++++++++++++++++

+Part02 SPRING의 MVC +

++++++++++++++++

++++++++++++++++++++++++++++++++

+Chapter 06 SPRING MVC를 이용한 WEB 요청 처리+

++++++++++++++++++++++++++++++++

: SPRING Framework는 DI나 AOP와 같은 기능뿐만 아니라 WEB 개발을 위한 MVC Framework도 함께 제공하고 있다.

SPRING MVC Framework는 SPRING을 기반으로 하고 있기 때문에 SPRING이 제공하는 Transaction 처리나 DI 및 AOP 등을

손쉽게 사용할 수 있다는 장점을 갖는다. 또한, Struts와 같은 Framework와 SPRING Framework를 연동하기 위해

추가적인 설정을 하지 않아도 된다는 장점을 갖고 있다. 이런 이유로 필자는 다른 WEB Framework보다 SPRING MVC를

선호하고 있다.

본 장에서는 SPRING MVC가 어떻게 동작하는 지를 살펴보고, SPRING MVC의 핵심 구성 요소인 컨트롤러의 구현 방법에

대해서 살펴볼 것이다.

===============================================================================

1. SPRING MVC의 주요 구성 요소 및 처리 흐름

: 다른 MVC 기반의 Framework와 마찬가지로 SPRING MVC도 컨트롤러를 사용하여 클라이언트의 요청을 처리하게 된다.

SPRING에서 DispatcherServlet이 MVC에서 C(Controller) 부분을 처리하는데, DispacherServlet을 비롯한 MVC의 주요

구성 요소는 다음과 같다.

--------------------------------------------------------------------------------------------------------------------------------------------------

[구성 요소] [설 명]

=========================================================================

DispacherServlet 클라이언트의 요청을 전달 받는다. 컨트롤러에게 클라이언트의 요청을 전달하고, 컨트롤러가

리턴한 결과 값을 View에 전달하여 알맞은 응답을 생성하도록 한다.

--------------------------------------------------------------------------------------------------------------------------------------------------

HandlerMapping 클라이언트의 요청 URL을 어떤 컨트롤러가 처리할 지를 결정한다.

--------------------------------------------------------------------------------------------------------------------------------------------------

Controller 클라이언트의 요청을 처리한 뒤, 그 결과를 DispatcherServlet에 알려 준다.

Struts의 Action과 동일한 역할을 수행한다.

--------------------------------------------------------------------------------------------------------------------------------------------------

ModelAndView 컨트롤러가 처리한 결과 정보 및 뷰 선택에 필요한 정보를 담는다.

--------------------------------------------------------------------------------------------------------------------------------------------------

ViewResolver 컨트롤러의 처리 결과를 생성할 뷰를 결정한다.

--------------------------------------------------------------------------------------------------------------------------------------------------

View 컨트롤러의 처리 결과 화면을 생성한다. JSP나 Velocity 템플릿 파일 등을 뷰로 사용한다.

--------------------------------------------------------------------------------------------------------------------------------------------------

\*\*\*SPRING MVC의 주요 구성 요소

이들 각 구성 요소의 Message 흐름은 다음과 같다.

[[:HandlerMapping]] [[:Controller]]

|| ||

|| ||

|| 2: 요청URL과 Mapping되는 3:처리요청|| 4: ModelAndView 리턴

|| Controller검색 ||

|| ||

|| ||

|| ||

|| ||

|| ||

[[클라이언트]] ============= [[: DispacherServlet]]

1 : 처리 요청 || ||

|| ||

|| ||

|| ||

|| ||

|| ||

|| ||

|| 6: 응답 출력 요청 || 5: Controller의 실행 결과를

|| || 보여줄 View 검색

[[:View]] [[:ViewResolver]]

\*\*\*SPRING MVC의 클라이언트 요청 처리 과정

각 흐름을 좀더 자세하게 설명하면 다음과 같다.

- 클라이언트의 요청이 DispacherServlet에 전달된다.

- DispatcherServlet은HandlerMapping을 사용하여 클라이언트의 요청을 처리할 컨트롤러 객체를 구한다.

- DispatcherServlet은 컨트롤러 객체를 이용해서 클라이언트의 요청을 처리한다.

- 컨트롤러는 클라이언트의 요청 처리 결과 정보를 담은 ModelAndView 객체를 리턴한다.

- DispatcherServlet은 ViewResolver로부터 응답 결과를 생성할 뷰 객체를 구한다.

- 뷰는 클라이언트에 전송할 응답을 생성한다.

이 과정에서 개발자가 직접해야 개발해야 할 부분은 클라이언트의 요청을 처리할 컨트롤러와 클라이언트에 응답 결과 화면을

전송할 JSP나 Velocity 템플릿 등의 뷰 Code이다. 나머지, DispatcherServlet이나 HandlerMapping, ViewResolver 등은 SPRING이

기본적으로 제공하는 구현 Class를 사용하게 된다.

===============================================================================

2. SPRING MVC Hello World

: SPRING MVC를 이용하여 WEB Application을 개발하는 과정을 단순하게 설명하면 다음과 같이 5단계로 나눌 수 있다.

1) 클라이언트의 요청을 받을 DispatcherServlet을 web.xml 파일에 설정한다.

2) 클라이언트의 요청을 처리할 컨트롤러를 작성한다.

3) ViewResolver를 설정한다. ViewResolver는 컨트롤러가 전달한 값을 이용해서 응답 화면을 생성할 뷰를 결정한다.

4) JSP나 Velocity등을 이용하여 뷰 영역의 Code를 작성한다.

5) 실행

본 절에서는 단순한 예제인 Hello World를 이용하여 각 단계를 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.1 단계1 : DispatcherServlet 설정 및 SPRING 컨텍스트 설정

: SPRING MVC를 사용하기에 앞서 가장 먼저 해야 할 작업은 Java WEB Application의 설정 파일인 web.xml 파일에 다음의

두 가지 정보를 추가하는 것이다.

- 클라이언트의 요청을 전달 받을 DispatcherServlet 설정

- 공통으로 사용할 Application 컨텍스트 설정

DispatherServlet은 클라이언트의 요청을 전달받는 Servlet으로서, 컨트롤러와 뷰와 같은 SPRING MVC의 구성 요소를 이용하여

클라이언트에게 서비스를 제공하게 된다. DispacherServlet의 설정은 WEB Application의 /WEB-INF/web.xml 파일에 추가하며,

다음과 같이 Servlet과 Servlet Mapping정보를 추가하면 DispacherServlet 설정이 완료된다.

1[/WEB-INF/web.xml]

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"

id="WebApp\_ID" version="2.5">

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>\*.do</url-pattern>

</servlet-mapping>

</web-app>

위 Code에서는 \*.do로 들어오는 클라이언트의 요청을 DispatcherServlet이 처리하도록 설정하였다.

DispatcherServlet은 WEB-INF/ 디렉터리에 위치한 [Servlet이름]-servlet.xml 파일을 SPRING 설정 파일로 사용한다.

예를 들어, 위 Code의 경우 'dispatcher-servlet.xml' 파일을 설정 파일로 사용하게 된다. 이 파일에서 MVC의 구성 요소인

Controller, ViewResolver, View등의 빈을 설정하게 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.2 단계2 : 컨트롤러 구현 및 설정 추가

: 컨트롤러를 구현하려면 먼저 @Controller Annotation을 Class에 적용한다. 그리고, @RequestMapping Annotation을

이용해서 클라이언트의 요청을 처리할 메서드를 지정한다. 아래 Code는 간단한 컨트롤러를 구현한 Class의 예를 보여주고 있다.

2[HelloController.java]

package madvirus.spring.chap06.controller;

import java.util.Calendar;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class HelloController {

@RequestMapping("/hello.do")

public ModelAndView hello() {

ModelAndView mav = new ModelAndView();

mav.setViewName("hello");

mav.addObject("greeting", getGreeting());

return mav;

}

private String getGreeting() {

int hour = Calendar.getInstance().get(Calendar.HOUR\_OF\_DAY);

if (hour >= 6 && hour <= 10) {

return "좋은 아침입니다.";

} else if (hour >= 12 && hour <= 15) {

return "점심 식사는 하셨나요?";

} else if (hour >= 18 && hour <= 22) {

return "좋은 밤 되세요";

}

return "안녕하세요";

}

}

라인09의 @Controller Annotation은 해당 Class가 SPRING MVC의 컨트롤러를 구현한 Class라는 것을 지정한다.

또한, @RequestMapping Annotation은 값으로 지정한 요청 경로를 처리할 메서드를 설정한다. 라인12의 경우

처리할 경로의 값으로 "/hello.do"를 지정했는데, 이 경우 http://host:port[/컨텍스트 경로]/hello.do 요청을

HelloController Class의 hello() 메서드가 처리하게 된다.

라인14~16에서 ModelAndView를 생성하고 있는데, ModelAndView는 컨트롤러의 처리 결과를 보여줄 뷰와 뷰에서

출력할 모델을 지정할 때 사용된다. 위의 Code의 경우 사용할 뷰 이름으로 "hello"를 지정하였고, 모델에 "greeting"이라는

이름으로 String 타입의 값을 추가하였다.

SPRING MVC는 ModelAndView 뿐만 아니라 String이나 ModelMap, 또는 Map과 같은 타입을 이용해서 뷰 이름과 모델 정보를

설정할 수 있도록 하고 있다. 이에 대한 내용은 본 장의 '컨트롤러 구현' 절에서 자세히 살펴볼 것이다.

DispatcherServlet은 SPRING Container에서 컨트롤러 객체를 검색하기 때문에 SPRING 설정 파일에 컨트롤러를 빈으로 등록해

주어야 한다. 다음 Code는 설정 파일의 작성 예를 보여주고 있다.

0

3[/WEB-INF/dispatcher-servlet.xml]

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="helloController" class="madvirus.spring.chap06.controller.HelloController" />

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/view/" />

<property name="suffix" value=".jsp" />

</bean>

</beans>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.3 단계3 : 설정 파일에 ViewResolver 설정 추가

: 컨트롤러 Class는 직접 또는 간접적으로 ModelAndView 객체를 생성하게 된다. 예를 들어, 앞서 작성한

HelloController Class는 다음과 같이 ModelAndView 객체를 생성해서 리턴하였다.

public ModelAndView hello() {

ModelAndView mav = new ModelAndView();

mav.setViewName("hello");

mav.addObject("greeting", getGreeting());

return mav;

}

여기서 ModelAndView.setViewName() 메서드를 이용해서 컨트롤러의 처리 결과를 보여줄 뷰 이름을 'hello'로

지정하였는데, DispacherServlet은 이 뷰 이름과 메칭되는 뷰 구현체를 찾기 위해 ViewResolver를 사용한다.

SPRING MVC는 JSP, Velocity, FreeMaker등의 뷰 구현 기술과의 연동을 지원하는데, JSP를 뷰 기술로 사용할 경우

다음과 같이 InternalResourceViewResolver 구현체를 빈으로 등록해 주면 된다.

<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="/WEB-INF/view/" />

<property name="suffix" value=".jsp" />

</bean>

위 Code에서 prefix 프로퍼티와 suffix 프로퍼티의 값으로 각각 "/WEB-INF/view/"와 ".jsp"를 설정하였는데, 이는

이 ViewResolver가 "/WEB-INF/vew/뷰이름.jsp"를 뷰 JSP로 사용한다는 것을 의미한다. 즉. 앞의 예에서 HelloController는

뷰 이름으로 "hello"를 리턴하므로, 실제로 사용되는 뷰 파일은 "/WEB-INF/view/hello.jsp" 파일이 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.4 단계4 : 뷰 Code 구현

: HelloController의 처리 결과를 보여줄 뷰 JSP Code는 아래 Code와 같다.

4[/WEB-INF/view/hello.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>인사</title>

</head>

<body>

인사말: <strong>${greeting}</strong>

</body>

</html>

hello.jsp의 라인08을 보면 ${greeting} Code를 이용하고 있는데, 이때 "greeting"이라는 이름은 앞서 HelloController에서

추가한 모델의 이름과 동일한 것을 확인할 수 있다.

[컨트롤러]

@RequestMapping("/hello.do")

public ModelAndView hello() {

ModelAndView mav = new ModelAndView();

mav.setViewName("hello");

mav.addObject("greeting", getGreeting());

return mav; ||

} ||

||

[뷰] ||

<body> ||

인사말: <strong>${greeting}</strong>

</body>

\*\*\*모델과 뷰 Code의 연결

위의 Code에서 보듯이 뷰 Code에서는 모델을 추가할 때 사용한 이름을 이용서 해당 모델의 값을 출력할 수 있게 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.5 단계5 : 실행

: 이제 남은 작업은 실제로 실행해서 결과를 확인하는 것이다. WEB 브라우저에서 http://localhost:8080/SpringHello/hello.do를

입력해보자. 그러면, 아래와 같은 결과 화면이 출력될 것이다.(실행 시간에 따라서 Message는 달라질 수 있다.)

인사말 : 좋은 아침입니다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

2.6 단계6 : 실행 흐름 정리

: 지금까지 SPRING MVC의 기본 구성 요소를 이용해서 Hello World를 구현해 보았는데, 전반적인 실행 흐름을 정리하면

아래와 같다.

[[:DefaultAnnotationHandlerMapping]] [[:HelloController]]

|| ||

|| ||

3.HelloController || 2: 요청URL과 Mapping되는 4:처리요청|| 5: ModelAndView 리턴

빈 리턴 || Controller검색 ||

|| ||

|| ||

|| ||

|| ||

|| ||

[[WEB브라우저]] ==========[[dispatcher: DispacherServlet]]

1 : /hello.do || ||

|| ||

|| ||

|| ||

|| ||

|| ||

|| ||

|| 8: 응답 생성 요청 7: 뷰리턴 || 6: Controller의 실행 결과를

|| || 보여줄 View 검색

[[:InternalResourceView]] [[viewResolver:InternalResourceViewResolver]]

||

|| 9: JSP를 이용하여 응답 생성

||

[[:hello.jsp]]

\*\*\*/hello.do 요청 처리 과정

어떤 Controller 구현을 사용하든, 어떤 뷰 기술을 사용하든, 요청 처리 과정은 위의 그림과 동일하므로, 위 흐름을 숙지하면

좀 더 수월하게 SPRING MVC를 적용할 수 있을 것이다.

===============================================================================

3. DispatcherServlet 설정과 ApplicationContext의 관계

: DispatcherServlet은 클라이언트의 요청을 중앙에서 처리하는 SPRING MVC의 핵심 구성 요소이다. web.xml 파일에 한 개 이상의

DispatcherServlet을 설정할 수 있으며, 각 DispatcherServlet은 한대의 WebApplicationContext를 갖게 된다. 또한, 각 DispatcherServlet과

WEB Application을 위한 설정 방법 및 둘 사이의 관계에 대해서 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

3.1 DispatcherServlet 설정

: DispatcherServlet은 기본적으로 WEB Application의 /WEB-INF/ 디렉터리에 위치한 [Servlet이름]-servlet.xml 파일로부터 SPRING 설정

정보를 읽어 온다. 예들 들어, 다음과 같이 web.xml 파일을 설정 했다면, dispatcher-servlet.xml 파일로부터 설정 정보를 읽어 오게 된다.

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

하지만, 경우에 따라 한 개 이상의 설정 파일을 사용해야 하는 경우가 있다. 또는 기본 설정 파일 이름이 아닌 다른 이름의 설정 파일을

사용하고 싶은 경우도 있을 것이다. 이런 경우, 다름과 같이 DispatcherServlet을 설정할 때 contextConfigLocation 초기화 파라미터에

설정 파일 목록을 지정하면 된다.

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/main.xml /WEB-INF/bbs.xml</param-value>

</init-param>

</servlet>

contextConfigLocation 초기화 파라미터는 설정 파일 목록을 값으로 갖는데, 이때 각 설정 파일은 콤마(","), 공백 문자(" "), 줄 바꿈(\n),

세미콜론(";")을 이용하여 구분한다. 각 설정 파일의 경로는 WEB Application 루트 디렉터리를 기준으로 한다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

3.2 WEB Application을 위한 ApplicationContext 설정

: DispatcherServlet은 그 자체가 Servlet이기 때문에 한 개 이상의 DispatcherServlet을 설정하는 것이 가능하다. 예를 들어, WEB 페이지를 위한

DispatcherServlet과 REST 기반의 WEB 서비스 연동을 위한 DispatcherServlet을 나누어 설정했다고 하자.

<servlet>

<servlet-name>front</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/front.xml</param-value>

</init-param>

</servlet>

<servlet>

<servlet-name>rest</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/rest.xml</param-value>

</init-param>

</servlet>

이 경우 DispatcherServlet은 각각 별도의 WebApplicationContext를 생성하게 된다. 예를 들어, 위 Code는

"front" DispatcherServlet은 front.xml 설정 파일을 사용하고, "rest" DispatcherServlet은 rest.xml 설정 파일을

사용하고 있는데, 이때, front.xml에서는 rest.xml에 설정된 빈 객체를 사용할 수 없게 된다.

일반적인 WEB Application에서 컨트롤러는 클라이언트의 요청을 비지니스 로직을 구현한 서비스 layer를 이용하여

처리하는 것이 일반적이다. 또한, 서비스 layer는 영속성 layer를 사용해서 데이터 접근을 처리하곤 한다. 따라서,

front 관련 컨트롤러와 rest 관련 컨트롤러는 아래와 같이 동일한 서비스 layer에 대한 의존 관계를 가질 것이다.

[[front 컨트롤러]] [[rest 컨트롤러]]

|| ||

|| ||

|| ||

|| ||

|| ||

[[ 서비스 layer ]]

||

||

[[ 영속성 layer ]]

\*\*\*WEB Application의 전형적인 layer 구성

위와 같이 서로 다른 DispatcherServlet이 공통 빈을 필요로 하는 경우, ContextLoaderListener를 사용하여 공통으로

사용될 빈을 정할 수 있게 된다. 다음과 같이 ContextLoaderListener를 ServletListener로 등록하고 contextConfigLocation

컨텍스트 파라미터를 이용하여 공통으로 사용될 빈 정보를 담고 있는 설정 파일 목록을 지정하면 된다.

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/service.xml, /WEB-INF/persistence.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<servlet>

<servlet-name>front</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

<servlet>

<servlet-name>rest</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

</servlet>

실제로 ContextLoaderListener와 DispatcherServlet은 각각 WebApplicationContext 객체를 생성하는데,

이때 생성되는 WebApplicationContext 객체 간의 관계는 다음과 같다.

[[front : WebApplicationContext]]

||

||

|| parent

||

||

[[root : WebApplicationContext]]

||

||

|| parent

||

||

[[rest : WebApplicationContext]]

\*\*\*생성되는 WebApplicationContext 간의 관계

ContextLoaderListener가 생성하는 WebApplicationContext는 WEB Application에서 루트 컨텍스트가 되며,

DispatcherServlet이 생성하는 WebApplicationContext는 루트 컨텍스트를 부모로 사용하는 자식 컨텍스트가 된다.

이때 자식은 root가 제공하는 빈을 사용할 수 있기 때문에 각각의 DispatcherServlet이 공통으로 필요로 하는 빈을

ContextLoaderListener를 이용하여 설정하는 것이다.

ContextLoaderListener는 contextConfigLocation 컨텍스트 파라미터를 명시하지 않으면 /WEB-INF/applicationContext.xml을

설정 파일로 사용한다. 또한, Class패스에 위치한 파일로부터 설정 정보를 읽어 오고 싶다면 다음과 같이 'classpath:' 접두어를

사용하여 설정 파일을 명시하면 된다.

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

classpath:config/service.xml

classpath:common.xml

/WEB-INF/config/message\_conf.xml

</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

===============================================================================

4. 캐릭터 인코딩 처리를 위한 필터 설정

: 요청 파라미터의 캐릭터 인코딩이 ISO-8859-1이 아닌 경우, 다음과 같이 request.setCharacterEncoding() 메서드를

사용해서 요청 파라미터의 캐릭터 인코딩을 알맞게 설정해 주어야 한다.

request.setCharacterEncoding("UTF-8");

모든 컨트롤러에서 위 Code를 실행할 수도 있지만, 그것 보다는 Servlet 필터를 이용해서 원하는 요청에 위 Code를

적용하는 것이 더 편리하다. SPRING은 요청 파라미터의 캐릭터 인코딩을 설정할 수 있는 필터 Class인

CharacterEncodingFilter Class를 제공하고 있다.

다음과 같이 web.xml 파일에 CharacterEncodingFilter Class를 설정함으로써 요청 파라미터의 캐릭터 인코딩을

손쉽게 설정할 수 있다.

<filter>

<filter-name>encodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

<init-param>

<param-name>encoding</param-name>

<param-value>EUC-KR</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>encodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

===============================================================================

5. 컨트롤러 구현

: SPRING 3.0버전부터는 @Controller Annotation을 이용해서 컨트롤러 Class를 구현하도록 권장하고 있으며 기존의

Controller Interface와 AbstractController Class, AbstractCommandController Class 등은 이제 고전 컨트롤러

(ClassicController)로 불리고 있다. 본 교재에서도 @Controller Annotation을 이용한 컨트롤러의 구현이 궁금한 학생은

SPRING API 문서를 참고하기 바란다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.1 @Controller Annotation과 @RequestMapping Annotation

: 컨트롤러 Class를 구현하려면 @Controller Annotation과 @RequestMapping Annotation을 이용한다.

- 컨트롤러 Class에 @Controller Annotation을 적용한다.

- 클라이언트의 요청을 처리할 메서드에 @RequestMapping Annotation을 적용한다.

- 설정 파일에 컨트롤러 Class를 빈으로 등록한다.

아래 Code는 가장 간단한 컨트롤러 구현 Class의 예이다.

import org.springframework.stereotype.Controller;

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

@Controller

public class HelloController {

@RequestMapping("/hello.do")

public String hello() {

return "hello";

}

}

@RequestMapping Annotation은 해당 메서드에서 처리할 URI를 값으로 갖는다. 위 Code에서 hello() 메서드는 클라이언트의

/hello.do 요청을 처리하도록 설정하였다. 실제로 처리할 URI는 web.xml에서 설정한 DispatcherServlet에 설정한 Mapping과

SPRING 설정 파일의 전체 경로 이용 여부 설정에 따라서 달라진다. 이에 대한 내용은 본 장의 '전체 경로와 Servlet 기반 경로

매칭 설정'절에서 살펴보도록 하겠다.

위 Code에서 hello() 메서드의 리턴 타입은 String인데, 이 경우 메서드의 리턴 값을 컨트롤러의 처리 결과를 보여줄 뷰 이름으로

사용하게 된다. 컨트롤러 메서드의 파라미터와 리턴 타입은 상황에 따라 알맞은 타입을 선택할 수 있는데, 이에 대한 내용은

뒤에서 살펴볼 것이며, 일단은 가장 간단한 구현 방법부터 살펴보도록 하자.

컨트롤러 Class를 구현하면 DispatcherServlet이 사용하는 SPRING 설정 파일에 해당 컨트롤러 Class를 등록해 준다.

<bean id="helloController" class="madvirus.spring.chap06.controller.HelloController"/>

SPRING 설정 파일에 등록이 완료되면, @RequestMapping Annotation의 값으로 설정한 URI와 매칭되는 클라이언트의 요청을

해당 메서드에서 처리하게 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.2 컨트롤러 메서드의 HTTP 전송 방식(method) 한정

: 하나의 요청 URL에 대해 HTTP GET 요청과 POST 요청을 한 개의 컨트롤러에서 처리해 주어야 할 때가 있다. 예를 들어,

GET 요청이 들어오면 글쓰기 폼을 보여주고 POST 요청이 들어오면 글쓰기 폼 전송을 처리하고 싶을 때가 있다. 이런 경우

@RequestMapping Annotation의 method 속성을 이용해서 메서드가 처리할 HTTP 메서드를 제한할 수 있다.

아래 Code는 @RequestMapping Annotation의 method 속성의 사용 예를 보여주고 있다.

import org.springframework.stereotype.Controller;

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

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

@Controller

public class NewArticleController {

@RequestMapping(value="/article/newArticle.do", method = RequestMethod.GET)

public String form() {

return "article/newArticleForm";

}

@RequestMapping(value="/article/newArticle.do", method = RequestMethod.POST)

public String submit() {

return "article/newArticleSubmitted";

}

}

NewArticleController Class의 form() 메서드나 submit() 메서드는 둘 다 /article/newArticle.do로 요청을 처리하도록 설정하였다.

단, GET 요청이 들어올 경우 form() 메서드가 처리하고 POST 요청이 들어올 경우 submit() 메서드가 처리하게 된다.

예를 들어, WEB 브라우저에서 http://host/chap06/article/newArticle.do를 입력하면 form() 메서드가 실행된다. HTML 폼에 다음과

같이 전송 방식을 POST로 지정한 경우, 폼 전송 버튼을 클릭하면 method 속성의 값으로 RequestMethod.POST를 지정한 submit()

메서드가 실행된다.

