



create table board3(

seq number primary key,

title varchar2(100),

writer varchar2(100),

contents varchar2(2000),

regdate DATE,

hitcount number

);

create sequence board\_seq;

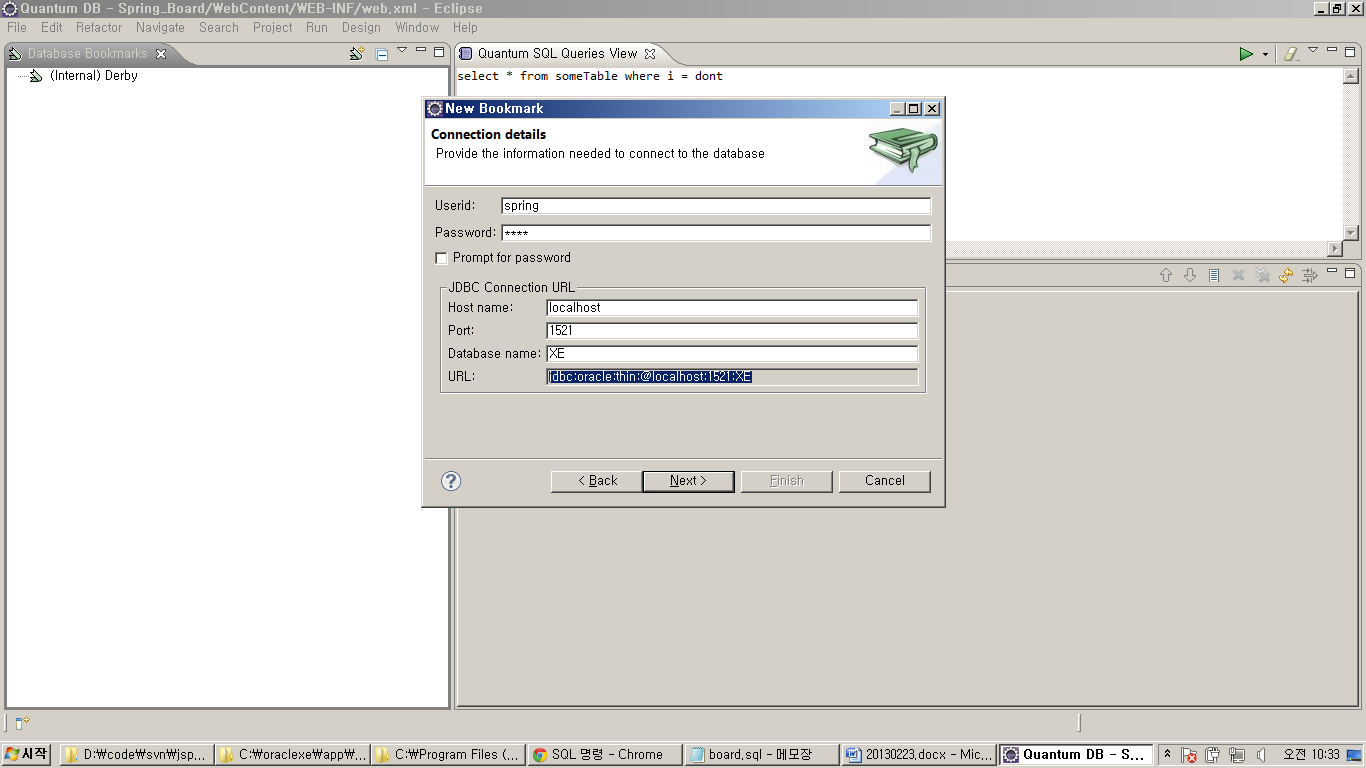
create table admin(

mem\_id varchar2(50),

mem\_name varchar2(50)

);

insert into admin values('aaa', 'bbb');



jdbc:oracle:thin:@localhost:1521:XE

* **DispatcherServlet 설정** 
  + 다른 Servlet처럼 web.xml에 등록
  + 기본적으로 “<servlet-name>-servlet.xml” 파일에서 어플리케이션 컨텍스트 정보를 로드
  + <url-pattern>은 DispatcherServlet이 처리하는 URL 매핑 패턴을 정의

<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>

</servlet-mapping>

하나이상의xml정의

* **한 개 이상의 설정파일을 사용하거나**
* **다른 이름의 설정파일을 사용하고 싶은 경우**

<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/dispatcher.xml

/WEB-INF/test.xml

</param-value>

</init-param>

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

</servlet>

<servlet-mapping>

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

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

</servlet-mapping>

각각 dispatcherServlet으로 분리

* **DispatcherServlet은 여러 개 설정 가능**
* **각 DispatcherServlet마다 각각의 ApplicationContext를 생성**

<servlet>

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

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

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

</servlet>

<servlet>

<servlet-name>dispatcher2</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>

<servlet-mapping>

<servlet-name>dispatcher2</servlet-name>

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

</servlet-mapping>

* **어플리케이션 컨텍스트 분리** 
  + 어플리케이션 레이어에 따라 어플리케이션 컨텍스트 분리
* **최상위 루트 컨텍스트 로더 설정** 
  + 컨텍스트 설정 파일들을 로드하기 위해 web.xml 파일에 리스너(ContextLoaderListener) 설정

<listener>

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

</listener>

* + 리스너 설정이 되면 /WEB-INF/applicationContext.xml파일을 읽어서 공통적으로 사용되는 최상위 컨텍스트를 생성
  + 그외의 다른 컨텍스트 파일들을 최상위 어플리케이션 컨텍스트에 로드하기 위해서는…

<context-param>

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

<param-value>/WEB-INF/board-service.xml

/WEB-INF/board-data.xml

classpath:board-security.xml

</param-value>

</context-param>

(classpath: 클래스 패스에 위치한 설정파일로부터 로드)

빈등록

<?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.xsd">*

**<bean id=*"boardController" class="com.kosta.board.controller.BoardController">***

<property name=*"boardService" ref="boardService"></property>*

</bean>

</beans>

* **컨트롤러 클래스 자동스캔** 
  + context:component-scan 선언
  + @Controller 어노테이션이 적용된 클래스는 자동스캔 대상.

|  |
| --- |
| <beans xmlns=*"http://www.springframework.org/schema/beans"*  xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance”*  xmlns:context=*"http://www.springframework.org/schema/context"*  xsi:schemaLocation=*"http://www.springframework.org/schema/beans*  *http://www.springframework.org/schema/beans/spring-beans.xsd*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context-3.0.xsd">*  **<context:component-scan base-package=*"com.kosta.board.controller" />***  </beans> |
| **특정 페키지 안쪽에 있는 모든 클래스를 빈을 등록한다 짱인다** |

* **@RequestMapping 선언** 
  + 요청 URL 매핑 정보를 설정
  + 클래스타입과 메소드에 설정 가능

|  |  |
| --- | --- |
| **@Controller**  **public class BoardController {**  **private BoardService boardService;**  **public void setBoardService(BoardService boardService) {**  **this.boardService = boardService;**  }  **@RequestMapping(“board/write.do")**  **public String initWrite() {**  **return "board/write";**  }  **@RequestMapping(“board/list.do")**  **public ModelAndView list() {**  List<Board> boards = boardService.findBoards();  … | **@Controller**  **@RequestMapping(“board")**  **public class BoardController {**  **private BoardService boardService;**  **public void setBoardService(BoardService boardService) {**  **this.boardService = boardService;**  }  **@RequestMapping(“write.do")**  **public String initWrite() {**  **return "board/write";**  }  **@RequestMapping(“list.do")**  **public ModelAndView list() {**  List<Board> boards = boardService.findBoards();  … |

\* 컨트롤러 메소드의 HTTP 메소드 한정

같은 URL의 요청에 대하여 HTTP메소드(POST, GET…) 에 따라 서로 다른 메소드를 Mapping 할 수 있음

|  |
| --- |
| @Controller  public class HelloController {  @RequestMapping(value=“hello.do”, **method=RequestMethod.GET**)  public String hello() {  return “helloget”;  }  @RequestMapping(value=“hello.do”, **method=RequestMethod.POST**)  public String hello2() {  return “hellopost”;  }  } |
| 전송방식에 따른 구분을줄수있다 ㅋㅋ |

인자를 받아보자.

|  |  |
| --- | --- |
| **파라미터 타입** | **설명** |
| HttpServletRequest,  HttpServletResponse | 서블릿 API |
| java.util.Locale | 현재 요정에 대한 Locale |
| InputStream, Reader | 요청 컨텐츠에 직접 접근할 때 사용 |
| OutputStream, Writer | 응답 컨텐츠를 생성할 때 사용 |
| @PathVariable 어노테이션 적용 파라메터 | URI 템플릿 변수에 접근할 때 사용 |
| @RequestParam어노테이션 적용 파라메터 | HTTP 요청 파라미터를 매핑 |
| @RequestHeader어노테이션 적용 파라메터 | HTTP 요청 헤더를 매핑 |
| @CookieValue어노테이션 적용 파라미터 | HTTP 쿠키 매핑 |
| @RequestBody 어노테이션 적용 파라미터 | HTTP 요청의 몸체 내용에 접근할 때 사용 |
| Map, Model, ModelMap | 뷰에 전달할 모델 데이터를 설정할 때 사용 |
| 커맨드 객체 | HTTP 요청 파라미터를 저장 한 객체, 기본적으로 클래스 이름을 모델명으로 사용  @ModelAttribute 어노테이션 설정으로 모델명을 설정할 수 있음 |
| Errors, BindingResult | HTTP 요청 파라미터를 커맨드 객체에 저장한 결과, 커맨드 객체를 위한 파라미터 바로 다음에 위치 |
| SessionStatus | 폼 처리를 완료 했음을 처리하기 위해 사용. @SessionAttributes 어노테이션을 명시한 session속성을 제거하도록 이벤트를 발생 시킨다. |

* **@RequestParam 어노테이션을 이용한 파라메터 매핑**

|  |
| --- |
| @Controller  public class HelloController {  @RequestMapping(“/hello.do”)  public String hello(@RequestParam(“name”) String name, @RequestParam(“age”) int age) {  system.out.println(name);  system.out.println(age);  …  }  } |
| @Controller  public class HelloController {  @RequestMapping(“/hello.do”)  public String hello(  **@RequestParam(value=“name”, required=false) String name,**  **@RequestParam(value=“age”, defaultValue=“0”) int age) {**  system.out.println(name);  system.out.println(age);  …  }  } |

