Struts

Índice

1. Introducción a Struts: El controlador y las acciones

Struts es un *framework* para construir aplicaciones web Java basadas en la filosofía MVC. Veremos en este módulo en qué consiste la arquitectura MVC y cómo la implementa Struts.

1.1. Introducción a Struts

Antes de entrar en detalles sobre el funcionamiento de Struts vamos a ver primero la filosofía general de funcionamiento. Veremos también por qué usar Struts y las posibles alternativas.

Hay dos "generaciones" de Struts: la 1.x y la 2. El cambio de la generación 1 a la 2 no significa únicamente que se hayan añadido nuevas características. Es un cambio completo, tanto a nivel interno como externo. Cuando surgió Struts 1 era el único *framework* existente de este tipo y en su diseño original había deficiencias en cuanto a flexibilidad y simplicidad de uso. Todo esto se ha solucionado en la versión 2. No obstante, y de manera paradójica, Struts 2, aun siendo mucho más potente, flexible y fácil de usar, no ha tenido ni de lejos tanta difusión como la versión 1, simplemente porque ahora ya tiene la competencia de otros frameworks como Spring o JSF.

Struts 1.x es por tanto, un *framework* básico en JavaEE si se considera la gran cantidad de aplicaciones en producción que lo usan y que habrá que mantener todavía durante unos años. Por eso vamos a ver sobre todo esta versión. No obstante, para nuevos proyectos la versión recomendada es la 2.

1.1.1. Por qué usar Struts. Alternativas

Antes de hablar de las supuestas bondades de Struts, conviene detenerse un momento a recalcar la diferencia entre *framework* y *librería*. Struts es un *framework*, lo cual significa que no solo nos proporciona un API con el que trabajar (esto ya lo haría una librería) sino también una filosofía de desarrollo, una "forma de hacer las cosas". Por tanto, el primer beneficio de usar un framework es que estamos haciendo las cosas de una forma ya probada, la misma idea que constituye la base de los patrones de diseño software.

Por supuesto Struts no es el único framework MVC existente en el mundo J2EE.Aunque existen muchos otros, Struts es el más extendido con mucha diferencia, hasta el punto de haberse convertirdo en un estándar "de facto" en el mundo J2EE. Por tanto, usando Struts estamos seguros de que dispondremos de una gran cantidad de recursos: documentación (tutoriales, artículos, libros,...) interacción con otros usuarios del framework a través de foros y similares y una amplia base de desarrolladores expertos a los que podremos acudir si necesitamos personal para un proyecto.

Hay varios frameworks "alternativos" a Struts. Spring, que veremos en uno de los

módulos siguientes, incorpora también su parte MVC. Webwork, que hasta hace poco era un framework competidor de Struts se ha unificado con éste dando lugar a Struts 2. Esta breve discusión sobre otros frameworks "alternativos" no quedaría completa sin nombrar a JavaServer Faces (JSF), que también se aborda en el curso. JSF se solapa en algunos aspectos con Struts, ya que también implementa MVC (aunque de modo distinto). No obstante, la aportación principal de JSF no es MVC sino los componentes gráficos de usuario (GUI) de "alto nivel" para la web.

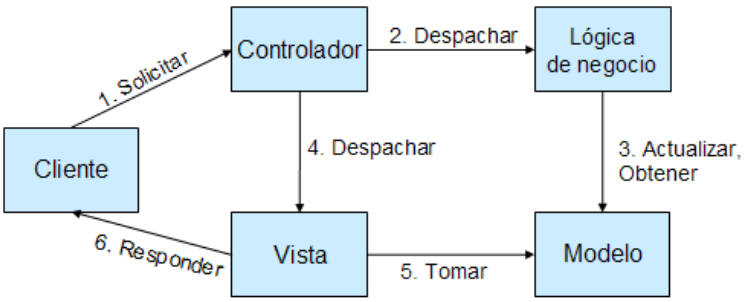
1.1.2. MVC y Struts

Veamos cómo implementa Struts los componentes del patrón Modelo-Vista-Controlador:

- El **controlador** es un servlet, de una clase proporcionada por Struts. Será necesario configurar la aplicación web (a través del fichero `web.xml`) para que todas las peticiones del usuario se redirijan a este servlet.
- El controlador despacha las peticiones del usuario a la clase adecuada para ejecutar la **acción**. En struts, las clases que ejecuten las acciones deben *heredar de la clase **Action***.
- La **vista** se implementará normalmente mediante páginas JSP. Struts ofrece dos herramientas para ayudar en la presentación de datos: los **ActionForm**s son clases que capturan los datos introducidos en formularios y permiten su validación. Las **librerías de etiquetas** permiten mostrar errores y facilitar el trabajo con formularios.
- La implementación del **modelo** corre enteramente a cargo del desarrollador, ya que es propio de la capa de negocio y no está dentro del ámbito de Struts.

En esta primera sesión trataremos la parte del controlador y las acciones. En la sesión 2 trataremos de la vista, describiendo algunas de las librerías de etiquetas del framework.

El ciclo que se sigue cuando Struts procesa una petición HTTP aparece en la siguiente figura



El ciclo de proceso de MVC en Struts

1. El cliente realiza la **petición**, que recibe el controlador de Struts. Todas las peticiones pasan por él, ya que la petición no es una URL física (no es un servlet o un JSP) sino que es un nombre simbólico para una **acción**.
2. El controlador **despacha la petición**, identificando la acción y disparando la lógica de

negocio apropiada.
3. La lógica de negocio **actualiza el modelo** y obtiene datos del mismo, almacenándolos en beans.
4. En función del valor devuelto por la lógica de negocio, el controlador elige la **siguiente vista** a mostrar.
5. **La vista toma los datos** obtenidos por la lógica de negocio.
6. **La vista muestra los datos** en el cliente

1.2. El controlador

La instalación de Struts es sencilla. Basta con colocar en la carpeta `WEB-INF/lib` de nuestra aplicación web las librerías (ficheros .jar) que vienen con la distribución estándar de Struts.

La instanciación del servlet que hará de controlador, como en todos los servlets, hay que hacerla en el web.xml de la aplicación. Salvo que tengamos necesidades muy especiales, podemos usar directamente la clase `org.apache.struts.action.ActionServlet`, que ya viene implementada en la distribución de Struts. Al servlet se le puede pasar opcionalmente como parámetro (en la forma estándar, con `<init-param>`) el nombre que va a tener el fichero de configuración de Struts, que normalmente es `struts-config.xml` dentro de la carpeta `WEB-INF`. Por último, para que todas las peticiones del usuario se redirijan al servlet habrá que mapear el servlet en la forma habitual. Veamos un ejemplo de web.xml:

```
<!-- Definir el servlet que hace de controlador -->
 <servlet>
    <servlet-name>controlador</servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <!-- Podemos cambiar la carpeta por defecto del fich. de
configuracion
            (aunque no es algo que se suela hacer a menudo) -->
    <init-param>
       <param-name>config</param-name>
       <param-value>/WEB-INF/config/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
 </servlet>
 <!-- redirigir ciertas peticiones al controlador -->
 <servlet-mapping>
    <servlet-name>controlador</servlet-name>
    <url-pattern>*.do</url-pattern>
 </servlet-mapping>
```

En el ejemplo anterior, todas las peticiones que sigan el patrón `*.do` se redirigirán al controlador de Struts. Por ejemplo la petición `login.do` será capturada por Struts y redirigida a la acción de nombre `login`.

1.3. Las acciones

En Struts, las acciones son clases Java, uno de cuyos métodos se ejecutará en respuesta a una petición HTTP del cliente. Para que todo funcione adecuadamente hay que asociar las peticiones con las acciones que dispararán e implementar la lógica de negocio dentro de las acciones.

1.3.1. Código java

Las clases encargadas de ejecutar las acciones deben descender de la clase abstracta `org.apache.struts.action.Action`, proporcionada por Struts. Cuando se ejecuta una acción lo que hace Struts es llamar a su método `execute`, que debemos sobreescribir para que realice la tarea deseada. Por ejemplo, para el caso de una hipotética acción de login en una aplicación web:

```
package acciones;
import javax.servlet.http.*;
import org.apache.struts.action.*;
public class AccionLogin extends Action
   public ActionForward execute(ActionMapping mapping, ActionForm
form,
                        HttpServletRequest request,
HttpServletResponse response)
                        throws Exception {
        boolean usuarioOK;
        //obtener login y password y autentificar al usuario
        //si es correcto, poner usuarioOK a 'true'
        ...
        //dirigirnos a la vista adecuada según el resultado
        if (usuarioOK)
           return mapping.findForward("OK");
        else
           return mapping.findForward("errorUsuario");
   }
}
```

Hay que destacar varias cosas del código de la acción:

- Como se ha dicho, una acción debe heredar de la clase `org.apache.struts.action.Action`.
- El método `execute` recibe como **parámetros** la petición y la respuesta HTTP, lo que nos permite interactuar con ellas. No obstante, también tenemos accesibles los datos incluidos en la petición (normalmente a través de formularios) mediante el objeto `ActionForm`, si es que hemos asociado un objeto de esta clase a la acción en el `struts-config.xml`. El uso de `ActionForm` lo trataremos en el tema 2.
- El método execute debe **devolver** un objeto de la clase `ActionForward`, que especifica la siguiente vista a mostrar.
- Una acción puede tener varios resultados distintos, por ejemplo en nuestro caso el login puede ser exitoso o no. Lo lógico en cada caso es mostrar una vista distinta. Para evitar el acoplamiento en el código Java entre la vista a mostrar y el resultado de la acción, Struts nos ofrece el objeto `ActionMapping` que se pasa como parámetro del

método `execute`. Dicho objeto contiene un mapeo entre nombres simbólicos de resultados para una acción y vistas a mostrar. La llamada al método `findForward` de dicho objeto nos devuelve un `ActionForward` que representa la vista asociada al resultado de la acción.

