Struts 처음 배우기

손권남
kwon37xi@yahoo.co.kr
http://kwon37xi.egloos.com
기준 버전 : Struts 1.2.7
작성일 : 2005/06/14 ~

목차

1. 스트럿츠는 왜 필요한가?

- **Model 1**

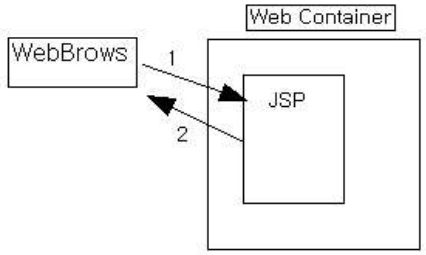
Model 1 방식의 웹 어플리케이션이란, 한 개의 JSP에서 모든 비지니스 로직(데이터베이스 쿼리, 업데이트 등 실제 업무 작업)을 수행하고, 그 결과를 바로 출력하는 방식이다. 현재 가장 쉽게 많이 사용되는 방식의 웹 프로그래밍 모델이다.

이 방식은 아주 단순한 웹 어플리케이션(JSP 10개 미만?)에서는 빠르고 단순하게 프로그램을 짤 수 있어 편하지만 어플리케이션이 커지게 되면 그 복잡도가 크게 증가하여 디버깅이 어렵고, 한 번 수정할 것이 생기면 수정할 위치를 찾기도 어려울 뿐만 아니라, 단순히 비지니스 로직이 바뀌는 것임에도 화면 출력 부분의 수정이 필요하게 된다(그 반대로 디자인 변경을 위해 프로그램 로직을 바꿔야 하는 경우도 많다).

이로인해 유지보수 비용과 시간이 증가하게 되고, 자바 웹 프로그래머의 밤샘의 원흉이 된다.

단순한 예로, Model 1 방식으로 짠 프로그램은 한 개의 JSP가 작업을 수행하는 자바 코드(스크립틀릿)과 화면에 출력되는 HTML을 포함하며 보통 수천 줄에 달하게 되는데, 거기에다 자바 코드가 HTML 코드 구석 구석에 숨어 있기 때문에 오류가 발생할 경우 오류를 고치는 시간보다는 오류가 발생한 부분을 찾는데 거의 모든 시간을 소모하게 되어 버린다. 오류를 찾는다 하여도 로직과 디자인 코드가 함께 있어 소스 코드 변경하기가 녹록치 않다.

또한, 한 개 파일의 프로그램과 디자인이 섞여 있기 때문에, 디자이너와 프로그래머의 업무 경계가 불명확해진다. 디자이너가 디자인 하나 바꿀 때마다 프로그래머가 없으면 아무것도 변경하지 못하는 사태가 발생하게 된다. 아니면 디자이너 스스로 프로그램을 짤 줄 알아야 하거나.



삽화 1 Model 1 개요

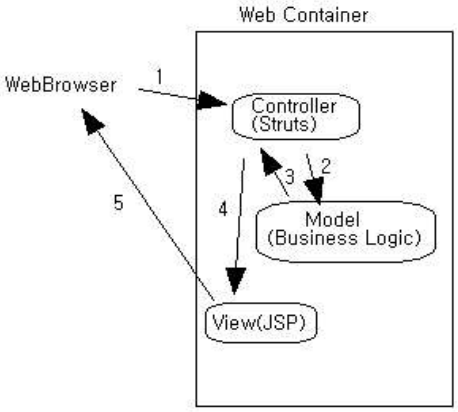
- **Model 2**

그래서 등장한 것이 Model 2 이다. **모델 2란 실제 프로그램 수행 부분과 화면에 결과를 뿌려주는 부분을 서로 분리 하는 것**이다.

간단하게 모든 요청을 서블릿이 받고, 비지니스 로직을 서블릿에서 수행하고, 결과 출력을 위해 JSP로 포워딩하도록 해도 어플리케이션 복잡도가 많이 감소하게 된다.

하지만 그렇게 할 경우 서블릿의 갯수가 지나치게 늘어나고, web.xml 이 복잡하고 어지러워지게 된다.

그래서 모델 2 그 중에서도 Model 2를 MVC(Model-View-Controller) 패턴으로 설계하는 것이 주류로 자리잡게 되었다.



삽화 2 Model 2 개요

1. Model 2 는 사용자의 요청을 Controller가 받는다. 컨트롤러는 사용자의 요청을 분석하여 할 일이 뭔지를 판단하고, 실제로 비지니스 로직을 수행하는 자바 클래스를 호출한다.
2. 비지니로 로직을 수행하는 부분을 모델이라고 부른다. 이 모델에서 데이터베이스 쿼리 및 업데이트 등을 수행한다. 그리고는 그 수행 결과를 컨트롤러가 request나 session, application 웹 서블릿 스코프 객체들에게 setAttribute() 메소드를 이용해 저장한다.
3. 그리고 모델은 실행을 종료 한 뒤에 컨트롤러로 복귀한다.
4. 컨트롤러는 설정파일을 통해서 모델의 실행 결과를 HTML로 뿌려줄 View(즉, JSP 페이지)로 포워딩을 한다.
5. View JSP는 getAttribute()로 컨트롤러가 저장한 수행 결과를 얻어와 이를 화면에 출력해준다.

이렇게 해서 Model-View-Controller 로 각각의 역할을 구분한 것이 Model 2 이다.

- **MVC 패턴의 구현**

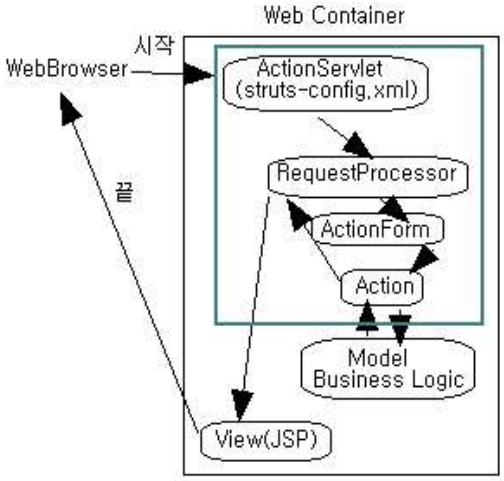
  그렇다면 MVC 패턴으로 프로그램을 어떻게 짜는가 하는 문제가 남았다.

  - MVC 패턴에서 **Model** 부분은 자바 클래스와 자바 빈이 맡는다. Model은 데이타베이스 등을 액세스 하는 자바 클래스와 그 결과 데이타를 저장하는 Java Beans로 구성되는 것이다.
  - **View**는 JSP가 맡는다. JSP는 뭐 다 아니까 넘어가기~
  - **Controller**가 문제이다. 컨트롤러는 Servlet을 기반으로 직접 개발자가 만들어서 사용할 수도 있고 이미 많이 나와 있는 오픈 소스 프레임워크를 사용해도 된다.

- **스트럿츠?**

  스트럿츠(http://struts.apache.org)는 이 MVC 패턴에서 Controller 역할을 하는 웹 어플리케이션 프레임워크로서 가장 유명하고 널리 쓰이는 것이다.

  스트럿츠의 작동구조를 MVC에 대입하여 비교해 보자.



삽화 3 스트럿츠 개요

스트럿츠의 구조는 MVC 패턴 그림에 비해 약간 복잡해 보이지만 사실 MVC 패턴의 범주를 벗어나지 않는다.

웹 컨테이너 안의 작은 직사각형 부분에 있는 **ActionServlet, RequestProcessor, ActionForm, Action 이 모두 스트럿츠에 해당하는 부분이며, 모두 컨트롤러의 역할**을 하게 된다.

Model과 View는 MVC 패턴 소개에서 본 대로 컨트롤러인 스트럿츠와는 상관없이 움직이게 된다.

- **ActionServlet** : 어떤 Action(비지니스 로직을 호출하는 역할을 하는 자바 클래스)를 사용할지 여부 등 스트럿츠 전반에 대한 설정을 저장하고 있는 struts-config.xml 파일을 읽어 저장하는 Servlet 클래스이다.
- **RequestProcessor** : 사용자의 요청을 받으면, struts-config.xml 설정에 따라 실제로 호출할 Action을 선택하고, request와 response 서블릿 객체를 이용해서 미리 선행작업을 하는 역할을 한다.

  Action을 선택하는 방법은 바로 요청 URL에 있다.

  예를 들어 사용자가 http://localhost:8080/**login.do**를 호출 했다면 "/login.do"를 인지하고 struts-config.xml에서 "/login"이라는 이름(".do" 제외)으로 지정된 행동 지침관련 설정(<action>)을 찾아서 그에 따라 비지니스 로직을 실행시킨다.
- **ActionForm** : 사용자가 GET 혹은 POST 방식으로 넘긴 파라미터들을 저장하고 분석하여 알맞은 값이 들어왔는지 여부 등을 판단(유효성 검사)하는 Java Bean 이다. ActionForm은 있어도 되고 없어도 되며 ActionForm에서 각 파라미터가 유효성 검사를 통과하지 않으면 Action을 호출하지 않는다.
- **Action** : 비지니스 로직을 호출하는 부분이다. 실질적으로 자바 웹 서블릿과 같은 역할을 한다. 비지니스 로직은 Action에서 직접 수행하지 말고 따로 Model 전용 클래스로 만들어서 파라미터들을 넘겨 호출만 하는 형태로 작성한다.

스트럿츠는 이 이외에도 아주 다양한 기능을 수행한다. 내가 생각하는 스트럿츠의 단점은 기능이 너무 다양하다는 것이다. 그리고 그로인한 설정 파일의 복잡도도 높은 편이다. 개발자는 그 모든 기능을 익히려고 욕심을 부리다가 아무것도 못하는 포기하는 상태에 빠질 수도 있다.

이 문서는 꼭 필요하다고 생각되는 기능들로만 각 단위별 예제와 함께 설명한다.

그 외의 더 많은 다양한 기능들은 잘 짜여진 책을 참고로 하여 학습하도록 한다.

2. 스트럿츠 기반 웹 어플리케이션 구성하기

http://struts.apache.org 에서 스트럿츠 프레임워크 바이너리를 다운로드 하면 된다.

- 웹 어플리케이션 디렉토리 구조로 새 어플리케이션을 구성한다.
- WEB-INF/lib 디렉토리에 스트럿츠 바Batang이너리 압축파일에 있는 struts.jar와 기타 lib/*.jar 파일들을 넣는다.
- WEB-INF/config/struts-config.xml 파일을 생성한다.
- WEB-INF/tlds/ 에 스트럿츠 바이너리 파일에 포함된 각 *.tld 파일을 복사한다.
- struts-config.xml 과 *.tld 파일은 WEB-INF/ 디렉토리에 두어도 된다.
- WEB-INF/web.xml 파일을 스트럿츠를 사용할 수 있도록 수정한다.
- 스트럿츠 바이너리 파일을 보면 struts-blank.war 파일이 있는데, 이 파일은 기본적인 스트럿츠 설정이 된 상태의 웹 어플리케이션 구조를 미리 만들어둔 예제 파일이다. 스트럿츠 어플리케이션을 새로 만들기 시작할 때 압축을 풀어서 사용하면 위의 설정이 다 된상태가 된다. 단, config 디렉토리와 tlds 디렉토리를 따로 구분하지는 않아 뒀기 때문에 따로 생성하고 파일을 각각의 위치에 옮기기를 권장한다 (개인적으로 WEB-INF/ 디렉토리에 많은 파일 두는것을 싫어한다.
- 스트럿츠를 위한 기본적인 web.xml 구성. 아래는 J2EE 1.3, JSP 1.2, Servlet 2.3 스펙에 기반한 web.xml 파일이다. 모든 스트럿츠 어플리케이션은 다음과 같은 내용의 web.xml을 가지고 있어야만 한다.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <display-name>스트럿츠 어플리케이션</display-name>

 <!– ActionServlet를 등록한다. –>
 <servlet>
   <servlet-name>action</servlet-name>
   <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
   <init-param>
     <param-name>config</param-name>
     <param-value>/WEB-INF/config/struts-config.xml</param-value>
   </init-param>
   <init-param>
     <param-name>debug</param-name>
     <param-value>2</param-value>
   </init-param>
   <init-param>
     <param-name>detail</param-name>
     <param-value>2</param-value>
   </init-param>
   <!– ActionServlet은 이 웹 어플리케이션 시작시에 함께 시작되어야 한다. –>
   <load-on-startup>1</load-on-startup>
 </servlet>

 <!– “*.do”로 끝나는 모든 URL 패턴은 ActionServlet을 거쳐서 수행되어야 한다. –>
 <servlet-mapping>
   <servlet-name>action</servlet-name>
   <url-pattern>*.do</url-pattern>
 </servlet-mapping>

 <!– 웹으로 접속한 사용자가 JSP 파일로 직접 접근할 수 없게 한다. –>
 <security-constraint>
   <web-resource-collection>
     <web-resource-name>PreventViewingJSPs</web-resource-name>
     <description>웹으로 접속한 사용자가 JSP파일로 직접 접근할 수 없도록 한다.</description>
     <url-pattern>*.jsp</url-pattern>
```

```
    <http-method>GET</http-method>
    <http-method>POST</http-method>
   </web-resource-collection>
   <auth-constraint>
    <role-name></role-name>
   </auth-constraint>
  </security-constraint>
</web-app>
```

- <servlet-mapping>의 <url-pattern>*.do</url-pattern> 에 의해서 현재 웹 컨텍스트의 **“.do”로 끝나는 모든 요청이 ActionServlet을 거쳐서 가게 된다**.
- ActionServlet으로 진입한 사용자의 요청은 ActionServlet의 파라미터로 지정한 struts-config.xml의 설정에 따라 지정된 작업을 수행하게 된다.
- <security-constraint> 부분으로 인해서 **어떠한 사용자도 *.jsp 파일에 직접 접근할 수 없다.** 모두 스트럿츠 컨트롤러를 통해서만 JSP 파일에 접근할 수 있도록 제약한다.
- struts-config.xml 의 기본 구성

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

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
          <data-sources />
          <form-beans />
          <global-exceptions />
          <global-forwards />
          <action-mappings />
          <controller />
          <message-resources />
          <plug-in />
</struts-config>
```

struts-config.xml 파일은 위와 같은 구성에 각 요소별로 상세 설정이 더 들어가게 된다. 기본 설정 요소들은 **항상 위의 순서대로 작성**되어야 한다.

예를들어 <action-mappings />관련 설정은 항상 <global-forwards />보다 뒤에 나와야만 한다. 각 요소들 중 불필요한 것을 일부러 빈 값으로 놔둘 필요 없이 그냥 아예 없애버려도 된다. 하지만 일단 기입하면 위의 순서에 따라야만 한다(이것은 XML DTD의 특징이며 Servlet Spec 2.3 이하의 web.xml 에서도 마찬가지로 적용된다).

이 문서에서는 <data-sources /> 설정을 제외한 모든 설정을 사용한다.

3. 스트럿츠 어플리케이션 시작하기

웹 어플리케이션의 컨텍스트패스는 각 어플리케이션 별로 다르다. 컨텍스트 패스가 들어가는 부분은 “ContextPath”로 해 놓았다. 이 부분은 각자의 상황에 맞게 바꾼다.

• 단순히 JSP 페이지로 포워딩하기

스트럿츠에서는 기본적으로 모든 JSP가 직접 사용자에 의해 액세스 되는 것을 금기시 하고 있다. 그러므로 Model 부분이 필요 없이 단순히 화면에 HTML만 출력하는 JSP도 컨트롤러를 거쳐서 사용자에게 전달 되도록 하는 것이 좋다.

그리고 우리는 이미 web.xml 에서 일반 사용자가 *.jsp 페이지를 호출할 수 없도록 막아버렸다(<security-constraint /> 이용). 스트럿츠 컨트롤러를 통하지 않고는 누구도 *.jsp 를 호출할 수 없다.

여기서는 단순한 HTML만 출력하는 JSP를 컨트롤러를 거쳐서 가도록 만들어본다.

- WEB-INF/config/struts-config.xml 을 다음과 같이 작성한다.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
<!– 앞으로는 상단 제외하고 예제를 싣는다. –>
<struts-config>
   <action-mappings>
      <action
         path="/Welcome"
         forward="/Welcome.jsp"/>
   </action-mappings>
</struts-config>
```

이 예제에서는 Model을 호출하는 Action 부분을 따로 작성하지 않고 단지 지정된 JSP로 포워딩 하도록 하고 있다.

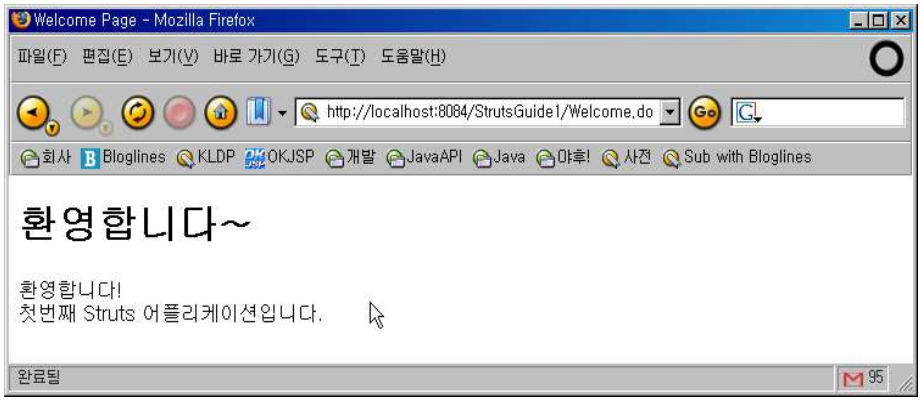
- <action> 요소에서 path=”/Welcome” : URL에 http://localhost:8080/ContextPath/Welcom.do 처럼 호출하면 지정된 액션을 수행한다고 지정했다. **URL의 “.do”는 path 속성에 지정하지 않는다.**
- <action> 요소에서 forward=”/Welcome.jsp” : 특별한 비지니스 로직 수행작업을 하지 않고 곧바로 Welcome.jsp로 포워딩을 한다.

- Welcome.jsp 을 작성하여 웹 어플리케이션 디렉토리에 둔다.

```
<%@page contentType="text/html; charset=euc-kr"%>
<!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=euc-kr">
      <title>Welcome Page</title>
   </head>
   <body>

   <h1>환영합니다~</h1>
   환영합니다!<br />
   첫번째 Struts 어플리케이션입니다.
   </body>
</html>
```

이 어플리케이션에서는 Action이 실질적으로 아무런 역할도 수행하지 않고 바로 JSP페이지로 포워딩을해서 내용을 출력한다.

그 결과를 보면 다음과 같다.



첫번째 스트럿츠 어플리케이션을 완성했다.

- **Forward와 Redirect의 차이**

JSP/Servlet에는 현재 작업중인 페이지에서 다른 페이지로 이동하는 두 가지 방식의 페이지 전환 기능이 있다. 하나는 Forward이고 하나는 Redirect이다.

둘 다 다른 웹 페이지로 이동하지만 행동 양태가 다르다.

- **Forward** : Web Container 차원에서 페이지 이동만 있다. 실제로 웹 브라우저는 다른 페이지로 이동했음을 알 수 없다. 그렇기 때문에, 웹 브라우저에는 최초에 호출한 URL이 표시되고 이동한 페이지의 URL 정보는 볼 수 없다. 동일한 웹 컨테이너에 있는 페이지로만 이동할 수 있다. 현재 실행중인 페이지와 Forwad에 의해 호출될 페이지는 **request와 response 객체를 공유**한다.
- **Redirect** : Web Container는 Redirect 명령이 들어오면 웹 브라우저에게 다른 페이지로 이동하라고 명령을 내린다. 그러면 웹 브라우저는 URL을 지시된 주소로 바꾸고 그 주소로 이동한다. 다른 웹 컨테이너에 있는 주소로 이동이 가능하다. 새로운 페이지에서는 **request와 response객체가 새롭게 생성**된다.

- **Action을 사용하는 기본적인 예제**

이번에는 컨트롤러에서 모델을 호출하는 예제를 살펴보자.

비지니스 로직(Model)을 호출하는 역할을 하는 Action 클래스를 작성해 보도록 한다.

Action 클래스는 **"org.apache.struts.action.Action"** 클래스를 상속받아서 작성하며, 실제 프로그램 수행은 "**execute**()" 메소드에 의해 이뤄진다.

- Action 클래스의 역할
  - 사용자가 GET/POST 방식으로 넘겨준 파라미터들을 분석한다(이것은 Action이 호출 되기 전에 ActionForm에 위임할 수 있다).
  - 자신이 수행할 비지니스 로직을 구현한 Model Java Class를 호출한다.
  - Model Java 클래스가 수행한 결과를 넘겨 받는다.
  - 그 결과를 request, session, application 등의 스코프에 저장하고, View 로 사용될 페이지를 지정한뒤에 리턴한다.
  - RequestProcessor가 Action에 의해 지정된 페이지로 포워딩 혹은 리다이렉트 한다.
- Action 클래스는 기본적으로 다음과 같은 형태로 만들어진다.

```
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionForm;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ActionClass extends Action {
   // execute 메소드를 필히 구현해야 한다.
```

```
    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception {

            // 실제 작업 수행..

        return forward;
    }
}
```

자, 이제 사용자를 로그인 시키는 스트럿츠 자바 어플리케이션을 생성해 본다.

- WEB-INF/config/struts-config.xml 을 다음과 같이 작성한다.