<form action="/chap06/article/newArticle.do" method="POST">

...

<input type="submit"/>

</form>

클라이언트의 요청을 처리하는 두 메서드가 동일한 URI를 처리할 경우, @RequestMapping Annotation을 Class에 적용해서

해당 Class가 처리할 기본 URI를 지정할 수 있다.

@Controller

@RequestMapping("/article/newArticle.do")

public class NewArticleController {

@RequestMapping(method = RequestMethod.GET)

public String form() {

...

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") NewArticleCommand command) {

articleService.writeArticle(command);

...

}

...

}

위 Code의 경우 @RequestMapping Annotation을 이용해서 NewArticleController Class가 처리할 기본 URI를 '/article/newArticle.do'로

지정하였다. 그리고, form() 메서드와 submit() 메서드의 @RequestMapping Annotation은 URI를 설정하지 않았고 method 속성을

이용해서 처리할 전송방식만 지정하였다. 이 경우 form() 메서드와 submit() 메서드는 NewArticleController Class의 @RequestMapping

Annotation에서 지정한 URI인 '/article/newArticle.do'를 처리하게 된다.

@RequestMapping Annotation에 method 속성을 설정하지 않을 경우 GET, POST, DELETE 등 모든 HTTP 전송 방식을 처리하게 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.3 HTML 폼과 커맨드 객체(JavaBean)

: SPRING MVC는 HTML 폼에 입력한 데이터를 Java빈 객체를 이용해서 전달 받을 수 있도록 하고 있다. 예을 들어, 다음과 같이 HTML 폼의

항목 이름과 Java빈 Class의 프로퍼티 이름이 일치할 경우 폼에 입력한 값을 해당 Java빈 Class의 프로퍼티 값으로 설정해주는 기능을

제공하고 있다.

<form method="post">

<input type="hidden" name="parentId" value="0"/>

제목 : <input type="text" name="title"/> <br/>

내용 : <textarea name="content"></textarea> <br/>

<input type="submit"/>

</form>

||

||

|| 입력 항목의 이름과 일치하는

|| 프로퍼티에 값이 저장

||

||

public class NewArticleCommand {

private String title;

private String content;

private int parentId;

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

public int getParentId() {

return parentId;

}

public void setParentId(int parentId) {

this.parentId = parentId;

}

@Override

public String toString() {

return "NewArticleCommand [content=" + content + ", parentId="

+ parentId + ", title=" + title + "]";

}

}

\*\*\*HTML 폼과 Java빈 Class 사이의 Mapping

HTML 폼에 입력한 데이터를 Java빈 객체로 전달받는 방법은 매우 간단하다. 단지 @RequestMapping Annotation이 적용된

메서드의 파라미터로 Java빈 타입을 추가해주기만 하면 된다. 예를 들어, 폼에 입력한 값을 NewArticleController Class로

전달 받고 싶다면 다음과 같이 요청 처리 메서드에 NewArticleCommand 타입의 파라미터를 추가해주기만 하면 된다.

@Controller

@RequestMapping("/article/newArticle.do")

public class NewArticleController {

@RequestMapping(method = RequestMethod.POST)

public String submit(NewArticleCommand command) {

//command.getTitle() : title 파라미터 값 저장

//command.getContent() : content 파라미터 값 저장

//command.getParentId() : parentId 파라미터 값 저장

return "article/newArticleSubmitted";

}

...

}

(1) 뷰에서 커맨드 객체 접근하기

: 뷰 Code에서는 컨트롤러의 @RequestMapping Annotation 메서드에서 전달받은 커맨드 객체에 접근할 수 있다. 예를 들어,

아래와 같이 NewArticleCommand 타입의 커맨드 객체를 전달받는다고 하자.

@RequestMapping(method = RequestMethod.POST)

public String submit(NewArticleCommand command) {

...

}

이 경우 컨트롤러의 처리 결과를 보여주는 뷰 Code에서는 커맨드 객체의 Class 이름을 이용해서 커맨드 객체에 접근할 수

있다. 즉, 커맨드 객체는 자동으로 모델에 추가된다. (단, 첫 글자는 소문자이다.)

<body>

...

제목 : ${newArticleCommand.title}

...

뷰에서 사용할 모델의 이름을 변경하고 싶다면 다음과 같이 @ModelAttribute Annotation을 이용해서 커맨드 객체의 모델 이름을

지정할 수 있다.

import org.springframework.stereotype.Controller;

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

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

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

@Controller

@RequestMapping("/article/newArticle.do")

public class NewArticleController {

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") NewArticleCommand command) {

articleService.writeArticle(command);

return "article/newArticleSubmitted";

}

}

@ModelAttribute Annotation을 이용해서 커맨드 객체의 모델 이름을 지정했다면, 뷰 Code에서는 다음과 같이 해당 모델 이름을

사용해서 커맨드 객체에 접근할 수 있다.

제목 : ${command.title}

<Example001>

3[NewArticleController.java]

package madvirus.spring.chap06.controller;

import madvirus.spring.chap06.service.ArticleService;

import madvirus.spring.chap06.service.NewArticleCommand;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

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

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

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

@Controller

@RequestMapping("/article/newArticle.do")

public class NewArticleController {

@Autowired

private ArticleService articleService;

@RequestMapping(method = RequestMethod.GET)

public String form() {

return "article/newArticleForm";

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") NewArticleCommand command) {

articleService.writeArticle(command);

return "article/newArticleSubmitted";

}

public void setArticleService(ArticleService articleService) {

this.articleService = articleService;

}

}

1[NewArticleCommand.java]

package madvirus.spring.chap06.service;

public class NewArticleCommand {

private String title;

private String content;

private int parentId;

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

public String getContent() {

return content;

}

public void setContent(String content) {

this.content = content;

}

public int getParentId() {

return parentId;

}

public void setParentId(int parentId) {

this.parentId = parentId;

}

@Override

public String toString() {

return "NewArticleCommand [content=" + content + ", parentId="

+ parentId + ", title=" + title + "]";

}

}

2[ArticleService.java]

package madvirus.spring.chap06.service;

public class ArticleService {

public void writeArticle(NewArticleCommand command) {

System.out.println("신규 게시글 등록: " + command);

}

}

[/WEB-INF/dispatcher-servlet.xml]

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

...

<bean id="newArticleController" class="madvirus.spring.chap06.controller.NewArticleController"

p:articleService-ref="articleService" />

<bean id="articleService" class="madvirus.spring.chap06.service.ArticleService" />

...

[/WEB-INF/view/article/newArticleForm.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>게시글 쓰기</title>

</head>

<body>

게시글 쓰기 입력 폼:

<form method="post">

<input type="hidden" name="parentId" value="0" />

제목: <input type="text" name="title" /><br/>

내용: <textarea name="content"></textarea><br/>

<input type="submit" />

</form>

</body>

</html>

[[/WEB-INF/view/article/newArticleSubmitted.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>게시글 쓰기</title>

</head>

<body>

게시글 등록됨:

<br/>

제목: ${command.title}

</body>

</html>

http://localhost:8080/SpringMVC/article/newArticle.do

(2) 커맨드 객체로 List 받기

Command Object == JavaBean

: SPRING MVC는 List 타입의 프로퍼티에 대한 바인딩도 처리해 준다. 예를 들어, 아래 Code는 OrderItem 목록을 갖는 List 타입의

orderItem 프로퍼티를 갖고 있다.

public class OrderCommand {

private List<OrderItem> orderItems;

private Address address;

public List<OrderItem> getOrderItems() {

return orderItems;

}

public void setOrderItems(List<OrderItem> orderItems) {

this.orderItems = orderItems;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

}

List 타입의 프로퍼티에 값을 전달하고 싶다면 폼에서 "프로퍼티명[인덱스].프로퍼티" 같이 입력 폼의 이름을 구성하면 된다.

예를 들어, 위 Code의 orderItem 프로퍼티에 값을 전달하고 싶다면 아래 Code와 같이 폼을 구성하면 된다.

<form method="post">

상품1 : ID - <input type="text" name="orderItems[0].itemId"/>

개수 : - <input type="text" name="orderItems[0].number"/>

주의 : - <input type="text" name="orderItems[0].remark"/>

<br/>

상품2 : ID - <input type="text" name="orderItems[1].itemId"/>

개수 : - <input type="text" name="orderItems[1].number"/>

주의 : - <input type="text" name="orderItems[1].remark"/>

<br/>

...

<input type="submit"/>

</form>

폼의 <input>이나 <select>등의 name에 인덱스 값을 포함시키면 List 타입의 프로퍼티에 값을 전달 받을 수 있다. 컨트롤러

Code에서는 다음과 같이 커맨드 객체를 @RequestMapping 메서드에 지정해주기만 하면 된다.

@Controller

@RequestMapping("/order/order.do")

public class OrderController {

@RequestMapping(method = RequestMethod.POST)

public String submit(OrderCommand orderCommand) {

return "order/orderCompletion";

}

}

<Example002>

4[OrderController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

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

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

@Controller

@RequestMapping("/order/order.do")

public class OrderController {

@RequestMapping(method = RequestMethod.GET)

public String form() {

return "order/orderForm";

}

@RequestMapping(method = RequestMethod.POST)

public String submit(OrderCommand orderCommand) {

return "order/orderCompletion";

}

}

1[OrderItem.java]

package madvirus.spring.chap06.model;

public class OrderItem {

private Integer itemId;

private Integer number;

private String remark;

public Integer getItemId() {

return itemId;

}

public void setItemId(Integer itemId) {

this.itemId = itemId;

}

public Integer getNumber() {

return number;

}

public void setNumber(Integer number) {

this.number = number;

}

public String getRemark() {

return remark;

}

public void setRemark(String remark) {

this.remark = remark;

}

}

2[OrderCommand.java]

package madvirus.spring.chap06.controller;

import java.util.List;

import madvirus.spring.chap06.model.Address;

import madvirus.spring.chap06.model.OrderItem;

public class OrderCommand {

private List<OrderItem> orderItems;

private Address address;

public List<OrderItem> getOrderItems() {

return orderItems;

}

public void setOrderItems(List<OrderItem> orderItems) {

this.orderItems = orderItems;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

}

3[Address.java]

package madvirus.spring.chap06.model;

public class Address {

private String zipcode;

private String address1;

private String address2;

public String getZipcode() {

return zipcode;

}

public void setZipcode(String zipcode) {

this.zipcode = zipcode;

}

public String getAddress1() {

return address1;

}

public void setAddress1(String address1) {

this.address1 = address1;

}

public String getAddress2() {

return address2;

}

public void setAddress2(String address2) {

this.address2 = address2;

}

@Override

public String toString() {

return "Address [address1=" + address1 + ", address2=" + address2

+ ", zipcode=" + zipcode + "]";

}

}

[/WEB-INF/dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.OrderController" />

...

[/WEB-INF/view/order/orderCompletion.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>주문 완료</title>

</head>

<body>

주문 완료

<br/>

주문 아이템

<ul>

<c:forEach var="item" items="${orderCommand.orderItems}">

<li>${item.itemId} / ${item.number} / ${item.remark}</li>

</c:forEach>

</ul>

배송지: ${orderCommand.address}

</body>

</html>

[/WEB-INF/view/order/orderForm.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>주문</title>

</head>

<body>

<form method="post">

상품1: ID - <input type="text" name="orderItems[0].itemId" />

개수 - <input type="text" name="orderItems[0].number" />

주의 - <input type="text" name="orderItems[0].remark" />

<br/>

상품2: ID - <input type="text" name="orderItems[1].itemId" />

개수 - <input type="text" name="orderItems[1].number" />

주의 - <input type="text" name="orderItems[1].remark" />

<br/>

상품3: ID - <input type="text" name="orderItems[2].itemId" />

개수 - <input type="text" name="orderItems[2].number" />

주의 - <input type="text" name="orderItems[2].remark" />

<br/>

배송지:

우편번호 - <input type="text" name="address.zipcode" />

주소1 - <input type="text" name="address.address1" />

주소2 - <input type="text" name="address.address2" />

<br/>

<input type="submit" />

</form>

</body>

</html>

http://localhost:8080/SpringMVC/order/order.do

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.4 컨트롤러 메서드의 파라미터 타입

: 컨트롤러의 @RequestMapping Annotation이 적용된 메서드는 커맨드 Class뿐만 아니라 HttpServletRequest, HttpSession, Locale

등 WEB Application과 관련된 다양한 타입의 파라미터를 가질 수 있는데, 전달 가능한 파라미터 타입은 아래와 같다.

[[파라미터 타입]] [[설 명]]

===============================================================================

HttpServletRequest, HttpServletResponse, Servlet API

HttpSession

--------------------------------------------------------------------------------------------------------------------------------------------------------------

java.util.Locale 현재 요청에 대한 Locale

--------------------------------------------------------------------------------------------------------------------------------------------------------------

InputStream, Reader 요청 컨텐츠에 직접 접근할 때 사용

--------------------------------------------------------------------------------------------------------------------------------------------------------------

OutputStream, Writer 응답 컨텐츠를 생성할 때 사용

--------------------------------------------------------------------------------------------------------------------------------------------------------------

@PathVariable Annotation 적용 파라미터 URI 템플릿 변수에 접근할 때 사용

--------------------------------------------------------------------------------------------------------------------------------------------------------------

@RequestParam Annotation 적용 파라미터 HTTP 요청 파라미터를 Mapping

--------------------------------------------------------------------------------------------------------------------------------------------------------------

@RequestHeader Annotation 적용 파라미터 HTTP 요청 헤더를 Mapping

--------------------------------------------------------------------------------------------------------------------------------------------------------------

@CookieValue Annotation 적용 파라미터 HTTP 쿠키 Mapping

--------------------------------------------------------------------------------------------------------------------------------------------------------------

@RequestBody Annotation 적용 파라미터 HTTP 요청의 몸체 내용에 접근할 때 사용. HttpMessageConverter를 이용해서

HTTP 요청 데이터를 해당 타입으로 변환한다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

Map, Model, ModelMap 뷰에 전달할 모델 데이터를 설정할 때 사용

--------------------------------------------------------------------------------------------------------------------------------------------------------------

커맨드 객체 HTTP 요청 파라미터를 저장한 객체, 기본적으로 Class 이름을 모델명으로

사용. @ModelAttribute Annotation을 사용하여 모델명을 설정할 수 있다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

Errors, BindingResult HTTP 요철 파라미터를 커맨드 객체에 저장한 결과, 커맨드 객체를 위한

파라미터 바로 다음에 위치

--------------------------------------------------------------------------------------------------------------------------------------------------------------

SessionStatus 폼 처리를 완료 했음을 처리하기 위해 사용. @SessionAttribute Annotation을

명시한 session 속성을 제거하도록 이벤트를 발생시킨다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

Errors와 BindingResult는 연관된 커맨드 객체 바로 다음에 위치해야 한다. 이 두 타입을 제외한 나머지 타입의 파라미터는 순서에

상관 없이 위치할 수 있다.

(1) @RequestParam Annotation을 이용한 파라미터 Mapping

: 컨트롤러를 구현하면서 가장 많이 사용되는 Annotation이 바로 @RequestParam Annotation이다.

@RequestParam Annotation은 HTTP 요청 파라미터를 메서드의 파라미터로 전달받을 때 사용된다. @RequestParam Annotation과

HTTP 요청 파라미터의 관계는 아래 그림과 같다.

http://host/chap06/search/internal.do?query=spring&p=3

||

||

@Controller

public class SearchController {

@RequestMapping("/search/internal.do")

public ModelAndView searchInternal(

@RequestParam("query") String query,

@RequestParam("p") int pageNumber) {

...

}

}

\*\*\* @RequestParam Annotation을 통한 HTTP 요청 파라미터 Mapping

위의 Code에서 첫 번째 파라미터는 query 요청파라미터의 값을 전달받으며, 두 번째 파라미터인 pageNumber 파라미터는

p 파라미터의 값을 전달받는다.

@RequestParam Annotation이 적용된 파라미터가 String이 아닐 경우 실제 타입에 따라서 알맞은 타입 변환을 수행한다.

예를 들어, 위의 Code에서 pageNumber 파라미터의 타입은 int인데, 이 경우 자동으로 문자열을 int 타입으로 변환해준다.

만약 다음과 같이 int 타입으로 변환할 수 없는 값인 "a"를 pageNumber 파라미터에 매피되는 HTTP 요청 파라미터로

전달하면, SPRING MVC는 잘못된 요청(Bad Request)을 의미하는 400 응답 Code를 WEB 브라우저에 전송한다.

http://localhost:8080/chap06/search/internal.do?query=spring&p=a

@RequestParam Annotation이 적용된 파라미터는 기본적으로 필수 파라미터이다. 따라서 @RequestParam Annotation에

명시한 HTTP 요청 파라미터가 존재하지 않을 경우 SPRING MVC는 잘못된 요청(Bad Request)을 의미하는 400 응답 Code를

WEB 브라우저에 전송한다. 예를 들어, 위의 Code와 같이 @RequestParam Annotation의 병도 설정 을 하지 않은 경우, 다음의

URL을 요청하면 400 에러가 발생하게 된다.

http://localhost:8080/chap06/search/internal.do?query=spring

필수가 아닌 파라미터인 경우 required 속성 값을 false로 지정해 주면 된다. 아래 Code는 적용 예이다. 참고로, requred 속성의

기본 값은 true이다.

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

...

...

@RequestMapping("/search/external.do")

public ModelAndView searchExternal(

@RequestParam(value="query", required=false) String query,

@RequestParam(value = "p", required=false) int pageNumber) {

System.out.println("query=" + query + ",pageNumber=" + pageNumber);

...

}

필수가 아닌 요청 파라미터의 값이 존재하지 않을 경우 null 값을 할당한다. 그런데, null을 할당할 수 없는 기본 데이터 타입인

경우에는 타입 변환 에러가 발생하게 된다. 예를 들어, 위 Code에서 "p" 요청 파라미터를 필수가 아닌 파라미터로 설정했는데,

이 상태에서 "p" 요청 파라미터를 지정하지 않은 경우 null을 기본 데이터 타입으로 변환할 수 없다는 예외가 발생하게 된다.

기본 데이터 타입을 사용할 경우 HTTP 요청 파라미터가 존재하지 않으면 기본 값을 할당하는 경우가 많은데, 이런 경우에는

다음과 같이 defaultValue 속성을 이용해서 기본값을 지정할 수 있다.

@RequestMapping("/search/external.do")

public ModelAndView searchExternal(

@RequestParam(value="query", required=false) String query,

@RequestParam(value = "p", defaultValue = "1") int pageNumber) {

System.out.println("query=" + query + ",pageNumber=" + pageNumber);

...

}

defaultValue 속성을 이용해서 기본 값을 지정하면 해당 요청 파라미터를 지정하지 않을 경우 defaultValue 속성에 지정한 문자열을

값으로 이용하게 된다. 예를 들어, 위와 같이 "p" 요청 파라미터의 기본 값을 1로 지정했다면,

http://localhost:8080/chap06/search/internal.do?query=spring

"p" 파라미터가 존재하지 않으므로 기본 값으로 지정한 "1"을 "p"파라미터의 값으로 사용하게 되고, 따라서 pageNumber 파라미터의

값은 1이 된다.

<Example003>

[SearchController.java]

package madvirus.spring.chap06.controller;

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 SearchController {

@RequestMapping("/search/internal.do")

public ModelAndView searchInternal(@RequestParam("query") String query,

@RequestParam(value = "p", defaultValue = "1") int pageNumber) {

System.out.println("query=" + query + ",pageNumber=" + pageNumber);

return new ModelAndView("search/internal");

}

@RequestMapping("/search/external.do")

public ModelAndView searchExternal(@RequestParam(value="query", required=false) String query,

@RequestParam(value = "p", defaultValue = "1") int pageNumber) {

System.out.println("query=" + query + ",pageNumber=" + pageNumber);

return new ModelAndView("search/external");

}

}

[/WEB-INF/dispatcher-servlet.xml]

...

<bean id="searchController" class="madvirus.spring.chap06.controller.SearchController" />

...

[/WEB-INF/view/search/internal.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>내부 검색</title>

</head>

<body>

내부 검색

</body>

</html>

[/WEB-INF/view/search/external.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>외부 검색</title>

</head>

<body>

외부 검색

</body>

</html>

http://localhost:8080/SpringMVC01/search/internal.do?query=q&p=2

http://localhost:8080/SpringMVC01/search/internal.do?query=q

http://localhost:8080/SpringMVC01/search/internal.do

http://localhost:8080/SpringMVC01/search/external.do?query=p&p=22

http://localhost:8080/SpringMVC01/search/external.do?query=p

http://localhost:8080/SpringMVC01/search/external.do

(2) @CookieValue Annotation을 이용한 쿠키 Mapping

: @CookieValue Annotation을 이용하면 쿠키 값을 파라미터로 전달받을 수 있다. 다음은 "auth" 쿠키의 값을 authValue 파라미터를

통해서 전달받도록 작성한 Code의 예를 보여주고 있다.

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

...

...

@RequestMapping("/cookie/view.do")

public String view(@CookieValue(value = "auth") String authValue) {

...

return "cookie/view";

}

@CookieValue Annotation은 해당 쿠키가 존재하지 않으면 기본적으로 500 에러를 발생시킨다. 따라서, 쿠키가 필수가 아닌 경우에는

다음과 같이 required 속성의 값을 false로 지정해 주어야 한다. 참고로, required 속성의 기본 값은 true이다.

@RequestMapping("/cookie/view.do")

public String view(@CookieValue(value = "auth", required=false) String auth) {

...

return "cookie/view";

}

required 속성의 값을 false로 지정할 경우, 해당 쿠키가 존재하지 않으면 null을 값으로 전달받게 된다.

@RequestParam Annotation과 마찬가지로 defalutValue 속성을 이용해서 기본 값을 설정할 수도 있다. 아래 Code는 "auth" 쿠키가

존재하지 않을 경우 auth 파라미터는 "0"을 값으로 갖게 된다.

@RequestMapping("/cookie/view.do")

public String view(

@CookieValue(value = "auth", defaultValue = "0") String auth) {

...

return "cookie/view";

}

<Example004>

[CookieController.java]

package madvirus.spring.chap06.controller;

import javax.servlet.http.Cookie;

import javax.servlet.http.HttpServletResponse;

import org.springframework.stereotype.Controller;

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

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

@Controller

public class CookieController {

@RequestMapping("/cookie/make.do")

public String make(HttpServletResponse response) {

response.addCookie(new Cookie("auth", "1"));

return "cookie/made";

}

@RequestMapping("/cookie/view.do")

public String view(

@CookieValue(value = "auth", defaultValue = "0") String auth) {

System.out.println("auth 쿠키: " + auth);

return "cookie/view";

}

}

[/WEB-INF/dispatcher-servlet.xml]

...

<bean id="cookieController" class="madvirus.spring.chap06.controller.CookieController" />

...

[/WEB-INF/view/cookie/made.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>쿠키</title>

</head>

<body>

쿠키 생성함

</body>

</html>

[/WEB-INF/view/cookie/view.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>쿠키</title>

</head>

<body>

쿠키 확인

</body>

</html>

http://localhost:8080/SpringMVC01/cookie/make.do

http://localhost:8080/SpringMVC01/cookie/view.do

(3) @RequestHeader Annotation을 이용한 헤더 Mapping

: @RequestHeader Annotation을 이용하면 HTTP 요청 헤더의 값을 메서드의 파라미터로 전달 받을 수 있다.

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

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

@Controller

public class HeaderController {

@RequestMapping("/header/check.do")

public String check(@RequestHeader("Accept-Language") String languageHeader) {

System.out.println(languageHeader);

return "header/pass";

}

}

@RequestHeader Annotation도 @RequestCookie Annotation과 마찬가지로 해당 해더가 존재하지 않으면 500

응답 에러 Code를 전송한다. 또한, required 속성과 defalutValue 속성을 이용해서 필수 여부와 기본 값을 설정할 수 있다.

<Example005>

[HeaderController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

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

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

@Controller

public class HeaderController {

@RequestMapping("/header/check.do")

public String check(@RequestHeader("Accept-Language") String languageHeader) {

System.out.println(languageHeader);

return "header/pass";

}

}

[/WEB-INF/dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.HeaderController" />

...

[/WEB-INF/view/header/pass.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>헤더 검사 통과</title>

</head>

<body>

헤더 검사 통과됨

</body>

</html>

http://localhost:8080/SpringMVC01/header/check.do

(4) Servlet API 직접 사용

: 컨트롤러 Class의 @RequestMapping Annotation이 적용 메서드는 다음의 다섯 가지 타입의 파라미터를 전달받을 수 있다.

- javax.servlet.http.HttpServletRequest / javax.servlet.ServletRequest

- javax.servlet.http.HttpServletResponse / javax.servlet.ServletResponse

- javax.servlet.http.HttpSession

앞에서 살펴봤듯이 Servlet API를 사용할 필요 없이 SPRING MVC가 제공하는 Annotation을 이용해서 요청 파라미터, 헤더,

쿠키, 세션 등의 정보에 접근할 수 있기때문에, 직접적으로 Servlet API를 사용해야 하는 경우는 매우 드물다. 하지만 다음의

경우에는 Servlet API를 사용하는 것이 더 편리할 수 있다.

