

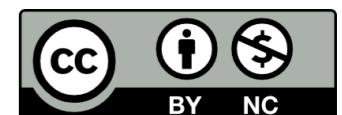
스프링 웹 프로그래밍

주입식

BETA

<https://springrunner.io>

이 문서는 베타 버전으로 지속적으로 내용 개선 및 추가됩니다.



이 문서의 내용은 크리에이티브 커먼즈 저작자표시-비영리 4.0 국제 라이선스에 따라 이용하실 수 있습니다.
<https://creativecommons.org/licenses/by/4.0/deed.ko>

목차

제 1장, 자바 웹 애플리케이션

제 2장, 스프링 웹 기술

제 3장, 안녕, 스프링 웹 프로그래밍

제 4장, 코드로 익히는 스프링 웹 프로그래밍 (워크숍 Workshop)

제 5장, HTTP 요청과 핸들러 연결하기

제 6장, 핸들러로 요청 데이터 다루기

제 7장, HTTP 응답 콘텐트 만들기

제 8장, 예외 처리하기

제 9장, 요청과 응답 가로채기

제 10장, 국제화와 메세지 다루기

제 11장, 스프링 웹 테스트 (준비 중)

부록, 스프링 IoC 컨테이너 구성과 사용 (준비 중)

부록, 디스패처서블릿을 들여다보다 (준비 중)

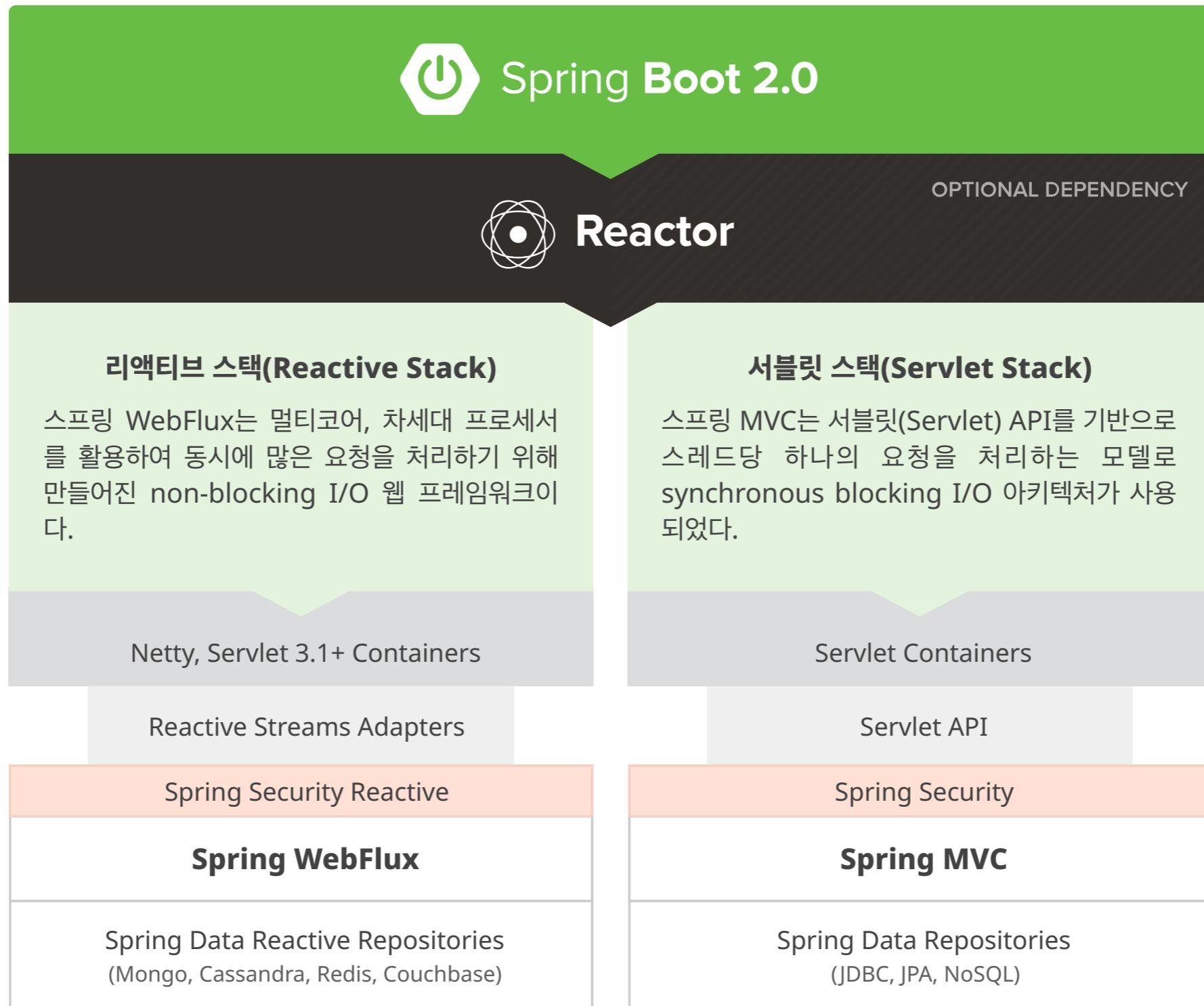
부록, 웹 애플리케이션 아키텍처

제 1장, 자바 웹 애플리케이션

 비공개

제 2장, 스프링 웹 기술

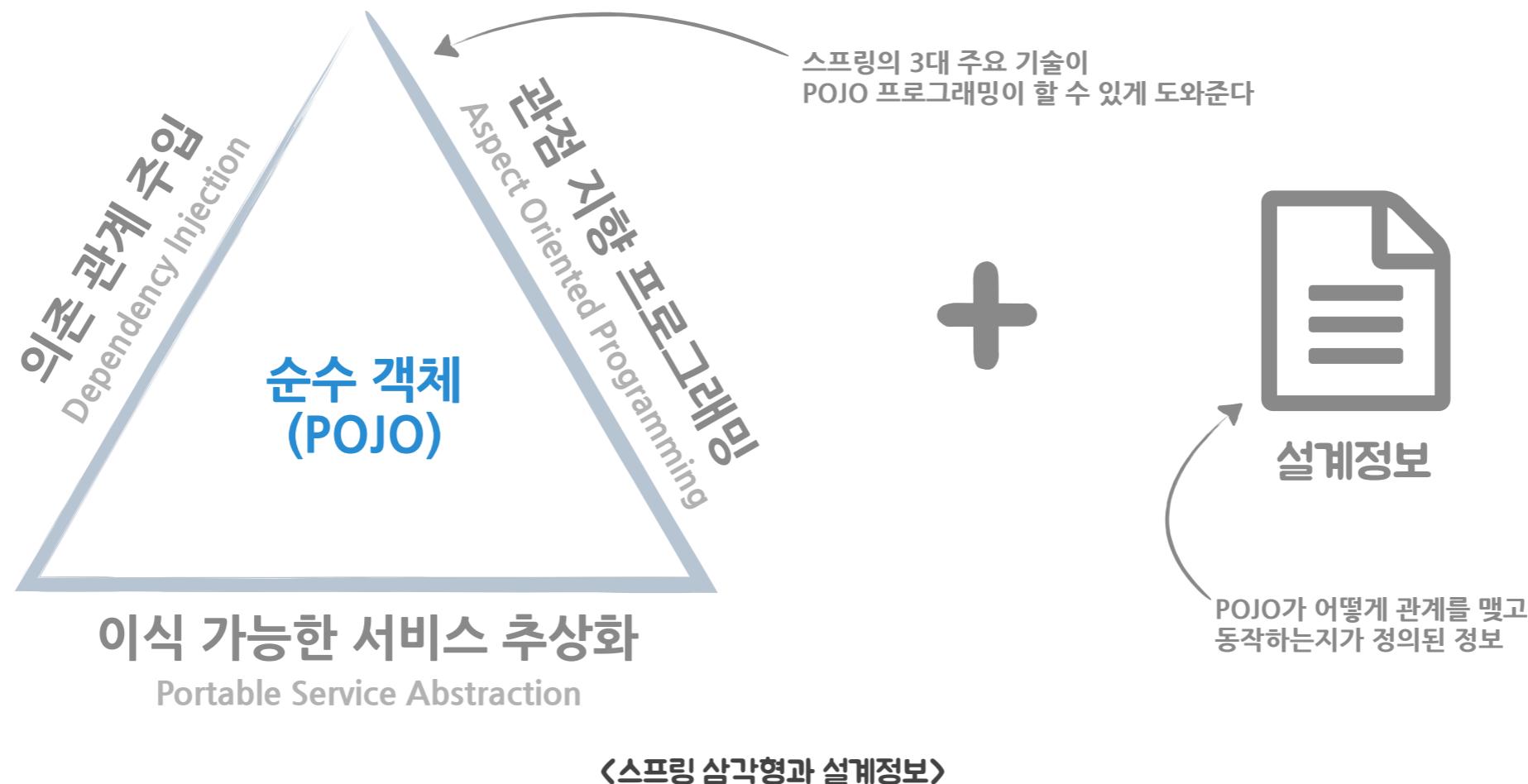
스프링 웹 애플리케이션 스택


 이 문서는 서블릿 스택만 다룹니다.


 2017년 9월 28일, 스프링 5 버전이 출시된 이후 스프링 웹 기술은 두 가지 스택으로 구성할 수 있다.

스프링 프레임워크(Spring Framework)

- ✓ 자바 플랫폼으로 애플리케이션을 개발하는데 필요한 하부 구조를 포괄적으로 제공한다.
- ✓ 스프링이 하부 구조를 처리하므로 개발자는 애플리케이션 개발에 집중할 수 있다.
- ✓ 핵심은 엔터프라이즈 서비스 기능을 POJO에게 제공하는 것이다.



- ✓ POJO란 Plain Old Java Object의 약자로 Martin Fowler에 의해 공표되었다.
- ✓ 자바의 단순한 오브젝트를 이용해 애플리케이션의 핵심 비즈니스 로직을 구현하는 것을 뜻한다.

제 2장, 스프링 웹 기술

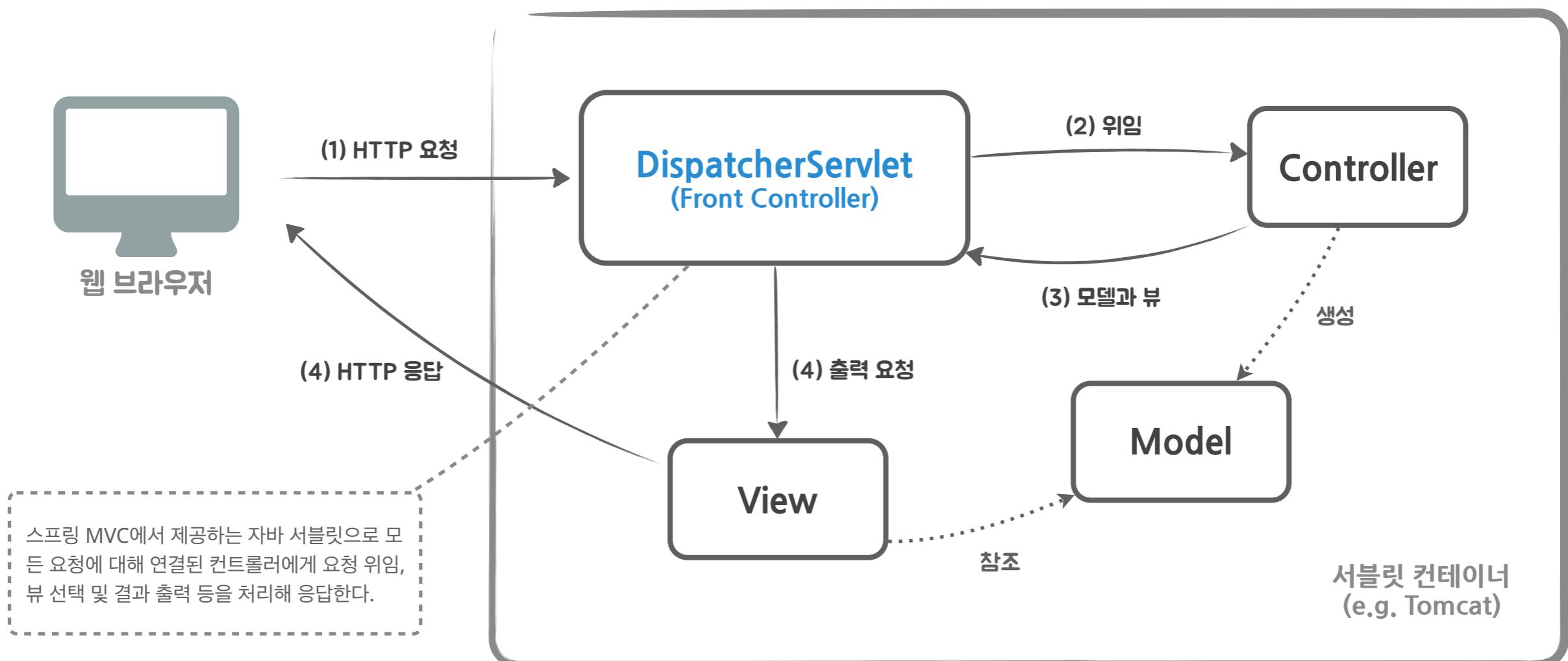
2.1, Spring MVC

Spring MVC

- ✓ 스프링 프레임워크를 구성하는 모듈, 정식 명칭은 Spring Web MVC이다.
- ✓ 서블릿 API를 기반으로 작성된 웹 프레임워크(Web Framework)이다.
- ✓ **프론트 컨트롤러 패턴**과 **MVC 아키텍처 패턴**을 사용한다.
- ✓ 유연하고 확장성이 뛰어난 구조를 제공한다.
- ✓ 견고한 웹 애플리케이션을 만드는데 도움이 되는 풍부한 기능을 제공한다.
- ✓ 자바 언어의 애노테이션(Annotation)과 리플렉션(Reflection) API 적극 사용한다.
- ✓ **애노테이션 기반 프로그래밍**(annotation-based programming) **모델**을 제공한다.
- ✓ 경량 함수형 프로그래밍 모델(functional programming model)을 지원한다. (스프링 버전 5.2 이상)
- ✓ CoC(Convention over Configuration, 설정보다 관례) 지원한다.

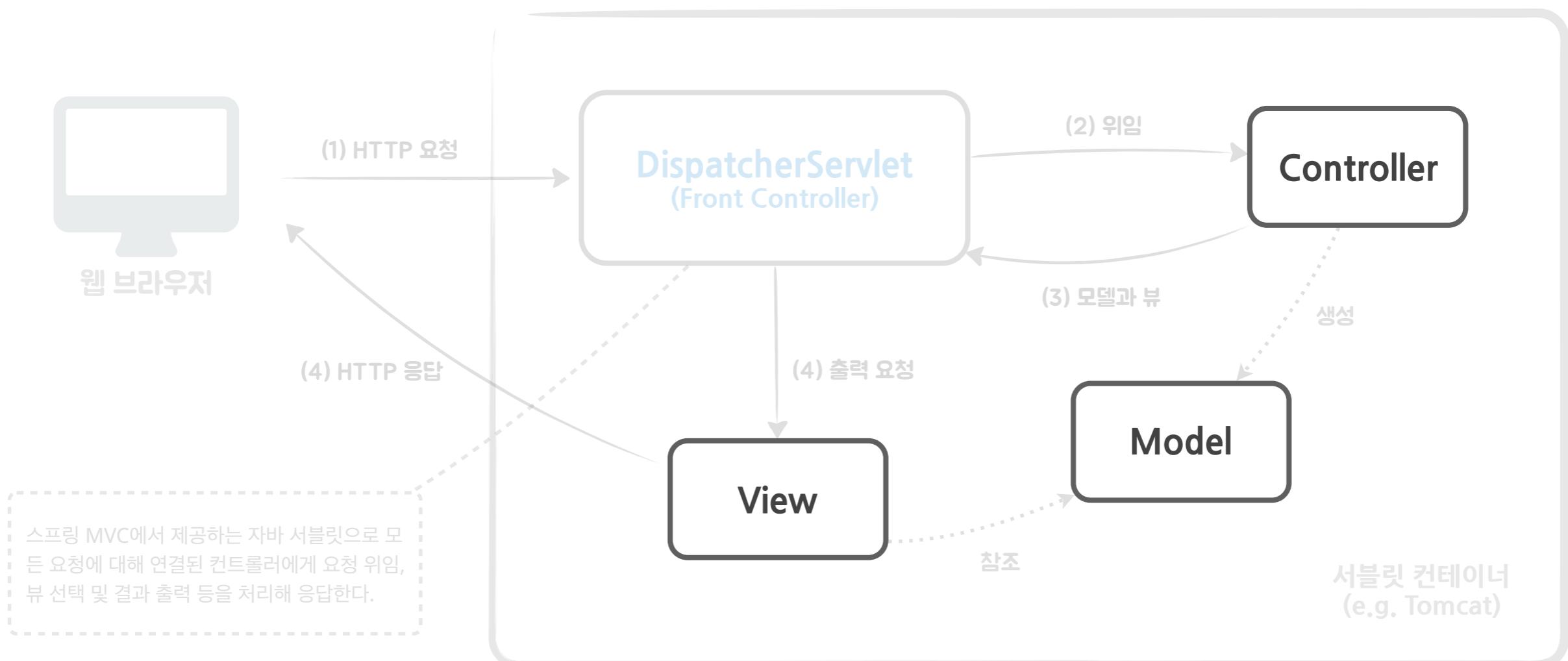
스프링 MVC의 요청/응답 처리 흐름

- ✓ 프론트 컨트롤러인 DispatcherServlet 을 중심으로 동작한다.
- ✓ MVC(Model-View-Controller)가 협력해 웹 요청과 응답을 처리한다.



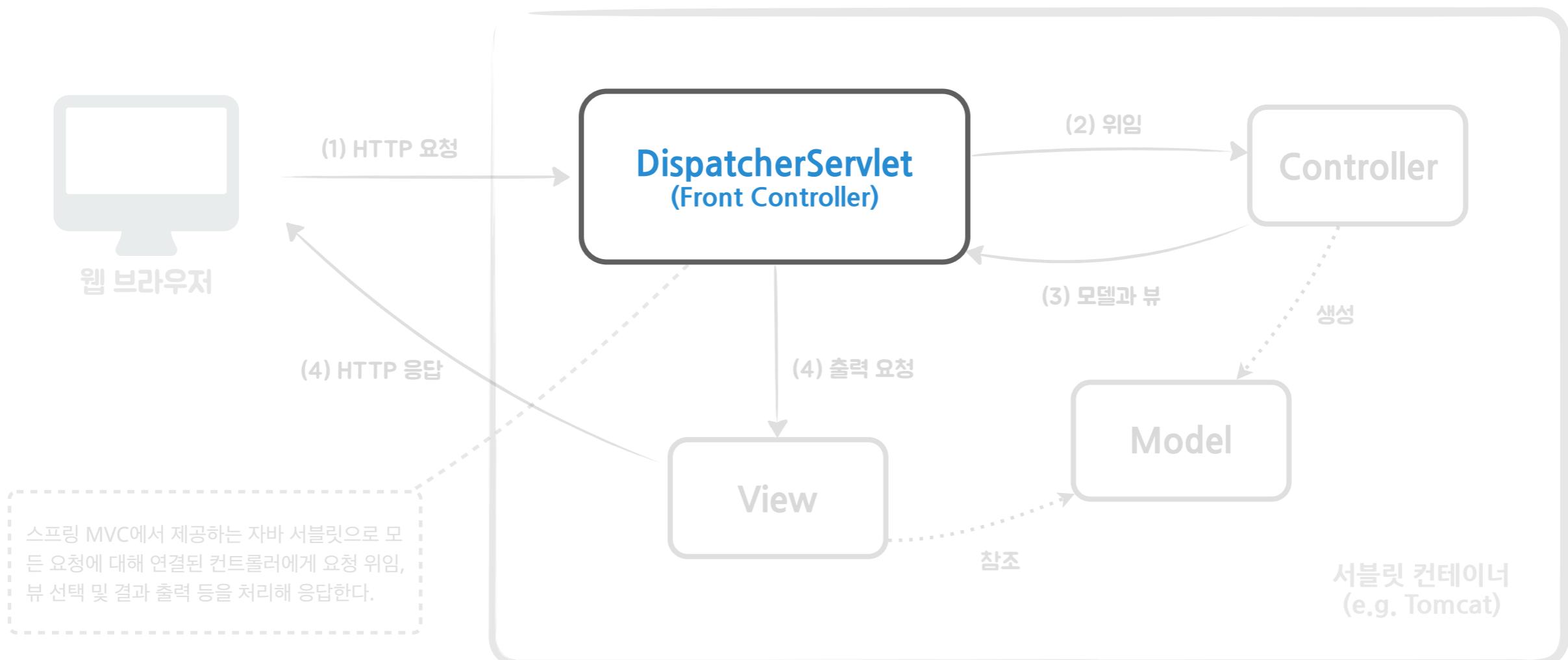
MVC 아키텍처 패턴: 로직과 프리젠테이션을 분리

- ✓ 화면의 구성요소와 데이터를 담은 모델(Model)
- ✓ 화면 출력 로직을 담은 뷰(View)
- ✓ 애플리케이션 제어 로직을 담은 컨트롤러(Controller)



프론트 컨트롤러(Front Controller) 패턴

- ✓ 컨트롤러를 대표해서 모든 요청과 응답을 대응한다.
- ✓ 요청 및 응답에 대한 횡단 관심사(예외 처리, 보안 등)를 적용한다.
- ✓ 적절한 하위 컨트롤러에게 작업 위임, 뷰 선택 및 결과를 출력한다.



애노테이션 기반 프로그래밍 모델(annotation-based programming)

- ✓ @Controller, @RequestMapping 등 다양한 애노테이션 제공한다.
- ✓ 애노테이션을 통한 요청 연결, 데이터 가공, 예외 처리 등 구성한다.
- ✓ 메타 애노테이션을 통해 사용자 정의 애노테이션 지원한다.

```
/*
 * 애노테이션으로 이 클래스가 컨트롤러 컴포넌트임을 선언
 */
@Controller
public class HelloController {

    /*
     * 애노테이션으로 웹 요청을 연결할 URL 선언
     */
    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // View 생성
        View view = new InternalResourceView("/WEB-INF/templates/HelloView.jsp");

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setView(view);

        return mav;
    }
}
```

애노테이션(Annotation)

- ✓ 코드의 메타-데이터(Metadata)로 작성, 컴파일 또는 런타임에 활용한다.
- ✓ JDK가 제공하는 빌트인(Built-in)과 직접 작성하는 커스텀(Custom)으로 분류된다.
- ✓ 패키지, 클래스, 메소드, 필드에 선언 할 수 있다.
- ✓ @{AnnotationName} 으로 표기한다.

```
class Item implements Ordered {
    @Override
    public int getOrder() {
        return 0;
    }
}

class Data {
    @Deprecated
    public Data(Object value) {
    }
}
```

해당 메소드가 부모 클래스에 있는 메소드를 오버라이드 했다는 것을 명시적으로 선언

향후 더 이상 사용되지 않은 클래스나 메소드에 선언, 컴파일 시 경고 문구가 노출

사용자 정의 애노테이션(Custom Annotation)

- ✓ 용도에 맞춰 직접 작성하는 애노테이션이다.
- ✓ @interface 키워드로 사용자 정의 애노테이션을 작성한다.
- ✓ 메타-애노테이션(meta-annotation)을 함께 사용해서 작성한다.

```
@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Component {  
    String name();  
}  
  
@Component(name = "블로그 서비스")  
class BlogService {  
}
```

메타-애노테이션(meta-annotation)
애노테이션에 사용하는 애노테이션으로 커스텀 애노테이션의
동작대상 및 유지시간을 결정

리플렉션 API(Reflection API)

- ✓ 클래스나 객체의 메소드, 필드 등의 정보를 확인 또는 이용할 수 있는 API이다.
- ✓ 객체의 생성이나 메소드 호출을 동적으로 할 수 있는 강력한 기법이다.

```
// Without reflection
Duck duck = new Duck();
duck.quack();

// With reflection
Object duckObject = Class.forName("examples.spring.Reflection$Duck").newInstance();
Method quack = duckObject.getClass().getDeclaredMethod("quack", new Class<?>[0]);
quack.invoke(duckObject);

class Duck {
    void quack() {
        System.out.println("꽥꽥!");
    }
}
```

애노테이션과 리플렉션 API 활용

```
@Target({ElementType.METHOD, ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@interface Component {  
    String name();  
}
```

```
@Component(name = "블로그 서비스")  
class BlogService {
```

```
}
```

```
Class<BlogService> blogServiceClass = BlogService.class;  
Component componentAnnotation = blogServiceClass.getAnnotation(Component.class);  
  
out.println(format("componentAnnotation = name: %s", componentAnnotation.name()));
```

@Component 애노테이션을 리플렉션 API로 획득

BlogService 클래스에 선언된
@Component 애노테이션의 name 속성 값 출력

제 2장, 스프링 웹 기술

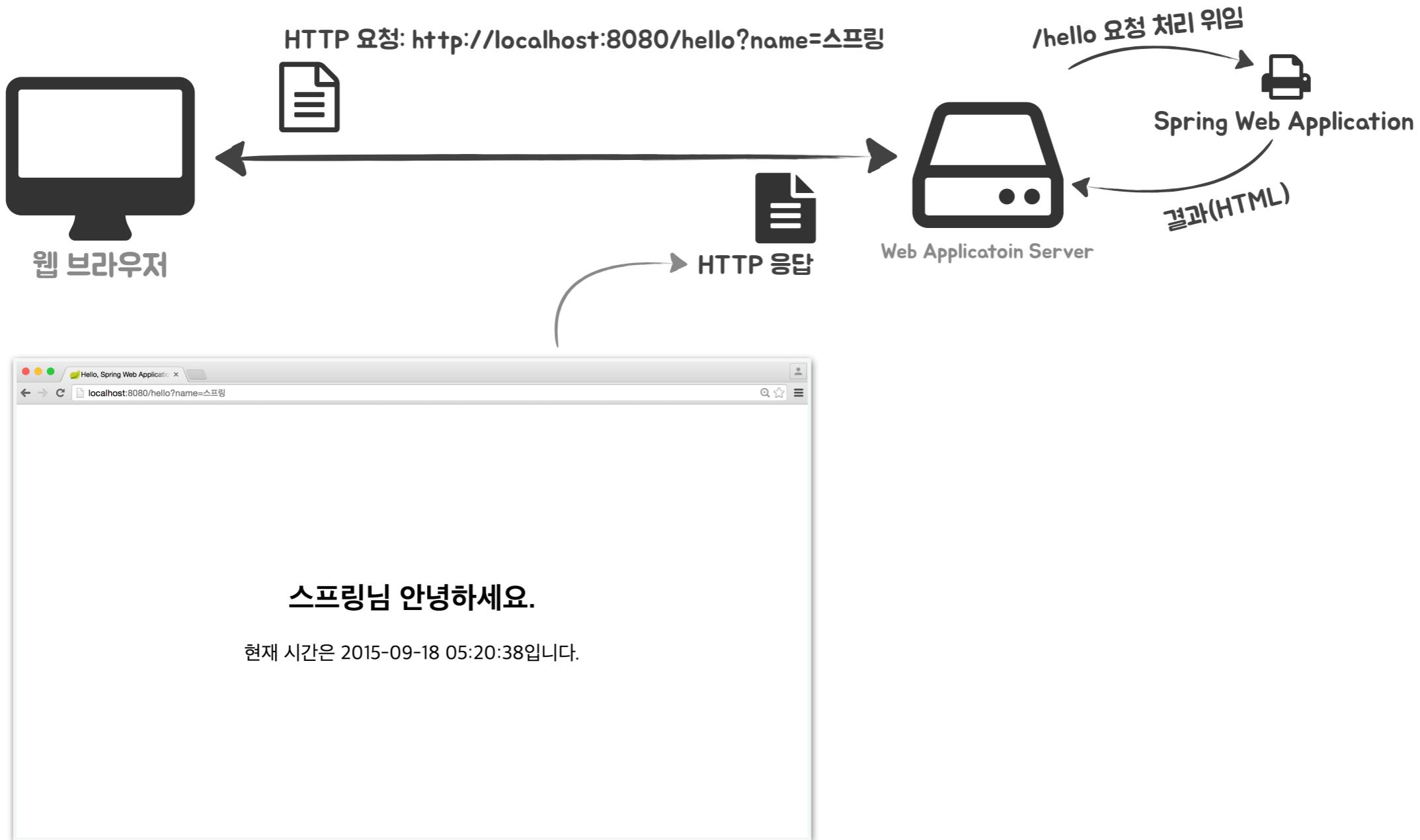
2.1, Spring Boot 웹 지원

■ 비공개

제 3장, **안녕, 스프링 웹 프로그래밍**

간단한 스프링 웹 애플리케이션 개발하기

- ✓ 인삿말과 함께 현재 날짜와 시간을 출력하는 웹 애플리케이션 만들기
- ✓ 사용자의 이름을 전송하면, 그 이름으로 인삿말을 작성하는 기능을 구현



라이브 코딩

<https://springrunner.io>

프로젝트 디렉토리 구조

Package Explorer

- hello-spring-web [boot]
 - src/main/java
 - examples.spring
 - HelloSpringWebApplication.java
 - src/main/resources
 - templates
 - static
 - application.properties
 - src/test/java
 - examples.spring
 - HelloSpringWebApplicationTests.java
 - JRE System Library [JavaSE-1.8]
 - Project and External Dependencies
 - bin
 - gradle
 - src
 - main
 - webapp
 - test
 - build.gradle
 - gradlew
 - gradlew.bat
 - settings.gradle
- servlet-web [boot]

- ✓ src/main/java : 자바 소스 파일
- ✓ src/main/resource : 자원 파일 (애플리케이션 구성 또는 설정 파일)
- ✓ src/main/webapp : 웹 애플리케이션 디렉토리
- ✓ src/test/java : 테스트 자바 소스 파일
- ✓ src/test/resource : 테스트 자원 파일

스프링 부트 기반 웹 애플리케이션의 진입점

메이븐(Maven)이 업계에 자리 잡은 후 자바 프로젝트에 표준 구조는 대부분 메이븐의 표준 디렉토리 구조를 따라 구성한다. 그레이들도 메이븐에 표준 디렉토리 구조를 따라 구성된다.