```
<!– 상단 생략 –>
<struts-config>
    <action-mappings>
        <action
            path="/Welcome"
            forward="/Welcome.jsp"/>
        <!– 로그인 폼으로 이동하는 Forward 액션 –>
        <action
            path="/login1/logInForm"
            forward="/login1/logInForm.jsp"/>

        <!– 로그인을 실행하는 Action –>
        <action
            path="/login1/logIn"
            type="strutsguide.actions.Login1Action"
            validate="false"
            >
            <!– 로그인을 수행한 뒤에 성공/실패 표시를 위해 이동할 View 페이지 –>
            <forward name="success" path="/login1/logInSuccess.jsp" redirect="true" />
            <forward name="fail" path="/login1/logInFail.jsp" />
        </action>

    </action-mappings>
</struts-config>
```

- /login1/logInForm.jsp 를 다음과 같이 작성한다.

```
<%@page contentType="text/html; charset=euc-kr"%>
<!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=euc-kr">
        <title>로그인 폼</title>
    </head>
    <body>

    <h1>로그인 폼</h1>
    <!– 폼의 데이터를 "login.do"로 전송한다. –>
    <form name="login" method="get" action="login.do">
        Username : <input name="username" type="text" size="16" /><br />
        Password : <input name="password" type="password" size="16" /><br />
        <input type="submit" />
    </form>
    </body>
</html>
```

- /login1/logInSuccess.jsp 를 다음과 같이 작성한다.

```
<%@page contentType="text/html; charset=euc-kr"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<%– Action에서 Session 객체에 저장한 userInfo 자바 빈 객체를 사용한다. –%>
<jsp:useBean class="strutsguide.beans.UserInfoBean" id="userInfo" scope="session"/>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=euc-kr">
      <title>로그인 성공</title>
   </head>
   <body>

   <h1>로그인 성공</h1>

   로그인 사용자명 : <jsp:getProperty name="userInfo" property="userName"/><br />
   전화번호 : <jsp:getProperty name="userInfo" property="phone"/><br />
   이메일 : <jsp:getProperty name="userInfo" property="email"/><br />
   </body>
</html>
```

- /login1/logInFail.jsp를 다음과 같이 작성한다.

```
<%@page contentType="text/html; charset=euc-kr"%>
<!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=euc-kr">
      <title>로그인 실패</title>
   </head>
   <body>

   <h1>로그인 실패</h1>
   로그인에 실패했습니다.<br />
   <a href="logInForm.do">다시 로그인</a>
   </body>
</html>
```

- 실제 로그인하는 역할을 하는 Model 클래스인 LogInProcess.java를 작성한다.

  이 클래스는 임의로 HashMap에 사용자 정보와 비밀번호를 넣고, 사용자가 logInForm.do를 통해 입력한 정보를 대입하여 정확한 사용자명과 비밀번호라면 사용자의 정보 자바 빈 객체를 리턴하고, 그렇지 않다면 null을 리턴한다.

  이 클래스는 객체가 여러개 생성될 이유가 없기 때문에 Singleton 패턴으로 작성하여 단 한개의 객체만 생성되도록 하였다.

```
package strutsguide.beans;

import java.util.HashMap;

/**
 * 로그인을 실제로 수행하는 비지니스 로직 클래스
 */
public class LogInProcess {

   /** 사용자 정보를 담고 있는 해시맵 */
   private HashMap userInfos = null;

   /** 사용자의 비밀번호를 담고 있는 해시맵 */
   private HashMap userPasswords = null;

   /** 유일한 LogInProcess 객체 */
   private static LogInProcess instance = new LogInProcess();
```

```
    /** 싱글턴으로 오로지 객체 한 개만 생성한다. 생성자 */
    private LogInProcess() {
        userPasswords = new HashMap();
        userInfos = new HashMap();

        // 가상의 사용자를 추가한다.
        // 원하는 대로 추가할 수 있다.
        userPasswords.put("kwon37xi", "1234");
        userInfos.put("kwon37xi", new UserInfoBean("kwon37xi", "011-222-4444", "kwon37xi@yahoo.co.kr"));

        userPasswords.put("narae", "n111");
        userInfos.put("narae", new UserInfoBean("narae", "010-333-5555", "narae@mymail.xxx"));

        userPasswords.put("woori", "w222");
        userInfos.put("woori", new UserInfoBean("woori", "010-777-9999", "woori@yourmail.ooo"));
    }

    /**
     * LogInProcess의 유일한 객체를 얻는다.
     */
    public static LogInProcess getInstance() {
        return instance;
    }

    /**
     * 로그인을 수행하고, 성공하면 해당하는 사용자의 UserInfoBean 객체를 리턴한다.
     *
     * @param userName 로그인할 사용자명
     * @param password 비밀번호
     * @return UserInfoBean 로그인한 사용자의 정보. 로그인 실패하면 null 리턴
     */
    public UserInfoBean logIn(String userName, String password) {
        String userPassword = (String)userPasswords.get(userName);

        // 사용자가 존재하지 않으면 null 리턴
        if (userPassword == null) {
            return null;
        }

        // 비밀번호가 일치하지 않아도 null 리턴
        if (!userPassword.equals(password)) {
            return null;
        }

        UserInfoBean userInfo = (UserInfoBean)userInfos.get(userName);
        return userInfo;
    }
}
```

- 사용자 정보를 저장하는 모델 자바 빈인 UserInfoBean.java를 작성한다.

```
package strutsguide.beans;

public class UserInfoBean {
    /** 사용자명 */
    private String userName = null;

    /** 사용자의 전화번호 */
    private String phone = null;

    /** 사용자의 이메일 */
    private String email = null;
```

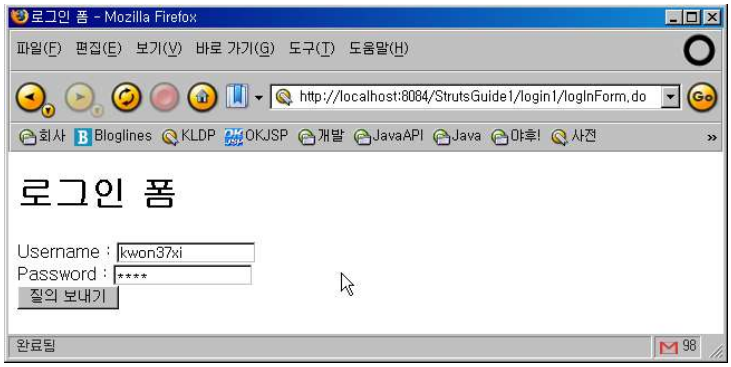
```
    /** 기본 생성자 */
    public UserInfoBean() {
        // do nothing;
    }

    /** 사용자 정보를 받는 생성자 */
    public UserInfoBean(String userName, String phone, String email) {
        this.userName = userName;
        this.phone = phone;
        this.email = email;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

- 모델 클래스를 호출하여 실제 작업을 수행하는 액션 클래스인 LogIn1Action.java을 작성한다.

```
package strutsguide.actions;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionForm;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import strutsguide.beans.UserInfoBean;
import strutsguide.beans.LogInProcess;

public class Login1Action extends Action {

    /**
     * 액션을 수행한다.
     */
    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        LogInProcess logInProcess = LogInProcess.getInstance();

        // 사용자가 전달한 파라미터를 저장한다.
```

```
        String userName = request.getParameter("username");
        String password = request.getParameter("password");

        UserInfoBean userInfo = logInProcess.logIn(userName, password);

        ActionForward forward = null;

        if (userInfo == null) {
            // userInfo가 null 이면 로그인이 실패한 것이다. 실패 페이지로 이동한다.

            // 이동할 페이지를 찾는다.
            forward = mapping.findForward("fail");
        } else {
            // null이 아니면 성공 페이지로 이동한다. 이동하기 전에 사용자 정보를 session에 저장한다.
            HttpSession session = request.getSession();
            session.setAttribute("userInfo", userInfo);

            // 이동할 페이지를 찾는다.
            forward = mapping.findForward("success");
        }

        // 비지니스 로직을 마치고 View 페이지로 이동하도록 지시한다.
        return forward;
    }
}
```

이제, 실행 해 보자. http://localhost:8080/ContextPath/login1/logInForm.do 를 호출하면 struts-config.xml의 <action path="/login1/logInForm" forward="/login1/logInForm.jsp"/>에 의해 login1/logInForm.jsp 으로 포워딩 되면서 프로그램을 시작할 수 있다.



삽화 4 로그인 폼

여서서 LogInProcess.java에 지정한 올바른 사용자명과 비밀번호를 입력하면 logInForm.jsp의 <form action="logIn.do">에 의해 Login1Action 액션 클래스로 제어가 넘어가게 된다.

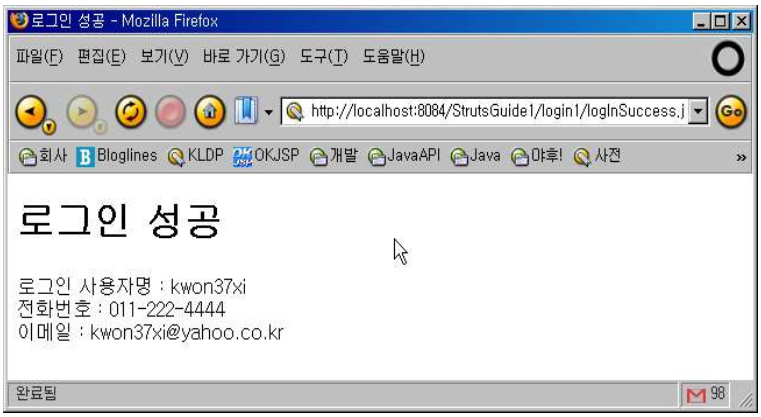
Login1Actioin 클래스는 사용자의 파라미터를 분석하고, 그 파라미터를 LogInProcess 의 logIn() 메소드에 넘겨 올바른 사용자인지 검사한다.

검사가 성공적이면 mapping.findForward("success"); 에 의해 사용자가 다음 struts-config.xml에 지정한 성공했을 때 이동할 페이지로 포워딩을 한다.

```
    <!- 로그인을 실행하는 Action ->
    <action
        path="/login1/logIn"
        type="strutsguide.actions.Login1Action"
        validate="false"
        >
        <!- 로그인을 수행한 뒤에 성공/실패 표시를 위해 이동할 View 페이지 ->
        <forward name="success" path="/login1/logInSuccess.jsp" redirect="true" />
        <forward name="fail" path="/login1/logInFail.jsp" />
    </action>
```

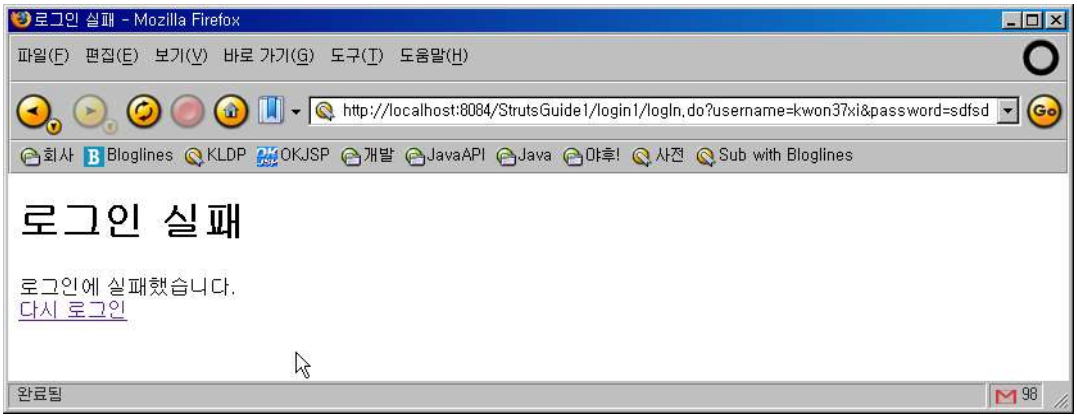
저기서는 성공했을 때 "success"라는 문저열로 포워딩할 정보를 찾으면 "/login1/logInSuccess.jsp" 페이지로 리다이렉트(redirect="true")하도록 지정했다. 이렇게 리다이렉트를 하면 다음과 같이 jsp 페이지를 URL에

서 직접 볼 수 있게 된다.(좋지 않다. 다음 번에는 리다이렉트 되는 페이지도 모두 컨트롤러를 통해 가도록 수정해 보자.) -- 이 부분을 작성할 때는 web.xml 에 *.jsp 접근 금지관련 설정을 하지 않았었다. 그 부분을 주석처리하고 실행해 볼 것.



삽화 5 로그인 성공

실패하면 “fail”이라는 포워딩 정보에 의해서 “/login1/logInFail.jsp”로 포워딩을 하게 된다. 리다이렉트가 아닌 포워딩을 하므로 브라우저에 보이는 URL은 변함없이 “/login1/login.do”가 된다.



삽화 6로그인 실패

저기 보면 URL이 login.do 인것을 볼 수 있다.

그리고 노파심에서 말하자면 form의 method를 “GET”으로 지정한 것은 읽는이에게 정보 전달 과정을 보여주기 위해서일 뿐이다. 실제로 로그인 하는 폼 처럼 비밀번호 같은 숨겨야할 정보가 있는 폼은 “POST” 방식으로 만들어야 한다.

첫 번째 MVC 패턴 기반의 웹 어플리케이션을 만들어 보았다.

4. 제대로 사용해보기

스트럿츠의 기본적인 흐름을 다 익혔다. 이제 스트럿츠를 본격적으로 사용해 보자. 아래는 사실상 스트럿츠의 뼈대가 되는 거의 모든 기능을 사용하는 예제이다.

스트럿츠의 기본 구조를 설명할 때 스트럿츠는 Action을 실행하기 전에 ActionForm 이라는 것을 거치면서 사용자가 입력한 파라미터들이 유효한지 여부를 검사할 수 있다고 했다.

ActionForm에서 파라미터들이 유효하지 않다고 판단하면 Action까지 가지 않고 곧바로 오류 메시지를 보여주며 사용자에게 다시 입력하도록 할 수 있다.

여기서 오류 메시지를 뿌려주는 역할이 ActionMessages의 역할이다. ActionMessages는 ActionForm에서 난 오류 뿐만 아니라 Action 클래스에서 난 오류나 혹은 일반 메시지도 저장하고 있다가 뷰 JSP에서 출력해 줄 수 있다.

- **ActionForm의 역할은**
  - HTML 폼("<input type='text' name='phone'/>" 형식)으로 부터 받은 입력은 ActionForm 빈으로 전달되고(ActionForm 객체에 대해 **setPhone(request.getAttribute("phone")** 과 같은 호출이 일어난다), 프라퍼티(파라미터)에 대한 검증이 끝나면 폼으로 부터 받은 입력 값을 잘 정돈된 자바빈 데이타로 만들어 Action 에 전달한다.
  - **주의:** ActionForm 객체를 직접 이용해서 작업을 수행(비지니스 프로세스 수행 - 모델 부분)을 해서는 안된다! - **모델 부분은 컨트롤러와 뷰와 완전히 분리된 상태로 작성**해야 한다. ActionForm은 컨트롤러에 속한다.
- **ActionForm 을 구현하려면..**
  - org.apache.struts.action.ActionForm 클래스를 상속 받아야 한다.
  - **각 프라터티는 HTML Form의 Input의 name과(<input name=""/>) 같은 이름**을 가져야 한다.
  - **각 프라퍼티별로 Setter와 Getter**가 있으면 된다. Setter와 Getter가 꼭 둘 다 있을 필요는 없다.
  - 각 프라퍼티는 되도록 **String 과 boolean 형**으로 만든다. 잘못 입력한 데이터를 다시 사용자의 입력화면에 보여주려면 잘 못 입력한 데이타가 String 으로 보전되어 있어야 하기 때문이다. ActionForm의 프라퍼티를 int 등의 형으로 만들면 사용자가 잘못 입력한 데이터 중에서 숫자가 아닌 부분이 모두 사라지게 되기 때문에 무엇을 어떻게 잘 못 입력했는지 알 수 없게 된다.
  - 각 프라퍼티들을 ActionForm 객체에 채우기 전에 먼저 초기화 작업을 거치고 싶다면 "public void reset (ActionMapping mapping, HttpServletRequest request)" 메소드를 구현해야 한다.
  - 파라미터 값을 ActionForm에서 Action으로 전달하기 전에 유효성 검증 과정을 거치려면 **"public ActionErrors validate(ActionMapping mapping, HttpServletRequest request)"** 메소드를 구현해야 한다. (ActionErrors를 리턴한다는 것 주의)
  - struts-config.xml에 다음과 같은 부분을 추가하여, ActionForm을 등록한다. 등록된 ActionForm은 여러 Action에서 사용될 수도 있다.

```
<struts-config>
   <form-beans>
      <form-bean
         name="폼의이름"
         type="myproject.form.FormClass"/>
   </form-beans>
</struts-config>
```

- **ActionMessages의 역할은...**

ActionMessages의 역할은 기본적으로 컨트롤러(그 중에서도 ActionForm과 Action)에서 수행도중 발생한 오류나 기타 메시지들을 저장하고 있다가 뷰 단에서 보여 줄 수 있도록 하는 역할이다.

이에 대해서는 예제를 본 후 살펴본다.

이제, ActionForm과 ActionMessages를 기본적으로 사용한 예제를 만들어 본다. 바로 전에 만들었던 login1을 개선해서 로그인을 하고, 만약 사용자가 넘긴 username과 password 파라미터가 빈 값이거나 공백등을 포함한다면 ActionForm에서 오류가 발생하여 다시 로그인 폼으로 돌아가서 잘못된 부분에 대한 에러 메시지를 출력하도록 변경한다.

또한 로그아웃 기능도 만들고, 로그인 폼은 POST 방식으로 전달되도록 하며, 몇몇 JSP 페이지들은 Action이 불필요하더라도 모든 JSP가 스트럿츠 컨트롤러를 통해 보여질 수 있도록 설정한다.

- WEB-INF/config/struts-config.xml 을 다음과 같이 작성한다. 기존의 login1을 위한 정보들도 그냥 남겨두었다. 동일한 웹 컨텍스트이기 때문이다. 굵은 글씨가 바뀐 부분이다.

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

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
        <form-beans>
                <!– Login2를 위한 ActionForm 설정 –>
                <form-bean name="login2Form"
                        type="strutsguide.forms.Login2Form">
                </form-bean>
        </form-beans>

        <global-forwards>
                <!– Login2 의 로그인 폼 화면으로 이동하기 위한 global forward –>
                <forward name="login2Form" path="/login2/loginForm.do"
                        redirect="true" />
        </global-forwards>

        <action-mappings>
                <action path="/Welcome" forward="/Welcome.jsp" />

                <!– // Login2를 위한 Action Mapping 시작 –>
                <!– 로그인 폼 화면 출력 액션 –>
                <action path="/login2/loginForm"
                        forward="/login2/loginForm.jsp" />

                <!– 로그인을 수행하는 액션 –>
                <action path="/login2/login"
                        type="strutsguide.actions.Login2Action" name="login2Form"
                        validate="true" scope="request" input="/login2/loginForm.jsp">
                </action>

                <!– 로그 아웃 수행 액션 –>
                <action path="/login2/logout"
                        type="strutsguide.actions.LogoutAction" />

                <!– // Login2 를 위한 Action Mapping 끝 –>

                <action path="/login1/logInForm"
                        forward="/login1/logInForm.jsp" />

                <action path="/login1/logIn"
                        type="strutsguide.actions.Login1Action" validate="false">
                        <forward name="success" path="/login1/logInSuccess.jsp"
                                redirect="true" />
                        <forward name="fail" path="/login1/logInFail.jsp" />
                </action>

        </action-mappings>

        <!–
```

```
                login2에서 메시지 출력에 사용할 프라퍼티 파일.
                strutsguide.resource 패키지의 application.properties 파일임을 의미한다.
        -->
        <message-resources parameter="strutsguide.resources.application" />

</struts-config>
```

- /login2/loginForm.jsp 를 다음과 같이 작성한다.

```
<%@page contentType="text/html; charset=euc-kr"%>
<%@taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>
<%@taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean"%>
<%@taglib uri="/WEB-INF/tlds/struts-logic.tld" prefix="logic"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=euc-kr">
      <title>로그인 폼</title>
   </head>
   <body>

   <h1>로그인 폼</h1>

        <logic:present name="userInfo" scope="session">
                <%-
                        session 스코프에 "userInfo" 라는 Attribute가 존재하면 이 부분이 실행된다.
                -%>
                <b>이미 로그인 하셨습니다.</b><br />
                로그인한 사용자명 : <bean:write name="userInfo" property="userName" scope="session"/><br />
                전화번호 : <bean:write name="userInfo" property="phone" scope="session"/><br />
                이메일 : <bean:write name="userInfo" property="email" scope="session"/><br />
                <html:link action="/login2/logout">로그아웃하기</html:link>
        </logic:present>

        <logic:notPresent name="userInfo" scope="session">
                <%-
                        session 스코프에 "userInfo"라는 Attribute가 존재하지 않으면 이 부분이 실행된다.
                -%>
                <b>로그인 하십시오.</b><br/>
        </logic:notPresent>

   <!- 폼의 데이터를 "login.do"로 전송한다. ->
   <html:form action="/login2/login" method="POST" focus="username">

        <html:messages id="msg" message="true">
                <%-
                        ActionMessages.GLOBAL_MESSAGE 키로 저장된 ActionMessage 객체가 없다면
                        이 부분은 실행되지 않는다.
                -%>
                <b><bean:write name="msg"/></b> <br />
        </html:messages>

      Username : <html:text property="username"/>
        <html:messages id="msg" property="invalidUsernameError">
                <%-
                        invalidUsernameError 라는 키로 저장된 ActionMessage 객체가 없다면
                        이 부분은 실행되지 않는다.
                -%>
                <b><bean:write  name="msg"/></b>
        </html:messages>
        <br />
```

```
    Password : <html:password property="password" redisplay="false"/>
        <html:messages id="msg" property="invalidPasswordError">
                <%–
                        invalidPasswordError 라는 키로 저장된 ActionMessage 객체가 없다면
                        이 부분은 실행되지 않는다.
                –%>
                <b><bean:write name="msg"/></b>
        </html:messages>
                <br />

    <html:submit value="로그인"/>   <html:reset value="초기화" />

  </html:form>
  </body>
</html:html>
```

- 실제 로그인하는 역할을 하는 Model 클래스인 LogInProcess.java클래스는 기존의 것을 그대로 사용하면 된다.

이것이 Action과 비지니스 로직 클래스를 분리했을 때의 힘이다. 컨트롤러 부분이 아무리 바뀌어도 비지니스 작동이 바뀌지 않는다면, 모델의 비지니스 로직은 전혀 바꿀 필요가 없다. 만약 이 부분을 Action 클래스에서 작성했다면, 새로운 login2 를 작성하면서 이 부분도 완전히 새로 작성해야 했을 것이다.

- 사용자 정보를 저장하는 모델 자바 빈인 UserInfoBean.java도 기존 것을 그대로 사용한다.
- 새로운 로그인 Action인 Login2Action.java를 작성한다.

