

본 강의 자료는 저작권법에 의해 보호받고 있습니다.

이 자료는 오직 교육 목적으로만 사용되어야 하며, 무단 복제, 배포, 상업적 사용이 금지되며 저작권자 허가를 받은 후에 인터넷에 게시해야 합니다.

강의 내용과 자료는 저작권자의 명시적 허가 없이 공유되어서는 안 됩니다.

모든 저작권은 해당 저작권자에게 귀속됩니다.



스프링 웹 MVC 완전 정복

Lecturer 정 수 원
<https://github.com/onjsdnjs>

✓ 개설 강의 목록

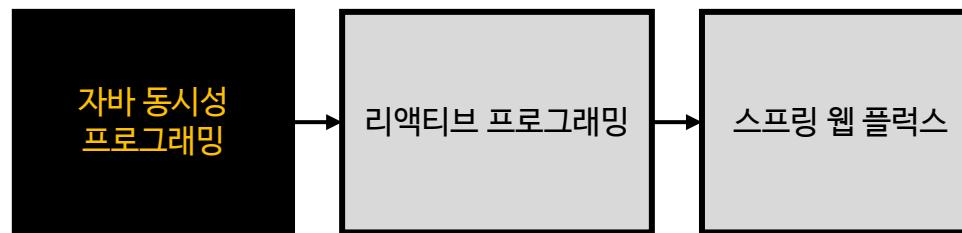


✓ 강의 로드맵

Synchronous Programming



Asynchronous Programming



Object Oriented Programming





스프링 웹 MVC 완전 정복

✓ 강좌 소개

1. 커리큘럼 소개
2. 개발환경 & 필요사항

커리큘럼 소개

- 1) 자바 웹 변천사 - 서블릿, 모델1, 모델2
- 2) 프론트 컨트롤러 패턴 - 프론트 컨트롤러 패턴과 스프링 MVC
- 3) 서블릿 이해 - 서블릿개념, 이벤트 및 생명주기, 요청 프로세스, 컨테이너 동작원리 등
- 4) 서블릿과 스프링 - 서블릿 컨테이너와 스프링 컨테이너 연결 구조, 스프링 구동 원리
- 5) 스프링 웹 MVC 초기화 - 스프링 부트 기반 초기화 과정, 서블릿 컨테이너 & 스프링 컨테이너, 초기화 클래스들
- 6) 스프링 웹 MVC 기본 - 아키텍처, DispatcherServlet 외 핵심 클래스
- 7) 스프링 웹 MVC 활용 - 요청 프로세스, 핸들러 메서드, 다양한 어노테이션 기법, 내부 실행 구조 분석
- 8) 타입변환 - 데이터 변환 방법과 적용하기, Converter, ConversionService, Formatter
- 9) 검증 - 폼 검증(Form Validation), Validator 검증, Bean Validation
- 10) 스프링 웹 MVC 실행 구조 이해 - 자바 리플렉션 이해, 요청 매핑 및 메서드 파라미터 실행 원리 및 구조 이해
- 11) 스프링 웹 MVC 공통 기능 - 인터셉터, @ControllerAdvice
- 12) 예외 처리 - 예외 처리 종류와 방법, @ExceptionHandler, HandlerExceptionResolver
- 13) Multipart - 스프링의 파일 처리 개념, 파일 업로드, @RequestPart
- 14) Rest Clients - RestClient, HTTP Interface

개발환경 & 필요사항

✓ 개발환경

- JDK 21 이상
- Gradle
- 인텔리제이, STS

✓ 선수지식

- Java
- Spring framework
- Spring Boot



스프링 웹 MVC 완전 정복

✓ 자바 웹 변천사

1. 서블릿 방식 - Servlet
2. 모델1 방식 - JSP
3. 모델2 방식 - MVC

서블릿 방식 - Servlet

<https://github.com/onjsdnjs/servlet/tree/servlet>

✓ 개요

- 서블릿은 자바 기반의 웹 애플리케이션 개발 초기 단계에서 사용된 기술로 자바로 작성된 서버 측 프로그램으로 사용되었다
- 클라이언트의 요청을 처리하고 동적 콘텐츠를 생성하는데 사용되었는데 주로 HTTP 요청과 응답을 처리하며 자바 클래스 형태로 HTTP 요청을 처리한다

✓ 서블릿 컨테이너에서 실행

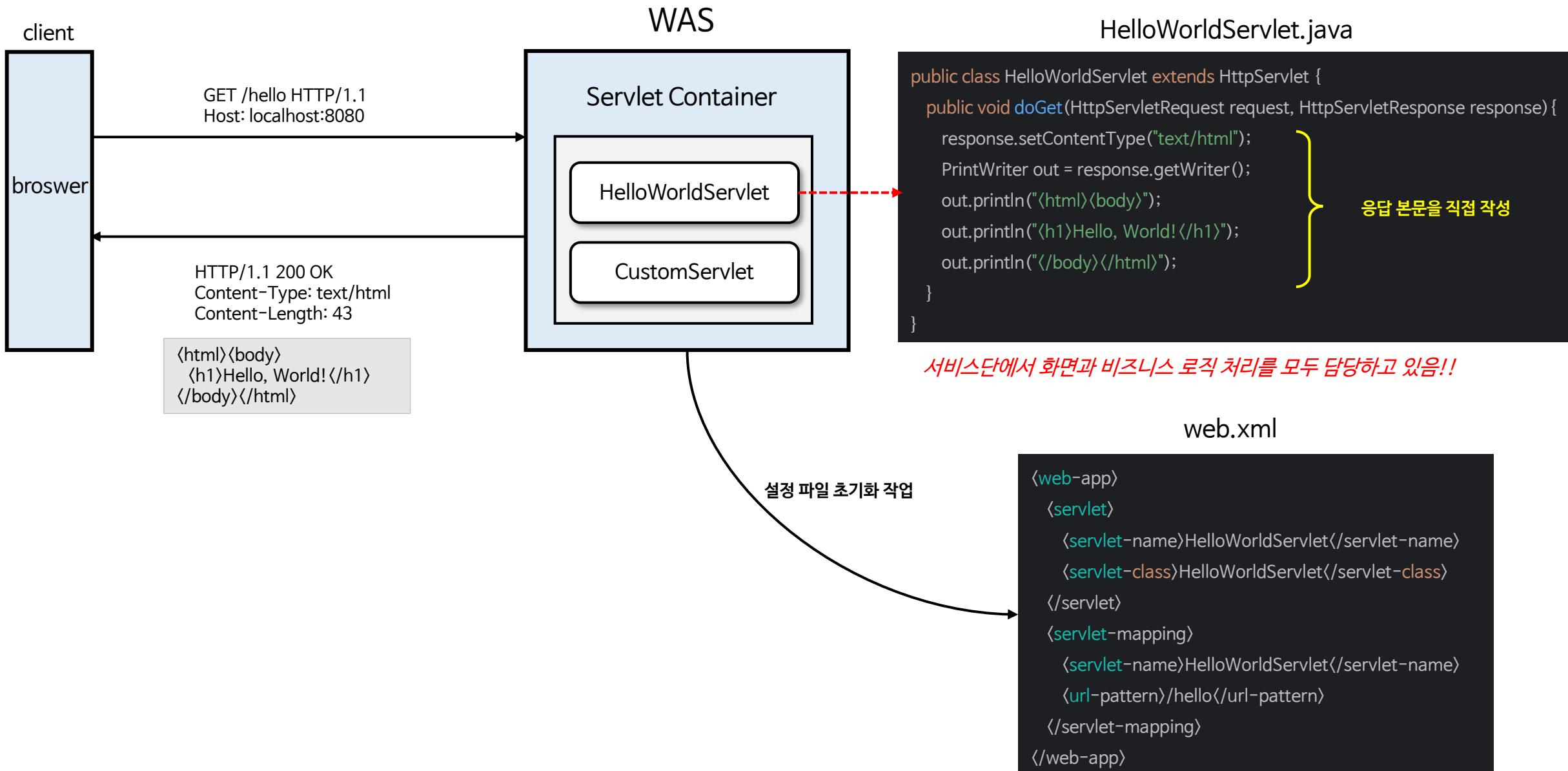
- 서블릿이 실행되는 환경으로, 서블릿을 관리하고 요청을 처리한다.
- Apache Tomcat, JBoss, JBoss 와 같은 WAS(Web Application Server)에서 실행되는 환경이다

✓ 특징

- 클래스 단위로 직접적인 HTTP 요청/응답 처리를 세밀하게 제어할 수 있다는 장점은 있다
- HTML과 비즈니스 로직이 하나의 서블릿 클래스 안에 혼재되어 코드가 복잡해지고 유지보수가 어려워진다
- 서블릿 방식은 코드의 재사용성을 어렵게 한다. 즉 동일한 비즈니스 로직을 여러 서블릿에서 반복해서 작성하게 되는 경우가 많다

```
public class HelloServlet extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
        User user = userRepository.getUser("leaven");  
        response.getWriter().println("<html><body>");  
        response.getWriter().println("<h1>Hello " + user.getUsername() + "</h1>");  
        response.getWriter().println("</body></html>");  
    }  
}
```

✓ 서블릿 기반 요청 프로세스



✓ Servlet 프로젝트 시작하기

1. 의존성을 추가한다

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
}
```

2. 서블릿 구현 - @WebServlet 어노테이션은 서블릿컨테이너가 서블릿 클래스를 등록, 생성하도록 메타 정보를 제공한다

```
@WebServlet(name = "helloServlet", urlPatterns = "/hello") // name:서블릿 이름, urlPatterns: URL 매팅  
public class HelloServlet extends HttpServlet {  
  
    @Override  
    protected void service(HttpServletRequest request, HttpServletResponse response) throws IOException {  
        response.setContentType("text/html");  
        response.getWriter().println("<h1>Hello, Servlet!</h1>");  
    }  
}
```

3. @ServletComponentScan 을 추가한다 - @WebServlet, @WebFilter, @WebListener와 같은 어노테이션으로 정의된 클래스들을 자동으로 스캔하고 등록한다

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

- 클라이언트는 urlPatterns에 있는 /hello 경로로 요청한다
`http://localhost:8080/hello`
- 서블릿 컨테이너는 요청을 받아 서블릿의 service() 메서드를 호출하고 요청을 처리한다
`Hello, Servlet!`

모델1 방식 – JSP

<https://github.com/onjsdnjs/servlet/tree/model-1>

✓ 개요

- JSP (Java Server Page)를 사용하여 웹 애플리케이션의 모든 로직을 한 파일에서 처리하는 방식이다
- 1999년 등장하여 초창기 Java 웹 애플리케이션 개발에서 많이 사용되었다

✓ JSP(Java Server Page) 등장

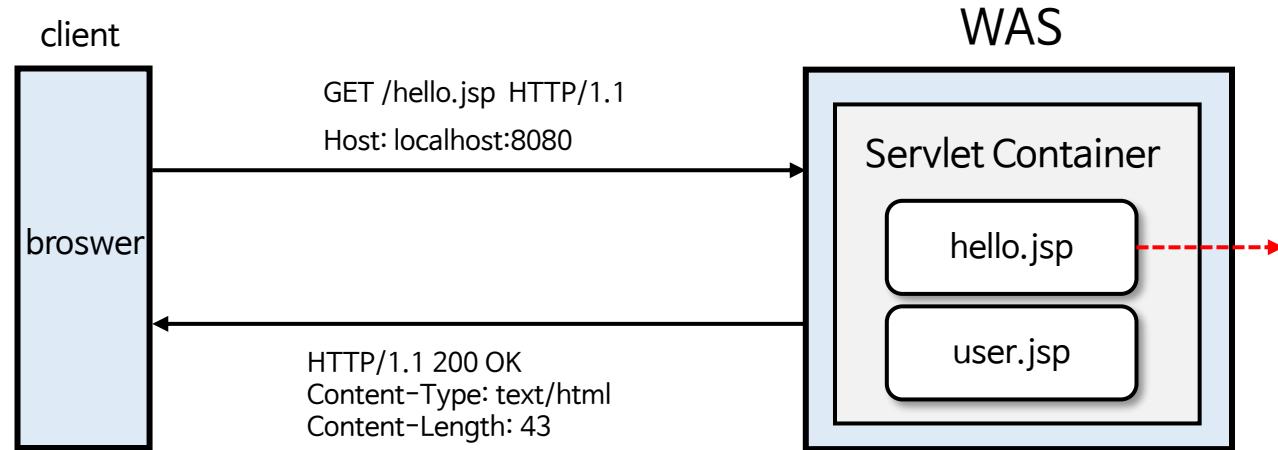
- HTML 내에 Java 코드를 삽입하여 동적 웹 페이지를 생성하는 기술로서 서블릿의 단점을 보완하기 위해 등장한 서블릿 기반의 스크립트 언어이다
- 모든 처리 로직이 하나의 JSP 파일에 존재한다

✓ 특징

- HTML 내에 Java 코드를 직접 삽입할 수 있어 화면 로직을 작성하는데 매우 편리하고 JSTL과 같은 태그 라이브러리를 사용하여 반복, 조건, 포맷 등 다양한 작업을 쉽게 처리할 수 있다
- 그러나 여전히 화면코드와 자바코드 모두 만들어야 하고 하나로 합치는 구조라서 서블릿에 비해 더 복잡하고 객체 지향적인 흐름을 방해한다

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%
String name = request.getParameter("name");
if (name == null) name = "World";
%>
<html>
<head><title>Model 1 Example</title> </head>
<body> <h1>Hello, <%= name %>!</h1> </body>
</html>
```

✓ 모델1 기반 요청 프로세스



WAS

Servlet Container

hello.jsp

user.jsp

```
%@ page import="java.util.*" %>  
<html>  
<head><title>Hello</title></head>  
<body>  
    <h1>Hello, JSP!</h1>  
    <%  
        Date date = new Date();  
        out.println("<h1>Current Date and Time: " + date.toString() + "</h1>");  
    %>  
</body>  
</html>
```

hello.jsp

```
public class hello_jsp extends HttpServlet {  
    public void _jspService(HttpServletRequest request, HttpServletResponse response) {  
        response.setContentType("text/html; charset=UTF-8");  
        JspWriter out = response.getWriter();  
        out.write("<html>");  
        out.write("<head>");  
        out.write("<title>JSP custom</title>");  
        out.write("</head>");  
        out.write("<body>");  
        out.write("<h1>Hello, JSP!</h1>");  
        out.write("Current date and time: " + new java.util.Date());  
        out.write("</body>");  
        out.write("</html>");  
    }  
}
```

화면 단에서 HTML 태그와 비즈니스 로직 처리를 모두 담당하고 있음!!

✓ 모델1 프로젝트 시작하기

1. 의존성을 추가한다

```
implementation 'org.springframework.boot:spring-boot-starter-web'  
implementation 'jakarta.servlet:jakarta.servlet-api:6.0.0'  
implementation 'jakarta.servlet.jsp:jakarta.servlet.jsp-api:3.1.0'  
implementation 'org.apache.tomcat.embed:tomcat-embed-jasper:10.1.8'
```

2. 디렉토리 구조 예시

```
src/  
└── main/  
    ├── java/  
    ├── resources/  
    └── webapp/  
        ├── jsp/  
        │   ├── index.jsp  
        │   ├── home.jsp  
        │   └── about.jsp
```

- 1) 서블릿은 존재하지 않고 jsp 로만 구성된다
- 2) 요청 진입점은 항상 jsp 파일이 위치한 경로이다

3. JSP 구현

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>  
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  
<!DOCTYPE html>  
<html>  
<head><title>JSP Example</title></head>  
<body><h1>Hello, Servlet!</h1></body>  
</html>
```

- ① 클라이언트는 jsp 가 위치한 URL 경로로 바로 요청한다

```
http://localhost:8080/jsp/hello.jsp
```

- ② 서블릿 컨테이너는 요청을 받아 jsp 파일을 호출하고 요청을 처리한다

```
Hello, Servlet!
```

모델2 방식 – MVC

<https://github.com/onjsdnjs/servlet/tree/model-2>

✓ 개요

- MVC (Model-View-Controller) 패턴을 따르는 구조로, 서블릿과 JSP를 결합하여 웹 애플리케이션을 개발하는 방식이다
- 2000년대 초반부터 주류 아키텍처로 자리잡았다.

✓ MVC 패턴으로 진화

- Model (Java Bean): 비즈니스 로직과 데이터 관리
- View (Jsp): 사용자 인터페이스
- Controller (Servlet): 사용자 요청을 처리하고 모델과 뷰를 연결

✓ 특징

- 화면은 JSP 가 담당하고 비즈니스 로직은 Servlet 이 담당하는 구조로 진화 함으로서 코드의 분리가 명확하여 유지보수가 용이하고 확장성이 높아졌다
- 작은 프로젝트에 비해 구현이 복잡하며 서블릿이나 모델1에 비해 성능적으로 약간의 오버헤드가 발생할 수 있다
- MVC 구조에 대한 확실한 이해를 바탕으로 한 전문적인 설계가 필요하며 잘못된 설계는 오히려 유지보수를 더 어렵게 할 수 있다

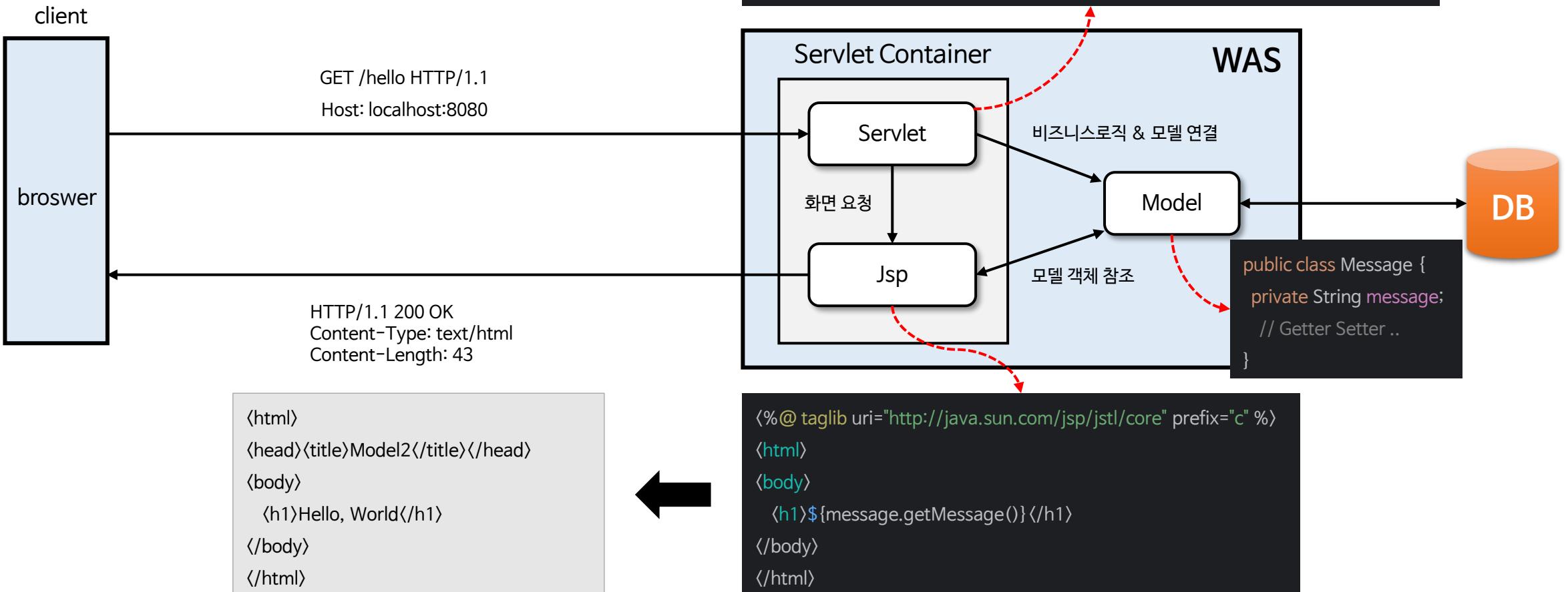
```
public class GreetController extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {  
        String name = request.getParameter("name");  
        if (name == null || name.isEmpty()) {  
            name = "World";  
        }  
        request.setAttribute("name", name);  
        request.getRequestDispatcher("/greet.jsp").forward(request, response);  
    }  
}
```



greet.jsp

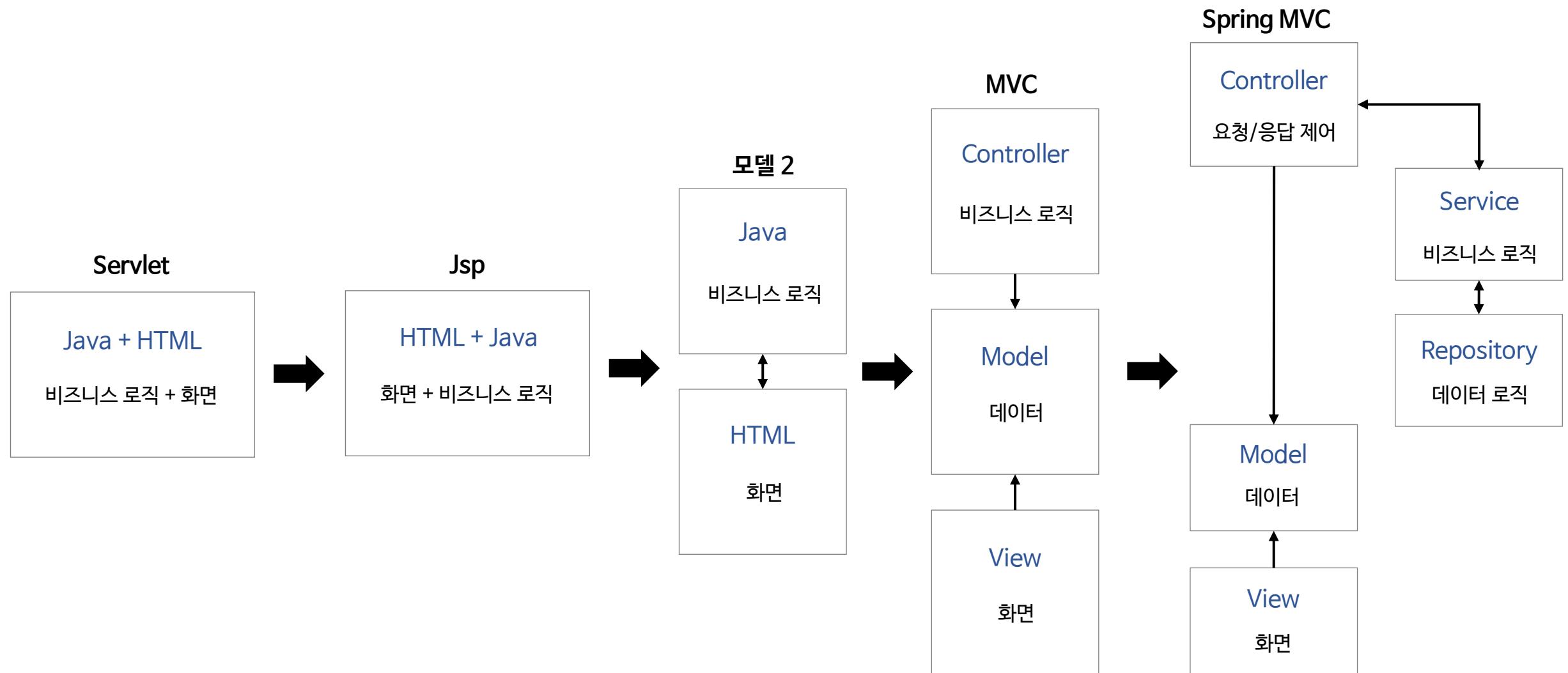
```
<%@ page contentType="text/html;charset=UTF-8"  
language="java" %>  
<html>  
<head>  
    <title>Model 2 Example</title>  
</head>  
<body>  
    <h1>Hello, ${name}!</h1>  
</body>  
</html>
```

✓ 모델2 기반 요청 프로세스



화면 단과 서비스 단의 역할을 분리하여 담당하고 있음!!

✓ Servlet vs Jsp vs MVC

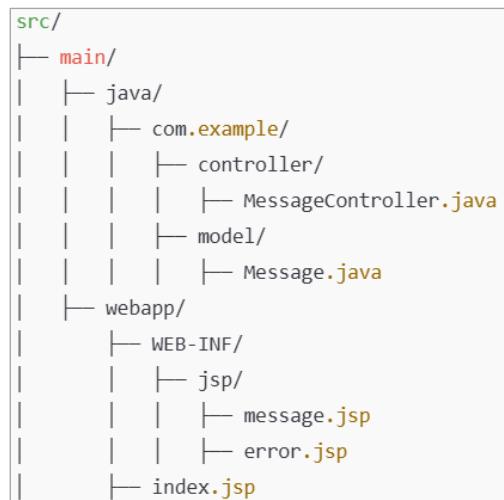


✓ 모델2 프로젝트 시작하기

1. 의존성을 추가한다

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'jakarta.servlet.jsp:jakarta.servlet.jsp-api:3.1.0'  
    implementation 'org.apache.tomcat.embed:tomcat-embed-jasper:10.1.8'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
}
```

2. 디렉토리 구조 예시



- 1) 모델 2 는 서블릿과 JSP 모두 사용한다
- 2) Jsp 는 WEB-INF 폴더 내에 위치하고 있어서 외부에서 바로 접근할 수 없는 경로이다
(Http://localhost:8080/jsp/message.jsp 처럼 외부에서 바로 접근 안됨)
- 3) 요청 진입점은 항상 서블릿이 되어야 하고 최종 응답은 서블릿 내부에서 Jsp 를 요청(포워딩)하는 식으로 이루어진다

3. 서블릿 구현

```
@WebServlet(name = "helloServlet", urlPatterns = "/hello") // name:서블릿 이름, urlPatterns: URL 매팅
public class HelloServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws IOException {
        request.setAttribute("message", "Welcome to MVC!");
        request.getRequestDispatcher("/WEB-INF/jsp/hello.jsp").forward(request, response);
    }
}
```

4. Jsp 구현

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html>
<html>
<head><title>JSP Example</title></head>
<body><h1>${message}</h1></body>
</html>
```

- ① 클라이언트는 urlPatterns 에 있는 /hello 경로로 요청한다

```
http://localhost:8080/hello
```

- ② 서블릿 컨테이너는 요청을 받아 처리한 후 jsp 파일을 호출하면 브라우저는 화면을 출력한다

```
Hello, Servlet!
```



스프링 웹 MVC 완전 정복

✓ 프론트 컨트롤러 패턴

1. 프론트 컨트롤러 패턴 이해
2. 스프링 웹 MVC 간단 예제

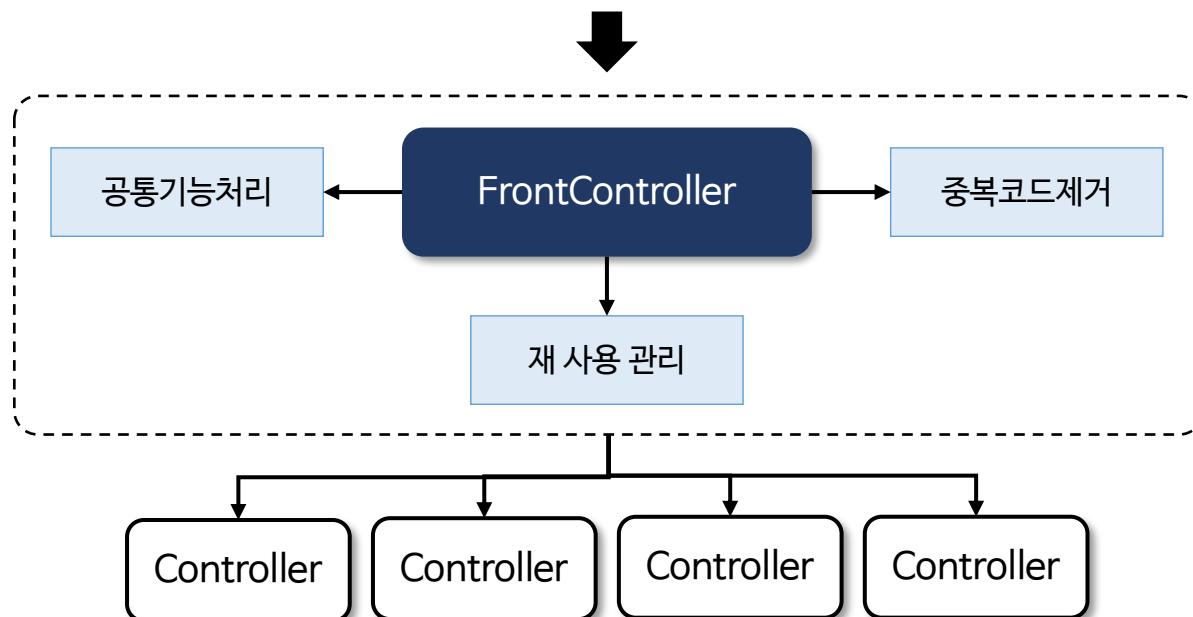
프론트 컨트롤러 패턴 이해

<https://github.com/onjsdnjs/servlet/tree/frontcontroller>

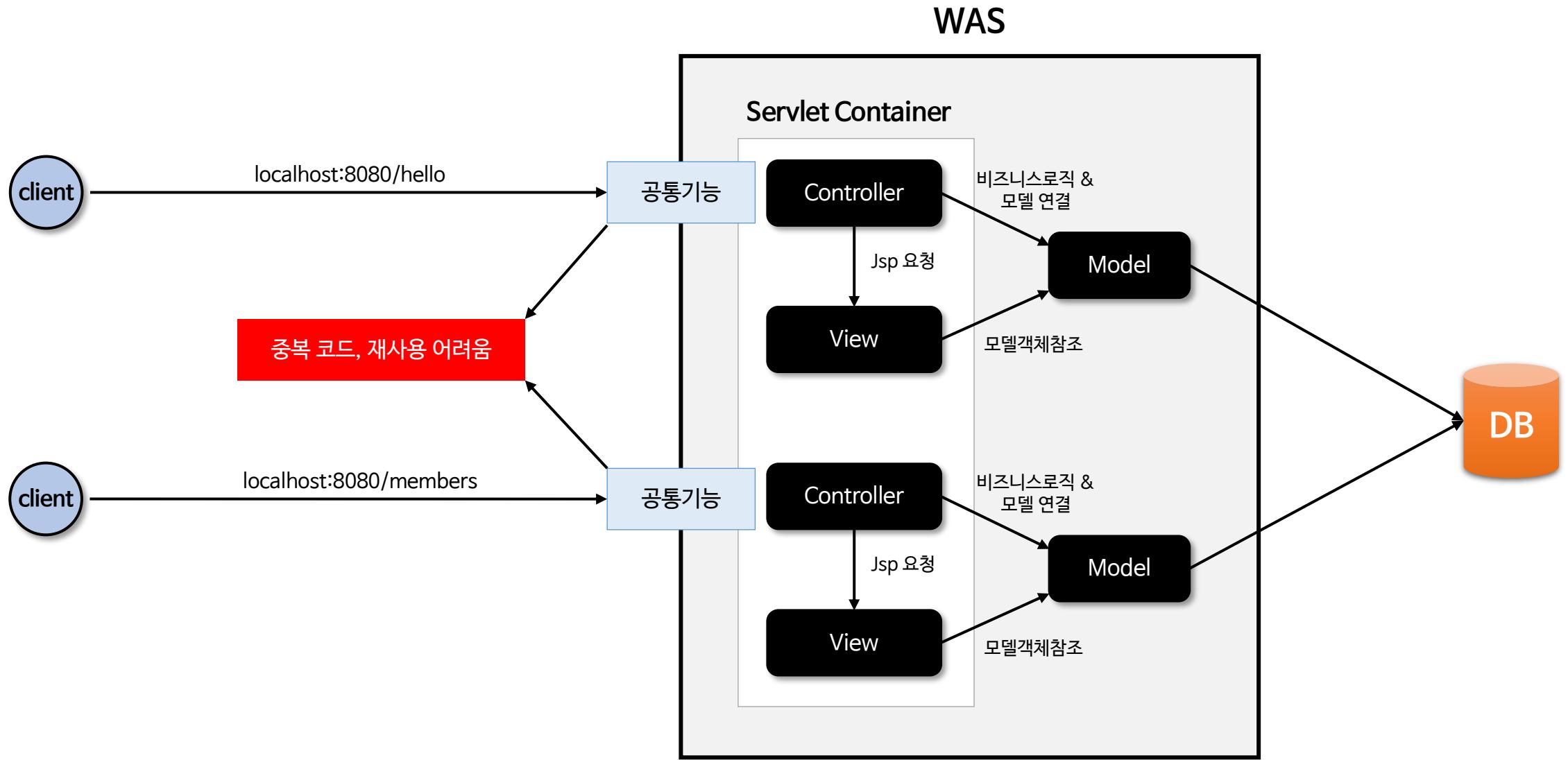
✓ 개요

- 웹 애플리케이션이 점점 복잡해지고 유지보수와 확장성을 필요로 하게 되면서 모델 2 방식을 개선한 프론트 컨트롤러 패턴이 등장하게 되었다
- 프론트 컨트롤러 패턴은 모델 2 방식의 단점을 보완하고 서블릿의 요청과 응답을 좀 더 구조적이고 체계적으로 관리하기 위해 탄생한 패턴이다

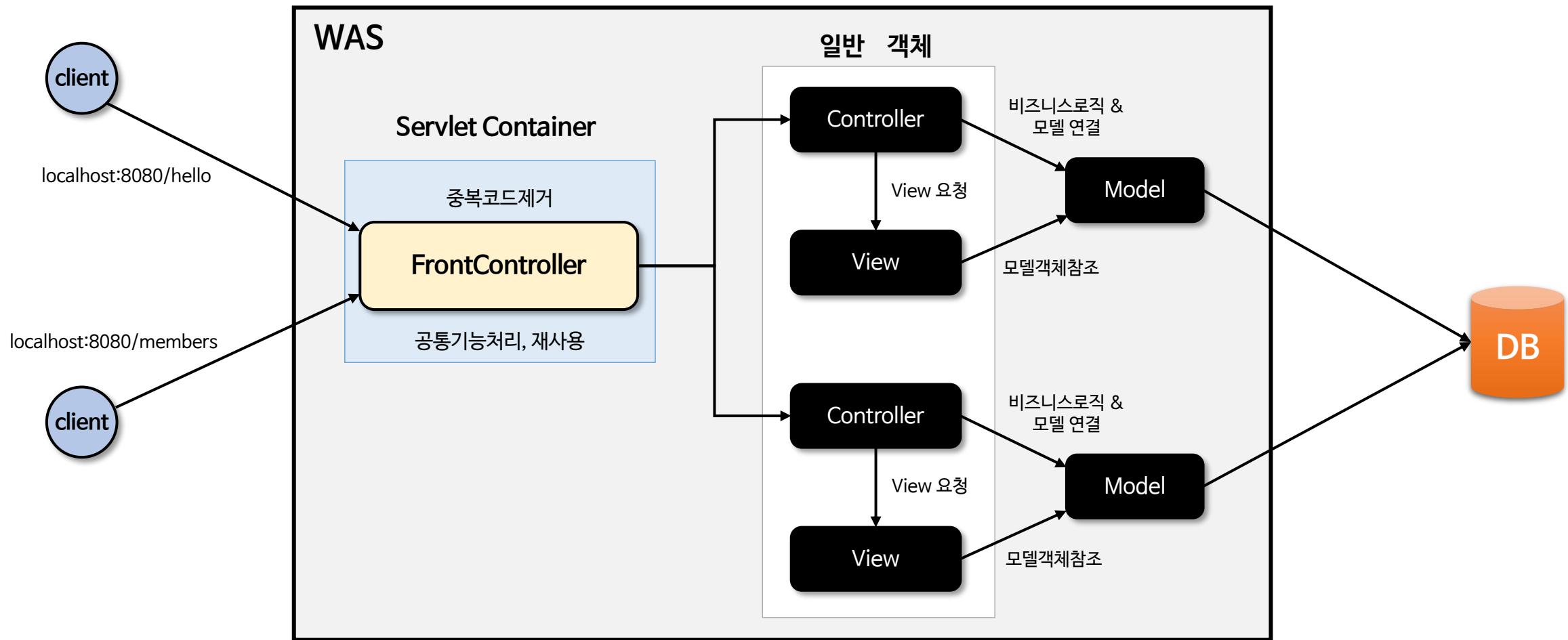
1. 서블릿 객체들이 요청마다 동일한 기능을 똑같이 처리해야 하는 문제..
2. 서블릿 객체들이 비슷한 기능의 유사 코드를 중복해서 실행하는 문제..
3. 서블릿 객체들이 재 사용해야 하는 객체들의 관리 문제…
4. 기타 유지 보수를 어렵게 만드는 요소들…



✓ 모델 2 구조



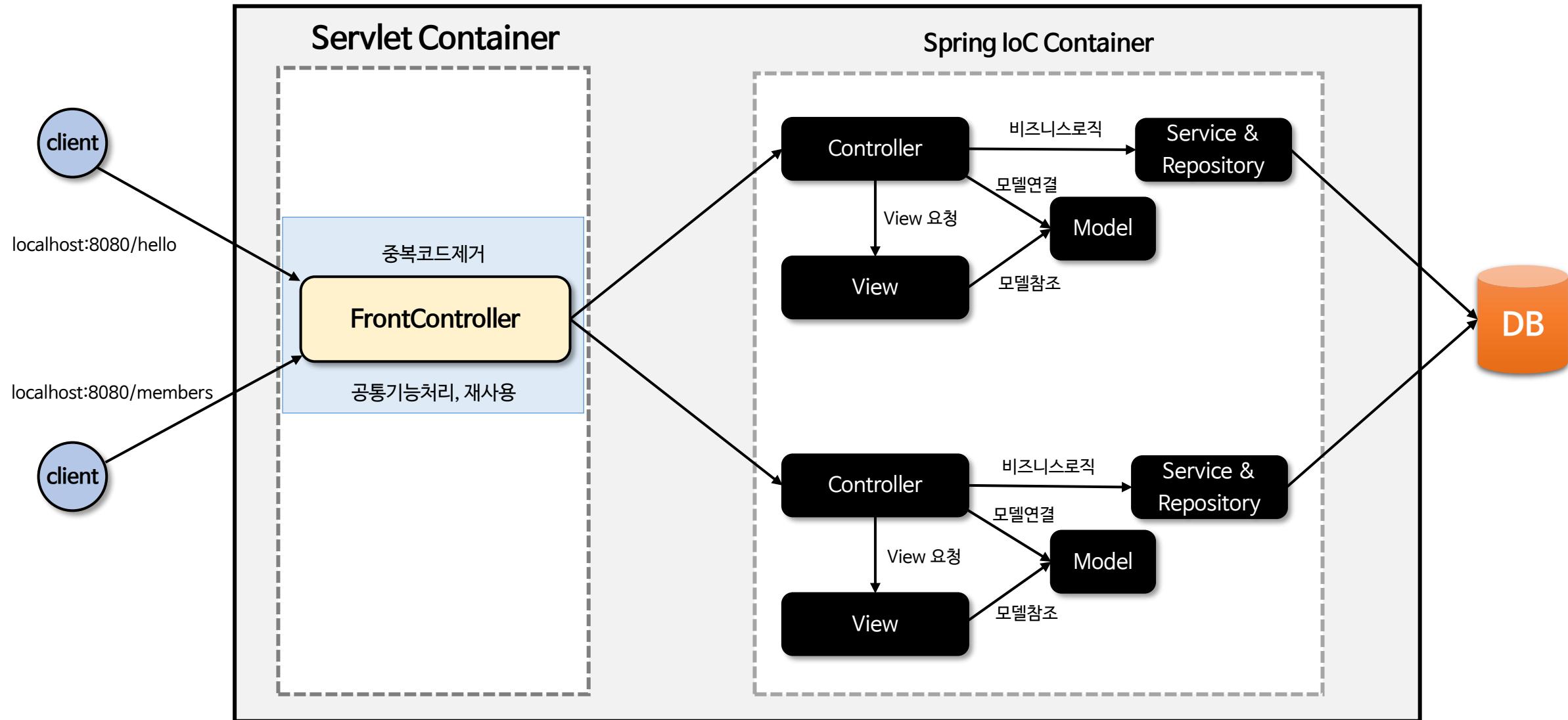
✓ 프론트 컨트롤러 패턴 구조



- 모든 클라이언트 요청을 무조건 단일 진입점을 거치도록 강제함으로서 요청 처리의 일관성을 유지할 수 있고 중앙에서 관리할 수 있게 된다
- 인증, 권한 부여, 로깅, 예외 처리 등의 공통 기능을 프론트 컨트롤러에서 일괄 처리해서 코드 중복을 줄이고, 유지보수를 용이하게 할 수 있다
- 공통 기능을 수정할 때 프론트 컨트롤러에서만 변경하면 되므로 수정이 용이하고 필요 시 각 서블릿을 쉽게 추가하고 확장 할 수 있다

✓ 스프링 웹 MVC

WAS



✓ 프론트 컨트롤러 패턴 프로젝트 시작하기

1. 의존성을 추가한다

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.apache.tomcat.embed:tomcat-embed-jasper'  
    implementation 'jakarta.servlet:jakarta.servlet-api'  
    implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api'  
    implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl'  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.springframework.boot:spring-boot-configuration-processor'  
    annotationProcessor 'org.projectlombok:lombok'  
}
```

- 롬복을 사용하기 위한 의존성을 추가한다

2. 디렉토리 구조 예시

```
src/  
└── main/  
    ├── java/  
    │   └── com.example/  
    │       ├── controller/  
    │       │   ├── FrontController.java      (프론트 컨트롤러)  
    │       │   ├── ProductFormController.java (상품 폼 출력)  
    │       │   ├── SaveProductController.java (상품 저장 처리)  
    │       │   ├── ProductListController.java (상품 리스트 출력)  
    │       └── model/  
    │           ├── Product.java          (상품 정보 모델 클래스)  
    │           └── ProductRepository.java (상품 저장소)  
    └── webapp/  
        ├── WEB-INF/  
        └── jsp/  
            ├── productForm.jsp          (상품 폼 뷰)  
            ├── productList.jsp         (상품 리스트 뷰)  
            └── success.jsp             (상품 저장 성공 뷰)
```

스프링 MVC 간단 예제

<https://github.com/onjsdnjs/servlet/tree/springmvc>

✓ 스프링 웹 MVC 간단 예제

- 스프링 웹 MVC를 사용해서 간단한 웹 어플리케이션을 구현해 보자
- 스프링 웹 MVC를 사용하면 기존의 방식에 비해 어떤 부분이 차이나는지 빠르게 살펴보자
- 프론트 컨트롤러 패턴 방식에 의해 구현된 예제와 비교해 보면서 스프링 웹 MVC의 장점을 이해한다

✓ 구성

1. 디렉토리 구조

```
src/
└── main/
    ├── java/
    │   └── com.example/
    │       ├── controller/
    │       │   └── ProductController.java      (상품 관련 요청 처리)
    │       ├── model/
    │       │   └── Product.java                (상품 모델 클래스)
    │       ├── service/
    │       │   └── ProductService.java        (상품 비즈니스 로직)
    │       └── repository/
    │           └── ProductRepository.java    (상품 저장소)
    └── webapp/
        ├── WEB-INF/
        └── jsp/
            └── productForm.jsp              (상품 폼 뷰)
            └── productList.jsp             (상품 리스트 뷰)
            └── success.jsp                 (상품 저장 성공 뷰)
```

2. application.properties

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



스프링 웹 MVC 완전 정복

✓ 서블릿 이해

1. 서블릿(Servlet)
2. HttpServletRequest
3. HttpServletResponse
4. 서블릿 컨테이너

서블릿(Servlet)

<https://github.com/onjsdnjs/servlet/tree/servletlifecycle>

✓ 개요

- 서블릿은 Jakarta EE (Enterprise Edition) 플랫폼의 핵심 기술 중 하나로 클라이언트-서버 모델에서 서버 측에서 실행되는 작은 자바 프로그램이다
- 주로 HTTP 요청과 응답을 처리하기 위해 사용되며 자바 서블릿 API를 통해 웹 애플리케이션 개발을 쉽게 할 수 있도록 해 준다



서블릿을 구현하기 위해서는 일반 서블릿 또는 HTTP 서블릿을 상속받아 구현할 수 있다

✓ 서블릿 생명 주기(Servlet Lifecycle)

- 서블릿은 서블릿 컨테이너에 의해 클래스 로드 및 객체 생성이 이루어지며 서블릿의 생명주기는 init, service, destroy 과정을 거친다

✓ 서블릿 로드 및 생성

1) 서블릿 로드 – 서블릿 컨테이너는 서블릿을 처음으로 요청받거나 혹은 애플리케이션이 시작될 때 서블릿을 메모리에 로드한다. 즉 두가지 방식이 있다

- 지연 로딩 (Lazy Loading) – 첫 번째 클라이언트 요청이 들어올 때 서블릿이 로드되고 서블릿 객체를 생성한다
- 즉시 로딩 (Eager Loading) – 애플리케이션이 시작될 때 서블릿이 로드되고 객체가 생성된다
- 이를 위해 <load-on-startup> 속성을 사용한다

```
<servlet>
  <servlet-name>customServlet</servlet-name>
  <servlet-class>com.custom.CustomServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

⟨ load-on-startup ⟩

- 음수 값 또는 설정하지 않은 경우: 서블릿은 지연 로딩된다
- 양수 값은 즉시 서블릿이 로딩되면 값이 낮을수록 더 높은 우선 순위를 가지며 1이 가장 먼저 로된다
- 0 값은 애플리케이션 시작 시 서블릿을 로드하지만 양수 값들보다 우선하지 않을 수 있다

✓ 서블릿 생명 주기

1) init() - 초기화 작업

```
public abstract class HttpServlet extends GenericServlet {  
    public void init(ServletConfig config) throws ServletException {  
        super.init(config);  
        // 초기화 작업 수행  
    }  
}
```

- 서블릿이 생성되고 init 메서드를 통해 초기화 되며 최초 한 번만 호출된다
- 주로 초기화 파라미터를 읽거나 DB 연결 설정 등 초기 설정 작업을 수행한다

2) service() - 요청 처리 작업

```
public abstract class HttpServlet extends GenericServlet {  
    public void service(HttpServletRequest req, HttpServletResponse resp)  
        throws ServletException, IOException {  
        doGet();  
        doPost();  
        ...  
    }  
}
```

- 클라이언트로부터의 모든 요청은 service 메서드를 통해 처리되며 HttpServletRequest 와 HttpServletResponse 객체가 생성되어 전달된다
- HTTP 메서드(GET, POST 등)에 따라 doGet, doPost 등의 메서드를 호출한다

3) destroy() - 종료 작업

```
public abstract class HttpServlet extends GenericServlet {  
    public void destroy(){  
        // 리소스 해제 작업 수행  
    }  
}
```

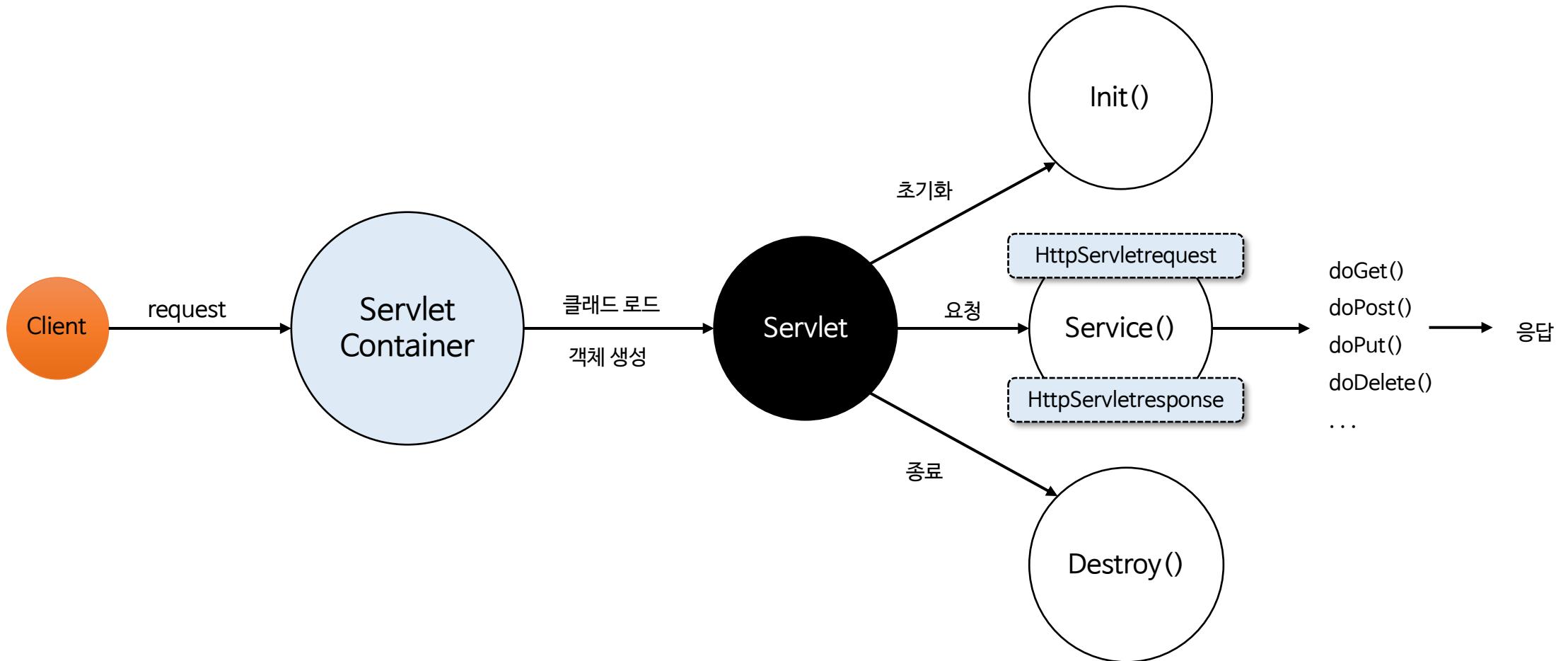
- 서블릿이 서비스에서 제외되면 destroy() 메서드를 통해 종료되고 이후 가비지 컬렉션 및 finalize 과정을 거친다

✓ 기본 구현

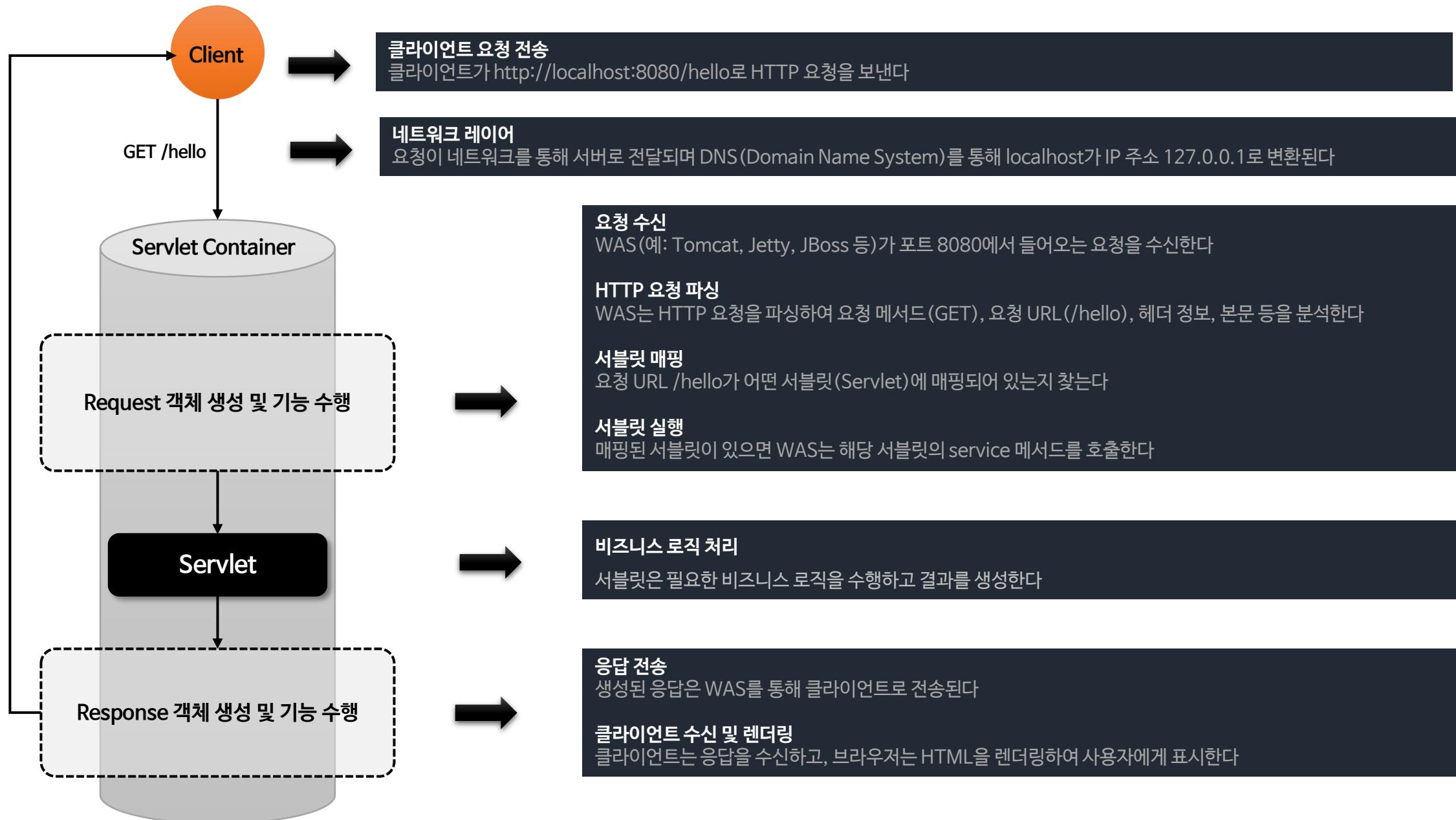
```
@WebServlet(name = "customServlet", urlPatterns = {"/hello"}, loadOnStartup = 1)
public class CustomServlet extends HttpServlet {
    public customServlet() {
        super(); // 객체 생성      싱글톤 객체 생성
    }
    @Override
    public void init() throws ServletException { 1
        super.init(); // 초기화 작업 수행    최초 한번 호출
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().println("Hello from customServlet!"); // GET 요청 처리
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.getWriter().println("Hello from customServlet!"); // POST 요청 처리
    }
    @Override
    public void destroy() { 3
        // 정리 작업 수행      종료 시 한번 호출
    }
}
```

1. init() : 최초 한번 호출
2. doGet() 및 doPost() : 요청마다 호출
3. destroy() : 종료 시 한번 호출

✓ 간단 흐름도



✓ 서블릿 요청 및 응답 흐름



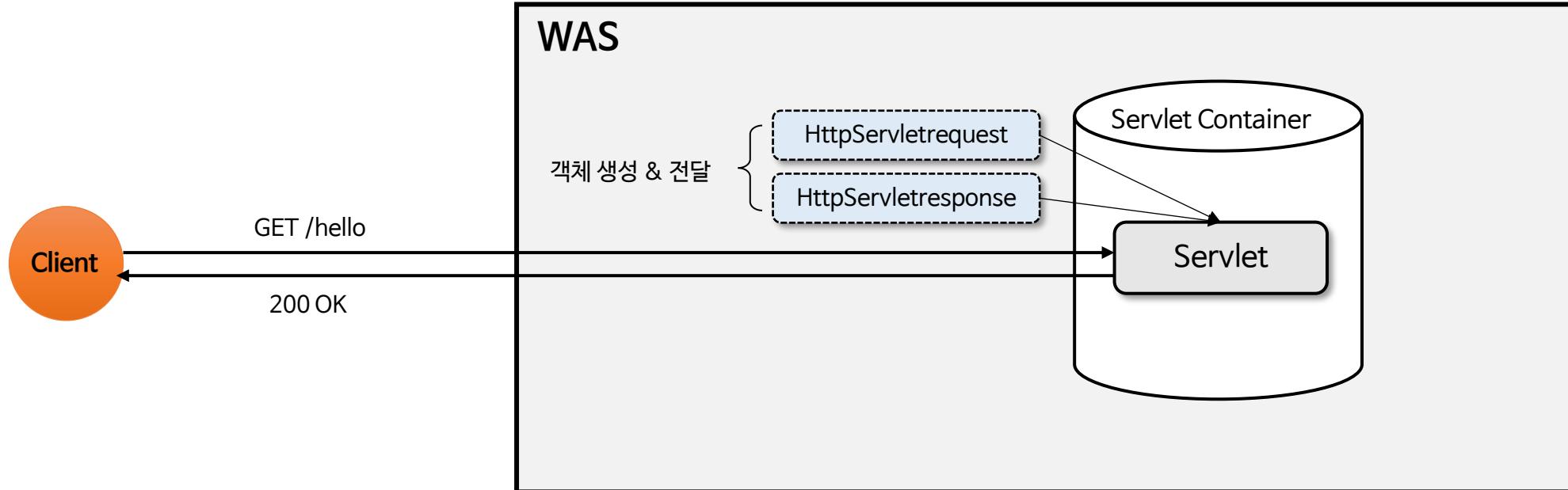
HttpServletRequest

<https://github.com/onjsdnjs/servlet/tree/HttpServletRequest>

✓ 개요

- HttpServletRequest 는 클라이언트로부터 Http 요청이 들어오면 요청 데이터를 분석하고 분석한 정보들이 저장되어 HttpServletResponse 와 함께 서블릿으로 전달되는 객체이다

```
public abstract class HttpServlet {  
    public void service( ServletRequest req, ServletResponse resp ) throws ServletException, IOException {}
```



✓ HttpServletRequest 구조

POST /order HTTP/1.1

Host: localhost:8080

Content-Type: application/x-www-form-urlencoded

orderNum=32&user=springmvc

HTTP 요청 데이터



이 외에도 쿠키 정보 접근, 세션 관리, 인증 정보 제공, 멀티파트 처리 등 많은 API를 제공하고 있다

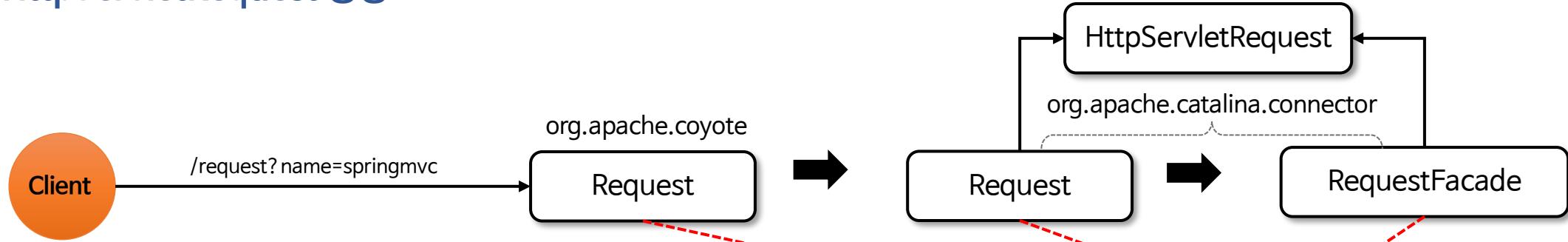
HTTP에 특화

HttpServletRequest	
getAuthType()	String
authenticate(HttpServletRequest)	boolean
isUserInRole(String)	boolean
getPathTranslated()	String
getHeaderNames()	Enumeration<String>
login(String, String)	void
getHeader(String)	String
getMethod()	String
getUserPrincipal()	Principal
getSession(boolean)	HttpSession
getParts()	Collection<Part>
getTrailerFields()	Map<String, String>
getDateHeader(String)	long
upgrade(Class<T>)	T
getHeaders(String)	Enumeration<String>
getPart(String)	Part
getServletPath()	String
getRequestURL()	StringBuffer
getRemoteUser()	String
getRequestedSessionId()	String
logout()	void
isTrailerFieldsReady()	boolean
isRequestedSessionIdFromCookie()	boolean
getQueryString()	String
changeSessionId()	String
getCookies()	Cookie[]
newPushBuilder()	PushBuilder
getIntHeader(String)	int
getPathInfo()	String
getContextPath()	String
isRequestedSessionIdFromURL()	boolean
getHttpServletMapping()	HttpServletMapping
getRequestURI()	String
isRequestedSessionIdValid()	boolean
getSession()	HttpSession

웹 공통

ServletRequest	
getServletContext()	ServletContext
removeAttribute(String)	void
getLocales()	Enumeration<Locale>
getParameter(String)	String
getParameterNames()	Enumeration<String>
getLocale()	Locale
getReader()	BufferedReader
getRequestDispatcher(String)	RequestDispatcher
isAsyncStarted()	boolean
getContentType()	String
getRemoteHost()	String
setCharacterEncoding(String)	void
getDispatcherType()	DispatcherType
getCharacterEncoding()	String
getParameterMap()	Map<String, String[]>
getInputStream()	ServletInputStream
startAsync()	AsyncContext
getAttribute(String)	Object
setAttribute(String, Object)	void
isAsyncSupported()	boolean
getAttributeNames()	Enumeration<String>
getScheme()	String
getAsyncContext()	AsyncContext
getLocalPort()	int
getProtocol()	String
getRequestId()	String
getLocalAddr()	String
getRemoteAddr()	String
getContentLengthLong()	long
getProtocolRequestId()	String
isSecure()	boolean
getRemotePort()	int
getLocalName()	String
getServletConnection()	ServletConnection
startAsync(ServletRequest, ServletResponse)	AsyncContext
getParameterValues(String)	String[]
getServerName()	String
getContentLength()	int

✓ HttpServletRequest 생성



HTTP 요청이 시작되면 총 3 개의 Request 객체가 생성된다

① org.apache.coyote.Request 객체 생성

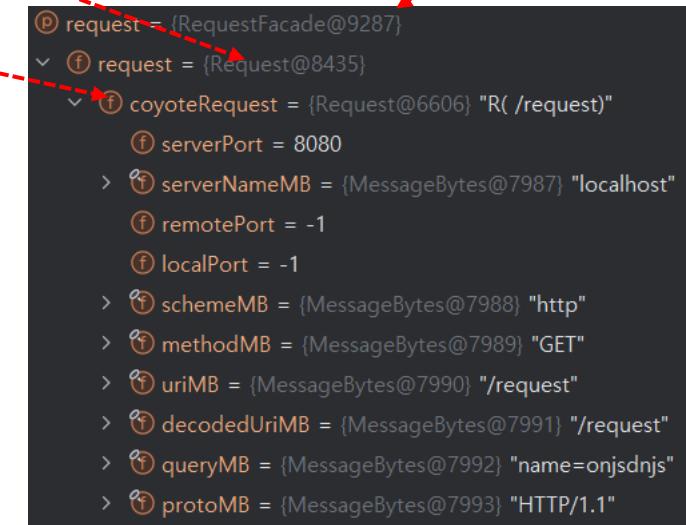
- 낮은 수준의 HTTP 요청 처리를 담당하여 서블릿 컨테이너와 독립적으로 동작

② org.apache.catalina.connector.Request 객체 생성

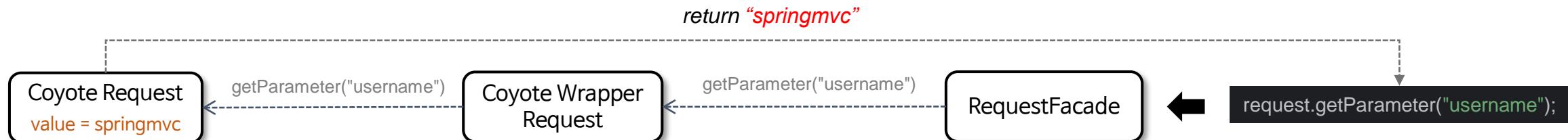
- 서블릿 API 규격을 구현하여 고수준 요청 데이터를 처리

③ org.apache.catalina.connector.RequestFacade 객체 생성

- 캡슐화를 통해 서블릿 API 사용을 표준화하고 내부 구현을 보호



- HTTP 요청 데이터를 분석한 값을 org.apache.coyote.Request 객체에 저장하고 이를 서블릿 컨테이너에서 동작할 수 있도록 HttpServletRequest 구현체들을 생성하고 연결한다



✓ 스프링의 Request

1) WebRequest

HTTP 요청의 파라미터, 헤더 등 메타데이터에 대한 추상화된 접근을 제공하는 인터페이스

2) NativeWebRequest

네이티브 서블릿 객체(HttpServletRequest)에 접근할 수 있는 인터페이스

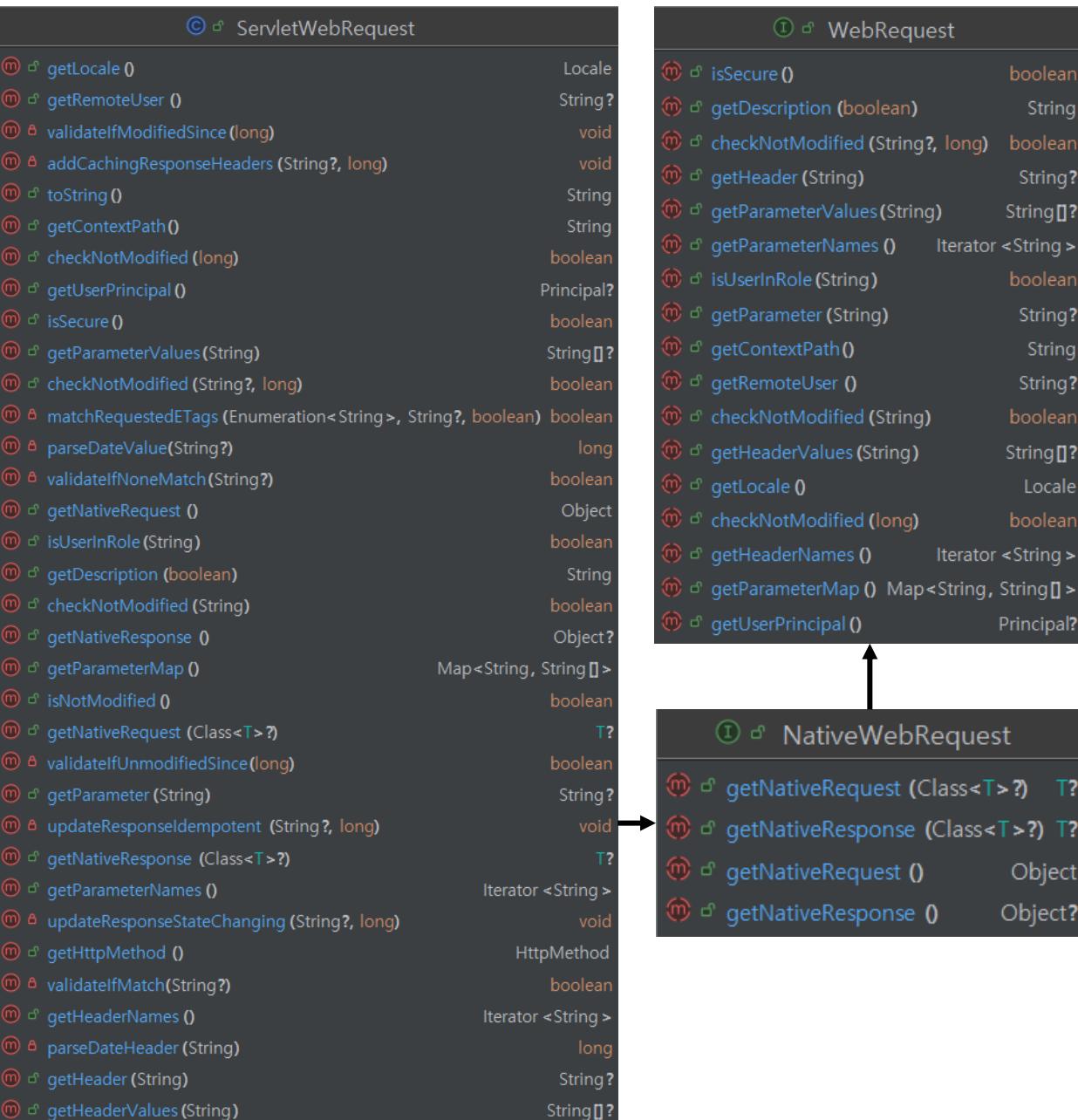
3) ServletWebRequest

HttpServletRequest 와 HttpServletResponse 를 감싸며 Spring 자체 추상화된 요청/응답 기능을 제공하는 구현체

```
ServletWebRequest webRequest = new ServletWebRequest(httpRequest, httpResponse);
String headerValue = webRequest.getHeader("User-Agent");
String paramValue = webRequest.getParameter("name");

// 네이티브 요청 객체 활용 가능
HttpServletRequest request =
servletWebRequest.getNativeRequest(HttpServletRequest.class);
System.out.println("Native Request URI: " + request.getRequestURI());

// 네이티브 응답 객체 활용 가능
servletWebRequest.getResponse().setStatus(HttpStatus.SC_OK);
servletWebRequest.getResponse().getWriter().write("successful");
```



HttpServletRequest 기본

<https://github.com/onjsdnjs/servlet/tree/httpRequestServletApi>

✓ 개요

- HttpServletRequest는 클라이언트 요청에 대한 서블릿 API의 핵심 인터페이스로서 개발 시 자주 활용되거나 기본적으로 숙지해야 할 API 들이 있다

Header 관련 API

- ① getHeader(String name)
- ② getHeaders(String name)
- ③ getHeaderNames()

요청 메서드 및 경로 관련

- ① getMethod()
- ② getRequestURI()
- ③ getRequestURL()
- ④ getContextPath()
- ⑤ getServletPath()
- ⑥ getPathInfo()

세션 관련

- ① getSession(boolean create)
- ② getSession()
- ③ getRequestedSessionId()

속성 관련

- ① getAttribute(String name)
- ② getAttributeNames()
- ③ setAttribute(String name, Object value)
- ④ removeAttribute(String name)

원격 및 로컬 정보 관련

- ① getRemoteAddr()
- ② getRemoteHost()
- ③ getRemotePort()
- ④ getLocalAddr()
- ⑤ getLocalName()
- ⑥ getLocalPort()

기타 메타데이터 관련

- ① getContentType()
- ② getContentLength()
- ③ getProtocol()
- ④ getScheme()
- ⑤ getServerName()
- ⑥ getServerPort()

✓ Header 관련 API

```
public class HelloServlet extends HttpServlet {  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
    {  
        System.out.println("==== Header Information ====");  
  
        // 단일 Header  
        System.out.println("User-Agent: " + request.getHeader("User-Agent"));  
  
        // 복수 Headers  
        System.out.println("Accept-Language Headers:");  
        request.getHeaders("Accept-Language").asIterator()  
            .forEachRemaining(header -> System.out.println(" - " + header));  
  
        // All Headers  
        System.out.println("All Headers:");  
        request.getHeaderNames().asIterator()  
            .forEachRemaining(headerName -> System.out.println(headerName + ": " + request.getHeader(headerName)));  
    }  
}
```

요청

http://localhost:8080/example/path

결과

==== Header Information ====
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...
Chrome/91.0.4472.124 Safari/537.36

Accept-Language Headers:
- en-US,en;q=0.9
- ko-KR,ko;q=0.8

All Headers:
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...
Chrome/91.0.4472.124 Safari/537.36
Accept-Language: en-US,en;q=0.9,ko-KR,ko;q=0.8

✓ 요청 메서드 및 경로 관련 API

```
public void handleRequestInfo(HttpServletRequest request) {  
    System.out.println("==== Request Information ====");  
  
    System.out.println("HTTP Method: " + request.getMethod());  
    System.out.println("Request URI: " + request.getRequestURI());  
    System.out.println("Request URL: " + request.getRequestURL());  
}
```

요청

http://localhost:8080/example/path

결과

==== Request Information ====
HTTP Method: GET
Request URI: /example/path
Request URL: http://localhost:8080/example/path

✓ 세션 관련 API

```
public void handleSession(HttpServletRequest request) {  
    System.out.println("==== Session Information ====");  
  
    HttpSession session = request.getSession(false) or true;  
    System.out.println("Session: " + (session != null ? "Exists" : "None"));  
    System.out.println("Requested Session ID: " + request.getRequestedSessionId());  
}
```

결과

==== Session Information ====
Session: Exists
Requested Session ID: ABC1234567890

✓ 속성 관련 API

```
public void handleAttributes(HttpServletRequest request) {  
    request.setAttribute("attribute1", "Value1");           // HTTP 요청 범위 안에서 데이터가 유지되고 요청이 끝나면 데이터는 사라진다  
    request.setAttribute("attribute2", "Value2");  
  
    System.out.println("속성 설정:");  
    System.out.println("attribute1: Value1");  
    System.out.println("attribute2: Value2");  
  
    // 모든 속성 출력  
    System.out.println("\n모든 속성 출력:");  
    request.getAttributeNames().asIterator()  
        .forEachRemaining(attributeName -> System.out.println(attributeName + ": " + request.getAttribute(attributeName)));  
  
    // 속성 삭제  
    request.removeAttribute("attribute1");  
    System.out.println("\n삭제 속성: 'attribute1'");  
  
    // 삭제 후 모든 속성 출력  
    System.out.println("\n삭제 후 속성:");  
    request.getAttributeNames().asIterator()  
        .forEachRemaining(attributeName -> System.out.println(attributeName + ": " + request.getAttribute(attributeName)));
```

결과

속성 설정
attribute1: Value1
attribute2: Value2

모든 속성 출력
attribute1: Value1
attribute2: Value2

속성 삭제
'attribute1'

삭제 후 속성
attribute2: Value2

✓ 원격 및 로컬 정보 관련 API

```
public void handleRequestInfo(HttpServletRequest request) {  
    System.out.println("Remote Addr: " + request.getRemoteAddr());  
    System.out.println("Remote Host: " + request.getRemoteHost());  
    System.out.println("Remote Port: " + request.getRemotePort());  
    System.out.println("Local Addr: " + request.getLocalAddr());  
    System.out.println("Local Name: " + request.getLocalName());  
    System.out.println("Local Port: " + request.getLocalPort());  
}
```

요청

http://localhost:8080/example/path

결과

```
Remote Addr: 127.0.0.1  
Remote Host: 127.0.0.1  
Remote Port: 60962  
Local Addr: 127.0.0.1  
Local Name: display.ad.daum.net  
Local Port: 8080
```

-Djava.net.preferIPv4Stack=true // VM Option에 추가

✓ 기타 메타데이터 API

```
public void handleSession(HttpServletRequest request) {  
    System.out.println("Content Type: " + request.getContentType());  
    System.out.println("Content Type: " + request.getContentLength());  
    System.out.println("Content Type: " + request.getCharacterEncoding());  
    System.out.println("Protocol: " + request.getProtocol());  
    System.out.println("Scheme: " + request.getScheme());  
    System.out.println("Server Name: " + request.getServerName());  
    System.out.println("Server Port: " + request.getServerPort());  
}
```

요청

http://localhost:8080/request?name=onjsdnjs

결과

```
Content Type: null  
Content Type: -1  
Content Type: UTF-8  
Protocol: HTTP/1.1  
Scheme: http  
Server Name: localhost  
Server Port: 8080
```

HttpServletRequest – 요청 처리

<https://github.com/onjsdnjs/servlet/tree/httpRequestServletProsess>

✓ 개요

- HttpServletRequest는 클라이언트의 다양한 데이터 포맷과 요청 유형에 따라 정보를 읽고 처리할 수 있는 API를 제공한다
- 요청을 처리하는 방식으로 URL 쿼리 파라미터, 폼 데이터(GET/POST), REST API 처리 등 세 가지로 나누어 구분 할 수 있다

✓ URL 쿼리 파라미터

- URL 쿼리 파라미터는 HTTP 요청의 쿼리 문자열(Query String)에 포함되어 전달되며 이는 보통 GET 요청에서 사용된다
- 주요 API

```
request.getParameter(String name):           // 단일 파라미터 값을 반환한다  
Request.getParameterValues(String name):    // 다중 파라미터 값을 배열로 반환한다  
Request.getParameterMap():                 // 모든 파라미터를 Map<String, String[]> 형식으로 반환한다
```

1. 단일 파라미터: 키-값 쌍으로 구성되며 하나의 키에 단일 값이 매핑된다
 - ?name=leaven
 2. 키가 동일한 복수 파라미터: 하나의 키에 여러 값이 매핑된다
 - ?hobby=reading&hobby=writing
- 주의 사항
 - URL에 민감한 정보를 포함하면 보안 위험이 있기 때문에 사용하면 위험하며 (예: 비밀번호, 개인정보) HTTPS를 사용해 암호화를 보장해야 한다.
 - 브라우저마다 URL 길이에 제한이 있으므로 대용량 데이터를 쿼리 파라미터로 전달하기 어렵다

✓ URL 쿼리 파라미터 구현 예제

```
@WebServlet(name="/urlParameterServlet", urlPatterns="/handleGetRequest")
public class UrlParameterServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/plain");
        response.getWriter().println("== URL 요청 파라미터 처리 ==");

        String name = request.getParameter("name");
        response.getWriter().println("단수 파라미터 'name': " + name);

        String[] hobbies = request.getParameterValues("hobby");
        response.getWriter().println("복수 파라미터 'hobby':");
        if (hobbies != null) {
            for (String hobby : hobbies) {
                response.getWriter().print(" - " + hobby);
            }
        }

        Map<String, String[]> parameterMap = request.getParameterMap();
        response.getWriter().println("모든 요청 파라미터:");
        parameterMap.forEach((key, value) -> {
            response.getWriter().println(key + ": " + String.join(", ", value));
        });
    }
}
```

요청

```
http://localhost:8080/handleGetRequest?name=leaven&hobby=reading&hobby=writing
```

결과

```
== URL 요청 파라미터 처리 ==
```

```
단수 파라미터 name: leaven
```

```
복수 파라미터 hobby: - reading - writing
```

```
모든 요청 파라미터:
```

```
name: leaven
```

```
hobby: reading, writing
```

✓ FORM 데이터 처리

- FORM 데이터는 HTML <form> 태그를 통해 클라이언트에서 서버로 전달되며 데이터는 요청 메서드에 따라 다르게 처리된다
- GET 방식
 - 요청 데이터가 URL의 쿼리 문자열에 포함된다
 - 전송 데이터의 크기는 URL 길이 제한에 의해 제약을 받지 않고 URL이 노출되므로 민감한 데이터 전송에 적합하지 않다
- POST 방식
 - 요청 데이터가 HTTP 요청 본문에 포함된다
 - Content-Type
 - application/x-www-form-urlencoded – 기본 폼 데이터 전송 방식으로서 키-값 쌍이 URL 인코딩된 형태로 전달된다
 - Body: name=leaven&hobby=reading&hobby>writing
- 주요 API
 - URL 쿼리 파라미터 요청 처리와 마찬가지로 동일한 API를 사용한다

```
request.getParameter(String name):           // 단일 파라미터 값을 반환한다  
Request.getParameterValues(String name):    // 다중 값 파라미터를 배열로 반환한다  
Request.getParameterMap():                 // 모든 파라미터를 Map<String, String[]> 형식으로 반환한다
```

✓ REST API 데이터 처리

- REST API 요청은 클라이언트가 JSON 또는 Plain Text 형태의 데이터를 HTTP 요청 본문에 포함하여 서버로 전송하는 방식으로서 getParameter() 와 같은 방법이 아닌 InputStream 으로부터 본문 데이터를 직접 읽어야 한다
- **JSON 데이터**
 - Content-Type: application/json
 - 요청 본문에 JSON 형식으로 데이터를 포함한다
 - JSON 데이터를 처리하려면 요청 본문을 파싱해야 한다
- **Plain Text 데이터**
 - Content-Type: text/plain
 - 요청 본문에 단순 문자열 데이터를 포함한다
- **주요 API**

`request.getReader()`: 요청 본문을 문자 스트림으로 읽는다

`request.getInputStream()`: 요청 본문을 바이너리 스트림으로 읽는다

✓ Plain Text 구현 예제

```
@WebServlet(name="/plainTextServlet", urlPatterns="/rest/plain-text")
public class PlainTextServlet extends HttpServlet {

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

        System.out.println("==> Plain Text Request ==>");
        StringBuilder stringBuilder = new StringBuilder();
        try (BufferedReader reader = request.getReader()) {
            String line;
            while ((line = reader.readLine()) != null) {
                stringBuilder.append(line);
            }
        }
        String requestBody = stringBuilder.toString();
        System.out.println("Received Plain Text: " + requestBody);

        response.setContentType("text/plain");
        response.getWriter().write("Processed Plain Text: " + requestBody);
    }
}
```

요청

```
http://localhost:8080/rest/plain-text
Content-Type: text/plain
Hello, World!
```

결과

```
HTTP/1.1 200 OK
Content-Type: text/plain

Processed Plain Text: Hello, World!
```

✓ JSON 데이터 구현 예제

```
@WebServlet(name="/jsonServlet", urlPatterns="/rest/json")
public class JsonServlet extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
        System.out.println("==== JSON Request ====");

        ObjectMapper objectMapper = new ObjectMapper();
        Map<String, Object> requestBody = objectMapper.readValue(request.getReader(), Map.class);

        System.out.println("Received JSON: " + requestBody);
        Map<String, Object> responseBody = Map.of(
            "status", "success",
            "received", requestBody
        );
        response.setContentType("application/json");
        response.getWriter().write(objectMapper.writeValueAsString(responseBody));
    }
}
```

요청

```
http://localhost:8080/rest/json
Content-Type: application/json
{
    "name": "leaven",
    "age": 30,
    "city": "New York"
}
```

결과

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "status": "success",
    "received": {
        "name": "leaven",
        "age": 30,
        "city": "New York"
    }
}
```

HttpServletResponse

<https://github.com/onjsdnjs/servlet/tree/httpResponseServletApi>

✓ 개요

- HttpServletResponse는 서버가 클라이언트의 요청에 대한 응답을 생성하고 반환할 때 사용되며 HTTP 응답의 상태 코드, 헤더, 본문 데이터를 설정하고 제어하는 다양한 메서드를 제공한다

✓ 구조

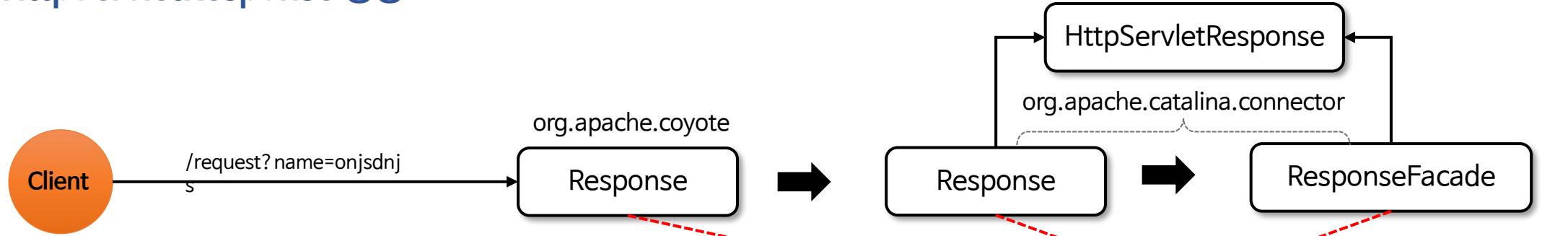
ServletResponse	
reset()	void
getCharacterEncoding()	String
getBufferSize()	int
getWriter()	PrintWriter
setBufferSize(int)	void
getContentType()	String
setLocale(Locale)	void
setContentLength(int)	void
resetBuffer()	void
setContentType(String)	void
getOutputStream()	ServletOutputStream
getLocale()	Locale
setCharacterEncoding(String)	void
setContentLengthLong(long)	void
isCommitted()	boolean
flushBuffer()	void

HttpServletResponse	
getHeader(String)	String
encodeURL(String)	String
setDateHeader(String, long)	void
addDateHeader(String, long)	void
setIntHeader(String, int)	void
sendError(int)	void
setTrailerFields(Supplier<Map<String, String>>)	void
addIntHeader(String, int)	void
addCookie(Cookie)	void
sendRedirect(String)	void
getTrailerFields()	Supplier<Map<String, String>>
getStatus()	int
setStatus(int)	void
encodeRedirectURL(String)	String
sendError(int, String)	void
setHeader(String, String)	void
containsHeader(String)	boolean
addHeader(String, String)	void
getHeaders(String)	Collection<String>
getHeaderNames()	Collection<String>

주요 API

메서드	설명
setStatus(int sc)	응답 상태 코드를 설정
sendError(int sc, String msg)	상태 코드와 메시지를 설정하여 에러 응답을 전송
setHeader(String name, String value)	응답 헤더 설정
addHeader(String name, String value)	응답 헤더 추가
setContentType(String type)	응답 본문의 콘텐츠 타입 설정
getWriter()	응답 본문 작성을 위한 PrintWriter 반환
sendRedirect(String location)	클라이언트를 다른 URL로 리다이렉트
addCookie(Cookie cookie)	쿠키를 추가

✓ HttpServletResponse 생성



HTTP 요청이 시작되면 총 3 개의 Response 객체가 생성된다

① org.apache.coyote.Response 객체 생성

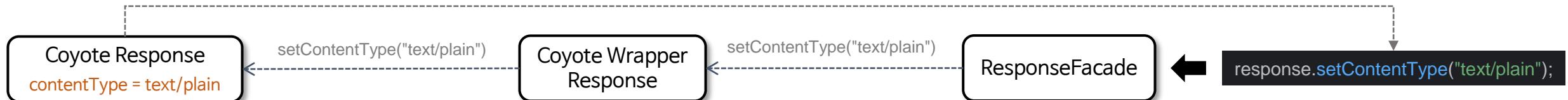
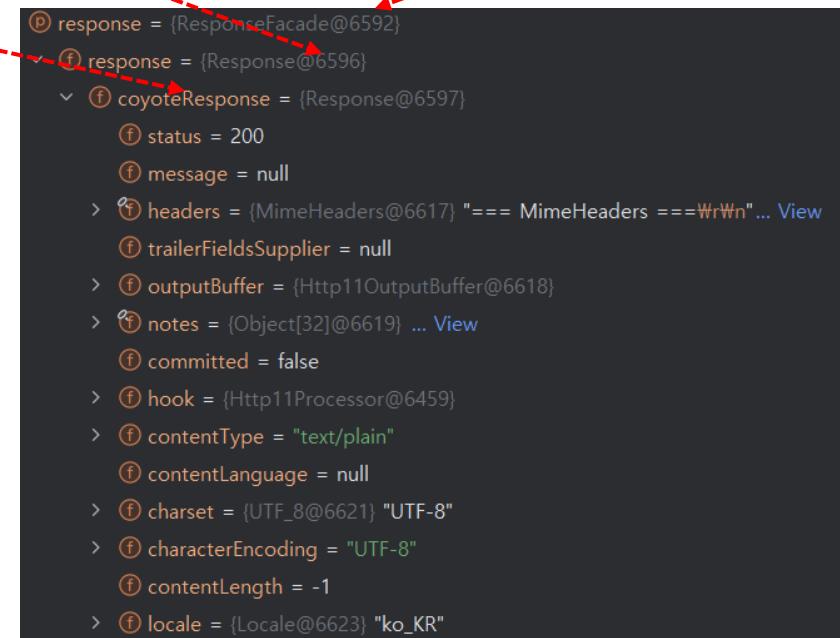
- 낮은 수준의 HTTP 응답 처리를 담당하여 서블릿 컨테이너와 독립적으로 동작

② org.apache.catalina.connector.Response 객체 생성

- 서블릿 API 규격을 구현하여 고수준 응답 데이터를 처리

③ org.apache.catalina.connector.ResponseFacade 객체 생성

- 캡슐화를 통해 서블릿 API 사용을 표준화하고 내부 구현을 보호



✓ setStatus(int sc)

```
@WebServlet(name = "setStatusServlet", urlPatterns = "/setStatusExample")
public class SetStatusServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.setStatus(HttpServletResponse.SC_ACCEPTED); // 202 상태 코드 설정
        response.getWriter().write("Response status set to 202 Accepted");
    }
}
```

요청

http://localhost:8080/statusExample

결과

HTTP/1.1 202 Accepted

Content-Type: text/plain; charset=UTF-8

Response Body: Response status set to 202 Accepted

✓ sendError(int sc, String msg)

```
@WebServlet(name = "sendErrorServlet", urlPatterns = "/sendErrorExample")
public class SendErrorServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.sendError(HttpServletResponse.SC_BAD_REQUEST, "Invalid request");
    }
}
```

요청

http://localhost:8080/sendErrorExample

결과

HTTP/1.1 400 Bad Request

Content-Type: text/html; charset=UTF-8

Response Body: Invalid request (브라우저 기본 에러 페이지)

- 1) sendError를 실행하면 기존 응답 버퍼의 내용을 모두 지우고 오류 응답만 클라이언트에 전달되도록 보장한다
- 2) sendError는 설계상 정상적인 처리 흐름이 아니라 오류 상황을 클라이언트에 알리기 위해 존재한다. 그렇기 때문에 특별한 예외가 발생하지 않아도 응답이 오류로 전송된다
- 3) sendError를 실행하면 응답이 이미 커밋되었음을 나타내는 플래그가 설정되고 이후로는 응답을 수정할 수 없다. Response.getWriter()로 추가 데이터를 출력하려고 시도하면 예외가 발생한다

✓ setHeader(String name, String value)

```
@WebServlet(name = "setHeaderServlet", urlPatterns = "/setHeaderExample")  
public class SetHeaderServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {  
        response.setHeader("X-Custom-Header", "CustomValue");  
        response.getWriter().write("Header : X-Custom-Header=CustomValue");  
    }  
}
```

요청

http://localhost:8080/setHeaderExample

결과

HTTP/1.1 200 OK
X-Custom-Header: CustomValue
Content-Type: text/plain;charset=UTF-8
Response Body: Header: X-Custom-Header=CustomValue

✓ addHeader(String name, String value)

```
@WebServlet(name = "addHeaderServlet", urlPatterns = "/addHeaderExample")  
public class AddHeaderServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {  
        response.addHeader("X-Additional-Header", "AdditionalValue");  
        response.getWriter().write("Header: X-Additional-Header=AdditionalValue");  
    }  
}
```

요청

http://localhost:8080/addHeaderExample

결과

HTTP/1.1 200 OK
X-Additional-Header: AdditionalValue
Content-Type: text/plain;charset=UTF-8
Response Body: Header : X-Additional-Header=AdditionalValue

✓ setContentType(String type)

```
@WebServlet(name = "setContentTypeServlet", urlPatterns = "/setContentTypeExample")
public class SetContentTypeServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.setContentType("application/json");
        response.getWriter().write("{\"message\":\"Content type set to JSON\"}");
    }
}
```

요청

http://localhost:8080/setContentTypeExample

결과

HTTP/1.1 200 OK

Content-Type: application/json

Response Body: {"message":"Content type set to JSON"}

✓ getWriter()

```
@WebServlet(name = "GetWriterServlet", urlPatterns = "/getWriterExample")
public class GetWriterServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        PrintWriter out = response.getWriter();
        out.write("This is body written using getWriter().");
    }
}
```

요청

http://localhost:8080/addHeaderExample

결과

HTTP/1.1 200 OK

Content-Type: text/plain;charset=UTF-8

Response Body: This is body written using getWriter().

✓ sendRedirect(String location)

```
@WebServlet(name = "sendRedirectServlet", urlPatterns = "/sendRedirectExample")
public class SendRedirectServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        response.sendRedirect("/newLocation");
    }
}
```

요청

http://localhost:8080/sendRedirectExample

결과

HTTP/1.1 302 Found
Location: /newLocation

✓ addCookie(Cookie cookie)

```
@WebServlet(name = "AddCookieServlet", urlPatterns = "/addCookieExample")
public class AddCookieServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        Cookie cookie = new Cookie("userToken", "abcd1234");
        cookie.setMaxAge(3600); // 쿠키 유효 기간 1시간
        response.addCookie(cookie);
        response.getWriter().write("Cookie added: userToken=abcd1234");
    }
}
```

요청

http://localhost:8080/sendRedirectExample

결과

HTTP/1.1 200 OK
Set-Cookie: userToken=abcd1234; Max-Age=3600
Content-Type: text/plain;charset=UTF-8
Response Body: Cookie added: userToken=abcd1234

HttpServletResponse - 응답 처리

<https://github.com/onjsdnjs/servlet/tree/httpResponseServletProcess>

✓ 개요

- HttpServletResponse는 응답을 처리하는 방식으로 단순 텍스트 응답, HTML 화면 처리 응답, HTTP 본문 응답 등 세 가지로 나누어 구분 할 수 있다
- 스프링에서도 응답 패턴이 이 세 가지 범주에서 크게 벗어나지 않으며 사용하기 쉽게 추상화 해서 제공하고 있다

✓ 단순 텍스트 응답

- 간단한 문자열 데이터를 응답 본문으로 클라이언트에 전송

```
@WebServlet(name = "simpleTextResponseServlet", urlPatterns = "/textResponse")
public class SimpleTextResponseServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        // 응답 Content-Type 설정
        response.setContentType("text/plain;charset=UTF-8");
        // 응답 본문 작성
        PrintWriter writer = response.getWriter();
        writer.write("Hello, this is a plain text response.");
        writer.close();
    }
}
```

요청

```
http://localhost:8080/textResponse
```

결과

```
HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 39

Hello, this is a plain text response.
```

✓ HTML 화면 처리 응답

- HTML 컨텐츠를 렌더링하여 브라우저가 화면에 표시하도록 전송

```
@WebServlet(name = "htmlResponseServlet", urlPatterns = "/htmlResponse")
public class HtmlResponseServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        // 응답 Content-Type 설정
        response.setContentType("text/html;charset=UTF-8");

        // 응답 본문 작성
        PrintWriter writer = response.getWriter();
        writer.write("<!DOCTYPE html>");
        writer.write("<html><head><title>HTML Response</title></head>");
        writer.write("<body>");
        writer.write("<h1>Welcome to the HTML Response</h1>");
        writer.write("<p>This is an HTML response.</p>");
        writer.write("</body></html>");
        writer.close();
    }
}
```

요청

```
http://localhost:8080/htmlResponse
```

결과

HTTP/1.1 200 OK

Content-Type: text/html;charset=UTF-8

```
<!DOCTYPE html>
<html>
<head><title>HTML Response</title></head>
<body>
<h1>Welcome to the HTML Response</h1>
<p>This is an HTML response.</p>
</body>
</html>
```

✓ HTTP 본문 응답

- JSON, XML, 바이너리 데이터 등 다양한 형식의 HTTP 응답 본문 데이터를 전송

```
@WebServlet(name = "JsonResponseServlet", urlPatterns = "/jsonResponse")
public class JsonResponseServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        // 응답 Content-Type 설정
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        // User 객체 생성
        User user = new User("leaven", 30, "leaven@example.com");
        // ObjectMapper를 사용하여 User 객체를 JSON으로 변환
        ObjectMapper objectMapper = new ObjectMapper();
        String jsonResponse = objectMapper.writeValueAsString(user);
        // 응답 본문 작성
        PrintWriter writer = response.getWriter();
        writer.write(jsonResponse);
        writer.close();
    }
}
```

요청

```
http://localhost:8080/jsonResponse
```

결과

HTTP/1.1 200 OK

Content-Type: application/json; charset=UTF-8

```
{
    "name": "leaven",
    "age": 30,
    "email": "leaven@example.com"
}
```



스프링 웹 MVC 완전 정복

✓ 스프링 웹 MVC 초기화

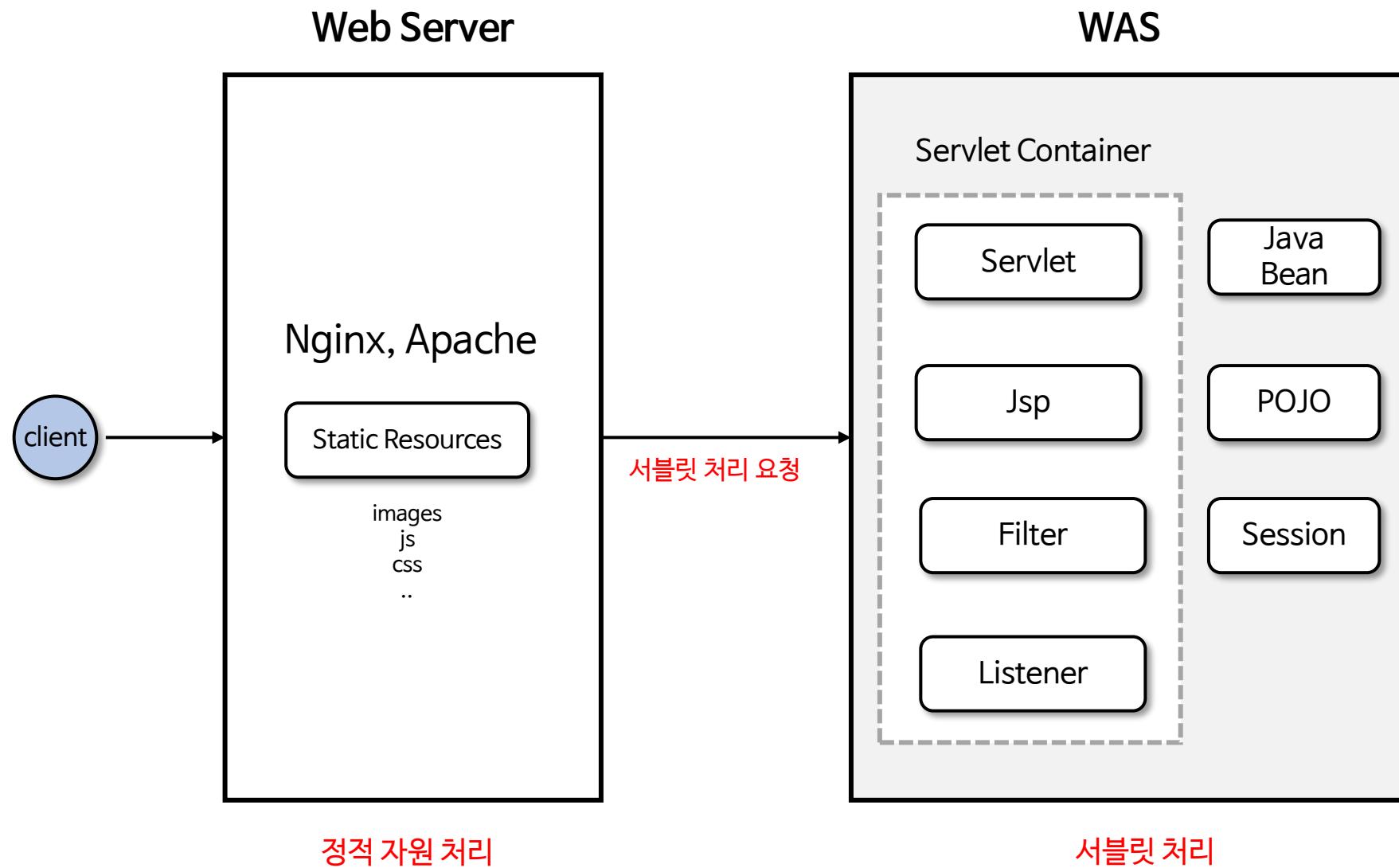
1. 서블릿 컨테이너 & 스프링 컨테이너
2. 초기화 클래스들

서블릿 컨테이너 & 스프링 컨테이너

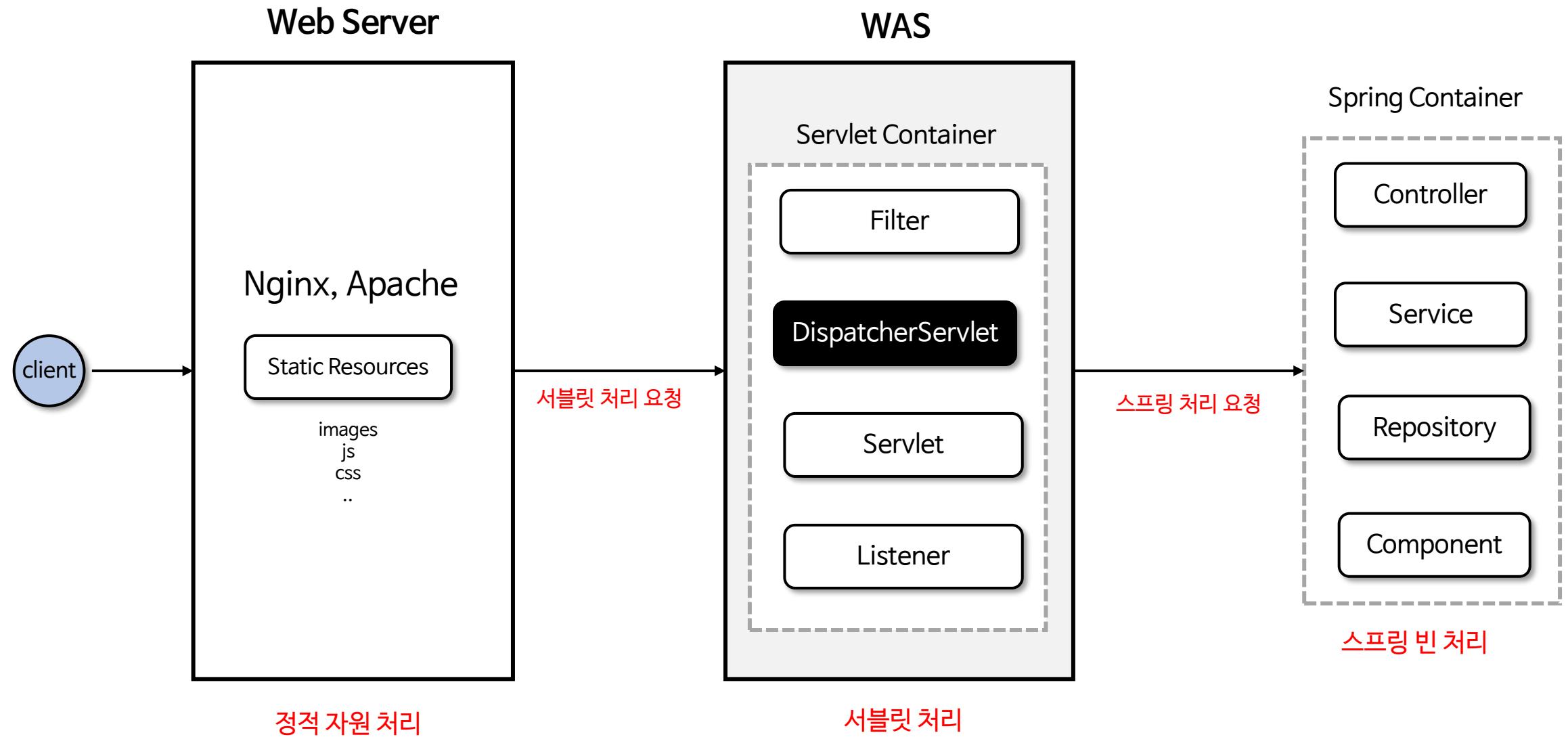
<https://github.com/onjsdnjs/servlet-spring>

<https://github.com/onjsdnjs/spring-mvc-master/tree/서블릿-스프링-연결>

✓ 웹 어플리케이션 서버 구조 - 1

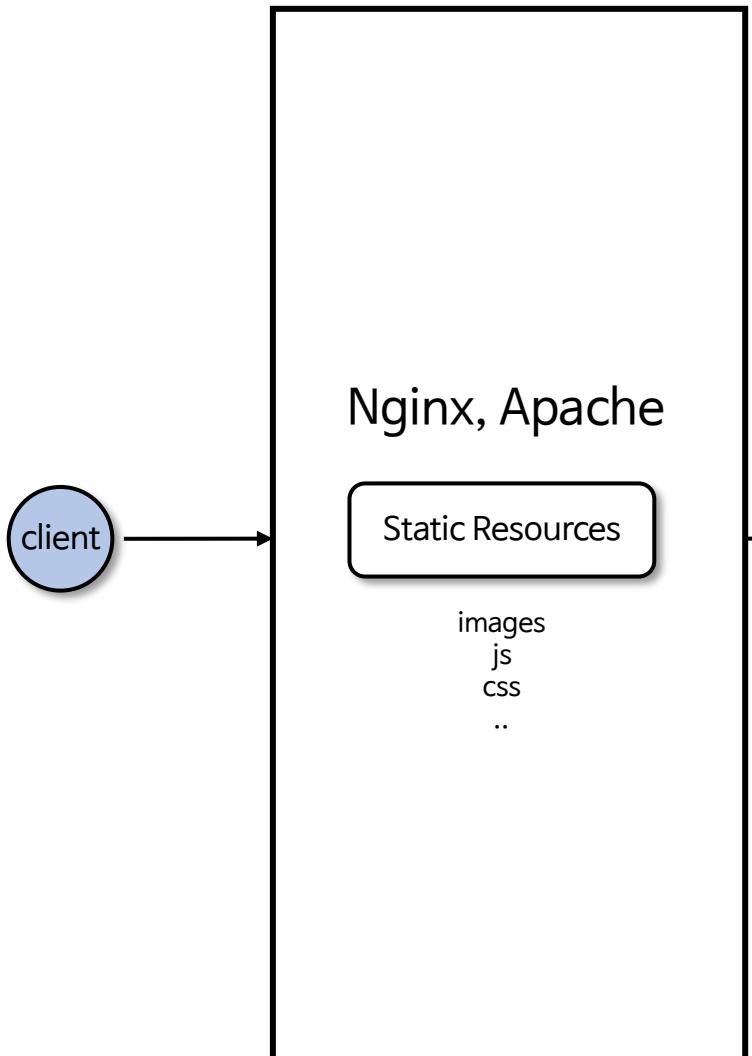


✓ 웹 어플리케이션 서버 구조 - 2

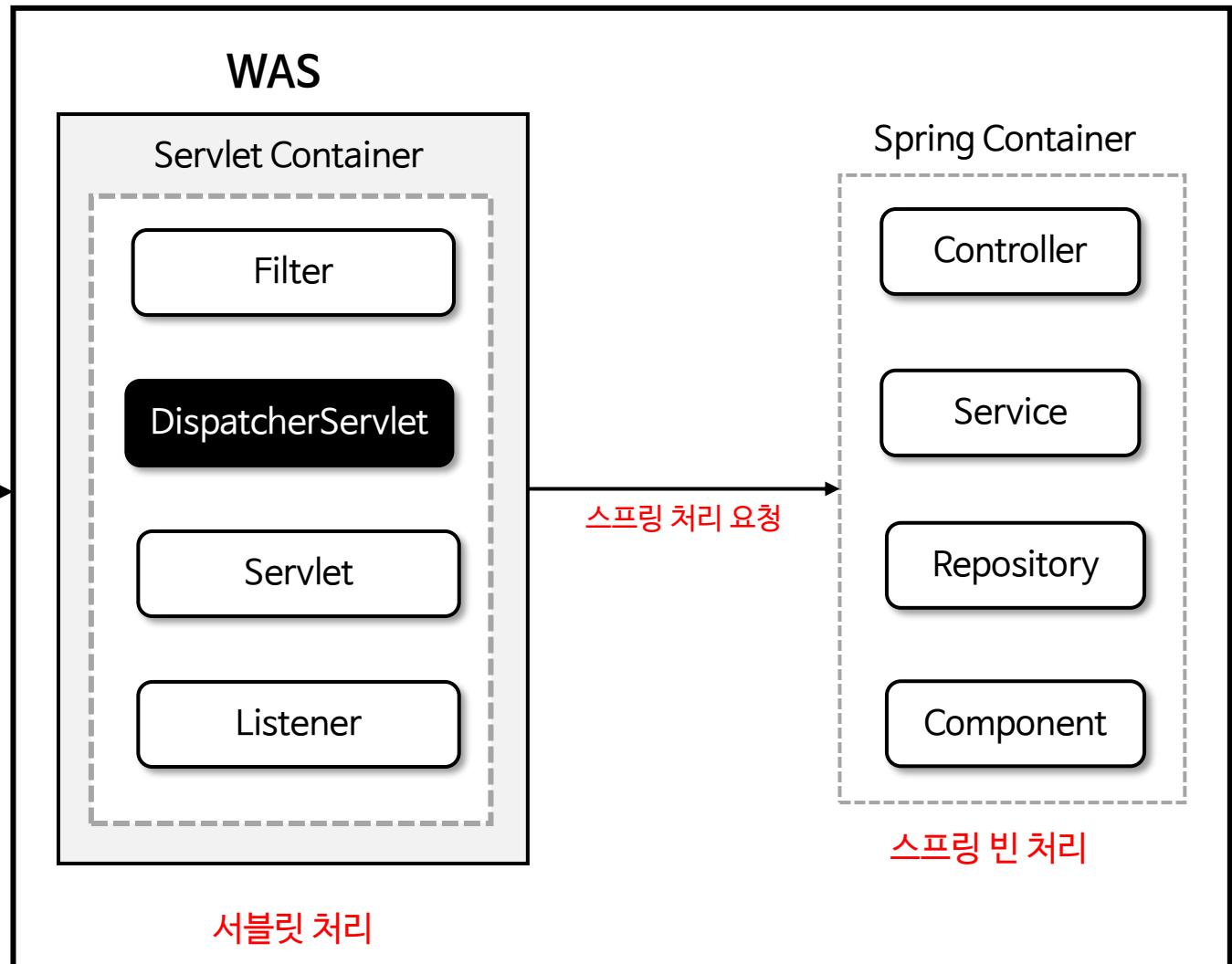


✓ 웹 어플리케이션 서버 구조 - 3

Web Server

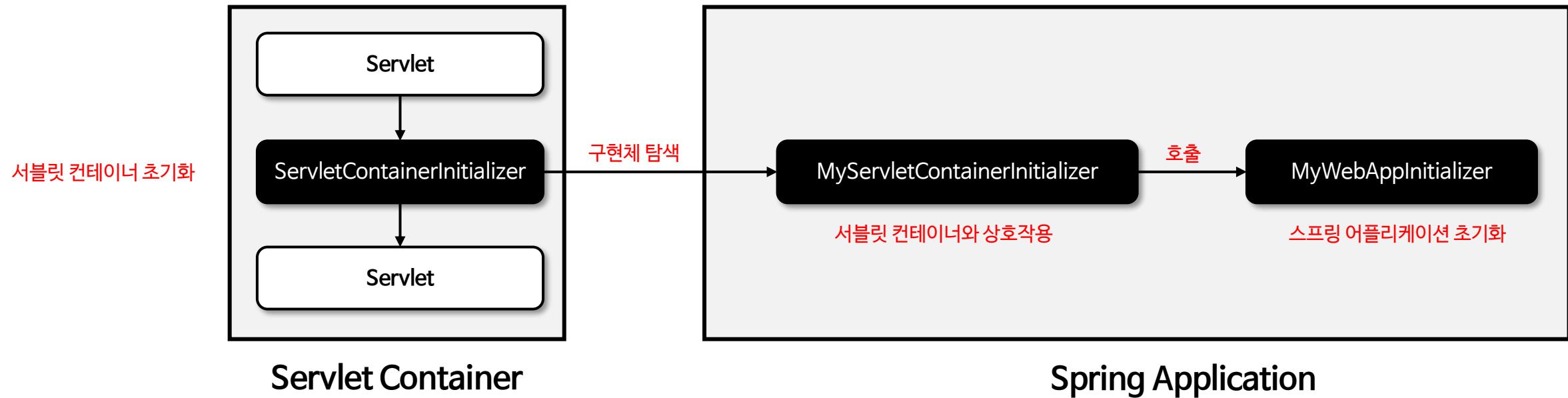


Spring Boot



✓ 서블릿 컨테이너 & 스프링 컨테이너 연결

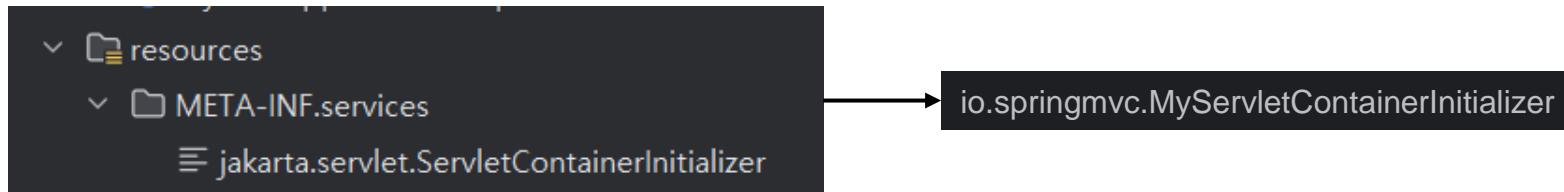
- 1) WAS 가 구동되면 서블릿 컨테이너가 META-INF/services/jakarta.servlet.ServletContainerInitializer 파일을 검색하여 서블릿 3.0 ServletContainerInitializer 인터페이스를 구현한 클래스를 로드한다
- 2) ServletContainerInitializer 구현체는 @HandlesTypes(MyWebApplInitializer.class) 와 같이 설정을 할 수 있으며 MyWebApplInitializer 를 호출하여 스프링 어플리케이션을 초기화 한다



