Struts 2

El framework de desarrollo de aplicaciones Java EE

Este libro sobre Struts 2 se dirige a los **desarrolladores Java** que quieren tener un libro de referencia para la utilización del **framework de Java EE**más popular. El libro está dividido en 23 capítulos que explican el**funcionamiento** y la **ejecución de proyectos Web** a partir del **framework**.
Los primeros capítulos describen el **framework de Java EE** de referencia con sus **servicios** y su **instalación** a través del **modelo de diseño MVC**. Los capítulos 3 y 4 presentan un ejemplo concreto de **proyecto de Struts 2**para **gestionar los registros y la depuración. En el siguiente capítulo, el lector podrá conocer en detalle la administración de las acciones**, el**mapping**, los **formularios** y las **redirecciones**. El capítulo 6 presenta de manera extensa la **biblioteca de etiquetas de Struts**.
En el resto del libro, el lector aprenderá a administrar los **mensajes** y la**internacionalización**, así como la **validación de las entradas**, los **tipos** y las**conversiones**. Un capítulo está dedicado al modelo de **acceso a los datos**, a la **carga** y **descarga** de datos y a la **carga de las páginas**.
El desarrollo de los **interceptores** se detalla ampliamente en el capítulo 14, así como la gestión de los **resultados** en el capítulo 15.
Los siguientes capítulos están dedicados a **Ajax** Struts y a los **motores de plantillas**, así como a la visualización de la información a través de **XSLT**. Los últimos capítulos se refieren a la utilización y el desarrollo de**complementos** con Struts, así como a la **configuración cero** y el lenguaje**OGNL**.
Cada concepto se expone de forma teórica y técnica para permitir a los diseñadores que tengan conocimientos en Java EE **utilizar una API** que facilita el desarrollo de aplicaciones Web.
Las aplicaciones utilizadas en los capítulos proceden de **ejemplos concretos**y se pueden **descargar** en esta página.

Jérôme LAFOSSE

Ingeniero en informática y graduado en la CNAM, **Jérôme Lafosse** trabaja como asesor, diseñador y formador en las tecnologías Java. Especialista en tecnologías Web, trabaja para fomentar las herramientas y soluciones de código abierto (Open Source) para el desarrollo de proyectos de Internet. También enseña la plataforma Java Enterprise Edition y el diseño de proyectos Web en Licenciaturas y Masters.

¿Qué es un framework?

En programación existen dos tipos de individuos:

- Los programadores de sistemas.
- Los programadores de aplicaciones.

Los programadores de sistemas escriben el código que utilizarán los programadores de aplicaciones. Los programadores de sistemas desarrollan los lenguajes Java, PHP, C o C++ y los programadores de aplicaciones utilizan estos lenguajes y herramientas para crear valor añadido con fines comerciales. Los programadores de aplicaciones se concentran en sus proyectos sin preocuparse de las técnicas y las mecánicas de bajo nivel. Los programadores de aplicaciones utilizan una serie de bibliotecas o herramientas que reciben el nombre de framework.

Un framework es un conjunto de bibliotecas, herramientas y normas a seguir que ayudan a desarrollar aplicaciones. Los frameworks los desarrollan los programadores de sistemas. Un framework está compuesto por varios segmentos/componentes que interactúan los unos con los otros. Las aplicaciones pueden escribirse de manera más eficaz si utilizamos un framework adaptado al proyecto en lugar de tener que volver a inventar la rueda cada vez. Un framework Java proporciona un conjunto de características a partir de una implementación de objeto. En proyectos de desarrollo a gran escala y de diseño en equipo, los frameworks son muy útiles, incluso imprescindibles.

En la actualidad, existen diferentes tipos de framework:

- los frameworks de infraestructura de sistema, que permiten desarrollar sistemas de explotación, herramientas gráficas y plataformas Web (Struts, Spring...);
- los frameworks comunicativos (llamados software);
- los frameworks de empresa (desarrollos específicos);
- los frameworks de gestión de contenido (tipo *Content Management System*).

Los frameworks permiten la reutilización de código, la estandardización del desarrollo y la utilización del ciclo de desarrollo de tipo interactivo-incremental (especificación, codificación, mantenimiento y evolución). En ocasiones hablamos de paquetes de programas evolucionados cuando diseñamos un framework y su ciclo de vida. En la actualidad, existen muchos frameworks en todos los dominios de aplicación y en prácticamente cualquier idioma. Esta es una lista no exhaustiva de los frameworks utilizados en Java:

- Apache Struts
- WebWork
- JSF (*Java Server Faces*)
- Spring
- Wicket

¿Por qué utilizar un framework?

Los Servlets se definieron en 1998 y dos años después, algunas grandes empresas ya habían apostado por Java para sus aplicaciones Web. Durante varios años, estas empresas desarrollaron sus propios proyectos de manera independiente y sin seguir ningún estándar. Hoy en día, todas estas sociedades tienen en cuenta la importancia de los frameworks. La elección del framework de desarrollo forma parte de la estrategia de una empresa ya que será determinante para la calidad, la productividad y la durabilidad de los proyectos.

1. Normas y estándares

El desarrollo de las TI con la utilización de normas permite generalizar las buenas prácticas y armonizar los desarrollos dentro de la empresa, facilitando el mantenimiento. La plataforma de desarrollo Java EE permite utilizar normas, además de complejas herramientas que vienen también acompañadas de otras normas.

2. Framework y desarrollo Web

La definición inicial del API Servlet se queda corta a la hora de abarcar un desarrollo complejo de aplicaciones totalmente basadas en Servlets. Al principio, las aplicaciones Java se basaban en el principio del API Common Gateway Interface (CGI) y progresivamente fueron apareciendo los frameworks de Java para cubrir las carencias o debilidades del API Servlet y JavaServer Pages (JSP). La elección del API tendrá un impacto significativo en el rendimiento, el funcionamiento, la calidad y el mantenimiento de la aplicación. Del mismo modo, puesto que el framework constituirá los cimientos sobre los que se construirá el software, su durabilidad será también fundamental.

Los distintos tipos de framework

Existen varios tipos de herramientas para el desarrollo de aplicaciones. Un framework de tipo "interno", es decir, desarrollado por la empresa, no es la mejor solución. Desde los primeros años de Java, los equipos de informáticos inventaron sus propias herramientas para el desarrollo y las grandes empresas, en ocasiones, creaban su propio framework. Hay que evitar este tipo de desarrollos, ya que ninguna empresa podrá dedicar suficientes esfuerzos para el mantenimiento y la evolución del framework. Además, los frameworks de código abierto (Open Source) se convierten en estándares, se prueban y validan a escala mundial a través de los proyectos realizados.

Los frameworks de editor presentan un riesgo para las empresas desde un punto de vista de desarrollo. De hecho, siempre tienen un objetivo oculto, que es la fidelización de la empresa a las herramientas del editor.

Los frameworks de código abierto (Open Source) son en la actualidad los más numerosos y los que más éxito tienen. Aquí, nos volvemos a encontrar con la calidad del trabajo y la misma dinámica que el proyecto Apache. Una buena parte de estos proyectos de framework nace del consorcio Apache. Los frameworks son herramientas complejas independientemente de la calidad del desarrollo y del origen de los proyectos. No es necesario dominar todos los frameworks existentes, pero sí es necesario saber utilizarlos correctamente. Una vez se ha elegido el framework, es necesario formarse y constituir una unidad de asistencia para los equipos de desarrollo.

¿Qué framework elegimos?

Poco a poco, las API y herramientas de todo tipo han ido sumergiendo el desarrollo de Internet basado en la tecnología Java. La elección de un framework debe basarse en los siguientes criterios:

- ¿Debemos diseñar todo de principio a fin?
- ¿El desarrollo permite el uso de una aplicación previamente desarrollada o de una parte?
- ¿Podemos usar un entorno como base para la aplicación?

El diseño de principio a fin permite dominar perfectamente una tecnología, pero requiere de mucho tiempo y dinero. El desarrollo a partir de aplicaciones existentes es interesante únicamente si los desarrolladores de los proyectos anteriores están presentes. El tercer enfoque (usar un entorno como base para la aplicación) es sin duda la mejor opción en la mayoría de los casos.

Introducción a la programación en Java Entreprise Edition

Las tecnologías Servlets y JavaServer Pages (JSP) son los cimientos del desarrollo en Java EE. El problema con estas tecnologías es la cantidad de código que es necesario desarrollar para las comunicaciones HTML/JSP, Servlets y Modelos. De igual modo, sin usar un modelo de diseño de tipo Modelo Vista Controlador (MVC), mezclar scripts HTML, SQL y Java es una mala idea ya que la depuración es entonces más compleja y lleva más tiempo. La mezcla de código evita la reutilización y la visibilidad de las estructuras de control, haciendo confusa la presentación y el acceso a los datos.

Por último, el uso de objetos JavaBeans y de administradores de etiquetas como JSTL *(Java Standard Tag Library)* permite un desarrollo simple y consistente, incluso para proyectos complejos. La escritura de código HTML/XHTML en páginas JSP es muy rápida para el desarrollador. De hecho, los JSP no reemplazan a los Servlets sino que son más bien complementarios.

Con un modelo de sitio muy simple realizado completamente en JSP, las páginas se compilan y se transforman en Servlets. Este es el modelo más utilizado por los programadores principiantes debido a su simplicidad y rapidez de aplicación.

Con un modelo de diseño un poco más complejo, los desarrolladores usan los Servlets para los procesamientos y las páginas JSP para la presentación. Este modelo es más difícil de aplicar, pero también es el más natural y el que ofrece un mejor mantenimiento.

Por último, para una arquitectura multinivel de tipo MVC, los Servlets representan el aspecto de Control, los Modelos el acceso a los datos y los JPS la parte Vista. Este modelo es más complejo de desarrollar, pero mucho más simple de probar, mantener y hacer evolucionar. Del mismo modo, cabe destacar el concepto de reutilización con este tipo de modelo.

Struts 1

El proyecto Open Source Jakarta-Struts desarrollado por el consorcio Apache permite agilizar el desarrollo de aplicaciones de Internet. Struts 1 casi se ha convertido en el estándar de facto para los proyectos Java EE. Struts 1 es un entorno agradable y potente que gestiona la aplicación así como las tareas más frecuentes (enrutamiento, acciones, validaciones...). Otra ventaja de su uso es el número creciente de usuarios que tienden a perpetuar el proyecto. En la actualidad, muchos entornos de desarrollo como Eclipse ofrecen herramientas para la programación de Struts 1.

Struts 1 es un framework que ofrece herramientas de validación de las entradas de los usuarios (introducidas mediante un formulario), bibliotecas de etiquetas JSP para la creación rápida de páginas, una técnica de enrutamiento para las páginas y acceso Web, así como un proceso de creación de formularios basados en archivos XML. Struts 1 también ofrece otras ventajas:

- Struts 1 funciona con todos los servidores Java EE (Tomcat, WebSphere, Weblogic...).
- Struts 1 ofrece una arquitectura sólida y estable (proyecto Apache).
- Struts 1 se adapta a las aplicaciones Web de gran tamaño.
- Struts 1 permite descomponer una aplicación compleja en componentes más simples.
- Struts 1 garantiza un desarrollo similar para los equipos de programadores.
- Struts 1 posee una documentación abundante.
- Struts 1 permite un desarrollo rápido y poco costoso.

El término Struts hace referencia a pilares o columnas, en el sentido arquitectónico del término, con el concepto de ladrillos que sostienen los edificios, las casas y los puentes para evitar que se conviertan en ruinas.

Struts 2

Como acabamos de mencionar en la introducción, en la actualidad se utiliza el modelo de diseño de tipo MVC para el desarrollo de aplicaciones Web evolucionadas. Sin embargo, los principales inconvenientes de este tipo de diseño son:

- la dificultad de comprender el modelo y el nivel de conocimientos que requiere;
- el número de archivos que es necesario producir (cerca de tres veces más);
- el aspecto reiterativo de las tareas que se realizan.

La pregunta entonces es: ¿cómo reducir estos inconvenientes y aumentar la productividad?

Poco antes del año 2000, Craig R. McClanahan's decidió crear una herramienta de diseño de tipo framework para agilizar los desarrollos Java EE. Después, en mayo de 2000, donó a la fundación Apache su framework llamado Struts 1.0 cuyo primer lanzamiento se programó para junio de 2001, que después se convertiría en el framework de diseño Java EE más popular del mundo.

Casi al mismo tiempo, varios desarrolladores trabajaban en la creación de otro framework de desarrollo llamado WebWork. Este framework nunca llegaría a tener la popularidad de Struts, a pesar de ser superior en varios puntos, incluyendo la aplicación de validaciones de formularios y la arquitectura global para la gestión de JavaBeans (sin necesidad de utilizar Beans de formularios). Un punto importante de WebWork en relación con Struts 1.X es el concepto de las pruebas. Con Struts 1.X se requiere un navegador Web para realizar una parte de las pruebas, mientras que WebWork puede funcionar sin utilizar un navegador.

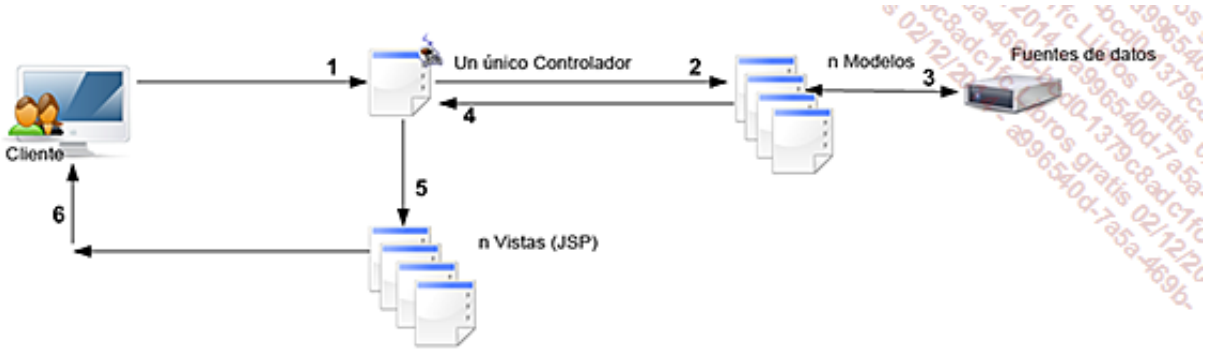
A finales de 2005, el producto WebWork y el framework más popular Struts 1.0 se fusionaron para fundar Struts TI (Titanium) que poco después pasó a ser Struts 2.0. Los diseñadores de Struts 1.0 y Jason Carreira's, responsable de WebWork, proponen entonces un framework que reagrupa las ventajas de las dos herramientas anteriores (WebWork y Struts 1.0). Sin embargo, Struts 2 no es una extensión de Struts 1 y, desafortunadamente, eso puede decepcionar a muchos desarrolladores y arquitectos Web, ya que este nuevo framework tiene un diseño completamente nuevo. Por lo tanto, los especialistas deberán volver a aprender completamente los comandos y las funcionalidades de Struts 2.0. El nuevo diseño correspondería en mayor medida a WebWork 2.2 (WebWork basado en XWork de Open Symphony) que a una evolución de Struts 1.0.

El framework Struts 2 se basa en una declaración de la arquitectura en forma de archivos XML o con anotaciones Java localizadas en los archivos de clases de acciones. Struts 2 es un framework orientado a acciones. Las acciones se descomponen en tres funciones. Primero, las acciones tienen la función más importante del framework, encapsular el procesamiento y el trabajo que deberá realizar el servicio. Segundo, las acciones permiten manipular automáticamente los datos de las consultas durante las transferencias. Tercero, el framework determina qué resultado debe ser devuelto y la vista presentada en respuesta a un procesamiento. Las acciones de Struts 2 implementan objetos JavaBeans (clases Java simples) para cada grupo de datos enviado en la consulta. Cada parámetro de la consulta se declara en la clase de acción con un nombre idéntico para realizar automáticamente la asignación de valores. La finalidad de una acción es devolver una cadena de caracteres, permitiendo seleccionar el resultado que se va a mostrar.

