

Struts

2ª edición

Struts

2ª edición

Antonio J. Martín Sierra



<TITULO TITULO TITULO TITULO TITULO TITULO TITULO TITULO>

© <AUTOR AUTOR AUTOR AUTOR AUTOR>

© De la edición RA-MA 2007

MARCAS COMERCIALES. Las marcas de los productos citados en el contenido de este libro (sean o no marcas registradas) pertenecen a sus respectivos propietarios. RA-MA no está asociada a ningún producto o fabricante mencionado en la obra, los datos y los ejemplos utilizados son ficticios salvo que se indique lo contrario.

RA-MA es marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa.

Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa ni de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente, reprodujeran o plagiaran, en todo o en parte, una obra literaria, artística o científica.

Editado por:

RA-MA Editorial

Calle Jarama, 3A, Polígono Industrial Igarsa

28860 PARACUELLOS DE JARAMA, Madrid

Teléfono: 91 6584280

TeleFax: 91 6628139

Correo electrónico: editorial@ra-ma.com

Internet: www.ra-ma.es y www.ra-ma.com

ISBN: 84-7897-XXX-X

BAN: 978847897XXXX

Depósito Legal: M-XXXXX-2007

Autoedición: Autor

Filmación e impresión: <IMPRESA>

Impreso en España

A mi sobrino Hugo.

ÍNDICE

PRÓLOGO.....	13
CAPÍTULO 1. LA ARQUITECTURA MODELO VISTA	
CONTROLADOR	19
1.1 EL PATRÓN MVC.....	20
1.1.1 EL CONTROLADOR	21
1.1.2 LA VISTA	22
1.1.3 EL MODELO	22
1.2 FUNCIONAMIENTO DE UNA APLICACIÓN MVC	23
CAPÍTULO 2. EL FRAMEWORK STRUTS	35
2.1 FUNDAMENTOS DE STRUTS	36
2.2 COMPONENTES DE STRUTS	37
2.2.1 ARCHIVOS DE CONFIGURACIÓN	37
2.2.2 EL API DE STRUTS	39
2.2.3 LIBRERÍAS DE ACCIONES JSP	41
2.3 FUNCIONAMIENTO DE UNA APLICACIÓN STRUTS.....	42
CAPÍTULO 3. DESARROLLO DE UNA APLICACIÓN CON STRUTS.....	45
3.1 DESCARGA E INSTALACIÓN DEL FRAMEWORK STRUTS.....	45
3.2 APLICACIÓN PRÁCTICA PARA VALIDACIÓN Y REGISTRO DE	
USUARIOS	47
3.2.1 FUNCIONAMIENTO DE LA APLICACIÓN.....	48
3.2.2 ESQUEMA DE LA APLICACIÓN	48
3.2.3 CONSTRUCCIÓN DE LA APLICACIÓN	50

3.2.3.1 Estructura de una aplicación Web Struts	50
3.2.3.2 Registro del servlet ActionServlet	52
3.2.3.3 Captura de datos de usuario: Las clases ValidacionForm y RegistroForm	53
3.2.3.4 Implementación del Modelo	56
3.2.3.5 Procesamiento de peticiones: Las clases ValidarAction y RegistrarAction	59
3.2.3.6 Objetos forward globales	64
3.2.3.7 Las páginas de la vista	65
CAPÍTULO 4. ANÁLISIS DEL API DE STRUTS	87
4.1 PROCESAMIENTO DE UNA PETICIÓN: CLASES ACTIONSERVLET Y REQUESTPROCESSOR	87
4.2 CLASES DE ACCIÓN	91
4.2.1 CLASE DISPATCHACTION	92
4.2.2 CLASE LOOKUPDISPATCHACTION	106
4.2.3 CLASE MAPPINGDISPATCHACTION	112
4.2.4 CLASE ACTIONFORM	125
4.2.4.1 Ciclo de vida de un ActionForm	125
4.2.5 ACTIONERRORS Y ACTIONMESSAGE	128
4.3 CONTROL DE EXCEPCIONES EN STRUTS	133
4.3.1 GESTIÓN DECLARATIVA DE EXCEPCIONES	134
4.3.2 IMPLEMENTACIÓN DE LA GESTIÓN DECLARATIVA DE EXCEPCIONES	135
4.3.3 CLASES PERSONALIZADAS PARA LA GESTIÓN DE EXCEPCIONES	139
CAPÍTULO 5. LIBRERÍAS DE ACCIONES JSP DE STRUTS	143
5.1 LIBRERÍA BEAN	143
5.1.1 WRITE	144
5.1.2 PARAMETER	145
5.1.3 COOKIE	145
5.1.4 HEADER	146
5.1.5 MESSAGE	146
5.1.6 DEFINE	147
5.1.7 PAGE	149
5.1.8 SIZE	149
5.2 LIBRERÍA LOGIC	150
5.2.1 EQUAL	151
5.2.2 NOTEQUAL	154

5.2.3 GREATEREQUAL, LESSEQUAL, GREATERTHAN Y LESSTHAN	154
5.2.4 MATCH	154
5.2.5 NOMATCH	155
5.2.6 FORWARD	155
5.2.7 REDIRECT	155
5.2.8 ITERATE.....	157

CAPÍTULO 6. VALIDACIÓN DE DATOS DE USUARIO.....171

6.1 COMPONENTES DE UN VALIDADOR	171
6.1.1 PLUG-IN VALIDATOR	172
6.1.2 ARCHIVOS DE CONFIGURACIÓN	172
6.1.2.1 validator-rules.xml.....	173
6.1.2.2 validation.xml	174
6.1.3 CLASE VALIDATORFORM.....	175
6.1.4 ARCHIVO APPLICATIONRESOURCE.PROPERTIES.....	176
6.2 UTILIZACIÓN DE VALIDADORES	177
6.2.1 CREACIÓN DE LA CLASE VALIDATORFORM.....	178
6.2.2 DEFINICIÓN DE LOS CRITERIOS DE VALIDACIÓN	178
6.2.3 HABILITACIÓN DE LA VALIDACIÓN EN CLIENTE.....	180
6.2.4 MENSAJES DE ERROR.....	182
6.3 VALIDADORES PREDEFINIDOS DE STRUTS	183
6.3.1 MINLENGTH.....	183
6.3.2 MAXLENGTH.....	184
6.3.3 BYTE, SHORT, INTEGER, LONG, FLOAT Y DOUBLE.....	185
6.3.4 INTRANGE	185
6.3.5 FLOATRANGE Y DOUBLERANGE.....	186
6.3.6 DATE	186
6.3.7 MASK.....	187
6.3.8 EMAIL.....	187
6.4 MENSAJES DE ERROR PERSONALIZADOS.....	187
6.5 VALIDACIONES PERSONALIZADAS.....	193
6.5.1 SOBRESCRITURA DEL MÉTODO VALIDATE().....	193
6.5.2 CREACIÓN DE VALIDADORES PERSONALIZADOS	195
6.5.2.1 Implementación del método de validación	196
6.5.2.2 Registro del validador	198
6.5.2.3 Mensajes de error.....	200
6.5.2.4 Utilización del validador.....	200

CAPÍTULO 7. UTILIZACIÓN DE PLANTILLAS.....201

7.1 CONFIGURACIÓN DE LA APLICACIÓN PARA EL USO DE PLANTILLAS	202
7.2 CREACIÓN DE UNA APLICACIÓN STRUTS BASADA EN PLANTILLAS.....	202
7.2.1 CREACIÓN DE LA PLANTILLA.....	203
7.2.2 CREACIÓN DE PIEZAS DE CONTENIDO	205
7.2.3 CREACIÓN DE LAS PÁGINAS DE APLICACIÓN.....	207
7.2.4 DECLARACIÓN DE LA PLANTILLA	207
7.2.5 INCLUSIÓN DE PÁGINAS DE CONTENIDO	208
7.3 DEFINICIONES	209
7.3.1 CREACIÓN DE UNA DEFINICIÓN	210
7.3.1.1 Definiciones base.....	210
7.3.1.2 Definiciones derivadas.....	211
7.3.2 PÁGINAS DE APLICACIÓN.....	212

CAPÍTULO 8. STRUTS 2223

8.1 COMPONENTES DE STRUTS 2	224
8.1.1 FILTERDISPATCHER	224
8.1.2 INTERCEPTORES	225
8.1.3 ACTION	226
8.1.4 LIBRERÍAS DE ACCIONES	227
8.1.5 ARCHIVO DE CONFIGURACIÓN STRUTS.XML	227
8.1.5.1 Paquetes	228
8.1.5.2 Herencia de paquetes	229
8.1.5.3 Modularidad de ficheros de configuración	237
8.2 BENEFICIOS DEL USO DE STRUTS 2.....	237
8.3 CREACIÓN DE UNA APLICACIÓN DE EJEMPLO DE STRUTS 2	238
8.3.1 DESCARGA DEL PAQUETE DE DISTRIBUCIÓN DE STRUTS 2.....	238
8.3.2 REQUERIMIENTOS SOFTWARE	239
8.3.3 DESCRIPCIÓN DE LA APLICACIÓN	239
8.3.4 ESTRUCTURA DE DIRECTORIOS DE LA APLICACIÓN.....	240
8.3.5 REGISTRO DE FILTERDISPATCHER.....	241
8.3.6 IMPLEMENTACIÓN DE LA CLASE DE ACCIÓN	242
8.3.7 REGISTRO DE LA CLASE DE ACCIÓN	243
8.3.8 REGLAS DE NAVEGACIÓN	243
8.3.8.1 Acción por defecto.....	244
8.3.9 VISTAS	244

8.4 UTILIZACIÓN DE INTERCEPTORES	245
8.4.1 DECLARACIÓN DEL INTERCEPTOR	246
8.4.2 ASIGNACIÓN DE UN INTERCEPTOR A UNA ACCIÓN	248
8.4.3 INYECCIÓN DE DEPENDENCIA	248
8.4.4 INTERCEPTORES PERSONALIZADOS	258
8.4.4.1 El método intercept()	258
8.5 LA LIBRERÍA DE ACCIONES STRUTS-TAGS	267
8.5.1 EL STACK DE OBJETOS	267
8.5.2 ACCIONES DE MANIPULACIÓN DE DATOS.....	269
8.5.2.1 bean.....	269
8.5.2.2 param	269
8.5.2.3 property.....	269
8.5.2.4 push.....	270
8.5.2.5 set.....	270
8.5.3 ACCIONES DE CONTROL.....	270
8.5.3.1 if.....	271
8.5.3.2 iterator.....	271
8.5.4 ACCIONES UI	272
8.5.4.1 form	273
8.5.4.2 textfield.....	273
8.5.4.3 password	275
8.5.4.4 textarea.....	275
8.5.4.5 submit	275
8.5.4.6 radio	277
8.5.4.7 checkbox	279
8.5.4.8 CheckboxList	279
8.5.4.9 select	279
8.5.4.10 actionerror.....	286
8.6 VALIDADORES	286
8.6.1 VALIDADORES PREDEFINIDOS	286
8.6.2 UTILIZACIÓN DE VALIDADORES EN UNA APLICACIÓN	288
8.6.3 VALIDACIÓN MEDIANTE ANOTACIONES	291
8.6.3.1 Tipos de anotaciones de validación	294
APÉNDICE A. EL LENGUAJE DE EXPRESIONES DE JSP	305
EXPRESIONES EL	306
ACCESO A VARIABLES DE ÁMBITO.....	307

OBJETOS IMPLÍCITOS EL	308
OPERADORES EL	310

APÉNDICE B. LA LIBRERÍA DE ACCIONES ESTÁNDAR

DE JSP (JSTL).....	313
---------------------------	------------

INSTALACIÓN DE JSTL	313
---------------------------	-----

UTILIZACIÓN DE ACCIONES JSTL	314
------------------------------------	-----

ANÁLISIS DE LAS PRINCIPALES ACCIONES JSTL.....	314
--	-----

ACCIONES GENÉRICAS	315
--------------------------	-----

out	315
-----------	-----

set.....	315
----------	-----

remove	316
--------------	-----

catch.....	316
------------	-----

redirect.....	317
---------------	-----

CONTROL DE FLUJO.....	317
-----------------------	-----

if.....	317
---------	-----

choose	318
--------------	-----

foreach	319
---------------	-----

fortokens	319
-----------------	-----

ÍNDICE ALFABÉTICO	321
--------------------------------	------------

PRÓLOGO

El rápido crecimiento que ha experimentado en los últimos años el uso de aplicaciones Web ha ido paralelo con el aumento en la demanda de nuevas funcionalidades que los usuarios solicitan a estas aplicaciones.

Esto ha supuesto al mismo tiempo un incremento en la complejidad de los desarrollos, provocando la proliferación de lenguajes, herramientas y tecnologías orientadas a facilitar el trabajo de los programadores.

Una de esas “tecnologías” es Struts. Struts se enmarca dentro del ámbito del desarrollo de aplicaciones para Internet bajo arquitectura JavaEE, si bien, no se trata de un elemento más de dicha arquitectura sino de un marco de trabajo que ayuda a los programadores en la creación de aplicaciones en este entorno, reduciendo la complejidad y el tiempo de los desarrollos y, sobre todo, haciéndolos mucho más robustos y fáciles de mantener.

Struts fue lanzado por primera vez al mercado en el año 2002 y desde ese mismo momento tuvo una enorme aceptación en la comunidad de desarrolladores, hasta el punto que hoy en día es, con diferencia, el framework más utilizado en la construcción de aplicaciones JavaEE.

En los últimos dos años han ido apareciendo otros framework JavaEE que están teniendo bastante aceptación, como Spring o Java Server Faces (JSF). Pero este hecho, lejos de perjudicar a Struts, ha propiciado su evolución y su adaptación a los requerimientos de las aplicaciones modernas. Además, la utilización de Struts no es incompatible con estos nuevos frameworks, siendo posible su integración con los anteriormente mencionados de cara a aumentar la potencia de los desarrollos.

OBJETIVO DEL LIBRO

El objetivo que se pretende con este libro es presentar al lector todos los elementos que componen el framework Struts y guiarle en el desarrollo de las aplicaciones, para lo cual se presentan una serie de prácticas que servirán para aclarar los conceptos tratados y mostrarle la forma en la que los distintos componentes deben ser utilizados en los desarrollos.

Así pues, al finalizar el estudio de este libro el lector estará capacitado para construir aplicaciones Web bajo arquitectura JavaEE, totalmente estructuradas y robustas, utilizando toda la potencia que le ofrece Struts, incluidas las nuevas prestaciones y metodología de trabajo que se incluye en la última versión Struts 2.

A QUIÉN VA DIRIGIDO

Para poder comprender y utilizar los conceptos que se exponen en este libro es necesario tener conocimientos de programación en Java y estar familiarizado con el uso de las principales tecnologías JavaEE, como son el API Servlet y las páginas JSP.

Este libro será por tanto de gran utilidad a aquellos programadores JavaEE que se vayan a enfrentar al reto de desarrollar grandes aplicaciones empresariales en este entorno, pues es en estas circunstancias donde la utilización de un framework con las prestaciones de Struts se hace más que necesaria.

El libro puede ser utilizado también por estudiantes y, en general, por cualquier conocedor de la plataforma JavaEE que desee ampliar sus conocimientos adentrándose en el análisis de las posibilidades que ofrece Struts.

Por su enfoque didáctico y práctico, este libro puede ser utilizado como manual de estudio en cursos de programación en Java sobre plataforma JavaEE donde se incluya un módulo de Struts, siendo de gran ayuda los ejemplos y prácticas que se incluyen.

ESTRUCTURA DEL LIBRO

El libro está organizado en ocho Capítulos y dos apéndices.

Al ser el principio sobre el que está construido el framework Struts, el Capítulo 1 nos presenta la arquitectura Modelo Vista Controlador, analizando los roles desempeñados por cada capa y las ventajas que este patrón nos ofrece frente al desarrollo tradicional.

El Capítulo 2 nos introduce de lleno en el framework Struts, presentando los distintos componentes que lo integran y el funcionamiento general de la arquitectura.

Será durante el Capítulo 3 cuando se ponga en práctica la utilización de Struts con el desarrollo de una aplicación de ejemplo, aplicación que nos servirá para comprender la funcionalidad de cada uno de los componentes presentados en el Capítulo anterior y la forma en que interactúan entre sí.

En los Capítulos 4 y 5 analizaremos en detalle los dos grandes tipos de piezas que componen este framework: el API de Struts y las librerías de acciones JSP, respectivamente.

Los Capítulos 6 y 7 servirán para profundizar en dos de las características avanzadas más interesantes de Struts. En concreto, el Capítulo 6 aborda el estudio de los validadores, con cuya ayuda podremos evitar la codificación de grandes cantidades de código dedicadas únicamente a comprobar los datos suministrados por el usuario, mientras que el Capítulo 7 analiza el uso de las plantillas (tiles) para la reutilización de código en las vistas de una aplicación.

El Capítulo 8 nos presenta las características de la nueva versión del framework: Struts 2, analizando la arquitectura planteada para el desarrollo de las aplicaciones, así como la funcionalidad, utilización y configuración de los nuevos componentes del framework.

Por último, los Apéndices A y B nos presentan dos elementos que, sin formar parte de framework Struts, se complementan perfectamente con él y ayudan a mejorar los desarrollos; concretamente se trata del lenguaje de expresiones para JSP, más conocido como lenguaje EL, y de la librería de acciones estándares JSP, conocida también como librería JSTL.

MATERIAL ADICIONAL

Los ejercicios prácticos desarrolladas en los distintos Capítulos del libro pueden ser descargados desde la Web de Ra-Ma. Estas aplicaciones han sido creadas con el entorno de desarrollo Java NetBeans 6.8, de modo que si el lector cuenta con dicho entorno instalado en su máquina podrá abrir directamente los proyectos y probar su funcionamiento. No obstante, cada proyecto incluye una carpeta con los códigos fuente a fin de que puedan ser utilizados en otro entorno.

CONVENCIONES UTILIZADAS

Se han utilizado las siguientes convenciones a lo largo del libro:

- Uso de **negrita** para resaltar ciertas definiciones o puntos importantes a tener en cuenta.
- Utilización de *cursiva* para métodos y propiedades de objetos y atributos de etiquetas, así como para presentar el formato de utilización de algún elemento de código.
- Empleo de estilo `courier` para listados, tanto de código Java como de etiquetado XHTML/JSP. Para destacar los comentarios dentro del código se utilizará el tipo *courier en cursiva*, mientras que para resaltar las instrucciones importantes se empleará `courier en negrita`.

AGRADECIMIENTOS

Quiero mostrar mi agradecimiento al equipo de Ra-Ma, que ha hecho posible que este libro salga a la luz, en especial a Luis San José y Jesús Ramírez.

También quiero agradecer a Juan García Sanz, Gerente de Formación de élogos, a Juan de Dios Izquierdo, Jefe de Estudios de CICE y a Ramón Egido, director de Syncrom, su apoyo y esfuerzo en la difusión de mis libros.

DIRECCIÓN DE CONTACTO

Espero que este libro resulte de utilidad al lector, y pueda ayudarle a integrar el framework Struts en sus desarrollos y a poder sacar partido a todas las ventajas que éste ofrece.

Si desea realizar alguna consulta u observación, puede contactar conmigo a través de la siguiente dirección de correo electrónico:

ajms66@gmail.com

LA ARQUITECTURA MODELO VISTA CONTROLADOR

Las aplicaciones Web están organizadas siguiendo una arquitectura de tres capas, donde la capa cliente, implementada mediante páginas Web, tiene como misión la captura de datos de usuario y su envío a la capa intermedia, así como la presentación de resultados procedentes de ésta. Es la capa intermedia la que constituye el verdadero núcleo de la aplicación Web, encargándose del procesamiento de los datos de usuario y de la generación y envío de las respuestas a la capa cliente. Durante este proceso, la capa intermedia deberá interaccionar con la capa de datos para el almacenamiento y recuperación de información manejada por la aplicación (figura 1).

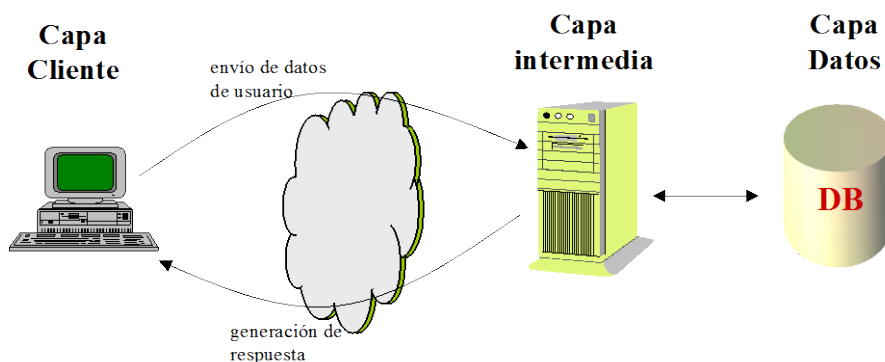


Fig. 1. Esquema de capas de una aplicación Web

Son muchas las tecnologías y lenguajes que los programadores tienen a su disposición para acometer el desarrollo de la capa intermedia de una aplicación Web (Java/JavaEE, PHP, ASP.NET, etc.). No obstante, de cara a afrontar con éxito su implementación, se hace necesario establecer un modelo o esquema que permita estructurar esta capa en una serie de bloques o componentes, de modo que cada uno de estos bloques tenga unas funciones bien definidas dentro de la aplicación y pueda desarrollarse de manera independiente al resto.

Uno de estos esquemas y, con toda seguridad, el más utilizado por los desarrolladores que utilizan JavaEE es la arquitectura Modelo Vista Controlador (MVC), la cual proporciona una clara separación entre las distintas responsabilidades de la aplicación.

1.1 EL PATRÓN MVC

Cuando hablamos de arquitectura Modelo Vista Controlador nos referimos a un patrón de diseño que especifica cómo debe ser estructurada una aplicación, las capas que van a componer la misma y la funcionalidad de cada una.

Según este patrón, la capa intermedia de una aplicación Web puede ser dividida en tres grandes bloques funcionales:

- Controlador.
- Vista.
- Modelo.

En la figura 2 se muestra esta arquitectura para el caso de una aplicación desarrollada con tecnologías JavaEE. En ella podemos ver cómo se relacionan estos tres bloques funcionales entre sí, su interacción con el resto de las capas de la aplicación y la tecnología con la que están implementados.

Seguidamente vamos a analizar detalladamente cada uno de estos bloques, presentando las características de cada uno de ellos, funcionalidades y tipo de tecnología empleada en el desarrollo de cada uno de ellos.

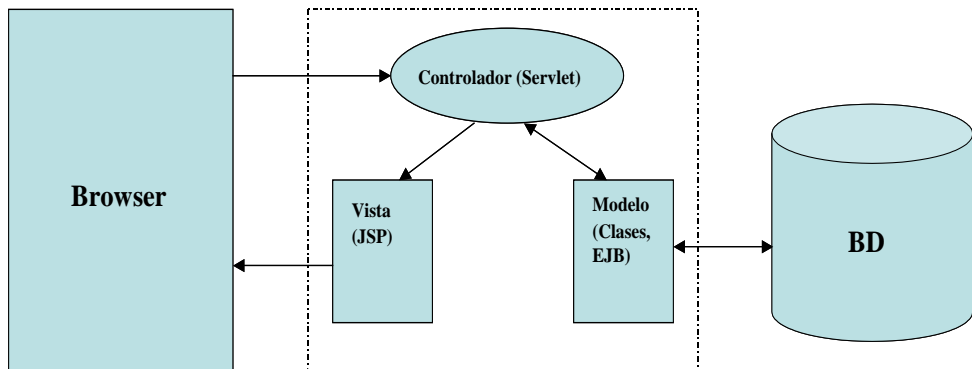


Fig. 2. Esquema de una aplicación MVC

1.1.1 El Controlador

Se puede decir que el Controlador es el “cerebro” de la aplicación. **Todas las peticiones a la capa intermedia que se realicen desde el cliente son dirigidas al Controlador**, cuya misión es determinar las acciones a realizar para cada una de estas peticiones e invocar al resto de los componentes de la aplicación (Modelo y Vista) para que realicen las acciones requeridas en cada caso, encargándose también de la coordinación de todo el proceso.

Por ejemplo, en el caso de que una petición requiera enviar como respuesta al cliente determinada información existente en una base de datos, el Controlador solicitará los datos necesarios al modelo y, una vez recibidos, se los proporcionará a la Vista para que ésta les aplique el formato de presentación correspondiente y envíe la respuesta al cliente.

La centralización del flujo de peticiones en el Controlador proporciona varias ventajas al programador, entre ellas:

- Hace que el desarrollo sea más sencillo y limpio.
- Facilita el posterior mantenimiento de la aplicación haciéndola más escalable.
- Facilita la detección de errores en el código.

En aplicaciones JavaEE el Controlador es implementado mediante un **servlet** central que, dependiendo de la cantidad de tipos de peticiones que debe gestionar, puede apoyarse en otros servlets auxiliares para procesar cada petición (figura 3).

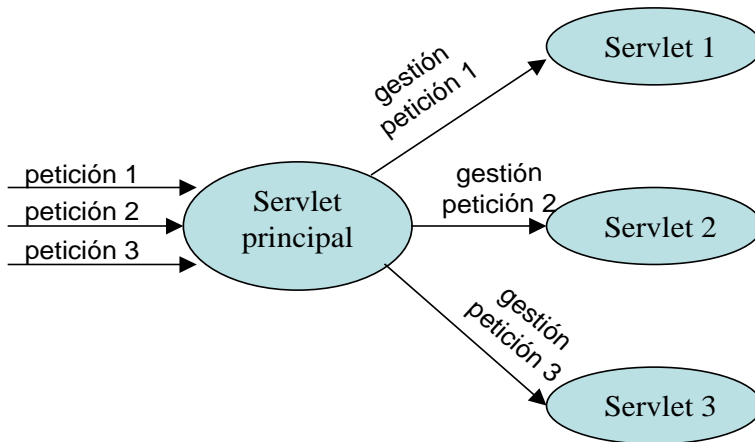


Fig. 3. Despacho de la gestión de peticiones entre distintos servlets

1.1.2 La Vista

Tal y como se puede deducir de su nombre, la Vista es la encargada de generar las respuestas (habitualmente XHTML) que deben ser enviadas al cliente. Cuando esta respuesta tiene que incluir datos proporcionados por el Controlador, el código XHTML de la página no será fijo si no que deberá ser generado de forma dinámica, por lo que su implementación correrá a cargo de una **página JSP**.

Las páginas JSP resultan mucho más adecuadas para la generación de las vistas que los servlets pues, al ser documentos de texto, resulta sencilla la inclusión de bloques estáticos XHTML y pueden ser fácilmente mantenibles por diseñadores Web con escasos conocimientos de programación.

Cuando la información que se va a enviar es estática, es decir, no depende de datos extraídos de un almacenamiento externo, podrá ser implementada por una página o documento XHTML.

1.1.3 El Modelo

En la arquitectura MVC la lógica de negocio de la aplicación, incluyendo el acceso a los datos y su manipulación, está encapsulada dentro del modelo. El Modelo lo forman una serie de componentes de negocio independientes del Controlador y la Vista, permitiendo así su reutilización y el desacoplamiento entre las capas.

En una aplicación JavaEE el modelo puede ser implementado mediante clases estándar Java o a través de **Enterprise JavaBeans**.

1.2 FUNCIONAMIENTO DE UNA APLICACIÓN MVC

Una vez analizados los distintos bloques MVC resulta sencillo comprender el funcionamiento de este tipo de aplicaciones. Para ello, analizaremos los procesos que tienen lugar en la capa intermedia desde que llega la petición procedente de la capa cliente hasta que se genera la respuesta:

- **Captura de la petición en el Controlador.** Como hemos dicho, todas las peticiones que se reciben en la aplicación son centralizadas en el Controlador, el cual a partir de la URL de la solicitud determina el tipo de la operación que quiere llevar a cabo el cliente. Normalmente, esto se hace analizando el valor de algún parámetro que se envía anexo a la URL de la petición y que se utiliza con esta finalidad:

url?operacion=validar

Otra opción es utilizar la propia URL para codificar la operación a realizar, en este caso, se utilizaría el *path info* de la dirección como indicativo del tipo de acción. En este sentido, la figura 4 nos muestra las distintas partes en las que se puede descomponer la URL completa asociada a una petición.



Fig. 4. Partes de una URL

Por ejemplo, si en un servidor de nombre de dominio *www.libros.com* tenemos desplegada una aplicación llamada “biblioteca”, cuyo Controlador es un servlet que tiene como *url pattern* el valor “/control”, la URL asociada a la operación de *validar* podría ser:

www.libros.com/biblioteca/control/validar

Mientras que otra operación, por ejemplo *registrar*, tendría como URL:

www.libros.com/biblioteca/control/registrar

Todas estas peticiones provocarán la ejecución del servlet controlador, el cual utilizará el método *getPathInfo()* del API servlet para determinar la operación a realizar.

- **Procesamiento de la petición.** Una vez que el Controlador determina la operación a realizar, procede a ejecutar las acciones pertinentes, invocando para ello a los diferentes métodos expuestos por el Modelo.

Dependiendo de las acciones a realizar (por ejemplo, un alta de un usuario en un sistema), el Modelo necesitará manejar los datos enviados por el cliente en la petición, datos que le serán proporcionados por el Controlador. De la misma manera, los resultados generados por el Modelo (por ejemplo, la información resultante de una búsqueda) serán entregados directamente al Controlador.

Para facilitar este intercambio de datos entre Controlador y Modelo y, posteriormente, entre Controlador y Vista, las aplicaciones MVC suelen hacer uso de JavaBeans. Un JavaBean no es más que una clase que encapsula un conjunto de datos con métodos de tipo *set/get* para proporcionar un acceso a los mismos desde el exterior. El siguiente listado representa un JavaBean de ejemplo que permite encapsular una serie de datos asociados a una persona:

```
public class Persona{
    private String nombre;
    private String apellido;
    private int edad;

    public void setNombre(String nombre){
        this.nombre=nombre;
    }
    public String getnombre(){
        return this.nombre;
    }
    public void setApellido(String apellido){
```



```
        this.apellido=apellido;
    }
    public String getApellido(){
        return this.apellido;
    }
    public void setEdad(int edad){
        this.edad=edad;
    }
    public int getEdad(){
        return this.edad;
    }
}
```

- **Generación de respuestas.** Los resultados devueltos por el Modelo al Controlador son depositados por éste en una variable de petición, sesión o aplicación, según el alcance que deban tener. A continuación, el Controlador invoca a la página JSP que debe encargarse de generar la vista correspondiente, esta página accederá a la variable de ámbito donde estén depositados los resultados y los utilizará para generar dinámicamente la respuesta XHTML que será enviada al cliente.

PRÁCTICA 1.1. ENVÍO Y VISUALIZACIÓN DE MENSAJES

Descripción

Para comprender la importancia de esta arquitectura, vamos a desarrollar una aplicación Web siguiendo este patrón MVC. La aplicación consistirá en un sencillo sistema de envío y visualización de mensajes a través de la Web, cuyas páginas se muestran en la figura 5. Cada mensaje estará formado por un destinatario, un remitente y un texto.

La página de inicio muestra dos enlaces con las opciones del usuario, la de visualización de mensajes le llevará a otra página (“mostrar.htm”) donde se le solicitará el nombre del destinatario cuyos mensajes quiere visualizar. En caso de tener mensajes asociados se le enviará a una página donde se le mostrará una tabla con todos sus mensajes, indicando para cada uno de ellos el remitente y el contenido del mensaje. Por otro lado, la opción de envío de mensajes le llevará a una página en la que se le solicitarán los datos del mensaje que quiere enviar, devolviéndolo después a la página de inicio una vez que el mensaje ha sido almacenado.

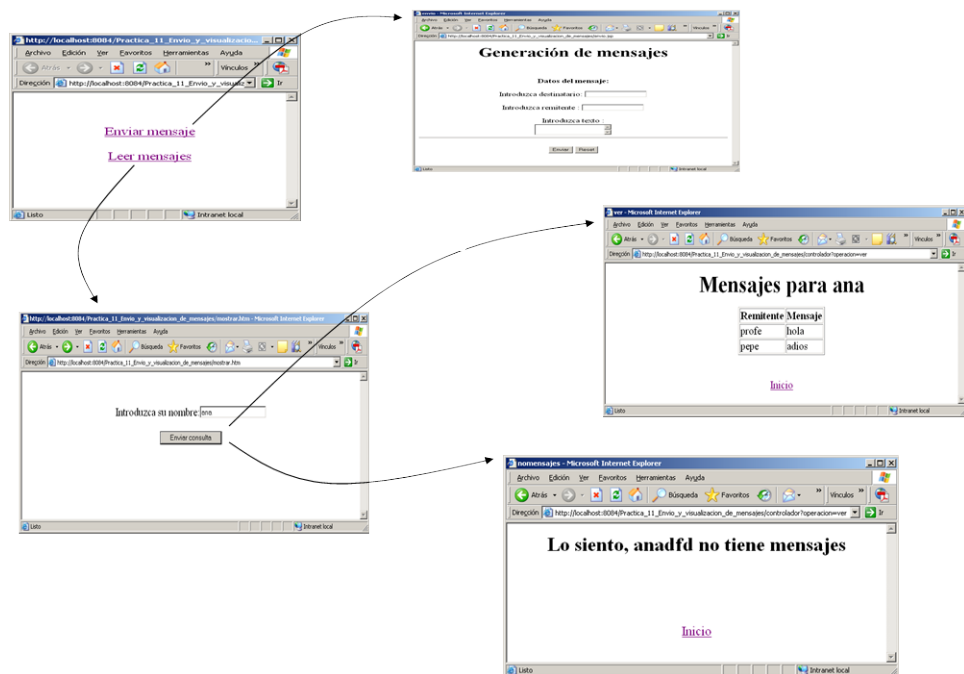


Fig. 5. Páginas de la aplicación

Desarrollo

Los mensajes manejados por la aplicación serán almacenados en una tabla cuya estructura se indica en la figura 6.

Nombre Campo	Tipo Datos
remitente	cadena de texto
destinatario	cadena de texto
texto	cadena de texto

Fig. 6. Tabla para el almacenamiento de mensajes

El desarrollo de esta aplicación se realizará siguiendo el patrón Modelo Vista Controlador, donde tendremos un servlet llamado “controlador” en el que se centralizarán todas las peticiones procedentes desde el cliente.

El Modelo estará implementado mediante una clase a la que llamaremos Operaciones que dispondrá de dos métodos: *grabarMensaje()*, encargado de almacenar en la base de datos los datos de un mensaje, y *obtenerMensajes()*, cuya función será la de recuperar la lista de mensajes asociados al destinatario que se proporciona como parámetro.

Los mensajes serán manipulados mediante una clase JavaBean llamada Mensaje, que encapsulará los tres datos asociados a un determinado mensaje.

En cuanto a las vistas, serán implementadas mediante cinco páginas, dos XHTML (inicio.htm y mostrar.htm) y tres JSP (envio.jsp, ver.jsp y nomensajes.jsp). Utilizando el parámetro “operacion” insertado en la URL, las páginas inicio.htm, mostrar.htm y envio.jsp indicarán al servlet controlador el tipo de acción que se debe llevar a cabo en cada petición.

Listados

A continuación presentamos el código de cada uno de los elementos de la aplicación.

Clase Mensaje

```
package javabeans;
public class Mensaje {
    private String remite;
    private String destino;
    private String texto;
    public Mensaje(){}
    //constructor que permite crear un objeto
    //Mensaje a partir de los datos del mismo
    public Mensaje(String remite, String destino,
                    String texto){
        this.remite=remite;
        this.destino=destino;
        this.texto=texto;
    }
    public void setRemite(String remite){
        this.remite=remite;
    }
}
```

```
public String getRemite(){
    return this.remite;
}
public void setDestino(String destino){
    this.destino=destino;
}
public String getDestino(){
    return this.destino;
}
public void setTexto(String texto){
    this.texto=texto;
}
public String getTexto(){
    return this.texto;
}
}
```

Clase Controlador

```
package servlets;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import javax beans.*;
import modelo.*;
public class Controlador extends HttpServlet {
    public void service(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String op=request.getParameter("operacion");
        //acceso a la página de envío de mensajes
        if(op.equals("envio"))
            response.sendRedirect("envio.jsp");
        //grabación de un mensaje
        if(op.equals("grabar")){
            Mensaje men=(Mensaje)request.getAttribute("mensa");
            Operaciones oper=new Operaciones();
            oper.grabaMensaje(men);
            response.sendRedirect("inicio.htm");
        }
    }
}
```

```
//acceso a la página de solicitud de mensajes
if(op.equals("muestra"))
    response.sendRedirect("mostrar.htm");
//acceso a la lista de mensajes del usuario
if(op.equals("ver")){
    Operaciones oper=new Operaciones();
    ArrayList mensajes=oper.obtenerMensajes(
        request.getParameter("nombre"));
    request.setAttribute("mensajes",mensajes);
    RequestDispatcher rd=request.
        getRequestDispatcher("/ver.jsp");
    rd.forward(request,response);
}
}
}
```

Clase Operaciones

```
package modelo;
import java.sql.*;
import javax.swing.*;
import java.util.*;
public class Operaciones {
    //método común para la obtención
    //de conexiones
    public Connection getConnection(){
        Connection cn=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.getConnection("jdbc:odbc:mensajes");
        }
        catch(Exception e){e.printStackTrace();}
        return cn;
    }
    public ArrayList obtenerMensajes(String destino){
        Connection cn=null;
        ArrayList mensajes=null;
        Statement st;
        ResultSet rs;
        try{
            cn=getConnection();
            st=cn.createStatement();
        }
```

```
String tsql;
tsql="select * from mensajes where destinatario='"+
        destino+"'";
rs=st.executeQuery(tsql);
mensajes=new ArrayList();
//para cada mensaje encontrado crea un objeto
//Mensaje y lo añade a la colección ArrayList
while(rs.next()){
    Mensaje m=new Mensaje(rs.getString("remitente"),
        rs.getString("destinatario"),
        rs.getString("texto"));
    mensajes.add(m);
}
cn.close();
}
catch(Exception e){e.printStackTrace();}
return(mensajes);
}

public void grabaMensaje(Mensaje m){
    Connection cn;
    Statement st;
    ResultSet rs;
    try{
        cn=getConnection();
        st=cn.createStatement();
        String tsql;
        //a partir de los datos del mensaje construye
        //la cadena SQL para realizar su inserción
        tsql="Insert into mensajes values('";
        tsql+=m.getDestino()+"', '"+
            m.getRemite()+"', '"+
            m.getTexto()+"'";
        st.execute(tsql);
        cn.close();
    }
    catch(Exception e){e.printStackTrace();}
}
}
```

inicio.htm

<html>

```
<body>
<center>
    <br/><br/>
    <a href="controlador?operacion=envio">
        Enviar mensaje
    </a><br/><br/>
    <a href="controlador?operacion=muestra">
        Leer mensajes
    </a>
</center>
</body>
</html>
```

mostrar.htm

```
<html>
<body>
<center>
    <br/><br/>
    <form action="controlador?operacion=ver" method="post">
        Introduzca su nombre:
        <input type="text" name="nombre"><br><br>
        <input type="submit">
    </form>
</center>
</body>
</html>
```

envio.jsp

```
<html>
<head>
<title>envio</title>
</head>
<!--captura de datos e inserción en el Javabeans-->
<jsp:useBean id="mensa" scope="request"
    class="javabeans.Mensaje" />
<jsp:setProperty name="mensa" property="*" />
<%if(request.getParameter("texto")!=null){%>
    <jsp:forward page="controlador?operacion=grabar"/>
<%}%>
```

```

<body>
<center>
  <h1>Generación de mensajes</h1>
  <form method="post">
    <br/><br/>
    <b>Datos del mensaje:</b><br/><br/>
    Introduzca destinatario:  <input type="text"
                             name="destino"><br/>
    <br/>
    Introduzca remitente :  <input type="text"
                             name="remite"><br/>
    <br/>
    Introduzca texto : <br/>
    <textarea name="texto">
  </textarea>
  <hr/><br/>
  <input type="submit" name="Submit" value="Enviar">
  <input type="reset" value="Reset">
  </form>
</center>
</body>
</html>

```

ver.jsp

```

<%@ page import="javabeans.*,java.util.*"%>
<html>
<head>
<title>ver</title>
</head>
<body>
<center>
<%String nombre=request.getParameter("nombre");%>
<h1>
Mensajes para <%=nombre%>
</h1>
<table border=1>
<tr><th>Remitente</th><th>Mensaje</th></tr>
<%boolean men=false;
ArrayList mensajes=
    (ArrayList)request.getAttribute("mensajes");
if(mensajes!=null)

```



```

//si existen mensajes para ese destinatario
//se generará una tabla con los mismos
for(int i=0;i<mensajes.size();i++){
    Mensaje m=(Mensaje)mensajes.get(i);
    if((m.getDestino()).equalsIgnoreCase(nombre)){
        men=true;%>
        <tr><td><%=m.getRemite()%></td><td>
            <%=m.getTexto()%></td>
        </tr>
        <%>
    }
}
if(!men){%>
    <!--si no hay mensajes se envía al usuario
    a la página nomensajes.jsp-->
    <jsp:forward page="nomensajes.jsp"/>
<%}%>
</table>
<br/><br/>
<a href="inicio.htm">Inicio</a>
</center>
</body>
</html>

```

nomensajes.jsp

```

<html>
<head>
<title>nomensajes</title>
</head>
<body>
    <center>
        <h2>
            Lo siento, <%=request.getParameter("nombre")%>
            no tiene mensajes
        </h2>
        <br/><br/><br/><br/>
        <a href="inicio.htm">Inicio</a>
    </center>
</body>
</html>

```


EL FRAMEWORK STRUTS

La arquitectura Modelo Vista Controlador constituye como hemos visto una excelente solución a la hora de implementar una aplicación Web en Java, aunque desde luego no está exenta de inconvenientes.

Si observamos detenidamente el ejemplo de código presentado en el Capítulo anterior, comprobaremos la existencia de dos puntos débiles que para grandes aplicaciones podrían traducirse en un desarrollo excesivamente complejo y difícilmente mantenible. Estos puntos débiles son:

- **Extensión del Controlador.** En aplicaciones en donde haya que gestionar varios tipos de peticiones, esto se traducirá en un elevado número de líneas de código en el servlet controlador, haciéndolo demasiado complejo y propenso a errores.
- **Código Java en páginas JSP.** La presencia de código Java en una página JSP, aunque proporciona una gran flexibilidad a la hora de generar las vistas, hace difícilmente mantenibles las páginas y obstaculiza la labor del diseñador de las mismas.

Como solución a estos problemas se puede optar por dividir el servlet controlador en varios servlets más pequeños, encargándose cada uno de ellos de gestionar un tipo de petición, así como crear librerías de acciones JSP personalizadas que encapsulen la mayor parte del código Java utilizado en la generación de las vistas.

Otra opción consiste en emplear algún tipo de utilidad que facilite las tareas descritas anteriormente; y es aquí donde entra en juego **Struts**.

2.1 FUNDAMENTOS DE STRUTS

Struts es un framework o marco de trabajo desarrollado por el grupo Apache, que proporciona un conjunto de utilidades cuyo objetivo es facilitar y optimizar los desarrollos de aplicaciones Web con tecnología JavaEE, siguiendo el patrón MVC.

El empleo de Struts en los desarrollos ofrece numerosos beneficios al programador, entre los que podríamos destacar:

- **Control declarativo de peticiones.** Con Struts el programador no tiene que preocuparse por controlar desde código las distintas acciones a realizar en función del tipo de petición que llega al Controlador. Este proceso es implementado por defecto por uno de los componentes que proporciona Struts, encargándose únicamente el programador de definir en un archivo de configuración XML los mapeos entre tipos de petición y acciones a ejecutar.
- **Utilización de direcciones virtuales.** A fin de evitar la inclusión directa dentro del código de la aplicación de las URL de los recursos para hacer referencia a los mismos, Struts proporciona unos tipos de objetos que permiten referirse a estos recursos mediante direcciones virtuales asociadas a los mismos. La asociación entre una dirección virtual y su correspondiente URL o dirección real se realiza en un archivo de configuración, de este modo cualquier cambio en la localización de un recurso no afectará al código de la aplicación.
- **Manipulación de datos con JavaBean.** Como hemos podido comprobar, los JavaBeans constituyen una pieza importante dentro de la arquitectura MVC al facilitar el transporte de datos entre las capas de la aplicación. Struts proporciona un sólido soporte para la manipulación de datos mediante JavaBeans, encargándose automáticamente el framework de la instanciación de los mismos, su rellenado con los datos procedentes del cliente y la recuperación y almacenamiento de los objetos en las variables de ámbito; todo esto sin necesidad de que el programador tenga que escribir una sola línea de código.

- **Juego de librerías de acciones JSP.** A fin de poder reducir al mínimo la cantidad de instrucciones de código Java dentro de una página JSP, Struts proporciona un amplio conjunto de librerías de acciones predefinidas con las que se pueden realizar la mayor parte de las operaciones que tienen lugar dentro de una página JSP, tales como el acceso a las variables de ámbito y parámetros de petición, la iteración de colecciones, manipulación de JavaBeans, etc.

La aportación de Struts al desarrollo de aplicaciones MVC se dirige básicamente a la construcción del Controlador y la Vista de una aplicación, dejando total libertad al programador en la implementación del Modelo y pueda así elegir la tecnología que crea más conveniente en cada caso.

Esto no significa que los beneficios en la utilización de Struts no se vean reflejados también en esta capa, ya que la manera en la que Struts permite implementar el Controlador ofrece un fuerte desacoplamiento entre éste y el Modelo, al tiempo que facilita una buena cohesión entre ambos.

2.2 COMPONENTES DE STRUTS

El marco de trabajo Struts está constituido por los siguientes elementos o componentes:

- Archivos de configuración.
- El API de Struts.
- Librerías de acciones JSP.

2.2.1 Archivos de configuración

Además del descriptor de despliegue `web.xml` definido por la especificación JavaEE, una aplicación Struts requiere de otros archivos de configuración adicionales. El más importante de ellos es **`struts-config.xml`**, entre otros elementos, en él se registran y configuran los distintos objetos Struts que van a ser utilizados por la aplicación, por lo que cada una deberá tener su propio archivo `struts-config.xml`.

La figura 7 ilustra la estructura tipo de este documento. Según vayamos avanzando en el estudio de los objetos Struts y la manera de utilizarlos en las aplicaciones, iremos conociendo los distintos elementos o tags que se utilizan en la construcción de un documento `struts-config.xml`.

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

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">

<struts-config>
  <form-beans>
    <!--registro de FormBeans-->

  </form-beans>

  <global-exceptions>
    <!--definición de excepciones globales-->

  </global-exceptions>

  <global-forwards>
    <!--definición de objetos forward globales-->

  </global-forwards>

  <action-mappings>
    <!--asociación entre tipos de petición y objetos Action, además
      de objetos forward locales-->

  </action-mappings>

  <!--archivos de recursos-->
  <message-resources ... />
  :
</struts-config>
```

Fig. 7. Estructura básica de un documento *struts-config.xml*

Además de *struts-config.xml* una aplicación Struts puede incorporar otros archivos de configuración, entre los más importantes destacamos:

- **ApplicationResource.properties.** Se trata de un archivo de texto plano para el almacenamiento de cadenas de texto en forma de parejas nombre=valor, siendo *valor* el texto y *nombre* una clave o identificador asociado. Cada vez que se requiera hacer uso de alguna de estas cadenas desde la aplicación, se hará utilizando la clave asociada a la misma. De esta manera, si el texto tiene que ser modificado, el cambio se realizará en el archivo de texto, no en el código.

Entre las utilidades de este archivo está la internacionalización de aplicaciones, pudiéndose definir tantos archivos de este tipo como idiomas se quiera mostrar en la aplicación, o el almacenamiento de mensajes de error personalizados. En Capítulos posteriores se estudiará con detalle la utilización de este archivo.

Este archivo deberá estar incluido dentro de alguno de los paquetes de la aplicación, además, se deberá incluir la siguiente entrada en el archivo de configuración `struts-config.xml` para que Struts pueda localizarlo:

```
<message-resources
```

```
    parameter="paquete/ApplicationResource"/>
```

- **validator-rules.xml.** Contiene las reglas de los validadores utilizados para la validación automática de los datos de usuario.
- **validation.xml.** Archivo utilizado para la asignación de validadores a los campos de los formularios. En el Capítulo dedicado a la validación de datos de usuario analizaremos con detalle el significado y uso tanto de este archivo como del anterior.
- **tiles-defs.xml.** Como tendremos oportunidad de ver, los tiles o plantillas representan una de las características más interesantes de Struts de cara a optimizar la creación de vistas. A fin de reutilizar los distintos modelos de plantillas dentro de una aplicación, éstos deberán ser definidos dentro de este archivo.

2.2.2 El API de Struts

El API de Struts lo forman el conjunto de clases de apoyo que el framework proporciona para estructurar las aplicaciones y simplificar su desarrollo. La mayor parte de estas clases se utilizan en la creación del Controlador y de JavaBeans para el tratamiento de datos, siendo los paquetes más importante de todos `org.apache.struts.action` y `org.apache.struts.actions`.

De cara a ir familiarizándonos con ellas, vamos a describir a continuación las características de las principales clases que forman este API:

- **ActionServlet.** Un objeto `ActionServlet` constituye el punto de entrada de la aplicación, recibiendo todas las peticiones HTTP que llegan de la capa cliente. Se trata básicamente de un servlet HTTP cuya clase hereda a `HttpServlet`.

El objeto `ActionServlet` lleva a cabo la siguiente funcionalidad dentro de una aplicación: cada vez que recibe una petición desde el cliente y a fin de determinar la operación a realizar, extrae la

última parte de la URL y la contrasta con la información contenida en el archivo de configuración `struts-config.xml`, a partir de la cual, el objeto lleva a cabo la instanciación del JavaBean (`ActionForm`) asociado a la acción, lo rellena con los datos procedentes del formulario cliente y deposita la instancia en el contexto correspondiente, pasando a continuación el control de la petición al objeto `Action` encargado de procesarla.

Es por ello que en la mayoría de las aplicaciones los programadores no necesitan extender la funcionalidad de `ActionServlet` y utilizan directamente esta clase, debiendo **únicamente realizar su registro en `web.xml`**.

- **Action.** Como acabamos de comentar, los objetos `Action` son los responsables de procesar los distintos tipos de peticiones que llegan a la aplicación. El **principal método con que cuenta esta clase es `execute()`**, método que será invocado por `ActionServlet` al transferir la petición al objeto. Así pues, por cada tipo de petición que se vaya a controlar, el programador deberá **definir una subclase de `Action` y sobrescribir el método `execute()`**, incluyendo en él las instrucciones requeridas para el tratamiento de la petición, tales como llamadas a los métodos de la lógica de negocio implementada en el modelo o la transferencia de resultados a las vistas para su presentación.
- **ActionForm.** Los objetos `ActionForm` son un tipo especial de JavaBean que facilitan el transporte de datos entre las capas de la aplicación. Son utilizados por `ActionServlet` para capturar los datos procedentes de un formulario XHTML y enviárselos al objeto `Action` correspondiente, todo ello sin recurrir a los incómodos `request.getParameter()`. Para ello, el programador deberá extender esta clase y proporcionar los datos miembro necesarios para el almacenamiento de los datos, así como los correspondientes métodos `set/get` que den acceso a los mismos.
- **ActionMapping.** Un objeto de tipo `ActionMapping` representa una asociación entre una petición y el objeto `Action` que la tiene que procesar. Contiene información sobre el path o tipo de URL que provoca la ejecución de la acción, así como de las posibles vistas que se pueden presentar al cliente tras su procesamiento.

Cuando el Controlador `ActionServlet` invoca al método `execute()` de un objeto `Action` para el procesamiento de una petición, proporciona como parámetro un objeto `ActionMapping` con la información asociada a la misma, pudiéndose hacer uso de sus métodos para encaminar al usuario a cualquiera de las vistas asociadas al objeto una vez que la petición ha sido procesada.

- **ActionForward.** Como ya mencionamos al hablar de las ventajas de Struts, las aplicaciones que utilizan este framework trabajan con direcciones virtuales en vez de reales. Las direcciones virtuales, más conocidas en Struts como **forwards**, son manejadas desde código a través de objetos `ActionForward`.

La clase `ActionForward` encapsula los detalles sobre la localización de un recurso, de modo que cada vez que se quiera transferir una petición desde código o redirigir al usuario a una determinada página se hará utilizando objetos `ActionForward`. Por ejemplo, tal y como tendremos oportunidad de ver más adelante, la manera de indicar desde el método `execute()` de un objeto `Action` a `ActionServlet` que debe generar una determinada vista será devolviéndole el objeto `ActionForward` asociada a la misma.

Todos estos objetos Struts utilizados por la aplicación, y otros que estudiaremos en próximos Capítulos, deberán estar apropiadamente definidos en el archivo de configuración `struts-config.xml`.

2.2.3 Librerías de acciones JSP

Las librerías de tags o acciones JSP constituyen otro de los componentes esenciales de Struts. El amplio conjunto de acciones existentes permite realizar la gran mayoría de las tareas propias de una página JSP sin necesidad de incluir en la misma ni una sola línea de código Java, facilitando así la comprensión y posterior mantenimiento de las vistas.

Struts proporciona las siguientes librerías:

- **HTML.** Incluye acciones para la construcción de formularios HTML, incorporando parte de la funcionalidad encargada de la creación de beans de tipo `ActionForm` y el rellenado de los mismos con los valores de los controles. Además de este tipo de acciones, la librería *html* proporciona otros tags que facilitan la realización

de tareas habituales en una página Web, como el enlace a otros recursos de la aplicación o la visualización de mensajes de error.

- **Bean.** Esta librería incluye acciones para el acceso a propiedades de un bean y parámetros de la petición desde una página JSP. También incluye acciones para la definición de nuevos beans y su almacenamiento en el contexto de la aplicación.
- **Logic.** Los tags incluidos en esta librería realizan las operaciones que normalmente se llevan a cabo utilizando las estructuras lógicas de un lenguaje de programación, como el recorrido de una colección o la evaluación de condiciones.
- **Nested.** Extiende parte de los tags de las librerías anteriores a fin de que puedan ser utilizados de forma anidada.
- **Tiles.** Esta librería incluye acciones para la definición de plantillas y su reutilización en diferentes páginas.

2.3 FUNCIONAMIENTO DE UNA APLICACIÓN STRUTS

En la figura 8 podemos ver el esquema general de una aplicación Web Struts con cada uno de los componentes que la integran y la interacción entre ellos.

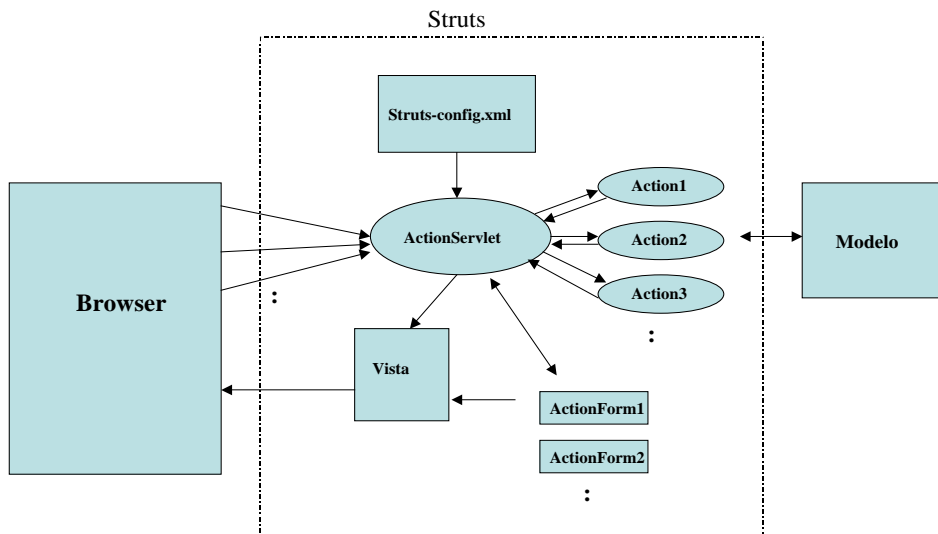


Fig. 8. Estructura de una aplicación Struts

Cada vez que desde el navegador cliente llega una petición al contenedor, asociada con una aplicación Struts, tendrá lugar el proceso que describimos a continuación:

- **Análisis de la URL de la aplicación.** El contenedor Web pasa la petición al objeto `ActionServlet`, éste, como ya hemos indicado, utiliza la última parte de la URL de la petición para determinar la acción a realizar.

Como veremos en el primer ejemplo de código que se presentará en el Capítulo siguiente, es habitual asignar como *url-pattern* del objeto `ActionServlet` el valor `*.do`, esto significa que cualquier URL procedente del cliente que termine en `.do` provocará que la petición sea capturada por este servlet.

Por ejemplo, suponiendo que el *context path* de una aplicación fuera `“/ejemplo”` y estuviera desplegada en el servidor `www.pruebas.com`, las siguientes URL provocarían la ejecución de `ActionServlet`:

`www.pruebas.com/ejemplo/listado.do`

`www.pruebas.com/ejemplo/registro.do`

`www.pruebas.com/ejemplo/busqueda.do`

En este primer paso, `ActionServlet` extrae la parte de la URL que se encuentra entre el *context path* de la aplicación y la extensión `.do`, obteniendo en cada caso los valores `“/listado”`, `“/registro”` y `“/busqueda”`.

- **Determinación de la acción a realizar.** Utilizando el dato obtenido en el paso anterior, el objeto `ActionServlet` realiza una consulta en el archivo `struts-config.xml` para determinar las operaciones a realizar. Para cada tipo de acción el archivo de configuración define la subclase `Action` que debe ser instanciada, así como el objeto `ActionForm` asociado a la operación. Tras realizar la consulta en el archivo, `ActionServlet` lleva a cabo las siguientes acciones:
 - Crea u obtiene la instancia del objeto `ActionForm` y lo rellena con los datos del formulario cliente.

- Crea una instancia del objeto Action correspondiente e invoca a su método *execute()*, pasándole como parámetro una referencia al objeto ActionForm.
- **Procesamiento de la petición.** En el método *execute()* de la subclase Action correspondiente se codificarán las acciones para el procesamiento de la petición. Se debe procurar aislar toda la lógica de negocio en el Modelo, de manera que dentro de *execute()* únicamente se incluyan las instrucciones para interaccionar con los métodos de éste, además de aquellas otras encargadas del almacenamiento de resultados en variables de contexto para su utilización en las vistas.
- **Generación de la vista.** Como resultado de su ejecución, el método *execute()* devuelve a ActionServlet un objeto ActionForward que identifica al recurso utilizado para la generación de la respuesta. A partir de este objeto, ActionServlet extrae la dirección virtual encapsulada en el mismo y utiliza este valor para obtener del archivo struts-config.xml la dirección real de la página XHTML o JSP correspondiente.

DESARROLLO DE UNA APLICACIÓN CON STRUTS

A lo largo del Capítulo anterior hemos estado analizando las características y componentes del framework Struts, así como las ventajas que su utilización ofrece para los programadores de aplicaciones Web.

Es el momento de verlo en acción, de aprender a utilizar y encajar cada una de sus piezas. Para ello, vamos a comenzar con el desarrollo de una sencilla aplicación que construiremos con los distintos componentes que nos proporciona Struts y que fueron estudiados en el Capítulo anterior. Tras un primer análisis sobre el funcionamiento y estructura de la aplicación, iremos detallando los diferentes pasos a seguir para su creación, aprovechando cada uno de ellos para presentar las características a nivel de código de los componentes que se van a utilizar y conocer los detalles de su configuración.

3.1 DESCARGA E INSTALACIÓN DEL FRAMEWORK STRUTS

En el sitio Web <http://struts.apache.org> encontramos el paquete de instalación de Struts con todas las librerías y utilidades para poder trabajar con este framework.

Dicho paquete se distribuye como un archivo .zip, pudiendo elegir entre distintas versiones existentes en producción de este framework. Dejando al margen las versiones Struts 2.x, que serán tratadas en el último Capítulo del libro,

descargaremos la última versión del tipo 1.x, que en el momento de escribir estas líneas es la 1.3.9 (figura 9).

pack completo
de instalación

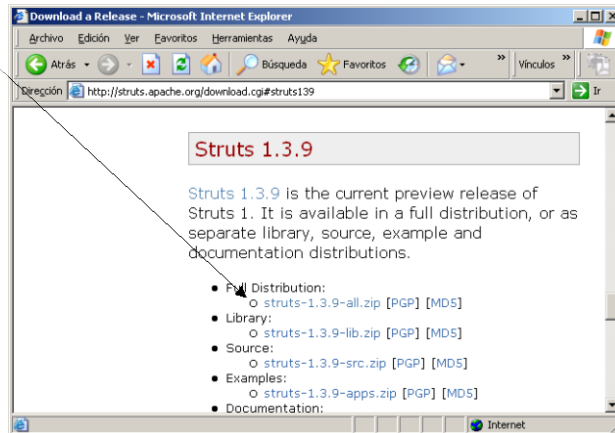


Fig. 9. *Página de descarga del framework Struts*

Una vez descargado y descomprimido el archivo, comprobamos que incluye cuatro carpetas principales:

- **apps.** Aplicaciones Struts de ejemplo.
- **docs.** Documentación y ayuda del API.
- **src.** Código fuente de Struts.
- **lib.** Librerías con los componentes Struts.

Es esta última carpeta (*lib*) la que contiene los elementos principales para poder crear aplicaciones Struts. Las principales librerías incluidas en esta carpeta son:

- **struts-core-1.3.9.jar.** Es la más importante de todas, pues contiene las principales clases que forman el API de struts.
- **struts-extras-1.3.9.jar.** Incluye algunas clases extras para la implementación del Controlador.

- **struts-taglib-1.3.9.jar.** Contiene todas las clases que implementan las distintas acciones de las librerías de tags de struts.
- **struts-tiles-1.3.9.jar.** Proporciona todos los componentes necesarios para trabajar con plantillas.
- **commons-validator-1.3.1.jar.** Incluye todo el soporte necesario (clases y scripts de cliente) para la utilización de validadores en las aplicaciones.

Si utilizamos algún IDE, como NetBeans o Eclipse, no será necesario realizar la descarga de estos archivos, pues los entornos de desarrollo ya los llevan incorporados. No tendremos más que indicar en el proyecto Web que queremos usar el plug-in de Struts para que dichas librerías sean incorporadas al proyecto.

3.2 APLICACIÓN PRÁCTICA PARA VALIDACIÓN Y REGISTRO DE USUARIOS

El programa de ejemplo que vamos a desarrollar en este apartado va a consistir en una práctica y útil aplicación Web encargada de la validación y registro de usuarios en un sistema.

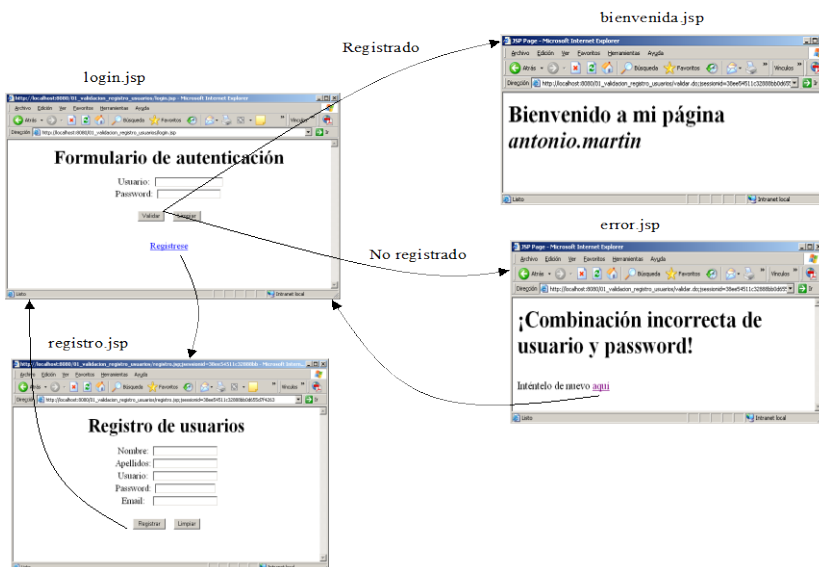


Fig. 10. Diagrama con las páginas de la aplicación de validación y registro

Posteriormente, ampliaremos este ejemplo con el desarrollo de un programa de consulta vía Web de las llamadas realizadas por un determinado usuario a través de su operador de telefonía. En esta primera versión tan sólo vamos a centrarnos, tal y como hemos indicado, en la funcionalidad para la validación de los usuarios contra la base de datos del operador, así como del registro de éstos en la misma.

La figura 10 muestra un esquema de la páginas que intervienen en la aplicación y la navegación a través de ellas.

3.2.1 Funcionamiento de la aplicación

La página inicial de la aplicación es login.jsp, y es la encargada de solicitar los credenciales al usuario. En caso de que la combinación de usuario y password exista en la base de datos, se mostrará simplemente una página con un mensaje de bienvenida personalizado para el cliente. Si la combinación no es válida, se mostrará una página de error con un enlace a login.jsp para que vuelva a intentar validarse. Así mismo, la página inicial contendrá un enlace a la página de registro donde los usuarios no validados podrán darse de alta en el sistema.

Aunque la base de datos completa contiene un total de cuatro tablas, en la figura 11 mostramos únicamente la tabla “clientes” de la base de datos “telefonía”, que es la que vamos a utilizar en este ejemplo.

Campo	Tipo
nombre	cadena de texto
apellidos	cadena de texto
usuario	cadena de texto
password	cadena de texto
email	cadena de texto

Fig. 11. Campos de la tabla “clientes”

3.2.2 Esquema de la aplicación

Haciendo un primer análisis del funcionamiento descrito, determinaremos que, al margen del acceso inicial a la página login.jsp, existen cuatro tipos de peticiones que se pueden lanzar desde el navegador cliente a la aplicación:

- Petición para validación del usuario. Se produce desde la página login.jsp al pulsar el botón “validar”, teniendo como objetivo la validación del usuario a partir de sus credenciales.
- Petición para registrar. Es la petición que se produce desde la página registro.jsp al pulsar el botón “registrar” y cuyo objetivo será llevar a cabo el alta del cliente a partir de los datos introducidos.
- Petición para acceder a la página de login. Esta petición es la que se realiza desde el *link* “aquí” que aparece en la página error.jsp y tiene como objetivo llevar de nuevo al usuario a la página de login.
- Petición para acceder a la página de registro. Esta petición es la que se realiza desde el *link* “regístrate” situado en la página login.jsp y que tiene como objetivo llevar al usuario a la página de registro.

Las dos primeras peticiones requieren un tratamiento especial por parte del Controlador, mientras que las otras dos son simples enlaces a sendas vistas, sin necesidad de ninguna acción previa por parte de la aplicación.

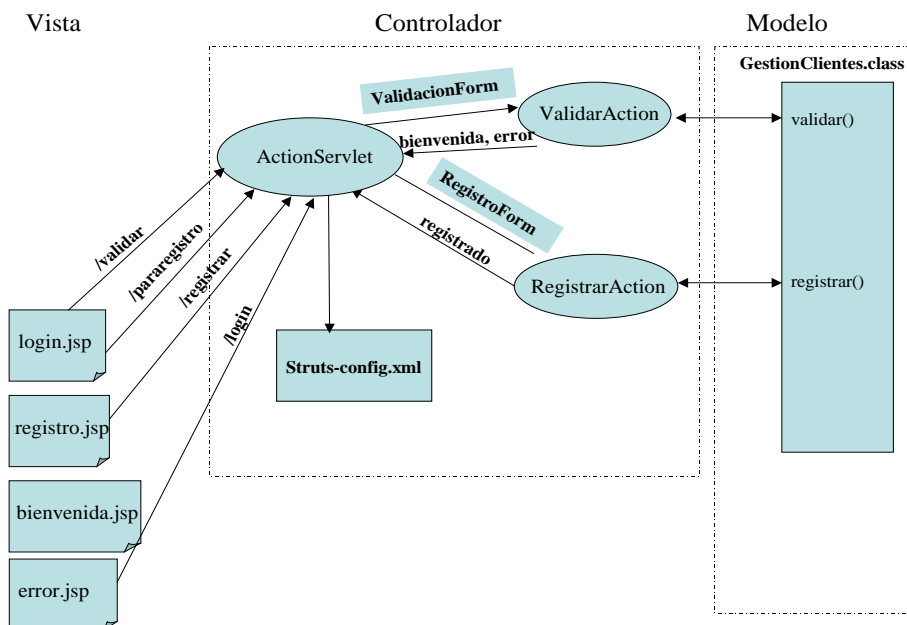


Fig. 12. Esquema Struts de la aplicación

En la figura 12 se presenta un esquema general de cómo quedará la estructura de la aplicación, indicando los distintos componentes utilizados en la construcción de la misma.

En ella podemos ver los cuatro tipos de peticiones comentadas anteriormente que llegan a `ActionServlet`, indicando en la parte superior de la flecha el path asociado a cada una de ellas.

En el interior del Controlador podemos apreciar las dos clases de acción encargadas de gestionar las peticiones activas:

- **ValidarAction.** Se encarga de gestionar la petición “/validar”. Su misión será comprobar si el usuario está o no registrado en la base de datos del sistema, para lo que se apoyará en el método *validar()* proporcionado por la clase `GestionClientes` del Modelo. El método *execute()* de `ValidarAction` recibirá como parámetro un `JavaBean` de la clase `ValidacionForm` con los credenciales suministrados por el usuario y, como resultado de su ejecución, devolverá a `ActionServlet` el objeto forward “bienvenida” o “error”, según el usuario esté o no registrado.
- **RegistrarAction.** Se encarga de gestionar la petición “/registrar” que envía los datos de usuario encapsulados en el `JavaBean` `RegistroForm` para su ingreso en la base de datos, apoyándose para ello en el método *registrar()* del Modelo. Su método *execute()* devuelve el objeto forward “registrado” como indicación de que el registro se ha realizado correctamente. A efectos de simplificar la aplicación de ejemplo no se ha tenido en cuenta la situación en la que no haya sido posible registrar al usuario.

3.2.3 Construcción de la aplicación

A continuación, vamos a analizar en detalle el desarrollo de los distintos componentes de la aplicación, comenzando por la creación de la estructura de directorios de la misma.

3.2.3.1 ESTRUCTURA DE UNA APLICACIÓN WEB STRUTS

Una aplicación Struts sigue la misma estructura de directorios de cualquier aplicación Web JavaEE, aunque antes de comenzar el desarrollo habrá que incluir en ella los distintos componentes del framework como son el archivo de configuración `struts-config.xml`, inicialmente vacío y que será situado dentro del directorio `WEB-INF`, y los archivos de librería (`.jar`) descargados con el paquete de

instalación, los cuales se incluirán dentro del directorio `\lib`, aunque es posible que no todos ellos sean utilizados en todas las aplicaciones.

Como ya hemos indicado, esta operación se realizará de manera automática si utiliza algún IDE, como NetBeans o Eclipse, a través del asistente para la creación de proyectos Web.

La estructura de directorios de la aplicación deberá quedar entonces tal y como se indica en la figura 13. Si hemos utilizado algún IDE es posible que los nombres de los archivos `.jar` sean algo diferentes a los indicados, si bien contendrán los mismos elementos.

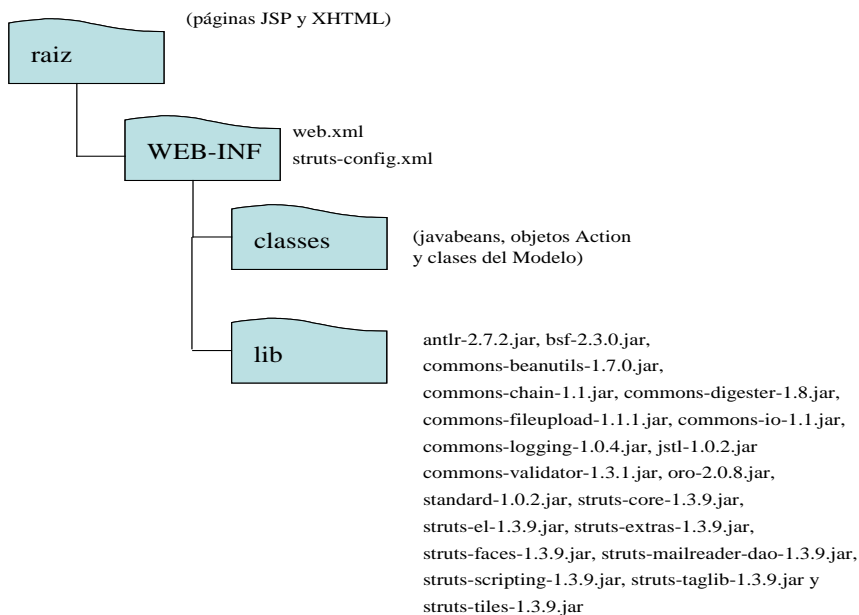


Fig. 13. Organización física de directorios de una aplicación Web Struts

Como se aprecia en la figura anterior, en el directorio WEB-INF tan sólo hemos incluido, además de `web.xml`, el archivo de configuración `struts-config.xml`. Cuando vayamos a construir aplicaciones que utilicen algunos de los componentes especiales de Struts, como validadores o plantillas, veremos que será necesario incluir archivos de configuración adicionales, tal y como se indicó en el Capítulo anterior.

3.2.3.2 REGISTRO DEL SERVLET ACTIONSERVLET

Como ya se ha indicado, la mayoría de las veces utilizaremos directamente la propia clase `ActionServlet` para la creación del servlet principal de la aplicación. Esta clase está incluida en el archivo `struts-core-1.3.9.jar` que hemos situado en el directorio `\lib` de la aplicación y habrá que registrarla en el archivo de configuración `web.xml`, tal y como se indica en la figura 14.

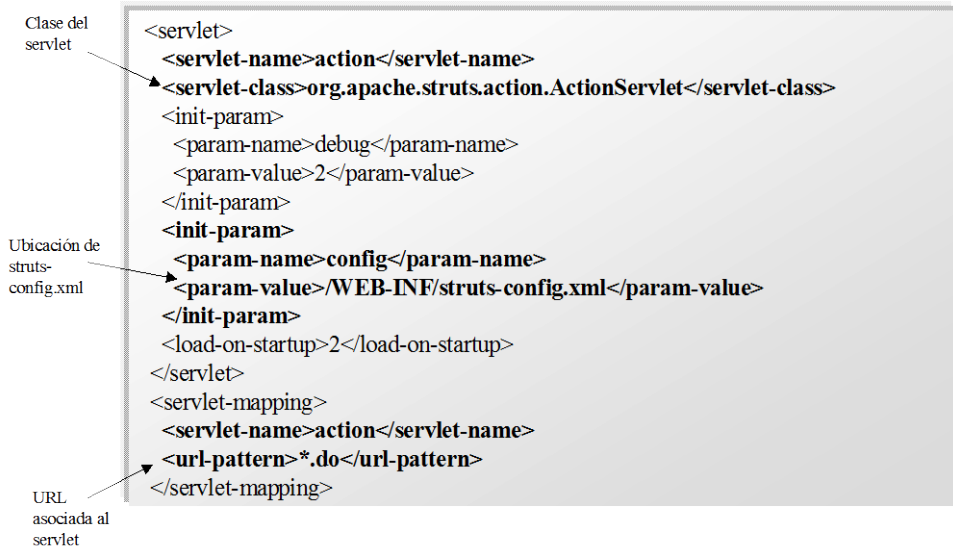


Fig. 14. Registro del servlet controlador `ActionServlet`

Los principales elementos relacionados con el registro de `ActionServlet` son:

- **servlet-class.** Contiene el nombre cualificado de la clase.
- **init-param.** `ActionServlet` utiliza dos parámetros de inicialización: por un lado el parámetro *config* que contiene la dirección relativa a la aplicación Web del archivo de configuración de Struts, y por otro *debug*, cuyo valor se establecerá a 2.
- **url-pattern.** Contiene el path de la URL de la petición que da acceso al servlet, en este caso cualquier valor que termine con la extensión `.do`. Como ya se indicó anteriormente, este formato suele ser el utilizado habitualmente en Struts, pudiéndose utilizar cualquier otro valor distinto a `do`. En vez del formato de extensión,

otro valor típico que también suele utilizarse como *url-pattern* en algunos casos es */do/**, que significa que cualquier petición con URL cuyo valor de *servlet path* sea “/do” hará que se ejecute el *ActionServlet*, utilizando el valor del *path info* para determinar la acción a realizar.

3.2.3.3 CAPTURA DE DATOS DE USUARIO: LAS CLASES VALIDACIONFORM Y REGISTROFORM

Estas clases serán utilizadas por Struts para encapsular los datos procedentes de los formularios de validación y registro, respectivamente. Ambas heredan su funcionalidad de la clase *ActionForm* del API de Struts, encargándose el programador únicamente de proporcionar los datos miembros para el almacenamiento de los valores y de la implementación de los métodos *setXxx/getXxx* correspondientes.

El siguiente listado corresponde al código de estas dos clases, ambas las hemos situado en el paquete *javabeans*:

```
package javabeans;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.*;
public class ValidacionForm extends ActionForm {
    //datos miembro
    private String usuario;
    private String password;
    //métodos de acceso
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String nombre) {
        this.usuario = nombre;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
package javabeans;
import javax.servlet.http.HttpServletRequest;
```

```
import org.apache.struts.action.*;
public class RegistroForm extends ActionForm {
    //datos miembro
    private String nombre;
    private String apellidos;
    private String usuario;
    private String password;
    private String email;
    //métodos de acceso
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
}
```

Aunque no se va a hacer uso de ellos en este primer ejemplo, la clase `ActionForm` proporciona además dos métodos `reset()` y `validate()`, que pueden ser sobrescritos para tareas de limpieza y validación de datos. Analizaremos en profundidad estos métodos en próximos Capítulos.

A fin de que Struts pueda crear y gestionar objetos `ActionForm`, las clases correspondientes deben quedar registradas en el archivo `struts-config.xml`, para lo cual debemos añadir el elemento `<form-beans>` al principio del documento, inmediatamente después del elemento raíz, e incluir en él un subelemento `<form-bean>` por cada una de las subclases `ActionForm` que vayamos a utilizar. En nuestro ejemplo quedaría de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">

<struts-config>
  <form-beans>
    <form-bean name="RegistroForm"
      type="javabeans.RegistroForm"/>
    <form-bean name="ValidacionForm"
      type="javabeans.ValidacionForm"/>
  </form-beans>
  :
```

Cada elemento `<form-bean>` incluye los siguientes atributos:

- **name.** Identificador que asignará Struts a la instancia creada.
- **type.** Nombre cualificado de la clase.

Indicando esta información en el archivo de configuración, el programador no necesitará incluir en la página JSP ninguna acción de tipo *useBean* para crear u obtener la instancia, ni tampoco acciones *setProperty/getProperty* para acceder a sus propiedades; **Struts se encargará de realizar todas estas tareas de forma transparente para el programador.**

3.2.3.4 IMPLEMENTACIÓN DEL MODELO

Al no depender directamente de Struts, su implementación puede llevarse a cabo en cualquier momento. Si la hemos incluido en este punto es porque, como veremos ahora en el listado, los métodos de negocio implementados en la clase reciben como parámetros objetos de las clases `ActionForm` presentadas anteriormente. No obstante, si se quiere tener un total desacoplamiento entre el Modelo y el Controlador, podemos optar por utilizar como parámetros datos de tipos básicos de Java que representen a cada una de las propiedades del bean que serán utilizadas por el método del Modelo, en vez de pasar el objeto `ActionForm` completo. Otra opción sería utilizar como parámetro un segundo `JavaBean` “plano” independiente del API Struts.

Tal y como quedó reflejado en el esquema de la figura 12, la implementación del Modelo se lleva a cabo mediante una clase llamada `GestionClientes`; esta clase tiene dos métodos `validar()` y `registrar()` que se encargan respectivamente de comprobar la existencia del usuario en la base de datos y de registrar nuevos usuarios en la misma. Ambos métodos actúan sobre la tabla “clientes” cuya estructura se muestra en la figura 15.

Campo	Tipo
nombre	Texto
apellidos	Texto
usuario	Texto
password	Texto
email	Texto

Fig. 15. *Tabla de clientes*

Además de `GestionClientes`, se incluye otra clase en el Modelo llamada `Datos` que proporciona métodos para la obtención y cierre de conexiones con la base de datos. Los métodos de esta clase sirven de soporte para el resto de métodos del Modelo.

En el siguiente listado se muestra el código de ambas clases:

```
package modelo;
import java.sql.*;

public class Datos {
    private String driver;
    private String cadenacon;
    public Datos() {
    }
    public Datos(String driver,String cadenacon){
        //almacena los datos para la conexión
        //con la base de datos
        this.driver=driver;
        this.cadenacon=cadenacon;
    }
    public Connection getConexion(){
        Connection cn=null;
        try{
            Class.forName(driver).newInstance();
            cn=DriverManager.getConnection(cadenacon);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        return cn;
    }
    public void cierraConexion(Connection cn){
        try{
            if(cn!=null && !cn.isClosed()){
                cn.close();
            }
        }
        catch(SQLException e){
            e.printStackTrace();
        }
    }
}

package modelo;
import java.sql.*;
import javax.swing.*;
```

```
public class GestionClientes {
    Datos dt;
    public GestionClientes(String driver, String cadenacon) {
        dt=new Datos(driver,cadenacon);
    }
    public boolean validar(ValidacionForm vf){
        boolean estado=false;
        try{
            Connection cn=dt.getConexion();
            //instrucción SQL para obtener los datos
            //del usuario indicado
            String query = "select * from clientes ";
            query+="where password ='"+vf.getPassword();
            query+="' and usuario='"+vf.getUsuario()+"'";
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            estado= rs.next();
            dt.cierraConexion(cn);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            return estado;
        }
    }
    public void registrar(RegistroForm rf){
        //genera la instrucción SQL de inserción a partir
        //de los datos almacenados en el JavaBean Usuario
        String query = "INSERT INTO clientes ";
        query+=values("'" +rf.getNombre()+"', '"+
            rf.getApellidos()+"', '"+rf.getUsuario()+
            "', '"+rf.getPassword()+
            "', '"+rf.getEmail()+"'");
        try{
            Connection cn=dt.getConexion();
            Statement st =cn.createStatement();
            st.execute(query);
            st.close();
            dt.cierraConexion(cn);
        }
    }
}
```

```
        catch(Exception e){e.printStackTrace();}  
    }  
}
```

3.2.3.5 PROCESAMIENTO DE PETICIONES: LAS CLASES VALIDARACTION Y REGISTRARACTION

Tras el registro del servlet controlador `ActionServlet`, procedemos a la definición de las acciones, éstas se implementarán mediante clases cuyos objetos se van a encargar de procesar las peticiones que llegan al Controlador. Estas clases deberán heredar a `Action` y sobrescribir el método `execute()` incluido en dicha clase, incluyendo en él las instrucciones para el procesamiento de la petición.

El formato del método `execute()` se muestra en la figura 16.

```
public ActionForward execute(ActionMapping mapping,  
                             ActionForm form,  
                             HttpServletRequest request,  
                             HttpServletResponse response) {  
  
    /*  
        instrucciones para el tratamiento de la petición  
    */  
  
}
```

Fig. 16. Método `execute()` de la clase `Action`

Como podemos comprobar, el método debe devolver al Controlador `ActionServlet` un objeto `ActionForward` con los datos de la vista que deberá ser enviada al usuario.

Así mismo, `execute()` recibe cuatro parámetros cuyo significado se describe a continuación:

- **mapping.** Se trata de un objeto de la clase `ActionMapping` que encapsula las opciones de configuración definidas para la acción en el archivo `struts-config.xml`, entre ellas las diferentes vistas a las que podrá ser encaminado el usuario una vez completada la acción. Normalmente este objeto se emplea para obtener el objeto `ActionForward` que será devuelto al Controlador, para lo que podrán utilizarse cualquiera de sus siguientes métodos:

- `findForward(String forward)`. Devuelve el objeto `ActionForward` asociado a la vista cuya dirección virtual se especifica en el parámetro.
- `getInputForward()`. Devuelve un objeto `ActionForward` asociado a la vista cuya dirección se especifica en el atributo `input` del elemento `<action>` asociado a la acción.
- **form**. Contiene el objeto de la subclase `ActionForm` creado por el Controlador y que almacena los datos del formulario cliente. En caso de que no se haya utilizado ningún `ActionForm` en la petición, este parámetro tendrá el valor *null*.
- **request y response**. Contienen respectivamente los objetos `HttpServletRequest` y `HttpServletResponse` proporcionados por el contenedor Web a `ActionServlet`. Utilizando estos objetos el método `execute()` podrá tener acceso a las mismas opciones que se encuentran disponibles para cualquier servlet HTTP.

A continuación se muestra el código de la clase `ValidarAction`, utilizada en este programa de ejemplo para gestionar la acción de validación de un usuario:

```
package servlets;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import modelo.*;
import javabeans.*;
public class ValidarAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        //obtenemos los datos de conexión con la base
        //de datos (driver y cadena de conexión) de
        //los parámetros de contexto definidos
        // en el archivo web.xml
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
        String cadenaCon=this.getServlet().
            getServletContext().getInitParameter("cadenaCon");
        GestionClientes gc=
```

```
        new GestionClientes(driver,cadenaCon);
ValidacionForm vf=(ValidacionForm)form;
//invoca al método validar() del Modelo para saber si
//el usuario está o no registrado
if(gc.validar(vf)){
    return mapping.findForward("bienvenida");
}
else{
    return mapping.findForward("error");
}
    }
}
```

Como podemos ver en este listado, el método *execute()* no incluye ninguna instrucción que contenga lógica de negocio de la aplicación. Para saber si el usuario está o no registrado en la aplicación hace uso del método *validar()* implementado en la clase *GestionClientes*. Si el usuario está registrado (el método *validar()* devolverá el valor *true*), se devolverá al Controlador el objeto *ActionForward* asociado a la vista cuya dirección virtual es “bienvenida”, en caso contrario el usuario será encaminado a la vista “error”.

En cuanto a la clase *RegistrarAction*, he aquí el código de la misma:

```
package servlets;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import modelo.*;
import javax beans.*;
public class RegistrarAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws Exception {
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
        String cadenaCon=this.getServlet().
            getServletContext().getInitParameter("cadenaCon");
        GestionClientes gc=
            new GestionClientes(driver,cadenaCon);
        RegistroForm rf=(RegistroForm)form;
        gc.registrar(rf);
    }
}
```

```

        return mapping.findForward("registrado");
    }
}

```

Tanto en esta clase como en la anterior podemos observar cómo los datos de conexión con la base de datos están definidos fuera del código de la aplicación, concretamente como parámetros de contexto accesibles para todos los componentes de la aplicación dentro del archivo web.xml:

```

<web-app version="2.5"
  xmlns=http://java.sun.com/xml/ns/javaee
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>driver</param-name>
    <param-value>sun.jdbc.odbc.JdbcOdbcDriver
    </param-value>
  </context-param>
  <context-param>
    <param-name>cadenaCon</param-name>
    <param-value>jdbc:odbc:telefonía</param-value>
  </context-param>
  <servlet>
    :

```

Una vez implementadas las clases Action, es necesario registrarlas en el archivo struts-config.xml dentro del elemento <action-mappings>, utilizando un subelemento <action> para la definición de los parámetros asociados a cada acción. A través de estos parámetros se le suministra a Struts toda la información que necesita conocer sobre cada acción, como el *path* asociado a la misma o el objeto ActionForm que deberá ser utilizado en la llamada a *execute()*. A continuación se muestran las entradas a incluir en el archivo struts-config.xml para las acciones utilizadas en esta aplicación de ejemplo:

```

<struts-config>
:
<action-mappings>
  <action name="ValidacionForm" path="/validar"
    scope="request" type="servlets.ValidarAction">
    <forward name="bienvenida" path="/bienvenida.jsp"/>
    <forward name="error" path="/error.jsp"/>

```

```
</action>
<action name="RegistroForm" path="/registrar"
        scope="request" type="servlets.RegistrarAction">
    <forward name="registrado" path="/login.jsp"/>
</action>
</action-mappings>
:
```

Seguidamente, describiremos el significado de los atributos indicados en el elemento `<action>`:

- **path.** Representa la dirección que debe ser utilizada en la URL de la petición para que `ActionServlet` derive la petición a este objeto `Action`, es decir, el valor que debe aparecer entre el *context path* de la aplicación y la extensión *.do* (*servlet path*). En el caso de la página de entrada `login.jsp`, el atributo *action* del formulario deberá tomar el valor `"/validar.do"`, aunque si se utiliza el tag de Struts `<html:form>`, basta con indicar `"/validar"` en dicho atributo.
- **name.** Nombre del objeto `ActionForm` en el que `ActionServlet` volcará los datos del formulario cliente desde el que se lanza la petición, y que será pasado como parámetro en la llamada al método *execute()* de `Action`. El valor de este atributo debe corresponder con alguno de los `ActionForm` definidos en el apartado `<form-beans>`.
- **type.** Nombre cualificado de la subclase `Action`.
- **validate.** Indica si se deben validar o no los datos del `ActionForm`. Si su valor es *true*, se invocará al método *validate()* del bean antes de hacer la llamada a *execute()*. El valor predeterminado es *false*.
- **scope.** Ámbito de utilización del objeto `ActionForm`. Su valor puede ser *request*, *session* o *application*, siendo *request* el valor predeterminado.
- **input.** Aunque no se utiliza en ninguno de los `Action` de este ejemplo, indica la URL de la vista que debe ser enviada al cliente en caso de que durante la validación del bean (ejecución del método *validate()*) se genere algún error. Más adelante veremos un ejemplo de utilización de este atributo.

Como vemos además en el listado anterior, cada elemento `<action>` puede incluir uno o varios elementos `<forward>`. Estos elementos representan las posibles vistas a las que se puede encaminar el usuario tras la ejecución de la acción, indicándose en cada uno de ellos la asociación entre la dirección virtual de la vista y la dirección física. Cada elemento `<forward>` dispone de dos atributos:

- **name.** Nombre lógico o virtual asociado a la vista. Este valor es el utilizado por el método *findForward()* de *ActionMapping* para crear el objeto *ActionForward* asociado a la vista a la que se tiene que dirigir al usuario.
- **path.** Contiene la URL relativa de la vista.

3.2.3.6 OBJETOS FORWARD GLOBALES

Además de las peticiones “/validar” y “/registrar” que provocan la ejecución de los objetos *Action* anteriores, *ActionServlet* puede recibir también dos peticiones por parte de cliente (“/login” y “/pararegistro”) que simplemente deberían tener como consecuencia el reenvío de la petición a una de las vistas de la aplicación, sin que sea necesario realizar ningún tratamiento especial por parte del Controlador.

Para que esto sea posible se deberán definir unos elementos `<forward>` globales que relacionen las peticiones con las páginas a las que será reenviado el usuario. En estos elementos `<forward>` el atributo *name* contendrá el *path* de la petición asociada y *path* la URL relativa de la página destino.

La definición de los elementos `<forward>` globales se incluirá dentro del elemento `<global-forward>`, que a su vez estará situado entre `<form-beans>` y `<action-mappings>`:

```
:
</form-beans>
<global-forwards>
    <forward name="login" path="/login.jsp"/>
    <forward name="pararegistro" path="/registro.jsp"/>
</global-forwards>
<action-mappings>
:
```


3.2.3.7 LAS PÁGINAS DE LA VISTA

La vista de esta aplicación de ejemplo está constituida por cuatro páginas JSP: login.jsp, registro.jsp, bienvenida.jsp y error.jsp. Todas ellas harán uso de alguna de las librerías de acciones JSP proporcionadas por Struts. Estas librerías y las clases que implementan las acciones se encuentran incluidas en el archivo de librería struts-core-1.3.9.jar, pero para poder utilizar en una página JSP las acciones definidas en una determinada librería habrá que hacer uso de la directiva *taglib* e indicar en su atributo *uri* el identificador asociado a dicha librería. Los identificadores de las librerías de acciones de Struts están definidos en los propios archivos de librería *.tld*, siendo sus valores los indicados en la tabla de la figura 17.

librería	identificador
html	http://struts.apache.org/tags-html
bean	http://struts.apache.org/tags-bean
logic	http://struts.apache.org/tags-logic
nested	http://struts.apache.org/tags-nested

Fig. 17. *Identificadores de las librerías de Struts*

Seguidamente vamos a analizar cada una de las cuatro páginas JSP utilizadas en esta aplicación.

login.jsp

Es la página de validación de usuarios. Los datos son recogidos por un formulario XHTML y enviados a ActionServlet que los encapsulará en el objeto ValidarForm. El siguiente listado corresponde con el código de esta página:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-html"
    prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
  <body>
    <center>
      <h1>Formulario de autenticación</h1>
```


- **<html:form>**. Equivale a la etiqueta <form> y como en el caso de ésta sus principales atributos son *method* y *action*. El atributo *method* contiene el método de envío utilizado por la petición, siendo su valor por defecto “POST”. En cuanto a *action*, contiene el path asociado al objeto Action que se encargará de tratar la petición, según la información indicada en struts-config.xml. Como se puede apreciar en el código de las páginas anteriores, la cadena indicada en *action* no incluye la extensión .do; una vez que la acción sea interpretada en el servidor y transformada en su equivalente etiqueta <form>, se establecerá en el atributo *action* de esta etiqueta la dirección relativa completa de la petición, añadiendo a la izquierda del valor original el *context path* de la aplicación y a su derecha la extensión .do asociada a ActionServlet.

En el caso de login.jsp, y suponiendo que el directorio virtual de la aplicación Web es “/ejemplo”, la acción:

```
<html:form action="/validar" method="post">
```

será transformada en:

```
<form action="/ejemplo/validar.do" method="post">
```

- **<html:text>**. En vez de utilizar una única etiqueta para todos, Struts dispone de un tag diferente para la generación de cada uno de los controles gráficos de la interfaz, haciendo más intuitivo su uso. En el caso de <html:text>, genera una caja de texto de una única línea. Entre los atributos más importantes de esta acción se encuentran:
 - **property**. Este atributo lo tienen todos los controles gráficos de la librería y representa el nombre de la propiedad del objeto ActionForm en la que se volcará el valor suministrado en el campo de texto. En caso de que la propiedad del bean disponga de un valor inicial, este será utilizado para inicializar el contenido del control.
 - **maxlength**. Número máximo de caracteres que admite el control.

- **readonly**. Indica si el contenido del control es de sólo lectura (*true*) o de lectura y escritura (*false*).
 - **onxxx**. Representa cada uno de los manejadores de eventos JavaScript que pueden ser capturados por el control, siendo *xxx* el nombre del evento.
- **<html:password>**. Genera una caja de texto de tipo password, donde los caracteres introducidos se ocultan al usuario. Dispone de los mismos atributos que **<html:text>**.
 - **<html:textarea>**. Genera un control de texto multilínea.
 - **<html:submit>**. Equivale al control XHTML **<input type="submit">**. Al igual que éste, mediante su atributo *value* se establece el texto mostrado por el botón.
 - **<html:reset>**. Limpia todos los campos del formulario, dejando los valores por defecto establecidos. Equivale al control XHTML **<input type="reset">**.
 - **<html:select>**. Genera una lista de selección de opciones. Además de *property*, este elemento dispone del atributo *size*, mediante el cual se indica si se generará una lista desplegable (*size*≤1) o una lista abierta (*size*>1).
 - **<html:option>**. Se emplea para la generación de las opciones en el interior de una lista de selección. En su atributo *value* se especifica el valor que será asignado a la propiedad del bean especificada en el atributo *property* de **<html:select>**, cuando la opción sea seleccionada.
 - **<html:options>**. Cuando las opciones de la lista se deben generar dinámicamente a partir del contenido de una colección, resulta más cómodo utilizar este elemento en vez del anterior. En su atributo *name* se indica el nombre del bean, existente en alguno de los ámbitos de la aplicación, que contiene la colección de datos a mostrar. Si la colección está contenida en alguna de las propiedades del bean, se debería indicar en el atributo *property* el nombre de dicha propiedad.
 - **<html:optionsCollection>**. Es similar al anterior, aunque este tag se utiliza en aquellos casos en que la colección que contiene los

elementos de la lista es de objetos de tipo `JavaBean`. Además de *name* y *property*, habrá que indicar en los atributos *label* y *value* los nombres de las propiedades del bean que representan el texto y valor de la opción, respectivamente. El siguiente bloque de código de ejemplo genera una lista con los datos de los objetos de tipo `Libro` contenidos en una colección llamada “books”, utilizando las propiedades *titulo* e *isbn* del bean como texto y valor de cada opción:

```
<html:select>
    <html:optionsCollection name="books"
        label="titulo" value="isbn" />
</html:select>
```

- **<html:checkbox>**. Genera una casilla de verificación. Uno de sus principales atributos, *value*, indica el valor que se asignará a la propiedad del bean especificada en *property*, siempre y cuando la casilla se encuentre activada al hacer el submit del formulario.
- **<html:radio>**. Genera un control de tipo `radiobutton`. Al igual que ocurre con `checkbox`, el atributo *value* contiene el valor que será asignado a la propiedad del bean especificada en *property*. Todos aquellos controles `radiobutton` con el mismo valor de atributo *property* serán excluyentes entre sí, siendo el contenido del atributo *value* de aquél que esté seleccionado el que se almacenará en la propiedad del bean al realizar el submit del formulario.

Todos los tags anteriores corresponden a controles gráficos de la interfaz de usuario, por lo que deberán aparecer siempre anidados dentro del elemento `<html:form>`.

Además de éstos, hay otros tags importantes de esta librería que no están relacionados con el uso de formularios, como es el caso de **<html:link>** que también se utiliza en la página `login.jsp` mostrada anteriormente. Este tag tiene como finalidad la inserción de un enlace XHTML de tipo `<a>`, contando entre sus principales atributos con:

- **forward**. Nombre lógico del elemento `<forward>` global definido en `struts-config.xml`, que hace referencia a la vista donde será enviado el usuario cuando se pulse el enlace. En la página `login.jsp` existe un elemento `<html:link>` con el siguiente aspecto:

```
<html:link forward="pararegistro">Regístrese</html:link>
```

que al ser solicitada la página será transformado en:

```
<a href="/ejemplo/registro.jsp">Regístrese</a>
```

- **action.** Su significado es el mismo que el del atributo *action* de `<html:form>`, es decir, indica el path asociado al objeto Action que se encargará de tratar la petición. En caso de que deba ejecutarse algún tipo de acción al activar este enlace, se utilizará este atributo en vez de `forward`.
- **href.** URL a la que será enviado el usuario al pulsar el enlace.
- **linkName.** Nombre de la marca local existente en la propia página donde será enviado el usuario al pulsar el enlace.

Según se desprende de su significado, un elemento `<html:link>` **debe contener uno y sólo uno de los atributos anteriores**.

Otros elementos de esta librería como `<html:errors>` o `<html:messages>` serán estudiados en el próximo Capítulo.

registro.jsp

Se trata de la página de registro de nuevos usuarios. Los datos son recogidos por un formulario XHTML y enviados a ActionServlet que los encapsulará en el objeto RegistroForm. El código de esta página se muestra en el siguiente listado:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-html"
    prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
<body>
<center>
<h1>Registro de usuarios</h1>
<html:form action="/registrar" method="POST">
<table>
```


bienvenida.jsp

Es la página que se muestra al usuario una vez que se ha validado, he aquí el código de la misma:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-bean"
    prefix="bean" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML
    4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Bienvenido a mi página <i>
      <bean:write name="ValidacionForm"
        property="usuario"/></i></h1>
  </body>
</html>
```

En esta página se hace uso también de una de las acciones de la librería *tags-bean*, concretamente de **<bean:write>**. En el Capítulo 5 se analizará el funcionamiento de ésta y del resto de las acciones que componen la librería, de momento comentar que **<bean:write>** permite mostrar en la página los valores almacenados en las propiedades de un bean.

PRÁCTICA 3.1. LISTADO DE LLAMADAS DE USUARIOS

Con los conocimientos que ya tenemos sobre Struts estamos en condiciones de implementar nuestra primera práctica con Struts. Para ello partiremos del ejercicio de ejemplo que hemos ido desarrollando a lo largo del Capítulo.

Descripción

Se trata de desarrollar una aplicación que permita a los usuarios visualizar el listado de las llamadas realizadas con cualquiera de los teléfonos contratados con un operador de telefonía. Inicialmente el usuario accederá a una página de login donde deberá validarse y, en caso de que se trate de un usuario registrado, se le mostrará una página donde deberá elegir en una lista desplegable el teléfono cuyo listado de llamadas quiere visualizar, tras lo cual se le mostrará una nueva página con los datos de las llamadas realizadas. La figura 18 muestra el aspecto de las páginas de la aplicación

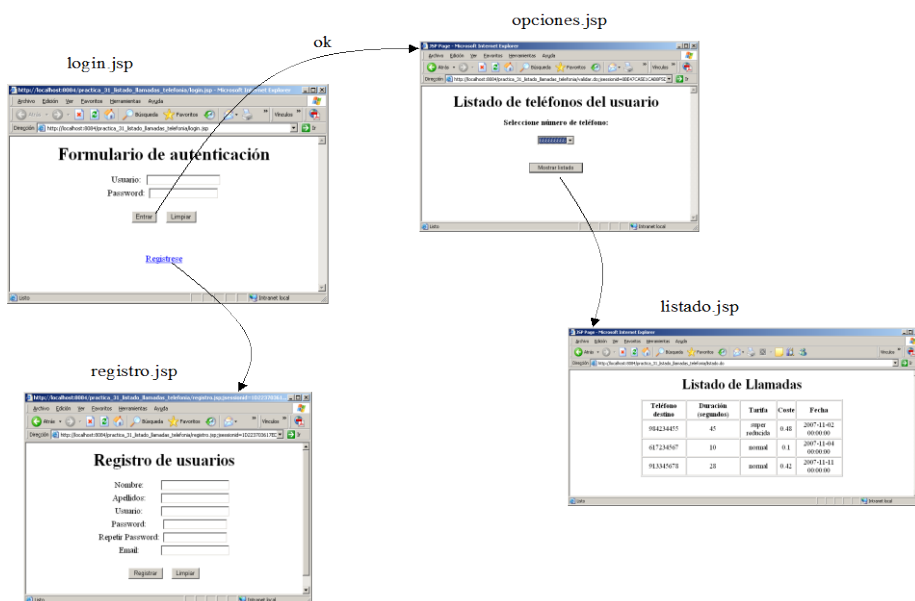


Fig. 18. Páginas de la aplicación

Desarrollo

Como ya se ha indicado, esta aplicación será desarrollada mediante Struts, partiendo del desarrollo realizado en el ejemplo presentado anteriormente. En esta

implementación, además de `ValidarAction` y `RegistrarAction` debemos crear una nueva acción a la que llamaremos `ListadoAction` que se encargará de gestionar la petición que solicita el listado de llamadas a partir del teléfono seleccionado. Así mismo, la acción `ValidarAction` deberá ser modificada incluyendo las instrucciones necesarias para que se recupere el listado de teléfonos asociados al usuario antes de devolver el objeto `ActionForward` asociado a la vista, la cual ya no será `bienvenida.jsp` sino `opciones.jsp`.

Por otro lado, será necesario crear un nuevo `ActionForm`, al que llamaremos `OpcionesForm`, que contendrá el número de teléfono seleccionado por el usuario. Crearemos además un JavaBean “plano” llamado `LlamadaBean` que encapsule los datos de las llamadas telefónicas para facilitar su tratamiento desde la aplicación, el cual se apoyará a su vez en otro JavaBean llamado `TarifaBean` que representa los datos de las tarifas. También modificaremos `ValidacionForm` a fin de que pueda almacenar la lista de teléfonos asociada al usuario validado.

En cuanto al modelo, se crearán dos nuevas clases llamadas `GestionTelefonos` y `GestionLlamadas` que proporcionarán toda la lógica necesaria para obtener la información relativa a los números de teléfono del usuario y las llamadas realizadas.

En la figura 19 tenemos un esquema completo de la base de datos “telefonía” que utilizaremos en esta aplicación, indicándose las tablas que la componen y los campos definidos en cada una de ellas.

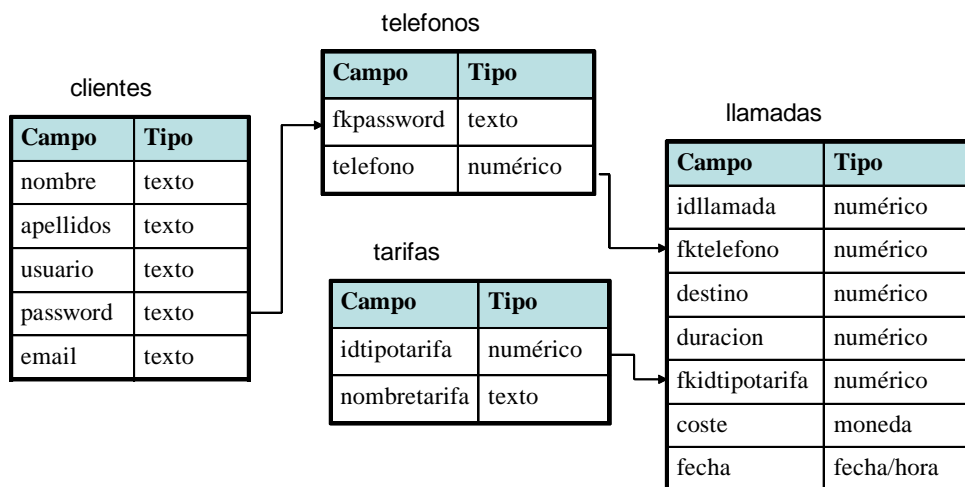


Fig. 19. Base de datos de la aplicación

Aunque no parezca muy ortodoxo, a fin de simplificar el modelo de datos hemos utilizado el campo *password* del usuario como campo clave de la tabla *clientes*. Este campo se utilizaría, por tanto, como clave ajena en la tabla *telefonos* para identificar los teléfonos de cada cliente.

Listado

En primer lugar, mostraremos cómo quedará el archivo de configuración `struts-config.xml` de la aplicación, en el que podemos ver los componentes que la forman:

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

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">

<struts-config>
    <form-beans>
        <form-bean name="OpcionesForm"
            type="javabeans.OpcionesForm"/>
        <form-bean name="RegistroForm"
            type="javabeans.RegistroForm"/>
        <form-bean name="ValidacionForm"
            type="javabeans.ValidacionForm"/>
    </form-beans>
    <global-forwards>
        <forward name="login" path="/login.jsp"/>
        <forward name="toregistro" path="/registro.jsp"/>
    </global-forwards>
    <action-mappings>
        <action name="ValidacionForm" path="/validar"
            scope="request" type="servlets.ValidarAction">
            <forward name="bienvenida" path="/opciones.jsp"/>
            <forward name="error" path="/login.jsp"/>
        </action>
        <action input="/registro.jsp" name="RegistroForm"
            path="/registrar" scope="request"
            type="servlets.RegistrarAction">
            <forward name="registrado" path="/login.jsp"/>
        </action>
```

```
<action path="/listado" name="OpcionesForm"
        type="servlets.ListadoAction" parameter="operacion">
    <forward name="listado" path="/listado.jsp"/>
</action>
</action-mappings>
</struts-config>
```

Seguidamente mostramos el listado de los nuevos componentes de la aplicación y de aquéllos que han sido modificados respecto al ejemplo analizado, indicando en este último caso los cambios realizados en fondo sombreado.