✓ ServletContainerInitializer

```
public interface ServletContainerInitializer {  
    void onStartup(Set<Class<?>> c, ServletContext ctx) throws ServletException;  
}
```

- 1) /META-INF/services/ jakarta.servlet.ServletContainerInitializer 에 클래스 정보를 입력하면 초기화 시 해당 파일이 있는지 검색한 후 실행한다



- 2) ServletContainerInitializer 클래스에 @HandlesTypes 어노테이션을 붙이면 주어진 타입에 대해 컨테이너가 자동으로 스캐닝을 수행한다

```
@HandlesTypes(MyWebAppInitializer.class) // 특정 타입을 다룰 수 있다.  
public class MyServletContainerInitializer implements ServletContainerInitializer {  
    @Override  
    public void onStartup(Set<Class<?>> c, ServletContext ctx) throws ServletException {  
        for (Class<?> initClass : c) {  
            MyWebAppInitializer initializer  
                = (MyWebAppInitializer) initClass.getDeclaredConstructor().newInstance();  
            initializer.onStartup(ctx);  
        }  
    }  
}
```

```
public class MyWebAppInitializerImpl implements MyWebAppInitializer {  
    @Override  
    public void onStartup(ServletContext context) throws ServletException {  
        // 루트 컨텍스트 설정  
        // 서블릿 전용 컨텍스트 설정  
    }  
}
```

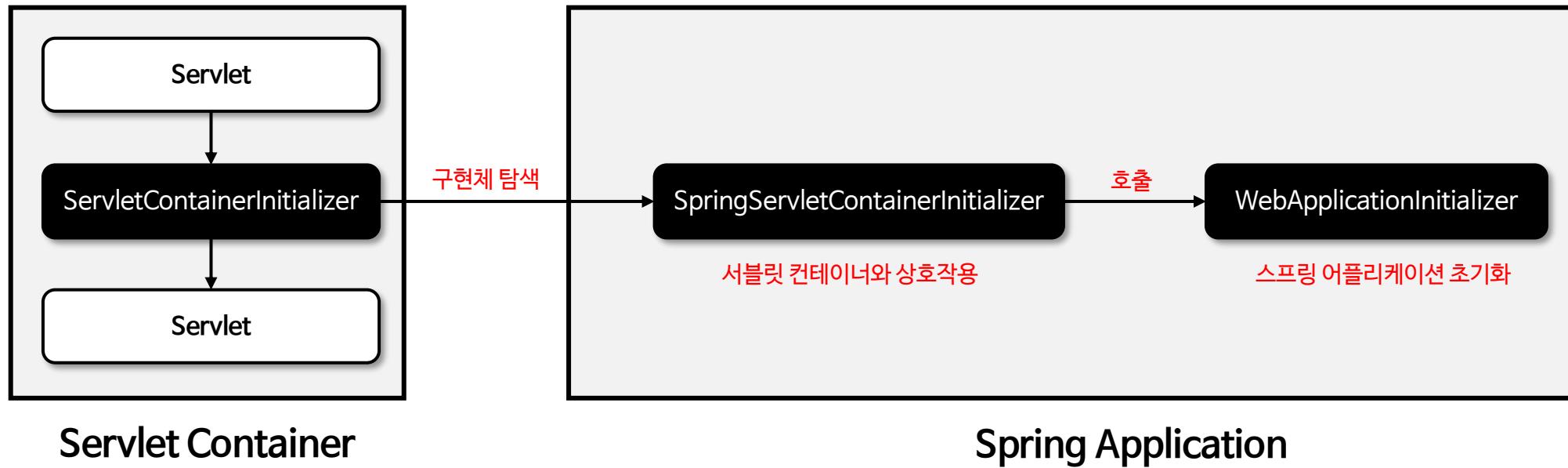
✓ SpringServletContainerInitializer

- 스프링은 SpringServletContainerInitializer라는 구현체를 제공하며 이는 스프링 애플리케이션에서 서블릿 컨테이너와의 초기 상호작용을 담당한다
- SpringServletContainerInitializer는 @HandlesTypes 어노테이션에 WebApplicationInitializer 타입이 선언되어 있으며 이는 WebApplicationInitializer 인터페이스를 구현한 클래스를 자동으로 탐색하고 이를 호출하여 스프링 어플리케이션을 초기화 한다

```
@HandlesTypes(WebApplicationInitializer.class)
```

```
public class SpringServletContainerInitializer implements ServletContainerInitializer {  
    // WebApplicationInitializer 타입의 클래스를 호출  
}
```

```
> 0 = {Class@3764} "class org.springframework.web.servlet.support.AbstractDispatcherServletInitializer" ... Navigate  
> 1 = {Class@3765} "class org.springframework.web.context.AbstractContextLoaderInitializer" ... Navigate  
> 2 = {Class@3766} "class org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer" ... Navigate  
> 3 = {Class@3767} "class org.springframework.web.server.adapter.AbstractReactiveWebInitializer" ... Navigate
```



✓ WebApplicationInitializer

- 스프링 애플리케이션이 구동되면 WebApplicationInitializer 타입의 클래스가 실행되고 여기서 스프링 컨테이너 초기화 및 설정 작업이 이루어진다

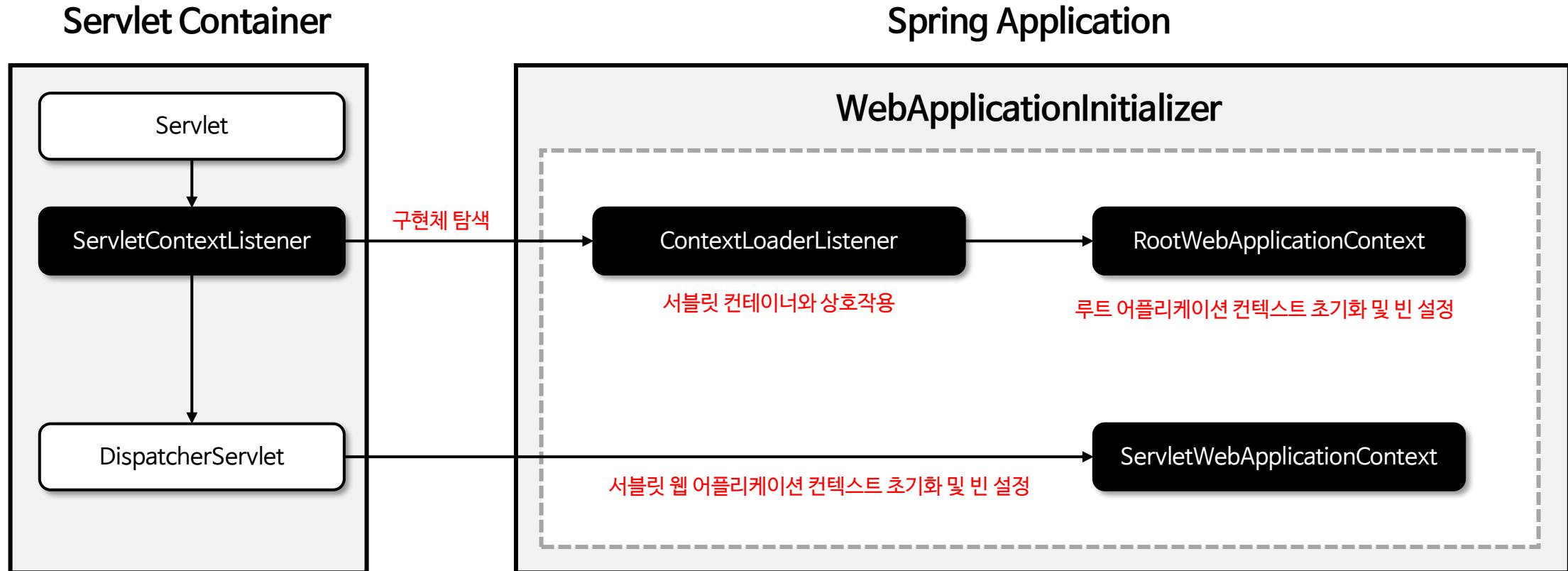
```
public interface WebApplicationInitializer {  
    void onStartup(ServletContext servletContext) throws ServletException;  
}
```

```
public class MyWebAppInitializer implements WebApplicationInitializer {  
    @Override  
    public void onStartup(ServletContext servletContext) throws ServletException {  
  
        // 루트 컨텍스트 설정  
        AnnotationConfigWebApplicationContext rootContext = new AnnotationConfigWebApplicationContext();  
        rootContext.register(RootConfig.class);  
        servletContext.addListener(new ContextLoaderListener(rootContext));  
  
        // 서블릿 컨텍스트 설정  
        AnnotationConfigWebApplicationContext servletContext = new AnnotationConfigWebApplicationContext();  
        context.setServletContext(servletContext);  
        context.register(AppConfig.class);  
        ServletRegistration.Dynamic servlet = servletContext.addServlet("dispatcher", new DispatcherServlet(context));  
        servlet.setLoadOnStartup(1);  
        servlet.addMapping("/");  
    }  
}
```



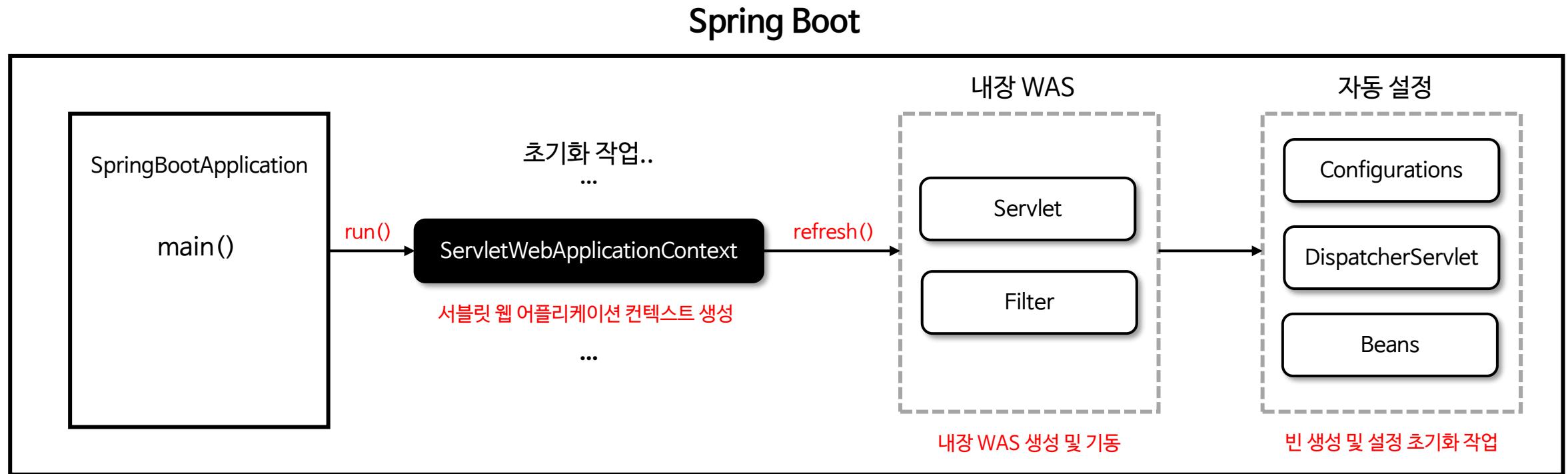
스프링 컨테이너 초기화 및 설정

✓ 스프링 어플리케이션 초기화



✓ Spring Boot 어플리케이션 초기화

- 스프링 부트 애플리케이션이 구동되면 내부적으로 WAS가 실행되고 자동설정에 의해 스프링 컨테이너 초기화 및 설정 작업이 이루어진다



- 스프링 부트는 스프링 컨테이너 객체 생성 및 초기화 작업을 자동으로 수행하며 여러 다양한 빈들을 자동설정에 의해 생성해 준다
- 스프링 부트는 웹 어플리케이션 컨텍스트 하나만 생성해서 빈들을 관리한다
- 스프링 부트는 기본 스프링 프레임워크와는 달리 `DispatcherServlet`를 빈으로 등록해서 관리하고 있다

초기화 클래스들

<https://github.com/onjsdnjs/spring-mvc-master/tree/초기화-클래스들>

✓ 개요

- 스프링 부트는 웹 MVC 초기화와 실행을 위한 여러 빈들을 생성하고 초기 값을 설정하기 위한 여러 클래스들을 정의하고 있다

✓ WebMvcProperties

- 스프링 MVC의 여러 속성을 환경설정파일(properties, yml)을 통해 설정할 수 있다

```
@ConfigurationProperties(prefix = "spring.mvc")
public class WebMvcProperties {
    ...
    private final View view = new View();
    private final Servlet servlet = new Servlet();
    ...
}
```



- application.yml

```
spring:
  mvc:
    view:
      prefix: /WEB-INF/views/
      suffix: .jsp
    servlet:
      load-on-startup: 1
```

✓ DispatcherServletAutoConfiguration

DispatcherServletAutoConfiguration

DispatcherServlet 자동 설정 클래스



DispatcherServlet 빈 등록



DispatcherServlet 서블릿 등록

ServletRegistrationBean을 상속받아 스프링 부트에서 서블릿과 관련된 설정(매핑, 부가 설정 등)을 수행한다

```
registration = {DispatcherServletRegistrationBean@5898} "dispatcherServlet urls=[/]"  
    path = "/"  
    servlet = {DispatcherServlet@5867}  
    urlMappings = {LinkedHashSet@5902} size = 1  
        0 = "/"  
        alwaysMapUrl = true  
        loadOnStartup = -1  
        multipartConfig = {MultipartConfigElement@5903}  
        name = "dispatcherServlet"
```

✓ DispatcherServlet 컨테이너 관리

- DispatcherServlet 은 스프링 MVC 에서 생성하는 유일한 서블릿으로서 서블릿 컨테이너 및 스프링 빈으로서 스프링 컨테이너에서 관리한다

Spring Container



Servlet Container



✓ WebMvcAutoConfiguration

WebMvcAutoConfiguration

- @EnableWebMvc 를 명시적으로 사용하지 않을 경우 스프링 부트 기반 스프링 MVC 설정을 자동으로 구성한다
- @EnableWebMvc 를 선언하면 WebMvcConfigurationSupport 빈이 생성되어 자동설정은 실행 되지 않는다

```
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
```

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
@Documented  
@Import(DelegatingWebMvcConfiguration.class)  
public @interface EnableWebMvc {  
}
```

상속관계

- 메시지 컨버터, 뷰 리졸버, 포맷터, 인터셉터, 리소스 핸들러 등 Web MVC에서 자주 사용하는 Bean들이 이 클래스를 통해 자동 구성된다
- @SpringBootApplication 만 붙였는데 모든 MVC 기능이 자동으로 작동할까?”라는 질문에 대한 답을 가장 직접적으로 보여주는 클래스

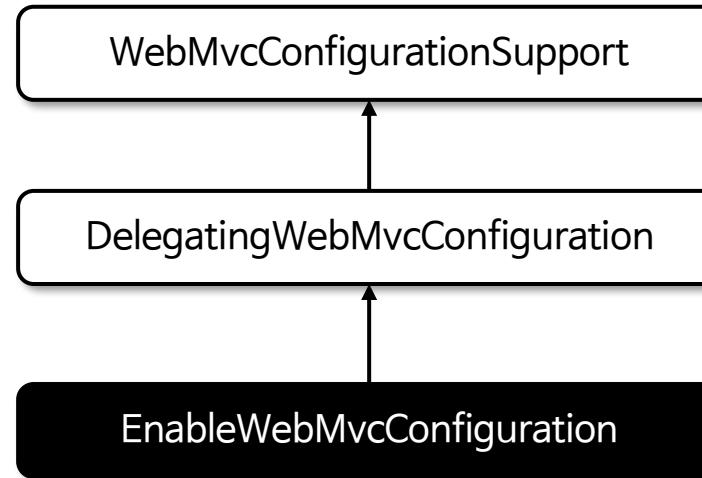
WebMvcAutoConfigurationAdapter

- WebMvcConfigurer 인터페이스를 구현한 추상 클래스로서 스프링 부트에서 제공하는 여러 기본 MVC 설정을 제공하며 (ViewResolvers, Formatter, ResourceHandler..) 사용자가 직접 WebMvcConfigurer 인터페이스를 직접 구현하여 커스트 마이징 할 수 있다

EnableWebMvcConfiguration

- 스프링 MVC의 핵심 설정을 제공하는 클래스로서 WebMvcConfigurationSupport 를 상속받아 스프링 MVC에서 필 요한 다양한 설정을 구성한다

✓ EnableWebMvcConfiguration



- 전통적인 스프링 MVC에서 `@EnableWebMvc`를 사용할 때 MVC 구성에 필요한 핵심 Bean 등록(핸들러 매핑, 핸들러 어댑터, 뷰 리졸버 등)을 담당하는 추상 클래스로서 스프링 MVC의 ‘기본 골격’이라고 볼 수 있다
- `WebMvcConfigurationSupport`를 확장하고 내부적으로 여러 개의 `WebMvcConfigurer`(개발자가 작성한 커스텀 설정 클래스들)를 찾아서 적용(Delegation)해 주는 역할을 한다
- `@EnableWebMvc` 또는 스프링 부트가 제공하는 자동 설정과 사용자가 정의한 `WebMvcConfigurer`들을 연결해 주는 위임 클래스라 볼 수 있다

1. 핸들러 매핑 및 핸들러 어댑터 설정:

- `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, `ExceptionHandlerExceptionResolver` 등의 다양한 핸들러 매핑과 어댑터를 설정한다

2. 뷰 리졸버(ViewResolver) 설정:

- 뷰 리졸버는 컨트롤러에서 반환된 뷰 이름을 실제 뷰로 변환하는 역할을 한다

3. MessageConverters 설정:

- 메시지 컨버터는 요청과 응답을 객체로 변환하거나 객체를 JSON이나 XML과 같은 형식으로 변환하는 역할을 한다

4. Validator와 Formatting 설정:

- `Validator`와 `ConversionService`를 설정하여 데이터 유효성 검사와 포매팅을 처리하는데 주로 폼 데이터 바인딩이나 데이터 검증에 사용된다

5. 인터셉터(Interceptors) 및 기타 커스터마이징 포인트:

- 다양한 인터셉터를 추가하거나 설정할 수 있는 확장 포인트를 제공한다. 이를 통해 요청 전후에 특정 로직을 삽입할 수 있다

6. 정적 리소스 제공 설정:

- 정적 리소스(예: CSS, JavaScript, 이미지 파일 등)를 제공하기 위한 설정을 포함하고 있다.

✓ WebMvcConfigurer

- WebMvcConfigurer는 Web MVC 관련 설정을 커스터마이징할 수 있는 다양한 메서드를 정의하고 있으며 스프링 부트가 기본적으로 제공하는 자동 구성의 일부만 맞춤형으로 설정할 수 있다
- 예를 들어 정적 리소스 매핑(addResourceHandlers), CORS 설정(addCorsMappings), 메시지 컨버터 등록(configureMessageConverters), 인터셉터 추가(addInterceptors) 등 MVC 전반에 걸친 세부 설정들을 손쉽게 추가하거나 변경할 수 있다

✓ 기본 구현

```
@Configuration
public class MyWebMvcConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/static/**").addResourceLocations("classpath:/my-static-files/")
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new MyCustomInterceptor()).addPathPatterns("/api/**")
            .excludePathPatterns("/api/public/**");
    }

    @Override
    public void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
        MappingJackson2HttpMessageConverter converter = new MappingJackson2HttpMessageConverter();
        converters.add(0, converter);
    }

    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> resolvers) {
        resolvers.add(new MyCustomArgumentResolver());
    }
}
```

WebMvcConfigurer	
(m)	extendHandlerExceptionResolvers (List<HandlerExceptionResolver>) void
(m)	addFormatters (FormatterRegistry) void
(m)	configureMessageConverters (List<HttpMessageConverter<?>>) void
(m)	configurePathMatch (PathMatchConfigurer) void
(m)	configureContentNegotiation (ContentNegotiationConfigurer) void
(m)	addResourceHandlers (ResourceHandlerRegistry) void
(m)	addCorsMappings (CorsRegistry) void
(m)	configureViewResolvers (ViewResolverRegistry) void
(m)	extendMessageConverters (List<HttpMessageConverter<?>>) void
(m)	configureAsyncSupport (AsyncSupportConfigurer) void
(m)	getMessageCodesResolver () MessageCodesResolver ?
(m)	addArgumentResolvers (List<HandlerMethodArgumentResolver>) void
(m)	configureDefaultServletHandling (DefaultServletHandlerConfigurer) void
(m)	addErrorResponseInterceptors (List<Interceptor>) void
(m)	addInterceptors (InterceptorRegistry) void
(m)	addReturnValueHandlers (List<HandlerMethodReturnValueHandler>) void
(m)	addViewControllers (ViewControllerRegistry) void
(m)	getValidator () Validator ?
(m)	configureHandlerExceptionResolvers (List<HandlerExceptionResolver>) void



스프링 웹 MVC 완전 정복

✓ 스프링 웹 MVC 기본

1. 아키텍처 이해
2. DispatcherServlet 개요
3. DispatcherServlet init() / service()

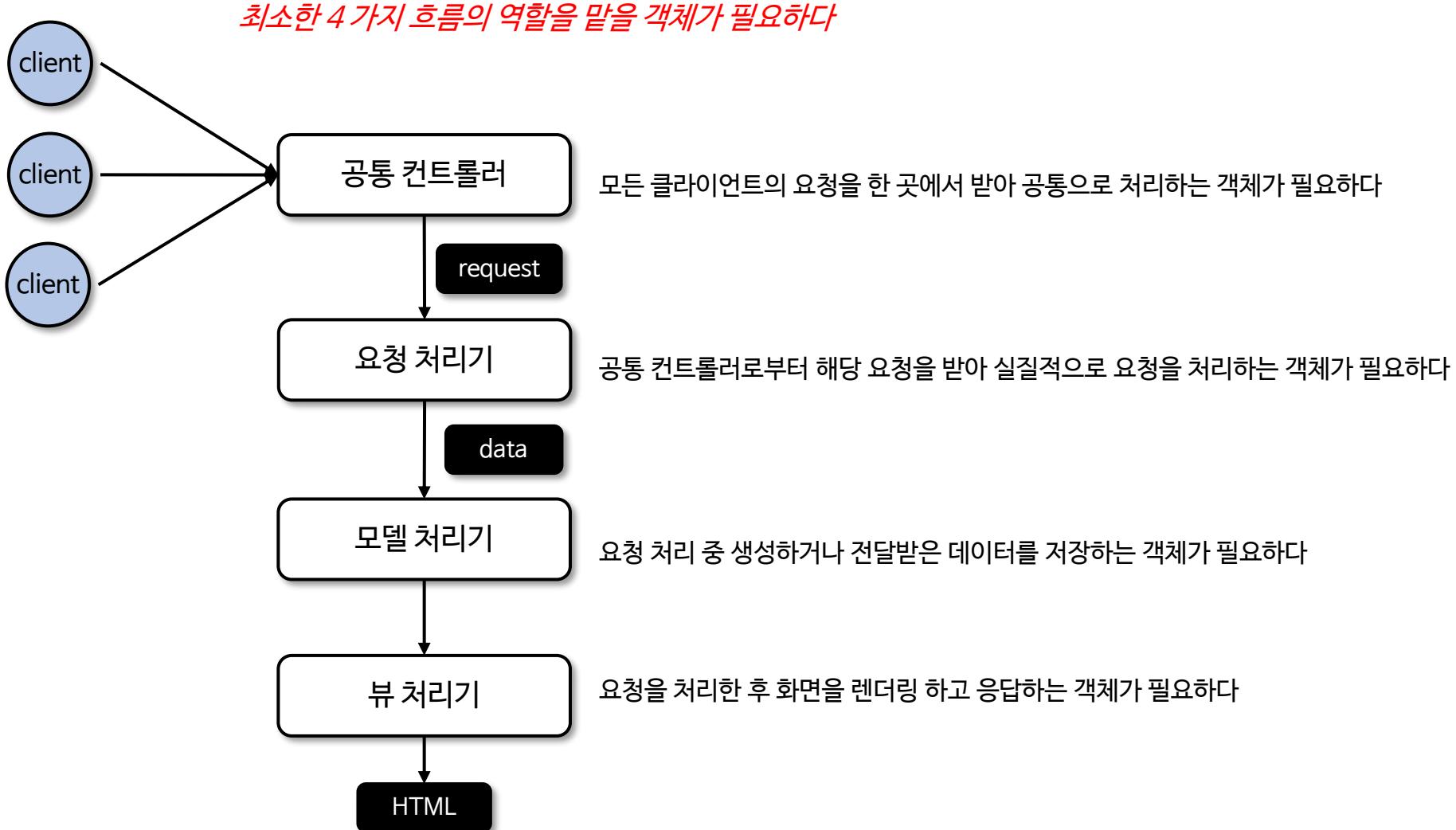
아키텍처 이해

<https://github.com/onjsdnjs/spring-mvc-master/tree/아키텍처이해>

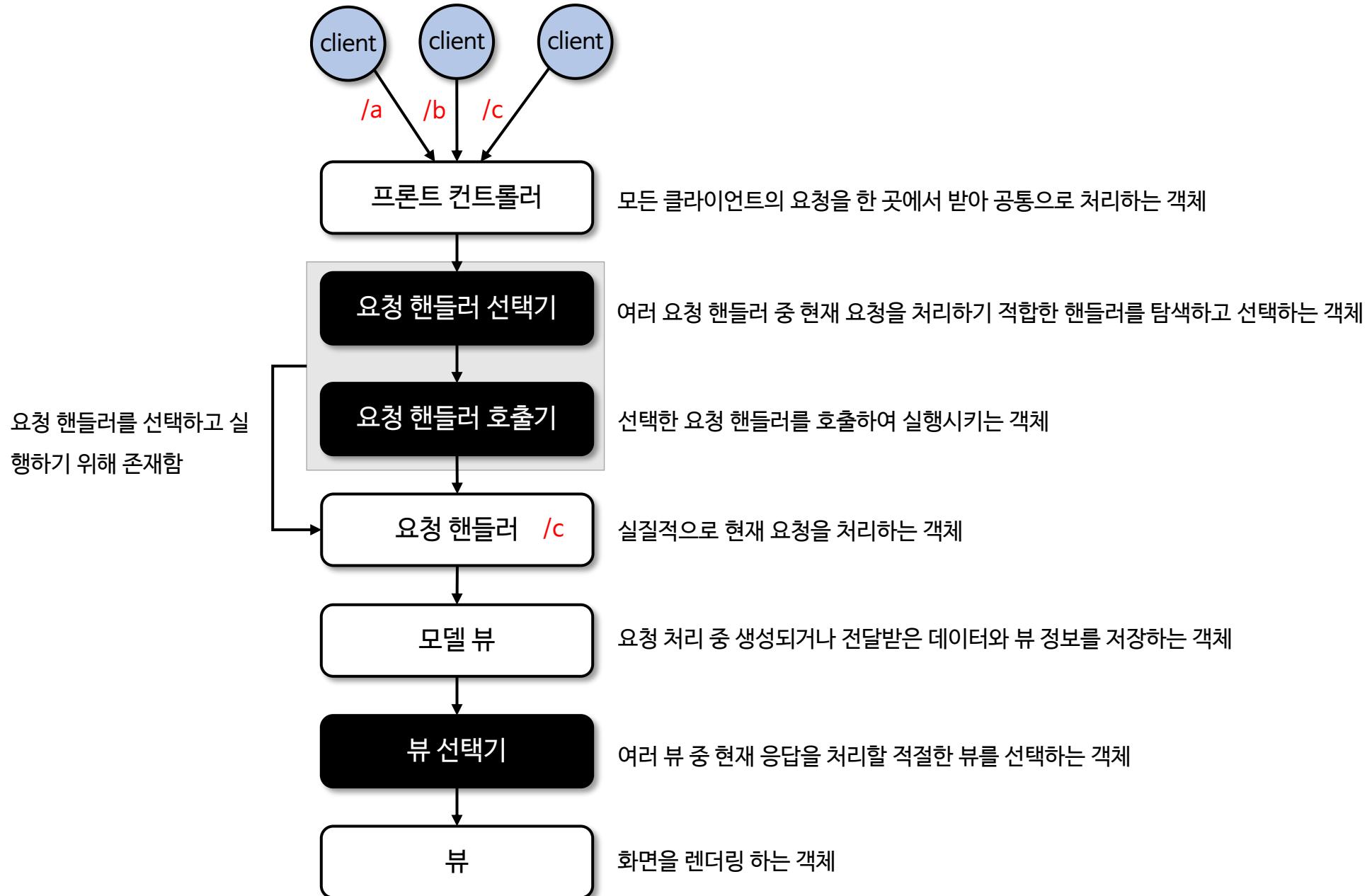
<https://github.com/onjsdnjs/servlet/tree/frontcontroller>

✓ 아키텍처 이해

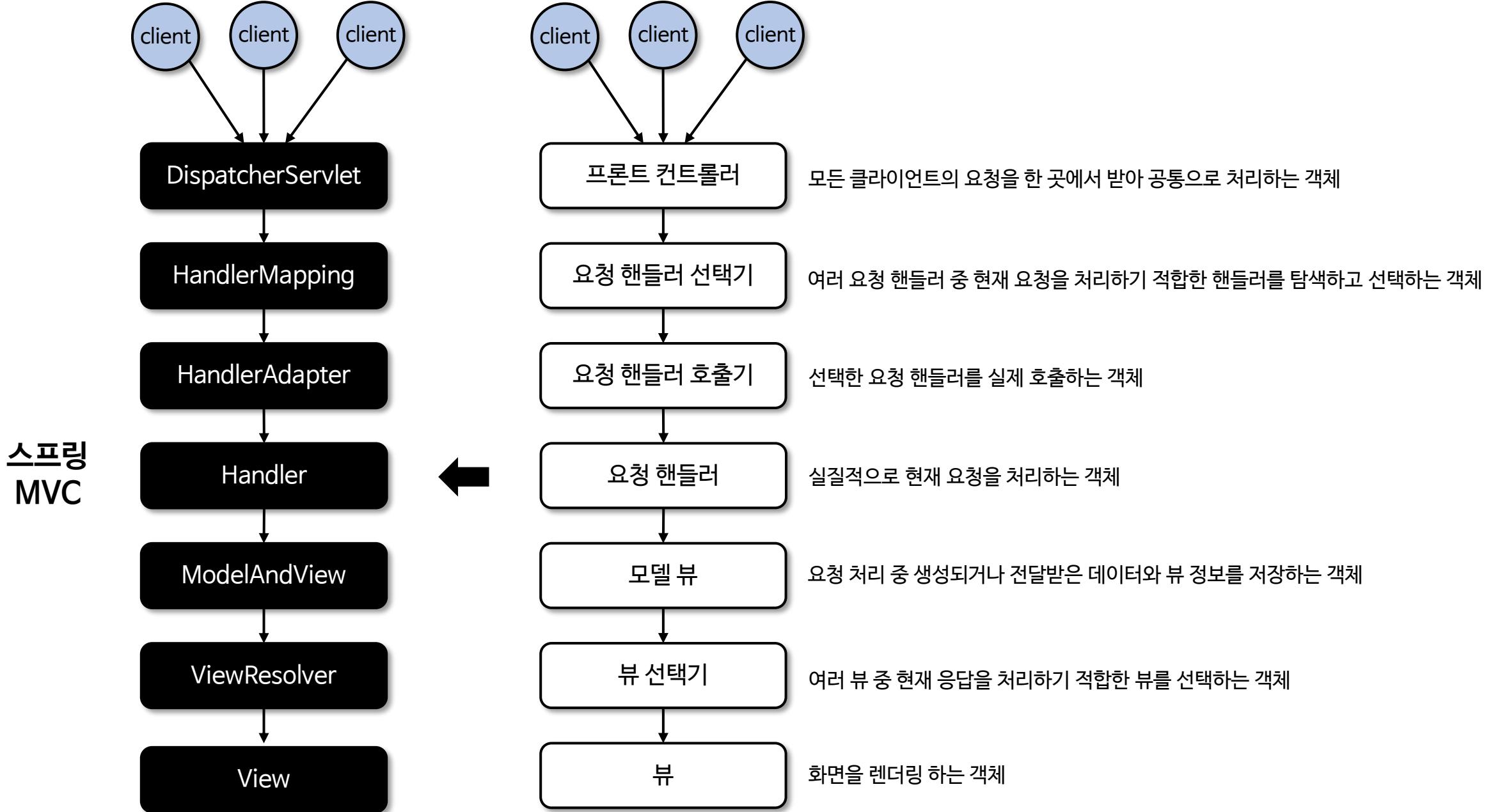
- 스프링 웹 MVC의 전체 아키텍처 구조를 이해하기 위해서 먼저 큰 흐름을 파악하고 세부적인 기능을 살펴보도록 한다
- 클라이언트가 요청을 하게 되면 서버에서 요청을 처리하기 위해서 어떤 기능들이 필요한지 나열해 보도록 한다



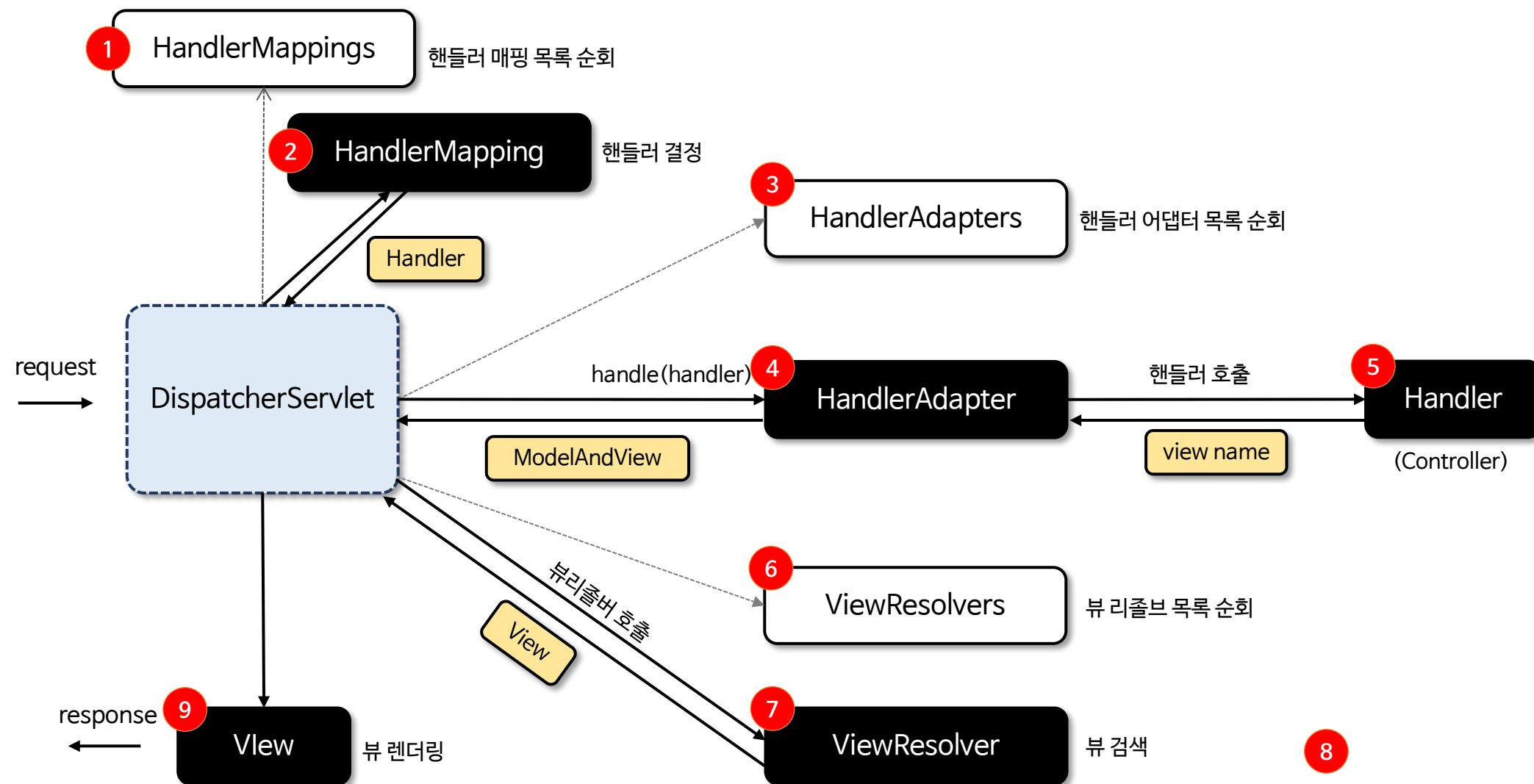
✓ 아키텍처 이해



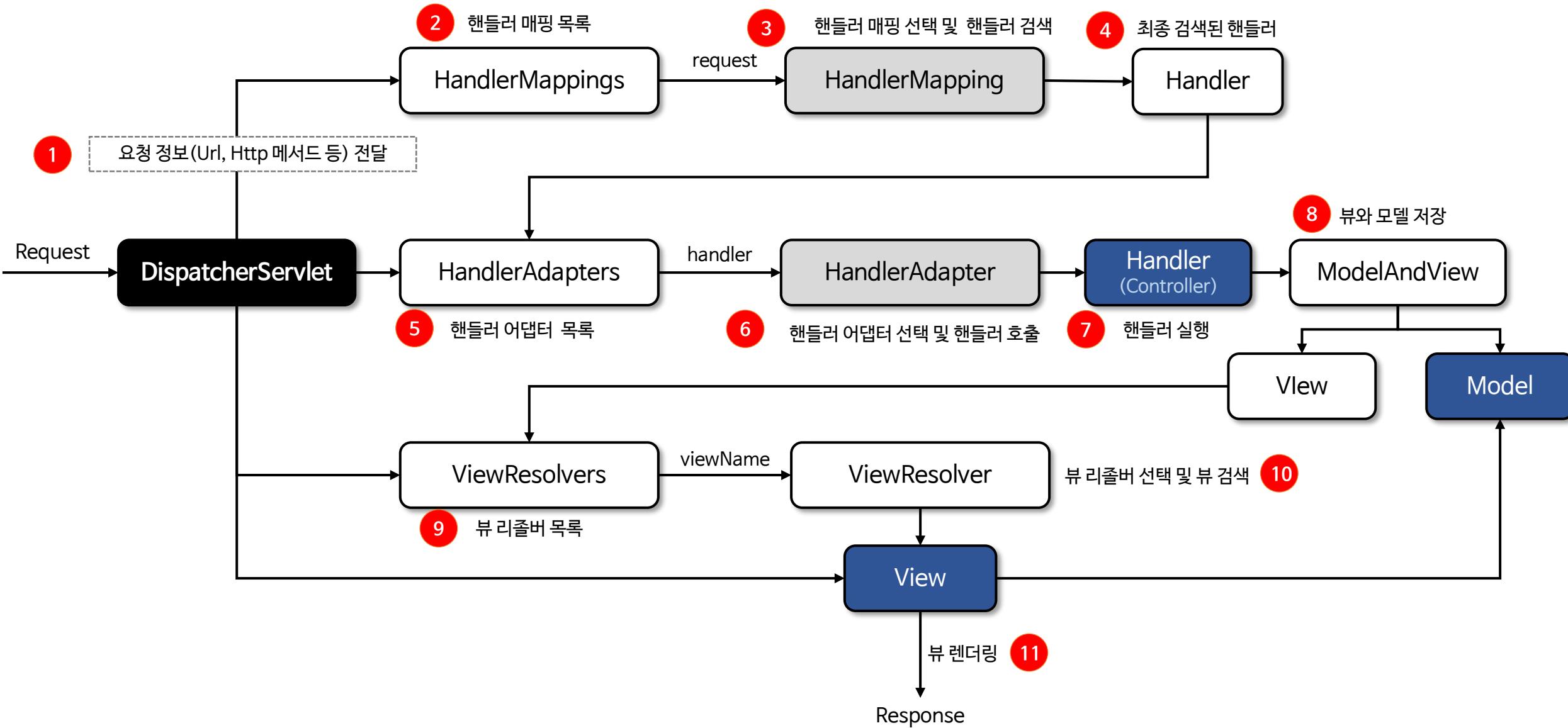
✓ 아키텍처 이해



✓ 전체 아키텍처

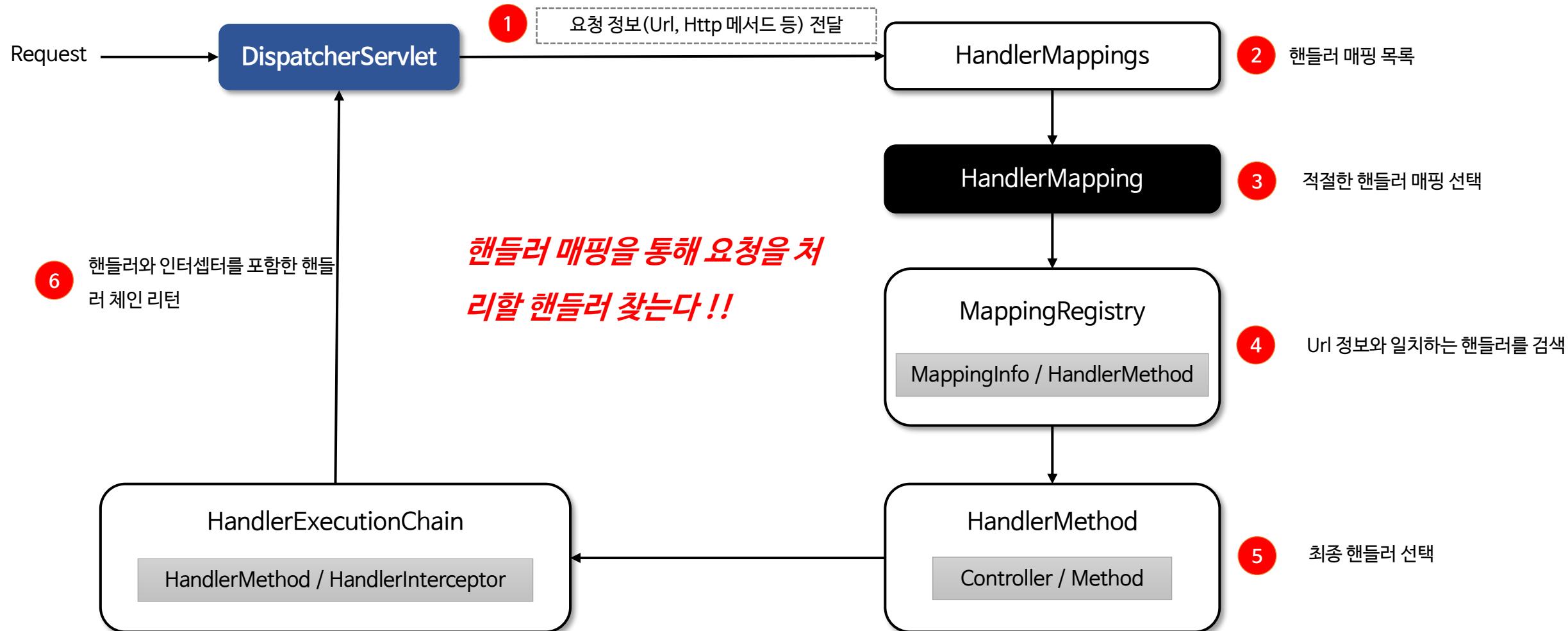


✓ 흐름도



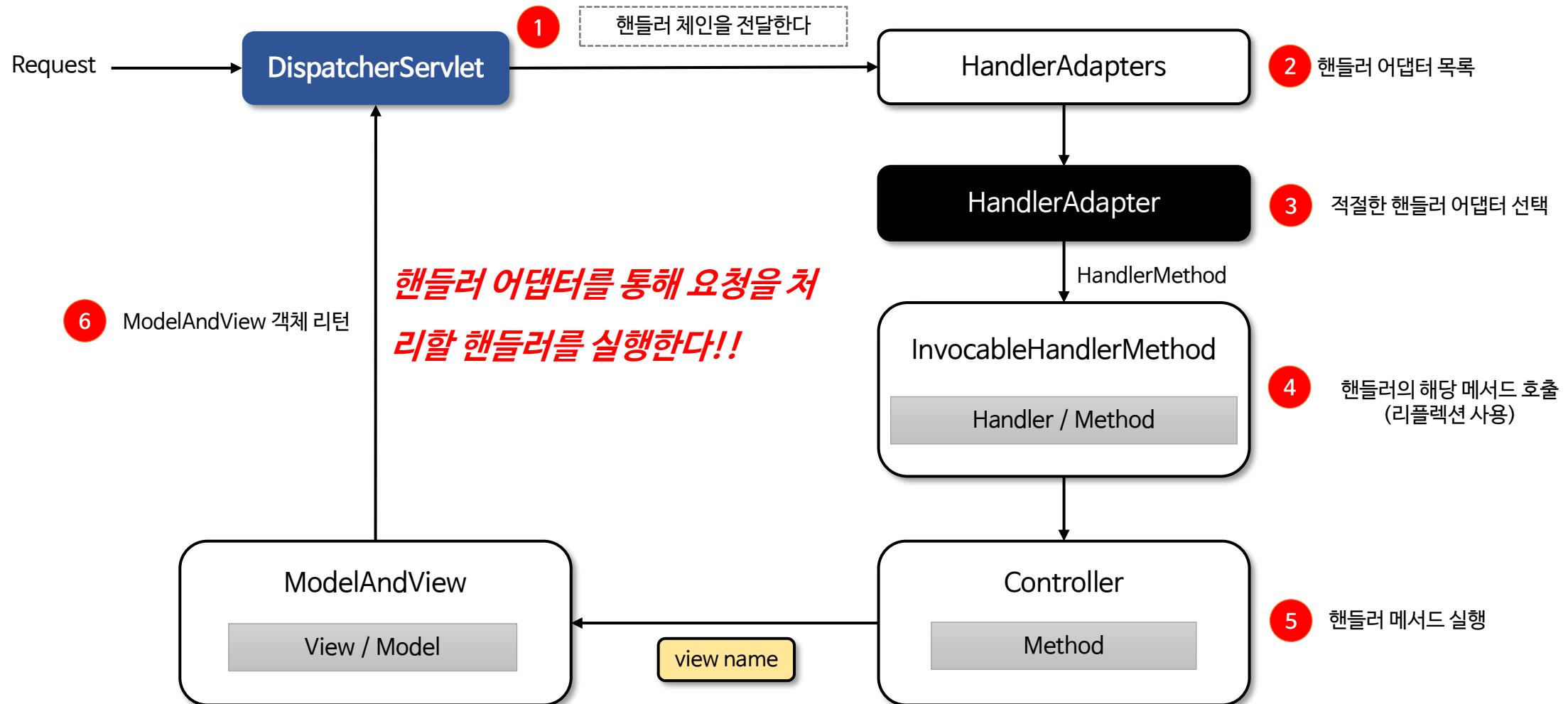
✓ HandlerMapping (핸들러 검색)

- 클라이언트의 요청을 처리할 핸들러를 검색한다
- 핸들러는 요청을 처리할 실제 빈 컨트롤러와 호출 메서드의 정보를 가지고 있다



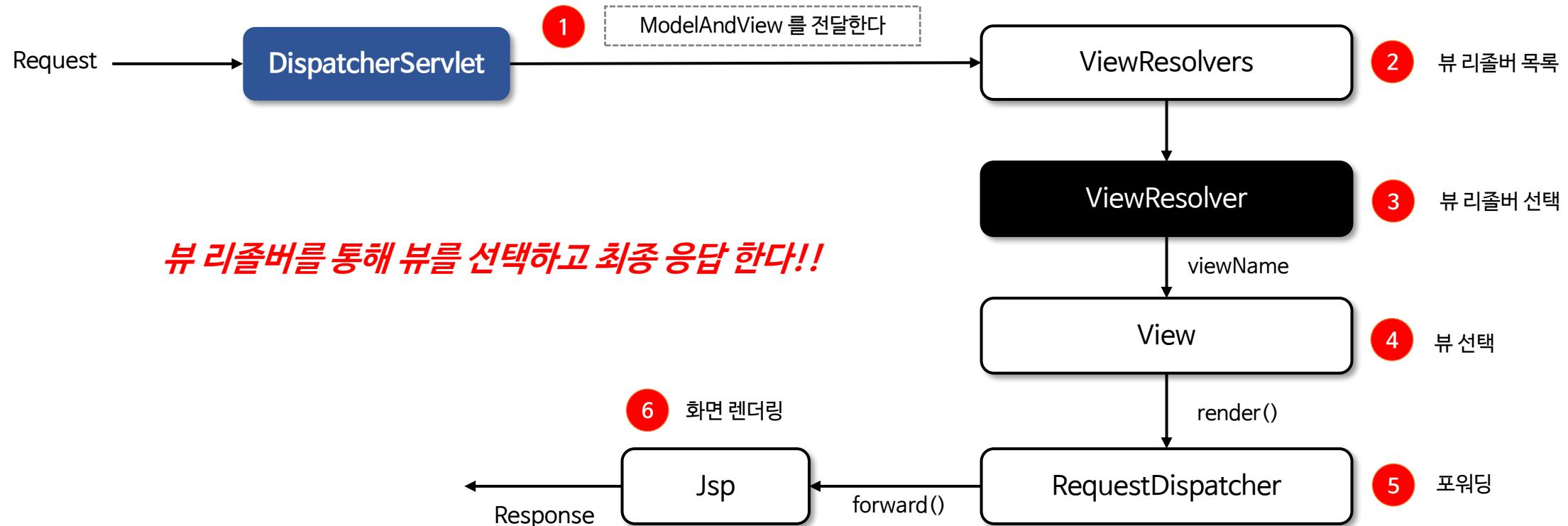
✓ HandlerAdapter (핸들러 호출)

- 클라이언트의 요청을 처리할 핸들러를 실행한다



✓ ViewResolver & View (화면 처리)

- 뷰 리졸버를 통해 뷰를 선택하고 최종 응답 한다



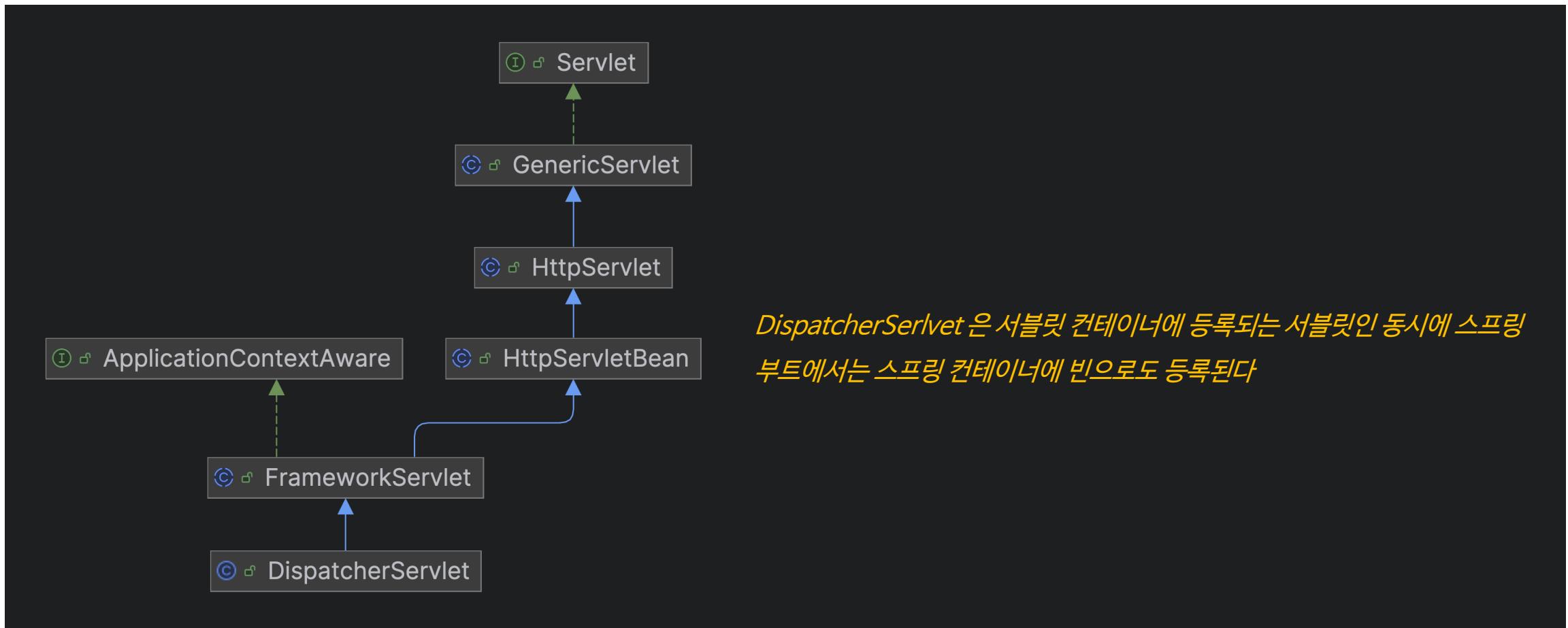
DispatcherServlet 개요

<https://github.com/onjsdnjs/spring-mvc-master/tree/아키텍처이해>

✓ 개요

- DispatcherServlet 은 스프링 MVC의 핵심 프론트 컨트롤러로 모든 HTTP 요청을 중앙에서 받아 처리하는 역할을 한다
- DispatcherServlet 은 요청을 적절한 핸들러(일반적으로 컨트롤러)로 라우팅하고 요청 처리 후에는 적절한 뷰를 선택하여 응답을 반환한다
- DispatcherServlet 은 핸들러 맵핑, 뷰 리졸버, 인터셉터 등을 조정하여 요청 처리 흐름을 관리하고 스프링 MVC 애플리케이션에서 중심적인 역할을 수행한다

✓ 클래스 계층도

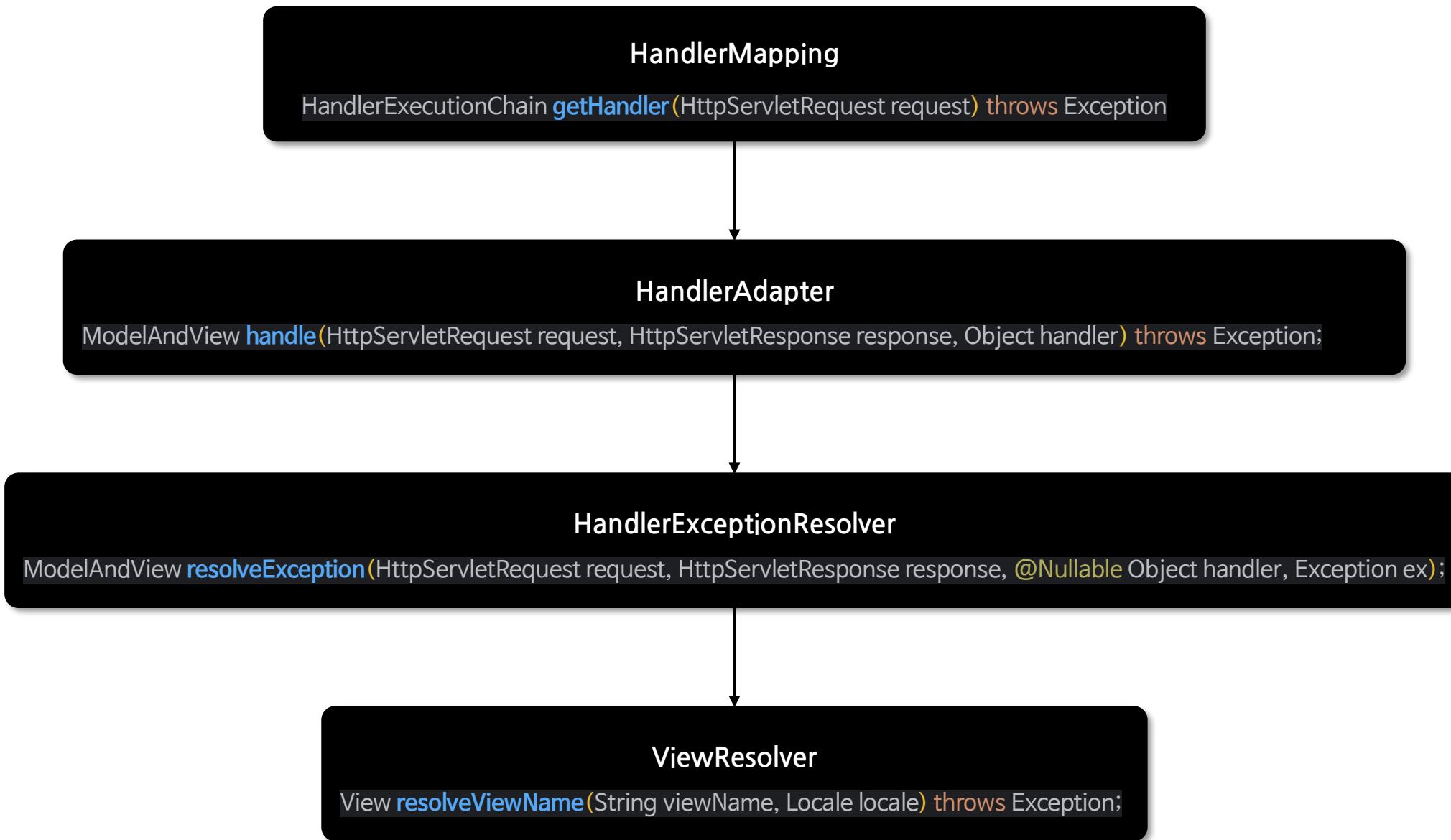


✓ 핵심 클래스들

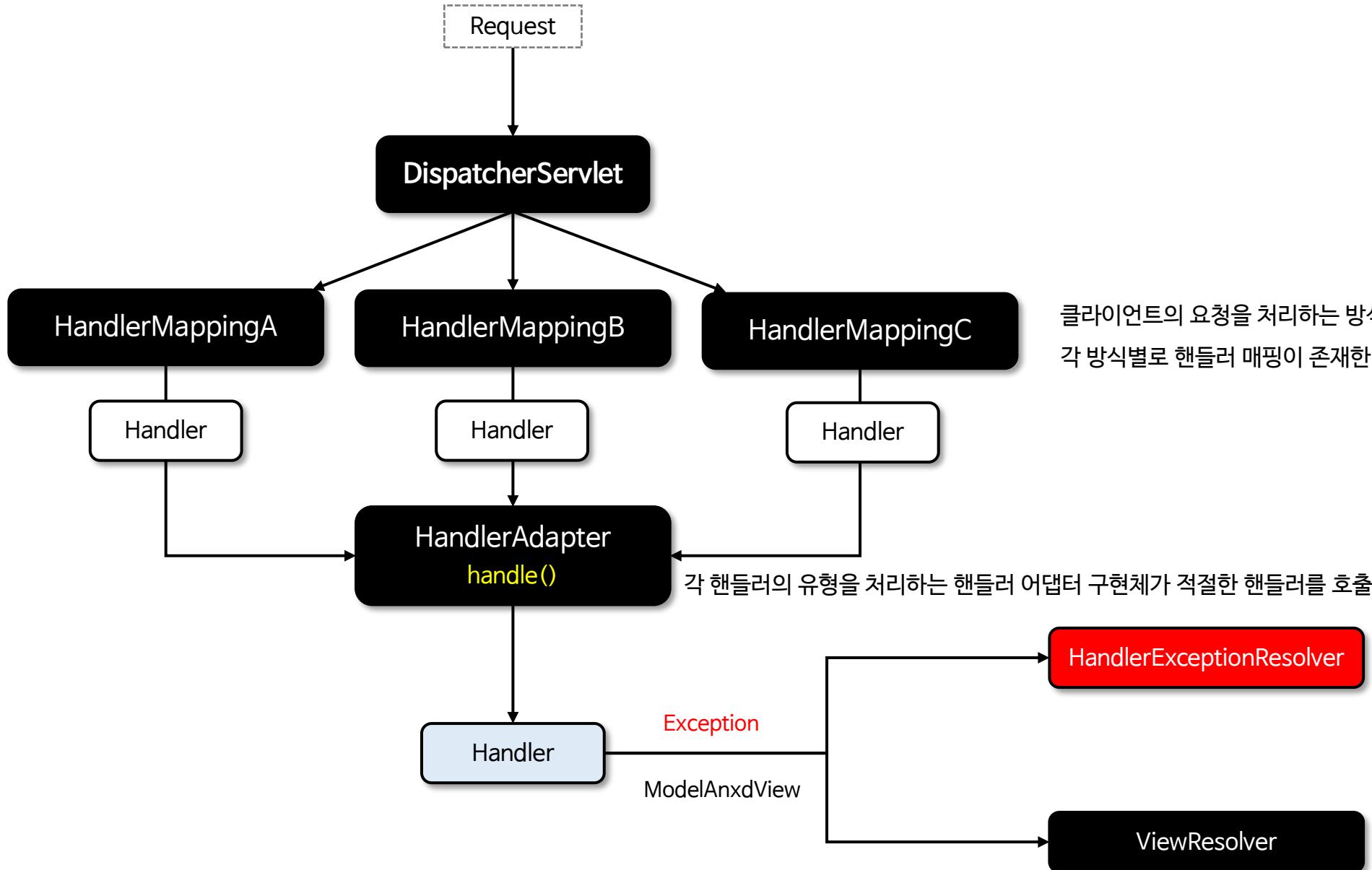
빈 유형	설명
HandlerMapping	클라이언트 요청을 적절한 핸들러(컨트롤러 등)와 매핑하고, 이 과정에서 요청 전후에 실행할 인터셉터 목록도 함께 관리한다
HandlerAdapter	DispatcherServlet가 요청에 매핑된 핸들러를 호출할 때 핸들러의 호출 방식에 관계없이 일관되게 동작할 수 있도록 처리한다
HandlerExceptionResolver	예외가 발생했을 때 이를 적절하게 처리하는 전략을 제공한다. 예를 들어, 특정 핸들러로 예외를 전달하거나, 오류 페이지를 보여주는 방식으로 처리할 수 있다
ViewResolver	컨트롤러가 반환한 뷰 이름을 실제 뷰(예: JSP 페이지, Thymeleaf 템플릿)로 변환하여 응답으로 렌더링할 수 있도록 한다
LocaleResolver, LocaleContextResolver	사용자가 요청한 언어와 지역 설정(로케일)을 해석하여 이 정보에 맞는 국제화된 뷰를 제공할 수 있게 한다(다국어처리)
MultipartResolver	브라우저에서 파일 업로드와 같은 멀티파트 요청을 처리한다
FlashMapManager	리다이렉션 후에 속성을 전달할 수 있도록 데이터를 임시로 저장하고 전달하는 역할을 한다 예를 들어, 폼 제출 후 리다이렉션된 페이지에서 메시지를 보여줄 때 사용할 수 있다

위에 나열한 대부분의 핵심 인터페이스는 사용자가 직접 구현하여 스프링의 기본 기능을 확장하거나 새롭게 구현 할 수 있다

✓ 핵심 인터페이스



✓ 핵심 클래스 흐름도



DispatcherServlet init() / service()

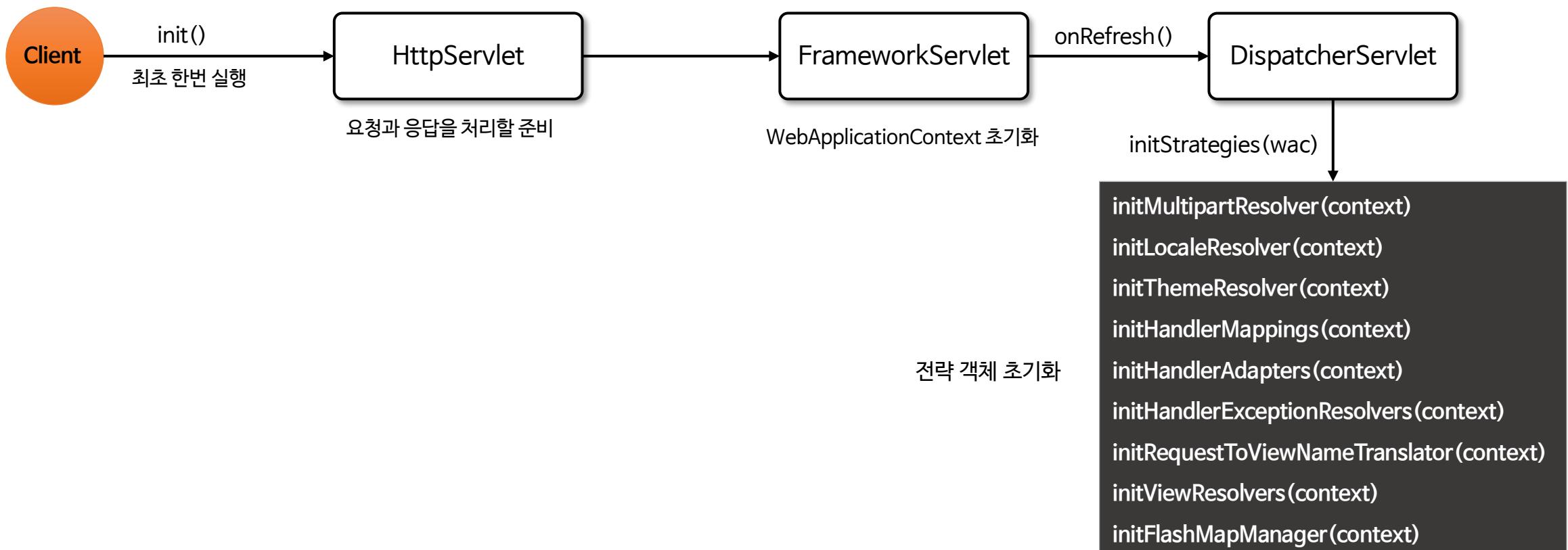
<https://github.com/onjsdnjs/spring-mvc-master/tree/아키텍처이해>

✓ 개요

- DispatcherServlet 은 요청이 시작되면 서블릿의 생명주기에 따라 `init()` 메서드와 `service()` 메서드가 실행되어 초기화 작업 및 실제 요청을 처리하게 된다.

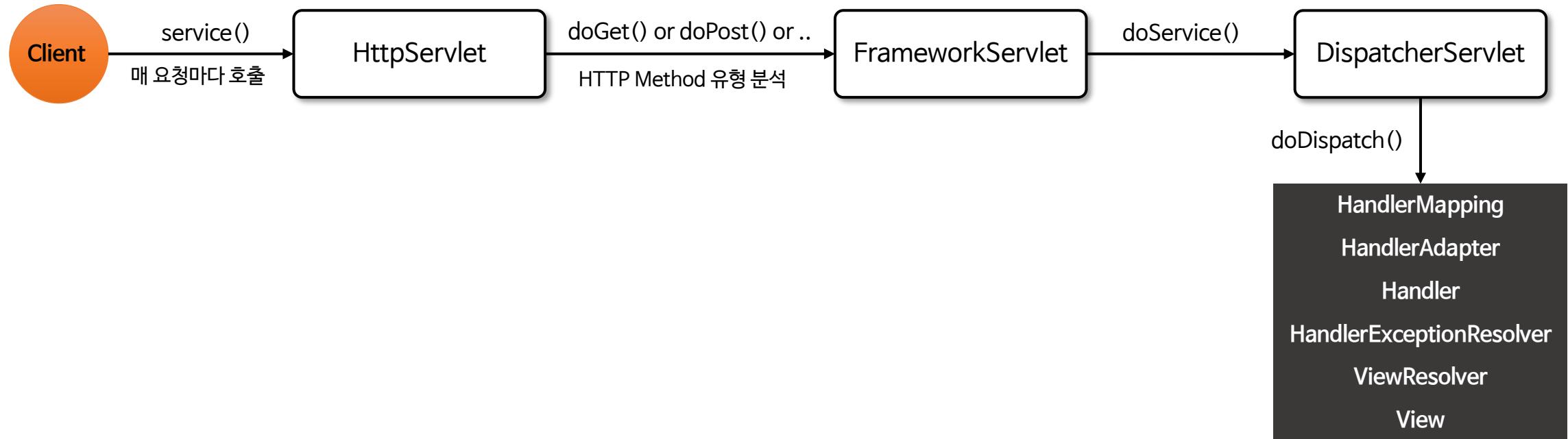
✓ DispatcherServlet#init()

- 요청이 시작되면 DispatcherServlet 의 `init()` 메서드가 호출되며 최초 한번 실행된다
- WebApplicationContext 를 생성 및 초기화하며 HandlerMapping, HandlerAdapter, ViewResolver 등의 필수 구성요소를 초기화하고 모든 요청을 처리할 준비를 완료한다



✓ DispatcherServlet#service()

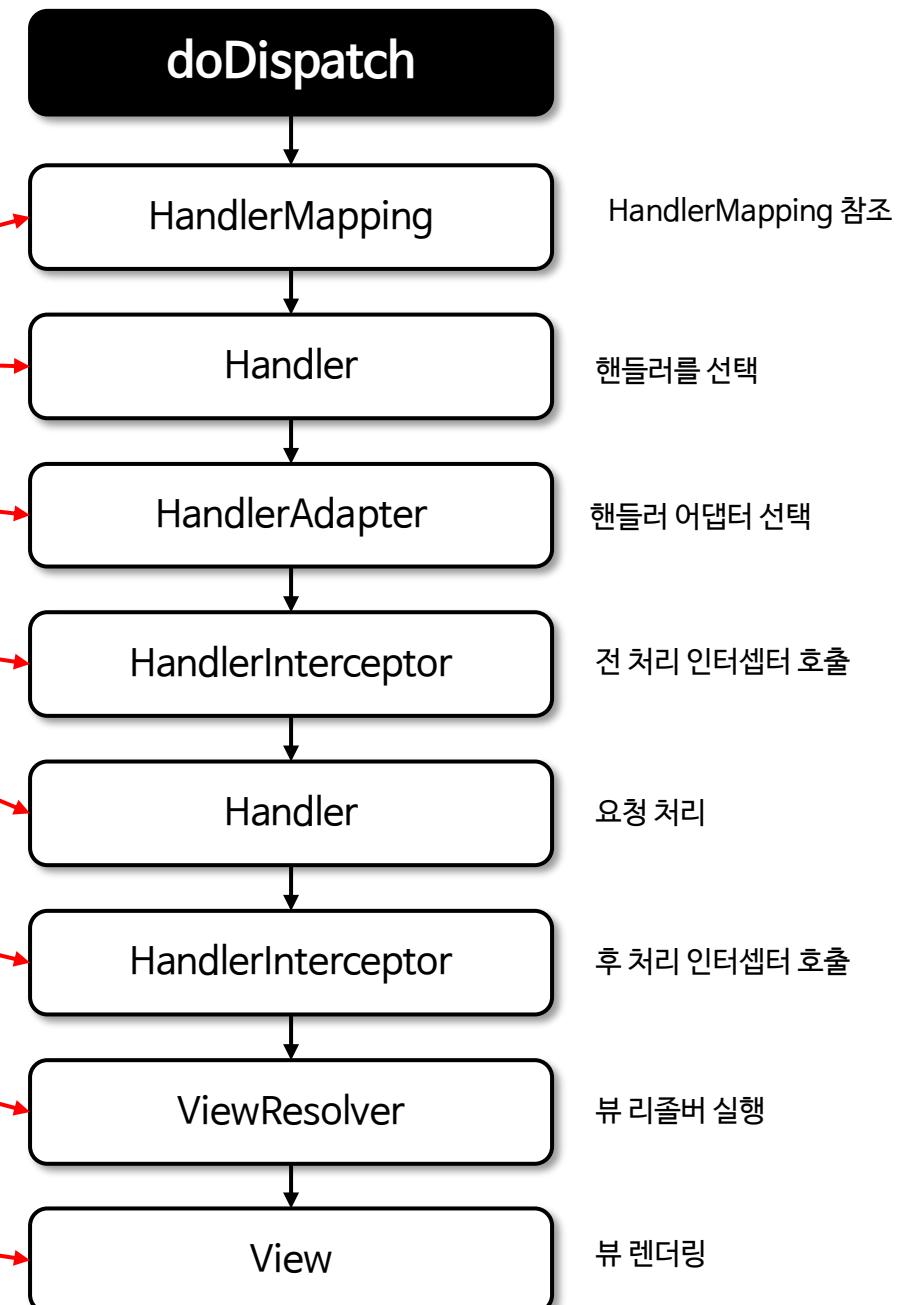
- 매 요청마다 실행되는 메서드로 HTTP 요청을 분석하여 적합한 핸들러(Controller)를 찾고 실행하는 역할을 한다
- 실행 결과를 기반으로 뷰(View)를 렌더링하여 클라이언트에게 응답을 반환한다



✓ doDispatch()

- 실제 핸들러로 요청을 디스패치하는 작업을 처리한다
- 모든 HTTP 메서드는 doDispatch() 메서드에 의해 처리가 이루어진다

```
protected void doDispatch(HttpServletRequest request, HttpServletResponse response) {  
    // 현재 요청을 처리할 핸들러 검색  
    HandlerExecutionChain mappedHandler = getHandler(request);  
    // 현재 요청에 대한 핸들러 어댑터를 결정  
    HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());  
    // 전 처리 인터셉터 수행  
    if (!mappedHandler.applyPreHandle(request, response)) return;  
    // 실제 핸들러 호출 후 ModelAndView 반환  
    ModelAndView mv = ha.handle(request, response, mappedHandler.getHandler());  
    // 후 처리 인터셉터 수행  
    mappedHandler.applyPostHandle(request, response, mv);  
    // 뷰 이름 확인  
    String viewName = mv.getViewName();  
    // 뷰 리졸버를 통해 뷰 선택  
    View view = resolveViewName(viewName, mv.getModelInternal(), locale, request);  
    // 뷰 렌더링  
    view.render(mv.getModelInternal(), request, response);  
}
```





스프링 웹 MVC 완전 정복

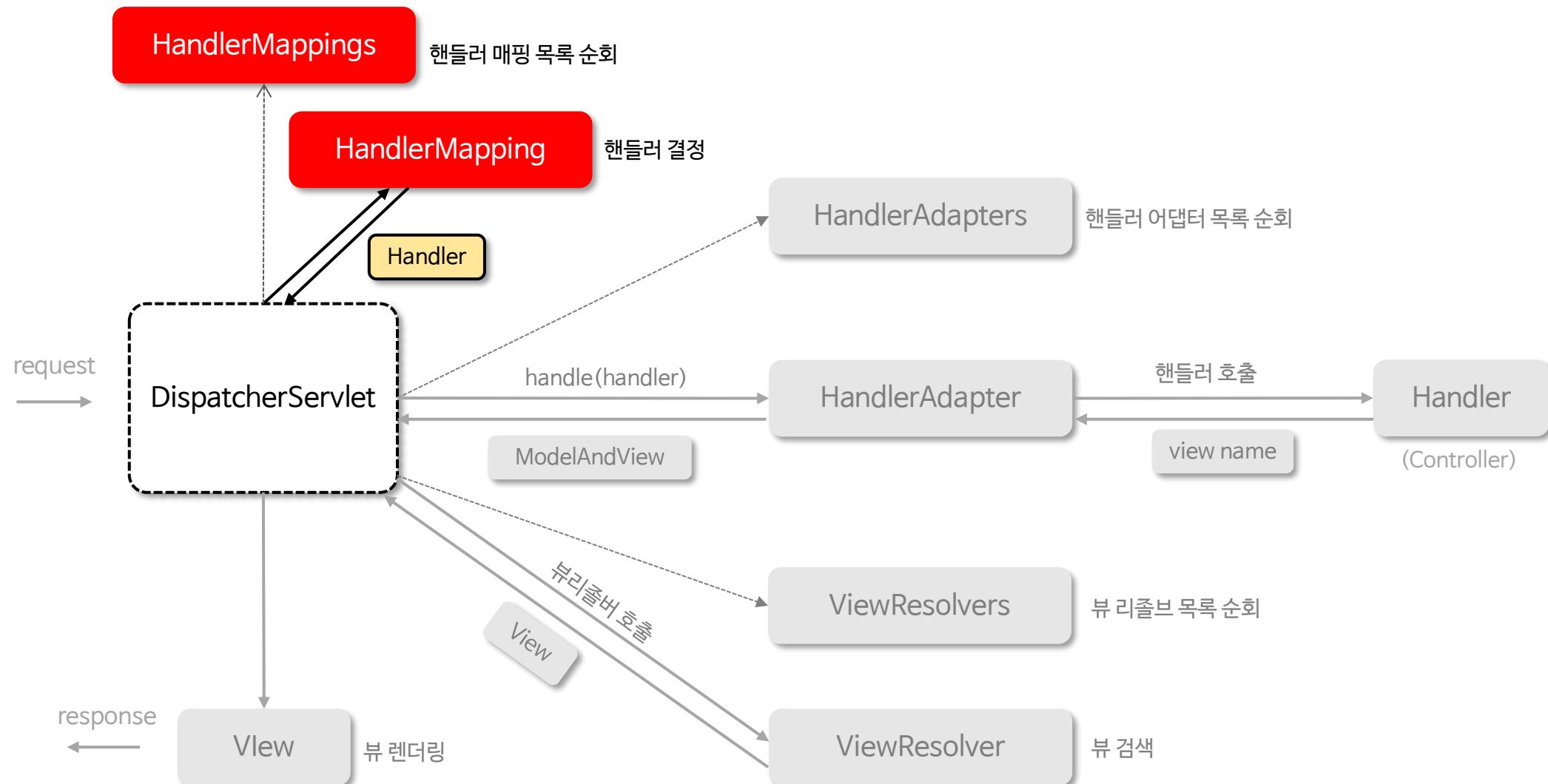
✓ 스프링 웹 MVC 활용

1. HandlerMapping
2. @RequestMapping
3. @RequestMapping 원리 이해
4. HandlerAdapter
5. Method Arguments
6. 바인딩 - DataBinder, @InitBinder
7. 타입 변환 (Type Conversion)
8. 검증 (Validation)
9. Return Values
10. View / ViewResolver

HandlerMapping

<https://github.com/onjsdnjs/spring-mvc-master/tree/HandlerMapping>

✓ 스프링 MVC 아키텍처



✓ 개요

- HandlerMapping 은 요청 URL과 이를 처리할 핸들러(Handler, 일반적으로 Controller)를 매핑하는 인터페이스이다
- 클라이언트 요청이 들어오면 DispatcherServlet 은 등록된 모든 HandlerMapping 구현체를 탐색하여 적합한 핸들러를 찾아 반환하고 이후 적절한 HandlerAdapter 를 통해 실행한다
- HandlerMapping 은 핸들러를 검색해서 찾는 역할만 할 뿐 핸들러를 실행하지는 않는다. 핸들러 실행은 HandlerAdapter 가 담당한다

✓ 구조

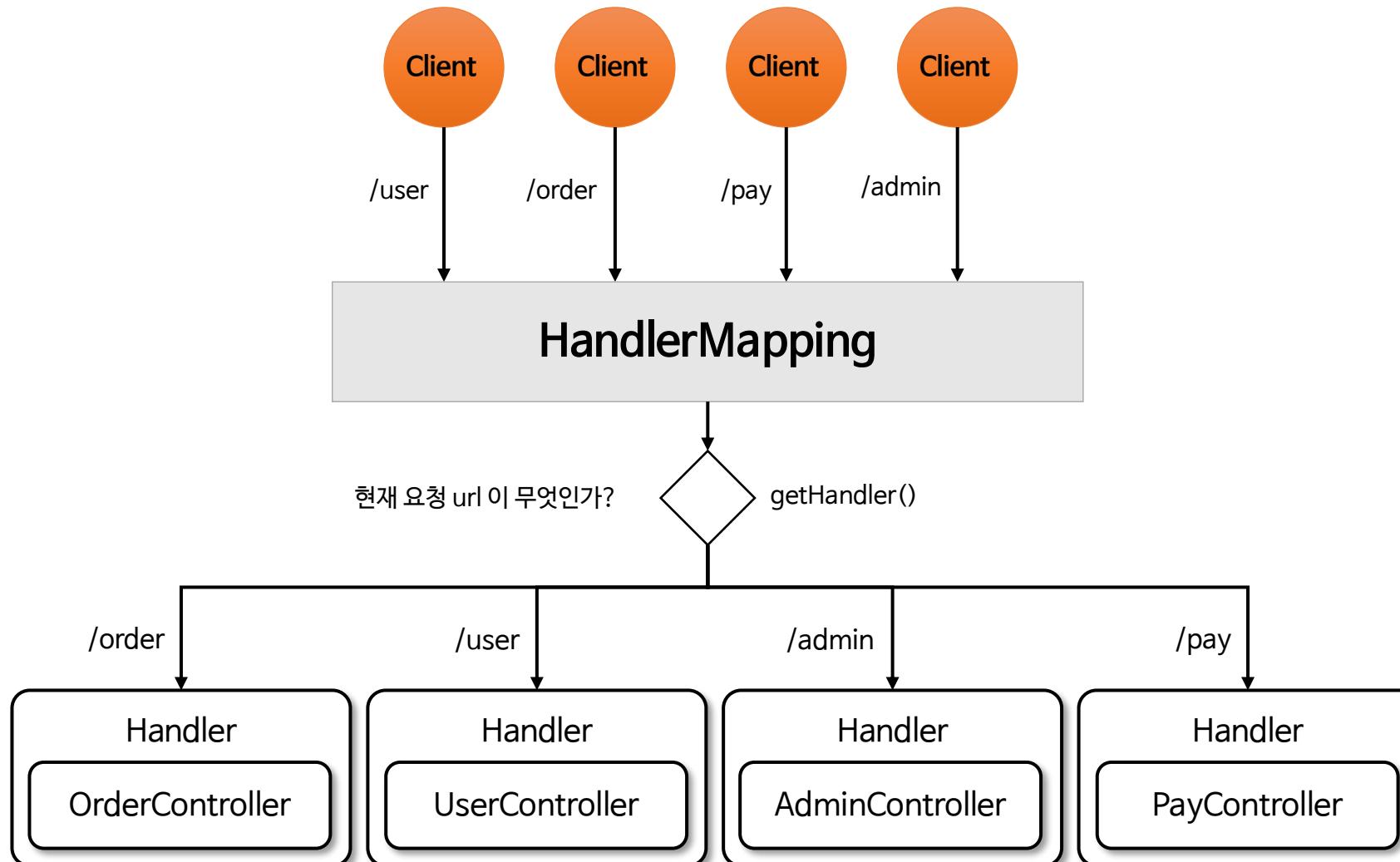
HandlerMapping

HandlerExecutionChain **getHandler**(HttpServletRequest request) throws Exception

- BeanNameUrlHandlerMapping
 - Bean 이름을 URL로 매핑합니다.
 - 현재는 잘 사용되지 않는 이전 방식이다
- RequestMappingHandlerMapping
 - 가장 우선순위가 높고 대부분 이 방식을 사용한다
 - @RequestMapping 또는 @GetMapping, @PostMapping 과 같은 애노테이션을 기반으로 URL과 핸들러를 매핑한다
- SimpleUrlHandlerMapping
 - 명시적으로 URL 패턴을 핸들러와 매핑하는 방식으로서 간단한 URL 매핑에 사용된다

✓ HandlerMapping 설계 의도

- DispatcherServlet 은 수 많은 요청들을 핸들러를 통해서 처리하는데 현재 요청이 어떤 핸들러에 의해 처리될 것인지 결정해서 DispatcherServlet 에게 알려 주어야 한다
- HandlerMapping 은 클라이언트의 요청 Url 정보를 분석해서 해당 Url 과 매핑이 되어 있는 핸들러를 결정한다



✓ Handler 구현 방식

- **@Controller, @RestController**

- `@Controller` 는 Spring MVC 에서 가장 널리 사용되는 요청 처리 방식으로서 클래스에 `@Controller` 를 붙이고 메서드에 `@RequestMapping` 과 같은 어노테이션을 사용하여 요청을 처리한다

```
@Controller  
public class ExampleController {  
    @RequestMapping(value = "/hello", method = RequestMethod.GET)  
    public String hello(Model model) { . . . }  
}
```

- **Controller 인터페이스**

- Spring 2.5 이전에 사용되던 요청 처리 방식으로서 Controller 인터페이스를 구현하여 요청을 처리한다

```
public class ExampleController implements Controller {  
    @Override  
    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) throws Exception { ... }  
}
```

- **HttpRequestHandler**

- `HttpRequestHandler` 인터페이스를 구현하여 요청을 처리하는 방식으로서 Spring의 가장 저수준 API 중 하나로 서블릿에 가까운 형태로 동작한다

```
public class ExampleHttpRequestHandler implements HttpRequestHandler {  
    @Override  
    public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws IOException { ... }  
}
```

✓ BeanNameUrlHandlerMapping

```
@Component("/beanHandler") // Bean 이름이 요청 URL과 매핑됨
public class BeanNameHandler implements Controller {
    @Override
    public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse res) throws Exception {
        response.getWriter().write("BeanNameUrlHandlerMapping!");
        return null;
    }
}
```

http://localhost:8080/beanHandler

Spring 컨테이너에 등록된 빈 이름을 기반으로 URL 을 자동 매핑하여 빈 이름이 URL 경로와 일치하면 해당 요청을 처리한다

✓ SimpleUrlHandlerMapping

```
public class MyHttpRequestHandler implements HttpRequestHandler {
    @Override
    public void handleRequest(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        response.setContentType("text/plain");
        response.getWriter().write("SimpleUrlHandlerMapping!");
    }
}
```

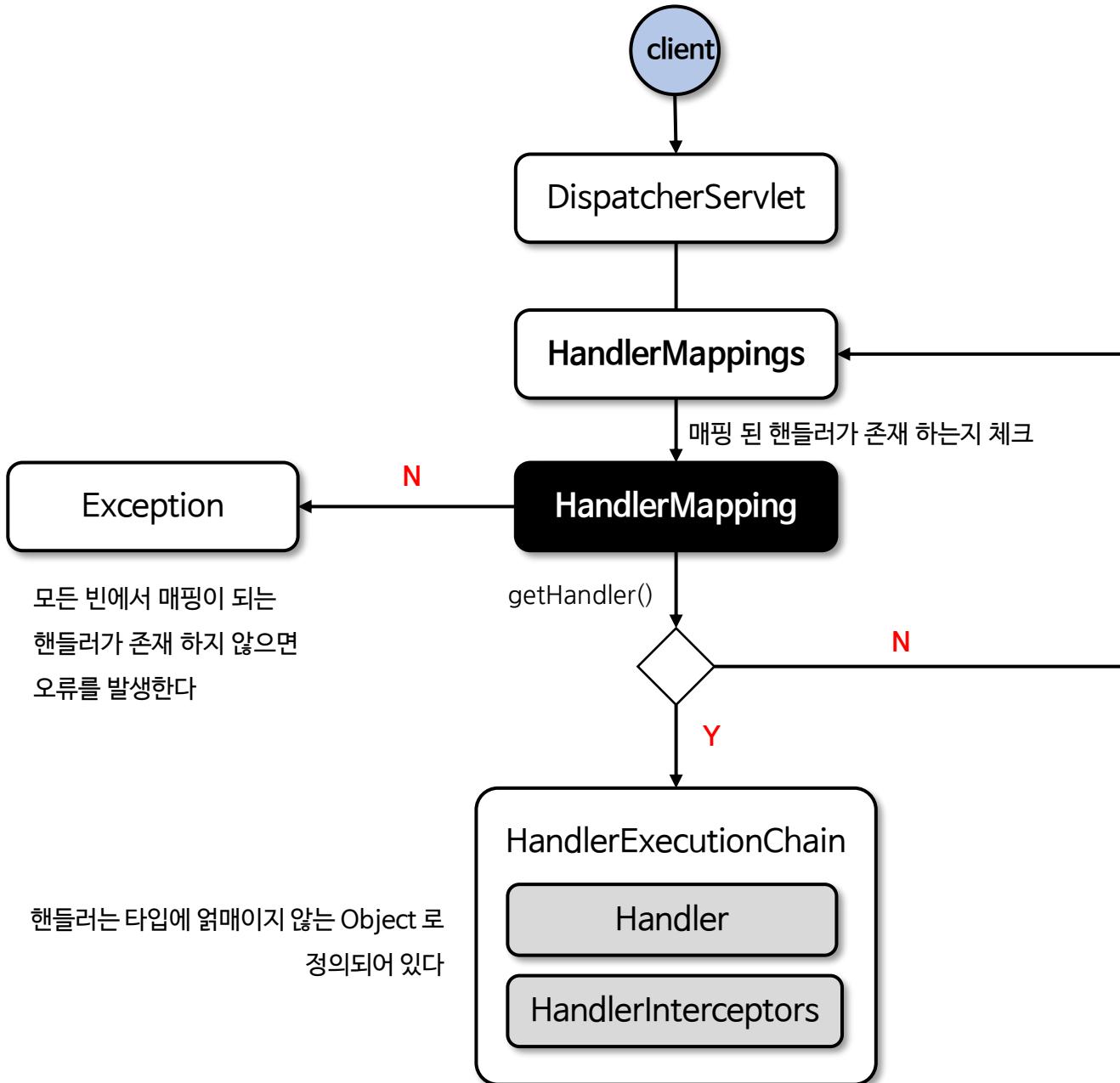
http://localhost:8080/simpleUrl

설정 파일이나 코드에서 명시적으로 URL 패턴과 핸들러를 매핑하여 특정 URL 요청을 지정된 핸들러로 처리한다

```
@Bean
public SimpleUrlHandlerMapping simpleUrlHandlerMapping() {
    SimpleUrlHandlerMapping mapping = new SimpleUrlHandlerMapping();
    Properties urlMappings = new Properties();
    urlMappings.setProperty("/simpleUrl", "myHttpRequestHandler");
    mapping.setMappings(urlMappings);
    mapping.setOrder(1);
    return mapping;
}

@Bean
public MyHttpRequestHandler myHttpRequestHandler() {
    return new MyHttpRequestHandler();
}
```

✓ 흐름도



핸들러가 없으면 null 반환하고 재 시도

```
if (this.handlerMappings != null) {  
    for (HandlerMapping mapping : this.handlerMappings) {  
        HandlerExecutionChain handler = mapping.getHandler(request);  
        if (handler != null) {  
            return handler;  
        }  
    }  
}  
return null;
```

```
⌚ this.handlerMappings = {ArrayList@6732} size = 6 ... View  
> ⌚ 0 = {RouterFunctionMapping@8143}  
> ⌚ 1 = {RequestMappingHandlerMapping@8147}  
> ⌚ 2 = {WelcomePageHandlerMapping@8148}  
> ⌚ 3 = {BeanNameUrlHandlerMapping@8149}  
> ⌚ 4 = {WelcomePageNotAcceptableHandlerMapping@8150}  
> ⌚ 5 = {SimpleUrlHandlerMapping@8151}
```

@RequestMapping

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestMapping>

✓ 개요

- `@RequestMapping` 은 특정 URL 경로에 대한 요청을 처리할 메서드를 매핑하는 데 사용되는 어노테이션이다
- `@RequestMapping` 은 `@Controller` 및 `@RestController` 로 선언된 클래스에서 사용되며 내부적으로 `RequestMappingHandlerMapping` 클래스가 처리하고 있다

✓ 기본 구조

@ RequestMapping	
Ⓜ️ ⌂ method()	RequestMethod[]
Ⓜ️ ⌂ path()	String[]
Ⓜ️ ⌂ params()	String[]
Ⓜ️ ⌂ headers()	String[]
Ⓜ️ ⌂ consumes()	String[]
Ⓜ️ ⌂ value()	String[]
Ⓜ️ ⌂ name()	String
Ⓜ️ ⌂ produces()	String[]

```
@Controller  
public class HandlerMappingController {  
    @RequestMapping("/home")  
    public String home (Model model) {  
        model.addAttribute("message", "Hello")  
        return "home";  
    }  
}
```

```
@RestController  
public class HandlerMappingController {  
    @RequestMapping("/home")  
    public String home (HttpServletRequest req) {  
        model.addAttribute("message", "Hello")  
        return "home";  
    }  
}
```

- `@Controller` 및 `@RestController` 를 선언하면 스프링이 자동으로 빈으로 등록한다

```
...  
@Component // 컴포넌트 스캔 대상  
public @interface Controller {...}
```

```
@Controller  
@ResponseBody // HTTP Body 에 응답을 기록한다  
public @interface RestController {...}
```

- 메서드 위에 `@RequestMapping` 을 정의하고 요청 경로를 지정한다
- `@Controller`에서 String 타입으로 반환하면 반환 값으로 된 페이지(HTML, JSP)를 검색하고 렌더링한다
- `@RestController`에서 String 타입으로 반환하면 반환 값이 HTTP 본문에 기록되고 화면에 출력된다

✓ URL 매팅 – 1

```
@RestController  
public class HomeController {  
  
    @RequestMapping("/profile")  
    public String profile () {  
        return "This is the profile page";  
    }  
  
    @RequestMapping({" /user/**", "/mypage"})  
    public String multi() {  
        return "This is the profile and about page";  
    }  
}
```

- 경로 매팅 URI 는 예를 들어 "/profile"과 같은 형태를 가진다
- Ant 스타일의 경로 패턴("/user/**") 지원한다
- 다중 설정({" /user/**", "/mypage"})도 가능하다

✓ URL 매핑 - 2

```
@RestController  
@RequestMapping("/profile") // 탑 수준 매핑  
public class ProfileController {  
  
    // "/profile/view"로 매핑  
    @RequestMapping("/view") // 메서드 수준 매핑  
    public String viewProfile() {  
        return "Viewing Profile";  
    }  
    // "/profile/edit"로 매핑  
    @RequestMapping("/edit") // 메서드 수준 매핑  
    public String editProfile() {  
        return "Editing Profile";  
    }  
}
```

- 탑 수준과 메서드 수준 모두 지원되며 탑 수준에서 사용되는 경우 모든 메서드 수준 매핑은 탑 수준 매핑을 상속한다
- 즉 탑 수준에서 경로를 정의하면 모든 메서드는 해당 경로를 포함한 URI를 가지게 된다

✓ URL 매팅 – 3

```
@RestController  
@RequestMapping("/{profile_path}") // 플레이스홀더 사용  
public class ProfileController {  
  
    @RequestMapping("/view")  
    public String viewProfile() {  
        return "Viewing Profile";  
    }  
  
    @RequestMapping("/edit")  
    public String editProfile() {  
        return "Editing Profile";  
    }  
}
```

application.properties profile_path=profile

- 매팅 URI는 플레이스홀더(예: "/\${profile_path}")를 포함할 수 있다
- profile_path 값을 profile로 설정했으므로 최종 경로는 다음과 같다
 - /profile/view → viewProfile() 메서드 실행
 - /profile/edit → editProfile() 메서드 실행

✓ HTTP 메서드 매핑

```
@RestController
@RequestMapping(value = "/api")
public class UserController {

    @RequestMapping(value = "/order")
    public String getOrder() {
        return "Order data";
    }

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public String getUser() {
        return "User data";
    }

    @RequestMapping(value = "/user", method = RequestMethod.POST)
    public String createUser(@RequestBody User user) {
        return "User created";
    }
}
```

- GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE 메서드를 설정할 수 있다
- 타입 수준에서 사용되는 경우 모든 메서드 수준 매핑은 해당 HTTP 메서드 설정을 상속받는다
- 타입 수준과 메서드 수준이 함께 설정되었을 경우 메서드 수준의 HTTP 메서드 설정이 타입 수준의 설정을 덮어쓴다. 즉, 메서드 수준의 설정이 우선 적용된다

✓ param 매핑

```
@RestController  
public class OrderController {  
  
    @RequestMapping(value = "/order", params = "type=pizza")  
    public String order1() {  
        return "pizza";  
    }  
  
    @RequestMapping(value = "/order", params = "type= chicken ")  
    public String order2() {  
        return "chicken";  
    }  
}
```

} http://localhost:8080/order?type=pizza
http://localhost:8080/order?type=chicken

type 파라미터 값에 따라 다른 메서드가 호출된다

- 표현식은 != 연산자를 사용하여 부정할 수 있다(예: "myParam!=myValue")
- "myParam" 표현식도 지원되며 요청에 존재해야 하지만 어떤 값이라도 가질 수 있다
- "!myParam" 표현식은 해당 매개변수가 요청에 포함되지 않아야 함을 나타낸다

✓ headers 매핑

```
@RestController
@RequestMapping("/api")
public class ContentTypeController {
    // Content-Type이 text/* 형식인 요청만 매핑
    @RequestMapping(value = "/text", method = RequestMethod.POST, headers = "content-type=text/*") // text/html, text/plain ...
    public String handleText() {
        return "text/* Content-Type";
    }
    // Content-Type이 application/json인 요청만 매핑
    @RequestMapping(value = "/json", method = RequestMethod.POST, headers = "content-type=application/json")
    // @RequestMapping(value = "/json", method = RequestMethod.POST, headers = "content-type!=application/json")
    public String handleJson() {
        return "application/json Content-Type";
    }
    // 헤더가 지정된 값을 가질 경우에만 요청이 매핑
    @RequestMapping(value = "/header", method = RequestMethod.GET, headers = "My-Header=myValue")
    public String handleNoContentType() {
        return "My-Header";
    }
}
```

- 타입 수준과 메서드 수준 모두에서 지원되며 타입 수준에서 사용되는 경우 모든 메서드 수준 매핑은 이 헤더 설정을 상속받는다

✓ produces 와 consumes 설정

- consumes

```
@RequestMapping(value = "/report", method = RequestMethod.POST, consumes = "application/json")
```

```
public void saveReport(@RequestBody Report report) {
```

```
    // JSON 형식의 Report 데이터를 처리하는 로직
```

```
}
```

```
consumes = "text/plain"
```

```
consumes = {"text/plain", "application/*"}
```

```
consumes = MediaType.TEXT_PLAIN_VALUE
```

- 클라이언트가 서버로 전송하는 데이터의 형식을 제한한다
- 클라이언트에서 Content-Type: application/json 헤더를 지정하고 스프링에서 consumes = "application/json" 으로 설정하면 해당 메서드는 JSON 형식의 데이터를 처리할 수 있다

- produces

```
@RequestMapping(value = "/report", method = RequestMethod.GET, produces = "application/json")
```

```
public Report getReport() {
```

```
    return new Report("title");
```

```
}
```

```
produces = "text/plain"
```

```
produces = {"text/plain", "application/*"}
```

```
produces = MediaType.TEXT_PLAIN_VALUE
```

```
produces = "text/plain; charset=UTF-8"
```

- 클라이언트가 서버에 요청을 보낼 때 서버가 어떤 형식의 데이터를 반환할지를 지정한다

- 클라이언트가 Accept: application/json 헤더를 지정하고 스프링에서 produces = "application/json" 으로 설정하면 해당 메서드는 JSON 형식의 응답을 반환한다

✓ HTTP Content-Type / Accept 헤더 이해

```
POST /report HTTP/1.1
```

```
Host: example.com
```

```
Accept: application/json
```

```
Content-Type: application/json
```

1. consumes 속성과 Content-Type 헤더

- **일치할 경우:** 서버의 consumes 속성에 정의된 미디어 타입이 클라이언트의 Content-Type 헤더와 일치하는 경우 서버는 해당 요청을 정상적으로 처리한다. consumes 속성은 서버가 수락할 수 있는 요청의 미디어 타입을 지정하며, 클라이언트는 그 형식으로 데이터를 서버에 전송해야 한다
- **일치하지 않을 경우:** 만약 클라이언트가 전송한 요청의 Content-Type 헤더가 서버의 consumes 속성에 정의된 형식과 일치하지 않으면 서버는 HTTP 415 (Unsupported Media Type) 상태 코드를 반환할 수 있으며 이 상태 코드는 서버가 클라이언트가 보낸 데이터 형식을 처리할 수 없음을 나타낸다
- **consumes 속성 미지정:** 만약 서버에서 consumes 속성을 지정하지 않았다면, 서버는 기본적으로 요청을 처리할 수 있는 미디어 타입에 대해 특별한 제한을 두지 않지만 클라이언트의 Content-Type 헤더와 서버의 처리 능력이 일치하지 않으면, 요청이 처리되지 않을 수 있다

2. produces 속성과 Accept 헤더

- **일치할 경우:** 서버의 produces 속성에 정의된 미디어 타입이 클라이언트의 Accept 헤더와 일치하는 경우, 서버는 해당 미디어 타입으로 응답을 생성한다
- **일치하지 않을 경우:** 만약 produces 속성에 정의된 미디어 타입이 클라이언트의 Accept 헤더와 일치하지 않으면, 서버는 HTTP 406 (Not Acceptable) 상태 코드를 반환할 수 있으며 이 상태 코드는 서버가 클라이언트가 요청한 형식으로 응답을 생성할 수 없음을 나타낸다
- **produces 속성 미지정:** 만약 서버에서 produces 속성을 지정하지 않았다면, 서버는 클라이언트의 Accept 헤더와 일치하는 미디어 타입으로 응답을 생성하려고 시도하며 이 경우 클라이언트가 요청한 형식과 서버가 반환할 수 있는 형식이 일치하면 그 형식으로 응답을 생성한다

✓ @RequestMapping 축약 어노테이션

- **@GetMapping**

- GET 요청을 처리하는 메서드 맵핑

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {…}
```

- **@PostMapping**

- POST 요청을 처리하는 메서드 맵핑

- **@PutMapping**

- PUT 요청을 처리하는 메서드 맵핑

- **@DeleteMapping**

- DELETE 요청을 처리하는 메서드 맵핑

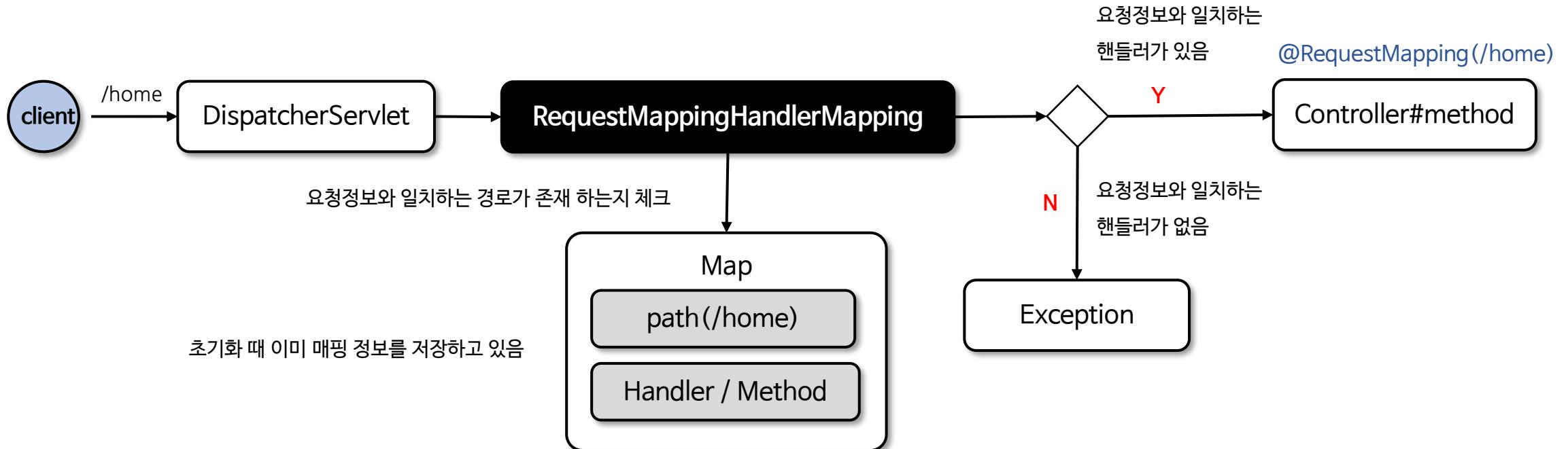
```
@GetMapping(value = "/order")
public String order() {
    return " pizza ";
}
```



```
@RequestMapping(value = "/order", method = RequestMethod.GET)
public String order() {
    return " pizza ";
}
```

✓ RequestMappingHandlerMapping

- RequestMappingHandlerMapping 은 @RequestMapping 이 선언된 핸들러와 URL 경로를 매핑하고 검색하는 기능을 제공한다
- 1) RequestMappingHandlerMapping 은 스프링 부트의 초기화 과정에서 @RequestMapping 이 선언된 모든 핸들러를 검사하고 검사 결과 해당 핸들러와 URL 경로를 매핑하여 저장소에 저장한다
 - 2) 클라이언트 요청이 들어오면 매핑 저장소로부터 URL 패턴과 일치하는 핸들러를 검색하고 적합한 핸들러를 찾아 반환한다



@RequestMapping 원리 이해

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestMapping-원리이해>

✓ 개요

- RequestMappingHandlerMapping 클래스의 실행 구조 및 클라이언트의 요청을 어떻게 맵핑하고 있는지 내부 클래스들을 분석함으로서 스프링 MVC 의 내부 흐름을 더 깊이 이해하고 개발적 관점에서의 인사이트를 넓혀 갈수 있다

✓ 핵심 클래스 이해

RequestMappingHandlerMapping

- @RequestMapping 이 선언된 모든 핸들러를 검사하고 해당 핸들러와 URL 경로를 맵핑하여 저장소에 저장한다
- 요청이 들어오면 맵핑 저장소로부터 URL 패턴과 일치하는 핸들러를 검색하고 적합한 핸들러를 찾아 반환한다

HandlerMethod

- 핸들러 객체로서 최종 호출할 컨트롤러와 메서드 정보를 포함하고 있다
- 메서드의 매개변수, 반환 값, 메서드에 정의된 어노테이션 등에 쉽게 접근할 수 있는 기능을 제공한다