En resumen, Struts 2 se basa en el modelo de diseño de tipo MVC II tal y como se explica en el siguiente esquema. Permite un desarrollo más rápido, más flexible y resuelve varios problemas de diseño ofreciendo los siguientes servicios:

- un sistema evolucionado de gestión del enrutamiento o la navegación;
- un sistema de validación de formularios y de entradas, fácil de aplicar;
- un potente sistema de complementos o de extensiones (para los gráficos, fuentes de datos...);
- gestión de la internacionalización para el desarrollo de sitios multilingües;
- compatibilidad con la tecnología Ajax;

- una herramienta de depuración incluida;
- una potente biblioteca de etiquetas.



Arquitectura MVC II

Struts 2 ofrece (al igual que Struts 1) la posibilidad de utilizar un mínimo de reglas del diseño que no son obligatorias:

- No utilizar código 100% Java en las páginas JSP. Toda la lógica de control reside en las clases de acciones (Servlets).
- Utilizar bibliotecas de etiquetas para acceder a los objetos y navegar por las colecciones.
- Escribir un mínimo código repetitivo y utilizar las herramientas que ofrece el framework.

Este framework también tiene como objetivo ayudar a los desarrolladores de aplicaciones Web en Java a crear proyectos de calidad siguiendo una norma o estándar. Este framework también ayuda a los desarrolladores a organizar la lógica de la aplicación.

La elección del framework Struts 2 se basa en los siguientes puntos:

- Fiabilidad: el proyecto se desarrolló en mayo de 2000 y desde entonces ha tenido un seguimiento. Este proyecto tiene una reputación excelente y mejora constantemente sus defectos.
- Flexibilidad: cada acción puede ser personalizada, los archivos de configuración son muy flexibles en términos de utilización y las validaciones son fáciles de aplicar.
- Rendimiento: la arquitectura diseñada por Struts 2 está basada en WebWork. Tiene un rendimiento especialmente alto y su mantenimiento es sencillo gracias a la separación por capas.

Las principales características del framework Struts 2 son las siguientes:

- los tipos de conversiones automáticas para las colecciones procedentes de consultas HTTP;
- los archivos de configuración modulables que se usan por paquetes;
- las anotaciones Java 5 que reducen las líneas de código para la configuración;
- el uso de etiquetas permite aplicar temas o plantillas (templates);
- el uso del lenguaje de expresión OGNL;
- el uso opcional del complemento interceptor, que permite ejecutar consultas complejas y largas como tareas en segundo plano con el envío múltiple (actualización de páginas);
- la integración simple de herramientas como JSTL, Spring o Hibernate.

El framework Struts 2 es una segunda generación de framework MVC. La principal ventaja del concepto de los interceptores es la flexibilidad del conjunto y la configuración propuesta. Struts 2 responde también al principio de empaquetamiento de la acciones. Cuando declaramos clases de

acción con un archivo XML o anotaciones Java, el framework organiza todos estos componentes en forma de un software de paquete (packages). Los paquetes de Struts 2 son similares a los de Java. Este mecanismo permite igualmente agrupar las acciones por dominio. Las URL de la aplicación se asociarán a los paquetes donde están declaradas las acciones.

Instalación del framework Struts 2

El framework Struts 2 está basado en Java EE 5 (Servlets 2.4 y JSP 2.0 como mínimo). Para utilizar los ejemplos del libro, usamos las últimas versiones de las herramientas con el JDK 1.6, Tomcat 6.X, los Servlets 2.5, Struts 2.1.6 y Eclipse/Lomboz 3.4. Puede visitar el sitio oficial de Struts 2 a través de la siguiente dirección: http://struts.apache.org/2.x/

La instalación del API es sencilla, basta con copiar los archivos descargados en las carpetas de una aplicación Web Java EE tradicional. Podemos crear un nuevo proyecto y copiar los archivos necesarios (bibliotecas *.jar*) o bien instalar el framework utilizando una aplicación Struts virgen proporcionada con el framework (http://apache.cict.fr/struts/examples/struts-2.1.6-apps.zip).

Existen varias versiones de proyectos para instalar Struts 2. En la sección *Download* del sitio, podemos encontrar las últimas versiones de Struts con archivos en formato *.zip*. Las versiones *struts-version-all.zip* incluyen todas las bibliotecas, archivos fuente y ejemplos de aplicación. La versión utilizada en este manual es *struts-2.1.6-all.zip* de 90 MB. La versión *struts-version-lib.zip* contiene únicamente las bibliotecas en formato *.jar* necesarias para el funcionamiento de Struts 2.

El framework Struts V2 está compuesto de varios archivos. Las bibliotecas Java (*.jar*) contienen todas las clases utilizadas por el framework:

- *commons-fileupload.jar* (biblioteca de gestión de la carga en Java).
- *commons-logging.jar* (biblioteca de logging/registros).
- *commons-io-version.jar* (biblioteca de gestión de las entradas/salidas).
- *freemarker-version.jar* (biblioteca utilizada para la presentación y el motor de plantillas).
- *ognl-version.jar* (biblioteca utilizada para la manipulación de objetos Java).
- *junit-version.jar* (biblioteca del framework de gestión de las pruebas unitarias).
- *struts2-core-version.jar* (biblioteca completa de Struts 2, es la biblioteca principal).
- *xwork-version.jar* (biblioteca de XWork con las dependencias).

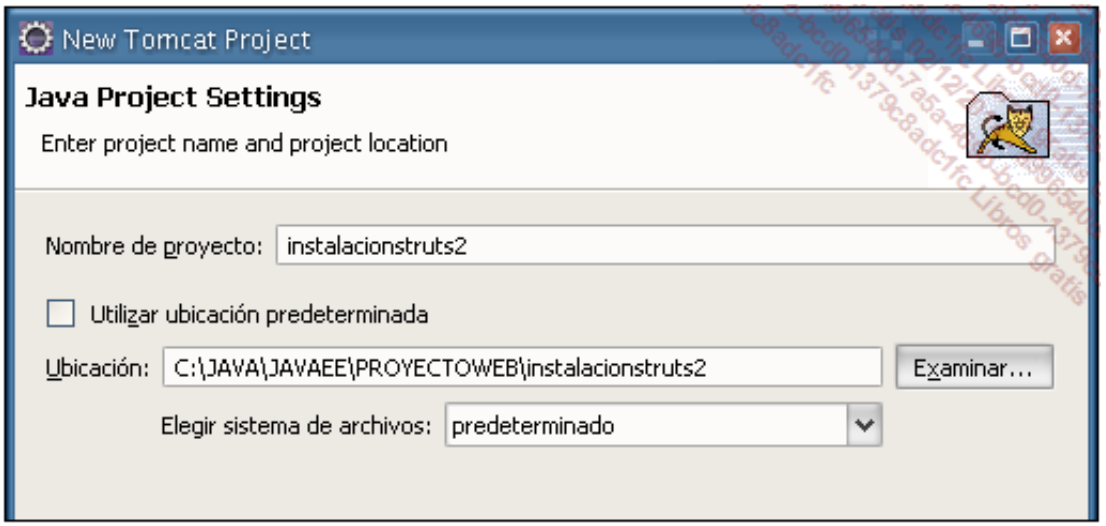
La biblioteca nativa del framework es *struts2-core-version.jar* mientras que las bibliotecas *commons*las proporciona la fundación Apache y el proyecto Commons.

Existen dos maneras de instalar el framework Struts:

- Copiar las bibliotecas *.jar* en el directorio */WEB-INF/lib* de una nueva aplicación.
- Utilizar una aplicación de Struts vacía, incluida en el paquete, que permite instalar el framework. Esta aplicación vacía lleva el nombre de *struts2-blank-version.war* para indicar que está en blanco.

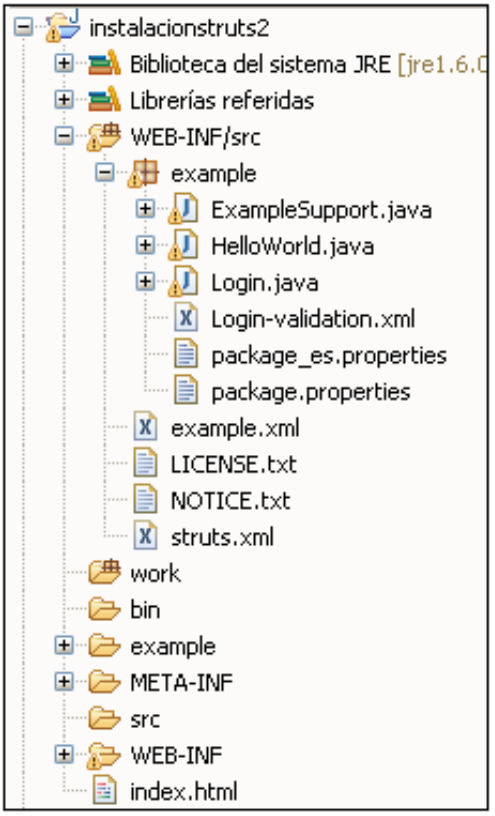
Para instalar esta aplicación, es necesario:

- Copiar el archivo *struts-blank-version.war* en una carpeta (por ejemplo, *instalacionstruts2*).
- Descomprimir su contenido.
- Abrir Eclipse y hacer clic en **Archivo - Nuevo - Proyecto - Java - Proyecto Tomcat** o **Proyecto Web**.
- Dar un nombre a su proyecto (por ejemplo, *instalacionstruts2*) y seleccionar la carpeta anterior.



Proyecto Eclipse

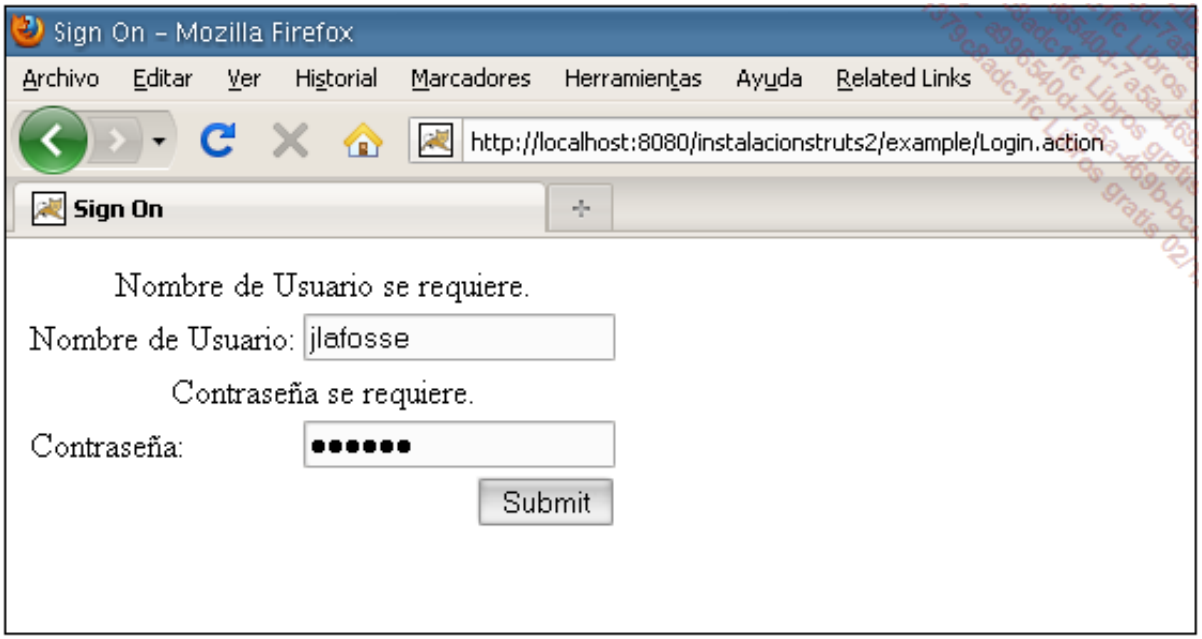
- Editar el proyecto y cambiar el nombre de los paquetes sin el término *java*.
- El árbol del proyecto debe ser el siguiente:



Árbol de instalacionstruts2

- Iniciar Tomcat y abrir un navegador.
- Introducir la siguiente URL en el navegador: http://localhost:8080/instalacionstruts2/

También podemos probar la aplicación de autenticación que viene por defecto:
http://localhost:8080/instalacionstruts2/example/Login.action



Formulario de instalacionstruts2

En resumen

En este capítulo se ha explicado el concepto de framework y las ventajas de la utilización de dicha herramienta para los desarrollos Web.

En la segunda parte se han presentado los distintos framework y se ha aportado la información necesaria para reflexionar acerca de la elección de un framework. En el siguiente punto se recuerdan los servicios que ofrece el framework Struts 1 al mismo tiempo que se presenta la nueva herramienta Struts 2, se explica su instalación y la puesta en marcha del primer ejemplo en blanco.

Presentación

Como ya se ha explicado, es recomendable utilizar el modelo de diseño Modelo Vista Controlador (MVC) para el desarrollo de aplicaciones Web en Java. Antes de comenzar el diseño, es importante comprender el funcionamiento de este modelo de desarrollo. La arquitectura MVC que ofrece Sun es la solución de desarrollo Web del lado del servidor que permite separar la parte lógica de la presentación en una aplicación de Internet. Este es un punto esencial del desarrollo de proyectos ya que permite a todo el equipo trabajar por separado (cada usuario gestiona sus propios archivos, sus programas de desarrollo y sus componentes).

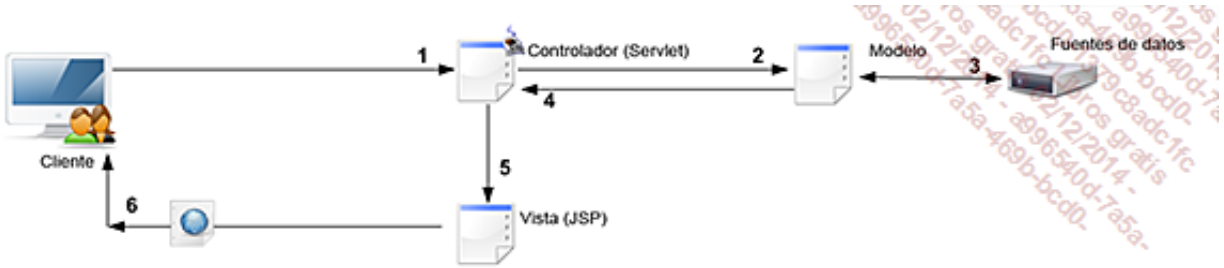
Esta arquitectura tiene su origen en el lenguaje SmallTalk a principios de la década de 1980, por lo tanto no es un nuevo modelo (design pattern) únicamente vinculado a Java EE. El objetivo principal es el de dividir la aplicación en tres partes distintas: **el modelo**, **la vista** y **el controlador**.

En la arquitectura MVC nos encontramos con:

El modelo representado por los EJB y/o JavaBeans y/o sistemas de persistencia (Hibernate, objetos serializados en XML, almacenamiento de datos por medio de JDBC...).

La vista representada por los JSP o clases SWING.

El controlador representado por los Servlets o clases Java.



Arquitectura MVC I

Principio de funcionamiento de la arquitectura MVC

- El cliente envía una consulta HTTP al servidor. En general, esta consulta es un Servlet (o un programa ejecutable del lado del servidor) que procesa la solicitud.
- El Servlet recupera la información transmitida por el cliente y delega el procesamiento a un componente adaptado.
- Los componentes del modelo manipulan o no los datos del sistema de información (lectura, escritura, actualización, eliminación).
- Una vez finalizados los procesamientos, los componentes le devuelven el resultado al Servlet. El Servlet entonces almacena el resultado en el contexto adaptado (sesión, consulta, respuesta...).
- El Servlet llama a la página JSP adecuada que puede acceder al resultado.
- El JSP se ejecuta, utiliza los datos transmitidos por el Servlet y genera la respuesta al cliente.

