

3. MVC 설정

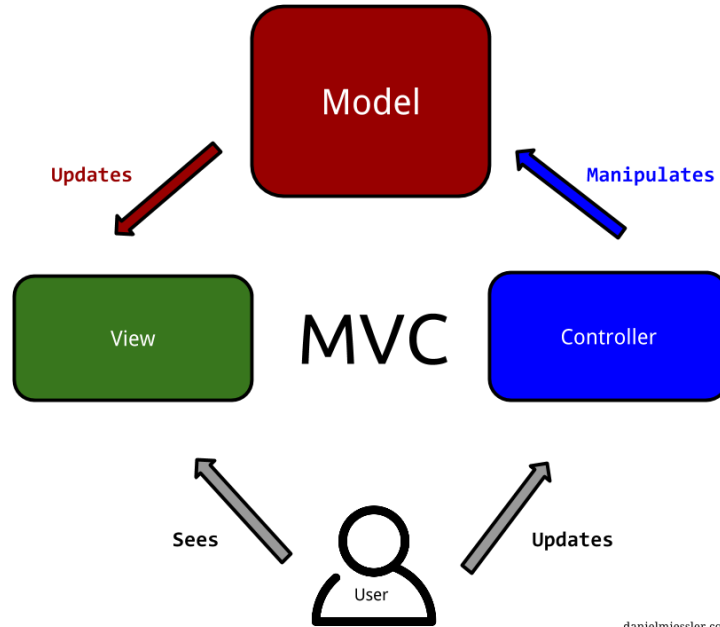


Objectives

- MVC구조의 이해
- 스프링 MVC의 다양한 예제의 학습
- 스프링 MVC를 이용하는 파일 업로드 연습
- 스프링 MVC의 예외처리

MVC(Model-View-Controller)