1.3.2. Flujo de navegación

Una vez el controlador recibe la petición debe despacharla a las clases Java que implementan la lógica de negocio.La asociación entre el nombre simbólico de la acción y la clase Java que la procesa se realiza en el fichero de configuración `struts-config.xml`, que se coloca en el directorio `WEB-INF` de la aplicación. A lo largo de estos dos temas iremos viendo los distintos elementos de este fichero de configuración.

Todos los mapeados entre peticiones y acciones se colocan dentro de la etiqueta XML `<action-mappings>` del fichero `struts-config.xml`. Cada mapeado concreto es un `<action>`, cuyo atributo `path` es la petición (sin el .do, que en nuestro ejemplo es lo que llevan todas las peticiones que van para Struts) y `type` el nombre de la clase que implementa la acción. Por ejemplo, recordemos nuestra clase Java `acciones.AccionLogin` y supongamos que es la encargada de procesar la petición a la URL `login.do`. El fichero `struts-config.xml` quedaría:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
<!--definición de otros elementos del fichero de configuración -->
   ...
   ...
<!--definición de acciones -->
<action-mappings>
   <!-- hacer login -->
   <action path="/login" type="acciones.AccionLogin">
       <forward name="OK" path="/personal.jsp"/>
       <forward name="errorUsuario" path="/error.html"/>
   </action>
   <!-- definición de otras acciones -->
   ...
</action-mappings>
</struts-config>
```

Como se ve en el ejemplo, dentro de cada `<action>` hay uno o más `<forward>`, que son los posibles resultados de la acción, y que asocian un nombre simbólico con una vista, en este caso un JSP, que es lo más común. Revisad el código del método `execute` del apartado anterior para comprobar que los nombres simbólicos se corresponden con los que aparecen en el código Java.

¡Cuidado con los errores en el struts-config.xml!

El fichero struts-config.xml es crítico, y cualquier mínimo error en el XML va a hacer que Struts no llegue a arrancar y por tanto nuestra aplicación no funcione correctamente. Hay que revisar el fichero con sumo cuidado. Por desgracia, la cantidad de información que hay que colocar en el fichero es considerable y por tanto también lo es la posibilidad de cometer errores. Este es uno de los problemas de Struts 1.x, que se soluciona en la versión 2.

Los forwards así definidos son forwards "locales" a la acción. Es habitual que varias acciones acaben redirigiendo a la misma página (por ejemplo el menú principal, o una página de error). Por eso, en la mayoría de aplicaciones es útil también tener forwards "globales". Estos se definen en el fichero de configuración, antes de los `action-mappings`, de la siguiente forma:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
1.1//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

<global-forwards>
        <forward name="errorUsuario" path="/error.html"/>
</global-forwards>

<!--definición de acciones -->
<action-mappings>
        ...
</action-mappings>
</struts-config>
```

¡Cuidado!

Cualquier modificación del fichero struts-config.xml requerirá la recarga de la aplicación en el servidor, ya que Struts no detecta automáticamente los cambios en este fichero.

1.3.3. Tratamiento de errores

La ejecución de la acción puede generar uno o varios errores que deseamos mostrar al usuario. En Struts, el tratamiento de errores requiere seguir una serie de pasos:

- Crear una lista de errores vacía. Esta lista se modela con el objeto `ActionMessages`.
- Añadir errores a la lista. Cada error es un objeto `ActionMessage`.
- Finalmente, si la lista contiene algún error
  - Guardar la lista en la petición HTTP para que no se pierda. Para ello se utiliza el método `saveErrors`.
  - La acción debe devolver un `forward` que indique que se ha producido un error.

Por ejemplo, el siguiente código, que habría que colocar dentro del método `execute` de la acción, realiza los pasos descritos:

```
//crear una lista de errores vacía
ActionMessages errores = new ActionMessages();
try {
   //código que ejecuta la lógica de negocio.
   ...
}
catch(Exception e) {
   //añadir errores a la lista
   errores.add(ActionMessages.GLOBAL_MESSAGE,
               new ActionMessage("error.bd");
}
//comprobar si la lista de errores está vacía
if (!errores.empty()) {
   //guardar los errores en la petición HTTP
   saveErrors(request, errores);
   //devolver un resultado que indique error. En struts-config.xml
   //estará definida la página jsp asociada a este resultado
   return mapping.findForward("error");
}
```

El constructor de un `ActionMessage` requiere como mínimo un argumento: el mensaje de error. Los mensajes de error no son directamente cadenas, sino claves dentro de un fichero de texto del tipo `properties`. Por ejemplo, la clave "`error.bd`" significará que debe haber un fichero `.properties` en el que se especifique algo como:

```
error.bd = se ha producido un error con la base de datos
```

Para indicar a struts cómo encontrar el fichero `.properties`, utilizamos el fichero de configuración `struts-config.xml`, mediante la etiqueta `<message-resources>` (que se pone detrás de la etiqueta <action-mappings>). Por ejemplo:

```
 <message-resources parameter="util.recursos"/>
```

Indicaría a struts que busque un fichero **recursos.properties** dentro de la carpeta `util`. Normalmente, se toma como base de la búsqueda la carpeta `/WEB_INF/classes`, por lo que el fichero buscado será finalmente`/WEB-INF/classes/util/recursos.properties`.

Las claves de error pueden utilizar hasta 4 parámetros. Por ejemplo supongamos que se desea mostrar un error indicando que hay un campo requerido para el que el usuario no ha introducido valor. Sería demasiado tedioso hacer un mensaje distinto para cada campo: ("login requerido", "password requerido", etc). Es más fácil definir un mensaje con parámetros:

```
error.requerido = Es necesario especificar un valor para el campo
{0}
```

Como se ve, los parámetros son simplemente números entre `{}`. El constructor de la clase

`ActionMessage` permite especificar los parámetros (como se ha dicho, hasta 4) además de la clave de error

```
ActionMessage error = new ActionMessage("error.requerido",
"login");
```

Vamos a ver con detalle cómo funciona el código anterior. Como ya se ha dicho, para añadir los errores a la lista se emplea el método `add` de la clase `ActionMessages`. Este método admite como primer parámetro una cadena que indica el tipo de error (lo más típico es usar la constante `ActionMessages.GLOBAL_MESSAGE`) y como segundo parámetro el propio objeto `ActionMessage`. Si en el tipo de error utilizamos un nombre arbitrario, quedará asociado a este y podremos mostrarlo específicamente en la página JSP.

Para mostrar los errores en la página JSP se puede la etiqueta `<html:messages/>` , que viene con las *taglibs* de Struts. Dicha etiqueta va iterando por los `ActionMessages`, de manera que podemos mostrar su valor. En un JSP incluiríamos código similar al siguiente:

```
<!-- referenciar la taglib de Struts que include la etiqueta -->
<%@taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
...
<!-- mostrar los mensajes almacenados -->
<html:messages id="e">
<ul>
   <li>${e}</li>
</ul>
</html:messages>
```

como se ve, la etiqueta `<html:messages>` va iterando por una variable, de nombre arbitrario, fijado por nosotros. Al mostrar el contenido de la variable estamos mostrando el mensaje de error. En el ejemplo anterior se utiliza una lista con viñetas de HTML simplemente por mejorar el aspecto de los mensajes.

También podemos mostrar solo ciertos mensajes, en lugar de todos. En ese caso, a la hora de guardarlos debemos asignarles un identificador arbitrario en lugar de `ActionMessages.GLOBAL_MESSAGE`

```
errors.add("password", new ActionMessage("error.passwordcorto"));
```

Para mostrar solo este mensaje en el JSP, usaríamos el atributo `property` de la etiqueta `</html:messages>`:

```
<html:messages id="e" property="password">
  ${e}
</html:messages>
```

Otra posibilidad más simple de mostrar solo un mensaje es usar la etiqueta

`<html:errors>`, que tiene también el mismo atributo `property` pero solo muestra un mensaje, no haciendo iteración por la lista de mensajes disponibles.

Veremos con más detalle otras etiquetas de las *taglibs* de Struts en el tema siguiente.

1.3.4. Tratamiento de excepciones

Si una de nuestras acciones lanza una excepción comprobada, la capturará el servlet controlador de Struts. Podemos especificar qué debe hacer el controlador ante una determinada excepción con el elemento `exception` en el fichero XML. Este elemento puede estar dentro de un `action` concreto o bien puede ser global a todas las acciones. En él especificamos la página responsable de mostrar el error y la clave del fichero `.properties` del que se sacará su texto:

```
<?xml version = "1.0" encoding = "ISO-8859-1"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts
 Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>

  <global-exceptions>
        <exception type="es.ua.jtech.ejemplos.ExcepcionEjemplo"
                           key="error.ejemplo"
                           path="/error.jsp"/>
  </global-exceptions>

  <global-forwards>
        <forward name="errorUsuario" path="/error.html"/>
  </global-forwards>
  ...
</struts-config>
```

El mensaje de error de la excepción se puede mostrar con la etiqueta `<html:errors>`, permitiendo uniformizar el tratamiento de excepciones y el de errores.

