

Part 2

스프링 MVC 설정

M : Model : 예제가 처리한 정보, 데이터 처리 과정
V : View : 사용자 인터페이스 (화면)
C : Controller : Model과 View 간 상호 동작 조정

PART 2에서는 스프링 프레임워크가 가장 많이 활용되는 Web 관련 개발 환경과 라이브러리 설정에 대한 내용으로 진행합니다. 스프링 MVC라고 불리는 웹 관련 스프링 라이브러리는 최근 웹 개발에서 필수적으로 사용되는 구조로 기존에 Servlet/JSP를 이용하는 개발에 비해서 간단하고, 빠른 개발이 가능하기 때문에 수많은 웹 관련 프레임워크들 중에서 독보적인 인기를 얻고 있습니다.

국내에서는 전자정부 표준 프레임워크를 스프링 프레임워크와 스프링 MVC를 이용해서 공공 프로젝트에서 표준으로 사용하고 있습니다. 일반적인 경우라면 스프링 프레임워크를 이용한다는 의미는 스프링 MVC 프로젝트인 경우가 대부분입니다.

PART 2에서는 다음과 같은 내용들을 학습합니다.

- 스프링 MVC 프로젝트의 생성과 동작 과정
- 스프링 MVC 구조의 이해와 다양한 예제 작성
- 스프링 MVC를 이용하는 파일 업로드 처리
- 스프링 MVC의 예외 처리

05

스프링 MVC의 기본 구조

Part 01

Part 02

Part 03

Part 04

Part 05

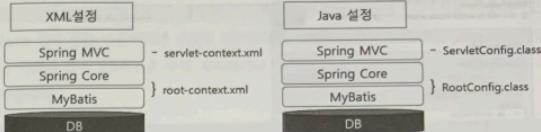
Part 06

Part 07



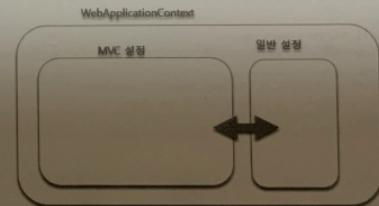
스프링은 하나님의 기능을 위해서만 만들어진 프레임워크가 아니라 '고어'라고 할 수 있는 프레임워크에 여러 서브 프로젝트를 결합해서 다양한 상황에 대처할 수 있도록 개발되어 있습니다. 서브 프로젝트라는 의미를 개발자의 입장에서 가장 쉽게 이해할 수 있는 방법은 별도의 설정이 존재할 수 있다'라는 개념입니다. Spring Legacy Project로 생성한 예제의 경우에도 servlet-context.xml과 root-context.xml로 설정 파일이 분리된 것을 볼 수 있습니다. 스프링 MVC가 서브 프로젝트이므로 구성 방식이나 설정 역시 조금 다르다고 볼 수 있습니다.

이 책의 예제에서 만드는 구조는 다음 그림과 같은 구조가 됩니다.



5.1 스프링 MVC 프로젝트의 내부 구조

스프링 MVC 프로젝트를 구성해서 사용한다는 의미는 내부적으로는 root-context.xml로 사용하는 일반 Java 영역(흔히 POJO(Plain Old Java Object))과 servlet-context.xml로 설정하는 Web 관련 영역을 같이 연동해서 구동하게 됩니다. 그림으로 간단하게 표현하면 다음과 같은 구조라고 볼 수 있습니다.



바깥쪽에 있는 WebApplicationContext라는 존재는 기존의 구조에 MVC 설정을 포함하는 구조로 만들어 집니다. 스프링은 원래 목적 자체가 웹 애플리케이션을 목적으로 나온 프레임워크가 아니기 때문에 달라지는 영역에 대해서는 완전히 분리하고 연동하는 방식으로 구현되어 있습니다.

Eclipse(STS) 내 'Spring Legacy Project'를 이용해서 'ex01' 프로젝트를 생성합니다. 프로젝트는 'Spring MVC Project'로 생성합니다. 패키지명은 기존과 동일하게 'org.zerock.controller'로 지정합니다.

Part 01

```

INFO : org.springframework.web.servlet.DispatcherServlet - 
FrameworkServlet 'appServlet': initialization started
INFO : org.springframework.web.context.support.XmlWebApplicationContext
- Refreshing WebApplicationContext for namespace 'appServlet-servlet':
startup date ...
INFO : org.springframework.beans.factory.xml.XmlBeanDefinitionReader .
Loading XML bean definitions from ServletContext resource [/WEB-INF/
spring/appServlet/servlet-context.xml]
INFO : org.springframework.beans.factory.annotation.
AutowireByAnnotationBeanPostProcessor - JSR-330 'javax.inject.Inject'.
annotation found and supported for autowiring

```

Part 02

Part 03

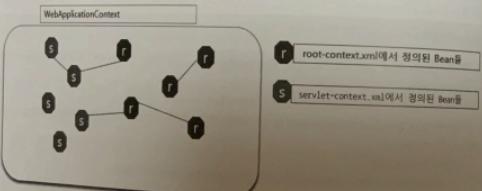
Part 04

DispatcherServlet에서 XmlWebApplicationContext를 이용해서 servlet-context.xml을 로딩하고 해석하기 시작합니다. 이 과정에서 등록된 객체(Bean)들은 기존에 만들어진 객체(Bean)들과 같이 연동되게 됩니다.