ValidacionForm.java

```
package javabeans;

import org.apache.struts.action.*;
import java.util.*;

public class ValidacionForm extends ActionForm {
    //almacena los credenciales del usuario
    private String usuario;
    private String password;
    //almacena el mensaje asociado al resultado
    //de la validación
    private String mensaje;
    //almacena el conjunto de teléfonos del usuario validado
    private ArrayList<Integer> telefonos;

    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String nombre) {
        this.usuario = nombre;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public void setMensaje(String mensaje){
        this.mensaje=mensaje;
    }
```

```
    }  
    public String getMensaje(){  
        return mensaje;  
    }  
    public ArrayList<Integer> getTelefonos() {  
        return telefonos;  
    }  
    public void setTelefonos(ArrayList<Integer> telefonos) {  
        this.telefonos = telefonos;  
    }  
}
```

OpcionesForm.java

```
package javabeans;  
import org.apache.struts.action.*;  
public class OpcionesForm extends ActionForm {  
    //almacena el número de teléfono  
    private int numero;  
    public int getNumero() {  
        return numero;  
    }  
    public void setNumero(int i) {  
        numero = i;  
    }  
}
```

ValidarAction.java

```
package servlets;  
  
import javax.servlet.http.*;  
import org.apache.struts.action.*;  
import modelo.*;  
import javabeans.*;  
public class ValidarAction extends Action {  
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,  
                                HttpServletRequest request,  
                                HttpServletResponse response)  
        throws Exception {  
        //obtiene los datos del driver y de la cadena de
```

```

//conexión de los parámetros de contexto
String driver=this.getServlet().
    getServletContext().getInitParameter("driver");
String cadenaCon=this.getServlet().
    getServletContext().getInitParameter("cadenaCon");
GestionClientes gc=
    new GestionClientes(driver,cadenaCon);
ValidacionForm vf=(ValidacionForm)form;
if(gc.validar(vf)){
    GestionTelefonos gt=
        new GestionTelefonos(driver,cadenaCon);
    //recupera los números asociados al password y
    //los almacena en el bean ValidacionForm
    //para que puedan ser accesibles
    //a la vista "bienvenida"
    vf.setTelefonos(
        gt.getTelefonos(vf.getPassword()));
    return mapping.findForward("bienvenida");
}
else{
    vf.setMensaje("<h2>Combinación de usuario
        y password incorrecta!</h2>");
    return mapping.findForward("error");
}
}
}

```

ListadoAction.java

```

package servlets;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import org.apache.struts.actions.*;
import javaxBeans.*;
import modelo.*;
import java.util.*;

public class ListadoAction extends Action {
    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)

```

```

        throws Exception {
        OpcionesForm of=(OpcionesForm)form;
        GestionLlamadas gl=getGestionLlamadas(request);
        //recupera el listado de llamadas asociadas al
        //número y reenvia la petición a la vista
        request.setAttribute("llamadas",
            gl.getTodasLlamadasTelefono(of.getNumero()));
        return mapping.findForward("listado");
    }
    //método de apoyo a execute
    private GestionLlamadas getGestionLlamadas
        (HttpServletRequest request){
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
        String cadenaCon=this.getServlet().
            getServletContext().getInitParameter("cadenaCon");
        return new GestionLlamadas(driver,cadenaCon);
    }
}

```

LlamadaBean.java

```

package javabeans;

import java.io.*;

public class LlamadaBean implements Serializable{
    private int telefono;
    private int destino;
    private int duracion;
    private String tarifa;
    private double coste;
    private String fecha;
    public LlamadaBean() {
    }
    public int getTelefono() {
        return telefono;
    }
    public void setTelefono(int telefono) {
        this.telefono = telefono;
    }
    public int getDestino() {
        return destino;
    }
}

```

```
    }
    public void setDestino(int destino) {
        this.destino = destino;
    }
    public int getDuracion() {
        return duracion;
    }
    public void setDuracion(int duracion) {
        this.duracion = duracion;
    }
    public String getTarifa() {
        return tarifa;
    }
    public void setTarifa(String tarifa) {
        this.tarifa = tarifa;
    }
    public double getCoste() {
        return coste;
    }
    public void setCoste(double coste) {
        this.coste = coste;
    }
    public String getFecha() {
        return fecha;
    }
    public void setFecha(String fecha) {
        this.fecha = fecha;
    }
}
```

TarifaBean.java

```
package javabeans;

public class TarifaBean {
    private int idtipotarifa;
    private String nombretarifa;
    public int getIdtipotarifa() {
        return idtipotarifa;
    }
    public void setIdtipotarifa(int idtipotarifa) {
        this.idtipotarifa = idtipotarifa;
    }
}
```



```
    }  
    public String getNombretarifa() {  
        return nombretarifa;  
    }  
    public void setNombretarifa(String nombretarifa) {  
        this.nombretarifa = nombretarifa;  
    }  
}
```

GestionLlamadas.java

```
package modelo;  
  
import java.sql.*;  
import java.util.*;  
import javax.swing.*;  
import java.text.*;  
public class GestionLlamadas {  
    Datos dt;  
    public GestionLlamadas(String driver, String cadenacon) {  
        dt=new Datos(driver,cadenacon);  
    }  
    //obtiene todas las llamadas asociadas a  
    //un determinado teléfono  
    public ArrayList<LlamadaBean> getTodasLlamadasTelefono(  
                                                int telefono){  
        String query = "select * from llamadas where ";  
        query+="fktelefono="+telefono;  
        return getLlamadas(query,telefono);  
    }  
    private ArrayList<LlamadaBean> getLlamadas(  
        String sql, int telefono){  
        ArrayList<LlamadaBean> llamadas=  
            new ArrayList<LlamadaBean>();  
        try{  
            Connection cn=dt.getConexion();  
            Statement st =cn.createStatement();  
            ResultSet rs = st.executeQuery(sql);  
            while(rs.next()){  
                //para cada llamada asociada al teléfono crea  
                //un javabean LlamadaBean y lo rellena con los  
                //datos de la misma
```

```
LlamadaBean llamada=new LlamadaBean();
llamada.setTelefono(telefono);
llamada.setDestino(rs.getInt("destino"));
llamada.setDuracion(rs.getInt("duracion"));
llamada.setTarifa(this.getTarifa(
    rs.getInt("fkidtipotarifa")));
llamada.setCoste(rs.getDouble("coste"));
llamada.setFecha(rs.getString("fecha"));
llamadas.add(llamada);
}
dt.cierraConexion(cn);
}
catch(Exception e){
    e.printStackTrace();
}
finally{
    return llamadas;
}
}

//recupera el nombre de una tarifa en la tabla de
//tarifas a partir de su identificador
private String getTarifa(int id){
    String tarifa=null;
    try{
        Connection cn=dt.getConnection();
        String query = "select nombretarifa from tarifas ";
        query += "where idtipotarifa="+id;
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(query);
        //si existe una tarifa asociada a ese identificador
        //devolverá su nombre, si no devolverá null
        if(rs.next()){
            tarifa=rs.getString("nombretarifa");
        }
        dt.cierraConexion(cn);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        return tarifa;
    }
}
```

```

    }
}
}

```

GestionTelefonos.java

```

package modelo;

import java.sql.*;
import java.util.*;

public class GestionTelefonos {
    Datos dt;
    public GestionTelefonos(String driver, String cadenacon)
    {
        dt=new Datos(driver,cadenacon);
    }
    //devuelve una colección con todos los teléfonos
    //asociados al usuario
    public ArrayList<Integer> getTelefonos(String password){
        ArrayList<Integer> numeros=
            new ArrayList<Integer>();
        try{
            Connection cn=dt.getConexion();
            String query = "select telefono from
                                telefonos where fkpassword ='";
            query+=password+"'";
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            while(rs.next()){
                numeros.add(rs.getInt("telefono"));
            }
            dt.cierraConexion(cn);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            return numeros;
        }
    }
}

```

opciones.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@ taglib uri="http://struts.apache.org/tags-bean"
prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html:html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <center>
      <h1>Listado de teléfonos del usuario</h1>
      <html:form action="/listado" method="POST">
        <b> Seleccione número de teléfono:</b>
        <br/> <br/>
        <html:select property="numero">
          <!--Recupera el bean ValidacionForm almacenado
            en una variable de petición-->
          <html:options name="ValidacionForm"
            property="telefonos" />
        </html:select>
        <br/> <br/> <br/>
        <html:submit value="Mostrar listado"/>
      </html:form>
    </center>
  </body>
</html:html>
```

listado.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <center>
      <%ArrayList<LlamadaBean> llamadas=
        (ArrayList)request.getAttribute("llamadas");%>
      <h1>Listado de Llamadas</h1>
      <table border="1" width="60%">
        <th>Teléfono destino</th>
        <th>Duración (segundos)</th>
        <th>Tarifa</th>
        <th>Coste</th>
        <th>Fecha</th>
        <%for (int i=0;i<llamadas.size();i++){
          LlamadaBean llamada=llamadas.get(i);%>
          <tr>
            <td><%=llamada.getDestino()%></td>
            <td><%=llamada.getDuracion()%></td>
            <td><%=llamada.getTarifa()%></td>
            <td><%=llamada.getCoste()%></td>
            <td><%=llamada.getFecha()%></td>
          </tr>
        <%}%>
      </table>
    </center>
  </body>
</html>
```


ANÁLISIS DEL API DE STRUTS

Después de ver en acción durante el Capítulo anterior a los diferentes componentes de Struts, a lo largo de los siguientes Capítulos vamos a analizar en detalle estos componentes a fin de poder explotar a fondo todas sus posibilidades.

En este Capítulo profundizaremos en el estudio del API de Struts, tanto en lo que se refiere a los componentes del Controlador como a los JavaBeans de tipo `ActionForm`.

4.1 PROCESAMIENTO DE UNA PETICIÓN: CLASES `ACTIONSERVLET` Y `REQUESTPROCESSOR`

Como ya quedó indicado durante el análisis del Capítulo anterior, `ActionServlet` representa el punto de entrada a la aplicación Struts, de tal manera que todas las peticiones HTTP que llegan desde la capa cliente a la aplicación son dirigidas a este servlet.

Una vez recibida la petición, `ActionServlet` delega el análisis y procesamiento de la misma en un objeto **`RequestProcessor`**. Este objeto es el encargado de realizar todas las operaciones relativas al tratamiento de la petición descritas en el Capítulo 2, como son el análisis de la URL de la petición, instanciación y relleno del `ActionForm`, determinación del objeto `Action` a ejecutar e invocación a su método *`execute()`*, análisis del objeto `ActionForward` retornado y transferencia de la petición a la vista correspondiente.

Cada una de estas tareas es realizada de forma independiente por los distintos métodos proporcionados por `org.apache.struts.action.RequestProcessor`. Estos métodos tienen una implementación por defecto que, normalmente, suele ser adecuada en la mayoría de los escenarios. No obstante, el hecho de que cada tarea sea tratada por un método diferente permite al programador personalizar de manera individualizada estas operaciones, creando subclases de `RequestProcessor` en los que sobrescribiremos únicamente aquellos métodos cuyo comportamiento predeterminado nos interese cambiar.

He aquí algunos de los principales métodos de la clase `RequestProcessor`. Los analizaremos según el orden en que son ejecutados:

- **`processPath()`**. Analiza la URL de la petición y obtiene el path que será utilizado para determinar la acción a ejecutar.
- **`processPreprocess()`**. Indica si se continuará o no procesando la petición después de que este método se haya ejecutado. En su implementación por defecto *`processPreprocess()`* únicamente tiene como función devolver el valor *`true`* para que la petición siga procesándose, es por ello que si el programador desea introducir algún tipo de control en función del cual el procesamiento de la petición pueda ser cancelado, la sobrescritura de este método constituye una buena opción.
- **`processMapping()`**. A partir del path obtenido durante la ejecución de *`processPath()`*, la implementación por defecto de este método se encarga de localizar el elemento `<action>` correspondiente dentro del archivo de configuración `struts-config.xml`, devolviendo un objeto `ActionMapping` asociado al mismo.
- **`processActionForm()`**. Utilizando el objeto `ActionMapping` proporcionado por el método anterior devuelve el objeto `ActionForm` asociado a la acción, localizándolo en el ámbito especificado o creándolo si no existiera. En caso de no indicarse ningún `ActionForm` en la acción se devolverá el valor *`null`*.
- **`processPopulate()`**. Con los datos del formulario cliente, *`processPopulate()`* procede al rellenado del objeto `ActionForm` obtenido por el método anterior.
- **`processValidate()`**. Invoca al método *`validate()`* del objeto `ActionForm`, una vez que éste ha sido rellenado.

- **processActionCreate()**. Tras la validación del formulario, en este método se procede a la creación del objeto Action definido en el ActionMapping.
- **processActionPerform()**. Una vez creado el objeto Action desde este método se realiza la llamada al método *execute()* del mismo, devolviendo como repuesta el objeto ActionForward generado por éste.
- **processForwardConfig()**. Por último, a partir del objeto ActionForward anterior y la información incluida en el archivo struts-config.xml, este método se encarga de redirigir al usuario a la vista correspondiente.

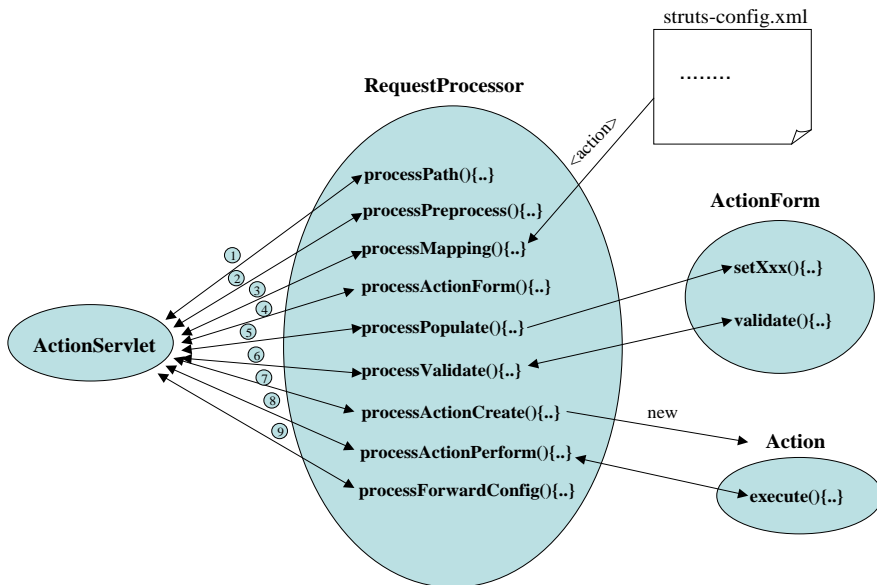


Fig. 20. Ciclo de vida de un objeto *RequestProcessor*

Para que *ActionServlet* utilice la subclase de *RequestProcessor* definida por el programador en vez de la propia *RequestProcessor*, es necesario registrar esta clase en *struts-config.xml* dentro del elemento `<controller>`, tal y como se indica a continuación:

```
<struts-config>
:
<action-mappings>
:
```

```

</action-mappings>
<controller processorClass="misclases.MiRequestProcessor"/>

```

Mediante el atributo *processorClass* se especifica el nombre cualificado de la clase, que como el resto de las clases de la aplicación deberá estar ubicada en el directorio *raiz_aplicacion\WEB-INF\classes*.

Supongamos por ejemplo que en la aplicación presentada en la práctica 3.1, un usuario intenta acceder a la página de listado de llamadas incluyendo directamente la URL asociada a esa opción en su navegador. Al hacerlo se producirá una excepción en el servidor, enviándole al usuario una respuesta con contenido indeterminado.

Para evitar esta situación, definiremos una subclase *RequestProcessor* personalizada en el que sobrescribiremos el método *processPreprocess()* de manera que evite el procesamiento de la petición en caso de que se intente acceder a la página sin haberse validado.

Lo primero que haremos es modificar la clase *ValidarAction* para que almacene en una variable de sesión un dato boolean que nos indique que el usuario está validado:

```

if(gc.validar(vf)){
    GestionTelefonos gt=
        new GestionTelefonos(driver,cadenaCon);
    //recupera los números asociados al password y
    //los almacena en el bean ValidacionForm
    //para que puedan ser accesibles
    //a la vista "bienvenida"
    vf.setTelefonos(
        gt.getTelefonos(vf.getPassword()));
    //guarda en una variable de sesión
    //el indicativo de que el usuario se ha validado
    request.getSession().
        setAttribute("validado",true);
    return mapping.findForward("bienvenida");
}

```

Después crearemos una subclase de *RequestProcessor* a la que llamaremos *PreProcesamiento*, cuya implementación será la siguiente:

```

package servlets;
import javax.servlet.http.*;

```

```
import org.apache.struts.action.*;
import org.apache.struts.actions.*;

public class PreProcesamiento extends RequestProcessor{
    public boolean processPreprocess(HttpServletRequest request,
        HttpServletResponse response){
        HttpSession sesion=request.getSession();
        //de esta manera, si se intenta solicitar el
        //listado de llamadas sin haberse validado
        //previamente, se recibirá una página en blanco
        if(request.getServletPath().
            equals("/listado.do")&&
            sesion.getAttribute("validado")==null){
            return false;
        }
        else{
            return true;
        }
    }
}
```

Finalmente, habrá que registrar esta clase en struts-config.xml para que sea utilizada por ActionServlet:

```
<controller processorClass="servlets.PreProcesamiento"/>
```

4.2 CLASES DE ACCIÓN

Como norma general, solemos crear subclases de Action para gestionar los distintos tipos de peticiones que llegan desde el cliente a la aplicación. Además de ésta, en el paquete **org.apache.struts.actions** se incluyen otras clases cuya utilización puede ser más adecuada en determinadas aplicaciones a la hora de gestionar ciertos tipos de peticiones, de modo que en vez de crear una subclase de Action para su tratamiento se crearía un subtipo de una de estas otras clases.

Vamos a analizar a continuación las clases más interesantes que forman este paquete.


```

        //método que lleva a cabo la recuperación
        //del contenido completo del carrito

    }
}

```

Como vemos, la subclase de `DispatchAction` no necesita sobrescribir el método `execute()`, de hecho no debe sobrescribirse ya que es la implementación por defecto de este método incluida en `DispatchAction` la encargada de determinar el método a ejecutar según la petición realizada.

En este caso, las tres peticiones deben provocar la ejecución del mismo tipo de objeto (`GestionCarritoAction`) por lo que las tres deberán incluir el mismo path asociado, por ejemplo “`gestioncarrito.do`”. Será a través **de un parámetro enviado en la petición** como el método `execute()` podrá determinar cuál de los métodos definidos en la clase para la gestión de las acciones ha de ser ejecutado. El nombre de este parámetro será elegido por el programador y su valor en cada petición tendrá que **coincidir exactamente con el nombre del método** que se quiera ejecutar (figura 21).

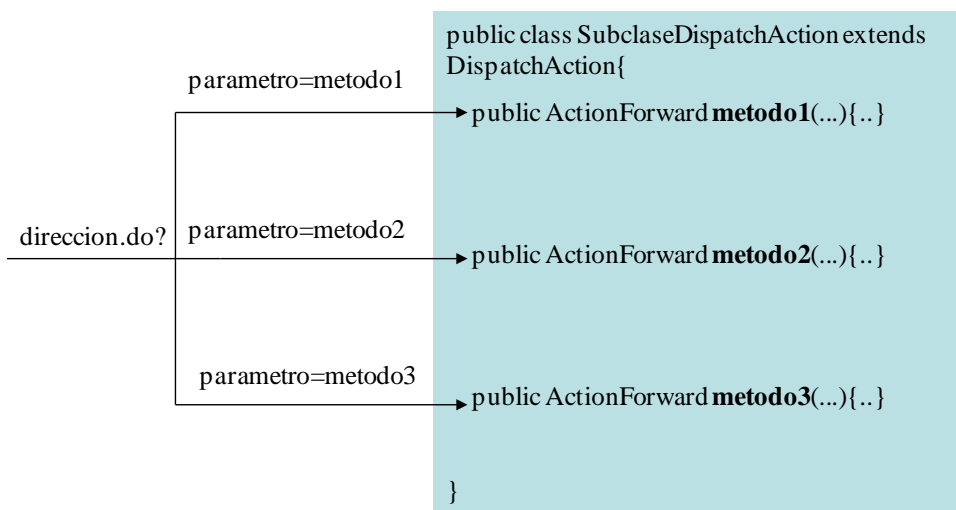


Fig. 21. Equivalencia entre valores de parámetro y métodos

Por ejemplo, si elegimos “operacion” como nombre de parámetro las siguientes URL provocarán la ejecución de los métodos `insertarItem()`, `eliminarItem()` y `obtenerItems()`, respectivamente, definidos en la clase `GestionCarrito` indicada anteriormente:

http://miservidor/miaplicacion/gestioncarrito.do?operacion=insertarItem

http://miservidor/miaplicacion/gestioncarrito.do?operacion=eliminarItem

http://miservidor/miaplicacion/gestioncarrito.do?operacion=obtenerItems

El parámetro que determina el nombre del método a ejecutar no tiene que ir necesariamente insertado en la URL, podría emplearse cualquier control del formulario enviado en la petición. Los posibles valores de este control serían entonces los que determinarían el método que debe ser ejecutado.

El nombre del parámetro o control deberá quedar especificado en la configuración del elemento `<action>` de `struts-config.xml` a través de su atributo `parameter`. En el ejemplo que estamos analizando la configuración de este elemento para la gestión de peticiones relativas al carrito de la compra quedaría como se indica a continuación:

```
<action path="/gestioncarrito"
        type="misclases.GestionCarritoAction"
        parameter="operacion"/>
```

En la definición de este elemento de ejemplo hemos omitido los parámetros `name` y `scope` por ser irrelevantes para el tema que estamos tratando, sin embargo en un caso real deberían ser especificados si la aplicación hace uso de algún objeto `ActionForm`.

PRÁCTICA 4.1. OPCIONES PARA LISTADO DE LLAMADAS DEL USUARIO

Descripción

Se trata de realizar una versión ampliada de la aplicación desarrollada en la práctica 3.1 del Capítulo anterior. En esta nueva versión proporcionaremos al usuario una serie de opciones de cara a visualizar el listado de llamadas de un número, opciones que consistirán en la posibilidad de elegir el tipo de llamadas a visualizar, pudiendo optar por visualizar todas las llamadas, aquellas que correspondan a un determinado tipo o las realizadas a partir de una determinada fecha.

La figura 22 muestra el nuevo aspecto que tendrá ahora la página de opciones de la aplicación, una vez que el usuario se ha validado.

opciones.jsp

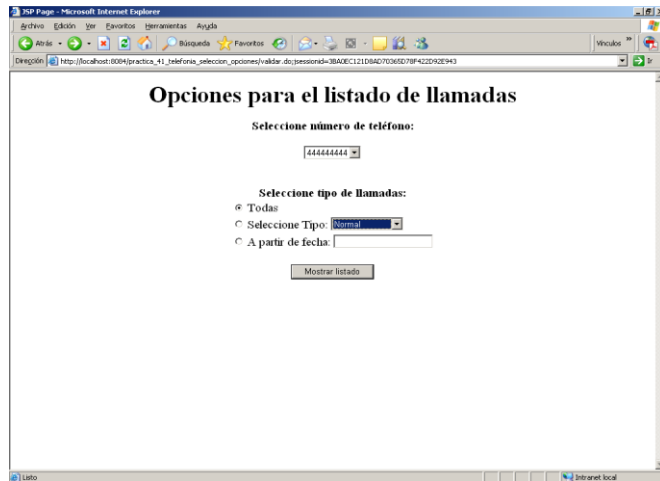


Fig. 22. Aspecto de la página de opciones

Desarrollo

En vez de crear una clase Action por cada opción a gestionar, utilizaremos una única clase DispatchAction con tres métodos, uno para cada opción. A través del parámetro llamado “operation” generado por los botones de radio enviaremos al servidor el nombre del método a ejecutar para cada opción elegida.

Tanto el número de teléfono, como la opción elegida, así como los datos adicionales asociados a cada una, serán encapsulados en el bean OpcionesForm.

Por otro lado, será necesario implementar un nuevo bean, llamado Tarifa, que encapsule los datos asociados a las tarifas. Este JavaBean será utilizado para generar la lista de tarifas, cuyos datos serán mostrados en la lista desplegable asociada al tipo de llamada.

Listado

Los siguientes listados corresponden a los nuevos elementos implementados en esta versión de la aplicación, así como a aquellos que han sido modificados, indicando en este caso en fondo sombreado los cambios introducidos.

struts-config.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">

<struts-config>
  <form-beans>
    <form-bean name="OpcionesForm"
      type="javabeans.OpcionesForm"/>
    <form-bean name="RegistroForm"
      type="javabeans.RegistroForm"/>
    <form-bean name="ValidacionForm"
      type="javabeans.ValidacionForm"/>

  </form-beans>

  <global-forwards>
    <forward name="login" path="/login.jsp"/>
    <forward name="toregistro" path="/registro.jsp"/>
  </global-forwards>

  <action-mappings>
    <action input="/" name="ValidacionForm"
      path="/validar" scope="request"
      type="servlets.ValidarAction">
      <forward name="bienvenida" path="/opciones.jsp"/>
      <forward name="error" path="/login.jsp"/>
    </action>
    <action input="/registro.jsp" name="RegistroForm"
      path="/registrar" scope="request"
      type="servlets.RegistrarAction">
      <forward name="registrado" path="/login.jsp"/>
    </action>
```



```
<action path="/listado" name="OpcionesForm"
        type="servlets.ListadoAction"
        parameter="operacion">
    <forward name="listado" path="/listado.jsp"/>
</action>
</action-mappings>
</struts-config>
```

Tarifa.java

```
package javabeans;

public class Tarifa {
    private int idtipotarifa;
    private String nombretarifa;
    public int getIdtipotarifa() {
        return idtipotarifa;
    }
    public void setIdtipotarifa(int idtipotarifa) {
        this.idtipotarifa = idtipotarifa;
    }
    public String getNombretarifa() {
        return nombretarifa;
    }
    public void setNombretarifa(String nombretarifa) {
        this.nombretarifa = nombretarifa;
    }
}
```

OpcionesForm.java

```
package javabeans;

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

public class OpcionesForm extends ActionForm {
    private int numero;
    private String operacion;
    private int tipo;
    private String fecha;
```

```

    public int getNumero() {
        return numero;
    }
    public void setNumero(int i) {
        numero = i;
    }
    public String getOperacion() {
        return operacion;
    }
    public void setOperacion(String operacion) {
        this.operacion = operacion;
    }
    public int getTipo() {
        return tipo;
    }
    public void setTipo(int tipo) {
        this.tipo = tipo;
    }
    public String getFecha() {
        return fecha;
    }
    public void setFecha(String fecha) {
        this.fecha = fecha;
    }
}

```

GestionTelefonos.java

```

package modelo;

import java.sql.*;
import java.util.*;
import javabeans.Tarifa;
public class GestionTelefonos {

    Datos dt;
    public GestionTelefonos(String driver,
                           String cadenacon) {
        dt=new Datos(driver,cadenacon);
    }
    public ArrayList<Integer> getTelefonos(String password){
        ArrayList<Integer> numeros=new ArrayList<Integer>();

```

```
try{
    Connection cn=dt.getConnection();
    //instrucción SQL para obtener los datos
    //del usuario indicado
    String query = "select telefono from telefonos ";
    query+="where password ='"+password+"'";
    Statement st =cn.createStatement();
    ResultSet rs = st.executeQuery(query);
    while(rs.next()){
        numeros.add(rs.getInt("telefono"));
    }
    dt.cierraConexion(cn);
}
catch(Exception e){
    e.printStackTrace();
}
finally{
    return numeros;
}
}
```

```
public ArrayList<Tarifa> getTarifas(){
    ArrayList<Tarifa> tarifas=new ArrayList<Tarifa>();
    try{
        Connection cn=dt.getConnection();
        Tarifa tf=null;
        //instrucción SQL para obtener todos
        //los tipos de tarifas
        String query = "select * from tarifas";
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(query);
        while(rs.next()){
            tf=new Tarifa();
            tf.setIdtipotarifa(rs.getInt("idtipotarifa"));
            tf.setNombretarifa(
                rs.getString("nombretarifa"));
            tarifas.add(tf);
        }
        dt.cierraConexion(cn);
    }
}
```

```

        catch(Exception e){
            e.printStackTrace();
        }

        finally{
            return tarifas;
        }
    }
}

```

GestionLlamadas.java

```

package modelo;

import java.sql.*;
import java.util.*;
import javax.swing.*;
import java.text.*;

public class GestionLlamadas {
    Datos dt;

    public GestionLlamadas(String driver, String cadenacon) {
        dt=new Datos(driver,cadenacon);
    }

    public ArrayList<LlamadaBean> getTodasLlamadasTelefono(
        int telefono){
        String query = "select * from llamadas where ";
        query+="telefono="+telefono;
        return getLlamadas(query,telefono);
    }

    public ArrayList<LlamadaBean>
        getLlamadasTelefonoPorTipoTarifa(int telefono,
            int idtipotarifa){
        String query = "select * from llamadas where ";
        query+="telefono="+telefono+
            " and idtipotarifa="+idtipotarifa;
        return getLlamadas(query,telefono);
    }

    //método que obtiene las llamadas, común para los dos
    //métodos anteriores
    private ArrayList<LlamadaBean> getLlamadas(String sql,
        int telefono){

```

```

        ArrayList<LlamadaBean> llamadas=
            new ArrayList<LlamadaBean>();
    try{
        Connection cn=dt.getConexion();
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(sql);
        while(rs.next()){
            LlamadaBean llamada=new LlamadaBean();
            llamada.setTelefono(telefono);
            llamada.setDestino(rs.getInt("destino"));
            llamada.setDuracion(rs.getInt("duracion"));
            llamada.setTarifa(this.getTarifa(
                rs.getInt("idtipotarifa")));
            llamada.setCoste(rs.getDouble("coste"));
            llamada.setFecha(rs.getString("fecha"));
            llamadas.add(llamada);
        }
        dt.cierraConexion(cn);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        return llamadas;
    }
}

```

```

public ArrayList<LlamadaBean>
    getLlamadasTelefonoApartirFecha(int telefono,
                                     String fecha){
    ArrayList<LlamadaBean> llamadas=
        new ArrayList<LlamadaBean>();
    DateFormat feuropa=DateFormat.getDateInstance(
        DateFormat.SHORT,Locale.getDefault());
    try{
        Connection cn=dt.getConexion();
        String query = "select * from llamadas where ";
        query+="telefono="+telefono;
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(query);
    }
}

```

```

        while(rs.next()){
            //convierte las dos fechas a un formato común
            //para poder compararlas
            java.util.Date fechallamada=rs.getDate("fecha");
            java.util.Date fechacomparacion=
                feuropa.parse(fecha);
            //almacena en el ArrayList únicamente las
            //llamadas cuya fecha sea posterior a la
            // suministrada como parámetro
            if(fechallamada.after(fechacomparacion)){
                LlamadaBean llamada=new LlamadaBean();
                llamada.setTelefono(telefono);
                llamada.setDestino(rs.getInt("destino"));
                llamada.setDuracion(rs.getInt("duracion"));
                llamada.setTarifa(this.getTarifa(
                    rs.getInt("idtipotarifa")));
                llamada.setCoste(rs.getDouble("coste"));
                llamada.setFecha(
                    feuropa.format(fechallamada));
                llamadas.add(llamada);
            }
        }
        dt.cierraConexion(cn);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        return llamadas;
    }
}

private String getTarifa(int id){
    String tarifa=null;
    try{
        Connection cn=dt.getConexion();
        String query = "select nombretarifa from tarifas ";
        query += "where idtipotarifa="+id;
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(query);

```

```

        if(rs.next()){
            tarifa=rs.getString("nombretarifa");
        }
        dt.cierraConexion(cn);
    }
    catch(Exception e){
        e.printStackTrace();
    }

    finally{
        return tarifa;
    }
}
}

```

ListadoAction.java

```
package servlets;

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

import javax beans.*;
import modelo.*;
import java.util.*;

public class ListadoAction extends DispatchAction {
    public ActionForward todas(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        OpcionesForm of=(OpcionesForm)form;
        GestionLlamadas gl=getGestionLlamadas(request);
        request.setAttribute("llamadas",
            gl.getTodasLlamadasTelefono(of.getNumero()));
        return mapping.findForward("listado");
    }

    public ActionForward portipo(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
```

```

        throws Exception {
            OpcionesForm of=(OpcionesForm)form;
            GestionLlamadas gl=getGestionLlamadas(request);
            request.setAttribute("llamadas",
                                gl.getLlamadasTelefonoPorTipoTarifa(
                                    of.getNumero(),of.getTipo()));
            return mapping.findForward("listado");
        }

        public ActionForward porfecha(ActionMapping mapping,
                                      ActionForm form,
                                      HttpServletRequest request,
                                      HttpServletResponse response)
            throws Exception {
            OpcionesForm of=(OpcionesForm)form;
            GestionLlamadas gl=getGestionLlamadas(request);
            request.setAttribute("llamadas",
                                gl.getLlamadasTelefonoApartirFecha(
                                    of.getNumero(),of.getFecha()));
            return mapping.findForward("listado");
        }
    }

    private GestionLlamadas getGestionLlamadas(
        HttpServletRequest request){
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
        String cadenaCon=this.getServlet().
            getServletContext().getInitParameter("cadenaCon");
        return new GestionLlamadas(driver,cadenaCon);
    }
}

```

opciones.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@      taglib      uri="http://struts.apache.org/tags-html"
prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                Transitional//EN"

```



```
"http://www.w3.org/TR/html4/loose.dtd">
<html:html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <center>
      <h1>Opciones para el listado de llamadas </h1>
      <html:form action="/listado" method="POST">
        <b> Seleccione número de teléfono:</b>
        <br/> <br/>
        <html:select property="numero">
          <!--Recupera el bean ValidacionForm
            almacenado en una variable de petición-->
          <html:options name="ValidacionForm"
            property="telefonos" />
        </html:select>
        <br/> <br/> <br/>
        <b>Seleccione tipo de llamadas:</b><br/>
        <table>
          <tr>
            <td><html:radio tabindex="0"
              property="operacion" value="todas"/>
            </td><td align="left">Todas</td>
          </tr>
          <tr>
            <td><html:radio property="operacion"
              value="portipo"/></td>
            <td align="left">Seleccione Tipo:
              <html:select property="tipo">
                <!--Recupera el ArrayList de tarifas
                  almacenada en una variable de petición-->
                <html:optionsCollection
                  name="ValidacionForm"
                  property="telefonos"
                  value="idtipotarifa"
                  label="nombretarifa"/>
              </html:select>
            </td>
          </tr>
        </table>
      </html:form>
    </center>
  </body>
</html>
```

```
</tr>
<tr>
    <td><html:radio property="operacion"
        value="porfecha"/></td>
    <td align="left">A partir de fecha:
    <html:text property="fecha"/></td>
</tr>
<tr>
    <td colspan="2"><br/>
    <html:submit value="Mostrar listado"/>
    </td>
</tr>
</table>
</html:form>
</center>
</body>
</html:html>
```

4.2.2 Clase LookupDispatchAction

Se trata de una subclase de `DispatchAction` que, al igual que ésta, tiene como misión gestionar varias peticiones en una misma clase mediante la definición de un método personalizado para cada acción, utilizándose el valor de un parámetro enviado con cada petición para determinar el método que se tiene que ejecutar.

A diferencia de `DispatchAction`, donde el valor del parámetro contiene directamente el nombre del método asociado a la acción, `LookupDispatchAction` utiliza el valor de este parámetro para localizar en el archivo de recursos `ApplicationResource.properties` una clave asociada al mismo entre todas las parejas `clave=valor` almacenadas. A partir de este dato el objeto recupera el nombre real del método que tiene que ejecutar, para ello tendrá que acceder a una tabla de tipo `Map` donde se encuentran almacenados los nombres de los métodos del objeto con sus correspondientes claves asociadas (figura 23).

La implementación por defecto del método `execute()` existente en `LookupDispatchAction` invoca al método `getKeyMethodMap()` de la propia clase para obtener el mapa de métodos con sus correspondientes claves. Es responsabilidad del programador sobrescribir este método e incluir en él las instrucciones apropiadas para generar esta colección.

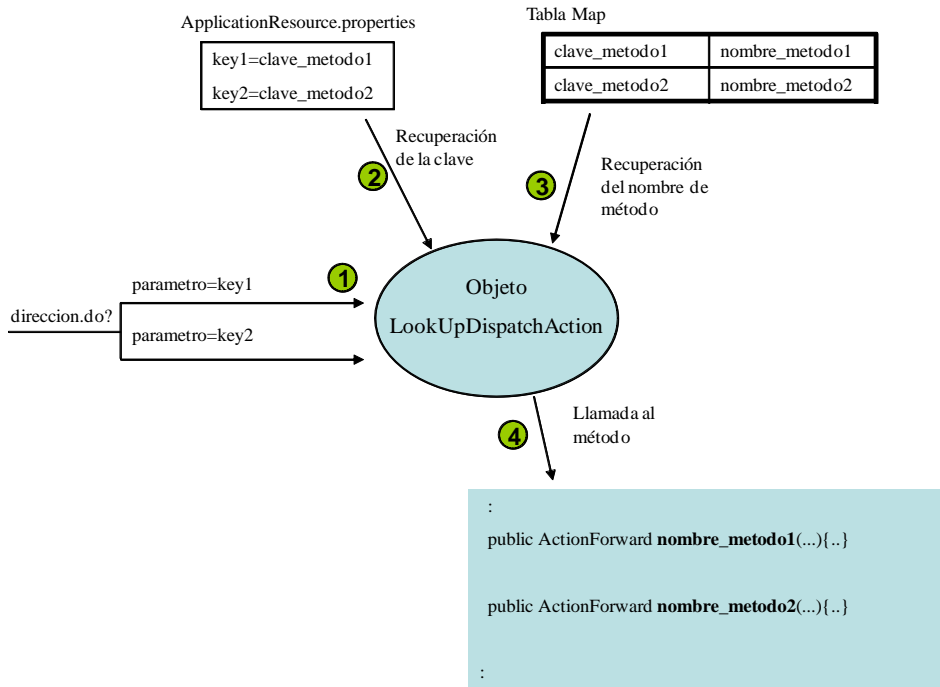


Fig. 23. Tratamiento de peticiones con *LookUpDispatchAction*

Por ejemplo, volvamos al caso de la clase `GestionCarritoAction` comentada anteriormente y supongamos que queremos asociar los siguientes valores del parámetro `operación`: “opcion1”, “opcion2” y “opcion3” a los métodos `insertarItem()`, `eliminarItem()` y `obtenerItems()`, respectivamente. Lo primero que debemos hacer es asignar una clave a cada uno de los valores posibles del parámetro, por ejemplo:

oper.insertar=opcion1

oper.eliminar=opcion2

oper.recuperar=opcion3

Las líneas anteriores se incluirán dentro del archivo de recursos `ApplicationResource.properties` de la aplicación Web.

A continuación debemos implementar el método `getKeyMethodMap()` en la clase `GestionCarritoAction`, de manera que genere una tabla `Map` con los nombres de los métodos de la clase utilizando como claves las definidas anteriormente en el

archivo de recursos. Esta será la nueva implementación de la clase `GestionCarritoAction`:

```
public class GestionCarritoAction extends LookupDispatchAction{

    public Map getKeyMethodMap(){

        //definición de mapa de métodos

        Map mp=new HashMap();

        mp.put("opcion.insertar","insertarItem");

        mp.put("opcion.eliminar","eliminarItem");

        mp.put("opcion.recuperar","obtenerItems");

        return mp;

    }

    public ActionForward insertarItem(...) {

        //método que lleva a cabo la inserción de un elemento

        //en el carrito

    }

    public ActionForward eliminarItem(...) {

        //método que lleva a cabo la eliminación de un elemento

        //en el carrito

    }

    public ActionForward obtenerItems(...) {

        //método que lleva a cabo la recuperación

        //del contenido completo del carrito

    }

}
```

Este desacoplamiento entre los nombres de los métodos y los valores del parámetro enviado en la petición permite que puedan ser modificados los valores de éste sin necesidad de alterar el código de la aplicación, tan sólo habrá que hacer los cambios pertinentes en el archivo `ApplicationResource.properties`.

Al igual que sucede en el caso `DispatchAction`, en el registro del elemento `<action>` asociado al objeto `LookupDispatchAction` habrá que especificar en el atributo *parameter* el nombre del parámetro enviado en la petición.

Como ejemplo de utilización de esta clase, vamos a gestionar las opciones del listado de llamadas de la aplicación de la práctica 4.1 mediante un objeto `LookupDispatchAction`. Lo primero será añadir las siguientes cadenas con sus claves al archivo `ApplicationResource.properties`:

opcion.todas=todas

opcion.portipo=tipo

opcion.porfecha=fecha

El código de la clase `ListadoAction` sería prácticamente igual a la de la práctica 4.1, añadiendo simplemente la implementación del método `getKeyMethodMap()`:

```
package servlets;

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

import javax beans.*;
import modelo.*;
import java.util.*;

public class ListadoAction extends DispatchAction {
    public Map getKeyMethodMap(){
        //definición de mapa de métodos con sus
        // correspondientes claves
        Map mp=new HashMap();
        mp.put("opcion.todas","todas");
        mp.put("opcion.portipo","portipo");
        mp.put("opcion.porfecha","porfecha");
        return mp;
    }
}
```

```
}  
  
public ActionForward todas(ActionMapping mapping,  
                           ActionForm form,  
                           HttpServletRequest request,  
                           HttpServletResponse response)  
    throws Exception {  
    OpcionesForm of=(OpcionesForm)form;  
    GestionLlamadas gl=getGestionLlamadas(request);  
    request.setAttribute("llamadas",  
        gl.getTodasLlamadasTelefono(of.getNumero()));  
    return mapping.findForward("listado");  
}  
  
public ActionForward portipo(ActionMapping mapping,  
                              ActionForm form,  
                              HttpServletRequest request,  
                              HttpServletResponse response)  
    throws Exception {  
    OpcionesForm of=(OpcionesForm)form;  
    GestionLlamadas gl=getGestionLlamadas(request);  
    request.setAttribute("llamadas",  
        gl.getLlamadasTelefonoPorTipoTarifa(  
            of.getNumero(),of.getTipo()));  
    return mapping.findForward("listado");  
}  
  
public ActionForward porfecha(ActionMapping mapping,  
                               ActionForm form,  
                               HttpServletRequest request,  
                               HttpServletResponse response)  
    throws Exception {  
    OpcionesForm of=(OpcionesForm)form;  
    GestionLlamadas gl=getGestionLlamadas(request);  
    request.setAttribute("llamadas",  
        gl.getLlamadasTelefonoApartirFecha(  
            of.getNumero(),of.getFecha()));  
    return mapping.findForward("listado");  
}  
  
private GestionLlamadas getGestionLlamadas(  
    HttpServletRequest request){  
    String driver=this.getServlet().  
        getServletContext().getInitParameter("driver");  
    String cadenaCon=this.getServlet().
```

```

        getServletContext().getInitParameter("cadenaCon");
        return new GestionLlamadas(driver, cadenaCon);
    }
}

```

En cuanto a la página `opciones.jsp`, tan sólo modificaríamos los valores del parámetro *operaciones* (para que no tengan por qué coincidir con los nombres de los métodos), según lo indicado en el archivo `ApplicationResource.properties`:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@ taglib uri=http://struts.apache.org/tags-html
        prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <h1>Opciones para el listado de llamadas </h1>
            <html:form action="/listado" method="POST">
                <b> Seleccione número de teléfono:</b>
                <br/> <br/>
                <html:select property="numero">
                    <!--Recupera el bean ValidacionForm
                         almacenado en una variable de petición-->
                    <html:options name="ValidacionForm"
                                property="telefonos" />
                </html:select>
                <br/> <br/> <br/>
                <b>Seleccione tipo de llamadas:</b><br/>
                <table>
                <tr>
                    <td><html:radio tabIndex="0"
                                property="operacion" value="todas"/>

```

```

        </td><td align="left">Todas</td>
    </tr>
    <tr>
        <td><html:radio property="operacion"
            value="tipo"/></td>
        <td align="left">Seleccione Tipo:
        <html:select property="tipo">
            <!--Recupera el ArrayList de tarifas
            almacenada en una variable de petición-->
            <html:optionsCollection
                name="ValidacionForm"
                property="telefonos"
                value="idtipotarifa"
                label="nombretarifa"/>
            </html:select>
        </td>
    </tr>
    <tr>
        <td><html:radio property="operacion"
            value="fecha"/></td>
        <td align="left">A partir de fecha:
        <html:text property="fecha"/></td>
    </tr>
    <tr>
        <td colspan="2"><br/>
        <html:submit value="Mostrar listado"/>
        </td>
    </tr>
</table>
</html:form>
</center>
</body>
</html:html>

```

4.2.3 Clase MappingDispatchAction

Se trata también de una subclase de `DispatchAction` que, al igual que las anteriores, permite definir en la misma clase un grupo de métodos para la gestión de peticiones diferentes.

En este caso no se utiliza ningún parámetro que permita determinar el método a ejecutar, sino que cada petición utilizará un *path* diferente para acceder al mismo objeto, lo que obligará a definir en el `struts-config.xml` tantos elementos `<action>` asociados a la misma subclase de `MappingDispatchAction` como peticiones distintas deba gestionar el objeto. **Utilizando el atributo *parameter* del elemento `<action>` se indicará para cada petición el nombre del método que se debe ejecutar**, consiguiendo así el desacoplamiento entre el *path* y el nombre del método (figura 24).

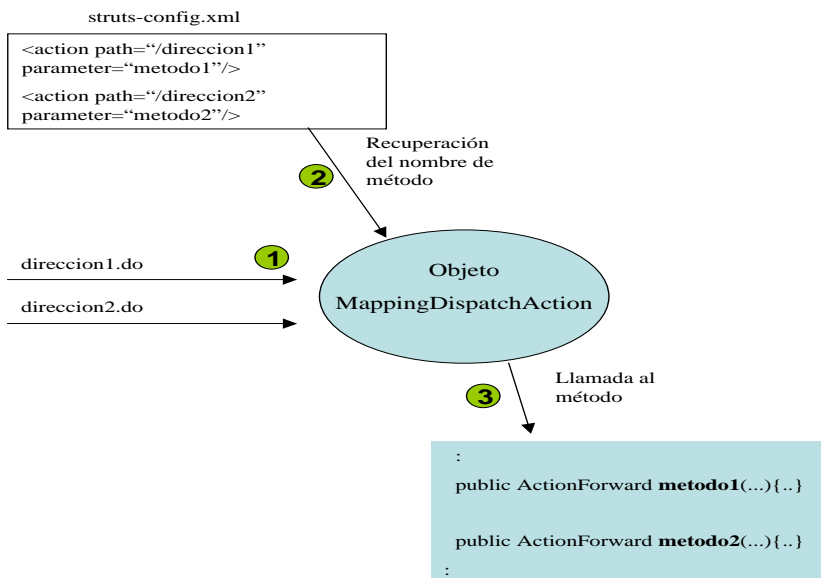


Fig. 24. Tratamiento de peticiones con *MappingDispatchAction*

Para entender el funcionamiento volvamos al ejemplo del carrito de la compra que hemos utilizado con las clases anteriores. Supongamos que tenemos tres enlaces en la página cliente que lanzan las peticiones para la realización de las tres operaciones descritas sobre el carrito:

`Añadir elemento`

`Eliminar elemento`

`Recuperar contenido`

Para que las tres peticiones provoquen la ejecución del objeto `GestionCarritoAction` deberíamos registrar los siguientes elementos `<action>` en el archivo `struts-config.xml`:

```
<action path="/agregar" type="misclases.GestionCarritoAction"
        parameter="insertarItem" />
<action path="/eliminar" type="misclases.GestionCarritoAction"
        parameter="eliminarItem" />
<action path="/recuperar"
        type="misclases.GestionCarritoAction"
        parameter="obtenerItems" />
```

Los valores indicados en el atributo *parameter* de cada elemento `<action>` anterior representan los nombres de los métodos de la clase `GestionCarritoAction` asociados a cada acción. La estructura de esta clase será ahora la que se indica en el siguiente listado:

```
public class GestionCarritoAction extends MappingDispatchAction{

    public ActionForward insertarItem(...){

        //método que lleva a cabo la inserción de un elemento

        //en el carrito

    }

    public ActionForward eliminarItem(...){

        //método que lleva a cabo la eliminación de un elemento

        //en el carrito

    }

    public ActionForward obtenerItems(...){

        //método que lleva a cabo la recuperación

        //del contenido completo del carrito

    }

}
```

PRÁCTICA 4.2. INFORMACIÓN TELEFÓNICA

Descripción

La presente práctica consistirá en la implementación de una variante de la aplicación desarrollada en la práctica 4.1. En esta ocasión, una vez validado el usuario, se le presentará una página de opciones de información que le permitirán acceder a la lista de teléfonos registrados a su nombre, conocer el consumo acumulado o visualizar el histórico de llamadas. La figura 25 ilustra las páginas de la aplicación, sin incluir las de validación y registro.

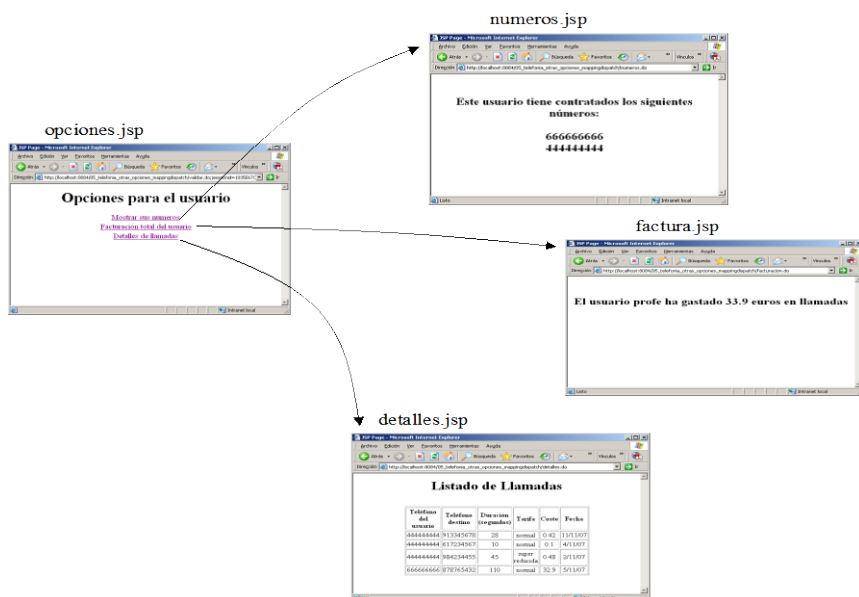


Fig. 25. Páginas de la aplicación

Desarrollo

La gestión de las opciones será llevada a cabo por la clase `OpcionesAction` que en este caso será una subclase de `MappingDispatchAction` con tres métodos asociados a cada uno de los enlaces de la página `opciones.jsp`, asociación que como hemos indicado se realizará en cada elemento `<action>` de `struts-config.xml`.

Por otro lado, se ha modificado el ámbito del objeto `ValidacionForm`, pasando de `request` a `session`. De esta manera, será posible tener acceso al login de usuario desde la página `factura.jsp` a fin de personalizar el mensaje mostrado al usuario.

Listado

Seguidamente mostraremos el código de los nuevos elementos desarrollados en esta práctica, así como las modificaciones realizadas en los ya creados.

struts-config.xml

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

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">

<struts-config>
    <form-beans>
        <form-bean name="OpcionesForm"
            type="javabeans.OpcionesForm"/>
        <form-bean name="RegistroForm"
            type="javabeans.RegistroForm"/>
        <form-bean name="ValidacionForm"
            type="javabeans.ValidacionForm"/>
    </form-beans>
    <global-forwards>
        <forward name="login" path="/login.jsp"/>
        <forward name="toregistro" path="/registro.jsp"/>
    </global-forwards>
    <action-mappings>
        <!--se define como ámbito de sesión para que
            siempre pueda recordar qué usuario
            es durante toda la navegación-->
        <action input="/" name="ValidacionForm"
            path="/validar" scope="session"
            type="servlets.ValidarAction">
            <forward name="bienvenida" path="/opciones.jsp"/>
            <forward name="error" path="/login.jsp"/>
        </action>
        <action input="/registro.jsp" name="RegistroForm"
            path="/registrar" scope="request"
            type="servlets.RegistrarAction">
```

```

        <forward name="registrado" path="/login.jsp"/>
    </action>
    <action path="/facturacion"
            type="servlets.OpcionesAction"
            parameter="factura">
        <forward name="vistafactura"
                path="/factura.jsp"/>
    </action>
    <action path="/numeros"
            type="servlets.OpcionesAction"
            parameter="numeros">
        <forward name="vistanumeros"
                path="/numeros.jsp"/>
    </action>
    <action path="/detalles"
            type="servlets.OpcionesAction"
            parameter="detalles">
        <forward name="vistadetalles"
                path="/detalles.jsp"/>
    </action>
</action-mappings>
</struts-config>

```

GestionLlamadas.java

```

package modelo;

import java.sql.*;
import java.util.*;
import javax.swing.*;
import java.text.*;

public class GestionLlamadas {
    Datos dt;

    public GestionLlamadas(String driver, String cadenacon) {
        dt=new Datos(driver,cadenacon);
    }

    public ArrayList<LlamadaBean> getLlamadas(
        String password){
        String query = "select * from llamadas where ";
        query+="telefono in (select telefono from
                        telefonos where ";
        query+="password ='"+password+"' ) order by telefono";
    }
}

```

```
        return getListado(query);
    }
    //método auxiliar de apoyo al anterior
    private ArrayList<LlamadaBean> getListado(String sql){
        //se utiliza para poder aplicar el formato
        //de fecha europeo
        DateFormat feuropa=DateFormat.getDateInstance(
            DateFormat.SHORT,Locale.getDefault());
        ArrayList<LlamadaBean> llamadas=
            new ArrayList<LlamadaBean>();
        try{
            Connection cn=dt.getConnection();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(sql);
            while(rs.next()){
                LlamadaBean llamada=new LlamadaBean();
                llamada.setTelefono(rs.getInt("telefono"));
                llamada.setDestino(rs.getInt("destino"));
                llamada.setDuracion(rs.getInt("duracion"));
                llamada.setTarifa(this.getTarifa(
                    rs.getInt("idtipotarifa")));
                llamada.setCoste(rs.getDouble("coste"));
                llamada.setFecha(feuropa.format(
                    rs.getDate("fecha")));
                llamadas.add(llamada);
            }
            dt.cierraConexion(cn);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            return llamadas;
        }
    }
    public double getFactura(String password){
        String query = "select sum(coste) as total from
                        llamadas where ";
        query+="telefono in (select telefono from telefonos ";
        query+=" where password = '"+password+"'";
        double total=0.0;
```

```

        try{
            Connection cn=dt.getConexion();
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            if(rs.next()){
                total=rs.getDouble("total");
            }
            dt.cierraConexion(cn);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            return total;
        }
    }

private String getTarifa(int id){
    String tarifa=null;
    try{
        Connection cn=dt.getConexion();
        String query = "select nombretarifa from tarifas ";
        query += "where idtipotarifa="+id;
        Statement st =cn.createStatement();
        ResultSet rs = st.executeQuery(query);
        if(rs.next()){
            tarifa=rs.getString("nombretarifa");
        }
        dt.cierraConexion(cn);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    finally{
        return tarifa;
    }
}
}

```

OpcionesAction.java

```
package servlets;
```

```
import javax.servlet.http.*;

import org.apache.struts.action.*;
import org.apache.struts.actions.*;
import javax beans.*;
import modelo.*;
import java.util.*;

public class OpcionesAction extends MappingDispatchAction {
    public ActionForward factura(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        HttpSession sesion=request.getSession();
        ValidacionForm vf=(ValidacionForm)sesion.
            getAttribute("ValidacionForm");
        GestionLlamadas gl=getGestionLlamadas(request);

        request.setAttribute("precio",
            gl.getFactura(vf.getPassword()));
        return mapping.findForward("vistafactura");
    }
    public ActionForward numeros(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        //el FormBean ValidacionForm contiene los teléfonos
        //asociados al usuario
        return mapping.findForward("vistanumeros");
    }
    public ActionForward detalles(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        HttpSession sesion=request.getSession();
        ValidacionForm vf=(ValidacionForm)sesion.
            getAttribute("ValidacionForm");
```



```

        GestionLlamadas gl=getGestionLlamadas(request);
        request.setAttribute("llamadas",
            gl.getLlamadas(vf.getPassword()));
        return mapping.findForward("vistadetalles");
    }
    private GestionLlamadas getGestionLlamadas(
        HttpServletRequest request){
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
        String cadenaCon=this.getServlet().
            getServletContext().getInitParameter("cadenaCon");
        return new GestionLlamadas(driver,cadenaCon);
    }
}

```

opciones.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
        <title>JSP Page</title>
        <style>
            .td{margin-left:0px}
        </style>
    </head>
    <body>
        <center>
            <h1>Opciones para el usuario </h1>
            <table>
                <tr>
                    <td><html:link page="/numeros.do">
                        Mostrar sus numeros

```

```

        </html:link>
    </td>
</tr>
<tr>
    <td><html:link page="/facturacion.do">
        Facturación total del usuario
    </html:link>
    </td>
</tr>
<tr>
    <td><html:link page="/detalles.do">
        Detalles de llamadas
    </html:link>
    </td>
</tr>
</table>
</center>
</body>
</html:html>

```

numeros.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                        Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <br/><br/>
            <h2> Este usuario tiene contratados
            los siguientes números: <br/><br/>
            <%ArrayList<Integer> lista=
                ((ValidacionForm)session.