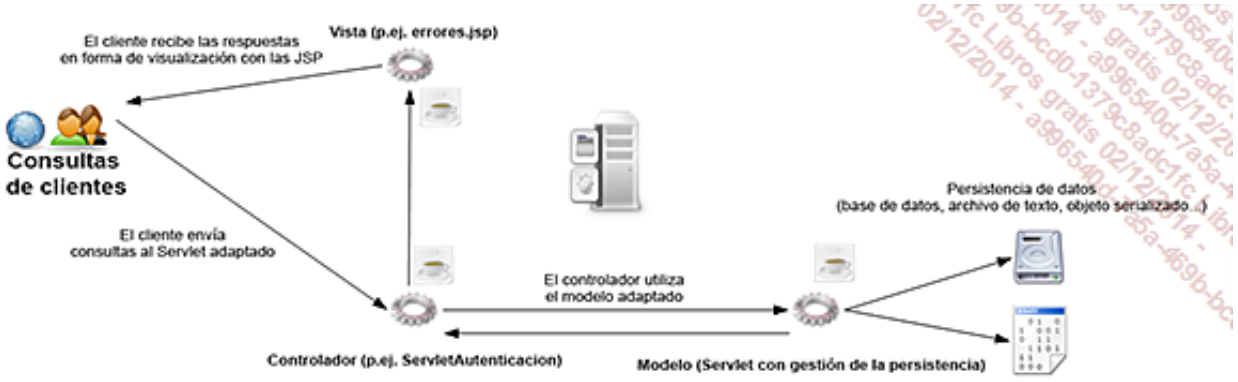
En proyectos simples, las consultas HTTP las administran los componentes Web que reciben las consultas, crean las respuestas y las devuelven a los clientes. En este caso tenemos un único componente responsable de la lógica de visualización, de la lógica empresarial y de la lógica de persistencia. En la arquitectura anterior, la visualización y la manipulación de los datos se mezclan en un único componente de Servlet. Esto puede ser en gran medida adecuado para un servicio específico no evolutivo y simple, pero puede convertirse en un problema cuando el sistema se desarrolla. En esta arquitectura se introduce código Java y código HTML en los Servlets o JSP. Existen varias soluciones a este problema. La más sencilla tiene que ver con la aparición de las páginas JSP y consiste en crear archivos de encabezado, de pie de página, de procesamiento... e incluir todo en una página general.

La arquitectura MVC separa la lógica empresarial de la visualización. En este modelo, un componente se encarga de recibir las consultas (Servlets), otro procesa los datos (Clases) y un tercero administra la visualización (JSP). Si la interfaz entre estos tres componentes está claramente definida, será más fácil modificar un componente sin afectar a los otros dos.

En una aplicación Web evolucionada, la lógica MVC es la siguiente:

El cliente emite consultas al servidor. Cada acción precisa corresponde a un Servlet que redirige las consultas a una página JSP adecuada, realiza un procesamiento o accede a los datos y, en este caso, inicia otro programa que se encargará de responder a la solicitud del usuario actual.

El siguiente esquema presenta una estructura de tipo MCV con la utilización de Servlets y páginas JSP.



Arquitectura MVC, Servlets y JSP

Los Servlets, que tienen una función de controlador en una aplicación MVC, deben disponer de un medio para transmitir las consultas a los componentes encargados de la visualización. Este medio lo proporciona el objeto *RequestDispatcher*. Este componente permite trasladar una consulta de un componente a otro.

Obtenemos un objeto *RequestDispatcher* con el método `getServletContext()`. Desde este objeto, es posible obtener un RequestDispatcher con la ayuda de los métodos siguientes:`getNamedDispatcher(nombre)` o `getRequestDispatcher(ruta)`. El método `getRequestDispatcher(...)`funciona con una ruta que comienza por la barra y que es relativo al contexto de la aplicación. El método `getNamedDispatcher(...)` corresponde a un sub-elemento `<servlet-name>` de un elemento`<servlet-mapping>` del descriptor de implementación *web.xml*.

Los componentes son muy numerosos, pero también muy simples. Sus especificaciones hacen que puedan desarrollarse por parte de especialistas: los Servlets y EJB por desarrolladores Java, los JSP por desarrolladores y Webdesigners, los accesos a los datos por especialistas de SQL... Esta división permite también un mantenimiento más sencillo del sistema. Así, podremos cambiar fácilmente la identidad gráfica utilizando las vistas sin tocar el modelo y el controlador.

En el modelo de diseño MVC, tenemos un Servlet o un filtro que representa el controlador del modelo. Struts 1 utiliza un Servlet, mientras que Struts 2 utiliza un filtro. Para el modelo utilizamos los POJO (*Plain Old Java Object*) que son simples objetos en oposición a los EJB.

El acrónimo POJO se utiliza para hacer referencia a la simplicidad de la utilización de un objeto Java en comparación con la dificultad de la utilización de un componente EJB (*Enterprise Java Bean*). Los JavaBeans (no confundir con los EJB) son componentes lógicos simples reutilizables y manipulables. Para que sea una clase JavaBean, deberá respetar ciertas convenciones de nombre para respetar la utilización, la reutilización, la sustitución y la conexión JavaBean. La clase debe ser serializable (para guardar y leer). La clase debe tener un constructor por defecto (sin argumento). Las propiedades de los métodos deben ser accesibles a través de los métodos (descriptores de acceso). La única diferencia real entre un POJO y un JavaBean es la posibilidad de los JavaBeans de gestionar eventos.

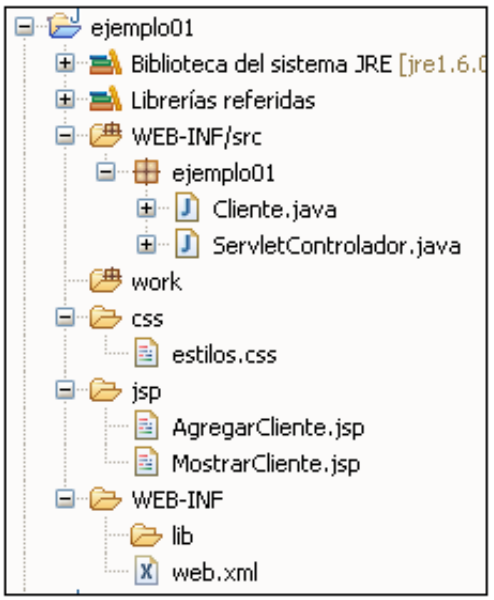
Con el modelo MVC, cada consulta HTTP debe ser enviada al controlador. La sintaxis del URI (*Uniform Resource Identifier*) indica al controlador qué comando debe ser ejecutado. Con Struts 2, una clase de acción puede ejecutar varias operaciones sobre el mismo principio que las *MappingDispatchAction* de Struts 1.

Primer proyecto MVC

Vamos a comenzar con un ejemplo simple de formulario de registro de un nuevo cliente (identificador y contraseña) y la visualización de los datos que ha introducido. La aplicación se llamará *ejemplo01*. El usuario introducirá un identificador y una contraseña, su información se mostrará en otra página después de la creación del usuario, sin persistencia de un objeto *cliente*.

La aplicación se compone de los siguientes elementos:

- Una clase llamada *Cliente*, que es un JavaBean.
- Un Servlet controlador llamado *ServletControlador*.
- Dos páginas JSP para las respectivas visualizaciones del formulario y de los datos introducidos.
- Una hoja de estilos llamada *estilos.css* para dar formato a las visualizaciones.



Árbol del proyecto ejemplo01

El archivo fuente *Cliente.java* es una simple clase POJO.

```
Código: ejemplo01.Cliente.java
package ejemplo01;

import java.io.Serializable;

@SuppressWarnings("serial")
public class Cliente implements Serializable {
    private String identificador;
    private String contrasena;
     public Cliente()
     {

     }
     public String getIdentificador() {
            return identificador;
     }
     public void setIdentificador(String identificador) {
            this.identificador= identificador;
     }

     public String getContrasena() {
```

```
        return contrasena;
    }

    public void setContrasena(String contrasena) {
        this.contrasena= contrasena;
    }
}
```

El Servlet *ServletControlador*, hereda de la clase `javax.servlet.http.HttpServlet` y permite procesar consultas HTTP de tipo Post y Get. El código de este Servlet es muy sencillo, analiza el URI, ejecuta la acción y la redirección adaptada en consecuencia. Por ejemplo, si el URI es*ConfirmarAgregar_cliente.action*, se crea un objeto *cliente* a partir de los datos introducidos en el formulario y se redirige al internauta a la página JSP *MostrarCliente.jsp*.

Código: ejemplo01.ServletControlador.java

```
package ejemplo01;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class ServletControlador extends HttpServlet {

    public void doPost(HttpServletRequest request,
HttpServletResponse response)throws IOException, ServletException
    {
        // el uri tiene el siguiente formato
        /ejemplo01/Agregar_cliente.action
        String uri=request.getRequestURI();
        // se utiliza el código para dividir este URI
        int lastIndex=uri.lastIndexOf("/");
        String action=uri.substring(lastIndex+1);
        System.out.println("ACCIÓN: "+action);

        // para las redirecciones
        String urlRetorno=null;

        // ejecutar la acción adaptada
        if (action.equals("Agregar_cliente.action"))
        {
            urlRetorno="/jsp/AgregarModificarCliente.jsp";
        }
        else if (action.equals("ConfirmarAgregar_cliente.action"))
         {
            // instanciar el objeto
            Cliente cliente=new Cliente();
            // actualización del objeto
            cliente.setIdentificador(request.getParameter("identificador"));
            cliente.setContrasena(request.getParameter("contrasena"));
            // devolver el objeto en la vista
            request.setAttribute("cliente", cliente);
            //página de visualización del cliente
            urlRetorno="/jsp/MostrarCliente.jsp";
        }

       // redirección a la vista adaptada
        if (urlRetorno!=null)
        {
            request.getRequestDispatcher(urlRetorno).forward(request,
response);
         }
```

```
    }

    public void doGet(HttpServletRequest request,
HttpServletResponse response)throws IOException, ServletException
    {
            doPost(request, response);
    }
}
```

Por último, las dos vistas son muy sencillas. Permiten mostrar el formulario de registro de un nuevo cliente (*AgregarCliente.jsp*) así como los datos de este mismo cliente (*MostrarCliente.jsp*).

Código: /jsp/AgregarCliente.jsp

```
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <form method="post" action="ConfirmarAgregar_cliente.action">
      <table>
      <tr>
            <td>Identificador:</td>
            <td><input type="text" name="identificador"/></td>
      </tr>
      <tr>
            <td>Contrase&ntilde;a:</td>
            <td><input type="text" name="contrasena"/></td>
      </tr>
      <tr>
            <td colspan="2" align="center"><input type="submit"
value="Agregar el cliente"/></td>
      </tr>
      </table>
      </form>
</div>
</body>
</html>
```

Código: /jsp/MostrarCliente.jsp

```
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <p>
            <h4>Informaci&oacute;n sobre el cliente:</h4>
            Identificador: ${cliente.identificador}<br/>
            Contrase&ntilde;a: ${cliente.contrasena}<br/>
      </p>
</div>
</body>
</html>
```

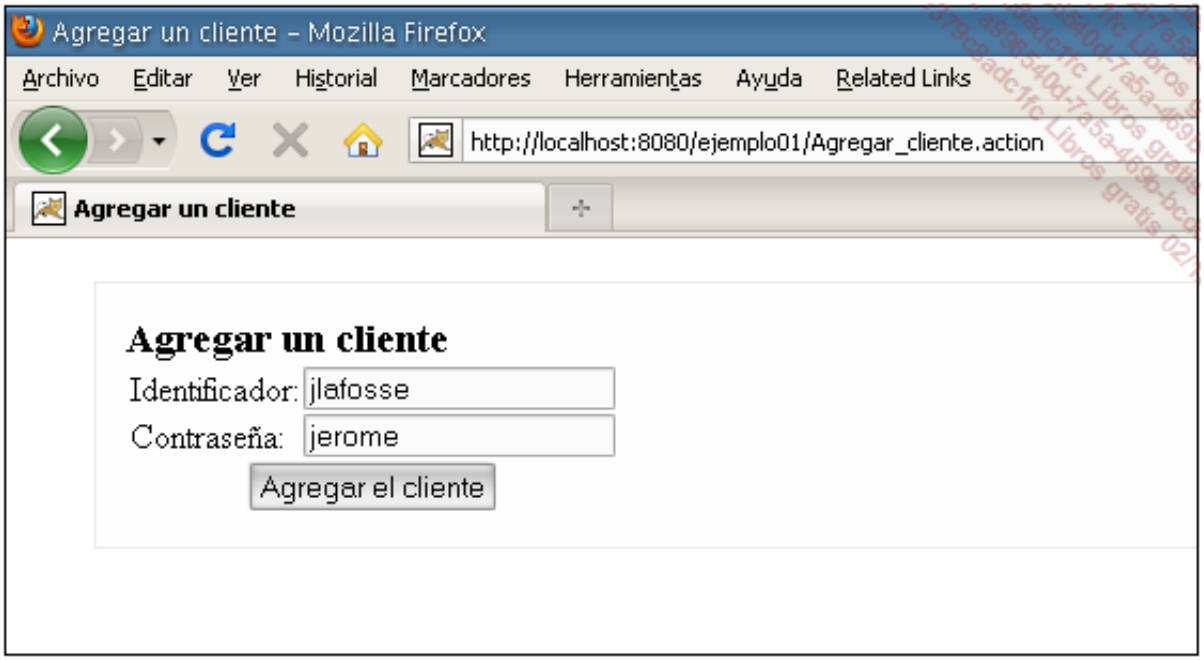
El archivo de configuración de la aplicación, también llamado descriptor de implementación *web.xml*, es muy simple. Posee una sola definición de Servlet que permite administrar todos los URI que terminan por *.action*. Así, podremos administrar sin problemas nuestras

acciones *Agregar_cliente.action* y*Mostrar_cliente.action*.

```
Código: /WEB-INF/web.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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"
    version="2.5">

    <servlet>
        <servlet-name>ServletControlador</servlet-name>
        <servlet-class>ejemplo01.ServletControlador</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>ServletControlador</servlet-name>
        <url-pattern>*.action</url-pattern>
    </servlet-mapping>

</web-app>
```

El modelo *.action permite no escuchar todos los URI (*), lo cual sería inútil para los recursos llamados estáticos como los archivos JavaScript (.js), las imágenes u hojas de estilos CSS (.css). Por lo tanto, nuestro controlador general ignorará estos archivos.

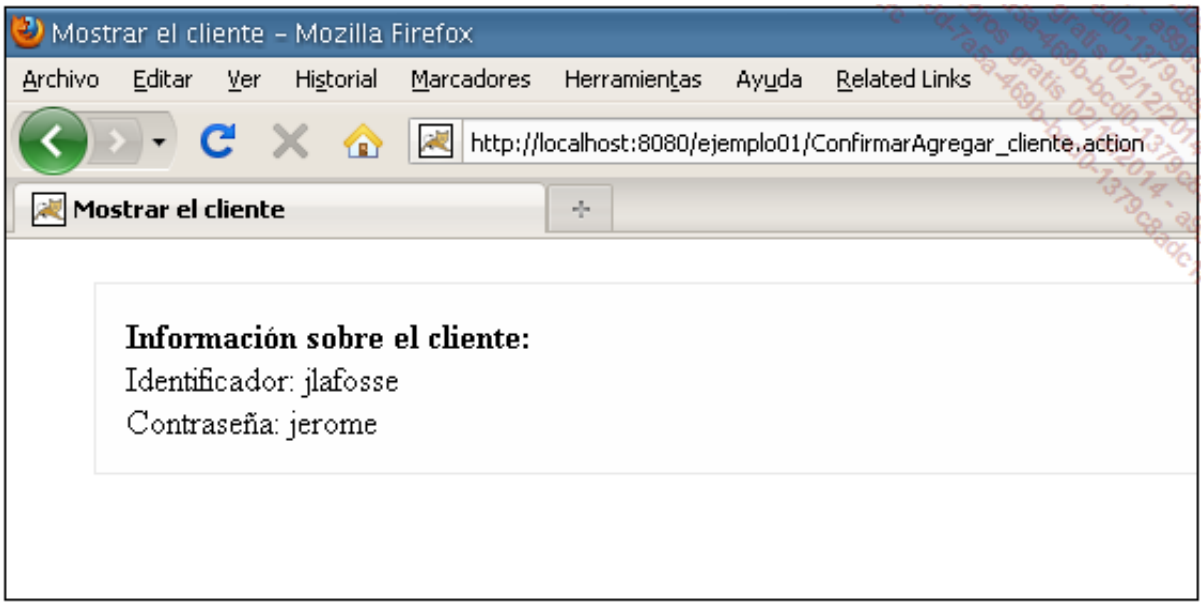
Podemos probar esta aplicación con la siguiente URL: http://localhost:8080/ejemplo01/Agregar_Cliente.action



Formulario de registro de un cliente ejemplo01

Cuando el usuario confirma la creación de un cliente, se ejecutará la siguiente URL:

http://localhost:8080/ejemplo01/ConfirmarAgregar_cliente.action



Visualización de la información del cliente ejemplo01

Proyecto MVC con filtro

La anterior aplicación reposa sobre un Servlet de gestión que permite efectuar las acciones adaptadas en función de la solicitud del internauta. Los filtros permiten dar a una aplicación una estructura modular. Permiten agrupar diferentes tareas que pueden ser imprescindibles para procesar las consultas. La principal función de un Servlet es la de recibir las consultas y responder a los clientes afectados.