그레이들(Gradle) 빌드 스크립트 파일

Boot Dashboard

Servers

Type tags, projects, or working set names to filter

local

- hello-spring-web
- servlet-web

1 elements hidden by filter

hello-spring-web

Problems Javadoc Declaration Progress Console

```
<terminated> servlet-web - ServletWebApplication [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/bin/java (Dec 12, 2018, 5:39:16 PM)
2018-12-12 17:39:17.577  INFO 23721 --- [           main] examples.servlet.ServletWebApplication : No active profile set, falling back to default profiles
2018-12-12 17:39:18.491  INFO 23721 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080
2018-12-12 17:39:18.517  INFO 23721 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-12-12 17:39:18.517  INFO 23721 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.0.28
2018-12-12 17:39:18.524  INFO 23721 --- [           main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: [/System/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/lib/server:/System/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/lib:/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/lib:/System/Library/Java/JavaVirtualMachines/jdk1.8.0_191.jdk/Contents/Home/lib/server]
2018-12-12 17:39:18.712  INFO 23721 --- [           main] org.apache.jasper.servlet.TldScanner : At least one JAR was scanned for TLDs.
2018-12-12 17:39:18.716  INFO 23721 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebAppContext
2018-12-12 17:39:18.716  INFO 23721 --- [           main] o.s.web.context.ContextLoader : Root WebApplicationContext: initializers fully initialized
2018-12-12 17:39:19.096  INFO 23721 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http://0.0.0.0:8080)
2018-12-12 17:39:19.099  INFO 23721 --- [           main] examples.servlet.ServletWebApplication : Started ServletWebApplication in 1.099 seconds (version 1.0-SNAPSHOT)
2018-12-12 17:42:16.105  INFO 23721 --- [n(11)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.
```

그레이들로 JSP 및 JSTL 라이브러리 의존성 추가

1. 그레이들 빌드 스크립트 선택

```

buildscript {
    ext {
        springBootVersion = '2.1.1.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    runtimeOnly('org.apache.tomcat.embed:tomcat-embed-jasper')
    runtimeOnly('javax.servlet:jstl')
    testImplementation('org.springframework.boot:spring-boot-starter-test')
}

sourceCompatibility = JavaVersion.VERSION_1_8
targetCompatibility = JavaVersion.VERSION_1_8
compileJava.options.encoding = 'UTF-8'

```

2. dependencies에 의존성 추가

Console Output:

```

2018-12-12 17:39:17.577  INFO 23721 --- [           main] examples.servlet.ServletWebApplication : No active profile set, falling back to default profiles: Tomcat initialized with port(s): 8080
2018-12-12 17:39:18.491  INFO 23721 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Starting service [Tomcat]
2018-12-12 17:39:18.517  INFO 23721 --- [           main] o.apache.catalina.core.StandardService : Starting Servlet Engine: Apache Tomcat/9.0.13
2018-12-12 17:39:18.517  INFO 23721 --- [           main] o.apache.catalina.core.StandardEngine : The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path: The APR based Apache Tomcat Native library which allows optimal performance in production environments was not found on the java.library.path
2018-12-12 17:39:18.524  INFO 23721 --- [           main] o.a.catalina.core.AprLifecycleListener : At least one JAR was scanned for TLDs.
2018-12-12 17:39:18.712  INFO 23721 --- [           main] o.apache.jasper.servlet.TldScanner : Initialization Spring embedded WebApp
2018-12-12 17:39:18.716  INFO 23721 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Root WebApplicationContext: initial
2018-12-12 17:39:18.716  INFO 23721 --- [           main] o.s.web.context.ContextLoader : Tomcat started on port(s): 8080 (ht
2018-12-12 17:39:19.096  INFO 23721 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Started ServletWebApplication in 1.
2018-12-12 17:39:19.099  INFO 23721 --- [           main] examples.servlet.ServletWebApplication : Application shutdown requested.
2018-12-12 17:42:16.105  INFO 23721 --- [n(11)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.

```

HTTP 요청을 처리할 컨트롤러(Controller) 클래스 작성

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.view.JstlView;

@Controller ← 웹 요청을 처리할 컨트롤러(Controller)임을 선언
public class HelloController {
    @RequestMapping("/hello") ← /hello URL로 들어오는 HTTP 요청을 처리 할 것을 선언
    public ModelAndView hello() {
        // View 생성
        View view = new JstlView("/WEB-INF/templates/HelloView.jsp");

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.setView(view);

        return mav;
    }
}
```

HTTP 요청을 처리할 컨트롤러(Controller) 클래스 작성

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.view.JstlView;

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello() {
        // View 생성
        View view = new JstlView("/WEB-INF/templates/HelloView.jsp");

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.setView(view);

        return mav;
    }
}
```

HTTP 요청을 처리할 목적으로 작성한
메소드를 **핸들러**라고 부른다

핸들러는 **모델과 뷰(ModelAndView)**를 반환해야 한다

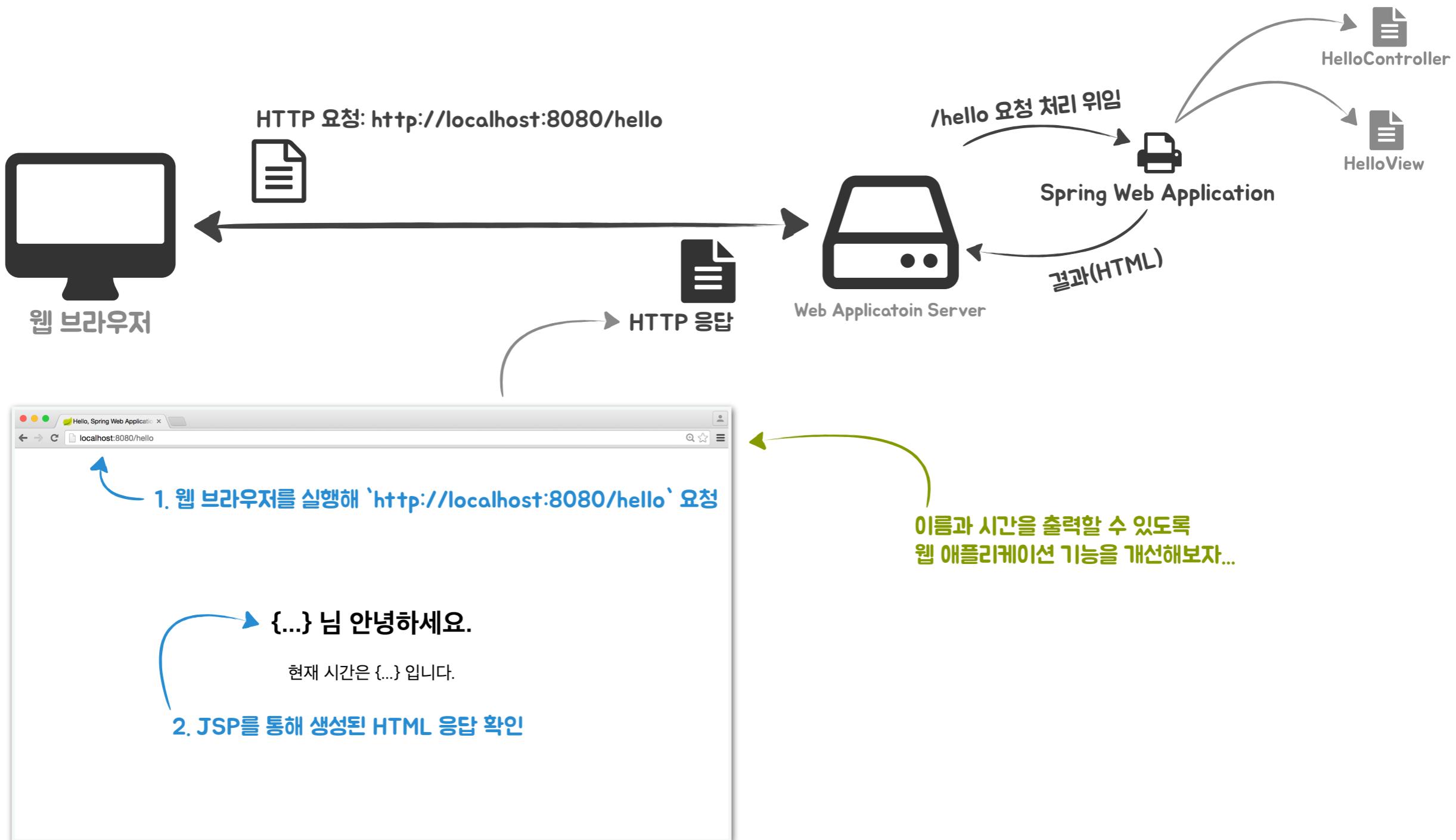
HTTP 응답으로 보낼 HTML을 출력할 뷰 템플릿(JSP) 작성

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8"/>
<title>Hello, Spring Web Application</title>

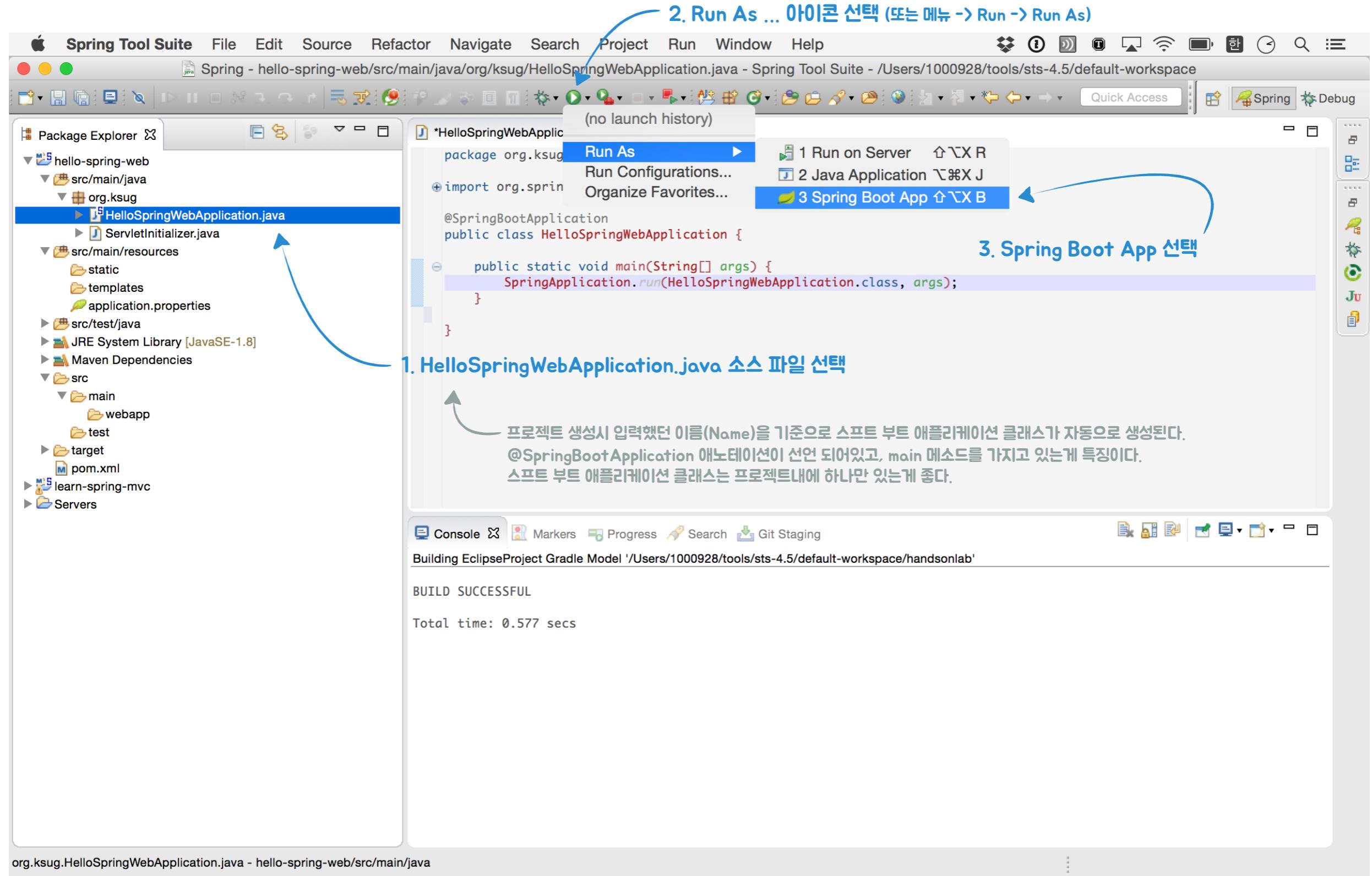
<style type="text/css">
    html, body {height:100%;}
    html {display:table; width:100%;}
    body {display:table-cell; text-align:center; vertical-align:middle;}
</style>
</head>
<body>
<h2>{...} 님 안녕하세요.</h2>
<p>현재 시간은 {...} 입니다.</p>
</body>
</html>
```

웹 브라우저로 웹 애플리케이션 접속해보기

- ✓ Spring Web Application은 웹 브라우저에서 전송된 HTTP 요청을 받아 HTTP 응답을 수행
- ✓ Spring Web Application은 요청과 응답을 수행하기 위해 HelloController와 HelloView를 사용



스프링 웹 애플리케이션 실행 및 종료 (1/4)



스프링 웹 애플리케이션 실행 및 종료 (2/4)

The screenshot shows the Spring Tool Suite interface. In the top center, the file `HelloSpringWebApplication.java` is open, displaying the main method of a Spring Boot application. Below it, the `Console` view shows the application's startup logs. A blue arrow points from the text "1. Console 창을 통해 실행 결과 확인" to the console window. Another blue arrow points from the text "2. http 프로토콜 / 8080 포트로 톰캣 시작 그리고 애플리케이션이 실행됨" to the bottom right of the console window.

```

package org.ksug;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class HelloSpringWebApplication {
    public static void main(String[] args) {
        SpringApplication.run(HelloSpringWebApplication.class, args);
    }
}

```

1. Console 창을 통해 실행 결과 확인

2. http 프로토콜 / 8080 포트로 톰캣 시작
그리고 애플리케이션이 실행됨

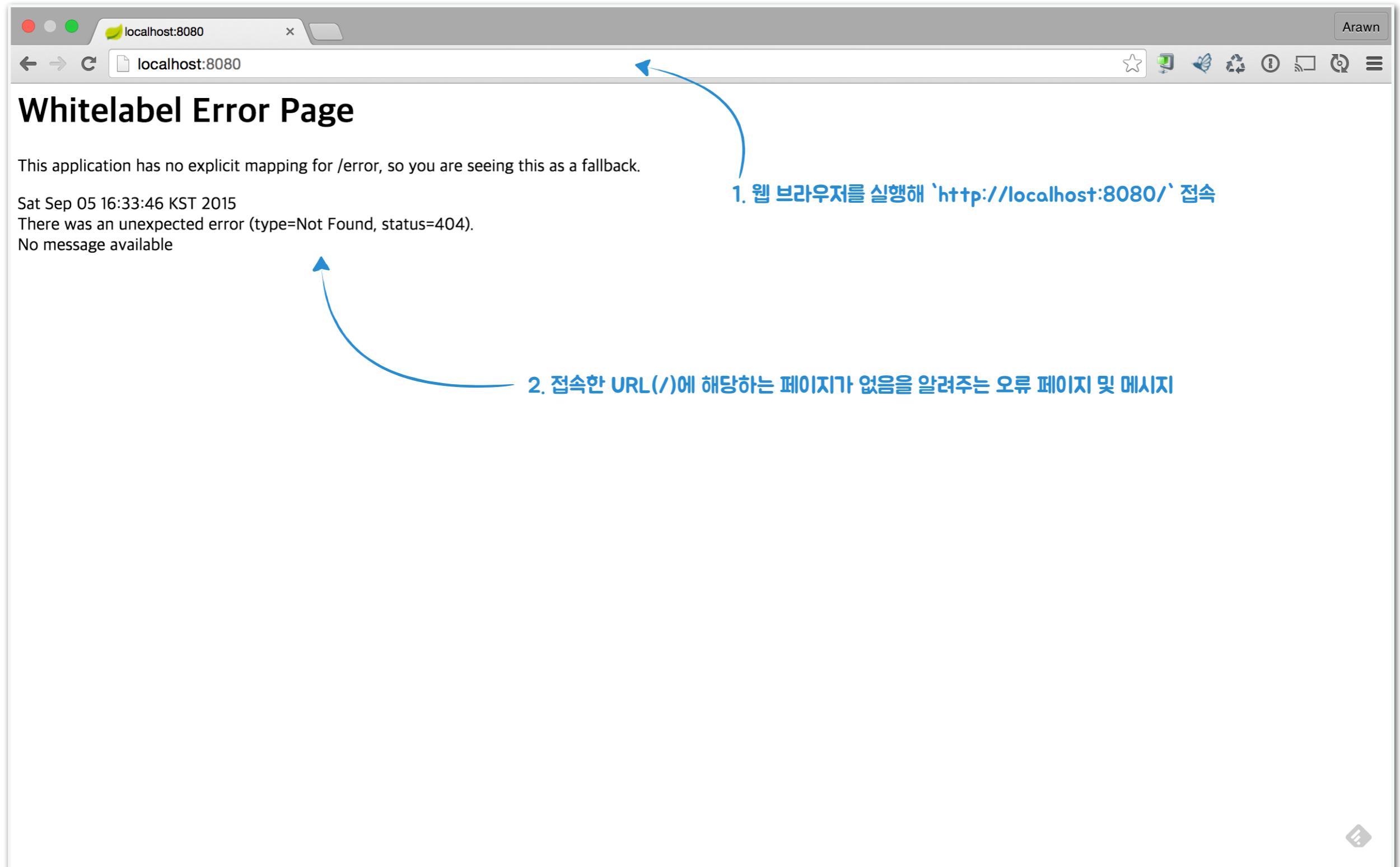
톰캣(Tomcat)은 웹 애플리케이션 서버(Web Application Server, WAS)다

```

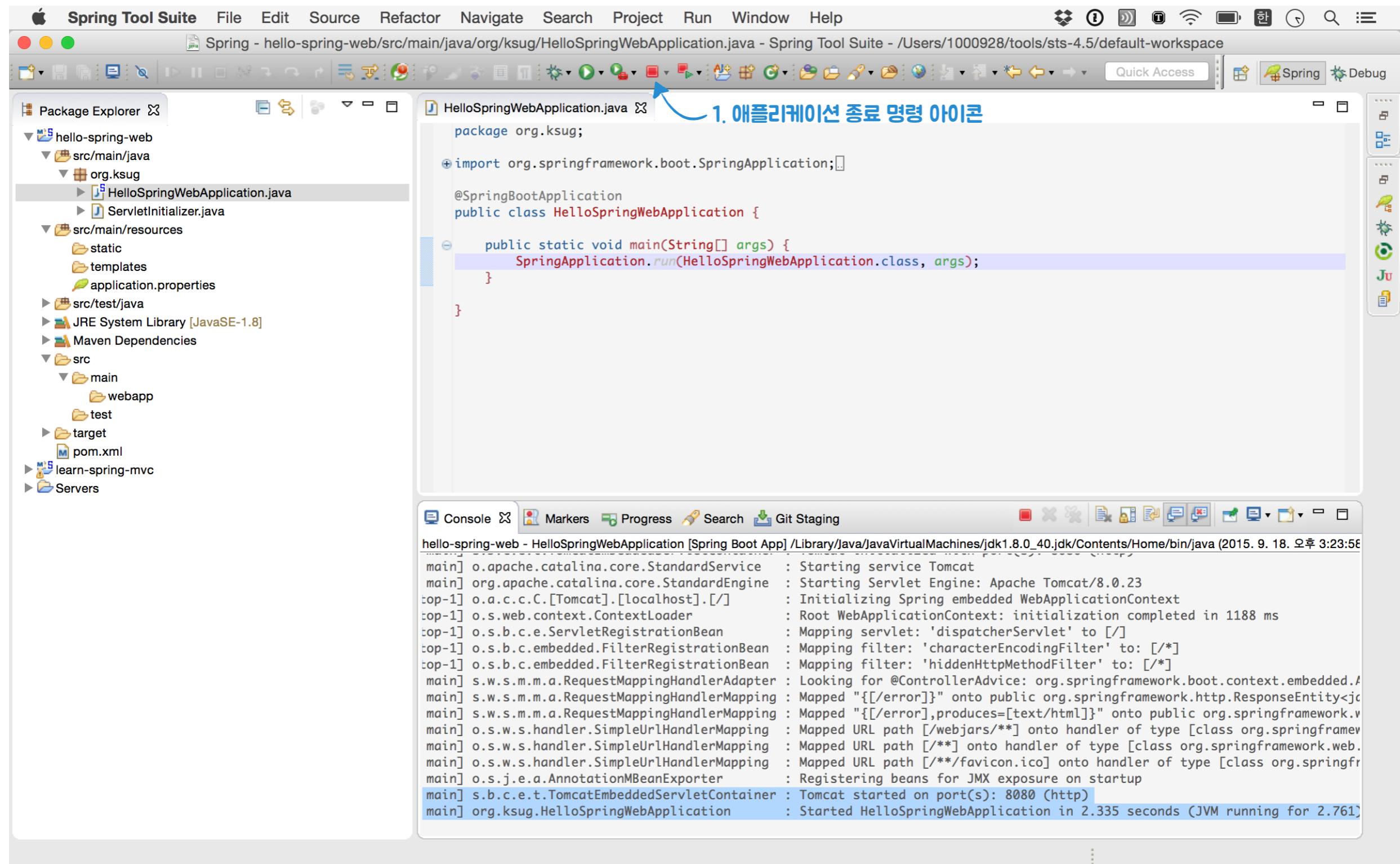
main] o.apache.catalina.core.StandardService : Starting service Tomcat
main] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.0.23
top-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
top-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1188 ms
top-1] o.s.b.c.e.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to [/]
top-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
top-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@53d333e5: startup date [2015.9.18 오전 3:23:56]
main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.util.List<java.lang.String>>> org.springframework.web.servlet.error.DefaultErrorAttributes.error(Map)
main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet.ModelAndView org.springframework.web.servlet.error.DefaultErrorAttributes.errorHtml(Map)
main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.handler.SimpleUrlHandlerMapping]
main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
main] org.ksug.HelloSpringWebApplication : Started HelloSpringWebApplication in 2.335 seconds (JVM running for 2.761)

```

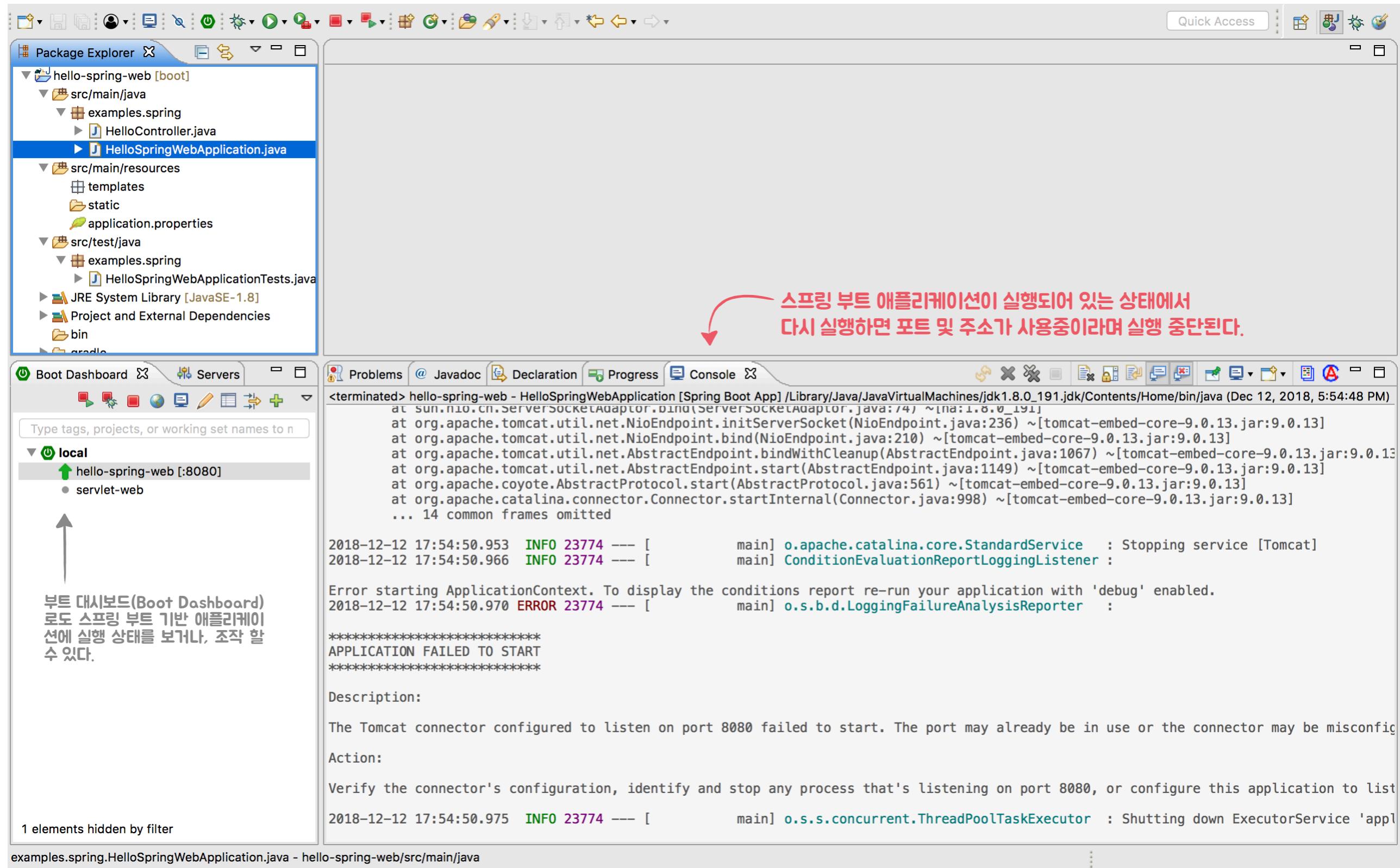
스프링 웹 애플리케이션 실행 및 종료 (3/4)



스프링 웹 애플리케이션 실행 및 종료 (4/4)



스프링 부트 애플리케이션은 두번 실행하면 오류가 뜬다!



HTTP 요청 데이터를 담을 모델 클래스 작성