RequestMappingInfo

- @RequestMapping 에 선언된 여러 속성들의 값을 추출해서 가지고 있는 객체

MappingRegistry

- 모든 HandlerMethod 및 맵핑된 경로를 저장하고 관리하는 클래스로서 요청을 처리할 핸들러를 조회할 수 있다

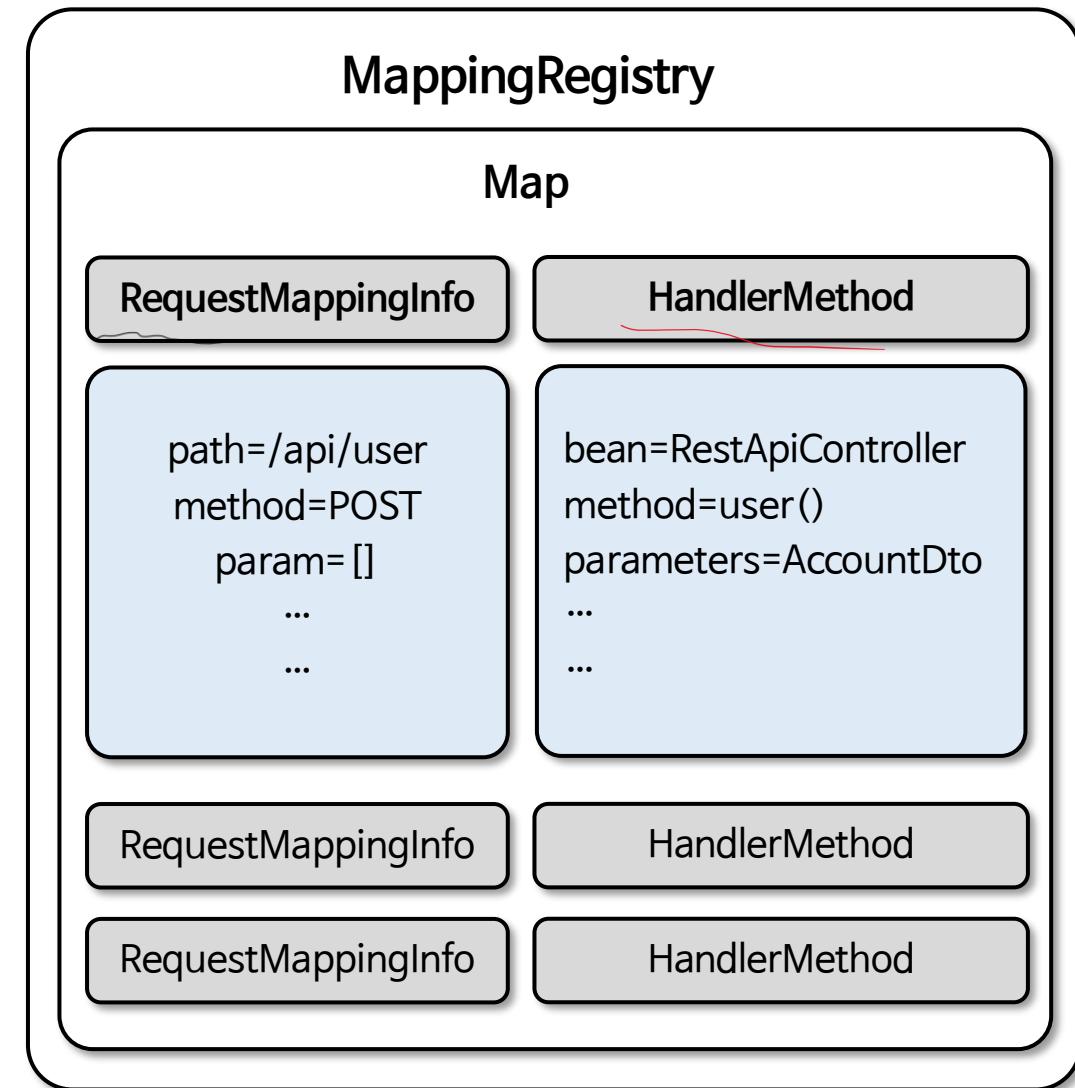
HandlerExecutionChain

- HandlerMethod 와 HandlerInterceptor 를 포함하고 있으며 HandlerAdapter 에게 전달된다

✓ 초기화 매핑 구조

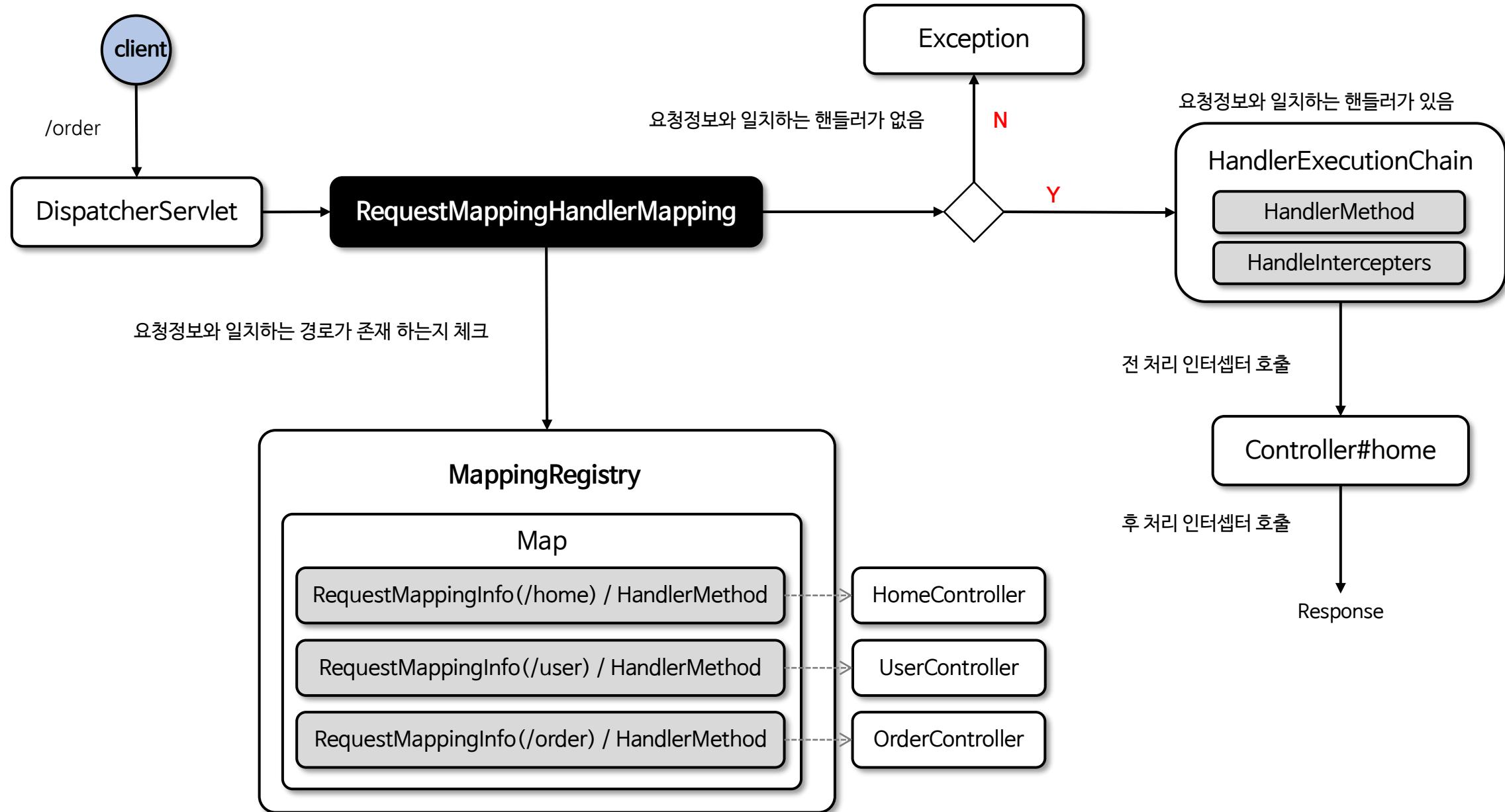
```
@RestController  
@RequestMapping("/api")  
  
public class RestApiController {  
    @PostMapping("/user")  
    public AccountDto user(@RequestBody AccountDto accountDto) {  
        return accountDto;  
    }  
  
    @PostMapping("/order")  
    public ResponseEntity<AccountDto> order(@RequestBody ProductDto productDto) {  
        return ResponseEntity.ok(new AccountDto("onjsdnjs", "1111"));  
    }  
}
```

초기화 시 `@RequestMapping` 이 선언된 클래스와 메서드를 추출해서 Handler 객체를 만들고 Url 경로와 매핑이 이루어짐



RequestMappingInfo에는 `@RequestMapping` 정보들이 저장 되어 있으며
HandlerMethod에는 컨트롤러와 메서드 정보가 들어 있다

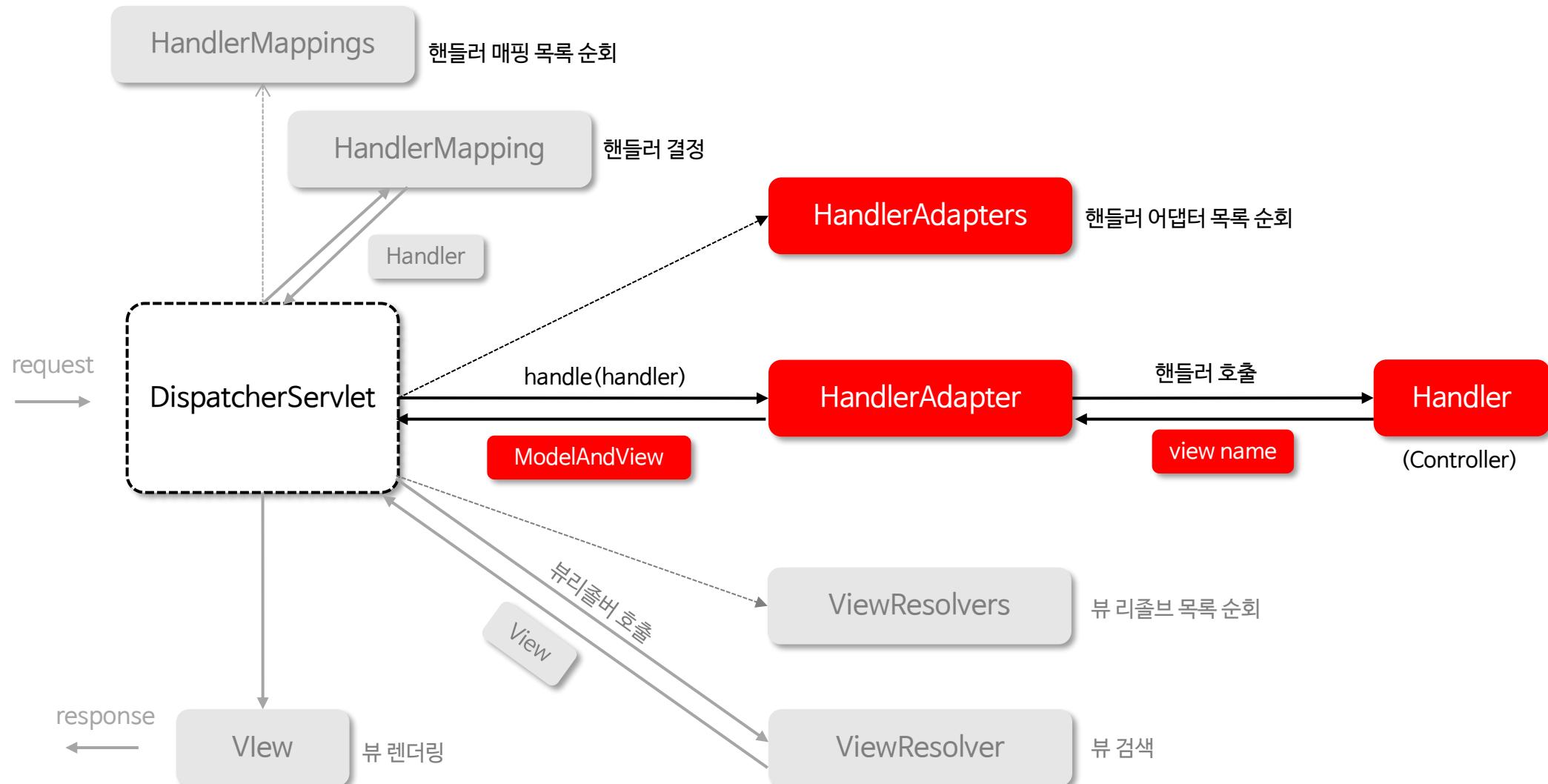
✓ 요청 흐름도



HandlerAdapter

<https://github.com/onjsdnjs/spring-mvc-master/tree/HandlerAdapter>

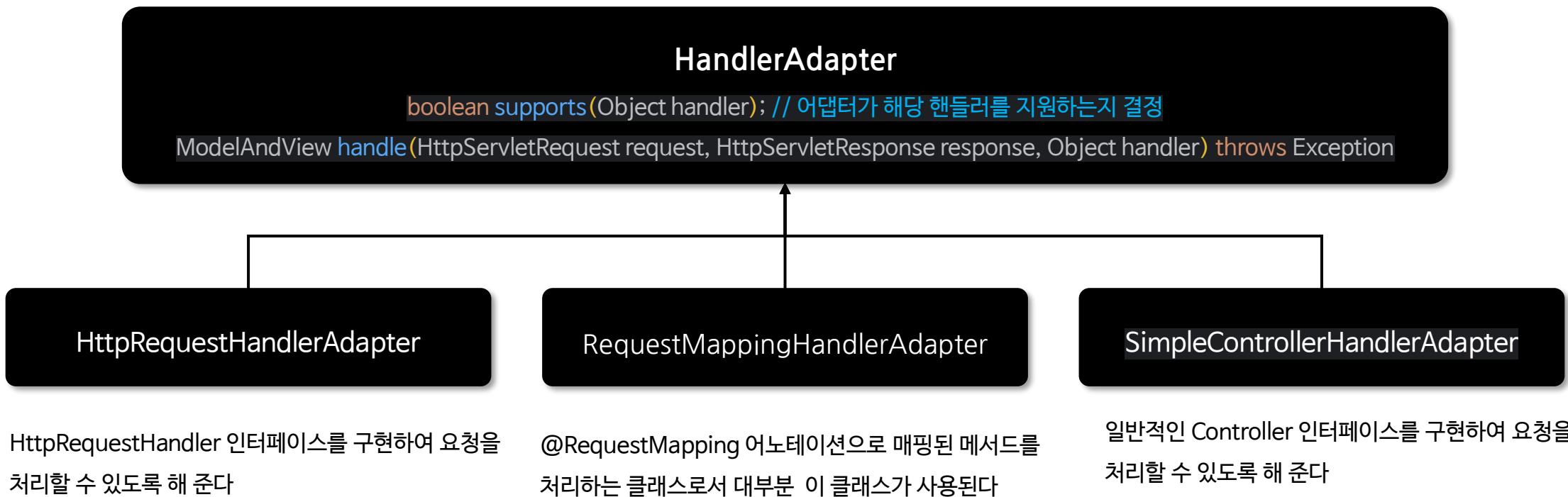
✓ 스프링 MVC 아키텍처



✓ 개요

- HandlerAdapter는 스프링 MVC에서 핸들러(Handler)를 호출하는 역할을 하는 인터페이스이다
- HandlerAdapter는 다양한 타입의 핸들러들이 일관된 방식으로 호출 될 수 있도록 해 주며 핸들러가 다양한 타입으로 정의되더라도 그에 맞는 호출 방식을 제공해 준다
- 요청이 들어왔을 때 어떤 핸들러가 해당 요청을 처리할지 결정하는 것이 HandlerMapping 이라면 HandlerAdapter는 결정된 핸들러를 호출하여 실행하는 역할을 한다

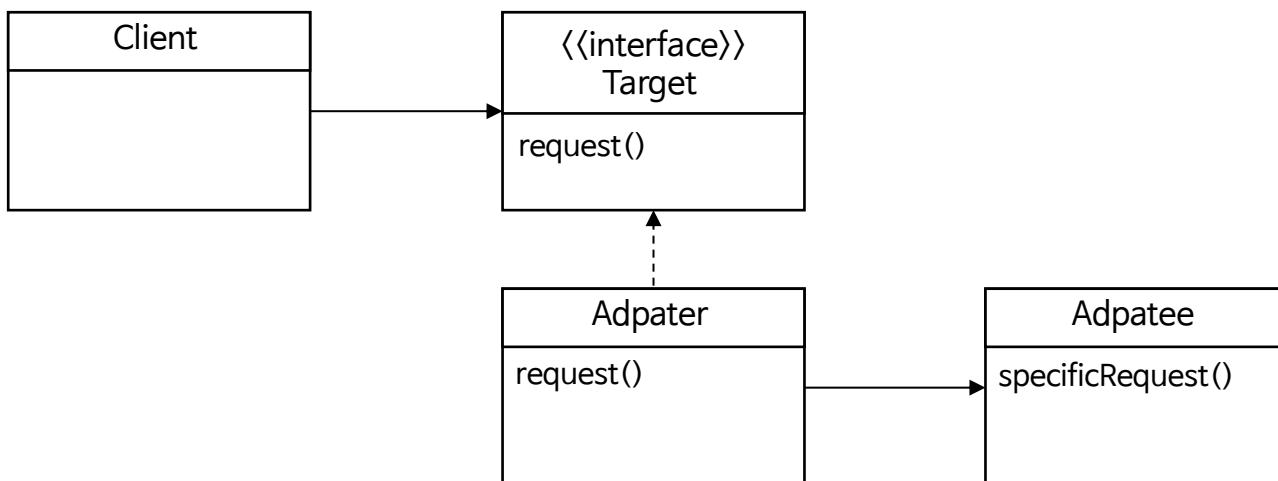
✓ 구조



✓ Adapter 패턴

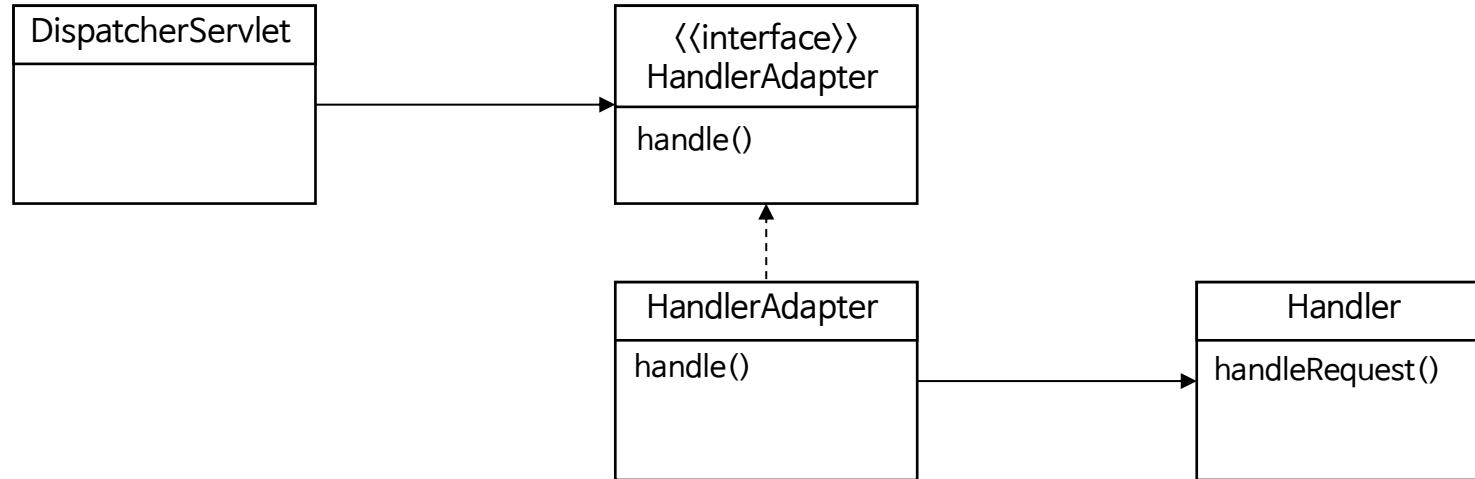
- 어댑터 패턴은 서로 호환되지 않는 인터페이스를 가진 클래스들이 함께 작동할 수 있도록 중간에서 변환 역할을 하는 디자인 패턴으로서 기존 클래스의 코드를 변경하지 않고도 호환성을 제공할 수 있다

1. Client: 새로운 인터페이스를 사용하려는 주체
2. Target: 클라이언트가 사용하고자 하는 인터페이스
3. Adapter: 타깃 인터페이스를 구현하여 기존 클래스(Adapter)와 클라이언트를 연결
4. Adaptee: 기존 인터페이스를 가진 클래스



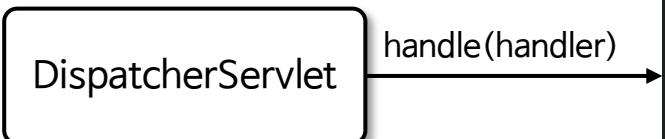
```
interface Target {  
    void request();  
}  
  
class Adaptee {  
    public void specificRequest() {  
        System.out.println("Adaptee: Specific request called.");  
    }  
}  
  
class Adapter implements Target {  
    private Adaptee adaptee;  
  
    public Adapter(Adaptee adaptee) {  
        this.adaptee = adaptee;  
    }  
  
    @Override  
    public void request() {  
        // Adaptee의 메서드를 호출하면서 타깃에 맞게 변환  
        adaptee.specificRequest();  
    }  
}  
  
public class AdapterPatternExample {  
    public static void main(String[] args) {  
        Adaptee adaptee = new Adaptee();  
        Target target = new Adapter(adaptee);  
  
        target.request(); // "Adaptee: Specific request called." 출력  
    }  
}
```

✓ HandlerAdapter 코드



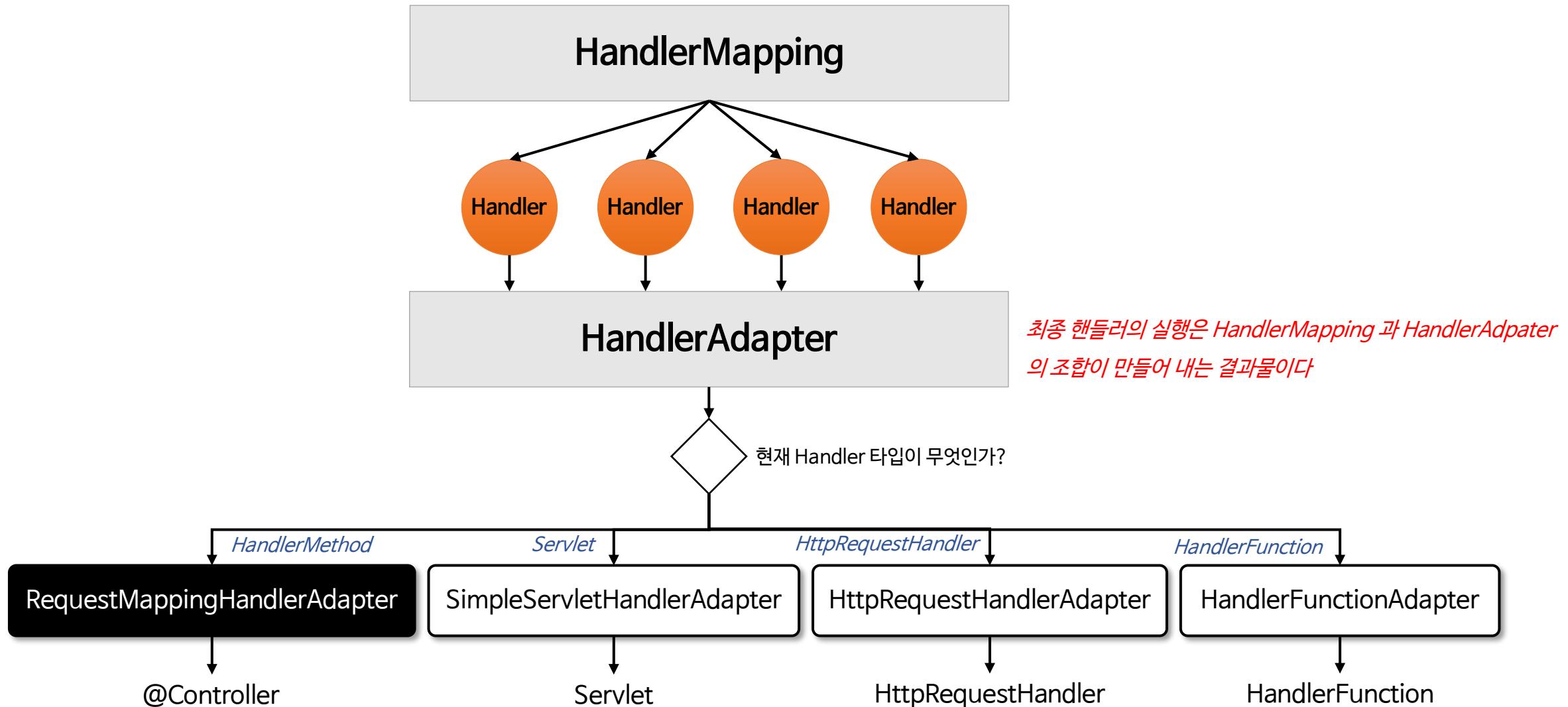
```
public class DefaultServletHttpRequestHandler implements HttpRequestHandler, ServletContextAware {  
    @Override  
    public void handleRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException {  
        // 비즈니스 로직 ...  
    }  
}
```

```
public class HttpRequestHandlerAdapter implements HandlerAdapter {  
    @Override  
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
        ((HttpRequestHandler) handler).handleRequest(request, response);  
    }  
}
```

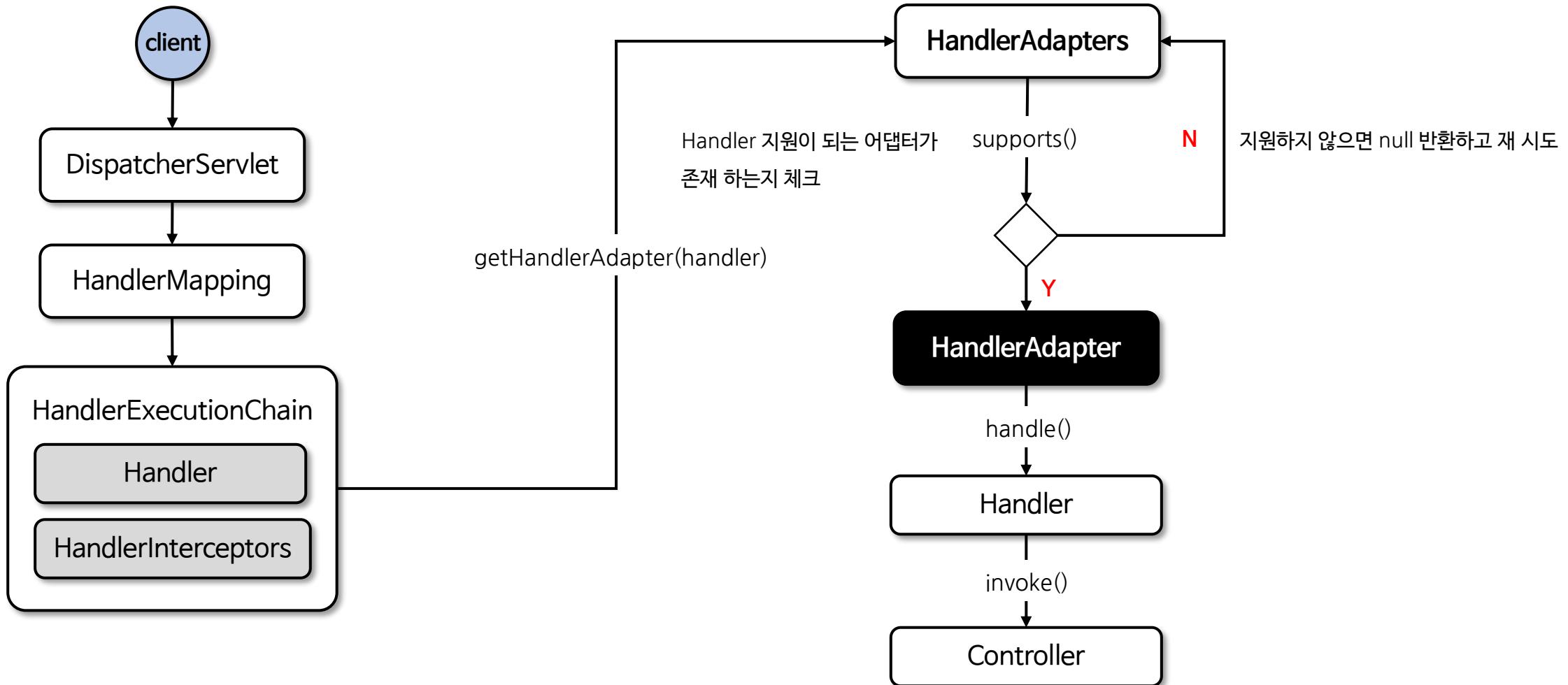


✓ HandlerAdapter 설계 의도

- HandlerAdapter는 HandlerMapping이 어떤 타입의 핸들러를 결정하더라도 타입과 상관없이 공통된 방식으로 핸들러를 호출할 수 있어야 한다
- HandlerAdapter는 HandlerMapping으로부터 전달받은 핸들러의 타입 정보를 분석해서 해당 핸들러를 지원하는지 여부를 판단하고 true이면 핸들러를 호출한다

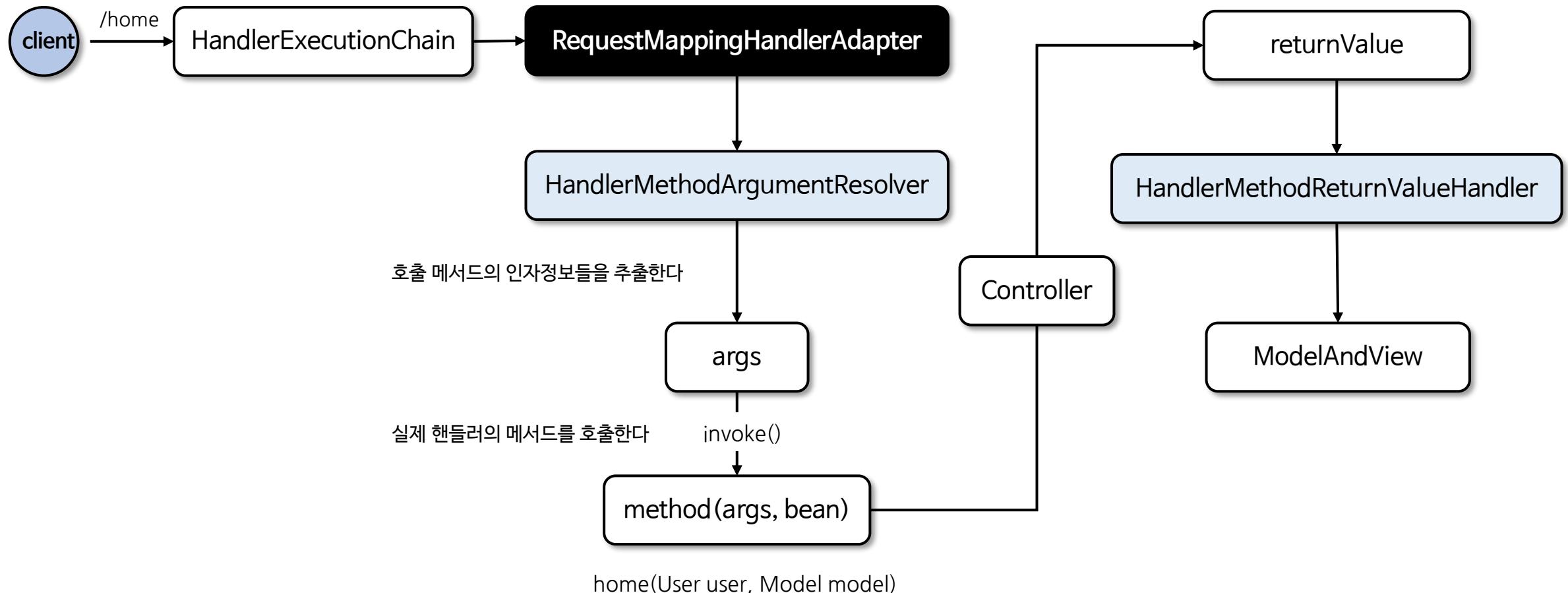


✓ **흐름도**



✓ RequestMappingHandlerAdapter

- RequestMappingHandlerAdapter 는 @RequestMapping 어노테이션이 적용된 컨트롤러 메서드를 처리하는 데 사용되는 어댑터이다
- 이 어댑터는 HandlerMethod 를 호출하고 메서드의 인자와 반환 값을 처리하는 역할을 하며 필요할 경우 사용자 정의 Argument Resolver 및 ReturnValueHandler 를 구현하여 설정 할 수 있다

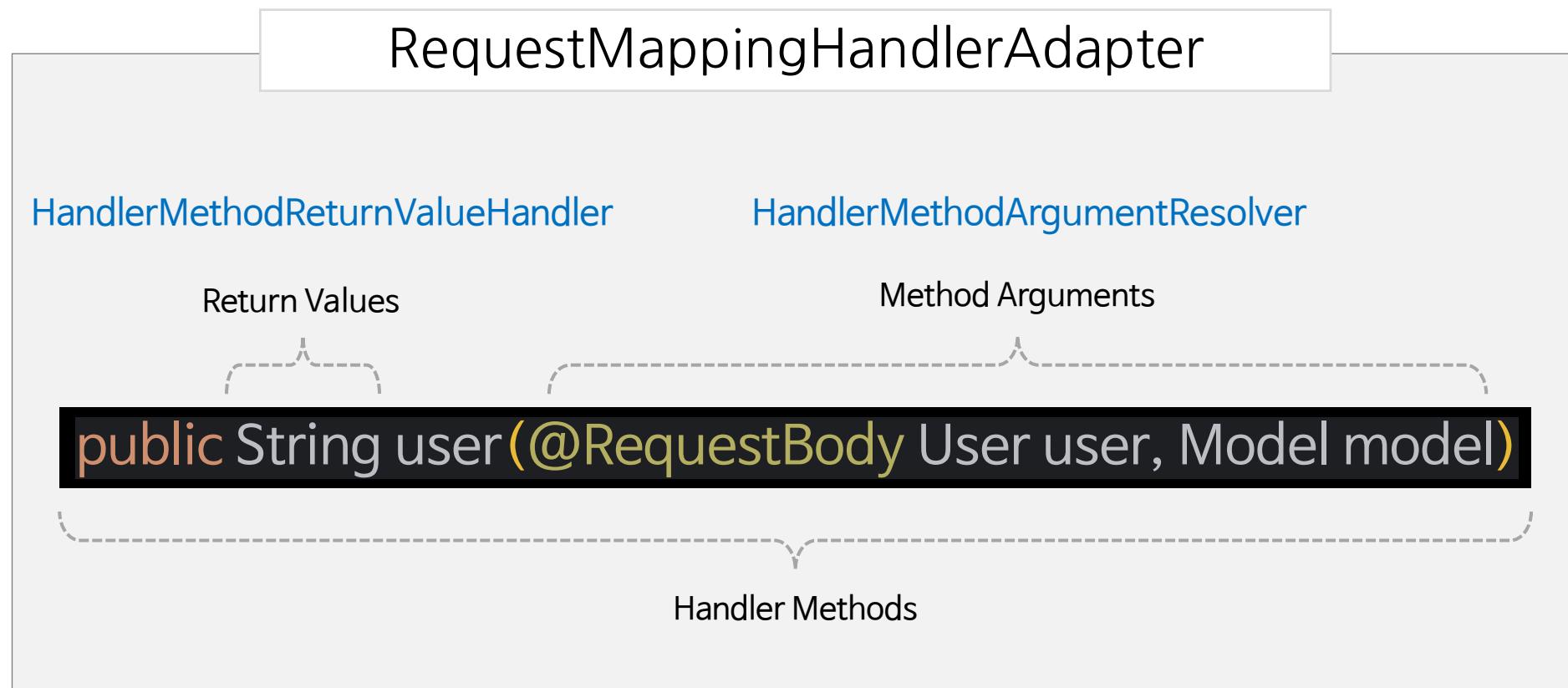


✓ Handler Methods

- @RequestMapping 이 선언된 핸들러 메서드는 메서드에 전달되는 인자와 반환 값을 다양한 타입과 방식으로 유연하게 설정할 수 있다
- 메서드 호출과 반환을 자유롭게 구현하기 위해서는 호출에 필요한 인자 정보와 반환에 필요한 타입 정보를 알 수 있어야 한다. 이것을 스프링에서는 Method Arguments & Return Values 로 나누어 API 를 제공하고 있다

Method Arguments & Return Values

@RequestParam
@RequestHeader
@CookieValue
@ModelAttribute
@SessionAttributes
@SessionAttribute
@RequestAttribute
Redirect Attributes
Flash Attributes
Multipart
@RequestBody
HttpEntity
@ResponseBody
ResponseEntity



HandlerMethodArgumentResolver

```
argumentResolvers = {ArrayList@8030} size = 30
> 0 = {RequestParamMethodArgumentResolver@8033}
> 1 = {RequestParamMapMethodArgumentResolver@8034}
> 2 = {PathVariableMethodArgumentResolver@8035}
> 3 = {PathVariableMapMethodArgumentResolver@8036}
> 4 = {MatrixVariableMethodArgumentResolver@8037}
> 5 = {MatrixVariableMapMethodArgumentResolver@8038}
> 6 = {ServletModelAttributeMethodProcessor@8039}
> 7 = {RequestResponseBodyMethodProcessor@8040}
> 8 = {RequestPartMethodArgumentResolver@8041}
> 9 = {RequestHeaderMethodArgumentResolver@8042}
> 10 = {RequestHeaderMapMethodArgumentResolver@8043}
> 11 = {ServletCookieValueMethodArgumentResolver@8044}
> 12 = {ExpressionValueMethodArgumentResolver@8045}
> 13 = {SessionAttributeMethodArgumentResolver@8046}
> 14 = {RequestAttributeMethodArgumentResolver@8047}
> 15 = {ServletRequestMethodArgumentResolver@8048}
> 16 = {ServletResponseMethodArgumentResolver@8049}
> 17 = {HttpEntityMethodProcessor@8050}
> 18 = {RedirectAttributesMethodArgumentResolver@8051}
> 19 = {ModelMethodProcessor@8052}
> 20 = {MapMethodProcessor@8053}
> 21 = {ErrorsMethodArgumentResolver@8054}
> 22 = {SessionStatusMethodArgumentResolver@8055}
> 23 = {UriComponentsBuilderMethodArgumentResolver@8056}
> 24 = {UserSessionArgumentResolver@8057}
> 25 = {CustomParamResolver@8058}
> 26 = {CustomModelResolver@8059}
> 27 = {PrincipalMethodArgumentResolver@8060}
> 28 = {RequestParamMethodArgumentResolver@8061}
> 29 = {ServletModelAttributeMethodProcessor@8062}
```



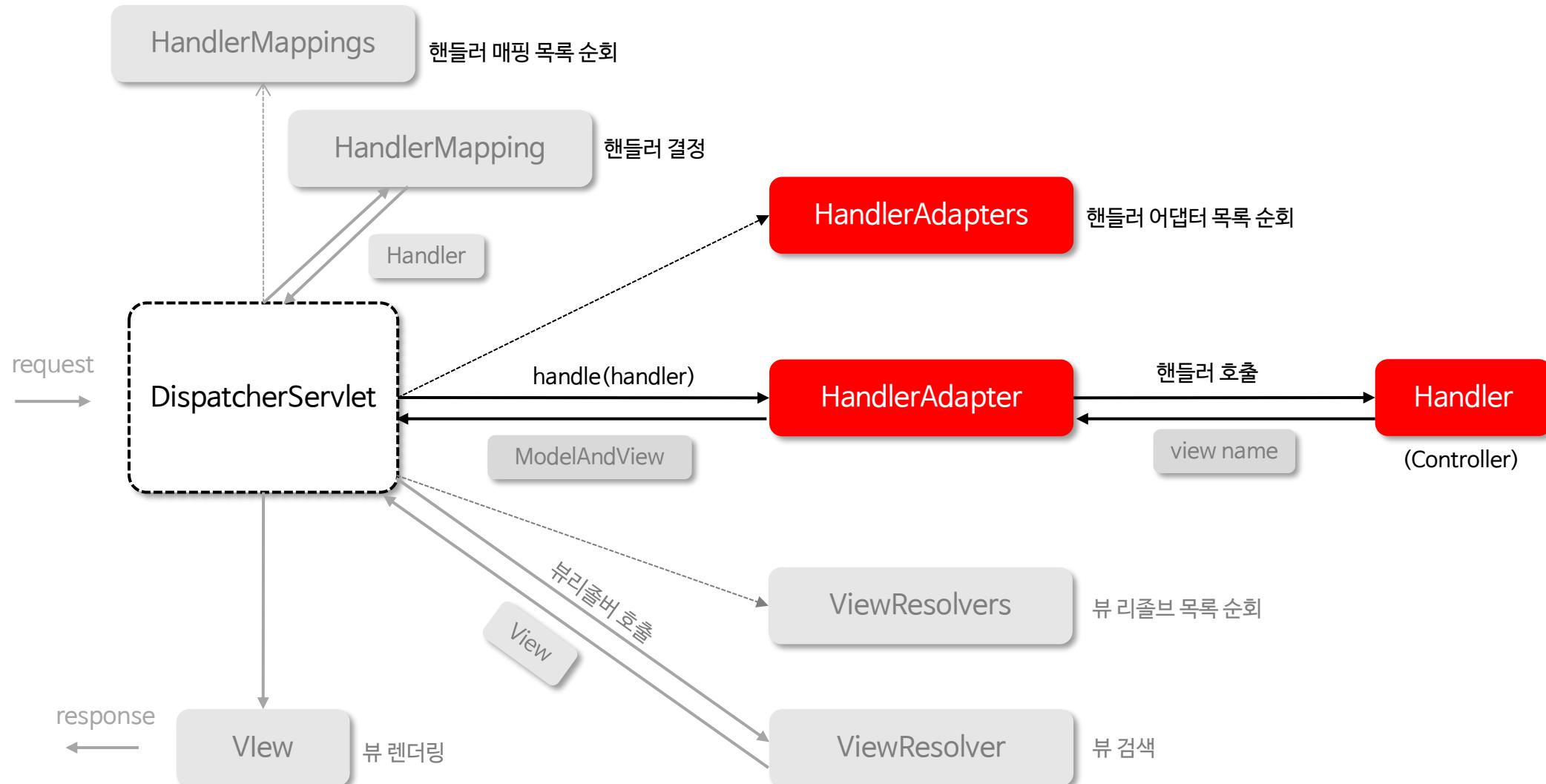
HandlerMethodReturnValueHandler

```
returnValueHandlers = {ArrayList@8063} size = 17
> 0 = {ModelAndViewMethodReturnValueHandler@8065}
> 1 = {ModelMethodProcessor@8066}
> 2 = {ViewMethodReturnValueHandler@8067}
> 3 = {ResponseBodyEmitterReturnValueHandler@8068}
> 4 = {StreamingResponseBodyReturnValueHandler@8069}
> 5 = {HttpEntityMethodProcessor@8070}
> 6 = {HttpHeadersReturnValueHandler@8071}
> 7 = {CallableMethodReturnValueHandler@8072}
> 8 = {DeferredResultMethodReturnValueHandler@8073}
> 9 = {AsyncTaskMethodReturnValueHandler@8074}
> 10 = {ServletModelAttributeMethodProcessor@8075}
> 11 = {RequestResponseBodyMethodProcessor@8076}
> 12 = {ViewNameMethodReturnValueHandler@8077}
> 13 = {MapMethodProcessor@8078}
> 14 = {CustomJsonReturnValueHandler@8079}
> 15 = {CustomHandlerMethodReturnValueHandler@8080}
> 16 = {ServletModelAttributeMethodProcessor@8081}
```

Method Arguments

<https://github.com/onjsdnjs/spring-mvc-master/tree/Method-Arguments>

✓ 스프링 MVC 아키텍처



✓ 개요

- 스프링의 메서드 매개변수(Method Arguments)는 컨트롤러 메서드에서 HTTP 요청 데이터를 직접 접근하고 처리할 수 있도록 다양한 매개변수를 지원한다
- 요청의 URL, 헤더, 본문, 쿠키, 세션 데이터 등과 같은 정보를 자동으로 매팅하여 개발자가 이를 쉽게 활용할 수 있도록 제공한다

✓ HandlerMethodArgumentResolver

HandlerMethodArgumentResolver

```
boolean supportsParameter(MethodParameter parameter); // 주어진 메서드 매개변수가 이 리졸버에 의해 지원되는지 여부를 반환한다
```

```
Object resolveArgument(MethodParameter param, ModelAndViewContainer container, NativeWebRequest request, WebDataBinderFactory factory) throws Exception
```

- HTTP 요청과 관련된 데이터를 컨트롤러 메서드의 파라미터로 변환하는 작업을 담당하는 클래스이다
- 다양한 유형의 파라미터 (예: @RequestParam, @PathVariable, @RequestBody 등)를 처리하기 위해 여러 HandlerMethodArgumentResolver 기본 구현체를 제공한다
- 개발자가 필요에 따라 HandlerMethodArgumentResolver 인터페이스를 직접 구현할 수 있다



✓ HandlerMethodArgumentResolver 설계 의도

```
@RestController  
public class RestApiController {  
    @GetMapping("/index")  
    public String index(@RequestParam("name") String name, @RequestParam("age") String age, User user) { . . . }
```

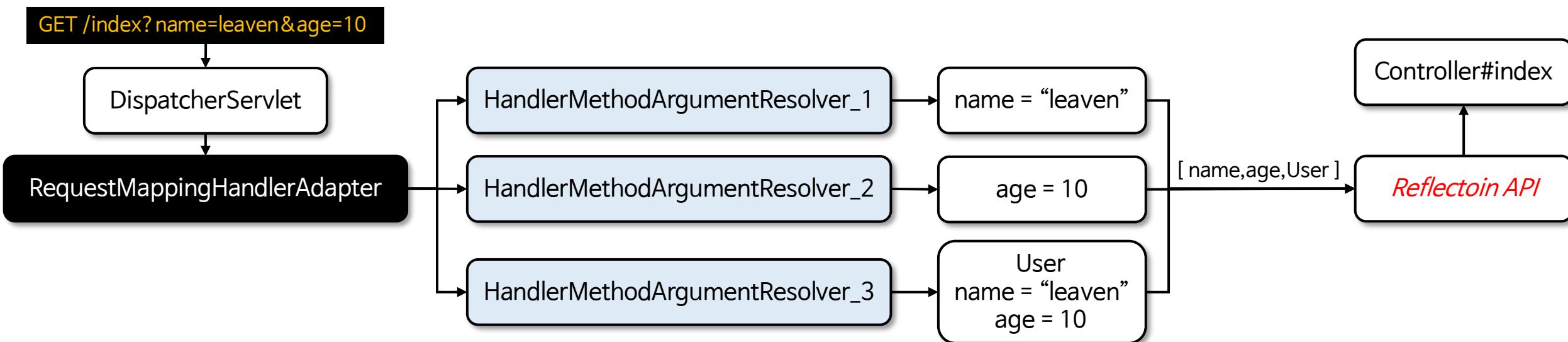
〈 메소드 호출은 최소한 두 가지가 만족되어야 한다〉

1) 클래스와 호출 메서드의 시그니처 정보

- 스프링은 HandlerMapping 을 통해 매핑한 핸들러에 RestApiController 와 public String index(String name, String age, User user) 와 같은 정보를 저장한다

2) 메서드 매개변수 개수만큼 각 타입별로 바인딩할 데이터를 생성해서 메서드 호출 시 전달해 주어야 함 (리플렉션 기술 사용)

- 스프링은 메서드 매개변수의 값을 요청 파라미터로부터 추출해서 생성하는데 이 역할을 하는 클래스가 바로 HandlerMethodArgumentResolver 이다



요약하면 `RequestMappingHandlerAdapter` 은 요청이 들어오면 여러 `ArgumentResolver` 를 사용하여 메서드 매개변수 값을 생성해서 실제 메서드를 호출한다

✓ 매개 변수로 사용할 수 있는 타입과 ArgumentResolver

Method Arguments

WebRequest, NativeWebRequest
jakarta.servlet.ServletRequest, jakarta.servlet.ServletResponse
jakarta.servlet.http.HttpSession
jakarta.servlet.http.PushBuilder
java.security.Principal
HttpMethod
java.util.Locale
java.util.TimeZone + java.time.ZoneId
java.io.InputStream, java.io.Reader
java.io.OutputStream, java.io.Writer
@PathVariable
@MatrixVariable
@RequestParam
@RequestHeader
@CookieValue
@RequestBody
HttpEntity
@RequestPart
java.util.Map, org.springframework.ui.Model, ModelMap
RedirectAttributes
@ModelAttribute
Errors, BindingResult
SessionStatus + class-level @SessionAttributes
UriComponentsBuilder
@SessionAttribute
@RequestAttribute
Any other argument

resolveArgument()

```
argumentResolvers = {ArrayList@8030} size = 30
> 0 = {RequestParamMethodArgumentResolver@8033}
> 1 = {RequestParamMapMethodArgumentResolver@8034}
> 2 = {PathVariableMethodArgumentResolver@8035}
> 3 = {PathVariableMapMethodArgumentResolver@8036}
> 4 = {MatrixVariableMethodArgumentResolver@8037}
> 5 = {MatrixVariableMapMethodArgumentResolver@8038}
> 6 = {ServletModelAttributeMethodProcessor@8039}
> 7 = {RequestResponseBodyMethodProcessor@8040}
> 8 = {RequestPartMethodArgumentResolver@8041}
> 9 = {RequestHeaderMethodArgumentResolver@8042}
> 10 = {RequestHeaderMapMethodArgumentResolver@8043}
> 11 = {ServletCookieValueMethodArgumentResolver@8044}
> 12 = {ExpressionValueMethodArgumentResolver@8045}
> 13 = {SessionAttributeMethodArgumentResolver@8046}
> 14 = {RequestAttributeMethodArgumentResolver@8047}
> 15 = {ServletRequestMethodArgumentResolver@8048}
> 16 = {ServletResponseMethodArgumentResolver@8049}
> 17 = {HttpEntityMethodProcessor@8050}
> 18 = {RedirectAttributesMethodArgumentResolver@8051}
> 19 = {ModelMethodProcessor@8052}
> 20 = {MapMethodProcessor@8053}
> 21 = {ErrorsMethodArgumentResolver@8054}
> 22 = {SessionStatusMethodArgumentResolver@8055}
> 23 = {UriComponentsBuilderMethodArgumentResolver@8056}
> 24 = {UserSessionArgumentResolver@8057}
> 25 = {CustomParamResolver@8058}
> 26 = {CustomModelResolver@8059}
> 27 = {PrincipalMethodArgumentResolver@8060}
> 28 = {RequestParamMethodArgumentResolver@8061}
> 29 = {ServletModelAttributeMethodProcessor@8062}
```

메서드 기본 매개변수

<https://github.com/onjsdnjs/spring-mvc-master/tree/메서드-기본-매개변수>

✓ 개요

- 스프링 MVC에서 메서드 파라미터에 공통적으로 선언할 수 있는 기본 인자들이 있으며 요청 및 응답 처리, 세션 관리, 인증 정보 접근 등 다양한 상황에서 적절하게 사용될 수 있다
- WebRequest, NativeWebRequest, HttpSession, Principal, HttpMethod, Reader, Writer에 대해 살펴 본다

✓ WebRequest / NativeWebRequest

- WebRequest와 NativeWebRequest는 웹 요청에 대한 다양한 정보를 제공하는 객체로서 HttpServletRequest보다 더 많은 메서드와 웹 요청 전반에 쉽게 접근한다

```
@GetMapping("/example")
public String handleWebRequest(WebRequest webRequest, NativeWebRequest nativeWebRequest) {

    String paramValue = webRequest.getParameter("name");
    String headerValue = webRequest.getHeader("User-Agent");
    long requestTime = webRequest.getTimestamp();

    HttpServletRequest request = nativeWebRequest.getNativeRequest(HttpServletRequest.class);
    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        Arrays.stream(cookies)
            .map(cookie -> cookie.getName() + "=" + cookie.getValue())
    }
    return "Parameter value: " + param;
}
```

✓ HttpSession

- HttpSession은 서버에 저장된 세션 데이터를 다룰 수 있게 해주는 객체로서 사용자의 세션 정보를 읽거나 설정할 수 있다

```
@GetMapping("/session")
public String handleHttpSession(HttpSession session) {
    session.setAttribute("user", "leaven ");
    return "Session set with user: leaven ";
}
```

HttpSession

Ⓜ ⌂ setAttribute (String, Object)	void
Ⓜ ⌂ getLastAccessedTime ()	long
Ⓜ ⌂ isNew()	boolean
Ⓜ ⌂ getServletContext ()	ServletContext
Ⓜ ⌂ setMaxInactiveInterval (int)	void
Ⓜ ⌂ getId()	String
Ⓜ ⌂ getCreationTime ()	long
Ⓜ ⌂ removeAttribute (String)	void
Ⓜ ⌂ getMaxInactiveInterval ()	int
Ⓜ ⌂ getAttribute (String)	Object
Ⓜ ⌂ getAttributeNames ()	Enumeration <String >
Ⓜ ⌂ invalidate()	void

✓ Principal

- Principal은 현재 인증된 사용자의 정보를 나타내는 객체로서 사용자 이름이나 인증된 사용자와 관련된 데이터를 제공해 준다
- 스프링 시큐리티와 통합되어 제공하는 기능이다

```
@GetMapping("/user")
public String handlePrincipal(Principal principal) {
    return "Logged in user: " + principal.getName();
}
```

✓ HttpMethod

- HttpMethod 는 요청 메서드(GET, POST 등)를 나타내며 현재 요청이 어떤 HTTP 메서드인지 확인할 수 있다

```
@PostMapping("/checkMethod")
public String handleHttpMethod(HttpMethod method) {
    if (method == HttpMethod.POST) {
        return "This is a POST request";
    }
    return "This is not a POST request";
}
```

HttpMethod	
serialVersionUID	long
OPTIONS	HttpMethod
HEAD	HttpMethod
name	String
DELETE	HttpMethod
PATCH	HttpMethod
TRACE	HttpMethod
values	HttpMethod []
GET	HttpMethod
POST	HttpMethod
PUT	HttpMethod

✓ InputStream&Reader / OutputStream&Writer

- Reader는 요청 본문을 읽는 데 사용되며 Writer는 응답 본문에 데이터를 작성하는 데 사용된다

```
@PostMapping("/readwrite")
public String handleReader(Reader reader, Writer writer) throws IOException {
    char[] buffer = new char[1024];
    int bytesRead = reader.read(buffer);
    System.out.println("Read " + bytesRead + " characters from request body");
    writer.write("This is the response body");
}
```

요청 파라미터 매개변수

@RequestParam

@PathVariable

@ModelAttribute

@RequestParam

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestParam>

✓ 개요

- @RequestParam 어노테이션은 HTTP 요청의 파라미터를 메서드의 매개변수에 바인딩 되도록 해 준다
- @RequestParam 은 URL 쿼리 파라미터, 폼 데이터, 그리고 멀티파트 요청을 매핑하며 HTTP 본문 요청은 처리하지 않는다 (HttpMessageConvertter 가 처리)
- 주로 int, long 과 같은 기본형, 기본형 래퍼 클래스, String 형 매개변수를 바인딩할 때 사용하며 대부분의 객체 타입은 처리하지 않는다 (@ModelAttribute 가 처리)
- 내부적으로 RequestParamMethodArgumentResolver 구현체가 사용되며 request.getParameterValues() 와 같은 API 를 사용하여 바인딩을 해결하고 있다

✓ 구조

@	RequestParam
Ⓜ	required() boolean
Ⓜ	defaultValue() String
Ⓜ	value() String
Ⓜ	name() String

- name - 파라미터의 이름을 지정한다
- required - 해당 파라미터가 반드시 있어야 하는지 여부를 설정한다. 기본값은 true 이다
- defaultValue - 파라미터가 없을 경우 기본값을 설정한다

✓ 기본 구현

```
@RequestMapping("/greet")
public String greetUser(@RequestParam(name = "username") String username) {
    return "Hello, " + username;
}
```

- @RequestParam 의 name 속성에 요청 파라미터와 바인딩할 이름을 입력하고 매개변수 앞에 선언한다
- 쿼리 파라미터와 폼데이터 모두 바인딩 되는데 getParameter() 가 두 방식의 파라미터를 다 받기 때문이다

URL 쿼리 파라미터

```
http://localhost:8080/greet?username=leaven
```

폼 데이터

```
<form action="/greet" method="GET">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required>
    <button type="submit">Submit</button>
</form>
```

✓ required 속성

```
@RestController
public class RequiredExampleController {
    @GetMapping("/required-true")
    public String requiredTrue(@RequestParam(name = "param", required = true) String param) { // required=true (기본값)
        return "필수 파라미터 값: " + param;
    }
    @GetMapping("/required-false")
    public String requiredFalse(@RequestParam(name = "param", required = false) String param) { // required=false
        return param != null ? "받은 파라미터 값: " + param : "파라미터가 없습니다.";
    }
}
```

- 요청 및 결과

- 1) /required-true?param=test → 필수 파라미터 값: test
- 2) /required-true → 400 Bad Request (필수 파라미터가 없어서 발생)
- 3) /required-false?param=test → 받은 파라미터 값: test
- 4) /required-false → 파라미터가 없습니다.

- 설명

- 1) required=true 는 클라이언트가 요청 시 파라미터를 반드시 포함해야 하므로, 누락 시 400 오류가 발생합니다.
- 2) 파라미터가 선택적인 경우 required=false 로 설정하는 것이 안전하다

✓ defaultValue 속성

```
@RestController
public class DefaultValueExampleController {

    @GetMapping("/default-value")
    public String defaultValue(@RequestParam(name = "param", defaultValue = "기본값") String param) {
        return "받은 값: " + param;
    }
}
```

- 요청 및 결과

- 1) /default-value?param=test → 받은 값: test
- 2) /default-value → 받은 값: 기본값
- 3) /default-value?param= → 받은 값: 기본값

- 설명

- 1) defaultValue 가 설정되면 required 는 자동으로 false로 동작한다
- 2) 파라미터 값이 비어있는 경우(? param=)에도 기본값이 적용된다

✓ name 속성을 생략하거나 @RequestParam 을 선언하지 않은 경우

```
@RestController  
public class NameExampleController {  
    @GetMapping("/noName")  
    public String noName(@RequestParam String param) { // name 속성 생략  
        return "받은 값: " + param;  
    }  
    @GetMapping("/noRequestparam")  
    public String noRequestParam(String param) { // @RequestParam 생략  
        return "받은 값: " + param;  
    }  
}
```

• 요청 및 결과

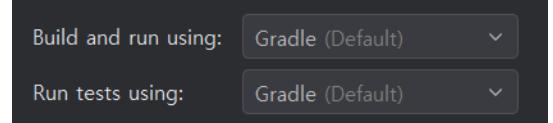
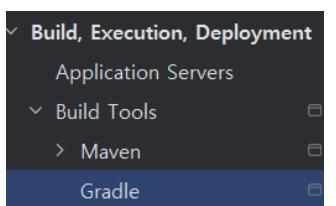
- 1) /noName?param=test → 받은 값: test
- 2) /noRequestparam?param=test → 받은 값: test

• 설명

- 1) 요청 파라미터 이름과 메서드 매개변수 이름이 동일하면 @RequestParam을 생략해도 동작한다
- 2) 명시적으로 선언하는 것이 유지보수 및 가독성이 좋다. 가급적 적는 것을 권장한다

주의 사항

- 스프링 부트 3.2 이상부터 name 속성을 생략하거나 @RequestParam을 선언하지 않으면 오류가 발생한다 (@PathVariable 동일)
- 해결 방안이 몇 가지 있으나 name 속성을 사용하면 아무런 문제 될 것이 없으니 습관적으로 선언하고 사용하길 바란다
- 인텔리제이 기준 Settings 의 Build, Execution, Deployment Build Tools Gradle 로 가서 Build and run using 을 Gradle 로 선택한다



✓ 기본형 매개변수와 빈 값 처리

```
@RestController  
public class PrimitiveExampleController {  
    @GetMapping("/primitive")  
    public String primitive(@RequestParam int number) { // 기본형 매개변수  
        return "숫자: " + number;  
    }  
  
    @GetMapping("/primitive-with-default")  
    public String primitiveWithDefault(@RequestParam(name = "number", defaultValue = "0") int number) { // 기본형 매개변수와 빈값 처리  
        return "숫자: " + number;  
    }  
}
```

- 요청 및 결과

- 1) /primitive?number=42 → 숫자: 42
- 2) /primitive → 400 Bad Request (기본형에 null 값 전달 불가)
- 3) /primitive-with-default?number= → 숫자: 0 (빈 값 처리)
- 4) /primitive-with-default → 숫자: 0

- 설명

- 1) 기본형은 null 을 허용하지 않으므로 기본형의 Wrapper 클래스나 defaultValue 를 사용하여 기본값을 설정해야 한다

✓ Map과 MultiValueMap 매개변수

```
@RestController
public class MapExampleController {
    @GetMapping("/map")
    public String map(@RequestParam Map<String, String> params) { // 모든 요청 파라미터를 Map으로 처리
        return "받은 파라미터: " + params.toString();
    }
    @GetMapping("/multivalue-map")
    public String multiValueMap(@RequestParam MultiValueMap<String, String> params) { // MultiValueMap으로 처리
        return "받은 파라미터: " + params.toString();
    }
}
```

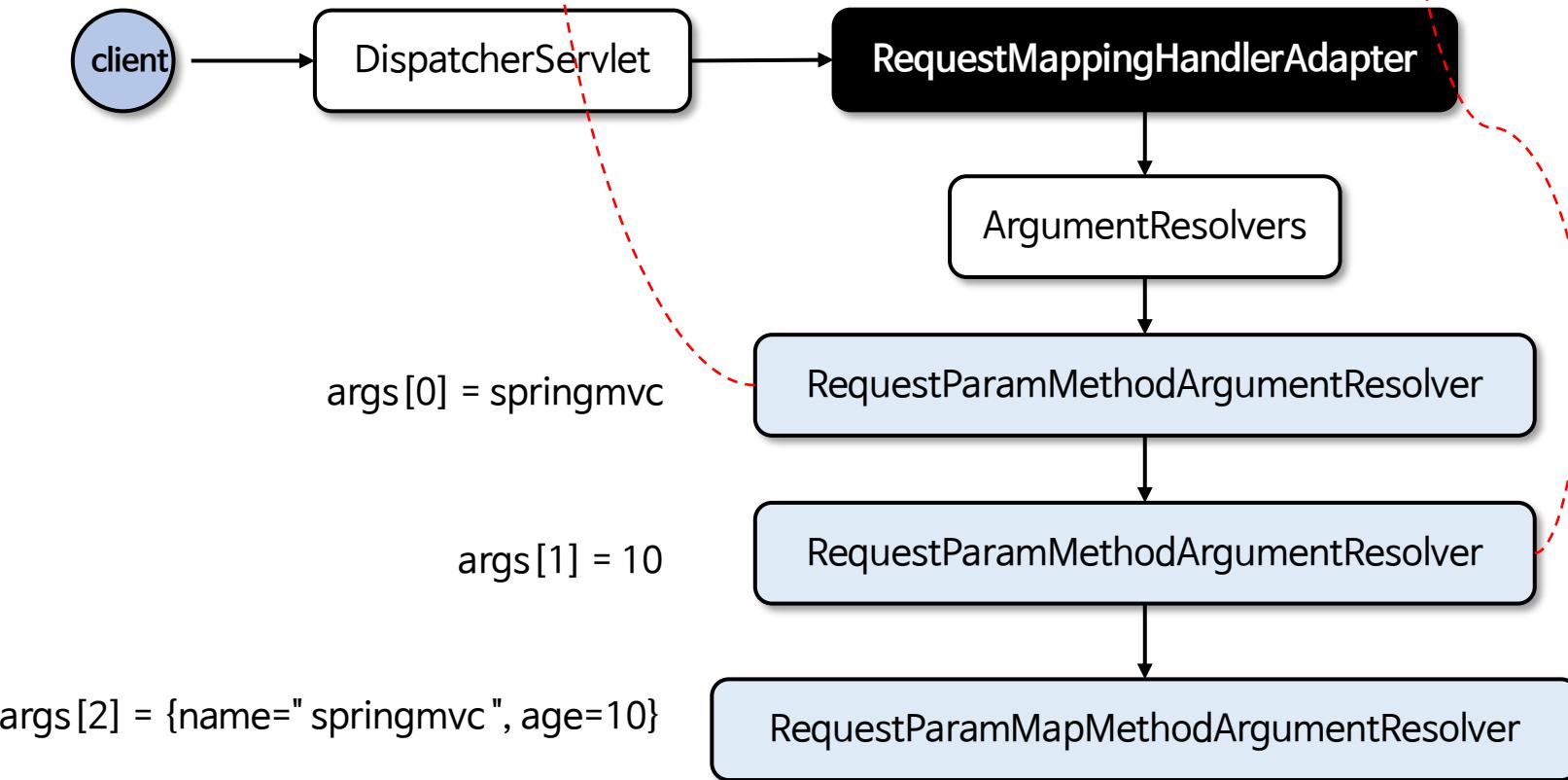
- 요청 및 결과

- 1) /map?key1=value1&key2=value2 → 받은 파라미터: {key1=value1, key2=value2}
- 2) /multivalue-map?key1=value1&key1=value2 → 받은 파라미터: {key1=[value1, value2]}

✓ RequestParam [Map] MethodArgumentResolver

```
@GetMapping("/greet")
public String greetUser(@RequestParam(name = "name") String name, @RequestParam(name = "age") String age, @RequestParam Map<String, String> allParams) {
    return "Hello, " + name + " | " + age;
}
```

/greet?age=10&name=springmvc



@PathVariable

<https://github.com/onjsdnjs/spring-mvc-master/tree/@PathVariable>

✓ 개요

- @PathVariable 은 @RequestMapping 에 지정한 URI 템플릿 변수에 포함된 값을 메서드의 매개변수로 전달하기 위해 사용하는 어노테이션이다
- @PathVariable 은 GET, DELETE, PUT, POST 요청에서 사용 할 수 있다

✓ URI 패턴

경로 변수에서 한 문자와 일치

```
@GetMapping("/resources/ima?e.png") /resources/image.png 또는 /resources/imaxe.png가 매치된다
public String getSingleCharacterMatch() {
    return "경로에서 한 문자만 일치하는 URL입니다.";
}
```

경로 변수에서 0개 이상의 문자와 일치

```
@GetMapping("/resources/ *.png")
public String getSingleCharacterMatch() {
    return "경로에서 0개 이상의 문자와 일치하는 URL입니다.";
}
```

여러 경로 변수와 일치

```
@GetMapping("/resources/**") /resources/images/test.png, /resources/css/style.css 등이 매치된다
public String getSingleCharacterMatch() {
    return "resources 디렉터리 및 그 하위 모든 경로와 일치합니다.";
}
```

✓ 구조

@ PathVariable	
Ⓜ️ ⌂ value()	String
Ⓜ️ ⌂ required()	boolean
Ⓜ️ ⌂ name()	String

- **name** - URL에서 추출할 변수의 이름을 지정합니다
- **required** - 해당 파라미터가 반드시 있어야 하는지 여부를 설정한다. 기본값은 true이다
- **value** - name과 동일한 기능을 한다

✓ 기본 구현

```
@GetMapping("/projects/{project}")
public String getProjectVersions(@PathVariable("project") String project) {
    return "프로젝트 ID: " + project + " 정보를 표시합니다.";
}
```

/projects/myProject 으로 요청하면 myProject 가 캡처된다.

✓ 여러 군데 지정

```
@GetMapping("/projects/{project}/versions/{version}")
public String getProjectVersions(@PathVariable("project") String project, @PathVariable("version") String version) {
    return "프로젝트 ID: " + project + "의 버전 " + version + " 정보를 표시합니다.";
}
```

/projects/myProject/versions/1.0 으로 요청하면 myProject 와 1.0 이 캡처된다.

✓ 정규 표현식과 함께 변수를 매칭하고 캡처

```
@GetMapping("/projects/{projectId:[a-z]+}/details")
public String getProjectDetails(@PathVariable("projectId") String projectId) {
    return "Project ID: " + projectId;
}
```

이 패턴에서 {projectId:[a-z]+} 는 projectId 가 하나 이상의 소문자로만 구성되어야 함을 나타낸다.

이 패턴은 /projects/alpha/details와 같은 URL에 매칭되며 /projects/123/details와 같은 URL은 매칭되지 않는다

✓ @PathVariable에서 name을 정의하지 않는 경우

- @PathVariable에서 name 속성을 정의하지 않으면 메서드 파라미터의 이름이 경로 변수의 이름과 동일할 경우 자동으로 매핑된다
- 스프링 부트 3.2 이상에서는 지원하지 않음 - @RequestParam 챕터 참고

```
@GetMapping("/users/{userId}")
public String getUser(@PathVariable String userId) { GET /users/123 => userId 에는 123 이 바인딩 된다
    return "User ID: " + userId;
}
```

✓ @PathVariable 자체를 선언하지 않는 경우

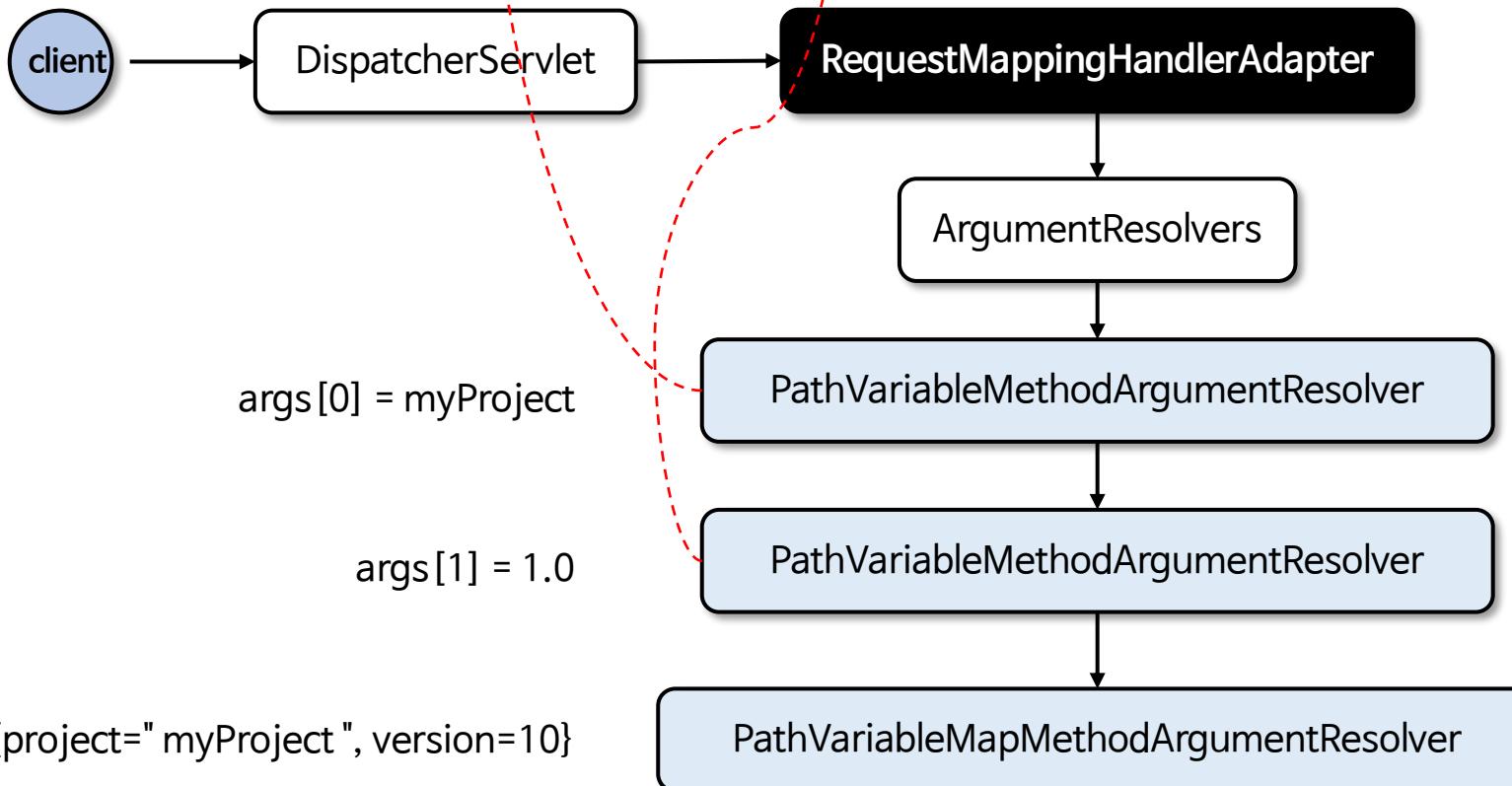
- @PathVariable 애노테이션을 아예 사용하지 않으면 URL 경로의 변수를 메서드의 파라미터로 바인딩할 수 없다

```
@GetMapping("/users/{userId}")
public String getUser(String userId) { GET /users/123 => userId 파라미터는 전달되지 않으며 null 이 되거나 에러가 발생한다
    return "User ID: " + userId;
}
```

✓ PathVariable(Map)MethodArgumentResolver

```
@GetMapping("/projects/{project}/versions/{version}")
public String getProjectVersions(@PathVariable String project, @PathVariable String version, @PathVariable Map<String, String> allParams) {
    return "프로젝트 ID: " + project + "의 버전 " + version + " 정보를 표시합니다.";
}
```

/projects/myProject/versions/1.0



@ModelAttribute

<https://github.com/onjsdnjs/spring-mvc-master/tree/@ModelAttribute>

✓ 개요

- @ModelAttribute는 스프링 MVC에서 주로 폼 데이터나 요청 파라미터를 모델 객체에 바인딩할 때 사용하는 어노테이션이다
- 이 어노테이션은 요청 파라미터를 특정 객체의 각 필드(요청 파라미터명과 일치)에 바인딩하고 이후 자동으로 모델에 추가하여 뷰에서 사용할 수 있게 한다
- 일반적으로 기본형 타입(int, long, String ..)의 바인딩은 @RequestParam 이 처리하고 객체 타입은 @ModelAttribute 가 처리한다고 보면 된다

✓ @ModelAttribute 기본

```
POST /submit  
Content-Type: application/x-www-form-urlencoded  
username=springmvc&email=a@a.com
```

```
@PostMapping("/submit")  
public String processSubmit(@ModelAttribute("user") User user) {  
    // 'user' 객체는 요청 파라미터를 바탕으로 자동으로 값이 바인딩 됨  
    return "result";  
}
```

바인딩 결과

```
User [username= springmvc, email=a@a.com]
```

모델 객체 접근

```
<h1>사용자 이름: <span th:text="${user.username}"></span></h1>
```

```
public class User {  
  
    private String username;  
    private String email;  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.username = username;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
}
```

참고 : @ModelAttribute는 파라미터가 객체일 경우 생략 가능하며 동일하게 데이터를 바인딩 해 준다

✓ 데이터 바인딩 없이 모델에 접근

- 데이터 바인딩 없이 모델에 접근하고자 할 경우 `@ModelAttribute(binding=false)`로 설정하여 데이터 바인딩 없이 객체에 접근할 수 있다

```
@PostMapping("/update")
public String update(@ModelAttribute(binding=false) User user) {
    // user'는 데이터 바인딩 없이 접근 가능
    return "result";
}
```

✓ 생성자 바인딩 : `@BindParam`

- `@ModelAttribute`은 요청 파라미터와 일치하는 생성자를 통해 객체를 생성할 수도 있으며 생성자 바인딩을 사용할 때는 `@BindParam`을 이용해 요청 파라미터의 이름을 매핑할 있다

```
@PostMapping("/account")
public String processAccount(@ModelAttribute Account account) {
    System.out.println("First Name: " + account.getFirstName());
    return "accountResult";
}
```

```
public class Account {
    private final String firstName;
    public Account(@BindParam("first-name") String firstName) {
        this.firstName = firstName;
    }
}
```

```
POST /account
Content-Type: application/x-www-form-urlencoded
first-name=leaven
```

✓ 경로 변수 or 요청 파라미터 객체 바인딩

- Converter<String, T>가 등록되어 있고 @ModelAttribute 속성 이름이 경로 변수와 일치하는 경우 Converter를 사용하여 모델 객체를 가져올 수 있다

```
@PutMapping("/accounts/{account}")
public String save(@ModelAttribute("account") Account account) {
    System.out.println("Account ID: " + account.getId());
    System.out.println("Account Name: " + account.getName());
    return "accountSaved"; // 결과 페이지로 이동
}
```

```
@Component
public class StringToAccountConverter implements Converter<String, Account> {
    @Override
    public Account convert(String id) {
        return new Account(id, "Default Name");
    }
}
```

```
PUT /accounts/12345
```

```
public class Account {
    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

✓ 메서드에 @ModelAttribute 선언

- 컨트롤러에서 모델에 데이터를 추가하는 역할을 한다. 이 경우 메서드가 리턴한 객체가 자동으로 모델에 추가된다
- 주로 뷰에서 공통적으로 사용되는 데이터를 미리 모델에 추가할 때 사용된다. 예를 들어 드롭다운 리스트에 넣을 데이터나 공통적으로 사용되는 객체 등을 미리 준비하는 경우 유용하다

```
// 메서드에 @ModelAttribute 선언: 뷰에서 공통적으로 사용될 데이터를 모델에 추가
ModelAttribute("user")

public AccountDto addUser() { // 먼저 호출된다
    return new AccountDto("leaven", "1234");
}

@GetMapping("/form")
public String showForm(Model model, AccountDto account) {
    return "userForm";
}
```

① model = {BindingAwareModelMap@9355} size = 1

 └ "user" -> {AccountDto@9360}

 └ key = "user"

 └ value = {AccountDto@9360}

 └ username = "leaven"

 └ password = "1234"

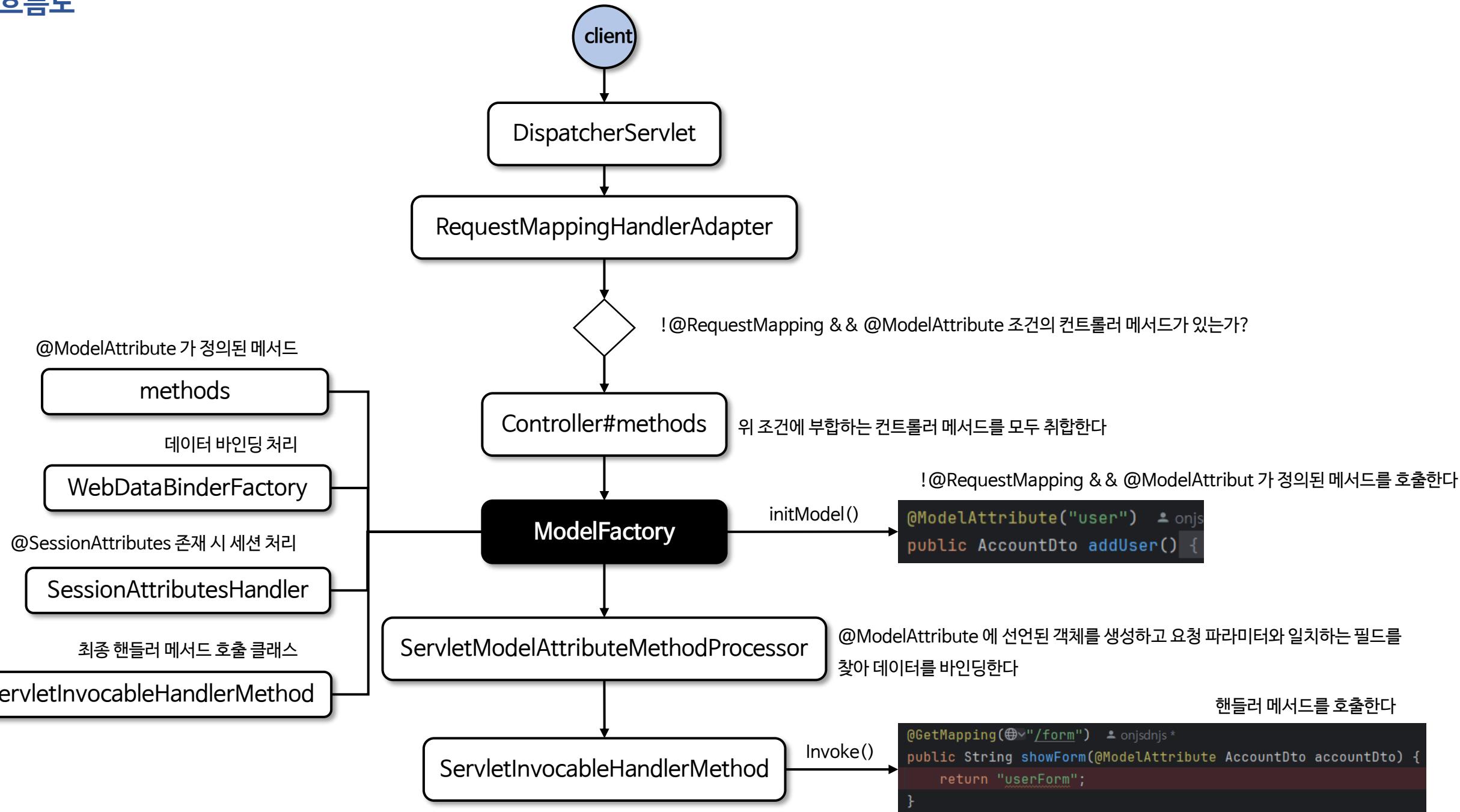
✓ userForm.html

```
<form action="/submitUser" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" th:value="${user.username}" />

    <label for="email">Email:</label>
    <input type="email" id="email" name="password" th:value="${user.password}" />

    <button type="submit">Submit</button>
</form>
```

✓ 흐름도



HTTP 본문 매개변수

HttpEntity & RequestEntity
@RequestBody

HttpEntity / RequestEntity

<https://github.com/onjsdnjs/spring-mvc-master/tree/HttpEntity-RequestEntity>

✓ 개요

- HTTP 요청이 파라미터나 폼 데이터가 아닌 요청 본문(Body)일 경우 앞서 보았던 `@RequestParam`이나 `@ModelAttribute`는 요청을 매개변수에 바인딩 할 수 없다
- 일반적으로 헤더정보가 Content-type=application/json과 같이 되어 있는 HTTP 본문 요청은 `getParameter()`로 읽어 드릴 수 없으며 직접 본문 데이터를 파싱해서 읽는 방식으로 처리해야 한다

```
@RestController  
public class HttpEntityController {  
  
    @PostMapping("/param")  
    public String handleRequestParam(@RequestParam String name) {  
        return "Received name: " + name;  
    }  
  
    @PostMapping("/model")  
    public String handleModelAttribute(@ModelAttribute User user) {  
        return "Received user: " + user.getName() + ", age: " + user.getAge();  
    }  
  
    public static class User {  
        private String name;  
        private int age;  
        // Getters and Setters  
    }  
}
```

요청

```
POST /param HTTP/1.1  
Content-Type: application/json  
{  
    "name": "leaven",  
    "age": 25  
}
```

@RequestParam 요청 결과

요청을 보내면 Spring은 JSON 데이터에서 "name" 값을 읽을 수 없으며
`@RequestParam`은 요청 본문을 매핑하려 하지 않고 쿼리 파라미터를 찾으려 한다

```
Required request parameter 'name' for method parameter type String is not present
```

@ModelAttribute 요청 결과

요청을 보내면 JSON 데이터가 제공되었지만 `@ModelAttribute`는 이를 파싱하지 못하고 객체의 기본값(null 또는 0)으로 설정됩니다

```
Received user: null, age: 0
```

✓ HttpServletRequest - InputStream, Reader

- HTTP 요청 본문(Body) 은 HttpServletRequest의 InputStream 또는 Reader를 통해 접근할 수 있으며 요청 본문은 getInputStream() 또는 getReader() 메서드를 사용하여 읽을 수 있다

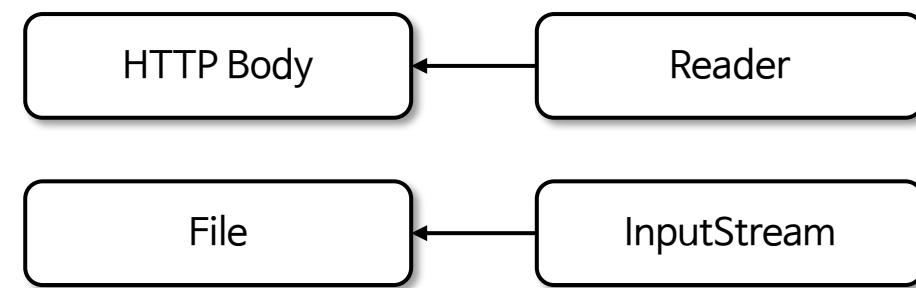
```
@RestController  
public class RequestBodyController {  
  
    @PostMapping("/readbody")  
    public String readBody(HttpServletRequest request) throws IOException {  
  
        StringBuilder requestBody = new StringBuilder();  
        try (BufferedReader reader = request.getReader()) {  
            String line;  
            while ((line = reader.readLine()) != null) {  
                requestBody.append(line);  
            }  
        }  
        return "Received Body: " + requestBody.toString();  
    }  
}
```

요청

```
POST /readbody HTTP/1.1  
Content-Type: application/json  
{  
    "name": "leaven",  
    "age": 25  
}
```

응답

```
Received Body: {"name": "John", "age": 25}
```



HTTP Body 나 이진 데이터들은 자바의 스트림을 사용해서 직접 데이터를 읽어드려야 한다

✓ HttpEntity

- HttpEntity는 기존 HttpServletRequest나 HttpServletResponse를 사용하여 요청 및 응답 데이터를 처리하는 복잡성을 해결하기 위해 도입되었다
- HttpHeaders와 Body 데이터를 하나의 객체로 통합하였고 JSON, XML, 문자열, 바이너리 데이터 등 다양한 본문 데이터 형식을 처리 가능하게 하였다
- 내부적으로 HttpMessageConverter 객체가 작동되어 본문을 처리한다

✓ 구조

© HttpEntity<T>	
∅	EMPTY HttpEntity<?>
∅	headers HttpHeaders
∅	body T?
(m)	hashCode() int
(m)	getHeaders() HttpHeaders
(m)	getBody() T?
(m)	hasBody() boolean
(m)	equals(Object?) boolean
(m)	toString() String

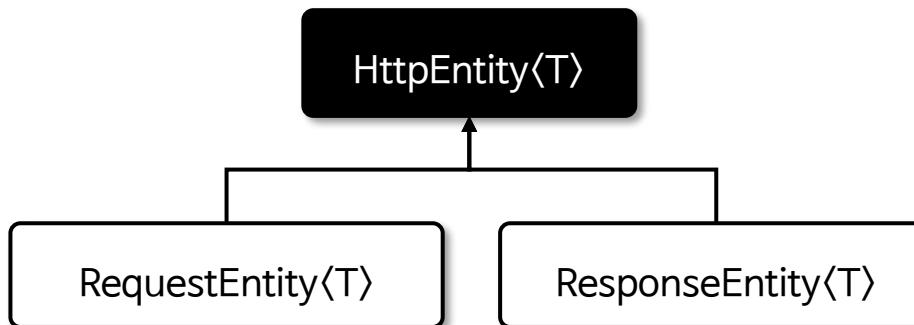
1. 생성자

- HttpEntity<T>() - 본문과 헤더 없이 객체 생성.
- HttpEntity<T>(T body) - 본문만 포함.
- HttpEntity<T>(T body, HttpHeaders headers) - 본문과 헤더를 포함.

2. 메서드

- T getBody() - 요청 또는 응답의 본문 데이터 반환.
- HttpHeaders getHeaders() - 요청 또는 응답의 헤더 반환.

3. 상속 구조



✓ 구현 예제 1

```
@RestController  
public class HttpEntityController {  
  
    @PostMapping("/process")  
    public ResponseEntity<String> processRequest(HttpEntity<String> httpEntity) {  
        // 본문과 헤더를 추출  
        String body = httpEntity.getBody();  
        HttpHeaders headers = httpEntity.getHeaders();  
  
        // ObjectMapper를 사용하여 JSON 본문을 User 객체로 변환  
        ObjectMapper objectMapper = new ObjectMapper();  
        User user = objectMapper.readValue(body, User.class);  
  
        System.out.println("Headers: " + headers);  
        System.out.println("User: Name=" + user.getName() + ", Age=" + user.getAge());  
  
        // 응답 헤더 설정  
        HttpHeaders responseHeaders = new HttpHeaders();  
        responseHeaders.add("Custom-Header", "Processed");  
        return new ResponseEntity<>(  
            "Processed User: Name=" + user.getName() + ", Age=" + user.getAge(),  
            responseHeaders, HttpStatus.OK  
        );  
    }  
}
```

✓ 구현 예제 2

```
@RestController  
public class HttpEntityController {  
  
    @PostMapping("/process")  
    public ResponseEntity<String> processRequest(HttpEntity<User> httpEntity) {  
        // 본문과 헤더를 추출  
        User user = httpEntity.getBody(); // 요청 본문을 User 객체로 직접 매팅  
        HttpHeaders headers = httpEntity.getHeaders(); // 요청 헤더  
  
        System.out.println("Headers: " + headers);  
        System.out.println("User: Name=" + user.getName() + ", Age=" + user.getAge());  
  
        // 응답 헤더 설정  
        HttpHeaders responseHeaders = new HttpHeaders();  
        responseHeaders.add("Custom-Header", "Processed");  
        return new ResponseEntity<>(  
            "Processed User: Name=" + user.getName() + ", Age=" + user.getAge(),  
            responseHeaders, HttpStatus.OK  
        );  
    }  
}
```

✓ 요청 및 응답 결과

요청

```
POST /process HTTP/1.1
Authorization: Bearer token
Content-Type: application/json
{
  "name": "leaven",
  "age": 25
}
```

로그

```
Headers: [Authorization:"Bearer token", ContentType:"application/json"]
User: Name=leaven, Age=25
```

응답

```
HTTP/1.1 200 OK
Custom-Header: Processed
Content-Type: text/plain

Processed User: Name=leaven, Age=25
```

```
@RestController
public class HttpEntityController {

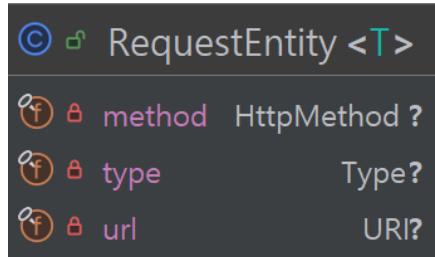
    @PostMapping("/process")
    public ResponseEntity<String> processRequest(HttpEntity<User> httpEntity) {
        // 본문과 헤더를 추출
        User user = httpEntity.getBody(); // 요청 본문을 User 객체로 직접 매팅
        HttpHeaders headers = httpEntity.getHeaders(); // 요청 헤더

        System.out.println("Headers: " + headers);
        System.out.println("User: Name=" + user.getName() + ", Age=" + user.getAge());

        // 응답 헤더 설정
        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.add("Custom-Header", "Processed");
        return new ResponseEntity<>(
            "Processed User: Name=" + user.getName() + ", Age=" + user.getAge(),
            responseHeaders, HttpStatus.OK
        );
    }
}
```

✓ RequestEntity

- HttpEntity의 확장 버전으로 HTTP 메서드와 대상 URL도 포함하며 RestTemplate에서 요청을 준비하거나 @Controller 메서드에서 요청 입력을 나타낼 때 사용된다



```
@RestController
public class RequestEntityController {
    @PostMapping("/process")
    public ResponseEntity<String> handleRequest(RequestEntity<User> requestEntity) {
        User user = requestEntity.getBody(); // 요청 본문(User 객체)
        String method = requestEntity.getMethod().name(); // HTTP 메서드
        String url = requestEntity.getUrl().toString(); // 요청 URL

        System.out.println("HTTP Method: " + method);
        System.out.println("URL: " + url);
        System.out.println("User : Name=" + user.getName() + ", Age=" + user.getAge());

        return ResponseEntity.ok("User processed: Name=" + user.getName() + ", Age=" + user.getAge());
    }
}
```

요청

```
POST /process HTTP/1.1
Content-Type: application/json
{
    "name": "leaven",
    "age": 25
}
```

서버 로그

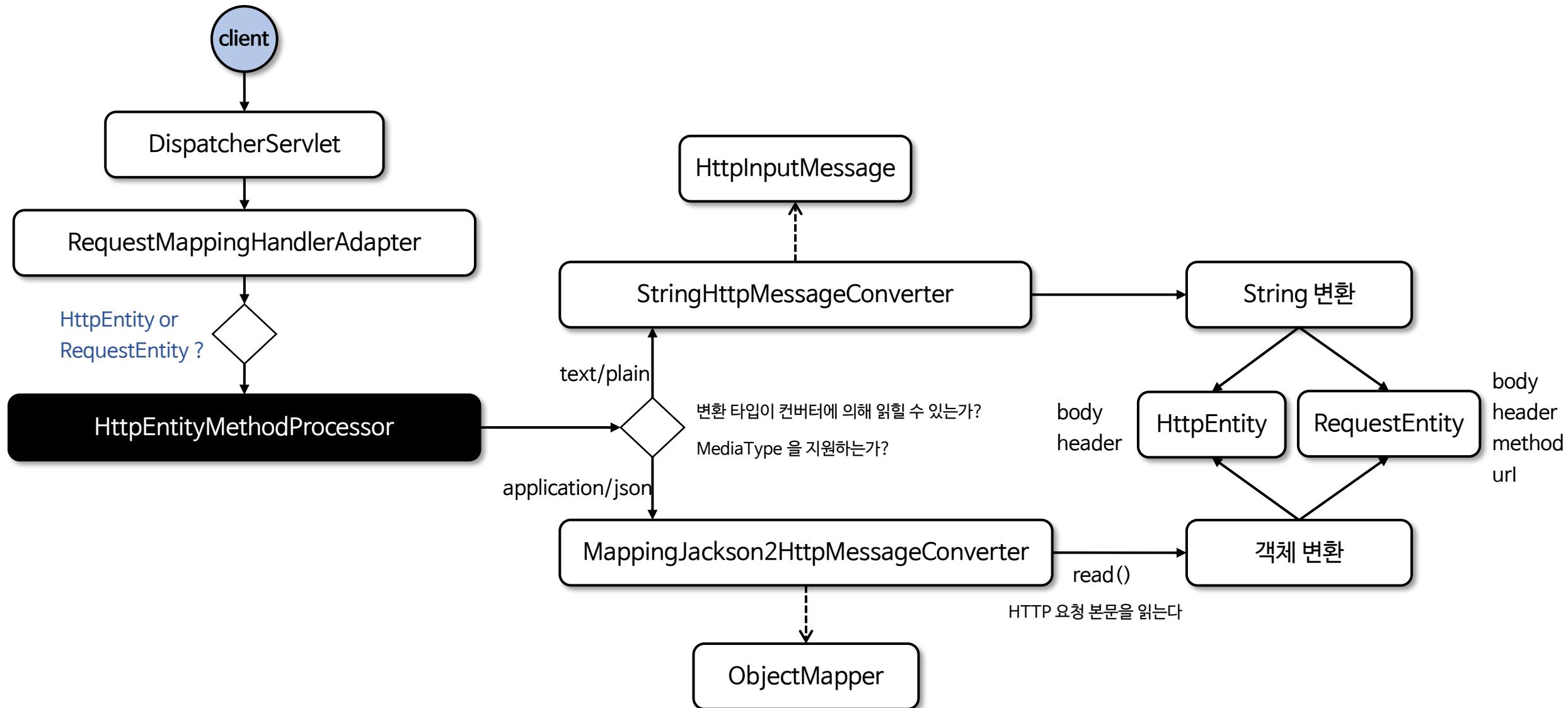
```
HTTP Method: POST
URL: http://localhost:8080/api/process
User : Name=leaven, Age=25
```

응답

```
HTTP/1.1 200 OK
Content-Type: text/plain
User processed: Name=leaven, Age=25
```

✓ 흐름도

- 스프링은 `HttpEntity`, `RequestEntity`, `@RequestBody` 등 Http 요청 본문 처리를 위해 `HttpMessageConverter` 인터페이스를 제공한다

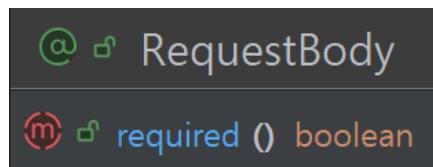


@RequestBody

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestBody>

✓ 개요

- @RequestBody 는 HTTP 요청 본문(HTTP Body)을 자동으로 객체로 매핑하는 데 사용되며 내부적으로 HttpMessageConverter 객체가 작동되어 본문을 처리한다
- HttpEntity 및 RequestEntity 도 요청 본문을 처리해 주지만 지정된 객체로 자동 매핑을 해 주지는 않는다
- @Valid 애너테이션과 함께 사용하면 요청 본문의 유효성을 쉽게 검증할 수 있다



- `@RequestBody` 는 반드시 요청 본문이 있어야 하며 요청 본문이 없을 경우 `HttpMessageNotReadableException` 이 발생한다
- `required = false` 로 설정하면 요청 본문이 없을 때 `null` 을 허용한다

✓ @RequestBody vs HttpEntity

```
@RestController
public class RequestBodyController {
    @GetMapping("/users")
    public ResponseEntity<String> createUser(@RequestBody User user) {
        return ResponseEntity.ok("User: " + user.getName());
    }
}
```

▼ ④ user = {User@6954} "User(name=leaven, age=10)"
 > ④ name = "leaven"
 > ④ age = 10

```
GET http://localhost:8080/users
Content-Type: application/json
{
    "name": "leaven",
    "age": "10"
}
```

VS

```
@RestController
public class HttpEntityController {
    @GetMapping("/users")
    public ResponseEntity<String> processRequest(HttpEntity<User> httpEntity) {
        User user = httpEntity.getBody();
        return ResponseEntity.ok("User : " + user.getName());
    }
}
```

④ httpEntity = {HttpEntity@6931} "<User(name=leaven, age=10)"
 > ④ headers = {ReadOnlyHttpHeaders@6938} size = 6
 > ④ body = {User@6939} "User(name=leaven, age=10)"
 > ④ name = "leaven"
 > ④ age = 10

- `@RequestBody` 는 요청의 Content-Type 헤더를 기반으로 적절한 `HttpMessageConverter` 를 선택하기 때문에 Content-Type 헤더가 올바르게 설정되어야 한다
- `@RequestBody` 는 생략하면 안된다. 생략할 경우 기본형은 `@RequestParam`, 객체 타입은 `@ModelAttribute` 가 작동하게 되지만 정확한 결과를 보장할 수 없다

✓ 구현 예제

```
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<String> createUser(@RequestBody UserDto userDto) {
    return ResponseEntity.ok("User name: " + userDto.getName());
}
```

application/json 타입으로 JSON 데이터 받기

```
@PostMapping(consumes = MediaType.TEXT_PLAIN_VALUE)
public ResponseEntity<String> handlePlainText(@RequestBody String message) {
    return ResponseEntity.ok("message: " + message);
}
```

text/plain 타입으로 문자열 데이터 받기

```
@PostMapping(consumes = MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public ResponseEntity<String> handleForm(@RequestBody MultiValueMap<String, String> formData) {
    String name = formData.getFirst("name");
    String email = formData.getFirst("email");
    return ResponseEntity.ok("Name = " + name + ", Email = " + email);
}
```

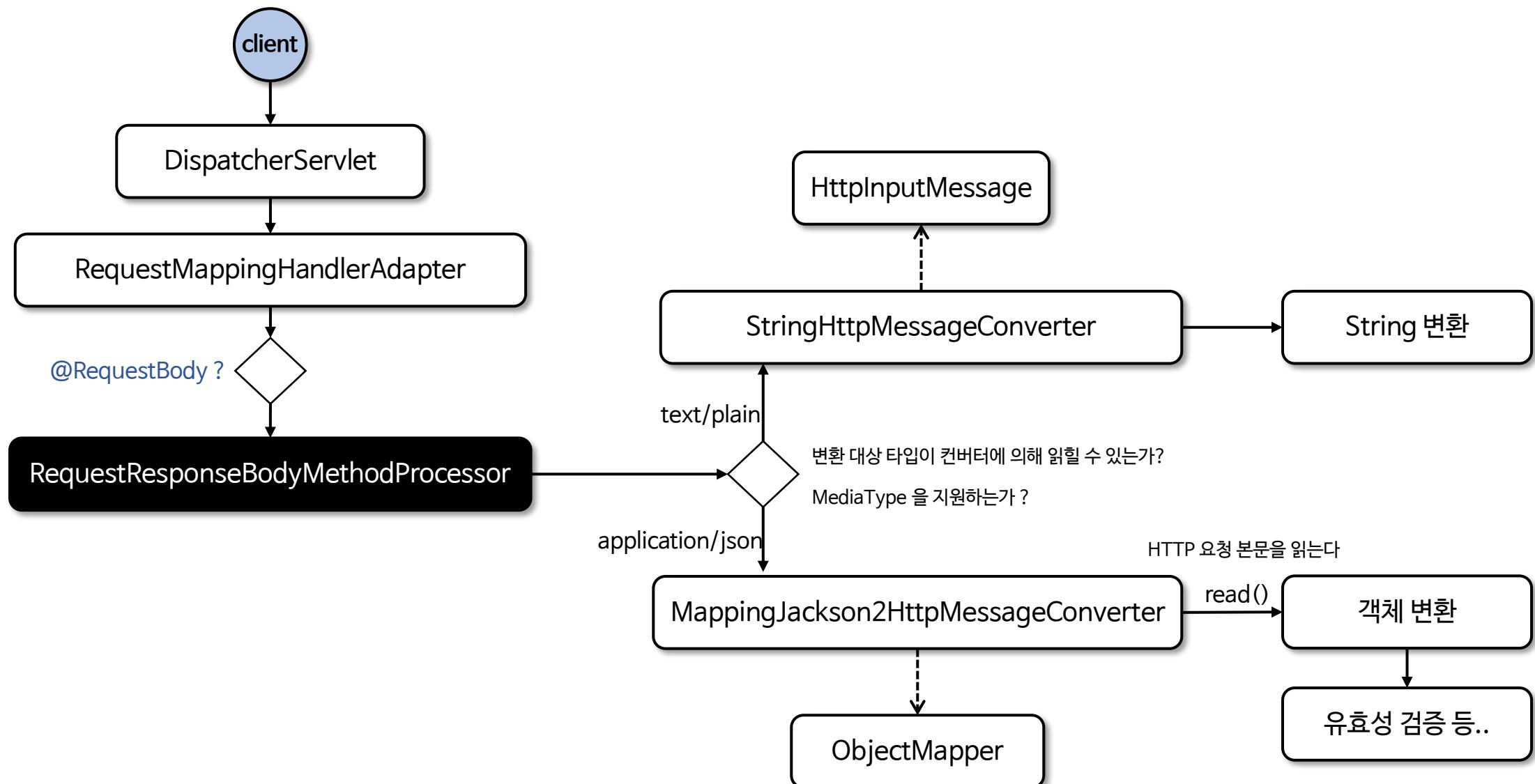
application/x-www-form-urlencoded 데이터 받기

주로 @RequestParam 이나 @ModelAttribute 를 사용하지만
일부 객체에 한해 @RequestBody 도 가능함

```
@PostMapping(consumes = {MediaType.APPLICATION_JSON_VALUE, MediaType.TEXT_PLAIN_VALUE})
public ResponseEntity<String> handleMixedContent(@RequestBody Object data) {
    if (data instanceof String) return ResponseEntity.ok("Received plain text: " + data);
} else {
    UserDto userDto = (UserDto) data;
    return ResponseEntity.ok("User: " + userDto.getName());
}
```

Mixed Content-Type 지원 (application/json + text/plain)

✓ 흐름도

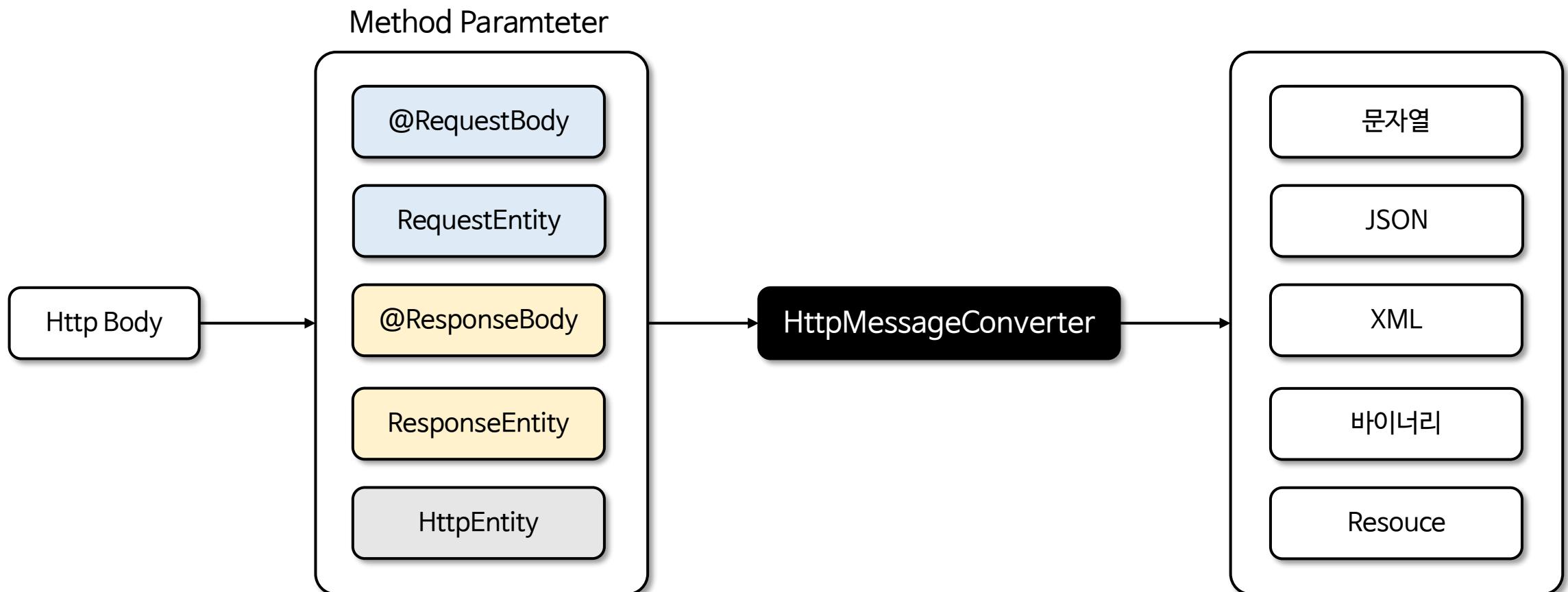


HttpMessageConverter

<https://github.com/onjsdnjs/spring-mvc-master/tree/HttpMessageConverter>

✓ 개요

- `HttpMessageConverter`는 HTTP 요청과 응답의 바디(body) 내용을 객체로 변환하고 객체를 HTTP 메시지로 변환하는 데 사용되는 인터페이스이다
- `HttpMessageConverter`는 클라이언트와 서버 간의 데이터를 직렬화/역직렬화하는 기능을 담당하며 주로 JSON, XML, Plain Text와 같은 다양한 데이터 포맷을 지원한다
- `HttpMessageConverter`는 주로 Rest API 통신에서 사용된다



✓ API 구조

① HttpMessageConverter<T>

Ⓜ️ ⌂ canRead (Class<?>, MediaType ?)	boolean
Ⓜ️ ⌂ canWrite (Class<?>, MediaType ?)	boolean
Ⓜ️ ⌂ read (Class<T>, HttpInputMessage)	T
Ⓜ️ ⌂ getSupportedMediaTypes (Class<?>)	List<MediaType >
Ⓜ️ ⌂ getSupportedMediaTypes ()	List<MediaType >
Ⓣ️ ⌂ write (T, MediaType ?, HttpOutputMessage)	void

- 주어진 객체 타입과 미디어 타입(Content-Type)을 읽을 수 있는지 확인한다
- 주어진 객체 타입과 미디어 타입(Content-Type)에 데이터를 쓸 수 있는지 확인한다
- HTTP 요청 데이터를 Java 객체로 변환한다
- 특정 클래스에 대해 이 컨버터가 지원하는 미디어 타입의 목록을 반환한다
- 이 컨버터가 지원하는 미디어 타입(예: JSON, XML 등)의 목록을 반환한다
- Java 객체를 HTTP 응답 데이터로 변환한다

✓ 초기화 시 기본 9개 생성