Part 05

Part 06

Part 07



5.3 스프링 MVC의 기본 사상

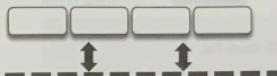
Java를 이용하는 웹 애플리케이션을 제작해 본적이 있다면 Servlet/JSP 기술을 활용해 서 제작하는 방식을 먼저 배우게 됩니다. 이후에는 모델 2라는 방식에 대해서 학습하게 되는데, 스프링 MVC의 경우 이러한 부분은 개발자들에게 보여주지 않고, 개발자들은 자신이 필요한 부분만을 집중해서 개발할 수 있는 구조로 만들어져 있습니다.

웹 프로그래밍을 배워본 적이 있다면 가장 익숙한 단어들 중 하나는 Request/Response 일 것입니다. Servlet/JSP에서는 HttpServletRequest/HttpServletResponse라는 타입의 객체를 이용해 브라우저에서 전송한 정보를 처리하는 방식입니다. 스프링 MVC의

120 Part 2

경우 이 위에 하나의 계층을 더한 형태가 됩니다.

개발자의 코드 영역



개발자는 Servlet/JSP의 API에 신경 쓰지 않고 웹 애플리케이션을 제작

Spring MVC는 내부적으로
Servlet/JSP 처리

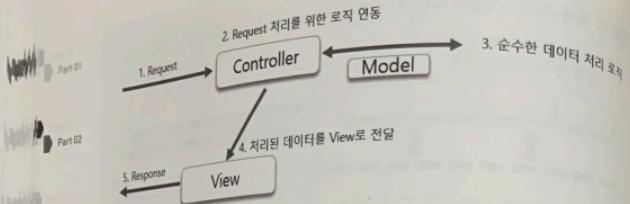
Servlet/JSP

스프링 MVC를 이용하게 되면 개발자들은 직접적으로 HttpServletRequest/
HttpServletResponse 등과 같이 Servlet/JSP의 API를 사용할 필요성이 현저하게 줄어듭니다. 스프링은 중간에 연결 역할을 하기 때문에 이러한 코드를 작성하지 않고도 원하는 기능을 구현할 수 있게 됩니다.

개발자의 코드는 스프링 MVC에서 동작하기 때문에 과거에는 스프링 MVC의 특정한 클래스를 상속하거나 인터페이스를 구현하는 형태로 개발할 수 있었지만, 스프링 2.5버전부터 등장한 어노테이션 방식으로 인해 최근 개발에는 어노테이션이나 XML 등의 설정만으로 개발이 가능하게 되었습니다.

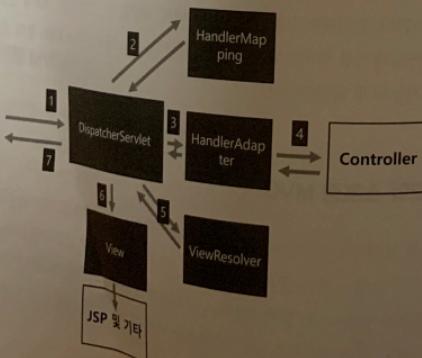
5.4 모델 2와 스프링 MVC

위의 그림에서 보여주듯이 스프링 MVC 역시 내부적으로는 Servlet API를 활용합니다. 스프링 MVC는 '모델 2'라는 방식으로 처리되는 구조로이므로 모델 2방식에 대해서 간단히 살펴볼 필요가 있습니다. 모델 2방식은 쉽게 말해서 '로직과 화면을 분리'하는 스타일의 개발 방식입니다. 모델 2방식은 MVC의 구조를 사용하는데, 이를 그림으로 표현하면 아래와 같습니다.



모델 2방식에서 사용자의 Request는 특별한 상황이 아닌 이상 먼저 Controller를 호출하게 됩니다. 이렇게 설계하는 가장 중요한 이유는 나중에 View를 교체하더라도 사용자가 호출하는 URL 자체에 변화가 없게 만들어 주기 때문입니다. 컨트롤러는 데이터를 처리하는 존재를 이용해서 데이터(Model)를 처리하고 Response 할 때 필요한 데이터(Model)를 View 편으로 전달하게 됩니다. Servlet을 이용하는 경우 개발자들은 Servlet API의 RequestDispatcher 등을 이용해서 이를 직접 처리해 왔지만 스프링 MVC는 내부에서 이러한 처리를 하고, 개발자들은 스프링 MVC의 API를 이용해서 코드를 작성하게 됩니다.

스프링 MVC의 기본 구조는 아래 그림과 같이 표현할 수 있습니다.



- 1 사용자의 Request는 Front-Controller인 DispatcherServlet을 통해서 처리합니다. 생성된 프로젝트의 web.xml을 보면 아래와 같이 모든 Request를 DispatcherServlet이 받도록 처리하고 있습니다.