```
import java.util.Date;  
  
import org.springframework.format.annotation.DateTimeFormat;  
import org.springframework.util.StringUtils;  
  
public class HelloModel {  
  
    private String name;  
  
    @DateTimeFormat(pattern = "yyyy-MM-dd hh:mm:ss")  
    private Date currentDatetime;  
  
    public HelloModel(String name) {  
        if (!StringUtils.hasText(name)) {  
            this.name = "스프링 웹 애플리케이션";  
        } else {  
            this.name = name;  
        }  
        this.currentDatetime = new Date();  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Date getCurrentDatetime() {  
        return currentDatetime;  
    }  
}
```

뷰에서 날짜와 시간을 출력할 때 사용할 포맷을 정의하는 애노테이션 선언

HTTP 요청 데이터로 모델 생성 후 추가

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

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // View 생성
        View view = new JstlView("/WEB-INF/templates/HelloView.jsp");

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setView(view);

        return mav;
    }
}
```

HTTP 요청 데이터에서 name 값 획득한다

뷰에 전달할 모델을 추가 (key, value)한다

뷰 템플릿에서 모델을 사용해 응답을 출력하도록 코드 개선

```

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8"/>
    <title>Hello, Spring Web Application</title>
    <style type="text/css">
        html, body {height:100%;}
        html {display:table; width:100%;}
        body {display:table-cell; text-align:center; vertical-align:middle;}
    </style>
</head>
<body>
    <h2>${hello.name}님 안녕하세요.</h2>
    <p>현재 시간은 <spring:eval expression="hello.currentDatetime"/>입니다.</p>
</body>
</html>

```

spring 태그를 사용할 것을 선언한다

\${} 블록은 표현식 언어(expression language)로 뷰에 전달된 모델에 값을 출력할 때 사용한다
이때 출력할 모델은 ModelAndView에 담을 때 입력한 이름(key)으로 찾을 수 있다

날짜와 시간을 출력, spring 태그를 사용해 출력하면
@DateTimeFormat 애노테이션으로 작성한 출력 포맷을 사용한다

```

ModelAndView mav = new ModelAndView();
mav.addObject("hello", model);
mav.setView(view);

```

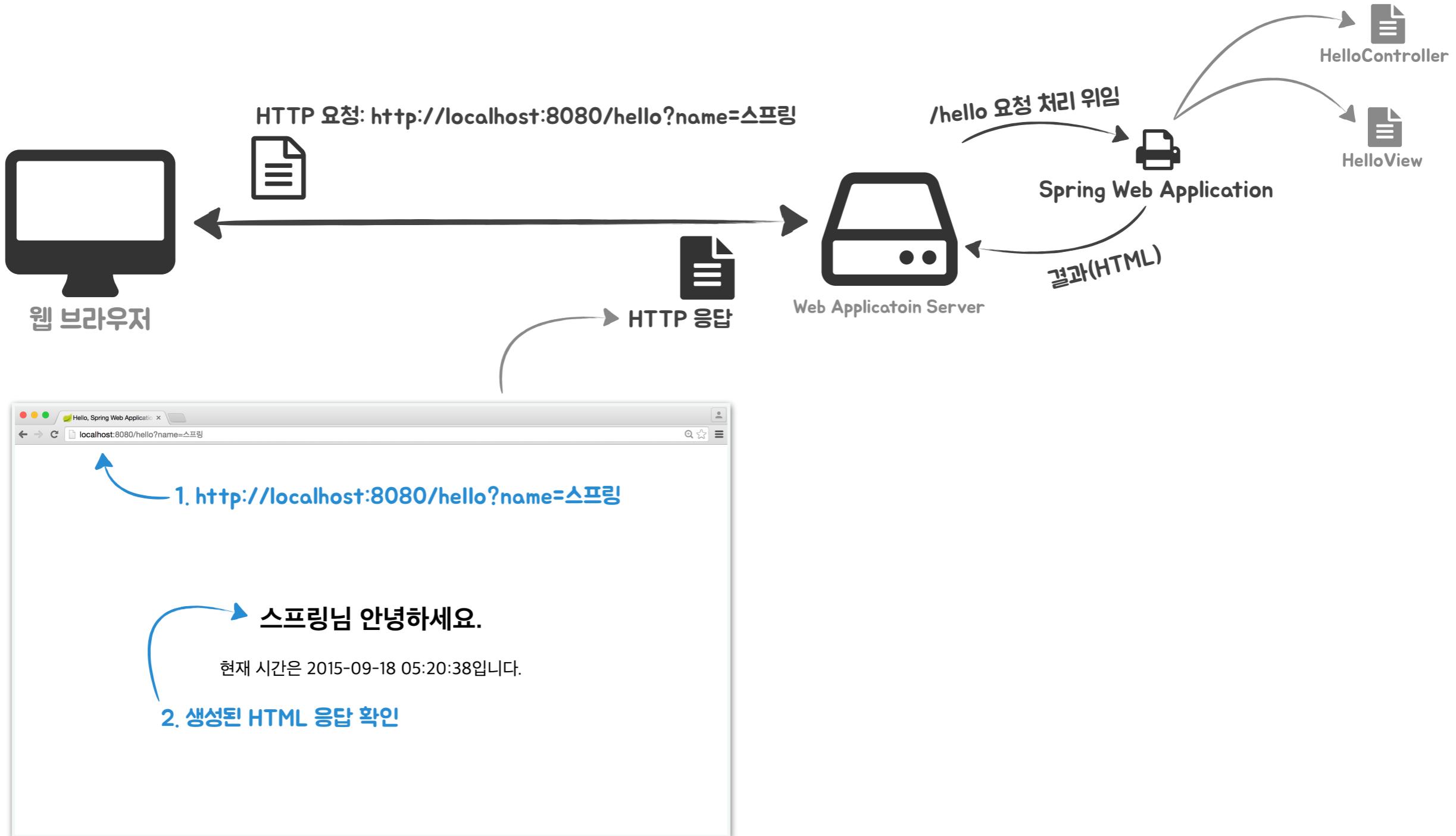
```

HelloModel
@DateTimeFormat(pattern = "yyyy-MM-dd hh:mm:ss")
private Date currentDatetime;

```

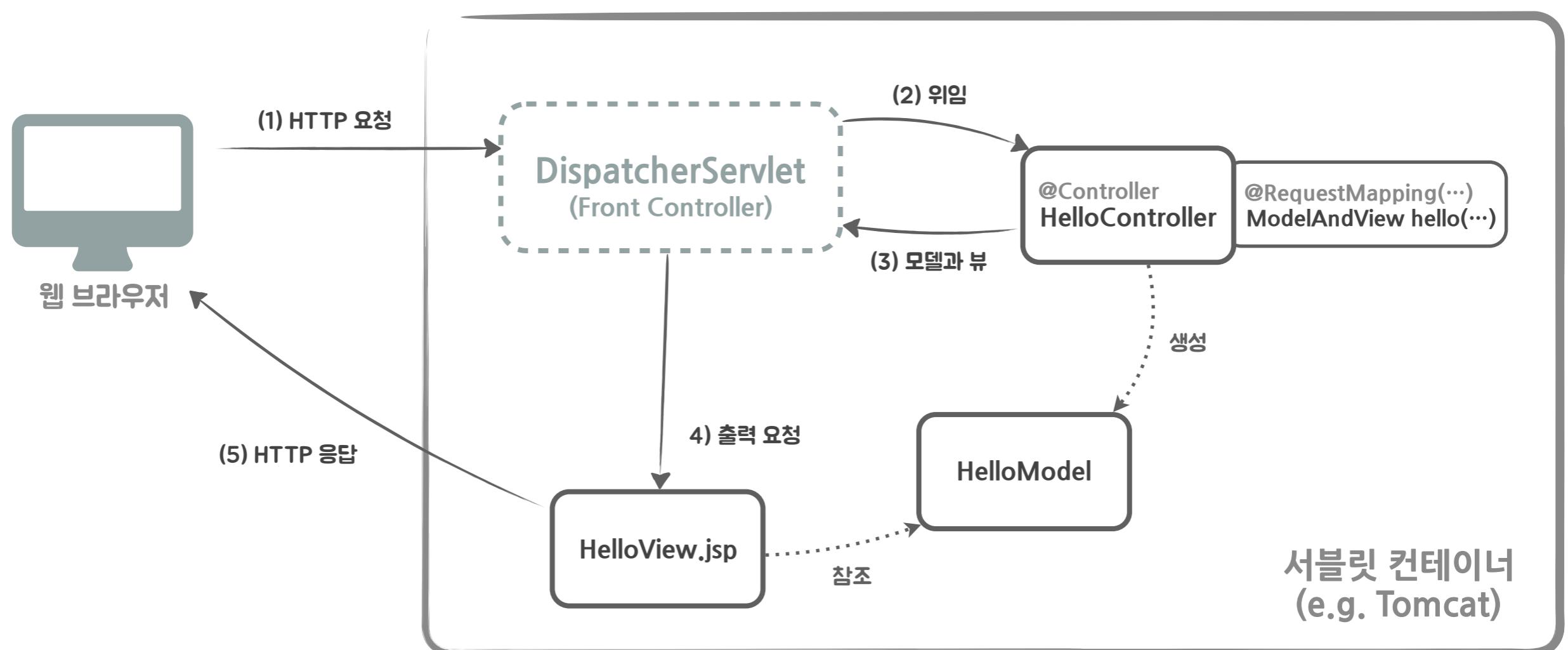
전송된 이름과 현재 날짜, 시간을 출력하는 결과 보기

- ✓ 컨트롤러는 쿼리 스트링(Query String)으로 전송된 요청 데이터를 사용해 모델을 생성해서 뷰에게 전달
- ✓ 뷰는 전달된 모델을 사용해 HTML을 동적으로 생성



예제를 통해 알 수 있는 스프링 웹 프로그래밍

- ✓ 디스패처서블릿에게 전달된 요청은 URL이 연결된 핸들러에 위임
- ✓ 핸들러는 요청 데이터를 받아, 로직 수행 후 모델과 뷰를 반환
- ✓ 디스패처서블릿은 뷰를 찾아 모델을 전달하고 응답 출력 요청
- ✓ 애노테이션 기반으로 요청 연결, 수행, 응답 처리



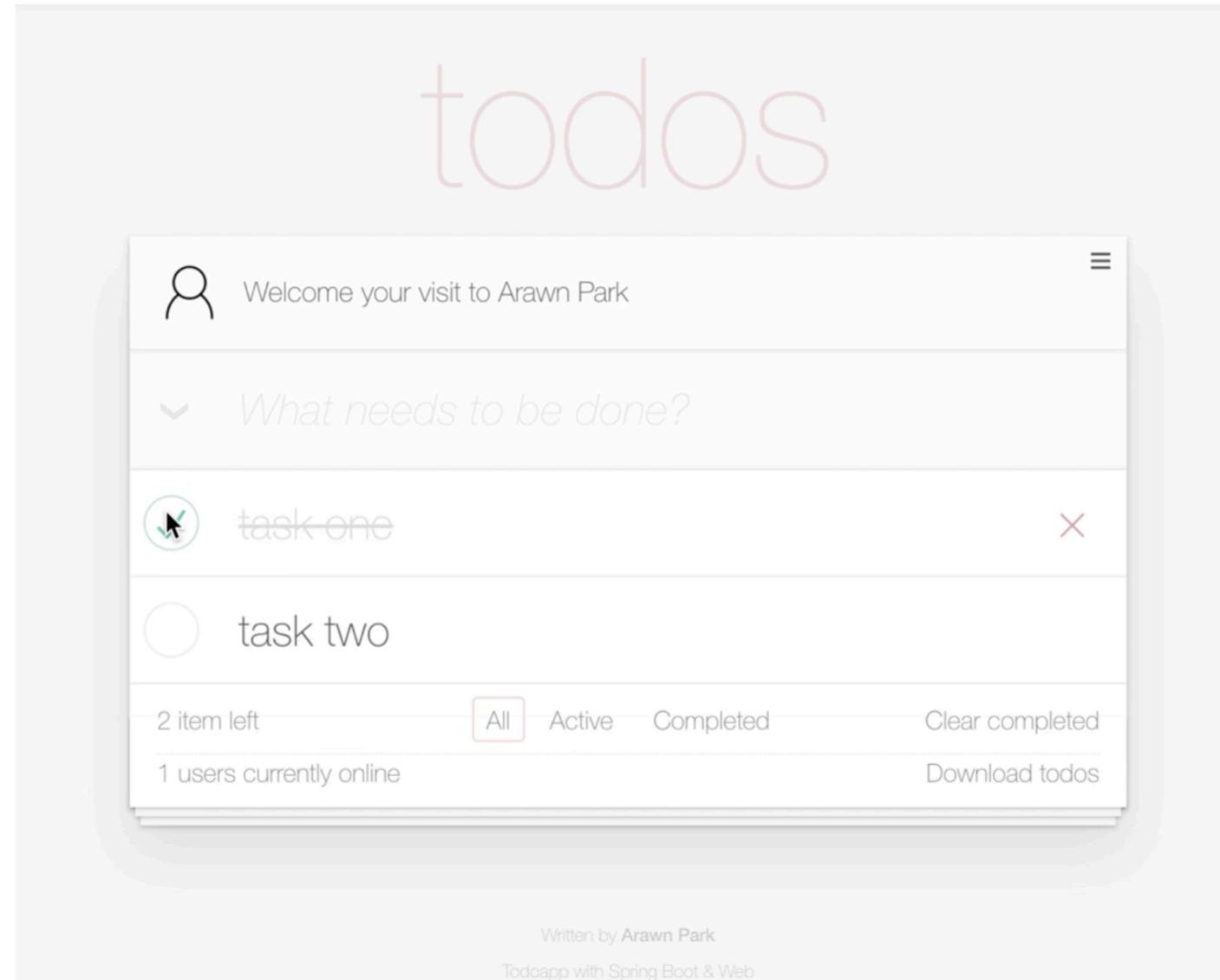
제 4장, 코드로 익히는 스프링 웹 프로그래밍

<https://springrunner.io/training/mastering-spring-web-101-workshop/>

Mastering Spring Web 101 Workshop

진행방식

웹 클라이언트와 애플리케이션 정의서가 제공됩니다. 참가자는 애플리케이션 정의서를 바탕으로 웹 클라이언트가 동작할 수 있도록 서버 사이드를 개발합니다.



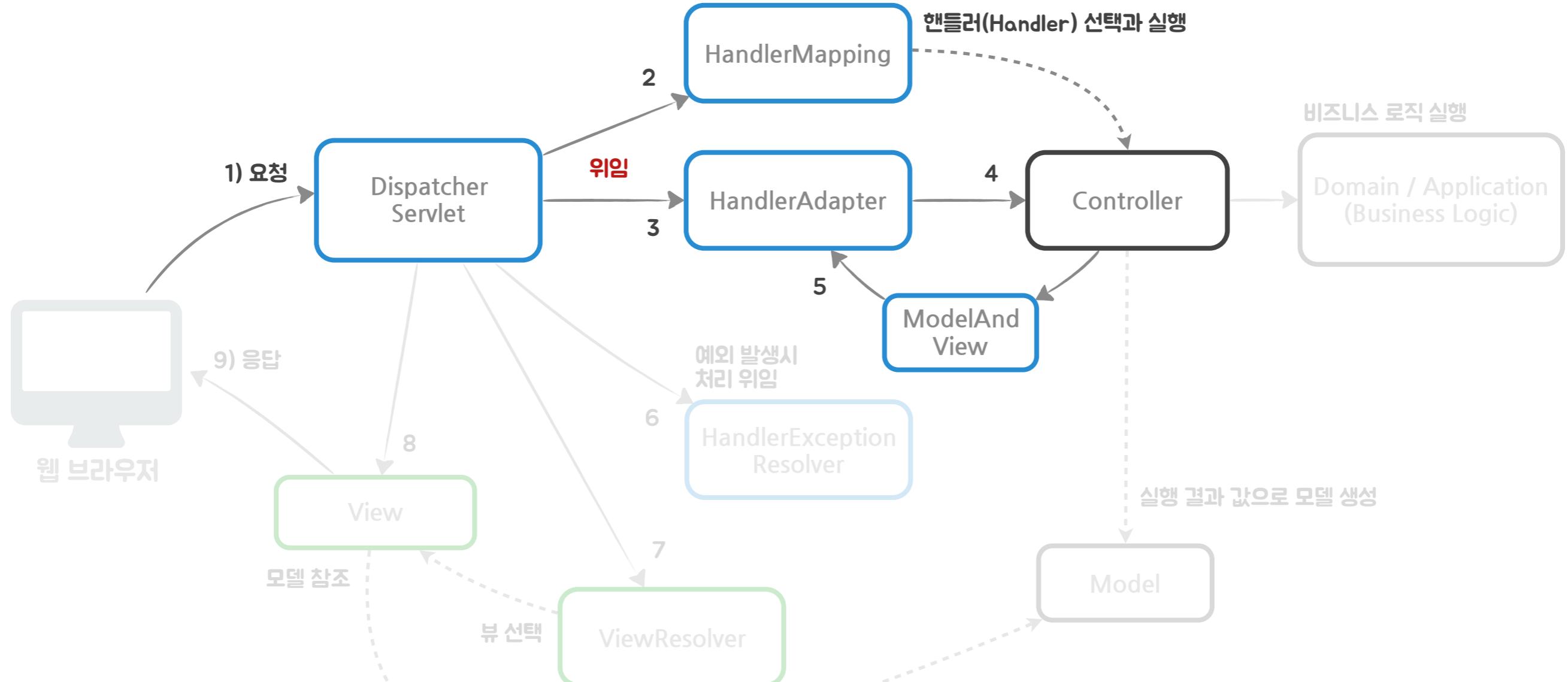
강사가 라이브 코딩과 함께 Spring MVC(또는 Spring Boot)에 기능을 설명해주면, 참가자는 해당 코드를 직접 작성하고 실행-테스트 하는 과정을 반복하며 완전한 웹 애플리케이션을 개발해보는 방식으로 진행됩니다.

백견이불여일타(百見而不如一打) 백번 보는것보다 한번 쳐보는게 낫다

<https://springrunner.io/training/mastering-spring-web-101-workshop/>

제 5장, HTTP 요청과 핸들러(Handler) 연결하기

HTTP 요청은 컨트롤러내에 작성된 핸들러에게 위임



핸들러(Handler)란?

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

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // ViewName 작성
        String viewName = "/WEB-INF/templates/HelloView.jsp";

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setViewName(viewName);

        return mav;
    }
}
```

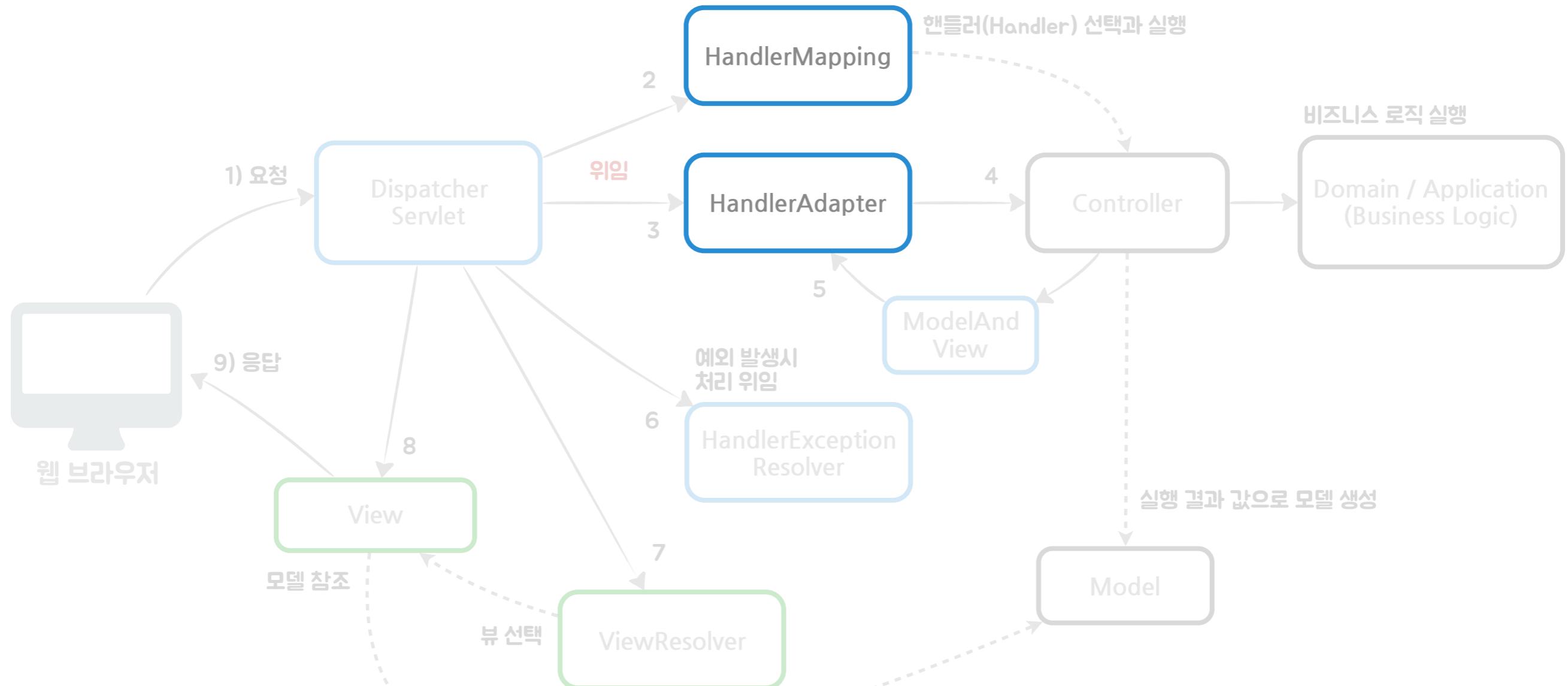
HTTP 요청을 처리할 목적으로 작성한
메소드를 핸들러라고 부른다

스프링 내부에는 핸들러를 다루기 위한
HandlerMethod 컴포넌트가 들어있다

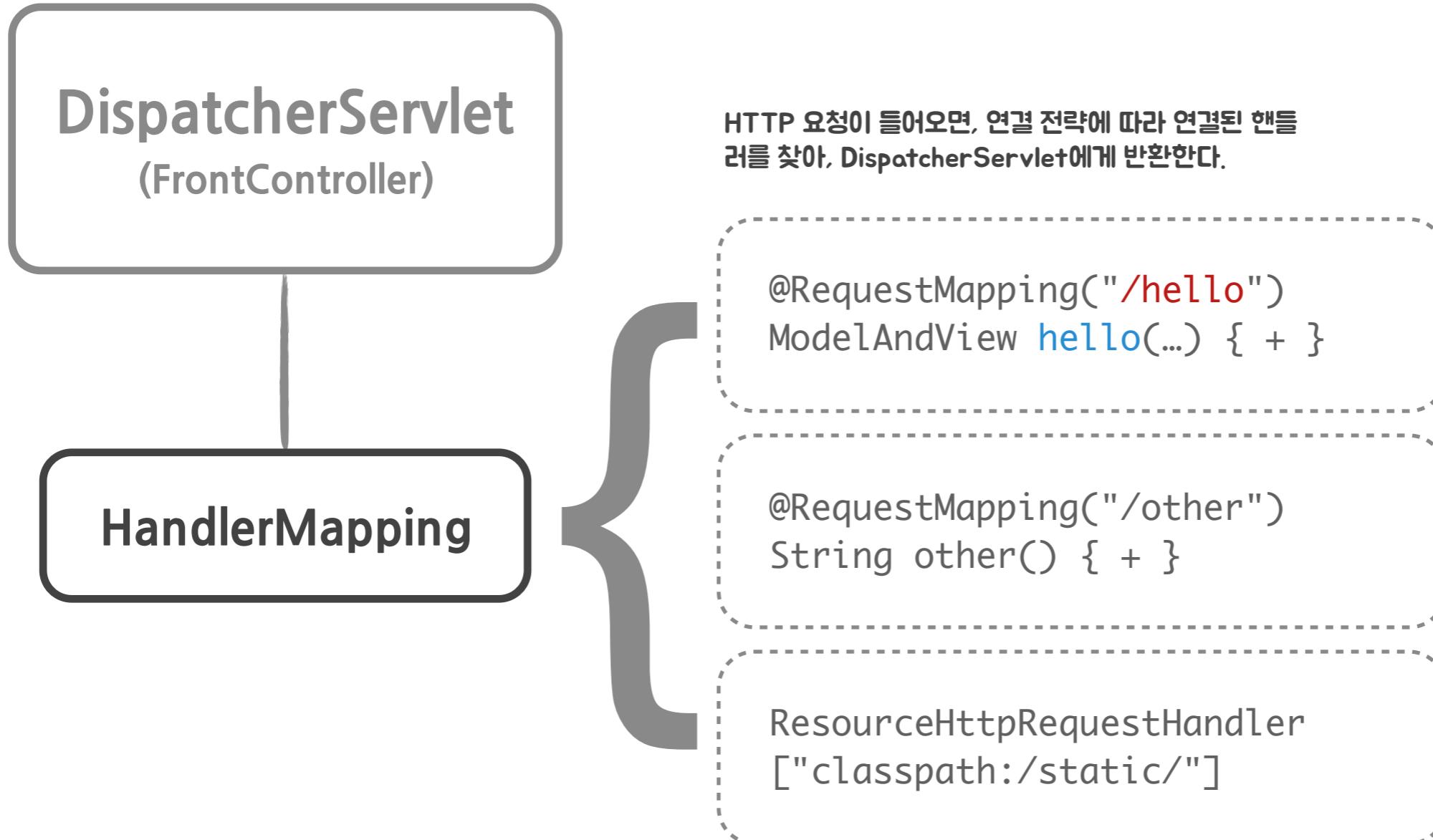
제 5장, HTTP 요청과 핸들러(Handler) 연결하기

5.1. 핸들러를 다루는 스프링 컴포넌트

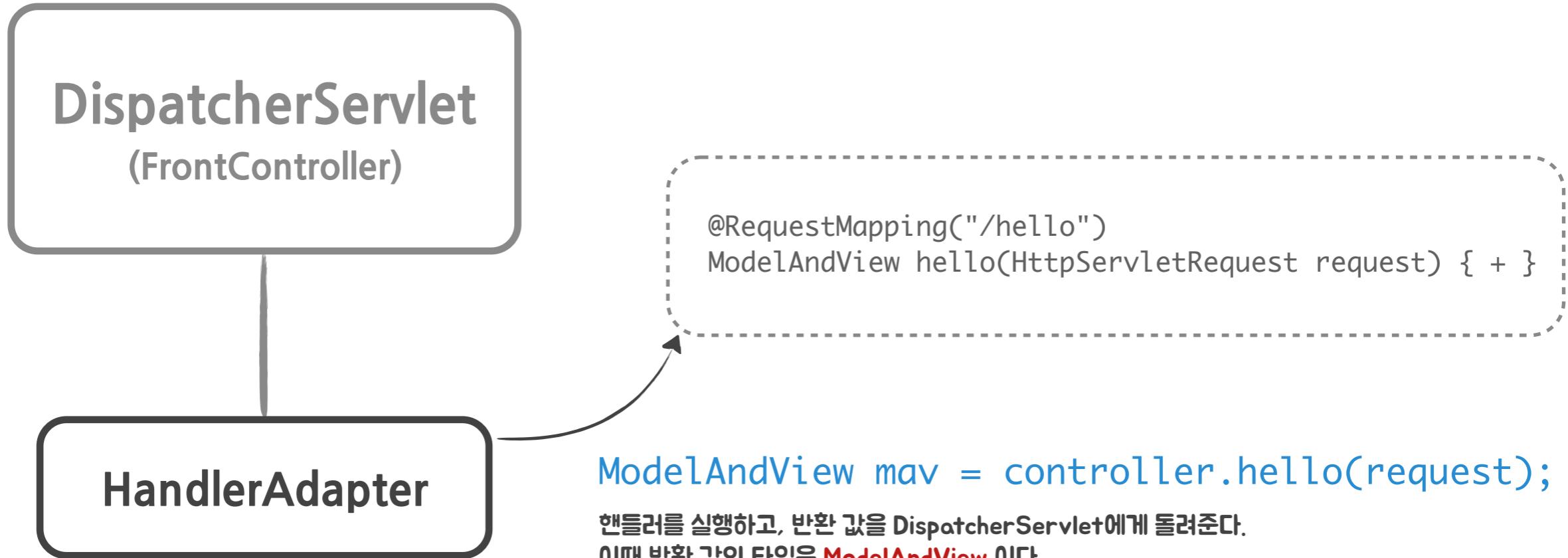
핸들러를 다루는 두 가지 컴포넌트



HTTP 요청과 핸들러에 연결하는 HandlerMapping



핸들러를 실행하고, 반환 값을 돌려주는 HandlerAdapter



제 5장, HTTP 요청과 핸들러(Handler) 연결하기

5.2, 애노테이션 기반 연결하기

@RequestMapping은 HTTP 요청과 핸들러 연결한다

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

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // ViewName 작성
        String viewName = "/WEB-INF/templates/HelloView.jsp";

        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setViewName(viewName);

        return mav;
    }
}
```