Por contra, a menudo es necesario realizar una función idéntica para cada Servlet en relación con las consultas y las respuestas HTTP. Queremos enrutar o transportar un parámetro que se encuentra en todas las consultas sin estar obligado a escribir el código para cada Servlet... La interfaz *Filter* que viene incluida con el API Servlet 2.3 permite resolver este tipo de problema.

Los filtros permiten procesar:

- las consultas procedentes de los clientes antes de que sean procesadas por los Servlets;
- las respuestas procedentes de los Servlets antes de que no sean devueltas a los clientes.

Podemos, por ejemplo:

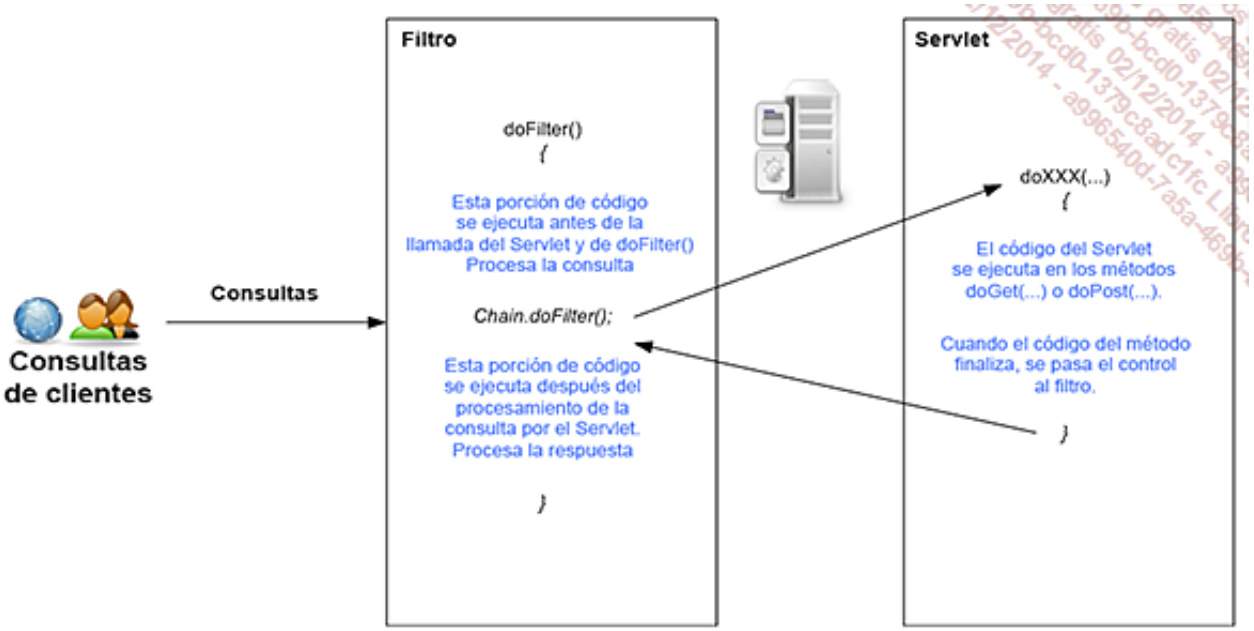
- descifrar las consultas enviadas a los Servlets, procesar los datos con los Servlets y cifrar las respuestas para los clientes;
- gestionar la autenticación de los clientes;
- convertir los formatos de imágenes, aplicar transformaciones XSLT a los datos XML u otros.

Para utilizar un filtro es necesario realizar dos operaciones. La primera consiste en escribir una clase que implemente la interfaz *Filter*. La segunda consiste en modificar el descriptor de implementación (archivo *web.xml*) de la aplicación para indicar al contenedor que utilice el filtro.

Cuando se ha creado un filtro, el contenedor llama a su método `init(...)`. En este método, podemos acceder a los parámetros de inicialización con la interfaz *FilterConfig*. Durante el procesamiento de la consulta, el contenedor llama al método `doFilter(...)`. Por último, antes de destruir el filtro, el contenedor llama su método `destroy(...)`.

Cuando el filtro llama al método `chain.doFilter()`, se ejecuta el siguiente filtro de la cadena. El código situado antes de *chain.doFilter()* se ejecuta antes del procesamiento del Servlet. Cualquier modificación que deba realizarse en el filtro antes de la ejecución de la consulta debe efectuarse antes de esta llamada. El código situado después de esta llamada se ejecuta después del procesamiento del Servlet. También es posible encadenar los filtros y utilizar un filtro para un procesamiento específico.

El siguiente esquema presenta el funcionamiento de un filtro antes y después del procesamiento por el Servlet invocado.



Estructura de un filtro Java EE

El descriptor de implementación *web.xml* permite indicar el o los filtros que se ejecutarán para cada Servlet o URL. El primer elemento `<filter>` permite declarar la clase asociada al filtro. Este elemento debe colocarse al principio del archivo de configuración después de la declaración de variables globales al contexto (`<context-param>`).

En nuestro caso, decidimos un filtro asociado a la clase `FiltreJournalisation` que permite gestionar el acceso a las páginas del sitio.

```
...
<!-- definición del filtro -->
<filter>
    <filter-name>filtroregistro</filter-name>
    <filter-class>cajaherramientas.FiltroRegistro</filter-class>
</filter>
...
```

El segundo elemento necesario es `<filter-mapping>`. Permite, como para los Servlets, administrar el mapping (las relaciones) entre un nombre y un Servlet o una URL. Por ejemplo, aquí el filtro se aplica únicamente al Servlet *servletautenticacion*.

```
...
<filter-mapping>
    <filter-name>filtroregistro</filter-name>
    <servlet-name>servletautenticacion</servlet-name>
</filter-mapping>
...
```

Con la utilización de un filtro, podremos ofrecer a cada recurso (dinámicos y estáticos) de nuestra aplicación un único controlador (un Servlet). Con el anterior ejemplo de descriptor de implementación*web.xml*, sólo los URI con el sufijo *.action* serán procesados por nuestro Servlet. No gestionaremos los recursos estáticos, como por ejemplo las imágenes, archivos JavaScript u hojas de estilos.

```
<servlet>
      <servlet-name>ServletControlador</servlet-name>
      <servlet-class>ejemplo01.ServletControlador</servlet-class>
</servlet>
```

```
<servlet-mapping>
      <servlet-name>ServletControlador</servlet-name>
      <url-pattern>*.action</url-pattern>
</servlet-mapping>
```

El funcionamiento de un filtro es diferente, cada recurso es procesado en la función*filterChain.doFilter()* que sean las consultas o los elementos estáticos. La principal ventaja es la capacidad tanto de gestionar los recursos estáticos como de prohibir el acceso directo.

La nueva aplicación se llamará *ejemplo01-2*. Este proyecto está basado en el anterior, pero el archivo de configuración de la aplicación *web.xml* utiliza en este caso un filtro en su lugar e introduce una definición de Servlet.

La clase `Cliente` y las páginas JSP son idénticas; sin embargo, vamos a desarrollar un filtro al que llamaremos *FiltroControlador* en lugar del archivo *ServletControlador*.

Código: ejemplo01.FiltroControlador.java

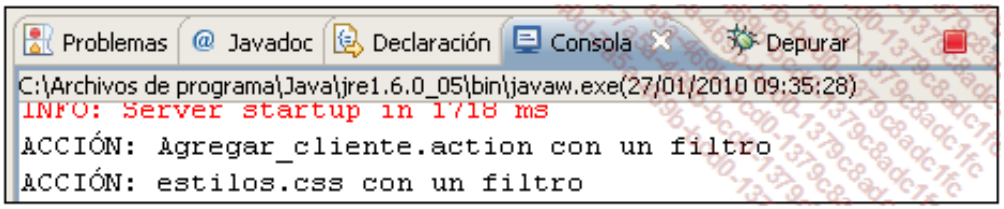
```
package ejemplo01;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@SuppressWarnings("serial")
public class FiltroControlador implements Filter {

    private FilterConfig filterConfig;

    public void init(FilterConfig filterConfig) throws
ServletException
      {
            this.filterConfig=filterConfig;
      }

    public void doFilter(ServletRequest req, ServletResponseresp,
FilterChain filterChain)throws IOException, ServletException
      {
          // conversión
          HttpServletRequest request=(HttpServletRequest)req;
          HttpServletResponse response=(HttpServletResponse)resp;
          // el uri tiene el siguiente formato /ejemplo01-2/Agregar_
cliente.action
          String uri=request.getRequestURI();
          // utilizamos el código para dividir este URI
          int lastIndex=uri.lastIndexOf("/");
          String action=uri.substring(lastIndex + 1);
          System.out.println("ACCIÓN: "+action+" con un filtro");

          // para las redirecciones
          String urlRetorno=null;

          // ejecutar la acción adaptada
          if (action.equals("Agregar_cliente.action"))
          {
              urlRetorno="/jsp/AgregarCliente.jsp";
          }
          else if (action.equals("ConfirmarAgregar_cliente.action"))
          {
```

```
            // instanciar el objeto
            Cliente cliente=new Cliente();
            // actualización del objeto
            cliente.setIdentificador(request.getParameter("identificador"));
            cliente.setContrasena(request.getParameter("contrasena"));
                // devolver el objeto en la vista
                request.setAttribute("cliente", cliente);
                //página de visualización del cliente
                urlRetorno="/jsp/MostrarCliente.jsp";
          }

          // redirección a la vista adaptada
          if (urlRetorno!=null)
          {
            request.getRequestDispatcher(urlRetorno).forward(request,
response);
          }
          // para los recursos estáticos
          else
          {
              filterChain.doFilter(request, response);
          }
    }


    public void destroy()
    {
        this.filterConfig=null;
    }

}
```

El código del controlador es idéntico desde el punto de vista algorítmico. El internauta puede visualizar el formulario de creación y después de la confirmación, visualizar sus datos a través de un objeto *cliente*. El código del descriptor de implementación se modifica para declarar un filtro a la escucha de las consultas.

```
Código: /WEB-INF/web.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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"
    version="2.5">

    <filter>
        <filter-name>FiltroControlador</filter-name>
        <filter-class>ejemplo01.FiltroControlador</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>FiltroControlador</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Podemos utilizar esta aplicación con la siguiente URL para agregar un cliente: http://localhost:8080/ejemplo01-2/Agregar_cliente.action



Consola y registros del filtro

En resumen

En este capítulo se ha presentado el mecanismo de desarrollo Web en Java con el modelo de diseño MVC Modelo Vista Controlador. También hemos observado dos técnicas de programación. La primera utiliza el principio de mapping de los URI para interceptar las consultas HTTP con la ayuda de un Servlet controlador. La segunda introduce el mecanismo de filtro destacado en Struts 2. En el siguiente capítulo, escribiremos nuestra primera aplicación con el framework Struts 2.

Presentación

Este capítulo presenta el framework Struts 2 a través de un ejemplo concreto para acelerar y facilitar el desarrollo de aplicaciones Web.

A partir de este punto del libro, se utilizará el término Struts para hacer referencia a Struts 2.

Las aplicaciones de Struts tienen un archivo de configuración llamado *struts.xml*. Este archivo de configuración es el más importante y sustituye a la definición de los Servlets en el descriptor de implementación *web.xml*. Este archivo permite administrar la configuración de las acciones que se van a realizar.

Struts también tiene un archivo de propiedades presente por defecto en el archivo de la aplicación*struts2-core-version.jar* llamado *default.properties*. Este archivo utilizado por defecto contiene los textos de validación (mensajes de error y de confirmación) y los parámetros de configuración de Struts. Este archivo predeterminado es suficiente para comenzar con una aplicación simple (en inglés).

Como ya hemos dicho en el capítulo anterior, Struts utiliza un filtro para realizar el enrutamiento hacia un solo controlador de administración correspondiente al modelo MVC II. El controlador de Struts es capaz de:

- Determinar el URI para la acción que se va a ejecutar.
- Utilizar una clase de acción.
- Ejecutar el método de acción de la clase si está asociada.
- Si se han introducido datos, crear un objeto y actualizarlo o posicionar los valores de los parámetros.
- Regresar a la vista (página JSP) para mostrar la respuesta.

Gracias a la utilización de un filtro y de un controlador global, no hemos escrito un controlador complejo para cada servicio (por ejemplo, administración de clientes, de artículos...) para administrar el enrutamiento de la aplicación. Lo más importante para el desarrollador es administrar las acciones asociadas a una demanda específica.

Funcionamiento general de Struts 2

Struts utiliza un filtro como en el caso de la aplicación *ejemplo01-2*. Este filtro debe estar declarado en el descriptor de implementación de nuestra aplicación *web.xml*. Este filtro se define en la clase`org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter`.