```
package strutsguide.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;

import strutsguide.beans.LogInProcess;
import strutsguide.beans.UserInfoBean;
import strutsguide.forms.Login2Form;

/**
 * Login2 에서 로그인을 실행한다.
 */
public class Login2Action extends Action {

   public ActionForward execute(ActionMapping mapping, ActionForm form,
         HttpServletRequest request, HttpServletResponse response)
         throws Exception {

      LogInProcess logInProcess = LogInProcess.getInstance();

      // 이미 기본적인 파라미터 검증이 끝난 ActionForm 객체를 가져온다.
      Login2Form login2Form = (Login2Form) form;

      // 사용자가 전달한 파라미터를 저장한다.
      String userName = login2Form.getUsername();
      String password = login2Form.getPassword();

      UserInfoBean userInfo = logInProcess.logIn(userName, password);

      ActionForward forward = null;
```

```
        if (userInfo == null) {
            // userInfo가 null이면 사용자가 존재하지 않거나 비밀번호가 잘못된 것이다.
            ActionMessages messages = new ActionMessages();

            // 글로벌 메시지를 추가한다.
            // 이것은 <html:messages id="msg" message="true">에 의해 JSP에서 출력된다.
            messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage(
                    "error.invalidLogin"));

            // request 객체에 메시지 추가
            saveMessages(request, messages);

            // 원래 입력화면으로 돌아가도록 설정한다.
            // <action input="" /> 에서 input에 설정된 페이지를 의미한다.
            return mapping.getInputForward();

        }

        // null이 아니면 성공 페이지로 이동한다. 이동하기 전에 사용자 정보를 session에 저장한다.
        HttpSession session = request.getSession();
        session.setAttribute("userInfo", userInfo);

        // 이동할 페이지를 찾는다.
        // <forward name="login2Form" />을 찾아서 포워딩한다.
        forward = mapping.findForward("login2Form");

        return forward;
    }
}
```

- loginForm.jsp가 전달해주는 파라미터를 Login2Action에 가기전에 데이터를 저장하고 유효성을 검증하는 Login2Form.java ActionForm 클래스를 작성한다.

```
package strutsguide.forms;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

/**
 * Login2Action을 위한 Action Form
 *
 */
public class Login2Form extends ActionForm {

    /** 사용자명 */
    private String username = null;

    /** 비밀번호 */
    private String password = null;

    /**
     * 사용자명 설정하기
     * @param username 사용자명
     */
    public void setUsername(String username) {
        this.username = username;
    }

    /**
```

```
    * 사용자명 리턴하기
    * return 사용자명
    */
   public String getUsername() {
      return username;
   }

   /**
    * 비밀번호 설정하기
    * @param password 비밀번호
    */
   public void setPassword(String password) {
      this.password = password;
   }

   /**
    * 비밀번호 리턴하기
    */
   public String getPassword() {
      return password;
   }

   /**
    * 사용자명과 비밀번호를 올바르게 입력했는지 여부를 검사한다.
    *
    * validate()가 호출되기 전에 이미 setUsername()과 setPassword()가 호출되어
    * username과 password 멤버 변수에 값을 설정한 상태이다.
    *
    * @param mapping 액션 매핑 객체
    * @param request HTTP Request 객체
    * @return 에러 여부
    */
   public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {
      ActionErrors errors = new ActionErrors();

      // 사용자명을 입력하지 않았거나 공백을 포함하고 있을 경우
      if (username == null || username.length() == 0) {
         errors.add("invalidUsernameError",
               new ActionMessage("error.invalidUsername", "사용자명을 입력해주세요."));
      } else if (username.indexOf(" ") >= 0 || username.indexOf("\t") >= 0 ||
            username.indexOf("\n") >= 0) {
         errors.add("invalidUsernameError",
               new ActionMessage("error.invalidUsername", "사용자명은 공백을 포함할 수 없습니다."));
      }

      // 비밀번호를 입력하지 않았을 경우
      if (password == null || password.length() == 0) {
         errors.add("invalidPasswordError",
               new ActionMessage("error.invalidPassword"));
      }

      /*
       * ActionErrors errors 객체가 null이거나 아무런 ActionMessage 객체도 포함하고 있지 않으면
       * 오류가 발생하지 않았다고 가정한다.
       */
      return errors;
   }
}
```

- 로그 아웃 역할을 하는 LogoutAction.java를 작성한다.

```
package strutsguide.actions;

import org.apache.struts.action.*;
```

```
import javax.servlet.http.*;

/**
 * 로그아웃 한다.
 */
public class LogoutAction extends Action {

    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        HttpSession session = request.getSession();

        // 세션 정보를 삭제한다.
        session.invalidate();

            // <forward name="login2Form" path="/login2/loginForm.do" /> 이 정보에 따라
        // 포워딩을 한다.
        return (mapping.findForward("login2Form"));
    }
}
```

- 오류 메시지들을 저장하고 있는 프라퍼티 파일을 생성하자. 프라퍼티 파일은 /WEB-INF/classes/strutsguide/resources/application.properties 로 생성한다.

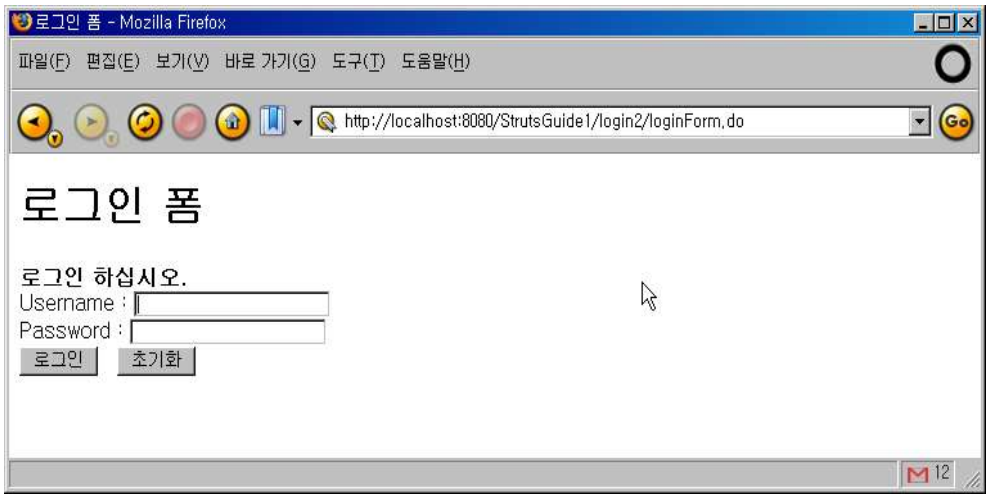
```
error.invalidUsername=잘못된 사용자명입니다. {0}
error.invalidPassword=비밀번호를 입력하지 않았습니다.
error.invalidLogin=로그인 사용자명이 존재하지 않거나 비밀번호가 일치하지 않습니다.
```

헌데, *.properties 파일은 원칙적으로 한글을 포함할 수 없다. 저것을 그대로 복사해서 넣으면 한글이 다 깨져 버린다. 일단 한글로 작성하고 Java에 함께 포함된 "native2ascii" 유틸리티를 이용해 다음과 같이 변환해서 넣어야만 한다. 줄이 나뉘었을 경우 합쳐서 한 줄에 두어야만 한다.

```
error.invalidUsername=\uc798\ubabb\ub41c \uc0ac\uc6a9\uc790\uba85\uc785\ub2c8\ub2e4. {0}
error.invalidPassword=\ube44\ubc00\ubc88\ud638\ub97c \uc785\ub825\ud558\uc9c0
\uc54a\uc558\uc2b5\ub2c8\ub2e4.
error.invalidLogin=\ub85c\uadf8\uc778 \uc0ac\uc6a9\uc790\uba85\uc774 \uc874\uc7ac\ud558\uc9c0
\uc54a\uac70\ub098 \ube44\ubc00\ubc88\ud638\uac00 \uc77c\uce58\ud558\uc9c0 \uc54a\uc2b5\ub2c8\ub2e4.
```

자, 이제 한 번 실행하면서 이 어플리케이션의 흐름을 따라가보자.

/login2/loginForm.do 를 호출하면 된다.

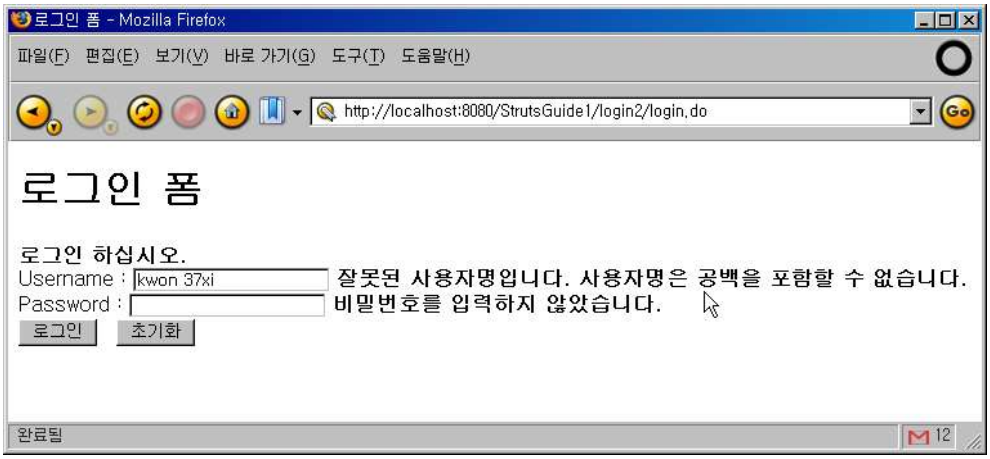


삽화 7 로그인 정보 입력화면

로그인 정보를 입력하는 화면으로, /login2/loginForm.do 를 호출하면 struts-config.xml의 <action path="/login2/loginForm" forward="/login2/loginForm.jsp" /> 에 의해 /login2/loginForm.jsp를 호출하게

된다.

여기서 올바르지 않은 정보를 입력하고 "로그인" 버튼을 누르게 되면 다음과 같은 오류가 발생하게 된다.



삽화 8 username과 password를 형식에 맞지 않게 입력

사용자명과 비밀번호를 입력하고서 "로그인" 버튼을 누르면 제일 먼저 <form-bean name="login2Form" type="strutsguide.forms.Login2Form"/> 과 아래 액션 설정에 의해 ActionForm 인 Login2Form에 전달된다. 아래에서 굵은 글씨로 된 부분 모두가 Action이 사용할 ActionForm에 대한 설정이다.

```
<action path="/login2/login"
        type="strutsguide.actions.Login2Action" name="login2Form"
        validate="true" scope="request" input="/login2/loginForm.jsp">
</action>
```

/login2/login.do Action이 호출되면 액션을 수행하기 전에 먼저 login2Form 이라는 <form-bean> ActionForm 설정을 찾아서 파라미터 값을 설정(name="login2Form")하고 유효성 검증을 하며 (validate="true") ActionForm 객체를 request 스코프에 저장하고(scope="request") ActionForm의 유효성 검증에서 에러가 발생하면 "/login2/loginForm.jsp"로 포워딩해서 다시 입력을 받으라 (input="/login2/loginForm.jsp")는 의미이다.

그 상태에서 Login2Form.setUsername()과 Login2Form.setPassword() 메소드가 자동으로 호출되어 파라미터 username과 파라미터 password의 값이 설정되게 된다.

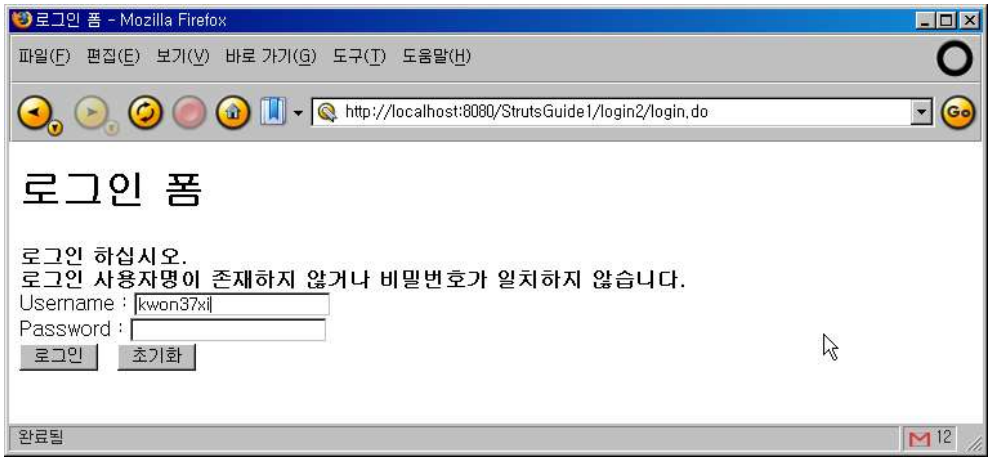
그 직후 Login2Form.validate() 메소드가 호출되어 username과 password 값의 유효성을 검사한다. 여기서는 username이 빈값이거나 공백을 포함할 수 없고 패스워드가 빈 값일수 없다는 규칙이 있으며, 저 그림에 보이는 입력은 그 규칙을 모두 위반한다. **validate() 메소드는 위반한 규칙에 대한 안내 메시지를 ActionErrors 객체에 ActionMessage 객체를 만들어 저장하고, 그것을 리턴**한다.

이 상태에서 RequestProcessor는 Login2Form.validate()의 리턴값인 ActionErrors 객체를 검사한다. ActionErrors 객체가 null 이거나 아무런 ActionMessage 객체도 포함하고 있지 않으면 그대로 Action 을 호출하지만 이번 경우에는 **에러 메시지가 존재하기 때문에 Action을 전혀 호출하지 않고 바로 input="/login2/loginForm.jsp"에 따라 "/login2/loginForm.jsp"로 포워딩**을 하게 된다.

저 위 그림에서 주소창에 나온 최종 URL은 "/login2/loginForm.jsp"가 아니라 "/login2/login.do"라는 것을 볼 수 있다. 포워딩한 것이기 때문이다.

이와같이 오류가 발생해서 다시 입력 폼으로 돌아 올 때 스트럿츠는 ActionForm 객체를 scope="request"에 따라 request 스코프에 저장한다. 그리고 입력폼에 돌아와서는 잘못 입력했던 값들을 화면에 다시 보여주게 된다. 그리고 그 옆에 개발자가 ActionErrors 객체에 저장한 오류 메시지들도 함께 출력해준다.

다음 화면은 형식에 맞는 사용자명과 비밀번호를 입력했지만 사용자명이 존재하지 않거나 비밀번호를 틀리게 입력하고 "로그인"을 하고 난 뒤의 상황이다.



삽화 9 비밀번호가 틀렸거나 사용자명이 존재하지 않음

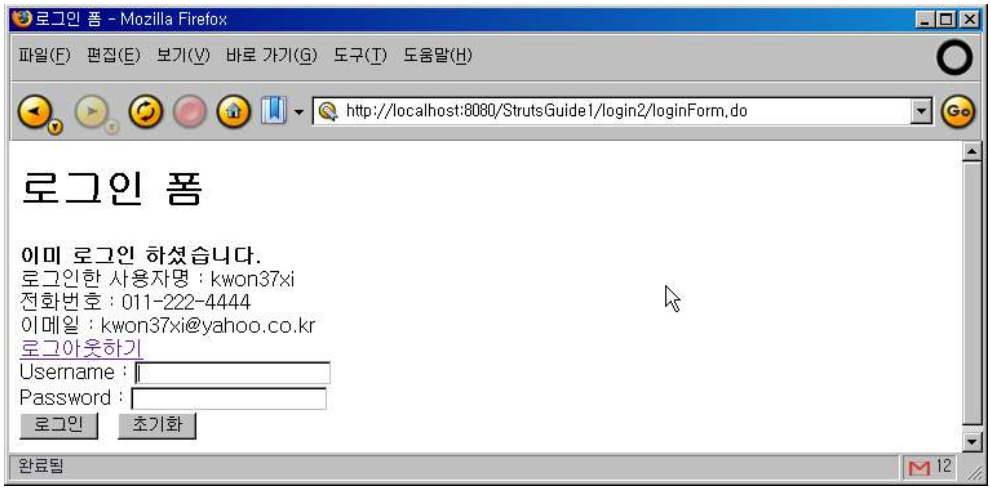
이번 경우에는 ActionForm에서 유효성 검증을 통과 했다. 그리고서 Action을 실행했다.

Action을 실행하면서 비밀번호가 잘못됐기 때문에 ActionMessages 객체에 사용자가 존재하지 않거나 비밀번호가 틀렸다는 메시지를 저장하고는 **return mapping.getInputForward();**과 input="/login2/loginForm.jsp" 에 의해서 다시 입력화면으로 포워딩을 하고 에러 메시지와 기존에 입력했던 사용자명을 출력하게 된다.

아니 뭔가 이상하다?

Action에서는 에러 메시지를 ActionMessages 객체에 저장하고, ActionForm.validate()에서는 ActionErrors 객체에 저장해서 리턴한다. 왜 다르지? 조금만 뒤에서 알아보자.

자, 이제 제대로된 사용자명과 비밀번호를 입력해보자. 그러면 다음과 같이 사용자 정보를 출력하고 로그 아웃 링크와 다시 로그인 할 수 있는 폼을 볼 수 있게 된다.



삽화 10 정상적으로 로그인 됨

위 화면을 보면 최종 URL이 /login2/loginForm.do 인 것을 볼 수 있다. 그것은 login.do에서 정상적으로 로그인을 마치면 **forward = mapping.findForward("login2Form");** 에 의해 login2Form이라는 <forward>를 찾아서 포워딩을 하게 되는데, login2Form은 다음과 같이 지정되어 있다.

```
<global-forwards>
        <!-  Login2 의 로그인 폼 화면으로 이동하기 위한 global forward ->
        <forward name="login2Form" path="/login2/loginForm.do"
                redirect="true" />
</global-forwards>
```

<global-forwards>는 특정 액션에 종속적이지 않은 포워드 정보들을 모아 놓는 곳이다. 여기서 지정된 <forward>들은 모든 Action에서 사용할 수 있다. 단, 각 개별 액션 매핑에 이미 <global-forwards>에 지정된 <forward>와 동일한 이름(name 속성)의 <forward>가 존재할 경우 항상 액션 매핑에 정의된 지역 포워드 정보가 우선적으로 선택된다.

여기서 login2Form 포워드를 지정했고, **redirect="true"** 설정에 의해 포워딩이 아닌 리다이렉트로 작동하게 되는 것이다.

최종적으로 "로그아웃하기" 링크(/login2/logout.do)를 누르면 **<action path="/login2/logout" type="strutsguide.actions.LogoutAction" />** 에 의해서 로그아웃 액션인 LogoutActon을 호출하게 되고 세션에 있는 사용자 정보를 삭제하여 로그아웃 한 다음에 바로 위에서 처럼 login2Form 포워드를 타고 로그인 하기 전의 첫 화면으로 돌아가게 된다.

그런데, 프로그램의 흐름은 다 설명 했지만, 어떻게 오류 메시지들이 출력되고, 잘못 입력했던 사용자명이 입력화면에 다시 나타나는지 등과 /login2/loginForm.jsp에 나오는 <html:form> 등의 이상한 태그에 대해서는 설명하지 않았다.

이제 그러한 스트럿츠의 HTML 폼과 오류 메시지 출력 등에 관한 라이브러리와 커스텀 태그 라이브러리에 대해서 간단하게 알아보자.

- **커스텀 태그**

JSP에서는 커스텀 태그가 나오기 전 까지 스크립틀릿(Scriptlet)이라고 하여 자바 코드를 직접 JSP 페이지에 코딩하는 방법으로 출력할 내용들을 제어했다. 이제는 커스텀 태그 덕분에 JSP에서 자바 코드를 완전히 몰아 내는게 가능하고 그렇게 할 것을 널리 권장하고 있다.

커스텀 태그는 자바 코드를 미리 클래스로 만들고 ***.tld** 파일에 태그 이름과 그 태그를 만날 때 실행할 자바 클래스를 지정해 놓고서 그 태그를 만나면 실제로는 지정된 자바 클래스가 실행되는 것이다.

헌데 실질적으로 커스텀 태그가 있다고 해서 자바 코드를 없앨 수는 없다. Model 1 방식으로 설계를 했을 경우에는 자바 코드(스크립틀릿)를 완전히 없앨 수 없다. 복잡한 비지니스 로직을 JSP 내에서 커스텀 태그로 모두 대체한다는 것은 거의 불가능하고 가능하다 해도 그렇게 하려고 하면 오히려 코드의 복잡도를 증가시키기 때문에 어쩔 수 없이 자바 코드를 쓰게 된다.

하지만 Model 2로 설계하게 되면 문제가 달라진다. 복잡한 비지니스 로직이 모두 Action 같은 자바 클래스로 들어가 버리고 JSP에서는 순수하게 출력만 하게 된다. 순수하게 출력만 할 경우의 로직은 그다지 복잡하지 않으며 스트럿츠의 커스텀 태그와 JSTL 만으로도 그 모든 로직을 커버할 수 있다.

커스텀 태그를 사용해서 얻어지는 이점은 굉장히 많다. 일일이 설명하지 않겠다. 그냥 가능하다면 무조건 커스텀 태그를 사용하고 자신의 JSP 파일에서 자바 코드를 몽땅 몰아낼 수록 유지 보수성이 높아 진다는 것만 알아두고, Model 2 에서는 이게 충분히 가능하므로 적극적으로 활용하자.

스트럿츠는 기본적으로 Html, Logic, Bean, Nested, Tiles 커스텀 태그를 제공하며 각기 하는 역할이 다 다르다. 여기서는 이 모든 것을 설명하지 않는다. 다른 스트럿츠 전문 서적을 참조할 것.

스트럿츠에서는 /WEB-INF/tlds/*.tld 파일의 경로를 uri로 지정하여 커스텀 태그를 사용한다고 JSP에 설정할 수 있다. 다음과 같다.

