request.getAttribute()` 로 데이터를 조회하기 때문에, 모델의 데이터를 꺼내서 `request.setAttribute()` 로 담아둔다.
- JSP로 포워드 해서 JSP를 렌더링 한다.

MyView

```
package hello.servlet.web.frontcontroller;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;

public class MyView {

    private String viewPath;

    public MyView(String viewPath) {
        this.viewPath = viewPath;
    }

    public void render(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }

    public void render(Map<String, Object> model, HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        modelToRequestAttribute(model, request);
        RequestDispatcher dispatcher = request.getRequestDispatcher(viewPath);
        dispatcher.forward(request, response);
    }

    private void modelToRequestAttribute(Map<String, Object> model,
HttpServletRequest request) {
        model.forEach((key, value) -> request.setAttribute(key, value));
    }
}
```

실행

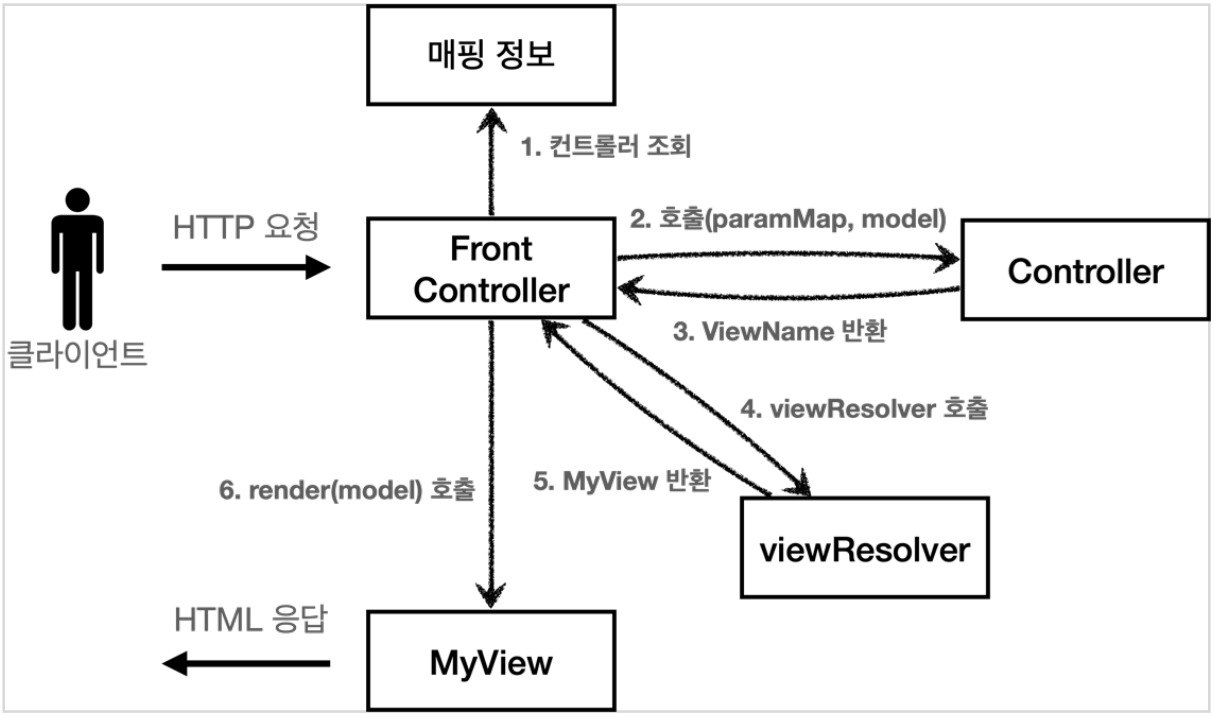
- 등록: http://localhost:8080/front-controller/v3/members/new-form
- 목록: http://localhost:8080/front-controller/v3/members

단순하고 실용적인 컨트롤러 - v4

앞서 만든 v3 컨트롤러는 서블릿 종속성을 제거하고 뷰 경로의 중복을 제거하는 등, 잘 설계된 컨트롤러이다. 그런데 실제 컨트톨러 인터페이스를 구현하는 개발자 입장에서 보면, 항상 ModelView 객체를 생성하고 반환해야 하는 부분이 조금은 번거롭다.
좋은 프레임워크는 아키텍처도 중요하지만, 그와 더불어 실제 개발하는 개발자가 단순하고 편리하게 사용할 수 있어야 한다. 소위 실용성이 있어야 한다.

이번에는 v3를 조금 변경해서 실제 구현하는 개발자들이 매우 편리하게 개발할 수 있는 v4 버전을 개발해보자.

V4 구조



- 기본적인 구조는 V3와 같다. 대신에 컨트롤러가 `ModelView`를 반환하지 않고, `ViewName`만 반환한다.

ControllerV4

```
package hello.servlet.web.frontcontroller.v4;
```

```
import java.util.Map;

public interface ControllerV4 {

    /**
     * @param paramMap
     * @param model
     * @return viewName
     */
    String process(Map<String, String> paramMap, Map<String, Object> model);
}
```

이번 버전은 인터페이스에 ModelView가 없다. model 객체는 파라미터로 전달되기 때문에 그냥 사용하면 되고, 결과로 뷰의 이름만 반환해주면 된다.

실제 구현 코드를 보자.

MemberFormControllerV4

```
package hello.servlet.web.frontcontroller.v4.controller;

import hello.servlet.web.frontcontroller.v4.ControllerV4;

import java.util.Map;

public class MemberFormControllerV4 implements ControllerV4 {

    @Override
    public String process(Map<String, String> paramMap, Map<String, Object>
model) {
        return "new-form";
    }
}
```

정말 단순하게 `new-form` 이라는 뷰의 논리 이름만 반환하면 된다.

MemberSaveControllerV4

```
package hello.servlet.web.frontcontroller.v4.controller;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.v4.ControllerV4;

import java.util.Map;

public class MemberSaveControllerV4 implements ControllerV4 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public String process(Map<String, String> paramMap, Map<String, Object>
model) {

        String username = paramMap.get("username");
        int age = Integer.parseInt(paramMap.get("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);

        model.put("member", member);

        return "save-result";
    }
}
```

`model.put("member", member)`
모델이 파라미터로 전달되기 때문에, 모델을 직접 생성하지 않아도 된다.

MemberListControllerV4

```
package hello.servlet.web.frontcontroller.v4.controller;
```

```
import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import hello.servlet.web.frontcontroller.v4.ControllerV4;

import java.util.List;
import java.util.Map;

public class MemberListControllerV4 implements ControllerV4 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @Override
    public String process(Map<String, String> paramMap, Map<String, Object>
model) {
        List<Member> members = memberRepository.findAll();
        model.put("members", members);

        return "members";
    }
}
```

FrontControllerServletV4

```
package hello.servlet.web.frontcontroller.v4;

import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v4.controller.MemberFormControllerV4;
import hello.servlet.web.frontcontroller.v4.controller.MemberListControllerV4;
import hello.servlet.web.frontcontroller.v4.controller.MemberSaveControllerV4;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
```

```
import java.util.Map;

@WebServlet(name = "frontControllerServletV4", urlPatterns = "/front-
controller/v4/*")
public class FrontControllerServletV4 extends HttpServlet {

    private Map<String, ControllerV4> controllerMap = new HashMap<>();

    public FrontControllerServletV4() {
        controllerMap.put("/front-controller/v4/members/new-form", new
MemberFormControllerV4());
        controllerMap.put("/front-controller/v4/members/save", new
MemberSaveControllerV4());
        controllerMap.put("/front-controller/v4/members", new
MemberListControllerV4());
    }

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        String requestURI = request.getRequestURI();

        ControllerV4 controller = controllerMap.get(requestURI);
        if (controller == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        Map<String, String> paramMap = createParamMap(request);
        Map<String, Object> model = new HashMap<>(); //추가

        String viewName = controller.process(paramMap, model);

        MyView view = viewResolver(viewName);
        view.render(model, request, response);
    }
```

```
    private Map<String, String> createParamMap(HttpServletRequest request) {
        Map<String, String> paramMap = new HashMap<>();
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
        return paramMap;
    }

    private MyView viewResolver(String viewName) {
        return new MyView("/WEB-INF/views/" + viewName + ".jsp");
    }

}
```

`FrontControllerServletV4`는 사실 이전 버전과 거의 동일하다.

모델 객체 전달

`Map<String, Object> model = new HashMap<>(); //추가`

모델 객체를 프론트 컨트롤러에서 생성해서 넘겨준다. 컨트롤러에서 모델 객체에 값을 담으면 여기에 그대로 담겨있게 된다.

뷰의 논리 이름을 직접 반환

```
String viewName = controller.process(paramMap, model);
MyView view = viewResolver(viewName);
```

컨트롤로가 직접 뷰의 논리 이름을 반환하므로 이 값을 사용해서 실제 물리 뷰를 찾을 수 있다.

실행

- 등록: http://localhost:8080/front-controller/v4/members/new-form
- 목록: http://localhost:8080/front-controller/v4/members

정리

이번 버전의 컨트롤러는 매우 단순하고 실용적이다. 기존 구조에서 모델을 파라미터로 넘기고, 뷰의 논리 이름을 반환한다는 작은 아이디어를 적용했을 뿐인데, 컨트롤러를 구현하는 개발자 입장에서 보면 이제 군더더기 없는 코드를 작성할 수 있다.

또한 중요한 사실은 여기까지 한번에 온 것이 아니라는 점이다. 프레임워크가 점진적으로 발전하는 과정 속에서 이런 방법도 찾을 수 있었다.

프레임워크나 공통 기능이 수고로워야 사용하는 개발자가 편리해진다.

유연한 컨트롤러1 - v5

> **참고**
> 이번 강의 영상은 가끔 지직 거리는 소리가 납니다. 양해 부탁드립니다.

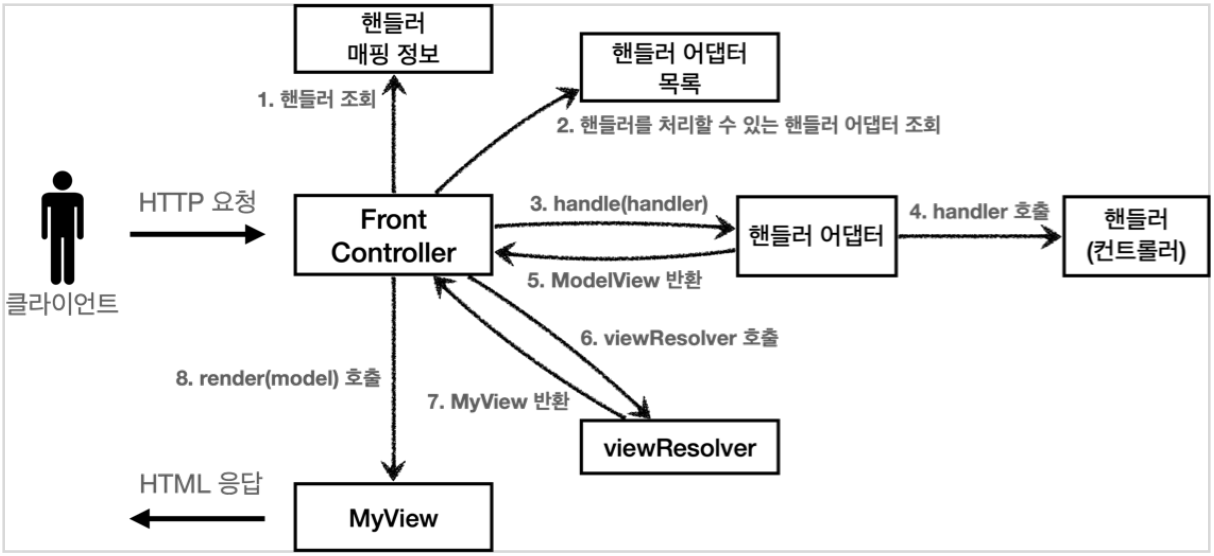
만약 어떤 개발자는 `ControllerV3` 방식으로 개발하고 싶고, 어떤 개발자는 `ControllerV4` 방식으로 개발하고 싶다면 어떻게 해야할까?

```
public interface ControllerV3 {
    ModelView process(Map<String, String> paramMap);
}
```

```
public interface ControllerV4 {
    String process(Map<String, String> paramMap, Map<String, Object> model);
}
```

어댑터 패턴
지금까지 우리가 개발한 프론트 컨트롤러는 한가지 방식의 컨트롤러 인터페이스만 사용할 수 있다.
`ControllerV3`, `ControllerV4` 는 완전히 다른 인터페이스이다. 따라서 호환이 불가능하다. 마치 v3는 110v이고, v4는 220v 전기 콘센트 같은 것이다. 이럴 때 사용하는 것이 바로 어댑터이다.
어댑터 패턴을 사용해서 프론트 컨트롤러가 다양한 방식의 컨트롤러를 처리할 수 있도록 변경해보자.

V5 구조



- **핸들러 어댑터**: 중간에 어댑터 역할을 하는 어댑터가 추가되었는데 이름이 핸들러 어댑터이다. 여기서 어댑터 역할을 해주는 덕분에 다양한 종류의 컨트롤러를 호출할 수 있다.
- **핸들러**: 컨트롤러의 이름을 더 넓은 범위인 핸들러로 변경했다. 그 이유는 이제 어댑터가 있기 때문에 꼭 컨트롤러의 개념 뿐만 아니라 어떠한 것이든 해당하는 종류의 어댑터만 있으면 다 처리할 수 있기 때문이다.

MyHandlerAdapter

어댑터는 이렇게 구현해야 한다는 어댑터용 인터페이스이다.

```
package hello.servlet.web.frontcontroller.v5;

import hello.servlet.web.frontcontroller.ModelView;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface MyHandlerAdapter {

    boolean supports(Object handler);

    ModelView handle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws ServletException, IOException;
}
```

- `boolean supports(Object handler)`
  - handler는 컨트롤러를 말한다.
  - 어댑터가 해당 컨트롤러를 처리할 수 있는지 판단하는 메서드다.
- `ModelView handle(HttpServletRequest request, HttpServletResponse response, Object handler)`
  - 어댑터는 실제 컨트롤러를 호출하고, 그 결과로 ModelView를 반환해야 한다.
  - 실제 컨트롤러가 ModelView를 반환하지 못하면, 어댑터가 ModelView를 직접 생성해서라도 반환해야 한다.
  - 이전에는 프론트 컨트롤러가 실제 컨트롤러를 호출했지만 이제는 이 어댑터를 통해서 실제 컨트롤러가 호출된다.

실제 어댑터를 구현해보자.
먼저 ControllerV3를 지원하는 어댑터를 구현하자.

ControllerV3HandlerAdapter

```
package hello.servlet.web.frontcontroller.v5.adapter;

import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.v3.ControllerV3;
import hello.servlet.web.frontcontroller.v5.MyHandlerAdapter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.HashMap;
import java.util.Map;

public class ControllerV3HandlerAdapter implements MyHandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        return (handler instanceof ControllerV3);
    }

    @Override
    public ModelView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
        ControllerV3 controller = (ControllerV3) handler;
```

```
        Map<String, String> paramMap = createParamMap(request);

        ModelView mv = controller.process(paramMap);
        return mv;
    }

    private Map<String, String> createParamMap(HttpServletRequest request) {
        Map<String, String> paramMap = new HashMap<>();
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
        return paramMap;
    }

}
```

하나씩 분석해보자.

```
public boolean supports(Object handler) {
    return (handler instanceof ControllerV3);
}
```

`ControllerV3`을 처리할 수 있는 어댑터를 뜻한다.

```
ControllerV3 controller = (ControllerV3) handler;
Map<String, String> paramMap = createParamMap(request);
ModelView mv = controller.process(paramMap);
return mv;
```

handler를 컨트롤러 V3로 변환한 다음에 V3 형식에 맞도록 호출한다.
`supports()`를 통해 `ControllerV3`만 지원하기 때문에 타입 변환은 걱정없이 실행해도 된다.
ControllerV3는 ModelView를 반환하므로 그대로 ModelView를 반환하면 된다.

FrontControllerServletV5

```
package hello.servlet.web.frontcontroller.v5;

import hello.servlet.web.frontcontroller.ModelView;
import hello.servlet.web.frontcontroller.MyView;
import hello.servlet.web.frontcontroller.v3.controller.MemberFormControllerV3;
import hello.servlet.web.frontcontroller.v3.controller.MemberListControllerV3;
import hello.servlet.web.frontcontroller.v3.controller.MemberSaveControllerV3;
import hello.servlet.web.frontcontroller.v5.adapter.ControllerV3HandlerAdapter;
import hello.servlet.web.frontcontroller.v5.adapter.ControllerV4HandlerAdapter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@WebServlet(name = "frontControllerServletV5", urlPatterns = "/front-
controller/v5/*")
public class FrontControllerServletV5 extends HttpServlet {

    private final Map<String, Object> handlerMappingMap = new HashMap<>();
    private final List<MyHandlerAdapter> handlerAdapters = new ArrayList<>();

    public FrontControllerServletV5() {
        initHandlerMappingMap();
        initHandlerAdapters();
    }

    private void initHandlerMappingMap() {
        handlerMappingMap.put("/front-controller/v5/v3/members/new-form", new
MemberFormControllerV3());
```

```
        handlerMappingMap.put("/front-controller/v5/v3/members/save", new
MemberSaveControllerV3());
        handlerMappingMap.put("/front-controller/v5/v3/members", new
MemberListControllerV3());
    }

    private void initHandlerAdapters() {
        handlerAdapters.add(new ControllerV3HandlerAdapter());
    }

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {

        Object handler = getHandler(request);
        if (handler == null) {
            response.setStatus(HttpServletResponse.SC_NOT_FOUND);
            return;
        }

        MyHandlerAdapter adapter = getHandlerAdapter(handler);
        ModelView mv = adapter.handle(request, response, handler);

        MyView view = viewResolver(mv.getViewName());
        view.render(mv.getModel(), request, response);
    }

    private Object getHandler(HttpServletRequest request) {
        String requestURI = request.getRequestURI();
        return handlerMappingMap.get(requestURI);
    }

    private MyHandlerAdapter getHandlerAdapter(Object handler) {
        for (MyHandlerAdapter adapter : handlerAdapters) {
            if (adapter.supports(handler)) {
                return adapter;
            }
        }
```

```
            throw new IllegalArgumentException("handler adapter를 찾을 수 없습니다.
handler=" + handler);
        }

        private MyView viewResolver(String viewName) {
            return new MyView("/WEB-INF/views/" + viewName + ".jsp");
        }

}
```

컨트롤러(Controller) → 핸들러(Handler)

이전에는 컨트롤러를 직접 매핑해서 사용했다. 그런데 이제는 어댑터를 사용하기 때문에, 컨트롤러 뿐만 아니라 어댑터가 지원하기만 하면, 어떤 것이라도 URL에 매핑해서 사용할 수 있다. 그래서 이름을 컨트롤러에서 더 넓은 범위의 핸들러로 변경했다.

생성자

```
public FrontControllerServletV5() {
    initHandlerMappingMap(); //핸들러 매핑 초기화
    initHandlerAdapters(); //어댑터 초기화
}
```

생성자는 핸들러 매핑과 어댑터를 초기화(등록)한다.

매핑 정보

`private final Map<String, Object> handlerMappingMap = new HashMap<>();`

매핑 정보의 값이 `ControllerV3`, `ControllerV4` 같은 인터페이스에서 아무 값이나 받을 수 있는 `Object` 로 변경되었다.

핸들러 매핑

`Object handler = getHandler(request)`

```
private Object getHandler(HttpServletRequest request) {
    String requestURI = request.getRequestURI();
    return handlerMappingMap.get(requestURI);
```

```
}
```

핸들러 매핑 정보인 `handlerMappingMap`에서 URL에 매핑된 핸들러(컨트롤러) 객체를 찾아서 반환한다.

핸들러를 처리할 수 있는 어댑터 조회

`MyHandlerAdapter adapter = getHandlerAdapter(handler)`

```
for (MyHandlerAdapter adapter : handlerAdapters) {
    if (adapter.supports(handler)) {
        return adapter;
    }
}
```

`handler`를 처리할 수 있는 어댑터를 `adapter.supports(handler)`를 통해서 찾는다.
handler가 `ControllerV3` 인터페이스를 구현했다면, `ControllerV3HandlerAdapter` 객체가 반환된다.

어댑터 호출

`ModelView mv = adapter.handle(request, response, handler);`

어댑터의 `handle(request, response, handler)` 메서드를 통해 실제 어댑터가 호출된다.
어댑터는 handler(컨트롤러)를 호출하고 그 결과를 어댑터에 맞추어 반환한다.
`ControllerV3HandlerAdapter`의 경우 어댑터의 모양과 컨트롤러의 모양이 유사해서 변환 로직이 단순하다.

실행

- 등록: http://localhost:8080/front-controller/v5/v3/members/new-form
- 목록: http://localhost:8080/front-controller/v5/v3/members

정리

지금은 V3 컨트롤러를 사용할 수 있는 어댑터와 `ControllerV3`만 들어 있어서 크게 감흥이 없을 것이다.
`ControllerV4`를 사용할 수 있도록 기능을 추가해보자.

유연한 컨트롤러2 - v5

`FrontControllerServletV5`에 `ControllerV4` 기능도 추가해보자.

```
private void initHandlerMappingMap() {
    handlerMappingMap.put("/front-controller/v5/v3/members/new-form", new
MemberFormControllerV3());
    handlerMappingMap.put("/front-controller/v5/v3/members/save", new
MemberSaveControllerV3());
    handlerMappingMap.put("/front-controller/v5/v3/members", new
MemberListControllerV3());

    //V4 추가
    handlerMappingMap.put("/front-controller/v5/v4/members/new-form", new
MemberFormControllerV4());
    handlerMappingMap.put("/front-controller/v5/v4/members/save", new
MemberSaveControllerV4());
    handlerMappingMap.put("/front-controller/v5/v4/members", new
MemberListControllerV4());
}

private void initHandlerAdapters() {
    handlerAdapters.add(new ControllerV3HandlerAdapter());
    handlerAdapters.add(new ControllerV4HandlerAdapter()); //V4 추가
}
```

핸들러 매핑(`handlerMappingMap`)에 `ControllerV4`를 사용하는 컨트롤러를 추가하고, 해당 컨트롤러를 처리할 수 있는 어댑터인 `ControllerV4HandlerAdapter`도 추가하자.

ControllerV4HandlerAdapter

```
package hello.servlet.web.frontcontroller.v5.adapter;

import hello.servlet.web.frontcontroller.ModelView;
```

```
import hello.servlet.web.frontcontroller.v4.ControllerV4;
import hello.servlet.web.frontcontroller.v5.MyHandlerAdapter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.HashMap;
import java.util.Map;

public class ControllerV4HandlerAdapter implements MyHandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        return (handler instanceof ControllerV4);
    }

    @Override
    public ModelView handle(HttpServletRequest request, HttpServletResponse
response, Object handler) {

        ControllerV4 controller = (ControllerV4) handler;

        Map<String, String> paramMap = createParamMap(request);
        Map<String, Object> model = new HashMap<>();

        String viewName = controller.process(paramMap, model);

        ModelView mv = new ModelView(viewName);
        mv.setModel(model);

        return mv;
    }

    private Map<String, String> createParamMap(HttpServletRequest request) {
        Map<String, String> paramMap = new HashMap<>();
        request.getParameterNames().asIterator()
                .forEachRemaining(paramName -> paramMap.put(paramName,
request.getParameter(paramName)));
        return paramMap;
    }
```

```
}
```

하나씩 분석해보자.

```
public boolean supports(Object handler) {
    return (handler instanceof ControllerV4);
}
```

`handler`가 `ControllerV4`인 경우에만 처리하는 어댑터이다.

실행 로직

```
ControllerV4 controller = (ControllerV4) handler;

Map<String, String> paramMap = createParamMap(request);
Map<String, Object> model = new HashMap<>();

String viewName = controller.process(paramMap, model);
```

handler를 ControllerV4로 케스팅 하고, paramMap, model을 만들어서 해당 컨트롤러를 호출한다.
그리고 viewName을 반환 받는다.

어댑터 변환

```
ModelView mv = new ModelView(viewName);
mv.setModel(model);

return mv;
```

어댑터에서 이 부분이 단순하지만 중요한 부분이다.

어댑터가 호출하는 `ControllerV4`는 뷰의 이름을 반환한다. 그런데 어댑터는 뷰의 이름이 아니라 `ModelView`를 만들어서 반환해야 한다. 여기서 어댑터가 꼭 필요한 이유가 나온다.
`ControllerV4`는 뷰의 이름을 반환했지만, 어댑터는 이것을 ModelView로 만들어서 형식을 맞추어

반환한다. 마치 110v 전기 콘센트를 220v 전기 콘센트로 변경하듯이!

어댑터와 ControllerV4

```
public interface ControllerV4 {
    String process(Map<String, String> paramMap, Map<String, Object> model);
}

public interface MyHandlerAdapter {

    ModelView handle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws ServletException, IOException;
}
```

실행

- 등록: http://localhost:8080/front-controller/v5/v4/members/new-form
- 목록: http://localhost:8080/front-controller/v5/v4/members

정리

지금까지 v1 ~ v5로 점진적으로 프레임워크를 발전시켜 왔다.
지금까지 한 작업을 정리해보자.