* **커맨드 객체 이용** 
  + - **뷰의 입력한 데이터를 자바빈 객체를 이용해서 전송.**

|  |  |
| --- | --- |
| **@Controller**  **public class BoardController {**  **@RequestMapping(“board/saveBoard.do”)**  **public String save(Board board) {**  **}** | **public class Board {**  **private String title;**  **private String content;**  **public void setTitle(String title) {**  **this.title = title;**  **}**  **public void setContent(String content) {**  **this.content = content;**  **}**  **…**  **}** |
| **<form method=“post action=“${pageContext.request.contextPath}/board/saveBoard.do”>**  **제목: <input type=“text” name=“title” /><br/>**  **내용: <textarea name=“content”></textarea><br/>**  **<input type=“submit”/>**  **</form>**  **이름이 같아야된다.** | |

**리스트 객체도 받을수있다 .**

|  |  |
| --- | --- |
| **public class OrderCommand {**  **private List<OrderItem> orderItems;**    **public void setOrderItems(List<OrderITem> orderItems) {**  **this.orderItems = orderItems;**  **}**  **}** | **@Controller**  **@RequestMapping(“/hello.do”)**  **public class HelloController {**  **@RequestMapping(method=RequestMethod.GET)**  **public String hello(OrderCommand command) {**  **…**  **}**  **}** |
| **<form action=“hello.do” method=“post”>**  **<input type=“text” name=“orderItems[0].itemId” />**  **<input type=“text” name=“orderItems[0].number” />**  **<input type=“text” name=“orderItems[0].remark” />**  **<br/>**  **<input type=“text” name=“orderItems[1].itemId” />**  **<input type=“text” name=“orderItems[1].number” />**  **<input type=“text” name=“orderItems[1].remark” />**  **<br/>**  **<input type=“submit” />**  **</form>** | |

* **View에서 커맨드 객체에 접근하기** 
  + **커맨드 객체는 자동으로 반환되는 모델에 추가됨.**
  + **컨트롤러의 @RequestMapping 어노테이션 메소드에서 전달받은 커맨드 객체에 접근.**

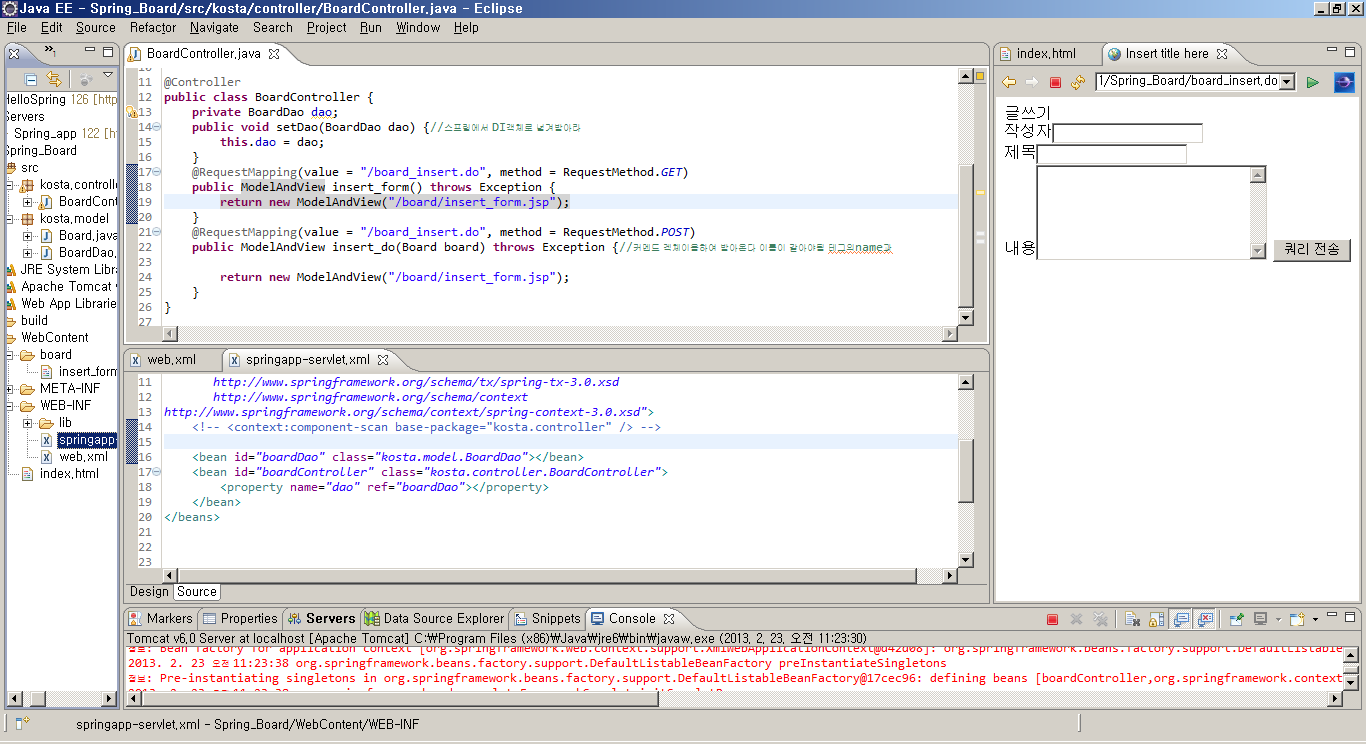
|  |  |
| --- | --- |
| **@Controller**  **public class BoardController {**  **@RequestMapping(“board/saveBoard.do”)**  **public String hello(Board command) {**  **…**  **}**  **}** |  |
| **@Controller**  **@RequestMapping(“/hello.do”)**  **public class HelloController {**  **@RequestMapping(method=RequestMethod.GET)**  **public String hello(@ModelAttribute(“faq”) Board board) {**  **…**  **}**  **}** |  |

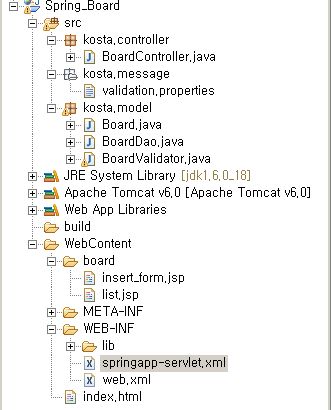
* + **서블릿 API 직접 사용**
  + **HttpSession의 생성을 직접 제어해야 하는 경우**
  + **컨트롤러에서 쿠키를 생성해야 하는 경우**
  + **서블릿 API를 선호하는 경우**

|  |
| --- |
| **javax.servlet.http.HttpServletRequest/javax.servlet.ServletRequest**  **javax.servlet.http.HttpServletResponse/javax.servlet.ServletResponse**  **javax.servlet.http.HttpSession**  **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public String hello(HttpServletRequest request, HttpServletResponse response) {**  **…**  **}**  **}** |

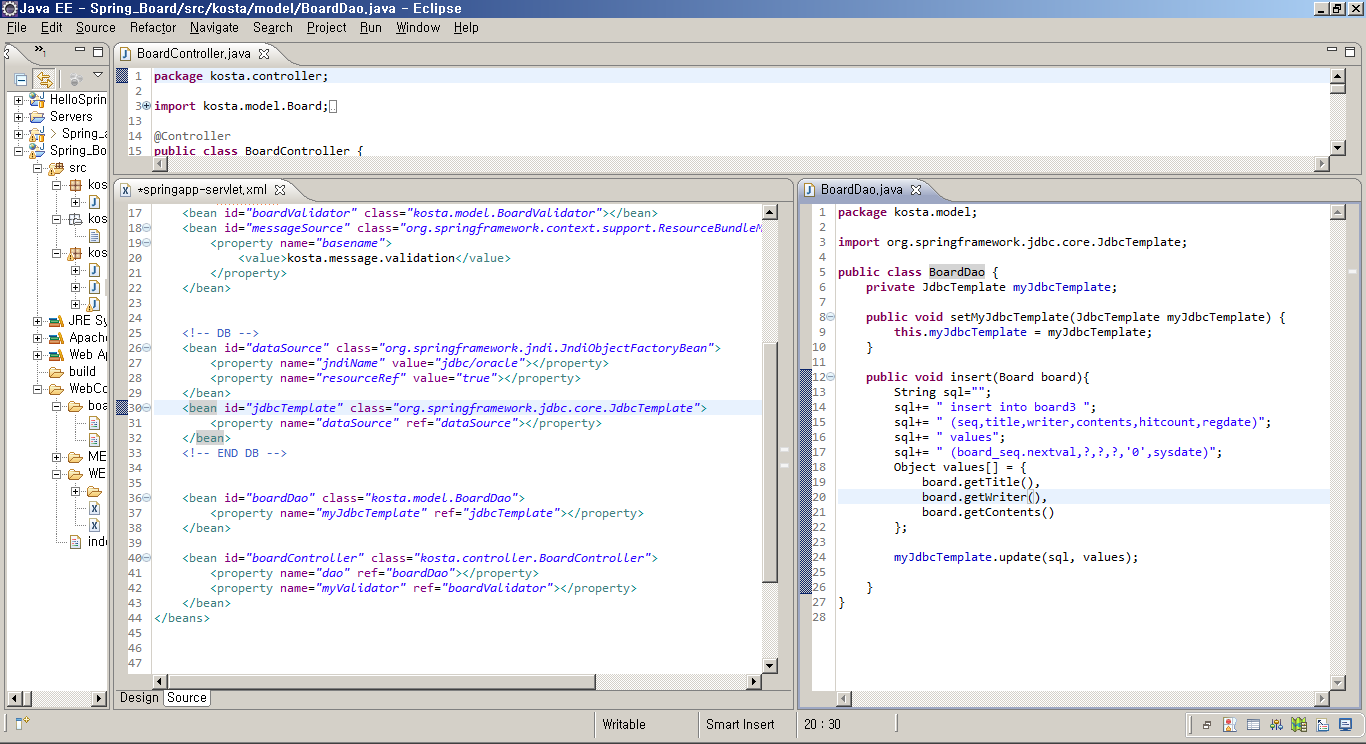
* **컨트롤러 메소드의 리턴타입**

|  |  |
| --- | --- |
| **리턴 타입** | **설명** |
| **ModelAndView** | **뷰 정보 및 모델 정보를 담고 있는 ModelAndView 객체** |
| **Model** | **뷰에 전달할 객체 정보를 담고 있는 Model을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다.(RequestToViewNameTranslator)** |
| **Map** | **뷰에 전달할 객체 정보를 담고 있는 Map을 리턴한다. 이때 뷰 이름은 요청 URL로부터 결정된다.(RequestToViewNameTranslator)** |
| **String** | **뷰 이름을 리턴한다.** |
| **View 객체** | **View 객체를 직접 리턴, 해당 View 객체를 이용해서 뷰를 생성한다.** |
| **void** | **메서드가 ServletResponse나 HttpServelResponse 타입의 파라미터를 갖는 경우 메서드가 직접 응답을 처리한다고 가정한다. 그렇지 않을 경우 요청 URL로부터 결정된 뷰를 보여준다.**  **(RequestToViewNameTranslator)** |
| **@ResponseBody 어노테이션 적용** | **메소드에서 @ResponseBody 어노테이션이 적용된 경우, 리턴 객체를 HTTP 응답으로 전송한다. HttpMessageConverter를 이용해서 객체를 HTTP응답 스트림으로 변환한다.** |