```
<%@taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>
<%@taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean"%>
<%@taglib uri="/WEB-INF/tlds/struts-logic.tld" prefix="logic"%>
<%@taglib uri="/WEB-INF/tlds/struts-tiles.tld" prefix="tiles"%>
```

prefix는 사실 아무렇게나 줘도 상관없지만 위와 같이 주는 것이 보편적이다. 그냥 지키는게 낫다.

uri의 경우 *.tld 파일에서 <uri>...</uri> 지정된 "http://struts.apache.org/..." 로 시작되는 값을 줘도 상관 없다. 이 방식을 사용할 경우 서블릿 스펙 2.3 이전(톰캣 3.x)에서는 web.xml에 *.tld 파일과 그에 매칭될 uri 정보를 작성해줘야 했으나 서블릿 스펙 2.3 이상(톰캣 4.x 이상)에서는 그럴 필요가 없다. 그리고 이 경우에 tld 파일을 따로 WEB-INF/tlds 디렉토리에 복사할 필요도 없다. 웹 컨테이너가 struts.jar 파일에 있는 *.tld 파일 정보를 자동으로 인식한다.

이 문서에서는 *.tld 파일의 경로를 기준으로 삼겠다. *.tld 파일의 경로를 기준으로 삼을 경우에는 web.xml에서 특별히 뭘 해줄 필요가 없다. 단지 WEB-INF/tlds 디렉토리에 *.tld 파일을 복사해 두기만 하면 된다.

http://struts.apache.org/userGuide/index.html 이 페이지에서 각 태그의 용법을 익힐 수 있다.

- **HTML 폼을 스트럿츠 커스텀 태그로 만들기 - 폼의 값이 ActionForm에 의해 자동으로 채워짐**

/login2/loginForm.jsp를 보면 일반 HTML 태그가 아니라 **<html:어쩌구>**하는 태그들을 이용해서 이뤄짐을

볼 수 있다.

실제 /login2/loginForm.do 를 실행하고서 소스보기를 해 보면 다음과 같은 HTML이 자동으로 생성되어 있음을 볼 수 있다.

```
<!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=euc-kr">
      <base href="http://localhost:8080/StrutsGuide1/login2/loginForm.jsp">
      <title>로그인 폼</title>
   </head>
   <body>

   <h1>로그인 폼</h1>
                  <b>로그인 하십시오.</b><br/>
   <!– 폼의 데이터를 "login.do"로 전송한다. –>
   <form name="login2Form" method="POST"
action="/StrutsGuide1/login2/login.do;jsessionid=56BA04512188FA92077A907127CF1D34">
      Username : <input type="text" name="username" value="">
          <br />
      Password : <input type="password" name="password" value="">
                  <br />
      <input type="submit" value="로그인">   <input type="reset" value="초기화">
   </form>
<script type="text/javascript" language="JavaScript">
  <!–
  var focusControl = document.forms["login2Form"].elements["username"];

  if (focusControl.type != "hidden" && !focusControl.disabled) {
     focusControl.focus();
  }
  // –>
</script>
   </body>
</html>
```

위와 같이 JSP에 입력한 적이 없는 정보들이 커스텀 태그의 작동에 의해 자동으로 입력되었음을 볼 수 있다.

- **<html:html>**

  <html> 태그를 의미한다.

- **<html:base/>**

<base href=””/> 태그를 자동 생성해준다. 스트럿츠 처럼 가상의 URL을 JSP에 매핑 시켜 사용할 경우, 이미지 파일 등의 상대경로가 JSP 기준이 아닌 가상 URL 경로에 의해 결정되게 된다. 그것을 막고 JSP의 경로에 의해 다른 리소스의 상대경로를 지정할 수 있게 해주는 것이다 <html:base/>의 역할이다.

- **<html:form action="/login2/login" method="POST" focus="username">**

<form> 태그를 생성한다. action 은 <form>태그에서와는 달리 struts-config.xml 의 **<action path=””/>**의 path에 해당하는 값을 의미한다. method 는 GET과 POST 두가지 방식이다. focus는 폼이 출력된뒤에 포커스를 둘 입력 박스를 의미한다.

action=”/login2/login” 은 자동으로 struts-config.xml의 <action path=”/login2/login” name=”login2Form”/>의 name 속성을 통해서 <form-bean name=”login2Form” ... /> 의 정보를 읽어들인다.

그리고 ActionForm의 validate()에서 에러가 발생하면, ActionForm 객체가 request 혹은 session 등 <action scope=””/>에 설정한 스코프에 저장되고, <html:어쩌구> 태그들은 해당 액션 폼 빈의 값을 각 <input> 요소들에 자동으로 설정해 주게 된다.

- **<html:text property="username"/>**

<input type="text">를 생성해준다. property="username"은 <input>에 name="username"을 추가하고 ActionForm 객체가 session혹은 request에 존재할 경우 ActionForm에서 getUsername() 해서 가져온 값을 이 input 박스의 value로 설정하겠다는 의미가 된다.

- **<html:password property="password" redisplay="false"/>**

<input type="password"/>를 생성해준다. property는 <html:text>와 마찬가지이다. 단, redisplay="false"는 ActionForm이 스코프에 존재하더라도 화면상에 value="" 설정을 하지 않도록 해준다. 비밀번호니까 당연히 하면 안된다.

- **<html:submit value="로그인"/>**

<input type="submit"> 버튼을 생성한다.

- **<html:reset value="초기화" />**

<input type="reset"> 버튼을 생성한다.

이 이외에도 HTML의 FORM 과 관련된 거의 모든 태그가 <HTML:어쩌구> 커스텀 태그로 존재하며 스트럿츠 프레임워크와 유기적으로 작동한다.

• 에러 혹은 일반 메시지 출력하기

스트럿츠는 ActionForm의 validate() 메소드에서 ActionErrors를 리턴하거나 Action 에서 **saveMessages()** 메소드를 이용해 JSP에서 출력해줄 에러 혹은 일반 메시지를 설정할 수 있다.

메시지를 저장하는 클래스는 두가지 종류로 나뉘었다. ActionErrors와 ActionError 그리고 ActionMessages와 ActionMessage이다. 이 두 종류의 메시지 저장 방식은 사실상 동일한 역할을 하고 동일한 방식으로 작동하는데 단지 에러 메시지냐 일반 메시지냐에 따라 구분한 것이다.

이제부터는 이 중에서 ActionErrors와 ActionError(deprecated 됨)는 사용을 권장하지 않는다. 앞으로는 **ActionMessages와 ActionMessage만 사용하도록 한다.**

- **ActionMessages** 는 메시지들을 키 값을 할당하여 저장하는 역할을 한다.
- **ActionMessage** 는 메시지 그 자체이며 *.properties로 저장된 리소스 번들로 부터 메시지를 가져다 필요한 값을 더 채워넣은 것이다.

바로 위에서 ActionErrors와 ActionError를 사용하지 말라고 했는데 한가지 문제가 있다.

ActionForm의 validate() 메소드의 리턴형이 ActionErrors라는 것이다. 이미 이 메소드는 몇년 전부터 사용되었기 때문에 함부로 바꾸면 스트럿츠 버전업에 문제가 생긴다.

이 때문에 ActionError는 deprecated 되었지만 ActionErrors는 deprecated 될 수 없었다(하지만 앞으로 나올 버전에서는 deprecated 될 수도 있다고 한다). **ActionForm.validate()에서는 ActionErrors를 메시지 저장소로 사용하고 실제 메시지 객체는 ActionMessage로 생성**하며, **일반 Action 에서는 오류 메시지와 일반 메시지 모두 ActionMessages와 ActionMessage를 이용해서 생성**하도록 한다.

ActionForm.validate()와 Action 에서 사용하는 예는 이미 보았지만 하나하나 따라가며 설명해보자.

먼저 에러 메시지의 내용을 저장하고 있을 *.properties 를 생성해야 한다. 그리고 그 properties 파일을 struts-config.xml에 설정해야 한다.

```
<message-resources parameter="strutsguide.resources.application" />
```

보다싶이 자바의 패키지 구조와 같게 설정하고 맨 마지막에 .properties는 적지 않는다. 위와 같이 설정하면 스트럿츠는 /WEB-INF/classes/**strutsguide/resources/application.properties** 를 찾아서 메모리에 올려두게 된다.

그리고는 ActionErrors 혹은 ActionMessages 객체를 생성한다 (ActionForm.validate() 메소드에서는 물론 ActionErrors 객체로 생성해야 한다).

```
ActionMessages messages = new ActionMessages();
```

이제 오류가 발견되면 ActionMessage 객체를 하나하나 생성해서 messages 객체에 저장하면 된다.

```
messages.add("invalidUsernameError",
             new ActionMessage("error.invalidUsername", "사용자명을 입력해주세요."));
```

첫번째 인자 "invalidUsernameError"는 메시지의 키값이다. JSP 파일에서 이 키를 이용해서 지정된 에러 메

시지를 원하는 위치에 출력할 수 있다.

두번째 인자는 ActionMessage의 객체이다.

ActionMessage 객체의 첫번째 인자는 application.properties 에 있는 프라퍼티 키값이다. ActionMessage는 프라퍼티 파일에서 해당 키의 실제 문자열(여기서는 "잘못된 사용자명입니다. {0}")을 가져온다. 그리고는 {0} 을 그 다음 인자로 대체한다. 즉, 최종 문자열은 "잘못된 사용자명입니다. 사용자명을 입력해주세요." 가 되는 것이다.

{0} 처럼 대체 문자열을 지정하는 것은 최대 4개까지 가능하다. 즉 {0}, {1}, {2}, {3} 을 프라퍼티 파일의 값으로 지정할 수 있다.

또한 ActionMessage의 객체 생성시 그에 따라 최대 4개까지의 대체 문자열을 생성자에서 제공해 줄 수 있다.

이렇게 메시지를 저장하고서 ActionForm.validate() 는 "return messages"를 함으로써, Action 의 경우에는 "**saveMessages(messages)**"를 함으로써 그 다음에 오는 JSP 페이지에 이 메시지들을 제공할 수 있게 되는 것이다.

메시지를 제공 받은 JSP 페이지에서는 다음 처럼 출력할 수 있다.

```
<html:messages id="msg" property="invalidUsernameError">
        <%–
                invalidUsernameError 라는 키로 저장된 ActionMessage 객체가 없다면
                이 부분은 실행되지 않는다.
        –%>
        <b><bean:write  name="msg"/></b>
</html:messages>
```

<html:messages>에서 property="" 는 메시지를 add 할때 지정한 키를 의미한다. 그 키에 해당하는 메시지가 있다면 그 메시지를 "msg" 라는 변수에 저장하라는 것이 id="msg" 의 의미이다.

이렇게 저장된 msg 변수를 <bean:write name="msg"/> 가 출력을 시켜준다.

위와 같이 메시지에 키를 지정하여 저장할 수도 있지만, 그렇지 않고 특별한 키값 없이 메시지를 저장 할 수도 있다.

```
messages.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage("error.invalidLogin"));
```

위와 같이 "ActionMessages.GLOBAL_MESSAGE"를 키로주면 된다.. ActionMessages 가 아닌 ActionErrors 의 경우에는 "ActionErrors.GLOBAL_ERROR"를 키로 줘야한다.

그리고 출력은

```
<html:messages id="msg" message="true">
        <%–
                ActionMessages.GLOBAL_MESSAGE 키로 저장된 ActionMessage 객체가 없다면
                이 부분은 실행되지 않는다.
        –%>
        <b><bean:write name="msg"/></b> <br />
</html:messages>
```

위와 같다. message="true"는 이것이 ActionErrors가 아니라 ActionMessages의 메시지임을 의미한다. ActionErrors의 경우에는 message="false"로 지정한다.

동일한 키로 여러개의 메시지를 저장할 수 있는데, <html:messages>는 id="msg"에 지정된 msg 변수에 각 메시지를 저장해서 내부의 <bean:write name="msg"/>
 를 반복호출한다.

여기서는 간단히만 언급하자면 *.properties를 통해 여러 나라말로 메시지를 작성할 수 있고, 웹 브라우저에 설정된 언어에 따라 서로 다른 언어의 메시지를 보여주는 것이 가능하다(i18n; 국제화 기능).

5. 스트럿츠 커스텀 태그들

- **<bean:write>**

Action은 Action의 실행 결과를 request[session,application].setAttribute() 등으로 스코프에 저장해서 뷰인 JSP로 전달한다고 했다.

- 이것은 대부분 <bean:write name="attribute이름" property="프라퍼티이름"/> 으로 출력가능하다.
- <bean:write>에서 property를 생략하면 name 에 지정된 객체를 그냥 출력한다.
- 하지만 property="email" 처럼 지정하면 name에 지정된 객체에서 .getEmail()을 실행하여 그 결과 값을 출력한다.
- <bean:write name="userInfo" property="address.postcode"/> 처럼 지정했다면, userInfo.getAddress().getPostcode() 와 같은 코드를 실행해서 그 결과를 출력한다.
- <bean:write>는 기본적으로 HTML의 특수 문자들(&, <, > 등)을 &, < 등 처럼 바꿔서 출력해준다. 그렇게 하지 않고 있는 그대로 출력하고자 할 때는 <bean:write filter="false"/> 에서 처럼 filter 속성을 "false"로 지정하면 된다.

- **<html:rewrite>**

<html:rewrite>는 HTML 작성에서 매우 유용하게 사용될 수 있다.

일반적으로 HTML 페이지에 <img src="/image/button.gif"> 와 같이 지정하면 **절대로 안된다**.

웹 컨텍스트의 경로가 "/" 일 경우에는 상관없지만 만약에라도 웹 컨텍스트 경로가 "/strutsexample" 과 같은 식으로 바뀐다면 저기서 src에 지정한 URL은 아무 의미가 없어진다. 실제로 요청해야할 URL이 "/strutsexample/image/button.gif"로 바뀌어 버리기 때문이다.

그렇다고 상대 경로로 지정하면 매우 일이 귀찮아지고 잘못 입력할 확률도 높아진다.

이럴 때 <html:rewrite page=""/>를 사용하면 웹 컨텍스트 경로의 변경을 자동으로 반영해준다. 즉, 전자의 경우에는 "/image/button.gif"를 그대로 출력했다가 웹 컨텍스트가 "strutsexample"로 바뀔 경우에는 "/strutsexample/image/button.gif"로 출력했다가 하는 것을 알아서 해주는 것이다.

위의 경우에는 <img src='**<html:rewrite page="/image/button.gif"/>**'> 처럼 사용하면 된다. src속성을 잘 보면 값을 작은따옴표로 감쌌음을 볼 수 있다. 이유는 <html:rewrite page="">에서 큰 따옴표가 사용되었기 때문이다. 서로 겹치지 않게 해줘야 한다.

웹 컨텍스트의 이름이 바뀌는 문제와 form의 action 속성 혹은 img 의 src 속성등에 "/" 로 시작하는 절대 경로 지정문제는 많은 초보 웹 개발자들이 실수하는 부분이다. 내 글이 시원찮아 이해가 잘 안된다면 꼭 이에관한 사항을 다른 이들에게 물어서라도 확인하고 이해하고 넘어가야 한다.

6. RequestProcessor 상속하기

MVC 패턴에서 Controller가 있기 때문에 좋은점이 많다.

그 중의 한가지는 모든 액션을 수행하기 전에 실행해야할 어떤 작업이 있을 때 그 작업을 Controller 한 곳에 시켜두면 다른 모든 액션에 일괄적으로 적용된다는 것이다.

이와 같이 Controller에서 사용자가 원하는 코드를 집어넣을 수 있는 방법이 RequestProcessor를 상속받아서 원하는 기능을 구현하는 것이다.

RequestProcessor는 스트럿츠의 기본 컨트롤러이다.

RequestProcessor는 request와 response 등의 객체를 차례대로 여러 메소드에 통과 시켜 전처리 작업을 수행하고, Action을 호출하는 과정을 수행한다.

RequestProcessor의 그러한 과정의 메소드들 중에서 사용자가 주로 오버라이드해서 사용하는 것들이 있는데,

- **processPreprocess** : 범용 전처리 훅. Action을 호출하기 전에 request와 response 객체에 이거 저거 필요한 일들을 여기서 해준다.
- **processRoles** : Action을 호출하기 전에, 이 Action을 호출하려면 특정한 롤에 속해야 하는지를 검사한다. 이것을 통해 단순히 스트럿츠 설정 파일만을 이용해서 사용자가 특정 페이지에 접속할 수 있는 권한이 있는지 검사 할 수 있다.

• 사용자가 상속받아 만든 RequestProcessor 지정하기

struts-config.xml에 다음 설정을 해주면 된다.

```
<controller processorClass="strutsguide.controller.NewRequestProcessor"/>
```

processorClass 속성이 자신이 작성한 새로운 RequestProcessor 클래스를 등록하면 된다.

<controller>는 <action-mapping>아래에 위치한다. 위치가 틀리면 작동하지 않는다. 각 요소별 자세한 위치는 struts-blank.war 에 있는 struts-config.xml을 참조하면 알 수 있다.

• processRoles() 메소드 오버라이드

- 기본적인 오버라이드 형태

```
protected boolean processRoles(
            HttpServletRequest request,
            HttpServletResponse response,
            ActionMapping mapping) throws IOException, ServletException {

    // 현재 수행할 Action이 요구하는 Role의 목록을 얻는다.
    String roles[] = mapping.getRoleNames();

    // request나 session 객체의 정보를 통해서
    // 현재 사용자가 roles[] 에 해당하는지 검사한다.

    // 해당 Action을 수행할 수 있는 Role에 속하면
    return true;

    // 아니면
    return false;
}
```

- 현재 Action의 수행에 필요한 Role 목록은 struts-config의 action에서 설정한다.

```
<action-mappings>
  <action
    roles="admin,manager"
```

```
        ....
    />
</action-mappings>
```

따로 예제를 살펴보지는 않겠다.

- **processPreprocess 오버라이드**

이 메소드는 사실상 잡다한 일들을 다 해주면 된다고 봐도 된다.

여기서는 우리나라 개발자들이 폼에서 넘어온 한글 파라미터 값을 제대로 인식시키기 위해 JSP에 필수적으로 사용하는 request.setCharacterEncoding(); 작업을 수행해 보자.

- NewRequestProcessor 컨트롤러 작성

```
package strutsguide.controller;

import java.io.UnsupportedEncodingException;

import org.apache.struts.action.RequestProcessor;
import javax.servlet.http.*;

/**
 * RequestProcessor를 상속받아 한글 인코딩 부분을 설정한다.
 */
public class NewRequestProcessor extends RequestProcessor {

    /**
     * 모든 Action이 실행되기 전에 request와 response에 실행되어야 할 사항들을 실행한다.
     */
    protected boolean processPreprocess(HttpServletRequest request,
            HttpServletResponse response) {

        try {
            // HTTP 파라미터의 인코딩을 설정한다.
            request.setCharacterEncoding("euc-kr");
        } catch (UnsupportedEncodingException e) {
            // do nothing;
        }

        return true;
    }
}
```

이제부터는 이 컨트롤러를 사용하는 모든 액션과 JSP에 **request.setCharacterEncoding("euc-kr")**을 전혀 해줄 필요가 없다.

모든 사용자의 요청의 request와 response 객체가 이 컨트롤러의 processPreprocess()를 거쳐가기 때문에 자동으로 된다.

이것을 실제 사용한 예는 Validator 부분에서 볼 수 있다.

그리고, 뒤에서 Tiles와 함께 이것을 사용하게 될 텐데, **Tiles를 사용할 경우에는 RequestProcessor가 아니라 org.apache.struts.tiles.TilesRequestProcessor를 상속받아야만 한다.**

7. Tiles 이해하기

- **Tiles를 왜써?**
  - Tiles는 웹 페이지의 상단이나 하단, 메뉴와 같은 반복적으로 사용되는 부분들에 대한 정보를 한 곳에 모아 두도록 해주는 프레임워크이다.
  - <jsp:include>나 <%@include%> 등을 이용해 비슷한 역할을 할 수 있지만 Tiles가 훨씬 뛰어나다.
  - JSP Include 방식의 문제점은 매 JSP 페이지마다 모두 동일한 페이지들을 include 한다고 표시를 해야한다. 만약 include 하는 페이지의 이름이 바뀌기라도 한다면 그에 해당하는 모든 페이지를 다 수정해야만 한다.
  - Tiles는 반복 사용되는 부분을 설정 파일에 한 번만 설정하고, 계속해서 변하는 부분만을 따로 지정할 수 있다. 반복 사용되는 부분에 변화가 있더라도 설정 파일의 한 부분만 바꾸면 된다.

- **Tiles의 기본**
  - Tiles는 기본적으로 두 가지로 이루어 진다. 레이아웃(Layout)과 Tiles Definition.
  - **레이아웃** : 모든 페이지에 기본적으로 표시될 HTML 태그들과 반복 적으로 사용되는 부분들이 어디에 위치해야 할지를 정의한 JSP 파일이다.
  - **Tiles Definition** : 레이아웃에서 반복적으로 사용되는 부분들의 실제 HTML 혹은 JSP 파일을 지정하는 설정이다. XML혹은 JSP로 설정 할 수 있는데, 보통은 XML을 사용한다.

- **Tiles를 사용할꺼야**
  - struts-config.xml에 다음과 같이 Tiles Plugin을 설정한다.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
    <set-property property="definitions-config" value="/WEB-INF/config/tiles-defs.xml"/>
    <set-property property="definitions-debug" value="2"/>
    <set-property property="definitions-parser-details" value="2"/>
    <set-property property="definitions-parser-validate" value="true"/>
</plug-in>
```

<set-property property="definitions-config" value="/WEB-INF/config/tiles-defs.xml"/> 이 부분이 Tiles Definition 설정 파일을 지정하는 부분이다. 이제 타일즈를 사용할 수 있게 되었다.