```
Código: /WEB-INF/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">

    <display-name>Struts Blank</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>
</web-app>
```

Con Struts, el método de acción de la clase se ejecuta después de que todas las propiedades hayan sido procesadas y afectadas. Por su parte, un método de acción devuelve una cadena de caracteres de tipo *String*. Esta cadena indica a Struts donde debe redirigirse al controlador. Por ejemplo, la cadena *success* indica a Struts que regrese a la página en caso de que el procesamiento haya sido correcto y la cadena *error* determina la página que se mostrará en caso de error.

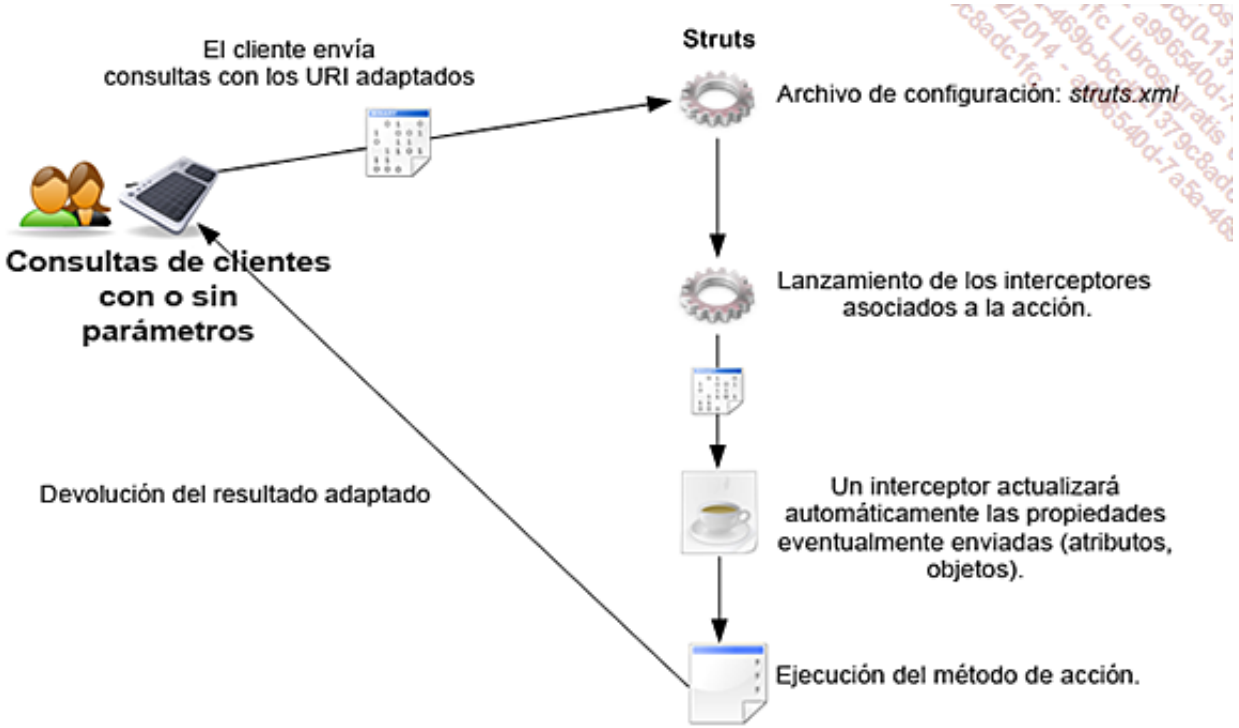
En la mayoría de los casos, Struts redirige al usuario a una vista JSP con la ayuda de la interfaz*RequestDispatcher* de Java. En general, las vistas devueltas están en formato JSP, pero también pueden ser modelos de Velocity o FreeMarker. Además, Struts también puede devolver un flujo multimedia de tipo imagen, por ejemplo.

Para probar y analizar los URI, Struts utiliza el archivo de configuración llamado *struts.xml*. Este archivo de configuración debe colocarse en el directorio */WEB-INF/src* (o */WEB-INF/classes* después de la compilación) para que funcione por defecto. Todas las acciones se declararán en el archivo de enrutamiento y a cada declaración de acción le corresponde un nombre de URI. Este archivo de configuración *struts.xml* se lee al iniciar la aplicación. Podemos, para simplificar y para evitar volver a cargar el administrador, declarar la aplicación en modo de desarrollo. Con esta modalidad de diseño, el archivo se volverá a cargar cada vez que haya un cambio en la aplicación. Con esta etiqueta, no es necesario recargar el contenedor.

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

Cada declaración de acción esta asociada a una clase completamente cualificada (`nombrepaquete.nombreclase`). En caso contrario, podemos definir una acción de Struts por defecto. Una clase de acción deberá tener al menos un tipo de cadena de caracteres, pero también puede tener varios (confirmación, error, consulta, lista...).

A la hora de acceder a un recurso o de ejecutar una acción, Struts utiliza este proceso de llamada:



Proceso de ejecución de Struts

1) El cliente envía consultas desde URL adaptadas. También puede enviar parámetros dentro de la URL o mediante un formulario.

2) Struts consulta su archivo de configuración *struts.xml* para encontrar la configuración de la acción.

3) Se ejecuta cada interceptor asociado a la acción. Uno de estos interceptores es el encargado de asignar automáticamente los valores recibidos en la consulta con las propiedades de la clase de acción, en función de los nombres (por ejemplo, *identificador*, *contrasena*).

4) Struts ejecuta el método en de acción asociada a la clase.

5) Se devuelve el resultado adaptado al usuario solicitante.

Los interceptores son se ponen en marcha después de que se ejecute el método de la acción, lo que permite a estos mismos interceptores ejecutar las operaciones sobre los parámetros después del procesamiento del método de acción.

Los interceptores de Struts 2

Un interceptor de Struts es un filtro que puede efectuar distintos procesamientos sobre una acción. El código presente en un interceptor es modulable y puede añadirse o suprimirse directamente en el archivo de configuración *struts.xml*. También es posible añadir código especifico a una aplicación sin recompilar el framework principal.

En este punto del libro, es necesario comprender que un interceptor es un filtro que tiene una función específica y permite, por ejemplo, administrar las cookies, los parámetros HTTP, la depuración, la carga de archivos (upload) o incluso los alias de las acciones.

El archivo de configuración struts.xml

Una aplicación de Struts tiene un archivo de configuración *struts.xml* el formato XML y el archivo de propiedades predeterminado *default.properties*, pero también puede tener otros archivos de configuración.

Es posible que no haya ningún archivo de configuración en una aplicación de Struts. A este tipo de programación se le conoce como configuración cero (zero configuration). Esta técnica utiliza las anotaciones Java en las clases para no declarar las acciones en un archivo de configuración.

En el archivo *struts.xml* vamos a definir la configuración general de la aplicación:

- Los parámetros de configuración de Struts.
- Las acciones.
- Los interceptores.
- Los resultados de las acciones.

El archivo de configuración *struts.xml* siguiente se incluye con la aplicación *struts-blank.war*.

```
<?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" />

    <include file="example.xml"/>

    <package name="default" namespace="/" extends="struts-default">
        <default-action-ref name="index" />
        <action name="index">
            <result type="redirectAction">
                <param name="actionName">HelloWorld</param>
                <param name="namespace">/example</param>
            </result>
        </action>
    </package>
</struts>
```

1. La etiqueta <package/>

Las acciones de Struts se reagrupan por paquete según un principio semejante al de los paquetes Java. En el ejemplo anterior, el paquete se llama *default*.

Una etiqueta `<package/>` debe tener un atributo `name` para hacer referencia al paquete. El atributo opcional `namespace` posee el valor por defecto /. Este espacio de nombre siempre se añade a los URI que ejecutan las acciones del paquete.

Los atributos de la etiqueta `<package/>` son los siguientes:

- `name`: atributo obligatorio que determina el nombre del paquete.
- `namespace`: atributo que determina el espacio de nombre para todas las acciones del paquete.
- `extends`: atributo obligatorio que determina el paquete padre que se va a heredar.
- `abstract`: atributo poco utilizado que determina si el paquete puede ser heredado o no.

Para invocar un URI se utiliza el siguiente modelo: /contextoaplicacion/nombreaccion.action (o nombreaccion.do).

Ahora, para invocar una acción en un espacio de nombre, debemos utilizar la siguiente sintaxis:

```
 /contextoaplicacion/espaciodenombre/nombreaccion.action (o nombreaccion.do)
```

Por ejemplo, para ejecutar la parte administrativa del proyecto *ejemplo01* definida como sigue, utilizamos el siguiente URI: http://localhost:8080/ejemplo01/admin

```
<?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="ejemplo01" namespace="/admin" extends="struts-
default">
```

La etiqueta `<package/>` debe, en la mayoría de los casos, extender el paquete *struts-default* definido en el archivo *struts-default.xml*. En este caso, todas las acciones pueden utilizar los interceptores declarados en el archivo *struts-default.xml*. Los desarrolladores pueden crear

```
        ...
    </package>
...
</struts>
```

sus propios paquetes heredando de este paquete predeterminado que pone a disposición la mayoría de los interceptores útiles para la programación de aplicaciones de Internet.

2. La etiqueta <include/>

Como ya hemos mencionado, el archivo de configuración de Struts se llama *struts.xml*. En una aplicación compleja, podemos tener varios paquetes y una gran cantidad de líneas en este archivo de configuración. Para mejorar la administración y el mantenimiento de este archivo, es posible dividir este archivo en varios sub-archivos. Cada sub-archivo se incluye en este archivo principal. La etiqueta `<include/>` permite dividir el archivo principal en varios sub-archivos. Idealmente, cada sub-archivo define su propio paquete.

Mayúscula inicial por ejemplo, podemos dividir el archivo de configuración de la aplicación por las partes frontoffice y backoffice del sitio. Cada archivo incluido debe utilizar la misma gramática (*DOCTYPE*) y la etiqueta raíz del documento `<struts/>`.

```
Código: 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" />
    <include file="frontoffice.xml"/>
    <include file="backoffice.xml"/>
</struts>
```

```
Código: frontoffice.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="default" namespace="/" extends="struts-default">
        <default-action-ref name="index" />
        <action name="index">
            <result type="redirectAction">
                <param name="actionName">HelloWorld</param>
                <param name="namespace">/example</param>
            </result>
        </action>
    </package>
</struts>
```

3. La etiqueta <action/>

La etiqueta `<action/>` se utiliza con una etiqueta `<package/>` y representa a la acción proporcionada por el paquete. Una acción debe tener un nombre. El nombre es elección del desarrollador, pero debe ser lo más explícito posible para mejorar el mantenimiento. Una acción está, en general, asociada a una clase, pero esto no es obligatorio para las redirecciones. Una acción que no precisa de clase utiliza una instancia de la clase por defecto, *ActionSupport*.

La sintaxis es la siguiente:

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

Por contra, si se ha definido una clase, el nombre de la clase debe estar completamente cualificado (`nombrepaquete.nombreclase`).

```
 <action name="unaaccion" class="nombrepaquete.nombreclase"/>
```

También podemos especificar el nombre del método de la clase que debe ejecutarse al inicio de la acción.

La sintaxis es la siguiente:

```
 <action name="unaaccion" class="nombrepaquete.nombreclase"
 method="nombremetodo"/>
```

Si el atributo `class` está presente en la definición, pero solamente se ha precisado el atributo`method`, por defecto se utilizará el método `execute()` de la clase al inicio de la acción.

Por lo tanto, las siguientes declaraciones son idénticas:

```
<action name="Agregar_Cliente" class="ejemplo02.acciones.Cliente"
method="execute"/>
<action name="Agregar_Cliente" class="ejemplo02.acciones.Cliente"/>
```

4. La etiqueta <result/>

La etiqueta `<result/>` se utiliza en el interior de una etiqueta `<action/>` como sub-elemento. Esta etiqueta permite indicar a Struts donde devolver el resultado de la acción. Una acción puede devolver varios resultados (todos diferentes), por lo tanto puede tener varias etiquetas `<result/>`, cada una correspondiente a un resultado posible. Por ejemplo, una acción puede devolver una página de error en caso de existir un problema de sintaxis en los datos introducidos (*input*) o una página de confirmación para mostrar el resultado (*con success*).

El siguiente es un ejemplo de configuración para nuestro proyecto de creación de una cuenta de cliente a partir del identificador y de la contraseña:

```
<action name="ConfirmarAgregar_Cliente" class="ejemplo04.Cliente"
method="agregar">
    <result name="input">/jsp/AgregarCliente.jsp</result>
    <result name="success">/jsp/Mostrar.jsp</result>
</action>
```

Se ejecutará el primer resultado si el método `agregar()` devuelve la cadena de caracteres *success*. En este caso, se mostrará la página *Mostrar.jsp* en el navegador. Se ejecutará el segundo resultado si el método `agregar()` devuelve la cadena de caracteres *input* cuando se produce un error en la introducción de datos. En este caso, se muestra la página de introducción de datos de nuevo sin perder los datos.

El atributo `type` de la etiqueta `<result/>`, determina el tipo de resultado que se va a devolver. Este tipo debe especificarse en el paquete o un paquete heredado. En el caso de una redirección, por ejemplo, se define el tipo *dispatcher* en el paquete *struts-default*. Si no utilizamos el atributo `type` en la etiqueta `<result/>`, el valor *dispatcher* (redirección) se utiliza por defecto. Si no utilizamos el atributo `name` en la etiqueta `<result/>`, el valor `success` se utiliza por defecto.

Por lo tanto, las siguientes declaraciones son idénticas:

```
<result name="success" type="dispatcher">/jsp/mostrarcliente.jsp
</result>
<result>/jsp/mostrarcliente.jsp</result>
```

Si un método de una clase de acción devuelve un resultado que no está presente en una etiqueta `<result/>` de la acción, Struts buscará entonces un resultado correspondiente en el grupo `<global-results/>`. Esta etiqueta permite realizar agrupaciones de etiquetas para evitar las definiciones repetitivas. En esta etiqueta, por ejemplo, podemos encontrar las redirecciones hacia la página de inicio del sitio, la página de error, de autenticación o de administración.

5. La etiqueta <param/>

La etiqueta `<param/>` puede utilizarse con los elementos `<action/>`, `<result-type/>` e`<interceptors/>` para asignar un valor al objeto afectado. La etiqueta `<param/>` posee un atributo`name` para nombrar el valor.

Si se utiliza con una acción, la etiqueta `<param/>` permite asignar un valor a una propiedad.

Por ejemplo, el identificador por efecto para el cliente se determina con la etiqueta `<param/>`.

```
<action name="agregar_Cliente" class"...">
    <param name="identificador">jlafosse</param>
</action>
```

6. La etiqueta <constant/>

El archivo de configuración *struts.xml* puede estar asociado opcionalmente con el archivo de propiedades *struts.properties*. Podemos crear parejas de claves/valores para modificar los valores predeterminados definidos en el archivo *default.properties* incluido con el paquete *struts2-core-version.jar*.

Para modificar o sobrecargar un valor del archivo *default.properties* sin utilizar el archivo*struts.properties*, podemos utilizar la etiqueta `<constant/>` en el archivo *struts.xml*.

La etiqueta `<constant>` posee los atributos `name` y `value`. Por ejemplo, para utilizar Struts en modo de desarrollo para la administración de los registros y depuraciones, podemos utilizar la siguiente configuración:

```
Código: 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.devMode" value="false" />
    ...
</struts>
```

7. La etiqueta <global-results/>

La etiqueta `<global-results/>` se utiliza para definir los resultados generales. Si una acción no encuentra definición de resultado en su propia definición, Struts entonces busca un resultado en la definición de esta etiqueta. La declaración de los resultados en esta etiqueta es idéntica a la declaración presente en las acciones.

Si Struts no encuentra ninguna correspondencia de resultado, de manera local o en la etiqueta `<global-results/>`, se produce una excepción.

```
<global-results>
    <result
name="inicio">/vistas/usuarios/index.jsp</result>
    <result name="admin"
type="redirectAction">admin/Admin.action</result>
</global-results>
```

8. La etiqueta <interceptors/>

La etiqueta `<interceptors/>` se utiliza para definir a los interceptores (una especie de filtro). Como ya se ha explicado, una acción puede contener una lista de interceptores que operarán sobre las acciones. Antes del poder utilizar un interceptor, debemos declararlo en la etiqueta `<interceptors/>`.

Por ejemplo, el siguiente código registra dos interceptores: *confirmación* (para las confirmaciones del usuario) y *alias* (para la administración de los parámetros).

```
<package name="nombredelpaquete" extends="struts-default">
 <interceptors>
    <interceptor name="confirmacion" class="..."/>
    <interceptor name="alias" class="..."/>
 </interceptors>
</package>
```

Ahora que ya se han registrado los interceptores, podemos utilizarlos con la etiqueta `<interceptor-ref/>` que es un sub-elemento de la etiqueta `<action/>`.

A continuación se muestra un ejemplo de la utilización de los interceptores anteriores:

```
<package name="nombredelpaquete" extends="struts-default">
 <interceptors>
    <interceptor name="confirmacion" class="..."/>
    <interceptor name="alias" class="..."/>
 </interceptors>
 <action name="nombreaccion" class="nombrepaquete.nombreclase">
    <interceptor-ref name="confirmacion"/>
    <interceptor-ref name="alias"/>
    <result>/jsp/mostrarcliente.jsp</result>
 </action>
</package>
```

El orden de declaración de los interceptores tiene una gran importancia. De hecho, determina el orden de ejecución de los interceptores para cada acción. En el ejemplo anterior, el interceptor confirmación se ejecutará en primer lugar seguido de interceptor alias.

Para evitar la repetición de declaraciones de interceptores, Struts ofrece la posibilidad de crear grupos de interceptores. El desarrollador ya no tendrá necesidad de realizar múltiples referencias para cada etiqueta `<action/>` que utiliza estos interceptores. A continuación se muestra un ejemplo de definición de un grupo de interceptores:

```
<interceptor-stack name="interceptorGlobal">
    <interceptor-ref name="confirmacion"/>
    <interceptor-ref name="alias"/>
</interceptor-stack>
```

Para hacer referencia a un grupo de interceptores, basta con agregar la etiqueta `<interceptor-ref/>` y el atributo `name` dentro de una acción.

```
<action name="nombreaccion" class="nombrepaquete.nombreclase">
    <interceptor-ref name="interceptorGlobal"/>
    <result>/jsp/mostrarcliente.jsp</result>
</action>
```

El paquete *struts-default* definir varios grupos de interceptores. La etiqueta `<default-interceptor-ref/>` especifica el grupo de interceptores por defecto.