- HttpSession의 생성을 직접 제어해야 하는 경우

- 컨트롤러에서 쿠키를 생성해야 하는 경우

- Servlet API 사용을 선호하는 경우

예를 들어, 조건에 따라서 HttpSession을 생성해야 하는 경우를 생각해 보자. 이 때에는 다음과 같이 HttpServletRequest 타입의

파라미터를 전달받아야만 HttpSession을 조건에 따라 생성하거나 생성하지 않을 수 있다.

@RequestMapping("/someUrl")

public ModelAndView process(HttpServletRequest request, ...){

if(someCondition){

HttpSession session = request.getSession();

}

...

}

HttpSession 타입의 파라미터를 가질 경우 세션이 생성된다는 점에 유의해야 한다. 즉, 기존에 세션이 존재한다면 해당 세션이

전달되고 그렇지 않다면 새로운 세션이 생성되고 관련 HttpSession 인스턴스가 파라미터에 전달된다. 따라서, HttpSession

타입의 파라미터는 항상 null이 아니다.

@RequestMapping("/someUrl")

public ModelAndView process(HttpSession session, ...){

session != null; //항상 true

...

}

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.5 컨트롤러 메서드의 리턴 타입

: 컨트롤러 메서드는 ModelAndView를 비롯한 몇 가지 리턴 타입을 가질 수 있으며, 이들 리턴 타입은 다음과 같다.

[[리턴 타입]] [[설 명]]

================================================================================

ModelAndView 뷰 정보 및 모델 정보를 담고 있는 ModelAndView 객체.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

Model 뷰에 전달할 객체 정보를 담고 있는 Model을 리턴한다. 이때 뷰 이름은 요청 URL로부터 경정된다.

(RequestToViewNameTranslator를 통해 뷰 결정)

----------------------------------------------------------------------------------------------------------------------------------------------------------------

Map 뷰에 전달할 객체 정보를 담고 있는 Map을 리턴한다. 이때 뷰 이름은 요청 URL로부터 경정된다.

(RequestToViewNameTranslator를 통해 뷰 결정)

----------------------------------------------------------------------------------------------------------------------------------------------------------------

String 뷰 이름을 리턴한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

View 객체 View 객체를 직접 리턴. 해당 View 객체를 이용해서 뷰를 생성한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

void 메서드가 ServletResponse나 HttpServletResponse 타입의 파라미터를 갖는 경우 메서드가 직접

응답을 처리한다고 가정한다. 그렇지 않을 경우 요청 URL로부터 결정된 뷰를 보여준다.

(RequestToViewNameTranslator를 통해 뷰 결정)

----------------------------------------------------------------------------------------------------------------------------------------------------------------

@RepositoryBody 메서드에서 @RepositoryBody Annotation이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다.

Annotation 적용 HttpMessageConverter를 이용해서 객체를 HTTP 응답 스트림으로 변환한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

ModelAndView, Model, Map을 이용해서 모델을 설정하는 방법은 본 장의 '모델 생성하기' 절에서 자세히 살펴볼 것이다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

5.6 컨트롤러 Class 자동 스캔

: @Controller Annotation은 @Component Annotation과 마찬가지로 컴포넌트 스캔 대상이다. 따라서, <context:component-scan>

태그를 이용해서 @Controller Annotation이 적용된 컨트롤러 Class를 자동으로 로딩할 수 있다.

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="madvirus.spring.chap06.controller" />

</beans>

===============================================================================

6. 뷰 지정

: 컨트롤러 처리 메서드는 처리 결과를 보여줄 뷰 이름이나 View 객체를 리턴하고, DispatcherServlet은 뷰 이름이나 View 객체를

이용해서 뷰를 생성하게 된다. 뷰 이름은 직접 또는 자동으로 설정될 수 있는데, 본 절에서는 컨트롤러에서 뷰를 어떻게 지정하는 지

살펴보고 또한 리다이렉트를 지정하는 방법을 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

6.1 뷰 이름 명시적 지정 : ModelAndView와 String 리턴 타입

: 뷰 이름을 명시적으로 지정하려면 ModelAndView나 String을 리턴해야 한다. 아래 Code는 ModelAndView를 리턴해서 뷰를 지정하는

예를 보여주고 있다.

@RequestMapping("/index.do")

public ModelAndView index(){

ModelAndView mav = new ModelAndView("index");

...

return mav;

}

ModelAndView를 리턴할 경우 ModelAndView Class의 생성자나 setViewName() 메서드를 이용해서 뷰 이름을 지정할 수 있다.

위 Code는 ModelAndView Class의 생성자를 이용해서 뷰 이름을 지정하고 있다. 다음과 같이 setViewName() 메서드를 사용해도 된다.

ModelAndView mav = new ModelAndView("index");

mav.setViewName("search/game");

String 타입을 리턴할 경우, 문자열 값이 뷰 이름으로 사용된다. 아래 Code는 String을 리턴하는 예를 보여주고 있다. 아래 Code는 결과를

보여줄 뷰 이름으로 "help/main"을 지정하고 있다.

@RequestMapping("/help/main.do")

public String helpMain(ModelMap model){

...

return "help/main";

}

---------------------------------------------------------------------------------------------------------------------------------------------------------------

6.2 뷰 이름 자동 지정

: 다음의 경우는 RequestToViewNameTranslator를 이용해서 URL로부터 뷰 이름을 결정한다.

- 리턴 타입이 Model이나 Map인 경우

- 리턴 타입이 void이면서 ServletResponse나 HttpServletResponse 타입의 파라미터가 없는 경우

SPRING 설정 파일에 RequestToViewNameTranslator 빈이 존재하지 않을 경우 기본적으로 DefaultRequestToViewNameTranslator를

사용한다. 이 Class는 요청 URI로부터 맨 앞의 슬래시와 확장자를 제외한 나머지 부분을 뷰 이름으로 사용한다. (더 정확하게는

전체 경로를 사용하지 않도록 설정한 경우 Servlet 경로를 제외한 나머지 경로가 사용된다.) 예를 들어, 아래 Code를 보자

@RequestMapping("/search/game2.do")

public Map<String, Object) search(){

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

...

return model;

}

위 Code는 뷰에 전달할 모델 데이터를 갖고 있는 Map을 리턴하고 있다. 이 경우 RequestToViewNameTranslator를 이용해서 결과를

보여줄 뷰 이름을 결정하게 되는데, DefaultRequestToViewNameTranslator가 사용될 경우 다음과 같이 URL로부터 뷰 이름이 결정된다.

/search/game2.do ===> search/game2

전체 경로 사용 여부에 따라서 뷰 이름이 결정되는 방식이 달라지는데 이에 대한 내용은 '요청 URI 매칭'절에서 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

6.3 리다이렉트 뷰

: 뷰 이름에 "redirect:" 접두어를 붙이면, 지정한 페이지로 리다이렉트 된다. 리다이렉트 URL은 다음과 같이 두 가지 방식으로 입력할 수 있다.

- redirect:/bbs/list - 현재 Servlet 컨텍스트에 대한 상대적인 경로로 리다이렉트

- redirect:http://host/bbs/list - 지정한 절대 URL로 리다이렉트

아래 Code는 "redirect:" 접두어를 사용하여 뷰 이름을 지정한 예이다.

ModelAndView mav = new ModelAndView();

mav.setViewName("redirect:/error.do");

return mav;

===============================================================================

7. 모델 생성하기

: @RequestMapping Annotation이 적용된 메서드의 파라미터나 리턴타입으로 ModelAndView, Model, Map, 커맨드 객체 등을 이용해서

모델을 뷰에 전달하게 된다. 본 절에서는 컨트롤러가 이들 Class를 이용해서 모델을 어떻게 뷰에 전달하는지 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

7.1 뷰에 전달되는 모델 데이터

: @RequestMapping 메서드가 ModelAndView, Model, Map을 리턴하는 경우 이들에 담긴 모델 데이터가 뷰에 전달된다. 또한, 추가로 다음의

항목도 뷰에 함께 모델로 전달된다.

- 커맨드 객체(JavaBean)

- @ModelAttribute Annotation이 적용된 메서드가 리턴한 객체

- 메서드의 Map, Model, ModelMap 타입의 파라미터를 통해 설정된 모델

아래 Code는 다양한 방법으로 모델 데이터를 설정하고 있다.

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

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

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class GameSearchController {

@ModelAttribute("searchTypeList")

public List<SearchType> referenceSearchTypeList() {

List<SearchType> options = new ArrayList<SearchType>();

options.add(new SearchType(1, "전체"));

options.add(new SearchType(2, "아이템"));

options.add(new SearchType(3, "캐릭터"));

return options;

}

@RequestMapping("/search/game.do")

public ModelAndView search(@ModelAttribute("command") SearchCommand command) {

String[] queryList = getPopularQueryList();

model.addAttribute("popularQueryList", queryList);

ModelAndView mav = new ModelAndView("search/game");

SearchResult result = searchService.search(command);

mav.addObject("searchResult", result);

return mav;

}

}

이 Code에서 @ModelAttribute Annotation이 적용된 referenceSearchTypeList() 메서드가 리턴한 모델 객체는

'searchTypeList'라는 이름으로 뷰에 전달된다.

@RequestMapping Annotation이 적용된 search() 메서드는 파라미터로 커맨드 객체와 ModelMap 객체를 전달받는다.

search() 메서드 내부에서는 ModelMap 타입 파라미터인 model을 이용해서 'popularQueryList' 이름의 모델 객체를 뷰에

전달한다. 또한, 내부적으로 ModelAndView 타입의 mav 객체를 만든 뒤 mav를 이용해서 'searchResult'라는 이름의

모델 객체를 뷰에 전달한다.

이들 모델 객체는 모두 서로 다른 방법으로 생성되었지만, DispatcherServlet은 이들 모델 객체에 저장된 모델 데이터를

모두 뷰에 전달한다. 따라서 뷰 Code에서는 다음과 같이 이들 모델 객체를 모두 사용할 수 있게 된다.

<body>

인기 키워드 :

<c:forEach var="popularQuery" items="${popularQueryList}">

${popularQuery}

</c:forEach>

<form action="game.do">

<select name="type">

<c:forEach var="searchType" items= ">

<option value="${searchType.code}"

<c:if test="${command.type==searchType.code}">

selected

</c:if>

${searchType.text}

</option>

</c:forEach>

</select>

<input type="text" name="query" value="${command.query}"/>

<input type="submit" value="검색"/>

</form>

검색 결과 : ${searchResult}

</body>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

7.2 Map, Model, ModelMap을 통한 모델 설정

: Map, Model, ModelMap을 이용하면 뷰에 전달할 모델을 생성할 수 있다. 첫 번째 방법은 이들 세 가지 타입 중

한 가지를 파라미터로 전달 받는 것이다.

@RequestMapping("/search1.do")

public String search1(Map model){

model.put("result", searchResult);

...

}

@RequestMapping("/search2.do")

public String search1(Model model){

model.put("result", searchResult);

...

}

@RequestMapping("/search3.do")

public String search1(ModelMap model){

model.put("result", searchResult);

...

}

Map, Model, ModelMap을 파라미터로 전달받는 경우, 이들이 제공하는 메서드를 이용해서 모델 데이터를 추가할 수 있다.

두 번째 방법은 Map과 Model을 리턴하는 것이다. 이 둘은 모두 Interface이기 때문에 실제로는 Interface를 구현한

Class의 객체를 생성해서 리턴하면 된다. 아래 Code는 사용 예를 보여주고 있다.

@RequestMapping("/search1.do")

public Map<String, Object> search1(Map model){

...

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

model.put("result", searchResult);

return model;

}

@RequestMapping("/search2.do")

public Model search1(Model model){

...

Model model = new ExtendedModelMap();

model.put("result", searchResult);

return model;

}

SPRING은 org.springframework.ui.Model Interface의 구현 Class인 ExtendedModelMap Class를 동일한 패키지에 제공하고

있으므로, ExtendedModelMap Class를 이용해서 모델을 설정하면 된다. Map의 경우 많이 사용하는 java.util.HashMap Class를

이용해서 모델을 설정한다.

(1) Model Interface의 주요 메서드

: org.springframework.ui.Model Interface는 모델을 설정할 수 있도록 다음과 같은 메서드를 제공하고 있다.

- Model addAttrubute(String name, Object value)

: value 객체를 name 이름으로 추가한다. 뷰 Code에서는 name으로 지정한 이름을 통해서 value를 사용한다.

- Model addAttrubute(Object value)

: value를 추가한다. value의 패키지 이름을 제외한 단순 Class 이름을 모델 이름으로 사용한다. 이 때 첫 글자는 소문자로 처리한다.

value가 배열이거나 컬렉션인 경우 첫 번째 원소의 Class 이름 뒤에 "List"를 붙인 걸 모델 이름으로 사용한다. 이 경우에도 Class

이름의 첫 자는 소문자로 처리한다.

- Model addAllAttrubutes(Collection<?> values)

: addAttribute(Object value) 메서드를 이용해서 컬렉션에 포함된 객체들을 차례대로 추가한다.

- Model addAllAttrubutes(Map<String,?> attributes)

: Map에 포함된 <키,값>에 대해 키를 모델 이름으로 사용해서 값을 모델로 추가한다.

- Model mergeAttrubutes(Map<String,?> attributes)

: Map에 포함된 <키,값>을 현재 모델에 추가한다. 단, 키와 동일한 이름을 갖는 모델 객체가 존재하지 않는 경우에만 추가된다.

- boolean containsAttribute(String name)

: 지정한 이름의 모델 객체를 포함하고 있는 경우 true를 리턴한다.

다음 Code는 컨트롤러 요청 처리 메서드에서 Model 타입의 파라미터를 이용해서 모델을 설정하는 예를 보여주고 있다.

@RequestMapping("/search/game.do")

public String search(SearchCommand command, Model model){

...

model.addAttribute("searchResult", result);

model.addAttribute("searchTypeList", searchTypeList);

...

}

컨트롤러 요청 처리 메서드에서 Model 타입을 리턴하고 싶다면 Model Interface의 구현 Class인

org.springframework.ui.ExtendedModelMap Class를 사용하면 된다.

@RequestMapping("/search/game.do")

public Model search(SearchCommand command){

...

Model model = new ExtendedModelMap();

model.addAttribute("searchResult", result);

model.addAttribute("searchTypeList", searchTypeList);

return model;

}

(2) ModelMap Class

: org.springframework.ui.ModelMap Class는 구현 Class로서 제공하는, 컨트롤러 메서드의 파라미터를 통해서

전달받는다.

@RequestMapping("/search/game.do")

public String search(SearchCommand command, ModelMap model){

...

model.addAttribute("result", searchResult);

...

}

ModelMap Class가 제공하는 모델 설정 관련 메서드는 Model Interface와 동일하므로 추가 설명은 하지 않겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

7.3 ModelAndView를 통한 모델 설정

: org.springframework.web.servlet.ModelAndView Class는 컨트롤러의 처리 결과를 보여 줄 뷰와 뷰에 전달할 값을 저장하는

용도로 사용되는데, 본 절에서는 ModelAndView 객체를 생성하는 방법을 살펴보도록 하겠다.

ModelAndView 객체를 생성하는 방법은 다음과 같이 setViewName(String viewName) 메서드를 이용하여 뷰 이름을 설정하고,

addObject(String name, Object value) 메서드를 이용하여 뷰에 전달할 값을 추가하는 것이다.

@RequestMapping("/search/game.do")

public ModelAndView search(SearchCommand command){

...

ModelAndView mav = new ModelAndView();

mav.setViewName("search/game");

mav.addObject("searchResult", searchResult);

mav.addObject("searchTypeList", typeList);

return mav;

}

Map에 저장된 <키,값> 쌍 전체를 뷰에 전달할 값으로 ModelAndView 객체에 추가하고 싶다면 다음과 같이 addAllObjects(Map modelMap)

메서드를 사용하면 된다.

Map referenceMap = referenceData();

mav.addAllObjects(referenceMap);

생성자를 사용해서 뷰 이름과 Map을 전달할 수도 있다.

Map referenceMap = referenceData();

return new ModelAndView("search/game",referenceMap);

뷰에 전달할 객체가 한 개뿐이라면 다음과 같은 생성자를 이용해서 Code 분량을 줄일 수 있을 것이다.

return new ModelAndView("search/game","result", searchResult);

---------------------------------------------------------------------------------------------------------------------------------------------------------------

7.4 @ModelAttribute Annotation을 이용한 모델 데이터 처리

: @ModelAttribute Annotation을 이용하면 다음의 두 가지 작업을 수행할 수 있다.

- @ModelAttribute Annotation이 적용되지 않은 별도 메서드로 모델에 추가될 객체를 생성

- 커맨드 객체의 초기화 작업을 수행

본 절에서는 @ModelAttribute Annotation의 두 가지 용법을 살펴보도록 하자.

(1) 참조 데이터 생성

: WEB Application을 구현하다 보면 동일한 모델 데이터를 두 개 이상의 요청 처리 결과 화면에서 보여주어야 할 때가 있다.

예를 들어, 검색 메인 화면과 검색 결과 화면에서 검색 타입과 인기 검색어를 보여줄 수 있을 것이다. 이 경우 다음과 같이

이들 공통 모델 데이터를 설정해 주는 메서드를 구현한 뒤 요청 처리 메서드에서 호출하도록 구현할 수 있을 것이다.

private void referenceData(Model model){

...

model.addAttribute("searchTypeList", searchTypeList);

model.addAttribute("popularQueryList", queryList);

}

@RequestMapping("/search/main.do")

public String main(Model model) {

referenceData(model);

return "search/main";

}

@RequestMapping("/search/game.do")

public ModelAndView search(@ModelAttribute("command") SearchCommand command, Model model) {

ModelAndView mav = new ModelAndView("search/game");

SearchResult result = searchService.search(command);

mav.addObject("searchResult", result);

return mav;

}

하지만, 위와 같이 구현하려면 다음과 같은 단점이 발생한다.

- Model이나 ModelMap과 같이 모델 정보를 설정할 때 사용할 타입이 동일해야 한다.

- 각 요청 처리 메서드를 공통으로 사용되는 모델을 설정하기 위한 Code가 중복된다.

@ModelAttribute Annotation을 사용하면 이런 단점 없이 두 개 이상의 요청 처리 메서드에서 공통으로 사용되는 모델을

설정할 수가 있다. @ModelAttribute Annotation을 메서드에 적용하면 해당 메서드가 생성한 객체가 뷰에 전달된다. 따라서,

@ModelAttribute Annotation을 사용하면 두 개 이상의 요청 처리 메서드에서 공통으로 사용되는 모델 객체를 생성할 수 있다.

import org.springframework.stereotype.Controller;

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

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class GameSearchController {

@Autowired

private SearchService searchService;

@ModelAttribute("searchTypeList")

public List<SearchType> referenceSearchTypeList() {

List<SearchType> options = new ArrayList<SearchType>();

options.add(new SearchType(1, "전체"));

options.add(new SearchType(2, "아이템"));

options.add(new SearchType(3, "캐릭터"));

return options;

}

@ModelAttribute("popularQueryList")

public String[] getPopularQueryList() {

return new String[] { "게임", "창천2", "위메이드" };

}

@RequestMapping("/search/main.do")

public String main() {

return "search/main";

}

@RequestMapping("/search/game.do")

public ModelAndView search(@ModelAttribute("command") SearchCommand command) {

ModelAndView mav = new ModelAndView("search/game");

SearchResult result = searchService.search(command);

mav.addObject("searchResult", result);

return mav;

}

}

위 Code에서 main() 메서드와 search() 메서드는 둘 다 referenceSearchTypeList() 메서드와 getPopularQueryList() 메서드를

호출하고 있지 않다. 하지만, referenceSearchTypeList() 메서드와 getPopularQueryList() 메서드에는 @ModelAttribute Annotation이

적용되어 있으므로 main() 메서드와 search() 메서드가 실핼될 때 referenceSearchTypeList() 메서드와 getPopularQueryList() 메서드가

리턴한 객체가 모델로 뷰에 함께 전달된다. 이때 @ModelAttribute Annotation의 값이 뷰에 전달되는 객체의 이름으로 사용되므로,

뷰 Code에서는 다음과 같이 @ModelAttribute Annotation의 값을 이용해서 모델 객체에 접근할 수 있게 된다.

<body>

인기 키워드 :

<c:forEach var="popularQuery" items="${popularQueryList}">

${popularQuery}

</c:forEach>

<form action="game.do">

<select name="type">

<c:forEach var="searchType" items="{searchTypeList}">

<option value="${searchType.code}">

${searchType.text}

</option>

</c:forEach>

</select>

<input type="text" name="query" value=""/>

<input type="submit" value="검색"/>

</form>

검색 결과 : ${searchResult}

</body>

<Example006>

5[GameSearchController.java]

package madvirus.spring.chap06.controller;

import java.util.ArrayList;

import java.util.List;

import madvirus.spring.chap06.service.SearchCommand;

import madvirus.spring.chap06.service.SearchResult;

import madvirus.spring.chap06.service.SearchService;

import madvirus.spring.chap06.service.SearchType;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

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

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

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class GameSearchController {

@Autowired

private SearchService searchService;

@ModelAttribute("searchTypeList")

public List<SearchType> referenceSearchTypeList() {

List<SearchType> options = new ArrayList<SearchType>();

options.add(new SearchType(1, "전체"));

options.add(new SearchType(2, "아이템"));

options.add(new SearchType(3, "캐릭터"));

return options;

}

@ModelAttribute("popularQueryList")

public String[] getPopularQueryList() {

return new String[] { "게임", "창천2", "위메이드" };

}

@RequestMapping("/search/main.do")

public String main() {

return "search/main";

}

@RequestMapping("/search/game.do")

public ModelAndView search(@ModelAttribute("command") SearchCommand command) {

ModelAndView mav = new ModelAndView("search/game");

System.out.println("검색어 = " + command.getQuery().toUpperCase());

SearchResult result = searchService.search(command);

mav.addObject("searchResult", result);

return mav;

}

@ExceptionHandler(NullPointerException.class)

public String handleNullPointerException(NullPointerException ex) {

return "error/nullException";

}

public void setSearchService(SearchService searchService) {

this.searchService = searchService;

}

}

1[SearchCommand.java]

package madvirus.spring.chap06.service;

public class SearchCommand {

private String type;

private String query;

private int page;

public String getType() {

return type;

}

public void setType(String type) {

this.type = type;

}

public String getQuery() {

return query;

}

public void setQuery(String query) {

this.query = query;

}

public int getPage() {

return page;

}

public void setPage(int page) {

this.page = page;

}

}

2[SearchType.java]

package madvirus.spring.chap06.service;

public class SearchType {

private int code;

private String text;

public SearchType(int code, String text) {

this.code = code;

this.text = text;

}

public int getCode() {

return code;

}

public void setCode(int code) {

this.code = code;

}

public String getText() {

return text;

}

public void setText(String text) {

this.text = text;

}

}

3[SearchService.java]

package madvirus.spring.chap06.service;

public class SearchService {

public SearchResult search(SearchCommand command) {

return new SearchResult();

}

}

4[SearchResult.java]

package madvirus.spring.chap06.service;

public class SearchResult {

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.GameSearchController"

p:searchService-ref="searchService" />

<bean id="searchService" class="madvirus.spring.chap06.service.SearchService" />

...

[/WEB-INF/view/search/main.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>게임 검색 메인</title>

</head>

<body>

인기 키워드: <c:forEach var="popularQuery" items="${popularQueryList}">${popularQuery} </c:forEach>

<form action="game.do">

<select name="type">

<c:forEach var="searchType" items="${searchTypeList}">

<option value="${searchType.code}">${searchType.text}</option>

</c:forEach>

</select>

<input type="text" name="query" value=""/>

<input type="submit" value="검색" />

</form>

</body>

</html>

[/WEB-INF/view/search/game.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>게임 검색 결과</title>

</head>

<body>

인기 키워드: <c:forEach var="popularQuery" items="${popularQueryList}">${popularQuery} </c:forEach>

<form action="game.do">

<select name="type">

<c:forEach var="searchType" items="${searchTypeList}">

<option value="${searchType.code}" <c:if test="${command.type == searchType.code}">selected</c:if>>

${searchType.text}</option>

</c:forEach>

</select>

<input type="text" name="query" value="${command.query}"/>

<input type="submit" value="검색" />

</form>

검색 결과: ${searchResult}

</body>

</html>

http://localhost:8080/SpringMVC01/search/main.do

(2) 커맨드 객체 초기화

: @ModelAttribute Annotation을 사용하면 커맨드 객체의 초기화 작업을 수행할 수도 있다. 예를 들어, GET 요청시에는

폼의 특정 값을 미리 초기화하고 POST 요청시에는 폼에 입력한 값을 커맨드 객체로 받아야 한다고 해 보자. 이런 경우

@ModelAttribute Annotation을 사용하면 다음과 같이 GET 요청과 POST 요청에 대해 알맞게 커맨드 객체를 초기화 할 수

있다. 아래 Code는 @ModelAttribute Annotation의 사용 예를 보여주고 있다.

