Struts: Validación automática. Pruebas

Índice

Veremos en esta sesión un par de cuestiones que creemos que resultan interesantes en cualquier aplicación web con Struts: la validación automática de datos y el desarrollo dirigido por pruebas.

1. Validación de datos con el plugin Validator

El uso del método `validate` del `ActionForm` requiere escribir código Java. No obstante, muchas validaciones pertenecen a uno de varios tipos comunes: dato requerido, dato numérico, verificación de longitud, verificación de formato de fecha, etc. Struts dispone de un paquete opcional llamado **Validator** que permite efectuar distintas validaciones típicas de manera automática, sin escribir código. Validator tiene las siguientes características principales:

- Es configurable sin necesidad de escribir código, modificando un fichero XML
- Es extensible, de modo que el usuario puede escribir sus propios validadores, que no son más que clases Java, si los estándares no son suficientes.
- Puede generar automáticamente JavaScript para efectuar la validación en el cliente o puede efectuarla en el servidor, o ambas una tras otra.

1.1. Configuración

Antes que nada, necesitaremos incluir en nuestro proyecto el archivo .jar con la implementación de validator (`commons-validator-nº_version.jar`).

Validator es un plugin de struts. Los plugins que se van a usar en una aplicación deben colocarse en la etiqueta `<plugin>` del `struts-config.xml`. Esta etiqueta se coloca al final del fichero, inmediatamente antes de la etiqueta de cierre de `</struts-config>`

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

El atributo `value` de la propiedad `pathnames` indica dónde están y cómo se llaman los dos ficheros de configuración de Validator, cuya sintaxis veremos en el siguiente apartado. Podemos tomar el `validator-rules.xml` que viene por defecto con la distribución de Struts, mientras que el `validation.xml` lo escribiremos nosotros.

1.2. Definir qué validar y cómo

La definición de qué hay que validar y cómo efectuar la validación se hace, como ya se ha

visto, en dos ficheros de configuración separados:

- `validator-rules.xml`: en el que se definen los validadores. Un validador comprobará que un dato cumple ciertas condiciones. Por ejemplo, podemos tener un validador que compruebe fechas y otro que compruebe números de DNI con la letra del NIF. Ya hay bastantes validadores predefinidos, aunque si el usuario lo necesita podría definir los suyos propios (sería este el caso de validar el NIF). En la mayor parte de casos podemos usar el fichero que viene por defecto. Definir nuevos validadores queda fuera del ámbito de estos apuntes introductorios.
- `validation.xml`: en él se asocian las propiedades de los `actionForm` a alguno o varios de los validadores definidos en el fichero anterior. Las posibilidades de Validator se ven mejor con un fichero de ejemplo que abordando todas las etiquetas XML una por una:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
   "-//Apache Software Foundation//DTD Commons Validator
   Rules Configuration 1.0//EN"
   "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">
<form-validation>
  <formset>
    <form name="registro">
      <field property="login"
         depends="required,minlength">
         <arg0 key="registro.nombre"/>
         <var>
            <var-name>minlength</var-name>
            <var-value>5</var-value>
         </var>
      </field>
      ...
    </form>
  </formset>
  ...
</form-validation>
```

Simplificando, un `<formset>` es un conjunto de `ActionForms` a validar. Cada `ActionForm` viene representado por la etiqueta `<form>`. Dentro de él, cada propiedad del bean se especifica con `<field>`. Cada propiedad tiene su nombre (atributo `property`) y una lista de los validadores que debe cumplir (atributo `depends`). Si los mensajes de error que disparen los validadores tienen parámetros, estos se especifican con etiquetas `<arg0>`, `<arg1>`.... Además, algunos validadores tienen parámetros, por ejemplo el validador `minlength` requiere especificar la longitud mínima deseada. Los parámetros se pasan con la etiqueta `<var>`, dentro de la cual `<var-name>` es el nombre del parámetro y `<var-value>` su valor.

1.3. Los validadores estándar

Validator incluye 14 validadores básicos ya implementados, los cuales se muestran en la tabla siguiente

Validador	**Significado**	**Parámetros**
`required`	dato requerido y no vacío (no todo blancos)	ninguno
`mask`	emparejar con una expresión regular	`mask`: e.r. a cumplir
`range`	valor en un rango	`min`, `max`
`maxLength`	máxima longitud para un dato	`maxLength`
`minLength`	longitud mínima para un dato	`minLength`
`byte`,`short`, `integer`, `long`, `double`, `float`	dato válido si se puede convertir al tipo especificado	ninguno
`date`	verificar fecha	`datePattern`, formato de fecha especificado como lo hace `java.text.SimpleDateFormat`. Si se utiliza el parámetro `datePatternStrict` se verifica el formato de modo estricto. Por ejemplo, si el formato especifica dos días para el mes, hay que poner 08, no valdría con 8, que sí valdría con datePattern
`creditCard`	verificar número de tarjeta de crédito	ninguno
`email`	verificar direccion de e-mail	ninguno

el mensaje de error asociado al validador se supone en el fichero de recursos bajo la clave

```
errors.nombre-del-validador
```

No obstante, si dentro de algún `<field>` se desea mostrar un mensaje de error específico para un validador, se puede utilizar la etiqueta `<msg>`

```
<field property="login" depends="required, email">
   <msg name="required" key="login"/>
   ...
```

En la que el atributo `name` especifica el nombre del validador, y `key` la clave asociada al

mensaje.

1.4. Modificar el ActionForm para Validator

Si deseamos usar Validator, en lugar de `ActionForm` lo que debemos utilizar es una clase llamada `ValidatorForm`. Así nuestra clase extenderá dicha clase, teniendo un encabezado similar al siguiente