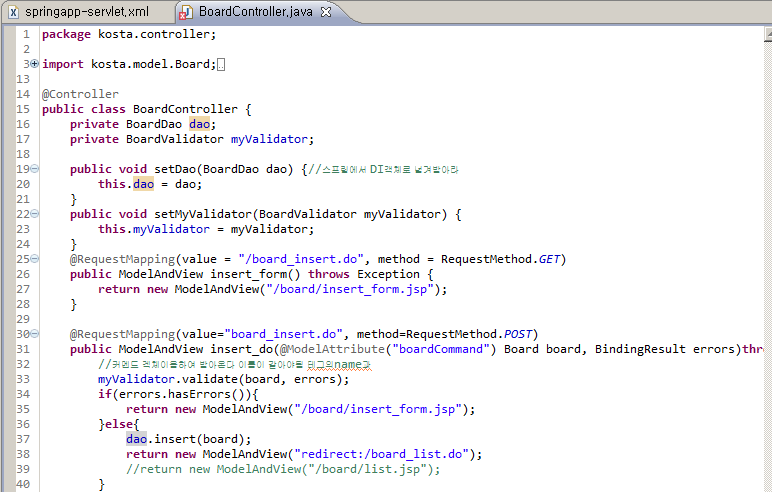
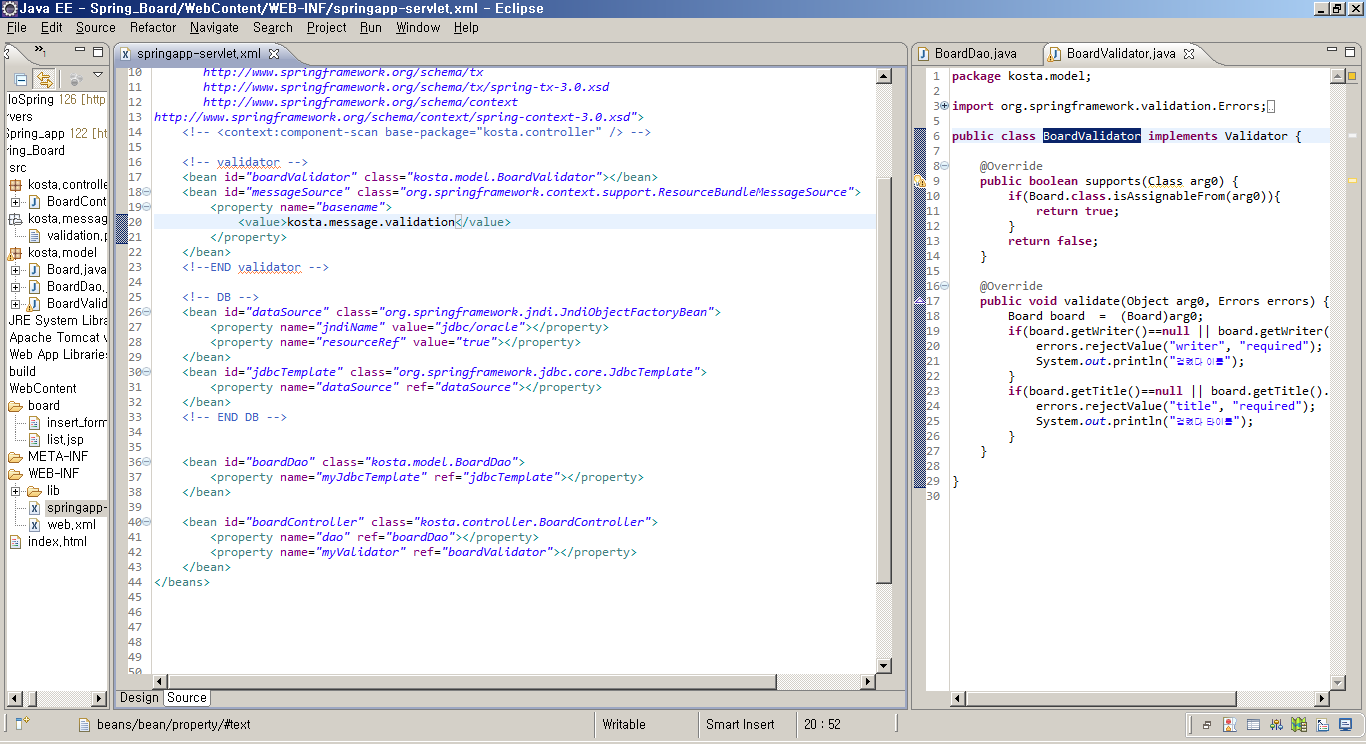
****

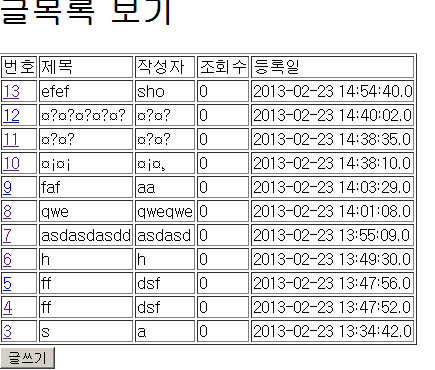
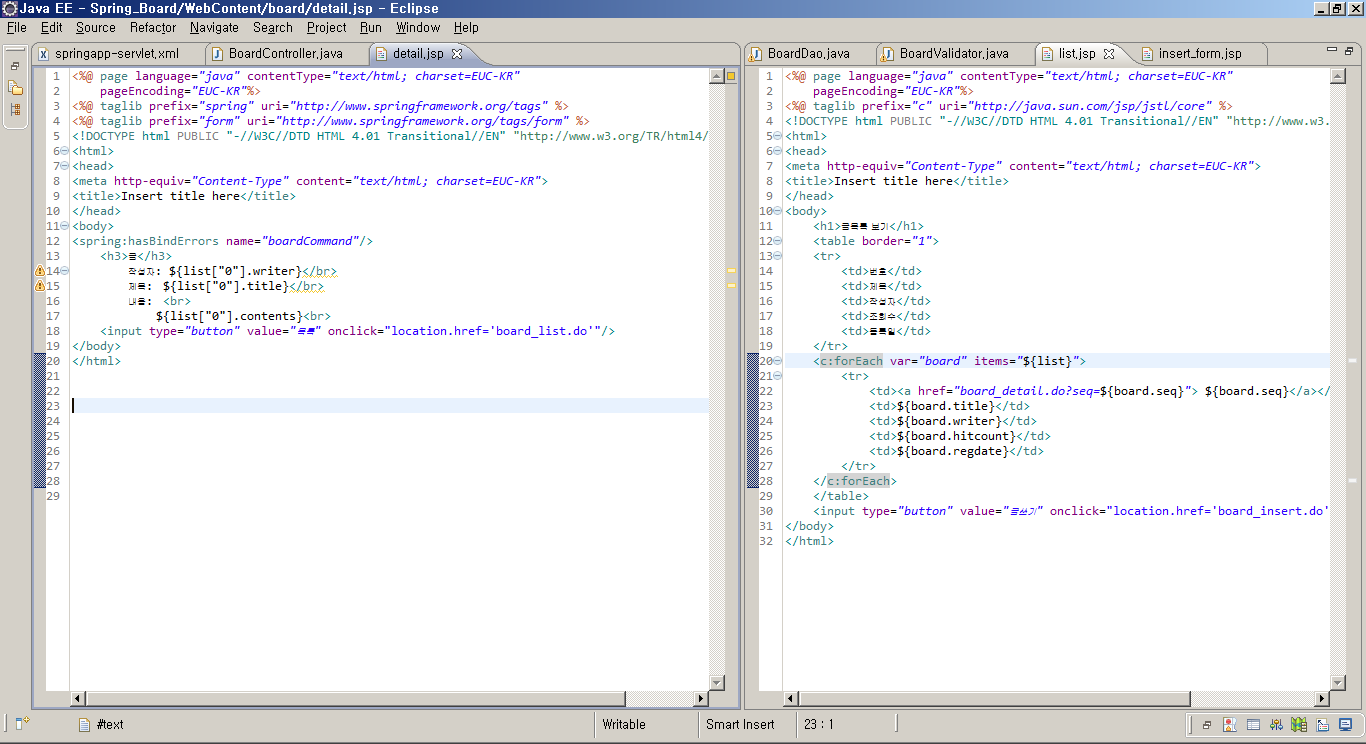
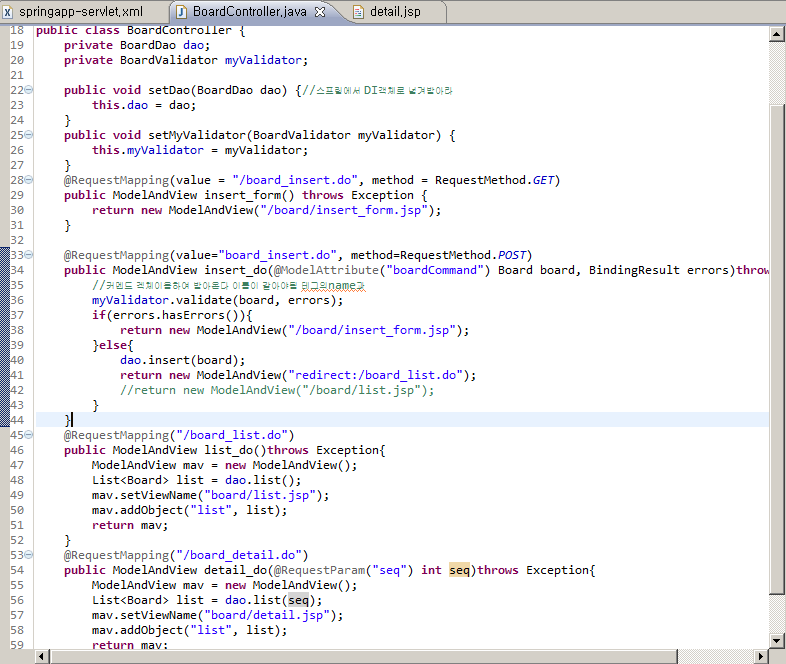
****

