Validación. Internacionalización

Índice

1. Validación

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 con la distribución**

de Struts. 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="nombre" depends="required,minlength">
         <var>
            <var-name>minlength</var-name>
            <var-value>6</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`). 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.

Resumiendo, en el ejemplo anterior estamos diciendo que el campo "nombre" debe contener algún valor no vacío (`required`) y que su longitud mínima (`minLength`) debe ser de 6 caracteres

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. Se usan las librerías de Jakarta RegExp.

`intRange`, `longRange`, `floatI`	valor dentro de un rango	`min`, `max`. Estos validadores dependen respectivamente de `int`,`long`,... por tanto en el `depends` de por ejemplo un `intRange` debe aparecer también `int`.
`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
`url`	verificar URL	Varios. Consultar documentación de Struts

Por ejemplo, supongamos que tenemos un campo en el que el usuario debe escribir un porcentaje, permitiendo decimales. Podríamos usar el siguiente validador:

```
<field property="porcentaje" depends="required,double,doubleRange">
<var><var-name>min</var-name><var-value>100</var-value></var>
        <var><var-name>max</var-name><var-value>0</var-value></var>
</field>
```

Obsérvese que `doubleRange` usa también `double` y por eso lo hemos tenido que incluir en el `depends`.

1.4. Mensajes de error

1.4.1. ¿Dónde están definidos realmente los mensajes de error?

Lo normal será mostrar algún mensaje de error si algún validador detecta que los datos no son correctos. Para ello podemos usar las etiquetas de Struts `<html:messages>` o `<html:errors>` que vimos en sesiones anteriores. Recordemos que dichas etiquetas asumen que el error está en un fichero .properties almacenado bajo una determinada clave. Validator supone automáticamente que la clave asociada a cada validador es "errors.*nombre_del_validador*". Por tanto, para personalizar los mensajes del ejemplo anterior, deberemos incluir en el fichero .properties algo como:

```
errors.required=campo vacío
errors.minLength=no se ha cumplido la longitud mínima
```

Por razones históricas,el mensaje de error asociado al validador `mask` no se busca bajo la clave `errors.mask`, sino `errors.invalid`. Nótese que si no definimos estas claves e intentamos mostrar el error, Struts nos dirá que no se encuentra el mensaje de error (es decir, Struts no define mensajes de error por defecto). No obstante, si no nos gusta el nombre de la clave por defecto, podemos cambiarlo mediante la etiqueta `<msg>`

```
<form name="registro">
  <field property="nombre" depends="required,minlength">
  <msg name="required" key="campo.nombre.noexiste"/>
     <var>
         <var-name>minlength</var-name>
         <var-value>6</var-value>
     </var>
  </field>
  ...
</form>
```

Y en el fichero `.properties`, tendríamos:

```
campo.nombre.noexiste=el nombre no puede estar vacío
```

Con lo cual personalizamos el mensaje, pudiendo poner mensajes distintos en caso de que el campo vacío sea otro. En el siguiente apartado veremos una forma mejor de personalizar los mensajes: pasarles parámetros.

1.4.2. Mensajes de error con parámetros

Es mucho más intuitivo para el usuario personalizar el mensaje de error, adaptándolo al error concreto que se ha producido. En el ejemplo anterior, es mucho mejor un mensaje "el campo nombre está vacío" que simplemente "campo vacío". Podemos usar las etiquetas `<arg0>...<arg4>` para pasar parámetros a los mensajes de error.

```
<form name="registro">
  <field property="nombre" depends="required,minlength">
  <arg0 name="required" key="nombre" resource="false"/>
     <var>
```

```
            <var-name>minlength</var-name>
            <var-value>6</var-value>
        </var>
    </field>
    ...
</form>
```

El atributo `name` indica que este argumento solo se usará para el mensaje de error del validador `required`. Si el atributo no está presente, el argumento se usa para todos (Es decir, en este caso, también para `minlength`). En principio, el atributo `key` es el valor del argumento, aunque ahora lo discutiremos con más profundidad, junto con el significado del atributo `resource`.