@Controller

@RequestMapping("/account/create.do")

public class CreateAccountController {

@ModelAttribute("command")

public MemberInfo formBacking(HttpServletRequest request) {

if (request.getMethod().equalsIgnoreCase("GET")) {

MemberInfo mi = new MemberInfo();

Address address = new Address();

address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));

mi.setAddress(address);

return mi;

} else {

return new MemberInfo();

}

}

@RequestMapping(method = RequestMethod.GET)

public String form() {

return "account/creationForm";

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") MemberInfo memberInfo) {

return "account/created";

}

...

}

위 Code에서 주목할 점은 formBacking() 메서드의 @ModelAttribute Annotation 값과 submit() 메서드의 파라미터에서

사용된 @ModelAttribute Annotation의 값이 'command'로 동일하다는 것이다. submit() 메서드는 POST 요청을 처리하도록

되어 있는데, POST 요청이 전송될 경우 formBacking() 메서드가 생성한 MemberInfo 객체는 'command'라는 이름의 모델

객체로 저장되며, 이렇게 생성된 'command' MemberInfo 객체는 submit() 메서드의 동일한 모델 이름을 갖는 커맨드 객체

(첫 번째 파라미터)로 전달된다.

@ModelAttribute Annotation이 적용된 메서드가 @RequestMapping Annotation이 적용된 메서드 보다 먼저 호출되기 때문에,

커맨드 객체에 대한 초기화 작업이 필요하다면 객체와 동일한 이름을 갖는 @ModelAttribute Annotation이 적용된 메서드를

이용하면 된다.

(3) @ModelAttribute Annotation 적용 메서드에 전달 가능한 파라미터 타입

: @ModelAttribute Annotation이 적용된 메서드는 @RequestMapping Annotation이 적용된 메서드와 동일한 타입의 파라미터를

가질 수 있다. 앞서 Code에서는 HTTP 메서드를 확인하기 위해 HttpServletRequest를 파라미터로 전달받았다.

@ModelAttribute("command")

public MemberInfo formBacking(HttpServletRequest request) {

if (request.getMethod().equalsIgnoreCase("GET")) {

...

} else {

return new MemberInfo();

}

}

@ModelAttribute Annotation이 적용 메서드는 HttpServletRequest 뿐만 아니라 필요에 따라 Locale, @RequestParam Annotation

적용 파라미터, @PathVariable Annotation 적용 파라미터 등을 이용해서 모델 객체를 생성하는 데 필요한 정보를 구할 수 있다.

<Example007>

4[CreateAccountController.java]

package madvirus.spring.chap06.controller;

import javax.servlet.http.HttpServletRequest;

import madvirus.spring.chap06.model.Address;

import madvirus.spring.chap06.model.MemberInfo;

import madvirus.spring.chap06.validator.MemberInfoValidator;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

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

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

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

@Controller

@RequestMapping("/account/create.do")

public class CreateAccountController {

@ModelAttribute("command")

public MemberInfo formBacking(HttpServletRequest request) {

if (request.getMethod().equalsIgnoreCase("GET")) {

MemberInfo mi = new MemberInfo();

Address address = new Address();

address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));

mi.setAddress(address);

return mi;

} else {

return new MemberInfo();

}

}

private String autoDetectZipcode(String remoteAddr) {

return "000000";

}

@RequestMapping(method = RequestMethod.GET)

public String form() {

return "account/creationForm";

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") MemberInfo memberInfo,

BindingResult result) {

new MemberInfoValidator().validate(memberInfo, result);

if (result.hasErrors()) {

return "account/creationForm";

}

return "account/created";

}

}

1[MemberInfo.java]

package madvirus.spring.chap06.model;

public class MemberInfo {

private String id;

private String name;

private Address address;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}

}

5[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.CreateAccountController" />

...

<bean id="messageSource"

class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>messages.validation</value>

</list>

</property>

</bean>

[/WEB-INF/view/account/creationForm.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>계정 생성</title>

</head>

<body>

<spring:hasBindErrors name="command" />

<form:errors path="command" />

<form method="post">

아이디: <input type="text" name="id" value="${command.id}" />

<form:errors path="command.id" />

<br/>

이름: <input type="text" name="name" value="${command.name}" />

<form:errors path="command.name" />

<br/>

우편번호: <input type="text" name="address.zipcode" value="${command.address.zipcode}" />

<form:errors path="command.address.zipcode" />

<br/>

주소1: <input type="text" name="address.address1" value="${command.address.address1}" />

<form:errors path="command.address.address1" />

<br/>

주소2: <input type="text" name="address.address2" value="${command.address.address2}" />

<form:errors path="command.address.address2" />

<br/>

<input type="submit" />

</form>

</body>

</html>

[/WEB-INF/view/account/created.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>계정 생성</title>

</head>

<body>

계정 생성됨

<ul>

<li>아이디: ${command.id}</li>

<li>이름: ${command.name}</li>

<li>우편번호: ${command.address.zipcode}</li>

<li>주소: ${command.address.address1} ${command.address.address2}</li>

</ul>

</body>

</html>

2[/src/messages/validation.properties]

required=필수 항목입니다.

invalidIdOrPassword.loginCommand=아이디({0})와 암호가 일치하지 않습니다.

required.loginCommand.userId=사용자 아이디는 필수 항목입니다.

required.password=암호는 필수 항목입니다.

typeMismatch.from=시작일 값 형식은 yyyy-MM-dd 여야 합니다.

typeMismatch.to=종료일 값 형식은 yyyy-MM-dd 여야 합니다.

3[MemberInfoValidator.java]

package madvirus.spring.chap06.validator;

import madvirus.spring.chap06.model.Address;

import madvirus.spring.chap06.model.MemberInfo;

import org.springframework.validation.Errors;

import org.springframework.validation.Validator;

public class MemberInfoValidator implements Validator {

@Override

public boolean supports(Class<?> clazz) {

return MemberInfo.class.isAssignableFrom(clazz);

}

@Override

public void validate(Object target, Errors errors) {

MemberInfo memberInfo = (MemberInfo) target;

if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {

errors.rejectValue("id", "required");

}

if (memberInfo.getName() == null

|| memberInfo.getName().trim().isEmpty()) {

errors.rejectValue("name", "required");

}

Address address = memberInfo.getAddress();

if (address == null) {

errors.rejectValue("address", "required");

}

if (address != null) {

errors.pushNestedPath("address");

try {

if (address.getZipcode() == null

|| address.getZipcode().trim().isEmpty()) {

errors.rejectValue("zipcode", "required");

}

if (address.getAddress1() == null

|| address.getAddress1().trim().isEmpty()) {

errors.rejectValue("address1", "required");

}

} finally {

errors.popNestedPath();

}

}

}

}

===============================================================================

8. 요청 URI 매칭

: 클라이언트가 요청한 URL과 @RequestMapping Annotation의 값을 이용해서 클라이언트의 요청을 어떤 컨트롤러의 어떤 메서드가

처리할 지가 결정된다. 매칭이 어떻게 이루어지는 지 알지 못할 경우 클라이언트의 요청 URL 매칭되는 메서드를 설정할 수 없기 때문에,

어떻게 매칭이 이루어지는 지 아는 것은 작지만 매우 중요하다. 본 절에서는 매칭 관련된 기본 내용인 경로와 @RequestMapping Annotation 간의

관계 그리고 @PathVariable을 이용한 URI 템플릿에 대해서 살펴보도록 하자.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

8.1 전체 경로와 Servlet 기반 경로 매칭 설정

: DispatcherServlet은 DefaultAnnotationHandlerMapping Class를 기본 HandlerMapping 구현체로 사용한다. 기본 설정은 컨텍스트 내에서의

전체 경로가 아닌 Servlet 경로를 제외한 나머지 경로에 대해 Mapping을 처리한다. 예를 들어, DispatcherServlet에 대해서 다음과 같이

Servlet Mapping을 설정했다고 하자.

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>\*.do</url-pattern>

<url-pattern>/game/\*</url-pattern>

</servlet-mapping>

위 Code는 \*.do로 오는 요청과 /game/ 으로 오는 요청을 dispatcher Servlet이 처리하도록 설정하고 있다. 이 경우 다음의 두

@RequestMapping은 어떻게 될까?

@RequestMapping("/search/game.do")

public String search(..){

...

}

@RequestMapping("/game/info")

public String info(..){

...

}

Code만 보면 /search/game.do에 대한 요청은 search() 메서드가 처리하고 /game/info에 대한 요청은 info() 메서드가 처리한다고 생각하기

쉽다. 하지만, search() 메서드만 /search/game.do에 대한 요청을 처리하고, info() 메서드는 /game/info 요청을 처리하지 않는다.

info() 메서드가 /game/info 요청을 처리하지 않는 이유는 Servlet Mapping 설정에서 <url-pattern>의 값을 "/game/\*"으로 설정했기 때문이다.

<url-pattern>의 값으로 디렉터리를 포함한 패턴을 지정하게 되면 Servlet 경로는 "/game"이 되며, Servlet 경로를 제외한 나머지 경로를

이용해서 @RequestMapping Annotation의 값과 매칭 여부를 판단하게 되므로, 실제 비교할 때 사용되는 요청 URI는 "/info"가 된다.

@RequestMapping Annotation의 값은 "/game/info"이므로 비교할 요청 URI인 "info"와는 매칭되지 않게 된다.

반면에 /search/game.do 요청은 <url-pattern>이 \*.do인 확장자 기반의 매칭이 되므로 Servlet 경로가 ""이 된다. 따라서, /search/game.do

요청은 값이 "/search/game.do"인 @RequestMapping Annotation과 매칭된다.

Servlet 경로를 제외한 나머지 경로를 사용하는 이유는 DispatcherServlet이 기본적으로 사용하는 HandlerMapping 구현체

(DefaultAnnotationHandlerMapping)와 HandlerAdapter 구현체(AnnotationMethodHandlerAdapter)가 전체 경로를 사용하지 않도록

설정되어 있기 때문이다. 만약 Servlet 경로를 포함한 전체 경로를 이용해서 매칭 여부를 판단하도록 설정하려면, 다음과 같이 SPRING 설정

파일에 이 두 구현체의 alwaysUserFullPath 프로퍼티 값이 true가 되도록 설정해 부어야 한다.

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" p:alwaysUserFullPath="true"/>

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter" p:alwaysUserFullPath="true"/>

DispatcherServlet은 DispatcherServlet이 사용하는 SPRING 설정 파일에 HandlerMapping 구현 Class와 HandlerAdapter 구현 Class가 빈으로

등록되어 있으면, 기본 구현체를 사용하지 않고 설정 파일에 등록된 구현체를 사용하게 된다. 따라서, 위와 같이 설정 파일에 두 Class를 빈으로

등록해주면 alwaysUserFullPath 프로퍼티 값이 false인 기본 설정이 아니라 alwaysUserFullPath 값이 true인 설정을 사용하게 된다.

alwaysUserFullPath 프로퍼티의 설정 값을 true로 지정해주면 Servlet 경로를 포함한 전체 경로를 사용하게 되므로, 아래와 같이 Servlet Mapping을

설정한 경우,

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>\*.do</url-pattern>

<url-pattern>/game/\*</url-pattern>

</servlet-mapping>

http://host/컨텍스트경로/game/info 요청 URL에 대해 다음의 @RequestMapping Annotation이 매칭이 되어 info() 메서드가 /game/info 요청을

처리하게 된다.

@RequestMapping("/game/info") //game은 Servlet 경로

public String info(..){

...

}

<Example008>

[GameInfoController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

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

@Controller

public class GameInfoController {

@RequestMapping("/game/info")

public String gameInfo() {

return "game/users/info";

}

@RequestMapping("/game/list")

public String gameList() {

return "game/list";

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.GameInfoController" />

...

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="alwaysUseFullPath" value="true" />

</bean>

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="alwaysUseFullPath" value="true" />

</bean>

...

[/WEB-INF/view/game/users/info.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>캐릭터 정보</title>

</head>

<body>

${userId} 회원의 정보 출력

</body>

</html>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

8.2 @PathVariable Annotation을 이용한 URI 템플릿

: RESTful 서비스가 유행하면서 URI를 다음과 같이 REST 방식으로 구성하는 경우가 늘어나고 있다.

- http://somehost/users/madvirus

- http://somehost/games

- http://somehost/forum/board1/10

기존에는 http://somehost/users/userinfo?id=madvirus와 같이 파라미터를 이용해서 아이디나 이름을 전달받았다면, 이제는 URI에 아이디나

이름 등이 포함되도록 URL을 구성하고 있다.

SPRING 3버전에 추가된 기능 중 필자가 가장 반기는 기능 중의 하나가 URI 템플릿이며, 이 URI 템플릿을 이용하면 REST 방식의 URL 매칭을

쉽게 처리할 수 있다. URI 템플릿을 사용하는 방법은 매우 간단하며, 다음과 같이 두 가지만 추가로 작업해주면 된다.

- @RequestMapping Annotation의 값으로 {템플릿변수}를 사용한다.

- @PathVariable Annotation을 이용해서 {템플릿변수}와 동일한 이름을 갖는 파라미터를 추가한다.

실제 URI 템플릿이 적용된 Code는 다음과 같다.

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

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

@Controller

public class CharacterInfoController {

@RequestMapping("/game/users/{userId}/characters/{characterId}")

public String characterInfo(@PathVariable String userId,

@PathVariable int characterId, ModelMap model) {

model.addAttribute("userId", userId);

model.addAttribute("characterId", characterId);

return "game/character/info";

}

}

위 Code를 보면 @RequestMapping Annotation은 이름이 userId와 characterId인 변수를 포함하고 있다. 이들 변수는 @PathVariable Annotation이

적용된 동일한 이름을 갖는 파라미터에 매칭된다. 따라서 요청 URI가 "/game/users/madvirus/character/1"이면, characterInfo() 메서드의 userId

파라미터와 characterId 파라미터의 값은 각각 "madvirus"와 "1"이 된다.

만약, 파라미터 이름과 URI 템플릿의 변수 이름이 동일하지 않다면, 다음과 같이 @PathVariable Annotation에 매칭될 템플릿 변수 이름을 지정해

주면 된다.

public String characterInfo(@PathVariable String userId, @PathVariable int characterId, ModelMap model) {

...

}

---------------------------------------------------------------------------------------------------------------------------------------------------------------

8.3 @RequestMapping Annotation의 추가 설정 방법

: @RequestMapping Annotation을 Class와 메서드에 함께 적용할 경우, 메서드에 적용한 @RequestMapping Annotation의 값은 Class에 적용한

@RequestMapping Annotation의 값을 기본 경로로 사용하게 된다. 다음 Code는 사용 예이다.

@Controller

@RequestMapping("/game/users/{userId}")

public class CharacterInfoController {

@RequestMapping("/characters/{characterId}")

public String characterInfo(@PathVariable String userId,

@PathVariable int characterId, ModelMap model) {

model.addAttribute("userId", userId);

model.addAttribute("characterId", characterId);

return "game/character/info";

}

}

위 Code에서 characterInfo() 메서드에 적용된 @RequestMapping Annotation의 값은 "/characters/{characterId}"인데, 실제로 매칭되는 값은 Class에

적용된 @RequestMapping Annotation의 값을 포함한 "/game/users/{userId}/characters/{chracterId}"가 된다.

@RequestMapping Annotation은 Ant 스타일의 패턴을 지원한다. 따라서, 다음과 같이 경로명에 "\*"나 "\*\*"를 값으로 사용할 수 있다.

@RequestMapping("/members/\*.do")

public String memberInfo(...){

...

}

@RequestMapping("/game/\*/items/{itemId}")

public String itemInfo(...){

...

}

<Example009>

[CharacterInfoController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

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

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

@Controller

@RequestMapping("/game/users/{userId}")

public class CharacterInfoController {

@RequestMapping("/characters/{characterId}")

public String characterInfo(@PathVariable String userId,

@PathVariable int characterId, ModelMap model) {

model.addAttribute("userId", userId);

model.addAttribute("characterId", characterId);

return "game/character/info";

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.CharacterInfoController" />

...

[/WEB-INF/view/character/users/info.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>캐릭터 정보</title>

</head>

<body>

${userId} 회원의 ${characterId} 캐릭터 정보 출력

</body>

</html>

http://localhost:8080/SpringMVC01/game/users/madvirus/characters/10

===============================================================================

9. 폼 입력 값 검증

: SPRING은 객체가 유효한 지의 여부를 검사할 때 사용되는 Validator Interface와 유효성 검사 결과를 저장할 Errors Interface를 제공하고 있다.

SPRING MVC도 Validator Interface를 이용해서 폼 값을 저장한 커맨드 객체의 유효성 여부를 검사하고, 입력 값이 유효하지 않을 경우 Errors에

에러 Message를 저장해서 뷰에 전달하게 된다. 본 절에서는 SPRING MVC에서 폼 값을 검증하고 에러 Message를 처리하는 방법을 살펴보도록 하자.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

9.1 Validator Interface를 이용한 폼 값 검증

: org.springframework.validation.Validator Interface는 객체를 검증할 때 사용된다. 커맨드 객체를 사용하는 컨트롤 Class들은 커맨드 객체를

생성한 뒤 Validator에 커맨드 객체를 전달하여 검증을 요청한다.

Validator Interface는 다음과 같이 두 개의 메서드를 정의하고 있다.

- boolean supports(Class<?> clazz) : Validator가 해당 Class에 대한 값 검증을 지원하는 지의 여부를 리턴한다.

- void validate(Object target, Errors errors) : target 객체에 대한 검증을 실행한다. 검증 결과 문제가 있을 경우 errors 객체에 어떤 문제인지에 대한

정보를 저장한다.

아래 Code는 MemberInfo 타입의 객체를 검증하는 Validator의 구현 예를 보여주고 있다.

[MemberInfoValidator.java]

package madvirus.spring.chap06.validator;

import madvirus.spring.chap06.model.Address;

import madvirus.spring.chap06.model.MemberInfo;

import org.springframework.validation.Errors;

import org.springframework.validation.Validator;

public class MemberInfoValidator implements Validator {

@Override

public boolean supports(Class<?> clazz) {

return MemberInfo.class.isAssignableFrom(clazz);

}

@Override

public void validate(Object target, Errors errors) {

MemberInfo memberInfo = (MemberInfo) target;

if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {

errors.rejectValue("id", "required");

}

if (memberInfo.getName() == null || memberInfo.getName().trim().isEmpty()) {

errors.rejectValue("name", "required");

}

Address address = memberInfo.getAddress();

if (address == null) {

errors.rejectValue("address", "required");

}

if (address != null) {

errors.pushNestedPath("address");

try {

if (address.getZipcode() == null || address.getZipcode().trim().isEmpty()) {

errors.rejectValue("zipcode", "required");

}

if (address.getAddress1() == null || address.getAddress1().trim().isEmpty()) {

errors.rejectValue("address1", "required");

}

} finally {

errors.popNestedPath();

}

}

}

}

supports() 메서드는 검증할 객체의 타입(Class) 정보를 파라미터로 전달받는데, 위의 Code는 검사 대상 객체의

MemberInfo Class 또는 그 하위 Class인지의 여부를 검사한다.

validate() 메서드는 두 개의 파라미터를 전달받는데, 첫 번째는 검사할 객체이고, 두 번째는 에러 정보를 저장하기 위한

Errors 객체이다. 위의 Code에서는 첫 번째 파라미터를 MemberInfo 타입으로 변환한 뒤에 값을 겁사한다. 만약, 값이

올바르지 않다면 관련 정보를 Errors 객체에 저장한다. 라인 19~20은 id 프로퍼티가 필수 입력인 경우의 처리 예를 보여주고 있다.

if (memberInfo.getName() == null || memberInfo.getName().trim().isEmpty()) {

errors.rejectValue("name", "required");

}

위 Code에서 errors.rejectValue() 메서드는 "id" 프로퍼티의 값이 올바르지 않고 에러 Code는 "required"라는 것을 기록한다.

이 에러 Code는 뷰에서 에러 Message를 출력할 때 사용된다. Errors가 제공하는 메서드에 대해서는 본 장의 'Errors Interface와

BindingResult Interface'에서 살펴보도록 하겠다.

Validator Class를 작성했다면 그 다음으로 할 작업은 컨트롤러 Class에서 Validator를 이용해서 커맨드 객체를 검증하는 것이다.

검증하는 Code를 추가하려면 다음의 작업을 구핸하면 된다.

- @RequestMapping Annotation 메서드에서 커맨드 객체 다음 파라미터로 BindingResult 타입이나 Errors 타입의 파라미터를 추가한다.

- 메서드 내에서 Validator 객체를 생성한 뒤 validate() 메서드에서 커맨드 객체와 BindingResult 또는 Errors 타입 파라미터를 전달한다.

- Errors hasErrors() 메서드를 이용해서 에러가 존재하는지 확인하고, 에러가 존재할 경우 알맞은 처리를 수행한다.

앞서 작성한 MemberValidator를 이용해서 폼 값을 검증하는 컨트롤러 Code는 다음과 같이 작성할 수 있다.

[CreateAccountController.java]

package madvirus.spring.chap06.controller;

import javax.servlet.http.HttpServletRequest;

import madvirus.spring.chap06.model.Address;

import madvirus.spring.chap06.model.MemberInfo;

import madvirus.spring.chap06.validator.MemberInfoValidator;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

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

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

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

@Controller

@RequestMapping("/account/create.do")

public class CreateAccountController {

@ModelAttribute("command")

public MemberInfo formBacking(HttpServletRequest request) {

if (request.getMethod().equalsIgnoreCase("GET")) {

MemberInfo mi = new MemberInfo();

Address address = new Address();

address.setZipcode(autoDetectZipcode(request.getRemoteAddr()));

mi.setAddress(address);

return mi;

} else {

return new MemberInfo();

}

}

private String autoDetectZipcode(String remoteAddr) {

return "000000";

}

@RequestMapping(method = RequestMethod.GET)

public String form() {

return "account/creationForm";

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") MemberInfo memberInfo, BindingResult result) {

new MemberInfoValidator().validate(memberInfo, result);

if (result.hasErrors()) {

return "account/creationForm";

}

return "account/created";

}

}

위의 Code는 MemberInfoValidator를 이용해서 memberInfo 객체가 유효한 지의 여부를 검사한다. 라인 44에서 validate()

메서드의 두 번째 파라미터로 BindingResult 타입인 result를 전달하는 대, BindingResult Interface는 Errors Interface를

상속받았기 때문에 두 번째 파라미터로 전달 가능하다.

validate() 메서드 내부에서 Errors.rejectValue() 메서드 또는 Errors.reject() 메서드가 한 번 이상 호출되면 검증 과정에서 에러가

발생하는 것으로 인식되며, 이 경우 Errors.hasErrors() 메서드는 true를 리턴한다. 위의 Code에서는 에러가 발생하면 입력 폼을

보여주는 뷰 이름을 리턴해서 다시 폼 입력을 받도록 구현하고 있다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

9.2 Errors Interface와 BindingResult Interface

: org.springframework.validation.Errors Interface는 유효성 검증 결과를 저장할 때 사용되며,

org.springframework.validation.BindingResult Interface는 Errors Interface의 하위 Interface로서 폼 값을 커맨드 객체에

바인딩(binding)한 결과를 저장하고 에러 Code로부터 에러 Message를 가져온다.

Errors Interface는 값이 올바르지 않을 경우 reject() 또는 rejectValue() 메서드를 이용해서 어떤 필드가 값이 잘못 되었으며,

관련된 에러 Code가 무엇인지 입력받는다. reject() 메서드는 검증 대상 객체의 전체적인 에러를 설정하며, rejectValue() 메서드는

특정 프로퍼티(필드)의 검증 에러를 설정한다. Errors Interface가 제공하는 reject() 메서드 및 rejectValue() 메서드는 다음과 같다.

각 메서드의 리턴 타입은 void이다.

- reject(String errorCode)

: 전체 객체에 대한 글로벌 에러 Code를 추가한다.

- reject(String errorCode, String defaultMessage)

: 전체 객체에 대한 글로벌 에러 Code를 추가한다. 에러 Code에 대한 Message가 존재하지 않을 경우 defaultMessage를 사용한다.

- reject(String errorCode, Object[] errorArgs, String defaultMessage)

: 전체 객체에 대한 글로벌 에러 Code를 추가한다. Message 인자로 errorArgs를 전달한다. 에러 Code에 대한 Message가 존재하지 않을

경우 defaultMessage를 사용한다.

- rejectValue(String field, String errorCode)

: 필드에 대한 에러 Code를 추가한다.

- rejectValue(String field, String errorCode, String defaultMessage)

: 필드에 대한 에러 Code를 추가한다. 에러 Code에 대한 Message가 존재하지 않을 경우 defaultMessage를 사용한다.

- rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage)

: 필드에 대한 에러 Code를 추가한다. Message 인자로 errorArgs를 전달한다. 에러 Code에 대한 Message가 존재하지 않을 경우

defaultMessage를 사용한다.

reject() 메서드는 객체 자체의 에러 정보를 추가할 때 사용된다. 예를 들어, 로그인을 처리하는 경우 아이디와 암호가 일치하지 않아서

로그인 폼을 보여주고, 에러 정보를 출력해 주곤 한다. 이 경우는 아이디 필드나 암호 필드 자체에 에러가 있다기 보다는 객체 자체에

문제가 있는 것이다. 이렇게 객체 자체에 문제가 있는 경우에는 아래 Code와 같이 reject() 메서드를 사용해서 글로벌 에러 정보를

추가할 수 있다.

@Controller

@RequestMapping("/login/login.do")

public class LoginController {

private String formViewName = "login/form";

@Autowired

private Authenticator authenticator;

@RequestMapping(method = RequestMethod.GET)

public String form() {

return formViewName;

}

@ModelAttribute

public LoginCommand formBacking() {

return new LoginCommand();

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@Valid LoginCommand loginCommand, BindingResult result) {

if (result.hasErrors()) {

return formViewName;

}

try {

authenticator.authenticate(loginCommand);

return "redirect:/index.jsp";

} catch (AuthenticationException e) {

result.reject("invalidIdOrPassword", new Object[] { loginCommand.getUserId() }, null);

return formViewName;

}

}

...

}

rejectValue() 메서드는 객체의 개별 프로퍼티(필드)에 대한 에러 정보를 추가할 때 사용된다. 앞서 살펴봤던 MemberInfoValidator.java의

validate() 메서드의 일부 Code는 아래와 같은데, 아래 Code를 보면 각 프로퍼티의 값이 유효한 지 검사하고 유효하지 않을 경우

rejectValue() 메서드를 이용해서 해당 프로퍼티에 대한 에러 정보를 추가하는 것을 확인할 수 있다.

@Override

public void validate(Object target, Errors errors) {

MemberInfo memberInfo = (MemberInfo) target;

if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {

errors.rejectValue("id", "required");

}

if (memberInfo.getName() == null || memberInfo.getName().trim().isEmpty()) {

errors.rejectValue("name", "required");

}

...

}

글로벌 에러 정보나 특정 필드에 대한 에러 정보는 두 번 이상 추가될 수 있다. 예를 들어, 다음과 같이 특정 필드에 대해 에러 정보를

두 번 이상 등록할 수 있다.

errors.rejectValue("id", "id.invalidLength");

errors.rejectValue("id", "id.invalidCharacter");

두 개 이상의 에러 정보가 추가되면, 뷰에서 에러 Message를 보여줄 때 추가된 개수 만큼의 에러 Message가 출력된다.

(1) Errors Interface의 에러 발생 여부 확인 메서드

: Errors Interface는 에러가 발생했는지의 여부를 확인할 수 있도록 다음과 같은 메서드를 제공하고 있다.

- boolean hasErrors() : 에러가 존재하는 경우 true를 리턴한다.

- int getErrorCount() : 여러 개수를 리턴한다.

- boolean hasGlobalErrors() : reject() 메서드를 이용해서 추가된 글로벌 에러가 존재할 경우 true를 리턴한다.

- int getGlobalErrorCount() : reject() 메서드를 이용해서 추가된 글로벌 에러 개수를 리턴한다.

- boolean hasFieldErrors() : rejectValue() 메서드를 이용해서 추가된 에러가 존재할 경우 true를 리턴한다.

- int getFieldErrorCount() : rejectValue() 메서드를 이용해서 추가된 에러 개수를 리턴한다.

- boolean hasFieldErrors(String field) : rejectValue() 메서드를 이용해서 추가한 특정 필드의 에러가 존재할 경우 true를 리턴한다.

- int getFieldErrorCount(String field) : rejectValue() 메서드를 이용해서 추가한 특정 필드의 에러 개수를 리턴한다.

위 메서드를 이용하면 에러 존재 여부에 따라서 알맞은 처리를 수행할 수 있다. 예를 들어, 폼 검증 결과 에러가 존재할 경우 다시 폼을

보여주도록 하고 싶다면, 다음과 같이 hasErrors() 메서드를 이용하면 된다.

@Controller

@RequestMapping("/account/create.do")

public class CreateAccountController {

...

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") MemberInfo memberInfo, BindingResult result) {

new MemberInfoValidator().validate(memberInfo, result);

if (result.hasErrors()) {

return "account/creationForm";

}

return "account/created";

}

...

}

---------------------------------------------------------------------------------------------------------------------------------------------------------------

9.3 DefaultMessageCodesResolver와 에러 Message

: 앞서 Errors Interface의 reject() 메서드와 rejectValue() 메서드는 에러 Code를 사용하여 에러 정보를 전달한다는 것을 살펴봤었다.

BindingResult Interface의 기본 구현 Class인 AbstractBindingResult Class는 MessageCodesResolver를 사용하여 에러 Code에 대한

에러 Message를 추출하는데, 기본적으로 DefaultMessageCodesResolver Class를 MessageCodesResolver로 사용한다.

DefaultMessageCodesResolver는 MessageSource로부터 에러 Code에 해당하는 Message를 로딩한다. 따라서, DefaultMessageCodesResolver를

이용하여 에러 Message를 생성하려면 다음과 같이 알맞은 MessageSource를 빈 객체로 등록해 주어야 한다.

<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">

<property name="basenames">

<list>

<value>messages.validation</value>

</list>

</property>

</bean>

DefaultMessageCodesResolver는 에러 Code로부터 Message Code를 생성한 뒤, Message Code에 해당하는 Message를 로딩한다. reject() 메서드를

이용하여 커맨드 객체 자체에 대한 에러 Code를 입력한 경우, 다음의 순서로 Message Code를 생성한다.

[1] 에러 Code + "." + 커맨드 객체 이름

[2] 에러 Code

rejectValue() 메서드를 이용하여 특정 필드에 대한 에러 Code를 입력했다면, 다음의 순서로 Message Code를 생성한다.

[1] 에러 Code + "." + 커맨드 객체 이름 + "." + 필드명

[2] 에러 Code + "." + 필드명

[3] 에러 Code + "." + 필드 타입

[4] 에러 Code

만약, 첫 번째롤 사용한 Message Code가 존재하지 않으면, 두 번째 Message Code를 사용한다. 비슷한 방식으로 Message Code가 발견될 때까지

세 번째와 네 번째 Message Code를 차례대로 사용하게 된다.

예를 들어, 커맨드 객체의 이름이 "loginCommand"이고, 다음과 같이 rejectValue() 메서드를 사용하여 "userId" 프로퍼티(필드)에 대하여

에러 Code "required"를 추가했다고 하자.

errors.rejectValue("userId", "required");

이 경우, 사용되는 Message Code의 순서는 다음과 같다.

[1] required.loginCommand.userId

[2] required.userId

[3] required.java.lang.String

[4] required

에러 Message를 출력할 때에는 SPRING이 제공하는 <form:errors> 커스텀 태그를 사용하면 된다. JSP가 아닌 Velocity와 같은 기술을 사용하는

경우에도 SPRING은 Velocity에 알맞은 매크로를 제공하고 있어서 에러 Message를 손쉽게 출력할 수 있다.

뷰로 JSP를 사용할 경우 다음의 두 가지 방식으로 에러 Message를 설정할 수 있다.

- <spring:hasBindErrors> 커스텀 태그를 이용하여 에러 정보를 설정

- <form:form> 태그를 이용하여 에러 정보를 설정

다음 Code는 <spring:hasBindErrors> 태그를 이용하여 에러 Message를 출력하는 예제 Code이다. 다음 Code는 회원 가입 폼을 보여주는 JSP Code의

예로, <spring:hasBindErrors> 태그와 <form:errors> 태그를 이용하여 커맨드 객체의 id, name, address 등의 에러 Message를 출력하고 있다.

(MemberInfoValidator.java의 MemberInfoValidator Class가 생성한 에러Message를 출력한다.)

[createForm.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>계정 생성</title>

</head>

<body>

<spring:hasBindErrors name="command" />

<form:errors path="command" />

<form method="post">

아이디: <input type="text" name="id" value="${command.id}" />

<form:errors path="command.id" />

<br/>

이름: <input type="text" name="name" value="${command.name}" />

<form:errors path="command.name" />

<br/>

우편번호: <input type="text" name="address.zipcode" value="${command.address.zipcode}" />

<form:errors path="command.address.zipcode" />

<br/>

주소1: <input type="text" name="address.address1" value="${command.address.address1}" />

<form:errors path="command.address.address1" />

<br/>

주소2: <input type="text" name="address.address2" value="${command.address.address2}" />

<form:errors path="command.address.address2" />

<br/>

<input type="submit" />

</form>

</body>

</html>

라인 10에서 <spring:hasBindErrors> 커스텀 태그는 name 속성에 명시한 커맨드 객체와 에러 정보를 <form:errors>

커스텀 태그에서 사용할 수 있도록 설정한다. 일단, 에러 정보를 설정하고 나면 <form:errors> 태그를 이용하여 특정

프로퍼티와 관련된 에러 Message를 출력할 수 있다. 라인 14의 경우 command 커맨드 객체의 id 프로퍼티와 관련된 모든

에러 Message를 출력한다.

앞서 MemberInfoValidator.java의 MemberInfoValidator Class의 Code를 보면 아래 Code와 같이 "id" 프로퍼티의 에러 Code를

"required"로 설정하였다.

public class MemberInfoValidator implements Validator {

...

@Override

public void validate(Object target, Errors errors) {

MemberInfo memberInfo = (MemberInfo) target;

if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {

errors.rejectValue("id", "required");

}

...

}

}

createForm.jsp의 라인 14에서 사용한 <form:errors> 커스텀 태그는 "command.id"와 관련된 모든 에러 Message를 출력하므로

"required" 에러 Code에 해당하는 Message를 출력하게 된다.

아래 Code는 <form:form> 커스텀 태그를 이용하여 에러 정보를 설정하는 경우의 예를 보여 주고 있다. <form:form> 커스텀 태그는

커맨드 객체와 입력 폼 사이의 정보를 처리할 때 사용되는 커스텀 태그로서 커맨드 객체와 관련된 에러 정보도 함께 설정하게 된다.

[form.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>로그인</title>

</head>

<body>

<form:form commandName="loginCommand">

<form:errors element="div"/>

아이디: <form:input path="userId"/>

<form:errors path="userId" />

<br/>

암호: <form:password path="password" showPassword="false"/>

<form:errors path="password" />

<br/>

<input type="submit" />

</form:form>

</body>

</html>

<form:form> 커스텀 태그의 commandName 속성은 커맨드 객체의 이름을 값으로 갖는다. <form:form> 커스텀 태그는

<form:form> 커스텀 태그에서 설정한 커맨드 객체를 기본으로 사용하기 때문에 path 속성에서는 프로퍼티 이름만 설정하면

된다. 위의 Code에서 라인 13의 경우 loginCommand 객체의 userId 프로퍼티와 관련된 모든 에러 Message를 출력하게 된다.

SPRING이 제공하는 커스텀 태그 및 Velocity를 위한 매크로는 '7장. 뷰 영역 구현'에서 보다 자세히 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

9.4 ValidationUtils Class를 이용한 값 검증

: Validator 구현 Class의 validate() 메서드에서는 프로퍼티의 값이 존재하는 지의 여부를 판단하기 위해 다음과 같이 값이 null 인지의

여부와 공백 문자열인지의 여부를 확인한다.

public void validate(Object target, Errors errors) {

MemberInfo memberInfo = (MemberInfo) target;

if (memberInfo.getId() == null || memberInfo.getId().trim().isEmpty()) {

errors.rejectValue("id", "required");

}

}

각 프로퍼티에 대해서 매번 위와 같은 Code를 작성하는 것은 매우 성가신 일이다. org.springframework.validation.ValidationUtils Class가

제공하는 메서드를 사용하면 이런 성가신 Code를 조금 더 편하게 작성할 수 있다. ValidationUtils Class는 다수의 rejectIfEmpty() 메서드와

rejectIfEmptyOrWhitespace() 메서드를 제공하고 있으며, 이들 메서드를 이용해서 다음과 같이 위 Code를 대체할 수 있다.

public void validate(Object target, Errors errors) {

ValidatorUtils.rejectIfEmptyOrWhitespace(errors, "id", "required");

...

}

ValidatorUtils.rejectIfEmpty() 메서드는 값이 null이거나 길이가 0인 경우 에러Code를 추가하며, rejectIfEmptyOrWhitespace() 메서드는

값이 null이거나 길이가 0이거나 또는 값이 공백 문자로 구성되어 있는 경우 에러 Code를 추가한다. 이 두 이름의 메서드 목록은 아래와 같다.

- rejectIfEmpty(Errors errors, String field, String errorCode)

- rejectIfEmpty(Errors errors, String field, String errorCode, String defaultMessage)

- rejectIfEmpty(Errors errors, String field, String errorCode, Object[] errorArgs)

- rejectIfEmpty(Errors errors, String field, String errorCode, Object[] errorArgs, String defaultMessage)

- rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode)

- rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, String defaultMessage)

- rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, Object[] errorArgs)

- rejectIfEmptyOrWhitespace(Errors errors, String field, String errorCode, Object[] errorArgs, String defaultMessage)

메서드 파라미터에서 errorArgs는 에러 Message에 삽입될 인자 목록이고, defaultMessage는 에러 Code에 해당하는 에러 Message가 존재하지

않을 때 사용할 기본 Message다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

9.5 @Valid Annotation과 @InitBinder Annotation을 이용한 검증 실행

: 앞서 살펴봤던 Validator 이용 Code를 보면 다음과 같이 Validator 객체를 생성한 뒤 직접 validate() 메서드를 호출했었다.

@RequestMapping(method = RequestMethod.POST)

public String submit(@ModelAttribute("command") MemberInfo memberInfo, BindingResult result) {

new MemberInfoValidator().validate(memberInfo, result);

if (result.hasErrors()) {

return "account/creationForm";

}

...

}

JSR 303의 @Valid Annotation을 이용하면 커맨드 객체를 검사하는 Code를 직접적으로 호출하지 않고 SPRING Framework가 호출하도록

설정할 수 있다. JSR 303은 Bean Validation API로서 Bean Validation API에 정의된 @Valid Annotation은 연관된 객체의 유효성을 검증한다는 것을

표시하기 위한 Annotation이다.

SPRING 3 MVC는 JSR 303의 @Valid Annotation과 SPRING Framework의 @InitBinder Annotation을 이용해서 Validator에 대한 직접적인

호출없이 SPRING Framework가 유효성 검사 Code를 실행하도록 할 수 있다. 아래 Code는 @Valid Annotation과 @InitBinder Annotation을

사용한 Code의 작성 예이다.

[LoginController.java]

package madvirus.spring.chap06.controller;

import javax.validation.Valid;

import madvirus.spring.chap06.service.AuthenticationException;

import madvirus.spring.chap06.service.Authenticator;

import madvirus.spring.chap06.service.LoginCommand;

import madvirus.spring.chap06.validator.LoginCommandValidator;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.WebDataBinder;

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

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

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

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

@Controller

@RequestMapping("/login/login.do")

public class LoginController {

private String formViewName = "login/form";

@Autowired

private Authenticator authenticator;

@RequestMapping(method = RequestMethod.GET)

public String form() {

return formViewName;

}

@ModelAttribute

public LoginCommand formBacking() {

return new LoginCommand();

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@Valid LoginCommand loginCommand, BindingResult result) {

if (result.hasErrors()) {

return formViewName;

}

try {

authenticator.authenticate(loginCommand);

return "redirect:/index.jsp";

} catch (AuthenticationException e) {

result.reject("invalidIdOrPassword", new Object[] { loginCommand.getUserId() }, null);

return formViewName;

}

}

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.setValidator(new LoginCommandValidator());

}

public void setAuthenticator(Authenticator loginService) {

this.authenticator = loginService;

}

}

위의 Code 라인 36을 보면 커맨드 객체 파라미터에 @Valid Annotation을 적용하고 있고, submit() 메서드 내부에서

Validator.validate() 메서드를 명시적으로 호출하고 있지 않다. submit() 메서드는 단지 두 번째 파라미터로 전달 받은

BindingResult를 이용해서 라인 38~40과 같이 에러가 존재할 경우 다시 폼을 보여주도록 하고 있다.

어떤 Validator가 커맨드 객체를 검증할 지의 여부는 라인 51~54에 정의한 initBinding() 메서드를 통해서 결정된다.

SPRING은 @InitBinder Annotation이 적용된 메서드를 이용해서 폼과 커맨드 객체 사이의 Mapping을 처리해주는

WebDataBinder를 초기화 할 수 있도록 하고 있다. 이 메서드에서 WebDataBinder.setValidator() 메서드를 이용해서

커맨드 객체의 유효성 여부를 검색할 때 사용할 Validator를 설정하게 된다.

앞서 작성했던 form.jsp 파일이 LoginController와 관련된 뷰 Code인데, 실제로 WEB 브라우저에서 로그인 폼에 아이디와

암호를 입력하지 않으면 LoginValidator가 설정한 에러 Code에 해당하는 에러 Message가 폼에 출력된다.

아래 Code와 같이 @Valid Annotation이 적용된 커맨드 객체 다음에 BindingResult나 Errors 타입의 파라미터를 전달받지

않는다고 해 보자.

@RequestMapping(method = RequestMethod.POST)

public String submit(@Valid LoginCommand loginCommand) {

try {

authenticator.authenticate(loginCommand);

return "redirect:/index.jsp";

} catch (AuthenticationException e) {

...

}

}

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.setValidator(new LoginCommandValidator());

}

위와 같이 @RequestMapping Annotation이 적용된 메서드가 BindingResult나 Errors 타입의 파라미터를 갖지 않는다는 것은

에러가 발생한 경우에 대한 처리를 메서드에서 수행하지 않는다는 것을 의미한다. 이런 경우 SPRING MVC는 @Valid Annotation이

적용된 커맨드 객체를 검사한 뒤 에러가 존재하면 500 에러를 발생시킨다.

<Example010>

8[LoginController.java]

package madvirus.spring.chap06.controller;

import javax.validation.Valid;

import madvirus.spring.chap06.service.AuthenticationException;

import madvirus.spring.chap06.service.Authenticator;

import madvirus.spring.chap06.service.LoginCommand;

import madvirus.spring.chap06.validator.LoginCommandValidator;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.WebDataBinder;

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

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

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

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

@Controller

@RequestMapping("/login/login.do")

public class LoginController {

private String formViewName = "login/form";

@Autowired

private Authenticator authenticator;

@RequestMapping(method = RequestMethod.GET)

public String form() {

return formViewName;

}

@ModelAttribute

public LoginCommand formBacking() {

return new LoginCommand();

}

@RequestMapping(method = RequestMethod.POST)

public String submit(@Valid LoginCommand loginCommand,

BindingResult result) {

if (result.hasErrors()) {

return formViewName;

}

try {

authenticator.authenticate(loginCommand);

return "redirect:/index.jsp";

} catch (AuthenticationException e) {

result.reject("invalidIdOrPassword", new Object[] { loginCommand

.getUserId() }, null);

return formViewName;

}

}

@InitBinder

protected void initBinder(WebDataBinder binder) {

binder.setValidator(new LoginCommandValidator());

}

public void setAuthenticator(Authenticator loginService) {

this.authenticator = loginService;

}

}

1[LoginCommand.java]

package madvirus.spring.chap06.service;

public class LoginCommand {

private String userId;

private String password;

public String getUserId() {

return userId;

}

public void setUserId(String userId) {

this.userId = userId;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

}

2[AuthenticationException.java]

package madvirus.spring.chap06.service;

@SuppressWarnings("serial")

public class AuthenticationException extends Exception {

}

3[Authenticator.java]

package madvirus.spring.chap06.service;

public interface Authenticator {

void authenticate(LoginCommand loginCommand)

throws AuthenticationException;

}

7[MockAuthenticator.java]

package madvirus.spring.chap06.service;

import madvirus.spring.chap06.common.CommonLogger;

public class MockAuthenticator implements Authenticator {

private CommonLogger commonLogger;

@Override

public void authenticate(LoginCommand loginCommand)

throws AuthenticationException {

if (!loginCommand.getUserId().equals(loginCommand.getPassword())) {

commonLogger.log("인증 에러 - " + loginCommand.getUserId());

throw new AuthenticationException();

}

}

public void setCommonLogger(CommonLogger commonLogger) {

this.commonLogger = commonLogger;

}

}

6[LoginCommandValidator.java]

package madvirus.spring.chap06.validator;

import madvirus.spring.chap06.service.LoginCommand;

import org.springframework.validation.Errors;

import org.springframework.validation.ValidationUtils;

import org.springframework.validation.Validator;

public class LoginCommandValidator implements Validator {

@Override

public boolean supports(Class<?> clazz) {

return LoginCommand.class.isAssignableFrom(clazz);

}

@Override

public void validate(Object target, Errors errors) {

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "userId", "required");

ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",

"required");

}

}

4[CommonLogger.java]

package madvirus.spring.chap06.common;

public interface CommonLogger {

void log(String message);

}

5[CommonLoggerImpl.java]

package madvirus.spring.chap06.common;

public class CommonLoggerImpl implements CommonLogger {

@Override

public void log(String message) {

System.out.println("CommonLogger - " + message);

}

}

[/src/common.xml]

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:context="http://www.springframework.org/schema/context" xmlns:p="http://www.springframework.org/schema/p"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<bean id="commonLogger" class="madvirus.spring.chap06.common.CommonLoggerImpl" />

</beans>

[dispatcher-servlet.xml]

...

<bean id="authenticator" class="madvirus.spring.chap06.service.MockAuthenticator"

p:commonLogger-ref="commonLogger" />

<bean class="madvirus.spring.chap06.controller.LoginController">

<property name="authenticator" ref="authenticator" />

</bean>

...

[web.xml]

...

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

classpath:common.xml

</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

...

[/WEB-INF/view/login/form.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>로그인</title>

</head>

<body>

<form:form commandName="loginCommand">

<form:errors element="div"/>

아이디: <form:input path="userId"/>

<form:errors path="userId" />

<br/>

암호: <form:password path="password" showPassword="false"/>

<form:errors path="password" />

<br/>

<input type="submit" />

</form:form>

</body>

</html>

[/WebContent/index.jsp]

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<!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>스프링 예제</title>

</head>

<body>

<ul>

<li><a href="<c:url value='/hello.do' />">/hello.do</a> (HelloController)</li>

<li><a href="<c:url value='/article/newArticle.do' />">/article/newArticle.do</a> (NewArticleController, 커맨드 객체 예제)</li>

<li><a href="<c:url value='/order/order.do' />">/order/order.do</a> (OrderController, 커맨드 객체로 List 받기 예제)</li>

<li><a href="<c:url value='/search/internal.do' />?query=q&p=2">/search/internal.do</a>

<a href="<c:url value='/search/external.do' />?query=p&">/search/external.do</a>

(SearchController, @RequestParam 예제)</li>

<li><a href="<c:url value='/cookie/make.do' />">/cookie/make.do</a>

<a href="<c:url value='/cookie/view.do' />">/cookie/view.do</a>

(CookieController, @CookieValue 예제)</li>

<li><a href="<c:url value='/header/check.do' />">/header/check.do</a>

(HeaderController, @RequestHeader 예제)</li>

<li><a href="<c:url value='/search/main.do' />">/search/main.do</a>

(GameSearchController, @ModelAttribute 예제)</li>

<li><a href="<c:url value='/account/create.do' />">/account/create.do</a>

(CreateAccountController, @ModelAttribute 및 폼 값 검증 예제)</li>

<li><a href="<c:url value='/game/users/madvirus/characters/10' />">/game/users/{userId}/characters/{characterId}</a>

(CharacterInfoController, @PathVariable 예제)</li>

<li><a href="<c:url value='/login/login.do' />">/login/login.do</a>

(LoginController, @InitBinder 및 @Valid를 이용한 폼 값 검증)</li>

<li><a href="<c:url value='/report/submission.do' />">/report/submission.do</a>

(ReportSubmissionController, 파일 업로드 예제)</li>

<li><a href="<c:url value='/log/query.do' />?from=2010-04-01&to=2010-04-30">/log/query.do</a>

(QueryLogController, @InitBinder 및 WebBindingInitializer 예제)</li>

<li><a href="<c:url value='/event/list.do' />">/event/list.do</a>

(EventController, HandlerInterceptor 예제)</li>

<li><a href="<c:url value='/search/game.do' />">/search/game.do</a>

(GameSearchController, @ExceptionHandler 예제)</li>

<li><a href="<c:url value='/math/divide.do' />?op1=1&op2=0">/math/divide.do?op1=1&amp;op2=0</a>

(ArithmeticOperatorController, SimpleMappingExceptionResolver 예제)</li>

<li><a href="<c:url value='/guestmessage/xml.do' />">/guestmessage/xml.do</a>

(GuestMessageController, @ResponseBody 예제)</li>

</ul>

</body>

</html>

===============================================================================

10. 파일 업로드 처리

: 파일 업로드가 필요한 경우 HTML 폼의 속성을 "multipart/form-data"로 설정해야 한다.