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/spring/appServlet/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- 2 3 HandlerMapping은 Request의 처리를 담당하는 컨트롤러를 찾기 위해서 존재합니다. HandlerMapping 인터페이스를 구현한 여러 객체들 중 RequestMappingHandlerMapping 같은 경우는 개발자가 @RequestMapping 어노테이션에 적용된 것을 기준으로 판단하게 됩니다. 적절한 컨트롤러가 찾아졌다면 HandlerAdapter를 이용해서 해당 컨트롤러를 동작시킵니다.

- 4 Controller는 개발자가 작성하는 클래스로 실제 Request를 처리하는 로직을 작성하게 됩니다. 이때 View에 전달해야 하는 데이터는 주로 Model이라는 객체에 담아서 전달합니다. Controller는 다양한 타입의 결과를 반환하는데 이에 대한 처리는 ViewResolver를 이용하게 됩니다.

Part 01

ViewResolver는 Controller가 반환한 결과를 어떤 View를 통해서 처리하는 것이 좋을지
해석하는 역할입니다. 가장 흔하게 사용하는 설정은 servlet-context.xml에 정의된 internalResourceViewResolver입니다.

Part 02

```
<beans:bean class="org.springframework.web.servlet.view.  
InternalResourceViewResolver">  
    <beans:property name="prefix" value="/WEB-INF/views/" />  
    <beans:property name="suffix" value=".jsp" />  
</beans:bean>
```

Part 03

Part 04

6.7 View는 실제로 응답 보내야 하는 데이터를 Jsp 등을 이용해서 생성하는 역할을 하게 됩니다. 만들어진 응답은 DispatcherServlet을 통해서 전송됩니다.

Part 05

Part 06

위의 그림을 보면 모든 Request는 DispatcherServlet을 통하여 실계되는데, 이런 방식을 Front-Controller 패턴이라고 합니다. Front-Controller 패턴을 이용하면 전체 흐름을 강제로 제한할 수 있습니다. 예를 들어 HttpServlet을 상속해서 만든 클래스를 이용하는 경우 특정 개발자는 이를 활용할 수 있지만 다른 개발자는 자신이 원래 하던 방식대로 HttpServlet을 그대로 상속해서 개발할 수도 있습니다. Front-Controller 패턴을 이용하는 경우에는 모든 Request의 처리에 대한 분배가 정해진 방식대로만 동작하기 때문에 좀 더 엄격한 구조를 만들어 낼 수 있습니다.

Part 07

Chapter

06

스프링 MVC의 Controller

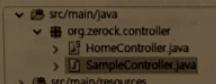
스프링 MVC를 이용하는 경우 작성되는 Controller는 다음과 같은 특징이 있습니다.

- 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 어노테이션으로도 필요한 설정 가능

다른 프레임워크들과 달리 **스프링 MVC는 어노테이션을 중심으로 구성되기 때문에 예제**들을 작성할 때에도 각 어노테이션의 의미에 대해서 주의해야하며 학습해야 합니다.

6.1 @Controller, @RequestMapping

프로젝트 내 org.zerock.controller 패키지 폴더에 SampleController라는 이름의 클래스를 작성합니다.



SampleController 클래스

```
package org.zerock.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;
```

Part 01
Part 02
Part 03
Part 04
Part 05
Part 06
Part 07

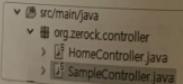
```
@Controller  
@RequestMapping("/sample/*")  
public class SampleController {  
}
```

SampleController의 클래스 선언부에는 @Controller라는 스프링 MVC에서 사용하는 어노테이션을 적용하고 있습니다. 작성된 SampleController 클래스는 위의 그림과 같이 자동으로 스프링의 객체(Bean)로 등록되는데 servlet-context.xml에 그 이유가 있습니다.

servelt-context.xml의 일부
<context:component-scan base-package="org.zerock.controller" />
</beans:beans>

servelt-context.xml에는 <context:component-scan>이라는 태그를 이용해서 지정된 패키지를 조사(스캔)하도록 설정되어 있습니다. 해당 패키지에 선언된 클래스들을 조사하면서 스프링에서 객체(Bean) 설정에 사용되는 어노테이션들을 가진 클래스들을 파악하고 필요하다면 이를 객체로 생성해서 관리하게 됩니다.

SampleController 클래스가 스프링에서 관리되면 화면상에는 클래스 옆에 작게 's' 모양의 아이콘이 추가됩니다.



클래스 선언부에는 @Controller와 함께 @RequestMapping을 많이 사용합니다.
@RequestMapping은 현재 클래스의 모든 메서드들의 기본적인 URL 경로가 됩니다.

예를 들어, SampleController 클래스를 다음과 같이 '/sample/*'이라는 경로로 지정한다면 다음과 같은 URL은 모두 SampleController에서 처리됩니다.

- /sample/aaa
- /sample/bbb

@RequestMapping 어노테이션은 클래스의 선언과 메서드 선언에 사용할 수 있습니다.

