

# Struts2.

---

06 de Noviembre de 2009

## Índice

---

1.Introducción .....	5
Historia de Struts.....	5
Por qué usar Struts 2 .....	6
Struts 1 vs Struts 2.....	7
2.  Arquitectura .....	10
Ciclo de vida de una petición .....	10
Arquitectura MVC .....	11
Controlador .....	11
Modelo .....	11
Vista .....	12
Gestor de Excepciones.....	12
Interceptores .....	12
Gestión de Configuración.....	12
3.  Instalación de Struts 2.....	15
Descargar Struts 2.....	15
Descargar Tomcat .....	16
4.  Aplicación Hello World .....	19
Desarrollo de la Vista .....	19
Desarrollo del Action.....	20
El Controler Configuration File .....	21
¿Cómo funciona la aplicación?.....	21
5.  Fichero de configuración struts.xml .....	23
Estructura del fichero struts.xml .....	24
struts.xml en detalle.....	25
El Tag Package .....	26
El tag Include.....	26
El Tag Bean .....	27
El tag Constant.....	27
6.  Aplicación LOGIN con Struts 2.....	29
Desarrollo del formulario login .....	29
Desarrollo del action. ....	31

Configuración del action .....	33
Validación en el servidor .....	34
Validación en el cliente .....	36
Validación utilizando anotaciones .....	38
7. Login/Logout usando la sesión .....	41
Fichero de configuración .....	41
Desarrollo de los action .....	41
Desarrollo del formulario .....	42
8. Struts 2 Tags .....	46
Tags genéricos .....	46
Control Tags-If / Else If / Else .....	47
Tag Append .....	48
Tag Generator .....	49
Tag iterator .....	50
Tag Merge .....	52
Tag subset .....	53
Tag action .....	55
Tag Bean .....	56
Tag Date .....	57
Tag Include .....	59
Tag Param .....	60
Tag set .....	60
Tag Text .....	60
Tag property .....	61
Struts UI tags .....	61
Tag Autocompleter .....	62
Tag checkbox .....	65
Tag checkboxlist .....	65
Tag combobox .....	67
Tag Datapicker .....	68
Tag doubleselect .....	70
Tag file .....	71
Tag form .....	72
Tag Label .....	73

Tag Optiontransfersselect .....	73
Tag optgroup .....	75
Tag password .....	76
Tag radio .....	77
Tag reset.....	78
Tag select.....	78
Tag submit.....	79
Tag textarea .....	79
Tag textfield .....	80
Tag Updownselect.....	80
Tags Actionerror y Actionmessage .....	81
Tag fielderror.....	81

## 1. Introducción

---

Struts 2 es, como el nombre sugiere, la nueva versión del popular framework de desarrollo del Java Apache Struts. Sin embargo, contrariamente a lo que cabría esperar, el código de ambos tiene poco que ver, dado que Struts 2 no se basa en el código de Struts 1, sino en el de otro framework de desarrollo web en Java llamado WebWork, un framework que el creador de Struts consideraba superior a Struts 1 en varios aspectos.

Struts 2 está basado en el patrón MVC (Modelo-Vista-Controlador), una arquitectura que busca reducir el acoplamiento dividiendo las responsabilidades en 3 capas claramente diferenciadas:

- El modelo, que hace referencia a los datos que maneja la aplicación y las reglas de negocio que operan sobre ellos y que se traducen en Struts 2 en las acciones.
- La vista, encargada de generar la interfaz con la que la aplicación interacciona con el usuario. En Struts 2 equivale a los resultados.
- El controlador, que comunica la vista y el modelo respondiendo a eventos generados por el usuario en la vista, invocando cambios en el modelo, y devolviendo a la vista la información del modelo necesaria para que pueda generar la respuesta adecuada para el usuario. El controlador se implementa en Struts 2 mediante el filtro FilterDispatcher.

Struts2 está pensado para facilitar las tareas propias del ciclo de vida del software, incluyendo la construcción, desarrollo y mantenimiento de la solución. Es fácilmente extensible, ya que todos los elementos están basados en interfaces y clases bases.

La plataforma básica que requiere Struts 2 requiere las versiones Servlet API 2.4, JSP API 2.0 y Java 5.

Otra característica nueva que incluye Struts2 es un conjunto de librerías desarrolladas para facilitar la creación de aplicaciones web dinámicas, facilitando la creación de elementos de cliente que permitan mostrar información, validarla, localizarla de forma sencilla y sin que impacte en los tiempos de desarrollo, haciendo los desarrollos más sencillos de mantener.

### Historia de Struts

Struts2 es un framework de código abierto usado para desarrollar aplicaciones web. Fue originalmente desarrollado por Craig R. McClanahan, y en el año 2002 pasa a pertenecer a la Apache Software Foundation.

Struts provee un framework que facilita el desarrollo organizando los elementos J2EE: JSP, clases, HTMLs, etc, consiguiendo convertirse en un referente de los demás frameworks, integrándose de una forma efectiva con las tecnologías Java disponibles en el momento.

Struts se define como un framework basado en la arquitectura MVC, la cual articula los desarrollos definiendo claramente los roles de los elementos que se desarrollan. Así encontramos el modelo que gestiona la lógica de negocios y el acceso a datos. La vista que es el responsable de la percepción final del usuario. Por último el Controlador que es responsable de los aspectos navegacionales y de coordinación.

Los motivos por los que surge Struts2 hay que buscarlos en limitaciones de Struts, ya que la versión 1 plantea restricciones en la forma de desarrollar, la aparición de nuevos frameworks que aportan nuevos conceptos y puntos de vistas, y el surgimiento de nuevas tecnologías como Ajax.

El desarrollo de Struts 2 está pensado para mejorar Struts aportando flexibilidad, sencillez y robustez a los elementos que se emplean en el desarrollo, facilitar el desarrollo de nuevas aplicaciones y el mantenimiento de aquellas que se desarrollen. Por ello se hace una nueva arquitectura, una nueva API, nuevos Tags, etc., a partir de un framework existente WebWork, pero manteniendo todo lo bueno que tenía Struts. El resultado es un framework sencillo y fiable.

## Por qué usar Struts 2

La nueva versión de Struts 2 es una mezcla entre los Action de Struts y Webwork, lo cual le confiere de las siguientes características:

1. **Diseño Simplificado:** Uno de los principales problemas que tenía el framework Struts 1 era el uso de clases abstractas, cosa que cambia en la versión 2 en la cual se hace uso de Interfaces. Esto le provee una mayor facilidad para la extensión y la adaptación, ya que los interfaces son más fáciles de adaptar que las clases abstractas.  
Otro cambio es que se busca que las clases que sean lo más simple posible con lo que los actions se convierten en POJOs, elementos que además estarán poco acoplados. Los POJOs son clases que cuentan con getter y setter para poder recibir valores desde páginas, y cuentan con algunos métodos en los cuáles pondremos la lógica de negocio.
2. **Simplificación de los actions:** Como hemos dicho los actions son POJOs. Cualquier clase java con un método execute puede actuar como un Action. Así no se hace necesario implementar ningún interfaz. Además se introduce la Inversión de control en la gestión de los actions.
3. **Desaparecen los ActionForms:** se ven reemplazados por simples Java Beans que son usados para leer las propiedades directamente. Lo usual es que el propio Action actúe de JavaBean, con lo que se facilita el

desarrollo. Además de ha mejorado la lectura de parámetros con el objetivo de no tener únicamente propiedades de tipo String.

4. **Test simplificados:** como los actions engloban la lógica de negocio y los JavaBeans, es más sencillo hacer test unitarios.
5. **Fácil selección de opciones por defecto:** casi todos los elementos de configuración tienen definidos un valor por defecto que se puede parametrizar, lo que facilita la elección de acciones por defecto.
6. **Results mejorados:** a diferencia de los Action Forwards, los results de Struts 2 son más flexibles a la hora de poder definir múltiples elementos de la vista. Además desaparece la complejidad de utilizar ActionForwards, ya que se sustituye por la devolución de Strings.
7. **Mejoras en Tags:** struts 2 permite añadir capacidades utilizando tags que permite hacer páginas consistentes sin añadir código. Los tags presentan más opciones, están orientados a los results y pueden ser cambiados de forma sencilla. Además se añaden tags de marcado (markup) los cuáles son editables usando templates FreeMarker. Esto significa que podemos hacer que un mismo tag se comporte de forma diferente sin tener que hacer ninguna tarea de programación.
8. **Se introducen anotaciones:** las aplicaciones en struts 2 puede usar anotaciones como alternativa a XML y configuraciones basadas en properties.
9. **Arranque rápido:** muchos cambios se pueden hacer sin necesidad de reiniciar el contenedor web.
10. **Parametrización del controlador:** Struts 1 permitía parametrizar el procesador de peticiones a nivel de módulo. Struts 2 lo permite a nivel de action.
11. **Fácil integración con Spring.**
12. **Fácilidad para añadir plugins.**
13. **Soporte de Ajax:** se incluye un theme AJAX que nos permite hacer aplicaciones interactivas de forma más sencilla.

## Struts 1 vs Struts 2

En esta sección vamos a comparar las diferencias entre las versiones 1 y 2 de struts. En líneas generales podemos afirmar que Struts 2 es más simple y eficiente que Struts 1. No sólo explota mejor las potencialidades de J2EE, si no que también hace más fácil el desarrollo de nuevas aplicaciones y la integración con otras tecnologías y herramientas.

Algunas de las mejoras de la nueva versión son:

- **Dependencia del Servlet.**  
En la versión 1 de Struts hay una gran dependencia con el API Servlet, cosa que se pone de manifiesto en que los parámetros del método execute necesitan los objetos HttpServletRequest y HttpServletResponse.

En Struts1 se ha definido así para posibilitar el acceso a la sesión, al contexto del servlet, etc.

En Struts2 los actions están desacoplados del contenedor, siendo el Servlet Controller el responsable de realizar los set de parámetros. Ésta característica hace que sea sencillo el testeo de los actions de forma unitaria, ya que no se depende del contexto del servlet.

Lo anterior también ocurre con la sesión, la cual se establece en Struts2 vía el método `setSession`, y no vía el objeto `Request`.

- **Clase Action.**

El gran cambio que se ha producido en el diseño de los Actions es que en vez de extender clases abstractas como se hacía en la versión 1, se hace utilizando interfaces, lo cual salva algunas de las restricciones en cuanto al modelado del software que surgían con la versión anterior. Además también existe la opción de tener Actions que directamente no implementa ninguna clase, ya que se usa el Api de introspección para determinar el método a ejecutar.

Además el Action puede comportarse como `JavaBean` para leer la información desde las pantallas, haciendo más fácil realizar las acciones que sean requeridas.

- **Validaciones.**

Tanto Struts1 como Struts2 permiten la validación manual vía el método `validate`. En el caso de Struts1 dicho método está en la clase `ActionForm`, o bien hace la validación vía la extensión del `Validator`.

Struts2 permite la validación manual vía el método `validate` que se encuentra en el Action, o bien puede realizar validaciones usando el framework `XWork`. Dicha tecnología permite hacer validaciones encadenadas usando reglas definidas por el usuario.

- **Modelo de hebras.**

En Struts1 los Actions son sincronizados empleando el patrón sigleto. Esto hace que sólo una instancia de clase va a responder todas las peticiones que le lleguen a ese action.

En el caso de Struts2 se instancia un objeto para cada petición, con lo que podemos tener que el mismo código lo están ejecutando más de una hebra.

- **Recuperación de datos de entrada.**

Para la captura de los datos de entrada Struts1 utiliza clases que extienden de la clase genérica `ActionForm`. Esto hace que un `java` vean no pueda ser utilizado en Struts1, con lo cual los desarrolladores han de replicar el código del Bean en el `ActionForm` correspondiente.

En Struts2 se emplean propiedades del Action con sus sets y gets para poder recuperar la información. De esta forma si tenemos un `JavaBean`, sólo tenemos que ponerlo en el Action con sus correspondientes gets y sets para poder utilizarlos en la pantalla. Como se puede ver de esta forma no estamos replicando código.



Además en struts2 incorpora un conjunto de TLDs que permiten la plena integración de los Beans en las pantallas utilizando etiquetas estandarizadas que se pueden usar de forma conjunta con los estándares de J2EE.

- **Tag Libraries y Expresiones de Lenguaje para JSP.**

Struts1 integra JSTL, con lo que tenemos la posibilidad de utilizar JSTL-EL (lenguaje de expresiones de JSTL).

Struts2 incluye no sólo JSTL, sino también un conjunto de librerías que permiten una integración efectiva con las propiedades de los actions. Además las etiquetas permiten múltiples configuraciones utilizando los themes. Éstos permiten tener diferentes resultados de la misma etiqueta gracias al lenguaje OGNL (Object Graph Notation Language).

- **Uso de valores en las vista.**

Para trabajar con las vistas Struts1 usa el mecanismo estándar de JSP para vincular objetos a la propiedad en el contexto de página. Las jsp que requieran acceder a otros contextos tendrán que hacerlo de forma programativa.

Sin embargo Struts2 utiliza una tecnología "ValueStack" que permite recuperar los valores con independencia del contexto (sesión, aplicación, request) en el cual esté guardado el valor. Así las vistas son más fáciles de reusar.

- **Control de la ejecución de los actions.**

Struts1 permite separar el procesador de peticiones que gestiona el ciclo de vida de los actions para cada módulo, es decir, todos los actions dentro de un módulo tienen el mismo ciclo de vida. En struts2 se puede parametrizar a nivel de action, con una granularidad mayor.

## 2. Arquitectura

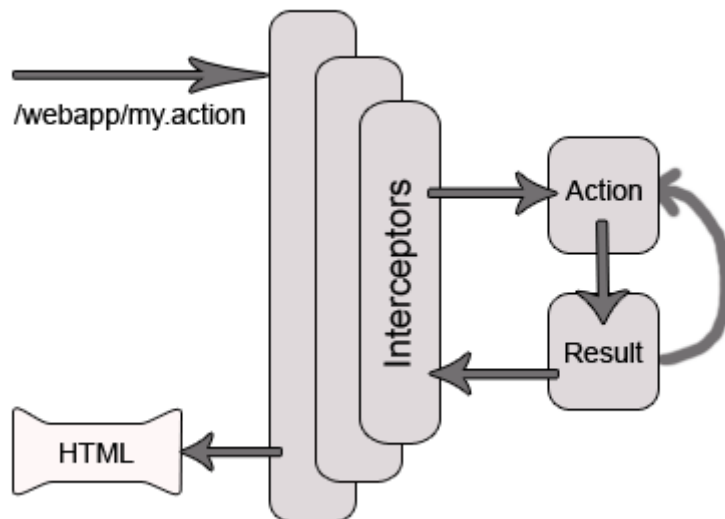
---

En esta sección se va a explicar cuál es la arquitectura del framework Struts2.

Struts 2 está concebido para ser un framework extensible y elegante para los desarrolladores de aplicaciones J2EE. Como primer paso se va a ver cuál es el ciclo de vida de una petición en una aplicación desarrollada bajo esta arquitectura:

### Ciclo de vida de una petición

1. **Un usuario envía una petición:** Un usuario realiza la petición de un recurso dentro del servidor.
2. **El elemento FilterDispatcher determina la acción que deberá responder:** El framework dispone de los elementos requeridos para que el dispatcher sea capaz de determinar qué action es el responsable de recibir la petición y procesarla. Para ello se apoya en el framework para la publicación del recurso, y para su ejecución.
3. **Se aplican los interceptores definidos:** Existen diferentes interceptores que se pueden configurar para que ejecuten diferentes funcionalidades como [workflows](#), validaciones, upload de ficheros, etc.
4. **Se ejecuta el Action:** Tras la ejecución de los diferentes interceptores el método específico del Action es ejecutado, realizándose aquellas operaciones y acciones que se hayan definido. El action termina devolviendo un resultado el cual se utiliza para determinar la página a devolver.
5. **Se renderiza la salida:** Tras la ejecución del action se determina cuál es la página que se devuelve y se ejecutan el forward a dicha página.
6. **Se devuelve la petición.:** Para realizar la devolución se ejecutan los interceptores que correspondan y se procede a devolver la petición al cliente. De esta forma es posible añadir lógica externa a los servidores también en la devolución.
7. **Se muestra el resultado al cliente final:** Finalmente el control es devuelto al cliente quien podrá visualizar el resultado en su navegador.



## Arquitectura MVC

Struts 2 está basado en el patrón MVC (Modelo-Vista-Controlador), una arquitectura que busca reducir el acoplamiento dividiendo las responsabilidades en 3 capas claramente diferenciadas el modelo, la vista y el controlador. Cada uno de éstos elementos presenta unas características que a continuación reseñamos. Adicionalmente se presentan algunos elementos que tienen un papel fundamental dentro de la arquitectura y de la configuración de struts.

### Controlador

Es el responsable de recibir las peticiones de los clientes y es el responsable de:

1. Procesar las peticiones, determinando qué clase es la que tiene que responder.
2. Modificar el modelo en función de los parámetros que recibe ejecutando la lógica de negocio que el desarrollador haya definido.
3. Redirigir a la vista en función de la lógica de negocio.

Como no podía ser de otra forma, el controlador permite el uso de tecnologías estándares tales como Java Filters, Java Beans, ResourceBundles, XML, etc.

### Modelo

Es la parte responsable de la gestión de la información. En este punto se suelen incluir aquellas clases, herramientas, librerías, etc., que permiten el acceso a los datos, así como el modelado de los mismos.

Algunas de las tecnologías que se suelen emplear son JDBC, EJB, Hibernate o JPA, entre otros.

## Vista

Es la responsable de la percepción que tienen los usuarios finales de la aplicación. Se incluyen las páginas Html, JS, etc, y en general todos los elementos de visualización de la aplicación. Se nutre de la información que el controlador ha captado del modelo para pintar las páginas finales.

Algunas de las tecnologías que se emplean son Html, JSP, XSLT, PDF, templates, etc.

Si nos abstraemos de la arquitectura MVC podemos encontrar que la arquitectura de Struts 2 presenta los siguientes elementos que debemos conocer para nuestras aplicaciones:

## Gestor de Excepciones

Éste elemento existía en Struts y está pensado para definir procedimientos de gestión de excepciones tanto locales como globales, de tal manera que cuando ocurra un error podamos definir qué se ha de hacer, redirigiendo a páginas genéricas de error, etc.