```

```

        getAttribute("ValidacionForm")).
            getTelefonos();
        for(int i=0;i<lista.size();i++){%>
        <%=lista.get(i).toString()%>
        <br/>
        <}%>
    </h2>
</center>
</body>
</html>

```

detalles.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                        Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
                                charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <%ArrayList<LlamadaBean> llamadas=(ArrayList)request.
                                getAttribute("llamadas");%>
            <h1>Listado de Llamadas</h1> <br/>
            <table border="1" width="60%">
                <th>Teléfono del usuario</th>
                <th>Teléfono destino</th>
                <th>Duración (segundos)</th>
                <th>Tarifa</th>
                <th>Coste</th>
                <th>Fecha</th>
                <%for (int i=0;i<llamadas.size();i++){
                    LlamadaBean llamada=llamadas.get(i);%>
                <tr>
                    <td><%=llamada.getTelefono()%></td>

```

```

        <td><%=llamada.getDestino()%></td>
        <td><%=llamada.getDuracion()%></td>
        <td><%=llamada.getTarifa()%></td>
        <td><%=llamada.getCoste()%></td>
        <td><%=llamada.getFecha()%></td>
    </tr>
<}%%>
</table>
<br/>
</center>
</body>
</html>

```

factura.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">

        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <br/><br/>
            <%ValidacionForm usuario=(ValidacionForm)session.
                getAttribute("ValidacionForm");%>
            <h2>El usuario <%=usuario.getUsuario()%> ha gastado
                <b><%=request.getAttribute("precio")%></b>
                euros en llamadas
            </h2>
        </center>
    </body>
</html>

```

4.2.4 Clase ActionForm

Como vimos en el Capítulo anterior la clase ActionForm facilita la creación de JavaBeans para la gestión de los datos de usuario procedentes de un formulario XHTML.

Basta con crear una subclase de ActionForm en la que se incluyan los miembros y métodos *set/get* y asociar la misma con el objeto Action correspondiente, para que las clases ActionServlet y RequestProcessor se encarguen automáticamente de todo el proceso de instanciación, recuperación y rellenado del objeto.

4.2.4.1 CICLO DE VIDA DE UN ACTIONFORM

La mejor manera de comprender el tratamiento que realiza Struts con este objeto es analizar su ciclo de vida. En la figura 26 tenemos un diagrama en el que se aprecian los estados por los que puede pasar un ActionForm durante el procesamiento de una petición en el Controlador.

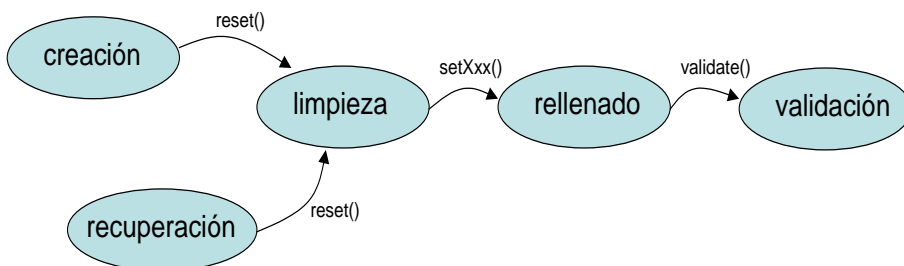


Fig. 26. *Ciclo de vida de un ActionForm*

Pasemos a continuación a describir estos estados:

- **Creación.** Cuando llega al Controlador una petición desde el cliente para la que se determina que debe ser utilizado un determinado ActionForm que almacene los datos de usuario, se comprueba la existencia de una instancia previa de la clase en el ámbito definido mediante el atributo *scope* del elemento <action>. En caso de no encontrarla, el Controlador procederá a la creación de una nueva instancia y a su almacenamiento en el ámbito especificado.

Como ya se indicó al principio del Capítulo durante el análisis del procesamiento de una petición, este proceso es realizado por el método *processActionForm()* del objeto *RequestProcessor*.

- **Recuperación.** Retomando el proceso descrito anteriormente, si se encuentra una instancia de *ActionForm* asociada al tipo de petición en curso, el Controlador obtiene una referencia a dicha instancia para su reutilización.
- **Limpieza.** Una vez que el Controlador dispone de la referencia al objeto *ActionForm* (nuevo o ya existente), procede a invocar al método *reset()* del mismo. El programador puede sobrescribir este método e incluir algún tipo de operación de limpieza o inicialización de los datos miembro del bean, a fin de que la existencia de datos previos (algo que puede suceder al reutilizar instancias ya existentes) no desvirtúe el nuevo estado de la instancia.
- **Rellenado.** Tras el reseteo de la instancia el Controlador procede a rellenar los datos miembro de la misma con el contenido del formulario enviado en la petición. Para llevar a cabo esta operación el Controlador realiza de forma automática las conversiones de datos necesarias y posteriormente invoca a los métodos *setXxx()* asociados con cada dato.
- **Validación.** Antes de pasar la petición al objeto *Action* encargado de su procesamiento se procede a la validación de los datos introducidos en el bean, para lo cual el Controlador, desde el método *processValidate()* del objeto *RequestProcessor*, realiza una llamada al método *validate()* de *ActionForm*. Será responsabilidad del programador sobrescribir este método e incluir en él las instrucciones encargadas de validar los datos, según los criterios establecidos por la propia lógica de la aplicación. El formato del método *validate()* es el que se muestra a continuación:

public ActionErrors() validate()

La llamada al método *validate()* devolverá al Controlador un objeto *ActionErrors* con los errores detectados durante el proceso de validación. Si el valor devuelto por el método es **distinto de null**, entonces **el Controlador cancelará el procesamiento de la petición** y en vez de invocar al método *execute()* del objeto *Action*

asociado a la misma, redirigirá al usuario a la página indicada en el atributo *input* del elemento `<action>`, que normalmente suele ser la misma página desde la que se lanzó la petición.

Por ejemplo, dada la siguiente definición del elemento `<action>`:

```
<action name= "LoginForm"
```

```
:
```

```
input= "/login.jsp"/>
```

En caso de que el método *validate()* del objeto `LoginForm` devuelva un valor distinto de *null* al ser invocado, el usuario será redirigido a la página `login.jsp`.

Se puede impedir la llamada al método *validate()* del objeto `ActionForm` para una determinada acción, especificando el valor *false* en el atributo *validate* del elemento `<action>` correspondiente. De manera predeterminada la validación de objetos `ActionForm` se encuentra habilitada.

La siguiente implementación de ejemplo del método *validate()* generaría un error en caso de que la longitud del campo `password` del bean fuera inferior a 8 caracteres:

```
:
public ActionErrors validate(){
    ActionErrors errores=null;
    if(this.password.length()<8){
        errores=new ActionErrors();
        errores.add(new ActionMessage("error",
                                      "error.password.lowlength"));
    }
    return errores;
}
:
```

Para poder mostrar los mensajes de los errores generados por *validate()* en la página a la que ha sido redirigido el usuario, utilizaremos el tag `<html:errors/>`. Este elemento recorre todos los objetos de error registrados en `ActionErrors` y, para cada uno de

ellos, muestra una línea con el mensaje de error asociado en el lugar de la página donde se encuentra el elemento.

4.2.5 ActionErrors y ActionMessage

Tal y como se acaba de indicar en el punto anterior, la clase `ActionErrors` encapsula los mensajes de error generados en el interior del método `validate()` de un `ActionForm`. Cada error está descrito a su vez mediante un objeto `ActionMessage`, el cual encapsula la clave del mensaje de error asociado. Los mensajes de error con sus claves correspondientes se registran en el archivo `ApplicationResource.properties`.

Para añadir un error a la colección `ActionErrors` se utilizará el método `add()` de esta clase, pasando como argumento en la llamada un nombre de propiedad asociada al error y el objeto `ActionMessage` que lo describe:

```
public void add(String property, ActionMessage mensaje)
```

El nombre de la propiedad puede ser utilizado como valor del atributo *property* del tag `<html:errors/>` en caso de que se quiera mostrar únicamente el mensaje asociado a esa propiedad. En el ejemplo de método `validate()` presentado en el punto anterior se crea un objeto `ActionErrors` formado por un único error, error cuyo mensaje asociado se encuentra registrado en el archivo `ApplicationResource.properties` con la clave “`error.password.lowlength`”.

PRÁCTICA 4.3. CONTROL DE DATOS DE REGISTRO

Descripción

En esta práctica implementaremos el código necesario para realizar la verificación de los datos de la página de registro de usuarios utilizada en prácticas anteriores (figura 27), antes de que éstos sean procesados por la aplicación.

Los criterios para considerar válidos los datos serán los siguientes:

- Los campos usuario y password serán obligatorios.
- El password introducido en segundo lugar debe coincidir con el primero.
- La dirección de correo electrónico deberá contar con el carácter @.

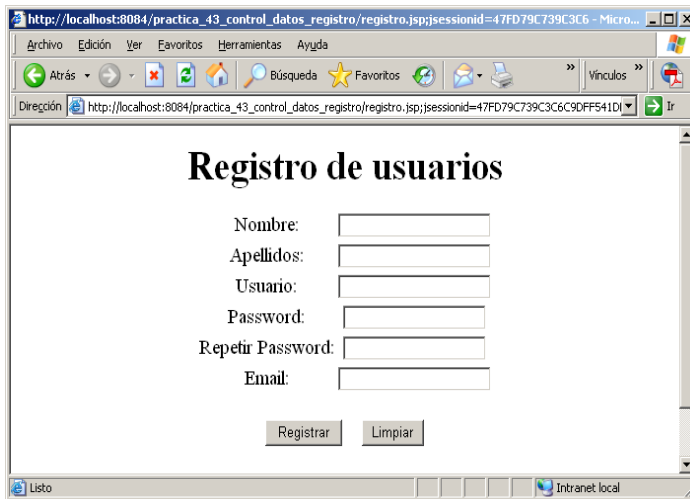


Fig. 27. *Página de registro de usuarios*

En caso de incumplirse alguno de los criterios de validación anteriores el usuario será mantenido en la propia página de registro, indicándole mediante algún mensaje descriptivo el motivo del error.

Desarrollo

Los anteriores criterios de validación serán codificados dentro del método `validate()` de la clase `RegistroForm`, creando un objeto `ActionMessage` por cada criterio incumplido.

En cuanto a los mensajes de error, serán registrados en el archivo de recursos con sus correspondientes claves. Utilizando un elemento `<html:errors/>` por cada tipo de error mostraremos el mensaje correspondiente al lado de cada control.

Listado

A continuación mostraremos únicamente el contenido de los componentes involucrados en este proceso de validación.

ApplicationResource.properties

Los mensajes de error con sus correspondientes claves serán:

```
error.usuario=El campo usuario no puede estar vacío
```

```
error.password=El campo password no puede estar vacío
error.email=Debe introducir una dirección de correo válida
error.password.nomatch=El password introducido en segundo
                    lugar no coincide con el primero
```

RegistroForm.java

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

public class RegistroForm extends
        org.apache.struts.action.ActionForm {
    private String nombre;
    private String apellidos;
    private String usuario;
    private String password;
    private String passwordrep;
    private String email;

    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
```

```
        this.password = password;
    }
    public String getPasswordrep() {
        return passwordrep;
    }
    public void setPasswordrep(String passwordrep) {
        this.passwordrep = passwordrep;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public ActionErrors validate(ActionMapping mapping,
                                HttpServletRequest req){
        ActionErrors errors=new ActionErrors();
        //comprueba si se ha suministrado un valor para
        //el campo usuario
        if(usuario==null||usuario.equals("")){
            errors.add("usuario",
                      new ActionMessage("error.usuario"));
        }
        //comprueba si se ha suministrado un valor para
        //el campo password
        if(password==null||password.equals("")){
            errors.add("password",
                      new ActionMessage("error.password"));
        }
        //comprueba si el password introducido en segundo
        //lugar coincide con el primero
        if(!password.equals(passwordrep)){
            errors.add("passwordnomatch",
                      new ActionMessage("error.password.nomatch"));
        }
        //comprueba si la dirección de email es válida
        if(email.indexOf("@")==-1){
            errors.add("email",
                      new ActionMessage("error.email"));
        }
        return errors;
    }
}
```

```
    }
}
```

registro.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-logic
        prefix="logic" %>
<%@ taglib uri=http://struts.apache.org/tags-bean
        prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
        prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
  <body>
    <center>
      <h1>Registro de usuarios</h1>
      <html:form action="/registrar" method="POST">
        <table>
          <tr>
            <td>Nombre:</td>
            <td><html:text property="nombre"/></td>
            <td>&nbsp;</td>
          </tr>
          <tr>
            <td>Apellidos:</td>
            <td><html:text property="apellidos"/></td>
            <td>&nbsp;</td>
          </tr>
          <tr>
            <td>Usuario:</td>
            <td><html:text property="usuario"/></td>
            <td><html:errors property="usuario"/></td>
          </tr>
          <tr>
            <td>Password:</td>
            <td><html:password property="password"/></td>
            <td><html:errors property="password"/></td>
          </tr>
        </table>
      </html:form>
    </center>
  </body>
</html>
```


config.xml. Será este mecanismo de control el que analicemos en este apartado.

4.3.1 Gestión declarativa de excepciones

A fin de evitar el uso de incómodos bloques *try-catch* para capturar y gestionar las excepciones dentro del propio código de la aplicación, Struts proporciona un elemento llamado `<exception>` que permite realizar esta captura de las excepciones desde el archivo de configuración de la aplicación.

Hay que recalcar que este tipo de control **sólo puede ser aplicado con aquellas excepciones que se producen en las subclases `Action`** de la aplicación, pero no en el Modelo o en las clases `ActionForm`.

El elemento `<exception>` puede ser incluido en dos lugares diferentes del archivo `struts-config.xml`:

- **Dentro del elemento `<global-exceptions>`.** En este caso la excepción se controla a nivel global, es decir, para todos los `Action` de la aplicación en donde ésta se pueda producir. Para cada tipo de excepción que se quiera capturar se utilizará un elemento `<exception>` diferente:

```
<struts-config>
    <global-exceptions>
        <exception .../>
        <exception.../>
    </global-exceptions>
    :
</struts-config>
```

- **Dentro de un elemento `<action>`.** En este caso la excepción será capturada únicamente si se produce en el interior de la clase `Action` a la que hace referencia el elemento. En el siguiente ejemplo la captura de la excepción sólo se realizaría en la clase `misacciones.EjemploAction`:

```
<action-mappings>
```

```
<action type="misacciones.EjemploAction" ....>
    <exception.../>
</action>
:
</action-mappings>
```

4.3.2 Implementación de la gestión declarativa de excepciones

Cuando dentro de un objeto Action se produce una excepción que ha sido declarada mediante el elemento `<exception>`, bien como una excepción global o bien como una excepción asociada a esa acción, Struts genera un objeto `ActionErrors` con el `ActionMessage` asociado a la excepción y transfiere el control de la aplicación al recurso indicado en el propio elemento `<exception>`.

Todo este proceso se controla a través de los atributos del elemento `<exception>`, cuyos valores determinan los distintos parámetros relativos a la gestión de la excepción. Estos atributos son:

- **type.** Nombre de la clase de excepción que se va a capturar.
- **key.** Nombre de la clave definida en el archivo de recursos `ApplicationResource.properties` asociada al mensaje de error. Esta clave será utilizada para construir el objeto `ActionMessage`.
- **path.** URL relativa a la página JSP donde será trasferido el usuario en caso de que se produzca la excepción. En el interior de esta página podremos hacer uso del elemento `<html:errors/>` para acceder al mensaje de error asociado a la excepción.
- **handler.** En vez de redirigir al usuario a una página de la aplicación podemos optar por realizar un control más exhaustivo de la excepción. En este caso tendremos que crear una clase especial para el tratamiento de la excepción, clase que deberá ser referenciada en este atributo a fin de que Struts pueda crear un objeto de la misma y transferirla el control del programa cuando se produzca la excepción. Más adelante analizaremos con detalle este proceso.

Para comprender el funcionamiento del control declarativo de excepciones vamos a incluir un control de este tipo en la aplicación que desarrollamos en la práctica 3.1 relativa a la información sobre llamadas realizadas.

Lo que haremos será controlar una excepción personalizada que se producirá desde la propia aplicación en caso de que el usuario validado no disponga de teléfonos asociados, redirigiéndole a una página llamada “sintelefonos.jsp” donde se le informará de esta situación.

Lo primero será definir la clase de excepción, a la que llamaremos “SinTelefonosException” y cuyo código será tan simple como el que se indica en el siguiente listado:

```
package excepciones;

public class SinTelefonosException extends Exception{
}
```

Como vemos esta clase hereda toda la funcionalidad que necesitamos de Exception, no siendo necesario sobrescribir o añadir método alguno.

Por otro lado, tendremos que modificar la clase ValidarAction para que en caso de que el usuario no disponga de teléfonos asociados se lance la excepción. He aquí el código de esta clase, indicándose las modificaciones realizadas en fondo sombreado:

```
package servlets;

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

public class ValidarAction extends Action {
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
        throws SinTelefonosException {
        //obtiene los datos del driver y de la cadena de
        //conexión de los parámetros de contexto
        //definidos en Web.xml
        String driver=this.getServlet().
            getServletContext().getInitParameter("driver");
```



```

String cadenaCon=this.getServlet().
    getServletContext().getInitParameter("cadenaCon");
GestionClientes gc=
    new GestionClientes(driver,cadenaCon);
ValidacionForm vf=(ValidacionForm)form;
if(gc.validar(vf)){
    GestionTelefonos gt=
        new GestionTelefonos(driver,cadenaCon);
    //si no dispone de teléfonos
    //se lanza la excepción
    if(gt.getTelefonos(vf.getPassword()).size()==0){
        throw new SinTelefonosException();
    }
    vf.setTelefonos(
        gt.getTelefonos(vf.getPassword()));
    return mapping.findForward("bienvenida");
}
else{
    vf.setMensaje("<h2>Combinación de usuario
        y password incorrecta!</h2>");
    return mapping.findForward("error");
}
}
}

```

Seguidamente, añadiremos en el archivo `ApplicationResource.properties` el mensaje de error que queremos mostrar al usuario con su clave asociada:

error.sintelefonos=El usuario no tiene teléfonos registrados!!

A continuación, crearemos la página JSP a la que redireccionaremos al usuario a la que, como ya hemos indicado, llamaremos “`sintelefonos.jsp`”. El código será muy sencillo, tan sólo mostrará el mensaje de error al usuario:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

```

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title>Página de error</title>
  </head>
  <body>
    <h1>
      <html:errors/>
    </h1>
  </body>
</html>
```

Por último, registraremos el elemento `<exception>` con los datos para la captura de la excepción. Este elemento lo incluiremos dentro del elemento `<action>` asociado a `ValidarAction`:

```
<action input="/" name="ValidacionForm" path="/validar"
  scope="request" type="servlets.ValidarAction">
  <forward name="bienvenida" path="/opciones.jsp"/>
  <forward name="error" path="/login.jsp"/>
  <exception type=
    "excepciones.SinTelefonosException"
    key="error.sintelefonos"
    path="/errores.jsp"/>
</action>
```

Será necesario además incluir en el archivo de configuración la localización de `ApplicationResource`, para ello se utilizará el elemento `<message-resources>`:

```
<message-resources
  parameter="com.myapp.struts.ApplicationResource"/>
```

4.3.3 Clases personalizadas para la gestión de excepciones

Si lo que queremos es realizar una gestión más elaborada de la excepción en vez de una simple redirección a una determinada página de la aplicación, debemos crear una clase especial para el tratamiento de la excepción e incluir en ella todas las instrucciones que queramos ejecutar en caso de que ésta llegue a producirse.

Esta clase deberá heredar **org.apache.struts.action.ExceptionHandler**, la cual proporciona un método *execute()* que será sobrescrito por el programador y en el que incluirá todas las instrucciones necesarias para el tratamiento de la excepción.

El método *execute()* de *ExceptionHandler* es muy similar al *execute()* de *Action*, declarando, además de los cuatro parámetros de éste, otros dos adicionales que proporcionan al método información relativa a la excepción:

```
public ActionForward execute(Exception ex,  
                             ExceptionConfig config,  
                             ActionMapping mapping,  
                             ActionForm form,  
                             HttpServletRequest request,  
                             HttpServletResponse response)  
    throws ServletException
```

Estos dos nuevos parámetros son:

- **ex.** Este parámetro contiene al objeto *Exception* generado en la excepción.
- **config.** Se trata de un objeto *ExceptionConfig* en el que se encapsula toda la información almacenada en el elemento `<exception>` asociado a la excepción.

Una vez implementada la clase para el tratamiento de la excepción, habrá que registrarla en el elemento `<exception>` con el que se va a capturar la excepción dentro de su atributo *handler*. De esta forma, en el momento en que se produzca la

excepción se creará un objeto de la subclase de `ExceptionHandler` y se pasará el control al mismo invocando a su método *execute()*.

Como podemos comprobar, este método también debe devolver un objeto `ActionForward` asociado a la vista a la que tendrá que ser redirigido el usuario después de tratar la excepción.

Supongamos que en el caso del ejemplo anterior, en vez de mandar al usuario a la página “sintelefonos.jsp” al producirse la excepción, quisiéramos por ejemplo almacenar el password del usuario en una variable de sesión y redirigirle después a una nueva página de alta de teléfonos llamada “telefonos.jsp”, cuya dirección virtual asociada en `struts-config.xml` es “alta”.

La clase para la gestión de la excepción que tendríamos que implementar en esta situación podría ser:

```
package excepciones;

public class Gestion extends ExceptionHandler{
    public ActionForward execute(Exception ex,
        ExceptionConfig config,
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException{
        HttpSession s=request.getSession();
        //recuperación del bean que almacena
        //los credenciales del usuario
        ValidacionForm vf=(ValidacionForm)request.
            getAttribute("ValidacionForm");
        //almacenamiento del password en la variable
        //de sesión
        s.setAttribute("pwd", vf.getPassword());
        return mapping.findForward("alta");
    }
}
```

En cuanto al registro de la excepción en el elemento `<action>`, quedaría como se indica a continuación:

```
<action input="/" name="ValidacionForm"
        path="/validar"
        scope="request"
        type="servlets.ValidarAction">

    <forward name="bienvenida"
            path="/opciones.jsp"/>
    <forward name="error"
            path="/login.jsp"/>
    <exception type=
        "excepciones.SinTelefonosException"
        key="error.sintelefonos"
        handler="excepciones.Gestion"/>

</action>
```

Como podemos observar en la definición del elemento `<exception>` anterior, la utilización del atributo *handler* para la especificación de la clase que realizará la gestión de la excepción hace que **no sea necesario utilizar el atributo *path***.

Por otro lado, aunque no se haga uso de la página de error, el mensaje de error asociado al objeto `ActionMessage` generado en la excepción podrá ser utilizado también en la página a la que será redirigido el usuario desde la clase manejadora (la que tiene como dirección virtual asociada “alta”).

LIBRERÍAS DE ACCIONES JSP DE STRUTS

Los tags JSP que Struts pone a disposición del programador a través de las librerías de acciones incluidas en el framework permiten implementar la mayor parte de la funcionalidad de las páginas JSP que forman la vista de una manera rápida y sencilla y sin apenas tener que utilizar scriptlets de código Java.

Durante el Capítulo 4 tuvimos la oportunidad de analizar el funcionamiento de la librería `html` que, entre otras, incluye acciones que facilitan el volcado de datos en un `JavaBean`, suministrados a través de un formulario `XHTML`.

A lo largo de este Capítulo analizaremos las librerías *bean* o *logic*, de manera que al finalizar el mismo seremos capaces de crear páginas JSP totalmente funcionales sin utilizar scriptlets de código Java de servidor. Otra de las librerías de acciones importantes de Struts, *tiles*, será estudiada durante el Capítulo dedicado a la creación de plantillas.

Tanto el código de estas acciones como los archivos de librería `.tld` se encuentran en el archivo `struts-taglib-1.3.9.jar` suministrado con el pack de distribución de Struts y, como ya indicamos en el Capítulo 3, debería estar incluido en el directorio `WEB-INF/lib` de nuestra aplicación.

5.1 LIBRERÍA BEAN

La librería *bean* proporciona acciones para manipular objetos en una aplicación, objetos que pueden ser de diferente naturaleza y ofrecer distinta funcionalidad.

Para poder hacer uso de sus acciones en una página JSP, habrá que incluir mediante la directiva *taglib* la siguiente referencia a la librería *struts-bean.tld*:

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

Analizaremos a continuación las acciones más importantes de esta librería.

5.1.1 write

Esta acción la utilizamos en el primer ejemplo de aplicación Struts que se desarrolló en el Capítulo 3 y tiene como función incluir en la página de respuesta la información almacenada en un determinado objeto. Sus principales atributos son:

- **name.** Nombre o identificador del objeto que contiene la información.
- **scope.** Ámbito en el que se encuentra el objeto. Si no se indica ningún valor para este atributo se intentará localizar el objeto en todos los ámbitos siguiendo el orden: *page*, *request*, *session* y *application*.
- **property.** Propiedad del objeto cuyo valor se quiere incluir en la página. Si no se utiliza este atributo se recuperará el resultado de invocar al método *toString()* sobre el objeto.
- **format.** Cadena de caracteres que define el formato de presentación que se va a aplicar sobre el dato. Por ejemplo, la cadena “0.00” indica que el valor debe mostrarse como una cantidad numérica con dos cifras decimales. Si no se especifica este atributo se utilizará el formato por defecto para el tipo de dato.
- **filter.** Se trata de un atributo de tipo boolean. Si su valor es *true* se aplicará un filtro para los caracteres especiales incluidos en el dato, a fin de que no sean interpretados como texto XHTML por el navegador. Por ejemplo, si dentro de la cadena de texto de presentación se incluye el carácter “<”, éste será sustituido por su correspondiente referencia <.

Mediante la siguiente acción de ejemplo se mostraría en la página de respuesta, dentro de un elemento <h1>, el contenido de la propiedad “titulo” de un bean de sesión llamado “libro”:


```
<h1><bean:write name="libro" property="titulo"
scope="session" /></h1>
```

La acción anterior equivale al siguiente scriptlet de Java:

```
<%Libro libro;
libro = (Libro)session.getAttribute("libro");%>
<h1><%=libro.getTitulo()%></h1>
```

5.1.2 parameter

Recupera el valor de un parámetro enviado en la petición y lo almacena en una variable JSP. Esta variable tendrá ámbito de página y podrá ser referenciada desde otras acciones de la misma página JSP.

Entre sus atributos destacamos:

- **id.** Nombre o identificador asignado a la variable.
- **name.** Nombre del parámetro cuyo valor se quiere almacenar en la variable.
- **value.** En caso de que no se haya enviado en la petición ningún parámetro con el nombre indicado en *name*, se asignará a la variable el valor definido en este atributo.

El siguiente ejemplo recupera el valor del parámetro “password” y lo muestra más adelante en la misma página:

```
<bean:parameter id="pwd" name="password" />
:
<p>El password enviado vale:
<b><bean:write name="pwd" /></b>
</p>
```

Como se puede observar en este ejemplo, dado que lo que contiene una variable JSP es una referencia a un objeto, podemos utilizar también la acción *write* para acceder a este tipo de variables.

5.1.3 cookie

Recupera el valor de una cookie enviada en la petición y lo almacena en una variable JSP.

Los principales atributos de la acción son:

- **id.** Nombre o identificador asignado a la variable.
- **name.** Nombre de la cookie cuyo valor se quiere almacenar en la variable.
- **value.** En caso de que no se haya enviado en la petición ninguna cookie con el nombre indicado en *name*, se asignará a la variable el valor indicado en este atributo.

El siguiente ejemplo muestra en la página el contenido de la cookie “usuario”, mostrándose el valor “sin datos” en caso de que ésta no exista:

```
<bean:cookie id="user" name="usuario"/>
:
<p>Le doy la bienvenida a mi página sr./a:
<b><bean:write name="user" /></b>
</p>
```

5.1.4 header

Recupera el valor de un encabezado enviado en la petición y lo almacena en una variable JSP.

Al igual que en las dos acciones anteriores los principales atributos de header son:

- **id.** Nombre o identificador asignado a la variable.
- **name.** Nombre del encabezado cuyo valor se quiere almacenar en la variable.
- **value.** En caso de que no se haya enviado en la petición ningún encabezado con el nombre indicado en *name*, se asignará a la variable el valor definido en este atributo.

5.1.5 message

Permite mostrar en la página un mensaje de texto almacenado en un archivo externo, concretamente en `ApplicationResource.properties`. Este tag es muy utilizado en internacionalización de aplicaciones, permitiendo que los textos

específicos de cada idioma puedan residir en archivos de recursos externos en vez de ser introducidos directamente en la página.

Los principales atributos de esta acción son:

- **key**. Clave asociada en `ApplicationResource.properties` a la cadena de caracteres. Struts localizará en el archivo de recursos el texto que tenga la clave especificada en este atributo y lo mostrará en la página.
- **name**. En vez de suministrar directamente en *key* el valor de la clave se puede extraer ésta de un objeto, en cuyo caso el nombre o identificador del objeto se indicará en este atributo.
- **property**. Nombre de la propiedad del objeto indicado en *name* que contiene la clave del mensaje.
- **scope**. Ámbito del objeto especificado en *name*.

Suponiendo que tenemos registrada la siguiente entrada en el archivo de recursos:

struts.message=Struts is easy

el bloque de acciones que se muestra a continuación almacena el nombre de la clave anterior en la propiedad “clave” de un objeto de la clase `javabeans.Datos`, mostrando a continuación el mensaje asociado a dicha clave:

```
<jsp:useBean id="info" class="javabeans.Datos">
  <jsp:setProperty name="info" property="clave"
    value="welcome.message"/>
</jsp:useBean>
<b>Mensaje</b>:
<h2>
  <bean:message name="info" property="clave"/>
</h2>
```

5.1.6 define

Esta acción almacena en una variable JSP una referencia a un objeto existente en la aplicación o, en su defecto, un nuevo objeto `String` creado por la propia acción a partir de una determinada cadena de texto.

Entre los atributos más significativos proporcionados por la acción tenemos:

- **id.** Nombre o identificador asignado a la variable JSP.
- **name.** Nombre del objeto existente en cualquiera de los ámbitos de aplicación cuya referencia será almacenada en la variable.
- **property.** Si se especifica este atributo en la acción, en vez de asignar a la variable JSP la referencia al objeto indicado en *name* se almacenará en ésta el valor de la propiedad del objeto cuyo nombre se especifica en este atributo. Por ejemplo, si el valor del atributo *name* es “mipersona” y el de *property* es “apellido”, en la variable JSP indicada en *id* se almacenará una referencia al objeto devuelto (String en este caso) por la llamada a *getApellido()* sobre el bean “mipersona”.
- **scope.** Ámbito donde se debe localizar al objeto especificado en *name*. Si no se emplea este atributo el objeto será buscado en todos los ámbitos de la aplicación siguiendo el orden: *page*, *request*, *session* y *application*.
- **toScope.** Indica el ámbito en el que se almacenará la referencia asignada a la variable JSP. Si no se especifica este atributo la variable JSP tendrá ámbito de página.

Para entender mejor el funcionamiento de la acción *define* vamos a pensar en el siguiente ejemplo: supongamos que tenemos una clase de tipo JavaBean llamada “Datos” que encapsula una cadena de caracteres dentro de la propiedad “valor”. Desde una página *inicio.jsp* creamos una instancia de la misma y redireccionamos al usuario a otra página llamada *pagina1.jsp*, tal y como se indica a continuación:

```
<jsp:useBean id="obj" class="javabeans.Datos"
            scope="session"/>
<jsp:setProperty name="obj" property="valor"
value="cadena de prueba Datos"/>
<%response.sendRedirect("pagina1.jsp");%>
```

Por su parte, *pagina1.jsp* utilizará la acción *define* para almacenar en una variable JSP el objeto *obj* con ámbito *request* para después pasar la petición a *pagina2.jsp*:

```
<bean:define id="mivar" name="obj"
    scope="session" toScope="request"/>
<jsp:forward page="pagina2.jsp"/>
```

En cuanto a `pagina2.jsp`, utilizará el elemento `write` para mostrar en la página de respuesta el contenido de la propiedad `valor` del objeto, utilizando la variable JSP creada en `pagina1.jsp`:

```
<bean:write name="mivar" scope="request" property="valor"/>
```

De esta manera, al solicitar la página `inicio.jsp` desde su navegador el usuario recibirá una página de respuesta en la que le aparecerá el mensaje:

“cadena de prueba Datos”

5.1.7 page

Almacena en una variable JSP a uno de los objetos implícitos de JSP a fin de que puedan ser referenciados posteriormente desde alguna otra acción de la librería `bean`, como por ejemplo `write`.

Los atributos disponibles para esta acción son:

- **id.** Nombre de la variable JSP en la que se almacenará la referencia al objeto.
- **property.** Nombre del objeto implícito JSP (*page*, *request*, *response*, *session*, *application* o *config*).

5.1.8 size

Almacena en una variable JSP el número de elementos de un array o colección. Sus atributos son:

- **id.** Nombre o identificador de la variable JSP.
- **name.** Nombre del objeto en el que se encuentra la colección.
- **property.** Propiedad del objeto especificado en `name` que contiene la colección cuyo tamaño se almacenará en la variable. Si no se indica ningún valor para este atributo se considerará al propio objeto indicado en `name` como el objeto de colección.

- **scope.** Ámbito en el que se encuentra el objeto indicado en name. Si no se utiliza este atributo se buscará el objeto en todos los ámbitos de la aplicación, siguiendo el mismo criterio indicado en acciones anteriores.
- **collection.** Como alternativa a los atributos anteriores se puede indicar directamente en este atributo, mediante una expresión JSP, el objeto de colección o array cuyo tamaño debe ser almacenado en la variable JSP.

El siguiente ejemplo nos muestra el número de autores de un libro. El objeto “libro” se encuentra almacenado en una variable de sesión y una de sus propiedades, autores, contiene la colección con los nombres de los autores del mismo:

```
<bean:size id="totalautores" name="libro"
          scope="session" property="autores"/>
El número de autores del libro es:
<bean:write name="totalautores"/>
```

5.2 LIBRERÍA LOGIC

En la librería logic de Struts encontramos un amplio abanico de acciones con las que podemos realizar cualquier operación de control de flujo de ejecución en la página, como la generación de resultados diferentes en función de una condición, la iteración sobre colecciones de objetos, etc. Todo ello sin necesidad de escribir una sola línea de código Java en la página.

Las acciones logic se encuentran registradas en el archivo struts-logic.tld incluido en la librería struts-core-1.3.9.jar. Para poder utilizar estas acciones debemos incluir la siguiente referencia *taglib* en la página JSP:

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

Veamos a continuación las acciones más importantes que se encuentran en esta librería.

5.2.1 equal

Evalúa el contenido anidado de la acción si el dato especificado en uno de sus atributos es igual a la constante indicada en su atributo *value*. La comparación puede ser numérica o String, según la naturaleza de los valores a comparar.

El formato de utilización de esta acción sería pues:

```
<logic:equal ....>
```

```
<!--contenido evaluado si se produce la igualdad-->
```

```
</logic:equal>
```

Como ya se ha indicado, el atributo *value* debe contener el valor con el que se comparará el dato u objeto indicado en los restantes atributos de la acción, valor que será suministrado como un literal numérico o de texto. **El uso de este atributo es obligatorio.**

En cuanto al resto de los atributos de la acción éstos son:

- **cookie.** Nombre de la cookie enviada en la petición HTTP cuyo valor será comparado con *value*.
- **parameter.** Nombre del parámetro enviado en la petición HTTP cuyo valor será comparado con *value*.
- **header.** Nombre del encabezado de la petición HTTP cuyo valor será comparado con *value*.
- **name.** Nombre del objeto, existente en alguno de los ámbitos de la aplicación, con el que será comparado *value*.
- **property.** El valor de este atributo representa el nombre de la propiedad del bean indicado en *name*, cuyo valor será comparado con *value*.
- **scope.** Ámbito del objeto especificado en *name*. Si no se especifica este atributo el objeto será buscado en todos los ámbitos, siguiendo el orden: *page*, *request*, *session* y *application*.

Veamos el siguiente ejemplo:

Supongamos que tenemos una página `inicio.jsp` desde la que redireccionamos al usuario a otra página llamada `pagina1.jsp`:

```
<%response.sendRedirect("pagina1.jsp?par=35");%>
```

Por otro lado, en `pagina1.jsp` tenemos el siguiente código:

```
<body>
    <logic:equal parameter="par" value="35">
        <h1>La condición se cumple!</h1>
    </logic:equal>
    Página de resultados
</body>
```

Si el usuario lanza una solicitud de la página `inicio.jsp` le aparecerá en su navegador una página como la que se indica en la figura 28.

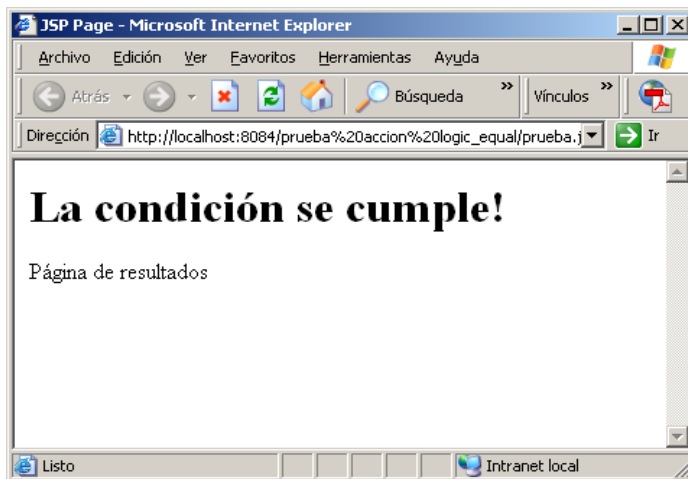


Fig. 28. Aspecto de la página de respuesta

Los atributos *cookie*, *parameter*, *header* y *name* no son excluyentes entre sí, pudiéndose utilizar más de uno de ellos en una misma acción. En este caso, el contenido anidado de `<logic:equal>` **sólo será evaluado si la constante especificada en `value` coincide con los datos indicados en todos y cada uno de los atributos anteriores.**

Volviendo al ejemplo anterior, supongamos que tenemos además una clase de tipo JavaBean llamada `Datos` con una propiedad *numero* que almacena valores

numéricos de tipo entero, incluyéndose ahora en la página inicio.jsp el contenido que se indica a continuación:

```
<jsp:useBean id="obj" class="javabeans.Datos"
            scope="session" />
<jsp:setProperty name="obj" property="numero" value="10" />
<%response.sendRedirect("pagina1.jsp?par=35");%>
```

En cuanto al contenido de pagina1.jsp, incluimos ahora en ella el siguiente código:

```
<body>
    <logic:equal name="obj" scope="session"
        property="numero" parameter="par" value="35">
        <h1>La condición se cumple!</h1>
    </logic:equal>
    Página de resultados
</body>
```

En este caso se comparará el número 35 con el parámetro “par” y con el valor de la propiedad “numero” del objeto de la clase Datos instanciado en la página anterior; como la condición de igualdad sólo se cumple con el parámetro “par” (el valor de la propiedad “numero” del objeto es igual a 10) el cuerpo de la acción no será evaluado, mostrándose al usuario la página indicada en la figura 29 al lanzar la solicitud de inicio.jsp.

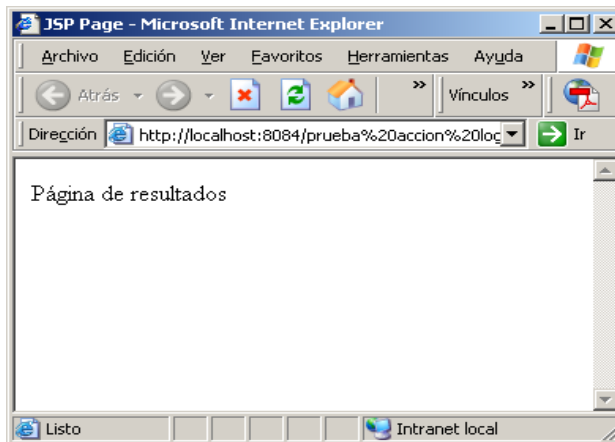


Fig. 29. *Página de respuesta recibida al solicitar inicio.jsp*

5.2.2 notEqual

Evalúa el contenido anidado de la acción si el dato especificado en uno de sus atributos no es igual al valor de la constante indicada en *value*. Dispone exactamente de los mismos atributos que *equal* con idéntico significado.

Al igual que sucede con *equal*, en una misma acción *notEqual* pueden aparecer los atributos *cookie*, *parameter*, *header* y *name*, aunque en este caso **bastará con que el dato especificado en uno de ellos sea diferente a la constante indicada en value para el cuerpo de la acción sea evaluado.**

5.2.3 greaterEqual, lessEqual, greaterThan y lessThan

Se trata de cuatro acciones que tienen la misma estructura y atributos que las dos anteriores, diferenciándose de éstas y entre ellas en la condición para que el contenido anidado de la acción sea evaluado.

El criterio será que el dato indicado en los atributos *cookie*, *header*, *parameter* y *name* sea mayor o igual que la constante indicada en *value* para la acción *greaterEqual*, menor o igual para *lessEqual*, mayor para *greaterThan* y menor para *lessThan*.

Si en una misma acción de cualquiera de estas cuatro se emplea más de un atributo selector de dato (*cookie*, *parameter*, *header* y *name*), **la condición se aplicará sobre cada uno de los datos indicados, evaluándose el cuerpo de la acción si la condición se cumple en todos ellos.**

5.2.4 match

Evalúa el contenido anidado de la acción si el dato especificado en sus atributos contiene a la constante indicada en *value*.

Además de disponer de los mismos atributos analizados en las acciones anteriores, *match* incluye el atributo *location*, el cual puede tomar los valores “start” o “end”. Si se indica el valor “start”, la constante especificada en *value* deberá aparecer al principio del dato para que el contenido de la acción sea evaluado, mientras que si el valor de *location* es “end” la constante deberá aparecer al final del dato. En caso de que no se emplee este atributo, la constante será buscada en cualquier parte del dato.

Si tenemos una página *inicio.jsp* con la siguiente instrucción:

```
<%response.sendRedirect(
    "prueba.jsp?par=Prueba de atributo match en Struts"); %>
```

y en la página `prueba.jsp` incluimos el siguiente código:

```
<body>
    <logic:match parameter="par" location="end"
                value="match">
        <h1>Valor encontrado en el parámetro</h1>
    </logic:match>
    Página de resultados
</body>
```

el cuerpo de la acción `match` no será evaluado, ya que, aunque el valor “match” se encuentra incluido en la cadena “Prueba de atributo match en Struts” enviada como parámetro, no está situado al final de la misma tal y como se especifica en *location*.

Si en una misma acción se emplea más de un atributo selector de dato (*cookie*, *parameter*, *header* y *name*) **la búsqueda de la constante se llevará a cabo sobre cada uno de los datos indicados, evaluándose el cuerpo de la acción si la coincidencia se da en alguno de ellos.**

5.2.5 noMatch

Dispone de los mismos atributos que *match* aunque, a diferencia de ésta, evaluará su contenido si el dato **no contiene** a la constante especificada en *value*. En caso de emplear más de un atributo selector de dato, **el contenido será evaluado siempre y cuando la constante no esté contenida en ninguno de los datos.**

5.2.6 forward

Esta acción transfiere la petición del usuario a la URL asociada a uno de los elementos `<forward>` globales definidos en el archivo de configuración `struts-config.xml`, indicando a través de su único atributo *name* el nombre del elemento `<forward>` utilizado.

5.2.7 redirect

Redirecciona al usuario a una determinada URL, pudiendo enviar diferentes parámetros en la misma. Esta URL puede ser especificada de distintas

formas según los atributos utilizados por la acción, más concretamente, en uno de los cuatro atributos que se indican a continuación:

- **action.** Nombre del elemento <action> global que representa el destino a donde será redirigido el usuario.
- **forward.** Nombre del elemento <forward> global a cuya URL será redireccionado el usuario.
- **href.** Especifica la URL absoluta del destino.
- **page.** URL relativa del destino.

En una determinada acción redirect **sólo podrá definirse uno** de los atributos anteriores.

Por otro lado, en todos los casos se aplicará automáticamente la reescritura en URL para mantener el identificador de sesión en caso de que se desactive el uso de cookies por el usuario.

Para enviar parámetros en la URL podrán emplearse los siguientes atributos de la acción:

- **paramId.** Nombre del parámetro que se quiere enviar.
- **paramName.** Nombre del objeto que será utilizado como valor del parámetro especificado en *paramId*.
- **paramProperty.** Propiedad del objeto indicado en *paramName* cuyo valor será enviado como parámetro, en vez del propio objeto.
- **paramScope.** Ámbito del objeto especificado en *paramName*. Si no se utiliza este atributo se intentará localizar el objeto en todos los ámbitos de la aplicación, siguiendo el orden: *page*, *request*, *session* y *application*.
- **name.** Como alternativa a *paramId* y *paramName* puede especificarse en el atributo *name* un objeto colección de tipo `java.util.Map` cuyo contenido será enviado en parámetros de la URL, siendo la clave de cada elemento el nombre del parámetro y el propio elemento el valor del mismo.

- **property.** Si se utiliza este atributo, la colección de elementos que será enviada en los parámetros de la URL será obtenida a partir de la propiedad del objeto especificado en *name*, cuyo nombre se indica en este atributo.

Veamos un ejemplo de utilización de esta acción: supongamos como en el ejemplo anterior que tenemos una clase JavaBean llamada Datos con una propiedad “numero” que almacena valores numéricos enteros. La siguiente página redireccionaría al usuario a la URL asociada al elemento <forward> “prueba”, pasando como parámetro en la URL el valor de la propiedad “numero” del objeto Datos instanciado en la propia página:

```
<body bgcolor="white">
<jsp:useBean id="obj" class="javabeans.Datos"
              scope="request" />
<jsp:setProperty name="obj" property="numero" value="34" />
<logic:redirect forward="prueba"
                paramId="edad"
                paramName="obj"
                paramProperty="numero" />
</body>
```

La página destino por su parte utilizará las siguientes acciones para mostrar el contenido del parámetro “edad” en la página, visualizándose: “Edad: 34”:

```
<body>
    <bean:parameter id="var" name="edad" />
    Edad: <bean:write name="var" />
</body>
```

5.2.8 iterate

Esta acción recorre todos los elementos contenidos en una colección, evaluando para cada uno de ellos el contenido anidado de la acción.

La colección sobre la que se va a iterar debe ser de uno de los siguientes tipos:

- Un array de tipos primitivos u objetos Java.
- Un objeto java.util.Collection.

- Un objeto `java.util.Enumeration`.
- Un objeto `java.util.Iterator`.
- Un objeto `java.util.Map`.

Esta colección puede ser especificada de dos posibles maneras, según cuál de los dos siguientes atributos de `<logic:iterate>` se emplee:

- **collection.** Representa una expresión JSP que devuelve la colección a recorrer.
- **name.** Nombre del objeto que contiene la colección.

En el caso de que se utilice el atributo *name*, podrán especificarse también los atributos:

- **property.** Nombre de la propiedad del objeto indicado en *name* que contiene la colección.
- **scope.** Ámbito del objeto indicado en *name*.

Aparte de los atributos anteriores, la acción *iterate* dispone de estos otros:

- **id.** Nombre o identificador de la variable JSP que almacenará la referencia al elemento de la colección en cada iteración. Su ámbito está limitado a la página.
- **indexId.** Nombre o identificador de la variable JSP en donde se almacenará el índice de la iteración en curso. Su ámbito está limitado a la página.
- **type.** Nombre cualificado de la clase a la que pertenecen los elementos de la colección.
- **length.** Número de iteraciones que se realizarán sobre la colección. Si no se especifica se recorrerá completamente la colección.

En el siguiente ejemplo se muestra en una página la lista de nombres almacenada en una colección de ámbito de sesión llamada “listado”, para, a continuación, indicar el número de nombres presentados:

```
<logic:iterate name="listado" scope="session" id="nombre"
```

```
                                indexId="total">
        <bean:write name="nombre"/><br/>
</logic:iterate>
Total de nombres: <bean:write name="total"/>
```

PRÁCTICA 5.1. LISTADO DE LLAMADAS CON TAGS STRUTS

Descripción

Se trata de realizar la misma aplicación desarrollada en la práctica 4.1, eliminando los scriptlets de código Java de las páginas JSP y empleando únicamente etiquetas XHTML y tags de Struts en su construcción.

Desarrollo

Los únicos componentes que tendremos que modificar respecto a la versión anterior serán las vistas listado.jsp y opciones.jsp, puesto que login.jsp y registro.jsp no incluían ningún tipo de scriptlet.

Listados

Seguidamente mostraremos el nuevo contenido de las páginas JSP mencionadas anteriormente.

opciones.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@ taglib uri=http://struts.apache.org/tags-logic
        prefix="logic" %>
<%@ taglib uri=http://struts.apache.org/tags-bean
        prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
        prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
        Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
```

```

<title>JSP Page</title>
</head>
<body>
  <center>
    <h1>Opciones para el listado de llamadas </h1>
    <html:form action="/listado" method="POST">
      <b> Seleccione número de teléfono:</b>
      <br/> <br/>
      <html:select property="numero">
        <!--Recupera el bean ValidacionForm
             almacenado en una variable de petición-->
        <logic:iterate id="telefono"
          name="ValidacionForm" property="telefonos"
          type="java.lang.Integer" scope="request">
          <option value=
            "<bean:write name='telefono'/>">
            <bean:write name="telefono"/>
          </option>
        </logic:iterate>
      </html:select>
      <br/> <br/> <br/>
      <b>Seleccione tipo de llamadas:</b><br/>
      <table>
        <tr>
          <td><html:radio tabindex="0"
            property="operacion" value="todas"/></td>
          <td align="left">Todas</td>
        </tr>
        <tr>
          <td><html:radio property="operacion"
            value="portipo"/></td>
          <td align="left">Seleccione Tipo:
            <html:select property="tipo">
              <html:option value="1">Normal
            </html:option>
              <html:option value="2">Reducida
            </html:option>
              <html:option value="3">Super reducida
            </html:option>
            </html:select>
          </td>
        </tr>
      </table>
    </center>
  </body>

```



```

        </tr>
        <tr>
            <td><html:radio property="operacion"
                value="porfecha" /></td>
            <td align="left">A partir de fecha:
                <html:text property="fecha" /></td>
        </tr>
        <tr>
            <td colspan="2"><br />
                <html:submit value="Mostrar listado" /></td>
        </tr>
    </html:form>
</center>
</body>
</html:html>

```

listado.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="java.util.*,javabeans.*" %>
<%@ taglib uri=http://jakarta.apache.org/struts/tags-bean
    prefix="bean" %>
<%@ taglib uri=http://jakarta.apache.org/struts/tags-logic
    prefix="logic" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <h1>Listado de Llamadas</h1>
            <table border="1" width="60%">
                <th>Teléfono destino</th>
                <th>Duración (segundos)</th>
                <th>Tarifa</th>
                <th>Coste</th>
            </table>

```

```
<th>Fecha</th>
<logic:iterate id="llamada" name="llamadas"
    scope="request" type="javabeans.LlamadaBean">
    <tr>
        <td><bean:write name="llamada"
            property="destino"/></td>
        <td><bean:write name="llamada"
            property="duracion"/></td>
        <td><bean:write name="llamada"
            property="tarifa"/></td>
        <td><bean:write name="llamada"
            property="coste" format="0.00"/></td>
        <td><bean:write name="llamada"
            property="fecha"/></td>
    </tr>
</logic:iterate>
</table>
</center>
</body>
</html>
```

PRÁCTICA 5.2. ENVÍO Y VISUALIZACIÓN DE MENSAJES

Descripción

Se trata en esta práctica de implementar la versión Struts de la aplicación para envío y recepción de mensajes que desarrollamos durante la práctica 1.1. El aspecto de las páginas será el mismo que el de esta aplicación.

Desarrollo

Además de los componentes Struts del Controlador, en esta versión utilizaremos los tags de las librerías *bean* y *logic* a fin de no tener que incluir ningún scriptlet de código Java en las páginas.

Las operaciones para la gestión de las acciones grabar mensajes y recuperar mensajes serán implementadas por dos subclases Action llamadas EnviarAction y RecuperarAction, respectivamente.

