Struts

Sesión 3: Validación automática. Internacionalización

Indice

- **Validación automática con el plugin validator**
- Internacionalización
- Evitación de envíos duplicados

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**

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()

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

- Validación automática con el plugin validator
- **Internacionalización**
- Evitación de envíos duplicados

Internacionalización (i18n)

- El proceso de diseño y desarrollo que lleva a que una aplicación pueda ser adaptada fácilmente a diversos idiomas y regiones sin necesidad de cambios en el código
- El factor más costoso es la traducción de textos, aunque no es lo único que varía
  - Formatos de fechas, números, moneda, etc.
- **Localización**: adaptación de una aplicación a un idioma/región concreto (I10n)

El soporte de i18n de Java

- La plataforma Java ofrece soporte nativo a la *i18n*,en el que se basa Struts, así que conviene conocerlo antes de ver cosas propias de Struts.
- Tres clases básicas:
  - **Locale**: combinación de idioma y país
  - **ResourceBundle**: permite almacenar textos fuera del código fuente,para no tener que recompilar si cambia algo (p.ej. idioma)
  - **MessageFormat**: permite hacer mensajes con parámetros a los que se da valor en tiempo de ejecución

Locale

- Combinación de **idioma**, y opcionalmente **país** y dialecto/región (este último no se suele usar). Tanto el idioma como el país se especifican con códigos ISO

```
Locale pibeLocale = new Locale("es","AR");
```

- Algunos métodos de Java son “sensibles” al *locale*, aceptándolo como parámetro

```
NumberFormat nf = NumberFormat.getCurrencyInstance(new Locale("es","ES"));
//Esto imprimirá "100,00 €"
System.out.println(nf.format(100));
```

ResourceBundle

- Almacena textos **aparte** del código fuente. En **Struts** se usan ficheros .properties
  - Recordemos en struts-config.xml cómo se decía dónde estaban los mensajes

```
//El fichero será mensajes.properties, estando en cualquier sitio del CLASSPATH
<message-resources parameter="mensajes"/>
```

  - Para **i18n**, basta con tener varios .properties que tengan al final el idioma y el país. Se usará automáticamente el del *locale* actual (luego veremos cómo cambiar el *locale* actual)

```
mensajes_es_ES.properties
mensajes_es_AR.properties
mensajes_en.properties        //No es necesario especificar país
mensajes.properties           //Por defecto, si el sistema no encuentra el apropiado
```

MessageFormat

- Para generar mensajes con parámetros. Ya lo hemos usado en *validator*

```
Se ha producido un error con el campo {0} del formulario
```

- Aunque se le puede dar valor a los parámetros con un API estándar de Java, lo habitual es que los "rellene" Struts por nosotros
- También se pueden formatear números, fechas y horas (*ver el API de Java SE*)

```
Se ha realizado un cargo de {0,number,currency} a su cuenta con fecha {1,date,long}
```

¿Qué aporta Struts a todo esto?

- Nos permite obtener y cambiar el locale actual en las acciones
- Nos automatiza la recuperación de mensajes del fichero .properties adecuado
- Las *taglibs* son "sensibles" al locale
  - Si las usamos adecuadamente, la aplicación estará automáticamente internacionalizada

Obtener y cambiar el locale actual en una acción

- Struts guarda el locale actual en la sesión HTTP

```
Locale locale = request.getSession().getAttribute(Globals.LOCALE_KEY);
```

  - Por defecto, el locale actual es el del servidor
  - Si queremos saber el del navegador del usuario, podemos hacer

```
request.getLocale();
```

- Para cambiar el locale se crea uno nuevo (ya que son objetos inmutables) y se guarda en la sesión

```
Locale nuevo = new Locale("es","ES");
request.getSession().setAttribute(Globals.LOCALE_KEY,nuevo);
```

Recuperar mensajes localizados

- A través de la clase MessageResources
- Es automático, aunque también se puede hacer a través de un API

```
Locale locale = request.getSession().getAttribute(Globals.LOCALE_KEY);
MessageResources mens = getResources(request);
String m = mens.getMessage(locale, "error");
```

Componentes de Struts "sensibles" al locale

- Los mensajes gestionados con ActionError y ActionMessage
- Muchas etiquetas de las *taglibs*
  - Sustituir todos los textos de los JSP por *tags* de Struts. La típica usada para esto es <bean:message>

```
<%@ taglib uri="http://struts.apache.org/tags-bean-el" prefix="bean" %>
…
<bean:message key="saludo“/>
<bean:message key="saludo“ arg0="${usuarioActual.login}" />
```

Localización de *validator*

- Los mensajes de error de los validadores estarán localizados automáticamente
- Podemos hacer validadores dependientes del *locale*

```
<form name="registro" locale="es" country="ES">
 <field property="codigoPostal" depends="required,mask">
   <var> <var-name>mask</var-name> <var-value>^[0-9]{5}$</var-value> </var>
 </field>
</form>
<!-- en Argentina, los C.P. tienen 1 letra seguida de 4 dígitos y luego 3 letras más -->
<form name="registro" locale="es" country="AR">
 <field property="codigoPostal" depends="required,mask">
     <var>
       <var-name>mask</var-name> <var-value>^[A-Z][0-9]{4}[A-Z]{3}$</var-value>
     </var>
</field>
</form>
```

Indice

- Validación automática con el plugin validator
- Internacionalización
- **Evitación de envíos duplicados**

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…
}
```

¿Preguntas...?