## Interceptores

Los interceptores son clases que se emplean para especificar el ciclo de vida de una petición, y están pensadas para añadir funcionalidad extra a las acciones. Podríamos decir que un interceptor es parecido a un filtro de J2EE salvo que su uso es más localizado y se puede hacer depender de actions concretos y no que se ejecute de forma global. Para configurarlos se hace de forma declarativa.

Los interceptores se suelen usar para realizar validaciones, workflows, etc..., y cuando se definen sobre un action, se ejecutan como los filtros, es decir, de forma consecutiva como una cadena.

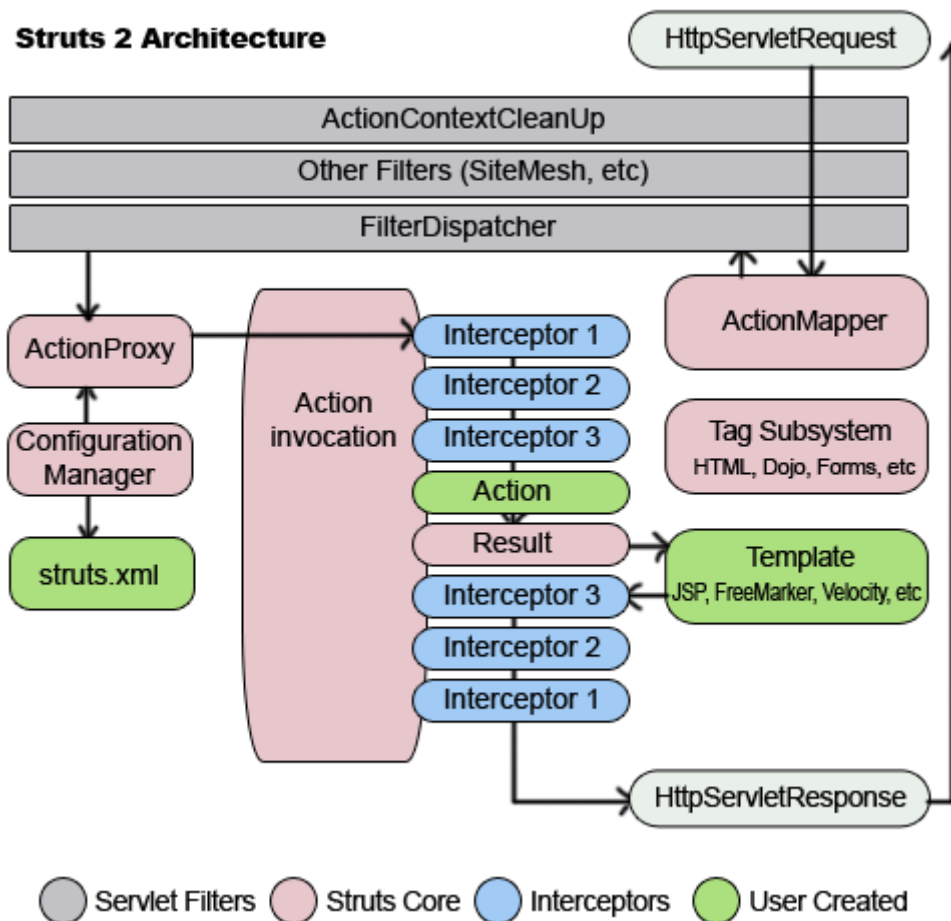
## Gestión de Configuración

El gestor de la configuración es el elemento encargado de leer el fichero de control (struts.xml), analizarlo y en el caso de que sea correcto desplegar aquellos elementos que se referencian.

Para aclarar ideas y mostrar cómo es la arquitectura de Struts 2 se añade un diagrama en el cual se puede ver de forma gráfica qué elementos son los que intervienen dentro del procesamiento de una petición HTTP, distinguiéndose cuáles son elementos de estándares de J2EE, cuáles son elementos propios de Struts 2 y cuáles son creados por el usuario.

Por orden los elementos que intervienen son:

1. Una petición llega a un servlet (HttpServletRequest) el cual es la parte del controlador que va a recibir la petición.
2. El servlet delega en el ActionMapper para determinar cuál es la acción que se va a ejecutar.
3. A continuación se ejecuta la "cadena de filtros", entre los que destacamos:
  - a. Action Context Clean Up Filter. Éste filtro se emplea cuando hay que realizar una integración con otras tecnologías (compartición de cookies,etc.)
  - b. FilterDispatch. Éste es encargado de determinar si la resolución de la petición requiere la ejecución o no de un action. Hemos de tener en cuenta que existen redirecciones u otros que no necesitan de un action que las implemente. En el caso de que haya que usar un action, se delega en el Action Proxy.
4. Action Proxy: se ejecuta cuando la petición requiere la ejecución de un action. Lo que hace es utilizar el Gestor de Configuraciones para crear ActionInvocation. Éste es una clase que implementa el patrón Command, el cuál se ejecuta., el cu
5. Action Invocation: se lanza tras leer el fichero de configuración, y por tanto es conocedor de qué clase se debe ejecutar, así como de los interceptores que intervienen. Por ello es responsable de ejecutar cada uno de los interceptores, y luego ejecutará el Action correspondiente. Éste Action contendrá la lógica de negocio que se debe ejecutar.  
Como resultado de la ejecución del Action se obtendrá un Result, el cual se emplea para determinar cuál de los elementos de la vista se deben llamar.
6. Se ejecuta el elemento que debe configurar la vista y el resultado se devuelve deshaciendo el camino hasta el cliente.



Con esta introducción se han visto los elementos básicos que se emplean en la arquitectura y se ha hecho un repaso de cómo se procesa una petición.

En el siguiente punto vamos a explicar cómo se instala Struts 2, para a continuación a base de ejemplos mostrar en qué consiste realizar aplicaciones utilizando el framework Struts2.

## 3. Instalación de Struts 2

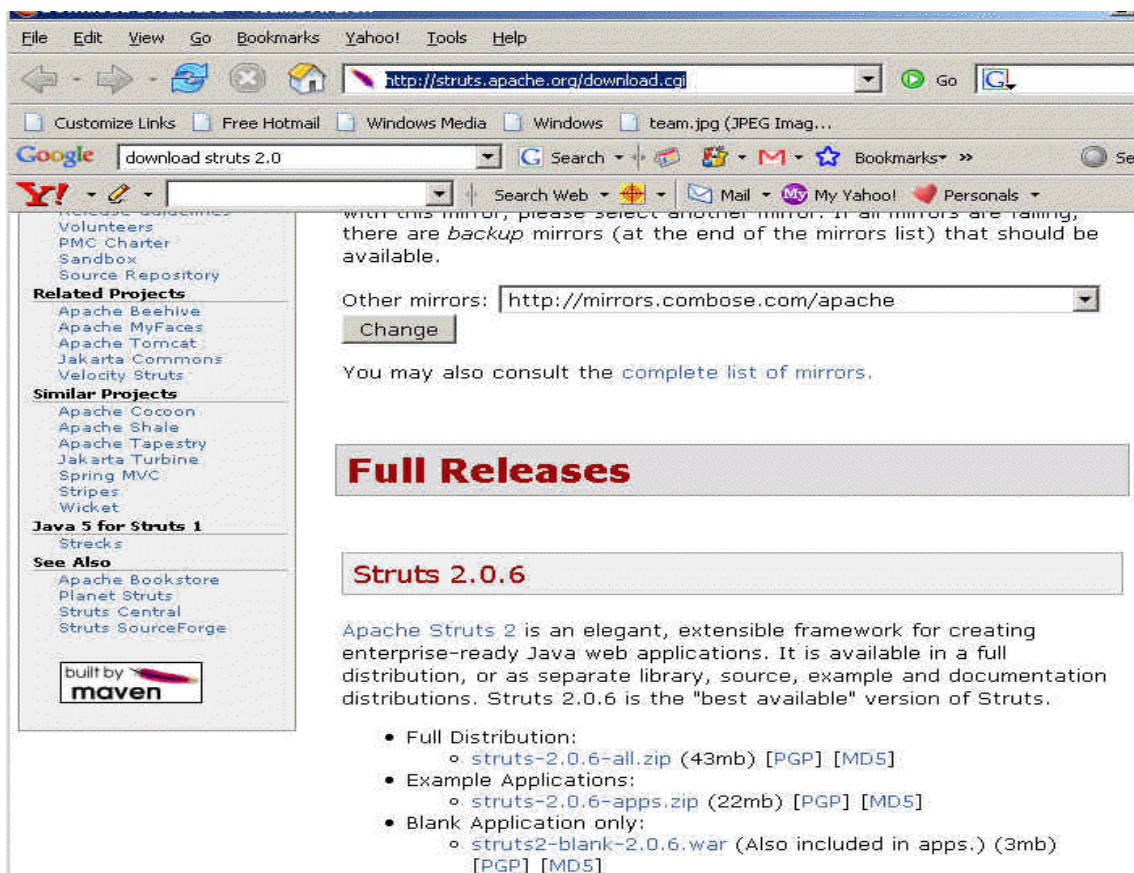
En esta sección vamos a mostrar el proceso de descarga e instalación de la última versión de Struts 2. Para ello también vamos a mostrar cómo instalar Tomcat, ya que requeriremos de un servidor Web para poder montar la aplicación.

El primer paso será descargar tomcat y configurarlo como servidor de desarrollo.

A continuación, descargaremos Struts 2 e instalaremos la web application struts-blank, la cual contiene la instalación mínima de Struts 2.

### Descargar Struts 2

Para descargar Struts 2 tenemos que dirigirnos a las páginas del proyecto Apache, concretamente a la dirección <http://struts.apache.org>. Dentro del site encontraremos un enlace con el texto "Download", cuya dirección es <http://struts.apache.org/download.cgi>. En la página resultante tendremos la opción de bajarnos la versión de Struts 2 que requerimos tal y como se puede ver en la imagen siguiente:



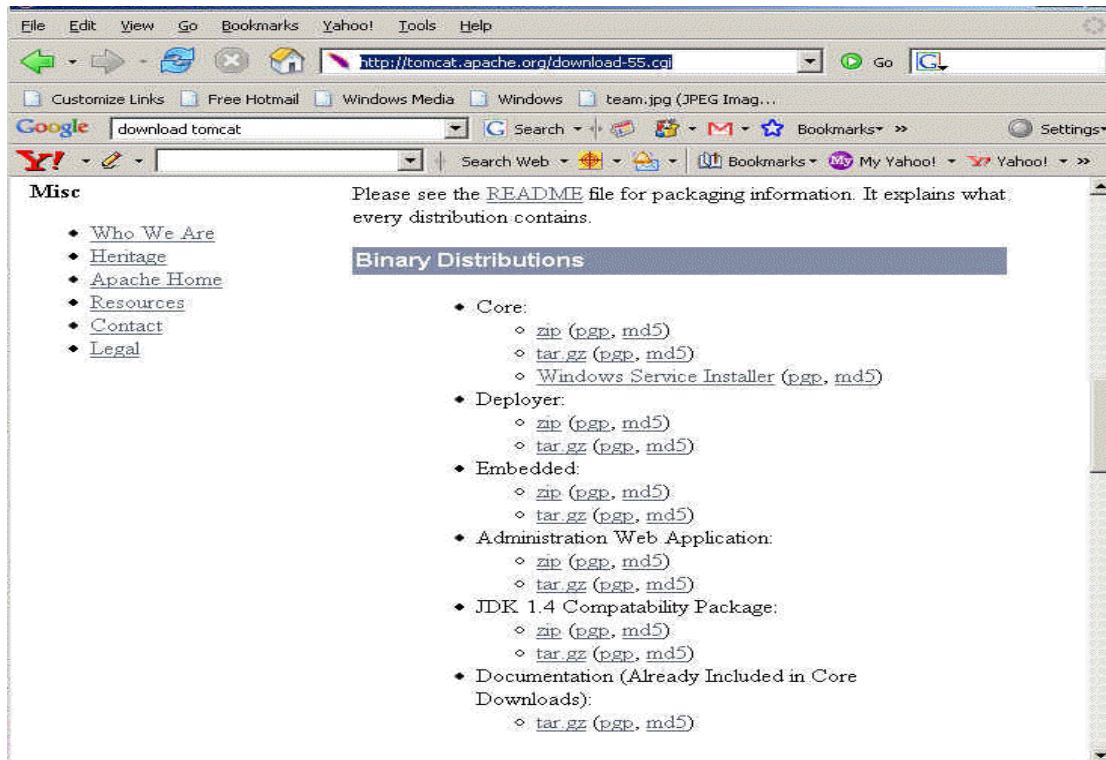
Además de tener disponible la descarga del product, en el site <http://struts.apache.org> disponemos de multiple documentación sobre el producto, así como información de otros proyectos de la Fundación Apache, foros para desarrolladores, etc.

El siguiente punto sera descargarnos Tomcat.

## Descargar Tomcat

Para poder instalar Tomcat el único requisito que vamos a necesitar es tener una version de Java, preferentemente a partir de la 1.5

Para realizar la instalación de tomcat lo primero que tenemos que hacer es descargárnosla. En la url <http://tomcat.apache.org/download-55.cgi> tenemos disponible una instalación de la versión 5.5. Para bajárnosla elegiremos **apache-tomcat-5.5.20.zip** tal y como se puede ver en la imagen adjunta.

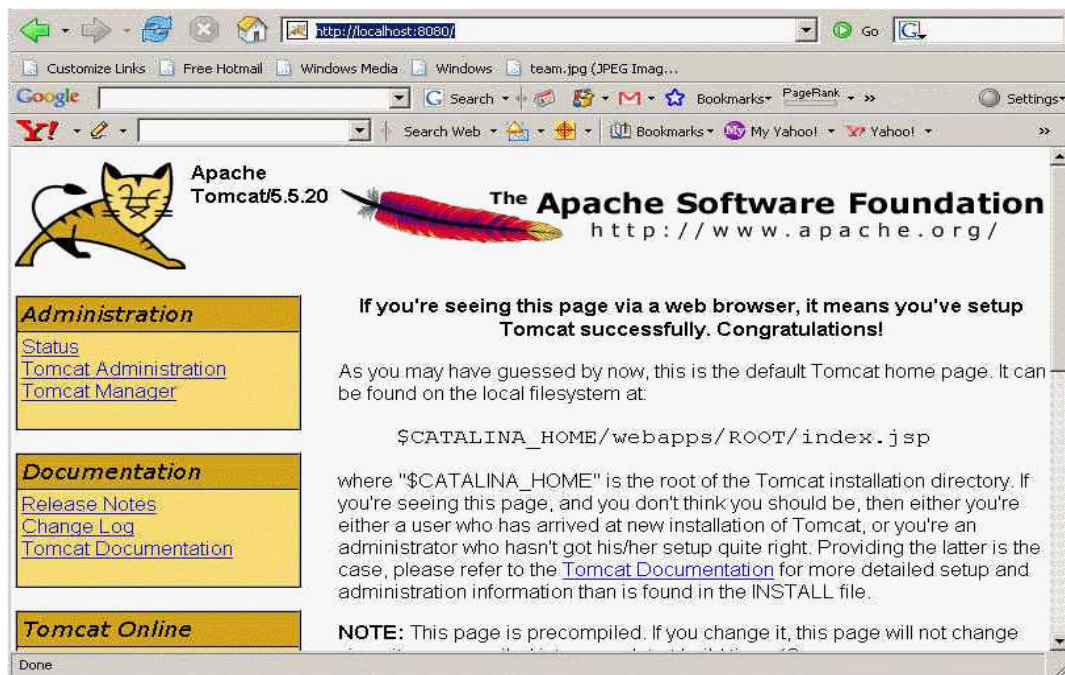


A continuación extraemos el fichero que descargamos en c:\ y ejecutamos C:\apache-tomcat-5.5.20\bin\startup.bat. Éste comando es el encargado de arrancar tomcat. Si la instalación ha funcionado veremos algo como lo siguiente:



```
Feb 25, 2007 11:42:23 PM org.apache.coyote.http11.Http11BaseProtocol
start
INFO: Starting Coyote HTTP/1.1 on http-8080
Feb 25, 2007 11:42:24 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Feb 25, 2007 11:42:24 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/141 config=null
```

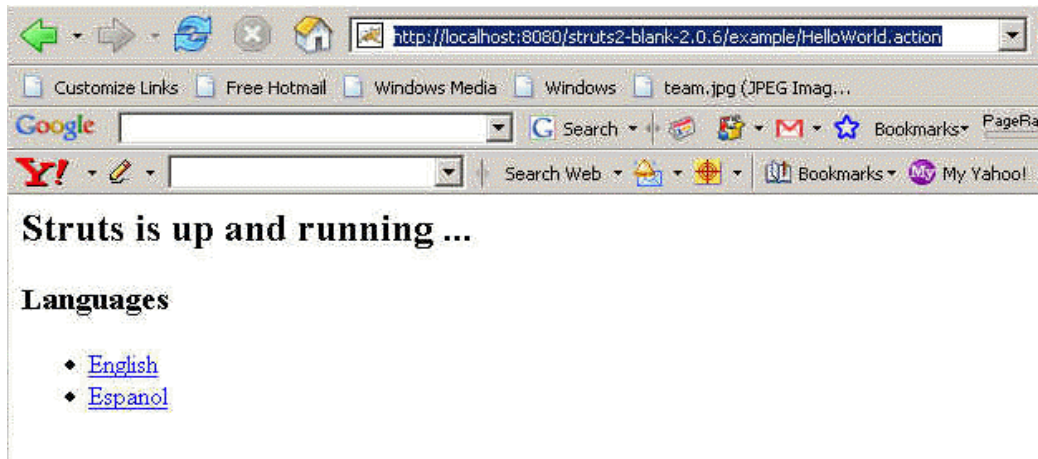
Para comprobar que se ha instalado correctamente abrimos un navegador y escribimos <http://localhost:8080/>, y si todo está bien obtenemos la página de bienvenida tal y como se muestra a continuación.