```
oo this.converters = {Collections$UnmodifiableRandomAccessList@68
>   0 = {ByteArrayHttpMessageConverter@6852}
>   1 = {StringHttpMessageConverter@6833}
>   2 = {StringHttpMessageConverter@6853}
>   3 = {ResourceHttpMessageConverter@6854}
>   4 = {ResourceRegionHttpMessageConverter@6855}
>   5 = {AllEncompassingFormHttpMessageConverter@6846}
>   6 = {MappingJackson2HttpMessageConverter@6834}
>   7 = {MappingJackson2HttpMessageConverter@6850}
>   8 = {Jaxb2RootElementHttpMessageConverter@6856}
```

- 위로부터 아래로 한 개씩 호출되어 실행된다
- 실행할 객체가 없을 경우 예외가 발생한다 (HttpMessageNotReadableException)

✓ HttpMessageConverter 요청 처리 흐름

1. 클라이언트의 Content-Type 헤더

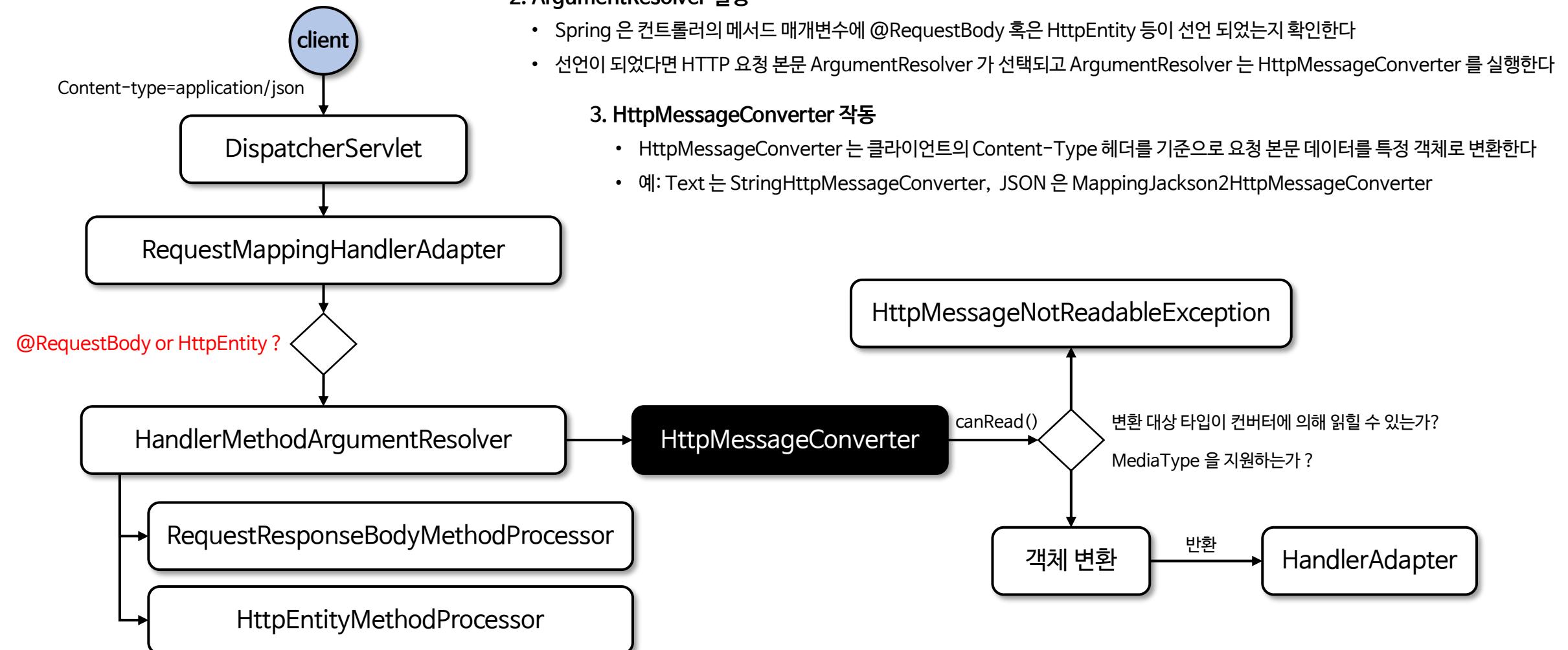
- 클라이언트가 서버로 데이터를 전송한다. 이때 HTTP 헤더에 Content-Type 을 포함하여 서버에 데이터 형식을 알린다
- 예: Content-type=application/json, application/xml, text/plain

2. ArgumentResolver 실행

- Spring 은 컨트롤러의 메서드 매개변수에 @RequestBody 혹은 HttpEntity 등이 선언 되었는지 확인한다
- 선언이 되었다면 HTTP 요청 본문 ArgumentResolver 가 선택되고 ArgumentResolver 는 HttpMessageConverter 를 실행한다

3. HttpMessageConverter 작동

- HttpMessageConverter 는 클라이언트의 Content-Type 헤더를 기준으로 요청 본문 데이터를 특정 객체로 변환한다
- 예: Text 는 StringHttpMessageConverter, JSON 은 MappingJackson2HttpMessageConverter



✓ HttpMessageConverter 응답 처리 흐름

1. 클라이언트의 Accept 헤더

- 클라이언트는 Accept 헤더를 통해 서버가 어떤 형식의 데이터를 반환해야 하는지 명시한다
- 예: Accept: application/json, Accept: application/xml

2. ReturnValueHandler 실행

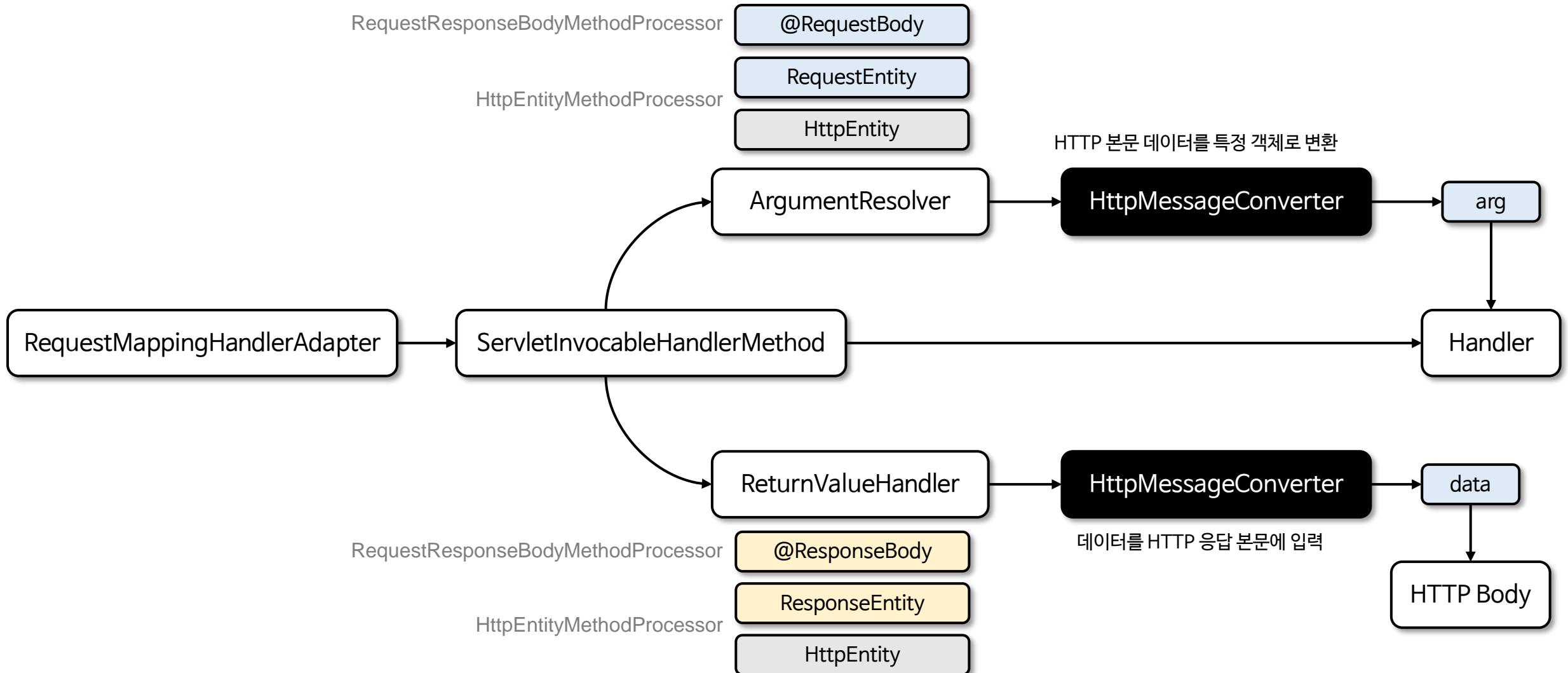
- Spring은 컨트롤러의 반환 타입에 @ResponseBody 또는 ResponseEntity가 선언되었는지 확인한다
- 선언이 되었다면 HTTP 응답 본문 ReturnValueHandler가 선택되고 ReturnValueHandler는 HttpMessageConverter를 실행한다

3. HttpMessageConverter 작동

- HttpMessageConverter는 클라이언트의 Accept 헤더를 기준으로 데이터를 응답 본문에 기록한다
- 예: Text는 StringHttpMessageConverter, JSON은 MappingJackson2HttpMessageConverter



✓ HttpMessageConverter 요청/응답 구조



✓ **HttpMessageConverter 주요 구현체**

1. **ByteArrayHttpMessageConverter**

- application/octet-stream과 같은 바이너리 데이터를 처리하며 주로 파일 전송에 사용된다

2. **StringHttpMessageConverter**

- text/plain과 같은 문자열 데이터를 String 객체로 변환하거나 String 객체를 text/plain 형식으로 변환하여 HTTP 본문에 넣는다

3. **ResourceHttpMessageConverter**

- Resource 타입의 데이터를 HTTP 요청과 응답으로 변환하거나 처리하는 데 사용된다

4. **MappingJackson2HttpMessageConverter**

- application/json 형식의 데이터를 파싱하여 Java 객체를 JSON으로 변환하거나 JSON을 Java 객체로 변환한다

5. **FormHttpMessageConverter**

- MultiValueMap + application/x-www-form-urlencoded 형식의 데이터를 파싱하여 MultiValueMap 형태로 변환한다

✓ HttpMessageConverter 가 작동하지 않는 요청

1. GET 요청과 같은 본문이 없는 요청

- GET, DELETE와 같은 HTTP 메서드는 일반적으로 본문을 포함하지 않으므로 HttpMessageConverter가 작동하지 않는다

2. Content-Type 헤더가 지원되지 않는 요청

- POST, PUT 등의 본문이 포함된 HTTP 요청이라도 Content-Type 헤더가 지원되지 않는 미디어 타입일 경우 HttpMessageConverter가 작동하지 않는다

3. @RequestParam, @ModelAttribute 를 사용하는 경우

- @RequestParam, @ModelAttribute 와 같은 어노테이션을 사용하여 쿼리 파라미터나 application/x-www-form-urlencoded 형식의 폼 데이터를 처리하는 경우 HttpMessageConverter 가 필요하지 않다

4. 파일 업로드 요청 중 @RequestPart 또는 MultipartFile 을 사용한 경우

- multipart/form-data 요청에서 파일을 업로드할 때, MultipartFile 이나 @RequestPart 를 사용하면 HttpMessageConverter 가 작동하지 않으며 이 경우에는 MultipartResolver 가 요청을 처리한다

5. 컨트롤러에서 단순 문자열(String) 반환 시 @ResponseBody나 @RestController 가 없는 경우

- 컨트롤러 메서드가 String 을 반환하지만 @ResponseBody나 @RestController가 없는 경우 반환된 String은 뷰 이름으로 간주되며 이 경우에는 ViewResolver 가 요청을 처리한다

`@RequestHeader`
`@RequestAttribute`
`@CookieValue`

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestHeader-@RequestAttribute-@CookieValue>

✓ @RequestHeader

- 클라이언트의 요청 헤더를 컨트롤러의 메서드 인자에 바인딩 하기 위해 @RequestHeader 애노테이션을 사용할 수 있다
- RequestHeaderMethodArgumentResolver 클래스가 사용된다

✓ 기본 구현

@ RequestHeader	
Ⓜ️ ⌂ value()	String
Ⓜ️ ⌂ defaultValue()	String
Ⓜ️ ⌂ name()	String
Ⓜ️ ⌂ required()	boolean

Host	localhost:8080
Accept	text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language	fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding	gzip,deflate
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive	300

```
@GetMapping("/header")
public void handle( @RequestHeader("Accept-Encoding") String encoding, @RequestHeader("Keep-Alive") long keepAlive) {
    System.out.println("Accept-Encoding : " + encoding);      Accept-Encoding : gzip,deflate
    System.out.println("Keep-Alive: " + keepAlive);           Keep-Alive: 300
}
```

- 특정한 헤더를 바인딩해야 할 때 @RequestHeader("헤더명") 이라고 정의하면 된다
- 바인딩이 필요한지 여부와 기본 값을 주고자 할 때 required 와 defaultValue 속성을 지정하면 된다

```
@RequestHeader("Accept-Encoding", required=false, defaultValue="none")
```

✓ Map 을 파라미터로 사용

```
GET /headers  
Host: localhost:8080  
Accept: text/html  
User-Agent: Mozilla/5.0
```

```
@GetMapping("/headers")  
public void handleHeaders(@RequestHeader Map<String, String> headers) {    모든 헤더가 Map으로 전달된다  
    headers.forEach((key, value) -> {  
        System.out.println(key + ": " + value);  
    });  
}
```

host: localhost:8080
accept: text/html
user-agent: Mozilla/5.0

✓ MultiValueMap 을 파라미터로 사용 - 하나의 키에 여러 값을 매핑할 수 있는 자료구조(구현체로 HttpHeaders 가 있다)

```
GET /headers  
Host: localhost:8080  
Accept: text/html,application/xhtml+xml  
User-Agent: Mozilla/5.0
```

```
@GetMapping("/headers")  
public void handleHeaders(@RequestHeader MultiValueMap<String, String> headers) {  
    headers.forEach((key, values) -> {  
        System.out.println(key + ": " + values);  
    });  
}
```

host: [localhost:8080]
accept: [text/html, application/xhtml+xml] // 한 개의 키에 두개 값 출력
user-agent: [Mozilla/5.0]

✓ 쉼표로 구분된 문자열을 배열이나 List 타입으로 변환

```
GET /accept
```

```
Host: localhost:8080
```

```
Accept: text/html,application/xhtml+xml,application/xml
```

```
@GetMapping("/accept")  
public void handleAccept(@RequestHeader("Accept") List<String> acceptHeaders) {  
    System.out.println("Accept Headers: " + acceptHeaders);  ["text/html", "application/xhtml+xml", "application/xml"] 과 같이 변환된다  
}
```

✓ @RequestAttribute

- HTTP 요청 속성(request attribute)을 메서드 파라미터에 바인딩할 때 사용하는 어노테이션으로서 주로 필터나 인터셉터에서 설정한 값을 컨트롤러 메서드에서 사용할 때 유용하다
- RequestAttributeMethodArgumentResolver 클래스가 사용 된다

```
@WebFilter("/example")
public class MyAttributeFilter implements Filter {

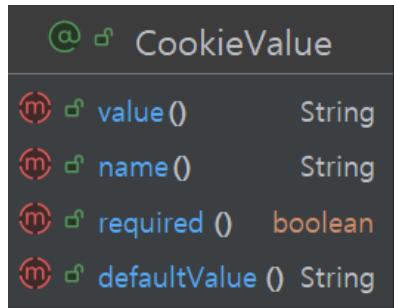
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        request.setAttribute("myAttribute", "This is a request attribute");
        chain.doFilter(request, response);
    }

}
```

```
@GetMapping("/example")
public String handleRequest(@RequestAttribute("myAttribute") String myAttribute) {
    return "Attribute Value: " + myAttribute; // Attribute Value : This is a request attribute
}
```

✓ @CookieValue

- HTTP 요청의 쿠키 값을 메서드 파라미터에 바인딩할 때 사용하는 애노테이션으로서 클라이언트에서 전송한 쿠키 값을 쉽게 받아 처리할 수 있다
- @CookieValue는 특정 쿠키의 값을 메서드 파라미터로 전달하며 기본값을 설정하거나 쿠키가 존재하지 않을 때 예외를 처리할 수 있는 옵션을 제공한다
- ServletCookieValueMethodArgumentResolver 클래스가 사용 된다



```
@GetMapping("/cookie")
public String getCookie(@CookieValue(value = "userSession", defaultValue = "defaultSession") String session) {
    return "Session ID: " + session;
}
```

클라이언트가 userSession 쿠키를 전송한 경우

```
GET /cookie
Cookie: userSession=abc123
```

Session ID: abc123

클라이언트가 userSession 쿠키를 전송하지 않은 경우

```
GET /cookie
```

Session ID: defaultSession

Model

<https://github.com/onjsdnjs/spring-mvc-master/tree/Model>

✓ 개요

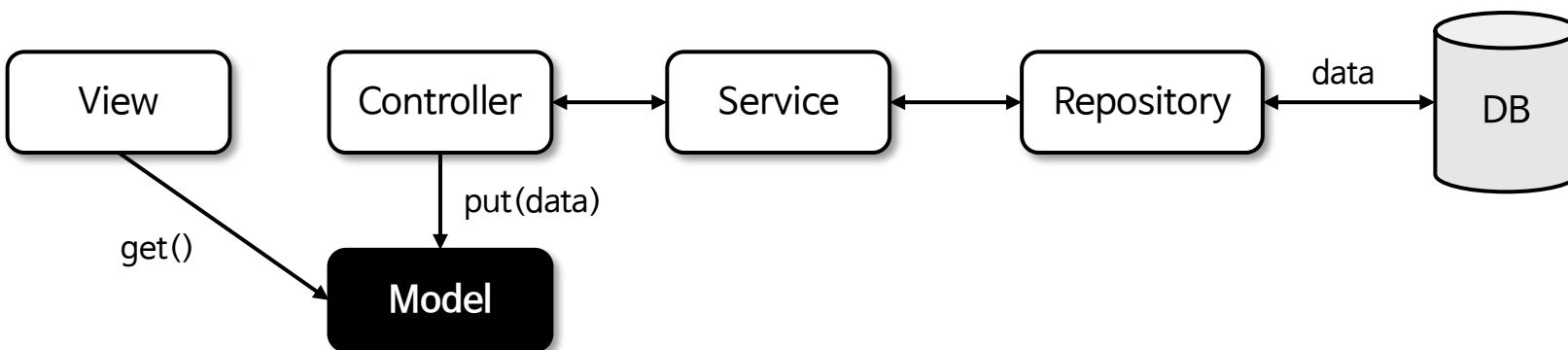
- Model은 컨트롤러와 뷰 사이의 데이터를 전달하는 역할을 하며 컨트롤러에서 데이터를 Model 객체에 추가하면 그 데이터는 뷰에서 접근할 수 있게 된다
- Model 인터페이스는 주로 HTML 렌더링을 위한 데이터 보관소 역할을 하며 Map과 유사한 방식으로 동작한다
- 내부적으로 ModelMethodProcessor 클래스가 사용된다

✓ 구조

Model	
Ⓜ️ ↗ addAllAttributes(Collection<?>)	Model
Ⓜ️ ↗ addAttribute(Object)	Model
Ⓜ️ ↗ getAttribute(String)	Object?
Ⓜ️ ↗ addAllAttributes(Map<String, ?>)	Model
Ⓜ️ ↗ mergeAttributes(Map<String, ?>)	Model
Ⓜ️ ↗ containsAttribute(String)	boolean
Ⓜ️ ↗ asMap()	Map<String, Object>
Ⓜ️ ↗ addAttribute(String, Object?)	Model

주요 메서드

- addAttribute(String attributeName, Object attributeValue) : - 모델에 데이터를 추가할 때 사용한다
- addAllAttributes(Map<String, ?> attributes) - 여러 개의 속성을 한 번에 추가할 수 있다
- asMap() - 모델 데이터를 Map으로 변환하여 반환한다



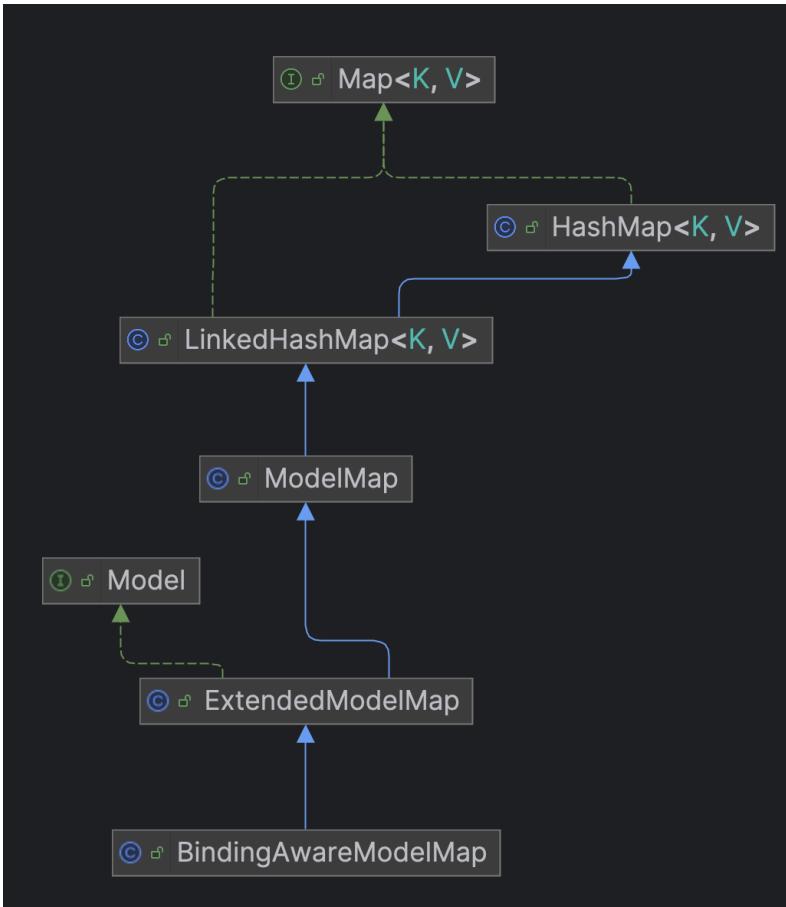
✓ 기본 구현

```
@Controller
public class GreetingController {
    @GetMapping("/greeting")
    public String greeting(Model model) {
        model.addAttribute("name", "leaven");
        model.addAttribute("message", "Welcome");
        return "greeting";
    }
    @GetMapping("/greeting2")
    public String greeting2(Model model) {
        Map<String, Object> data = new HashMap<>();
        data.put("name", "leaven");
        data.put("message", "Welcome");
        model.addAllAttributes(data);
        return "greeting";
    }
    @GetMapping("/greeting3")
    public String greeting3(Model model) {
        User user = new User("leaven", "Welcome to the application!");
        model.addAttribute("user", user);
        return "greeting";
    }
}
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Greeting Page</title>
</head>
<body>
    <h1>Greeting Page</h1>
    <p th:text="${name}"></p>
    <p th:text="${message}"></p>
    <p th:text="${user.name}"></p>
    <p th:text="${user.message}"></p>
</body>
</html>
```

✓ BindingAwareModelMap

- BindingAwareModelMap 은 Model 구현체로서 @ModelAttribute 로 바인딩된 객체를 가지고 있으며 바인딩 결과를 저장하는 BindingResult 를 생성하고 관리한다



```
@GetMapping={"/index"}  
public String index(@ModelAttribute("accountDto") AccountDto accountDto, BindingAwareModelMap model){  
    Object accountDto1 = model.get("accountDto");  
    Object bindingResult = model.get("org.springframework.validation.BindingResult.accountDto");  
    return "index";  
}
```

```
model = {BindingAwareModelMap@7603} size = 2  
  "accountDto" -> {AccountDto@7602}  
    key = "accountDto"  
    value = {AccountDto@7602}  
  "org.springframework.validation.BindingResult.accountDto" -> {BeanPropertyBindingResult@7612} "org.springframework.validation.BeanPropertyBindingResult: 0 errors"  
    key = "org.springframework.validation.BindingResult.accountDto"  
    value = {BeanPropertyBindingResult@7612} "org.springframework.validation.BeanPropertyBindingResult: 0 errors"
```

@SessionAttributes

<https://github.com/onjsdnjs/spring-mvc-master/tree/@SessionAttributes>

✓ 개요

- @SessionAttributes 는 세션(Session)에 속성 값을 저장하고 그 값을 다른 요청에서도 사용할 수 있도록 하기 위해 사용되는 어노테이션이다
- @SessionAttributes 는 컨트롤러 클래스 레벨에 선언되며 특정 모델 속성 이름을 지정하면 세션에 자동으로 저장된다
- @SessionAttributes 는 모델에 저장된 객체를 세션에 저장하는 역할과 세션에 저장된 객체를 모델에 저장하는 역할을 한다

✓ @SessionAttributes 기본 - 세션 저장

```
@Controller  
@SessionAttributes("user") ②  
public class UserController {  
    @GetMapping("/users")  
    public String users (Model model) {  
        model.addAttribute("user",new User("springmvc","a@a.com")); ①  
        return "redirect:/getUser";  
    }  
    @GetMapping("/getUser")  
    public String getUser(@ModelAttribute("user") User user) { ③  
        return "redirect:/getUser2";  
    }  
}
```

- /users 으로 요청하면 1->2->3 번 순서로 처리된다
 - @SessionAttributes("user") 가 정의되어 있을 경우 1차적으로 세션에서 User 객체를 찾고 존재하지 않으면 다음으로 수행한다
 - users() 이 호출되고 Model 에 "user" 이름으로 객체를 저장한다
 - users() 메서드 실행 이후 @SessionAttributes("user") 의 속성명과 모델에 저장한 속성명("user")이 동일한 경우 세션에 객체를 저장한다
 - getUser() 메서드를 실행하면 2번 과정을 통해 세션에 저장된 User 객체가 Model 에 저장되며 그 Model로부터 User 객체를 꺼내어 와서 매개변수로 전달한다

✓ @SessionAttributes 기본 - @ModelAttribute 와 함께 사용하는 경우

```
@Controller  
@SessionAttributes("user") 1  
public class UserController2 {  
  
    @GetMapping("/users")  
    public String users(Model model) {  
        // model.addAttribute ("user",new User("springmvc","a@a.com"));  
        return "redirect:/getUser";  
    }  
    @GetMapping("/getUser")  
    public String getUser(@ModelAttribute("user") User user, Model model) { 2  
        return "redirect:/getUser2";  
    }  
}
```

- @ModelAttribute 와 함께 사용하는 경우 다음과 같이 처리된다
 - @ModelAttribute 와 @SessionAttributes 의 속성명을 체크해서 서로 동일한 속성명을 가지고 있는지 체크한다
 - 속성이 동일할 경우 먼저 Model 에 User 객체가 존재하는지 체크하고 없으면 다음으로 수행한다
 - 마지막으로 세션에서 객체를 찾는데 세션에도 존재하지 않으면 '*Expected session attribute 객체이름*'와 같은 오류가 난다.
- 이것을 해결하기 위해서는 아래 메서드처럼 Model 에 객체를 직접 추가 하든지 아니면 메서드 위에 @ModelAttribute 를 추가해서 Model 에 담긴 객체를 세션에 연결해 주도록 해야 한다

```
@ModelAttribute  
public User addUser() {  
    return new User("springmvc", "springmvc@gmail.com");  
}
```

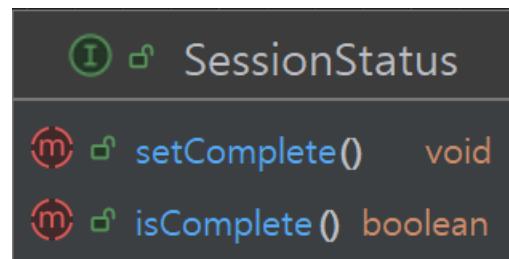
@SessionAttributes 는 세션에 있는 객체를 Model 에 연결해 주는 역할을 한다

✓ @SessionAttributes 기본 - 세션 초기화

- 세션 초기화는 서버에서 특정 세션에 저장된 데이터를 삭제하고 세션 상태를 초기화하는 과정을 말한다

✓ SessionStatus

- Spring에서는 SessionStatus 사용하여 세션 데이터를 제거할 수 있으며 세션 초기화 후에는 보통 다른 페이지로 리다이렉트하여 새로운 세션이 시작될 준비를 한다



- 현재 핸들러의 세션 처리가 완료되었음을 표시하여 세션 속성들을 초기화 한다
- 현재 핸들러의 세션 처리가 완료되었는지 여부를 반환한다

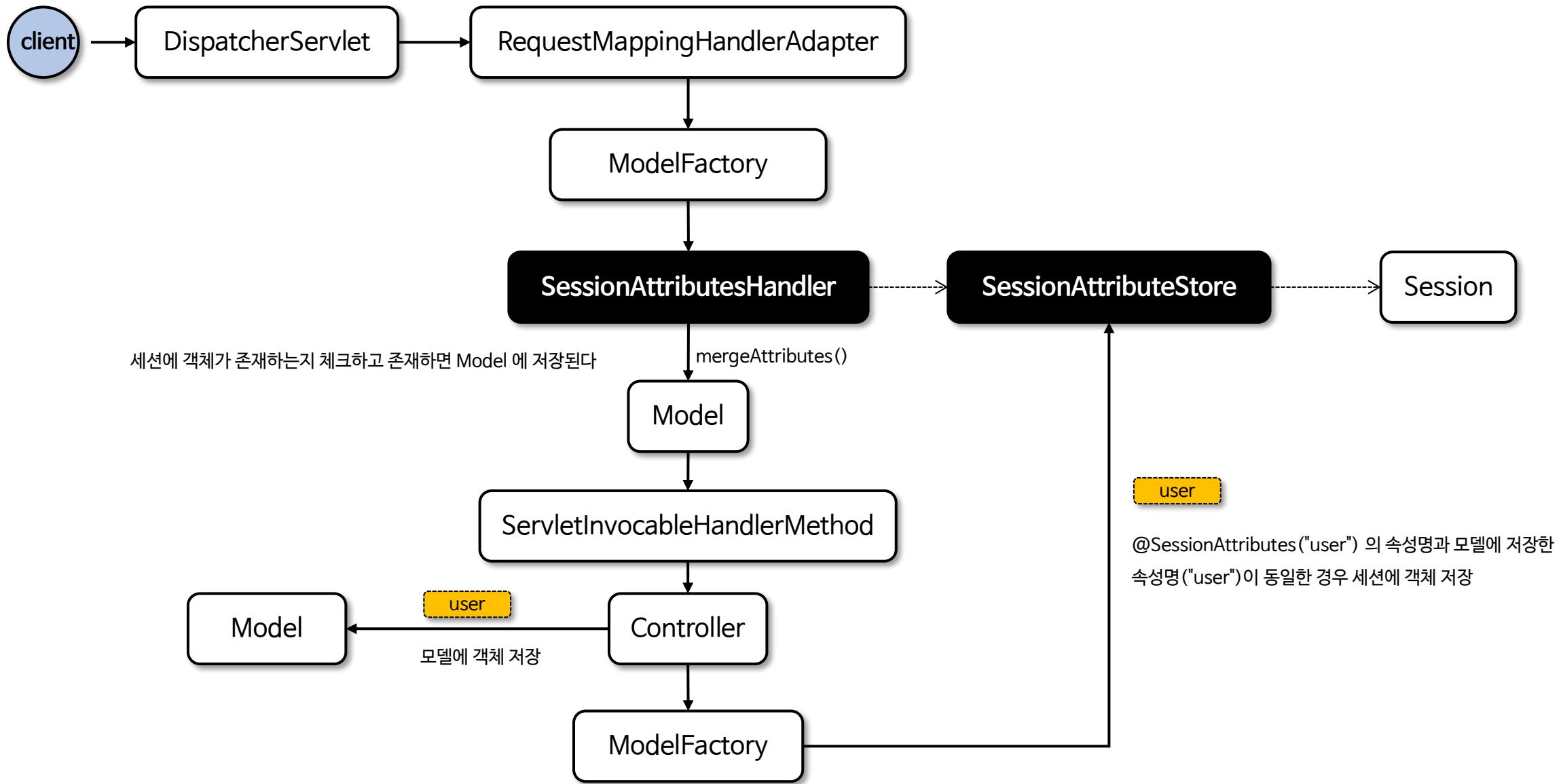
```
@Controller
@SessionAttributes("user") // 'user' 속성을 세션에 저장
public class UserController {

    // 세션 초기화 메서드

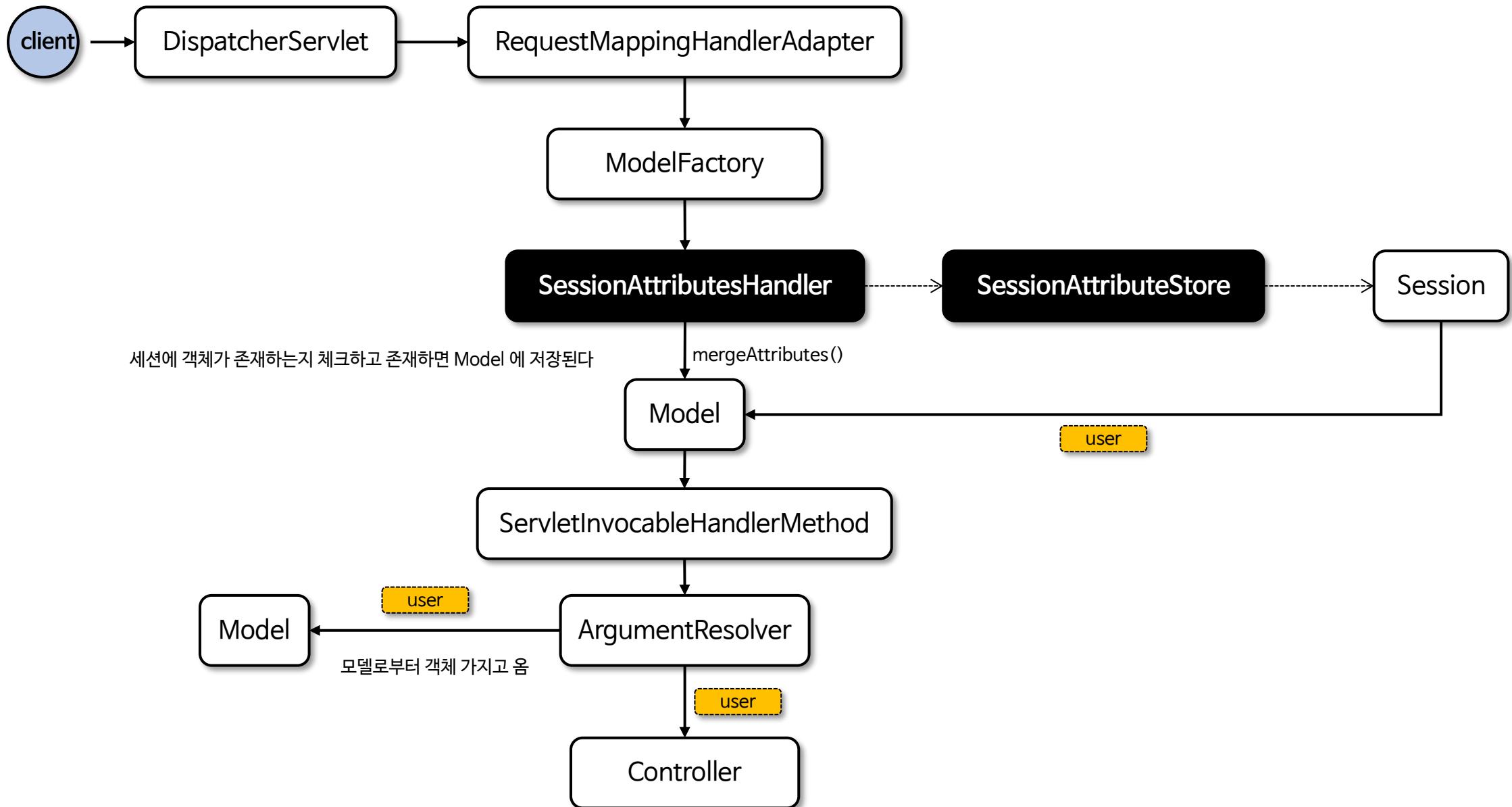
    @PostMapping("/clearSession")
    public String clearSession(SessionStatus sessionStatus) {
        sessionStatus.setComplete(); // 'user' 속성만 세션에서 제거
        return "redirect:/userForm";
    }
}
```

- 세션 초기화 범위는 컨트롤러에서 `@SessionAttributes`로 선언된 세션 속성들에 한정된다
- `SessionStatus.setComplete()`는 해당 컨트롤러에서 관리하는 특정 세션 속성들만을 초기화(삭제)하고, 다른 곳에서 저장된 세션 데이터나 전체 세션 자체를 삭제하지는 않는다
- 세션 전체 무효화를 원한다면 `HttpSession.invalidate()`를 사용할 수 있다

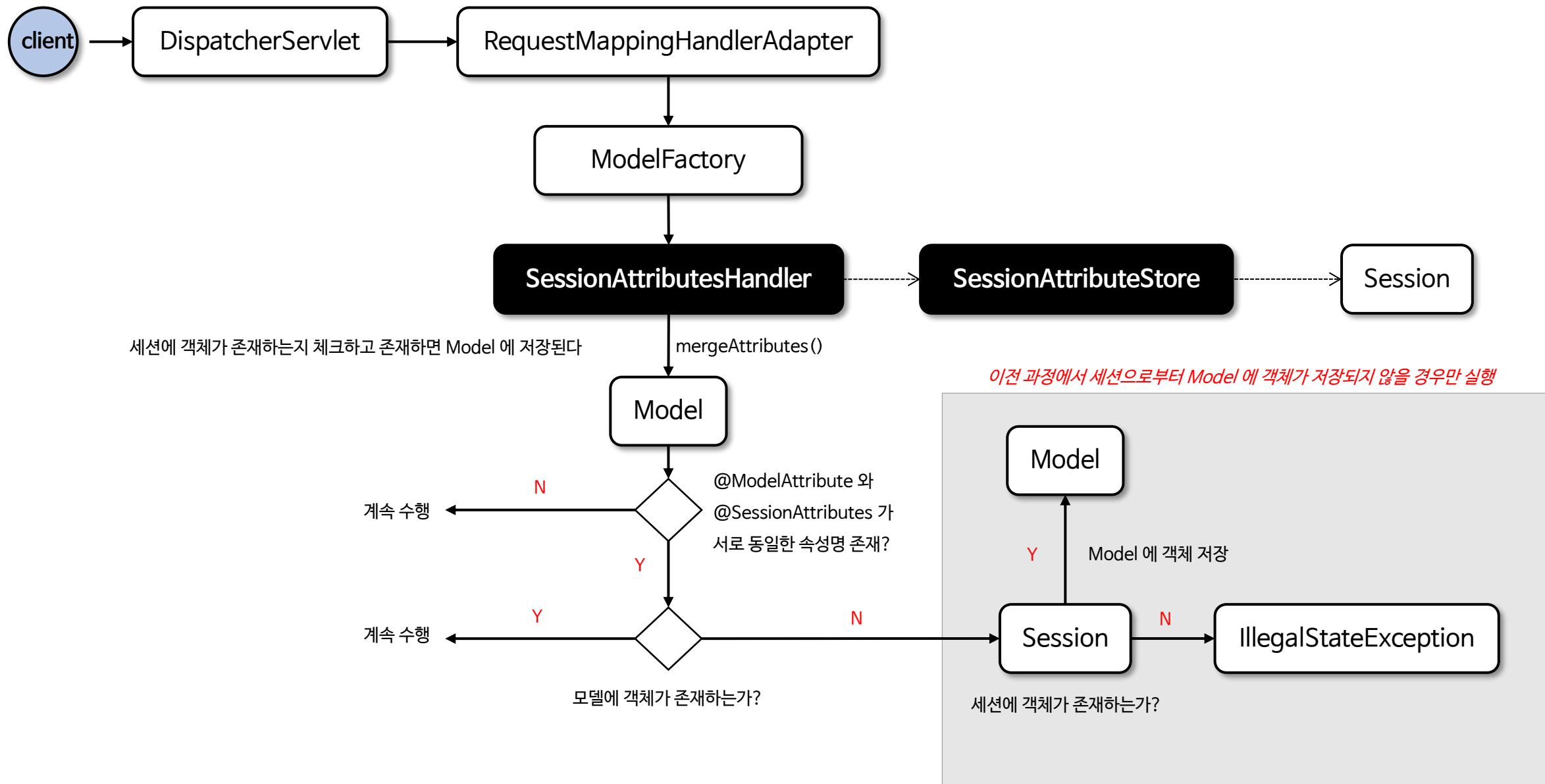
✓ 흐름도 -1



✓ 흐름도 -2



✓ 흐름도 - 3

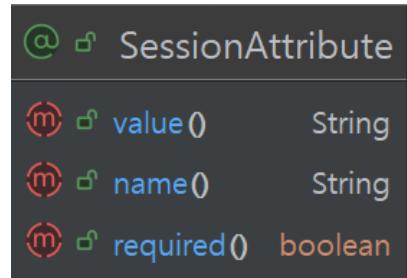


@SessionAttribute

<https://github.com/onjsdnjs/spring-mvc-master/tree/@SessionAttribute>

✓ 개요

- `@SessionAttribute`는 세션에 저장된 특정 속성을 메서드 파라미터로 가져오기 위해 사용되는 어노테이션이다
- 세션에 저장된 속성 값을 컨트롤러의 메서드에서 직접 접근할 수 있도록 해주며 전역적으로 관리되는 세션 속성에 접근할 때 유용하게 사용할 수 있다



- 세션에 저장된 속성을 메서드 파라미터로 쉽게 접근할 수 있다
- 속성이 존재하지 않을 때의 동작을 제어할 수 있다

✓ @SessionAttribute vs HttpSession

```
@GetMapping("/getUser")
public String getUser(@SessionAttribute(name = "user", required = false) User user) {
    if (user != null) {
        System.out.println("User from session: " + user.getName());
    } else {
        System.out.println("No user in session");
    }
    return "userPage"; // 사용자 정보를 보여주는 페이지로 이동
}
```

```
@GetMapping("/getUser")
public String getUser(HttpSession session) {
    User user = (User) session.getAttribute("user"); // 객체를 얻고 타입 변환
    if (user != null) {
        System.out.println("User from session: " + user.getName());
    } else System.out.println("No user in session");

    return "userPage"; // 사용자 정보를 보여주는 페이지로 이동
}
```

- `@SessionAttribute(name = "user", required = false)` – 세션에 저장된 user 속성을 읽어온다
- `required = false`로 설정하면 세션에 user 속성이 없을 경우 null이 반환되고 있으면 해당 속성 값에 접근해 사용자 이름을 출력한다
- `required = true`(기본값)로 설정하면 해당 속성이 세션에 반드시 있어야 하며 없으면 예외가 발생한다

✓ @SessionAttributes + @SessionAttribute

- 컨트롤러에서 @SessionAttributes 와 @SessionAttribute 를 함께 사용하여 세션을 활용하는 예제를 살펴보자

```
@Controller  
@SessionAttributes("user") // "user"를 세션에 저장  
public class SessionController {  
    @GetMapping("/setSession")  
    public String setSession(Model model) {  
        User user = new User("leaven", "Welcome to the session!");  
        model.addAttribute("user", user); // "user"를 모델과 세션에 저장  
        return "session";  
    }  
    @GetMapping("/getSession") // 세션 데이터를 활용  
    public String getSession(@SessionAttribute(name = "user") User user, Model model) {  
        model.addAttribute("message", "User in session: " + user.getName());  
        return "session";  
    }  
  
    @GetMapping("/clearSession")  
    public String clearSession(SessionStatus sessionStatus) {  
        sessionStatus.setComplete(); // 세션에서 "user" 데이터 삭제  
        return "session";  
    }  
}
```

session.html

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<body>  
    <h1>Session Example</h1>  
    <p th:text="${user.name}"></p>  
    <p th:text="${user.message}"></p>  
    <a href="/clearSession">Clear Session</a>  
</body>  
</html>
```

1) /setSession 요청

- User 객체를 생성하여 세션에 저장한다
- @SessionAttributes("user")로 인해 "user"가 세션에 저장된다

2) /getSession 요청

- @SessionAttribute(name = "user")를 통해 세션에 저장된 "user" 객체를 가져온다
- Thymeleaf에서 user.name과 user.message를 출력한다

3) /clearSession 요청

- SessionStatus.setComplete()를 호출하여 "user" 세션 데이터를 제거한다

커스텀 HandlerMethodArgumentResolver 구현

<https://github.com/onjsdnjs/spring-mvc-master/tree/CustomArgumentResolver>

✓ 개요

- `@CurrentUser`라는 어노테이션을 붙인 컨트롤러 파라미터에 로그인 사용자 정보를 자동으로 주입해 주는 기능을 만들고자 한다

✓ 클래스 정의

1. AuthFilter

- 필터 단계에서 인증 로직을 수행하고, 인증된 사용자 정보를 `request.setAttribute`에 저장한다.

2. FilterConfig

- Spring Boot에서 `AuthFilter`를 등록하기 위해 `FilterRegistrationBean`을 설정하는 구성 클래스이다.

3. `@CurrentUser`

- 컨트롤러 메서드 파라미터에 붙여, 인증된 사용자 객체를 주입받기 위한 커스텀 애노테이션이다.

4. `CurrentUserArgumentResolver`

- 컨트롤러 파라미터에 `@CurrentUser`가 붙어 있으면, `request`에서 `LoginUser` 정보를 찾아 파라미터에 주입한다.

5. `WebConfig`

- 스프링 MVC 설정을 담당하며, `CurrentUserArgumentResolver`를 `addArgumentResolvers`를 통해 등록한다.

6. `LoginUser`

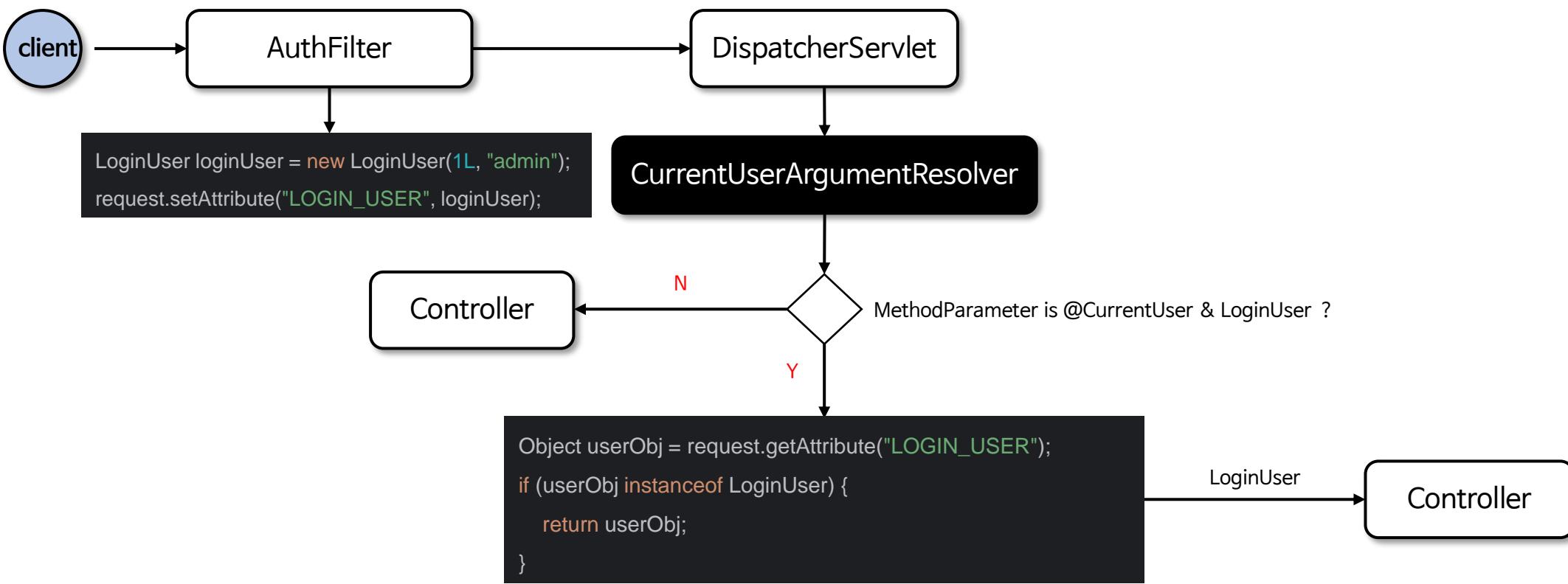
- 인증된 사용자의 ID, 이름 등 필요한 정보를 담는 DTO(또는 VO) 역할을 한다.

7. `DemoController`

- `@CurrentUser LoginUser` 파라미터를 통해 자동 주입된 사용자 정보를 활용해 로직을 처리하는 예시 컨트롤러이다.

✓ 처리 과정

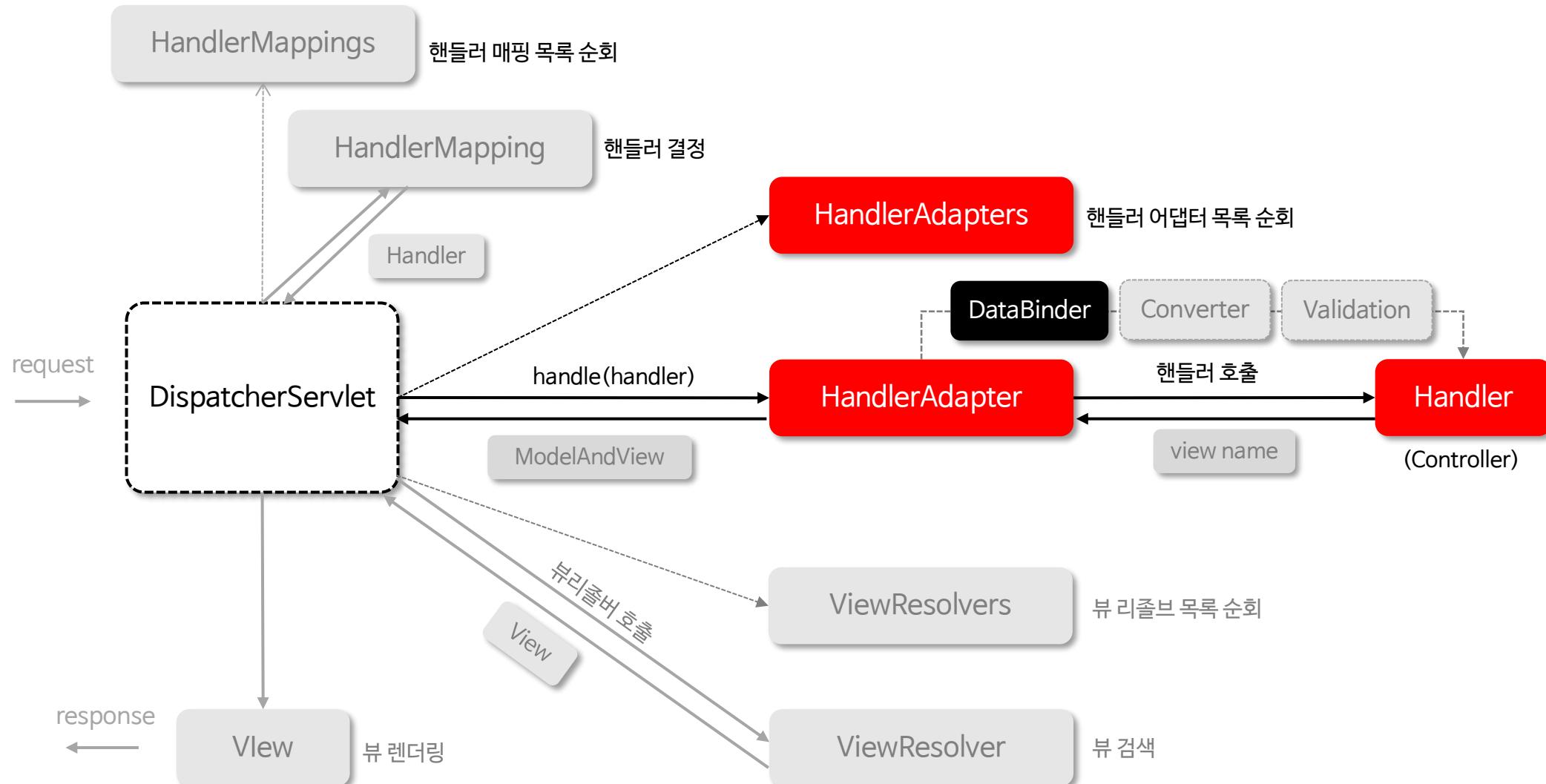
1. 클라이언트가 HTTP 요청을 보낸다.
2. 등록된 필터(AuthFilter)가 먼저 실행되어 인증 정보를 확인한다.
3. 인증 성공 시 AuthFilter에서 LoginUser 객체를 생성해 request.setAttribute로 저장한다.
4. 필터 체인 통과 후, 스프링의 DispatcherServlet이 컨트롤러를 찾는다.
5. 스프링은 컨트롤러 파라미터를 분석해 @CurrentUser와 LoginUser 타입을 확인한다.
6. CurrentUserArgumentResolver가 request.getAttribute("LOGIN_USER")를 꺼내 파라미터에 주입한다.
7. 컨트롤러 메서드는 주입된 LoginUser 정보를 사용해 응답을 처리한다.



바인딩 - DataBinder

<https://github.com/onjsdnjs/spring-mvc-master/tree/DataBinder>

✓ 스프링 MVC 아키텍처



✓ 개요

- HTTP 요청에 대한 데이터 바인딩 방식은 크게 두 가지로 구분할 수 있는데 바로 **쿼리 파라미터** 및 **폼 데이터 바인딩**과 **HTTP 본문 데이터 바인딩**이다

✓ 데이터 바인딩 분류

1. 쿼리 파라미터 및 폼 데이터 바인딩

- 기본형, 문자열, 래퍼 클래스
 - @RequestParam 적용하여 RequestParamMethodArgumentResolver 가 요청 처리
 - @PathVariable 적용하여 PathVariableMethodArgumentResolver 가 적용 처리
 - 객체 : @ModelAttribute 적용하여 ServletModelAttributeMethodProcessor 가 요청 처리
- **DataBinder는 여기에 관여하는 전략이다 !!**

2. HTTP 본문 바인딩

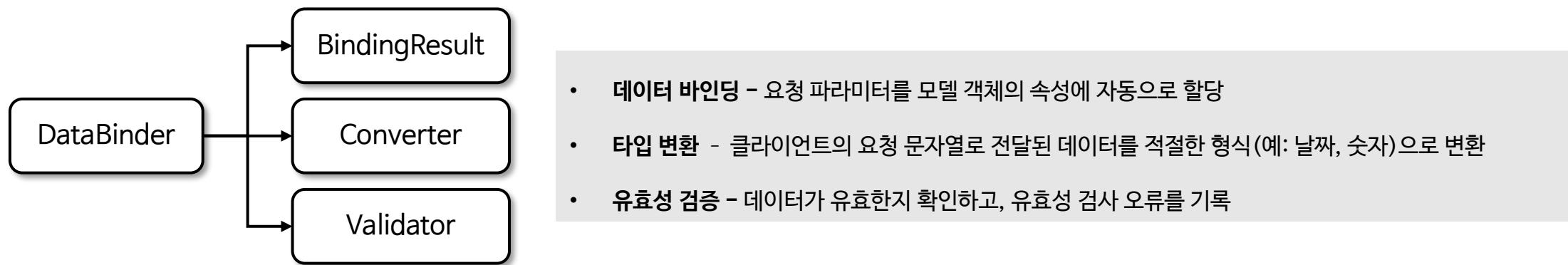
- Text, JSON : @RequestBody 적용하여 RequestResponseBodyMethodProcessor 와 HttpMessageConverter 가 요청 처리

✓ DataBinder vs HttpMessageConverter

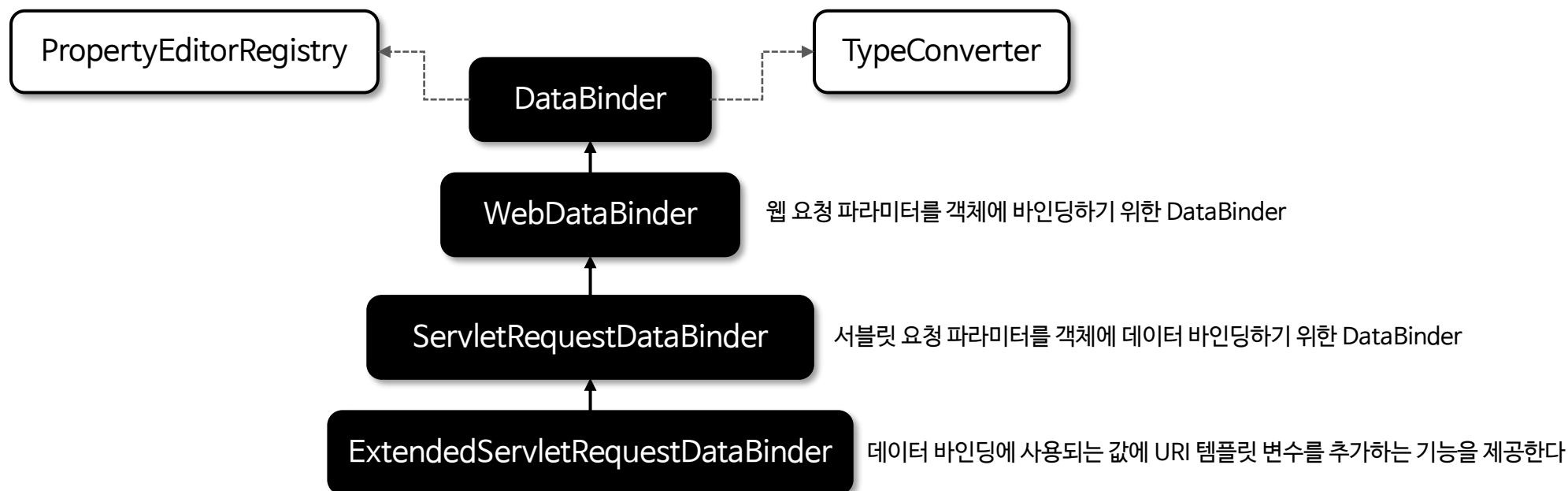
DataBinder	HttpMessageConverter
웹 요청 파라미터를 객체에 바인딩	HTTP 요청/응답 본문을 객체로 변환하거나 객체를 HTTP 본문으로 변환
주로 폼 데이터(key-value), 쿼리 스트링, URL 패스 변수	JSON, XML, Plain Text, Binary 등 HTTP 본문 데이터
메서드의 파라미터에서 @ModelAttribute 을 선언해서 사용	@RequestBody, @ResponseBody, HttpEntity, ResponseEntity 등과 함께 사용
바인딩에 필요한 타입변환과 검증 과정을 거침	본문 변환 후 검증 과정만 거침

✓ DataBinder

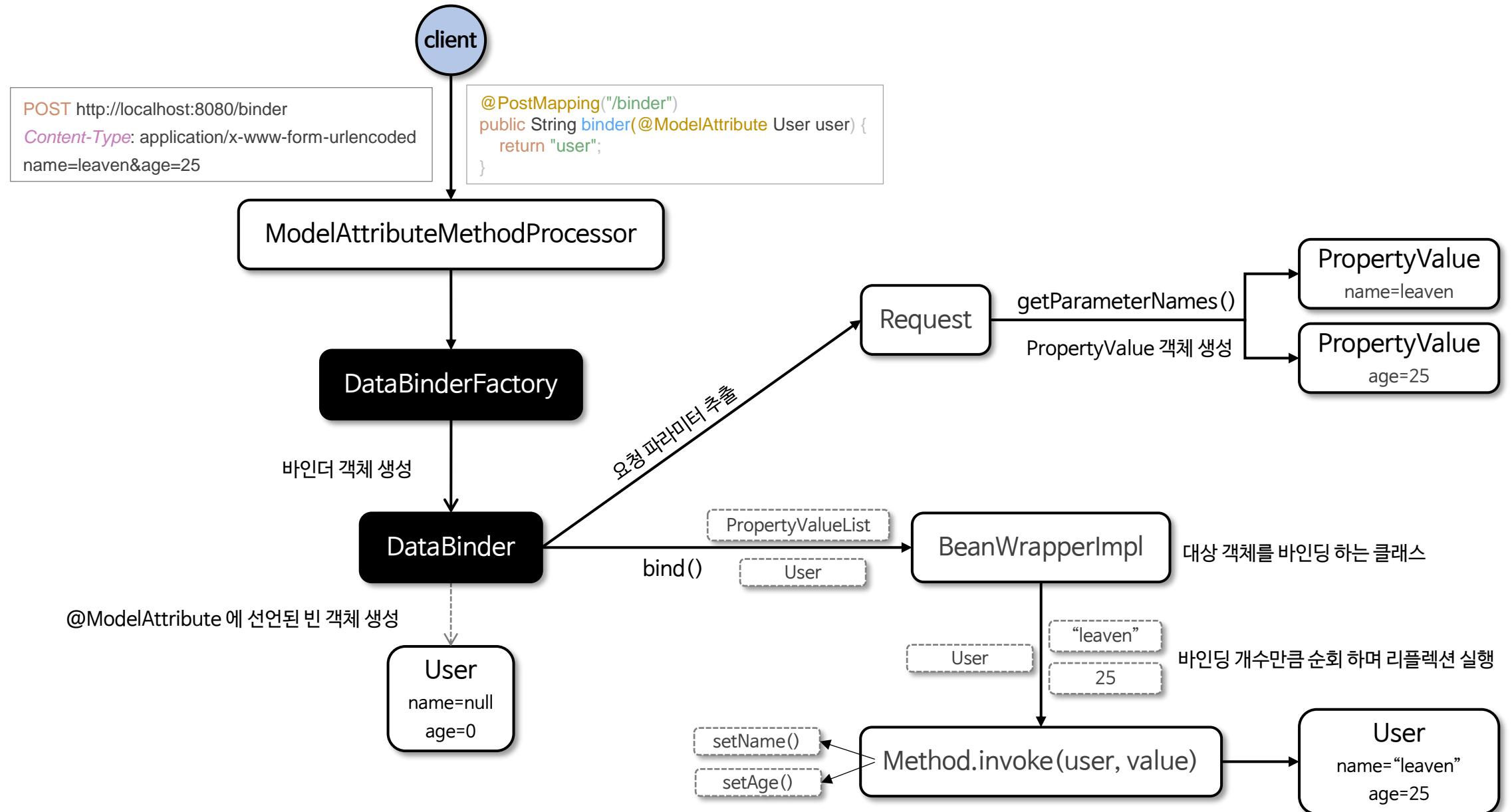
- Spring 의 DataBinder 는 크게 3 가지 특징을 가지고 있는데 바로 HTTP 요청 파라미터를 객체에 바인딩 하고 타입변환 과정 및 유효성 검증을 진행하는 것이다



✓ 계층도



✓ 바인딩 흐름도



@InitBinder

<https://github.com/onjsdnjs/spring-mvc-master/tree/@InitBinder>



개요

- 1) @InitBinder 는 요청 파라미터를 객체의 속성에 바인딩할 때 데이터 바인딩 설정을 커스터마이징 하기 위한 어노테이션이다 (날짜형식, 숫자 형식 등을 지정)
 - ① @Controller에서 @InitBinder를 사용하면 해당 컨트롤러 내에서만 적용
 - ② @ControllerAdvice 와 함께 사용하면 모든 컨트롤러에 적용
- 2) @InitBinder 는 커스텀 유효성 검증기를 등록하여 검증 시 사용할 수 있다
- 3) 데이터 바인딩 시 외부 클라이언트가 필드를 임의로 설정하지 못하도록 allowedFields 나 disallowedFields 설정을 통해 허용 또는 차단할 필드를 지정할 수 있다
- 4) @InitBinder 를 사용하여 PropertyEditor 또는 Formatter 를 등록할 수 있다. 이를 통해 일반적인 데이터 변환 외에 복잡한 데이터 변환 규칙을 적용할 수 있다



메서드 선언

- @InitBinder 메서드는 보통 WebDataBinder 인수를 가지며, 리턴 값이 없는 void 타입이다

```
@InitBinder  
public void initBinder(WebDataBinder binder)
```



WebDataBinder

- void setAllowedFields(@Nullable String... allowedFields) : 바인딩 허용 필드 설정
- void setDisallowedFields(@Nullable String... disallowedFields) : 바인딩 차단 필드 설정
- void registerCustomEditor(Class<?> requiredType, PropertyEditor propertyEditor) : 특정 데이터 타입에 대해 커스텀 변환기를 등록
- void addValidators(Validator... validators) : 커스텀 유효성 검사기 설정
- void setRequiredFields(@Nullable String... requiredFields) : 필수 필드를 지정하여 요청 파라미터에 포함되어야 함을 보장

✓ @InitBinder 를 사용한 날짜 형식 지정

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
    dateFormat.setLenient(false); //엄격한 검사를 실행하도록 설정함  
    // 날짜 형식의 변환기를 등록함  
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));  
}  
  
@RequestMapping(value = "/submitDate", method = RequestMethod.POST)  
public String submitDate(@Valid DateModel dateModel, BindingResult result, Model model) {  
    if (result.hasErrors()) {  
        model.addAttribute("errors", result.getAllErrors());  
        return "dateForm";  
    }  
    model.addAttribute("date", dateModel.getDate());  
    return "dateSuccess";  
}
```

```
public class DateModel {  
    @NotNull  
    private Date date;  
  
    public Date getDate() {  
        return date;  
    }  
  
    public void setDate(Date date) {  
        this.date = date;  
    }  
}
```

```
date=2023/10/10  
date=2023-20-10
```

형식이 맞지 않거나 없는 월을 요청하므로 오류 발생 !!

└ propertyeditors
© ByteArrayPropertyEditor
© CharacterEditor
© CharArrayPropertyEditor
© CharsetEditor
© ClassArrayEditor
© ClassEditor
© CurrencyEditor
© CustomBooleanEditor
© CustomCollectionEditor
© CustomDateEditor
© CustomMapEditor
© CustomNumberEditor
© FileEditor
© InputSourceEditor

✓ allowedFields / disallowedFields 설정

```
public class User {  
    private String username;  
    private String email;  
    private String password; // 비밀번호는 외부에서 바인딩을 허용하지 않음  
  
    public String getUsername() {  
        return username;  
    }  
    public void setUsername(String username) {  
        this.username = username;  
    }  
    public String getEmail() {  
        return email;  
    }  
    public void setEmail(String email) {  
        this.email = email;  
    }  
    public String getPassword() {  
        return password;  
    }  
    public void setPassword(String password) {  
        this.password = password;  
    }  
}
```

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    // username과 email 필드만 바인딩 허용  
    binder.setAllowedFields("username", "email");  
    // binder.setDisallowedFields("password");  
}
```

```
POST /register  
Content-Type: application/x-www-form-urlencoded
```

```
username=leaven&email=leaven@example.com&password=secret123
```

- initBinder 메서드에서 setAllowedFields("username", "email")를 호출하여 username과 email 필드만 바인딩을 허용한다
- 외부 요청에서 password 필드를 포함하더라도 해당 필드는 바인딩되지 않는다.

✓ setRequiredFields 설정

```
public class User {  
    private String username; // 필수  
    private String email; // 필수  
    private String password; // 선택  
  
    public String getUsername() {  
        return username;  
    }  
  
    public void setUsername(String username) {  
        this.username = username;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String password) {  
        this.password = password;  
    }  
}
```

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    // username과 email 필드는 필수  
    binder.setRequiredFields("username", "email");  
}
```

```
POST /register  
Content-Type: application/x-www-form-urlencoded  
  
username=leaven&password=secret123
```

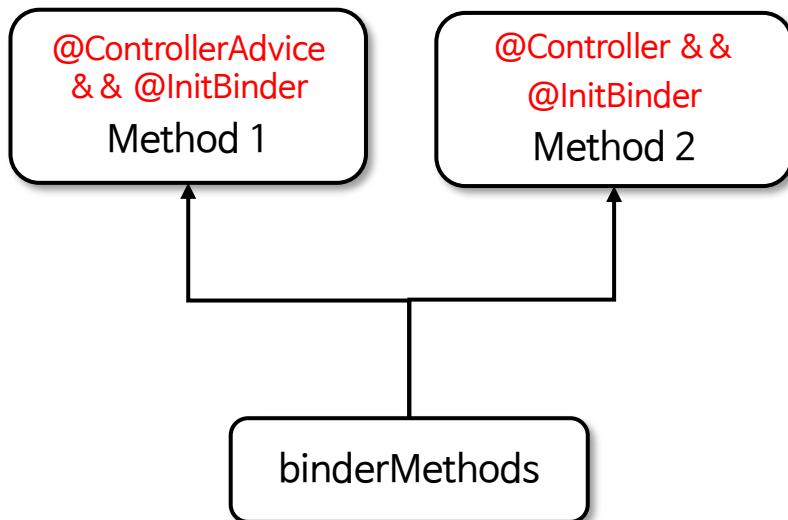
- initBinder 메서드에서 setRequiredFields("username", "email")를 사용하여 username과 email 필드를 필수로 지정하고 요청에 username이거나 email이 누락되면 바인딩 오류가 발생한다

✓ @ControllerAdvice 를 이용한 전역적 바인딩 설정

- @ControllerAdvice 에 @InitBinder 를 선언하면 모든 컨트롤러에서 전역적으로 데이터 바인딩 설정이 적용된다

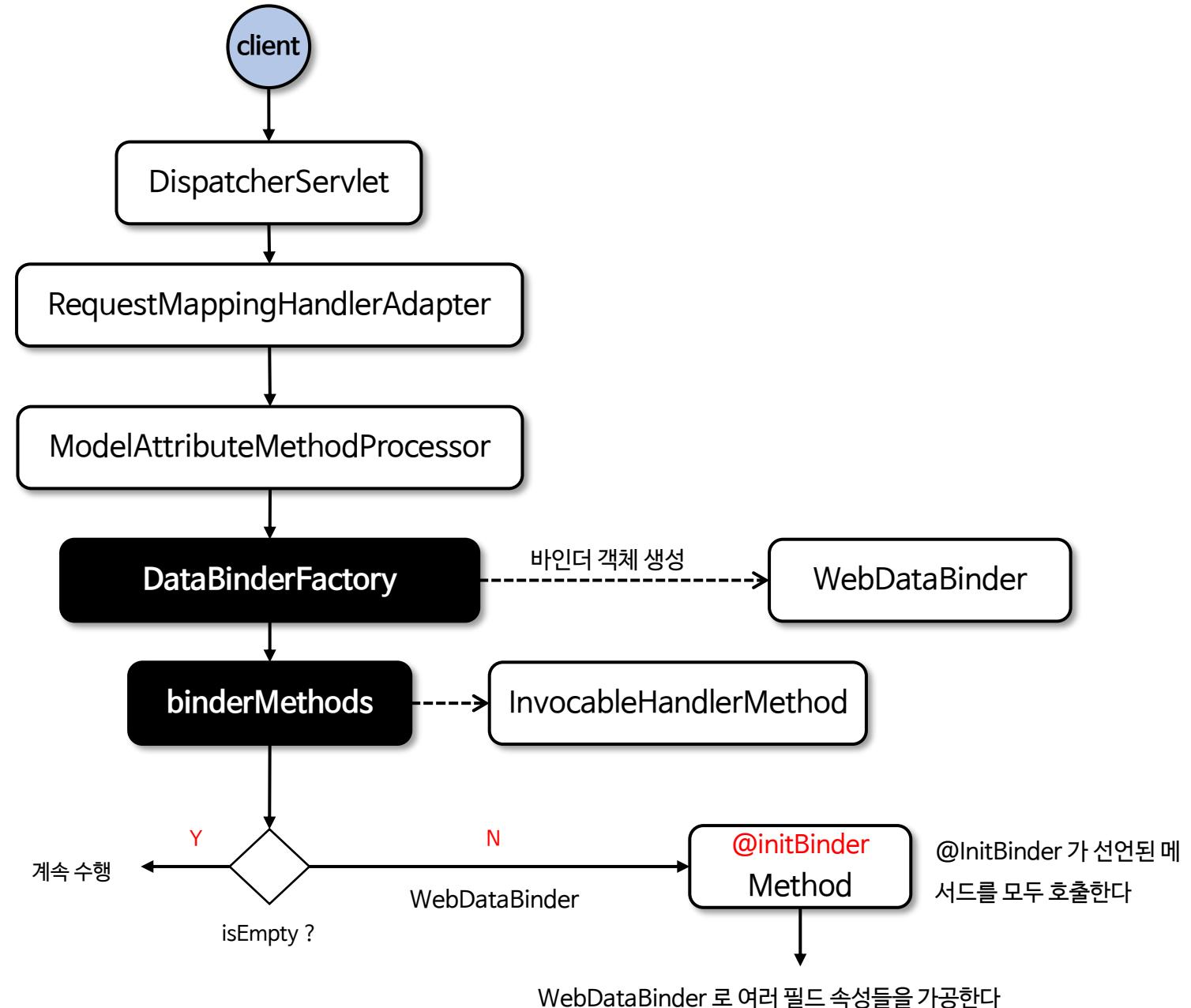
```
@ControllerAdvice  
public class GlobalExceptionHandler {  
  
    @InitBinder  
    public void initBinder(WebDataBinder binder) {  
        // email 필드는 바인딩에서 제외  
        binder.setDisallowedFields("email");  
    }  
  
    @ExceptionHandler(MethodArgumentNotValidException.class)  
    @ResponseBody  
    public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {  
        Map<String, String> errors = new HashMap<>();  
        for (FieldError error : ex.getBindingResult().getFieldErrors()) {  
            errors.put(error.getField(), error.getDefaultMessage());  
        }  
        return errors;  
    }  
}
```

✓ 초기화



초기화 시 `@InitBinder`가 선언된 메서드를 모두 저장한다

✓ 흐름도





스프링 웹 MVC 완전 정복

✓ 타입 변환 (Type Conversion)

1. 개요
2. Converter
3. ConversionService
- 4.ConverterFactory & ConditionalConverter
5. Converter 스프링 적용
6. Formatter
7. FormattingConversionService
8. Formatter 스프링 적용
9. 어노테이션 기반 포맷팅

개요

<https://github.com/onjsdnjs/spring-mvc-master/tree/타입-변환-개요>

✓ 타입 변환

- 타입 변환(Type Conversion)은 데이터를 한 데이터 타입에서 다른 데이터 타입으로 변경하는 과정을 의미한다
- 주로 서로 다른 데이터 타입 간의 호환성을 확보하거나 특정 연산을 수행하기 위해 필요하다

✓ 타입 변환의 종류

• 자동 변환 (Implicit Conversion)

- 컴파일러가 개발자의 개입 없이 자동으로 수행하는 변환.
- 주로 작은 크기의 데이터 타입이 큰 크기의 데이터 타입으로 변환될 때 사용.
- 예: int → double

```
int num = 10;  
double result = num; // 자동으로 int가 double로 변환
```

• 명시적 변환 (Explicit Conversion)

- 개발자가 직접 변환을 명시해야 하는 경우.
- 데이터 손실이 발생할 가능성이 있거나 타입이 다를 경우 사용.
- 예: String → int

```
String numberString = "100";  
int num = Integer.parseInt(numberString); // String 을 int 로 변환
```

✓ 타입 변환 예

문자열 → 숫자

GET /convert? number=123

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    String numberString = req.getParameter("number");  
    Integer number = Integer.valueOf(numberString);  
}
```

문자열 → 날짜

/convertDate? date=2024-11-29

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    String dateString = req.getParameter("date");  
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");  
    LocalDate date = LocalDate.parse(dateString, formatter);  
}
```

문자열 → 객체

/convertUser? name=leaven&age=30

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    String name = req.getParameter("name");  
    String ageString = req.getParameter("age");  
    Integer age = Integer.valueOf(ageString);  
    User user = new User(name, age);  
}
```

✓ 스프링 타입 변환 예

```
@GetMapping("/convert")
public String convertNumber(@RequestParam Integer number) {
    return "Converted number: " + number; // number는 Integer 타입
}
```

GET /convert?number=123

입력: ? number=123 (문자열)

변환: Integer로 자동 변환

출력: Converted number: 123

```
@GetMapping("/user/{id}")
public String getUserId(@PathVariable Long id) {
    return "User ID: " + id; // id는 Long 타입
}
```

GET /user/42

입력: /user/42 (문자열)

변환: Long로 자동 변환

출력: User ID: 42

```
@PostMapping("/submitUser")
public String submitUser(@ModelAttribute User user) {
    return "User: " + user.getName() + ", Age: " + user.getAge() +
        ", City: " + user.getAddress().getCity() +
        ", Zipcode: " + user.getAddress().getZipcode();
}
```

POST /submitForm

name=leaven&age=30&addr.city=Seoul&addr.zipcode=12345

입력: name=leaven&age=30&addr.city=Seoul&addr.zipcode=12345

(문자열)

변환: User 객체, Address 객체, 내부 필드는 Integer, String으로 변환

출력: User: leaven, Age: 30, City: Seoul, Zipcode: 12345

```
class User {
    private String name;
    private Integer age;
    private Address addr
    // Getters and Setters
}
```

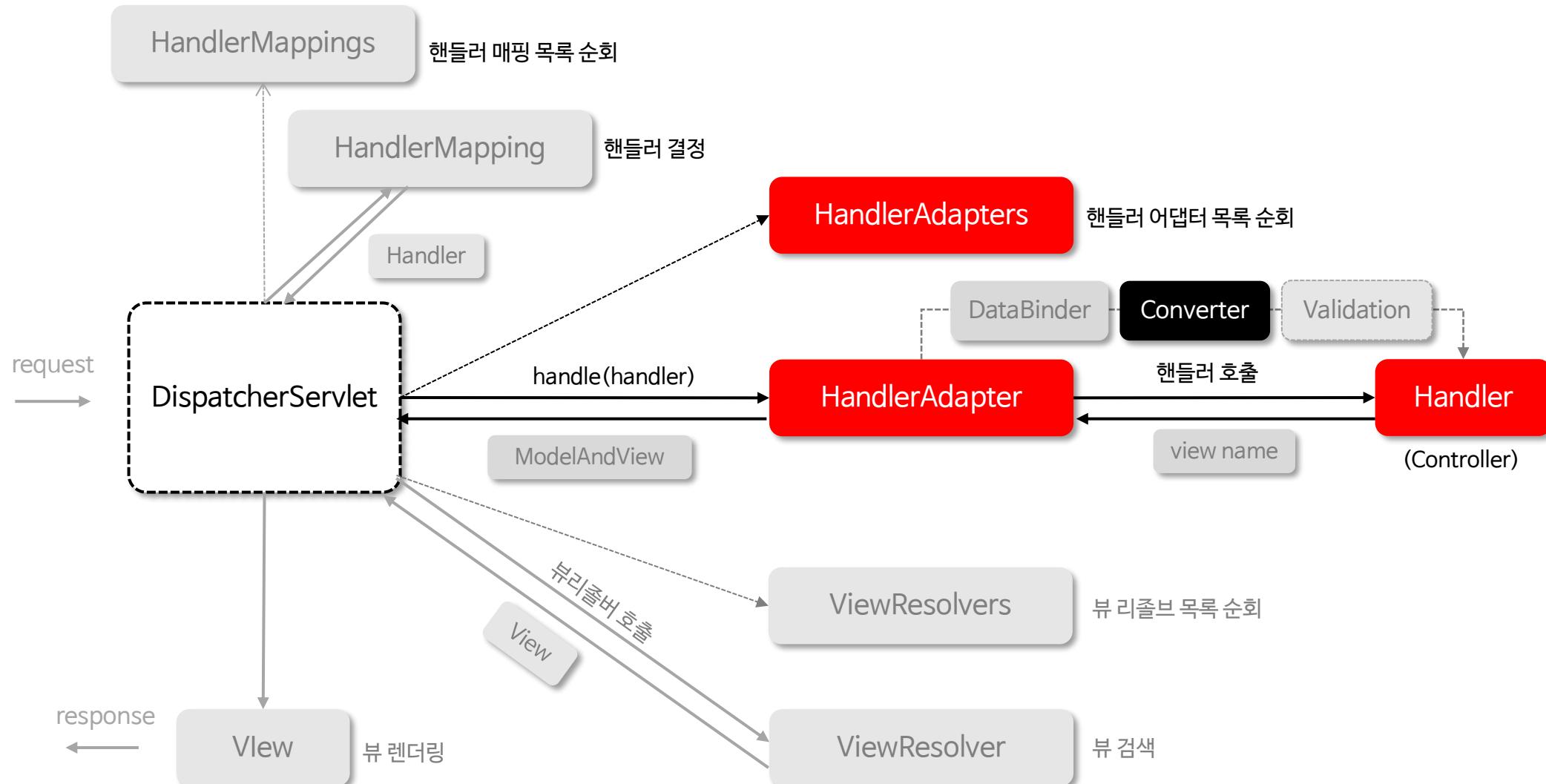
```
class Address {
    private String city;
    private Integer zipcode;
    // Getters and Setters
}
```

- 스프링은 요청 파라미터(문자열)을 기본 타입 및 객체 타입에 맞도록 자동으로 변환해 주고 있다
- @RequestParam, @ModelAttribute, @PathVariable 뒤에 정의한 데이터를 자동으로 타입변환 해 준다
- SpEL(Spring Expression Language), XML 빈 설정, 객체 검증, 뷰 렌더링 시 데이터 간 타입 변환이 이루어진다

Converter

<https://github.com/onjsdnjs/spring-mvc-master/tree/Converter>

✓ 스프링 MVC 아키텍처



✓ 개요

- 타입변환은 바인딩을 처리하는 과정속에 포함되어 있으며 타입 변환이 실패하면 더 이상 바인딩을 진행하지 않고 오류를 발생시킨다
- Spring 의 **Converter<S,T>** 는 입력 데이터를 원하는 데이터 타입으로 변환하기 위한 인터페이스로서 소스 객체(S)를 대상 객체(T)로 변환하는 데 사용된다
- 스프링은 이미 수많은 컨버터 구현체들을 제공하고 있으며 특별한 타입변환이 필요할 경우 Converter 를 직접 구현해서 사용할 수 있다

✓ 구조

```
package org.springframework.core.convert.converter;
```

```
public interface Converter<S, T> {  
    T convert(S source); // 타입 S의 소스 객체를 타입 T의 타겟 객체로 변환한다  
}
```

✓ 기본 구현

```
Integer num = Integer.valueOf(str);      문자열 → 숫자
```

어떤 의미에서는 왼쪽이 코드가 더 간결하게 보인다. 하지만 왼쪽은 수동으로 작업해야 하고 오른쪽은 자동으로 적용 된다는 차이점이 있다

```
public class StringToIntegerConverter implements Converter<String, Integer> {  
    @Override  
    public Integer convert(String source) { //source 는 null 이어서는 안된다  
        if (source == null || source.isEmpty()) throw new IllegalArgumentException();  
        return Integer.valueOf(source);  
    }  
}
```

```
StringToIntegerConverter converter = new StringToIntegerConverter();  
Integer num = converter.convert("10"); // num=10
```

✓ 구현 예제

- 문자열을 Url 객체로, Url 객체를 문자열로 변환하는 예제를 살펴보도록 한다
- Url 은 프로토콜 protocol + domain + port 으로 이루어져 있다
- 웹에서는 요청 파라미터가 컨버터의 소스 타입으로 전달되고 메서드의 매개변수가 컨버터로부터 반환하는 타겟 타입이 된다

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class Url {  
    private String protocol;  
    private String domain;  
    private int port;  
}
```

```
public class UrlToStringConverter implements Converter<Url, String> {  
    @Override  
    public String convert(Url source) {  
        if (source == null) {  
            throw new IllegalArgumentException("Url cannot be null");  
        }  
        return source.getProtocol() + "://" + source.getDomain() + ":" + source.getPort();  
    }  
}
```

http://localhost:8080/url=http://www.springmvc.com:8080

```
public class StringToUrlConverter implements Converter<String, Url> {  
    @Override  
    public Url convert(String source) {  
        if (source == null || source.trim().isEmpty()) {  
            throw new IllegalArgumentException("URL string cannot be null or empty");  
        }  
        String[] parts = url.split("://");  
        String protocol = parts[0];  
        String domainPort = parts[1];  
  
        String[] parts2 = domainPort.split(":");  
        String domain = parts2[0];  
        int port = Integer.parseInt(parts2[1]);  
        return new Url(protocol, domain, port);  
    }  
}
```

String someMethod(@RequestParam("url") Url url)

```

@RestController
@Slf4j
public class UrlController {

    private final StringToUrlConverter stringToUrlConverter = new StringToUrlConverter();
    private final UrlToStringConverter urlToStringConverter = new UrlToStringConverter();

    @GetMapping("/url")
    public String saveUrl(@RequestParam("url") String url) {
        Url result = stringToUrlConverter.convert(url);
        log.info("Url : {}", result);
        return "URL: " + result;
    }

    @PostMapping("/url")
    public String getUrl(@ModelAttribute Url url) {
        String result = urlToStringConverter.convert(url);
        log.info("Url : {}", result);
        return "URL: " + result;
    }
}

```

문제점

1. Converter를 직접 실행하는 것이 아닌 자동적으로 타입 변환이 이루어져야 한다
2. Converter가 내부에 숨겨진 상태에서 타입변환이 이루어져야 하고 사용자는 Converter를 알 필요가 없어야 한다

요청

GET http://localhost:8080/url?url=http://www.springmvc.com:8080

응답

HTTP/1.1 200

Content-Type: text/plain;charset=UTF-8

Content-Length: 63

Url : Url(protocol=http, domain=www.springmvc.com, port=8080)

요청

POST http://localhost:8080/url

Content-Type: application/x-www-form-urlencoded

protocol=http&domain=www.springmvc.com&port=8080

응답

HTTP/1.1 200

Content-Type: text/plain;charset=UTF-8

Content-Length: 63

Url1 : Url1(protocol=http, domain=www.springmvc.com, port=8080)

ConverterFactory & ConditionalConverter

<https://github.com/onjsdnjs/spring-mvc-master/tree/ConverterFactory-ConditionalConverter>

✓ConverterFactory

- ConverterFactory 는 클래스 계층 전체를 처리하기 위한 클래스로서 변환 로직을 따로따로 작성하지 않고 하나의 공통 로직으로 처리할 수 있도록 한다
- 예를 들어 문자열(String) 데이터를 다양한 열거형(Enum) 타입으로 변환해야 할 때 각 열거형마다 변환기를 만들 필요 없이 변환 로직을 일관되게 관리할 수 있다

```
public interfaceConverterFactory<S, R> {  
    <T extends R> Converter<S, T> getConverter(Class<T> targetType); // S를 R의 하위 타입인 T로 변환할 수 있는 변환기를 가져온다  
}
```

✓ 구현 예제

```
public class StringToEnumConverterFactory implementsConverterFactory<String, Enum> {  
  
    @Override  
    public <T extends Enum> Converter<String, T> getConverter(Class<T> targetType) {  
        return new StringToEnumConverter<>()(targetType);  
    }  
  
    private static class StringToEnumConverter<T extends Enum> implements Converter<String, T> {  
        private final Class<T> enumType;  
  
        public StringToEnumConverter(Class<T> enumType) {  
            this.enumType = enumType;  
        }  
        @Override  
        public T convert(String source) {  
            if (source == null || source.trim().isEmpty()) return null;  
            return (T) Enum.valueOf(this.enumType, source.trim().toUpperCase());  
        }  
    }  
}
```

```
public enum Color {  
    RED, GREEN, BLUE  
}  
public enum Status {  
    ACTIVE, INACTIVE, SUSPENDED  
}
```

Main.class

```
StringToEnumConverterFactory factory = new StringToEnumConverterFactory();  
  
// Color 열거형 변환기 가져오기  
Converter<String, Color> colorConverter = factory.getConverter(Color.class);  
Color color = colorConverter.convert("red");  
System.out.println("Converted Color: " + color); // 출력: Converted Color: RED  
  
// Status 열거형 변환기 가져오기  
Converter<String, Status> statusConverter = factory.getConverter(Status.class);  
Status status = statusConverter.convert("active");  
System.out.println("Converted Status: " + status); // 출력: Converted Status: ACTIVE
```

✓ ConditionalConverter

- ConditionalConverter는 특정 조건이 참일 때만 Converter를 실행하고 싶은 경우 사용할 수 있다
- 예를 들어 타겟 필드에 특정 주석이 있을 경우 Converter를 실행하거나 타겟 클래스에 특정 메서드가 정의된 경우 변환기를 실행하고 싶을 때 사용할 수 있다

```
public interface ConditionalConverter {  
    boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType); //소스 타입(sourceType)에서 타겟 타입(targetType)으로의 변환이 실행될 수 있는지 여부를 판단}
```

✓ TypeDescriptor

- 객체의 타입과 관련된 부가적인 정보를 표현하기 위해 사용된다 (클래스 타입 정보, 제네릭 정보, 주석(Annotation), 배열&컬렉션 요소 타입)

클래스 타입 정보

```
TypeDescriptor typeDescriptor = TypeDescriptor.forObject(new Example().names);  
System.out.println(typeDescriptor.getType()); // 출력 : List.class
```

```
public class Example {  
    @Deprecated  
    private List<String> names = new ArrayList<>();  
}
```

제네릭 정보

```
TypeDescriptor descriptor = TypeDescriptor.nested(Example.class.getDeclaredField("names"), 0);  
System.out.println(descriptor.getElementTypeDescriptor().getType()); // 출력: class java.lang.String
```

주석 정보

```
TypeDescriptor descriptor = TypeDescriptor.nested(Example.class.getDeclaredField("names"), 0);  
System.out.println(descriptor.getAnnotation(Deprecated.class));  
// 출력: @java.lang.Deprecated(forRemoval=false, since="")
```

✓ 구현 예제

```
public class CustomConditionalConverter implements ConditionalConverter {  
    @Override  
    public boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType) {  
        // 타겟 필드에 @CustomAnnotation 주석이 있는지 확인  
        return targetType.getAnnotation(CustomAnnotation.class) != null;  
    }  
}
```

```
public class StringToUserConditionalConverter implements Converter<String, User>,  
ConditionalConverter {
```

```
    @Override  
    public boolean matches(TypeDescriptor sourceType, TypeDescriptor targetType) {  
        // 타겟 필드에 @CustomAnnotation이 있는지 확인  
        return targetType.getAnnotations().stream()  
            .anyMatch(ann -> ann instanceof CustomAnnotation);  
    }  
  
    @Override  
    public User convert(String source) {  
        User target = new User();  
        target.setName(source);  
        return target;  
    }  
}
```

```
@Data  
public class User {  
    @CustomAnnotation  
    private String name;  
}
```

```
@Target(ElementType.FIELD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface CustomAnnotation {  
}
```

Main.class

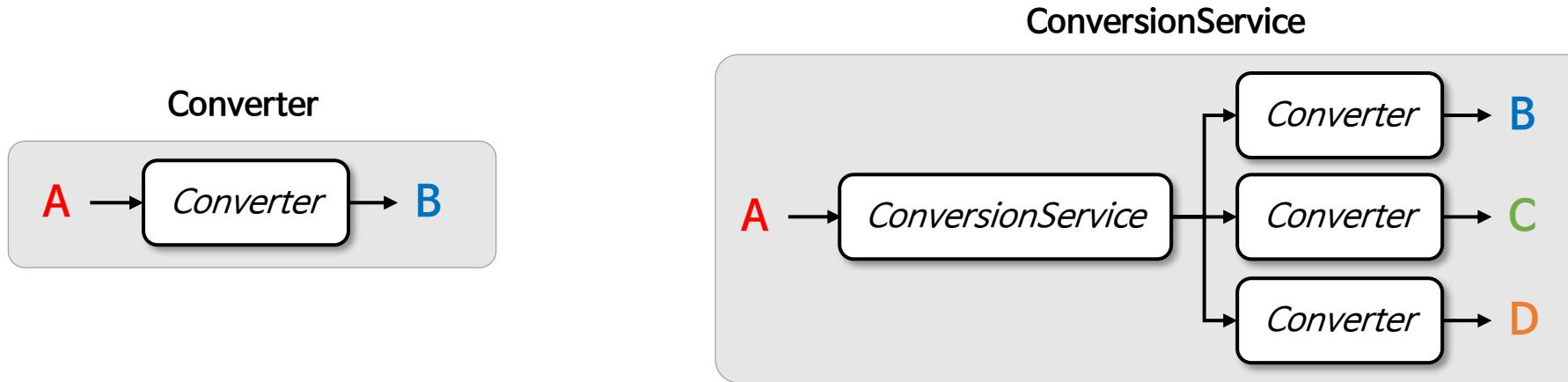
```
// 필드의 TypeDescriptor 생성  
Field field = User.class.getDeclaredField("name");  
TypeDescriptor sourceType = TypeDescriptor.valueOf(String.class); // 소스 타입  
TypeDescriptor targetType = TypeDescriptor.nested(field, 0); // 타겟 타입  
  
StringToUserConditionalConverter converter = new StringToUserConditionalConverter();  
  
if (converter.matches(sourceType, targetType)) {  
    User target = converter.convert("test");  
    System.out.println("name: " + target.getName()); // 출력: name: test  
} else {  
    System.out.println("Conversion not possible.");  
}
```

ConversionService

<https://github.com/onjsdnjs/spring-mvc-master/tree/ConversionService>

✓ 개요

- Converter 가 단일 변환 로직을 위한 것이라면 ConversionService 는 어플리케이션 전반에서 통합된 타입 변환 서비스를 제공한다
- ConversionService 는 타입 변환과 Converter 들을 등록하고 관리하는 기능을 제공하며 데이터 바인딩, 유효성 검사 등에서 통합적으로 사용하고 있다



✓ 타입 변환 시스템 구조

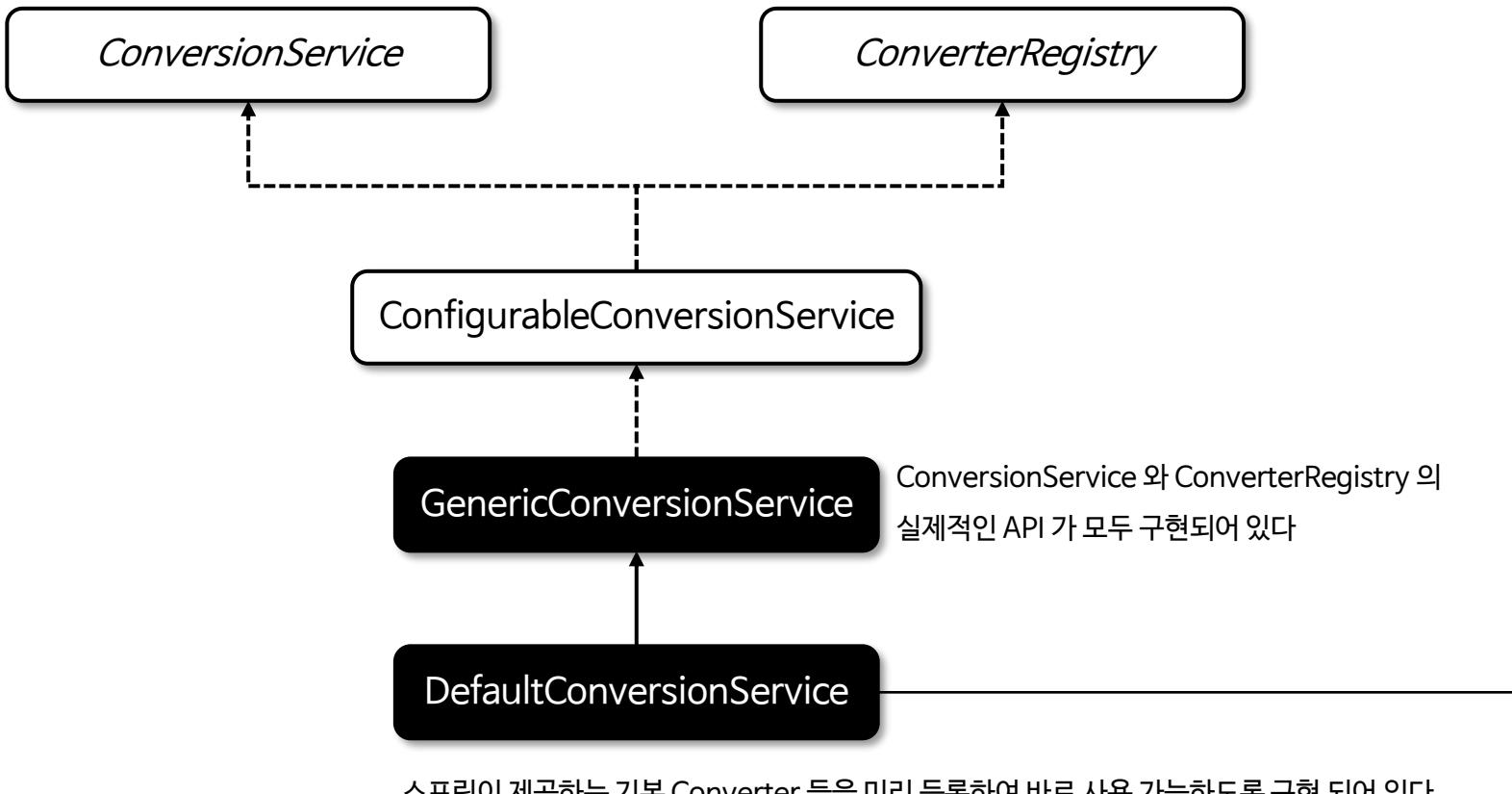
ConversionService
(변환)

```
public interface ConversionService {  
    boolean canConvert(Class<?> sourceType, Class<?> targetType); // sourceType 을 targetType 으로 변환할 수 있는지 여부를 반환  
    <T> T convert(Object source, Class<T> targetType); // 주어진 소스(source)를 지정된 대상 타입(targetType)으로 변환한다
```

ConverterRegistry
(등록)

```
public interface ConverterRegistry {  
    void addConverter(Converter<?, ?> converter); // Converter 를 변환 시스템에 등록한다  
}
```

✓ 클래스 계층도



```
converters = {ConcurrentHashMap@2990} size = 54
> [GenericConverter$ConvertiblePair@3231] "java.util.Collection -> java.util.Optional" -> {GenericConverter$ConvertiblePair@3231}
> [GenericConverter$ConvertiblePair@3233] "java.lang.String -> java.util.TimeZone" -> {GenericConverter$ConvertiblePair@3233}
> [GenericConverter$ConvertiblePair@3235] "java.lang.String -> java.util.Locale" -> {GenericConverter$ConvertiblePair@3235}
> [GenericConverter$ConvertiblePair@3237] "[B -> java.nio.ByteBuffer" -> {GenericConverter$ConvertiblePair@3237}
> [GenericConverter$ConvertiblePair@3060] "java.lang.Number -> java.lang.Number" -> {GenericConverter$ConvertiblePair@3060}
> [GenericConverter$ConvertiblePair@3240] "java.util.stream.Stream -> [Ljava.lang.Object;" -> {GenericConverter$ConvertiblePair@3240}
> [GenericConverter$ConvertiblePair@3242] "java.util.Properties -> java.lang.String" -> {GenericConverter$ConvertiblePair@3242}
> [GenericConverter$ConvertiblePair@3244] "java.lang.String -> java.util.Collection" -> {GenericConverter$ConvertiblePair@3244}
> [GenericConverter$ConvertiblePair@3246] "java.lang.Object -> java.lang.Object" -> {GenericConverter$ConvertiblePair@3246}
> [GenericConverter$ConvertiblePair@3248] "java.util.Collection -> java.util.Collection" -> {GenericConverter$ConvertiblePair@3248}
> [GenericConverter$ConvertiblePair@3250] "java.lang.Integer -> java.lang.Enum" -> {GenericConverter$ConvertiblePair@3250}
> [GenericConverter$ConvertiblePair@3252] "java.util.regex.Pattern -> java.lang.String" -> {GenericConverter$ConvertiblePair@3252}
> [GenericConverter$ConvertiblePair@3254] "java.util.UUID -> java.lang.String" -> {GenericConverter$ConvertiblePair@3254}
> [GenericConverter$ConvertiblePair@3256] "java.nio.ByteBuffer -> [B" -> {GenericConverter$ConvertiblePair@3256}
> [GenericConverter$ConvertiblePair@3258] "java.nio.charset.Charset -> java.lang.String" -> {GenericConverter$ConvertiblePair@3258}
> [GenericConverter$ConvertiblePair@3260] "java.util.Map -> java.util.Map" -> {GenericConverter$ConvertiblePair@3260}
> [GenericConverter$ConvertiblePair@3262] "java.lang.String -> java.util.Currency" -> {GenericConverter$ConvertiblePair@3262}
> [GenericConverter$ConvertiblePair@3264] "[Ljava.lang.Object; -> java.util.Collection" -> {GenericConverter$ConvertiblePair@3264}
> [GenericConverter$ConvertiblePair@3266] "java.lang.Character -> java.lang.Number" -> {GenericConverter$ConvertiblePair@3266}
> [GenericConverter$ConvertiblePair@3268] "java.util.stream.Stream -> java.util.Collection" -> {GenericConverter$ConvertiblePair@3268}
> [GenericConverter$ConvertiblePair@3270] "java.util.Collection -> java.lang.Object" -> {GenericConverter$ConvertiblePair@3270}
> [GenericConverter$ConvertiblePair@3272] "java.lang.Object -> java.util.Collection" -> {GenericConverter$ConvertiblePair@3272}
> [GenericConverter$ConvertiblePair@3274] "java.util.Locale -> java.lang.String" -> {GenericConverter$ConvertiblePair@3274}
> [GenericConverter$ConvertiblePair@3276] "java.lang.String -> java.util.regex.Pattern" -> {GenericConverter$ConvertiblePair@3276}
> [GenericConverter$ConvertiblePair@3278] "java.lang.String -> java.lang.Boolean" -> {GenericConverter$ConvertiblePair@3278}
> [GenericConverter$ConvertiblePair@3280] "java.nio.ByteBuffer -> java.lang.Object" -> {GenericConverter$ConvertiblePair@3280}
> [GenericConverter$ConvertiblePair@3282] "java.lang.String -> java.util.UUID" -> {GenericConverter$ConvertiblePair@3282}
> [GenericConverter$ConvertiblePair@3284] "java.util.Collection -> [Ljava.lang.Object;" -> {GenericConverter$ConvertiblePair@3284}
> [GenericConverter$ConvertiblePair@3286] "[Ljava.lang.Object; -> [Ljava.lang.Object;" -> {GenericConverter$ConvertiblePair@3286}
> [GenericConverter$ConvertiblePair@3288] "java.lang.String -> java.lang.Enum" -> {GenericConverter$ConvertiblePair@3288}
> [GenericConverter$ConvertiblePair@3290] "[Ljava.lang.Object; -> java.lang.Object" -> {GenericConverter$ConvertiblePair@3290}
```

✓ ConversionService 사용하기

```
@Configuration  
public class ConversionServiceConfig {  
    @Bean  
    public DefaultConversionService conversionService() {  
        DefaultConversionService conversionService = new DefaultConversionService();  
        conversionService.addConverter(new StringToUrlConverter()); //ConversionService에 사용할 Converter를 등록한다  
        conversionService.addConverter(new UrlToStringConverter()); //ConversionService에 사용할 Converter를 등록한다  
        return conversionService;  
    }  
}
```

```
@RestController  
public class ConversionController {  
    @Autowired  
    private DefaultConversionService conversionService;  
  
    @PostMapping("/url")  
    public String saveUrl(@RequestParam("url") String url) {  
        Url result = conversionService.convert(url, Url.class);  
        return "Url : " + result;  
    }  
    @GetMapping("/url")  
    public String getUrl(@ModelAttribute Url url) {  
        String result = conversionService.convert(url, String.class);  
        return "Url : " + result;  
    }  
}
```

문제점

1. Converter 예제처럼 여전히 ConversionService.convert() 메서드를 직접 실행하고 있다
2. ConversionService를 내부에 숨기고 자동적으로 변환이 이루어질 수 있는 API가 필요하다

Converter 스프링 적용

<https://github.com/onjsdnjs/spring-mvc-master/tree/Converter-%EC%A0%95%EB%8A%84%EB%8D%90>

✓ WebMvcConfigurer에 Converter 등록하기

- 앞에서 수동으로 변환했던 작업을 스프링에 의한 자동 변환 방식으로 적용되도록 WebMvcConfigurer를 사용해 Converter를 등록한다
- FormatterRegistry는 웹에서 전반적으로 사용되는 WebConversionService 구현체가 전달된다

```
@Configuration  
public class WebConfig implements WebMvcConfigurer {  
  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
  
        registry.addConverter(new StringToUrlConverter());  
  
        registry.addConverter(new UrlToStringConverter());  
  
        registry.addConverter(new StringToUserConditionalConverter());  
  
        registry.addConverterFactory(new StringToEnumConverterFactory());  
  
    }  
}
```

- 사용자가 명시적으로 Converter를 등록하면 소스와 타겟 타입이 동일하게 지정된 스프링의 기본 Converter 보다 우선적으로 적용된다

✓ 구현 예제

```
@RestController  
@Slf4j  
public class SpringConverterController {  
  
    @PostMapping("/url")  
    public String saveUrl(@ModelAttribute("url") Url url) {  
        return "url: " + url;  
    }  
  
    @PostMapping("/users")  
    public User user(@RequestParam("user") User user) {  
        return user;  
    }  
  
    @PostMapping("/color")  
    public OrderColor color(@RequestParam("color") OrderColor color) {  
        return color;  
    }  
}
```

컨버터를 통해 내부적으로 자동 변환이 이루어지고 있다

POST http://localhost:8080/url
Content-Type: application/x-www-form-urlencoded
url=http://www.springmvc.com:8080

new StringToUrlConverter()

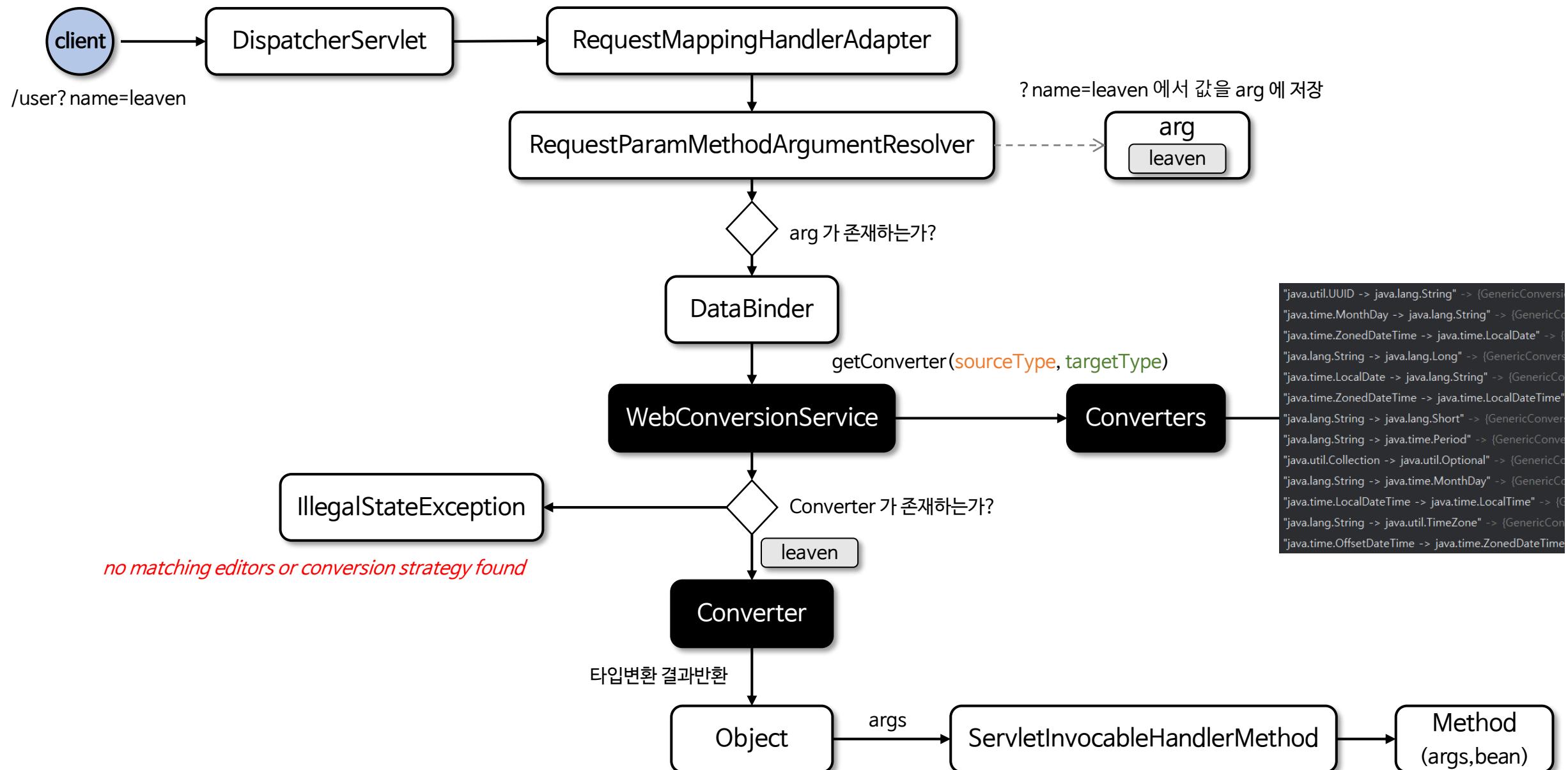
POST http://localhost:8080/users
Content-Type: application/x-www-form-urlencoded
user=springmvc

new StringToUserConditionalConverter()

GET http://localhost:8080/color?color=red

new StringToEnumConverterFactory()

✓ Converter 흐름도 - @RequestParam



바인딩과 타입변환 관계

<https://github.com/onjsdnjs/spring-mvc-master/tree/Converter-%ED%95%91%EC%8A%A4>

✓ 바인딩 vs 타입 변환 관계

- **DataBinder 의 역할**

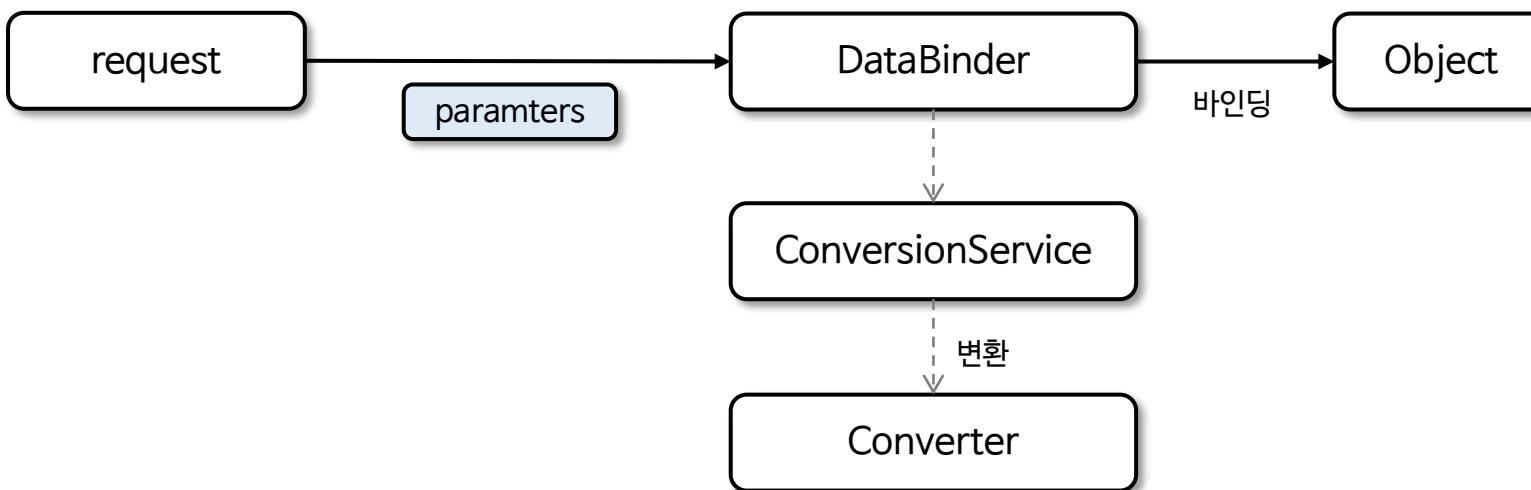
- DataBinder 는 요청 데이터를 객체로 바인딩하는 클래스로서 HTTP 요청 파라미터(쿼리 파라미터, 폼 데이터 등) 를 자바 객체의 속성에 매핑한다

- **ConversionService 의 역할**

- ConversionService 는 타입 변환에 특화된 서비스로서 바인딩 중 필요한 경우 Converter 를 호출해 특정 필드 값을 변환한다

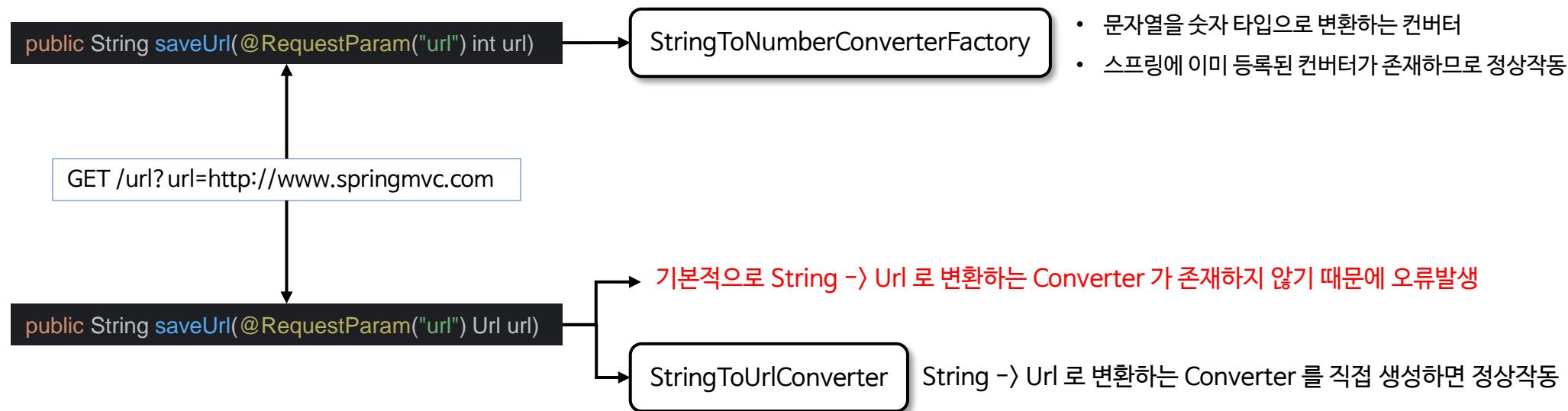
- **DataBinder 와 ConversionService 의 상호 작용**

- DataBinder 는 필요할 경우 ConversionService 를 통해 요청 데이터를 변환하고 변환된 데이터를 객체 필드에 할당한다
- 바인딩 중에 타입변환이 실패하게 되면 예외가 발생해 바인딩을 더 이상 진행하지 않거나 BindingResult 를 사용해 오류를 남기고 계속 바인딩을 진행할 수 있다



✓ @RequestParam 동작 방식

- 1) 클라이언트가 보낸 요청 파라미터를 문자열 형태로 가져온다 (기본적으로 모든 요청 파라미터는 String 형태로 전달된다)
- 2) 가져온 문자열을 @RequestParam에 지정한 매개변수에 저장하기 위해 ConversionService를 사용하여 타입 변환을 수행한 후 저장한다
- 3) 타입 변환이 실패하면 스프링은 즉시 TypeMismatchException을 발생시키며 오류 응답을 반환하거나 별도로 설정된 예외 처리 로직이 실행된다



- `@RequestParam`에 지정된 매개변수가 객체인 경우 보통 컨버터가 등록 안되어 있기 때문에 오류가 발생한다
- 객체를 지정해서 사용할 경우는 `@ModelAttribute`를 사용하든지 아니면 객체타입으로 변환할 수 있는 컨버터를 만들어야 한다

✓ @ModelAttribute 동작 방식

- @ModelAttribute 는 클라이언트의 요청 유형에 따라 객체 바인딩 방식과 타입변환 방식으로 객체가 생성되어 메서드에 전달된다

1. 객체 바인딩 방식