Ahora en el fichero .properties debemos reservar un lugar para colocar el argumento 0. Como ya vimos en la primera sesión, esto se hace poniendo el número del argumento entre llaves:

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

Por tanto, el mensaje final que se mostrará a través de `<html:messages>` o `<html:errors>` será "campo nombre vacío". En el ejemplo anterior, el parámetro `resource="false"` de la etiqueta `<arg0>` se usa para indicarle a validator que el valor de `key` debe tomarse como un literal. En caso de ponerlo a `false` (y también por defecto) el valor de `key` se toma como una clave en el fichero `.properties`, es decir, que `nombre` no se tomaría literalmente sino que en el `.properties` debería aparecer algo como:

```
nombre=nombre
```

Donde la parte de la izquierda es la clave y la de la derecha el valor literal. Nótese que en este ejemplo ambos son el mismo valor pero nada nos impide por ejemplo poner `nombre=nombre de usuario` o cualquier otra cosa. A primera vista, este nivel de indirección en los argumentos de los mensajes puede parecer absurdo, pero nótese que si se tradujera la aplicación a otro idioma solo necesitaríamos un nuevo `.properties`, mientras que de la otra forma también habría que modificar el `validation.xml`. Veremos más sobre internacionalización de aplicaciones en la siguiente sesión.

Con algunos validadores es útil mostrarle al usuario los parámetros del propio validador, por ejemplo sería útil indicarle que el nombre debe tener como mínimo 6 caracteres (sin tener que poner literalmente el "6" como parte del mensaje, por supuesto). Esto se puede hacer usando el lenguaje de expresiones (EL) en el parámetro `key` del argumento:

```
<form name="registro">
  <field property="nombre" depends="required,minlength">
  <arg0 key="nombre" resource="false"/>
  <arg1 name="minlength" key="${var:minlength}" resource="false"/>
     <var>
         <var-name>minlength</var-name>
         <var-value>6</var-value>
```

```
    </var>
  </field>
  ...
</form>
```

Y modificamos el `.properties` para que quede:

```
errors.required=campo {0} requerido
errors.minlength={0} debe tener una longitud mínima de {1}
caracteres
```

1.5. 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.

1.6. 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. Internacionalización

En una red sin fronteras, desarrollar una aplicación web multilenguaje es una necesidad, no una opción. Reconociendo esta necesidad, la plataforma Java proporciona una serie de facilidades para desarrollar aplicaciones que "hablen" el idioma propio del usuario. Struts se basa en ellas y añade algunas propias con el objeto de hacer lo más sencillo posible el proceso de desarrollo.

Nota:

La **Internacionalización** es 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. Algunas veces el nombre se abrevia a *i18n* porque en la palabra hay 18 letras entre la primera "i" y la última "n".La **Localización** es el proceso de adaptar un software a un idioma y región concretos. Este proceso se abrevia a veces como *l10n* (por motivos que ahora mismo deberían ser obvios...).

Aunque el cambio del idioma es la parte generalmente más costosa a la hora de localizar una aplicación, no es el único factor. Países o regiones diferentes tienen, además de idiomas diferentes, distintas monedas, formas de especificar fechas o de escribir números con decimales.

2.1. El soporte de internacionalización de Java

Como ya hemos comentado, la propia plataforma Java ofrece soporte a la internacionalización, sobre el que se basa el que ofrece Struts. Este soporte se fundamenta en tres clases básicas: `java.util.Locale`, `java.util.ResourceBundle` y `java.text.MessageFormat`

2.1.1. Locale

Es el núcleo fundamental de todo el soporte de internacionalización de Java. Un *locale* es una combinación de idioma y país (y opcionalmente, aunque muchas veces no se usa, una variante o dialecto). Tanto el país como el idioma se especifican con códigos ISO (estándares ISO-3166 e ISO-639). Por ejemplo, el siguiente código crearía un *locale* para español de Argentina

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

Algunos métodos de la librería estándar son "sensibles" al idioma y aceptan un *locale* como parámetro para formatear algún dato, por ejemplo

```
NumberFormat nf = java.text.NumberFormat.getCurrencyInstance(new
Locale("es","ES"));
```

```
//Esto imprimirá "100,00 €"
System.out.println(nf.format(100));
```

De una forma similar, Struts tiene también clases y etiquetas "sensibles" al *locale* actual de la aplicación. Posteriormente veremos cómo hacer en Struts para cambiar el *locale*.