<form method="post" enctype="multipart/form-data">

...

</form>

인코딩 타입이 Multipart인 경우 파라미터나 업로드 한 파일을 구하려면 전송 데이터를 알맞게 처리해 주어야 한다. SPRING은

Multipart 지원 기능을 제공하고 있기 때문에, 이 기능을 이용하면 추가적인 처리 없이 Multipart 지원 기능을 이용해서 업로드 한

파일을 처리하는 방법을 살펴보도록 하겠다.

본 절에서는 SPRING이 제공하는 Multipart 지원 기능을 이용해서 업로드 한 파일을 처리하는 방법을 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

10.1 MultipartResolver 설정

: Multipart 지원 기능을 사용하려면 먼저 MultipartResolver를 SPRING 설정 파일에 등록해 주어야 한다. MultipartResolver는

Multipart 형식으로 데이터가 전송된 경우, 해당 데이터를 SPRING MVC에서 사용할 수 있도록 변환해준다. 예를 들어,

@PathVariable Annotation을 이용해서 Multipart로 전송된 파라미터와 파일을 사용할 수 있도록 해 준다.

SPRING이 기본으로 제공하는 MultipartResolver는 CommonsMultipartResolver이다. CommonsMultipartResolver는 Commons

FileUpload API를 이용해서 Multipart를 처리해 준다. CommonsMultipartResolver를 MultipartResolver로 사용하려면 다음과 같이

빈 이름으로 "multipartResolver"를 사용해서 등록하면 된다.

<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

</bean>

CommonsMultipartResolver Class는 업로드와 관련해서 아래와 같은 프로퍼티를 제공하고 있다.

[[프로퍼티]] [[타 입]] [[설 명]]

==============================================================================

maxUploadSize long 최대 업로드 가능한 바이트 크기. -1은 제한이 없음을 의미한다.

기본 값은 -1이다.

------------------------------------------------------------------------------------------------------------------------------------------------------------

maxInMemorySize int 디스크에 임시 파일을 생성하기 전에 메모리에 보관할 수 있는 최대 바이트

크기. 기본 값은 10240 바이트이다.

------------------------------------------------------------------------------------------------------------------------------------------------------------

defaultEncoding String 요청을 파싱할 때 사용할 캐릭터 인코딩. 지정하지 않을 경우,

HttpServletRequest.setCharacterEncoding() 메서드로 지정한 캐릭터 셋이

사용된다. 아무 값도 없을 경우 ISO-8859-1을 사용한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------------------------------------------------------

10.2 @RequestParam Annotation을 이용한 업로드 파일 접근

: 업로드한 파일을 전달받는 첫 번째 방법은 @RequestParam Annotation이 적용된 MultipartFile 타입의 파라미터를 사용하는 것이다.

예를 들어, HTML 입력폼이 다음과 같이 작성되어 있다고 해 보자.

<form action="submitReport1.do" method="post" enctype="multipart/form-data">

학번 : <input type="text" name="studentNumber"/>

<br/>

리포트 파일 : <input type="file" name="report"/>

<br/>

<input type="submit"/>

</form>

위 HTML Code에서 파일은 report 파라미터를 통해서 전달된다. 이 경우 다음 Code와 같이 RequestParam Annotation과 MultipartFile

타입의 파라미터를 이용해서 업로드 파일 데이터를 전달받을 수 있다.

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

import org.springframework.web.multipart.MultipartFile;

@Controller

public class ReportSubmissionController {

...

@RequestMapping(value = "/report/submitReport1.do", method = RequestMethod.POST)

public String submitReport1(

@RequestParam("studentNumber") String studentNumber,

@RequestParam("report") MultipartFile report) {

.. // MultipartFile이 제곤하는 메서드를 이용해서 업로드 데이터 접근

return "report/submissionComplete";

}

...

}

MultipartFile Interface는 SPRING에서 업로드 한 파일을 표현할 때 사용되는 Interface로서, MultipartFile Interface를 이용해서

업로드 한 파일의 이름, 실제 데이터, 파일의 크기 등을 구할 수 있다. MultipartFile Interface 제공하는 메서드는 본 장의

'MultipartFile Interface 사용' 절에서 살펴볼 것이다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

10.3 MultipartHttpServletRequest를 이용한 업로드 파일 접근

: 업로드한 파일을 전달 받는 두 번째 방법은 MultipartHttpServletRequest Interface를 사용하는 것이다.

import org.springframework.web.multipart.MultipartFile;

import org.springframework.web.multipart.MultipartHttpServletRequest;

@Controller

public class ReportSubmissionController {

...

@RequestMapping(value = "/report/submitReport2.do", method = RequestMethod.POST)

public String submitReport2(MultipartHttpServletRequest request) {

String studentNumber = request.getParameter("studentNumber");

MultipartFile report = request.getFile("report");

...

}

}

MultipartHttpServletRequest Interface는 SPRING이 제공하는 Interface로서, Multipart 요청이 들어올 때 내부적으로 원본

HttpServletRequest 대신 사용되는 Interface이다. MultipartHttpServletRequest Interface는 실제로는 어떤 메서드도

선언하고 있지 않으며, 아래와 같이 HttpServletRequest Interface와 MultipartRequest Interface를 상속받고 있다.

<<Interface>> <<Interface>>

HttpServletRequest MultipartRequest

|| ||

|| ||

===================

||

<<Interface>>

MultipartHttpServletRequest

MultipartHttpServletRequest Interface는 javax.servlet.http.HttpServletRequest Interface를 상속받기 때문에 WEB 요청 정보를

구하기 위한 getParameter()나 getHeader()와 같은 메서드를 사용할 수 있으며, 추가로 MultipartRequest Interface가 제공하는

Multipart 관련 메서드를 사용할 수 있다.

MultipartRequest Interface가 제공하는 업로드 파일 관련 주요 메서드는 아래와 같다.

[[메서드]] [[설 명]]

=============================================================================

Iterator<String> getFileNames() 업로드 된 파일들의 이름 목록을 제공하는 Iterator를 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------

MultipartFile getFile(String name) 파라미터 이름이 name인 업로드 파일 정보를 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------

List<MultipartFile> getFiles(String name) 파라미터 이름이 name인 업로드 파일 정보 목록을 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------

Map<String, MultipartFile> getFileMap() 파라미터 이름을 키로 파라미터에 해당하는 파일 정보를 값으로 하는

Map을 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------------------------------------------------------

10.4 커맨드 객체를 통한 업로드 파일 접근

: 커맨드 객체를 이용해도 업로드 한 파일을 전달받을 수 있다. 단지 커맨드 Class에 파라미터와 동일한 이름의 MultipartFile 타입

프로퍼티만 추가해주기만 하면 된다. 예를 들어, 업로드 파일의 파라미터 이름이 "report"인 경우, 다음과 같이 "report" 프로퍼티를

커맨드 Class에 추가해 주면 된다.

package madvirus.spring.chap06.controller;

import org.springframework.web.multipart.MultipartFile;

public class ReportCommand {

private String studentNumber;

private MultipartFile report;

public String getStudentNumber() {

return studentNumber;

}

public void setStudentNumber(String studentNumber) {

this.studentNumber = studentNumber;

}

public MultipartFile getReport() {

return report;

}

public void setReport(MultipartFile report) {

this.report = report;

}

}

---------------------------------------------------------------------------------------------------------------------------------------------------------------

10.5 MultipartFile Interface 사용

: org.springframework.web.multipart.MultipartFile Interface는 업로드 한 파일 정보 및 파일 데이터를 표현하기 위한 용도로

사용된다. MultipartFile Interface가 제공하는 주요 메서드는 다음과 같다.

[[메 서 드]] [[설 명]]

==========================================================================

String getName() 파라미터 이름을 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------

String getOriginalFilename() 업로드 한 파일의 이름을 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------

boolean isEmpty() 업로드 한 파일이 존재하지 않을 경우 true를 리턴한다.

----------------------------------------------------------------------------------------------------------------------------------------------------

long getSize() 업로드 한 파일의 크기를 구한다.

----------------------------------------------------------------------------------------------------------------------------------------------------

byte[] getBytes() 업로드 한 파일의 데이터를 구한다.

throws IOException

----------------------------------------------------------------------------------------------------------------------------------------------------

InpuStream getInputStream() 업로드 한 파일 데이터를 읽어오는 InputStream을 구한다.

throws IOException InputStream의 사용이 끝나면 알맞게 종료해 주어야 한다.

----------------------------------------------------------------------------------------------------------------------------------------------------

void transferTo(File dest) 업로드 한 파일 데이터를 지정한 파일에 저장한다.

throws IOEception

----------------------------------------------------------------------------------------------------------------------------------------------------

업로드 한 파일 데이터를 구하는 가장 단순한 방법은 MultipartFile.getFiles() 메서드를 이용하는 것이다. 바이트 배열을 구한 뒤에

파일이나 DB 등에 저장하면 된다.

if(!multipartFile.isEmpty()){

byte[] fileData = multipartFile.getBytes();

//byte 배열을 파일/DB/네트워크 등으로 전송

...

}

업로드 한 파일 데이터를 특정 파일로 저장하고 싶다면 MultipartFile.transferTo() 메서드를 사용하는 것이 편리하다.

if(!multipartFile.isEmpty()){

File file = new File(fileName);

multipartFile.transferTo(file);

...

}

<Example011>

[ReportSubmissionController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

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

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

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

import org.springframework.web.multipart.MultipartFile;

import org.springframework.web.multipart.MultipartHttpServletRequest;

@Controller

public class ReportSubmissionController {

@RequestMapping(value = "/report/submission.do", method = RequestMethod.GET)

public String form() {

return "report/submissionForm";

}

@RequestMapping(value = "/report/submitReport1.do", method = RequestMethod.POST)

public String submitReport1(

@RequestParam("studentNumber") String studentNumber,

@RequestParam("report") MultipartFile report) {

printInfo(studentNumber, report);

return "report/submissionComplete";

}

private void printInfo(String studentNumber, MultipartFile report) {

System.out.println(studentNumber + "가 업로드 한 파일: "

+ report.getOriginalFilename());

}

@RequestMapping(value = "/report/submitReport2.do", method = RequestMethod.POST)

public String submitReport2(MultipartHttpServletRequest request) {

String studentNumber = request.getParameter("studentNumber");

MultipartFile report = request.getFile("report");

printInfo(studentNumber, report);

return "report/submissionComplete";

}

@RequestMapping(value = "/report/submitReport3.do", method = RequestMethod.POST)

public String submitReport3(ReportCommand reportCommand) {

printInfo(reportCommand.getStudentNumber(), reportCommand.getReport());

return "report/submissionComplete";

}

}

[ReportCommand.java]

package madvirus.spring.chap06.controller;

import org.springframework.web.multipart.MultipartFile;

public class ReportCommand {

private String studentNumber;

private MultipartFile report;

public String getStudentNumber() {

return studentNumber;

}

public void setStudentNumber(String studentNumber) {

this.studentNumber = studentNumber;

}

public MultipartFile getReport() {

return report;

}

public void setReport(MultipartFile report) {

this.report = report;

}

}

[dispatcher-servlet.xml]

...

<bean id="multipartResolver"

class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

</bean>

...

<bean class="madvirus.spring.chap06.controller.ReportSubmissionController" />

...

[/WEB-INF/view/report/submissionForm.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>리포트 제출</title>

</head>

<body>

<h3>@RequestParam 사용</h3>

<form action="submitReport1.do" method="post" enctype="multipart/form-data">

학번: <input type="text" name="studentNumber" />

<br/>

리포트파일: <input type="file" name="report" />

<br/>

<input type="submit" />

</form>

<h3>MultipartHttpServletRequest 사용</h3>

<form action="submitReport2.do" method="post" enctype="multipart/form-data">

학번: <input type="text" name="studentNumber" />

<br/>

리포트파일: <input type="file" name="report" />

<br/>

<input type="submit" />

</form>

<h3>커맨드 객체 사용</h3>

<form action="submitReport3.do" method="post" enctype="multipart/form-data">

학번: <input type="text" name="studentNumber" />

<br/>

리포트파일: <input type="file" name="report" />

<br/>

<input type="submit" />

</form>

</body>

</html>

[/WEB-INF/view/report/submissionComplete.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>리포트 제출 완료</title>

</head>

<body>

리포트 제출 완료

</body>

</html>

===============================================================================

11. @InitBinder Annotation과 커스텀 데이터 타입 변환 처리

: SPRING은 기본적으로 문자열을 지정한 타입의 값으로 변환할 때 Java빈의 Property Editor를 사용한다. SPRING MVC가 HTTP 요청

파라미터를 객체의 프로퍼티 값으로 저장할 때 사용하는 WebDataBinder 역시 내부적으로 PropertyEditor를 사용하게 되는데,

본 절에서는 WebDataBinder와 SPRING이 제공하는 PropertyEditor를 이용해서 HTTP 요청 파라미터 값을 알맞게 변환하는 방법을

살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

11.1 @InitBinder Annotation을 이용한 Date 타입 변환 처리

: 값이 '2011-05-05'인 HTTP 요청 파라미터를 java.util.Date 타입의 프로퍼티에 저장하고 싶을 때가 있는데, 이 경우 아래 Code와 같이

SPRING MVC의 WebDataBinder에 커스텀 PropertyEditor를 등록해주면 된다.

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import org.springframework.beans.propertyeditors.CustomDateEditor;

import org.springframework.web.bind.WebDataBinder;

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

...

@Controller

public class QueryLogController {

@ModelAttribute("command")

public QueryLogCommand formBacking() {

return new QueryLogCommand();

}

@RequestMapping("/log/query.do")

public String query(@ModelAttribute("command") QueryLogCommand command, BindingResult result) {

...

return "log/logList";

}

@InitBinder

protected void initBinder(WebDataBinder binder) {

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));

}

}

SPRING은 java.util.Date 타입에 대한 PropertyEditor인 CustomDateEditor를 제공하고 있으며, 이 CustomDateEditor를

WebDataBinder에 등록함으로써 문자열을 Date 타입으로 변환할 수 있다. 따라서 다음과 같이 커맨드 Class에 java.util.Date

타입의 프로퍼티를 추가해도 폼에 입력한 파라미터 값을 Date 타입의 프로퍼티로 전달받을 수 있다.

import java.util.Date;

public class QueryLogCommand {

private Date from;

private Date to;

public Date getFrom() {

return from;

}

public void setFrom(Date from) {

this.from = from;

}

public Date getTo() {

return to;

}

public void setTo(Date to) {

this.to = to;

}

}

CustomDateEditor Class의 생성자는 첫 번째 파라미터로 String을 Date 타입으로 변환할 때 사용할 DateFormat을 전달받는다.

두 번째 파라미터는 값으로 null이거나 빈 문자열이 오는 것을 허용할 지의 여부를 지정한다.

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));

위 Code와 같이 두 번째 파라미터 값을 true로 지정하면 HTTP 요청파라미터 값이 존재하지 않거나 빈 문자열인 경우 null을 값으로

설정하게 된다. 반면에 두 번째 파라미터 값을 false로 설정하면 요청 파라미터 값이 존재하지 않거나 빈 문자열인 경우 검증 에러가

발생하게 된다. 이 때 에러 Code는 "typeMismatche"가 된다. 따라서 BindingResult를 통해서 에러 발생 여부를 확인할 수 있다.

@RequestMapping("/log/query.do")

public String query(@ModelAttribute("command") QueryLogCommand command,BindingResult result) {

if(result.hasErrors()){

...

}

...

return "log/logList";

}

<Example012>

[QueryLogController.java]

package madvirus.spring.chap06.controller;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import org.springframework.beans.propertyeditors.CustomDateEditor;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;

import org.springframework.web.bind.WebDataBinder;

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

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

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

@Controller

public class QueryLogController {

@ModelAttribute("command")

public QueryLogCommand formBacking() {

return new QueryLogCommand();

}

@RequestMapping("/log/query.do")

public String query(@ModelAttribute("command") QueryLogCommand command,

BindingResult result) {

return "log/logList";

}

@InitBinder

protected void initBinder(WebDataBinder binder) {

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

binder.registerCustomEditor(Date.class, new CustomDateEditor(

dateFormat, true));

}

}

[QueryLogCommand.java]

package madvirus.spring.chap06.controller;

import java.util.Date;

public class QueryLogCommand {

private Date from;

private Date to;

public Date getFrom() {

return from;

}

public void setFrom(Date from) {

this.from = from;

}

public Date getTo() {

return to;

}

public void setTo(Date to) {

this.to = to;

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.QueryLogController" />

...

[/WEB-INF/view/log/logList.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>로그 목록</title>

</head>

<body>

<form:form method="get">

<form:errors path="from" element="div"/>

<form:errors path="to" element="div"/>

시작일: <form:input path="from" />

종료일: <form:input path="to" />

<input type="submit" value="조회" />

</form:form>

</body>

</html>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

11.2 WebBindingInitializer를 이용한 공통 PropertyEditor 등록

: @InitBinder Annotation이 적용된 메서드가 컨트롤러 단위로 PropertyEditor를 등록하는 데 사용된다면, WebBindingInitializer는 전체

컨트롤러에 공통으로 적용되는 PropertyEditor를 등록하는 데 사용된다. WebBindingInitializer Interface는 다음과 같이 정의되어 있다.

package org.springframework.web.bind.support;

import org.springframework.web.bind.WebDataBinder;

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

public interface WebBindingInitializer{

void initBinder(WebDataBinder binder, WebRequest request)

}

initBinder() 메서드는 컨트롤러에 전달되는 WebDataBinder를 초기화 할 때 사용되는 메서드로서, 첫 번째 파라미터로 전달되는

WebDataBinder의 registerCustomEditor() 메서드를 이용해서 전체 컨트롤러에서 공통으로 사용될 PropertyEditor를 등록해주면

된다. 아래 Code는 WebBindingInitializer Interface의 구현 예를 보여주고 있다.

[CustomWebBindingInitializer.java]

package madvirus.spring.chap06.binder;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import org.springframework.beans.propertyeditors.CustomDateEditor;

import org.springframework.web.bind.WebDataBinder;

import org.springframework.web.bind.support.WebBindingInitializer;

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

public class CustomWebBindingInitializer implements WebBindingInitializer {

@Override

public void initBinder(WebDataBinder binder, WebRequest request) {

DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, true));

}

}

WebBindingInitializer 구현 Class를 작성했다면, AnnotationMethodHandlerAdapter의 webBindingInitializer 프로퍼티

값으로 구현 Class의 빈을 전달해주면 된다. 아래 Code와 같이 설정해주면 CustomWebBindingInitializer에서 등록해주는

PropertyEditor가 모든 컨트롤러의 WebDataBinder에 등록된다.

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="alwaysUseFullPath" value="true"/>

<property name="webBindingInitializer" >

<bean class="madvirus.spring.chap06.binder.CustomWebBindingInitializer"/>

</property>

</bean>

===============================================================================

12. HandlerInterceptor를 통한 요청 가로채기

: SPRING이 기본적으로 제공하는 HandlerMapping은 HandlerInterceptor를 이용해서 컨트롤러가 요청을 처리하기 전과 처리한 후에

알맞은 기능을 수행할 수 있도록 하고 있다. 조건에 따라 컨트롤러에 요청을 전달하지 않고 싶거나 컨트롤러가 요청을 처리한 후에

ModelAndView 객체를 조작하고 싶은 경우에 HandlerInterceptor를 사용하면 된다.

HandlerInterceptor Interface는 다음과 같이 세 개의 메서드를 정의하고 있다.

- boolean preHanndle(HttpServletRequest request, HttpServletResponse, Object handler)

- void postHandle(HttpServletRequest request, HttpServletResponse, Object handler, ModelAndView modelAndView)

- void afterCompletion(HttpServletRequest request, HttpServletResponse, Object handler, Exception ex)

preHandle() 메서드는 클라이언트의 요청을 컨트롤러에 전달하기 전에 호출된다. 세 번째 파라미터인 handler에는 컨트롤러 객체가

전달된다. 한 개 이상의 HandlerInterceptor가 체인을 형성하게 되는데, preHandle() 메서드가 false를 리턴하면 체인의 다음 HandlerInterceptor

또는 컨트롤러를 실행하지 않고 요청 처리를 종료하게 된다.

postHandle() 메서드는 컨트롤러가 요청을 처리한 뒤에 호출된다. HandlerInterceptor 체인에서 postHandle() 메서드는 preHandle() 메서드의

실행 순서와 반대로 실행된다. 컨트롤러 실행 도중에 예외가 발생할 경우 postHandle() 메서드는 실행되지 않는다.

afterCompletion() 메서드는 클라이언트의 요청을 처리한 뒤, 즉, 뷰를 통해서 클라이언트에 응답을 전송한 뒤에 실행된다. 컨트롤러가 처리하는

도중이나 뷰를 생성하는 과정에 예외가 발생하더라도 afterCompletion() 메서드는 실행된다. afterCompletion() 메서드는 preHandle() 메서드의

실행 순서와 반대로 실행된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

12.1 HandlerInpterceptor Interface의 구현

: org.springframework.web.servlet.HandlerInpterceptor Interface를 직접 구현하여 Class를 작성할 수 있지만, 이 경우 사용하지 않는 메서드도

구현해 주어야 한다. 예를 들어, preHandle() 메서드만 필요하더라도 나머지 두 메서드도 구현해 주어야 하는 것이다. SPRING은 이런 불편함을

줄여 주기 위해 HandlerInpterceptorAdapter Class를 제공하고 있다.

HandlerInpterceptorAdapter Class는 HandlerInpterceptor Interface를 구현한 Class로서, 세 개의 메서드는 아무 기능도 제공하지 않는다.

HandlerInpterceptor Interface를 구현해야 하는 Class는 HandlerInpterceptorAdapter Class를 상속 받은 뒤, 세 개의 메서드 중에서

필요한 메서드 만을 오버라이딩 해서 구현해 주면 된다. 아래 Code는 HandlerInpterceptorAdapter Class를 상속 받아 구현한

HandlerInpterceptor의 구현 예이다.

[EventExpirationCheckInterceptor.java]

package madvirus.spring.chap06.interceptor;

import java.io.IOException;

import java.util.Calendar;

import java.util.Date;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class EventExpirationCheckInterceptor extends HandlerInterceptorAdapter {

@Override

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

if (checkEvent(request) && checkEventExpiration()) {

displayEventExpirationPage(request, response);

return false;

}

return true;

}

private boolean checkEvent(HttpServletRequest request) {

return request.getRequestURI().equals(

request.getContextPath() + "/event/list.do");

}

private boolean checkEventExpiration() {

Calendar calendar = Calendar.getInstance();

calendar.set(2017, 6, 30);

Date now = new Date();

return now.after(calendar.getTime());

}

private void displayEventExpirationPage(HttpServletRequest request,

HttpServletResponse response) throws IOException {

response.sendRedirect("eventExpired.do");

}

}

위의 Code는 컨트롤러에 요청을 전달하기 전에 preHandle() 메서드를 이용해서 이벤트 기간이 만료되었는지의

여부를 검사한 뒤 만료되었을 경우 종료 페이지를 출력하는 구현 HandlerInterceptor이다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

12.2 HandlerMapping에 HandlerInterceptor 설정하기

: HandlerInterceptor를 구현한 뒤에는 HandlerMapping의 interceptors 프로퍼티를 사용해서 HandlerInterceptor를

등록해 주면 된다. 아래 Code는 설정 예이다.

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

<property name="interceptors">

<list>

<ref bean="eventExpirationCheckInterceptor"/>

</list>

<property>

</bean>

<bean id="eventExpirationCheckInterceptor" class="madvirus.spring.chap06.interceptor.EventExpirationCheckInterceptor"/>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

12.3 HandlerInterceptor의 실행 순서

: HandlerMapping에는 한 개 이상의 HandlerInterceptor를 등록할 수 있는데, 이 경우 다음과 같은 순서로 HandlerInterceptor가 실행된다.

[1] 컨트롤러 실행 전 : 등록된 순서대로 preHandle() 메서드 실행

[2] 컨트롤러 실행 후 : 등록된 순서의 반대로 postHandle() 메서드 실행

[3] 처리 완료 후(뷰 생성 후) : 등록된 순서의 반대로 afterCompletion() 메서드 실행

예를 들어, 다음과 같이 3개의 HandlerInterceptor를 등록했다고 하자.

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="interceptors">

<list>

<ref bean="interceptor1"/>

<ref bean="interceptor2"/>

<ref bean="interceptor3"/>

</list>

<property>

...

</bean>

이 경우 클라이언트의 요청을 처리할 때 실행되는 메서드 순서는 다음과 같다.