```
public class LoginForm extends
org.apache.struts.validator.action.ValidatorForm {
```

Cuando se utiliza un ValidatorForm ya no es necesario implementar el método `validate`. Cuando debería dispararse este método, entra en acción Validator, verificando que se cumplen los validadores especificados en el fichero XML.

1.5. Validación en el cliente

Se puede generar código JavaScript para validar los errores de manera automática en el cliente. Esto se haría con un código similar al siguiente:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
...
<html:form action="/login.do"
           onsubmit="return validateLoginForm(this)">
...
</html:form>

<html:javascript formName="loginForm"/>
```

El JavaScript encargado de la validación de un Form se genera con la etiqueta `<html:javascript>`, especificando en el atributo `formName` el nombre del `Form` a validar. En el evento de envío del formulario HTML (`onsubmit`) hay que poner "`return validateXXX(this)`" donde `XXX` es el nombre del Form a validar. Validator genera la función JavaScript `validateXXX` de manera que devuelva `false` si hay algún error de validación. De este modo el formulario no se enviará. Hay que destacar que aunque se pase con éxito la validación de JavaScript, Validator sigue efectuando la validación en el lado del servidor, para evitar problemas de seguridad o problemas en la ejecución del JavaScript.

2. Pruebas

Ya habéis visto a lo largo del curso la importancia de las pruebas en cualquier metodología moderna de desarrollo de software. Aunque en principio una aplicación Struts se podría probar con Cactus, que es una capa sobre JUnit que permite probar aplicaciones web, aquí

usaremos*StrutsTestCase* (http://strutstestcase.sourceforge.net/) que es una extensión de *JUnit* que permite probar las acciones de manera individual, comprobando los resultados, los ActionErrors generados, etc. *StrutsTestCase* permite utilizar tanto el enfoque basado en objetos Mock (si nuestro caso hereda de `MockStrutsTestCase`) como el basado en *Cactus* (usando `CactusStrutsTestCase`), lo que permite probar el código fuera o dentro del contenedor web, respectivamente.

Al utilizar el controlador `ActionServlet` para probar el código, este framework permite probar tanto la implementación de los objetos Action, como los mappings, beans de formulario y declaraciones *forward*.

2.1. Una breve introducción a StrutsTestCase

Para mostrar como podemos probar un Action, vamos a realizar la prueba de la entrada de un usuario a una aplicación.

En el siguiente trozo de código tenemos el esqueleto básico de una prueba StrutsTestCase basada en *Cactus*, donde heredamos de la indicada y en el cuerpo del método configuramos cual es el Action que deseamos probar (el cual debe existir en el archivo `struts-config.xml`). A continuación, realizamos la llamada al Action para su ejecución.

```
import servletunit.struts.CactusStrutsTestCase;

public class LoginActionTest extends CactusStrutsTestCase {

    public void testLogin() {
        setRequestPathInfo("/login");

        actionPerform();
    }

}
```

Normalmente, sobre un mismo *Action* se realiza más de una prueba, para comprobar su funcionamiento cuando las cosas no van bien.

```
public class LoginActionTest extends CactusStrutsTestCase {

    public LoginActionTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
```

```
        // Configuracion del ActionMapping. Lo ponemos aqui
        // pq es comun a todos los casos de prueba de esta clase
        setRequestPathInfo("/login");
    }

    public void testLogin() {

        // Configuramos el formulario
        LoginForm unForm = new LoginForm();
        unForm.setLogin("j2ee");
        unForm.setPassword("j2ee");
        setActionForm(unForm);

        // Llamada al Action
        actionPerform();

        // Comprobamos que redirige al ActionForward correcto
        verifyForward(Tokens.SUCCESS);

        // Comprobamos que no han habido errores
        verifyNoActionErrors();
    }

    public void testLoginKO() {

        // Configuramos el formulario
        LoginForm unForm = new LoginForm();
        unForm.setLogin("fdsagdsa");
        unForm.setPassword("gdsahaas");
        setActionForm(unForm);

        // Llamada al Action
        actionPerform();

        // Comprobamos que redirige al InputForward
        verifyInputForward();

        // Comprobamos que hay errores
        verifyActionErrors(new String[] {Tokens.ERROR_KEY_LOGIN_KO});
    }
}
```

Del código podemos observar como toda prueba tiene 4 partes claramente diferenciadas:

1. Configuración del *Action* a probar. Opcionalmente debemos configurar la ruta del archivo `struts-config.xml` (si no se encuentra en el classpath)
2. Preparación de los datos de entrada al *Action*, ya sea mediante:
   - la creación de un objeto *ActionForm* y posterior inclusión en la prueba
   - o inclusión de parámetros en el objeto *Request* (mediante el método `addRequestParameter(String nombreParametro, String valorParametro)`)

3. Llamada a la ejecución del *Action*.
4. Validación final de los datos de salida, forwards y/o mensajes de error.

2.2. API de StrutsTestCase

Este framework extiende JUnit y ofrece múltitud de métodos que permiten acceder a todos los recursos de Struts. El API del framework está bastante bien documentada, pero a modo de resumen, vamos a exponer cuales son los métodos más utilizados.

Método	Explicación
`setRequestPathInfo(String pathInfo)`	Indica el ActionMapping a probar
`setConfigFile(String configFile)`	Ruta del archivo `struts-config.xml`
`actionPerform()`	Pasa el control al *Action* a probar
`verifyForward(String forward)`	Comprueba si el controlador ha utilizado este forward
`verifyInputForward()`	Comprueba que el controlador ha redirigido al forward de entrada definido
`verifyNoActionErrors()`	Comprueba que el controlador no ha enviado mensajes de error tras ejecutar el *Action*
`verifyActionErrors(String[] nombresError)`	Comprueba que el controlador ha enviado estos mensajes de error