SampleController 클래스

```
package org.zerock.controller;  
  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.RequestMapping;  
  
import lombok.extern.log4j.Log4j;  
  
@Controller  
@RequestMapping("/sample/*")  
@Log4j  
public class SampleController {  
  
    @RequestMapping("")  
    public void basic() {  
        log.info("basic.....");  
    }  
}
```

SampleController는 Lombok의 @Log4j를 사용합니다. @Log4j는 @Log가 java.util.Logging을 이용하는데 반해 Log4j 라이브러리를 활용합니다. Spring Legacy Project로 생성한 프로젝트는 기본적으로 Log4j가 추가되어 있으므로 별도의 설정이 필요하지 않습니다.

프로젝트를 WAS에서 실행해보면 스프링이 인식할 수 있는 정보가 출력되는 것을 볼 수 있는데, 위와 같은 경우에는 아래와 같이 로그가 보입니다. src/resources 폴더 내에 log4j.xml의 모든 'info'를 'debug'로 수정하면 아래와 같은 로그가 보입니다.

```
INFO : org.springframework.web.servlet.mvc.method.annotation.  
RequestMappingHandlerMapping - Mapped "[/,methods=[GET]]" onto public  
java.lang.String org.zerock.controller.HomeController.home(java.util.
```

Part 01
Locale, org.springframework.ui.Model)
INFO : org.springframework.web.servlet.mvc.method.annotation.
RequestMappingHandlerMapping - Mapped "[/{sample/*}]" onto public void
org.zerock.controller.SampleController.basic()

현재 프로젝트의 경우 '/와 '/sample/*'는 호출이 가능한 경로라는 것을 확인할 수 있습니다.

Part 02

Part 03

Part 04

Part 05

Part 06

Part 07

6.2 @RequestMapping의 변화

@Controller 어노테이션은 추가적인 속성을 지정할 수 없지만, @RequestMapping의 경우 몇 가지의 속성을 추가할 수 있습니다. 이 중에서도 가장 많이 사용하는 속성이 method 속성입니다. Method 속성은 흔히 GET 방식, POST 방식을 구분해서 사용할 때 이용합니다.

스프링 4.3 버전부터는 이러한 @RequestMapping을 줄여서 사용할 수 있는 @GetMapping, @PostMapping 등장하는데 축약형의 표현이므로, 아래와 같이 비교해서 학습하는 것이 좋습니다.

SampleController 클래스의 일부
@RequestMapping(value = "/basic", method = {RequestMethod.GET})
public void basicGet() {
 log.info("basic get");
}

Get: URL에 변수(데잍)를 포함시켜 요청한다
데이터를 Header에 포함하여 전송
URL에 데이터가 포함되어 요청 가능
캐싱이 가능.

Post: URL 변수를 노출하지 않고 요청
데이터를 Body에 포함시킨다.
보안성↑
캐싱을 하기가 어렵다.

Q) Caching(캐싱)이란?

캐싱이란 한번 접근 후, 요청할 시 빠르게 접근하기 위해 레지스터에 데이터를 저장시켜 놓는 것입니다.

Body에
데이터를 보내며
전송.

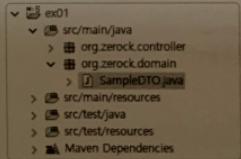
캐싱을 하기가 어렵다.

@RequestMapping은 GET, POST 방식 모두를 지원해야 하는 경우에 배열로 처리해 서 지정할 수 있습니다. 일반적인 경우에는 GET, POST 방식만을 사용하지만 최근에는 PUT, DELETE 방식 등도 점점 많이 사용하고 있습니다. @GetMapping의 경우 오직 GET 방식에만 사용할 수 있으므로, 간편하기는 하지만 기능에 대한 제한은 많은 편입니다.

6.3 Controller의 파라미터 수집

Controller를 작성할 때 가장 편리한 기능은 파라미터가 자동으로 수집되는 기능입니다. 이 기능을 이용하면 매번 request.getParameter()를 이용하는 불편함을 없앨 수 있습니다.

예제를 위해서 org.zerock.domain이라는 패키지를 작성하고, SampleDTO 클래스를 작성합니다.



SampleDTO 클래스

```
package org.zerock.domain;  
  
import lombok.Data;  
  
@Data  
public class SampleDTO {  
  
    private String name;  
    private int age;  
}
```

SampleDTO 클래스는 Lombok의 `@Data` 어노테이션을 이용해서 처리합니다. `@Data`를 이용하게 되면 `getter/setter`, `equals()`, `toString()` 등의 메서드를 자동 생성하는 때문에 편리합니다.

SampleController의 메서드가 SampleDTO를 파라미터로 사용하게 되면 `setter` 메서드가 동작하면서 파라미터를 수집하게 됩니다(이를 확인하고 싶다면 직접 메서드를 제작하고 `set` 메서드 내 간단한 로그 등을 출력해 보면 확인할 수 있습니다).