- 대부분의 서블릿 기반 프레임워크들이 사용하는 방식
- 데이터와 처리, 화면을 분리하는 방식
- 웹에서는 Model 2 방식으로 표현



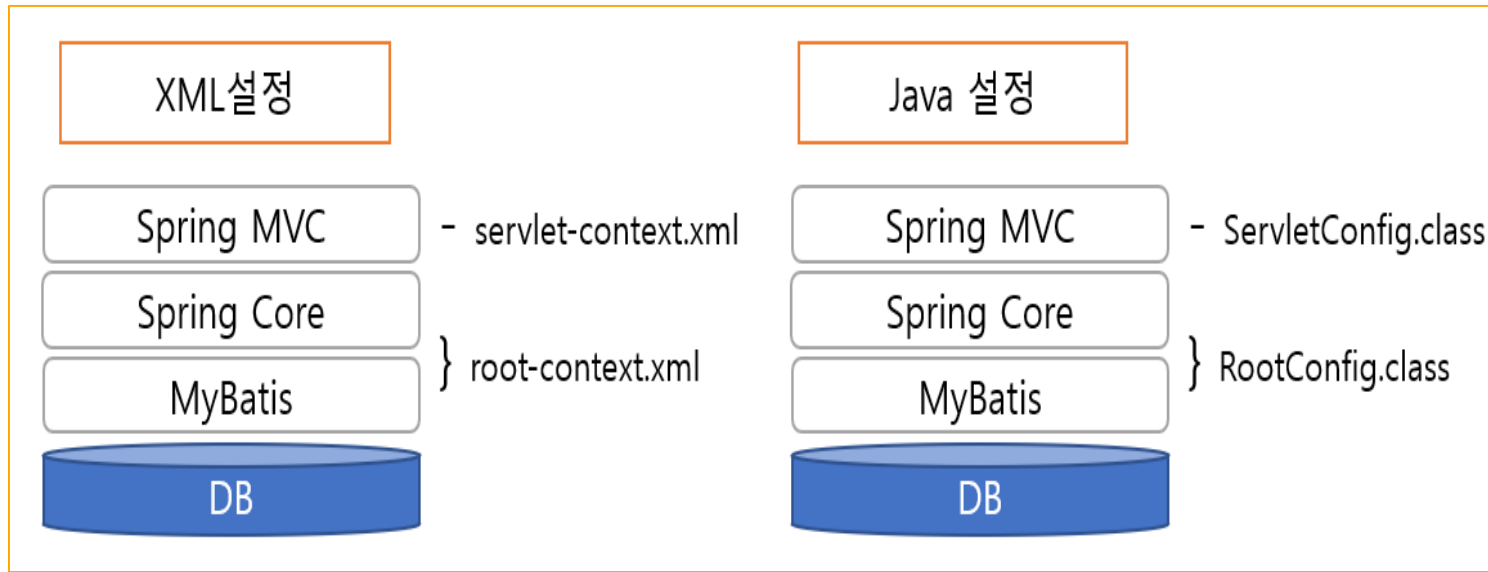
스프링과 스프링 MVC

- 스프링 프레임워크 Core + 여러 Sub 프로젝트들
- <https://spring.io/projects>

- 별도로 결합해서 사용하기 때문에 설정 역시 별도로 처리 가능



일반적인 스프링 + 스프링 MVC

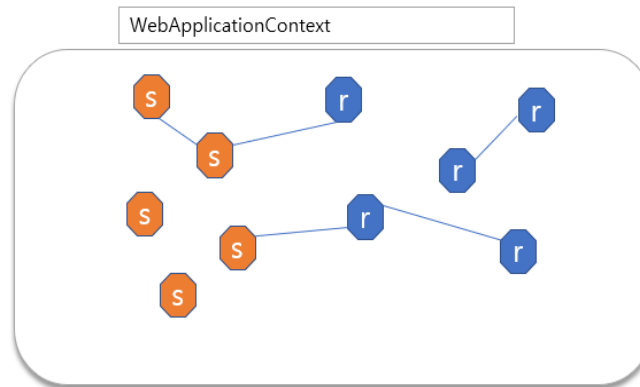
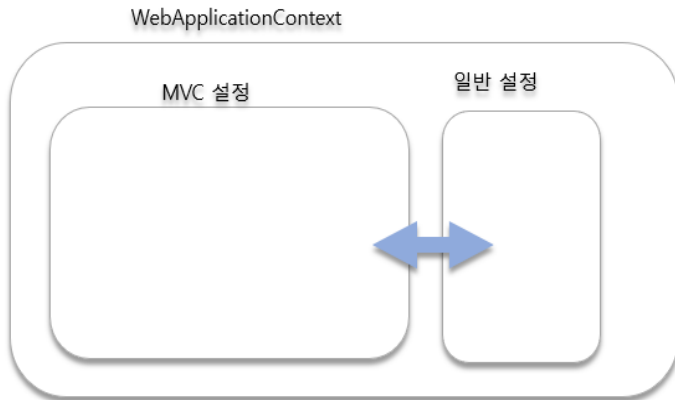



* XML이나 Java설정 이용시에 설정 분리


웹 프로젝트의 구조

■ 스프링을 실행하는 존재

- ApplicationContext => WebApplicationContext
- 같이 연동되는 방식으로 동작하기 때문에 설정을 분리해도 통합해서 사용가능



 root-context.xml에서 정의된 Bean들

 ~~root-context.xml에서 정의된 Bean들~~
servlet-context에서 정의된 bean들

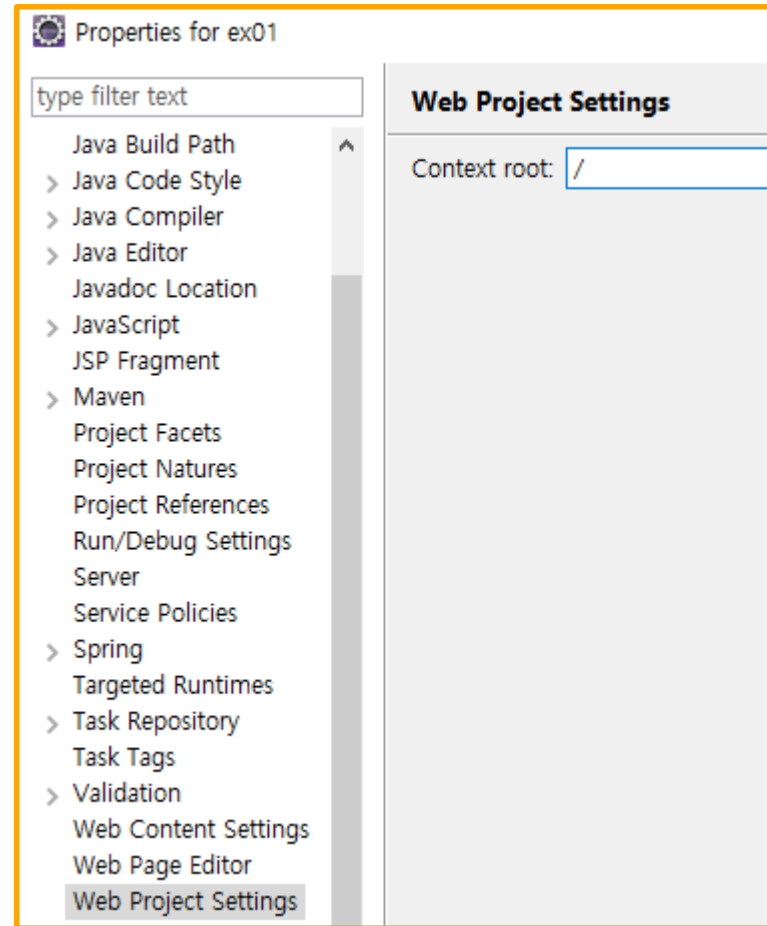
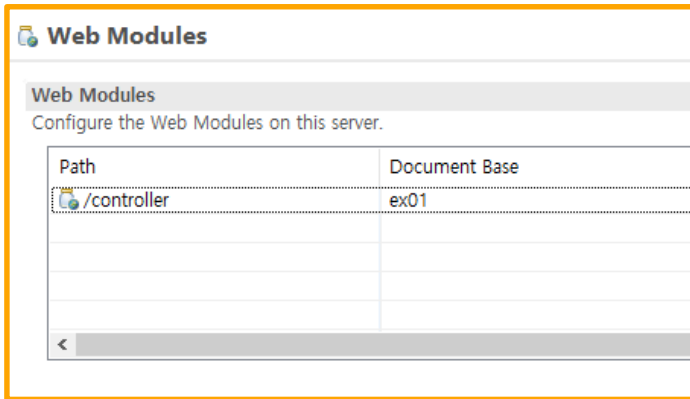
프로젝트 설정후 변경