Por otro lado, extraemos el fichero de struts descargado struts-2.0.6-all.zip en un directorio de nuestra elección. Para instalar la blank application, copia "struts2-blank-2.0.6.war" del "<directorio elegido>\struts-2.0.6-all\struts-2.0.6\apps" en el directorio de webapps de tomcat (c:\apache-tomcat-5.5.20\webapps). De este modo, Tomcat hará automáticamente un deploy de la aplicación, como se ve a continuación:

```
Feb 25, 2007 11:42:24 PM org.apache.catalina.storeconfig.StoreLoader load
INFO: Find registry server-registry.xml at classpath resource
Feb 25, 2007 11:42:24 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 6672 ms
Feb 25, 2007 11:52:55 PM org.apache.catalina.startup.HostConfig
deployWAR
INFO: Deploying web application archive struts2-blank-2.0.6.war
```

Para probar que struts está correctamente montado pondremos en un navegador <http://localhost:8080/struts2-blank-2.0.6>, obteniéndose:



Otras aplicaciones que se pueden probar en esta distribución de Struts2 son las siguientes:

- struts2-mailreader-2.0.6.war
- struts2-portlet-2.0.6.war
- struts2-showcase-2.0.6.war

Una vez copiados estos archivos en el directorio de webapps en tomcat, serán automáticamente desplegadas y estarán listas para ser probadas.

## 4. Aplicación Hello World

---

Como es tradicional cuando se explica un nuevo entorno o lenguaje de programación vamos a ver cómo se crearía la aplicación Hello World, cosa que nos permitirá ir introduciendo los conceptos de Struts2.

Se asume que los lectores utilizarán algún IDE para crear la webapp, por lo que nos centraremos en la creación de los elementos propios de Struts, despreocupándonos de detalles de deployment. De todas formas recomendamos utilizar Eclipse o Netbeans para este fin.

Vamos a asumir que la webapp que desarrollamos se llamará struts2tutorial, con lo que su url será <http://localhost:8080/struts2tutorial>

Nuestra aplicación consistirá en una página la cuál generará un mensaje de bienvenida, con la fecha y hora del servidor.

El resultado es como sigue:



Los elementos que se van a desarrollar son:

### Desarrollo de la Vista

Vamos a crear la página HelloWorld.jsp en el directorio pages, que es la jsp que nos muestra el mensaje de bienvenida. Esta página tendrá que tener ser dinámica para poder mostrar la hora del servidor.

A continuación podremos ver el código:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Struts 2 Hello World Application!</title>
</head>
<body>
<h2><s:property value="message" /></h2>
<p>Current date and time is: <b><s:property value="currentTime" /></b>
</body>
</html>
```

Dentro de esta jsp podemos destacar:

- La línea `<%@ taglib prefix="s" uri="/struts-tags" %>` incluye una librería de tags llamada struts-tags, y nos permite referenciarla utilizando el prefijo `s:`. Esta librería es una específica que se ofrece Struts2 en la distribución.
- Los tags `<s:property value="message" />` y `<s:property value="currentTime" />` llaman a los métodos `getMessage()` y `getCurrentTime()` respectivamente del action `Struts2HelloWorld` action class y los introducirá en la página. Como podemos ver la inclusion de propiedades se hace utilizando etiquetas de la librería struts-tag.

## Desarrollo del Action

Vamos a crear el fichero `Struts2HelloWorld.java`, que sera el action que de la aplicación con el código que se muestra a continuación:

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import java.util.Date;

public class Struts2HelloWorld extends ActionSupport {

    public static final String MESSAGE = "Struts 2 Hello World Tutorial!";

    public String execute() throws Exception {
        setMessage(MESSAGE);
        return SUCCESS;
    }

    private String message;

    public void setMessage(String message){
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public String getCurrentTime(){
        return new Date().toString();
    }
}
```

Como podemos ver esta clase define dos getters, uno llamado message y otro currentTime, así como un método llamado execute.

## El Controler Configuration File

Struts 2 usa el fichero struts.xml para configurar la aplicación. Éste fichero se ubicará en el directorio classes del WEN-INF y tundra el siguiente contenido:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
  <constant name="struts.enable.DynamicMethodInvocation" value="false" />
  <constant name="struts.devMode" value="true" />
  <package name="sample" namespace="/" extends="struts-default">
    <action name="HelloWorld" class="sample.Struts2HelloWorld">
      <result>/pages/HelloWorld.jsp</result>
    </action>
  </package>
</struts>
```

Para poder probar la aplicación tenemos que llamar a la siguiente ruta: <http://localhost:8080/struts2tutorial/HelloWorld.action>.

## ¿Cómo funciona la aplicación?

A continuación hacemos un breve resumen de cómo funciona la aplicación "Hello World":

El navegador envía una petición al servidor web mediante la dirección: <http://localhost:8080/struts2tutorial/HelloWorld.action>.

A continuación se producen los siguientes pasos en el servidor:

1. El servidor intenta buscar el recurso "HelloWorld.action". Struts 2 configura el web.xml de tal forma que envía las peticiones al filtro org.apache.struts2.dispatcher.FilterDispatcher. El fichero de configuración web.xml contiene lo siguiente:

```
<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>
```

2. Entonces, el framework busca el action "HelloWorld" utilizando el fichero struts.xml. Al encontrar una definición de action que corresponde con el patron de la url, crea una instancia de la clase correspondiente que aparece en el attribute class.  
Utilizando el API de introspección realiza la llamada al método *execute* del objeto, que en este caso es del tipo Struts2HelloWorld.
3. A continuación el método *execute* fija el mensaje y devuelve SUCCESS.
4. El framework determina qué página se carga en el caso de que SUCCESS sea devuelto. Para ello identifica que si se devuelve SUCCESS buscará en el xml la entidad de la etiqueta action llamada *result* que no tenga ningún name. En nuestro caso, el servidor carga la página HelloWorld.jsp y devuelve la salida.

En Struts2, los actions en el método *execute* pueden devolver *SUCCESS*, *ERROR* o *INPUT* de forma estándar. También pueden devolver cualquier cadena de caracteres que queramos ya que el método *execute* devuelve un String.

El mecanismo que emplea para determinar la salida depende del número de entidades *<result>* que tenga la entidad *<action>* en el *struts.xml*.

De forma general SUCCESS identifica la salida que no tiene name; error identifica como salida el contenido de la etiqueta *<result name="error">* e INPUT a la que tiene como formato *<result name="input">*. Ésta última sera empleada por el framework de validación para determina a donde tiene que redirigir.

## 5. Fichero de configuración struts.xml

---

En esta sección se analizará el fichero *struts.xml* y su utilización en los distintos tipos de proyectos.

El framework Struts 2 utiliza un fichero de configuración (*struts.xml*) para inicializar sus recursos, entre los cuales encontramos:

- Interceptores que puedan procesar antes o después una petición.
- Actions que llamen a la lógica de negocio y accede a los datos del código.
- Results que puedan preparar vistas usando JavaServer y plantillas FreeMaker.

Cuando se hace la carga del *struts.xml*, la aplicación web crea una única instancia de configuración. Esto no significa que exista sólo un único fichero de xml de configuración ya que podemos crear tantos xml como queramos para definir los objetos de struts, aunque al final los lee todos para crear la configuración. Un efecto de esto último es que un error en cualquiera de ellos provoca que no sea capaz de cargar la web application.

El fichero *struts.xml* es el fichero de configuración central para el framework, y debe estar presente en el classpath de la aplicación web. Las características de este fichero de configuración son las siguientes:

- Permite su descomposición en pequeños ficheros de configuración xml, que serán incluidos en el principal. Un ejemplo puede ser el siguiente:

```
<struts>
    .....
    .....
    <include file="file1.xml"/>
    <include file="file2.xml"/>
    .....
    .....
</struts>
```

- También se puede poner un *struts-plugin.xml* en el JAR, que sera introducido automáticamente en la aplicación. De este modo se podrá configurar los components de una manera más sencilla.
- Si se desean utilizar frameworks como Freemaker o Velocity, las plantillas se pueden cargar también desde el classpath. Esto permitirá al desarrollador empaquetar el modulo completo como un simple archive .JAR.

## Estructura del fichero struts.xml

El fichero *struts.xml* atiende a las características del Struts 2 Document Type Definition (DTD) que mostramos a continuación:

```
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<!ELEMENT struts (package|include|bean|constant)*>

<!ELEMENT package (result-types?, interceptors?,
default-interceptor-ref?, default-action-ref?, global-results?,
global-exception-mappings?, action*)>
<!--ATTLIST package
    name CDATA #REQUIRED
    extends CDATA #IMPLIED
    namespace CDATA #IMPLIED
    abstract CDATA #IMPLIED
    externalReferenceResolver NMTOKEN #IMPLIED
-->

<!--ELEMENT result-types (result-type+)-->

<!--ELEMENT result-type (param*)-->
<!--ATTLIST result-type
    name CDATA #REQUIRED
    class CDATA #REQUIRED
    default (true|false) "false"
-->

<!--ELEMENT interceptors (interceptor|interceptor-stack)+-->

<!--ELEMENT interceptor (param*)-->
<!--ATTLIST interceptor
    name CDATA #REQUIRED
    class CDATA #REQUIRED
-->

<!--ELEMENT interceptor-stack (interceptor-ref+)-->
<!--ATTLIST interceptor-stack
    name CDATA #REQUIRED
-->

<!--ELEMENT interceptor-ref (param*)-->
<!--ATTLIST interceptor-ref
    name CDATA #REQUIRED
-->

<!--ELEMENT default-interceptor-ref (param*)-->
<!--ATTLIST default-interceptor-ref
    name CDATA #REQUIRED
-->

<!--ELEMENT default-action-ref (param*)-->
<!--ATTLIST default-action-ref
    name CDATA #REQUIRED
-->

<!--ELEMENT global-results (result+)-->

<!--ELEMENT global-exception-mappings (exception-mapping+)-->

<!--ELEMENT action (param|result|interceptor-ref|exception-mapping)*-->
<!--ATTLIST action
    name CDATA #REQUIRED
```



```
        class CDATA #IMPLIED
        method CDATA #IMPLIED
        converter CDATA #IMPLIED
    >

    <!--ELEMENT param (#PCDATA)-->
    <!--ATTLIST param
        name CDATA #REQUIRED
    >

    <!--ELEMENT result (#PCDATA|param)*-->
    <!--ATTLIST result
        name CDATA #IMPLIED
        type CDATA #IMPLIED
    >

    <!--ELEMENT exception-mapping (#PCDATA|param)*-->
    <!--ATTLIST exception-mapping
        name CDATA #IMPLIED
        exception CDATA #REQUIRED
        result CDATA #REQUIRED
    >

    <!--ELEMENT include (#PCDATA)-->
    <!--ATTLIST include
        file CDATA #REQUIRED
    >

    <!--ELEMENT bean (#PCDATA)-->
    <!--ATTLIST bean
        type CDATA #IMPLIED
        name CDATA #IMPLIED
        class CDATA #REQUIRED
        scope CDATA #IMPLIED
        static CDATA #IMPLIED
        optional CDATA #IMPLIED
    >

    <!--ELEMENT constant (#PCDATA)-->
    <!--ATTLIST constant
        name CDATA #REQUIRED
        value CDATA #REQUIRED
    >
```

Es posible eliminar el fichero “struts.xml” del sistema en el caso de que la aplicación no dependa de él, ya que hay algunas configuraciones que pueden ser manejadas de manera alternativa como usando anotaciones, parámetros de inicio de web.xml, etc. No obstante, algunas configuraciones siempre necesitan del fichero *struts.xml* como resultados globales, excepciones y la pila de interceptores.

### struts.xml en detalle

El tag `<struts>` es la raíz del fichero el cual puede contener los siguientes tags: paquetes, includes, beans y constants.

## El Tag Package

Los paquetes se utilizan para agrupar actions, resultados, tipos de resultados, interceptores y pilas de interceptores en una unidad lógica de configuración. Conceptualmente, los paquetes son similares a los objetos de manera que pueden extenderse y tener partes que puedan ser "sobreescritas" en "sub" paquetes.

El tag `<package />` se usa para agrupar aquellas configuraciones que tengan atributos en común como la pila de interceptores o namespaces. También puede ser útil para tener organizadas funciones que puedan a su vez separarse en diferentes ficheros de configuración.

El package presenta los siguientes atributos:

1. **name:** debe identificar de manera única un paquete. Éste parámetro es el único que es obligatorio.
2. **extends:** identifica el nombre del paquete del que se heredará toda la información de configuración, incluyendo la configuración de los action. De esta manera, toda la configuración del paquete extendido estará disponible en un nuevo paquete, bajo un nuevo nombre.
3. **namespace:** provee un mapa de las URL al paquete. Por ejemplo, para dos paquetes distintos, con nombres "pack1" y "pack2", la URL tiene el aspecto `"/webApp/pack1/my.action"` y `"/webApp/pack2/my.action"`.
4. **abstract:** si el valor de este atributo es "true", el paquete es una agrupación de configuraciones y actions que no serán accesibles vía el nombre del paquete. Es importante asegurarse de que se está extendiendo del paquete correcto para que su configuración esté disponible.

## El tag Include

El tag `<include />` se usa para modularizar una aplicación de Struts2 que necesite incluir otro fichero de configuración. Sólo contiene un atributo, "file", con el nombre del fichero que será incluido. Los ficheros que se incluyen presentan la misma estructura que `struts.xml`.

Por ejemplo, para dividir un fichero de configuración en una aplicación sobre finanzas en otros que incluyan por separado: facturas, administración, reports, etc, podríamos hacerlo de la siguiente forma:

```
<struts>
  <include file="invoices-config.xml" />
  <include file="admin-config.xml" />
  <include file="reports-config.xml" />
</struts>
```

El orden de inclusión de fichero es importante, ya que cuando los mismos actions, u otros recursos aparecen con el mismo nombre, sobrescribe los valores antiguos con los nuevos.

Hay algunos ficheros que son incluidos de forma implícita. Este es el caso del fichero *strutsdefault.xml* y *struts-plugin.xml*. Ambos contienen las configuraciones por defecto de tipos de resultados, interceptores, pila de interceptores por defecto, paquetes así como información de configuración para el entorno de ejecución. En el caso de *struts-defaults.xml* contiene la configuración del core de Struts2, y el *struts-config.xml* contiene la configuración particular de los plug-ins.

### El Tag Bean

Este tag permite definir Beans globales, aunque es una opción que no suele ser necesaria en las aplicaciones. El elemento *bean* para su definición requiere del atributo *class* que es la clase que lo implementa.

El tag presenta los siguientes atributos:

1. *class*: Es el único atributo requerido e identifica a la clase que implementa el Bean.
2. *type*: Intefaz que implementa el Bean. No es requerido.
3. *name*: nombre del bean para que podamos recuperarlo en páginas.
4. *scope*: ámbito del bean, que puede ser default, singleton, request, session o thread.

Ejemplo de bean:

```
<struts>

    <bean type="net.ObjectFactory" name="factory"
        class="sample.MyObjectFactory" />

    ...
</struts>
```

### El tag Constant

Las constantes desempeñan dos funciones:

1. Por un lado, se usan para representar datos como el tamaño máximo del fichero a subir.
2. Por otro lado, especifican el Bean que debería ser elegido, de entre las múltiples implementaciones que existen.

Las constantes pueden ser declaradas en múltiples ficheros. Por defecto, se buscan en el siguiente orden, permitiendo sobrescribir valores anteriores:

- *struts-default.xml*

- struts-plugin.xml
- struts.xml
- struts.properties
- web.xml

El fichero de configuración de struts es compatible con WebWork. En el fichero *struts.properties*, cada entrada se trata como una constante. En el fichero *web.xml*, los parámetros de inicialización de cada *FilterDispatcher* se cargan como constantes.

En algunas variantes de XML, las constantes tienen dos atributos obligatorios: *name* y *value*.

Ejemplo de constante en *struts.xml*

```
<struts>

    <constant name="struts.devMode" value="true" />
    ...

</struts>
```

## 6. Aplicación LOGIN con Struts 2

---

En esta sección, se desarrolla una aplicación de LOGIN basada en el framework de Struts 2. Esta aplicación no valida que el usuario se encuentre en una base de datos, sino que se validará con los valores del HARDCODE en el action.

El ejemplo que vamos a implementar tendrá el siguiente funcionamiento:

1. La página de login recogerá los datos de entrada.
2. El usuario introducirá el nombre y el password y luego pinchará el botón de "Login".
3. La validación del usuario se hará en el action. Si el usuario introduce Admin/Admin en los campos de nombre/contraseña, se presentará una página que informe del éxito de la validación. En caso contrario, se verá una página de error con un mensaje advirtiendo del problema.

Como vemos esta aplicación tendrá que recoger datos del cliente, procesarlos en un action, validar si la información es correcta y redirigir a las vistas en función de la lógica de negocio.

Pasamos a desarrollar la aplicación comenzando por el formulario Login.

### Desarrollo del formulario login

El GUI de la aplicación consiste en el formulario de login (login.jsp) y en el mensaje de éxito (loginsuccess.jsp). El login.jsp se usa para presentar la página con el formulario que el usuario tendrá que introducir para registrarse en el sistema. En nuestra aplicación, se salvará en el directorio "webapps\struts2tutorial\pages". A continuación vemos el código del fichero login.jsp:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Struts 2 Login Application!</title>

<link href="<s:url value="/css/main.css"/>" rel="stylesheet" type="text/css"/>

</head>
<body>
<s:form action="doLogin" method="POST">
<tr>
<td colspan="2">
Login
</td>
</tr>

<tr>
<td colspan="2">
<s:actionerror />

```

```
        <s:fielderror />
    </td>
</tr>

<s:textfield name="username" label="Login name"/>
<s:password name="password" label="Password"/>
<s:submit value="Login" align="center"/>

</s:form>

</body>

</html>
```

En el código adjunto vemos que se ha incluido los tags de struts tal y como vimos en el ejemplo primero. Para ello tenemos la línea `<%@ taglib prefix="s" uri="/struts-tags" %>`.

Para tratar los mensajes de error utilizamos los tags `<s:actionerror />` y `<s:fielderror />`. Estos tags pintan la lista de errores utilizando el fichero `struts.properties` utilizando las listas de errores y de validaciones.

El código `<sform action="doLogin" method="POST">` genera el formulario html de la aplicación que como podemos ver apunta al action `doLogin` utilizando el método POST.