- **v1: 프론트 컨트롤러를 도입**
  - 기존 구조를 최대한 유지하면서 프론트 컨트롤러를 도입
- **v2: View 분류**
  - 단순 반복 되는 뷰 로직 분리
- **v3: Model 추가**
  - 서블릿 종속성 제거
  - 뷰 이름 중복 제거
- **v4: 단순하고 실용적인 컨트롤러**
  - v3와 거의 비슷
  - 구현 입장에서 ModelView를 직접 생성해서 반환하지 않도록 편리한 인터페이스 제공

- **v5: 유연한 컨트롤러**
  - 어댑터 도입
  - 어댑터를 추가해서 프레임워크를 유연하고 확장성 있게 설계

여기에 애노테이션을 사용해서 컨트롤러를 더 편리하게 발전시길 수도 있다. 만약 애노테이션을 사용해서 컨트롤러를 편리하게 사용할 수 있게 하려면 어떻게 해야할까? 바로 애노테이션을 지원하는 어댑터를 추가하면 된다!
다형성과 어댑터 덕분에 기존 구조를 유지하면서, 프레임워크의 기능을 확장할 수 있다.

스프링 MVC
여기서 더 발전시키면 좋겠지만, 스프링 MVC의 핵심 구조를 파악하는데 필요한 부분은 모두 만들어보았다. 사실은 여러분이 지금까지 작성한 코드는 스프링 MVC 프레임워크의 핵심 코드의 축약 버전이고, 구조도 거의 같다.

스프링 MVC에는 지금까지 우리가 학습한 내용과 거의 같은 구조를 가지고 있다.

5. 스프링 MVC - 구조 이해

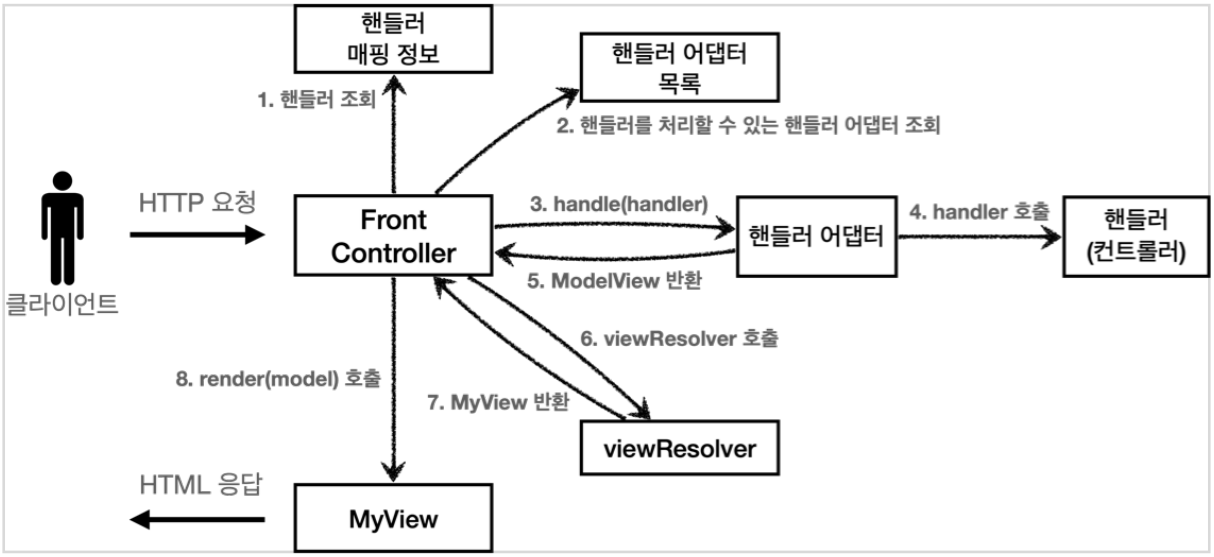
#인강/4. 스프링 MVC 1/강의#

목차

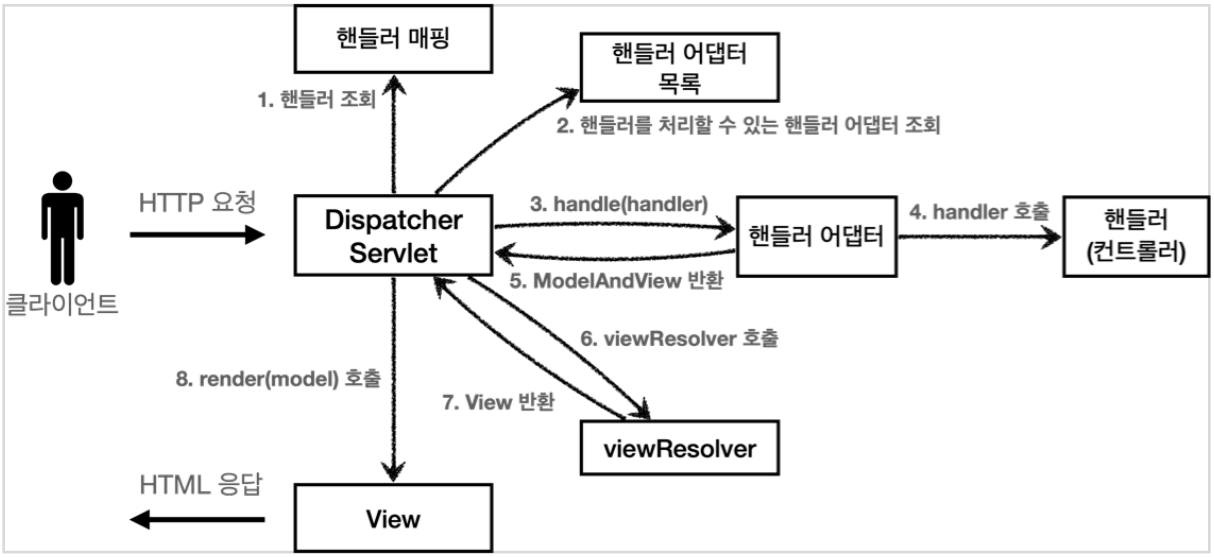
스프링 MVC 전체 구조

직접 만든 MVC 프레임워크와 스프링 MVC를 비교해보자.

직접 만든 MVC 프레임워크 구조



SpringMVC 구조



직접 만든 프레임워크 ➜ 스프링 MVC 비교

- FrontController ➜ DispatcherServlet
- handlerMappingMap ➜ HandlerMapping
- MyHandlerAdapter ➜ HandlerAdapter
- ModelView ➜ ModelAndView
- viewResolver ➜ ViewResolver
- MyView ➜ View

DispatcherServlet 구조 살펴보기

`org.springframework.web.servlet.DispatcherServlet`

스프링 MVC도 프론트 컨트롤러 패턴으로 구현되어 있다.
스프링 MVC의 프론트 컨트롤러가 바로 디스패처 서블릿(DispatcherServlet)이다.
그리고 이 디스패처 서블릿이 바로 스프링 MVC의 핵심이다.

DispacherServlet 서블릿 등록

- `DispacherServlet`도 부모 클래스에서 `HttpServlet`을 상속 받아서 사용하고, 서블릿으로 동작한다.
  - DispatcherServlet ➡ FrameworkServlet ➡ HttpServletBean ➡ HttpServlet
- 스프링 부트는 `DispacherServlet`을 서블릿으로 자동으로 등록하면서 **모든 경로**(`urlPatterns="/"`)에 대해서 매핑한다.
  - 참고: 더 자세한 경로가 우선순위가 높다. 그래서 기존에 등록한 서블릿도 함께 동작한다.

요청 흐름

- 서블릿이 호출되면 `HttpServlet`이 제공하는 `serivce()`가 호출된다.
- 스프링 MVC는 `DispatcherServlet`의 부모인 `FrameworkServlet`에서 `service()`를 오버라이드 해두었다.
- `FrameworkServlet.service()`를 시작으로 여러 메서드가 호출되면서 `DispacherServlet.doDispatch()`가 호출된다.

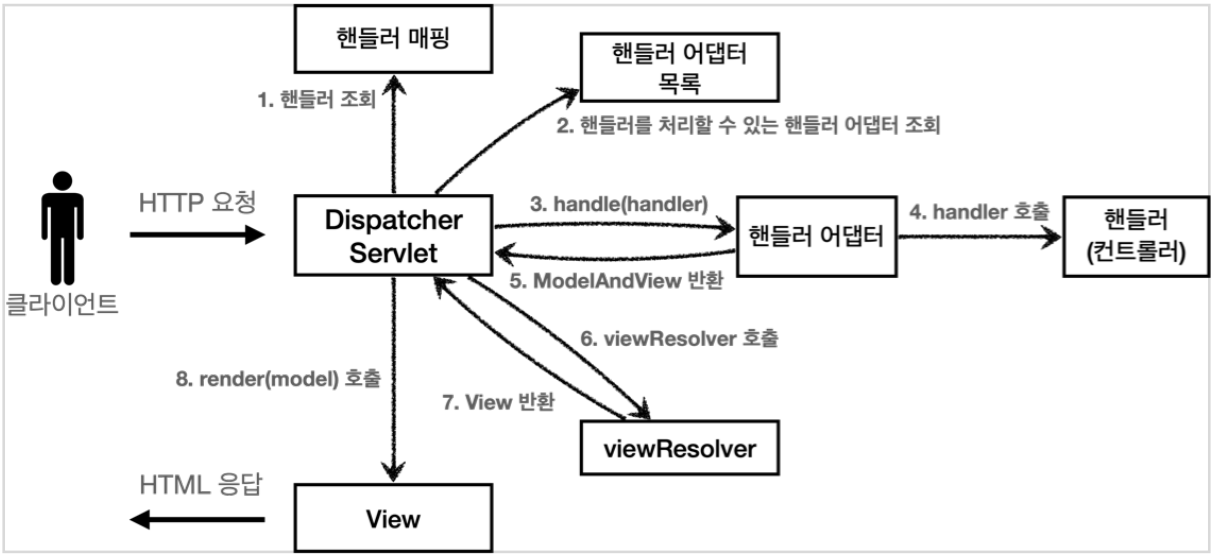
지금부터 `DispacherServlet`의 핵심인 `doDispatch()` 코드를 분석해보자. 최대한 간단히 설명하기 위해 예외처리, 인터셉터 기능은 제외했다.

`DispacherServlet.doDispatch()`

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {

  HttpServletRequest processedRequest = request;
  HandlerExecutionChain mappedHandler = null;
  ModelAndView mv = null;

  // 1. 핸들러 조회
  mappedHandler = getHandler(processedRequest);
  if (mappedHandler == null) {
```

```
            noHandlerFound(processedRequest, response);
            return;
        }

        // 2. 핸들러 어댑터 조회 - 핸들러를 처리할 수 있는 어댑터
        HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

        // 3. 핸들러 어댑터 실행 -> 4. 핸들러 어댑터를 통해 핸들러 실행 -> 5. ModelAndView 반환
        mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

        processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);

    }

    private void processDispatchResult(HttpServletRequest request,
HttpServletResponse response, HandlerExecutionChain mappedHandler, ModelAndView
mv, Exception exception) throws Exception {

        // 뷰 렌더링 호출
        render(mv, request, response);

    }

    protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response) throws Exception {

        View view;
        String viewName = mv.getViewName();

        // 6. 뷰 리졸버를 통해서 뷰 찾기, 7. View 반환
        view = resolveViewName(viewName, mv.getModelInternal(), locale, request);

        // 8. 뷰 렌더링
        view.render(mv.getModelInternal(), request, response);
    }
```

SpringMVC 구조



동작 순서

1. **핸들러 조회**: 핸들러 매핑을 통해 요청 URL에 매핑된 핸들러(컨트롤러)를 조회한다.
2. **핸들러 어댑터 조회**: 핸들러를 실행할 수 있는 핸들러 어댑터를 조회한다.
3. **핸들러 어댑터 실행**: 핸들러 어댑터를 실행한다.
4. **핸들러 실행**: 핸들러 어댑터가 실제 핸들러를 실행한다.
5. **ModelAndView 반환**: 핸들러 어댑터는 핸들러가 반환하는 정보를 ModelAndView로 **변환**해서 반환한다.
6. **viewResolver 호출**: 뷰 리졸버를 찾고 실행한다.
   - JSP의 경우: `InternalResourceViewResolver` 가 자동 등록되고, 사용된다.
7. **View 반환**: 뷰 리졸버는 뷰의 논리 이름을 물리 이름으로 바꾸고, 렌더링 역할을 담당하는 뷰 객체를 반환한다.
   - JSP의 경우 `InternalResourceView(JstlView)` 를 반환하는데, 내부에 `forward()` 로직이 있다.
8. **뷰 렌더링**: 뷰를 통해서 뷰를 렌더링 한다.

인터페이스 살펴보기

- 스프링 MVC의 큰 강점은 `DispatcherServlet` 코드의 변경 없이, 원하는 기능을 변경하거나 확장할 수 있다는 점이다. 지금까지 설명한 대부분을 확장 가능할 수 있게 인터페이스로 제공한다.
- 이 인터페이스들만 구현해서 `DispatcherServlet` 에 등록하면 여러분만의 컨트롤러를 만들 수도 있다.

주요 인터페이스 목록

- 핸들러 매핑: `org.springframework.web.servlet.HandlerMapping`
- 핸들러 어댑터: `org.springframework.web.servlet.HandlerAdapter`
- 뷰 리졸버: `org.springframework.web.servlet.ViewResolver`

- 뷰: `org.springframework.web.servlet.View`

정리
스프링 MVC는 코드 분량도 매우 많고, 복잡해서 내부 구조를 다 파악하는 것은 쉽지 않다. 사실 해당 기능을 직접 확장하거나 나만의 컨트롤러를 만드는 일은 없으므로 걱정하지 않아도 된다. 왜냐하면 스프링 MVC는 전세계 수 많은 개발자들의 요구사항에 맞추어 기능을 계속 확장왔고, 그래서 여러분이 웹 애플리케이션을 만들 때 필요로 하는 대부분의 기능이 이미 다 구현되어 있다.
그래도 이렇게 핵심 동작방식을 알아두어야 향후 문제가 발생했을 때 어떤 부분에서 문제가 발생했는지 쉽게 파악하고, 문제를 해결할 수 있다. 그리고 확장 포인트가 필요할 때, 어떤 부분을 확장해야 할지 감을 잡을 수 있다. 실제 다른 컴포넌트를 제공하거나 기능을 확장하는 부분들은 강의를 진행하면서 조금씩 설명하겠다. 지금은 전체적인 구조가 이렇게 되어 있구나 하고 이해하면 된다.

우리가 지금까지 함께 개발한 MVC 프레임워크와 유사한 구조여서 이해하기 어렵지 않았을 것이다.

핸들러 매핑과 핸들러 어댑터

핸들러 매핑과 핸들러 어댑터가 어떤 것들이 어떻게 사용되는지 알아보자.
지금은 전혀 사용하지 않지만, 과거에 주로 사용했던 스프링이 제공하는 간단한 컨트롤러로 핸들러 매핑과 어댑터를 이해해보자.

Controller 인터페이스

과거 버전 스프링 컨트롤러

`org.springframework.web.servlet.mvc.Controller`

```
public interface Controller {

  ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) throws Exception;

}
```

스프링도 처음에는 이런 딱딱한 형식의 컨트롤러를 제공했다.

> **참고**
> `Controller` 인터페이스는 `@Controller` 애노테이션과는 전혀 다르다.

간단하게 구현해보자.

OldController

```
package hello.servlet.web.springmvc.old;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component("/springmvc/old-controller")
public class OldController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        System.out.println("OldController.handleRequest");
        return null;
    }
}
```

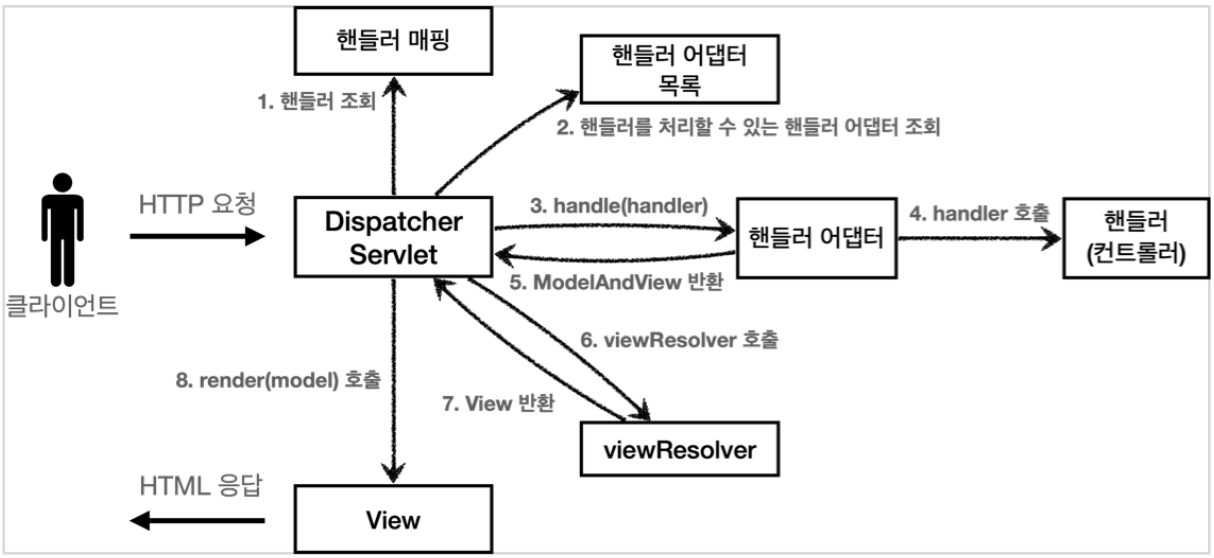
- `@Component` : 이 컨트롤러는 `/springmvc/old-controller` 라는 이름의 스프링 빈으로 등록되었다.
- **빈의 이름으로 URL을 매핑**할 것이다.

실행

- http://localhost:8080/springmvc/old-controller
- 콘솔에 `OldController.handleRequest` 이 출력되면 성공이다.

이 컨트롤러는 어떻게 호출될 수 있을까?

스프링 MVC 구조



이 컨트롤러가 호출되려면 다음 2가지가 필요하다.

- **HandlerMapping(핸들러 매핑)**
  - 핸들러 매핑에서 이 컨트롤러를 찾을 수 있어야 한다.
  - 예) **스프링 빈의 이름으로 핸들러를 찾을 수 있는 핸들러 매핑**이 필요하다.
- **HandlerAdapter(핸들러 어댑터)**
  - 핸들러 매핑을 통해서 찾은 핸들러를 실행할 수 있는 핸들러 어댑터가 필요하다.
  - 예) `Controller` 인터페이스를 실행할 수 있는 핸들러 어댑터를 찾고 실행해야 한다.

스프링은 이미 필요한 핸들러 매핑과 핸들러 어댑터를 대부분 구현해두었다. 개발자가 직접 핸들러 매핑과 핸들러 어댑터를 만드는 일은 거의 없다.

스프링 부트가 자동 등록하는 핸들러 매핑과 핸들러 어댑터
(실제로는 더 많지만, 중요한 부분 위주로 설명하기 위해 일부 생략)

HandlerMapping

```
0 = RequestMappingHandlerMapping    : 애노테이션 기반의 컨트롤러인 @RequestMapping에서 사용
1 = BeanNameUrlHandlerMapping       : 스프링 빈의 이름으로 핸들러를 찾는다.
```

HandlerAdapter

```
0 = RequestMappingHandlerAdapter    : 애노테이션 기반의 컨트롤러인 @RequestMapping에서 사용
```

```
1 = HttpRequestHandlerAdapter      : HttpRequestHandler 처리
2 = SimpleControllerHandlerAdapter : Controller 인터페이스(애노테이션X, 과거에 사용)
처리
```

핸들러 매핑도, 핸들러 어댑터도 모두 순서대로 찾고 만약 없으면 다음 순서로 넘어간다.

1. 핸들러 매핑으로 핸들러 조회

1. `HandlerMapping`을 순서대로 실행해서, 핸들러를 찾는다.
2. 이 경우 빈 이름으로 핸들러를 찾아야 하기 때문에 이름 그대로 빈 이름으로 핸들러를 찾아주는 `BeanNameUrlHandlerMapping`가 실행에 성공하고 핸들러인 `OldController`를 반환한다.

2. 핸들러 어댑터 조회

1. `HandlerAdapter`의 `supports()`를 순서대로 호출한다.
2. `SimpleControllerHandlerAdapter`가 `Controller` 인터페이스를 지원하므로 대상이 된다.

3. 핸들러 어댑터 실행

1. 디스패처 서블릿이 조회한 `SimpleControllerHandlerAdapter`를 실행하면서 핸들러 정보도 함께 넘겨준다.
2. `SimpleControllerHandlerAdapter`는 핸들러인 `OldController`를 내부에서 실행하고, 그 결과를 반환한다.

정리 - OldController 핸들러매핑, 어댑터

`OldController`를 실행하면서 사용된 객체는 다음과 같다.

`HandlerMapping = BeanNameUrlHandlerMapping`

`HandlerAdapter = SimpleControllerHandlerAdapter`

HttpRequestHandler

핸들러 매핑과, 어댑터를 더 잘 이해하기 위해 Controller 인터페이스가 아닌 다른 핸들러를 알아보자.

`HttpRequestHandler` 핸들러(컨트롤러)는 **서블릿과 가장 유사한 형태**의 핸들러이다.

HttpRequestHandler

```
public interface HttpRequestHandler {
```

```
    void handleRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException;

}
```

간단하게 구현해보자.

MyHttpRequestHandler

```
package hello.servlet.web.springmvc.old;

import org.springframework.stereotype.Component;
import org.springframework.web.HttpRequestHandler;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Component("/springmvc/request-handler")
public class MyHttpRequestHandler implements HttpRequestHandler {

    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        System.out.println("MyHttpRequestHandler.handleRequest");
    }
}
```

실행

- http://localhost:8080/springmvc/request-handler
- 웹 브라우저에 빈 화면이 나오고, 콘솔에 `MyHttpRequestHandler.handleRequest` 가 출력되면 성공이다.

1. 핸들러 매핑으로 핸들러 조회

1. `HandlerMapping` 을 순서대로 실행해서, 핸들러를 찾는다.

2. 이 경우 빈 이름으로 핸들러를 찾아야 하기 때문에 이름 그대로 빈 이름으로 핸들러를 찾아주는 `BeanNameUrlHandlerMapping`가 실행에 성공하고 핸들러인 `MyHttpRequestHandler`를 반환한다.

2. 핸들러 어댑터 조회

1. `HandlerAdapter`의 `supports()`를 순서대로 호출한다.
2. `HttpRequestHandlerAdapter`가 `HttpRequestHandler` 인터페이스를 지원하므로 대상이 된다.

3. 핸들러 어댑터 실행

1. 디스패처 서블릿이 조회한 `HttpRequestHandlerAdapter`를 실행하면서 핸들러 정보도 함께 넘겨준다.
2. `HttpRequestHandlerAdapter`는 핸들러인 `MyHttpRequestHandler`를 내부에서 실행하고, 그 결과를 반환한다.

정리 - MyHttpRequestHandler 핸들러매핑, 어댑터

`MyHttpRequestHandler`를 실행하면서 사용된 객체는 다음과 같다.

`HandlerMapping = BeanNameUrlHandlerMapping`

`HandlerAdapter = HttpRequestHandlerAdapter`

@RequestMapping

조금 뒤에서 설명하겠지만, 가장 우선순위가 높은 핸들러 매핑과 핸들러 어댑터는 `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`이다.
`@RequestMapping`의 앞글자를 따서 만든 이름인데, 이것이 바로 지금 스프링에서 주로 사용하는 애노테이션 기반의 컨트롤러를 지원하는 매핑과 어댑터이다. 실무에서는 99.9% 이 방식의 컨트롤러를 사용한다.

뷰 리졸버

이번에는 뷰 리졸버에 대해서 자세히 알아보자.

OldController - View 조회할 수 있도록 변경

```
package hello.servlet.web.springmvc.old;

import org.springframework.stereotype.Component;
import org.springframework.web.servlet.ModelAndView;
```