Listado

Seguidamente presentaremos el listado de los diferentes componentes de la aplicación, incluido el archivo de configuración struts-config.xml.

struts-config.xml

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

<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">
<struts-config>
    <form-beans>
        <form-bean name="MensajeForm"
            type="javabeans.MensajeForm"/>
    </form-beans>
    <global-forwards>
        <forward name="previoenvio" path="/envio.jsp"/>
        <forward name="previolectura" path="/mostrar.htm"/>
        <forward name="inicio" path="/inicio.jsp"/>
    </global-forwards>
    <action-mappings>
        <action name="MensajeForm" path="/grabar"
            scope="request" type="servlets.EnviaAction" >
            <forward name="grabado" path="/inicio.jsp"/>
        </action>
        <action path="/mostrar"
            type="servlets.RecuperarAction">
            <forward name="visualiza" path="/ver.jsp"/>
            <forward name="sinmensajes"
                path="/nomensajes.jsp"/>
        </action>
    </action-mappings>
</struts-config>
```

MensajeForm.java

```
package javabeans;

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

public class MensajeForm extends ActionForm {
```

```
private String remite;
private String destino;
private String texto;
public MensajeForm(){
    //constructor que permite crear un objeto
    //Mensaje a partir de los datos del mismo
    public MensajeForm(String remite, String destino,
        String texto){
        this.remite=remite;
        this.destino=destino;
        this.texto=texto;
    }
    public void setRemite(String remite){
        this.remite=remite;
    }
    public String getRemite(){
        return this.remite;
    }
    public void setDestino(String destino){
        this.destino=destino;
    }
    public String getDestino(){
        return this.destino;
    }
    public void setTexto(String texto){
        this.texto=texto;
    }
    public String getTexto(){
        return this.texto;
    }
}
```

EnviarAction.java

```
package servlets;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import javaxBeans.*;
import modelo.*;
public class EnviarAction extends Action {
    public ActionForward execute(ActionMapping mapping,
```

```
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
    MensajeForm men=(MensajeForm)form;
    Operaciones oper=new Operaciones();
    oper.grabaMensaje(men);
    return mapping.findForward("grabado");
}
}
```

RecuperarAction.java

```
package servlets;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import java.util.*;
import javax beans.*;
import modelo.*;

public class RecuperarAction extends Action {
    public ActionForward execute(ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {
        Operaciones oper=new Operaciones();
        ArrayList mensajes=oper.
            obtenerMensajes(request.getParameter("nombre"));
        //si hay mensajes se visualizan
        if(mensajes!=null&&mensajes.size(>0){
            request.setAttribute("mensajes",mensajes);
            return mapping.findForward("visualiza");
        }
        else{
            return mapping.findForward("sinmensajes");
        }
    }
}
```

Operaciones.java

```
package modelo;
import java.sql.*;
import javax.*;
import java.util.*;
public class Operaciones {
    //método común para la obtención
    //de conexiones
    public Connection getConnection(){
        Connection cn=null;
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            cn=DriverManager.getConnection("jdbc:odbc:mensajes");
        }
        catch(Exception e){e.printStackTrace();}
        return cn;
    }
    public ArrayList obtenerMensajes(String destino){
        Connection cn=null;
        ArrayList mensajes=null;
        Statement st;
        ResultSet rs;
        try{
            cn=getConnection();
            st=cn.createStatement();
            String tsq;
            tsq="select * from mensajes where
                destinatario='"+destino+"'";
            rs=st.executeQuery(tsq);
            mensajes=new ArrayList();
            //para cada mensaje encontrado crea un objeto
            //Mensaje y lo añade a la colección ArrayList
            while(rs.next()){
                MensajeForm m=new MensajeForm(
                    rs.getString("remitente"),
                    rs.getString("destinatario"),
                    rs.getString("texto"));
                mensajes.add(m);
            }
            cn.close();
        }
```

```

    }
    catch(Exception e){e.printStackTrace();}
    return(mensajes);
}
public void grabaMensaje(MensajeForm m){
    Connection cn;
    Statement st;
    ResultSet rs;
    try{
        cn=getConnection();
        st=cn.createStatement();
        String tsq;
        //a partir de los datos del mensaje construye
        //la cadena SQL para realizar su inserción
        tsq="Insert into mensajes values('";
        tsq+=m.getDestino()+
            "' , '" +m.getRemite()+
            "' , '" +m.getTexto()+"' )";
        st.execute(tsq);
        cn.close();
    }
    catch(Exception e){e.printStackTrace();}
}
}

```

inicio.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
<body>
<center>
    <br/><br/>

    <html:link forward="previoenvio">
        Enviar mensaje</html:link>
    <br/><br/>

```

```

        <html:link forward="previolectura">
            Leer mensajes</html:link>
    </center>
</body>
</html:html>

```

envio.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-bean
    prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
    <body>
        <center>
            <h1>Generación de mensajes</h1>
            <html:form action="/grabar" method="POST">
                <br/><br/>
                <b>Datos del mensaje:</b><br/><br/>
                Introduzca destinatario:
                    <html:text property="destino"/><br/>
                <br/>
                Introduzca remitente :
                    <html:text property="remite"/><br/>
                <br/>
                Introduzca texto : <br/>
                <html:textarea property="texto">
                </html:textarea>
                <hr/><br/>
                <html:submit value="Enviar"/>
            </html:form>
        </center>
    </body>
</html:html>

```


mostrar.htm

```
<html>
<body>
<center>
    <br/><br/>
    <form action="mostrar.do" method="post">
        Introduzca su nombre:<input
            type="text" name="nombre"><br><br>
        <input type="submit" value="Mostrar mensajes">
    </form>
</center>
</body>
</html>
```

ver.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javabeans.*,java.util.*"%>
<%@ taglib uri=http://struts.apache.org/tags-logic
    prefix="logic" %>
<%@ taglib uri=http://struts.apache.org/tags-bean
    prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
<body>
<center>
<bean:parameter id="nombre" name="nombre"/>
<h1>
Mensajes para <bean:write name="nombre"/>
</h1>
<table border="1">
    <tr><th>Remitente</th><th>Mensaje</th></tr>
    <logic:iterate id="mensaje" name="mensajes"
        scope="request">
        <tr><td><bean:write name="mensaje"
```

```

        property="remite" /></td>
        <td><bean:write name="mensaje"
            property="texto" /></td>
    </tr>
</logic:iterate>
</table>
<br/><br/>
<html:link forward="inicio">Inicio</html:link>
</center>
</body>
</html:html>

```

nomensajes.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javabeans.*, java.util.*"%>
<%@ taglib uri=http://struts.apache.org/tags-logic
    prefix="logic" %>
<%@ taglib uri=http://struts.apache.org/tags-bean
    prefix="bean" %>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
<bean:parameter id="nombre" name="nombre"/>
<body>
    <center>
        <h2>
            Lo siento, <bean:write name="nombre"/> no tiene mensajes
        </h2>
        <br/><br/><br/><br/>
        <html:link forward="inicio">Inicio</html:link>
    </center>
</body>
</html:html>

```

VALIDACIÓN DE DATOS DE USUARIO

La utilización del método *validate()* de *ActionForm* permite, como hemos visto en Capítulos anteriores, validar los datos suministrados por el usuario a través de un formulario XHTML antes de que éstos sean procesados por la aplicación.

Pero este sistema de validación requiere codificar de forma manual las instrucciones para la comprobación de datos, lo que además de tedioso resulta ineficiente, ya que la definición de criterios de validación idénticos en varios *ActionForm* implica la repetición de las mismas instrucciones de verificación de datos en los métodos *validate()* de cada una de las clases.

La utilización de validadores simplifica enormemente el proceso de validación de datos de usuario en una aplicación Struts al proporcionar un mecanismo de validación automática, de tipo declarativo, que evita la codificación de estas tareas desde código, permitiendo además la validación de los datos tanto en el cliente como en el servidor.

6.1 COMPONENTES DE UN VALIDADOR

La utilización de validadores en una página JSP se apoya en el empleo de una serie de componentes que proporcionan toda la funcionalidad necesaria para llevar a cabo la validación automática y que habrá que configurar adecuadamente en cada aplicación, éstos componentes son:

- Plug-in validator.
- Archivos de configuración.
- La clase ValidatorForm.
- Archivo de recursos ApplicationResource.properties.

Veamos a continuación el significado y función de estos componentes antes de entrar en detalle sobre la manera en que se deben utilizar los validadores.

6.1.1 Plug-in validator

El plug-in validator es la clase incorporada en el API de Struts encargada de gestionar las validaciones automáticas dentro de una aplicación Web.

Para que puedan utilizarse las validaciones automáticas en una determinada aplicación será necesario registrar esta clase en el archivo `struts-config.xml` utilizando el elemento `<plug-in>`:

```
<plug-in className=
    "org.apache.struts.validator.ValidatorPlugIn">
    <set-property
        property="pathnames"
        value="/WEB-INF/validator-rules.xml,
            /WEB-INF/validation.xml"/>
</plug-in>
```

Este elemento dispone de un atributo *className* en el que se debe especificar el nombre de la clase Struts.

Se debe indicar además mediante el subelemento `<set-property>` las propiedades necesarias para el funcionamiento del plug-in, si bien en la mayoría de los casos únicamente será necesario especificar la propiedad *pathnames*, la cual debe contener la URL relativa de los archivos de configuración utilizados por el plug-in.

6.1.2 Archivos de configuración

Además del archivo de configuración de Struts, `struts-config.xml`, las aplicaciones que utilizan validadores necesitan dos archivos de configuración adicionales: **validator-rules.xml** y **validation.xml**.

6.1.2.1 VALIDATOR-RULES.XML

A fin de que el programador no tenga que codificar las instrucciones para la validación de los datos, Struts incorpora una serie de clases predefinidas cuyos métodos realizan las tareas habituales de validación de datos de usuario requeridas por la mayoría de las aplicaciones Web.

El archivo `validator-rules.xml`, incluido en el paquete de distribución de Struts, contiene la declaración de estas clases de validación así como los métodos utilizados para realizar cada una de las rutinas de validación predefinidas, asociando a cada uno de estos métodos un nombre lógico que luego podrá ser utilizado para asignar una determinada regla de validación a un componente gráfico.

El siguiente listado muestra la estructura del archivo `validator-rules.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.1.3//EN"

"http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
  <global>
    <validator name="required"
      classname="org.apache.struts.
        validator.FieldChecks"
      method="validateRequired"
      methodParams=" java.lang.Object,
        org.apache.commons.
          validator.ValidatorAction,
          org.apache.commons.validator.Field,
          org.apache.struts.
            action.ActionMessages,
          org.apache.commons.
            validator.Validator,
          javax.servlet.http.HttpServletRequest"
      msg="errors.required"/>

    :

  </global>
</form-validation>
```

Cada una de estas reglas de validación se declaran mediante un elemento <validator>, definiéndose a través de sus atributos los parámetros de configuración de cada regla. Estos atributos son:

- **name.** Nombre lógico asignado a la regla de validación.
- **classname.** Nombre de la clase de Struts en la que se encuentran definidas las instrucciones para realizar la tarea de validación.
- **method.** Método de la clase indicada en **classname** que implementa la regla de validación.
- **methodParams.** Parámetros declarados por el método y que necesita para llevar a cabo su función. Será el plug-in validator el encargado de invocar al método pasándole los argumentos apropiados en la llamada.
- **msg.** Clave asociada al mensaje de error que se mostrará al usuario cuando se incumpla el criterio de validación definido en la regla. Estos mensajes se registrarán en el archivo `ApplicationResource.properties`.

Además de las reglas de validación predefinidas por Struts, es posible crear nuestras propias reglas de validación personalizadas a fin de poderlas reutilizar en las aplicaciones. Una vez creadas las clases que implementan estas reglas con sus respectivos métodos, será necesario declararlas en el archivo `validator-rules.xml`. Más adelante en este Capítulo analizaremos la creación de reglas de validación personalizadas.

6.1.2.2 VALIDATION.XML

En este archivo de configuración el programador deberá asignar a cada campo cuyos datos quiera validar la regla o reglas de validación definidas en el archivo `validator-rules.xml`.

En el siguiente listado vemos un ejemplo de definición de una regla de validación sobre un campo de un formulario capturado mediante un `ActionForm`:

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

<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.1.3//EN"
```

```
"http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">

<form-validation>
  <formset>
    <form name="RegistroForm">
      <field
        property="nombre"
        depends="required">
        <arg key="RegistroForm.username"/>
      </field>
    </form>
  :

</formset>
</form-validation>
```

Para cada formulario que se desee validar se definirá un elemento `<form>`, cuyo atributo *name* deberá contener el nombre del objeto `ActionForm` encargado de capturar el dato de usuario. Cada campo que se quiera validar del formulario se definirá en el interior de `<form>` mediante el elemento `<field>`.

Más adelante analizaremos las distintas reglas de validación personalizadas proporcionadas por Struts y la forma de utilizarlas.

6.1.3 Clase ValidatorForm

A la hora de definir la clase `JavaBean` en la que se recogerán los datos procedentes del formulario cliente y de cara a poder utilizar la validación automática proporcionada por los métodos definidos en `validator-rules.xml`, debemos utilizar como clase base `ValidatorForm` en vez de `ActionForm`.

La clase `ValidatorForm` se encuentra definida en el paquete `org.apache.struts.validator` y es a su vez una subclase de `ActionForm` que proporciona su propia implementación del método `validate()`. En ella se invoca a los distintos métodos de validación definidos en `validator-rules.xml`, según la información suministrada en `validation.xml`. Por tanto, a la hora de crear nuestra propia subclase de `ValidatorForm` y de cara a utilizar la validación automática proporcionada por los validadores, no tendremos que sobrescribir el método `validate()`, debiendo mantenerse la implementación proporcionada por esta clase base.

6.1.4 Archivo `ApplicationResource.properties`

Como hemos tenido oportunidad de ver en Capítulos anteriores, el archivo de recursos `ApplicationResource.properties` se utiliza en las aplicaciones Struts para almacenar cadenas de texto a las que se les asocia una clave, cadenas que pueden tener diferentes usos dentro de la aplicación.

En el caso concreto de los validadores se hace uso de este archivo de recursos para almacenar los mensajes de error devueltos por cada validador cuando se incumpla el criterio de validación definido para los campos. El archivo `ApplicationResource.properties`, incluido en el paquete de distribución de Struts, define una serie de mensajes de error predeterminados para cada uno de los validadores proporcionados. El siguiente listado muestra el contenido del archivo de recursos con los mensajes de error incluidos en el mismo y sus claves asociadas:

`errors.invalid={0} is invalid.`

`errors.maxlength={0} can not be greater than {1} characters.`

`errors.minlength={0} can not be less than {1} characters.`

`errors.range={0} is not in the range {1} through {2}.`

`errors.required={0} is required.`

`errors.byte={0} must be an byte.`

`errors.date={0} is not a date.`

`errors.double={0} must be an double.`

`errors.float={0} must be an float.`

`errors.integer={0} must be an integer.`

`errors.long={0} must be an long.`

`errors.short={0} must be an short.`

`errors.creditcard={0} is not a valid credit card number.`

`errors.email={0} is an invalid e-mail address.`

El convenio utilizado a la hora de definir un mensaje de error para un determinado validador es: *errors.nombre_validador*. Por supuesto se pueden modificar estos mensajes y sustituirlos por otros que consideremos más adecuados, si bien lo más conveniente en estos casos será sobrescribir los mensajes añadiendo nuevas parejas clave=mensaje, en vez de modificar los predefinidos. Más adelante en este Capítulo veremos cómo realizar esta tarea.

De cara a poder introducir cierto grado de personalización en los mensajes, cada uno de ellos incluye una serie de parámetros ({0}, {1}, ...) que pueden ser sustituidos durante el proceso de validación por valores concretos, tal y como veremos más adelante.

6.2 UTILIZACIÓN DE VALIDADORES

La utilización de los métodos de validación proporcionados por Struts para realizar la validación automática de los datos de usuario resulta tremendamente sencilla, consistiendo básicamente estas tareas en incluir unas pocas instrucciones de declaración en un archivo de configuración.

Para ver el proceso a seguir realizaremos como ejemplo la validación de los credenciales de usuario suministrados a través de una página de login, para lo que partiremos de la página login.jsp utilizada en alguna de las prácticas de los Capítulos precedentes. El criterio de validación que vamos a utilizar en el ejemplo consistirá en verificar que se han suministrado tanto el identificador de usuario como el password en los campos del formulario, antes de proceder a la identificación del usuario en la base de datos.

Como paso previo habrá que registrar el plug-in validator en struts-config.xml, tal y como se mostró en el punto anterior, indicando en la propiedad “pathnames” la dirección relativa a la aplicación Web de los archivos validator-rules.xml y validation.xml. Habitualmente estos archivos se situarán en el directorio raiz_aplicacion\WEB-INF, junto con el resto de archivos de configuración de la aplicación. Como archivo validator-rules.xml utilizaremos el proporcionado por Struts, mientras que validation.xml estará inicialmente vacío.

Después de realizar estas operaciones previas veremos el resto de pasos a seguir.

6.2.1 Creación de la clase ValidatorForm

La clase `ValidacionForm` que encapsula los credenciales del usuario deberá heredar ahora `ValidatorForm` en vez de `ActionForm`, manteniendo por lo demás los mismos datos miembro y métodos *set/get* que antes:

```
import org.apache.struts.validator.*;

public class ValidacionForm extends ValidatorForm {
    //datos miembro
    private String usuario;
    private String password;
    //métodos de acceso
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String nombre) {
        this.usuario = nombre;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Esta clase deberá estar registrada en el archivo `struts-config.xml` exactamente igual que cuando heredaba a `ActionForm`.

6.2.2 Definición de los criterios de validación

Para realizar el tipo de validación indicada anteriormente debemos utilizar el validador de tipo “required”, nombre con el que se encuentra registrado el método de validación correspondiente en el archivo `validator-rules.xml`.

Dado que la validación se va a realizar sobre los campos del objeto `ValidacionForm`, tendremos que añadir al archivo `validation.xml` un elemento `<form>` asociado al mismo, elemento que deberá estar definido dentro del elemento `<formset>` que se incluye a su vez en el elemento raíz `<form-validation>` del documento:

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

```
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD
        Commons Validator Rules Configuration 1.1.3//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
    <formset>
        <form name="ValidacionForm">
            :
        </form>
    </formset>
</form-validation>
```

En el interior de `<form>` incluiremos dos elementos `<field>`, uno por cada campo que queremos validar. Cada elemento `<field>` dispondrá de dos atributos:

- **property.** Propiedad del objeto `ValidacionForm` o nombre del campo que queremos validar.
- **depends.** Nombre del validador o validadores cuyos criterios de validación queremos aplicar sobre el campo, en nuestro caso será “required” para ambos campos. Si queremos aplicar más de una regla de validación sobre un campo, habrá que indicar en este atributo los nombres de todos los validadores separados por una coma.

Cada elemento `<field>` incluirá además un elemento `<arg>` donde se indicará el argumento correspondiente al parámetro definido en el mensaje de error. Para ello se deberán definir los siguientes atributos del elemento:

- **key.** Nombre de la clave asociada a la cadena que se suministrará como argumento, cadena que estará definida en `ApplicationResource.properties` y cuyo nombre de clave suele seguir el convenio: `objeto_ValidatorForm.campo`.
- **resource.** Se trata de un atributo de tipo *boolean*. Si su valor es *true* (es el valor predeterminado en caso de que el atributo no se utilice) el valor del atributo *key* será interpretado como nombre de clave, mientras que si es *false* el valor de este atributo será tomado como el literal correspondiente al valor del argumento.

Además de `<arg>`, un elemento `<field>` puede incluir opcionalmente los elementos `<arg0>`, `<arg1>`, `<arg2>` y `<arg3>`, mediante los cuales se suministrarán

los argumentos para el resto de parámetros que pudieran incluir los mensajes de error. Estos elementos disponen de los mismos atributos que `<arg>`. Si el mensaje de error requiere únicamente un argumento, puede utilizarse `<arg>` o `<argo>` indistintamente.

Con todo ello, el archivo `validation.xml` del ejemplo que estamos analizando quedaría:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD
        Commons Validator Rules Configuration 1.1.3//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">
<form-validation>
    <formset>
        <form name="ValidacionForm">
            <field
                property="usuario"
                depends="required">
                <arg key="ValidacionForm.usuario"/>
            </field>
            <field
                property="password"
                depends="required">
                <arg key="ValidacionForm.password"/>
            </field>
        </form>
    </formset>
</form-validation>
```

6.2.3 Habilitación de la validación en cliente

Siempre que sea posible se debería realizar la validación de los datos de usuario en la página cliente, esto hará que la petición de envío de datos al servidor no sea lanzada en caso de que éstos no cumplan con los criterios de validación definidos, evitando así la sobrecarga del servidor y mejorando el rendimiento de la aplicación.

Pero aunque la validación en cliente constituye una buena práctica de programación, por motivos de seguridad y porque el navegador cliente puede tener desactivado el uso de JavaScript (que es el lenguaje utilizado por el código cliente para validar los datos en la página Web), será necesario también realizar la

validación de los datos en el servidor antes de su procesamiento. Ambos tipos de validación, en servidor y en cliente, siguen los criterios definidos en los validadores, tal y como se ha explicado en las secciones anteriores.

A fin de poder realizar la validación desde el navegador, el plug-in validator inyecta una serie de instrucciones JavaScript en la página cliente que serán ejecutadas antes de proceder al envío de la petición. Pero para que esto suceda será necesario habilitar la validación en cliente añadiendo la siguiente acción html en cualquier parte de la página:

```
<html:javascript formName="objeto_form"/>
```

La acción deberá indicar a través del atributo *formName* el nombre del objeto ValidatorForm.

Así mismo, desde el manejador del evento *onsubmit* del formulario se deberá incluir una llamada a la función JavaScript encargada de desencadenar el proceso de validación. Dado que esta función se genera de forma dinámica, su nombre tendrá el formato *validateObjetoForm*, siendo ObjetoForm el nombre del objeto ValidatorForm. La llamada a ese método incluirá como argumento el valor *this*. Así quedará por tanto la página login.jsp del ejemplo:

```
<html:html>
  <body>
    <center>
      <h1>Formulario de autenticación</h1>
      <html:javascript formName="ValidacionForm"/>
      <html:form action="/validar" method="post"
        onsubmit="return validateValidacionForm(this);">
        <table>
          <tr>
            <td>Usuario:</td>
            <td><html:text property="usuario"/></td>
          </tr>
          <tr>
            <td>Password:</td>
            <td><html:password property="password"/></td>
          </tr>
          <tr>
            <td colspan="2">
              <br/>
              <html:submit property="submit" value="Validar"/>
            </td>
          </tr>
        </table>
      </html:form>
    </center>
  </body>
</html:html>
```




Fig. 30. Mensaje de error de validación

6.3 VALIDADORES PREDEFINIDOS DE STRUTS

Ya hemos visto el funcionamiento de uno de los validadores clásicos que se utilizan en la mayoría de las aplicaciones Web, el validador “required”. Seguidamente analizaremos otros de los validadores proporcionados por Struts que se emplean con más frecuencia en las aplicaciones.

6.3.1 minlength

Este validador comprueba que el valor numérico suministrado en un campo sea mayor o igual a una determinada constante.

Además de los subelementos <argn>, el elemento <field> utilizado en la declaración de este validador en validation.xml deberá incluir un subelemento <var> en el que se indique la constante con la que se comparará el valor del campo. La estructura de este elemento es:

```
<var>

    <var-name>nombre</var-name>

    <var-value>valor</var-value>

</var>
```

donde `<var-name>` contiene el nombre de la constante (en el caso de *minlength* el nombre de la variable será el mismo, “minlength”) y `<var-value>` su valor.

El siguiente ejemplo define una regla de validación para el campo “edad” de un formulario, que fuerza al usuario a la introducción de un valor en el mismo igual o superior a 18:

```
<field
  property="edad"
  depends=" minlength">
  <arg0 key="RegistroForm.edad"/>
  <arg1 key="RegistroForm.edad.min"/>
  <var>
    <var-name>minlength</var-name>
    <var-value>18</var-value>
  </var>
</field>
```

En este validador vemos cómo el mensaje de error requiere de dos argumentos, el nombre del campo y el valor mínimo.

6.3.2 maxlength

Su estructura es igual a la de *minlength*, validando el campo indicado en el atributo `property` de `<field>` si su valor es igual o inferior a la constante indicada en el elemento `<var>`. El siguiente ejemplo establece que el valor del campo “limite” debe ser igual o inferior a 600:

```
<field
  property="limite"
  depends="required, maxlength">
  <arg0 key="InfoForm.limite"/>
  <arg0 name="maxlength" key="InfoForm.limite"/>
  <arg1 name="maxlength" key="InfoForm.limite.max"/>
  <var>
    <var-name>maxlength</var-name>
    <var-value>600</var-value>
  </var>
</field>
```

En el ejemplo anterior vemos cómo en el mismo elemento `<field>` se definen dos validadores para el mismo campo. En estos casos, el atributo *name* del

elemento `<argn>` permite indicar el tipo de validador al que está asociado el argumento.

6.3.3 byte, short, integer, long, float y double

Mediante estos validadores podemos comprobar si el contenido de un campo es compatible con el tipo de dato representado por el validador. Por ejemplo, la siguiente regla comprueba que el dato suministrado en el campo “saldo” sea de tipo *double*:

```
<field
  property="saldo"
  depends="double">
  <arg key="CuentaForm.saldo" />
</field>
```

6.3.4 intRange

El validador *intRange* comprueba que el valor del campo se encuentra comprendido dentro de un determinado rango numérico entero, cuyos valores máximo y mínimo estarán determinados por los subelementos `<var>` llamados “max” y “min”.

La siguiente regla permite verificar que el valor del campo “nota” está comprendido entre 1 y 10:

```
<field
  property="nota"
  depends="intRange">
  <arg0 key="InfoForm.nota" />
  <arg1 key="InfoForm.nota.min" />
  <arg2 key="InfoForm.nota.max" />
  <var>
    <var-name>min</var-name>
    <var-value>1</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>10</var-value>
  </var>
</field>
```

Como se puede observar en el ejemplo y también en el archivo de recursos `ApplicationResource.properties`, el mensaje de error asociado a este validador necesita tres parámetros: el nombre del campo, el valor mínimo y el máximo. Los argumentos que se utilizan en el ejemplo se encontrarían almacenados en el archivo de recursos con las claves `InfoForm.nota`, `InfoForm.nota.min` e `InfoForm.nota.max`, respectivamente.

6.3.5 floatRange y doubleRange

Al igual que el anterior, estos validadores comprueban que el valor del campo se encuentra delimitado dentro de un determinado rango numérico, sólo que en estos caso ese rango es de tipo *float* y *double*, respectivamente. La estructura del elemento `<field>` es exactamente igual a la de *intRange*.

6.3.6 date

Comprueba que el valor del campo tiene un formato de fecha válido:

```
<field
  property="fechainicio"
  depends="date">
  <arg key="EmpleadoForm.fecha"/>
</field>
```

Opcionalmente el elemento `<field>` admite una variable, llamada *datePattern*, mediante la que se puede especificar el tipo de formato de fecha admitida para el campo. Por ejemplo, la siguiente regla establece para el campo “fechainicio” el formato de tipo día/mes/año:

```
<field
  property="fechainicio"
  depends="date">
  <arg key="EmpleadoForm.fecha"/>
  <var>
    <var-name>datePattern</var-name>
    <var-value>dd/MM/yyyy</var-value>
  </var>
</field>
```

6.3.7 mask

A través de este validador obligamos a que el contenido de un campo se ajuste al formato definido en una máscara. Esta máscara estará definida dentro de la variable “mask”. Para definir una máscara utilizaremos las reglas de construcción de aplicaciones regulares en Java. Por ejemplo, la siguiente regla de validación establece que el contenido del campo “url” debe ser la dirección de un dominio .com:

```
<field
  property="url"
  depends="mask">
  <arg key="EmpresaForm.url"/>
  <var>
    <var-name>mask</var-name>
    <var-value>www\..+\.com</var-value>
  </var>
</field>
```

6.3.8 email

Otro de los campos habituales que nos podemos encontrar en un formulario cliente es la dirección de correo electrónico. Mediante este validador y sin necesidad de proporcionar ningún tipo de información adicional, se comprueba que la cadena de caracteres almacenada en un determinado campo se ajusta al formato de dirección de correo:

```
<field
  property="correo"
  depends="email">
  <arg key="EmpleadoForm.email"/>
</field>
```

6.4 MENSAJES DE ERROR PERSONALIZADOS

Si queremos modificar los mensajes de error que se muestran al usuario cuando el campo incumple un criterio de validación, la mejor forma de hacerlo será seguir los siguientes pasos:

- **Añadir el mensaje al archivo de recursos.** En el archivo de recursos `ApplicationResource.properties` escribiremos el mensaje personalizado que queremos mostrar, asignándole una clave al

misimo. Dado que los mensajes predefinidos tienen como clave asociadas: `errors.nombre_validador`, deberíamos utilizar un convenio diferente para las claves personalizadas, como por ejemplo: `errors.personalizados.nombre_validador`.

- **Indicar el mensaje a utilizar en `<field>`.** Para indicar que en la validación de un determinado campo se va a utilizar un mensaje de error distinto al predeterminado, debemos emplear el subelemento `<msg>` de `<field>`, indicando mediante sus atributos *name* y *key* el nombre del validador al que se asocia el mensaje (sólo es necesario cuando se definen más de un validador para el campo) y la clave del mismo, respectivamente. En el siguiente ejemplo asociamos un mensaje de error personalizado con la validación de una dirección de correo electrónico:

```
<field
    property="correo"
    depends="email">
    <msg key="errors.personalizados.email"/>
    <arg0 key="EmpleadoForm.email"/>
</field>
```

Los argumentos indicados en `<argn>` se referirán en estos casos al nuevo mensaje definido.

PRÁCTICA 6.1. VALIDACIÓN DE LOS CAMPOS DE LA PÁGINA DE REGISTRO

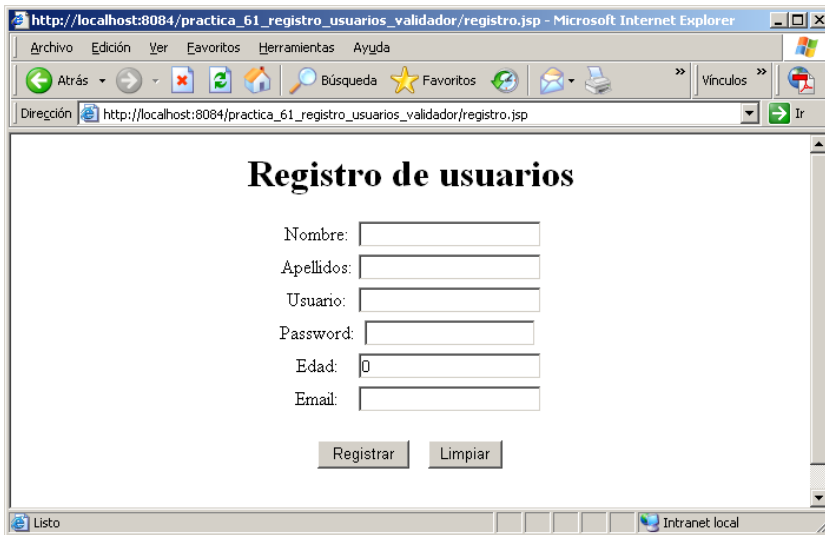
Descripción

Utilizando los validadores proporcionados por Struts vamos a definir una serie de reglas de validación para cada uno de los seis campos incluidos en una página de registro de usuarios como la que se indica en la figura 31.

Los criterios para considerar como válidos los datos suministrados a través de estos campos serán los siguientes:

- Todos los campos serán de obligada cumplimentación.
- El campo “password” tendrá que tener una longitud mínima de cuatro caracteres.
- El campo “edad” deberá ser de tipo entero.

- El campo “email” tendrá que tener una dirección de correo con formato válido.



The screenshot shows a Microsoft Internet Explorer window with the address bar displaying `http://localhost:8084/practica_61_registro_usuarios_validador/registro.jsp`. The page content is a registration form titled "Registro de usuarios". The form contains the following fields and controls:

- Nombre:
- Apellidos:
- Usuario:
- Password:
- Edad:
- Email:
- Buttons: "Registrar" and "Limpiar"

The status bar at the bottom shows "Listo" and "Intranet local".

Fig. 31. *Página de registro de usuarios*

Desarrollo

Según los criterios anteriores, todos los campos deberán tener asignado el validador *required*. Además de éste, los campos “password”, “edad” e “email” deberán tener asignado el validado *minlength*, *integer* e *email*, respectivamente.

Para realizar únicamente la tareas de validación de datos no será necesario incluir ninguna sentencia de código Java, tan sólo será necesario asignar en `validation.xml` los validadores indicados a cada control.

Por otro lado, también será necesario registrar en el archivo de recursos `ApplicationResource.properties` el texto que se utilizará como argumento en los mensajes de error para cada control.

En cuanto a la página `registro.jsp`, añadiremos las instrucciones necesarias para habilitar la validación en cliente, así como el volcado de los mensajes de error en la página para la validación en servidor.

Listado

Seguidamente mostraremos el código de los archivos de texto relacionados con la validación.

ApplicationResource.properties

Será necesario añadir a este archivo las siguientes líneas de texto:

```
RegistroForm.nombre="nombre"
RegistroForm.apellidos="apellidos"
RegistroForm.usuario="nombre"
RegistroForm.password="password"
RegistroForm.edad="edad"
RegistroForm.email="email"
RegistroForm.password.min="4"
```

validation.xml

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

<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.1.3//EN"

    "http://jakarta.apache.org/commons/dtds/validator_1_1_3.dtd">

<form-validation>
    <formset>
        <form name="RegistroForm">
            <field
                property="nombre"
                depends="required">
                <arg key="RegistroForm.nombre"/>
            </field>
            <field
                property="apellidos"
                depends="required">
                <arg key="RegistroForm.apellidos"/>
            </field>
            <field
                property="usuario"
                depends="required">
```

```

        <arg key="RegistroForm.usuario"/>
    </field>
    <field
        property="password"
        depends="required,minlength">
        <arg0 key="RegistroForm.password"/>
        <arg1 name="minlength"
            key="RegistroForm.password.min"/>
        <var>
            <var-name>minlength</var-name>
            <var-value>4</var-value>
        </var>
    </field>
    <field
        property="edad"
        depends="required, integer">
        <arg0 key="RegistroForm.edad"/>
        <arg1 key="RegistroForm.edad"/>
    </field>
    <field
        property="email"
        depends="required,email">
        <arg0 key="RegistroForm.email"/>
    </field>
</form>
</formset>
</form-validation>

```

registro.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html:html>
    <body>
        <center>
            <h1>Registro de usuarios</h1>

```



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

6.5 VALIDACIONES PERSONALIZADAS

En algunas ocasiones puede ocurrir que con los validadores predefinidos de Struts no cubramos todas las necesidades de validación para los campos de nuestra aplicación.

En estos casos, si tuviéramos que aplicar una regla de validación no incluida en los validadores de Struts, podemos optar por una de las siguientes soluciones:

- Sobrescribir el método *validate()* del *ValidatorForm* para incluir nuestras propias instrucciones de la validación.
- Definir un método validador personalizado.

6.5.1 Sobrescritura del método *validate()*

Anteriormente comentamos que a la hora de crear la clase bean que heredase *ValidatorForm* debíamos mantener la implementación por defecto del método *validate()* proporcionado por esta clase base.

Si queremos incluir algún criterio de validación personalizado en nuestra clase y, al mismo tiempo, mantener la validación automática proporcionada por Struts, podemos sobrescribir en nuestra clase *ValidatorForm* el método *validate()* e incluir manualmente en él las instrucciones de validación propias, siempre y cuando agreguemos también a éste una **llamada al método *validate()* de la superclase**.

El formato de método *validate()* proporcionado por *ValidatorForm* y que podemos sobrescribir es el siguiente:

```
public ActionErrors validate(ActionMapping mapping,  
  
                             javax.servlet.http.HttpServletRequest request)
```

Por ejemplo, supongamos que en la página *registro.jsp* de la práctica anterior quisiéramos incluir un campo adicional (llamado “passwordrep”) donde el usuario deba volver a introducir el password por segunda vez, comprobando a

continuación que el valor introducido en el segundo campo coincide con el del primero.

Dado que este criterio de validación no es implementado por ninguno de los validadores proporcionados por Struts, deberíamos sobrescribir el método *validate()* de nuestra subclase RegistroForm e incluir en él las instrucciones necesarias para realizar esta comprobación. El siguiente listado muestra cómo quedaría la nueva implementación de la clase RegistroForm, indicando en fondo sombreado las instrucciones añadidas:

```
package javabeans;
import org.apache.struts.validator.*;
public class RegistroForm extends ValidatorForm {
    private String nombre;
    private String apellidos;
    private String usuario;
    private String password;
    private String passwordrep;
    private String email;
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellidos() {
        return apellidos;
    }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos;
    }
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public String getPassword() {
        return password;
    }
}
```

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

public String getPasswordrep() {
    return passwordrep;
}

public void setPasswordrep(String passwordrep) {
    this.passwordrep = passwordrep;
}

public String getEmail() {
    return email;
}

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

public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest req){
    ActionErrors errors=super.validate(mapping,req);
    if(!password.equals(passwordrep)){
        errors.add("password",
            new ActionMessage("error.password.nomatch"));
    }
    return errors;
}
}
```

6.5.2 Creación de validadores personalizados

Si queremos definir una regla de validación que pueda ser utilizada en diversos proyectos, la solución será crear nuestro propio método validador. Esta operación requiere realizar las siguientes tareas:

- Implementar el método de validación.
- Registrar el validador en validator-rules.xml.
- Definir los mensajes de error predefinidos.
- Utilizar el validador.
- Implementación de un método de validación.

6.5.2.1 IMPLEMENTACIÓN DEL MÉTODO DE VALIDACIÓN

En una clase estándar de Java implementaremos el método que deberá ser invocado por Struts para validar el contenido de un campo asociado al mismo, incluyendo en dicho método la lógica necesaria para la comprobación de los datos según nuestras propias reglas personalizadas.

El método podrá llamarse de cualquier forma, si bien deberá respetar el siguiente formato:

```
public boolean nombre_metodo(Object ob,  
  
org.apache.commons.validator.ValidatorAction v,  
  
org.apache.commons.validator.Field f,  
  
org.apache.struts.action.ActionMessages msgs,  
  
org.apache.commons.validator.Validator vt,  
  
javax.servlet.http.HttpServletRequest request)
```

Como vemos, el método deberá declarar seis parámetros cuyo significado es el siguiente:

- **Object.** Objeto bean que contiene el campo que será validado.
- **ValidatorAction.** Representa el elemento <validator> definido en validator-rules.xml, asociado a esta regla de validación.
- **Field.** Representa el elemento <field> definido en validation.xml, asociado al campo que está siendo validado.
- **ActionMessages.** Contiene los mensajes de error asociados al campo que se está validando.
- **Validator.** Instancia actual del objeto Validator que controla el proceso de validación
- **HttpServletRequest.** Objeto que contiene los datos de la petición en curso.

Por otro lado, el método deberá devolver un valor de tipo *boolean* que indicará a Struts el resultado de la validación (*true* validación correcta, *false* validación incorrecta).

Por ejemplo, supongamos que quisiéramos crear una regla de validación especial para las contraseñas suministradas mediante un formulario cliente, consistente en obligar al usuario a incluir un carácter de puntuación en la misma, carácter que limitaremos a uno de los siguientes: “.”, “,” o “-”. Dicha regla de validación la incluiremos dentro de un método llamado “passwordValida” que implementaremos en una clase llamada “Personalizado”. He aquí el código de la clase:

```
package validadores;

import org.apache.commons.validator.util.*;

public class Personalizado {

    public boolean passwordValida(Object ob,
        org.apache.commons.validator.ValidatorAction v,
        org.apache.commons.validator.Field f,
        org.apache.struts.action.ActionMessages msgs,
        org.apache.commons.validator.Validator vt,
        javax.servlet.http.HttpServletRequest request){
        String s=ValidatorUtils.
            getValueAsString(ob,f.getProperty());
        if(s.contains(".")||s.contains(",")||
            s.contains("-")){
            //contraseña válida
            return true;
        }
        else{
            return false;
        }
    }
}
```

Como podemos apreciar en este código, hacemos uso del método estático *getValueAsString()* incluido en la clase *ValidatorUtils* para recuperar el valor contenido en el campo a validar, método que requiere el bean donde se encuentra incluido el campo y la propiedad que lo define, la cual puede obtenerse a partir del objeto *Field*.

6.5.2.2 REGISTRO DEL VALIDADOR

Una vez definido el método debemos registrar la nueva regla de validación en el archivo de configuración `validator-rules.xml` de la aplicación en donde queramos hacer uso de ella. Para ello, debemos añadir un nuevo elemento `<validator>` con los datos del validador, como son el nombre asignado al mismo, el método que implementa la regla de validación y la clase donde está definido, los parámetros que recibirá el método y la clave asociada al mensaje de error. Estos datos se suministrarán a través de los atributos de `<validator>` analizados al principio del Capítulo.

Así mismo, se deberá incluir dentro del elemento `<validator>` un subelemento `<javascript>` en el que se defina la función `javascript` que tendrá que ser ejecutada por el navegador para realizar la validación en cliente. A esta función se le asigna normalmente el mismo nombre que al método de la clase de servidor.

La información a incluir en `validator-rules.xml` para el caso del ejemplo que se está analizando se muestra en el siguiente listado:

```
<validator name="pwdValid"
    classname="validadores.Personalizado"
    method="passwordValida"
    methodParams="java.lang.Object,
        org.apache.commons.validator.ValidatorAction,
        org.apache.commons.validator.Field,
        org.apache.struts.action.ActionMessages,
        org.apache.commons.validator.Validator,
        javax.servlet.http.HttpServletRequest"
    depends=""
    msg="errors.pwd">
<javascript>
<![CDATA[
function passwordValida(form) {
    var bValid = false;
    var focusField = null;
    var i = 0;
    var fields = new Array();
    var formName = form.getAttributeNode("name");
    oPwd = eval('new ' +
        formName.value + '_pwdValid()');
    for (x in oPwd) {
        var field = form[oPwd[x][0]];
```

```
//comprobación de tipo de campo
    if ((field.type == 'hidden' ||
        field.type == 'text' ||
        field.type == 'textarea' ||
        field.type == 'password' ||
        field.type == 'select-one' ||
        field.type == 'radio') &&
        field.disabled == false) {
        var value = '';
        //obtiene el valor del campo
        value = field.value;
        //aplica la regla de validación
        if (value.length > 0) {
            focusField = field;
            fields[i++] = oPwd[x][1];
            if (value.indexOf(".") != -1 ||
                value.indexOf(",") != -1 ||
                value.indexOf("-") != -1) {
                bValid = true;
            }
        }
    }
}
if ((fields.length > 0) && !bValid) {
    focusField.focus();
    alert(fields.join('\n'));
}
return bValid;
}]]>
</javascript>
</validator>
```

Para definir la función JavaScript anterior podemos utilizar como modelo cualquiera de las que utilizan los validadores proporcionados por Struts. Toda función de validación deberá cumplir dos mínimas reglas, la primera es que deberá declarar un parámetro en el que recibirá el formulario con el campo a validar, y la segunda es que la función deberá devolver un valor de tipo *boolean* que informe del resultado de la validación.

6.5.2.3 MENSAJES DE ERROR

Los mensajes de error por defecto asociados al validador se incluirán en el archivo `ApplicationResource` de la aplicación. Para realizar la asociación se indicará en el atributo `msg` del elemento `<validator>` el nombre de la clave que se le ha asociado. Este mensaje puede incluir parámetros si así lo deseamos. La siguiente línea correspondería al mensaje asociado al validador del ejemplo:

errors.pwd=Un password debe contener signos de puntuación

6.5.2.4 UTILIZACIÓN DEL VALIDADOR

Una vez registrada la regla de validación, para poder hacer uso de la misma en una aplicación hay que seguir exactamente los mismos pasos que indicamos con los validadores predefinidos de Struts, esto es, tendremos que asociar la regla al campo que queramos validar dentro del archivo `validation.xml` y activar, si procede, en la página JSP la validación en cliente.

En el ejemplo que estamos tratando, si quisiéramos validar el campo `password` de un formulario con la nueva regla que hemos definido, tendremos que añadir el siguiente elemento `<field>` al archivo `validation.xml`:

```
<form name="RegistroForm">
  :
  <field
    property="password"
    depends="pwdValid">
    <arg key="RegistroForm.password"/>
  </field>
</form>
```

En cuanto a la activación de la validación en cliente, incluiremos la siguiente definición en la página JSP:

```
<h1>Registro de usuarios</h1>
<html:javascript formName="RegistroForm"/>
<html:form action="/registrar" method="POST"
  onsubmit="return validateRegistroForm(this);">
  :
  :
```

La función `validateRegistroForm()`, invocada desde el manejador de evento `onsubmit` del formulario, es generada dinámicamente por Struts. El nombre de esta función deberá tener siempre el formato `validateObjetoForm`.

UTILIZACIÓN DE PLANTILLAS

Las plantillas o tiles, como habitualmente se les conoce en Struts, fueron incorporados al núcleo de la especificación a partir de la versión 1.1.

Su objetivo no es otro que posibilitar la reutilización de código XHTML/JSP durante el desarrollo de la Vista en una aplicación Web MVC, simplificando el desarrollo de las páginas y haciendo también más cómodos y menos propensos a errores los posibles cambios que se puedan realizar a posteriori sobre esta capa de la aplicación.

La utilización de plantillas resulta adecuada por tanto en aquellas aplicaciones Web en las que tenemos un grupo de páginas con estructura similar, en las que el contenido de ciertas partes de las mismas es igual en todas ellas.

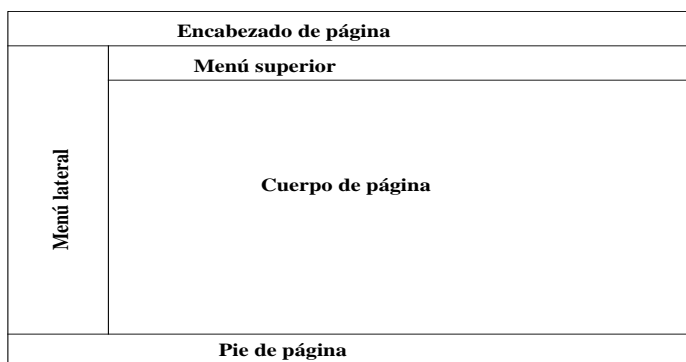


Fig. 32. *Clásico tipo de páginas para utilización de plantillas*

Este es el caso de las páginas que suelen tener la típica estructura que se muestra en la figura 32, donde la zona de encabezado, pie y menú suele ser idéntica para un cierto grupo de páginas de la aplicación, cambiando sólo el cuerpo de una a otra.

7.1 CONFIGURACIÓN DE LA APLICACIÓN PARA EL USO DE PLANTILLAS

Para poder utilizar plantillas en una aplicación Struts es necesario realizar dos operaciones previas en el archivo de configuración `struts-config.xml`.

La primera de ellas consiste en indicar al Controlador de Struts que debe utilizar un objeto `RequestProcessor` a partir de una clase diseñada explícitamente para el uso de plantillas, como es el caso de la **clase `TilesRequestProcessor`** que se encuentra en el paquete `org.apache.struts.tiles`. Así pues, debemos añadir la siguiente entrada en el archivo de configuración `struts-config.xml`, a continuación del elemento `<action-mappings>`:

```
<controller processorClass=
    "org.apache.struts.tiles.TilesRequestProcessor" />
```

La segunda de las operaciones previas será la activación del plug-in `tiles` encargado de gestionar las plantillas. Para ello añadiremos la siguiente entrada en `struts-config.xml`:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
    <set-property property="definitions-config"
        value="/WEB-INF/tiles-defs.xml" />
    <set-property property="moduleAware" value="true" />
</plug-in>
```

Los elementos anteriores se encuentran ya incluidos de forma predeterminada en el archivo `struts-config.xml` que se proporciona con el paquete de distribución de Struts, por lo que no será necesario añadirlos de forma manual.

7.2 CREACIÓN DE UNA APLICACIÓN STRUTS BASADA EN PLANTILLAS

La idea básica en la que se fundamenta este tipo de aplicaciones consiste en la definición de una plantilla o página maestra en la que se define una

determinada estructura tipo de página, indicando las distintas zonas en la que éstas estarán divididas y la distribución de las mismas.

Esta página maestra sirve de base para la creación de las páginas JSP que forman la vista de la aplicación, en la que únicamente habrá que indicar el contenido que debe ser incluido dentro de cada zona.

Tanto la creación de la página maestra como de las páginas de aplicación requieren de la utilización de la librería de acciones *tiles* incluida en el núcleo de Struts, por lo que habrá que añadir la siguiente directiva *taglib* al principio de cada una de las páginas:

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

A continuación analizaremos en detalle los pasos a seguir en la construcción de este tipo de aplicaciones.

7.2.1 Creación de la plantilla

Como acabamos de comentar, la plantilla será implementada mediante una página JSP, dicha página deberá estar situada en el directorio privado de la aplicación WEB-INF a fin de que no pueda ser accesible directamente desde el navegador cliente.

Utilizaremos etiquetas XHTML estándares para indicar el contenido estático que va a ser idéntico en todas las páginas, así como las acciones y scriptlets JSP que deben ser ejecutados en todas ellas.

En cuanto a las zonas de contenido dependiente de cada página, deberán ser marcadas mediante dos posibles tipos de acciones *tiles*:

- **insert.** Esta acción tiene diversos usos como iremos viendo a lo largo del Capítulo. En las páginas maestras se utiliza para indicar la existencia de una sección de contenido, sección a la que se le asignará un nombre mediante el atributo *attribute* de la acción. El siguiente ejemplo define una sección de contenido llamada “header”, situada dentro de una etiqueta `<div>`:

```
<div><tiles:insert attribute="header"></div>
```

Durante la creación de las páginas de la aplicación estas secciones serán sustituidas por el contenido definido en determinados ficheros externos, habitualmente páginas JSP.

- **getAsString.** Define una sección de texto, utilizando el atributo name para asignar el nombre a la sección:

```
<h1><tiles:getString name="titulo"/></h1>
```

Como sucede con <tiles:insert>, será durante la creación de la página de aplicación cuando esta acción sea sustituida por un texto real.

Vamos a ver un ejemplo concreto que sirva para aclarar conceptos. Imaginemos que queremos crear una aplicación con dos páginas cuyo aspecto se muestra en la figura 33.

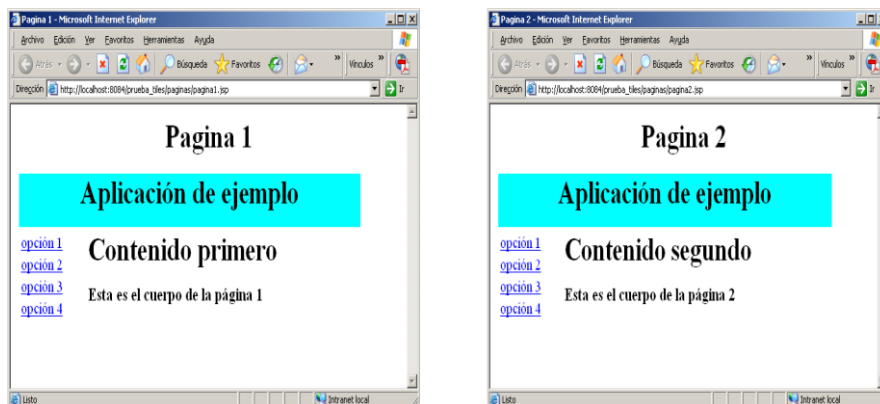


Fig. 33. Páginas de aplicación de ejemplo

Como vemos, ambas páginas están organizadas en cuatro zonas: un texto en la parte superior, que es distinto para cada página, un encabezado, un menú en el lateral izquierdo, ambos idénticos en las dos páginas, y un cuerpo que ocupa la parte central de la página y que también estará personalizado para cada una de ellas.

La plantilla que define la organización anterior quedaría tal y como se indica en el siguiente listado:

```
<%@page contentType="text/html" %>
```

```
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
                                prefix="tiles" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                                Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
                                charset=UTF-8">
    <title><tiles:getAsString name="titulo"/></title>
  </head>
  <body>
    <center>
      <h1><tiles:getAsString name="titulo"/></h1>
    </center>
    <table width="90%">
      <tr>
        <td height="25%" colspan="2">
          <tiles:insert attribute="encabezado"/>
        </td>
      </tr>
      <tr>
        <td>
          <tiles:insert attribute="menu"/>
        </td>
        <td>
          <tiles:insert attribute="cuerpo"/>
        </td>
      </tr>
    </table>
  </body>
</html>
```

7.2.2 Creación de piezas de contenido

Las piezas de contenido representan las porciones de código XHTML/JSP que sustituirán a los elementos `<tiles:insert>` durante la creación de las páginas de aplicación.

Por lo general, estos bloques de contenido se almacenan en archivos .jsp y, dado que su contenido formará parte de las páginas finales, pueden incluir tanto código XHTML/JSP estándar como acciones de Struts.

Al igual que sucede con las páginas maestras, para evitar que puedan ser directamente accesibles desde el cliente, estos archivos deberían estar situados en el directorio WEB-INF de la aplicación.

Los siguientes ejemplos representan las páginas que podrían ser utilizadas como contenidos para las zonas definidas en la plantilla anterior:

titulo.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<div align="center" style="background-color:aqua">
    <h1>Aplicación de ejemplo</h1>
</center>
```

menu.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri="http://struts.apache.org/tags-html"
           prefix="html" %>

<table>
    <tr>
        <td>
            <a href="">opción 1</a><br/>
        </td>
    </tr>
    <tr>
        <td>
            <a href="">opción 2</a><br/>
        </td>
    </tr>
    <tr>
        <td>
            <a href="">opción 3</a><br/>
        </td>
    </tr>
</table>
```

```
<td>
    <a href="">opción 4</a><br/>
</td>
</tr>
</table>
```

body1.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<h1>Contenido primero</h1>
<h3>Este es el cuerpo de la página 1</h3>
```

body2.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<h1>Contenido segundo</h1>
<h3>Este es el cuerpo de la página 2</h3>
```

7.2.3 Creación de las páginas de aplicación

Una vez que se dispone de la plantilla y de las piezas de contenido, crear las páginas de la aplicación resulta una tarea tremendamente sencilla. Para ello tan sólo será necesario declarar al principio de la página la plantilla que se va a utilizar e indicar a continuación las piezas que queremos situar en cada zona de contenido.

7.2.4 Declaración de la plantilla

La declaración de la plantilla se lleva a cabo mediante la acción `<tiles:insert>`, utilizando en este caso el atributo *page* de la acción para indicar la dirección relativa de la página maestra. Por ejemplo, suponiendo que la plantilla se encuentra definida en un archivo llamado `base.jsp` que está situado en el directorio `WEB-INF`, la declaración de la plantilla en la página de aplicación sería:

```
<tiles:insert page="/WEB-INF/base.jsp">

    <!--Aquí referencias a páginas de contenido-->

</tiles:insert>
```

7.2.5 Inclusión de páginas de contenido

En el interior de insert debemos indicar las piezas que queremos incluir en cada zona de contenido especificada en la plantilla. Esta sencilla operación es realizada mediante la acción **<tiles:put>**, cuya configuración se realiza a través de sus dos atributos:

- **name.** Nombre de la zona de contenido a la que hace referencia. Éste deberá coincidir con alguno de los valores de los atributos `attribute` o `name` definidos en las acciones `<tiles:insert>` y `<tiles:getAsString>` de la plantilla.
- **value.** Contenido a incluir en la zona especificada en `name`. En el caso de zonas definidas mediante `<tiles:insert>`, `value` especificará la URL relativa de la página de contenido, mientras que en el caso de `<tiles:getAsString>` contendrá la cadena de texto por la que será sustituido el elemento.

En el ejemplo que estamos analizando, las páginas de aplicación indicadas en la figura 33 se generarán de la forma que se indica en los siguientes listados:

pagina1.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
                        prefix="tiles" %>

<!--declaración de plantilla-->
<tiles:insert page="/WEB-INF/plantillas/plantilla.jsp">
    <tiles:put name="titulo"
              value="Pagina 1"/>
    <tiles:put name="encabezado"
              value="/WEB-INF/plantillas/titulo.jsp"/>
    <tiles:put name="menu"
              value="/WEB-INF/plantillas/menu.jsp"/>
    <tiles:put name="cuerpo"
              value="/WEB-INF/plantillas/body1.jsp"/>
</tiles:insert>
```


pagina2.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
                        prefix="tiles" %>

