

Struts 2 - Parte 1: Configuración

Cuando desarrollamos aplicaciones web es importante que estas puedan ser creadas de forma rápida y eficiente. Hoy en día existen muchos frameworks, los cuales nos proporcionan un cascarón para las aplicaciones. Nosotros solo debemos implementar la lógica propia de la aplicación, configurar el framework mediante algún mecanismo [como](#) anotaciones o archivos XML, y estos se encargan de hacer la mayor parte del trabajo tedioso o repetitivo.

Struts 2 es un framework para el desarrollo de aplicaciones web, el cual hace [que](#) la implementación de las mismas sea más sencillo, más rápido, y con menos complicaciones. Además hace que estas sean más robustas y flexibles. El objetivo de **Struts 2** es muy sencillo: **hacer que el desarrollo de aplicaciones web sea simple para los desarrolladores**.

En esta serie de tutoriales veremos cómo desarrollar aplicaciones usando este framework web, cómo configurar el controlador que implementa **Struts 2**, y las distintas opciones que nos ofrece.

Struts 2 es un framework de presentación, dentro de las capas en las que se divide una aplicación en la arquitectura **JEE**, el cual implementa el controlador del patrón de diseño [MVC \(Modelo Vista Controlador\)](#), y que podemos configurar de varias maneras; además proporciona algunos componentes para la capa de vista. [Por](#) si fuera poco, proporciona una integración perfecta con otros frameworks para implementar la capa del modelo (como [Hibernate](#) y [Spring](#)).

Para hacer más fácil presentar datos dinámicos, el framework incluye una biblioteca de etiquetas web. Las etiquetas interactúan con las validaciones y las características de internacionalización del framework, para asegurar que las entradas son válidas, y las salidas están localizadas. La biblioteca de etiquetas puede ser usada con **JSP**, **FreeMarker**, o **Velocity**; también pueden ser usadas otras bibliotecas de etiquetas como **JSTL** y soporta el uso de componentes **JSF**.

Además permite agregarle funcionalidades, mediante el uso de plugins, de forma transparente, ya que los plugins no tienen que ser declarados ni configurados de ninguna forma. Basta con agregar al **classpath** el jar que contiene al plugin, y [eso](#) es todo.

Como dije antes: el objetivo de **Struts 2** es hacer que el desarrollo de aplicaciones web sea fácil para los desarrolladores. Para lograr esto, **Struts 2** cuenta con características que permiten reducir la configuración gracias a que proporciona un conjunto inteligente de valores por default. Además hace uso de anotaciones y proporciona una forma de hacer la configuración de manera automática si usamos una serie de convenciones (y si hacemos uso de un plugin especial).

Struts 2 no es precisamente el heredero de **Struts 1**, sino que es la mezcla de dos frameworks: **WebWork 2** y **Struts** (aunque en realidad me parece que de **Struts 1** solo tomó algunos nombres ^_^).

Componentes de Struts 2

Comencemos hablando un poco de los componentes que forman a **Struts 2**.

El corazón de **Struts 2** es un filtro, conocido como el "**FilterDispatcher**". Este es el punto de entrada del framework. A partir de él se lanza la ejecución de todas las peticiones que involucran al framework.

Las principales responsabilidades del "**FilterDispatcher**" son:

- Ejecutar los **Actions**, que son los manejadores de las peticiones.
- Comenzar la ejecución de la cadena de interceptores (de la que hablaremos en un momento).
- Limpiar el "**ActionContext**", para evitar fugas de memoria.

Struts 2 procesa las peticiones usando tres elementos principales:

- Interceptores
- Acciones
- Resultados

Interceptores

Los interceptores son clases que siguen [el patrón interceptor](#). Estos permiten que se implementen funcionalidades cruzadas o comunes para todos los **Actions**, pero que se ejecuten fuera del **Action** (por ejemplo validaciones de datos, conversiones de tipos, población de datos, etc.).

Los interceptores realizan tareas **antes y después** de la ejecución de un **Action** y también pueden evitar que un **Action** se ejecute (por ejemplo si estamos haciendo alguna validación que no se ha cumplido).

Sirven para ejecutar algún proceso particular que se quiere aplicar a un conjunto de **Actions**. De hecho muchas de las características con que cuenta **Struts 2** son proporcionadas por los interceptores.

Si alguna funcionalidad que necesitamos no se encuentra en los interceptores de **Struts** podemos crear nuestro **propio interceptor** y agregarlo a la cadena que se ejecuta por default.

De la misma forma, podemos modificar la cadena de interceptores de **Struts**, por ejemplo para quitar un interceptor o modificar su orden de ejecución.

La siguiente tabla muestra solo algunos de los interceptores más importantes que vienen integrados y pre-configurados en **Struts 2**:

| Interceptor | Nombre | Descripción |
|------------------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------|
| Alias | alias | Permite que los parámetros tengan distintos nombres entre peticiones. |
| Chaining | chaining | Permite que las propiedades del Action ejecutado previamente estén disponibles en el Action actual |
| Checkbox | checkbox | Ayuda en el manejo de checkboxes agregando un parámetro con el valor " false " para checkboxes que no están marcadas (o checadas) |
| Conversion Error | conversionError | Coloca información de los errores convirtiendo cadenas a los tipos de parámetros adecuados para los campos del Action . |
| Create Session | createSession | Crea de forma automática una sesión HTTP si es que aún no existe una. |
| Execute and Wait | execAndWait | Envía al usuario a una página de espera intermedia mientras el Action se ejecuta en background. |
| File Upload | fileUpload | Hace que la carga de archivos sea más fácil de realizar. |
| Logging | logger | Proporciona un logging (salida a bitácora) simple, mostrando el nombre del Action que se está ejecutando. |

| Interceptor | Nombre | Descripción |
|-----------------------|---------------|-------------------------------------------------------------------------------------------------------|
| Parameters | params | Establece los parámetros de la petición en el Action . |
| Prepare | prepare | Llama al método " prepare " en los acciones que implementan la interface " Preparable " |
| Servlet Configuration | servletConfig | Proporciona al Action acceso a información basada en Servlets . |
| Roles | roles | Permite que el Action se ejecutado solo si el usuario tiene uno de los roles configurados. |
| Timer | timer | Proporciona una información sencilla de cuánto tiempo tardo el Action en ejecutarse. |
| Validation | validation | Proporciona a los Actions soporte para validaciones de datos. |
| Workflow | workflow | Redirige al result " INPUT " sin ejecutar el Action cuando una validación falla. |

Cada interceptor proporciona una característica distinta al **Action**. Para sacar la mayor ventaja posible de los interceptores, un **Action** permite que se aplique más de un interceptor. Para lograr esto **Struts 2** permite crear pilas o **stacks de interceptores** y aplicarlas a los **Actions**. Cada interceptor es aplicado en el orden en el que aparece en el stack. También podemos formar pilas de interceptores en base a otras pilas ^_^.

Una de estas pilas de interceptores es aplicada por default a todos los **Actions** de la aplicación (aunque si queremos, también podemos aplicar una pila particular a un **Action**).

La siguiente tabla muestra algunos de los stacks de interceptores que vienen pre-configurados con **Struts 2**:

| Nombre del Stack | Interceptores Incluidos | Descripción |
|---------------------------------|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| basicStack | exception, servletConfig, prepare, checkbox, multiselect, actionMappingParams, params, conversionError | Los interceptores que se espera se usen en todos los casos, hasta los más básicos. |
| validationWorkflowStack | basicStack, validation, workflow | Agrega validación y flujo de trabajo a las características del stack básico. |
| fileUploadStack | fileUpload, basicStack | Agrega funcionalidad de carga de archivos a las características del stack básico. |
| paramsPrepareParamsStack | alias, i18n, checkbox, multiselect, params, | Proporciona un stack completo para manejo de |

| Nombre del Stack | Interceptores Incluidos | Descripción |
|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| | servletConfig, prepare, chain, modelDriven, fileUpload, staticParams, actionMappingParams, params, conversionError, validation, workflow | casi cualquier cosa que necesitemos en nuestras aplicaciones. El interceptor " params " se aplica dos veces, la primera vez proporciona los parámetros antes de que el método " prepare " sea llamado, y la segunda vez re-aplica los parámetros a los objetos que hayan podido ser recuperados durante la fase de preparación. |
| defaultStack | alias, servletConfig, i18n, prepare, chain, debugging, scopedModelDriven, modelDriven, fileUpload, checkbox, multiselect, staticParams, actionMappingParams, params, conversionError, validation, workflow | Es la pila que se aplica por default a todos los Actions de la aplicación. |
| executeAndWaitStack | execAndWait, defaultStack, execAndWait | Proporciona al stack básico las características de execute and wait, lo cual funciona para aplicaciones en las que deben subirse archivos que pueden tardar algún tiempo. |

Acciones

Las acciones o **Actions** son clases encargadas de realizar la lógica para servir una petición. Cada **URL es mapeada a una acción específica**, la cual proporciona la lógica necesaria para servir a cada petición hecha por el usuario.

Estrictamente hablando, las acciones no necesitan implementar una interface o extender de alguna clase base. El único requisito para que una clase sea considerada un **Action** es que debe tener **un método** que **no reciba argumentos** que **regrese ya sea un String o un objeto de tipo Result**. Por default el nombre de este método debe ser "**execute**" aunque podemos ponerle el nombre que queramos y posteriormente indicarlo en el archivo de configuración de **Struts**.

Cuando el resultado es un **String** (lo cual es lo más común), el **Result** correspondiente se obtiene de la configuración del **Action**. Esto se usa para generar una respuesta para el usuario, lo cual veremos un poco más adelante.

Los **Actions** pueden ser objetos java simples (**POJOs**) que cumplan con el requisito anterior, aunque también pueden implementar la interface "[com.opensymphony.xwork2.Action](#)" o extender una clase base que proporciona **Struts 2: "com.opensymphony.xwork2.ActionSupport"** (lo cual nos hace más sencilla su creación y manejo).

La interface **Action** proporciona un conjunto de constantes con los nombres de los **Results** más comunes. Esta interface luce de la siguiente forma:

```
public interface Action
{
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

    public String execute() throws Exception;
}
```

La clase "**ActionSupport**" implementa la interface "**Action**" y contiene una implementación del método "**execute()**" que regresa un valor de "**SUCCESS**". Además proporciona unos cuantos métodos para establecer mensajes, tanto de error como informativos, que pueden ser mostrados al usuario.

Results

Después que un **Action** ha sido procesado se debe enviar la respuesta de regreso al usuario, esto se realiza usando **results**. Este proceso tiene dos componentes, el tipo del **result** y el **result** mismo.

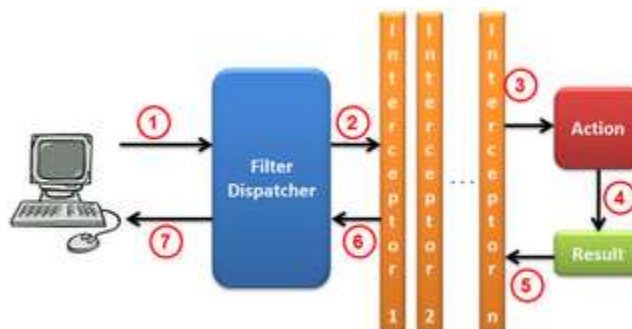
El tipo del **result** indica **cómo debe ser tratado** el resultado que se le regresará al cliente. Por ejemplo un tipo de **Result** puede enviar al usuario de vuelta una **JSP** (lo que haremos más a menudo), otro puede redirigirlo hacia otro sitio, mientras otro puede enviarle un flujo de bytes (para descargar un archivo por ejemplo).

Entraremos en más detalles de los tipos de resultados de **Struts 2** en otro tutorial. Por ahora solo hay que saber que el tipo de result por default es "**dispatcher**".

Un **Action** puede tener más de un **result** asociado. Esto nos permitirá enviar al usuario a una vista distinta dependiendo del resultado de la ejecución del **Action**. Por ejemplo en caso de que todo salga bien, enviaremos al usuario al **result "sucess"**, si algo sale mal lo enviaremos al **result "error"**, o si no tiene permisos lo enviaremos al **result "denied"**.

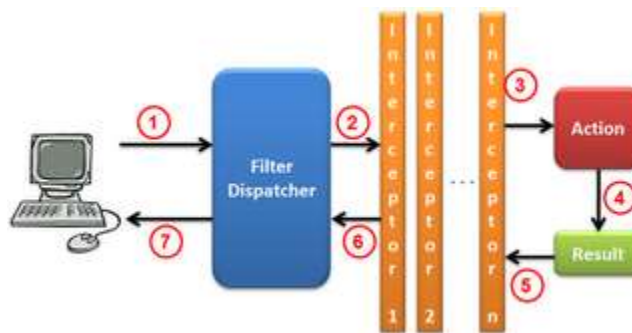
Funcionamiento de Struts 2

Ahora que conocemos a grandes rasgos los componentes que forman **Struts 2**, veamos de forma general cómo son procesadas las peticiones por una aplicación desarrollada en **Struts 2**. La siguiente imagen nos ayudará con esto:



Los pasos que sigue una petición son:

1. **1** - El navegador web hace una petición para un recurso de la aplicación (**index.action**, **reporte.pdf**, etc.). El filtro de **Struts** (al cual llamamos **FilterDispatcher** aunque esto no es del todo correcto, retomaré esto un poco más adelante) revisa la petición y determina el **Action** apropiado para servirla.
2. **2** - Se aplican los interceptores, los cuales realizan algunas funciones como validaciones, flujos de trabajo, manejo de la subida de archivos, etc.
3. **3** - Se ejecuta el método adecuado del **Action** (por default el método "**execute**"), este método usualmente almacena y/o regresa alguna información referente al proceso.
4. **4** - El **Action** indica cuál **result** debe ser aplicado. El **result** genera la salida apropiada dependiendo del resultado del proceso.
5. **5** - Se aplican al resultado los mismos interceptores que se aplicaron a la petición, pero en orden inverso.
6. **6** - El resultado vuelve a pasar por el **FilterDispatcher** aunque este ya no hace ningún proceso sobre el resultado (por definición de la especificación de **Servlets**, si una petición pasa por un filtro, su respuesta asociada pasa también por el mismo filtro).
7. **7** - El resultado es enviado al usuario y este lo visualiza.



(Vuelvo a poner la imagen para que no tengan que regresar para verla ^_^).

Este camino podría no siempre seguirse en orden. Por ejemplo, si el interceptor de validación de datos encuentra que hay algún problema, el **Action** no se ejecutará, y será el mismo interceptor el que se encargará de enviar un **Result** al usuario.

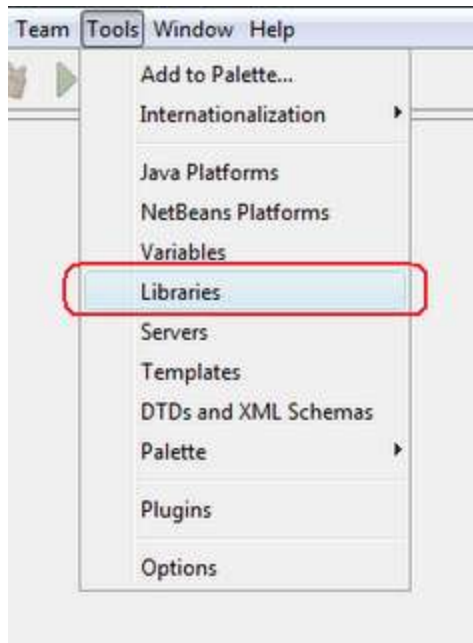
Otra característica importante de **Struts 2** es que no hace uso del lenguaje de expresiones propio de las **JSPs**, sino que hace uso de **OGNL (Object-Graph Navigation Language)**, un lenguaje de expresiones más poderoso. Hablaremos más de **OGNL** en el siguiente tutorial.

Comencemos viendo un pequeño ejemplo que nos mostrará cómo configurar nuestras aplicaciones con **Struts 2**. Haremos una configuración haciendo uso de un archivo de configuración en **XML** y otro haciendo uso de anotaciones.

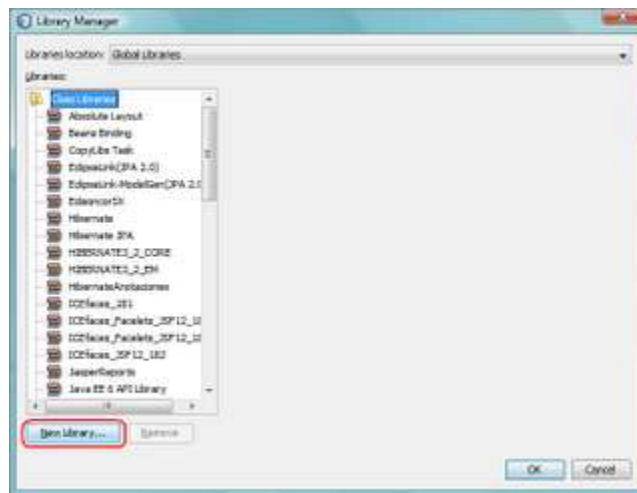
Nota: Yo usaré dos proyectos, uno con un archivo XML y otro con anotaciones, para que el código de ambos no se mezcle.

Lo primero que haremos es crear una biblioteca llamada "**Struts2**". Para esto debemos descargar la última versión de **Struts 2** (actualmente la **2.2.3**) del [sitio de descargas de Struts 2](#). Podemos descargar cualquiera de los paquetes. Yo les recomiendo que bajen el "**lib**" o el paquete "**all**".

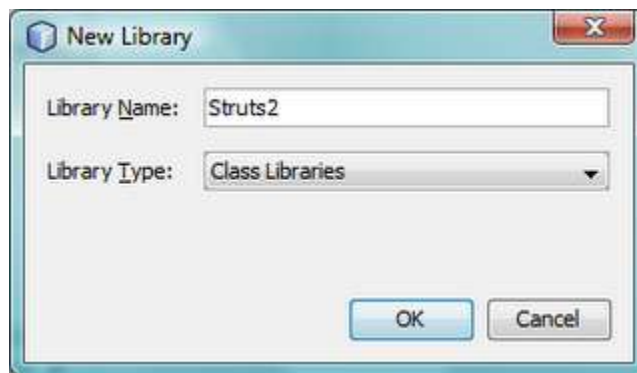
Una vez que tengamos el archivo, vamos al **NetBeans** y nos dirigimos al menú "**Tools -> Libraries**".



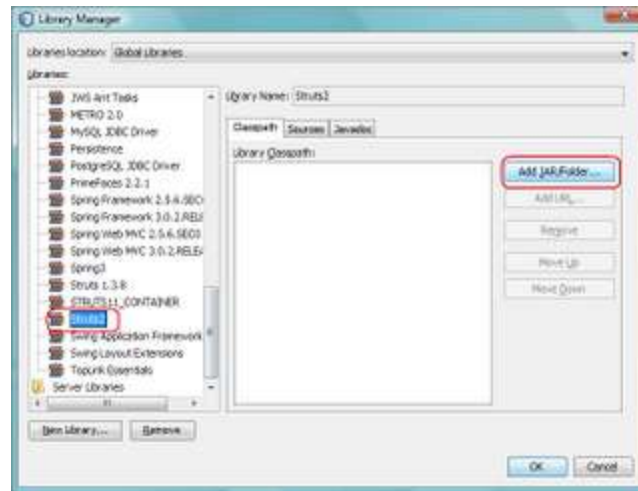
En la ventana que se abre presionamos el botón **"New Library..."**:



En esta nueva ventana colocamos como nombre de la biblioteca **"Struts2"** y como tipo dejamos **"Class Libraries"**:



Ahora que tenemos nuestra biblioteca, presionamos el botón **"Add Jar/Folder"** para agregar los nuevos archivos que conformarán la biblioteca:



Agregamos los siguientes archivos:

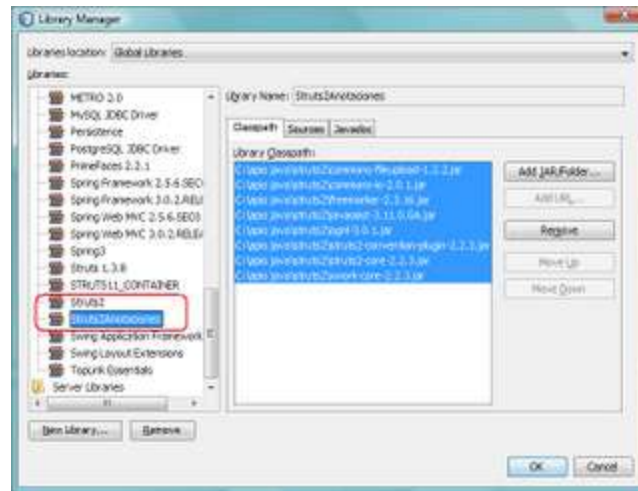
- **commons-fileupload-1.2.2.jar**
- **commons-io-2.0.1.jar**
- **commons-lang-2.5.jar**
- **freemarker-2.3.16.jar**
- **javassist-3.11.0.GA.jar**
- **ognl-3.0.1.jar**
- **struts2-core-2.2.3.jar**
- **xwork-core-2.2.3.jar**

Adicionalmente crearemos una segunda biblioteca para cuando trabajemos con anotaciones. Seguimos el mismo proceso para crear la biblioteca, que en este caso se llamará "**Struts2Anotaciones**". Esta biblioteca debe incluir los siguientes jars:

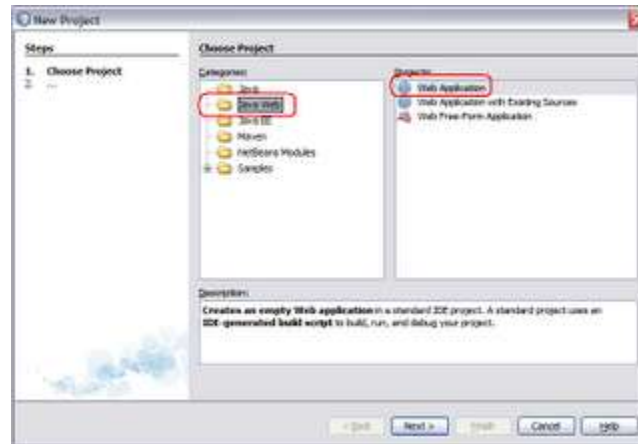
- **asm-3.1.jar**
- **asm-commons-3.1.jar**
- **commons-fileupload-1.2.2.jar**
- **commons-io-2.0.1.jar**
- **commons-lang-2.5.jar**
- **freemarker-2.3.16.jar**
- **javassist-3.11.0.GA.jar**
- **ognl-3.0.1.jar**
- **struts2-core-2.2.3.jar**
- **xwork-core-2.2.3.jar**
- **struts2-convention-plugin-2.2.3.jar**

O sea, todos los que tiene la biblioteca "**Struts 2**" + el jar "**asm-3.1.jar**" + el jar "**asm-commons-3.1.jar**" + el jar "**struts2-convention-plugin-2.2.3.jar**".

Ahora tenemos dos bibliotecas de **Struts 2**, una que usaremos cuando trabajemos con archivos de configuración en **XML**, y otra que usaremos cuando hagamos nuestra configuración con anotaciones:



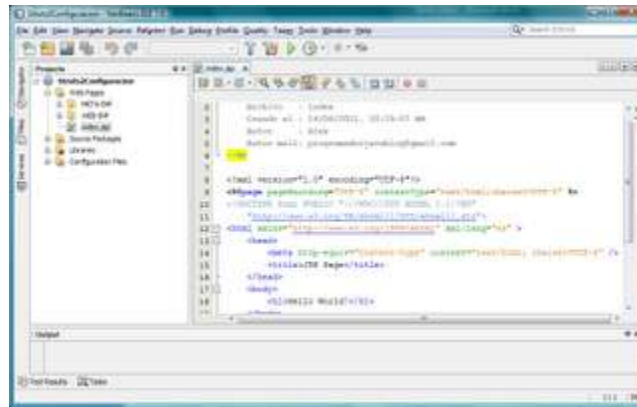
Lo siguiente es crear un nuevo proyecto **web** en **NetBeans**. Para esto vamos al Menú "**File->New Project...**". En la ventana que se abre seleccionamos la categoría "**Java Web**" y en el tipo de proyecto "**Web Application**":



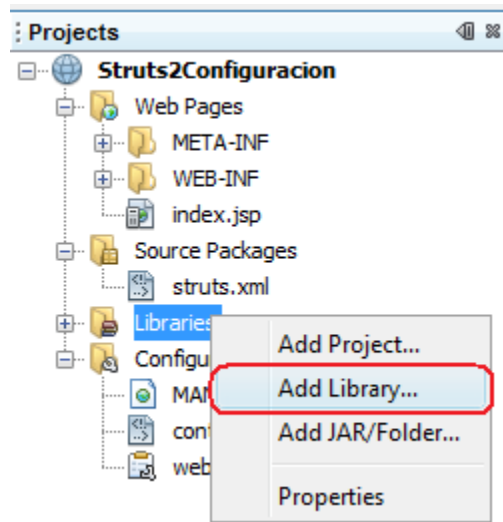
Le damos una ubicación y un nombre al proyecto, en mi caso será "**Struts2Configuracion**". Presionamos el botón "**Next**". En la siguiente pantalla deberemos configurar el servidor de aplicaciones que usaremos. Yo usaré la versión 7 de **Tomcat** (si no lo tienen configurado, pueden seguir los mismos pasos que en [el tutorial de instalación de plugins en NetBeans](#)). Como versión de **Java EE** usaré la **5** (pueden usar también la 6 si quieren, solo que en ese caso **NetBeans** no genera el archivo "**web.xml**" por default, y tendrán que crearlo a mano o usando el wizard correspondiente). De la misma forma, si quieren pueden modificar el **context path** de la aplicación (el **context path** es la ruta que se colocará en el navegador para acceder a nuestra aplicación desde nuestro servidor):



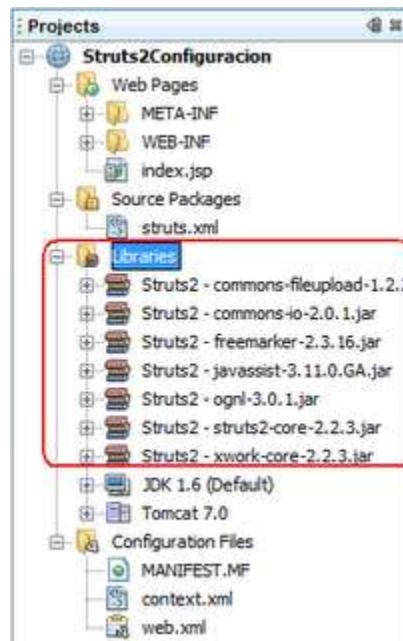
Presionamos el botón **"Finish"**, con lo que veremos aparecer la página **"index.jsp"** en nuestro editor.



Ahora agregaremos la biblioteca **"Struts2"** que creamos hace un momento. Para esto hacemos clic derecho en el nodo **"Libraries"** del panel de proyectos. En el menú que aparece seleccionamos la opción **"Add Library..."**:



En la ventana que aparece seleccionamos la biblioteca **"Struts2"** y presionamos **"Add Library"**. Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto:



Lo siguiente que debemos hacer es configurar el filtro de **Struts 2**, para que todas las llamadas que se hagan a nuestra aplicación web pasen a través de este filtro. El filtro de **Struts 2** es llamado "**StrutsPrepareAndExecuteFilter**". En versiones anteriores de **Struts 2** se usaba un filtro llamado "**FilterDispatcher**" (es por eso que algunas veces hago referencia a este filtro usando ese nombre), sin embargo este ha sido marcado como deprecated y por lo tanto ya no debe ser usado.

Abrimos el archivo de configuración de nuestra aplicación web, el deployment descriptor conocido sencillamente como el archivo "**web.xml**".

Para la declaración del filtro de **Struts 2**, debemos usar el elemento "**<filter>**", dentro del cual debemos indicar un nombre para el filtro, por convención usamos el nombre "**struts2**", y la clase que implementa dicho filtro, que en nuestro caso es "**org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter**".

Después será necesario indicar qué peticiones pasarán por el filtro de **Struts 2**. Para eso usamos el elemento "**<filter-mapping>**". Aquí indicamos que queremos que TODAS las peticiones que se hagan al sitio pasen a través del filtro, usando el patrón de URL "**/***".

La declaración y mapeo del filtro quedan de la siguiente forma:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

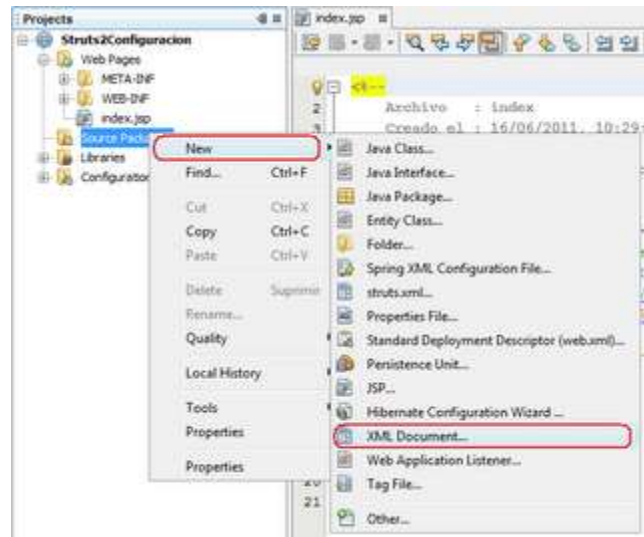
Ahora modificaremos la página "**index.jsp**", para probar que la configuración funciona correctamente. En esta página colocaremos solo un texto que diga: "**Bienvenido a Struts 2**".

El siguiente paso es realizar la configuración de **Struts 2**. Existen dos formas de hacer esto: la primera es usando un archivo de configuración en **XML**, y la segunda es mediante anotaciones. Primero veremos cómo configurarlo usando un archivo de configuración **XML**.

Configuración de Struts 2 Usando un Archivo XML

Para esta configuración debemos crear un archivo llamado "**struts.xml**", que **debe ser colocado en el paquete raíz de la aplicación**. En este archivo se especifica la relación entre la **URL** de una petición, la clase Java que ejecutará la acción de la petición, y la página **JSP** que se encargará de mostrar la respuesta a la petición del usuario.

Para crear el archivo hacemos clic derecho sobre el nodo "**Source Packages**" en el panel de proyectos. En el menú contextual que nos aparece seleccionamos la opción "**New -> XML Document**":



Le damos como nombre del archivo "**struts**", el wizard se encargará de colocarle la extensión ".xml":



Presionamos el botón "**Next**" y en la siguiente pantalla seleccionamos la opción "**Well-formed Document**". Presionamos el botón "**Finish**" y tendremos un archivo **XML**, al cual debemos quitarle todo el contenido.

Dentro de este archivo se configuran algunas constantes que modifican, de una manera mínima, el comportamiento del filtro de **Struts**. También se configuran paquetes, que son conjuntos de acciones que podemos organizar debajo de un mismo **namespace**. Dentro de los paquetes podemos definir un conjunto de acciones, cada una de las cuales responderá a una petición. Y dentro de las acciones podemos definir resultados, que son las páginas o vistas a las que se enviará al usuario dependiendo del resultado de una acción (como ya habíamos dicho: existen muchos tipos de resultados, no solo páginas **JSP**, por ejemplo podemos hacer que un usuario descargue un archivo, o que visualice un reporte o una gráfica; incluso es posible redirigir a un usuario a otra acción).

Este archivo de configuración debe iniciar con el elemento "**<struts>**":

```
<?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>

</struts>
```

Dentro de estas etiquetas se configuran los demás elementos.

Lo siguiente que haremos será configurar una constante que le indicará a **Struts** que nos encontramos en la etapa de desarrollo, con esto generará más mensajes de salida para que sepamos si estamos haciendo algo mal, y nos mostrará los errores de una forma más clara. Para esto establecemos la constante "**struts.devMode**" con un valor de "**true**":

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

Otra constante útil que podemos definir es "**struts.configuration.xml.reload**". Esta constante permite que cuando modifiquemos los archivos de configuración de **Struts 2** no tengamos que volver a hacer un deploy de la aplicación para que los cambios tomen efecto:

```
<constant name="struts.configuration.xml.reload" value="true" />
```

Hasta ahora nuestro archivo de configuración debe verse de la siguiente manera:

```
<?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.devMode" value="true" />
<constant name="struts.configuration.xml.reload" value="true" />

</struts>
```

Ahora definimos nuestro primer paquete, usando el elemento "**<package>**". Los paquetes nos permiten tener una configuración común para un conjunto de **Actions**. Por ejemplo, se puede definir el conjunto de interceptores que se aplican sobre ellos, resultados comunes, manejo de excepciones, etc.

Cada uno de los paquetes que declaremos debe tener un **nombre** y **extender de algún otro paquete**. Opcionalmente también pueden tener un **namespace**, el cual es un fragmento que deberá ser agregado a la **URL** de la petición para hacer referencia a ese paquete. Por ejemplo, si definimos un paquete con el namespace "**administracion**", y dentro de este hay un **Action** cuyo nombre es "**registro**", para ejecutar ese **Action** deberemos hacer una petición a la siguiente **URL**:

```
http://servidor:puerto/aplicacion/administracion/registro.action
```

Cuando extendemos de otro paquete **heredamos su configuración** (de hecho podemos tener paquetes abstractos con una configuración general, y hacer que varios paquetes extiendan de estos). Como en este momento no tenemos otro paquete, este debe extender de "**struts-default**", que es el paquete base para las aplicaciones de **Struts 2**. Como nombre de nuestro paquete colocaremos "**demo-struts**":

```
<package name="demo-struts" extends="struts-default">
</package>
```

Como podemos ver, no hemos indicado ningún **namespace** en nuestro paquete. Esto quiere decir que estamos usando el namespace por **default** (o sea ""). Cuando trabajamos con namespaces podemos usar principalmente 3 tipos: el namespace por **default** (como en este caso), el namespace **raíz** ("/") en cual en la **URL** de petición luce exactamente igual al namespace default, o un namespace propio. Si invocamos un **Action** usando un namespace distinto al default (como en el ejemplo anterior "**adminsitracion/registro.action**") y **Struts 2** no puede encontrar ningún **Action** que coincida con nuestra petición, en ese namespace, lo buscará en el namespace default. O sea, en el ejemplo anterior, si no encuentra el **Action** "**registro**" en el namespace "**administracion**", irá a buscarlo al namespace default.

Algo importante que hay que decir sobre los namespaces, es que estos **NO se comportar como nombres de directorios**. Esto quiere decir que si hacemos una petición a un **Action** en el namespace "**administracion/usuarios**", digamos: "**adminsitracion/usuarios/registro.action**" y **Struts 2** no encuentra el **Action** "**registro**" en ese namespace, irá a buscarlo al namespace por default, y no a un namespace "**adminsitracion**".

Ahora mapearemos nuestro primer **Action**. Para esto usamos el elemento "**<action>**". Todas las acciones deben tener un nombre que **debe ser único dentro de un namespace**. Las acciones deben estar asociadas con una clase que será la que ejecute la acción adecuada. Si no indicamos una clase, **Struts 2** usará una clase por default. Esta clase por default lo que hace es regresar siempre un resultado exitoso o "**SUCCESS**" de su ejecución. En este caso llamaremos a nuestra acción "**index**":

```
<action name="index">
</action>
```

Finalmente, debemos indicar el resultado de la ejecución de esta acción, con el elemento "**<result>**". Una acción puede tener varios resultados, dependiendo de la ejecución de la misma. Cada uno de estos resultados debe tener un nombre al que se hará referencia dentro de la acción para definir qué vista regresar al usuario. Se puede tener, por ejemplo, un resultado para el caso de una acción exitosa, otro para una situación de error, otra para cuando el usuario no ha ingresado todos los datos requeridos en un formulario, otra para cuando el usuario que ingresa es un administrador, otro para cuando es un usuario anónimo, etc.

Definimos que en el caso de una ejecución exitosa (**success**) se debe enviar al usuario a la página "**index.jsp**", de la siguiente forma (noten que la ruta de la página debe indicarse a partir de la raíz del directorio web):

```
<result name="success">/index.jsp</result>
```

El nombre por default de un **result** es "**success**", así que podemos dejar la línea anterior de la siguiente manera:

```
<result>/index.jsp</result>
```

Normalmente tratamos de usar los nombres de los **result** que se definen dentro de la interface **Action** ("**success**", "**error**", "**input**", etc.), para poder hacer uso de estas constantes cuando nos referimos a ellos, pero en realidad podemos darles el nombre que queramos (como "**exito**" o "**fallo**") y regresar estos valores en el método "**execute**".

Esto es todo, por lo que nuestro archivo de configuración queda de la siguiente forma:

```

<?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.devMode" value="true" />
<constant name="struts.configuration.xml.reload" value="true" />

<package name="demo-struts" extends="struts-default">
<action name="index">
<result>/index.jsp</result>
</action>
</package>
</struts>

```

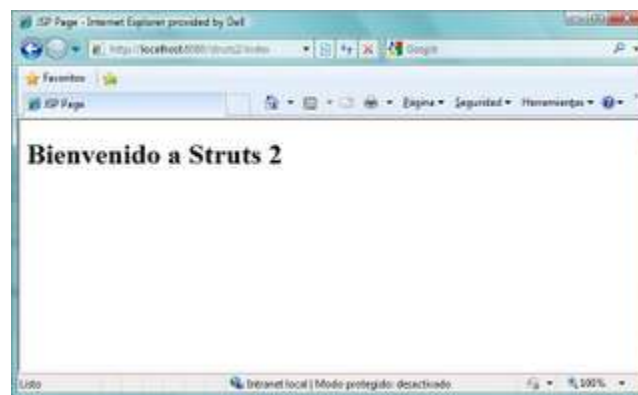
Ejecutamos la aplicación y entramos a la **URL**:

<http://localhost:8080/struts2/index.action>

Noten que estamos haciendo referencia al **Action "index"** mediante su nombre y agregando al final de este el postfijo **"action"**. **Struts 2** tiene configurados 2 postfijos que se usan por default para las acciones: **"action"** y **"**", por lo que también podemos hacer la petición a la siguiente **URL**:

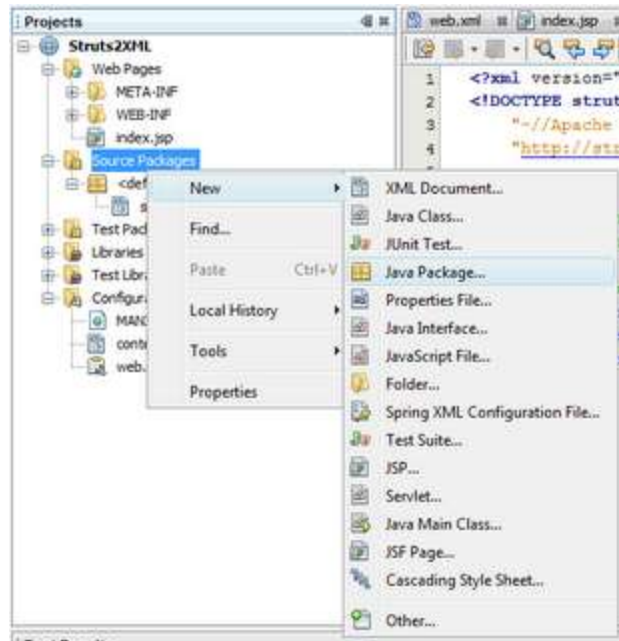
<http://localhost:8080/struts2/index>

Si todo está bien configurado deberemos ver una pantalla como la siguiente:

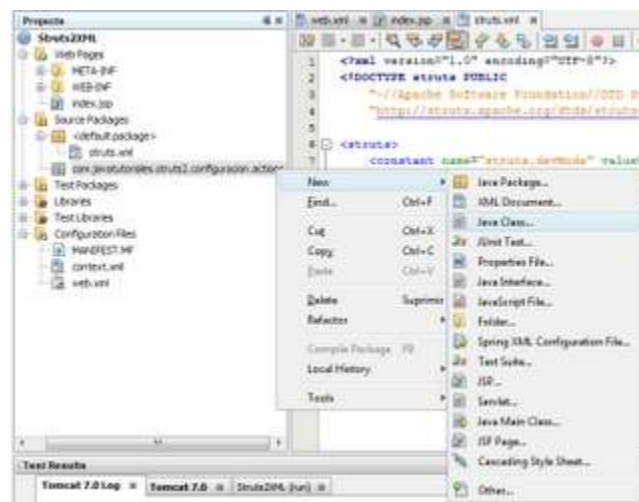


Con este ejemplo pudimos ver algunos de los detalles de configuración de **Struts 2**, sin embargo la salida es muy sencilla. Hagamos algo un poco más interesante: haremos que el mensaje de la página sea obtenido desde un **Action**.

Primero crearemos un nuevo paquete. Hacemos clic derecho en el nodo **"Source Packages"**, en el menú que aparece seleccionamos la opción **"New -> Java Package..."**:



El nombre de mi paquete será "**com.javatutoriales.struts2.configuracion.actions**". Dentro de este paquete crearemos una nueva clase. Hacemos clic derecho en el paquete que acabamos de crear, en el menú que aparece seleccionamos la opción "**New -> Java Class...**":



El nombre de la clase será "**SaludoAction**". En esta clase colocaremos un atributo de tipo **String** que se llame mensaje. También proporcionaremos un **getter** para este atributo:

```
public class SaludoAction
{
    private String mensaje;

    public String getMensaje()
    {
        return mensaje;
    }
}
```

Ahora crearemos el método de negocio que realiza el proceso propio de la acción. Para esto **Struts 2** siempre invoca un método llamado "**execute**" (aunque hay forma de modificar esto en el archivo de configuración). Este método debe regresar un **String**, que indica el nombre de **result** que debe ser mostrado al usuario.

En este caso lo único que hará nuestro método "**execute**" es establecer el valor del mensaje:

```
public String execute()  
{  
    mensaje = "Bienvenido al mundo de Struts 2";  
}
```

E indicaremos que se debe regresar el **result** cuyo nombre sea "**success**":

```
public String execute()  
{  
    mensaje = "Bienvenido al mundo de Struts 2";  
  
    return "success";  
}
```

Si se dan cuenta, estamos regresando el nombre del **result** usando una cadena literal ("**success**") esto parece sencillo en un principio, pero puede ser fuente de errores y dolores de cabeza si cometemos un error y escribimos, por ejemplo "**succes**", o "**sucess**". Como habíamos dicho antes, para simplificar esto (entre otras cosas que veremos más adelante) **Struts** proporciona una clase base de la que podemos extender. Esta clase es "**ActionSupport**", la cual implementa la interface "**Action**" que define las siguientes constantes para regresar los nombres de los **results** correspondientes:

- **ERROR**
- **INPUT**
- **LOGIN**
- **NONE**
- **SUCCESS**

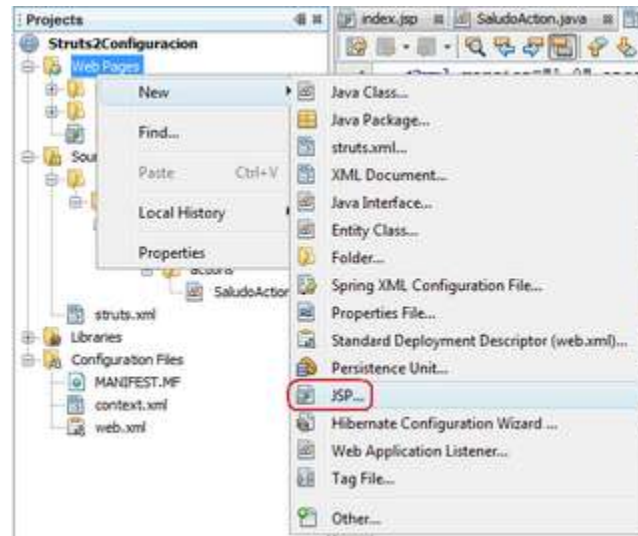
Modificamos nuestra clase para que quede de la siguiente forma:

```
public class SaludoAction extends ActionSupport  
{  
    private String mensaje;  
  
    @Override  
    public String execute()  
    {  
        mensaje = "Bienvenido al mundo de Struts 2";  
  
        return SUCCESS;  
    }  
  
    public String getMensaje()  
    {  
        return mensaje;  
    }  
}
```

Ahora debemos modificar el archivo de configuración. En este caso agregaremos un **Action** el cual responderá al nombre "**saludo**", y la clase que se encargará de ejecutar la lógica de esta acción será "**com.javatutoriales.struts2.configuracion.actions.SaludoAction**". En caso de un resultado exitoso (el único que se tiene) se mostrará al usuario la página "**saludo.jsp**", de esta forma:

```
<action name="saludo"
class="com.javatutoriales.struts2.configuracion.actions.SaludoAction">
<result>/saludo.jsp</result>
</action>
```

Crearemos la página "**saludo.jsp**" en el directorio raíz de las páginas web. Hacemos clic derecho en el nodo "**Web Pages**". En el menú que se abre seleccionamos la opción "**New -> JSP...**":



Damos "**saludo**" como nombre de la página y presionamos el botón "**Finish**".

Dentro de la página usaremos las etiquetas de **Struts** para mostrar el valor correspondiente a la propiedad del objeto. Indicamos que usaremos la biblioteca de etiquetas (taglib) "**/struts-tags**". Por convención usaremos el prefijo "**s**" para hacer referencia a esta biblioteca:

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

Dentro de esta biblioteca se encuentran contenidas todas las etiquetas de **Struts 2**. Tiene etiquetas control de flujo (**if**, **iterator**), de datos (**a**, **bean**, **i18n**), de formulario (**checkbox**, **form**, **label**, **submit**), para el manejo de componentes ajax (**autocomplete**, **tree**). En [el sitio de struts](#) pueden encontrar todas las etiquetas con las que cuenta **Struts 2**.

Usaremos una etiqueta de datos: "**property**". Dentro de esta etiqueta debemos indicar el nombre de la propiedad que queremos mostrar, en este caso "**mensaje**", usando su atributo "**value**":

```
<s:property value="mensaje" />
```

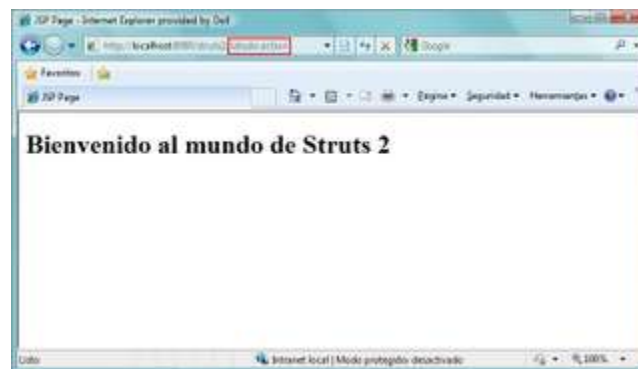
Nuestra página debe quedar de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<%@taglib uri="/struts-tags" prefix="s" %>
<%@page pageEncoding="UTF-8" contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="es" >
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>JSP Page</title>
</head>
<body>
<h1><s:property value="mensaje" /></h1>
</body>
</html>
```

Ahora, cuando entremos a la siguiente **URL**:

<http://localhost:8080/struts2/saludo.action>

Debemos ver el mensaje que establecimos en el **Action**:



Aumentaremos aún más el grado de complejidad del ejemplo y haremos que la página nos muestre un mensaje saludando al usuario, este proporcionará su nombre, junto con su número de la suerte, a través de un formulario.

Creamos una nueva página con el formulario, a esta página la llamaremos "**datos.jsp**", y la colocaremos en el directorio raíz de las páginas web.

En esta página importaremos la biblioteca de etiquetas de **Struts**, como lo habíamos hecho anteriormente:

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

Ahora crearemos un formulario con dos campos, uno para el nombre del usuario y otro para su número de la suerte. Para esto usaremos la etiqueta "**form**", la cual nos permite definir un formulario, y dentro de estas colocaremos los campos del mismo. En este formulario debemos colocar un atributo, "**action**", que indica el **Action** al que se deben enviar los datos para ser procesados, en este caso la acción se llamará "**saludoUsuario**", y lo crearemos en un momento:

```
<s:form action="saludoUsuario">
</s:form>
```

Dentro de estas etiquetas colocaremos tres componentes, dos campos de entrada de texto (**textfield**) y un botón para enviar la información (**submit**). **Struts 2** proporciona etiquetas para cada componente que podemos colocar en un formulario. Además proporciona distintos temas que pueden aplicarse a los sitios o a componentes específicos (**normal**, **xhtml**, **xhtml_css**, y **simple**). Hablaremos más de esto un poco más adelante. Gracias al tema por default (**xhtml**) podemos colocar en las etiquetas información con respecto al formulario, como por ejemplo las etiquetas (labels) de los componentes, o indicar si estos datos son requeridos, como atributos de las etiquetas de **Struts**:

```
<s:form action="saludoUsuario">
<s:textfield label="Nombre" name="nombre" />
<s:textfield label="Número de la suerte" name="numero" />
<s:submit value="Enviar" />
</s:form>
```

Podemos ver que en los campos de texto (los textfield) hemos colocado, en los atributos "**name**" correspondientes, los nombres "**nombre**" y "**numero**". Es con estos nombres que se hará referencia a las propiedades (o al menos a los **setters**) del **Action** que se encargará de procesar este formulario.

La página debe haber quedado de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<%@page pageEncoding="ISO-8859-1" contentType="text/html; charset=UTF-8" %>
<%@taglib uri="/struts-tags" prefix="s" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="es" >
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSP Page</title>
</head>
<body>

<s:form action="saludoUsuario">
<s:textfield label="Nombre" name="nombre" />
<s:textfield label="Número de la suerte" name="numero" />
<s:submit value="Enviar" />
</s:form>

</body>
</html>
```

Ahora crearemos el **Action** que se encargará de procesar estos datos para saludar al usuario. Creamos una nueva clase Java, en el paquete "**com.javatutoriales.struts2.configuracion.actions**", llamada "**SaludoUsuarioAction**" y hacemos que extienda de "**ActionSupport**":

```
public class SaludoUsuarioAction extends ActionSupport
{

}
```

A esta clase le agregamos dos atributos, un **String** llamado "**nombre**" que contendrá el nombre del usuario, y un **int** llamado "**numero**" que contendrá el número de la suerte del usuario. También agregamos sus **setters** correspondientes:

```

public class SaludoUsuarioAction extends ActionSupport
{
    private String nombre;
    private int numero;

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

    public void setNumero(int numero)
    {
        this.numero = numero;
    }
}

```

Ahora agregamos un **String**, llamado "**mensaje**", que contendrá el mensaje que se regresará al usuario. Colocamos también un **getter** para esta propiedad:

```

private String mensaje;

public String getMensaje()
{
    return mensaje;
}

```

Lo siguiente que haremos es sobre-escribir el método "**execute**" estableciendo el mensaje del usuario:

```

@Override
public String execute() throws Exception
{
    mensaje = "Bienvenido " + nombre + " al munddo de Struts 2. Tu número de la suerte de hoy es " + numero;

    return SUCCESS;
}

```

La clase "**SaludoUsuarioAction**" completa queda de la siguiente forma:

```

public class SaludoUsuarioAction extends ActionSupport
{
    private String nombre;
    private int numero;
    private String mensaje;

    @Override
    public String execute() throws Exception
    {
        mensaje = "Bienvenido " + nombre + " al munddo de Struts 2. Tu número de la
suerte de hoy es " + numero;

        return SUCCESS;
    }

    public String getMensaje()
    {
        return mensaje;
    }

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

    public void setNumero(int numero)
    {
        this.numero = numero;
    }
}

```

¿Por qué hemos colocado **setters** de unas propiedades y **getters** de otras? Bien, esto es porque algunas de las propiedades del action ("**nombre**" y "**mensaje**") serán establecidas por los interceptores. Estos valores son recibidos a través del formulario que creamos anteriormente. En el caso del atributo "**mensaje**" este será leído por el framework para presentar los datos en una página **JSP**.

En otras palabras, debemos proporcionar **setters** para las propiedades que serán establecidas por el framework, y **getters** para las propiedades que serán leídas.

Ahora haremos la página que mostrará el saludo al usuario. Agregamos una página llamada "**saludoUsuario.jsp**". En esta página indicaremos que haremos uso de la biblioteca de etiquetas de **Struts 2**, como lo hemos hecho anteriormente. También usaremos la etiqueta "**property**" para mostrar el mensaje que generamos en el action:

```

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

<s:property value="mensaje" />

```

El último paso es configurar este **Action** en el archivo "**struts.xml**". Aquí indicaremos un **Action** llamado "**saludoUsuario**" que será procesado por la clase "**SaludoUsuarioAction**". También indicaremos un **result** que enviará al usuario a la página "**saludoUsuario.jsp**":

```

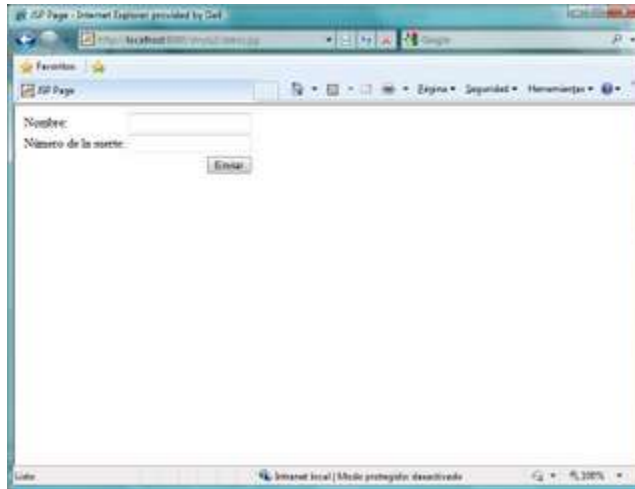
<action name="saludoUsuario"
class="com.javatutoriales.struts2.configuracion.actions.SaludoUsuarioAction">
<result>/saludoUsuario.jsp</result>
</action>

```

Esto es todo. Ahora iniciamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/struts2/datos.jsp>

Deberemos ver un formulario como el siguiente:

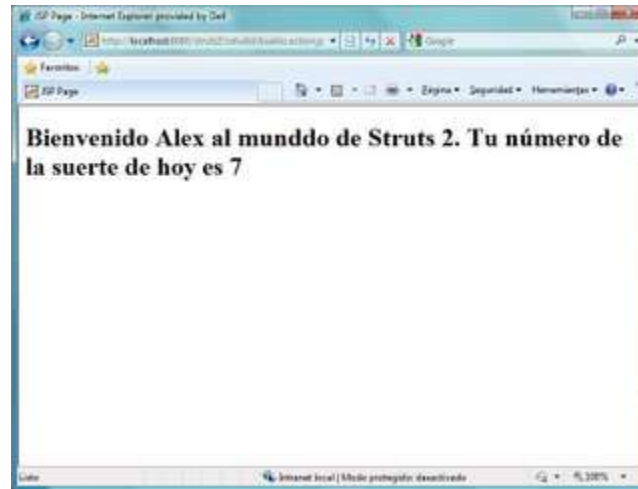


Como podemos ver, las etiquetas de **Struts** han generado un formulario y lo han acomodado para que se vea bonito :D. Si revisamos el código fuente de la página podremos ver que dentro del formulario hay una tabla para acomodar los componentes:

```
<form id="saludoUsuario" name="saludoUsuario" action="/struts2/saludoUsuario.action"
method="post">
<table class="wwFormTable">
<tr>
<td class="tdLabel">
<label for="saludoUsuario_nombre" class="label">Nombre:</label>
</td>
<td>
<input type="text" name="nombre" value="" id="saludoUsuario_nombre"/>
</td>
</tr>
<tr>
<td class="tdLabel">
<label for="saludoUsuario_numero" class="label">Número de la suerte:</label>
</td>
<td>
<input type="text" name="numero" value="" id="saludoUsuario_numero"/>
</td>
</tr>
<tr>
<td colspan="2">
<div align="right"><input type="submit" id="saludoUsuario_0" value="Enviar"/></div>
</td>
</tr>
</table>
</form>
```

Aunque el formulario se ve muy bien de esta forma, el uso de tablas para el acomodo de los componentes se considera una mala práctica, veremos cómo hacerlo de forma correcta en algún otro tutorial ^_^.

Ahora, cuando llenemos los datos del formulario y hagamos clic en el botón enviar el **Action** procesará nuestros datos y nos enviará la respuesta correspondiente, por lo que deberemos ver la siguiente pantalla:



Como podemos ver, los datos fueron establecidos correctamente y el mensaje mostrado es correcto. También podemos ver que en este caso ocurre algo interesante, el atributo "**numero**" que establecimos como tipo "**int**" fue tratado como eso, como un número, no como un **String**, como ocurre normalmente en los **Servlets**. Esto es debido a que los interceptores se encargan de transformar los datos que reciba al tipo indicado. Lo mismo ocurre con valores booleanos, arreglos, fechas, archivos, etc. Sin que tengamos que hacer la conversión manual.

Ahora que veremos cómo configurar nuestra aplicación usando anotaciones.

Configuración de Struts 2 Usando Anotaciones

Cuando usamos anotaciones para realizar la configuración de una aplicación con **Struts 2**, necesitamos hacer uso de un plugin, el plugin "**convention**" (el cual se encuentra en el jar "**struts2-convention-plugin-2.2.3**" que agregamos para la biblioteca de "**Struts2Anotaciones**"). Este plugin proporciona la característica de poder crear una aplicación **Struts**, siguiendo una serie de convenciones de nombres tanto para los **results**, los **Actions**, en fin, para todo lo relacionado con **Struts**. Esto sin necesidad de crear ningún archivo de configuración ni usar ninguna anotación.

Veremos el uso de este plugin en un tutorial exclusivo, un poco más adelante, por ahora debemos crear un nuevo proyecto web en el **NetBeans** siguiendo los pasos anteriores, hasta antes de llegar al título que dice "**Configuración de Struts 2 Usando un Archivo XML**" ^_^. O sea, debemos tener creado un proyecto web, configuramos el filtro "**struts2**" en el archivo "**web.xml**" y agregamos, en vez de la biblioteca "**Struts2**" que usamos para configurar usando archivos **XML**, la biblioteca "**Struts2Anotaciones**".

Una vez llegado a este punto, creamos un paquete en nuestra aplicación, en mi caso será "**com.javatutoriales.struts2.configuracion.acciones**", y será en este paquete en el que coloquemos todas las acciones que creamos en el tutorial.

Dentro de este paquete crearemos una clase llamada "**SaludoAction**", esta clase puede extender de "**ActionSupport**", aunque como vimos no es necesario. Yo si extenderé de "**ActionSupport**":

```
public class SaludoAction extends ActionSupport
{
```



```
}
```

A esta clase le agregaremos un atributo, de tipo **String**, llamado "**mensaje**", con su **getter** correspondiente:

```
public class SaludoAction extends ActionSupport
{
    private String mensaje;

    public String getMensaje()
    {
        return mensaje;
    }
}
```

Lo siguiente es sobre-escribir el método "**execute**" para que genere un mensaje de saludo al usuario:

```
@Override
public String execute()
{
    mensaje = "Hola Mundo de Struts 2";

    return SUCCESS;
}
```

La clase "**SaludoAction**" hasta el momento se ve de esta forma:

```
public class SaludoAction extends ActionSupport
{
    private String mensaje;

    @Override
    public String execute()
    {
        mensaje = "Hola Mundo de Struts 2";

        return SUCCESS;
    }

    public String getMensaje()
    {
        return mensaje;
    }
}
```

Ahora configuraremos este **Action**. Para eso tenemos dos formas de hacerlo. Ambas son casi iguales, con diferencias muy sutiles. En la primera debemos colocar las anotaciones **a nivel de la clase**. Las anotaciones son muy claras, y representan los elementos importantes del archivo **XML**.

Lo primero que debemos hacer es indicar el namespace para la acción, usando la anotación "**@Namespace**":

```
@Namespace(value="/")
public class SaludoAction extends ActionSupport
```

Ahora debemos usar la anotación "**@Action**" para indicar el nombre de esta acción, usando su atributo "**value**":

```
@Namespace(value="/")
```

```
@Action(value="saludo")
public class SaludoAction extends ActionSupport
```

Lo último que debemos hacer es configurar los **results** para este **Action**. Esto lo hacemos usando el atributo "**results**" de la anotación "**@Action**" y la anotación "**@Result**". Dentro de esta última colocaremos el nombre del **result**, en su atributo "**name**", y la ubicación de la página que mostrará el resultado apropiado, usando su atributo "**location**":

```
@Namespace(value="/")
@Action(value="saludo", results={@Result(name="success", location="/saludo.jsp")})
public class SaludoAction extends ActionSupport
```

Esto es todo para la primera forma de configuración usando anotaciones. La segunda forma es muy parecida, solo que esta vez colocamos las anotaciones **a nivel del método "execute"**. Si lo hacemos así no podemos usar la anotación "**@Namespace**", pero podemos agregar el namespace directamente en el atributo "**value**" de la anotación "**@Action**":

```
@Override
@Action(value="/saludo", results={@Result(name="success", location="/saludo.jsp")})
public String execute()
{
    mensaje = "Hola Mundo de Struts 2";

    return SUCCESS;
}
```

En lo personal prefiero la primera forma así que será esta la que usaremos en los tutoriales :D. La clase "**SaludoAction**" anotada queda de la siguiente forma:

```
@Namespace(value="/")
@Action(value="saludo", results=@Result(name="success", location="/saludo.jsp"))
public class SaludoAction extends ActionSupport
{
    private String mensaje;

    @Override
    public String execute()
    {
        mensaje = "Hola Mundo de Struts 2";

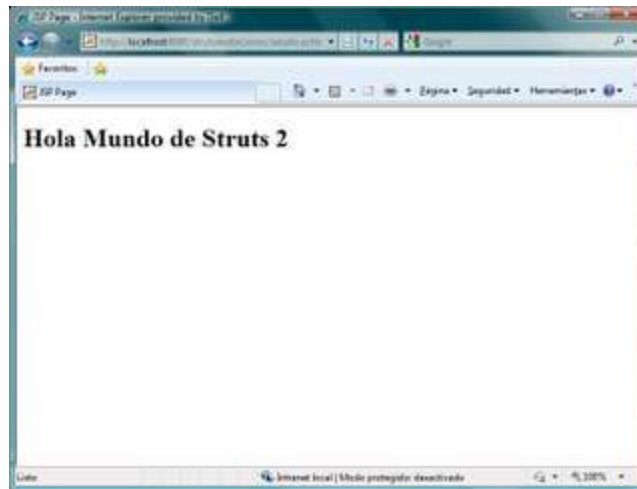
        return "success";
    }

    public String getMensaje()
    {
        return mensaje;
    }
}
```

Ahora podemos ejecutar nuestra aplicación y entrar a la siguiente dirección:

<http://localhost:8080/strutsanotaciones/saludo.action>

Con lo que deberemos ver la siguiente pantalla:



Esto indica que la configuración ha funcionado correctamente.

Como podemos ver, no es necesario dar ninguna indicación al filtro de **Struts 2** de en dónde se encuentran las clases anotadas. Esto en realidad es un poco engañoso, ya que si, por ejemplo, cambiamos el nombre del paquete de "**struts2**" a alguna otra cosa, como "**aplicacion**", al volver a ejecutar nuestra aplicación obtendremos el siguiente error:



Esto ocurre porque el plugin **convention** usa, como habíamos dicho, una convención de nombres para saber en dónde se encuentra cierto contenido. En el caso de los **Actions**, el plugin busca todas las clases **Action** que implementen la interface "**Action**" o cuyos nombres terminen con la palabra "**Action**" y que se encuentren en paquetes específicos. Estos paquetes son encontrados por el plugin **convention** usando una cierta metodología de búsqueda. Primero, el plugin busca los paquetes cuyos nombres sean "**struts**", "**struts2**", "**action**" o "**actions**". Cualquier paquete que coincida con estos nombres es considerado el paquete raíz de los **Action** por el plugin **convention**. Después, el plugin busca todas las clases en este paquete, y en sus sub-paquetes y determina si las clases implementan la interface "**Action**" o su nombre termina con "**Action**".

Debido a que en este caso, hemos cambiado el nombre del paquete que el plugin usaba para saber en dónde localizar los **Actions**, ya no puede encontrarlos. ¿Qué podemos hacer en este caso? Afortunadamente para esto existe un mecanismo que nos permite indicarle al plugin **convention** dónde puede encontrar los **Actions** de nuestra aplicación. Para este fin debemos agregar una constante: "**struts.convention.action.packages**", y su valor será una lista de los paquetes en los que se encuentran los **Actions**.

Ya que en esta ocasión no tenemos un archivo de configuración, debemos colocar esta constante como un parámetro del filtro de "**struts2**", en el archivo "**web.xml**".