- Servlet은 3.00이상으로 변경

```
<dependency>  
  <groupId>javax.servlet</groupId>  
  <artifactId>javax.servlet-api</artifactId>  
  <version>3.1.0</version>  
</dependency>
```

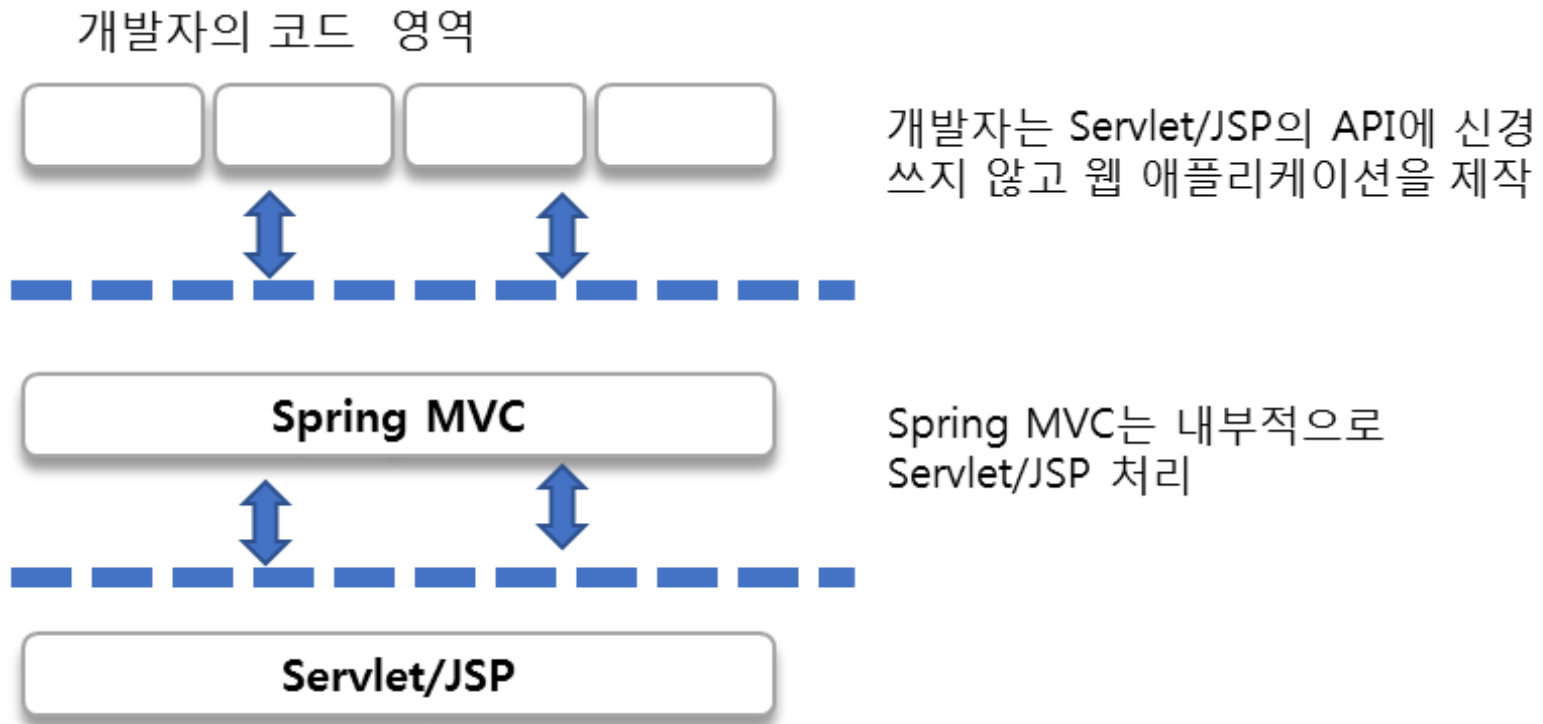
프로젝트의 경로(path)