**JNDI DATABASE**

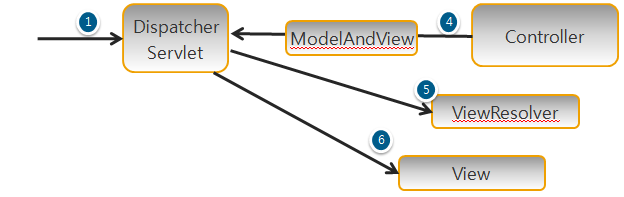
****

**Validation**

****

****

* **View Resolver : 논리적 뷰와 실제 JSP파일 매핑**
* **컨트롤러에서는 처리 결과를 보여줄 View 이름이나 객체를 리턴하고, DispatcherServlet은 View이름이나 View객체를 이용하여 뷰를 생성** 
  + **명시적 지정**
  + **자동 지정**

****

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

**<property name="prefix">**

***InternalResourceViewResolver는   
prefix + 논리뷰 + suffix로 설정***

***예) /WEB-INF/jsp/home.jsp***

**<value>/jsp/</value>**

**</property>**

**<property name="suffix">**

**<value>.jsp</value>**

**</property>**

**</bean>**

* **View 이름 명시적 지정** 
  + **ModelAndView와 String리턴타입**

|  |
| --- |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public ModelAndView hello() {**  **ModelAndView mav = new ModelAndView(“hello”);**  **return mav;**  **}** |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public ModelAndView hello() {**  **ModelAndView mav = new ModelAndView();**  **mav.setViewName(“hello”);**  **}** |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public String hello() {**  **return “hello”;**  **}** |

* **View 자동 지정** 
  + **RequestToViewNameTranslator를 이용하여 URL로부터 View이름을 결정한다.**
  + **자동 지정 유형** 
    - **리턴 타입이 Model이나 Map인경우**
    - **리턴 타입이 void 이면서 ServletResponse나 HttpServletResponse 타입의 파라미터가 없는 경우**

|  |
| --- |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public Map<String, Object> hello() {**  **HashMap<String, Object> model = new HashMap<String, Object>();**  **…**  **return model;**  **}** |

* **리다이렉트 뷰** 
  + **View 이름에 “redirect:” 접두어를 붙이면, 지정한 페이지로 리다이렉트 된다.**
  + **redirect:/product/productList.do**
  + **redirect:http://localhost/product/productList.do**

|  |
| --- |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public String hello() {**  **…**  **return “redirect:/product/productList.do”;**  **}** |

* **뷰에 전달하는 데이터** 
  + **@RequestMapping 어노테이션이 적용된 메서드의 Map, Model, ModelMap**
  + **@RequestMapping 메서드가 리턴하는 ModelAndView**
  + **@ModelAttribute 어노테이션이 적용된 메서드가 리턴 한 객체**
* **ModelAndView를 통한 모델 설정** 
  + **컨트롤러에서 처리결과를 보여줄 View와 View에 전달할 값(모델)을 저장하는 용도로 사용**
  + **setViewName(“viewName”)**
  + **addObject(String name, Object value);**

|  |
| --- |
| **@Controller**  **public class HelloController {**  **@RequestMapping(“/hello.do”)**  **public ModelAndView hello() {**  **ModelAndView mav = new ModelAndView();**  **mav.setViewName(“hello”);**  **mav.addObject(“message”, “안녕하세요”);**  **return mav;**  **}** |

* **전체경로와 서블릿기반 경로 매칭** 
  + **DispatcherServlet은 DefaultAnnotationHandlerMapping 클래스를 기본으로 HandlerMapping 구현체로 사용**
  + **Default로 컨텍스트 내의 경로가 아닌 서블릿 경로를 제외한 나머지 경로에 대해 매핑**

|  |  |
| --- | --- |
| **<servlet-mapping>**  **<servlet-name>dispatcher</servlet-name>**  **<url-pattern>\*.do</url-pattern>**  **<url-pattern>/game/\*</url-pattern>**  **</servlet-mapping>** | **@Controller**  **public class GameController {**  **@RequestMapping(“search/game.do")**  **public String getGame() {**  **return "game/info";**  **}**  **@RequestMapping(“info.do")**  **public String infoGame() {**  **return "game/info";**  **}** |
| **@RequestMapping(“/search/game.do”)**  **@RequestMapping(“/game/info”)** |

* + **서블릿 경로를 포함한 전체경로를 이용해서 매칭 하려는 경우**

|  |
| --- |
| **<bean**  **class=“org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping”**  **p:alwaysUseFullPath=“true” />**  **<bean**  **class=“org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerMapping”**  **p:alwaysUseFullPath=“true” />** |

* **@PathVariable 어노테이션을 이용한 URI 템플릿 짱인듯.**
  + **RESTful 방식** 
    - [**http://somehost/users/madvirus**](http://somehost/users/madvirus)
    - [**http://somehost/games/**](http://somehost/games/)
    - [**http://somehost/forum/board1/10**](http://somehost/forum/board1/10)
  + **@RequestMapping 어노테이션 값으로 {템플릿변수}를 사용한다.**
  + **@PathVariable 어노테이션을 이용해서 {템플릿변수}와 동일한 이름을 갖는 파라메터를 추가한다.**

|  |
| --- |
| **@Controller**  **Public class CharaterInfoController {**  **@RequestMapping(“/game/users/{userId}/characters/{characterId}”)**  **public String characterInfo( @PathVariable String userId**  **, @PathVariable int characterId, ModelMap model){**  **…**  **}**  **}** |

* **@RequestMapping 어노테이션의 추가설정 방법** 
  + **@RequestMapping 어노테이션을 클래스와 메소드에 함께 적용하는 경우**

|  |
| --- |
| **@Controller**  **@RequestMapping(“game/users/{userId}”)**  **Public class CharaterInfoController {**  **@RequestMapping(“characters/{characterId}”)**  **public String characterInfo(@PathVariable String userId**  **, @PathVariable int characterId, ModelMap model){**  **…**  **}**  **}** |

* + **Ant 스타일의 URI패턴 지원** 
    - **? : 하나의 문자열과 대치**
    - **\* : 하나 이상의 문자열과 대치**

**\*\* : 하나 이상의 디렉토리와 대치**

|  |
| --- |
| **@RequestMapping(“/members/\*.do”)**  **@RequestMapping(“/game/\*/items/{itemId}”)** |

**validate**

* **스프링은 유효성 검사를 위한 Validator 인터페이스와 유효성 검사 결과를 저장할 Errors 인터페이스 제공**
* **Validator 인터페이스를 이용한 폼 입력 값 검증**

|  |
| --- |
| **package org.springframework.validation;**  **public interface Validator {**  **boolean supports(Class<?> clazz); 🡪 Validator가 해당 클래스에 대한 validation 지원 여부**  **void validate(Object target, Errors errors);**  **🡪 validation실행 validation결과 문제가 있는 경우 errors 객체에 문제에 대한 정보를 저장**  **}** |

**구현**

|  |
| --- |
| package com.kosta.board.validator;  import org.springframework.validation.Errors;  import org.springframework.validation.ValidationUtils;  import org.springframework.validation.Validator;  import com.kosta.board.domain.Board;  public class BoardValidator implements Validator {  @Override  public boolean supports(Class<?> arg0) {  return Board.class.isAssignableFrom(arg0);  }  @Override  public void validate(Object object, Errors errors) {  ValidationUtils.rejectIfEmptyOrWhitespace(errors, "title", "required", "제목은 반드시 입력하세요.");  ValidationUtils.rejectIfEmptyOrWhitespace(errors, "content", "required", "내용을 입력하세요.");  }  } |

* **@Valid 어노테이션과 @InitBinder 어노테이션을 이용한 검증 실행** 
  + **@Valid : 스프링 프레임워크가 validation을 호출하도록 어노테이션을 이용하여 설정함**
  + **@InitBinder : Validator를 바인딩**

|  |
| --- |
| @RequestMapping("saveBoard.do")  public ModelAndView saveBoard(@Valid Board board, BindingResult result) {  ModelAndView mav = new ModelAndView();  if (result.hasErrors()) {  mav.setViewName("board/write");  mav.addObject("board", board);  return mav;  }  boardService.saveBoard(board);  mav.setViewName("board/list");  mav.addObject("boards", boardService.findBoards());  return mav;  }  @InitBinder  protected void initBinder(WebDataBinder binder) {  binder.setValidator(new BoardValidator());  } |

* **Validation 결과 출력** 
  + **spring 커스텀 태그를 사용**

|  |
| --- |
| **<%@taglib prefix=*"spring" uri="http://www.springframework.org/tags"%>***  **<%@taglib prefix=*"form" uri="http://www.springframework.org/tags/form"%>***  **…**  **…**  **<spring:hasBindErrors name=*"board" />***  **<form action=*"saveBoard.do" method="post">***  **<input type=*"hidden" name="number" value="${(board.number == '' || board.number == null) ? -1 : board.number}" />***  **<table>**  **<tr>**  **<td>제목</td>**  **<td><input type=*"text" name="title" value="${board.title}" /><form:errors path="board.title" />*** |

* **컨트롤러가 요청을 처리하기 전/후 처리**
* **HandlerInterceptor 제공 메서드** 
  + **boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**
    - **false를 리턴하면 request를 바로 종료한다.**
  + **void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**
  + **void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**
    - **뷰를 통해 클라이언트에 응답을 전송한 뒤에 실행**
    - **예외가 발생하여도 실행**
* **HandlerInterceptor 인터페이스 구현**
* **HandlerInterceptorAdaptor 클래스 제공**

|  |
| --- |
| **Public class XXXInterceptor extends HandlerInterceptorAdapter {**  **@Override**  **public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**  **throws Exception {**  **…**  **return true;**  **}**  **@Override**  **public void postHandle(HttpServletRequest request, HttpServletResponse response,**  **Object handler, ModelAndView modelAndView) {**  **…**  **}**  **@override**  **public void afterCompletion(HttpServletRequest request, HttpServletResponse response,**  **Object handler, Exception ex) {**  **…**  **}**  **}** |

* **HandlerMapping에 HandlerInterceptor 설정하기**

|  |
| --- |
| <bean class=“org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping”>  <property name=“alwaysUseFullPath” value=“true” />  <property name=“interceptors”>  <list>  <ref bean=“xxxInterceptor” />  <ref bean=“yyyInterceptor” />  <ref bean=“zzzInterceptor” />  </list>  </property>  </bean>  <bean id=“xxxInterceptor” class=“클래스패키지경로.XXXInterceptor” />  <bean id=“yyyInterceptor” class=“클래스패키지경로.YYYInterceptor” />  <bean id=“zzzInterceptor” class=“클래스패키지경로.ZZZInterceptor” /> |