- **Layout 만들기**
  - Layout은 화면에 출력할 내용의 뼈대를 구성하는 JSP 파일이다.
  - 다음 예제를 "/tiles/classicLayout.jsp"라는 이름으로 작성한다.

```
<%@page contentType="text/html; charset=euc-kr"%>
<%@taglib uri="/WEB-INF/tlds/struts-tiles.tld" prefix="tiles"%>
<!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=euc-kr">
      <title><tiles:getAsString name="title" /></title>
   </head>
   <body>

   <h1><tiles:getAsString name="title"/></h1>
   <table border="1" width="100%">
      <tr>
         <td colspan="2"><tiles:insert attribute="header"/></td>
      </tr>
      <tr>
```

```
        <td width="140"><tiles:insert attribute="menu"/></td>
        <td><tiles:insert attribute="body"/></td>
      </tr>
      <tr>
        <td colspan="2"><tiles:insert attribute="footer"/></td>
      </tr>
    </table>
    </body>
</html>
```

- **<%@taglib uri="/WEB-INF/tlds/struts-tiles.tld" prefix="tiles"%>**를 통해 Tiles 커스텀 태그를 사용하겠다고 설정한다.
- **<tiles:insert attribute="footer"/>** 를 통해서 이 커스텀 태그가 위치한 부분에 Tiles Definition(tiles-defs.xml)에서 "footer"라는 이름으로 지정한 JSP혹은 HTML 등을 출력해 주겠다고 표시한다.
- **<tiles:getAsString name="title"/>** 는 Tiles Definition에서 "title"로 지정한 값을 화면에 그대로 문자열로 출력하라는 뜻이다.
- 이 레이아웃 JSP 파일을 적당한 위치에 저장(여기서는 /tiles/classicLayout.jsp)한다.

• Tiles Definition : 레이아웃에 실제 내용 채워넣기

- 위의 레이아웃은 헤더, 메뉴, 바디, 푸터로 구성되어 있다. 여기서 헤더, 메뉴, 푸터는 항상 동일하고 바디 부분만 계속 바뀐다. 기존 JSP Include 방식에서는 항상 동일한 헤더, 메뉴, 푸터에 대한 Include 를 바디가 바뀌는 부분마다 모두 선언해야 했다. 하지만, Tiles는 그렇지 않다.
- 타일즈의 Definition을 다음과 같이 "/WEB-INF/config/tiles-defs.xml"이라는 이름으로 만든다.

```
<?xml version="1.0" encoding="euc-kr" ?>

 <!DOCTYPE tiles-definitions PUBLIC
      "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
      "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
   <definition name=".layout-main" path="/tiles/classicLayout.jsp">
      <put name="title" value="타일즈 사용하기~"/> <!– 레이아웃에서 <tile:getAsString>으로 그대로 문자열로 출력되는
값 –>
      <put name="header" value="/tiles/header.jsp"/>
      <put name="menu" value="/tiles/menu.jsp"/>
      <put name="body" value="/tiles/body.jsp"/>
      <put name="footer" value="/tiles/footer.jsp"/>
   </definition>

   <definition name=".layout-menu1" extends=".layout-main">
      <put name="body" value="body-menu1.jsp"/>
   </definition>

   <definition name=".layout-menu2" extends=".layout-main">
      <put name="body" value="body-menu2.jsp"/>
   </definition>
</tiles-definitions>
```

- <definition> : 레이아웃 내용 채우기 설정의 한 단위이다. 한 개의 레이아웃에 대해 여러개의 definition 들이 존재 할 수도 있다.
- <definition **name=””**/> : definition 이름. 다른 이름들과 동일하면 안된다. 이름을 통해 각 단계를 나타내기 위해 점(.)을 사용한다.
- <definition **path=””**/> : 이 definition에서 지정한 JSP 혹은 HTML 파일들이 실제로 삽입될 레이아웃 JSP 파일을 지정한다.
- <put> : 레이아웃에 삽입될 JSP 혹은 HTML 파일을 지정한다.
- <put **name=””**/> : 레이아웃에서 사용된 각 부분별 명칭.
- <put **value=””**/> : 실제로 삽입될 내용이 있는 JSP, HTML 등.

- definition 확장하기
  - Tiles의 장점은 바로 다른 definition을 확장하여 바뀌지 않는 부분은 부모 definition에 설정하고 바뀌는 부분(바디)만 자식 definition에 설정하여, 자식 definition을 사용해서 레이아웃에 삽입되어 들어갈 내용을 지정할 수 있다는 것이다.
  - <definition **extends="부모definition의 name"**/>을 이용해서 한다. 여기서는 "body" 로 지정된 부분의 JSP만 계속 바뀌므로 그것만 바꿔 지정한다.
  - 부모 definition과 자식 definition 모두 다 사용될 수 있다.

• Tiles 사용하기

- 이제 설정을 마쳤으니, 실제 출력할 부분을 만들 수 있다.
- 그전에 struts-config.xml 에서 지금 만든 데피니션을 Action의 Forward로 지정해야 한다.

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

<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
      <action
         path="/tiles/main"
         forward=".layout-main"/>
      <action
         path="/tiles/menu1"
         forward=".layout-menu1"/>
      <action
         path="/tiles/menu2"
         forward=".layout-menu2"/>

   </action-mappings>

   <plug-in className="org.apache.struts.tiles.TilesPlugin">
      <set-property property="definitions-config" value="/WEB-INF/config/tiles-defs.xml"/>
      <set-property property="definitions-debug" value="2"/>
      <set-property property="definitions-parser-details" value="2"/>
      <set-property property="definitions-parser-validate" value="true"/>
   </plug-in>
</struts-config>
```

- 각 Tiles Definition을 **<action forward="">**에서 forward의 값으로 지정하거나, **<forward path="">**에서 path의 값으로 지정할 수 있다.
- 여기서는 가장 단순하게 <action forward="">에제만 본다.
- 실질적으로 definition의 개수만큼의 forward가 필요하다.
- 이제는 실제로 화면에 추가될 JSP 파일들을 생성한다. header.jsp, footer.jsp, menu.jsp 는 세 가지 definition이 모두 공통으로 사용하므로 한 개씩만 만들고, body 부분에 대한 JSP는 세 개 모두 만든다.
- header.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<h2 align="center">Struts Tiles 예제입니다.~</h2>
```

- footer.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<div align="center" style="background: yellow;">우리 회사</div>
```

- menu.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>
```

```
<ul>
   <li><html:link action="/tiles/main" >첫화면</html:link>
   <li><html:link action="/tiles/menu1">메뉴 1</html:link>
   <li><html:link action="/tiles/menu2">메뉴 2</html:link>
</ul>
```

- body.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<b>첫 화면 바디~</b>
```

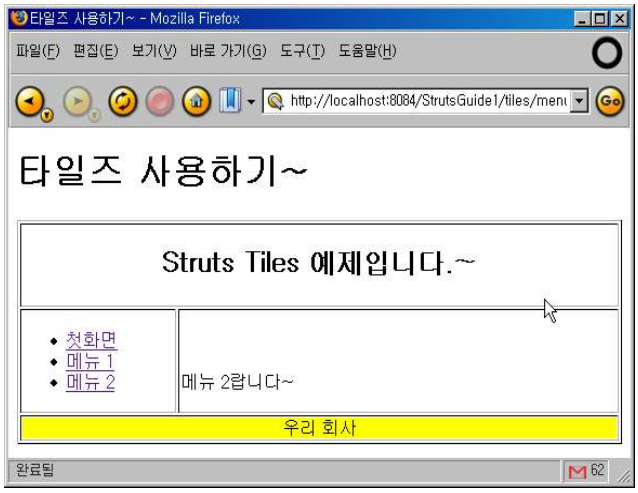
- body-menu1.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<br />
<br />
메뉴 1이랍니다~
```

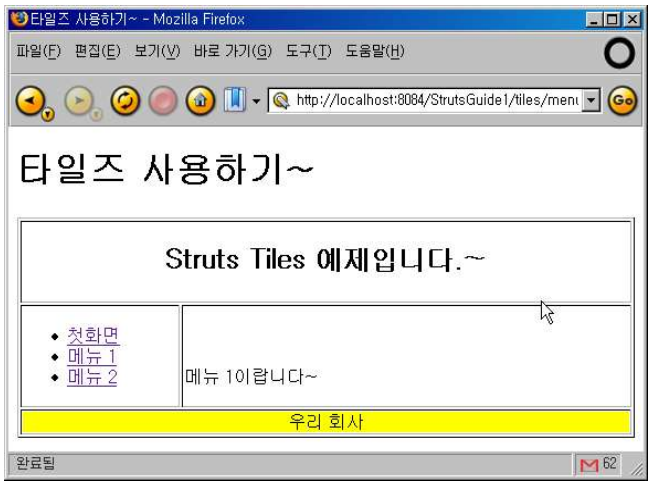
- body-menu2.jsp

```
<%@page contentType="text/html; charset=euc-kr"%>
<br />
<br />
메뉴 2랍니다~
```

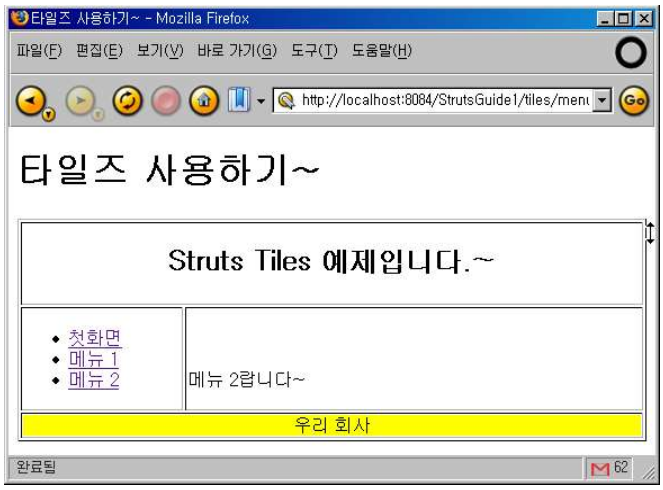
- 실제 화면 출력 모양
  - 주메뉴 (/tiles/main.do)



  - 메뉴 1 (/tiles/menu1.do)



  - 메뉴 2 (/tiles/menu2.do)


타일즈 사용하기~ - Mozilla Firefox
파일(F) 편집(E) 보기(V) 바로 가기(G) 도구(T) 도움말(H)
http://localhost:8084/StrutsGuide1/tiles/menu
타일즈 사용하기~
Struts Tiles 예제입니다.~
첫화면
메뉴 1
메뉴 2
메뉴 2랍니다~
우리 회사
완료됨

8. Validator 사용하기

지금까지 ActionForm에 저장된 값의 유효성을 검사할 때 validate() 메소드를 사용하였는데, 이렇게 할 경우 두 종류 이상의 ActionForm이 동일한 규칙을 가진 프라퍼티를 가지고 있을 때도 각각의 validate()에 동일한 유효성 검증 코드를 넣어야 하는 문제가 있고, 유효성 검증 규칙이 바뀔때마다 validate()의 코드를 수정해서 재컴파일 해야 한다는 문제가 있다.

그래서 스트럿츠 Validator(유효성검사기) Framework이 나왔다. Validator 프레임워크는 설정 XML 파일을 이용해서 각 ActionForm 프라퍼티들의 유효성을 검사할 수 있게 해준다. ActionForm에 validate()를 만들 필요가 없다.

Validator는 두 개의 설정 파일을 가진다.

- **valdator-rules.xml** : 검증 규칙을 선언한다. 예를들면 "주민등록번호는 xxxxxx-1xxxxxx 혹은 xxxxxx-2xxxxxx 처럼 앞에 6자리 뒤에 1혹은 2로 시작하는 7자리 숫자로 구성된다" 라는 식의 규칙만을 저장하고 있는 것이다. 여기서 선언된 규칙을 여러 ActionForm을 검증하는데 사용할 수 있게 된다.
- **validation.xml** : 각 ActionForm의 어떤 프라퍼티가 validator-rules.xml에 설정된 어떤 규칙을 만족해야 하는지를 설정한다. 예를들면 "UserInfoForm의 juminbunho 프라퍼티는 validator-rules.xml의 'juminbunho' 규칙을 지켜야한다."는 식의 설정을 하게 된다.

• validator-rules.xml

<validator> 요소를 통해서 규칙을 선언할 수 있다. 여기서는 새로운 검증 규칙을 어떻게 선언하는지는 설명하지 않는다.

기본적으로 알아야 할 것은 **<validator name="" msg=""/>** 에서 name과 msg 속성이다.

- name 속성 : 검증 규칙의 이름. validation.xml 에서 이 이름을 통해 액션 폼의 프라퍼티에 적용할 규칙을 지정한다.
- msg 속성 : struts-config.xml 에서 지정한 리소스 번들 .properties 파일에서 가져올 오류 메시지의 키를 지정한다. ActionForm.validate()에서 오류 메시지를 ActionErrors에 담아서 리턴하듯이 Validator는 리소스 번들 프라퍼티 파일에 있는 메시지를 ActionMessages에 담아서 리턴한다.

스트럿츠 Validation의 기본 예제 validator-rules.xml 파일을 보면 이미 사용가능한 유용한 검증 규칙들이 선언되어 있다. 그것들만 사용해도 별 문제 없다. (required, minlength, maxlength, ...)

또한 기본 검증 규칙들을 위한 리소스 번들 메시지를 자신의 리소스 번들 프라퍼티 파일에 추가할 필요가 있는데, 기본적으로는 영문 메시지가 validator-rules.xml에 예제로 들어 있다. 그걸 번역한게 아래이다.

```
# Struts Validator Error Messages
errors.required={0}(이)가 필요합니다.
errors.minlength={0}(은)는 최소한  {1} 문자 이상이어야 합니다.
errors.maxlength={0}(은)는 최대  {1} 문자 이하이어야 합니다.
errors.invalid={0}(이)가 유효하지 않습니다.
errors.byte={0}(은)는 byte 형이어야 합니다.
errors.short={0}(은)는 short 형이어야 합니다.
errors.integer={0}(은)는 integer 형이어야 합니다.
errors.long={0}(은)는 long 형이어야 합니다.
errors.float={0}(은)는 float 형이어야 합니다.
errors.double={0}(은)는 double 형이어야 합니다.
errors.date={0}(은)는 날짜형이 아닙니다.
errors.range={0}(은)는 {1}에서 {2} 사이 값이어야 합니다.
errors.creditcard={0}(은)는 유효하지 않은 신용카드 번호입니다.
errors.email={0}(은)는 유효하지 않은 이메일 주소입니다.
```

물론 native2ascii를 이용해서 변환해서 넣어야 한다.

여기 예제에서는 스트럿츠 1.2.7 의 struts-blank.war 파일의 WEB-INF 디렉토리에 포함된 validator-rule.xml을 /WEB-INF/config/에 복사한다. 그 내용을 한 번쯤 들여다 보자. required, minlength, .. 등의 규칙이 선언되어 있음을 볼 수 있다.

- **validation.xml**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE form-validation PUBLIC
        "-//Apache Software Foundation//DTD Commons Validator Rules Configuration 1.1.3//EN"
        "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">

<form-validation>

        <global>
                <!– 우편번호 상수 123-456 형태 –>
                <constant>
                        <constant-name>postalCode</constant-name>
                        <constant-value>^\d{3}-\d{3}$</constant-value>
                </constant>

        </global>

        <formset>
                <!– userInfoForm 폼 빈에 대한 유효성 검증 규칙 –>
                <form name="userInfoForm">
                        <!– userInfoForm의 name(이름) 프라퍼티에 대한 검증 규칙 –>
                        <field property="name" depends="required">
                                <!– arg 는 에러 메시지 출력시 {0}, {1}, .. 등에 채워줄 메시지의 키이다. –>
                                <arg key="userInfoForm.username" />
                        </field>

                        <!– userInfoForm의 address(주소) 프라퍼티에 대한 검증 규칙 –>
                        <field property="address" depends="required">
                                <arg key="userInfoForm.address" />
                        </field>

                        <!– userInfoForm의 postcode(우편번호) 프라퍼티에 대한 검증 규칙 –>
                        <field property="postcode" depends="required,mask">
                                <!– mask 규칙에서 오류가 발생했을 때 출력할 메시지 지정 –>
                                <msg name="mask" key="userInfoForm.postcodeError"/>
                                <arg key="userInfoForm.postcode" />
                                <var>
                                        <var-name>mask</var-name>
                                        <var-value>${postalCode}</var-value> <!– 상수 사용 –>
                                </var>
                        </field>

                        <field property="email" depends="required,email">
                                <!– email rule에 있는 에러 메시지가 아닌 아래에서 key로 지정한 에러
                                        메시지를 출력한다. –>
                                <arg key="userInfoForm.email" />
                        </field>
                </form>
        </formset>

</form-validation>
```

위는 이번 예제에서 사용할 validation.xml 파일이다. 이 파일을 /WEB-INF/config 디렉토리에 작성하여 저장한다.

- <global><**constant/**></global>에서 constant 는 반복적으로 사용되는 문자열을 상수로 선언하는 것이다. 여기서는 postalCode 라는 이름으로 상수를 만들고 정규표현식 문자열을 값으로 지정하였다. 이것을 <form> 부분에서 **${postalCode}** 라는 식으로 사용하게 된다.
- 실제로 ActionForm의 프라퍼티와 검증 규칙의 연결은 <formset><**form>..</form**></formset>에서 시작된다.
  - name 속성 : struts-config.xml에서 지정한 ActionForm의 이름이다.

- <form><**field**>...</**field**></form>부분에서 액션 폼의 각 필드와 검증 규칙을 연결한다.
  - property 속성 : 액션 폼의 프라퍼티이다.
  - depends 속성 : validator-rules.xml에 선언된 각 검증 규칙의 이름이다. 여기서 지정된 검증 규칙을 통과해야만 유효성 검사를 통과한 것이다. 검증 규칙이 여러개 필요할 때는 쉼표로 구분한다. 순서대로 검증이 실행된다.
  - <msg> 요소 : validator-rules.xml에 지정된 에러 메시지가 아닌 다른 에러메시지를 사용하기 위해 지정하는 요소
    - name 속성 : 어떤 검증 규칙에서 에러가 발생했을 때 출력할 메시지인지 지정한다. depends 속성에 있는 값들중 하나여야 한다.
    - key 속성 : 리소스 번들 프라퍼티의 어떤 키의 값을 에러 메시지로 사용할지를 결정한다.
  - <arg> 요소 : 오류 발생시 오류 메시지의 {0}, {1}, .. 부분에 삽입할 메시지들을 리소스 번들 프라퍼티에서 가져오도록 지정한다.
    - key 속성 : 리소스 번들 프라퍼티의 키
  - <msg>와 <arg>요소의 **resource** 속성 : resource 속성을 "**false**"로 두면 리소스 번들에서 키에 해당하는 값을 검색해서 가져오는 것이 아니라 key 속성에 지정된 문자열을 그대로 사용한다.
  - <var> 요소 : 유효성 검증기에 넘길 파라미터이다. 일반적으로 파라미터가 필요한 경우는 많지 않은데, mask 규칙의 경우에는 파라미터가 필요하다. mask 규칙은 액션 폼의 프라퍼티가 파라미터로 넘겨준 정규 표현식(regular expression)과 일치하는지 검사하는 규칙이다.
    - <var-name> 요소 : 파라미터 이름
    - <var-value> 요소 : 파라미터 값

여기 validation.xml 에서는 많은 리소스 번들 값들을 생성했다. 아래는 이것들을 포함한 최종 application.properties 파일의 내용이다.

```
error.invalidUsername=잘못된 사용자명입니다. {0}
error.invalidPassword=비밀번호를 입력하지 않았습니다.
error.invalidLogin=로그인 사용자명이 존재하지 않거나 비밀번호가 일치하지 않습니다.

# 메시지 출력시 상/하단에 출력될 머릿말과 꼬릿말
message.header=<ul>
message.footer=</ul>

# Struts Validator Error Messages
errors.required={0}(이)가 필요합니다.
errors.minlength={0}(은)는 최소한  {1} 문자 이상이어야 합니다.
errors.maxlength={0}(은)는 최대  {1} 문자 이하이어야 합니다.
errors.invalid={0}(이)가 유효하지 않습니다.
errors.byte={0}(은)는 byte 형이어야 합니다.
errors.short={0}(은)는 short 형이어야 합니다.
errors.integer={0}(은)는 integer 형이어야 합니다.
errors.long={0}(은)는 long 형이어야 합니다.
errors.float={0}(은)는 float 형이어야 합니다.
errors.double={0}(은)는 double 형이어야 합니다.
errors.date={0}(은)는 날짜형이 아닙니다.
errors.range={0}(은)는 {1}에서 {2} 사이 값이어야 합니다.
errors.creditcard={0}(은)는 유효하지 않은 신용카드 번호입니다.
errors.email={0}(은)는 유효하지 않은 이메일 주소입니다.

