Struts

Sesión 3: Validación automática. Algunas buenas prácticas

Indice

- **El plugin *validator*. Instalación**
- Configuración de los validadores
- Buenas prácticas

Plugin validator

- Permite validar automáticamente sin necesidad de programar el validate()
  - **Configurable:** qué validar y cómo se especifica en un archivo XML
  - **Extensible:** podemos programar nuestros propios validadores
  - Puede validar también en el **cliente** (con JavaScript generado automáticamente)

Instalación de validator

- Importar el *commons-validator.jar* (incluído con la distribución de Struts)
- Indicar en el struts-config.xml que vamos a usar *validator* y cómo se llaman los 2 fich. de 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>
```

- Para validator-rules.xml se usa el que viene con Struts salvo que definamos validadores. La configuración se hace en validation.xml

Indice

- El plugin *validator*. Instalación
- **Configuración de los validadores**
- Buenas prácticas

Ejemplo de validation.xml

```
<!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="RegistroForm"> (ActionForm que se valida)
            <field property="login" depends="required,minlength"> (propiedad y validadores)
                    <var>      (parámetro que se le pasa al validador)
                       <var-name>minlength</var-name>
                       <var-value>5</var-value>
                    </var>
            </field>
    ...
    </form>
  </formset>
…
</form-validation>
```

Algunos validadores predefinidos

- required: el dato no puede ser vacío (sin parámetros)
- date: fecha en formato válido
  - var-name: datePattern, datePatternStrict (String en el formato usado por simpleDateFormat)
- mask: emparejar con una expresión regular
  - var-name: mask (la e.r.)
- intRange, floatRange, doubleRange: valor númérico en un rango
  - var-name: min, max
- maxLength: longitud máxima (nº caracteres)
  - var-name: maxLength
- minLength: longitud mínima (nº caracteres)
  - var-name: minLength
- byte,short,integer,long,double,float (¿se puede convertir a…? – sin parámetros)
- email,creditcard (sin parámetros)

Definir los mensajes de error

- Por defecto, se suponen en el .properties bajo la clave `errors.`*`nombre_del validador`*

```
errors.required = el campo está vacío
errors.minlength = el campo no tiene la longitud mínima
```

- Podemos cambiar la clave con la etiqueta **<msg>**

(validation.xml)

```
…
<form name="registro">
  <field property="nombre" depends="required,minlength">
  <msg name="required" key="nombre.noexiste"/>
…
```

(fichero .properties)

```
nombre.noexiste = el campo nombre está vacío
```

Parámetros en los mensajes

- Con las etiquetas <arg0> hasta <arg3> o <arg position="0">

(fichero *validation.xml*)

```
…
<form name="RegistroForm">
    <field property="login" depends="required,minlength">
            <arg0 name="required" key="nombre" resource="false"/>
            <var>
               <var-name>minlength</var-name>  <var-value>5</var-value>
            </var>
     </field>
</form>
…
```

(fichero *.properties*)

```
errors.required = El campo {0} está vacío
```

(RESULTADO)

El campo nombre está vacío

Parámetros en los mensajes (II)

- Por defecto (o resource="true") el propio parámetro es una clave en el .properties

(fichero *validation.xml*)

```
…
<form name="RegistroForm">
    <field property="login" depends="required,minlength">
            <arg0 name="required" key="nombre"/>
            <var> <var-name>minlength</var-name>  <var-value>5</var-value> </var>
     </field>
</form>
…
```

(fichero *.properties*)

```
errors.required = El campo {0} está vacío
nombre = nombre de usuario
```

(RESULTADO)

El campo nombre de usuario está vacío

Parámetros en los mensajes (III)

- Parámetros propios del validador

(fichero *validation.xml*)

```
…
<form name=“RegistroForm">
    <field property="login" depends="required,minlength">
              <arg0 name=“minlength” key="nombre”/>
              <arg1 name="minlength" key="${var:minlength}" resource=“false”/>
              <var> <var-name>minlength</var-name>  <var-value>5</var-value> </var>
     </field>
</form>
…
```

(fichero *.properties*)

```
errors.minlength = El campo {0} debe tener como mínimo {1} caracteres
nombre = nombre de usuario
```

(RESULTADO)

El campo nombre de usuario debe tener como mínimo 5 caracteres

Finalmente, hay que modificar el ActionForm

- Se debe heredar de ValidatorForm (o DynaValidatorForm si es dinámico)