* **@ExceptionHandler어노테이션을 이용한 예외처리** 
  + **파라메터로 받을 수 있는 타입** 
    - **HttpServletRequest, HttpServletResponse, HttpSession**
    - **Locale, InputStream/Reader, OutputStream/Writer**
    - **예외타입**
  + **리턴타입** 
    - **ModelAndView, Model, Map, View, String, void**

|  |
| --- |
| @Controller  public class MyController {  @RequestMapping(“/user/{userId}”)  public ModelAndView getUser(@PathVariable String userId) {  …  }  @ExceptionHandler(NullPointerException.class)  public String handleNullPointerException(NullPointerException e) {  return “error/nullException”;  }  } |
| <bean class=“org.springframework.web.servlet.handler.SimpleMappingExceptionResolver”>  <property name=“exceptionMappings”>  <props>  <prop key=“java.lang.NullPointerException”>  error/nullException  </prop>  </props>  </property>  </bean> |

* **웹시스템 간의 XML, JSON 형식의 데이터를 주고받는 경우** 
  + **XML, JSON 🡪 Java Object (unmarshalling)**
  + **Java Object 🡪 XML, JSON (marshalling)**
  + **@RequestBody, @ResponseBody 어노테이션은 HTTP 메시지 Body에 Java객체를 XML이나 JSON등의 타입으로 변환하여 담고, 역으로 변환하는데 사용**
* **HttpMessageConverter를 이용한 변환 처리** 
  + **주요 HttpMessageConverter 구현 클래스**

|  |  |
| --- | --- |
| **구현 클래스** | **설명** |
| **ByteArrayHttpMessageConverter (\*)** | **HTTP 메시지와 byte 배열 사이의 변환을 처리. 컨텐츠 타입은 application/octet-stream** |
| **StringHttpMessageConverter (\*)** | **HTTP 메시지와 String 사이의 변환을 처리한다. 컨텐츠 타입은 text/plain:charset=ISO-8859-1** |
| **FormHttpMessageConverter (\*)** | **HTML 폼 데이터를 MultiValueMap으로 전달받을 때 사용 컨텐츠 타입은 application/x-www-form-urlencoded** |
| **SourceHttpMessageConverter (\*)** | **HTTP 메시지와 javax.xml.transform.Source 사이의 변환을 처리. 컨텐츠 타입은 application/xml 또는 text/xml** |
| **MarshallingHttpMessageConverter** | **스프링의 Marshaller와 Unmarshaller를 이용해서 XML HTTP 메시지와 객체 사이의 변환을 처리한다. application/xml 또는 text/xml** |
| **MappingJacksonHttpMessageConverter** | **Jackson 라이브러리를 이용해서 JSON HTTP 메시지와 객체 사이의 변환을 처리한다. 컨텐츠 타입은 application/json** |

* **Content-Type 과 Accept 헤더기반의 변환처리** 
  + **@AnnotationMethodHandlerAdapter가 HttpMessageConverter를 이용해서 요청 몸체 데이터를 @RequestBody 어노테이션이 적용된 자바 객체로 변환 할 때에는 HTTP요청 헤더의 Content-Type 헤더에 명시된 미디어 타입(MIME)을 지원하는 HttpMessageConverter를 구현체로 사용**

**@ResponseBody 어노테이션을 이용해서 리턴하는 객체를 HTTP 메시지의 body로 변환할 때에는 HTTP요청 헤더의 Accept 헤더에 명시된 미디어 타입을**

|  |
| --- |
| <script type=“”>  …  xmlhttp.open(“GET”, “json.do”, true);  xmlhttp.setRequestHeader(“Content-Type”, “application/x-www-form-rulencoded”);  xmlhttp.setRequestHeader(“Accept”, “application/json”);  xmlhttp.send();  …  </script> |
| @RequestMapping(value=“guestmessage/json.do”, method=RequestMethod.GET, headers = “accept=application/json”)  @ResponseBody  public GuestMessageList listJson() {  } |

**JSTL과 EL을 사용하므로써** **http://slog2.egloos.com/3581446**

**<% %>와 같은 스크립팅을 없앨 수 있다 !**

**JSP페이지상에  자바 소스는 이제 안녕 ~ ' '/**

Expression Language는 JSP에서 기본으로 지원한다

세팅해야 할 것은 JSTL( JavaServer Pages Standard Tag Library ) 이다

**1. 다운로드**

<http://tomcat.apache.org/> - Taglibs - Standard - JSTL 1.1 download - binaries - jakarta-taglibs-standard-1.1.2.zip

직접다운로드경로 - <http://archive.apache.org/dist/jakarta/taglibs/standard/binaries/jakarta-taglibs-standard-1.1.2.zip>

    JSTL 매뉴얼 -  <http://pds17.egloos.com/pds/201002/12/66/JSTL-Reference-11.pdf>

**2. 설치**

다운로드 받은 파일에 압축을 풀고 lib 폴더의 jstl.jar 와 standard.jar 파일을 /WEB-INF/lib 에 복사 후

JSP 파일 상단에 다음의 지시문을 추가한다

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

( core 태그 lib만 추가하였고 그외  fmt, sql, x 태그를 사용하기 위해선 지시문을 추가해야 한다 )

tip) 이클립스에서 JSP 생성시 자동으로 소스 추가하는 방법

window - Prefrences - Web - JSP Files - Templates - JSP HTML에 내용 추가

**3. Expression language**

EL이란 JSP에서 저장객체를 출력할때 스크립팅을 전혀 쓰지 않을 수 있는 기술이다

이것으로 인해 <%=request.getParameter("name")%> 와 같은 스크립팅 들을 쓸필요가 없어진다

(1) 저장객체 접근

${param.name} => request.getParameter("name");

${member} => request.getAttribute("member");

    저장객체의 attribute 에 자동으로 접근하는데

    자동검색순위는 page, request, session, application 순이다

${member.name} => Member m = (Member)request.getAttribute("member");

                                    m.getName();

    attribute 저장된 member의 name속성(또는 키)에 접근

    도트(.) 의 왼쪽은 Beans 이거나 Map 타입이어야 한다

${list["0"]} => List list = (List)request.getAttribute("list");

                         list.get(0);

    attribute 저장된 list를 가져온다

    []안에 값은 list의 키값 or 프로퍼티명 or 인덱스이다

    []의 왼쪽은 Map, Beans, 배열, List 타입이 올 수 있다

    여기서 알아야 할 점이 하나 있는데 EL에서는 해당값이 null이거나 공백일 경우에는

    아무 내용도 표시하지 않고 에러도 발생하지 않는다..

(2) 연산자

EL은 연산자들을 사용할 수 있다. 예제는 JSTL에서 같이 사용하는걸 보자

1) 산술연산자 : + - \* / (div) % (mod)

2) 논리연산자 : &&(and) ||(or) !(not)

3) 관계연산자 : ==(eq) !=(ne) <(lt) >(gt) <=(le) >=(ge)

4) 삼항연산자 : ?:

예) ${colors == null ? "transparent" : colors}

5) 그 외 예약어 : true, false, null, instanceof, empty( null이거나 공백일때 )

**4. JSTL**

        core태그 중에서 유용한 것들만 살펴보자

        모든 내용을 보고 싶다면 JSTL 매뉴얼 참조

        태그에 진하게 표시된 속성은 필수 항목이다

(1) **<c:out>**

단순히 내용을 출력하는 기능이다

<c:out**value="${member.name}"** default="이재원" />

위와 과 같이 쓸 경우 일반적으로 ${member.name} 을 쓴것과 같지만

${member.name}이 null일 경우 dafault에 있는 값으로 대채한다

(2) **<c:set>**

저장객체의 setAttriubte 기능을 한다

<c:set **var="addr"** v**alue="myaddr"** scope="session" />

<c:set **var="no"** **value="${member.age+1}"**  />

첫번째는 session 에 addr 이란이름으로 myaddr 스트링을 저장한 것이다

두번째는 no 이란이름으로 member.age+1으로 산술연산된 값을 저장하는데

scope를 생략하면 기본으로 page에 저장된다

(3) **<c:remove>**

    removeAttribute의 기능을 한다

    <c:remove **var="no"** scope="page" />

    지정된 scope의 no란 이름이 attribute를 remove한다

    scope를 생략할 경우 모든 범위의 attribute 가 지워진다

(4) **<c:if>**

    if 조건문을 사용한다.. 단 else if 와 else는 지원하진 않고 유사한 <c:choose> 가 존재한다

    <c:if **test="${member.age < 20}"** var="result" scope="page">

        당신은 미성년자입니다

    </c:if>

    <c:if **test="${!empty list}"** >

        content

    </c:if>

    test에서 조건을 검사하고 true 일 경우에만 <c:if>태그안의 내용이 보여진다

    여기서 EL 의 연산자들을 쓸 수 있다!!

    var는 조건을 검사하고 리턴되는 boolean값을 저장하는 attribute이름이고

    scope는 var가 저장되는 범위입니다. scope가 없을시 기본값은 page이다

(5) **<c:choose>**

if, else if, else 와 유사하다. 아니 똑같다고 봐도 된다

<c:choose> 태그안에는 <c:when> 과 <c:otherwise>가 들어간다

<c:choose>

<c:when **test="${vo.type==1}"**>

    <img src="<%=cp%>/images/antenna/type1.jpg" />

</c:when>

<c:when t**est="${vo.type==2}"**>

    <img src="<%=cp%>/images/antenna/type2.jpg" />

</c:when>

<c:when **test="${vo.type==0 or vo.type==3}"**>

    <img src="<%=cp%>/images/antenna/type\_ment.jpg" />

</c:when>

<c:otherwise>

    <img src="<%=cp%>/images/antenna/type\_heart.jpg" />

</c:otherwise>

</c:choose>

<c:choose>안에서 순서대로 <c:when> 의 조건문을 검사해 true가 나오면

해당 <c:when> 태그안에 내용을 보여주고 <c:choose>문은 끝나게 된다

만약 모든 <c:when>의 조건문이 false일 경우 <c:otherwise>안에 내용을 보여주게된다

(6) **<c:forEach>**

Collection을 반복할때 쓰인다. 게시판 같은 반복적인 처리를 할때 매우 유용하다

지원되는 컬렉션을 다음과 같다

Arrays ( 배열 )

java.util.Collection

java.util.Iterator

java.util.Enumeration

java.util.Map

<table>

<c:forEach **items="${list}"**  **var="notice"**  varStatus="status">

<tr>

    <td>${status.count}</td>

    <td>${notice.title}</td>

    <td>${notice.writer}</td>

    <td>${notice.wdate}</td>

    <td>${notice.readcount}</td>

</tr>

</c:forEach>

</table>

items는 Collection 객체를 말한다

var는 Collection에서 객체를 하나씩 가져올때마다 담는 객체다

그 객체가 Beans일 경우 Beans클래스를 자동으로 찾아 담는다

varStatus는 일련의 속성들을 정의한 객체다 ( javax.servlet.jsp.jstl.core.LoopTagStatus )

다음은 일반적은 for문 처럼 사용한 예다

<c:forEach var="x" begin="0" end="10" step="2">

${x},

${x\*x}

</c:forEach>

(7) **<c:forTokens>**

<c:forEach> 와 같이 반복 태그로 이것은 StringTokenizer 클래스를 이용한다.. 그다지 쓸일은 없다

구분자( delims ) 를 공백, 쉼표, 마침표 로 한 예이다

<c:forTokens **items="${param.text}"** **delims=" ,."** **var="word"** begin="1" end="10" step="1" >

    ${word}

</c:forTokens>

 JSTL 매뉴얼 -  <http://pds17.egloos.com/pds/201002/12/66/JSTL-Reference-11.pdf>

[**[Spring 3 - @MVC] 모델 바인딩과 검증 #4 - Validator 와 BindingResults, Errors**](http://springsource.tistory.com/18)

**http://springsource.tistory.com/18**

@ModelAttribute 로 지정된 모델 오브젝트의 바인딩 작업이 실패로 끝나는 경우는 두 가지가 있다. 하나는 타입 변환이 불가능한 경우이고, 다른 한 가지는 타입 변환은 성공했지만 검증기를 이용한 검사를 통화하지 못했기 때문이다.