# messages for Validation
userInfoForm.username=사용자명
userInfoForm.address=주소
userInfoForm.postcode=우편번호
userInfoForm.postcodeError={0}가 형식에 맞지 않습니다. \
우편번호는 123-456 과 같은 형식을 따릅니다.
userInfoForm.email=이메일 주소
```

- **struts-config.xml 작성하기**

Validator를 위한 struts-config.xml 의 추가 사항은 다음과 같다.

아래 내용을 그대로 복사하면 안된다. <form-bean>은 <form-beans>안에, <action>은 <action-mapping> 안에, 그리고 <plug-in>은 <action-mappings>이나 <message-resources>보다 더 뒤의 다른 plug-in과 같은 위치에 두어야 한다.

```
        <!– Validator 테스트용 액션 폼 –>
        <form-bean
                name="userInfoForm"
                type="strutsguide.forms.UserInfoForm">
        </form-bean>

        <!– Validator용 사용자 정보 입력/보여주기 화면 –>
        <action path="/validator/inputUserInfo" forward="/validator/inputUserInfo.jsp"/>

        <!– 입력된 사용자 정보를 Validator로 검증한 뒤에 request 스코프에 저장하고
                userInfo.jsp 로 포워딩하는 액션 –>
        <action
                path="/validator/addUserInfo"
                type="strutsguide.actions.AddUserInfoAction"
                name="userInfoForm"
                scope="request"
                validate="true"
                input="/validator/inputUserInfo.jsp"

                >
                <forward name="success" path="/validator/viewUserInfo.jsp"/>
        </action>

<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
        <set-property
                property="pathnames"
                value="/WEB-INF/config/validator-rules.xml,/WEB-INF/config/validation.xml"/>
</plug-in>
```

스트럿츠에게 Validator를 사용하겠다는 의사 표시를 하려면 ValidatorPlugIn을 설정해야 한다. 바로 <plug-in>부분이다. <set-property>를 이용해서 validator-rules.xml과 validation.xml 파일의 경로를 알려줘야만 한다.

먼저 /validator/inputUserInfo.do 로 접속해서 폼을 채워 넣으면 UserInfoForm 액션 폼에 값을 채우고, 그 값들을 Validator가 유효성 검증을 거친다음 유효성 검증을 통과하면 /validator/adduserInfo.do 액션을 수행하고서 /validator/viewUserInfo.jsp 에서 그대로 출력하는 구조이다.

액션 폼의 validate() 메소드를 호출하는 대신 그것을 Validator 프레임워크가 가로채는 것 빼고는 다를 것이 없다.

- **UserInfoForm 액션 폼**

Validator를 위해 액션 폼을 만들 때는 ActionForm 클래스를 상속받지 않고 **ValidatorForm** 클래스를 상속받아야 한다.

그리고 당연히 validator() 메소드를 만들 필요가 없다. Validator 프레임워크에 의해 자동으로 수행 될것이므로.

```
package strutsguide.forms;

import org.apache.struts.validator.ValidatorForm;

/**
 * Validator 테스트를 위한 사용자 정보 입력 액션 폼
 */
```

```
public class UserInfoForm extends ValidatorForm {

    /** 이름 */
    private String name = null;

    /** 주소 */
    private String address = null;

    /** 우편번호 */
    private String postcode = null;

    /** 이메일 */
    private String email = null;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPostcode() {
        return postcode;
    }

    public void setPostcode(String postcode) {
        this.postcode = postcode;
    }

    // validate() 메소드는 구현하지 않는다.
}
```

모두 required 규칙을 만족해야 하며 우편번호의 경우에는 123-456 처럼 “숫자3개-숫자3개” 형태의 정규 표현식(^\d{3}-\d{3}$)을 mask 검증 규칙에 적용해서 유효성을 검사한다. mask 검증 규칙은 지정된 정규표현식과 프라퍼티의 값이 일치하는지 검사하는 것이다.

이메일주소의 경우에도 email 검증 규칙을 만족하는지 검사한다. (validation.xml 참조)

- **/validator/inputUserInfo.jsp 작성하기**

이 JSP 페이지는 단순히 사용자로부터 이름과 주소, 우편번호 그리고 이메일 주소를 입력 받는다. Validator에 의한 검증이 실패하면 다시 이 페이지로 돌아와서 오류 메시지와 직전에 입력했던 내용들을 폼에 채워서 보여준다.

```
<%@ page contentType="text/html; charset=euc-kr"%>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean"%>
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
        <head>
    <meta http-equiv="Content-Type" content="text/html; charset=euc-kr">
                <title>Validator 테스트 - 사용자 정보 입력 폼</title>
                <html:base/>
        </head>
        <body>

                <h3>사용자 정보를 입력해주세요.</h3>

                <!– 오류 메시지 출력 부분 –>
                <html:messages id="msg" header="message.header" footer="message.footer">
                        <li><b><bean:write name="msg" /></b></li>
                </html:messages>

                <html:form action="/validator/addUserInfo" method="POST" focus="name">
                        이름 : <html:text size="20" property="name" /><br />
                        주소 : <html:text size="80" property="address"/><br />
                        우편번호 : <html:text size="7" property="postcode" maxlength="7" /><br />
                        이메일 : <html:text size="20" property="email"/><br />
                        <html:submit value="입력"/>   <html:reset value="재입력"/>
                </html:form>
        </body>
</html:html>
```

오류 메시지에 보면 header와 footer 속성이 있는데, 이것은 오류 메시지들을 출력하기 전에 상단과 하단에 추가할 텍스트를 의미하는 메시지 리소스 번들 키를 뜻한다. 여기서는 과 로 지정한 것이다.

- **AddUserInfoAction 액션 작성하기**

```
package strutsguide.actions;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import strutsguide.forms.UserInfoForm;

/**
 * 입력된 사용자 정보가 Validator로 검증이 끝난뒤에 이 액션에 오게 된다.
 * 이 액션에서는 단순히 각 값을 request 에 속성으로 저장한다.
 */
public class AddUserInfoAction extends Action {

    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        UserInfoForm uiform = (UserInfoForm)form;

        // ValidateActionForm 에서 정보 추출
        String name = uiform.getName();
        String address = uiform.getAddress();
        String postcode = uiform.getPostcode();
        String email = uiform.getEmail();
```

```
        // 각 정보를 request 스코프에 저장한다.
        request.setAttribute("name", name);
        request.setAttribute("address", address);
        request.setAttribute("postcode", postcode);
        request.setAttribute("email", email);

        return (mapping.findForward("success"));
    }

}
```

당연히 사용자의 입력값이 Validator를 통과 했을 때만 액션을 수행한다.

여기서 Action은 사실상 아무 기능도 없이 단순하게 사용자의 입력값을 request 스코프에 Attribute로 저장한다. 그리고는 viewUserInfo.jsp로 포워딩을 한다.

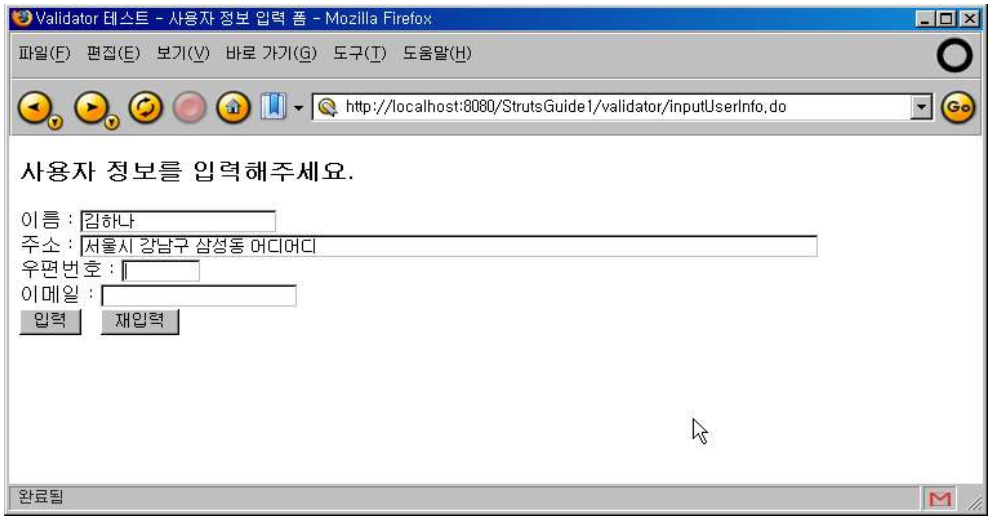
- **/validator/viewUserInfo.jsp 작성하기**

```
<%@ taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
        <head>
    <meta http-equiv="Content-Type" content="text/html; charset=euc-kr">
                <title>Validator 테스트 - 사용자 정보 출력</title>
                <html:base/>
        </head>
        <body>

                <h3>입력받은 사용자 정보</h3>

                <ul>
                        <li><b>이름</b> : <bean:write name="name"/></li>
                        <li><b>주소</b> : <bean:write name="address" /></li>
                        <li><b>우편번호</b> : <bean:write name="postcode"/></li>
                        <li><b>이메일</b> : <bean:write name="email"/></li>
                </ul>
        </body>
</html:html>
```

request 에 저장된 Attribute 들을 <bean:write>를 이용해서 출력한다.

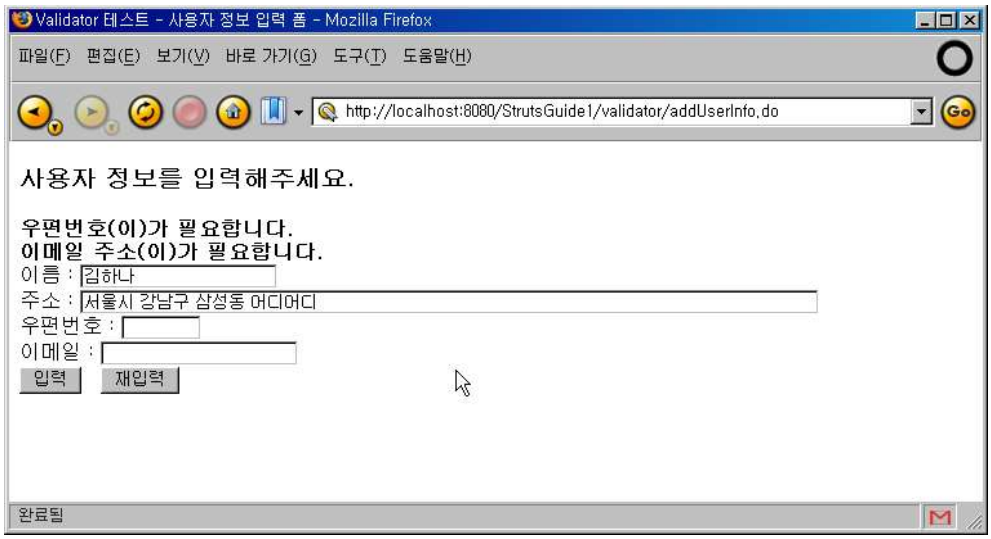
간단하다~!

- **화면을 다시보면서 흐름을 따라가보자**
    - 첫 화면 /validator/inputUserInfo.do 로 접속해보자.



보다시피 아무것도 없다. 이름과 주소만 입력하고 "입력"버튼을 눌러보면 다음처럼 오류메시지가 나오면서 기존에 입력했던 값들이 다시 그대로 보인다.

- 첫번째 오류화면

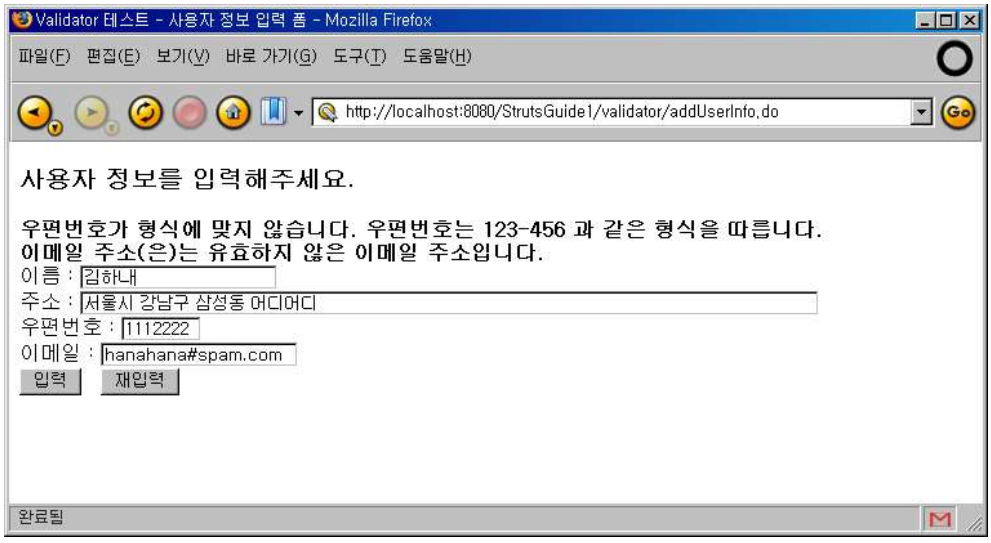


여기서 보면 한글이 안깨졌는데, 원래는 깨져야한다(?). 우리는 JSP 페이지나 Action 등 어디에서도 request.setCharacterEncoding("euc-kr")을 해준적이 없기 때문이다.

하지만 이 예제에서 한글이 안 깨진 이유는 몇 챕터 전에 새롭게 작성한 RequestProcessor를 상속받은 Controller 클래스 때문이다. (실제로 여기서는 TilesProcessor를 상속받았다. 바로 앞의 타일즈 예제 때문이다.)

우편번호와 이메일은 아예 입력을 안했을 때와 엉뚱한 값을 입력했을 때의 메시지가 다른 것을 확인해 보자.

- 우편번호와 이메일 잘못 입력했을 때

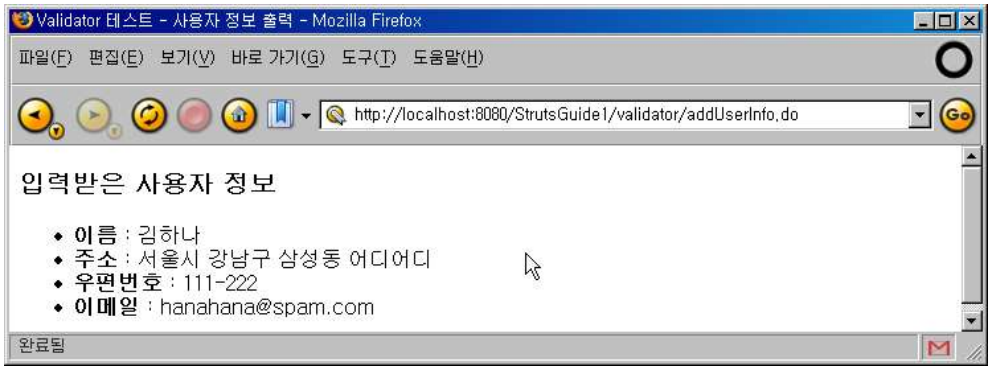


우편번호는 원래 111-222 처럼 입력해야 하는데 숫자로만 입력했고 이메일은 @ 대신에 #을 입력했다. 위 처럼 아까와는 다른 형식의 메시지가 나온다.

이유는 처음에 나온 에러 메시지는 required 검증 규칙에서 발생한 에러 메시지이고 그 다음 것은 각각 mask 와 email 검증 규칙에서 발생한 에러 메시지이기 때문이다.

그리고 특히 우편번호의 경우에는 validation.xml에서 <msg> 태그를 이용해서 mask의 기본 에러 메시지가 아닌 다른 에러 메시지를 사용하도록 했는데 그것이 출력되었음을 볼 수 있다.

- 이제 정상적으로 입력했을 때는?



모든 값을 정상적으로 입력하면 addUserInfo.do가 /validator/viewUserInfo.jsp로 포워딩되어 각 입력값을 보여주게 된다.

9. 예외 처리하기(Exception Handling)

- Action을 수행하는 도중 예외가 발생했을 때 예외를 사용자의 웹 브라우저에 바로 출력해버리지 않고 처리하는 과정을 거칠 수 있게 해준다.
- 특정 Action만을 위한 예외 핸들러는 <action>의 자식 노드로 <exception>을 추가하여 설정하고, 전역적으로 사용될 예외 핸들러는 <gloabal-exceptions>에 추가한다.

• <exception>

```
<exception
   type="org.apache.struts.webapp.example.ExpiredPasswordException"
   key="expired.password"
   path="/changePassword.jsp"
   handler="ExceptionHandlerClass" />
```

- type : 예외 클래스를 의미한다.
- key : 리소스번들 프라퍼티에서 가져올 메시지의 키 문자열
- hander : 예외를 처리할 예외 핸들러 클래스. ExceptionHandler를 상속받아야만 하며, 예외 핸들러를 거친 다음에 path에 지정된 페이지로 이동한다.
- path : 예외 처리후 사용자에게 보여줄 JSP 페이지 경로.

• path 지정시

- path에 지정된 페이지로 포워딩 하며, 그 페이지에서는 메시지 리소스 번들로 부터 key에 해당하는 메시지를 찾아 출력해준다.
- handler 없이 path만 지정해도 실제로는 hander="org.apache.struts.action.ExceptionHandler"를 지정한 것과 동일하게, 디폴트 예외 핸들러로서 ExceptionHandler를 수행한 뒤에 path에 지정된 JSP페이지로 이동한다.
- path 속성을 생략했다면 디폴트 예외 핸들러가 ActionMapping의 input 프라퍼티에 지정된 페이지로 포워딩 시킨다.
- 기본 ExceptionHandler는 JSP로 이동하기 전에 org.apache.struts.Globals.EXCEPTION_KEY에 지정된 문자열을 키로 이용해 request에 Exception 객체를 저장한다. 또한 key로 지정된 메시지를 ActionMessage 객체로 저장해서 JSP에 넘겨준다. JSP 페이지에서는 그 Exception 객체와 메시지를 가져다 사용할 수 있다.

• 예외 핸들러의 등록

다음 처럼 struts-config.xml에 디폴트 핸들러로 Exception 예외를 처리하는 예외 처리 설정을 등록한다.

Exception 은 모든 예외의 부모 클래스이기 때문에 Action에서 발생한 예외중에서 다른 예외 핸들러에 의해 잡히지 않는 모든 예외들이 지금 지정한 예외 처리 설정에 의해 처리되게 된다.

그리고는 무작정 예외만을 발생시키는 /exception/throwException.do 액션을 등록한다.

```
<global-exceptions>
      <!-- 예외 핸들러가 등록되지 않은 Action에서 발생한 모든 예외에 대한 처리 -->
      <exception
            type="java.lang.Exception"
            key="exception.all"
            path="/exception/exception.jsp"
            />
</global-exceptions>

<!- 무작정 예외만 발생시키는 액션을 등록한다. ->
<action
      path="/exception/throwException"
      type="strutsguide.actions.ThrowExceptionAction"  />
```

- **ThrowExceptionAction**

아래는 항상 Exception, NullPointerException, IOException을 무작위로 발생시키기만 하는 스트럿츠 액션이다.

```
package strutsguide.actions;

import java.io.IOException;

import org.apache.struts.action.Action;
import org.apache.struts.action.*;
import javax.servlet.http.*;

/**
 * 일부러 예외를 발생시켜 기본 global exception 핸들러를 사용해본다.
 */
public class ThrowExceptionAction extends Action {

    public ActionForward execute(ActionMapping mapping,
            ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) throws Exception {

        long currentTime = System.currentTimeMillis();

        int exType = (int)(currentTime % 3);

        if (exType == 0) {
            throw new Exception("일반 Exception 발생.");
        } else if (exType == 1) {
            throw new NullPointerException("NullPointerException 발생.");
        } else if (exType == 2) {
            throw new IOException("IOException 발생.");
        }

        // 무조건 예외를 발생시키므로 여기까지 오지도 못한다.
        return null;
    }
}
```

- **/exception/exception.jsp**

액션에서 발생한 예외는 예외 핸들러를 거쳐 <exception path="/exception/exception.jsp"/>로 지정된 아래의 JSP로 전달되게 된다.

아래에서는 스크립틀릿을 사용하여 예외의 메시지를 출력했는데, 실제 프로그램은 이런식으로 스크립틀릿은 사용하는 것은 피해야 하겠다. 이건 그냥 이렇게 하는 것도 가능하다는 예제일 뿐이다.