Debido a que en mi caso el paquete en el que se encuentran las clases es "**com.javatutoriales.aplicacion.configuracion.acciones**", mi configuración del filtro queda de la siguiente forma:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
<init-param>
<param-name>struts.convention.action.packages</param-name>
<param-value>com.javatutoriales.aplicacion.configuracion.acciones</param-value>
</init-param>
</filter>
```

El resto de la configuración queda como la teníamos.

Ahora al ejecutar la aplicación, deberemos ver nuevamente la siguiente salida:



La cual nos muestra que todo funciona correctamente ^_^.

Struts 2 - Parte 2: OGNL

OGNL es el acrónimo de **Object Graph Navigation Language**, un lenguaje de expresiones muy poderoso [que](#) nos permite leer valores de objetos Java. Este lenguaje nos permite leer valores y ejecutar métodos (que regresen algún valor) para mostrar los valores o resultados de los mismos en nuestras páginas **JSP** creadas usando las etiquetas de **Struts**. Además proporciona una conversión automática de tipos que permite convertir datos desde texto **HTTP** a objetos Java.

En este tutorial aprenderemos a usar este sencillo pero poderoso lenguaje dentro de nuestras aplicaciones, así [como](#) los objetos implícitos que tiene y cómo acceder a ellos. Además veremos cómo obtener valores de constantes, variables, y elementos enumerados, que se encuentran en nuestras clases.

Struts no funciona exactamente usando una versión estándar de **OGNL**, usa una versión propia a la que agrega ciertas características interesantes.

OGNL usa un contexto estándar de nombres para evaluar las expresiones, esto quiere decir que dependiendo de qué tan "**profundo**" esté nuestro objeto en el grafo, podremos hacer referencia a él de distintas formas. El objeto de más alto nivel en **OGNL** es un **Map**, al cual llamamos "**mapa de contexto**" (**context map**) o simplemente "**contexto**".

OGNL maneja siempre **un objeto raíz** dentro del contexto. Este objeto raíz **es el objeto default al que se hacen las llamadas**, a menos que se indique lo contrario. Cuando usamos una expresión, **las propiedades del objeto raíz pueden ser referenciadas sin ninguna marca especial**, esto quiere decir que si nuestro objeto tiene una propiedad llamada "**nombre**", hacemos referencia a él simplemente con la expresión "**nombre**". Las referencias a otros objetos son marcadas con un signo de número (**#**).

Para entender esto veamos un ejemplo de cómo funciona esto en el **OGNL** estándar. Supongamos que hay dos objetos en el mapa de contexto de **OGNL**: "**foo**" y "**bar**", y que el objeto "**foo**" **es el objeto raíz**. El siguiente código muestra cómo resuelve **OGNL** los valores pedidos:

```
#foo.blah    //regresa foo.getBlah()
#bar.blah    //regresa bar.getBlah()
blah        //regresa foo.getBlah(), porque foo es la raíz
```

Esto quiere decir que **OGNL** permite que haya varios objetos en el contexto, pero solo podemos acceder a los miembros del objeto raíz directamente. También es importante mencionar aquí que cuando hacemos referencia a una propiedad como "**blah**", **OGNL** buscará un método "**getBlah()**", que regrese algún valor y no reciba parámetros, para obtener el valor que mostrará.

Cuando queremos invocar métodos usamos el nombre del método junto con paréntesis, como en una invocación normal de Java. En el caso anterior pudimos [haber](#) hecho algo como lo siguiente:

```
#foo.getBlah()
#bar.getBlah()
```

En el **OGNL** estándar solo se tiene una raíz, sin embargo en el **OGNL** de **Struts 2** se tiene un "**ValueStack**", el cual permite simular la existencia de varias raíces. Todos los objetos que pongamos en el "**ValueStack**" se comportarán como la raíz del mapa de contexto.

En el caso de **Struts 2**, el framework establece el contexto como un objeto de tipo "**ActionContext**", que es el contexto en el cual se ejecuta un Action (cada contexto es básicamente un contenedor de objetos que un Action necesita para su ejecución, como los objetos "**session**", "**parameters**", "**locale**", etc.), y el "**ValueStack**" como el objeto raíz. El "**ValueStack**" es un conjunto de muchos objetos, pero para **OGNL** este aparenta ser solo uno.

Debido al "**ValueStack**", al que algunas veces llamamos solo "**stack**", en vez de que nuestras expresiones tengan que obtener el objeto que queremos del **stack** y después obtener las propiedades de él (como en el ejemplo de **#bar.blah**), el **OGNL** de **Struts 2** tiene un "**PropertyAccessor**" especial que buscará automáticamente en todos los objetos del **stack** (de arriba a abajo) hasta que encuentre un objeto con la propiedad que estamos buscando.

Siempre que **Struts 2** ejecuta uno de nuestros Actions, los coloca en la cima del **stack**, es por eso que en [el tutorial anterior](#) hacíamos referencia a la propiedad llamada "**mensaje**", del Action correspondiente, solamente indicando el nombre de la propiedad. **Struts 2** busca en el **stack** un objeto que tenga un **getter** para esa propiedad, en este caso nuestro Action.

Veamos otro ejemplo. Supongamos que el **stack** contiene dos objetos: "**Animal**" y "**Persona**". Ambos objetos tienen una propiedad "**nombre**", "**Animal**" tiene una propiedad "**raza**", y "**Persona**" tiene una propiedad "**salario**". "**Animal**" está en la cima del **stack**, y "**Persona**" está debajo de él. Si hacemos llamadas simples, como las mostradas a continuación, **OGNL** resuelve los valores de la siguiente forma:

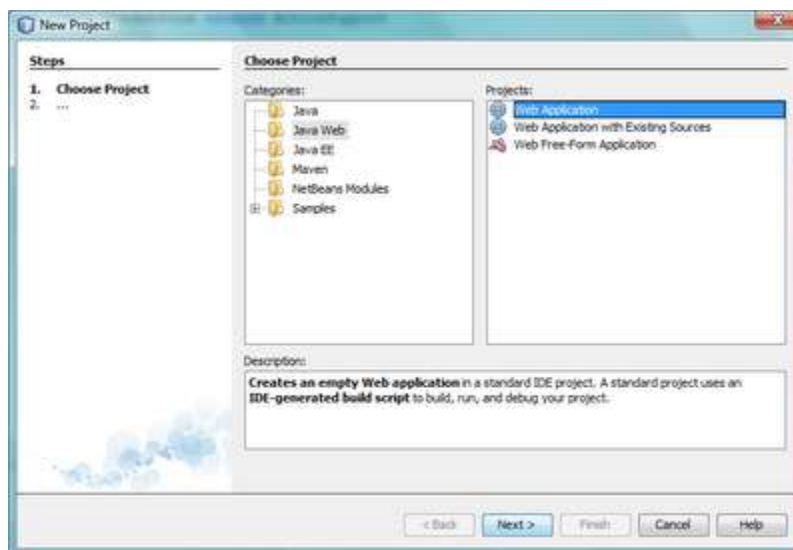
```
raza      //llama a animal.getRaza()
salario   //llama a persona.getSalario()
nombre    //llama a animal.getNombre(), porque animal está en la cima del stack
```

En el ejemplo anterior, se regresó el valor del "**nombre**" del **Animal**, que está en la cima del **stack**. Normalmente este es el comportamiento deseado, pero algunas veces nos interesa recuperar el valor de la propiedad de un objeto que se encuentra más abajo en el **stack**. Para hacer esto, **Struts 2** agrega soporte para índices en el **ValueStack**. Todo lo que debemos hacer es:

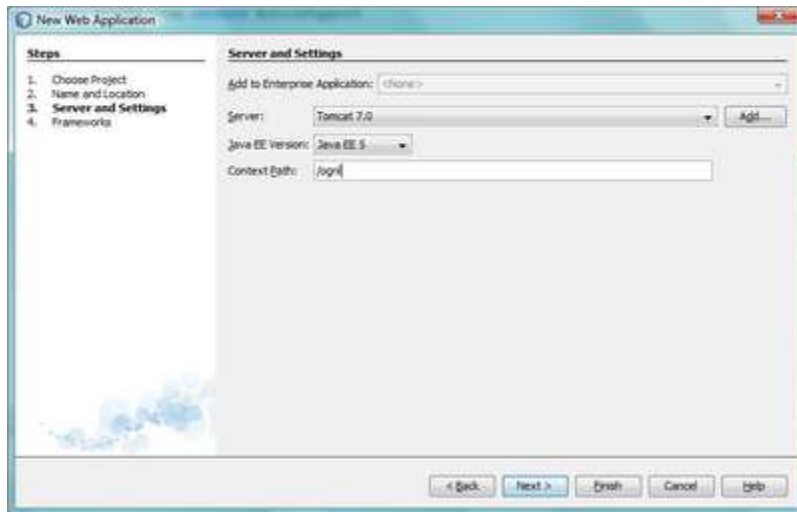
```
[0].nombre //llama a animal.getNombre()
[1].nombre //llama a persona.getNombre()
```

Con esto le indicamos a **OGNL** a partir de cuál índice queremos que inicie la búsqueda (digamos que cortamos el **stack** a partir del índice que le indicamos).

Suficiente teoría :D. Comencemos a ver lo anterior en código. Crearemos un nuevo proyecto web en **NetBeans**. Para esto vamos al menú "**File -> New Project...**". En la ventana que aparece seleccionamos la categoría "**Java Web**" y en el tipo de proyecto "**Web Application**":

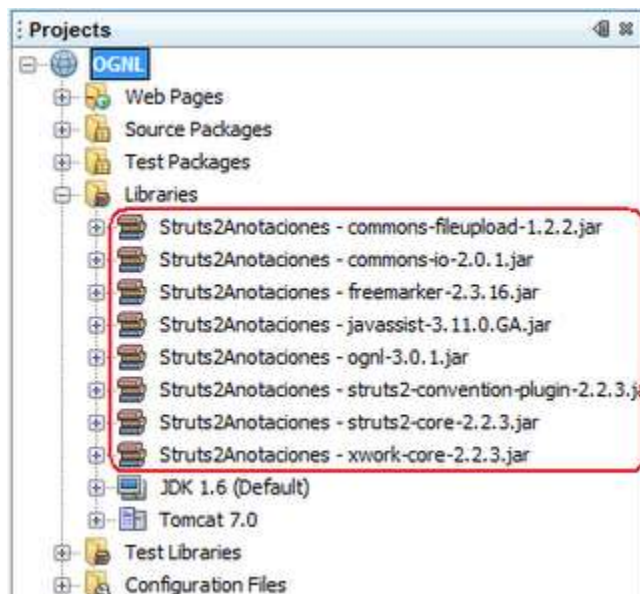


Presionamos el botón "**Next >**" y le damos un nombre y una ubicación a nuestro proyecto; presionamos nuevamente el botón "**Next >**" y en este punto se nos preguntará el servidor que queremos usar. En nuestro caso usaremos el servidor "**Tomcat 7.0**", con la versión 5 de JEE y presionamos el botón "**Finish**":



Con esto aparecerá en nuestro editor una página "**index.jsp**".

Ahora agregamos la librería "**Struts2Anotaciones**" que creamos en [el tutorial anterior](#). Hacemos clic derecho en el nodo "**Libraries**" del panel de proyectos. En el menú que aparece seleccionamos la opción "**Add Library...**". En la ventana que aparece seleccionamos la biblioteca "**Struts2Anotaciones**" y presionamos "**Add Library**". Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto:



Finalmente abrimos el archivo "**web.xml**" y agregamos la configuración del filtro de **Struts 2**, de la misma forma que lo hicimos en [el tutorial anterior](#):

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>
```

```
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Ya tenemos todo listo para comenzar. Lo primero que haremos en este ejemplo es crear un paquete que contendrá nuestras clases "**Animal**" y "**Persona**". En mi caso el paquete se llamará "**com.javatutoriales.struts2.ognl.modelo**". En el agregamos dos clases: "**Animal**" y "**Persona**".

La clase "**Animal**" tendrá, como habíamos dicho, dos atributos de tipo **String**: "**nombre**" y "**raza**", con sus correspondientes **setters** y **getters**:

```
public class Animal
{
    private String nombre;
    private String raza;

    public String getNombre()
    {
        return nombre;
    }

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

    public String getRaza()
    {
        return raza;
    }

    public void setRaza(String raza)
    {
        this.raza = raza;
    }
}
```

La clase "**Persona**" también contendrá dos atributos de tipo **String**: "**nombre**" y "**salario**", con sus correspondientes **setters** y **getters**:


```

public class Persona
{
    private String nombre;
    private String salario;

    public String getNombre()
    {
        return nombre;
    }

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

    public String getSalario()
    {
        return salario;
    }

    public void setSalario(String salario)
    {
        this.salario = salario;
    }
}

```

Ahora crearemos el Action que se encargará de poner una instancia de cada una de estas clases en el "**ValueStack**". Creamos una nueva clase llamada "**StackAction**" y hacemos que esta extienda de "**ActionSupport**":

```

public class StackAction extends ActionSupport
{
}

```

Agregamos las anotaciones correspondientes, y que explicamos en el tutorial anterior. Haremos que este Action responda al nombre de "**stack**" y que nos envíe a una página llamada **"/stack.jsp"**:

```

@Namespace(value="/")
@Action(value="stack", results={@Result(location="/stack.jsp")})
public class StackAction extends ActionSupport
{
}

```

Lo siguiente que haremos es sobre-escribir el método "**execute**" del Action para obtener una referencia al **ValueStack**. La forma de hacer esto último es a través de un método estático de la clase "**ActionContext**":

```

@Override
public String execute() throws Exception
{
    ValueStack stack = ActionContext.getContext().getValueStack();
}

```

Una vez que tenemos esta referencia solo nos resta crear una instancia de cada una de nuestras clases, establecer los valores de sus parámetros, y agregarlos al **ValueStack** usando su método "**push**":

```

@Override
public String execute() throws Exception
{
    ValueStack stack = ActionContext.getContext().getValueStack();

    Animal animal = new Animal();
    animal.setNombre("nombre del animal");
    animal.setRaza("perro labrador");

    Persona persona = new Persona();
    persona.setNombre("nombre de la persona");
    persona.setSalario("realmente poco");

    stack.push(persona);
    stack.push(animal);

    return SUCCESS;
}

```

Nota: Por lo regular no agregamos objetos al stack de forma manual como lo estamos haciendo para este ejemplo, dejamos que sea Struts quien agregue los objetos necesarios de forma automática. Esto solo lo hacemos en casos en los que no queda otra opción... como en este ejemplo ^_~!

Agregamos primero la referencia de la **Persona** y luego la del **Animal** porque, como en toda buena pila, el último elemento que se agregue al **ValueStack** será el que quede en su cima.

Ahora creamos la página "**stack.jsp**" en el directorio raíz de las páginas web. En esta página indicamos que se usará la biblioteca de etiquetas de **Struts 2**:

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

Usando la etiqueta "**<s:property>**" mostraremos los valores de "**raza**", "**salario**", y "**nombre**". Al colocarlos de esta forma, **Struts** buscará estos valores en todos los objetos que estén en el **ValueStack**:

```

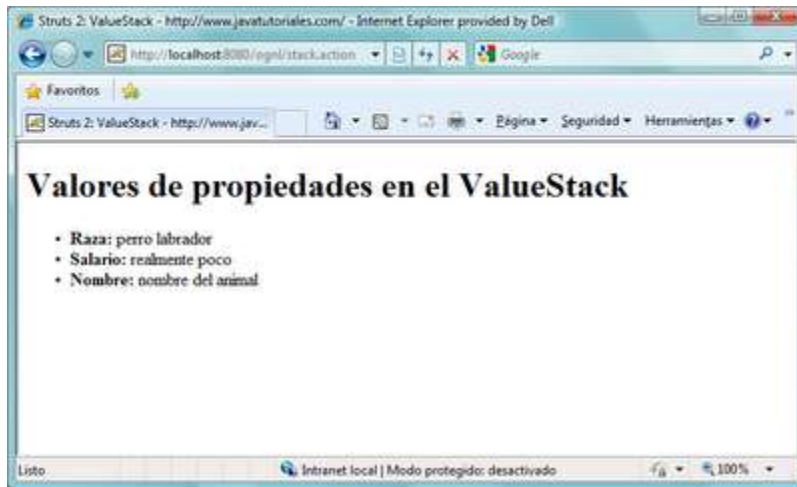
<ul>
<li><strong>Raza: </strong><s:property value="raza" /></li>
<li><strong>Salario: </strong><s:property value="salario" /></li>
<li><strong>Nombre: </strong><s:property value="nombre" /></li>
</ul>

```

Ya está todo listo para correr el ejemplo. Ejecutamos nuestra aplicación, y entramos a la siguiente dirección:

<http://localhost:8080/ognl/stack.action>

Deberemos ver una pantalla como la siguiente:

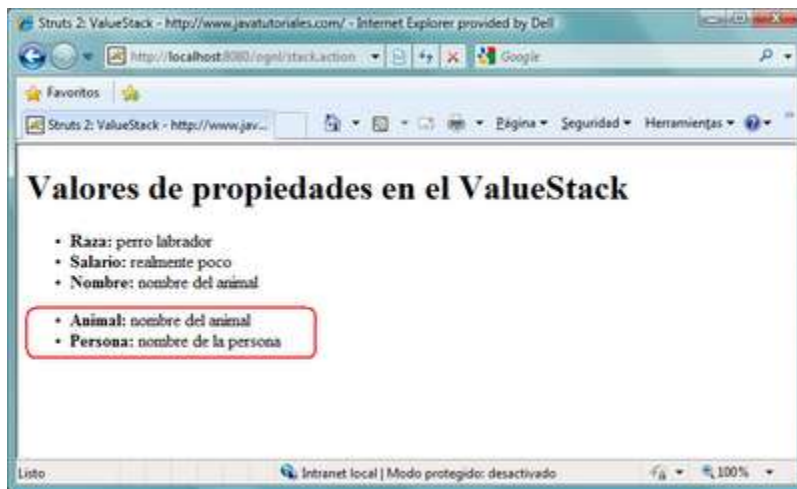


Como podemos ver, la teoría es correcta ^_^ . Al buscar la propiedad "**raza**", la encuentra en el objeto tipo "**Animal**", mostrando el valor de la misma; cuando busca la propiedad salario la encuentra en el objeto de tipo "**Persona**"; y al buscar la propiedad "**nombre**", que tienen ambos objetos, muestra el valor del objeto que se encuentra en la cima del **stack**, o sea el de "**Animal**".

Ahora hagamos una segunda prueba haciendo uso de los índices del **ValueStack**. Agreguemos las instrucciones que habíamos visto anteriormente: ("**[0].nombre**" y "**[1].nombre**") usando la etiqueta "**s:property**":

```
<ul>
<li><strong>Animal: </strong><s:property value="[0].nombre" /></li>
<li><strong>Persona: </strong><s:property value="[1].nombre" /></li>
</ul>
```

Si volvemos a ejecutar nuestra aplicación veremos la siguiente salida:



Como podemos ver, efectivamente, el uso de índices permite que seleccionemos valores de objetos que se encuentran a una profundidad mayor en el **ValueStack**.

Solo para recordar. Cuando **Struts 2** ejecuta un Action como consecuencia de una petición, este action es colocado en la cima del **ValueStack**, es por esto que podemos acceder a sus atributos haciendo una llamada directa al nombre del mismo, como lo hicimos en [el tutorial anterior](#).

Hagamos nuevamente una prueba para explicar un poco más en detalle lo que ocurre en ese caso. Primero crearemos una clase llamada "**SaludoAction**", la cual será igual a la [del tutorial anterior](#):

```

@Namespace("/")
@Action(value = "saludo", results = {@Result(location = "/saludo.jsp")})
public class SaludoAction extends ActionSupport
{
    private String mensaje;

    @Override
    public String execute() throws Exception
    {
        mensaje = "Hola Mundo!!";

        return SUCCESS;
    }

    public String getMensaje()
    {
        return mensaje;
    }
}

```

En este caso nuestro Action tiene un atributo "**mensaje**" que en el método "**execute**" será establecido con un valor. También se proporciona un **getter** para este atributo.

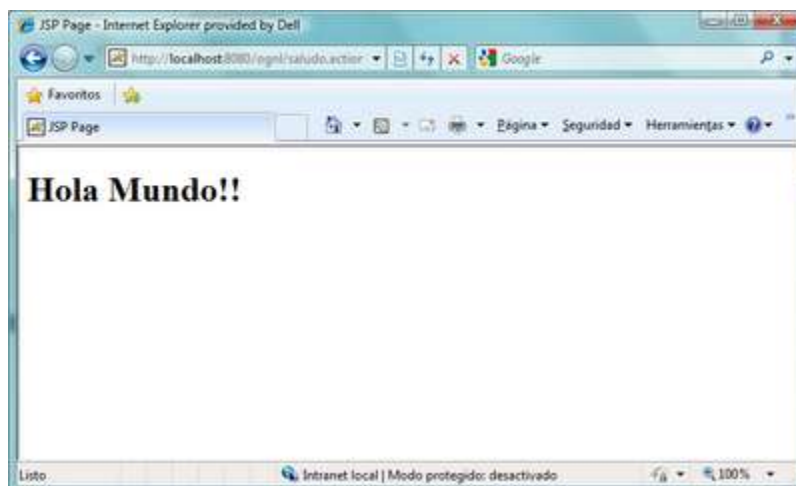
Crearemos una página **JSP** llamada "**saludo.jsp**", en la raíz de las páginas web de la aplicación. En esta página mostraremos el valor del atributo "**mensaje**" usando la biblioteca de etiquetas de **Struts 2**:

```
<s:property value="mensaje" />
```

Ejecutamos la aplicación, y al acceder a la siguiente dirección:

<http://localhost:8080/ognl/saludo.action>

Debemos ver la siguiente página:

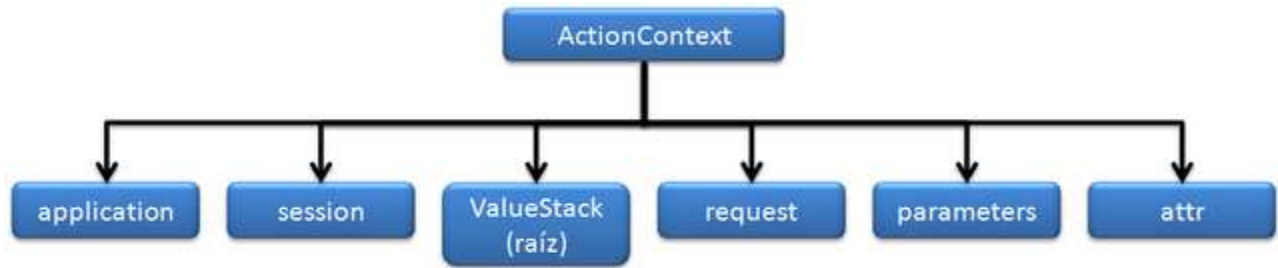


Una vez más, solo para que quede claro. Cuando realizamos una petición para el Action, el **DispatcherFilter** ejecutará el método "**execute**" del Action, después colocará este Action en la cima del **ValueStack** con lo cual lo tendremos disponible.

Cuando hacemos la petición para el atributo "**mensaje**", usando la etiqueta "**s:property**", **Struts** busca en el **ValueStack**, de arriba a abajo, un objeto que tenga un método "**getMensaje()**". Como el primer objeto que

encuentra con el método "**getMensaje()**" es "**SaludoAction**" (de hecho es el primer objeto en el que busca) ejecuta este método mostrando el valor correspondiente mostrando la pantalla que vimos anteriormente.

Además del **ValueStack**, **Struts 2** coloca otros objetos en el **ActionContext**, incluyendo **Maps** que representan los contextos de "**application**", "**session**", y "**request**". Una representación de esto puede verse en la siguiente imagen:



Recuerden que para hacer referencia a los objetos del contexto que NO son la raíz **debemos preceder el nombre del objeto con el símbolo de gato (#)**.

Hablaré brevemente de estos objetos:

- "**application**" representa el **ApplicationContext**, o sea, el contexto completo de la aplicación.
- "**session**" representa el **HTTPSession**, o sea, la sesión del usuario.
- "**request**" representa el **ServletRequest**, o sea, la petición que se está sirviendo.
- "**parameters**" representa los parámetros que son enviados en la petición, ya sea por **GET** o por **POST**.
- "**attr**" representa los atributos de los objetos implícitos. Cuando preguntamos por un atributo, usando "**attr**", este busca el atributo en los siguientes scopes: **page**, **request**, **session**, **application**. Si lo encuentra, regresa el valor del atributo y no continúa con la búsqueda, sino regresa un valor nulo.

Veamos un ejemplo de esto. No entraré en muchos detalles sobre el manejo de la sesión ya que eso lo dejaré para el próximo tutorial.

Creamos una nueva clase Java llamada "**DatosAction**" que extienda de "**ActionSupport**". Colocaremos algunas anotaciones para la configuración de **Struts 2**:

```
@Namespace(value = "/")

```

Sobre-escribimos el método "**execute**" de esta clase para obtener el **ActionContext**. Una vez teniendo este contexto podemos obtener la sesión y podemos agregar objetos a ella. En este caso agregaré una cadena solo para poder obtenerla después:

```
@Override
public String execute() throws Exception
{
    ActionContext.getContext().getSession().put("datoSesion", "dato en la sesion");

    return SUCCESS;
}
```

Ahora creamos una **JSP** llamada "**datos.jsp**" en el directorio raíz de las páginas web. En esta página indicaremos que usaremos la biblioteca de etiquetas de **Struts 2**; y obtendremos, usando la etiqueta "**<s:property>**", el valor del atributo "**datoSesion**" que colocamos en la sesión del usuario.

Como el objeto "**session**" no es la raíz del mapa de contexto, es necesario hacer referencia a él de la siguiente forma:

```
#session
```

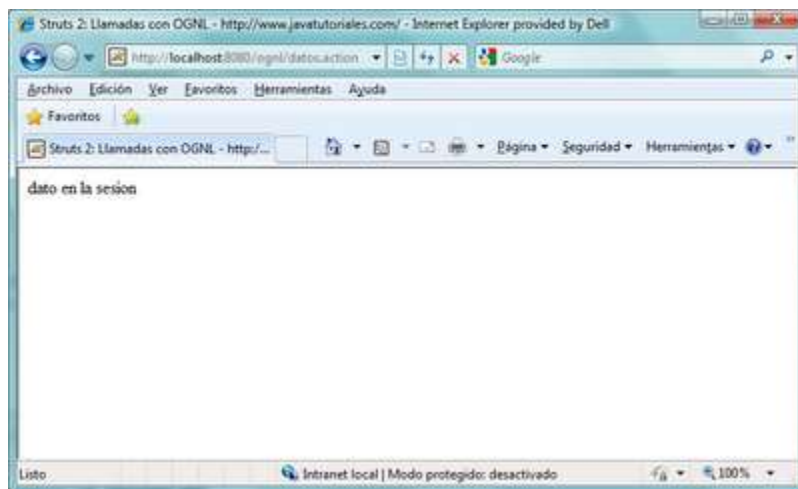
Y podemos obtener cualquiera de sus atributos mediante el nombre del atributo. Por lo que la etiqueta queda de la siguiente forma:

```
<s:property value="#session.datoSesion" />
```

También podemos hacerlo de esta otra forma:

```
<s:property value="#session['datoSesion']" />
```

Cuando ejecutemos la aplicación debemos ver la siguiente salida:



Ahora, digamos que "**parameters**" representa los parámetros que se reciben, ya sea por **GET** o por **POST**.

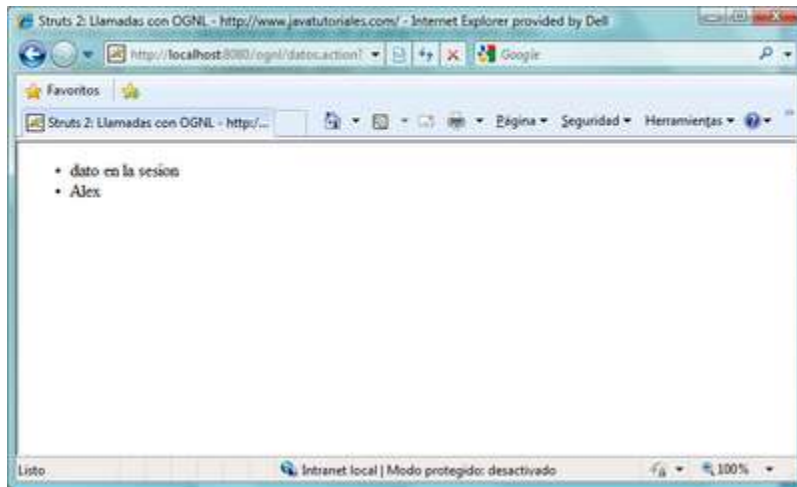
Colocaremos una etiqueta para obtener el valor de un parámetro llamado dato:

```
<s:property value="#parameters.dato" />
```

Y accedemos a la siguiente **URL**:

<http://localhost:8080/ognl/datos.action?dato=Alex>

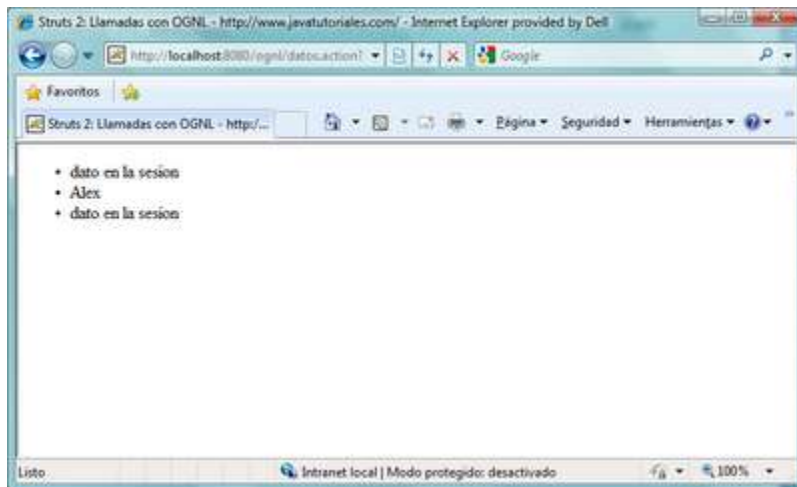
En donde estamos pasando, por **GET**, un parámetro llamado "**dato**" igualado al valor "**Alex**". Cuando entremos en la dirección anterior veremos una pantalla como la siguiente:



Si ahora, usamos el objeto "**attr**" para buscar el valor del atributo "**datoSesion**", de la siguiente forma:

```
<s:property value="#attr.datoSesion" />
```

Obtendremos la siguiente pantalla:



Con **OGNL** no solo es posible acceder a los objetos dentro del **ActionContext**, sino prácticamente a cualquier objeto java que sea visible. Para ilustrar esto veamos otro ejemplo.

Obteniendo Valores de Constantes y Variables con OGNL

Primero crearemos una nueva clase Java que contendrá una serie de atributos, métodos, y constantes. La clase se llamará "**Constantes**" (aunque también contendrá valores variables). Esta clase tendrá una variable de instancia llamada "**atributo**" (con su correspondiente **getter**), una constante llamada "**valor**", y una enumeración llamada "**Datos**" que tendrá un método "**getDato()**":

```

public class Constantes
{
    private String atributo = "atributo de instancia";
    public final static String valor = "variable estatica";

    private static enum Datos {PRIMERO, SEGUNDO, TERCERO; public String getDato(){ return
"dato";} }};

public String getAtributo()
{
    return atributo;
}
}

```

Sé que el atributo "**valor**" no sigue la convención de las constantes de Java, pero colocarla de esta forma ayudará a hacer las explicaciones más sencillas ^_^.

Ahora, para que las cosas sean un poco más interesantes, agregaremos unos cuantos métodos.

Primero agregaremos un método de instancia que regrese una cadena y no reciba parámetros; y una versión sobrecargada de este método que reciba una cadena:

```

public String metodoDeInstancia()
{
    return "metodo de instancia";
}

public String metodoDeInstancia(String mensaje)
{
    return mensaje;
}

```

Agregaremos también las versiones estáticas de estos métodos:

```

public static String metodoEstatico()
{
    return "metodo estatico";
}

public static String metodoEstatico(String mensaje)
{
    return mensaje;
}

```

La clase "**Constantes**" completa queda de la siguiente forma:

```

public class Constantes
{
    private String atributo = "atributo de instancia";
    public final static String valor = "variable estatica";

    private static enum Datos {PRIMERO, SEGUNDO, TERCERO; public String getDato(){ return
"dato";} }};

    public String metodoDeInstancia()

```



```

    {
        return "metodo de instancia";
    }

    public String metodoDeInstancia(String mensaje)
    {
        return mensaje;
    }

    public static String metodoEstatico()
    {
        return "metodo estatico";
    }

    public static String metodoEstatico(String mensaje)
    {
        return mensaje;
    }

    public String getAtributo()
    {
        return atributo;
    }
}

```

Como podemos ver, esta clase no tiene nada de especial: ninguna anotación, no usa ninguna librería, ni nada por el estilo.

Ahora creamos una página llamada "**constantes.jsp**", en el directorio raíz de las páginas web.

Para crear un objeto nuevo con **OGNL** podemos usar el operador "**new**" y el "fully qualified class name". Por ejemplo, para crear un nuevo objeto de la clase "**Constantes**" haríamos lo siguiente:

```
<s:property value="new com.javatutoriales.struts2.ognl.Constantes()" />
```

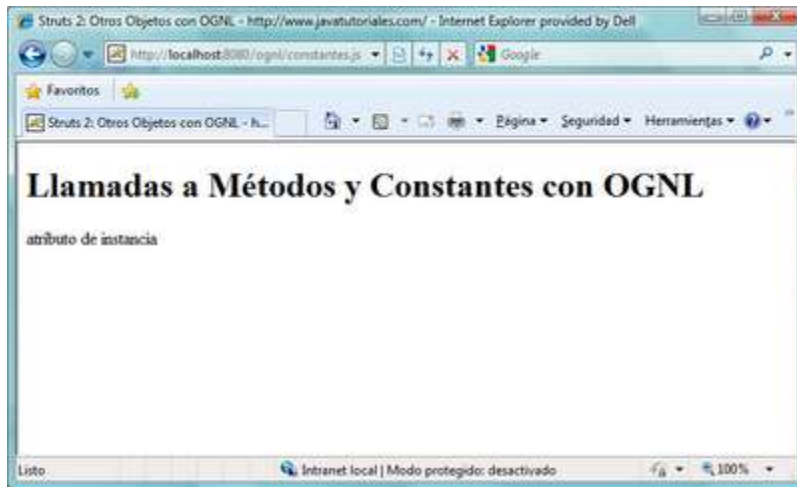
Teniendo esta instancia podemos obtener los valores de los atributos del objeto e invocar sus métodos, como lo haríamos en un objeto normal. Por ejemplo, para obtener el valor del atributo "**atributo**" se puede hacer de la siguiente forma:

```
<s:property value="new com.javatutoriales.struts2.ognl.Constantes().atributo" />
```

Con esto, si entramos a la siguiente dirección:

<http://localhost:8080/ognl/constantes.jsp>

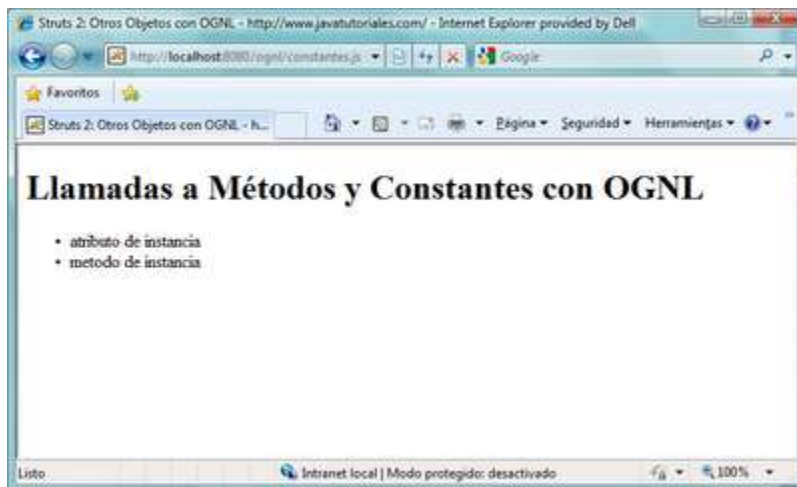
Obtenemos la siguiente pantalla:



Para invocar un método de instancia se hace de la misma forma (no olviden colocar los paréntesis al final):

```
<s:property value="new com.javatutoriales.struts2.ognl.Constantes().metodoDeInstancia()" />
```

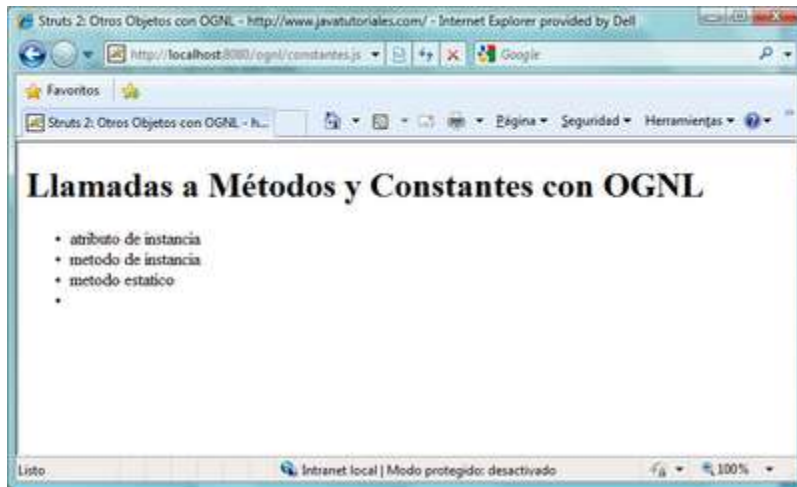
Con esto, obtenemos la siguiente pantalla:



Podemos invocar los miembros estáticos exactamente de la misma forma:

```
<s:property value="new com.javatutoriales.struts2.ognl.Constantes().metodoEstatico()" />
<s:property value="new com.javatutoriales.struts2.ognl.Constantes().valor" />
```

Con esto obtenemos la siguiente pantalla:



El valor de la constante "**valor**" no se muestra, porque recuerden que cuando llamamos a los atributos de esta forma:

```
<s:property value="new com.javatutoriales.struts2.ognl.Constantes().valor" />
```

OGNL busca un método llamado "**getValor()**" en la clase "**Constantes**", y este método no existe.

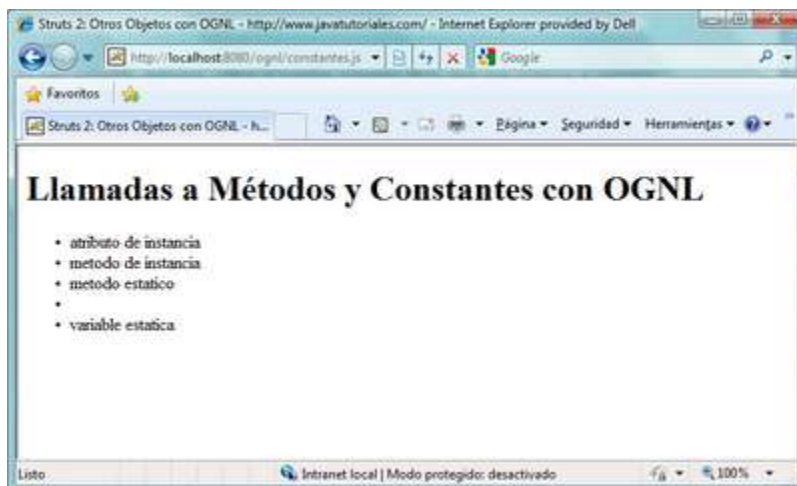
Ahora bien, una de las características de los miembros estáticos es que no es necesario tener una instancia de la clase en la que existe el miembro para poder llamar a estos miembros, pero aquí estamos creando una nueva instancia de las clases para llamarlos. También es posible hacer estas llamadas sin una instancia de la clase. En este caso debemos usar una notación especial de **OGNL**.

En esta notación, debemos indicar el nombre completo de la clase que contiene al miembro estático, precedida por una arroba ("@"). También se debe indicar el miembro que se quiere llamar precedido por una arroba.

Por ejemplo, para usar la constante "**valor**", la etiqueta debe estar colocada de esta forma:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes@valor" />
```

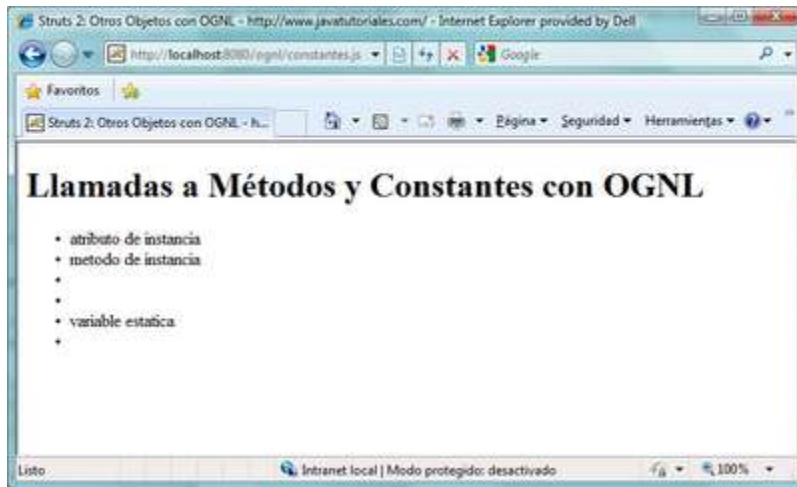
Con lo que obtenemos:



Y para invocar el método estático, de esta forma:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes@metodoEstatico()" />
```

Con esta etiqueta obtenemos la siguiente salida:



Bueno, en este caso parece que también ha habido un problema al invocar el método estático ^_^. En realidad lo que ocurre es que por default **Struts 2** impide la invocación de métodos estáticos desde **OGNL**.

¿Entonces qué podemos hacer? Pues **Struts** proporciona una forma para habilitar esta opción. Podemos colocar el valor de la constante "**struts.ognl.allowStaticMethodAccess**" en "**true**".

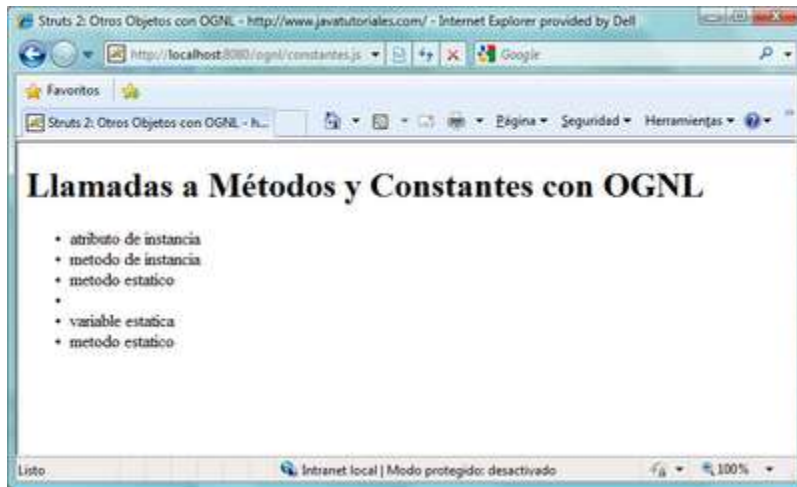
[En el tutorial anterior](#) hablamos un poco de las constantes (muy poco, casi nada ^_!), y declaramos un par de ellas en el archivo "**struts.xml**". Sin embargo, ahora no tenemos un archivo "**struts.xml**" ya que estamos usando anotaciones. Entonces ¿qué podemos hacer? (parece que salimos de un problema para entrar en otro, ¿no les ha pasado?).

Las constantes en **Struts 2** pueden ser declaradas en tres lugares:

- El archivo de configuración "**struts.xml**".
- El archivo "**struts.properties**" (del que aún no hemos hablado).
- Como un parámetro de inicio en el deployment descriptor (el archivo "**web.xml**").
- Como de los tres archivos anteriores solo tenemos el archivo "**web.xml**" será aquí en donde agregamos esta constante. Para esto debemos modificar la declaración del filtro de **Struts 2**, dejándola de la siguiente forma:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
<init-param>
<param-name>struts.ognl.allowStaticMethodAccess</param-name>
<param-value>true</param-value>
</init-param>
</filter>
```

Ahora sí, si ejecutamos nuevamente el ejemplo debemos ver la siguiente salida:



Podemos ver que en este caso, se pudo obtener el valor de la variable estática de forma directa.

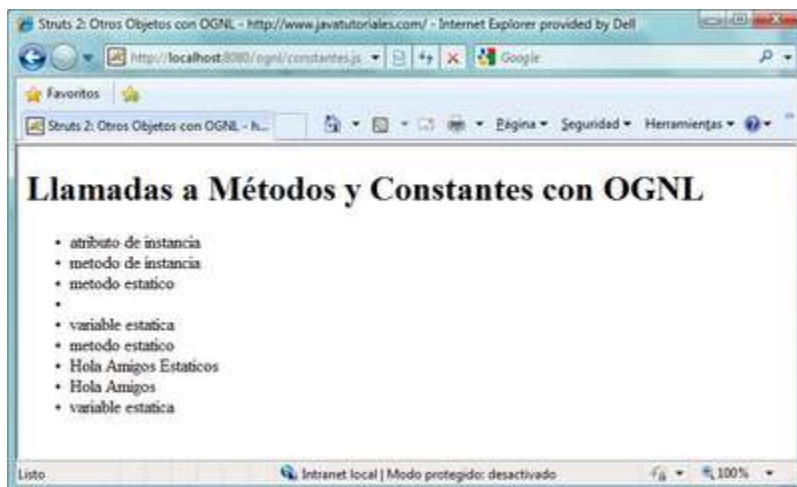
Para invocar las versiones de los métodos que reciben un parámetro basta con pasar el parámetro que esperan recibir:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes@metodoEstatico('HolaAmigos
Estaticos')" />
<s:property value="new
com.javatutoriales.struts2.ognl.Constantes().metodoDeInstancia('Hola Amigos')" />
```

Inclusive podemos pasar como parámetro uno de los atributos obtenidos anteriormente:

```
<li><s:property value="new com.javatutoriales.struts2.ognl.Constantes().metodoDeInstancia(@com.javatutoriales.struts
2.ognl.Constantes@valor)" /></li>
```

Con estos cambios obtenemos la siguiente pantalla:



Otra parte interesante del uso de **OGNL** viene cuando queremos obtener los valores de las constantes de una enumeración. Si recuerdan, nuestra clase "**Constantes**" tiene una enumeración llamada "**Datos**". Como cada uno de los valores de una enumeración es una constante (o sea una variable publica, estática, y final) obtener sus valores es igual a obtener cualquier otro valor de una variable estática. O sea que tenemos que usar la sintaxis de la doble arroba.

Ahora, el secreto aquí viene dado en el sentido de que las enumeraciones son en realidad un tipo especial de clase Java. Como aquí la enumeración está dentro de otra clase, la primera se convierte en una clase interna de la segunda. El fully qualified class name de una clase interna es el nombre de la primer clase, seguida de un signo de dólar (\$) seguido del

nombre de la clase interna. Por lo tanto, el fully qualified class name de la enumeración "**Datos**" es "**com.javatutoriales.struts2.ognl.Constantes\$Datos**". Teniendo el nombre de esta enumeración, podemos obtener los valores de sus constes de la siguiente forma:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes$Datos@PRIMERO" />
<s:property value="@com.javatutoriales.struts2.ognl.Constantes$Datos@SEGUNDO" />
```

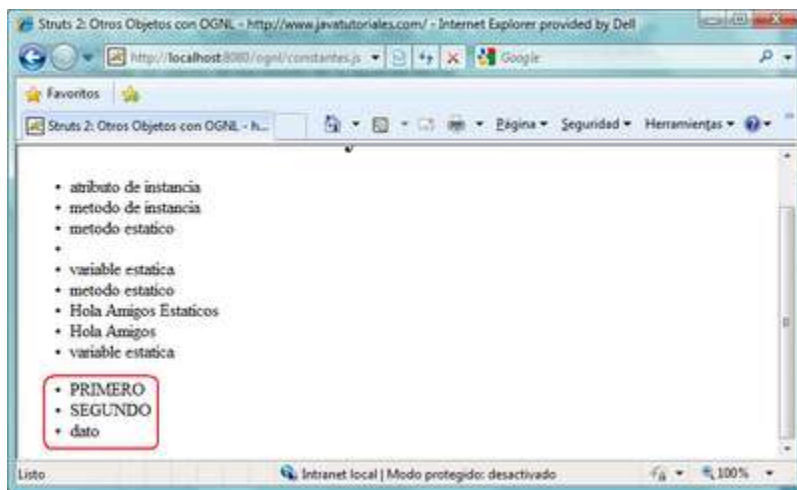
E igual que en los casos anteriores, teniendo estos valores podemos invocar cualquier método que exista dentro de la enumeración, como en este caso el método "**getDato()**" que invocamos de esta forma:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes$Datos@TERCERO.dato" />
```

Que también podría ser de esta forma:

```
<s:property value="@com.javatutoriales.struts2.ognl.Constantes$Datos@TERCERO.getDato()" />
```

En cualquiera de los dos casos, la salida obtenida es la siguiente:



Por último, veremos que en **OGNL** también se pueden declarar y acceder valores de tipos indexados, en particular de arreglos, listas, y mapas.

Obtenido Valores de Tipos Indexados con OGNL

Comencemos viendo los arreglos, que son los elementos más "complicados" de definir.

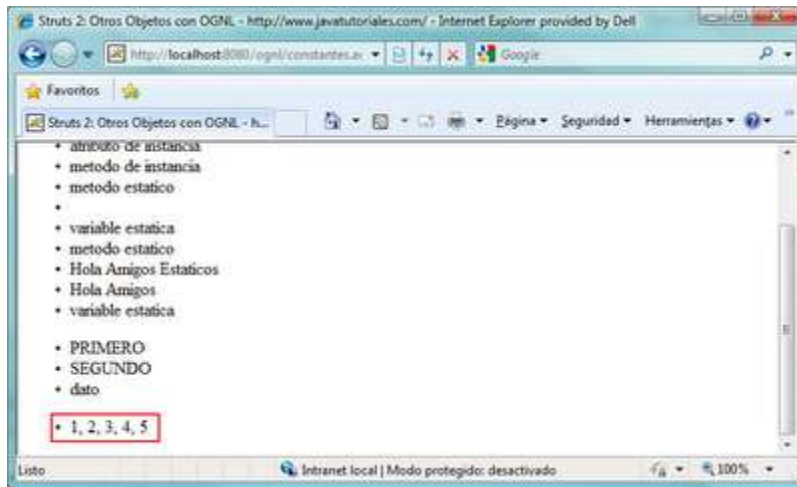
Quando queremos definir un arreglo de objetos o de primitivos, debemos usar la misma sintaxis que se usa para definir los arreglos anónimos. Por ejemplo, para definir un arreglo con los enteros del 1 al 5 lo hacemos de la siguiente forma:

```
new int[] {1, 2, 3, 4, 5}
```

Colocando esto en la etiqueta "**<s:property>**", de esta forma:

```
<s:property value="new int[]{1, 2, 3, 4, 5}" />
```

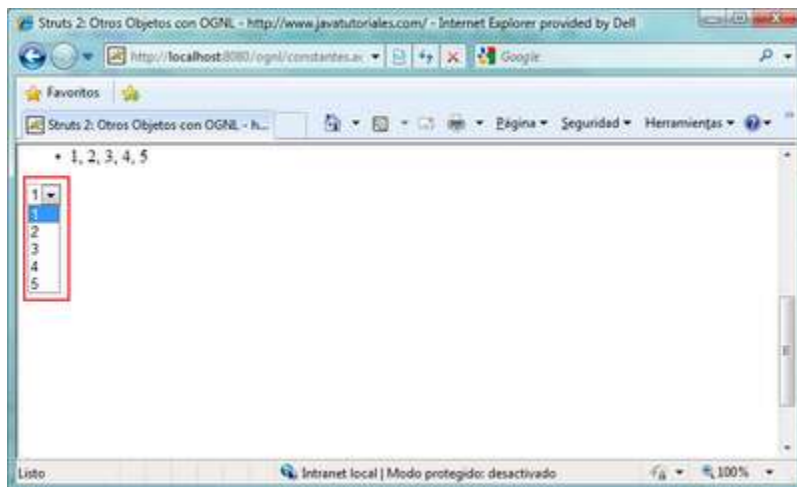
Obtenemos la siguiente salida:



Podemos colocar estos mismos elementos dentro de una etiqueta "<s:select>", que nos ayuda a crear listas desplegables (o combo boxes, o como quiera que gusten llamarle):

```
<s:select name="valores" list="new int[]{1, 2, 3, 4, 5}" />
```

Obteniendo la siguiente salida:



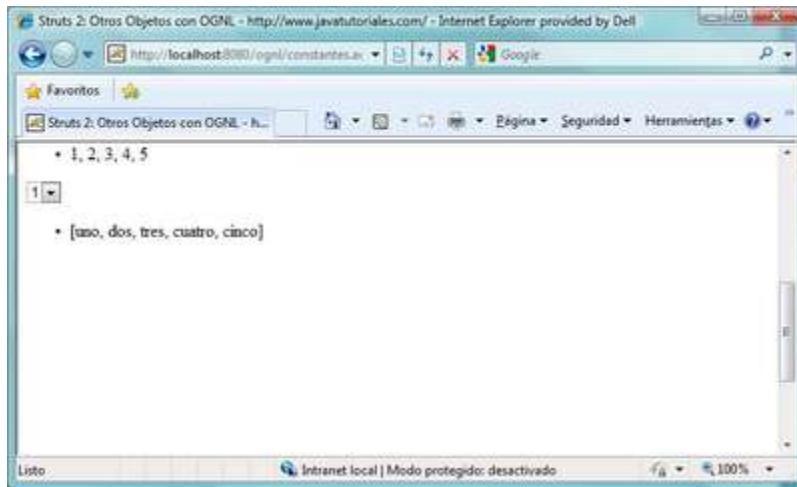
Para crear un arreglo con todos sus valores inicializados a **null**, podemos hacerlo de la siguiente forma:

```
new int[5]
```

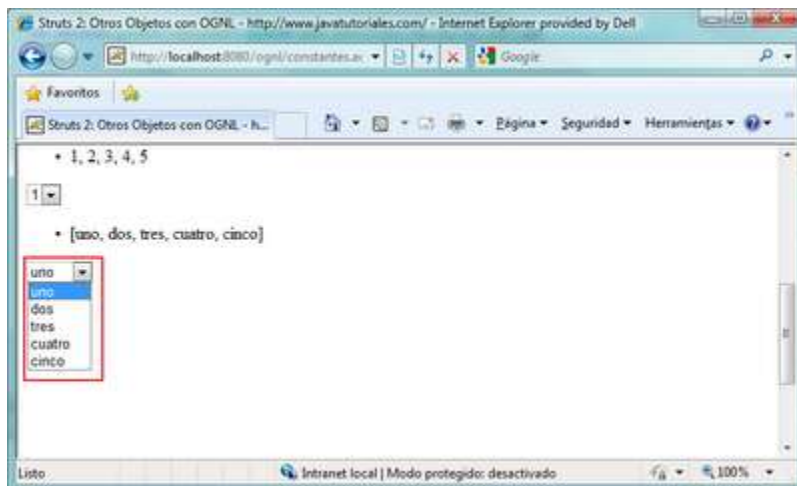
Si por alguna razón no podemos (o no queremos) hacer uso de arreglos, sino de algún tipo de colección, como una lista, podemos también construir una "al vuelo" usando la siguiente sintaxis:

```
{'uno', 'dos', 'tres', 'cuatro', 'cinco'}
```

Con esto obtenemos la siguiente salida:



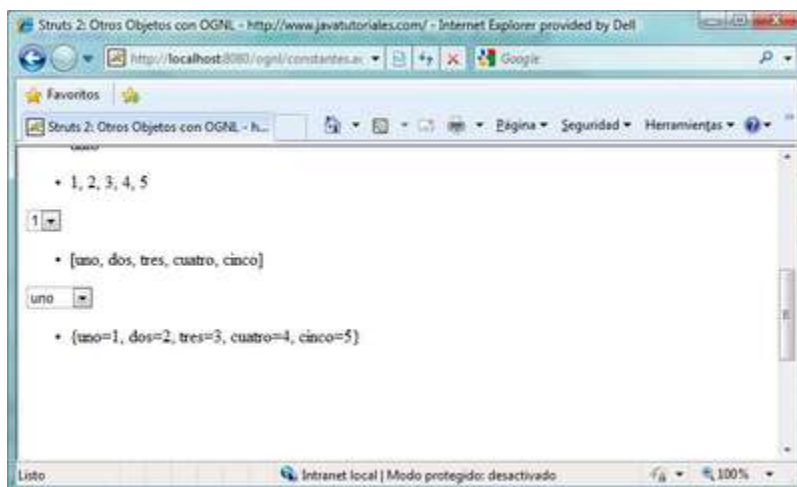
De la misma forma que con los arreglos, si colocamos esto dentro de una etiqueta "<s:select>" obtenemos la siguiente salida:



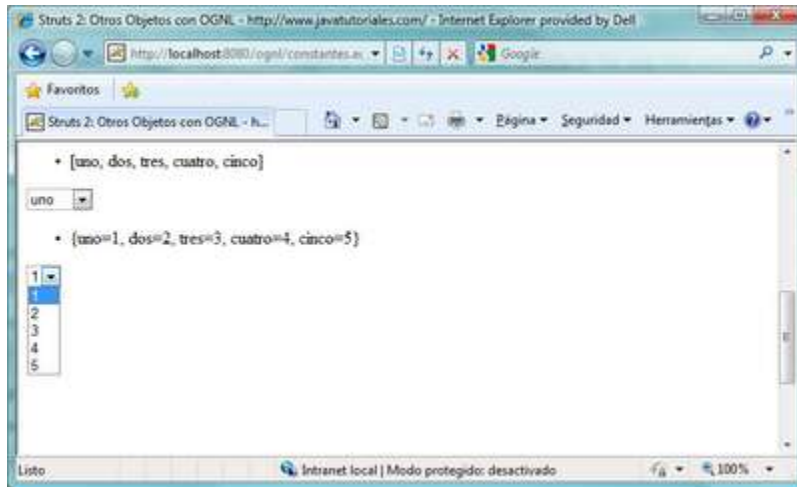
Finalmente, para crear un **Map**, podemos usar la siguiente sintaxis:

```
#{'uno':'1', 'dos':'2', 'tres':'3', 'cuatro':'4', 'cinco':'5'}
```

Donde el primer elemento es la llave, y el segundo es el valor. Con esto obtenemos la siguiente salida:



Igual que en los ejemplos anteriores, si colocamos esto en una etiqueta "**<s:select>**", obtenemos la siguiente pantalla:



Si vemos el código fuente del elemento generado podremos observar que la llave es usada como el atributo "**value**" de cada opción, y el valor se usa como la etiqueta que será mostrada:

```
<select name="mapas" id="mapas">
<option value="uno">1</option>
<option value="dos">2</option>
<option value="tres">3</option>
<option value="cuatro">4</option>
<option value="cinco">5</option>
</select>
```

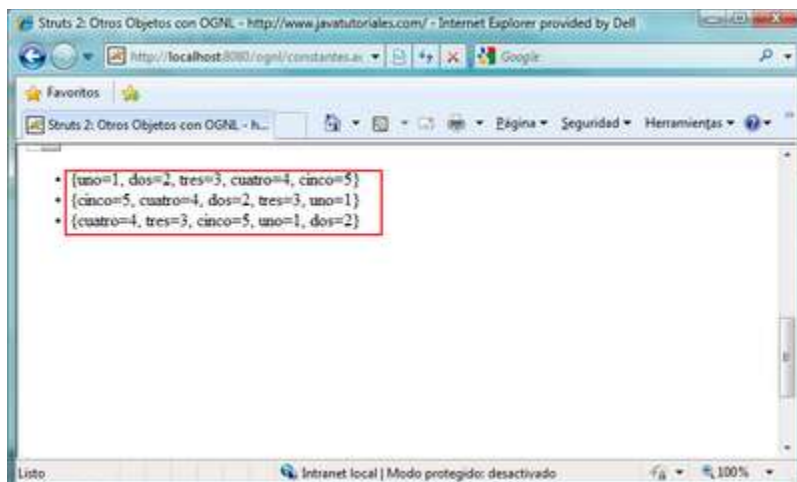
Si por alguna razón necesitamos alguna implementación particular de un **Map**, podemos indicarlo de la siguiente forma:

```
#@java.util.TreeMap@{'uno':'1', 'dos':'2', 'tres':'3', 'cuatro':'4', 'cinco':'5'}
```

Por ejemplo, colocando los siguientes elementos:

```
<s:property value="#@java.util.LinkedHashMap@{'uno':'1', 'dos':'2', 'tres':'3', 'cuatro':'4', 'cinco':'5'}" />
<s:property value="#@java.util.TreeMap@{'uno':'1', 'dos':'2', 'tres':'3', 'cuatro':'4', 'cinco':'5'}" />
<s:property value="#@java.util.HashMap@{'uno':'1', 'dos':'2', 'tres':'3', 'cuatro':'4', 'cinco':'5'}" />
```

Obtenemos la siguiente salida:



Podemos determinar si en una colección existe un elemento podemos usar los operadores "**in**" y "**not in**":

```
<s:if test="'foo' in {'foo','bar'}">
    muahahaha
</s:if>
<s:else>
    boo
</s:else>
```

```
<s:if test="'foo' not in {'foo','bar'}">
    muahahaha
</s:if>
<s:else>
    boo
</s:else>
```

Si necesitamos realizar la selección de un subconjunto de una colección (que en términos formales se llama proyección), **OGNL** proporciona un conjunto de comodines para eso:

- "?" – Para obtener todos los elementos que coinciden con la lógica de selección.
- "^" – Para seleccionar solo el primer elemento que coincida con la lógica de selección.
- "\$" - Para seleccionar solo el último elemento que coincida con la lógica de selección.

Por ejemplo, si tuviéramos una clase llamada "**Persona**" con un atributo "**genero**", y un objeto con una colección de **Personas** llamada "**familiares**". Si solo queremos obtener las **Personas** con genero "**masculino**", podemos hacerlo de la siguiente forma:

```
persona.familiares.{? #this.genero = 'masculino'}
```

Queda al lector hacer un ejemplo de esto ^_^.

Como nota final, hay que decir que algunas veces (en muy raras ocasiones en realidad) **OGNL** entiende las instrucciones que le damos como cadenas y no como expresiones. Para estos casos existe una forma de decirle que interprete lo que ve como una expresión, y esto es envolviendo dicha expresión entre los caracteres "%{" y "}".

[Por](#) ejemplo, para indicarle que la siguiente línea debe ser tomada como una expresión:

```
<s:property value="constantes.atributo" />
```

Lo indicamos de la siguiente forma:

```
<s:property value="%{constantes.atributo}" />
```

Como pudimos ver a lo largo de los ejemplos, esto casi nunca es necesario, pero si alguna vez, por alguna razón una de las expresiones que están creando no funciona, pueden resolverlo de esta forma ^_^.

Struts 2 - Parte 3: Trabajo con Formularios

En el desarrollo de aplicaciones web, una de las partes más importantes [que](#) existen (sino es que la más importante) es el manejo de datos que son recibidos del usuario a través de los formularios de nuestra aplicación.

Aunque es algo que usamos (y hacemos) todos los días, el manejo de los datos de los formularios puede ser un poco engañoso, [por](#) no decir complicado, cuando comenzamos a trabajar con la recepción de múltiples valores para un mismo parámetro, o cuando de antemano no conocemos los nombres de los parámetros que recibiremos; esto sin mencionar las validaciones para los distintos tipos de datos, la carga y descarga de archivos, etc.

En este tutorial aprenderemos la forma en la que se trabaja con formularios en **Struts 2**, y cómo manejar todas las situaciones mencionadas anteriormente. Concretamente aprenderemos cómo hacer **7 cosas**: recepción de parámetros simples, cómo hacer que el framework llene de forma automática los atributos de un objeto si es que todos los datos del formulario pertenecen a ese objeto, a recibir múltiples valores para un mismo parámetro, cómo recibir parámetros cuando no conocemos el nombre de los mismos, a realizar validaciones de datos de varias maneras, cómo subir archivos al servidor, y cómo enviar archivos desde el servidor hacia nuestros clientes.

En [el primer tutorial de la serie](#) vimos los pasos básicos para enviar datos a nuestros **Actions** a través de un formulario, sin embargo en esta ocasión veremos algunos conceptos un poco más avanzados que nos harán la vida más fácil cuando trabajemos con formularios.

Lo primero que haremos es crear un nuevo proyecto en NetBeans. Vamos al menú "**File -> New Project...**". En la ventana que aparece seleccionamos la categoría "**Java Web**" y en el tipo de proyecto "**Web Application**". Presionamos el botón "**Next >**" y le damos un nombre y una ubicación a nuestro proyecto; presionamos nuevamente el botón "**Next >**" y en este punto se nos preguntará el servidor que queremos usar. En nuestro caso usaremos el servidor "**Tomcat 7.0**", con la versión 5 de JEE y presionamos el botón "**Finish**".

Una vez que tengamos nuestro proyecto debemos recordar agregar la biblioteca "**Struts2**" (o "**Struts2Anotaciones**" si van a hacer uso de anotaciones, [como](#) es mi caso ^_^), que creamos en [el primer tutorial de la serie](#). Hacemos clic derecho sobre el nodo "**Libraries**" del proyecto. En el menú que aparece seleccionamos la opción "**Add Library...**". En la ventana que aparece seleccionamos la biblioteca "**Struts2**" o "**Struts2Anotaciones**" y presionamos "**Add Library**". Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto.

Ahora configuramos el filtro "**struts2**" en el deployment descriptor. Abrimos el archivo "**web.xml**" y colocamos el siguiente contenido, como se explicó en el primer tutorial de la serie:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Ahora procedemos a crear una clase que nos servirá como modelo de datos. Esta será una clase "**Usuario**" la cual tendrá atributos de varios tipos para que veamos como **Struts 2** realiza su "magia".

Creamos un paquete para nuestro modelo de datos. En mi caso el primer paquete se llamará "**com.javatutoriales.struts2.formularios.modelo**". Hacemos clic derecho sobre el nodo "**Source packages**" del proyecto, en el menú que aparece seleccionamos la opción "**new -> package**". En la ventana que aparece le damos el nombre correspondiente a nuestro paquete.

Ahora, dentro de este paquete creamos la clase "**Usuario**": sobre el paquete que acabamos de crear hacemos clic derecho y en el menú contextual que aparece seleccionamos la opción "**new -> Java Class**". En la ventana que aparece le damos el nombre "**Usuario**" y hacemos clic en el botón "**Finish**". Con esto aparecerá en nuestro editor la clase "**Usuario**" de la siguiente forma:

```
public class Usuario
{
}
```

Antes que nada hacemos que esta clase implemente la interface "**java.io.Serializable**" como deben hacerlo todas las clases de transporte de datos:

```
public class Usuario implements Serializable
{
}
```

Ahora agregaremos unos cuantos atributos a esta clase, de distintos tipos, representando algunas de las características de nuestro usuario. Dentro de estos atributos incluiremos su nombre, edad, y fecha de nacimiento:

```
public class Usuario implements Serializable
{
    private String nombre;
    private String username;
    private String password;
    private int edad;
    private Date fechaNacimiento;
}
```

Recuerden que el "**Date**" de "**fechaNacimiento**" debe ser de tipo "**java.util.Date**".

Para terminar con la clase **Usuario** agregaremos los **getters** y los **setters** de estos atributos, y adicionalmente dos constructores, uno que reciba todos los atributos, y uno vacío:

```
public class Usuario implements Serializable
{
    private String nombre;
    private String username;
    private String password;
    private int edad;
    private Date fechaNacimiento;

    public Usuario()
    {
    }

    public Usuario(String nombre, String username, String password, int edad, Date fechaNacimiento)
    {
        this.nombre = nombre;
        this.username = username;
        this.password = password;
        this.edad = edad;
        this.fechaNacimiento = fechaNacimiento;
    }
}
```

```

    public int getEdad()
    {
        return edad;
    }

    public void setEdad(int edad)
    {
        this.edad = edad;
    }

    public Date getFechaNacimiento()
    {
        return fechaNacimiento;
    }

    public void setFechaNacimiento(Date fechaNacimiento)
    {
        this.fechaNacimiento = fechaNacimiento;
    }

    public String getNombre()
    {
        return nombre;
    }

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

    public String getPassword()
    {
        return password;
    }

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

    public String getUsername()
    {
        return username;
    }

    public void setUsername(String username)
    {
        this.username = username;
    }
}

```

Lo primero que haremos es recordar cómo podemos obtener parámetros planos o simples desde nuestros formularios:

1. Recepción de parámetros simples

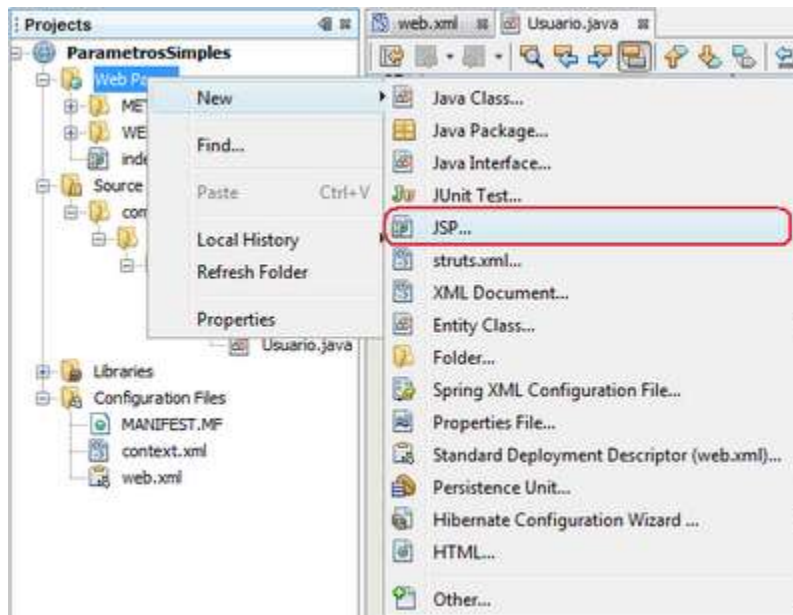
En [el primer tutorial de la serie de Struts 2](#) vimos de una forma muy rápida cómo obtener parámetros de un formulario. Lo único que hay que hacer es:

- Colocar un formulario en nuestra **JSP** usando la etiqueta **<s:form>**
- Colocar los campos del formulario usando las etiquetas correspondientes de **Struts**
- Crear un **Action** y colocar **setters** para cada uno de los elementos que recibiremos a través del formulario
- Procesar los datos del formulario en el método **"execute"**

Hagamos un pequeño ejemplo para refrescar la memoria.

En este primer ejemplo obtendremos los datos para crear un nuevo objeto **"Usuario"**, el tipo que definimos anteriormente.

Lo primero que hacemos es crear una nueva página **JSP**, en la raíz de las páginas web, llamada **"nuevo-usuario.jsp"**. Hacemos clic derecho en el nodo **"Web Pages"**. En el menú que se abre seleccionamos la opción **"New -> JSP..."**.



Colocamos **"nuevo-usuario"** como nombre de la página y presionamos el botón **"Finish"**.

En esta página indicamos que se usará la biblioteca de etiquetas de **Struts 2**:

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

Usamos la etiqueta **"<s:form>"** para colocar un formulario en nuestra página. En su atributo **"action"** colocamos el nombre del **Action** que se encargará de procesar los datos de este formulario:

```
<s:form action="datosUsuario">  
</s:form>
```

Dentro de este formulario colocaremos un campo para cada uno de los atributos que puede recibir un Usuario:

```

<s:form action="datosUsuario">
<s:textfield name="nombre" label="Nombre" />
<s:textfield name="username" label="Username" />
<s:password name="password" label="Password" />
<s:textfield name="edad" label="Edad" />
<s:textfield name="fechaNacimiento" label="Fecha de Nacimiento" />
<s:submit value="Enviar" />
</s:form>

```

Ya que nuestro formulario está listo, crearemos el **Action** que se encargará de procesar los datos del mismo.

Creamos un nuevo paquete, llamado "**actions**", a la misma altura que el paquete "**modelo**". Dentro de este paquete creamos una nueva clase Java llamada "**UsuarioAction**". Haremos que esta clase extienda de "**ActionSupport**":

```

public class UsuarioAction extends ActionSupport
{
}

```

Recuerden que para recibir los datos del formulario debemos colocar los atributos en los que se almacenarán estos datos, y **setters** para que los interceptores correspondientes puedan inyectar los valores dentro del **Action**:

```

public class UsuarioAction extends ActionSupport
{
    private String nombre;
    private String username;
    private String password;
    private int edad;
    private Date fechaNacimiento;

    public void setEdad(int edad)
    {
        this.edad = edad;
    }

    public void setFechaNacimiento(Date fechaNacimiento)
    {
        this.fechaNacimiento = fechaNacimiento;
    }

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

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

    public void setUsername(String username)
    {
        this.username = username;
    }
}

```

Cuando esté **Action** termine su ejecución (la cual aún no hemos implementado), queremos poder crear un **Usuario** y mostrar sus datos en otra página. Para hacer eso debemos colocar un atributo que almacene una referencia al **Usuario**, y un **getter** para [poder](#) obtener esta referencia:

```
private Usuario usuario;

public Usuario getUsuario()
{
    return usuario;
}
```

Ahora sí, sobre-escribimos el método "**execute**" para crear un nuevo objeto de tipo **Usuario** y establecer sus datos usando los valores que recibimos del formulario:

```
@Override
public String execute() throws Exception
{
    usuario = new Usuario();
    usuario.setNombre(nombre);
    usuario.setUsername(username);
    usuario.setPassword(password);
    usuario.setEdad(edad);
    usuario.setFechaNacimiento(fechaNacimiento);

    return SUCCESS;
}
```

Y eso es todo, ya hemos creado un nuevo **Usuario** con los datos que recibimos a través del formulario de captura.

Para que este ejemplo funcione aún tenemos que hacer un par de cosas. Primero marcaremos nuestra clase con la anotación "**@Action**" para indicar que esta clase debe ser tratada con un **Action** de **Struts 2**, como lo vimos en [el primer tutorial de la serie](#):

```
@Namespace(value="/")
@Action(value="datosUsuario", results={@Result(name="success", location="/datos-usuario.jsp")})
public class UsuarioAction extends ActionSupport
{
}
```

Como podemos ver, el nombre de nuestro **Action** es "**datosUsuario**" (que es el mismo que colocamos en el atributo "**action**" del formulario que creamos hace un momento). Si todo el proceso de nuestro **Action** sale bien seremos enviados a la página "**datosUsuario.jsp**". Esta página solamente mostrará los datos del usuario que se acaba de crear. Antes de ver esta página veamos cómo queda la clase "**UsuarioAction**":

```
@Namespace(value="/")
@Action(value="datosUsuario", results={@Result(name="success", location="/datos-usuario.jsp")})
public class UsuarioAction extends ActionSupport
{
    private String nombre;
    private String username;
    private String password;
    private int edad;
    private Date fechaNacimiento;
}
```



```

private Usuario usuario;

@Override
public String execute() throws Exception
{
    usuario = new Usuario();
    usuario.setNombre(nombre);
    usuario.setUsername(username);
    usuario.setPassword(password);
    usuario.setEdad(edad);
    usuario.setFechaNacimiento(fechaNacimiento);

    return SUCCESS;
}

public Usuario getUsuario()
{
    return usuario;
}

public void setEdad(int edad)
{
    this.edad = edad;
}

public void setFechaNacimiento(Date fechaNacimiento)
{
    this.fechaNacimiento = fechaNacimiento;
}

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

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

public void setUsername(String username)
{
    this.username = username;
}
}

```

Ahora sí, creamos una nueva página **JSP**; el nombre de esta página será "**datos-usuario**". En esta página haremos uso de algunas etiquetas de **Struts**, por lo que deberemos indicarlo usando la directiva "**taglib**" correspondiente:

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

En esta página solamente mostraremos los datos del usuario que acabamos de crear. Recordemos que para esto debemos usar la etiqueta "**<s:property>**". Mostramos cada uno de los datos anteriores del usuario:

```
Nombre: <strong><s:property value="usuario.nombre" /></strong><br />
Username: <strong><s:property value="usuario.username" /></strong><br />
Password: <strong><s:property value="usuario.password" /></strong><br />
Edad: <strong><s:property value="usuario.edad" /></strong><br />
Fecha de Nacimiento: <strong><s:property value="usuario.fechaNacimiento" /></strong>
```

Cuando entremos a la siguiente página:

<http://localhost:8080/formularios/nuevo-usuario.jsp>

Veremos el formulario para capturar los datos del usuario. En mi caso, con los siguientes datos:



Datos de Usuario

Nombre: Ale

Username: programadorjava

Password: *****

Edad: 15

Fecha de Nacimiento: 22/07/2011

Enviar

Obtengo la siguiente salida:



Nombre: Ale

Username: programadorjava

Password: 123456

Edad: 15

Fecha de Nacimiento: 22/07/11

En el ejemplo anterior podemos ver que aunque hemos pasado pocos datos al formulario, todos los hemos establecido en el objeto "**Usuario**", que se creó dentro método "**execute**", y los pasamos "como van", es decir sin realizar ninguna transformación o procesamiento sobre ellos. Cuando tenemos 5 o 6 atributos esto puede no ser algo muy pesado. Pero ¿qué pasa si tenemos que llenar un objeto con 30 o 40 propiedades? Nuestro **Action** sería muy grande por todos los atributos y los **setters** y **getters** de las propiedades; el trabajo dentro del método "**execute**" también sería bastante cansado llamar a los **setters** de nuestros modelos para pasarle los parámetros.

Para esos casos **Struts 2** proporciona una forma de simplificarnos estas situaciones usando un mecanismo conocido como "**Model Driven**" el cual veremos a continuación.

2. Model Driven

Model Driven es una forma que **Struts 2** proporciona para poder establecer todos los parámetros que se reciben de un formulario directamente dentro de un objeto. Este objeto es conocido como el **modelo**.

Usando este modelo nos evitamos [estar](#) llamando nosotros mismos a los **setters** de este objeto modelo.

Esto también permite que si realizamos validaciones de datos del formulario (lo cual veremos cómo hacer un poco más adelante) estas se realizarán sobre este objeto modelo.

Un interceptor especial, llamado model driven interceptor, se encarga de manejar todo esto de forma automática ^_^.

Extenderemos nuestro ejemplo para usar model driven.

Creamos una nueva clase Java, en el paquete "**actions**", llamada "**UsuarioActionMD**". Esta clase extenderá de "**ActionSupport**":

```
public class UsuarioActionMD extends ActionSupport
{
}
```

Para indicarle a **Struts 2** que este **Action** será **Model Driven**, la clase "**UsuarioActionMD**" debe implementar la interface "**ModelDriven**", indicando de qué tipo de objeto será usado como modelo:

```
public class UsuarioActionMD extends ActionSupport implements ModelDriven<Usuario>
{
}
```

Ahora pondremos un objeto **Usuario** dentro de nuestra clase, con una variable de instancia, que será el objeto que usaremos como modelo:

```
public class UsuarioActionMD extends ActionSupport implements ModelDriven<Usuario>
{
    private Usuario usuario = new Usuario();
}
```

Es importante tener creado este objeto antes de que nuestro **Action** reciba alguna petición, por lo que podemos inicializarlo en la misma declaración o en el constructor del **Action**, en caso de tener alguno.

La interface "**ModelDriven**" tiene tan solo un método: "**getModel**". Este método no recibe ningún parámetro, y regresa el objeto que estamos usando como modelo:

```
public Usuario getModel()
{
    return usuario;
}
```

Lo único que queda es realizar algún proceso en el método "**execute**", que podría ser almacenar al **Usuario** en alguna base de datos, enviarlo por algún stream, etc. Como en este caso no haremos nada con el **Usuario** más que regresarlo para poder mostrar sus datos en una página, nuestro método "**execute**" solo regresará un valor de "**SUCCESS**".

```

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

```

Noten que en esta ocasión no es necesario tener un **getter** para poder obtener la instancia del "**Usuario**" desde nuestra **JSP**.

Finalmente, anotamos nuestra clase para que **Struts 2** lo reconozca como un **Action**. El nombre del **Action** será "**datosUsuario**". Para diferenciar este **Action** del anterior (que tiene el mismo nombre) lo colocaremos en el namespace "**modeldriven**".

El **Action** al terminar de ejecutarse nos enviará a la página **"/modeldriven/datos-usuario.jsp"**:

```

@Namespace(value = "/modeldriven")


```

La clase "**UsuarioActionMD**" completa (omitiendo los imports) queda de la siguiente forma:

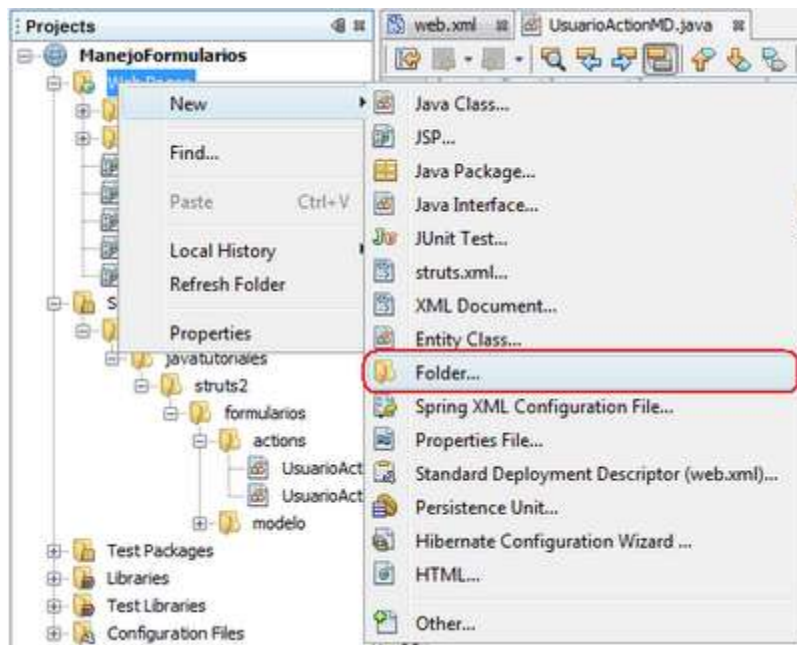
```

@Namespace(value = "/modeldriven")


```

Podemos ver que el código de este **Action** es mucho más pequeño que el que hicimos para el ejemplo anterior.

Ahora antes de crear la página que se encargará de mostrar el resultado, creamos un nuevo directorio para nuestras páginas web. Para eso hacemos clic derecho sobre el nodo "**WebPages**" de nuestro proyecto. En el menú que aparece seleccionamos la opción "**New -> Folder**":



Damos "**modeldriven**" como nombre del directorio y presionamos el botón "**Finish**".

Dentro de este directorio crearemos dos páginas. Primero crearemos la página que contendrá el formulario que nos permitirá capturar los datos del usuario. Hacemos clic derecho en el directorio "**modeldriven**" del nodo "**Web Pages**". En el menú que se abre seleccionamos la opción "**New -> JSP...**". Damos "**nuevo-usuario**" como nombre de la página y presionamos el botón "**Finish**".

El formulario de la página será idéntico al que creamos en la primer parte del tutorial, con la única diferencia de que enviará los datos al **Action "datosUsuarioMD"**:

```
<s:form action="datosUsuario">
<s:textfield name="nombre" label="Nombre" />
<s:textfield name="username" label="Username" />
<s:password name="password" label="Password" />
<s:textfield name="edad" label="Edad" />
<s:textfield name="fechaNacimiento" label="Fecha de Nacimiento" />
<s:submit value="Enviar" />
</s:form>
```

La segunda página, que se llamará "**datos-usuario.jsp**" y que también estará en el directorio "**modeldriven**", mostrará los valores de los atributos del objeto "**usuario**" que se usa como modelo. Esta página también será muy parecida a la que creamos anteriormente, solo que en esta ocasión no será necesario indicar que los atributos que estamos usando corresponden al objeto "**usuario**":

```
Nombre: <strong><s:property value="nombre" /></strong><br />
Username: <strong><s:property value="username" /></strong><br />
Password: <strong><s:property value="password" /></strong><br />
Edad: <strong><s:property value="edad" /></strong><br />
Fecha de Nacimiento: <strong><s:property value="fechaNacimiento" /></strong>
```

Al ejecutar la aplicación y entrar en la siguiente dirección:

<http://localhost:8080/formularios/modeldriven/nuevo-usuario.jsp>

Veremos un formulario muy parecido al anterior:

Cuando llenemos los datos y enviemos el formulario, deberemos ver una salida también muy parecida a la anterior:

Pero ¿qué ocurre si queremos recibir un parámetro que no sea un atributo del "**usuario**"? En ese caso, solo tenemos que agregar un nuevo atributo, con su correspondiente **setter** para establecer su valor (y su **getter** si es que pensamos recuperarlo posteriormente) dentro de nuestro **Action**.

Modifiquemos un poco el ejemplo anterior para ver esto. Agregaremos al formulario un campo de texto para que el usuario proporcione un número de confirmación:

```
<s:form action="datosUsuario">
<s:textfield name="nombre" label="Nombre" />
<s:textfield name="username" label="Username" />
<s:password name="password" label="Password" />
<s:textfield name="edad" label="Edad" />
<s:textfield name="fechaNacimiento" label="Fecha de Nacimiento" />
<s:textfield name="numero" label="Número de Confirmación" />

<s:submit value="Enviar" />
</s:form>
```

También cambiaremos nuestro **Action**. Colocamos un atributo de tipo entero para contener este número, junto con sus correspondientes **setter** (para establecer el valor desde el formulario) y **getter** (para obtener el valor desde la **JSP**):

```
private int numero;

public void setNumero(int numero)
{
    this.numero = numero;
}

public int getNumero()
{
    return numero;
}
```

De igual forma modificaremos la página "**datos-usuarios.jsp**" del directorio "**modeldriven**" para poder mostrar el valor del número de confirmación:

```
Nombre: <strong><s:property value="nombre" /></strong><br />
Username: <strong><s:property value="username" /></strong><br />
Password: <strong><s:property value="password" /></strong><br />
Edad: <strong><s:property value="edad" /></strong><br />
Fecha de Nacimiento: <strong><s:property value="fechaNacimiento" /></strong><br />
Número de confirmación: <strong><s:property value="numero" /></strong>
```

Al ejecutar la aplicación y entrar al formulario, veremos el campo que hemos agregado:

Al enviar los datos del formulario obtendremos la siguiente pantalla:

Ahora veremos cómo podemos hacer para recibir un conjunto de valores en un solo atributo de nuestro **Action**.

3. Recepción de múltiples parámetros

En los ejemplos anteriores cada uno de nuestros atributos recibe solo un valor de algún tipo. Sin embargo algunas veces nos será conveniente recibir un conjunto de parámetros, ya sea como un arreglo de valores, o como una lista de valores para poder procesar cada uno de sus elementos.

Esto es útil para cuando, por ejemplo, tenemos campos de formulario que se van generando dinámicamente, o que tienen el mismo significado en datos (por ejemplo cuando tenemos varios campos para cargar archivos adjuntos en correos electrónicos, o para poder subir varias imágenes de una vez a un sitio):



The screenshot shows a web form titled "Adjuntar" (Attach). It features a tab labeled "Mi PC". Below the tab, there are five rows, each containing an "Adjuntar:" label, a text input field, an "Examinar..." button, and a blue "Eliminar" link. At the bottom of the form, there is a button labeled "Añadir más archivos adjuntos". In the bottom right corner, there are two buttons: "Adjuntar" and "Cancelar".

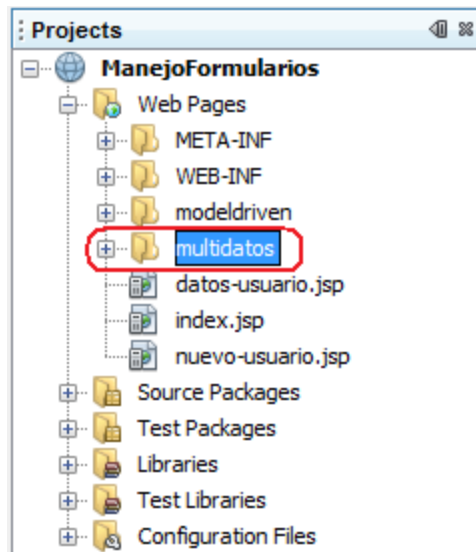
Poder obtener estos parámetros es sencillo gracias a los interceptores de **Struts 2**.

Veamos un ejemplo parecido a la pantalla anterior. Tendremos un formulario que nos permitirá recibir un conjunto de correos electrónicos de personas para ser procesados en el servidor. Colocaremos de forma estática 5 campos (claro que estos podrían ir aumentando de forma dinámica usando JavaScript).

Aunque el formulario tendrá 5 campos para correos, el usuario podría no llenarlos todos, por lo que no sabremos cuántos elementos estaremos recibiendo.

El formulario tendrá además un campo que permitirá que el usuario coloque su correo nombre, para saber quién está enviando los correos.

Comencemos primero con el formulario que mostrará los campos para los correos. Crearemos un nuevo directorio, en la raíz de las páginas web del proyecto, llamado "**multidatos**" (lo sé, no es un nombre muy original, pero refleja el cometido del ejemplo):



En este directorio crearemos una nueva JSP llamada "**datos.jsp**". Esta página contendrá el formulario con los 6 campos mencionados antes: un campo para que el usuario coloque su nombre, y 5 para colocar una dirección de correo electrónico:

```
<s:form action="envioCorreo">
<s:textfield name="nombre" label="Nombre" />

<s:textfield name="correo" label="Correo" />
<s:textfield name="correo" label="Correo" />
<s:textfield name="correo" label="Correo" />
<s:textfield name="correo" label="Correo" />
<s:textfield name="correo" label="Correo" />

<s:submit value="Enviar" />
</s:form>
```

No olviden indicar que usaremos las etiquetas de **Struts 2**, con la directiva **taglib** correspondiente:

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

Podemos ver en el formulario que los 5 campos para colocar los correos electrónicos son exactamente iguales, todos tienen por nombre "**correo**". Esto es necesario para podamos recibir los valores como un solo conjunto de datos.

Veamos el **Action** que recibirá estos valores. Primero crearemos una nueva clase java, llamada "**MultiplesDatosAction**", en el paquete "**Actions**". Esta clase extenderá de **ActionSupport**:

```
public class MultiplesDatosAction extends ActionSupport
{
}
```

Ahora colocaremos el atributo que nos permitirá recibir los múltiples parámetros. Para esto debemos colocar un arreglo del tipo de datos que estemos esperando. En este caso será un arreglo de **Strings**:

```
private String[] correos;
```

En realidad, ese es todo el secreto para poder recibir estos múltiples parámetros ^_^ . Uno de los interceptores de **Struts 2** se encargará de recibir los parámetros del formulario que tienen el mismo nombre; este interceptor colocará los parámetros en un arreglo o una colección, dependiendo de qué es lo que estemos esperando, y lo colocará en nuestro **Action** usando el **setter** apropiado.

Según la explicación anterior, debemos colocar el **setter** del atributo llamado "**correos**" para poder establecer el arreglo de correos, y su getter para poder obtenerlo posteriormente:

```
public String[] getCorreos()
{
    return correos;
}

public void setCorreos(String[] correos)
{
    this.correos = correos;
}
```

Ahora que tenemos el arreglo de valores colocaremos un atributo para mantener el nombre del usuario, con su correspondiente **getter** y **setter**:

```
private String nombre;

public String getNombre()
{
    return nombre;
}

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

Como en esta ocasión no se realizará ningún proceso sobre los datos, por lo que nuestro método "**execute**" solo regresará un valor de "**SUCCESS**":

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

Para terminar anotaremos esta clase. Primero indicamos que este **Action** estará en el namespace "**multidatos**". Además responderá al nombre de "**envioCorreo**" (el mismo que pusimos en el formulario). Al terminar el proceso, el **Action** nos enviará a una página llamada "**datos-enviados.jsp**" del directorio "**multidatos**":

```
@Namespace(value="/multidatos")

```

El código completo de la clase "**MultiplesDatosAction**" queda de la siguiente forma:

```

@Namespace(value="/multidatos")
@Action(value="envioCorreo", results={@Result(location="/multidatos/datos-
enviados.jsp")})
public class MultiplesDatosAction extends ActionSupport
{
    private String[] correos;
    private String nombre;

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

    public String getNombre()
    {
        return nombre;
    }

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

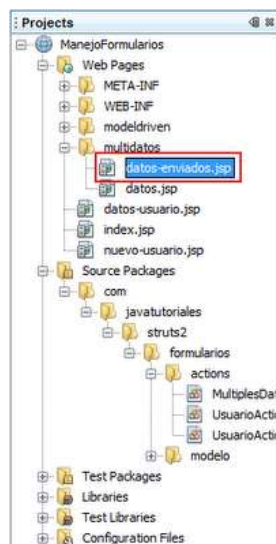
    public String[] getCorreos()
    {
        return correos;
    }

    public void setCorreos(String[] correos)
    {
        this.correos = correos;
    }
}

```

Ahora solo falta crear la página a la que regresaremos al terminar el **Action**.

Creemos, en el directorio "**multidatos**" de las páginas web, una nueva **JSP** llamada "**datos-enviados.jsp**":



En esta página lo que haremos será mostrar los mismos datos que recibimos del formulario.

Recuerden que primero debemos indicar que usaremos las etiquetas de **Struts 2**, usando el **taglib** correspondiente:

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

Primero mostraremos el valor del nombre del usuario, de la misma manera en la que hemos venido haciéndolo a lo largo de los ejemplos del tutorial:

```
<s:property value="nombre" />
```

Ahora mostraremos los valores de nuestro arreglo de cadenas, para eso **Struts 2** proporciona una etiqueta especial que nos permite iterar a través de todos los valores de un arreglo o colección, la etiqueta **"iterator"**.

La etiqueta **"iterator"** recibirá el arreglo o colección a través del cual ciclaremos para obtener sus valores. Indicamos este arreglo o colección usando el atributo **"value"**. En este caso indicaremos que el valor a través del cual queremos iterar es arreglo llamado **"correos"**:

```
<s:iterator value="correos">
</s:iterator>
```

Ya podemos ciclar a través de todos los elementos del arreglo, ahora necesitamos una manera de obtener el elemento actual dentro del ciclo. Para esto usamos el atributo **"var"**, en este indicaremos el nombre de la variable que mantendrá el valor del elemento actual. No debemos de preocuparnos por crear esta variable, ya que **Struts 2** la creará automáticamente y la pondrá a nuestra disposición. Esta variable será del tipo de elementos que sea mantenido por nuestro arreglo o colección:

```
<s:iterator value="correos" var="correo">
</s:iterator>
```

Algunas veces queremos poder obtener alguna información relativa a la iteración, como el índice del elemento actual, el número total de elementos que tiene el arreglo o colección, si el elemento actual es par o impar, etc. Para obtener esta información podemos indicarle a la etiqueta **"iterator"** que la coloque, como una instancia de la clase **"IteratorStatus"** y que la ponga a nuestra disposición en una variable. Para esto, indicamos el nombre de la variable en el atributo **"status"**:

```
<s:iterator value="correos" var="correo" status="estatus">
</s:iterator>
```

El estatus es una variable que no será colocada dentro en la raíz del stack de **Struts 2** (el cual explicamos en [el segundo tutorial de la serie](#)), por lo que es necesario hacer referencia a él usando el operador **"#"**.

Ahora que podemos iterar a través de cada uno de los elementos de nuestro arreglo, solo hace falta mostrar el valor de cada uno de estos elementos, adicionalmente también mostraremos su índice. Colocaremos estos valores en una lista para que se vea más presentable:

```
<ul>
<s:iterator value="correos" var="correo" status="estatus">
<li><s:property value="#estatus.index" /> - <s:property value="correo" /></li>
</s:iterator>
</ul>
```

Ahora, cuando coloquemos los siguientes datos en el formulario:

Obtendremos la siguiente salida:

Podemos ver que de una forma sencilla colocamos el conjunto de valores dentro del arreglo. Pero qué pasa si por alguna razón no podemos o no queremos usar un arreglo, y preferimos trabajar con una colección, como un **Set** o un **List**, lo único que tenemos que hacer es cambiar el tipo del atributo "**correos**" (junto con sus correspondientes **getters** y **setters**). Usemos un "**Set**", el cual es una colección que no permite elementos duplicados.

Modifiquemos el atributo "**correos**" para que quede de la siguiente forma:

```
private Set<String> correos;

public Set<String> getCorreos()
{
    return correos;
}

public void setCorreos(Set<String> correos)
{
    this.correos = correos;
}
```

Al introducir los siguientes datos en el formulario:

Obtendremos la siguiente salida:



Podemos ver que en realidad no fue necesario realizar grandes cambios para modificar la forma en la que recibimos los datos. Si quisiéramos que "**correos**" fuera una Lista, lo único que debemos hacer es, nuevamente, cambiar solo el tipo de dato.

El arreglo o la colección que usemos para mantener los valores, pueden ser de cualquier tipo de dato (cadenas, fechas, wrappers, archivos, etc).

Los ejemplos anteriores funcionan bien cuándo sabemos cuántos y cuáles serán los parámetros que estamos esperando recibir. Pero ¿y si no supiéramos de antemano los parámetros que recibiremos?

Como pueden imaginarse, **Struts 2** proporciona una manera elegante de manejar estas situaciones.

4. Recepción de Parámetros Desconocidos

Algunas veces estamos esperando recibir, dentro de nuestro **Action**, un conjunto de parámetros de los cuales no conocemos ni su cantidad ni sus nombres, como por ejemplo cuando estamos haciendo un filtrado de datos y no sabes qué filtros recibiremos y cuáles serán los valores de estos filtros, o cuando la generación de componentes se realiza de forma dinámica.

Para estos casos **Struts 2** proporciona una manera de indicar que deberá entregarnos todos los parámetros en un objeto tipo "**java.util.Map**". Las llaves de este mapa representarán los nombres de los parámetros, y sus valores representarán un arreglo de **Strings** con los valores correspondientes de cada parámetro. ¿Por qué un arreglo de **Strings**? Porque como acabamos de ver, algunos de los parámetros que recibimos pueden tener más de un valor.

Para lograr que **Struts 2** nos proporcione estos parámetros, nuestro **Action** debe implementar la interface "**ParameterAware**". Esta interface se ve de la siguiente forma:

```
public interface ParameterAware
{
    void setParameters(Map<String, String[]> parameters);
}
```

O sea, que esta interface solo tiene un método llamado "**setParameters**", que tiene como argumento el mapa de parámetros recibidos en la petición.

Veamos un ejemplo para entender cómo funciona esta interface.

Primero crearemos un nuevo directorio, dentro de nuestras páginas web, llamado "**multiparametros**". Dentro de este directorio crearemos dos nuevas **JSPs**. La primera, llamada "**datos.jsp**" contendrá un pequeño formulario que nos permitirá enviar los parámetros dinámicos a nuestro **Action**. La segunda, llamada "**parametros.jsp**", mostrara el nombre y el valor de cada uno de los parámetros que hemos colocado en el formulario. Comencemos creando la pagina "**datos.jsp**".

Primero, indicaremos que haremos uso de las etiquetas de **Struts 2** usando el taglib correspondiente:

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

Ahora crearemos un formulario, de una forma un poco distinta a como lo hemos estado haciendo hasta ahora. Primero indicaremos, con la etiqueta "**<s:form>**" que haremos uso de un formulario, los datos del mismo serán procesados por un **Action** cuyo nombre será "**multiparametros**":

```
<s:form action="multiparametros">
</s:form>
```

Para que sea más claro entender un paso que haremos un poco más adelante, en vez de dejar que **Struts 2** sea quien defina la estructura de nuestro formulario, lo haremos nosotros mismos. Para eso debemos indicar, en el atributo "**theme**" del formulario, el valor "**simple**":

```
<s:form action="multiparametros" theme="simple">
</s:form>
```

Ahora colocaremos un elemento en el formulario, un campo de entrada de texto usando el la etiqueta "**<s:textfield>**":

```
<s:form action="multiparametros" id="formulario" theme="simple">
<s:textfield id="valor1" name="valor1" />
</s:form>
```

Como en esta ocasión no se colocará una etiqueta de forma automática en el campo de texto, deberemos ponerla nosotros mismos, usando la etiqueta "**<s:label>**", de la siguiente manera:

```
<s:form action="multiparametros" theme="simple">
<s:label for="valor1" value="Valor 1: " />
<s:textfield id="valor1" name="valor1" />
</s:form>
```

Como pueden ver hemos tenido que colocar un valor en el atributo "**id**" del textfield, y poner este mismo valor en el atributo "**for**" de la etiqueta "**<s:label>**".

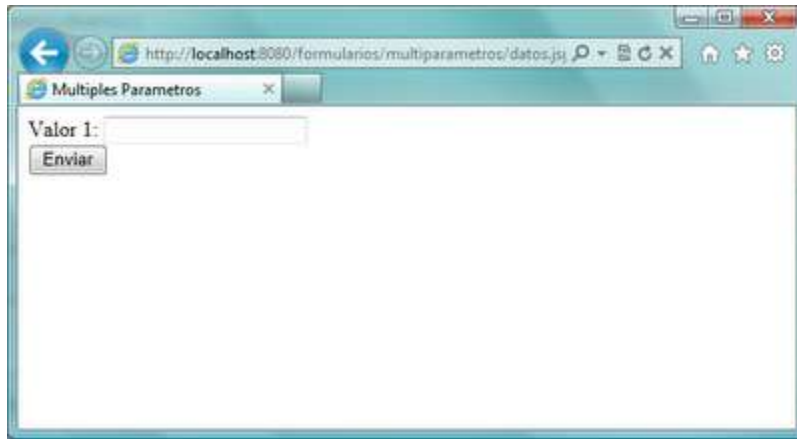
Como estamos haciendo nosotros mismos el acomodo de los componentes, también debemos colocar un salto de línea entre los elementos del formulario; de lo contrario aparecerían todos en la misma línea. Para eso usamos la etiqueta "**
**":

```
<s:form action="multiparametros" theme="simple">
<s:label for="valor1" value="Valor 1: " />
<s:textfield id="valor1" name="valor1" /><br />
</s:form>
```

El siguiente paso consiste en colocar el botón de envío del formulario, usando la etiqueta "**<s:submit>**":

```
<s:form action="multiparametros" theme="simple">
<s:label for="valor1" value="Valor 1: " />
<s:textfield id="valor1" name="valor1" /><br />
<s:submit value="Enviar" />
</s:form>
```

Hasta ahora tenemos un formulario que se ve de la siguiente forma:



Podemos ver que tenemos un solo parámetro, sin embargo para el ejemplo necesitamos varios campos de texto. Haremos que estos se agreguen de forma dinámica usando una biblioteca de JavaScript: **jQuery**.

Colocaremos un botón que al presionarlo agregará un nuevo campo para introducir texto, junto con su etiqueta y salto de línea correspondiente.

El **HTML** del botón es el siguiente:

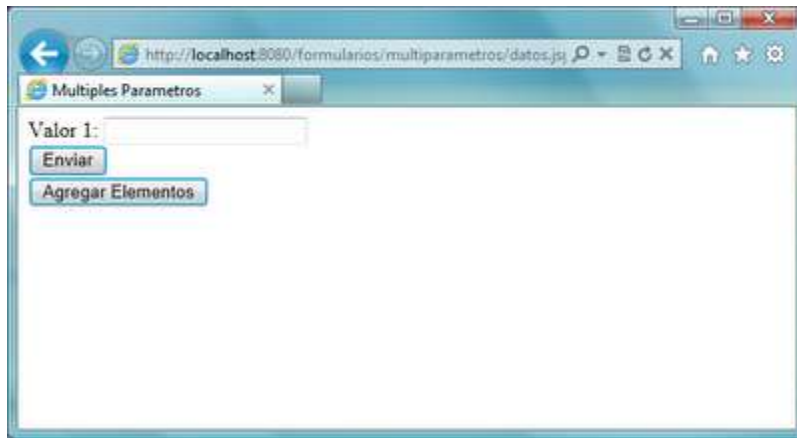
```
<button id="btnAgregar">Agregar Elemento</button>
```

Y el código de **jQuery** para agregar los elementos al formulario es el siguiente:

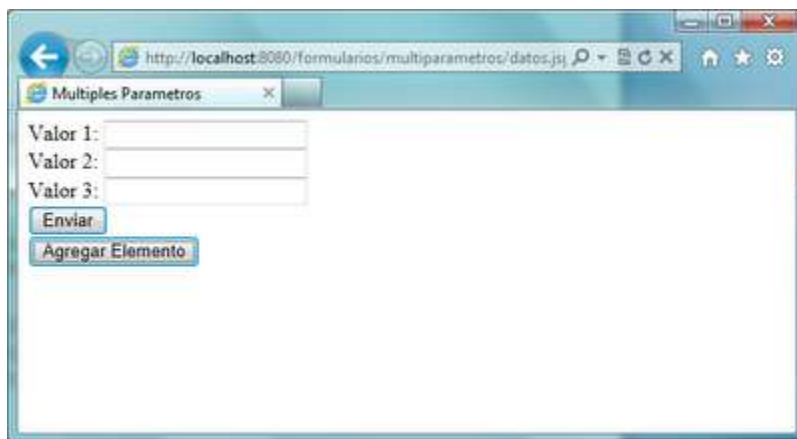
```
$(document).ready(function ()
{
    $("#btnAgregar").click(function()
    {
        var num = $("input[type=text]").length;
        var numeroSiguiente = num + 1;
        var elementoNuevo = $("#valor" + num).clone().attr('id',
        'valor'+numeroSiguiente).attr("name", "valor"+numeroSiguiente);
        var etiquetaNueva = $("label[for=valor"+num+"]").clone().attr("for",
        "valor"+numeroSiguiente).text("Valor " + numeroSiguiente + ": ");

        $("#valor" + num).after(elementoNuevo);
        elementoNuevo.before(etiquetaNueva);
        etiquetaNueva.before("<br />");
    });
});
```

Como este no es un tutorial de **jQuery** no explicaré el código anterior ^_^. Pero el resultado es el siguiente:



Cada vez que presionemos el botón **"Agregar Elemento"** se agregará un nuevo campo de entrada, de la siguiente forma:



Ahora que tenemos un formulario pasemos a crear nuestro **Action**. Creamos una nueva clase **Java**, en el paquete **"Actions"**. El nombre de esta clase será **"MultiplesParametrosAction"**, extenderá de **"ActionSupport"** e implementará la interface **"ParameterAware"**:

```
public class MultiplesParametrosAction extends ActionSupport implements ParameterAware
{
}
```

Debemos colocar una variable que nos permita almacenar la referencia al mapa que los interceptores inyectarán con la lista de parámetros recibidos:

```
private Map<String, String[]> parametros;
```

Además debemos implementar el método **"setParameters"** de la interface **"ParameterAware"**:

```
public void setParameters(Map<String, String[]> parametros)
{
    this.parametros = parametros;
}
```

Adicionalmente pondremos un **getter** para el mapa de parámetros, para poder recuperarlo desde la **JSP** correspondiente:

```
public Map<String, String[]> getParametros()
{
    return parametros;
}
```

Nuestro método "**execute**" nuevamente será muy simple y solo regresara el valor "**SUCCESS**":

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

El siguiente paso es anotar nuestra clase para convertirla en un **Action**. Las anotaciones (y la explicación) serán las mismas que hemos venido usando hasta ahora:

```
@Namespace(value="/multiparametros")


```

La clase "**MultiplesParametrosAction**" completa queda de la siguiente forma:

```
@Namespace(value="/multiparametros")


```

Finalmente debemos crear el contenido de la pagina "**parametros.jsp**". Nuevamente usaremos una etiqueta "**<s:iterator>**". En este caso en vez de pasarle una lista, le pasaremos el mapa con los parámetros que recibimos desde el **Action**, y nuevamente colocaremos una variable en donde se colocará el valor del elemento actual del ciclo:

```
<s:iterator value="parametros" var="parametro">
</s:iterator>
```

Como un **Map** tiene dos elementos importantes, la llave y el valor, debemos obtener cada uno de estos valores para mostrarlos en nuestra página. Para tener acceso a la llave usamos la propiedad "**key**" del elemento actual, y para obtener el valor, usamos la propiedad "**value**" la cual nos regresará el arreglo de cadenas asociado con el parámetro. Para obtener el primer valor de este parámetro debemos accederlo usando la sintaxis de los arreglos, como vimos en el segundo tutorial de la serie:

```
<s:iterator value="parametros" var="parametro">
<s:property value="#parametro.key" />: <s:property value="#parametro.value[0]" />
</s:iterator>
```

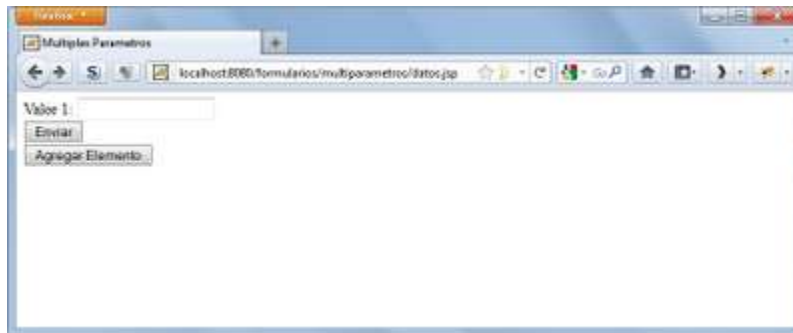
Para que [todo](#) se vea más ordenado, colocaremos los resultados en una lista:

```
<ul>
<s:iterator value="parametros" var="parametro">
<li><s:property value="#parametro.key" />: <s:property value="#parametro.value[0]"
/></li>
</s:iterator>
</ul>
```

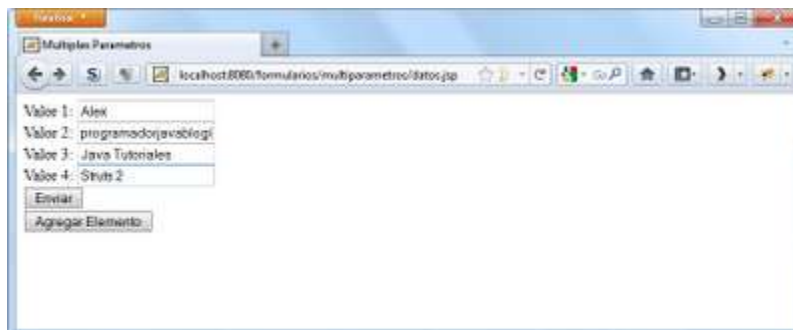
Ahora que ya tenemos el código listo ejecutemos la aplicación; cuando entremos a la dirección:

<http://localhost:8080/formularios/multiparametros/datos.jsp>

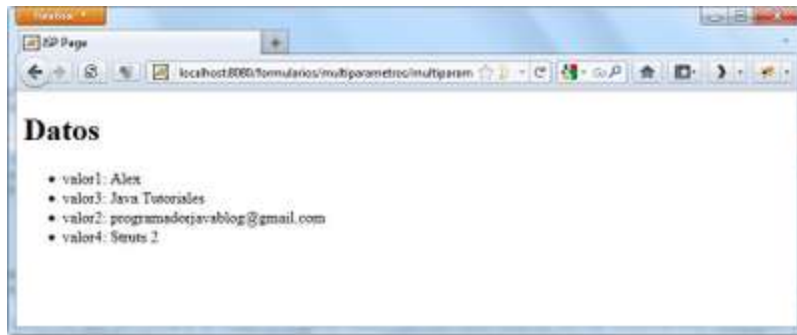
Veremos el formulario que creamos:



Si ingresamos los siguientes datos, y enviamos el formulario:



Veremos la siguiente salida:



Como podemos ver, aunque no tenemos ni idea del nombre o la cantidad de parámetros que recibiremos en la petición, **Struts 2** aún nos proporciona una forma sencilla y elegante de manejarlos.

Hasta ahora hemos visto varias maneras de recibir parámetros desde nuestros formularios. Sin embargo otra parte muy importante del trabajo con estos últimos es el poder realizar algunas validaciones sobre los datos que recibimos. En la siguiente sección veremos cómo usar validaciones en nuestros formularios.

5. Validaciones

Una parte importante del trabajo con formularios es que podamos asegurarnos que la información de los mismos cumpla con ciertos requisitos para poder procesarlos. Por ejemplo que las direcciones de correo electrónico cumplan con el formato establecido, o que ciertos valores estén dentro de un rango válido, o simplemente que se estén proporcionando los campos obligatorios del formulario.

Hacer esto con **Struts 2** es muy simple, ya que proporciona varias maneras de realizar validaciones de forma automática para los datos de nuestros formularios.

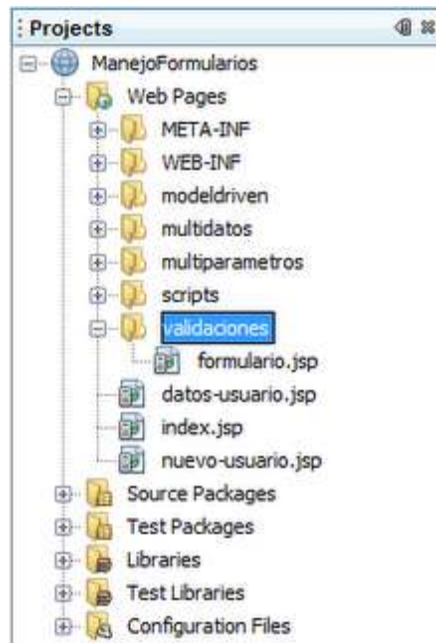
Las validaciones son realizadas por dos interceptores: "**validation**" y "**workflow**". El primero realiza las validaciones y crea una lista de errores específicos para cada uno de los campos que no pase la validación. Posteriormente el interceptor "**workflow**" verifica si hay errores de validación; si encuentra algún error regresa un resultado "**input**", por lo que **es necesario que proporcionemos un result con este nombre**.

Struts 2 tiene básicamente tres formas de realizar validaciones:

- Validaciones mediante un archivo **XML**
- Validaciones mediante anotaciones
- Validaciones manuales

Primero comencemos creando la estructura con los elementos que usaremos en las validaciones:

Lo primero que haremos será crear un nuevo directorio en las páginas web, llamado "**validaciones**". Dentro de este directorio crearemos una nueva **JSP** llamada "**formulario.jsp**":



Dentro de esta página crearemos un formulario sencillo que solicitará al usuario algunos datos básicos como nombre, correo electrónico, teléfono, etc. No olviden colocar en la JSP la directiva taglib correspondiente para indicar que esta hará uso de las bibliotecas de etiquetas de Struts. El formulario quedará, por el momento, de la siguiente forma:

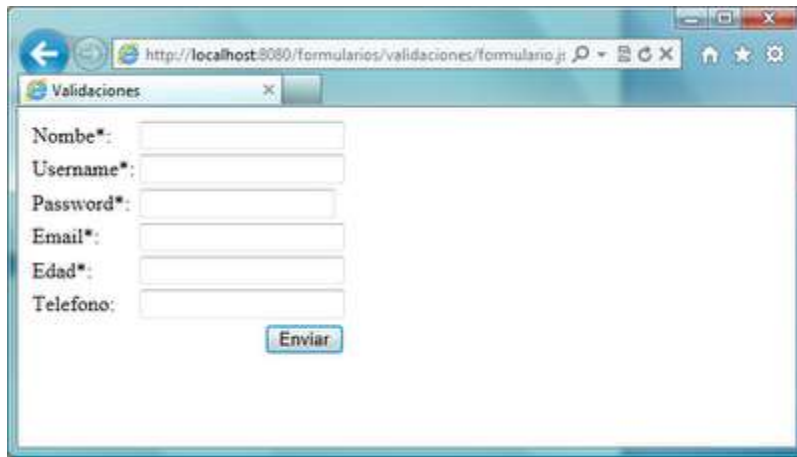
```
<s:form action="validacionDatos">
<s:textfield name="nombre" label="Nombre" />
<s:textfield name="username" label="Username" />
<s:password name="password" label="Password" />
<s:textfield name="email" label="Email" />
<s:textfield name="edad" label="Edad" />
<s:textfield name="telefono" label="Telefono" />
<s:submit value="Enviar" />
</s:form>
```

Otra de las pequeñas ayudas que nos proporcionan las etiquetas de **Struts 2** es que nos permiten indicar si cuáles campos son obligatorios, con lo que se le colocará una pequeña marca en la etiqueta para que el usuario sepa cuáles campos son los que está obligado a llenar. Esta marca es solo una ayuda visual, no realizará ningún proceso de validación del campo.

Para agregar dicha marca e indicar que el campo será obligatorio, debemos colocar el valor de **"true"** en el atributo **"required"** de las etiquetas del formulario. En este caso todos los campos, con excepción del teléfono, serán requeridos:

```
<s:form action="validacionDatos">
<s:textfield name="nombre" label="Nombre" required="true" />
<s:textfield name="username" label="Username" required="true" />
<s:password name="password" label="Password" required="true" />
<s:textfield name="email" label="Email" required="true" />
<s:textfield name="edad" label="Edad" required="true" />
<s:textfield name="telefono" label="Telefono" />
<s:submit value="Enviar" />
</s:form>
```

Con el código anterior obtenemos el siguiente formulario:



Debido al tema ("**theme**") que se usa por default para construir el formulario, los errores serán mostrados directamente sobre los campos que no pasen la validación. Sin embargo, si quisiéramos mostrar, de manera adicional, una lista con todos los errores del formulario, debemos agregar en nuestra **JSP** la etiqueta "**<s:fielderror />**". Solo la colocamos sobre el formulario, de la siguiente forma:

```
<s:fielderror />
```

```
<s:form Action="validacionDatos">
<s:textfield name="nombre" label="Nombre" required="true" />
<s:textfield name="username" label="Username" required="true" />
<s:password name="password" label="Password" required="true" />
<s:textfield name="email" label="Email" required="true" />
<s:textfield name="edad" label="Edad" required="true" />
<s:textfield name="telefono" label="Telefono" />
<s:submit value="Enviar" />
</s:form>
```

Ahora creamos el **Action** que se encargará de procesar los datos anteriores. Para eso creamos una nueva clase llamada "**ValidacionDatos**" en el paquete "**Actions**". Esta clase extenderá de "**ActionSupport**" y tendrá las anotaciones que para este momento ya deben sernos más que familiares:

```
@Namespace(value="/validaciones")
@Action(value="validacionDatos", results={@Result(location="/validaciones/alta-
exitosa.jsp")})
public class ValidacionDatos extends ActionSupport
{
}
```

La clase "**ValidacionDatos**" tendrá **setters** y **getters** para cada uno de los valores que recibiremos del formulario. En este caso los **getters** son importantes por dos razones, la primera es que queremos que el formulario sea repoblado con los valores que el usuario ya había introducido (los cuales solo pueden obtenerse a partir de los **getters**); la segunda es que, por alguna razón que no alcanzo a entender, las validaciones se realizan sobre los **getters** de las propiedades. La clase "**ValidacionDatos**" hasta ahora se ve de la siguiente forma:

```
public class ValidacionDatos extends ActionSupport
{
    private String nombre;
    private String username;
    private String password;
    private String email;
```

```
private int edad;  
private String telefono;
```

```
public void setEdad(int edad)  
{  
    this.edad = edad;  
}
```

```
public void setEmail(String email)  
{  
    this.email = email;  
}
```

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

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

```
public void setTelefono(String telefono)  
{  
    this.telefono = telefono;  
}
```

```
public void setUsername(String username)  
{  
    this.username = username;  
}
```

```
public int getEdad()  
{  
    return edad;  
}
```

```
public String getEmail()  
{  
    return email;  
}
```

```
public String getNombre()  
{  
    return nombre;  
}
```

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

```

    public String getTelefono()
    {
        return telefono;
    }

    public String getUsername()
    {
        return username;
    }
}

```

En su método "**execute**" esta clase solo se encargará de imprimir los valores que se han recibido a través del formulario:

```

@Override
public String execute() throws Exception
{
    System.out.println("nombre: " + nombre);
    System.out.println("username: " + username);
    System.out.println("password: " + password);
    System.out.println("email: " + email);
    System.out.println("edad: " + edad);
    System.out.println("telefono: " + telefono);

    return SUCCESS;
}

```

Ahora crearemos la página que se encargará de mostrar el resultado del alta de usuario. Para esto crearemos una **JSP** llamada "**alta-exitosa.jsp**" en el directorio "**validaciones**". Esta página será muy sencilla y solo mostrará el mensaje "**Usuario dado de alta exitosamente.**"

Finalmente, recordemos que si existe un error de validación el interceptor correspondiente regresará un resultado con valor "**input**". Este resultado debe enviarnos a una página en la que se mostrará la lista de errores de datos de entrada. Normalmente esta página es el mismo formulario de entrada para que el usuario pueda corregir los datos que introdujo de forma incorrecta. Así que modificaremos la anotación "**@Action**" de la clase "**ValidacionDatos**" para agregar un segundo resultado, que en caso de un error de entrada ("**input**") nos regrese al formulario:

```

@Action(value = "validacionDatos", results =
{
    @Result(location = "/validaciones/alta-exitosa.jsp"),
    @Result(name="input", location="/validaciones/formulario.jsp")
})

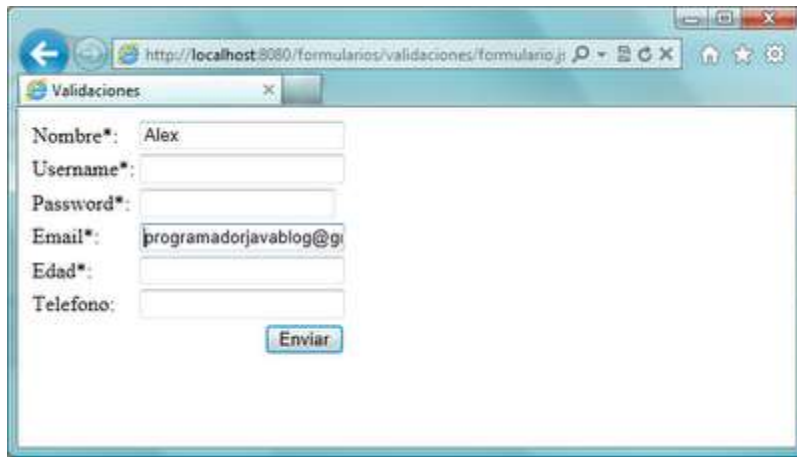
```

Como podemos ver, en este caso si es necesario indicar de forma explícita el nombre del result usando el atributo "**name**".

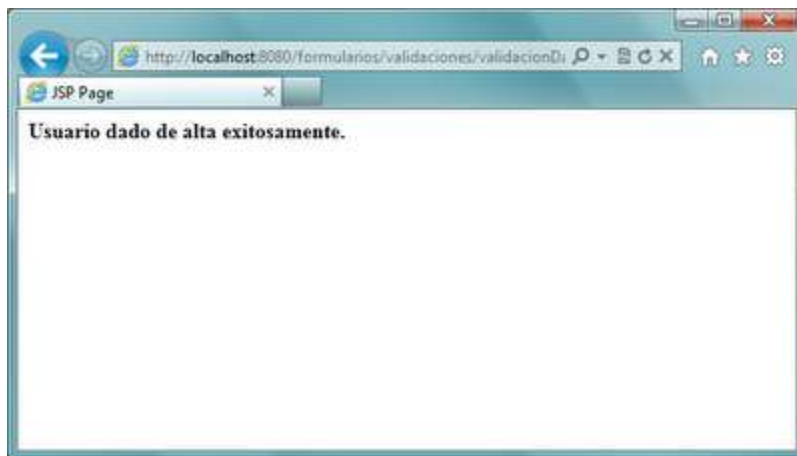
Al ejecutar nuestra aplicación y entrar en la siguiente dirección:

<http://localhost:8080/formularios/validaciones/formulario.jsp>

Deberemos ver el formulario que creamos hace un momento. Llenemos un par de campos para ver el resultado obtenido:



Al enviar los datos del formulario deberemos ver la siguiente salida:



Si analizamos la salida anterior veremos que esta no es correcta, ya que el usuario no ha proporcionado todos los datos requeridos, por lo que debería ver un mensaje de error. Solucionaremos esto agregando validaciones en los datos. Primero veremos cómo hacer validaciones usando archivos **XML**.

5.1 Validaciones usando archivos XML

Para realizar validaciones usando archivos en **XML** debemos crear un archivo por cada una de las clases que será validada. El nombre de este archivo debe seguir una convención muy sencilla:

<NombreDeNuestroAction>-validation.xml

O sea, que debe tener el mismo nombre del **Action** + la cadena **"-validation"** y la extensión **".xml"**. El archivo debe estar colocado en el mismo directorio que la clase **Action** que validará.

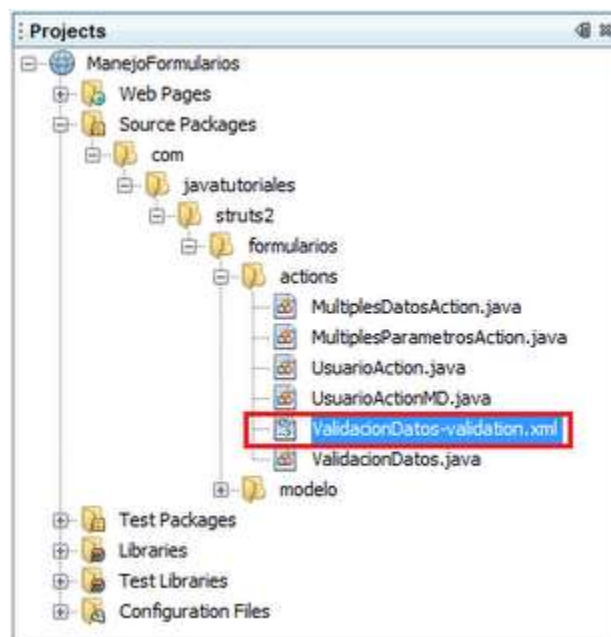
Dentro de este archivo se indicará cuáles campos serán validados y qué validaciones se le aplicarán. Cada una de estas validaciones tiene un parámetro obligatorio, que es el nombre del campo que validará. Adicionalmente pueden o no tener más parámetros.

Struts 2 ya viene con algunas validaciones predeterminadas que podemos usar, aunque también tenemos la posibilidad de crear nuestras propias validaciones en caso de ser necesario. Las validaciones incluidas con **Struts 2** se muestran en la siguiente tabla:

| Validador | Descripción |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| required | Verifica que el campo especificado no sea nulo. |
| requiredstring | Verifica que un campo de tipo String no sea nulo y que tenga una longitud mayor a 0 . |
| stringlength | Verifica que una cadena tenga una cierta longitud. |
| int, long, y short | Verifica que el int , long , o short especificado estén dentro de un rango determinado. |
| double | Verifica que el valor double especificado esté dentro de cierto rango. |
| date | Verifica que la fecha proporcionada esté dentro del rango proporcionado. Por default se usa el formato Date.SHORT para indicar las fechas. |
| email | Verifica que el campo cumpla con el formato de una dirección de email válida |
| url | Verifica que el campo cumpla con el formato de una URL válida. |
| visitor | Permite enviar la validación a las propiedades del objeto del Action usando los propios de archivos de validación del objeto. Esto permite usar ModelDriven y administrar las validaciones de los objetos de modelo en un solo lugar. |
| conversion | Verifica si ocurre un error de conversión en el campo. |
| expression | Realiza una validación basada en una expresión regular en OGNL . |
| fieldexpression | Valida un campo usando una expresión OGNL . |
| regex | Valida un campo usando una expresión regular. |

Sabiendo qué validadores podemos aplicar a los campos del formulario, pasemos a crear nuestro archivo de validación.

Creemos en el paquete "**actions**" un nuevo archivo **XML** llamado "**ValidacionDatos-validation.xml**":



Todas las validaciones deben estar contenidas dentro del elemento "<validators>":

```
<?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>

</validators>
```

Lo siguiente es indicar cada campo que será validado usando el elemento "<field>", e indicamos el nombre del campo que queremos validar en su atributo "name":

```
<field name="nombre">
</field>
```

Posteriormente indicamos cada una de las validaciones que se aplicarán a ese campo, usando el elemento "<field-validator>" y su atributo "type":

```
<field name="nombre">
<field-validator type="requiredstring">

</field-validator>
</field>
```

Para agregar un parámetro a la validación usamos el elemento "<param>" y colocamos el nombre del parámetro en su atributo "name". En este caso usaremos el parámetro "trim" del validator "requiredstring" para indicar que deseamos que se eliminen los espacios en blanco del inicio y fin de la cadena (en caso de existir) antes de que la validación sea aplicada:

```
<field-validator type="requiredstring">
<param name="trim">true</param>
</field-validator>
```

Para terminar, debemos agregar un mensaje que será mostrado en caso de que la validación no sea exitosa. Para esto usamos el elemento "<message>":

```
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>El nombre del usuario es un campo obligatorio.</message>
</field-validator>
```

Y esto es todo, con esto el campo "nombre" del formulario ha quedado validado.