폼의 서브밋을 처리하는 컨트롤러 메소드에서는 반드시 검증기를 이용해 입력 값을 확인하고, 그 결과에 따라 폼을 정상 처리하고 다음 작업으로 넘어가거나 폼을 다시 띄워서 사용자가 잘못 입력한 값을 수정하도록 해줘야 한다.

스프링은 이런 검증 과정에서 사용할 수 있는 validator 라는 이름의 표준 인터페이스를 제공한다. Validator 를 통한 검증과정의 결과는 BindingResult 를 통해 확인할 수 있다. BindingResult 는 Errors 의 서브인터페이스다. Errors 인터페이스에는 바인딩 또는 검증과정에서 오류를 발견하면 그에 대한 정보를 등록할 수 있는 메소드가 제공되고 있다. 이 세가지 API 의 구체적인 사용방법을 알아 보자.

**Validator**

스프링의 Validator 는 스프링에서 범용적으로 사용할 수 있는 오브젝트 검증기를 정의할 수 있는 API 다. 물론 @Controller로 HTTP 요청을 @ModelAttribute 모델에 바인딩할 때 주로 사용된다. 또는 비즈니스 로직에서 검증로직을 분리하고 싶을 때도 사용할 수 있다.

Validator 인터페이스는 다음과 같이 두 개의 메소드로 구성되어 있다. supports() 는 이 검증기가 검증할 수 있는 오브젝트 타입인지를 확인해 주는 메소드다. supports() 메소드를 통과한 경우에만 validate() 가 호출된다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package org.springframework.validation;    public interface Validator {        boolean supports(Class<?> clazz);        void validate(Object target, Errors errors);    } |

Validator 를 이용해 검증한 결과, 아무 문제가 없다면 메소드를 정상 종료하면 된다. 만약 오류가 발견되면 Errors 인터페이스를 통해서 특정 필드나 모델 오브젝트 전체에 대해 오류 정보를 등록할 수 있다. Errors 인터페이스를 통해서 등록된 오류는 최종적으로 BindingResult 에 담겨 컨트롤러에 전달된다. 검증결과를 보고 컨트롤러는 그에 맞는 처리를 하고 뷰를 리턴해야 한다. 일반적으로 폼을 처리하는 컨트롤러 메소드라면 바인딩 또는 검증 과정에서 오류가 하나라도 발견되면 다시 폼을 띄워서 에러 메시지를 보여주고 재입력을 요구한다.

그렇다면 어떤 것을 검증해야 할까? 가장 대표적인 검증은 값을 입력했는지 확인하는 것이다. 필수 입력 필드에 값이 입력되지 않았다면 폼을 정상 처리할 수 없다. 물론 이 필수 입력조건은 브라우저에서 자바스크립트를 통해서도 얼마든지 확인이 가능하다. 자바스크립트를 잘 이용하면 입력한 값의 타입이나 형식도 확인할 수 있다.

그렇다면 자바스크립트로 입력 값을 모두 확인한 경우에는 서버의 검증 작업이 필요없을까? 그렇지 않다. 자바스크립트의 검증은 서버로 자주 전송이 일어나는 것을 막아주려는 목적으로만 사용해야 한다. 자바스크립트의 검증을 통과했다고 서버의 검증 작업을 생략하면 위험하기 때문이다. 브라우저에서 자바스크립트가 동작하지 않도록 만들 수도 있다. 또, 스크립트를 무시하고 폼을 조작해서 오류가 있는 정보를 강제로 서버로 보낼 수도 있다. 따라서 서버에서 최종적으로 검증하는 작업이 생략되면 잘못된 값이 등록되거나 기타 문제를 일으킬 가능성이 있다. 따라서 모델에 대한 검증기를 통한 검증작업은 생략하면 안되고 필수 입력 값 확인도 반드시 거쳐야 한다.

사용자 입력 폼을 처리하는 컨트롤러 메소드에서 사용할 UserValidator를 만들어 보자. 모델 오브젝트는 User 타입이어야 한다. 따라서 supports() 메소드는 다음과 같이 만들면 된다. 때로는 User 의 서브클래스가 사용될 수도 있으므로 isAssignableFrom() 메소드를 사용하는 것이 좋다.

|  |  |
| --- | --- |
| 1  2  3 | public boolean supports(Class<?> clazz) {      return (User.class.isAssignableFrom(clazz));  } |

다음은 validator() 메소드를 만들 차례다. 파라미터로 전달되는 target 오브젝트는 결국 User 타입 오브젝트일 것이다. 따라서 다음처럼 User 타입으로 캐스팅한 후에 검증 작업을 진행해야 한다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public void validate(Object target, Errors errors) {      User user = (User)target;      if(user.getName() == null || user.getName().length() == 0) {          errors.rejectValue("name", "field.required");      }  } |

먼저 user의 getName() 을 이용해 name 필드의 값이 입력됐는지 확인하자. 다음과 같이 null 인 것과 빈 스트링인 것, 두 가지 모두 확인할 필요가 있다. 입력 값이 없다면, 이때는 errors 의 rejectValue() 메소드를 호출해서 어떤 필드에 문제가 있는지 등록해 줘야 한다.

rejectValue() 메소드의 두 번째 파라미터는 에러코드다. 에러 코드의 사용방법은 뒤에서 다시 설명하겠다. 일단은 에러 코드를 넣는 다는 사실만 기억해두자.

스트링 타입의 프로퍼티 값을 매번 if 문으로 체크하는 것은 번거롭다. 이럴 때 ValidationUtils 의 유틸리티 메소드를 사용하면 편리하다. 위의 validate() 메소드는 다음과 같이 간단하게 수정할 수 있다.

|  |  |
| --- | --- |
| 1  2  3  4 | public void validate(Object target, Errors errors) {      User user = (User)target;      ValidationUtils.rejectIfEmpty(errors, "name", "field.required");  } |

공백문자도 허용하지 않으려면 rejectIfEmptyOrWhitespace() 메소드를 사용하면 된다. 0보다 작으면 안되는 age 필드가 있다고 해보자. 사용자가 숫자를 입력하면 바인딩까지는 정상적으로 진행되겠지만, 허용되는 범위 밖에 있다면 검증 작업에서 오류가 있다고 확인해줘야 한다. 따라서 다음과 같은 검증코드를 추가한다. rejectValue() 의 세번째와 네번째 파라미터는 에러코드와 함께 에러 메시지를 작성할 때 사용될 정보다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public void validate(Object target, Errors errors) {      User user = (User)target;      ValidationUtils.rejectIfEmpty(errors, "name", "field.required");        if(user.getAge() < 0) {          errors.rejectValue("age", "field.min", new Object[]{0}, null);      }  } |

여러 개의 필드의 조건을 결합해서 검증하는 경우도 있다. 이때는 검증 과정에서 문제가 발견되더라도 특정 필드에 대한 오류로 만드는 것은 적당하지 않다. 이럴 때 필드 이름을 지정해야 하는 rejectValue() 대신 글로벌 에러를 추가할 수 있는 reject() 메소드를 사용하는 것이 적당하다.

Validator 는 보통 미리 정해진 단순 조건을 이용해 검증하는 데 사용된다. 필수 값의 입력 여부나 값의 범위, 길이, 형식 등이 주로 검증조건이 된다. 하지만 때로는 비즈니스 로직을 적용해서 값을 검증할 수도 있다. Validator 는 싱글톤 빈으로 등록돼서 사용할 수 있다. 따라서 UserService 와 같은 빈을 이용해 비즈니스 로직 관점에서 입력 값을 검증하는 작업을 넣을 수 있다. 대표적인 예가 사용자 아이디가 이미 등록된 것인지 확인하는 경우다. 아이디 중복 문제는 서비스 계층을 거쳐 DAO 까지 참여해야만 알 수 있다. Validator 에서는 이런 길이나 형식 정도만 판단하고 아이디 중복문제는 컨트롤러가 아니라 서비스 계층의 오브젝트에서 판단하게 할 수 있다. 이 경우 오류가 있다면 비즈니스 로직으로서 의미가 있는 체크 예외나 특별한 리턴 값을 돌려주고, 이를 다시 컨트롤러가 확인해서 그에 따른 조치를 취하게 할 수도 있다.

자바 서버 개발자들 사이에서 지속되는 논쟁 중에 "검증 로직은 어느 계층의 로직인가" 라는 것이 있다. 검증은 프리젠테이션 계층의 컨트롤러 로직이며 그 안에서 모든 사용자 요청에 대한 검증이 끝나야 한다고 보는 개발자가 있는 반면, 검증의 상당 부분은 비즈니스 로직과 관련이 있으니 서비스 계층에서 담당해야 한다고 보는 개발자도 있다. 어떤 경우에는 DAO 에서 DB에 데이터를 반영할 때 도메인 오브젝트의 정보를 활용해서 검증을 수행하는 경우도 있다. 따라서 검증 로직은 특정 계층에 종속되기 보다는 도메인 오브젝트처럼 독립적으로 만드는 것이 좋다. 스프링의 Validator 를 구현한 검증 클래스를 도메인 오브젝트 단위로 만들고 이를 필요한 곳에서 호출해서 사용하거나, 반대로 Validator에서 필요한 로직을 담고 있는 빈을 호출하는 것이 좋다. 다만 Validator 를 써서 검증 작업을 진행하는 곳을 어디로 할지는 고민해봐야 한다. 스프링에서는 앞으로 설명할 네 가지 방법으로 Validator를 적용할 수 있다.