```
@PostMapping("/url")
public String saveUrl(@ModelAttribute Url url) {
    return "Url :" + url;
}
```

```
POST http://localhost:8080/url
Content-Type: application/x-www-form-urlencoded
protocol=http&domain=www.springmvc.com&port=8080
```

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Url {
    private String protocol;
    private String domain;
    private String port;
}
```

- 클라이언트의 요청 파라미터가 @ModelAttribute 에 지정된 매개변수명과 다른 경우 객체 바인딩 방식으로 객체가 생성된다
 - 1) 요청 파라미터 명 : protocol, domain, port
 - 2) @ModelAttribute 매개변수명 : url
 - 3) 객체를 만들고 객체의 필드명과 동일한 요청 파라미터의 값을 설정함
 - 4) 필드타입과 요청파라미터 타입이 다를 경우 타입변환을 거쳐 설정함

2. 타입 변환 방식

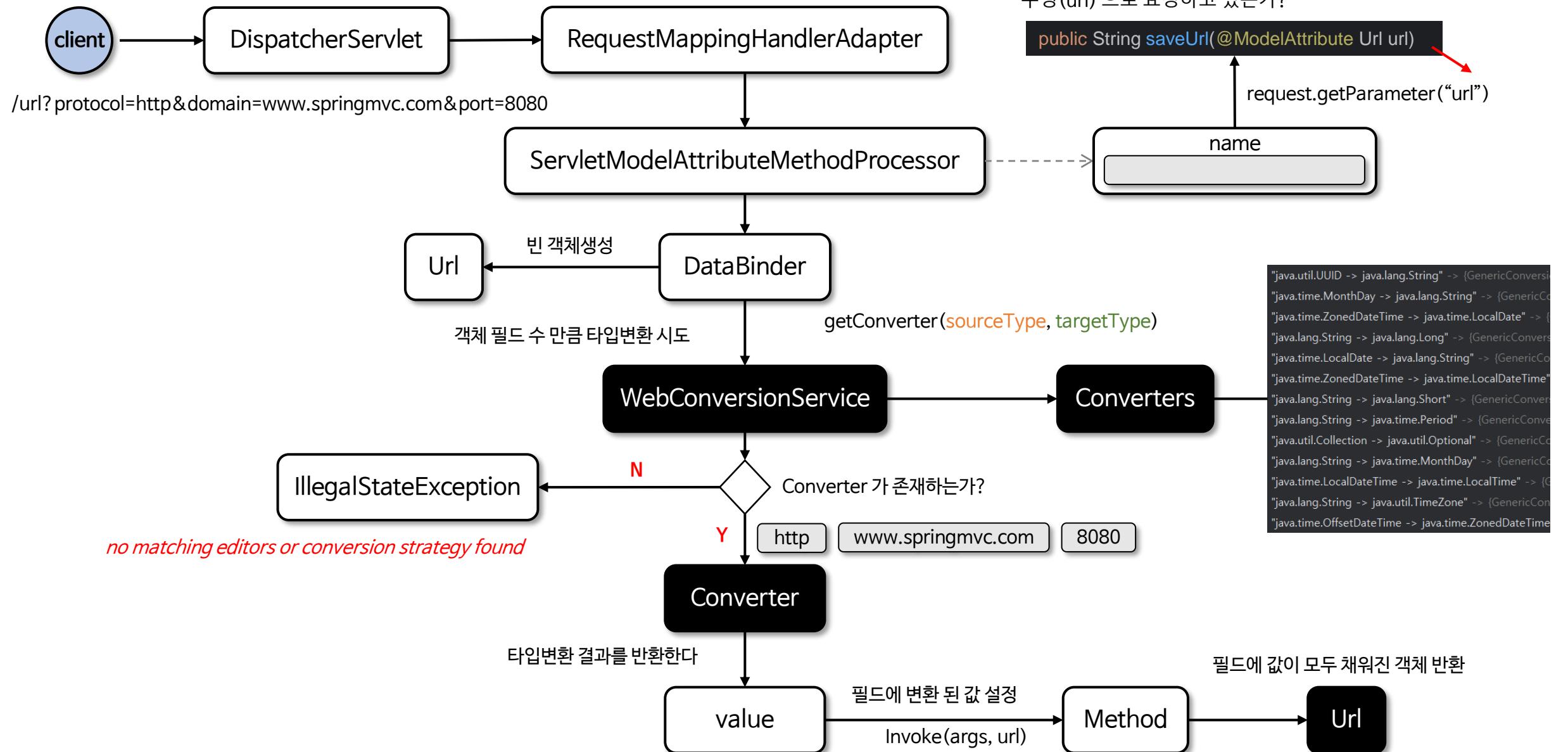
```
@PostMapping("/url")
public String saveUrl(@ModelAttribute Url url) {
    return "Url :" + url;
}
```

```
POST http://localhost:8080/url
Content-Type: application/x-www-form-urlencoded
url=http://www.springmvc.com:8080
```

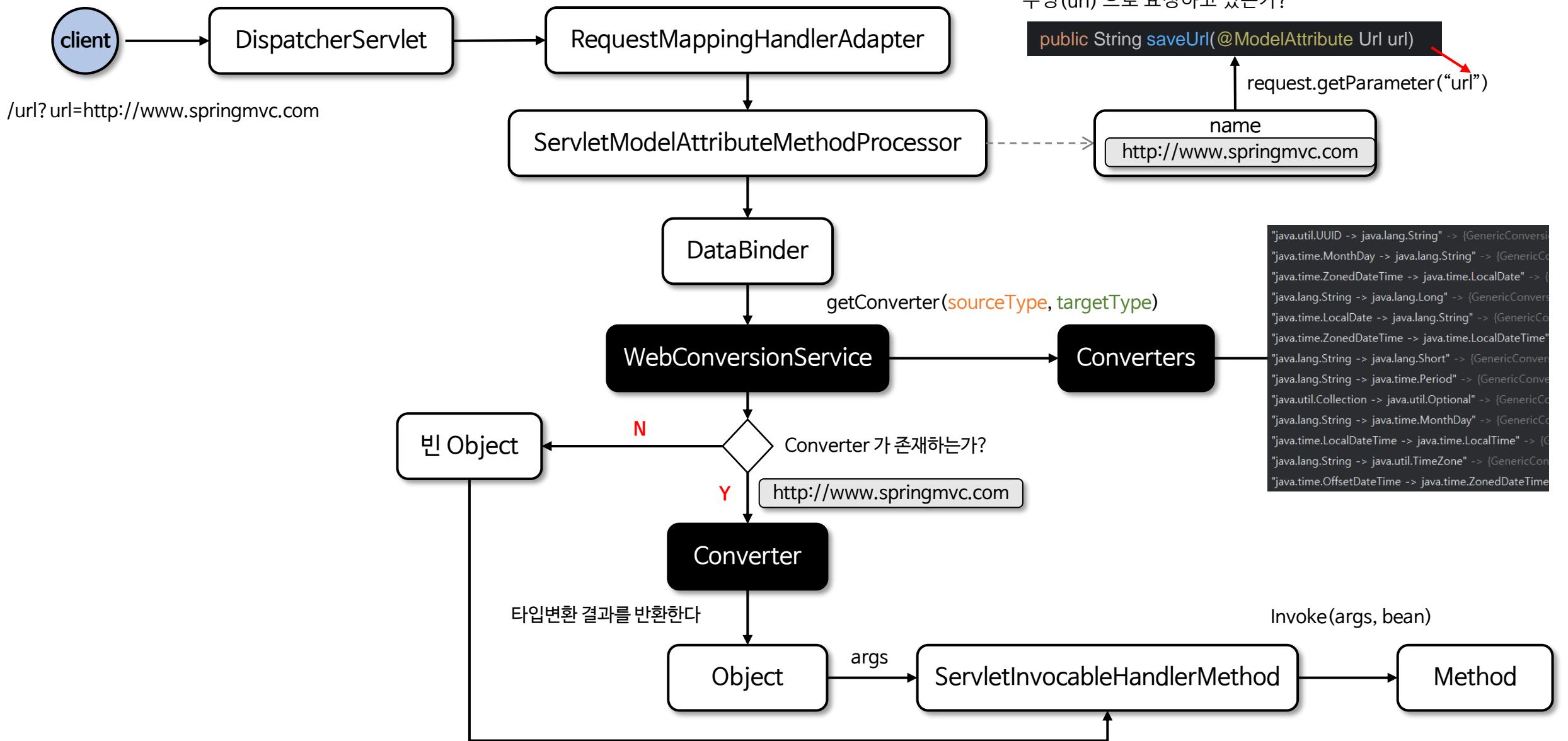
```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Url {
    private String protocol;
    private String domain;
    private String port;
}
```

- 클라이언트의 요청 파라미터가 @ModelAttribute 에 지정된 매개변수명과 같은 경우 타입변환 방식으로 객체가 생성된다
 - 1) 요청 파라미터 명 : url
 - 2) @ModelAttribute 매개변수명 : url
 - 3) StringToUrlConverter 를 통해 Url 객체 생성 후 메서드에 전달
 - 4) Converter 가 존재하지 않으면 빈 객체 생성 후 메서드에 전달

✓ Converter 흐름도 - @ModelAttribute (객체 바인딩)



✓ Converter 흐름도 - @ModelAttribute (타입 변환)



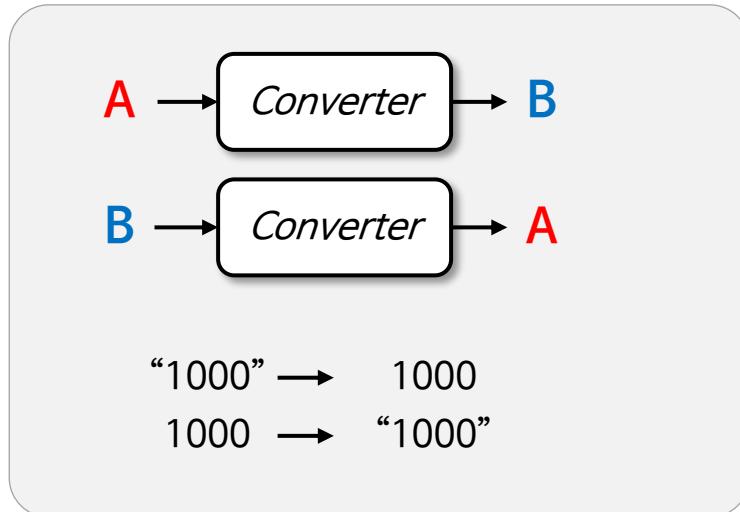
Formatter

<https://github.com/onjsdnjs/spring-mvc-master/tree/Formatter>

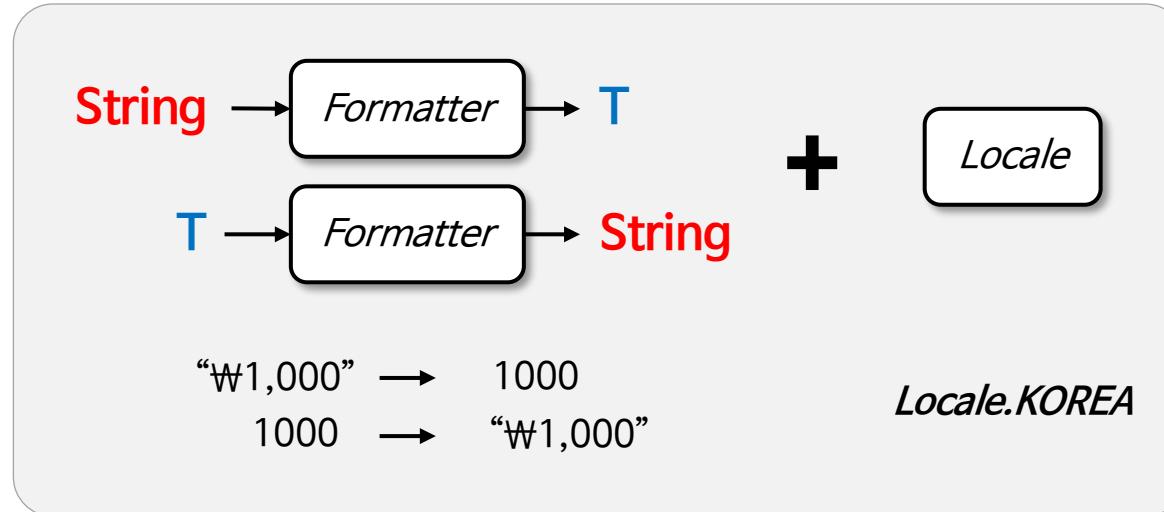
✓ 개요

- 앞에서 학습한 Converter는 범용 타입 변환 시스템이다. 즉 타입에 제한 없이 데이터 타입 간 일반적인 변환을 다루는데 목적이 있다
- Formatter는 특정한 환경(예: 웹 애플리케이션)에서 데이터를 특정 형식에 맞게 변환하거나 특정 형식에서 객체로 변환하는 것에 목적이 있다
- Converter는 로컬화(Localization)를 고려하지 않지만 Formatter는 지역(Locale)에 따라 데이터 표현방식을 다르게 처리할 수 있다
- Converter가 주로 서버 내부 데이터 변환에 사용된다면 Formatter는 뷰(View)와 클라이언트 간 데이터 변환에 사용된다고 볼 수 있다

Converter



Formatter



범용 타입 변환에 적합

로컬화를 고려한 특정 타입 변환에 적합

스프링의 **ConversionService**는 이 두 SPI를 모두 지원하는 통합된 타입 변환 API를 제공한다

✓ 구조

T 타입의 객체를 문자열 형식으로 변환

```
public interface Printer<T> {  
    String print(T object, Locale locale);  
}
```

문자열을 파싱하여 T 타입의 객체를 생성

```
public interface Parser<T> {  
    T parse(String text, Locale locale) throws ParseException;  
}
```

```
public interface Formatter<T> extends Printer<T>, Parser<T> {  
}
```

✓ 기본 구현

```
public class CustomNumberFormatter implements Formatter<Number> {  
    @Override  
    public String print(Number number, Locale locale) {  
        NumberFormat format = NumberFormat.getInstance(locale);  
        return format.format(number);  
    }  
    @Override  
    public Number parse(String text, Locale locale) throws ParseException {  
        NumberFormat format = NumberFormat.getInstance(locale);  
        return format.parse(text);  
    }  
}
```

```
CustomNumberFormatter formatter = new CustomNumberFormatter();  
  
// 1,000 -> 1000 변환  
String currencyText = "1,000";  
Number number = formatter.parse(currencyText, Locale.KOREA);  
System.out.println("Parsed number: " + number); // 출력: 1000  
  
// 1000 -> 1,000 변환  
String formattedText = formatter.print(1000, Locale.KOREA);  
System.out.println("Formatted text: " + formattedText); // 출력: 1,000
```

• 스프링은 다양한 Formatter 구현체를 제공하고 있다

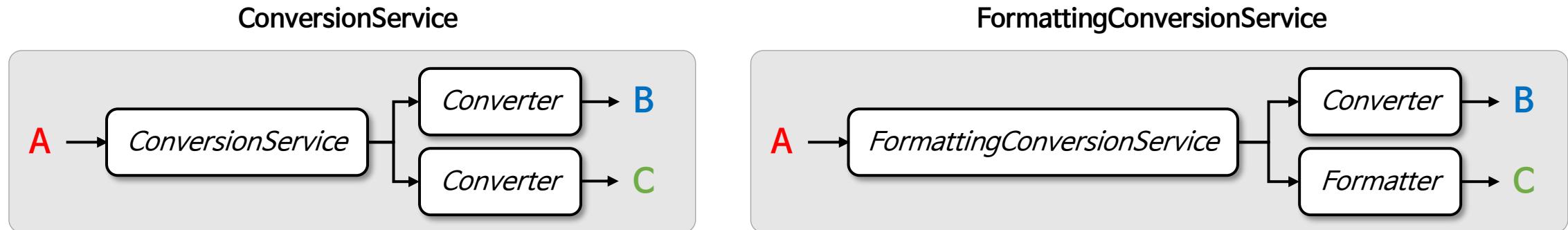
- 숫자 포맷터 (NumberStyleFormatter, CurrencyStyleFormatter, PercentStyleFormatter): 숫자 데이터를 포맷
- 날짜 포맷터 (DateFormatter, DurationFormatter): 날짜와 시간 데이터를 포맷

FormattingConversionService

<https://github.com/onjsdnjs/spring-mvc-master/tree/FormattingConversionService>

✓ 개요

- FormattingConversionService 는 Converter 와 Formatter 를 통합적으로 등록할 수 있는 구현체로서 타입 변환과 로컬화된 데이터 포맷팅을 함께 처리할 수 있다



✓ 구조

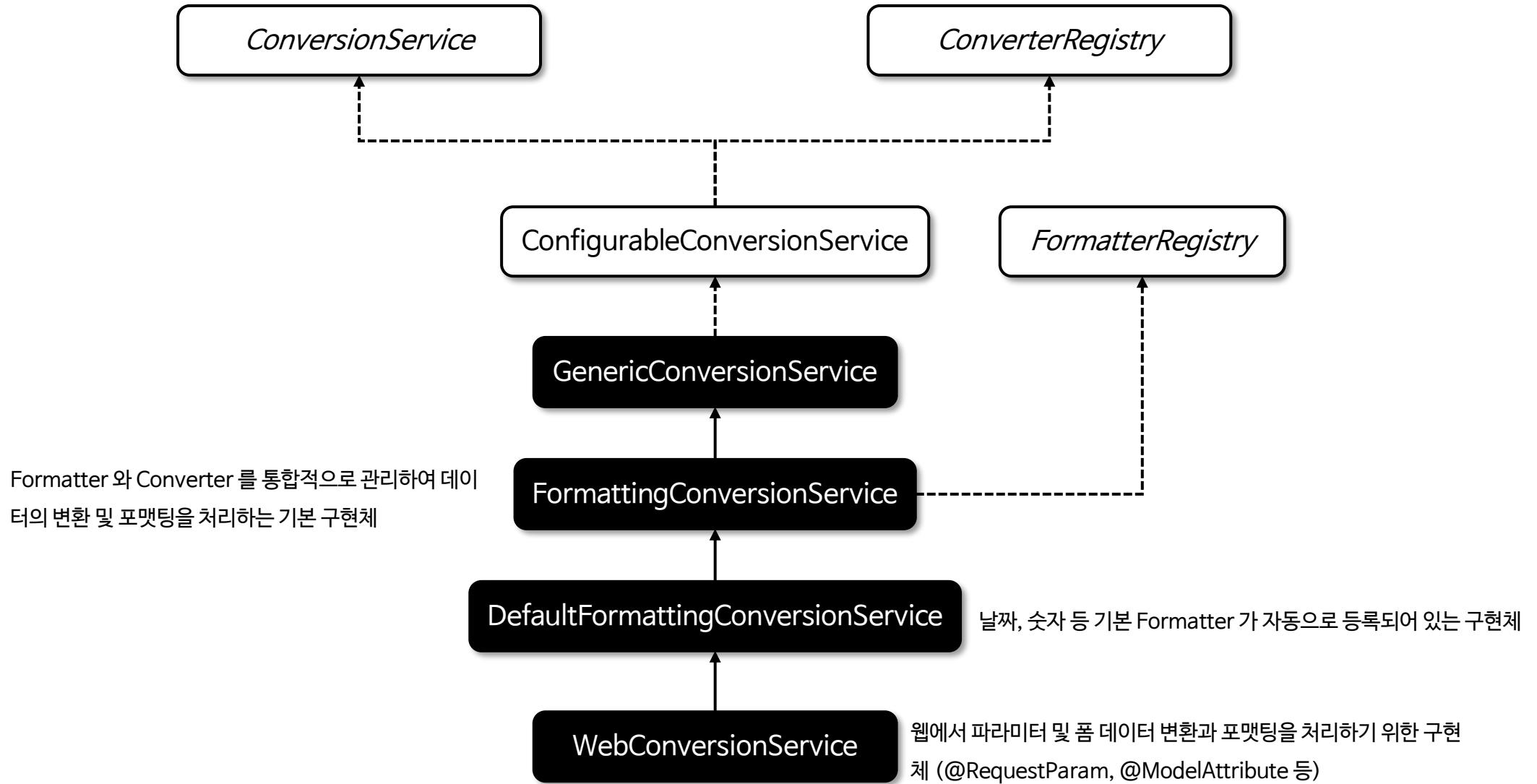
FormatterRegistry

```
public interface FormatterRegistry extends ConverterRegistry // Converter와 Formatter 를 모두 등록한다
void addPrinter(Printer<?> printer); // 필드를 출력할 Printer 추가
void addParser(Parser<?> parser); // 필드를 파싱할 Parser 추가
void addFormatter(Formatter<?> formatter); // 필드를 출력하고 파싱할 Formatter 추가
void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter); // 특정 타입의 필드를 지정해서 출력하고 파싱할 Formatter 추가
void addFormatterForFieldType(Class<?> fieldType, Printer<?> printer, Parser<?> parser); // 특정 타입의 필드를 지정해서 출력할 Printer 와 파싱할 Parser 를 추가
void addFormatterForFieldAnnotation(AnnotationFormatterFactory<? extends Annotation> annotationFormatterFactory); // 어노테이션 방식으로 포맷
```

FormatterRegistrar

```
public interface FormatterRegistrar // FormatterRegistry 를 사용해 Converter와 Formatter 를 한 번에 FormattingConversionService 에 등록
void registerFormatters (FormatterRegistry registry);
```

✓ 클래스 계층도



✓ FormattingConversionService 사용하기

```
class KoreanCurrencyFormatter implements Formatter<Number> {  
    @Override  
    public String print(Number number, Locale locale) {  
        NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);  
        return currencyFormat.format(number);  
    }  
  
    @Override  
    public Number parse(String text, Locale locale) throws ParseException {  
        NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);  
        return currencyFormat.parse(text);  
    }  
}
```

- 숫자를 전달하면 원화로 출력한다
 - 3000 -> ₩3,000
- 원화를 전달하면 숫자를 반환한다
 - ₩3,000 -> 3000

```
@Test  
void testCurrencyFormatting() throws Exception {  
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService();  
    conversionService.addFormatter(new KoreanCurrencyFormatter());  
  
    String currency = "₩1,000";  
    Number parsedCurrency = conversionService.convert(currencyString, Number.class); // 문자열 -> Number 변환 테스트  
    assertThat(parsedCurrency).isEqualTo(1000);  
  
    String printedCurrency = conversionService.convert(1000, String.class); // Number -> 문자열 변환 테스트  
    assertThat(printedCurrency).isEqualTo("₩1,000");  
}
```

✓ CustomFormatterRegistrar 사용하기

```
public class CustomFormatterRegistrar implements FormatterRegistrar {  
    @Override  
    public void registerFormatters(FormatterRegistry registry) { // 컨버터와 포맷터를 통합해서 등록할 수 있다  
        registry.addFormatter(new KoreanCurrencyFormatter());  
        registry.addConverter(new StringToUrlConverter());  
    }  
}
```

```
@Test  
void testCurrencyFormatting() throws Exception {  
    DefaultFormattingConversionService conversionService = new DefaultFormattingConversionService();  
    CustomFormatterRegistrar registrar = new CustomFormatterRegistrar();  
    registrar.registerFormatters(conversionService);  
  
    String currencyString = "₩1,000";  
    // 문자열 -> Number 변환 테스트  
    Number parsedCurrency = conversionService.convert(currencyString, Number.class);  
    assertThat(parsedCurrency).isEqualTo(1000);  
  
    // Number -> 문자열 변환 테스트  
    String formattedCurrency = conversionService.convert(1000, String.class);  
    assertThat(formattedCurrency).isEqualTo("₩1,000");  
}
```

Formatter 스프링 적용

<https://github.com/onjsdnjs/spring-mvc-master/tree/Formatter-%ED%95%91%EC%8A%A4%ED%8A%B8>

✓ WebMvcConfigurer 에 Formatter 등록하기

```
@Configuration  
public class WebConfig implements WebMvcConfigurer {  
  
    @Override  
    public void addFormatters(FormatterRegistry registry) {  
        registry.addFormatter(new KoreanCurrencyFormatter());  
    }  
}
```

```
@Override  
public void addFormatter(Formatter<?> formatter) {  
    addFormatterForFieldType(getFieldType(formatter), formatter);  
}  
  
@Override  
public void addFormatterForFieldType(Class<?> fieldType, Formatter<?> formatter) {  
    addConverter(new PrinterConverter(fieldType, formatter, this));  
    addConverter(new ParserConverter(fieldType, formatter, this));  
}
```

- **fieldType** - 변환 대상인 T 타입을 의미한다
- **formatter** - 실제 포매팅을 적용하는 객체이다
- **this** - FormattingConversionService 객체이다

PrinterConverter

```
public Object convert(Object source, TypeDescriptor source, TypeDescriptor target) {  
  
    // 생략 ..  
    return this.printer.print(source, LocaleContextHolder.getLocale());  
}
```

ParseConverter

```
public Object convert(Object source, TypeDescriptor source, TypeDescriptor target) {  
  
    // 생략  
    return this.parser.parse(text, LocaleContextHolder.getLocale());  
}
```

- 구조를 보면 Formatter는 두 개의 Converter로 나누어지고 등록된다. 즉, 한 개의 Formatter는 두 개의 Converter로부터 출력(print) 및 파싱(parse) 메서드가 호출되어 실행된다
- 실행 관점에서 보면 타입 변환은 항상 Converter가 최초 진입점이 되고 Formatter는 Converter 내에서 실행되는 어댑터 패턴 구조라 볼 수 있다

✓ 뷰 템플릿과 타입변환

- 뷰 템플릿인 Thymeleaf는 모델 데이터에 타입 변환 표현식을 사용하면 뷰 렌더링 시점에 ConversionService가 자동적으로 타입 변환 시스템을 가동시킨다
- 타입 변환 표현식은 `${{...}}`이며 모델 데이터를 표현식에 입력하면 된다(일반 변수 표현식은 `$...` 인 것과 구분해서 사용)

currency-form.html

```
1 <form th:action="@{/currency}" method="post">  
    <label for="amount">Enter Amount:</label>  
    <input type="text" id="amount" name="amount" th:value="${{amount}}"/>  
    <button type="submit">Submit</button>  
</form>
```

- ① GET /currency 요청
- ② 폼 필드 amount에 변환된 ₩123,456 가 렌더링 됨

Currency Form

Enter Amount: ₩12,345

Submit

- ③ POST /currency 요청
- ④ 폼 필드 amount의 값인 ₩123,456 가 서버에 전달
- ⑤ 서버의 KoreanCurrencyFormatter 가 ₩123,456 을 12345 숫자로 변환하여 매개변수인 amount에 바인딩

```
2 @PostMapping("/currency")  
public String handleCurrencyForm(@RequestParam("amount") int amount,  
    model.addAttribute(attributeName: "amount", amount); amount: 12345  
    return "currency-result";  
}
```

```
@Controller  
public class CurrencyController {  
    @GetMapping("/currency")  
    public String showCurrencyForm(Model model) {  
        model.addAttribute("amount", 12345);  
        return "currency-form";  
    }  
    @PostMapping("/currency")  
    public String handleCurrencyForm(@RequestParam("amount") int amount, Model model) {  
        model.addAttribute("amount", amount);  
        return "currency-result";  
    }  
}
```

currency-result.html

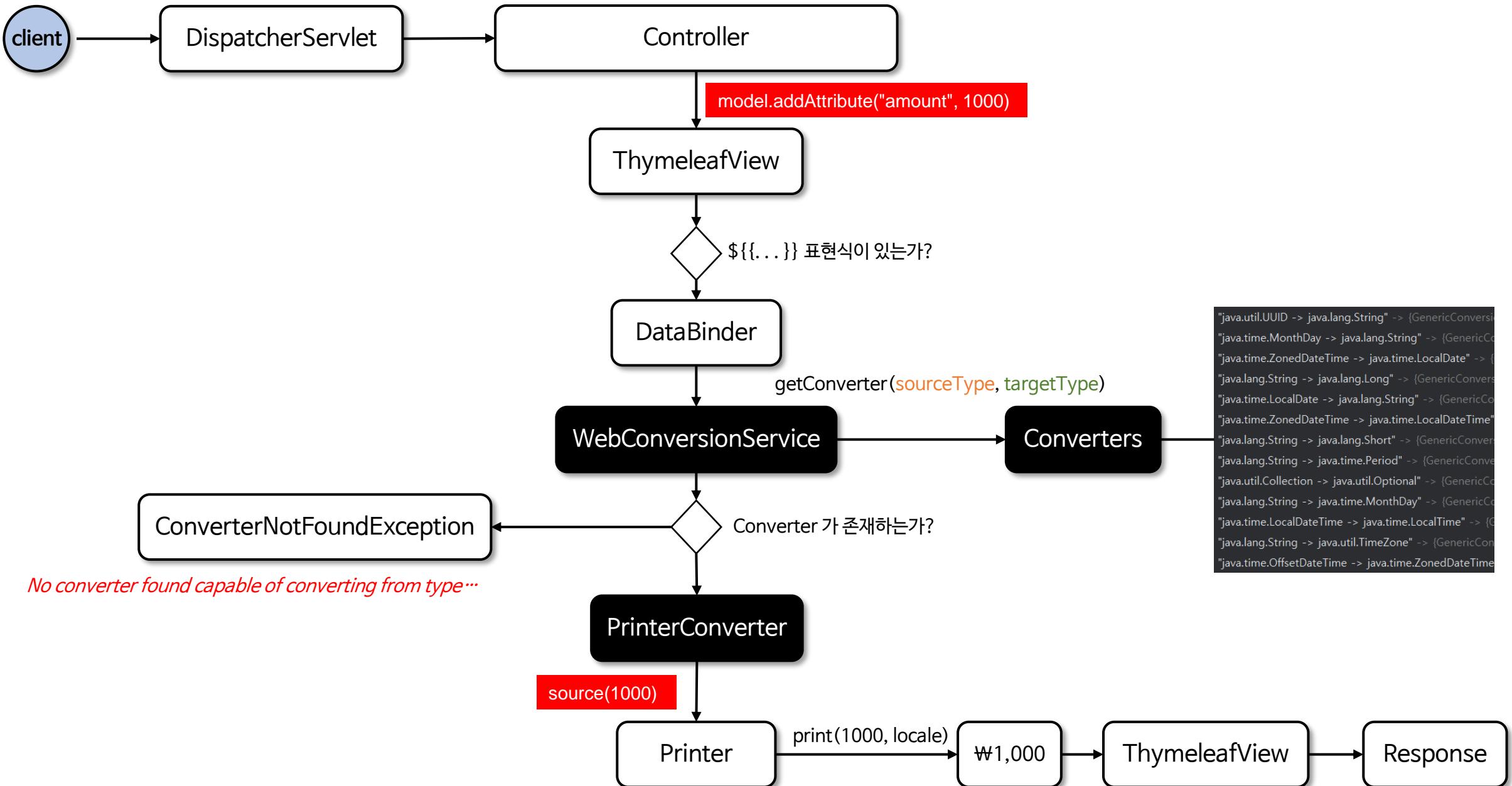
```
3 <p>Processed Amount: <span th:text="${{amount}}"></span></p>
```

- 결과 페이지 amount에는 변환된 ₩123,456 가 렌더링 됨
- `${{amount}}` 로 하면 객체의 기본 `toString()` 인 숫자 12345가 출력된다

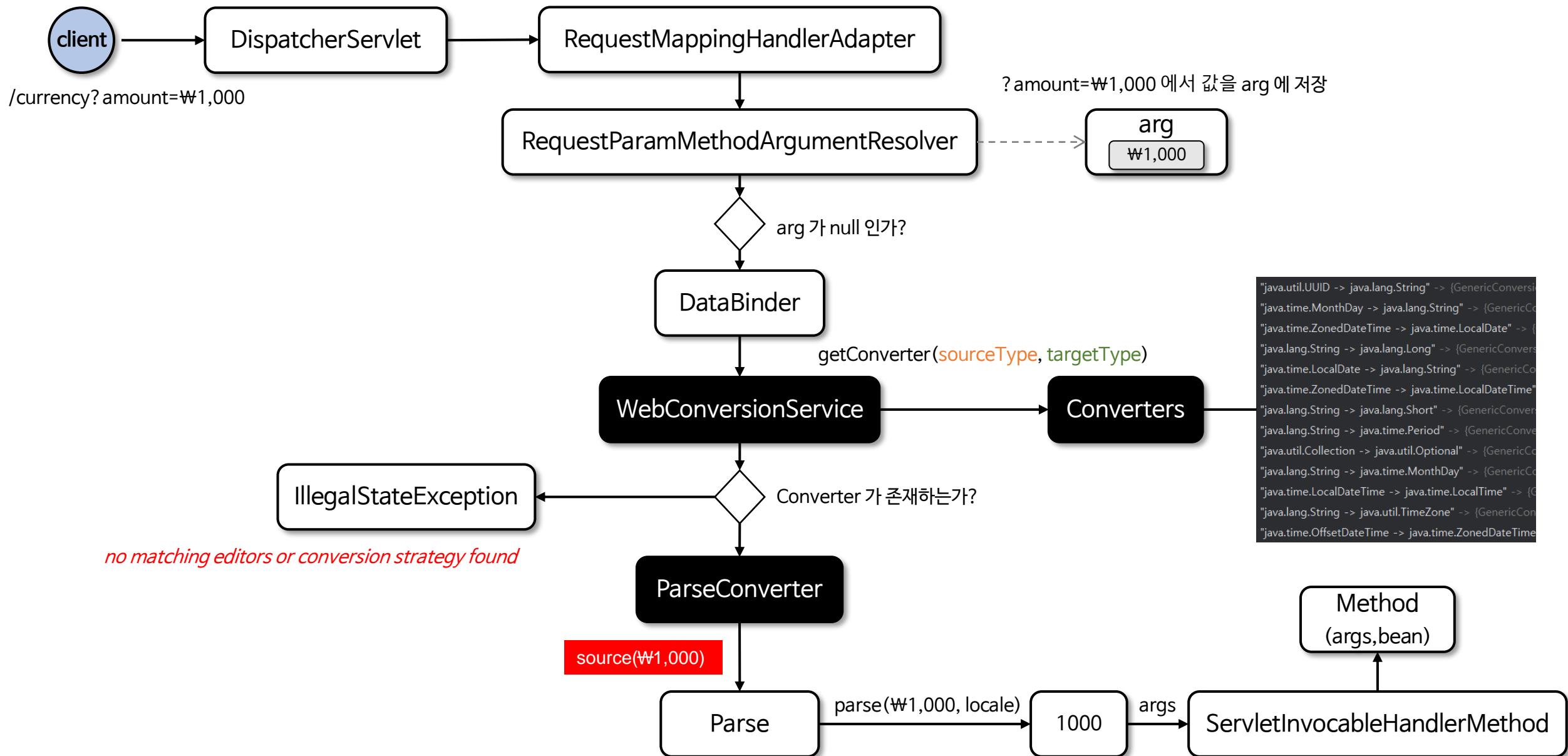
Currency Result

```
Processed Amount: 12345  
Processed Amount: ₩12,345
```

✓ Formatter 흐름도 - Print



✓ Formatter 흐름도 - Parse



어노테이션 기반 포매팅

<https://github.com/onjsdnjs/spring-mvc-master/tree/어노테이션-기반-포매팅>

✓ 개요

- 기본 포맷터는 전체적으로 동일한 형식으로만 동작하기 때문에 각 필드마다 다른 형식을 지정하려는 요구사항을 처리하기 어렵다

new **DateFormatter**("yyyy-MM-dd") ➡

```
public class MyModel {  
    private Date startDate;  
    private Date endDate;  
}
```

startDate=2024-01-01
endDate=2024/12/31 ✗

- 어노테이션 기반 포맷터(AnnotationFormatterFactory)를 활용하면 필드별로 포맷을 다르게 지정할 수 있다

@DateTimeFormat ➡

```
public class MyModel {  
    @DateTimeFormat(pattern = "yyyy-MM-dd")  
    private Date startDate;  
  
    @DateTimeFormat(pattern = "MM/dd/yyyy")  
    private Date endDate;  
}
```

startDate=2024-01-01
endDate=2024/12/31

✓ AnnotationFormatterFactory

- AnnotationFormatterFactory 는 특정 어노테이션이 붙은 필드의 값을 지정된 형식으로 변환해 주는 Formatter 를 생성하는 팩토리 클래스이다
- 예를 들어 Jsr310DateTimeFormatAnnotationFormatterFactory 는 @DateTimeFormat 어노테이션이 붙은 필드에서 날짜(LocalDateTime) 값을 지정된 형식으로 변환해 주는 Formatter 를 만들어 사용할 수 있다

```
public interface AnnotationFormatterFactory<A extends Annotation> {  
  
    Set<Class<?>> getFieldTypes(); // 어노테이션이 적용될 수 있는 필드의 타입 목록을 반환한다  
  
    Printer<?> getPrinter(A annotation, Class<?> fieldType); // 어노테이션이 적용된 필드 값을 출력하기 위한 Printer 객체를 반환한다  
  
    Parser<?> getParser(A annotation, Class<?> fieldType); // 어노테이션이 적용된 필드 값을 파싱하기 위한 Parser 객체를 반환한다  
  
}
```

✓ 주요 구현체

- DateTimeFormatAnnotationFormatterFactory
 - java.util.Date 와 java.util.Calendar 같은 레거시 날짜/시간 API 를 사용하여 @DateTimeFormat 어노테이션이 지정된 필드를 포맷한다
- Jsr310DateTimeFormatAnnotationFormatterFactory
 - JDK 8의 LocalDateTime 와 ZonedDateTime 과 같은 날짜/시간 API 를 사용하여 @DateTimeFormat 어노테이션이 지정된 필드를 포맷한다
- NumberFormatAnnotationFormatterFactory
 - @NumberFormat 어노테이션이 붙은 필드에서 숫자 값을 지정된 형식으로 포맷한다

✓ 기본 구현

```
public class MyModel {  
  
    @NumberFormat(style = NumberFormat.Style.CURRENCY)  
    private BigDecimal price;  
  
    @DateTimeFormat(pattern = "yyyy-MM-dd")  
    private LocalDate date;  
  
    // Getter와 Setter  
}
```

```
@Controller  
public class MyController {  
  
    @GetMapping("/form") // 폼 페이지를 반환  
    public String showForm(Model model) {  
        model.addAttribute("myModel", new MyModel());  
        return "form";  
    }  
  
    @PostMapping("/submit") // 폼 데이터를 처리하고 결과를 반환  
    public String handleSubmit(MyModel myModel, Model model) {  
        model.addAttribute("myModel", myModel);  
        return "result";  
    }  
}
```

form.html

```
<form action="/submit" method="post">  
    <label for="price">Price (Currency):</label>  
    <input type="text" id="price" name="price" />  
    <br>  
    <label for="date">Purchase Date (yyyy-MM-dd):</label>  
    <input type="text" id="date" name="date" />  
    <br>  
    <button type="submit">Submit</button>  
</form>
```

1

Enter Price and Purchase Date

Price (Currency):
Purchase Date (yyyy-MM-dd):

- ① GET /form 요청
- ② 폼 필드에 값 입력
- ③ POST /submit 요청

result.html

```
<body>  
    <h1>Data</h1>  
    <p>Price: <strong>${{myModel.price}}</strong></p>  
    <p>Date: <strong>${{myModel.date}}</strong></p>  
</body>
```

2

Data

Price: ₩1,000.00
Date: 2024-01-01

@NumberFormat과 @DateTimeFormat에 의해 포맷된 데이터 출력

✓ 커스텀 구현

```
public class CustomCurrencyFormatterFactory
    implements AnnotationFormatterFactory<CustomCurrencyFormat> {

    @Override
    public Set<Class<?>> getFieldTypes() {
        Set<Class<?>> fieldTypes = new HashSet<>();
        fieldTypes.add(BigDecimal.class);
        return fieldTypes;
    }

    @Override
    public Printer<BigDecimal> getPrinter(CustomCurrencyFormat annotation, Class<?> fieldType) {
        return new CustomCurrencyFormatter(annotation.pattern(), annotation.decimalPlaces());
    }

    @Override
    public Parser<BigDecimal> getParser(CustomCurrencyFormat annotation, Class<?> fieldType) {
        return new CustomCurrencyFormatter(annotation.pattern(), annotation.decimalPlaces());
    }
}
```

```
public @interface CustomCurrencyFormat {
    String pattern() default "#,###.##"; // 통화 형식
    int decimalPlaces() default 2; // 소수점 자리수
}
```

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldAnnotation(new CustomCurrencyFormatterFactory());
    }
}
```

```
public class CustomCurrencyFormatter implements Formatter<BigDecimal> {

    private final String pattern;
    private final int decimalPlaces;

    public CustomCurrencyFormatter(String pattern, int decimalPlaces) {
        this.pattern = pattern; // 통화 형식
        this.decimalPlaces = decimalPlaces; // 소수점 자리수
    }

    @Override
    public BigDecimal parse(String text, Locale locale) throws ParseException {
        NumberFormat numberFormat = new DecimalFormat(pattern);
        Number number = numberFormat.parse(text);
        return BigDecimal.valueOf(number.doubleValue())
            .setScale(decimalPlaces, RoundingMode.HALF_UP);
    }

    @Override
    public String print(BigDecimal object, Locale locale) {
        NumberFormat format = NumberFormat.getCurrencyInstance(locale);
        currencyFormat.setMaximumFractionDigits(decimalPlaces);
        return currencyFormat.format(object);
    }
}
```

```
@Controller
```

```
public class ItemController {  
  
    @GetMapping("/item")  
    public String showForm(Model model) {  
        model.addAttribute("item", new Item());  
        return "itemForm";  
    }  
  
    @PostMapping("/item")  
    public String handleForm(@ModelAttribute Item item, Model model) {  
        model.addAttribute("item", item);  
        return "itemResult";  
    }  
}
```

```
@Data
```

```
public class Item {  
  
    @CustomCurrencyFormat(pattern = "#,###.##", decimalPlaces = 2)  
    private BigDecimal price;  
  
    @CustomCurrencyFormat(pattern = "#,###.####", decimalPlaces = 3)  
    private BigDecimal discount;  
}
```

itemForm.html

```
<form th:action="@{/item}" method="post">  
    <label for="price">Price:</label>  
    <input type="text" id="price" name="price" th:value="${item.price}"><br><br>  
  
    <label for="discount">Discount:</label>  
    <input type="text" id="discount" name="discount" th:value="${item.discount}"><br><br>  
  
    <button type="submit">Submit</button>  
</form>
```

Product Form

```
Price:   
Discount:   

```

- ① GET /itemForm 요청
- ② 폼 필드에 값 입력
- ③ POST /item 요청

itemResult.html

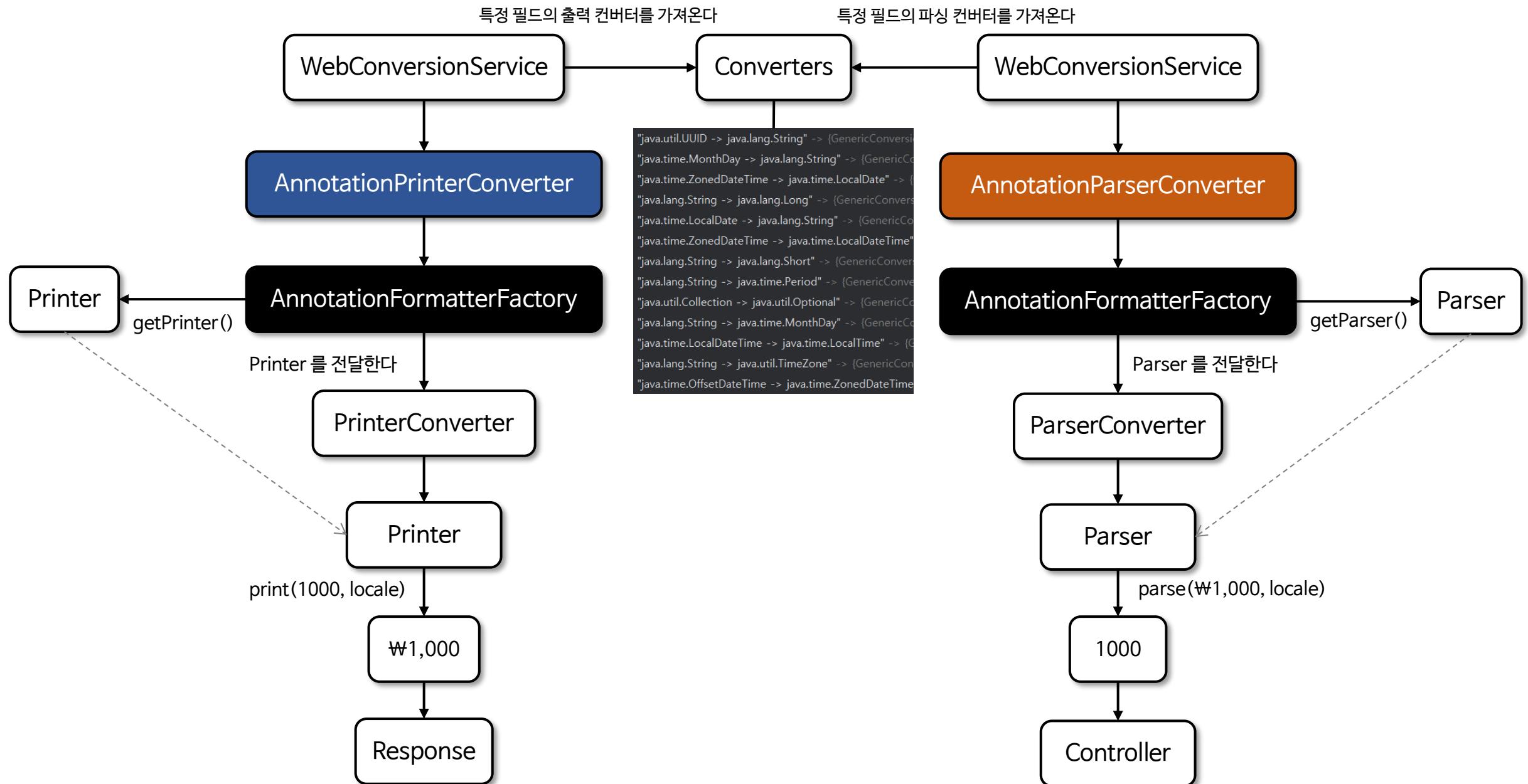
```
<h1>Submitted Item</h1>  
<p><strong>Price:</strong> <span th:text="${item.price}"></span></p>  
<p><strong>Discount:</strong> <span th:text="${item.discount}"></span></p>
```

Submitted Product

```
Price: $123.46  
Discount: $123.456
```

@CustomCurrencyFormat에 의해 포맷된 데이터 출력

✓ 어노테이션 컨버터 흐름





스프링 웹 MVC 완전 정복

✓ 검증 (Validation)

1. 개요
2. 폼 검증(Form Validation)
3. MessageSource 검증
4. Validator 검증
5. Bean Validation
6. 커스텀 검증 어노테이션
7. 바인딩과 검증 관계

개요

✓ 개요

- 웹 개발에서 검증은 클라이언트 검증(Client-side Validation)과 서버 검증(Server-side Validation)으로 구분할 수 있으며 두 검증 모두 사용자의 데이터 입력 오류를 방지하고 보안을 강화하기 위해 중요한 역할을 한다

✓ Client-side Validation (클라이언트 검증)

- 클라이언트 검증은 사용자가 데이터를 입력할 때 브라우저나 클라이언트 측 코드(JavaScript)를 통해 실시간으로 입력된 데이터의 유효성을 확인하는 방법이다
- **장점**
 - 사용자가 폼을 제출하기 전에 오류를 즉시 확인하고 수정할 수 있으며 서버와의 통신 없이 로컬에서 검증이 이루어지므로 속도가 빠르다
 - 사용자는 제출 전 데이터를 바로 확인하고 수정할 수 있어 더 나은 경험을 제공한다
- **단점**
 - 클라이언트 검증만으로는 보안을 보장할 수 없다. 사용자가 브라우저의 개발자 도구를 통해 검증 코드를 우회하거나 비활성화할 수 있기 때문이다.

✓ Server-side Validation (서버 검증)

- 서버 검증은 폼 데이터가 서버로 전송된 후 서버 측 코드(예: PHP, Node.js, Java 등)에서 데이터를 검증하는 방식이다
- **장점**
 - 클라이언트에서 우회된 검증을 차단할 수 있어 보안이 강화되며 서버에서 정확한 검증을 통해 데이터베이스에 잘못된 데이터가 저장되는 것을 방지한다
- **단점**
 - 서버에 요청을 보내고 응답을 받는 과정에서 네트워크 지연이 발생할 수 있으며 사용자는 폼 제출 후 오류 메시지를 받게 되므로 사용자 경험이 다소 떨어질 수 있다

✓ 기존의 Java 검증 방식

- 전통적인 Java에서는 데이터를 검증하기 위해 보통 조건문(if-else)과 예외(Exception)를 사용하여 각 필드에 대해 수동으로 검증 로직을 작성해야 한다
- 복잡한 객체나 컬렉션의 검증을 위해서는 로직이 반복적으로 중첩되거나 별도의 검증 메서드를 생성해야 한다
- 이 방식은 검증 로직의 일관성이 부족하고 코드의 양이 많아 유지보수성이 떨어진다

```
@Override  
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
    String username = request.getParameter("username");  
  
    if (username == null || username.length() < 3) {  
        response.setStatus(HttpServletResponse.SC_BAD_REQUEST);  
        response.getWriter().write("Username must be at least 3 characters long.");  
    } else if (...) {  
        ...  
        ...  
    }  
    } else {  
        // 비즈니스 로직 수행  
        response.setStatus(HttpServletResponse.SC_OK);  
        response.getWriter().write("Data is valid and processed.");  
    }  
}
```

✓ 스프링 검증 방식

- Spring은 다양한 방법과 도구를 통해 클라이언트로부터 입력된 데이터를 유효성 검증하고 데이터 무결성을 보장하고 있으며 어노테이션 기반 검증부터 커스텀 검증 구현까지 유연하고 강력한 검증 기능을 제공한다
- 검증을 위한 도구로서 BindingResult, Errors, FieldError, ObjectError, MessageCodesResolver 등의 클래스를 제공하고 있다

1) 폼 검증 (Form Validation)

- @ModelAttribute로 폼 객체를 바인딩하고 해당 객체의 유효성을 검사하며 검증 오류가 발생하면 BindingResult 객체에 오류 정보가 저장되며 뷰에 오류 메시지를 전달한다
- 스프링의 메시지 소스(MessageSource)를 활용하여 다국어 지원이나 커스텀 오류 메시지를 쉽게 관리할 수 있다

2) Validator 인터페이스를 이용한 검증

- Validator 인터페이스를 구현하여 커스텀 검증 로직을 작성할 수 있으며 이 방법은 어노테이션 기반 검증만으로는 처리할 수 없는 복잡한 검증 로직이 필요할 때 유용하다
- 검증 로직을 별도의 클래스로 분리하여 코드의 재사용성과 확장성을 높일 수 있다

3) Bean Validation

- Bean Validation (JSR-303/JSR-380) 표준을 기반으로 Hibernate Validator가 기본적으로 통합되어 있으며 이 방식은 객체의 필드에 어노테이션을 적용하여 유효성을 검사한다
- 스프링에서는 컨트롤러에서 @Valid 및 @Validated 어노테이션을 사용해 자동으로 검증을 수행할 수 있다

4) 커스텀 검증 어노테이션 (Custom Validation Annotations)

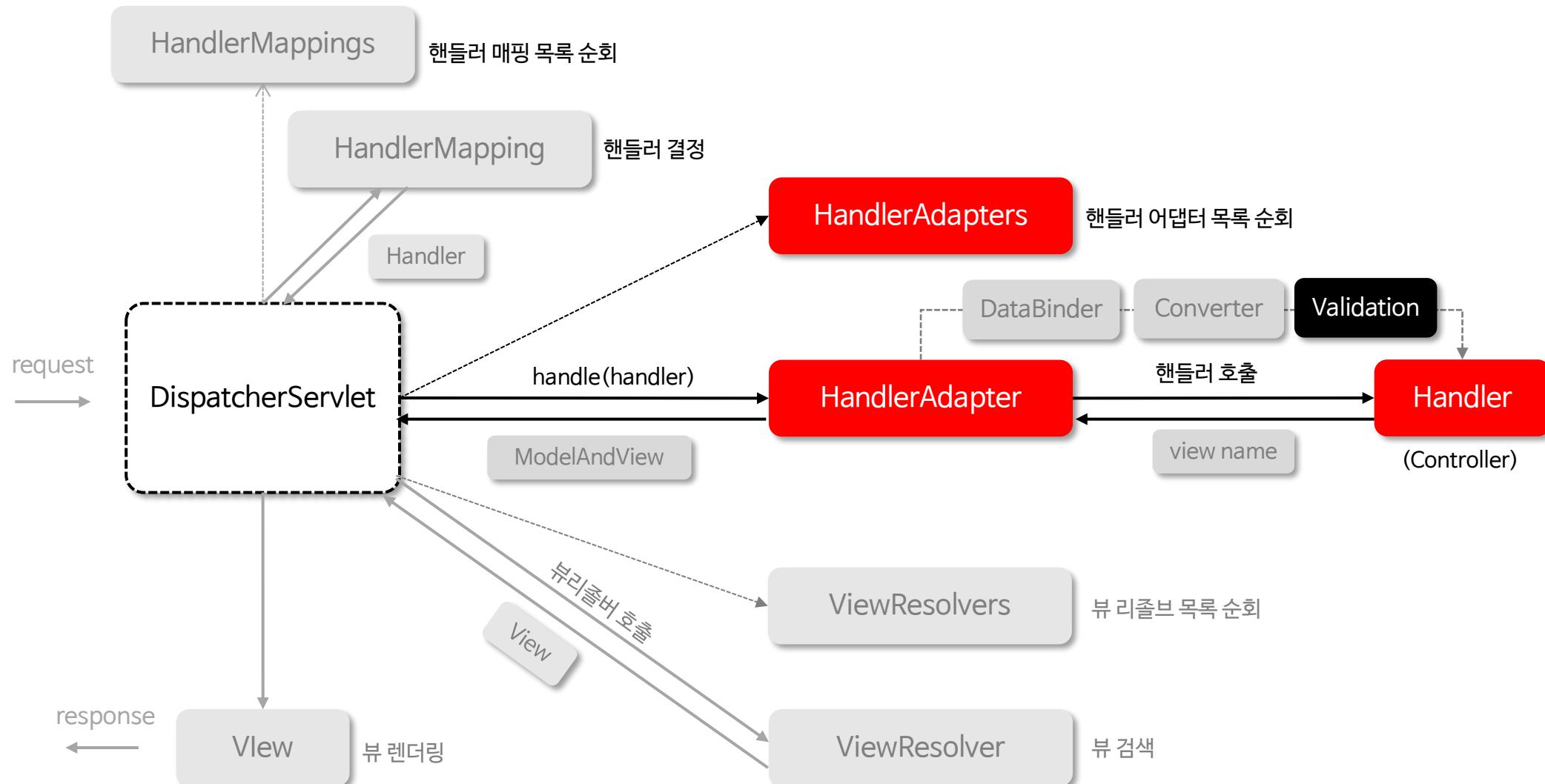
- 스프링은 기본 제공 어노테이션 외에도 커스텀 검증 어노테이션을 구현할 수 있는 기능을 제공하며 ConstraintValidator 인터페이스를 구현하여 적용할 수 있다

폼 검증 (Form Validation)

BindingResult - 1

<https://github.com/onjsdnjs/spring-mvc-master/tree/BindingResult-1>

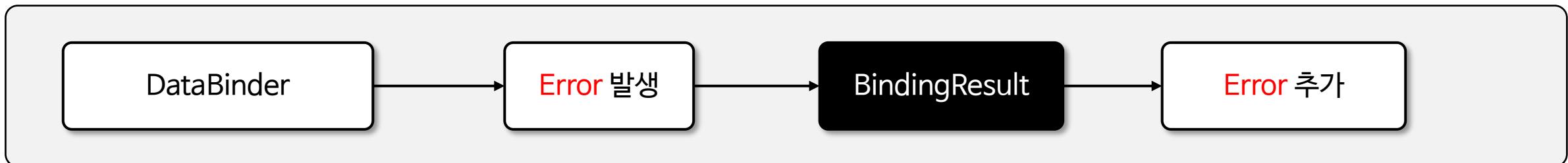
✓ 스프링 MVC 아키텍처



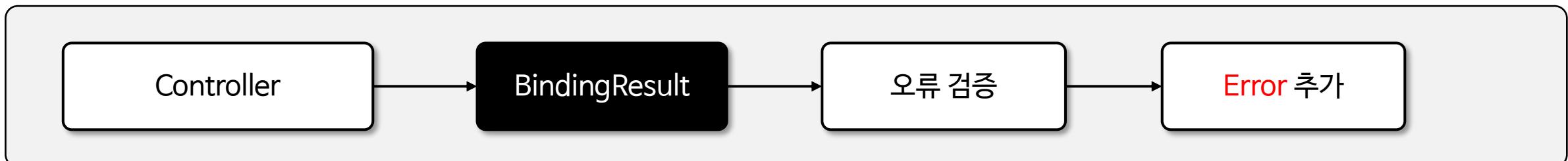
✓ 개요

- 스프링의 검증은 데이터 바인딩 과정과 밀접하게 연관되어 있다. 즉, 데이터 바인딩은 요청 데이터를 객체로 변환하는 과정인데 이 과정에서 데이터를 검증함으로써 어플리케이션의 안정성과 데이터 무결성을 보장하게 된다
- 스프링에서는 크게 두 가지로 구분해서 검증이 이루어진다
 1. 스프링은 데이터 바인딩 시 검증 로직을 자동으로 실행하도록 설계되었으며 BindingResult 통해 오류 정보 및 검증 결과를 저장하고 관리한다
 2. 컨트롤러에서 사용자가 직접 BindingResult를 통해 오류 데이터를 추가하고 검증을 진행할 수 있다.

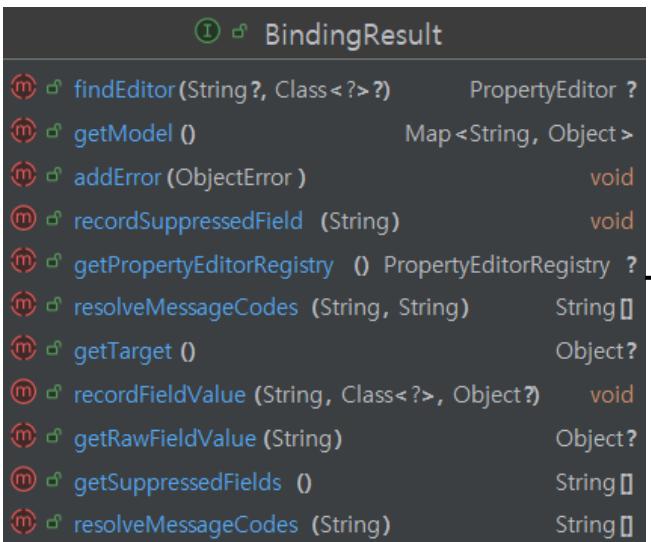
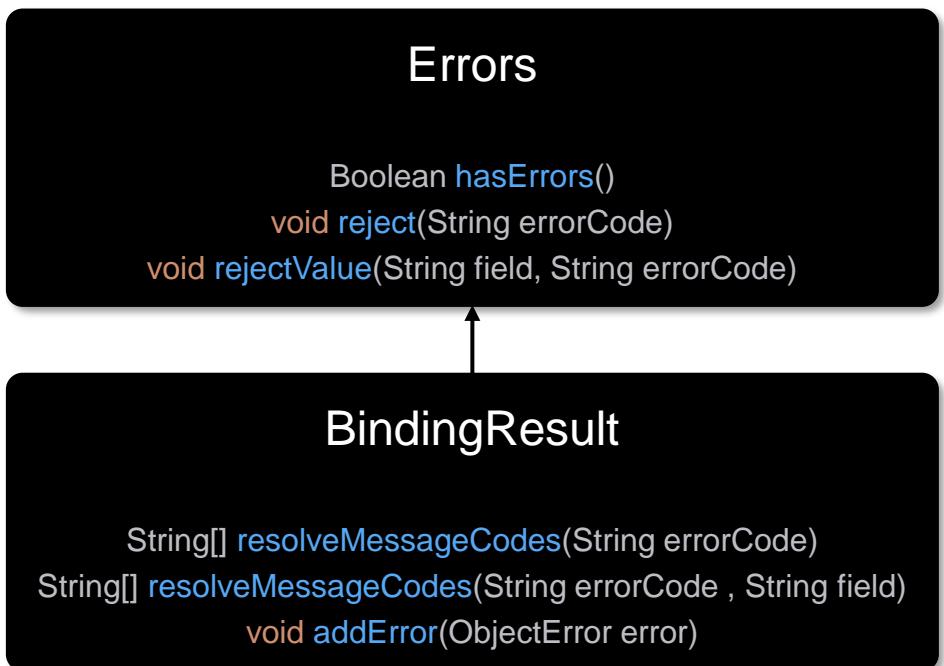
스프링 검증



사용자 검증



✓ Errors / BindingResult 구조



Errors	
m ⚡ reject (String, String)	void
m ⚡ getFieldErrors ()	List <FieldError >
m ⚡ getFieldValue (String)	Object?
m ⚡ toString ()	String
m ⚡ getNestedPath ()	String
m ⚡ popNestedPath ()	void
m ⚡ getErrorCount ()	int
m ⚡ hasErrors ()	boolean
m ⚡ setNestedPath (String)	void
m ⚡ addAllErrors (Errors)	void
m ⚡ getObjectName ()	String
m ⚡ getFieldErrorCount ()	int
m ⚡ getAllErrors ()	List <ObjectError >
m ⚡ hasFieldErrors (String)	boolean
m ⚡ getFieldType (String)	Class<?>?
m ⚡ getGlobalErrorCount ()	int
m ⚡ reject (String)	void
m ⚡ getFieldErrors (String)	List <FieldError >
m ⚡ rejectValue (String ?, String, String)	void
m ⚡ getFieldError (String)	FieldError?
m ⚡ hasFieldErrors ()	boolean
m ⚡ rejectValue (String ?, String, Object[], String?)	void
m ⚡ getGlobalError ()	ObjectError?
m ⚡ rejectValue (String ?, String)	void
m ⚡ pushNestedPath (String)	void
m ⚡ getGlobalErrors ()	List <ObjectError >
m ⚡ hasGlobalErrors ()	boolean
m ⚡ failOnError(Function<String, T>)	void
m ⚡ reject (String, Object[], String?)	void
m ⚡ getFieldError ()	FieldError?
m ⚡ getFieldErrorCount (String)	int

1. Errors

- 검증 과정에서 기본적인 오류를 추가하고 확인하는 인터페이스이다

2. BindingResult

- 컨트롤러에서 데이터 바인딩과 검증을 동시에 처리해야 하는 상황에서 주로 사용된다
- 메시지 코드 해석과 세부적인 오류 관리 기능 등 보다 정교하고 확장된 기능을 제공한다

✓ BindingResult 기본 전략

- 스프링의 BindingResult는 세 가지 기본 전략을 가진다

1) 스프링은 데이터 바인딩 시 발생하는 모든 오류 정보를 자동으로 BindingResult의 errors 속성에 저장 한다

2) 사용자가 BindingResult의 오류 정보를 활용하기 위해서는 컨트롤러 메서드 매개변수로 지정 해야 한다

- 매개변수로 지정 시 BindingResult는 @ModelAttribute 객체 바로 뒤에 위치해야 한다. 아래 코드에서 BindingResult는 user 객체 바로 다음에 와야 한다

```
public String method(@ModelAttribute User user, BindingResult bindingResult) { . . . }
```

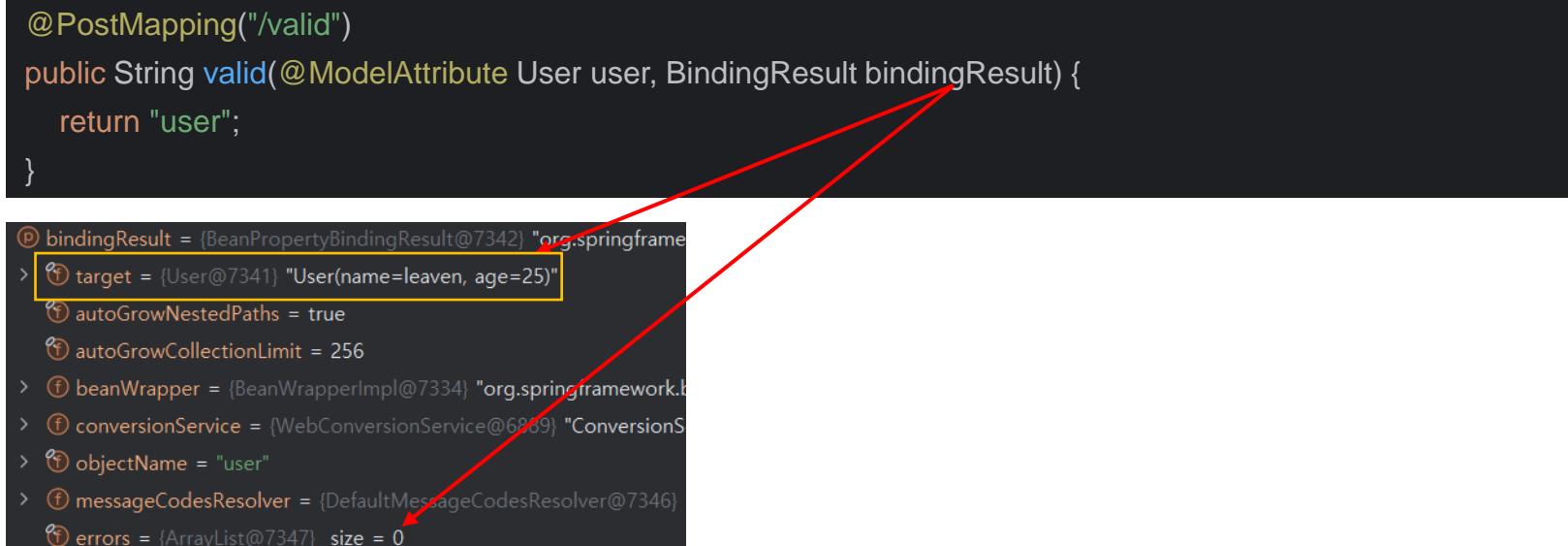
- 매개변수로 지정하게 되면 객체 바인딩 오류가 나서 예외가 발생하더라도 컨트롤러는 정상으로 실행된다

3) BindingResult API를 사용해서 추가적인 검증을 진행하거나 검증 결과를 클라이언트에게 전달할 수 있다.

✓ BindingResult과 바인딩 객체

- DataBinder에 의해 바인딩이 완료되면 BindingResult는 바인딩 된 객체를 저장한다. 그리고 errors 속성은 비어 있는 상태다

```
@PostMapping("/valid")
public String valid(@ModelAttribute User user, BindingResult bindingResult) {
    return "user";
}
```



```
bindingResult = {BeanPropertyBindingResult@7342} "org.springframework.b
> target = {User@7341} "User(name=leaven, age=25)"
  autoGrowNestedPaths = true
  autoGrowCollectionLimit = 256
> beanWrapper = {BeanWrapperImpl@7334} "org.springframework.b
> conversionService = {WebConversionService@6289} "ConversionS
  objectName = "user"
> messageCodesResolver = {DefaultMessageCodesResolver@7346}
  errors = {ArrayList@7347} size = 0
```

✓ BindingResult 기본

1. 바인딩 오류 발생 : BindingResult 를 메서드에 선언하지 않는 경우

- 1) 스프링은 MethodArgumentNotValidException 예외를 발생시키고 요청한 컨트롤러는 실행되지 않는다.
- 2) 가장 기본이 되는 바인딩 오류는 타입 불일치이다 즉. 요청 데이터와 필드 타입이 서로 맞지 않아 바인딩이 실패하는데 이때 내부적으로 TypeMismatchException 예외가 발생한다

```
@PostMapping("/nonValidation")
public String nonValidation (@ModelAttribute Member member) {
    return "member";
}
```

```
@Data
public class Member {
    private String name;
    private int age;
    private int height;
}
```

BindingResult 를 선언하지 않은 상태에서 오류가 발생하면 스프링은 예외를 던지고 요청을 중단한다

타입 불일치로 바인딩 실패

```
POST http://localhost:8080/nonValid
Content-Type: application/x-www-form-urlencoded

name=leaven&age=w25&height=33
```

✓ BindingResult 기본

2. 바인딩 오류 발생 : BindingResult 를 메서드에 선언한 경우

- 1) MethodArgumentNotValidException 예외를 건너뛰고 컨트롤러 메서드가 호출된다. 이유는 간단하다. BindingResult 를 메서드에 선언했기 때문이다.
- 2) 스프링은 BindingResult 의 메서드 선언 여부에 따라 요청을 계속 진행시킬지 아니면 MethodArgumentNotValidException 예외로 요청을 중단할지 결정한다

```
@PostMapping("/validation")
public String validation(@ModelAttribute Member member, BindingResult bindingResult) {
    return "member";
}
```

```
@Data
public class Member {
    private String name;
    private int age;
    private int height;
}
```

타입 불일치로 바인딩 실패

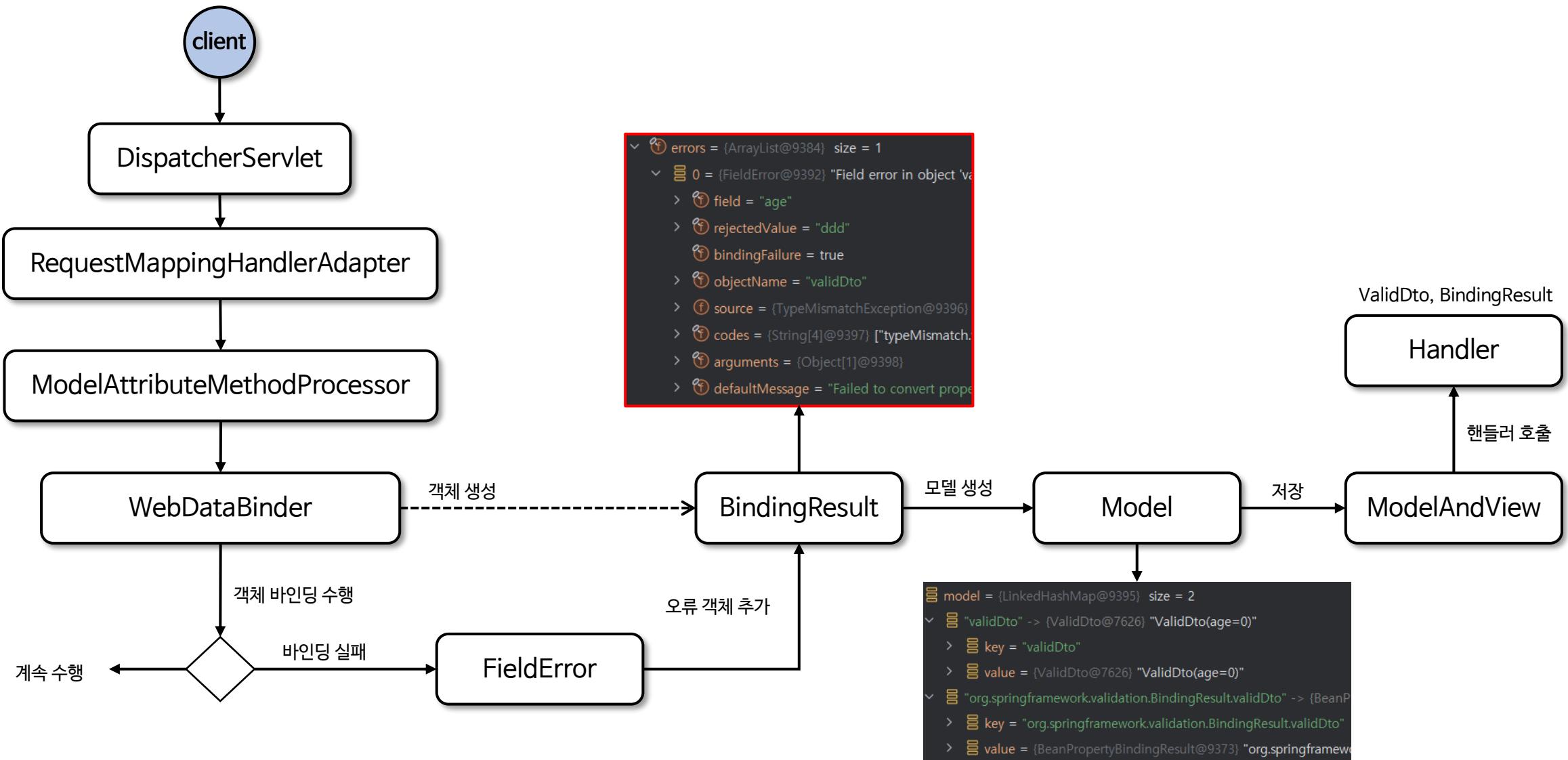
```
POST http://localhost:8080/nonValid
Content-Type: application/x-www-form-urlencoded

name=leaven&age=w25&height=33
```

```
bindingResult = {BeanPropertyBindingResult@6886} "org.springframework.validation.BeanPropertyBindingResult: 1 errors"
  target = {User@6635} "User(name=leaven, age=0)"
  autoGrowNestedPaths = true
  objectName = "user"
  messageCodesResolver = {DefaultMessageCodesResolver@6891}
  errors = {ArrayList@6892} size = 1 ... View
    0 = {FieldError@6902} "Field error in object 'user' on field 'age': rejected value [w25]; codes [typeMismatch.user.age,typeMismatch.age,typeMismatch.java.lang.Integer]; arguments [{}]; defaultMessage = Failed to convert property value of type 'java.lang.String' to required type 'int' for property 'age'; For input string: \"w25\""
      field = "age"
      rejectedValue = "w25"
      bindingFailure = true
      objectName = "user"
      source = {TypeMismatchException@6906} "org.springframework.beans.TypeMismatchException: Failed to convert property value of type 'java.lang.String' to required type 'int' for property 'age'; For input string: \"w25\""
      codes = {String[4]@6907} ["typeMismatch.user.age", "typeMismatch.age", "typeMismatch.integer", "typeMismatch"]
      arguments = {Object[1]@6908} ...
      defaultMessage = Failed to convert property value of type 'java.lang.String' to required type 'int' for property 'age'; For input string: \"w25\""
```

BindingResult 가 메서드에 선언된 상태에서 오류가 발생하면 스프링은 예외를 건너뛰고 요청을 계속 진행시킨다

✓ 바인딩 오류 흐름도



BindingResult - 2

<https://github.com/onjsdnjs/spring-mvc-master/tree/BindingResult-2>

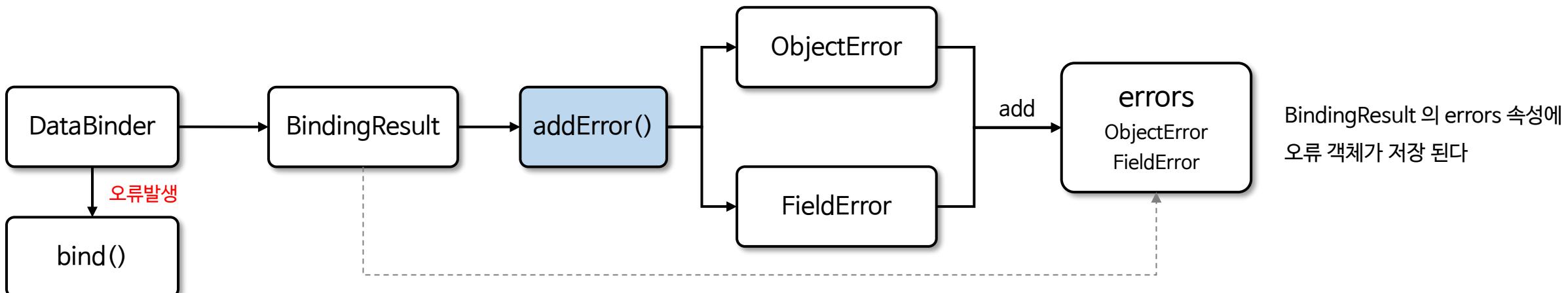
✓ 개요

- BindingResult 은 문자 그대로 바인딩한 결과를 담는 객체라 할 수 있다. 즉 성공한 바인딩 결과와 실패한 오류 결과를 저장한다(두 결과를 동시에 저장할 수는 없다)
- 우리의 주 관심사는 오류 결과이며 그 오류 정보에 어떻게 접근할 수 있는지와 그리고 어떻게 커스텀하게 오류를 추가 할 수 있는지 관련 API 들을 살펴 보는 것이다
- 참고로 BindingResult 는 @ModelAttribute 가 선언되어 있는 객체를 대상으로 오류를 검증한다. 단순 기본형이나 문자열은 검증 대상이 아니다. 이것은 DataBinder 가 객체를 대상으로 바인딩을 시도한다는 사실을 알고 있다면 쉽게 이해가 갈 것이다

✓ BindingResult API

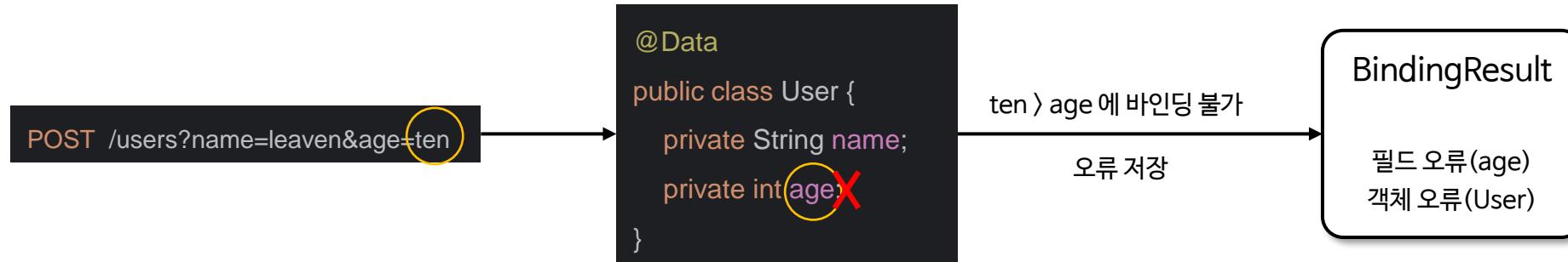
void addError(ObjectError error);

- addError() 는 필수 값 누락, 길이 제한 등 어떤 조건이 맞지 않을 경우 오류를 추가할 수 있는 API 로서 인자 값으로는 ObjectError 와 FieldError 객체가 올 수 있다
- 스프링은 바인딩 오류 시 내부적으로 BindingResult 의 addError() API 를 사용해서 오류 정보를 저장하고 있으며 이 때 FieldError 객체를 생성해서 전달한다
- addError() API 를 통해 추가된 FieldError, ObjectError 는 BindingResult 의 errors 속성에 저장 된다



✓ 객체 오류(글로벌 오류) 와 필드 오류

- 스프링은 오류를 추가할 때 객체 오류(or 글로벌 오류) 와 필드 오류로 구분하도록 API 를 설계했다
- 객체 오류라는 것은 말 그대로 객체 수준에서 오류를 표현한다는 의미이고 필드 오류라는 것은 객체 보다 좀 더 구체적인 필드 수준에서 오류를 표현한다는 의미이다
- 오류는 사용자나 클라이언트에게 이해하기 쉬운 문장으로 설명해야 되며 상황에 맞게끔 구체적인 오류와 종합적인 오류를 잘 조합해서 나타내야 한다



필드 수준 오류

- 구체적으로 어떤 객체의 어떤 필드에서 오류가 났는지를 표현한다
- “사용자 나이는 숫자만 가능합니다.”, “사용자 나이 입력 중 오류가 발생했습니다.”.

→ *FieldError*

객체 수준 오류

- 필드가 아닌 객체 혹은 전역 수준에서의 오류 사항을 표현한다
- “사용자 정보가 맞지 않습니다”, “숫자만 가능합니다”

→ *ObjectError*

✓ 오류 객체 사용

- ObjectError 와 FieldError 는 생성자를 통해 오류 정보를 전달받는다

ObjectError

```
public ObjectError (String objectName, String defaultMessage)
```

FieldError

```
public FieldError (String objectName, String field, String defaultMessage)
```

objectName 바인딩 객체 이름(@ModelAttribute 이거나 null)
defaultMessage 오류 기본 메시지

objectName 바인딩 객체 이름(@ModelAttribute 이거나 null)
field 오류 필드명
defaultMessage 오류 기본 메시지

✓ 기본 구현

```
POST http://localhost:8080/users
```

Content-Type: application/x-www-form-urlencoded

name=&age=ten

```
@PostMapping("/users")  
public String registerUser(@ModelAttribute("user") User user, BindingResult bindingResult) {  
    if (!StringUtils.hasText(user.getName())) {  
        bindingResult.addError(new FieldError("user", "name", "사용자 이름은 필수입니다."));  
    }  
    if (!StringUtils.hasText(user.getName()) && (user.getAge() < 1)) {  
        bindingResult.addError(new ObjectError("user", "사용자 입력 값은 모두 필수입니다."));  
    }  
    return "user";  
}
```

BindingResult

```
(errors = [ArrayList@8197] size = 3... View  
> 0 = {FieldError@8206} "Field error in object 'user' on field 'name'  
<--> 1 = {FieldError@8207} "Field error in object 'user' on field 'age'  
<--> field = "username"  
<--> rejectedValue = null  
<--> bindingFailure = false  
> 2 = {ObjectError@8208} "Error in object 'user': codes []; arguments []  
<--> objectName = "user"  
<--> source = null  
<--> codes = null  
<--> arguments = null  
<--> defaultMessage = "사용자 이름은 필수입니다."  
> 3 = {ObjectError@8208} "Error in object 'user': codes []; arguments []  
<--> objectName = "user"  
<--> source = null  
<--> codes = null  
<--> arguments = null  
<--> defaultMessage = "사용자 입력 값은 모두 필수입니다."
```

✓ 구현 예제

```
@Controller
public class OrderController {

    @GetMapping("/order")
    public String showOrderForm(Model model) {
        model.addAttribute("order", new Order());
        return "orderForm";
    }

    @PostMapping("/order")
    public String submitOrder(@ModelAttribute("order") Order order, BindingResult result) {

        // 필드 오류 검증 (productName)
        if (!StringUtils.hasText(order.getItem())) {
            result.addError(new FieldError("order", "item", "상품명은 필수입니다."));
        }

        // 필드 오류 검증 (quantity)
        if (order.getQuantity() <= 0) {
            result.addError(new FieldError("order", "quantity", "수량은 0보다 커야 합니다."));
        }

        // 객체 수준 오류
        if (!StringUtils.hasText(order.getItem()) && (order.getQuantity() <= 0)) {
            result.addError(new ObjectError("order", "유효하지 않은 상품명입니다."));
        }

        if (result.hasErrors()){
            return "orderForm"; // 오류가 있으면 주문 폼으로 돌아가기
        }
        return "orderResult";
    }
}
```

```
@Data
public class Order {
    private String item;
    private int quantity;
}
```

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head><title>주문 입력</title></head>
<body>
<h1>주문 입력</h1>

<form th:action="@{/order}" th:object="${order}" method="post">
    <label>상품명: <input type="text" th:field="*{item}"/></label>
    <span th:if="#{fields.hasErrors(item)}" th:errors="*{item}">상품명 오류</span><br/>

    <label>수량: <input type="number" th:field="*{quantity}"/></label>
    <span th:if="#{fields.hasErrors('quantity')}" th:errors="*{quantity}">수량 오류</span><br/>

    <!-- 객체 수준 오류 -->
    <div th:if="#{fields.hasGlobalErrors()}">
        <p th:each="error : #{fields.globalErrors()}" th:text="${error}">글로벌 오류</p>
    </div>

    <button type="submit">제출</button>
</form>
</body>
</html>
```

1. BindingResult는 자동으로 모델에 추가되며 Thymeleaf에서 #fields를 통해 접근할 수 있다
2. #fields는 BindingResult를 감싸고 있는 Thymeleaf 유ти리티 객체이며 BindingResult의 필드 및 글로벌 오류에 쉽게 접근할 수 있다
3. <https://www.thymeleaf.org/apidocs/thymeleaf-spring6/3.1.2.RELEASE/org/thymeleaf/spring6/expression/Fields.html>

BindingResult – 3

<https://github.com/onjsdnjs/spring-mvc-master/tree/BindingResult-3>

✓ 개요

- FieldError 및 ObjectError 클래스는 두 개의 생성자가 있으며 이 두 생성자는 오류의 상세 정도에 따라 사용되며 두 번째 생성자는 보다 세부적인 오류 정보를 포함한다
- 첫 번째 생성자는 객체 이름, 필드 이름, 그리고 기본 오류 메시지를 사용해 FieldError를 생성한다. 이때 거부된 값(rejectedValue)이나 다른 상세 정보(바인딩 실패 여부, 메시지 코드 등)는 포함하지 않는다

```
bindingResult.addError(new FieldError("order", "quantity", "수량은 0보다 커야 합니다."));
```

간단하게 필드에 대한 오류 메시지만 추가할 때 사용한다

✓ 두번째 생성자 API

ObjectError

```
public ObjectError (String objectName, String[] codes, Object[] arguments, String defaultMessage)
```



FieldError

```
FieldError(String objectName, String field, Object rejectedValue, boolean bindingFailure, String[] codes, Object[] arguments, String defaultMessage)
```

- ① **objectName**: 오류가 발생한 객체 이름
- ② **field**: 오류가 발생한 필드 이름
- ③ **rejectedValue**: 클라이언트가 입력한 잘못된 값
- ④ **bindingFailure**: 데이터 바인딩 실패 여부 (true면 바인딩 실패, 스프링에서 바인딩 하다가 난 오류인지 아닌지 구분할 수 있다)
- ⑤ **codes**: 오류 코드 목록 (메시지 소스에서 사용)
- ⑥ **arguments**: 메시지에 사용될 인자 목록
- ⑦ **defaultMessage**: 기본 오류 메시지

✓ 구현 예제

- 사용자가 입력한 내용이 잘못되어 바인딩 오류가 발생하면 서버는 클라이언트에게 오류 정보를 전달해서 화면에 보여준다
- 문제는 기존의 사용자 화면에 잘못 입력한 내용까지 다 사라지게 된다는 것이다. FieldError 객체의 rejectedValue 속성을 활용하면 이 문제를 해결할 수 있다

```
@PostMapping("/order")
public String submitOrder(@ModelAttribute("order") Order order, BindingResult bindingResult) {
    if (!StringUtils.hasText(order.getProductName())) {
        bindingResult.addError(new FieldError("order", "productName", order.getProductName(),
            false, null, null, "상품명은 비워둘 수 없습니다."));
    }

    if (order.getQuantity() <= 0) { // 수량 검증
        bindingResult.addError(new FieldError("order", "quantity", order.getQuantity(),
            false, null, null, "수량은 0보다 커야 합니다."));
    }

    if (order.getPrice() < 1 || order.getPrice() > 1_000_000) { // 가격 범위 제한: 1 ~ 1,000,000
        bindingResult.addError(new FieldError("order", "price", order.getPrice(),
            false, null, null, "가격은 1 이상 1,000,000 이하여야 합니다."));
    }

    if (order.getQuantity() > 100 && order.getPrice() < 1000) { // 객체 수준 오류
        bindingResult.addError(new ObjectError(
            "order", "수량이 많을 때는 가격이 1000 이상이어야 합니다."));
    }

    if (bindingResult.hasErrors()) {
        return "orderForm";
    }

    return "orderForm";
}
```

```
@Data
public class Order {
    private String productName;
    private int quantity;
    private int price;
}
```

- rejectedValue는 사용자가 화면에 입력한 값을 보관하고 있으며 이 값을 화면에 보여줄 수 있다
- 빨간 테두리 박스를 보면 타임리프에서 th:field="*{…}"으로 되어 있는데 다음과 같이 동작한다
 - 정상 값 : th:field="*{…}"는 모델 객체의 값을 사용한다
 - 오류 발생 시 : th:field는 자동으로 FieldError의 rejectedValue를 사용한다

```
<form th:action="@{/order}" th:object="${order}" method="post">
    <!-- 글로벌 오류 (ObjectError) 출력 -->
    <div th:if="#{fields.hasGlobalErrors()}">
        <p th:each="error : ${fields.globalErrors()}" th:text="${error}">글로벌 오류 메시지</p>
    </div>
    <!-- 상품명 -->
    <label>상품명: <input type="text" th:field="*{productName}" /></label>
    <span th:if="#{fields.hasErrors('productName')}" th:errors="*{productName}">상품명 오류</span><br/>

    <!-- 수량 -->
    <label>수량: <input type="number" th:field="*{quantity}" /></label>
    <span th:if="#{fields.hasErrors('quantity')}" th:errors="*{quantity}">수량 오류</span><br/>

    <!-- 가격 -->
    <label>가격: <input type="text" th:field="*{price}" /></label>
    <span th:if="#{fields.hasErrors('price')}" th:errors="*{price}">가격 오류</span><br/>

    <button type="submit">제출</button>
</form>
```

BindingResult 와 MessageSource 연동 - 1

<https://github.com/onjsdnjs/spring-mvc-master/tree/MessageSource-연동-1>

✓ 개요

- DataBinder 의 바인딩 시 발생한 오류나 BindingResult 의 유효성 검증 오류가 발생했을 때 MessageSource 를 사용해서 해당 오류메시지를 사용자에게 제공할 수 있다
- 이 방식은 유효성 검증에 필요한 오류 메시지를 외부 파일(properties 파일 등)에서 검색 및 관리할 수 있다. 즉 오류 메시지를 MessageSource 에 위임한다고 보면 된다

✓ MessageSource 오류 메시지

FieldError

```
FieldError(String objectName, String field, Object rejectedValue, boolean bindingFailure, String[] codes, Object[] arguments, String defaultMessage)
```

- ① **codes**: 오류 코드 목록 (메시지 소스에서 사용)
- ② **arguments**: 메시지에 사용될 인자 목록

오류 메시지 직접 입력

```
bindingResult.addError(new FieldError("order", "productName", "상품명은 필수입니다."));
```



오류 메시지 외부 연동

```
bindingResult.addError(new FieldError("order", "productName", new String[] {"required.order.item"}))
```

오류 코드 전달

MessageSource

오류 메시지 적용

오류 코드 검색

```
new FieldError("order", "item", "상품명은 필수입니다.");
```

required.order.item = 상품명은 필수입니다.

validation.properties

✓ 구현 예제

application.properties

```
spring.messages.basename=messages,validation
```

```
@PostMapping("/order")
public String submitOrder(@ModelAttribute("order") Order order, BindingResult bindingResult) {
    if (!StringUtils.hasText(order.getProductName())) {
        bindingResult.addError(new FieldError("order", "item", null, false,
            new String[]{"required.order.item"}, null, null));
    }
    if (order.getPrice() < 100 || order.getPrice() > 10000) {
        bindingResult.addError(new FieldError("order", "price", order.getPrice(), false,
            new String[]{"range.order.price"}, new Object[]{100, 10000}, " "));
    }
    if (bindingResult.hasErrors()) {
        return "orderForm";
    }
    return "orderResult";
}
```

validation.properties

```
range.order.price=가격은 {0} 이상 {1} 이하여야 합니다.
required.order.item=상품명은 필수입니다.
```

```
@Data
public class Order {
    private String item;
    private int price;
}
```

```
<form th:action="@{/order}" th:object="${order}" method="post">
    <!-- 상품명 입력 -->
    <label>상품명: <input type="text" th:field="*{item}" /></label>
    <span th:if="#{fields.hasErrors(item)}" th:errors="*{item}">상품명 오류</span><br/>

    <!-- 가격 입력 -->
    <label>가격: <input type="number" th:field="*{price}" /></label>
    <span th:if="#{fields.hasErrors('price')}" th:errors="*{price}">가격 오류</span><br/>

    <button type="submit">제출</button>
</form>
```

BindingResult 와 MessageSource 연동 - 2

<https://github.com/onjsdnjs/spring-mvc-master/tree/MessageSource-연동-2>

✓ 개요

- ObjectError 또는 FieldError API를 사용해서 오류 코드를 기입하는 방식은 세밀한 제어는 가능하나 번거롭고 관리적인 측면에서 간단하지 않다
- BindingResult에는 FieldError를 사용하지 않고 오류 코드를 자동화하고 광범위하게 MessageSource와 연동하는 API를 제공하고 있다

✓ reject() / rejectValue()

```
void reject(String errorCode, Object[] errorArgs, String defaultMessage);
```

객체 오류

```
void rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage);
```

필드 오류

- **field**: 오류 필드 이름
- **errorCode**: 오류 코드 (메시지 소스에서 사용)
- **errorArgs**: 메시지에 사용될 인자 목록
- **defaultMessage**: 기본 오류 메시지

✓ 기본 구현

```
@PostMapping("/order")
public String submitOrder(@ModelAttribute("order") Order order, BindingResult bindingResult) {
    if (order.getPrice() < 100 || order.getPrice() > 10000) {
        bindingResult.rejectValue("price", "range", new Object[]{100, 10000});
    }
}
```

- rejectValue의 인자를 보면 필드명 "price" 앞에 객체명인 "order"가 빠져 있는데 사실 BindingResult는 바인딩 과정에서 객체 정보인 target을 저장하기 때문에 객체 정보를 입력하지 않아도 참조 가능하다

✓ 구현 예제

```
@PostMapping("/order")
public String submitOrder(@ModelAttribute("order") Order order, BindingResult bindingResult) {
    if (!StringUtils.hasText(order.getItem())) {
        bindingResult.rejectValue("item", "required", null, "상품명은 필수입니다");
    }
    if (order.getPrice() < 100 || order.getPrice() > 10000) {
        bindingResult.rejectValue("price", "range", new Object[]{100, 10000},
            "가격은 100 이상 10000 이하여야 합니다.");
    }
    if (bindingResult.hasErrors()) {
        return "orderForm";
    }
    return "orderResult";
}
```

validation.properties

```
range.order.price=가격은 {0} 이상 {1} 이하여야 합니다.
required.order.item=상품명은 필수입니다
```

```
@Data
public class Order {
    private String item;
    private int price;
}
```

```
<form th:action="@{/order}" th:object="${order}" method="post">
    <!-- 상품명 입력 -->
    <label>상품명: <input type="text" th:field="*{item}" /></label>
    <span th:if="${#fields.hasErrors(item')}" th:errors="*{item}">상품명 오류</span><br/>

    <!-- 가격 입력 -->
    <label>가격: <input type="number" th:field="*{price}" /></label>
    <span th:if="${#fields.hasErrors('price')}" th:errors="*{price}">가격 오류</span><br/>

    <button type="submit">제출</button>
</form>
```

- 위 예제를 실행해 보면 정상적으로 동작하고 클라이언트 화면에도 정확하게 오류 메시지를 보여준다
- 근데 한가지 이상한 점이 있다. `rejectValue`에 설정한 오류 코드 “range” 와 메시지 파일에 입력한 오류 코드 “range.order.price”의 값이 서로 일치하지 않는데도 정확하게 오류 코드를 찾아 주고 있다는 점이다
- 그 이유는 `BindingResult`가 내부적으로 오류 코드와 관련된 특별한 작업을 수행하고 있는데 바로 `MessageCodesResolver` 클래스가 그 해결책을 가지고 있다

✓ MessageCodesResolver

- MessageCodesResolver 는 검증 오류 발생 시 오류 메시지의 메시지 코드를 생성하는 인터페이스이다
- 유효성 검증 시 필드 오류 또는 글로벌 오류가 발생하면 이 오류들을 MessageSource 와 연동하여 해당 오류 메시지를 찾기 위한 메시지 코드 목록을 생성한다

✓ 오류 메시지 전략

- 일반적으로 어떤 기능의 오류 메시지를 표현 할 때는 범용적인 것과 세부적인 것들을 잘 구분해서 설정하게 된다.

required=필수입니다 max=초과할 수 없습니다. min=최소 사항입니다.	범용적인 메시지
required.int=숫자 입력입니다. max.String=최대값은 문자만 가능합니다. min.int=최소값은 숫자만 가능합니다.	타입 제한 메시지
required.user=사용자 정보는 필수입니다 max.order=주문량을 초과할 수 없습니다. min.order=최소 주문량을 만족해야 합니다.	객체 수준의 메시지
required.name=이름은 필수입니다 max.item=상품명이 너무 깁니다 min.price=최소 가격을 만족해야 합니다.	필드 수준 메시지
required.user.name=사용자 이름은 필수입니다 max.order.item=주문 하신 상품명이 너무 깁니다. min.order.price=주문 시 최소 가격을 만족 해야 합니다.	객체 + 필드 수준 메시지

- ① 왼쪽을 보면 범용적인 것 부터 세부적인 것 순으로 오류 메시지를 설정하고 있다
- ② 이 메시지 외에도 필요할 경우 원하는 대로 추가할 수 있고 재활용이 얼마든지 가능하다
- ③ 만약 필드에 대한 구체적인 오류 메시지를 표현해야 한다면 맨 아래 설정이 필요할 것이고 단순히 객체에 대한 것이나 숫자 혹은 문자와 같은 타입과 관련된 오류 메시지를 표현해야 한다면 두번째와 세번째 설정이 필요할 것이다.
- ④ 왼쪽은 5 가지 정도로 구분했지만 이 보다 더 많이 혹은 작게 구분할 수도 있다
- ⑤ 스프링은 왼쪽과 같이 오류 메시지를 표현할 수 있도록 객체 기준 2개, 필드 기준 4개로 구분해서 오류코드 목록을 생성 해준다
- ⑥ 즉 reject() 혹은 rejectValue() API 가 실행되면 내부적으로 MessageCodesResolver 가 오류 코드를 생성하고 그 오류 코드를 MessageSource 가 참조해서 오류 메시지를 검색한다

✓ 구조

```
public interface MessageCodesResolver {  
    String[] resolveMessageCodes(String errorCode, String objectName); // 오류코드, 객체이름  
    String[] resolveMessageCodes(String errorCode, String objectName, String field, Class<?> fieldType); // 오류코드, 객체이름, 필드이름, 필드타입  
}
```

- MessageCodesResolver 는 객체 오류인 경우 두 가지, 필드 오류인 경우 네 가지 형식으로 오류 코드를 생성한다
- 각 코드 형식은 우선순위에 따라 적용되는데 가장 구체적인 규칙부터 범용적인 규칙 순으로 찾고 적용된다
- 스프링은 기본 구현체인 DefaultMessageCodesResolver 를 제공하고 있다

1. 객체 수준 오류 코드 생성

- ① code + "." + objectName
- ② code

예) 코드가 "required", 객체명이 "user" 인 경우



- ① required.user → 특정 객체에 대한 메시지
- ② required → 일반적인 오류 메시지

메시지를 찾는 순서



2. 필드 수준 오류 코드 생성

- ① code + "." + objectName + "." + field
- ② code + "." + field
- ③ code + "." + fieldType
- ④ code



예) 코드가 "required", 객체명이 "user", 필드가 "age"인 경우

- ① required.user.age → 객체명과 필드명이 포함된 가장 구체적인 메시지
- ② required.age → 특정 필드에 대한 메시지
- ③ required.int → 필드 타입에 대한 메시지
- ④ required → 일반적인 오류 메시지



✓ MessageCodesResolver 메시지 생성

```
MessageCodesResolver resolver = new DefaultMessageCodesResolver();  
String[] objectErrorCodes = resolver.resolveMessageCodes("required", "user"); // 객체 수준 오류 코드 생성  
String[] fieldErrorCodes = resolver.resolveMessageCodes("required", "user", "age", Integer.class); // 필드 수준 오류 코드 생성  
  
System.out.println("객체 수준 코드:");  
Arrays.stream(objectErrorCodes).forEach(System.out::println);  
  
System.out.println("필드 수준 코드:");  
Arrays.stream(fieldErrorCodes).forEach(System.out::println);
```

객체 수준 코드:
required.user
required

필드 수준 코드:
required.user.age
required.age
required.int
required

✓ 오류 메시지 적용 원리

```
@PostMapping("/users")  
public String submitOrder(@ModelAttribute("order") User user, BindingResult bindingResult) {  
    if (user.getAge() <= 0) {  
        bindingResult.rejectValue("age", "required", null, "나이는 필수입니다"); // 오류 발생  
    }  
}
```

MessageCodesResolver 가 오류 메시지 코드를 생성

required.user.age
required.age
required.int
required

```
<form th:action="@{/order}" th:object="${order}" method="post">  
    <label>나이: <input type="text" th:field="*{age}" /></label>  
    <span th:if="#{#fields.hasErrors('age')}" th:errors="*{age}">나이 오류</span><br/>  
    <button type="submit">제출</button>  
</form>
```

required.user.age=사용자 나이는 필수입니다
required.age=나이는 필수입니다
required.int=숫자만 가능합니다
required=필수입니다

validation.properties

- ① MessageSource 는 생성된 오류 메시지 코드의 가장 구체적인 것부터 시작하여 순차적으로 돌면서 validation.properties 에 일치하는 코드 값이 존재하는지 검사한다
- ② 일치하는 값이 존재하면 해당 메시지를 타임리프 변수에 적용하고 만약 존재하지 않으면 기본 오류 메시지를 적용한다

✓ 오류 메시지 설정 및 우선순위 - validation.properties

객체 오류

required.user=사용자 정보를 입력해야 합니다.

필드 + 객체 오류

required.user.userName=사용자의 이름(userName)을 입력해야 합니다.

max.user.age=사용자의 나이(age)는 {0} 이하이어야 합니다.

min.user.age=사용자의 나이(age)는 {0} 이상이어야 합니다.

필드 오류

required.userName=이름(userName)은 필수 입력 사항입니다.

max.age=나이는 {0} 이하이어야 합니다.

min.age=나이는 {0} 이상이어야 합니다.

공통 타입별 메시지

required.java.lang.String=필수 문자입니다.

min.java.lang.String={0} 이상의 문자를 입력해주세요.

required.java.lang.Integer=필수 숫자입니다.

min.java.lang.Integer={0} 이상의 숫자를 입력해주세요.

max.java.lang.Integer={0} 이하의 숫자를 입력해주세요.

공통 메시지 (우선순위 가장 낮음)

required=값을 입력해야 합니다.

max=값은 {0} 이하이어야 합니다.

min=값은 {0} 이상이어야 합니다.

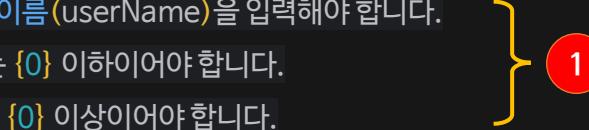
1. 오류 메시지는 객체 수준이든 필드 수준이든 구체적인 것부터 찾아서 적용한다

2. 메시지 처리 흐름: 바인딩 객체는 User

- reject("required") 는 다음과 같이 오류코드가 생성된다
 - required.user
 - required
- rejectValue("required", "userName") 는 다음과 같이 오류코드가 생성된다
 - required.user.userName
 - required.userName
 - required.java.lang.String
 - required

3. 최종 사용자 출력:

- 왼쪽 설정처럼 오류 메시지가 구성되어 있다면 번호 순서대로 출력된다
- 출력 순서를 조정하고 싶으면 구체적인 규칙의 오류 코드를 지워 나가면서 확인 해 본다



✓ DefaultBindingErrorProcessor

- 앞서 언급했듯이 스프링의 데이터 바인딩 오류 시에는 BindingResult 객체의 errors 속성에 오류 객체(FieldError)가 저장 된다고 했다
- 주로 스프링의 바인딩 오류는 데이터 간 타입이 서로 맞지 않아 TypeMismatchException 예외가 발생하는데 이 예외로부터 기본적인 오류 메시지 코드가 생성이 된다

```
codes = {String[4]@8576} ["typeMismatch"
  > 0 = "typeMismatch.user.age"
  > 1 = "typeMismatch.age"
  > 2 = "typeMismatch.java.lang.Integer"
  > 3 = "typeMismatch"
```

'typeMismatch' 코드로 필드(age) 수준의 오류 메시지 코드가 생성

- DefaultBindingErrorProcessor는 오류 코드를 포함하여 BindingResult로 부터 여러가지 오류 정보를 가져와서 실제로 FieldError 객체를 생성하는 클래스이다
- 만약 validation.properties에 위의 네 가지 오류 메시지 코드 중 아무것도 설정이 되어 있지 않으면 다음과 같은 기본 오류 메시지가 출력된다

Failed to convert property value of type 'java.lang.String' to required type 'java.lang.Integer' for property 'age'; For input string: "w250"

- validation.properties에 다음과 같이 오류 메시지 코드를 추가하면 원하는 메시지를 보여 줄 수 있다

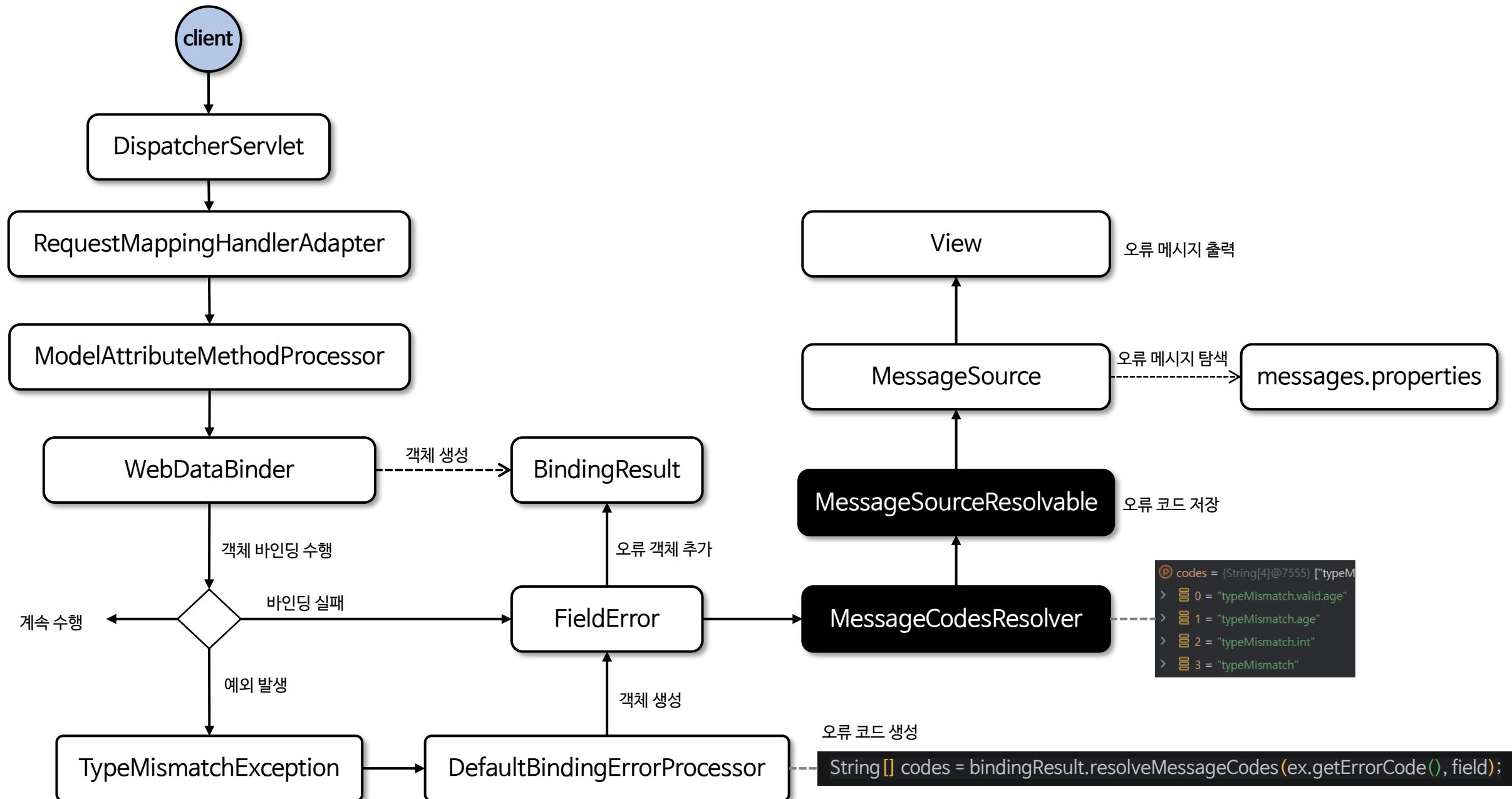
```
'typeMismatch.java.lang.Integer=숫자만 가능합니다.
'typeMismatch=형식이 맞지 않습니다.'
```

✓ MessageSourceResolvable

① ↗	MessageSourceResolvable
② ↗	getArguments() Object[]?
③ ↗	getCodes() String[]?
④ ↗	getDefaultMessage() String?

- MessageSource가 메시지를 찾을 때 오류 코드를 제공하는 클래스로서 순차적으로 메시지를 탐색하고 적절한 메시지를 찾아 반환한다
- 기본 구현체로 DefaultMessageSourceResolvable 클래스가 제공되며 ObjectError의 부모 클래스이다.
- 즉 ObjectError로부터 오류코드, 메시지 인자, 기본메시지를 전달 받는다

✓ 오류 메시지 흐름도



Validator

<https://github.com/onjsdnjs/spring-mvc-master/tree/Validator>

✓ 개요

- Spring 의 Validator 는 객체의 유효성을 검증하기 위해 설계된 인터페이스로 데이터 유효성 검증을 단순화하면서 비즈니스 로직과 검증 로직을 분리하는 데 사용된다.