La etiqueta `<exception/>` también se puede colocar dentro de un `<action/>`, si sabemos que la excepción solamente la va a generar una acción concreta.

1.3.5. Acceder a una página a través de una acción

Si al movernos de una página a otra no vamos a realizar ninguna operación, sino que únicamente estamos navegando, podemos poner simplemente un enlace al .jsp o .html. No obstante, esto va "contra la filosofía" de MVC, en la que todas las peticiones pasan por el controlador. Además y como veremos, el controlador puede verificar automáticamente los permisos de acceso, de modo que en muchos casos será vital que todas las peticiones pasen por el controlador.La forma más sencilla de conseguirlo en Struts es mediante el atributo `forward` de la etiqueta `<action>`

```
<action path="/registroNuevoUsuario" forward="/registro.jsp">
</action>
```

De esta manera, la petición a la URL `registroNuevoUsuario.do`, se redirige a la página `registro.jsp` pero pasando antes por el controlador.

1.4. Seguridad declarativa en Struts

La versión 1.1 de Struts introdujo la seguridad basada en acciones. Esto quiere decir que podemos combinar los mecanismos estándar de seguridad declarativa J2EE con el funcionamiento de nuestra aplicación Struts. Por tanto, primero necesitamos configurar el web.xml para restringir el acceso a los recursos protegidos a los usuarios que tengan determinado rol.

Para cada acción especificaremos qué rol o roles pueden ejecutarla, mediante el atributo `roles` de la etiqueta `<action>` en el `struts-config.xml`.Por ejemplo:

```
<action roles="admin,manager"
        path="/admin/borrarUsuario"
        ...
</action>
```

Si se intenta ejecutar una acción para la que no se tiene permiso, el controlador lanzará una excepción de tipo `org.apache.struts.chain.commands.UnauthorizedActionException`. Ya hemos visto cómo capturar una excepción y hacer que el navegador se redirija a una página de error.

2. Ejercicios sesión 1 - Introducción a Struts

2.1. Instalación de la aplicación de ejemplo

Durante las cuatro sesiones de Struts realizaremos los ejercicios sobre la misma aplicación de ejemplo. Se trata de una sencilla aplicación para almacenar las tareas pendientes (ToDo) de un usuario. Hay que tener en cuenta que es simplemente una aplicación de ejemplo. El objetivo es tener una aplicación para poder "trastear" con Struts más compleja que un "hola mundo" pero no tanto como una aplicación real. Tanto la arquitectura de la aplicación como el interfaz de usuario se deberían mejorar en una implementación más realista.

Los casos de uso de la aplicación son:

- Hacer login
- Ver la lista de tareas pendientes
- Crear una nueva tarea
- Eliminar una tarea existente

Para instalar la aplicación, seguid estos pasos:

1. Bajáos el proyecto de Eclipse e importadlo a vuestro espacio de trabajo
2. **Crear la base de datos:** en la carpeta `database` hay un script SQL para crear la base de datos de MySQL. Podéis crearla con las herramientas gráficas de MySQL o bien tecleando desde una terminal

```
mysql -u root -p < todo.sql (os pedirá el password: especialista)
```

3. Para aseguraros de que la conexión con la base de datos funciona, haced login con el usuario "struts" y el password "mola". Debe aparecer una lista de tareas pendientes.

2.2. Implementar un caso de uso completo (1 punto)

Como mejor se ve el funcionamiento de Struts es siguiendo el flujo de ejecución de un caso de uso completo. Para ello vais a implementarlo vosotros y luego probarlo. El caso de uso en cuestión es el de mostrar todos los datos de una tarea. En la página que lista todas las tareas hay un enlace "Ver detalles" al lado de cada una, que apunta a "verTarea.do" . Falta por implementar:

- La clase Java con la acción
- El JSP que muestre el resultado (verTarea.jsp)
- El mapeo en el struts-config.xml entre la acción y los posibles resultados. En este caso habrá un único resultado posible, "OK".

En la plantilla, la clase `es.ua.jtech.struts.acciones.VerTareaAccion` devuelve simplemente el forward "OK". El JSP "verTarea.jsp" muestra simplemente un mensaje de

saludo. Se recomienda que probéis primero a hacer el mapeo en el struts-config.xml, comprobéis que funciona y luego hagáis la implementación real de la acción y del JSP.

Veámoslo con un poco más de detalle:

1. En cuanto a **La clase Java VerTareaAccion**.
   - El único posible resultado (*forward*) por el momento será "OK".
   - El acceso a la base de datos nos lo da la clase `TareaDAO`. Para obtener una instancia, llamar a `TareaDAO.getInstance()`. Para obtener los datos de una tarea, llamar a `getTarea(id_de_la_tarea)`. El id de la tarea lo podemos sacar del parámetro HTTP "id"
   - Una vez se obtenga un objeto `Tarea` con todos los datos, colocarlo en el `request` para que se pueda mostrar desde el JSP
2. En cuanto a **La vista `verTarea.jsp`**: Usad el lenguaje de expresiones de JSP para simplificar la sintaxis, podéis tomar como ejemplo `tareas.jsp`.

2.3. Gestionar errores en las acciones (1 punto)

Añadir gestión de errores a la acción `VerTarea`, de modo que si se solicita el id de una tarea no existente, se detecte la situación y aparezca un mensaje de error.

Pasos a seguir:

1. En el fichero `mensajes.properties` (carpeta `resources`) añadir un mensaje de error para este caso.
2. En el código de la acción:
   - Si el método `getTarea(id)` de `TareaDAO` devuelve `null`, la tarea no existe. Por tanto en este caso habrá que devolver otro resultado distinto de "OK". Llamadlo "error".
   - En caso de error, crear un `ActionMessage` asociado al mensaje del fichero `mensajes.properties` y guardarlo con `saveErrors` como se explica en los apuntes.
3. En el struts-config.xml, asociar el resultado "error" con la página error.jsp, que deberás crear.
4. En dicha página usar la etiqueta `<html:messages>` para mostrar el error.

2.4. Seguridad declarativa (1 punto)

La aplicación de tareas pendientes usa seguridad declarativa. Hay dos clases de usuarios (dos roles): "registrado" y "pro". Modifica la aplicación para que solo los usuarios "pro" puedan ver los detalles de la tarea.

- Modifica el struts-config.xml para que la acción de VerTarea solo se pueda ejecutar con rol "pro"
- Haz que si intenta acceder un usuario que no tenga este rol se salte a la página "sinPermiso.jsp", que deberás crear, mostrando en ella un mensaje de error apropiado.

3. La vista: ActionForms y taglibs propias

En este tema trataremos sobre la vista en Struts. En este aspecto el framework nos aporta unos componentes muy útiles que actúan de interfaz entre la vista y las acciones: los *ActionForms*. Como veremos, estos objetos nos permiten recoger automáticamente los valores de los formularios y validar sus datos, entre otras funciones. Veremos también las librerías de etiquetas propias de Struts. Aunque algunas de ellas han quedado obsoletas tras la aparición de JSTL, otras siguen siendo de utilidad en conjunto con los ActionForms.

3.1. ActionForms

3.1.1. Introducción

Aunque los datos introducidos en formularios pueden obtenerse dentro del código de las acciones directamente de la petición HTTP (como hemos visto en los ejemplos del tema anterior), Struts ofrece un mecanismo alternativo que proporciona distintas ventajas: los **ActionForm**s. Empleando ActionForms podemos conseguir:

- **Recolección automática de datos** a partir de los incluidos en la petición HTTP.
- **Validación de datos modular** (realizada fuera del código de la acción), y en caso de utilizar el plugin *Validator*, validación automática según lo especificado en un fichero de configuración.
- **Recuperación de los datos** para volver a rellenar formularios. De esta forma se evita el típico problema de que el usuario tenga que volver a rellenar un formulario entero porque uno de los datos es incorrecto.

Podemos considerar un ActionForm como si fuera un JavaBean que captura los datos de un formulario. Los datos se pueden extraer, validar, cambiar y volver a colocar en otro formulario. No obstante, no tiene por qué haber una correspondencia uno a uno entre un ActionForm y un formulario HTML, de manera que se puede utilizar el mismo ActionForm para englobar varios formularios separados en distintas páginas (caso típico de un asistente). También se puede reutilizar el mismo ActionForm en distintos formularios que compartan datos (por ejemplo, para dar de alta o modificar los datos de un usuario registrado).

En Struts hay dos tipos principales de ActionForms

- Clases descendientes de la clase base **ActionForm**. Deben incorporar métodos Java para obtener/cambiar cada uno de los datos (al estilo JavaBeans), cuya definición puede resultar tediosa. Desde Struts 1.1 las propiedades se pueden almacenar en un Map, con lo que solo es necesario definir un único método para acceso a todas las propiedades, pasando como parámetro el nombre de la deseada.
- Instancias de la clase **DynaActionForm** o descendientes de ésta. Permiten definir los

campos en el fichero `struts-config.xml`, de manera que se pueden añadir o modificar campos minimizando la necesidad de recompilar código. Mediante la clase **DynaValidatorForm** se puede incluso validar datos de manera automática, según las reglas especificadas en un fichero de configuración aparte.