2.1.2. ResourceBundle

Esta clase sirve para almacenar de manera independiente del código los mensajes de texto que necesitaremos traducir para poder internacionalizar la aplicación. De este modo no se necesita recompilar el código fuente para cambiar o adaptar el texto de los mensajes.

Un *resource bundle* es una colección de objetos de la clase `Properties`. Cada `Properties` está asociado a un determinado `Locale`, y almacena los textos correspondientes a un idioma y región concretos.

`ResourceBundle` es una clase abstracta con dos implementaciones en la librería estándar, `ListResourceBundle` y `PropertyResourceBundle`. Esta última es la que se usa en Struts, y almacena los mensajes en ficheros de texto del tipo `.properties`. Este fichero es de texto plano y puede crearse con cualquier editor. Por convención, el nombre del *locale* asociado a cada fichero se coloca al final del nombre del fichero, antes de la extensión. Recordemos de la sesión 1 que en Struts se emplea una etiqueta en el fichero de configuración para indicar dónde están los mensajes de la aplicación:

```
<message-resources parameter="mensajes"/>
```

Si internacionalizamos la aplicación, los mensajes correspondientes a los distintos idiomas y regiones se podrían almacenar en ficheros como los siguientes:

```
mensajes_es_ES.properties
mensajes_es_AR.properties
mensajes_en_UK.properties
```

Así, el fichero `mensajes_es_ES.properties`, con los mensajes en idioma español (es) para España (ES), podría contener algo como lo siguiente:

```
saludo = Hola
error= lo sentimos, se ha producido un error
```

No es necesario especificar tanto el idioma como el país, se puede especificar solo el idioma (por ejemplo `mensajes_es.properties`). Si se tiene esto y el locale actual es del mismo idioma y otro país, el sistema tomará el fichero con el idioma apropiado. Si no existe ni siquiera fichero con el idioma apropiado,entonces acudirá al fichero por defecto, que en nuestro caso sería `mensajes.properties`.

2.1.3. MessageFormat

Los mensajes totalmente genéricos no son muy ilustrativos para el usuario: por ejemplo el mensaje "ha habido un error en el formulario" es mucho menos intuitivo que "ha habido un error con el campo login del formulario". Por ello, Java ofrece soporte para crear mensajes con parámetros, a través de la clase `MessageFormat`. En estos mensajes los parámetros se indican con números entre llaves. Así, un mensaje como

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

Se puede rellenar en tiempo de ejecución, sustituyendo el **{0}** por el valor deseado. Aunque la asignación de los valores a los parámetros se puede hacer con el API estándar de Java, lo habitual en una aplicación de Struts es que el framework lo haga por nosotros (recordemos que es lo que ocurría con los mensajes de error gestionados a través de la clase`ActionMessage`).

Además de Strings, se pueden formatear fechas, horas y números. Para ello hay que especificar, separado por comas, el tipo de parámetro: fecha (`date`), hora (`time`) o número (`number`) y luego el estilo (`short`, `medium`, `long`, `full` para fechas e `integer`,`currency` o `percent` para números). Por ejemplo:

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

2.2. El soporte de internacionalización de Struts

Vamos a ver en los siguientes apartados cómo internacionalizar una aplicación Struts. El *framework* pone en marcha por defecto el soporte de internacionalización, aunque podemos desactivarlo pasándole al servlet controlador el parámetro `locale` con valor `false` (se pasaría con la etiqueta `<init-param>` en el `web.xml`).