- Tomcat을 이용하는 경로 변경
- Web Project Setting를 이용하는 변경

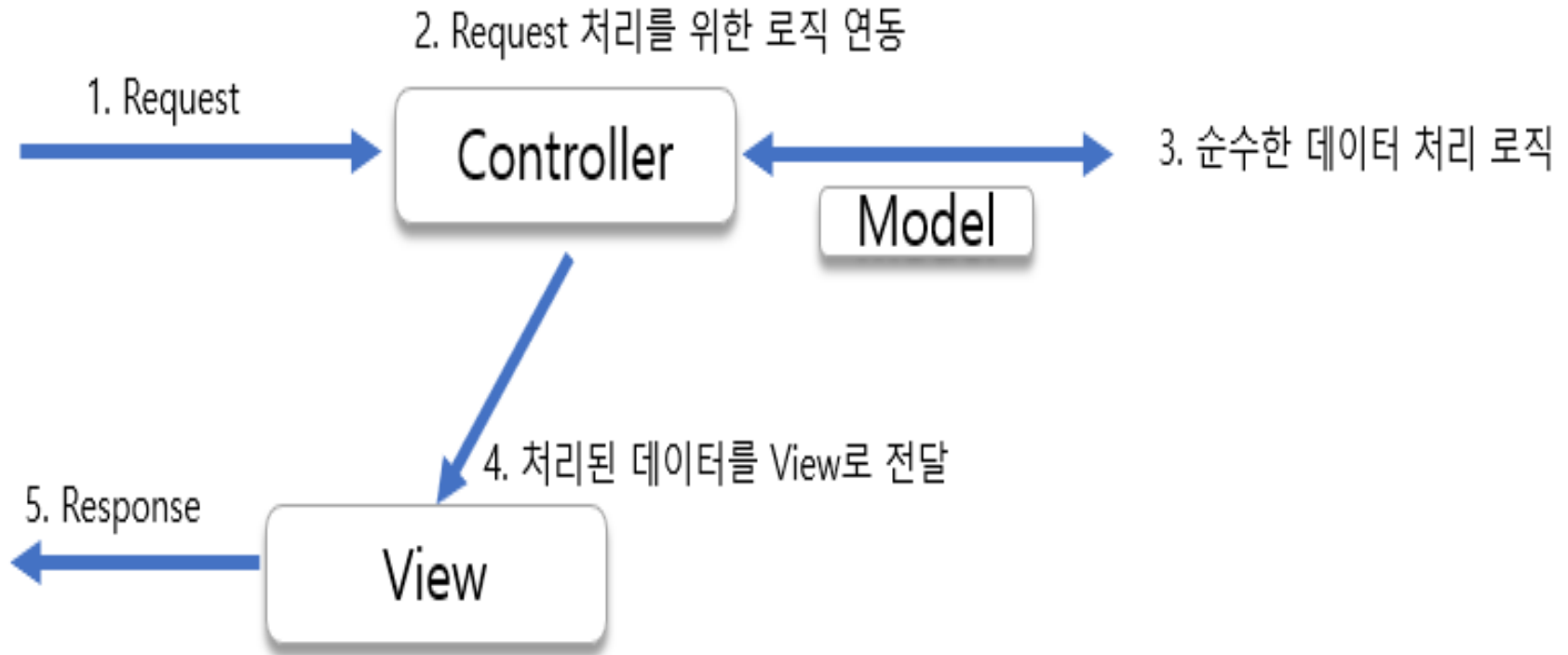


스프링 MVC의 기본 사상

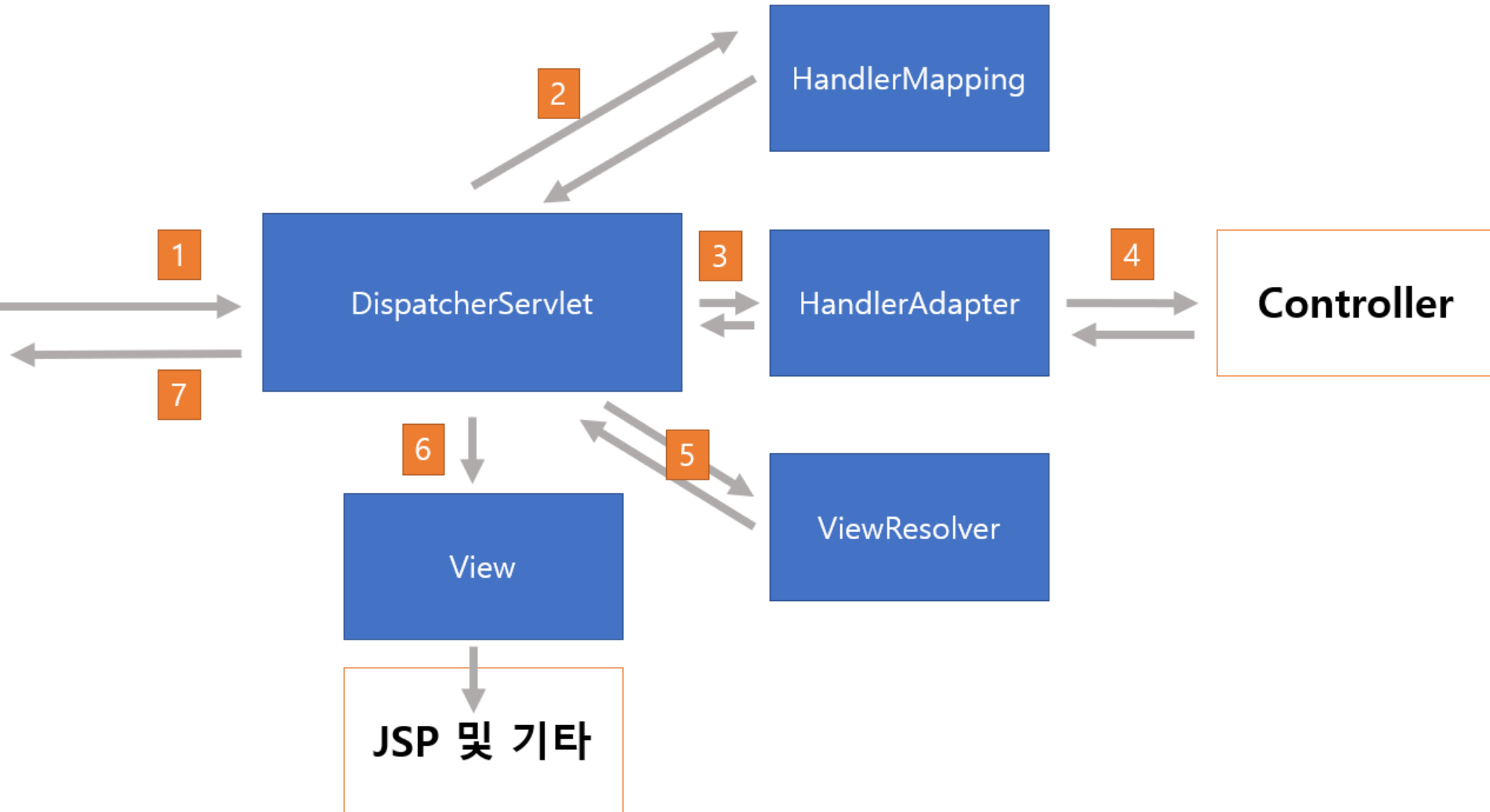
- 서블릿 기반이긴 하지만 한 단계 더 추상화된 수준의 개발 지향
- 서블릿 API없이도 개발이 가능한 수준