```
import org.apache.struts.validator.ValidatorForm
public class RegistroForm extends ValidatorForm {
   private String login;
   ...
}
```

- Ya no hace falta implementar validate()
  - ¡¡¡Cuidado, si aprovechábamos el *validate* para convertir datos manualmente, tendremos que hacerlo de otra forma!!!

Validación en el cliente

- Validator genera automáticamente el JavaScript

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

Indice

- El plugin *validator*. Instalación
- Configuración de los validadores
- **Buenas prácticas**

Evitar envío de duplicados

- Cuando una operación implica cambios en la B.D., hay que evitar que el usuario pueda volver atrás en el navegador y repetir la misma operación
  - Hacer el mismo pedido dos veces, borrar dos veces el mismo dato, registrarse dos veces en el sitio,…
- **Solución:** cada operación llevará un identificador único, *"empotrado" en el formulario*. Si se repite, no ejecutar la operación

```
<form method="post" action="/nuevoUsuario.do">
  <input type="hidden" name="TOKEN“ value="d9f6dec7b65a6ab65a6a">
  …
</form>
```

  - Struts puede generar y comprobar el identificador por nosotros (con un poco de ayuda por nuestra parte)

Generar y comprobar el identificador

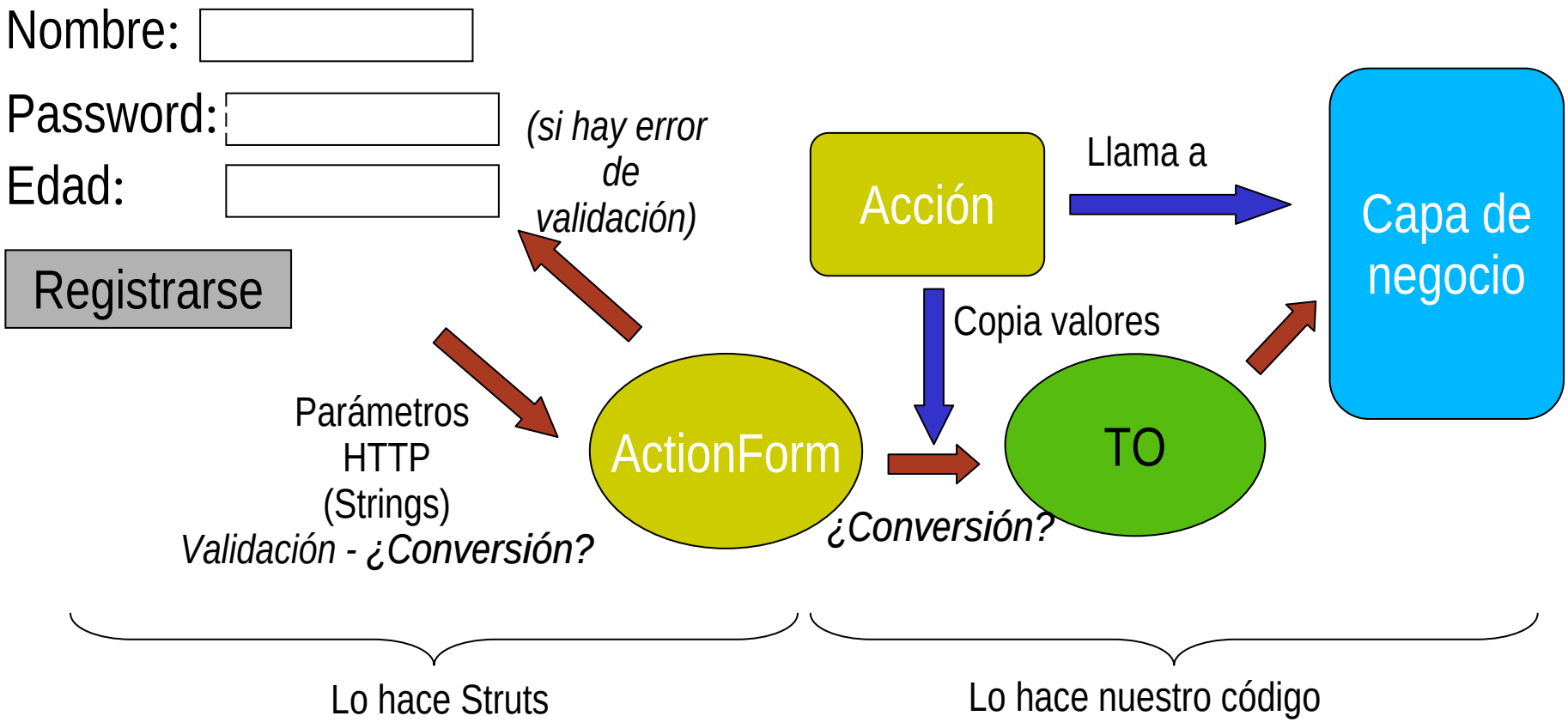
- **Condición:** el formulario debe usar la taglib HTML de Struts
- **Paso 1**: En la acción **que lleva al formulario**