```
 <default-interceptor-ref name="defaultStack"/>
```

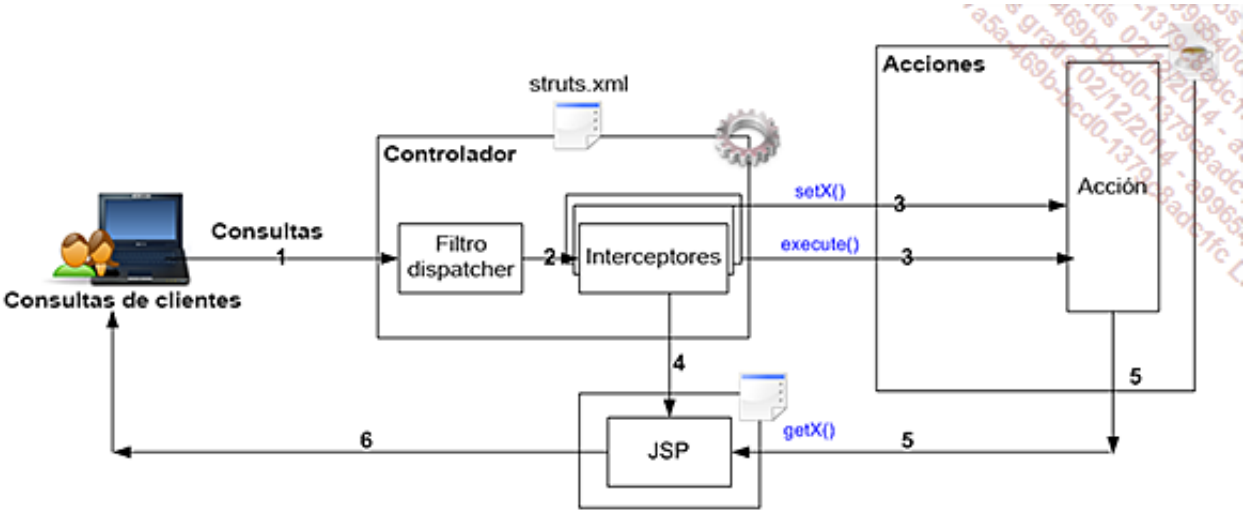
Si una acción necesita utilizar otros interceptores distintos a los definidos por defecto, podemos redefinir el grupo de interceptores por defecto. El interceptor `params` es el más importante, ya que permite administrar la copia y el acceso automático a los datos de los parámetros de la consulta, en nuestras acciones y para los JavaBeans utilizados. A continuación, se presenta una lista de los interceptores de Struts utilizados por defecto:

```
<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>
```

Arquitectura de Struts 2

El esquema arquitectónico siguiente presenta el funcionamiento del framework basado en la utilización de interceptores y de propiedades de JavaBeans.



Arquitectura de Struts 2

1) El cliente envía consultas hacia un servicio de la aplicación con los parámetros eventuales.

2) Se consulta el archivo de configuración de la aplicación o las anotaciones de clases.

3) Los interceptores asociados a la acción se ejecutan y realizan los servicios asociados (conservar los parámetros, administrar las sesiones, guardar los mensajes de error...). El interceptor `params` asignan los valores presentes en la consulta a la clase de acción asociada por medio de sus descriptores de acceso y ejecutar el método de procesamiento (`execute()` predeterminado).

4) La vista que se mostrará se selecciona de acuerdo con el archivo de configuración *struts.xml* o la anotación correspondiente.

5) La clase de acción transmite los datos necesarios a la vista.

6) La vista muestra al cliente los resultados procesados.

Los archivos de propiedades struts.properties y default.properties

Para administrar la configuración de Struts, podemos utilizar las etiquetas `<constant/>` presentes en el archivo de configuración *struts.xml* como se ha indicado anteriormente, o bien utilizar un archivo*struts.properties*.

Para ello, debemos crear un archivo *struts.properties* en la carpeta */WEB-INF/src* (o */WEB-INF/classes*) de la aplicación y sobrecargar los valores para las parejas claves/valores (principio de los archivos de propiedades a Java). Estos valores por defecto se presentan en el archivo *default.properties* presente en el paquete *struts2-core-version.jar*.

Para volver a tomar el ejemplo, podemos utilizar Struts en modo de depuración en el archivo*struts.properties*:

```
struts.devMode=true
```

Los valores predeterminados de las propiedades del archivo *default.properties* se detallan a continuación:

struts.i18n.encoding=UTF-8: codificación predeterminada utilizada por Struts.

struts.objectFactory=spring: el objeto Factory utilizado por defecto por Struts.

struts.objectFactory.spring.autoWire=name: el valor utilizado por defecto cuando utilizamos el objeto SpringObjectFactory. Los valores posibles son: *name*, *type*, *auto* y *constructor*.

struts.objectFactory.spring.useClassCache=true: este parámetro permite indicar si las instancias de la integración de Struts-Spring deben almacenarse en el caché.

struts.objectFactory.spring.autoWire.alwaysRespect=false: este parámetro permite administrar la estrategia del autowire.

struts.objectTypeDeterminer=notiger: este parámetro permite administrar la implementación de la clase`com.opensymphony.xwork2.util.ObjectTypeDeterminer`.

struts.multipart.parser=jakarta: este parámetro permite especificar el analizador utilizado por las consultas multi-parte (upload).

struts.multipart.saveDir: este parámetro permite especificar la carpeta por defecto para la carga de archivos.

struts.multipart.maxSize=2097152: este parámetro permite especificar el tamaño máximo para los archivos que se cargan.

struts.custom.properties=application,org/apache/struts2/extension/custom: este parámetro permite especificar la lista de archivos de propiedades que deben cargarse.

struts.mapper.class=org.apache.struts2.dispatcher.mapper.DefaultActionMapper: este parámetro permite precisar cómo se asignan las URL a las acciones. El valor por defecto de este parámetro es:

org.apache.struts2.dispatcher.mapper.DefaultActionMapper.

struts.action.extension=action: este parámetro permite administrar por una lista separada por comas, la lista de extensiones para las acciones (por ejemplo, *.action*, *.do*).

struts.serve.static=true: este parámetro indica si Struts debe administrar los contenidos estáticos presentes en los archivos *.jar*.

struts.serve.static.browserCache=true: este parámetro indica si el filtro utilizado por Struts debe o no almacenar en el caché del navegador del cliente, los contenidos estáticos (imágenes, css, javascript...).

struts.enable.DynamicMethodInvocation=true: este parámetro indica si la invocación de métodos dinámicos es posible con Struts. Las invocaciones dinámicas consisten en crear URI de la siguiente forma: `agregar!Cliente.action`.

El valor por defecto es *true* pero por motivos de seguridad, podemos pasarlo a *false* si no utilizamos ninguna invocación dinámica.

struts.enable.SlashesInActionNames=false: este parámetro indica si se autoriza o no el uso de barras en los nombres de las acciones.

struts.tag.altSyntax=true: este parámetro indica si se autoriza o no el uso de expresiones regulares %{}.

struts.devMode=false: este parámetro indica si nos encontramos en modo de desarrollo o no. Cuando este parámetro tiene el valor *true*, el archivo de configuración *struts.xml* se recarga así como los archivos de confirmación y los mensajes de propiedades (bundles). Esto significa que no necesitamos volver a cargar la aplicación cada vez que se realice una modificación de un archivo estático. Asimismo, este parámetro permite una visualización en modo detallado de la depuración. Sin embargo, el modo de producción este parámetro debe establecerse como *false* para evitar mostrar las excepciones y reducir la memoria.

struts.i18n.reload=false: este parámetro permite precisar si los archivos de lenguas (bundles) deben ser recargados para cada consulta. Este principio es muy útil en la fase de desarrollo, pero no debe ser utilizado en un servidor de producción.

struts.ui.theme=xhtml: este parámetro permite precisar el tema utilizado por defecto con la biblioteca de etiquetas de Struts.

struts.ui.templateDir=template: este parámetro permite especificar la ruta de las plantillas (templates).

struts.ui.templateSuffix=ftl: este parámetro permite especificar el tipo de plantilla determinado. El valor predeterminado corresponde a la plantilla propuesta por FreeMarker (ftl), aunque también están disponibles Velocity (vm) y JSP (jsp).

struts.configuration.xml.reload=false: este parámetro indica si los archivos de configuración en formato XML deben ser recargados (*struts.xml* y los archivos eventuales incluidos).

struts.velocity.configfile=velocity.properties: este parámetro permite especificar el archivo de configuración predeterminado para el motor de plantillas Velocity.

struts.velocity.contexts=: este parámetro permite declarar una lista de clases que puede utilizarse por el contexto Velocity.

struts.velocity.toolboxlocation=: este parámetro permite declarar la ruta de la caja de herramientas Velocity.

struts.url.http.port=80: este parámetro permite especificar el puerto utilizado para las consultas y URL HTTP.

struts.url.https.port=443: este parámetro permite especificar el puerto utilizado por las consultas y URL HTTPS.

struts.url.includeParams=none: este parámetro permite especificar si los parámetros se incluyen en la URL.

struts.custom.i18n.resources=testmessages,testmessages2: este parámetro permite cargar los archivos de mensajes (bundles) por defecto.

struts.dispatcher.parametersWorkaround=false: este parámetro indica si la función*HttpServletRequest.getParameterMap()* debe activarse o no.

struts.freemarker.manager.classname=org.apache.struts2.views.freemarker.FreemarkerManager: este parámetro permite especificar la clase que utilizará FreeMarker.

struts.freemarker.templatesCache=false: este parámetro permite especificar si el motor de FreeMarker debe utilizar el caché o no.

struts.freemarker.beanwrapperCache=false: este parámetro permite especificar si el cache de los beans debe ser utilizado uno con FreeMarker.

struts.freemarker.wrapper.altMap=true: este parámetro permite precisar si el mapping está autorizado con FreeMarker.

struts.freemarker.mru.max.strong.size=100: este parámetro permite configurar el tamaño máximo para el caché de FreeMarker.

struts.xslt.nocache=false: este parámetro permite especificar si los resultados XSLT deben utilizar el caché.

struts.mapper.alwaysSelectFullNamespace=false: este parámetro indica si Struts debe utilizar o no el nombre de espacio antes de la última barra.

El archivo de propiedades de la aplicación web.xml

Para la administración de la configuración de Struts, podemos utilizar las etiquetas `<constant/>` y el archivo *struts.properties*. Existe también una última manera de configurar Struts, por medio del archivo de descripción de la aplicación *web.xml*.

Para ello, debemos utilizar las constantes de inicialización del filtro con la siguiente sintaxis:

```
Código: /WEB-INF/web.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
</filter-class>
        <init-param>
            <param-name>struts.devMode</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
...
</web-app>
```

El archivo de configuración struts-default.xml

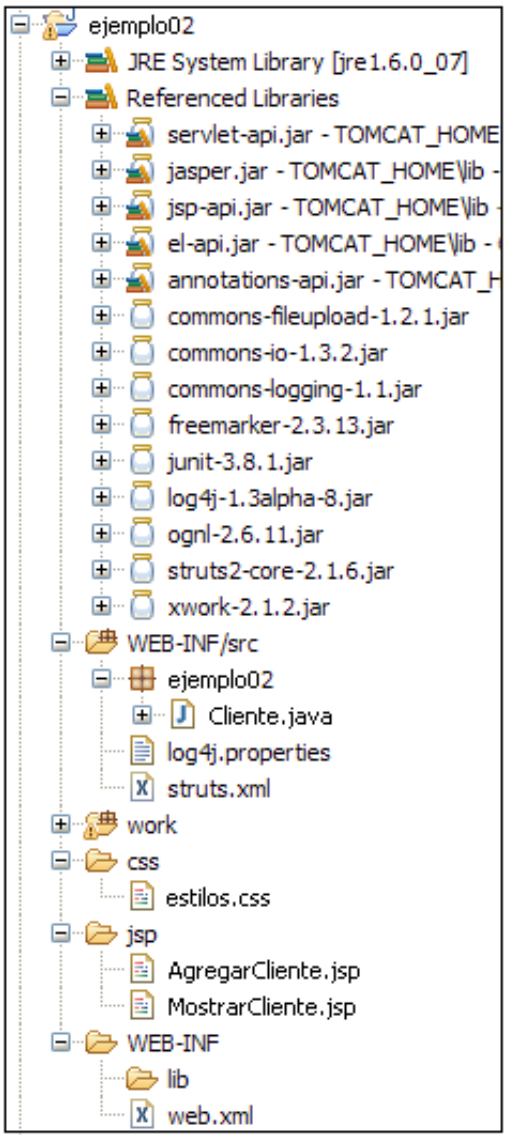
En el archivo *struts-default.xml* presente en el archivo *struts2-core-version.jar*, están definidos los resultados predeterminados y los interceptores. Por este motivo, podemos utilizar este archivo por defecto sin volverlo a escribir en nuestro archivo *struts.xml* para que sea más pequeño y legible.

La estructura del archivo *struts-default.xml* es la siguiente:

```
<?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>
    <bean class="com.opensymphony.xwork2.ObjectFactory"
name="xwork" />
    <bean type="com.opensymphony.xwork2.ObjectFactory"
name="struts"
class="org.apache.struts2.impl.StrutsObjectFactory" />
    ...
    <package name="struts-default" abstract="true">
        <result-types>
            <result-type name="chain"
class="com.opensymphony.xwork2.ActionChainResult"/>
            ...
        </result-types>

        <interceptors>
            <interceptor name="alias"
class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
            ...
        </interceptors>

        <default-interceptor-ref name="defaultStack"/>
    </package>
</struts>
```

Nuestra primera aplicación con Struts

Vamos a retomar nuestra aplicación de administración de clientes con el identificador y la contraseña para adaptarla a Struts.



Árbol del proyecto ejemplo02

El descriptor de implementación *web.xml* contiene únicamente la declaración de la utilización del framework Struts.

```
Código: /WEB-INF/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">

    <display-name>Struts Blank</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>

</web-app>
```

El archivo de configuración del enrutamiento *struts.xml* se presenta a continuación. Nos encontramos con las acciones posibles, que son *Agregar_Cliente.action* y *ConfirmarAgregar_Cliente.action*. Utilizamos igualmente dos constantes para suprimir las invocaciones dinámicas y para pasar al modo de desarrollo. También nos encontramos con la definición del paquete con el nombre *ejemplo02*. La acción*Agregar_Cliente.action* realiza una simple redirección a la vista JSP de añadido. La acción*ConfirmarAgregar_Cliente.action* utiliza la clase `Cliente`, pero como el parámetro *método* no está definido en la acción, por defecto se llamará al método `execute()`.

```
Código: 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="true" />
    <package name="ejemplo02" namespace="/" extends="struts-default">
         <default-action-ref name="Agregar_Cliente" />

         <action name="Agregar_Cliente">
             <result>/jsp/AgregarCliente.jsp</result>
         </action>

         <action name="ConfirmarAgregar_Cliente"
         class="ejemplo02.Cliente">
             <result>/jsp/MostrarCliente.jsp</result>
         </action>
    </package>
</struts>
```

Ahora, podemos presentar la clase de acción llamada `Cliente`. Esta clase utilizar las dos propiedades del formulario, el *identificador* y la *contraseña*. Este archivo es una simple clase POJO. Cada propiedad posee descriptores de acceso que serán reasignados automáticamente durante la llamada de la acción gracias al interceptor `params`. Así, los métodos `setIdentificador()` y `setContrasena()` serán ejecutados y afectados por Struts.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import java.io.Serializable;

@SuppressWarnings("serial")
public class Cliente implements Serializable {

    private String identificador;
    private String contrasena;

      public String getIdentificador() {
             return identificador;
      }

      public void setIdentificadort(String identificador) {
             this.identificador= identificador;
      }
```

```
	public String getContrasena() {
		return contrasena;
	}

	public void setContrasena(String contrasena) {
		this.contrasena= contrasena;
	}

	public String execute()
	{
		System.out.println("En el método de la acción");
		return "success";
	}
}
```

Gracias a Struts, ya no hay necesidad de administrar las recuperaciones de los parámetros en la consulta (*request.getParameter()* o *request.getAttribute()*), ni de administrar los reenvíos de los de parámetros en la respuesta (*request.setAttribute()*). El simple hecho de utilizar los descriptores de acceso de cada propiedad permite administrar los parámetros de manera global.