```
import org.springframework.web.servlet.mvc.Controller;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Component("/springmvc/old-controller")
public class OldController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception {
        System.out.println("OldController.handleRequest");
        return new ModelAndView("new-form");
    }
}
```

View를 사용할 수 있도록 다음 코드를 추가했다.
`return new ModelAndView("new-form");`

실행

- http://localhost:8080/springmvc/old-controller
- 웹 브라우저에 `Whitelabel Error Page`가 나오고, 콘솔에 `OldController.handleRequest`이 출력될 것이다.

실행해보면 컨트롤러를 정상 호출되지만, Whitelabel Error Page 오류가 발생한다.

`application.properties`에 다음 코드를 추가하자

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

뷰 리졸버 - InternalResourceViewResolver
스프링 부트는 `InternalResourceViewResolver`라는 뷰 리졸버를 자동으로 등록하는데, 이때 `application.properties`에 등록한 `spring.mvc.view.prefix`, `spring.mvc.view.suffix` 설정 정보를 사용해서 등록한다.

참고로 권장하지는 않지만 설정 없이 다음과 같이 전체 경로를 주어도 동작하기는 한다.

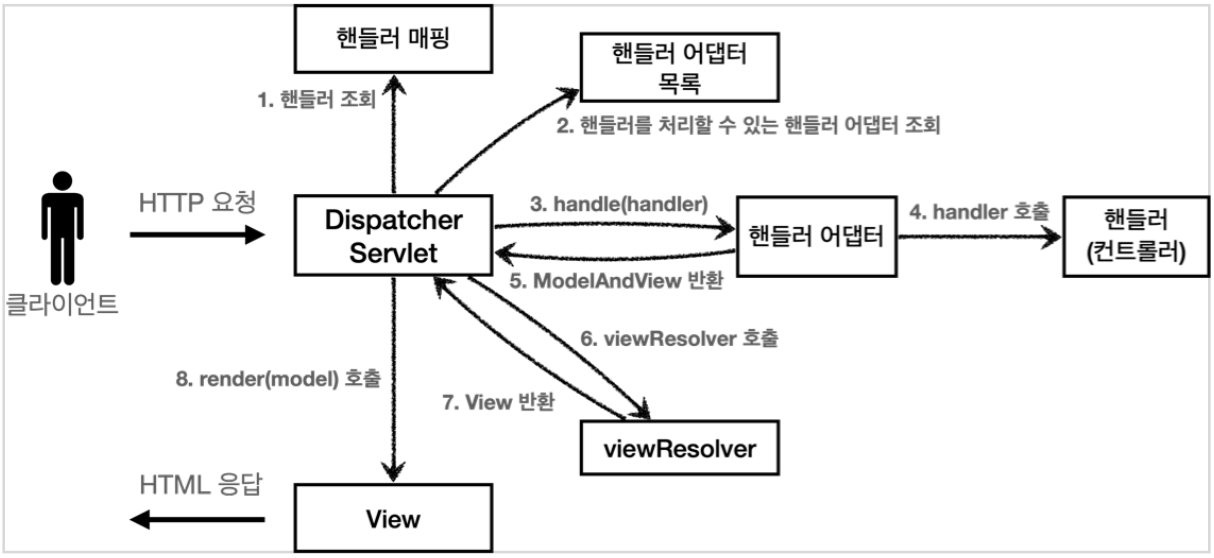
```
return new ModelAndView("/WEB-INF/views/new-form.jsp");
```

실행

- http://localhost:8080/springmvc/old-controller
- 등록 폼이 정상 출력되는 것을 확인할 수 있다. 물론 저장 기능을 개발하지 않았으므로 폼만 출력되고, 더 진행하면 오류가 발생한다.

뷰 리졸버 동작 방식

스프링 MVC 구조



스프링 부트가 자동 등록하는 뷰 리졸버

(실제로는 더 많지만, 중요한 부분 위주로 설명하기 위해 일부 생략)

```
1 = BeanNameViewResolver          : 빈 이름으로 뷰를 찾아서 반환한다. (예: 엑셀 파일 생성 기능에 사용)
2 = InternalResourceViewResolver : JSP를 처리할 수 있는 뷰를 반환한다.
```

1. 핸들러 어댑터 호출

핸들러 어댑터를 통해 `new-form`이라는 논리 뷰 이름을 획득한다.

2. ViewResolver 호출

- `new-form`이라는 뷰 이름으로 viewResolver를 순서대로 호출한다.
- `BeanNameViewResolver`는 `new-form`이라는 이름의 스프링 빈으로 등록된 뷰를 찾아야 하는데 없다.
- `InternalResourceViewResolver`가 호출된다.

3. InternalResourceViewResolver

이 뷰 리졸버는 `InternalResourceView`를 반환한다.

4. 뷰 - InternalResourceView

`InternalResourceView`는 JSP처럼 포워드 `forward()`를 호출해서 처리할 수 있는 경우에 사용한다.

5. view.render()

`view.render()`가 호출되고 `InternalResourceView`는 `forward()`를 사용해서 JSP를 실행한다.

> **참고**
> `InternalResourceViewResolver`는 만약 JSTL 라이브러리가 있으면 `InternalResourceView`를 상속받은 `JstlView`를 반환한다. `JstlView`는 JSTL 태그 사용시 약간의 부가 기능이 추가된다.

> **참고**
> 다른 뷰는 실제 뷰를 렌더링하지만, JSP의 경우 `forward()` 통해서 해당 JSP로 이동(실행)해야 렌더링이 된다. JSP를 제외한 나머지 뷰 템플릿들은 `forward()` 과정 없이 바로 렌더링 된다.

> **참고**
> Thymeleaf 뷰 템플릿을 사용하면 `ThymeleafViewResolver`를 등록해야 한다. 최근에는 라이브러리만 추가하면 스프링 부트가 이런 작업도 모두 자동화해준다.

이제 본격적으로 스프링 MVC를 시작해보자.

스프링 MVC - 시작하기

스프링이 제공하는 컨트롤러는 애노테이션 기반으로 동작해서, 매우 유연하고 실용적이다. 과거에는 자바 언어에 애노테이션이 없기도 했고, 스프링도 처음부터 이런 유연한 컨트롤러를 제공한 것은 아니다.

@RequestMapping

스프링은 애노테이션을 활용한 매우 유연하고, 실용적인 컨트롤러를 만들었는데 이것이 바로 `@RequestMapping` 애노테이션을 사용하는 컨트롤러이다. 다들 한번쯤 사용해보았을 것이다.
여담이지만 과거에는 스프링 프레임워크가 MVC 부분이 약해서 스프링을 사용하더라도 MVC 웹 기술은 스트럿츠 같은 다른 프레임워크를 사용했었다. 그런데 `@RequestMapping` 기반의 애노테이션 컨트롤러가 등장하면서, MVC 부분도 스프링의 완승으로 끝이 났다.

`@RequestMapping`

- `RequestMappingHandlerMapping`
- `RequestMappingHandlerAdapter`

앞서 보았듯이 가장 우선순위가 높은 핸들러 매핑과 핸들러 어댑터는 `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter` 이다.
`@RequestMapping` 의 앞글자를 따서 만든 이름인데, 이것이 바로 지금 스프링에서 주로 사용하는 애노테이션 기반의 컨트롤러를 지원하는 핸들러 매핑과 어댑터이다. **실무에서는 99.9% 이 방식의 컨트롤러를 사용**한다.

그럼 이제 본격적으로 애노테이션 기반의 컨트롤러를 사용해보자.
지금까지 만들었던 프레임워크에서 사용했던 컨트롤러를 `@RequestMapping` 기반의 스프링 MVC 컨트롤러 변경해보자.

SpringMemberFormControllerV1 - 회원 등록 폼

```
package hello.servlet.web.springmvc.v1;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class SpringMemberFormControllerV1 {

    @RequestMapping("/springmvc/v1/members/new-form")
    public ModelAndView process() {
        return new ModelAndView("new-form");
    }
}
```

- `@Controller` :
  - 스프링이 자동으로 스프링 빈으로 등록한다. (내부에 `@Component` 애노테이션이 있어서 컴포넌트 스캔의 대상이 됨)
  - 스프링 MVC에서 애노테이션 기반 컨트롤러로 인식한다.
- `@RequestMapping` : 요청 정보를 매핑한다. 해당 URL이 호출되면 이 메서드가 호출된다. 애노테이션을 기반으로 동작하기 때문에, 메서드의 이름은 임의로 지으면 된다.
- `ModelAndView` : 모델과 뷰 정보를 담아서 반환하면 된다.

`RequestMappingHandlerMapping` 은 스프링 빈 중에서 `@RequestMapping` 또는 `@Controller` 가 클래스 레벨에 붙어 있는 경우에 매핑 정보로 인식한다.
따라서 다음 코드도 동일하게 동작한다.

```
@Component //컴포넌트 스캔을 통해 스프링 빈으로 등록
@RequestMapping
public class SpringMemberFormControllerV1 {

    @RequestMapping("/springmvc/v1/members/new-form")
    public ModelAndView process() {
        return new ModelAndView("new-form");
    }
}
```

물론 컴포넌스 스캔 없이 다음과 같이 스프링 빈으로 직접 등록해도 동작한다.

```
@RequestMapping
public class SpringMemberFormControllerV1 {

    @RequestMapping("/springmvc/v1/members/new-form")
    public ModelAndView process() {
        return new ModelAndView("new-form");
    }
}
```

ServletApplication

```
//스프링 빈 직접 등록
@Bean
TestController testController() {
  return new TestController();
}
```

실행

- http://localhost:8080/springmvc/v1/members/new-form
- 폼을 확인할 수 있다. 나머지 코드도 추가해보자.

SpringMemberSaveControllerV1 - 회원 저장

```
package hello.servlet.web.springmvc.v1;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Controller
public class SpringMemberSaveControllerV1 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @RequestMapping("/springmvc/v1/members/save")
    public ModelAndView process(HttpServletRequest request, HttpServletResponse
response) {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));

        Member member = new Member(username, age);
```

```
        System.out.println("member = " + member);
        memberRepository.save(member);

        ModelAndView mv = new ModelAndView("save-result");
        mv.addObject("member", member);
        return mv;

    }
}
```

- `mv.addObject("member", member)`
  - 스프링이 제공하는 `ModelAndView`를 통해 Model 데이터를 추가할 때는 `addObject()`를 사용하면 된다. 이 데이터는 이후 뷰를 렌더링 할 때 사용된다.

SpringMemberListControllerV1 - 회원 목록

```
package hello.servlet.web.springmvc.v1;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import java.util.List;

@Controller
public class SpringMemberListControllerV1 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @RequestMapping("/springmvc/v1/members")
    public ModelAndView process() {

        List<Member> members = memberRepository.findAll();

        ModelAndView mv = new ModelAndView("members");
        mv.addObject("members", members);
```

```
        return mv;
    }
}
```

실행

- 등록: http://localhost:8080/springmvc/v1/members/new-form
- 목록: http://localhost:8080/springmvc/v1/members

스프링 MVC - 컨트롤러 통합

`@RequestMapping`을 잘 보면 클래스 단위가 아니라 메서드 단위에 적용된 것을 확인할 수 있다. 따라서 컨트롤러 클래스를 유연하게 하나로 통합할 수 있다.

SpringMemberControllerV2

```
package hello.servlet.web.springmvc.v2;

import hello.servlet.domain.member.Member;
import hello.servlet.domain.member.MemberRepository;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.List;

/**
 * 클래스 단위 -> 메서드 단위
 * @RequestMapping 클래스 레벨과 메서드 레벨 조합
 */
@Controller
@RequestMapping("/springmvc/v2/members")
public class SpringMemberControllerV2 {
```

```
    private MemberRepository memberRepository = MemberRepository.getInstance();

    @RequestMapping("/new-form")
    public ModelAndView newForm() {
        return new ModelAndView("new-form");
    }

    @RequestMapping("/save")
    public ModelAndView save(HttpServletRequest request, HttpServletResponse
response) {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));

        Member member = new Member(username, age);
        memberRepository.save(member);

        ModelAndView mav = new ModelAndView("save-result");
        mav.addObject("member", member);
        return mav;
    }

    @RequestMapping
    public ModelAndView members() {

        List<Member> members = memberRepository.findAll();

        ModelAndView mav = new ModelAndView("members");
        mav.addObject("members", members);
        return mav;
    }

}
```

조합

컨트롤러 클래스를 통합하는 것을 넘어서 조합도 가능하다.

다음 코드는 `/springmvc/v2/members` 라는 부분에 중복이 있다.

- `@RequestMapping("/springmvc/v2/members/new-form")`
- `@RequestMapping("/springmvc/v2/members")`
- `@RequestMapping("/springmvc/v2/members/save")`

물론 이렇게 사용해도 되지만, 컨트롤러를 통합한 예제 코드를 보면 중복을 어떻게 제거했는지 확인할 수 있다.
클래스 레벨에 다음과 같이 `@RequestMapping`을 두면 메서드 레벨과 조합이 된다.

```
@Controller
@RequestMapping("/springmvc/v2/members")
public class SpringMemberControllerV2 {}
```

조합 결과

- 클래스 레벨 `@RequestMapping("/springmvc/v2/members")`
  - 메서드 레벨 `@RequestMapping("/new-form")` ➡ `/springmvc/v2/members/new-form`
  - 메서드 레벨 `@RequestMapping("/save")` ➡ `/springmvc/v2/members/save`
  - 메서드 레벨 `@RequestMapping` ➡ `/springmvc/v2/members`

실행

- 등록: http://localhost:8080/springmvc/v2/members/new-form
- 목록: http://localhost:8080/springmvc/v2/members

스프링 MVC - 실용적인 방식

MVC 프레임워크 만들기에서 v3은 ModelView를 개발자가 직접 생성해서 반환했기 때문에, 불편했던 기억이 날 것이다. 물론 v4를 만들면서 실용적으로 개선한 기억도 날 것이다.

스프링 MVC는 개발자가 편리하게 개발할 수 있도록 수 많은 편의 기능을 제공한다.
실무에서는 지금부터 설명하는 방식을 주로 사용한다.

SpringMemberControllerV3

```
package hello.servlet.web.springmvc.v3;

import hello.servlet.domain.member.Member;
```

```java
import hello.servlet.domain.member.MemberRepository;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import java.util.List;

/**
 * v3
 * Model 도입
 * ViewName 직접 반환
 * @RequestParam 사용
 * @RequestMapping -> @GetMapping, @PostMapping
 */
@Controller
@RequestMapping("/springmvc/v3/members")
public class SpringMemberControllerV3 {

    private MemberRepository memberRepository = MemberRepository.getInstance();

    @GetMapping("/new-form")
    public String newForm() {
        return "new-form";
    }

    @PostMapping("/save")
    public String save(
            @RequestParam("username") String username,
            @RequestParam("age") int age,
            Model model) {

        Member member = new Member(username, age);
        memberRepository.save(member);

        model.addAttribute("member", member);
        return "save-result";
```

```
    }

    @GetMapping
    public String members(Model model) {
        List<Member> members = memberRepository.findAll();
        model.addAttribute("members", members);
        return "members";
    }

}
```

Model 파라미터

`save()`, `members()`를 보면 Model을 파라미터로 받는 것을 확인할 수 있다. 스프링 MVC도 이런 편의 기능을 제공한다.

ViewName 직접 반환

뷰의 논리 이름을 반환할 수 있다.

@RequestParam 사용

스프링은 HTTP 요청 파라미터를 `@RequestParam`으로 받을 수 있다.
`@RequestParam("username")`은 `request.getParameter("username")`와 거의 같은 코드라 생각하면 된다.
물론 GET 쿼리 파라미터, POST Form 방식을 모두 지원한다.

@RequestMapping → @GetMapping, @PostMapping

`@RequestMapping`은 URL만 매칭하는 것이 아니라, HTTP Method도 함께 구분할 수 있다.
예를 들어서 URL이 `/new-form`이고, HTTP Method가 GET인 경우를 모두 만족하는 매핑을 하려면 다음과 같이 처리하면 된다.

```
@RequestMapping(value = "/new-form", method = RequestMethod.GET)
```

이것을 `@GetMapping`, `@PostMapping`으로 더 편리하게 사용할 수 있다.
참고로 Get, Post, Put, Delete, Patch 모두 애노테이션이 준비되어 있다.

`@GetMapping` 코드를 열어서 `@RequestMapping` 애노테이션을 내부에 가지고 있는 모습을 확인하자.

실행

- 등록: http://localhost:8080/springmvc/v3/members/new-form
- 목록: http://localhost:8080/springmvc/v3/members

정리

6. 스프링 MVC - 기본 기능

#인강/4. 스프링 MVC 1/강의#

목차

프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.4.x
- Project Metadata
  - Group: hello
  - Artifact: springmvc
  - Name: springmvc
  - Package name: hello.springmvc
  - Packaging: **Jar (주의!)**
  - Java: 11

- Dependencies: **Spring Web**, **Thymeleaf**, **Lombok**

> **주의!**
> Packaging는 War가 아니라 **Jar를 선택**해주세요. JSP를 사용하지 않기 때문에 Jar를 사용하는 것이 좋습니다. 앞으로 스프링 부트를 사용하면 이 방식을 주로 사용하게 됩니다.
> Jar를 사용하면 항상 내장 서버(톰캣등)을 사용하고, `webapp` 경로도 사용하지 않습니다. 내장 서버 사용에 최적화 되어 있는 기능입니다. 최근에는 주로 이 방식을 사용합니다.
> War를 사용하면 내장 서버도 사용가능 하지만, 주로 외부 서버에 배포하는 목적으로 사용합니다.

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.4.3'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}
```

```
repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
  implementation 'org.springframework.boot:spring-boot-starter-web'
  compileOnly 'org.projectlombok:lombok'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
  useJUnitPlatform()
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`SpringmvcApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

Welcome 페이지 만들기

이번 장에서 학습할 내용을 편리하게 참고하기 위해 Welcome 페이지를 만들자.

스프링 부트에 `Jar` 를 사용하면 `/resources/static/index.hml` 위치에 `index.html` 파일을 두면 Welcome 페이지로 처리해준다. (스프링 부트가 지원하는 정적 컨텐츠 위치에 `/index.html` 이 있으면 된다.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
```

```
<body>
<ul>
    <li>로그 출력
        <ul>
            <li><a href="/log-test">로그 테스트</a></li>
        </ul>
    </li>
    <!-- -->
    <li>요청 매핑
        <ul>
            <li><a href="/hello-basic">hello-basic</a></li>
            <li><a href="/mapping-get-v1">HTTP 메서드 매핑</a></li>
            <li><a href="/mapping-get-v2">HTTP 메서드 매핑 축약</a></li>
            <li><a href="/mapping/userA">경로 변수</a></li>
            <li><a href="/mapping/users/userA/orders/100">경로 변수 다중</a></li>
            <li><a href="/mapping-param?mode=debug">특정 파라미터 조건 매핑</a></li>
            <li><a href="/mapping-header">특정 헤더 조건 매핑(POST MAN 필요)</a></
li>
            <li><a href="/mapping-consume">미디어 타입 조건 매핑 Content-Type(POST
MAN 필요)</a></li>
            <li><a href="/mapping-produce">미디어 타입 조건 매핑 Accept(POST MAN
필요)</a></li>
        </ul>
    </li>
    <li>요청 매핑 - API 예시
        <ul>
            <li>POST MAN 필요</li>
        </ul>
    </li>
    <li>HTTP 요청 기본
        <ul>
            <li><a href="/headers">기본, 헤더 조회</a></li>
        </ul>
    </li>
    <li>HTTP 요청 파라미터
        <ul>
            <li><a href="/request-param-v1?username=hello&age=20">요청 파라미터
v1</a></li>
            <li><a href="/request-param-v2?username=hello&age=20">요청 파라미터
```

```
v2</a></li>
            <li><a href="/request-param-v3?username=hello&age=20">요청 파라미터
v3</a></li>
            <li><a href="/request-param-v4?username=hello&age=20">요청 파라미터
v4</a></li>
            <li><a href="/request-param-required?username=hello&age=20">요청
파라미터 필수</a></li>
            <li><a href="/request-param-default?username=hello&age=20">요청
파라미터 기본 값</a></li>
            <li><a href="/request-param-map?username=hello&age=20">요청 파라미터
MAP</a></li>
            <li><a href="/model-attribute-v1?username=hello&age=20">요청 파라미터
@ModelAttribute v1</a></li>
            <li><a href="/model-attribute-v2?username=hello&age=20">요청 파라미터
@ModelAttribute v2</a></li>
        </ul>
    </li>
    <li>HTTP 요청 메시지
        <ul>
            <li>POST MAN</li>
        </ul>
    </li>
    <li>HTTP 응답 - 정적 리소스, 뷰 템플릿
        <ul>
            <li><a href="/basic/hello-form.html">정적 리소스</a></li>
            <li><a href="/response-view-v1">뷰 템플릿 v1</a></li>
            <li><a href="/response-view-v2">뷰 템플릿 v2</a></li>
        </ul>
    </li>
    <li>HTTP 응답 - HTTP API, 메시지 바디에 직접 입력
        <ul>
            <li><a href="/response-body-string-v1">HTTP API String v1</a></li>
            <li><a href="/response-body-string-v2">HTTP API String v2</a></li>
            <li><a href="/response-body-string-v3">HTTP API String v3</a></li>
            <li><a href="/response-body-json-v1">HTTP API Json v1</a></li>
            <li><a href="/response-body-json-v2">HTTP API Json v2</a></li>
        </ul>
    </li>
</ul>
```

```
</body>
</html>
```

> **참고**
> 스프링 부트 Welcome 페이지 지원
> https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-spring-mvc-welcome-page

로깅 간단히 알아보기

앞으로 로그를 사용할 것이기 때문에, 이번시간에는 로그에 대해서 간단히 알아보자.

운영 시스템에서는 `System.out.println()` 같은 시스템 콘솔을 사용해서 필요한 정보를 출력하지 않고, 별도의 로깅 라이브러리를 사용해서 로그를 출력한다.
참고로 로그 관련 라이브러리도 많고, 깊게 들어가면 끝이 없기 때문에, 여기서는 최소한의 사용 방법만 알아본다.

로깅 라이브러리
스프링 부트 라이브러리를 사용하면 스프링 부트 로깅 라이브러리(`spring-boot-starter-logging`)가 함께 포함된다.
스프링 부트 로깅 라이브러리는 기본으로 다음 로깅 라이브러리를 사용한다.

- SLF4J - http://www.slf4j.org
- Logback - http://logback.qos.ch

로그 라이브러리는 Logback, Log4J, Log4J2 등등 수 많은 라이브러리가 있는데, 그것을 통합해서 인터페이스로 제공하는 것이 바로 SLF4J 라이브러리다.
쉽게 이야기해서 SLF4J는 인터페이스이고, 그 구현체로 Logback 같은 로그 라이브러리를 선택하면 된다.
실무에서는 스프링 부트가 기본으로 제공하는 Logback을 대부분 사용한다.

로그 선언

- `private Logger log = LoggerFactory.getLogger(getClass());`
- `private static final Logger log = LoggerFactory.getLogger(Xxx.class)`

- `@Slf4j` : 롬복 사용 가능

로그 호출

- `log.info("hello")`
- `System.out.println("hello")`

시스템 콘솔로 직접 출력하는 것 보다 로그를 사용하면 다음과 같은 장점이 있다. 실무에서는 항상 로그를 사용해야 한다.

LogTestController

```
package hello.springmvc.basic;

import lombok.extern.slf4j.Slf4j;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

//@Slf4j
@RestController
public class LogTestController {

    private final Logger log = LoggerFactory.getLogger(getClass());

    @RequestMapping("/log-test")
    public String logTest() {
        String name = "Spring";

        log.trace("trace log={}", name);
        log.debug("debug log={}", name);
        log.info(" info log={}", name);
        log.warn(" warn log={}", name);
        log.error("error log={}", name);

        //로그를 사용하지 않아도 a+b 계산 로직이 먼저 실행됨, 이런 방식으로 사용하면 X
        log.debug("String concat log=" + name);
        return "ok";
    }
}
```

실행

- http://localhost:8080/log-test

매핑 정보

- `@RestController`
  - `@Controller`는 반환 값이 `String`이면 뷰 이름으로 인식된다. 그래서 **뷰를 찾고 뷰가 랜더링** 된다.
  - `@RestController`는 반환 값으로 뷰를 찾는 것이 아니라, **HTTP 메시지 바디에 바로 입력**한다. 따라서 실행 결과로 ok 메세지를 받을 수 있다. `@ResponseBody`와 관련이 있는데, 뒤에서 더 자세히 설명한다.

테스트

- 로그가 출력되는 포멧 확인
  - 시간, 로그 레벨, 프로세스 ID, 쓰레드 명, 클래스명, 로그 메시지
- 로그 레벨 설정을 변경해서 출력 결과를 보자.
  - LEVEL: `TRACE > DEBUG > INFO > WARN > ERROR`
  - 개발 서버는 debug 출력
  - 운영 서버는 info 출력
- `@Slf4j`로 변경

로그 레벨 설정

`application.properties`

```
#전체 로그 레벨 설정(기본 info)
logging.level.root=info

#hello.springmvc 패키지와 그 하위 로그 레벨 설정
logging.level.hello.springmvc=debug
```

올바른 로그 사용법

- `log.debug("data="+data)`
  - 로그 출력 레벨을 info로 설정해도 해당 코드에 있는 "data="+data가 실제 실행이 되어 버린다. 결과적으로 문자 더하기 연산이 발생한다.
- `log.debug("data={}", data)`

- 로그 출력 레벨을 info로 설정하면 아무일도 발생하지 않는다. 따라서 앞과 같은 의미없는 연산이 발생하지 않는다.

로그 사용시 장점

- 쓰레드 정보, 클래스 이름 같은 부가 정보를 함께 볼 수 있고, 출력 모양을 조정할 수 있다.
- 로그 레벨에 따라 개발 서버에서는 모든 로그를 출력하고, 운영서버에서는 출력하지 않는 등 로그를 상황에 맞게 조절할 수 있다.
- 시스템 아웃 콘솔에만 출력하는 것이 아니라, 파일이나 네트워크 등, 로그를 별도의 위치에 남길 수 있다. 특히 파일로 남길 때는 일별, 특정 용량에 따라 로그를 분할하는 것도 가능하다.
- 성능도 일반 System.out보다 좋다. (내부 버퍼링, 멀티 쓰레드 등등) 그래서 실무에서는 꼭 로그를 사용해야 한다.