@RequestMapping 애노테이션을 사용해 핸들러와 HTTP 요청 URL을 연결 할 수 있다. URL 외에도 HTTP 메소드(Method), 헤더(Header) 등 세분화된 조건을 걸 수 있다

@RequestMapping 속성 목록

속성	설명
String[] value()	/hello 같이 핸들러에 연결할 URL을 작성 ANT 스타일의 와일드카드(*) 사용 가능
RequestMethod[] method()	HTTP 요청 메소드(HEAD, GET, POST, PUT, DELETE, OPTIONS, TRACE)를 작성 RequestMethod는 HTTP 메소드를 정의한 이늄(enum)
String[] params()	요청 데이터의 이름과 표현식 작성 표현식으로 데이터의 유무 또는 특정 값 판단이 가능
String[] headers	HTTP 헤더의 이름과 표현식 작성 표현식으로 헤더의 유무 또는 특정 값 판단이 가능
String[] consumes()	HTTP 헤더의 Content-Type 지정을 쉽게 작성
String[] produces()	HTTP 헤더의 Accept 지정을 쉽게 작성

@RequestMapping : URL 패턴 연결

```
import javax.servlet.http.HttpServletRequest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class MappingController {
```

```
    @RequestMapping(value="/mapping/path")
    public String byPath() {
        return "Mapped by path!";
    }
```

하나에 URL(<https://spring.io/mapping/path>)을 연결한다

```
    @RequestMapping({"/mapping/path/first", "/mapping/path/second"})
    public String byPaths(HttpServletRequest request) {
        return "Mapped by multiple path (" + request.getRequestURI() + ")";
    }
```

value는 기본 속성이라 생략 할 수 있고, 2개 이상의 URL을 배열로 연결 할 수 있다

```
    @RequestMapping(value="/mapping/path/*")
    public String byPathPattern(HttpServletRequest request) {
        return "Mapped by path pattern (" + request.getRequestURI() + ")";
    }
```

앤틴(Ant-style) 경로 패턴을 사용해 정의되지 않은 URL 패턴을 지정할 수 있다

? : 1개의 문자와 매칭
* : 0개 이상의 문자와 매칭
** : 0개 이상의 디렉터리와 매칭

```
}
```

@RequestMapping : URL 템플릿 연결

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

```
import org.springframework.web.bind.annotation.PathVariable;
```

```
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
```

```
public class MappingController {
```

```
    @RequestMapping("/mapping/template/{value}")
```

```
    public String byTemplate(@PathVariable("value") String value) {
```

```
        return "Mapped by template (value = '" + value + "');
```

```
}
```

URL 경로에 URL 템플릿({})을 사용 할 수 있다

/mapping/template/hello 형태로 사용

```
    @RequestMapping("/mapping/template/{value}/path")
```

```
    public String byTemplateAndPath(@PathVariable("value") String value) {
```

```
        return "Mapped by template and path (value = '" + value + "');
```

```
}
```

{ }에 들어가는 값을 경로 변수라고 부르는데, 이 값은
@PathVariable 애노테이션으로 핸들러에서 사용 할 수 있다

```
}
```

@RequestMapping : URL + HTTP 메소드 연결

```

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MappingController {

    @RequestMapping(value="/mapping/method", method=RequestMethod.GET)
    public String byGetMethod() {
        return "Mapped by path + get method";
    }

    @RequestMapping(value="/mapping/method", method=RequestMethod.POST)
    public String byPostMethod() {
        return "Mapped by path + post method";
    }

    @RequestMapping(value="/mapping/methods", method={RequestMethod.GET, RequestMethod.POST})
    public String byMethods() {
        return "Mapped by path + methods";
    }
}

```



하나의 URL에 HTTP 메소드를 달리해서 연결한다



여러개의 메소드를 배열로 연결 할 수 있다

@RequestMapping : URL + HTTP 파라미터 연결

```

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MappingController {
    @RequestMapping(value="/mapping/parameter", method=RequestMethod.GET, params="value")
    public String byParameter() {
        return "Mapped by path + method + presence of query parameter!";
    }

    @RequestMapping(value="/mapping/parameter", method=RequestMethod.GET, params="!value")
    public String byParameterNegation() {
        return "Mapped by path + method + not presence of query parameter!";
    }
}

```

/mapping/parameter?value=spring 과 같이
파라미터에 value가 존재하면 연결된다

파라미터에 value가 없을때만 연결된다

@RequestMapping : URL + HTTP 헤더 연결

```

import javax.servlet.http.HttpServletRequest;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MappingController {

    @RequestMapping(value="/mapping/header", method=RequestMethod.GET, headers="MyHeader")
    public String byHeader() {
        return "Mapped by path + method + presence of header!";
    }

    @RequestMapping(value="/mapping/header", method=RequestMethod.GET, headers="!MyHeader")
    public String byHeaderNegation() {
        return "Mapped by path + method + absence of header!";
    }
}

```

HTTP 헤더에 MyHeader가 있거나,
또는 없으면 연결된다

POST /mapping/consumes HTTP/1.1
MyHeader: examples, spring
Content-Type: application/json; charset=UTF-8
Content-Length: 15
Accept: application/json, text/plain, /*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate

{"name": "arawn"}

@RequestMapping : URL + consumes 연결

```

import javax.servlet.http.HttpServletRequest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MappingController {

    @RequestMapping(value="/mapping/consumes", consumes="application/json")
    public String byConsumes(@RequestBody JavaBean javaBean) {
        return "Mapped by path + consumable media type (javaBean '" + javaBean + "')";
    }
}

```

Content-Type 헤더 값이
"application/json" 이면 연결된다

POST /mapping/consumes HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 15
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Host:spring.io

{"name": "arawn"}

@RequestMapping : URL + produces 연결

```

import javax.servlet.http.HttpServletRequest;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MappingController {

    @RequestMapping(value="/mapping/produces", produces="application/json")
    public JavaBean byProducesJson() {
        return new JavaBean("JSON");
    }

    @RequestMapping(value="/mapping/produces", produces="application/xml")
    public JavaBean byProducesXml() {
        return new JavaBean("XML");
    }
}

```

Accept 헤더 값이
선택된 값(json, xml)과 일치하면 연결된다

POST /mapping/produces HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 15
Accept: application/json, text/plain, */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Host:spring.io

{"name": "arawn"}

@RequestMapping을 탑재(클래스) 수준에서 사용하기

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/type-mapping/**")
public class TypeMappingController {
```

```
    @RequestMapping("/path")
    public String byPath() {
        return "Mapped by path!";
    }
```

```
    @RequestMapping(value="/method", method=RequestMethod.GET)
    public String byMethod() {
        return "Mapped by path + method";
    }
```

// 파라미터(parameter), 헤더(header) 등도 동일하게 사용 가능

`@RequestMapping` 애노테이션은 탑재(클래스)에도 선언 할 수 있다.
이렇게 선언하면 컨트롤러내 모든 핸들러에게 적용된다

'/type-mapping/path'로 연결되는데, 상위 속성 값에 하위 속성 값이 추가로 붙어서 동작한다

HTTP 메소드 연결 코드를 간결하게 작성하기

```

import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value="/mapping/method-level")
public class HttpMethodMappingController {

    @GetMapping
    public String get() {
        return "Mapped by path + get method";
    }

    @PostMapping
    public String post() {
        return "Mapped by path + post method";
    }

    @PutMapping
    public String put() {
        return "Mapped by path + put method";
    }

    @PatchMapping
    public String patch() {
        return "Mapped by path + patch method";
    }

    @DeleteMapping
    public String delete() {
        return "Mapped by path + delete method";
    }
}

```

POST /mapping/method-level HTTP/1.1
Content-Type: application/json; charset=UTF-8
Content-Length: 15
Accept: application/json, text/plain, /*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Host:spring.io

{"name":"springrunner"}

@{HttpMethod}Mapping 애노테이션을 사용하면, 더 간결하게 연결 할 수 있다. 이 코드는 타입에 선언된 URL 맵핑과 메소드에 선언된 메소드 맵핑이 합해져 동작한다

메타-애노테이션으로 사용된 @RequestMapping

```

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

    /**
     * Alias for {@link RequestMapping#path}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] path() default {};

    /**
     * Alias for {@link RequestMapping#params}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] params() default {};

    /**
     * Alias for {@link RequestMapping#headers}.
     */
    @AliasFor(annotation = RequestMapping.class)
    String[] headers() default {};

    /**
     * 생략
     */
}

```

메소드 수준에 맵핑 애노테이션에는

@RequestMapping 애노테이션이 선언되어 있다

필요하다면 직접 사용자 정의 애노테이션을 작성하고,

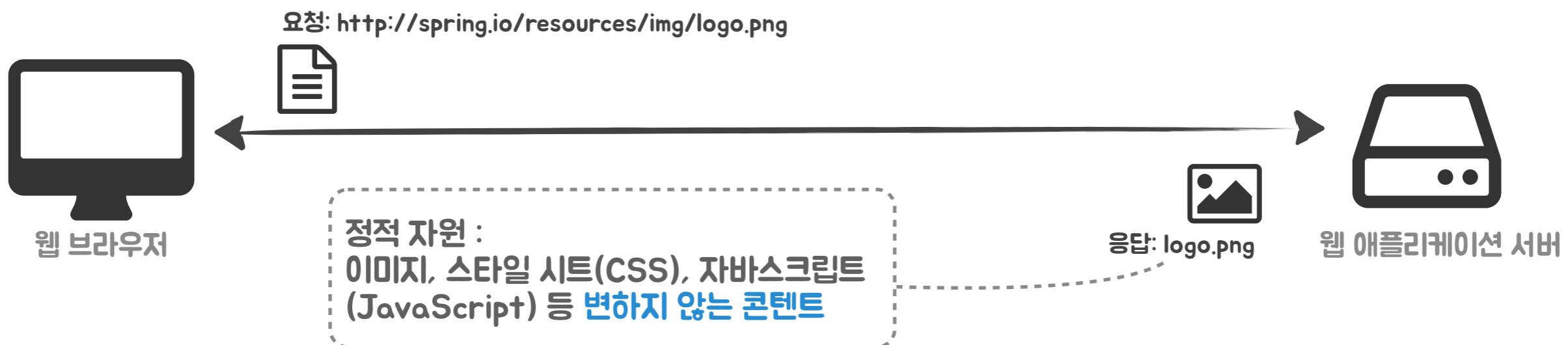
@RequestMapping을 선언하면, Spring MVC가 알아서 잘 해석한다

@AliasFor 애노테이션을 통해 메타-애노테이션에 속성을 오버라이드 할 수 있다

제 5장, HTTP 요청과 핸들러(Handler) 연결하기

5.2, 정적 자원(Static Resources) 연결하기

다시 살펴보는 정적 자원(콘텐트)



정적 자원 연결 URL과 위치 설정하기

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
```

```
@Configuration
public class ResourceMappingConfiguration implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {

        registry.addResourceHandler("정적 자원에 접근할 URL 패턴")
            .addResourceLocations("정적 자원 위치")
            .setCachePeriod(31556926);
    }
}
```

@Configuration은 이 클래스가
스프링 애플리케이션 구성 빈(Bean)임을 명시한다.

WebMvcConfigurer 인터페이스를 통해
Spring MVC 설정을 할 수 있다

정적 자원은 연결하기 위한
설정 메소드를 오버라이드 한다

정적 자원에 접근할 URL 패턴 정의하기

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ResourceMappingConfiguration implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {

        registry.addResourceHandler("/resources/**")
            .addResourceLocations("정적 자원 위치")
            .setCachePeriod(31556926);
    }
}
```

앤프(Ant-style) 경로 패턴을 사용해 정적 자원에 접근할 URL 패턴을 정의한다

? : 1개의 문자와 매칭
* : 0개 이상의 문자와 매칭
** : 0개 이상의 디렉터리와 매칭

http://spring.io/resources/css/default.css
http://spring.io/resources/js/spring-by-pivotal.png
http://spring.io/resources/img/spring-by-pivotal.png

정적 자원 위치 설정하기

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ResourceMappingConfiguration implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("static/", "classpath:static/", "file:./files/static/")
            .setCachePeriod(31556926);
    }
}
```

서블릿 자원을 포함 클래스패스, 파일 등 여러 위치를 설정 할 수 있다

http://spring.io/resources/css/default.css

```
├── WEB-INF
│   └── classes
│       └── static
└── public
    └── css
        └── default.css
```

효율적인 정적 자원 제공을 위한 캐시 설정

```

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ResourceMappingConfiguration implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {

        registry.addResourceHandler("/resources/**")
            .addResourceLocations("static/", "classpath:static/", "file:./files/static/")
            .setCachePeriod(31556926);
    }
}

```

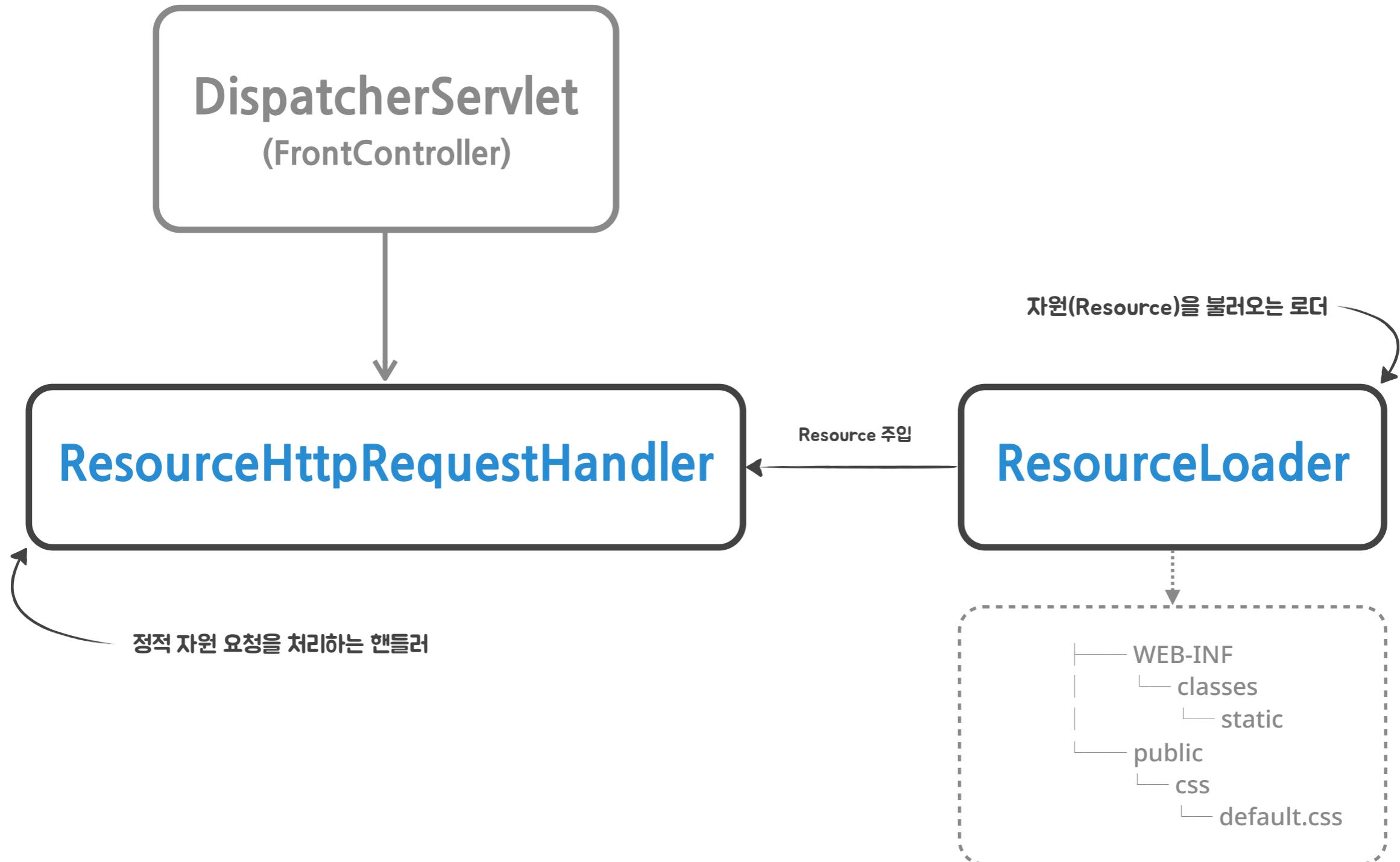
브라우저 캐시를 초 단위로 설정 할 수 있다. 보편적으로 1년을 권장한다

Response headers :

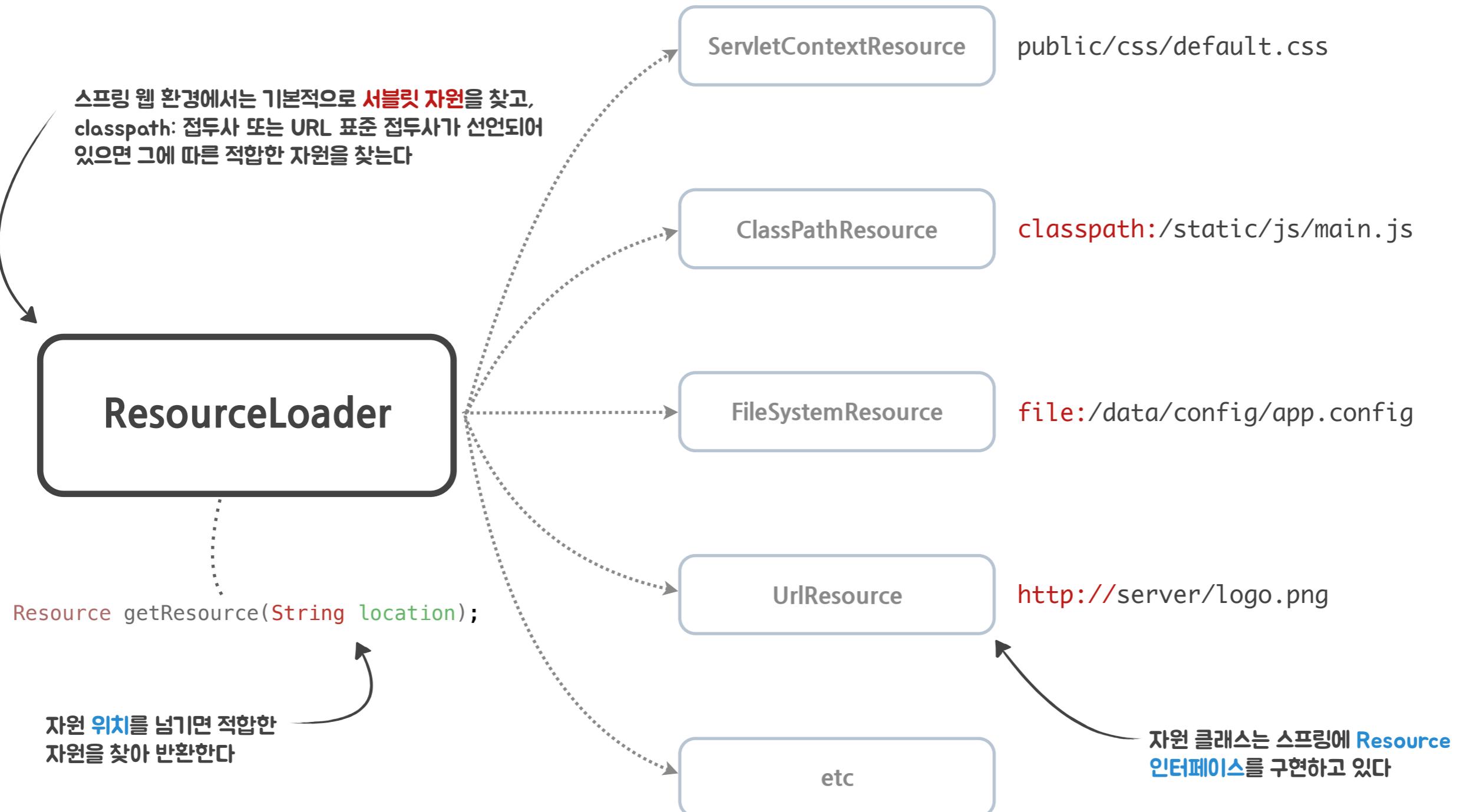
Cache-Control: "max-age=31556926, must-revalidate"
Expires: "Sun, 16 Nov 2019 07:39:20 GMT"
Last-Modified: "Thu, 20 Nov 2018 10:49:18 GMT"

핸들러가 Last-Modified 헤더도 적절히 평가 하며, 필요에 따라 304 상태코드를 반환한다

정적 자원을 처리하는 두 가지 핵심 컴포넌트

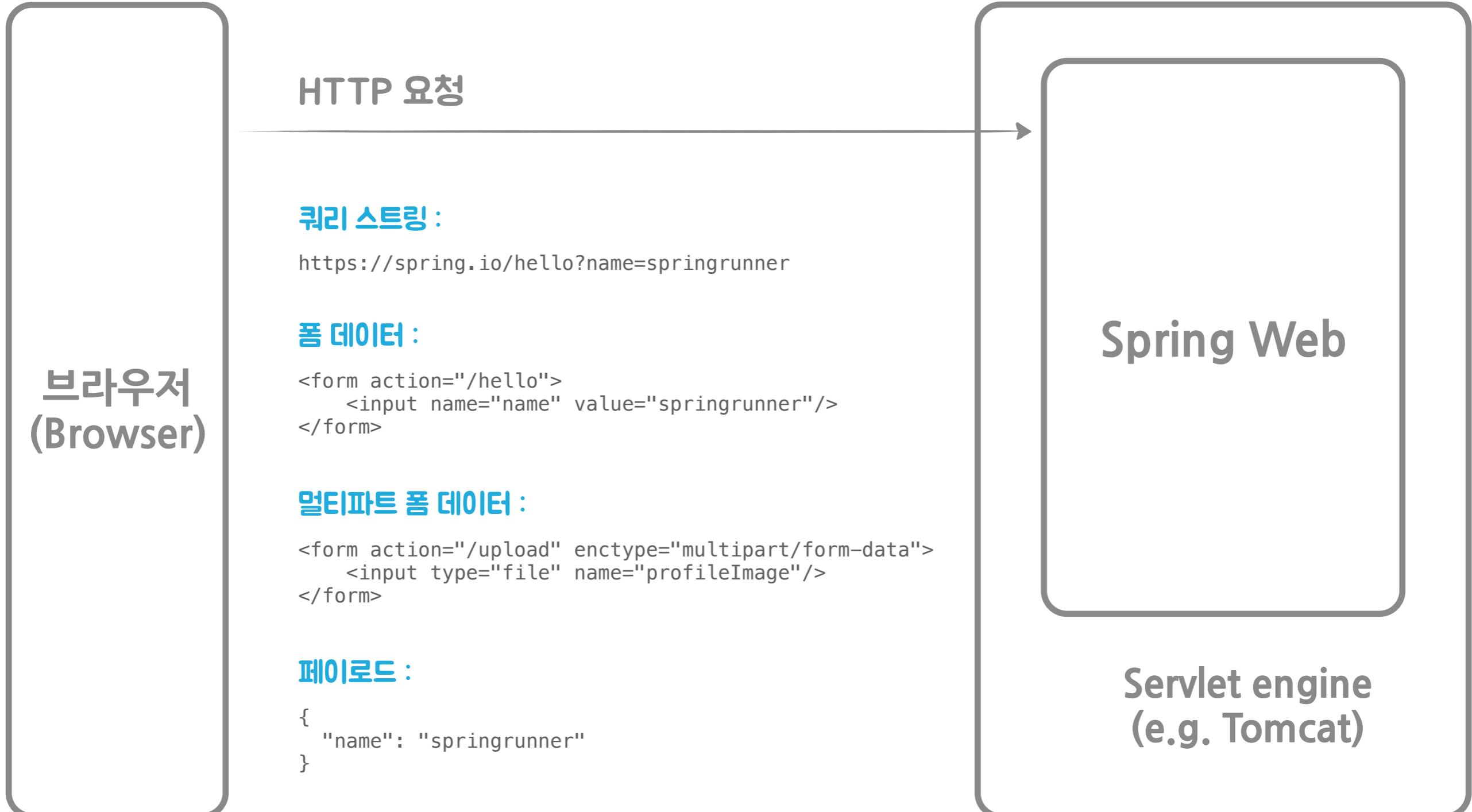


자원(Resource)을 찾아주는 ResourceLoader 컴포넌트



제 6장, **핸들러로 요청 데이터 다루기**

요청 데이터: 클라이언트가 전송한 데이터



메소드 파라미터(MethodParameter)