3.1.2. El ciclo de vida de un ActionForm

La generación y procesamiento de un ActionForm pasa por varias etapas:

1. El controlador recibe la petición del usuario, y chequea si la acción asociada utiliza un ActionForm. De ser así, crea el objeto
2. Se llama al método `reset()` del objeto, que el desarrollador puede sobreescribir para "limpiar" sus campos. Esto tiene sentido si el ActionForm persiste más allá de la petición actual (lo cual se puede especificar al definirlo).
3. El ActionForm se almacena en el ámbito especificado en su definición (petición, sesión o aplicación)
4. Los datos del ActionForm se rellenan con los que contiene la petición HTTP. Cada parámetro HTTP se asocia con el dato del mismo nombre del ActionForm. Un punto importante es que en HTTP los parámetros son cadenas, con lo que en principio las propiedades del ActionForm deben ser Strings, aunque los valores booleanos se convierten a boolean. No obstante, más allá de esta conversión básica no hay conversión automática de tipos.
5. Se validan los datos, llamando al método `validate()`, que el desarrollador debe sobreescribir para implementar la validación deseada.
6. Si se han producido errores de validación, se efectúa una redirección a la página especificada para este caso. Si no, se llama al `execute()` de la acción y finalmente se muestra la vista asociada a esta.
7. Si en la vista se utilizan las *taglibs* de Struts para mostrar datos, aparecerán los datos del ActionForm.

3.1.3. Cómo definir un ActionForm

Los ActionForm se definen dentro del fichero `struts-config.xml`, dentro de la sección `<form-beans>`. Cada ActionForm viene definido por un elemento `<form-bean>`. Por ejemplo, para definir un ActionForm con el nombre `FormLogin` y asociado a la clase Java `acciones.forms.FormLogin` haríamos:

```
<form-beans>
   <form-bean name="FormLogin" type="acciones.forms.FormLogin">
</form-beans>
```

Dentro de un momento veremos qué condiciones debe cumplir la clase `acciones.forms.FormLogin`, que tendremos que definir nosotros.

Para que los datos de un `ActionForm` sean accesibles a una acción, hay que definir una serie de atributos dentro del elemento `action` de `struts-config.xml`. Por ejemplo, para

asociar un ActionForm de la clase `FormLogin` a la acción `login` de los ejemplos anteriores, se podría hacer:

```
<action path="/login" type="acciones.AccionLogin"
        name="FormLogin" scope="session"
        validate="true" input="/index.jsp">
   <forward name="OK" path="/personal.jsp"/>
   <forward name="errorUsuario" path="/error.html"/>
</action>
```

El atributo `name` indica el nombre simbólico para el `ActionForm`. El `scope` tiene el mismo significado que cuando tratamos con JavaBeans. En caso de que se desee validar los datos, hay que especificar el atributo `validate=true` y utilizar el atributo `input` para designar la página a la que se volverá si se han producido errores de validación.

Nota:

Como puede verse, los nombres de los atributos del ActionForm en el XML no son demasiado descriptivos del papel que desempeñan (excepto, quizás, scope). Este es un problema reconocido por los propios desarrolladores de Struts y que se soluciona en la versión 2 del framework.

Por otro lado hay que escribir el código Java con la clase del ActionForm. Esta tarea es muy similar a definir un JavaBean. Hay que especificar métodos get/set para cada campo, y colocar la lógica de validación en el método `validate()`. Por ejemplo:

```
package acciones.formularios;
import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class FormLogin extends ActionForm {
   private String login;
   private String password;

   public void setLogin(String login) {
      this.login = login;
   }
   public String getLogin() {
      return login;
   }

   public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest request) {
      ActionErrors errores = new ActionErrors();
      if ((getlogin()==null)||(getLogin.equals("")))
         errores.add(ActionMessages.GLOBAL_MESSAGE
                     , new
ActionMessage("error.requerido.usuario"));
      return errores;
   }
}
```

Nótese que el ActionForm no es más que un bean, eso sí, debe heredar de la clase

`ActionForm`.

El método `validate()` debe devolver una colección de errores (o null), representada por el objeto **ActionErrors**.

Aviso:
Si en el struts-config.xml ponemos validate="true" o no especificamos el valor de este atributo, Struts llamará al método validate() de nuestro ActionForm, generando un error si éste no existe. Si no deseamos validar los datos, debemos ponerlo específicamente con validate="false"

Desde la versión 1.1 de Struts, se pueden utilizar ActionForms que almacenen internamente las propiedades en un objeto Map (*Map-backed Action Forms)*. Para el acceso a/modificación de cualquier propiedad solo será necesario implementar dos métodos

```
public void setValue(String clave, Object valor)
public Object getValue(String clave)
```

3.1.4. Tipos de datos del ActionForm: conversión y validación

En el ejemplo anterior, todas las propiedades del ActionForm eran Strings, pero esto no tiene por qué ser siempre así. Una propiedad de un ActionForm puede ser de cualquier tipo, pero como Struts copia automáticamente los valores de la petición HTTP a las propiedades del ActionForm tiene que "saber hacer" la conversión de tipos. Struts puede convertir de String (el tipo que tienen todos los parámetros HTTP) a cualquier tipo primitivo de Java. El problema es que si usamos una propiedad de tipo int, por ejemplo, y el usuario escribe en el campo de formulario correspondiente algo que no se puede convertir a entero, la conversión fallará y el campo pasará a valer 0. Esto puede causar problemas si queremos mostrar de nuevo los valores introducidos para que el usuario los pueda corregir (veremos cómo se hace esto con las taglibs de Struts). Para el usuario puede ser un poco desconcertante ver un 0 donde antes se había tecleado otra cosa totalmente distinta. Adoptando por tanto un punto de vista pragmático, muchos desarrolladores de Struts usan solo Strings para las propiedades de los ActionForms y se ocupan manualmente de hacer la conversión y de detectar los posibles errores. Esto es tedioso, pero es difícil de evitar dado el diseño de Struts.

3.1.5. DynaActionForms

En ActionForms con muchos campos, resulta tedioso tener que definir manualmente métodos *get/set* para cada uno de ellos. En su lugar, podemos utilizar `DynaActionForms`, en los que los campos se definen en el fichero de configuración de Struts. Por ejemplo:

```
 <form-bean name="FormLogin"
type="org.apache.struts.action.DynaActionForm">
```

```
  <form-property name="login" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
</form-bean>
```

Como se ve, en este caso la clase del ActionForm no la definimos nosotros sino que es propia de Struts. Para acceder a los campos del `DynaActionForm` se usa un método genérico **get** al que se pasa como parámetro el nombre del campo (por ejemplo `get("login")`). Algo similar ocurre para cambiar el valor de un campo (método **set**).

Nota:

En JSP, el acceso a una propiedad de un DynaActionForm con EL se debe hacer a través de la propiedad predefinida `map`. Así en el ejemplo anterior, la propiedad login sería accesible con ${*nombreDelBean*.map.login}

Para validar un `DynaActionForm`, tendremos que definir una clase propia que herede de `org.apache.struts.action.DynaActionForm`, y sobreescribir su método `validate()`, o mejor aún, podemos validar datos automáticamente utilizando el plugin de Struts denominado **validator**, que será objeto del siguiente tema.

3.2. La taglib HTML de Struts

Struts viene con varias taglibs: de ellas, la más importante es la HTML, que es la que sirve para capturar/mostrar los datos de los ActionForms. Esta taglib es muy útil en cualquier aplicación de Struts. Entre otras cosas, podemos volver a mostrar los valores introducidos en un formulario si se ha producido un error en alguno sin que el usuario tenga que rellenarlos todos de nuevo y generar *checkboxes*, opciones de un `select` etc a partir de colecciones o arrays, sin tener que hacerlas una por una.

Las otras taglibs (*logic*, *bean*,...) aunque también muy útiles en su momento, quedaron obsoletas con la aparición de JSTL. Por tanto discutiremos aquí únicamente la taglib de HTML. Además, nos centraremos sobre todo en la definición de formularios y campos de formulario.

Para poder usar la taglib HTML en un JSP, hay que incluir en el JSP la directiva

```
  <%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
```

La versión "clásica" de la taglib HTML no permite el uso del lenguaje de expresiones (EL). Si deseamos usarlo en nuestras páginas, hay que importar a nuestro proyecto el archivo `struts-el.jar` y usar en los JSPs la directiva

```
 <%@ taglib uri="http://struts.apache.org/tags-html-el"
prefix="html-el" %>
```

Aviso:
En algunas versiones anteriores de Struts las URIs comienzan por jakarta.apache.org en lugar de struts.apache.org, consecuencia del cambio de dominio del proyecto Struts.