Si necesitáramos agregar más validaciones a este campo, solo bastaría con agregar un segundo elemento "<field-validator>". Agregamos un validator de tipo "stringlength" para asegurarnos que el nombre que introduzca el usuario tenga una longitud mínima de 4 caracteres, y una longitud máxima de 12. El proceso es básicamente el mismo por lo que no lo explicaré, pero al final la validación queda de la siguiente forma:

```
<field-validator type="stringlength">
<param name="trim">true</param>
<param name="minLength">4</param>
<param name="maxLength">12</param>
<message>El nombre del usuario debe tener entre 4 y 12 caracteres</message>
</field-validator>
```

Hasta ahora nuestro archivo de validaciones se ve de la siguiente forma:

```

<?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="nombre">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>El nombre del usuario es un campo obligatorio.</message>
</field-validator>

<field-validator type="stringlength">
<param name="trim">true</param>
<param name="minLength">4</param>
<param name="maxLength">12</param>
<message>El nombre del usuario debe tener entre 4 y 12 caracteres</message>
</field-validator>
</field>
</validators>

```

El proceso es básicamente el mismo para todos los campos del formulario que vamos a validar. No explicaré cada una de las validaciones de los campos porque creo que la idea se entiende bastante en este momento, pero el archivo final de validaciones queda de la siguiente forma:

```

<?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="nombre">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>El nombre del usuario es un campo obligatorio.</message>
</field-validator>

<field-validator type="stringlength">
<param name="trim">true</param>
<param name="minLength">4</param>
<param name="maxLength">12</param>
<message>El nombre del usuario debe tener entre 4 y 12 caracteres</message>
</field-validator>
</field>

<field name="username">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>El username es un campo obligatorio.</message>
</field-validator>

<field-validator type="stringlength">
<param name="trim">true</param>
<param name="minLength">6</param>
<message>El username debe tener al menos 6 caracteres</message>
</field-validator>
</field>

```

```

<field name="password">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>La contraseña es un campo obligatorio.</message>
</field-validator>

<field-validator type="stringlength">
<param name="trim">true</param>
<param name="minLength">6</param>
<param name="maxLength">8</param>
<message>La contraseña debe tener entre 6 y 8 caracteres</message>
</field-validator>

<field-validator type="regex">
<param name="expression">^\w*(?=\w*\d) (?=\w*[a-z]) (?=\w*[A-Z]) \w*$</param>
<message>La contraseña debe ser alfanumérica, debe tener al menos una letra mayúscula,
una letra minúscula, y al menos un número.</message>
</field-validator>
</field>

<field name="email">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message>El correo electrónico es un campo obligatorio.</message>
</field-validator>

<field-validator type="email">
<message>El correo electrónico está en un formato inválido.</message>
</field-validator>
</field>

<field name="edad">
<field-validator type="required">
<message>La edad es un campo obligatorio.</message>
</field-validator>

<field-validator type="conversion">
<message>La edad debe contener solo números enteros.</message>
</field-validator>

<field-validator type="int">
<param name="min">0</param>
<param name="max">200</param>
<message>La edad proporcionada no está dentro del rango permitido.</message>
</field-validator>
</field>

</validators>

```

Ahora que todo está listo podemos ejecutar nuestra aplicación y entrar a la siguiente dirección:

<http://localhost:8080/formularios/validaciones/formulario.jsp>

Nuevamente veremos nuestro formulario anterior. Si no colocamos ningún valor en ningún campo y simplemente presionamos el botón **"Enviar"** veremos la siguiente lista de errores:

A screenshot of a web browser window displaying a form titled "Validaciones". The form contains several input fields: "Nombre*", "Username*", "Password*", "Email*", "Edad*", and "Telefono:". Above the form, a list of error messages is displayed, each enclosed in a red box. The errors are: "El nombre del usuario es un campo obligatorio.", "El username es un campo obligatorio.", "La contraseña es un campo obligatorio.", "El correo electrónico es un campo obligatorio.", and "La edad debe contener solo números enteros." Below the form, there is an "Enviar" button.

Podemos ver que estamos obteniendo los errores que definimos en el archivo de validaciones, con los mensajes correspondientes. Si observamos el código fuente de la pagina generada veremos que cada uno de los errores marcados está colocado en un elemento que tiene como clase CSS **"errorMessage"**, por lo que podemos personalizar la forma en la que se muestran los errores de validación de la aplicación a través de una hoja de estilos.

Si ahora colocamos algunos valores incorrectos en los campos del formulario:

A screenshot of a web browser window displaying the same form titled "Validaciones". The form fields are now filled with the following values: "Nombre*" is "Alex", "Username*" is "programadorjava", "Password*" is "*****", "Email*" is "programador", "Edad*" is "-10", and "Telefono:" is empty. The "Enviar" button is visible at the bottom right.

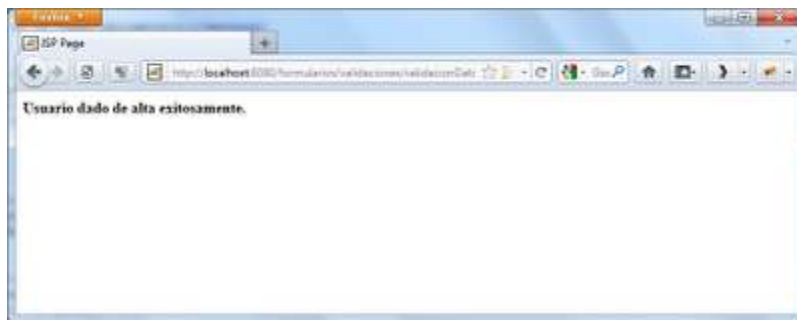
Obtendremos una lista distinta de errores:

A screenshot of a web browser window displaying the form titled "Validaciones" with the same values as the previous screenshot. The error messages are now different, reflecting the incorrect data entered. The errors are: "El nombre del usuario debe tener entre 4 y 12 caracteres", "El username debe tener al menos 6 caracteres", "La contraseña debe ser alfanumérica, debe tener al menos una letra y al menos un número.", "El correo electrónico está en un formato inválido.", and "La edad proporcionada no está dentro del rango permitido." Each error message is enclosed in a red box. The "Enviar" button is at the bottom right.

Si ahora colocamos datos correctos:



Veremos la página correspondiente:



Por lo que los datos han sido validados correctamente.

A pesar de que la mayoría de las veces realizaremos validaciones como las anteriores, o sea validaciones sobre los campos de un formulario, algunas veces será necesario realizar validaciones que no son sobre los campos (llamadas **validaciones planes**).

Las validaciones planas son un tipo especial de validación que no está atada a un campo específico, como la validación de expresiones. Por ejemplo, agreguemos una validación extra que verifique que el "**nombre**" y el "**username**" no sean iguales.

Para agregar una validación plana en vez de usar el elemento "**<field>**" usamos el elemento "**<validator>**", que es muy similar a "**<field-validator>**", indicamos usando su atributo "**type**" el tipo de validación que se realizará; si la validación necesita algún parámetro lo agregamos con el elemento "**<param>**"; el mensaje que se mostrará en caso de que la validación no pase lo colocamos con el elemento "**<message>**".

La validación para verificar que el nombre del **usuario** y el **username** no sean iguales queda de la siguiente forma (pueden colocarla después de todas las validaciones "**field**"):

```
<validator type="expression">
<param name="expression">!nombre.equals(username)</param>
<message>El nombre de usuario y el username no deben ser iguales.</message>
</validator>
```

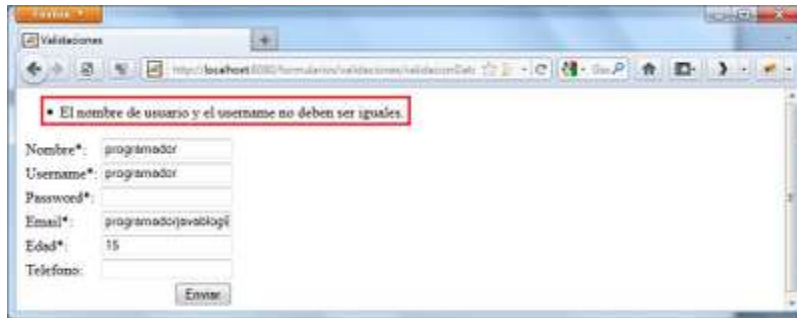
Como estas validaciones no son realizadas sobre ningún campo, sus mensajes de error no se mostrarán usando la etiqueta "**<s:fielderror />**", si queremos verlos en la **JSP** debemos usar la etiqueta "**<s:actionerror />**":

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

Ahora, si volvemos a ejecutar la aplicación e introducimos el mismo nombre de usuario y contraseña:



Y presionamos el botón enviar obtendremos el siguiente resultado:



Con lo que podemos comprobar que estas validaciones también funcionan correctamente.

Ahora que vimos cómo realizar validaciones usando archivos **XML** veamos cómo realizar esto mismo usando anotaciones.

5.2 Validaciones usando anotaciones

Realizar validaciones de los datos de un formulario usando anotaciones es un proceso muy simple. Modificaremos un poco la clase "**ValidacionDatos**" que creamos hace un momento (de hecho yo crearé una clase aparte para no modificar el código que ya teníamos).

Las anotaciones que podemos usar para realizar las validaciones se encuentran en el paquete "**com.opensymphony.xwork2.validator.annotations**", y tenemos las siguientes validaciones disponibles:

- **@ConversionErrorFieldValidator**
- **@DateRangeFieldValidator**
- **@DoubleRangeFieldValidator**
- **@EmailValidator**
- **@ExpressionValidator**
- **@FieldExpressionValidator**
- **@IntRangeFieldValidator**
- **@RegexFieldValidator**
- **@RequiredFieldValidator**
- **@RequiredStringValidator**
- **@StringLengthFieldValidator**
- **@UrlValidator**
- **@VisitorFieldValidator**

Como podemos ver, existe una anotación por cada una de las validaciones que se tienen cuando trabajamos con los archivos **XML**; la explicación para cada una de las validaciones también es la misma que para el caso anterior.

Esta anotación puede ser colocada tanto en el **setter** como en el **getter** de la propiedad que queremos validar. También podemos usar varias validaciones para una misma propiedad.

Veamos un primer ejemplo validando el campo "**nombre**". En la clase "**ValidacionDatos**" primero verificaremos que el nombre del usuario no esté vacío cuando se reciba del formulario, para esto usaremos la anotación "**@RequiredStringValidator**", yo la pondré en el **setter** de la propiedad:

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

Todas las anotaciones de validaciones proporcionan una serie de atributos que nos permiten configurar el comportamiento de la validación. Casi en todos los casos lo único que tendremos que configurar es el mensaje que se mostrará en caso de que la validación no sea correcta. Para eso usamos el atributo "**message**":

```
@RequiredStringValidator(message = "El nombre de usuario es un campo obligatorio.")
public void setNombre(String nombre)
{
    this.nombre = nombre;
}
```

También validaremos que el nombre tenga una cierta longitud, o sea que sea mayor a cierto número de caracteres y menor a otro cierto número. Para eso usamos la anotación la anotación "**@StringLengthFieldValidator**" que colocamos justo debajo de la anotación anterior:

```
@RequiredStringValidator(message = "El nombre de usuario es un campo obligatorio.")
@StringLengthFieldValidator(minLength="4", maxLength="12", message="El nombre del usuario
debe tener entre 4 y 12 caracteres")
public void setNombre(String nombre)
{
    this.nombre = nombre;
}
```

Haremos lo mismo para cada uno de los campos que necesitemos validar. Nuevamente, no explicaré cada una de las anotaciones ya que son bastante intuitivas, pero al final quedan de la siguiente forma:

```
@RequiredFieldValidator(message="La edad es un campo obligatorio.")
@ConversionErrorFieldValidator(message="La edad debe contener solo números enteros.")
@IntRangeFieldValidator(min="0", max="200", message="La edad proporcionada no está dentro
del rango permitido.")
public void setEdad(int edad)
{
    this.edad = edad;
}
```

```
@RequiredStringValidator(message="El correo electrónico es un campo obligatorio.")
@EmailValidator(message="El correo electrónico está en un formato inválido.")
public void setEmail(String email)
{
    this.email = email;
}
```

```

@RequiredStringValidator(message = "El nombre de usuario es un campo obligatorio.")
@StringLengthFieldValidator(minLength = "4", maxLength = "12", message = "El nombre del
usuario debe tener entre 4 y 12 caracteres")
public void setNombre(String nombre)
{
    this.nombre = nombre;
}

@RequiredStringValidator(message="La contraseña es un campo obligatorio.")
@StringLengthFieldValidator(minLength="6", maxLength="8", message="La contraseña debe
tener entre 6 y 8 caracteres")
@RegexFieldValidator(expression="^\\w*(?=\\w*\\d) (?!\\w*[a-z]) (?!\\w*[A-Z])\\w*$",
message="La contraseña debe ser alfanumérica, debe tener al menos una letra mayúscula,
una letra minúscula, y al menos un número.")
public void setPassword(String password)
{
    this.password = password;
}

public void setTelefono(String telefono)
{
    this.telefono = telefono;
}

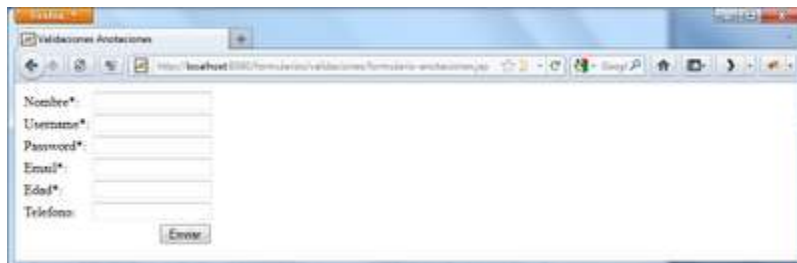
@RequiredStringValidator(message="El username es un campo obligatorio.")
@StringLengthFieldValidator(minLength="6", message="El username debe tener al menos 6
caracteres")
public void setUsername(String username)
{
    this.username = username;
}

```

Probemos nuevamente que todo funcione correctamente. Al ejecutar la aplicación y entrar en nuestro formulario, en la siguiente dirección:

<http://localhost:8080/formularios/validaciones/formulario.jsp>

(Bueno, de hecho mi dirección es un poco distinto porque cree el contenido en una página alterna ^^) Debemos ver el formulario que ya teníamos anteriormente:



The screenshot shows a web browser window with the address bar displaying 'http://localhost:8080/formularios/validaciones/formulario.jsp'. The page title is 'Validaciones Anotaciones'. The form contains the following fields: 'Nombre*' (text), 'Username*' (text), 'Password*' (password), 'Email*' (text), 'Edad*' (text), and 'Telefono' (text). An 'Enviar' button is located at the bottom right of the form.

Al hacer clic en el botón **"Enviar"**, con los campos vacíos, obtendremos la siguiente salida:

Validaciones Anotaciones

- La contraseña es un campo obligatorio.
- El username es un campo obligatorio.
- El nombre de usuario es un campo obligatorio.
- La edad debe contener solo números enteros.
- El correo electrónico es un campo obligatorio.

El nombre de usuario es un campo obligatorio

Nombre*

El username es un campo obligatorio

Username*

La contraseña es un campo obligatorio

Password*

El correo electrónico es un campo obligatorio

Email*

La edad debe contener solo números enteros

Edad*

Telefono*

Enviar

Podemos ver que, hasta el momento, las validaciones han funcionado correctamente. Si colocamos algunos datos de forma equivocada:

Validaciones Anotaciones

Nombre*: Alex

Username*: programadorjava

Password*: *****

Email*: programador

Edad*: -10

Telefono*

Enviar

Obtendremos la siguiente salida:

Validaciones Anotaciones

- La contraseña debe ser alfanumérica, debe tener al menos una letra mayúscula, una letra minúscula, y al menos un número.
- La edad proporcionada no está dentro del rango permitido.
- El correo electrónico está en un formato inválido.

Nombre*: Alex

Username*: programadorjava

La contraseña debe ser alfanumérica, debe tener al menos una letra mayúscula, una letra minúscula, y al menos un número.

Password*

El correo electrónico está en un formato inválido.

Email*: programador

La edad proporcionada no está dentro del rango permitido

Edad*: -10

Telefono*

Enviar

Si ahora, colocamos los datos de forma correcta:

Validaciones Anotaciones

Nombre*: Alex

Username*: programadorjava

Password*: *****

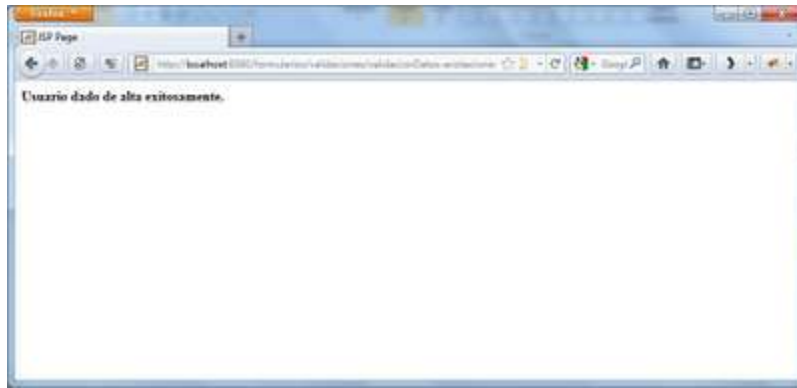
Email*: programador@gmail.com

Edad*: 15

Telefono*

Enviar

Obtendremos la salida que estamos esperando:



Como podrán imaginar, todas las validaciones anteriores son validaciones de campos. Usando anotaciones también podemos realizar validaciones planas (las validaciones que no están relacionadas con algún campo del formulario). Solo que en este caso las validaciones planas se colocan directamente en el método **"execute"** del **Action**, usando una anotación especial: **"@Validations"**. En los atributos de esta anotación podemos indicar cada una de las validaciones que se necesiten en la aplicación (campos requeridos, emails, urls, longitud de las cadenas, rango de enteros, fechas, etc.). En nuestro caso lo que nos interesa es realizar una validación sobre una expresión. Para esto usamos el atributo **"expressions"** de la anotación **"@Validations"**:

```
@Override
@Validations(expressions={})
public String execute() throws Exception
```

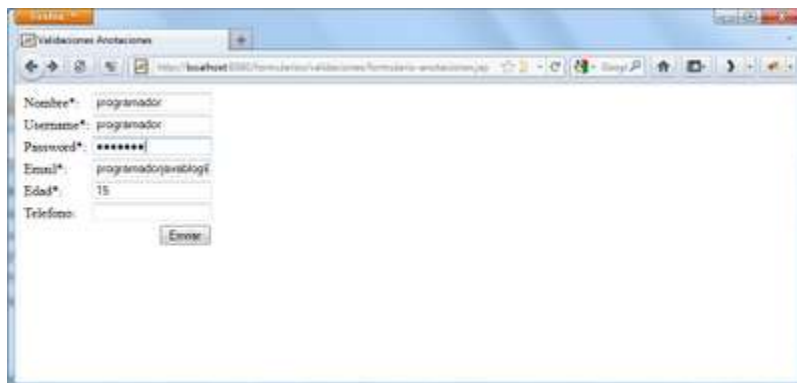
En este atributo colocamos, usando la anotación **"@ExpressionValidator"**, la expresión que será validada, en este caso queremos asegurarnos de que el nombre del usuario y su username no son iguales, por lo que la validación queda de la siguiente forma:

```
@ExpressionValidator(expression="!nombre.equals(username)", message="El nombre de usuario
y el username no deben ser iguales.")
```

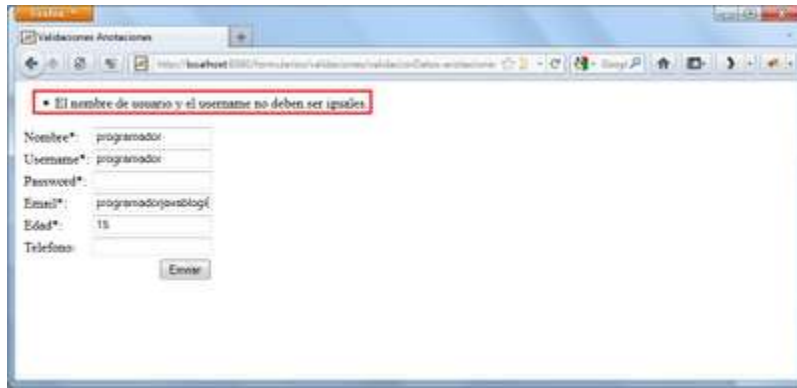
La anotación completa queda de la siguiente forma:

```
@Override
@Validations(expressions =
{
    @ExpressionValidator(expression = "!nombre.equals(username)", message = "El nombre de
usuario y el username no deben ser iguales.")
})
public String execute() throws Exception
```

Cuando ejecutemos nuevamente la aplicación, poniendo el mismo nombre de usuario y contraseña:



Obtendremos el mensaje correspondiente de error:



Cuando coloquemos los datos correctos deberíamos dejar de ver mensajes de error.

Los dos tipos de validaciones que hemos visto nos permiten verificar que los datos que proporcionemos sean correctos, siempre y cuando conozcamos de antemano los valores, los rangos, o las reglas que los campos del formulario pueden tener pero ¿qué pasa si no conocemos estos valores? por ejemplo, en el caso de que debamos comparar un valor contra un registro de la base de datos o contra algún web service o alguna cosa más sofisticada, las validaciones anteriores nos servirán de poco.

Afortunadamente, para esos casos, **Struts 2** nos permite realizar un tercer tipo de validación: las validaciones manuales.

5.3 Validaciones Manuales

Ya sé qué es lo que deben estar pensando: que las validaciones manuales son las que estamos acostumbrados a realizar, que no es un mecanismo automático que nos ayude a simplificar nuestro trabajo de validación de datos. Si es así, no están tan equivocados ^_^; bueno, más o menos.

Las validaciones manuales son aquellas que son tan particulares a nuestro proceso de negocio que estas no se pueden tomar de alguna plantilla, como los ejemplos que mencioné anteriormente.

Este tipo de validación, aunque las tenemos que realizar nosotros mismos, sigue siendo administrado por el framework. Para ello nos proporciona una interface llamada "**Validateable**", que proporciona únicamente un método:

```
public void validate();
```

Afortunadamente para nosotros, la clase "**ActionSupport**" implementa esta interface, por lo que lo unico que tenemos que hacer es sobre-escribirlo en nuestros **Actions** y, ayudados de algunos métodos auxiliares de la clase "**ActionSupport**", enviar los errores correspondientes en caso de que estos existan.

Para poder indicar si existe algún error en las validaciones nos ayudamos del método "**addFieldError**", en caso de que exista un error de la validación de un campo, y de "**addActionError**", en caso de existir un error que no esté relacionado con un campo.

"**addFieldError**" recibe dos parámetros, el primero es el nombre del campo que generó el error y el segundo es la descripción del error. "**addActionError**" recibe un solo parámetro que es la descripción del error.

Como se podrán imaginar, si existe algún error en un campo este se mostrará en el campo correspondiente del formulario y en la etiqueta "<s:fielderror />". Si existe un error de otro tipo este se mostrará en la etiqueta "<s:actionerror />".

Ahora que conocemos la teoría sobre las validaciones manuales pasemos verlas en la práctica. Para esto modificaremos las validaciones que ya tenemos en nuestra clase.

Lo primero que debemos hacer es sobre-escribir el método **"validate"**:

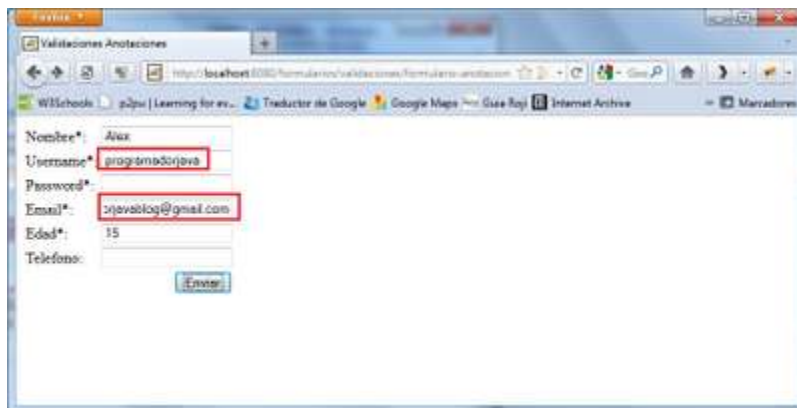
```
@Override
public void validate()
{
}
```

Las validaciones que realizaremos serán muy simples: si el username es **"programadorjava"** (supongamos que este fue un nombre que se trajo de la base de datos, para comprobar que no haya nombres repetidos) o el correo es **"programadorjavablog@gmail.com"** entonces no se le permitirá al usuario completar el registro:

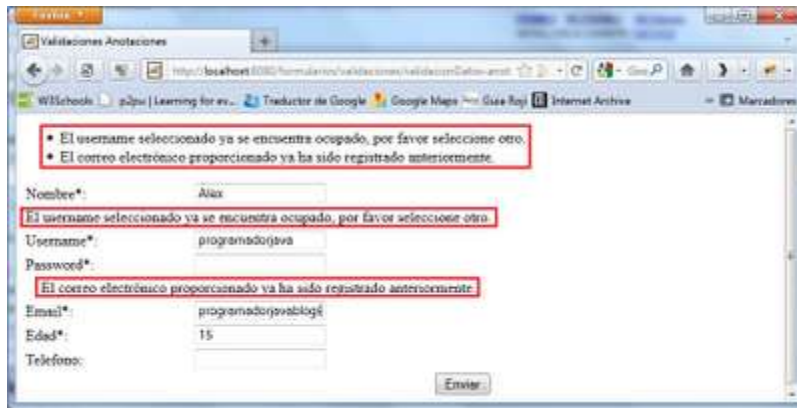
```
@Override
public void validate()
{
    if ("programadorjava".equals(username))
    {
        addFieldError("username", "El username seleccionado ya se encuentra ocupado, por favor seleccione otro.");
    }
    if ("programadorjavablog@gmail.com".equals(email))
    {
        addFieldError("email", "El correo electrónico proporcionado ya ha sido registrado anteriormente.");
    }
}
```

El framework detectará de forma automática si hemos agregado algo a los campos de error, si es así regresará esta lista de errores a nuestro resultado **"input"**, de lo contrario continuará con la ejecución del método **"execute"**.

Si ejecutamos la aplicación y entramos a nuestro formulario, proporcionando alguno de los datos reservados y que son validados dentro del método **"validate"**:

A screenshot of a web browser window titled "Validaciones Anotaciones". The browser's address bar shows a local host URL. The page displays a registration form with the following fields: "Nombre*" (filled with "Alex"), "Username*" (filled with "programadorjava" and highlighted with a red border), "Password*" (empty), "Email*" (filled with "programadorjavablog@gmail.com" and highlighted with a red border), "Edad*" (filled with "15"), and "Telefono*" (empty). At the bottom of the form is a blue "Enviar" button. The browser's address bar contains several bookmarks, including "W3Schools", "p2pu | Learning for ev...", "Traductor de Google", "Google Maps", "Gua Rapi", and "Internet Archive".

Veremos los mensajes de error que hemos establecido:



Podemos comprobar nuevamente que las validaciones han funcionado correctamente ^_^.

Como vemos, realizar validaciones usando **Struts 2** es muy sencillo y puede ahorrarnos bastante tiempo que podemos dedicar a mejorar otros aspectos de nuestra aplicación.

Ahora veremos cómo realizar otro de los trabajos que normalmente hacemos al hacer uso de formularios, la carga de archivos desde el cliente al servidor:

6. Carga de archivos

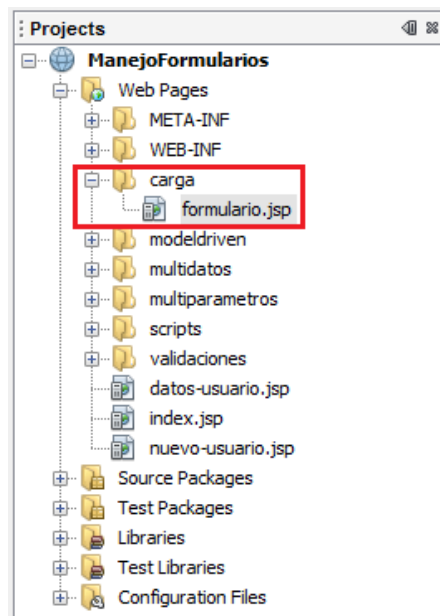
Hacer carga o upload de archivos desde un cliente hacia nuestro servidor, ya sea para almacenarlos o para procesarlos, es una operación que no siempre es sencilla de realizar. Sin embargo **Struts 2** cuenta con una manera que hace que lograr esto sea tan sencillo como el resto de las cosas que hasta ahora hemos aprendido.

Struts 2 proporciona el soporte para la carga de archivos conforme a la especificación de **HTML**, esto nos permite subir uno o varios archivos desde el cliente al servidor.

Cuando un archivo es cargado, este será almacenado en un directorio temporal por el interceptor correspondiente (**fileUpload**). El archivo deberá entonces ser procesado o movido a otra ubicación, por nuestro **Action**, ya que al terminar la petición el interceptor se encargará de eliminar este archivo temporal.

Veamos cómo realizar la carga de archivos con un ejemplo.

Lo primero que haremos será crear un nuevo directorio en las páginas web, llamado "**carga**". Dentro de este directorio crearemos una nueva **JSP** llamada "**formulario.jsp**":



Como hasta ahora, debemos indicar en la **JSP** que haremos uso de la biblioteca de etiquetas de **Struts 2**:

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

Lo siguiente es crear un formulario. Este será un poco distinto a los que hemos creado hasta ahora. Lo primero es que los datos del formulario deben codificarse de una forma especial antes de que estos sean enviados al servidor. Afortunadamente es el navegador el que se encarga de hacer esta codificación, lo único que nosotros debemos hacer es indicar, usando el atributo "**enctype**" del formulario, cuál codificación será; cuando cargamos archivos estos deben ir codificados en "**multipart/form-data**".

Además los datos deben ser enviados por **POST** (en los formularios **HTML** el método por default para enviar datos es **GET** (por la **URL**), pero en **Struts 2** por default se envían por **POST** (en el cuerpo del mensaje de la petición), así que no debemos agregarle ninguna cosa extra en este caso). El formulario, hasta ahora, se ve de la siguiente forma:

```
<s:form enctype="multipart/form-data">
</s:form>
```

Los archivos son cargados con un tipo de campo especial el cual es generado usando la etiqueta "**<s:file />**". Esta etiqueta funciona de la misma forma que las demás que hemos venido usando hasta el momento:

```
<s:form action="cargaArchivo" enctype="multipart/form-data">
<s:file name="archivo" label="Archivo" />
</s:form>
```

En este caso he llamado a mi archivo "**archivo**" pero podría tener cualquier nombre, como "**imagen**", "**reporte**", "**datos**", etc.

Agregaré otro campo solo para que vemos que podemos subir archivos junto con datos "**planos**" del formulario:

```
<s:form action="cargaArchivo" enctype="multipart/form-data">
<s:file name="archivo" label="Archivo" />
<s:textfield name="autor" label="Autor" />
</s:form>
```

Terminaremos el formulario agregando un botón de envío de formulario y una etiqueta que nos mostrará cualquier error que pudiera ocurrir en el proceso de carga:

```
<s:actionerror />
<s:form action="cargaArchivo" enctype="multipart/form-data">
<s:file name="archivo" label="Archivo" />
<s:textfield name="autor" label="Autor" />
<s:submit value="Enviar" />
</s:form>
```

Como ven, el formulario es tan sencillo como todos los que hemos estado haciendo a lo largo del tutorial.

Ahora crearemos el **Action** que se encargará de procesar los datos de este formulario, y que es donde se encuentra la parte interesante del ejemplo ^^.

Creamos una nueva clase en el paquete "**actions**" llamada "**CargaArchivo**". Esta clase extenderá de "**ActionSupport**":

```
public class CargaArchivo extends ActionSupport
{
}
```

Agregaremos una propiedad de tipo **String** para almacenar el nombre del autor:


```
public class CargaArchivo extends ActionSupport
{
    private String autor;
}
```

Ahora viene la parte importante, **para poder obtener una referencia al archivo, debemos colocar, en nuestro Action, una propiedad de tipo "File"**:

```
public class CargaArchivo extends ActionSupport
{
    private String autor;
    private File archivo;
}
```

Con esto, cada vez que se cargue un archivo usando el formulario, la variable de tipo "**File**" que tenemos, hará referencia a este archivo. Lo que nos queda es agregar los respectivos **setters** para las dos propiedades anteriores:

```
public class CargaArchivo extends ActionSupport
{
    private String autor;
    private File archivo;

    public void setArchivo(File archivo)
    {
        this.archivo = archivo;
    }

    public void setAutor(String autor)
    {
        this.autor = autor;
    }
}
```

Ahora sobrecargaremos el método "**execute**" de nuestro **Action** para mover el archivo a una nueva ubicación en nuestro sistema de archivos:

```
@Override
public String execute() throws Exception
{
    File nuevoArchivo = new File("/", archivo.getName());
    archivo.renameTo(nuevoArchivo);

    return SUCCESS;
}
```

Podemos ver que colocaremos el nuevo archivo en el directorio raíz del sistema operativo y que el nombre del archivo será el mismo que el del archivo que estamos subiendo.

Al terminar el proceso mostraremos un poco de información con respecto al archivo. La información del archivo se verá en la página que enviemos como resultado "**SUCCESS**", por lo que deberemos proporcionar los **getters** correspondientes para esta información; en este caso mostraremos el nombre del archivo y la ruta en la que queda almacenado:

```

public String getNombre()
{
    return archivo.getName();
}

public String getRuta()
{
    return archivo.getAbsolutePath();
}

```

Como ven estamos obteniendo la información directamente del objeto **File** que representa al archivo que estamos cargando.

Crearemos la página que mostrará los resultados por lo que en el directorio "**carga**" creamos una nueva **JSP** llamada "**archivoCargado**". Dentro de este archivo hacemos indicamos usando la directiva "**taglib**" que haremos uso de la biblioteca de etiquetas de **Struts 2**. Después, usando la etiqueta "<s:property>" mostraremos los dos valores de los atributos anteriores:

```

Nombre: <s:property value="nombre" /><br />
Ruta: <s:property value="ruta" />

```

Anotaremos este clase como ya sabemos hacerlo, para que el framework la trate como un **Action**:

```

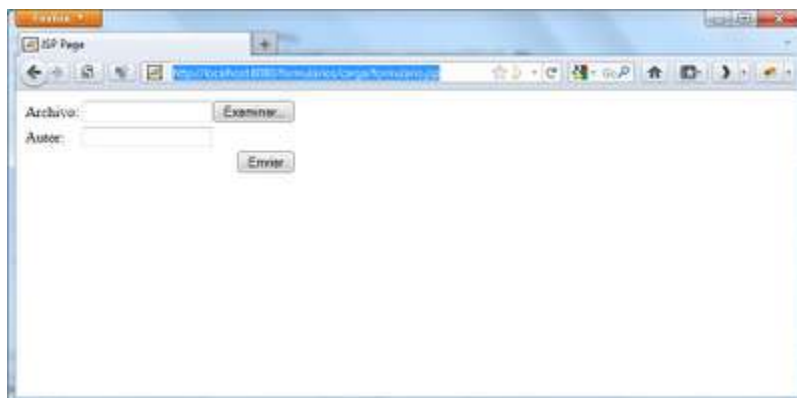
@Namespace(value = "/carga")
@Action(value = "cargaArchivo", results =
{
    @Result(location = "/carga/archivoCargado.jsp"),
    @Result(name="input", location = "/carga/formulario.jsp")
})
public class CargaArchivo extends ActionSupport

```

Ya tenemos todo así que procedemos a ejecutar la aplicación y a ingresar a la siguiente dirección:

<http://localhost:8080/formularios/carga/formulario.jsp>

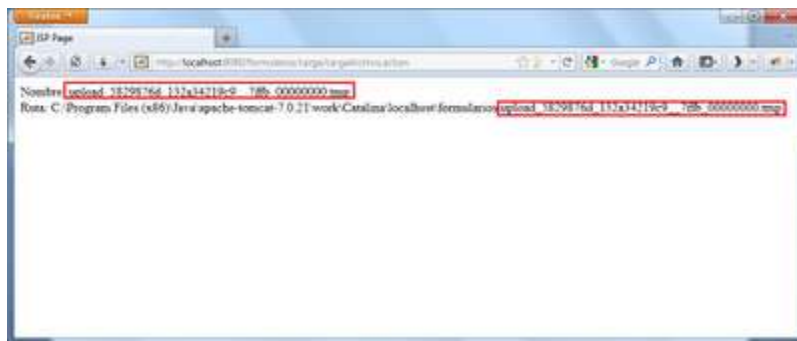
Con lo que veremos el formulario que creamos:



Podemos ver que en el campo de tipo "**file**" que declaramos, existe un pequeño botón que dice "**Examinar...**". Al presionar este botón se abrirá un cuadro de dialogo que nos permitirá seleccionar un archivo; en mi caso seleccionaré una imagen al azar de mi biblioteca de imágenes:

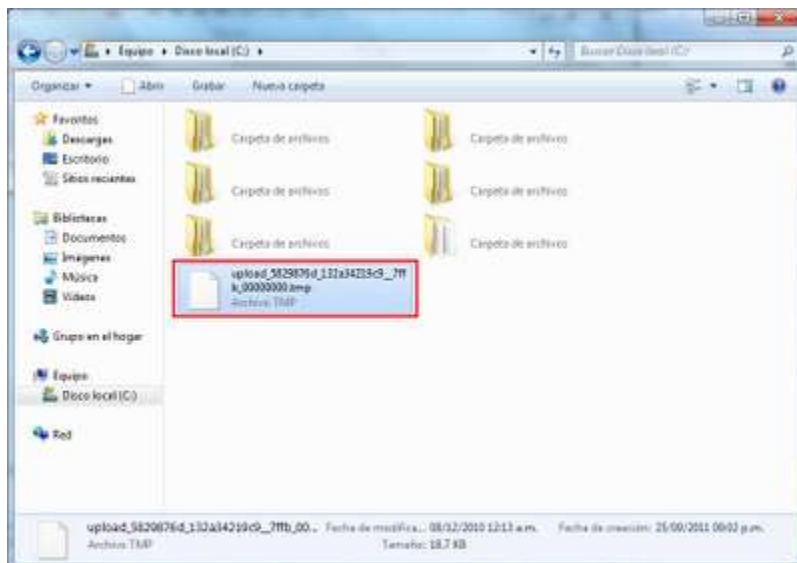


Cuando presionemos el botón **"Enviar"** de nuestro formulario, veremos en la página resultante una salida similar a la siguiente:



Podemos notar antes que nada que el archivo que habíamos subido, llamado **"logoJT.png"**, ahora se llama **"upload_5829876d_132a34219c9_7ffb_00000000.tmp"**. Esto ocurre porque cuando enviamos un archivo usando un formulario, el archivo no viaja como tal por la red; son sus bytes los que son enviados (lo sé técnicamente sería lo mismo ya que un archivo es un conjunto de bytes), el interceptor **"fileUpload"** toma estos bytes y los coloca en un archivo nuevo al cual le coloca un nombre extraño.

Si observamos el directorio a donde hemos movido el archivo que habíamos obtenido notaremos que nuestro archivo efectivamente ha quedado ahí, pero con el nuevo nombre (y extensión) que le ha colocado el interceptor:



Podemos ver que no tenemos información del nombre original del archivo ni de su tipo de contenido. Como podemos imaginar, ambos son datos muy importantes que nos pueden servir en algún momento.

Afortunadamente para nosotros, **Struts 2** proporciona una forma en la que podemos obtener esta información de una forma fácil y sencilla (como nos tiene mal acostumbrados este framework ^^).

Para obtener esta información debemos proporcionar dos atributos extra para el archivo (con sus correspondientes **getters**). El nombre de estos atributos (en realidad solo los **setters**, pero creo que es más fácil entender de esta forma) debe seguir una cierta convención si queremos que **Struts 2** proporcione la información de manera correcta.

Para obtener el nombre original del archivo debemos proporcionar un atributo, de tipo **String**, cuyo identificador sea "<nombre_del_campo_del_archivo>FileName"; o sea que si el identificador del campo del archivo es "documento", el identificador del campo para el nombre del archivo debe ser "documentoFileName", si el identificador del campo es "miArchivo", el campo para el nombre debe ser "miArchivoFileName".

Para obtener el tipo de contenido del archivo, o sea el "content type", debemos hacer algo similar y proporcionar un campo cuyo nombre sea "<nombre_del_campo_del_archivo>ContentType". Los campos para estos dos datos, junto con sus **setters**, queda de la siguiente forma:

```
private String archivoFileName;
private String archivoContentType;

public void setArchivoContentType(String archivoContentType)
{
    this.archivoContentType = archivoContentType;
}

public void setArchivoFileName(String archivoFileName)
{
    this.archivoFileName = archivoFileName;
}
```

Ahora con estos datos podemos modificar un poco nuestro **Action** dejando el método "execute" de la siguiente forma:

```
public String execute() throws Exception
{
    File nuevoArchivo = new File("/", archivoFileName);
    archivo.renameTo(nuevoArchivo);

    return SUCCESS;
}
```

Ahora el nombre con el que guardaremos el archivo que recibimos, es el mismo nombre del archivo original.

Agregaremos también un **getter** para cada una de las propiedades que no hemos utilizado aún, o sea para "autor" y "archivoContentType" para poder leerlos desde la **JSP** del resultado:

```

public String getArchivoContentType()
{
    return archivoContentType;
}

```

```

public String getAutor()
{
    return autor;
}

```

Al final, nuestra clase "**CargaArchivo**" queda de la siguiente forma:

```

@Namespace(value = "/carga")
@Action(value = "cargaArchivo", results =
{
    @Result(location = "/carga/archivoCargado.jsp"),
    @Result(name="input", location = "/carga/formulario.jsp")
})
public class CargaArchivo extends ActionSupport
{
    private String autor;
    private File archivo;
    private String archivoFileName;
    private String archivoContentType;

    @Override
    public String execute() throws Exception
    {
        File nuevoArchivo = new File("/", archivoFileName);
        archivo.renameTo(nuevoArchivo);

        return SUCCESS;
    }

    public String getArchivoContentType()
    {
        return archivoContentType;
    }

    public String getAutor()
    {
        return autor;
    }

    public void setArchivoContentType(String archivoContentType)
    {
        this.archivoContentType = archivoContentType;
    }

    public void setArchivoFileName(String archivoFileName)
    {
        this.archivoFileName = archivoFileName;
    }
}

```

```

    }

    public String getNombre()
    {
        return archivoFileName;
    }

    public String getRuta()
    {
        return archivo.getAbsolutePath();
    }

    public void setArchivo(File archivo)
    {
        this.archivo = archivo;
    }

    public void setAutor(String autor)
    {
        this.autor = autor;
    }
}

```

Debemos modificar ahora la **JSP "archivoCargado"** del directorio "**carga**" para que quede de la siguiente forma:

```

Nombre: <s:property value="nombre" /><br />
Ruta: <s:property value="ruta" /><br />
Autor: <s:property value="autor" /><br />
Content Type: <s:property value="archivoContentType" />

```

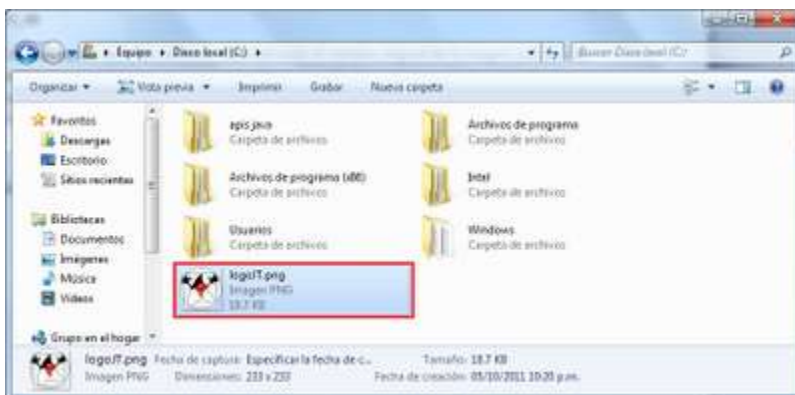
Con todos los datos que necesitamos. volvemos a ejecutar la aplicación, y debemos ver el mismo formulario de las últimas veces:



Subimos nuevamente nuestro archivo, y al presionar el botón "**Enviar**" deberemos ver la siguiente pantalla:

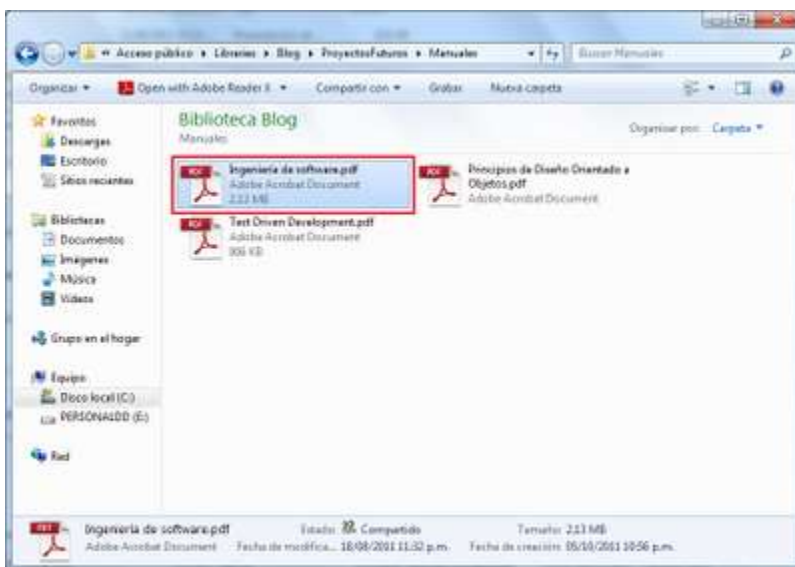


Como podemos observar, ahora se está indicando el nombre original de la imagen, y el tipo de archivo que se subió (que en este caso es "**image/png**"). Si vamos nuevamente al directorio raíz de nuestro sistema operativo veremos que ahora la imagen se ha almacenado de forma correcta, y podemos ver un preview de la misma:

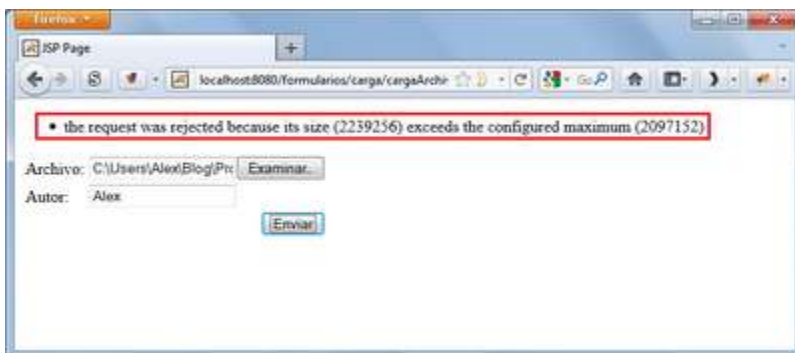


Con lo que podemos comprobar que la carga se realizó de forma correcta ^_^.

Hagamos una segunda prueba, intentemos subir el siguiente archivo (**como tip para lo que viene a continuación, fíjense en el tamaño del archivo**).



Al tratar de enviar este archivo veremos que obtenemos... un mensaje de error:



Lo que el mensaje básicamente dice es que el tamaño de nuestro archivo excede el tamaño máximo de **2MB** que **Struts 2** tiene configurado por defecto. ¿Qué podemos hacer entonces para subir archivos que sean más grandes? Como deben

estarse imaginando, **Struts 2** proporciona una forma de configurar el tamaño máximo de los archivos que se pueden cargar, a través de una constante o a través de un parámetro del interceptor "**fileUpload**". Estos dos valores no son exactamente para lo mismo, pero lo explicaremos en su debido momento.

Primero veamos cómo establecer este valor como una constante.

Como recordarán, del primer tutorial de la serie, hay dos formas de definir las constantes de **Struts 2**, la primera es en el archivo "**struts.xml**" que normalmente usamos cuando realizamos una configuración con **XML**. El nombre de la constante que debemos definir es "**struts.multipart.maxSize**".

Esta variable quedaría de la siguiente forma en el archivo "**struts.xml**" si quisiéramos que el tamaño máximo del archivo fuera de **10MB**:

```
<constant name="struts.multipart.maxSize" value="10485760" />
```

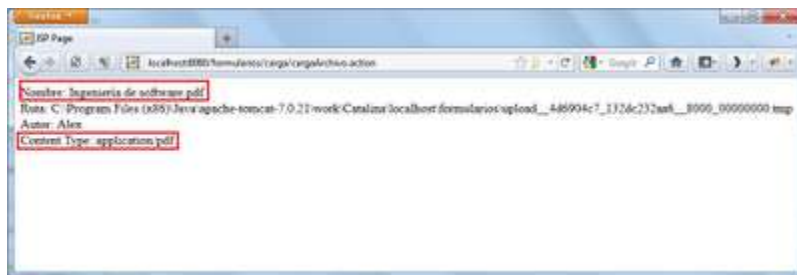
En donde el valor indica el peso máximo que puede tener el archivo en bytes (así es, leyeron bien: bytes). El valor es calculado de la siguiente forma:

```
10 * 1024 * 1024  
//MB   KB   Bytes
```

Si no estamos usando un archivo de configuración, como es nuestro caso, la constante se define como un parámetro de inicialización en el filtro de **Struts 2**:

```
<filter>  
<filter-name>struts2</filter-name>  
<filter-  
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-  
class>  
<init-param>  
<param-name>struts.multipart.maxSize</param-name>  
<param-value>5097152</param-value>  
</init-param>  
</filter>
```

Si intentamos nuevamente subir nuestro archivo, veremos la siguiente salida:



Podemos ver que en esta ocasión logramos subir el archivo de forma correcta.

El utilizar la constante anterior modifica el límite del tamaño de los archivos en toda la aplicación. Esto es bueno si deseamos aumentar el límite en el tamaño de los archivos que se vaya a subir en cualquier formulario de la aplicación. ¿Pero qué pasaría si necesitáramos que en algún formulario particular el tamaño máximo de los archivos a subir fuera menor que en el resto de la aplicación? Para estos casos es que existe una forma para especificar un límite para un **Action** particular, estableciendo el parámetro "**maximumSize**" del interceptor "**fileUpload**" en el **Action** que queremos modificar.

Ojo con lo dicho anteriormente, esta segunda forma **lo único que permite es hacer que el tamaño máximo para un Action particular sea menor que el resto de la aplicación**, lo contrario (que el tamaño máximo para un **Action** sea mayor que para el resto de la aplicación) no se puede hacer.

Para personalizar el valor de este interceptor para un **Action**, cuando trabajamos con el archivo "**struts.xml**", usamos el elemento "**<interceptor-ref>**" para indicar qué interceptores son los que queremos aplicar a un **Action** particular y los valores de los parámetros de dichos interceptores. Por ejemplo, para el **Action "cargaArchivo"**, si estuviéramos trabajando con un el archivo "**struts.xml**" quedaría de la siguiente forma para que el tamaño máximo de un archivo sea de **2MB (2* 1024 * 1024)**:

```
<action name="cargaArchivo"
class="com.javatutoriales.struts2.formularios.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="maximumSize">2097152</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>

<result>/carga/archivoCargado.jsp</result>
<result name="input">/carga/formulario.jsp</result>

</action>
```

Para cuando trabajamos con anotaciones la configuración queda de la siguiente forma:

```
@InterceptorRefs(value =
{
    @InterceptorRef(value = "fileUpload", params =
    {
        "maximumSize", "2097152"
    } ),
    @InterceptorRef(value = "defaultStack")
})
```

Como de esta forma lo que hacemos es indicar los interceptores que se le aplicarán a este **Action**, debemos decir que además del interceptor "**fileUpload**" queremos aplicar el resto de los interceptores que se aplican normalmente a un **Action** y que se encuentran en el "**defaultStack**", como explicamos en [el primer tutorial de la serie](#).

Si intentamos subir nuevamente nuestro archivo obtendremos el siguiente mensaje de error:



Podemos ver que este mensaje es ligeramente distinto al que habíamos obtenido anteriormente, pero nos sirve para comprobar que efectivamente el límite se ha modificado.

Si estamos realizando la carga de varios archivos, el tamaño máximo será el de la suma del tamaño de todos los archivos, y no de cada archivo individual. Además el tamaño máximo de archivos que el framework soporta, según la documentación oficial, es de **2GB**.

Adicionalmente a restringir el tamaño máximo de los archivos que podemos cargar, **Struts 2** nos permite limitar también el tipo (**content-type**) de los archivos que se cargarán, estableciendo el parámetro "**allowedTypes**" del interceptor "**fileUpload**". Supongamos que para el **Action** anterior, solo queremos permitir que se carguen archivos de imagen en

formato **"png"**. Para lograr esto debemos configurar el interceptor **"fileUpload"** para nuestro **Action** de la siguiente forma:

Si usamos el archivo **"struts.xml"**:

```
<action name="cargaArchivo"
class="com.javatutoriales.struts2.formularios.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="maximumSize">2097152</param>
<param name="allowedTypes">image/png</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>

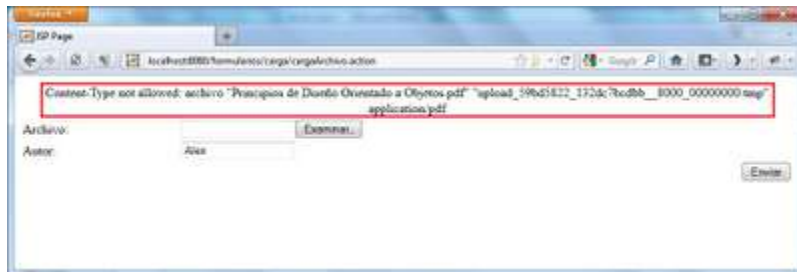
<result>/carga/archivoCargado.jsp</result>
<result name="input">/carga/formulario.jsp</result>

</action>
```

Si usamos anotaciones:

```
@InterceptorRefs(value =
{
    @InterceptorRef(value = "fileUpload", params =
    {
        "maximumSize", "2097152", "allowedTypes", "image/png"
    }) ,
    @InterceptorRef(value = "defaultStack")
})
```

Al intentar subir cualquier archivo que no sea una imagen en formato **png**, obtendremos el siguiente error:



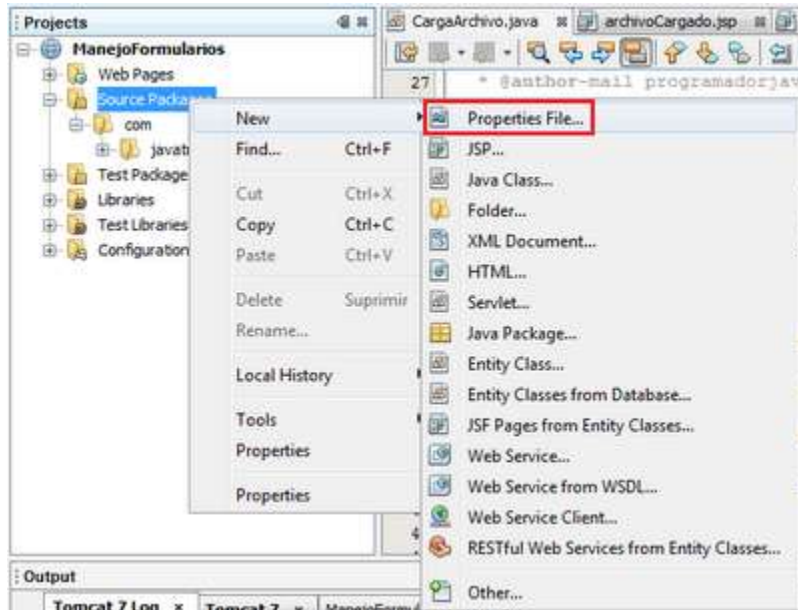
Podemos indicar varios formatos permitidos separando cada elemento con comas, por ejemplo:

```
"image/jpg, image/png, image/ico"
```

Para terminar de hablar de carga de archivos debemos saber cómo personalizar los mensajes de error que se generan al cargar archivos. Los mensajes que vimos anteriormente, aunque bastante claros tal vez no sean los que queremos que nuestros usuarios finales vean en la aplicación. Para personalizar estos mensajes tenemos tres llaves que nos permiten indicar los mensajes que queremos mostrar en cada caso. Estas llaves son:

- **struts.messages.error.uploading** - Un error general que ocurre y que impide subir un archivo
- **struts.messages.error.file.too.large** - Ocurre cuando un archivo cargado es más grande que el tamaño máximo especificado
- **struts.messages.error.content.type.not.allowed** - Ocurre cuando el archivo cargado no es del tipo permitido (content-type) para la carga

Estas llaves debemos colocarlas en un archivo de propiedades que contendrá los mensajes de error de la aplicación. Para crear este archivo hacemos clic derecho sobre el nodo "**Source Package**" del panel de proyectos. En el menú contextual que aparece seleccionamos la opción "**New -> Properties File...**" (si no tienen esa opción en el menú, seleccionen la opción "**Other...**" y en la ventana que se abre seleccionen la categoría "**Other**" y el tipo de archivo "**Properties File**"):



Llamaremos a este archivo "**struts-mensajes**" (el IDE se encargará de colocar automáticamente la extensión **.properties**). Damos clic en el botón "**Finish**" y veremos aparecer en el editor nuestro archivo de propiedades. En este archivo colocaremos los textos que los usuarios verán en caso de que ocurra algún error, por ejemplo podemos poner:

```
struts.messages.error.file.too.large=El archivo proporcionado supera el tamaño máximo
struts.messages.error.content.type.not.allowed=El archivo proporcionado no es del tipo
adecuado
```

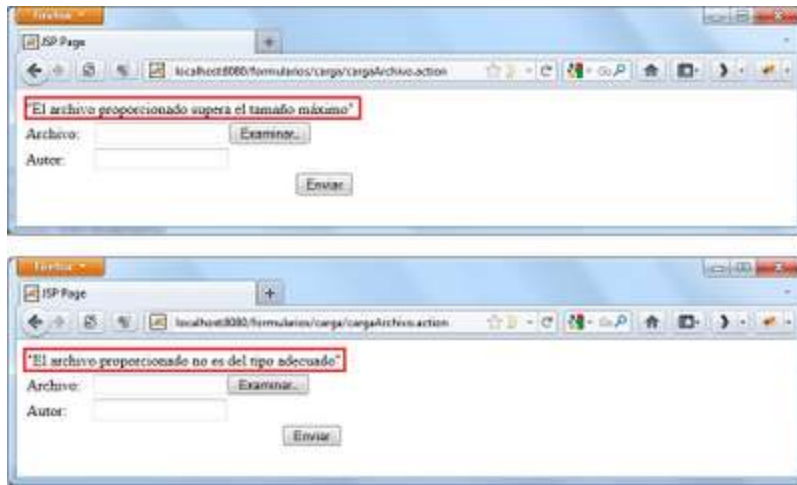
Ya que tenemos definidos los mensajes, lo siguiente que debemos hacer es indicarle a **Struts 2** dónde se localiza este archivo. Para ello (si, adivinaron) usamos una constante para indicar el nombre del archivo (el cual será buscado a partir del directorio raíz de los paquetes de la aplicación, o sea en el nodo "**Source Packages**"). La constante que usamos es "**struts.custom.i18n.resources**", y como ya hemos visto, podemos colocarle en el archivo "**struts.xml**" de la siguiente forma:

```
<constant name="struts.custom.i18n.resources" value="struts-mensajes" />
```

O como un parámetro de inicialización en el filtro de **struts**:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
<init-param>
<param-name>struts.custom.i18n.resources</param-name>
<param-value>struts-mensajes</param-value>
</init-param>
</filter>
```

Cuando volvamos a ejecutar la aplicación veremos los siguientes mensajes de error:



Esto ya está mejor, ya que podemos colocar un texto tan descriptivo como queramos, sin embargo ahora hemos perdido algo de información importante que los otros mensajes nos daban, como el nombre del archivo que estamos subiendo, y algunos parámetros particulares para cada error.

Afortunadamente una vez más **Struts 2** nos da una solución a esto ya que dentro de los mensajes podemos colocar algo llamado "**placeholders**" o contenedores, que básicamente es un espacio en nuestro mensaje donde **Struts 2** se encargará de colocar un parámetro. Los parámetros que proporciona cada tipo de error son:

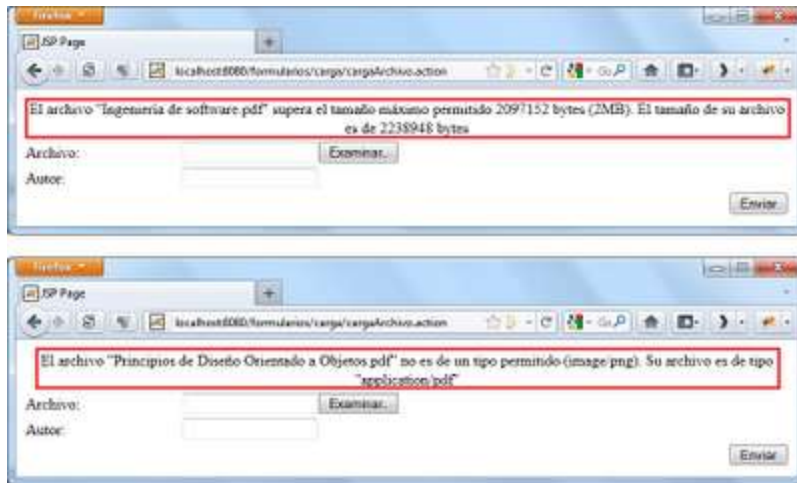
- Para "**struts.messages.error.file.too.large**":
 - **0** - Nombre del parámetro que causó el error (en nuestro ejemplo "**archivo**")
 - **1** - Nombre del archivo que causó el error
 - **2** - Nombre del archivo en el servidor (el archivo temporal creado por el interceptor)
 - **3** - Tamaño del archivo cargado
- Para "**struts.messages.error.content.type.not.allowed**":
 - **0** - Nombre del parámetro que causó el error (en nuestro ejemplo "**archivo**")
 - **1** - Nombre del archivo que causó el error
 - **2** - Nombre del archivo en el servidor (el archivo temporal creado por el interceptor)
 - **3** - **Content-type** del archivo cargado

Como podemos ver, en ambos casos los parámetros son prácticamente los mismos.

Podemos modificar los mensajes anteriores para que queden de la siguiente forma:

```
struts.messages.error.file.too.large=El archivo "{1}" supera el tamaño máximo permitido
2097152 bytes (2MB). El tamaño de su archivo es de {3} bytes
struts.messages.error.content.type.not.allowed=El archivo "{1}" no es de un tipo
permitido (image/png). Su archivo es de tipo "{3}"
```

Con lo que obtendríamos los siguientes mensajes de error:



Que ya son mucho más claros para nuestros usuarios finales ^_^.

Como una nota final sobre la carga de archivos debemos decir que existe una constante más, "**struts.multipart.saveDir**", que podemos usar para indicar dónde queremos que se guarden los archivos temporales de las cargas que genera **Struts 2**. El uso de esta variable sale del propósito de este tutorial ^_^.

Ahora que hemos hablado bastante de la carga de archivos, pasemos al último tema del tutorial. Veamos como envías archivos desde nuestro servidor a los clientes usando **Struts 2**:

7. Descarga de Archivos

El poder enviar información binaria o archivos a los usuarios, aunque no es estrictamente parte del trabajo con formularios, es también una parte importante del desarrollo de aplicaciones web ya que tiene muchas aplicaciones prácticas como el enviar un reporte que generamos de forma dinámica, o un archivo que tengamos almacenado en algún lugar, como una base de datos o un directorio externo de la aplicación, o también mostrar imágenes que se hayan cargado o generado de forma dinámica en el sistema.

El realizar la descarga o envío de archivos es una tarea muy simple de hacer cuando trabajamos con **Struts 2**, ya que nos proporciona un tipo de resultado especial que permite enviar bytes directamente al "**ServletOutputStream**" del "**HttpServletResponse**".

Dedicaremos un tutorial completo a hablar de tipos de resultados, así que no entraré en muchos detalles de cómo funcionan estos, pero debemos saber que existen varios tipos de resultados, cada uno de los cuales altera de alguna forma la respuesta que enviamos al cliente. Podemos establecer algunos parámetros para indicar alguna información a nuestro **result**, entre otras cosas.

El **result** que nos ayudará en este caso es llamado "**Stream Result**", y básicamente es un **result** que permite enviar un flujo de bytes al cliente. Los parámetros que acepta este **result** son:

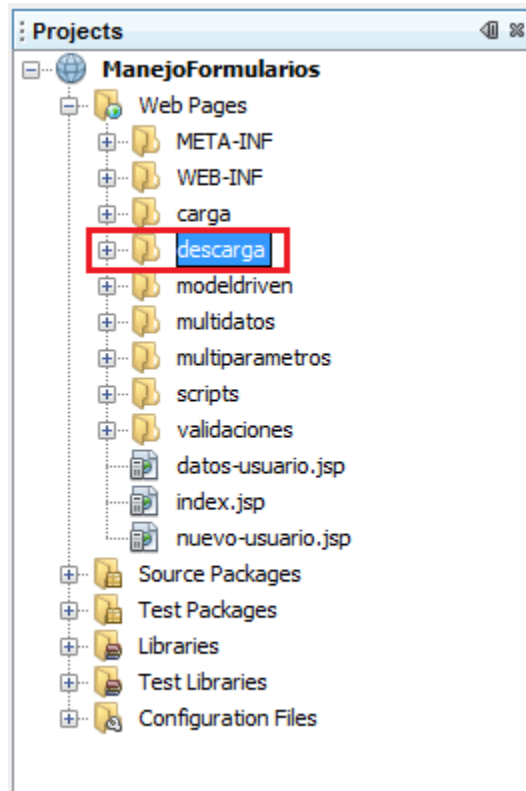
- **contentType**: El tipo de contenido que se está enviando al usuario, por default es "**text/plain**"
- **contentLength**: En número de bytes que se le enviarán al usuario (el navegador lo usa para mostrar de forma correcta una barra de progreso).
- **contentDisposition**: Establece la cabecera "**content disposition**" que especifica el nombre del archivo, por lo regular ponemos algo como "**attachment;filename='documento.pdf'**" que muestra un cuadro de dialogo para guardar el documento, su valor por default es "**inline**" que nos muestra el documento en el mismo navegador
- **inputName**: El nombre del "**InputStream**" que tiene los bytes del archivo que enviaremos al usuario (quedará más claro con el ejemplo), por default es "**inputStream**"
- **bufferSize**: El tamaño del buffer que se usa para del flujo de entrada al de salida, por default es "**1024**"

- **allowCaching:** Indica si el archivo debe ser guardado en caché por el cliente, por default es "**true**"
- **contentCharSet:** El tipo de caracteres que tiene el contenido que enviamos al archivo, no tiene valor por default y si no lo establecemos no pasa nada ^_^.

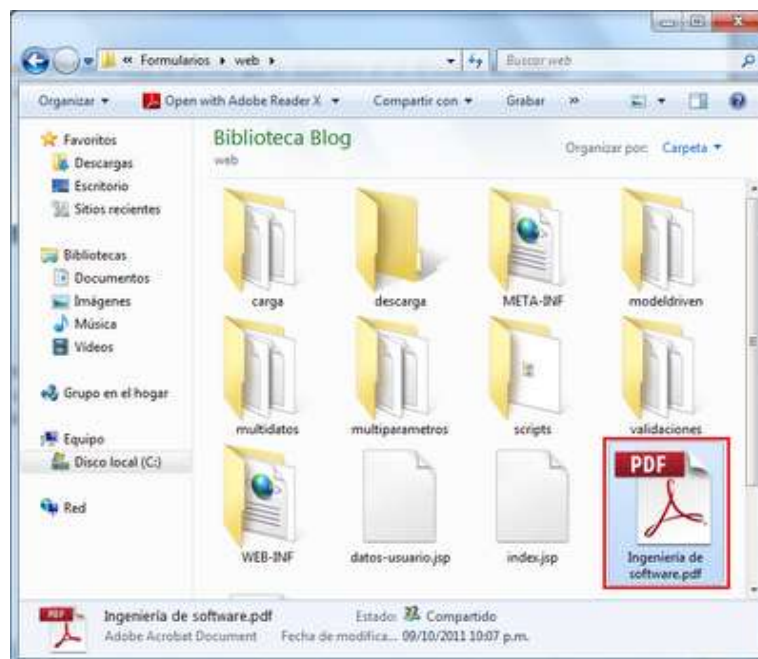
Todos los atributos anteriores son opcionales, y por lo regular solo establecemos dos o tres.

Veremos tres ejemplos de este tipo de **result**.

Lo primero que haremos es crear un nuevo directorio, dentro de las páginas, llamado "**descarga**":



Para el primer ejemplo tomaremos un archivo que se encuentra en un directorio de nuestra aplicación y se lo enviaremos al usuario. Pueden tomar cualquier archivo que les guste, de preferencia alguno que puedan ver en el navegador como un **PDF**. Yo tomaré un archivo al azar de mi máquina y lo colocaré en el directorio raíz de la aplicación web, o sea en el directorio "**web**" de la aplicación que estamos desarrollando:



Abriré este archivo solo para ver el contenido y que cuando lo descarguemos podamos ver que es el mismo:



Ahora crearemos una nueva clase llamada **"MuestraArchivo"**, en el paquete **"actions"**. Esta clase extenderá de **"ActionSupport"**:

```
public class MuestraArchivo extends ActionSupport
{
}
```

Lo primero que debemos hacer es colocar en nuestro **Action** un atributo de tipo **"java.io.InputStream"**. Esta variable contendrá los bytes del archivo que se le enviará al cliente. Existen muchas formas de obtener una referencia a un **"InputStream"** de manera que solo tengamos que indicar la fuente donde está el archivo, y alguna clase especial se encargue del resto. En el peor de los casos tendremos que obtener estos bytes uno a uno y colocarlos en un objeto especial, pero tampoco es tan complicado; nosotros usaremos las dos formas en distintos ejemplos.

Si le damos a esta variable el nombre de **"inputStream"** no tendremos que declarar nada más en la configuración del **result**. Como es lo más fácil para el momento, lo haremos así y agregaremos el correspondiente **getter** de la propiedad:

```
public class MuestraArchivo extends ActionSupport
{
    private InputStream inputStream;

    public InputStream getInputStream()
    {
        return inputStream;
    }
}
```

Ahora sobre-escribiremos el método **"execute"** para poder obtener un flujo de lectura (**"InputStream"**) a un archivo, usando una clase especial, un **"FileInputStream"**. Para obtener este flujo lo único que debemos saber es la ubicación del archivo. Aquí hay que hacer un paréntesis para explicar un poco cómo se obtienen las rutas dentro de nuestra aplicación cuando estamos trabajando con aplicaciones web.

Dentro de la especificación de **Servlets** y **JSPs** se indica que cuando se hace la instalación o deploy de una aplicación web en un servidor (más específicamente en un contenedor de **Servlets** y **JSPs**) este debe quedar en un directorio de la máquina donde está instalado el servidor. ¿En cuál directorio?... esa es la cuestión interesante.

En la especificación no se indica dónde debe quedar cada aplicación que despleguemos, así que una vez instalada la aplicación, no podemos estar seguros de en dónde se encuentra físicamente dentro del servidor, por lo tanto si quisiéramos simplemente leer un archivo de nuestra aplicación no podríamos hacerlo ya que no sabríamos que ruta indicar; y todo esto en nuestro servidor local, si hiciéramos la instalación en un servidor remoto, rentado, o del cliente, sería aún peor ya que tal vez ni siquiera sabríamos el sistema operativo del mismo.

Para remediar esto, la especificación también proporciona una forma de obtener la ruta a cualquier recurso de nuestra aplicación. Para hacer esto debemos obtener una referencia al "**ServletContext**" de la aplicación y luego usar su método "**getRealPath**" indicando el recurso que del cual queremos obtener la ruta, iniciando desde la raíz de la aplicación (lo que vendría siendo el directorio "**web**" en **NetBeans**).

Ahora bien, como estamos usando un framework que se encarga de ocultar algunos detalles de la implementación web de la aplicación (como todo buen framework), no podemos obtener de una forma directa el "**ServletContext**" ya que no tenemos ninguna referencia a la interface de **Servlets** dentro de nuestros **Actions**.

Para remediar esto, **Struts 2** nos proporciona una clase auxiliar que ayuda a obtener la referencia al "**ServletContext**" de una forma sencilla (de hecho también podemos obtener la referencia al "**HttpServletRequest**" y al "**HttpServletResponse**"). Esta clase es "**ServletActionContext**", y tiene una serie de métodos estáticos que permiten obtener referencias a los objetos anteriores.

Después de toda esa explicación, espero que quede claro el siguiente paso dentro de nuestro ejemplo.

Ya tenemos declarada una instancia de "**InputStream**" y hemos colocado un archivo **.pdf**, llamado "**Ingeniería de software**", en la raíz de la aplicación web, ahora crearemos un flujo de entrada para leer ese archivo y así enviarlo al cliente. Hacerlo en realidad es más fácil que decirlo, ya que **Java** proporciona una clase llamada "**FileInputStream**" que hace todo esto de forma automática, lo único que debemos hacer es indicarle la ruta en la que está almacenado nuestro archivo, de la siguiente forma:

```
@Override
public String execute() throws Exception
{
    String ruta = ServletActionContext.getServletContext().getRealPath("/Ingenieria de
software.pdf");

    inputStream = new FileInputStream(ruta);

    return SUCCESS;
}
```

Esto es todo lo que debemos hacer para poder enviar el archivo al usuario, el framework se encargará de hacer el resto.

El paso final en este **Action** es colocar las anotaciones, que en este momento ya debemos conocer de memoria, para indicarle a **Struts 2** que debe tratar esta clase como un **Action**. Primero que nada indicamos el namespace en el que se colocará al **Action**, y se le dará un nombre al mismo:

```
@Namespace(value = "/descarga")

```

Lo siguiente es indicar el resultado de la ejecución de este **Action**, la respuesta que será enviada al cliente, para lo cual usamos el atributo "**results**" y la anotación "**@Result**". En esta ocasión, como estamos usando un tipo distinto de **result**, debemos indicarlo dentro de la anotación usando su atributo "**type**". En este caso debemos indicar que el result es de tipo "**stream**" o flujo de bytes, de la siguiente forma:


```

@Namespace(value = "/descarga")
@Action(value = "muestraArchivo", results =
{
    @Result(type = "stream")
})
public class MuestraArchivo extends ActionSupport

```

El último paso es establecer alguno de los parámetros en el **result**, para eso usamos el atributo "**params**" de esta anotación. Este atributo recibe como argumento **un arreglo de cadenas**, donde **los elementos no representan el nombre del argumento que se quiere establecer**, y **los pares representan el valor de dicho argumento**. En este ejemplo solo usaré el atributo "**contentType**" para indicar que el tipo de archivo que regresaré al usuario es un archivo **pdf** ("**application/pdf**"). Si están usando un archivo de algún otro tipo, una búsqueda rápida en Google les dará el tipo de contenido de su archivo:

```

@Namespace(value = "/descarga")
@Action(value = "muestraArchivo", results =
{
    @Result(type = "stream", params =
    {
        "contentType", "application/pdf"
    })
})
public class MuestraArchivo extends ActionSupport

```

Nuestra clase "**MuestraArchivo**" queda de la siguiente forma:

```

@Namespace(value = "/descarga")
@Action(value = "muestraArchivo", results =
{
    @Result(type = "stream", params =
    {
        "contentType", "application/pdf"
    })
})
public class MuestraArchivo extends ActionSupport
{
    private InputStream inputStream;

    @Override
    public String execute() throws Exception
    {
        String ruta = ServletActionContext.getServletContext().getRealPath("/Ingenieria
de software.pdf");

        inputStream = new FileInputStream(ruta);

        return SUCCESS;
    }

    public InputStream getInputStream()
{
    return inputStream;
}
}

```

Si estuviéramos trabajando con archivos de configuración, el **Action** quedaría de la siguiente forma:

```
<action name="muestraArchivo"
class="com.javatutoriales.struts2.formularios.actions.MuestraArchivo ">
<result type="stream">
<param name="contentType">application/pdf</param>
</result>
</action>
```

Para este tipo de **result**, hablando estrictamente, no se necesita una página para mostrarlo, basta con colocar el nombre del **Action** en la barra de direcciones del navegador; nosotros crearemos una solo para ver cómo podemos invocarlos. Dentro del directorio "**descarga**" de las páginas web del proyecto creamos una nueva **JSP** llamada "**archivo**". Dentro de esta **JSP**, indicamos que haremos uso de la biblioteca de etiquetas de **Struts 2**, con la directiva "**taglib**" correspondiente:

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

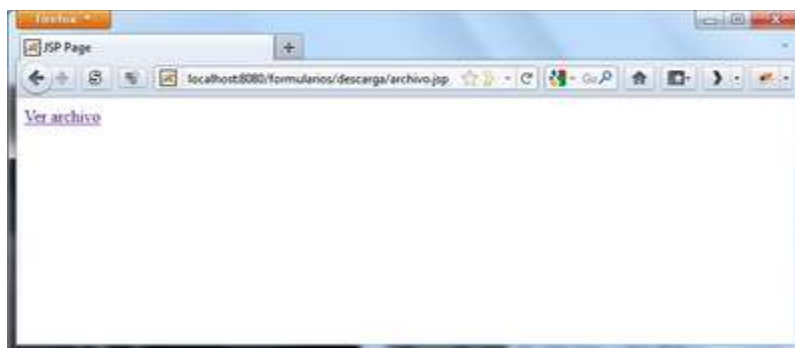
Ahora usaremos la etiqueta "**<s:a>**" para crear un enlace a nuestro **Action**. Esta etiqueta es smuy fácil de utilizar, solo hay que indicar el nombre del **Action**, en su atributo "**action**", su namespace, en su atributo "**namespace**", junto con el texto que tendrá el enlace, de la siguiente forma:

```
<s:a action="muestraArchivo" namespace="/descarga">Ver archivo</s:a>
```

Ahora que está todo listo, ejecutamos nuestra aplicación e ingresamos a la siguiente dirección:

<http://localhost:8080/formularios/descarga/archivo.jsp>

Con lo que debemos ver la siguiente página:



Lo único que tiene esta página es un enlace a nuestro **Action**, cuando entramos en la misma deberemos ver el siguiente resultado:



Como podemos ver, el documento se ha incrustado en nuestro navegador, por lo que nuestro primer ejemplo ha funcionado de forma correcta ^_^.

En el ejemplo anterior, el archivo que se envió al usuario se mostró dentro del mismo navegador. Esto nos sirve con ciertos tipos de archivos y bajo ciertas circunstancias, pero no siempre queremos que el usuario vea los archivos en su navegador, algunas veces queremos que sea forzoso que los descargue a su computadora.

En nuestro segundo ejemplo veremos cómo hacer este cambio. Afortunadamente es algo muy sencillo de hacer, basta con agregar otro parámetro, "**contentDisposition**", a nuestro **result**. "**contentDisposition**", como pueden ver en la lista anterior de parámetros, especifica cómo será enviado este archivo al cliente, si "**inline**" (para mostrarse en el navegador) o "**attachment**" (para que el archivo sea descargado en la máquina del cliente). Ambas opciones nos permiten indicar un "**filename**" que es el nombre que tendrá el archivo al ser enviado.

Como el código del **Action** que se necesita para hacer este ejemplo, es muy parecido al anterior, crearé una nueva clase llamada "**DescargaArchivo**" y copiaré el código anterior.

Agregaremos este parámetro a nuestro **result**, y como nombre del archivo colocaremos simplemente "**tutorial.pdf**":

```
@Namespace(value = "/descarga")

```

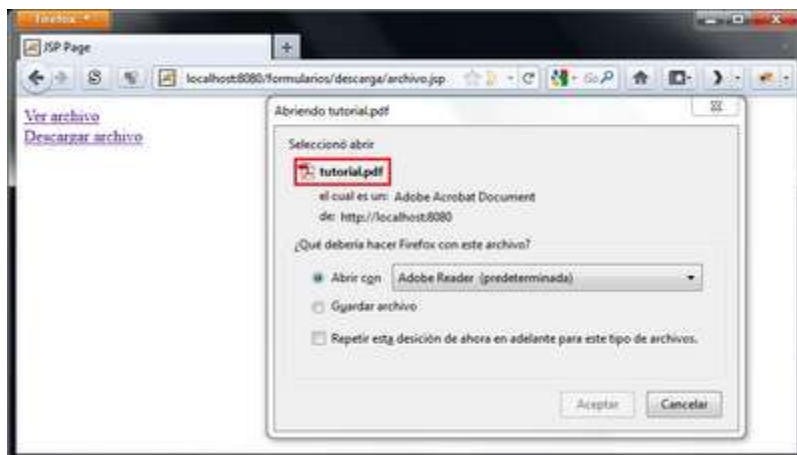
Con archivos **XML** quedaría de la siguiente forma:

```
<result type="stream">
<param name="contentType">application/pdf</param>
<param name="contentDisposition">attachment;filename="tutorial.pdf"</param>
</result>
```

Agregamos la liga correspondiente a este nuevo **Action** en la página "**archivo.jsp**":

```
<s:a action="descargaArchivo" namespace="/descarga">Descargar archivo</s:a>
```

Con este simple cambio, al volver a ejecutar nuestra aplicación y entrar en nuestro **Action**, ahora en lugar de ver el documento en pantalla veremos el cuadro de diálogo del navegador que nos pregunta qué queremos hacer con el archivo:



Podemos ver en la imagen anterior, que el nombre que tendrá el archivo es el mismo nombre que estamos regresando como el **"filename"**. En esta ocasión hemos puesto el nombre de forma estática, pero esta no será siempre la forma en la que queremos indicar el nombre del archivo, en la mayoría de las ocasiones este nombre deberá ser generado de forma dinámica. Para lograr esto, todas las propiedades de todos los **results** de **Struts 2** permiten establecer sus valores usando expresiones, esto es colocar una marca que indique que esa propiedad deberá ser leída de algún **getter** de la clase.

En este caso indicaremos que el nombre del archivo deberá ser obtenido usando el método **"getNombreArchivo"** de la clase:

```
@Result(type = "stream", params =
{
    "contentType", "application/pdf",
    "contentDisposition", "attachment;filename=\"${nombreArchivo}\""
})
```

Podemos ver que solo colocamos **"nombreArchivo"** entre los signos **"\${"** y **"}"** (que indican que lo que esté contenido entre ambos es una expresión) y el framework automáticamente sabrá que debe buscar un **getter**.

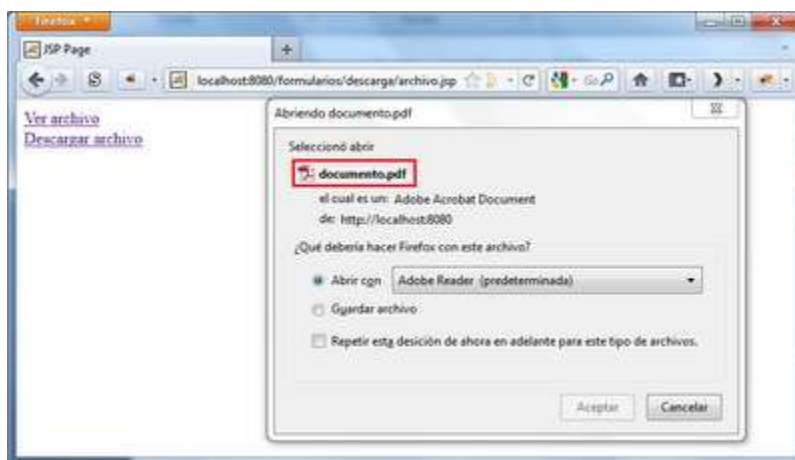
Con archivos **XML** queda exactamente igual:

```
<result type="stream">
<param name="contentType">application/pdf</param>
<param name="contentDisposition">attachment;filename="${nombreArchivo}"</param>
</result>
```

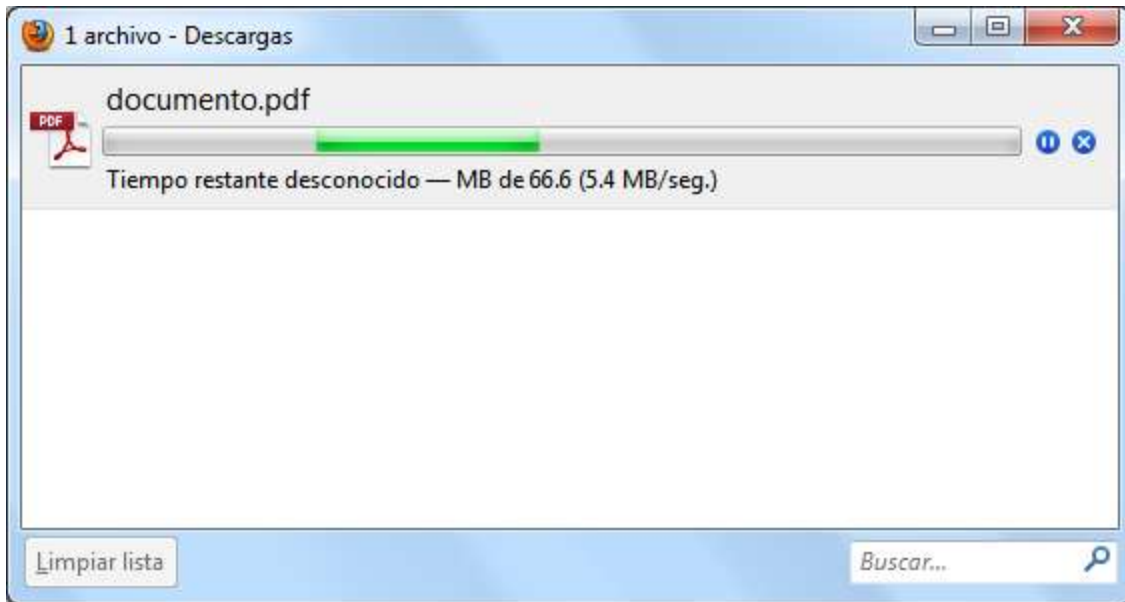
En este caso, nuestro método **"getNombreArchivo"** regresará una cadena estática, pero en una aplicación real esta podría ser generada de forma dinámica:

```
public String getNombreArchivo()
{
    return "documento.pdf";
}
```

Ejecutamos nuevamente la aplicación y podremos ver que ahora el nombre del archivo está siendo efectivamente leído a través del **getter** correspondiente:



Hasta ahora nuestra descarga parece realizarse de forma correcta, pero si prestamos un poco de atención al cuadro de descarga podremos ver que hay algo que no nos agrada mucho:



¿Lo han notado? Y no, no me refiero a mi velocidad de descarga (aunque a mí no me agrada mucho que digamos ^^). El cuadro de descarga nos dice que no conoce el tamaño del archivo que estamos enviando al cliente, solo sabe la cantidad de información que ya ha descargado. Esto es muy molesto para los clientes, porque no saben cuánto tiempo deberán esperar para descargar su archivo.

Remedemos este pequeño error, para eso usaremos un parámetro más que nos proporciona el result **"stream"**, **"contentLength"**. Este parámetro, como su nombre lo indica, permite especificar cuál es el tamaño del archivo que estamos enviando, para que el navegador pueda colocar la barra de progreso de descarga del archivo de forma correcta.

Para usar esta propiedad deberemos hacer unos pequeños cambios en nuestro código. Antes que nada debemos agregar un atributo, con su correspondiente **getter**, que contenga el número de bytes que pesa nuestro archivo:

```
private long bytesArchivo;  
  
public long getBytesArchivo()  
{  
    return bytesArchivo;  
}
```

Ahora debemos modificar el código del método **"execute"** para que obtenga el dato anterior, para eso haremos uso de otro de los constructores de la clase **"FileInputStream"** que hemos estado usando para leer el archivo. Este segundo constructor recibe, en vez de la ruta en la que se encuentra el archivo a leer, un objeto de tipo **"File"** que representa el archivo; por lo que ahora crearemos un objeto de tipo **"File"** y se lo pasaremos al constructor de **"FileInputStream"**:

```
@Override  
public String execute() throws Exception  
{  
    String ruta = ServletActionContext.getServletContext().getRealPath("/Ingenieria de software.pdf");  
  
    File archivo = new File(ruta);  
  
    inputStream = new FileInputStream(archivo);  
  
    return SUCCESS;  
}
```

¿Por qué hemos hecho esto? Bueno, porque la clase **"File"** contiene un método que nos permite obtener justamente el tamaño del archivo, de la siguiente forma:

```
@Override
public String execute() throws Exception
{
    String ruta = ServletActionContext.getServletContext().getRealPath("/Ingenieria de
software.pdf");

    File archivo = new File(ruta);
    bytesArchivo = archivo.length();

    inputStream = new FileInputStream(archivo);

    return SUCCESS;
}
```

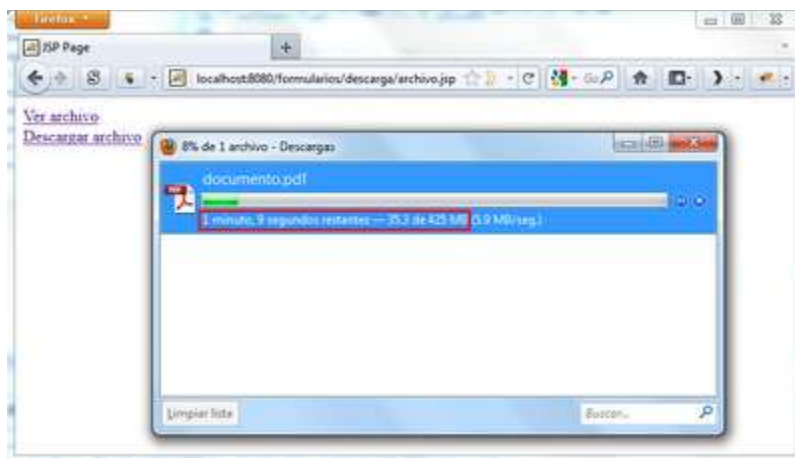
Con esto hemos terminado con las modificaciones al método **"execute"** y lo único que falta es agregar el parámetro correspondiente a nuestro **result**, para esto nuevamente haremos uso de una expresión:

```
@Result(type = "stream", params =
{
    "contentType", "application/pdf",
    "contentDisposition", "attachment;filename=\"${nombreArchivo}\"",
    "contentLength", "${bytesArchivo}"
})
```

Con archivos **XML** queda de la siguiente forma:

```
<result type="stream">
<param name="contentType">application/pdf</param>
<param name="contentDisposition">attachment;filename="${nombreArchivo}"</param>
<param name="contentLength">${bytesArchivo}</param>
</result>
```

Y esto es todo. Ahora cuando volvamos a intentar descargar nuestro archivo obtendremos el siguiente dialogo de descargar:



En esta ocasión se indica el tamaño del archivo y el tiempo que hace falta para concluir con la descarga, por lo que nuestro segundo ejemplo ha funcionado correctamente ^^

Ahora pasaremos a ver el tercer y último ejemplo. En este veremos un uso aún más común del **result "stream"**: **enviar imágenes del servidor al navegador y mostrárselas al usuario**.

Para este último ejemplo del tutorial crearemos una nueva clase llamada **"GeneradorImagenes"**, dentro del paquete **"actions"**, esta clase extenderá de **"ActionSupport"**:

```
public class GeneradorImagen extends ActionSupport
{
}
```

Como regresaremos un flujo de bytes al usuario, debemos declarar una variable de tipo **"InputStream"** con su correspondiente **getter**. El identificador de esta variable puede ser cualquiera que queramos, en mi caso lo llamaré **"imagenDinamica"**:

```
private InputStream imagenDinamica;
```

```
public InputStream getImagenDinamica()
{
    return imagenDinamica;
}
```

Ahora viene la parte interesante, generaremos la imagen de manera dinámica y la guardaremos de forma que al ser enviada al navegador este pueda entenderla para mostrarla.

Sobre-escribimos el método **"execute"** e indicamos cuál será el alto y el ancho de nuestra imagen:

```
@Override
public String execute() throws Exception
{
    final int ANCHO_IMAGEN = 260;
    final int ALTO_IMAGEN = 130;
}
```

El hecho de que las variables estén marcadas como **final** es solo para asegurarme de que su valor no cambie durante la ejecución del método.

Ahora crearemos un objeto que nos permita manipular de una forma sencilla imágenes generadas por nosotros mismos. Para esto haremos uso de la clase **"BufferedImage"**. Esta clase recibe en su constructor el ancho y alto de la imagen, además de indicar el tipo de la imagen que será creada, en nuestro caso será una sencilla imagen en **RGB**:

```
BufferedImage imagen = new BufferedImage(ANCHO, ALTO, BufferedImage.TYPE_INT_RGB);
```

Existen muchas formas de establecer los colores de cada uno de los pixeles de una **"BufferedImage"**: podemos obtener un objeto **"Graphics2D"** usando su método **"createGraphics()"** y posteriormente dibujar sobre este usando el API 2D de Java. Aunque la forma más rápida para este ejemplo es usar directamente su método **"setRGB"**. A este método se le indica la posición del pixel que queremos establecer, en coordenadas "x,y", y el color del pixel en exadecimal, o sea **"0xFF0000"** para el rojo, **"0x00FF00"** para el verde, y **"0x0000FF"** para el azul.

Crearemos dos ciclos, uno para recorrer todas las filas de la imagen, y otro para recorrer todas sus columnas. Dentro del ciclo más anidado estableceremos el color del pixel apropiado usando una instancia de la clase **"Color"** que recibe el valor correspondiente al rojo, verde, azul, y la transparencia del color. Los valores deben estar dentro de un rango de **0** a **255**, siendo **0** el tono más oscuro (negro) y **255** el más claro (blanco). Si el color se sale del rango la imagen no se mostrará, por lo que usaremos el operador modulo **"%"** para hacer que los valores se queden en este rango. Como el valor del pixel debe ser pasado como un entero, usaremos al final el método **"getRGB()"** de **"Color"** para obtener el valor correspondiente al color que estamos estableciendo:

```

for(int alto = 0; alto < ALTO_IMAGEN; alto++)
{
    for(int ancho = 0; ancho < ANCHO_IMAGEN; ancho++)
    {
        imagen.setRGB(ancho, alto, new Color((ancho*alto)%255, (ancho*alto)%255,
(ancho*alto)%255, 255).getRGB());
    }
}

```

También pudimos haber hecho algo más sencillo como:

```

imagen.setRGB(ancho, alto, ancho*alto);

```

Pero no se obtiene el mismo resultado, y es un poco menos claro lo que está pasando ^_^.

Nuestra imagen ya tiene color, ahora debemos transformarla a un formato que pueda ser entendido por un navegador web. Para esto haremos uso de la clase "**ImageIO**", la cual tiene un método estático, "**write**", que permite convertir las imágenes en formatos "**png**", "**jpg**", "**gif**" y "**bmp**". Este método recibe como parámetros la imagen que queremos dibujar, el formato en el queremos que quede la imagen, y un "**OutputStream**" en donde quedarán los bytes de la imagen.

Para el ultimo parámetro usaremos un tipo de "**OutputStream**" que nos permita recuperar los bytes de esa imagen de una forma simple, por lo que usaremos un objeto de tipo "**ByteArrayOutputStream**":

```

ByteArrayOutputStream bytesImagen = new ByteArrayOutputStream();
ImageIO.write(imagen, "png", bytesImagen);

```

El último paso es crear el "**InputStream**" desde el cual **Struts 2** leerá nuestra imagen. La imagen ya está en un arreglo de bytes, el del objeto de tipo "**ByteArrayOutputStream**", por lo que usaremos un objeto que nos permita usar este arreglo de bytes para crear un "**InputStream**". Usaremos un objeto de tipo "**ByteArrayInputStream**" que en su constructor recibe un arreglo de bytes, en este caso el arreglo de bytes que representa nuestra imagen:

```

imagenDinamica = new ByteArrayInputStream(bytesImagen.toByteArray());

```

Eso es todo. Nuestro método "**execute**" queda de la siguiente forma:


```

public String execute() throws Exception
{
    final int ANCHO_IMAGEN = 260;
    final int ALTO_IMAGEN = 130;

    BufferedImage imagen = new BufferedImage(ANCHO_IMAGEN, ALTO_IMAGEN,
    BufferedImage.TYPE_INT_RGB);

    for(int alto = 0; alto < ALTO_IMAGEN; alto++)
    {
        for(int ancho = 0; ancho < ANCHO_IMAGEN; ancho++)
        {
            imagen.setRGB(ancho, alto, new Color((ancho*alto)%255, (ancho*alto)%255,
(ancho*alto)%255, 255).getRGB());
        }
    }

    ByteArrayOutputStream bytesImagen = new ByteArrayOutputStream();
    ImageIO.write(imagen, "png", bytesImagen);

    imagenDinamica = new ByteArrayInputStream(bytesImagen.toByteArray());

    return SUCCESS;
}

```

Como ven, es más sencillo de lo que parecía ^_^.

Para terminar con nuestro **Action** debemos colocar las anotaciones que indican que esta clase debe ser tratada como un **Action** de **Struts 2**. Casi todas ya las hemos visto hasta el cansancio, por lo que solo comentaré que nuestro **result** debe ser de tipo **"stream"** y que le estableceremos dos parámetros: **"inputName"** que indica el nombre de la propiedad de tipo **"InputStream"** que contiene los bytes que serán regresados al usuario, en nuestro caso esta es **"imagenDinamica"**.

El otro parámetro que debemos establecer es el **"contentType"** que indica el formato de nuestra imagen, en este caso **"image/png"**. También podríamos indicar el tamaño de la imagen usando el parámetro **"contentLength"**, pero esto queda como ejercicio para el lector ^_^!.

```

@Namespace(value = "/descarga")
@Action(value = "imagenGenerada", results =
{
    @Result(type = "stream", params =
    {
        "inputName", "imagenDinamica",
        "contentType", "image/png"
    })
})

```

Con archivos **XML** quedaría de la siguiente forma:

```

<result type="stream">
<param name="inputName">imagenDinamica</param>
<param name="contentType">application/pdf</param>
</result>

```

La clase **"GeneradorImagen"** queda finalmente de la siguiente forma:

```

@Namespace(value = "/descarga")
@Action(value = "imagenGenerada", results =
{
    @Result(type = "stream", params =
    {
        "inputName", "imagenDinamica",
        "contentType", "image/png"
    })
})
public class GeneradorImagen extends ActionSupport
{
    private InputStream imagenDinamica;

    @Override
    public String execute() throws Exception
    {
        final int ANCHO_IMAGEN = 260;
        final int ALTO_IMAGEN = 130;

        BufferedImage imagen = new BufferedImage(ANCHO_IMAGEN, ALTO_IMAGEN,
BufferedImage.TYPE_INT_RGB);

        for(int alto = 0; alto < ALTO_IMAGEN; alto++)
        {
            for(int ancho = 0; ancho < ANCHO_IMAGEN; ancho++)
            {
                imagen.setRGB(ancho, alto, new Color((ancho*alto)%255, (ancho*alto)%255,
(ancho*alto)%255, 255).getRGB());
            }
        }

        ByteArrayOutputStream bytesImagen = new ByteArrayOutputStream();
        ImageIO.write(imagen, "png", bytesImagen);

        imagenDinamica = new ByteArrayInputStream(bytesImagen.toByteArray());

        return SUCCESS;
    }

    public InputStream getImagenDinamica()
{
    return imagenDinamica;
}
}

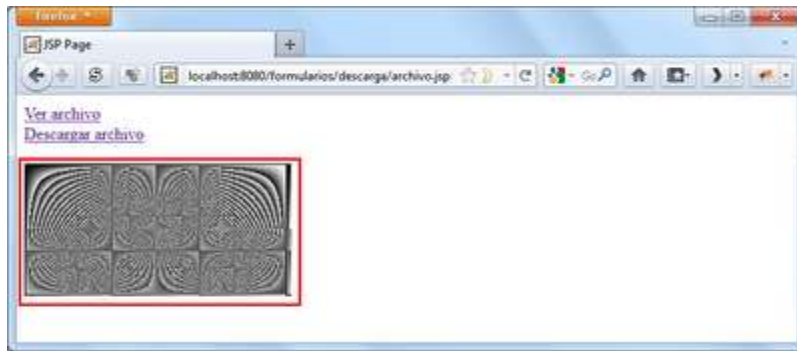
```

El último paso consiste en indicar en una **JSP** que debe mostrar una imagen usando este **Action**. Para eso usaremos la etiqueta "****" de **HTML**. Para no crear una nueva **JSP** colocaremos esta etiqueta en la página "**archivo.jsp**" que hemos estado usando. En esta etiqueta debemos indicar que la fuente (el source) de la imagen en nuestro **Action**, de la siguiente forma:

```

```

Así de simple. Ahora si volvemos a ejecutar nuestra aplicaciones, debemos ver la siguiente salida



No es la mejor imagen del mundo, pero es dinámica y es nuestra ^_^

Struts 2 - Parte 4: Scopes de Objetos Web

Cuando estamos desarrollando una aplicación web debemos almacenar información que será procesada de distinta manera. Dependiendo de cuál sea el propósito de esta información queremos que tenga un tiempo de vida más corto o más largo, alguna información deberá permanecer disponible durante todo el momento que viva nuestra aplicación, mientras que otra solo nos interesará que viva durante una petición. Además habrá información que pertenecerá a cada usuario que acceda a la aplicación y que deberá estar disponible sólo para el usuario correspondiente.

Estos tiempos de vida son llamados **scopes**, y en las aplicaciones web tenemos un cierto número de ellos. Es importante conocer estos **scopes** y ver qué tipo de información es conveniente colocar en cada uno de ellos. A la información que colocamos en los distintos **scopes** les llamamos **atributos**.

También algunas veces es necesario tener un acceso directamente a los objetos del API de **Servlets**, como el **"HttpServletRequest"**, o el **"ServletContext"**, o a los parámetros de la petición,

Struts 2 nos proporciona una forma simple y elegante, además de diversa, para manejar todas estas cosas y en este tutorial aprenderemos estas maneras ^_^.

Las aplicaciones web con Java tienen básicamente tres scopes o tiempos de vida:

- **Application:** Es el scope más largo ya que abarca **el tiempo de vida completo de la aplicación**; esto es, los datos vivirán mientras la aplicación esté activa.
- **Session:** Este scope nos permite tener datos que vivirán a lo largo de **múltiples peticiones HTTP para un mismo usuario**, mientras el usuario esté dentro de la aplicación. Cada usuario verá únicamente sus datos y no habrá forma de que vea los de los demás.
- **Request:** Este es el scope más pequeño, los datos asociados con la petición únicamente estarán disponibles **mientras se realiza dicha petición**.

Existen algunos otros scopes, pero estos son los más y importantes, además no todos los frameworks proporcionan acceso a los demás scopes.

La información o atributos que se puede colocar dentro de estos scopes son **pares nombre valor**, en donde **el nombre debe ser una cadena** y **el valor puede ser cualquier objeto** que nosotros queramos.

Struts 2 nos proporciona tres formas para colocar y leer los atributos que se encuentren en estos scopes:

- Implementación de interfaces **Aware**
- Uso del objeto **"ActionContext"**
- Uso del objeto **"ServletActionContext"**

Las tres son igual de sencillas y nos permiten obtener más o menos los mismos resultados.

Además podemos usar dos de los métodos anteriores, las interfaces **Aware** y el objeto "**ServletActionContext**", para obtener acceso directo a los objetos "**HttpServletRequest**" y "**ServletContext**"

Lo primero que haremos es crear un nuevo proyecto web en NetBeans. Vamos al menú "**File -> New Project...**". En la ventana que aparece seleccionamos la categoría "**Java Web**" y en el tipo de proyecto "**Web Application**". Presionamos el botón "**Next >**" y le damos un nombre y una ubicación a nuestro proyecto; presionamos nuevamente el botón "**Next >**" y en este punto se nos preguntará el servidor que queremos usar. En nuestro caso usaremos el servidor "**Tomcat 7.0**", con la versión 5 de **JEE** y presionamos el botón "**Finish**".

Una vez que tengamos nuestro proyecto debemos recordar agregar la biblioteca "**Struts2**" (o "**Struts2Anotaciones**" si van a hacer uso de anotaciones, como es mi caso ^_^), que creamos en [el primer tutorial de la serie de Struts 2](#). Hacemos clic derecho sobre el nodo "**Libraries**" del proyecto. En el menú que aparece seleccionamos la opción "**Add Library...**". En la ventana que aparece seleccionamos la biblioteca "**Struts2**" o "**Struts2Anotaciones**" y presionamos "**Add Library**". Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto.

Ahora configuramos el filtro "**struts2**" en el deployment descriptor. Abrimos el archivo "**web.xml**" y colocamos el siguiente contenido, como se explicó en [el primer tutorial de la serie](#):

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Comenzaremos viendo cómo poder agregar y leer datos en los scopes anteriores, y cómo mostrarlos en nuestras JSPs.

Manejo de Scopes

Manejo de Scopes usando interfaces Aware

Struts 2 proporciona un conjunto de interfaces llamadas interfaces **Aware** (bueno, creo que solo yo las llamo así ^_^) las cuales permite que nuestros **Actions** reciban cierta información, al momento de inicializarlos. La mayoría de estas interfaces proporcionan un **Map** con los pares nombre valor de los atributos de cada uno de los scopes.

Struts 2 contiene las siguientes interfaces **Aware** para obtener y leer atributos de los scopes:

- **ApplicationAware**
- **SessionAware**
- **RequestAware**

Además de un par de interfaces para recibir el "**HttpServletRequest**" y el "**ServletContext**", pero hablaremos de ellas en su momento.

El uso de estas interfaces es muy intuitivo, creamos un **Action** que implemente la interface que nos interese. Cada una de estas interfaces proporciona un método **setter** que permite inyectar el mapa con los atributos del scope correspondiente.

Los métodos de estas interfaces se muestran a continuación:

```
public interface RequestAware
{
    public void setRequest(Map<String, Object> map);
}
```

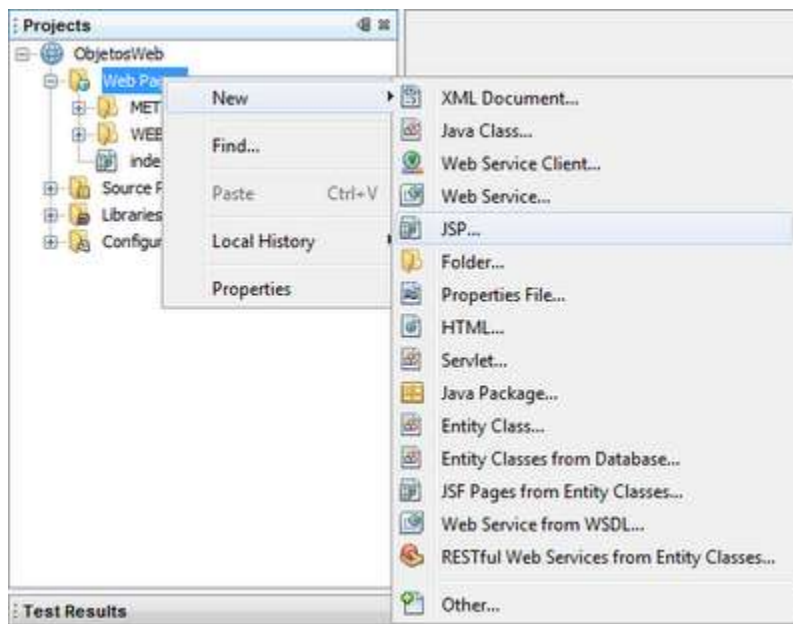
```
public interface SessionAware
{
    public void setSession(Map<String, Object> map);
}
```

```
public interface ApplicationAware
{
    public void setApplication(Map<String, Object> map);
}
```

La forma de los tres métodos es prácticamente la misma: no regresan nada y reciben como único parámetro un **Map** donde el nombre del atributo es un **String** y el valor es un **Object**.

Como la forma de trabajo con estas tres interfaces es también el mismo, hagamos un ejemplo de todo en uno.

Lo primero que haremos es crear una nueva **JSP** llamada "**datosScopesInterfaces.jsp**":



Esta página contendrá un formulario que nos permitirá establecer el valor de algunos datos que colocaremos posteriormente en cada uno de los scopes. Como usaremos las etiquetas de **Struts 2** para generar este formulario, lo primero que debemos hacer es indicar, con la directiva "**taglib**", que usaremos la bibliotecas de etiquetas de **Struts 2**:

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

Ahora crearemos un formulario con unos cuantos campos de texto, para colocar los valores que enviaremos a nuestro **Action** para ser procesados. Cada uno de estos valores tendrá un nombre que indica en cuál scope quedará. Este formulario se enviará a un **Action** llamado "**scopesInterfaces**" que crearemos un poco más adelante:

```
<s:form action="scopesInterfaces">
<s:textfield name="datoSesion" label="Sesion" />
<s:textfield name="datoRequest" label="Request" />
<s:textfield name="datosAplicacion" label="Aplicacion" />
<s:submit />
</s:form>
```

Ahora crearemos un nuevo paquete, llamado "**com.javatutoriales.actions**", para colocar nuestros **Actions**. Dentro de este paquete crearemos una clase llamada "**ScopesInterfacesAction**", esta clase deberá extender de "**ActionSupport**":

```
public class ScopesInterfacesAction extends ActionSupport
{
}
```

Agregaremos un atributo de tipo **String**, con sus correspondientes **setters**, para cada uno de los campos del formulario:

```
public class ScopesInterfacesAction extends ActionSupport
{
    private String datoSesion;
    private String datoRequest;
    private String datosAplicacion;

    public void setDatoRequest(String datoRequest)
    {
        this.datoRequest = datoRequest;
    }

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }

    public void setDatosAplicacion(String datosAplicacion)
    {
        this.datosAplicacion = datosAplicacion;
    }
}
```

Sólo hemos colocado los **setters** porque estableceremos estos valores en los scopes correspondientes y, por lo tanto, los leeremos posteriormente de una manera diferente ^_^.

Ahora el tema que nos interesa, las interfaces. Como toda buena interface de Java, para implementar las interfaces **Aware** lo primero que debemos hacer es declarar que nuestra clase implementará estas interfaces; implementaremos las tres interfaces de una sola vez:

```
public class ScopesInterfacesAction extends ActionSupport implements RequestAware,
SessionAware, ApplicationAware
{
}
```

Como mencioné antes: estas interfaces tienen solo un método cada una (el método que mencioné anteriormente). Normalmente colocamos un atributo, de tipo "**Map<String, Object>**" para almacenar el argumento que es establecido por el framework y poder usarlo posteriormente en el método "**execute**" de nuestro **Action**; después todo lo que hay que hacer es implementar cada uno de los métodos de estas interfaces de la siguiente forma:

```
private Map<String, Object> sesion;  
private Map<String, Object> application;  
private Map<String, Object> request;  
  
public void setRequest(Map<String, Object> map)  
{  
    this.request = map;  
}  
  
public void setApplication(Map<String, Object> map)  
{  
    this.application = map;  
}  
  
public void setSession(Map<String, Object> map)  
{  
    this.sesion = map;  
}
```

En el método "**execute**" lo único que haremos es agregar el valor recibido desde el formulario al scope correspondiente, usando el "**Map**" apropiado:

```
@Override  
public String execute() throws Exception  
{  
    application.put("datoAplicacion", datosAplicacion);  
    sesion.put("datoSesion", datoSesion);  
    request.put("datoRequest", datoRequest);  
  
    return SUCCESS;  
}
```

Así de simple ^^.

Para terminar este **Action** debemos agregar las anotaciones que vimos en [el primer tutorial de la serie](#), y que a estas alturas ya conocemos de memoria. El nombre de nuestro **Action** será "**scopesInterfaces**" y el resultado será enviado a la página "**resultado.jsp**":

```
@Namespace(value = "")  
@Action(value = "scopesInterfaces", results = {@Result(name = "success", location =  
    "/resultado.jsp")})  
public class ScopesInterfacesAction extends ActionSupport implements RequestAware,  
    SessionAware, ApplicationAware  
{  
}
```

La clase "**ScopesInterfacesAction**" completa queda de la siguiente forma:

```

@Namespace(value = "")
@Action(value = "scopesInterfaces", results ={@Result(name = "success", location =
"/resultado.jsp")})
public class ScopesInterfacesAction extends ActionSupport implements RequestAware,
SessionAware, ApplicationAware
{
    private String datoSesion;
    private String datoRequest;
    private String datosAplicacion;
    private Map<String, Object> sesion;
    private Map<String, Object> application;
    private Map<String, Object> request;

    @Override
    public String execute() throws Exception
    {
        application.put("datoAplicacion", datosAplicacion);
        sesion.put("datoSesion", datoSesion);
        request.put("datoRequest", datoRequest);

        return SUCCESS;
    }

    public void setRequest(Map<String, Object> map)
    {
        this.request = map;
    }

    public void setApplication(Map<String, Object> map)
    {
        this.application = map;
    }

    public void setSession(Map<String, Object> map)
    {
        this.sesion = map;
    }

    public void setDatoRequest(String datoRequest)
    {
        this.datoRequest = datoRequest;
    }

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }

    public void setDatosAplicacion(String datosAplicacion)
    {
        this.datosAplicacion = datosAplicacion;
    }
}

```


Ahora crearemos la página "**resultado.jsp**", en la raíz de las páginas web, para comprobar que los datos se han establecido correctamente. En esta página primero indicaremos que haremos uso de la biblioteca de etiquetas de **Struts 2**, como lo hicimos anteriormente. Además usaremos la etiqueta "**<s:property>**" para mostrar el valor en cada uno de los scopes. Para obtener el valor usaremos **OGNL** con el cual, como recordarán del segundo tutorial de la serie, tiene una sintaxis especial para los objetos que se encuentran en los scopes anteriores. Si no recuerdan la sintaxis pueden regresar a ver [el tutorial correspondiente](#)... o continuar leyendo este.

Recuerden que **Struts 2** coloca en el "**ActionContext**" algunos objetos extra además del objeto raíz, para acceder a estos objetos necesitamos usar un operador especial, el operador "#", junto con el nombre del objeto al que queremos acceder. En este caso **Struts 2** coloca 3 objetos especiales que hacen referencia cada uno a uno a uno de los distintos scopes:

- **application**
- **session**
- **request**

Y como habíamos visto, es posible obtener el valor de los atributos colocados en estos scopes usando el operador punto (".") o colocando el nombre del parámetro entre corchetes ("[" y "]").

En este caso, para mostrar el valor almacenado en el **request** debemos hacer:

```
<s:property value="#request.datoRequest" />
```

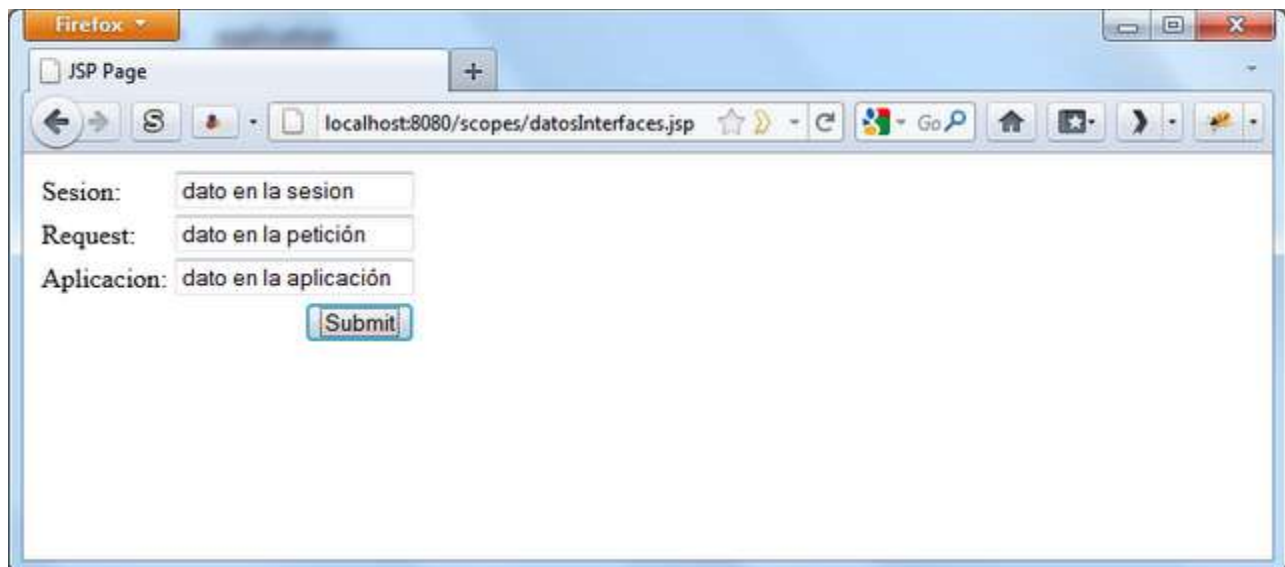
Para el resto de los parámetros se hace algo similar. Al final la página queda de la siguiente forma:

```
Request: <s:property value="#request.datoRequest" /><br />
Sesión: <s:property value="#session.datoSesion" /><br />
Aplicacion: <s:property value="#application.datoAplicacion" /><br />
```

Cuando ejecutemos la aplicación y entremos a la siguiente dirección:

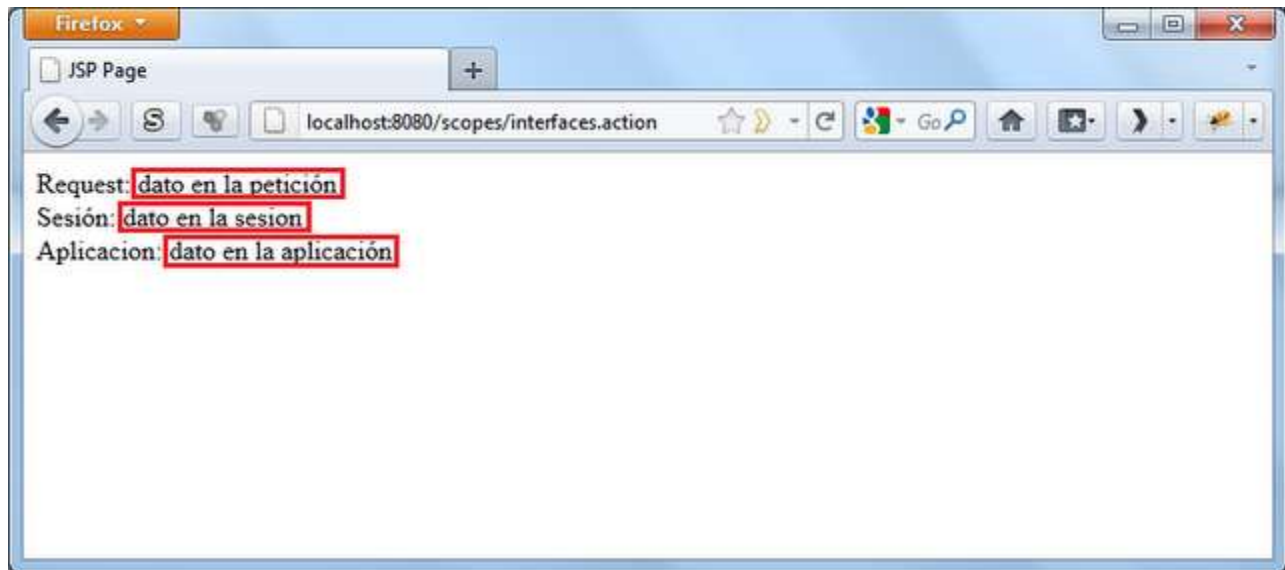
<http://localhost:8080/scopes/datosScopesInterfaces.jsp>

Debemos ver el formulario que creamos anteriormente. Si colocamos los siguientes datos:



The screenshot shows a Firefox browser window with the address bar displaying "localhost:8080/scopes/datosInterfaces.jsp". The page content includes a form with three input fields and a submit button. The first field is labeled "Sesion:" and contains the text "dato en la sesion". The second field is labeled "Request:" and contains the text "dato en la petición". The third field is labeled "Aplicacion:" and contains the text "dato en la aplicación". Below these fields is a "Submit" button.

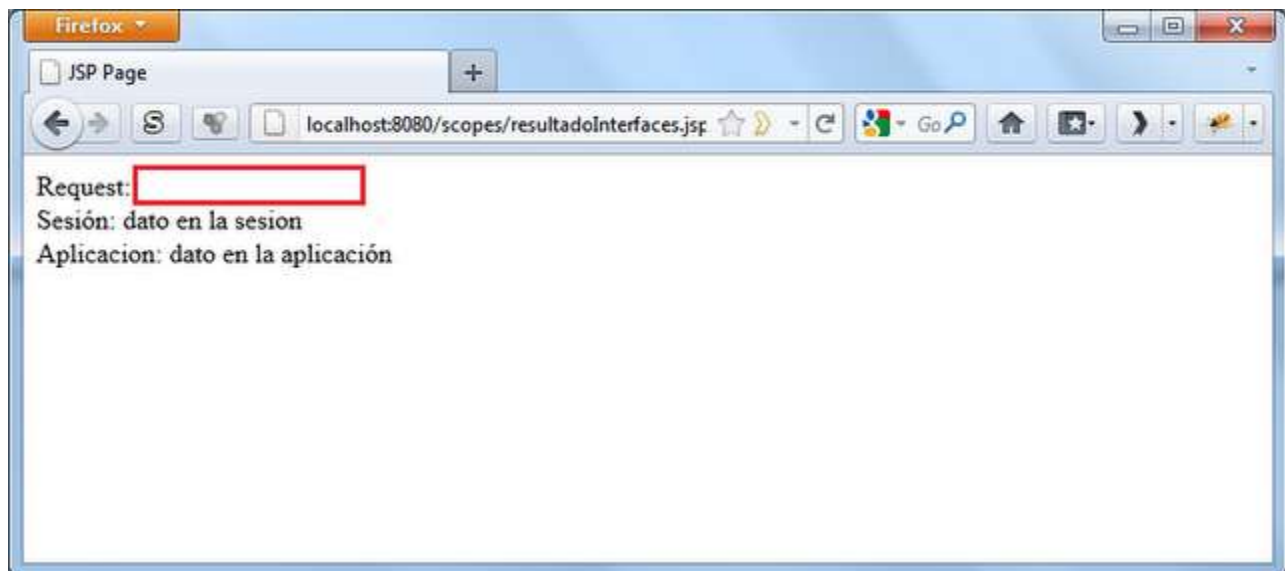
Y presionamos el botón de enviar, veremos la siguiente salida:



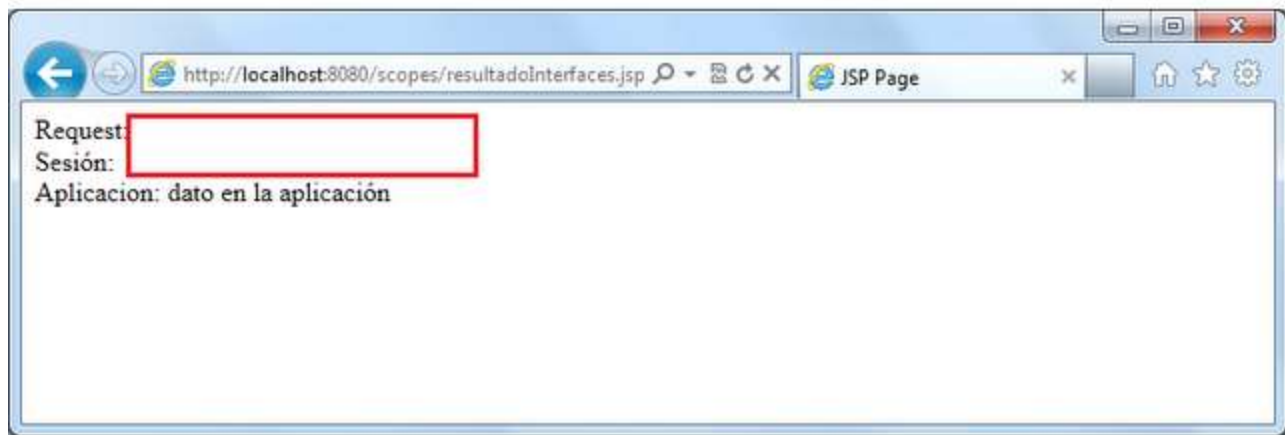
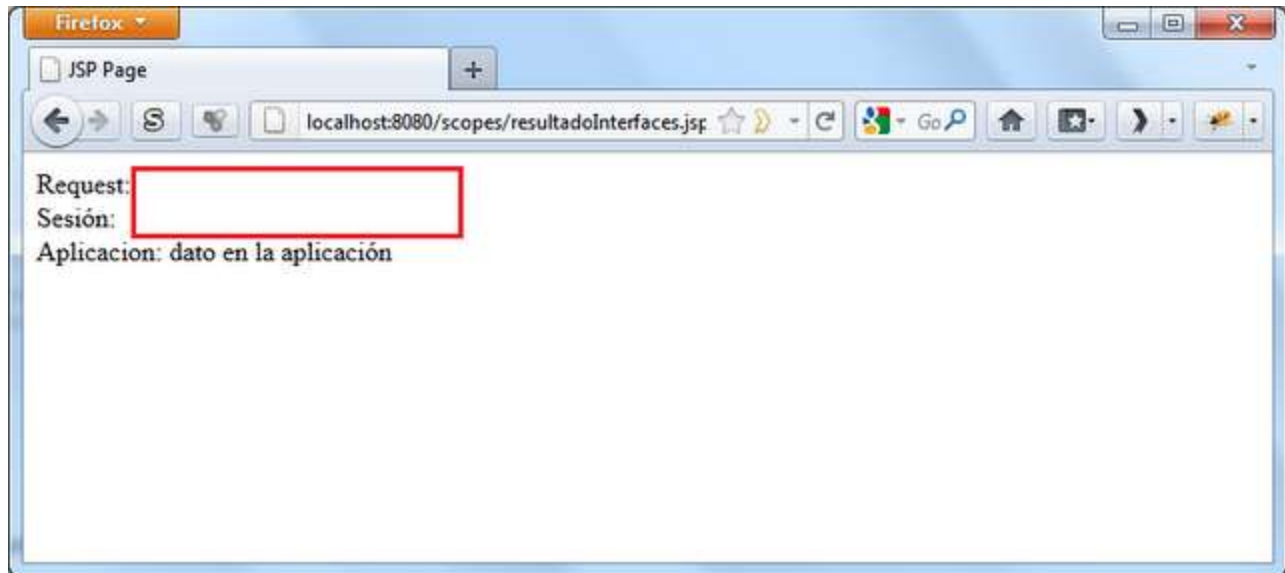
Ahora, ¿cómo podemos comprobar que cada dato está efectivamente en los scopes indicados? Bueno para esto hay varias formas. Primero, para comprobar el atributo del **request** podemos entrar directamente a la página del resultado:

<http://localhost:8080/scopes/resultado.jsp>

Con esto estamos creando una nueva petición, por lo que el atributo que teníamos anteriormente debería desaparecer:



Para el dato de la **sesión**, podemos esperar a que termine nuestra sesión de forma automática (después de 30 minutos), podemos usar otro navegador para entrar a la misma dirección, o simplemente podemos terminar las sesiones activas de nuestro navegador. Con esto los atributos de la sesión deben desaparecer:



Finalmente, y como podemos ver por la imagen anterior del explorer, los atributos del scope aplicación permanecerán hasta que detengamos nuestro servidor (con lo cual, por cierto, ya no podremos entrar al sitio ^_^).

Como podemos ver, esta forma de establecer (y leer) atributos de los distintos scopes es bastante sencilla pero, como dijimos al inicio del tutorial: no es la única forma. Ahora veremos la segunda forma de hacer esto: usando el objeto "**ActionContext**".

MANEJO DE ATRIBUTOS DE SCOPE USANDO ACTIONCONTEXT

La segunda forma que tenemos de manejar los atributos de los scopes es a través de un objeto especial de **Struts 2** llamado "**ActionContext**". Este objeto es el contexto en el cual el **Action** se ejecuta. Cada contexto es básicamente un contenedor para objetos que un **Action** necesita para su ejecución, como la sesión, sus parámetros, etc.

Dentro de los objetos que se encuentran dentro del "**ActionContext**" están justamente los mapas que contienen los atributos de **sesión** y **aplicación** (así es, no se pueden establecer los atributos de **request** usando este objeto), que son los que nos interesan para esta parte del tutorial.

Obtener una referencia al "**ActionContext**" es en realidad muy sencillo ya que, aunque este objeto no es un singleton como tal, lo obtenemos como si fuera uno; veremos esto dentro de un momento, primero crearemos una nueva página que contendrá un formulario como el anterior.

Creemos una nueva página llamada "**datosScopesActionContext.jsp**". Esta página contendrá un formulario parecido al que creamos anteriormente, con la diferencia de que los datos de este serán procesados por un **Action** distinto:

```
<s:form action="scopesActionContext">
<s:textfield name="datoSesion" label="Sesion" />
<s:textfield name="datosAplicacion" label="Aplicacion" />
<s:submit />
</s:form>
```

Ahora crearemos, en el paquete "**actions**", una nueva clase llamada "**ScopesActionContextAction**" que extienda de "**ActionSupport**":

```
public class ScopesActionContextAction extends ActionSupport
{
}
```

Esta clase tendrá dos atributos de tipo **String**, uno para el dato que irá en sesión y otro para el dato que irá en el scope aplicación, con sus correspondientes **setters**:

```
public class ScopesActionContextAction extends ActionSupport
{
    private String datoSesion;
    private String datosAplicacion;

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }

    public void setDatosAplicacion(String datosAplicacion)
    {
        this.datosAplicacion = datosAplicacion;
    }
}
```

Lo último que esta clase necesita es sobre-escribir su método "**execute**" para, haciendo uso del "**ActionContext**", establecer los valores en los scopes adecuados.

Para obtener una instancia del "**ActionContext**" se usa el método estático "**getContext()**" que nos regresa la instancia de "**ActionContext**" adecuada para el **Action** que estamos ejecutando:

```
ActionContext contexto = ActionContext.getContext();
```

Y una vez teniendo una referencia al "**ApplicationContext**" lo único que debemos hacer es usar el método "**getApplication()**" para obtener un mapa con los atributos del scope **application** y "**getSession()**" para obtener un mapa con los atributos del scope **session**:

```
Map<String, Object> application = contexto.getApplication();
Map<String, Object> sesion = contexto.getSession();
```

El resto es solo colocar los atributos que recibidos del formulario en los mapas correspondientes:

```
application.put("datoAplicacion", datosAplicacion);
sesion.put("datoSesion", datoSesion);
```

El método "**execute**" queda de la siguiente forma:

```

@Override
public String execute() throws Exception
{
    ActionContext contexto = ActionContext.getContext();

    Map<String, Object> application = contexto.getApplication();
    Map<String, Object> session = contexto.getSession();

    application.put("datoAplicacion", datosAplicacion);
    session.put("datoSesion", datoSesion);

    return SUCCESS;
}

```

Lo último que falta es anotar nuestro **Action** para indicar que responderá al nombre de "**scopesActionContext**" y que regresará a la página **"/resultado.jsp"** (la cual creamos para el ejemplo anterior así que ahora la reutilizaremos):