**컨트롤러 메소드 내의 코드**

Validator 는 빈으로 등록이 가능하므로 이를 컨트롤러에서 DI 받아 두고 각 메소드에서 필요에 따라 직접 validate() 메소드를 호출해서 검증 작업을 진행해도 좋다. 이때는 모델 오브젝트의 타입은 굳이 확인할 필요가 없으므로 supports() 는 생략해도 좋다. validator() 를 호출할 때 필요한 Errors 타입 오브젝트는 메소드 파라미터로 제공받을 수 있는 BindingResult 타입 오브젝트를 사용하면 된다. BindingResult 는 Errors 의 서브 인터페이스다.

앞에서 만든 User 오브젝트에 대한 검증기인 UserValidator를 컨트롤러 메소드내에서 사용하는 코드를 살펴보자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @Controller  public class UserController {        @Autowired      private UserValidator validator;        @RequestMapping("/add")      public void add(@ModelAttribute User user, BindingResult result) {            validator.validate(user, result);          if(result.hasErrors()) {              // 검증에 실패한 경우          } else {              // 검증에 성공한 경우          }      }    } |

validate() 메소드를 실행한 후에 BindingResult 의 hasErrors() 메소드로 검증 작업에서 등록된 오류가 있는지 확인한다.  하나라도 오류가 있으면 그에 따른 조치를 취해야 한다.

**@Valid 를 이용한 자동검증**

Validator 를 적용하는 두 번째 방법은 JSR-303(Bean Validation) 의 @javax.validation.Valid 애노테이션을 사용하는 것이다.  사실 스프링이 이 @Valid 라는 애노테이션을 차용했을 뿐 내부적으로는 스프링의 Validator 를 이용한 검증이 수행된다.

앞의 방법과는 달리, 컨트롤러 메소드 코드에서 Validator 오브젝트를 가져다 validate() 메소드를 실행하는 대신 바인딩 과정에서 자동으로 검증이 진행되게 한다. WebDataBinder 에는 프로퍼티 에디터, 컨버전 서비스 뿐 아니라 Validator 타입의 검증용 오브젝트도 설정해줄 수 있다. 이 방식을 적용한 아래 UserController 소스를 살펴보자.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | @Controller  public class UserController {        @Autowired      private UserValidator validator;        @InitBinder      public void initBinder(WebDataBinder dataBinder) {          dataBinder.setValidator(validator);        }        @RequestMapping("/add")      public void add(@ModelAttribute @Valid User user, BindingResult result) {            ...      }    } |

컨트롤러 메소드의 @ModelAttribute 파라미터에 @Valid 애노테이션을 하나 더 추가한다. 그러면 이 모델을 바인딩하는 WebDataBinder 는 @InitBinder 메소드에서 등록된 Validator 를 이용해 모델을 검증하고, 그 결과를 BindingResult 에 넣어서 전달해 준다. 따라서 메소드의 파라미터로 전달받는 BindingResult 에는 이미 UserValidator 로 검증한 결과가 담겨 있을 것이다.

@InitBinder 를 통해 Validator 를 미리 등록해주면 해당 컨트롤러의 모든 메소드에서 검증 기능을 사용할 수 있기 때문에 편리하다.

**서비스 계층 오브젝트에서의 검증**

자주 사용되는 방법은 아니지만 원한다면 얼마든지 Validator 를 컨트롤러 밖에서도 사용할 수 있다. 비즈니스 로직이 적용된 Validator 는 서비스 계층에서도 활용할 수 있다. 여러 개의 서비스 계층 오브젝트에서 반복적으로 같은 검증 기능이 사용된다면 Validator 로 검증 코드를 분리하고 이를 DI 받아 사용할 수 있다. 이때는 BindingResult 타입의 오브젝트를 직접 만들어서 Validator 의 validate() 메소드로 전달해야 하는데, BeanPropertyBindingResult 를 사용하는 것이 적당하다.

비즈니스 로직을 담은 코드에서 스프링의 API 가 등장하는게 부담스럽다면 Validator 를 이용하는 대신 독자적인 검증 API 나 코드를 사용해도 좋다. 이런 경우에 검증결과를 다시 사용자에게 알려줘야 하므로 컨트롤러로 검증 오류 정보를 전달해 줘야 한다. 리턴 값은 정상적인 처리를 위해 사용돼야 하므로 이런 경우에는 체크 예외를 던져서 컨트롤러가 확인할 수 있게 해야 하는 번거로움이 있다.

**서비스 계층을 활용하는 Validator**

코드에 의한 것이든 @Valid 에 의한 것이든 상관없이 Validator 를 적용했을 경우에는 Validator 를 빈으로 등록하고 서비스 계층의 기능을 사용해 검증 작업을 수행할 수 있다. 이렇게 하면 서비스 계층에 담긴 검증 로직을 재사용할 수 있으면서 그 결과는 컨트롤러 내에서 BindingResult 를 통해 전달받을 수 있기 때문에 번거롭게 예외를 만들어서 던지는 코드가 없어도 된다.

예를 들어 현재 사용자의 최대 계좌이체 한도를 체크해서 만약 한도 이상을 요청했을 경우에는 에러 메시지를 보여줘야 한다고 생각해보자. 현재 로그인한 사용자의 최대 계좌이체 한도를 DAO를 통해 가져와 이체 요청금액과 비교하는 로직은 서비스 계층의 오브젝트에 넣는 것이 적당하다. Validator 에서는 이 서비스 계층 오브젝트를 호출해서 요청금액의 한도가 넘었는지만 확인받을 수 있으면 된다. 만약 문제가 있다면 다른 검증 로직과 마찬가지로 해당 필드정보를 rejectValue() 에 넣어서 검증 오류를 추가한 뒤에 컨트롤러에 넘겨주면 된다. 컨트롤러는 검증과정에서 오류가 있으니 다시 처음 입력 폼을 띄울 것이고, 이때 이체한도와 관련된 적절한 메시지를 출력해 주면 된다.

이 방식의 단점은 컨트롤러에서 두 번 이상 서비스 계층을 호출한다는 점이다. 이 때문에 서비스 계층에 트랜잭션 경계를 설정해뒀다면 트랜잭션이 두 번 생성될 수도 있다. 그럼에도 전체적으로 코드는 깔끔해지고, 각 오브젝트는 자기 책임에 충실하게 독립적으로 작성할 수 있다는 장점이 있다.

비즈니스 로직이 반영되는 검증 작업은 세번째 방법과 네번째 방법 중에서 하나를 선택해서 적용하면 된다. 네 번째 방법이 좀 더 스프링다운 방법이라고 볼 수 있긴하지만 취향에 따라 세번째 방식을 사용해도 무방하다.

**JSR-303 빈 검증 기능**

최근에 표준 스펙으로 인증받은 JSR-303 빈 검증 방식도 스프링에서 사용할 수 있다. 스프링에서는 LocalValidatorFactoryBean 을 이용해 JSR-303 검증 기능을 사용할 수 있다. LocalValidatorFactoryBean 은 JSR-303 의 검증 기능을 스프링의 Validator 처럼 사용할 수 있게 해주는 일종의 어댑터다. LocalValidatorFactoryBean 을 빈으로 등록하고, 컨트롤러에서 Validator 타입으로 DI 받아서 @InitBinder 에서 WebDataBinder 에 설정하거나 코드에서 직접 Validator 처럼 사용할 수 있다.

JSR-303의 빈 검증 기술의 특징은 모델 오브젝트의 필드에 달린 제약조건(constraints) 애노테이션을 이용해 검증을 진행할 수 있다는 점이다. @NotNull 애노테이션은 필드의 값이 null 이 아님을 확인하는 제약조건이다. @Min 은 최소값을 지정할 수 있는 제약조건이다. 이 두가지를 이용하면 이전의 UserValidator 클래스를 다음과 같이 간단하게 만들 수 있다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class User {        @NotNull      private String name;        @Min(0)      private int age;    } |

@NotNull 로는 빈 문자열은 검증되지 않는다는 점을 주의하자, 폼에 텍스트 필드가 있고 아무 값을 입력하지 않으면 빈 문자열이 들어올 수도 있다. 빈 문자열까지 확인하려면 @Size(min=1) 을 사용해야 한다.

이 제약조건 애노테이션을 사용해 검증을 수행하려면 LocalValidatorFactoryBean 을 빈으로 등록해줘야 한다. 이 팩토리 빈이 생성하는 빈의 타입은 스프링의 Validator 다.

|  |  |
| --- | --- |
| 1  2 | <bean id="localValidator"      class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean" /> |

이제 등록된 검증기 빈을 UserValidator 처럼 컨트롤러에 DI 받아서 사용하면 된다. @InitBinder 에서 WebDataBinder 에 넣고 @Valid 를 이용해 자동검증해도 되고, 코드 내에서 직접 validate() 메소드를 호출해도 된다.

JSR-303 의 @NotNull 과 같은 애노테이션을 이용한 제약조건은 어럽지 않게 새로 만들 수 있다. 검증 로직을 담은 별도의 JSR-303 ConstraintValidator(스프링의 Validator 와 다르다는 점 주의) 를 만들고 이를 애노테이션에서 이용할 수도 있다. 검증정보가 도메인 오브젝트 클래스 안에 담겨도 상관없다면 JSR-303 의 빈 검증 기능을 활용하는 것이 Validator 를 구현한 검증기를 사용하는 것 보다 편리하다.

다음은 ConstraintValidator 를 구현한 MemberNoValidator 를 이용해 값을 검증하는 @MemberNo라는 애노테이션 제약조건을 정의한 예다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Target({ElementType.METHOD, ElementType.FIELD})  @Retention(RetentionPolicy.RUNTIME)  @Constraint(validatedBy=MemberNoValidator.class)  public @interface MemberNo {    } |

스프링의 Validator 는 PropertyEditor, ConversionService 와 마찬가지로 WebDataBinder 를 이용해 초기화할 수 있으므로 WebBindingInitializer 를 이용해 모든 컨트롤러에 일괄적으로 적용할 수도 있다. 특히 애노테이션을 활용하는 JSR-303 빈 검증 기능은 필드 단위로 적용이 가능하기 때문에 하나의 Validator 를 다양한 모델에 적용할 수 있다. 따라서 매번 컨트롤러의 @InitBinder 메소드에서 등록해주는 대신 WebBindingInitializer 를 이용해 일괄 적용하면 편리하다.