Los códigos `<s:textfield name="username" label="Login name"/>` y `<s:password name="password" label="Password" />` generan los campos para introducir el nombre y la contraseña. Como se puede ver ambos se crean a partir de la librería struts, y tienen la característica de que gestionan la lectura y escritura de las propiedades username y password que presente el bean a pintar en la pantalla.

El botón de submit se genera con el código `<s:submit value="Login" align="center" />`.

Cuando se ejecute la página se va a generar el siguiente HTML:

```
<html>
  <head>
    <title>Struts 2 Login Application!</title>

    <link href="/struts2tutorial/css/main.css" rel="stylesheet"
          type="text/css"/>

  </head>
  <body>
<form id="doLogin" name="doLogin" onsubmit="return true;"
action="/struts2tutorial/roseindia/doLogin.action" method="POST">
<table class="wwFormTable">

  <tr>
    <td colspan="2">
      Login
    </td>

  </tr>
```

```
<tr>
  <td class="tdLabel"><label for="doLogin_username" class="label">
    Login name:</label>
  </td>
  <td><input type="text" name="username" value="" id="doLogin_username"/>
</td>
</tr>

<tr>
  <td class="tdLabel"><label for="doLogin_password" class="label">
    Password:</label></td>
  <td><input type="password" name="password" id="doLogin_password"/>
</td>
</tr>

<tr>
  <td colspan="2"><div align="center"><input type="submit"
    id="doLogin_0" value="Login"/>
</div></td>
</tr>
</table></form>
</body>

</html>
```

En el fichero generado en html se puede ver que Struts2 genera de manera automática los formularios, las tablas, las etiquetas.

La página loginsuccess.jsp muestra el mensaje de éxito del login una vez el usuario se ha autenticado de manera satisfactoria. El código del fichero es el que sigue:

```
<html>
<head>
<title>Login Success</title>
</head>
<body>
<p align="center"><font color="#000080" size="5">Login Successful</font></p>
</body>
</html>
```

Como se puede ver es un Html trivial que pinta el mensaje Succesful.

### Desarrollo del action.

A continuación, vamos a desarrollar la clase que maneje la petición del login. En Struts2 no es necesario implementar un action que haga de interfaz ya que cualquier POJO con un método execute puede ser utilizado. En nuestro caso hemos decidido implementar la interfaz *ActionSupport*, como podemos ver el código siguiente:

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import java.util.Date;

/**
 * <p> Validate a user login. </p>
```

```
*/
public class Login extends ActionSupport {

    public String execute() throws Exception {
        System.out.println("Validating login");
        if(!getUsername().equals("Admin") || !getPassword().equals("Admin")){
            addActionError("Invalid user name or password! Please try again!");
            return ERROR;
        }else{
            return SUCCESS;
        }
    }
}

// ---- Username property ----

/**
 * <p>Field to store User username.</p>
 * <p/>
 */
private String username = null;

/**
 * <p>Provide User username.</p>
 *
 * @return Returns the User username.
 */
public String getUsername() {
    return username;
}

/**
 * <p>Store new User username</p>
 *
 * @param value The username to set.
 */
public void setUsername(String value) {
    username = value;
}

// ---- Username property ----

/**
 * <p>Field to store User password.</p>
 * <p/>
 */
private String password = null;

/**
 * <p>Provide User password.</p>
 *
 * @return Returns the User password.
 */
public String getPassword() {
    return password;
}

/**
 * <p>Store new User password</p>
 *
 * @param value The password to set.
 */
public void setPassword(String value) {
    password = value;
}
```



```
}
```

En esta clase vemos que se ha definido las propiedades `username` y `password`. Dichas propiedades para poder ser utilizadas necesitan sus correspondientes métodos `setUsername` y `setPassword` y `getUsername` y `getPassword`.

La lógica de negocio, que en este caso consiste en validar el usuario y la password, como vemos está en el método `execute`. Vemos que se devuelve `SUCCESS` cuando la combinación es correcta y `ERROR` en caso contrario.

Cuando se hace la validación del usuario y la password, en el caso de que la combinación no sea correcta vemos que se devuelve `ERROR`, pero antes se ha añadido la siguiente línea:

```
addActionError("Invalid user name or password! Please try again!");
```

El método `addActionError` añade a la lista `errors` la línea que aparece. Dicha lista es utilizada por la etiqueta `<s:actionerror />`, que mostrará el texto de error en la pantalla.

### Configuración del action

El código que tenemos que añadir en el `struts.xml` es el que sigue:

```
<action name="showLogin">
    <result>/pages/login.jsp</result>
</action>

<action name="doLogin" class="sample.Login">
    <result name="input">/pages/login.jsp</result>
    <result name="error">/pages/login.jsp</result>
    <result>/pages/loginsuccess.jsp</result>
</action>
```

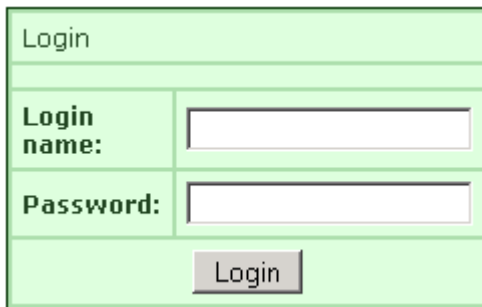
Como se ve hemos definido dos action:

El primero `showLogin` lo que hace es redirigir a la página `login.jsp` cuando se hace la petición <http://localhost:8080/struts2tutorial/showLogin.action>. Cuando un action no tiene clase utiliza una clase del core de Struts2 que lo que hará es redirigir al result `SUCCESS` (el que no tiene name).

El segundo es el action `doLogin`, el cual implementa la clase `sample.Login`. Este action tiene tres posibles destinos dependiendo de la lógica de negocio y de la lógica de navegación. En el caso que nos atañe tenemos:

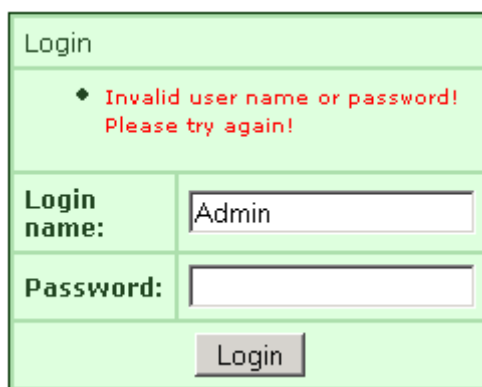
1. Si se devuelve `SUCCESS` se mostrará `loginsuccess.jsp`
2. Si se devuelve `ERROR` se mostrará `login.jsp`
3. Si se devuelve `input` o bien hay algún error de validación se mostrará `login.jsp`

Si se lanza en un navegador la dirección <http://localhost:8080/struts2tutorial/showLogin.action>, aparecerá:



The screenshot shows a web form titled "Login" with a light green background. It contains two input fields: "Login name:" and "Password:". Below these fields is a "Login" button. The form is simple and functional, with no error messages or additional text.

Si ponemos en la caja username admin, el resultado sera:



The screenshot shows the same login form, but with an error message displayed above the input fields. The message is "Invalid user name or password! Please try again!" in red text. The "Login name:" field now contains the text "Admin". The "Password:" field is empty. The "Login" button remains at the bottom.

Y finalmente si ponemos Admin/Admin aparecerá:

## Login Successful

Según se puede ver en éste ejemplo, el Action es el encargado de capturar los datos de las pantallas y determinar qué elemento de la vista se debe llamar. Hemos visto cómo la vista que se presentará dependerá de lo que sea devuelto en el método `execute`, y configuraremos el tag `action` según la navegación que hayamos pensado.

Vamos a continuar con este ejercicio, pero vamos a introducir una variante: vamos a realizar la validación de los datos que se remiten desde el cliente, mostrando los errores que se puedan producir.

## Validación en el servidor

En esta sección escribiremos el código necesario para validar los datos introducidos en el formulario. Tras completar esta sección, seremos capaces de escribir validaciones para Struts 2.

Struts 2 es un framework muy elegante que proporciona muchas funcionalidades para desarrollar aplicaciones web de manera rápida y

sencilla. Para las validaciones, el script puede ser añadido tanto en la `.jsp` como en el `action`.

El objetivo de este ejemplo es hacerlo utilizando la validación de campos automática de Struts2 utilizando el fichero de configuración `<nombre del action>-validation.xml`.

Adjuntamos el fichero `doLogin-validation.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!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="username">
        <field-validator type="requiredstring">
            <param name="trim">true</param>
            <message>Login name is required</message>
        </field-validator>
    </field>
    <field name="password">
        <field-validator type="requiredstring">
            <param name="trim">true</param>
            <message>Password is required</message>
        </field-validator>
    </field>
</validators>
```

En el fichero de configuración anterior, `fieldname` corresponde a una propiedad del bean. Siguiendo el ejemplo hemos aplicado las restricciones de requeridos para los campos `username` y `password`. Vemos que entre los tag `<message> ... </message>` escribimos el mensaje que se presentará en caso de que la validación falle. Intermente cuando no se cumpla alguna de las restricciones que se han definido, se redirigirá la petición del `action` al `result` cuyo `name` sea `input`. Así si no lo añadimos el resultado será un error.

Cuando un usuario no ponga ni usuario ni password, aparecerá:



The screenshot shows a login form titled "Login" with a light green background. At the top, there are two red error messages, each preceded by a red diamond icon: "Login name is required" and "Password is required". Below these, the form has two input fields. The first is labeled "Login name:" and the second is labeled "Password:". Both fields are empty. At the bottom of the form is a "Login" button. The entire form is enclosed in a light green border.

Además de la validación automática también podemos hacer validaciones manuales. Para ello tenemos que implementar el método `public void validate()`. En él haremos las validaciones y añadiremos tantas llamadas

`addActionError` como errores detectamos. A partir de ahí funciona exactamente igual que en el caso visto.

## Validación en el cliente

En esta sección veremos cómo escribir código que genere *JavaScript* para la validación en el cliente. En la sección anterior, se desarrolló el fichero de configuración `doLogin-validator.xml` para definir la validación del servidor. En esta sección, usaremos el mismo fichero de configuración para generar el código en el cliente.

El código correspondiente a la página de login (`loginClientSideValidation.jsp`) en `jsp` es el siguiente:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Struts 2 Login Application!</title>

<link href="<s:url value="/css/main.css"/>" rel="stylesheet"
type="text/css"/>

</head>
<body>

<s:form action="doLoginClientSideValidation" method="POST" validate="true">

<tr>
<td colspan="2">
Login
</td>

</tr>

<s:actionerror />
<s:fielderror />

<s:textfield name="username" label="Login name"/>
<s:password name="password" label="Password"/>
<s:submit value="Login" align="center"/>

</s:form>

</body>

</html>
```

Hay que tener en cuenta que lo único añadido en el código a lo anterior es `validate="true"` en el tag `<s:form>`: es todo el trabajo que tenemos que hacer en la `.jsp`, pues de todo lo demás se encargará el framework de validación de Struts2, el cuál generará el *JavaScript* para la validación del cliente.

En el `struts.xml` tendremos que añadir.

```
<action name="showLoginClientSideValidation">
    <result>/pages/loginClientSideValidation.jsp</result>
```

```
</action>
<action name="doLoginClientSideValidation" class="sample.Login">
    <result name="input">/pages/loginClientSideValidation.jsp</result>
    <result name="error">/pages/loginClientSideValidation.jsp</result>
    <result>/pages/loginsuccess.jsp</result>
</action>
```

El action `showLoginClientSideValidation` presenta el formulario de login mientras que `doLoginClientSideValidation` maneja la respuesta de la validación.

Si lanzamos la url <http://localhost:8080/struts2tutorial/showLoginClientSideValidation> veremos cómo el resultado es el mismo, salvo que se realiza en el cliente.

Si vemos la página `loginClientSideValidation.jsp` y vemos su código nos encontramos con:

```
<html>
<head>
<title>Struts 2 Login Application!</title>
<link href="/struts2tutorial/css/main.css" rel="stylesheet"
type="text/css"/>
</head>
<body>
<script src="/struts2tutorial/struts/xhtml/validation.js"></script>
<form namespace="/roseindia" id="doLoginClientSideValidation"
name="doLoginClientSideValidation" onsubmit="return
validateForm_doLoginClientSideValidation();"
action="/struts2tutorial/roseindia/doLoginClientSideValidation.action"
method="POST">
<table class="wwFormTable">
<tr>
<td colspan="2">
Login
</td>
</tr>
<tr>
<td class="tdLabel"><label for="doLoginClientSideValidation_username"
class="label">Login name:</label></td>
<td>
<input type="text" name="username" value=""
id="doLoginClientSideValidation_username"/>
</td>
</tr>
<tr>
<td class="tdLabel"><label for="doLoginClientSideValidation_password"
class="label">Password:</label></td>
<td>
<input type="password" name="password"
id="doLoginClientSideValidation_password"/>
</td>
</tr>
<tr>
<td colspan="2"><div align="center"><input type="submit"
id="doLoginClientSideValidation_0" value="Login"/>
</div></td>
</tr>
</table></form>

<script type="text/javascript">
```

```
function validateForm_doLoginClientSideValidation() {
form = document.getElementById("doLoginClientSideValidation");
clearErrorMessages(form);
clearErrorLabels(form);
var errors = false;
// field name: username
// validator name: requiredstring
if (form.elements['username']) {
field = form.elements['username'];
var error = "Login name is required";
if (field.value != null && (field.value == "" ||
field.value.replace(/^\\s+|\\s+$/g, "").length == 0)) {
addError(field, error);
errors = true;
}
}
// field name: password
// validator name: requiredstring
if (form.elements['password']) {
field = form.elements['password'];
var error = "Password is required";
if (field.value != null && (field.value == "" ||
field.value.replace(/^\\s+|\\s+$/g, "").length == 0)) {
addError(field, error);
errors = true;
}
}
return !errors;
}
</script>
</body>
</html>
```

Como podemos ver se a añadido una function *validateForm\_doLoginClientSideValidation* asociada al evento *onsubmit* del formulario que realiza las validaciones antes de hacer el envío.

### Validación utilizando anotaciones

En esta sección, vamos a validar la aplicación Login desarrollada anteriormente usando para ello anotaciones en el action. Nuestra aplicación no valida accediendo a la base de datos, sino teniendo en cuenta el código introducido en el action.

Pasos a seguir en la aplicación

1. La página de Login se muestra para permitir la introducción de los datos.
2. El usuario introduce el nombre y la contraseña y entonces pincha en "Login".
3. La validación se hace en el action y si el usuario introduce Admin/Admin como usuario/contraseña, se muestra una página de éxito. En otro caso, se verá un mensaje de error.

Los pasos a seguir para desarrollar la página de Login son los siguientes:

El GUI de la aplicación consiste en el formulario de login (*log-in.jsp*) y la página con el mensaje de éxito (*loginsuccess.jsp*).

La página *log-in.jsp* se usa para mostrar el formulario al usuario cuyo código se muestra a continuación:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Struts 2 Login Application!</title>

<link href="<s:url value="/css/main.css"/>" rel="stylesheet" type="text/css"/>

</head>
<body>

<s:form action="AnnotationAction" method="POST" validate="true">
  <tr>
    <td colspan="2">
      Login
    </td>
  </tr>

  <tr>
    <td colspan="2">
      <s:actionerror />
      <s:fielderror />
    </td>
  </tr>

  <s:textfield name="username" label="Login name"/>
  <s:password name="password" label="Password"/>
  <s:submit value="Login" align="center"/>
</s:form>
</body>
</html>
```

Como vemos es igual que en el caso anterior (ojo al `validate=true`).

A continuación vamos a desarrollar el action que maneje la petición de login. Struts2 provee de un *ActionSupport* para implementar las interfaces más comúnmente utilizadas. En nuestro action (*AnnotationAction.java*), hemos extendido la clase *ActionSupport* e importado el paquete *com.opensymphony.xwork2.validator.annotations*.

Para validar los campos del login, se puede añadir código tanto en la página *jsp* como en el action, pero Struts2 provee otro método muy sencillo para validar los formularios utilizando anotaciones en el action.

Se necesitan dos anotaciones:

- `@Validation` annotation indica que el action necesita ser validado.
- `@RequiredStringValidator` annotation se usa para validar el texto introducido.

El código del Action quedaría como sigue:

```
package sample;

import com.opensymphony.xwork2.ActionSupport;
```

## Manual de Struts2

```
import com.opensymphony.xwork2.validator.annotations.*;

@Validation
public class AnnotationAction extends ActionSupport {

    private String username = null;
    private String password = null;

    @RequiredStringValidator(message="Supply name")
    public String getUsername() {
        return username;
    }

    public void setUsername(String value) {
        username = value;
    }

    @RequiredStringValidator(message="Supply password")
    public String getPassword() {
        return password;
    }

    public void setPassword(String value) {
        password = value;
    }

    public String execute() throws Exception {

        if(!getUsername().equals("Admin") || !getPassword().equals("Admin")){
            addActionError("Invalid user name or password! Please try again!");
            return ERROR;
        } else{
            return SUCCESS;
        }
    }
}
```

En el *struts.xml* tendremos que añadir:

```
<action name="LoginAnnotation">
    <result>/pages/log-in.jsp</result>
</action>

<action name="AnnotationAction" class="sample.AnnotationAction">
    <result name="input">/pages/log-in.jsp</result>
    <result name="error">/pages/log-in.jsp</result>
    <result>/pages/loginsuccess.jsp</result>
</action>
```

Vemos que "LoginAnnotation" se usa para mostrar la página del formulario, y "AnnotationAction" hace la validación utilizando el action. Si lanzamos la url <http://localhost:8080/struts2tutorial/LoginAnnotation> veremos cómo el resultado es el mismo.



## 7. Login/Logout usando la sesión

---

El objetivo del ejemplo que a continuación se muestra es ver cómo se emplea la integración con elementos que están en la sesión desde un action.

Vamos a mostrar los elementos que se van a ir añadiendo en el ejemplo.