2.2.1. Cambiar el locale actual

Struts almacena el *locale* actual en la sesión HTTP como un atributo cuyo nombre es el valor de la constante `Globals.LOCALE_KEY`. Así, para obtener el *locale* en una acción podríamos hacer:

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

Al locale inicialmente se le da el valor del que tiene por defecto del servidor. El locale del usuario que está "al otro lado" navegando se puede obtener con la llamada al método `getLocale()` del objeto `HttpServletRequest`. La información sobre el locale del usuario se obtiene a través de las cabeceras HTTP que envía su navegador.

Nota:

> Los navegadores generalmente ofrecen la posibilidad al usuario de cambiar el idioma preferente para visualizar las páginas. Esto lo que hace es cambiar la cabecera HTTP `Accept-Language` que envía el navegador, y como hemos visto, puede utilizarse desde nuestra aplicación de Struts para darle un valor al locale.

Los *locales* son objetos inmutables, por lo que para cambiar el actual por otro, hay que crear uno nuevo:

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

A partir de este momento, los componentes de Struts "sensibles" al locale mostrarán la información teniendo en cuenta el nuevo locale.

2.2.2. MessageResources: el ResourceBundle de Struts

En Struts, la clase `MessageResources` es la que sirve para gestionar los ficheros con los mensajes localizados. Se basa en la clase estándar `PropertyResourceBundle`, por lo que los mensajes se almacenan en ficheros `.properties`. Recordemos de la sesión 1 que para indicar cuál es el fichero con los mensajes de la aplicación se usa la etiqueta `<message-resources>` en el fichero `struts-config.xml`. Ahora ya sabemos que podemos tener varios ficheros .properties, cada uno para un *locale* distinto.

Aunque lo más habitual es mostrar los mensajes localizados a través de las etiquetas de las *taglibs* de Struts, también podemos acceder a los mensajes directamente con el API de `MessageResources`. Por ejemplo, en una acción podemos hacer lo siguiente:

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

2.2.3. Componentes de Struts "sensibles" al locale

Ya vimos en la primera sesión cómo se gestionaban los errores en las acciones a través de las clases `ActionMessage` y `ActionErrors`. Estas clases están internacionalizadas, de modo que si tenemos adecuadamente definidos los `.properties`, los mensajes de error estarán localizados sin necesidad de esfuerzo adicional por nuestra parte.

Varias etiquetas de las *taglibs* de Struts están internacionalizadas. La más típica es `<bean:message>`, que se emplea para imprimir mensajes localizados. La mayor parte de etiquetas para mostrar campos de formulario también están internacionalizadas (por ejemplo `<html:option>`).

Por ejemplo, para escribir en un JSP un mensaje localizado podemos hacer:

```
<bean:message key="saludo">
```

Donde el parámetro `key` es la clave que tiene el mensaje en el `.properties` del *locale* actual. Si el mensaje tiene parámetros, se les puede dar valor con los atributos `arg0` a `arg4`, por ejemplo:

```
<bean:message key="saludo" arg0="${usuarioActual.login}">
```

Aunque no es muy habitual, se puede también especificar el *locale* en la propia etiqueta, mediante el parámetro del mismo nombre o el *resource bundle* a usar, mediante el parámetro `bundle`. En estos dos últimos casos, los valores de los parámetros son los nombres de beans de sesión que contienen el *locale* o el *resource bundle*, respectivamente.

2.2.4. Localización de "validator"

Los mensajes que muestra el *plugin* validator utilizan el `MessageResources`, por lo que ya aparecerán localizados automáticamente. No obstante, puede haber algunos elementos cuyo formato deba cambiar con el *locale*, como podría ser un número de teléfono, un código postal, etc. Por ello, en validator se puede asociar una validación con un determinado *locale*. Recordemos que en validator se usaba la etiqueta `<form>` para definir la validación a realizar sobre un ActionForm.

```
<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>
```

Como vemos, los parámetros `locale` y `country` de la etiqueta nos permiten especificar el idioma y el país, respectivamente. No son los dos obligatorios, podemos especificar solo el `locale`. Además, estos atributos también los admite la etiqueta `<formset>`, de modo que podríamos crear un conjunto de `forms` distinto para cada locale.