SampleController의 일부

```
package org.zerock.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.zerock.domain.SampleDTO;

import lombok.extern.log4j.Log4j;

@Controller
@RequestMapping("/sample/*")
@Log4j
public class SampleController {

    //생략~

    @GetMapping("/ex01")
    public String ex01(SampleDTO dto) {
        log.info("+" + dto);
        return "ex01";
    }
}
```

SampleController의 경로가 '/sample/*'이므로 `ex01()` 메서드를 호출하는 경로는 '/sample/ex01'이 됩니다. 메서드에는 `@GetMapping`이 사용되었으므로, 필요한 파라미터를 URL 뒤에 '?name=AAA&age=10'과 같은 형태로 추가해서 호출할 수 있습니다.

localhost:8080/sample/ex01?name=AAA&age=10
HTTP Status 404 – Not Found
INFO : org.zerock.controller.SampleController - SampleDTO(name=AAA, age=10)

실행된 결과를 보면 SampleDTO 객체 안에 name과 age 속성이 제대로 수집된 것을 볼 수 있습니다(특히 주목할 점은 자동으로 타입을 변환해서 처리한다는 점입니다). 프로젝트를 실행할 때 기본 경로는 '/controller'라는 경로로 동작하므로, 이를 앞의 예제와 같이 '/로 동작하도록 변경해서 실행해야 합니다.).

Path	Document Base
/	ex01

6.3.1 파라미터의 수집과 변환

Controller가 파라미터를 수집하는 방식은 파라미터 타입에 따라 자동으로 변환하는 방식을 이용합니다. 예를 들어, SampleDTO에는 int 타입으로 선언된 age가 자동으로 숫자로 변환되는 것을 볼 수 있습니다.

만일 기본 자료형이나 문자열 등을 이용한다면 파라미터의 타입만을 맞게 선언해주는 방식을 사용할 수 있습니다.

SampleController에 추가

```
@GetMapping("/ex02")
public String ex02(@RequestParam("name") String name,
@RequestParam("age") int age) {
    log.info("name: " + name);
    log.info("age: " + age);
}
```

```
return "ex02";
```

ex02() 메서드는 파라미터에 @RequestParam 어노테이션을 사용해서 작성되었습니다. @RequestParam은 파라미터로 사용된 변수의 이름과 전달되는 파라미터의 이름이 같은 경우에 유동하게 사용됩니다(지금 예제의 경우 변수명과 파라미터의 이름이 동일 때문에 사용할 필요는 없었지만 @RequestParam의 소개 차원에서 사용해 보았습니다).

브라우저에서 `http://localhost:8080/sample/ex02?name=AAA&age=10`과 함께 호출하면 이전과 동일하게 데이터가 수집된 것을 볼 수 있습니다.

```
INFO : org.zerock.controller.SampleController - name: AAA  
INFO : org.zerock.controller.SampleController - age: 10
```

6.3.2 리스트, 배열 처리

동일한 이름의 파라미터가 여러 개 전달되는 경우에는 `ArrayList<>` 등을 이용해서 처리 가능합니다.

SampleController에 추가

```
@GetMapping("/ex02List")
public String ex02List(@RequestParam("ids")ArrayList<String> ids) {
    log.info("ids: " + ids);
    return "ex02List";
```

스프링은 파라미터의 타입을 보고 객체를 생성하므로 파라미터의 타입은 `List<>`와 같이 인터페이스 타입이 아닌 실제적인 클래스 타입으로 지정합니다. 위 코드의 경우 'ids'라는 이름의 파라미터가 여러 개 전달되더라도 `ArrayList<String>`이 생성되어 자동으로 수

집됩니다. 브라우저 등을 이용해서 '프로젝트 경로/sample/ex02List?ids=111&ids=222&ids=333'을 호출한다면 아래와 같이 로그가 출력됩니다.

```
INFO : org.zerock.controller.SampleController - ids: [111, 222, 333]
```

배열의 경우도 동일하게 처리할 수 있습니다.

SampleController에 추가

```
@GetMapping("/ex02Array")
public String ex02Array(@RequestParam("ids") String[] ids) {
    log.info("array ids: " + Arrays.toString(ids));
    return "ex02Array";
}
```

6.3.3 객체 리스트

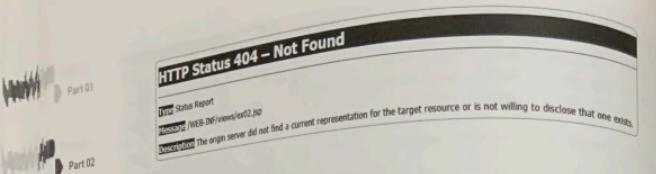
만일 전달하는 데이터가 SampleDTO와 같이 객체 타입이고 여러 개를 처리해야 한다면 약간의 작업을 통해서 한 번에 처리를 할 수 있습니다. 예를 들어 SampleDTO를 여러 개 전달받아서 처리하고 싶다면 다음과 같이 SampleDTO의 리스트를 포함하는 SampleDTOList 클래스를 설계합니다.

```
src/main/java
  org.zerock.controller
    HomeController.java
    SampleController.java
  org.zerock.domain
    SampleDTO.java
    SampleDTOList.java
```

SampleDTOList 클래스

```
package org.zerock.domain;

import java.util.ArrayList;
import java.util.List;
```