```
<%@ page contentType="text/html; charset=euc-kr"%>
<%@ page import="org.apache.struts.Globals" %>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html"%>
<%@ taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title>예외 처리 페이지</title>
    </head>

    <body>
        <h2>예외 핸들러가 전달한 에러 메시지 목록</h2>
        <html:messages id="error" header="message.header" footer="message.footer">
            <li><bean:write name="error"/></li>
```

```
            </html:messages>
            <br />
            <h2>Exception 객체로 부터 가져온 예외 메시지</h2>
            <blockquote style="background: wheat;">
            <%
                  Exception ex = (Exception)request.getAttribute(Globals.EXCEPTION_KEY);
                  if (ex == null) {
                        out.println("ex == null");
                  } else {
                        out.println(ex.getMessage());
                  }
            %>
            </blockquote>
      </body>
</html>
```

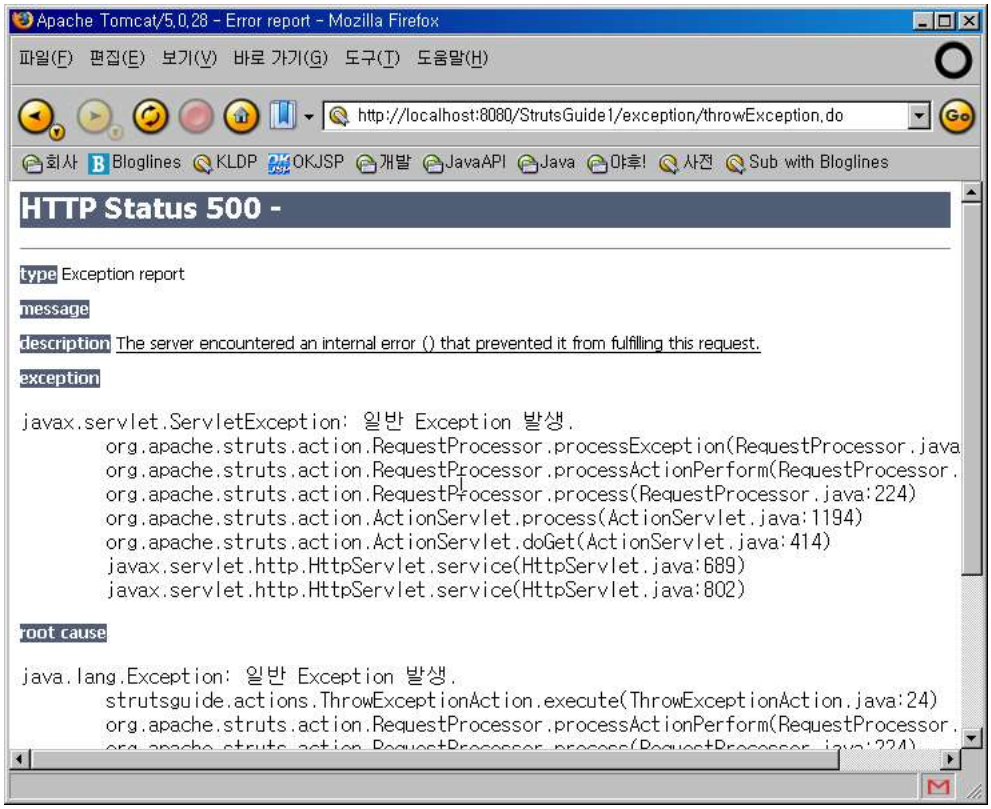
<html:messages>를 이용해서 <exception key=""/>의 key로 지정된 메시지를 출력해준다. 그리고 스크립틀릿 부분에서는 Globals.EXCEPTION_KEY로 request 스코프에 저장된 예외 객체의 메시지를 받아다 출력해준다.

- **application.properties에 추가**

  리소스 번들 application.properties에 다음을 추가한다.

```
# for exception handler
exception.all=오류가 발생했습니다.
exception.sghandler={0} : 오류가 발생했습니다. 오류 메시지 : {1}
```

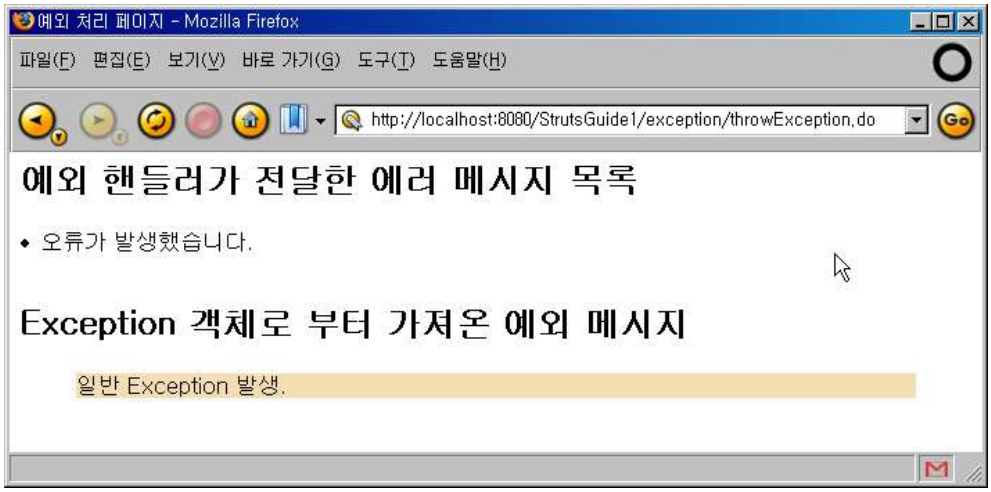
- **수행화면**
  - 아래 화면은 struts-config.xml 에서 <global-exceptions> 이하 부분을 없앤 상태(즉, 예외 핸들러를 지정하지 않은 상태)에서의 /exception/throwException.do 호출 화면이다.



보는 것과 같이 이럴 경우에는 웹 컨테이너(여기서는 Tomcat)에 의해 예외 메시지와 예외 스택 트레이스가 화면에 출력되게 된다.

실제 웹 어플리케이션에서는 이런식으로 일반 사용자들이 예외 스택 트레이스를 보게 되는 일이 있어서는 안 된다.

- 다음 화면은 <globa-exceptions>를 지정한 상태에서의 화면이다.



• 예외 핸들러 지정시

ExceptionHandler를 **상속**받은 예외 핸들러 클래스의 execute() 메소드를 수행한다. execute() 메소드는 예외에 대한 처리를 수행한 다음에 Action과 마찬가지로 ActionForward를 반환하면 그 것으로 이동하게된다.

```
public ActionForward execute(
            Exception ex,
            ExceptionConfig ae,
            ActionMapping mapping,
            ActionForm formInstance,
            HttpServletRequest request,
            HttpServletResponse response) throws ServletException {
      //....

      return 액션포워드;
}
```

• 사용자 정의 예외 핸들러의 등록

이 번에는 <global-exceptions>가 아니라, <action>의 로칼 예외 핸들러로 등록해보자. 이 예외 핸들러는 해당 <action>에서만 작동하게 된다. 그리고 전역적으로 지정된 동일한 예외를 처리하는 예외 핸들러가 있을 경우에는 로칼 예외 핸들러가 우선적으로 실행된다.

여기서는 IOException과 NullPointerException에 대해서만 예외 핸들러를 등록했다. 그렇기 때문에 이 두가지를 제외한 다른 모든 예외는 자동으로 전역 핸들러로 전달되어 처리되게 된다.

기존에 등록했던 /exception/throwException.do 액션 설정 부분을 아래와 같이 살짝 바꿔주자.

```
      <!-  일부러 예외를 발생시킨다. ->
      <!- 주석처리
      <action
            path="/exception/throwException"
            type="strutsguide.actions.ThrowExceptionAction"  />
      ->

      <action
            path="/exception/throwException"
            type="strutsguide.actions.ThrowExceptionAction" >
```

```
        <!– NullPointerException과 IOException을 처리하는 핸들러 지정 –>
        <exception
            key="exception.sghandler"
            handler="strutsguide.exhandlers.SGExceptionHandler"
            path="/exception/exception.jsp"
            type="java.lang.NullPointerException"/>

        <exception
            key="exception.sghandler"
            handler="strutsguide.exhandlers.SGExceptionHandler"
            path="/exception/exception.jsp"
            type="java.io.IOException"/>

    </action>
```

- **SGExceptionHandler 작성하기**

ExceptionHandler를 상속받아 만든 SGExceptionHandler는 다음과 같다.

```
package strutsguide.exhandlers;

import java.util.Date;

import org.apache.struts.action.*;
import org.apache.struts.config.ExceptionConfig;

import javax.servlet.ServletException;
import javax.servlet.http.*;

import org.apache.commons.logging.*;

/**
 * 스트럿츠 ExceptionHandler 예제.
 * Action에서 try/catch로 잡지 못하고 그대로 throw 된 예외들이
 * struts-config.xml의 &lt;exception&gt; 설정에 따라 이곳으로
 * 전달된다.
 */
public class SGExceptionHandler extends ExceptionHandler {

   Log log = LogFactory.getLog(SGExceptionHandler.class);

   /**
    * Action에서 예외가 발생하면 &lt;exception&gt; 설정에 따라 아래를 실행한다.
    */
   public ActionForward execute(Exception ex,
         ExceptionConfig config,
         ActionMapping mapping,
         ActionForm form,
         HttpServletRequest request,
         HttpServletResponse response) throws ServletException {

      // 발생한 예외시간을 포함해서 새로운 문자열로 구성한다.
      String [] args = new String[2];

      long currentTimeMillis = System.currentTimeMillis();
      Date currentTime = new Date(currentTimeMillis);

      // 현재 시간을 인자로 저장.
      args[0] = currentTime.toString();
      // 오류 메시지를 인자로 저장.
      args[1] = ex.getMessage();

      // 지정된 키의 메시지를 메시지 리소스 번들에서 가져와 {0}, {1}을 지정된
```

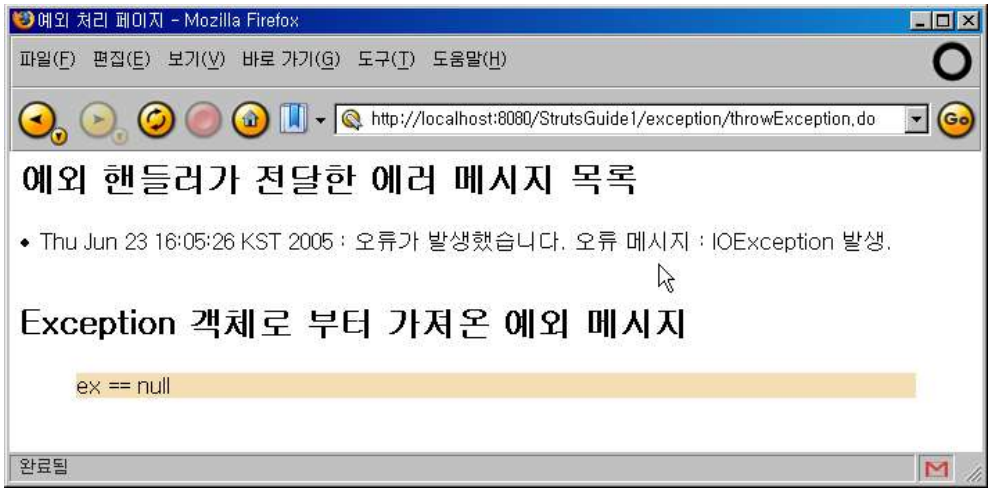
```
        // 인자로 대체하여 새로운 메시지 구성.
        ActionMessage errorMessage = new ActionMessage(config.getKey(), args);

        // 예외 정보를 표준 에러 스트림으로 출력한다.
        System.err.println(errorMessage.toString());
        ex.printStackTrace();


        // 예외 정보를 가지고 포워딩할 JSP페이지를 찾는다. <exception path=""/>에 지정된 페이지
        ActionForward forward = new ActionForward(config.getPath());

        // 예외 사항들을 request에 저장한다.
        storeException(request, config.getKey(), errorMessage, forward, config.getScope());

        // 포워딩~!
        return forward;
    }
}
```

이 SGExceptionHandler은 예외를 전달받게 되면 <exception **key="exception.sghandler"**/> 메시지를 가져다가 현재 시간과 ex.getMessage()를 실행하여 예외의 메시지를 합쳐서 ActionMessage 객체를 구성하게 된다.

그리고 **config.getPath()**를 이용해서 <exception **path="/exception/exception.jsp"**/> 로 지정된 포워딩할 JSP 주소를 얻어다 ActionForward 객체를 구성한다.

최종적으로 **storeException()**메소드를 실행하여 에러 메시지를 지정된 스코프(**config.getScope()**)에 저장하고는 포워드 객체를 리턴한다.

이제 IOException 이나 NullPointerException이 발생하면 아래와 같은 화면을 보게 된다.



하지만, 그냥 Exception이 발생하면 <global-exceptions>에서 지정한 예외 핸들러로 예외를 처리하게 되기 때문에 이전 예제와 같은 화면을 보게 된다.

이번 예제에서 예외가 발생했을 때 날짜 정보등을 추가하여 JSP로 보냈는데, 실제 웹 어플리케이션에서는 이렇게 하지 않는 것이 좋겠다.

Logger를 사용하여 예외 객체를 넘기고 오류 메시지를 출력하고, 사용자는 단지 친절하게 "로그인을 해주세요." 같은 메시지만을 받도록 해야 겠다. 로거의 사용법은 다음에 나오는 장에 설명되어 있다.

또한 이 화면에 보면 예외 객체가 JSP로 전달되지 않은 것을 볼 수 있는데(ex == null), 실제 웹 어플리케이션에서도 예외 객체를 JSP 페이지로 전달하는 일이 없어야 겠다. **예외에 관해 프로그래머가 알아야할 모든 정보는 예외 핸들러에서 로거를 이용해 저장하고 사용자는 단지 사용자에게 필요한 메시지만 보게 하자.**

10.어플리케이션 로그 남기기(Logging)

만약 당신의 웹 어플리케이션이 어떠한 것이든 로거를 사용하지 않고 "System.out.println()"으로 디버깅용 메시지를 찍는다면, 어플리케이션을 실제로 운영하다가 오류가 발생하면 십중팔구 길고긴 밤샘의 나날을 보내게 될 것이다.

굳이 로깅의 중요성을 구구절절이 나열하지 않겠다. **그냥 무조건 로거를 사용하라! 내 웹 어플리케이션에는 절대로 단 한줄의 System.out.println() 도 용납하지 않겠다!!!**

로깅이 뭐하는 건지 모르겠다면 간단히 설명하면, 각 메시지의 중요도에 따라 프로그래머가 지정한 메시지를 출력시켜주는 것이라고 말 할 수 있다.

하지만 중요도에 따라 마음대로 로거를 껐다 켰다 할 수 있으며, 단순 텍스트 메시지 외에도 로깅을 수행한 클래스의 이름과 소스코드 줄 번호, 쓰레드 번호, 메시지를 출력한 시간 등을 알 수 있게 해주며, 단순히 화면에 텍스트로 찍는 것 뿐만 아니라 소스 코드 변경이 전혀 없이 로거의 설정 파일 변경만으로 메시지를 이메일로 보낼 수도 있고, 파일로 저장할 수도 있고, 심지어는 메신저로 전송해 줄 수도 있는 그런 도구이다.

스트럿츠는 기본적으로 Jakarta Commons-Logging 패키지를 함께 제공한다. Commons-Logging 은 그 자체로 로깅을 수행하지 않는다. 다른 로거를 설정에 따라 호출하여 로깅을 수행하도록 한다.

굳이 Commons-Logging을 사용하는 이유는 특별한 로거에 대한 종속을 없애야 하기 때문이다.

어쨌든 여기서는 Commons Logging에 Log4j를 연결하여 로그를 남기도록 하겠다.

• Commons-Logging 사용법

스트럿츠를 설치했다면 Commons-Logging을 위해 특별히 할 일은 없다. 기본적으로 패키지가 포함되어 있기 때문이다.

스트럿츠를 사용하지 않아도 물론 Commons-Logging을 쓰는데 아무런 문제도 없다. 자세한 사항은 http://jakarta.apache.org/commons/logging 을 참조하면 된다.

- 로그를 남기기 위해 소스코드에서 할일

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import 기타등등..;

public class SomeAction extends Action {
   // 로거 객체를 얻어온다. 인자로 클래스이름.class 를 주는 것을 주의할 것.
   private static Log log = LogFactory.getLog(SomeAction.class);

   public ActionForward execute(ActionMapping mapping,
         ActionForm form,
         HttpServletRequest request,
         HttpServletResponse response) throws Exception {

      // 이런저런 작업...
      if (log.isDebugEnabled()) {
         // Debug 메시지를 사용할 때는 필수적으로 if (log.isDebugEnabled()) 검사를 할 것!
         log.debug("출력할 디버깅용 메시지...");
      }

      // 기타 등등 작업
      log.info("운영자에게 제공할 정보 로그");

      try {
         // 그외 작업
      } catch (Exception ex) {
         // 에러 로그 남기기. Exception Stack Trace 까지 남겨준다.
         log.error("에러 발생~~!!", ex);
      }
      // 어쩌구 저쩌구..
   }
```

```
}
```

- **log.debug()** : 개발자를 위한 디버깅용 메시지 남기기. debug()는 보통 운영중에는 끄게 마련이다. 이럴 때를 대비해 if (log.isDebugEnabled()) { } 를 사용하면 운영중에 debug 메시지 출력에 들어가는 시간을 단축할 수 있다.
- **log.info()** : 운영중에 운영자에게 제공할 정보. 불필요하게 남용하면 안된다. 운영중에 부하를 줄 수 있다.
- **log.error()**, **log.fatal()** : 일반적인 오류와 심각한 오류 발생에 관한 로그 메시지 남기기. Exception 객체까지 함께 넘겨주면 Exception StackTrace 까지도 로그로 남겨준다.
- 그 외의 로그 레벨은 Commons-Logging 관련 설명서를 참조한다.

• Log4j 설정하기

Commons-Logging은 실제 로깅을 수행하지 않는다. http://logging.apache.org/log4j/docs/index.html 에서 Log4j를 받아서 log4j-version.jar를 /WEB-INF/lib 에 복사하면 그 때부터 Commons-Logging이 Log4j를 자동인식하고 Log4j를 통해 로깅을 수행한다.

실제로 Log4j는 책 한권을 쓸만큼 기능이 많은데, 그런거 다 빼고 아래 내용만 자신의 **/WEB-INF/classes 에 log4j.properties 라는 이름으로 저장**해 두면 강력한 로깅의 힘을 느낄 수 있게 될 것이다.

```
# 개발시에는 DEBUG로, 운영시에는 INFO 모드로 바꾼다.
log4j.rootLogger = DEBUG, stdout, dailyfile

log4j.appender.stdout = org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout = org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p (%F[%M]:%L) [%d] - %m%n

log4j.appender.dailyfile = org.apache.log4j.DailyRollingFileAppender
# 아래 파일이름을 원하는 대로 절대 혹은 상대 경로로 줄 것.
log4j.appender.dailyfile.File = struts.log
log4j.appender.dailyfile.layout = org.apache.log4j.PatternLayout
log4j.appender.dailyfile.layout.ConversionPattern=%5p (%F[%M]:%L) [%d] - %m%n
```

- 개발시에는 DEBUG 모드로 개발하여 모든 디버깅 메시지를 보도록 하고 운영시에는 DEBUG 대신 INFO 로 바꾸어서 불필요한 메시지가 표시되지 않도록 한다.
- "log4j.appender.stdout.layout.ConversionPattern=%5p (%F[%M]:%L) [%d] - %m%n" 부분을 편집하면 원하는대로 자신만의 로그 메시지 구성을 할 수 있다. 이건 알아서~
- 혹시나 해서 더 글을 남기자면 Log4j는 단순히 log4j.properties 파일을 편집하는 것 만으로 로그 메시지를 이메일, 메신저, Socket 통신을 통해 원격 클라이언트에서 로그 메시지 보기 (http://traxel.com/lumbermill/) 등을 할 수 있다. 그 외에도 무궁무진한 방법으로 메시지를 사용자에게 전달 할 수 있다는 것을 알고는 있으시라.

11.ActionForm에 대한 정리

HTML 입력폼으로부터 받은 파라미터들을 일일이 request.getParameter("name");과 같은 형식으로 받지 않고, ActionForm 객체를 이용해 일괄적으로 파라미터를 받고 검증하여 Struts/Action 으로 넘겨주는 역할을 한다.

- **ActionForm은?**
  - HTML 폼("<input type='text' name='phone'/>" 형식)으로 부터 받은 입력은 ActionForm 빈으로 전달되고, 프라퍼티(파라미터)에 대한 검증이 끝나면 폼으로 부터 받은 입력 값을 잘 정돈된 자바빈(모델)으로 만들어 Action 에 전달한다.
  - 주의: **ActionForm 객체를 직접 이용해서 작업을 수행(비지니스 프로세스 수행 - 모델 부분)을 해서는 안된다! - 모델 부분은 컨트롤러와 뷰와 완전히 분리된 상태로 작성해야 한다.** ActionForm은 컨트롤러에 속한다.
- **ActionForm 요구사항**
  - ActionForm은 "org.apache.struts.action.ActionForm"을 상속한다.
  - 파라미터를 Action에 전달하기 전에 검증하는 validate() 구현.
  - 파라미터가 폼에 채워지기 전에 초기화 하는 "reset()" 구현. ActionForm이 폼에 채워지기 전에 호출된다.
  - ActionForm의 모든 프라퍼티는 String혹은 boolean 으로 지정한다. 그래야 잘못된 입력을 사용자에게 다시 보여주고 다시 입력하도록 할 수 있다.
  - HTML 폼에서 체크 박스가 공백일 경우에는 브라우저가 그에 대해 아무런 정보(체크박스 폼의 존재 자체도)도 전송하지 않는다. 그러므로 이에 대한 처리가 들어가야 한다.
  - 만일 애플리케이션이 몇 가지 ActionForm 클래스들을 사용한다면, 애플리케이션의 폼들에 공통으로 존재하는 프라퍼티나 유틸리티를 포함하는 기본 객체를 정의하여 상속해서 구현한다.
  - 아래는 ActionForm을 struts-config.xml 에서 설정하는 예이다.

```
<form-beans>
      <form-bean
            name="menuForm"
            type="myproject.form.MenuForm"/>
</form-beans>
```

    - name 속성 : 폼의 이름. Action에서 이 이름을 참조한다.
    - type 속성 : ActionForm 클래스.
- **DynaActionForm (Struts 1.1 이상)**
  - DynaActionForm은 Jakarta Commons DynaBean을 기반으로 하여 내부적으로는 Map으로 프러퍼티를 저장한다.
  - 실제 클래스를 작성할 필요 없이 type="org.apache.struts.action.DynaActionForm"으로 struts-config.xml 설정하여 ActionForm 구현.
  - reset() 메소드는 모든 필드를 초기화 해버린다.
  - validate()메소드를 구현하려면 DynaActionForm을 상속받아 직접 만들어야 한다. 보통은 validate()메소드를 만들지 않고 Validator Framework을 이용한다.
  - ActionForm 클래스가 지나치게 많아지는 것을 막아준다.
  - DynaActionForm 설정 예이다.

```
<form-bean
      name="myForm"
      type="org.apache.struts.action.DynaActionForm">
```

```
    <form-property name="name" type="java.lang.String"/>
    <form-property name="address" type="java.lang.String"/>
</form-bean>
```

<form-property>를 이용해서 각 프라퍼티의 이름과 타입을 지정한다.