✓ 구조

Validator

```
boolean supports(Class<?> clazz)  
void validate(Object target, Errors errors)
```

• supports()

- 해당 Validator가 특정 객체 타입을 지원하는지 확인하는 메서드이다
- 이 메서드는 일반적으로 이렇게 구현된다
 - Foo.class.isAssignableFrom(clazz) – Foo는 실제로 검증할 객체 인스턴스의 클래스 또는 상위 클래스이다

• validate()

- 주어진 대상 객체가 supports(Class) 메서드에서 true를 반환하는 클래스여야만 검증할 수 있다
- 실제 유효성 검사를 수행하는 메서드이며 유효성 검사에 실패한 경우 Errors 객체를 사용하여 오류를 추가한다

✓ 기본 구현

```
@Controller  
public class UserController {  
  
    @Autowired  
    private UserValidator userValidator;   
  
    @PostMapping("/users")  
    public String createUser(@ModelAttribute User user, BindingResult result) {  
  
        userValidator.validate(user, result);  
  
        if (result.hasErrors()) {  
            result.getAllErrors().forEach(error -> log.info("errors={}", error.getDefaultMessage()));  
            return "userForm";  
        }  
        return "redirect:/success";  
    }  
}
```

```
@Component  
public class UserValidator implements Validator {  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return User.class.isAssignableFrom(clazz);  
    }  
  
    @Override  
    public void validate(Object target, Errors errors) {  
        User user = (User) target;  
        // 공백 또는 빈 값 검증 유틸  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "userName", "required");  
        if (user.getAge() > 100) {  
            errors.rejectValue("age", "max", new Object[] {100});  
        }  
        // 글로벌 오류  
        if ("admin".equals(user.getUserName()) && user.getAge() < 20) {  
            errors.reject("authority");  
        }  
    }  
}
```

- 검증 로직과 비즈니스 로직이 완벽하게 분리 되었고 컨트롤러에서 오류 검증하는 부분도 훨씬 깔끔해졌다
- Validator는 이후 학습할 빈 검증에서 살펴볼 건데 매우 핵심적인 역할을 하게 된다

✓ 다중 검증기 구현

```
@Component
public class UserValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return User.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        User user = (User) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "userName", "required");

        if (user.getAge() > 100) {
            errors.rejectValue("age", "max", new Object[]{100}, "Age cannot exceed 100.");
        }
    }
}

@Component
public class AdminValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return Admin.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
        Admin admin = (Admin) target;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "adminName", "required");

        if (admin.getLevel() < 1) {
            errors.rejectValue("level", "min", new Object[]{1}, "Admin level must be at least 1.");
        }
    }
}
```

```
@Controller
@Controller
public class UserController {

    @Autowired
    private List<Validator> validators;

    @PostMapping("/users")
    public String createUser(@ModelAttribute Admin admin, BindingResult result) {
        for (Validator validator : validators) {
            if (validator.supports(result.getTarget().getClass())) {
                validator.validate(result.getTarget(), result);
                break;
            }
        }

        if (result.hasErrors()) {
            result.getAllErrors().forEach(error ->
                log.info("errors={}", error.getDefaultMessage())
            );
            return "form";
        }

        return "success";
    }
}
```

여러 개의 검증기를 활용할 경우에는 Validator의 supports() 메서드를 사용하여 현재 검증하고자 하는 객체의 타입을 체크하여 true를 반환하는 검증기를 선택해서 검증하는 식으로 진행하면 된다

✓ @InitBinder 와 Validator

```
@Controller  
public class UserController {  
  
    @Autowired  
    private UserValidator userValidator;  
  
    @InitBinder("user")  
    protected void initBinder(WebDataBinder binder) {  
        binder.setValidator(userValidator);  
    }  
    @PostMapping("/users")  
    public String registerUser(@Validated @ModelAttribute("user") User user, BindingResult result) {  
        if (result.hasErrors()) {  
            return "userForm"; // 유효성 검사 실패 시 반환할 뷰  
        }  
        return "redirect:/success";  
    }  
}
```

```
@Component  
public class UserValidator implements Validator {  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return User.class.isAssignableFrom(clazz);  
    }  
    @Override  
    public void validate(Object target, Errors errors) {  
        User user = (User) target;  
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "userName", "required");  
        if (user.getAge() > 100) {  
            errors.rejectValue("age", "max", new Object[] {100});  
        }  
        if ("admin".equals(user.getUserName()) && user.getAge() < 20) {  
            errors.reject("authority");  
        }  
    }  
}
```

- @InitBinder 의 WebDatabinder 객체는 호출될 때마다 새롭게 생성되는 객체로서 해당 컨트롤러에서만 binder 의 속성이 유지된다
- @ModelAttribute 앞에 @Validated 어노테이션이 선언되어 있으면 WebDatabinder 가 가지고 있는 검증기들을 실행시며 검증하게 된다
- @InitBinder("user") 에서 "user" 를 지정하게 되면 @ModelAttribute("user") 에서 지정한 "user" 와 동일한 경우에만 호출된다

✓ 다중 @InitBinder 와 Validator

```
@Controller
public class UserController2 {
    @Autowired
    private UserValidator userValidator;
    @Autowired
    private AdminValidator adminValidator;

    @InitBinder("user")
    protected void initBinder1(WebDataBinder binder) {
        binder.addValidators(userValidator);
    }
    @InitBinder("admin")
    protected void initBinder2(WebDataBinder binder) {
        binder.addValidators(adminValidator);
    }
    @PostMapping("/users")
    public String registerUser(@Validated @ModelAttribute("user") User user, BindingResult result) {
        if (result.hasErrors()) {
            return "userForm";
        }
        return "redirect:/success";
    }
    @PostMapping("/admin")
    public String registerAdmin(@Validated @ModelAttribute("admin") Admin admin, BindingResult result) {
        if (result.hasErrors()) {
            return "userForm";
        }
        return "redirect:/success";
    }
}
```

- 여러 개의 @InitBinder 와 Validator 를 사용 할 경우엔 setValidator 가 아닌 addValidators 를 사용해서 Validator 를 추가해야 한다
- @InitBinder("user"), @InitBinder("admin") 과 같이 @ModelAttribute에서 지정한 모델명을 기준으로 서로 구분해서 지정하고 Validator 도 구분해서 설정한다
- /users 를 호출하면 @InitBinder("user") 와 userValidator 가 호출되고 /admin 을 호출하면 @InitBinder("admin") 와 adminValidator 가 호출된다

Bean Validation

개요

<https://github.com/onjsdnjs/spring-mvc-master/tree/BeanValidation-개요>

✓ 개요

- Bean Validation은 Java 애플리케이션에서 객체의 유효성 검증을 위해 도입된 기술로서 데이터 검증 로직을 일관되게 적용할 수 있도록 하는 표준화 된 API를 제공한다
- 다양한 Java 프레임워크(Spring, Jakarta EE 등)에서 공통으로 사용할 수 있도록 JSR-303(Bean Validation 1.0)과 JSR-380(Bean Validation 2.0) 표준이 제정되었다

✓ Bean Validation & Spring 폼 검증

• Spring 폼 검증

- 개발자가 직접 검증 로직을 작성해야 하며, 각각의 필드에 대한 조건을 수동으로 명시해야 한다
- 검증 중 발생한 오류는 Errors 객체에 기록되며 사용자는 어떤 필드에서 어떤 오류가 발생 했는지 확인할 수 있다
- 검증 조건이 복잡하거나 커스텀 검증이 필요할 때 유용하다

```
if (user.getUsername() == null) {  
    errors.rejectValue("username", "required");  
}  
  
if (user.getAge() <= 0 || user.getAge() > 120) {  
    errors.rejectValue("age", "range");  
}
```

• Bean Validation

- 어노테이션을 사용해 검증 규칙을 선언적으로 정의할 수 있으며 개발자는 별도로 검증 로직을 작성할 필요가 없다
- 객체의 필드에 선언된 어노테이션을 기반으로 자동으로 검증이 수행된다
- 객체가 중첩된 경우에도 중첩된 객체에 대한 검증을 함께 수행할 수 있다



```
public class User {  
  
    @NotNull(message = "Username is required")  
    private String username;  
  
    @Range(min = 1, max = 120, "Age must be between 1 and 120")  
    private Integer age;  
}
```

✓ JSR-303 / JSR-380 주요 어노테이션

- JSR-303과 JSR-380은 둘 다 Java Bean Validation의 표준 사양이지만 JSR-380이 JSR-303을 확장하여 추가된 기능을 제공하는 방식으로 발전했다

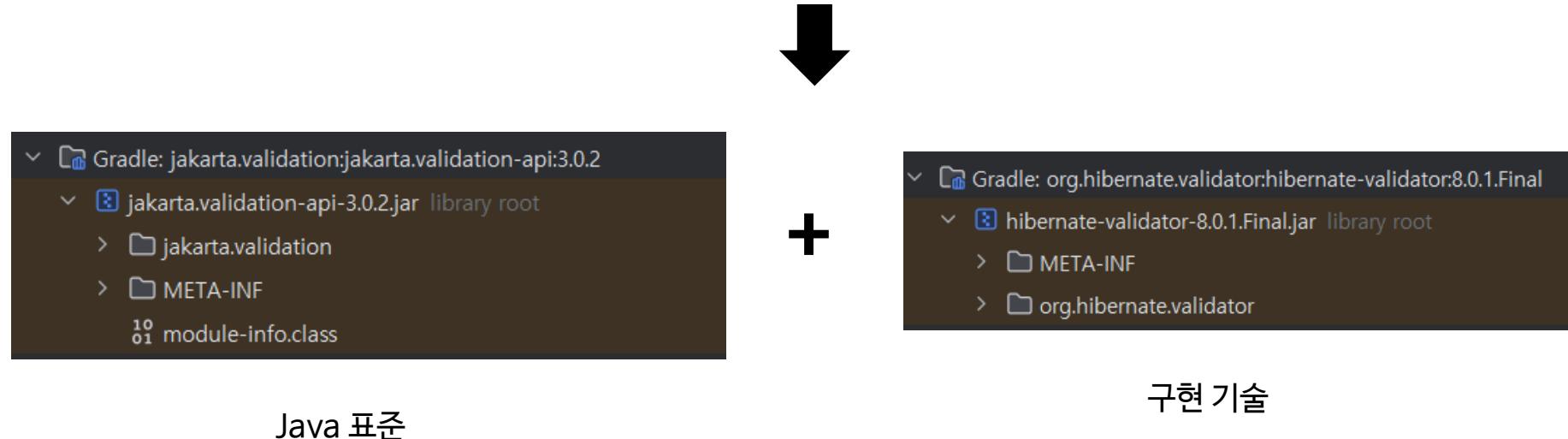
JSR-303 어노테이션	설명	JSR-380 어노테이션	설명
@NotNull	필드가 null 이 아니어야 함을 검증.	@Email	필드가 유효한 이메일 형식인지 검증.
@Size(min, max)	문자열, 배열, 컬렉션 등의 크기가 지정된 범위 내에 있어야 함을 검증	@NotEmpty	필드 값의 길이가 0이면 오류, 공백도 글자로 인식
@Min(value)	값이 지정된 최소값 이상이어야 함을 검증.	@NotBlank	필드 값의 길이가 0이면 오류, 공백은 글자로 인식 안함
@Max(value)	값이 지정된 최대값 이하여야 함을 검증.	@Positive	필드 값이 양수인지 검증.
@Pattern(regex)	문자열이 지정된 정규 표현식을 만족하는지 검증.	@PositiveOrZero	필드 값이 양수 또는 0인지 검증.
@NotEmpty	문자열, 배열, 컬렉션이 빈 값이 아니어야 함을 검증.	@Negative	필드 값이 음수인지 검증.
@NotBlank	공백을 제외한 문자열이 비어있지 않음을 검증.	@NegativeOrZero	필드 값이 음수 또는 0인지 검증.
@DecimalMin(value)	소수를 포함한 값이 지정된 최소값 이상이어야 함을 검증.	@PastOrPresent	날짜 값이 과거 또는 현재여야 함을 검증.
@DecimalMax(value)	소수를 포함한 값이 지정된 최대값 이하여야 함을 검증.	@FutureOrPresent	날짜 값이 미래 또는 현재여야 함을 검증.
@Digits(integer, fraction)	숫자의 정수부와 소수부의 자릿수를 검증.		
@AssertTrue	필드 값이 true여야 함을 검증.		
@AssertFalse	필드 값이 false여야 함을 검증.		
@Past	날짜 값이 과거 날짜여야 함을 검증.		
@Future	날짜 값이 미래 날짜여야 함을 검증.		
@Valid	중첩된 객체나 필드를 재귀적으로 검증.		

<https://hibernate.org/validator/documentation/getting-started/>

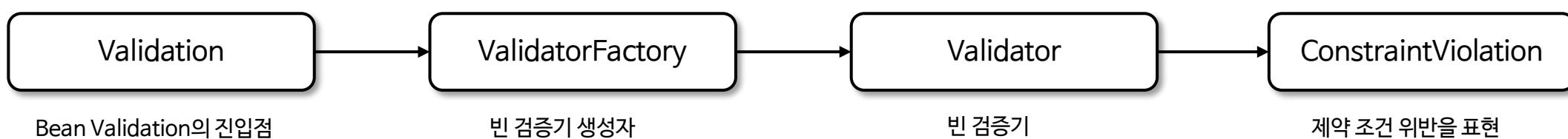
✓ Bean Validation 사용

- 어노테이션 검증을 수행하기 위해서는 스프링 부트에서 제공하는 의존성을 추가해야한다
- Bean Validation 표준을 구현한 대표적인 기술들이 있으며 Hibernate Validator, Apache Bval 등이 있으며 주로 Hibernate Validator 를 사용한다

implementation 'org.springframework.boot:spring-boot-starter-validation'



✓ 주요 클래스



✓ 기본 구현

```
public class User {  
  
    @NotNull(message = "사용자명은 필수입니다.")  
    @Size(min = 3, max = 15, message = "사용자명은 3~15 길이입니다. ")  
    private String username;  
  
    @Email(message = "이메일 형식에 맞지 않습니다. ")  
    private String email;  
  
    @Min(value = 0, message = "나이는 0 보다 커야 합니다.")  
    @Max(value = 120, message = "나이는 120 보다 같거나 작아야 합니다.")  
    private Integer age;  
  
    // Getters and Setters  
}
```

1. 객체의 필드에 어노테이션을 선언하여 유효성 검사를 수행한다
2. 유효성 검사는 내부적으로 ValidatorFactory 와 Validator 객체를 통해 수행된다
3. 검증 결과는 ConstraintViolation 객체에 저장되며 이를 통해 검증 실패 시 발생한 오류 메시지와 오류 필드를 확인 할 수 있다

```
@Test  
void testInvalidUser() throws ParseException {  
  
    ValidatorFactory factory = Validation.buildDefaultValidatorFactory();  
    Validator validator = factory.getValidator();  
  
    User user = new User();  
    user.setUsername("ab"); // 길이가 3보다 짧음  
    user.setEmail("invalid-email"); // 이메일 형식이 아님  
    user.setAge(-1); // 나이가 0보다 작음  
    Set<ConstraintViolation<User>> violations = validator.validate(user);  
    Assertions.assertThat(3).isEqualTo(violations.size());  
  
    for (ConstraintViolation<User> violation : violations) {  
        String propertyPath = violation.getPropertyPath().toString();  
        String message = violation.getMessage();  
  
        if (propertyPath.equals("username")) {  
            Assertions.assertThat("사용자명은 3~15 길이입니다.").isEqualTo(message);  
        } else if (propertyPath.equals("email")) {  
            Assertions.assertThat("이메일 형식에 맞지 않습니다.").isEqualTo(message);  
        } else if (propertyPath.equals("age")) {  
            Assertions.assertThat("나이는 0 보다 커야 합니다.").isEqualTo(message);  
        }  
    }  
}
```

Java Bean Validation + Spring 통합

<https://github.com/onjsdnjs/spring-mvc-master/tree/BeanValidation-Spring-%ED%8A%B8>

✓ 개요

- Spring 에서는 org.springframework.validation.Validator 를 통해 Spring 의 자체 검증 메커니즘과 Java Bean Validation 을 통합하고 이를 통해 Bean Validation 표준 검증을 Spring 에서 바로 활용할 수 있다
- Spring MVC 는 컨트롤러 계층에서 @Valid 또는 @Validated 를 사용하여 자동으로 Bean Validation 을 수행하며 검증 실패 시 발생하는 오류를 BindingResult 를 통해 보관하고 프로그램적으로 처리할 수 있다
- Bean Validation 표준의 그룹(Group)화를 지원하여 특정 조건에서만 검증을 수행할 수 있으며 Bean Validation을 확장하여 커스텀 Validator 를 생성할 수 있다
- Spring 에서는 MessageSource 를 통해 여러 메시지를 커스터 마이징할 수 있다

✓ 기본 사용법

- 스프링에서는 어노테이션 기반 검증을 위해 @Valid 와 @Validated 어노테이션을 사용할 수 있으며 사용 방식에 있어 약간 차이가 있다
- @Valid 는 jakarta.validation 에 포함되어 있고 @Validated 은 org.springframework.validation.annotation 에 포함되어 있으며 @Valid 를 사용하기 위해서는 org.springframework.boot:spring-boot-starter-validation 의존성이 필요하다
- 두 어노테이션 모두 객체 탑재에만 사용 할 수 있으며 검증할 객체 바로 앞에 위치해야 하고 검증된 결과는 BindingResult 에 담긴다

```
public String registerUser(@Validated @ModelAttribute("user") User user, BindingResult bindingResult)
```



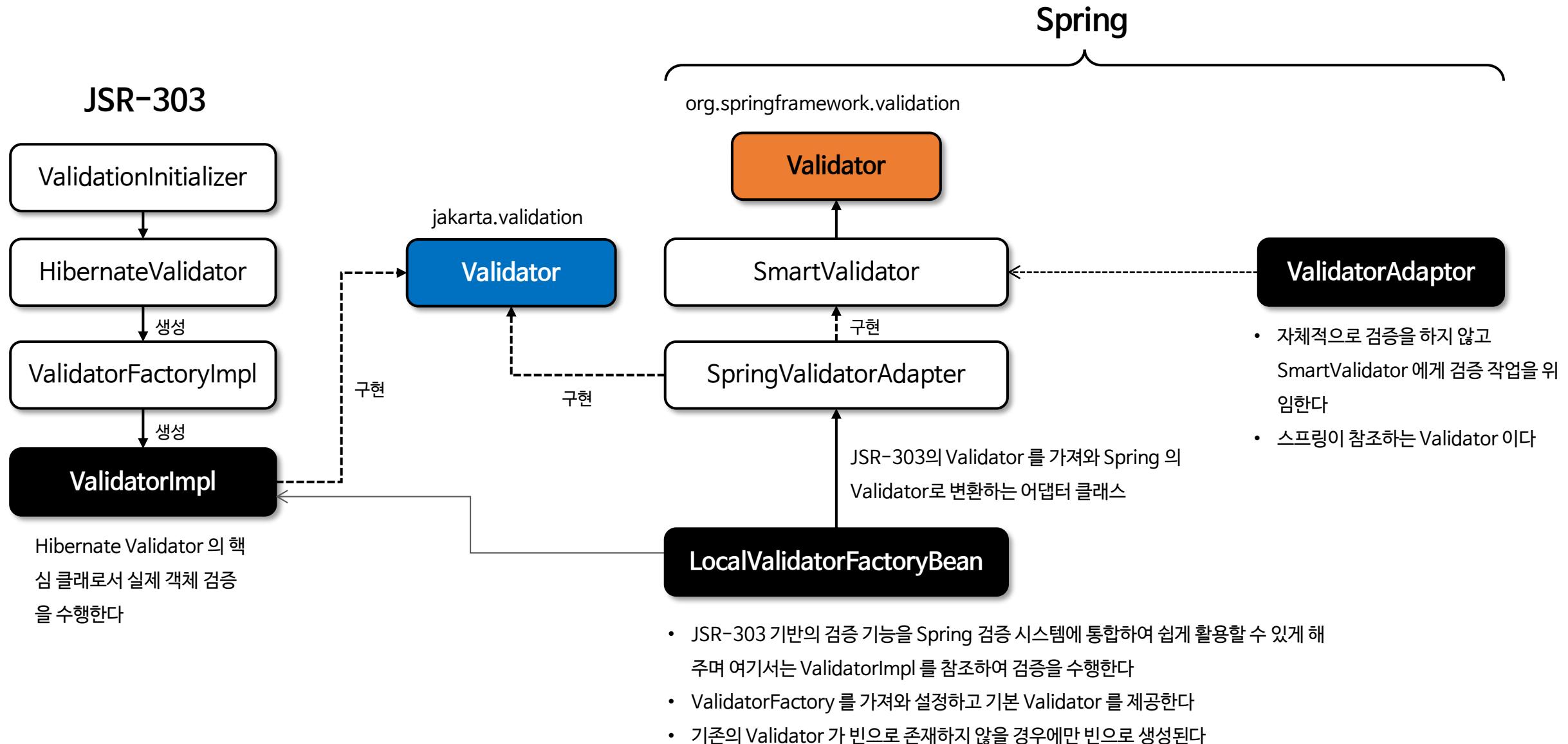
- 검증은 바인딩의 맨 마지막 처리 과정이며 기본적으로 바인딩에 성공한 필드는 검증이 이루어진다
- 만약 필드의 탑재변환이 실패하면 실패 결과가 FieldError 객체에 담기고 BindingResult 에 보관된다
- 타입변환에 실패한 필드는 기본 값이 저장된 상태에서 검증이 이루어지지만 Validator 구현체에 따라 예외가 발생 할 수도 있고 기본 검증이 이루어질 수 있다

```
public class User {  
    @NotEmpty(message = "사용자명은 필수입니다")  
    private String username;
```

```
    @Email(message = "이메일형식이 맞지 않습니다")  
    private String email;
```

```
    // Getter and Setter ..  
}
```

✓ Java Bean Validation 과 Spring 통합 구조



✓ @Valid

- JSR-380 표준에 속하는 기본적인 검증 어노테이션으로서 객체의 필드뿐만 아니라 중첩된 객체(nested object)에서도 유효성 검사를 수행할 수 있다
- 주로 단순한 유효성 검사를 수행하는 데 사용되며 jakarta.validation 패키지에 속해 있다

```
@PostMapping("/submitUser")
public String submitUser(@Valid @ModelAttribute User user, BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "userForm"; // 폼 페이지로 다시 이동
    }

    model.addAttribute("user", user); // 모델에 User 객체 추가
    return "userSuccess"; // 성공 페이지로 이동
}
```

```
POST /users
Content-Type: application/x-www-form-urlencoded

username=hong&email=a@a.com
```

```
public class User {
    @NotEmpty(message = "사용자명은 필수입니다")
    private String username;

    @Email(message = "이메일 형식이 맞지 않습니다")
    private String email;

    public String getUsername() {
        return username;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

✓ @RequestBody + @Valid

- @RequestBody 와 @Valid 클라이언트의 요청 데이터를 Java 객체로 매팅하고 이를 검증하는 데 사용된다

```
@PostMapping("/submitUser")
public String submitUser(@Valid @RequestBody User user, BindingResult result, Model model) {
    if (result.hasErrors()) {
        return "userForm"; // 폼 페이지로 다시 이동
    }

    model.addAttribute("user", user); // 모델에 User 객체 추가
    return "userSuccess"; // 성공 페이지로 이동
}
```

```
POST http://localhost:8080/registers
Content-Type:application/json
{
    "username": "leaven",
    "email": "10"
}
```

```
public class User {
    @NotEmpty(message = "사용자명은 필수입니다")
    private String username;

    @Email(message = "이메일 형식이 맞지 않습니다")
    private String email;

    public String getUsername() {
        return username;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

- @RequestBody 는 객체 바인딩이 아닌 HttpMessageConverter 가 문자열 데이터를 객체로 변환하는 방식으로 이루어진다
- 그래서 필드 단위로 오류를 저장하는 것이 아니기 때문에 필드 제약 오류가 아닌 타입 오류가 나면 예외(HttpMessageNotReadableException)가 발생해서 빈 검증이 더 이상 진행되지 않고 중단된다

✓ 오류 검증 할 Validator 가 없는 경우

- 아래 예제와 같이 제약 조건에 맞는 Validator 자체가 존재하지 않을 경우 예외가 발생하여 정상적인 동작이 이루어지지 않는다

```
@PostMapping("/submitUser")
public String submitUser(@Valid @ModelAttribute Users user, BindingResult result, Model model) {

    if (result.hasErrors()) {
        return "userForm"; // 폼 페이지로 다시 이동
    }

    model.addAttribute("user", user); // 모델에 User 객체 추가
    return "userSuccess"; // 성공 페이지로 이동
}
```

```
@Data
public class Users {

    @NotNull(message = "Password is required.")
    @Size(min = 6, message = "At least 6 characters.")
    private int password;

    @NotNull(message = "Username is required.")
    private String username;

    @NotNull(message = "Email is required.")
    private String email;
}
```

```
POST /users
Content-Type: application/x-www-form-urlencoded

username=leaven&email=25&password=dd
```

타입변환에 실패하여 password 필드에는 값이 저장되지 않음

```
ex = {UnexpectedTypeException@7373} "jakarta.validation.UnexpectedTypeException: HV000030: No validator could be found for constraint 'jakarta.validation.constraints.Size' validating type 'java.lang.Integer'. Check configuration for 'password'"
> ⚡ backtrace = {Object[7]@9086} ... View
> ⚡ detailMessage = "HV000030: No validator could be found for constraint 'jakarta.validation.constraints.Size' validating type 'java.lang.Integer'. Check configuration for 'password'"
> ⚡ cause = {UnexpectedTypeException@7373} "jakarta.validation.UnexpectedTypeException: HV000030: No validator could be found for constraint 'jakarta.validation.constraints.Size' validating type 'java.lang.Integer'. Check configuration for 'password'"
```

- *@Size 제약조건을 검증하는 Validator 중에 Integer 타입을 검증하는 객체를 찾을 수 없다고 함*
- *이런 경우가 흔치는 않겠지만 정상적으로 진행이 안되는 이유를 알고 대처하는 것은 필요함*

✓ @Validated

- Spring 프레임워크에서 제공하는 어노테이션으로서 그룹 기반 검증(validation group)을 지원하며 복잡한 검증 요구사항이 있는 경우 유용하다

```
public interface Vgroups {  
    interface CreateGroup {}  
    interface UpdateGroup {}  
}
```

```
public class User {  
  
    @NotEmpty(message = "Username is required",  
             groups = VGroups.CreateGroup.class)  
    private String username;  
  
    @Email(message = "Email should be valid",  
          groups = {VGroups.CreateGroup.class,  
                    VGroups.UpdateGroup.class})  
    private String email;  
  
    이하 생략..  
}
```

```
@PostMapping("/createUser")  
public String createUser(@Validated(VGroups.CreateGroup.class) @ModelAttribute User user,  
                        BindingResult result, Model model) {
```

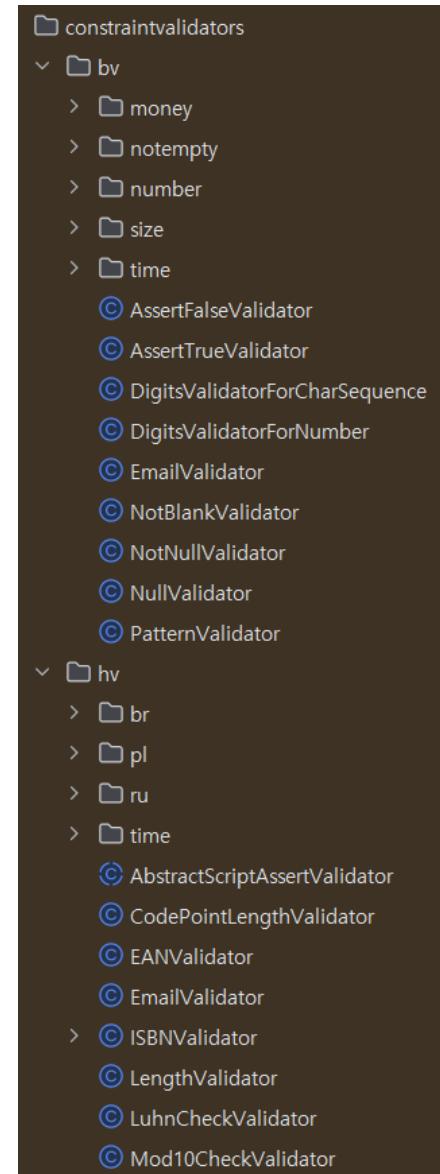
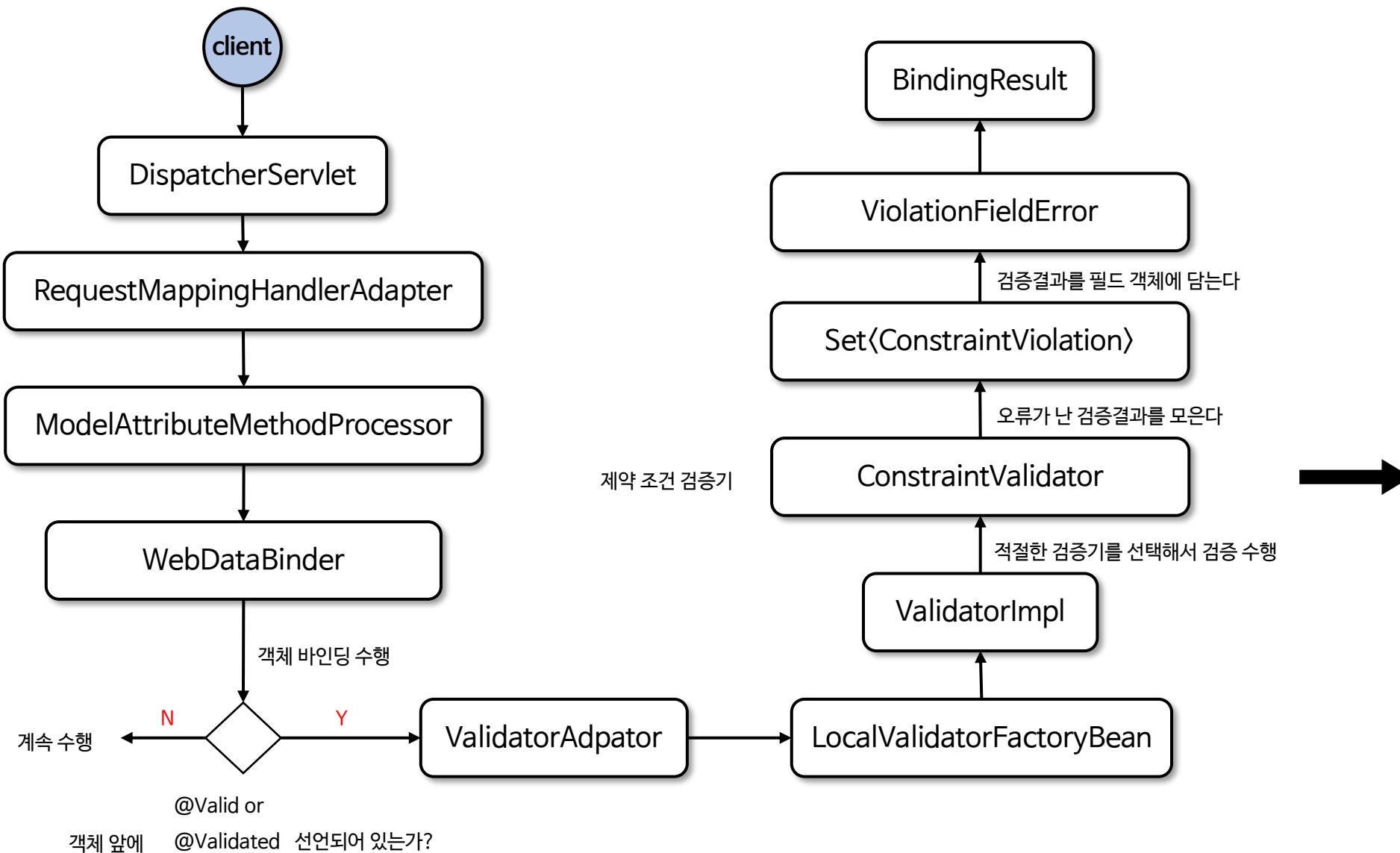
```
    if (result.hasErrors()) return "userForm";  
  
    model.addAttribute("user", user);  
  
    return "userSuccess"; // 성공 페이지로 이동  
}
```

```
@PostMapping("/updateUser")  
public String updateUser(@Validated(VGroups.UpdateGroup.class) @ModelAttribute User user,  
                        BindingResult result, Model model) {
```

```
    if (result.hasErrors()) return "userUpdateForm";  
  
    model.addAttribute("user", user);  
  
    return "userUpdateSuccess"; // 성공 페이지로 이동  
}
```

- username 의 @NotEmpty 검증은 createUser() 인 회원가입에만 적용되며 email 의 @Email 검증은 createUser() 와 updateUser() 인 회원가입과 회원수정 모두 적용된다
- 예를 들어 등록하기와 수정하기를 하나의 객체를 가지고 분기해서 동시에 사용할 수는 있으나 비교적 필드나 로직이 간단한 경우는 유용하지만 필드가 많고 여러 상황에 따른 로직이 복잡할 경우에는 등록하는 객체와 수정하는 객체를 별도로 구분하여 검증에 사용하는 것이 더 좋다

✓ 흐름도

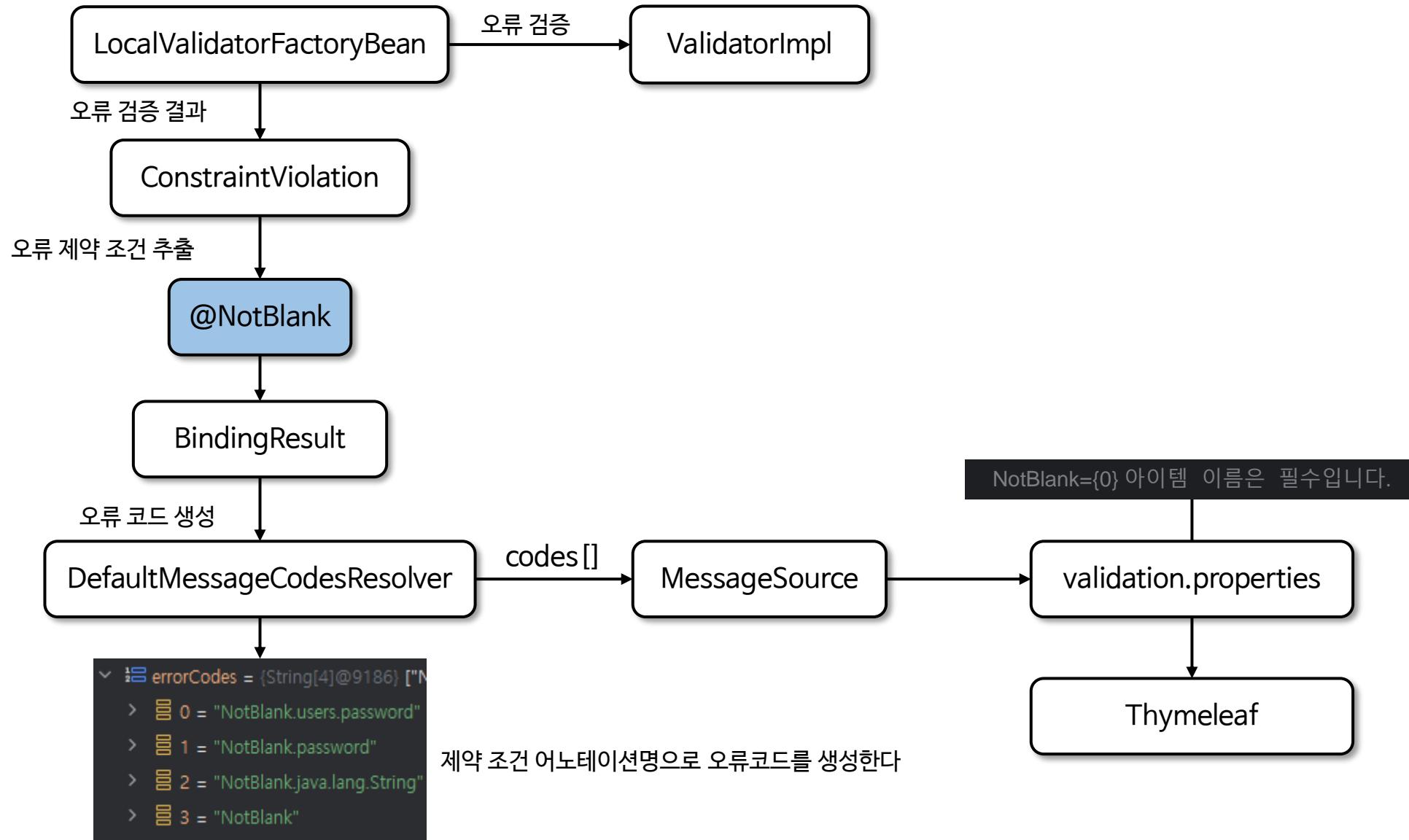


MessageSource 연동

<https://github.com/onjsdnjs/spring-mvc-master/tree/BeanValidation-MessageSource-연동>

✓ 개요

- Bean Validation에서 검증 메시지는 메시지를 하드 코딩하지 않고 MessageSource를 통해 다양한 언어로 메시지를 관리할 수 있다



✓ 기본 구현

POST /product

Content-Type: application/x-www-form-urlencoded

productName=computer&price=123

120 보다 초과해서 오류 발생!

```
@Data  
public class Product {  
  
    @NotNull  
    private String productName;  
  
    @Min(1)  
    @Max(120)  
    private int price;  
}
```

제약 조건을 가지고 스프링이 오류 코드 생성!

Max.price. productName
Max.pirce
Max.int
Max

MessageSource

NotNull={0} 값이 비어 있음
Min={0}, 최소 {1}
Max={0}, 최대 {1}

validation.properties

```
<form th:action="@{/product}" th:object="${product}" method="post">  
    <!-- 글로벌 오류 (ObjectError) 출력 -->  
    <div th:if="#{fields.hasGlobalErrors()}">  
        <p th:each="error : fields.globalErrors()" th:text="${error}">글로벌 오류 메시지</p>  
    </div>  
    <!-- 상품명 -->  
    <label>상품명: <input type="text" th:field="*{productName}" /></label>  
    <span th:if="#{fields.hasErrors('productName')}" th:errors="*{productName}">상품명  
    오류</span><br/>  
  
    <!-- 가격 -->  
    <label>가격: <input type="text" th:field="*{price}" /></label>  
    <span th:if="#{fields.hasErrors('price')}" th:errors="*{price}">가격 오류</span><br/>  
  
    <button type="submit">제출</button>  
</form>
```



price, 최대 120

{0} 은 필드명, {1}.. 은 어노테이션에 설정한 값

커스텀 검증 어노테이션

<https://github.com/onjsdnjs/spring-mvc-master/tree/BeanValidation-커스텀-검증-어노테이션>

✓ 개요

- 커스텀 검증 어노테이션은 기본적인 유효성 검증(예: @NotNull, @Email 등) 외에 특정한 유효성 검사에 적용하기 위해 직접 어노테이션을 정의하는 방법을 말한다

① ↗ ConstraintValidator <A, T>
② ↗ isValid (T, ConstraintValidatorContext) boolean
③ ↗ initialize (A) void

- 주어진 제약 조건(예: @NotNull) A에 대해 검증 대상 타입 T를 검증하는 인터페이스이다
- 검증 작업을 수행하며 불린 값을 반환한다. 검증 대상 객체(T)는 변경하지 않고 그대로 유지해야 한다
- 검증기의 isValid 호출을 준비하기 위해 초기화 한다. 검증에 사용되기 전에 먼저 초기화 된다

✓ 구현 방법

- 유효성 검사를 위한 새로운 어노테이션을 정의한다
- 실제 검증 로직을 수행하기 위한 ConstraintValidator 인터페이스를 구현한다
- 어노테이션에 ConstraintValidator 클래스를 연결하고 유효성 검사 시 해당 어노테이션이 ConstraintValidator를 사용하도록 한다

✓ ValidPassword 어노테이션 정의

// 그룹화

```
@Documented
@Constraint(validatedBy = PasswordValidator.class) // 검증기(PasswordValidator)를 지정
@Target({ ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidPassword {
    String message() default "잘못된 비밀번호입니다. 비밀번호는 최소 8자 이상이어야 하며, 대문자, 소문자, 숫자를 포함해야 합니다."; // 검증 실패 시 출력할 메시지
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    int minLength() default 8; // 최소 길이 속성 추가
}
```

✓ PasswordValidator 클래스 구현

```
public class PasswordValidator implements ConstraintValidator<ValidPassword, String> {

    private int minLength;

    @Override
    public void initialize(ValidPassword constraintAnnotation) {
        this.minLength = constraintAnnotation.minLength(); // 어노테이션에서 최소 길이 설정
    }

    @Override
    public boolean isValid(String password, ConstraintValidatorContext context) {
        if (password == null) {
            return false;
        }

        // 비밀번호가 최소 길이와 대문자, 소문자, 숫자를 포함하는지 검사
        boolean hasUpperCase = password.chars().anyMatch(Character::isUpperCase);
        boolean hasLowerCase = password.chars().anyMatch(Character::isLowerCase);
        boolean hasDigit = password.chars().anyMatch(Character::isDigit);

        return password.length() >= minLength && hasUpperCase && hasLowerCase && hasDigit;
    }
}
```

✓ 유효성 검사 클래스에 어노테이션 적용

```
public class User {

    @NotNull(message = "Password must not be null")
    @ValidPassword(minLength = 10) // 최소 길이를 10자로 설정
    private String password;

    // Getters and Setters
}
```

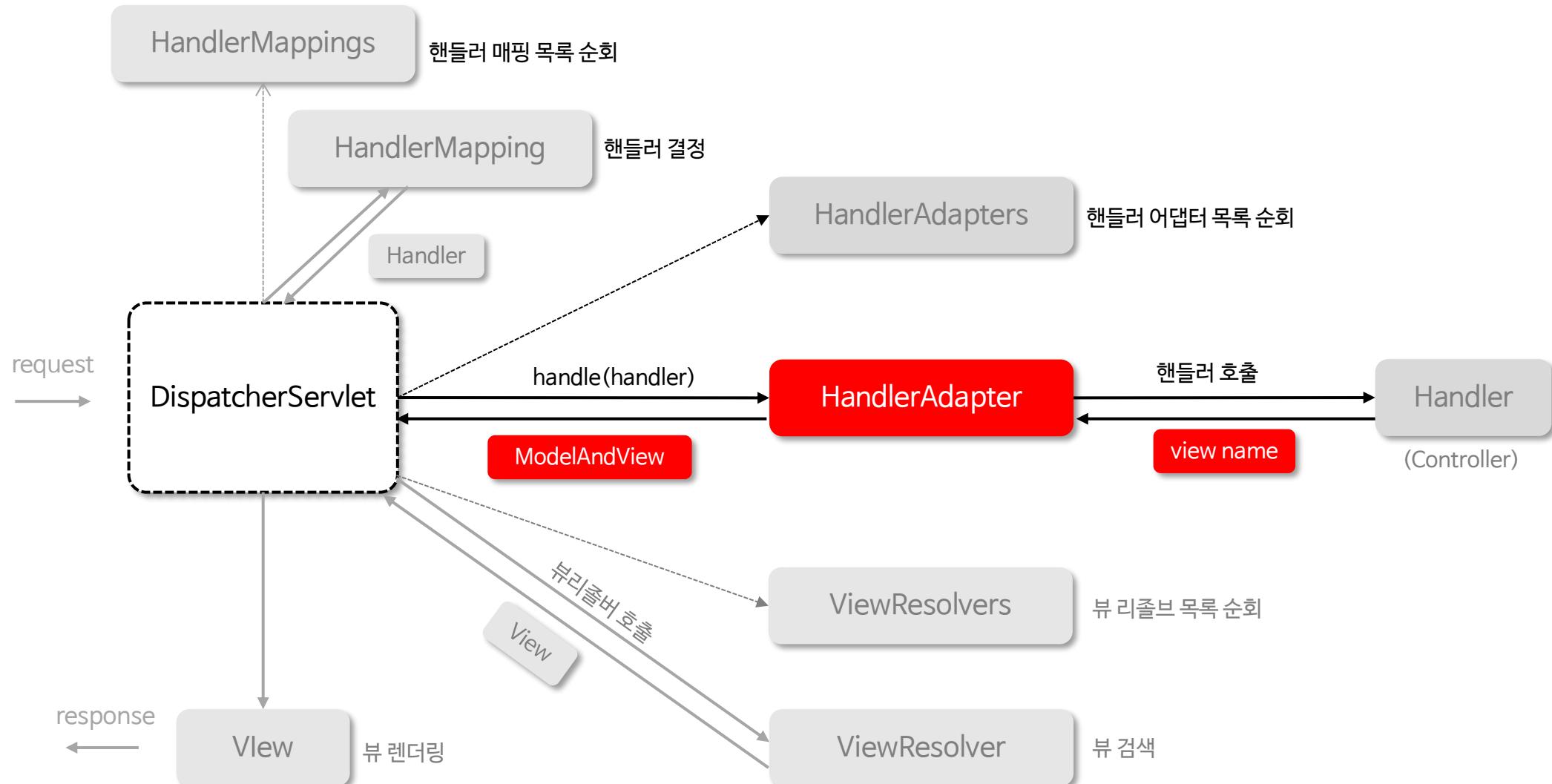
✓ 컨트롤러에서 유효성 검사 수행

```
@PostMapping("/register")
public String registerUser(@Valid @ModelAttribute("user") User user,
                           BindingResult bindingResult, Model model) {
    if (bindingResult.hasErrors()) {
        model.addAttribute("formErrors", bindingResult.getAllErrors());
        return "register";
    }
    return "registrationSuccess";
}
```

Return Values

<https://github.com/onjsdnjs/spring-mvc-master/tree/Return-Values>

✓ 스프링 MVC 아키텍처



✓ 개요

- 스프링 MVC에서 컨트롤러 메서드의 반환 방식은 크게 View와 HTTP 본문 응답으로 나눌 수 있으며 각 방식은 클라이언트에게 반환하는 응답의 형태를 결정한다

1) View 렌더링

- HTML과 같은 페이지를 클라이언트에게 반환하는 방식으로서 컨트롤러가 뷰 이름을 반환하고 뷰 레이어에서 해당 이름을 해석하여 적절한 HTML을 생성한다

2) HTTP 본문 응답

- JSON, XML 등 데이터 형식으로 응답을 직접 반환하는 방식으로서 REST API와 같은 데이터 중심의 애플리케이션에서 주로 사용된다
- 뷰가 아닌 데이터를 응답 본문에 담아 전달해서 클라이언트는 페이지가 아닌 데이터를 수신하게 된다

✓ HandlerMethodReturnValueHandler

HandlerMethodReturnValueHandler

```
boolean supportsReturnType (MethodParameter returnType); // 주어진 메서드의 반환 타입이 이 핸들러에 의해 지원되는지 여부를 확인한다
```

```
void handleReturnValue (Object returnValue, MethodParameter returnType, ModelAndView mavContainer, NativeWebRequest webRequest) throws Exception
```

- 컨트롤러로부터 응답결과를 받아 응답처리를 위한 작업을 담당하는 클래스이다
- 다양한 유형의 파라미터 (예: String, View, @ResponseBody 등)를 처리하기 위해 여러 HandlerMethodReturnValueHandler 기본 구현체를 제공한다
- 개발자가 필요에 따라 HandlerMethodReturnValueHandler 인터페이스를 직접 구현할 수 있다



✓ View 렌더링 반환 타입

1. String - 문자열로 뷰 이름을 반환하면 ViewResolver에 의해 해당 이름에 맞는 뷰가 렌더링 된다
2. ModelAndView - 뷰와 모델 데이터를 함께 담아 반환하는 객체입니다. 뷰 이름뿐만 아니라 모델 데이터를 함께 설정하여 전달할 수 있다
3. View - View 인터페이스를 구현한 객체를 직접 반환할 수 있다. 이 경우 ViewResolver에 의존하지 않고 특정 View 인스턴스를 직접 렌더링 한다
4. Model 또는 Map - 모델 데이터를 반환하면 뷰 이름은 요청 경로에 따라 자동으로 결정된다. Model이나 Map은 뷰에 전달할 데이터로만 사용된다

1. String

```
public String viewName(Model model) {  
    return "home"; // ViewResolver에 의해 home.jsp 또는 home.html 렌더링  
}
```

2. ModelAndView

```
public ModelAndView modelAndView() {  
    ModelAndView mav = new ModelAndView("home"); // 렌더링할 뷰 이름 설정  
    mav.addObject("message", "Hello World"); // 데이터 저장  
    return mav;  
}
```

3. View

```
public View view() {  
    return new ExcelView(); // 엑셀 데이터로 렌더링, View 객체를 직접 반환하여 ViewResolver를 사용하지 않음  
}
```

4. Model or Map

```
public Model model(Model model) {  
    model.addAttribute("message", "Hello World");  
    return model; // Model을 반환하여 데이터만 제공 (뷰 이름은 요청 경로를 기준으로 자동 결정)  
}
```

✓ ModelViewContainer

- Spring MVC가 내부적으로 요청 처리 시점에 자동으로 생성 및 관리하는 클래스로서 요청 처리가 완료될 때까지 모델과 뷰 관련 데이터를 임시로 보관한다

ModelAndViewContainer		
(f) redirectModel	ModelMap ?	리다이렉션 시에 사용될 모델 객체이다
(o) sessionStatus	SessionStatus	세션 상태를 관리하기 위한 객체로 세션의 완료 상태를 확인하거나 설정할 수 있다
(o) defaultModel	ModelMap	요청에서 사용될 기본 모델 객체로서 <i>BindingAwareModelMap</i> 구현체를 사용한다
(f) ignoreDefaultModelOnRedirect	boolean	@Deprecated 되었다
(f) redirectModelScenario	boolean	리다이렉션 상황을 나타내는 플래그로 리다렉션이 발생하면 true로 설정된다
(o) noBinding	Set<String>	데이터 바인딩이 필요하지 않은 모델 속성의 이름을 저장하는 집합이다
(f) view	Object?	컨트롤러가 반환할 뷰 이름 또는 뷰 객체를 저장하는 필드이다
(f) requestHandled	boolean	요청이 핸들러 내에서 완전히 처리되었는지 나타내는 플래그로서 true 이면 추가적인 뷰 해석이 필요 없음을 나타낸다
(f) status	HttpStatusCode?	HTTP 상태 코드를 나타내는 필드로 응답의 상태를 설정할 때 사용된다
(o) bindingDisabled	Set<String>	바인딩이 비활성화된 속성 이름을 저장하는 집합이다

- 리다이렉션 시에 사용될 모델 객체이다
- 세션 상태를 관리하기 위한 객체로 세션의 완료 상태를 확인하거나 설정할 수 있다
- 요청에서 사용될 기본 모델 객체로서 *BindingAwareModelMap* 구현체를 사용한다
- @Deprecated 되었다
- 리다이렉션 상황을 나타내는 플래그로 리다렉션이 발생하면 true로 설정된다
- 데이터 바인딩이 필요하지 않은 모델 속성의 이름을 저장하는 집합이다
- 컨트롤러가 반환할 뷰 이름 또는 뷰 객체를 저장하는 필드이다
- 요청이 핸들러 내에서 완전히 처리되었는지 나타내는 플래그로서 true 이면 추가적인 뷰 해석이 필요 없음을 나타낸다
- HTTP 상태 코드를 나타내는 필드로 응답의 상태를 설정할 때 사용된다
- 바인딩이 비활성화된 속성 이름을 저장하는 집합이다

✓ ModelAndView

- 최종적으로 뷰를 렌더링하기 위한 모델과 뷰의 정보를 제공하는 클래스이다
- ModelAndView 객체를 직접 반환해도 되고 뷰 이름이나 뷰 객체를 반환하게 되면 내부적으로 ModelAndView 객체가 생성되어 응답을 구성한다

ModelAndView		
(f) view	Object ?	컨트롤러 메서드가 반환할 뷰의 이름 또는 뷰 객체로서 String 타입의 뷰 이름이나 특정 뷔 객체(View 타입의 객체)를 가질 수 있다
(f) model	ModelMap ?	뷰에 전달될 데이터를 저장하는 모델 객체로서 ModelMap 을 통해 데이터를 추가할 수 있으며 추가된 데이터는 뷔에서 사용하게 된다
(f) cleared	boolean	ModelAndView 객체가 초기화 되었는지 나타내는 플래그로서 clear() 메서드를 호출하면 model과 view가 초기화되고 cleared 플래그가 true 가 된다
(f) status	HttpStatusCode?	응답의 HTTP 상태 코드를 설정하는 속성이며 HttpStatus.OK (200), HttpStatus.NOT_FOUND (404) 등의 HTTP 상태 코드를 설정할 수 있다

- 컨트롤러 메서드가 반환할 뷔의 이름 또는 뷔 객체로서 String 타입의 뷔 이름이나 특정 뷔 객체(View 타입의 객체)를 가질 수 있다
- 뷰에 전달될 데이터를 저장하는 모델 객체로서 ModelMap 을 통해 데이터를 추가할 수 있으며 추가된 데이터는 뷔에서 사용하게 된다
- ModelAndView 객체가 초기화 되었는지 나타내는 플래그로서 clear() 메서드를 호출하면 model과 view가 초기화되고 cleared 플래그가 true 가 된다
- 응답의 HTTP 상태 코드를 설정하는 속성이며 HttpStatus.OK (200), HttpStatus.NOT_FOUND (404) 등의 HTTP 상태 코드를 설정할 수 있다

✓ HTTP 본문 응답 반환 타입

1. `@ResponseBody` - 컨트롤러 메서드의 반환 값을 `HttpMessageConverter`를 통해 JSON, XML 등으로 변환하여 응답 본문에 직접 작성한다
2. `HttpEntity<T>`, `ResponseEntity<T>` - HTTP 응답(헤더와 본문 모두)을 구성할 수 있다. `ResponseEntity`는 상태 코드, 헤더, 본문을 모두 포함할 수 있어 더 정밀한 응답 구성이 가능하다
3. `Callable<V>`, `ListenableFuture<V>`, `CompletableFuture<V>` - 비동기 작업의 결과로 반환되는 타입을 사용할 수 있다

1. `@ResponseBody` 예제

```
@ResponseBody  
public User responseBody() {  
    return new User("leaven", "leaven@example.com"); // User 객체가 JSON 형식으로 응답 본문에 작성된다  
}
```

2. `ResponseEntity` 예제

```
public ResponseEntity<User> responseEntity() {  
    User user = new User("leaven", "leaven@example.com");  
    return new ResponseEntity<>(user, HttpStatus.OK); // User 객체가 JSON 형식으로 반환되고 상태 코드는 200 OK가 된다  
}
```

3. `Callable` 예제

```
public Callable<User> callable() {  
    return () -> {  
        Thread.sleep(2000);  
        return new User("leaven", "leaven@example.com"); // 비동기 작업이 완료되면 User 객체가 JSON 형식으로 응답 본문에 작성된다  
    };  
}
```

✓ 반환 타입으로 사용할 수 있는 ReturnValueHandler

ReturnValue Handler

@ResponseBody
HttpEntity, ResponseEntity
HttpHeaders
ErrorResponse
ProblemDetail
String
View
java.util.Map, org.springframework.ui.Model
@ModelAttribute
ModelAndView object
FragmentsRendering, Collection<ModelAndView>
void
DeferredResult<V>
Callable<V>
ListenableFuture<V>, CompletionStage<V>, CompletableFuture<V>
ResponseBodyEmitter, SseEmitter
StreamingResponseBody
Other return values

handleReturnValue()

```
① returnValueHandlers = {ArrayList@8063} size = 17
> 0 = { ModelAndViewReturnValueHandler@8065}
> 1 = { ModelMethodProcessor@8066}
> 2 = { ViewMethodReturnValueHandler@8067}
> 3 = { ResponseBodyEmitterReturnValueHandler@8068}
> 4 = { StreamingResponseBodyReturnValueHandler@8069}
> 5 = { HttpEntityMethodProcessor@8070}
> 6 = { HttpHeadersReturnValueHandler@8071}
> 7 = { CallableMethodReturnValueHandler@8072}
> 8 = { DeferredResultMethodReturnValueHandler@8073}
> 9 = { AsyncTaskMethodReturnValueHandler@8074}
> 10 = { ServletModelAttributeMethodProcessor@8075}
> 11 = { RequestResponseBodyMethodProcessor@8076}
> 12 = { ViewNameMethodReturnValueHandler@8077}
> 13 = { MapMethodProcessor@8078}
> 14 = { CustomJsonReturnValueHandler@8079}
> 15 = { CustomHandlerMethodReturnValueHandler@8080}
> 16 = { ServletModelAttributeMethodProcessor@8081}
```

@ResponseBody

<https://github.com/onjsdnjs/spring-mvc-master/tree/@ResponseBody>

✓ 개요

- `@ResponseBody`는 메서드의 반환 값을 HTTP 응답 본문에 직접 직렬화하여 클라이언트에 전송하며 `HttpMessageConverter`를 사용하여 변환이 이루어진다
- 일반적으로 컨트롤러는 HTTP 요청을 처리하고 뷰(View)를 반환하는 방식으로 동작하는데 `@ResponseBody`를 사용하면 뷰를 반환하는 대신 메서드가 반환하는 객체를 바로 HTTP 응답 본문에 직렬화하여 전송하게 된다

✓ `@ResponseBody & @RestController`

- `ResponseBody`는 메서드뿐만 아니라 클래스 수준에서도 사용될 수 있으며 이와 같은 효과를 가진 것이 바로 `@RestController` 라 할 수 있다
- `@RestController`는 `@Controller`와 `@ResponseBody`를 결합한 메타 어노테이션으로서 `@RestController` 가 선언된 클래스는 모든 메서드에서 반환되는 값이 자동으로 `@ResponseBody`처럼 처리가 이루어진다

• 메서드

```
@GetMapping("/accounts/{id}")
@ResponseBody
public Account getAccount(@PathVariable Long id) {
    Account account = accountService.getAccountById(id);
    return account; // Account 객체가 JSON 으로 직렬화되어 응답 본문에 포함된다
}
```

• 클래스

```
@RestController
public class AccountController {

    @GetMapping("/accounts/{id}")
    public Account getAccount(@PathVariable Long id) {
        Account account = accountService.getAccountById(id);
        return account; // @RestController 때문에 자동으로 @ResponseBody가 적용된다
    }
}
```

요청

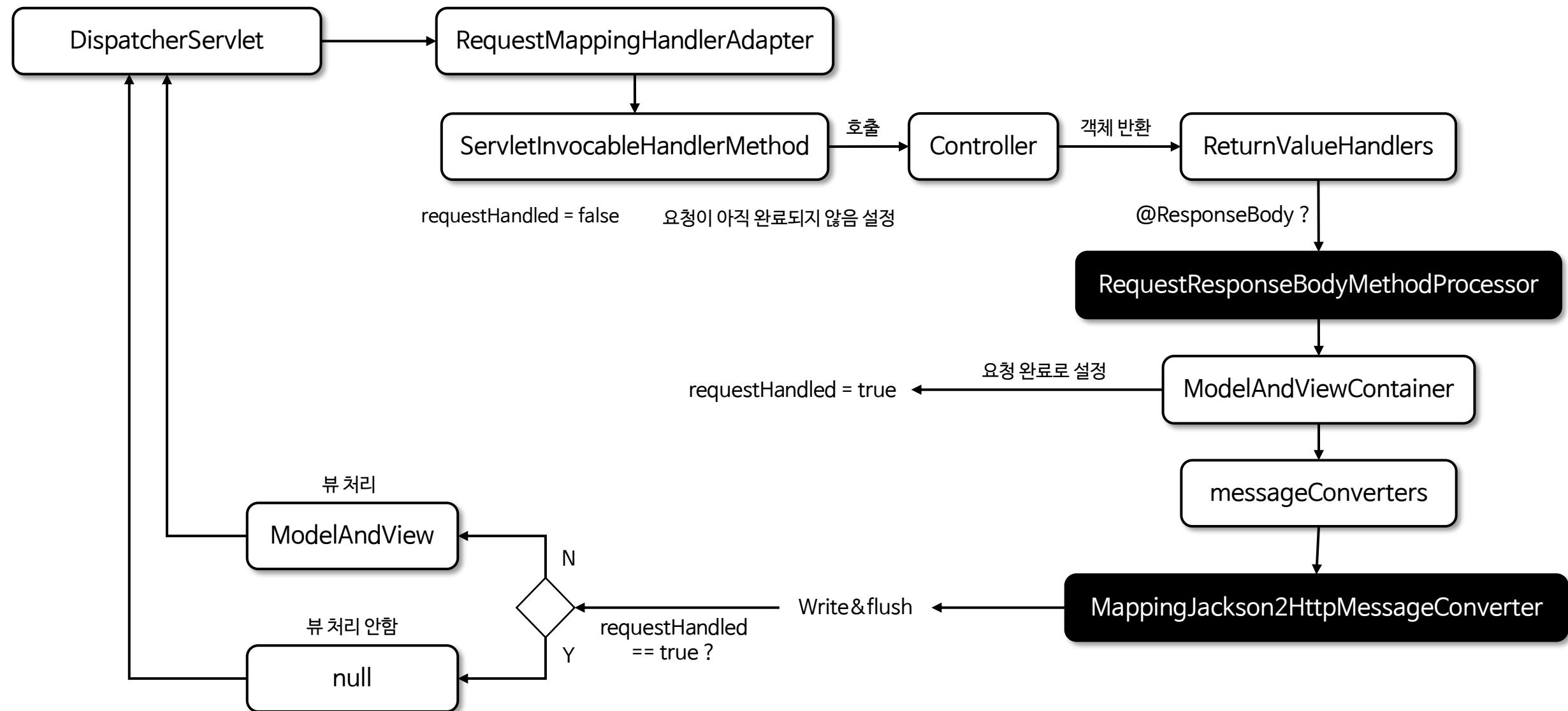
```
GET /accounts/1 HTTP/1.1
Host: localhost:8080
```

응답

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 45
```

```
{
    "id": 1,
    "name": "leaven",
    "age": 30
}
```

✓ HTTP 본문 응답 흐름도



ResponseEntity<T>

<https://github.com/onjsdnjs/spring-mvc-master/tree/ResponseEntity>

✓ 개요

- ResponseEntity<T> 는 HTTP 응답을 나타내는 클래스로서 주로 응답 상태 코드, 헤더, 본문을 제어하고 반환하는 데 사용되며 HttpEntity<T>를 상속하고 있다
- ResponseEntity<T> 는 @ResponseBody 와 비슷하지만 @ResponseBody 는 메서드 반환 값을 HTTP 응답 본문으로 기록하는 반면 ResponseEntity 는 상태 코드와 헤더 그리고 본문 까지 세밀하게 제어할 수 있는 기능을 제공한다
- ResponseEntity<T> 는 @RestController 나 @ResponseBody 가 없어도 적절한 HTTP 응답을 반환할 수 있다

✓ 구조

© <code>ResponseEntity<T></code>	
Ⓜ️ ⚡ <code>status(HttpStatusCode)</code>	BodyBuilder
Ⓜ️ ⚡ <code>internalServerError ()</code>	BodyBuilder
Ⓜ️ ⚡ <code>status (int)</code>	BodyBuilder
Ⓜ️ ⚡ <code>ok(T?)</code>	ResponseEntity <T>
Ⓜ️ ⚡ <code>notFound ()</code>	HeadersBuilder <?>
Ⓜ️ ⚡ <code>of(Optional <T>)</code>	ResponseEntity <T>
Ⓜ️ ⚡ <code>ok()</code>	BodyBuilder
Ⓜ️ ⚡ <code>equals (Object ?)</code>	boolean
Ⓜ️ ⚡ <code>getStatusCode ()</code>	HttpStatusCode
Ⓜ️ ⚡ <code>accepted ()</code>	BodyBuilder
Ⓜ️ ⚡ <code>ofNullable (T?)</code>	ResponseEntity <T>
Ⓜ️ ⚡ <code>noContent ()</code>	HeadersBuilder <?>
Ⓜ️ ⚡ <code>created (URI)</code>	BodyBuilder
Ⓜ️ ⚡ <code>hashCode ()</code>	int
Ⓜ️ ⚡ <code>of(ProblemDetail)</code>	HeadersBuilder <?>
Ⓜ️ ⚡ <code>badRequest ()</code>	BodyBuilder

`new ResponseEntity(HttpStatusCode status)`

`new ResponseEntity(T body, HttpStatusCode status)`

`new ResponseEntity(T body, MultiValueMap<String, String> headers, HttpStatusCode statusCode)`

✓ 상태 코드 설정 API

- **ResponseEntity.status(HttpStatus status)**: 지정한 상태 코드로 ResponseEntity를 생성한다
 - `ResponseEntity.status(HttpStatus.NOT_FOUND) → 404 상태 코드 설정`
- **ResponseEntity.ok()**: 200 OK 상태 코드를 설정한다
 - `ResponseEntity.ok().body(user)`
- **ResponseEntity.created(URI location)**: 201 Created 상태 코드와 함께, Location 헤더에 지정된 URI를 추가한다
 - `ResponseEntity.created(URI.create("/api/users/1")).build()`
- **ResponseEntity.noContent()**: 204 No Content 상태 코드를 설정합니다. 본문이 없는 응답을 보낼 때 사용한다
 - `ResponseEntity.noContent().build()`
- **ResponseEntity.accepted()**: 202 Accepted 상태 코드를 설정한다
 - `ResponseEntity.accepted().build()`
- **ResponseEntity.badRequest()**: 400 Bad Request 상태 코드를 설정한다
 - `ResponseEntity.badRequest().body(errorMessage)`
- **ResponseEntity.notFound()**: 404 Not Found 상태 코드를 설정한다
 - `ResponseEntity.notFound().build()`
- **ResponseEntity.internalServerError()**: 500 Internal Server Error 상태 코드를 설정한다
 - `ResponseEntity.internalServerError().body(errorMessage)`

✓ 본문 설정 API

- ResponseEntity.ok().body(T body), ResponseEntity.status(HttpStatus.OK).body(T body)

- 응답 본문을 설정한다

```
User user = userService.createUser(newUser);
return ResponseEntity.ok().body(user) or ResponseEntity.status(HttpStatus.OK).body(user)
```

- ResponseEntity.noContent().build()

- 본문이 없는 응답을 생성할 때 사용되며 상태 코드가 204 No Content 일 때 사용된다

✓ 헤더 설정 API

- ResponseEntity.ok().header(String headerName, String headerValue)

- 하나의 헤더를 설정한다

```
return ResponseEntity.ok().header(HttpHeaders.CONTENT_TYPE, "application/json").body(user)
```

- ResponseEntity.ok().headers(HttpHeaders headers)

- 여러 헤더를 설정할 수 있는 HttpHeaders 객체를 전달하여 여러 헤더를 한 번에 설정한다

```
HttpHeaders headers = new HttpHeaders();
headers.set("X-Custom-Header", "value");
headers.set("Y-Custom-Header", "value");
return ResponseEntity.ok().headers(headers).body(user);
```

✓ 구현 예제

생성자 방식

```
@PostMapping("/api/users")
public ResponseEntity<User> createUser(@RequestBody User newUser) {
    User createdUser = userService.createUser(newUser);

    HttpHeaders headers = new HttpHeaders();
    headers.setLocation(URI.create("/api/users/" + createdUser.getId()));
    headers.add("Custom-Header", "CreateUserHeader");

    return new ResponseEntity<>(createdUser, headers, HttpStatus.CREATED);
}
```

요청

```
POST /api/users HTTP/1.1
Content-Type: application/json
{
    "name": "leaven",
    "email": "leaven@example.com"
}
```

빌더 패턴

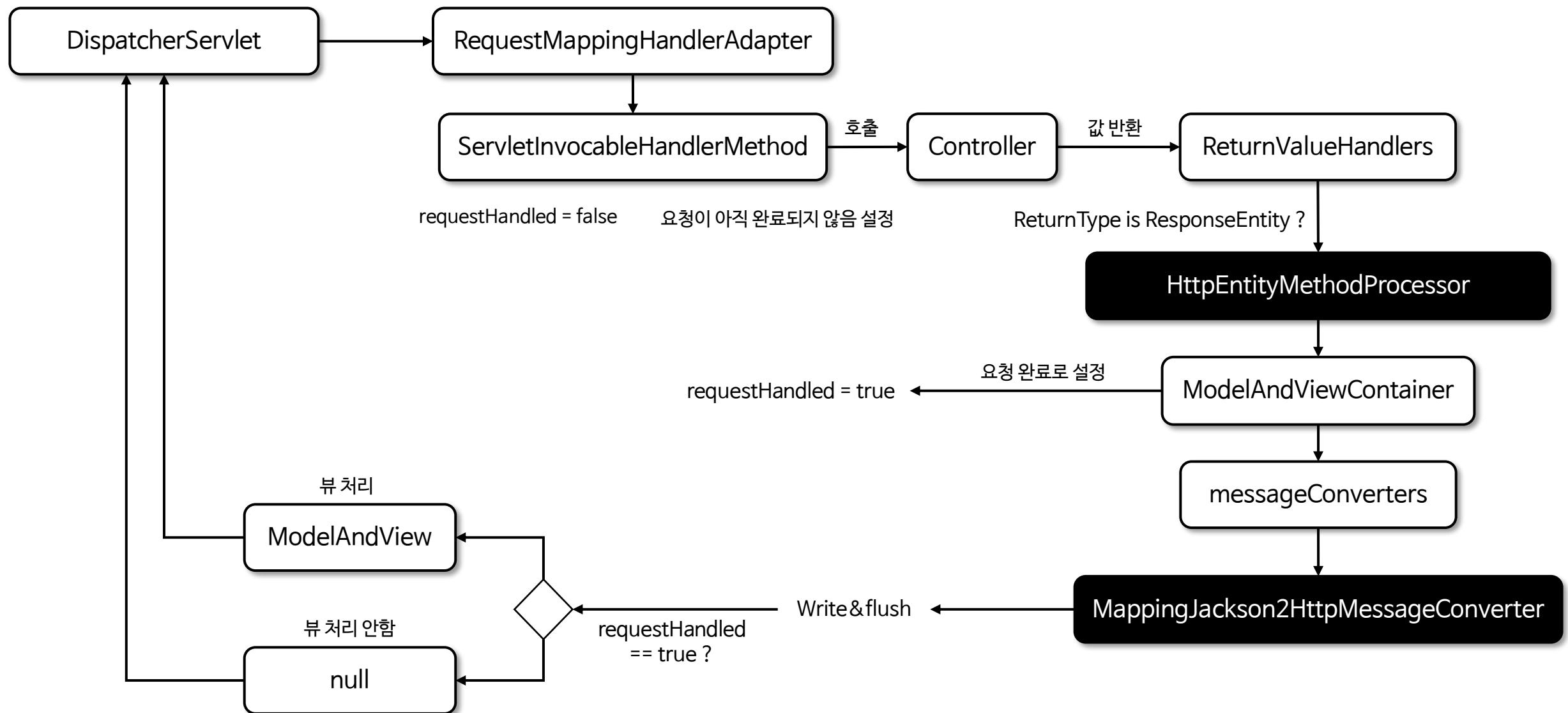
```
@PostMapping("/api/users")
public ResponseEntity<User> createUserWithBuilder(@RequestBody User newUser) {
    User createdUser = userService.createUser(newUser);

    return ResponseEntity
        .status(HttpStatus.CREATED)
        .header("Custom-Header", "CreateUserHeader")
        .location(URI.create("/api/users/" + createdUser.getId()))
        .body(createdUser);
}
```

응답

```
HTTP/1.1 201 Created
Custom-Header: CreateUserHeader
Location: /api/users/1
Content-Type: application/json
{
    "id": 1,
    "name": "leaven",
    "email": "leaven@example.com"
}
```

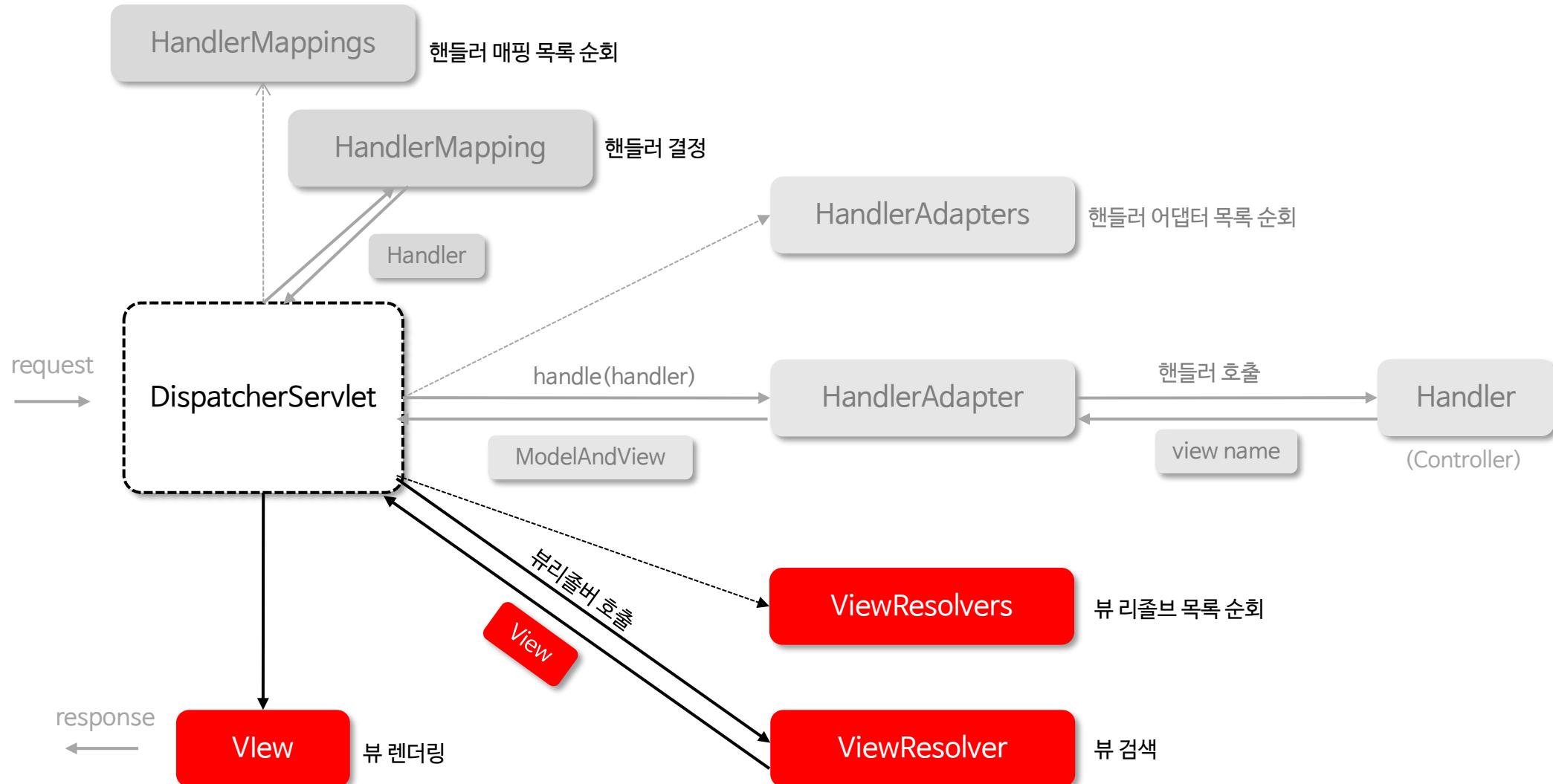
✓ 흐름도



View / ViewResolver

<https://github.com/onjsdnjs/spring-mvc-master/tree/View-ViewResolver>

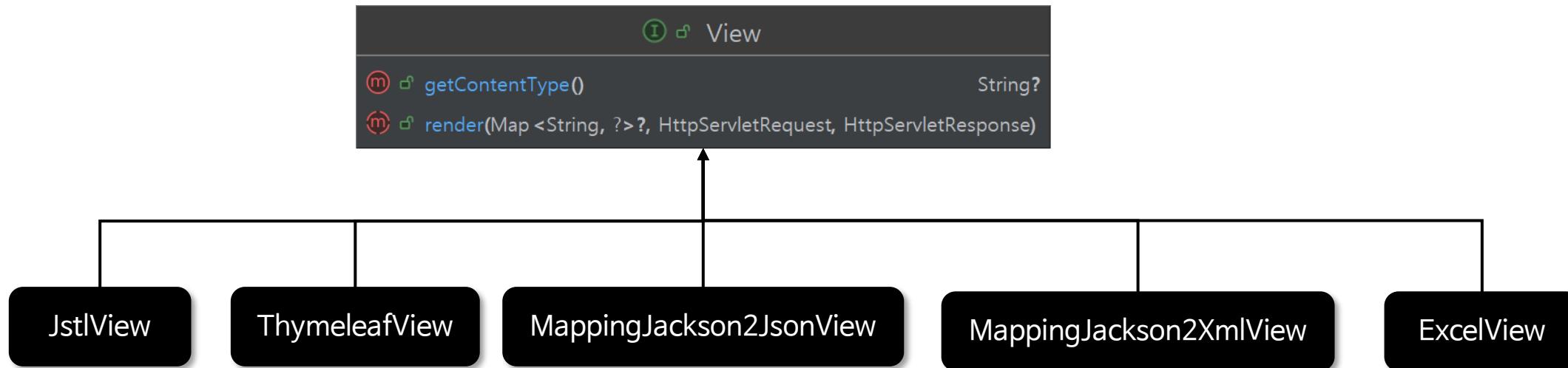
✓ 스프링 MVC 아키텍처



✓ View

- 스프링 MVC에서 View는 웹 페이지를 사용자에게 보여주는 역할을 한다
- View는 컨트롤러가 처리한 데이터를 사용자에게 보여주는 화면이며 사용자가 볼 수 있는 HTML, JSON 같은 결과물을 만들어주는 역할을 한다

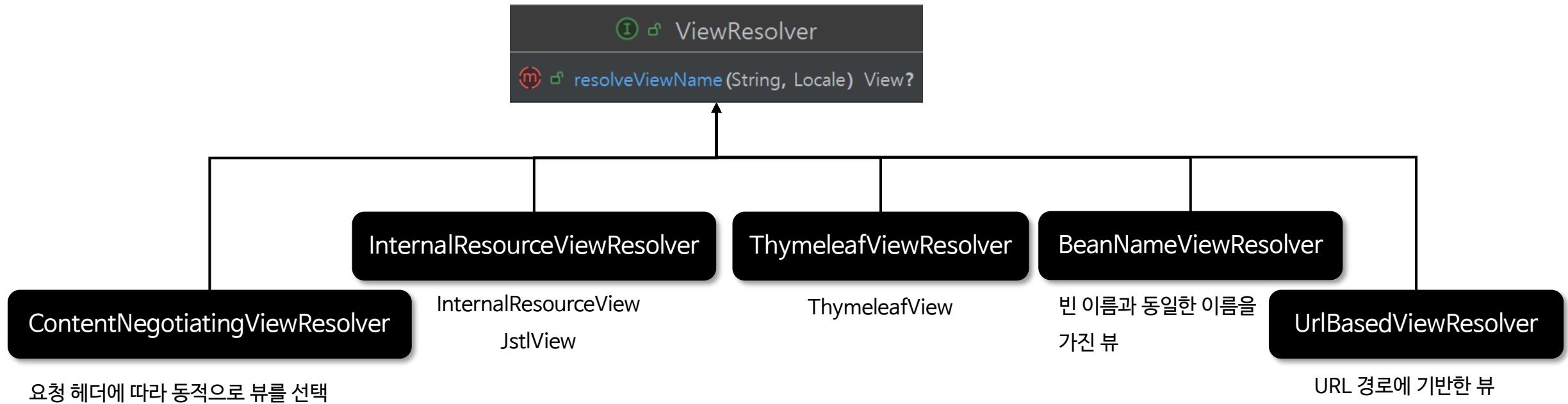
✓ 계층도



✓ ViewResolver

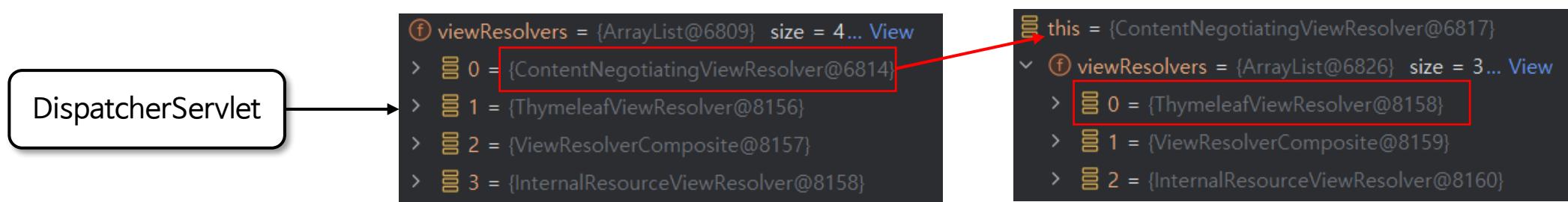
- 스프링의 View 시스템은 ViewResolver 와 View 인터페이스를 기반으로 동작하며 ViewResolver 는 특정 URL 요청에 따라 어떤 View를 사용할지를 결정하는 역할을 하고 View는 최종적으로 데이터를 렌더링하여 클라이언트에 반환한다

✓ 계층도



✓ ViewResolver 구현체

- 스프링 MVC는 초기화 시점에 자동으로 기본 ViewResolver 구현체를 생성하고 등록한다
- ContentNegotiatingViewResolver, InternalResourceViewResolver, ViewResolverComposite, 그리고 타임리프 의존성이 추가 되면 ThymeleafViewResolver 까지 포함한다
- ContentNegotiatingViewResolver 가 가장 우선 순위가 높으며 이 클래스 안에 ThymeleafViewResolver, InternalResourceViewResolver, ViewResolverComposite 가 목록으로 저장되어 있으며 ThymeleafViewResolver 가 우선순위가 가장 높다



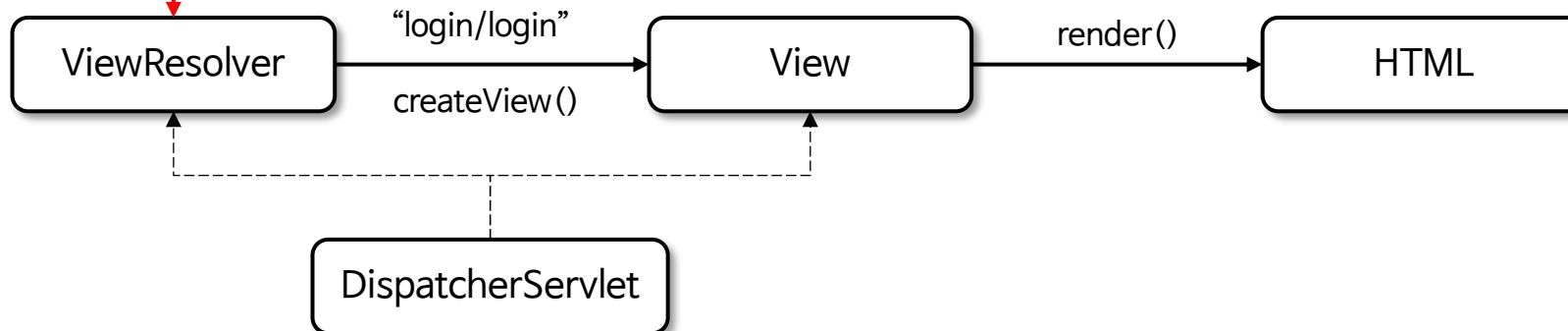
✓ ViewResolver의 View 결정 기준

- ContentNegotiatingViewResolver 은 각 ViewResolver 에 의해 생성된 View 객체들을 순회하며 가장 적합한 View 를 결정해서 반환한다
 - 클라이언트의 요청 헤더에 포함된 MediaType (Accept 헤더) 목록과 View 의 Content-Type 을 비교해서 서로 호환이 되는 MediaType 이 존재하는지 확인한다
 - MediaType 이 호환되는 첫 번째 View 가 최종 선택되어 반환되고 적합한 View 가 없으면 다음 View 로 넘어간다. 만약 View 가 없으면 예외가 발생한다
 - 만약 ThymeleafView 와 InternalResourceView 모두 선택 대상인데 ThymeleafView 가 먼저 선택되면 InternalResourceView 는 호환 여부를 검사하지 않는다

✓ View & ViewResolver 처리 구조

```
@GetMapping("/view")
public String view() {
    return "login/login";
}
```

뷰 이름 반환

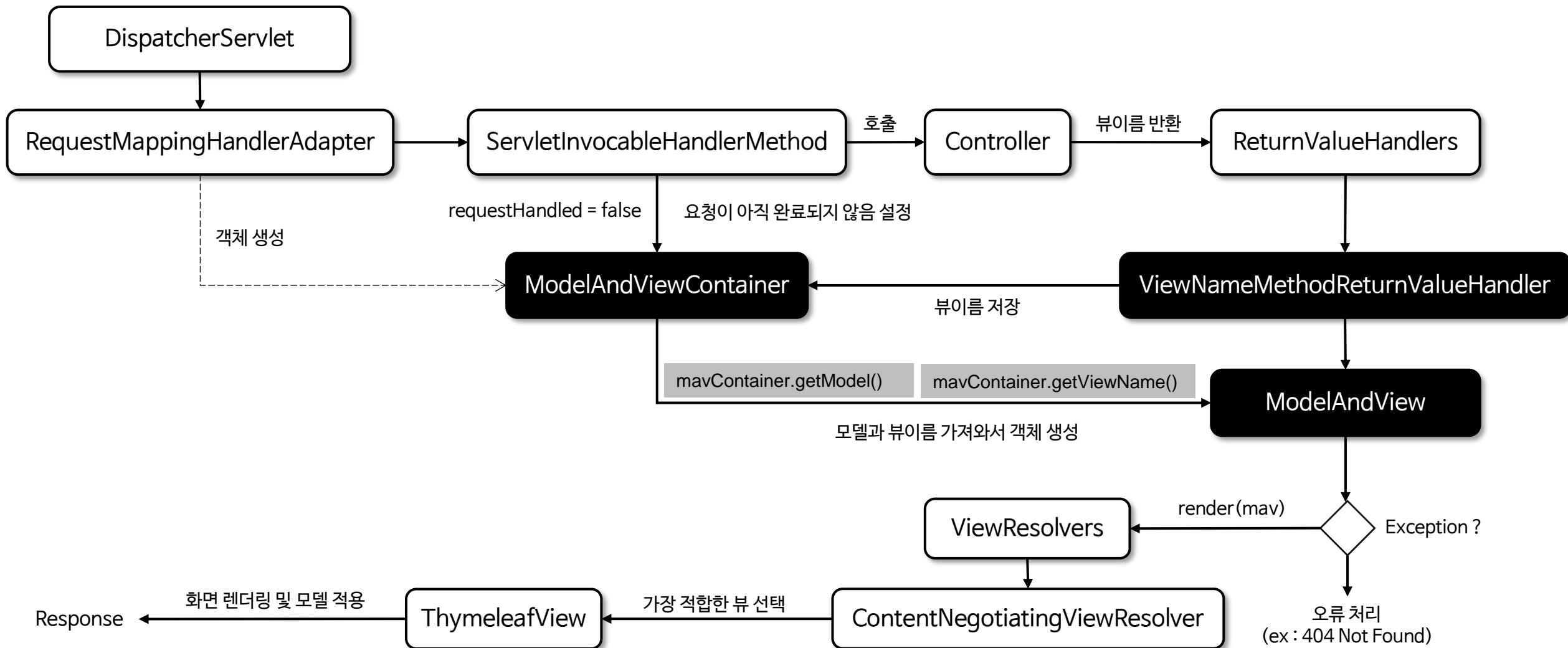


✓ View vs HttpMessageConverter

View	HttpMessageConverter
템플릿 기반 HTML 렌더링	REST API 응답 데이터 변환
모델 데이터를 기반으로 HTML 등으로 변환	Java 객체를 JSON, Text 등으로 변환
@Controller, @RequestMapping	@RestController, @ResponseBody, ResponseEntity
HTML, JSP와 같은 화면 렌더링	JSON, Text 를 포함한 API 응답
ViewResolver에 의해 View 선택	HttpMessageConverter는 ViewResolver 와 독립적

✓ View / ViewResolver 흐름도

- 클라이언트의 요청을 처리할 핸들러를 실행한다



Thymeleaf

<https://github.com/onjsdnjs/spring-mvc-master/tree/Thymeleaf>

✓ 개요

- 스프링에서 ViewTemplate 은 웹 애플리케이션의 뷰를 구성하고 렌더링하기 위한 템플릿 엔진을 가리킨다
- ViewTemplate 을 통해 서버에서 처리된 데이터를 HTML 이나 다른 형식으로 출력할 수 있으며 모델 데이터를 뷰로 전달하여 클라이언트에 보여주는 역할을 한다
- 스프링에서 사용되는 ViewTemplate 엔진으로는 Thymeleaf 와 JSP 가 있으며 요즘은 대부분 Thymeleaf 엔진을 사용하고 있다

✓ Thymeleaf 특징

1. 자연 템플릿 (Natural Templates)

- Thymeleaf 는 HTML 파일이 브라우저에서 열렸을 때도 정상적인 HTML 페이지로 보일 수 있도록 설계된 "자연 템플릿" 기능을 제공한다.
- 즉, Thymeleaf 템플릿은 서버 측 렌더링 없이도 개발자가 미리보기 할 수 있다 (JSP 은 서블릿의 일종이기 때문에 서블릿컨네이터 안에서 렌더링이 필요하다)

2. HTML 친화적인 문법

- Thymeleaf 는 HTML5 와 잘 호환되도록 설계되어 HTML 문서 작성과 비슷하게 템플릿을 구성할 수 있고 스프링 부트와 완벽하게 통합되어 쉽게 설정하고 사용할 수 있다

3. 더 강력한 표현식 지원

- Thymeleaf는 다양한 표현식을 지원하며 복잡한 데이터 처리를 위한 기능을 내장하고 있다. 예를 들어, 조건문(th:if, th:unless), 반복문(th:each), URL 처리, 메시지 처리 등을 다양한 속성으로 지원한다

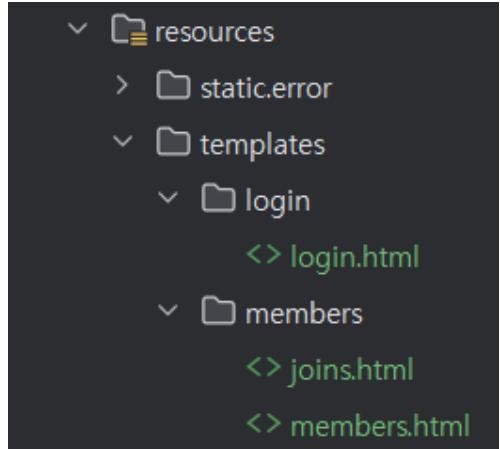
✓ Thymeleaf 사용법

1) 의존성 추가

```
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

2) 기본 디렉토리 구조

- 기본적으로 src/main/resources/templates 폴더가 루트 위치가 된다
- templates 폴더 아래에 원하는 폴더 및 .html 파일을 생성한다



3) HTML 사용 선언

```
<html xmlns:th="http://www.thymeleaf.org">
```

4) 컨트롤러 구현

```
@Controller  
public class UserController {  
  
    @GetMapping("/user")  
    public String user(Model model) {  
        User user = new User("leaven", 30, "leaven@example.com");  
        model.addAttribute("user", user);  
  
        return "members/joins"; // templates 폴더의 members 폴더의 joins.html 파일을 렌더링  
    }  
}
```

5) HTML 작성

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
    <head>  
        <meta charset="UTF-8">  
    </head>  
    <body>  
        <h1>User Profile</h1>  
        <p>Name:</strong> <span th:text="${user.name}">Name</span></p>  
        <p>Age:</strong> <span th:text="${user.age}">Age</span></p>  
    </body>  
</html>
```

<https://www.thymeleaf.org/>

RedirectAttributes Flash Attributes

<https://github.com/onjsdnjs/spring-mvc-master/tree/RedirectAttributes-FlashAttributes>

✓ 개요

- 웹 애플리케이션에서 페이지 이동 중 데이터 전달이 필요한 경우가 발생한다. 예를 들어, 사용자가 폼을 제출한 후, 성공 메시지나 에러 메시지를 다음 화면에서 표시하고자 할 때, 리다이렉트를 사용하여 다른 URL로 이동시키면서 데이터를 전달해야 하는 상황이 자주 발생할 수 있다

✓ 문제점

- URL에 데이터 포함
 - 리다이렉트 요청에서 데이터를 전달하는 가장 기본적인 방법은 쿼리 파라미터를 URL에 추가하는 것이다. 하지만 이 방식은 보안적으로 불안정하며 사용자가 URL에서 민감한 정보를 볼 수 있다는 단점이 있다
- 세션 사용
 - 세션을 이용하여 데이터를 전달할 수도 있지만 세션은 다음 요청 이후에도 남아 있는 데이터를 수동으로 제거해야 하며 메모리를 많이 사용할 수 있다
- 일회성 데이터 필요
 - 많은 경우 리다이렉트 후 한 번만 사용할 데이터를 전달하는 것이 필요하다. 하지만 기본 세션이나 URL 쿼리 파라미터는 이 요구를 충족하지 못한다

✓ RedirectAttributes 와 Flash Attributes 의 등장

- RedirectAttributes 는 리다이렉트 요청 시 데이터를 안전하고 효율적으로 전달할 수 있도록 돋는 인터페이스로서 리다이렉트 요청 간에 필요한 데이터를 URL에 포함할 수 있으며 Flash Attributes 를 사용해서 URL에 표시되지 않도록 임시 데이터를 세션을 통해 전달할 수도 있다

✓ Post-Redirect-Get 패턴

- RedirectAttributes 와 Flash Attributes는 주로 Post-Redirect-Get (PRG) 패턴에서 유용하게 사용되는데 사용자가 폼을 제출한 후 같은 URL에서 폼을 다시 불러오는 대신 다른 페이지로 리다이렉트 시키는 패턴을 말한다
- 이 패턴을 통해 중복 제출 방지 및 리다이렉트된 페이지에서 Flash Attributes를 통해 성공 또는 에러 메시지를 전달하여 사용자에게 정보를 표시할 수 있다

✓ RedirectAttributes

RedirectAttributes	
Ⓜ️ ⌂ addFlashAttribute (Object)	RedirectAttributes
Ⓜ️ ⌂ addAttribute (String, Object?)	RedirectAttributes
Ⓜ️ ⌂ addAttribute (Object)	RedirectAttributes
Ⓜ️ ⌂ addAllAttributes (Collection <?>)	RedirectAttributes
Ⓜ️ ⌂ addFlashAttribute (String, Object?)	RedirectAttributes
Ⓜ️ ⌂ getFlashAttributes()	Map <String, ?>
Ⓜ️ ⌂ mergeAttributes(Map <String, ?>)	RedirectAttributes

- 자동 생성된 이름으로 플래시 속성을 추가한다
- 지정된 이름과 값을 URL 쿼리 매개변수로 추가한다
- 객체의 이름을 자동 생성하여 URL 쿼리 매개변수로 추가한다
- 여러 객체를 한 번에 URL 쿼리 매개변수로 추가한다
- 이름과 값을 가진 플래시 속성을 추가하여 일회성 데이터로 사용한다(URL에 노출되지 않음).
- 현재 플래시 속성들을 반환한다
- 기존 속성을 유지하며 새로운 속성을 덧붙인다

✓ 구현 예제

```
@PostMapping("/user/save")
public String saveUser(@ModelAttribute("user") User user, RedirectAttributes redirectAttributes) {
    userService.save(user);
    // URL에 파라미터 값 노출
    redirectAttributes.addAttribute("userId", user.getId());
    // URL에 파라미터 값 노출되지 않음
    redirectAttributes.addFlashAttribute("message", "사용자가 성공적으로 저장되었습니다.");
    return "redirect:/user/success"; // 리다이렉트 - PRG 패턴 적용
}
@GetMapping("/user/success")
public String successPage(Model model) {
    return "successPage";
}
```

```
<!DOCTYPE html>
<html>
<body>
    <!-- 플래시 속성에서 메시지를 읽어와 표시 -->
    <div th:if="${message}">
        <p th:text="${message}"></p>
    </div>
    <p>User ID: <span th:text="${userId}"></span></p>
</body>
</html>
```

✓ 구현 예제 - URL 파라미터에 노출되는 방식

```
@PostMapping("/user/save")
public String saveUser(@ModelAttribute("user") User user, RedirectAttributes redirectAttributes) {
    userService.save(user);
    // URL에 파라미터 값 노출
    redirectAttributes.addAttribute("userId", user.getId());
    return "redirect:/user/success"; // 리다이렉트 - PRG 패턴 적용
}
@GetMapping("/user/success")
public String successPage(String userId, Model model) {
    System.out.println("userId : " + userId);
    return "successPage";
}
```

http://localhost:8080/user/success?userId=leaven

RedirectAttributesModelMap

RedirectAttributesModelMap	
Ⓜ️ ⌂ addAllAttributes (Collection<?>?)	RedirectAttributesModelMap
Ⓜ️ ⌂ addAllAttributes (Map<String, ?>?)	RedirectAttributesModelMap
Ⓜ️ ⌂ putAll(Map<String, Object>?)	void
Ⓜ️ ⌂ addFlashAttribute (Object)	RedirectAttributes
Ⓜ️ ⌂ addAttribute (String, Object?)	RedirectAttributesModelMap
Ⓜ️ ⌂ formatValue(Object?)	String?
Ⓜ️ ⌂ addFlashAttribute (String, Object?)	RedirectAttributes
Ⓜ️ ⌂ put(String, Object?)	Object?
Ⓜ️ ⌂ addAttribute (Object)	RedirectAttributesModelMap
Ⓜ️ ⌂ asMap()	Map<String, Object>
Ⓜ️ ⌂ getFlashAttributes()	Map<String, ?>
Ⓜ️ ⌂ mergeAttributes (Map<String, ?>?)	RedirectAttributesModelMap

✓ 구현 예제 - URL 파라미터에 노출되지 않는 방식

```
@GetMapping("/setFlash")
public String setFlashAttribute(RedirectAttributes redirectAttributes) {
    redirectAttributes.addFlashAttribute("flashMessage", "이것은 1회성 플래시 메시지입니다.");
    return "redirect:/getFlash";
}

@GetMapping("/getFlash")
public String getFlashAttribute(Model model) {
    // 플래시 속성에서 메시지를 가져와 출력
    String message = (String) model.asMap().get("flashMessage");
    return "redirect:/getFlashAgain";
}

// 다시 요청하여 플래시 속성이 유지되는지 확인하는 메서드
@GetMapping("/getFlashAgain")
public String getFlashAttributeAgain(Model model) {
    String message = (String) model.asMap().get("flashMessage");
    return "logi/login";
}
```

- FlashAttribute 방식으로 저장된 데이터는 내부적으로 Model 객체에 바인딩 된다

✓ Flash Attributes

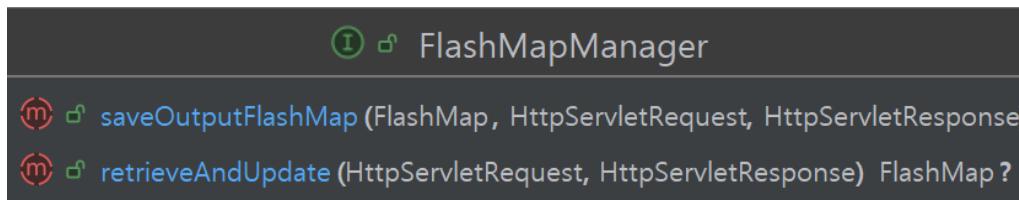
- Spring MVC는 Flash Attributes 을 지원하기 위해 두 가지 주요 추상화인 FlashMap 과 FlashMapManager 를 제공하며 FlashMap 은 플래시 속성을 저장하는 데 사용되고 FlashMapManager 는 FlashMap 객체를 저장, 조회 및 관리하는 역할을 한다

✓ FlashMap

- FlashMap 은 리다이렉트 후 다음 요청에서만 사용할 수 있는 임시적인 속성을 저장하는 Map 형태의 객체로서 리다이렉트가 일어나면 이 FlashMap 이 세션에 저장되고 다음 요청에서 자동으로 제거된다
- 현재 요청에서 이후 요청으로 데이터를 전달하는 출력 FlashMap 과 이전 요청에서 전달된 데이터를 담고 있는 입력 FlashMap 으로 구분하며 FlashMap 객체는 RequestContextUtils 의 정적 메서드를 통해 Spring MVC 의 어디서든 접근이 가능하다

✓ FlashMapManager

- FlashMapManager 는 FlashMap 객체를 생성하고 관리하는 역할을 하며 FlashMap 을 세션에 저장하고 다음 요청에서 이를 가져거나 제거하는 작업을 수행한다



saveOutputFlashMap

- 리다이렉트 전에 호출하여 HTTP 세션이나 응답 쿠키에 FlashMap을 저장한다

retrieveAndUpdate

- 이전 요청에서 저장된 FlashMap 중 현재 요청과 일치하는 것을 찾아 이를 저장소에서 제거하고, 만료된 다른 FlashMap 객체도 함께 제거한다
- 이 메서드는 요청이 시작될 때마다 호출된다

✓ FlashMap & FlashMapManager 구현 예제

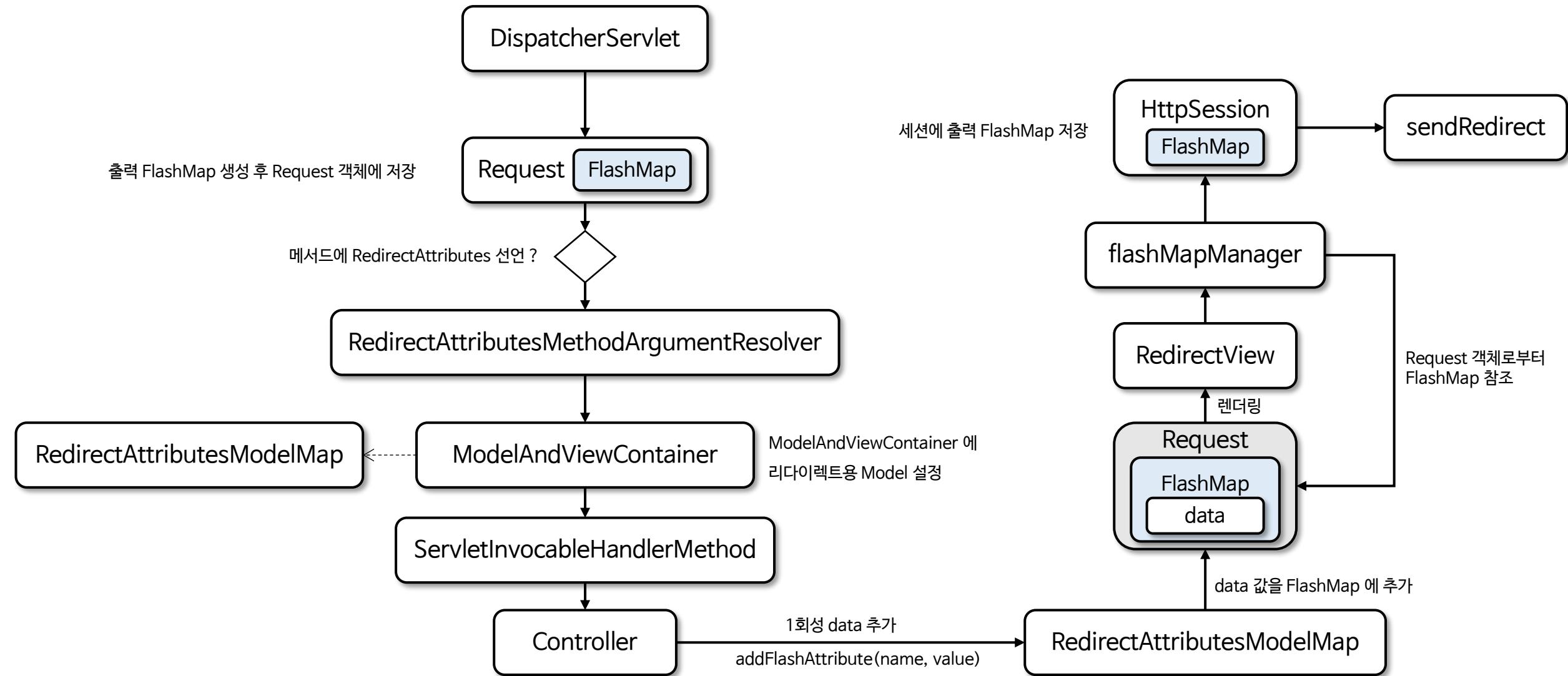
- 대부분의 일반적인 리다이렉트 시나리오에서는 RedirectAttributes를 사용하는 것이 더 간편하고 충분하지만 특수한 상황이나 세밀한 제어가 필요한 경우에는 FlashMap과 FlashMapManager를 직접 사용하는 것도 가능하다

```
@GetMapping("/setFlashAttribute")
public RedirectView setFlashAttribute(HttpServletRequest request, HttpServletResponse response) {
    FlashMapManager flashMapManager = RequestContextUtils.getFlashMapManager(request); // FlashMapManager 가져오기
    FlashMap outputFlashMap = RequestContextUtils.getOutputFlashMap(request);
    outputFlashMap.put("message", "플래시 속성이 설정되었습니다!");
    if (flashMapManager != null) {
        flashMapManager.saveOutputFlashMap(outputFlashMap, request, response); // FlashMapManager를 통해 FlashMap을 세션에 저장
    }
    return new RedirectView("/getFlashAttribute");
}