[4] preHandle() 메서드 실행 :

a) interceptor1.preHandle()

b) interceptor2.preHandle()

c) interceptor3.preHandle()

[5] 컨트롤러의 handleRequest() 메서드 실행

[6] postHandle() 메서드 실행 :

a) interceptor3.postHandle()

b) interceptor2.postHandle()

c) interceptor1.postHandle()

[7] 뷰 객체의 render() 메서드 실행해서 응답 결과 생성

[8] afterCompletion() 메서드 실행 :

a) interceptor3.afterCompletion()

b) interceptor2.afterCompletion()

c) interceptor1.afterCompletion()

<Example013>

[EventController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

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

@Controller

public class EventController {

@RequestMapping("/event/list.do")

public String list() {

return "event/list";

}

@RequestMapping("/event/eventExpired.do")

public String expired() {

return "event/eventExpired";

}

}

[EventExpirationCheckInterceptor.java]

package madvirus.spring.chap06.interceptor;

import java.io.IOException;

import java.util.Calendar;

import java.util.Date;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class EventExpirationCheckInterceptor extends HandlerInterceptorAdapter {

@Override

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

if (checkEvent(request) && checkEventExpiration()) {

displayEventExpirationPage(request, response);

return false;

}

return true;

}

private boolean checkEvent(HttpServletRequest request) {

return request.getRequestURI().equals(

request.getContextPath() + "/event/list.do");

}

private boolean checkEventExpiration() {

Calendar calendar = Calendar.getInstance();

calendar.set(2015, 3, 15);

Date now = new Date();

return now.after(calendar.getTime());

}

private void displayEventExpirationPage(HttpServletRequest request,

HttpServletResponse response) throws IOException {

response.sendRedirect("eventExpired.do");

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.EventController" />

...

<bean

class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">

<property name="alwaysUseFullPath" value="true" />

<property name="interceptors">

<list>

<ref bean="eventExpirationCheckInterceptor" />

</list>

</property>

</bean>

<bean id="eventExpirationCheckInterceptor" class="madvirus.spring.chap06.interceptor.EventExpirationCheckInterceptor" />

...

[/WEB-INF/view/event/list.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>이벤트 목록</title>

</head>

<body>

이벤트 목록 출력

</body>

</html>

[/WEB-INF/view/event/eventExpired.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>이벤트 종료</title>

</head>

<body>

이벤트 종료됨

</body>

</html>

===============================================================================

13. 예외 처리

: 컨트롤러 Class의 @RequestMapping 메서드는 모든 타입의 예외를 발생시킬 수 있다. @RequestMapping 메서드가 예외를

발생시킬 경우 WEB 브라우저에 500 응답 Code와 함께 Servlet Container가 출력한 에러 페이지가 출력된다. 만약 Servlet Container가

생성한 에러 페이지가 아니라 예외 타입에 따라 SPRING MVC와 연동된 뷰를 이용해서 에러 페이지를 출력하고 싶다면, SPRING이

제공하는 HandlerExceptionResolver Interface를 사용하면 된다.

HandlerExceptionResolver Interface는 다음과 같이 정의되어 있다.

public interface HandlerExceptionResolver{

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

}

HandlerExceptionResolver Interface의 resolveException() 메서드는 컨트롤러 객체(handler)와 발생한 예외(ex)를 파라미터로

전달받는다. 또한, 에러 페이지 뷰 이름과 모델 객체를 담고 있는 ModelAndView 타입의 객체를 리턴한다.

따라서, HandlerExceptionResolver Interface를 이용하면 발생한 예외 타입에 따라 알맞은 뷰를 이용해서 에러 화면을 출력할 수 있다.

SPRING 3이 제공하는 HandlerExceptionResolver Interface의 주요 구현 Class는 다음과 같다.

- AnnotationMethodHandlerExceptionResolver : @ExceptionHandler Annotation이 이용된 메서드를 이용해서 예외를 처리한다.

- DefaultHandlerExceptionResolver : NoSuchRequestHandlingMethodException과 같이 SPRING 관련 예외 타입을 처리해준다.

- SimpleMappingExceptionResolver : 예외 타입 별로 뷰 이름을 지정할 때 사용된다.

SPRING 설정 파일에 HandlerExceptionResolver가 등록되어 있지 않을 경우 AnnotationMethodHandlerExceptionResolver와

DefaultHandlerExceptionResolver이 기본적으로 사용된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

13.1 @ExceptionHandler Annotation을 이용한 예외 처리

: AnnotationMethodHandlerExceptionResolver Class는 @Controller Annotation이 적용된 Class에서 @ExceptionHandler Annotation

적용된 메서드를 이용해서 예외를 처리한다. 아래 Code는 적용 예이다.

import org.springframework.stereotype.Controller;

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

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

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

import org.springframework.web.servlet.ModelAndView;

@Controller

public class GameSearchController {

@RequestMapping("/search/game.do")

public ModelAndView search(@ModelAttribute("command") SearchCommand command) {

ModelAndView mav = new ModelAndView("search/game");

...

SearchResult result = searchService.search(command);

mav.addObject("searchResult", result);

return mav;

}

@ExceptionHandler(NullPointerException.class)

public String handleNullPointerException(NullPointerException ex) {

return "error/nullException";

}

...

}

위 Code에서 @ExceptionHandler Annotation은 값으로 예외 타입의 Class를 갖는데, 이 경우 해당 컨트롤러의 @RequestMapping

Annotation이 적용된 메서드에서 @ExceptionHandler Annotation에서 지정한 타입의 예외가 발생하면 @ExceptionHandler Annotation이

적용된 메서드를 이용해서 예외를 보여줄 뷰를 지정하게 된다.

@ExceptionHandler Annotation이 적용된 메서드는 @RequestMapping Annotation이 적용된 메서드와 마찬가지로 다양한 타입의

파라미터와 리턴 타입을 가질 수 있다. @ExceptionHandler 메서드가 가질 수 있는 파라미터 타입은 다음과 같다.

- HttpServletRequest, HttpServletResponse, HttpSession

- Locale

- InputStream / Reader

- OutputStream / Writer

- 예외 타입

@ExceptionHandler 메서드가 가질 수 있는 리턴 타입은 다음과 같다.

- ModelAndView

- Model, Map

- View

- String

- void

Model과Map을 리턴 타입으로 사용할 경우 RequestToViewNameTranslator를 이용해서 뷰 이름이 결정되며, String을 리턴 타입으로

사용할 경우 리턴 값이 뷰 이름으로 사용된다.

@ExceptionHandler Annotation이 적용된 메서드는 ModelAndView, Model, Map을 이용해서 모델 정보를 전달할 수 있기 때문에,

예외에 따라 알맞은 모델 객체를 뷰에 전달할 수 있다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

13.2 SimpleMappingExceptionResolver Class를 이용한 에러 페이지 지정

: SimpleMappingExceptionResolver Class는 예외 타입 이름과 특정 뷰 이름을 Mapping할 때 사용된다. SimpleMappingExceptionResolver를

사용하려면 다음과 같이 SPRING 파일을 설정해 주면 된다.

<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<prop key="java.lang.ArithmeticException">

error/mathException

</prop>

<prop key="java.lang.Exception">

error/exception

</prop>

</props>

</property>

</bean>

위 Code는 ArithmeticException 예외가 발생할 경우 error/mathException을 뷰로 사용하고, Exception 예외가 발생할 경우 error/Exception을

뷰로 사용한다.

SimpleMappingExceptionResolver를 빈으로 등록할 경우 AnnotationMethodHandlerExceptionResolver가 사용되지 않으므로, @ExceptionHandler

Annotation을 사용하려면 AnnotationMethodHandlerExceptionResolver를 빈으로 함께 등록해 주어야 한다.

<Example014>

[ArithmeticOperatorController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;

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

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

@Controller

public class ArithmeticOperatorController {

@RequestMapping("/math/add.do")

public String add(@RequestParam("op1") int op1, @RequestParam("op2") int op2, Model model) {

model.addAttribute("result", op1 + op2);

return "math/result";

}

@RequestMapping("/math/subtract.do")

public String subtract(@RequestParam("op1") int op1, @RequestParam("op2") int op2, Model model) {

model.addAttribute("result", op1 - op2);

return "math/result";

}

@RequestMapping("/math/multiply.do")

public String multiply(@RequestParam("op1") int op1, @RequestParam("op2") int op2, Model model) {

model.addAttribute("result", op1 \* op2);

return "math/result";

}

@RequestMapping("/math/divide.do")

public String divide(@RequestParam("op1") int op1, @RequestParam("op2") int op2, Model model) {

model.addAttribute("result", op1 / op2);

return "math/result";

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.ArithmeticOperatorController" />

...

<bean

class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<prop key="java.lang.ArithmeticException">

error/mathException

</prop>

<prop key="java.lang.Exception">

error/exception

</prop>

</props>

</property>

</bean>

...

[/WEB-INF/view/event/exception.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>예외 발생</title>

</head>

<body>

요청을 처리하는 도중에 예외가 발생하였습니다:

${exception.message}

<%

Throwable e = (Throwable) request.getAttribute("exception");

e.printStackTrace();

%>

</body>

</html>

[/WEB-INF/view/event/mathException.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>예외 발생</title>

</head>

<body>

연산 과정에서 예외가 발생하였습니다.

</body>

</html>

[/WEB-INF/view/event/nullException.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>예외 발생</title>

</head>

<body>

요청을 처리하는 과정에서 예외(null)가 발생하였습니다.

</body>

</html>

===============================================================================

14. 캐시 옵션 설정

: AnnotationMethodHandlerAdapter는 캐시 헤더를 설정하는 기능을 제공하고 있다. 따라서, 모든 컨트롤러에 동일한 캐시 옵션을 설정해야 할 경우,

동일한 Code를 모든 컨트롤러 Code에서 중복할 필요 없이 AnnotationMethodHandlerAdapter의 캐시 관련 프로퍼티를 이용해서 캐시 헤더를

설정할 수 있다. 캐시 관련 프로퍼티는 다음과 같다.

[[프 로 퍼 티]] [[설 명]]

=======================================================================================

useExpiredHeader HTTP 1.0의 Expires 헤더를 지정할 지의 여부를 설정한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

useCacheControlHeader HTTP 1.1의 Cache-Control 헤더를 지정할 지의 여부를 설정한다. 기본 값은 true이다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

useCacheControlNoStore HTTP 1.1의 Cache-Control 헤더의 값으로 no-store를 지정할 지의 여부를 설정한다.

기본 값은 true이다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

cacheSeconds 캐시 시간을 설정한다. 값이 0 이상인 경우에만 캐시 관련 헤더를 생성한다. 기본 값은 -1 이다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

cacheSeconds 프로퍼티의 값이 0인 경우 캐시를 하지 않도록 헤더를 생성하고, -1인 경우 캐시 관련 헤더를 생성하지 않는다. cacheSeconds 프로퍼티의

값이 1 이상인 경우 지정한 시간만큼 캐시하도록 헤더를 설정한다. 아래 Code는 캐시를 하지 않도록 설정한 Code의 예를 보여주고 있다.

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="cacheSeconds" value="0"/>

<property name="alwaysUseFullPath" value="true"/>

<property name="webBindingInitializer">

<bean class="madvirus.spring.chap06.binder.CustomWebBindingInitializer"/>

</property>

</bean>

캐시 시간을 0으로 설정한 경우 실제 응답 결과에는 다음과 같이 캐시를 설정하지 않도록 관련 헤더 값이 포함된다.

Pragma : no-cache

Expires : Thu, 01 Jan 1970 00:00:00 GMT

Cache-Control : no-cache, no-store

===============================================================================

15. Servlet 관련 Code에서 WebApplicationContext 직접 접근하기

: Java Code를 작성하다 보면 SPRING Container를 통해서 관리되지 않는 객체에서 SPRING 빈을 사용하고 싶은 때가 있다. 예를 들어, 보조 Class의

static 메서드에서 SPRING Container가 관리하는 빈 객체를 이용해야 하는 경우나 JSP Code나 Velocity Code에서 SPRING이 제공하는 빈 객체를

이용해야 할 때가 있다. 이런 경우에는 SPRING이 제공하는 WebApplicationContextUtils Class를 이용해서 WebApplicationContext에 접근할 수 있다.

org.springframework.web.context.support.WebApplicationContextUtils Class는 DispatcherServlet이나 ContextLoaderListener가 생성한

WebApplicationContext에 접근할 수 있는 static 메서드를 제공하고 있으며, 이들 메서드는 다음과 같다.

- WebApplicationContext getWebApplicationContext(ServletContext sc)

: ContextLoaderListener가 생성한 루트 WebApplicationContext를 구한다.

- WebApplicationContext getWebApplicationContext(ServletContext sc, String attrName)

: ServletContext에 지정한 속성 이름(attrName)으로 저장된 WebApplicationContext를 구한다. 주로 DispatcherServlet이 생성한 WebApplicationContext에

접근할 때 사용된다.

루트 WebApplicationContext에 접근하는 Code는 다음과 같다.

WebApplicationContext rootContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext());

Object commonLogger = rootContext.getBean("commonLogger");

DispatcherServlet이 생성한 WebApplicationContext에 접근할 때에는 속성 이름을 사용해야 하는데, 이때 속성 이름을 생성하는 규칙은 다음과 같다.

org.springframework.web.servlet.FrameworkServlet.CONTEXT.Servlet이름

예를 들어, web.xml 파일에 DispatcherServlet을 아래 Code와 같이 등록했다고 해보자.

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

이 경우, DispatcherServlet이 생성한 WebApplicationContext는 다음과 같은 이름을 사용해서 ServletContext의 속성으로 저장된다.

org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher

따라서, 아래의 Code를 이용해서 DispatcherServlet이 생성한 WebApplicationContext에 접근할 수 있게 된다.

WebApplicationContext dispatcherContext = WebApplicationContextUtils.getWebApplicationContext(getServletContext(),

"org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher");

Object helloController = dispatcherContext.getBean("helloController");

WebApplicationContextUtils.getWebApplicationContext() 메서드는 첫 번째 파라미터로 ServletContext를 필요로 하기 때문에, JSP나 Servlet,

커스텀 태그등 ServletContext에 접근할 수 있는 Code에서 SPRING Container에 등록된 빈을 필요로 할 때 주로 사용된다.

===============================================================================

16. DelegatingFilterProxy를 이용한 Servlet 필터 등록

: Servlet 필터를 SPRING 빈으로 등록할 때 사용되는 Class가 바로 DelegatingFilterProxy이다. DelegatingFilterProxy Class는 SPRING Container에

빈으로 등록된 Servlet 필터에 필터 처리를 위임한다. DelegatingFilterProxy를 사용하기 위해서는 다음과 같이 web.xml 파일에 DelegatingFilterProxy를

Servlet 필터로 등록해 주면 된다.

<filter>

<filter-name>profileFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

<init-param>

<param-name>targetBeanName</param-name>

<param-value>webProfileBean</param-value>

</init-param>

<init-param>

<param-name>contextAttribute</param-name>

<param-value>

org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher

</param-value>

</init-param>

</filter>

위 Code의 경우 profileFilter는 그 요청을 targetBeanName 초기화 파라미터로 지정한 빈에 위임한다. targetBeanName 초기화 파라미터를

지정하지 않으면, <filter-name>에 지정한 이름을 빈 이름으로 사용한다.

지정한 이름을 가진 빈이 DispatcherServlet이 생성한 WebApplicationContext에 등록되어 있다면, contextAttribute 초기화 파라미터를

이용해서 DispatcherServlet이 사용하는 속성 이름을 지정한다. contextAttribute 초기화 파라미터를 지정하지 않으면

루트 WebApplicationContext에 등록된 빈을 사용한다.

DelegatingFilterProxy는 기본적으로 Filter.init() 메서드와 Filter.destroy() 메서드에 대한 호출은 위임하지 않는데, 그 이유는 SPRING Container가

제공하는 라이프 사이클이 있기 때문이다. 만약 Filter의 init() 메서드와 destroy() 메서드에 대해서도 위임을 하고 싶다면 다음과 같이

targetFilterLifecycle 초기화 파라미터의 값을 true로 지정해주면 된다.

<filter>

<filter-name>profileFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

<init-param>

<param-name>targetBeanName</param-name>

<param-value>webProfileBean</param-value>

</init-param>

<init-param>

<param-name>contextAttribute</param-name>

<param-value>

org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher

</param-value>

</init-param>

<init-param>

<param-name>targetFilterLifecycle</param-name>

<param-value>true</param-value>

</init-param>

</filter>

[ProfileFilter.java]

package madvirus.spring.chap06.filter;

import java.io.IOException;

import javax.servlet.Filter;

import javax.servlet.FilterChain;

import javax.servlet.FilterConfig;

import javax.servlet.ServletException;

import javax.servlet.ServletRequest;

import javax.servlet.ServletResponse;

import madvirus.spring.chap06.common.CommonLogger;

public class ProfileFilter implements Filter {

private CommonLogger commonLogger;

@Override

public void destroy() {

commonLogger.log("ProfileFilter.destroy() 호출됨");

}

@Override

public void doFilter(ServletRequest request, ServletResponse response,

FilterChain filterChain) throws IOException, ServletException {

long start = System.currentTimeMillis();

try {

filterChain.doFilter(request, response);

} finally {

long finish = System.currentTimeMillis();

commonLogger.log("실행 시간 : " + (finish - start));

}

}

@Override

public void init(FilterConfig arg0) throws ServletException {

commonLogger.log("ProfileFilter.init() 호출됨");

}

public void setCommonLogger(CommonLogger commonLogger) {

this.commonLogger = commonLogger;

}

}

[dispatcher-servlet.xml]

...

<bean id="webProfileBean" class="madvirus.spring.chap06.filter.ProfileFilter"

p:commonLogger-ref="commonLogger" />

...

===============================================================================

17. @RequestBody Annotation과 @ResponseBody Annotation 사용

: WEB 서비스와 REST 방식이 시스템을 구성하는 주요 요소로 자리 잡으면서 WEB 시스템간에 XML이나 JSON등의 형식으로 데이터를 주고 받는

경우가 증가하고 있다. 이에 따라 SPRING MVC도 클라이언트에서 전송한 XML 데이터나 JSON 또는 기타 테이터를 컨트롤러에서 DOM 객체나

Java 객체로 변환해서 받을 수 있는 기능을 제공하고 있으며, 비슷하게 Java 객체를 XML이나 JSON 또는 기타 형식으로 변환해서 전송할 수

있는 기능을 제공하고 있다.

@RequestBody Annotation과 @ResponseBody Annotation은 각각 HTTP 요청 몸체를 Java 객체로 변환하고 Java 객체를 HTTP 응답 몸체로

변환해주는 데 사용된다.

@RequestBody Annotation을 이용하면 HTTP 요청 몸체를 Java 객체로 전달받을 수 있다. @ResponseBody Annotation을 이용하면 Java 객체를

HTTP 응답 몸체로 전송할 수 있다. 아래 Code는 @RequestBody Annotation과 @ResponseBody Annotation의 간다한 사용 예를 보여주고 있다.

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

import org.springframework.util.MultiValueMap;

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

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

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

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

@Controller

@RequestMapping("/test/SimpleTest.do")

public class SimpleConverterController {

@RequestMapping(value = "/test/simpleTest.do", method = RequestMethod.GET)

public String simpleTestForm() {

return "test/simpleTestForm";

}

@RequestMapping(value = "/test/simpleTest.do", method = RequestMethod.POST)

@ResponseBody

public String simpleTest(@RequestBody String body) {

return body;

}

}

라인 21에서 @RequestBody Annotation은 @RequestMapping에 의해 POST 방식으로 전송된 HTTP 요청 데이터를 String 타입의

body 파라미터로 전달한다. 그리고 @ResponseBody Annotation이 @RequestMapping 메서드에서 적용되면 해당 메서드의 리턴

값을 HTTP 응답 데이터로 사용한다. 위의 Code의 경우 simpleTest() 메서드의 리턴 값이 String 타입이므로 String 데이터를 HTTP

응답 데이터로 전송한다.

위의 Code에서 SimpleConverterController에 GET 방식으로 요청이 전달되면 simpleTestForm 뷰가 사용되는데, 이 뷰가 아래와 같은

폼을 생성한다고 해 보자.

<form method="POST">

이름 : <input type="text" name="name"/><br/>

나이 : <input type="text" name="age"/>

<input type="submit"/>

</form>

이 때 HTTP 몸체로 전송되는 데이터는 다음과 같은 형식을 취한다.

name=%C3%D6%B9%FC%B1%D5&age=34

SimpleConverterController 콘트롤러는 POST 방식으로 요청이 들어오는 경우 @RequestBody Annotation을 이용해서 HTTP 요청 몸체

데이터를 HTTP 요청 몸체 데이터를 body 파라미터로 전달받고 있으며, body 파라미터를 그대로 결과 값으로 리턴한다. 그런데,

@ResponseBody Annotation이 적용되어 있으므로 결과적으로 HTTP 요청 몸체 데이터가 HTTP 응답 몸체로 전송된다. 실제로 실행

결과를 보면 HTTP 응답 결과가 다음과 같은 것을 확인할 수 있다.

name=%C3%D6%B9%FC%B1%D5&age=34

SPRING MVC는 HttpMessageConverter를 이용해서 Java 객체와 HTTP 요청/응답 몸체 사이의 변환을 처리하는데, @ResponseBody

Annotation 적용 메서드의 리턴 타입이 String인 경우 HTTP 응답 데이터의 컨텐츠 타입은 "text/plain;charset=ISO-8859-1"이 된다.

<Example015>

[SimpleConverterController.java]

package madvirus.spring.chap06.controller;

import org.springframework.stereotype.Controller;

import org.springframework.util.MultiValueMap;

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

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

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

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

@Controller

public class SimpleConverterController {

@RequestMapping(value = "/test/simpleTest.do", method = RequestMethod.GET)

public String simpleTestForm() {

return "test/simpleTestForm";

}

@RequestMapping(value = "/test/simpleTest.do", method = RequestMethod.POST)

@ResponseBody

public String simpleTest(@RequestBody String body) {

return body;

}

@RequestMapping(value = "/test/simpleTest1.do", method = RequestMethod.GET)

public String simpleTestForm1() {

return "test/simpleTestForm";

}

@RequestMapping(value = "/test/simpleTest1.do", method = RequestMethod.POST)

@ResponseBody

public byte[] simpleTest1(@RequestBody byte[] body) {

return body;

}

@RequestMapping(value = "/test/simpleTest2.do", method = RequestMethod.GET)

public String simpleTestForm2() {

return "test/simpleTestForm";

}

@RequestMapping(value = "/test/simpleTest2.do", method = RequestMethod.POST)

@ResponseBody

public String simpleTest2(@RequestBody MultiValueMap<String, String> body) {

return body.toString();

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.SimpleConverterController" />

...

[/WEB-INF/view/test/simpleTestForm.jsp]

<%@ page contentType="text/html; charset=EUC-KR"%>

<%@ taglib uri="http://www.springframework.org/tags" prefix="spring"%>

<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>단순 테스트</title>

</head>

<body>

<form method="POST">

이름: <input type="text" name="name" /> <br/>

나이: <input type="text" name="age" />

<input type="submit" />

</form>

</body>

</html>

---------------------------------------------------------------------------------------------------------------------------------------------------------------

17.1 HttpMessageConverter를 이용한 변환 처리

: AnnotationMethodHandlerAdapter Class는 @RequestBody Annotation이 적용된 파라미터나 @ResponseBody Annotation이

적용된 메서드에 대해 HttpMessageConverter를 사용해서 변환을 처리한다. 주요 HttpMessageConverter 구현 Class는 다음과 같다.

[[구현 Class]] [[설 명]]

================================================================================

ByteArrayHttpMessageConverter(\*) HTTP Message와 byte 배열 사이의 변환을 처리한다.

컨텐츠 타입은 application/octet-stream 이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

StringHttpMessageConverter(\*) HTTP Message와 String 사이의 변환을 처리한다.

컨텐츠 타입은 text/plain;charset=ISO-8859-1이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

FormHttpMessageConverter(\*) HTML 폼 데이터를 MultiValueMap으로 전달받을 때 사용된다.

지원하는 컨텐츠 타입은 application/x-www-form-urlencoded이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

SourceHttpMessageConverter(\*) HTTP Message와 javax.xml.transform.Source 사이의 변환을 처리한다.

컨텐츠 타입은 application.xml 또는 text/xml이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

MarshallingHttpMessageConverter SPRING의 marshaller와 Unmarshaller를 이용해서 XML, HTTP Message와 객체

사이의 변환을 처리한다. 컨텐츠 타입은 application.xml 또는 text/xml이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

MappingJacksonHttpMessageConverter Jackson Library를 이용해서, JSON HTTP Message와 객체 사이의 변환을

처리한다. 컨텐츠 타입은 application/json 이다.

----------------------------------------------------------------------------------------------------------------------------------------------------------------

AnnotationMethodHandlerAdapter는 위에서 (\*)로 표시된 HttpMessageConverter 구현 Class를 기본적으로 사용한다. 만약 기본 구현

Class가 아닌 다른 구현 Class를 사용하려면 다음과 같이 AnnotationMethodHandlerAdapter에 명시적으로 구현 Class 목록을

지정해 주어야 한다.