La página JSP */jsp/AgregarCliente.jsp* tiene la siguiente estructura:

```
Código: /jsp/AgregarCliente.jsp
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
	<h3>Agregar un cliente</h3>
	<form method="post" action="ConfirmarAgregar_Cliente.action">
	<table>
	<tr>
		<td>Identificador:</td>
		<td><input type="text" name="identificador"/></td>
	</tr>
	<tr>
		<td>Contrase&ntilde;a:</td>
		<td><input type="text" name="contrasena"/></td>
	</tr>
	<tr>
		<td colspan="2" align="center"><input type="submit"
value="Agregar un cliente"/></td>
	</tr>
	</table>
	</form>
</div>
</body>
</html>
```

Por último, el archivo JSP */jsp/MostrarCliente.jsp* tiene la siguiente estructura:

```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
```

```
	<p>
		<h4>Informaci&oacute;n sobre el cliente:</h4>
		Identificador: <s:property value="identificador"/><br/>
		Contrase&ntilde;a: <s:property value="contrasena"/><br/>
	</p>
</div>
</body>
</html>
```

Especificamos la utilización de la biblioteca de etiquetas de Struts con la siguiente línea:

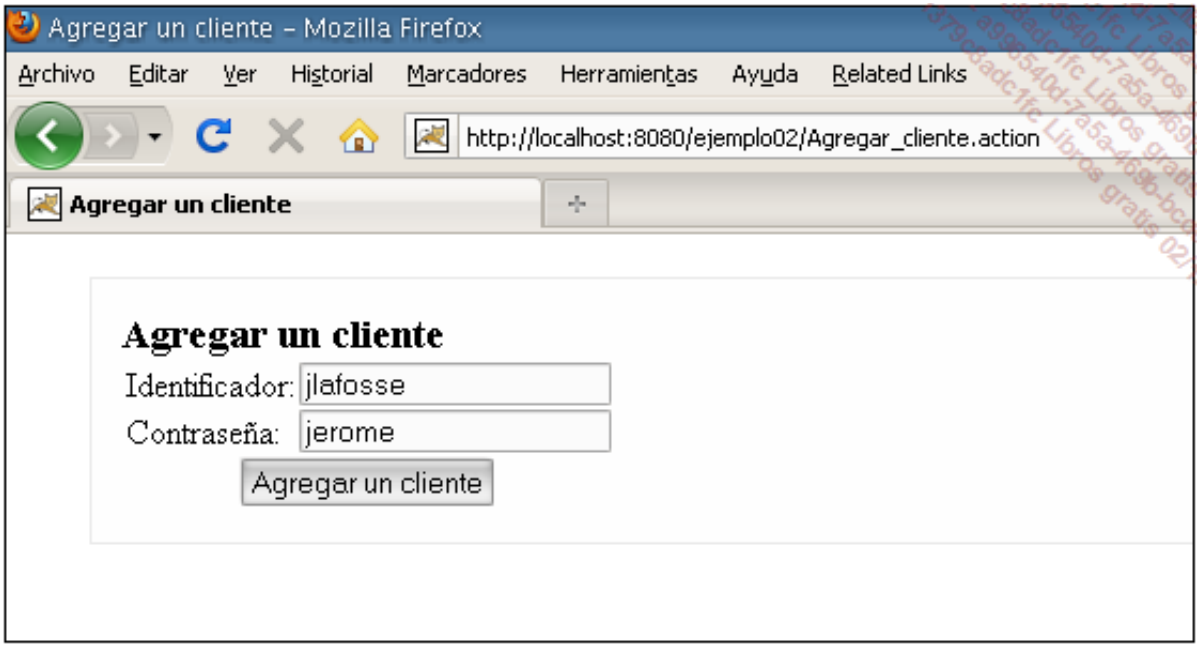
```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

Por último, cada propiedad es accesible con la etiqueta Struts `<s:property/>` gracias a los getters de la clase de acción.

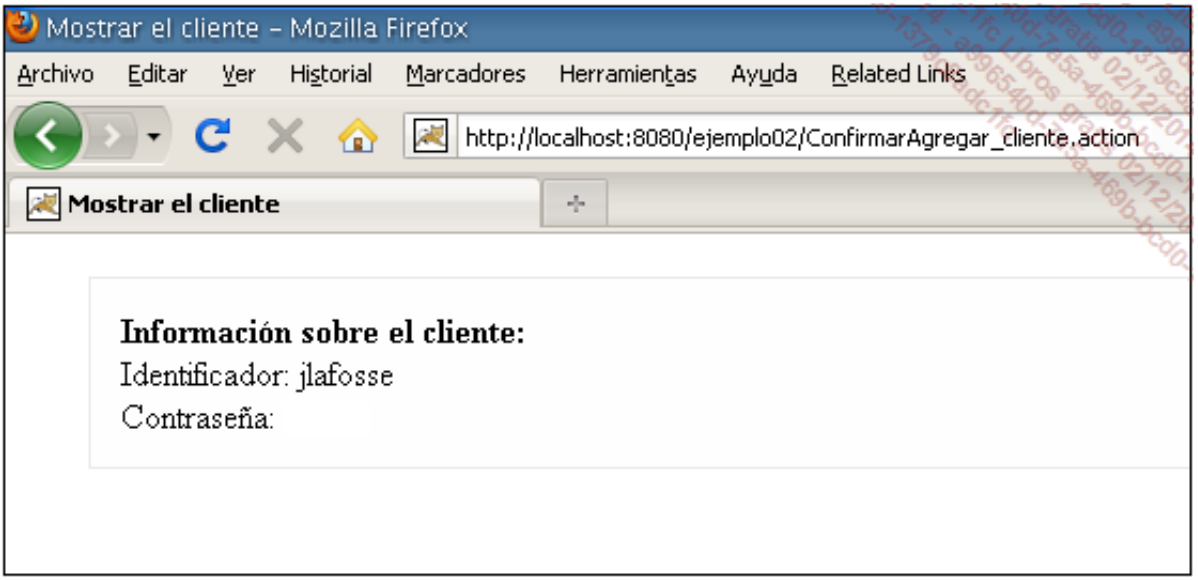
```
<s:property value="identificador"/>
```

Podemos utilizar esta aplicación, con la siguiente URL para agregar un cliente:

http://localhost:8080/ejemplo02/Agregar_cliente.action

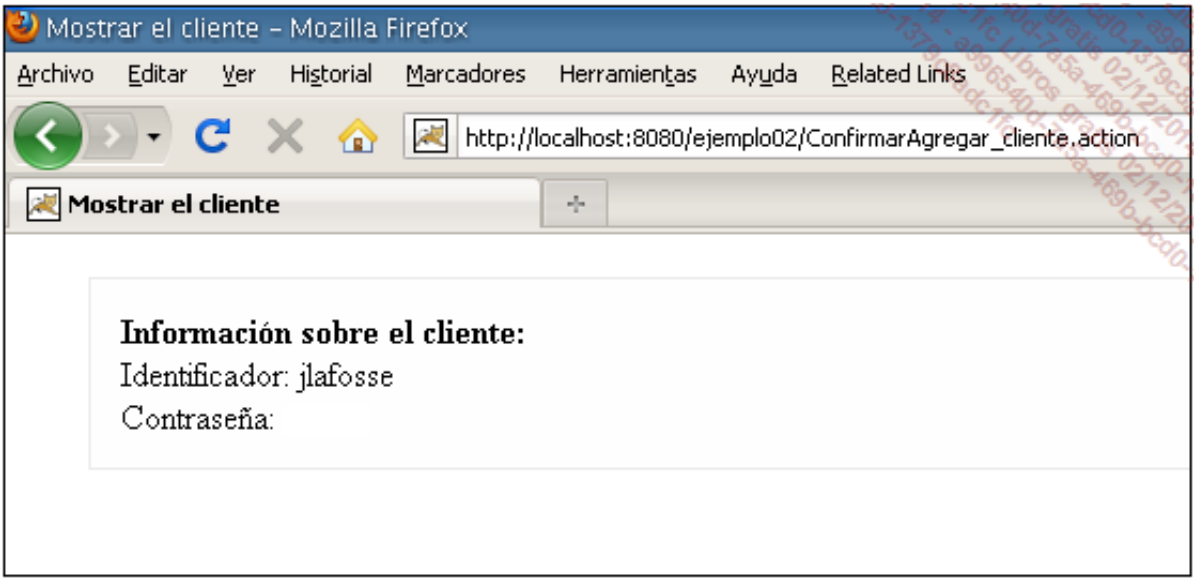


Formulario de ejemplo02



Visualización del cliente de ejemplo02

Nuestra primera aplicación con Struts ya esta operativa. Permite administrar dos atributos presentes en un formulario. La simple declaración de los descriptores de acceso permite administrar toda la cadena de desarrollo (consulta/respuesta). Para probarlo, podemos simplemente suprimir el setter*setContrasena()* en la clase `Cliente` y comprobar el resultado.



Visualización del cliente sin el setter de la contraseña

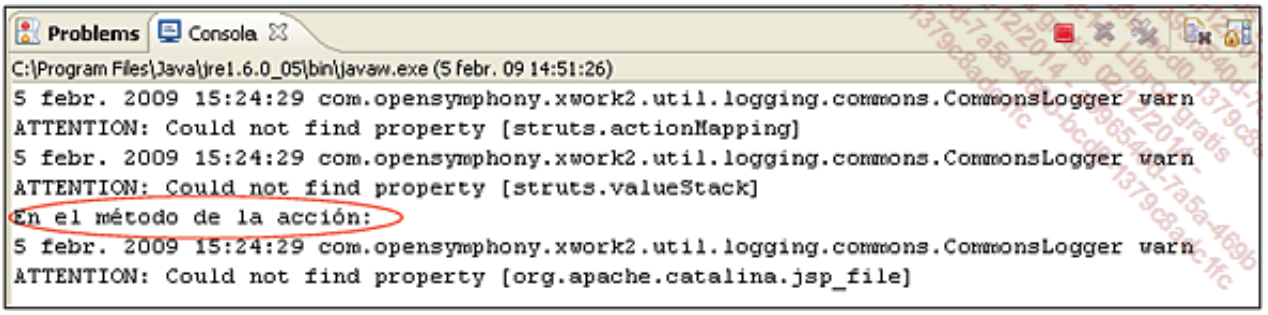
En resumen

Este capítulo ha presentado el funcionamiento general de Struts con los interceptores, el archivo de configuración de la aplicación *struts.xml* y las distintas etiquetas XML de declaración. También se han detallado los archivos de configuración del framework y se ha explicado en profundidad un primer ejemplo de administración de clientes.

Presentación

En el capítulo anterior se ha presentado un primer proyecto simple de Struts. El método `execute()` de la clase de acción `Cliente` muestra un registro en la consola de Java con la ayuda del método `System.out.println()`.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import java.io.Serializable;

@SuppressWarnings("serial")
public class Cliente implements Serializable {

    private String identificador;
    private String contrasena;

      ...

      public String execute()
      {
             System.out.println("En el método de la acción");
             return "success";
      }
}
```



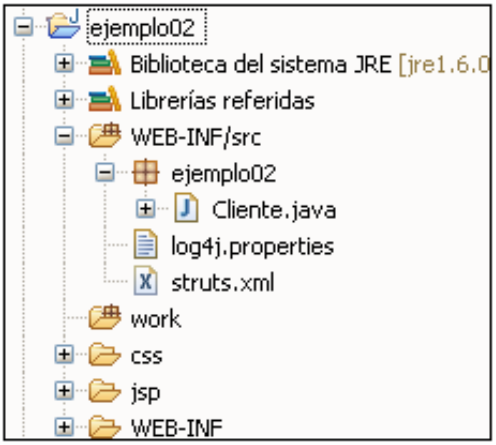
Registro de Java

Observamos que Struts es muy detallado, muestra varios registros de advertencia para una simple aplicación. El framework reposa sobre la interfaz de registro Log4J (http://logging.apache.org/) y, por lo tanto, puede utilizar un archivo de configuración para eliminar estos registros o adaptarlos según nuestras necesidades.

Estos registros corresponden a las herramientas FreeMarker y XWork. Para eliminarlos, basta con crear un archivo llamado *log4j.properties* en el directorio */WEB-INF/src* (o */WEB-INF/classes*) de la aplicación al mismo nivel que el archivo de gestión *struts.xml*.

El archivo de configuración más simple para detener los registros es el siguiente:

```
Código: ejemplo02.log4j.properties
#definición del nivel y de los Appender del rootLogger (orden: DEBUG
- INFO - WARN - ERROR - FATAL)
log4j.logger.freemarker=OFF
log4j.logger.com.opensymphony.xwork2=OFF
log4j.logger.org.apache=INFO
log4j.rootLogger=DEBUG
```



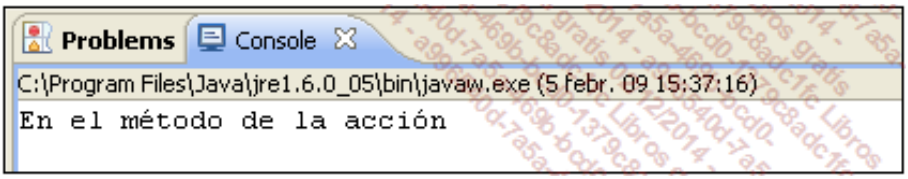
Árbol y archivo log4j

A continuación, es necesario copiar el archivo de Log4J en formato *.jar*, *log4j-version.jar* en el directorio de bibliotecas de la aplicación */WEB-INF/lib*. Ahora podemos volver a ejecutar el proyecto y comprobar el registro en la consola Java.

```
INFO: La recarga de este contexto a iniciado
log4j: INFO Using URL [file:C:/PROYECTOWEB/ejemplo02/WEB-INF/classes/log4j.properties] for automatic log4 configuration
```

Registro de la consola Java con Log4J

Podemos mostrar un nuevo formulario de registro de un cliente y confirmar la acción para visualizar el seguimiento sin registro.



Registro de la consola Java sin Log4J

Administración de la depuración

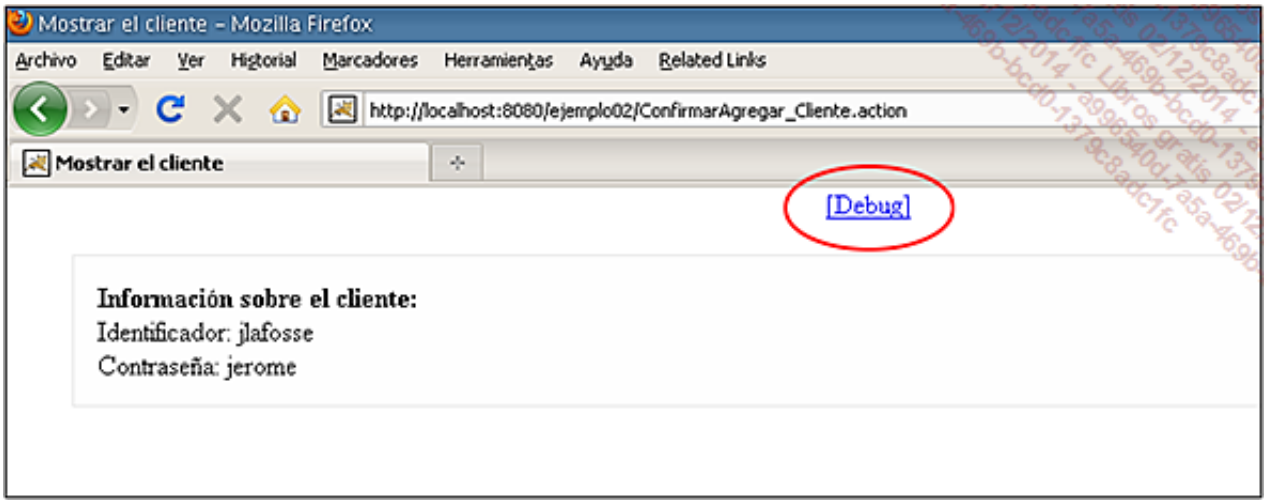
Ahora, la depuración es una operación sencilla con Struts. El framework propone una etiqueta XHTML`<s:debug/>` para mostrar los registros con la información de los parámetros, la sesión en curso o incluso los objetos.