@GetMapping("/getFlashAttribute")
public String getFlashAttribute(HttpServletRequest request, Model model) {
    FlashMap inputFlashMap = RequestContextUtils.getInputFlashMap(request); // 입력 FlashMap을 가져옴
    if (inputFlashMap != null) {
        model.addAttribute("message", inputFlashMap.get("message"));
    } else {
        model.addAttribute("message", "플래시 속성이 없습니다.");
    }
    return "displayMessage";
}
```

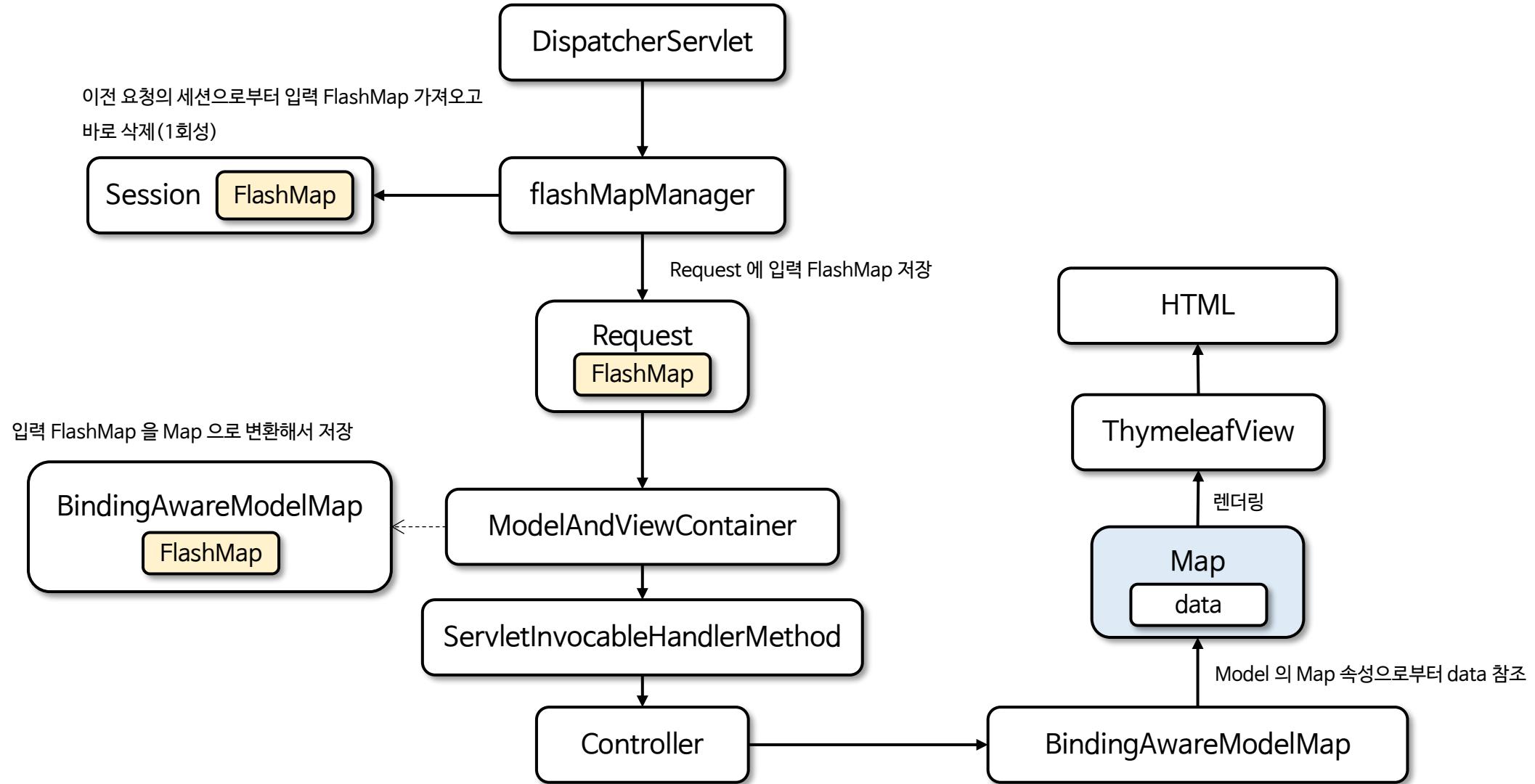
✓ PRG 흐름도

1. Post-Redirect



✓ PRG 흐름도

2. Redirect-Get





스프링 웹 MVC 완전 정복

✓ 스프링 웹 MVC 실행 구조 이해

1. 자바 리플렉션 이해
2. 리플렉션 실전 예제
3. 핸들러 메서드 호출 원리
4. 메서드 파라미터 실행 구조 이해
5. 메서드 파라미터 커스텀 구현

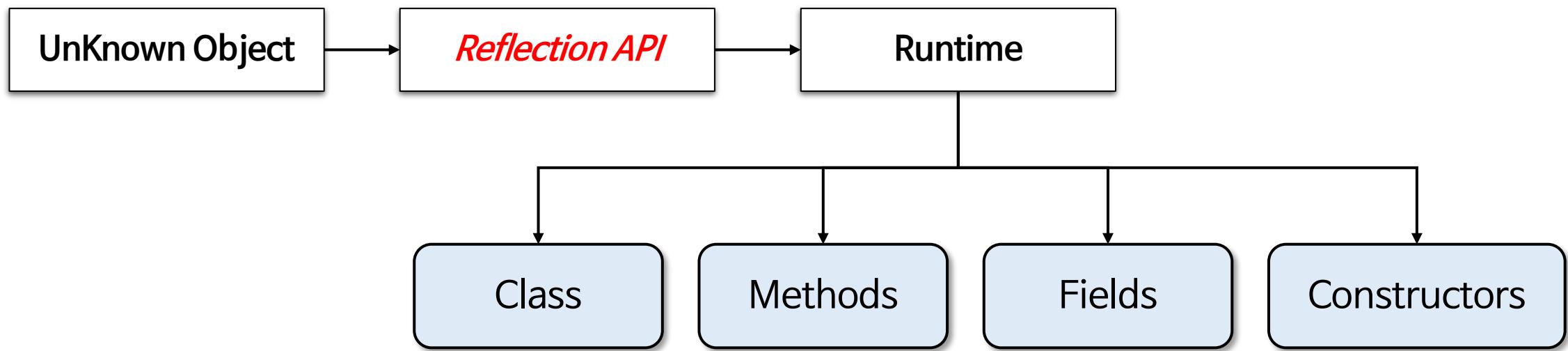
자바 리플렉션 이해

<https://github.com/onjsdnjs/spring-mvc-master/tree/자바-리플렉션-이해>

✓ 개요

- 리플렉션(Reflection)은 어플리케이션의 실행 시점에 클래스, 메서드, 필드 등을 분석하고 조작할 수 있는 기능을 제공하는 강력한 도구로서 객체의 클래스 타입 정보를 동적으로 가져오고 객체의 메서드 실행이나 필드 값을 읽고 수정하는 등의 작업을 수행할 수 있다

✓ 동작 방식



임의의 객체를 동적으로 전달받아 클래스타입을 참조하고 클래스타입으로 부터 메서드, 필드, 생성자 등을 참조할 수 있다

✓ 주요 클래스

• Class

- Reflection 의 핵심 클래스로서 클래스의 메타데이터를 표현한다
- 참조 방식
 - **Class.forName(String className)**
 - `Class<?> clazz = Class.forName("java.util.ArrayList");`
 - **.class 키워드**
 - `Class<?> clazz = String.class;`
 - **.getClass() 메서드**
 - `Class<?> clazz = str.getClass();`

• Field

- 클래스의 필드를 나타내며 필드 이름, 타입, 접근 제어자, 값을 조회 및 수정할 수 있다
- 참조 방식
 - **getDeclaredFields()**
 - private 접근 제어자를 가진 필드 포함
 - `Field[] fields = clazz.getDeclaredFields();`
 - **getDeclaredField(String fieldName)**
 - `Field field = clazz.getDeclaredField("name");`
 - **getFields()**
 - public 접근 제어자를 가진 필드만 참조 가능
 - `Field[] publicFields = clazz.getFields();`

✓ 주요 클래스

• Method

- 클래스의 메서드를 나타내며 메서드 이름, 반환 타입, 매개변수 타입, 접근 제어자 정보를 조회할 수 있으며 런타임에 메서드를 실행할 수 있다
- 참조 방식
 - **getDeclaredMethods**
 - private 접근 제어자를 가진 필드 포함
 - Method[] methods = clazz.getDeclaredMethods();
 - **getDeclaredMethod(String methodName, Class<?>... parameterTypes)**
 - Method method = clazz.getDeclaredMethod("add", int.class, Object.class);
 - **getMethods()**
 - public 접근 제어자를 가진 필드만 참조 가능
 - Method[] publicMethods = clazz.getMethods(); (public 접근 제어자를 가진 필드만)

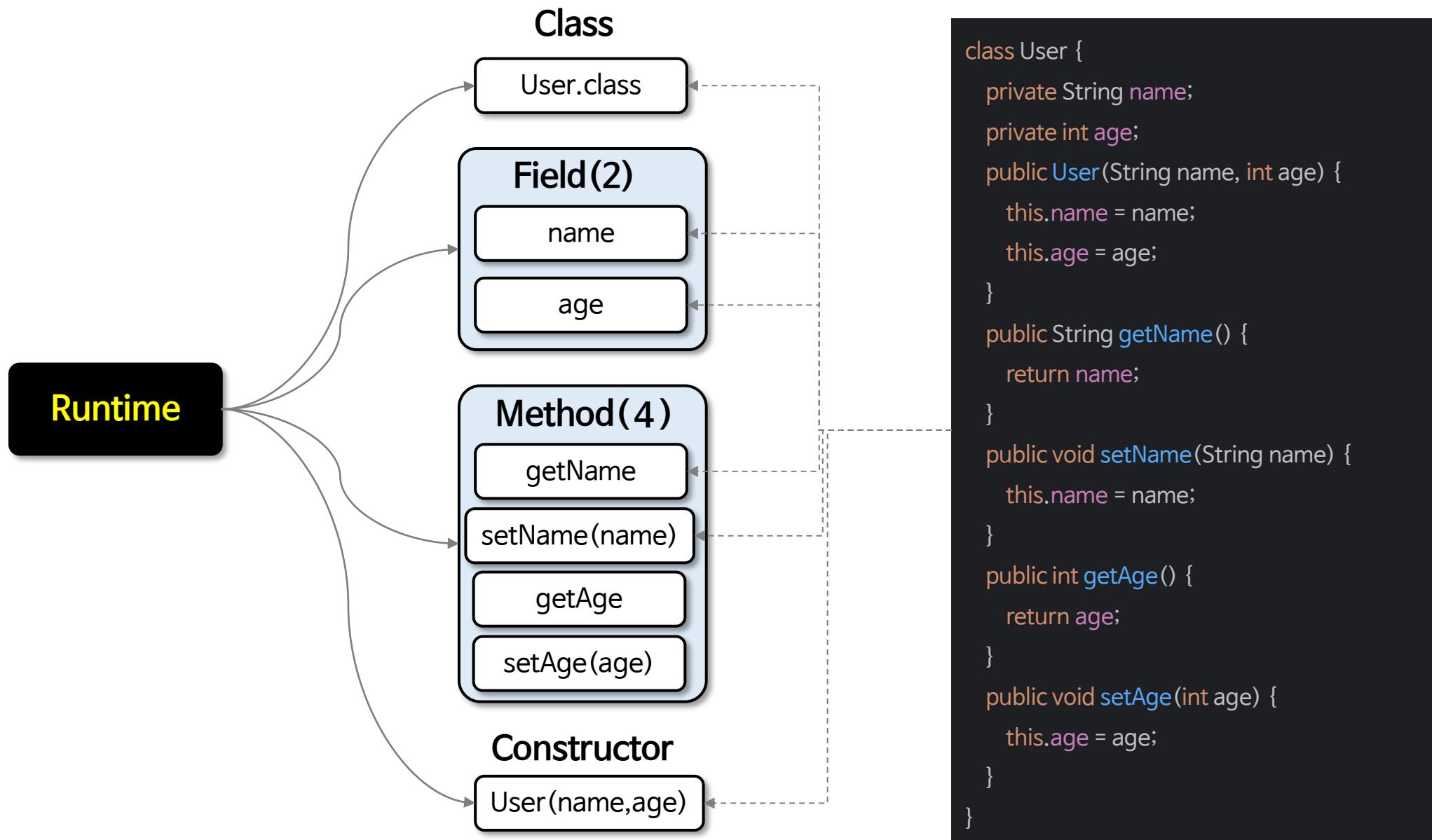
• Constructor

- 클래스의 생성자를 나타내며 생성자 정보 조회 및 객체 생성에 사용된다
- 참조 방식
 - **getDeclaredConstructors()** – private 접근 제어자를 가진 필드 포함
 - Constructor<?>[] constructors = clazz.getDeclaredConstructors();
 - **getDeclaredConstructor(Class<?>... parameterTypes)**
 - Constructor<?> constructor = clazz.getDeclaredConstructor(String.class);
 - **getConstructors()** – public 접근 제어자를 가진 필드만 참조 가능
 - Constructor<?>[] publicConstructors = clazz.getConstructors(); (public 접근 제어자를 가진 필드만)

✓ 각 클래스의 주요 API

클래스	메서드	설명
Class	getDeclaredFields()	모든 필드 가져오기
	getDeclaredMethods()	모든 메서드 가져오기
	getDeclaredConstructors()	모든 생성자 가져오기
	forName(String name)	클래스 이름으로 Class 객체 가져오기
Field	getName()	필드 이름 가져오기
	getType()	필드 타입 가져오기
	setAccessible(boolean flag)	접근 제어자 무시 설정
	get(Object obj)	특정 객체의 필드 값 가져오기
	set(Object obj, Object value)	특정 객체의 필드 값 수정
Method	getName()	메서드 이름 가져오기
	getReturnType()	메서드 반환 타입 가져오기
	getParameterTypes()	매개변수 타입 배열 가져오기
	setAccessible(boolean flag)	접근 제어자 무시 설정
	invoke(Object obj, Object...)	메서드 실행
Constructor	getParameterTypes()	생성자의 매개변수 타입 배열 가져오기
	newInstance(Object...)	새로운 객체 생성

✓ 리플렉션 실행 결과



✓ 클래스 정보 조회

```
public class ReflectionExample {  
    public static void main(String[] args) {  
        try {  
            Class<?> clazz = Class.forName("java.util.ArrayList");  
            //Class<?> clazz = ArrayList.class  
            //Class<?> clazz = new ArrayList().getClass();  
  
            System.out.println("Class Name: " + clazz.getName());  
            System.out.println("Superclass: " + clazz.getSuperclass());  
  
            System.out.println("\nImplemented Interfaces:");  
            for (Class<?> inf : clazz.getInterfaces()) {  
                System.out.println(" " + inf.getName());  
            }  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

✓ 생성자 호출 및 객체 생성

```
class User {  
    private String username;  
  
    public User(String username) {  
        this.username = username;  
    }  
    @Override  
    public String toString() {  
        return "User{" + "username='" + username + '\'' + '}';  
    }  
}  
  
public class ReflectionConstructorExample {  
    public static void main(String[] args) throws Exception {  
        Class<?> clazz = User.class;  
  
        // 생성자 호출  
        Constructor<?> constructor = clazz.getConstructor(String.class);  
        User user = (User) constructor.newInstance("springmvc");  
  
        System.out.println("Created User: " + user);  
    }  
}
```

✓ 필드 정보 조회 및 조작

```
@Data
class User {
    private String name = "Default Name";
    private int age = 25;
}

public class ReflectionFieldExample {
    public static void main(String[] args) throws Exception {
        User user = new User();
        Class<?> clazz = user.getClass();

        for (Field field : clazz.getDeclaredFields()) {
            field.setAccessible(true); // private 필드 접근 허용
            System.out.println("Field Name: " + field.getName());
            System.out.println("Field Value: " + field.get(user));

            if (field.getType() == String.class) {
                field.set(user, "springmvc");
            }
        }

        System.out.println("Updated Name: " + clazz.getDeclaredField("name").get(user));
    }
}
```

✓ 메서드 정보 조회 및 조작

```
public static void main(String[] args) {
    Class<?> clazz = Class.forName("User");
    Object userInstance = clazz.getDeclaredConstructor().newInstance();

    // setName 메서드 호출
    Method setName = clazz.getDeclaredMethod("setName", String.class);
    setName.invoke(userInstance, "springmvc");

    // setAge 메서드 호출
    Method setAge = clazz.getDeclaredMethod("setAge", int.class);
    setAge.invoke(userInstance, 25);

    // getName 메서드 호출
    Method getName = clazz.getDeclaredMethod("getName");
    getName.invoke(userInstance)

    // getAge 메서드 호출
    Method getAge = clazz.getDeclaredMethod("getAge");
    getAge.invoke(userInstance);
}
```

```
@Data
public class User {
    private String name;
    private int age;
}
```

리플렉션 실전 예제

<https://github.com/onjsdnjs/spring-mvc-master/tree/리플렉션-실전-예제>

✓ 개요

- 이 예제는 Java Reflection을 사용하여 객체의 필드(Field) 와 메서드(Method) 정보를 동적으로 관리하고 실행하는 기능을 제공한다
- 핵심 클래스로 ReflectionFieldManager, ReflectionMethodManager, ReflectionExecutor 를 사용한다

✓ ReflectionFieldManager - 필드 관련 작업을 관리

```
class ReflectionFieldManager {  
    private final Map<String, Field> fieldMap = new HashMap<>();  
  
    public ReflectionFieldManager(Class<?> clazz) {  
        for (Field field : clazz.getDeclaredFields()) {  
            field.setAccessible(true);  
            fieldMap.put(field.getName(), field);  
        }  
    }  
  
    public Field getField(String fieldName) {  
        return fieldMap.get(fieldName);  
    }  
  
    public void setFieldValue(Object target, String fieldName, Object value) throws IllegalAccessException {  
        Field field = getField(fieldName);  
        if (field != null) field.set(target, value);  
    }  
  
    public Object getFieldValue(Object target, String fieldName) throws IllegalAccessException {  
        Field field = getField(fieldName);  
        if (field != null) return field.get(target);  
        return null;  
    }  
}
```

✓ ReflectionMethodManager – 메서드 관련 작업을 관리

```
// 메서드 객체를 저장, 조회할 수 있는 클래스
class ReflectionMethodManager {

    private final Map<String, Method> methodMap = new HashMap<>();

    public ReflectionMethodManager(Class<?> clazz) {
        for (Method method : clazz.getDeclaredMethods()) {
            method.setAccessible(true);
            methodMap.put(method.getName(), method);
        }
    }

    public Method getMethod(String methodName) {
        return methodMap.get(methodName);
    }

    public Object invokeMethod(Object target, String methodName, Object... args) throws Exception {
        Method method = getMethod(methodName);
        if (method != null) {
            return method.invoke(target, args);
        }
        return null;
    }
}
```

✓ ReflectionExecutor - 필드와 메서드를 실행하는 역할

```
class ReflectionExecutor {  
    private final ReflectionFieldManager fieldManager;  
    private final ReflectionMethodManager methodManager;  
  
    public ReflectionExecutor(Class<?> clazz) {  
        this.fieldManager = new ReflectionFieldManager(clazz);  
        this.methodManager = new ReflectionMethodManager(clazz);  
    }  
  
    public ReflectionFieldManager getFieldManager() {  
        return fieldManager;  
    }  
  
    public ReflectionMethodManager getMethodManager() {  
        return methodManager;  
    }  
}
```

✓ Main

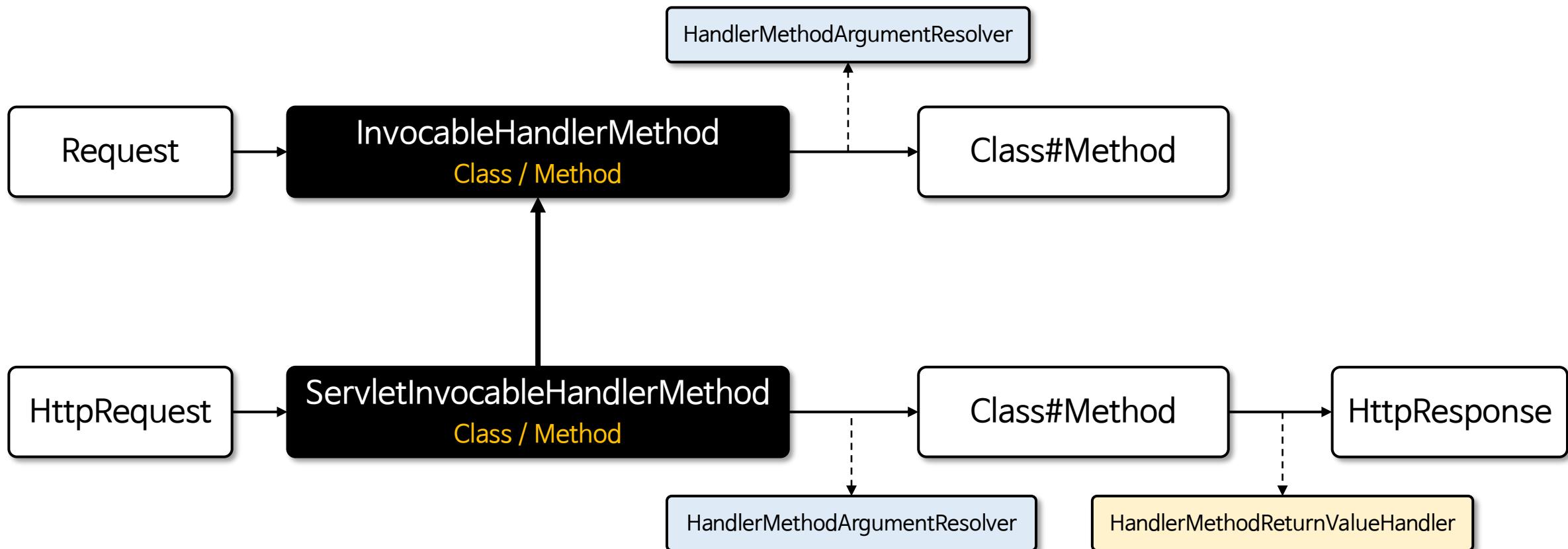
```
public class Main {  
    public static void main(String[] args) throws Exception {  
  
        Class<?> userClass = Class.forName("io.springmvc.springmvcmaster.User");  
        Constructor<?> userConstructor = userClass.getDeclaredConstructor(String.class, int.class);  
        Object user = userConstructor.newInstance("Alice", 25); // 동적 User 객체 생성  
  
        ReflectionExecutor executor = new ReflectionExecutor(userClass);  
  
        // 필드 관리  
        ReflectionFieldManager fieldManager = executor.getFieldManager();  
        fieldManager.setFieldValue(user, "name", "Bob");  
  
        // 메서드 관리  
        ReflectionMethodManager methodManager = executor.getMethodManager();  
        methodManager.invokeMethod(user, "greet");  
  
        // 결과 출력  
        Object updatedName = fieldManager.getFieldValue(user, "name");  
        System.out.println("Updated User Name: " + updatedName);  
    }  
}
```

핸들러 메서드 호출 원리

<https://github.com/onjsdnjs/spring-mvc-master/tree/핸들러-메서드-호출원리>

✓ 개요

- 스프링의 핸들러 메서드를 호출하기 위해 내부적으로 사용되는 클래스로서 InvocableHandlerMethod 와 이를 확장한 ServletInvocableHandlerMethod 클래스가 있다
- 메서드 호출 시 메서드 파라미터를 해석하기 위한 HandlerMethodArgumentResolver 와 메서드 반환 값을 처리하기 위한 HandlerMethodReturnValueHandler 와 같은 클래스와 연계하여 요청을 처리하고 있다
- InvocableHandlerMethod 는 범용적 메서드 호출 클래스이며 ServletInvocableHandlerMethod HTTP 요청/응답 처리 클래스로 구분할 수 있다



✓ InvocableHandlerMethod

- HTTP 프로토콜에 독립적이며 범용적인 메서드 호출 기능을 제공하고 매개변수 처리를 위해 HandlerMethodArgumentResolver 를 사용한다
- 메서드를 호출한 결과를 반환하지만 내부적으로 반환값을 추가로 처리하지는 않는다

✓ 주요 API 및 사용방법

```
public InvocableHandlerMethod(HandlerMethod handlerMethod) { // 호출 대상 정보가 들어 있는 HandlerMethod 객체  
    super(handlerMethod);  
}  
  
public InvocableHandlerMethod(Object bean, Method method) { // 호출 대상 객체, 호출 대상 메서드  
    super(bean, method);  
}
```

생성자

호출 API

```
Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndView mavContainer, Object... providedArgs)
```

사용 방법

```
// InvocableHandlerMethod를 생성  
InvocableHandlerMethod handlerMethod = new InvocableHandlerMethod(myService, serviceMethod);  
  
// Service 메서드 호출  
Object result = handlerMethod.invokeForRequest(webRequest, mavContainer);
```

✓ 구현 예제

```
@Controller  
@RequiredArgsConstructor  
public class MyController {  
  
    private final MyService myService;  
  
    @GetMapping("/invokeService")  
    public void invokeService(HttpServletRequest request, HttpServletResponse response) throws Exception {  
  
        Method serviceMethod = myService.getClass().getMethod("processRequest", User.class); // Service의 메서드 정보를 가져옴  
        InvocableHandlerMethod handlerMethod = new InvocableHandlerMethod(myService, serviceMethod); // InvocableHandlerMethod를 생성  
  
        HandlerMethodArgumentResolverComposite resolver = new HandlerMethodArgumentResolverComposite(); // 서비스 메서드 파라미터 해석을 위해 ArgumentResolver 설정  
        resolver.addResolvers(new MyArgumentResolver());  
        handlerMethod.setHandlerMethodArgumentResolvers(resolver);  
  
        ServletWebRequest webRequest = new ServletWebRequest(request, response); // 요청 처리 객체 생성  
        ModelAndView mavContainer = new ModelAndView(); // 모델 데이터 저장을 위해 필요  
  
        Object result = handlerMethod.invokeForRequest(webRequest, mavContainer, null); // Service 메서드 호출  
        response.getWriter().write(result.toString()); // 결과를 응답으로 출력  
    }  
}
```

✓ 구현 예제

```
@Service
public class MyService {
    public String processRequest(User user) {
        return "Hello from MyService, " + user.getUsername() + " : " + user.getEmail();
    }
}

public class MyArgumentResolver implements HandlerMethodArgumentResolver {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        return parameter.getParameterType().equals(User.class);
    }

    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,
                                  NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {
        User user = new User();
        user.setUsername(webRequest.getParameter("username"));
        String emailParam = webRequest.getParameter("email");
        user.setEmail(emailParam != null ? Integer.parseInt(emailParam) : 0);
        return user;
    }
}
```

```
public class User {
    private String name;
    private String email;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

✓ ServletInvocableHandlerMethod

- InvocableHandlerMethod 를 상속한 클래스로서 Servlet 환경에 특화되어 있으며(HttpServletRequest/HttpServletResponse) HTTP 요청 및 응답을 포함한 핸들러 메서드를 호출 한다
- 메서드의 반환값을 HandlerMethodReturnValueHandler 를 통해 처리할 수 있으며 메서드 레벨의 @ResponseStatus 어노테이션을 지원하여 HTTP 응답 상태를 설정할 수 있다

✓ 주요 API 및 사용방법

```
public ServletInvocableHandlerMethod (HandlerMethod handlerMethod) { // 호출 대상 정보가 들어 있는 HandlerMethod 객체  
    super(handlerMethod);  
}  
  
public ServletInvocableHandlerMethod (Object handler, Method method) { // 호출 대상 객체, 호출 대상 메서드  
    super(bean, method);  
}
```

생성자

호출 API

```
Object invokeAndHandle (NativeWebRequest request, @Nullable ModelAndView mavContainer, Object... providedArgs)
```

사용 방법

```
// ServletInvocableHandlerMethod 를 생성  
ServletInvocableHandlerMethod handlerMethod = new ServletInvocableHandlerMethod(myService, serviceMethod);  
  
// Service 메서드 호출  
Object result = handlerMethod.invokeAndHandle(webRequest, mavContainer);
```

✓ 구현 예제

```
@Controller
@ControllerArgsConstructor
public class MyController {
    private final MyService myService;
    @GetMapping("/invokeService2")
    public void saveUser2(HttpServletRequest request, HttpServletResponse response) throws Exception {
        Method serviceMethod = myService.getClass().getMethod("processRequest", User.class); // Service의 메서드 정보를 가져옴
        ServletInvocableHandlerMethod handlerMethod = new ServletInvocableHandlerMethod (myService, serviceMethod); // InvocableHandlerMethod를 생성
        HandlerMethodArgumentResolverComposite resolver = new HandlerMethodArgumentResolverComposite(); // 서비스 메서드 파라미터 해석을 위해 ArgumentResolver 설정
        resolver.addResolver(new MyArgumentResolver());
        HandlerMethodReturnValueHandlerComposite returnValueHandler = new HandlerMethodReturnValueHandlerComposite(); // 서비스 메서드 반환값 처리를 위해 설정
        returnValueHandler.addHandler(new MyReturnValueHandler());
        handlerMethod.setHandlerMethodArgumentResolvers(resolver);
        handlerMethod.setHandlerMethodReturnValueHandlers(returnValueHandler);
        ServletWebRequest webRequest = new ServletWebRequest(request, response); // 요청 처리 객체 생성
        ModelAndViewContainer mavContainer = new ModelAndViewContainer(); // 모델 데이터 저장을 위해 필요
        handlerMethod.invokeAndHandle(webRequest, mavContainer); // Service 메서드 호출
        if (mavContainer.containsAttribute("result")) {
            response.getWriter().write(mavContainer.getModel().get("result").toString()); // ModelAndViewContainer에 저장된 값으로 응답 작성
        }
    }
}
```

✓ 구현 예제

```
@Service
public class MyService {
    public String processRequest(User user) {
        return "Hello from MyService, " + user.getUsername() + ":" + user.getEmail();
    }
}
```

```
public class MyReturnValueHandler implements HandlerMethodReturnValueHandler {

    @Override
    public boolean supportsReturnType(MethodParameter returnType) {
        return returnType.getParameterType().equals(String.class);
    }

    @Override
    public void handleReturnValue(Object returnValue, MethodParameter returnType,
            ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {
        if (returnValue != null) {
            mavContainer.addAttribute("result", returnValue);
        }
    }
}
```

메서드 파라미터 실행 구조 이해

<https://github.com/onjsdnjs/spring-mvc-master/tree/메서드-파라미터-실행-구조-이해>

✓ 개요

- RequestMappingHandlerAdapter 클래스와 연관 클래스들의 실행 구조를 분석함으로서 스프링 MVC의 내부 흐름을 더 깊이 이해하고 개발적 관점에서의 인사이트를 넓혀 갈수 있다

✓ 핵심 클래스 이해

ServletInvocableHandlerMethod

HTTP 요청 데이터를 받아 메서드를 실행시키는 클래스로서 HandlerMethodArgumentResolver를 통해 요청 데이터를 메서드의 매개변수에 맞게 변환하고 이를 사용해 메서드를 호출한다

AnnotatedMethod

호출 메서드 및 메서드 파라미터 정보를 가지고 있으며 파라미터에 어노테이션 정보를 탐색할 수 있다

MethodParameter

메서드에 있는 파라미터의 정보를 관리하는 클래스로서 메서드의 어노테이션을 분석하고 파라미터가 어디에 있고 어떤 타입인지를 편리하게 알 수 있게 해주는 도구이다

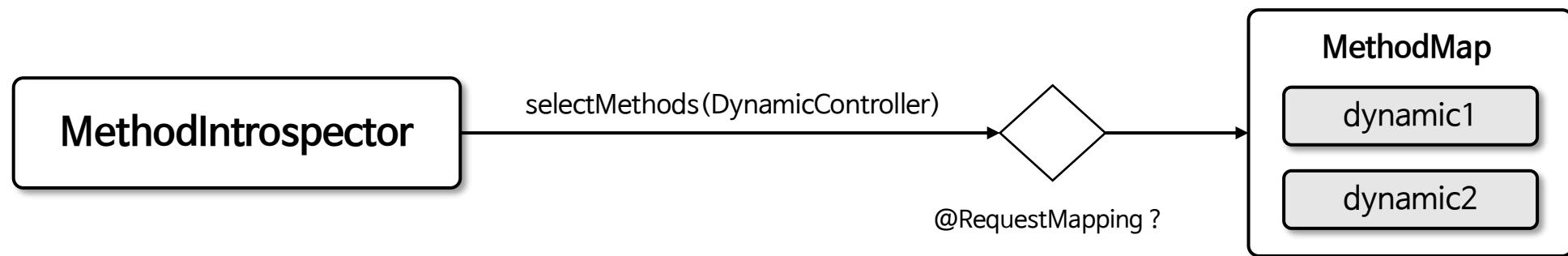
MethodIntrospector

메서드를 탐색할 수 있게 해주는 클래스로서 특정 어노테이션이 적용된 메서드들의 리스트를 구하거나 메서드의 메타데이터를 찾도록 돕는 도구이다

✓ MethodIntrospector 구조

- 스프링 부트가 구동이 되면 초기화 시점에 빈 클래스들을 스캔하면서 @RequestMapping 이 선언된 Method 객체들을 생성한다

```
@Controller  
public class DynamicController {  
    @GetMapping("/dynamic1")  
    public AccountDto dynamic1(@RequestParam("name") String name, @ModelAttribute AccountDto accountDto, Model model){  
        return accountDto;  
    }  
  
    @GetMapping("/dynamic2")  
    public String dynamic2(HttpServletRequest request, HttpServletResponse response, Model model){  
        return "leaven";  
    }  
}
```



✓ AnnotatedMethod & MethodParamter 구조

- 스캔한 메서드들의 파라미터 메타 정보들을 가지는 MethodParameter 객체를 생성한다

```
@Controller  
public class DynamicController {  
    @GetMapping("/dynamic1")  
    public AccountDto dynamic1(@RequestParam("name") String name, @ModelAttribute AccountDto accountDto,  
                               Model model){  
        return accountDto;  
    }  
  
    @GetMapping("/dynamic2")  
    public String dynamic2(HttpServletRequest request, HttpServletResponse response){  
        return "leaven";  
    }  
}
```

AnnotatedMethod

dynamic1

MethodParameters

MethodParameter 0

MethodParameter 1

MethodParameter 2

AnnotatedMethod

dynamic2

MethodParameters

MethodParameter 0

MethodParameter 1

```
∞ this.parameters = {MethodParameter[3]@7123}
```

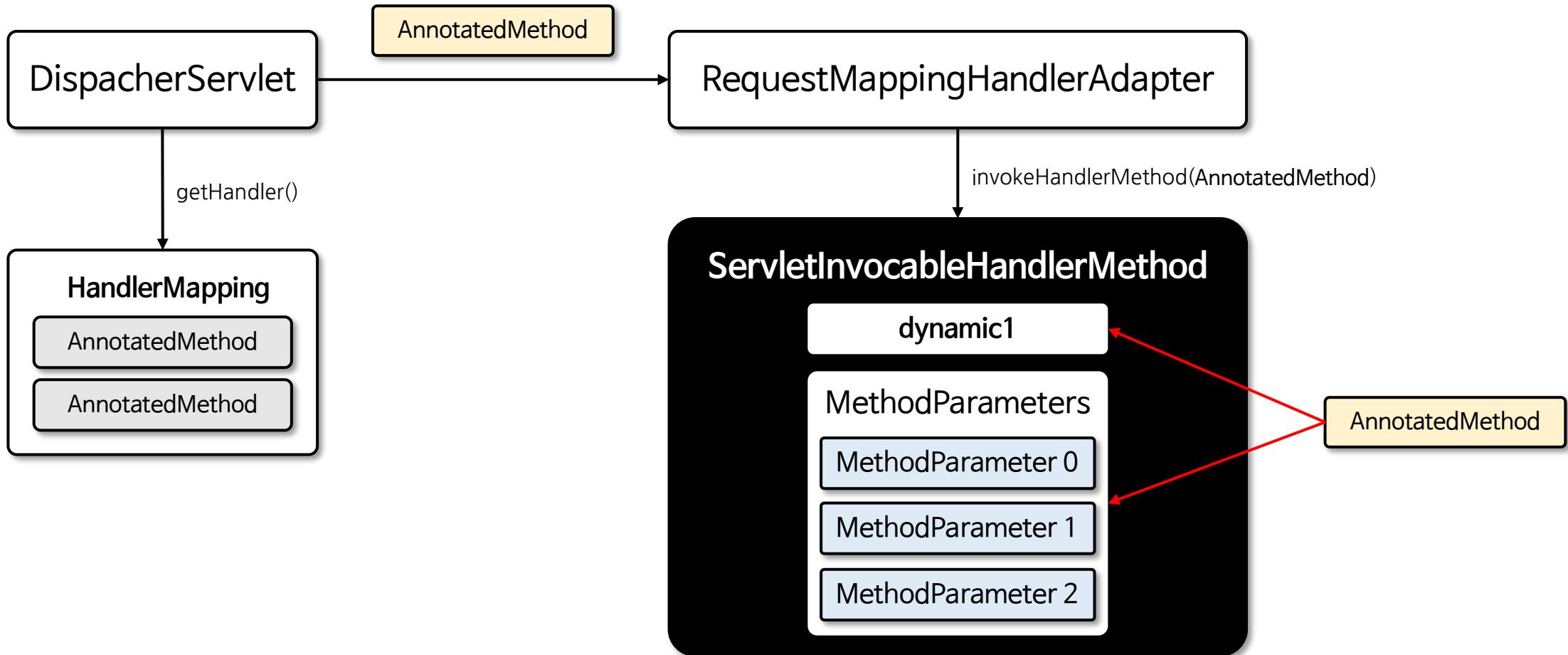
```
> └─ 0 = {AnnotatedMethod$AnnotatedMethodParameter@7139} "method 'dynamic1' parameter 0"  
> └─ 1 = {AnnotatedMethod$AnnotatedMethodParameter@7140} "method 'dynamic1' parameter 1"  
> └─ 2 = {AnnotatedMethod$AnnotatedMethodParameter@7141} "method 'dynamic1' parameter 2"
```

```
∞ this.parameters = {MethodParameter[2]@7121}
```

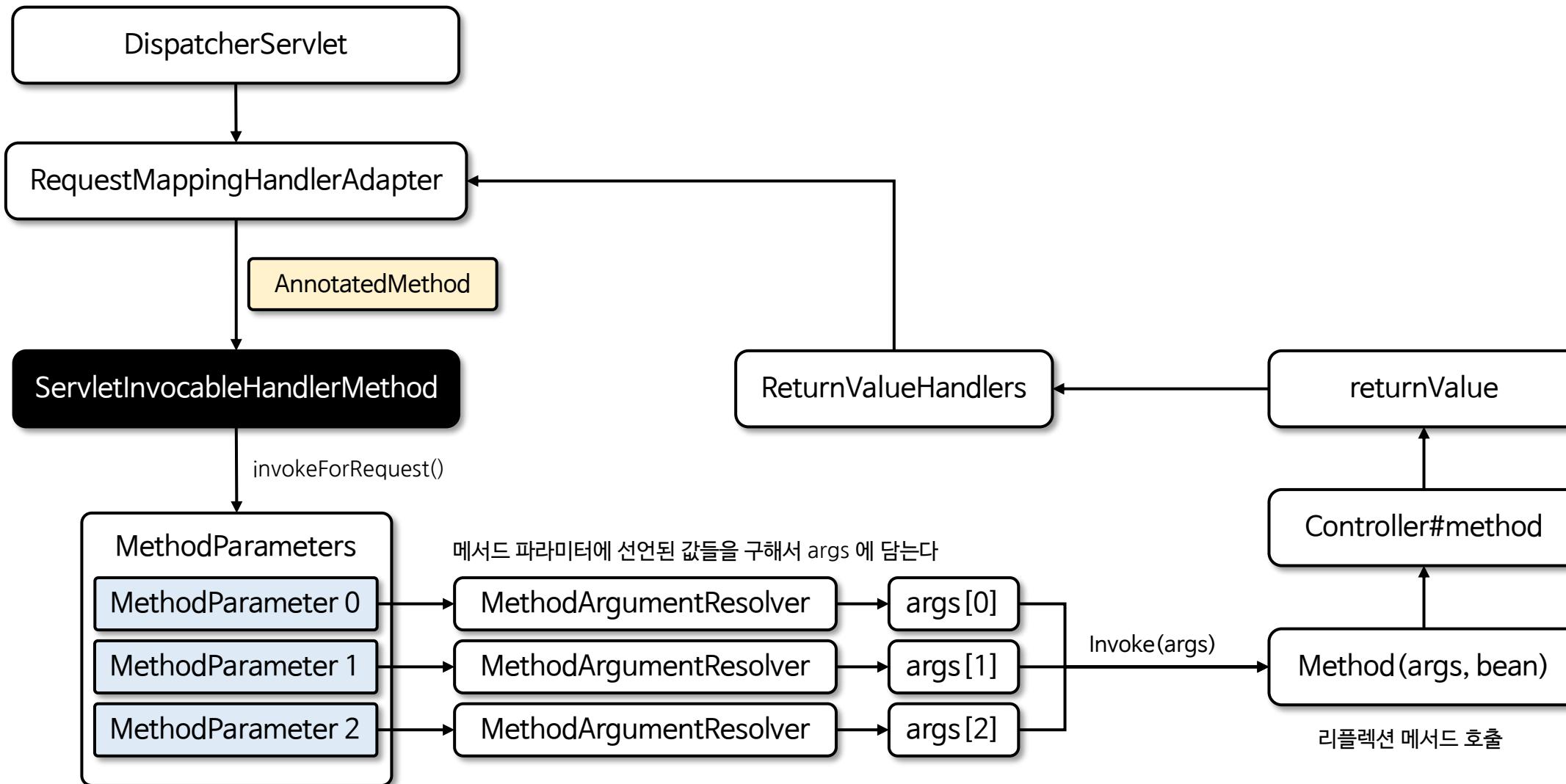
```
> └─ 0 = {AnnotatedMethod$AnnotatedMethodParameter@7125} "method 'dynamic2' parameter 0"  
> └─ 1 = {AnnotatedMethod$AnnotatedMethodParameter@7126} "method 'dynamic2' parameter 1"
```

✓ ServletInvocableHandlerMethod 구조

- InvocableHandlerMethod 클래스를 확장한 웹처리용 클래스로서 핸들러의 메서드를 호출하는 기능을 가지고 있다
- HTTP 요청 정보를 분석하고 해당 요청에 매핑되어 있는 AnnotatedMethod 객체를 찾아 새로운 ServletInvocableHandlerMethod 를 생성하고 전달한다
- AnnotatedMethod 에 저장되어 있는 Method, MethodParameter 정보들을 ServletInvocableHandlerMethod 에 복사한다



✓ 메서드 파라미터 실행 흐름도



메서드 파라미터 커스텀 구현

<https://github.com/onjsdnjs/spring-mvc-master/tree/메서드-파라미터-커스텀-구현>

✓ 개요

- 스프링 내부 클래스를 활용 및 확장하여 HTTP 요청에 대한 응답 과정을 커스텀하게 구현할 수 있다

✓ 주요 클래스 이해

DispatcherController

클라이언트의 요청을 받아 동적으로 서비스의 메서드를 호출할 수 있도록 한다

MethodExtractor

서비스 클래스의 커스텀 어노테이션이 적용된 메서드들의 리스트를 추출한다

DynamicMethodInvoker

InvocableHandlerMethod 를 사용하여 커스텀 어노테이션이 적용된 메서드를 호출한다

CustomHandlerMethodArgumentResolver

메서드 호출에 필요한 파라미터 값을 구한다.

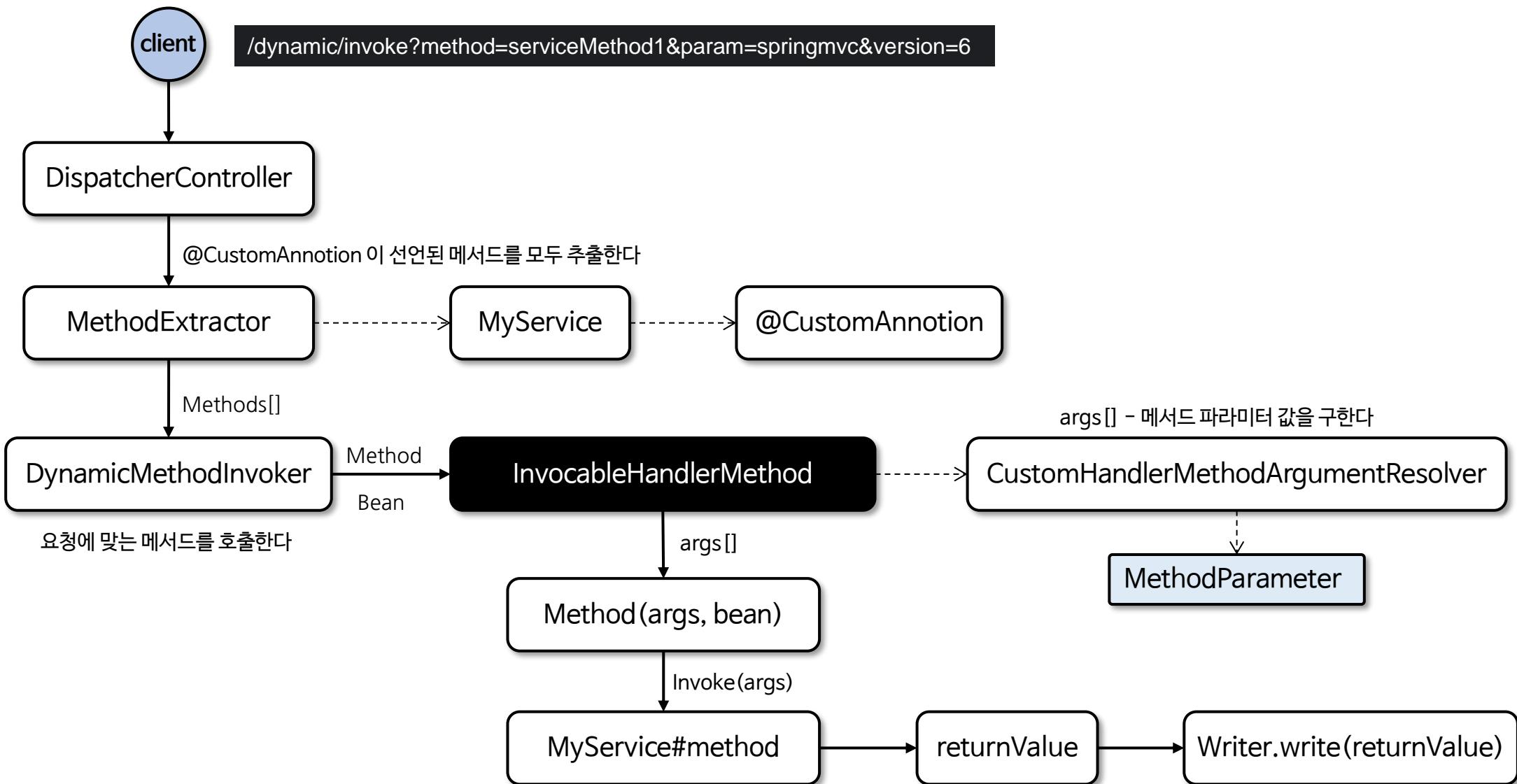
@CustomAnnotation

특정 메서드에 선언하기 위한 커스텀 어노테이션

MyService

커스텀 어노테이션이 선언된 메서드를 정의하며 DynamicMethodInvoker 에 의해 호출된다

✓ 구현 흐름





스프링 웹 MVC 완전 정복

✓ 스프링 웹 MVC 공통 기능

1. 인터셉터(Interceptor)
2. @ControllAdvice

인터셉터(Interceptor)

<https://github.com/onjsdnjs/spring-mvc-master/tree/Interceptor>

✓ 개요

- 인터셉터(Interceptor)는 핸들러의 실행 전후 또는 뷰 렌더링 이후 특정 로직을 실행할 수 있으며 HandlerInterceptor 인터페이스를 구현하여 사용할 수 있다
- 주로 여러 컨트롤러에서 공통으로 사용하는 기능을 구현하거나 재사용성을 높이고자 할 때 사용한다 (인증, 인가, 로깅, 통계집계 등..)

✓ 구조

HandlerInterceptor

boolean **preHandle** (HttpServletRequest request, HttpServletResponse response, Object handler)

void **postHandle** (HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView)

void **afterCompletion** (HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception ex)

- **preHandle**

- 컨트롤러 실행 전에 호출되며 호출 할 Handler 객체가 인자로 전달된다
- Boolean 반환값으로 True를 반환하면 다음 단계로 진행하고 false를 반환하면 요청 처리를 즉시 중단한다

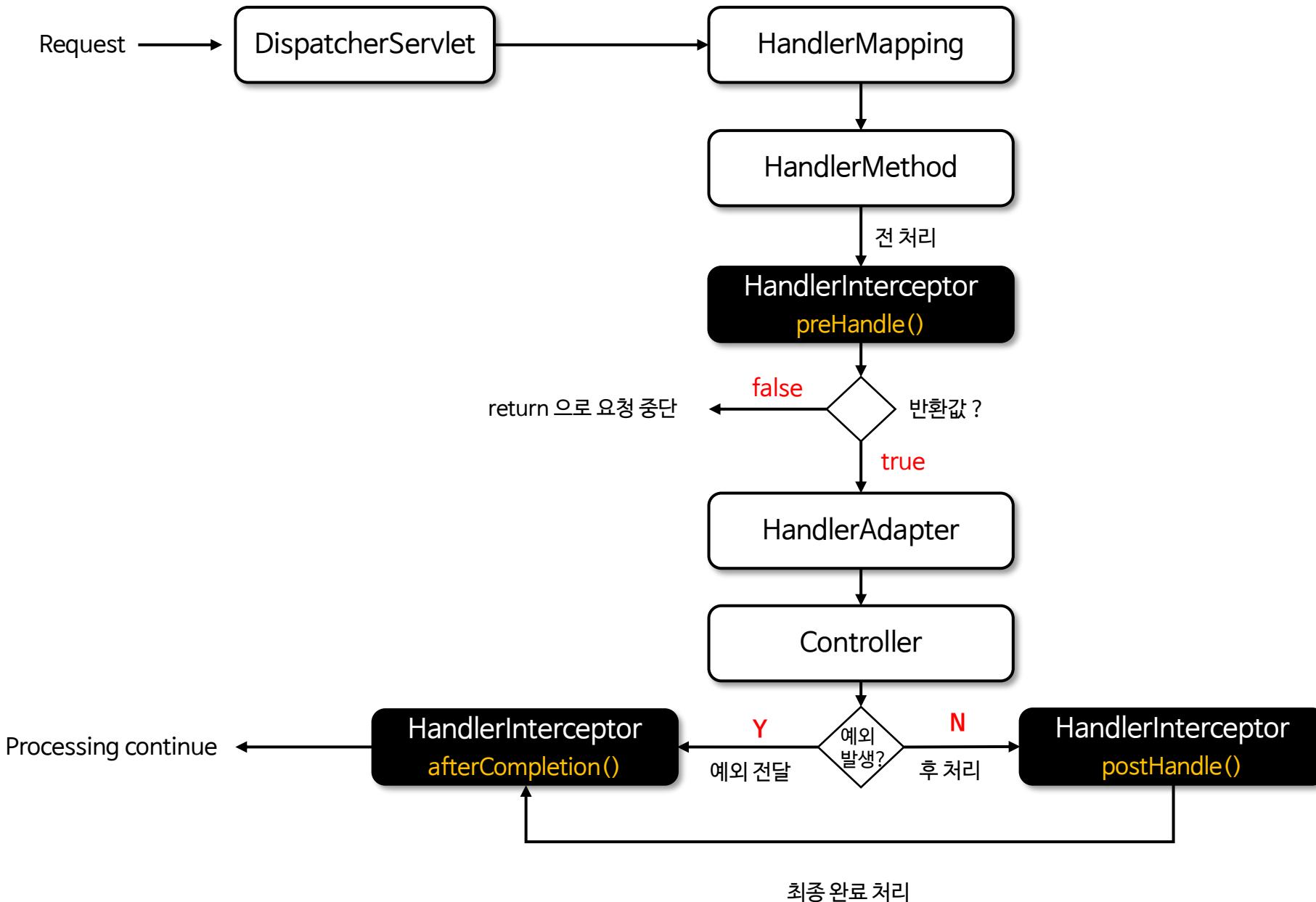
- **postHandle**

- 컨트롤러 실행 후 뷰 렌더링 전에 호출되며 호출된 Handler 및 ModelAndView 객체가 인자로 전달된다

- **afterCompletion**

- 뷰 렌더링이 완료된 후 호출되며 호출된 Handler 및 예외 발생 시 예외타입이 인자로 전달된다
- afterCompletion은 예외가 발생해도 무조건 호출되므로 반드시 해야 할 공통 작업이 있다면 여기서 수행하도록 한다

✓ 흐름도



✓ 인터셉터 사용

- HandlerInterceptor 인터페이스 또는 HandlerInterceptorAdapter 클래스를 상속하여 구현한다
- WebMvcConfigurer 를 사용하여 인터셉터를 등록한다
 - 특정 URL 패턴에만 인터셉터를 적용하거나 제외 할 수 있다
 - order 속성을 통해 인터셉터의 호출 순서를 지정할 수 있다

✓ 인터셉터 기본 구현

```
@Component
public class CustomInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        System.out.println("PreHandle: Request URI is " + request.getRequestURI());
        return true; // true를 반환하면 요청 처리가 계속 진행된다
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("PostHandle: Handler executed");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        System.out.println("AfterCompletion: Request completed");
    }
}
```

✓ 인터셉터 등록

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Autowired
    private CustomInterceptor1 customInterceptor1;
    @Autowired
    private CustomInterceptor2 customInterceptor2;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(customInterceptor1)
            .order(1) // 숫자가 낮을수록 우선 호출된다
            .addPathPatterns("/api/**") // 인터셉터를 "/api/**" 패턴에만 적용
            .excludePathPatterns("/api/public/**"); // "/api/public/**" 경로는 제외

        registry.addInterceptor(customInterceptor2)
            .order(2)
            .addPathPatterns("/log/**")
            .excludePathPatterns("/api/public/**");
    }
}
```

✓ 인터셉터 활용 - LoggingInterceptor

- 인터셉터에서 각 메서드마다 데이터를 저장하고 불러오기 위해서는 요청 객체(HttpServletRequest)의 속성을 이용하거나 ThreadLocal을 사용해서 구현할 수 있다

```
@Component
public class LoggingInterceptor implements HandlerInterceptor {

    private static final Logger logger = LoggerFactory.getLogger(LoggingInterceptor.class);
    private static final ThreadLocal<String> requestIdHolder = new ThreadLocal<>(); // ThreadLocal을 사용하여 요청 아이디 저장

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        String requestId = UUID.randomUUID().toString(); // 요청 아이디 생성 (UUID 사용)
        requestIdHolder.set(requestId); // 요청 아이디 저장, request.setAttribute(requestId, requestId);
        logger.info("Request ID {} - PreHandle: Starting request {} {}", requestId, request.getMethod(), request.getRequestURI());
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        String requestId = requestIdHolder.get(); // ThreadLocal에서 요청 아이디 가져오기, request.getAttribute(requestId)
        logger.info("Request ID {} - PostHandle: Handler executed", requestId);
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        String requestId = requestIdHolder.get(); // ThreadLocal에서 요청 아이디 가져오기
        if (ex != null) logger.error("Request ID {} - AfterCompletion: Exception occurred - {}", requestId, ex.getMessage(), ex);
        else logger.info("Request ID {} - AfterCompletion: Request completed successfully", requestId);

        requestIdHolder.remove(); // ThreadLocal 비우기
    }
}
```

✓ 인터셉터 활용 - preHandle(.., HandlerMethod)

- RequestMappingHandlerMapping 클래스로부터 생성된 HandlerMethod 객체는 인터셉터의 preHandle 메서드로 전달된다

```
@Component
public class CustomInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        if (handler instanceof HandlerMethod) { // handler가 HandlerMethod인지 확인
            HandlerMethod handlerMethod = (HandlerMethod) handler;
            String controllerName = handlerMethod.getBeanType().getName(); // 컨트롤러 클래스 이름 가져오기
            String methodName = handlerMethod.getMethod().getName(); // 컨트롤러 메서드 이름 가져오기
            AdminOnly adminOnly = handlerMethod.getMethodAnnotation(AdminOnly.class); // 특정 어노테이션 확인 (예: @AdminOnly)

            if (adminOnly != null) {
                String userRole = request.getHeader("Role");
                if (userRole == null || !userRole.equals("ADMIN")) {
                    response.setStatus(HttpStatus.SC_FORBIDDEN);
                    response.getWriter().write("Access Denied: Admins only.");
                    return false; // 요청 차단
                }
            }
            System.out.println("Controller: " + controllerName + ", Method: " + methodName);
        }
        return true; // 요청 계속 진행
    }
}
```

```
handlerMethod = {HandlerMethod@8049}
>   ↗ bean = {ExampleController@6706}
>   ↗ beanFactory = {DefaultListableBeanFactory@6706}
>   ↗ messageSource = {AnnotationConfigServletWebServerApplicationContext$2@6706}
>   ↗ beanType = {Class@5759} "class io.mvc.ExampleController"
>   ↗ validateArguments = false
>   ↗ validateReturnValue = false
>   ↗ responseStatus = null
>   ↗ responseStatusReason = null
>   ↗ resolvedFromHandlerMethod = {HandlerMethod@8049}
>   ↗ description = "io.mvc.simplemvc.interceptor.CustomInterceptor"
>   ↗ method = {Method@6710} "public java.lang.Object io.mvc.simplemvc.interceptor.CustomInterceptor.preHandle(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, java.lang.Object)"
>   ↗ bridgedMethod = {Method@6710} "public java.lang.Object io.mvc.ExampleController.preHandle(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, java.lang.Object)"
>   ↗ parameters = {MethodParameter[0]@6711} "javax.servlet.http.HttpServletRequest"
>   ↗ inheritedParameterAnnotations = null
```

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface AdminOnly { }
```

✓ 인터셉터 활용 - postHandle(.., ModelAndView)

- RequestMappingHandlerAdapter 클래스 실행 후 반환된 ModelAndView는 인터셉터의 postHandle 메서드로 전달된다

```
@Component
public class CustomInterceptor implements HandlerInterceptor {

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
                           @Nullable ModelAndView modelAndView) throws Exception {
        if (modelAndView != null) {
            String viewName = modelAndView.getViewName(); // 뷰 이름 확인

            if ("userProfile".equals(viewName)) {
                Map<String, Object> model = modelAndView.getModel(); // 컨트롤러에서 넘긴 데이터 확인

                if (model.containsKey("username")) {
                    String username = (String) model.get("username");
                    String role;
                    if ("admin".equalsIgnoreCase(username)) { // 사용자 이름에 따라 권한 추가
                        role = "Administrator";
                    } else if ("guest".equalsIgnoreCase(username)) {
                        role = "Guest";
                    } else {
                        role = "User";
                    }
                    modelAndView.addObject("role", role); // 모델에 권한 정보 추가, 뷰 화면에서 참조 가능
                }
            }
        }
    }
}
```

```
(P) modelAndView = {ModelAndView@6737}
  > (f) view = "userProfile"
  < (f) model = {ModelMap@8038} size = 1
    > [ "username" -> "admin" ]
      (f) status = null
      (f) cleared = false
```

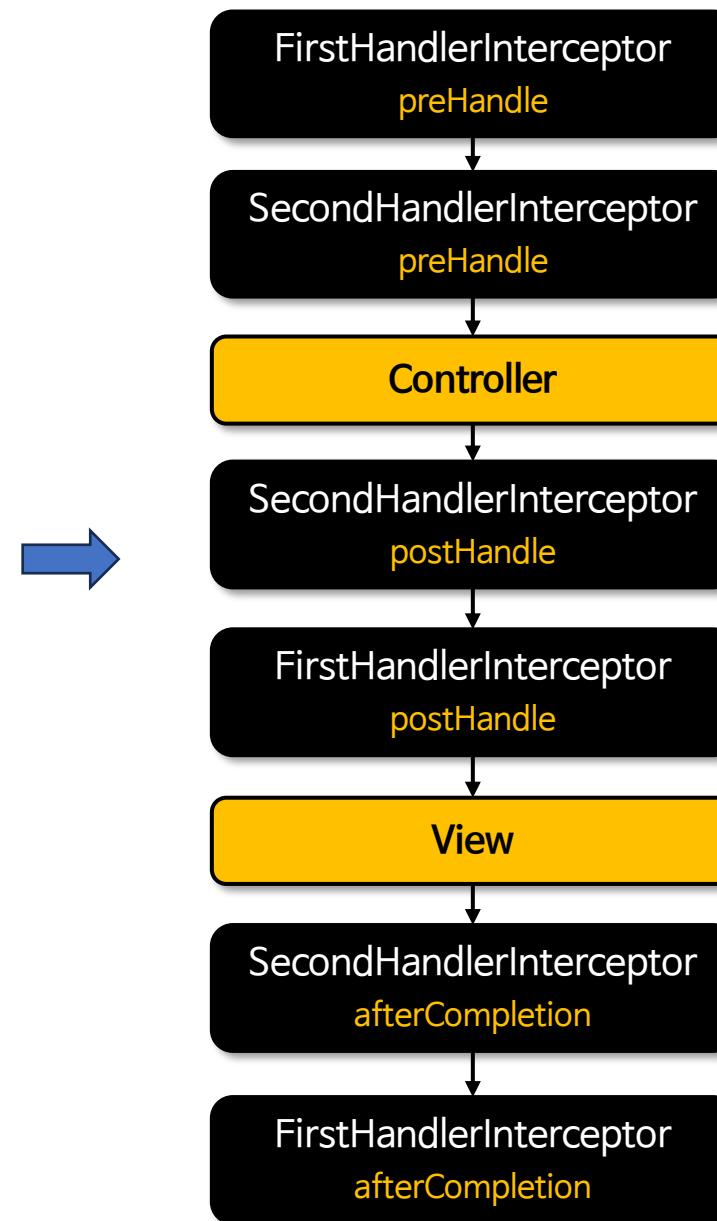
✓ 다중 인터셉터 실행 호출 순서

- 여러 개의 인터셉터를 등록할 경우 등록된 순서대로 preHandle이 호출되고 등록된 역순으로 postHandle 및 afterCompletion이 호출된다

```
@Configuration  
public class WebConfig implements WebMvcConfigurer {  
  
    @Autowired  
    private FirstInterceptor firstInterceptor;  
  
    @Autowired  
    private SecondInterceptor secondInterceptor;  
  
    @Override  
    public void addInterceptors(InterceptorRegistry registry) {  
        registry.addInterceptor(firstInterceptor).addPathPatterns("/**").order(1); // 먼저 등록  
        registry.addInterceptor(secondInterceptor).addPathPatterns("/**").order(2); // 나중에 등록  
    }  
}
```

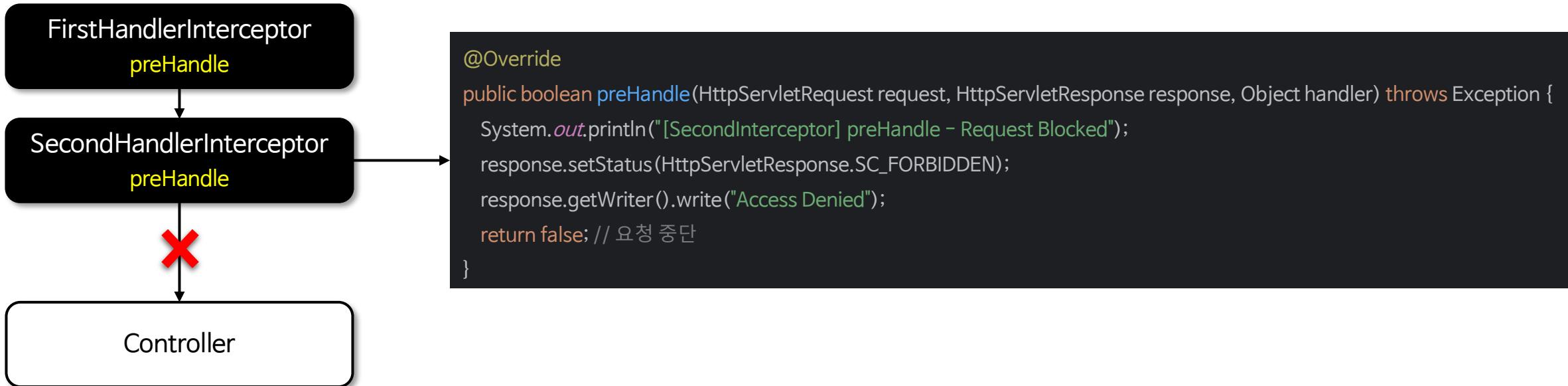
출력 결과

- 1) FirstInterceptor - preHandle
- 2) SecondInterceptor - preHandle
- 3) SecondInterceptor - postHandle
- 4) FirstInterceptor - postHandle
- 5) SecondInterceptor - afterCompletion
- 6) FirstInterceptor - afterCompletion



✓ 다중 인터셉터 흐름 차단

- 다중 인터셉터 실행 중 이전의 인터셉터 중 하나라도 preHandle에서 false를 반환하면 이후의 인터셉터와 핸들러는 실행되지 않고 요청 처리를 차단하며 사용자에게 적절한 응답을 반환할 수 있다

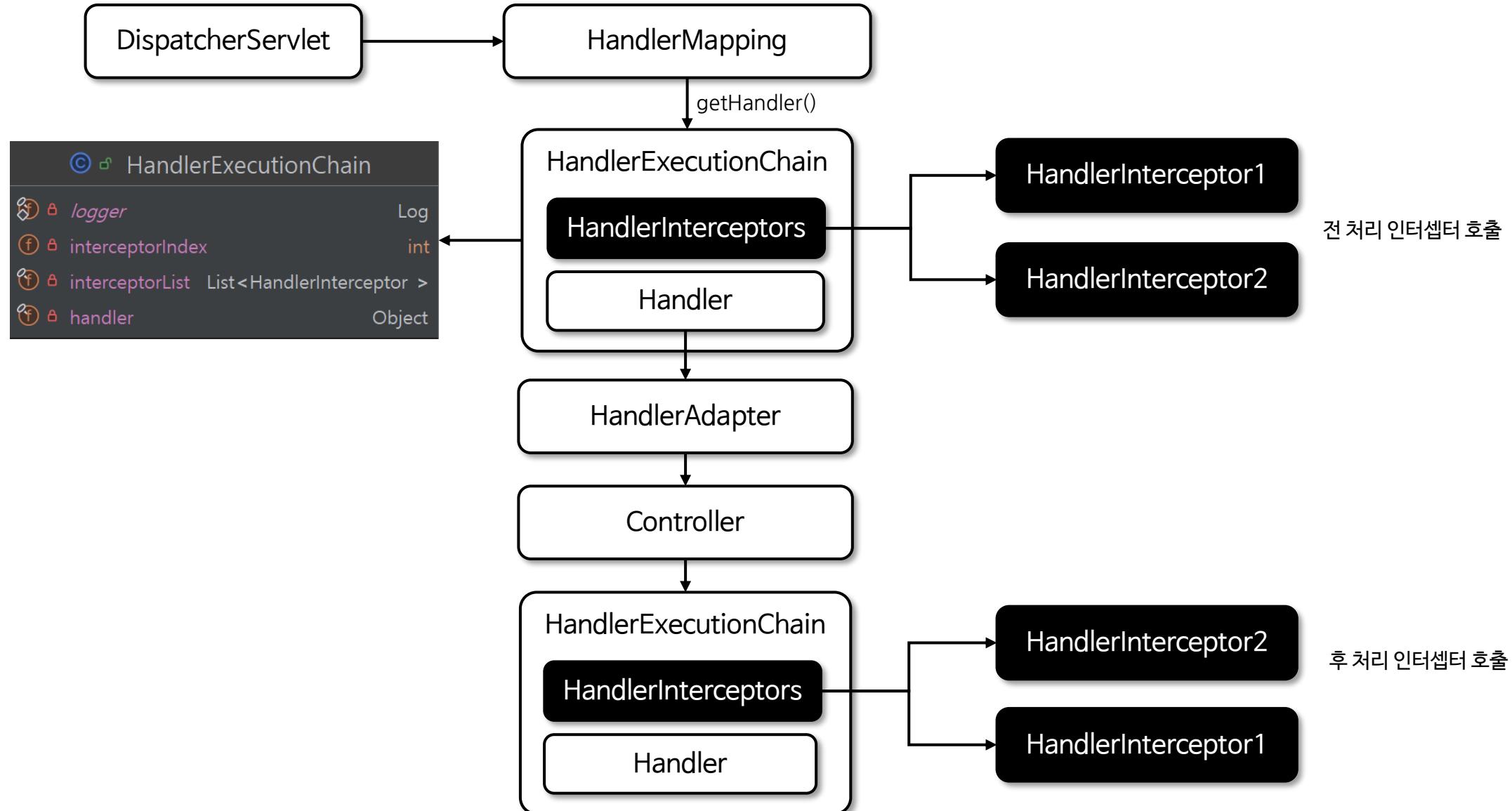


✓ 체크 사항

- 인터셉터는 보안 계층으로 사용하기에는 적합하지 않을 수 있다. 왜냐하면 어노테이션 기반 컨트롤러 경로 매핑과의 불일치 가능성이 있기 때문에 보안 작업은 Spring Security 또는 서블릿 필터를 사용하는 것이 권장된다

✓ 인터셉터 체인

- 인터셉터는 HandlerExecutionChain 객체의 속성으로 한개 이상 저장되며 각 인터셉터는 체인형태로 호출되어 실행된다



@ControllerAdvice

<https://github.com/onjsdnjs/spring-mvc-master/tree/@ControllerAdvice>

✓ 개요

- `@ControllerAdvice`는 특정 컨트롤러 또는 전체 컨트롤러에서 발생하는 예외를 전역적으로 처리하거나 컨트롤러와 관련된 공통적인 기능을 구현하는데 사용된다
- `@ControllerAdvice`는 클래스 레벨에 선언하며 메서드 레벨에 선언하는 어노테이션(`@ExceptionHandler`, `@ModelAttribute`, `@InitBinder`)과 함께 사용할 수 있다

✓ 주요 용도

용도	설명	주요 애노테이션	적용 순서
예외 처리	컨트롤러에서 발생한 예외를 전역적으로 처리할 수 있으며 적절한 응답을 생성할 수 있다	<code>@ExceptionHandler</code>	메서드 실행 후에 적용
모델 속성 관리	컨트롤러의 모든 요청에 공통적으로 필요한 데이터를 추가할 수 있다	<code>@ModelAttribute</code>	메서드 실행 전에 적용
데이터 바인딩	요청 파라미터를 특정 형식으로 변환하거나 검증 로직을 적용할 수 있다	<code>@InitBinder</code>	

✓ 선언 방법 & 구조

`@ControllerAdvice`

```
public class SomeClass { }
```

<code>@ControllerAdvice</code>	속성	설명	예제
<code>name()</code>	<code>String annotations</code>	특정 애노테이션이 적용된 컨트롤러만 선택.	<code>@ControllerAdvice(annotations = RestController.class)</code>
<code>basePackages()</code>	<code>String[] basePackages</code>	특정 패키지 내의 컨트롤러만 선택.	<code>@ControllerAdvice(basePackages = "com.example.controllers")</code>
<code>value()</code>	<code>String[] value()</code>	특정 클래스가 속한 패키지의 컨트롤러만 선택.	<code>@ControllerAdvice(value = {SomeController.class})</code>
<code>basePackageClasses()</code>	<code>Class<?>[] basePackageClasses</code>	특정 클래스가 속한 패키지의 컨트롤러만 선택.	<code>@ControllerAdvice(basePackageClasses = {SomeController.class})</code>
<code>assignableTypes()</code>	<code>Class<?>[] assignableTypes</code>	특정 클래스 또는 클래스 계층 구조에 속하는 컨트롤러만 선택	<code>@ControllerAdvice(assignableTypes = {AbstractController.class})</code>
<code>annotations()</code>	<code>Class<Annotation>[] annotations</code>		

✓ @ControllerAdvice + @ModelAttribute

```
@ControllerAdvice  
public class GlobalModelAttributes {  
  
    @ModelAttribute("appName") // 모든 뷰에서 전역적으로 모델 데이터를 참조할 수 있다  
    public String addApplicationName() {  
        return "My Application";  
    }  
}
```

```
@Controller  
public class ViewController {  
  
    @GetMapping("/view")  
    public String showView(Model model) { // model에는 appName=My Application이 담겨져 있다  
        model.addAttribute("pageTitle", "Welcome Page");  
        return "welcome";  
    }  
}
```

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
    <title th:text="${pageTitle}">Page Title</title> // Welcome Page  
</head>  
<body>  
    <h1 th:text="${appName}"></h1> // My Application  
</body>  
</html>
```

✓ @ControllerAdvice + @InitBinder

```
@ControllerAdvice  
public class GlobalDataBinder {  
  
    @InitBinder  
    public void initBinder(WebDataBinder binder) {  
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));  
    }  
}
```

```
@RestController  
public class DateController {  
  
    @GetMapping("/test/date")  
    public String testDate(@RequestParam Date date) {  
        return "date: " + date.toString();  
    }  
}
```

요청: /test/date?date=2024-11-01

결과:

HTTP 200 OK
Body: date: Fri Nov 01 00:00:00 UTC 2024

요청: /test/date?date=01-11-2024

HTTP 400 Bad Request
Body: Invalid date format. Please use 'yyyy-MM-dd'.

✓ @ControllerAdvice & @RestControllerAdvice

기준	@ControllerAdvice	@RestControllerAdvice
적용 대상	@Controller	@RestController
기본 응답 형식	View 이름 반환 또는 HTML 렌더링	JSON 또는 XML 응답
내부 메타 어노테이션	@Component	@ControllerAdvice + @ResponseBody

@ControllerAdvice

```
@ControllerAdvice  
public class GlobalExceptionHandler {  
  
    @ExceptionHandler(IllegalArgumentException.class)  
    public ModelAndView handleIllegalArgumentException(IllegalArgumentException ex) {  
        ModelAndView mav = new ModelAndView("error");  
        mav.addObject("message", ex.getMessage());  
        return mav;  
    }  
}
```

@RestControllerAdvice

```
@RestControllerAdvice  
public class RestControllerExceptionHandler {  
  
    @ExceptionHandler(Exception.class)  
    public ResponseEntity<String> handleException(Exception ex) {  
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("error: " + ex.getMessage());  
    }  
}
```

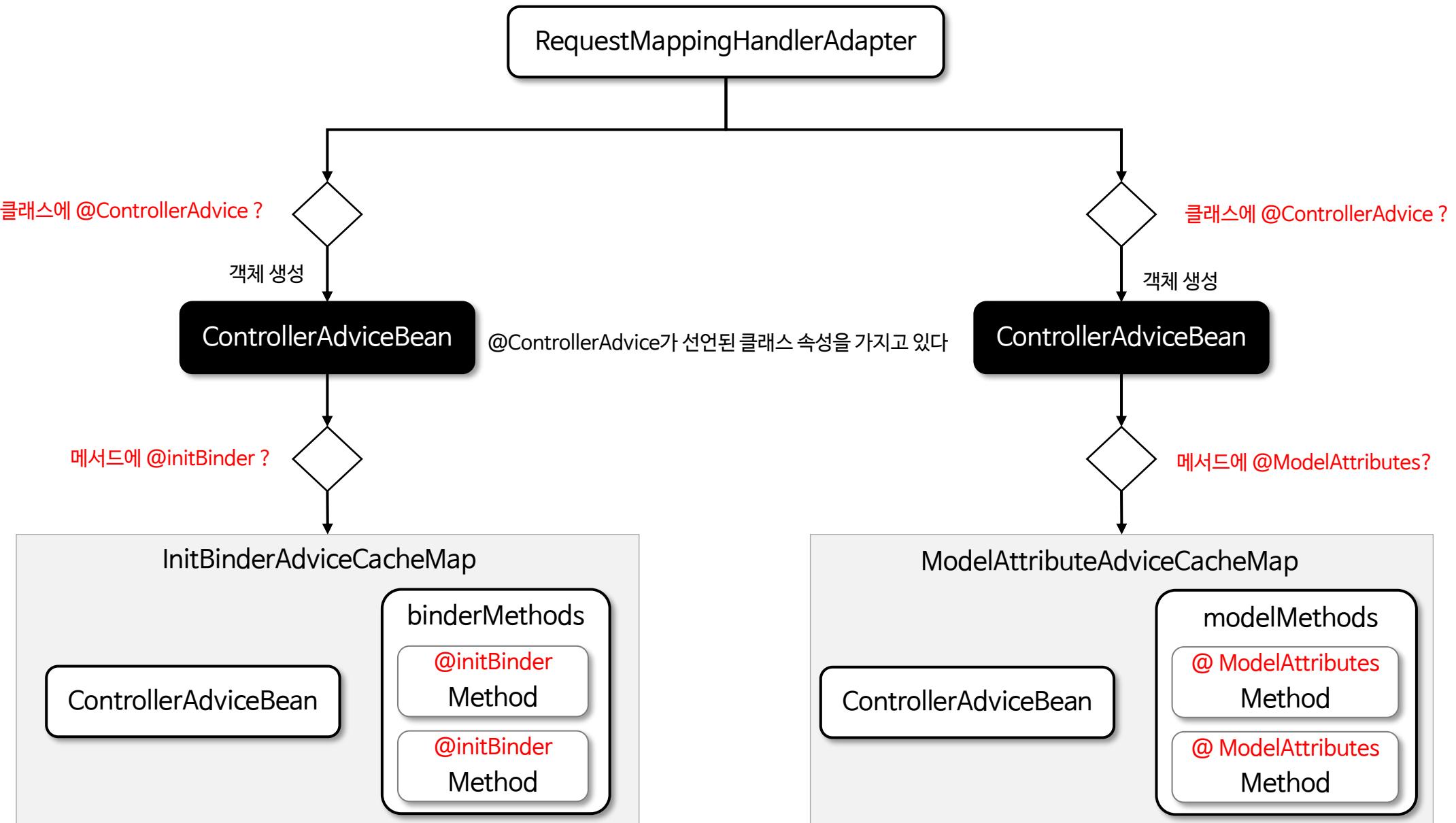
✓ 여러 @ControllerAdvice 적용 순서

```
@RestControllerAdvice  
@Order(1) // 우선순위 높음  
public class RestGlobalExceptionHandler {  
    @ExceptionHandler(IllegalArgumentException.class)  
    public ResponseEntity<String> handleIllegalArgumentException(IllegalArgumentException ex) {  
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("[RestControllerAdvice] Error: " + ex.getMessage());  
    }  
}  
  
@ControllerAdvice  
@Order(2) // 우선순위 낮음  
public class GlobalExceptionHandler {  
    @ExceptionHandler(IllegalArgumentException.class)  
    public ResponseEntity<String> handleIllegalArgumentException(IllegalArgumentException ex) {  
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("[ControllerAdvice] Error: " + ex.getMessage());  
    }  
}
```

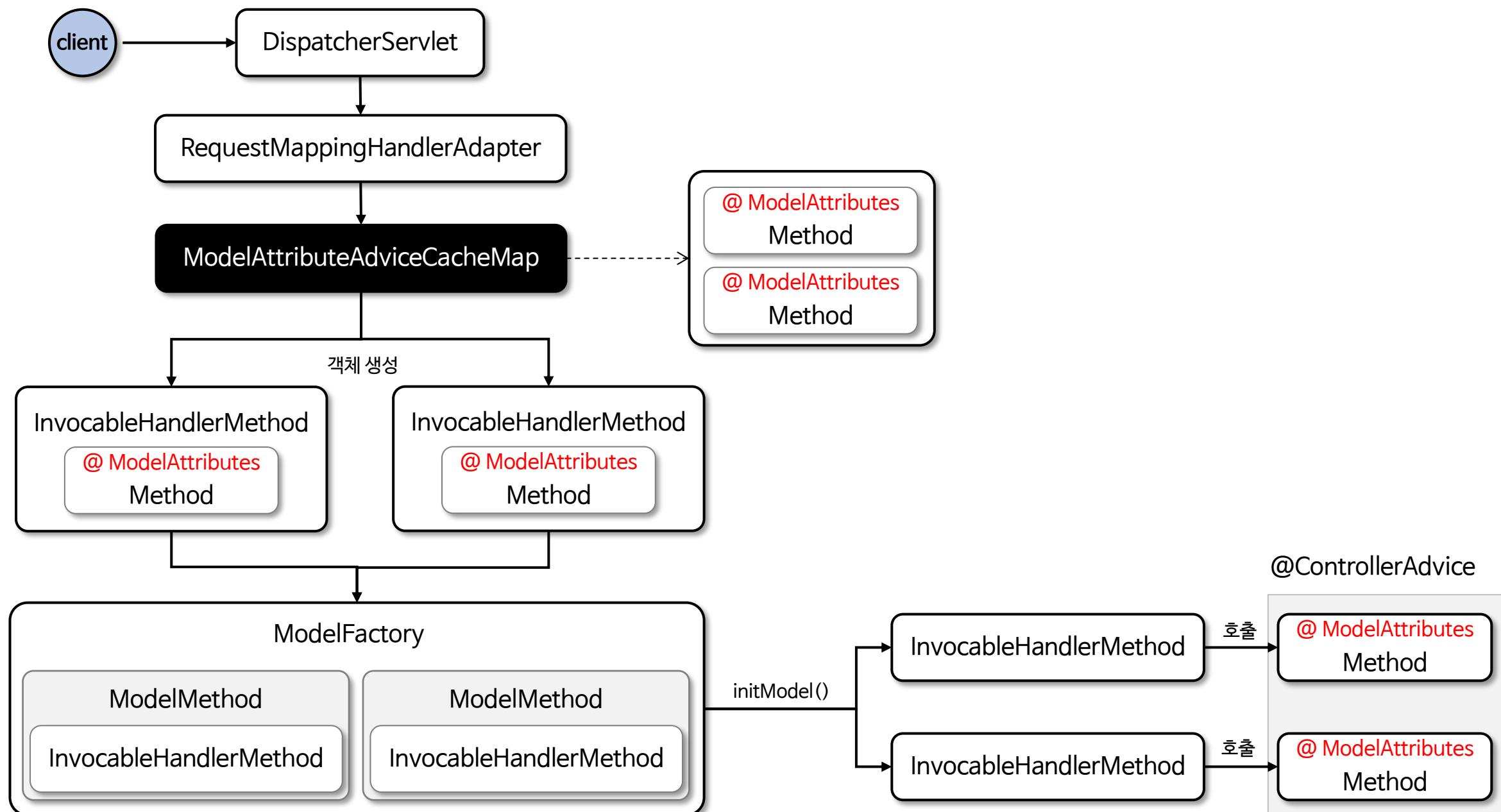
```
@Controller  
@RequestMapping("/api")  
public class TestController {  
    @GetMapping("/error")  
    public String throwException() {  
        throw new IllegalArgumentException();  
    }  
}
```

- 다중 @ControllerAdvice 가 있을 경우 @Order 숫자가 낮을수록 높은 우선순위를 가진다
- 여러 개의 ControllerAdvice 에서 동일한 예외 타입이 선언되어 있을 경우 어떤 @ControllerAdvice 가 우선순위가 높은지 명확한 순서를 보장하지 않기 때문에 @Order 를 사용해 순서를 지정해 사용하도록 한다
- 컨트롤러 내부에 @ExceptionHandler가 있으면 @ControllerAdvice 보다 우선 적용됨

✓ 초기화



✓ 흐름도



@ControllerAdvice 응용하기

<https://github.com/onjsdnjs/spring-mvc-master/tree/@ControllerAdvice-응용하기>

✓ 개요

- @ControllerAdvice 의 원리 및 내부 구조를 파악하고 이를 활용해서 커스텀한 기능을 구현해 본다

✓ 주요 클래스 이해

CustomRequestMappingHandlerAdapter

RequestMappingHandlerAdapter 를 상속한 클래스로서 @ControllerAdvice 의 초기화 작업과 CustomServletInvocableHandlerMethod 를 생성하고 요청을 진행한다

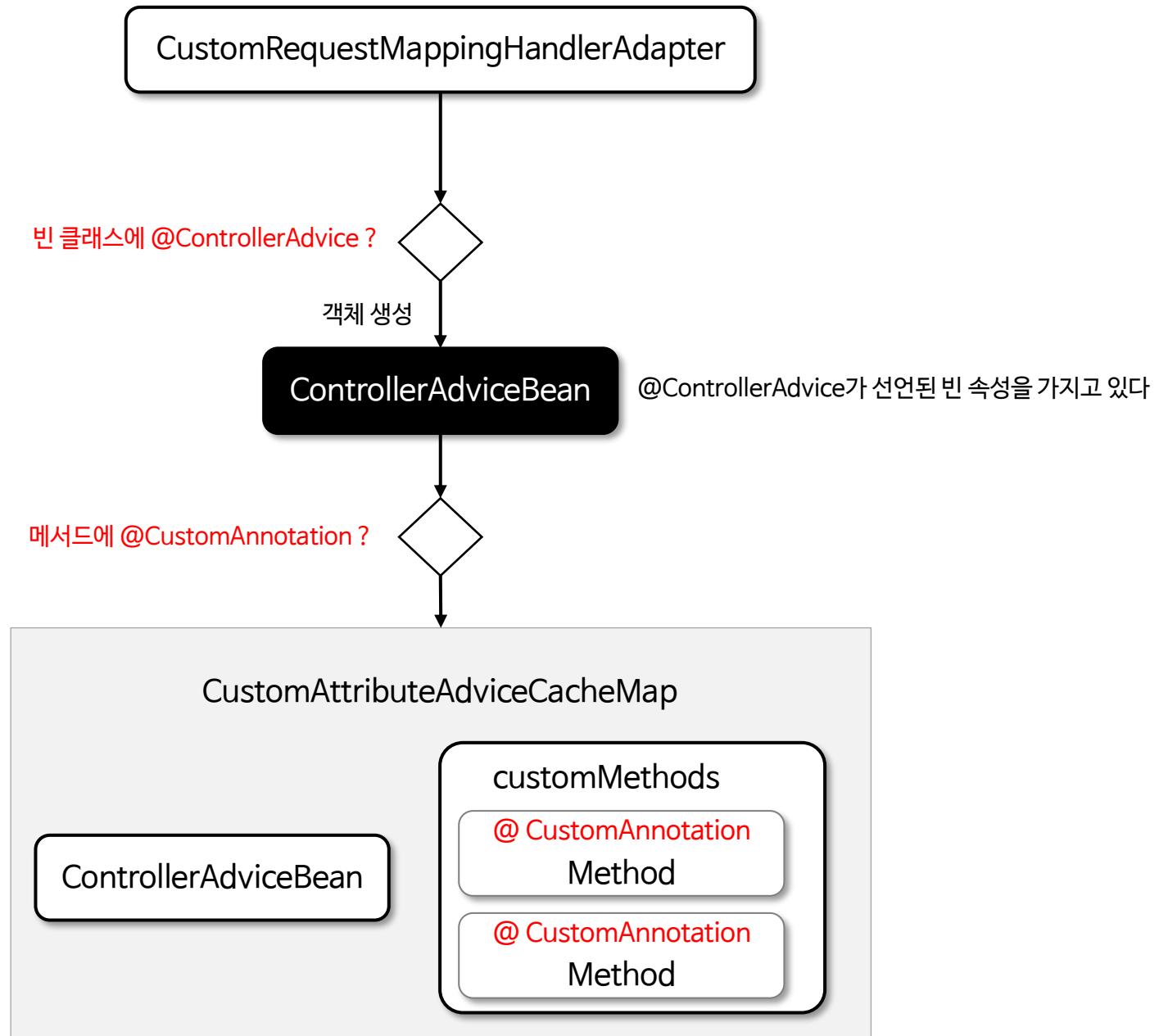
CustomServletInvocableHandlerMethod

ServletInvocableHandlerMethod 클래스를 상속한 클래스로서 @ControllerAdvice 의 클래스에 선언된 @CustomAnnotation 의 메서드를 호출하고 핸들러를 실행한다

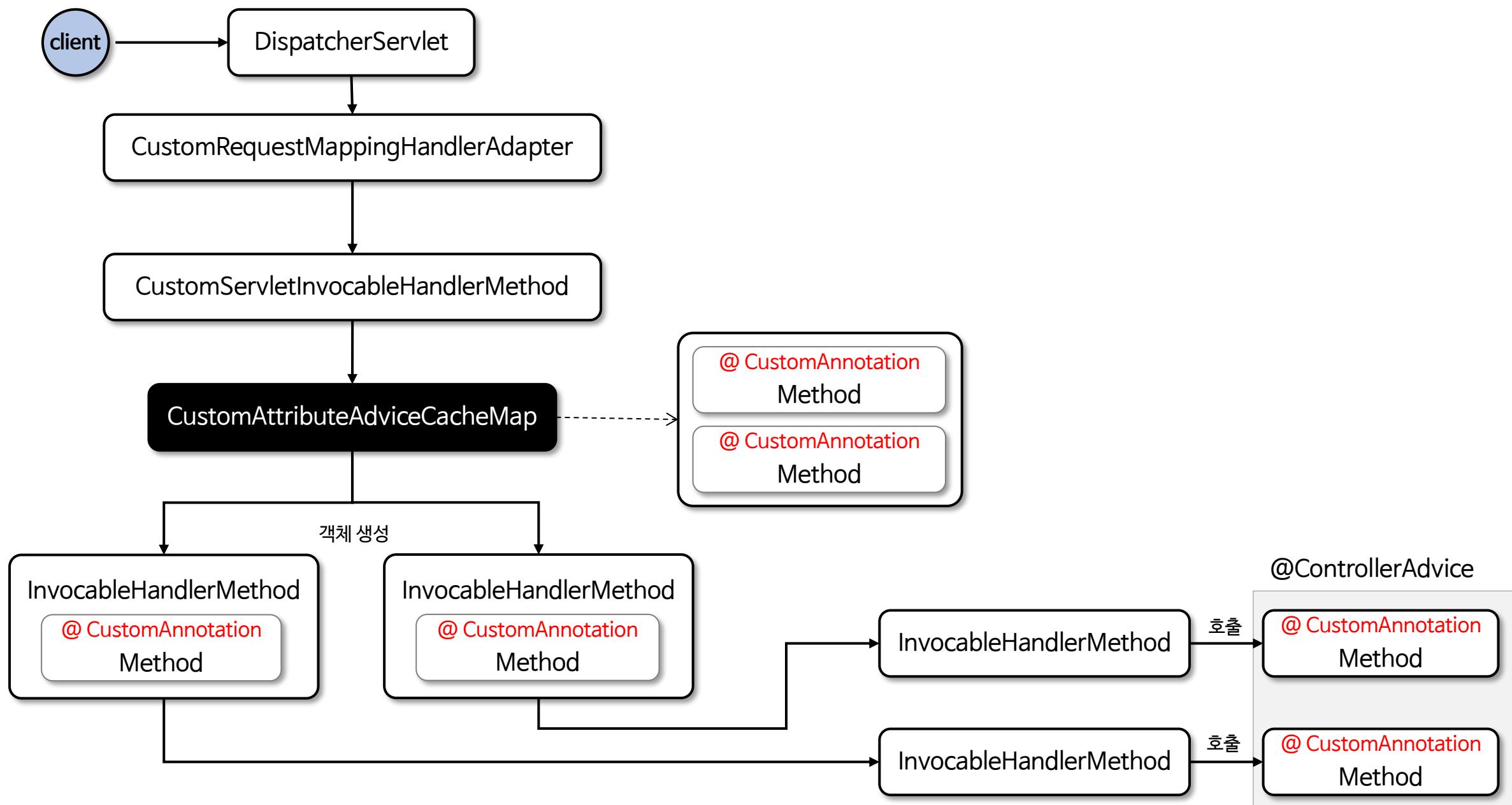
@CustomAnnotation

@ControllerAdvice 에서 사용될 커스텀 어노테이션

✓ 초기화



✓ 흐름도





스프링 웹 MVC 완전 정복

✓ 예외 처리

1. 서블릿 예외 처리
2. WAS 표준 오류 정책 – ErrorPage
3. 스프링의 기본 오류 처리 – BasicErrorController
4. 스프링의 통합 예외 전략
5. HandlerExceptionResolver 개요
6. HandlerExceptionResolver 기본 구현체들
7. ExceptionHandlerExceptionResolver – @ExceptionHandler
8. ExceptionHandlerExceptionResolver – @ControllerAdvice

서블릿 예외 처리

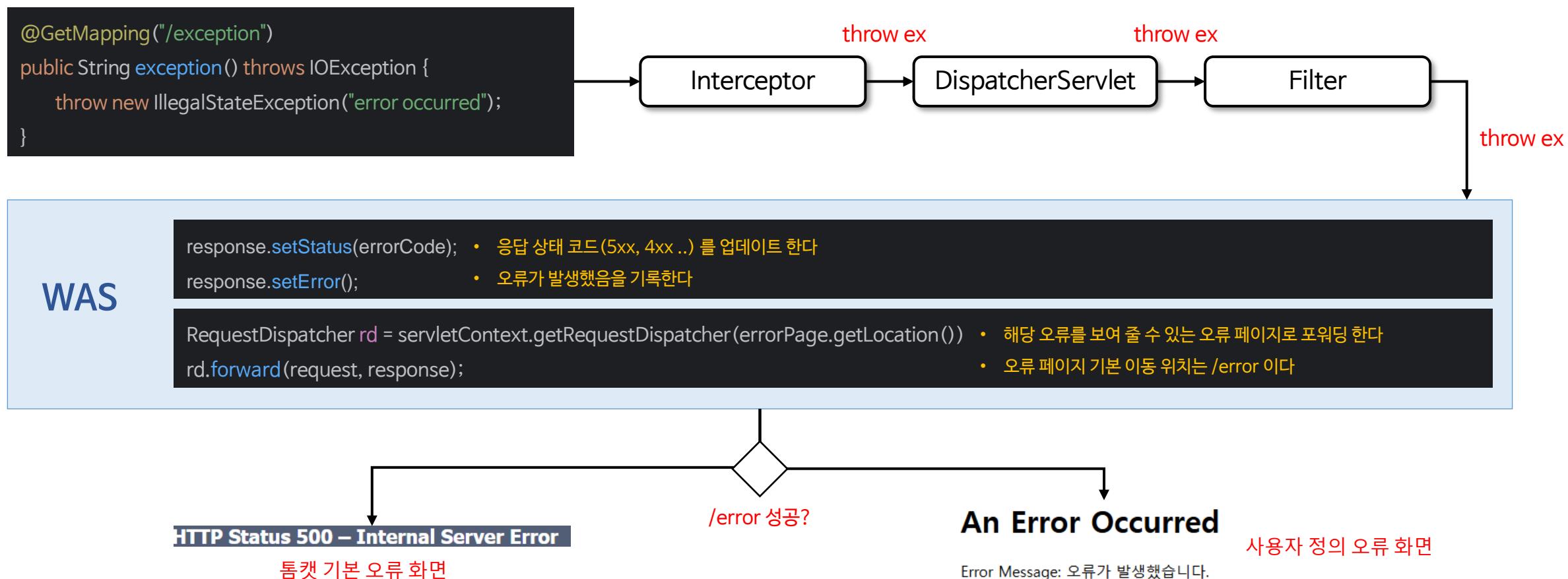
<https://github.com/onjsdnjs/spring-mvc-master/tree/서블릿-예외-처리>

✓ 개요

- 클라이언트 요청 처리 중 발생한 예외는 컨트롤러, 필터, 서블릿, DispatcherServlet 등에서 처리되지 않을 경우 상위 계층으로 전파되어 최종 WAS 까지 전달 된다
- 서블릿은 크게 두 가지 유형으로 오류처리를 지원하는데 하나는 실제 예외가 발생하는 경우, 또 하나는 Response.sendError() 를 통해 오류 상태를 기록하는 경우이다

✓ 예외 발생 (Throw Exception)

- 컨트롤러에서 발생한 예외나 오류는 응답상태코드에 따라 서버내부오류, 페이지 찾을 수 없음 오류 등으로 나타나며 WAS 에서 최종 오류 처리를 제어한다



✓ Response.sendError()

- Response.sendError() 는 요청이 잘못되었거나 서버에서 처리할 수 없는 상황이 발생했을 때 클라이언트에게 HTTP 상태 코드와 오류 메시지를 전송하기 위해 사용된다

✓ API

- sendError(int sc) - HTTP 상태 코드만 전송
- sendError(int sc, String msg) - HTTP 상태 코드와 함께 사용자 정의 오류 메시지 전송

✓ 동작 방식

```
@GetMapping("/senderror")
public void senderror(HttpServletRequest response) {
    response.sendError(500);
}
```

```
response.setStatus(errorCode);
response.setError();
```

응답 상태 코드(5xx, 4xx ..) 및 오류가 발생했음을 기록한다

Interceptor

DispatcherServlet

Filter

WAS

```
RequestDispatcher rd = servletContext.getRequestDispatcher(errorPage.getLocation())
rd.forward(request, response);
```

- 해당 오류를 보여 줄 수 있는 오류 페이지로 포워딩 한다
- 오류 페이지 기본 이동 위치는 /error 이다

HTTP Status 500 – Internal Server Error
톰캣 기본 오류 화면

/error 성공?

An Error Occurred

사용자 정의 오류 화면

Error Message: 오류가 발생했습니다.

WAS 표준 오류 정책 – ErrorPage

<https://github.com/onjsdnjs/spring-mvc-master/tree/ErrorPage>

✓ ErrorPage

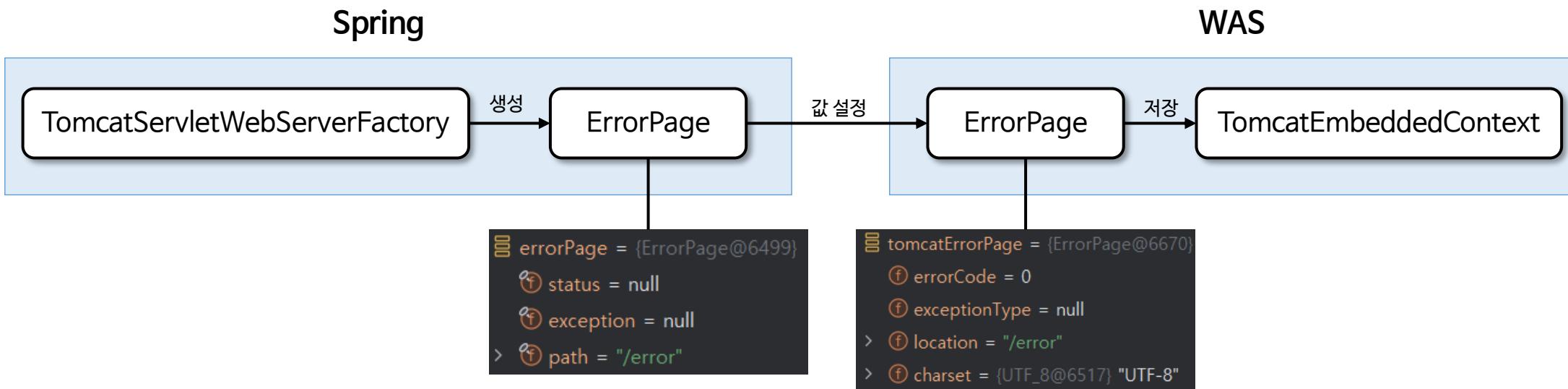
- ErrorPage 는 WAS 에서 발생하는 예외나 특정 HTTP 상태 코드에 대해 오류 페이지를 설정하고 렌더링하는 기능을 제공하는 클래스이다
- ErrorPage 는 클라이언트에게 서버의 오류 상황을 명확히 전달하고 사용자 친화적인 메시지를 제공하기 위한 표준적인 방법이다

ErrorPage	
exceptionType	String
location	String
errorCode	int

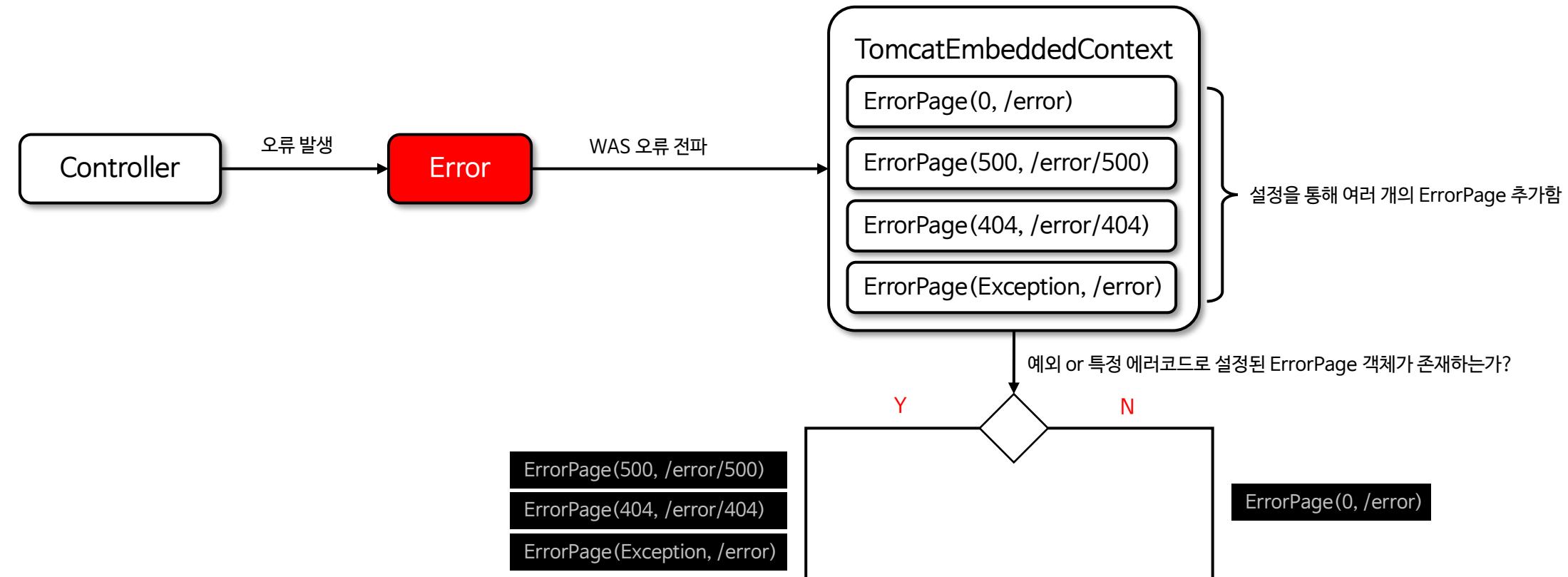
- 특정 Java 예외가 발생했을 때 이를 처리하기 위한 오류 페이지가 활성화 된다
- 오류나 예외를 처리하기 위한 경로이다
- 특정 HTTP 상태 코드(예: 404, 500)가 발생했을 때 이를 처리하기 위한 오류 페이지가 활성화된다

✓ 초기화 작업

- 어플리케이션이 초기화 되면 스프링의 ErrorPage 와 WAS 의 ErrorPage 를 각각 생성하고 기본값들로 채우게 되며 WAS 에는 기본 오류 페이지 한 개가 생성된다
- ErrorPage 는 설정을 통해 여러 개 추가 할 수 있으며 application.properties 파일에 server.error.xxx 로 ErrorPage 의 기본 속성 값을 변경할 수 있다



✓ ErrorPage 동작 방식



WAS

```
RequestDispatcher rd = servletContext.getRequestDispatcher(errorPage.getLocation());
rd.forward(request, response);
```

- 해당 오류를 보여 줄 수 있는 오류 페이지로 포워딩 한다
- 오류 페이지 기본 이동 위치는 /error 이다

HTTP Status 500 – Internal Server Error
톰캣 기본 오류 화면

/error 성공?

An Error Occurred

Error Message: 오류가 발생했습니다.