모델2 방식과 스프링 MVC



스프링 MVC의 기본 흐름



Controller

- HttpServletRequest, HttpServletResponse를 거의 사용할 필요 없이 필요한 기능 구현
- 다양한 타입의 파라미터 처리, 다양한 타입의 리턴 타입 사용 가능
- GET 방식, POST 방식 등 전송 방식에 대한 처리를 어노테이션으로 처리 가능
- 상속/인터페이스 방식 대신에 어노테이션만으로도 필요한 설정 가능

@Controller, @RequestMapping

- @Controller – 해당 클래스의 인스턴스를 스프링의 빈으로 등록하고 컨트롤러로 사용
- <component-scan>과 같이 활용
- @RequestMapping – 특정한 URI에 대한 처리를 해당 컨트롤러나 메서드에서 처리

SampleController 클래스

```
package org.zerock.controller;
```

```
import org.springframework.stereotype.Controller;
```

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

```
@Controller
```

```
@RequestMapping("/sample/*")
```

```
public class SampleController {
```

```
}
```

@RequestMapping의 변화

- 스프링 4.3 전까지는 @RequestMapping(method = 'get') 방식으로 사용
- 스프링 4.3이후에는 @GetMapping, @PostMapping등으로 간단히 표현 가능

SampleController 클래스의 일부

```
@RequestMapping(value = "/basic", method = {RequestMethod.GET, RequestMethod.POST})
```

```
public void basicGet() {  
  
    log.info("basic get.....");  
  
}
```

```
@GetMapping("/basicOnlyGet")  
public void basicGet2() {  
  
    log.info("basic get only get.....");  
  
}
```

컨트롤러의 파라미터 수집

- 스프링 MVC의 컨트롤러는 메서드의 파라미터를 자동으로 수집, 변환하는 편리한 기능을 제공
- Java Beans 규칙에 맞게 작성되어야 한다.
 - 생성자가 없거나 빈 생성자
 - 올바른 규칙으로 만들어진 Getter/Setter

```
@GetMapping("/ex01")
public String ex01(SampleDTO dto) {

    log.info("" + dto);

    return "ex01";
}
```

```
@Data
public class SampleDTO {

    private String name;
    private int age;
}
```

리스트/배열

```
@GetMapping("/ex02List")
public String ex02List(@RequestParam("ids")ArrayList<String> ids) {

    log.info("ids: " + ids);

    return "ex02List";
}
```


@InitBinder

```
public class TodoDTO {  
  
    private String title;  
    private Date dueDate; //날짜 타입의 변환 필요  
}
```

```
@InitBinder  
public void initBinder(WebDataBinder binder) {  
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));  
}
```

...생략...

```
@GetMapping("/ex03")  
public String ex03(TodoDTO todo) {  
    log.info("todo: " + todo);  
    return "ex02";  
}
```

@DateTimeFormat

- @InitBinder 외에도 날짜에 대한 처리가 쉽게 추가된 어노테이션

```
package org.zerock.domain;

import java.util.Date;

import org.springframework.format.annotation.DateTimeFormat;

import lombok.Data;

@Data
public class TodoDTO {

    private String title;

    @DateTimeFormat(pattern = "yyyy/MM/dd")
    private Date dueDate;
}
```

Model이라는 데이터전달자

- Model 객체는 JSP에 컨트롤러에서 생성된 데이터를 담아서 전달하는 역할을 하는 존재
- 모델 2 방식에서 사용하는 request.setAttribute()와 유사한 역할

```
public String home(Model model) {  
  
    model.addAttribute("serverTime", new java.util.Date());  
  
    return "home";  
}
```