Part 01

Part 02

Part 03

Part 04

Part 05

Part 06

Part 07

6.3.5 @DateTimeFormat

@InitBinder를 이용해서 날짜를 변환할 수도 있지만, 파라미터로 사용되는 인스턴스에 @DateTimeFormat을 적용해도 변환이 가능합니다(@DateTimeFormat을 이용하는 경우에는 @InitBinder는 필요하지 않습니다.).

```
package org.zerock.domain;

import java.util.Date;

import org.springframework.format.annotation.DateTimeFormat;

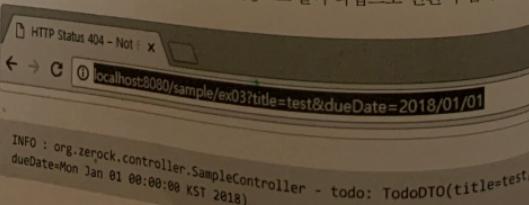
import lombok.Data;

@Data
public class TodoDTO {

    private String title;

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

문자열로 'yyyy/MM/dd'의 형식이 맞다면 자동으로 날짜 타입으로 변환이 됩니다.



6.4 Model이라는 데이터 전달자

Controller의 메서드를 작성할 때는 특별하게 Model이라는 타입을 파라미터로 지정할 수 있습니다. Model 객체는 JSP와 컨트롤러에서 생성된 데이터를 담아서 전달하는 역할을 하는 존재입니다. 이를 이용해서 JSP와 같은 뷰(View)로 전달해야 하는 데이터를 담아보낼 수 있습니다. 메서드의 파라미터에 Model 타입이 지정된 경우에는 스프링은 특별하게 Model 타입의 객체를 만들어서 메서드에 주입하게 됩니다.

Model은 모델 2 방식에서 사용하는 request.setAttribute()와 유사한 역할을 합니다. Servlet을 이용해 본 적이 있다면 다음과 같은 코드에 익숙할 것입니다.

```
Servlet에서 모델 2 방식으로 데이터를 전달하는 방식
request.setAttribute("serverTime", new java.util.Date());
RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/jsp/home.jsp");
dispatcher.forward(request, response);
```

위의 코드를 스프링에서는 Model을 이용해서 다음과 같이 처리하게 됩니다.

스프링 MVC에서 Model을 이용한 데이터 전달

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

메서드의 파라미터를 Model 타입으로 선언하게 되면 자동으로 스프링 MVC에서 Model 타입의 객체를 만들어 주기 때문에 개발자의 입장에서는 필요한 데이터를 담아 주는 작업만으로 모든 작업이 완료됩니다. Model을 사용해야 하는 경우는 주로 Controller에 전달된 데이터를 이용해서 추가적인 데이터를 가져와야 하는 상황입니다.

예를 들어, 다음과 같은 경우들을 생각해 볼 수 있습니다.

리스트 페이지 번호를 파라미터로 전달받고, 실제 데이터를 View로 전달해야 하는 경우

파라미터들에 대한 처리 후 결과를 전달해야 하는 경우

6.4.1 @ModelAttribute 어노테이션

웹페이지의 구조는 Request에 전달된 데이터를 가지고 필요하다면 추가적인 데이터를 생성해서 화면으로 전달하는 방식으로 동작합니다. Model의 경우는 파라미터로 전달된 데이터는 존재하지 않지만 화면에서 필요한 데이터를 전달하기 위해서 사용합니다. 예를 들어 페이지 번호는 파라미터로 전달되지만, 결과 데이터를 전달하려면 Model에 담아서 전달합니다.

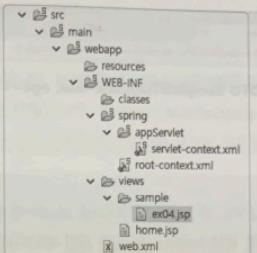
스프링 MVC의 Controller는 기본적으로 Java Beans 규칙에 맞는 객체는 다시 화면으로 객체를 전달합니다. 좁은 의미에서 Java Beans의 규칙은 단순히 생성자가 없거나 빈 생성자를 가야 하며, getter/setter를 가진 클래스의 객체들을 의미합니다. 앞의 예제에서 파라미터로 사용된 SampleDTO의 경우는 Java Bean의 규칙에 맞기 때문에 자동으로 다시 화면까지 전달됩니다. 전달될 때에는 클래스명의 앞글자는 소문자로 처리됩니다.

반면에 기본 자료형의 경우는 파라미터로 선언하더라도 기본적으로 화면까지 전달되지는 않습니다.

SampleController에 추가

```
@GetMapping("/ex04")
public String ex04(SampleDTO dto, int page) {
    log.info("dto: " + dto);
    log.info("page: " + page);
    return "/sample/ex04";
}
```

ex04()는 SampleDTO 타입과 int 타입의 데이터를 파라미터로 사용합니다. 결과를 확인하기 위해서 '/WEB-INF/views' 폴더 아래 sample 폴더를 생성하고 리턴값에서 사용한 'ex04'에 해당하는 ex04.jsp를 작성합니다.



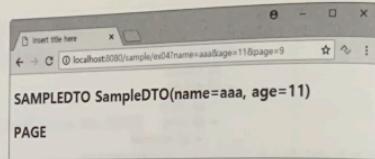
ex04.jsp의 일부

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>

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

</body>
</html>
```

서버를 실행하고 브라우저를 통해서 'http://localhost:8080/sample/ex04?name=aaa&age=11&page=9'와 같이 호출하면 화면에 SampleDTO만이 전달된 것임을 확인할 수 있습니다. int 타입으로 선언된 page는 전달되지 않습니다.