사용자 정의 오류 화면

✓ ErrorPage 추가

- xml 방식의 웹 환경에서는 web.xml 에 추가하면 되고 스프링 부트에서는 Java Config 방식 또는 빈으로 만들어 추가할 수 있다

web.xml

```
<web-app>
    <!-- 404 상태 코드에 대한 오류 페이지 -->
    <error-page>
        <error-code>404</error-code>
        <location>/errors/404.html</location>
    </error-page>

    <!-- 특정 예외 유형에 대한 오류 페이지 -->
    <error-page>
        <exception-type>java.lang.RuntimeException</exception-type>
        <location>/errors/error.html</location>
    </error-page>
</web-app>
```

Java Config or Bean

```
@Component
public class CustomTomcatWebServerCustomizer implements
WebServerFactoryCustomizer<TomcatServletWebServerFactory> {

    @Override
    public void customize(TomcatServletWebServerFactory factory) {
        // 401 에러에 대한 사용자 정의 페이지 설정
        factory.addErrorPages(new ErrorPage(HttpStatus.UNAUTHORIZED, "/error/401"));

        // 404 에러에 대한 사용자 정의 페이지 설정
        factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND, "/error/404"));

        // 500 에러에 대한 사용자 정의 페이지 설정
        factory.addErrorPages(new ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/error/500"));

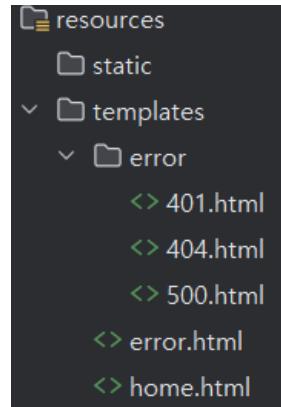
        // Exception 발생 시 사용자 정의 페이지 설정
        factory.addErrorPages(new ErrorPage(ServletException.class, "/error/exception"));
    }
}
```

✓ 오류 페이지 구현

```
@Controller  
public class ExceptionController {  
    @GetMapping("/exception")  
    public String exception(HttpServletRequest response) throws IOException {  
        throw new IllegalStateException("error occurred");  
    }  
  
    @GetMapping("/error500")  
    public void error500(HttpServletRequest response) throws IOException {  
        response.sendError(500);  
    }  
  
    @GetMapping("/error404")  
    public void error404(HttpServletRequest response) throws IOException {  
        response.sendError(404);  
    }  
  
    @GetMapping("/error401")  
    public void error401(HttpServletRequest response) throws IOException {  
        response.sendError(401);  
    }  
}
```



```
@Controller  
public class ErrorPageController {  
    // 401 에러 페이지  
    @GetMapping("/error/401")  
    public String unAuthorized() {  
        return "error/401"; // error/401.html로 이동  
    }  
    // 404 에러 페이지  
    @GetMapping("/error/404")  
    public String handleNotFound() {  
        return "error/404"; // error/404.html로 이동  
    }  
    // 500 에러 페이지  
    @GetMapping("/error/500")  
    public String handleServerError() {  
        return "error/500"; // error/500.html로 이동  
    }  
    // Exception 에러 페이지  
    @GetMapping("/error/ex")  
    public String handleExceptionError() {  
        return "error"; // error.html로 이동  
    }  
}
```



✓ 오류 정보 전달 메커니즘

- WAS는 오류가 발생했을 때 오류 페이지를 다시 요청하는 것 외에 오류에 대한 상세 정보를 HttpServletRequest의 속성에 추가하여 전달한다
- HttpServletRequest 속성에 저장된 요청의 상태 코드, 발생한 예외, 오류 메시지 등을 동적으로 보여 주거나 활용할 수 있다

✓ 오류 속성

속성 이름	설명	상수 이름
jakarta.servlet.error.status_code	HTTP 상태 코드 (예: 404, 500 등)	RequestDispatcher.ERROR_STATUS_CODE
jakarta.servlet.error.message	오류 메시지	RequestDispatcher.ERROR_MESSAGE
jakarta.servlet.error.exception	발생한 예외 객체	RequestDispatcher.ERROR_EXCEPTION
jakarta.servlet.error.servlet_name	오류가 발생한 서블릿의 이름	RequestDispatcher.ERROR_SERVLET_NAME
jakarta.servlet.error.request_uri	오류가 발생한 요청 URI	RequestDispatcher.ERROR_REQUEST_URI
jakarta.servlet.error.exception_type	발생한 예외 객체의 클래스 타입	RequestDispatcher.ERROR_EXCEPTION_TYPE

```
@GetMapping("/error/401")
public String unAuthorized(HttpServletRequest request) {
    String servletName = (String)request.getAttribute(RequestDispatcher.ERROR_SERVLET_NAME);
    String exception = (String)request.getAttribute(RequestDispatcher.ERROR_EXCEPTION);
    String message = (String)request.getAttribute(RequestDispatcher.ERROR_MESSAGE);
    String requestUri = (String)request.getAttribute(RequestDispatcher.ERROR_REQUEST_URI);
    String exceptionType = (String)request.getAttribute(RequestDispatcher.ERROR_EXCEPTION_TYPE);
    int attribute = (int)request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
    return "error/401";
}
```

```
✉ servletName = "dispatcherServlet"
✉ exception = null
✉ message = "Unauthorized"
✉ requestUri = "/401"
✉ exceptionType = null
✉ attribute = 401
```

스프링의 기본 오류 처리

BasicErrorController

<https://github.com/onjsdnjs/spring-mvc-master/tree/BasicErrorController>

✓ 개요

- BasicErrorController 는 Spring Boot 에서 제공하는 기본적인 오류 처리 컨트롤러로 어플리케이션에서 발생하는 예외 또는 오류를 처리하고 기본적인 오류 페이지 및 JSON 형식의 오류 응답을 반환한다
- BasicErrorController 는 기본적으로 /error 경로로 요청하는 모든 오류를 처리하고 있으며 이는 WAS 에서 오류 페이지를 요청하는 기본 경로인 /error 와 일치한다

✓ 구조 및 특징

BasicErrorController

ModelAndView **errorHtml**(HttpServletRequest request, HttpServletResponse response)

ResponseEntity<Map<String, Object>> **error**(HttpServletRequest request)

- HTML 오류 페이지 반환
- JSON 형식으로 본문 응답

• 오류 처리의 기본 동작 제공

- 클라이언트로부터 발생하는 HTTP 상태 코드를 기반으로 HTML 오류 페이지(Whitelabel Error Page) 혹은 REST API의 경우 JSON 형식으로 응답한다

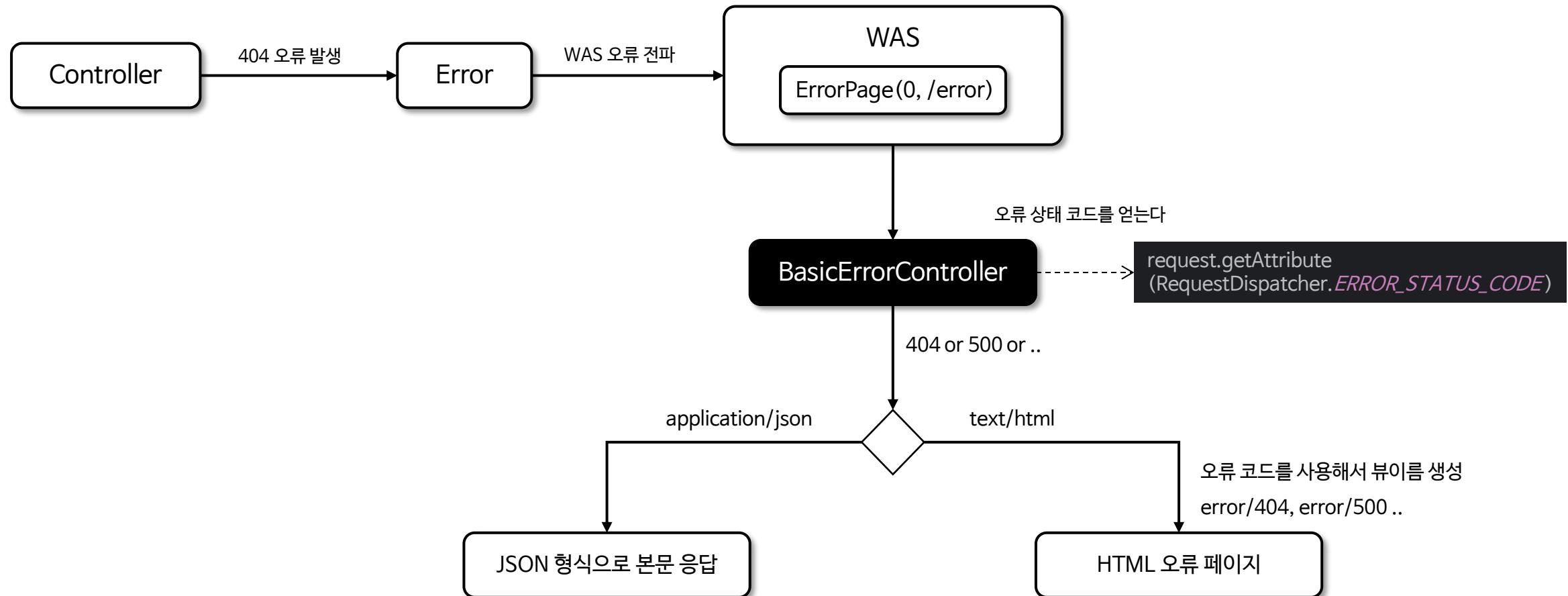
• 사용자 정의 오류 처리

- 개발자는 ErrorController 를 구현하여 기본 동작을 커스터마이징할 수 있다

• ErrorAttributes 와 연동

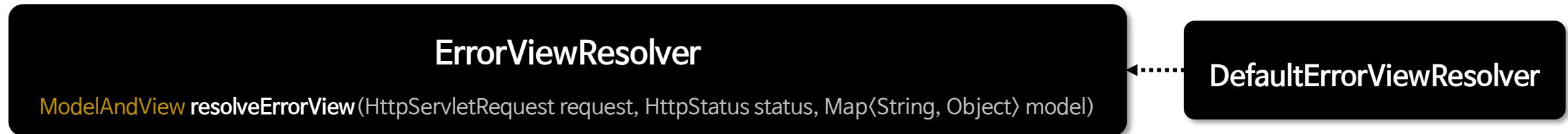
- 오류 관련 데이터를 제공하는 ErrorAttributes 와 연계하여 오류의 세부 정보를 응답에 포함할 수 있다

✓ 오류 처리 방식

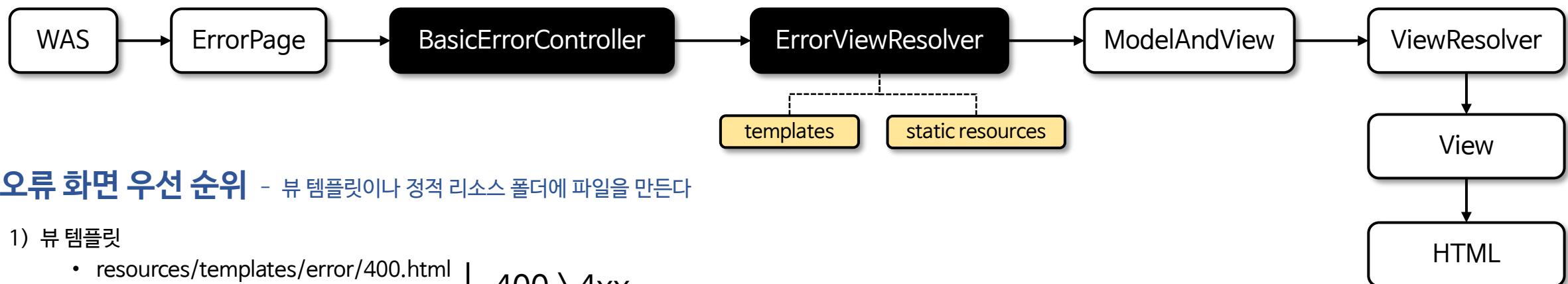


✓ View 방식의 오류 처리 - ErrorViewResolver

- ErrorViewResolver 는 오류가 발생했을 때 보여줄 화면(오류 페이지)을 찾는 역할을 한다
- 기본적으로 /error/ 경로 아래에서 오류 코드(예: 404, 500) 나 오류의 종류에 맞는 템플릿 파일이나 정적 리소스를 찾아서 적절한 화면을 보여주는 역할을 한다



✓ 흐름도



✓ 오류 화면 우선 순위 - 뷰 템플릿이나 정적 리소스 폴더에 파일을 만든다

1) 뷰 템플릿

- resources/templates/error/400.html
 - resources/templates/error/4xx.html
- | 400 > 4xx

2) 정적 리소스

- resources/static/error/500.html
 - resources/static/error/5xx.html
- | 500 > 5xx

3) 적용 대상이 없을 때(error)

- resources/templates/error.html

```
staticLocations = {String[4]@6772} ["classpath:/META-INF/resources/", "classpath:/resources/", "classpath:/static/", "classpath:/public/"]
```

✓ Rest API 방식의 오류 처리

- Spring Boot는 REST 요청(Accept: application/json)이 발생했을 때 BasicErrorController를 사용해 JSON 형식의 오류 응답을 자동으로 생성해 준다

✓ 흐름도



✓ 오류 처리

요청

```
GET http://localhost:8080/api/404 HTTP/1.1  
Accept: application/json
```

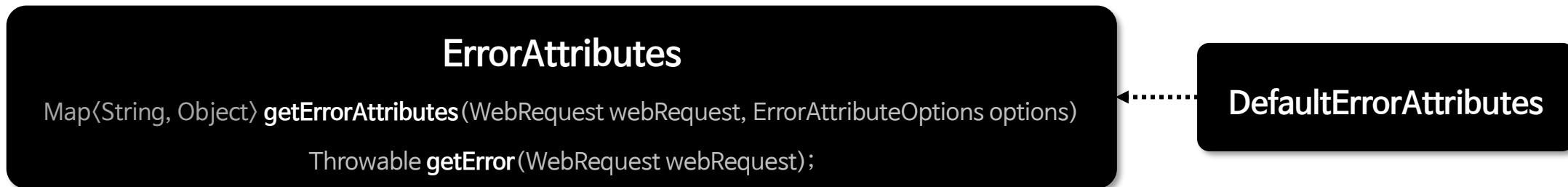
응답

```
GET http://localhost:8080/api/404  
HTTP/1.1 404  
Content-Type: application/json  
  
{  
    "timestamp": "2024-11-23T09:55:01.705+00:00",  
    "status": 404,  
    "error": "Not Found"  
}
```

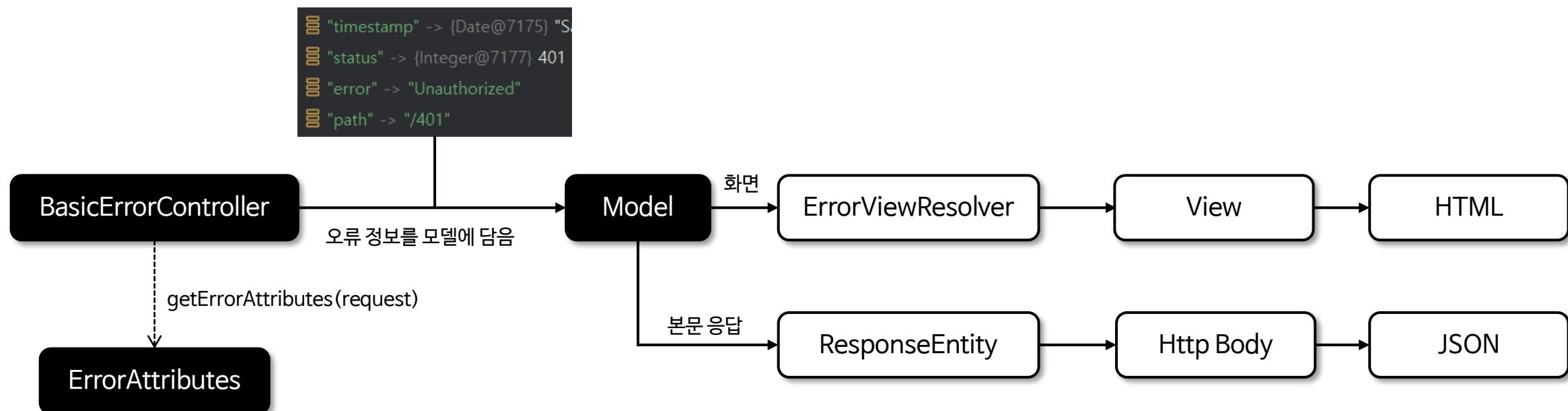
*BasicErrorController는 기본적인 화면 오류 처리에는 매우 유용하지만 API 오류 처리를 위한 세밀한 요구사항을 충족하는 데는 한계가 있다
자세한 내용은 @ExceptionHandler 챕터에서 설명하겠다*

✓ 오류 정보 처리 - ErrorAttributes

- 오류가 발생했을 때 오류와 관련된 정보를 가져올 수 있게 해주는 기능으로 Model에 담아서 이 정보를 로그로 기록하거나 사용자에게 보여줄 때 사용할 수 있다



✓ 흐름도



✓ 오류 정보

- **timestamp** - 오류 정보가 추출된 시간
- **status** - HTTP 상태 코드(404, 500)
- **error** - 오류 원인("Not Found", "Internal Server Error")
- **exception** - 최상위 예외 클래스 이름
- **message** - 예외 메시지
- **errors** - BindingResult에서 발생한 ObjectErrors
- **trace** - 예외 스택 트레이스
- **path** - 예외가 발생한 URL 경로

404.html

```
<p>Timestamp: <span th:text="${timestamp}"></span></p>  
<p>Status Code: <span th:text="${status}"></span></p>  
<p>Error: <span th:text="${error}"></span></p>  
<p>Message: <span th:text="${message}"></span></p>  
<p>Path: <span th:text="${path}"></span></p>  
<p>Exception: <span th:text="${exception}"></span></p>  
<p>Stack Trace:</p><pre th:text="${trace}"></pre>
```

Json

```
{  
    "timestamp": "2024-11-23T09:55:01",  
    "status": 404,  
    "error": "Not Found"  
}
```

✓ 오류 정보 제어 설정 - application.properties

- server.error.include-message
- server.error.include-binding-errors
- server.error.include-exception
- server.error.include-stacktrace
- server.error.include-path

설정 값	설명	사용 환경	예시 사용
never	정보를 절대 포함하지 않음	운영 환경	보안이 중요한 API
always	정보를 항상 포함	개발 환경	로컬 디버깅
on_param	요청에 특정 파라미터가 있을 때만 포함	개발 (꼭 필요 시 운영)	?message=&exception=

운영 환경 (application-prod.properties)

```
server.error.include-message=never  
server.error.include-binding-errors=never  
server.error.include-exception=false  
server.error.include-stacktrace=never
```

개발 환경 (application-dev.properties)

```
server.error.include-message=always  
server.error.include-binding-errors=always  
server.error.include-exception=true  
server.error.include-stacktrace=always
```

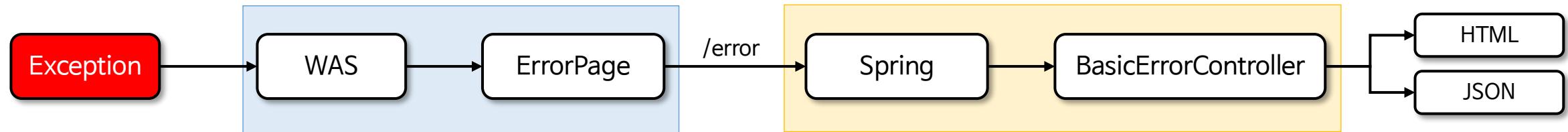
스프링의 통합 예외 전략

개요

<https://github.com/onjsdnjs/spring-mvc-master/tree/스프링의-통합-예외-전략-개요>

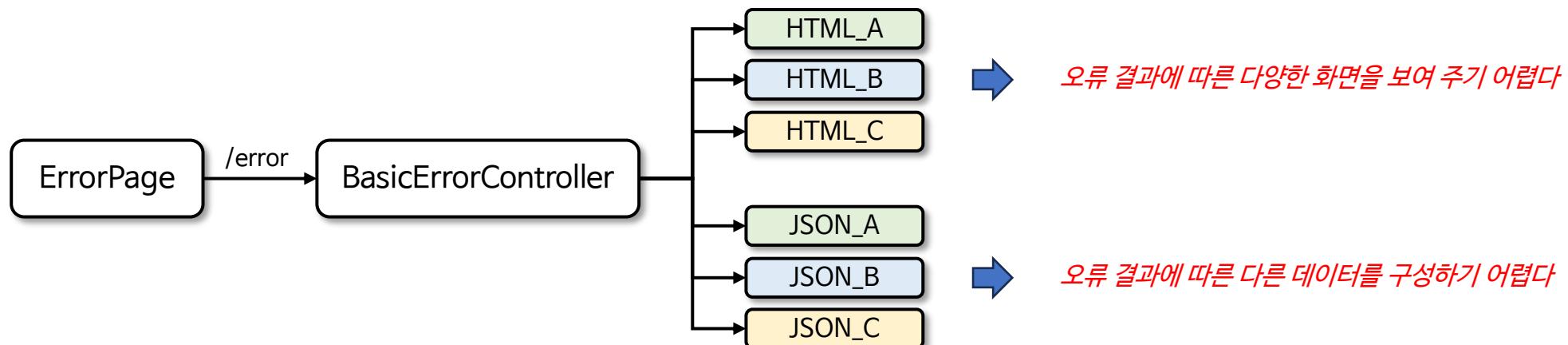
✓ ErrorPage & BasicErrorHandler 예외 전략

- 오류 발생 시 WAS는 오류 정보를 담은 ErrorPage를 사용해서 /error로 요청을 포워딩하고 BasicErrorHandler가 그 요청을 받아 적절한 오류 처리를 한다
- text/html 방식의 요청이 들어오면 사용자 정의 오류 페이지(HTML)를 반환하고 REST API 방식의 요청이 들어오면 JSON 형태로 오류 정보를 반환 한다
- 템플릿이나 정적 리소스에 오류 페이지를 구성해 놓으면 오류 상태 코드나 예외에 따라 자동으로 적절한 오류 화면을 반환하거나 데이터를 반환한다



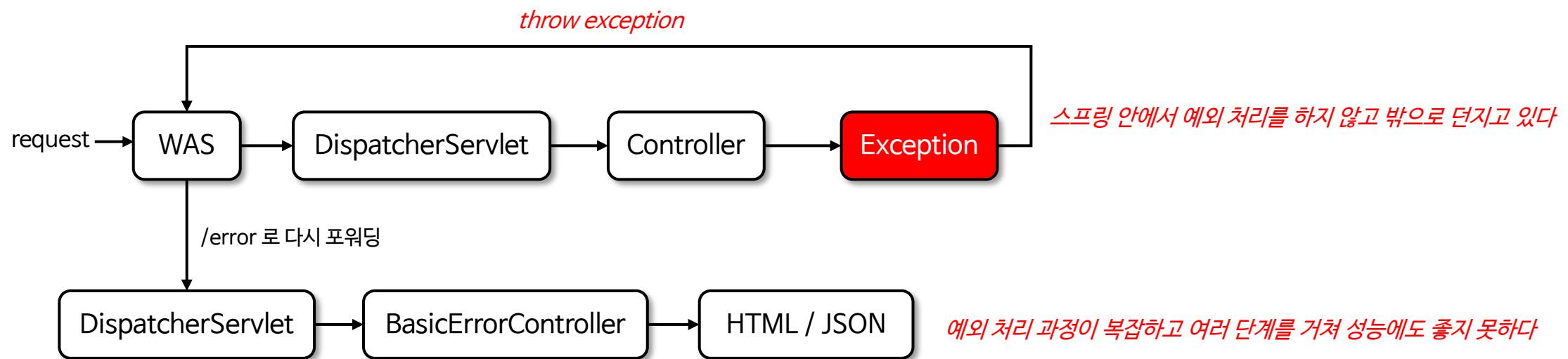
✓ ErrorPage & BasicErrorHandler 한계점

- WAS의 ErrorPage는 주로 정적인 HTML 페이지 또는 JSP로 연결되며 동적인 데이터나 사용자 정의 응답을 제공하기 어렵다
- BasicErrorHandler는 예외를 전역적으로 처리하지만 특정 컨트롤러나 요청 경로에 따른 세분화된 처리 로직을 구현하기 어렵다
- REST API에서 특정 예외(예: 인증 오류, 회원 오류, 주문 오류)에 대해 메시지와 상태 코드를 반환 하려면 개별적인 오류 형태 구현이 필요하다



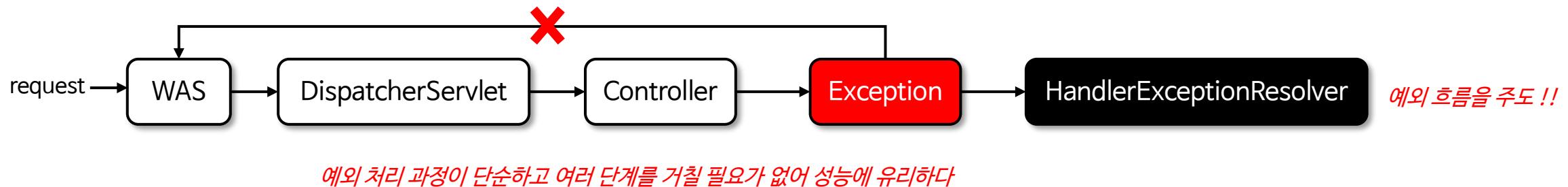
✓ ErrorPage & BasicErrorHandler 오류 처리 과정

- ① 클라이언트 요청 → WAS로 요청 전달
- ② WAS → DispatcherServlet 요청 전달
- ③ DispatcherServlet → 컨트롤러 맵핑
- ④ 컨트롤러에서 예외 발생
- ⑤ WAS 까지 예외가 전파되어 예외를 처리 → /error로 포워딩
- ⑥ BasicErrorHandler에서 /error 요청을 받고 JSON 또는 HTML 오류 화면 생성하고 클라이언트로 응답



✓ 개선 된 오류 처리 과정

- ① 클라이언트 요청 → WAS 로 요청 전달
- ② WAS → DispatcherServlet 요청 전달
- ③ DispatcherServlet → 컨트롤러 매핑
- ④ 컨트롤러에서 예외 발생
- ⑤ 스프링 MVC 의 흐름 안에서 예외 해결
- ⑥ WAS 까지 예외가 전파되어 예외를 처리 → /error로 포워딩
- ⑦ BasicErrorHandler에서 /error 요청을 받고 JSON 또는 HTML 오류 화면 생성하고 클라이언트로 응답



✓ 더 강력한 스프링 예외 전략 제공

- 특정 예외마다 다른 처리 로직을 구현할 수 있으며 예외 유형별로 HTTP 상태 코드, 응답 메시지, 추가 데이터 등을 원하는 대로 설정할 수 있다
- WAS의 오류 처리 메커니즘에 의존하지 않고 애플리케이션 코드 내에서 예외를 처리하고 응답을 반환할 수 있다
- 스프링 MVC에서 발생한 예외는 HandlerExceptionResolver 클래스가 해결하도록 한다

HandlerExceptionResolver

<https://github.com/onjsdnjs/spring-mvc-master/tree/HandlerExceptionResolver>

✓ 개요

- HandlerExceptionResolver 인터페이스는 요청 처리 중에 발생한 예외를 처리하고 그 결과를 사용자에게 보여줄 수 있는 에러 페이지로 연결해 주는 역할을 한다
- 즉 컨트롤러나 핸들러 실행 중에 문제가 생기면 이 문제를 어떻게 해결하고 어떤 에러 화면을 보여줄지를 정해주는 역할을 한다

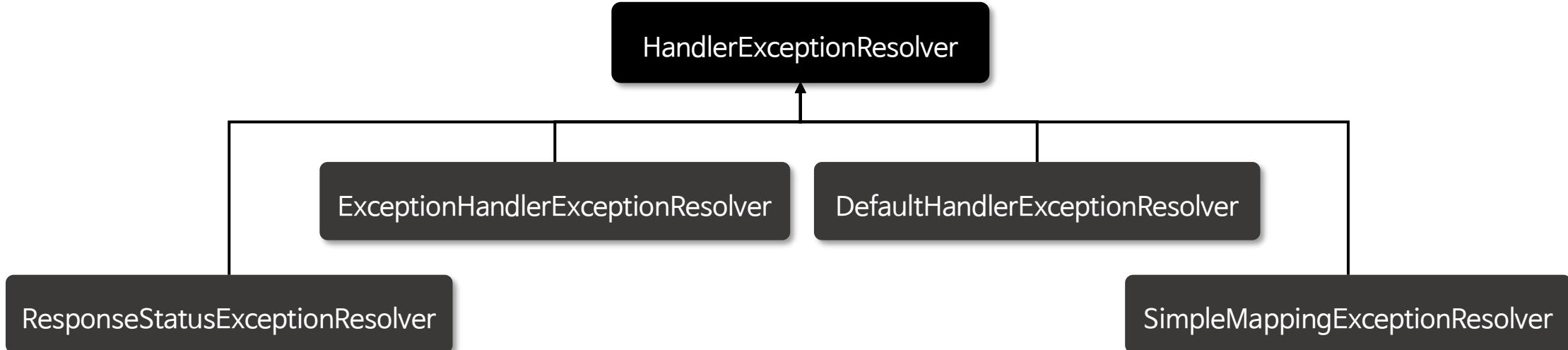
✓ 구조

HandlerExceptionResolver

ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, @Nullable Object handler, Exception ex)

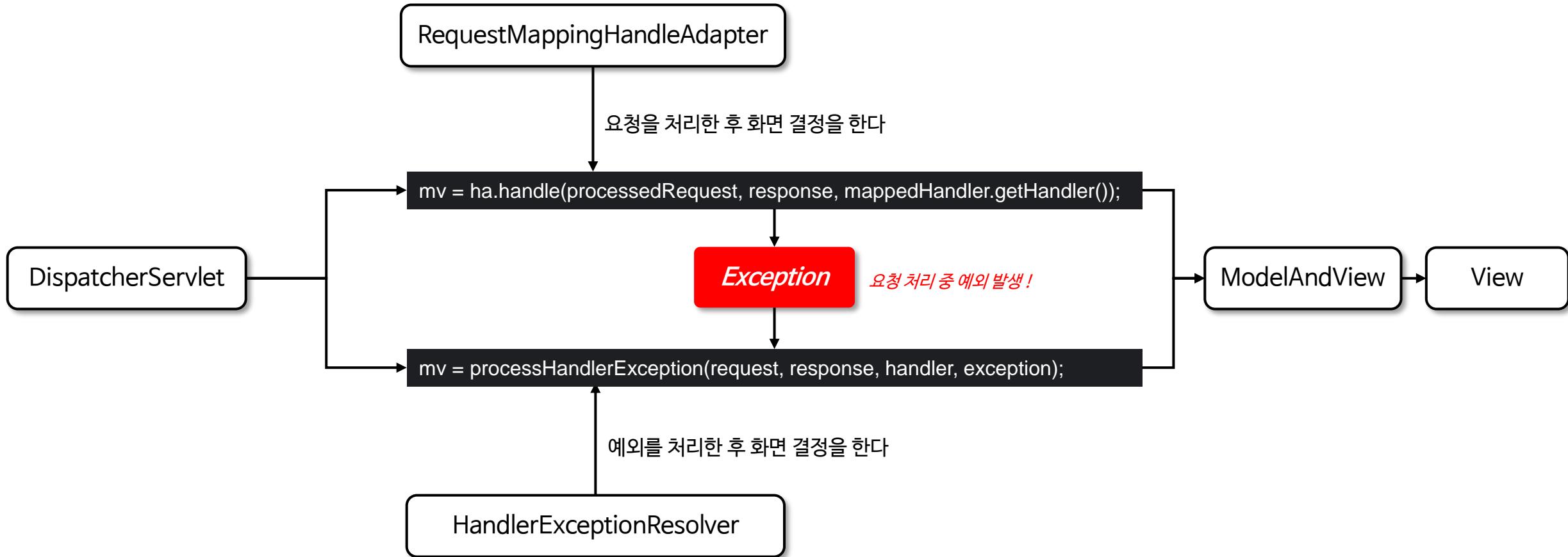
- handler - 요청을 처리하기 위해 선택된 핸들러 객체(컨트롤러)
- ex - 요청을 처리하는 도중에 발생한 예외 객체

✓ 구현체



✓ HandlerExceptionResolver 의 예외 전략

- HandlerExceptionResolver 는 RequestMappingHandleAdapter 가 핸들러 실행 후 ModelAndView 객체를 반환하는 것과 동일한 구조를 가지고 있다
- 즉 예외 상황에서도 기존의 MVC 처리 흐름을 벗어나지 않고 정상적인 흐름 안에서 예외 처리를 구현할 수 있다



✓ 기본 구현

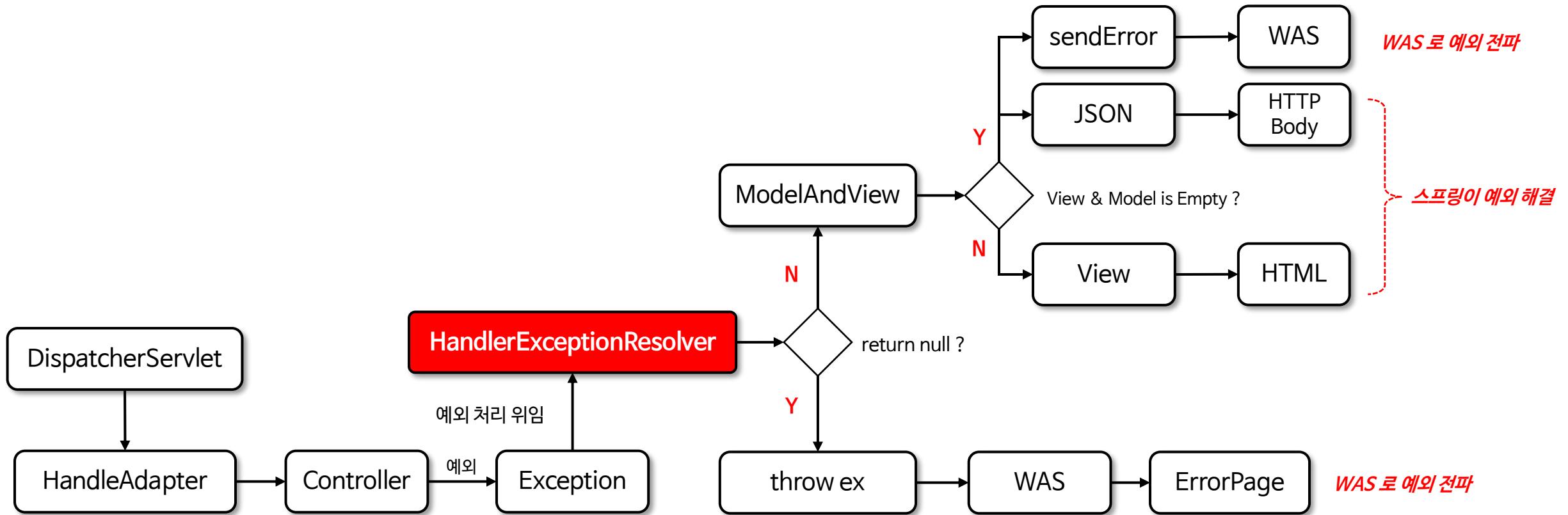
```
public class CustomHandlerExceptionResolver implements HandlerExceptionResolver {  
    @Override  
    public ModelAndView resolveException(HttpServletRequest req, HttpServletResponse resp, Object handler, Exception ex) {  
        if (ex instanceof CustomException) {  
            return new ModelAndView(viewName,model); // 오류 화면으로 응답  
        } else if (ex instanceof RuntimeException) {  
            response.sendError(HttpServletRequest.SC_BAD_REQUEST, ex.getMessage()); // WAS로 오류 전파  
            return new ModelAndView(); // 오류 화면 생성하지 않음  
        }  
        return null;  
    }  
}
```

- 1) `new ModelAndView()` - 화면을 렌더링 하지 않고 JSON 형태로 예외를 응답하거나 `sendError(errCode)`로 예외 처리를 WAS로 넘기고자 할 경우이다
- 2) `new ModelAndView(viewName,model)` - 지정한 경로로 화면을 렌더링 한다
- 3) `null` - 예외를 그대로 WAS로 전달하고 기본 `ErrorPage`의 `/error`로 오류 처리가 이루어진다

✓ WebMvcConfigurer 등록

```
@Override  
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {  
    resolvers.add(new CustomHandlerExceptionResolver());  
}
```

✓ HandlerExceptionResolver 예외 처리 흐름도



✓ 구현 예제 - CustomHandlerExceptionResolver

```
@Component
public class CustomHandlerExceptionResolver implements HandlerExceptionResolver {
    @Override
    public ModelAndView resolveException(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        String acceptHeader = request.getHeader("Accept");
        response.setStatus(HttpStatus.INTERNAL_SERVER_ERROR.value());
        try {
            if (acceptHeader.contains(MediaType.TEXT_HTML_VALUE)) {
                return new ModelAndView("error-view"); // 오류 화면으로 응답
            }
            if (acceptHeader.contains(MediaType.APPLICATION_JSON_VALUE)) {
                Map<String, Object> errorDetails = new HashMap<>();
                errorDetails.put("message", ex.getMessage());
                response.getWriter().write(errorDetails.toString()); // JSON 형태로 응답
                return new ModelAndView();
            }
            if (ex instanceof RuntimeException) {
                response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), ex.getMessage()); // WAS로 오류 전파
                return new ModelAndView();
            }
            return null; // WAS로 예외 전달
        } catch (Exception handlerEx) { return null; }
    }
}
```

✓ 구현 예제 - ExceptionController

```
@Controller  
public class TestController {  
    @GetMapping("/error-view")  
    public void throwViewError() {  
        throw new IllegalArgumentException ("HTML View Error Occurred!");  
    }  
    @GetMapping("/error-json")  
    public void throwJsonError() {  
        throw new IllegalStateException("JSON Error Occurred!");  
    }  
    @GetMapping("/error-sendError")  
    public void throwSendError() {  
        throw new RuntimeException("sendError triggered!");  
    }  
}
```

/error-view

```
curl -H "Accept: text/html" http://localhost:8080/error-view  
error-view.html
```

/error-json

```
curl -H "Accept: application/json" http://localhost:8080/error-json  
{  
    "status": 500,  
    "error": "Internal Server Error",  
    "message": "JSON Error Occurred!"  
}
```

/error-sendError

```
curl http://localhost:8080/error-sendError  
HTTP/1.1 500 Internal Server Error  
Content-Type: text/plain
```

HandlerExceptionResolver 기본 구현체들

<https://github.com/onjsdnjs/spring-mvc-master/tree/HandlerExceptionResolver-기본-구현체들>

✓ 개요

- Spring은 기본적으로 예외 처리에 사용할 수 있는 HandlerExceptionResolver 구현체를 제공하며 각 구현체는 특정 시나리오에 따라 예외를 처리하도록 설계되어 있다
- 구현체는 ExceptionHandlerExceptionResolver, ResponseStatusExceptionResolver, DefaultHandlerExceptionResolver, SimpleMappingExceptionResolver로서 총 4개의 클래스가 제공된다

✓ 초기화 구성



- 예외 처리시 HandlerExceptionResolverComposite 가 가장 우선순위가 높으며 다음으로 resolvers 에 들어 있는 순서대로 호출 된다
- CustomHandlerExceptionResolver 를 직접 구현하게 되면 기본 구현체들 다음으로 순서가 정해진다
- 실무에서 가장 많이 사용하는 구현체는 ExceptionHandlerExceptionResolver이며 @ExceptionHandler 어노테이션과 함께 사용한다

✓ ResponseStatusExceptionResolver

- ResponseStatusExceptionResolver 는 예외에 대해 HTTP 상태 코드와 메시지를 매핑하여 클라이언트에게 반환할 수 있도록 설계된 예외 처리 전략이다
- 이 구현체는 두 가지 방식으로 예외 및 HTTP 상태 코드를 처리하는데 @ResponseStatus 와 ResponseStatusException 을 사용하여 구현한다
- 이 클래스는 예외를 sendError(code, msg) 로 처리하기 때문에 뷰 렌더링 없이 WAS 의 ErrorPage 전략에 의해 예외 처리가 이루어지도록 한다

✓ @ResponseStatus

- @ResponseStatus 는 code 와 reason 속성으로 이루어져 있으며 reason 은 값을 직접 설정하거나 MessageSource 기능을 사용해 얻을 수 있다

```
@ □ ResponseStatus
  m □ code()    HttpStatus
  m □ value()   HttpStatus
  m □ reason()  String
```

code - 응답에 사용할 상태 코드

reason - 응답에 사용할 이유

```
@ResponseStatus(code = HttpStatus.NOT_FOUND, reason = "Resource Not Found") or
@ResponseStatus(code = HttpStatus.NOT_FOUND, reason = "error.resource")
public class ResourceNotFoundException extends RuntimeException {
}
```

messages.properties

```
error.resource=Resource Not Found
```

```
@GetMapping("/resource/{id}")
public String getResource(@PathVariable String id) {
    throw new ResourceNotFoundException();
}
```

http://localhost:8080/resource/456

```
{
  "timestamp": "2024-11-27T02:27:37.490+00:00",
  "status": 404,
  "error": "Not Found",
  "path": "/resource/452"
}
```

✓ ResponseStatusException

- @ResponseStatus 는 정적으로 예외와 상태 코드를 매핑하기 때문에 런타임에 다른 값을 설정할 수 없다
- ResponseStatusException 는 동적으로 상태 코드와 메시지를 설정할 수 있어 다양한 상황에서 재사용 가능하다

ResponseStatusException

```
public ResponseStatusException(HttpStatusCode status, @Nullable String reason)
```

✓ 구현 예제

```
@GetMapping("/validate")
public String validateInput(@RequestParam(required = false) String input) {
    if (input == null || input.isEmpty()) {
        throw new ResponseStatusException(
            HttpStatus.BAD_REQUEST,
            "Input is invalid"
        );
    }
    return "Valid Input: " + input;
}
```

http://localhost:8080/validate

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
{
    "status": 400,
    "error": "Bad Request",
    "message": "Input is invalid",
    "path": "/validate"
}
```

✓ DefaultHandlerExceptionResolver

- DefaultHandlerExceptionResolver 는 Spring 의 표준 예외와 HTTP 상태 코드를 자동으로 매핑하여 처리하는 클래스다
- 주로 Spring MVC 내부에서 발생하는 예외들을 처리하며 특정 예외를 HTTP 상태 코드에 매핑시켜 클라이언트로 반환하는 역할을 한다
- 이 클래스는 예외를 sendError(code, msg) 로 처리하기 때문에 뷰 렌더링 없이 WAS 의 ErrorPage 전략에 의해 예외 처리가 이루어지도록 한다

✓ 지원 예외 목록

Exception	HTTP Status Code
HttpRequestMethodNotSupportedException	405 (SC_METHOD_NOT_ALLOWED)
HttpMediaTypeNotSupportedException	415 (SC_UNSUPPORTED_MEDIA_TYPE)
HttpMediaTypeNotAcceptableException	406 (SC_NOT_ACCEPTABLE)
MissingPathVariableException	500 (SC_INTERNAL_SERVER_ERROR)
MissingServletRequestParameterException	400 (SC_BAD_REQUEST)
MissingServletRequestPartException	400 (SC_BAD_REQUEST)
ServletRequestBindingException	400 (SC_BAD_REQUEST)
ConversionNotSupportedException	500 (SC_INTERNAL_SERVER_ERROR)
TypeMismatchException	400 (SC_BAD_REQUEST)
HttpMessageNotReadableException	400 (SC_BAD_REQUEST)
HttpMessageNotWritableException	500 (SC_INTERNAL_SERVER_ERROR)
MethodArgumentNotValidException	400 (SC_BAD_REQUEST)
MethodValidationException	500 (SC_INTERNAL_SERVER_ERROR)
HandlerMethodValidationException	400 (SC_BAD_REQUEST)
NoHandlerFoundException	404 (SC_NOT_FOUND)
NoResourceNotFoundException	404 (SC_NOT_FOUND)
AsyncRequestTimeoutException	503 (SC_SERVICE_UNAVAILABLE)
AsyncRequestNotUsableException	Not applicable

- Spring 표준 예외를 자동으로 처리하므로 기본 예외가 필요한 경우 추가 설정을 할 필요가 없다
- 예외에 적절한 상태 코드를 설정하여 REST API 응답을 손쉽게 지원한다

✓ 예시

1) HttpRequestMethodNotSupportedException (405)

```
@PostMapping("/test")
public String testMethod() {
    return "POST request processed";
}
```

```
GET http://localhost:8080/test
```

```
HTTP/1.1 405 Method Not Allowed
Allow: POST
Content-Type: application/json
{
    "timestamp": "2024-11-26T10:15:30.123+00:00",
    "status": 405,
    "error": "Method Not Allowed",
    "message": "Request method 'GET' not supported",
    "path": "/test"
}
```

2) HttpMediaTypeNotSupportedException (415)

```
@PostMapping(value = "/consume", consumes = "application/json")
public String consumeJson(@RequestBody String body) {
    return "Consumed JSON: " + body;
}
```

```
POST -H "Content-Type:text/plain" -d "Sample Text" http://localhost:8080/consume
```

```
HTTP/1.1 415 Unsupported Media Type
Content-Type: application/json
{
    "timestamp": "2024-11-26T10:20:45.456+00:00",
    "status": 415,
    "error": "Unsupported Media Type",
    "message": "Content type 'text/plain' not supported",
    "path": "/consume"
}
```

✓ SimpleMappingExceptionResolver

- SimpleMappingExceptionResolver 는 특정 예외와 View 이름을 매핑하여 예외 발생 시 지정된 뷰(View)로 전환해 주는 클래스로서 어플리케이션 전역적으로 작동하며 모든 컨트롤러에 동일한 예외 처리 로직을 적용할 수 있다
- REST API 보다는 주로 HTML 기반의 전통적인 웹 애플리케이션에서 사용하기 적합하다

✓ Java Config 기반 설정

```
@Override
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
    resolvers.add(simpleMappingExceptionResolver());
}

public SimpleMappingExceptionResolver simpleMappingExceptionResolver() {
    SimpleMappingExceptionResolver resolver = new SimpleMappingExceptionResolver();

    Properties exceptionMappings = new Properties(); // 예외와 뷰 매핑
    exceptionMappings.put("java.lang.ArithmeticsException", "errorArithmetics"); // 예외는 전체 이름을 적는다
    exceptionMappings.put("java.lang.NullPointerException", "errorNullPointer");
    resolver.setExceptionMappings(exceptionMappings);

    resolver.setDefaultErrorView("error-default"); // 기본 오류 뷰, 발생한 예외에 맞는 뷰가 없을 경우

    Properties statusCodes = new Properties(); // 뷰 이름과 HTTP 상태 코드 매핑
    statusCodes.put("errorArithmetics ", "400");
    statusCodes.put("errorNullPointer ", "500");
    resolver.setStatusCodes(statusCodes);
    return resolver;
}
```

✓ 구현 예제

```
@Controller  
public class DemoController {  
    @GetMapping("/arithmetic-error")  
    public String arithmeticError() {  
        throw new ArithmeticException("Arithmetic Exception Occurred!");  
    }  
    @GetMapping("/nullpointer-error")  
    public String nullPointerError() {  
        throw new NullPointerException("Null Pointer Exception Occurred!");  
    }  
}
```

errorArithmetic.html

```
<!DOCTYPE html>  
<html>  
<head>  
<title>Arithmetic Error</title>  
</head>  
<body>  
<h1>Arithmetic Exception 발생</h1>  
<p>계산 오류가 발생했습니다.</p>  
</body>  
</html>
```

errorNullpointer.html

```
<!DOCTYPE html>  
<html>  
<head>  
<title>Null Pointer Error</title>  
</head>  
<body>  
<h1>Null Pointer Exception 발생</h1>  
<p>Null 참조 오류가 발생했습니다.</p>  
</body>  
</html>
```

errorDefault.html

```
<!DOCTYPE html>  
<html>  
<head>  
<title>Default Error</title>  
</head>  
<body>  
<h1>오류 발생</h1>  
<p>알 수 없는 오류가 발생했습니다.</p>  
</body>  
</html>
```

ExceptionHandlerExceptionResolver

@ExceptionHandler

<https://github.com/onjsdnjs/spring-mvc-master/tree/ExceptionHandlerExceptionResolver-@ExceptionHandler>

✓ 개요

- ExceptionHandlerExceptionResolver 는 Spring MVC 의 예외 처리 메커니즘 중 가장 널리 사용되는 구현체로 컨트롤러 내부 또는 전역에서 @ExceptionHandler 로 정의된 메서드를 호출하여 예외를 처리한다
- REST API 에서는 요청 데이터나 비즈니스 로직에 따라 오류 정보를 세밀하게 제어해야 할 경우가 많은데 이런 동적이고 유연한 예외 처리가 가능하다
- 특정 컨트롤러와 밀접하게 연결된 예외 처리 뿐 아니라 @ControllerAdvice 를 사용하면 모든 컨트롤러에서 공통적인 예외 처리 로직을 적용할 수 있다

✓ 구조 및 주요 속성

ExceptionHandlerExceptionResolver

```
private HandlerMethodArgumentResolverComposite argumentResolvers // @ExceptionHandler 메서드가 지원하는 파라미터 값 처리
```

```
Private HandlerMethodReturnValueHandlerComposite returnValueHandlers; // @ExceptionHandler 메서드가 지원하는 반환 값 처리
```

```
private final List<HttpMessageConverter<? >> messageConverters // 예외 처리 시 Http 본문 응답으로 변환 하기 위한 컨버터
```

```
private final Map<Class<? >, ExceptionHandlerMethodResolver> exceptionHandlerCache // 컨트롤러에서 처리하는 예외 메서드
```

```
private final Map<ControllerAdviceBean, ExceptionHandlerMethodResolver> exceptionHandlerAdviceCache // @ControllerAdvice 에서 처리하는 예외 메서드
```

```
protected ModelAndView doResolveHandlerMethodException(HttpServletRequest request, HttpServletResponse response, HandlerMethod handlerMethod,  
Exception exception)
```

✓ @ExceptionHandler

- `@ExceptionHandler`는 컨트롤러에 특정 예외를 처리하기 위한 메서드를 정의할 때 사용하는 어노테이션으로서 `ExceptionHandlerExceptionResolver`를 통해 실행되며 컨트롤러 클래스에서만 작동하거나 `@ControllerAdvice`와 함께 사용하여 애플리케이션 전역적으로 동작하도록 설정할 수도 있다

ExceptionHandler

```
Class<? extends Throwable>[] value() default {}
```

value()

- `@ExceptionHandler`에서 어떤 예외를 처리할지 한 개 이상 지정 할 수 있다
- 만약 지정 하지 않으면 메서드 매개변수에 적혀 있는 예외를 기본적으로 처리한다

✓ 기본 구현

```
@RestController
public class UserController {

    @GetMapping("/users/{id}")
    public String getUser(@PathVariable String id) {
        throw new UserNotFoundException("User with ID " + id + " not found");
    }

    @ExceptionHandler(UserNotFoundException.class) // 현재 컨트롤러에서 처리하고자 하는 예외를 메서드에 정의 한다
    public ResponseEntity<String> handleUserNotFound(UserNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

✓ 우선 순위에 따른 예외 처리

- 예외가 발생했을 때 자식 클래스 예외 처리 메서드는 항상 상위 클래스 예외 처리 메서드보다 우선적으로 호출된다
- 즉 구체적인 예외 클래스가 선언된 `@ExceptionHandler` 가 우선적으로 호출되며 덜 구체적인 예외 처리 메서드는 그 다음 순위로 처리된다

```
@RestController
public class ExceptionHandlerController {
    @GetMapping("/name")
    public String handlePriority(@RequestParam("name") String name) {
        if ("child1".equals(name)) {
            throw new ChildException1("ChildException1"); → @ExceptionHandler(ChildException1.class) 메서드 호출
        } else if ("child2".equals(name)) {
            throw new ChildException2("ChildException2"); → @ExceptionHandler(ParentException.class) 메서드 호출
        }
        throw new ParentException("ParentException"); → @ExceptionHandler(ParentException.class) 메서드 호출
    }
    @ExceptionHandler(ChildException1.class) // 보다 구체적인 예외를 먼저 처리
    public String handleChildException1 (ChildException1 ex) {
        return "ChildException1 : " + ex.getMessage();
    }
    @ExceptionHandler(ParentException.class) // 상위 클래스인 ParentException 은 마지막에 처리
    public String handleParentException (ParentException ex) {
        return "ParentException: " + ex.getMessage();
    }
}
```

```
// 부모 예외 클래스
class ParentException extends RuntimeException {
    public ParentException(String message) {
        super(message);
    }
}

// 자식 예외 클래스 1
class ChildException1 extends ParentException {
    public ChildException1(String message) {
        super(message);
    }
}

// 자식 예외 클래스 2
class ChildException2 extends ParentException {
    public ChildException2(String message) {
        super(message);
    }
}
```

✓ 여러 개의 예외를 지정

- 하나의 @ExceptionHandler에서 여러 예외를 동시에 처리할 수 있다

```
@GetMapping("/name")
public String handlePriority(@RequestParam("name") String name) {
    if ("child1".equals(name)) throw new ChildException1("ChildException1");
    } else if ("child2".equals(name)) throw new ChildException2("ChildException2");
    throw new ParentException("ParentException");
}

@ExceptionHandler({ChildException1.class, ChildException2.class, ParentException.class})
public String handleException (Exception ex) {
    return "Exception : " + ex.getMessage();
}
```

✓ 예외를 지정하지 않는 경우

```
@GetMapping("/default")
public String handleDefault() {
    throw new IllegalStateException("Default exception occurred!");
}

@ExceptionHandler → IllegalStateException 예외가 지정됨
public String handleIllegalStateException(IllegalStateException ex) {
    return "Handled default exception: " + ex.getMessage();
}
```

✓ HTML 오류 화면 응답

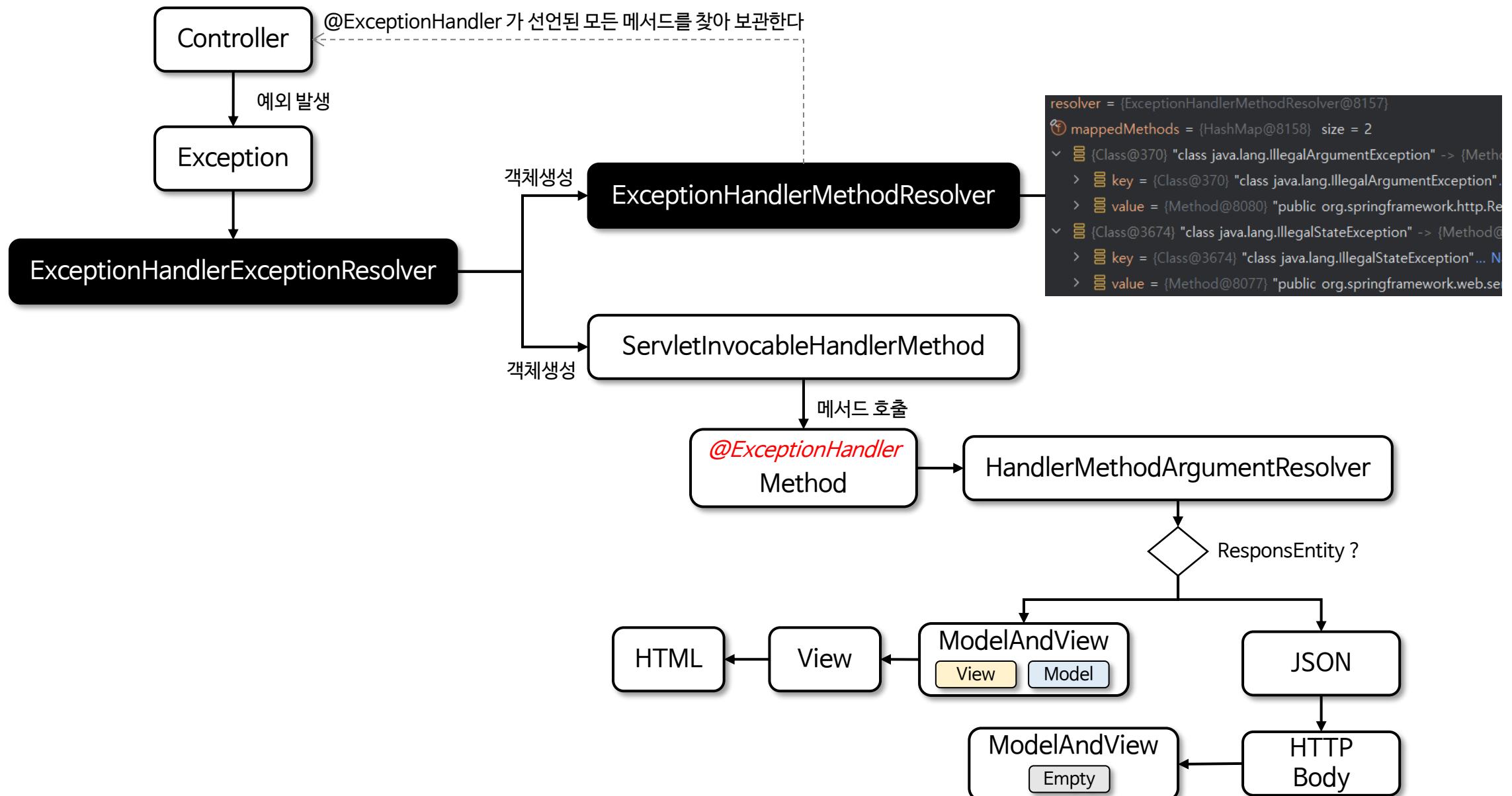
```
@ExceptionHandler(IllegalArgumentException.class)
public ModelAndView argumentException(IllegalArgumentException ex) {
    ModelAndView mav = new ModelAndView("error");
    mav.addObject("message", ex.getMessage());
    return mav;
}
```

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(IllegalStateException.class)
    public String argumentException(IllegalStateException ex, Model model) {
        model.addAttribute("message", ex.getMessage());
        return "error";
    }
}
```

✓ HTTP 본문 응답

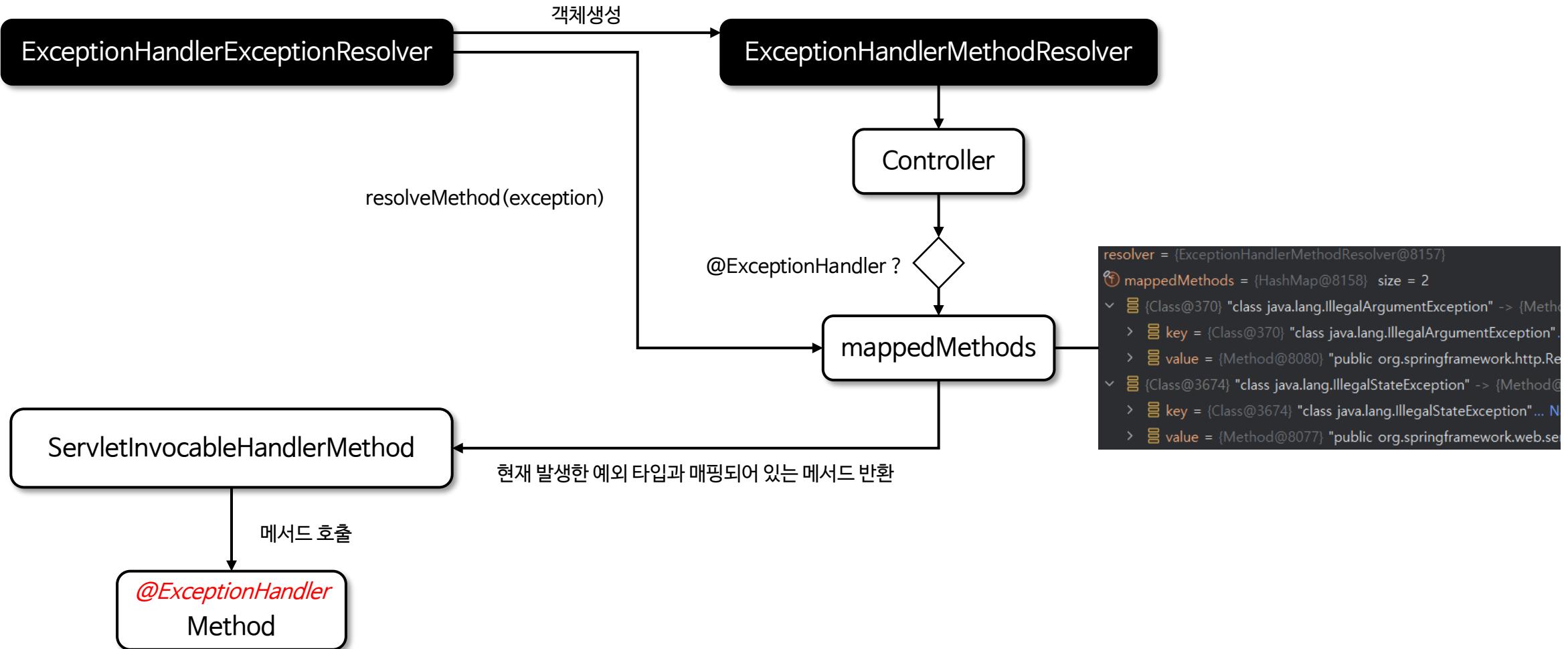
```
@ExceptionHandler(CustomException.class)
//@ResponseBody
public ResponseEntity<ErrorResponse> handleException(CustomException ex) {
    ErrorResponse errorResponse = new ErrorResponseException(HttpStatus.BAD_REQUEST, ex);
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(errorResponse);
}
```

✓ ExceptionHandlerExceptionResolver 구조 이해



✓ ExceptionHandlerMethodResolver

- 주어진 클래스와 해당 클래스의 모든 상위 클래스에서 @ExceptionHandler 메서드를 찾아내어 선언된 예외와 메서드를 매핑하는 역할을 한다
- ExceptionHandlerExceptionResolver 는 ExceptionHandlerMethodResolver 에서 찾아낸 메서드를 호출하는 구조로 되어 있다



ExceptionHandlerExceptionResolver @ControllerAdvice

<https://github.com/onjsdnjs/spring-mvc-master/tree/ExceptionHandlerExceptionResolver-@ControllerAdvice>



개요

- @ControllerAdvice 는 여러 컨트롤러에서 발생하는 예외를 전역적으로 처리할 수 있는 어노테이션으로 ExceptionHandlerExceptionResolver 와 결합하여 작동한다
- @ControllerAdvice 를 사용하면 어플리케이션의 모든 컨트롤러에서 발생하는 예외를 하나의 클래스에서 통합적으로 처리할 수 있으며 이를 통해 중복 코드를 제거하고 예외 흐름을 컨트롤러로부터 분리할 수 있어 유지보수에도 유리하다



사용방법

- 클래스에 @ControllerAdvice 를 선언하고 @ExceptionHandler 메서드를 예외타입 별로 정의한다

```
@ControllerAdvice  
public class GlobalExceptionHandler {  
  
    @ExceptionHandler(CustomException1.class)  
    public ResponseEntity<ErrorResponse> handleException1(CustomException1 ex) {  
        // 예외 처리 로직  
    }  
  
    @ExceptionHandler(CustomException2.class)  
    public ResponseEntity<ErrorResponse> handleException2(CustomException2 ex) {  
        // 예외 처리 로직  
    }  
}
```

✓ 기본 구현 예제

```
@ControllerAdvice  
public class GlobalExceptionHandler {  
  
    @ExceptionHandler(CustomNotFoundException.class)  
    public ResponseEntity<ErrorResponse> handleException1(CustomNotFoundException ex) {  
        ErrorResponse errorResponse = new ErrorResponseException(HttpStatus.NOT_FOUND, ex);  
        return new ResponseEntity<>(errorResponse, HttpStatus.NOT_FOUND);  
    }  
  
    @ExceptionHandler(CustomValidationException.class)  
    public ResponseEntity<ErrorResponse> handleException2(CustomValidationException ex) {  
        ErrorResponse errorResponse = new ErrorResponseException(HttpStatus.BAD_REQUEST, ex);  
        return new ResponseEntity<>(errorResponse, HttpStatus.BAD_REQUEST);  
    }  
  
    @ExceptionHandler(Exception.class)  
    public ResponseEntity<ErrorResponse> handleException3(Exception ex) {  
        ErrorResponse errorResponse = new ErrorResponseException(HttpStatus.INTERNAL_SERVER_ERROR, ex);  
        return new ResponseEntity<>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);  
    }  
}
```

```
@RestController  
@RequestMapping("/api/resources")  
public class ResourceController {  
    @GetMapping("/{id}")  
    public String getResource(@PathVariable String id) {  
        if ("notfound".equals(id)) {  
            throw new CustomNotFoundException("Resource not found");  
        } else if ("invalid".equals(id)) {  
            throw new CustomValidationException("Invalid input");  
        }  
        return "Resource with ID: " + id;  
    }  
}
```

컨트롤러는 정상적인 로직만 처리하고 예외 관련 모든 처리는

`@ControllerAdvice` 클래스가 맙도록 역할을 분리한다

✓ 특정 영역에서만 @ControllerAdvice 적용

```
@ControllerAdvice(assignableTypes = {UserController.class}) // UserController 에만 적용
public class UserControllerAdvice {

    @ExceptionHandler(IllegalArgumentException.class)
    public String handleIllegalArgumentException(IllegalArgumentException ex) {
        return "error";
    }
}
```

```
@RestControllerAdvice(annotations = RestController.class) // @RestController 에만 적용
public class RestControllerAdvice {

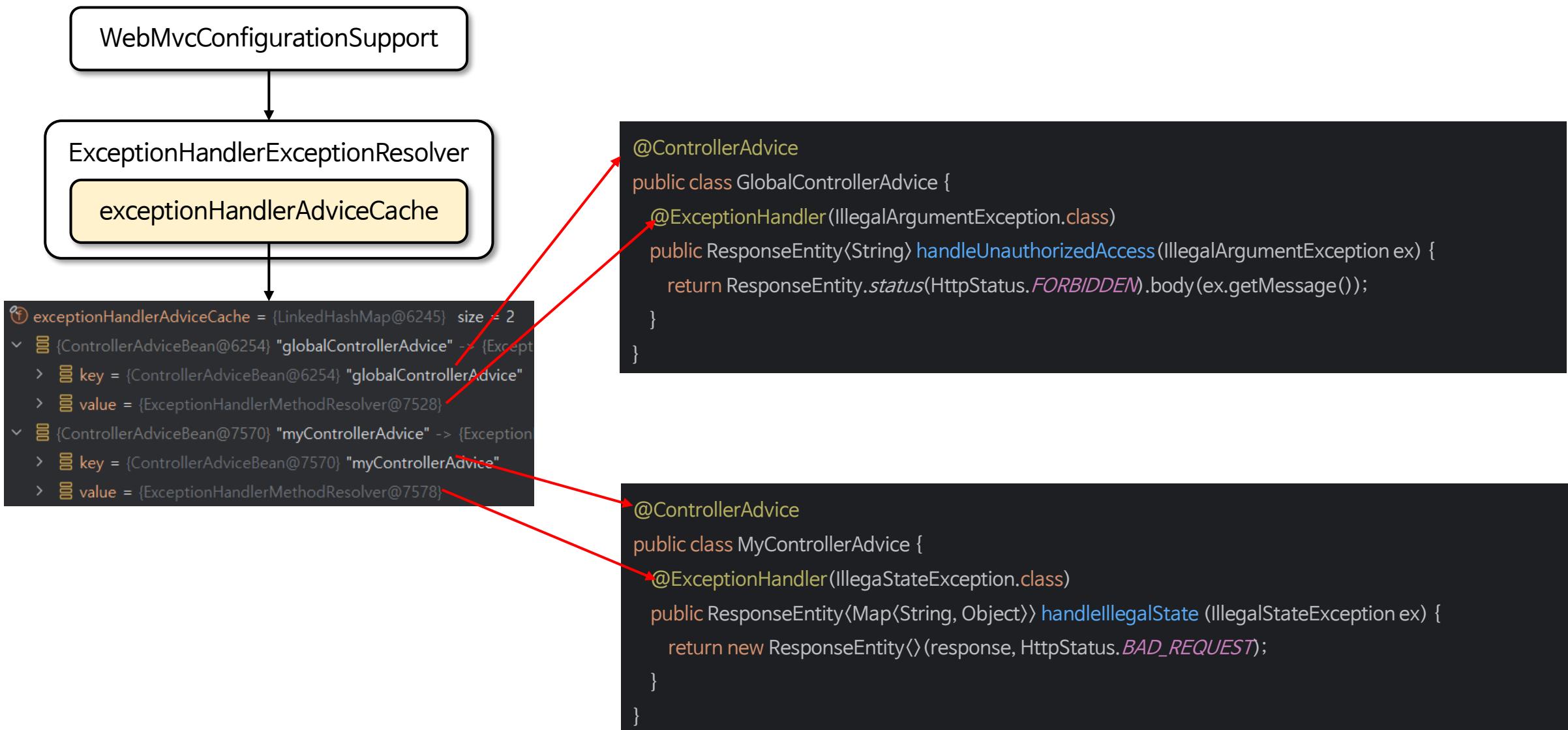
    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleIllegalArgumentException(IllegalArgumentException ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body("Handled in RestControllerAdvice: " + ex.getMessage());
    }
}
```

```
@ControllerAdvice(basePackages = "com.example.user") // com.example.user 패키지 컨트롤러에만 적용
public class UserPackageAdvice {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleIllegalArgumentException(Exception ex) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body("Handled in UserPackageAdvice: " + ex.getMessage());
    }
}
```

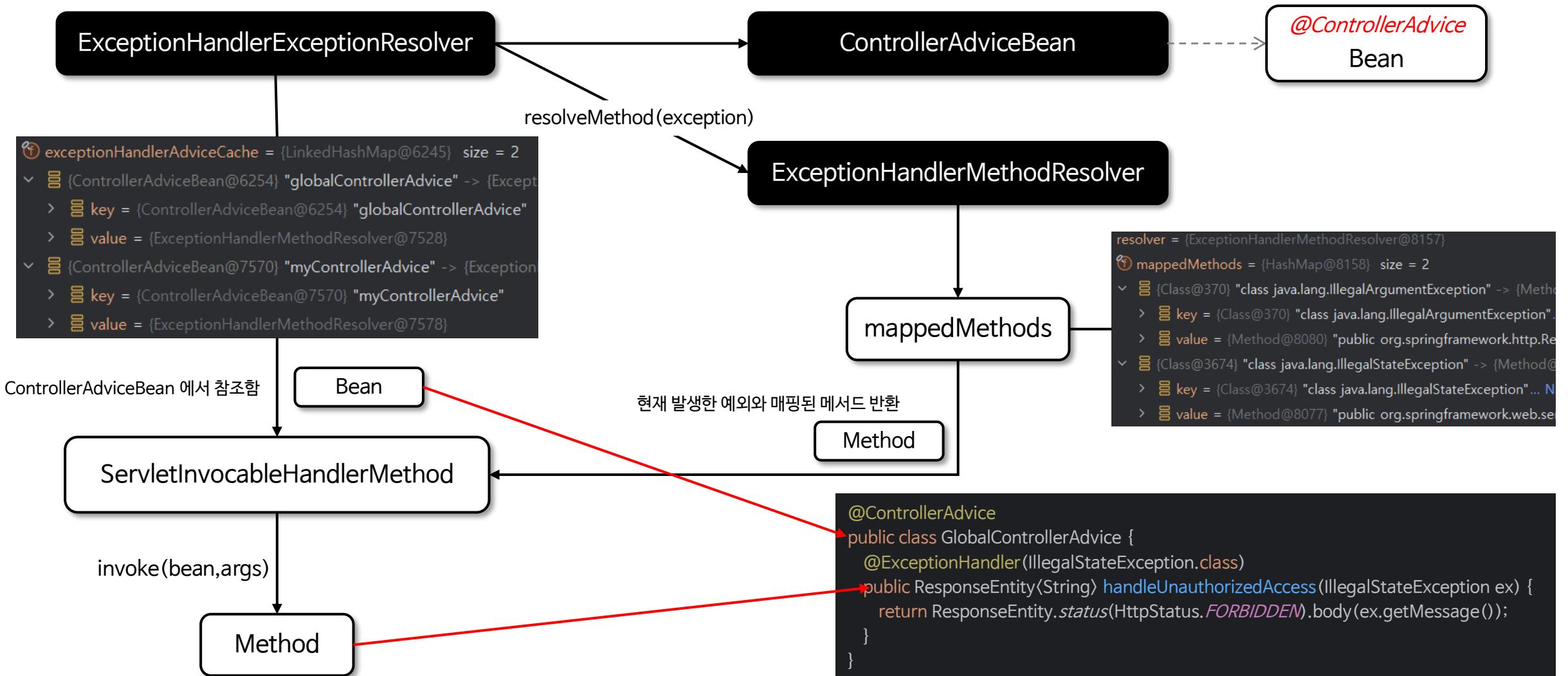
✓ 초기화 구성

- ExceptionHandlerExceptionResolver 클래스는 초기화 시 @ControllerAdvice 대상 컨트롤러를 탐색하고 ControllerAdviceBean 객체와 ExceptionHandlerMethodResolver 객체를 매핑한다



✓ @ControllerAdvice 예외 처리 흐름도

- exceptionHandlerAdviceCache에서 매핑된 ControllerAdviceBean과 ExceptionHandlerMethodResolver를 참조해서 예외 메서드를 호출하는 구조로 되어 있다





스프링 웹 MVC 완전 정복

✓ Multipart

1. 개요
2. 파일 업로드
3. @RequestPart

개요

<https://github.com/onjsdnjs/spring-mvc-master/tree/Multipart-%ED%8A%B8>

✓ 개요

- Multipart는 일반 텍스트 형식이 아니라 하나의 HTTP 요청/응답 바디 내에서 여러 개의 파트를 나누어 전송하는 형식으로서 파일 업로드나 여러 데이터 파트(텍스트 파트, 바이너리 파트 등)를 함께 전송해야 할 때 주로 사용한다
- 웹 브라우저에서 <form> 태그에 enctype="multipart/form-data"를 지정하고 파일 업로드를 수행하면 'multipart/form-data' 형식의 HTTP 요청이 전송 된다

✓ HTTP 요청 및 Raw 바디 예시

```
<form action="/upload" method="post" enctype="multipart/form-data">
    <label for="username">이름 (username)</label>
    <input type="text" id="username" name="username" placeholder="이름을 입력하세요" />
    <label for="file">파일 (file)</label>
    <input type="file" id="file" name="file" multiple />
    <button type="submit">업로드</button>
</form>
```

```
public String getResource (HttpServletRequest request) {
    ByteArrayOutputStream buffer = new ByteArrayOutputStream();
    try (InputStream in = request.getInputStream()) {
        byte[] temp = new byte[1024];
        int read;
        while (read = in.read(temp)) != -1) {
            buffer.write(temp, 0, read);
        }
    }
    byte[] requestData = buffer.toByteArray();
    String rawRequestBody = new String(requestData);
    System.out.println(rawRequestBody);
}
```

- 스프링의 Multipart 처리 비활성화

```
spring.servlet.multipart.enabled=false
```

```
POST /upload HTTP/1.1
Host: localhost
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryABC123
Content-Length: 12345

-----WebKitFormBoundaryABC123
Content-Disposition: form-data; name="username"

leaven
-----WebKitFormBoundaryABC123
Content-Disposition: form-data; name="file"; filename="profile.jpg"
Content-Type: image/jpeg
```

```
ÿØÿÙ ..JFIF.... (--- 바이너리 데이터 (JPEG 파일 내용)
...중략...
-----WebKitFormBoundaryABC123--
```

- WebKitFormBoundaryABC123 : 각 파트(part)를 구분하는 구분선(boundary)
- Content-Disposition : 해당 파트의 종류와 속성(이름, 파일명 등)을 지정
- filename="profile.jpg" 와 Content-Type: image/jpeg : 파일 파트임을 나타내는 정보
- username, leaven : 일반 Form 필드(텍스트 데이터)

✓ 파트 구분

- 요청 바디를 구체적으로 나눠 보면, 크게 3개의 구간(2개의 파트 + 마지막 boundary 종료선)으로 이루어진다

1. 텍스트(Form 필드) 파트

-----WebKitFormBoundaryABC123 Content-Disposition: form-data; name="username" alice

- name="username" : 필드 이름(username)

- 실제 데이터: alice

- 이 부분은 별도의 파일이 아닌 일반 폼 필드 텍스트

2. 파일 파트

-----WebKitFormBoundaryABC123 Content-Disposition: form-data; name="file";

filename="profile.jpg" Content-Type: image/jpeg ÿØÿ..JFIF....<바이너리 데이터>...

- name="file" : 업로드되는 파일 필드 이름

- filename="profile.jpg" : 업로드 파일의 원본 파일명

- Content-Type: image/jpeg : MIME 타입(이미지)

- 이후 줄부터는 실제 바이너리 파일 데이터가 들어 있음

POST /upload HTTP/1.1

Host: localhost

Content-Type: multipart/form-data; boundary=-----WebKitFormBoundaryABC123

Content-Length: 12345

-----WebKitFormBoundaryABC123

Content-Disposition: form-data; name="username"

leaven

-----WebKitFormBoundaryABC123

Content-Disposition: form-data; name="file"; filename="profile.jpg"

Content-Type: image/jpeg

ÿØÿ..JFIF.... <--- 바이너리 데이터 (JPEG 파일 내용)

...중략...

-----WebKitFormBoundaryABC123--

3. 종료선 (boundary 마무리)

-----WebKitFormBoundaryABC123--

- 마지막 구분선 뒤에 --가 붙어 multipart 데이터가 종료됨을 알림

✓ 서블릿의 HttpServletRequest로 파일 처리

```
public class UploadServlet extends HttpServlet {  
    private static final String UPLOAD_DIR = "/path/to/upload";  
    @Override  
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
        String username = request.getParameter("username");  
        Part filePart = request.getPart("file");  
        String fileName = filePart.getSubmittedFileName(); // 일반 폼 필드와 Part 객체를 구분하여 수동으로 처리해야 하는 불편함이 있다  
        long fileSize = filePart.getSize(); // 또한 직접 InputStream 을 얻어와서 파일을 복사하는 식으로 처리해야 한다  
        String contentType = filePart.getContentType();  
  
        File uploadDir = new File(UPLOAD_DIR);  
        String filePath = UPLOAD_DIR + File.separator + fileName;  
  
        try (InputStream inputStream = filePart.getInputStream()) {  
            Files.copy(inputStream, Paths.get(filePath));  
        }  
    }  
}
```

✓ 스프링의 MultipartFile로 파일 처리

```
@RestController  
public class FileUploadController {  
    private static final String UPLOAD_DIR = "/path/to/upload";  
    @PostMapping("/upload")  
    public String handleFileUpload(@RequestParam("username") String username, @RequestParam("file") MultipartFile file) throws IOException {  
        String originalFilename = file.getOriginalFilename();  
        long fileSize = file.getSize();  
        String contentType = file.getContentType();  
        // 일반 폼 필드와 Part 객체를 구분할 필요가 없으며 Part 부분을 MultipartFile 이 추상화 해서 처리해 준다  
        // InputStream 을 참조할 필요가 없이 파일을 바로 복사한다. 쉽게 말해 MultipartFile 만 잘 사용하면 된다  
        File uploadDir = new File(UPLOAD_DIR);  
        File dest = new File(uploadDir, originalFilename);  
        file.transferTo(dest);  
  
        return "list";  
    }  
}
```

```
① files = {ArrayList@9496} size = 4 ... View  
  ↘ ② 0 = {StandardMultipartHttpServletRequest$StandardMultipartFile@9500}  
    > ③ part = {ApplicationPart@9504}  
    > ④ filename = "example.xlsx"  
  ↘ ⑤ 1 = {StandardMultipartHttpServletRequest$StandardMultipartFile@9501}  
    > ⑥ part = {ApplicationPart@9506}  
    > ⑦ filename = "강의안.pdf"  
  ↘ ⑧ 2 = {StandardMultipartHttpServletRequest$StandardMultipartFile@9502}  
    > ⑨ part = {ApplicationPart@9508}  
    > ⑩ filename = "정산내역.xlsx"  
  ↘ ⑪ 3 = {StandardMultipartHttpServletRequest$StandardMultipartFile@9503}  
    > ⑫ part = {ApplicationPart@9510}  
    > ⑬ filename = "spring-security-project.png"
```

✓ MultiPart 지원 도구

- Spring(특히 Spring Boot 3.x 기준)에서 Multipart(파일 업로드)를 처리하기 위한 도구로서 MultipartResolver, MultipartFile, MultipartHttpServletRequest 그리고 이를 사용하는 Controller/Service 구조가 유기적으로 연결되어 있다
- **MultipartAutoConfiguration**
 - Spring Boot에서 multipart/form-data 요청 처리를 자동으로 구성해주는 설정 클래스로서 추가적인 설정 없이도 @RequestParam("file") MultipartFile file 을 사용하면 자동으로 멀티파트 요청을 처리하도록 구성된다
- **MultipartHttpServletRequest**
 - HttpServletRequest를 상속(혹은 래핑)하여 멀티파트 폼 데이터를 처리할 수 있는 추가 메서드를 제공하는 인터페이스로서 기본 구현체로 StandardMultipartHttpServletRequest 클래스가 제공된다
- **MultipartResolver**
 - multipart/form-data 요청을 해석하여 MultipartHttpServletRequest 를 만들어주는 인터페이스로서 기본 구현체로 StandardServletMultipartResolver 가 제공된다
- **MultipartFile**
 - 업로드된 파일을 다루기 위한 인터페이스로서 getName(), getOriginalFilename(), getSize(), transferTo(File dest) 와 같은 API 가 있다
- **MultipartProperties**
 - Spring Boot에서 멀티파트 설정을 위한 구성 설정 클래스이다
- **@RequestPart**
 - multipart 요청의 특정 파트를 직접 바인딩하기 위한 어노테이션으로서 @RequestParam 보다 좀 더 확장된 기능을 제공한다

파일 업로드

<https://github.com/onjsdnjs/spring-mvc-master/tree/파일-업로드>

✓ MultipartFile

- 스프링에서 파일 업로드를 처리하기 위해 제공하는 인터페이스로서 HTML 폼에서 enctype="multipart/form-data"로 전송된 파일 데이터를 서버 측에서 쉽게 다룰 수 있도록 해 준다
- 단일 파일뿐 아니라 여러 파일(List<MultipartFile> 형태 등) 업로드 또한 지원한다

① ↗ MultipartFile	
Ⓜ️ ↗ isEmpty()	boolean 업로드된 파일이 비어있는지 여부를 반환 한다
Ⓜ️ ↗ getOriginalFilename()	String? 클라이언트 파일 시스템에서의 원본 파일명을 반환 한다. 파일이 선택되지 않았다면 빈 문자열을, 파일명이 정의되지 않았거나 사용할 수 없는 경우 null 을 반환 한다
Ⓜ️ ↗ getBytes()	byte[] 파일의 내용을 바이트 배열로 반환 한다. 비어있다면 빈 배열을 반환 한다
Ⓜ️ ↗ getInputStream()	InputStream 파일 내용을 읽을 수 있는 InputStream 을 반환 한다
Ⓜ️ ↗ transferTo(Path)	void 받은 파일을 지정된 목적지 파일로 전송한다. 목적지 파일이 이미 존재하면 먼저 삭제되며 한 번 이동된 파일은 재전송할 수 없으므로 이 메서드는 한 번만 호출해야 한다
Ⓜ️ ↗ getResource()	Resource MultipartFile 을 Resource 형태로 반환 한다
Ⓜ️ ↗ transferTo(File)	void 받은 파일을 지정된 목적지 파일로 전송한다. 인자로 File 객체를 받는다
Ⓜ️ ↗ getContentType()	String? 파일의 Content-Type 을 반환 한다
Ⓜ️ ↗ getName()	String 멀티파트 Form 에서 해당 파라미터 이름을 반환 한다
Ⓜ️ ↗ getSize()	long 파일의 크기를 바이트 단위로 반환하며 비어있다면 0 을 반환 한다

✓ 기본 구현

```
@PostMapping("/uploadMultiple")
public String upload(@RequestParam("files") MultipartFile file) {
    String originalFilename = file.getOriginalFilename();
    ...
    return "list";
}
```

✓ 다중 구현

```
@PostMapping("/uploadMultiple")
public String upload(@RequestParam("files") List<MultipartFile> files) {
    for (MultipartFile file : files) {
        // 파일 업로드 처리..
    }
    return "list";
}
```

✓ 속성 설정

- MultipartProperties 는 Multipart(파일 업로드)에 대한 주요 설정 값들을 관리하며 Spring Boot 에서 파일 업로드와 관련된 동작을 손쉽게 제어할 수 있게 해 준다.
- application.properties(yml) 에서 spring.servlet.multipart.* 를 중심으로 설정하며 이를 통해 파일 크기 제한, 임시 저장 경로, 요청 크기 제한 등 다양한 옵션을 간단하게 지정할 수 있다

1. enabled

- spring.servlet.multipart.enabled (기본값: true)
- 파일 업로드(멀티파트)를 지원할지 여부를 설정하며 true 로 설정하면 멀티파트 업로드 기능이 활성화되며, false 로 설정하면 비활성화 된다

2. maxFileSize

- spring.servlet.multipart.max-file-size (기본값: 1MB)
- 업로드 가능한 단일 파일의 최대 사이즈를 제한하며 10MB, 1GB 등과 같은 식으로 원하는 크기를 문자열로 설정할 수 있다

3. maxRequestSize

- spring.servlet.multipart.max-request-size (기본값: 10MB)
- 멀티파트 요청 전체의 최대 사이즈를 제한하며 여러 개의 파일을 동시에 업로드하는 경우, 전체 파일 사이즈 합이 이 값을 넘지 않도록 설정합니다.

4. location

- spring.servlet.multipart.location
- 파일 업로드 중 임시로 파일을 저장할 경로를 지정하며 지정하지 않으면, 서블릿 스펙에 따라 임시 폴더(예: /tmp)가 기본값으로 사용됩니다.

5. resolve-lazily (기본값: false)

- 요청이 들어온 즉시 멀티파트 데이터를 해석하고 저장할 것인지 컨트롤러나 필터/인터셉터에서 MultipartFile 이나 파라미터에 접근할 때 비로소 파일을 해석하고 저장할 것인지 설정

```
spring.servlet.multipart.enabled=true  
spring.servlet.multipart.location=/tmp/upload  
spring.servlet.multipart.max-file-size=10MB  
spring.servlet.multipart.max-request-size=30MB  
spring.servlet.multipart.resolve-lazily=true
```

✓ MultipartFile - @RequestParam 사용

```
@Controller  
public class RequestParamFileUploadController {  
    private final FileUploadService uploadService;  
  
    public RequestParamFileUploadController(FileUploadService uploadService) {  
        this.uploadService = uploadService;  
    }  
  
    @PostMapping("/upload")  
    public String uploadFile(@RequestParam("file") MultipartFile file) {  
        FileUploadInfo savedInfo = uploadService.storeFile(file);  
        return "user/list"  
    }  
}
```

```
<!DOCTYPE html>  
<html>  
<body>  
<form action="/upload" method="post" enctype="multipart/form-data">  
    <p><label>파일 선택: <input type="file" name="file"></label></p>  
    <button type="submit">업로드</button>  
</form>  
</body>  
</html>
```

```
@Service  
public class FileUploadService {  
  
    private final FileUploadInfoRepository fileUploadInfoRepository;  
  
    @Value("${file.upload.dir:/data/uploaded}")  
    private String uploadDir;  
  
    public FileUploadService(FileUploadInfoRepository fileUploadInfoRepository) {  
        this.fileUploadInfoRepository = fileUploadInfoRepository;  
    }  
  
    public FileUploadInfo storeFile(MultipartFile multipartFile) throws IOException {  
        String originalFilename = multipartFile.getOriginalFilename(); // 원본파일명  
        int dotIndex = originalFilename.lastIndexOf('.');  
        if (dotIndex > 0) extension = originalFilename.substring(dotIndex); // 확장자  
        String storedFilename = UUID.randomUUID().toString() + extension; // 실제 업로드 파일명  
        String filePath = uploadDir + File.separator + storedFilename; // 파일 저장 폴 경로  
  
        File destFile = new File(filePath);  
        multipartFile.transferTo(destFile);  
  
        FileUploadInfo uploadInfo = new FileUploadInfo(originalFilename, storedFilename,  
            multipartFile.getSize(), //파일 크기  
            multipartFile.getContentType(), //컨텐트 타입  
            filePath  
        );  
        return fileUploadInfoRepository.save(uploadInfo);  
    }  
}
```

✓ MultipartFile - @ModelAttribute 사용

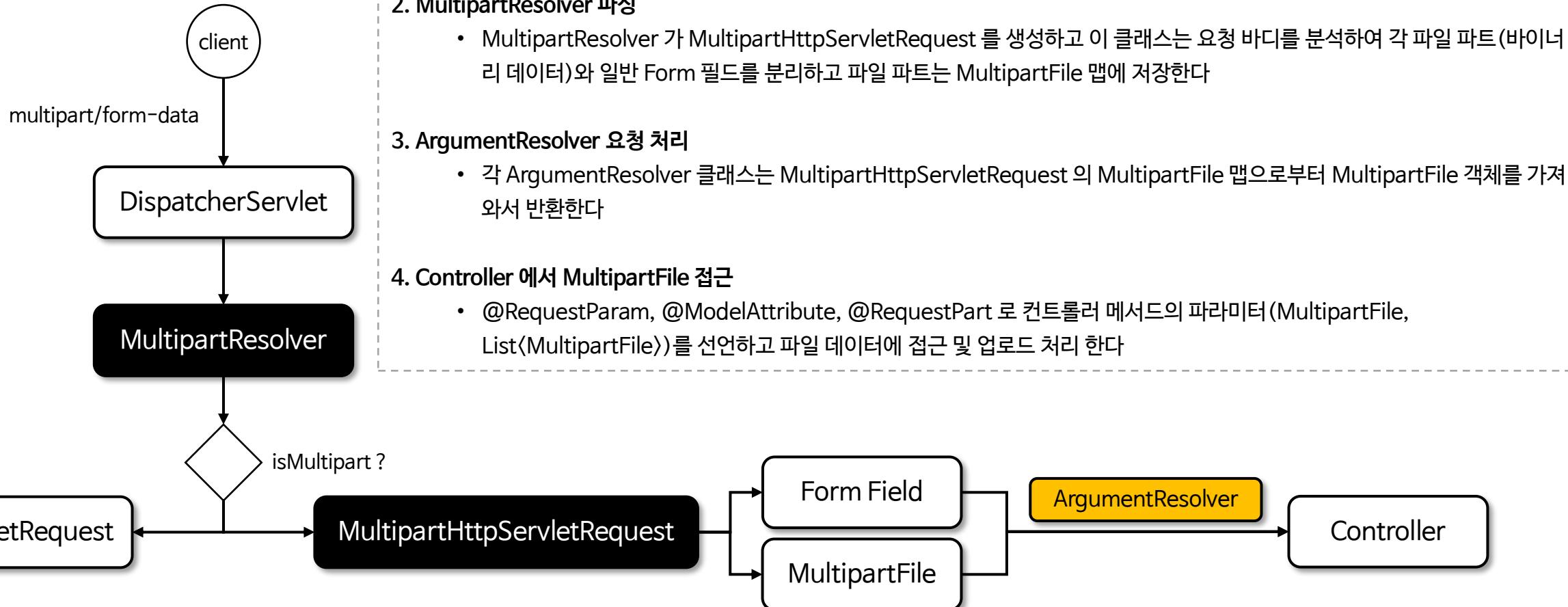
- 게시글과 파일업로드를 동시에 처리해야 하는 경우 객체를 구성해 다수의 폼 필드와 함께 파일을 받을 때 @ModelAttribute 방식으로 구현 할 수 있다

```
@Controller  
public class ModelAttributeFileUploadController {  
    private final FileUploadService uploadService;  
  
    public ModelAttributeFileUploadController(FileUploadService uploadService) {  
        this.uploadService = uploadService;  
    }  
  
    @PostMapping("/upload")  
    public String uploadFile(@ModelAttribute UploadForm form) {  
        MultipartFile file = form.getFile(); // DTO 안에 있는 MultipartFile 추출  
        FileUploadInfo savedInfo = fileUploadService.storeFile(file);  
        return "user/list"  
    }  
}
```

```
<!DOCTYPE html>  
<html>  
<body>  
<form action="/upload" method="post" enctype="multipart/form-data">  
    <p><label>제목: <input type="text" name="title"></label></p>  
    <p><label>파일 선택: <input type="file" name="file"></label></p>  
    <button type="submit">업로드</button>  
</form>  
</body>  
</html>
```

```
@Data  
public class UploadForm {  
    private String title;  
    private MultipartFile file;  
}
```

✓ Multipart Process



@RequestPart

<https://github.com/onjsdnjs/spring-mvc-master/tree/@RequestPart>

✓ 개요

- @RequestPart 는 multipart 요청에서 특정 part(부분)를 매핑하여 컨트롤러의 파라미터로 바인딩하는 어노테이션으로서 JSON 요청 처리 + 파일 업로드를 동시에 처리 가능하다
- @RequestParam 과 다르게 파일뿐만 아니라 JSON 객체도 받을 수 있다

✓ 기본 구현

```
@PostMapping("/uploadPost")
public ResponseEntity<String> uploadPost(
    @RequestPart("post") PostInfo postInfo,
    @RequestPart("file") MultipartFile file) {
}
```

POST /upload HTTP/1.1
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary12345

-----WebKitFormBoundary12345
Content-Disposition: form-data; name="post"
Content-Type: application/json

{"title": "My Post", "description": "This is a test post"}

// JSON 형태의 요청

-----WebKitFormBoundary12345
Content-Disposition: form-data; name="file"; filename="image.jpg"
Content-Type: image/jpeg

(binary data of the image file)
-----WebKitFormBoundary12345—

// Multipart 파일 요청

✓ BoardController

```
@PostMapping("/upload")
public ResponseEntity<PostInfo> uploadFile(@RequestPart("post") PostInfo post, @RequestPart(value = "files", required = false) List<MultipartFile> files) throws
IOException {
    PostInfo postInfo = boardService.storeFile(post, files);
    return ResponseEntity.ok().body(postInfo);
}
```

✓ BoardService

```
public PostInfo storeFile(PostInfo postInfo, List<MultipartFile> files) throws IOException {
    postRepository.save(postInfo);
    if (files != null && !files.isEmpty()) {
        for (MultipartFile file : files) {
            FileUploadInfo uploadedFile = fileUpload(file);
            postInfo.getFiles().add(uploadedFile);
        }
    }
    return postInfo;
}

private FileUploadInfo fileUpload(MultipartFile multipartFile) throws IOException {
    String originalFilename = multipartFile.getOriginalFilename(); // 원본파일명
    int dotIndex = originalFilename.lastIndexOf('.');
    // 생략...
```

✓ uploadForm.html

```
<form id="uploadForm">
    <label>제목:</label>
    <input type="text" id="title" required>
    <label>내용:</label>
    <input type="text" id="content" required>
    <label>사용자:</label>
    <input type="text" id="userId" required>
    <label>댓글:</label>
    <input type="text" id="comment">
    <label>파일 선택:</label>
    <div class="file-input">
        <input type="file" id="files" multiple>
    </div>
    <button type="submit">업로드</button>
</form>
```

```
<script>
    document.addEventListener('DOMContentLoaded', function() {
        const uploadForm = document.getElementById('uploadForm');

        uploadForm.addEventListener('submit', function(event) {
            const formData = new FormData();
            const titleInput = document.getElementById('title');
            const contentInput = document.getElementById('content');
            const userIdInput = document.getElementById('userId');
            const commentInput = document.getElementById('comment');
            const fileInput = document.getElementById('files');

            for (let i = 0; i < fileInput.files.length; i++) {
                formData.append("files", fileInput.files[i]);
            }

            const metaData = {
                title: titleInput.value,
                content: contentInput.value,
                userId: userIdInput.value,
                comment: commentInput.value
            };

            formData.append("post", new Blob([JSON.stringify(metaData)], {
                type: "application/json"
            }));

            fetch("/upload", {
                method: 'POST',
                body: formData
            })
                .then(response => response.json())
                .then(result => {
                    alert("업로드 성공: " + result.title);
                    window.location.href = "/listForm";
                })
            );
        });
    });
</script>
```

요청 방식	사용 가능 애너테이션
application/json	@RequestBody , @RequestPart
application/x-www-form-urlencoded	@RequestParam, @ModelAttribute
multipart/form-data	@RequestParam, @ModelAttribute, @RequestPart



스프링 웹 MVC 완전 정복

✓ Rest Clients

1. RestClient
2. HTTP Interface

RestClient

<https://github.com/onjsdnjs/spring-mvc-master/tree/RestClient>

✓ 개요

- RestClient는 Spring 6에서 새롭게 도입된 동기식 HTTP 클라이언트로서, 기존의 RestTemplate을 대체하거나 보완할 수 있는 보다 모던한 API를 제공한다
- 내부적으로 다양한 HTTP 클라이언트 라이브러리 위에 추상화를 제공하며 개발자가 손쉽게 HTTP 요청과 응답을 다룰 수 있도록 지원한다

✓ 특징

1. 메서드 체이닝 방식의 API

- get(), post(), put() 등 HTTP 메서드를 코드상에서 직관적으로 체이닝할 수 있으며 retrieve(), exchange() 등을 통해 설정 → 요청 → 응답 흐름을 명확히 표현할 수 있다

2. 동기식 처리

- 내부 동작이 동기식이므로 블로킹 모델을 따른다

3. Builder 기반 설정

- RestClient.builder()에서 baseUrl, 기본 헤더, 인터셉터, requestFactory 등을 간편하게 설정할 수 있으며 한 번 만들어진 RestClient는 여러 스레드에서 안전하게 사용할 수 있다

4. 다양한 HTTP 라이브러리 연동

- 자동으로 Apache HttpClient, Jetty HttpClient, java.net.http 등을 감지하여 활용하며 직접 requestFactory(..)를 지정하여 원하는 라이브러리를 명시적으로 선택할 수도 있다

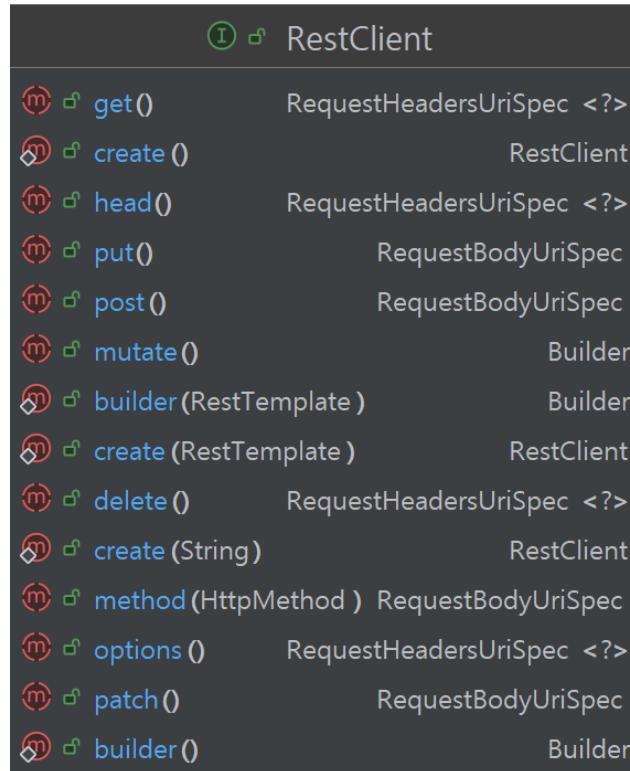
5. 편리한 메시지 변환

- body(Object), body(ParameterizedTypeReference) 등을 제공해 제네릭 구조를 포함한 다양한 타입 변환을 지원하며 accept(), contentType() 등으로 헤더를 설정할 수 있다

6. 유연한 오류 처리

- 4xx, 5xx 상태 코드를 받으면 기본적으로 RestClientException 계열 예외를 던지지만, onStatus()를 사용해 특정 상태 코드에 대한 맞춤 예외 처리가 가능합니다.
- exchange() 메서드를 사용하면 요청과 응답을 더 세밀하게 다루면서 상태 코드별로 직접 처리를 구성할 수 있습니다.

✓ RestClient 구조 및 생성



```
// 기본 RestClient 인스턴스 생성
RestClient defaultClient = RestClient.create();

// 커스텀 구성으로 RestClient 빌드
RestClient customClient = RestClient.builder()
    // 사용할 HTTP 라이브러리 지정
    .requestFactory(new HttpComponentsClientHttpRequestFactory())
    // 메시지 컨버터 추가
    .messageConverters(converters -> converters.add(new MyCustomMessageConverter()))
    // 기본 URL 지정
    .baseUrl("https://example.com")
    // 기본 URI 변수 지정
    .defaultUriVariables(Map.of("variable", "foo"))
    // 기본 요청 헤더
    .defaultHeader("My-Header", "Foo")
    // 기본 쿠키
    .defaultCookie("My-Cookie", "Bar")
    // 요청 인터셉터
    .requestInterceptor(myCustomInterceptor)
    // 요청 초기화 로직
    .requestInitializer(myCustomInitializer)
    .build();
```

implementation 'org.apache.httpcomponents.client5:httpclient5:5.4.2'

- 정적 메서드 사용해서 생성
 - RestClient.create()
- 빌더 패턴 사용해서 생성
 - RestClient.builder()

구현체	특징	추천 사용 환경
HttpComponentsClientHttpRequestFactory	Apache HttpClient 기반	Spring Boot 설정, 동기 요청
SimpleClientHttpRequestFactory	JDK 기본 HTTP 요청	간단한 HTTP 요청
OkHttp3ClientHttpRequestFactory	OkHttp 기반	API 클라이언트, 대규모 트래픽 처리
Netty4ClientHttpRequestFactory	WebFlux(Netty) 기반	비동기 API 클라이언트 (WebFlux)
JettyClientHttpRequestFactory	Jetty 기반	Jetty 기반 애플리케이션

✓ HTTP 메서드 및 Request URL 설정

```
// 1. RestClient 빌더로 baseUrl 설정
```

```
RestClient restClient = RestClient.builder()  
    .requestFactory(new HttpComponentsClientHttpRequestFactory())  
    .baseUrl("http://localhost:8080/api") // 기본 URL  
    .build();
```

```
// 2. URI 템플릿 변수를 활용한 GET 요청
```

```
// 최종 호출 경로: http://localhost:8080/api/users/42  
int userId = 42;  
restClient.get()  
    .uri("/users/{id}", userId) // baseUrl + "/users/42"  
    .retrieve();
```

```
// 3. 람다를 사용해 UriBuilder를 세밀하게 조작하기
```

```
// 최종 호출 경로 예시: http://localhost:8080/api/search?keyword=Spring&sort=desc&page=2
```

```
String keyword = "spring";  
String sortOrder = "desc";  
int page = 2;  
restClient.get()  
    .uri(uriBuilder ->  
        uriBuilder.path("/search")  
        .queryParam("keyword", keyword)  
        .queryParam("sort", sortOrder)  
        .queryParam("page", page)  
        .build()  
    )  
    .retrieve()
```

- 1) `restClient.get()`, `restClient.post()`, `restClient.put()` 등으로 선언한다
- 2) RestClient 빌더에서 이미 baseUrl을 지정했다면 필요한 부분만 추가로 설정해도 된다
- 3) URI 템플릿 변수를 활용하여 동적으로 경로를 구성하거나, 함수형 인자를 사용해 query parameter 등을 세밀하게 제어할 수도 있다

✓ Request Headers 설정

// 1. RestClient 빌더로 baseUrl 설정

```
RestClient restClient = RestClient.builder()
    .requestFactory(new HttpComponentsClientHttpRequestFactory())
    .baseUrl("http://localhost:8080/api") // 기본 URL
    .build();
```

// 2) header(String, String) 로 단순 헤더 추가

```
ResponseEntity<String> response1 = restClient.get()
    .uri("/headers/test")
    .header("Authorization", "Bearer my-secret-token") // 단일 헤더 추가
    .header("X-Custom-Header", "customValue") // 또 다른 헤더 추가
    .accept(MediaType.APPLICATION_JSON) // 표준 Accept 헤더
    .retrieve()
    .toEntity(String.class);
```

// 3) headers(Consumer<HttpHeaders>) 를 통해 여러 헤더를 체이닝으로 설정

```
ResponseEntity<String> response2 = restClient.get()
    .uri("/headers/test2")
    .headers(httpHeaders -> {
        httpHeaders.set("X-Api-Key", "abcd-1234");
        httpHeaders.add("Another-Header", "hello"); // add는 여러 값 누적
    })
    // 편의 메서드로 표준 헤더 설정
    .accept(MediaType.APPLICATION_XML) // Accept: application/xml
    .acceptCharset(StandardCharsets.UTF_8) // Accept-Charset: utf-8
    .retrieve()
    .toEntity(String.class);
```

- header(String, String) 또는 headers(Consumer<HttpHeaders>)를 사용해 원하는 헤더를 추가하거나, accept(MediaType), acceptCharset(Charset) 등 편의 메서드로 표준 헤더를 간단히 지정할 수 있다

✓ Request Body 설정 – JSON 바디 전송

```
User user = restClient.get()  
    .uri("http://localhost:8080/users/{id}", 42)  
    .accept(MediaType.APPLICATION_JSON) // Accept: application/json  
    .retrieve()  
    .body(User.class);  
        // Jackson 등을 통해 JSON → User 역직렬화
```

- GET 요청 시 JSON 응답 받기

```
User user = new User("springmvc", 20);  
  
// POST: /users/new, 요청 바디로 user 객체 전달  
ResponseEntity<Void> response = restClient.post()  
    .uri(" http://localhost:8080/users/new")  
    .contentType(MediaType.APPLICATION_JSON)  
    .body(user)  
    .retrieve()  
    .toBodilessEntity(); // 바디 없는 응답(ResponseEntity<Void>)으로 변환
```

- POST 요청 시 JSON 바디 전송

```
List<User> users = restClient.get()  
    .uri(" http://localhost:8080/users")  
    .accept(MediaType.APPLICATION_JSON)  
    .retrieve()  
    .body(new ParameterizedTypeReference<List<User>>() {});
```

- 제네릭 타입 처리 (ParameterizedTypeReference)

✓ Exchange

- `exchange()`는 `retrieve()`와 달리 HTTP 요청(Request)과 응답(Response)을 모두 직접 제어할 수 있게 해 주는 기능으로서 람다 형태의 콜백 함수를 인자로 받는다

✓ 기본 구조

```
(exchangeRequest, exchangeResponse) -> {  
    // 여기서 request, response를 직접 다룰 수 있음  
    return ... // 필요한 처리를 한 뒤 원하는 타입으로 리턴  
};
```

- **exchangeRequest**
 - 실제로 전송될 요청 정보를 다룬다
- **exchangeResponse**
 - 서버로부터 받은 응답(상태 코드, 헤더, 바디 등)에 직접 접근할 수 있다

```
User user = restClient.get()  
.uri("/users/{id}", userId)  
.accept(MediaType.APPLICATION_JSON)  
.exchange((request, response) -> {  
    HttpStatusCode status = response.getStatusCode(); // 상태 코드 확인  
    if (status.is4xxClientError() || status.is5xxServerError()) {  
        throw new RuntimeException("오류 상태 코드: " + status.value()); // 예: 4xx/5xx일 경우 커스텀 예외 던지기  
    }  
    try (InputStream bodyStream = response.getBody()) { // 정상 응답 바디를 직접 파싱  
        if (bodyStream == null) {  
            return null;  
        }  
        return objectMapper.readValue(bodyStream, User.class); // Jackson을 이용하여 JSON -> User 객체로 변환  
    }  
});
```

오류 처리

<https://github.com/onjsdnjs/spring-mvc-master/tree/RestClient-오류처리>

✓ 개요

- RestClient에서 오류 처리는 기본적으로 retrieve() 또는 exchange() 메서드에서 수행할 수 있으며 HTTP 4xx 및 5xx 오류 발생 시 자동으로 예외(RestClientException)를 던진다
- 기본 동작을 커스터마이징 하려면 onStatus() 또는 exchange()를 활용하여 오류 상태 코드를 직접 처리해야 한다

✓ 자동 예외 처리

```
try {  
    String response = restClient.get()  
        .uri("https://example.com/api/not-found") // 존재하지 않는 URL  
        .retrieve()  
        .body(String.class);  
} catch (RestClientException e) {  
    System.err.println("RestClientException 발생: " + e.getMessage());  
}
```

- retrieve()를 사용할 경우 자동으로 4xx, 5xx 응답 시 예외를 던짐
- try-catch 블록에서 RestClientException을 잡아 예외를 처리

✓ onStatus()를 이용한 사용자 정의 오류 처리

```
try {  
    String response = restClient.get()  
        .uri("https://example.com/api/bad-request")  
        .retrieve()  
        .onStatus(HttpStatusCode::is4xxClientError, (request, response) -> {  
            throw new RuntimeException("클라이언트 오류 발생: " + response.getStatusCode());  
        })  
        .onStatus(HttpStatusCode::is5xxServerError, (request, response) -> {  
            throw new RuntimeException("서버 오류 발생: " + response.getStatusCode());  
        })  
        .body(String.class);  
} catch (Exception e) {  
    System.err.println("오류 처리: " + e.getMessage());  
}
```

- onStatus()를 이용하여 특정 상태 코드에 대한 맞춤형 예외 처리 가능
- 4xx(클라이언트 오류) 또는 5xx(서버 오류) 상태 코드에 따라 커스텀 예외 발생

✓ onStatus() 를 사용한 특정 오류 매팅

```
try {
    String result = restClient.get()
        .uri("https://example.com/api/user/invalid")
        .retrieve()
        .onStatus(status -> status.value() == 404, (request, response) -> {
            throw new UserNotFoundException(response.getStatusCode());
        })
        .body(String.class);
} catch (UserNotFoundException e) {
    System.err.println("사용자 예외 처리: " + e.getMessage());
} catch (RestClientException e) {
    System.err.println("기타 예외 발생: " + e.getMessage());
}

// 커스텀 예외 정의
class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

- 일부 상태 코드에 대해 명확한 예외 클래스를 매팅할 수 있다
- onStatus() 내부에서 특정 HTTP 상태 코드(예: 404)에 대해 사용자 정의 예외를 던질 수 있음
- 다른 상태 코드는 기본적으로 RestClientException 이 발생하도록 유지

✓ exchange()를 이용한 세부 오류 처리

```
try {
    String result = restClient.get()
        .uri("https://example.com/api/unknown-endpoint") // 존재하지 않는 엔드포인트
        .exchange((request, response) -> {
            HttpStatusCode status = response.getStatusCode();

            // 4xx 오류 발생 시
            if (status.is4xxClientError()) {
                throw new RuntimeException("잘못된 요청 (4xx) : " + status.value());
            }

            // 5xx 오류 발생 시
            if (status.is5xxServerError()) {
                throw new RuntimeException("서버 오류 (5xx) : " + status.value());
            }

            // 정상 응답 바디 변환
            try (InputStream bodyStream = response.getBody()) {
                return new String(bodyStream.readAllBytes()); // 바디 내용을 문자열로 변환
            } catch (Exception e) {
                throw new RuntimeException("응답 바디 읽기 오류", e);
            }
        });

    } catch (RestClientException e) {
        System.err.println("오류 발생: " + e.getMessage());
    }
}
```

- exchange()를 사용하면 response.getStatusCode()를 통해 모든 HTTP 응답 상태 코드를 직접 확인할 수 있음
- response.getBody()로 직접 응답 바디를 읽어올 수 있음
- 원하는 방식으로 오류를 처리할 수 있으며, 커스텀 예외를 던질 수도 있음

HTTP Interface

<https://github.com/onjsdnjs/spring-mvc-master/tree/HTTP-Interface>

✓ 개요

- HTTP Interface 는 클래스가 아닌 인터페이스를 사용하여 REST API 호출을 간결하고 선언적으로 정의할 수 있는 기능이다
- @HttpExchange 어노테이션을 사용하여 API의 경로, 요청 메서드, 요청 매개변수 등을 선언적으로 구성하며, 스프링의 RestClient 또는 WebClient 를 기반으로 구현된다

✓ 특징

1. @HttpExchange 어노테이션 사용

- HTTP 요청의 기본 경로, HTTP 메서드(GET, POST 등), 헤더 등을 정의하며 각 메서드에서 필요한 추가 정보(예: @PathVariable, @RequestBody)를 선언적으로 추가할 수 있다

2. 인터페이스 중심의 선언적 접근

- HTTP 요청을 Java 인터페이스로 정의하며, 구현체를 별도로 작성할 필요 없이 스프링이 이를 자동으로 생성한다

3. RestClient 기반 통합

- 스프링 부트 3.1 이상부터는 RestClient 를 사용하여 HTTP Interface 와 통합된다

4. 가독성과 재사용성

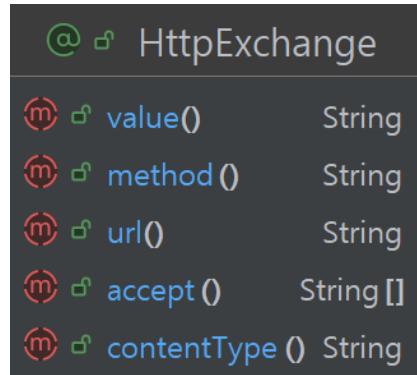
- HTTP 호출 로직을 인터페이스로 분리하여 코드가 간결하고 재사용이 쉬워지며 복잡한 HTTP 요청 처리 코드가 없어지고 테스트도 간편해 진다

5. Spring Integration

- 스프링의 의존성 주입(DI) 및 스프링의 환경설정과 어노테이션을 그대로 활용할 수 있다

✓ @HttpExchange

- HTTP 엔드포인트로 선언하기 위한 어노테이션으로서 이 어노테이션을 사용하면 자바 인터페이스의 메서드를 HTTP 엔드포인트로 지정할 수 있으며 HttpServiceProxyFactory에 전달하면 자동으로 HTTP 요청을 보내는 클라이언트를 만들어 준다
- @GetExchange, @PostExchange, @PutExchange, @PatchExchange, @DeleteExchange 등의 단축 어노테이션을 지원한다



- `url` 과 같은 값을 설정할 때 사용한다
- 사용할 HTTP 메서드(GET, POST 등)를 지정한다
- 요청의 URL을 지정한다. 타입 수준의 선언을 지원하며 공통 URL 경로(base URL)를 설정을 할 수 있고 메서드에서 이를 상속받아 사용할 수 있다
- 응답으로 받을 데이터 타입(Accept 헤더)을 지정한다
- 요청의 Content-Type(요청 데이터의 타입)을 지정한다

✓ HTTP Interface 정의

```
interface RepositoryService {  
    @GetExchange("/repos/{owner}/{repo}")  
    Repository getRepository(@PathVariable String owner, @PathVariable String repo);  
    // 추가적인 HTTP 요청 메서드...  
}
```

```
@HttpExchange("/repos")  
interface RepositoryService {  
    @GetExchange("/{owner}/{repo}")  
    Repository getRepository(@PathVariable String owner, @PathVariable String repo);  
    // 추가적인 HTTP 요청 메서드...  
}
```

Method Parameters & Return values

<https://docs.spring.io/spring-framework/reference/integration/rest-clients.html#rest-http-interface>

✓ 클라이언트용 프록시 생성

1. RestClient 사용

```
RestClient restClient = RestClient.builder().baseUrl("https://api.github.com/").build();
RestClientAdapter adapter = RestClientAdapter.create(restClient);
HttpServiceProxyFactory factory = HttpServiceProxyFactory.builderFor(adapter).build();

RepositoryService service = factory.createClient(RepositoryService.class);
```

- RestClient.builder()를 사용해 GitHub API에 연결할 기본 URL을 설정한다.
- RestClientAdapter.create(restClient)를 통해 RestClient를 어댑터로 변환한다.
- HttpServiceProxyFactory.builderFor(adapter).build()를 사용하여 프록시 팩토리를 생성한다.
- factory.createClient(RepositoryService.class)를 호출하면 RepositoryService 인터페이스를 구현하는 프록시 객체가 생성된다

2. WebClient 사용

```
WebClient webClient = WebClient.builder().baseUrl("https://api.github.com/").build();
WebClientAdapter adapter = WebClientAdapter.create(webClient);
HttpServiceProxyFactory factory = HttpServiceProxyFactory.builderFor(adapter).build();

RepositoryService service = factory.createClient(RepositoryService.class);
```

3. RestTemplate 사용

```
RestTemplate restTemplate = new RestTemplate();
restTemplate.setUriTemplateHandler(new DefaultUriBuilderFactory("https://api.github.com/"));
RestTemplateAdapter adapter = RestTemplateAdapter.create(restTemplate);
HttpServiceProxyFactory factory = HttpServiceProxyFactory.builderFor(adapter).build();

RepositoryService service = factory.createClient(RepositoryService.class);
```

✓ 기본 구현

```
@HttpExchange("/posts")
public interface PostClient {

    @GetExchange("/{id}")
    Post getPostById(@PathVariable Long id);

    @PostExchange
    Post createPost(@RequestBody Post post);
}
```

/posts/{id}
/posts

```
@Configuration
public class ClientConfig {

    @Bean
    public PostClient userClient(RestClient.Builder builder) {
        RestClient restClient = builder.baseUrl("https://jsonplaceholder.typicode.com").build();
        return HttpServiceProxyFactory.builderFor(RestClientAdapter.create(restClient))
            .build()
            .createClient(PostClient.class);
    }
}
```

https://jsonplaceholder.typicode.com

```
@Service
@RequiredArgsConstructor
public class PostService {

    private final PostClient postClient;

    public void execute() {
        Post post = postClient.getPostById(1L);
        Post newPost = new Post(null, "title", "new post", 1L);
        Post createdPost = postClient.createPost(newPost);
    }
}
```

proxy
https://jsonplaceholder.typicode.com/posts/1
https://jsonplaceholder.typicode.com/posts

✓ RestClient vs HTTP Interface

```
RestClient restClient = RestClient.create("https://api.example.com");

User user = restClient.get()
    .uri("/users/{id}", 1)
    .retrieve()
    .body(User.class);
```

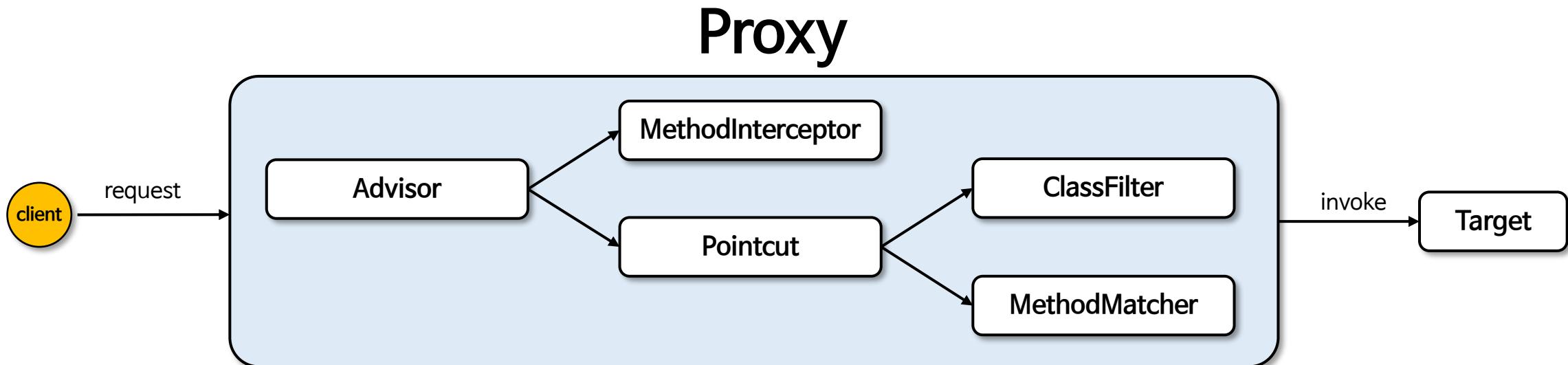
```
@HttpExchange("/users")
public interface UserClient {
    @GetExchange("/{id}")
    User getUserById(@PathVariable Long id);
}

@Bean
UserClient userClient(RestClient.Builder builder) {
    return builder.baseUrl("https://api.example.com").build()
        .create(UserClient.class);
}
```

항목	RestClient	HTTP Interface
코드 스타일	명령형(Imperative)	선언형(Declarative)
구현 방식	RestClient.get(), RestClient.post() 등을 직접 호출	인터페이스에 @GetExchange, @PostExchange 어노테이션을 선언
Spring 내부 동작	RestClient를 사용하여 HTTP 요청을 직접 생성 및 실행	내부적으로 RestClient를 사용하며 Spring이 인터페이스 구현체를 자동 생성
가독성	요청마다 메서드 체이닝을 사용해야 해서 길어질 수 있음	인터페이스에 메서드 정의만 하면 되므로 깔끔함
재사용성	요청마다 개별적으로 RestClient를 설정해야 함	@HttpExchange 인터페이스를 여러 곳에서 공통으로 재사용 가능
적합한 사용 사례	유연한 HTTP 요청을 여러 곳에서 개별적으로 실행할 때	API 클라이언트 인터페이스를 선언적으로 만들고 싶을 때
	요청별로 다른 설정(Timeout, Header 등)을 세밀하게 조정해야 할 때	여러 개의 API를 호출해야 하고 유지보수성을 고려할 때

✓ AOP 요소 이해

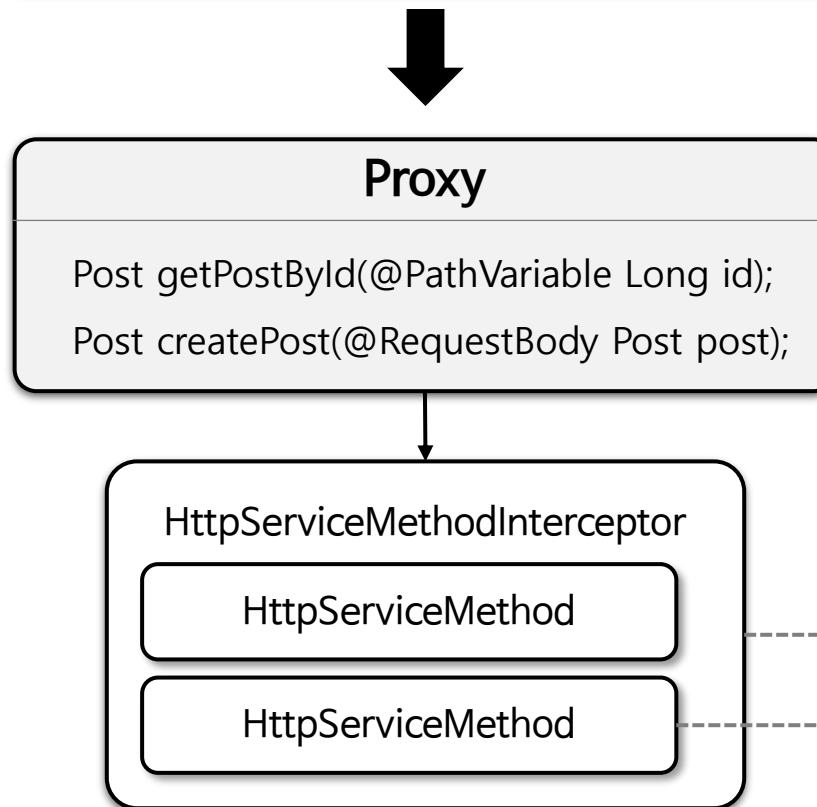
- Advisor
 - AOP Advice 와 Advice 적용 가능성을 결정하는 포인트컷을 가진 기본 인터페이스이다
- MethodInterceptor(Advice)
 - 대상 객체를 호출하기 전과 후에 추가 작업을 수행하기 위한 인터페이스로서 수행 이후 실제 대상 객체의 메서드를 호출한다
- Pointcut
 - AOP에서 Advice가 적용될 메소드나 클래스를 정의하는 것으로서 어드バイ스가 실행되어야 하는 '적용 지점'이나 '조건'을 지정한다



✓ @HttpExchange 초기화

```
@HttpExchange("/posts")
public interface PostClient {
    @GetExchange("/{id}")
    Post getPostById(@PathVariable Long id);

    @PostExchange
    Post createPost(@RequestBody Post post);
}
```



✓ @HttpExchange 처리 구조 이해