더 공부하실 분

- 로그에 대해서 더 자세한 내용은 slf4j, logback을 검색해보자.
  - SLF4J - http://www.slf4j.org
  - Logback - http://logback.qos.ch
- 스프링 부트가 제공하는 로그 기능은 다음을 참고하자.
  - https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-logging

요청 매핑

MappingController

```
package hello.springmvc.basic.requestmapping;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.*;

@RestController
public class MappingController {

    private Logger log = LoggerFactory.getLogger(getClass());
```

```
    /**
     * 기본 요청
     * 둘다 허용 /hello-basic, /hello-basic/
     * HTTP 메서드 모두 허용 GET, HEAD, POST, PUT, PATCH, DELETE
     */
    @RequestMapping("/hello-basic")
    public String helloBasic() {
        log.info("helloBasic");
        return "ok";
    }

}
```

매핑 정보(한번 더)

- `@RestController`
  - `@Controller` 는 반환 값이 `String` 이면 뷰 이름으로 인식된다. 그래서 **뷰를 찾고 뷰가 랜더링** 된다.
  - `@RestController` 는 반환 값으로 뷰를 찾는 것이 아니라, **HTTP 메시지 바디에 바로 입력**한다. 따라서 실행 결과로 ok 메세지를 받을 수 있다. `@ResponseBody` 와 관련이 있는데, 뒤에서 더 자세히 설명한다.
- `@RequestMapping("/hello-basic")`
  - `/hello-basic` URL 호출이 오면 이 메서드가 실행되도록 매핑한다.
  - 대부분의 속성을 `배열[]` 로 제공하므로 다중 설정이 가능하다. `{"/hello-basic", "/hello-go"}`

Postman으로 테스트 해보자.

둘다 허용

다음 두가지 요청은 다른 URL이지만, 스프링은 다음 URL 요청들을 같은 요청으로 매핑한다.

- 매핑: `/hello-basic`
- URL 요청: `/hello-basic` , `/hello-basic/`

HTTP 메서드

`@RequestMapping` 에 `method` 속성으로 HTTP 메서드를 지정하지 않으면 HTTP 메서드와 무관하게 호출된다.

모두 허용 GET, HEAD, POST, PUT, PATCH, DELETE

HTTP 메서드 매핑

```
/**
 * method 특정 HTTP 메서드 요청만 허용
 * GET, HEAD, POST, PUT, PATCH, DELETE
 */
@RequestMapping(value = "/mapping-get-v1", method = RequestMethod.GET)
public String mappingGetV1() {
    log.info("mappingGetV1");
    return "ok";
}
```

만약 여기에 POST 요청을 하면 스프링 MVC는 HTTP 405 상태코드(Method Not Allowed)를 반환한다.

HTTP 메서드 매핑 축약

```
/**
 * 편리한 축약 애노테이션 (코드보기)
 * @GetMapping
 * @PostMapping
 * @PutMapping
 * @DeleteMapping
 * @PatchMapping
 */
@GetMapping(value = "/mapping-get-v2")
public String mappingGetV2() {
    log.info("mapping-get-v2");
    return "ok";
}
```

HTTP 메서드를 축약한 애노테이션을 사용하는 것이 더 직관적이다. 코드를 보면 내부에서 `@RequestMapping`과 `method`를 지정해서 사용하는 것을 확인할 수 있다.

PathVariable(경로 변수) 사용

```
/**
```

```
 * PathVariable 사용
 * 변수명이 같으면 생략 가능
 * @PathVariable("userId") String userId -> @PathVariable userId
 */
@GetMapping("/mapping/{userId}")
public String mappingPath(@PathVariable("userId") String data) {
    log.info("mappingPath userId={}", data);
    return "ok";
}
```

실행

- http://localhost:8080/mapping/userA

최근 HTTP API는 다음과 같이 리소스 경로에 식별자를 넣는 스타일을 선호한다.

- `/mapping/userA`
- `/users/1`

- `@RequestMapping`은 URL 경로를 템플릿화 할 수 있는데, `@PathVariable`을 사용하면 매칭 되는 부분을 편리하게 조회할 수 있다.
- `@PathVariable`의 이름과 파라미터 이름이 같으면 생략할 수 있다.

PathVariable 사용 - 다중

```
/**
 * PathVariable 사용 다중
 */
@GetMapping("/mapping/users/{userId}/orders/{orderId}")
public String mappingPath(@PathVariable String userId, @PathVariable Long
orderId) {
    log.info("mappingPath userId={}, orderId={}", userId, orderId);
    return "ok";
}
```

실행

- http://localhost:8080/mapping/users/userA/orders/100

특정 파라미터 조건 매핑

```
/**
 * 파라미터로 추가 매핑
 * params="mode",
 * params="!mode"
 * params="mode=debug"
 * params="mode!=debug" (! = )
 * params = {"mode=debug","data=good"}
 */
@GetMapping(value = "/mapping-param", params = "mode=debug")
public String mappingParam() {
    log.info("mappingParam");
    return "ok";
}
```

실행

- http://localhost:8080/mapping-param?mode=debug

특정 파라미터가 있거나 없는 조건을 추가할 수 있다. 잘 사용하지는 않는다.

특정 헤더 조건 매핑

```
/**
 * 특정 헤더로 추가 매핑
 * headers="mode",
 * headers="!mode"
 * headers="mode=debug"
 * headers="mode!=debug" (! = )
 */
@GetMapping(value = "/mapping-header", headers = "mode=debug")
public String mappingHeader() {
    log.info("mappingHeader");
    return "ok";
}
```

파라미터 매핑과 비슷하지만, HTTP 헤더를 사용한다.

Postman으로 테스트 해야 한다.

미디어 타입 조건 매핑 - HTTP 요청 Content-Type, consume

```
/**
 * Content-Type 헤더 기반 추가 매핑 Media Type
 * consumes="application/json"
 * consumes="!application/json"
 * consumes="application/*"
 * consumes="*\/*"
 * MediaType.APPLICATION_JSON_VALUE
 */
@PostMapping(value = "/mapping-consume", consumes = "application/json")
public String mappingConsumes() {
    log.info("mappingConsumes");
    return "ok";
}
```

Postman으로 테스트 해야 한다.

HTTP 요청의 Content-Type 헤더를 기반으로 미디어 타입으로 매핑한다.
만약 맞지 않으면 HTTP 415 상태코드(Unsupported Media Type)을 반환한다.

예시) consumes

```
consumes = "text/plain"
consumes = {"text/plain", "application/*"}
consumes = MediaType.TEXT_PLAIN_VALUE
```

미디어 타입 조건 매핑 - HTTP 요청 Accept, produce

```
/**
```

```
 * Accept 헤더 기반 Media Type
 * produces = "text/html"
 * produces = "!text/html"
 * produces = "text/*"
 * produces = "*\/*"
 */
@PostMapping(value = "/mapping-produce", produces = "text/html")
public String mappingProduces() {
    log.info("mappingProduces");
    return "ok";
}
```

HTTP 요청의 Accept 헤더를 기반으로 미디어 타입으로 매핑한다.
만약 맞지 않으면 HTTP 406 상태코드(Not Acceptable)을 반환한다.

예시)

```
produces = "text/plain"
produces = {"text/plain", "application/*"}
produces = MediaType.TEXT_PLAIN_VALUE
produces = "text/plain;charset=UTF-8"
```

요청 매핑 - API 예시

회원 관리를 HTTP API로 만든다 생각하고 매핑을 어떻게 하는지 알아보자.
(실제 데이터가 넘어가는 부분은 생략하고 URL 매핑만)

회원 관리 API

- 회원 목록 조회: GET `/users`
- 회원 등록: POST `/users`
- 회원 조회: GET `/users/{userId}`
- 회원 수정: PATCH `/users/{userId}`
- 회원 삭제: DELETE `/users/{userId}`

MappingClassController

```
package hello.springmvc.basic.requestmapping;

import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/mapping/users")
public class MappingClassController {

    /**
     * GET /mapping/users
     */
    @GetMapping
    public String users() {
        return "get users";
    }

    /**
     * POST /mapping/users
     */
    @PostMapping
    public String addUser() {
        return "post user";
    }

    /**
     * GET /mapping/users/{userId}
     */
    @GetMapping("/{userId}")
    public String findUser(@PathVariable String userId) {
        return "get userId=" + userId;
    }

    /**
     * PATCH /mapping/users/{userId}
     */
    @PatchMapping("/{userId}")
    public String updateUser(@PathVariable String userId) {
```

```
        return "update userId=" + userId;
    }

    /**
     * DELETE /mapping/users/{userId}
     */
    @DeleteMapping("/{userId}")
    public String deleteUser(@PathVariable String userId) {
        return "delete userId=" + userId;
    }

}
```

- `/mapping`는 강의의 다른 예제들과 구분하기 위해 사용했다.
- `@RequestMapping("/mapping/users")`
  - 클래스 레벨에 매핑 정보를 두면 메서드 레벨에서 해당 정보를 조합해서 사용한다.

Postman으로 테스트

- 회원 목록 조회: GET `/mapping/users`
- 회원 등록: POST `/mapping/users`
- 회원 조회: GET `/mapping/users/id1`
- 회원 수정: PATCH `/mapping/users/id1`
- 회원 삭제: DELETE `/mapping/users/id1`

매핑 방법을 이해했으니, 이제부터 HTTP 요청이 보내는 데이터들을 스프링 MVC로 어떻게 조회하는지 알아보자.

HTTP 요청 - 기본, 헤더 조회

애노테이션 기반의 스프링 컨트롤러는 다양한 파라미터를 지원한다.
이번 시간에는 HTTP 헤더 정보를 조회하는 방법을 알아보자.

RequestHeaderController

```java
package hello.springmvc.basic.request;

import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpMethod;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Locale;

@Slf4j
@RestController
public class RequestHeaderController {

    @RequestMapping("/headers")
    public String headers(HttpServletRequest request,
                          HttpServletResponse response,
                          HttpMethod httpMethod,
                          Locale locale,
                          @RequestHeader MultiValueMap<String, String>
headerMap,
                          @RequestHeader("host") String host,
                          @CookieValue(value = "myCookie", required = false)
String cookie
                          ) {

        log.info("request={}", request);
        log.info("response={}", response);
        log.info("httpMethod={}", httpMethod);
        log.info("locale={}", locale);
        log.info("headerMap={}", headerMap);
        log.info("header host={}", host);
        log.info("myCookie={}", cookie);

        return "ok";
    }

}
```

- `HttpServletRequest`
- `HttpServletResponse`
- `HttpMethod` : HTTP 메서드를 조회한다. `org.springframework.http.HttpMethod`
- `Locale` : Locale 정보를 조회한다.
- `@RequestHeader MultiValueMap<String, String> headerMap`
  - 모든 HTTP 헤더를 MultiValueMap 형식으로 조회한다.
- `@RequestHeader("host") String host`
  - 특정 HTTP 헤더를 조회한다.
  - 속성
    - 필수 값 여부: `required`
    - 기본 값 속성: `defaultValue`
- `@CookieValue(value = "myCookie", required = false) String cookie`
  - 특정 쿠키를 조회한다.
  - 속성
    - 필수 값 여부: `required`
    - 기본 값: `defaultValue`

`MultiValueMap`

- MAP과 유사한데, 하나의 키에 여러 값을 받을 수 있다.
- HTTP header, HTTP 쿼리 파라미터와 같이 하나의 키에 여러 값을 받을 때 사용한다.
  - **keyA=value1&keyA=value2**

```
MultiValueMap<String, String> map = new LinkedMultiValueMap();
map.add("keyA", "value1");
map.add("keyA", "value2");

//[value1,value2]
List<String> values = map.get("keyA");
```

@Slf4j

다음 코드를 자동으로 생성해서 로그를 선언해준다. 개발자는 편리하게 `log` 라고 사용하면 된다.

```
private static final org.slf4j.Logger log =
org.slf4j.LoggerFactory.getLogger(RequestHeaderController.class);
```

> **참고**
> `@Conroller`의 사용 가능한 파라미터 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
> https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-arguments

> **참고**
> `@Conroller`의 사용 가능한 응답 값 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
> https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-return-types

HTTP 요청 파라미터 - 쿼리 파라미터, HTML Form

HTTP 요청 데이터 조회 - 개요

서블릿에서 학습했던 HTTP 요청 데이터를 조회 하는 방법을 다시 떠올려보자. 그리고 서블릿으로 학습했던 내용을 스프링이 얼마나 깔끔하고 효율적으로 바꾸어주는지 알아보자.

HTTP 요청 메시지를 통해 클라이언트에서 서버로 데이터를 전달하는 방법을 알아보자.

클라이언트에서 서버로 요청 데이터를 전달할 때는 주로 다음 3가지 방법을 사용한다.

- **GET - 쿼리 파라미터**
  - /url**?username=hello&age=20**
  - 메시지 바디 없이, URL의 쿼리 파라미터에 데이터를 포함해서 전달
  - 예) 검색, 필터, 페이징등에서 많이 사용하는 방식
- **POST - HTML Form**
  - content-type: application/x-www-form-urlencoded
  - 메시지 바디에 쿼리 파리미터 형식으로 전달 username=hello&age=20
  - 예) 회원 가입, 상품 주문, HTML Form 사용
- **HTTP message body**에 데이터를 직접 담아서 요청
  - HTTP API에서 주로 사용, JSON, XML, TEXT
  - 데이터 형식은 주로 JSON 사용
  - POST, PUT, PATCH

하나씩 알아보자.

요청 파라미터 - 쿼리 파라미터, HTML Form

`HttpServletRequest`의 `request.getParameter()`를 사용하면 다음 두가지 요청 파라미터를 조회할 수 있다.

GET, 쿼리 파라미터 전송

예시

`http://localhost:8080/request-param?username=hello&age=20`

POST, HTML Form 전송

예시

```
POST /request-param ...
content-type: application/x-www-form-urlencoded

username=hello&age=20
```

GET 쿼리 파리미터 전송 방식이든, POST HTML Form 전송 방식이든 둘다 형식이 같으므로 구분없이 조회할 수 있다.
이것을 간단히 **요청 파라미터(request parameter) 조회**라 한다.

지금부터 스프링으로 요청 파라미터를 조회하는 방법을 단계적으로 알아보자.

RequestParamController

```
package hello.springmvc.basic.request;

import hello.springmvc.basic.HelloData;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;
```

```
@Slf4j
@Controller
public class RequestParamController {

    /**
     * 반환 타입이 없으면서 이렇게 응답에 값을 직접 집어넣으면, view 조회X
     */
    @RequestMapping("/request-param-v1")
    public void requestParamV1(HttpServletRequest request, HttpServletResponse
response) throws IOException {

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));
        log.info("username={}, age={}", username, age);

        response.getWriter().write("ok");
    }
}
```

request.getParameter()

여기서는 단순히 HttpServletRequest가 제공하는 방식으로 요청 파라미터를 조회했다.

GET 실행

http://localhost:8080/request-param-v1?username=hello&age=20

Post Form 페이지 생성

먼저 테스트용 HTML Form을 만들어야 한다.

리소스는 `/resources/static` 아래에 두면 스프링 부트가 자동으로 인식한다.

`main/resources/static/basic/hello-form.html`

```
<!DOCTYPE html>
<html>
<head>
```

```
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form action="/request-param-v1" method="post">
        username: <input type="text" name="username" />
        age:      <input type="text" name="age" />
        <button type="submit">전송</button>
    </form>
</body>
</html>
```

Post Form 실행

http://localhost:8080/basic/hello-form.html

> **참고**
> `Jar`를 사용하면 `webapp` 경로를 사용할 수 없다. 이제부터 정적 리소스도 클래스 경로에 함께 포함해야 한다.

HTTP 요청 파라미터 - @RequestParam

스프링이 제공하는 `@RequestParam`을 사용하면 요청 파라미터를 매우 편리하게 사용할 수 있다.

requestParamV2

```
/**
 * @RequestParam 사용
 * - 파라미터 이름으로 바인딩
 * @ResponseBody 추가
 * - View 조회를 무시하고, HTTP message body에 직접 해당 내용 입력
 */
@ResponseBody
@RequestMapping("/request-param-v2")
public String requestParamV2(
        @RequestParam("username") String memberName,
```

```
        @RequestParam("age") int memberAge) {

    log.info("username={}, age={}", memberName, memberAge);
    return "ok";
}
```

- `@RequestParam` : 파라미터 이름으로 바인딩
- `@ResponseBody` : View 조회를 무시하고, HTTP message body에 직접 해당 내용 입력

@RequestParam의 `name(value)` 속성이 파라미터 이름으로 사용

- @RequestParam("**username**") String **memberName**
- ➡ request.getParameter("**username**")

requestParamV3

```
/**
 * @RequestParam 사용
 * HTTP 파라미터 이름이 변수 이름과 같으면 @RequestParam(name="xx") 생략 가능
 */
@ResponseBody
@RequestMapping("/request-param-v3")
public String requestParamV3(
        @RequestParam String username,
        @RequestParam int age) {
    log.info("username={}, age={}", username, age);
    return "ok";
}
```

HTTP 파라미터 이름이 변수 이름과 같으면 `@RequestParam(name="xx")` 생략 가능

requestParamV4

```
/**
 * @RequestParam 사용
 * String, int 등의 단순 타입이면 @RequestParam 도 생략 가능
 */
```

```
@ResponseBody
@RequestMapping("/request-param-v4")
public String requestParamV4(String username, int age) {
    log.info("username={}, age={}", username, age);
    return "ok";
}
```

`String`, `int`, `Integer` 등의 단순 타입이면 `@RequestParam` 도 생략 가능

> **주의**
> `@RequestParam` 애노테이션을 생략하면 스프링 MVC는 내부에서 `required=false` 를 적용한다.
> `required` 옵션은 바로 다음에 설명한다.

> **참고**
> 이렇게 애노테이션을 완전히 생략해도 되는데, 너무 없는 것도 약간 과하다는 주관적 생각이 있다.
> `@RequestParam` 이 있으면 명확하게 요청 파리미터에서 데이터를 읽는 다는 것을 알 수 있다.

파라미터 필수 여부 - requestParamRequired

```
/**
 * @RequestParam.required
 * /request-param -> username이 없으므로 예외
 *
 * 주의!
 * /request-param?username= -> 빈문자로 통과
 *
 * 주의!
 * /request-param
 * int age -> null을 int에 입력하는 것은 불가능, 따라서 Integer 변경해야 함(또는 다음에 나오는
defaultValue 사용)
 */
@ResponseBody
@RequestMapping("/request-param-required")
public String requestParamRequired(
        @RequestParam(required = true) String username,
```

```
        @RequestParam(required = false) Integer age) {

    log.info("username={}, age={}", username, age);
    return "ok";
}
```

- `@RequestParam.required`
  - 파라미터 필수 여부
  - 기본값이 파라미터 필수(`true`)이다.

- `/request-param` 요청
  - `username` 이 없으므로 400 예외가 발생한다.

주의! - 파라미터 이름만 사용

`/request-param?username=`
파라미터 이름만 있고 값이 없는 경우 ➡ 빈문자로 통과

주의! - 기본형(primitive)에 null 입력

- `/request-param` 요청
- `@RequestParam(required = false) int age`

`null` 을 `int` 에 입력하는 것은 불가능(500 예외 발생)
따라서 `null` 을 받을 수 있는 `Integer` 로 변경하거나, 또는 다음에 나오는 `defaultValue` 사용

기본 값 적용 - requestParamDefault

```
/**
 * @RequestParam
 * - defaultValue 사용
 *
 * 참고: defaultValue는 빈 문자의 경우에도 적용
 * /request-param?username=
 */
@ResponseBody
@RequestMapping("/request-param-default")
public String requestParamDefault(
        @RequestParam(required = true, defaultValue = "guest") String username,
```

```
        @RequestParam(required = false, defaultValue = "-1") int age) {

    log.info("username={}, age={}", username, age);
    return "ok";
}
```

파라미터에 값이 없는 경우 `defaultValue`를 사용하면 기본 값을 적용할 수 있다.
이미 기본 값이 있기 때문에 `required`는 의미가 없다.

`defaultValue`는 빈 문자의 경우에도 설정한 기본 값이 적용된다.
`/request-param?username=`

파라미터를 Map으로 조회하기 - requestParamMap

```
/**
 * @RequestParam Map, MultiValueMap
 * Map(key=value)
 * MultiValueMap(key=[value1, value2, ...] ex) (key=userIds, value=[id1, id2])
 */
@ResponseBody
@RequestMapping("/request-param-map")
public String requestParamMap(@RequestParam Map<String, Object> paramMap) {
    log.info("username={}, age={}", paramMap.get("username"),
paramMap.get("age"));
    return "ok";
}
```

파라미터를 Map, MultiValueMap으로 조회할 수 있다.

- `@RequestParam Map`,
  - `Map(key=value)`
- `@RequestParam MultiValueMap`
  - `MultiValueMap(key=[value1, value2, ...] ex) (key=userIds, value=[id1, id2])`

파라미터의 값이 1개가 확실하다면 `Map`을 사용해도 되지만, 그렇지 않다면 `MultiValueMap`을 사용하자.

HTTP 요청 파라미터 - @ModelAttribute

실제 개발을 하면 요청 파라미터를 받아서 필요한 객체를 만들고 그 객체에 값을 넣어주어야 한다. 보통 다음과 같이 코드를 작성할 것이다.

```
@RequestParam String username;
@RequestParam int age;

HelloData data = new HelloData();
data.setUsername(username);
data.setAge(age);
```

스프링은 이 과정을 완전히 자동화해주는 `@ModelAttribute` 기능을 제공한다.

먼저 요청 파라미터를 바인딩 받을 객체를 만들자.

HelloData

```
package hello.springmvc.basic;

import lombok.Data;

@Data
public class HelloData {
    private String username;
    private int age;
}
```

- 롬복 `@Data`
  - `@Getter`, `@Setter`, `@ToString`, `@EqualsAndHashCode`, `@RequiredArgsConstructor`를 자동으로 적용해준다.

@ModelAttribute 적용 - modelAttributeV1

```
/**
```

```
 * @ModelAttribute 사용
 * 참고: model.addAttribute(helloData) 코드도 함께 자동 적용됨, 뒤에 model을 설명할 때
자세히 설명
 */
@ResponseBody
@RequestMapping("/model-attribute-v1")
public String modelAttributeV1(@ModelAttribute HelloData helloData) {
    log.info("username={}, age={}", helloData.getUsername(),
helloData.getAge());
    return "ok";
}
```

마치 마법처럼 `HelloData` 객체가 생성되고, 요청 파라미터의 값도 모두 들어가 있다.

스프링MVC는 `@ModelAttribute`가 있으면 다음을 실행한다.

- `HelloData` 객체를 생성한다.
- 요청 파라미터의 이름으로 `HelloData` 객체의 프로퍼티를 찾는다. 그리고 해당 프로퍼티의 setter를 호출해서 파라미터의 값을 입력(바인딩) 한다.
- 예) 파라미터 이름이 `username` 이면 `setUsername()` 메서드를 찾아서 호출하면서 값을 입력한다.

프로퍼티

객체에 `getUsername()`, `setUsername()` 메서드가 있으면, 이 객체는 `username` 이라는 프로퍼티를 가지고 있다.

`username` 프로퍼티의 값을 변경하면 `setUsername()` 이 호출되고, 조회하면 `getUsername()` 이 호출된다.

```
class HelloData {
    getUsername();
    setUsername();
}
```

바인딩 오류

`age=abc` 처럼 숫자가 들어가야 할 곳에 문자를 넣으면 `BindException` 이 발생한다. 이런 바인딩 오류를 처리하는 방법은 검증 부분에서 다룬다.

@ModelAttribute 생략 - modelAttributeV2

```
/**
 * @ModelAttribute 생략 가능
 * String, int 같은 단순 타입 = @RequestParam
 * argument resolver 로 지정해둔 타입 외 = @ModelAttribute
 */
@ResponseBody
@RequestMapping("/model-attribute-v2")
public String modelAttributeV2(HelloData helloData) {
    log.info("username={}, age={}", helloData.getUsername(),
helloData.getAge());
    return "ok";
}
```

`@ModelAttribute`는 생략할 수 있다.
그런데 `@RequestParam`도 생략할 수 있으니 혼란이 발생할 수 있다.

스프링은 해당 생략시 다음과 같은 규칙을 적용한다.

- `String`, `int`, `Integer` 같은 단순 타입 = `@RequestParam`
- 나머지 = `@ModelAttribute` (argument resolver 로 지정해둔 타입 외)

> **참고**
> argument resolver는 뒤에서 학습한다.

HTTP 요청 메시지 - 단순 텍스트

서블릿에서 학습한 내용을 떠올려보자.

- **HTTP message body**에 데이터를 직접 담아서 요청
  - HTTP API에서 주로 사용, JSON, XML, TEXT
  - 데이터 형식은 주로 JSON 사용
  - POST, PUT, PATCH

요청 파라미터와 다르게, HTTP 메시지 바디를 통해 데이터가 직접 데이터가 넘어오는 경우는 `@RequestParam`, `@ModelAttribute`를 사용할 수 없다. (물론 HTML Form 형식으로 전달되는 경우는

요청 파라미터로 인정된다.)

- 먼저 가장 단순한 텍스트 메시지를 HTTP 메시지 바디에 담아서 전송하고, 읽어보자.
- HTTP 메시지 바디의 데이터를 `InputStream` 을 사용해서 직접 읽을 수 있다.

RequestBodyStringController

```
@Slf4j
@Controller
public class RequestBodyStringController {

    @PostMapping("/request-body-string-v1")
    public void requestBodyString(HttpServletRequest request,
HttpServletResponse response) throws IOException {

        ServletInputStream inputStream = request.getInputStream();
        String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);

        log.info("messageBody={}", messageBody);

        response.getWriter().write("ok");
    }
}
```

Postman을 사용해서 테스트 해보자.

- POST http://localhost:8080/request-body-string-v1
- Body ➡ row, Text 선택

Input, Output 스트림, Reader - requestBodyStringV2

```
/**
 * InputStream(Reader): HTTP 요청 메시지 바디의 내용을 직접 조회
 * OutputStream(Writer): HTTP 응답 메시지의 바디에 직접 결과 출력
 */
@PostMapping("/request-body-string-v2")
public void requestBodyStringV2(InputStream inputStream, Writer responseWriter)
```

```
throws IOException {
    String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);
    log.info("messageBody={}", messageBody);
    responseWriter.write("ok");
}
```

스프링 MVC는 다음 파라미터를 지원한다.