### Fichero de configuración

Se añade el correspondiente action en el fichero de configuración struts.xml:

```
<action name="login" class="sample.loginAction" >
  <result name="success" type="dispatcher">/pages/uiTags/Success.jsp</result>
  <result name="error" type="redirect">/pages/uiTags/Login.jsp</result>
</action>

<action name="logout" class="sample.logoutAction" >
  <result name="success" type="redirect">/pages/uiTags/checkLogin.jsp</result>
</action>
```

### Desarrollo de los action

Vamos a desarrollar un action que maneje la petición de login. El framework proporciona la clase *ActionSupport* que implementa los interfaces más comúnmente utilizados. En nuestra clase (loginAction.java), hemos extendido de la clase *ActionSupport*. El código para loginAction.java es el siguiente:

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ActionContext;
import java.util.*;

public class loginAction extends ActionSupport {
    private String userId;
    private String password;
    public String execute() throws Exception{

        if ("admin".equals(userId) && "admin".equals(password)) {
            Map session = ActionContext.getContext().getSession();
            session.put("logged-in","true");
            return SUCCESS;
        }
        else{
            return ERROR;
        }
    }

    public String logout() throws Exception {

        Map session = ActionContext.getContext().getSession();
        session.remove("logged-in");
        return SUCCESS;
    }

    public String getPassword() {
        return password;
    }
}
```

```
public void setPassword(String password) {
    this.password = password;
}

public String getUserId() {
    return userId;
}

public void setUserId(String userId) {
    this.userId = userId;
}
}
```

Para manejar la petición de logout vamos a desarrollar un segundo action cuyo código es el que sigue:

```
package sample;
import javax.servlet.http.HttpSession;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ActionContext;
import java.util.*;

public class logoutAction extends ActionSupport {
    public String execute() throws Exception {
        Map session = ActionContext.getContext().getSession();
        session.remove("logged-in");
        return SUCCESS;
    }
}
```

Como podemos ver en el código, para poder trabajar con la sesión en un action utilizamos `ActionContext.getContext().getSession()`, que nos devuelve un `Map` con las propiedades que se encuentran en la sesión. Apartir de ahí podemos leer o añadir elementos en la sesión gestionados desde el `Map`. Como se puede ver este método de trabajo con la sesión es independiente de API Servlet, con lo que facilita las acciones a realizar tanto de validación como de escritura en la sesión.

La clase `ActionContext` es útil porque nos permite no sólo recuperar el contexto de sesión, si no también la request, el contexto de página, de aplicación, etc.

### Desarrollo del formulario

La vista de nuestro ejemplo va a consistir en una jsp, que llamamos `login.jsp` y cuyo código es el que sigue:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html" %>

<html>
  <head>
    <title>Insert Data here!</title>
    <link href="<s:url value="/css/main.css"/>" rel="stylesheet"
          type="text/css"/>
  </head>
  <body>
    <s:form action="/login.action" method="POST">
      <s:textfield name="userId" label="Login Id"/><br>
    </s:form>
  </body>
</html>
```

```
<s:password name="password" label="Password"/><br>
<s:submit value="Login" align="center"/>
</s:form>
</body>
</html>
```

Vamos a emplear también una segunda jsp que muestra el mensaje de éxito:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html" import="java.util.*"%>
<jsp:include page="/struts2tags/pages/uiTags/loginCheck.jsp" />

<html>
  <head>
    <title>Welcome, you have loggedin!</title>
  </head>
  <body>
    Welcome, you have loggedin. <br />
    <b>Session Time: </b><%=new Date(session.getLastAccessedTime())%>
    <br /><br />
    <a href="<%= request.getContextPath() %>/logout.action">Logout</a>
    <br />
  </body>
</html>
```

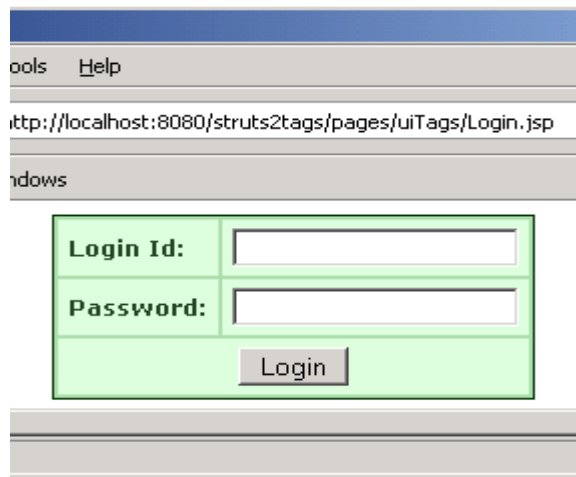
La página *success.jsp* muestra el mensaje de éxito (Welcome, you have logged-in.) y de session (Session Time: Wed Aug 01 11:26:38 GMT+05:30 2007 and Logout) cuando el usuario se identifica satisfactoriamente. Si pincha en *Logout*, se volverá a mostrar el formulario de entrada de datos de usuario.

Vemos que esta página se apoya en una llamada *loginCheck.jsp* que muestra los datos del usuario en la sesión. Adjuntamos el código de esta página.

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html" import="java.util.*"%>
<html>
  <head>
    <title>Check validate!</title>
  </head>
  <body>
    <s:if test="#session.login != 'admin'">
      <jsp:forward page="/pages/uiTags/Login.jsp" />
    </s:if>
  </body>
</html>
```

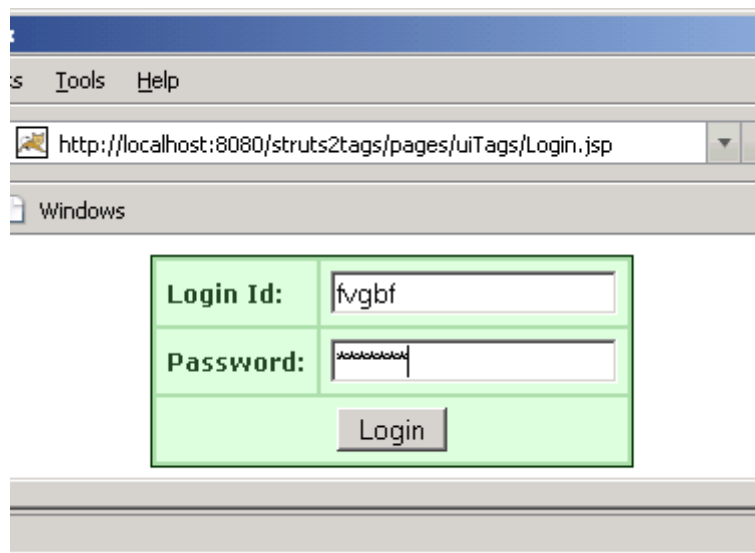
En una jsp para poder referenciar un elemento de la sesión utilizaremos desde las etiquetas de struts session, y luego la propiedad que se quiere utilizar. Por ejemplo en la línea `<s:if test="#session.login != 'admin'">` se accede a la sesión. Hemos de tener en cuenta que struts dentro de una etiqueta permite acceder a propiedades y elementos del bean o de la sesión. Para ello se antepone # al elemento que se pretende acceder. Esto no ocurre cuando el elemento está en el bean del action, en cuyo caso no se pone el símbolo #.

Si ejecutamos las páginas obtendremos:



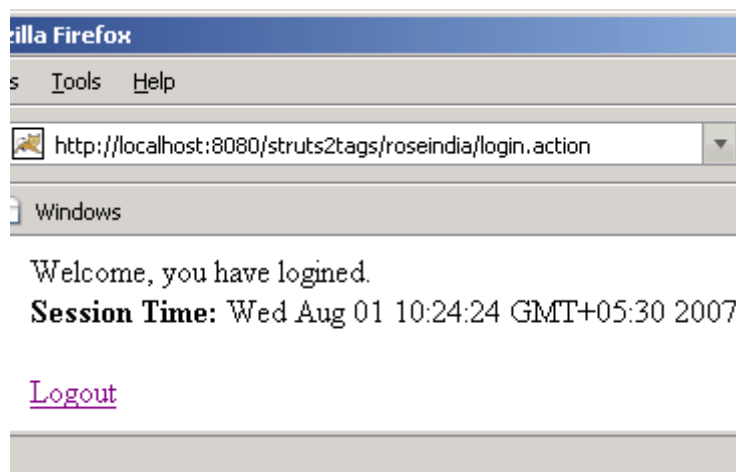
The screenshot shows a web browser window with a blue title bar. The address bar displays the URL `http://localhost:8080/struts2tags/pages/uiTags/Login.jsp`. Below the address bar is a menu bar with 'ools' and 'Help'. The main content area contains a login form with a green border. The form has two input fields: 'Login Id:' and 'Password:'. Below these fields is a 'Login' button.

Si ponemos un usuario erróneo obtendremos:



The screenshot shows the same login form as before, but with an error message. The 'Login Id:' field contains the text 'fvgbf'. The 'Password:' field contains a series of asterisks. Below the fields is a 'Login' button.

Si ponemos datos correctos lo que saldrá es:



The screenshot shows a web browser window with a blue title bar. The address bar displays the URL `http://localhost:8080/struts2tags/roseindia/login.action`. Below the address bar is a menu bar with 's', 'Tools', and 'Help'. The main content area contains a success message: 'Welcome, you have logged in.' Below this message is the text 'Session Time: Wed Aug 01 10:24:24 GMT+05:30 2007'. At the bottom of the page is a link labeled 'Logout'.

Existe otra forma de trabajar con la sesión que es vía la implementación de la interfaz `org.apache.struts2.interceptor.SessionAware`. Esta interfaz obliga a

definir los métodos `setSession` y `getSession`. A continuación mostramos un ejemplo de uso en una clase de dicha intrefaz:

```
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class GetSession extends ActionSupport implements SessionAware{

    private Map session;
    public String execute() throws Exception{
        sesion.put("dato", "valor");
        return SUCCESS;
    }

    public void setSession(Map session){
        session = this.getSession();
    }

    public Map getSession(){
        return session;
    }
}
```

Este action pone en la sesión una variable "dato" con un valor "valor".

## 8. Struts 2 Tags

---

En esta sección vamos a introducir la librería de Tags del framework Struts2, intentando explicar los tags que se incorporan.

The Struts 2 Tags can be divided into two types:

### 1. Tags Genéricos

Los tags genéricos se emplean para controlar el flujo de ejecución durante la renderización de una página.

Dentro de esta categoría vamos a encontrar dos tipos de tags, los de Control, que son aquellos que se usan para el flujo de control como if, else, etc, y los de Datos que se emplean para el uso o manipulación de datos.

### 2. Tags de Interfaz de usuarios.

Éstos tags estan diseñados para mostrar información en el navegador utilizando información que proviene del action o de tags de datos. Lo que generan es HTML que podrá ser visualizado.

Los tags utilizan *templates* y *themes* para generar el HTML final que se devuelve.

are mainly designed to use the data from your action/value stack or from Data Tags.

### Tags genéricos

Como hemos comentado anteriormente, los tags genéricos se emplean para controlar el flujo de ejecución durante la renderización de una página.

Hay dos tipos de tags genéricos: los de control y los de datos.

Los tags de control son:

- **if**
- **elseif**
- **else**
- **append**
- **generator**
- **iterator**
- **merge**
- **sort**
- **subset**

Los tags de Datos son:

- **a**

- **action**
- **bean**
- **date**
- **Debug**
- **i18n**
- **include**
- **param**
- **Push**
- **set**
- **text**
- **url**
- **property**

A continuación se muestran ejemplos de uso de los tags.

### Control Tags-If / Else If / Else

En esta sección vamos a ver el uso de los tags de selección.

El tag *If* puede usarse solo, o con el tag *Else If* y/o con el tag *Else* tal y como se usa en Java. Vemos un ejemplo de uso en el ejemplo siguiente:

```
<html>
<head>
  <title>Struts 2 Control Tag Example</title>
</head>
<body>
<s:set name="technologyName" value="%{'Java'}"/>

  <s:if test="%{#technologyName=='Java'}">
    <div><s:property value="%{#technologyName}" /></div>
  </s:if>

  <s:elseif test="%{#technologyName=='Jav'}">
    <div><s:property value="%{#technologyName}" /></div>
  </s:elseif>

  <s:else>
    <div>Technology Value is not Java</div>
  </s:else>

</body>
</html>
```

En el ejemplo lo primero que podemos ver es cómo se asigna una variable usando tags de Struts2, utilizando `<s:set name="technologyName" value="%{'Java'}"/>`. El resultado es que se crea un atributo asociado al *pageContext* con nombre *technologyName* y con valor *Java*.

A continuación vemos cómo se va a ir comprobando si el valor es *Java* o *Jav*. Vemos que para referenciar una variable en una etiqueta de struts se pone % y entre llaves la variable. En este caso como se ve se pone antes de la variable #.

El orden de ejecución entre los tag *if*, *elseif* y *else* es igual que en todos los lenguajes de programación

### Tag Append

El tag *append* se usa para crear un iterador a partir de multiples iteradores. De esta forma permite juntar listas o colecciones en una única lista o colección. Veamos un ejemplo:

En el struts.xml tenemos:

```
<action name="AppendTag" class="sample.AppendTag">
    <result>/pages/genericTags/AppendTag.jsp</result>
</action>
```

El action es como sigue.

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class AppendTag extends ActionSupport{

    private List myList;
    private List myList1;

    public String execute()throws Exception{
        myList = new ArrayList();
        myList.add("Deepak Kumar");
        myList.add("Sushil Kumar");
        myList.add("Vinod Kumar");
        myList.add("Amit Kumar");

        myList1 = new ArrayList();
        myList1.add("www.javajazzup.com");
        myList1.add("Himanshu Raj");
        myList1.add("Mr. khan");
        myList1.add("John");
        myList1.add("Ravi Ranjan");
        return SUCCESS;
    }

    public List getMyList(){
        return myList;
    }

    public List getMyList1(){
        return myList1;
    }
}
```

Como podemos ver el action define dos listas que podremos usar en una jsp como la siguiente:

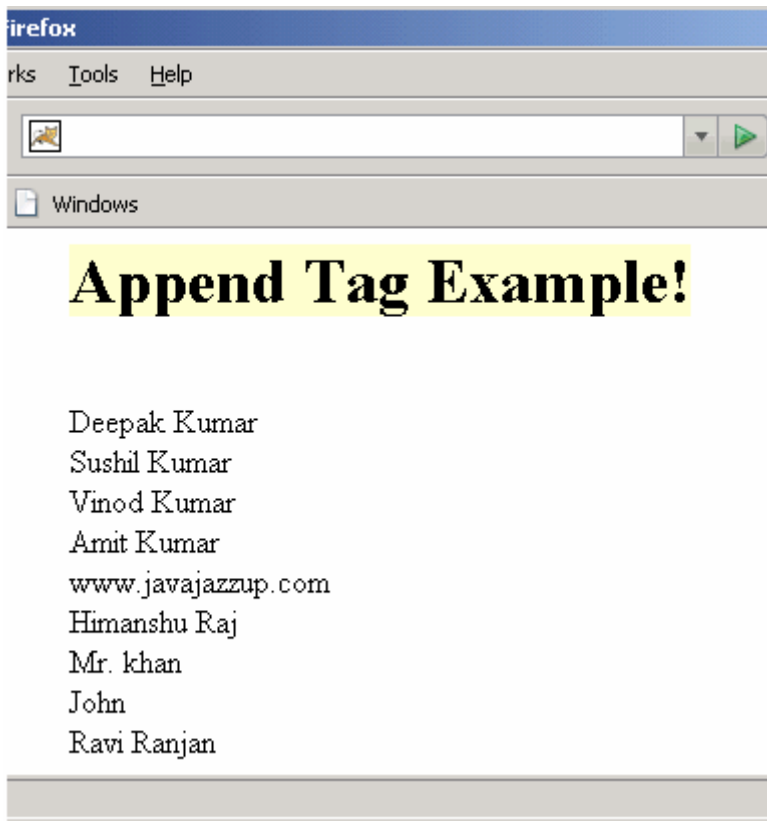
```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
    <title> Append Tag Example!</title>
</head>
<body>
<h1><span style="background-color: #FFFFcc"> Append Tag Example!</span></h1>
    <s:append id="myAppendList">
```



```
<s:param value="%{myList}" />
<s:param value="%{myList1}" />
</s:append>
<s:iterator value="%{#myAppendList}">
  <s:property /><br>
</s:iterator>
</body>
</html>
```

En el código de la jsp que lo se hace es concatenar la lista *myList* con *myList1* creando una nueva lista *myAppendList*. Si ejecutamos el action el resultado que vamos a obtener es:



## Tag Generator

El tag *generator* se emplea para generar iteradores a partir de diferentes atributos que se pasan como parámetros. Para ver un ejemplo vamos a mostrar solo la jsp con el uso del tag y veremos cómo sería el resultado si se hiciese una llamada.

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title> Generator Tag Example! </title>
  </head>
  <body>
    <h1>
      <span style="background-color: #FFFFcc">Generator Tag Example!</span></h1>
      <h3><font color="#0000FF"> Generates a Simple Iterator </font></h3>
      <s:generator val="%{'Deepak Kumar,Sushil Kumar,
                          Vinod Kumar,Amit Kumar'}" separator=",">
```

```
<s:iterator>
  <s:property /><br/>
</s:iterator>
</s:generator>
</body>
</html>
```

El tag generator lo que hará es generar una lista a partir de la cadena 'Deepak Kumar,Sushil Kumar, Vinod Kumar,Amit Kumar', utilizando la ',' como delimitador para determinar cada uno de los elementos. Esto es equivalente a hacer un *split* de Java. A continuación dentro del generatos tenemos un iterador el cual itera sobre cada uno de los elementos y muestra el valor.

Esta forma de usar el tag *iterator* se hace posible porque el generator crea una lista que le pasa al iterador, y a su vez cada valor es lo que toma por defecto el tag property. Al final el resultado es:



El tag generator puede tomar un parámetro count que permite delimitar cuántos elementos son los que va a dejar en la lista que genera. Así si ponemos *count=3* el ultimo element *Amit Kumar* no aparacería.