```

@Namespace(value = "/")
@Action(value = "scopesActionContext", results =
{
    @Result(name = "success", location = "/resultado.jsp")
})
public class ScopesActionContextAction extends ActionSupport
{
}

```

La clase "**ScopesActionContextAction**" completa queda de la siguiente forma:

```

@Namespace(value = "/")
@Action(value = "scopesActionContext", results ={@Result(name = "success", location =
"/resultado.jsp")})
public class ScopesActionContextAction extends ActionSupport
{
    private String datoSesion;
    private String datosAplicacion;

    @Override
    public String execute() throws Exception
    {
        ActionContext contexto = ActionContext.getContext();

        Map<String, Object> application = contexto.getApplication();
        Map<String, Object> session = contexto.getSession();

        application.put("datoAplicacion", datosAplicacion);
        session.put("datoSesion", datoSesion);

        return SUCCESS;
    }

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }
}

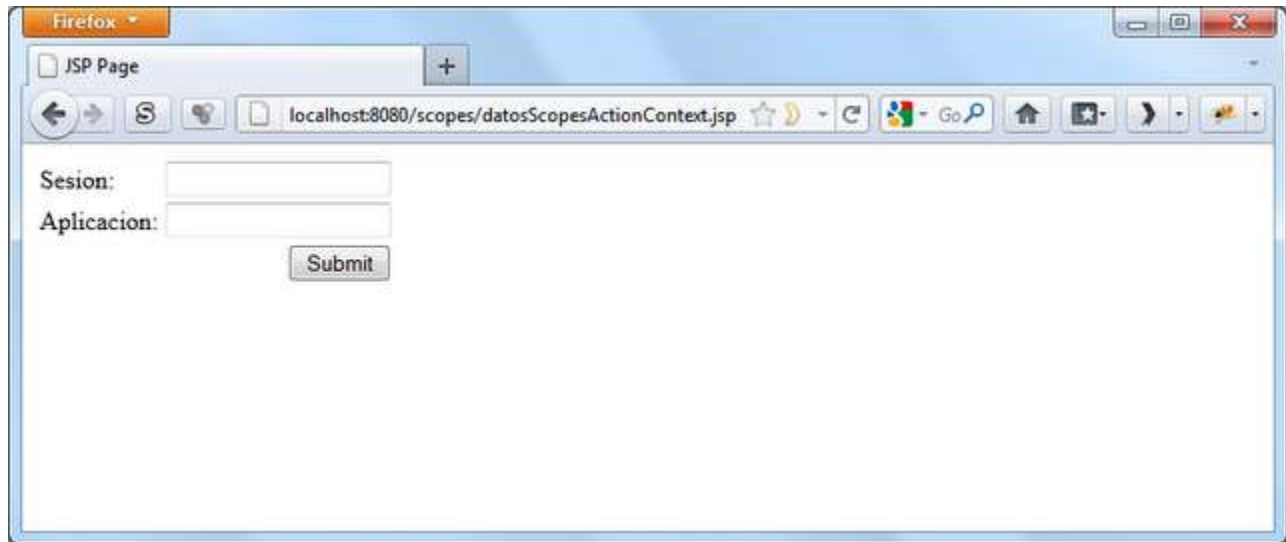
```

```
public void setDatosAplicacion(String datosAplicacion)
{
    this.datosAplicacion = datosAplicacion;
}
}
```

Ahora que ya tenemos todo listo ejecutamos la aplicación y al entrar a la siguiente dirección:

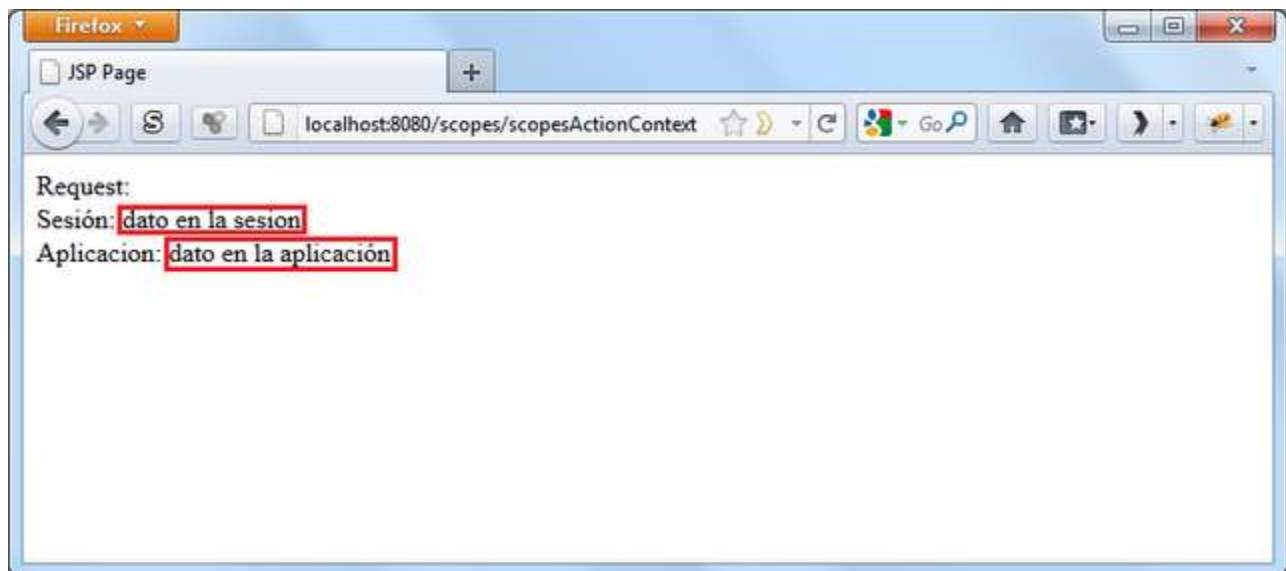
<http://localhost:8080/scopes/datosScopesActionContext.jsp>

Debemos ver el siguiente formulario, muy parecido al anterior:



The screenshot shows a Firefox browser window with a single tab titled 'JSP Page'. The address bar displays 'localhost:8080/scopes/datosScopesActionContext.jsp'. The page content includes two text input fields: the first is labeled 'Sesion:' and the second is labeled 'Aplicacion:'. Below these fields is a 'Submit' button.

Cuando ingresemos algunos valores y demos clic en el botón de enviar, veremos un resultado como el siguiente:



The screenshot shows the same Firefox browser window after a form submission. The page content now displays 'Request:' followed by two lines of text: 'Sesión: dato en la sesion' and 'Aplicacion: dato en la aplicación'. Both lines of text are enclosed in red rectangular boxes.

El dato del **request** queda vacío ya que no lo hemos establecido en esta ocasión.

Podemos ver que esta segunda forma de establecer los datos es también muy simple de utilizar y da buenos resultados.

Veremos la tercer y última forma de manejar los atributos de los scopes **request**, **session** y **application**. Debo decir que esta tercer forma no es recomendable ya que rompe con el esquema de trabajo de **Struts 2**, el cual oculta detalles de la especificación de **Servlets**, pero eso lo veremos en un momento.

Manejo de Atributos de Scope usando **ServletActionContext**

Con esta última forma tendremos acceso directo a los objetos "**HttpServletRequest**" y "**ServletContext**", de la especificación de **Servlets**; haciendo uso de otro objeto especial de **Struts 2**: "**ServletActionContext**".

Este objeto especial "**ServletActionContext**" contiene una serie de métodos estáticos que nos dan acceso a otros objetos de utilidad (entre ellos "**ActionContext**"). Los métodos que en este momento nos interesan son: "**getRequest**", con el que podemos obtener el objeto "**HttpServletRequest**" (desde el cual tenemos además acceso a la sesión) asociado con la petición que el **Action** está sirviendo, y "**getServletContext**", que nos regresa el objeto "**ServletContext**" que representa el contexto de la aplicación web.

El uso de estos dos métodos es muy directo y lo veremos en un momento, pero primero crearemos una nueva página **JSP** que contendrá el formulario, como los anteriores, que nos permitirá introducir los datos que se agregarán como atributos de cada uno de los scopes. Esta página se llamará "**datosScopesServletActionContext.jsp**" y contendrá un formulario como el primero que creamos con la diferencia de los datos de este formulario serán procesados por otro **Action**:

```
<s:form action="scopesServletActionContext">
<s:textfield name="datoSesion" label="Sesion" />
<s:textfield name="datoRequest" label="Request" />
<s:textfield name="datosAplicacion" label="Aplicacion" />
<s:submit />
</s:form>
```

Ahora creamos, en el paquete "**actions**", una nueva clase llamada "**ScopesServletActionContextAction**" que extienda de "**ActionSupport**":

```
public class ScopesServletActionContextAction extends ActionSupport
{
}
```

Esta clase tendrá tres atributos de tipo **String**, con sus correspondientes **setters**, uno para cada valor que recibimos del formulario:

```
public class ScopesServletActionContextAction extends ActionSupport
{
    private String datoSesion;
    private String datoRequest;
    private String datosAplicacion;

    public void setDatoRequest(String datoRequest)
    {
        this.datoRequest = datoRequest;
    }

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }
}
```

```

    public void setDatosAplicacion(String datosAplicacion)
    {
        this.datosAplicacion = datosAplicacion;
    }
}

```

Ahora sobre-escribiremos el método **"execute"** para, haciendo uso de los métodos estáticos de la clase **"ServletActionContext"**, obtener referencias al **"HttpServletRequest"** y al **"ServletContext"**. Como había dicho **"ServletActionContext"** nos proporciona métodos estáticos que nos regresan estas referencias de forma directa, por lo que lo único que tenemos que hacer es invocarlos:

```

HttpServletRequest request = ServletActionContext.getRequest();
ServletContext context = ServletActionContext.getServletContext();

```

La sesión (el objeto **"HttpSession"**) se obtiene usando el método **"getSession()"** del objeto **"request"**:

```

HttpSession session = request.getSession();

```

Y ya con estas referencias podemos obtener los atributos de los scopes correspondientes como lo hacemos cuando trabajamos directamente con **Servlets**: usando el método **"setAttribute"** de cada uno de los objetos:

```

request.setAttribute("datoRequest", datoRequest);
session.setAttribute("datoSesion", datoSesion);
context.setAttribute("datoAplicacion", datosAplicacion);

```

Y eso es todo lo que hay que hacer. El método **"execute"** queda de la siguiente forma:

```

@Override
public String execute() throws Exception
{
    HttpServletRequest request = ServletActionContext.getRequest();
    ServletContext context = ServletActionContext.getServletContext();
    HttpSession session = request.getSession();

    request.setAttribute("datoRequest", datoRequest);
    session.setAttribute("datoSesion", datoSesion);
    context.setAttribute("datoAplicacion", datosAplicacion);

    return SUCCESS;
}

```

Lo único que resta hacer es colocar las anotaciones para indicar que esta clase será un **Action**. Al igual que en los casos anteriores, reutilizaremos la página **"resultados.jsp"**. Como ya conocemos estas anotaciones de memoria, solo mostraré como queda finalmente la clase **"ScopesServletActionContextAction"**

```

@Namespace(value = "")
@Action(value = "scopesServletActionContext", results = {@Result(name = "success",
location = "/resultado.jsp")})
public class ScopesServletActionContextAction extends ActionSupport
{
    private String datoSesion;
    private String datoRequest;
    private String datosAplicacion;

    @Override
    public String execute() throws Exception

```



```

{
    HttpServletRequest request = ServletActionContext.getRequest();
    ServletContext context = ServletActionContext.getServletContext();
    HttpSession session = request.getSession();

    request.setAttribute("datoRequest", datoRequest);
    session.setAttribute("datoSesion", datoSesion);
    context.setAttribute("datoAplicacion", datosAplicacion);

return SUCCESS;
}

public void setDatoRequest(String datoRequest)
{
    this.datoRequest = datoRequest;
}

public void setDatoSesion(String datoSesion)
{
    this.datoSesion = datoSesion;
}

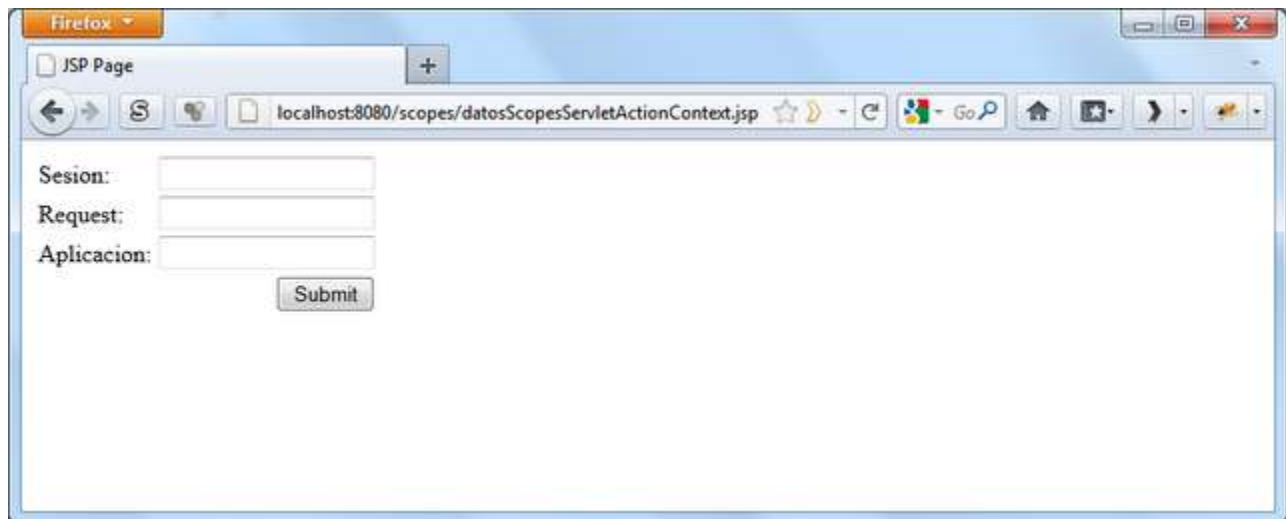
public void setDatosAplicacion(String datosAplicacion)
{
    this.datosAplicacion = datosAplicacion;
}
}

```

Cuando ejecutemos nuestra aplicación y entremos a la siguiente dirección:

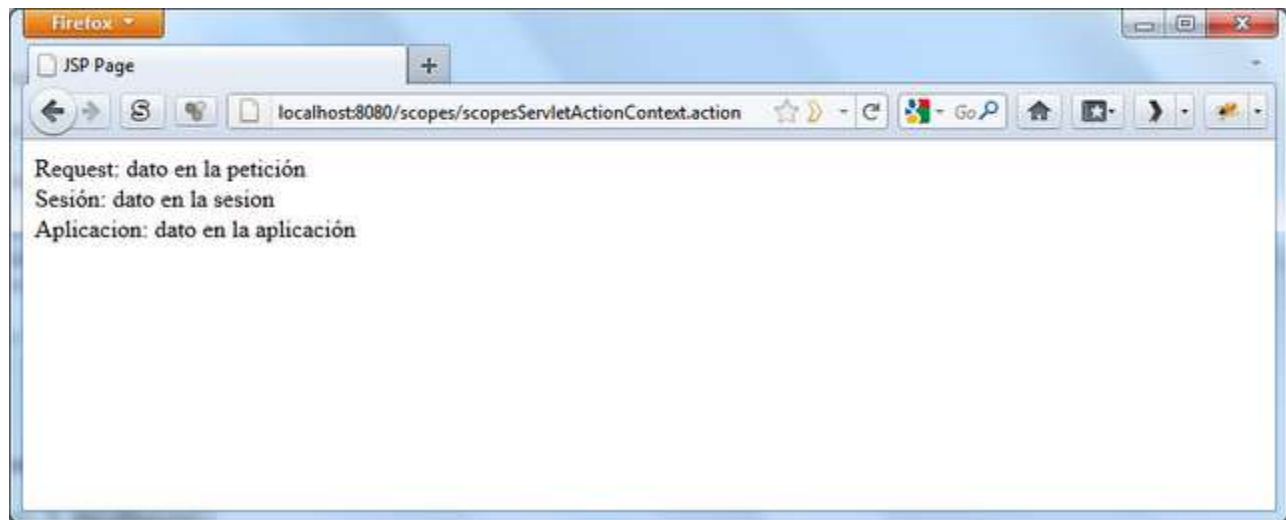
<http://localhost:8080/scopes/datosScopesServletActionContext.jsp>

Debemos ver el mismo formulario que en el primer ejemplo:



The screenshot shows a Firefox browser window displaying a JSP page titled "JSP Page". The address bar shows the URL "localhost:8080/scopes/datosScopesServletActionContext.jsp". The page contains a form with three input fields labeled "Sesion:", "Request:", and "Aplicacion:". Below these fields is a "Submit" button. The browser's toolbar includes navigation buttons (back, forward, stop, reload), a search bar with "Go" and a magnifying glass icon, and other standard browser controls.

Al rellenar los campos con algunos datos y enviar el formulario debemos ver el siguiente resultado:



Como podemos ver esta tercer forma abarca en realidad los dos temas del tutorial, establecer valores en los scopes de la aplicación web y obtener los objetos "**HttpServletRequest**" y "**ServletContext**", así que a continuación veremos la segunda forma de obtener estos objetos:

Obtención de Objetos "**HttpServletRequest**" y "**ServletContext**"

Obtención de Objetos de Servlet usando interfaces **Aware**

En la primer parte del tutorial vimos cómo trabajan las interfaces **Aware**, y en el ejemplo anterior vimos cómo tener acceso directo a los objetos de la especificación de **Servlets**. Así que en este último ejemplo veremos una forma de obtener estos objetos, usando las interfaces **Aware**.

Para no perder la costumbre de este tutorial, lo primero que haremos es crear una nueva **JSP** llamada "**datosServletsInterfaces.jsp**" que contendrá el formulario que también ya debemos conocer de memoria ^_^, en donde, nuevamente, lo único que cambiará es el valor del atributo "**action**":

```
<s:form action="datosInterfaces">
<s:textfield name="datoSesion" label="Sesion" />
<s:textfield name="datoRequest" label="Request" />
<s:textfield name="datosAplicacion" label="Aplicacion" />
<s:submit />
</s:form>
```

Ahora crearemos una nueva clase llamada "**ObjetosServletAction**", en el paquete "**actions**". Esta clase deberá extender "**ActionSupport**":

```
public class ObjetosServletAction extends ActionSupport
{
}
```

Struts 2 permite obtener una referencia al objeto "**HttpServletRequest**", que representa la petición actual que está siendo atendida por el **Action**, y al objeto "**ServletContext**", que representa el contexto de la aplicación web, implementando las interfaces "**ServletRequestAware**" y "**ServletContextAware**" respectivamente.

Ambas interfaces son igual de sencillas que sus contrapartes para los atributos de los scopes. La interface "**ServletContextAware**" luce de la siguiente forma:

```
interface ServletContextAware
{
    public void setServletContext(ServletContext sc);
}
```

Y la interface "**ServletRequestAware**" se ve así:

```
interface ServletRequestAware
{
    public void setServletRequest(HttpServletRequest hsr);
}
```

Como podemos ver, ambas interfaces tienen tan solo un método que debemos implementar, y que reciben el objeto adecuado perteneciente a la especificación de **Servlets** (en vez de algún objeto propio de **Struts 2** o algún objeto genérico, como estamos acostumbrados al trabajar con este framework), por lo que al implementar estos métodos ya tenemos una referencia directa a estos objetos sin tener que hacer ningún otro proceso, conversión, o invocación.

Junto con la implementación de estas interfaces proporcionaremos un par de atributos, uno de tipo "**ServletContext**" y uno de tipo "**HttpServletRequest**", para almacenar los valores recibidos con la implementación.

Así que lo primero que hacemos entonces es declarar que nuestra clase "**ObjetosServletAction**" implementará las interfaces "**ServletContextAware**" y "**ServletRequestAware**":

```
public class ObjetosServletAction extends ActionSupport implements ServletContextAware,
ServletRequestAware
{
}
```

Ahora colocaremos los atributos (sin **getters** ni **setters**) para almacenar las referencias recibidas por los métodos de las interfaces anteriores:

```
private ServletContext application;
private HttpServletRequest request;
```

Lo siguiente es proporcionar la implementación de las interfaces, en los cuales almacenaremos las referencias recibidas por estas, en los atributos que hemos colocado en nuestra clase:

```
public void setServletContext(ServletContext sc)
{
    this.application = sc;
}

public void setServletRequest(HttpServletRequest hsr)
{
    this.request = hsr;
}
```

Ahora que ya tenemos las referencias que nos interesaban agregaremos los tres atributos, junto con sus **setters**, que nos permitirán recibir los parámetros provenientes del formulario:

```

private String datoSesion;
private String datoRequest;
private String datosAplicacion;

public void setDatoRequest(String datoRequest)
{
    this.datoRequest = datoRequest;
}

public void setDatoSesion(String datoSesion)
{
    this.datoSesion = datoSesion;
}

public void setDatosAplicacion(String datosAplicacion)
{
    this.datosAplicacion = datosAplicacion;
}

```

El siguiente paso es sobre-escribir el método "**execute**" para establecer los valores de los atributos en cada uno de los scopes. Primero obtendremos el objeto "**HttpSession**" usando el "**HttpServletRequest**", como en el ejemplo anterior, para posteriormente poder establecer los atributos en los scopes usando los objetos apropiados:

```

@Override
public String execute() throws Exception
{
    HttpSession session = request.getSession();

    application.setAttribute("datoAplicacion", datosAplicacion);
    session.setAttribute("datoSesion", datoSesion);
    request.setAttribute("datoRequest", datoRequest);

    return SUCCESS;
}

```

Como ven, la lógica del método es tan sencilla como en todo el tutorial.

El último paso es anotar esta clase para indicarle a **Struts 2** que se trata de un **Action**. Una vez más reutilizaremos la página "**resultado.jsp**" para mostrar los resultados de la ejecución del **Action**:

```

@Namespace(value = "")


```

Al final la clase "**ObjetosServletAction**" queda de la siguiente forma:

```

@Namespace(value = "")
@Action(value = "datosInterfaces", results ={@Result(name = "success", location =
"/resultado.jsp")})
public class ObjetosServletAction extends ActionSupport implements ServletRequestAware,
ServletContextAware
{
    private ServletContext application;
    private HttpServletRequest request;
    private String datoSesion;
    private String datoRequest;
    private String datosAplicacion;

    @Override
    public String execute() throws Exception
    {
        HttpSession session = request.getSession();

        application.setAttribute("datoAplicacion", datosAplicacion);
        session.setAttribute("datoSesion", datoSesion);
        request.setAttribute("datoRequest", datoRequest);

        return SUCCESS;
    }

    public void setServletContext(ServletContext sc)
    {
        this.application = sc;
    }

    public void setServletRequest(HttpServletRequest hsr)
    {
        this.request = hsr;
    }

    public void setDatoRequest(String datoRequest)
    {
        this.datoRequest = datoRequest;
    }

    public void setDatoSesion(String datoSesion)
    {
        this.datoSesion = datoSesion;
    }

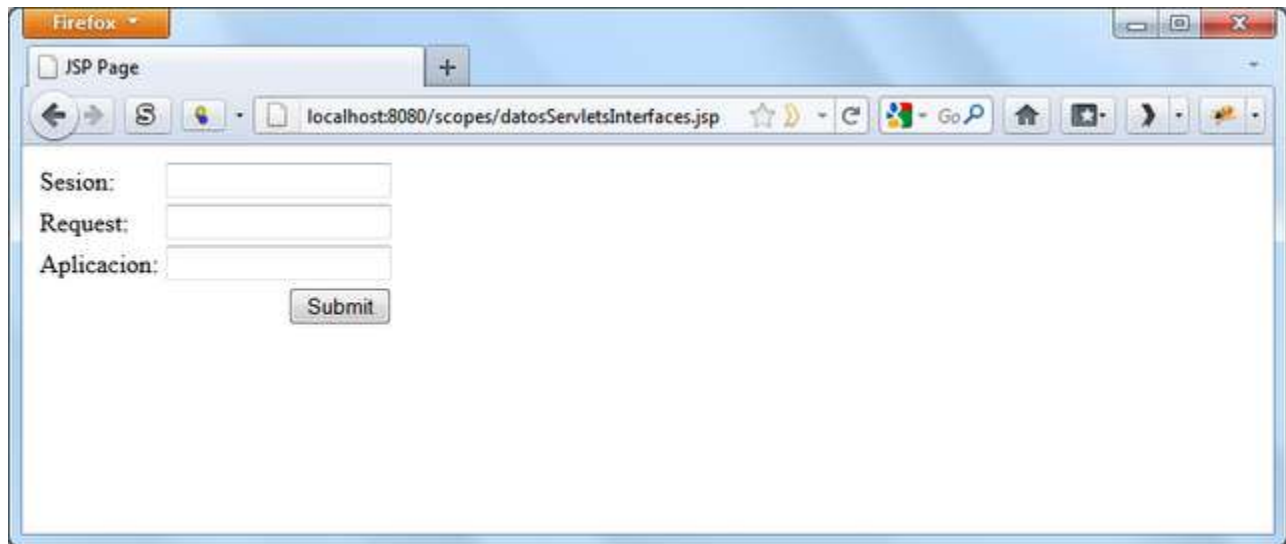
    public void setDatosAplicacion(String datosAplicacion)
    {
        this.datosAplicacion = datosAplicacion;
    }
}

```

Cuando ejecutemos la aplicación y entremos a la siguiente dirección:

<http://localhost:8080/scopes/datosServletsInterfaces.jsp>

Debemos ver el formulario al que ahora ya debemos estar acostumbrados ^_^!:



Firefox

JSP Page

localhost:8080/scopes/datosServletsInterfaces.jsp

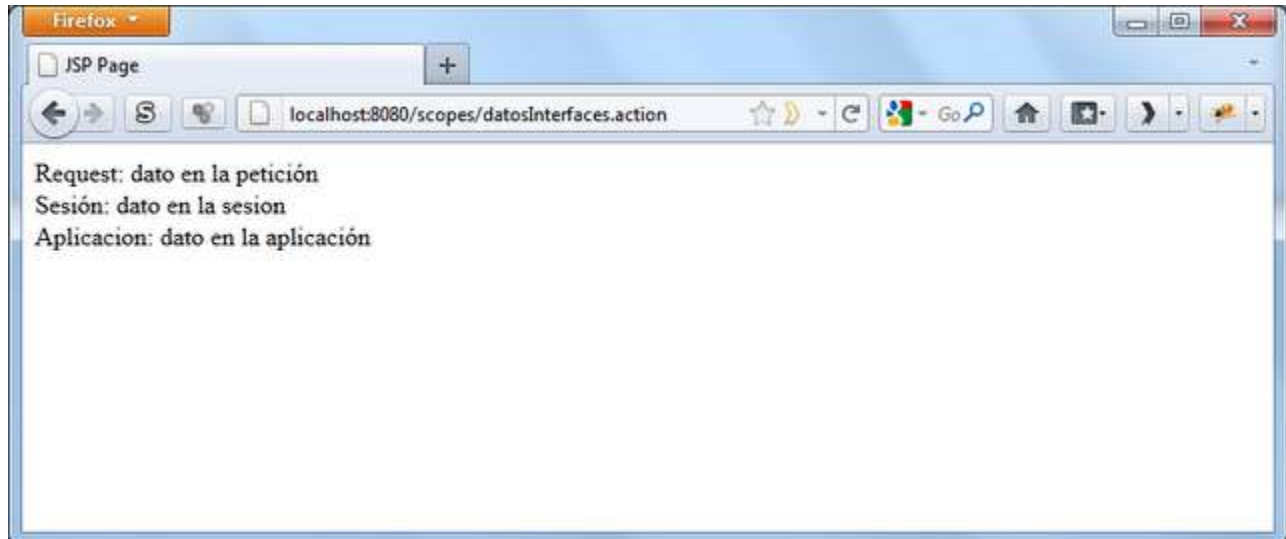
Sesion:

Request:

Aplicacion:

Submit

Y, nuevamente, al ingresar algunos datos y presionar el botón enviar debemos ver la siguiente salida:



Firefox

JSP Page

localhost:8080/scopes/datosInterfaces.action

Request: dato en la petición

Sesión: dato en la sesion

Aplicacion: dato en la aplicación

Como podemos ver, una vez más todo ha funcionado correctamente.

Algunos se preguntarán ¿para qué queremos tener acceso a estos objetos si **Struts 2** se encarga de manejar todos los aspectos relacionados con **Servlets** por nosotros? Bueno me alegra que hagan esa pregunta ^_^.

En realidad esto tiene varios usos, los dos más utilizados que se me ocurren en este momento es el poder obtener la ubicación absoluta de un recurso (un archivo o algún otro elemento) que se encuentra en nuestra aplicación y que por lo tanto no sabemos en qué directorio del disco duro está. Para esto se necesita el método "**getRealPath**" del objeto "**ServletContext**" de la siguiente forma:

```
application.getRealPath("/recurso.ext");
```

El segundo y más común uso es para invalidar una sesión. **Struts 2** no nos da directamente una manera para invalidar la sesión de un usuario de nuestra aplicación web, así que para esto es necesario tener objeto al objeto "**HttpSession**", y para esto al objeto "**HttpServletRequest**":

```
request.getSession().invalidate();
```

Así que, como podemos ver, algunas veces es útil tener acceso a los objetos de la especificación de **Servlets** cuando el framework que estamos usando no nos pueda, por la razón que sea, darnos todas las facilidades que necesitemos para hacer algunas cosas específicas.

Struts 2 - Parte 5: Tipos de Results

Cada vez [que](#) un **Action** termina su ejecución, se muestra un resultado al usuario. Estos resultados pueden ser de muchos tipos y tener muchos significados. El tipo más común de resultado es mostrar al usuario una nueva página web cierta información, pero ¿si quisiéramos hacer otra cosa? ¿Qué ocurre si queremos regresar un archivo [como](#) texto plano? ¿Y cuando queremos regresar un archivo binario? ¿Y si necesitamos redirigir la petición a otro **Action** o enviar al usuario a otra página?

Los casos anteriores también son muy comunes cuando desarrollamos una aplicación web.

Struts 2 ofrece una gran variedad de tipos de **Results** para manejar estas y otras situaciones y que aprenderemos cómo usar en este tutorial.

Primero lo primero, aunque los hemos estado usando a lo largo de esta serie de tutoriales, nunca hemos definido de una forma clara qué es un **Result**. Cuando el método **"execute"** de un **Action** termina de ejecutarse este siempre regresa un objeto de tipo **String**. El valor de ese **String** se usa para seleccionar un elemento de tipo **"<result>"**, si trabajamos con archivos de configuración en **XML**, o **"@Result"**, si trabajamos con anotaciones. Dentro del elemento se indica qué tipo de **Result** se quiere regresar al usuario (enviar un recurso, redirigirlo a otro **Action**, un archivo de texto plano, etc.), el recurso que se quiere regresar, algunos parámetros adicionales que requiera el **Result** y, lo más importante, **el nombre** que tendrá este **Result**, en donde este nombre es precisamente lo que se indica con la cadena que es regresada de la ejecución del método **"execute"**.

Cada result tiene un nombre que **debe ser único** para un **Action** y es en base a este nombre que **Struts 2** sabrá qué es lo que debe mostrarle al usuario. La interface **"Action"**, que es implementada [por](#) la clase **"ActionSupport"**, que hemos estado utilizando como base para nuestros **Actions**, proporciona un conjunto de constantes que contienen los nombres de los results más comunes para que podamos usarlos dentro de nuestros **Actions**. El identificador de la constante es el mismo que el valor que contiene (usado las respectivas convenciones) para que sea fácil identificar cuándo debemos usar cada uno:

```
String SUCCESS = "success";
String NONE    = "none";
String ERROR   = "error";
String INPUT   = "input";
String LOGIN   = "login";
```

Claro que no es obligatorio que regresemos alguna de estas constantes en el método **"execute"**, podemos usar cualquier cadena que queramos, siempre y cuando exista un **"<result>"** con ese nombre para nuestro **Action**. Por ejemplo en caso de algún valor incorrecto introducido por el usuario podríamos regresar la cadena **"fail"** ^^; o en caso de que dependiendo del perfil que tenga el usuario que ingresa a la aplicación podríamos redirigirlo a una página apropiada para ese perfil regresando el nombre del perfil, por ejemplo **"operador"**, **"administrador"**, **"auditor"**, etc.

Hasta ahora solo hemos utilizado los valores **"SUCCESS"** e **"INPUT"** (cuando trabajamos con formularios) en nuestros results, pero el valor de cada una de las constantes anteriores representa una situación que puede ocurrir durante la ejecución de nuestro **Action**:

- **"success"** - Indica que [todo](#) ha salido correctamente (validaciones, y el proceso de lógica en general) durante la ejecución del método, por lo que el usuario puede ver la salida esperada.
- **"error"** - Indica que alguna cosa dentro del método ha salido mal. Puede ser que algún cálculo no pueda realizarse, o que algún servicio externo que estemos utilizando no esté disponible en ese momento o haya lanzado alguna excepción, o cualquier otra cosa que se les ocurra que pueda salir mal.

- **"input"** - Indica que algún campo proporcionado en un formulario es incorrecto. Puede ser que el valor no sea del tipo esperado, o no cumpla con el formato o rango requerido, o cosas más sofisticadas como que un valor esté repetido (digamos que alguien está tratando de usar un nombre de usuario que ya está siendo usado por otra persona).
- **"login"** - Indica que el recurso al que el usuario está intentado acceder solo está disponible para usuarios registrados del sitio y por lo tanto debe loguearse primero. Para poder usar este **result** de forma correcta debemos crear un result global en la configuración del sitio. Veremos cómo hacer esto al final del tutorial.
- **"none"** - Este es un **result** de un tipo especial ya que le indica a **Struts 2** que no debe enviar al usuario a ningún lugar (o en términos formales que el procesamiento del resultado sea cancelado). Esto es usado cuando el **Action** maneja el proceso de regresar el resultado al usuario usando, por ejemplo, de forma directa los objetos **"HttpServletResponse"** o **"ServletOutputStream"**.

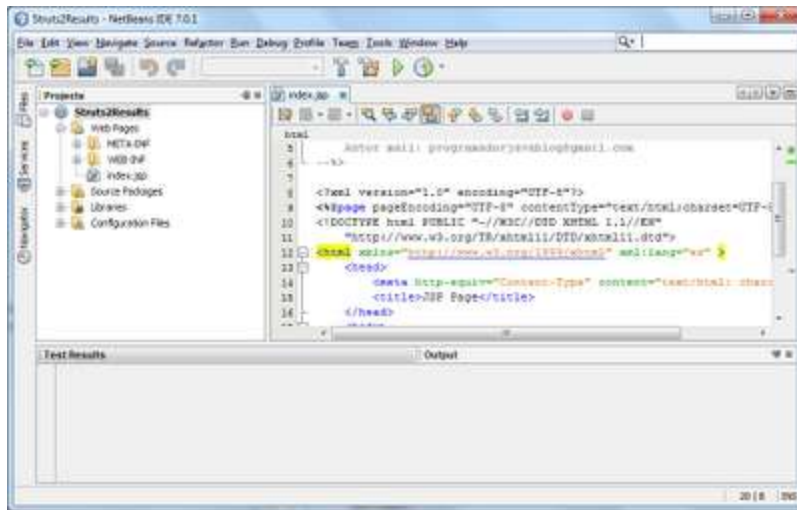
Como podrán imaginar un **Action** puede tener más de un **result** y estos pueden ser de distintos tipos, uno para cada situación que necesite manejar (siempre y cuando cada uno tenga un nombre único para ese **Action**).

Struts 2 maneja por default **11 tipos de results**, aunque los plugins (como por ejemplo el de jasperreports) pueden definir sus propios tipos. Los results que por default maneja **Struts 2** son:

- **"dispatcher"** - Este es el result más usado y el default. Envía como resultado una nueva vista, usualmente una **jsp**.
- **"redirect"** - Le indica al navegador que debe redirigirse a una nueva página, que puede estar en nuestra misma aplicación o en algún sitio externo, y por lo tanto este creará una nueva petición para ese recurso.
- **"redirectAction"** - Redirige la petición a otro **Action** de nuestra aplicación. En este caso se crea una nueva petición hacia el nuevo **Action**.
- **"chain"** - Al terminarse la ejecución del **Action** se invoca otro **Action**. En este caso se usa la misma petición para el segundo **Action**, el cual se ejecuta de forma completa, con todo su stack de interceptores y sus **results** (podemos crear cadenas de cuantos **Actions** queramos).
- **"stream"** - Permite enviar un archivo binario de vuelta al usuario.
- **"plaintext"** - Envía el contenido del recurso que indiquemos como un texto plano. Típicamente se usa cuando necesitamos mostrar una **JSP** o **HTML** sin procesar
- **"httpheader"** - Permite establecer el valor de la cabecera **HTTP** de código de estatus que se regresará al cliente. Así por ejemplo podemos usarlo para enviar un error al cliente (estatus **500**), un recurso no encontrado (**404**), un recurso al que no tiene acceso (**401**), o por el que requiere pagar (**402**).
- **"xslt"** - Se usa cuando el resultado generará un **XML** será procesado por una hoja de transformación de estilos para generar la vista que se mostrará al cliente.
- **"freemarker"** - Para integración con **FreeMarker**
- **"velocity"** - Para integración con **Velocity**
- **"tiles"** - Para integración con **Tiles**

En este tutorial veremos cómo funcionan los **primeros 7** tipos de **results** (desde **"dispatcher"** hasta **"httpheader"**).

Suficiente teoría, comencemos con la práctica. Comencemos creando un nuevo proyecto web en **NetBeans**, vamos al Menú **"File->New Project..."**. En la ventana que se abre seleccionamos la categoría **"Java Web"** y en el tipo de proyecto **"Web Application"**. Le damos una ubicación y un nombre al proyecto, en mi caso será **"Struts2Results"**. Presionamos el botón **"Next"**. En la siguiente pantalla deberemos configurar el servidor de aplicaciones que usaremos. Yo usaré la versión 7 de **Tomcat**. Como versión de **Java EE** usaré la 5 (pueden usar también la 6 si quieren, solo que en ese caso **NetBeans** no genera el archivo **"web.xml"** por default, y tendrán que crearlo a mano o usando el wizard correspondiente). De la misma forma, si quieren pueden modificar el **context path** de la aplicación. Presionamos el botón **"Finish"**, con lo que veremos aparecer la página **"index.jsp"** en nuestro editor:



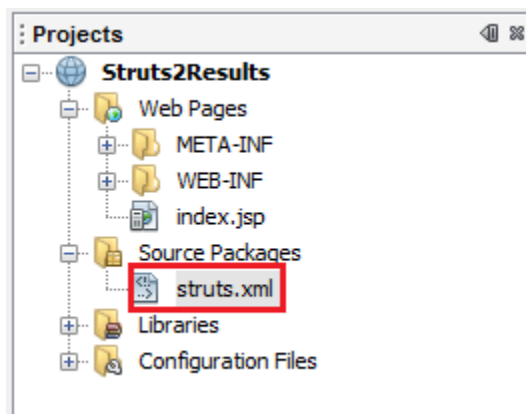
Una vez creado nuestro proyecto debemos agregar la biblioteca **"Struts2"** que creamos en [el primer tutorial de la serie](#). Para esto hacemos clic derecho en el nodo **"Libraries"** del panel de proyectos. En el menú que aparece seleccionamos la opción **"Add Library..."**. En la ventana que aparece seleccionamos la biblioteca **"Struts2"** y presionamos **"Add Library"**. Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto.

A continuación configuramos el filtro **"struts2"** en el **deployment descriptor**. Abrimos el archivo **"web.xml"** y colocamos el siguiente contenido, como se explicó en [el primer tutorial de la serie](#):

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Ahora crearemos un nuevo archivo llamado **"struts.xml"** en el paquete default de código fuente de nuestra aplicación:



Este archivo contendrá la configuración de **Struts 2** de nuestra aplicación. Colocaremos la configuración inicial, con un par de contantes para indicar que nos encontramos en modo de desarrollo y que cualquier cambio que hagamos en el archivo de configuración debe recargarse de inmediato, y un paquete inicial:

```

<?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.devMode" value="true" />
<constant name="struts.configuration.xml.reload" value="true" />

<package name="struts-results" extends="struts-default">
<action name="index">
<result>/index.jsp</result>
</action>
</package>
</struts>

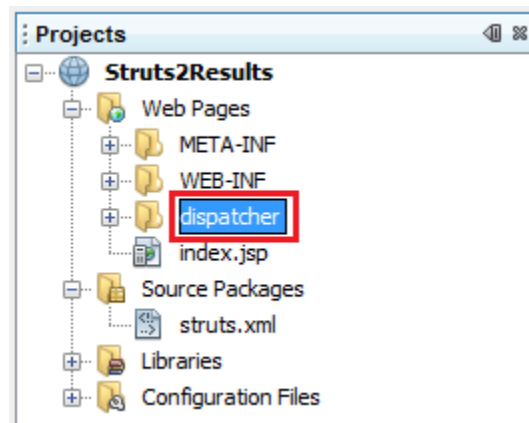
```

Creamos un nuevo paquete en el nodo "**Source Packages**" llamado "**com.javatutoriales.results.actions**" que será donde colocaremos las clases **Action** de nuestra aplicación.

Ya con la configuración inicial comencemos viendo el primer tipo de **result**:

Result "dispatcher"

Lo primero que haremos será crear un directorio llamado "**dispatcher**" en el nodo de "**Web Pages**" del proyecto:



En este nuevo directorio crearemos una nueva **JSP** llamada "**index.jsp**" que contendrá un sencillo formulario con solo dos campos, un campo de texto para ingresar nuestro **nombre**, y un combo-box, lista desplegable, select o cómo quiere que llamen, para seleccionar **sulenguaje** de programación favorito ;). Este formulario será procesado por un **Action** llamado "**dispatcher**". No olviden que debemos indicar que haremos uso de las etiquetas de **Struts 2** con el taglib correspondiente:

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

El formulario queda de la siguiente forma:

```

<s:form action="dispatcher">
<s:textfield name="nombre" label="Nombre" />
<s:select name="lenguaje" label="Lenguaje de Programacion" list="{ 'Java', 'PHP', '.Net' }"
/>
<s:submit value="Enviar" />
</s:form>

```

Ahora crearemos, en nuestro paquete **"actions"** una nueva clase **Java** que extenderá de **"ActionSupport"**, el nombre de esta clase será **"DispatcherResultAction"**:

```

public class DispatcherResultAction extends ActionSupport
{

}

```

En esta clase colocaremos un par de atributos de tipo **String**, con sus **setters** y **getters**, para recibir los parámetros del formulario que acabamos de crear:

```

private String nombre;
    private String lenguaje;

    public String getLenguaje()
    {
        return lenguaje;
    }

    public void setLenguaje(String lenguaje)
    {
        this.lenguaje = lenguaje;
    }

    public String getNombre()
    {
        return nombre;
    }

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

```

Ahora sobre-escribiremos el método **"execute"** para realizar una simple validación: en el caso de que el lenguaje de programación seleccionado por el usuario no sea **Java**, vamos a enviar al usuario a una página de error **^_^**:

```

@Override
public String execute() throws Exception
{
    if(!"Java".equals(lenguaje))
    {
        return ERROR;
    }

    return SUCCESS;
}

```

Como podemos ver en el código anterior, estamos haciendo uso de la constantes definidas en el interface "**Action**", la cual es implementada por la clase "**ActionSupport**" para hacer referencia a los nombres de los **results**.

Ahora crearemos dos nuevas **JSPs** en el directorio "**dispatcher**" una para mostrar el resultado en caso de que todo haya salido bien y una para mostrar el mensaje de error. La primer **JSP** que crearemos se llamará "**exito.jsp**" y como pueden imaginarse es la que mostrará los resultados en caso de que todo sea correcto.

En esta página primero indicamos que haremos uso de las etiquetas de **Struts 2**, con el mismo taglib que usamos hace un momento, además colocaremos como único contenido el siguiente mensaje:

```
Bienvenido <s:property value="nombre" /> al mundo de la programación Java.
```

Donde estamos usando la etiqueta "**<s:property>**" para mostrar el nombre del usuario que ha quedado almacenado en la propiedad "**nombre**" de nuestro **Action**, y que a su vez se encuentra en el "**ValueStack**" (pueden encontrar una explicación más clara en [el segundo tutorial de la serie](#)).

Ahora vamos a la página "**error.jsp**", en ella también indicaremos que haremos uso de la biblioteca de etiquetas de **Struts 2**, y colocaremos el siguiente mensaje:

```
Lo siento <s:property value="name" />, este es un sitio exclusivo para programadores  
Java, pide perdón y retírate u_u
```

Así de directo ^_^

Ahora que tenemos nuestras dos páginas vamos a configurar ambas en los **results** para nuestro **Action**, lo haremos con archivos de mapeo en **XML**, para los que trabajen con anotaciones, las opciones son las mismas.

En el archivo "**struts.xml**" agregamos un elemento "**<action>**" que tendrá como nombre "**dispatcher**" y que esté implementado por la clase "**com.javatutoriales.results.actions. DispatcherResultAction**":

```
<action name="dispatcher"  
class="com.javatutoriales.results.actions.DispatcherResultAction">  
</action>
```

Dentro de este elemento "**<action>**" definiremos dos elementos "**result**" de tipo "**dispatcher**", uno para cada el resultado "**success**" y otro para el "**error**":

```
<result name="success" type="dispatcher">  
</result>  
<result name="error" type="dispatcher">  
</result>
```

Cada uno de los tipos de results pueden recibir varios parámetros. Estos parámetros son indicados usando el elemento "**<param>**", para cuál parámetro queremos establecer se usa el atributo "**name**" de este elemento. El valor del parámetro se coloca como contenido de este elemento. Por ejemplo para establecer un parámetro llamado, por ejemplo, "**tiempo**" a "**15**" segundos lo haríamos de la siguiente forma:

```
<param name="tiempo">15</param>
```

En el caso de los results de tipo "**dispatcher**" pueden recibir dos parámetros: "**location**" y "**parse**". "**location**" indica dónde se encuentra el recurso que se enviará al usuario cuando se termine la ejecución del **Action** (usualmente una **JSP**). No se preocupen por el otro atributo porque casi no se usa (aún la explicación de lo que hace es un poco complicada ^_^).

Así que indicaremos, usando el parámetro "**location**" cuáles son las **JSPs** que deben regresar en cada caso:

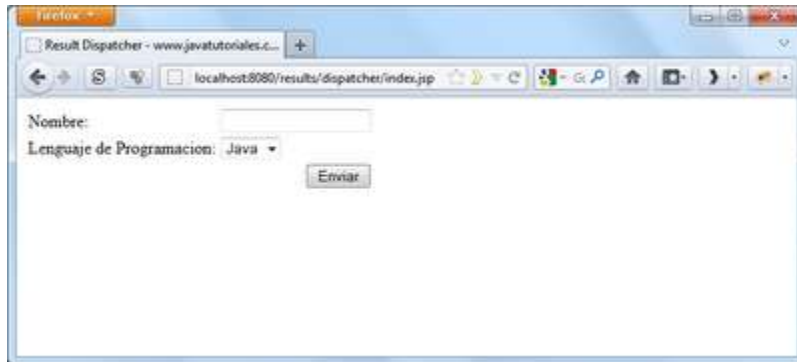
```
<result name="success" type="dispatcher">
<param name="location">/dispatcher/exito.jsp</param>
</result>
```

```
<result name="error" type="dispatcher">
<param name="location">/dispatcher/error.jsp</param>
</result>
```

Si ahora ejecutamos nuestra aplicación y entramos en la siguiente dirección:

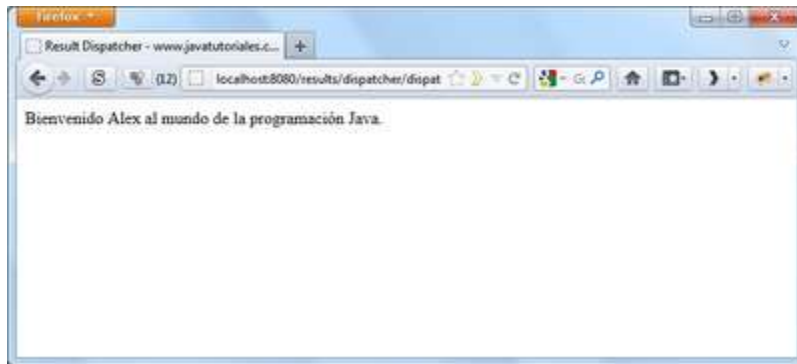
<http://localhost:8080/results/dispatcher/index.jsp>

Debemos ver el siguiente formulario:



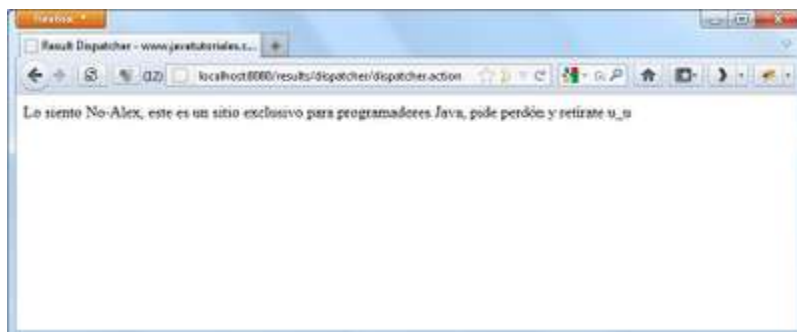
A screenshot of a web browser window titled "Result Dispatcher - www.javatutoriales.com". The address bar shows "localhost:8080/results/dispatcher/index.jsp". The page contains a form with two input fields: "Nombre:" and "Lenguaje de Programacion:". The "Lenguaje de Programacion:" field has a dropdown menu with "Java" selected. Below the fields is a button labeled "Enviar".

Al colocar nuestro nombre, seleccionar **"Java"** como nuestro lenguaje de programación y enviar nuestro formulario, veremos la siguiente salida:



A screenshot of a web browser window titled "Result Dispatcher - www.javatutoriales.com". The address bar shows "localhost:8080/results/dispatcher/dispatcher.action". The page displays a welcome message: "Bienvenido Alex al mundo de la programación Java."

Si seleccionamos otro lenguaje de programación veremos la siguiente salida:



A screenshot of a web browser window titled "Result Dispatcher - www.javatutoriales.com". The address bar shows "localhost:8080/results/dispatcher/dispatcher.action". The page displays a message: "Lo siento No-Alex, este es un sitio exclusivo para programadores Java, pide perdón y retírate u_u".

Como podemos ver, la validación se realiza de forma correcta y nos envía al result adecuado en cada uno de los casos que seleccionamos.

Ahora regresemos a la configuración de los results para explicar algunas cosas interesantes.

En [el primer tutorial de la serie](#) dijimos que una de las ventajas que nos da **Struts 2** es que nos proporciona muchos valores por default que son bastante útiles. El caso de los **Actions** es un ejemplo perfecto de los valores por default. Si regresamos a la definición básica de los results en **XML** veremos que tenemos lo siguiente:

```
<result name="success" type="dispatcher">
<param name="location">/dispatcher/exito.jsp</param>
</result>
```

```
<result name="error" type="dispatcher">
<param name="location">/dispatcher/error.jsp</param>
</result>
```

Podemos ver que en realidad la configuración es muy corta y muy sencilla; pero imaginen que en vez de dos results, necesitaríamos 10, y lo mismo para cada uno de los **Actions** de nuestra aplicación... ya no suena tan bien ¿cierto?

Bueno, el enunciado anterior puede ser un poco exagerado, pero sirve para ilustrar el deseo de querer simplificar un poco las cosas.

Lo primero que hay que saber es que cada uno de los tipos de **results** tienen siempre un parámetro por default. Si solo vamos a utilizar un parámetro en nuestro **result**, y ese parámetro es el parámetro por default entonces podemos omitir el elemento "**<param>**" y colocar directamente el valor dentro del elemento "**<result>**". En el caso del **result** de tipo "**dispatcher**" el parámetro por default es "**location**", por lo que podemos simplificar los dos elementos anteriores de la siguiente forma:

```
<result name="success" type="dispatcher">/dispatcher/exito.jsp</result>
<result name="error" type="dispatcher">/dispatcher/error.jsp</result>
```

Podemos ver que con esto la configuración ya ha quedado mucho más sencilla.

Otro valor por default útil que tenemos es el caso del atributo "**type**" del elemento "**<result>**". Por default el tipo de **result** es "**dispatcher**", por lo que en las definiciones anteriores podemos omitir este atributo. Esto es útil ya que en nuestras aplicaciones, en la mayoría de los casos nuestros **results** serán de tipo "**dispatcher**". Por lo tanto las definiciones anteriores quedan de la siguiente forma:

```
<result name="success">/dispatcher/exito.jsp</result>
<result name="error">/dispatcher/error.jsp</result>
```

Podemos ver que cada vez queda más corta la declaración de nuestros **results** ^_^.

Para terminar eliminaremos un valor más. También el atributo "**name**" tiene un valor por default: "**success**". Esto quiere decir que si vamos a definir el **result** para el caso "**success**", podemos poner un **result** sin nombre (solo uno por **Action**). Si nuestro **result** tiene otro nombre, si debemos colocarlo **de forma explícita**. Con este nuevo cambio la definición anterior queda de la siguiente forma:

```
<result>/dispatcher/exito.jsp</result>
<result name="error">/dispatcher/error.jsp</result>
```

Como podemos ver, gracias a los valores por default proporcionados por **Struts 2**, pasamos de esto:

```
<result name="success" type="dispatcher">
<param name="location">/dispatcher/exito.jsp</param>
</result>
```

```
<result name="error" type="dispatcher">
<param name="location">/dispatcher/error.jsp</param>
</result>
```

A esto:

```
<result>/dispatcher/exito.jsp</result>
<result name="error">/dispatcher/error.jsp</result>
```

Con lo que no solo nos ahorramos unas cuantas líneas de código, sino que las definiciones son más fáciles de leer.

La definición del **Action** queda de la siguiente forma:

```
<action name="dispatcher"
class="com.javatutoriales.results.actions.DispatcherResultAction">
<result>/dispatcher/exito.jsp</result>
<result name="error">/dispatcher/error.jsp</result>
</action>
```

Si volvemos a ejecutar nuestra aplicación debemos ver exactamente la misma salida. Podemos ver con esto que los valores por default proporcionados en todos los elementos de **Struts 2** pueden ahorrarnos muchas líneas de código. En cada uno de los **results** les diré cuál es el parámetro por default.

Una última cosa importante que hay que aprender desde el inicio acerca de los **results** es que algunos parámetros (por lo regular los más importantes) se pueden establecer de forma dinámica usando una expresión en **OGNL**.

Vemos un ejemplo para ver cómo podemos indicar cuál será la ubicación a la que debe enviarse al usuario en caso de un error. Para esto debemos indicar en el archivo "**struts.xml**" que este parámetro será recibido usando una expresión, un parámetro llamado "**ubicacion**":

```
<result name="error">${ubicacion}</result>
```

Ahora modificaremos un poco nuestra clase "**DispatcherResultAction**". Primero tendremos que agregar un **getter** para una propiedad llamada "**ubicacion**". **Struts 2** usará este **getter** cuando busque el valor para la expresión "**\${ubicacion}**". Este **getter** debe regresar un **String** que indicará, en este caso, la ubicación a la que se enviará al usuario. El valor que regrese el método puede ser una cadena estática o generada de forma dinámica. En este caso regresaremos la ubicación de manera estática:

```
public String getUbicacion()
{
    return "/dispatcher/error.jsp";
}
```

Al ejecutar nuevamente nuestra aplicación y seleccionemos un lenguaje que no sea **Java**, deberemos volver a ver nuestra página de error normal.

Casi todos los parámetros de los **results** pueden ser establecidos de forma dinámica usando expresiones (con excepción de los que indican si los parámetros pueden aceptar expresiones ^^, los parámetros llamados "**parse**"), esto nos da una gran ventaja para que nuestra aplicación sea realmente dinámica.

Ahora veamos el siguiente tipo de **result**:

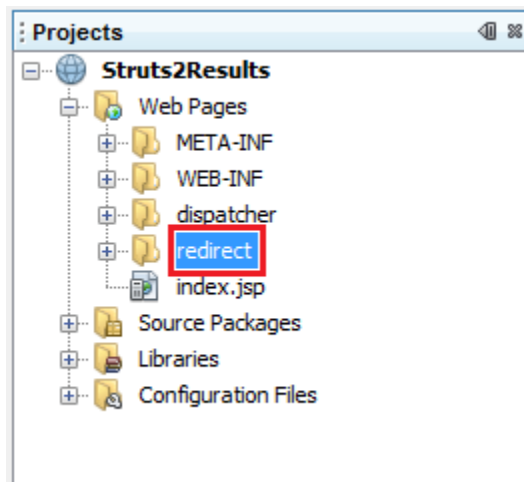
Result "redirect"

Este **result** le indica al navegador que debe dirigirse hacia otro recurso, de nuestra aplicación o algún lugar externo a nuestro sitio. Esto termina con la ejecución de la petición actual del cliente. Muy útil cuando queremos enviar al usuario a otro sitio como Google o algún otro servicio de nuestra empresa.

Este **result** tiene tres parámetros:

- **location** (default): indica la ubicación a donde irá el resultado (puede ser una página dentro de nuestro sitio o afuera de él).
- **parse**: indica si el parámetro "**location**" (el anterior) puede ser obtenido a través de una expresión en **OGNL**, logrando que la ubicación se pueda establecer de forma dinámica. Por default es verdadero.
- **anchor**: se puede especificar un ancla para el resultado (un ancla es un lugar en una página html al que se le ha dado un nombre especial, único en el documento y que nos permite enviar al navegador del usuario a ese punto específico). Este parámetro es opcional. Sobre este parámetro, aunque en la documentación oficial no lo dice, parece que sólo funciona cuando redireccionamos a otra parte de nuestro mismo sitio.

Para ver un ejemplo de este **result** en acción primero agregaremos un directorio llamado "**redirect**" en el nodo "**Web Pages**":



Dentro de este directorio crearemos una nueva **JSP** llamada "**index**". Como en este caso lo único que interesa es llamar al **Action** que regresará el **result** de tipo "**redirect**", que crearemos en un momento. Lo único que colocaremos en esta página será una liga, usando la etiqueta "**<s:a>**" para invocar directamente un **Action** llamado "**redirect**":

```
<s:a action="redirect">Redireccionar</s:a>
```

Ahora crearemos, en el paquete "**actions**" una nueva clase llamada "**RedirectResultAction**" que extienda de "**ActionSupport**":

```
public class RedirectResultAction extends ActionSupport
{
}
```

Como en este caso no recibimos parámetros o algo por el estilo, lo único que necesitaremos hacer es sobre-escribir el método "**execute**" del **Action** para que regrese el valor "**SUCCESS**":


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

Lo dejaremos solamente así ya que no nos interesa, en este caso, realizar ningún tipo de procesamiento para la entrada.

La clase "**RedirectResultAction**" completa queda de la siguiente forma:

```
public class RedirectResultAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        return SUCCESS;
    }
}
```

Ahora, en el archivo "**struts.xml**" agregaremos la siguiente definición para nuestro **Action**:

```
<action name="redirect" class="com.javatutoriales.results.actions.RedirectResultAction">
</action>
```

Agregaremos un único **result**, que será de tipo "**redirect**". Como el parámetro por default para este **result** es "**location**" (que indica a dónde será redirigido el usuario), podemos indicarlo como valor del elemento "**<result>**"; en este caso redirigiremos al usuario a la primer página que creamos, el "**index.jsp**" del directorio "**dispatcher**":

```
<result type="redirect">/dispatcher/index.jsp</result>
```

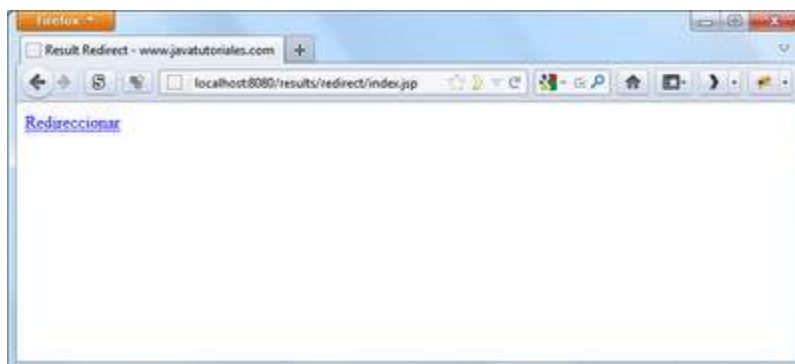
Recuerden que cuando no colocamos un nombre al **result** por default el result se llama "**success**". La configuración de nuestro **Action** queda de la siguiente forma:

```
<action name="redirect" class="com.javatutoriales.results.actions.RedirectResultAction">
<result type="redirect">/dispatcher/index.jsp</result>
</action>
```

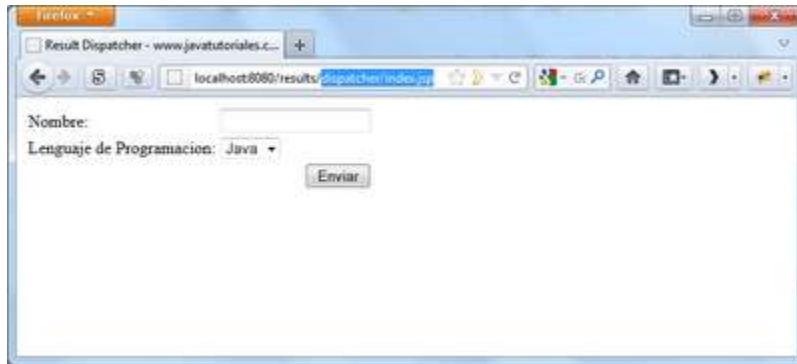
Nuestro ejemplo ya está listo, así que ahora ejecutamos nuestra aplicación y entramos en la siguiente dirección:

<http://localhost:8080/results/redirect/index.jsp>

Deberemos ver la siguiente página:



En donde lo único que hay es la liga que nos enviará al **Action "redirec"**. Al hacer clic en esa liga seremos enviados al **Action** correspondiente, el cual nos enviará a su vez a la página **"/dispatcher/index.jsp"**, por lo que terminaremos viendo el primer formulario:



Ahora veremos cómo redireccionar a una página que no se encuentre en nuestro sitio. Para esto crearemos una nueva clase, en el paquete **"actions"**, llamada **"RedirectExternoResultAction"** que, como podrán imaginarse, extenderá de **"ActionSupport"**:

```
public class RedirectExternoResultAction extends ActionSupport
{
}
```

Como esta clase tampoco hará nada especial, solo sobre-escribiremos su método **"execute"** para que regresa un valor de **"success"**, usando la constante apropiada:

```
public class RedirectExternoResultAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        return SUCCESS;
    }
}
```

Ahora iremos al archivo de configuración **"struts.xml"** y declaramos un nuevo elemento **"<action>"**, cuyo nombre sea **"redirect-externo"** y la clase que lo implemente sea nuestra clase **"RedirectExternoResultAction"**:

```
<action name="redirect-externo"
class="com.javatutoriales.results.actions.RedirectExternoResultAction">
</action>
```

A continuación agregaremos un elemento **"<result>"**, de tipo **"redirect"**, en cuyo valor colocaremos cualquier página de un sitio externo, yo elegiré una al azar que será **"<http://www.javatutoriales.com/2011/10/struts-2-parte-3-trabajo-con.html>"**, así que el result quedará definido de la siguiente forma:

```
<result type="redirect">http://www.javatutoriales.com/2011/10/struts-2-parte-3-trabajo-con.html</result>
```

Y el mapeo de nuestro **Action** de la siguiente forma:

```
<action name="redirect-externo"
class="com.javatutoriales.results.actions.RedirectExternoResultAction">
<result type="redirect">http://www.javatutoriales.com/2011/10/struts-2-parte-3-trabajo-con.html</result>
</action>
```

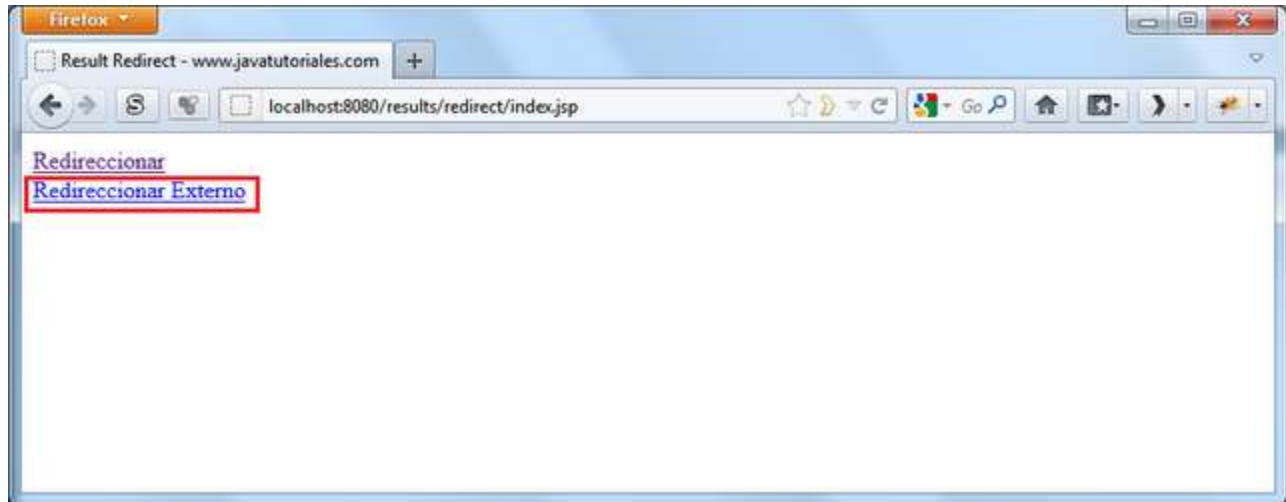
Agregaremos una liga a este **Action** en la página "**index.jsp**" del directorio "**redirect**" para poder invocar a este **Action**:

```
<s:a action="redirect-externo">Redireccionar Externo</s:a>
```

Ahora ejecutamos la aplicación y entramos en la siguiente dirección:

<http://localhost:8080/results/redirect/index.jsp>

En la que veremos los dos enlaces a los **Actions** de redirección:



Hacemos clic en el enlace que acabamos de agregar, y en ese momento deberemos ser re-dirigidos a la página que hayamos indicado:



Si están viendo la página que indicaron en el elemento "**<result>**" quiere decir que todo ha funcionado correctamente ^_^.

Ahora veremos cómo funciona el siguiente tipo de **result**: "**redirectAction**"

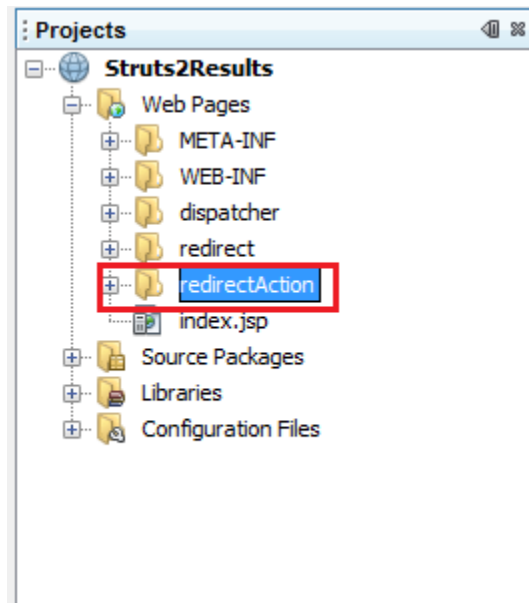
Result "redirectAction"

Este tipo de **result** lo podemos ver como una forma especializada del **result** anterior, en donde el navegador del usuario es redirigido y ejecuta el **Action** que le especifiquemos. Nos permite especificar qué parámetros queremos enviar a este **Action**, desde el archivo de configuración, los cuales pueden ser parámetros estáticos o dinámicos (en realidad podemos hacer esto con todos los **results**, pero este es el que me parece más claro para mostrar esta característica).

Este **result** tiene tres parámetros:

- **actionName** (default): El nombre del **Action** al que seremos redirigidos.
- **namespace**: El namespace al que pertenece el **Action**. Si no lo indicamos, se usa por default el namespace actual.
- **supressEmptyParameters**: Evita que los parámetros que estén vacíos sean enviados al siguiente **Action**. Su valor por default es "**false**", o sea que los parámetros vacíos sí serán enviados.

Hagamos un ejemplo. Primero creamos un nuevo directorio, en las páginas web del proyecto, llamado "**redirectAction**":



Dentro de este directorio creamos una página llamada "**index.jsp**". Por el momento dejaremos esta página tal como está y comenzaremos a crear nuestro **Action**.

Agregamos una nueva clase llamada "**RedirectActionAction**" dentro del paquete "**actions**". Esta clase debe extender de "**ActionSupport**":

```
public class RedirectActionAction extends ActionSupport
{
}
```

Al igual que en el ejemplo anterior, esta clase será muy simple: no tendrá atributos y sólo sobre-escribiremos su método "**execute**" para regresar el valor de la constante "**SUCCESS**":

```
public class RedirectActionAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        return SUCCESS;
    }
}
```

Ahora que nuestro **Action** está listo, vamos al archivo de configuración "**struts.xml**" y hacemos su declaración:

```
<action name="redirect-action"
class="com.javatutoriales.results.actions.RedirectActionAction">
</action>
```

Lo siguiente es declarar el **result** al que iremos en el caso de la respuesta "**success**". El **result** será del tipo "**redirectAction**" y nos redirigirá al **Action** llamado "**dispatcher**" (el primero que creamos en el tutorial), recordemos que el parámetro por default de este **result** es el nombre del **Action**:

```
<result type="redirectAction">dispatcher</result>
```

La declaración de este **Action** queda de la siguiente forma:

```
<action name="redirect-action"
class="com.javatutoriales.results.actions.RedirectActionAction">
<result type="redirectAction">dispatcher</result>
</action>
```

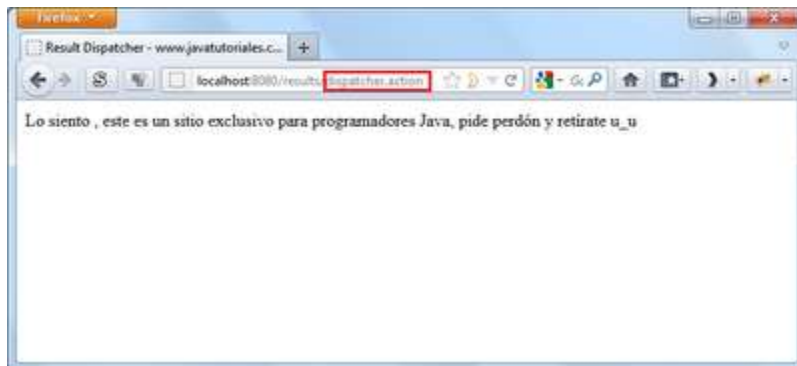
Regresemos a la página "**index.jsp**" que creamos hace unos momentos, y agreguemos un enlace para el **Action** "**redirect-action**" (no olviden indicar que haremos uso de las etiquetas de **Struts 2** con la directiva taglib correspondiente):

```
<s:a action="redirect-action">Redireccionar</s:a>
```

Ahora ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/redirectAction/index.jsp>

En donde únicamente debemos ver la liga que colocamos hace un momento. Al hacer clic sobre ella seremos redirigidos al **Action** llamado "**dispatcher**":



Recordemos que el **Action** anterior recibe dos parámetros, el nombre del usuario que envía el formulario y el lenguaje de programación que este usa. Por el resultado anterior podemos ver que no se están recibiendo ninguno de los dos parámetros. Ahora resolveremos esto pasando ambos parámetros, uno de forma estática y el otro de forma dinámica.

Como ahora pasaremos más parámetros al **result**, ya no podemos seguir indicando el nombre del **Action** como el parámetro por default, y debemos hacerlo de forma explícita:

```
<result type="redirectAction">
<param name="actionName">dispatcher</param>
</result>
```

Para pasar parámetros a los **Actions**, los definimos dentro del **result** que los invoca, y es tan simple como definirlos, usando su nombre, dentro del elemento "**<param>**"; el valor del parámetro lo colocaremos dentro de este elemento. Por ejemplo, para definir el parámetro "**nombre**", con un valor de "**Alex**", lo hacemos de la siguiente forma:

```
<param name="nombre">Alex</param>
```

El parámetro anterior está definido de forma estática, pero también podemos definirlos de forma dinámica. Los parámetros dinámicos necesitan tener dos partes, una declaración en el **result**, y un método **getter** en el **Action** del cual se obtendrá el valor. Definamos el parámetro "**lenguaje**".

Primero declararemos en el **result** un nuevo parámetro cuyo nombre será **lenguaje**:

```
<param name="lenguaje"></param>
```

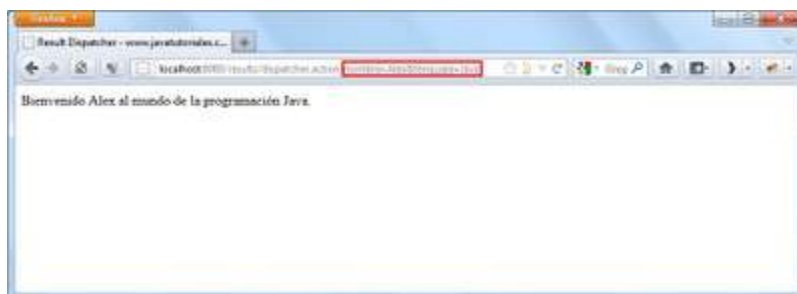
Queremos que el valor de este parámetro sea establecido de forma dinámica, por lo que lo debemos indicar, entre los elementos "**`\${}**" y "**`\${}**", qué atributo del **Action** que se debe llamar para obtener este valor. El valor se obtendrá del **getter** de este atributo, que debe seguir la convención de nombres de los **JavaBeans**; o sea que si queremos obtener el valor de un atributo llamado "**lenguajeProgramacion**", nuestro **Action** debe tener un método "**public String getLenguajeProgramacion()**". Por lo tanto, la definición de parámetro queda de la siguiente forma:

```
<param name="lenguaje">${lenguajeProgramacion}</param>
```

También debemos modificar la clase "**RedirectAction**", agregándole el **getter** necesario. En este caso el regresaremos un valor constante, pero este parámetro podría ser obtenido de forma dinámica, conectándose a una base de datos, de la sesión del usuario, o de alguna otra forma. Regresaremos el valor "**Java**" como nuestro lenguaje de programación favorito ^_^:

```
public String getLenguajeProgramacion()
{
    return "Java";
}
```

Ahora, cuando volvamos a ejecutar nuestra aplicación debemos ver la siguiente salida:



Esto nos indica que el ejemplo ha funcionado correctamente ^_^. Ahora veremos cómo funciona el siguiente tipo de **result**:

Result "chain"

Este **result** permite que se invoque otro **Action** completo, incluyendo todos sus interceptores y su **result**.

Aunque puede sonar como el **result** anterior, no lo es, ya que en el caso anterior el **result** le indicaba al navegador que debía redirigirse hacia otro **Action**, en este caso todos los **Actions** de la cadena se ejecutan en el servidor, y este nos regresa el resultado del último **Action**. Por la explicación anterior podemos entender además que es posible colocar más de un **Action** en la cadena de invocación (para esto el segundo **Action** debe regresar a un **result** que también sea de tipo "chain").

Este **Action** tiene cuatro parámetros:

- **actionName** (default) - El nombre del **Action** que será encadenado.
- **namespace** - Indica en cuál namespace se encuentra el **Action**
- **method** - Si queremos invocar un método en el **Action** que no sea el método "execute" lo indicamos con este parámetro.
- **skipActions** - Una lista, separada por comas, de los nombres de los **Actions** que pueden ser encadenados a este.

Comencemos con el ejemplo, en el cual encadenaremos todos los **Actions** que hemos creado hasta el momento ^_^.

Primero crearemos un nuevo directorio, en la parte de "Web Pages" del proyecto, llamado "chain", dentro de este creamos una nueva página llamada "index.jsp". En esta página nuevamente sólo pondremos una liga que nos enviará al **Action** que definiremos en un momento y que iniciará con la cadena de llamadas. El nombre de este **Action** será "chain", así que colocamos la liga de la siguiente forma (recuerden indicar con la directiva taglib correspondiente que haremos uso de las etiquetas de **Struts 2**):

```
<s:a action="chain">Encadenar</s:a><br />
```

Como en esta ocasión haremos uso de varios **Actions** "vacíos" no crearemos ninguna clase, esto ayudará para mostrar cómo se va llamando uno después de otro... y para aprovechar a mostrar otra cosa interesante de **Struts 2**.

Vamos al archivo de configuración "struts.xml" y declaramos un nuevo **action**, llamado "chain", sólo que en esta ocasión no declararemos ninguna clase para este **action**, de la siguiente forma:

```
<action name="chain">
</action>
```

Cuando **no** colocamos un valor en el atributo "class" del elemento "<action>" este por default usa la clase "ActionSupport" (aunque podemos cambiar esto en los archivos de configuración de **Struts 2**), de la cual han extendido todos nuestros **Actions** hasta el momento. **ActionSupport** tiene un método "execute" que regresa el valor de "success", y un método "input" que regresa el valor de "input". Eso quiere decir que el elemento "<action>" que acabamos de declarar, siempre irá a su **result** llamado "success".

Ahora colocaremos el **result** "success", el cual recuerden que es el nombre por default de los **results**, indicando que este será de tipo "chain" y que al terminar ejecutará un **action** llamado "chain2":

```
<action name="chain">
<result type="chain">chain2</result>
</action>
```

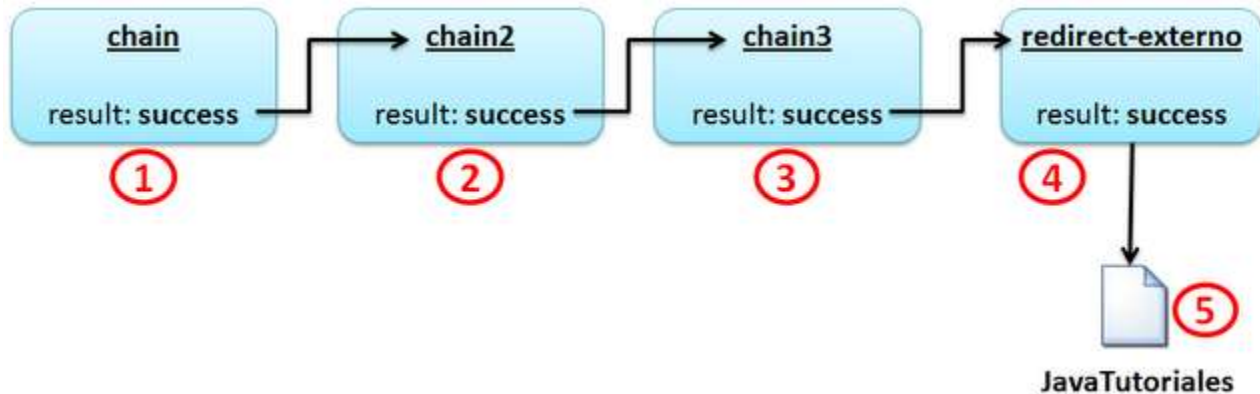
Declararemos otro **action** llamado "chain2", que será muy parecido al anterior, y en su **result**, también de tipo "chain", encadenará la llamada a un **action** llamado "chain3":

```
<action name="chain2">
<result type="chain">chain3</result>
</action>
```

Para terminar, declaramos un **action** llamado "**chain3**", que será muy parecido a los dos anteriores, con la diferencia de que este nos enviará ahora a un **Action** que si realiza una acción, yo encadenaré el **Action "redirect-externo"** pero pueden usar cualquiera que ustedes prefieran:

```
<action name="chain3">
<result type="chain">redirect-externo</result>
</action>
```

La invocación de los **Actions** será más o menos la siguiente:



- 1- Primero el **action "chain"** ejecuta su método "**execute**" que regresa "**success**". Como este **result** es de tipo "**chain**", se manda llamar el siguiente **action** de la cadena, que en este caso es "**chain2**".
- 2- El **action "chain2"** ejecuta su método "**execute**", este regresa el valor "**success**", el cual es un **result** de tipo "**chain**", por lo que se ejecuta el siguiente action de la cadena: "**chain3**".
- 3- "**chain3**" ejecuta su método "**execute**" el cual regresa el valor "**success**", este envía a un **result** de tipo "**chain**" el cual hará una llamada al **action** llamado "**redirect-externo**".
- 4- el **action** llamado "**redirect-externo**" ejecuta su método "**execute**", al terminar de ejecutarse este regresa el valor "**success**", el cual nos envía a un **result** de tipo "**redirect**" que nos envía a la página de **www.javatutoriales.com**
- 5- Nuestro navegador nos muestra la entrada correspondiente del blog ^_^.

En cada uno de los casos se ejecutarán todos los **actions**, del lado del servidor, y cuando se llegué a un **result** que no sea de tipo "**chain**" este se ejecutará de forma normal, en este caso será redirigir el navegador a la página de **Java Tutoriales :)**.

Ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/chain/index.jsp>

Veremos el enlace que creaos hace un momento. Al hacer clic sobre él, se ejecutarán nuestros cuatro **Actions**, y posteriormente veremos la página correspondiente del blog.

Como podemos imaginarnos, este tipo de **result** es muy útil cuando necesitamos ejecutar dos o más **Actions** juntos, y el primero no necesita un resultado visual (como un proceso de login y una carga de permisos del usuario).

Ahora que sabemos cómo funciona este **result**, pasemos al siguiente:

Result "stream"

Ya conocemos un poco acerca de este **result**, ya que lo vimos en [el tutorial número 4](#), cuando trabajamos con formularios. Por lo cual ahora solo hablaremos de los pequeños detalles.

Este **result** nos permite regresar un flujo de bytes al navegador del usuario para que este los interprete de alguna manera. Por ejemplo, podemos enviar un flujo de bytes que representen un archivo **PDF**, y si el navegador del cliente tiene el plugin adecuado lo podrá visualizar. Por lo regular, si el navegador no sabe como mostrar la información que le estamos enviando, le preguntará al usuario si quiere guardar el archivo o abrirlo con alguna aplicación particular.

Este result tiene los siguientes parámetros

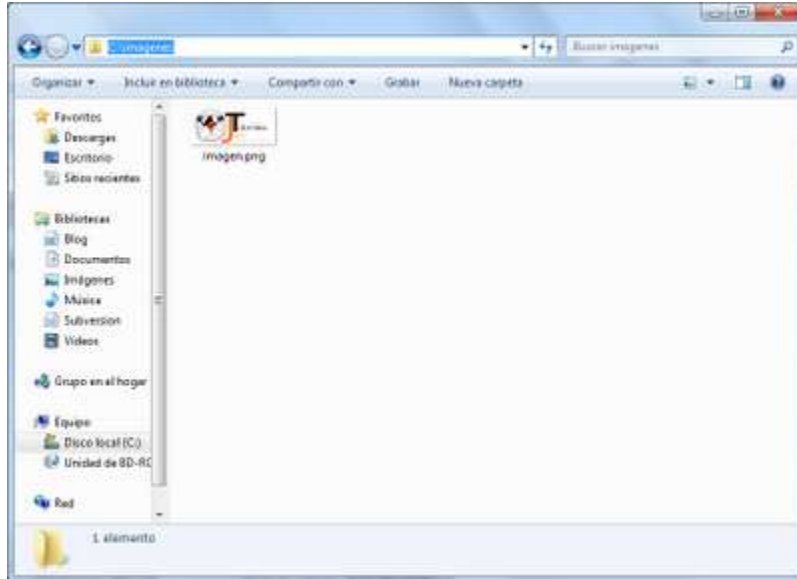
- **contentType**: El **mime-type** (el tipo del contenido) que es enviado al usuario. Por default su valor es **"text/plain"**.
- **contentLength**: El número de bytes contenidos en el flujo que enviaremos al usuario. Como vimos en [el tutorial anterior](#), este es usado para que el navegador pueda mostrar al usuario, de forma correcta, una barra de progreso de la descarga.
- **contentDisposition**: Este parámetro sirve para establecer el valor de la cabecera **"content-disposition"**, que el navegador usa para establecer el nombre del archivo que recibirá. El valor por default es **"inline"**.
- **inputName**: El nombre del atributo de nuestro **Action** que contiene el flujo de bytes. El atributo debe ser de tipo **"InputStream"**. Por default se busca un atributo de nombre **"inputStream"**.
- **bufferSize**: El tamaño del buffer que se usa para copiar los valores de la entrada al flujo de salida del navegador. Por default es **"1024"**.
- **allowCaching**: Indica si el navegador debe o no mantener en caché los datos que le hemos enviado. Esto quiere decir que solamente descargará el archivo una vez, y posteriormente lo regresará del caché. Esto es útil si el contenido que le enviaremos siempre será el mismo (digamos una imagen estática); pero si el contenido que le enviaremos cambiará de manera constante debemos indicarle que no queremos que lo guarde en caché, de esta forma siempre lo descargará del servidor. Su valor por default es **"true"**.
- **contentCharSet**: El juego de caracteres que se usan en el contenido (este parámetro es útil si regresamos texto plano en algún formato como **XML**).

Como podemos ver, este **result** no tienen ningún parámetro por default, por lo que siempre tendremos que usar el elemento **"<param>"** para establecer sus valores.

Haremos un pequeño ejemplo en el que regresaremos al usuario una imagen de forma dinámica. Lo primero que necesitamos es una imagen, yo usaré la siguiente:



Colocaremos la imagen anterior en cualquier ubicación de nuestro disco duro, por facilidad yo la pondré en "**C:\imagenes**":



Ahora que tenemos la imagen, creamos un nuevo directorio, en la zona de páginas web de nuestro proyecto, llamado "**stream**". Dentro creamos una página llamada "**index.jsp**". Por el momento dejaremos la página como está.

Creamos en el paquete "**actions**" una nueva clase llamada "**StreamResultAction**", esta clase debe extender de "**ActionSupport**":

```
public class StreamResultAction extends ActionSupport
{
}
```

Esta clase se encargará de crear un **InputStream** para leer la imagen, indicar el tamaño de la misma, y decir cuál será el nombre de esta imagen. Lo primero que debemos hacer es colocar unos cuantos atributos para mantener los valores de los datos anteriores. Colocamos en esta clase tres atributos, con sus correspondientes **getters**, de la siguiente forma:

```
private int bytesArchivo;
private InputStream streamImagen;
private String nombreImagen;

public int getBytesArchivo()
{
    return bytesArchivo;
}

public String getNombreImagen()
{
    return nombreImagen;
}

public InputStream getStreamImagen()
{
    return streamImagen;
}
```

Los **getters** son necesarios ya que es por medio de estos que el **result** puede leer los valores del **Action**.

Lo siguiente que debemos hacer es sobre-escribir el método "**execute**" para poder establecer el valor de los atributos anteriores. Dentro del método lo primero que haremos es crear un nuevo objeto de tipo "**File**" que apunte al archivo de la imagen, de esta manera podremos obtener el tamaño de la misma con el método "**length**" y su nombre usando el método "**getName()**":

```
@Override
public String execute() throws Exception
{
    File archivoImagen = new File("/imagenes/imagen.png");
    bytesArchivo = (int)archivoImagen.length();
    nombreImagen = archivoImagen.getName();

    return SUCCESS;
}
```

Usando el **File** anterior, creamos un **InputStream**, un objeto de tipo "**FileInputStream**" que será el flujo de bytes de la imagen:

```
streamImagen = new FileInputStream(archivoImagen);
```

Y esto es todo lo que debemos hacer. La clase "**StreamResultAction**" completa queda de la siguiente forma:

```
public class StreamResultAction extends ActionSupport
{
    private int bytesArchivo;
    private InputStream streamImagen;
    private String nombreImagen;

    @Override
    public String execute() throws Exception
    {
        File archivoImagen = new File("/imagenes/imagen.png");
        bytesArchivo = (int)archivoImagen.length();
        nombreImagen = archivoImagen.getName();

        streamImagen = new FileInputStream(archivoImagen);

        return SUCCESS;
    }

    public int getBytesArchivo()
    {
        return bytesArchivo;
    }

    public String getNombreImagen()
    {
        return nombreImagen;
    }

    public InputStream getStreamImagen()
    {
        return streamImagen;
    }
}
```

Lo siguiente que debemos hacer es declarar este **Action** en el archivo "**struts.xml**", el nombre que le daremos será "**stream**":

```
<action name="stream" class="com.javatutoriales.results.actions.StreamResultAction">
</action>
```

Ahora declararemos el **result** para el caso "**success**", este **result** será de tipo "**stream**":

```
<result type="stream">
</result>
```

Lo siguiente es declarar cada uno de los parámetros necesarios. En este ejemplo, algunos parámetros, como el "**contentType**" serán estáticos y otros como el "**contentLength**" serán dinámicos:

```
<result type="stream">
<param name="contentType">image/png</param>
<param name="contentLength">${bytesArchivo}</param>
<param name="inputName">streamImagen</param>
<param name="contentDisposition">attachment;filename="${nombreImagen}"</param>
</result>
```

La declaración del **Action** queda de la siguiente forma:

```
<action name="stream" class="com.javatutoriales.results.actions.StreamResultAction">
<result type="stream">
<param name="contentType">image/png</param>
<param name="contentLength">${bytesArchivo}</param>
<param name="inputName">streamImagen</param>
<param name="contentDisposition">attachment;filename="${nombreImagen}"</param>
</result>
</action>
```

Regresemos a la página **index.jsp** que creamos hace un momento. En esta colaremos una etiqueta "****" indicando que la imagen será devuelta por nuestro **Action**, para eso sólo debemos declarar en su atributo "**src**" con el valor del nombre de nuestro **Action**, o sea de la siguiente forma:

```

```

Ahora podemos ejecutar nuestra aplicación. Entramos en la siguiente dirección:

<http://localhost:8080/results/stream/index.jsp>

Y con eso deberemos ver la página con la imagen que cargamos:



Podemos ver con esto que el ejemplo ha funcionado de forma correcta, o sea el flujo de bytes se envió al navegador, indicando el tipo de contenido del mismo flujo, y este lo interpretó de forma correcta para mostrarlo al usuario.

Veamos ahora el penúltimo tipo de result de este tutorial:

Result "plaintext"

Este tipo de **result** envía el contenido de un archivo como texto plano al navegador. Con este **result** podemos, por ejemplo enviar el contenido de una **JSP**, sin que esta sea procesada, o cualquier otro archivo que pueda ser abierto con un editor de texto plano.

Este result tiene dos parámetros:

- **location** (default): la ubicación del archivo que será mostrada como texto plano. Esta ubicación será relativa a la raíz del proyecto web.
- **charset**: El conjunto de caracteres con el que será mostrado el archivo.

Veamos el ejemplo correspondiente. Lo primero que haremos es crear un nuevo directorio, en la sección de páginas web, llamado **"plaintext"**. Dentro de este directorio crearemos dos páginas, la primera será el típico **"index.jsp"** el cual dejaremos por el momento, y la otra será una página llamada **"pagina.html"** (así es, **html**).

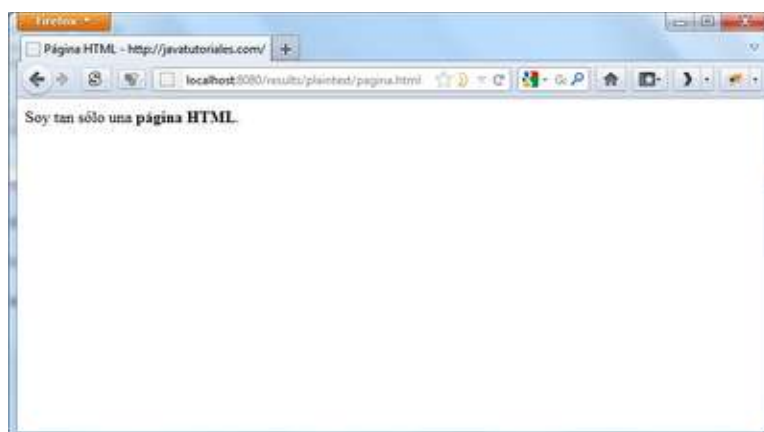
Esta página tendrá un simple mensaje que dirá: **"soy tan sólo una página HTML"**. La página (en **html5**) es la siguiente:

```
<!DOCTYPE html>
<html>
<head>
<title>Página HTML - http://javatutoriales.com/</title>
<meta charset="UTF-8" />
</head>
<body>
Soy tan sólo una <strong>página HTML</strong>.
</body>
</html>
```

Si ejecutamos la aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/plaintext/pagina.html>

Debemos ver el siguiente contenido:



Con [eso](#) comprobamos que el navegador pueda ejecutar correctamente esta página.

Ahora crearemos en el paquete "**actions**" una nueva clase llamada "**PlainTextResultAction**", la cual extenderá de "**ActionSupport**" y lo único que tendrá será un método "**execute**" sobrecargado que regresará el valor de la constante "**SUCCESS**":

```
public class PlainTextResultAction extends ActionSupport
{
    @Override
    public String execute() throws Exception
    {
        return SUCCESS;
    }
}
```

A continuación pasemos a la declaración de este **Action** en el archivo "**struts.xml**". El nombre de este **Action** será "**plaintext**":

```
<action name="plaintext"
class="com.javatutoriales.results.actions.PlainTextResultAction"></action>
```

Declaramos el **result** para este **Action**. El tipo del **result** será "**plainText**". Como en el juego de caracteres de nuestra página (**UTF-8**) es distinta a la del proyecto (**ISO-8859-1**), debemos indicar el valor del mismo en el parámetro "**charSet**". Además colocaremos el valor del parámetro "**location**" como la página **HTML** que creamos hace un momento (recuerden que este valor debe ser relativo a la raíz de las páginas web):

```
action name="plaintext" class="com.javatutoriales.results.actions.PlainTextResultAction">
<result type="plainText">
<param name="location">/plaintext/pagina.html</param>
<param name="charSet">UTF-8</param>
</result>
</action>
```

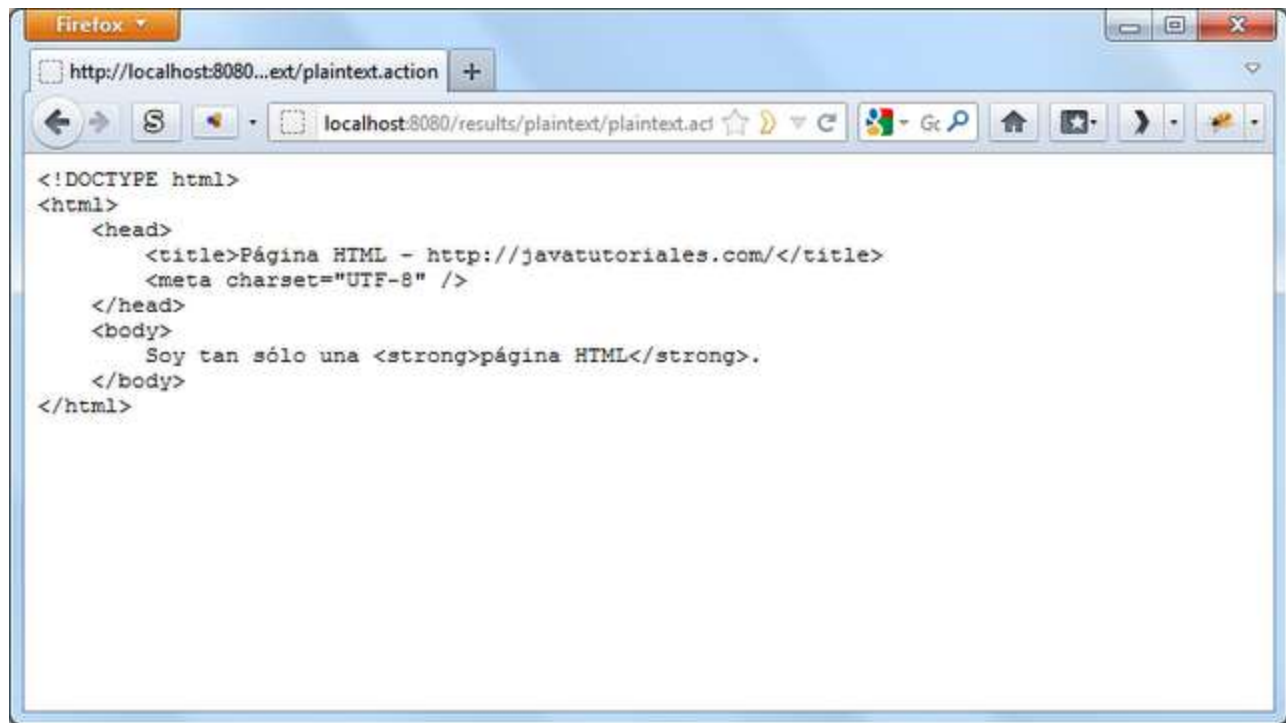
Ahora que tenemos esta declaración, lo último que haremos es regresar a la página "**index.jsp**" que creamos hace un momento y colocaremos un enlace al **Action** que acabamos de declarar, de esta forma (recuerden indicar que haremos uso de las etiquetas de **Struts 2** con el **taglib** correspondiente):

```
<s:a action="plaintext">Ver página</s:a>
```

Entramos a la siguiente dirección:

<http://localhost:8080/results/plaintext/index.jsp>

Con lo que deberemos ver el enlace que acabamos de declarar. Al hacer clic sobre el nos enviará al **Action** correspondiente, con lo cual deberemos ver en el contenido de la página **HTML** que creamos (en texto plano, sin ser procesada por el navegador):



Si revisan el código fuente de la página verán que es exactamente el mismo que creamos, lo que ocurre es que este **result** cambia el **content-type** para que pueda ser mostrado como texto.

Ahora veremos el último **result** de este tutorial:

Result "httpheader"

Este **result** permite establecer el valor de las cabeceras de la respuesta que se le envía al cliente. Con este **result** podemos, por ejemplo enviar estatus de error al cliente, indicando errores o falta de permisos para ejecutar ciertas acciones.

Este **result** puede recibir los siguientes parámetros:

- **status:** El estatus de la respuesta http.
- **parse:** Indica si los parámetros "**headers**" deben ser parseados como expresiones **OGNL** (por default es true)
- **headers:** Los valores de las cabeceras que queramos establecer. Estos se establecen colocando el nombre de la cabecera que queremos establecer precedida por un punto, como por ejemplo "**headers.a**", "**headers.b**", etc.
- **error:** El código de error **HTTP** que será enviado como respuesta.
- **errorMessage:** El mensaje de error que será mostrado en la respuesta.

La mayor utilidad de este tipo **result** es, como habrán imaginado por los parámetros, es para enviar estatus de error al usuario (ya que los mensajes de éxito muestran el contenido correspondiente del sitio).

Para los que tienen dudas de cuáles son los estatus que se pueden tener, pueden consultar la siguiente entrada de Wikipedia:

http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

También esta es la lista de los campos de las cabeceras que se pueden establecer para las peticiones y respuestas, en Wikipedia:

http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

Como una nota adicional, cada uno de los códigos de estatus debe ser usado con alguna o algunas cabeceras adicionales.

Veamos algunos ejemplos del uso de este **result**. Lo primero que hacemos es crear un nuevo directorio, en la sección de páginas web, llamado "**headers**". Dentro de este directorio crearemos una página llamada "**index.jsp**" en la que colocaremos las ligas de los **actions** que vayamos creando, por lo que indicaremos que haremos uso de las etiquetas de **Struts 2** con la directiva **taglib** correspondiente:

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

En este caso no crearemos ninguna clase **Action**, ya que solamente necesitamos su declaración, con su correspondiente "**<result>**" dentro del archivo "**struts.xml**".

El primer **result** que crearemos será para el caso que el usuario busque un recurso (una página, una imagen, un archivo, etc.) que haya sido movido de nuestro servidor a algún otro lugar. Si revisamos la lista de códigos de estatus, vemos que existe uno para este caso (vaya coincidencia ^^!), el código "**301 - Moved Permanently**". Si buscamos información sobre este código, veremos que, según Wikipedia (http://en.wikipedia.org/wiki/HTTP_301) este código debe de usarse junto con la cabecera "**location**".

Vamos al archivo "**struts.xml**" y declaramos un nuevo action llamado "**httpstatus301**", el cual no será implementado por ninguna clase:

```
<action name="httpstatus301">
</action>
```

A continuación declaremos un "**<result>**" para el caso "**success**", el tipo de este **result** será "**httpheader**":

```
<action name="httpstatus301">
<result type="httpheader">
</result>
</action>
```

Ahora viene lo interesante; "**301**" no es un código de error, sino que indica que la petición se realizó correctamente pero el recurso se movió a otra ubicación, la cual por cierto conocemos y queremos que el usuario sea redirigido de forma automática a esa nueva ubicación. Como **301** no es un código de error, lo indicamos usado el parámetro "**status**":

```
<result type="httpheader">
<param name="status">301</param>
</result>
```

Y como vimos hace un momento, cuando usamos el estatus **301**, también debemos indicar la cabecera "**Location**" con la nueva ubicación del recurso. Supongamos que en este caso la ubicación nueva del recurso que estamos buscando es: "**http://www.javatutoriales.com/**", por lo que lo indicamos de la siguiente forma:

```
<result type="httpheader">
<param name="status">301</param>
<param name="headers.Location">http://www.javatutoriales.com/</param>
</result>
```

La declaración completa de nuestro **action** queda de la siguiente forma:


```
<action name="httpstatus301">
<result type="httpheader">
<param name="status">301</param>
<param name="headers.Location">http://www.javatutoriales.com/</param>
</result>
</action>
```

Ahora regresaremos a la página "**index.jsp**" y agregaremos un enlace al action que acabamos de declarar, de la siguiente forma:

```
<s:a action="httpstatus301">301 - Moved Permanently</s:a>
```

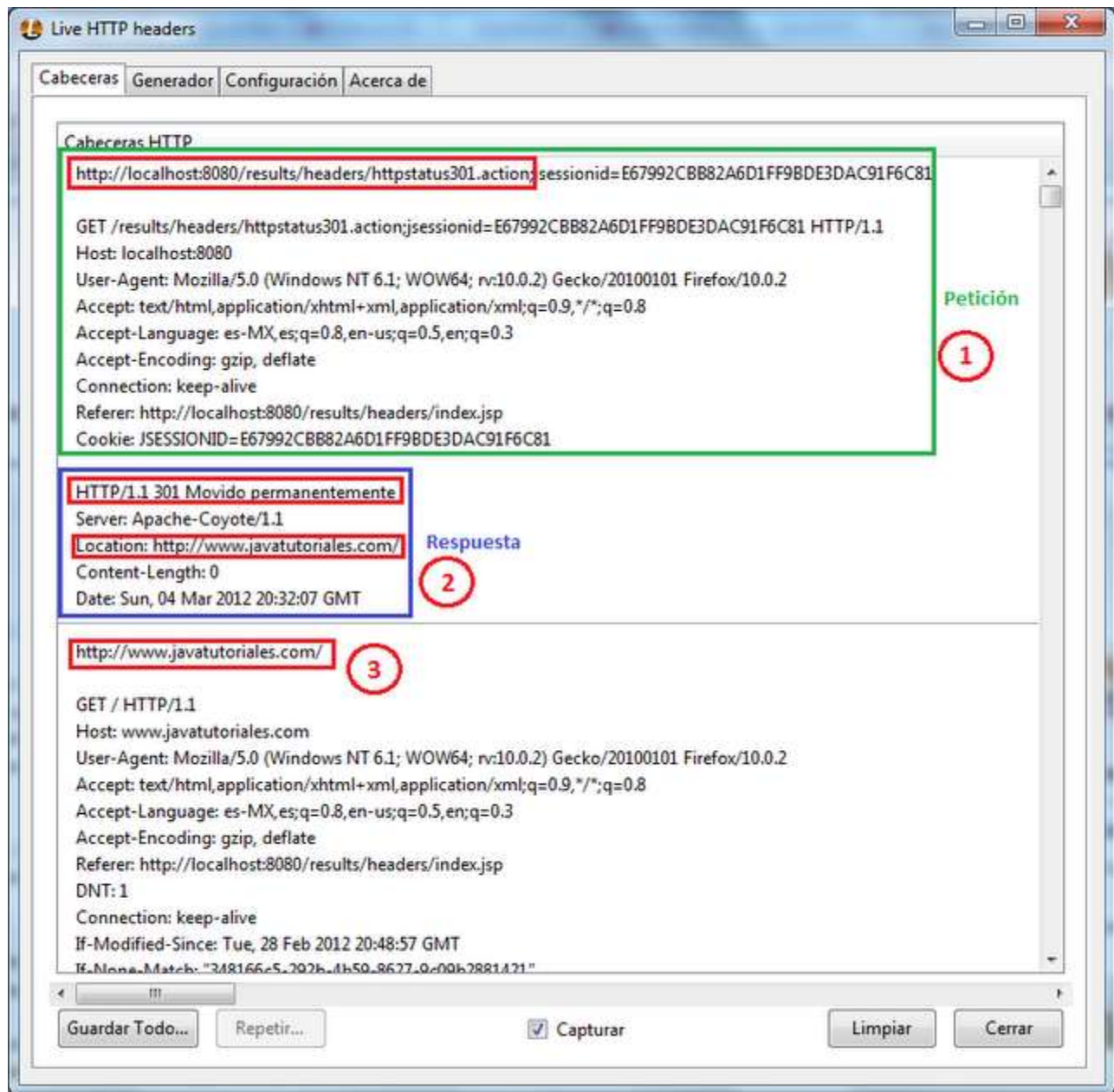
Ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/headers/index.jsp>

Con lo que debemos ver el enlace que creamos hace un momento. Cuando hagamos clic en el debemos ser redirigidos a otra página:



¿Qué fue lo que pasó? Para responder a esta pregunta haremos uso de una extensión de firefox llamada "**live http headers**" que nos permite analizar los encabezados de las peticiones y las respuestas que realizamos.



Podemos ver que en primer lugar (paso 1) el navegador hace una petición a la página "**http://localhost:8080/results/headers/index.jsp**" (la parte que dice "**jsessionid**" es la cookie por la cual se manejan las peticiones en los servidores java), el servidor envía la respuesta (paso 2) con el código "**301 Movido permanentemente**" y envía además la cabecera "**Location**" indicando que el recurso ahora se encuentra en "**http://www.javatutoriales.com**". El navegador hace una nueva petición a esta ubicación (paso 3) y se continúa con el proceso normal.

Podemos ver que la redirección ha funcionado correctamente.

Veamos ahora un ejemplo de un código de error. En **Http** existen dos tipos de errores: los errores generados por el cliente (cuando, por ejemplo, la petición no se entiende o cuando excede de un cierto tamaño) y los errores generados por el servidor (cuando ocurre algún error inesperado que no permite que la petición se procese).

Los errores de cliente tienen un código a partir del **400** y los del servidor a partir del **500**. Veamos primero un error generado por el cliente.

El status **400** indica que la petición no puede ser completada debido a que la sintaxis en la que se recibió no es correcta.

Declaramos un nuevo elemento "<action>" en el archivo "struts.xml" que se llamará "httpstatus400" y no será implementado por ninguna clase:

```
<action name="httpstatus400">
</action>
```

Dentro de este action declaramos el **result** para el caso "success", que será del tipo "httpheader".

```
<result type="httpheader">
</result>
```

Como en este caso "400" si se trata de un error, debemos indicarlo usando el parámetro "error" en vez de "status" como la vez anterior:

```
<result type="httpheader">
<param name="error">400</param>
</result>
```

Para terminar, cuando indicamos un error, en vez de proporcionar una cabecera se indica un mensaje de error para que el usuario tenga una idea de lo que ha salido mal, esto lo hacemos con el parámetro "errorMessage":

```
<result type="httpheader">
<param name="error">400</param>
<param name="errorMessage">Los datos proporcionados no pueden ser entendidos, tal vez ha
escrito algo mal o su navegador no puede colocarlos en la sintaxis adecuada, favor de
utilizar otro navegador.</param>
</result>
```

La declaración completa del **action** queda de la siguiente forma:

```
<action name="httpstatus400">
<result type="httpheader">
<param name="error">400</param>
<param name="errorMessage">Los datos proporcionados no pueden ser entendidos, tal vez ha
escrito algo mal o su navegador no puede colocarlos en la sintaxis adecuada, favor de
utilizar otro navegador.</param>
</result>
</action>
```

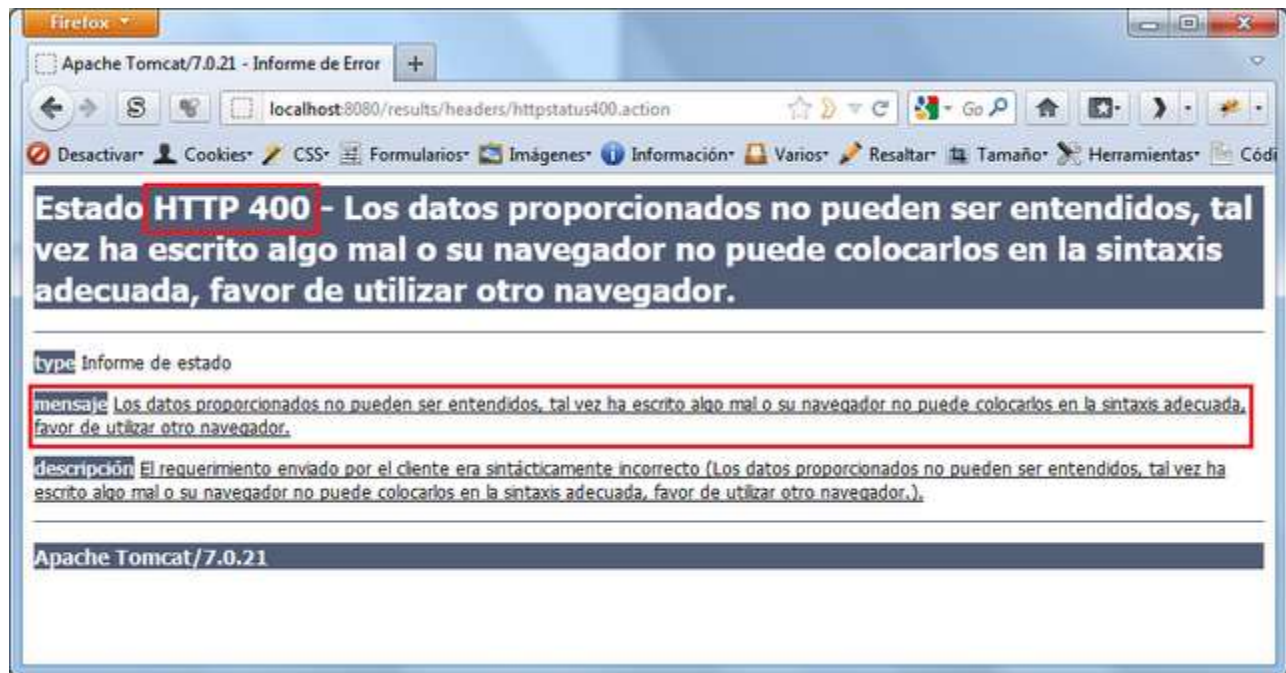
Ahora regresamos a la página "index.jsp" y agregamos un enlace más, para este nuevo **Action**:

```
<s:a action="httpstatus400">404 - Bad Request</s:a>
```

Ejecutamos nuestra aplicación y entramos en la siguiente dirección:

<http://localhost:8080/results/headers/index.jsp>

En donde deberemos ver enlace que acabamos de crear. Al hacer clic sobre él, deberemos ver la siguiente salida:



En donde podemos ver que el navegador nos muestra una ventana de error, con el mensaje que hemos indicado. Así el usuario sabrá que ha hecho algo mal y así podrá corregirlo.

Veamos el último ejemplo, en este crearemos otra respuesta de error, sólo que ahora será un error del servidor, en el cual usaremos el status "**501**" para indicar que la funcionalidad que está solicitada no ha sido implementada aún.

En el archivo "**struts.xml**" declaramos nuevamente un elemento "**<action>**" que se llamará "**httpstatus501**" y no será implementado por ninguna clase:

```
<action name="httpstatus501">
</action>
```

Lo siguiente que haremos es declarar un elemento "**<result>**" para el caso "**success**" que será de tipo "**httpheader**":

```
<result type="httpheader">
</result>
```

Como el estatus "**501**" se trata de un error, lo indicaremos usando el parámetro "**error**" y daremos un mensaje con la descripción del error usando el parámetro "**errorMessage**":

```
<result type="httpheader">
<param name="error">501</param>
<param name="errorMessage">Estimado usuario, la funcionalidad que está tratando de usar
aún no ha sido implementada. Le pedimos tenga paciencia y regrese más tarde ^^!</param>
</result>
```

La declaración completa del **Action** queda de la siguiente forma:

```
<action name="httpstatus501">
<result type="httpheader">
<param name="error">501</param>
<param name="errorMessage">Estimado usuario, la funcionalidad que está tratando de usar
aún no ha sido implementada. Le pedimos tenga paciencia y regrese más tarde ^^!</param>
</result>
</action>
```

Ahora regresamos nuevamente a la página "**index.jsp**" y agregamos un enlace al action que acabamos de declarar:

```
<s:a action="httpstatus501">501 - Not Implemented</s:a>
```

Ejecutamos nuevamente nuestra aplicación y entramos en la siguiente dirección:

<http://localhost:8080/results/headers/index.jsp>

Hacemos clic en el enlace que acabamos de colocar y deberemos ver el siguiente error:



Lo cual nos indica que el ejemplo ha funcionado correctamente ^^_^^.

Para terminar con este tutorial, veremos una funcionalidad que nos permite ejecutar un proceso después de que el **Action** se ha ejecutado, pero antes de regresar el **result**:

PreResultListener

Como dijimos hace un momento: este mecanismo permite realizar algún procesamiento después de que se ha ejecutado nuestro **Action**(o interceptor, pero eso lo veremos en el siguiente tutorial) y antes de que se ejecute el **result**. Esto nos permite que modifiquemos algunas cosas, como por ejemplo, podemos cambiar el **result** al que seremos dirigidos (en vez de ir al **result "input"** iremos a "**success**") o modificar los parámetros del **Action** antes de que el **result** se ejecute.

Struts 2 proporciona la interface "**com.opensymphony.xwork2.interceptor.PreResultListener**" para este fin. Esta interface puede ser agregada tanto a un **Action** como a un **Interceptor**, aunque la única forma de agregar

este **PreResultListener** es en código dentro del método "**execute**", esto quiere decir que no hay alguna forma de hacerlo en configuración por lo que, si queremos quitar este listener deberemos ir a nuestro código, eliminarlo y volver a compilarlo.

Veamos cómo funciona este listener con un ejemplo. Primero creamos un nuevo directorio llamado "**preresult**" en la sección de páginas web. Dentro de esta crearemos tres páginas, la primera se llamara "**index.jsp**", la segunda "**exito.jsp**", y la tercera "**error.jsp**". En la primer página colocaremos un formulario que será muy parecido al del primer ejemplo que vimos. En el formulario preguntaremos el nombre del usuario y su lenguaje de programación favorito, si el lenguaje es "**Java**" lo enviaremos a la página "**exito.jsp**", en caso contrario lo enviaremos a la página "**error.jsp**". El formulario queda de la siguiente forma:

```
<s:form action="presult">
<s:textfield name="nombre" label="Nombre" />
<s:select name="lenguaje" label="Lenguaje de Programacion" list="{ 'Java', 'PHP', '.Net' }"
/>
<s:submit value="Enviar" />
</s:form>
```

No olviden indicar que haremos uso de las etiquetas de **Struts 2** usando la directiva "**taglib**" correspondiente.

Antes de ver el **Action** crearemos el contenido de las páginas "**exito.jsp**" y "**error.jsp**". El contenido de "**exito.jsp**" será el siguiente:

```
Pasa por favor mi estimado <s:property value="nombre" />, has entrado al mundo de la
programación Java.
```

Si, sólo el contenido anterior. Y el de "**error.jsp**" será el siguiente:

```
Perdón <s:property value="nombre" />, este es un sitio exclusivo para programadores Java,
pero siéntate al fondo a ver si aprendes algo...
```

Ahora crearemos un nuevo **Action**, en el paquete "**actions**", llamado "**PreResultListenerAction**" que extenderá de "**ActionSupport**":

```
public class PreResultListenerAction extends ActionSupport
{
}
```

A esta clase le colocaremos dos atributos de tipo **String**: "**nombre**" y "**lenguaje**", con sus correspondientes **getters** y **setters**, estos atributos contendrán los valores que son recibidos a través del formulario:

```
private String nombre;
private String lenguaje;
```

```

    public String getLenguaje()
    {
        return lenguaje;
    }

    public void setLenguaje(String lenguaje)
    {
        this.lenguaje = lenguaje;
    }

    public String getNombre()
    {
        return nombre;
    }

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

```

Ahora sobre-escribiremos el método "**execute**" para, en base al lenguaje seleccionado, decidir a cuál página enviar al usuario. Si el lenguaje es "**Java**" lo enviaremos el **result** "**SUCCESS**", en caso contrario lo enviaremos al **result** "**ERROR**":

```

@Override
    public String execute() throws Exception
    {
        if (!"Java".equals(lenguaje))
        {
            return ERROR;
        }

        return SUCCESS;
    }

```

Ahora que tenemos todos los elementos, vamos al archivo "**struts.xml**" y declaramos un nuevo elemento "**<action>**" llamado "**presult**" que será implementado por la clase que acabamos de crear:

```

<action name="presult"
class="com.javatutoriales.results.actions.PreResultListenerAction"></action>

```

Agregaremos dos **results**, uno llamado "**succes**" que nos enviará a la página "**/preresult/exito.jsp**", y otro llamado "**error**" que nos enviara a la página "**/preresult/error.jsp**":

```

<action name="presult"
class="com.javatutoriales.results.actions.PreResultListenerAction">
<result>/preresult/exito.jsp</result>
<result name="error">/preresult/error.jsp</result>
</action>

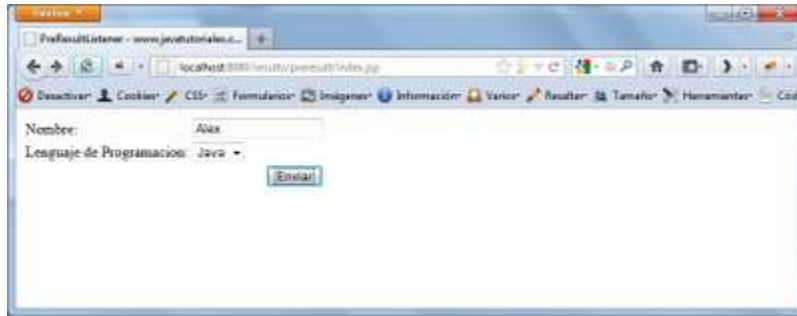
```

Como ya somos expertos en **results**, ¿de qué tipo son los dos **results** anteriores ;)?

Ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/preresult/index.jsp>

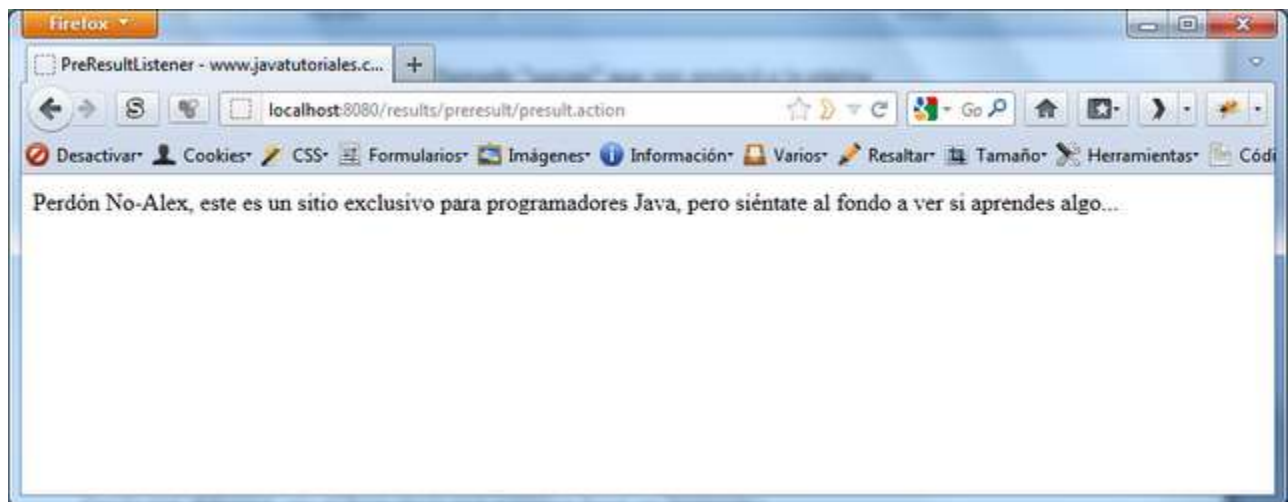
Con lo que debemos ver el formulario que creamos hace un momento:



Colocamos nuestro nombre, y si en el lenguaje de programación colocamos **"Java"** debemos ser enviados a la página **"exito.jsp"**:



De lo contrario iremos a la página **"error.jsp"**:



Con esto podemos ver que todo está configurado y funcionando de forma correcta.... ya sé lo que se están preguntando: "¿.... y dónde está el **PreResultListener**? :S". Bueno, esto sólo era para ver el comportamiento normal de la aplicación. Ahora agregaremos el listener:

Cada **Action**, como habíamos visto en el tutorial anterior, tiene una forma de obtener el contexto en el cual se ejecuta (el **"ActionContext"**). Este contexto nos permite obtener un objeto de tipo **"ActionInvocation"** que representa... bueno la

invocación del **Action** ^^!. Con este objeto podemos, entre otras cosas, agregar un **"PreResultListener"** a la invocación de este **Action**.

La obtención de estos dos objetos debe hacerse dentro del método **"execute"**:

```
@Override
public String execute() throws Exception
{
    ActionInvocation actionInvocation = ActionContext.getContext().getActionInvocation();
}
```

"PreResultListener" es una interface que tiene sólo un método:

```
void beforeResult(ActionInvocation invocation, String resultCode);
```

El cual será invocado cuando la ejecución del **Action** termine, pero antes de que se comience con el procesamiento de **result**.

Normalmente esta interface se implementa de dos formas: como una clase anidada dentro del **Action**, o como una clase interna anónima dentro del método **"execute"**, al momento de agregar el listener. La decisión de cuál de las dos usar es, como siempre, una cuestión de diseño y dependerá más de cuántas cosas haremos dentro de método **"beforeResult"**. Si haremos muchas cosas lo mejor podría ser crear una clase anidada que se encargue de realizar todas las tareas, si se harán pocas (como es nuestro caso) lo mejor es usar una clase anónima.

La clase **"ActionInvocation"** tiene un método llamado **"addResultListener"** que es el que usamos para agregar nuestro listener:

```
actionInvocation.addPreResultListener(new PreResultListener()
{
    public void beforeResult(ActionInvocation ai, String resultCode)
    {
    }
});
```

Dentro del método **"beforeResult"** colocaremos la lógica que se ejecutará antes de comenzar con el procesamiento del **result**. En nuestro caso haremos un cambio sencillo, no importa qué datos introduzca el usuario, siempre lo regresaremos al **result "success"**.

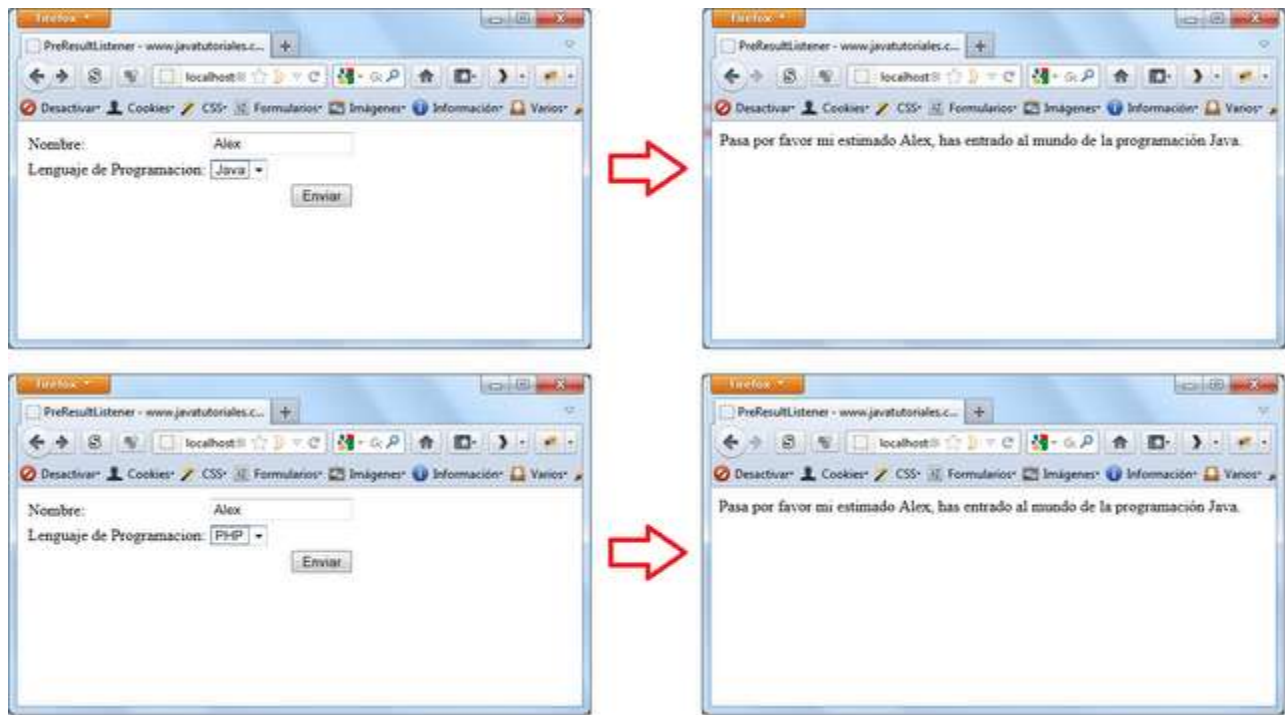
El método **"beforeResult"** recibe el nombre del **result** al que actualmente irá la petición, a través del parámetro **"resultCode"**, de esta forma sabremos a dónde se supone que debe ir el usuario. En nuestro caso no importa a cuál **result** vaya a ser enviado originalmente, de todas formas lo reenviaremos al **result "success"**. Para cambiar el **result** al que lo enviaremos establecemos usamos el método **"setResultCode"** del objeto **"ActionInvocation"** que recibimos como parámetro; a este método le pasamos como argumento el nombre del nuevo **result**, en nuestro caso usaremos la constante **"SUCCESS"**, de la siguiente forma:

```
public void beforeResult(ActionInvocation ai, String resultCode)
{
    ai.setResultCode(SUCCESS);
}
```

Probemos que efectivamente seamos enviados siempre a la página de éxito; ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/results/preresult/index.jsp>

Colocamos los datos en nuestro formulario, y no importa qué lenguaje de programación seleccionemos, siempre seremos enviados a la página del resultado de éxito:



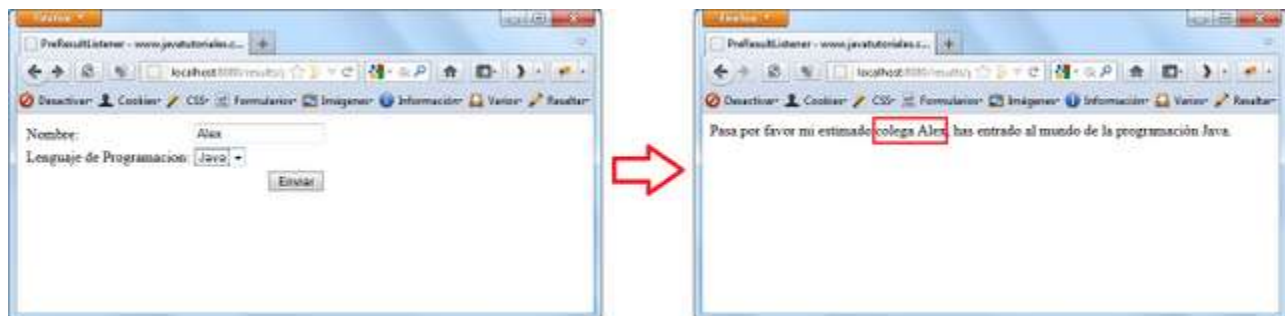
Hagamos un cambio más. Al momento de saludar al usuario podremos hacerlo de una forma aún más amable. En vez de saludarlo sólo por su nombre, lo saludaremos diciéndole "**colega**", para lo cual modificaremos el nombre del usuario, agregando siempre la palabra "**colega**" antes del nombre que nos proporciona:

```
public void beforeResult(ActionInvocation ai, String resultCode)
{
    ai.setResultCode(SUCCESS);
    nombre = " colega " + nombre;
}
```

Y listo, ahora podemos volver a ejecutar nuestra aplicación, entrar a la siguiente dirección y proporcionar nuestros datos:

<http://localhost:8080/results/preresult/index.jsp>

Al enviar nuestros datos, debemos ser saludados como nos lo merecemos ^_^:



Como vimos, usando un "**PreResultListener**" es muy fácil modificar el comportamiento de nuestros **results** sin tener que hacer modificaciones en la definición de los mismos.

Ahora veamos un último y pequeño tema extra acerca de los **results**:

Results Globales

Hasta ahora, todos los **results** que hemos visto son declarados dentro de un **Action**, pero ¿qué pasa si necesitamos que más de un **Action** tenga acceso al mismo **result**? Imaginemos el caso de una página para errores globales cuando algo dentro de nuestra aplicación sale mal, y no tenemos idea de por qué (aunque sé que esto nunca pasa ^^).

El caso más común es cuando necesitamos una página de "**login**" a la cual enviar a un usuario si está tratando de ejecutar un **Action** que sólo pueden usar los usuarios registrados en el sitio y este no ha iniciado una sesión.

Los **results globales** funcionan de la misma forma que el resto de los que hemos visto, y declararlos es igual de sencillo que el resto, sólo que en este caso no están dentro de un elemento "**<action>**", sino dentro del elemento "**<global-results>**" que debe estar siempre dentro de un paquete. Para declarar varios resultados globales lo hacemos de la siguiente forma:

```
<global-results>
<result name="login">/login.jsp</result>
<result name="error">/error.jsp</result>
</global-results>
```

Una pregunta que deben estarse haciendo en este momento es: ¿y qué pasa si tengo declarado un **result** con el nombre de "**error**" en mi "**<action>**" y otro con el mismo nombre en los **results** globales? Buena pregunta; en ese caso el **result** declarado dentro del **action** tiene mayor precedencia. Cuando indicamos que debemos ir a un **result** (digamos "**login**"), **Struts 2** siempre lo buscará dentro de la declaración del **action** en ejecución, si no lo encuentra irá a buscarlo a los resultados globales, y en caso de que este tampoco sea encontrado ahí, arrojará un error.

Struts 2 - Parte 6: Interceptores

Al desarrollar una aplicación, es necesario desarrollar algunas funcionalidades de utilidad [que](#) nos hagan más fácil el desarrollo de los procesos de negocio en los cuales estamos interesados. Algunos ejemplos de estas funcionalidades pueden ser el verificar que el usuario haya iniciado una sesión en el sistema, o que tenga permisos para ejecutar la operación que está solicitando, convertir los parámetros de un tipo a otro (de **String** a **Date**, o a **int**), verificar que se hayan proporcionado los datos que son obligatorios, agregar alguna información a la sesión del usuario, etc.

La mayoría de estas funcionalidades pueden ser creadas de una forma genérica y usadas en varias partes de nuestra aplicación o en varias aplicaciones.

Struts 2 integra un mecanismo que nos permite abstraer estas funcionalidades de una manera sencilla, aplicarlas de forma transparente a las acciones que la necesiten (que pueden ser desde una hasta todas las acciones de la aplicación) y configurarlas a través del uso de parámetros.

En este tutorial aprenderemos cómo configurar los interceptores que vienen integrados con **Struts 2**, además de crear nuestros propios interceptores, agregarlos a un **Action**, y agregarlos a la pila que se aplica [por](#) default a los **Actions** de nuestra aplicación.

Recordando un poco de la teoría que vimos en [el primer tutorial](#): los interceptores son clases que siguen el patrón **interceptor**. Estos permiten que se implementen funcionalidades "**cruzadas**" o comunes para todos los **Actions**, pero que se ejecuten fuera del **Action** (por ejemplo validaciones de datos, conversiones de tipos, población de datos, etc.).

Los interceptores realizan tareas antes y después de la ejecución de un **Action** y también pueden evitar que un **Action** se ejecute (por ejemplo si estamos haciendo alguna validación que no se ha cumplido). Sirven para ejecutar algún proceso particular que se quiere aplicar a un conjunto de **Actions**. De hecho muchas de las características con que cuenta **Struts 2** son proporcionadas por los interceptores.

Si alguna funcionalidad que necesitamos no se encuentra en los interceptores de **Struts 2**, podemos crear nuestro propio interceptor y agregarlo a la cadena o pila que se ejecuta por default.

De la misma forma, podemos modificar la pila de interceptores de **Struts 2**, por ejemplo para quitar un interceptor o modificar su orden de ejecución.

Cada interceptor proporciona una característica distinta al **Action**. Para sacar la mayor ventaja posible de los interceptores, un **Action** permite que se aplique más de un interceptor. Para lograr esto **Struts 2** permite crear pilas o stacks de interceptores y aplicarlas a los **Actions**. Cada interceptor es aplicado en el orden en el que aparece en el stack. También **podemos formar pilas de interceptores en base a otras pilas**. Una de estas pilas de interceptores es aplicada por default a todos los **Actions** de la aplicación (aunque si queremos también podemos aplicar una pila particular a un **Action**).

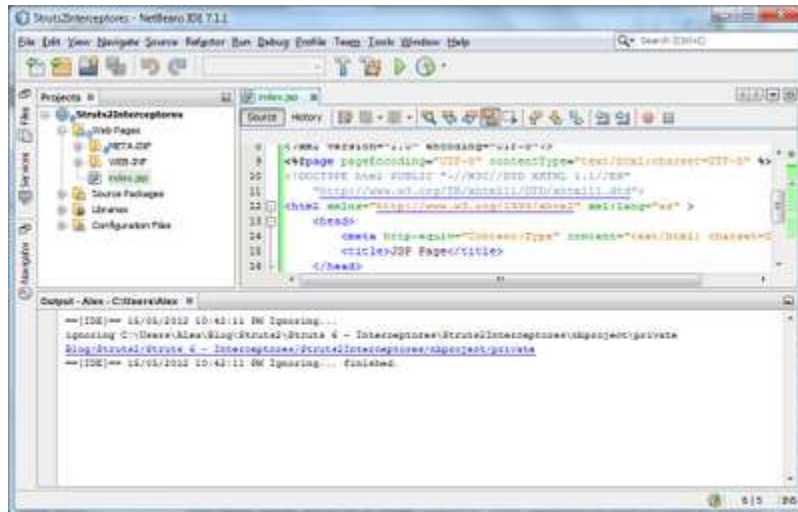
En [el primer tutorial](#) pueden ver una tabla con algunos de los interceptores más usados en **Struts 2**, y de las pilas de interceptores que vienen pre-configuradas.

Para decirlo de otra forma, los interceptores en **Struts 2** son los equivalentes a los [filtros en la especificación de Servlets](#).

Comencemos a ver cómo funcionan los interceptores en **Struts 2** de la forma en la que nos gusta a los programadores, con código ^^.

Creemos un nuevo proyecto web en **NetBeans**, vamos al Menú "**File->New Project...**". En la ventana que se abre seleccionamos la categoría "**Java Web**" y en el tipo de proyecto "**Web Application**". Le damos una ubicación y un nombre al proyecto, en mi caso será "**Struts2Interceptores**". Presionamos el botón "**Next**". En la siguiente pantalla deberemos configurar el servidor de aplicaciones que usaremos. Yo usaré la versión 7 de **Tomcat**. Como versión de **Java EE** usaré la 5 (pueden usar también la 6 si quieren, solo que en ese caso **NetBeans** no genera el archivo "**web.xml**" por default, y tendrán que crearlo a mano o usando el wizard correspondiente). De la misma forma, si quieren pueden

modificar el context path de la aplicación. Presionamos el botón **"Finish"**, con lo que veremos aparecer la página **"index.jsp"** en nuestro editor:



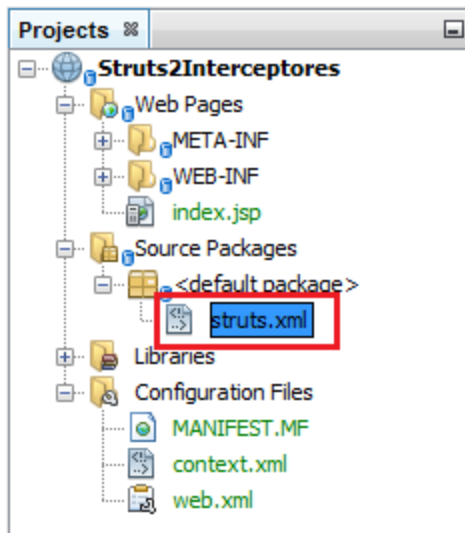
Ya que nuestro proyecto está creado, agregamos la biblioteca **"Struts2"** que creamos en [el primer tutorial de la serie](#). Para esto hacemos clic derecho en el nodo **"Libraries"** del panel de proyectos. En el menú que aparece seleccionamos la opción **"Add Library..."**. En la ventana que aparece seleccionamos la biblioteca **"Struts2"** y presionamos **"Add Library"**. Con esto ya tendremos los jars de **Struts 2** en nuestro proyecto.

A continuación configuramos el filtro "struts2" en el deployment descriptor. Abrimos el archivo "web.xml" y colocamos el siguiente contenido, como se explicó en el primer tutorial de la serie:

```
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

Ahora crearemos un nuevo archivo llamado **"struts.xml"** en el paquete default de código fuente de nuestra aplicación:



Este archivo contendrá la configuración de **Struts 2** de nuestra aplicación. Colocaremos la configuración inicial, con un par de constantes para indicar que nos encontramos en modo de desarrollo y que cualquier cambio que hagamos en el archivo de configuración debe recargarse de inmediato, y un paquete inicial:

```
<?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.devMode" value="true" />
<constant name="struts.configuration.xml.reload" value="true" />

<package name="struts-interceptores" extends="struts-default">
</package>
</struts>
```

Creamos un nuevo paquete en el nodo "**Source Packages**" llamado "**com.javatutoriales.interceptores.actions**" que será donde colocaremos las clases **Action** de nuestra aplicación.