- InputStream(Reader): HTTP 요청 메시지 바디의 내용을 직접 조회
- OutputStream(Writer): HTTP 응답 메시지의 바디에 직접 결과 출력

HttpEntity - requestBodyStringV3

```
/**
 * HttpEntity: HTTP header, body 정보를 편라하게 조회
 * - 메시지 바디 정보를 직접 조회(@RequestParam X, @ModelAttribute X)
 * - HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 *
 * 응답에서도 HttpEntity 사용 가능
 * - 메시지 바디 정보 직접 반환(view 조회X)
 * - HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 */
@PostMapping("/request-body-string-v3")
public HttpEntity<String> requestBodyStringV3(HttpEntity<String> httpEntity) {
    String messageBody = httpEntity.getBody();
    log.info("messageBody={}", messageBody);

    return new HttpEntity<>("ok");
}
```

스프링 MVC는 다음 파라미터를 지원한다.

- **HttpEntity**: HTTP header, body 정보를 편리하게 조회
  - 메시지 바디 정보를 직접 조회
  - 요청 파라미터를 조회하는 기능과 관계 없음 `@RequestParam` X, `@ModelAttribute` X
- **HttpEntity는 응답에도 사용 가능**
  - 메시지 바디 정보 직접 반환

- 헤더 정보 포함 가능
- view 조회X

`HttpEntity`를 상속받은 다음 객체들도 같은 기능을 제공한다.

- **RequestEntity**
  - HttpMethod, url 정보가 추가, 요청에서 사용
- **ResponseEntity**
  - HTTP 상태 코드 설정 가능, 응답에서 사용
  - `return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED)`

> **참고**
> 스프링MVC 내부에서 HTTP 메시지 바디를 읽어서 문자나 객체로 변환해서 전달해주는데, 이때 HTTP 메시지 컨버터(`HttpMessageConverter`)라는 기능을 사용한다. 이것은 조금 뒤에 HTTP 메시지 컨버터에서 자세히 설명한다.

@RequestBody - requestBodyStringV4

```
/**
 * @RequestBody
 * - 메시지 바디 정보를 직접 조회(@RequestParam X, @ModelAttribute X)
 * - HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 *
 * @ResponseBody
 * - 메시지 바디 정보 직접 반환(view 조회X)
 * - HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 */
@ResponseBody
@PostMapping("/request-body-string-v4")
public String requestBodyStringV4(@RequestBody String messageBody) {
    log.info("messageBody={}", messageBody);
    return "ok";
}
```

@RequestBody

`@RequestBody`를 사용하면 HTTP 메시지 바디 정보를 편리하게 조회할 수 있다. 참고로 헤더 정보가 필요하다면 `HttpEntity`를 사용하거나 `@RequestHeader`를 사용하면 된다.
이렇게 메시지 바디를 직접 조회하는 기능은 요청 파라미터를 조회하는 `@RequestParam`, `@ModelAttribute`와는 전혀 관계가 없다.

요청 파라미터 vs HTTP 메시지 바디

- 요청 파라미터를 조회하는 기능: `@RequestParam`, `@ModelAttribute`
- HTTP 메시지 바디를 직접 조회하는 기능: `@RequestBody`

@ResponseBody

`@ResponseBody`를 사용하면 응답 결과를 HTTP 메시지 바디에 직접 담아서 전달할 수 있다.
물론 이 경우에도 view를 사용하지 않는다.

HTTP 요청 메시지 - JSON

이번에는 HTTP API에서 주로 사용하는 JSON 데이터 형식을 조회해보자.

기존 서블릿에서 사용했던 방식과 비슷하게 시작해보자.

RequestBodyJsonController

```
package hello.springmvc.basic.request;

import com.fasterxml.jackson.databind.ObjectMapper;
import hello.springmvc.basic.HelloData;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.util.StreamUtils;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

import javax.servlet.ServletInputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```java
import java.io.IOException;
import java.nio.charset.StandardCharsets;

/**
 * {"username":"hello", "age":20}
 * content-type: application/json
 */
@Slf4j
@Controller
public class RequestBodyJsonController {

    private ObjectMapper objectMapper = new ObjectMapper();

    @PostMapping("/request-body-json-v1")
    public void requestBodyJsonV1(HttpServletRequest request,
HttpServletResponse response) throws IOException {

        ServletInputStream inputStream = request.getInputStream();
        String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);

        log.info("messageBody={}", messageBody);
        HelloData data = objectMapper.readValue(messageBody, HelloData.class);
        log.info("username={}, age={}", data.getUsername(), data.getAge());

        response.getWriter().write("ok");
    }

}
```

- HttpServletRequest를 사용해서 직접 HTTP 메시지 바디에서 데이터를 읽어와서, 문자로 변환한다.
- 문자로 된 JSON 데이터를 Jackson 라이브러리인 `objectMapper`를 사용해서 자바 객체로 변환한다.

Postman으로 테스트

- POST http://localhost:8080/request-body-json-v1
- raw, JSON, content-type: application/json
- `{"username":"hello", "age":20}`

requestBodyJsonV2 - @RequestBody 문자 변환

```
/**
 * @RequestBody
 * HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 *
 * @ResponseBody
 * - 모든 메서드에 @ResponseBody 적용
 * - 메시지 바디 정보 직접 반환(view 조회X)
 * - HttpMessageConverter 사용 -> StringHttpMessageConverter 적용
 */
@ResponseBody
@PostMapping("/request-body-json-v2")
public String requestBodyJsonV2(@RequestBody String messageBody) throws
IOException {
    HelloData data = objectMapper.readValue(messageBody, HelloData.class);
    log.info("username={}, age={}", data.getUsername(), data.getAge());
    return "ok";
}
```

- 이전에 학습했던 `@RequestBody`를 사용해서 HTTP 메시지에서 데이터를 꺼내고 messageBody에 저장한다.
- 문자로 된 JSON 데이터인 `messageBody`를 `objectMapper`를 통해서 자바 객체로 변환한다.

문자로 변환하고 다시 json으로 변환하는 과정이 불편하다. @ModelAttribute처럼 한번에 객체로 변환할 수는 없을까?

requestBodyJsonV3 - @RequestBody 객체 변환

```
/**
 * @RequestBody 생략 불가능(@ModelAttribute 가 적용되어 버림)
 * HttpMessageConverter 사용 -> MappingJackson2HttpMessageConverter (content-
type: application/json)
 *
 */
@ResponseBody
```

```
@PostMapping("/request-body-json-v3")
public String requestBodyJsonV3(@RequestBody HelloData data) {
    log.info("username={}, age={}", data.getUsername(), data.getAge());
    return "ok";
}
```

@RequestBody 객체 파라미터

- `@RequestBody HelloData data`
- `@RequestBody`에 직접 만든 객체를 지정할 수 있다.

`HttpEntity`, `@RequestBody`를 사용하면 HTTP 메시지 컨버터가 HTTP 메시지 바디의 내용을 우리가 원하는 문자나 객체 등으로 변환해준다.
HTTP 메시지 컨버터는 문자 뿐만 아니라 JSON도 객체로 변환해주는데, 우리가 방금 V2에서 했던 작업을 대신 처리해준다.
자세한 내용은 뒤에 HTTP 메시지 컨버터에서 다룬다.

@RequestBody는 생략 불가능

`@ModelAttribute`에서 학습한 내용을 떠올려보자.

스프링은 `@ModelAttribute`, `@RequestParam`해당 생략시 다음과 같은 규칙을 적용한다.

- `String`, `int`, `Integer` 같은 단순 타입 = `@RequestParam`
- 나머지 = `@ModelAttribute` (argument resolver 로 지정해둔 타입 외)

따라서 이 경우 HelloData에 `@RequestBody`를 생략하면 `@ModelAttribute`가 적용되어버린다.
`HelloData data` ➡ `@ModelAttribute HelloData data`
따라서 생략하면 HTTP 메시지 바디가 아니라 요청 파라미터를 처리하게 된다.

> **주의**
> HTTP 요청시에 content-type이 application/json인지 꼭! 확인해야 한다. 그래야 JSON을 처리할 수 있는 HTTP 메시지 컨버터가 실행된다.

물론 앞서 배운 것과 같이 HttpEntity를 사용해도 된다.

requestBodyJsonV4 - HttpEntity

```
@ResponseBody
@PostMapping("/request-body-json-v4")
```

```
public String requestBodyJsonV4(HttpEntity<HelloData> httpEntity) {
    HelloData data = httpEntity.getBody();
    log.info("username={}, age={}", data.getUsername(), data.getAge());
    return "ok";
}
```

requestBodyJsonV5

```
/**
 * @RequestBody 생략 불가능(@ModelAttribute 가 적용되어 버림)
 * HttpMessageConverter 사용 -> MappingJackson2HttpMessageConverter (content-type: application/json)
 *
 * @ResponseBody 적용
 * - 메시지 바디 정보 직접 반환(view 조회X)
 * - HttpMessageConverter 사용 -> MappingJackson2HttpMessageConverter 적용 (Accept: application/json)
 */
@ResponseBody
@PostMapping("/request-body-json-v5")
public HelloData requestBodyJsonV5(@RequestBody HelloData data) {
    log.info("username={}, age={}", data.getUsername(), data.getAge());
    return data;
}
```

`@ResponseBody`

응답의 경우에도 `@ResponseBody` 를 사용하면 해당 객체를 HTTP 메시지 바디에 직접 넣어줄 수 있다.

물론 이 경우에도 `HttpEntity` 를 사용해도 된다.

- `@RequestBody` 요청
  - JSON 요청 ➡ HTTP 메시지 컨버터 ➡ 객체
- `@ResponseBody` 응답
  - 객체 ➡ HTTP 메시지 컨버터 ➡ JSON 응답

HTTP 응답 - 정적 리소스, 뷰 템플릿

응답 데이터는 이미 앞에서 일부 다룬 내용들이지만, 응답 부분에 초점을 맞추어서 정리해보자.
스프링(서버)에서 응답 데이터를 만드는 방법은 크게 3가지이다.

- 정적 리소스
  - 예) 웹 브라우저에 정적인 HTML, css, js을 제공할 때는, **정적 리소스**를 사용한다.
- 뷰 템플릿 사용
  - 예) 웹 브라우저에 동적인 HTML을 제공할 때는 뷰 템플릿을 사용한다.
- HTTP 메시지 사용
  - HTTP API를 제공하는 경우에는 HTML이 아니라 데이터를 전달해야 하므로, HTTP 메시지 바디에 JSON 같은 형식으로 데이터를 실어 보낸다.

정적 리소스

스프링 부트는 클래스패스의 다음 디렉토리에 있는 정적 리소스를 제공한다.
`/static`, `/public`, `/resources`, `/META-INF/resources`

`src/main/resources`는 리소스를 보관하는 곳이고, 또 클래스패스의 시작 경로이다.
따라서 다음 디렉토리에 리소스를 넣어두면 스프링 부트가 정적 리소스로 서비스를 제공한다.

정적 리소스 경로
`src/main/resources/static`

다음 경로에 파일이 들어있으면
`src/main/resources/static/basic/hello-form.html`

웹 브라우저에서 다음과 같이 실행하면 된다.
`http://localhost:8080/basic/hello-form.html`

정적 리소스는 해당 파일을 변경 없이 그대로 서비스하는 것이다.

뷰 템플릿

뷰 템플릿을 거쳐서 HTML이 생성되고, 뷰가 응답을 만들어서 전달한다.
일반적으로 HTML을 동적으로 생성하는 용도로 사용하지만, 다른 것들도 가능하다. 뷰 템플릿이 만들 수 있는 것이라면 뭐든지 가능하다.

스프링 부트는 기본 뷰 템플릿 경로를 제공한다.

뷰 템플릿 경로

`src/main/resources/templates`

뷰 템플릿 생성

`src/main/resources/templates/response/hello.html`

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<p th:text="${data}">empty</p>
</body>
</html>
```

ResponseViewController - 뷰 템플릿을 호출하는 컨트롤러

```
package hello.springmvc.basic.response;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class ResponseViewController {

    @RequestMapping("/response-view-v1")
    public ModelAndView responseViewV1() {
        ModelAndView mav = new ModelAndView("response/hello")
                .addObject("data", "hello!");

        return mav;
```

```
    }

    @RequestMapping("/response-view-v2")
    public String responseViewV2(Model model) {
        model.addAttribute("data", "hello!!");
        return "response/hello";
    }

    @RequestMapping("/response/hello")
    public void responseViewV3(Model model) {
        model.addAttribute("data", "hello!!");
    }
}
```

String을 반환하는 경우 - View or HTTP 메시지

`@ResponseBody`가 없으면 `response/hello`로 뷰 리졸버가 실행되어서 뷰를 찾고, 렌더링 한다.
`@ResponseBody`가 있으면 뷰 리졸버를 실행하지 않고, HTTP 메시지 바디에 직접 `response/hello`라는 문자가 입력된다.

여기서는 뷰의 논리 이름인 `response/hello`를 반환하면 다음 경로의 뷰 템플릿이 렌더링 되는 것을 확인할 수 있다.

- 실행: `templates/response/hello.html`

Void를 반환하는 경우

- `@Controller`를 사용하고, `HttpServletResponse`, `OutputStream(Writer)` 같은 HTTP 메시지 바디를 처리하는 파라미터가 없으면 요청 URL을 참고해서 논리 뷰 이름으로 사용
  - 요청 URL: `/response/hello`
  - 실행: `templates/response/hello.html`
- **참고로 이 방식은 명시성이 너무 떨어지고 이렇게 딱 맞는 경우도 많이 없어서, 권장하지 않는다.**

HTTP 메시지

`@ResponseBody`, `HttpEntity`를 사용하면, 뷰 템플릿을 사용하는 것이 아니라, HTTP 메시지 바디에 직접 응답 데이터를 출력할 수 있다.

Thymeleaf 스프링 부트 설정

다음 라이브러리를 추가하면(**이미 추가되어 있다.**)

`build.gradle`

```
`implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'`
```

스프링 부트가 자동으로 `ThymeleafViewResolver` 와 필요한 스프링 빈들을 등록한다. 그리고 다음 설정도 사용한다. 이 설정은 기본 값 이기 때문에 변경이 필요할 때만 설정하면 된다.

`application.properties`

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

> **참고**
> 스프링 부트의 타임리프 관련 추가 설정은 다음 공식 사이트를 참고하자. (페이지 안에서 thymeleaf 검색)
> https://docs.spring.io/spring-boot/docs/2.4.3/reference/html/appendix-application-properties.html#common-application-properties-templating

HTTP 응답 - HTTP API, 메시지 바디에 직접 입력

HTTP API를 제공하는 경우에는 HTML이 아니라 데이터를 전달해야 하므로, HTTP 메시지 바디에 JSON 같은 형식으로 데이터를 실어 보낸다.
HTTP 요청에서 응답까지 대부분 다루었으므로 이번시간에는 정리를 해보자.

> **참고**
> HTML이나 뷰 템플릿을 사용해도 HTTP 응답 메시지 바디에 HTML 데이터가 담겨서 전달된다. 여기서 설명하는 내용은 정적 리소스나 뷰 템플릿을 거치지 않고, 직접 HTTP 응답 메시지를 전달하는 경우를 말한다.

ResponseBodyController

```
package hello.springmvc.basic.response;

import hello.springmvc.basic.HelloData;
import lombok.extern.slf4j.Slf4j;
```

```java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.*;

import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Slf4j
@Controller
//@RestController
public class ResponseBodyController {

    @GetMapping("/response-body-string-v1")
    public void responseBodyV1(HttpServletResponse response) throws IOException
{
        response.getWriter().write("ok");
    }

    /**
     * HttpEntity, ResponseEntity(Http Status 추가)
     * @return
     */
    @GetMapping("/response-body-string-v2")
    public ResponseEntity<String> responseBodyV2() {
        return new ResponseEntity<>("ok", HttpStatus.OK);
    }

    @ResponseBody
    @GetMapping("/response-body-string-v3")
    public String responseBodyV3() {
        return "ok";
    }

    @GetMapping("/response-body-json-v1")
    public ResponseEntity<HelloData> responseBodyJsonV1() {
        HelloData helloData = new HelloData();
        helloData.setUsername("userA");
        helloData.setAge(20);
```

```
        return new ResponseEntity<>(helloData, HttpStatus.OK);
    }

    @ResponseStatus(HttpStatus.OK)
    @ResponseBody
    @GetMapping("/response-body-json-v2")
    public HelloData responseBodyJsonV2() {
        HelloData helloData = new HelloData();
        helloData.setUsername("userA");
        helloData.setAge(20);

        return helloData;
    }
}
```

responseBodyV1

서블릿을 직접 다룰 때 처럼

HttpServletResponse 객체를 통해서 HTTP 메시지 바디에 직접 `ok` 응답 메시지를 전달한다.

`response.getWriter().write("ok")`

responseBodyV2

`ResponseEntity` 엔티티는 `HttpEntity` 를 상속 받았는데, HttpEntity는 HTTP 메시지의 헤더, 바디 정보를 가지고 있다. `ResponseEntity` 는 여기에 더해서 HTTP 응답 코드를 설정할 수 있다.

`HttpStatus.CREATED` 로 변경하면 201 응답이 나가는 것을 확인할 수 있다.

responseBodyV3

`@ResponseBody` 를 사용하면 view를 사용하지 않고, HTTP 메시지 컨버터를 통해서 HTTP 메시지를 직접 입력할 수 있다. `ResponseEntity` 도 동일한 방식으로 동작한다.

responseBodyJsonV1

`ResponseEntity` 를 반환한다. HTTP 메시지 컨버터를 통해서 JSON 형식으로 변환되어서 반환된다.

responseBodyJsonV2

`ResponseEntity` 는 HTTP 응답 코드를 설정할 수 있는데, `@ResponseBody` 를 사용하면 이런 것을 설정하기 까다롭다.

`@ResponseStatus(HttpStatus.OK)` 애노테이션을 사용하면 응답 코드도 설정할 수 있다.

물론 애노테이션이기 때문에 응답 코드를 동적으로 변경할 수는 없다. 프로그램 조건에 따라서 동적으로 변경하려면 `ResponseEntity`를 사용하면 된다.

@RestController

`@Controller` 대신에 `@RestController` 애노테이션을 사용하면, 해당 컨트롤러에 모두 `@ResponseBody`가 적용되는 효과가 있다. 따라서 뷰 템플릿을 사용하는 것이 아니라, HTTP 메시지 바디에 직접 데이터를 입력한다. 이름 그대로 Rest API(HTTP API)를 만들 때 사용하는 컨트롤러이다.

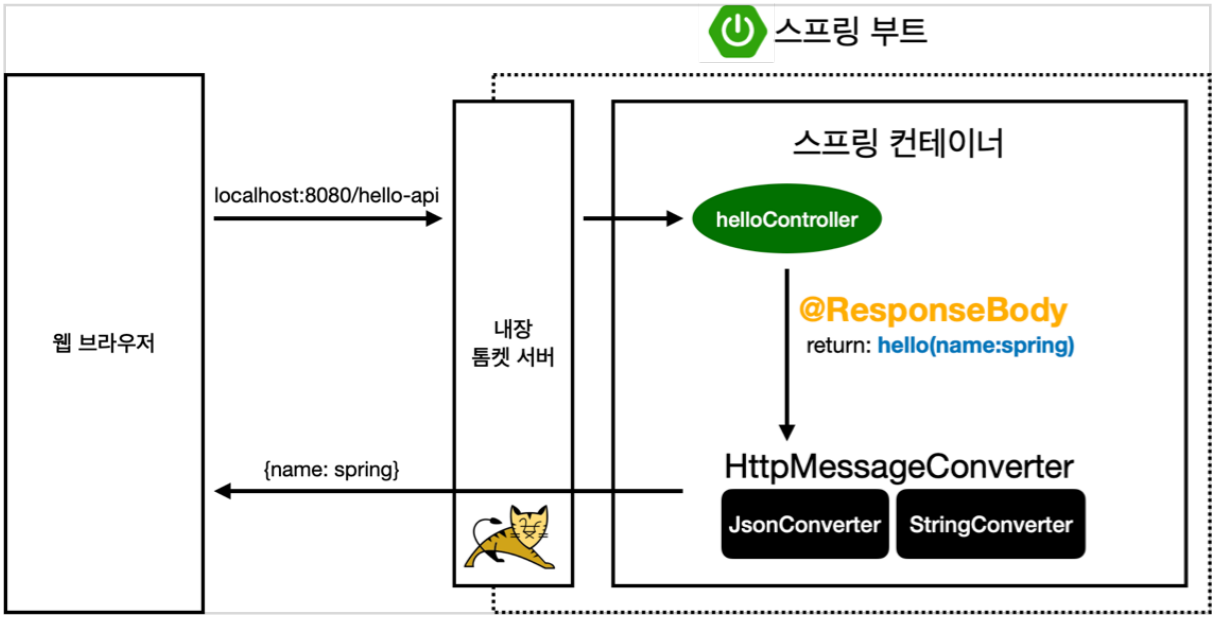
참고로 `@ResponseBody`는 클래스 레벨에 두면 전체에 메서드에 적용되는데, `@RestController` 에노테이션 안에 `@ResponseBody`가 적용되어 있다.

HTTP 메시지 컨버터

뷰 템플릿으로 HTML을 생성해서 응답하는 것이 아니라, HTTP API처럼 JSON 데이터를 HTTP 메시지 바디에서 직접 읽거나 쓰는 경우 HTTP 메시지 컨버터를 사용하면 편리하다.

HTTP 메시지 컨버터를 설명하기 전에 잠깐 과거로 돌아가서 스프링 입문 강의에서 설명했던 내용을 살펴보자.

@ResponseBody 사용 원리



- `@ResponseBody`를 사용
  - HTTP의 BODY에 문자 내용을 직접 반환
  - `viewResolver` 대신에 `HttpMessageConverter`가 동작

- 기본 문자처리: `StringHttpMessageConverter`
- 기본 객체처리: `MappingJackson2HttpMessageConverter`
- byte 처리 등등 기타 여러 HttpMessageConverter가 기본으로 등록되어 있음

> 참고: 응답의 경우 클라이언트의 HTTP Accept 해더와 서버의 컨트롤러 반환 타입 정보 둘을 조합해서 `HttpMessageConverter`가 선택된다. 더 자세한 내용은 스프링 MVC 강의에서 설명하겠다.

이제 다시 돌아와서.

스프링 MVC는 다음의 경우에 HTTP 메시지 컨버터를 적용한다.

- HTTP 요청: `@RequestBody`, `HttpEntity(RequestEntity)`,
- HTTP 응답: `@ResponseBody`, `HttpEntity(ResponseEntity)`,

HTTP 메시지 컨버터 인터페이스

`org.springframework.http.converter.HttpMessageConverter`

```
package org.springframework.http.converter;

public interface HttpMessageConverter<T> {

  boolean canRead(Class<?> clazz, @Nullable MediaType mediaType);
  boolean canWrite(Class<?> clazz, @Nullable MediaType mediaType);

  List<MediaType> getSupportedMediaTypes();

  T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
          throws IOException, HttpMessageNotReadableException;
  void write(T t, @Nullable MediaType contentType, HttpOutputMessage
outputMessage)
          throws IOException, HttpMessageNotWritableException;

}
```

HTTP 메시지 컨버터는 HTTP 요청, HTTP 응답 둘 다 사용된다.

- `canRead()`, `canWrite()`: 메시지 컨버터가 해당 클래스, 미디어타입을 지원하는지 체크
- `read()`, `write()`: 메시지 컨버터를 통해서 메시지를 읽고 쓰는 기능

스프링 부트 기본 메시지 컨버터

(일부 생략)

```
0 = ByteArrayHttpMessageConverter
1 = StringHttpMessageConverter
2 = MappingJackson2HttpMessageConverter
```

스프링 부트는 다양한 메시지 컨버터를 제공하는데, 대상 클래스 타입과 미디어 타입 둘을 체크해서 사용여부를 결정한다. 만약 만족하지 않으면 다음 메시지 컨버터로 우선순위가 넘어간다.

몇가지 주요한 메시지 컨버터를 알아보자.

- `ByteArrayHttpMessageConverter` : `byte[]` 데이터를 처리한다.
  - 클래스 타입: `byte[]` , 미디어타입: `*/*` ,
  - 요청 예) `@RequestBody byte[] data`
  - 응답 예) `@ResponseBody return byte[]` 쓰기 미디어타입 `application/octet-stream`
- `StringHttpMessageConverter` : `String` 문자로 데이터를 처리한다.
  - 클래스 타입: `String` , 미디어타입: `*/*`
  - 요청 예) `@RequestBody String data`
  - 응답 예) `@ResponseBody return "ok"` 쓰기 미디어타입 `text/plain`
- `MappingJackson2HttpMessageConverter` : application/json
  - 클래스 타입: 객체 또는 `HashMap` , 미디어타입 `application/json` 관련
  - 요청 예) `@RequestBody HelloData data`
  - 응답 예) `@ResponseBody return helloData` 쓰기 미디어타입 `application/json` 관련

StringHttpMessageConverter

```
content-type: application/json

@RequestMapping
void hello(@RequetsBody String data) {}
```

MappingJackson2HttpMessageConverter

```
content-type: application/json
```

```
@RequestMapping
void hello(@RequetsBody HelloData data) {}
```

?