<tiles:insert page="/WEB-INF/plantillas/plantilla.jsp">
    <tiles:put name="titulo" value="Pagina 2"/>
    <tiles:put name="encabezado"
        value="/WEB-INF/plantillas/titulo.jsp"/>
    <tiles:put name="menu"
        value="/WEB-INF/plantillas/menu.jsp"/>
    <tiles:put name="cuerpo"
        value="/WEB-INF/plantillas/body2.jsp"/>
</tiles:insert>
```

7.3 DEFINICIONES

El mecanismo de utilización de plantillas que se acaba de analizar nos permite reutilizar código en las aplicaciones, además de proporcionarnos gran flexibilidad a la hora de crear las páginas finales de la aplicación. Sin embargo, es en la creación de estas páginas finales donde observamos también la aparición de numerosos elementos repetitivos, por ejemplo, en el código de las páginas anteriores vemos cómo el elemento `<tiles:put>` utilizado en la definición de las zonas de encabezado y menú es idéntico en ambas páginas.

Otro defecto que presenta el sistema anterior es el hecho de que cualquier cambio en la ubicación de alguno de estos archivos de contenido implicará tener que actualizar todas las páginas de la aplicación donde se haga referencia a éstos, lo que puede resultar en una importante fuente de errores.

La solución que propone Struts para estos problemas consiste en la utilización de **definiciones**. Una definición, como su nombre indica, define una manera de distribuir las piezas de contenido en una plantilla, de modo que a la hora de crear una definición con contenido similar a otra ya existente no sea necesario volver a definir de nuevo los mismos elementos ya establecidos en la anterior definición, sino que bastará con que la **nueva definición herede a la primera** y redefina únicamente aquellas zonas cuyo contenido es diferente al indicado en la definición padre.

7.3.1 Creación de una definición

Antes de crear una definición debemos disponer de una plantilla y una serie de páginas de contenido. El desarrollo de estos elementos se lleva a cabo tal y como explicamos en la sección anterior.

En cuanto a las definiciones, éstas se crean en un nuevo archivo de configuración llamado **tiles-defs.xml**, cuya ubicación deberá estar indicada en el elemento `<plug-in>` definido en `struts-config.xml`, tal y como se indicó al comienzo del Capítulo. Lo habitual es que `tiles-defs.xml` se sitúe en el directorio `WEB-INF` de la aplicación.

La estructura de este archivo es la que se muestra a continuación:

```
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles
        Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/
        tiles-config_1_1.dtd">

<tiles-definitions>
:
</tiles-definitions>
```

Como vemos, existe un elemento `<tiles-definitions>` dentro del cual se incluirán todas las definiciones que se vayan a utilizar en la aplicación, cada una de ellas en su propio elemento `<definition>`.

7.3.1.1 DEFINICIONES BASE

Una definición base es aquélla que define explícitamente la distribución de los contenidos de cada zona indicada en la plantilla, sin heredar ninguna otra definición existente.

En este tipo de definición habrá que especificar los siguientes atributos del elemento `<definition>`:

- **name.** Nombre asignado a la definición.
- **path.** URL relativa de la página maestra que contiene la plantilla a partir de la que se va a crear la definición.

Para indicar el contenido de cada zona se utilizará el subelemento `<put>` de `<definition>`, el cual dispone a su vez de los atributos:

- **name.** Nombre de la zona para la que se va a indicar el contenido.
- **value.** En el caso de las zonas definidas con `<tiles:insert>` este atributo indicará la URL relativa de la página de contenido, mientras que para las zonas `<tiles:getAsString>` contendrá la cadena de texto equivalente.

7.3.1.2 DEFINICIONES DERIVADAS

A la hora de crear una definición derivada de otra debemos especificar en el atributo *extends* del elemento `<definition>` el nombre de la definición base. De esta manera, el elemento heredará todas las definiciones `<put>` especificadas en el `<definition>` padre. Si queremos redefinir el contenido de cualquiera de las zonas utilizaremos nuevamente el elemento `<put>`.

El siguiente listado corresponde a las tres definiciones que utilizaremos para implementar nuevamente las páginas del ejemplo presentado en la sección anterior, la primera de ellas corresponde a la definición base en la que se indican los elementos comunes a las dos páginas, mientras que las otras dos definiciones heredarán a la base, indicándose en ellas los elementos propios de cada página:

```
<tiles-definitions>
  <definition name=".principal"
    path="/WEB-INF/plantillas/plantilla.jsp">
    <put name="titulo" value=""/>
    <put name="encabezado"
      value="/WEB-INF/plantillas/titulo.jsp"/>
    <put name="menu"
      value="/WEB-INF/plantillas/menu.jsp"/>
    <put name="cuerpo" value=""/>
  </definition>
  <definition name=".parapagina1" extends=".principal">
    <put name="titulo" value="Pagina 1"/>
    <put name="cuerpo"
      value="/WEB-INF/plantillas/body1.jsp"/>
  </definition>
  <definition name=".parapagina2" extends=".principal">
    <put name="titulo" value="Pagina 2"/>
```

```
<put name="cuerpo"
      value="/WEB-INF/plantillas/body2.jsp" />
</definition>
</tiles-definitions>
```

7.3.2 Páginas de aplicación

Una vez creadas las definiciones, la implementación de las páginas de aplicación resulta tremendamente sencilla; utilizando el atributo *definition* del elemento `<tiles:insert>` indicaremos el nombre de la definición que queremos aplicar a cada página. El siguiente listado indica como quedaría la implementación de las páginas `pagina1.jsp` y `pagina2.jsp` del ejemplo:

pagina1.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib
      uri=http://struts.apache.org/tags-tiles
      prefix="tiles" %>
<tiles:insert definition=".parapagina1"/>
```

pagina2.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib
      uri=http://struts.apache.org/tags-tiles
      prefix="tiles" %>
<tiles:insert definition=".parapagina2"/>
```

PRÁCTICA 7.1. APLICACIÓN PARA EL ENVÍO Y VISUALIZACIÓN DE MENSAJES

Descripción

Se trata de realizar una nueva versión de la aplicación desarrollada en la práctica 5.2 para envío y visualización de mensajes, utilizando plantillas para la creación de un aspecto común en todas la página de la aplicación y definiciones para la reutilización de elementos comunes en las vistas.

El nuevo aspecto de las páginas de la aplicación es el que se muestra en la figura 34.

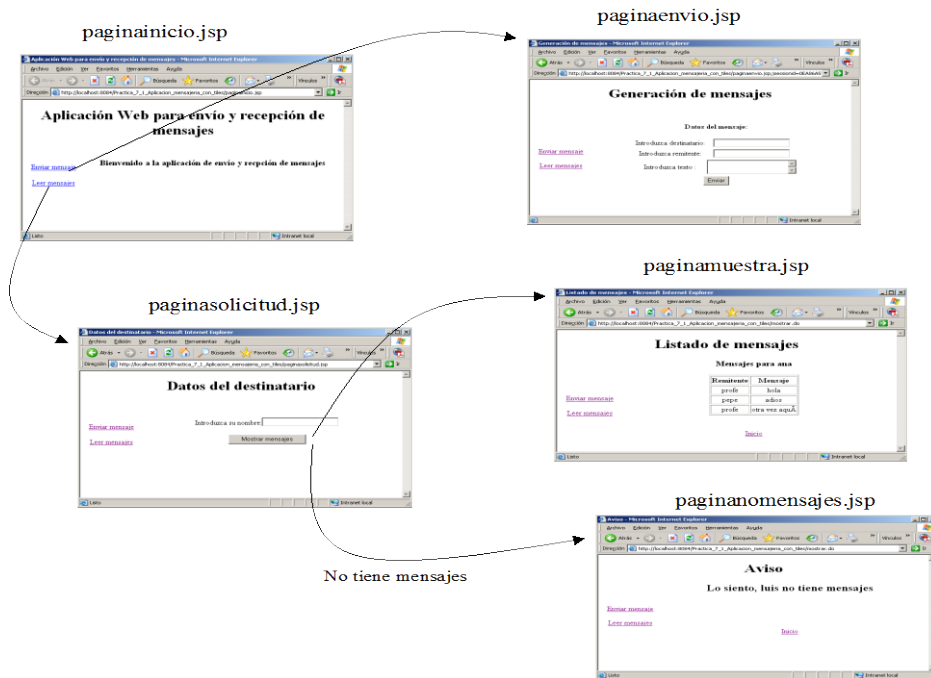


Fig. 34. Páginas de la aplicación

Desarrollo

En una página maestra (master.jsp) se creará la estructura de las páginas de la aplicación, la cual estará formada por una zona superior, un menú lateral izquierdo y un cuerpo central.

El menú será común a todas las páginas y estará formado por una página JSP (menu.jsp) con los dos enlaces que se muestran en la figura. La zona superior estará formada por un elemento de texto, mientras que el cuerpo será ocupado por una página JSP que será diferente para cada página. Tanto la página maestra como las páginas de contenido se colocarán en un directorio interno en WEB-INF.

En el archivo tiles-defs.xml se indicarán las definiciones. Habrá una principal de la que hereden otras cinco, una para cada página. De esta manera, las páginas de la aplicación sólo incluirán una referencia a la definición a aplicar.

Listado

Mostraremos únicamente las páginas JSP y los archivos de configuración, puesto que las clases controlador y de negocio no sufren ninguna alteración respecto a la versión presentada en la práctica 5.2.

struts-config.xml

En fondo sombreado se muestran las modificaciones realizadas para apuntar a las nuevas páginas, así como los nuevos elementos incluidos para permitir la utilización de *tiles*:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts-config PUBLIC
    "-//Apache Software Foundation//DTD Struts
    Configuration 1.2//EN"
    "http://jakarta.apache.org/struts/dtds/struts-
    config_1_2.dtd">
<struts-config>
  <form-beans>
    <form-bean name="MensajeForm"
      type="javabeans.MensajeForm"/>
  </form-beans>
  <global-forwards>
    <forward name="previoenvio"
      path="/paginaenvio.jsp"/>
    <forward name="previolectura"
      path="/paginasolicitud.jsp"/>
    <forward name="inicio"
      path="/paginainicio.jsp"/>
  </global-forwards>
  <action-mappings>
    <action name="MensajeForm" path="/grabar"
      scope="request" type="servlets.EnviarAction" >
      <forward name="grabado"
        path="/paginainicio.jsp"/>
    </action>
    <action path="/mostrar"
      type="servlets.RecuperarAction">
      <forward name="visualiza"
        path="/paginamuestra.jsp"/>
      <forward name="sinmensajes"
```

```

        path="/paginanomensajes.jsp"/>
    </action>
</action-mappings>
<controller processorClass="org.apache.struts.
    tiles.TilesRequestProcessor"/>
<plug-in className="org.apache.struts.
    tiles.TilesPlugin" >
    <set-property property="definitions-config"
        value="/WEB-INF/tiles-defs.xml" />
    <set-property property="moduleAware"
        value="true" />
</plug-in>
</struts-config>

```

tiles-defs.xml

```

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

<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles
        Configuration 1.1//EN"
    "http://jakarta.apache.org/struts/dtds/tiles-
        config_1_1.dtd">

<tiles-definitions>
    <definition name=".principal"
        path="/WEB-INF/plantillas/master.jsp">
        <put name="titulo" value="Aplicación Web para envío y
            recepción de mensajes"/>
        <put name="menu"
            value="/WEB-INF/plantillas/menu.jsp"/>
        <put name="cuerpo" value=""/>
    </definition>
    <definition name=".inicio" extends=".principal">
        <put name="cuerpo"
            value="/WEB-INF/plantillas/bodyinicio.jsp"/>
    </definition>
    <definition name=".solicitud" extends=".principal">
        <put name="titulo" value="Datos del destinatario"/>
        <put name="cuerpo"
            value="/WEB-INF/plantillas/bodysolicitud.jsp"/>
    </definition>

```

```

<definition name=".envio" extends=".principal">
  <put name="titulo" value="Generación de mensajes"/>
  <put name="cuerpo"
    value="/WEB-INF/plantillas/bodyenvio.jsp"/>
</definition>
<definition name=".muestra" extends=".principal">
  <put name="titulo" value="Listado de mensajes"/>
  <put name="cuerpo"
    value="/WEB-INF/plantillas/bodymuestra.jsp"/>
</definition>
<definition name=".nomensajes" extends=".principal">
  <put name="titulo" value="Aviso"/>
  <put name="cuerpo"
    value="/WEB-INF/plantillas/bodynomensajes.jsp"/>
</definition>
</tiles-definitions>

```

Plantillas y páginas de contenido

master.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
  prefix="tiles" %>
<%@ taglib uri="http://struts.apache.org/tags-logic"
  prefix="logic" %>
<%@ taglib uri="http://struts.apache.org/tags-bean"
  prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html"
  prefix="html" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
  Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html:html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8">
    <title><tiles:getAsString name="titulo"/></title>
  </head>

```



```
<body>
  <table width="100%">
    <tr valign="center">
      <td height="25%" align="center"
        colspan="2">
        <h1>
          <tiles:getAsString name="titulo"/>
        </h1>
      </td>
    </tr>
    <tr>
      <td width="100px">
        <tiles:insert attribute="menu"/>
      </td>
      <td>
        <tiles:insert attribute="cuerpo"/>
      </td>
    </tr>
  </table>
</body>
</html:html>
```

menu.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-html
  prefix="html" %>
<center>
  <br/><br/>
  <html:link forward="previoenvio">
    Enviar mensaje
  </html:link>
  <br/><br/>
  <html:link forward="previolectura">
    Leer mensajes
  </html:link>
</center>
```

bodyinicio.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<center>
    <h3>Bienvenido a la aplicación de envío y
        recpción de mensajes</h3>
</center>
```

bodyenvio.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@ taglib uri=http://struts.apache.org/tags-html
    prefix="html" %>
<center>
    <html:form action="/grabar" method="POST">
    <br/><br/>
    <b>Datos del mensaje:</b><br/><br/>
    <table>
    <tr>
        <td>Introduzca destinatario:</td>
        <td>
            <html:text property="destino"/>
        </td>
    </tr>
    <tr>
        <td>Introduzca remitente:</td>
        <td>
            <html:text property="remite"/>
        </td>
    </tr>
    <tr>
        <td>Introduzca texto : </td>
        <td>
            <html:textarea property="texto">
            </html:textarea>
        </td>
    </tr>
</center>
```

```

        <tr>
            <td colspan="2"><html:submit value="Enviar"/></td>
        </tr>
    </table>
</html:form>
</center>
</body>

```

bodymuestra.jsp

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javabeans.*,java.util.*"%>
<%@ taglib uri=http://struts.apache.org/tags-logic
    prefix="logic" %>

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

<center>
<bean:parameter id="nombre" name="nombre"/>
<h3>
Mensajes para <bean:write name="nombre"/>
</h3>
<table border="1">
    <tr>
        <th>Remitente</th>
        <th>Mensaje</th>
    </tr>
    <logic:iterate id="mensaje" name="mensajes"
        scope="request">
        <tr>
            <td>
                <bean:write name="mensaje"
                    property="remite"/>
            </td>
            <td>
                <bean:write name="mensaje"
                    property="texto"/>
            </td>
        </tr>
    </logic:iterate>

```

```
        </logic:iterate>
    </table>
    <br/><br/>
    <html:link forward="inicio">Inicio</html:link>
</center>
```

bodysolicitud.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<center>
    <br/><br/>
    <form action="mostrar.do" method="post">
        Introduzca su nombre:
        <input type="text" name="nombre"><br><br>
        <input type="submit" value="Mostrar mensajes">
    </form>
</center>
```

bodynomensajes.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ page import="javabeans.*, java.util.*"%>
<%@ taglib uri="http://struts.apache.org/tags-bean"
    prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html"
    prefix="html" %>
<bean:parameter id="nombre" name="nombre"/>
<center>
    <h2>
        Lo siento, <bean:write name="nombre"/> no tiene mensajes
    </h2>
    <br/><br/><br/><br/>
    <html:link forward="inicio">
        Inicio
    </html:link>
</center>
```

Páginas de aplicación

paginainicio.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
    prefix="tiles" %>
<tiles:insert definition=".inicio"/>
```

paginaenvio.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
    prefix="tiles" %>
<tiles:insert definition=".envio"/>
```

paginamuestra.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
    prefix="tiles" %>
<tiles:insert definition=".muestra"/>
```

paginasolicitud.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
    prefix="tiles" %>
<tiles:insert definition=".solicitud"/>
```

paginanomensajes.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib uri=http://struts.apache.org/tags-tiles
    prefix="tiles" %>
<tiles:insert definition=".nomensajes"/>
```


STRUTS 2

Las cada vez más exigentes demandas de las aplicaciones Web modernas han llevado al equipo de desarrollo de Struts, en su afán de mejorar continuamente las prestaciones del framework, a replantear la estructura y composición del mismo, adoptando un nuevo enfoque que resulte más flexible y sencillo de utilizar. En esta línea nace Struts 2.

Más que una nueva versión de Struts, Struts 2 es realmente la fusión de dos frameworks: por un lado el clásico Struts (conocido también a partir de ahora como Struts 1) con nuevas capacidades añadidas, y por otro WebWork (o mejor dicho WebWork 2, la versión actualizada y mejorada de éste), un framework desarrollado por OpenSymphony que proporciona un fuerte desacoplamiento entre las capas de la aplicación.

El resultado es un producto que integra las capacidades de ambos frameworks, lo que sin duda hace de Struts 2 el marco de trabajo más atractivo y potente para la creación de aplicaciones Web con arquitectura MVC.

Lo anterior no significa que Struts 1 haya entrado en vías de desaparición, ni mucho menos. Struts 1 es un framework muy consolidado, integrado en la mayoría de los entornos de desarrollo más importantes y utilizado por una amplísima comunidad de desarrolladores, mientras que Struts 2 tiene que madurar mucho aún, si bien tiene un gran futuro por delante.

8.1 COMPONENTES DE STRUTS 2

Como decimos, Struts 2 sigue siendo fiel a la arquitectura MVC, si bien los roles que desempeñan algunos de los componentes tradicionales cambian con respecto a las versiones 1.x.

La figura 35 muestra los principales componentes de Struts 2, su ubicación dentro de las distintas capas de la aplicación y la interacción entre los mismos.

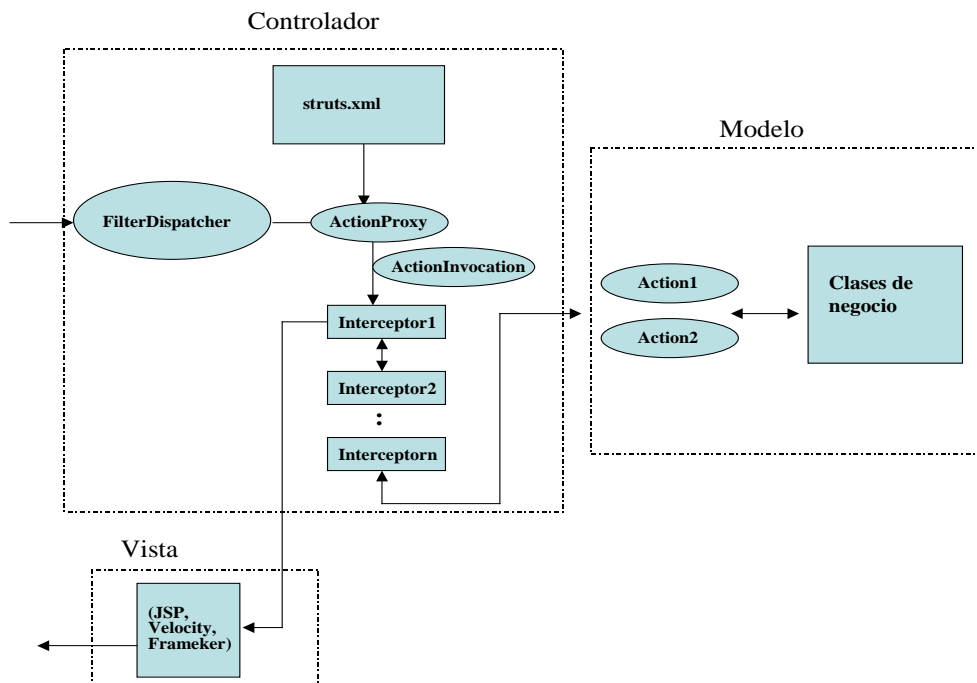


Fig. 35. Componentes y capas de una aplicación Struts 2

A continuación estudiaremos el funcionamiento y principales características de estos componentes, analizando su comportamiento a lo largo del ciclo de vida de la petición desde que ésta llega al Controlador hasta que se envía la respuesta al cliente.

8.1.1 FilterDispatcher

Este componente representa el punto de entrada a la aplicación, dirigiéndose a él todas las peticiones que llegan desde el cliente. FilterDispatcher hace en Struts 2 las veces de ActionServlet en Struts 1, analizando la petición

recibida y determinando con el apoyo de otros objetos auxiliares y de la información almacenada en el archivo de configuración `struts.xml` el tipo de acción a ejecutar.

`FilterDispatcher` forma parte del API de Struts 2, concretamente, se incluye dentro del paquete `org.apache.struts2.dispatcher`, y, como se deduce de su propio nombre, es implementado mediante un filtro, por lo que debe ser registrado en el archivo `web.xml` de la aplicación tal y como se indica en el siguiente listado:

```
:
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.
        dispatcher.FilterDispatcher
    </filter-class>
</filter>

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

8.1.2 Interceptores

Una vez determinada la acción a ejecutar, `FilterDispatcher` pasa el control de la petición a un objeto intermediario de tipo `ActionProxy`, éste crea un objeto `ActionInvocation` en el que almacena la información de la petición entrante, pasando a continuación el control de la misma a los interceptores.

Los interceptores constituyen una cadena de objetos que realizan una serie de tareas de pre-procesamiento previas antes de ejecutar la acción, como por ejemplo la validación de datos de usuario, rellenado de objetos con los campos de un formulario, etc., así como un post-procesamiento de los resultados una vez que éstos ha sido generados por las clases de acción.

El funcionamiento de los interceptores es muy similar a los filtros servlet, ejecutándose en cadena al recibir la petición, en el orden indicado en el archivo de configuración `struts.xml` y en orden inverso durante el envío de la respuesta al cliente.

El API de Struts incorpora un gran número de interceptores, pudiendo decidir el programador cuáles de ellos deben ejecutarse para cada acción simplemente indicándolo en el **archivo de configuración struts.xml**. Algunos de estos interceptores proporcionan a las clases de acción acceso a los distintos objetos del API Servlet, a fin de que éstas puedan controlar los datos manejados en la aplicación.

Como veremos más adelante, un programador también puede implementar sus propios interceptores para realizar algún tipo de pre-procesamiento o post-procesamiento personalizado que no esté contemplado por ninguno de los interceptores del API. Para este propósito Struts 2 proporciona la interfaz `Interceptor`.

8.1.3 Action

Como sucede en Struts 1, los objetos `Action` se encargan del procesamiento de las peticiones del cliente. Sin embargo, en Struts 2 este tipo de objetos presenta notables diferencias respecto a las versiones anteriores; analizaremos las más significativas:

- **Los objetos `Action` forman parte del modelo.** Esto se puede apreciar en el esquema que hemos presentado en la figura 35. Aunque por regla general, al igual que sucedía en Struts 1.x, la lógica de negocio se suele aislar en clases independientes o EJB, incluyéndose en las clases de acción las llamadas a los métodos expuestos por estos objetos.
- **Implementación como clases `POJO`.** La principal característica de las clases de acción de Struts 2 es que **no tienen que heredar ni implementar ninguna clase o interfaz del API**. Son clases estándares Java, comúnmente conocidas también como *Plain Old Java Objects (POJO)* y cuyo único requerimiento es tener que proporcionar un método, que por convenio es llamado `execute()` al igual que las clases de acción utilizadas en versiones anteriores del framework, en el que se deberán incluir las instrucciones a ejecutar para el procesamiento de la acción. Este método será invocado por el último interceptor de la cadena.
- **Inclusión de métodos `set/get`.** Otra de las características de Struts 2 es que se ha **eliminado la utilización de clases de tipo `ActionForm`** para la encapsulación de datos procedentes de un formulario cliente. En su defecto, estos datos son **capturados por**

el propio objeto de acción invocado desde el Controlador, operación que es realizada con ayuda de uno de los interceptores incluidos en el API de Struts 2, resultando así transparente para el programador. Lo único que en este sentido habrá que codificar en las clases de acción (además del método *execute()* para el tratamiento de la petición) serán los datos miembro para el almacenamiento de los campos con sus correspondientes métodos *set/get*, de forma similar a como se hacía en las clases *ActionForm* de Struts 1.

Este nuevo sistema permite simplificar doblemente el desarrollo de la aplicación, ya que por un lado se reduce el número de tipos distintos de clases a implementar y, por otro, se dispone directamente de los datos en la misma clase donde van a ser manipulados.

En caso de que queramos encapsular los datos en objetos para facilitar su transporte entre las clases *Action* y los componentes que encapsulan la lógica de negocio, se utilizarán *JavaBeans* estándares.

8.1.4 Librerías de acciones

Como en Struts 1, Struts 2 incluye un amplio conjunto de acciones o tags que facilitan la creación de las vistas. Estas acciones se incluyen en la librería de *uri* asociada “/struts-tags”.

Como novedad respecto a las versiones anteriores, estas acciones pueden utilizarse no sólo en la construcción de páginas JSP, también con otras tecnologías para la creación de vistas como son las plantillas *velocity* o *freemaker*.

Además de incluir una gran variedad de componentes gráficos y elementos de control de flujo, esta librería proporciona **acciones que permiten acceder directamente desde la vista a la acción que se acaba de ejecutar**, facilitándose así el acceso a los datos generados por el modelo.

8.1.5 Archivo de configuración *struts.xml*

Las aplicaciones Struts 2 utilizan un archivo de configuración llamado *struts.xml* en el que se registran y configuran los distintos componentes de la aplicación, realizándose estas tareas de una manera más simple que en *struts-config.xml* de Struts 1.

Entre otros elementos, en `struts.xml` debemos registrar los Action, con sus correspondientes reglas de navegación, y los interceptores.

Struts 2 soporta además la **herencia de archivos de configuración**, lo que se traduce en poder utilizar una serie de configuraciones por defecto en las aplicaciones sin tener que reescribirlas de nuevo en cada `struts.xml` particular.

La figura 36 nos muestra la estructura básica de un archivo `struts.xml`. Este fichero **debe residir en un directorio que esté incluido en el *classpath* de la aplicación**, habitualmente `raiz_aplicacion\WEB-INF\classes`.

```
<?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>
    <package ....>
        <interceptors>
            <!--definición de interceptores personalizados-->
        </interceptors>
        <action ...>
            <!--configuración de elementos para la acción-->
        </action>
        :
    </package>
    :
</struts>
```

Fig. 36. Archivo de configuración `struts.xml`

8.1.5.1 PAQUETES

Cada configuración definida dentro de `struts.xml` se incluye dentro de un paquete, el cual viene definido por el elemento `<package>`. Los paquetes permiten agrupar un conjunto de elementos (habitualmente acciones) que comparten una serie de atributos de configuración comunes.

Un elemento `<package>` puede incluir los siguientes atributos:

- **name.** Nombre o identificador asociado y que debe ser único para cada paquete.
- **namespace.** Proporciona un mapeo desde la URL al paquete. Por ejemplo, si al atributo *namespace* de un determinado paquete se le ha asignado el valor “packexample”, la URL para acceder a las acciones definidas en el interior del mismo será: `raiz_aplicacion\packexample\nombreaccion.action`.
- **extends.** Nombre del paquete heredado. Como hemos indicado anteriormente, Struts 2 permite la herencia entre archivos de configuración, concretamente, la herencia de paquetes.

8.1.5.2 HERENCIA DE PAQUETES

Struts 2 proporciona un archivo de configuración por defecto llamado `struts-default.xml`, en el que se incluye un paquete de nombre “struts-default” con una serie de configuraciones predefinidas disponibles para ser utilizadas en cualquier aplicación. Entre otras cosas, este paquete configura una serie de interceptores predefinidos de Struts 2, por lo que si queremos disponer de la funcionalidad de los mismos en nuestra aplicación, será conveniente crear paquetes que hereden a éste:

```
<package name="paquetel"
    namespace="packexample"
    extends="struts-default">
:
</package>
```

A modo de ejemplo, en el siguiente listado se muestra el contenido del archivo de configuración por defecto de Struts 2 `struts-default.xml`, el cual se encuentra incluido en la librería `struts2-core-2.0.11.jar` suministrada con el paquete de distribución de Struts 2:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
/*
 * $Id: struts-default.xml 559615 2007-07-25 21:25:25Z apetrelli $
 *
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership. The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License. You may obtain a copy of the License at
 *

```

```

* http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*/
-->

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

<struts>
    <bean class="com.opensymphony.xwork2.ObjectFactory"
        name="xwork" />
    <bean type="com.opensymphony.xwork2.ObjectFactory" name="struts"
class="org.apache.struts2.impl.StrutsObjectFactory" />

    <bean type="com.opensymphony.xwork2.ActionProxyFactory"
name="xwork"
class="com.opensymphony.xwork2.DefaultActionProxyFactory" />
    <bean type="com.opensymphony.xwork2.ActionProxyFactory"
name="struts"
class="org.apache.struts2.impl.StrutsActionProxyFactory" />

    <bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer"
name="tiger"
class="com.opensymphony.xwork2.util.GenericObjectTypeDeterminer" />
    <bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer"
name="notiger"
class="com.opensymphony.xwork2.util.DefaultObjectTypeDeterminer" />
    <bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer"
name="struts"
class="com.opensymphony.xwork2.util.DefaultObjectTypeDeterminer" />

    <bean type="org.apache.struts2.dispatcher.mapper.ActionMapper"
name="struts"
class="org.apache.struts2.dispatcher.mapper.DefaultActionMapper" />
    <bean type="org.apache.struts2.dispatcher.mapper.ActionMapper"
name="composite"
class="org.apache.struts2.dispatcher.mapper.CompositeActionMapper"
/>
    <bean type="org.apache.struts2.dispatcher.mapper.ActionMapper"
name="restful"
class="org.apache.struts2.dispatcher.mapper.RestfulActionMapper" />
    <bean type="org.apache.struts2.dispatcher.mapper.ActionMapper"
name="restful2"
class="org.apache.struts2.dispatcher.mapper.Restful2ActionMapper" />

    <bean
type="org.apache.struts2.dispatcher.multipart.MultiPartRequest"
name="struts"
class="org.apache.struts2.dispatcher.multipart.JakartaMultiPartReque
st" scope="default" optional="true"/>

```

```

    <bean
type="org.apache.struts2.dispatcher.multipart.MultiPartRequest"
name="jakarta"
class="org.apache.struts2.dispatcher.multipart.JakartaMultiPartReque
st" scope="default" optional="true" />

    <bean type="org.apache.struts2.views.TagLibrary" name="s"
class="org.apache.struts2.views.DefaultTagLibrary" />

    <bean
class="org.apache.struts2.views.freemarker.FreemarkerManager"
name="struts" optional="true" />
    <bean class="org.apache.struts2.views.velocity.VelocityManager"
name="struts" optional="true" />

    <bean
class="org.apache.struts2.components.template.TemplateEngineManager"
/>
    <bean
type="org.apache.struts2.components.template.TemplateEngine"
name="ftl"
class="org.apache.struts2.components.template.FreemarkerTemplateEngi
ne" />
    <bean
type="org.apache.struts2.components.template.TemplateEngine"
name="vm"
class="org.apache.struts2.components.template.VelocityTemplateEngine"
/>
    <bean
type="org.apache.struts2.components.template.TemplateEngine"
name="jsp"
class="org.apache.struts2.components.template.JspTemplateEngine" />

    <bean type="com.opensymphony.xwork2.util.XWorkConverter"
name="xwork1" class="com.opensymphony.xwork2.util.XWorkConverter" />
    <bean type="com.opensymphony.xwork2.util.XWorkConverter"
name="struts"
class="com.opensymphony.xwork2.util.AnnotationXWorkConverter" />
    <bean type="com.opensymphony.xwork2.TextProvider" name="xwork1"
class="com.opensymphony.xwork2.TextProviderSupport" />
    <bean type="com.opensymphony.xwork2.TextProvider" name="struts"
class="com.opensymphony.xwork2.TextProviderSupport" />

    <!-- Only have static injections -->
    <bean class="com.opensymphony.xwork2.ObjectFactory"
static="true" />
    <bean class="com.opensymphony.xwork2.util.XWorkConverter"
static="true" />
    <bean class="com.opensymphony.xwork2.util.OgnlValueStack"
static="true" />
    <bean class="org.apache.struts2.dispatcher.Dispatcher"
static="true" />
    <bean class="org.apache.struts2.components.Include"
static="true" />
    <bean class="org.apache.struts2.dispatcher.FilterDispatcher"
static="true" />
    <bean class="org.apache.struts2.views.util.ContextUtil"
static="true" />

```

```

    <bean class="org.apache.struts2.views.util.UrlHelper"
static="true" />

    <package name="struts-default" abstract="true">
        <result-types>
            <result-type name="chain"
class="com.opensymphony.xwork2.ActionChainResult" />
            <result-type name="dispatcher"
class="org.apache.struts2.dispatcher.ServletDispatcherResult"
default="true" />
            <result-type name="freemarker"
class="org.apache.struts2.views.freemarker.FreemarkerResult" />
            <result-type name="httpheader"
class="org.apache.struts2.dispatcher.HttpHeaderResult" />
            <result-type name="redirect"
class="org.apache.struts2.dispatcher.ServletRedirectResult" />
            <result-type name="redirectAction"
class="org.apache.struts2.dispatcher.ServletActionRedirectResult" />
            <result-type name="stream"
class="org.apache.struts2.dispatcher.StreamResult" />
            <result-type name="velocity"
class="org.apache.struts2.dispatcher.VelocityResult" />
            <result-type name="xslt"
class="org.apache.struts2.views.xslt.XSLTResult" />
            <result-type name="plainText"
class="org.apache.struts2.dispatcher.PlainTextResult" />
            <!-- Deprecated name form scheduled for removal in
Struts 2.1.0. The camelCase versions are preferred. See ww-1707 -->
            <result-type name="redirect-action"
class="org.apache.struts2.dispatcher.ServletActionRedirectResult" />
            <result-type name="plaintext"
class="org.apache.struts2.dispatcher.PlainTextResult" />
        </result-types>

        <interceptors>
            <interceptor name="alias"
class="com.opensymphony.xwork2.interceptor.AliasInterceptor" />
            <interceptor name="autowiring"
class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringIn
terceptor" />
            <interceptor name="chain"
class="com.opensymphony.xwork2.interceptor.ChainingInterceptor" />
            <interceptor name="conversionError"
class="org.apache.struts2.interceptor.StrutsConversionErrorIntercept
or" />
            <interceptor name="cookie"
class="org.apache.struts2.interceptor.CookieInterceptor" />
            <interceptor name="createSession"
class="org.apache.struts2.interceptor.CreateSessionInterceptor" />
            <interceptor name="debugging"
class="org.apache.struts2.interceptor.debugging.DebuggingIntercepto
r" />
            <interceptor name="externalRef"
class="com.opensymphony.xwork2.interceptor.ExternalReferencesInterce
ptor" />
            <interceptor name="execAndWait"
class="org.apache.struts2.interceptor.ExecuteAndWaitInterceptor" />

```



```

        <interceptor name="exception"
class="com.opensymphony.xwork2.interceptor.ExceptionMappingIntercept
or"/>
        <interceptor name="fileUpload"
class="org.apache.struts2.interceptor.FileUploadInterceptor"/>
        <interceptor name="i18n"
class="com.opensymphony.xwork2.interceptor.I18nInterceptor"/>
        <interceptor name="logger"
class="com.opensymphony.xwork2.interceptor.LoggingInterceptor"/>
        <interceptor name="modelDriven"
class="com.opensymphony.xwork2.interceptor.ModelDrivenInterceptor"/>
        <interceptor name="scopedModelDriven"
class="com.opensymphony.xwork2.interceptor.ScopedModelDrivenIntercep
tor"/>
        <interceptor name="params"
class="com.opensymphony.xwork2.interceptor.ParametersInterceptor"/>
        <interceptor name="prepare"
class="com.opensymphony.xwork2.interceptor.PrepareInterceptor"/>
        <interceptor name="staticParams"
class="com.opensymphony.xwork2.interceptor.StaticParametersIntercept
or"/>
        <interceptor name="scope"
class="org.apache.struts2.interceptor.ScopeInterceptor"/>
        <interceptor name="servletConfig"
class="org.apache.struts2.interceptor.ServletConfigInterceptor"/>
        <interceptor name="sessionAutowiring"
class="org.apache.struts2.spring.interceptor.SessionContextAutowirin
gInterceptor"/>
        <interceptor name="timer"
class="com.opensymphony.xwork2.interceptor.TimerInterceptor"/>
        <interceptor name="token"
class="org.apache.struts2.interceptor.TokenInterceptor"/>
        <interceptor name="tokenSession"
class="org.apache.struts2.interceptor.TokenSessionStoreInterceptor"/
>
        <interceptor name="validation"
class="org.apache.struts2.interceptor.validation.AnnotationValidatio
nInterceptor"/>
        <interceptor name="workflow"
class="com.opensymphony.xwork2.interceptor.DefaultWorkflowIntercepto
r"/>
        <interceptor name="store"
class="org.apache.struts2.interceptor.MessageStoreInterceptor"/>
        <interceptor name="checkbox"
class="org.apache.struts2.interceptor.CheckboxInterceptor"/>
        <interceptor name="profiling"
class="org.apache.struts2.interceptor.ProfilingActivationInterceptor
"/>
        <interceptor name="roles"
class="org.apache.struts2.interceptor.RolesInterceptor"/>

<!-- Basic stack -->
<interceptor-stack name="basicStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="servletConfig"/>
    <interceptor-ref name="prepare"/>
    <interceptor-ref name="checkbox"/>
    <interceptor-ref name="params"/>
    <interceptor-ref name="conversionError"/>

```

```

</interceptor-stack>

<!-- Sample validation and workflow stack -->
<interceptor-stack name="validationWorkflowStack">
    <interceptor-ref name="basicStack"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="workflow"/>
</interceptor-stack>

<!-- Sample file upload stack -->
<interceptor-stack name="fileUploadStack">
    <interceptor-ref name="fileUpload"/>
    <interceptor-ref name="basicStack"/>
</interceptor-stack>

<!-- Sample model-driven stack -->
<interceptor-stack name="modelDrivenStack">
    <interceptor-ref name="modelDriven"/>
    <interceptor-ref name="basicStack"/>
</interceptor-stack>

<!-- Sample action chaining stack -->
<interceptor-stack name="chainStack">
    <interceptor-ref name="chain"/>
    <interceptor-ref name="basicStack"/>
</interceptor-stack>

<!-- Sample i18n stack -->
<interceptor-stack name="i18nStack">
    <interceptor-ref name="i18n"/>
    <interceptor-ref name="basicStack"/>
</interceptor-stack>

<!-- An example of the params-prepare-params trick. This
stack
that it
interceptor:
parameters directly
as a DAO
object
loading
in the
interceptor to
is exactly the same as the defaultStack, except
includes one extra interceptor before the prepare
the params interceptor.

This is useful for when you wish to apply
to an object that you wish to load externally (such
or database or service layer), but can't load that
until at least the ID parameter has been loaded. By
the parameters twice, you can retrieve the object
prepare() method, allowing the second params
apply the values on the object. -->
<interceptor-stack name="paramsPrepareParamsStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="alias"/>
    <interceptor-ref name="params"/>
    <interceptor-ref name="servletConfig"/>

```

```

        <interceptor-ref name="prepare"/>
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="chain"/>
        <interceptor-ref name="modelDriven"/>
        <interceptor-ref name="fileUpload"/>
        <interceptor-ref name="checkbox"/>
        <interceptor-ref name="staticParams"/>
        <interceptor-ref name="params"/>
        <interceptor-ref name="conversionError"/>
        <interceptor-ref name="validation">
            <param
name="excludeMethods">input,back,cancel</param>
        </interceptor-ref>
        <interceptor-ref name="workflow">
            <param
name="excludeMethods">input,back,cancel</param>
        </interceptor-ref>
    </interceptor-stack>

    <!-- A complete stack with all the common interceptors
in place.
    Generally, this stack should be the one you use,
though it
    may do more than you need. Also, the ordering can
be
    switched around (ex: if you wish to have your
servlet-related
    objects applied before prepare() is called, you'd
need to move
    servlet-config interceptor up.

    This stack also excludes from the normal validation
and workflow
    the method names input, back, and cancel. These
typically are
    associated with requests that should not be
validated.

-->
    <interceptor-stack name="defaultStack">
        <interceptor-ref name="exception"/>
        <interceptor-ref name="alias"/>
        <interceptor-ref name="servletConfig"/>
        <interceptor-ref name="prepare"/>
        <interceptor-ref name="i18n"/>
        <interceptor-ref name="chain"/>
        <interceptor-ref name="debugging"/>
        <interceptor-ref name="profiling"/>
        <interceptor-ref name="scopedModelDriven"/>
        <interceptor-ref name="modelDriven"/>
        <interceptor-ref name="fileUpload"/>
        <interceptor-ref name="checkbox"/>
        <interceptor-ref name="staticParams"/>
        <interceptor-ref name="params">
            <param name="excludeParams">dojo\..*</param>
        </interceptor-ref>
        <interceptor-ref name="conversionError"/>
        <interceptor-ref name="validation">
            <param
name="excludeMethods">input,back,cancel,browse</param>

```

```

        </interceptor-ref>
        <interceptor-ref name="workflow">
            <param
name="excludeMethods">input,back,cancel,browse</param>
        </interceptor-ref>
    </interceptor-stack>

    <!-- The completeStack is here for backwards
compatibility for
    applications that still refer to the defaultStack
by the
        old name -->
    <interceptor-stack name="completeStack">
        <interceptor-ref name="defaultStack"/>
    </interceptor-stack>

    <!-- Sample execute and wait stack.
        Note: execAndWait should always be
        the *last* interceptor. -->
    <interceptor-stack name="executeAndWaitStack">
        <interceptor-ref name="execAndWait">
            <param name="excludeMethods">
                input,back,cancel</param>
        </interceptor-ref>
        <interceptor-ref name="defaultStack"/>
        <interceptor-ref name="execAndWait">
            <param name="excludeMethods">
                input,back,cancel</param>
        </interceptor-ref>
    </interceptor-stack>

    <!-- Deprecated name forms scheduled for removal in
        Struts 2.1.0. The camelCase
        versions are preferred. See ww-1707 -->
    <interceptor name="external-ref"
        class="com.opensymphony.xwork2.
            interceptor.ExternalReferencesInterceptor"/>
    <interceptor name="model-driven"
        class="com.opensymphony.xwork2.
            interceptor.ModelDrivenInterceptor"/>
    <interceptor name="static-params"
        class="com.opensymphony.xwork2.
            interceptor.StaticParametersInterceptor"/>
    <interceptor name="scoped-model-driven"
        class="com.opensymphony.xwork2.
            interceptor.ScopedModelDrivenInterceptor"/>
    <interceptor name="servlet-config"
        class="org.apache.struts2.
            interceptor.ServletConfigInterceptor"/>
    <interceptor name="token-session"
class="org.apache.struts2.interceptor.TokenSessionStoreInterceptor"/
>

    </interceptors>

    <default-interceptor-ref name="defaultStack"/>
</package>

</struts>

```

8.1.5.3 MODULARIDAD DE FICHEROS DE CONFIGURACIÓN

En una aplicación grande con un elevado número de elementos de configuración, puede ser conveniente distribuir estas configuraciones en archivos diferentes. Desde el interior de `struts.xml` se incluirá una referencia a cada uno de estos archivos mediante el elemento `<include>`, el cual indicará en su atributo *file* la ruta relativa del archivo a incluir.

El siguiente ejemplo distribuye la configuración de la aplicación en tres archivos `.xml`, incluyendo cada uno de ellos la configuración de cada rol que opera en la aplicación:

```
<struts>
    <include file="administrador.xml"/>
    <include file="tutor.xml"/>
    <include file="alumno.xml"/>
</struts>
```

Cada archivo incluido tendrá **exactamente la misma estructura** que cualquier archivo de configuración `struts.xml`.

8.2 BENEFICIOS DEL USO DE STRUTS 2

Del análisis que hemos ido realizando sobre los componentes de este nuevo framework se deducen ya algunos de los beneficios que su uso aporta a los programadores de aplicaciones Web. A continuación resumimos las principales características clave que aporta este nuevo framework:

- **Acciones de tipo POJO.** La posibilidad de utilizar clases simples para implementar las acciones reduce la complejidad del desarrollo de las mismas y permite un desacoplamiento entre capas.
- **Eliminación de ActionForms.** Las clases de acción permiten disponer de propiedades para almacenar los datos de usuario, haciendo innecesario el empleo de un nuevo tipo de clase adicional.
- **Flexibilidad en el diseño de las vistas.** La posibilidad de utilizar distintas tecnologías para la generación de una vista en aplicaciones Struts 2, tales como JSP, velocity o freemaker, proporciona una gran flexibilidad en el diseño de las mismas y ofrece un gran variedad de posibilidades a los desarrolladores.

- **Amplia librería de acciones.** El número de acciones personalizadas para la generación de la vista y las capacidades de éstas han aumentado respecto a versiones anteriores del framework.
- **Posibilidad de utilizar anotaciones.** Como alternativa al uso de ficheros XML, Struts 2 soporta el uso de anotaciones para definir diferentes parámetros de configuración en las acciones, reduciendo la complejidad de la aplicación.
- **Configuraciones por defecto.** La posibilidad de disponer de una serie de opciones de configuración predefinidas simplifica enormemente la labor del desarrollador.
- **Soporte para AJAX.** La utilización de AJAX en el desarrollo de aplicaciones para Web ha supuesto una enorme reducción en el tiempo de respuesta de las mismas, mejorando además la experiencia de los usuarios con la interfaz gráfica. Struts 2 proporciona una serie de tags que permiten añadir capacidades AJAX a las aplicaciones.

8.3 CREACIÓN DE UNA APLICACIÓN DE EJEMPLO DE STRUTS 2

Seguidamente se va a explicar cómo crear una sencilla aplicación Web de ejemplo con Struts 2, la cual nos servirá para conocer el funcionamiento del framework y aprender a crear y configurar los elementos principales.

La construcción de la aplicación puede realizarse de forma manual o utilizando algún IDE, como por ejemplo Eclipse o NetBeans.

8.3.1 Descarga del paquete de distribución de Struts 2

Lo primero que tenemos que hacer será descargar la última versión del paquete de distribución de Struts 2 desde la página <http://struts.apache.org/2.x/>. Este paquete de distribución consiste en un archivo .zip en el que se incluyen las librerías con los distintos componentes del framework, documentación del API y una serie de aplicaciones de ejemplo (figura 37).

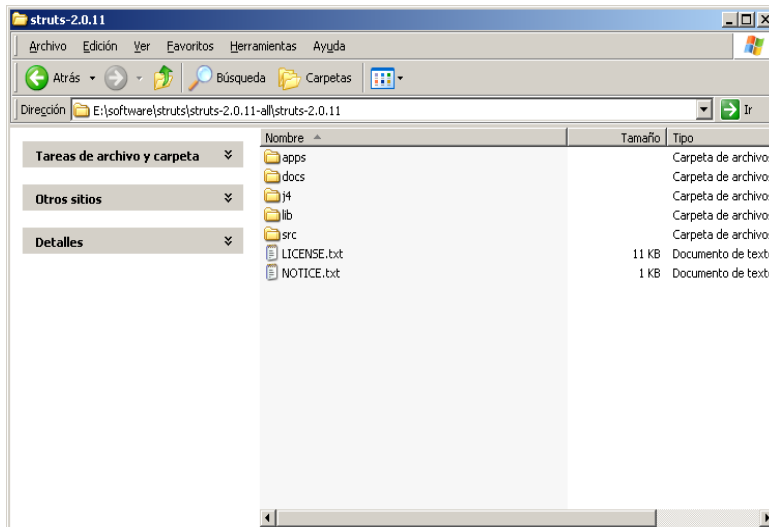


Fig. 37. *Contenido del paquete de distribución de Struts 2*

Si utilizamos un entorno de desarrollo como NetBeans o Eclipse, podemos crear una librería a la que asociaremos los .jar descargados, a fin de poder añadirla fácilmente a un proyecto Web en el que queramos hacer uso de este framework.

8.3.2 Requerimientos software

Las aplicaciones Struts 2 pueden ejecutarse con servidores Tomcat 5.0 en adelante, siendo además necesario disponer de las siguientes versiones Java/JavaEE:

- Servlet API 2.3.
- JSP API 2.0.
- Java 5.

8.3.3 Descripción de la aplicación

La aplicación de ejemplo que vamos a crear va a consistir en una página de inicio en la que aparecerá un enlace que al ser pulsado muestra una página con un mensaje de saludo y la hora del sistema (figura 38).

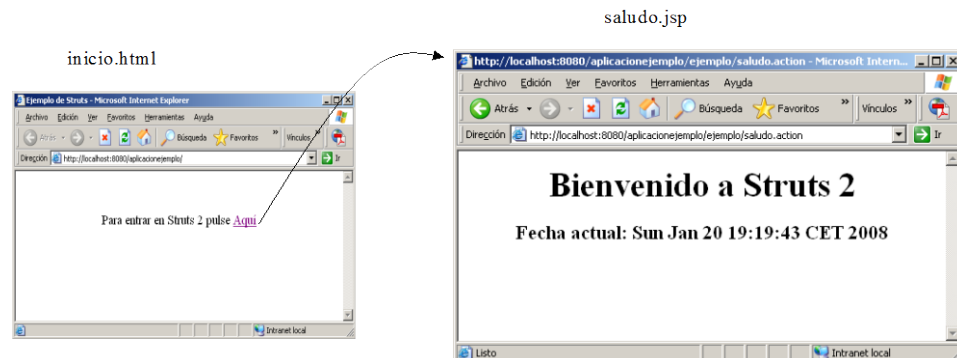


Fig. 38. Páginas de la aplicación de ejemplo

8.3.4 Estructura de directorios de la aplicación

Al tratarse de una aplicación Web, la estructura de directorios de la misma será la correspondiente a cualquier aplicación JavaEE. En este sentido, hemos de tener en cuenta que debemos incluir en el directorio WEB-INF\lib de la aplicación las librerías Struts 2 necesarias para el funcionamiento de la aplicación, que como mínimo deben ser: commons-logging-1.0.4.jar, freemarker-2.3.8.jar, ognl-2.6.11.jar, struts2-core-2.0.11.jar y xwork-2.0.4.jar.

Suponiendo que el directorio raíz de la aplicación lo llamamos “ejemplosaludo” y teniendo en cuenta que la única clase de acción que vamos a utilizar se llamará Saludo, la estructura de directorios de la aplicación quedará pues como se indica en la figura 39.

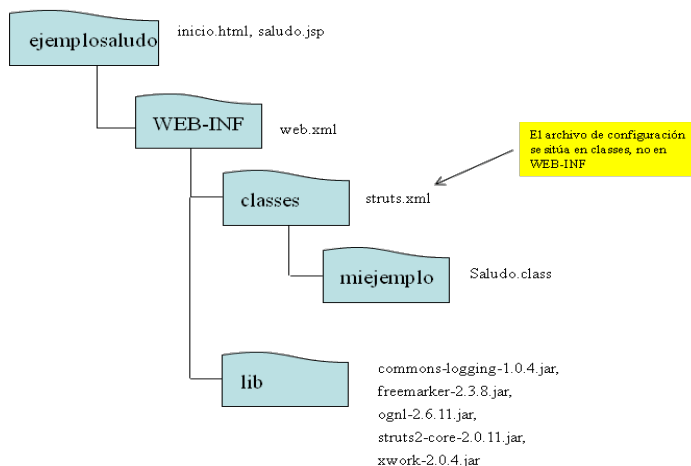


Fig. 39. Estructura de directorios de una aplicación básica Struts

En el caso de utilizar un IDE, no debemos preocuparnos de generar la estructura de directorios anterior, pues el entorno se encargará de hacerlo por nosotros. Como se indicó anteriormente, sólo tendremos que añadir la librería de archivos .jar de Struts 2 al proyecto para poder utilizar el framework.

Cuando se vaya a desarrollar una aplicación en la que se vaya a hacer uso de algunos elementos especiales como validadores o tiles, habrá que copiar también en el directorio WEB-INF\lib los archivos .jar correspondientes que se incluyen en el paquete de distribución para la utilización de estos componentes.