3.2.1. Enlaces

Se pueden generar enlaces a acciones con la etiqueta `link`. En su variante más simple, un enlace a la acción "login" se haría con:

```
<html:link action="/login">Login</html:link>
```

Nótese que en el tag se usa directamente el nombre de la acción, no la URL (que sería habitualmente `login.do`)

Normalmente los enlaces a acciones requerirán algún tipo de parámetro. Por ejemplo, podríamos necesitar que se generara el siguiente enlace:

```
<a href="editUsuario.do?login=pepe">Editar usuario</a>
```

Con el tag link podemos generar dicho parámetro. Lo interesante es que podemos sacar el valor del parámetro de un bean o de un actionform. Con el atributo `paramId` especificamos el nombre del parámetro a generar. `paramName` es el nombre del bean del que se obtiene el parámetro, y `paramProperty` la propiedad del bean que nos da su valor. Suponiendo que tenemos un bean accesible llamado usuario, con una propiedad login, podemos hacer

```
<html:link action="/editUsuario" paramId="login"
paramName="usuario" paramProperty="login">
  Editar usuario
</html:link>
```

3.2.2. Definición de un formulario

Lo primero que necesitamos para definir un formulario con las taglibs de Struts es una acción a ejecutar y un ActionForm asociado a dicha acción. Sin acción o ActionForm no tiene mucho sentido usar para el formulario la taglib de Struts (usaríamos directamente un formulario HTML).

En general, las etiquetas de la taglib HTML de Struts son muy similares a su contrapartida HTML. El caso de la definición de formulario no es una excepción. Por ejemplo, vamos a ir viendo cómo se podría hacer un formulario de registro de nuevo usuario con las taglibs de Struts:

```
  <%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"
%>
  <html>
  <head><title>Ejemplo de tag form<title></head>
```

```
  <body>
    <html:form action="/registrar">
     <-- aquí vienen los campos. Ahora veremos cómo se definen -->
    </html:form>
  </body>
  </html>
```

La primera diferencia que se puede observar entre este formulario y uno HTML (aparte del prefijo `html:`) es que el nombre del `action` no es una URL "física" sino una acción de Struts.

El primer campo que necesitamos para que al menos los datos lleguen al servidor es un botón de tipo `submit`. Struts nos ofrece además la posibilidad de definir un botón para cancelar. Veamos de nuevo el ejemplo anterior (pero ahora omitiendo el HTML de fuera del formulario).

```
<html:form action="/registrar">
   <html:submit>Registrar nuevo usuario</html:submit>
   <html:cancel>Cancelar</html:cancel>
</html:form>
```

Como vemos, la principal diferencia de sintaxis con el HTML "puro" es que el "cartel" del botón se pone dentro de la etiqueta, en lugar de con el atributo "value". La acción asociada se disparará se haya pulsado en el submit o en el cancel, pero en este último caso el método `isCancelled` de la clase `Action` devolverá `true`.

```
  ...
   public ActionForward execute(
           ActionMapping mapping,
           ActionForm form,
           HttpServletRequest request,
           HttpServletResponse response)
           throws Exception {
           ...
           if (isCancelled(request)) {
                   ...
           }
           ...
  }
```

Cuando una acción se cancela no se efectúa la validación del ActionForm. Struts controla la cancelación enviando un parámetro HTTP especial, lo cual puede dar lugar a que usuarios malintencionados intenten manipular este parámetro para evitar la validación. En Struts 1.2.X, a partir de la versión 1.2.9 cuando una acción es cancelable hay que especificarlo en el `struts-config.xml`:

```
<action path="/registrar"
```

```
        input="/registrar.jsp"
        validate="true">
    <set-property property="cancellable" value="true"/>
    <forward name="OK" path="/resultado.jsp"/>
</action>
```

En caso de no hacer esto, la acción lanzará una excepción de tipo `InvalidCancelException`.

A partir de Struts 1.3.X esto se hace con un atributo `cancellable`

```
<action path="/registrar"
        input="/registrar.jsp"
        validate="true" cancellable="true">
    <forward name="OK" path="/resultado.jsp"/>
</action>
```

3.2.3. Campos de texto

Los campos para introducir texto son muy similares a los de HTML, con la diferencia de que en lugar de elegir el tipo de campo con el atributo type se hace con la propia etiqueta. El atributo property nos permite asociar un campo a la propiedad del mismo nombre del ActionForm asociado al formulario. Por ejemplo:

```
<html:form action="/registrar">
   <html:text property="login">Introduce aquí el nombre que deseas
tener</html:text>
   <html:password property="password"/>
   <html:submit>Registrar nuevo usuario</html:submit>
   <html:cancel>Cancelar</html:cancel>
</html:form>
```

Para que el ejemplo funcione, debemos tener un ActionForm asociado a la acción "registrar" que tenga al menos las propiedades login y password. Como ya se ha visto en el ciclo de vida de los ActionForms, cuando se rellena el formulario Struts copiará los datos introducidos en las propiedades correspondientes del ActionForm. A la inversa, si el método validate() del ActionForm devuelve false y el struts-config.xml indica que hay que volver a la misma página, Struts copiará los datos desde el ActionForm al formulario, para que el usuario pueda corregirlos.

En el caso de las contraseñas, puede que no resulte adecuado volver a mostrar el valor introducido. Para ello se utiliza el atributo `redisplay`, al que hay que asignar el valor `false`

Como es de esperar a estas alturas (visto el resto de etiquetas) los campos de texto multilínea se consiguen con la etiqueta `html:textarea`.

```
<html:textarea property="descripcion" cols="40" rows="6"/>
```

3.2.4. Cuadros de lista

La sintaxis de los cuadros de lista es prácticamente idéntica a la de HTML, si es que vamos a poner las opciones manualmente, una por una:

```
<html:select property="sexo">
   <html:option value="H">Hombre</html:option>
   <html:option value="M">Mujer</html:option>
   <html:option value="N">No especificado</html:option>
</html:select>
```

Struts asociará este campo a la propiedad "sexo" del ActionForm correspondiente a la acción ejecutada

3.2.4.1. Tomar los valores para "option" automáticamente de un objeto

Lo interesante es que Struts puede tomar los valores del select de una colección o un array, ahorrándonos el tener que especificar las opciones "a mano". Para ello, debemos tener en la petición o la sesión el array con las opciones. La manera más sencilla de conseguir esto (sin meterlo en el propio JSP) es ponerlo en la acción que nos lleva a la página con el formulario. Veamos un ejemplo, con el siguiente struts-config.xml

```
...
<form-beans>
   <form-bean name="FormRegistro" type="actionForms.FormRegistro"/>
</form-beans>
...
<action-mappings>
    <action
       path="/prepararRegistro"
       type="acciones.AccionPrepararRegistro">
       <forward name="OK" path="/registro.jsp"/>
    </action>
    <action
       path="/registrar"
       name="FormRegistro" scope="request"
       type="acciones.AccionRegistrar">
       <forward name="OK" path="/resultRegistro.jsp"/>
    </action>
</action-mappings>
...
```

Como vemos, la acción `AccionPrepararRegistro` muestra la página `registro.jsp`, que contiene el formulario que veíamos en los ejemplos anteriores. Este formulario llama a `registrar.do`, que resulta en la ejecución de `AccionRegistrar`. El sitio más adecuado

para colocar en algún ámbito (petición, sesión o aplicación) la lista de opciones para el select sería entonces en `AccionPrepararRegiatro`, simplemente haciendo algo como

```
String[] lista = { "Hombre", "Mujer", "No especificado"};
request.setAttribute("listaSexos", lista);
```

Nótese que aunque hemos usado un array, `listaSexos` podría ser cualquier tipo de `Collection`

Ahora podemos decirle a Struts en el HTML que tome las opciones del objeto `listaSexos`. Automáticamente buscará este en la petición, sesión y aplicación, por este orden.

```
<html:select property="sexo">
   <html:options value="listaSexos"/>
</html:select>
```

Esto nos generará un select en HTML en el que tanto los atributos "value" como las etiquetas que se muestran en pantalla se toman de `listaSexos`. Si deseamos que los value sean distintos a las etiquetas haríamos algo como

```
<html:select property="sexo">
   <html:options name="codSexos" labelName="listaSexos"/>
</html:select>
```

Así, los valores del atributo value se tomarán del objeto `codSexos` (que debemos haber instanciado antes) y las etiquetas que se muestran en pantalla de `listaPerfiles`

3.2.4.2. Tomar los valores para "option" automáticamente de un ActionForm

Podemos también tomar los valores para las etiquetas "option" de un ActionForm. Lo más lógico es usar el asociado al formulario (hablando con más propiedad, el asociado a la acción disparada por el formulario). En ese caso, debemos añadir a nuestro ActionForm un método "getXXX" que devuelva un array o colección con el que se rellenará el cuadro de lista. Por ejemplo:

```
  public class FormRegistro {
   ...
   private static String[] listaSexos = { "Hombre", "Mujer", "No
especificado"};

   public String[] getListaSexos() {
          return listaSexos;
   }
   ...
```

Ahora, en el JSP hacemos

```
...
<html:select property="sexo">
  <html:options property="listaSexos"/>
</html:select>
...
```

También podemos tomar de un "getXXX" los valores del atributo "value" y de otro "getYYY" las etiquetas que se muestran en pantalla. Para especificar de dónde sacar las etiquetas, usaremos el atributo `labelProperty` de `<html:options>`.

3.2.5. Botones de radio

La etiqueta de Struts para definir botones de radio es prácticamente equivalente a la de HTM, con la diferencia básica de que podemos asociar el botón a una propiedad de un ActionForm:

```
<html:radio property="sexo" value="hombre"/> hombre
<html:radio property="sexo" value="mujer"/> mujer
```

Para generar varios botones de radio con distintos `value` de manera automática no tendremos más remedio que usar JSTL, ya que Struts no ofrece ningún mecanismo para hacerlo. Por ejemplo, supongamos que el ActionForm `FormRegistro` tiene un método `getListaSexos` que devuelve los valores posibles para el sexo, como en el ejemplo de la sección anterior. Podríamos hacer:

```
<%@ taglib uri="http://struts.apache.org/tags-html-el"
prefix="html-el" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
...
<c:forEach items="${FormRegistro.listaSexos}" var="sexo">
   <html-el:radio property="sexo" value="${sexo}"/> ${sexo}
</c:forEach>
```

Nótese que en el ejemplo anterior hemos usado la versión EL de la taglib para poder emplear el lenguaje de expresiones dentro de las etiquetas de Struts.