```
content-type: text/html

@RequestMapping
void hello(@RequetsBody HelloData data) {}
```

HTTP 요청 데이터 읽기

- HTTP 요청이 오고, 컨트롤러에서 `@RequestBody`, `HttpEntity` 파라미터를 사용한다.
- 메시지 컨버터가 메시지를 읽을 수 있는지 확인하기 위해 `canRead()` 를 호출한다.
  - 대상 클래스 타입을 지원하는가.
    - 예) `@RequestBody` 의 대상 클래스 (`byte[]`, `String`, `HelloData`)
  - HTTP 요청의 Content-Type 미디어 타입을 지원하는가.
    - 예) `text/plain`, `application/json`, `*/*`
- `canRead()` 조건을 만족하면 `read()` 를 호출해서 객체 생성하고, 반환한다.

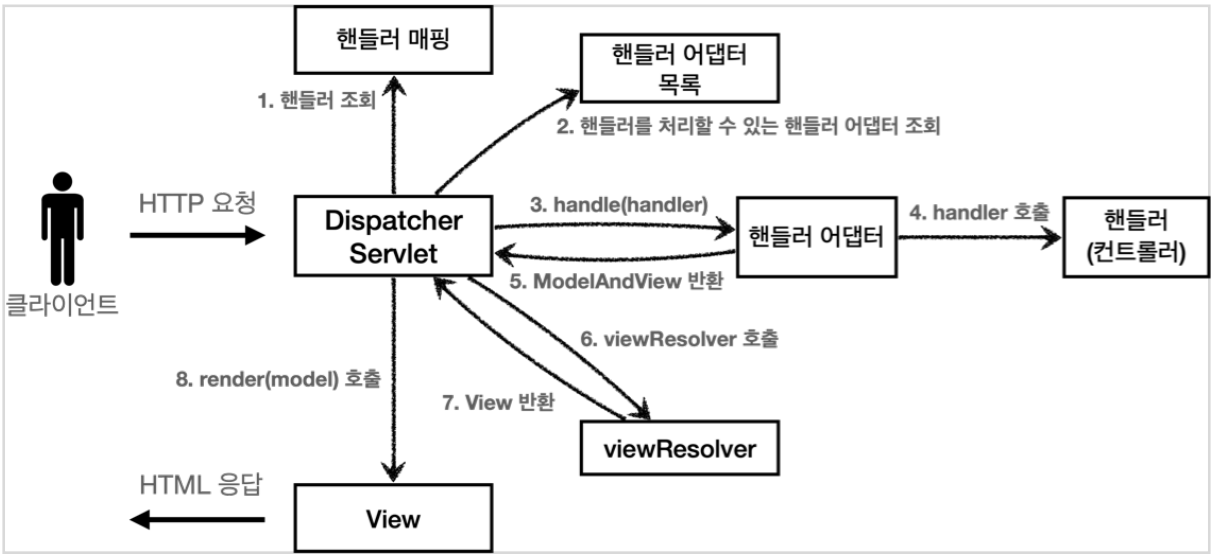
HTTP 응답 데이터 생성

- 컨트롤러에서 `@ResponseBody`, `HttpEntity` 로 값이 반환된다.
- 메시지 컨버터가 메시지를 쓸 수 있는지 확인하기 위해 `canWrite()` 를 호출한다.
  - 대상 클래스 타입을 지원하는가.
    - 예) return의 대상 클래스 (`byte[]`, `String`, `HelloData`)
  - HTTP 요청의 Accept 미디어 타입을 지원하는가.(더 정확히는 `@RequestMapping` 의 `produces`)
    - 예) `text/plain`, `application/json`, `*/*`
- `canWrite()` 조건을 만족하면 `write()` 를 호출해서 HTTP 응답 메시지 바디에 데이터를 생성한다.

요청 매핑 헨들러 어뎁터 구조

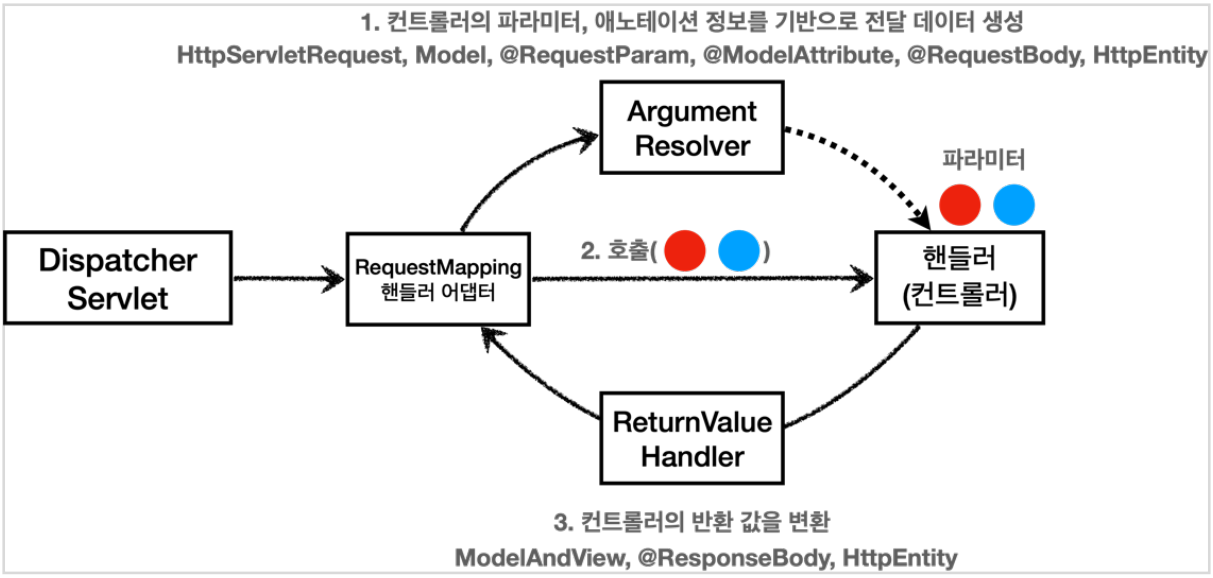
그렇다면 HTTP 메시지 컨버터는 스프링 MVC 어디쯤에서 사용되는 것일까?
다음 그림에서는 보이지 않는다.

SpringMVC 구조



모든 비밀은 애노테이션 기반의 컨트롤러, 그러니까 `@RequestMapping`을 처리하는 핸들러 어댑터인 `RequestMappingHandlerAdapter`(요청 매핑 헨들러 어뎁터)에 있다.

RequestMappingHandlerAdapter 동작 방식



ArgumentResolver

생각해보면, 애노테이션 기반의 컨트롤러는 매우 다양한 파라미터를 사용할 수 있었다.
`HttpServletRequest`, `Model`은 물론이고, `@RequestParam`, `@ModelAttribute` 같은 애노테이션 그리고 `@RequestBody`, `HttpEntity` 같은 HTTP 메시지를 처리하는 부분까지 매우 큰 유연함을 보여주었다.
이렇게 파라미터를 유연하게 처리할 수 있는 이유가 바로 `ArgumentResolver` 덕분이다.

애노테이션 기반 컨트롤러를 처리하는 `RequestMappingHandlerAdaptor`는 바로 이 `ArgumentResolver`를 호출해서 컨트롤러(핸들러)가 필요로 하는 다양한 파라미터의 값(객체)을 생성한다.
그리고 이렇게 파리미터의 값이 모두 준비되면 컨트롤러를 호출하면서 값을 넘겨준다.

스프링은 30개가 넘는 `ArgumentResolver`를 기본으로 제공한다.
어떤 종류들이 있는지 살짝 코드로 확인만 해보자.

> **참고**
> 가능한 파라미터 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
> https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-arguments

정확히는 `HandlerMethodArgumentResolver`인데 줄여서 `ArgumentResolver`라고 부른다.

```
public interface HandlerMethodArgumentResolver {

  boolean supportsParameter(MethodParameter parameter);

  @Nullable
  Object resolveArgument(MethodParameter parameter, @Nullable
ModelAndViewContainer mavContainer,
          NativeWebRequest webRequest, @Nullable WebDataBinderFactory
binderFactory) throws Exception;

}
```

동작 방식
`ArgumentResolver`의 `supportsParameter()`를 호출해서 해당 파라미터를 지원하는지 체크하고, 지원하면 `resolveArgument()`를 호출해서 실제 객체를 생성한다. 그리고 이렇게 생성된 객체가 컨트롤러 호출시 넘어가는 것이다.

그리고 원한다면 여러분이 직접 이 인터페이스를 확장해서 원하는 `ArgumentResolver`를 만들 수도 있다.
실제 확장하는 예제는 향후 로그인 처리에서 진행하겠다.

ReturnValueHandler

`HandlerMethodReturnValueHandler`를 줄여서 `ReturnValueHandler`라 부른다.
`ArgumentResolver`와 비슷한데, 이것은 응답 값을 변환하고 처리한다.

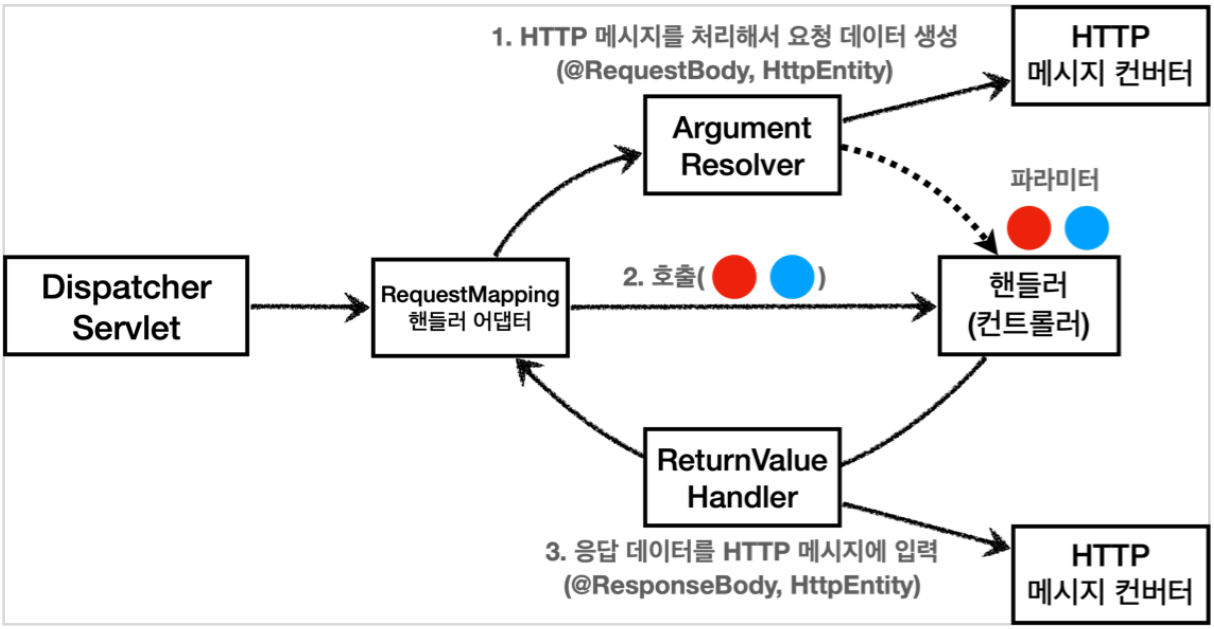
컨트롤러에서 String으로 뷰 이름을 반환해도, 동작하는 이유가 바로 ReturnValueHandler 덕분이다.
어떤 종류들이 있는지 살짝 코드로 확인만 해보자.

스프링은 10여개가 넘는 `ReturnValueHandler`를 지원한다.
예) `ModelAndView`, `@ResponseBody`, `HttpEntity`, `String`

> **참고**
> 가능한 응답 값 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
> https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-return-types

HTTP 메시지 컨버터

HTTP 메시지 컨버터 위치



HTTP 메시지 컨버터는 어디쯤 있을까?
HTTP 메시지 컨버터를 사용하는 `@RequestBody`도 컨트롤러가 필요로 하는 파라미터의 값에 사용된다.

`@ResponseBody`의 경우도 컨트롤러의 반환 값을 이용한다.

요청의 경우 `@RequestBody`를 처리하는 `ArgumentResolver`가 있고, `HttpEntity`를 처리하는 `ArgumentResolver`가 있다. 이 `ArgumentResolver`들이 HTTP 메시지 컨버터를 사용해서 필요한 객체를 생성하는 것이다. (어떤 종류가 있는지 코드로 살짝 확인해보자)

응답의 경우 `@ResponseBody`와 `HttpEntity`를 처리하는 `ReturnValueHandler`가 있다. 그리고 여기에서 HTTP 메시지 컨버터를 호출해서 응답 결과를 만든다.

스프링 MVC는 `@RequestBody` `@ResponseBody`가 있으면
`RequestResponseBodyMethodProcessor` (ArgumentResolver)
`HttpEntity`가 있으면 `HttpEntityMethodProcessor` (ArgumentResolver)를 사용한다.

> **참고**
> `HttpMessageConverter`를 구현한 클래스를 한번 확인해보자.

확장

스프링은 다음을 모두 인터페이스로 제공한다. 따라서 필요하면 언제든지 기능을 확장할 수 있다.

- `HandlerMethodArgumentResolver`
- `HandlerMethodReturnValueHandler`
- `HttpMessageConverter`

스프링이 필요한 대부분의 기능을 제공하기 때문에 실제 기능을 확장할 일이 많지는 않다. 기능 확장은 `WebMvcConfigurer`를 상속 받아서 스프링 빈으로 등록하면 된다. 실제 자주 사용하지는 않으니 실제 기능 확장이 필요할 때 `WebMvcConfigurer`를 검색해보자.

WebMvcConfigurer 확장

```
@Bean
public WebMvcConfigurer webMvcConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
resolvers) {
            //...
        }
```

```
        @Override
        public void extendMessageConverters(List<HttpMessageConverter<?>>
converters) {
            //...
        }
    };
}
```

정리

7. 스프링 MVC - 웹 페이지 만들기

#인강/4. 스프링 MVC 1/강의#

목차

프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택

    - Project: Gradle Project
    - Language: Java
    - Spring Boot: 2.4.x
- Project Metadata
    - Group: hello
    - Artifact: **item-service**
    - Name: item-service
    - Package name: **hello.itemservice**
    - Packaging: **Jar (주의!)**
    - Java: 11

- Dependencies: **Spring Web**, **Thymeleaf**, **Lombok**

build.gradle

```
plugins {
    id 'org.springframework.boot' version '2.4.3'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compileOnly 'org.projectlombok:lombok'
```

```
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`SpringmvcApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

Welcome 페이지 추가

편리하게 사용할 수 있도록 Welcome 페이지를 추가하자.

`/resources/static/index.html`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<ul>
    <li>상품 관리
        <ul>
            <li><a href="/basic/items">상품 관리 - 기본</a></li>
        </ul>
    </li>
</ul>
</body>
</html>
```

- 동작 확인

- 기본 메인 클래스 실행(`SpringmvcApplication.main()`)
- http://localhost:8080 호출해서 Welcome 페이지가 나오면 성공

요구사항 분석

상품을 관리할 수 있는 서비스를 만들어보자.

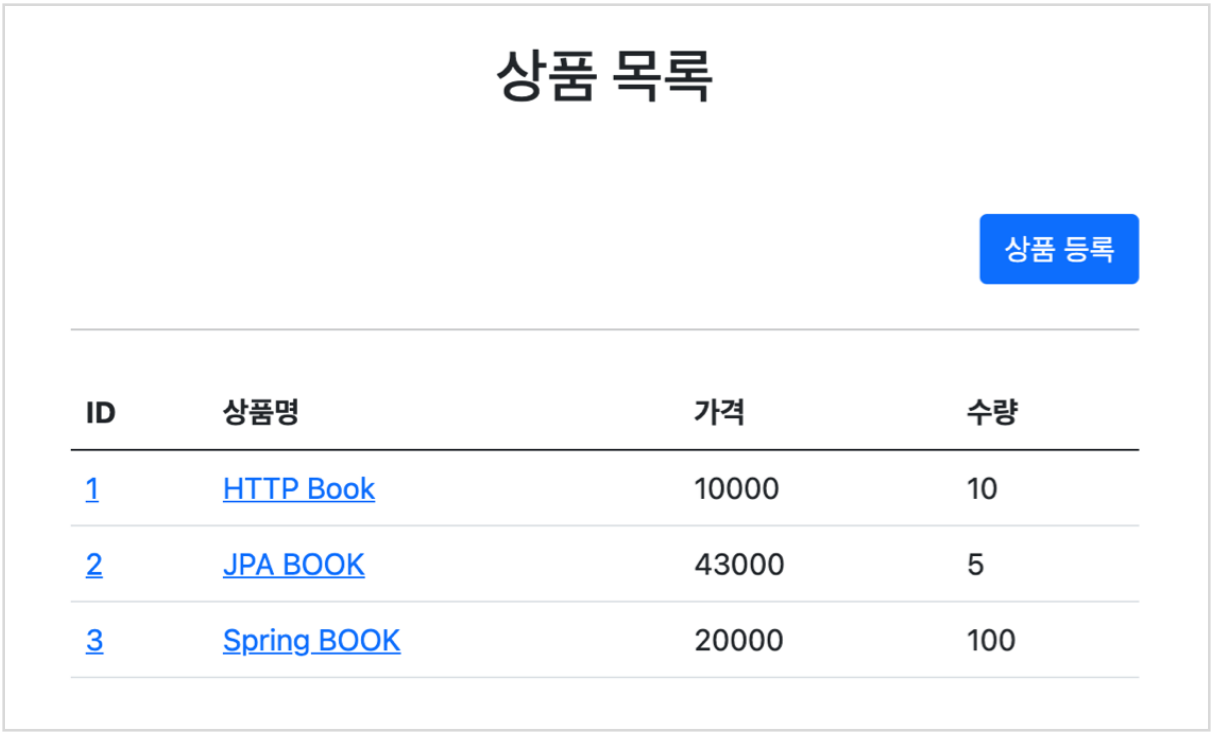
상품 도메인 모델

- 상품 ID
- 상품명
- 가격
- 수량

상품 관리 기능

- 상품 목록
- 상품 상세
- 상품 등록
- 상품 수정

서비스 화면



상품 상세

상품 ID

3

상품명

Spring BOOK

가격

20000

수량

100

상품 수정 목록으로

상품 등록 폼

상품 입력

상품명

Spring BOOK

가격

20000

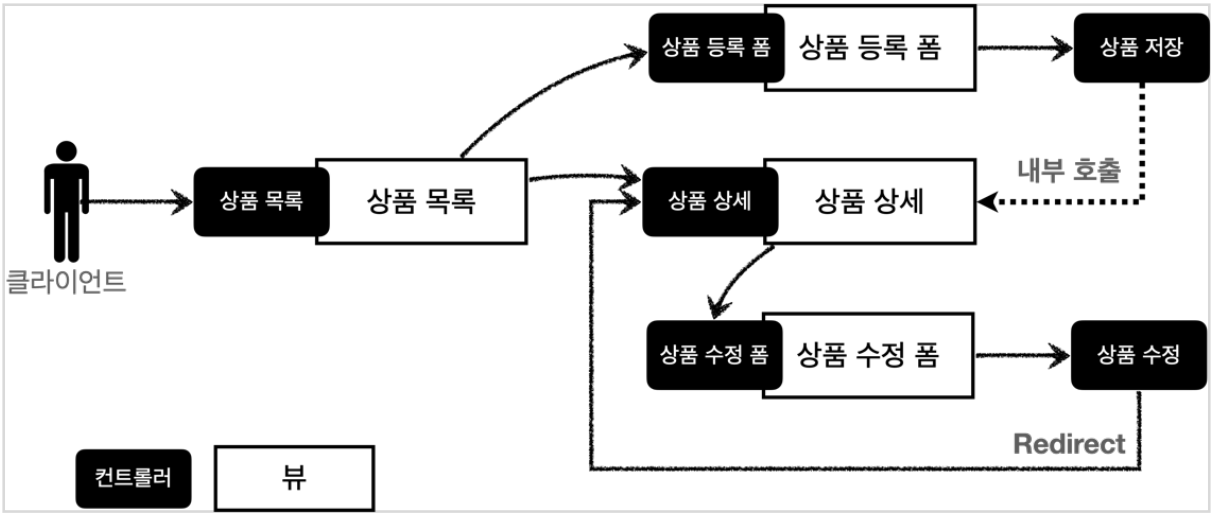
수량

100

상품 등록

취소

서비스 제공 흐름



요구사항이 정리되고 디자이너, 웹 퍼블리셔, 백엔드 개발자가 업무를 나누어 진행한다.

- **디자이너**: 요구사항에 맞도록 디자인하고, 디자인 결과물을 웹 퍼블리셔에게 넘겨준다.
- **웹 퍼블리셔**: 다자이너에서 받은 디자인을 기반으로 HTML, CSS를 만들어 개발자에게 제공한다.
- **백엔드 개발자**: 디자이너, 웹 퍼블리셔를 통해서 HTML 화면이 나오기 전까지 시스템을 설계하고, 핵심 비즈니스 모델을 개발한다. 이후 HTML이 나오면 이 HTML을 뷰 템플릿으로 변환해서 동적으로 화면을

그리고, 또 웹 화면의 흐름을 제어한다.

> **참고**
> React, Vue.js 같은 웹 클라이언트 기술을 사용하고, 웹 프론트엔드 개발자가 별도로 있으면, 웹 프론트엔드 개발자가 웹 퍼블리셔 역할까지 포함해서 하는 경우도 있다.
> 웹 클라이언트 기술을 사용하면, 웹 프론트엔드 개발자가 HTML을 동적으로 만드는 역할과 웹 화면의 흐름을 담당한다. 이 경우 백엔드 개발자는 HTML 뷰 템플릿을 직접 만지는 대신에, HTTP API를 통해 웹 클라이언트가 필요로 하는 데이터와 기능을 제공하면 된다.

상품 도메인 개발

Item - 상품 객체

```
package hello.itemservice.domain.item;

import lombok.Data;

package hello.itemservice.domain.item;

import lombok.Data;

@Data
public class Item {

    private Long id;
    private String itemName;
    private Integer price;
    private Integer quantity;

    public Item() {
    }

    public Item(String itemName, Integer price, Integer quantity) {
        this.itemName = itemName;
```

```
        this.price = price;
        this.quantity = quantity;
    }
}
```

ItemRepository - 상품 저장소

```
package hello.itemservice.domain.item;

import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Repository
public class ItemRepository {

    private static final Map<Long, Item> store = new HashMap<>(); //static 사용
    private static long sequence = 0L; //static 사용

    public Item save(Item item) {
        item.setId(++sequence);
        store.put(item.getId(), item);
        return item;
    }

    public Item findById(Long id) {
        return store.get(id);
    }

    public List<Item> findAll() {
        return new ArrayList<>(store.values());
    }

    public void update(Long itemId, Item updateParam) {
        Item findItem = findById(itemId);
```

```
        findItem.setItemName(updateParam.getItemName());
        findItem.setPrice(updateParam.getPrice());
        findItem.setQuantity(updateParam.getQuantity());
    }

    public void clearStore() {
        store.clear();
    }
}
```

ItemRepositoryTest - 상품 저장소 테스트

```
package hello.itemservice.domain.item;

import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.Test;

import java.util.List;

import static org.assertj.core.api.Assertions.assertThat;

class ItemRepositoryTest {

    ItemRepository itemRepository = new ItemRepository();

    @AfterEach
    void afterEach() {
        itemRepository.clearStore();
    }

    @Test
    void save() {
        //given
        Item item = new Item("itemA", 10000, 10);

        //when
        Item savedItem = itemRepository.save(item);
```

```
        //then
        Item findItem = itemRepository.findById(item.getId());
        assertThat(findItem).isEqualTo(savedItem);
    }

    @Test
    void findAll() {
        //given
        Item item1 = new Item("item1", 10000, 10);
        Item item2 = new Item("item2", 20000, 20);

        itemRepository.save(item1);
        itemRepository.save(item2);

        //when
        List<Item> result = itemRepository.findAll();

        //then
        assertThat(result.size()).isEqualTo(2);
        assertThat(result).contains(item1, item2);
    }


    @Test
    void updateItem() {
        //given
        Item item = new Item("item1", 10000, 10);

        Item savedItem = itemRepository.save(item);
        Long itemId = savedItem.getId();

        //when
        Item updateParam = new Item("item2", 20000, 30);
        itemRepository.update(itemId, updateParam);

        Item findItem = itemRepository.findById(itemId);

        //then
```

```
assertThat(findItem.getItemName()).isEqualTo(updateParam.getItemName());
        assertThat(findItem.getPrice()).isEqualTo(updateParam.getPrice());

assertThat(findItem.getQuantity()).isEqualTo(updateParam.getQuantity());