8.3.5 Registro de FilterDispatcher

Una vez definida la estructura de directorios, el primer paso será registrar el filtro receptor de peticiones FilterDispatcher en el archivo de configuración web.xml. Al mismo tiempo, definiremos también en este archivo como página de inicio de la aplicación a inicio.html:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4"
xmlns=http://java.sun.com/xml/ns/j2ee
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Struts ejemplo</display-name>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.
            dispatcher.FilterDispatcher
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <welcome-file-list>
        <welcome-file>inicio.html</welcome-file>
    </welcome-file-list>
</web-app>
```

8.3.6 Implementación de la clase de acción

En este ejemplo crearemos una única clase de acción, a la que como hemos visto llamaremos Saludo, que será invocada desde la página inicio.html cuando se pulse sobre el enlace. El código de esta clase se muestra en el siguiente listado:

```
package miejemplo;
import java.util.Date;
public class Saludo {
    public static final String texto=
        "Bienvenido a Struts 2";
    private String message;
    public String execute() throws Exception {
        setMessage(texto);
        return "ok";
    }
    public void setMessage(String message){
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
    public String getFechaHora(){
        return new Date().toString();
    }
}
```

Como vemos se trata de una clase normal Java en la que hemos definido un método, que por convenio **se llamará siempre *execute* en todas las clases de acción**, donde se incluyen las instrucciones asociadas a la acción y que en este ejemplo simplemente consistirán en almacenar un texto dentro de la propiedad “mensaje” del objeto. Este método deberá devolver una cadena de caracteres que será utilizada para determinar el resultado que deberá ser enviado al cliente.

Además de *execute()*, nuestra clase de acción de ejemplo cuenta con dos propiedades: “mensaje”, que contiene un texto con el mensaje a mostrar al usuario, y “fechaHora”, que contiene la fecha y hora actual del sistema. Esta última propiedad no necesita de ningún dato miembro para el almacenamiento de su valor ni, por tanto, de un método de tipo *set*, puesto que el valor de la misma se determinará en cada momento instanciando un objeto Date.

8.3.7 Registro de la clase de acción

Las clases de acción deben ser registradas en `struts.xml` mediante el elemento `<action>` dentro de un determinado paquete. Cada acción será registrada en su elemento `<action>` correspondiente, el cual deberá definir los siguientes atributos:

- **name.** Nombre asociado a la acción. Será el nombre utilizado en la URL de acceso a la acción.
- **class.** Clase que implementa la acción.

8.3.8 Reglas de navegación

Como hemos visto en la implementación de la acción anterior, el método `execute()` devuelve una cadena de caracteres que será utilizada por el controlador para determinar la vista que debe ser procesada tras la ejecución de la acción.

Cada uno de los posibles resultados que se vayan a generar al usuario para la acción deberá ser definido dentro de ésta mediante un elemento `<result>`, en lo que se conoce como reglas de navegación de la aplicación. Cada uno de estos elementos definirá en su atributo `name` el nombre asociado a este resultado, nombre que es utilizado por las clases de acción para referirse al mismo, mientras que en el cuerpo del elemento se indicará la URL asociada al resultado.

En nuestro ejemplo, el registro de la acción con su `<result>` asociado quedará como se indica en el siguiente listado, donde el resultado es implementado mediante una página JSP:

```
<?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>
    <package name="ejemplo" namespace="/ejemplo"
        extends="struts-default">
        <action name="saludo" class="miejemplo.Saludo">
            <result name="ok">/saludo.jsp</result>
        </action>
    </package>
</struts>
```

8.3.8.1 ACCIÓN POR DEFECTO

Cuando se quiere navegar directamente desde una página a otra sin necesidad de procesar ningún tipo de acción, se deberá utilizar la conocida como acción por defecto.

La acción por defecto en Struts 2 tiene como nombre “Name” y su única función es dirigir la petición a una determinada vista. Así pues, si en una página queremos incluir, por ejemplo, un enlace que al ser pulsado nos lleve a otra página, se deberá utilizar la siguiente instrucción:

```
<a href = "Name.action"> Ir a otra página </a>
```

En la regla de navegación correspondiente a la acción Name se especificará la URL de la página destino:

```
<action name="Name">
    <result>/paginadestino.jsp</result>
</action>
```

Como podemos ver, el elemento *result* en este caso no tiene que indicar ningún valor para el atributo *name*, pues **la acción por defecto únicamente puede definir un *result***.

8.3.9 Vistas

Como en cualquier aplicación Web, las vistas son generadas mediante páginas estáticas XHTML o, en el caso de respuestas dinámicas, a través de JSP.

La página de inicio de la aplicación es una simple página XHTML con un enlace que apunta a la acción saludo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
    Transitional//EN">
<html>
<head>
    <title>Ejemplo de Struts</title>
</head>
<body>
<br/><br/>
    <p align="center">Para entrar en Struts 2 pulse
        <a href="ejemplo/saludo.action"> Aquí</a>
    </p>
</body>
```

```
</html>
```

Como vemos, la URL incluida en el atributo *href* del enlace es una URL relativa a la aplicación Web y está formada por el valor incluido en el atributo *namespace* del elemento `<package>`, seguido del nombre de la acción terminada con la extensión `.action`.

Por su parte, la página de resultado que se procesa tras ejecutar la acción es una página JSP que muestra el mensaje de saludo definido en la acción, así como la hora del sistema. El código de esta página se muestra en el siguiente listado:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
    <title>Resultado de ejemplo</title>
</head>

<body>
<h1><s:property value="message" /></h1>
<h3>Fecha actual: <b><s:property value="fechaHora" /></b></h3>
</body>
</html>
```

Los datos mostrados por la página son obtenidos a partir de las propiedades “message” y “fechaHora” del objeto de acción.

Mediante el tag `<s:property>` de Struts 2 **es posible acceder desde la página de resultados a cualquiera de las propiedades del objeto Action** que se acaba de ejecutar. Este tag dispone de un atributo llamado *value* que contiene el nombre de la propiedad cuyo valor se quiere recuperar.

8.4 UTILIZACIÓN DE INTERCEPTORES

Como ya dijimos anteriormente, los interceptores son un nuevo tipo de componentes incluidos en la arquitectura Struts 2 que se ejecutan antes y después del procesamiento de una petición por parte de un objeto de acción.

El API de Struts 2 incluye un gran número de interceptores predefinidos que podemos utilizar en cualquiera de nuestras aplicaciones Web, siendo también posible la creación de interceptores personalizados.

Los interceptores son asignados a nivel de acción, es decir, el programador decide a través de `struts.xml` qué interceptor o interceptores van a ser ejecutados con cada acción.

La utilización de interceptores en una aplicación requiere realizar dos sencillas tareas en el archivo de configuración `struts.xml`:

- Declaración del interceptor.
- Asignación del interceptor a la acción.

8.4.1 Declaración del interceptor

Un interceptor se declara en el interior del elemento `<interceptors>` utilizando un elemento `<interceptor>`, el cual dispone de los siguientes atributos:

- **name.** Nombre asignado al interceptor.
- **class.** Clase a la que pertenece el interceptor.

El elemento `<interceptors>` debe estar situado directamente en el interior del elemento `<package>`. Por ejemplo, el siguiente bloque declara el interceptor `FileUploadInterceptor` de Struts 2, utilizado para facilitar el acceso a los ficheros subidos al servidor desde el cliente:

```
<struts>
  <package...>
  :
    <interceptors>
      <interceptor name="fileUpload"
        class="org.apache.struts2.
          interceptor.FileUploadInterceptor"/>
    :
  </interceptors>
```

También es posible declarar un grupo o pila de interceptores para permitir que puedan ser aplicados en bloque sobre una acción. En este caso, deberá previamente haberse declarado cada uno de los interceptores que componen el bloque mediante el elemento `<interceptor>` anterior, definiéndose después el bloque a través del elemento `<interceptor-stack>`.

Un elemento `<interceptor-stack>` dispone del atributo *name* que permite asignar un nombre al bloque. En su interior se utilizará un elemento `<interceptor-ref>` para referenciar de manera individualizada a cada uno de los interceptores que componen el bloque.

Al igual que `<interceptor>`, los elementos `<interceptor-stack>` se deben incluir en el interior de `<interceptors>`, siempre después de la declaración de los interceptores individuales.

El siguiente ejemplo declara dos interceptores y una pila de interceptores constituida por éstos:

```
<struts>
  <package...>
  :
    <interceptors>
      <interceptor name="fileUpload"
        class="org.apache.struts2.
          interceptor.FileUploadInterceptor"/>
      <interceptor name="params"
        class="com.opensymphony.xwork2.
          interceptor.ParametersInterceptor"/>
      <interceptor-stack name="ejemplostack">
        <interceptor-ref name="fileUpload"/>
        <interceptor-ref name="params"/>
      </interceptor-stack>
    :
  </interceptors>
```

Mediante el atributo *name* de `<interceptor-ref>` se indicará el nombre del interceptor que se quiere incluir en la pila.

También pueden incluirse dentro de una pila otras pilas existentes, en este caso el atributo *name* del elemento `<interceptor-ref>` deberá hacer referencia al nombre del grupo a incluir.

Como podemos ver en el listado que se presentó en el apartado dedicado a la introducción de los interceptores, el archivo de configuración `struts-default.xml` incluye la declaración de todos los interceptores del API de Struts 2, así como la de una serie de grupos cuyos interceptores suelen utilizarse habitualmente en las acciones.

8.4.2 Asignación de un interceptor a una acción

Un interceptor o grupo de interceptores se asigna a una acción a través de un elemento `<interceptor-ref>` utilizado en el interior de `<action>`. Por ejemplo, si quisiéramos asignar el grupo creado en el ejemplo anterior a la acción “personal”, deberíamos incluir lo siguiente:

```
<action name="personal" class="pack.Personal">
    <!--aquí se incluirían los resultados-->
    :
    <interceptor-ref name="ejemplostack"/>
</action>
```

También es posible asignar un interceptor o **pila de interceptores por defecto**, de manera que se aplique automáticamente a cualquier acción definida en `struts.xml`. Para esto utilizaremos el elemento `<default-interceptor-ref>`, tal y como se indica en el siguiente ejemplo:

```
<default-interceptor-ref name="ejemplostack"/>
```

La anterior etiqueta se incluirá directamente en el interior del elemento `<package>`.

Si observamos el contenido del archivo `struts-default.xml` presentado anteriormente, veremos que existe una asignación por defecto del grupo de interceptores `defaultStack`:

```
<default-interceptor-ref name="defaultStack"/>
```

Dado que todas las aplicaciones suelen heredar este archivo de configuración, lo anterior significa pues que **cualquier acción creada en una aplicación se beneficiará de la funcionalidad proporcionada por todos los interceptores definidos en este grupo** sin que el programador tenga que escribir una sola línea de código y, ni siquiera, ningún elemento de configuración adicional.

8.4.3 Inyección de dependencia

La inyección de dependencia es una técnica empleada por los interceptores que permite a los objetos de acción tener acceso a ciertos datos necesarios para poder procesar la petición.

Como hemos tenido oportunidad de comprobar, al tratarse de objetos POJO las clases de acción no heredan ninguna clase base del framework, ni su

método *execute()* recibe ningún tipo de parámetro durante su ejecución, por lo que de forma predeterminada no tienen acceso a ninguno de los objetos del contexto de aplicación, como son el objeto *request*, *response*, *session*, etc.

A través de la inyección de dependencia es posible proporcionar acceso a estos objetos desde el interior de una clase de acción en caso de que ello resulte necesario. Esta funcionalidad es proporcionada por el interceptor *ServletConfigInterceptor*, el cual forma parte del grupo de interceptores por defecto “*defaultStack*” que, como indicamos anteriormente, se encuentra asociado de forma predeterminada a todas las acciones de la aplicación que se encuentren registradas en el interior del elemento `<package>` que herede *struts-default*.

El interceptor *ServletConfigInterceptor* trabaja conjuntamente con una serie de interfaces que la clase de acción tendrá que implementar para poder tener acceso a los distintos tipos de objetos. Estas interfaces son:

- **ServletContextAware.** Proporciona acceso al objeto *ServletContext*.
- **ServletRequestAware.** Proporciona acceso al objeto *HttpServletRequest*.
- **ServletResponseAware.** Proporciona acceso al objeto *HttpServletResponse*.
- **PrincipalAware.** Proporciona acceso a un objeto *PrincipalProxy* que permite obtener información relacionada con la seguridad.
- **ParameterAware.** Proporciona acceso a todos los parámetros enviados en la petición.
- **RequestAware.** Proporciona acceso a todas las variables de petición.
- **SessionAware.** Proporciona acceso a todas las variables de sesión.
- **ApplicationAware.** Proporciona acceso a todas las variables de aplicación.

Todas ellas, salvo *ServletContextAware* que se encuentra en el paquete *org.apache.struts2.util*, están definidas en *org.apache.struts2.interceptor*.

Estas interfaces proporcionan un único método llamado *setXxx*, donde Xxx representa el nombre del objeto al que dan acceso. Por ejemplo, la interfaz *ServletContextAware* dispone del método *setServletContext()*, mientras que *ServletRequestAware* declara el método *setServletRequest()*.

Los métodos de estas interfaces reciben como parámetro un objeto que es inyectado por el interceptor *ServletConfigInterceptor* durante el procesamiento de la petición. En el caso de las cuatro primeras interfaces indicadas el parámetro corresponde al objeto de contexto de aplicación al que dan acceso (*ServletContext*, *ServletRequest*, *ServletResponse*, *PrincipalProxy*), mientras que en las cuatro restantes se tratará de una colección de tipo *Map* con los datos o parámetros existentes en el ámbito indicado.

Por ejemplo, utilizando la interfaz *SessionAware* la siguiente clase de acción comprobará la existencia de una variable de sesión llamada “user”, generando un resultado diferente en cada caso:

```
import java.util.*;
import org.apache.struts2.interceptor
public class ComprobarAction implements SessionAware{
    private Map variables;
    public void setSession(Map variables){
        this.variables=variables;
    }
    public String execute(){
        if(variables.get("user")!=null){
            return "ok";
        }
        else{
            return "error";
        }
    }
}
```

En esta otra clase de acción se utiliza la interfaz *ServletContextAware* para acceder a una variable de aplicación que lleva la cuenta del número de veces que se ha ejecutado la acción:

```
import javax.servlet.*;
import org.apache.struts2.interceptor
public class ComprobarAction implements ServletContextAware{
    private ServletContext context;
```

```

    public void setServletContext(ServletContext context){
        this.context=context;
    }
    public String execute(){
        Integer cont=(Integer)context.
            getAttribute("contador");
        if(cont!=null){
            cont++;
        }
        else{
            cont=1;
        }
        context.setAttribute("contador",cont);
        return "done";
    }
}

```

PRÁCTICA 8.1. VALIDACIÓN DE USUARIOS

Descripción

En esta práctica vamos a desarrollar el módulo de validación de usuarios que hemos estado utilizando en algunas prácticas de Capítulos anteriores, donde a través de una página de login se solicitará al usuario su identificador y password.

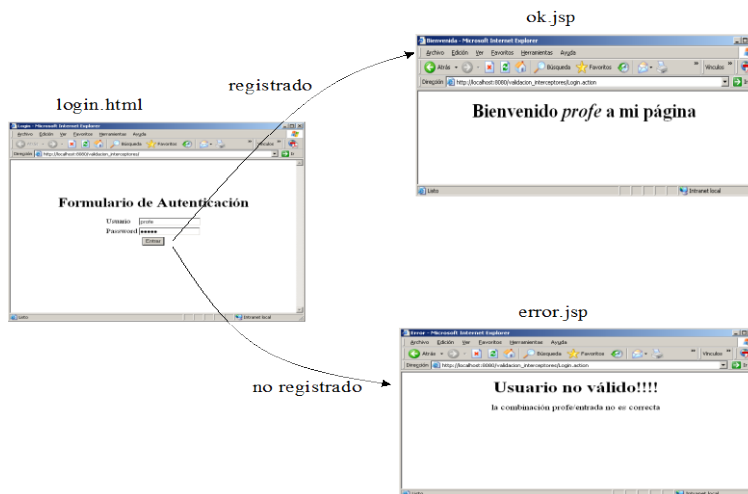


Fig. 40. Páginas de la aplicación

Si la combinación es válida se mostrará una página de bienvenida al usuario, mientras que si no lo es se le redirigirá a una página de error (figura 40).

Desarrollo

Para validar los datos del usuario utilizaremos una clase de acción que implementará las interfaces `org.apache.struts2.interceptor.ServletRequestAware` y `org.apache.struts2.util.ServletContextAware`, las cuales nos proporcionarán acceso a los objetos `HttpServletRequest` y `ServletContext`, respectivamente. Mediante estos objetos podremos acceder a los datos enviados en la petición así como a los parámetros de contexto en los que se encuentran almacenados los parámetros de conexión con la base de datos.

Los credenciales del usuario serán expuestos mediante dos propiedades del objeto de acción a fin de que sean accesibles fácilmente desde las páginas JSP de la vista, en las que se utilizará el tag `<s:property>` para recuperar ambos valores.

Utilizaremos una clase adicional, muy similar a la clase `GestionClientes` empleada en prácticas anteriores, para incluir la lógica de validación de usuarios contra la base de datos. Esta clase se apoyará a su vez en una clase `Datos` para obtener las conexiones con la base de datos.

Listado

A continuación mostramos el código de los diferentes componentes que forman la aplicación.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <context-param>
    <param-name>driver</param-name>
    <param-value>sun.jdbc.odbc.JdbcOdbcDriver
    </param-value>
  </context-param>
  <context-param>
    <param-name>cadenaCon</param-name>
    <param-value>jdbc:odbc:telefonía</param-value>
```

```

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

<welcome-file-list>
    <welcome-file>pages/login.html</welcome-file>
</welcome-file-list>
</web-app>

```

struts.xml

```

<?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>
    <package name="validacion" namespace="/"
        extends="struts-default">
        <!--registro de la clase de acción y las dos páginas
            de resultado asociadas a la misma-->
        <action name="Login" class="misclases.Validar">
            <result name="ok">/pages/ok.jsp</result>
            <result name="error">/pages/error.jsp</result>
        </action>
    </struts>

```

Validar.java

```

package misclases;
import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;

```

```
import javax.servlet.*;

public class Validar implements ServletRequestAware,
    ServletContextAware{
    String pwd;
    String user;
    ServletContext context;
    public String execute() throws Exception {
        String driver=context.getInitParameter("driver");
        String cadenaCon=context.getInitParameter("cadenaCon");
        GestionClientes gc=new
            GestionClientes(driver,cadenaCon);
        if(gc.validar(user,pwd))
            return "ok";
        else
            return "error";
    }
    //métodos de acceso a las propiedades user y pwd
    public String getUser(){
        return user;
    }
    public String getPwd(){
        return pwd;
    }
    //métodos de las interfaces asociadas al
    //interceptor
    public void setServletRequest(
        HttpServletRequest request){
        pwd=request.getParameter("password");
        user=request.getParameter("username");
    }
    public void setServletContext(ServletContext context){
        this.context=context;
    }
}
```

Datos.java

```
package misclases;

import java.sql.*;
public class Datos {
```

```
private String driver;
private String cadenacon;
public Datos() {
}
public Datos(String driver,String cadenacon){
    this.driver=driver;
    this.cadenacon=cadenacon;
}
public Connection getConexion(){
    Connection cn=null;
    try{
        Class.forName(driver).newInstance();
        cn=DriverManager.getConnection(cadenacon);
    }
    catch(Exception e){
        e.printStackTrace();
    }
    return cn;
}
public void cierraConexion(Connection cn){
    try{
        if(cn!=null && !cn.isClosed()){
            cn.close();
        }
    }
    catch(SQLException e){
        e.printStackTrace();
    }
}
}
```

GestionClientes.java

```
package misclases;

import java.sql.*;
public class GestionClientes {
    Datos dt;
    public GestionClientes(String driver, String cadenacon) {
        dt=new Datos(driver,cadenacon);
    }
    public boolean validar(String user, String pwd){
```

```

        boolean estado=false;
        try{
            Connection cn=dt.getConexion();
            //instrucción SQL para obtener los datos
            //del usuario indicado
            String query = "select * from clientes where ";
            query+="password='"+pwd+"' and usuario='"+user+"'";
            Statement st =cn.createStatement();
            ResultSet rs = st.executeQuery(query);
            estado= rs.next();
            dt.cierraConexion(cn);
        }
        catch(Exception e){
            e.printStackTrace();
        }
        finally{
            return estado;
        }
    }
}

```

login.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
                        Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Login</title>
</head>
<body>
<br/><br/><br/><br/>
<center>
<h1>Formulario de Autenticación</h1>
<form action="Login.action" method="post">
    <table>
        <tr><td>Usuario</td><td><input type="text"
            name="username" /></td></tr>
        <tr><td>Password</td><td><input type="password"
            name="password" /></td></tr>
        <tr><td align="center" colspan="2">
            <input type="submit" value="Entrar"/></td>

```



```
        </tr>
    </table>
</form>
<center>
</body>
</html>
```

ok.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
        Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Bienvenida</title>
</head>
<body>
<center>
    <h1>Bienvenido <i><s:property value="user"/></i>
        a mi página</h1>
</center>
</body>
</html>
```

error.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
        Transitional//EN"
        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Error</title>
</head>
<body>
<center>
    <h1>Usuario no válido!!!!</h1>
```

```
    la combinación <s:property value="user"/>/  
    <s:property value="pwd"/> no es correcta  
</center>  
</body>  
</html>
```

8.4.4 Interceptores personalizados

Además de utilizar los incluidos en el núcleo de Struts, un programador puede crear sus propios interceptores personalizados con sus propias reglas asociadas para luego poderlos utilizar en sus diferentes desarrollos, lo cual constituye además una forma elegante de distribuir funcionalidades entre los distintos componentes de la aplicación.

Los interceptores deben implementar la **interfaz Interceptor**, incluida en el paquete `com.opensymphony.xwork2.interceptor` y cuya definición se muestra en el siguiente listado:

```
public interface Interceptor{  
    public void init();  
    public void destroy();  
    public String intercept(ActionInvocation invocation);  
}
```

De estos tres métodos, *intercept()* es el más importante de ellos, siendo en éste donde se deberán incluir las instrucciones que realicen las tareas asignadas al interceptor. El método *intercept()* es invocado por el anterior interceptor de la cadena o por `FilterDispatcher` en caso de ser el primero.

La clase `AbstractInterceptor`, incluida en el mismo paquete que `Interceptor`, proporciona una implementación por defecto de los métodos de esta interfaz, de modo que si no necesitamos codificar los métodos *init()* y *destroy()*, **podemos heredar esta clase y sobrescribir únicamente el método *intercept()***, en vez de implementar la interfaz.

8.4.4.1 EL MÉTODO INTERCEPT()

Como ya hemos indicado anteriormente, el método *intercept()* debe contener las instrucciones que realicen las acciones asociadas al interceptor. Al igual que el método *execute()* de las clases de acción, *intercept()* devuelve un `String` que servirá para determinar el resultado a procesar para generar la respuesta del cliente.

Según se puede apreciar al examinar el formato del método, *intercept()* recibe como parámetro un objeto *ActionInvocation* el cual nos proporcionará información sobre la petición en curso y la acción asociada a la misma. A través de los métodos de este objeto el programador podrá acceder a los datos de la petición, manipular la respuesta cliente e incluso establecer el estado de la acción antes de que ésta sea procesada. Entre los métodos más importantes de *ActionInvocation* tenemos:

- **getAction()**. Devuelve una referencia al objeto de acción asociado a la petición en curso. Utilizando este objeto el programador podrá asignar valores a sus propiedades u obtener los valores de éstas a través de los métodos *set/get*.
- **getInvocationContext()**. Este método devuelve una referencia al objeto *ActionContext* que proporciona acceso al contexto de aplicación. Utilizando por ejemplo el método *get()* de este objeto podemos recuperar una referencia a los distintos objetos del contexto de la aplicación, como *request*, *response* o *session*. Entre los principales métodos de *ActionContext* tenemos:

get(Object clave). Como hemos indicado, este método permite obtener una referencia a cualquiera de los objetos del contexto de aplicación, recibiendo como parámetro la clave asociada al objeto cuya referencia se quiere recuperar.

La interfaz *org.apache.struts2.StrutsStatics* define una serie de constantes que pueden ser utilizadas como argumento en la llamada al método *get()* de *ActionContext*, como por ejemplo la constante *HTTP_REQUEST* que permitiría recuperar el objeto *HttpServletRequest* o la constante *HTTP_RESPONSE* que serviría para obtener una referencia a *HttpServletResponse*:

```
public String intercept(  
    ActionInvocation invocation){  
    ActionContext context=  
        invocation.getInvocationContext();  
    //referencia al objeto HttpServletRequest  
    HttpServletRequest request=  
        (HttpServletRequest)context.get(HTTP_REQUEST);  
    ;
```

```
}
```

- **getParameters()**. Devuelve una colección de tipo Map con todos los parámetros enviados en la petición. Por ejemplo, si en el interior del método `intercept()` quisiéramos recuperar el valor del parámetro “user” deberíamos escribir:

```
public String intercept(  
    ActionInvocation invocation){  
    ActionContext context=  
        invocation.getInvocationContext();  
    Map parametros=  
        context.getParameters();  
    String us=(String)parametros.get("user");  
    :  
}
```

- **getSession()**. Devuelve una colección Map con todas las variables de sesión.
 - **getApplication()**. Devuelve una colección Map con todas las variables de aplicación.
- **invoke()**. Después de realizar todas las operaciones el interceptor deberá pasar el control de la aplicación al siguiente interceptor o, en caso de tratarse del último elemento de la cadena, invocar al objeto de acción correspondiente. Esta operación se lleva a cabo llamando al método `invoke()` de `ActionInvocation`, cuyo valor de devolución representará la cadena asociada en `struts.xml` al resultado que se debe procesar como respuesta. Es por ello que este valor deberá ser utilizado como valor de retorno del método `intercept()`:

```
public String intercept(ActionInvocation invocation){  
    :  
    return invocation.invoke();  
}
```

PRÁCTICA 8.2. DETECCIÓN DE USUARIOS HABITUALES

Descripción

La siguiente práctica que vamos a realizar consistirá en una pequeña aplicación que sea capaz de detectar si un usuario es habitual del sitio, es decir, lo ha visitado con anterioridad, en cuyo caso se le dirigirá a una página de bienvenida personalizada que le mostrará un mensaje indicándole esta circunstancia.

Si el usuario no ha visitado nunca el sitio se le llevará a una página en la que deberá introducir un nombre de usuario que sirva para recordarle durante la próxima visita que realice al sitio. La imagen de la figura 41 muestra las distintas páginas involucradas en la aplicación.

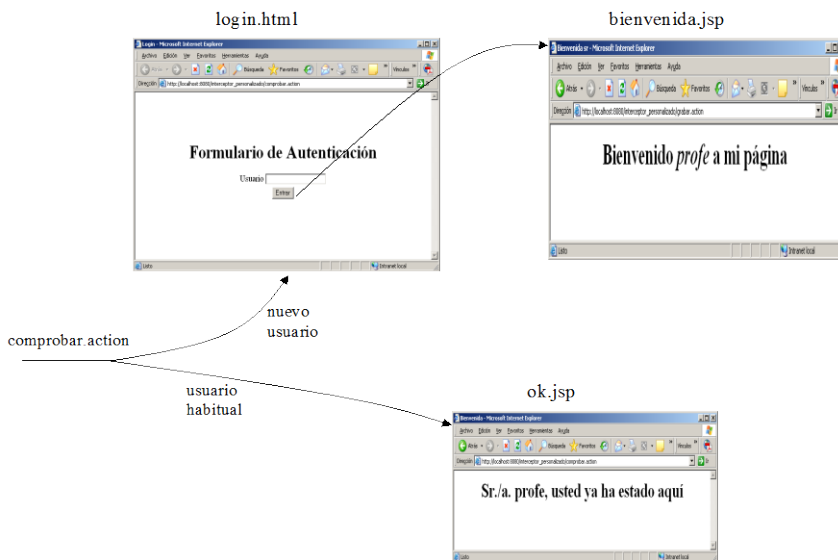


Fig. 41. Páginas de la aplicación

Desarrollo

Para recordar a los usuarios la aplicación utilizará una cookie en la que almacenará el nombre de usuario introducido en la página de inicio. Cada vez que un usuario entre en la aplicación, un interceptor comprobará la existencia de esta cookie, estableciendo en una propiedad de tipo *boolean* definida dentro de la clase de acción si se trata de un usuario habitual (*true*) o de un nuevo usuario (*false*).

En función del valor de esta propiedad la acción generará el resultado apropiado. Esta acción, llamada “comprobar”, deberá corresponder a la petición de inicio de la aplicación (comprobar.action).

Una segunda acción llamada “grabar” se encargará de crear la cookie para los usuarios que entran por primera vez en la aplicación.

Listado

Seguidamente se presenta el código de los distintos componentes que forman esta aplicación.

struts.xml

```
<?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>
    <package name="validacion" namespace="/"
        extends="struts-default">
        <interceptors>
            <!--registro del interceptor personalizado-->
            <interceptor name="compruebauser"
                class="misclases.UserInterceptor"/>
        </interceptors>
        <!--acción correspondiente a la petición de inicio-->
        <action name="comprobar"
            class="misclases.ComprobarAction">
            <result name="si">/pages/ok.jsp</result>
            <result name="no">/pages/login.html</result>
            <interceptor-ref name="compruebauser"/>
        </action>
        <!--acción asociada a la petición grabar.action
            generada desde la página login.html-->
        <action name="grabar" class="misclases.GrabarAction">
            <result name="grabado">
                /pages/bienvenida.jsp
            </result>
        </action>
    </package>
```

```
</struts>
```

UserInterceptor.java

```
package misclases;

import com.opensymphony.xwork2.*;
import com.opensymphony.xwork2.interceptor.*;
import org.apache.struts2.*;
import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;
import javax.servlet.*;

//clase correspondiente al interceptor personalizado
public class UserInterceptor extends AbstractInterceptor
    implements StrutsStatics{
    public String intercept(ActionInvocation invocation)
        throws Exception {
        ComprobarAction action =
            (ComprobarAction)invocation.getAction();

        //la interfaz StrutsStatics contiene una serie de
        //constantes que pueden ser utilizadas por el método
        //get() de ActionContext para obtener los distintos
        //objetos del contexto de la aplicación
        ActionContext context=
            invocation.getInvocationContext();
        HttpServletRequest request=
            (HttpServletRequest)context.get(HTTP_REQUEST);
        Cookie [] cookies=request.getCookies();
        if (cookies != null) {
            for (int i=0; i< cookies.length; i++) {
                if(cookies[i].getName().equals("user")){
                    action.setExiste(true);
                    action.setUser(cookies[i].getValue());
                }
            }
        }
        return invocation.invoke();
    }
}
```

ComprobarAction.java

```
package misclases;

import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;
import javax.servlet.*;

public class ComprobarAction{
    //propiedad que indica si el usuario es o no
    //habitual
    boolean existe;
    String user;
    public String execute() throws Exception {
        if(existe)
            return "si";
        else
            return "no";
    }
    public String getUser(){
        return user;
    }
    public void setUser(String user){
        this.user=user;
    }
    public boolean getExiste(){
        return existe;
    }
    public void setExiste(boolean existe){
        this.existe=existe;
    }
}
```

GrabarAction.java

```
package misclases;

import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
```



```
public class GrabarAction implements ServletRequestAware,
                                   ServletResponseAware{
    String user;
    HttpServletResponse response;
    public String execute() throws Exception {
        //genera la cookie para recordar al usuario
        Cookie ck=new Cookie("user",user);
        ck.setMaxAge(2000);
        response.addCookie(ck);
        return "grabado";
    }
    public String getUser(){
        return user;
    }
    public void setServletRequest(
        HttpServletRequest request){
        user=request.getParameter("username");
    }
    public void setServletResponse(
        HttpServletResponse response){
        this.response=response;
    }
}
```

login.html

```
<html>
<head>
    <title>Login</title>
</head>
<body>
<br/><br/><br/><br/>
<center>
<h1>Formulario de Autenticación</h1>
<form action="grabar.action" method="post">
    <table>
        <tr><td>Usuario</td><td><input type="text"
            name="username" /></td></tr>
        <tr><td align="center" colspan="2">
            <input type="submit" value="Entrar" /></td>
        </tr>
    </table>
</form>
```

```
</form>
<center>
</body>
</html>
```

bienvenida.jsp

```
<!--página para nuevos usuarios-->
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Bienvenida</title>
</head>
<body>
<center>
    <h1>Bienvenido <i><s:property value="user"/>
        </i> a mi p&aacute;gina</h1>
</center>
</body>
</html>
```

ok.jsp

```
<!--página para usuarios habituales-->
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Bienvenida</title>
</head>
<body>
<center>
    <h1>Sr./a. <i><s:property value="user"/></i>,
```

```
usted ya ha estado aquí</h1>
</center>
</body>
</html>
```

8.5 LA LIBRERÍA DE ACCIONES STRUTS-TAGS

En este punto vamos a analizar las principales acciones JSP que podemos utilizar para la creación de vistas en aplicaciones Struts 2, acciones que forman parte de la librería struts-tags cuyas clases de implementación se incluyen en el archivo struts2-core-2.0.11.jar del paquete de distribución de Struts 2.

Para incluir una referencia al archivo de librería struts-tags desde una página JSP, habrá que añadir la siguiente directiva *taglib* al principio de la misma, siendo habitual utilizar como prefijo para acceder a sus acciones la letra “s”:

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

El conjunto de acciones JSP proporcionadas por Struts 2 puede dividirse en tres grandes grupos:

- **Acciones de manipulación de datos.** En este grupo se incluyen todas las acciones para la manipulación y extracción de datos de aplicación, como el acceso a las propiedades de un objeto Action o el establecimiento de valores en variables JSP.
- **Acciones de control.** Incluye las utilizadas para el control del flujo de la aplicación.
- **Acciones UI.** Incluye acciones para la generación de controles gráficos en formularios XHTML. Estos componentes encapsulan además toda la funcionalidad necesaria para que los datos capturados por los mismos sean insertados, con la ayuda de uno de los interceptores predefinidos de Struts 2, en las propiedades del objeto Action que atenderá la petición.

8.5.1 El stack de objetos

Antes de pasar a analizar las acciones de Struts 2 es conveniente introducir el concepto de stack de objetos, conocido también como ValueStack, y comprender su funcionamiento, ya que la mayoría de los tags de Struts 2 realizan un acceso al mismo para llevar a cabo su misión.

El stack de objetos es una especie de pila donde se van acumulando los distintos objetos utilizados en la aplicación, como el objeto de acción que acaba de ser ejecutado, los parámetros de la petición o los objetos HTTP (*application*, *session* y *request*).

Suele ser habitual que desde el interior de una página JSP los tags de Struts 2 necesiten acceder a alguna de las propiedades de los objetos del Stack. La notación utilizada para ello se basa en las llamadas expresiones OGNL (Object-Graph Navigation Language).

El lenguaje OGNL es bastante simple; en el caso de querer acceder a una propiedad de uno de los objetos del stack, **bastará con indicar el nombre de la propiedad** a la que se quiere acceder en el atributo correspondiente de la acción. Por ejemplo, si quisiéramos acceder una propiedad llamada “password”, se indicaría directamente el nombre de la propiedad en el atributo *value* del tag *property*:

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

En este caso se iría interrogando a los distintos objetos del stack por esta propiedad, empezando por el que se encuentra en la posición superior (último objeto utilizado) hasta el más interno. En el momento en que se encuentre un objeto que cuente con dicha propiedad se devolverá su valor al punto de llamada.

En caso de que deseemos acceder al valor de la propiedad para un determinado objeto del Stack, la expresión OGNL a utilizar debe tener el siguiente formato:

```
#objeto.propiedad
```

Por ejemplo, para recuperar la propiedad *password* de un objeto identificado como *user1* sería:

```
<s:property value="#user1.password"/>
```

Así mismo, los atributos *value* de las acciones Struts 2 también admiten expresiones EL, las cuales forman parte del Java Enterprise, y cuya sintaxis se detalla en el Apéndice A del libro. Una expresión EL debe indicarse entre los símbolos “\${” y “}”.

Por ejemplo, la siguiente instrucción recupera la propiedad *password* existente en un objeto identificado como *user1* que se encuentra almacenado en un atributo de sesión:

```
<s:property value="${sessionScope['user1'].password}"/>
```

8.5.2 Acciones de manipulación de datos

Seguidamente pasaremos a analizar las principales acciones que se incluyen en este grupo.

8.5.2.1 BEAN

Realiza la instanciación de una clase de tipo JavaBean. Los atributos soportados por esta acción son:

- **name.** Nombre cualificado de la clase que se va a instanciar.
- **id.** Identificador de la variable donde se almacenará la referencia a la instancia.

8.5.2.2 PARAM

Se utiliza para establecer parámetros en otros tags, como por ejemplo las propiedades de un JavaBean creado con la acción `<s:bean>`. Mediante su atributo `name` se indica el nombre del parámetro, incluyendo en el cuerpo del elemento el valor que se quiere asignar.

El siguiente bloque de acciones de ejemplo crearía una instancia del bean `Mensaje` y le asignaría valores a sus propiedades `remite`, `destinatario` y `texto`:

```
<s:bean name="javabeans.Mensaje" id="mensa">
    <s:param name="remite">profe</s:param>
    <s:param name="destinatario">marta</s:param>
    <s:param name="texto">hola</s:param>
</s:bean>
```

8.5.2.3 PROPERTY

Recupera el valor de una determinada propiedad. Su principal atributo es *value*, el cual, como hemos visto anteriormente, debe contener el nombre de la propiedad que se quiere recuperar. El elemento `<s:property>` es aplicado sobre los objetos del stack, empezando desde el más externo hasta el más interno.

Si se emplea `property` sin especificar su atributo `value`, se recuperaría el último elemento depositado en el stack:

```
<!-- extrae el último elemento depositado en el stack-->
```

```
<s:property/>
```

8.5.2.4 PUSH

Este elemento añade un determinado objeto al stack. Mediante su atributo *value* se indica el objeto a añadir.

El siguiente ejemplo mostraría en la página el valor de la propiedad “destino” del objeto “mensa”:

```
<s:push value="mensa">
    <s:property value="destino"/>
</s:push>
```

8.5.2.5 SET

Asigna un valor a una variable y la deposita en un determinado ámbito. Sus atributos son:

- **name.** Nombre asignado a la variable.
- **value.** Valor asignado a la variable.
- **scope.** Ámbito de la variable. Su valor puede ser “action”, “page”, “request”, “session” o “application”, siendo “action” el valor predeterminado.

Después de establecer la variable ésta queda en el stack de objetos. El siguiente ejemplo asigna una cadena de texto a una variable y a continuación muestra su valor:

```
<s:set name="variable" value="${'hola'}"/>
valor de la variable: <h1><s:property
    value="#variable" default="defecto"/></h1>
```

Obsérvese cómo el valor de la variable aparece encerrado dentro de una expresión.

8.5.3 Acciones de control

Veamos a continuación las principales acciones que se incluyen en este grupo.

8.5.3.1 IF

Evalúa el cuerpo de la acción si se cumple la condición establecida en su atributo *test*. Dicha condición debe dar por tanto como resultado un tipo boolean. Al ejecutar el siguiente bloque de sentencias se mostrará el texto “usuario administrador” en la página de respuesta:

```
<s:set name="user" value="{#{'administrador'}}"/>
<s:if test="{#{#user=='administrador'}}">
    <h1>usuario <s:property value="#user"/></h1>
</s:if>
```

Una acción `<s:if>` puede ir seguida de una o varias acciones `<s:elseif>` que comprueben otras condiciones si la condición de `<s:if>` resulta *false*, y de una acción `<s:else>` cuyo cuerpo será evaluado si no se cumple ninguna de las condiciones `<s:if>` ni `<s:elseif>`:

```
<s:if test="condicion1">
    <!--evaluado si condicion1 es true-->
</s:if>
<s:elseif test="condicion2">
    <!--evaluado si condicion1 es false y
    condicion2 es true-->
</s:elseif>
<s:else>
    <!--evaluado si condicion1 y
    condicion2 son ambas false-->
</s:else>
```

8.5.3.2 ITERATOR

La acción iterator se utiliza para recorrer una colección de tipo Collection o Iterator. Dispone de los siguientes atributos:

- **value.** Colección sobre la que se realizará la iteración.
- **id.** Variable que contendrá una referencia al elemento de la colección correspondiente a la iteración actual.

Por ejemplo, si *ListaTareas* es una clase de tipo colección con una propiedad *agregar*, cuyo método *setAgregar()* realiza la inserción de un nuevo elemento en la colección, el siguiente grupo de sentencias añadiría tres tareas a la colección para después recorrerla y mostrar la lista de tareas en la página, siendo

“tareas” la propiedad de la clase `ListaTareas` que da acceso a la colección de elementos:

```
<s:bean name="clases.ListaTareas" var="tarea"/>
    <s:param name="agregar">sumar</s:param>
    <s:param name="agregar">restar</s:param>
    <s:param name="agregar">multiplicar</s:param>
</s:bean>
<s:iterator id="actual" value="#tarea.tareas">
    <!--muestra el valor actual del iterador-->
    <s:property value="#actual"/>
</s:iterator>
```

Si lo único que queremos hacer dentro del `iterator` es extraer el valor de la iteración actual, no es necesario indicar el atributo `id` en el tag. El motivo es que, cuando se recorre la colección, Struts2 deposita el valor actual en la parte superior del stack, pudiéndose recuperar directamente dicho valor utilizando el elemento *property* sin atributos.

Según lo indicado, el *iterator* anterior podría implementarse también de la siguiente forma:

```
<s:iterator value="#tarea.tareas">
    <!--muestra el valor actual del iterador-->
    <s:property/>
</s:iterator>
```

8.5.4 Acciones UI

Las acciones UI de struts 2 facilitan la construcción de las interfaces gráficas de una aplicación Web, simplificando las tareas de captura de datos de usuario y su validación.

Aunque muchos de los tags UI de struts se corresponden con un componente gráfico XHTML, el código embebido dentro de estos tags realiza una serie de funciones para el programador que, en caso de haber utilizado elementos simples XHTML, habrían tenido que ser definidos mediante código. Entre estas funciones están:

- Vinculación de los datos de los controles a propiedades de un *action*.

- Generación de variables en el ValueStack con los datos de los controles.
- Facilitar la validación y conversión de los valores suministrados en los controles.
- Formateado de los controles. Los UI tags de Struts 2 generan etiquetas de marcado adicionales para dar el formato adecuado de presentación a los controles.

Seguidamente, estudiaremos los diferentes tags proporcionados por Struts 2 para la construcción de formularios de captura de datos.

8.5.4.1 FORM

Es el tag más importante. Genera una etiqueta XHTML de tipo `<form>` que nos permitirá definir el action que procesará los datos recogidos por los controles del formulario.

Esta acción incluye la funcionalidad necesaria para evitar que el contenido de la página sea refrescado una vez que se han enviado los datos y, por tanto, se mantengan los valores de los controles en caso de que vuelva a mostrarse la página al usuario. Entre sus atributos más importantes están:

- **name.** Nombre asociado al elemento `<form>` XHTML.
- **method.** Método de envío de los datos.
- **action.** Nombre del objeto Action que capturará la petición. Si no se utiliza este atributo la petición será procesada por el objeto Action actual.
- **namespace.** Namespace en el que se encuentra la definición de la acción. Si no se utiliza este atributo, se asume que se trata del namespace actual.

8.5.4.2 TEXTFIELD

Genera una caja de texto de una línea. Como atributos más destacados tenemos:

- **name.** Nombre de la propiedad del objeto Action en la que se volcará el contenido del control. Con ayuda de uno de los

interceptores predefinidos de Struts 2, **la información recogida en los campos del formulario es volcada directamente en las propiedades del objeto Action** indicado en los elementos `<s:form>`, sin necesidad de incluir ninguna instrucción de código ni elementos de configuración adicionales.

- **value.** Valor de inicialización del control.
- **readonly.** Si su valor es *true*, el usuario no podrá modificar el contenido del campo de texto.
- **label.** Texto que aparecerá al lado del control. Dicho texto se incluirá dentro de un elemento XHTML de tipo `<label>`. Además de ello, de forma predeterminada el formulario aplicará sobre este componente un formato de tipo fila de tabla, incluyendo el label en una celda y el componente de texto en otra.

Por ejemplo, si definimos el siguiente formulario en una página JSP:

```
<s:form action="proceso.action">
  <s:textfield name="user" label="usuario"/>
  <s:submit value="entrar"/>
</s:form>
```

el bloque equivalente XHTML que se generará al procesar la página será similar al siguiente:

```
<form id="proceso" action="proceso.action"
      method="post">
  <table class="wwFormTable">
    <tr>
      <td class="tdLabel"><label for="proceso_user"
        class="label">usuario:</label></td>
      <td>
        <input type="text" name="user" value=""
          id="proceso_user"/></td></tr>
    <tr>
      <td colspan="2"><div align="right">
        <input type="submit" id="proceso_0"
          value="Entrar"/>
      </div></td>
    </tr>
```

```
</table>
</form>
```

8.5.4.3 PASSWORD

Genera una caja de texto de tipo password. Además de los atributos *name*, *label* y *value*, cuyo significado es el mismo que en el caso de la acción `<s:textfield>`, este control dispone del atributo *showPassword* de tipo boolean, mediante el que podemos indicar si queremos inicializar el control con el valor del *ValueStack*, en caso de que exista una propiedad con el mismo nombre que el control, o no.

8.5.4.4 TEXTAREA

Genera una caja de texto multilínea. Los atributos *name*, *readonly* y *value* tienen el mismo significado que en los controles anteriores, aunque dispone además de los siguientes atributos para establecer el tamaño del control:

- **cols.** Ancho del control en número de líneas.
- **rows.** Alto del control en número de filas.

8.5.4.5 SUBMIT

Genera un botón de tipo submit para realizar la petición y envío del formulario. Mediante su atributo *value* se indica el texto que se quiere mostrar en el control.

Para ver la potencia de estos elementos gráficos volvamos de nuevo a la práctica 8.1. En ella utilizábamos un formulario XHTML para recoger los credenciales del usuario y enviarlos a un objeto de acción, cuya clase debía implementar una de las interfaces asociadas al interceptor *ServletConfigInterceptor* para poder tener acceso al objeto *HttpServletRequest* y recuperar así los datos del formulario.

Si en vez de una página XHTML hubiéramos utilizado una página JSP con los controles Struts 2 indicados anteriormente, nada de esto habría sido necesario puesto que **los contenidos de los campos de texto se pueden volcar directamente en las correspondientes propiedades del objeto Action**. Con esta solución, la página JSP de inicio quedaría:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>
```

```
<html>
<head>
    <title>Validación</title>
</head>
<body>
<!--siempre tiene que incluirse la extensión .action en
la URL. En este caso, el tag form de Struts pone la extensión
automáticamente a la cadena incluida en action-->

<s:form action="Login">
    <table>
        <tr><td>Usuario</td><td>
            <s:textfield name="user"/></td></tr>
        <tr><td>Password</td><td>
            <s:password name="pwd"/></td></tr>
        <tr><td colspan="2">
            <s:submit/></td></tr>
    </s:form>
</body>
</html>
```

Por su parte, la clase de acción Login quedaría ahora algo más simplificada:

```
package misclases;

import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;
import javax.servlet.*;

//no es necesario implementar ServletRequestAware
public class Validar implements ServletContextAware{
    String pwd;
    String user;
    ServletContext context;
    public String execute() throws Exception {
        String driver=context.getInitParameter("driver");
        String cadenaCon=context.getInitParameter("cadenaCon");
        GestionClientes gc=
            new GestionClientes(driver,cadenaCon);
        if(gc.validar(user,pwd))
            return "ok";
    }
}
```

```
        else
            return "error";
    }
    public String getUser(){
        return user;
    }
    public String getPwd(){
        return pwd;
    }
    public void setServletContext(ServletContext context){
        this.context=context;
    }
}
```

8.5.4.6 RADIO

Con esta acción es posible generar una lista de elementos XHTML de tipo radio. Su propiedad más importante es *list*, la cual indicará el objeto de tipo lista cuyos datos serán utilizados para la generación de los controles.

Por ejemplo, supongamos que tenemos un objeto bean identificado como “calendario” que dispone de una propiedad “días” de tipo ArrayList de cadenas de caracteres. Por otro lado, el bean cuenta con una propiedad “agregar” cuyo método asociado *setAgregar()* se encarga de almacenar en la lista la cadena recibida como parámetro. Con esta situación, el siguiente bloque de sentencias generará cinco botones de radio, cada uno de los cuales tendrá como texto asociado el nombre de un día:

```
<html>
<body>
<s:bean name="calendario" id="diactual"/>
    <s:param name="agregar">lunes</s:param>
    <s:param name="agregar">martes</s:param>
    <s:param name="agregar">miércoles</s:param>
    <s:param name="agregar">jueves</s:param>
    <s:param name="agregar">viernes</s:param>
</s:bean>
<h1>Dias laborables:</h1>
    <s:radio name="semana" list="#calendario.dias"/>
</body>
</html>
```

En la imagen de la figura 42 se presenta el aspecto de la página generada a partir del código anterior.

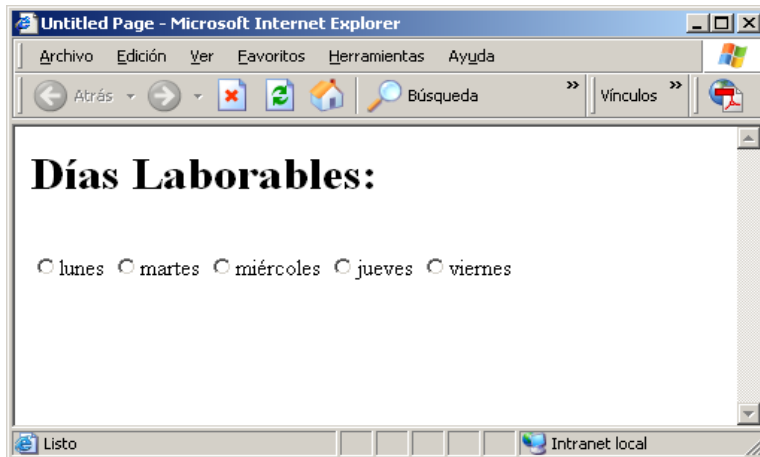


Fig. 42. Lista de botones con los días de la semana

Como podemos comprobar, el atributo `list` del componente se asocia a la propiedad de tipo lista del bean; por su parte, el atributo `name` del tag determina el nombre de la propiedad del objeto Action en la que se volcará el valor del atributo `value` del botón seleccionado.

En el ejemplo anterior, dado que se trata de una colección de cadenas de caracteres, el nombre del día de la semana se utiliza tanto como texto a mostrar en el componente como valor asociado al mismo, sin embargo, si la colección fuera de cualquier otro tipo de objeto JavaBean, sería necesario especificar los siguientes atributos adicionales de `<s:radio>`:

- **listKey.** Nombre de la propiedad del bean que se utilizará como valor asociado a cada botón de radio. En caso de colecciones de tipo Map, no será necesario utilizar esta propiedad, puesto que la clave de la colección será utilizada directamente como valor asociado
- **listValue.** Nombre de la propiedad del bean que se utilizará como texto a mostrar con cada botón de radio. En caso de colecciones de tipo Map, no será necesario utilizar esta propiedad, utilizándose el valor de cada elemento de la colección como texto del *radio button*.

8.5.4.7 CHECKBOX

Genera un elemento XHTML de tipo checkbox. Sus principales atributos son:

- **name.** Nombre de la propiedad del objeto Action donde se almacenará el valor del atributo *fieldValue* de la acción.
- **fieldValue.** *Value* asociado al elemento XHTML correspondiente.

8.5.4.8 CHECKBOXLIST

Es similar a `<s:radio>` sólo que en el caso de `<s:checkboxlist>` se generará una lista de casillas de verificación. Sus atributos son los mismos que los del tag `<s:radio>`, teniendo en cuenta que **la propiedad del objeto Action especificada en el atributo *name* deberá ser de tipo array o list** para poder almacenar el conjunto de valores seleccionados.

8.5.4.9 SELECT

Mediante esta acción se genera una lista de selección de elementos, donde cada elemento es codificado en la página mediante una etiqueta XHTML de tipo `<option>`. Sus principales atributos son:

- **name.** Nombre de la propiedad del objeto Action donde será volcado el valor del elemento seleccionado.
- **list.** Indica el objeto lista (List o Map) que contiene los datos con los que se rellenará el control.
- **listKey y listValue.** El significado de estas propiedades es el mismo que en el caso de los otros componentes de colección `<s:radio>` y `<s:checkboxlist>`.
- **headerValue.** Texto que aparecerá como primer elemento de la lista. Resulta útil en aquellos casos en que no queramos que aparezca ningún elemento de la lista preseleccionado.
- **headerKey.** Clave (value) asociada al primer elemento de la lista especificado en *headerValue*.

El siguiente ejemplo genera una lista con los días de la semana. Como se puede observar, la lista es generada mediante una expresión en el propio atributo *list*, indicando la clave y texto a visualizar para cada elemento:

```

<s:form>
<s:select name="semana"
    headerKey="0"
    headerValue="--Seleccione día --"
    list="#{'01':'lunes','02':'martes','03':'miércoles',
        '04':'jueves','05':'viernes',
        '06':'sábado','07':'domingo'}"/>
</s:form>

```

Si a la hora de definir un select, los elementos que componen la lista son conocidos durante la fase de diseño, cada uno de éstos puede ser especificado.

PRÁCTICA 8.3. SELECCIÓN MÚLTIPLE DE TEMAS

Descripción

Vamos a desarrollar una sencilla práctica como ejemplo de utilización de los controles de selección, más concretamente, del checkboxlist.

Tras pulsar un enlace existente en una página de inicio, se accederá a una página que nos mostrará una serie de temas entre los que podremos elegir varios de ellos, para lo que se utilizará un grupo de checkbox. Una vez seleccionados los temas y después de pulsar el botón “Entrar”, se mostrará una nueva página que simplemente nos mostrará los códigos de los temas seleccionados.

La figura 43 ilustra el flujo de páginas de la aplicación.

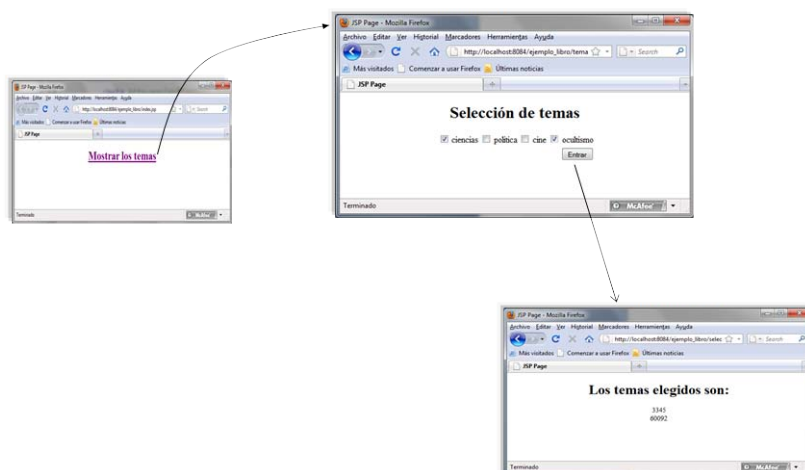


Fig. 43. Flujo de páginas de la aplicación

Desarrollo

Para el desarrollo de esta aplicación, crearemos un bean llamado Tema que encapsule los dos datos que caracterizarán a esta entidad: el código de tema y el nombre. La clase Action llamada Tematica, asociada a la pulsación del enlace de la página de inicio, será la encargada de generar la colección de temas y de depositarla en una propiedad del objeto, a fin de que sea accesible para el componente checkboxlist incluido en la página temas.jsp.

Los códigos de los temas seleccionados serán cargados en un ArrayList de tipo Integer por parte del objeto de acción Selecciones, asociado a la pulsación del botón “Entrar”. El contenido de esta colección será mostrado posteriormente por la página seleccion.jsp.