3.2.6. Casillas de verificación

Struts nos ofrece dos etiquetas para hacer casillas de verificación (*checkboxes*), según si queremos una sola casilla o una serie de casillas relacionadas. En el primer caso, la asociación se hace con una propiedad booleana de un ActionForm y en el segundo con un

array de valores, normalmente Strings (ya que en este caso podemos marcar varias casillas simultáneamente)

Cuando queremos una única casilla podemos usar la etiqueta `checkbox`, que es prácticamente equivalente a su contrapartida HTML:

```
<html:checkbox property="publi"/> Sí, deseo recibir publicidad
sobre sus productos
```

El checkbox se asociaría a una propiedad booleana del ActionForm llamada "publi". Deben existir los métodos isPubli/getPubli para que Struts pueda consultar/fijar su valor.

Aviso:

Hay que llevar mucho cuidado con las casillas de verificación si se usa un ActionForm con ámbito de sesión o de aplicación. Cuando el checkbox no se marca, el navegador **no envía** el valor asociado, ni siquiera como false ni nada similar. Esto puede causar el que aunque el usuario ha desmarcado la casilla, la propiedad asociada sigue con true, ya que Struts no ha llamado al setXXX correspondiente. Para evitarlo, se debe inicializar la propiedad asociada a `false` en el método `reset()` del ActionForm. De este modo, antes de mostrar el formulario Struts pone el valor a `false`. Cuando se envían los datos, si la casilla está marcada cambiará a `true` y si no, no ocurrirá nada.

Para hacer más de una casilla de verificación asociada a una sola propiedad de un ActionForm se usa la etiqueta `multibox`. Dentro de la etiqueta hay que poner el valor que se añadirá a la propiedad. Decimos que se añadirá porque la propiedad debe ser un array de valores, donde se colocarán los asociados a las casillas marcadas.

```
<html:multibox property="aficiones">cine</html:multibox> Cine
<html:multibox property="aficiones">música</html:multibox> Música
<html:multibox property="aficiones">videojuegos</html:multibox>
Videojuegos
```

Como en el caso de los botones de radio, si tenemos muchas opciones es más conveniente generar las casillas de manera automática. Esto tendremos que hacerlo iterando sobre una colección con los valores, como en la sección anterior. Suponiendo que el ActionForm tiene un método `getListaAficiones` que devuelve un array con "cine", "música" y "videojuegos", El ejemplo anterior se convertiría en:

```
<c:forEach items="${FormRegistro.listaAficiones}" var="aficion">
   <html-el:multibox
property="aficiones">${aficion}</html-el:multibox> ${aficion}
</c:forEach>
```

4. Ejercicios de ActionForms y TagLibs

En esta sesión, seguiremos trabajando sobre la aplicación de ejemplo de la sesión anterior.

4.1. Uso de ActionForms (1 punto)

En la acción java `NuevaTareaAccion` los datos se toman directamente de la petición HTTP. En vez de hacer esto, crear un ActionForm para recolectar los datos.

1. Crear una nueva clase java `es.ua.jtech.struts.actionforms.TareaForm` con las propiedades necesarias y los métodos get/set. **Importante:** ¿de qué tipo deberían ser las propiedades?. Implementar el método `validate` para controlar errores de validación (fecha incorrecta, prioridad no válida, días de aviso no es un número positivo). Tened en cuenta que podéis aprovechar código de validación que ahora está en el método `validar` de `NuevaTareaAccion`.
2. Cambiar el código de `NuevaTareaAccion` para que tome los datos del ActionForm en lugar de hacerlo directamente de la petición HTTP.
3. en el `struts-config.xml`, definir el ActionForm, dentro de la sección `<form-beans>` y asociarlo a la acción, dentro de la etiqueta `<action>`, usando los atributos `name`, `validate` e `input`. Si hay un error de validación hay que volver a la misma página, `nuevaTarea.jsp`.

4.2. Uso de la taglib HTML (1 punto)

Cambiar el formulario de la página `nuevaTarea.jsp` para que use las etiquetas propias de Struts. Comprobar que cuando hay un error se vuelven a mostrar los datos escritos en el formulario. Mostrar además al lado de cada campo el error asociado (si puede haberlo) usando la etiqueta `<html:messages>` que vimos en la sesión anterior.

4.3. Uso de DynaActionForms (1 punto)

Cambia el ActionForm de NuevaTareaAccion por un DynaActionForm. Deja comentado el código fuente que insertaste en el primer ejercicio para que quede constancia de que lo hiciste. Deja también comentados los cambios que introdujiste en el struts-config para el primer ejercicio.

5. Validación. Internacionalización

5.1. 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.

5.1.1. 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.

5.1.2. 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

5.1.3. 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`.

5.1.4. Mensajes de error

5.1.4.1. ¿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.

5.1.4.2. 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
```

5.1.5. 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.

5.1.6. 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.

5.2. 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.

5.2.1. 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`

5.2.1.1. 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*.

5.2.1.2. 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`.

5.2.1.3. 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}
```

5.2.2. 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`).

5.2.2.1. 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.

5.2.2.2. 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");
```

5.2.2.3. 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.

5.2.2.4. 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.

6. Ejercicios de validación e internacionalización

6.1. Validación automática de datos (1 punto)

Usar el plugin validator para realizar la validación de los datos en el formulario de nueva tarea. Ahora hay que validar:

- Que el título (descripción) de la tarea no sea vacío
- Que el número de días de aviso sea un entero entre 1 y 365
- Que la fecha tenga un formato correcto (dd/MM/aaaa)

Modificar la clase java del ActionForm para que ahora herede de ValidatorForm. Cambiad de nombre el nombre del método `validate` que ahora tenéis por `validateOld` (para que no interfiera).

6.2. Internacionalización (1 punto)

El objetivo es internacionalizar la aplicación StrutsToDo. Hay que localizarla para dos idiomas: español e inglés. Cambiad por lo menos los mensajes que aparecen en la página inicial (la de login) y la de la lista de tareas. El usuario podrá elegir el idioma en que desea ver la aplicación pinchando en una opción "español" o "ingles" en la página inicial. Dicha opción cambiará el locale actual usando una acción llamada `CambiaIdiomaAccion` y luego volverá de nuevo a la página inicial. **Opcionalmente**, haced que el locale por defecto sea el que tiene configurado el usuario en su navegador, de modo que la aplicación salga inicialmente en el idioma apropiado sin necesidad de elegir la opción.

6.3. Verificación de duplicados (1 punto)

Verificar que al crear una nueva tarea no se puedan crear dos veces la misma volviendo atrás en el navegador y volviéndolo a intentar. Fijaos en que habrá que crear una clase `AccionPreNuevaTarea` que será la que inicialice el token y lleve a mostrar el formulario de `nuevaTarea.jsp`

7. Introducción a Struts 2

Con el paso de los años, Struts se ha convertido en EL framework MVC para JavaEE por excelencia. No obstante, también se ha quedado un poco anticuado. Nacido en otra época, el diseño original de Struts no respeta principios que hoy se consideran básicos en JavaEE, como el uso de APIs "no invasivos" que minimicen las dependencias en nuestro código de los APIs del framework, o la facilidad para hacer pruebas unitarias.

Struts 2 es la "siguiente generación" de Struts, diseñado desde el comienzo con estos principios básicos en mente. Vamos a ver en esta sesión una introducción rápida a las ideas fundamentales de la versión 2. Dadas las limitaciones de tiempo y espacio, y la suposición previa de que se conoce Struts 1.x, comenzaremos por los cambios fundamentales introducidos en la versión 2 con respecto a la 1.x, seguida de una breve introducción a ciertas ideas (interceptores, *value stack*, etc), que aun siendo básicas en Struts 2 no existían en la versión 1.x. Para abordar Struts 2 partiendo de cero seguiríamos un orden distinto en la introducción de conceptos.

7.1. Configuración

Lo primero es copiar los JARs de Struts 2 en nuestro proyecto. En la web del framework se incluye una plantilla de ejemplo con los JARs necesarios. Como mínimo necesitaremos las versiones correspondientes de `struts-core`, `xwork`, `ognl`, `freemarker` y `commons-logging`.

Al igual que en Struts 1.x debemos hacer que todas las peticiones se redirijan al servlet que hace de controlador. En el `web.xml` de la aplicación haremos:

```
<filter>
   <filter-name>struts2</filter-name>
<filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
</filter>
<filter-mapping>
   <filter-name>struts2</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Nótese que en Struts 2 el mapeo de las peticiones se hace con un filtro de servlets en lugar de con un `servlet-mapping`.