Antes de comenzar con los ejemplos, dejaré la lista de los interceptores de dos de las pilas de interceptores que vienen configuradas en **Struts 2**. La primera es la pila básica ("**basicStack**"), que contiene, como pueden imaginar, los interceptores mínimos necesarios para que una aplicación pueda proporcionar las funcionalidades básicas. Los interceptores de esta pila son:

- **exception**
- **servletConfig**
- **prepare**
- **checkbox**
- **params**
- **conversionError**

La segunda pila importante es la pila que se aplica por default a todos los **Actions** de nuestra aplicación ("**defaultStack**"). Esta pila aplicará todos los interceptores, en el orden en que se encuentran:

- **exception**
- **alias**
- **servletConfig**
- **prepare**
- **i18n**
- **chain**
- **debugging**
- **profiling**
- **scopedModelDriven**
- **modelDriven**
- **fileUpload**
- **checkbox**
- **staticParams**
- **conversionError**
- **validation**
- **workflow**

Es importante tener en cuenta cuáles y en qué orden se ejecutan estos interceptores para comprender algunos de los ejemplos que haremos.

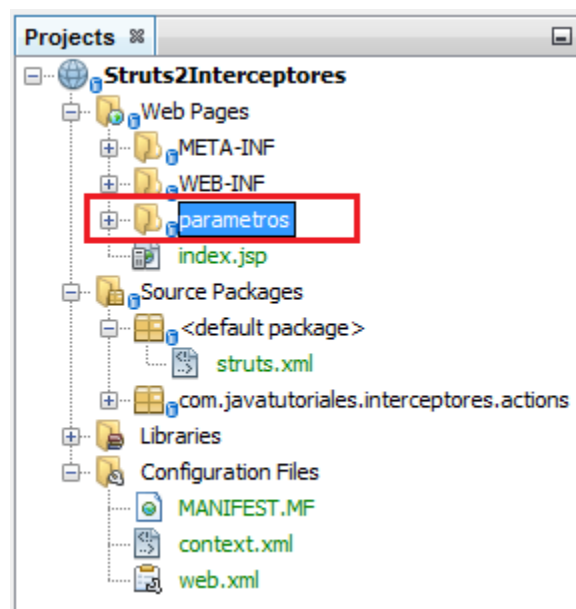
Ahora sí, ya con esta información comenzaremos con el primer ejemplo del tutorial:

Configurar parámetros de Interceptores

Iniciaremos modificando el comportamiento de uno de los interceptores que ya vienen en la pila por default. Ya hemos tratado un poco con este interceptor en [el cuarto tutorial de la serie](#): el interceptor **"fileUpload"** (que si revisan la lista anterior se aplica sólo en la **"defaultStack"**).

Este interceptor permite que se envíen a través de un formulario, además de la información en texto plano normal, archivos de cualquier tipo.

Veamos cómo funciona por default este interceptor. Crearemos un nuevo directorio llamado **"parametros"** dentro de las páginas web:



Dentro de este directorio creamos dos **JSPs**, la primera se llamará "**index.jsp**" y la segunda "**resultado.jsp**". "**index.jsp**" contendrá un formulario con un único campo de tipo "**file**" para subir un archivo. Este formulario será procesado por una **Action** llamada "**carga-archivo**".

Recuerden que cuando se realizará la carga de un archivo en una página web, el formulario debe estar codificado con el tipo "**multipart/form-data**".

También recuerden que para hacer uso de las etiquetas de **Struts 2** es necesario indicar en la página que se usará el siguiente **taglib**:

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

La declaración del formulario queda de la siguiente forma:

```
<s:form action="carga-archivo" enctype="multipart/form-data">
<s:file name="archivo" label="Archivo" />
<s:submit />
</s:form>
```

Ya teniendo listo el formulario vamos a la página "**resultado.jsp**". En esta página sólo pondremos un mensaje que indicará cuál es el **content-type** del archivo que hemos cargado. Para esto usamos la etiqueta "**<s:property>**" de la siguiente forma:

```
El archivo es de tipo <s:property value="archivoContentType" />
```

Ahora crearemos el **Action** que procesará este formulario. Dentro del paquete "**com.javatutoriales.interceptores.actions**" creamos una nueva clase llamada "**CargaArchivo**", esta clase debe extender de "**ActionSupport**":

```
public class CargaArchivo extends ActionSupport
{
}
```

Esta será una clase sencilla. Sólo contendrá dos atributos, uno de tipo "**File**" para almacenar el archivo que sea cargado desde la página, y otro de tipo "**String**" para mantener el **content-type** del archivo que subamos (para más detalles pueden revisar el cuarto tutorial de la serie):

```
private File archivo;
private String archivoContentType;
```

Para el ejemplo regresaremos el valor del **content-type**, por lo tanto colocaremos su **getter**, y como que el framework establezca los valores de los atributos anteriores necesitamos sus **setters**, colocamos los siguientes métodos en nuestra clase:

```
public void setArchivo(File archivo)
{
    this.archivo = archivo;
}

public String getArchivoContentType()
{
    return archivoContentType;
}

public void setArchivoContentType(String archivoContentType)
{
    this.archivoContentType = archivoContentType;
}
```


Ahora sobre-escribimos el método "**execute**" de la clase. En este caso lo único que hará será regresar el valor "**SUCCESS**" para que nos envíe a la página del resultado "**success**":

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

La clase "**CargaArchivo**" completa queda de la siguiente forma:

```
public class CargaArchivo extends ActionSupport
{
    private File archivo;
    private String archivoContentType;

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

    public void setArchivo(File archivo)
    {
        this.archivo = archivo;
    }

    public String getArchivoContentType()
    {
        return archivoContentType;
    }

    public void setArchivoContentType(String archivoContentType)
    {
        this.archivoContentType = archivoContentType;
    }
}
```

Eso es todo, como pueden ver es una clase muy sencilla que lo único que hace es mantener el archivo recibido a través del formulario, y regresar el **content-type** del archivo.

Ahora haremos la declaración de este. Vamos al archivo "**struts.xml**" y dentro del paquete "**struts-interceptores**" que creamos el inicio del tutorial declaramos un nuevo "**action**" llamado "**carga-archivo**" y que será implementado por la clase "**com.javatutoriales.interceptores.actions.CargaArchivo**":

```
<package name="struts-interceptores" extends="struts-default">
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
</action>
</package>
```

Este "**action**" tendrá dos **results**, el primero, "**success**" enviará a la página "**resultado.jsp**", y el segundo será un resultado de tipo "**input**" que nos regresará nuevamente a nuestro formulario:

```
<action name="carga-archivo"  
class="com.javatutoriales.interceptores.actions.CargaArchivo">  
<result>/parametros/resultado.jsp</result>  
<result name="input">/parametros/index.jsp</result>  
</action>
```

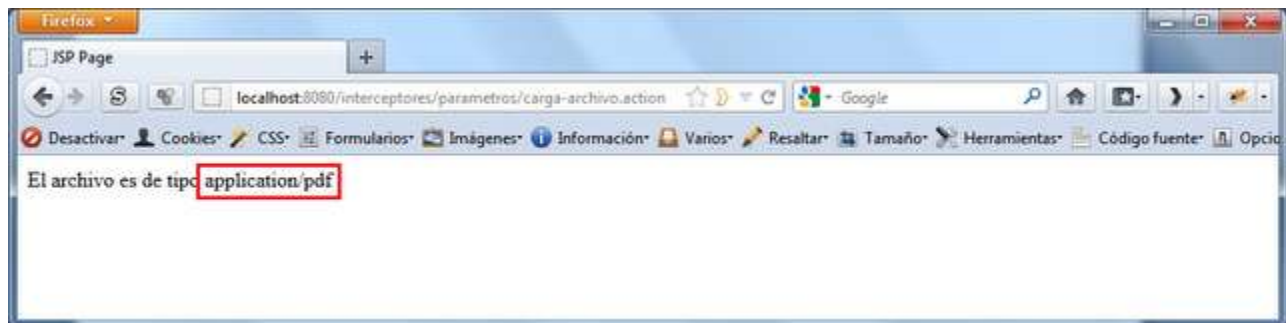
Ya está lista la primer parte del ejemplo. Ahora ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/interceptores/parametros/index.jsp>

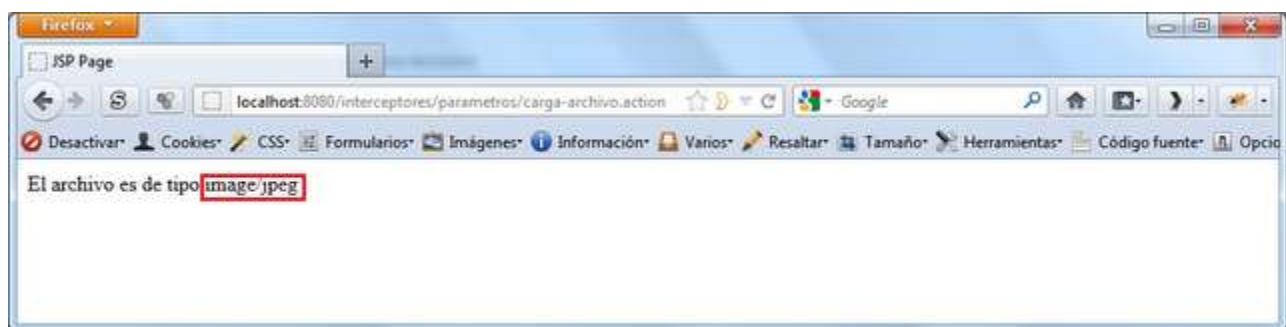
Con lo que deberemos ver nuestro formulario:



Seleccionamos un archivo de cualquier tipo, en este caso yo seleccionaré un archivo **.pdf**, y presionamos el botón para enviar el formulario. Al hacer esto debemos ver la siguiente salida:



Como podemos ver, el **content-type** del archivo es "**application/pdf**". Si ahora repetimos la operación seleccionando una imagen de tipo **jpg**, obtendremos la siguiente salida:



Podemos ver que ahora el archivo es de tipo "**image/jpeg**".

Podríamos [estar](#) todo el día subiendo archivos de distintos tipos para ver qué aparece en pantalla... pero eso evitaría que viéramos la parte del tutorial en la que tenemos que colocar los parámetros para configurar el interceptor ^^.

Struts 2 nos permite indicar de forma explícita qué interceptores utilizará cada **Action** de nuestra aplicación. Para esto podemos utilizar el elemento "**<interceptor-ref>**" que se coloca dentro de "**<action>**".

"**<interceptor-ref>**" tiene solamente un parámetro, "**name**", en el que se indica el nombre del interceptor que queremos configurar. En este caso, si miran la segunda tabla, podrán ver que el nombre del interceptor para la carga de archivos es "**fileUpload**":

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">

<interceptor-ref name="fileUpload">
</interceptor-ref>

<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Ahora que hemos indicado cuál interceptor vamos a parametrizar, el siguiente paso es realizar dicha parametrización. Los parámetros los colocamos como **pares nombre-valor** utilizando el elemento "**<param>**", dentro del interceptor que queremos parametrizar. El nombre lo indicamos con el atributo "**name**" del elemento anterior, y el valor lo indicamos como valor del elemento (o sea entre las dos etiquetas). Por ejemplo, para establecer el valor de un parámetro llamado "**nombre**" con el valor de "**Alex**", lo haríamos de la siguiente forma:

```
<param name="allowedTypes">image/png</param>
```

Si vamos a la documentación de la clase "[org.apache.struts2.interceptor.FileUploadInterceptor](#)", que es la que implementa el interceptor "**fileUpload**", podremos ver que puede recibir dos parámetros: "**maximumSize**" y "**allowedTypes**". El primero ya lo vimos en [el cuarto tutorial de la serie](#), así que ahora trabajaremos con el segundo.

Con las pruebas que hicimos anteriormente pudimos notar que podemos subir cualquier tipo de archivo al servidor, pero ¿qué pasaría si solo quisiéramos subir archivos de imágenes de tipo "**png**" y "**jpg**"? Para estos casos usamos el segundo parámetro, en el que debemos indicar los **content-types** de los archivos que serán aceptados por este **Action**. En este caso los **content-types** son: "**image/png**" y "**image/jpeg**".

Cuando tenemos más de un valor para este parámetro, los separamos usando una coma, por lo que el valor del parámetro queda de la siguiente forma:

```
<param name="allowedTypes">image/png,image/jpeg</param>
```

Y la configuración completa del **Action** así:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpeg</param>
</interceptor-ref>
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Si ahora ejecutamos nuestra aplicación y entramos a la siguiente dirección:

<http://localhost:8080/interceptores/parametros/index.jsp>

Veremos nuevamente nuestro formulario. Seleccionamos cualquier archivo que no sea una imagen **"png"** o **"jpg"**, en mi caso será nuevamente un archivo **PDF**, y presionamos el botón enviar. Es ese momento veremos... nada :|

Así es, no hay mensaje de error, ni ha aparecido el **content-type** del archivo, nada ¿Por qué ha pasado esto?...me alegra que pregunten ^_^.

En realidad si ha aparecido un mensaje de error, aunque un poco más discreto. Su vamos a la consola del **Tomcat** veremos un mensaje de error que dice algo así:

```
ADVERTENCIA: Content-Type not allowed: archivo "Ing de Sw Java tutoriales.pdf"
"upload_5010ddf2_137aed166ed__8000_00000002.tmp" application/pdf
```

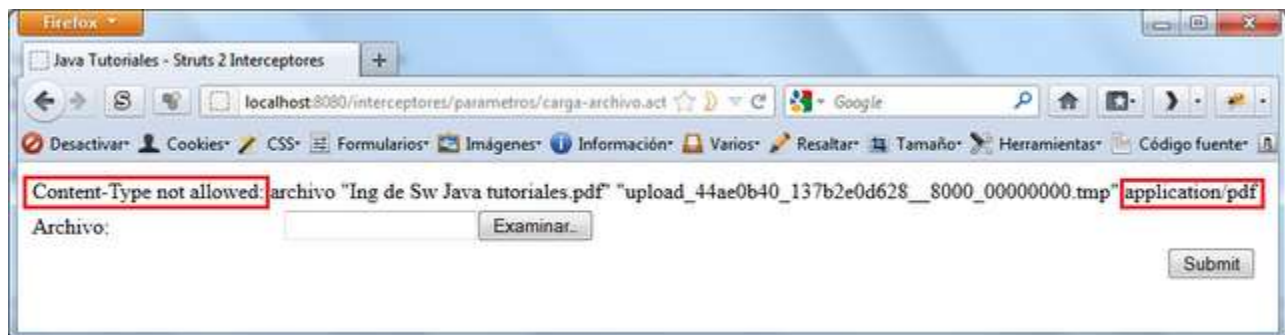
Anteriormente dije que "podemos configurar los interceptores que se aplicarán a un **Action**, directamente dentro del elemento **<action>**". Esto quiere **sólo se aplicarán los interceptores que indiquemos**. En el ejemplo anterior, lo que ocurre es que **sólo se aplica el interceptor "fileUpload" al Action, y no el resto de la cadena que teníamos por default**. O sea, no se aplican los interceptores que se encargan de enviar los parámetros al **Action**, ni los que se encargan de validar los datos, ni los que se encargan de mostrar los errores; sólo **"fileUpload"**.

Entonces ¿cómo hacemos para que el resto de los interceptores normales también se apliquen? Simplemente indicando que, además del interceptor **"fileUpload"**, se deben aplicar el resto de los interceptores del **"defaultStack"** y, esto es importante, se deben aplicar **después de "fileUpload"**. De esta forma **Struts 2** marcará que este interceptor ya se ha aplicado y no volverá a hacerlo. Si colocan la parametrización del interceptor **"fileUpload"** después del **"defaultStack"**, no verán ningún cambio con el comportamiento normal, así que lo repetiré, **coloquen siempre el "defaultStack" al final de los interceptores que hayan parametrizado o agregado**.

Por lo que la declaración, completa y correcta, del **Action** queda de la siguiente forma:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpg</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Si volvemos a ejecutar nuestra aplicación y entramos al formulario, al seleccionar un archivo que no sea una imagen **"jpg"** o **"png"** obtendremos el siguiente error:



Si seleccionamos una imagen debemos obtener el siguiente resultado:



Podemos ver que los parámetros se han aplicado correctamente al interceptor.

Existe una segunda forma de establecer los parámetros de los interceptores, la cual es útil cuando vamos a configurar muchos interceptores, y esta es usando como nombre del parámetro el nombre del interceptor, seguido de un punto y el nombre del parámetro que queremos configurar. Esto se hace dentro de la declaración del stack, por lo que la declaración anterior también puede quedar de la siguiente forma:

```
<interceptor-ref name="defaultStack">  
<param name="fileUpload.allowedTypes">image/png,image/jpg</param>  
</interceptor-ref>
```

Las dos dan los mismos resultados. Ahora que ya sabemos cómo configurar nuestros interceptores, veremos cómo agregar un interceptor que no se encuentra en la pila por default de **Struts 2**.

Uso de Interceptores que no están en el defaultStack

Struts 2 permite agregar a la ejecución de un **Action** interceptores que no estén en declarados en ninguna pila (y de igual forma configurarlos). En este ejemplo veremos cómo agregar dos interceptores muy útiles, **logger** y **timer**. Para no agregar un nuevo **<action>**, agregaremos este par de interceptores al que ya tenemos declarado.

Como su nombre lo indica, **logger** registra el inicio y el final de la ejecución de un **Action**. Agregarlo a su ejecución consiste solamente en dos cosas:

1. Conocer el nombre del interceptor (en este caso "**logger**")
2. Indicar en qué punto se debe comenzar a registrar la información.

El primer punto es sencillo, ya sabemos que el nombre del interceptor es "**logger**". El segundo punto necesita una pequeña explicación. El lugar en el que declaremos el interceptor es importante, ya que **su funcionamiento comenzará en el punto en el que lo coloquemos**. Esto quiere decir que si colocamos el interceptor antes del **Action** (o, como veremos, antes de los **results**), este registrará sólo la ejecución del **Action**. Si lo colocamos antes de la pila de interceptores, se registrará la ejecución de cada uno de los interceptores que conforman la pila además del **Action**. Por lo tanto, si colocamos la declaración de esta forma:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpeg</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>

<interceptor-ref name="logger" />

<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Se registrará sólo la ejecución del **Action**. Probémoslo ejecutando la aplicación. Al subir un archivo deberán de ver una salida parecida a esta en su consola:

```
4/06/2012 07:47:26 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Starting execution stack for action carga-archivo
4/06/2012 07:47:26 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Finishing execution stack for action carga-archivo
```

La salida deberá cambiar dependiendo de si el archivo que suben es válido o no, pero noten que prácticamente los únicos mensajes que aparecen son los de la ejecución del **Action "carga-archivo"**.

Si por el contrario, declaramos el interceptor de la siguiente forma:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">

<interceptor-ref name="logger" />

<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpeg</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Obtendremos la siguiente salida:

```
4/06/2012 07:58:11 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Starting execution stack for action carga-archivo
4/06/2012 07:58:11 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Finishing execution stack for action carga-archivo
```

(Si, se que se ve igual, pero eso es por el **Action** tan simple que estamos ejecutando)

Ahora agregaremos el interceptor "**timer**". Este interceptor se encarga de tomar el tiempo de ejecución de un **Action** y mostrarlo en la consola. Agregar el interceptor "**timer**" a nuestra aplicación es igual de sencillo.

Nota: Comentaré el interceptor "logger" para que la salida de la consola quede más limpia.

Podemos aplicar la misma lógica para este interceptor "**timer**": si lo colocamos antes del **Action** (de los **results**) sólo tomará el tiempo de ejecución del **Action**, si lo colocamos antes de los interceptores tomará el tiempo de ejecución del **Action** y del resto de los interceptores.

Por lo tanto, si declaramos el interceptor de esta forma:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpeg</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>

<interceptor-ref name="timer" />

<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Veremos más o menos la siguiente salida:

```
INFO: Executed action [carga-archivo!execute] took 16 ms.
```

Y si lo declaramos de esta forma:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">

<interceptor-ref name="timer" />

<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png,image/jpeg</param>
</interceptor-ref>
<interceptor-ref name="defaultStack"/>
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Veremos más o menos la siguiente salida:

```
INFO: Executed action [carga-archivo!execute] took 24 ms.
```

[Como](#) podemos ver, agregar un **Action** que no esté en la pila por default es, en realidad, muy sencillo.

Ahora, imaginen esta situación, ¿qué pasaría si quisiéramos aplicar uno de estos interceptores, digamos "**timer**" a todos los **Actions** de nuestra aplicación? Con lo que hemos visto hasta ahora, tendríamos que agregar la declaración del interceptor "**timer**" (y la correspondiente declaración del "**defaultStack**" en cada uno de los **Actions** de la aplicación. Esto, como podrán imaginarse, no es muy práctico, así que ahora veremos cómo anexar interceptores a la pila por default.

Agregando Interceptores a un Stack

Para evitar el tener que repetir declaraciones de interceptores que usaremos de manera frecuente en nuestros **Actions**, **Struts 2** permite que agreguemos los interceptores que queramos en una pila de interceptores, que puede estar formado por otras pilas, y luego declarar esta pila como la pila por default para los **Actions** de un paquete (un package de struts, no de Java) y de todos los paquetes que hereden de este.

Para [todo](#) trabajo relacionado con interceptores el archivo "**struts.xml**" tiene una sección especial dentro del elemento "**<interceptors>**". Dentro de este elemento podemos declarar interceptores nuevos (lo cual veremos cómo hacer en la siguiente parte del tutorial), o crear nuevas pilas. Para esto último usamos los elementos "**<interceptor-stack>**" y "**<interceptor-ref>**", que usamos hace un momento.

Para declarar una nueva pila le asignamos un nombre, que debe ser único dentro del paquete, usando el atributo "**name**" del elemento "**<interceptor-stack>**". El nombre que le daré a la nueva pila será "**defaultTimerStack**":

```
<interceptors>
<interceptor-stack name="defaultTimerStack">
</interceptor-stack>
</interceptors>
```

El siguiente paso es indicar qué interceptores conformarán esta pila. Para esto podemos indicar interceptores individuales (como "**timer**" o "**logger**") o indicar otras pilas de interceptores (como "**basicStack**" o "**defaultStack**"). Para ambos casos se utiliza el elemento "**<interceptor-ref>**", y se indica en su elemento "**name**" el nombre del interceptor o de la pila de interceptores al que hacemos referencia. Por ejemplo, para que la nueva pila sea una copia de la pila "**defaultStack**", colocamos la configuración de la siguiente forma:

```
<interceptors>
<interceptor-stack name="defaultTimerStack">
<interceptor-ref name="defaultStack" />
</interceptor-stack>
</interceptors>
```

Así, si aplicamos esta pila a un **Action** particular, como el **Action "carga-archivo"** que tenemos, debemos agregarla como en el ejemplo de la sección anterior, usando el elemento "**<interceptor-ref>**" dentro del elemento "**<action>**":

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<interceptor-ref name="defaultTimerStack" />
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jspl</result>
</action>
```

De esta forma se aplicará la pila por default a nuestro **Action**. Si ejecutan la aplicación, esta deberá comportarse de forma normal (subiendo cualquier tipo de archivo y sin mostrar ninguna salida en la consola).

El hacer esto nos permite que modifiquemos los interceptores de la pila (junto con sus atributos) y que se refleje el cambio en todos los **Actions** que utilicen esta pila. Por ejemplo, agreguemos a esta pila la parametrización del interceptor "**fileUpload**" para que sólo acepte imágenes de tipo "**png**":


```

<interceptors>
<interceptor-stack name="defaultTimerStack">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png</param>
</interceptor-ref>
<interceptor-ref name="defaultStack" />
</interceptor-stack>
</interceptors>

```

La definición de nuestro "**<action>**" permanece sin cambios.

Si ejecutamos nuevamente la aplicación, esta deberá aceptar sólo la carga de archivos cuyo **content-type** sea **"image/png"**.

Podemos agregar también los dos interceptores anteriores, **"logger"** y **"timer"**, recordando que las reglas anteriores (los interceptores se ejecutan en el orden en el que están declarados) continúan aplicando para las pilas propias:

```

<interceptors>
<interceptor-stack name="defaultTimerStack">
<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png</param>
</interceptor-ref>
<interceptor-ref name="defaultStack" />
<interceptor-ref name="timer" />
<interceptor-ref name="logger" />
</interceptor-stack>
</interceptors>

```

Si ahora ejecutamos nuestra aplicación, y cargamos una imagen válida, debemos ver la siguiente salida en la consola:

```

9/06/2012 07:40:37 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Starting execution stack for action carga-archivo
9/06/2012 07:40:37 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Finishing execution stack for action carga-archivo
9/06/2012 07:40:37 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Executed action [carga-archivo!execute] took 71 ms.

```

Podemos ver que la nueva pila de interceptores se aplica con la configuración adecuada a nuestro action.

Esto funciona bien si tenemos unos cuantos **Actions** en nuestra aplicación, o si la pila debe aplicarse sólo a unos cuantos **Actions**. Pero si esta debe aplicarse a todos o la mayor parte de los **Actions**, puede ser muy tedioso el tener que declarar dentro de cada uno que se desea aplicar ese stack. Para evitar esto, **Struts 2** proporciona una forma sencilla de decir que queremos que una pila sea aplicada por default a todos los **Actions** de nuestra aplicación.

Asignando una Pila de Interceptores por default para Nuestra Aplicación

Esta parte será realmente corta. Una vez que ya hemos creado y configurado una pila propia podemos indicar cuál de las pilas de nuestro paquete será la pila que debe usarse por default usando el elemento "**<default-interceptor-ref>**", en cuyo atributo **"name"** indicaremos cuál pila de interceptores deberá aplicarse por default a todos los **Actions** de la aplicación. En nuestro caso queremos que por default se aplique nuestra nueva pila de interceptores **"defaultTimerStack"**:

```
<default-interceptor-ref name="defaultTimerStack" />
```

Y... eso es todo. Como nuestra pila se aplica ahora por default, podemos dejar la declaración de nuestro action libre de esta información:

```
<action name="carga-archivo"
class="com.javatutoriales.interceptores.actions.CargaArchivo">
<result>/parametros/resultado.jsp</result>
<result name="input">/parametros/index.jsp</result>
</action>
```

Si volvemos a ejecutar nuestra aplicación, al subir una imagen válida debemos ver la siguiente salida en consola:

```
9/06/2012 08:22:11 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Starting execution stack for action carga-archivo
9/06/2012 08:22:11 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Finishing execution stack for action carga-archivo
9/06/2012 08:22:11 PM com.opensymphony.xwork2.util.logging.jdk.JdkLogger info
INFO: Executed action [carga-archivo!execute] took 70 ms.
```

Podemos ver, la pila se aplicó correctamente al único **Action** de la aplicación.

Ahora que conocemos los detalles del aspecto de configuración de los interceptores, veremos la última parte del tutorial, y la que seguramente la mayoría están esperando:

Creación de interceptores propios

Un interceptor es, como habíamos dicho, un componente que permite pre-procesar una petición y post-procesar una respuesta. O sea que podemos manipular una petición antes de que esta llegue al **Action**, y procesar una respuesta una vez que la ejecución del **Action** ha terminado pero antes de que sea regresada al usuario.

[Tener](#) la posibilidad de crear interceptores propios nos permite agregar bloques de lógica a nuestra aplicación de una forma más ordenada y reutilizable.

Para crear un interceptor debemos, como casi siempre en Java, implementar una interface. En este caso la interface a implementar es "**com.opensymphony.xwork2.interceptor.Interceptor**". Esta interface tiene tres métodos, y se ve de la siguiente forma:

```
public interface Interceptor
{
    void destroy();
    void init();
    String intercept(ActionInvocation invocation);
}
```

De estos tres métodos el que realmente nos interesa es "**intercept**", ya que es aquí en donde se implementa la lógica del interceptor. "**init**" se utiliza para inicializar los valores (conexiones a base de datos, valores iniciales, etc.) que usará el interceptor. "**destroy**" es su contraparte y se usa para liberar los recursos que hayan sido apartados en "**init**".

El argumento que recibe "**intercept**", un objeto de tipo "**ActionInvocation**", representa el estado de ejecución del **Action** y nos permitirá, entre otras cosas, obtener una referencia al **Action** que será ejecutado y modificar el resultado que será mostrado al usuario. Es gracias a este objeto que podemos agregar valores al **Action**, evitar que este se invoque, o permitir que continúe el flujo normal de la aplicación.

Para los casos en los que no usaremos los métodos "**init**" y "**destroy**", y no queramos proporcionar implementaciones vacías, también podemos extender de la clase "**com.opensymphony.xwork2.interceptor.AbstractInterceptor**"

Hagamos un primer ejemplo sencillo para asimilar estos conceptos. Nuestro primer interceptor indicará qué **Action** se está invocando junto con la fecha y la hora de dicha invocación; de paso también haremos que este nos salude ^_^.

Lo primero que haremos es crear un nuevo paquete llamado "**interceptores**" (o sea que le nombre completo del paquete será "**com.javatutoriales.interceptores.interceptores**"). Dentro de este paquete creamos una clase llamada "**InterceptorSaludo**" que implemente la interface "**com.opensymphony.xwork2.interceptor.Interceptor**":

```
public class InterceptorSaludo implements Interceptor
{

}
```

Debemos implementar los tres métodos (vistos anteriormente) de esta interface, por el momento permanecerán vacíos:

```
public class InterceptorSaludo implements Interceptor
{
    @Override
    public void destroy()
    {
    }

    @Override
    public void init()
    {
    }

    @Override
    public String intercept(ActionInvocation actionInvocation) throws Exception
    {
    }
}
```

En este caso no haremos nada en los métodos "**init**" y "**destroy**", sólo implementaremos el método "**intercept**". Iniciemos con la parte fácil, el saludo, para lo que usaremos el tradicional "**System.out.println**":

```
@Override
public String intercept(ActionInvocation actionInvocation) throws Exception
{
    System.out.println("Hola desarrollador");
}
```

Ahora, para obtener el nombre del **Action** que se está ejecutando usamos el objeto "**ActionContext**" que hemos usado en los tutoriales anteriores. Este objeto tiene un método "**get**" que permite obtener algunos datos referentes al **Action** en ejecución; debemos indicar qué valor queremos obtener usando algunas de las constantes que tiene el objeto "**ActionContext**", en particular a nosotros nos interesa "**ACTION_NAME**". Recordemos que este objeto es un "**Singleton**" por lo que debemos obtener esta instancia usando su método "**getContext**":

```
String actionName = (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);
```

Debemos castear el valor anterior a un String porque "**get**" regresa un "**Object**".

Para obtener la fecha y hora actual usamos un objeto tipo "**Date**" y para que se vea más presentable usaremos un objeto de tipo "**DateFormat**". Como no es el tema del tutorial, no explicaré lo que se está haciendo y sólo pondré el código:

```
String tiempoActual = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
DateFormat.MEDIUM).format(new Date());
```

Finalmente, imprimimos la información anterior en consola:

```
System.out.println("Ejecutando " + actionName + " a las " + tiempoActual);
```

Ya que hemos obtenido e impreso la información que nos interesaba, el siguiente paso es indicar que queremos que se continúe con la ejecución de la pila de interceptores (o dicho de otra forma, que no queremos que se interrumpa). Para esto, es necesario explicar dos cosas.

Lo primero que hay que saber es que el método **"intercept"** regresa un **"String"**, este **"String"** es el nombre del **result** que será mostrado al usuario.

Lo segundo que hay que saber es que no podemos regresar cualquier valor que queramos (bueno, en realidad sí, pero debemos hacerlo por una razón muy especial, veremos esto un poco más adelante) sino que debemos regresar el mismo valor que el **Action** dentro de su ejecución. ¿Y cómo obtenemos este valor?

Para obtener el nombre del **result** que regresa el método **"execute"** del **Action** debemos indicar que el interceptor se ha procesado correctamente y que se debe continuar con la ejecución normal de la aplicación (ya sea ejecutar el siguiente interceptor en la pila o ejecutar el **Action** en caso de que este sea el último interceptor), para lo cual debemos invocar el método **"invoke"** del **"ActionInvocation"** que el método recibe como parámetro. Este método regresa un **String** que es (así es, adivinaron) el nombre del **result**, por lo que lo único que debemos hacer es almacenar el valor que regresa este método para posteriormente volverlo a regresar o, como en mi caso, regresar directamente este valor:

```
return actionInvocation.invoke();
```

Por lo que nuestro método **"intercept"** completo queda de la siguiente forma:

```
@Override
public String intercept(ActionInvocation actionInvocation) throws Exception
{
    System.out.println("Hola desarrollador");
    String actionName = (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);
    String tiempoActual = DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
DateFormat.MEDIUM).format(new Date());

    System.out.println("Ejecutando " + actionName + " a las " + tiempoActual);

    return actionInvocation.invoke();
}
```

Si queremos realizar un proceso después de que el **Action** se haya terminado de ejecutar, debemos almacenar el valor regresado por la llamada a **"invoke"**, y realizar el post-procesamiento después de este punto. Por ejemplo, si queremos enviar un mensaje cuando el **Action** ha terminado de ejecutarse debemos dejar el método **"intercept"** de la siguiente forma:

```
@Override
```

```

public String intercept(ActionInvocation actionInvocation) throws Exception
{
    System.out.println("Hola desarrollador");
    String actionName = (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);
    String tiempoActual = DateFormat.getDateInstance(DateFormat.MEDIUM,
    DateFormat.MEDIUM).format(new Date());

    System.out.println("Ejecutando " + actionName + " a las " + tiempoActual);

    String resultado = actionInvocation.invoke();

    System.out.println("Gracias, regresa pronto :)");

    return resultado;
}

```

Y el interceptor completo de esta otra:

```

public class InterceptorSaludo implements Interceptor
{
    @Override
    public void destroy()
    {
    }

    @Override
    public void init()
    {
    }

    @Override
    public String intercept(ActionInvocation actionInvocation) throws Exception
    {
        System.out.println("Hola desarrollador");
        String actionName = (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);
        String tiempoActual = DateFormat.getDateInstance(DateFormat.MEDIUM,
        DateFormat.MEDIUM).format(new Date());

        System.out.println("Ejecutando " + actionName + " a las " + tiempoActual);

        String resultado = actionInvocation.invoke();

        System.out.println("Gracias, regresa pronto :)");

        return resultado;
    }
}

```

Ahora que ya tenemos lista la clase que implementa el interceptor, debemos configurarlo. Para esto vamos nuevamente a nuestro archivo "**struts.xml**". Si recuerdan, anteriormente dijimos que todo lo que tiene que ver con configuración de los interceptores se hace dentro de la sección "<interceptors>" de este archivo.

Para declarar un nuevo interceptor usamos el elemento..."**<interceptor>**". Este elemento tiene dos atributos: "**name**", en el que se indica cuál será el nombre que se use para hacer referencia a este interceptor, y "**class**", en el que se indica cuál clase implementa la funcionalidad de este interceptor.

En nuestro ejemplo, llamaremos al interceptor "**saludo**", y la clase que lo implementa es "**com.javatutoriales.interceptores.interceptores.InterceptorSaludo**". La declaración completa del interceptor, dentro de la sección "**<interceptors>**", queda de la siguiente forma:

```
<interceptor name="saludo" class="com.javatutoriales.interceptores.interceptores.InterceptorSaludo" />
```

Ahora lo único que debemos hacer es agregar este interceptor a la pila "**defaultTimerStack**" que creamos anteriormente. Recuerden que el interceptor se ejecutará en la posición en la que lo coloquen dentro de la pila; para este ejemplo pueden colocarlo en cualquier posición que deseen, en mí caso será el segundo interceptor que se ejecute:

```
<interceptor-stack name="defaultTimerStack">
  <interceptor-ref name="fileUpload">
    <param name="allowedTypes">image/png</param>
  </interceptor-ref>
```

```
<interceptor-ref name="saludo" />
```

```
<interceptor-ref name="defaultStack" />
<interceptor-ref name="timer" />
<interceptor-ref name="logger" />
</interceptor-stack>
```

Ahora ejecutamos nuestra aplicación, y al ejecutar nuestro **Action** debemos ver la siguiente salida en la consola:

```
Hola desarrollador
Ejecutando carga-archivo a las 23/06/2012 11:46:33 PM
Gracias, regresa pronto :)
```

Esto quiere decir que el interceptor se ejecutó de forma correcta ^_^

Para terminar con este tutorial, veremos un pequeño ejemplo que combina todos los elementos que hemos visto:

Creación de un Interceptor de Verificación de Sesión Válida

Una de las cosas más comunes y útiles que podemos, y queremos, hacer con interceptores propios es verificar que un usuario pueda ejecutar los **Actions** de nuestra aplicación, pero solamente si es un usuario válido, o sea, si existe una sesión creada para este usuario dentro del sistema.

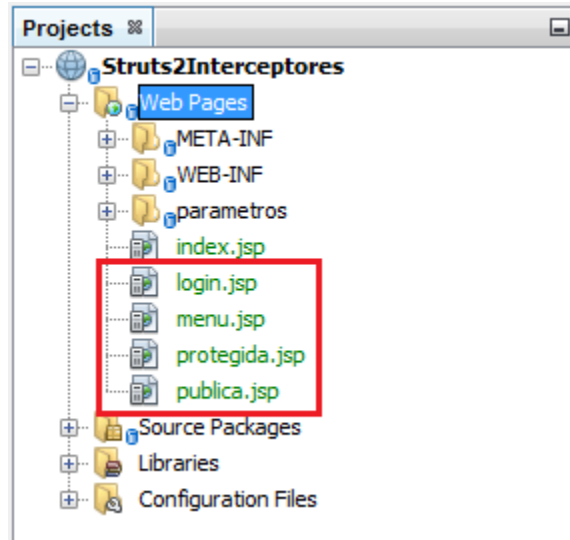
Para hacer esto debemos pensar básicamente en dos cosas:

- ¿Cuáles **Actions** dentro de la aplicación deben estar protegidos?
- ¿Cuáles **Actions** deben ser públicos?
- ¿A dónde debe dirigirse al usuario en caso de que este no se haya logueado dentro del sistema aún?

Para responder a estas preguntas diremos que normalmente queremos que casi todos los **Actions** de la aplicación estén protegidos....¿casi? Si, casi todos, debemos dejar al menos un **Action** sin proteger (público): el **Action** para el login de la aplicación. Claro, podemos dejar públicos tantos **Actions** como queramos, pero necesitamos al menos un punto de entrada a la aplicación para que el usuario pueda loguearse.

Si el usuario intenta ejecutar un **Action** que esté protegido, y aún no se ha logueado, podemos enviarlo a una página de error o directamente al formulario de logueo. Nosotros haremos esto último.

Ahora vamos al código del ejemplo. Lo primero que haremos es crear cuatro páginas: "**menu.jsp**", "**login.jsp**", "**publica.jsp**", y "**protegida.jsp**", en la raíz de las páginas web:



"**publica.jsp**" y "**protegida.jsp**" serán el resultado de la ejecución de un par de **Actions**, "**login.jsp**" será un formulario para que el usuario pueda ingresar su usuario y contraseña de acceso al sitio; finalmente "**menú.jsp**" será una página que mantendrá dos ligas, cada una a uno de los **Actions** de nuestro interés.

Las dos primeras páginas serán muy sencillas y sólo contendrán un mensaje indicando si es un recuso protegido o público. El contenido de "**publica.jsp**" es el siguiente:

```
<h1>Esta página puede ser vista por cualquier persona</h1>
```

El contenido de "**protegida.jsp**" es el siguiente:

```
<h1>Esta página sólo puede ser vista por los usuarios autorizados</h1>
```

En "**menú.jsp**" tendremos las ligas a dos **Actions** que crearemos en un momento, cada a una a uno de los recursos que queremos proteger o mantener públicos:

```
<ul>
<li><s:a action="protegido">Action Protegido</s:a></li>
<li><s:a action="publico">Action Público</s:a></li>
</ul>
```

No olviden agregar el taglib de **Struts 2** a la página anterior, ya que estamos haciendo uso de sus etiquetas.

El contenido de "**login.jsp**" es un poco más interesante, un formulario que permite ingresar dos campos, un nombre de usuario y una contraseña; además del botón que permite enviar estos datos. También agregaremos un elemento que nos mostrará los mensajes de error en caso de que algo malo ocurra:

```
<s:actionerror />
<s:form action="login">
<s:textfield name="username" label="Username" />
<s:password name="password" label="Password" />
<s:submit value="Ingresar" />
</s:form>
```

El siguiente paso es crear el **Action** para que el usuario pueda iniciar una sesión en el sistema. Para indicar que el usuario se ha logueado en el sistema, colocaremos un atributo preestablecido en la sesión, usando una de las técnicas que aprendimos en la cuarta parte de la serie.

***Nota: Normalmente crearíamos un objeto "Usuario" con algunas propiedades del cliente que va a loguearse, y haríamos todo el manejo de los datos de la sesión utilizando este objeto. Además haríamos una verificación de los permisos del usuario para saber a qué recursos tiene acceso (autorización). En este tutorial sólo agregaremos una cadena como indicador de la sesión para poder centrarnos en el objetivo del tutorial (los interceptores).**

Creemos una nueva clase llamada **"LoginAction"** en el paquete **"actions"**. Esta clase debe extender de **"ActionSupport"**:

```
public class LoginAction extends ActionSupport
{
}
```

Colocaremos un par de atributos, con sus correspondientes **setters**, para almacenar el nombre del usuario y su contraseña:

```
public class LoginAction extends ActionSupport
{
    private String username;
    private String password;

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

    public void setUsername(String username)
    {
        this.username = username;
    }
}
```

Para terminar con este **Action**, sobre-escribiremos su método **"execute"** para agregar al usuario a la sesión (en un sistema real debemos verificar a este usuario contra algún almacén de datos como una base de datos o un directorio ldap, pero aquí no lo haremos por... por lo mismo que dice la nota anterior ^_^):

```
@Override
public String execute() throws Exception
{
    HttpServletRequest request = ServletActionContext.getRequest();
    HttpSession session = request.getSession();

    session.setAttribute("usuario", username);

    return SUCCESS;
}
```

Nuestra clase **"LoginAction"** completa queda de la siguiente forma:


```

public class LoginAction extends ActionSupport
{
    private String username;
    private String password;

    @Override
    public String execute() throws Exception
    {
        HttpServletRequest request = ServletActionContext.getRequest();
        HttpSession session = request.getSession();

        session.setAttribute("usuario", username);

        return SUCCESS;
    }

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

    public void setUsername(String username)
    {
        this.username = username;
    }
}

```

Ahora necesitamos configurar estos elementos en el archivo "**struts.xml**". Lo primero que haremos es configurar los dos **Actions** con sus **results** correspondientes:

```

<action name="publico">
<result>/publica.jsp</result>
</action>

```

```

<action name="protegido">
<result>/protegida.jsp</result>
</action>

```

```

<action name="login" class="com.javatutoriales.interceptores.actions.LoginAction">
<result>/menu.jsp</result>
<result name="input">/login.jsp</result>
</action>

```

Podemos ver que en el caso de "**LoginAction**", cuando el usuario ingrese este será enviado de nuevo al menú, y si algo sale mal será enviado nuevamente el formulario de login.

Como el **result** para el caso del login será usado para todos los **Actions** de la aplicación, debemos colocarlo de forma global (recuerden que este elemento se configura después de los interceptores pero antes de los **Actions**, dentro del archivo de configuración):

```

<global-results>
<result name="login">/login.jsp</result>
</global-results>

```

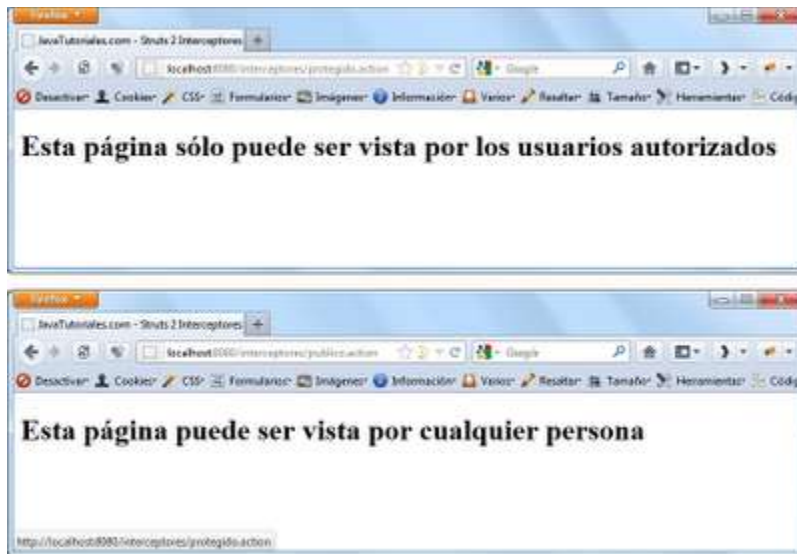
Si ahora ejecutamos la aplicación, y entramos a la siguiente dirección:

<http://localhost:8080/interceptores/menu.jsp>

Debemos ver nuestro menú:



Cuando entremos a cada una de las ligas, debemos ver las páginas correspondientes:



Podemos ver que hasta aquí la aplicación sigue funcionando de la misma forma (no se ha restringido el acceso a nada). Ahora crearemos el interceptor que nos permitirá proteger nuestras páginas.

Primero debemos crear una clase llama "**InterceptorAcceso**" en el paquete "**interceptores**" de la aplicación. En esta ocasión, esta clase extenderá de "**AbstractInterceptor**" y sólo sobre-escribirá su método "**intercept**":

```
public class InterceptorAcceso extends AbstractInterceptor
{
    @Override
    public String intercept(ActionInvocation ai) throws Exception
    {
    }
}
```

Por ahora sólo usaremos el método "**intercept**", en el cual verificaremos que exista la propiedad llamada "**usuario**" en la sesión, la cual se establece en el **Action** que se encarga de realizar el logueo del usuario. Si no encuentra este objeto entonces supondremos que el usuario aún no se ha logueado y lo enviaremos al formulario de ingreso al sistema:

```

@Override
public String intercept(ActionInvocation actionInvocation) throws Exception
{
    String result = Action.LOGIN;

    if (actionInvocation.getInvocationContext().getSession().containsKey("usuario"))
    {
        result = actionInvocation.invoke();
    }

    return result;
}

```

Primero asumimos que el usuario no se ha logueado, así que usamos la constante "**LOGIN**" de la interface "**Action**" como valor inicial de la variable que regresaremos en el interceptor. Verificamos, usando el mapa de sesión que obtenemos a través del objeto "**ActionInvocation**", si el existe un usuario en sesión. Si el usuario existe, entonces ejecutamos el **Action** y regresamos el valor que obtengamos de la ejecución de su método "**execute**", en caso contrario este será enviado al formulario de login.

Ya teniendo nuestro interceptor podemos configurarlo en el archivo "**struts.xml**" de la forma en la que lo hicimos hace un momento. El nombre del interceptor será "**sesionValida**":

```

<interceptor name="sesionValida" class="com.javatutoriales.interceptores.interceptores.InterceptorAcceso" />

```

Y colocamos el interceptor en la pila que se ejecuta por default (junto con el resto de los interceptores que hemos configurado en el tutorial) para que sea aplicado a todos los **Actions** de la aplicación. Podemos colocar el interceptor a cualquier altura que queramos de la pila:

```

<interceptor-stack name="defaultTimerStack">

<interceptor-ref name="sesionValida" />

<interceptor-ref name="fileUpload">
<param name="allowedTypes">image/png</param>
</interceptor-ref>
<interceptor-ref name="saludo" />
<interceptor-ref name="defaultStack" />
<interceptor-ref name="timer" />
<interceptor-ref name="logger" />
</interceptor-stack>

```

Ahora podemos ejecutar nuestra aplicación y entrar a la siguiente dirección:

<http://localhost:8080/interceptores/menu.jsp>

Con lo que veremos el menú de hace un momento. Cuando demos clic en el enlace para ir al **Action** protegido, la petición pasará primero por el interceptor "**sesionValida**", este interceptor verá que el usuario aún no ha iniciado sesión y lo enviará el formulario de acceso:

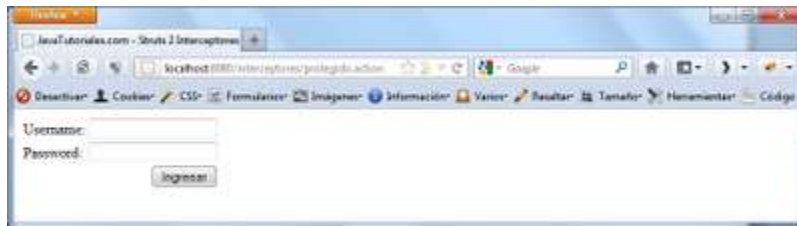


Con lo que podemos comprobar que la verificación está funcionando correctamente.

Si ahora regresamos y hacemos clic en el enlace para ir al **Action** público podremos comprobar que...



También nos envía al formulario de acceso °_°. Bueno, parece que algo está pasando. No importa. Si ahora ingresamos nuestro nombre de usuario y contraseña, y hacemos clic en el botón **"Ingresar"** (no importa los datos que coloquen, recuerden que no se están validando), deberemos ser enviados a...



Bueno, una vez más estamos siendo enviados al formulario de acceso, aun cuando ya hemos ingresado nuestros datos para entrar al sistema. ¿Qué es lo que está pasando?... me alegra que pregunten ^_^.

Lo que ocurre es que, como debe de ser, todas las peticiones (incluyendo la petición de acceso al sitio) están pasando por el interceptor **"sesionValida"**, el cual verifica que la sesión exista. Como en realidad la petición nunca llega al **Action "LoginAction"**, nunca iniciamos una sesión en el sistema (el filtro no tiene forma de saber que esta es la finalidad de este **Action**), y por lo tanto nunca podemos pasar a través del interceptor (de hecho si revisan el **Action** de la carga de imágenes, este también ha dejado de funcionar^_^).

¿Qué podemos para que esto no ocurra? Bueno, un paso muy importante, como acabamos de ver, es indicarle a nuestro interceptor cuáles **Actions no debe filtrar**, o dicho de otra forma: cuáles **Actions** deben poder ser accesibles aún sin una sesión de usuario válida. Para hacer eso le pasaremos a nuestro interceptor, como parámetros, la lista de los **Actions** que no debe filtrar.

Regresamos a nuestra clase **"InterceptorAcceso"** y agregamos un atributo, de tipo **"String"**, llamado **"actionsPublicos"**, con su correspondiente **setter**:

```
private String actionsPublicos;  
  
public void setActionsPublicos(String actionsPublicos)  
{  
    this.actionsPublicos = actionsPublicos;  
}
```

Este atributo recibirá una lista separada por comas con los nombres de los **Actions** que no serán filtrados. Como este parámetro es una cadena, la transformaremos en una lista para usar los métodos de la interface **"List"** para saber si el **Action** que se esté ejecutando es protegido o no. Esta transformación la haremos en el método **"init"**, por lo que lo sobre-escribimos de la siguiente forma:

```
private List<String> actionsSinFiltrar = new ArrayList<String>();

@Override
public void init()
{
    actionsSinFiltrar = Arrays.asList(actionsPublicos.split(","));
}
```

Finalmente, modificamos la condición del método "**execute**", para verificar si el nombre del **Action** que se está ejecutando actualmente está en la lista de los **Actions** que no deben ser filtrados:

```
String actionActual = (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);

if (actionInvocation.getInvocationContext().getSession().containsKey("usuario") ||
actionsSinFiltrar.contains(actionActual))
{
    result = actionInvocation.invoke();
}
```

Nuestra clase "**InterceptorAcceso**" completa queda de la siguiente forma:

```
public class InterceptorAcceso extends AbstractInterceptor
{
    private String actionsPublicos;
    private List<String> actionsSinFiltrar = new ArrayList<String>();

    @Override
    public void init()
    {
        actionsSinFiltrar = Arrays.asList(actionsPublicos.split(","));
    }

    @Override
    public String intercept(ActionInvocation actionInvocation) throws Exception
    {
        String result = Action.LOGIN;

        String actionActual =
        (String)ActionContext.getContext().get(ActionContext.ACTION_NAME);

        if (actionInvocation.getInvocationContext().getSession().containsKey("usuario")
|| actionsSinFiltrar.contains(actionActual))
        {
            result = actionInvocation.invoke();
        }

        return result;
    }

    public void setActionsPublicos(String actionsPublicos)
    {
        this.actionsPublicos = actionsPublicos;
    }
}
```

Ahora, para terminar, regresamos al archivo "**struts.xml**" e indicamos cuáles son los actions que queremos dejar públicos, pasándolos como el parámetro "**actionsPublicos**" del interceptor "**sesionValida**":

```
<interceptor-ref name="sesionValida">
<param name="actionsPublicos">login,publico</param>
</interceptor-ref>
```

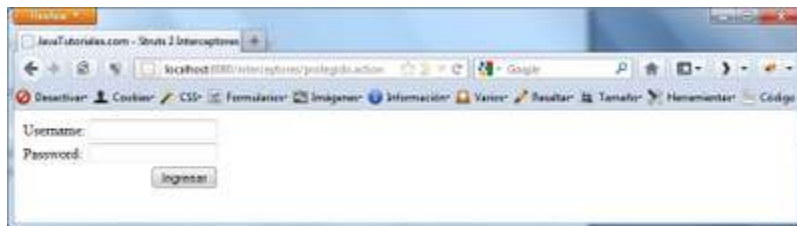
Y esto es todo (lo prometo ^_^). Ahora, volvemos a ejecutar nuestra aplicación. Ingresamos a la siguiente dirección:

<http://localhost:8080/interceptores/menu.jsp>

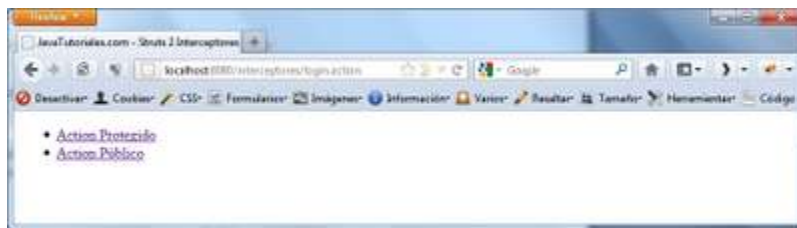
Cuando ingresemos en la liga "**Action Público**" debemos ver el siguiente mensaje:



Hasta ahora vamos bien. Ahora, cuando entremos a la liga "**Action Protegido**" debemos ver el formulario de login:



Cuando ingresemos un usuario y una contraseña debemos ser reenviados al menú:



Van dos de dos ^_^!. Si ahora volvemos a entrar a la liga "**Action Protegido**" debemos ver finalmente el mensaje que hemos estado esperando:



Con esto comprobamos que el interceptor está funcionando de forma adecuada (también el **Action** para la carga de las imágenes está funcionando correctamente bajo este esquema) ^_^.

Quiero que noten una cosa: con esto hemos protegido los **Actions**, sin embargo las páginas **JSPs** aún siguen siendo accesibles de forma libre. Para evitar esto existen varias alternativas, de hecho algunos frameworks para plantillas como [tiles](#) o [sitemesh](#) son muy útiles para estos casos. Pero si no usamos un framework de plantillas aún podemos proteger nuestras **JSPs** de varias formas, sólo las listaré y no entraré en detalles ^_^:

- Hacer que el filtro de **Struts 2** también procese las peticiones que lleguen con extensión "**.jsp**" (recuerden que por default procesa las peticiones que llegan con "**.action**" o que llegan sin extensión) e indicar en el filtro, además de los **Actions** públicos, las páginas que también serán accesibles libremente.
- Colocar los **JSPs** dentro del directorio **WEB-INF**, el cual no es regresado por el servidor. De esta forma ninguna página puede ser accedida por el navegador, esto significa que para ver las **JSPs** que deben ser públicas deberemos colocar "**<actions>**" vacíos (sin atributo "**class**") que en su **result "success"** regresen esta página. También se pueden dejar las **JSPs** públicas fuera del directorio **WEB-INF** y las protegidas dentro.
- Colocar una validación en cada una de las **JSPs** (por medio de tags) para que ellas deciden si deben mostrarse o no. También pueden ser colocadas en una página que sirva como encabezado a todas las páginas del sitio, e incluirlas usando la directiva apropiada, así sólo debemos colocar en un lugar este código.

Cada una de estas alternativas tiene ventajas y desventajas, así que elijan la que más se ajuste a sus necesidades.