@ModelAttribute

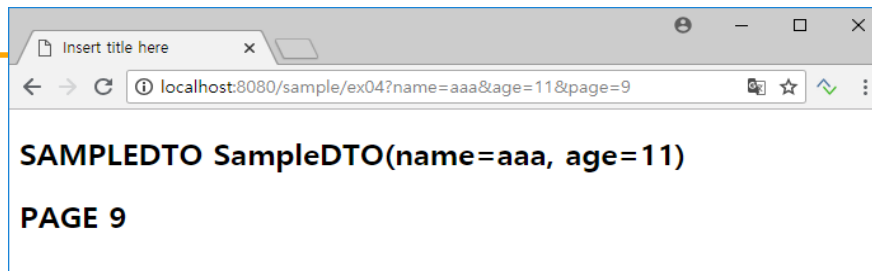
- 컨트롤러에서 메서드의 파라미터는 기본자료형을 제외한 객체형 타입은 다시 화면으로 전달
- @ModelAttribute는 명시적으로 화면에 전달되도록 지정

```
@GetMapping("/ex04")
public String ex04(SampleDTO dto, @ModelAttribute("page") int page) {

    log.info("dto: " + dto);
    log.info("page: " + page);

    return "/sample/ex04";
}
```

```
<h2>SAMPLEDTO ${sampleDTO }</h2>
<h2>PAGE ${page }</h2>
```



RedirectAttribute

- 화면에 한번만 전달되는 파라미터를 처리하는 용도
- 내부적으로 HttpSession객체에 담아서 한번만 사용되고, 폐기

```
rttr.addFlashAttribute("name", "AAA");  
rttr.addFlashAttribute("age", 10);
```

```
return "redirect:/";
```

```
response.sendRedirect("/home?name=aaa  
&age=10");
```

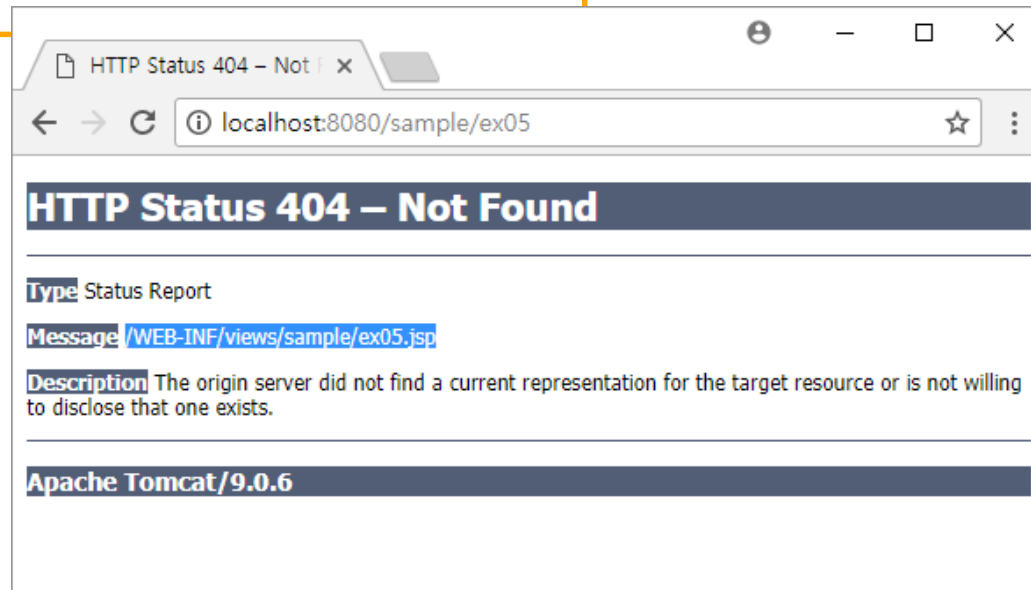
Controller의 리턴타입

- String: jsp를 이용하는 경우에는 jsp 파일의 경로와 파일이름을 나타내기 위해서 사용
- void: 호출하는 URL과 동일한 이름의 jsp를 의미
- VO, DTO 타입: 주로 JSON 타입의 데이터를 만들어서 반환하는 용도로 사용 (추가적인 라이브러리 필요).
- ResponseEntity 타입: response할 때 Http 헤더 정보와 내용을 가공하는 용도로 사용 (추가적인 라이브러리 필요).
- Model, ModelAndView: Model로 데이터를 반환하거나 화면까지 같이 지정하는 경우에 사용 (최근에는 많이 사용하지 않습니다.).
- HttpHeaders: 응답에 내용 없이 Http 헤더 메시지만 전달하는 용도로 사용

void타입

- 호출하는 URL과 동일한 이름의 jsp파일

```
@GetMapping("/ex05")  
public void ex05() {  
    log.info("/ex05.....");  
}
```



String 타입

- 상황에 따라 다른 화면을 보여줄 필요가 있을 경우에 유용하게 사용
- String 타입에는 다음과 같은 특별한 키워드를 붙여서 사용할 수 있음
 - redirect: 리다이렉트 방식으로 처리하는 경우
 - forward: 포워드 방식으로 처리하는 경우