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

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // ViewName 작성
        String viewName = "/WEB-INF/templates/HelloView.jsp";

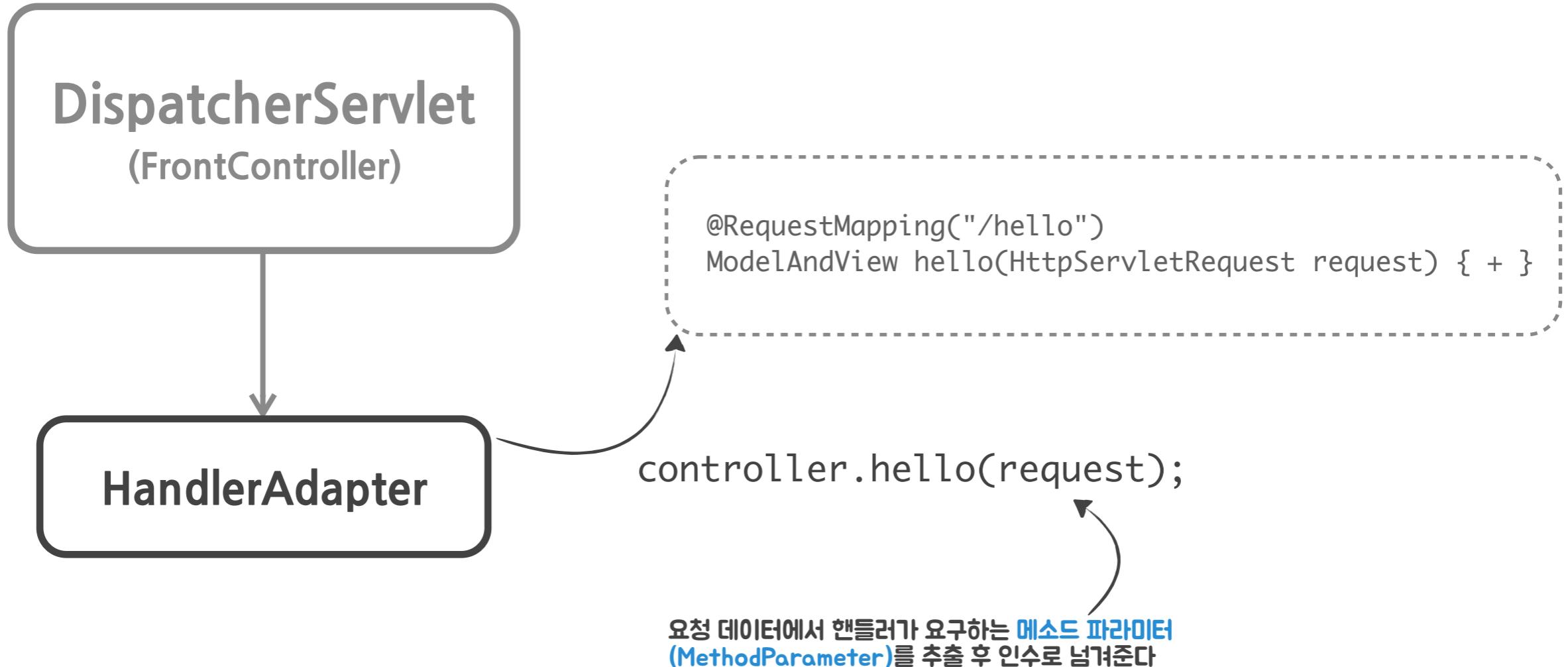
        // ModelAndView 생성 및 초기화
        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setViewName(viewName);

        return mav;
    }
}
```

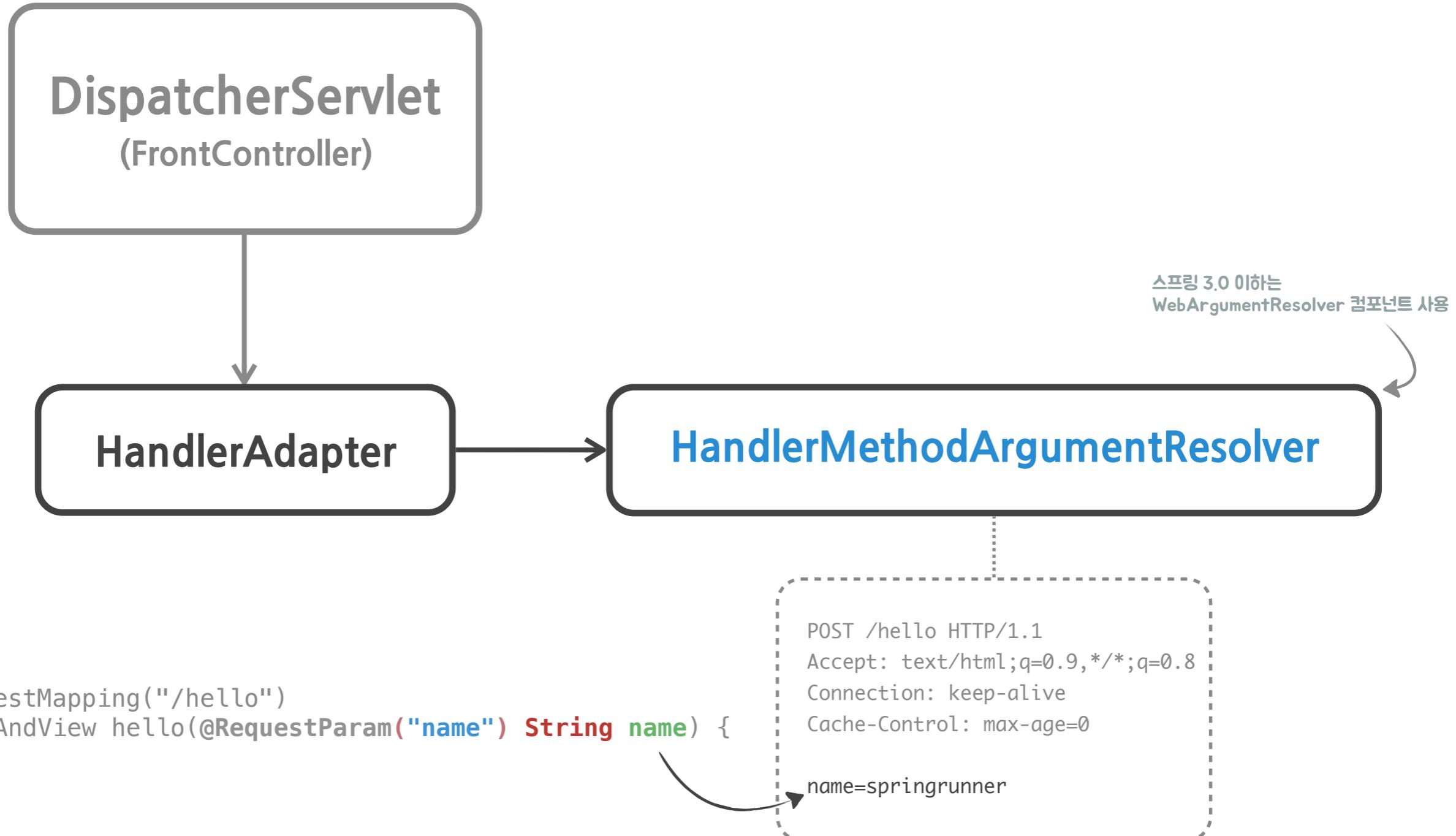
핸들러에 선언된 인자를 **메소드 파라미터**(MethodParameter)라고 부른다

핸들러는 메소드 파라미터를 통해
요청 데이터를 다룰 수 있다

핸들러를 실행할때 HandlerAdapter가 해주는 일



메소드 파라미터에 대한 요청 데이터를 추출하는 컴포넌트



제 6장, **핸들러로 요청 데이터 다루기**

6.1, **애노테이션 기반 요청 데이터 다루기**

단일 요청 데이터 얻기

GET <https://springrunner.io/data/param?foo=bar>

```
import org.springframework.web.bind.annotation.RequestParam;

@RestController
@RequestMapping("/data")
public class RequestDataController {

    @GetMapping(value="param")
    public String withParam(@RequestParam("foo") String foo) {
        return "Obtained 'foo' query parameter value '" + foo + "'";
    }

    @GetMapping(value="param/no-name")
    public String withParamByName(@RequestParam String foo) {
        return "Obtained 'foo' query parameter value '" + foo + "'";
    }

    @GetMapping(value="param/no-request-param")
    public String withParamByNoRequestParam(String foo) {
        return "Obtained 'foo' query parameter value '" + foo + "'";
    }

}
```

다중 요청 데이터 얻기

```
<form action="/data/group">
    <input name="param1" value="Spring MVC"/>
    <input name="param2" value="Juergen Hoeller"/>
    <input name="param3" value="Rossen Stoyanchev"/>
</form>
```

```
@RestController
@RequestMapping("/data")
public class RequestDataController {

    @GetMapping(value="group")
    public String withParamGroup(JavaBean bean) {
        return "Obtained parameter group " + bean;
    }

    public class JavaBean {

        private String param1;
        private String param2;
        private String param3;

        // getter, setter 메소드
    }
}
```

요청 URL에서 경로 변수 얻기

GET <https://springrunner.io/data/path/hello>

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.MatrixVariable;

@RestController
@RequestMapping("/data")
public class RequestDataController {

    @GetMapping(value="path/{var}")
    public String withPathVariable(@PathVariable("var") String var) {
        return "Obtained 'var' path variable value '" + var + "'";
    }

    @GetMapping(value="{path}/simple")
    public String withMatrixVariable(@PathVariable String path, @MatrixVariable String foo) {
        return "Obtained matrix variable 'foo=' + foo + '' from path segment '" + path + "'";
    }
}
```

HTTP 요청시 전송된 헤더와 쿠키 데이터 얻기

```
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.CookieValue;

@RestController
@RequestMapping("/data")
public class RequestDataController {

    @GetMapping(value="header")
    public String withHeader(@RequestHeader String Accept) {
        return "Obtained 'Accept' header '" + Accept + "'";
    }

    @GetMapping(value="cookie")
    public String withCookie(@CookieValue String foo) {
        return "Obtained 'foo' cookie '" + foo + "'";
    }

}
```

HTTP 요청 바디(body) 얻기

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.http.HttpEntity;

@RestController
@RequestMapping("/data")
public class RequestDataController {

    @PostMapping(value="body")
    public String withBody(@RequestBody String body) {
        return "Posted request body '" + body + "'";
    }

    @PostMapping("entity")
    public String withEntity(HttpEntity<String> entity) {
        return "Posted request body '" + entity.getBody() + "'"; headers = " + entity.getHeaders();
    }

}
```

```
InputStream is = request.getInputStream();
StringBuilder body = new StringBuilder();

try(BufferedReader br = new BufferedReader(new InputStreamReader(is))) {
    String line;
    while((line = br.readLine()) != null) {
        body.append(line);
    }
}

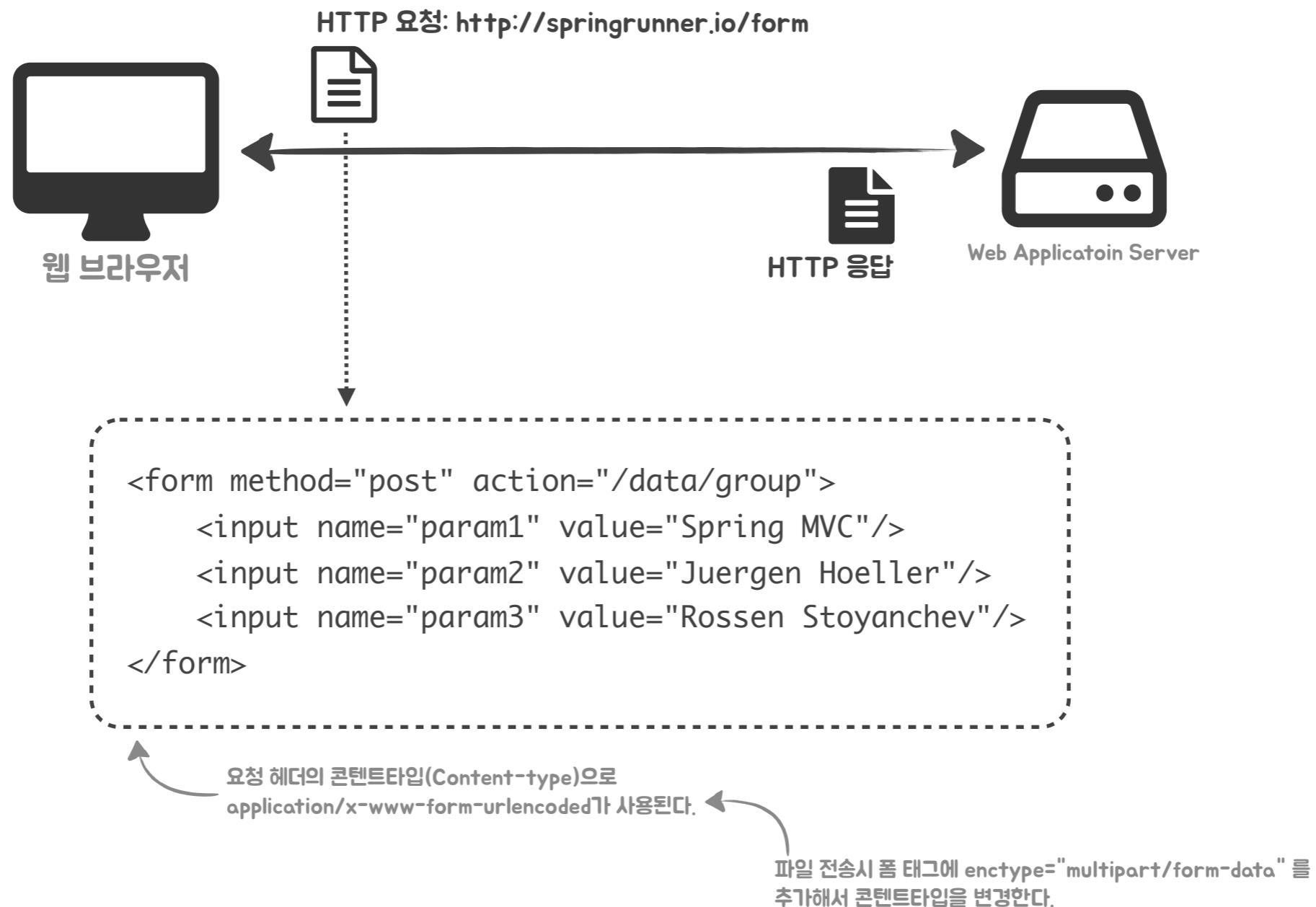
body.toString()
```

제 6장, **핸들러로 요청 데이터 다루기**

6.2, 폼(form) 데이터 다루기

HTML 폼(form)

- ✓ HTML 폼은 사용자가 웹 서버로 데이터를 전송하는 기술 중 하나이다.
- ✓ 일반적으로 데이터는 웹 서버로 전송되지만 웹페이지가 데이터를 사용하기 위하여 사용할 수도 있다.



폼(form) 요청 데이터 얻기

```
<form method="post" action="/form">
    <input name="name" value="SpringRunner"/>
    <input name="age" value="1"/>
    <input name="birthDate" value="2018-12-20"/>
</form>
```

```
@Controller
@RequestMapping("/form")
public class FormController {

    @PostMapping
    public String processSubmit(FormBean formBean) {
        // 생략
    }

    public class FormBean {

        private String name;
        private int age;

        @DateTimeFormat(iso = ISO.DATE)
        private Date birthDate;

        // getter, setter 메소드
    }
}
```

멀티파트 폼(multipart/form-data) 요청 데이터 얻기 [파일 업로드]

```
<form method="post" action="/form" enctype="multipart/form-data">
    <input name="file"/>
</form>
```

```
import org.springframework.web.multipart.MultipartFile;
import javax.servlet.http.Part;
```

```
@Controller
@RequestMapping("/fileupload")
public class FileUploadController {
```

파일 업로드는 Post 연결을 사용해야 한다

```
@PostMapping
public ModelAndView processMultipartFileUpload(@RequestParam("file") MultipartFile file) {
    // 생략
}
```

업로드된 파일은 핸들러에서
MultipartFile 타입으로 얻을 수 있다

@RequestParam은 생략 할 수 있다

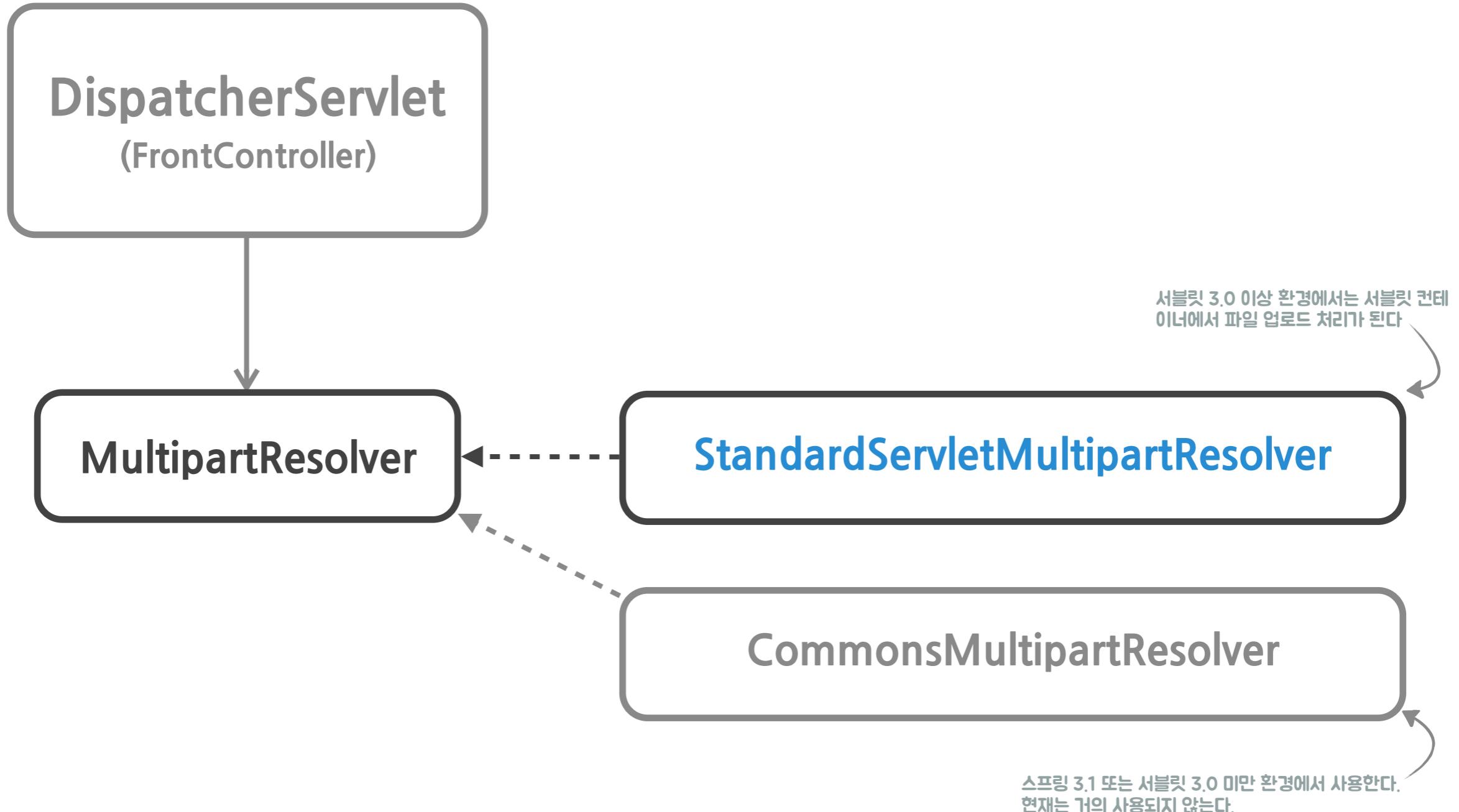
```
@PostMapping
public ModelAndView processPartUpload(Part file) {
    // 생략
}
```

서블릿 3.0 이상 사용시 Part 타입으로 파일을 획득 할 수 있다

}

List<Part> 또는 List<MultipartFile> 형태로
여러개의 파일도 다룰 수 있다.

멀티파트 폼 요청을 처리하는 컴포넌트

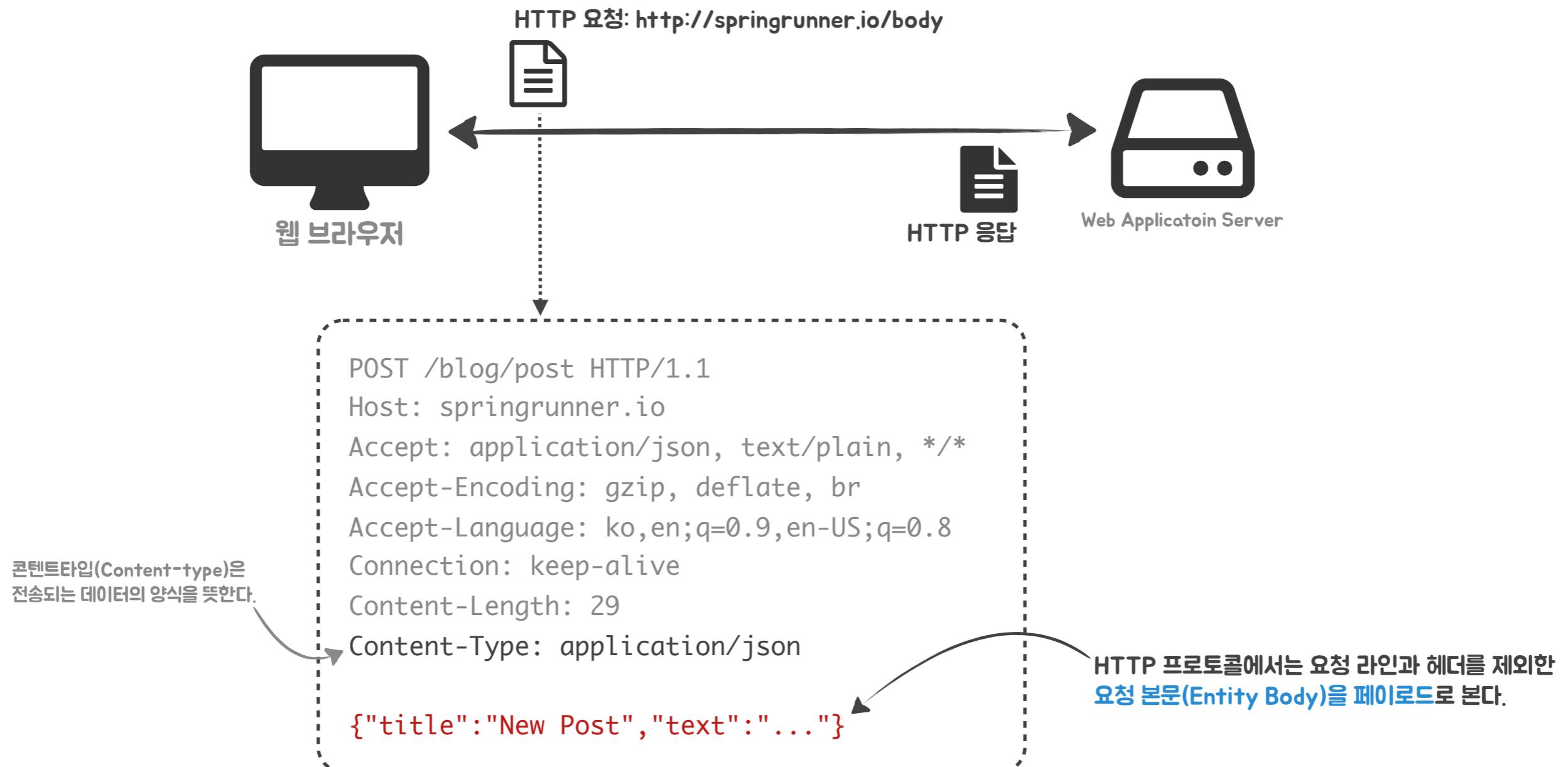


제 6장, **핸들러로 요청 데이터 다루기**

6.3, **요청 본문(Entity Body) 데이터 다루기**

요청 본문(Entity body)

- ✓ HTTP 프로토콜에서 서버로 데이터를 전송시 전달되는 데이터이다.
- ✓ 전송하고자 하는 목적인 데이터 자체를 의미하는 페이로드(Payload)라 부르기도 한다.



요청 본문 데이터 얻기

```
import org.springframework.web.bind.annotation.RequestBody;