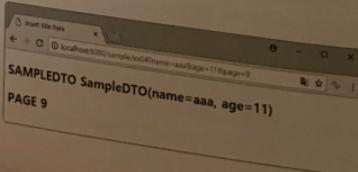


@ModelAttribute는 강제로 전달받은 파라미터를 Model에 담아서 전달하도록 할 때 필요한 어노테이션입니다. @ModelAttribute가 걸린 파라미터는 타입에 관계없이 무조건 Model에 담아서 전달되므로, 파라미터로 전달된 데이터를 다시 화면에서 사용해야 할 경우에 유용하게 사용됩니다.

기존의 코드에서 int 타입의 데이터가 화면까지 전달되지 않았으므로 @ModelAttribute를 추가하면 다음과 같은 형태가 됩니다.

```
@GetMapping("/ex04")
public String ex04(SampleDTO dto, @ModelAttribute("page") int page) {
    log.info("dto: " + dto);
    log.info("page: " + page);
    return "/sample/ex04";
}
```

@ModelAttribute가 붙은 파라미터는 화면까지 전달되므로 브라우저를 통해서 호출하면 아래와 같이 \${page}가 출력되는 것을 확인할 수 있습니다(기본 자료형에 @ModelAttribute를 적용한 경우에는 반드시 @ModelAttribute("page")와 같이 값(value)을 지정하도록 합니다).



6.4.2 RedirectAttributes

Model 타입과 더불어서 스프링 MVC가 자동으로 전달해 주는 타입 중에는 Redirect Attributes 타입이 존재합니다. RedirectAttributes는 조금 특별하게도 일회성으로 데이터를 전달하는 용도로 사용됩니다. RedirectAttributes는 기존에 Servlet에서는 response.sendRedirect()를 사용할 때와 동일한 용도로 사용됩니다.

Servlet에서 redirect 방식

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

스프링 MVC를 이용하는 경우에는 다음과 같이 변경됩니다.

스프링 MVC를 이용하는 redirect 처리

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

return "redirect:/";
```

RedirectAttributes는 Model과 같이 파라미터로 선언해서 사용하고, addFlashAttribute(이름, 값) 메서드를 이용해서 화면에 한 번만 사용하고 다음에는 사용되지 않는 데이터를 전달하기 위해서 사용합니다. RedirectAttributes의 용도는 PART 3에서 예제를 작성할 때 여러 번 사용할 것입니다.

6.5 Controller의 리턴 타입

스프링 MVC의 구조가 기존의 상속과 인터페이스에서 어노테이션을 사용하는 방식으로 변한 이후에 가장 큰 변화 중 하나는 리턴 타입이 자유로워졌다라는 점입니다.

Controller의 메서드가 사용할 수 있는 리턴 타입은 주로 다음과 같습니다.

- String: jsp를 이용하는 경우에는 jsp 파일의 경로와 파일 이름을 나타내기 위해서 사용합니다.
- void: 호출하는 URL과 동일한 이름의 jsp를 의미합니다.

- VO, DTO 타입: 주로 JSON 타입의 데이터를 만들어서 반환하는 용도로 사용합니다.
- ResponseEntity 타입: response 할 때 Http 헤더 정보와 내용을 제공하는 용도로 사용합니다.
- Model, ModelAndView: Model로 데이터를 반환하거나 화면까지 같이 지정하는 경우에 사용합니다 (최근에는 많이 사용하지 않습니다).
- HttpHeaders: 응답에 내용 없이 Http 헤더 메시지만 전달하는 용도로 사용합니다.

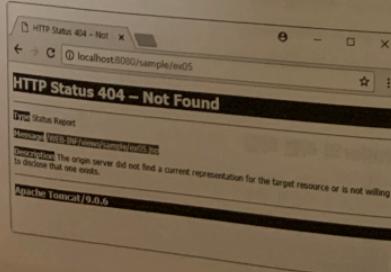
6.5.1 void 타입

예서드의 리턴 타입을 void로 지정하는 경우 일반적인 경우에는 해당 URL의 경로를 그대로 jsp 파일의 이름으로 사용하게 됩니다.

SampleController의 일부

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

브라우저에서 SampleController의 경로에 ex05()의 경로를 합쳐 '/sample/ex05'를 호출하면 다음과 같은 결과를 보게 됩니다.



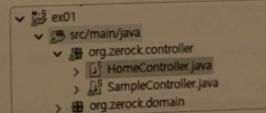
에러 메시지를 자세히 보면 에러 메시지의 원인인 '/WEB-INF/views/sample/ex05.jsp'가 존재하지 않아서 생기는 문제라는 것을 볼 수 있습니다. 이것은 servlet-context.xml의 아래 설정과 같이 맞물려 URL 경로를 View로 처리하기 때문에 생기는 결과입니다.

servlet-context.xml의 일부

