Internacionalización. Pruebas

Índice

1. 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.

1.1. 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`

1.1.1. 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*.

1.1.2. 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`.

1.1.3. 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}
```

1.2. 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`).

1.2.1. 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.

1.2.2. 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");
```

1.2.3. 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.

1.2.4. 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.

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