객체 타입

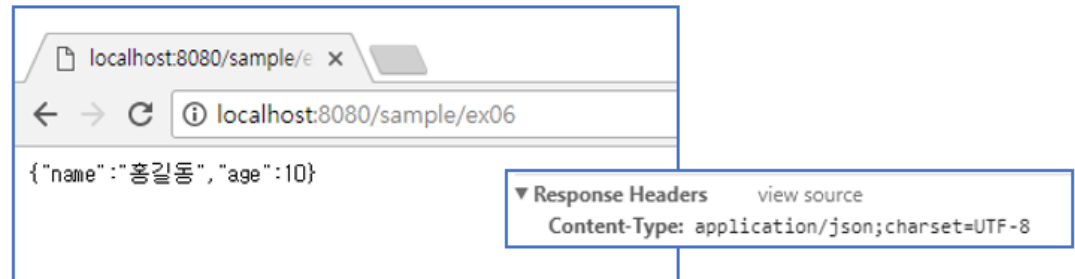
- XML이나 JSON으로 처리
- @ResponseBody 어노테이션과 같이 사용

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.4</version>
</dependency>
```

```
@GetMapping("/ex06")
public @ResponseBody SampleDTO ex06() {
    log.info("/ex06.....");

    SampleDTO dto = new SampleDTO();
    dto.setAge(10);
    dto.setName("홍길동");

    return dto;
}
```



ResponseEntity

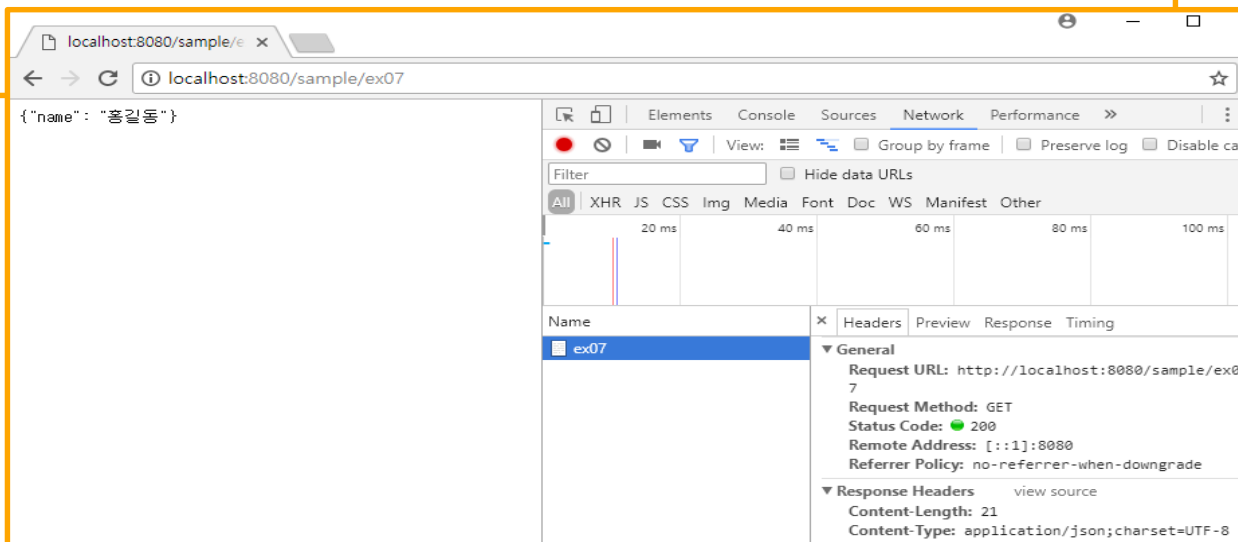
- HTTP헤더 정보와 추가적인 데이터를 전달할 때 사용

```
@GetMapping("/ex07")
public ResponseEntity<String> ex07() {
    log.info("/ex07.....");

    // {"name": "홍길동"}
    String msg = "{\W"name\W": \W"홍길동\W}";

    HttpHeaders header = new HttpHeaders();
    header.add("Content-Type", "application/json;charset=UTF-8");

    return new ResponseEntity<>(msg, header, HttpStatus.OK);
}
```



파일업로드 처리

- Servlet 3.0이후(Tomcat 7.0)에는 기본적으로 업로드 되는 파일을 처리할 수 있는 기능이 추가
- 별도로 commons-fileupload 라이브러리 등을 사용

```
<dependency>  
  <groupId>commons-fileupload</groupId>  
  <artifactId>commons-fileupload</artifactId>  
  <version>1.3.3</version>  
</dependency>
```

파일 업로드를 위한 servlet-context.xml

- multipartResolver라는 이름으로 스프링 빈 설정

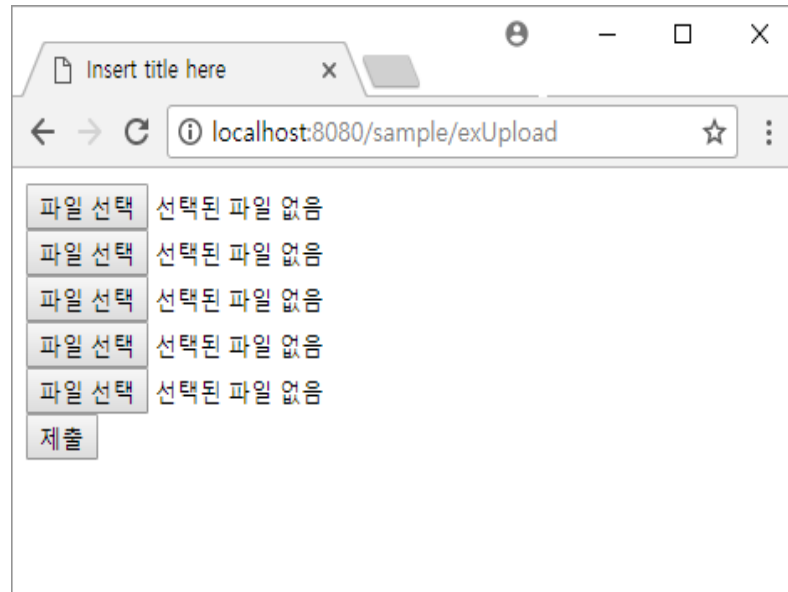
```
<beans:bean id="multipartResolver"
class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <beans:property name="defaultEncoding" value="utf-8"> </beans:property>
  <!-- 1024 * 1024 * 10 bytes 10MB -->
  <beans:property name="maxUploadSize" value="104857560"> </beans:property>
  <!-- 1024 * 1024 * 2 bytes 2MB -->
  <beans:property name="maxUploadSizePerFile" value="2097152"> </beans:property>
  <beans:property name="uploadTempDir" value
= "file:/C:/upload/tmp"> </beans:property>
  <beans:property name="maxInMemorySize" value="10485756"> </beans:property>
</beans:bean>
```

파일업로드를 위한 HTML

■ <form>태그내 enctype='multipart