```
<beans:bean class="org.springframework.web.servlet.view.  
InternalResourceViewResolver">  
    <beans:property name="prefix" value="/WEB-INF/views/" />  
    <beans:property name="suffix" value=".jsp" />  
</beans:bean>
```

6.5.2 String 타입

void 타입과 더불어서 가장 많이 사용하는 것은 String 타입입니다. String 타입은 상황에 따라 다른 화면을 보여줄 필요가 있을 경우에 유용하게 사용합니다(if ~ else와 같은 처리가 필요한 상황). 일반적으로 String 타입은 현재 프로젝트의 경우 JSP 파일의 이름을 의미합니다. 프로젝트 생성 시 기본으로 만들어진 HomeController의 코드를 보면 String을 반환 타입으로 사용하는 것을 볼 수 있습니다.



HomeController의 일부

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String home(Locale locale, Model model) {
    logger.info("Welcome home! The client locale is {}.", locale);

    Date date = new Date();
    DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.
    LONG, DateFormat.LONG, locale);

    String formattedDate = dateFormat.format(date);
```

```
Part 01
model.addAttribute("serverTime", formattedDate );
```

```
Part 02
return "home";
}
```

home() 메서드는 'home'이라는 문자열을 리턴했기 때문에 경로는 '/WEB-INF/views/home.jsp' 경로가 됩니다.

String 타입에는 다음과 같은 특별한 키워드를 붙여서 사용할 수 있습니다.

- redirect: 리다이렉트 방식으로 처리하는 경우
- forward: 포워드 방식으로 처리하는 경우

6.5.3 객체 탐색

Controller의 메서드 리턴 탐색을 VO(Value Object)나 DTO(Data Transfer Object) 탐색 등 복잡적인 데이터가 들어간 객체 탐색으로 지정할 수 있는데, 이 경우는 주로

JSON 데이터를 만들어 내는 용도로 사용합니다.

우선 이를 위해서는 jackson-databind 라이브러리를 pom.xml에 추가합니다.

pom.xml에 추가하는 Jackson-databind 라이브러리

```
<!-- 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>
```

SampleController에는 아래와 같은 메서드를 생성합니다.

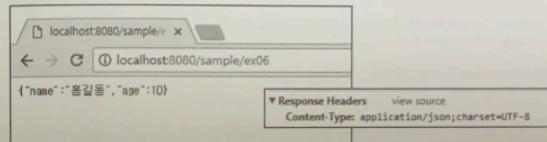
SampleController의 일부

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

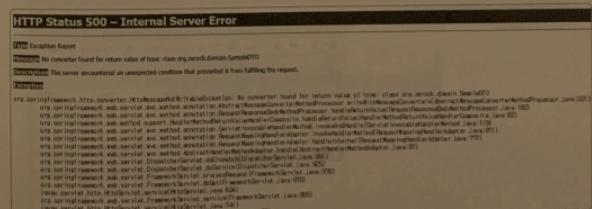
```
SampleDTO dto = new SampleDTO();
dto.setAge(10);
dto.setName("홍길동");

return dto;
}
```

스프링 MVC는 자동으로 브라우저에 JSON 탐색으로 객체를 변환해서 전달하게 됩니다.



개발자 도구를 통해서 살펴보면 서버에서 전송하는 MIME 탐색이 'application/json'으로 처리되는 것을 볼 수 있습니다. 만일 Jackson-databind 라이브러리가 포함되지 않았다면 다음과 같은 에러 화면을 보게 됩니다. 스프링 MVC는 리턴 탐색에 맞게 데이터를 변환해 주는 역할을 지정할 수 있는데 기본적으로 JSON은 처리가 되므로 별도의 설정이 필요로 하지 않습니다(스프링 3버전까지는 별도의 Converter를 작성해야만 했습니다.).



Part 01
Part 02
Part 03
Part 04
Part 05
Part 06
Part 07

6.5.4 ResponseEntity 태입
Web을 다루다 보면 HTTP 프로토콜의 헤더를 다루는 경우도 종종 있습니다. MVC의 사상은 HttpServletRequest나 HttpServletResponse를 직접 핸들링하기 어렵도록 작성되었기 때문에 이러한 처리를 위해 ResponseEntity 태입이 가능하도록 전달할 수 있습니다.

SampleController의 일부

```
@GetMapping("/ex07")
public ResponseEntity<String> ex07() {
    log.info("/ex07.....");
    // {"name": "홍길동"}
    String msg = "{\"name\": \"홍길동\"}";
    HttpHeaders header = new HttpHeaders();
    header.add("Content-Type", "application/json; charset=UTF-8");
    return new ResponseEntity<>(msg, header, HttpStatus.OK);
}
```

ResponseEntity는 HttpHeaders 객체를 같이 전달할 수 있고, 이를 통해서 원하는 HTTP 헤더 메시지를 가공하는 것이 가능합니다. ex07()의 경우 브라우저에는 JSON 타입이라는 헤더 메시지와 200 OK라는 상태 코드를 전송합니다.