Listado

Los siguientes listados de código corresponden a los diferentes componentes de la aplicación

Tema.java

```
package beans;

public class Tema {
    private int codigo;
    private String tema;
    public Tema() {
    }
    public Tema(int codigo, String tema) {
        this.codigo = codigo;
        this.tema = tema;
    }
    public int getCodigo() {
        return codigo;
    }
    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
    public String getTema() {
        return tema;
    }
    public void setTema(String tema) {
```

```
        this.tema = tema;
    }
}
```

Tematica.java

```
package acciones;
import java.util.*;
import beans.Tema;
public class Tematica {
    private ArrayList<Tema> temas;
    public ArrayList<Tema> getTemas() {
        return temas;
    }
    public void setTemas(ArrayList<Tema> temas) {
        this.temas = temas;
    }
    public String execute() throws Exception{
        temas=new ArrayList<Tema>();
        temas.add(new Tema(3345,"ciencias"));
        temas.add(new Tema(12432,"politica"));
        temas.add(new Tema(98371,"cine"));
        temas.add(new Tema(60092,"ocultismo"));
        return "completado";
    }
}
```

Selecciones.java

```
package acciones;
import java.util.*;

public class Selecciones {
    private ArrayList<Integer> seleccionados;
    public ArrayList<Integer> getSeleccionados() {
        return seleccionados;
    }
    public void setSeleccionados(
        ArrayList<Integer> seleccionados) {
        this.seleccionados = seleccionados;
    }
}
```

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

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
                        HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <h1><a href="tematica.action">
                Mostrar los temas</a></h1>
        </center>
    </body>
</html>
```

temas.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
                        HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <h1>Selección de temas</h1>
```

```

        <s:form action="selecciones.action" method="post" >
            <s:checkboxlist name="seleccionados"
                list="temas" listKey="codigo" listValue="tema"/>
            <s:submit value="Entrar"/>
        </s:form>
    </center>
</body>
</html>

```

selección.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
    HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <center>
            <h1>Los temas elegidos son:</h1>
            <s:iterator value="seleccionados">
                <s:property/><br/>
            </s:iterator>
        </center>
    </body>
</html>

```

struts.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
    2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
    <constant name="struts.enable.DynamicMethodInvocation"

```

```

        value="false" />
<constant name="struts.devMode" value="false" />
<package name="temas_libros"
    namespace="/" extends="struts-default">
    <action name="tematica" class="acciones.Tematica">
        <result name="completado">/temas.jsp</result>
    </action>
    <action name="selecciones"
        class="acciones.Selecciones">
        <result name="seleccion">/seleccion.jsp</result>
    </action>
</package>
</struts>

```

web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.
            FilterDispatcher</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>

```

8.5.4.10 ACTIONERROR

Aunque no se trata de un control gráfico, la acción `<s:actionerror/>` se encuentra dentro del grupo de acciones gráficas de Struts 2. Su cometido es mostrar los mensajes de error en caso de que hayan sido generados desde la acción.

El método `addActionError(String mensajerror)`, incluido en la clase `com.opensymphony.xwork2.ActionSupport`, permite añadir fácilmente un mensaje de error desde la clase de acción. Tan sólo hay que crear la clase de acción como una subclase de `ActionSupport` e invocar a este método en el momento en que se considere que hay que añadir un mensaje de error.

8.6 VALIDADORES

Al igual que sucede en Struts 1.x, Struts 2 cuenta con la posibilidad de llevar a cabo la validación de los datos de usuario de manera declarativa. En este sentido, Struts 2 dispone de una serie de validadores predefinidos que permiten realizar esta tarea sin que para ello el programador tenga que escribir una sola instrucción de código Java.

8.6.1 Validadores predefinidos

El paquete de distribución de Struts 2 cuenta con una serie de validadores predefinidos, incluidos en el archivo `xwork-2.0.4.jar` (si utiliza otra versión de Struts 2 los números que aparecen en el nombre del archivo pueden ser diferentes).

Para que una aplicación pueda hacer uso de estos validadores es necesario que estén declarados en un documento XML (`validation.xml`), cuya ubicación esté accesible desde el *classpath* de la aplicación. He aquí el aspecto que debe tener este documento:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
    "-//OpenSymphony Group//XWork Validator
    Config 1.0//EN"
    "http://www.opensymphony.com/xwork/xwork-validator-
    config-1.0.dtd">
<validators>
    <validator name="required" class="com.opensymphony.
        xwork2.validator.validators.RequiredFieldValidator"/>
    <validator name="requiredstring" class="com.opensymphony.
        xwork2.validator.validators.RequiredStringValidator"/>
```

```
<validator name="int" class="com.opensymphony.
    xwork2.validator.validators.IntRangeFieldValidator"/>
<validator name="double" class="com.opensymphony.
    xwork2.validator.validators.DoubleRangeFieldValidator"/>
<validator name="date" class="com.opensymphony.
    xwork2.validator.validators.DateRangeFieldValidator"/>
<validator name="expression" class="com.opensymphony.
    xwork2.validator.validators.ExpressionValidator"/>
<validator name="fieldexpression" class="com.
    opensymphony.xwork2.validator.
        validators.FieldExpressionValidator"/>
<validator name="email" class="com.opensymphony.
    xwork2.validator.validators.EmailValidator"/>
<validator name="url" class="com.opensymphony.
    xwork2.validator.validators.URLValidator"/>
<validator name="visitor" class="com.opensymphony.
    xwork2.validator.validators.VisitorFieldValidator"/>
<validator name="conversion" class="com.opensymphony.
    xwork2.validator.validators.
        ConversionErrorFieldValidator"/>
<validator name="stringlength" class="com.opensymphony.
    xwork2.validator.validators.
        StringLengthFieldValidator"/>
<validator name="regex" class="com.opensymphony.
    xwork2.validator.validators.RegexFieldValidator"/>
</validators>
```

El archivo `xwork-2.0.4.jar` también incluye este documento XML, por lo que no será necesario tener que generarlo manualmente. Por otro lado, al estar incluido en el directorio `WEB-INF\lib` de la aplicación no será necesario realizar ninguna operación adicional en la variable de entorno *classpath*.

Además de declarar los validadores, es necesario activar el interceptor Struts encargado de poner en marcha el proceso de validación automática:

```
<interceptor name="validator" class="com.opensymphony.
    xwork2.validator.ValidationInterceptor"/>
```

La anterior operación tampoco es necesaria que la realicemos explícitamente en nuestro `struts.xml`, puesto que **el documento de configuración por defecto ya incluye el registro de este validador** además de añadirlo al stack de validadores por defecto.

8.6.2 Utilización de validadores en una aplicación

La utilización de estos validadores en una aplicación Struts 2 es una tarea tremendamente sencilla: tan sólo tendremos que crear un archivo XML en el que asociaremos las reglas de validación con los campos del objeto Action a los que se las queramos aplicar.

Este archivo XML tendrá que nombrarse siguiendo el siguiente formato:

claseaccion-validation.xml

donde *claseaccion* se corresponderá con el nombre de la clase a la que pertenece el objeto Action cuyos campos se quieren validar.

Por ejemplo, supongamos que tenemos una clase de acción llamada Login.class, similar a la utilizada en una de las prácticas analizadas en este Capítulo, y que se encarga de comprobar los credenciales de un usuario suministrados desde una página de login:

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

public class Login extends ActionSupport {
    private String username;
    private String password;
    public String execute() throws Exception {
        if(getPassword().equals("admin"))
            return SUCCESS;
        else
            return ERROR;
    }

    public void setUsername(String us){
        this.username = us;
    }
    public String getUsername(){
        return this.username;
    }
    public void setPassword(String pwd){
        this.password = pwd;
    }
    public String getPassword(){
```



```
        return this.password;
    }
}
```

La clase comprueba el campo *password* y si su valor es igual a “admin”, devolverá una de las constantes predefinidas en la interfaz *ActionSupport*, que será interpretada como validación correcta.

Si quisiéramos añadir una regla de validación para forzar la entrada obligatoria de datos en ambos campos, tendríamos que crear un archivo llamado *Login-validation.xml* en el que asociaríamos cada regla con cada campo, según se indica en el siguiente listado:

```
<!DOCTYPE validators PUBLIC
    "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
    "http://www.opensymphony.com/xwork/xwork-validator-
    1.0.2.dtd">

<validators>
    <field name="username">
        <field-validator type="requiredstring">
            <message>Debes introducir un usuario</message>
        </field-validator>
    </field>
    <field name="password">
        <field-validator type="requiredstring">
            <message>Debes introducir un password</message>
        </field-validator>
    </field>
</validators>
```

Es importante indicar que este archivo **debe estar situado en el mismo directorio** en que se encuentre el archivo *.class* correspondiente a la clase de acción.

Como vemos en el listado anterior, para cada campo añadiremos un elemento *<field>* en el que se indicará mediante el subelemento *<field-validator>* la regla que se quiere aplicar a través de su atributo *type*. Así pues, para cada elemento *<field>* habrá que añadir tantos *<field-validator>* como reglas se quieran aplicar al campo.

Cada regla llevará su propio mensaje de error asociado a través del elemento `<message>` aunque al igual que en el caso de Struts 1, los mensajes de error pueden estar definidos en un archivo `ApplicationResources.properties` con sus correspondientes claves. En este caso, el elemento `message` simplemente deberá indicar la clave del mensaje de error:

```
<field-validator type="requiredstring">
    <message key="password.nulo"/>
</field-validator>
```

Para forzar a que el usuario sea redireccionado de nuevo a la página de login en caso de incumplirse las reglas de validación, simplemente habría que añadir una entrada `<result>` cuyo valor de atributo *name* sea “input” en el archivo de configuración `struts.xml`:

```
<action name="Login" class="misclases.Login">
    <result>/pages/ok.jsp</result>
    <result name="error">/pages/error.jsp</result>
    <result name="input">/pages/login.jsp</result>
</action>
```

Para poder mostrar los mensajes de error en la página de login cuando el usuario sea redireccionado a ella al incumplir las reglas de validación, será necesario incluir el elemento `<s:actionerror/>` en la página.

Si queremos que la validación se realice en cliente, tan sólo será necesario añadir el atributo *validate* con el valor *true* al elemento `<s:form>` en el que se incluyen los elementos gráficos para la captura de los datos:

```
<s:form action="Login" method="post" validate="true">
:
</s:form>
```

La inclusión de este atributo en el formulario hará que se genere todo el JavaScript necesario para realizar la comprobación de los datos en cliente, incluyendo los cuadros de diálogo que se mostrarán al usuario con los mensajes de error cuando el resultado de la validación sea negativo.

Otros tipos de validadores que se pueden especificar en `field-validator`, además de *requiredstring*, son:

- **email.** Comprueba que el valor introducido en el campo corresponda con una dirección de correo válida.

- **int.** Comprueba que el valor introducido en el campo sea un número entero comprendido entre dos valores dados. Dichos valores tendrán que ser suministrados como parámetros de field-validator a través del elemento param. Por ejemplo, para comprobar que el valor de un campo llamado “código” está comprendido entre 10 y 20 definiríamos el siguiente validator sobre el campo:

```
<field name="codigo">
  <field-validator type="int">
    <param name="min">10</param>
    <param name="max">20</param>
    <message>fuera de rango</message>
  </field-validator>
</field>
```

8.6.3 Validación mediante Anotaciones

Una de las novedades introducidas en Struts 2 respecto a Struts 1 es la posibilidad de utilizar anotaciones para simplificar las tareas de configuración en una aplicación, **eliminando así la necesidad de utilizar ficheros de configuración XML** para realizar esas funciones.

Por ejemplo, en las aplicaciones que hacen uso de validadores, los archivos de configuración de tipo ClaseAccion-validation.xml pueden ser sustituidos por anotaciones que serán introducidas en la propia clase de acción.

Para este caso concreto de aplicaciones basadas en validadores, la utilización de anotaciones requiere la realización de dos operaciones previas durante la implementación de la clase Action:

- **Que la clase herede ActionSupport.** La clase ActionSupport, que se encuentra en el paquete com.opensymphony.xwork2, proporciona una implementación por defecto de diversas interfaces de soporte del API de Struts 2, entre ellas las relacionadas con la funcionalidad relativa a la validación de datos.
- **Importar el paquete com.opensymphony.xwork2.validator.annotations.** Este proporciona todas las anotaciones asociadas a cada uno de los validadores predefinidos de Struts 2.

Una vez realizadas las tareas anteriores, será **necesario incluir la anotación @Validation** delante de la declaración de la clase para que Struts 2 reconozca las anotaciones que incluiremos a continuación para la validación de los datos:

```
package misclases;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.validator.annotations.*;
@Validation
public class Login extends ActionSupport {
:
}
```

Después, para aplicar una validación sobre un determinado campo será suficiente con indicar la anotación asociada al validador delante del método *getXxx* que devuelve el dato a validar, indicando entre paréntesis el mensaje de error que se quiere mostrar al usuario cuando falle la validación del dato.

Por ejemplo, para aplicar el validador `RequiredStringValidator` sobre los campos “username” y “password”, el cual se encarga de comprobar que se ha introducido al menos un carácter en un campo, habría que definir la clase Login de la siguiente manera:

```
package misclases;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.validator.annotations.*;
@Validation
public class Login extends ActionSupport {
    private String username;
    private String password;
    public String execute() throws Exception {
        if(getPassword().equals("admin"))
            return SUCCESS; //constante definida
                                //en ActionSupport
        else
            return ERROR; //constante definida
                                //en ActionSupport
    }
    public void setUsername(String us){
        this.username = us;
    }
}
```

```
//anotación cadena requerida
@RequiredStringValidator(message="debe introducir un
                           usuario")

public String getUsername(){
    return this.username;
}

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

//anotación cadena requerida
@RequiredStringValidator(message="debe introducir un
                           password")

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

La utilización de anotaciones en el ejemplo anterior para definir las reglas de validación, hace que ya no sea necesario crear el archivo Login-validation.xml.

En el caso de que los mensajes se encuentren en un archivo de recursos, en vez de utilizar el atributo *message* en la anotación emplearemos *key*, mediante el cual especificaremos la clave asociada al mensaje de error:

```
@RequiredStringValidator(key="password.nulo")
public String getPassword(){
    return this.password;
}
```

Si se incumple alguno de los criterios de validación definidos en los diferentes métodos *get* a través de las anotaciones, el usuario será redirigido a la página de resultado indicada con el nombre “input”.

Si esta página es la misma que contiene los campos de recogida de datos, no será necesario utilizar el tag `<s:actionerror>` para mostrar los mensajes de error, puesto que las anotaciones de validación incorporan como funcionalidad implícita la generación de los mensajes de error sobre los campos asociados a cada método *get()*.

8.6.3.1 TIPOS DE ANOTACIONES DE VALIDACIÓN

Además de la anotación `@RequiredStringValidator` utilizada como ejemplo anteriormente, el paquete `opensymphony.xwork2.validator.annotations` incluye otras anotaciones con las que poder realizar las validaciones habituales durante la captura de datos a través de un formulario. Entre ellas destacamos las siguientes:

- **@IntRangeFieldValidator.** Comprueba que el valor introducido en un campo está dentro de un determinado rango numérico entero. A través de los atributos `min` y `max` se define dicho rango.
- **@DoubleRangeFieldValidator.** Comprueba que el valor introducido en un campo está dentro de un determinado rango numérico decimal. A través de los atributos `min` y `max` se define dicho rango.
- **@StringLengthFieldValidator.** Comprueba que la longitud del texto introducido en el campo se encuentra dentro de un determinado rango. Los atributos `minLength` y `maxLength` definen la longitud mínima y máxima de la cadena, respectivamente.
- **@EmailValidator.** Comprueba que el valor del campo se ajusta a un formato de dirección de correo electrónico válido.

Se pueden definir todas las anotaciones que se consideren necesarias sobre un determinado método `get()`.

PRÁCTICA 8.4. LISTADO DE LLAMADAS DE USUARIO

Descripción

En esta última práctica vamos a desarrollar una nueva versión de la aplicación del listado de llamadas de usuario que presentamos en la práctica 3.1 del Capítulo 3.

Como recordaremos, la aplicación consta de una página inicial de login donde se solicitan los credenciales del usuario al que, una vez validado se le da la opción de elegir el teléfono cuyo listado de llamadas quiere ver. La aplicación también incluye una página de registro para la inserción de nuevos usuarios (figura 44).

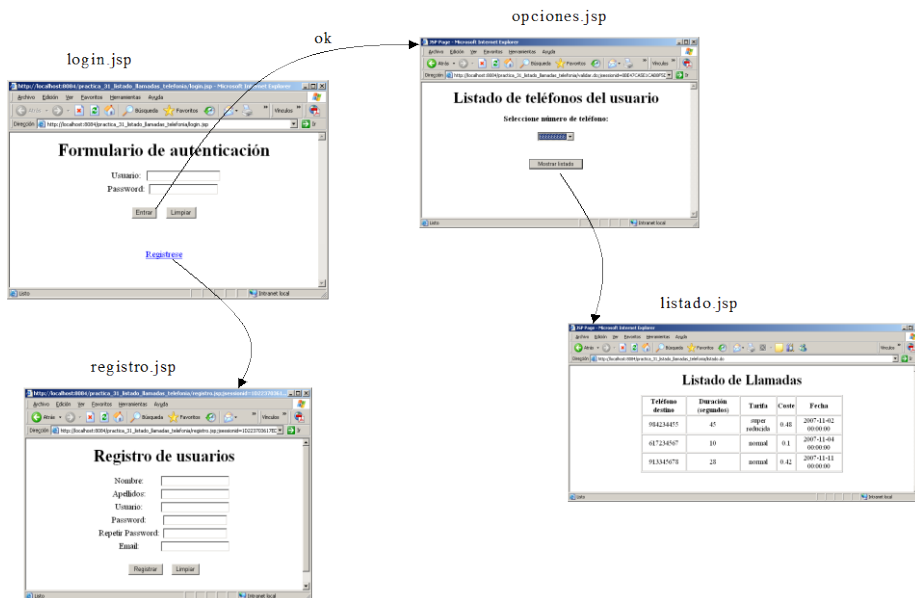


Fig. 44. Páginas de la aplicación

Desarrollo

El desarrollo de esta aplicación se realizará utilizando Struts 2. La lógica de negocio será la misma que en la versión anterior, con tres clases llamadas *GestionClientes*, *GestionTelefonos* y *GestionLlamadas*, en las que se incluirá todo el código de acceso a datos para la realización de las diferentes tareas requeridas por la aplicación.

Los beans *TarifaBean* y *LlamadaBean* que encapsulan los datos asociados a cada tipo de tarifa y llamada realizada son iguales también a los utilizados en la versión anterior.

Las tres operaciones a realizar por la operación, validar usuarios, registrar usuarios y listar las llamadas, serán controladas por tres clases de tipo *Action*. La navegación a la página de registro se realizará a través de una acción "Name".

Por otro lado, la aplicación realiza la validación de los datos suministrados a través del formulario de registro, para lo cual se incluirán las anotaciones pertinentes en el *Action* de registro.

Listado

Como ya hemos indicado, las clases de negocio y JavaBeans son los mismos que los desarrollados en la práctica 3.1, por lo que remitimos a dicha práctica para consultar los listados de código de estas clases.

Los siguientes listados corresponden a las tres clases Action de la aplicación:

Validar.java

```
package actions;

import java.util.ArrayList;
import org.apache.struts2.interceptor.*;
import org.apache.struts2.util.*;
import javax.servlet.http.*;
import javax.servlet.*;
import modelo.GestionClientes;
import modelo.GestionTelefonos;

public class Validar implements ServletContextAware{
    private String password="hola";
    private String username;
    private ArrayList<Integer> telefonos;
    ServletContext context;
    public String execute() throws Exception {
        String driver=context.getInitParameter("driver");
        String cadenaCon=context.getInitParameter("cadenaCon");
        GestionClientes gc=
            new GestionClientes(driver,cadenaCon);
        if(gc.validar(username,password)){
            GestionTelefonos gt=
                new GestionTelefonos(driver,cadenaCon);
            telefonos=gt.getTelefonos(password);
            return "ok";
        }
        else{
            return "error";
        }
    }
    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;
    }
    public ArrayList<Integer> getTelefonos() {
        return telefonos;
    }
    public void setTelefonos(ArrayList<Integer> telefonos) {
        this.telefonos = telefonos;
    }
    public void setServletContext(ServletContext context){
        this.context=context;
    }
}
```

Registrar.java

```
package actions;

import org.apache.struts2.util.*;
import javax.servlet.*;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.validator.annotations.*;
import modelo.GestionClientes;
@Validation
public class Registrar extends ActionSupport
    implements ServletContextAware{
    private String nombre;
    private String apellidos;
    private String password;
    private String usuario;
    private String email;
    ServletContext context;
    public String execute() throws Exception {
        String driver=context.getInitParameter("driver");
```

```
String cadenaCon=context.getInitParameter("cadenaCon");
GestionClientes gc=
    new GestionClientes(driver,cadenaCon);
gc.registrar(nombre,
    apellidos,
    usuario,
    password,
    email);
return "registrado";
}
//validación de cadena requerida y longitud
//mínima para el password
@RequiredStringValidator(message="debe introducir
                        un password")
@StringLengthFieldValidator(minLength="6",
    message="el password debe tener al menos 6 caracteres")
public String getPassword() {
    return password;
}
public void setPassword(String password) {
    this.password = password;
}
public String getApellidos() {
    return apellidos;
}
public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}
//validación de cadena requerida y valor
//válido para el email
@RequiredStringValidator(message="debe introducir
                        un email")
@EmailValidator(message="dirección de email no válida")
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public String getNombre() {
    return nombre;
```

```
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    //validación de cadena requerida para
    //el usuario
    @RequiredStringValidator(message="debe
                                introducir un usuario")
    public String getUsuario() {
        return usuario;
    }
    public void setUsuario(String usuario) {
        this.usuario = usuario;
    }
    public void setServletContext(ServletContext context){
        this.context=context;
    }
}
```

Listar.java

```
package actions;

import java.util.ArrayList;
import javabeans.LlamadaBean;
import javax.servlet.ServletContext;
import modelo.GestionLlamadas;
import org.apache.struts2.util.ServletContextAware;

public class Listar implements ServletContextAware{
    private int telefono;
    private ArrayList<LlamadaBean> llamadas;
    ServletContext context;
    public ArrayList<LlamadaBean> getLlamadas() {
        return llamadas;
    }
    public void setLlamadas(ArrayList<LlamadaBean> llamadas)
    {
        this.llamadas = llamadas;
    }
    public int getTelefono() {
```

```

        return telefono;
    }
    public void setTelefono(int telefono) {
        this.telefono = telefono;
    }
    public String execute() throws Exception {
        String driver=context.getInitParameter("driver");
        String cadenaCon=context.getInitParameter("cadenaCon");
        GestionLlamadas gl=
            new GestionLlamadas(driver,cadenaCon);
        llamadas=gl.getTodasLlamadasTelefono(telefono);
        return "llamadas";
    }
    public void setServletContext(ServletContext sc) {
        this.context=sc;
    }
}

```

Las vistas se han desarrollado utilizando los tags de Struts 2 comentados a lo largo del Capítulo. He aquí los listados de las mismas:

login.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
    HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1><br/><br/><br/><br/>
        <center>
            <h1>Formulario de Autenticación</h1>
            <s:form action="login.action" method="post" >
                <s:textfield name="username" label="usuario"/>
                <s:password name="password" label="password"
                    showPassword="false" />
            </s:form>
        </center>
    </body>
</html>

```

```
        <s:submit value="Entrar"/>
    </s:form>
</center>
<br/>
<br/>
<a href="Name.action">Registrese</a>
</body>
</html>
```

registro.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
    HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
    <head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Formulario de registro</h1>
        <s:form action="registrar.action" method="post">
            <s:textfield name="nombre" label="nombre"/>
            <s:textfield name="apellidos" label="apellidos"/>
            <s:textfield name="usuario" label="usuario"/>
            <s:password name="password" label="password" />
            <s:textfield name="email" label="email"/>
            <s:submit value="Entrar"/>
        </s:form>
        <br/>
        <s:actionerror />
    </body>
</html>
```

opciones.jsp

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
```

```

<!DOCTYPE      HTML      PUBLIC      "-//W3C//DTD      HTML      4.01
Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <title>Bienvenida</title>
</head>
<body>
<center>
    <h1>Listado de teléfonos</h1>
    <s:form action="listado">
        <s:select name="telefono" list="telefonos"
            headerValue="-seleccione un telefono-"
            headerKey="0"/>

        <s:submit value="Ver llamadas"/>
    </s:form>
</center>
</body>
</html>

```

listado.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE      HTML      PUBLIC      "-//W3C//DTD      HTML      4.01
Transitional//EN"

"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type"
        content="text/html; charset=UTF-8">
    <title>JSP Page</title>
</head>
<body>
    <h1>Llamadas realizadas</h1>
    <table border="1">
        <tr>
            <th>Destino</th>
            <th>Duración</th>
            <th>Fecha</th>
        </tr>
        <s:iterator value="llamadas">

```

```
        <tr>
            <td><s:property value="destino" /></td>
            <td><s:property value="duracion" /></td>
            <td><s:property value="fecha" /></td>
        </tr>
    </s:iterator>
</table>
</body>
</html>
```

En cuanto a los archivos de configuración, quedarán como se indica en los siguientes listados.

struts.xml

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

<struts>
    <constant name="struts.enable.DynamicMethodInvocation"
value="false" />
    <constant name="struts.devMode" value="false" />
    <package name="validacion" namespace="/"
        extends="struts-default">
        <action name="login" class="actions.Validar">
            <result name="ok">/opciones.jsp</result>
            <result name="error">/login.jsp</result>
        </action>
        <action name="registrar" class="actions.Registrar">
            <result name="registrado">/login.jsp</result>
            <result name="input">/registro.jsp</result>
        </action>
        <action name="listado" class="actions.Listar">
            <result name="llamadas">/listado.jsp</result>
        </action>
        <action name="Name" >
            <result>/registro.jsp</result>
        </action>
    </package>
</struts>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>driver</param-name>
    <param-value>com.mysql.jdbc.Driver</param-value>
  </context-param>
  <context-param>
    <param-name>cadenaCon</param-name>
    <param-value>
      jdbc:mysql://localhost:3306/telefonía
    </param-value>
  </context-param>
  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>
      org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <welcome-file-list>
    <welcome-file>login.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```


EL LENGUAJE DE EXPRESIONES DE JSP

El lenguaje de expresiones de JSP, más conocido como EL, es un lenguaje de programación basado en la utilización de expresiones dentro de una página JSP para la generación de resultados en el interior de la misma. Este lenguaje se incorporó a la especificación JSP a partir de la versión 2.0, la cual forma parte del conjunto de tecnologías JavaEE 1.4.

El objetivo de este lenguaje es reemplazar a las clásicas expresiones JSP basadas en la utilización de código Java, contribuyendo así a la reducción e incluso eliminación en algunos casos de la utilización de scriptlets Java dentro de una página JSP.

Mediante la utilización de sencillas instrucciones el lenguaje EL posibilita el acceso a los parámetros enviados en una petición, a las cookies o a los datos almacenados en cualquiera de los ámbitos de la aplicación.

Como otros lenguajes de programación, EL soporta la utilización de operadores y palabras reservadas.

Por sus características, el lenguaje EL constituye un excelente complemento de las acciones JSP de Struts para la creación de vistas, contribuyendo aún más a reducir la complejidad de las páginas JSP en las aplicaciones MVC.

EXPRESIONES EL

Las expresiones EL devuelven un valor al lugar de la página donde esté situada la expresión. Su sintaxis es:

`${expresion}`

donde *expresion* es cualquier expresión sintáctica válida EL que devuelva un resultado. Estas expresiones pueden incluir referencias a variables JSP, objetos implícitos EL, datos almacenados en cualquiera de los ámbitos de aplicación o incluso alguna operación entre datos utilizando alguno de los operadores soportados por EL.

Por ejemplo, siendo “midato” el nombre de alguna variable JSP, la siguiente instrucción mostraría en la página el valor de dicha variable:

```
El valor de la variable es ${midato}
```

Lo anterior es equivalente a:

```
El valor de la variable es <%=midato%>
```

pero con la ventaja en el caso de EL de no tener que utilizar scriptlets Java.

Es importante destacar que **una expresión EL no puede acceder a variables u objetos creados en scriptlets Java**. Por ejemplo, el siguiente bloque de sentencias no generaría ningún resultado en la página puesto que la variable “dato” no existiría para EL:

```
<%int dato=10;%>
resultado: ${dato}
```

Las expresiones EL pueden utilizarse bien para generar un texto dentro de la página de respuesta o bien para establecer el valor de un atributo de alguna acción JSP (incluidos los tags de Struts) que acepte expresiones. El siguiente ejemplo utiliza una expresión EL para obtener el valor a asignar a la variable “parametro”:

```
<bean:define id="parametro" value="${param['par']}" />
```

ACCESO A VARIABLES DE ÁMBITO

La recuperación de un dato almacenado en alguno de los ámbitos de la aplicación resulta tan simple con EL como indicar el nombre del dato entre los símbolos “\${” y “}”. Si el objeto es de tipo JavaBean el acceso a las propiedades de mismo se llevará a cabo utilizando la sintaxis:

objeto.propiedad

Por ejemplo, si tenemos un objeto de identificador “ValidacionForm” con dos propiedades, “usuario” y “password”, almacenado en un ámbito de sesión y desde una página JSP cualquiera de la aplicación quisiéramos obtener el valor de sus propiedades deberíamos utilizar las expresiones:

\${ValidacionForm.usuario}

y

\${ValidacionForm.password}

El siguiente bloque de instrucciones de ejemplo haría que se mostrase el texto “Mensaje de prueba” en la página de respuesta:

```
<jsp:useBean id="info" class="javabeans.Datos">
  <jsp:setProperty name="info"
    property="clave" value="Mensaje de prueba"/>
</jsp:useBean>
Mensaje: <h3>${info.clave}</h3>
```

Si el objeto es de tipo colección el acceso a sus elementos se realizará, en el caso de colecciones basadas en índices:

\${objeto_coleccion[indice]}

mientras que para colecciones basadas en claves será:

\${objeto_coleccion["nombre_clave"]}

En este último caso también es posible utilizar la siguiente expresión para acceder al valor del elemento de la colección:

\${objeto_coleccion.nombre_clave}

Por ejemplo, si tenemos una colección de tipo Map llamada “basedatos” en la que se almacenan los nombres de personas asociándoles como clave el DNI, la siguiente instrucción mostrará el nombre de la persona cuyo DNI sea 30005W:

```
Su nombre es <b>${basedatos["30005W"]}</b>
```

O también:

```
Su nombre es <b>${basedatos.30005W}</b>
```

OBJETOS IMPLÍCITOS EL

El lenguaje EL incluye una serie de objetos implícitos que permiten acceder de una forma sencilla a toda la información que los distintos objetos del API servlet proporcionan a la aplicación, como son las variables de página, petición, sesión o aplicación, los parámetros y encabezados de la petición, las cookies y los parámetros de contexto de inicialización.

Estos objetos son expuestos como colecciones de tipo Map, accediéndose a las propiedades y variables proporcionadas por éstos mediante la expresión:

```
${objeto_implicito[clave]}
```

siendo “clave” el nombre de la propiedad o variable cuyo valor se quiere obtener.

Al igual que con cualquier otro tipo de colección, también podríamos utilizar la siguiente expresión para acceder al valor del dato:

```
${objeto_implicito.clave}
```

Los objetos implícitos que forman parte del lenguaje EL son:

- **pageScope**. Proporciona acceso a las variables de ámbito de página.
- **requestScope**. Proporciona acceso a las variables de ámbito de petición. Por ejemplo, dado el siguiente bloque de sentencias incluidas en una página JSP:

```
<%int codigo=Math.ceil(Math.random()*500);  
    request.setAttribute("codigo", codigo);%>  
<jsp:forward page="prueba.jsp"/>
```

Si quisiéramos mostrar en la página prueba.jsp el valor de la variable de petición “codigo”, utilizaríamos:

El código generado es: `${requestScope["codigo"]}`

- **sessionScope.** Proporciona acceso a las variables de ámbito de sesión. Por ejemplo, supongamos que en una página JSP tenemos el siguiente bloque de instrucciones:

```
<jsp:useBean id="obj" class="javabeans.Datos"
            scope="session"/>
<jsp:setProperty name="obj"
                property="numero" value="35"/>
<%response.sendRedirect("prueba.jsp?par=35");%>
```

Por otro lado, en la página prueba.jsp queremos comparar el valor del atributo “par” con la propiedad “numero” del JavaBean de sesión, mostrando un mensaje en el caso de que sean iguales. Este será el bloque de código que tendremos que incluir en la página suponiendo que estamos utilizando Struts:

```
<logic:equal parameter="par"
             value="${sessionScope['obj'].numero}">
    <h1>La condición se cumple!</h1>
</logic:equal>
```

- **applicationScope.** Proporciona acceso a las variables de ámbito de aplicación.
- **param.** Mediante este objeto tenemos acceso a los parámetros enviados en la petición. Por ejemplo, el siguiente bloque de sentencias inicializaría la propiedad “nombre” del JavaBean “usuario” con el valor del parámetro “user” recibido en la petición:

```
<jsp:useBean id="usuario" class="javabeans.Usuario"/>
<jsp:setProperty name="usuario" property="nombre"
                value="${param['user']}" />
```

- **paramValues.** Al igual que el anterior proporciona acceso a los parámetros de petición, sólo que en este caso el valor de cada parámetro se recupera como un array de cadenas. Se utiliza en los casos en que el parámetro incluye múltiples valores.

- **header.** Proporciona acceso a los encabezados de la petición. El siguiente ejemplo mostraría en la página el valor del encabezado *user-agent* enviado en la petición actual:

```
Tipo navegador: ${header['user-agent']}
```

- **headerValues.** Al igual que header proporciona acceso a los encabezados de la petición, devolviendo en este caso un array de cadenas con todos los valores que le han sido asignados al encabezado.
- **cookie.** Proporciona acceso a las cookies enviadas en la petición. Cada elemento de la colección Map asociada representa un objeto cookie. La siguiente expresión de ejemplo mostraría en la página el contenido de la cookie “user”:

```
Usuario: ${cookie["user"]}
```

Además de los anteriores objetos Map, el lenguaje EL proporciona otro objeto implícito llamado **pageContext** que proporciona acceso al contexto de la aplicación. Entre otras dispone de una serie de propiedades que nos dan acceso a los objetos implícitos JSP, por ejemplo, la siguiente expresión EL permitiría recuperar el método de envío utilizado en la petición actual:

```
${pageContext.request.method}
```

OPERADORES EL

El lenguaje EL también incluye una serie de operadores que permiten manipular datos dentro de una expresión. La tabla de la figura 45 contiene los principales operadores EL agrupados por categorías. Para algunos de ellos se indica entre paréntesis otro símbolo o nombre alternativo a utilizar.

Categoría	Operadores
Aritméticos	+, -, *, / (div), % (mod)
Relacionales	==(eq), !=(ne), <(lt), >(gt), <=(le) y >=(ge)
Lógicos	&&(and), (or) y !(not)

Fig. 45. Tabla de operadores EL

Por otro lado, la tabla indicada en la figura 46 incluye algunos ejemplos de expresiones EL que utilizan algunos de los operadores anteriores, indicando el resultado generado en cada caso.



Fig. 46. *Ejemplos de expresiones EL con operadores*

Además de estos operadores clásicos, EL incluye otros dos operadores especiales:

- **empty.** Comprueba si una colección o cadena es vacía o nula, devolviendo el valor *true* en caso afirmativo y *false* si no lo es. Por ejemplo, la siguiente expresión devolvería *true* si no se ha recibido ningún valor en el parámetro “password”:

```
${empty param["password"]}
```

- **Operador condicional.** El operador condicional “?:” funciona de forma muy similar al operador Java del mismo tipo, siendo su sintaxis de utilización:

```
${condicion? expresionA:expresionB}
```

Si la condición da como resultado el valor *true* se evaluará *expresionA*, mientras que si el resultado es *false* se ejecutará *expresionB*.

El siguiente ejemplo devolverá el valor de la variable de sesión “contador” en caso de que no exista la cookie “user”, generando el valor “0” en caso contrario:

```
${empty cookie["user"]?sessionScope["contador"]:0}
```


LA LIBRERÍA DE ACCIONES ESTÁNDAR DE JSP (JSTL)

La librería de acciones JSTL consta de un conjunto de tags JSP que permiten realizar tareas habituales de procesamiento en una página JSP sin necesidad de recurrir a scriptlets Java.

La mayor parte de la funcionalidad proporcionada por las acciones *logic* y *bean* de Struts puede ser conseguida con las acciones estándares JSTL, resultando incluso más sencilla la utilización de estos tags que los de Struts. Por ello es común combinar ambos tipos de etiquetas en la construcción de vistas para las aplicaciones Struts.

Las acciones JSTL están pensadas para utilizar el lenguaje EL en la definición de los valores de sus atributos:

```
<c:set var="info" value="${dato}"/>
```

Durante este último apéndice estudiaremos las acciones más importantes de esta librería y los pasos a seguir para su utilización.

INSTALACIÓN DE JSTL

Aunque es soportada desde la versión JSP 1.2, no ha sido hasta la versión JavaEE 5.0 (JSP 2.1) cuando JSTL se ha integrado en la plataforma JavaEE. No obstante, podemos descargarla de forma independiente desde la dirección:

<http://java.sun.com/products/jsp/jstl/>

Dos son los archivos que proporcionan todo el soporte para la utilización de JSTL: **jstl.jar** y **standard.jar**, ambos deberán ser incluidos en el directorio WEB-INF\lib de la aplicación.

Probablemente, si estamos utilizando algún entorno de desarrollo con soporte para JavaEE 5, como Eclipse o NetBeans, los componentes de la librería se incluyan en otros archivos .jar diferentes a éstos y sean incorporados de forma automática al proyecto por el asistente de creación del mismo.

UTILIZACIÓN DE ACCIONES JSTL

Una vez desplegados los archivos .jar en el directorio de librerías podemos hacer uso de las acciones JSTL desde cualquiera de las páginas JSP de la aplicación.

El conjunto de acciones JSTL está compuesto realmente por cinco librerías, de las cuales es la librería core la que proporciona el grupo de acciones de uso general, que son en definitiva las de más amplia utilización y sobre las que nos centraremos en este estudio.

Para poder utilizar las acciones core en una página JSP es necesario incluir la siguiente directiva *taglib* en la página:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Como vemos, suele ser convenio utilizar el prefijo “c” para referirse a las acciones del core.

ANÁLISIS DE LAS PRINCIPALES ACCIONES JSTL

A lo largo de esta sección vamos a analizar las principales acciones JSTL que componen la librería core. Para facilitar nuestro estudio vamos a dividir este conjunto de acciones en dos grupos, según la funcionalidad proporcionada por las mismas:

- Acciones genéricas.
- Acciones de control de flujo.

Acciones genéricas

Incluye acciones de uso general en la realización de tareas habituales en una página JSP, como la inserción de datos en la página de respuesta, manipulación de variables, etc.

Veamos a continuación las acciones más importantes de este grupo.

OUT

Realiza la misma función que una scriptlet JSP del tipo:

```
<%=expresion%>
```

También es equivalente a la acción Struts `<bean:write/>`, es decir, inserta un determinado valor en la página de respuesta. Su principal atributo, `value`, contiene la expresión EL que determina el valor a incluir en la página. El siguiente ejemplo mostraría una frase con el valor del parámetro “email” enviado en la petición:

```
La dirección de correo electrónico es:  
<c:out value="${param['email']}"/>
```

Se puede indicar mediante el atributo `default` un valor por defecto para mostrar en la página en caso de que no se localice el dato indicado en `value`:

```
La dirección de correo electrónico es:  
<c:out value="${param['email']}"  
      default="default@direccion.com"/>
```

SET

Se emplea para establecer un valor en una variable JSP, en una propiedad de un JavaBean o en una colección de tipo Map. Sus atributos son:

- **var.** Identificador de la variable JSP a la que se asignará el valor.
- **target.** En caso de tratarse de una asignación a un objeto (JavaBean o colección), esta propiedad indicará el identificador asociado a dicho objeto.
- **property.** Propiedad del objeto JavaBean a la que se le asignará el valor. Si `target` especifica una colección Map en vez de un objeto

JavaBean, property contendrá la clave asociada al dato que se va a añadir a la colección.

- **scope.** Ámbito de contexto en el que se definirá la variable. Sus posibles valores son: *page*, *request*, *session* y *application*.
- **value.** Valor que se asignará a la variable, propiedad de JavaBean o colección, según cuáles de los atributos anteriores se hayan especificado.

La siguiente instrucción asigna el valor 40 a la variable “res”:

```
<c:set var="res" value="${5*8}"/>
```

En este otro ejemplo se asigna el valor del parámetro “telefono” a la propiedad “contacto” del JavaBean “persona”:

```
<c:set target="persona"
      property="contacto" value="${param['telefono']}"/>
```

Finalmente, en este otro ejemplo se asigna una cadena de caracteres existente en el atributo de sesión “user” en el elemento de la colección “usuarios” que tiene como clave asociada “5555J”:

```
<c:set target="usuarios" property="5555J"
      value="${sessionScope['user']}"/>
```

REMOVE

Elimina una variable existente en uno de los ámbitos de la aplicación. Sus atributos son:

- **var.** Identificador de la variable a eliminar.
- **scope.** Ámbito donde se encuentra la variable. Si no se especifica se buscará en todos los ámbitos de la aplicación.

CATCH

Permite capturar una excepción dentro de la página JSP. El formato de utilización de la acción es el siguiente:

```
<c:catch var="variable">
```

acciones posible excepción

```
</c:catch>
```

Si se produce alguna excepción en el cuerpo de la acción `<c:catch>` se almacenará el objeto `Exception` generado en la variable cuyo nombre se indica en el atributo `var` del tag, interrumpiéndose la ejecución del bloque y pasando el control del programa a la primera sentencia después del `catch`.

REDIRECT

Realiza la misma función que el método `sendRedirect()` del objeto `HttpServletResponse`, redireccionando al usuario hacia un nuevo recurso cuya dirección estará especificada en el atributo `url` de la acción.

Opcionalmente se pueden enviar parámetros al destinatario, para lo cual habrá que utilizar la acción `<c:param>` en el interior de `<c:redirect>`. El bloque de acciones del siguiente ejemplo redireccionaría al usuario al servlet “entrar”, pasándole en la petición los parámetros “codigo” y “localizacion”:

```
<c:redirect url="entrar">
    <c:param name="codigo" value="97811"/>
    <c:param name="localizacion" value="JAZ10"/>
</c:redirect>
```

Control de flujo

Las acciones incluidas en este grupo permiten controlar el flujo de ejecución dentro de las páginas JSP.

Seguidamente presentaremos las acciones más importantes de este grupo.

IF

Evalúa el cuerpo de la acción si el resultado de la condición indicada en su atributo `test` es *true*. El código del siguiente ejemplo mostraría en la página de respuesta el valor de la variable “num” en caso de que éste sea un número par:

```
<c:if test="${num%2 == 0}">
    El número es <c:out value="${num}"/>
</c:if>
```

```
</c:if>
```

CHOOSE

Su formato es similar al de la acción `<c:if>`, sólo que en este caso se comprueban varias condiciones, evaluándose el cuerpo de la primera que resulte verdadera. La estructura de `<c:choose>` es:

```
<c:choose>
    <c:when test="condicion1">
        cuerpo1
    </c:when>
    <c:when test="condicion2">
        cuerpo2
    </c:when>
    :
    <c:otherwise>
        otros
    </c:otherwise>
</c:choose>
```

El siguiente ejemplo muestra distintos mensajes en la página de respuesta en función del valor de la variable hora:

```
<c:choose>
    <c:when test="{hora}>8 && hora<13">
        Buenos días
    </c:when>
    <c:when test="{hora}>13 && hora<20">
        Buenas tardes
    </c:when>
    <c:otherwise>
        Buenas noches
    </c:otherwise>
</c:choose>
```

FOREACH

Procesa de forma repetitiva el cuerpo de la acción. Se puede utilizar de dos formas posibles:

1. **Recorrido de un rango de valores numérico.** En este caso la variable especificada en su atributo *var* es inicializada al valor numérico indicado en el atributo *begin*, evaluándose el cuerpo de la acción hasta que la variable alcance el valor indicado en el atributo *end*. El valor de incremento de la variable al final de cada iteración deberá ser especificado mediante el atributo *step*.

El siguiente bloque de acciones mostraría en la página de respuesta la tabla de multiplicar del número 7:

```
<table border="1">
<c:foreach var="i" begin="1" end="10" step="1">
    <tr><td><c:out value="${7*i}"/></td></tr>
</c:foreach>
</table>
```

2. **Recorrido de una colección.** En este caso la variable indicada en *var* recorrerá la colección especificada en el atributo *items*, apuntado con la variable indicada en *var* a cada elemento de la colección. El cuerpo de la acción será evaluado con cada iteración. El siguiente ejemplo mostrará en la página el contenido de la colección “nombres”:

```
<c:forEach var="nombre" items="${nombres}">
    <c:out value="${nombre}"/>
</c:forEach>
```

FORTOKENS

Esta acción se utiliza para recorrer una lista de objetos *String* o *tokens*, integrados en una cadena de caracteres. Dispone de los siguientes atributos:

- **items.** Cadena de caracteres cuyo contenido se va a recorrer.
- **var.** Variable que apuntará en cada iteración a uno de los *token* de la cadena.
- **delims.** Carácter utilizado como separador de *token*.

En la siguiente página JSP de ejemplo se recorre una cadena de caracteres formada por una lista con los nombres de los días de la semana y muestra su contenido en una tabla XHTML dentro de la página de respuesta:

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head></head>
<body>
    <c:set var="dias" value="lunes, martes,
        miércoles, jueves, viernes, sábado, domingo"/>
    <table border="1">
    <c:forEach var="dia" items="{dias}" delims=",">
        <tr>
            <td>
                <c:out value="{dia}"/>
            </td>
        </tr>
    </c:forEach >
    </table>
</body>
</html>
```


ÍNDICE ALFABÉTICO

@

@ actionerror.....	293
@DoubleRangeFieldValidator	294
@EmailValidator.....	294
@IntRangeFieldValidator	294
@RequiredStringValidator	292
@StringLengthFieldValidator	294
@validation	292

A

acciones JSTL.....	313
action	64, 134, 243
Action	40, 226
<i>action</i> , atributo.....	70, 273
Action, objetos Struts 2	226
ActionContext	259
actionerror	290
actionerror, acción	286
ActionErrors	127, 128, 135
ActionForm	40, 43, 125, 127, 171
ActionForward.....	41, 44, 74
ActionInvocation	260
ActionMapping	40
ActionMessage	128, 129, 135
ActionMessages.....	196

<i>actions</i> , paquete.....	91
ActionServlet.....	39, 43, 87
<i>add</i> , método	128
<i>addActionError</i> , método	286
AJAX	238
anotaciones	238
anotaciones, validación	291
ApplicationAware	249
ApplicationResource.properties .	176, 187
ApplicationResources.properties...	38
applicationScope	309

B

bean	
cookie	145
define.....	147
header	146
message	146
page	149
parameter.....	145
size.....	149
write.....	144
Bean	42
bean, acción.....	269
<i>bean</i> , librería	143

bean:write 72
 byte, validador 185

C

checkbox, acción 279
 choose, acción 318
cols, atributo 275
 Controlador 21
 controller 89
 cookie 310

D

defaultStack 248
 definición 209
 definition 210, 211
definition, atributo 212
destroy, método 258
 DispatchAction 92
 double, validador 185
 doubleRange, validador 186

E

EL, expresiones 306
 EL, lenguaje de expresiones 305
 EL, objetos implícitos 308
 EL, operadores 310
 EL, variables 307
 else, acción 271
 email, validador 187
 empty, operador EL 311
 excepciones 133
 exception 134
 ExceptionHandler 139
execute, método 40, 41, 44, 50, 59,
 62, 89, 92, 93, 106, 126, 139, 226,
 242, 249, 258

F

field 175, 183, 289
 Field 196

field-validator 289
fieldValue, atributo 279
 FileUploadInterceptor 246
 FilterDispatcher 224, 241
findForward(), método 60
 float, validador 185
 foreach, acción 319
 form 175
 form, acción 273
 form-bean 55
 form-beans 55
 formset 178
 form-validation 178
 forward 64
forward, atributo 69
 FrontControler 39

G

get, método 259
 getAction, método 259
 getApplication, método 260
getInputForward(), método 60
 getInvocationContext, método 259
getKeyMethodMap, método 106
 getParameters, método 260
getPathInfo, método 24
 getSession, método 260
getValueAsString, método 197
 global-exception 134
 global-forward 64

H

handler, atributo 135, 139
 header 310
headerKey, atributo 279
headerValue, atributo 279
 headerValues 310
href, atributo 70, 245
 HTML 41
 html:checkbox 69
 html:errors 127

html:form	67
html:html	66
html:link	69
html:option	68
html:password	68
html:radio	69
html:select	68
html:submit	68
html:text	67
html:textarea	68
HttpServletRequest	196

I

if, acción	271, 317
include	237
init, método	258
init-param	52
input, atributo	63
integer, validador	185
intercept, método	258, 260
interceptor	246
Interceptor, interfaz	258
interceptores	225, 245
interceptores personalizados	258
interceptor-ref	247
interceptor-stack	247
intRange, validador	185
invoke, método	260
iterator, acción	271

J

J2EE	21, 23, 37
JavaBean	24
JSTL, acciones de control de flujo	317
JSTL, acciones genéricas	315
JSTL, instalación	313
JSTL, librería de acciones	313
JSTL, utilización	314

K

key, atributo	135
---------------------	-----

L

label, atributo	274
lenguaje de expresiones	305
linkName, atributo	70
list, atributo	279
listKey, atributo	278
listValue, atributo	278
logic	
equal	151
forward	155
greaterEqual	154
greaterThan	154
iterate	157
lessEqual	154
lessThan	154
match	154
noMatch	155
notEqual	154
redirect	155
Logic	42
logic, librería	150
long, validador	185
LookupDispatchAction	106

M

MappingDispatchAction	113
mask, validador	187
maxlength, atributo	67
maxlength, validador	184
message	290
method, atributo	67, 273
minlength, validador	183
Modelo	22
Modelo Vista Controlador	20, 22, 35
msg	188
msg, atributo	174, 200
MVC	223

N

name, atributo 63, 64, 273, 279
namespace, atributo 273
 Nested 42

O

operador condicional EL 311
 operadores EL 310
 out, acción 315

P

package 228
page, atributo 207
 pageScope 308
 página JSP 22
 param 309
 param, acción 269
parameter, atributo 114
parameter, método 94
 ParameterAware 249
 paramValues 309
 password, acción 275
path, atributo 63, 135
 plantillas 39, 201
 POJO 226, 237, 248
 PrincipalAware 249
processActionCreate, método 89
processActionForm, método 88
processActionPerform, método 89
processForwardConfig, método 89
processMapping, método 88
processorClass, atributo 90
processPath, método 88
processPopulate, método 88
processPreprocess, método 88
processValidate, método 88, 126
 property 245
 property, acción 269
property, atributo 67, 128
 push, acción 270
 put 211

R

radio, acción 277
readonly, atributo 68, 274
 redirect, acción 317
 remove, acción 316
 RequestAware 249
 RequestProcessor 87, 126, 202
 requestScope 308
 requiredstring 290
reset, método 55, 126
 result 243, 290
 rows, atributo 275

S

scope, atributo 63
 select, acción 279
sendRedirect, método 317
 servlet 21
 servlet-class 52
 ServletConfigInterceptor 249, 250
 ServletContextAware 249
 ServletRequestAware 249
 ServletResponseAware 249
 SessionAware 249
 sessionScope 309
 set, acción 270, 315
setServletRequest, método 250
 short, validador 185
showPassword, atributo 275
 Struts 36, 37
 Struts 2 223
 Struts 2, archivo de
 configuración 227
 Struts 2, clases de acción 226
 Struts 2, componentes 224
 Struts 2, estructura de
 aplicaciones 240
 Struts 2, interceptores 225, 245
 Struts 2, librería de acciones 267
 Struts 2, librerías de acciones 227
 Struts 2, requerimientos 239

Struts 2, validadores 286
 Struts, API 39
 Struts, archivo de configuración.... 37
 Struts, componentes..... 37
 Struts, control de excepciones 139
 Struts, estructura de aplicaciones... 51
 Struts, instalación 45
 Struts, librería de acciones..... 143
 Struts, librerías..... 41
 Struts, tiles 201
 Struts, validadores 171
 struts.xml 226, 227
 struts-config.xml..... 37, 39
 struts-default.xml..... 229
 StrutsStatics 259
 submit, acción..... 275

T

taglib, directiva..... 65, 144, 150, 203,
 267, 314
 textarea, acción 275
 textfield, acción 273
 tiles 201
 Tiles 42
 tiles, librería..... 203
 tiles:getAsString 204
 tiles:insert 203, 207, 212
 tiles:put 208
 tiles-definitions 210
 tiles-defs.xml 39, 210
try-catch..... 133

type, atributo..... 63, 135, 289

U

uri, atributo..... 65
 url-pattern 52

V

validadores 171, 183
 validadores predefinidos 193
 validadores Struts 2 286
validate, atributo 63, 290
validate, método... 55, 126, 171, 175,
 193
 validation.xml..... 39, 172, 286
 validator..... 198
 Validator..... 196
 ValidatorAction..... 196
 ValidatorForm 175, 178, 193
 validator-rules.xml 39, 172
value, atributo..... 274
 var..... 183
 var-value..... 184
 Vista 22

W

WebWork 223

X

XHTML..... 22

TEXTO CONTRAPORTADA

Título del libro

La creciente demanda de nuevas funcionalidades y servicios requeridos a las aplicaciones Web modernas se está traduciendo en un aumento de la complejidad de los desarrollos, haciendo indispensable en la mayoría de los casos el empleo de algún tipo de utilidad que facilite la tarea del programador.

En este contexto se enmarca Struts, sin lugar a dudas el marco de trabajo más popular para la construcción de aplicaciones Web con tecnologías JavaEE. Su metodología de desarrollo y el amplio conjunto de utilidades y componentes que proporciona permite crear en un corto espacio de tiempo complejas aplicaciones Web totalmente robustas y escalables, lo que se ha traducido en el hecho de que Struts sea el framework preferido por la comunidad de desarrolladores JavaEE para la construcción de sus aplicaciones.

En este libro se analiza tanto la metodología de trabajo de Struts como los distintos elementos del framework, incluyendo las innovadoras adaptaciones del mismo que nos ofrece la última versión de Struts, conocida como Struts 2. Todo ello abordado de una manera didáctica y completado con numerosos ejemplos y prácticas que además de ayudar al lector a comprender los conceptos le ilustrará sobre todas las posibilidades que ofrecen los diferentes componentes del framework.

Además de constituir una guía útil para profesionales y conocedores de la plataforma JavaEE, el desarrollo secuencial y ordenado de los temas del libro facilita su uso como manual de estudio en cursos de formación donde se imparta esta materia.

.....