<bean class="org.springframework.web.servlet.annotation.AnnotationMethodHandlerAdapter">

<property name="cacheSeconds" value="0"/>

<property name="alwaysUseFullPath" value="true"/>

<property name="webBindingInitializer">

<bean class="madvirus.spring.chap06.binder.CustomWebBindingInitializer"/>

</property>

<property name="messageConverters">

<list>

<ref bean="byteArrayHttpMessageConverter"/>

<ref bean="marshallingHttpMessageConverter"/>

</list>

</property>

</bean>

<bean id="byteArrayHttpMessageConverter" class="org.springframework.http.converter.ByteArrayHttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter" class="org.springframework.http.converter.MarshallingHttpMessageConverter">

<porperty name="marshaller" ref="jaxb2Marshaller"/>

<property name="unmarshaller" ref="jaxb2Marshaller"/>

</bean>

예를 들어, 다음과 같이 JAXB2의 Annotation이 적용된 Java 객체가 있다고 하자.

public class WriteArticleServiceImpl{

@XmlAccessorType(XmlAccessType.FIELD)

@XmlRootElement(name="message-list")

public class GuestMessageList{

@XmlElement(name="message")

private List<GuestMessage> messages;

...

}

@XmlAccessorType(XmlAccessType.FIELD)

@XmlType(name="", propOrder={"id","message","creationTime"})

public class GuestMessage{

private Integer id;

private String message;

private Date creationTime;

...

}

컨트롤러에서는 아래 Code와 같이 @ResponseBody Annotation을 이용해서 GuestMessageList를 리턴하도록 구현할 수 있을 것이다.

@Controller

public class GuestMessageController {

@RequestMapping(value = "/guestmessage/xml.do", method = RequestMethod.GET)

@ResponseBody

public GuestMessageList listXml() {

GuestMessageList list=...;

return list;

}

...

}

JAXB2를 이용해서 Java 객체를 XML 응답으로 전송하려면 MarshallingHttpMessageConverter의 marshaller와 unmarshaller로 SPRING

OXM Module의 Jaxb2Marshaller를 사용하면 된다. 아래 Code는 설정 예이다.

<bean id="marshallingHttpMessageConverter" class="org.springframework.http.converter.MarshallingHttpMessageConverter">

<porperty name="marshaller" ref="jaxb2Marshaller"/>

<property name="unmarshaller" ref="jaxb2Marshaller"/>

</bean>

<bean id="jaxb2marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">

<property name="classesToBeBound">

<value>madvirus.spring.chap06.model.GuestMessageList</value>

</property>

</bean>

이제 SPRING MVC는 MarshallingHttpMessageConverter를 이용해서 GuestMessageList 객체를 XML로 변환할 수 있게 된다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

17.2 Content-Type과 Accept 헤더 기반의 변환 처리

: AnnotationMethodHandlerAdapter가 HttpMessageConverter를 이용해서 요청 몸체 데이터를 @RequestBody Annotation이 적용된

Java 객체로 변환할 때에는 HTTP 요청 헤더의 Content-Type 헤더에 명시된 미디어 타입(MIME)을 지원하는 HttpMessageConverter

구현체를 선택한다. 예를 들어, 요청 미디어 타입이 application/xml이고 @RequestBody Annotation이 적용된 파라미터가 Source

타입인 경우 SourceHttpMessageConverter가 선택된다.

비슷하게 @ResponseBody Annotation을 이용해서 리턴한 객체를 HTTP 응답 몸체로 변환할 때는 HTTP 요청 헤더의 Accept 헤더에

명시된 미디어 타입을 지원하는 HttpMessageConverter 구현체를 선택한다. 예를 들어, Accept 헤더에 명시된 값이 application/json이고

@ResponseBody Annotation이 적용된 메서드의 리턴 타입이 Java 객체인 경우 MappingJacksonHttpMessageConverter가 선택 된다.

예를 들어, Java스크립트에서 다음과 같은 Code를 이용해서 Accept 헤더의 값으로 "application/json"을 지정했다고 하자.

<script type="javascript">

...

xmlhttp.open("GET","json.do",true);

xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xmlhttp.send(null);

...

</script>

이 경우 다음과 같이 @ResponseBody Annotation이 적용된 메서드의 리턴 객체를 변환할 때 MappingJacksonHttpMessageConverter를

사용하게 된다.

@RequestMapping(value = "/guestmessage/json.do", method = RequestMethod.GET, headers="accept=application/json")

@ResponseBody

public GuestMessageList listJson(){

...

}

<Example016>

[GuestMessageController.java]

package madvirus.spring.chap06.controller;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

import madvirus.spring.chap06.model.GuestMessage;

import madvirus.spring.chap06.model.GuestMessageList;

import org.springframework.stereotype.Controller;

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

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

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

@Controller

public class GuestMessageController {

@RequestMapping(value = "/guestmessage/xml.do", method = RequestMethod.GET)

@ResponseBody

public GuestMessageList listXml() {

return getMessageList();

}

@RequestMapping(value = "/guestmessage/jsontest.do")

public String jsonForm() {

return "guestmessage/jsonform";

}

@RequestMapping(value = "/guestmessage/json.do", method = RequestMethod.GET, headers = "accept=application/json")

@ResponseBody

public GuestMessageList listJson() {

return getMessageList();

}

private GuestMessageList getMessageList() {

GuestMessage message1 = new GuestMessage(1, "메시지", new Date());

GuestMessage message2 = new GuestMessage(2, "메시지2", new Date());

GuestMessageList messageList = new GuestMessageList();

List<GuestMessage> messages = new ArrayList<GuestMessage>();

messages.add(message1);

messages.add(message2);

messageList.setMessages(messages);

return messageList;

}

}

[GuestMessage.java]

package madvirus.spring.chap06.model;

import java.util.Date;

import javax.xml.bind.annotation.XmlAccessType;

import javax.xml.bind.annotation.XmlAccessorType;

import javax.xml.bind.annotation.XmlType;

@XmlAccessorType(XmlAccessType.FIELD)

@XmlType(name = "", propOrder = { "id", "message", "creationTime" })

public class GuestMessage {

private Integer id;

private String message;

private Date creationTime;

public GuestMessage() {

}

public GuestMessage(Integer id, String message, Date creationTime) {

this.id = id;

this.message = message;

this.creationTime = creationTime;

}

public Integer getId() {

return id;

}

public void setId(Integer id) {

this.id = id;

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

public Date getCreationTime() {

return creationTime;

}

public void setCreationTime(Date creationTime) {

this.creationTime = creationTime;

}

@Override

public String toString() {

return "GuestMessage [creationTime=" + creationTime + ", id=" + id

+ ", message=" + message + "]";

}

}

[GuestMessageList.java]

package madvirus.spring.chap06.model;

import java.util.List;

import javax.xml.bind.annotation.XmlAccessType;

import javax.xml.bind.annotation.XmlAccessorType;

import javax.xml.bind.annotation.XmlElement;

import javax.xml.bind.annotation.XmlRootElement;

@XmlAccessorType(XmlAccessType.FIELD)

@XmlRootElement(name="message-list")

public class GuestMessageList {

@XmlElement(name = "message")

private List<GuestMessage> messages;

public List<GuestMessage> getMessages() {

return messages;

}

public void setMessages(List<GuestMessage> messages) {

this.messages = messages;

}

}

[dispatcher-servlet.xml]

...

<bean class="madvirus.spring.chap06.controller.GuestMessageController" />

...

<bean

class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

<property name="cacheSeconds" value="0" />

<property name="alwaysUseFullPath" value="true" />

<property name="webBindingInitializer">

<bean class="madvirus.spring.chap06.binder.CustomWebBindingInitializer" />

</property>

<property name="messageConverters">

<list>

<ref bean="byteArrayHttpMessageConverter" />

<ref bean="stringHttpMessageConverter" />

<ref bean="formHttpMessageConverter" />

<ref bean="sourceHttpMessageConverter" />

<ref bean="marshallingHttpMessageConverter" />

<ref bean="jsonHttpMessageConverter" />

</list>

</property>

</bean>

<bean id="byteArrayHttpMessageConverter"

class="org.springframework.http.converter.ByteArrayHttpMessageConverter" />

<bean id="stringHttpMessageConverter"

class="org.springframework.http.converter.StringHttpMessageConverter" />

<bean id="formHttpMessageConverter"

class="org.springframework.http.converter.FormHttpMessageConverter" />

<bean id="sourceHttpMessageConverter"

class="org.springframework.http.converter.xml.SourceHttpMessageConverter" />

<bean id="marshallingHttpMessageConverter"

class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter">

<property name="marshaller" ref="jaxb2Marshaller" />

<property name="unmarshaller" ref="jaxb2Marshaller" />

</bean>

<bean id="jaxb2Marshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">

<property name="classesToBeBound">

<list>

<value>madvirus.spring.chap06.model.GuestMessageList</value>

</list>

</property>

</bean>

<bean id="jsonHttpMessageConverter" class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter" />

...

[/WEB-INF/view/guestmessage/jsonform.jsp]

<%@ page language="java" contentType="text/html; charset=EUC-KR"%>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">

<title>JSON 테스트</title>

<script type="text/javascript">

var xmlhttp;

$(document).ready(function() {

if (window.XMLHttpRequest) {

xmlhttp=new XMLHttpRequest();

} else {

xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");

}

xmlhttp.onreadystatechange = function() {

if(xmlhttp.readyState == 4) {

document.getElementById('loadedJson').innerHTML=xmlhttp.responseText;

}

}

xmlhttp.open("GET","json.do",true);

xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xmlhttp.setRequestHeader("Accept", "application/json");

xmlhttp.send(null);

});

</script>

</head>

<body>

<div id="loadedJson"></div>

</body>

</html>

===============================================================================

18. HandlerMapping, HandlerAdapter 컨트롤러 구현체

: DispatcherServlet은 HandlerMapping을 이용해서 컨트롤러 구현체를 찾는다. 본 장에서는 지금까지 살펴본 컨트롤러 구현체는 @Controller

Annotation이 적용된 Class였는데, 이 뿐만 아니라 Controller Interface를 구현한 Class나 HttpRequestHandler Interface를 구현한

Class도 컨트롤러 구현체가 될 수 있다.

다양한 타입을 이용해서 컨트롤러 구현체를 구현할 수 있는데 각 타입의 컨트롤러 구현체를 알맞게 호출하기 위해 사용되는 것이 HandlerAdapter이다.

본 장에서 살펴본 AnnotationMethodHandlerAdapter는 @Controller Class의 @RequestMapping 메서드를 처리해주는 HandlerAdapter이다.

SPRING MVC는 AnnotationMethodHandlerAdapter 외에도 Controller Interface나 HttpRequestHandler Interface에 대한 HandlerAdapter를

제공하고 있다.

본 절에서는 주요 HandlerMapping Class와 HandlerAdapter Class 그리고 Controller Interface의 구현 Class 중 유용하게 사용되는

Class에 대해서 살펴보도록 하겠다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

18.1 HandlerMapping의 주요 구현 Class

: DispatcherServlet은 HandlerMapping을 이용해서 클라이언트가 요청한 URL과 매칭되는 컨트롤러 객체를 찾는다. SPRING이 제공하는

HandlerMapping Class는 다음과 같다.

[[클 래 스]] [[설 명]]

=================================================================================

DefaultAnnotationHandlerMapping @Controller Class에 대한 Mapping을 처리한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

SimpleUrlHandlerMapping URL과 빈 객체 사이의 Mapping을 설정한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

BeanNameUrlHandlerMapping URL과 일치하는 이름을 갖는 빈을 컨트롤러로 사용한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

ControllerBeanNameHandlerMapping URL과 매칭되는 빈 이름을 갖는 빈을 컨트롤러로 사용한다. 슬래시와 접두어,

접미사 등의 설정을 추가로 지원한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

ControllerClassNameHandlerMapping URL과 매칭되는 Class 이름을 갖는 빈을 컨트롤러로 사용한다.

------------------------------------------------------------------------------------------------------------------------------------------------------------------

DefaultAnnotationHandlerMapping Class는 @Controller Annotation을 이용해서 구현된 컨트롤러 구현체에 대한 Mapping을 처리할 때 사용되며,

나머지 HandlerMapping 구현 Class는 주로 Controller Interface를 상속 받는 컨트롤러 구현체와 URL의 Mapping을 처리할 때 사용된다.

SPRING 3버전부터 @Controller Annotation을 이용해서 컨트롤러를 구현하는 것을 권장하기 때문에 DefaultAnnotationHandlerMapping을

제외한 나머지 HandlerMapping을 사용해야 할 경우는 흔치 않다. 하지만, 경우에 따라 SPRING이 제공하는 Controller Interface 기반의

컨트롤러인 UrlFilenameViewContoller Class를 사용해야 하는 경우가 발생하는데, 이런 경우에는 DefaultAnnotationHandlerMapping 외에

나머지 HandlerMapping 구현 Class를 사용해야 할 경우가 발생한다.

DefaultAnnotationHandlerMapping를 제외한 나머지 구현 Class 중에서 자주 사용되는 HandlerMapping Class는 SimpleUrlHandlerMapping

Class와 BeanNameUrlHandlerMapping Class이므로, 본 절에서는 이 두 HandlerMapping Class의 사용방법에 대해서 살펴보도록 하겠다.

(1) SimpleUrlHandlerMapping을 이용한 컨트롤러 Mapping

: org.springframework.web.servlet.handler.SimpleUrlHandlerMapping은 Controller Interface를 구현한 컨트롤러 빈을 Mapping할 때 주로

사용되는 HandlerMapping으로서, 패턴 매칭을 이용해서 다양한 URL 경로를 컨트롤러에 Mapping 시켜준다.

SimpleUrlHandlerMapping은 다음 Code처럼 mappings 프로퍼티를 이용하여 패턴과 컨트롤러 사이의 Mapping을 지정한다.

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

<property name="mappings">

<props>

<prop key="/content/\*\*/\*.htm">contentController</prop>

<prop key="/greeting/hello.htm">helloController</prop>

</props>

</property>

</bean>

<bean name="contentController" .../>

<bean name="helloController" .../>

SimpleUrlHandlerMapping의 "mappings" 프로퍼티는 java.util.Properties 타입이다. "mappings" 프로퍼티의 값에 전달되는 각 <prop>의

key는 요청 URL과 매칭될 Ant 경로 패턴을 입력하며, <prop>의 값에는 Mapping 될 컨트롤러의 이름을 입력한다.

alwaysUseFullPath 프로퍼티의 값을 true로 지정하면 컨텍스트 경로를 제외한 나머지 전체 경로를 이용하여 Mapping되는 컨트롤러를

검색할 수 있다. alwaysUseFullPath 프로퍼티의 값이 false인 경우 Servlet 경로를 제외한 나머지 경로를 이용해서 Mapping되는 컨트롤러를

검색한다. 기본 값은 false이다.

SimpleUrlHandlerMapping은 다양한 경로 패턴을 손쉽게 설정할 수 있기 때문에, Controller Interface를 구현한 컨트롤러에 대한 WEB 요청

Mapping을 설정할 때 주로 사용 된다.

(2) BeanNameUrlHandlerMapping을 이용한 컨트롤러 Mapping

: org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping은 URL과 매칭되는 이름을 갖는 빈을 컨트롤러로 사용하는

HandlerMapping이다. alwaysUseFullPath 프로퍼티의 값을 true로 지정하면 컨텍스트 경로를 제외한 나머지 전체 경로를 이용하여 Mapping되는

컨트롤러를 검색할 수 있다. alwaysUseFullPath 프로퍼티의 값이 false인 경우 Servlet 경로를 제외한 나머지 경로를 이용해서 Mapping 되는

컨트롤러를 겁색한다. 기본 값은 false이다.

BeanNameUrlHandlerMapping은 빈의 이름은 Ant 경로 패턴으로 사용하여, 요청 URL이 패턴에 매칭되는 지의 여부를 검사한다. 그런 뒤,

매치된다면 해당 빈을 컨트롤러로 사용하게 된다. 예를 들어, 다음과 같이 BeanNameUrlHandlerMapping의 alwaysUseFullPath 프로퍼티 값을

true로 지정했고, 두 개의 컨트롤러를 빈으로 설정했다고 하자.

<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

</bean>

<bean name="/content/\*\*/\*.htm" class="..."/>

<bean name="/greeting/hello.htm" class="..."/>

이 경우, 요청 URL을 처리하는 컨트롤러 빈은 아래와 같이 Mapping된다.

/content/10/orm.htm -------------> /content/\*\*/\*.htm빈

|

/content/1/topic/list.htm -------

/greeting/hello.htm -------------> /greeting/hello.htm 빈

빈의 이름에 '\*'이 포함되어 있지 않다면 위에서 '/greeting/hello.htm' 빈 같이 정확하게 빈의 이름과 동일한 요청 URL인 경우에만 컨트롤러가

클라이언트 요청을 처리하게 된다. 하지만, 빈 이름에 '\*'가 포함되어 있다면 '/content/\*\*/\*.htm' 빈과 같이 빈의 이름을 Ant 경로 패턴으로

사용하여 요청 URL이 해당 패턴에 일치하면 클라이언트 요청을 처리하게 된다.

(3) DispatcherServlet의 기본 HandlerMapping 구현 Class와 HandlerMapping 적용 순서

: DispatcherServlet은 SPRING 설정 파일에 HandlerMapping이 등록되어 있지 않을 경우 다음의 두 HandlerMapping 구현 Class를 기본

HandlerMapping으로 사용한다.

- BeanNameUrlHandlerMapping

- DefaultAnnotationHandlerMapping

기본으로 등록된 HandlerMapping의 설정을 변경해주고 싶다거나 기본 HandlerMapping이 아닌 다른 HandlerMapping을 사용하고 싶은 경우

설정 파일에 직접 HandlerMapping을 등록해 주어야 한다. 예를 들어, DefaultAnnotationHandlerMapping Class와 SimpleUrlHandlerMapping

Class를 동시에 사용하고 싶다면 다음과 같이 설정해 주면 된다.

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

<property name="mappings">

<props>

<prop key="/content/\*\*/\*.htm">contentController</prop>

<prop key="/greeting/hello.htm">helloController</prop>

</props>

</property>

</bean>

<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" p:alwaysUseFullPath="true"/>

한 개 이상의 HandlerMapping이 등록되어 있을 경우, order 프로퍼티를 이용하여 사용 순서를 결정한다. 예를 들어, 다음과 같이 두 개의

HandlerMapping을 정의했다고 하자.

<bean id="handlerMapping1" class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">

<property name="order" value="2"/>

<property name="alwaysUseFullPath" value="true"/>

</bean>

<bean id="handlerMapping2" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="order" value="1"/>

<property name="alwaysUseFullPath" value="true"/>

<property name="mappings">

<props>

<prop key="/content/\*\*/\*.htm">contentController</prop>

<prop key="/greeting/hello.htm">helloController</prop>

</props>

</property>

</bean>

DispatcherServlet은 order 프로퍼티의 값이 낮은 HandlerMapping에 Mapping 되는 컨트롤러가 있는 지 먼저 의뢰한다. 위 Code의 경우는

'handlerMapping1'에게 먼저 의뢰하게 된다. 만약 'handlerMapping1'이 Mapping되는 컨트롤러를 발견하지 못했다면, 다음으로 낮은

order 프로퍼티 값을 갖는 'handlerMapping2'에게 의뢰하게 된다.

예를 들어, 클라이언트의 요청 URI가 "/content/read.do"라고 하자. 이 때 DispatcherServlet은 먼저 'handlerMapping1"에 의뢰하게

되는데, 'handlerMapping1'은 "/contnet/read.do"와 Mapping되는 컨트롤러 설정을 갖지 않고 있다. 그 다음 순서인 'handlerMapping2"를

이용하여 "/content/read.do" 와 Mapping되는 컨트롤러를 검색하게 된다.

"order" 프로퍼티를 입력하지 않은 경우, 설정 파일에 입력한 순서에 따라 Mapping되는 컨트롤러를 검색하게 된다.

일단 설정 파일에 HandlerMapping 설정이 추가되면 기본 설정 값은 사용되지 않으므로, DefaultAnnotationHandlerMapping이 아닌

다른 HandlerMapping을 설정해 주었다면 DefaultAnnotationHandlerMapping을 추가로 설정해서 @Controller Annotation이 적용된

Class가 컨트롤러로 적용될 수 있도록 해 주어야 한다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

18.2 HandlerAdapter의 주요 구현 Class

: DispatcherServlet은 HandlerMapping을 통해서 URL과 매칭되는 컨트롤러 객체를 찾은 뒤에는 컨트롤러 객체를 실행할 때 사용될

HandlerAdapter를 선택한다. 각 컨트롤러 객체 마다 타입이나 적용된 Annotation이 다를 수 있는데, HandlerAdapter는 각 컨트러의

실행 결과를 ModelAndView로 변환해주는 기능을 제공한다.

SPRING이 제공하는 주요 HandlerAdapter 구현 Class는 다음과 같다.

[[Class]] [[설 명]]

===============================================================================

HttpRequestHandlerAdapter HttpRequestHandler Interface를 구현한 Class를 처리한다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

SimpleControllerHandlerAdapter Controller Interface를 구현한 Class를 처리한다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

AnnotationMethodHandlerAdapter @Controller Annotation과 @RequestMapping Annotation이 적용된 Class를 처리한다.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

DispatcherServlet은 설정 파일에 한 개 이상의 HandlerAdapter 빈이 존재할 경우 해당 HandlerAdapter를 사용한다. 만약 설정 파일에

HandlerAdapter 빈이 존재하지 않는다면 위에 표시한 세 개의 HandlerAdapter를 기본으로 사용한다.

---------------------------------------------------------------------------------------------------------------------------------------------------------------

18.3 ParameterizableViewController와 UrlFilenameViewController를 이용한 정적 뷰 Mapping

: WEB Application을 개발하다 보면, 별도의 로직을 처리할 필요 없이 뷰 페이지를 출력하고 싶을 때가 있다. 예를 들어, 회사 소개 페이지를

생각해 보자. 회사 소개 페이지는 대부분 이미지 내지 플래시 등의 UI 요소가 강조되고 DB 연동과 같은 처리가 필요하지 않다.

이렇게 DB 연동과 같은 처리가 필요 없고 뷰 페이지를 바로 보여 주어야 하는 경우에는 ParameterizableViewController나 UrlFilenameViewController

를 사용하면 된다. 이 두 Class는 Controller Interface를 구현한 컨트롤러로서 정적인 뷰를 제공하고 싶은 경우에 사용된다.

ParameterizableViewControllerClass는 요청 URL과 상관없이 지정한 뷰를 보여줄 때 사용된다. ParameterizableViewController는

다음과 같이 viewName 프로퍼티를 이용하여 사용될 뷰 이름을 설정한다.

<bean id="comingSoonController" class="org.springframework.web.servlet.mvc.ParameterizableViewController" p:viewName="comingSoon"/>

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

<property name="mappings">

<props>

<prop key="/content/help.htm">contentController</prop>

<prop key="/bbs/help.htm">helloController</prop>

</props>

</property>

</bean>

위 Code는 "/content/help.htm" 요청이나 "/bbs/help.htm" 요청이 들어오면 comingSoonController가 요청을 처리하도록 설정했다.

comingSoonController는 별도의 로직을 수행하지 않고 단순히 "comingSoon" 뷰를 선택한다. 즉, 컨트러가 다음과 같이

ModelAndView를 리턴하는 것과 같다.

ModelAndView mav = new ModelAndView();

mav.setViewName("comingSoon");

return mav;

UrlFilenameViewController는 URL로부터 뷰 이름을 추출한다. 예를 들어, 다음과 같이 설정되어 있다고 하자.

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="alwaysUseFullPath" value="true"/>

<property name="mappings">

<props>

<prop key="/company/vision.htm">contentController</prop>

<prop key="/company/ceo.htm">helloController</prop>

</props>

</property>

</bean>

<bean id="urlController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

위 설정에서는 "/company/vision.htm" 요청과 "/company/ceo.htm" 요청이 들어 올 경우 urlController에 전달되는데, 이때

UrlFilenameViewController는 요청 URL에 따라서 다음과 같이 확장자와 앞의 '/'를 제외한 값을 뷰 이름으로 리턴한다.

- 요청 URL /company/vision.htm ===> 뷰 이름 company/vision

- 요청 URL /company/ceo.htm ===> 뷰 이름 company/ceo

URL로 부터 추출된 값의 앞뒤로 특정 값을 추가하고 싶다면, 'prefix' 프로퍼티와 'suffix' 프로퍼티를 이용하여 추가할 값을 지정하면 된다.

<bean id="urlController" class="org.springframework.web.servlet.mvc.UrlFilenameViewController" p:prefix="static"/>

위와 같이 prefix 프로퍼티를 이용하여 접두어를 적용하였다면, '/company/vision.htm' 요청 URL의 경우 'static/company/vision'을

뷰 이름으로 사용하게 된다.