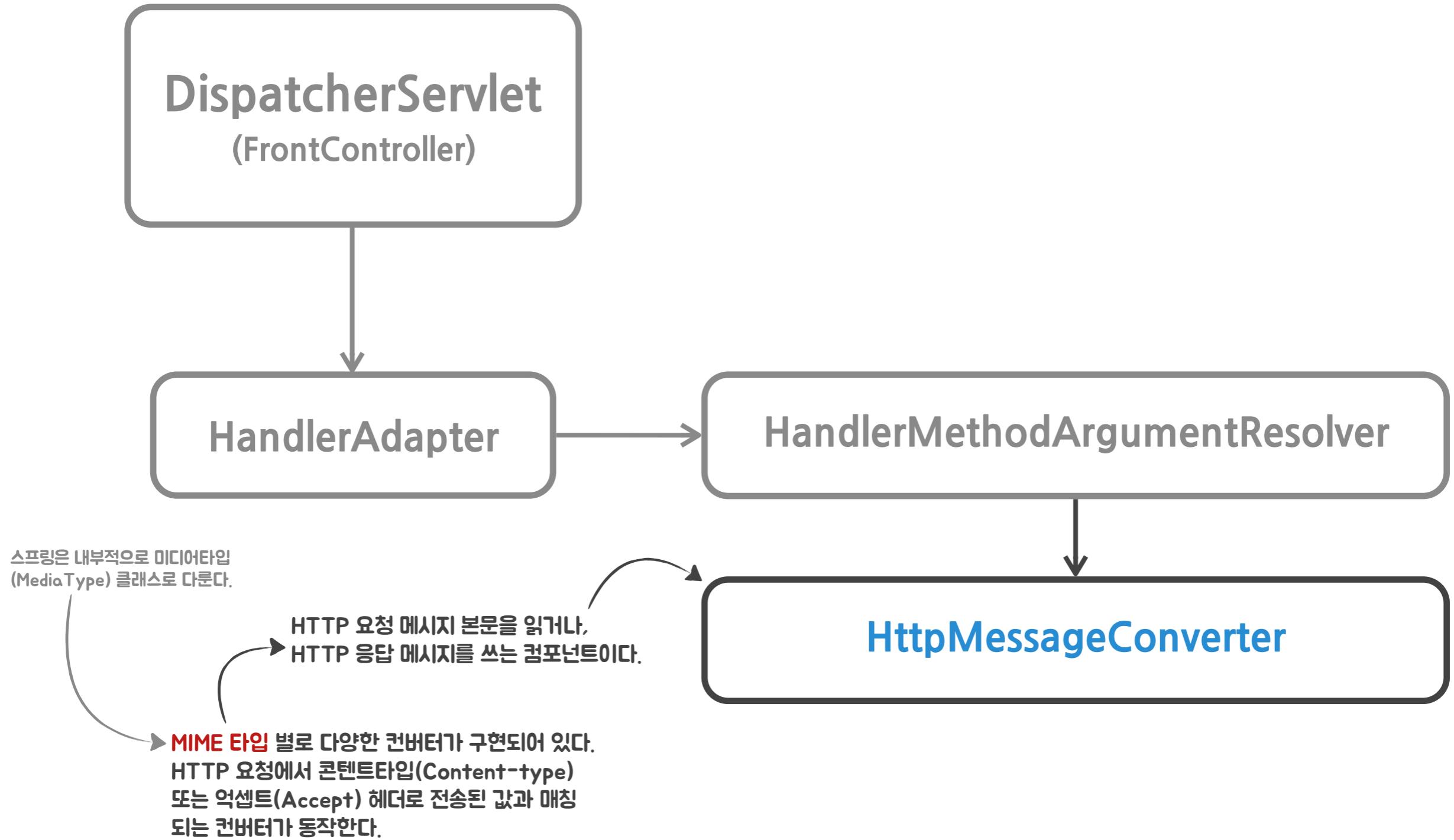
@RestController
@RequestMapping("/messageconverters")
public class MessageConvertersController {

    @PostMapping("/string")
    public String readString(@RequestBody String string) {
        return "Read string " + string + "";
    }

    @PostMapping(value = "/json", consumes = "application/json")
    public String readJson(@RequestBody JavaBean bean) {
        return "Read from JSON: " + bean;
    }

    @PostMapping(value = "/xml", consumes = "application/xml")
    public String readXml(HttpEntity<JavaBean> entity) {
        return "Read from XML: " + entity.getBody();
    }
}
```

요청 본문 데이터를 처리하는 컴포넌트



제 6장, **핸들러로 요청 데이터 다루기**

6.4. 스프링이 지원하는 메소드 파라메타 타입

핸들러에서 서블릿 API 객체 얻기

```
@RestController
public class StandardArgumentsController {

    // request related

    @GetMapping("/data/standard/request")
    public String standard(HttpServletRequest request, Principal principal, Locale locale) {
        return "request = " + request + ", " + "principal = " + principal + ", " + "locale = " + locale;
    }

    // response related

    @GetMapping("/data/standard/response")
    public String standard.HttpServletResponse response) {
        return "response = " + response;
    }

    // HttpSession

    @GetMapping("/data/standard/session")
    public String standard(HttpSession session) {
        return "session = " + session;
    }

}
```

핸들러 메소드 파라미터로 얻을 수 있는 타입

타입	설명
HttpServletRequest, HttpServletResponse	서블릿 컨테이너가 생성한 서블릿 요청/응답 개체 (ServletRequest, ServletResponse 타입으로도 받을 수 있다)
WebRequest, NativeWebRequest	HTTP 요청 및 응답 개체 (서블릿 또는 포틀릿 스펙에 종속적이지 않은 범용 인터페이스다)
HttpHeader	HTTP 헤더 개체
HttpEntity	HTTP 요청 헤더와 바디를 받아온다
HttpSession	HTTP 세션 개체
SessionStatus	@SessionAttribute로 저장된 모델 오브젝트에 상태를 조작할 수 있는 개체
Map, Model, ModelMap	뷰에 전달할 모델 오브젝트를 담아둘 수 있는 개체
RedirectAttributes	리다이렉트시 전달할 모델 오브젝트를 담아 전달할 수 있는 개체
InputStream, Reader	서블릿 요청(ServletRequest)에 담긴 InputStream, Reader 개체
OutputStream, Writer	서블릿 응답(ServletResponse)에 담긴 OutputStream, Writer 개체
Principal	인증된 사용자 개체 (서블릿 스펙에 맞게 구현된 기능이 있어야 올바른 값이 넘어온다)
Locale	지역정보 리졸버(Locale Resolver)가 결정한 로케일 개체
Errors, BindingResult	커맨드 오브젝트에 대한 데이터 바인딩 오류 또는 데이터 검증(Validation) 결과 개체

제 6장, **핸들러로 요청 데이터 다루기**

6.5. 사용자 정의 타입 얻기

■ 비공개

제 6장, 핸들러로 요청 데이터 다루기

6.6. 데이터 타입 변환

■ 비공개

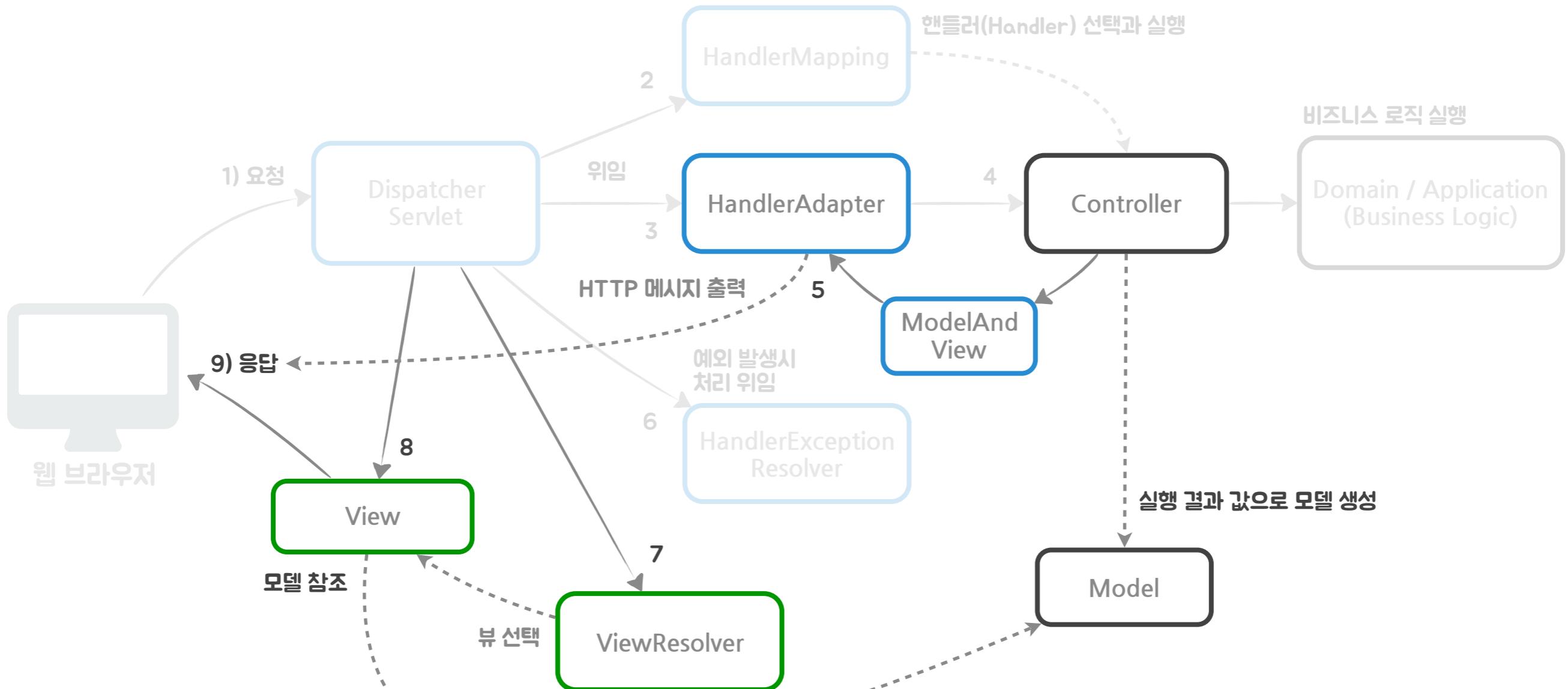
제 6장, 핸들러로 요청 데이터 다루기

6.7. 데이터 유효성 검사

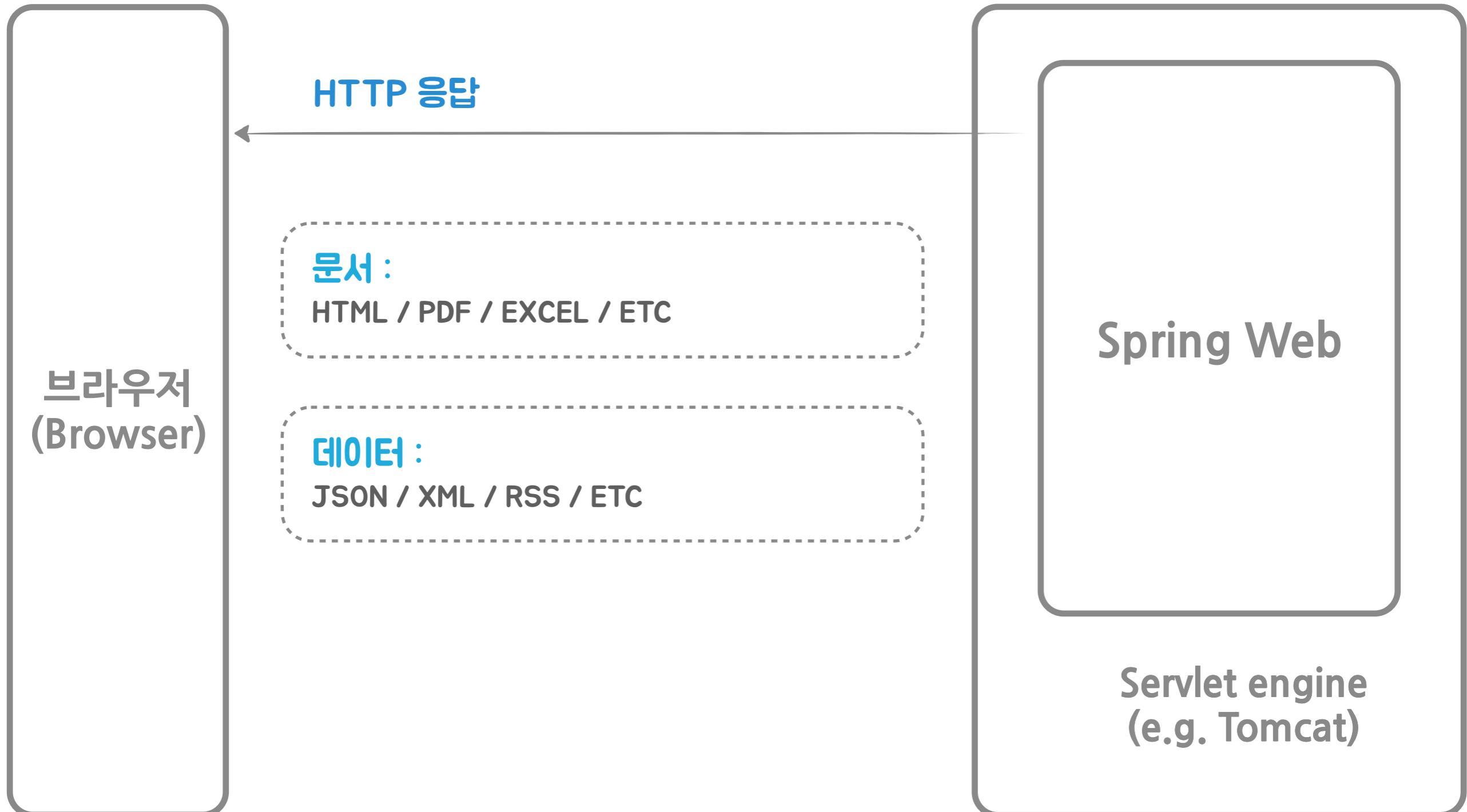
■ 비공개

제 7장, HTTP 응답 콘텐트 만들기

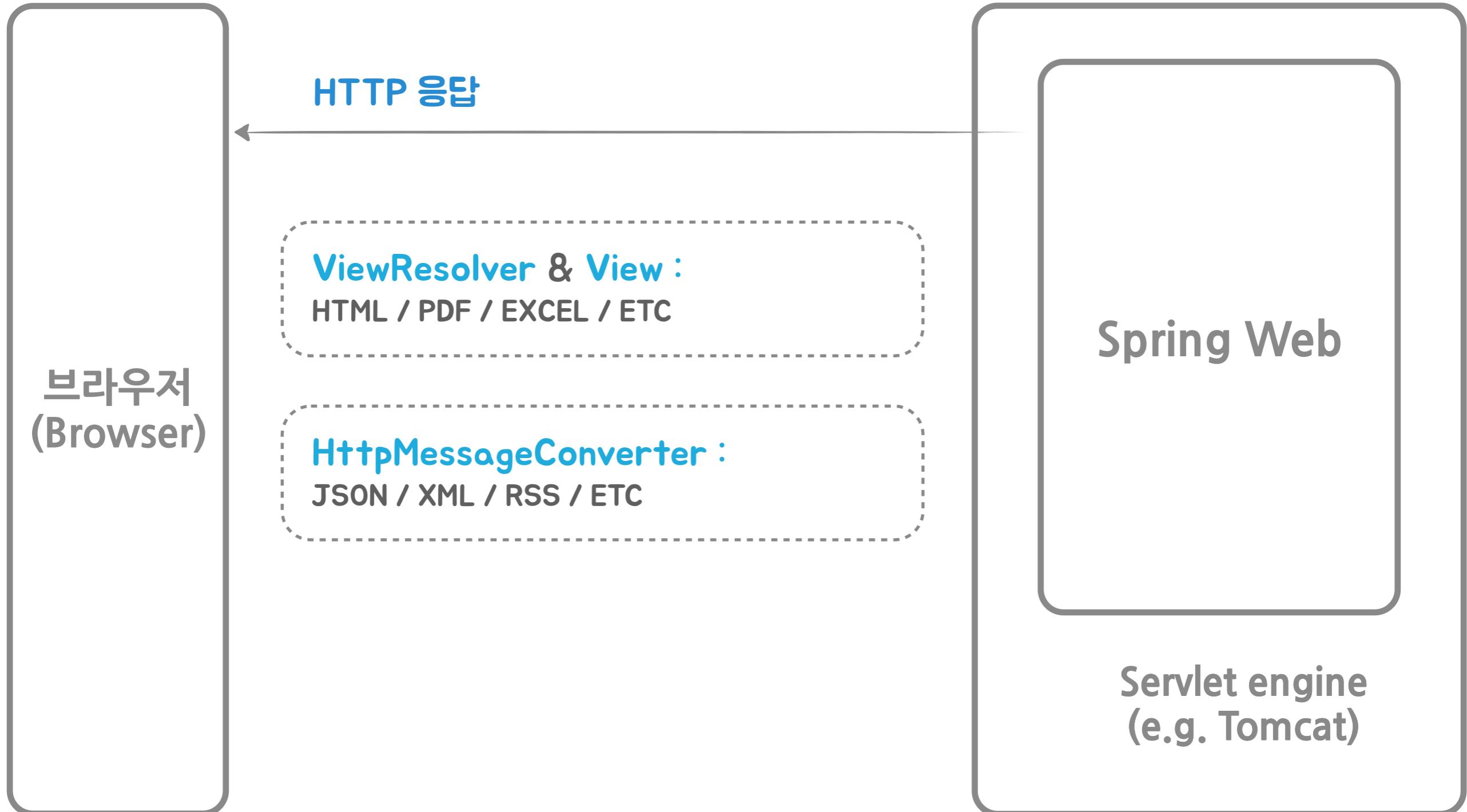
핸들러 반환 값은 HTTP 응답 콘텐트로 사용



두 가지 유형의 HTTP 응답 콘텐트



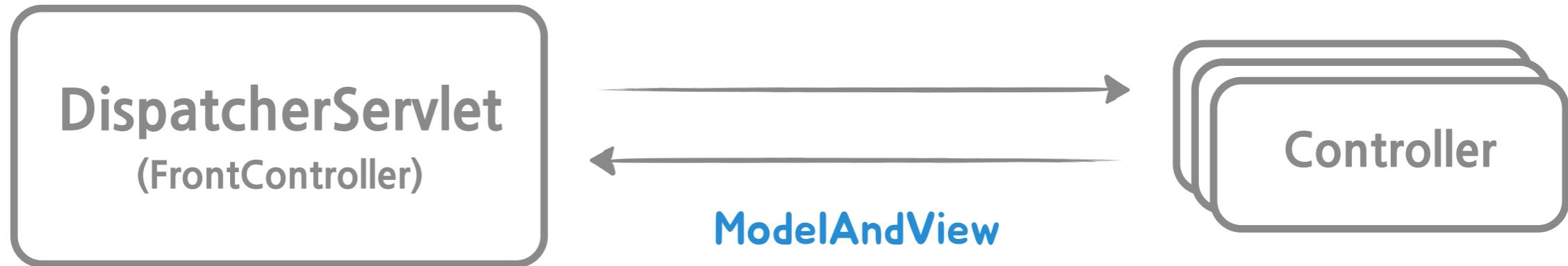
HTTP 응답을 처리하는 세가지 컴포넌트



제 7장, HTTP 응답 콘텐트 만들기

7.1, View & ViewResolver

디스패처서블릿은 모델과 뷰(ModelAndView)로 응답 콘텐트 생성



```

public class DispatcherServlet {

    void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        // 생략

        ModelAndView mv = handlerAdapter.handle(processedRequest, response, handler);
        핸들러는 ModelAndView 객체를 반환한다.

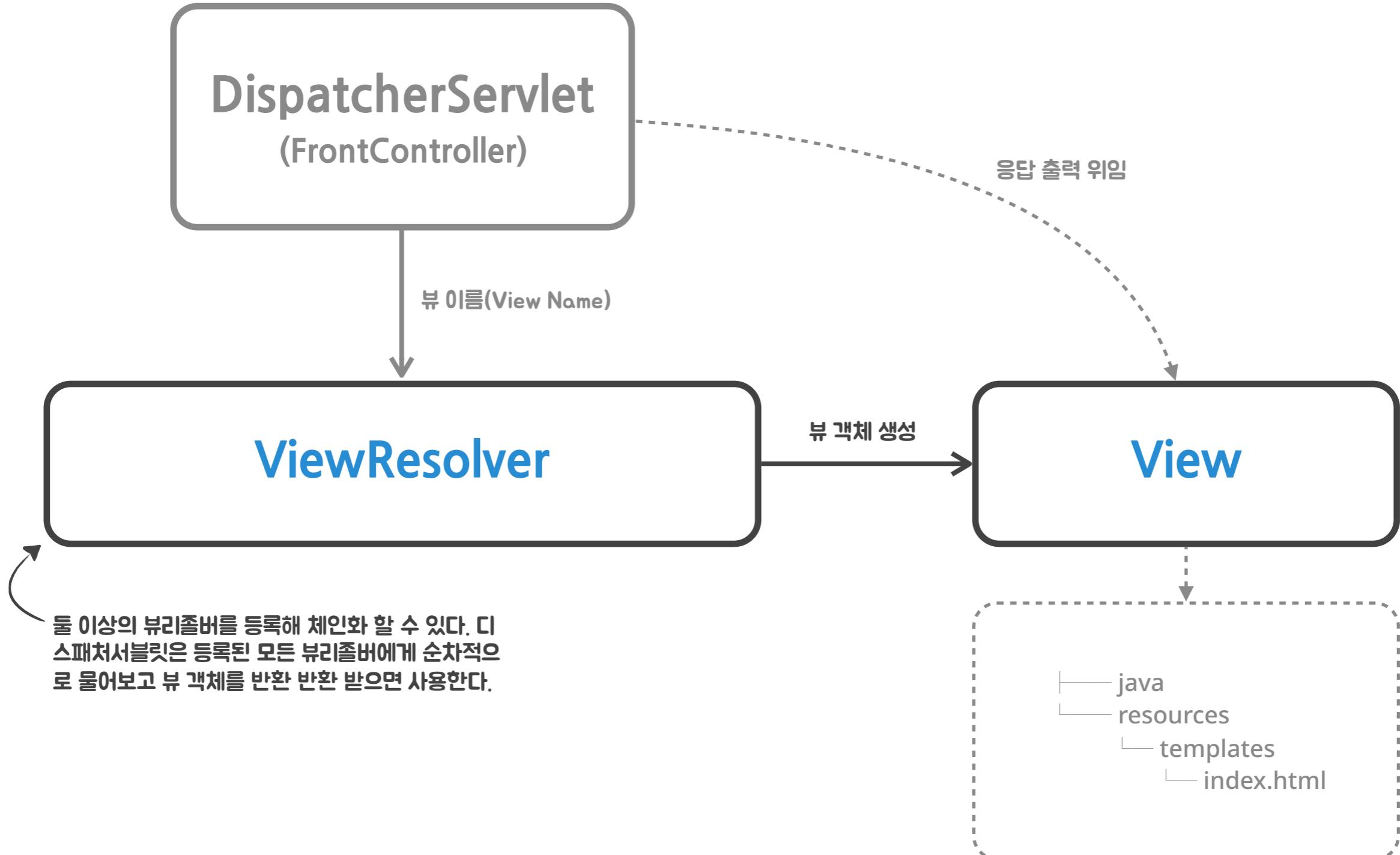
        View view;
        String viewName = mv.getViewName();
        if (viewName != null) {
            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
        } else {
            view = mv.getView();
        }

        view.render(mv.getModelInternal(), request, response);
        뷰 객체는 HTTP 응답을 출력한다.
    }
}
  
```

Annotations with arrows pointing to specific code snippets:

- An arrow points to the line `mv = handlerAdapter.handle(processedRequest, response, handler);` with the text "핸들러는 ModelAndView 객체를 반환한다."
- An arrow points to the line `view = resolveViewName(viewName, mv.getModelInternal(), locale, request);` with the text "뷰 이름이 지정되어 있으면 뷰리졸버 (ViewResolver)를 통해 뷰 객체를 얻는다."
- An arrow points to the line `view.render(mv.getModelInternal(), request, response);` with the text "뷰 객체는 HTTP 응답을 출력한다."

뷰리졸버는 뷰 객체를 생성하고, 뷰는 응답을 출력



뷰와 뷰리졸버로 응답 출력하기 (1/2)

```
import org.springframework.web.servlet.ModelAndView;  
  
@Controller  
@RequestMapping("/views/*")  
public class RenderingViewsController {  
  
    @GetMapping("text")  
    public ModelAndView simple() {  
  
        ModelAndView mv = new ModelAndView();  
        mv.addObject("foo", "bar");  
        mv.addObject("fruit", "apple"); ←  
        mv.setViewName("text");  
    }  
}
```

ModelAndView 생성 및 모델 구성, 뷰 이름 지정

뷰와 뷰리졸버로 응답 출력하기 (2/2)

```

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.View;
import org.springframework.web.servlet.ViewResolver;