### Tag iterator

Esta tag se emplea para iterar sobre un valor que tiene que ser de tipo *java.util.Collection*, *java.util.Iterator* o bien un array. Vamos a ver un ejemplo :

El action es la clase Java que sigue:

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class iteratorTag extends ActionSupport{
```

```
private List myList;

public String execute()throws Exception{
    myList = new ArrayList();
    myList.add("Fruits");
    myList.add("Apple");
    myList.add("Mango");
    myList.add("Orange");
    myList.add("Pine Apple");
    return SUCCESS;
}

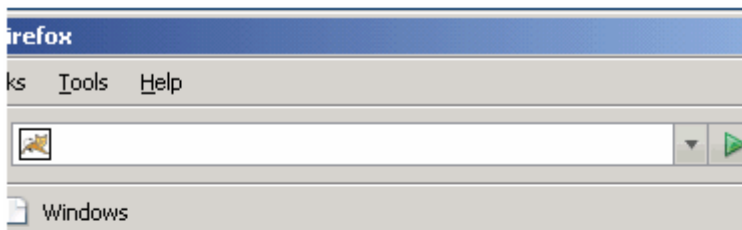
public List getMyList(){
    return myList;
}
}
```

Como se ve hemos definido una clase que tiene una lista de frutas. Dicha lista podemos recuperar en una jsp ya que se ha definido un método *getMyList*. La jsp siguiente hace uso de este action.

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title>Iterator Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-color:#FFFFcc">Iterator Tag Example!</span></h1>
    <s:iterator value="myList">
      <s:property /><br>
    </s:iterator>
  </body>
</html>
```

Como se puede ver el tag iterator hace uso de la lista myList para recorrerla. Igual que el ejemplo anterior usamos *s:property* para mostrar su valor. El resultado será:



## Iterator Tag Example!

Fruits  
Apple  
Mango  
Orange  
Pine Apple

## Tag Merge

Este tag se emplea para fusionar iteradores. La forma de hacer esta fusion es vía ir mezclando de forma sucesiva cada elemento de cada una de las listas. Así la primera llamada recupera el primer elemento de la primera lista, la segunda llamada el primero de la segunda, y así sucesivamente. En el ejemplo que sigue vamos a mezclar dos listas: la primera tiene 5 entradas y la segunda también. El resultado lo podremos ver en la imagen adjunta.

Primero mostramos el action que define las dos listas.

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class mergeTag extends ActionSupport {
    private List myList;
    private List myList1;

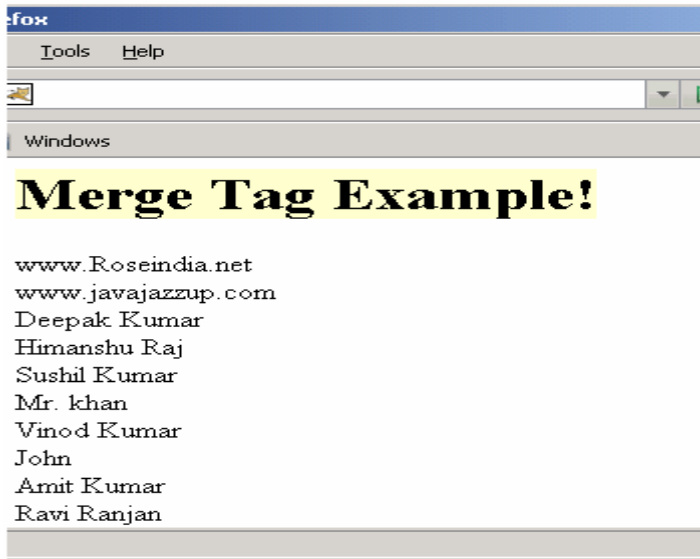
    public String execute() throws Exception{
        myList = new ArrayList();
        myList.add("www.Roseindia.net");
        myList.add("Deepak Kumar");
        myList.add("Sushil Kumar");
        myList.add("Vinod Kumar");
        myList.add("Amit Kumar");

        myList1 = new ArrayList();
        myList1.add("www.javajazzup.com");
        myList1.add("Himanshu Raj");
        myList1.add("Mr. khan");
        myList1.add("John");
        myList1.add("Ravi Ranjan");
        return SUCCESS;
    }
    public List getMyList(){
        return myList;
    }
    public List getMyList1(){
        return myList1;
    }
}
```

La jsp es la que sigue donde podemos ver cómo se hace para ejecutar el merge.

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
    <title>Merge Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-color: #FFFFcc">Merge Tag Example!</span></h1>
    <s:merge id="mergeId">
      <s:param value="%{myList}" />
      <s:param value="%{myList1}" />
    </s:merge>
    <s:iterator value="%{#mergeId}">
      <s:property /><br>
    </s:iterator>
  </body>
</html>
```

Y el resultado será:



### Tag subset

El tag *subset* toma como parámetro un iterador y genera un subconjunto del mismo utilizando la clase `org.apache.struts2.util.SubsetIteratorFilter`. Para ver cómo se usa mostraremos el action con la lista y la jsp que utiliza el tag. En nuestro caso vamos a emplear una lista de números.

El action es el que sigue:

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class subsetTag extends ActionSupport {
    private List myList;

    public String execute() throws Exception{
        myList = new ArrayList();
        myList.add(new Integer(50));
        myList.add(new Integer(20));
        myList.add(new Integer(100));
        myList.add(new Integer(85));
        myList.add(new Integer(500));
        return SUCCESS;
    }

    public List getMyList(){
        return myList;
    }
}
```

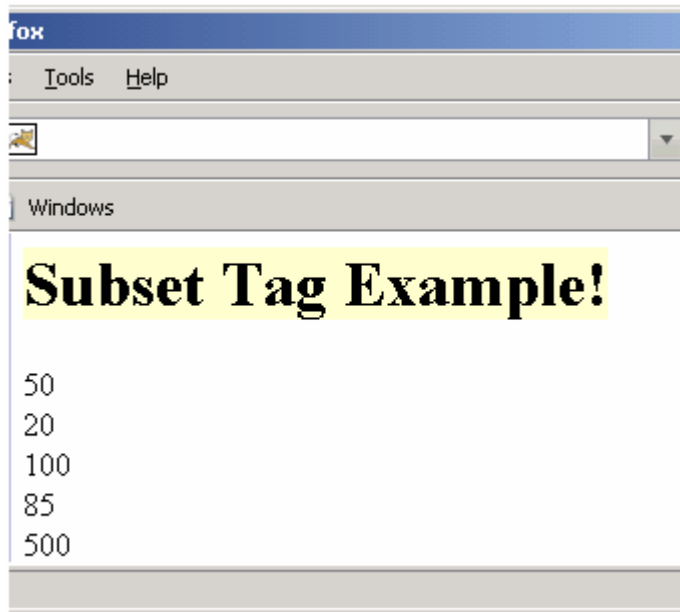
La jsp sería.

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
<title>Subset Tag Example!</title>
</head>
<body>
<h1><span style="background-
color: #FFFFcc">Subset Tag Example!</span></h1>
<s:subset source="myList">
```

```
<s:iterator>
  <s:property /><br>
</s:iterator>
</s:subset>
</body>
</html>
```

Como se ve su uso es similar al de los tags anteriores. El resultado sería:

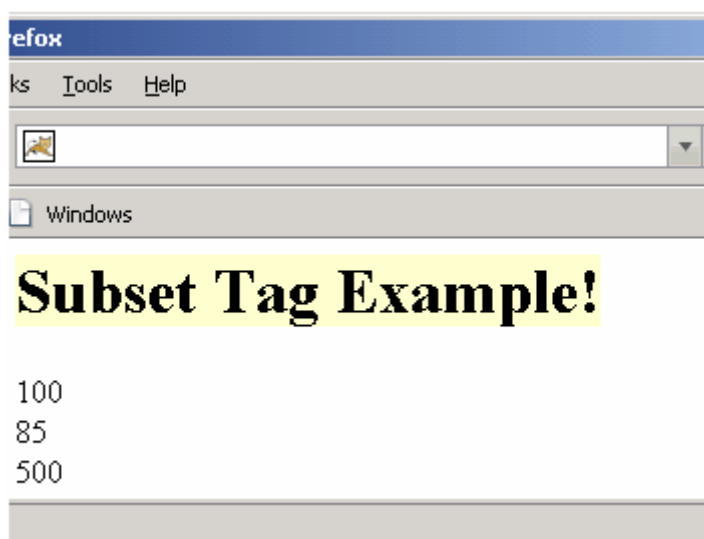


Como vemos en el caso que hemos visto no ha tenido ningún efecto ya que no hemos determinado las condiciones para crear el subconjunto.

Si cambiamos el tag *subset* como sigue:

```
<s:subset source="myList" count="3" start="2">
```

Obtenemos como resultado lo siguiente:



Como se ve ha recuperado tres elementos empezando por el Segundo. Este tag es algo así como el substring, de la clase String salvando las distancias.

## Tag action

Permite ejecutar una acción desde una vista indicando el nombre de la acción y, opcionalmente, el espacio de nombres. Se puede pasar parámetros utilizando la etiqueta `param`. Presenta los siguientes parámetros:

- `executeResult`: determina si el resultado de la acción (normalmente otra vista) se debe ejecutar / renderizar también.
- `ignoreContextParams`: indica si se deben incluir los parámetros de la petición actual al invocar la acción.
- `name` (requerido): el nombre de la acción a ejecutar.
- `namespace`: el espacio de nombres de la acción a ejecutar.

Para ver qué ocurriría vamos a ver un ejemplo:

La jsp que se llama es la que sigue. En ella se está incluye un action `Accion`, que llamará a la jsp `resultado.jsp`

```
<html>
<body>
Antes de s:action<br/>
<s:action name="Accion" executeResult="true" /><br/>
Después de s:action
</body>
</html>
```

El action es la clase `Accion.java` que sigue:

```
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Accion extends ActionSupport {
    private String web;

    public String getWeb() {
        return web;
    }

    public String execute() {
        web = "mundogeek.net";
        return SUCCESS;
    }
}
```

Y finalmente la página de resultado será:

```
<%@ taglib uri="/struts-tags" prefix="s"%>

Hola
```

El resultado de llamar a la primera jsp será una página en la que aparece:

```
Antes de s:action
Hola
Después de s:action
```

Como vemos lo que hace el tag es llamar a otro action y permitir introducir el resultado en una jsp.

### Tag Bean

Este tag instancia un Java Bean. Para ello en el atributo name ponemos la clase que tiene que instanciar y en var el nombre de la variable que se añadirá en el ValueStack.

Veamos un ejemplo:

Definimos la clase *companyName.java* como se ve a continuación.

```
package sample ;

public class companyName {

    private String name;

    public void setName(String name){
        this.name =name ;
    }

    public String getName(){
        return name;
    }
}
```

En la vista tenemos la siguiente jsp:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
    <title>Bean Tag Example!</title>
</head>
<body>
    <h1><span style="background-color: #FFFFcc">Bean Tag
        (Data Tags) Example!</span></h1>
    <s:bean name="sample.companyName" id="uid">
        <s:param name="name">RoseIndia</s:param>
        <s:property value="%{name}" /><br>
    </s:bean>
</body>
</html>
```

El resultado de ejecutar el action es:





Lo que se está haciendo en el ejemplo es definir en la jsp un bean con la etiqueta `<s:bean name="sample.companyName" id="uid">`, y con la etiqueta `<s:param name="name">RoseIndia</s:param>` se está asignando valor a la propiedad name de dicho bean.

Para terminar se muestra el valor que se acaba de asignar, como así aparece en la imagen que vemos.

### Tag Date

Permite mostrar una fecha almacenada en una cierta variable indicando opcionalmente el formato a utilizar. Emplea tres atributos que son:

- **format**: Formato a utilizar para mostrar la fecha. Si queremos usar siempre el mismo formato podemos crear un archivo properties con una entrada `struts.date.format`. Por defecto se utiliza el formato `DateFormat.MEDIUM`.
- **name** (requerido): Nombre de la variable que contiene la fecha a mostrar.
- **nice**: Utiliza un formato que facilita la lectura. Por defecto se utiliza inglés para mostrar los mensajes; si queremos traducirlo a algún otro idioma tendremos que recurrir a las funciones de internacionalización de Struts 2.

Veamos un ejemplo:

Sea la clase `dateTag.java` que tenemos a continuación:

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class dateTag extends ActionSupport {
    private Date currentDate;
    public String execute() throws Exception{
        setCurrentDate(new Date());
        return SUCCESS;
    }
    public void setCurrentDate(Date date){
```

```

        this.currentDate = date;
    }
    public Date getCurrentDate(){
        return currentDate;
    }
}

```

Este action define una propiedad de tipo date, currentDate.

La vista va a ser la jsp date.jsp:

```

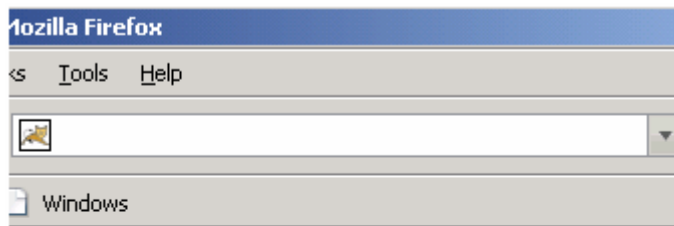
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
    <title>Date Tag (Data Tag) Example!</title>
</head>
<body>
    <h1><font color="#000080">Current Date Format</font></h1>
    <table border="1" width="35%" bgcolor="ffffcc">
        <tr>
            <td width="50%"><b><font color="#000080">Date Format</font></b></td>
            <td width="50%"><b><font color="#000080">Date</font></b></td>
        </tr>
        <tr>
            <td width="50%">Day/Month/Year</td>
            <td width="50%"><s:date name="currentDate" format="dd/MM/yyyy" /></td>
        </tr>
        <tr>
            <td width="50%">Month/Day/Year</td>
            <td width="50%"><s:date name="currentDate" format="MM/dd/yyyy" /></td>
        </tr>
        <tr>
            <td width="50%">Month/Day/Year</td>
            <td width="50%"><s:date name="currentDate" format="MM/dd/yy" /></td>
        </tr>
        <tr>
            <td width="50%">Month/Day/Year Hour<b>:</b>Minute</td>
            <td width="50%"><s:date name="currentDate" format="MM/dd/yy hh:mm" />
        </td>
        </tr>
        <tr>
            <td width="50%">Month/Day/Year Hour<b>:</b>Minute<b>:</b>Second</td>
            <td width="50%"><s:date name="currentDate" format="MM/dd/yy hh:mm:ss" />
        </td>
        </tr>
        <tr>
            <td width="50%">Nice Date (Current Date & Time)</td>
            <td width="50%"><s:date name="currentDate" nice="false" /></td>
        </tr>
        <tr>
            <td width="50%">Nice Date</td>
            <td width="50%"><s:date name="currentDate" nice="true" /></td>
        </tr>
    </table>

</body>
</html>

```

El resultado será:



## Current Date Format

Date Format	Date
Day/Month/Year	19/07/2007
Month/Day/Year	07/19/2007
Month/Day/Year	07/19/07
Month/Day/Year Hour:Minute	07/19/07 11:43
Month/Day/Year Hour:Minute:Second	07/19/07 11:43:35
Nice Date (Current Date & Time)	Jul 19, 2007 11:43:35 AM
Nice Date	an instant ago

Como vemos lo que ocurre es que gracias al tag action podemos cambiar la fecha va cambiando de formato.

### Tag Include

Parecido a `action` con el parámetro `executeResult`, pero, a diferencia de este, permite incluir cualquier recurso. Como el `include` de JSP también se le pueden pasar parámetros para que sea dinámico con etiquetas `param`, pero estos valores no se acceden como propiedades de `valueStack`, sino como parámetros de la petición.

Como ejemplo vamos a crear la jsp siguiente, que referencia al action `dateTag.action` visto ante:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title>Include Tag Example!</title>
  </head>
  <body><s:include value="dateTag.action" /></body>
</html>
```

El resultado será exactamente el mismo.

Este tag es igual que usar un include de jsp.

### Tag Param

Se utiliza para añadir parámetros a otras etiquetas. Tiene dos atributos que son *name*, para indicar el nombre del parámetro y *value*, para indicar el valor.

### Tag set

Asigna un valor a una variable, opcionalmente indicando el ámbito al que añadirla. El valor se puede indicar utilizando el atributo *value*, o encerrando el valor en la propia etiqueta. Para identificar el nombre de la variable a la que se asocia se utiliza el parámetro *name*.

### Tag Text

Este tag se emplea para poder renderizar mensajes de texto basados en la especificación I18n. Para su se crea un fichero de propiedades que tenga el mismo nombre que nuestro action y que acabe en .properties. Adicionalmente podemos añadir el código del lenguaje al nombre del fichero para internacionalizarlo.

Vamos a ver un ejemplo:

Creamos el action textTag.java.

```
import com.opensymphony.xwork2.ActionSupport;

public class textTag extends ActionSupport {

    public String execute() throws Exception{
        return SUCCESS;
    }
}
```

Creamos en el mismo directorio textTag.properties con los siguientes valores.

```
webname1 = http://www.RoseIndia.net
webname2 = http://www.javajazzup.com
webname3 = http://www.newstrackindia.com
```

Si creamos la vista con el siguiente código:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

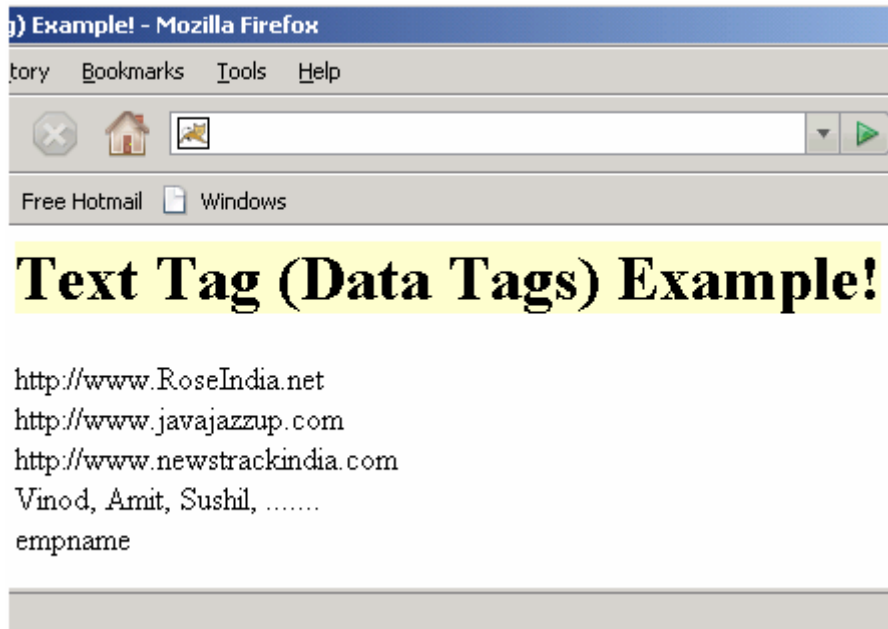
<html>
  <head>
    <title>Text Tag (Data Tag) Example!</title>
  </head>
  <body>
    <h1><span style="background-color: #FFFFcc">Text Tag
                                   (Data Tags) Example!</span></h1>

    <s:text name="webname1"></s:text><br>
```

```
<s:text name="webname2"></s:text><br>
<s:text name="webname3"></s:text><br>
<s:text name="empname">Vinod, Amit, Sushil, .....</s:text><br>
<s:text name="empname"></s:text>

</body>
</html>
```

Obtenemos:



### Tag property

Muestra una propiedad de ValueStack u otro objeto de ActionContext.