El fichero de configuración XML cambia de formato, haciéndose más conciso. Además, por defecto pasa a llamarse `struts.xml` en lugar de `struts-config.xml` y a residir en el CLASSPATH en lugar de en `WEB-INF`. Su estructura básica es:

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
2.0//EN"
```

```
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
  <package name="default" extends="struts-default">
    <action ...>
        <result>...</result>
    </action>
                ...
  </package>
</struts>
```

Donde cada `package` define un conjunto de acciones, heredando su configuración de algún otro `package`. En todos los ejemplos usaremos el `struts-default`, que se define en la distribución estándar, e incluye una serie de elementos básicos para el trabajo con el framework.

Siguiendo las "tendencias" actuales en JavaEE, el fichero de configuración puede ser sustituido casi totalmente por el uso de anotaciones, aunque dadas las limitaciones de tiempo y espacio de la sesión aquí usaremos el XML, por ser la forma más similar a Struts 1.x.

7.2. De Struts 1.x a Struts 2

Veamos de forma rápida cómo han cambiado en la versión 2 los principales componentes de una aplicación Struts: acciones, actionforms y taglibs.

7.2.1. Acciones

El código de una acción de Struts 1.x tiene fuertes dependencias del API de Struts y del API de servlets. Recordemos el primer ejemplo de acción que vimos en la sesión 1, donde se destacan en negrita las dependencias:

```
package acciones;

import javax.servlet.http.*;
import org.apache.struts.action.*;

public class AccionLogin extends Action
   public ActionForward execute(ActionMapping mapping, ActionForm
form,
                        HttpServletRequest request,
HttpServletResponse response)
                        throws Exception {
        boolean usuarioOK;
        //obtener login y password y autentificar al usuario
        //si es correcto, poner usuarioOK a 'true'
        ...
        //dirigirnos a la vista adecuada según el resultado
        if (usuarioOK)
           return mapping.findForward("OK");
        else
           return mapping.findForward("errorUsuario");
```

```
   }
}
```

Estas dependencias son una "molestia" a la hora de hacer pruebas unitarias. Para probar nuestra acción necesitamos tener Struts y el contenedor de servlets en marcha. Además dificultan la portabilidad de nuestro código a otro framework MVC distinto de Struts.

La tendencia actual en JavaEE es el uso de APIs "no invasivos", que no necesitan de dependencias explícitas en el código. Siguiendo esta filosofía, en Struts 2 una acción es una clase java convencional, que no es necesario que herede de ninguna clase especial:

```
package acciones;

public class AccionLogin {
        public String execute() {
          boolean usuarioOK;
          //obtener login y password y autentificar al usuario
          //si es correcto, poner usuarioOK a 'true'
          ...
          //dirigirnos a la vista adecuada según el resultado
          if (usuarioOK)
             return "OK";
          else
             return "errorUsuario";
        }
}
```

Nótese que las dependencias de APIs externos al código se han reducido a cero (luego veremos cómo obtener los parámetros HTTP y otra información de contexto). Así, nuestra acción se puede probar fácilmente y portarla a otro framework MVC es también mucho más sencillo.

Aviso:

Aunque no hay dependencias directas del API de Struts, el "formato" de nuestra acción no es totalmente libre. Por defecto, el método que ejecuta la acción debe llamarse `execute`, no tener parámetros y devolver un String.

Por supuesto, sigue siendo necesario mapear una determinada URL con la acción y asociar los resultados con la vista a mostrar. Esto se hace en el `struts.xml` (o de manera alternativa con anotaciones):

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
   <package name="default" extends="struts-default">
      <action name="login" class="acciones.AccionLogin">
         <result name="OK">/usuario.jsp</result>
         <result name="errorUsuario">/error.jsp</result>
```

```
      </action>
   </package>
</struts>
```

Por defecto, la acción se asocia con una URL igual a su `name` seguida del sufijo ".action". En este caso "login.action" sería la que dispara la acción.

Por conveniencia Struts 2 ofrece una clase llamada `ActionSupport` de la que pueden heredar nuestras acciones. Una de las ventajas de usar `ActionSupport` es disponer de una serie de constantes predefinidas con posibles resultados de la acción, en lugar de usar Strings arbitrarios como en el ejemplo anterior.

```
package acciones;

import com.opensymphony.xwork2.ActionSupport;

public class AccionLogin extends ActionSupport {

  @Override
  public String execute() throws Exception {
    boolean usuarioOK;
    //obtener login y password y autentificar al usuario
    //si es correcto, poner usuarioOK a 'true'
    ...
    //dirigirnos a la vista adecuada según el resultado
    if (usuarioOK)
          return SUCCESS;
    else
      return ERROR;
  }
}
```

7.2.2. El ocaso de los ActionForms

Los ActionForms son uno de los puntos más débiles de Struts 1.x, presentando una serie de problemas:

- Dependencia del API de Struts, ya que deben heredar de clases propias del framework
- Como los campos deben poder acomodar datos de tipo incorrecto, al final todos acaban siendo de tipo String. Esto implica que no podemos usar los ActionForms como objetos de negocio, llevando a:
  - La multiplicación del número de clases necesarias
  - La necesidad de convertir los valores al tipo apropiado
  - La necesidad de copiar los valores al objeto de negocio

Aunque librerías como BeanUtils alivian en parte los dos últimos problemas, es evidente que sería mejor que todo estuviera integrado en el propio framework antes que tener que usar herramientas adicionales.

Por todo ello, en Struts 2 se ha tomado la decisión de eliminar los ActionForms como tales. Por supuesto, las acciones seguirán necesitando acceso a beans Java para hacer su

trabajo, pero serán clases java convencionales sin ningún requisito especial. Veamos cómo podría obtener la acción de login los datos necesarios para comprobar usuario/contraseña. Antes que nada, necesitamos un bean para almacenarlos

```
package modelo;

public class Usuario {
        private String login;
        private String password;

        public String getLogin() {
                return login;
        }
        public void setLogin(String login) {
                this.login = login;
        }
        public String getPassword() {
                return password;
        }
        public void setPassword(String password) {
                this.password = password;
        }
}
```

Nótese que la clase anterior es un JavaBean convencional que no depende del API de Struts.

Ahora la acción recibirá el bean mediante *inyección de dependencias*. Esta es una filosofía de trabajo muy habitual actualmente en JavaEE en la que se asume que alguien externo (Struts en este caso) le pasará a nuestro código el/los objetos necesarios para hacer su trabajo. En nuestra acción estamos asumiendo que antes de llamar a `execute`, Struts habrá instanciado la variable `usuario` llamando al método `setUsuario`.

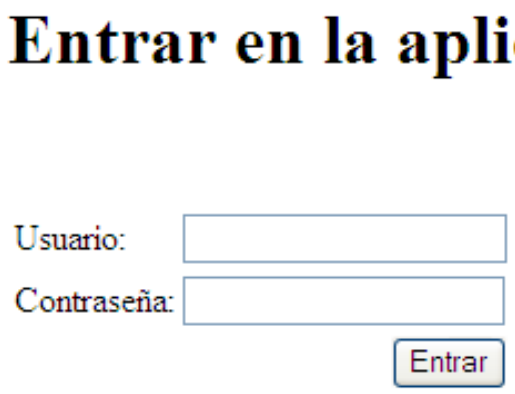
```
package acciones;

import modelo.Usuario;

public class LoginAccion {
        private Usuario usuario;

        public Usuario getUsuario() {
                return usuario;
        }

        public void setUsuario(Usuario usuario) {
                this.usuario = usuario;
        }

        public String execute() {
                String login = usuario.getLogin();
                ...
        }
}
```

Nótese la diferencia fundamental con Struts 1.x: en éste, el bean (ActionForm) se recibía como un parámetro del `execute`. En Struts 2 se recibe a través del *setter*.

Todo esto está muy bien, pero ¿de dónde salen los datos del objeto `Usuario`?. Resulta que si usamos las taglibs de Struts 2 para nuestros JSP, sus valores se rellenarán automáticamente, igual que pasaba con los ActionForms en Struts 1.x.

7.2.3. Taglibs

Al igual que Struts 1.x, la versión 2 incluye varias taglibs propias. Además de haberse mejorado, sobre todo se ha simplificado su uso. Por ejemplo, supongamos que queremos generar el siguiente formulario y asociar sus campos a un bean de tipo usuario. Además queremos mostrar los errores de validación, si los hay



formulario

Podemos hacerlo con el siguiente HTML

```
<%@taglib prefix="s2" uri="/struts-tags" %>
<html>
<head>
  <title>Ejemplo de taglib HTML en Struts 2</title>
</head>
<body>
  Entrar en la aplicación:<br/>
  <s2:form action="login">
    <s2:textfield label="usuario" name="usuario.login"/>
    <s2:password label="contraseña" name="usuario.password"/>
    <s2:submit value="login"/>
  </s2:form>
</body>
</html>
```

A partir del ejemplo, podemos ver que ha habido varios cambios con respecto a Struts 1.x:

- Algunas etiquetas han cambiado: por ejemplo, `textfield` era `text` en Struts 1.x

- Los campos de formulario generan también un rótulo con texto precediéndolos ("Usuario" y "Contraseña"). En Struts 1.x había que ponerlos manualmente. Tampoco hay que formatear explícitamente los campos para que aparezcan uno debajo del otro.
- Podemos asociar un campo a una propiedad de un bean sin más que poner `nombre_bean.nombre_propiedad`. Las taglibs de Struts 2 usan un lenguaje de expresiones similar al EL de JSP denominado OGNL.
- No es necesario poner etiquetas para mostrar los errores de forma explícita. Si activamos la validación (veremos posteriormente cómo hacerlo), al lado de cada campo aparecerá el mensaje de error correspondiente.