@Configuration
public class RenderingViewsConfiguration implements WebMvcConfigurer {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.viewResolver(new SimpleMappingViewResolver());
    }
}

static class TextView implements View {
    public void render(Map<String, ?> model, HttpServletRequest req, HttpServletResponse res) {
        // HttpServletResponse를 통해 응답 출력
    }
}

static class SimpleMappingViewResolver implements ViewResolver, Ordered {

    public View resolveViewName(String viewName, Locale locale) throws Exception {
        if (Objects.equals("text", viewName)) {
            return new TextView();
        }
        return null;
    }
}

```

@Configuration은 이 클래스가 스프링 애플리케이션 구성 빈(Bean)임을 명시한다.

WebMvcConfigurer 인터페이스를 통해 Spring MVC 설정을 할 수 있다

뷰리졸버를 등록한다.

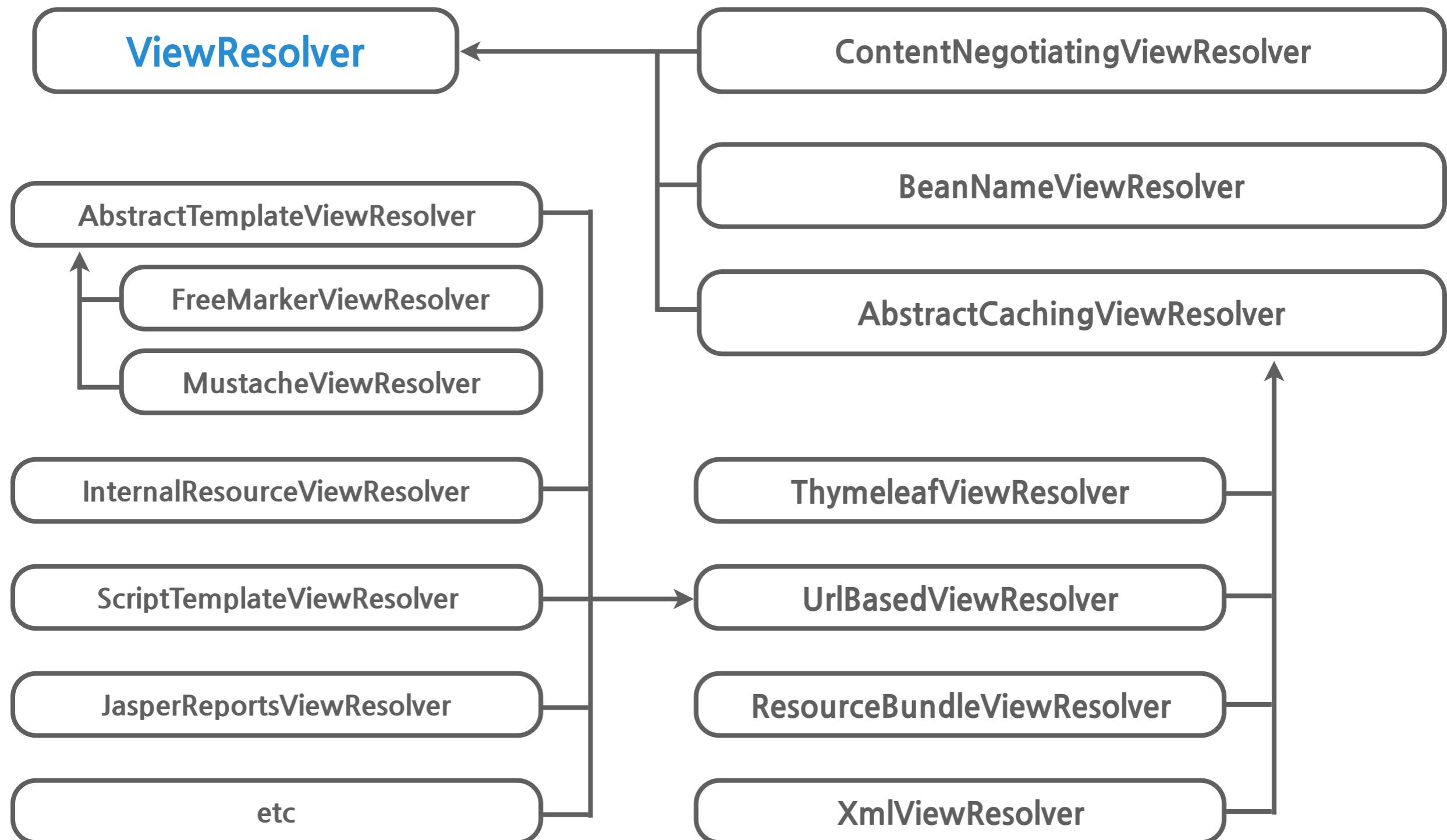
여러개의 뷰리졸버를 사용한다면 Ordered 인터페이스로 순서를 지정할 수 있다.

뷰리졸버는 뷰 이름을 보고 처리 가능한 뷰 객체 생성한다.

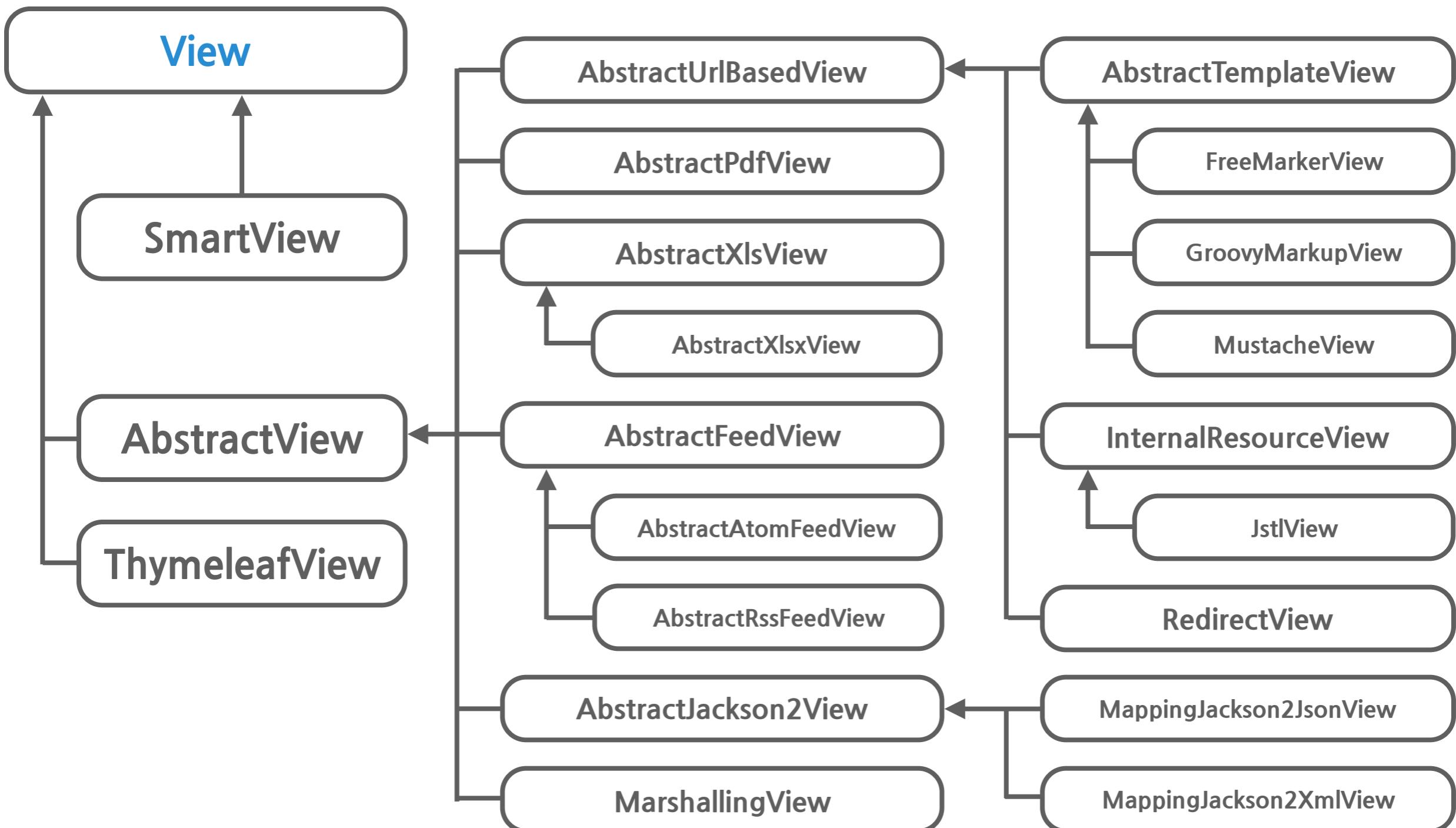
처리 할 수 없다면 널(null) 반환한다.

ModelAndView mv = new ModelAndView();
mv.setViewName("text");

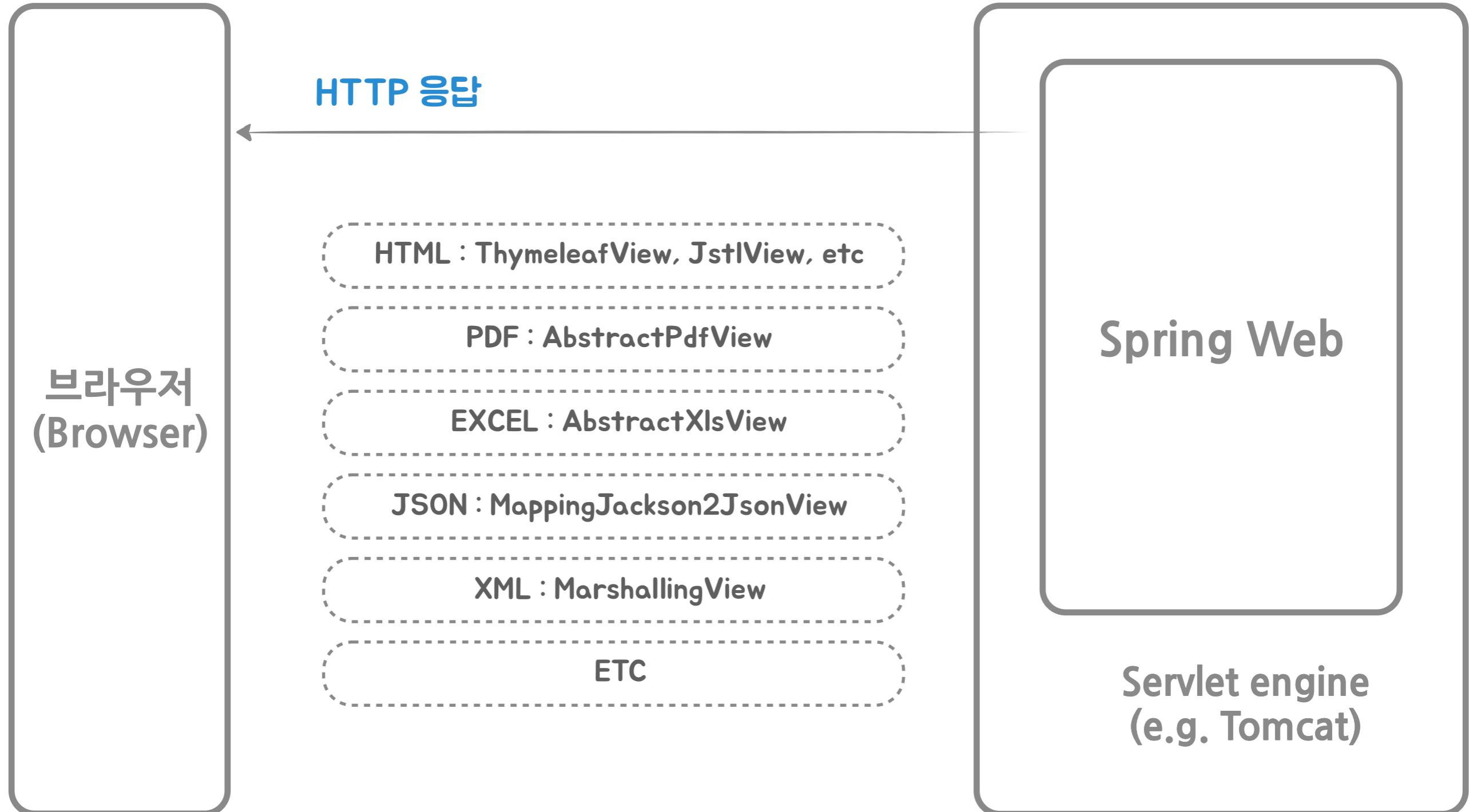
ViewResolver Hierarchy



View Hierarchy



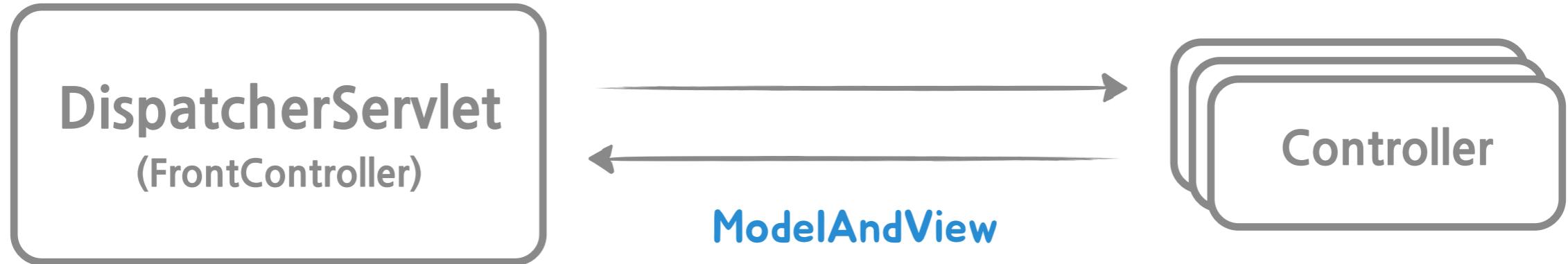
뷰와 뷰리졸버를 구성해 다양한 응답 콘텐트를 생성



제 7장, HTTP 응답 콘텐트 만들기

7.2, 모델과 뷰(ModelAndView)

디스패처서블릿은 모델과 뷰(ModelAndView)를 다룬다



```

public class DispatcherServlet {

    void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        // 생략

        ModelAndView mv = handlerAdapter.handle(processedRequest, response, handler);

        View view;
        String viewName = mv.getViewName();
        if (viewName != null) {
            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
        } else {
            view = mv.getView();
        }

        view.render(mv.getModelInternal(), request, response);
    }
}
  
```

핸들러는 ModelAndView 객체를 반환한다.

ModelAndView를 손쉽게 다루는 방법들

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

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name) {
        // Model 생성
        HelloModel model = new HelloModel(name);

        // ViewName 지정
        String viewName = "hello";

        ModelAndView mav = new ModelAndView();
        mav.addObject("hello", model);
        mav.setViewName(viewName);

        return mav;
    }
}
```

모델 오브젝트(Model / Map / ModelMap)

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello(@RequestParam("name") String name, Model models) {
        // Model 생성 및 추가
        HelloModel model = new HelloModel(name);
        models.addAttribute("hello", model);

        // ViewName 지정
        String viewName = "hello";

        ModelAndView mav = new ModelAndView();
        mav.setViewName(viewName);

        return mav;
    }
}
```

논리적인 뷰 이름 반환하기

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public String hello(@RequestParam("name") String name, Model models) {
        // Model 생성
        HelloModel model = new HelloModel(name);
        models.addAttribute("hello", model);

        // ViewName 지정
        String viewName = "hello";

        // ModelAndView mav = new ModelAndView();
        // mav.setViewName(viewName);

        return viewName;
    }

}
```

URL 연결 주소를 뷰 이름으로 사용하기

```
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;

@Controller
public class HelloController {

    @RequestMapping("/hello")
    public void hello(@RequestParam("name") String name, Model models) {
        // Model 생성
        HelloModel model = new HelloModel(name);
        models.addAttribute("hello", model);

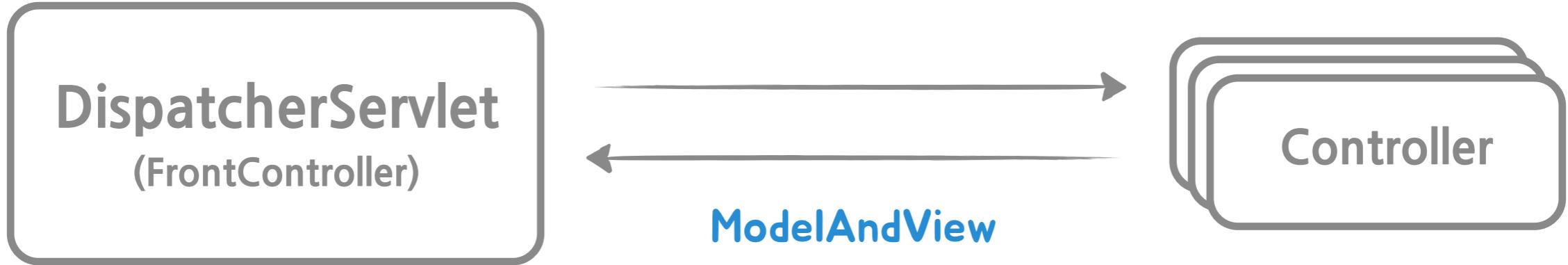
        // ViewName 지정
        // String viewName = "hello";

        // ModelAndView mav = new ModelAndView();
        // mav.setViewName(viewName);

        // return viewName;
    }

}
```

디스패처서블릿은 모델과 뷰(ModelAndView)만 다룬다며?



```

public class DispatcherServlet {

    void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        // 생략

        ModelAndView mv = handlerAdapter.handle(processedRequest, response, handler);

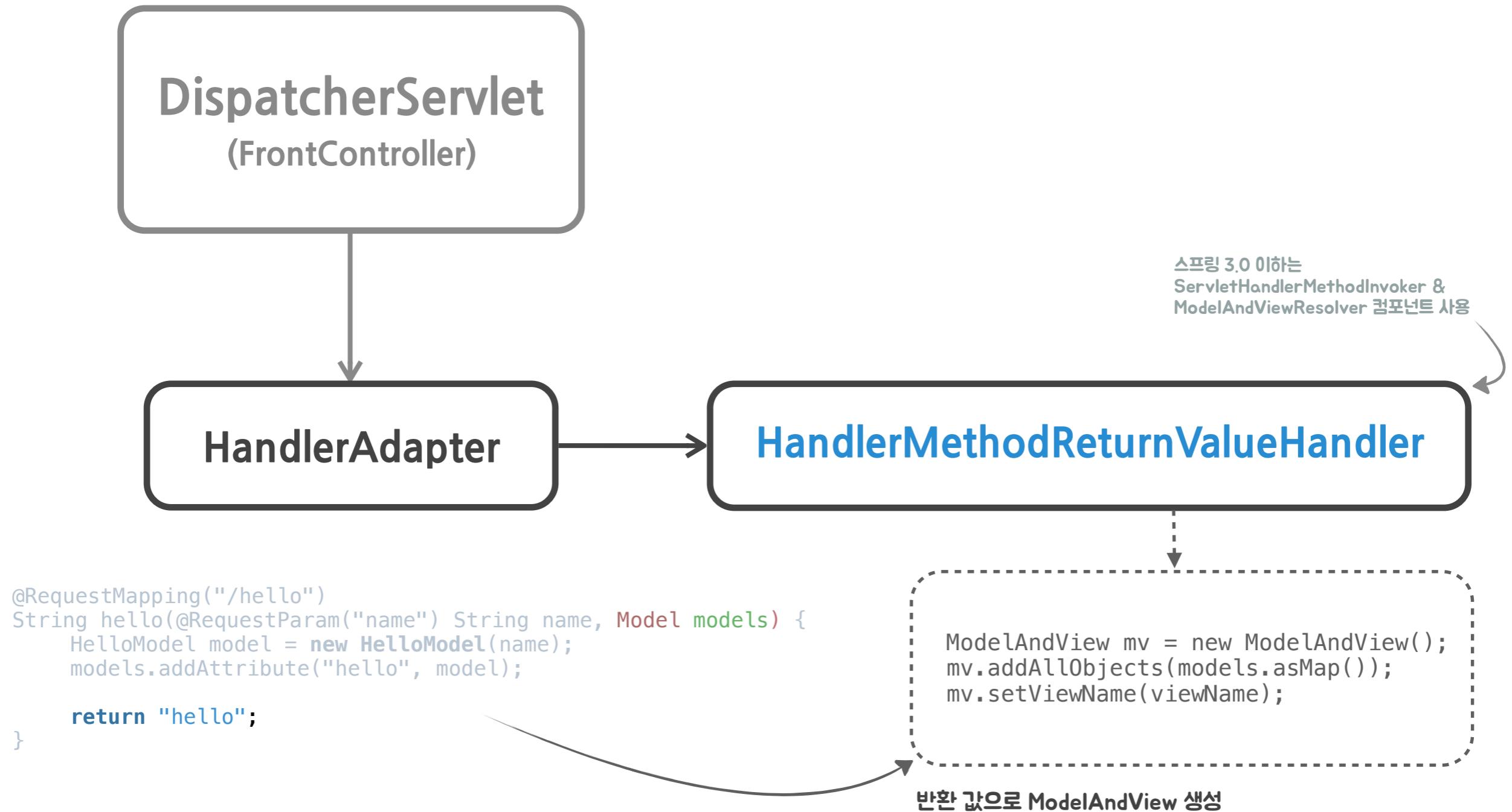
        View view;
        String viewName = mv.getViewName();
        if (viewName != null) {
            view = resolveViewName(viewName, mv.getModelInternal(), locale, request);
        } else {
            view = mv.getView();
        }

        view.render(mv.getModelInternal(), request, response);
    }
}

```

핸들러는 ModelAndView 객체를 반환해야 한다.

핸들러 반환 값을 다루는 컴포넌트



제 7장, HTTP 응답 콘텐트 만들기

7.3, `HttpMessageConverter`

핸들러 반환 값을 즉시 HTTP 응답 본문(Entity body)에 출력

```

import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.http.ResponseEntity;
import org.springframework.http.HttpStatus;

@Controller
@RequestMapping("/responses")
public class GeneratingResponsesController {

    @GetMapping("/annotation")
    @ResponseBody
    public String responseBody() {
        return "The String ResponseBody";
    }

    @GetMapping("/entity/status")
    public ResponseEntity<String> responseEntityStatusCode() {
        return ResponseEntity.status(HttpStatus.FORBIDDEN)
            .body("The String ResponseBody with custom status code (403 Forbidden)");
    }

}

```

핸들러에 @ResponseBody 애노테이션이 선언되었거나
ResponseEntity<?> 타입을 반환하면 뷰를 통하지 않고,
즉시 HTTP 응답 본문에 출력한다.

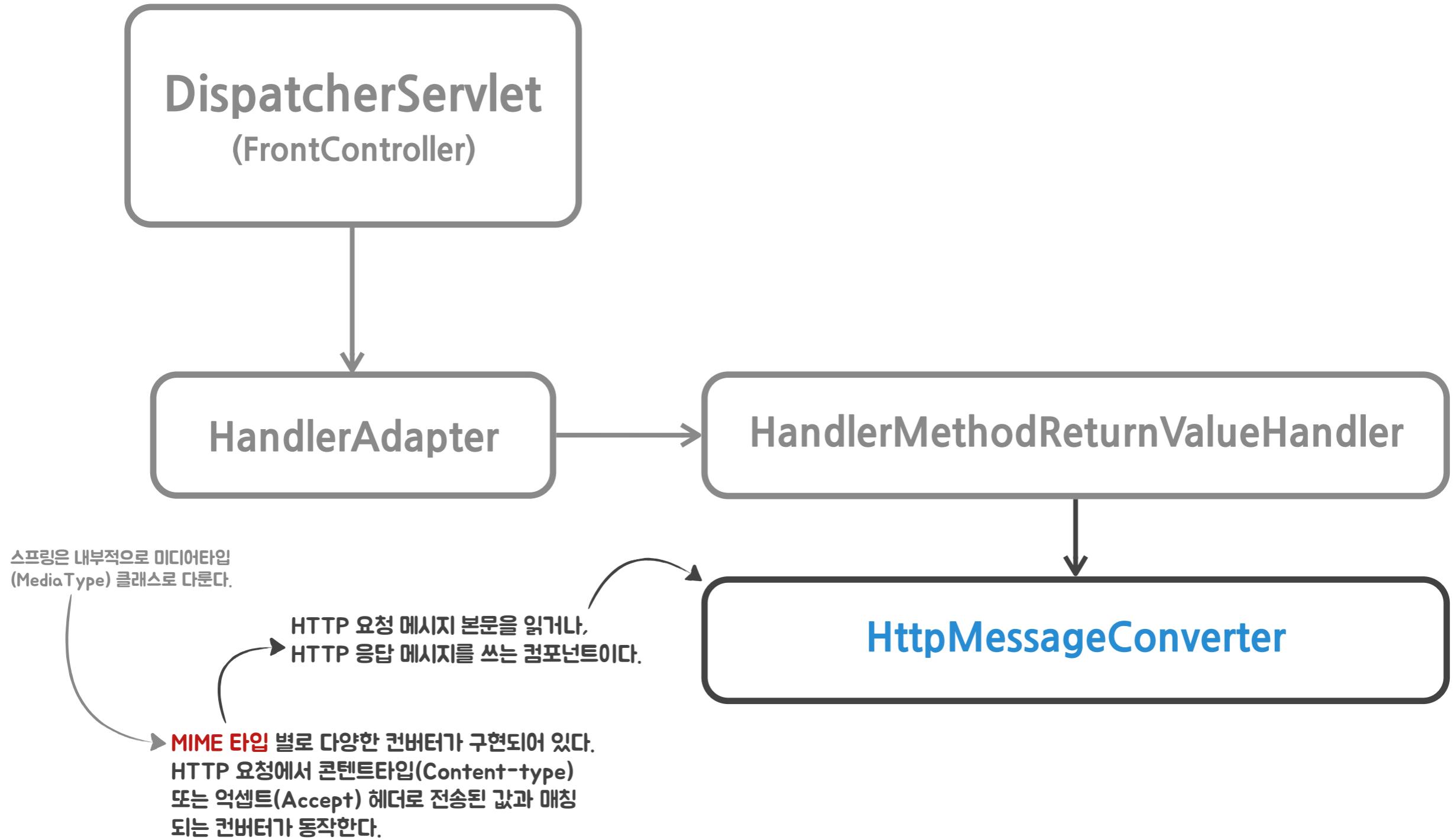
```

PrintWriter writer = response.getWriter();
writer.write(handlerReturnValue);

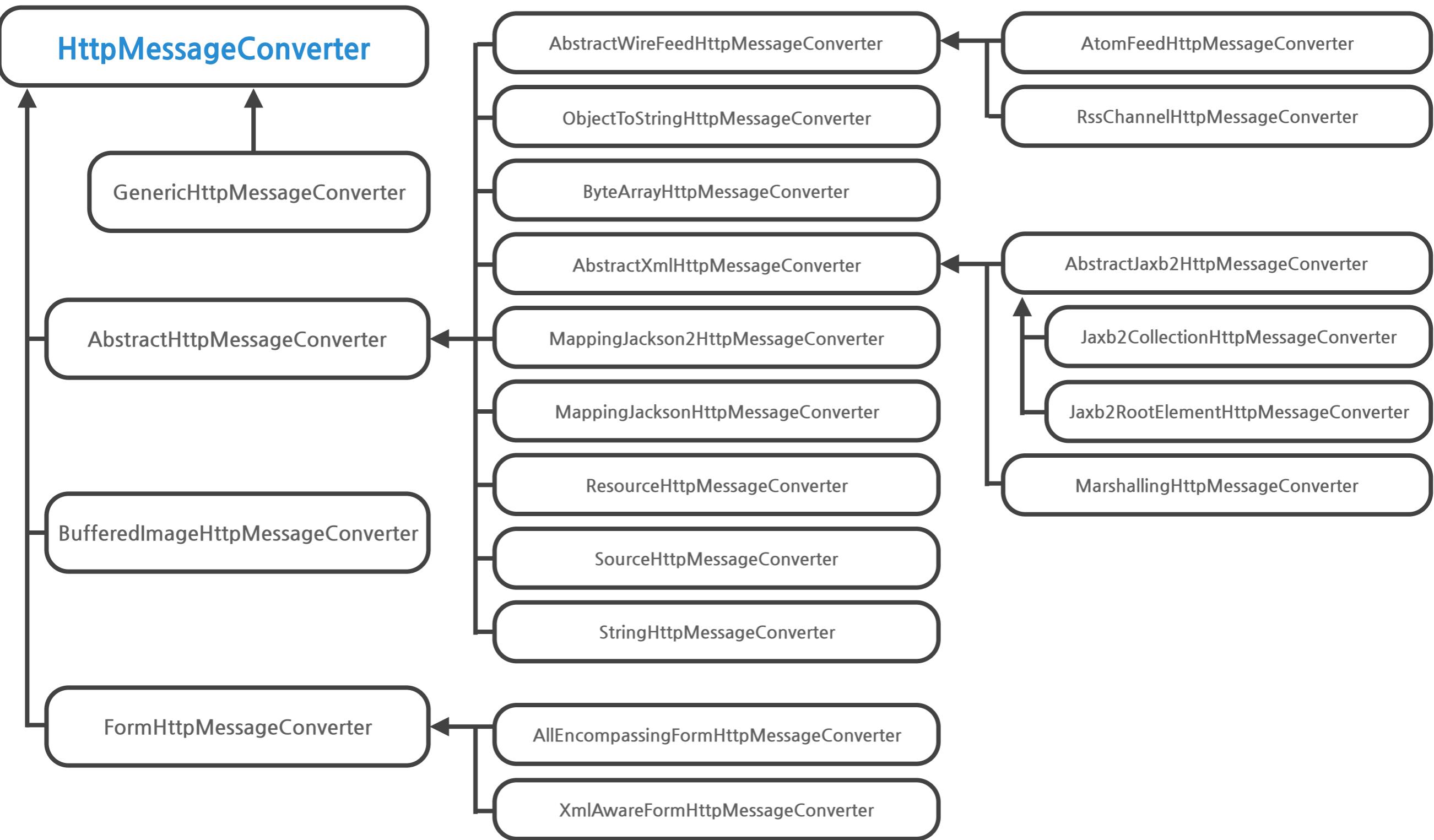
response.setStatus(200);
response.flushBuffer();

```

응답 본문에 출력 처리하는 컴포넌트



HttpMessageConverter Hierarchy



다양한 형식으로 응답 본문에 출력하기

```

import org.springframework.web.bind.annotation.RestController; ←
@RestController
@RequestMapping("/messageconverters")
public class MessageConvertersController {

    // MappingJackson2HttpMessageConverter (useful for serving clients that expect to work with JSON)
    @GetMapping("/json")
    public JavaBean writeJson() {
        return new JavaBean("bar", "apple");
    }

    // Jaxb2RootElementHttpMessageConverter (useful for serving clients that expect to work with XML)
    @GetMapping("/xml")
    public JavaBean writeXml() {
        return new JavaBean("bar", "apple");
    }

    // AtomFeedHttpMessageConverter (useful for serving Atom feeds)
    @GetMapping("/atom")
    public Feed writeFeed() {
        Feed feed = new Feed();
        feed.setFeedType("atom_1.0");
        return feed;
    }
}

```

@RestController은 웹 요청을 처리할 컨트롤러(Controller)이며, 요청 처리 결과를 응답 본문에 출력한다.