Los parámetros que acepta son

- default: Valor a mostrar en caso de que el valor pedido sea nulo.
- escape: Determina si queremos que se escape el HTML. Por defecto es true.
- value: Valor a mostrar.

### Struts UI tags

Este tipo de tags están especializados en el uso de datos desde el action o valueStack generando Html que puede verse en la vista.

Hay dos tipos de tags que son los Tags de interfaz de usuario y los tags que no son de formulario.

Entre los primeros encontramos:

- **autocomplete**
- **checkbox**

- **checkboxlist**
- **combobox**
- **datetimepicker**
- **doubleselect**
- **Head**
- **file**
- **form**
- **Hidden**
- **label**
- **optiontransferselect**
- **optgroup**
- **password**
- **radio**
- **reset**
- **select**
- **submit**
- **textarea**
- **textfield**
- **token**
- **updownselect**

Entre los segundos tenemos:

- **actionerror**
- **actionmessage**
- **fielderror**

De forma general todos los tags que se emplean para formularios necesitan el parámetro *name* para identificar cuál es el parametro que se enviará cuando se haga el *submit*. Además incluyen propiedades para hacer la maquetación, tales como *cssClass* que permite definir qué class se empleará y *cssStyle* que permite meter estilos en el elemento.

### Tag Autocompleter

El tag *autocompleter* muestra un combo y un textfield, y su funcionamiento permite buscar dentro de una lista que tiene como parametro aproximando el resultado más cercano de lo que se escribe en la caja de texto. Lo más sencillo es verlo con un ejemplo como el que sigue:

Definimos *autocompleter.java* que es el action del ejemplo. Éste action publica una propiedad state.

```
package sample;
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class autocompleter extends ActionSupport{
```

```
private List state;
public String execute() throws Exception{
    state = new ArrayList();
    state.add("Alabama");
    state.add("Alaska");
    state.add("Arizona");
    state.add("Arkansas");
    state.add("California");
    state.add("Colorado");
    state.add("Connecticut");
    state.add("Delaware");
    state.add("District of Columbia");
    state.add("Florida");
    state.add("Georgia");
    state.add("Hawaii");
    state.add("Idaho");
    state.add("Illinois");
    state.add("Indiana");
    state.add("Iowa");
    state.add("Kansas");
    state.add("Kentucky");
    state.add("Louisiana");
    state.add("Maine");
    state.add("Maryland");
    state.add("Massachusetts");
    state.add("Michigan");
    state.add("Minnesota");
    state.add("Mississippi");
    state.add("Missouri");
    state.add("Montana");
    state.add("Nebraska");
    state.add("Nevada");
    state.add("New Hampshire");
    state.add("New Jersey");
    state.add("New Mexico");
    state.add("New York");
    state.add("North Carolina");
    state.add("North Dakota");
    state.add("Ohio");
    state.add("Oklahoma");
    state.add("Oregon");
    state.add("Pennsylvania");
    state.add("Rhode Island");
    state.add("South Carolina");
    state.add("South Dakota");
    state.add("Tennessee");
    state.add("Texas");
    state.add("Utah");
    state.add("Vermont");
    state.add("Virginia");
    state.add("Washington");
    state.add("West Virginia");
    state.add("Wisconsin");
    state.add("Wyoming");
    return SUCCESS;
}
public List getState(){
    return state;
}
}
```

La jsp será:

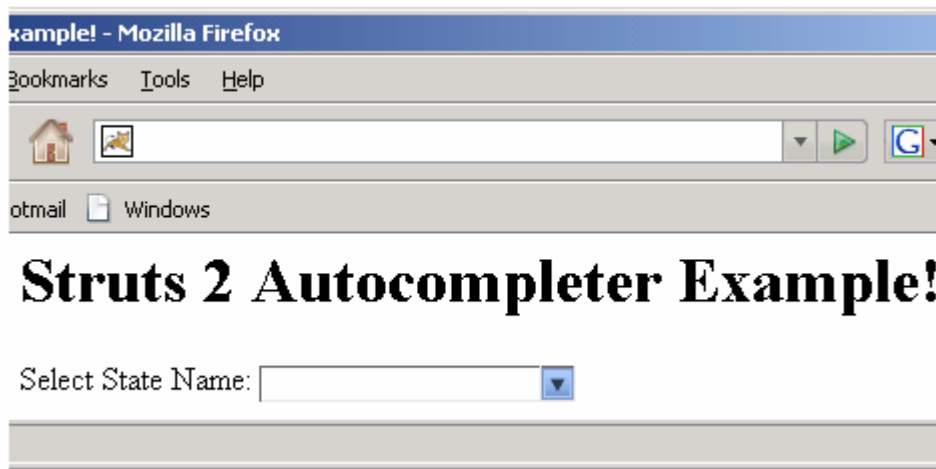
```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
    <title>Struts 2 Autocompleter Example!</title>
```

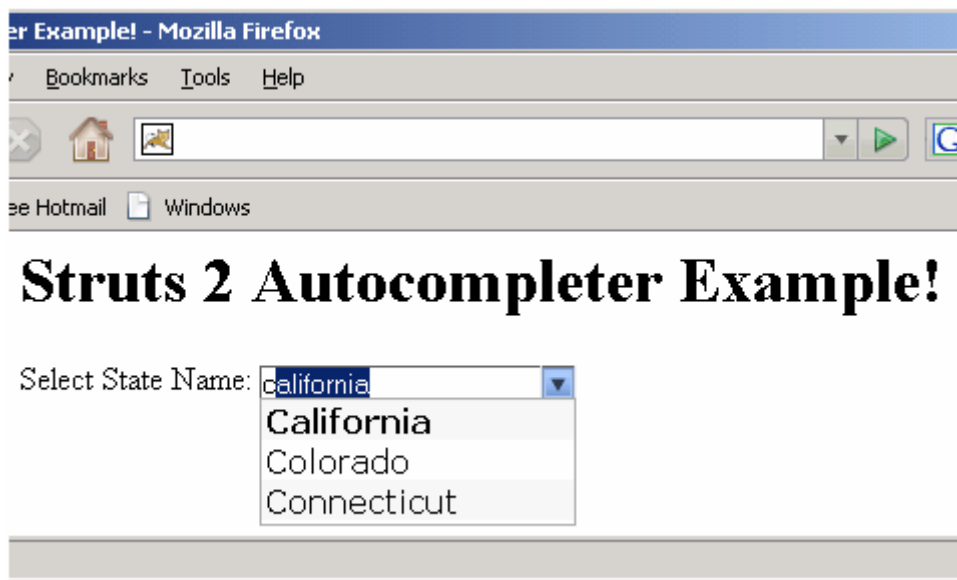
```
<s:head theme="ajax" />
</head>
<body>
  <h1>Struts 2 Autocompleter Example!</h1>
  <s:label name="stateName" value="Select State Name:" />
  <s:autocompleter theme="simple" list="state" name="StateName"/>
</body>
</html>
```

En la jsp vemos cómo el tag tiene como parámetro la lista y el nombre de la variable que se usará para el envío del formulario, ya que crea una caja de texto asociada.

El resultado visible es el siguiente:



Si en la caja escribimos una 'c', ocurre:





## Tag checkbox

El tag `checkbox` es un tag de formulario que se usa para renderizar un `checkbox HTML`, gestionando y publicando el valor desde el `ValueStack`. Veamos un ejemplo:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title>Checkbox (Form Tag) Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-
color: #FFFFcc">Checkbox Tag Example!</span></h1>
    <b>Sex</b><br>
    <s:checkbox label="Male" name="male" value="true" /><br>
    <s:checkbox label="Female" name="male" />
  </body>
</html>
```

Renderizado se ve:



## Tag checkboxlist

Muestra un conjunto de checkboxes desde una lista. Veamos un ejemplo:

`checkboxlistTag.java`

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class checkboxlistTag extends ActionSupport{

  private List fruits;
  private List animals;
  public String execute()throws Exception{
    fruits = new ArrayList();
    fruits.add("Apple");
    fruits.add("Mango");
    fruits.add("Orange");
    fruits.add("Pine Apple");
```

```

        animals = new ArrayList();
        animals.add("Dog");
        animals.add("Elephant");
        animals.add("Ox");
        animals.add("Fox");
        return SUCCESS;
    }

    public List getFruits(){
        return fruits;
    }

    public List getAnimals(){
        return animals;
    }
}

```

El action como vemos publica dos listas animals y fruits. La jsp tiene el código siguiente.

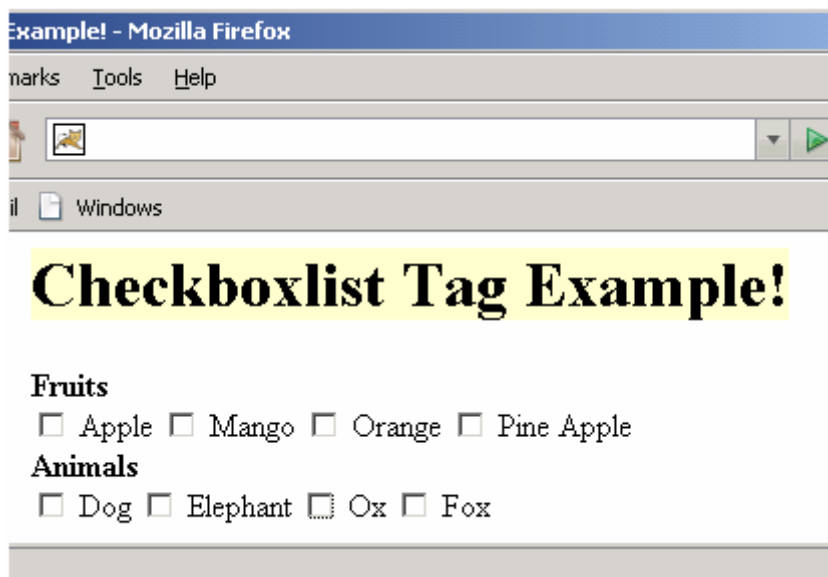
```

<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title>Checkboxlist (Form Tag) Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-color: #FFFFcc">Checkboxlist
      Tag Example!</span></h1>
    <b>Fruits</b><br>
    <s:checkboxlist name="Fruits-name" list="fruits" /><br>
    <b>Animals</b><br>
    <s:checkboxlist name="Animals-name" list="animals" /><br>
  </body>
</html>

```

El resultado será:



## Tag combobox

Este tag es básicamente un select de html que se carga desde una lista, junto a un textfield. La funcionalidad consiste en que podemos introducir un nuevo valor dentro de la caja, o seleccionar uno de la lista. Al hacer esto último, la selección se sitúa en la caja de texto.

Ejemplo:

El action define la lista que utilizaremos de referencia.

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class comboboxTag extends ActionSupport{

    private List fruits;
    public String execute()throws Exception{
        fruits = new ArrayList();
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Pine Apple");
        return SUCCESS;
    }

    public List getFruits(){
        return fruits;
    }
}
```

La jsp contiene lo siguiente:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

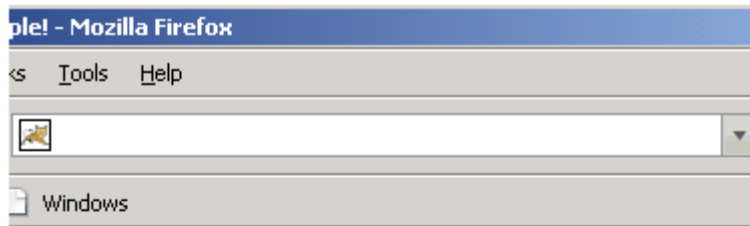
<html>
<head>
    <title>Combobox (Form Tag) Tag Example!</title>
    <link href="<s:url value="/css/main.css"/>" rel="stylesheet"
          type="text/css"/>
</head>
<body>
<h1><span style="background-color: #FFFFcc">Combobox
      Tag Example!</span></h1>
<s:form>
    <s:combobox label="Colors Name" name="colorNames"
                headerValue="--- Please Select ---"
                headerKey="1" list="{ 'Black','Green','White','Yellow',
                                     'Red','Pink'}" /><br>

    <!-- Use array list --><br>
    <s:combobox label="Fruits Name" name="fruitsNames"
                headerValue="--- Please Select ---"
                headerKey="1" list="fruits" />
</s:form>
</body>
</html>
```

Como se ve en el ejemplo el tag nos permite añadir un *headerValue* que es el valor que se muestra por defecto. Además también podemos definir el

nombre que el formulario utilizará para enviar este parámetro al servidor, usando *name*.

La ejecución nos dará:



## Combobox Tag Example!

Colors Name:	<input type="text" value="— Please Select —"/>
Fruits Name:	<input type="text" value="Orange"/>

### Tag Datapicker

El tag *datetimepicker* es un componente visual que se usa para mostrar un selector de día y hora utilizando combos.

Vamos a ver un ejemplo de su uso.

El action es *includeTag.java* con el siguiente código.

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class includeTag extends ActionSupport {
    private Date myBirthday;
    public String execute() throws Exception{
        setMyBirthday(new Date("Jan 12, 1984 11:21:30 AM"));
        return SUCCESS;
    }
    public void setMyBirthday(Date date){
        this.myBirthday = date;
    }
}
```

```
    public Date getMyBirthday(){
        return myBirthday;
    }
}
```

La jsp será datetimepicker.jsp que presenta el código que sigue:

```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
    <title>Datetimepicker (Form Tag) Tag Example!</title>
    <link href="<s:url value="/css/main.css"/>" rel="stylesheet"
          type="text/css"/>

    <s:head theme="ajax" />
</head>
<body>
    <h1><span style="background-color: #FFFFcc">Datetimepicker
      Tag Example!</span></h1>
    <s:datetimepicker name="myBirthday" label="My Birth Day
      (dd-MM-yyyy)" displayFormat="dd-MM-yyyy" />
</body>
</html>
```

El resultado como sigue:



Este componente se puede mejorar cambiando estilos, formatos ,etc.

### Tag doubleselect.

El tag *doubleselect* permite mostrar dos select de HTML relacionados entre sí, de tal manera que cuando cambia el valor del primero se actualiza el segundo. Lo vemos en el siguiente ejemplo.

Creemos la jsp *doubleselectTag.jsp* para ver cómo se emplea esta etiqueta.

```
<%@ taglib prefix="s" uri="/struts-tags" %>

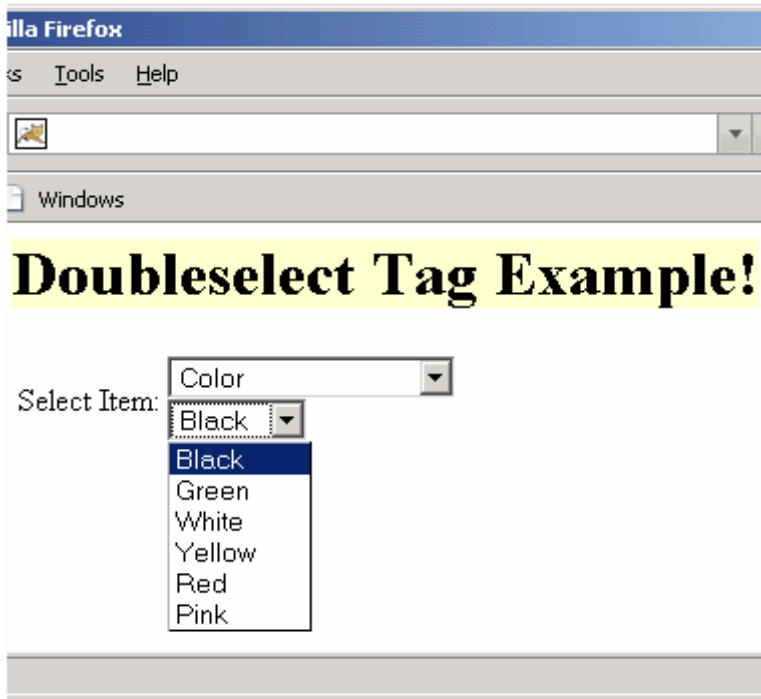
<html>
  <head>
    <title>Doubleselect Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-
color: #FFFFcc">Doubleselect Tag Example!</span></h1>
    <s:form>
      <s:doubleselect label="Select Item"
        headerValue="--- Please Select ---"
        headerKey="1" list="{ 'Color', 'Fruits' }"
        doubleName="dishes"
        doubleList="top == 'Color' ? { 'Black', 'Green', 'White',
          'Yellow', 'Red', 'Pink' } : { 'Apple', 'Banana', 'Grapes', 'Mango' }" />
      </s:form>
    </body>
  </html>
```

La jsp que hemos creado utiliza el tag `<s:doubleselect>` la cual renderiza los dos select HTML, mostrando los valores del segundo en función de la selección del primero. El tag contiene los siguientes parámetros:

- **headerKey**: establece la clave de la cabecera de la segunda lista.
- **headerValue**: establece el valor de la cabecera de la segunda lista.
- **doubleName**: establece el nombre de la segunda lista e identifica el campo que se enviará en un submit.
- **doubleList**: establece las reglas de publicación de la segunda lista.

Como se puede ver la lista primera tiene dos valores y en el atributo **doubleList** hemos puesto la regla para cambiar de lista. Como se puede ver dependiendo de un valor o del otro de la lista primera se muestran colores o frutas.

Gráficamente se ve:



### Tag file

Muestra un campo file de HTML. Para facilitarnos la vida podemos aprovechar el interceptor `fileUpload` que se encuentra en la selección de interceptores por defecto (`defaultStack`) y que funciona de forma parecida al interceptor `param`. Basta crear setters y getters en la acción para las nuevas propiedades `nombre` (el archivo en sí), `nombreContentType` (el tipo MIME del archivo subido) o `nombreFileName` (el nombre del archivo subido) para tener acceso a estos valores en la acción.

Éste tag tiene un atributo especial que se llama `accept` que define los tipos MIME que acepta el campo.

Veamos un ejemplo:

```
<%@ taglib uri="/struts-tags" prefix="s"%>

<html>
<body>

<s:form enctype="multipart/form-data" method="POST" action="Accion">
    <s:file label="Archivo a enviar" name="archivoTexto"
        accept="text/txt" />
    <s:submit value="Enviar" />
</s:form>

</body>
</html>
```

En la jsp tenemos definido el formulario con el file.

El action es como sigue:

```
import com.opensymphony.xwork2.ActionSupport;
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

@SuppressWarnings("serial")
public class Accion extends ActionSupport {
    private File archivoTexto;
    private String archivoTextoContentType;
    private String archivoTextoFileName;
    private String contenido;

    public String getContenido() {
        return contenido;
    }

    public File getArchivoTexto() {
        return archivoTexto;
    }

    public void setArchivoTexto(File archivoTexto) {
        this.archivoTexto = archivoTexto;
    }

    public String getArchivoTextoContentType() {
        return archivoTextoContentType;
    }

    public void setArchivoTextoContentType(String archivoTextoContentType) {
        this.archivoTextoContentType = archivoTextoContentType;
    }

    public String getArchivoTextoFileName() {
        return archivoTextoFileName;
    }

    public void setArchivoTextoFileName(String archivoTextoFileName) {
        this.archivoTextoFileName = archivoTextoFileName;
    }

    public String execute() throws IOException {
        BufferedReader input = new BufferedReader(new FileReader(archivoTexto));

        String linea = "";
        contenido = "";
        while ((linea = input.readLine()) != null)
            contenido = contenido + linea;

        return SUCCESS;
    }
}
```

Como podemos ver en el action tenemos que utilizar los datos que se han informado para `archivoTexto` para hacer el guardado en un directorio físico. Cuando subimos un file tendremos 3 propiedades que se nos envía, que son el fichero, el nombre del fichero y el *content-type*. Utilizando los 3 podremos implementar la lógica que sea requerida.

### Tag form

Crea un elemento form de HTML. Presenta los siguientes atributos:



- action: Acción a la que se enviará la petición con los datos del formulario. También se puede enlazar otras páginas o servlets. Si no utilizamos el atributo para especificar el destino se utiliza la misma página del formulario.
- namespace: Espacio de nombres al que pertenece la acción a la que se enviará la petición. Por defecto se utiliza el espacio de nombres actual.
- validate: Si queremos validar los campos del formulario antes de enviarlos.

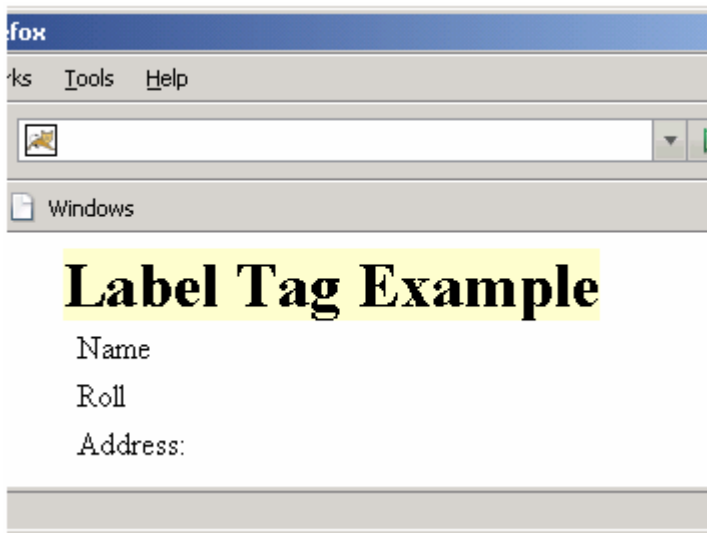
## Tag Label

Crea una etiqueta `<label>` de HTML.

Como ejemplo la jsp siguiente:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
    <title>Label Tag Example!</title>
  </head>
  <body>
    <h1><span style="background-color: #FFFFcc">Label Tag Example</span></h1>
    <s:form>
      <s:label name="name" value= "Name" />
      <s:label name="roll" value= "Roll" />
      <s:label name="address" value= "Address: " />
    </s:form>
  </body>
</html>
```

Generará:



## Tag Optiontransferselect

Crea un componente consistente en dos selects cuyos elementos pueden traspasarse de uno a otro.

### Atributos:

- doubleList (requerido): El iterador con los valores que tendrá el segundo select al comenzar.
- doubleName (requerido): Nombre a utilizar para el elemento.
- leftTitle: Título del primer select.
- list: El iterador con los valores que tendrá el primer select al comenzar.
- rightTitle: Título del segundo select.

Adjuntamos el *action.java* y la *jsp* para ver cómo se emplean.

```
import com.opensymphony.xwork2.ActionSupport;

import java.util.HashMap;

@SuppressWarnings("serial")
public class Accion extends ActionSupport {
    private HashMap<String, String> registrados;
    private HashMap<String, String> vips;

    public HashMap<String, String> getVips() {
        return vips;
    }

    public void setVips(HashMap<String, String> vips) {
        this.vips = vips;
    }

    public HashMap<String, String> getRegistrados() {
        return registrados;
    }

    public void setRegistrados(HashMap<String, String> registrados) {
        this.registrados = registrados;
    }

    public String execute() {
        registrados = new HashMap<String, String>();
        registrados.put("Juan", "Juan Encina");
        registrados.put("Manuel", "Manuel Robledo");

        vips = new HashMap<String, String>();
        vips.put("Pedro", "Pedro Peral");
        vips.put("María", "María Manzano");

        return SUCCESS;
    }
}
```

La *jsp* sería:

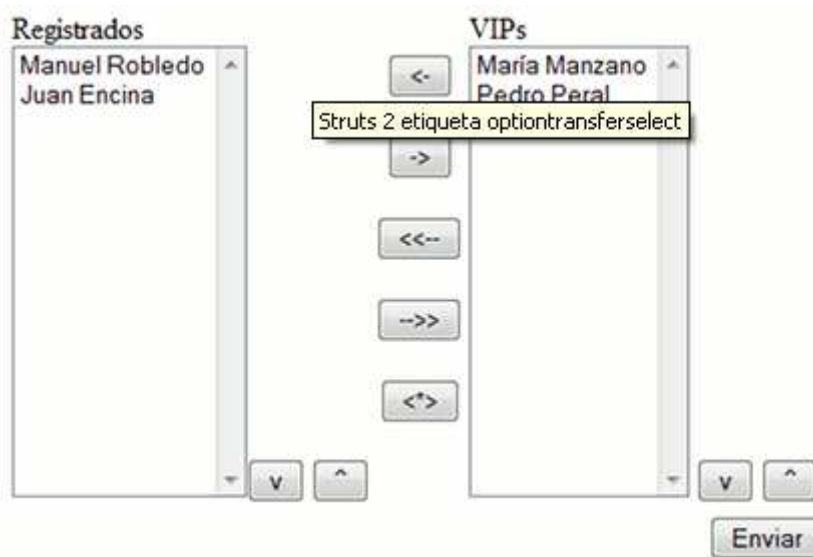
```
<%@ taglib uri="/struts-tags" prefix="s"%>

<html>
<body>

<s:form action="Otro">
    <s:optiontransferselect list="registrados" leftTitle="Registrados"
        doubleList="vips" rightTitle="VIPs" doubleName="usuarios" />
    <s:submit value="Enviar" />
</s:form>
```

```
</body>  
</html>
```

Gráficamente quedaría:



### Tag optgroup

Crea un nuevo elemento HTML `optgroup` (un grupo de opciones para un elemento `select`) utilizando una lista que se pasa como parámetro.

El action sería:

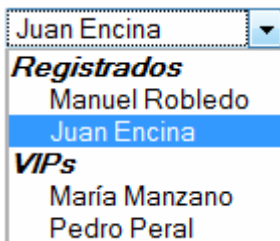
```
import com.opensymphony.xwork2.ActionSupport;  
  
import java.util.HashMap;  
  
@SuppressWarnings("serial")  
public class Accion extends ActionSupport {  
    private HashMap<String, String> registrados;  
    private HashMap<String, String> vips;  
  
    public HashMap<String, String> getVips() {  
        return vips;  
    }  
  
    public void setVips(HashMap<String, String> vips) {  
        this.vips = vips;  
    }  
  
    public HashMap<String, String> getRegistrados() {  
        return registrados;  
    }  
  
    public void setRegistrados(HashMap<String, String> registrados) {  
        this.registrados = registrados;  
    }  
  
    public String execute() {  
        registrados = new HashMap<String, String>();  
        registrados.put("Juan", "Juan Encina");  
        registrados.put("Manuel", "Manuel Robledo");  
  
        vips = new HashMap<String, String>();  
        vips.put("Pedro", "Pedro Peral");  
    }  
}
```

```
        vips.put("María", "María Manzano");  
        return SUCCESS;  
    }  
}
```

Un ejemplo de uso sería:

```
<%@ taglib uri="/struts-tags" prefix="s"%>  
  
<html>  
<body>  
  
<s:form>  
    <s:select list="{ }">  
        <s:optgroup label="Registrados" list="registrados" />  
        <s:optgroup label="VIPs" list="vips" />  
    </s:select>  
    <s:submit value="Enviar" />  
</s:form>  
  
</body>  
</html>
```

Y el resultado:



### Tag password

Este tag renderiza una caja para meter passwords creando el tag *input* asociado.

Para verlo en ejecución adjuntamos el código jsp siguiente:

```
<%@ taglib prefix="s" uri="/struts-tags" %>  
<html>  
    <head>  
        <title>Password Tag Example!</title>  
    </head>  
    <body>  
        <h1><span style="background-color: #FFFFcc">Password Tag Example!</span></h1>  
        <s:form>  
            <s:password label="Enter Password" name="password" size="10" maxlength="8"  
            />  
        </s:form>  
    </body>  
</html>
```

Que renderizará:



### Tag radio

Esta tag renderiza radio buttons. Para ello utiliza una lista con las diferentes opciones que tiene que tratar.

Código de ejemplo:

*checkboxlistTag.java*

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.*;

public class checkboxlistTag extends ActionSupport{

    private List fruits;
    private List animals;
    public String execute()throws Exception{
        fruits = new ArrayList();
        fruits.add("Apple");
        fruits.add("Mango");
        fruits.add("Orange");
        fruits.add("Pine Apple");

        animals = new ArrayList();
        animals.add("Dog");
        animals.add("Elephant");
        animals.add("Ox");
        animals.add("Fox");
        return SUCCESS;
    }

    public List getFruits(){
        return fruits;
    }

    public List getAnimals(){
        return animals;
    }
}
```

Y la jsp:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
    <title>Radio Tag Example!</title>
    <link href="<s:url value="/css/main.css"/>" rel="stylesheet"
          type="text/css"/>
  </head>
  <body>
    <h1><span style="background-color: #FFFFcc">Radio Tag Example!</span></h1>
    <s:form>
      <s:radio label="Fruits" name="fruitsname" list="fruits" />
      <s:radio label="Animals" name="animalsname" list="animals" />
    </s:form>
  </body>
</html>
```

Gráficamente obtenemos dos colecciones independientes de combos.



### Tag reset

Crea un botón reset de HTML.

### Tag select

Crea un elemento select de HTML utilizando una lista para generar los valores que se van a utilizar.

Lo vemos en el siguiente ejemplo:

### Accion.java

```
import com.opensymphony.xwork2.ActionSupport;
import java.util.List;
import java.util.ArrayList;

@SuppressWarnings("serial")
public class Accion extends ActionSupport {
    private List<String> lenguajes;

    public List<String> getLenguajes() {
```

```
        return lenguajes;
    }

    public String execute() {
        lenguajes = new ArrayList<String>();
        lenguajes.add("Python");
        lenguajes.add("Java");
        lenguajes.add("Ruby");
        lenguajes.add("C#");
        lenguajes.add("C++");
        lenguajes.add("Lisp");
        return SUCCESS;
    }
}
```


La jsp será:

```
<%@ taglib uri="/struts-tags" prefix="s"%>
<html>
<body>

<s:form action="Otro">
    <s:select label="Lenguaje preferido" list="lenguajes" />
    <s:submit value="Enviar" />
</s:form>

</body>
</html>
```

Gráficamente obtenemos:

Lenguaje preferido:  

También podemos hacer que la lista tenga pares clave, valor como en el código siguiente:

```
<s:select label="Select Month"
    name="monthname"
    headerKey="1"
    headerValue="-- Please Select --"
    list="#{'01':'January','02':'February','03':'March','04':'April',
'05':'May','06':'June','07':'July','08':'August','09':'September','10':
'October','11':'November','12':'December'}"
/>
```

Como se ve, puede tomar la lista con valores separados por ':' para identificar la clave y el valor.

### Tag submit

El tag submit va a crear un botón para hacer submit en el formulario que se crea. Este tag se puede aderezar con una imagen u otros.

### Tag textarea

Crea un *textarea* de HTML.

## Tag textfield

El tag `textfield` crea input tipo texto de HTML. Siempre lleva el *name* que se emplea para cargar el valor del campo si es que se encuentra en alguna propiedad del *ValueStack*, y para determinar cómo se llamará el parámetro que se envíe cuando se haga un submit del formulario. Como caja de texto se pueden añadir *maxlength* para determinar el tamaño máximo que tendrá , *size* que dará largo a la caja.

## Tag Updownselect

El tag `updownselect` crea un `select` de HTML con botones para mover arriba y abajo los elementos del mismo. Cuando se hace un submit del formulario que contiene este component, los valores se enviarán en el mismo orden que aparezca en el select tras hacer los movimientos.

Veamos un ejemplo:

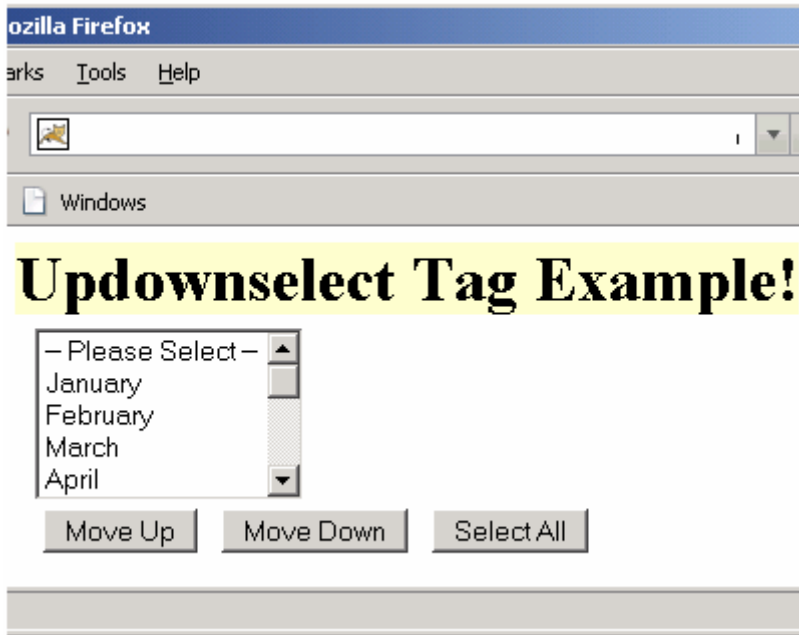
```
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
  <head>
    <title>Updownselect Tag Example</title>

  </head>
  <body>
    <h1><span style="background-
color: #FFFFcc">Updownselect Tag Example!</span></h>
    <s:form>
      <s:updownselect
        list="#{'01':'January','02':'February','03':'March','04':'April','05':'May',
'06':'June','07':'July','08':'August','09':'September','10':'October','11':'No
vember',
'12':'December'}"
        name="daysname"
        headerKey="1"
        headerValue="-- Please Select --"
        moveUpLabel="Move Up"
        moveDownLabel="Move Down"
        selectAllLabel="Select All"
      />
    </s:form>
  </body>
</html>
```

Esta jsp presentará gráficamente:





### Tags `Actionerror` y `Actionmessage`

Estos tags se emplean para mostrar errores de validación en formularios. Se pueden ver en el capítulo 6.

### Tag `fielderror`

Muestra los errores que se han detectado al validar los campos del formulario. Por defecto se muestran todos los errores, pero podemos seleccionar que se muestren sólo los relativos a ciertos campos pasándole etiquetas `param`. El funcionamiento es parecido al de `actionerror` y `actionmessage`: podemos añadir errores utilizando el método `addFieldError(String fieldName, String errorMessage)` de la interfaz `ValidationAware`, que `ActionSupport` implementa. Hay que tener en cuenta, no obstante, que si redirigimos al formulario de entrada el propio formulario imprimirá estos errores por defecto sin necesidad de añadir esta etiqueta. Los métodos y propiedades de `ValidationAware` también estarán disponibles en la vista, por lo que es posible en su lugar comprobar si existen errores con el método `hasFieldErrors()` e iterar directamente sobre la colección `fieldErrors`.

Veamos un ejemplo

*Index.jsp*

```
<%@ taglib uri="/struts-tags" prefix="s"%>

<html>
<body>
```

```
<s:fielderror />

<s:form action="Accion">
    <s:textfield label="Nombre" name="nombre" />
    <s:submit value="Enviar" />
</s:form>

</body>
</html>
```

### *Accion.java*

```
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Accion extends ActionSupport {
    private String nombre;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String execute() {
        if(nombre.length() > 10) {
            addFieldError("nombre", "El nombre no puede tener más de 10 caract
eres.");
            return ERROR;
        }

        return SUCCESS;
    }
}
```