Como vemos, la cantidad de HTML introducida por las etiquetas de Struts 2 es sensiblemente superior a la de sus equivalentes en 1.x. Este HTML es configurable , ya que Struts 2 usa plantillas predefinidas de *freemarker* para generarlo, que podemos cambiar por otras o adaptar a nuestras necesidades.

7.2.4. Internacionalización

Al igual que las tags de Struts 1.x, las de Struts 2 soportan internacionalización. Los mensajes se sacan también de ficheros .properties. En nuestro ejemplo usaremos el fichero "global" de .properties, cuyo nombre se define en el archivo `struts.properties`, en la raíz del CLASSPATH. Recordar que en struts 1.x este fichero se definía en el XML de configuración con la etiqueta `message-resources`. Nuestro `struts.properties` contendría una línea con:

```
struts.custom.i18n.resources=mensajes
```

Definiendo por tanto `mensajes.properties`, en la raíz del CLASSPATH, como nuestro fichero de mensajes. Para aumentar la modularidad, Struts 2 permite definir ficheros de mensajes propios de una acción o globales solo a un `package` java (entre otras opciones), aunque no veremos aquí como definirlos.

Nuestro `mensajes_es.properties`, para idioma español, podría contener:

```
loginTitle=Entrar en la aplicación
login=Usuario
password=Contraseña
enter=Entrar
```

Habría que cambiar las tags de Struts para que hagan uso del fichero de mensajes

```
<%@taglib prefix="s2" uri="/struts-tags" %>
<html>
<head>
  <title>Ejemplo de taglib HTML en Struts 2</title>
</head>
<body>
  <s2:label key="loginTitle"/>
  <s2:form action="login">
```

```
    <s2:textfield key="login" name="usuario.login"/>
    <s2:password key="password" name="usuario.password"/>
    <s2:submit key="enter"/>
  </s2:form>
</body>
</html>
```

7.2.5. Validación

Struts 2 también permite validación declarativa, aunque ya no usa el *commons validator*. De este modo se puede usar una sintaxis más concisa y adaptada a las necesidades del framework.

La práctica recomendada en Struts 2 es definir la validación de cada acción en un XML separado. Por convención el fichero debe tener el mismo nombre que la clase que define la acción, con el sufijo `-validation`. Es decir, para la clase `LoginAccion` debe ser `LoginAccion-validation.xml`, en el mismo paquete java que la clase de la acción. Veamos un ejemplo, suponiendo que queremos que login sea obligatorio

```
<!DOCTYPE validators PUBLIC
"-//OpenSymphony Group//XWork Validator 1.0.2//EN"
"http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
    <field name="usuario.login">
        <field-validator type="requiredstring">
            <message>Login no puede estar vacío</message>
        </field-validator>
    </field>
</validators>
```

Como se ve, la sintaxis se simplifica bastante con respecto a Struts 1.x. Aunque en el ejemplo el mensaje de error está fijo en el XML, para simplificar, por supuesto se puede externalizar a un properties.

Para que nuestra acción soporte validación, debe implementar el interfaz `ValidationAware`. Dicho interfaz define varios métodos, para gestionar los errores producidos. Afortunadamente no es necesario implementarlos si nuestra acción hereda de `ActionSupport`.

Aviso:

Si la acción no hereda de ActionSupport (o implementa los métodos del interfaz ValidationAware), la validación declarativa no funcionará

En el XML de configuración un error de validación se asocia con un resultado "input" (por "herencia" de la palabra clave usada en Struts 1.x). Así, el `action` del struts.xml quedaría:

```
<action name="login"
```

```
class="es.ua.jtech.struts2.prueba.acciones.LoginAccion">
  <result name="ok">/usuario.jsp</result>
  <result name="input">/index.jsp</result>
</action>
```

En `ActionSupport` se define la constante `INPUT` con el valor `"input"`, de modo que en el código java de la acción haríamos un `return INPUT` si tuviéramos que validar alguna condición manualmente en el `execute()` de la acción.

7.3. Conceptos nuevos en Struts 2

Por lo visto en los apartados anteriores puede dar la impresión de que, aunque Struts se ha mejorado y simplificado su uso, sigue siendo el mismo framework "por dentro". Nada más lejos de la realidad, su arquitectura y funcionamiento interno han cambiado completamente. De ahí que haya conceptos totalmente nuevos que no tienen equivalencia en la versión 1. Veremos aquí muy brevemente algunos de ellos

7.3.1. Interceptores

Los interceptores son elementos que interceptan la petición antes de que se ejecute la acción de destino. De este modo se pueden realizar tareas sin tener que modificar el código de la acción. Podemos decir que desempeñan un papel muy similar al de los filtros de servlets pero con relación a las acciones. La configuración por defecto de Struts incluye una pila completa de interceptores activos para todas las peticiones. De hecho, la inyección de dependencias que veíamos en el ejemplo de acción es responsabilidad de un interceptor. Y también la validación de campos, entre otras muchas tareas.

Pese a su importancia, es raro que el desarrollador de Struts tenga que "vérselas" directamente con los interceptores. Como mucho, tendremos que hacer que nuestras acciones implementen algún interfaz o hereden de alguna clase para interactuar con el interceptor. De hecho, recordemos que para la validación debíamos implementar el interfaz `ValidationAware` (o heredar de `ActionSupport`).

7.3.2. OGNL y value stack

El *value stack*, como su nombre sugiere, es una pila donde se colocan objetos a los que podemos necesitar acceso en un momento dado. Struts coloca en esta pila, entre otros, la acción a ejecutar y los ámbitos de aplicación, sesión, petición,...

OGNL es un lenguaje de expresiones que se usa para referenciar elementos en esta pila. Su sintaxis es similar a la del EL de JSP, aunque bastante más potente. En una aplicación Struts, es en las tags propias del framework donde se usa habitualmente OGNL. Cuando en un ejemplo anterior poníamos la expresión OGNL `usuario.login`, la estábamos buscando en la pila de valores. Como hemos dicho, la acción se coloca automáticamente en esta pila. OGNL nos permite ir navegando por objetos relacionados entre sí (de hecho

significa *Object Graph Navigation Language*). De este modo podemos acceder a la propiedad `usuario` de la acción y dentro de ella a la propiedad `login`. Al igual que en EL, cuando ponemos `objeto.propiedad`, se asume que accedemos a ésta a través de métodos get/set.

8. Ejercicios de Struts 2

En las plantillas de la sesión se incluye un proyecto de tipo "hola mundo" hecho con Struts 2. Para probarlo, ejecutar el proyecto y acceder a la url `http://localhost:8080/testStruts2/hola.action`. . El funcionamiento es como sigue:

- La URL está asociada a la clase `es.ua.jtech.struts2.acciones.HolaMundo`, a través del `struts.xml` de la carpeta `resources` (atributos "name" y "class")
- La acción se ejecuta y devuelve un resultado "success", que de nuevo en el `struts.xml` está asociado a la página `hola.jsp`
- En dicha página, con la tag `property` de Struts 2 se muestra una propiedad de un bean. En este caso la propia acción se trata como un bean. Para acceder a la propiedad `mensaje` se llama al getter de la acción (la expresión OGNL busca en la acción, entre otros sitios).

8.1. Implementación de un caso de uso sencillo

Supongamos que queremos implementar una página web desde la que se pueden mandar SMS gratuitos. Vamos a implementar parte de la funcionalidad (evidentemente, no el envío real de mensajes).

8.1.1. Creación y mapeo de la acción (1 punto)

Crear la acción que se ocuparía del envío de mensajes en la clase java `es.ua.jtech.struts2.acciones.EnviarSMS`. Por el momento, la acción no hace nada salvo devolver "success". Conseguir que la acción se asocie con la URL `http://localhost:8080/testStruts2/EnviarSMS.action` y que el resultado "success" se asocie con la página "enviado.html" que simplemente mostrará un mensaje indicando "el sms se ha enviado correctamente".

Comprobar que todo funciona adecuadamente.

8.1.2. Uso de javabeans (1 punto)

- Crear un javabean `es.ua.jtech.struts2.beans.SMS` para almacenar los datos de un SMS: número de teléfono y texto.
- En la acción `EnviarSMS`, crear un campo de tipo SMS llamado `sms` con métodos get/set. En el `execute` de la acción hacer que por la consola se muestre texto con los datos del mensaje, algo como `"Enviando al nº 626123456 el texto 'Struts2 mola'"`.
- Crear una página index.jsp y en ella implementar un formulario para enviar un nuevo SMS con las tags propias de Struts2. Asociar los campos con las propiedades del bean

`sms` de la acción `EnviarSMS`.

Comprobar que al rellenar el formulario y enviarlo, la acción imprime correctamente los datos del SMS en la consola.

8.1.3. Validación (1 punto)

Hacer que se validen los datos del formulario. Los dos campos son obligatorios. Podéis también validar que el número de teléfono solo tiene dígitos, pero entonces necesitaréis consultar la documentación sobre validación de Struts 2 para ver la definición de los validadores.

Necesitaréis:

- Definir la validación en un fichero `EnviarSMS-validation.xml` junto a la clase `EnviarSMS`.
- En el struts.xml, asociar el resultado de "input" a la página que contiene el formulario, para que si falla la validación se vuelvan a mostrar los datos