    }
}
```

상품 서비스 HTML

핵심 비즈니스 로직을 개발하는 동안, 웹 퍼블리셔는 HTML 마크업을 완료했다.
다음 파일들을 경로에 넣고 잘 동작하는지 확인해보자.

부트스트랩
참고로 HTML을 편리하게 개발하기 위해 부트스트랩 사용했다.
먼저 필요한 부트스트랩 파일을 설치하자

- **부트스트랩 공식 사이트**: https://getbootstrap.com
- **부트스트랩을 다운로드 받고 압축을 풀자.**
  - 이동: https://getbootstrap.com/docs/5.0/getting-started/download/
  - Compiled CSS and JS 항목을 다운로드하자.
  - 압축을 출고 `bootstrap.min.css` 를 복사해서 다음 폴더에 추가하자
  - `resources/static/css/bootstrap.min.css`

> **참고**
> 부트스트랩(Bootstrap)은 웹사이트를 쉽게 만들 수 있게 도와주는 HTML, CSS, JS 프레임워크이다.
> 하나의 CSS로 휴대폰, 태블릿, 데스크탑까지 다양한 기기에서 작동한다. 다양한 기능을 제공하여 사용자가 쉽게 웹사이트를 제작, 유지, 보수할 수 있도록 도와준다. - 출처: 위키백과

HTML, css 파일

- `/resources/static/css/bootstrap.min.css` ➜ 부트스트랩 다운로드
- `/resources/static/html/items.html` ➜ 아래 참조
- `/resources/static/html/item.html`
- `/resources/static/html/addForm.html`
- `/resources/static/html/editForm.html`

참고로 `/resources/static` 에 넣어두었기 때문에 스프링 부트가 정적 리소스를 제공한다.

- http://localhost:8080/html/items.html

그런데 정적 리소스여서 **해당 파일을 탐색기를 통해 직접 열어**도 동작하는 것을 확인할 수 있다.

> **참고**
> 이렇게 정적 리소스가 공개되는 `/resources/static` 폴더에 HTML을 넣어두면, 실제 서비스에서도 공개된다. 서비스를 운영한다면 지금처럼 공개할 필요없는 HTML을 두는 것은 주의하자.

상품 목록 HTML

`resources/static/html/items.html`

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>상품 목록</h2>
    </div>

    <div class="row">
        <div class="col">
            <button class="btn btn-primary float-end"
                    onclick="location.href='addForm.html'" type="button">상품
등록</button>
        </div>
    </div>

    <hr class="my-4">
    <div>
        <table class="table">
            <thead>
            <tr>
```

```
                <th>ID</th>
                <th>상품명</th>
                <th>가격</th>
                <th>수량</th>
            </tr>
            </thead>
            <tbody>
            <tr>
                <td><a href="item.html">1</a></td>
                <td><a href="item.html">테스트 상품1</a></td>
                <td>10000</td>
                <td>10</td>
            </tr>
            <tr>
                <td><a href="item.html">2</a></td>
                <td><a href="item.html">테스트 상품2</a></td>
                <td>20000</td>
                <td>20</td>
            </tr>
            </tbody>
        </table>
    </div>

</div> <!-- /container -->

</body>
</html>
```

상품 상세 HTML

`resources/static/html/item.html`

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css" rel="stylesheet">
    <style>
```

```
        .container {
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 상세</h2>
    </div>

    <div>
        <label for="itemId">상품 ID</label>
        <input type="text" id="itemId" name="itemId" class="form-control"
value="1" readonly>
    </div>
    <div>
        <label for="itemName">상품명</label>
        <input type="text" id="itemName" name="itemName" class="form-control"
value="상품A" readonly>
    </div>
    <div>
        <label for="price">가격</label>
        <input type="text" id="price" name="price" class="form-control"
value="10000" readonly>
    </div>
    <div>
        <label for="quantity">수량</label>
        <input type="text" id="quantity" name="quantity" class="form-control"
value="10" readonly>
    </div>

    <hr class="my-4">

    <div class="row">
        <div class="col">
            <button class="w-100 btn btn-primary btn-lg"
```

```
onclick="location.href='editForm.html'" type="button">상품 수정</button>
        </div>
        <div class="col">
            <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='items.html'" type="button">목록으로</button>
        </div>
    </div>

</div> <!-- /container -->
</body>
</html>
```

상품 등록 폼 HTML

resources/static/html/addForm.html

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css" rel="stylesheet">
    <style>
        .container {
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 등록 폼</h2>
    </div>

    <h4 class="mb-3">상품 입력</h4>

    <form action="item.html" method="post">
        <div>
```

```
            <label for="itemName">상품명</label>
            <input type="text" id="itemName" name="itemName" class="form-control" placeholder="이름을 입력하세요">
        </div>
        <div>
            <label for="price">가격</label>
            <input type="text" id="price" name="price" class="form-control" placeholder="가격을 입력하세요">
        </div>
        <div>
            <label for="quantity">수량</label>
            <input type="text" id="quantity" name="quantity" class="form-control" placeholder="수량을 입력하세요">
        </div>

        <hr class="my-4">

        <div class="row">
            <div class="col">
                <button class="w-100 btn btn-primary btn-lg" type="submit">상품 등록</button>
            </div>
            <div class="col">
                <button class="w-100 btn btn-secondary btn-lg" onclick="location.href='items.html'" type="button">취소</button>
            </div>
        </div>

    </form>

</div> <!-- /container -->
</body>
</html>
```

상품 수정 폼 HTML

`resources/static/html/editForm.html`

```
<!DOCTYPE HTML>
```

```
<html>
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css" rel="stylesheet">
    <style>
        .container {
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 수정 폼</h2>
    </div>

    <form action="item.html" method="post">
        <div>
            <label for="id">상품 ID</label>
            <input type="text" id="id" name="id" class="form-control" value="1"
readonly>
        </div>
        <div>
            <label for="itemName">상품명</label>
            <input type="text" id="itemName" name="itemName" class="form-
control" value="상품A">
        </div>
        <div>
            <label for="price">가격</label>
            <input type="text" id="price" name="price" class="form-control"
value="10000">
        </div>
        <div>
            <label for="quantity">수량</label>
            <input type="text" id="quantity" name="quantity" class="form-
control" value="10">
        </div>
```

```
        <hr class="my-4">

        <div class="row">
            <div class="col">
                <button class="w-100 btn btn-primary btn-lg" type="submit">저장
</button>
            </div>
            <div class="col">
                <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='item.html'" type="button">취소</button>
            </div>
        </div>

    </form>

</div> <!-- /container -->
</body>
</html>
```

상품 목록 - 타임리프

본격적으로 컨트롤러와 뷰 템플릿을 개발해보자.

BasicItemController

```
package hello.itemservice.web.item.basic;

import hello.itemservice.domain.item.Item;
import hello.itemservice.domain.item.ItemRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;
```

```
import javax.annotation.PostConstruct;
import java.util.List;

@Controller
@RequestMapping("/basic/items")
@RequiredArgsConstructor
public class BasicItemController {

    private final ItemRepository itemRepository;

    @GetMapping
    public String items(Model model) {
        List<Item> items = itemRepository.findAll();
        model.addAttribute("items", items);
        return "basic/items";
    }

    /**
     * 테스트용 데이터 추가
     */
    @PostConstruct
    public void init() {
        itemRepository.save(new Item("testA", 10000, 10));
        itemRepository.save(new Item("testB", 20000, 20));
    }
}
```

컨트롤러 로직은 itemRepository에서 모든 상품을 조회한 다음에 모델에 담는다. 그리고 뷰 템플릿을 호출한다.

- `@RequiredArgsConstructor`
  - `final`이 붙은 멤버변수만 사용해서 생성자를 자동으로 만들어준다.

```
public BasicItemController(ItemRepository itemRepository) {
  this.itemRepository = itemRepository;
}
```

  - 이렇게 생성자가 딱 1개만 있으면 스프링이 해당 생성자에 `@Autowired`로 의존관계를 주입해준다.

- 따라서 **final 키워드를 빼면 안된다!**, 그러면 `ItemRepository` 의존관계 주입이 안된다.
- 스프링 핵심원리 - 기본편 강의 참고

테스트용 데이터 추가

- 테스트용 데이터가 없으면 회원 목록 기능이 정상 동작하는지 확인하기 어렵다.
- `@PostConstruct` : 해당 빈의 의존관계가 모두 주입되고 나면 초기화 용도로 호출된다.
- 여기서는 간단히 테스트용 테이터를 넣기 위해서 사용했다.

items.html 정적 HTML을 뷰 템플릿(templates) 영역으로 복사하고 다음과 같이 수정하자

`/resources/static/items.html` ➡ 복사 ➡ `/resources/templates/basic/items.html`

`/resources/templates/basic/items.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css"
          th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>상품 목록</h2>
    </div>

    <div class="row">
        <div class="col">
            <button class="btn btn-primary float-end"
onclick="location.href='addForm.html'"
                    th:onclick="|location.href='@{/basic/items/add}'|"
type="button">상품 등록</button>
        </div>
    </div>

    <hr class="my-4">
    <div>
        <table class="table">
```

```
            <thead>
            <tr>
                <th>ID</th>
                <th>상품명</th>
                <th>가격</th>
                <th>수량</th>
            </tr>
            </thead>
            <tbody>
            <tr th:each="item : ${items}">
                <td><a href="item.html" th:href="@{/basic/items/{itemId}
(itemId=${item.id})}" th:text="${item.id}">회원id</a></td>
                <td><a href="item.html" th:href="@{|/basic/items/${item.id}|}"
th:text="${item.itemName}">상품명</a></td>
                <td th:text="${item.price}">10000</td>
                <td th:text="${item.quantity}">10</td>
            </tr>
            </tbody>
        </table>
    </div>

</div> <!-- /container -->

</body>
</html>
```

타임리프 간단히 알아보기

타임리프 사용 선언

`<html xmlns:th="http://www.thymeleaf.org">`

속성 변경 - th:href

`th:href="@{/css/bootstrap.min.css}"`

- `href="value1"` 을 `th:href="value2"` 의 값으로 변경한다.
- 타임리프 뷰 템플릿을 거치게 되면 원래 값을 `th:xxx` 값으로 변경한다. 만약 값이 없다면 새로 생성한다.
- HTML을 그대로 볼 때는 `href` 속성이 사용되고, 뷰 템플릿을 거치면 `th:href` 의 값이 `href` 로 대체되면서 동적으로 변경할 수 있다.

- 대부분의 HTML 속성을 `th:xxx` 로 변경할 수 있다.

타임리프 핵심

- 핵심은 `th:xxx` 가 붙은 부분은 서버사이드에서 렌더링 되고, 기존 것을 대체한다. `th:xxx` 이 없으면 기존 html의 `xxx` 속성이 그대로 사용된다.
- HTML을 파일로 직접 열었을 때, `th:xxx` 가 있어도 웹 브라우저는 `th:` 속성을 알지 못하므로 무시한다.
- 따라서 HTML을 파일 보기를 유지하면서 템플릿 기능도 할 수 있다.

URL 링크 표현식 - @{...},

`th:href="@{/css/bootstrap.min.css}"`

- `@{...}` : 타임리프는 URL 링크를 사용하는 경우 `@{...}` 를 사용한다. 이것을 URL 링크 표현식이라 한다.
- URL 링크 표현식을 사용하면 서블릿 컨텍스트를 자동으로 포함한다.

상품 등록 폼으로 이동

속성 변경 - th:onclick

- `onclick="location.href='addForm.html'"`
- `th:onclick="|location.href='@{/basic/items/add}'|"`

여기에는 다음에 설명하는 리터럴 대체 문법이 사용되었다. 자세히 알아보자.

리터럴 대체 - |...|

`|...|` :이렇게 사용한다.

- 타임리프에서 문자와 표현식 등은 분리되어 있기 때문에 더해서 사용해야 한다.
  - `<span th:text="'Welcome to our application, ' + ${user.name} + '!'">`
- 다음과 같이 리터럴 대체 문법을 사용하면, 더하기 없이 편리하게 사용할 수 있다.
  - `<span th:text="|Welcome to our application, ${user.name}!|">`

- 결과를 다음과 같이 만들어야 하는데
  - `location.href='/basic/items/add'`
- 그냥 사용하면 문자와 표현식을 각각 따로 더해서 사용해야 하므로 다음과 같이 복잡해진다.
  - `th:onclick="'location.href=' + '\'' + @{/basic/items/add} + '\''"`
- 리터럴 대체 문법을 사용하면 다음과 같이 편리하게 사용할 수 있다.
  - `th:onclick="|location.href='@{/basic/items/add}'|"`

반복 출력 - th:each

- `<tr th:each="item : ${items}">`
- 반복은 `th:each` 를 사용한다. 이렇게 하면 모델에 포함된 `items` 컬렉션 데이터가 `item` 변수에 하나씩 포함되고, 반복문 안에서 `item` 변수를 사용할 수 있다.
- 컬렉션의 수 만큼 `<tr>..</tr>` 이 하위 테그를 포함해서 생성된다.

변수 표현식 - ${...}

- `<td th:text="${item.price}">10000</td>`
- 모델에 포함된 값이나, 타임리프 변수로 선언한 값을 조회할 수 있다.
- 프로퍼티 접근법을 사용한다. (`item.getPrice()`)

내용 변경 - th:text

- `<td th:text="${item.price}">10000</td>`
- 내용의 값을 `th:text` 의 값으로 변경한다.
- 여기서는 10000을 `${item.price}` 의 값으로 변경한다.

URL 링크 표현식2 - @{...},

- `th:href="@{/basic/items/{itemId}(itemId=${item.id})}"`
- 상품 ID를 선택하는 링크를 확인해보자.
- URL 링크 표현식을 사용하면 경로를 템플릿처럼 편리하게 사용할 수 있다.
- 경로 변수(`{itemId}`) 뿐만 아니라 쿼리 파라미터도 생성한다.
- 예) `th:href="@{/basic/items/{itemId}(itemId=${item.id}, query='test')}"`
  - 생성 링크: `http://localhost:8080/basic/items/1?query=test`

URL 링크 간단히

- `th:href="@{|/basic/items/${item.id}|}"`
- 상품 이름을 선택하는 링크를 확인해보자.
- 리터럴 대체 문법을 활용해서 간단히 사용할 수도 있다.

> **참고**
> 타임리프는 순수 HTML을 파일을 웹 브라우저에서 열어도 내용을 확인할 수 있고, 서버를 통해 뷰 템플릿을 거치면 동적으로 변경된 결과를 확인할 수 있다. JSP를 생각해보면, JSP 파일은 웹 브라우저에서 그냥 열면 JSP 소스코드와 HTML이 뒤죽박죽 되어서 정상적인 확인이 불가능하다. 오직 서버를 통해서 JSP를 열어야 한다.
> 이렇게 **순수 HTML을 그대로 유지하면서 뷰 템플릿도 사용할 수 있는 타임리프의 특징을 네츄럴 템플릿** (natural templates)이라 한다.

상품 상세

상품 상세 컨트롤러와 뷰를 개발하자.

BasicItemController에 추가

```
@GetMapping("/{itemId}")
public String item(@PathVariable Long itemId, Model model) {
    Item item = itemRepository.findById(itemId);
    model.addAttribute("item", item);
    return "basic/item";
}
```

`PathVariable`로 넘어온 상품ID로 상품을 조회하고, 모델에 담아둔다. 그리고 뷰 템플릿을 호출한다.

상품 상세 뷰

정적 HTML을 뷰 템플릿(templates) 영역으로 복사하고 다음과 같이 수정하자.

`/resources/static/item.html` ➡ 복사 ➡ `/resources/templates/basic/item.html`

`/resources/templates/basic/item.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css"
          th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
    <style>
        .container {
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 상세</h2>
    </div>

    <div>
        <label for="itemId">상품 ID</label>
```

```
        <input type="text" id="itemId" name="itemId" class="form-control"
value="1" th:value="${item.id}" readonly>
    </div>
    <div>
        <label for="itemName">상품명</label>
        <input type="text" id="itemName" name="itemName" class="form-control"
value="상품A" th:value="${item.itemName}" readonly>
    </div>
    <div>
        <label for="price">가격</label>
        <input type="text" id="price" name="price" class="form-control"
value="10000" th:value="${item.price}" readonly>
    </div>
    <div>
        <label for="quantity">수량</label>
        <input type="text" id="quantity" name="quantity" class="form-control"
value="10" th:value="${item.quantity}" readonly>
    </div>

    <hr class="my-4">

    <div class="row">
        <div class="col">
            <button class="w-100 btn btn-primary btn-lg"
onclick="location.href='editForm.html'"
                    th:onclick="|location.href='@{/basic/items/{itemId}/
edit(itemId=${item.id})}'|" type="button">상품 수정</button>
        </div>
        <div class="col">
            <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='items.html'"
                    th:onclick="|location.href='@{/basic/items}'|"
                    type="button">목록으로</button>
        </div>
    </div>

</div> <!-- /container -->
</body>
</html>
```

속성 변경 - th:value

`th:value="${item.id}"`

- 모델에 있는 item 정보를 획득하고 프로퍼티 접근법으로 출력한다. (`item.getId()`)
- `value` 속성을 `th:value` 속성으로 변경한다.

상품수정 링크

- `th:onclick="|location.href='@{/basic/items/{itemId}/edit(itemId=${item.id})}'|"`

목록으로 링크

- `th:onclick="|location.href='@{/basic/items}'|"`

상품 등록 폼

상품 등록 폼

BasicItemController에 추가

```
@GetMapping("/add")
public String addForm() {
    return "basic/addForm";
}
```

상품 등록 폼은 단순히 뷰 템플릿만 호출한다.

상품 등록 폼 뷰

정적 HTML을 뷰 템플릿(templates) 영역으로 복사하고 다음과 같이 수정하자.

`/resources/static/addForm.html` ➡ 복사 ➡ `/resources/templates/basic/addForm.html`

`/resources/templates/basic/addForm.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
```

```
    <link href="../css/bootstrap.min.css"
          th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
    <style>
        .container {
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 등록 폼</h2>
    </div>

    <h4 class="mb-3">상품 입력</h4>

    <form action="item.html" th:action method="post">
        <div>
            <label for="itemName">상품명</label>
            <input type="text" id="itemName" name="itemName" class="form-
control" placeholder="이름을 입력하세요">
        </div>
        <div>
            <label for="price">가격</label>
            <input type="text" id="price" name="price" class="form-control"
placeholder="가격을 입력하세요">
        </div>
        <div>
            <label for="quantity">수량</label>
            <input type="text" id="quantity" name="quantity" class="form-
control" placeholder="수량을 입력하세요">
        </div>

        <hr class="my-4">

        <div class="row">
            <div class="col">
```

```
                <button class="w-100 btn btn-primary btn-lg" type="submit">상품
등록</button>
            </div>
            <div class="col">
                <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='items.html'"
                        th:onclick="|location.href='@{/basic/items}'|"
                        type="button">취소</button>
            </div>
        </div>

    </form>

</div> <!-- /container -->
</body>
</html>
```

속성 변경 - th:action

- `th:action`
- HTML form에서 `action`에 값이 없으면 현재 URL에 데이터를 전송한다.
- 상품 등록 폼의 URL과 실제 상품 등록을 처리하는 URL을 똑같이 맞추고 HTTP 메서드로 두 기능을 구분한다.
  - 상품 등록 폼: GET `/basic/items/add`
  - 상품 등록 처리: POST `/basic/items/add`
- 이렇게 하면 하나의 URL로 등록 폼과, 등록 처리를 깔끔하게 처리할 수 있다.

취소

- 취소시 상품 목록으로 이동한다.
- `th:onclick="|location.href='@{/basic/items}'|"`

상품 등록 처리 - @ModelAttribute

이제 상품 등록 폼에서 전달된 데이터로 실제 상품을 등록 처리해보자.

상품 등록 폼은 다음 방식으로 서버에 데이터를 전달한다.

- **POST - HTML Form**

- `content-type: application/x-www-form-urlencoded`
- 메시지 바디에 쿼리 파리미터 형식으로 전달 `itemName=itemA&price=10000&quantity=10`
- 예) 회원 가입, 상품 주문, HTML Form 사용

요청 파라미터 형식을 처리해야 하므로 `@RequestParam` 을 사용하자

상품 등록 처리 - @RequestParam

addItemV1 - BasicItemController에 추가

```
@PostMapping("/add")
public String addItemV1(@RequestParam String itemName,
                        @RequestParam int price,
                        @RequestParam Integer quantity,
                        Model model) {

    Item item = new Item();
    item.setItemName(itemName);
    item.setPrice(price);
    item.setQuantity(quantity);

    itemRepository.save(item);

    model.addAttribute("item", item);

    return "basic/item";
}
```

- 먼저 `@RequestParam String itemName` : itemName 요청 파라미터 데이터를 해당 변수에 받는다.
- `Item` 객체를 생성하고 `itemRepository` 를 통해서 저장한다.
- 저장된 `item` 을 모델에 담아서 뷰에 전달한다.

중요: 여기서는 상품 상세에서 사용한 `item.html` 뷰 템플릿을 그대로 재활용한다.

실행해서 상품이 잘 저장되는지 확인하자.

상품 등록 처리 - @ModelAttribute

`@RequestParam`으로 변수를 하나하나 받아서 `Item`을 생성하는 과정은 불편했다.
이번에는 `@ModelAttribute`를 사용해서 한번에 처리해보자.

addItemV2 - 상품 등록 처리 - ModelAttribute

```
/**
 * @ModelAttribute("item") Item item
 * model.addAttribute("item", item); 자동 추가
 */
@PostMapping("/add")
public String addItemV2(@ModelAttribute("item") Item item, Model model) {
    itemRepository.save(item);
    //model.addAttribute("item", item); //자동 추가, 생략 가능
    return "basic/item";
}
```

@ModelAttribute - 요청 파라미터 처리

`@ModelAttribute`는 `Item`객체를 생성하고, 요청 파라미터의 값을 프로퍼티 접근법(setXxx)으로 입력해준다.

@ModelAttribute - Model 추가

`@ModelAttribute`는 중요한 한가지 기능이 더 있는데, 바로 모델(Model)에 `@ModelAttribute`로 지정한 객체를 자동으로 넣어준다. 지금 코드를 보면 `model.addAttribute("item", item)`가 주석처리 되어 있어도 잘 동작하는 것을 확인할 수 있다.

모델에 데이터를 담을 때는 이름이 필요하다. 이름은 `@ModelAttribute`에 지정한 `name(value)` 속성을 사용한다. 만약 다음과 같이 `@ModelAttribute`의 이름을 다르게 지정하면 다른 이름으로 모델에 포함된다.

`@ModelAttribute("hello") Item item` ➡ 이름을 `hello`로 지정
`model.addAttribute("hello", item);` ➡ 모델에 `hello` 이름으로 저장

주의

실행전에 이전 버전인 `addItemV1`에 `@PostMapping("/add")`를 꼭 주석처리 해주어야 한다. 그렇지 않으면 중복 매핑으로 오류가 발생한다.

```
//@PostMapping("/add") 이전 코드의 매핑 주석처리!
public String addItemV1(@RequestParam String itemName,
```

addItemV3 - 상품 등록 처리 - ModelAttribute 이름 생략

```
/**
 * @ModelAttribute name 생략 가능
 * model.addAttribute(item); 자동 추가, 생략 가능
 * 생략시 model에 저장되는 name은 클래스명 첫글자만 소문자로 등록 Item -> item
 */
@PostMapping("/add")
public String addItemV3(@ModelAttribute Item item) {
    itemRepository.save(item);
    return "basic/item";
}
```

`@ModelAttribute`의 이름을 생략할 수 있다.

주의

`@ModelAttribute`의 이름을 생략하면 모델에 저장될 때 클래스명을 사용한다. 이때 **클래스의 첫글자만 소문자로 변경**해서 등록한다.

- 예) `@ModelAttribute` 클래스명 ➡ 모델에 자동 추가되는 이름
  - `Item` ➡ `item`
  - `HelloWorld` ➡ `helloWorld`

addItemV4 - 상품 등록 처리 - ModelAttribute 전체 생략

```
/**
 * @ModelAttribute 자체 생략 가능
 * model.addAttribute(item) 자동 추가
 */
@PostMapping("/add")
public String addItemV4(Item item) {
    itemRepository.save(item);
    return "basic/item";
```

```
}
```

`@ModelAttribute` 자체도 생략가능하다. 대상 객체는 모델에 자동 등록된다. 나머지 사항은 기존과 동일하다.

상품 수정

상품 수정 폼 컨트롤러

BasicItemController에 추가

```
@GetMapping("/{itemId}/edit")
public String editForm(@PathVariable Long itemId, Model model) {
    Item item = itemRepository.findById(itemId);
    model.addAttribute("item", item);
    return "basic/editForm";
}
```

수정에 필요한 정보를 조회하고, 수정용 폼 뷰를 호출한다.

상품 수정 폼 뷰

정적 HTML을 뷰 템플릿(templates) 영역으로 복사하고 다음과 같이 수정하자.

`/resources/static/editForm.html` ➡ 복사 ➡ `/resources/templates/basic/editForm.html`

`/resources/templates/basic/editForm.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
    <link href="../css/bootstrap.min.css"
          th:href="@{/css/bootstrap.min.css}" rel="stylesheet">
    <style>
        .container {
```

```
            max-width: 560px;
        }
    </style>
</head>
<body>

<div class="container">

    <div class="py-5 text-center">
        <h2>상품 수정 폼</h2>
    </div>

    <form action="item.html" th:action method="post">
        <div>
            <label for="id">상품 ID</label>
            <input type="text" id="id" name="id" class="form-control" value="1"
th:value="${item.id}" readonly>
        </div>
        <div>
            <label for="itemName">상품명</label>
            <input type="text" id="itemName" name="itemName" class="form-
control" value="상품A" th:value="${item.itemName}">
        </div>
        <div>
            <label for="price">가격</label>
            <input type="text" id="price" name="price" class="form-control"
th:value="${item.price}">
        </div>
        <div>
            <label for="quantity">수량</label>
            <input type="text" id="quantity" name="quantity" class="form-
control" th:value="${item.quantity}">
        </div>

        <hr class="my-4">

        <div class="row">
            <div class="col">
                <button class="w-100 btn btn-primary btn-lg" type="submit">저장
```

```
</button>
            </div>
            <div class="col">
                <button class="w-100 btn btn-secondary btn-lg"
onclick="location.href='item.html'"
   th:onclick="|location.href='@{/basic/items/{itemId}(itemId=${item.id})}'|"
                        type="button">취소</button>
            </div>
        </div>

    </form>

</div> <!-- /container -->
</body>
</html>
```

상품 수정 폼은 상품 등록과 유사하고, 특별한 내용이 없다.