```
<form action="/sample/exUploadPost" method="post" enctype="multiPart/form-data">
```

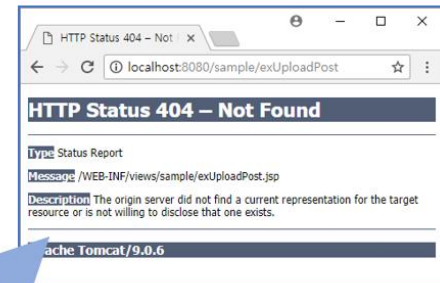
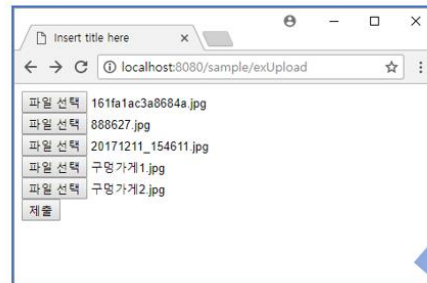
```
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='file' name='files'>
</div>
<div>
  <input type='submit'>
</div>
</form>
```



업로드되는 파일의 처리

■ MultipartFile을 이용해서 처리

```
@PostMapping("/exUploadPost")
public void exUploadPost(ArrayList<MultipartFile>
files) {
    files.forEach(file -> {
        log.info("-----");
        log.info("name:" + file.getOriginalFilename());
        log.info("size:" + file.getSize());
    });
}
```



```
INFO : org.zerock.controller.SampleController - name:161fa1ac3a8684a.jpg
INFO : org.zerock.controller.SampleController - size:1018843
INFO : org.zerock.controller.SampleController - name:888627.jpg
INFO : org.zerock.controller.SampleController - size:232402
INFO : org.zerock.controller.SampleController - name:20171211_154611.jpg
INFO : org.zerock.controller.SampleController - size:413005
INFO : org.zerock.controller.SampleController - name:구멍가게1.jpg
INFO : org.zerock.controller.SampleController - size:121493
INFO : org.zerock.controller.SampleController - name:구멍가게2.jpg
INFO : org.zerock.controller.SampleController - size:304061
```

컨트롤러의 예외(Exception)처리

- @ExceptionHandler와 @ControllerAdvice를 이용한 처리
- @ResponseBody를 이용하는 예외 메시지 구성

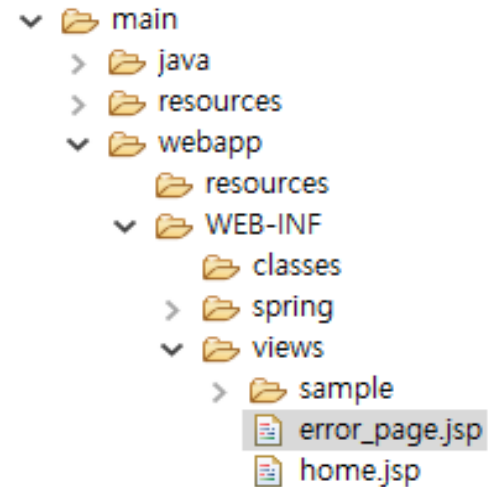
@ControllerAdvice

- 예외처리와 원래의 컨트롤러가 혼합된 형태의 클래스가 작성되는 방식
- @ExceptionHandler는 해당 메서드가 ()들어가는 예외 타입을 처리한다는 것을 의미

```
@ControllerAdvice
@Log4j
public class CommonExceptionHandler {

    @ExceptionHandler(Exception.class)
    public String except(Exception ex, Model model) {

        log.error("Exception ....." + ex.getMessage());
        model.addAttribute("exception", ex);
        log.error(model);
        return "error_page";
    }
}
```



error_page.jsp의 내용

```
<h4> <c:out value="${exception.getMessage()}"> </c:out> </h4>
<ul>
  <c:forEach items="${exception.getStackTrace()}" var="stack">
    <li> <c:out value="${stack}"> </c:out> </li>
  </c:forEach>
</ul>
```