```
saveToken(request);
```

- **Paso 2:** En la acción **llamada por el formulario** y que debe realizar la operación

```
if (isTokenValid(request)) {
        //OK, realizamos la operación
        …
        //Preparados para hacer una nueva operación
        resetToken(request);
}
else {
        //envío duplicado, saltar a una página de error…
}
```

Transferencia de datos entre capas



Transferencia de datos entre capas (II)

- **Validación** de datos: ya hemos visto que se puede automatizar totalmente con *validator*.
- **Conversión**: los parámetros HTTP son Strings, y en el TO no habrá solo Strings
  - **Opción 1**: en el ActionForm

    Aunque Struts solo puede convertir entre tipos primitivos y Strings, internamente usa una librería llamada BeanUtils, que permite crear conversores propios, luego en teoría es posible

    Aunque sea posible, en caso de fallar se perdería lo que el usuario haya escrito originalmente (sorpresa al mostrar de nuevo el formulario, como ya vimos)
  - **Opción 2**: en la acción, al rellenar el TO

Uso de BeanUtils para copiar/convertir propiedades

- Mediante *reflection*, copia las propiedades de igual nombre y convierte los tipos si es necesario

```
import org.apache.commons.beanutils.BeanUtils;

public class RegistroAccion extends Action {
   public ActionForward execute(ActionMapping mapping, ActionForm form,
                                                HttpServletRequest request,
                                                HttpServletResponse response) {
            RegistroForm rf = (RegistroForm) form;
            UsuarioTO uto = new UsuarioTO();
            BeanUtils.copyProperties(uto, rf);
            …
   }
}
```

Uso de BeanUtils para copiar/convertir propiedades (II)

- En caso de querer alguna conversión “especial” (por ejemplo, “10/10/2007” a Date), es necesario antes registrar un conversor “propio”

```
import org.apache.commons.beanutils.ConvertUtils;
import org.apache.commons.beanutils.converters.DateTimeConverter;

//DateTimeConverter es la clase por defecto para convertir fechas
DateTimeConverter dtc = new DateTimeConverter(Date.class);
//Le decimos que vamos a usar un formato especial
dtc.setPattern("dd/MM/yyyy");
//Registramos el conversor para que se use para convertir a Date
ConvertUtils.register(dtc, Date.class);
```

¿Dónde inicializar el conversor?

- Podemos crear un plugin: clase que implementa el interfaz org.apache.struts.action.PlugIn

```
(faltan los import…)
public class InitConverterPlugin implements PlugIn {
        private String pattern;
        public void destroy() {
        }
        public void init(ActionServlet as, ModuleConfig mc) throws ServletException {
                DateTimeConverter dtc = new DateTimeConverter(Date.class);
                dtc.setPattern(pattern);
                ConvertUtils.register(dtc, Date.class);
        }
        public void setPattern(String pattern) {
                this.pattern = pattern;
        }
}
```

Configurar el plugin

- En el struts-config.xml. Ya hemos visto cómo se hace en *validator*, en nuestro plugin es igual
- La etiqueta <set-property> llama al *setter* del plugin

```
<plug-in className="es.ua.jtech.strutsplugins.InitConverterPlugin">
   <set-property property="pattern" value="dd/MM/yyyy"/>
</plug-in>
```

¿Preguntas...?