상품 수정 개발

```
@PostMapping("/{itemId}/edit")
public String edit(@PathVariable Long itemId, @ModelAttribute Item item) {
    itemRepository.update(itemId, item);
    return "redirect:/basic/items/{itemId}";
}
```

상품 수정은 상품 등록과 전체 프로세스가 유사하다.

- GET `/items/{itemId}/edit` : 상품 수정 폼
- POST `/items/{itemId}/edit` : 상품 수정 처리

리다이렉트

상품 수정은 마지막에 뷰 템플릿을 호출하는 대신에 상품 상세 화면으로 이동하도록 리다이렉트를 호출한다.

- 스프링은 `redirect:/...`으로 편리하게 리다이렉트를 지원한다.
- `redirect:/basic/items/{itemId}"`
  - 컨트롤러에 매핑된 `@PathVariable`의 값은 `redirect`에도 사용 할 수 있다.

- `redirect:/basic/items/{itemId}` ➡ `{itemId}`는 `@PathVariable Long itemId`의 값을 그대로 사용한다.

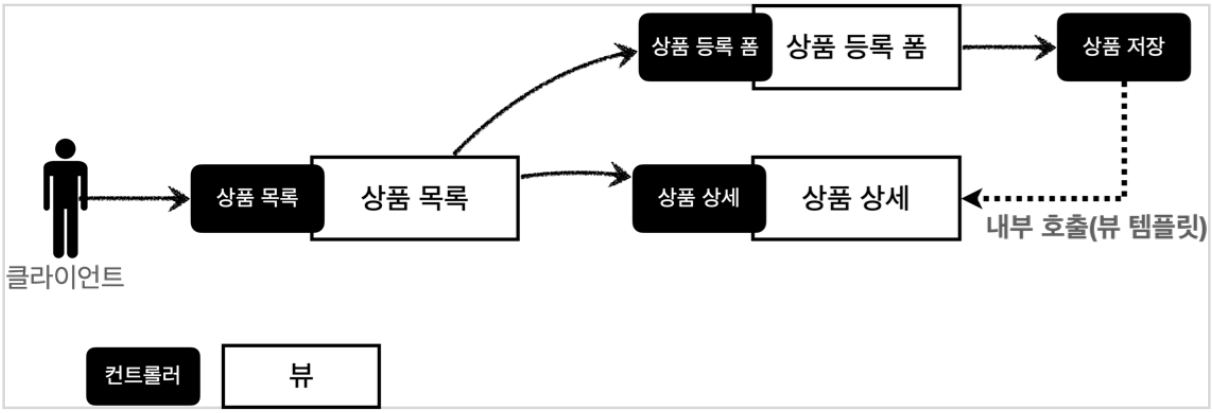
> **참고**
> 리다이렉트에 대한 자세한 내용은 모든 개발자를 위한 HTTP 웹 기본 지식 강의를 참고하자.

> **참고**
> HTML Form 전송은 PUT, PATCH를 지원하지 않는다. GET, POST만 사용할 수 있다.
> PUT, PATCH는 HTTP API 전송시에 사용
> 스프링에서 HTTP POST로 Form 요청할 때 히든 필드를 통해서 PUT, PATCH 매핑을 사용하는 방법이 있지만, HTTP 요청상 POST 요청이다.

PRG Post/Redirect/Get

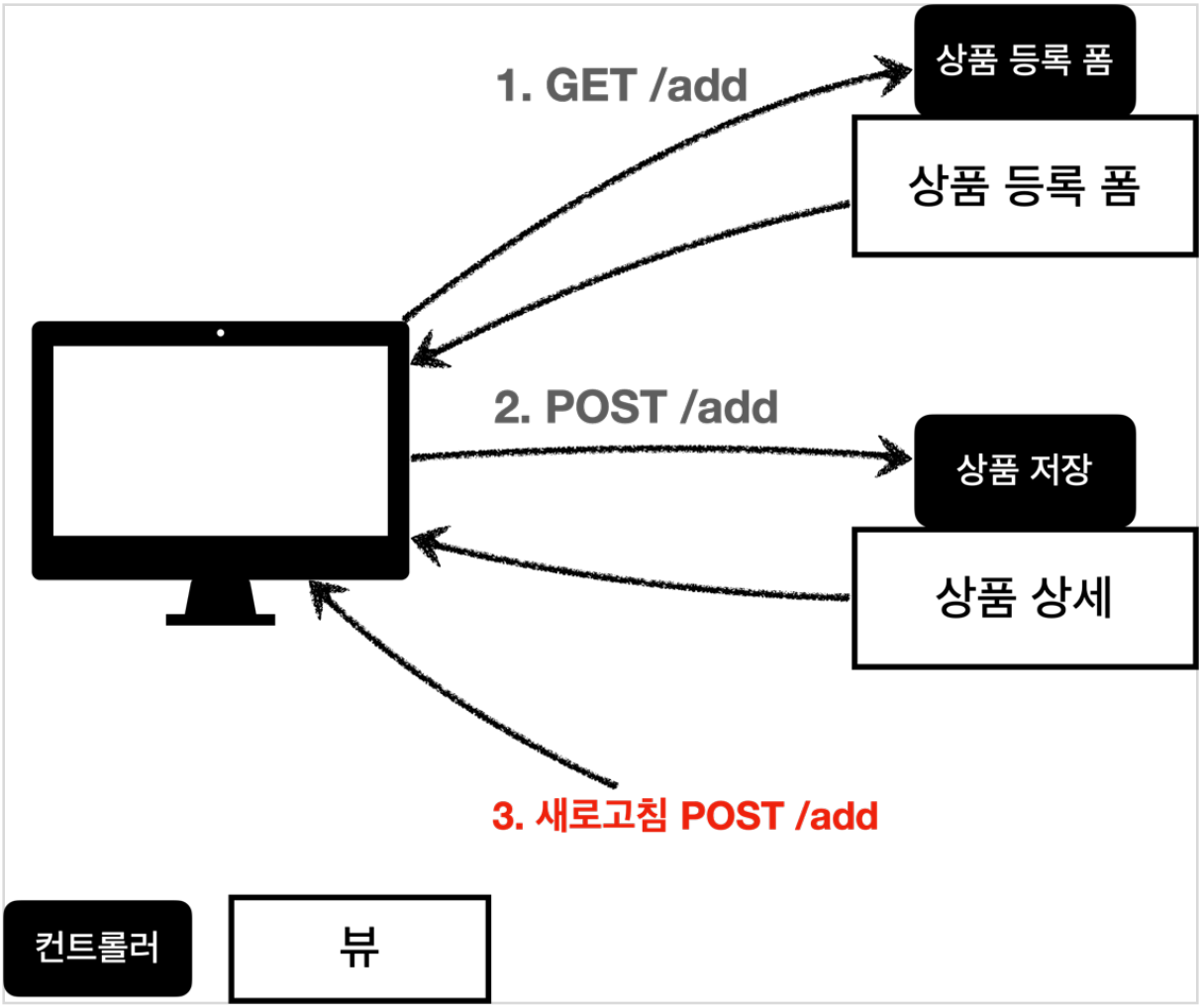
사실 지금까지 진행한 상품 등록 처리 컨트롤러는 심각한 문제가 있다. (addItemV1 ~ addItemV4)
상품 등록을 완료하고 웹 브라우저의 새로고침 버튼을 클릭해보자.
상품이 계속해서 중복 등록되는 것을 확인할 수 있다.

전체 흐름



그 이유는 다름 그림을 통해서 확인할 수 있다.

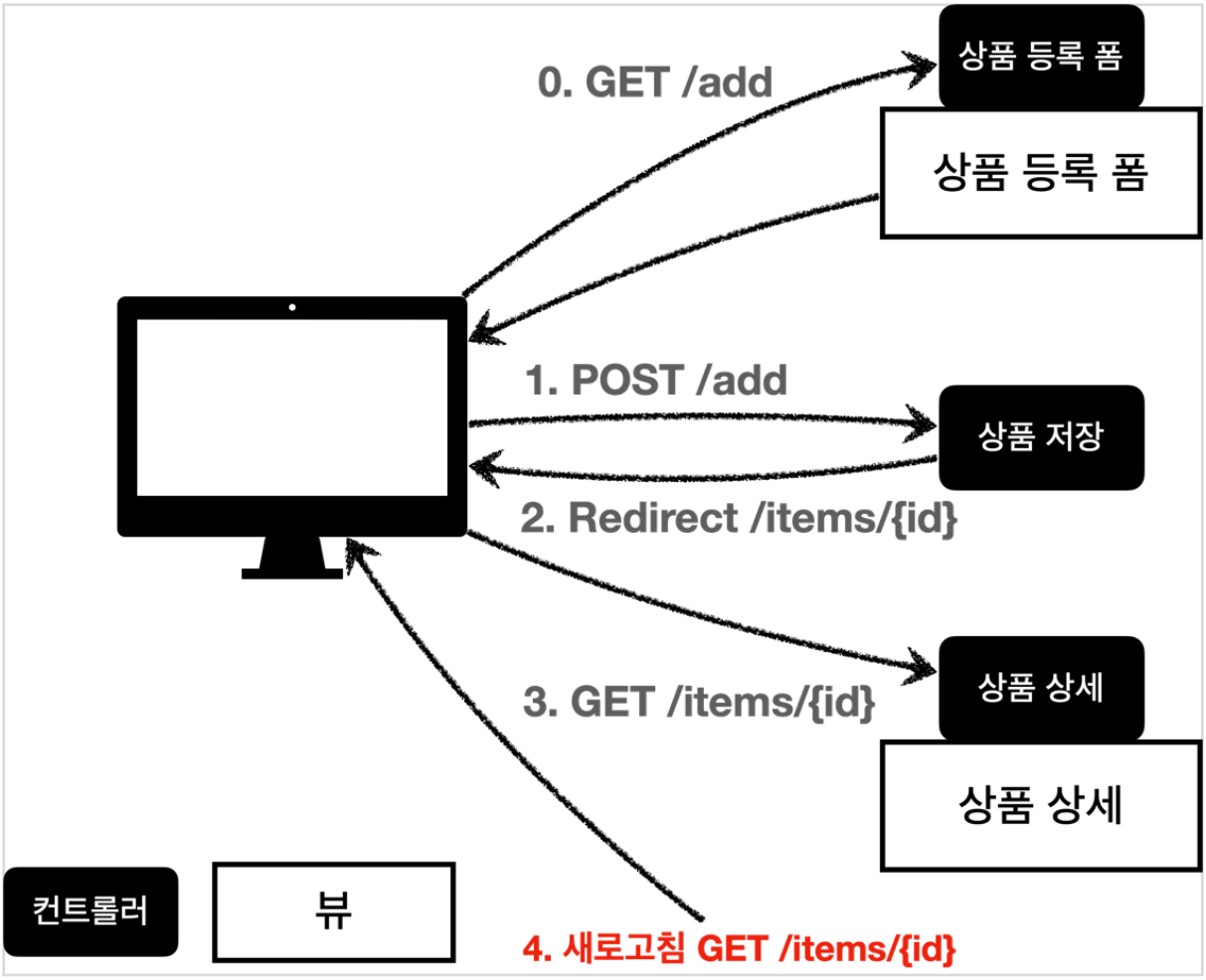
POST 등록 후 새로 고침



웹 브라우저의 새로 고침은 마지막에 서버에 전송한 데이터를 다시 전송한다.
상품 등록 폼에서 데이터를 입력하고 저장을 선택하면 `POST /add` + 상품 데이터를 서버로 전송한다.
이 상태에서 새로 고침을 또 선택하면 마지막에 전송한 `POST /add` + 상품 데이터를 서버로 다시 전송하게 된다.
그래서 내용은 같고, ID만 다른 상품 데이터가 계속 쌓이게 된다.

이 문제를 어떻게 해결할 수 있을까? 다음 그림을 보자.

POST, Redirect GET



웹 브라우저의 새로 고침은 마지막에 서버에 전송한 데이터를 다시 전송한다.
새로 고침 문제를 해결하려면 상품 저장 후에 뷰 템플릿으로 이동하는 것이 아니라, 상품 상세 화면으로 리다이렉트를 호출해주면 된다.
웹 브라우저는 리다이렉트의 영향으로 상품 저장 후에 실제 상품 상세 화면으로 다시 이동한다. 따라서 마지막에 호출한 내용이 상품 상세 화면인 `GET /items/{id}` 가 되는 것이다.
이후 새로고침을 해도 상품 상세 화면으로 이동하게 되므로 새로 고침 문제를 해결할 수 있다.

BasicItemController에 추가

```
/**
 * PRG - Post/Redirect/Get
 */
@PostMapping("/add")
public String addItemV5(Item item) {
    itemRepository.save(item);
    return "redirect:/basic/items/" + item.getId();
}
```

상품 등록 처리 이후에 뷰 템플릿이 아니라 상품 상세 화면으로 리다이렉트 하도록 코드를 작성해보자.
이런 문제 해결 방식을 `PRG Post/Redirect/Get` 라 한다.

주의

> `"redirect:/basic/items/" + item.getId()` redirect에서 `+item.getId()` 처럼 URL에 변수를 더해서 사용하는 것은 URL 인코딩이 안되기 때문에 위험하다. 다음에 설명하는 `RedirectAttributes` 를 사용하자.

RedirectAttributes

상품을 저장하고 상품 상세 화면으로 리다이렉트 한 것 까지는 좋았다. 그런데 고객 입장에서 저장이 잘 된 것인지 안 된 것인지 확신이 들지 않는다. 그래서 저장이 잘 되었으면 상품 상세 화면에 "저장되었습니다"라는 메시지를 보여달라는 요구사항이 왔다. 간단하게 해결해보자.

BasicItemController에 추가

```
/**
 * RedirectAttributes
 */
@PostMapping("/add")
public String addItemV6(Item item, RedirectAttributes redirectAttributes) {
    Item savedItem = itemRepository.save(item);
    redirectAttributes.addAttribute("itemId", savedItem.getId());
    redirectAttributes.addAttribute("status", true);
    return "redirect:/basic/items/{itemId}";
}
```

리다이렉트 할 때 간단히 `status=true` 를 추가해보자. 그리고 뷰 템플릿에서 이 값이 있으면, 저장되었습니다. 라는 메시지를 출력해보자.

실행해보면 다음과 같은 리다이렉트 결과가 나온다.
`http://localhost:8080/basic/items/3?status=true`

RedirectAttributes

`RedirectAttributes` 를 사용하면 URL 인코딩도 해주고, `pathVarible` , 쿼리 파라미터까지 처리해준다.

- `redirect:/basic/items/{itemId}`
  - pathVariable 바인딩: `{itemId}`
  - 나머지는 쿼리 파라미터로 처리: `?status=true`

뷰 템플릿 메시지 추가

`resources/templates/basic/item.html`

```
<div class="container">

    <div class="py-5 text-center">
        <h2>상품 상세</h2>

    </div>

    <!-- 추가 -->
    <h2 th:if="${param.status}" th:text="'저장 완료!'"></h2>
```

- `th:if` : 해당 조건이 참이면 실행
- `${param.status}` : 타임리프에서 쿼리 파라미터를 편리하게 조회하는 기능
  - 원래는 컨트롤러에서 모델에 직접 담고 값을 꺼내야 한다. 그런데 쿼리 파라미터는 자주 사용해서 타임리프에서 직접 지원한다.

뷰 템플릿에 메시지를 추가하고 실행해보면 "저장 완료!" 라는 메시지가 나오는 것을 확인할 수 있다. 물론 상품 목록에서 상품 상세로 이동한 경우에는 해당 메시지가 출력되지 않는다.

정리

8. 다음으로

#인강/4. 스프링 MVC 1/강의#

학습 내용 정리

전체 목차

- 1. 웹 애플리케이션 이해
- 2. 서블릿
- 3. 서블릿, JSP, MVC 패턴
- 4. MVC 프레임워크 만들기
- 5. 스프링 MVC - 구조 이해
- 6. 스프링 MVC - 기본 기능
- 7. 스프링 MVC - 웹 페이지 만들기

1. 웹 애플리케이션 이해

- 웹 서버, 웹 애플리케이션 서버
- 서블릿
- 동시 요청 - 멀티 쓰레드
- HTML, HTTP API, CSR, SSR
- 자바 백엔드 웹 기술 역사

2. 서블릿

- 2. 서블릿 - 프로젝트 생성
- 2. 서블릿 - Hello 서블릿
- 2. 서블릿 - HttpServletRequest - 개요
- 2. 서블릿 - HttpServletRequest - 기본 사용법
- 2. 서블릿 - HTTP 요청 데이터 - 개요
- 2. 서블릿 - HTTP 요청 데이터 - GET 쿼리 파라미터
- 2. 서블릿 - HTTP 요청 데이터 - POST HTML Form
- 2. 서블릿 - HTTP 요청 데이터 - API 메시지 바디 - 단순 텍스트
- 2. 서블릿 - HTTP 요청 데이터 - API 메시지 바디 - JSON
- 2. 서블릿 - HttpServletResponse - 기본 사용법
- 2. 서블릿 - HTTP 응답 데이터 - 단순 텍스트, HTML
- 2. 서블릿 - HTTP 응답 데이터 - API JSON

3. 서블릿, JSP, MVC 패턴

- 3. 서블릿, JSP, MVC 패턴 - 회원 관리 웹 애플리케이션 요구사항
- 3. 서블릿, JSP, MVC 패턴 - 서블릿으로 회원 관리 웹 애플리케이션 만들기
- 3. 서블릿, JSP, MVC 패턴 - JSP로 회원 관리 웹 애플리케이션 만들기
- 3. 서블릿, JSP, MVC 패턴 - MVC 패턴 - 개요
- 3. 서블릿, JSP, MVC 패턴 - MVC 패턴 - 적용
- 3. 서블릿, JSP, MVC 패턴 - MVC 패턴 - 한계

4. MVC 프레임워크 만들기

- 4. MVC 프레임워크 만들기 - 프론트 컨트롤러 패턴 소개
- 4. MVC 프레임워크 만들기 - 프론트 컨트롤러 도입 - v1
- 4. MVC 프레임워크 만들기 - View 분리 - v2
- 4. MVC 프레임워크 만들기 - Model 추가 - v3
- 4. MVC 프레임워크 만들기 - 단순하고 실용적인 컨트롤러 - v4
- 4. MVC 프레임워크 만들기 - 유연한 컨트롤러1 - v5
- 4. MVC 프레임워크 만들기 - 유연한 컨트롤러2 - v5

5. 스프링 MVC - 구조 이해

- 5. 스프링 MVC - 구조 이해 - 스프링 MVC 전체 구조
- 5. 스프링 MVC - 구조 이해 - 핸들러 매핑과 핸들러 어댑터
- 5. 스프링 MVC - 구조 이해 - 뷰 리졸버
- 5. 스프링 MVC - 구조 이해 - 스프링 MVC - 시작하기
- 5. 스프링 MVC - 구조 이해 - 스프링 MVC - 컨트롤러 통합
- 5. 스프링 MVC - 구조 이해 - 스프링 MVC - 실용적인 방식

6. 스프링 MVC - 기본 기능

- 6. 스프링 MVC - 기본 기능 - 프로젝트 생성
- 6. 스프링 MVC - 기본 기능 - 로깅 간단히 알아보기
- 6. 스프링 MVC - 기본 기능 - 요청 매핑
- 6. 스프링 MVC - 기본 기능 - 요청 매핑 - API 예시
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 - 기본, 헤더 조회
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 파라미터 - 쿼리 파라미터, HTML Form
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 파라미터 - @RequestParam
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 파라미터 - @ModelAttribute
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 메시지 - 단순 텍스트
- 6. 스프링 MVC - 기본 기능 - HTTP 요청 메시지 - JSON
- 6. 스프링 MVC - 기본 기능 - HTTP 응답 - 정적 리소스, 뷰 템플릿
- 6. 스프링 MVC - 기본 기능 - HTTP 응답 - HTTP API, 메시지 바디에 직접 입력
- 6. 스프링 MVC - 기본 기능 - HTTP 메시지 컨버터
- 6. 스프링 MVC - 기본 기능 - 요청 매핑 헨들러 어뎁터 구조

7. 스프링 MVC - 웹 페이지 만들기

- 7. 스프링 MVC - 웹 페이지 만들기 - 프로젝트 생성
- 7. 스프링 MVC - 웹 페이지 만들기 - 요구사항 분석
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 도메인 개발
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 서비스 HTML
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 목록 - 타임리프
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 상세
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 등록 폼
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 등록 처리 - @ModelAttribute
- 7. 스프링 MVC - 웹 페이지 만들기 - 상품 수정
- 7. 스프링 MVC - 웹 페이지 만들기 - PRG Post/Redirect/Get
- 7. 스프링 MVC - 웹 페이지 만들기 - RedirectAttributes

다음으로

스프링 MVC 2편 - 백엔드 웹 개발 활용 기술 안내

실제 예제에 단계적으로 기능을 발전시키며, 각 기능을 코드로 개발하면서 자연스럽게 학습

- 타임리프 뷰 템플릿 주요 기능 정리, 활용
  - 타임리프가 제공하는 기능 없이 사용할 때
  - 타임리프가 제공하는 기능을 학습하고 사용할 때 차이
  - 타임리프 뷰 템플릿 기능 정리, 다양한 기능 학습
- 메시지, 국제화 처리
  - 스프링의 메시지 처리 메커니즘 이해
  - 메시지, 국제화 처리 소개 및 구현
- 검증 - Validation
  - 컨트롤러에서 직접 검증 메커니즘 구현
  - 스프링이 제공하는 검증 메커니즘을 단계적으로 학습하고 적용
  - 수동 검증에서 애노테이션 기반의 검증까지
- 로그인 처리 - 쿠키, 세션
  - WAS가 제공하는 세션을 사용하지 않고, 직접 세션 구현해서 사용하기
  - WAS가 제공하는 세션 사용하기
- 로그인 처리 - 필터, 인터셉터
  - 개념과 차이 설명
  - 사용 예시
- 예외 처리

    - 예외 처리 매커니즘은 사실 매우 복잡해서 제대로 파악하기 쉽지 않다.
    - 제대로 파악하려면 서블릿의 예외처리 부터 스프링 부트가 제공하는 예외처리까지 복합적으로 이해해야 함
    - 순수한 서블릿 예외 처리 이해
    - 스프링 MVC가 제공하는 예외 처리 이해
    - 스프링 부트가 제공하는 예외 처리 이해
    - API 예외 처리
    - 예외 처리 완전 정복
- 자주 사용하는 기능, 기타
    - 나머지

여기까지 학습하면 스프링 MVC 완성!
실무 개발자들 보다 더 깊이있는 이해

로드맵 소개

스프링 완전 정복 시리즈(진행중)

스프링을 완전히 마스터 할 수 있는 로드맵

강의 목록

- 스프링 입문 - 코드로 배우는 스프링 부트, 웹 MVC, DB 접근 기술
- 스프링 핵심 원리 - 기본편
- 모든 개발자를 위한 HTTP 웹 기본 지식
- 스프링 웹 MVC 1편
- 스프링 웹 MVC 2편(5월)
- 스프링 DB 접근 기술 (예정)
- 실전! 스프링 부트 (예정)

실전 REST API 설계, 개발 강의 요청이 많아서, 고민중

스프링 부트와 JPA 실무 완전 정복 로드맵

최신 실무 기술로 웹 애플리케이션을 만들어보면서 학습하고 싶으면 스프링 부트와 JPA 실무 완전 정복

로드맵을 추천

URL: https://www.inflearn.com/roadmaps/149

강의 목록

- 자바 ORM 표준 JPA 프로그래밍
- 실전! 스프링 부트와 JPA 활용1 - 웹 애플리케이션 개발
- 실전! 스프링 부트와 JPA 활용2 - API 개발과 성능 최적화
- 실전! 스프링 데이터 JPA
- 실전! Querydsl

Q: 전체 코스 순서는 어떻게 듣는게 좋나요?

목표: 최신 기술의 자바 백엔드 개발자

- 스프링 완전 정복 로드맵
  - 스프링 입문 - 코드로 배우는 스프링 부트, 웹 MVC, DB 접근 기술
  - 스프링 핵심 원리 - 기본편
  - 모든 개발자를 위한 HTTP 웹 기본 지식
  - 스프링 웹 MVC 1편
  - 스프링 웹 MVC 2편
- 스프링 부트와 JPA 실무 완전 정복 로드맵

- **학자형 코스 vs 야생형 코스**
  - 학자형 코스: 이론을 먼저 차근차근 쌓아올린 다음에 실무 활용으로 넘어가는 방식
  - 야생형 코스: 일단 실무에서 어떤 기술을 어떤식으로 활용하는지, 깊이가 부족해도 일단 경험해본 다음에 필요에 의해서 이론 기술들을 공부하는 방식

추천 학습 방법

스프링 입문과 스프링 핵심 원리를 듣고 나면 스프링으로 개발하는 가장 중요한 기본 지식을 쌓은 상태가 됩니다. 그래서 바로 **스프링 부트와 JPA 실무 완전 정복 로드맵**에 들어가는 것도 좋은 선택입니다.
스프링을 실무에서 어떤 식으로 활용해서 개발하는지 먼저 배워두고, 이후에 **스프링 완전정복 로드맵**를 통해 스프링 MVC나 스프링 데이터 접근 기술 같은 부분은 더 깊이있게 학습하시면 됩니다.

문제 해결 방안과 야생형

야생형 코스가 맞는 사람도 있고, 학자형 코스가 맞는 사람도 있음