@Controller
@ResponseBody
public @interface RestController {
@AliasFor(annotation = Controller.class)
String value() default "";
}

제 7장, HTTP 응답 콘텐트 만들기

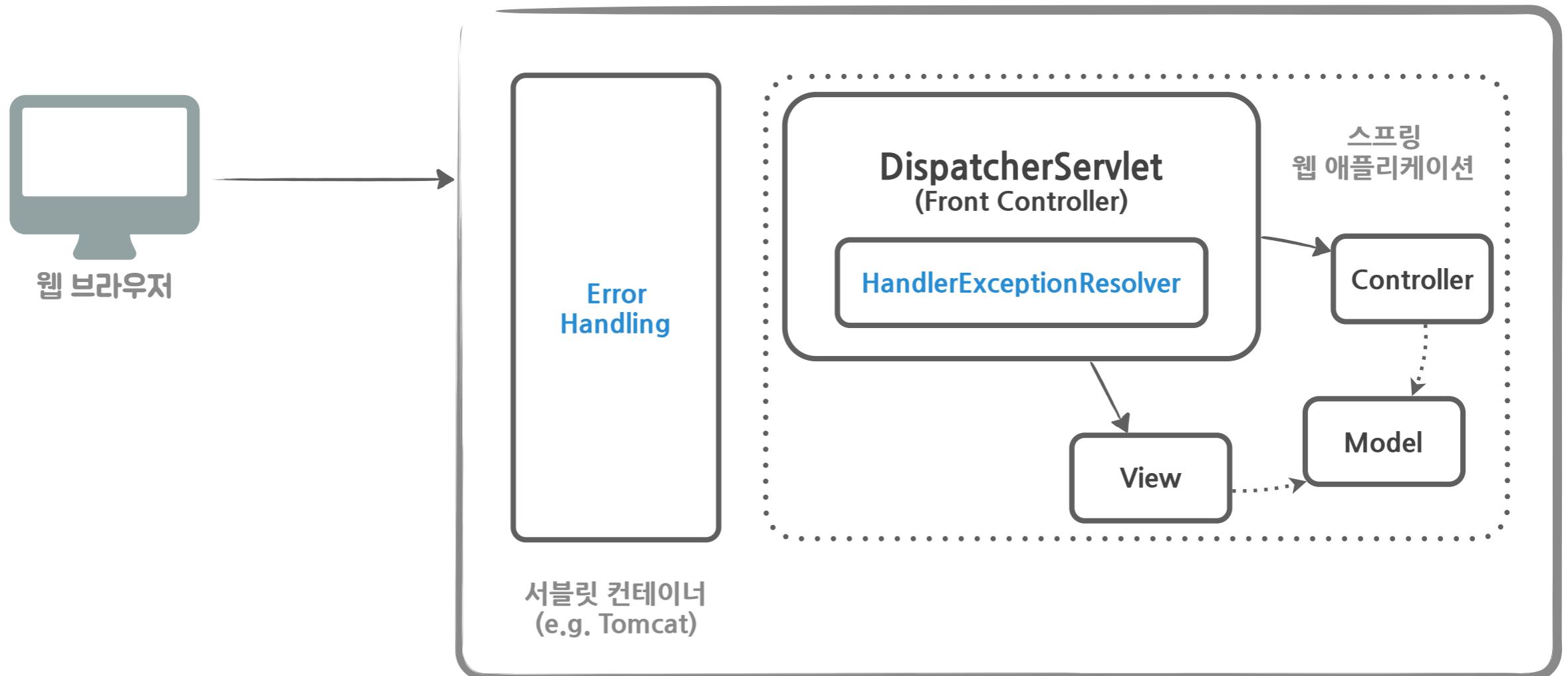
7.4, 콘텐트 협상 (Content Negotiation)

비공개

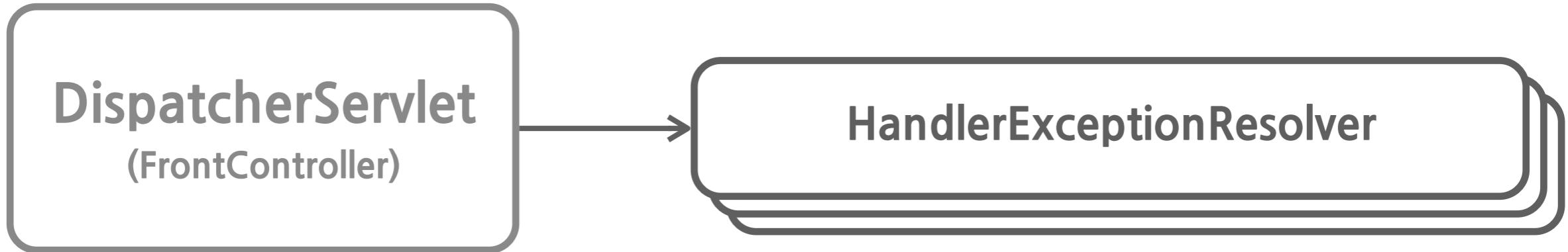
제 8장, 예외 처리하기

두 가지 방법으로 예외 처리 가능

- ✓ 디스패처Servlet 내부에서 HandlerExceptionResolver 컴포넌트로 해결 할 수 있다.
- ✓ 서블릿 컨테이너(servlet container) 오류 처리 전략을 등록해 해결 할 수 있다.



핸들러 수행 중 예외가 발생하면 예외 처리자에 위임



```

class DispatcherServlet {

    void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        // 생략

        ModelAndView mv = null;

        try {
            mv = handlerAdapter.handle(request, response, handler);
        } catch (Exception ex) {
            mv = processHandlerException(request, response, handler, exception);
        }

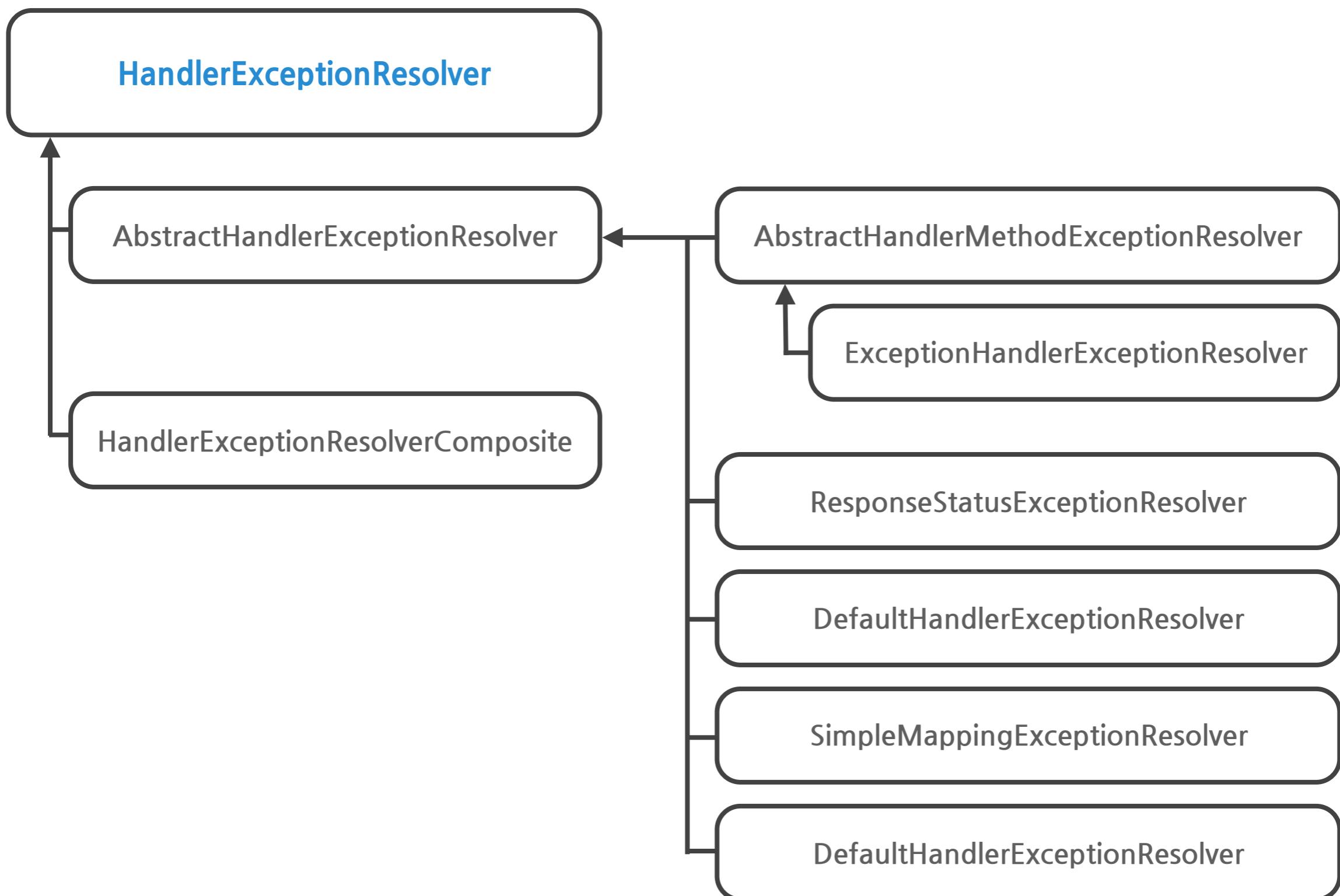
        render(mv, request, response);
    }
}
  
```

핸들러는 수행 중 예외가 발생하면 내부에 등록된 HandlerExceptionResolver에게 예외 처리를 위임한다.

디스패처서블릿에는 여러개의 HandlerExceptionResolver를 등록 할 수 있다.

예외 처리 후 ModelAndView가 반환되면 응답 컨텐트를 출력한다.

HandlerExceptionResolver Hierarchy



예외 처리자(HandlerExceptionResolver) 구성하기

```

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.handler.SimpleMappingExceptionResolver;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ExceptionHandlingConfiguration implements WebMvcConfigurer {

    @Override
    public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        // HandlerExceptionResolver를 직접 구성할 때 사용
    }

    @Override
    public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        // HandlerExceptionResolver 기본 전략 구성 외 추가 전략을 구성할 때 사용
        // 스프링 4.3 이후 사용 가능
        resolvers.add(createSimpleMappingExceptionResolver());
    }

    protected SimpleMappingExceptionResolver createSimpleMappingExceptionResolver() {
        Properties mappings = new Properties();
        mappings.put(MappedException.class.getName(), "mappedExceptionView");

        SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
        exceptionResolver.setExceptionMappings(mappings);

        return exceptionResolver;
    }

    @Bean
    public View mappedExceptionView() {
        return new View() { ... };
    }
}

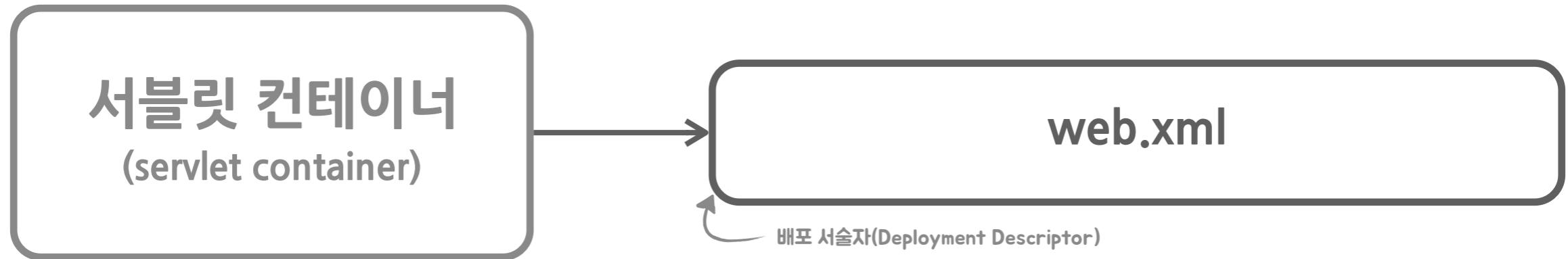
```

@Configuration은 이 클래스가 스프링 애플리케이션 구성 빈(Bean)임을 명시한다.

WebMvcConfigurer 인터페이스를 통해 Spring MVC 설정을 할 수 있다

WebMvcConfigurer를 통해 구성하지 않고, HandlerExceptionResolver를 빈(Bean)으로 등록해도 디스패처서블릿(DispatcherServlet)에 등록된다.

서블릿 컨테이너 오류 처리 전략



```

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                               http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
          version="4.0">

    <error-page>
        <error-code>404</error-code>           HTTP 응답 상태코드에 따라 예외 처리
        <location>/error/not-found</location>
    </error-page>

    <error-page>
        <exception-type>java.lang.Exception</exception-type>   예외 타입에 따라 예외 처리
        <location>/error</location>
    </error-page>

</web-app>
  
```

예외를 처리 할 서블릿 URL (또는 JSP 등)

제 8장, 예외 처리하기

8.1, 애노테이션 기반 예외 처리하기

@ExceptionHandler은 핸들러와 예외를 연결

```

import org.springframework.web.bind.annotation.ExceptionHandler;

@RestController
@RequestMapping("/exceptions")
public class HandlerExceptionResolversController {

    @GetMapping("/exception-handler")
    public void exceptionHandler() {
        throw new IllegalStateException("Sorry!");
    }

    @ExceptionHandler(IllegalStateException.class)
    public String handle(IllegalStateException error) {
        return "IllegalStateException handled!";
    }
}

```

@ExceptionHandler 애노테이션을 사용해 핸들러와 예외를 연결 할 수 있다.

→ `ExceptionHandlerExceptionResolver`가 등록되어 있어야 한다.

Spring MVC 기본 예외 처리 전략 중 하나로 사용된다.

핸들러 메소드 파라미터로 얻을 수 있는 타입

```
import org.springframework.web.bind.annotation.ExceptionHandler;
```

```
@RestController
@RequestMapping("/exceptions")
public class HandlerExceptionResolversController {

    @ExceptionHandler(IllegalStateException.class)
    public String handle(IllegalStateException error) {
        return "IllegalStateException handled!";
    }
}
```

메소드 파라미터로 발생된 예외 개체와 다양한 타입의 얻을 수 있다.

타입	설명
Exception type	발생된 예외 개체
HandlerMethod	예외가 발생한 컨트롤러 내 핸들러 개체
WebRequest, NativeWebRequest, HttpMethod	HTTP 요청 및 메소드, 그리고 응답 개체 (서블릿 스펙에 종속적이지 않은 범용 인터페이스)
Map, Model, ModelMap	뷰에 전달할 모델 오브젝트를 담아둘 수 있는 개체
RedirectAttributes	리다이렉트시 전달할 모델 오브젝트를 담아 전달할 수 있는 개체
java.util.Locale, java.util.TimeZone, java.time.ZoneId	HTTP 요청을 기반으로 LocalResolver가 결정한 언어, 시간, 지역 정보
java.security.Principal	인증된 사용자 개체
HttpServletRequest, HttpServletResponse, HttpSession	서블릿 API 개체 (ServletRequest, ServletResponse 또는 MultipartRequest 타입 지원)
OutputStream, Writer	서블릿 응답(ServletResponse)에 담긴 OutputStream, Writer 개체

핸들러 반환 값으로 사용할 수 있는 타입

```
import org.springframework.web.bind.annotation.ExceptionHandler;
```

```
@RestController
@RequestMapping("/exceptions")
public class HandlerExceptionResolversController {
    @ExceptionHandler(IllegalStateException.class)
    public String handle(IllegalStateException error) {
        return "IllegalStateException handled!";
    }
}
```

반환 값(Return Values)로 뷰 이름(view name) 외 다양한 타입을 반환 할 수 있다.

타입	설명
View	응답 컨텐트를 출력 할 뷰 개체
String	뷰 이름(view name), 뷰리졸버(ViewResolver) 기반 응답 컨텐트 처리
java.util.Map, org.springframework.ui.Model	뷰(View)에 전달할 모델 오브젝트를 담아둘 수 있는 개체
ModelAndView	응답 컨텐트 처리를 위한 모델과 뷰(ModelAndView) 개체
@ResponseBody	반환 값을 HttpMessageConverter로 즉시 출력 처리
HttpEntity, ResponseEntity	반환 값을 HttpMessageConverter로 즉시 출력 처리
void, null	@RestController의 경우 응답 본문이 없음, 그외 기본 뷰 또는 직접 응답 처리

@ResponseStatus는 예외와 HTTP 상태코드를 연결

```
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.http.HttpStatus;
```

```
@RestController
@RequestMapping("/exceptions")
public class HandlerExceptionResolversController {
```

```
    @GetMapping("/response-status")
    public void responseStatus() {
        throw new ConflictException();
    }
```

```
    @ResponseStatus(code = HttpStatus.CONFLICT, reason = "서버가 요청을 수행하는 중에 충돌이 발생했다.")
    public class ConflictException extends RuntimeException {
```

```
}
```

```
}
```

```
    class ResponseStatusExceptionResolver implements HandlerExceptionResolver {
```

```
        ModelAndView resolveException(HttpServletRequest request,
                                     HttpServletResponse response,
                                     Object handler,
                                     Exception ex) {
```

```
            ResponseStatus status = findMergedAnnotation(ex.getClass(), ResponseStatus.class);
            int statusCode = responseStatus.code().value();
            String reason = responseStatus.reason();
```

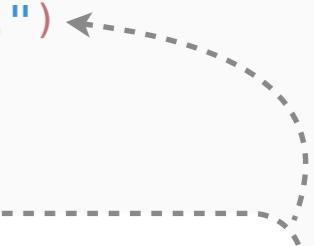
```
            response.sendError(statusCode, resolvedReason);
```

```
            return new ModelAndView();
```

```
}
```

```
}
```

예외에 @ResponseStatus 애노테이션을 선언하고,
HTTP 상태코드와 사유를 연결한다.

제 8장, 예외 처리하기

8.3. 구성 기반 예외 처리하기

예외와 뷰 이름을 연결해 예외 처리하기

```

import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.handler.SimpleMappingExceptionResolver;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ExceptionHandlingConfiguration implements WebMvcConfigurer {

    @Override
    public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        // HandlerExceptionResolver를 직접 구성할 때 사용
    }

    @Override
    public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver> resolvers) {
        // HandlerExceptionResolver 기본 전략 구성외 추가 전략을 구성할 때 사용
        // 스프링 4.3 이후 사용 가능
        resolvers.add(createSimpleMappingExceptionResolver());
    }

    protected SimpleMappingExceptionResolver createSimpleMappingExceptionResolver() {
        Properties mappings = new Properties();
        mappings.put(MappedException.class.getName(), "mappedExceptionView");

        SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
        exceptionResolver.setExceptionMappings(mappings);

        return exceptionResolver;
    }

    @Bean
    public View mappedExceptionView() {
        return new View() { ... };
    }
}

```

@Configuration은 이 클래스가 스프링 애플리케이션 구성 빈(Bean)임을 명시한다.

WebMvcConfigurer 인터페이스를 통해 Spring MVC 설정을 할 수 있다

예외 타입과 뷰 이름을 연결하고 설정한다.

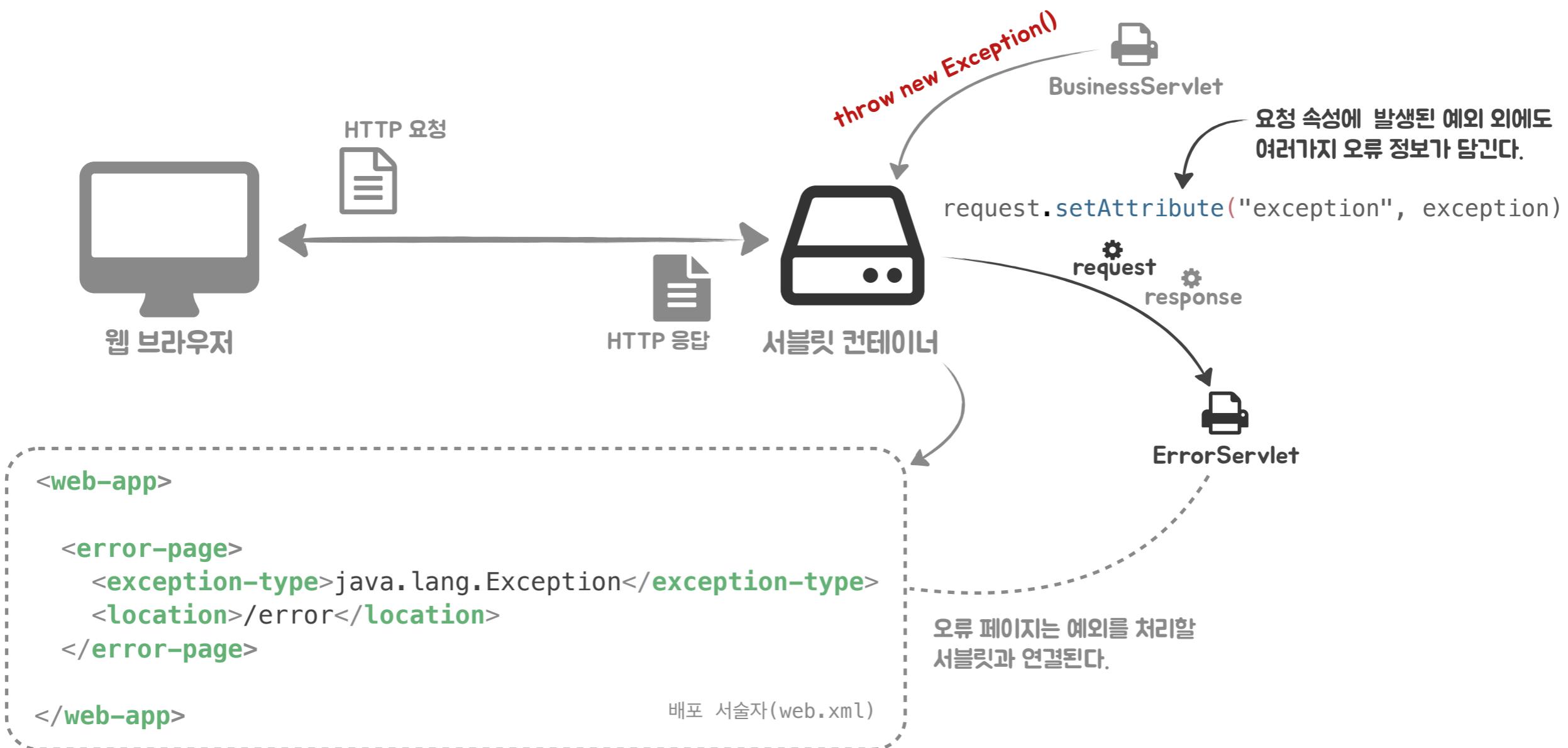
예외 타입외에도 HTTP 상태코드와 뷰 이름을 연결하거나, 모든 예외에 대한 기본 뷰를 구성 할 수도 있다.

제 8장, 예외 처리하기

8.4, 서블릿 컨테이너 오류 처리 전략 활용하기

서블릿 컨테이너 오류 처리 전략

- ✓ 예외가 컨테이너에 전파되면, 요청 속성(Request Attributes)에 오류 정보를 담는다.
- ✓ 등록된 오류 페이지(Error Page) 중 적합한 페이지를 찾아 전달(forward)한다.



요청 속성(Request Attributes)에 담기는 오류 정보

```
(Integer)    servletRequest.getAttribute("javax.servlet.error.status_code");
(Class<?>)  servletRequest.getAttribute("javax.servlet.error.exception_type");
(String)     servletRequest.getAttribute("javax.servlet.error.message");
(Exception)  servletRequest.getAttribute("javax.servlet.error.exception");
(String)     servletRequest.getAttribute("javax.servlet.error.request_uri");
(String)     servletRequest.getAttribute("javax.servlet.error.servlet_name");
```

속성 이름	속성 유형	설명
javax.servlet.error.status_code	java.lang.Integer	HTTP 상태코드
javax.servlet.error.exception_type	java.lang.Class	발생된 예외 클래스 타입
javax.servlet.error.message	java.lang.String	발생된 예외의 메시지
javax.servlet.error.exception	java.lang.Throwable	발생된 예외 객체
javax.servlet.error.request_uri	java.lang.String	예외가 발생한 요청 URI
javax.servlet.error.servlet_name	java.lang.String	예외가 발생한 서블릿 이름

스프링부트로 서블릿 컨테이너 오류 페이지 구성하기

```

import org.springframework.boot.web.server.ErrorPageRegistrar;
import org.springframework.boot.web.server.ErrorPageRegistry;
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ExceptionHandlingConfiguration implements WebMvcConfigurer, ErrorPageRegistrar {

    // 컨테이너 오류 페이지 구성
    //

    // 컨테이너 오류 페이지는 배포 서술자(web.xml)를 통해 구성 할 수 있습니다.
    // 아쉽게도 서블릿 API는 자바 코드로 오류 페이지를 구성하는 방법을 제공하지 않습니다.
    // 하지만 스프링부트 기반 임베디드 컨테이너를 사용하고 있다면, ErrorPageRegistrar 컴포넌트를 통해 구성 할 수 있습니다.

}

```

@Configuration은 이 클래스가 스프링 애플리케이션 구성 빈(Bean)임을 명시한다.

WebMvcConfigurer 인터페이스를 통해 Spring MVC 설정을 할 수 있다

```

@Override
public void registerErrorPages(ErrorPageRegistry registry) {
    registry.addErrorPages(
        new ErrorPage(java.lang.Exception.class, "/error")
    );
}

```

```

<web-app>
    <error-page>
        <exception-type>java.lang.Exception</exception-type>
        <location>/error</location>
    </error-page>
</web-app>

```

오류 처리 핸들러 작성하기

```
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.context.request.RequestAttributes;

@RestController
@RequestMapping("/error")
public class ContainerErrorHandlerController {

    @GetMapping("/throw")
    public void throwError() {
        throw new ContainerErrorHandlerException();
    }

    @GetMapping("/handle")
    public String handleContainerErrorHandlerException(WebRequest webRequest) {
        Exception exception = (Exception) webRequest.getAttribute(
            "javax.servlet.error.exception", RequestAttributes.SCOPE_REQUEST);

        return String.format("%s handled!", exception);
    }
}
```

제 8장, 예외 처리하기

8.5. 스프링 부트 예외 처리 전략 활용하기

■ 비공개

제 9장, 요청과 응답 가로채기

 비공개

제 10장, 국제화(i18n)와 메세지(Message) 다루기

 비공개

제 11장, 스프링 웹 테스트

 준비 중

부록, 스프링 IoC 컨테이너 구성과 사용

 준비 중

부록, 디스패처서블릿(DispatcherServlet)을 들여다보다

 준비 중

부록, 웹 애플리케이션 아키텍처 (Web Application Architecture)

 비공개

백견이불여일타(百見而不如一打) 백번 보는것보다 한번 쳐보는게 낫다

<https://springrunner.io>



이 문서의 내용은 크리에이티브 커먼즈 저작자표시-비영리 4.0 국제 라이선스에 따라 이용하실 수 있습니다.
<https://creativecommons.org/licenses/by/4.0/deed.ko>