**BindingResult 와 MessageCodeResolver**

BindingResult 에는 모델의 바인딩 작업 중에 발생한 타입 변환 오류정보와 검증 작업에서 발생한 검증 오류 정보가 모두 저장된다. 이 오류 정보는 보통 컨트롤러에 의해 폼을 다시 띄울 때 활용된다. 폼을 출력할 때 BindingResult 에 담긴 오류 정보를 활용해서 에러 메시지를 생성할 수 있다. 스프링은 기본적으로 messages.properties 와 같은 프로퍼티 파일에 담긴 메시지를 가져와 에러 메시지로 활용한다. 앞에서 UserValidator 를 작성할 때 다음과 같이 에러 코드를 지정했던 것을 기억할 것이다.

|  |  |
| --- | --- |
| 1 | ValidationUtils.rejectIfEmpty(errors, "name", "field.required"); |

여기서 field.required 는 messages.properties 파일의 프로퍼티 키 값이다. messages.properties 파일에는 다음과 같이 선언해줘야 한다.

|  |  |
| --- | --- |
| 1 | field.required=필수 입력 항목입니다. |

물론 프로퍼티 파일은 영어 알파벳 외에는 유니코드로 변환해서 넣어야 하지만, 여기서는 이해를 돕기 위해 한글 메시지를 그대로 표시했다. 바인딩과 검증과정에서 오류가 발생하면 BindingResult 에 오류 정보가 등록된다.

이렇게 등록된 오류 정보에는 필드와 에러 코드가 기본적으로 들어간다. 이 에러코드는 MessageCodeResolver 를 통해 좀 더 상세한 메시지 키 값으로 확장된다. user 라는 이름을 가진 모델을 바인딩하고 검증하는 중에 name 이라는 필드 이름과 field.required 라는 에러코드가 BindingResult 에 등록됐다고 해보자. 스프링이 디폴트로 사용하는 MessageCodeResolver 인 DefaultMessageCodeResolver 는 이를 확장해서 다음과 같이 네 개의 메시지 키 후보를 생성한다.

1. 에러코드.모델이름.필드이름 : field.required.user.name

2. 에러코드.필드이름 : field.required.name

3. 에러코드.타입이름 : field.required.User

4. 에러코드 :  field.required

에러코드와 모델 오브젝트 이름, 필드 이름 또는 필드 타입의 이름을 결합해서 네 종류의 메시지 키 후보를 만든 것이다. 이 네 가지 메시지 키 후보는 위에서부터 우선적으로 메시지를 찾는데 사용된다. 다음과 같이 우선순위가 높은 field.required.user.name 이라는 메시지 키가 먼저 발견되면 field.required 는 무시된다.

field.required.user.name=이름은 반드시 입력해야 합니다.

field.required=필수 입력 항목입니다.

바인딩 작업 중에 타입 변환을 할 수 없어서 오류가 나는 경우가 있다. 이때는 검증기를 사용했을 때처럼 직접 에러 코드를 지정할 수 없다. 대신 스프링이 typeMismatch라는 에러 코드를 지정해준다. 이 경우에도 DefaultCodeResolver 가 만들어주는 네 가지 메시지 키 후보를 이용해 에러 메시지를 찾을 것이므로 그에 맞는 메시지를 messages.properties 프로퍼티 파일에 넣어주면 된다. typeMismatch 의 경우는 세 번째 조합에 모델 타입대신 필드 타입이 사용된다는 점만 주의하면 된다. BigDecimal 타입이라면 세 번째 후버 이름은 typeMismatch.java.lang.BigDecimal 로 만들어 진다.

검증 작업 중에 발생한 오류가 특정 필드에 속한 것이 아니라면 reject() 를 사용해 모델 레벨의 글로벌 에러로 등록할 수 있다. 이때는 필드 이름이 없으므로 에러코드와 모델이름을 이용해서 DefaultMessageCodeResolver 가 다음과 같은 두 개의 메시지 키 후보를 만들어 준다. user 모델의 검증 에러코드를 invalid.data 라고 지정했다면 다음과 같이 두 개의 키 후보를 얻을 수 있다.

1. 에러코드.모델이름 : invalid.data.user

2. 에러코드 : invalid.data

JSR-303 의 애노테이션을 이용한 검증 작업에서도 마찬가지로 에러 코드가 지정된다. 에러 코드는 애노테이션 이름으로 지정된다. @NotNull 이라면 NotNull, @Min 이라면 Min 이 에러코드로 사용된다. 필드에 적용된 것이라면 같은 방법으로 MessageCodeResolver 에 의해 네 가지 메시지 키 후보가 만들어 진다.

MessageCodeResolver 는 WebDataBinder 내부적으로 사용되므로 직접 이용할 일은 없다. 하지만 이를 통해 어떤 이름의 메시지 키를 만들어주는지 정확히 알고 있어야 에러 메시지가 담긴 프로퍼티 파일을 작성할 수 있다.

**MessageSource**

MessageCodeResolver 를 통해 여러 개의 후보로 만들어진 메시지 코드는 어떻게 해서 messages.properties 파일에 담긴 메시지를 가져올 수 있는 것일까? 사실 메시지 코드가 messages.properties 리소스 번들 파일을 사용하도록 고정된 것은 아니다. 이 메시지 코드는 MessageSourceResolver 를 한 번 더 거쳐서 최종적인 메시지로 만들어진다.

스프링의 MessageSource 구현은 두 가지 종류가 있다. 하나는 코드로 메시지를 등록할 수 있는 StaticMessageSource 이며, 다른 하나는 messages.properties 리소스 번들 방식을 사용하는 ResourceBundleMessageSource 다. 보통 메시지는 리소스 번들 파일에 저장해두고 관리하는 것이 편하므로 ResourceBundleMessageSource 가 기본적으로 사용된다. 애플리케이션을 배치하고 서버를 기동한 후에는 메시지를 수정할 일이 없지만, 때로는 서버 가동중에 애플리케이션 재시작 없이 메시지를 변경하고 싶을 때도 있다. 이때는 일정 시간마다 messages.properties 파일의 변경 여부를 확인해서 메시지를 갱신해주는 ReloadableResourceBundleMessageSource 를 적용하면 된다.

MessageSource 는 기본적으로 다음 네 가지 정보를 활용해 최종적인 메시지를 만들어 낸다.

**코드**

BindingResult 또는 Errors 에 등록된 에러 코드를 DefaultMessageCodeResolver 를 이용해서 필드와 오브젝트 이름의 조합으로 만들어낸 메시지 키 후보 값이 있다. 이 메시지 키 이름 후보가 MessageSource 의 첫 번째 정보인 코드로 사용된다. 네 가지 후보가 있다면 우선순위를 따라 차례로 적용해서 첫 번째 발견되는 메시지를 선택한다.

**메시지 파라미터 배열**

BindingResult 나 Errors 의 rejectValue(), reject() 에는 Object[] 타입의 세 번째 파라미터를 지정할 수 있다. messages.properties 에는 다음과 같이 메시지에 적용할 수 있는 파라미터 값을 지정할 수도 있다.

field.min={0} 보다 작은 값은 사용할 수 없습니다.

그리고 rejectValue() 에서 {0} 에 들어갈 파라미터 값을 Object 배열로 넣어 줄 수 있다. new Object[] { 100 } 이라고 했다면 최종 메시지는 "100 보다 작은 값은 사용할 수 없습니다." 라고 만들어 질 것이다. 메시지 파라미터는 하나 이상을 사용할 수 있기 때문에 Object 배열로 되어 있다.

**디폴드 메시지**

메시지 키 후보 값을 모두 이용해 messages.properties 를 찾아 봤지만 메시지가 발견되지 않으면 어떻게 될까? 이 때를 위해 디폴트 에러 메시지를 등록해 둘 수 있다. rejectValue() 의 네 번째 파라미터가 바로 이 디폴트 에러 메시지다. 다음은 name 필드의 에러 코드를 field.required 로 지정하면서 디폴트 메시지를 "입력해주세요" 라고 설정한 예다. messages.properties 파일에서 field.required 를 비롯한 네 가지 키에 해당하는 메시지를 찾을 수 없다면, 그때 디폴트 메시지가 사용된다.

rejectValue("name", "field.required", null, "입력해주세요");

에러 코드를 충실히 적용했다면 디폴트 메시지는 생략하거나 null 로 넣어주면 된다. 별로 권장하고 싶지는 않지만, 이와 반대로 모든 메시지를 디폴트로 코드에서 지정한 것만 사용하고 싶다면 에러코드도 null 로 하고 디폴트 메시지만 넣는 방법도 있다. messages.properties 파일을 관리하기 귀찮기 때문인데, 애플리케이션 규모가 커지면 코드를 뒤져서 에러 메시지를 수정하는 일이 더 힘들테니 에러 코드와 메시지 프로퍼티 파일을 활용하자.

**지역정보**

네 번째 정보는 LocaleResolver 에 의해 결정된 현재의 지역정보다. 리소스 번들을 활용했으니 당연히 지역에 따른 다른 메시지 프로퍼티 파일을 사용할 수 있다. 디폰트를 messages.properties 라고 했다면 Locale.KOREAN 지역에 대해서는 messages\_ko.properties를, Locale.ENGLISH 라면 messages\_en.properties 를 찾게 될 것이다. 지역에 해당하는 메시지 파일이 없다면 디폴트인 messages.properties 가 사용된다.

여러 언어를 지원하는 지역화 기능을 적용한 사이트라면 지역정보에 따라 다른 에러 메시지를 출력하는 것을 매우 간단히 만들 수 있다.

이 네가지 정보에 의해 MessageSource 가 최종 결정한 메시지가 폼에 나타날 것이다. 폼에서 어떻게 필드 에러 메시지나 글로벌 에러 메시지를 출력하는지는 이후에 알아보도록 하자.

모든 코드에 대해 일치하는 메시지를 찾을 수 없고 디폴트 메시지마저 지정되지 않았다면 메시지를 찾을 수 없다는 예외가 발생하니 주의해야 한다. 바인딩 타입 변경 오류로 인해 저징되는 typeMismatch나 JSR-303 애노테이션에 의해 생성되는 바인딩 에러의 경우는 디폴트 에러 메시지가 자동으로 등록된다. 하지만 이런 디폴트 메시지는 일반 사용자가 이해할 만큰 친절하지 못하므로 메시지를 직접 등록 지정해주는게 좋다.

메시지 소스는 디폴트로 등록되지 않는다. 따라서 사용하려면 다음과 같이 빈으로 등록해줘야 한다. 빈 아이디는 messageSource 를 사용해야 한다.

|  |  |
| --- | --- |
| 1  2  3  4 | <bean id="messageSource"      class="org.springframework.context.support.ResourceBundleMessageSource">      <property name="basename" value="messages" />  </bean> |

보통 메시지를 담을 프로퍼티 파일의 이름으로 messages.properties 를 사용한다. 여기서 messages 에 해당하는 부분이 basename 프로퍼티에 설정해야 하는 값이다. 만약 다른 이름을 사용하고 싶다면 변경할 수 있다.