• ActionForm의 값으로 지정된 다른 빈의 값 가져오기

```
<html:text property="values.telephoneText" size="14" maxlength="14"/>
```

위 처럼 "."을 이용하면 된다. aForm이 ActionForm 객체라고 가정할 때 위 태그는 **"aForm.getValues(). getTelephoneText()"**를 호출한다.

• Map 기반 ActionForm (Struts 1.1 이상)

일일이 ActionForm 클래스를 만들면서 getter와 setter를 만드는 것은 매우 고역이다. 대신 Map을 이용해서 간단하게 처리할 수 있다.

- ActionForm 클래스는 다음 두 메소드를 포함하고 Map 객체를 가지고서 맵 객체에 값을 설정하고 가져온다. 물론 코드는 알아서 구현하기~
  - public void setValue(String key, Object value)
  - public Object getValue(String key);
- JSP 에서는 다음과 같이 사용한다. value(key)를 이용해서 값을 설정하고 가져올 수 있다.

```
<html:text property="value(key)"/> <!– 폼 세팅 –>
<bean:write name="formBean" property="value(key)"/> <!– 값 출력 –>
```

• ActionForm 빈의 값을 모델의 빈 값에 셋팅하기

- 여러가지 방법이 있지만 그 중에 Jakarta Commons BeanUtils (http://jakarta.apache.org/commons/beanutils/, Struts에 기본적으로 포함되어 있음)를 이용하는 것이 제일 좋다.
- 이것은 ActionForm 이 자바 빈이고(맵이 아닐 때), 모델도 빈일 때 한쪽의 프라퍼티 값을 다른 쪽의 동일한 프라퍼티 값으로 세팅해 준다. 다음은 actionFormBean의 get 메소드들을 통해 값을 가져와 modelBean의 set메소드로 값을 설정해준다.

```
BeanUtils.copyProperties(modelBean, actionFormBean);
```

- 두 프라퍼티가 형이 다를 경우(예: actionFormBean에는 "String getAge()", modelBean에는 "void setAge(int)" 메소드가 있을 경우) BeanUtils가 자동으로 형 변환을 해준다. 즉, 다음과 같은 작업이 자동적으로 이뤄지게 된다.

```
int value = Integer.parseInt(actionFormBean.getAge());
modelBean.setAge(value);
```

- 원한다면 두 빈 사이의 형변환기를 직접 만들 수도 있다.

12.ActionForward에 대한 정리

- **ActionForward란?**
  - Action이 모든 작업을 끝내고서 이동하는 위치(JSP혹은 다른 Action 등..)을 가상적으로 지정한 것이 ActionForward이다.
  - 브라우저요청 -> ActionServlet -> [ActionForm] -> Action -> ActionForward의 path에 의해 지정된 다른 웹 요소(HTML,JSP,Action ...)

- **ActionForward 설정**
  - ActionForward는 가상의 포워드 이름과 웹 경로(path)로 이루어진다.
  - Path는 GET 방식으로 파라미터를 넘겨줄 수도 있다.

```
    <forward name="logoff" path="/Logoff.do"/>
    <forward name="logon" path="/logon.jsp" redirect="true"/>
```

  - "redirect=true"이면 Forward되지 않고 Redirect된다.
    - forward : request를 비롯한 컨텍스트가 보존된다. 동일 웹 어플리케이션에서만 사용가능하다. 웹 브라우저는 포워드 된 사실을 모른다.
    - redirect : 웹 브라우저가 새로운 HTTP요청을 생성하도록 한다. request 컨텍스트가 보존되지 않는다. 동일 웹 어플리케이션이 아니어도 된다.
  - 전역과 지역 포워드
    - 전역 포워드 : 동일 웹 어플리케이션의 모든 Action이 사용할 수 있는 포워드이다.

```
<global-forwards>
   <forward name="logon" path="/logon.do" redirect="true"/>
</global-forwards>
```

    - 지역 포워드 : ActionMapping 항목에 설정되며, 해당 Action 객체만 사용할 수 있는 포워드이다.

```
<action-mappings>
      <action path="/welcome"
            type="app.ContinueAction">
            <forward name="continue" path="/pages/Welcome.jsp"/>
      </acton>
</action-mappings>
```

- **ActionForward 사용**
  - 일반적인 사용

Action에서 ActionForward를 선택한다. 아래는 forward name이 "continue"인 것을 선택한다.

```
ActionForward forward = mapping.findForward("continue");
return forward; // Action 종료.
```

  - 포워드에 GET 방식 파라미터 추가하기

<html:link>에서 포워드에 설정된 URL에 링크 걸고, 파라미터 넘겨주기

```
<html:link forward="article" paramId="key" paramProperty="key" name="articleForm" useLocalEncoding="true">
   <bean:write name="articleForm" property="name"/>
</html:link>
```

```
<!-다음과 비슷한 모양의 HTML을 생성한다. ->
<a href="http://localhost/artimus/do/article?key=17">News from the edge</a>
```

    - 저기서 "key"라는 파라미터가 추가된다.

- useLocalEncoding은 파라미터를 현재 문자셋으로 URL 인코딩하도록 한다. 저 옵션이 없거나 false 이면 기본적으로는 UTF-8로 인코딩한다.(이것은 Struts 1.2 이상에서만 가능하다.)
- GET 파라미터를 사용할 때는 항상 파라미터 값의 URL 인코딩에 신경써야 한다. 절대 한글 상태 그대로 넘기지 말 것.

- Action 내부에서 포워드에 파라미터 추가하기

```
ActionForward forward = mapping.findForward("article");
StringBuffer path = new StringBuffer(forward.getPath()); // struts-config.xml 에 설정된 기본 forward 경로(path)를 가져온다.

boolean isQuery = (path.indexOf("?") >= 0); // 이미 다른 파라미터가 추가되어 있는가?

if (isQuery) {
    path.append("&dispatch=view"); // 값에 한글이나 특수문자가 들어갈 경우 URL Encoding을 잊지 말 것!
} else {
    path.append("?dispatch=view"); // 값에 한글이나 특수문자가 들어갈 경우 URL Encoding을 잊지 말 것!
}

// 여기서 path의 값은 대략 "/do/article?dispatch=view" 와 같은 형태가 된다.

return new ActionForward(path.toString()); // Action을 종료하고 포워드 객체를 넘겨준다.
```

- 동적인 포워드

<forward> 설정을 무시하고, 직접 URL을 지정한다.

```
ActionForward forward = new ActionForward("/do/itemEdit?action=Edit");
```

13.ActionMapping에 대한 정리

- **ActionMapping은?**
  - URL과 Action 객체를 연결해 준다.
  - 사용자가 특정 URL을 호출하면 struts-config.xml에 설정된 어떤 ActionForm에 폼 값을 채워야 하는지 판단하고, 필요하다면 ActionForm객체를 생성한 뒤에, Action을 호출한다.
  - ActionMapping은 Struts로 만든 어플리케이션의 흐름을 관장한다.

```
<action-mappings>
    <!– 사용자가 http://hostname/logoff.do를 호출하면 app.LogoffAction
        액션 객체를 호출하여 비지니스로직을 수행한다.–>
    <action path="/logoff" type="app.LogoffAction"/>

    <!– 사용자가 http://host/logonSubmit.do를 호출하면,
        액션 객체를 호출하기 전에 "logonForm" ActionForm 객체를 생성하여
        폼 값을 채워서 액션에 넘겨준다.
        ActionForm의 validate()가 false를 리턴하면 "/pages/Logon.jsp"로
        돌아간다. –>
    <action
        path="/logonSubmit"
        type="app.LogonAction"
        name="logonForm"
        scope="request"
        validate="true"
        input="/pages/Logon.jsp"/>

    <action
        path="/logon"
        type="app.ContinueAction">
        <forward name="continue" path="/pages/Logon.jsp"/>
    </action>
</action-mappings>
```

- **<action> 프라퍼티 설정**
  - path : 가상의 URI. ActionServlet의 서블릿 매핑을 *.do로 지정했다면, path는 ".do"를 제외하고 기입한다.

  예) path="login" -> 호출 URL은 http://host/login.do 가 된다.

  - forward (속성) : Action 호출없이 바로 다른 페이지로 forward한다. (RequestDispatcher.forward 이용)
  - include : Action 호출없이 다른 페이지를 include한다.
  - type : 가장 일반적으로 사용된다. 작업을 처리할 Action 클래스를 지정한다.
  - name : ActionForm을 지정한다. <form-bean>에서 설정된 이름(name)을 지정한다.
  - roles : 보안 권한 지정. 쉼표로 보안 권한을 지정한다. 보안 권한은 RequestProcessor의 "processRoles" 메소드를 확장해서 구현한다.
  - scope : ActionForm의 스코프를 지정한다.(주로 request, session)
  - validate : ActionForm의 validate()를 호출할지 여부. true/false
  - input : ActionForm의 validate()가 false를 리턴할 경우 input에 지정된 페이지로 제어가 넘어간다.
  - inputForward : input과 같지만, <forward>에 지정된 이름을 사용한다. <controller inputForward="true"/> 여야 사용가능
  - unknown : <action>이 지정되지 않는 ActionMapping URL(/*.do) 요청이 들어오면 보여줄 페이지를 지정한다.

```
<action name="/debug"
      unknown="true"
      forward="/pages/debug.jsp"/>
```

- <forward> (엘리먼트) : 현재 ActionMapping에만 적용되는 지역 ActionForward 설정.
- <exception> (엘리먼트) : 현재 ActionMapping에만 적용되는 지역 ExceptionHandler 설정.

14.Action에 대한 정리

- **Action이란?**
  - 기존 Java Web Application에서 서블릿과 비슷한 역할을 하며, 서블릿이 할 수 있는 모든 일을 할 수 있다.
  - 실질적인 비지니스 로직을 수행한다.
  - Action은 ActionServlet으로부터 요청(request 객체)와 응답(response 객체)의 처리를 위임 받은 단순하고 가벼운 Java Class이다.
  - Exception 처리
  - Control을 Action Forward를 통해 적절한 뷰로 돌린다.
  - ActionServlet은 Action객체의 execute(strute 1.1) 메소드나 perform(struts 1.0)을 통해 Action을 실행한다.

- **execute 메소드에 전달되는 파라미터**
  - mapping : 이 Action을 선택하는데 사용된 ActionMapping
  - form : 매핑에서 정의된 ActionForm
  - request : 요청 컨텍스트
  - response : 응답을 생성해줄 객체

- **Action 수행순서**
  - request parameter들의 검증 : ActionForm 객체를 이용하거나, request 객체의 정보를 통해 정확하게 파라미터를 받았고, 이 액션을 이용할 수 있는 권한이 있는 사용자 인가 등을 검증.
  - 비지니스 로직 호출 : 실질적인 작업 수행. : Action 클래스 내에 어떠한 비지니스 로직이라도 위치해서는 안된다. Action 클래스는 단지 비지니스 메소드가 필요로 하는 데이터를 전달하거나 받기만 해야 한다. 비지니스 메소드는 Action이 호출할 수 있는 분리된 클래스에 넣어야 한다.
  - Action은 에러를 감지한다.
  - ActionErrors와 ActionError 객체 그리고 saveErrors()를 호출해서 에러를 등록한다.

```
ActonErrors errors = new ActionErrors(); // 빈 ActionErrors 객체 생성.
// 오류가 발생하면
errors.add(ActionErrors.GLOBAL_ERROR, new ActionError("error.detail", e.getMessage());

//...

if (!errors.empty()) { // 에러가 발생했으면
   saveErrors(request, errors);
   return (mapping.findForward("error")); // 오류 처리 페이지로 포워드
}
```

error.detail = {0} 일 경우, e.getMessage()의 내용이 {0}을 대체하여 들어간다. 4개 까지 메시지 대체가 가능하다.

  - 에러를 받아들이는 JSP 페이지(mapping.findForward("error")에 지정된 페이지)는

```
<logic:messagesPresent>
   <UL>
      <html:messages id="error">
         <LI><bean:write name="error"/></LI>
      </html:messages>
   </UL>
</logic:messagesPresent>
```

  - 에러 메시지와 일반 메시지

- 에러메시지는 ActionErrors에 담아 saveErrors()로 저장하고, 일반 메시지는 ActionMessages 객체에 담아 saveMessages()로 저장한다.
- 뷰 JSP에서는 <logic:messagesPresent message="true">를 이용해 일반 메시지를 출력한다.
- ActionErrors.GLOBAL_ERROR로 설정하지 않고 errors.add("usernameError", new ActionError ("error.username.required"); 처럼 특별한 키를 줘서 에러 메시지를 저장했다면 JSP에서는 <html:errors property="usernameError"/>로 에러 내용을 출력할 수 있다.
- **Struts 1.2 부터는 에러 메시지에 대해서도 ActionErrors와 ActionError보다는 ActionMessages와 ActionMessage 사용을 권장**하고 있다.

• Exception

- Action.execute()에서 발생한 예외는 ExceptionHandler를 이용해 특정 뷰 JSP 페이지로 보낼 수 있다.

• ActionForward가 null

- ActionForward를 null로 리턴하면 이미 response에 응답을 끝냈다는 의미가 된다. 다른 어떤 페이지로도 이동하지 않는다.
- 이것을 이용해서 파일 다운로드 액션을 생성할 수 있다. Action에서 파일의 내용을 reponse 객체를 통해 전송한 뒤에 ActionForward를 null로 리턴하기만 하면 된다.

• 표준 Action

- **ForwardAction**
  - 컨트롤을 다른 리소스에 포워드 한다.

```
<action
      path="/saveSubscription"
      type="org.apache.struts.actions.ForwardAction"
      name="subscriptionForm"
      parameter="/포워딩할/주소" />
```

  - 요청을 포워딩 하기 전에 해당하는 Action Form Bean의 인스턴스를 만들고, 그것이 적합한지 검증하는 과정을 거친 후에 포워딩한다.
- **DispatchAction**
  - 여러개의 관련된 일을 처리하는데 동일한 Action 클래스를 사용하고자 할 때 DispatchAction을 상속받아 사용한다.
  - DispatchAction은 숨겨진 필드상의 키 값을 이용해 Action을 수행할 메소드를 선택한다.
  - 각 Dispatch 메소드는 perform()이나 execute()와 동일한 서명을 사용해야 하며, perform()과 execute()를 오버라이드하면 안된다.
  - 관련성이 있는 명령을 통합된 DispatchAction으로 구성하도록 하자. 연관된 명령을 같을 곳에서 수행하는 것은 작업 흐름과 유지보수를 단순화 시킨다.

```
<!– DispatchAction을 상속은 Action 클래스 recordDispatchAction –>
<action
      path="/dataRecord"
      type="app.recordDispatchAction"
      name="dataForm"
      scope="request"
      input="/data.jsp"
      parameter="method"/> <!– parameter 속성에 지정된 파라미터를 통해 실행할 메소드 결정 –>
```

  - 위 Action을 실행하기 위해 "http://localhost/app/dataRecord.do?method=delete"를 호출하면, method 파라미터의 값 delete에 따라 recordDispatchAction 클래스의 public ActionForward delete(...)가 실행된다.

- **Action 사용시 주의점**
  - 쓰레드에 안전하게 설계해야 한다. 하나의 Action 클래스에 대해 단 한 개의 객체만이 생성된다. 그러므로 쓰레드에 안전한 프로그램을 짜야 하며, **명백한 이유 없이 Action 객체의 멤버 변수를 생성해서는 안된다.**
  - 한 Action에서 공통 기능을 하는 메소드를 작성할 때 쓰레드 안전을 위해, **그 메소드 내에서 사용할 변수는 호출자로부터 인자로 받거내 메소드 내에서 자체적으로 생성해야한다.** 결코 명백한 이유 없이 Action 클래스 멤버 변수를 공용 메소드에서 사용해서는 안된다. - 이 말은, **Action의 메소드들끼리는 데이터를 공유할 수 없고 인자로만 전달해야 함을 의미한다.**
  - Action 사이에 공유해야 할 기능이 있다면 분리된 비지니스 클래스로 만들거나, Action 수퍼 클래스 안의 헬퍼 메소드로 생성하여 사용해야 한다. **Action 자체에 비지니스 로직을 작성하면 안된다!**
  - 만약 한 개의 요청을 처리하기 위해 여러 Action들 사이에 포워딩이 일어난다면 설계를 잘못한 것이다.
  - 가장 일반적인 사용은 한 개의 요청은 한 개의 Action에서만 실행하는 것이다.

15.PlugIn에 대한 정리

스트럿츠 플러그인은 ActionServlet이 처음 로딩될 때 함께 로딩되며 플러그인의 init()메소드가 실행되고, ActionServlet이 종료될 때(즉, 웹 어플리케이션이 종료될 때) destroy() 메소드가 실행되는 Action이다.

이것은 일반 Servlet의 init(),destroy()와 동일한 역할을 하는 것이다.

org.apache.struts.action.PlugIn 인터페이스를 구현하는 Action을 만들면 된다.

다음을 통해 struts-config.xml에 등록한다.

```
<plug-in className="myApp.MyAction">
   <set-property property="key" value="값"/>
</plug-in>
```

className 속성으로 명시된 Action 클래스는 PlugIn인터페이스를 구현하고, **init()과 destroy() 메소드를 제공**해야 한다.

16.이제 뭘 하지?

스트럿츠의 기본을 끝냈다.

이 문서는 스트럿츠의 흐름을 이해하기 위해 만든 것이지 스트럿츠의 모든 것을 알려주기 위한 것이 아니다. 하지만 지금까지 한 것만 가지고도 충분히 스트럿츠를 활용한 웹 어플리케이션을 작성할 수는 있을 것이다. 다른 좋은 책을 보면서 더 많은 기능을 익히기 바란다.

• 더 많은 기능들..

스트럿츠에는 여기에 소개한 것보다 훨씬 많은 기능들이 있다.

- DynaActionForm 을 이용해서 ActionForm 클래스 생성하지 않고 ActionForm 기능 사용하기.
- 국제화 기능 : 리소스 번들 프라퍼티 파일을 추가해주는 것 만으로 다국어 홈페이지 제작이 가능하다.
- 모듈 기능 : struts-config.xml 을 여러 파일로 분리하여 작성한다. 대규모 어플리케이션 작성시 필요하다.
- JSTL : 스트럿츠의 bean과 logic 보다 훨씬 뛰어난 JSP 표준 태그 라이브러리. bean과 logic 보다는 되도록 JSTL을 사용하기를 권한다.
- XDoclet (http://xdoclet.sf.net) : struts-config.xml과 실제 ActionForm, Action 클래스들을 동기화하는 것은 매우 짜증나고 실수할 여지가 많다. XDoclet을 이용하면 각 클래스에 단 주석에 따라 자동으로 struts-config.xml을 만들어준다. 이 또한 여러사람이 대규모 어플리케이션을 작성할 때 힘을 발휘할 것이다.
- 여기 소개한 기능들 더 깊이있게 공부하기 : 이 문서는 단지 스트럿츠의 작동 흐름만을 보여주기 위해 최소한의 기능만들 사용한 것이다. 훨씬 더 깊이 있게 사용하는 방법들이 많이 있다.

• 책

추천하는 책은 한빛 미디어의 "자카르타 스트럿츠 프로그래밍"과 같은 출판사의 "스트럿츠 프레임워크 워크북", 그리고 Oreilly "Struts Cookbook", http://www.sourcebeat.com 의 "Struts Live" 등이다. Struts Live는 책 내용중 일부(사실상 핵심은 모두 포함하고 있음)를 무료 PDF로 구할 수 있으며 (http://www.theserverside.com/books/sourcebeat/JakartaStrutsLive/index.tss) 그 내용이 매우 쉽다.

"자카르타 스트럿츠 프로그래밍"의 경우에는 어떻게 웹 프로그램을 작성하는 것이 훌륭한가와 웹 프로그래밍 전반에 관한 좋은 말들이 많다.

"Struts in Action"도 매우 좋은 책이다. 스트럿츠의 아주 세세한 부분을 설명해서 초보자가 읽기엔 부담스럽지만 스트럿츠의 기본을 이해한 상태에서 보기에는 무리가 없고, 거의 레퍼런스 수준의 정보를 제공해준다. 단, 번역판은 약간은 보기 껄끄럽다.

이 외에도 좋은 책들을 찾아 보기 바란다.

• 홈페이지

- http://www.okjsp.pe.kr 에서 스트러츠, 스트럿츠, Struts 등의 검색어로 검색해보라.
- http://www.javajigi.net 의 게시판과 위키에 보면 좋은 내용들이 많다.
- http://www.openseed.net : Java Open Source 프레임워크 이야기들..

아래 문서들을 **필독**하라.

- JSP/Servlet 및 JDBC 프로그래밍시 필히 알고 있어야 할사항 : http://www.javaservice.net/~java/bbs/read.cgi?m=devtip&b=servlet&c=r_p&n=968185187&k=JDBC&d=tb#968185187
- 자바 웹 프로그래머의 기본 : http://youngrok.com/wiki/wiki.php/%C0%DA%B9%D9%C0%A5%C7%C1%B7%CE%B1%D7%B7%A1%B8%D3%C0%C7%B1%E2%BA%BB
- J2EE 어플리케이션 튜닝 (웹 어플리케이션이 속도가 너무 느리다구?) : http://www.j2eestudy.co.kr/lecture/lecture_read.jsp?table=j2ee&db=lecture0201_1&id=24