La etiqueta `<s:debug/>` puede ubicarse en cualquier página JSP. Esta etiqueta posee un único parámetro opcional llamado *id*.

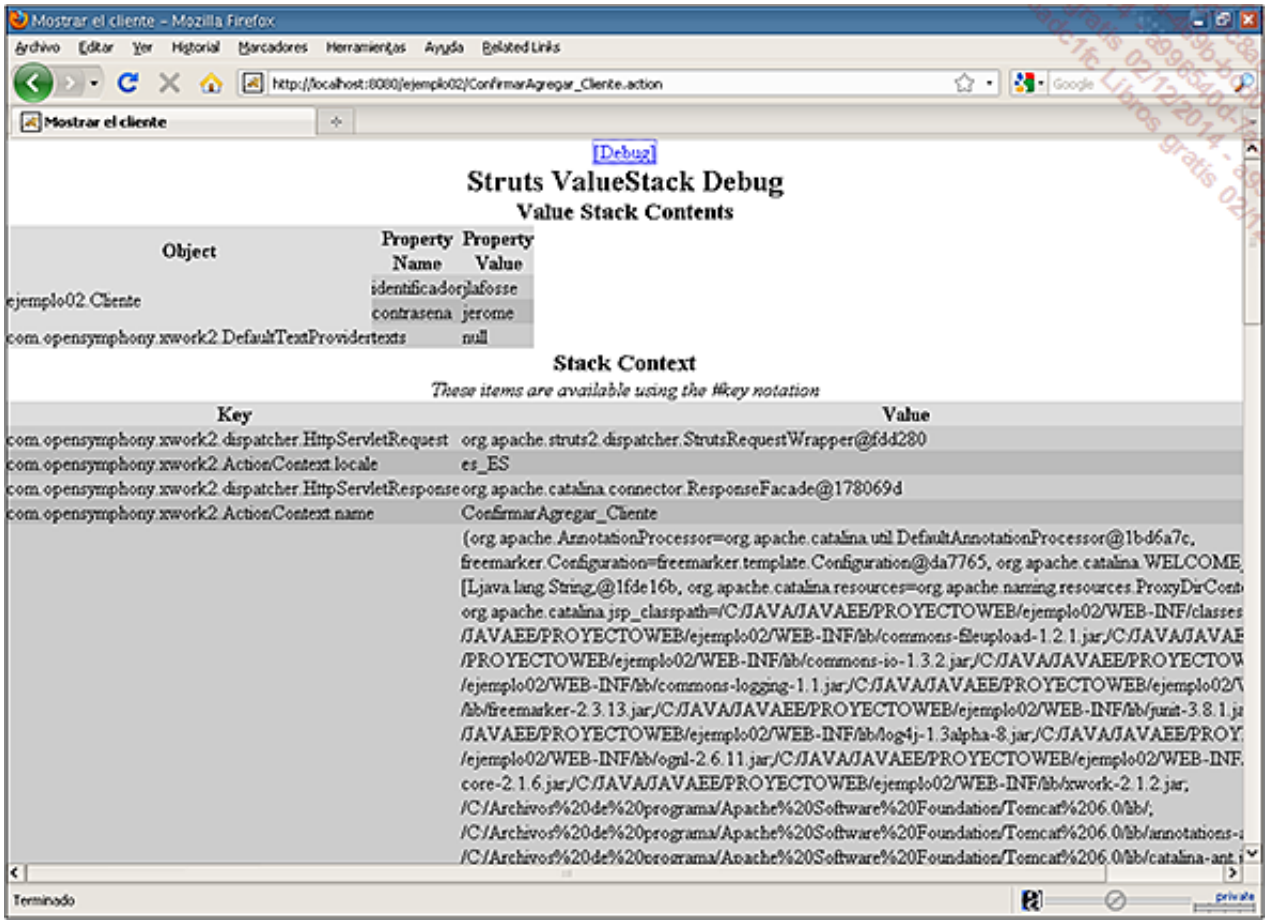
A continuación, retomamos nuestra página *MostrarCliente.jsp* y añadimos esta etiqueta al principio del archivo.

```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
...
</html>
```

Luego, hacemos clic sobre el enlace llamado **[debug]**, lo cual nos permite visualizar la pila de registros de los objetos presentes en el contexto de la aplicación. Esta etiqueta permite depurar fácilmente una aplicación al igual que visualizar las propiedades de la acción y el contenido de los objetos presentes en la sesión o en la colección de la aplicación.



Etiqueta de depuración <s:debug/>

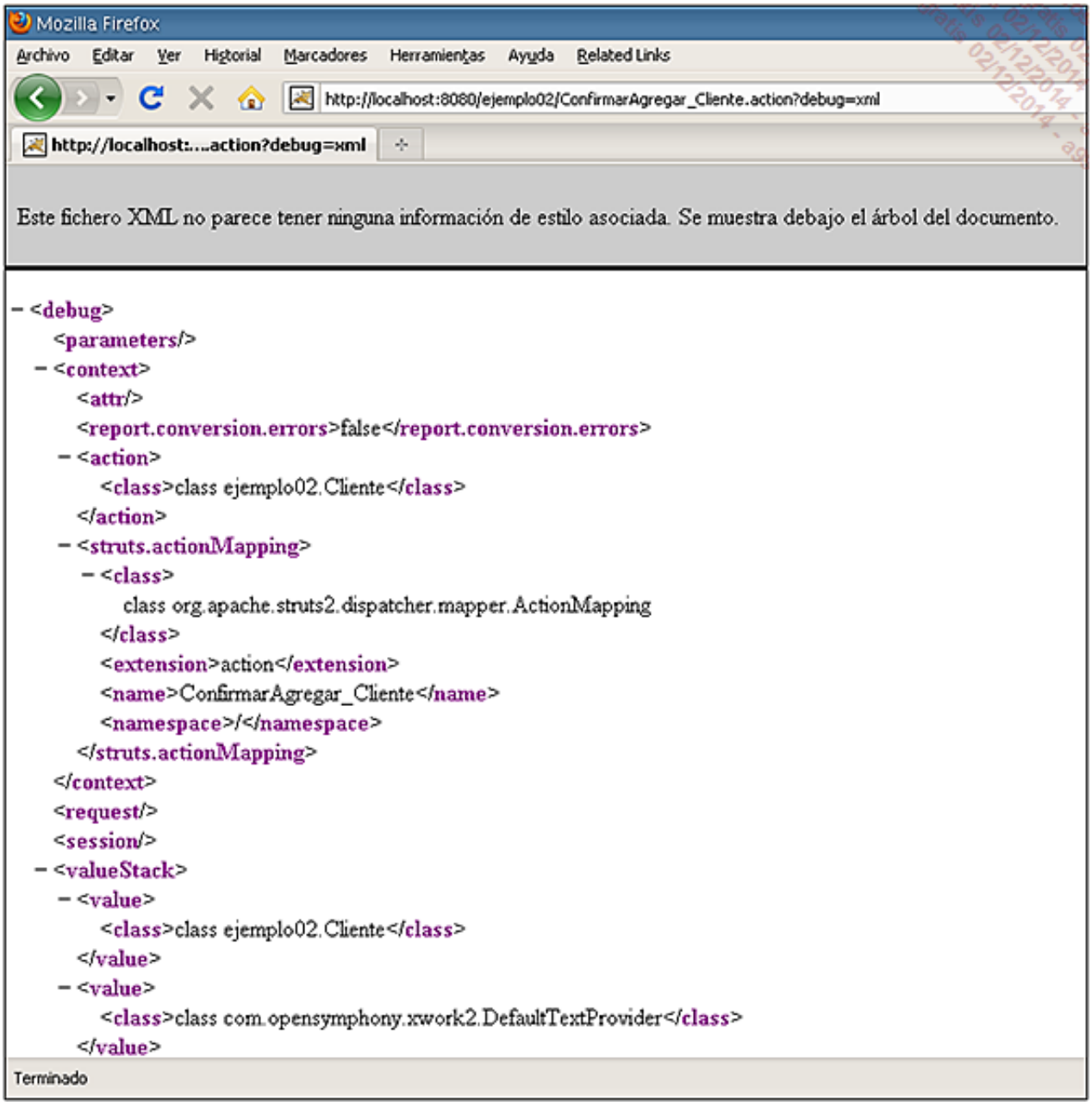


Visualización de los registros de la depuración

En Struts, la depuración se realiza mediante el interceptor *debugging*. Este interceptor está presente en la configuración predeterminada de Struts. Podemos ejecutar este interceptor directamente a partir de una URL utilizando el parámetro *debug=xml* o *debug=console*. El parámetro *debug=xml*permite visualizar un árbol XML que contiene la lista de valores de registro y los objetos.

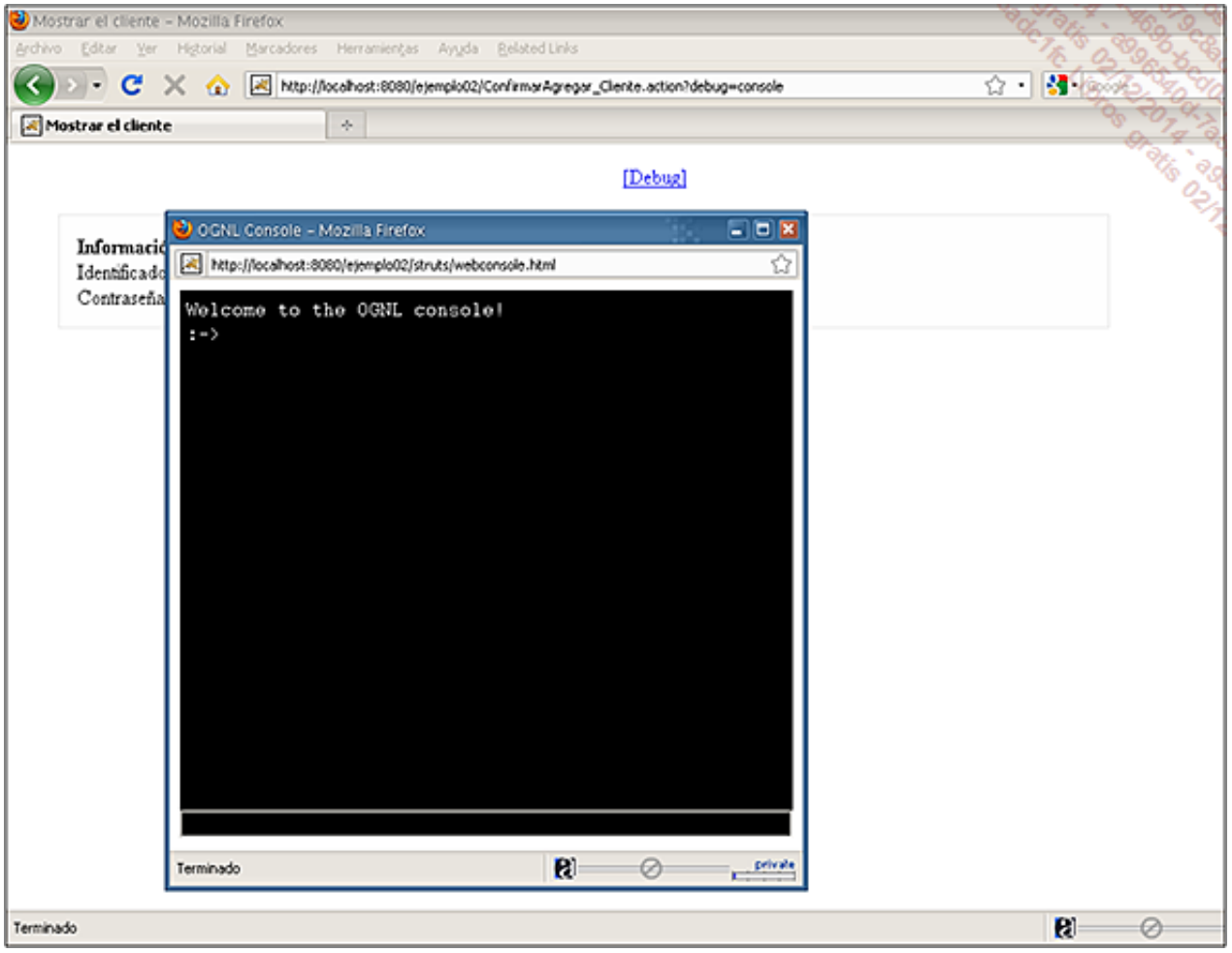
A continuación, se presenta un ejemplo con la ejecución de la siguiente URL:

http://localhost:8080/exemple02/ConfirmarAgregar_Cliente.accion?debug=xml



Visualización del árbol de depuración XML

Por último, el seguimiento en modo consola permite visualizar una ventana con la información sobre la aplicación en curso. En esta consola, podemos introducir expresiones de tipo Object-Graph Navigation Language (OGNL) para visualizar los parámetros de la página.



Consola de depuración OGNL

Esta consola no funciona actualmente con el navegador Internet Explorer, pero sí con Firefox.

Administración de la creación de perfiles (Profiling)

Struts incorpora por defecto una herramienta de administración de la creación de perfiles para nuestras aplicaciones. La creación de perfiles permite dar información precisa sobre el curso de una acción. La creación de perfiles permite, por ejemplo, mejorar el código o encontrar una parte del código particularmente lenta durante su ejecución. Struts proporciona un seguimiento sobre el tiempo de ejecución de las diferentes secciones, de sus filtros, interceptores y resultados. Cada uno de estos servicios utiliza la clase `UtilTimeStack` que se encuentra en el paquete*com.opensymphony.xwork2.util.profiling*. Por defecto, no se mostrarán los registros. Por contra, si se ha activado el seguimiento para una acción, por ejemplo, el resultado se mostrará en modo de registro en la consola Java o en el archivo de registro del servidor Java EE (*catalina.out* para Tomcat).

Para activar la creación de perfiles con Struts, existen varias técnicas:

1) Ejecutar una consulta con el parámetro *profiling=true* o *profiling=yes* en la URL de la aplicación. http://localhost:8080/ejemplo02/ConfirmarAgregar_Cliente.accion?profiling=yes

Para detener la creación de perfiles de una aplicación, basta con pasar del parámetro *profiling=no* o*profiling=false* http://localhost:8080/ejemplo02/ConfirmarAgregar_Cliente.accion?profiling=no

2) Activar el método `UtilTimerStack.setActive(true)` en la función que se va a perfilar.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import java.io.Serializable;
import com.opensymphony.xwork2.util.profiling.UtilTimerStack;

@SuppressWarnings("serial")
public class Cliente implements Serializable {

   ...
      public String execute()
      {
             UtilTimerStack.setActive(true);
             System.out.println("En el método de la acción");
             return "success";
      }
}
```

3) Activar la constante del sistema *UtilTimerStack.ACTIVATE_PROPERTY*.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import java.io.Serializable;
import com.opensymphony.xwork2.util.profiling.UtilTimerStack;

@SuppressWarnings("serial")
public class Cliente implements Serializable {
   ...
      public String execute()
      {


System.setProperty(UtilTimerStack.ACTIVATE_PROPERTY,"true");
             System.out.println("En el método de la acción");
             return "success";
      }
}
```

Para activar la creación de perfiles de una aplicación de Struts, ésta debe estar configurada en modo de desarrollo struts.devMode=true. La creación de perfiles únicamente puede

mostrarse cuando se ha activado el seguimiento y no se ha prohibido por Log4J y su archivo de propiedades.

A continuación, se presenta un ejemplo de registro con la siguiente URL:

http://localhost:8080/ejemplo02/ConfirmarAgregar_Cliente.accion?profiling=true

```
INFO: [500ms] - Handling request from Dispatcher
  [0ms] - create DefaultActionProxy:
  [0ms] - actionCreate: ConfirmarAgregar_Cliente
  [453ms] - invoke:
  [453ms] - interceptor: exception
  [453ms] - invoke:
  [453ms] - interceptor: alias
  [453ms] - invoke:
  [453ms] - interceptor: servletConfig
  [453ms] - invoke:
  [453ms] - interceptor: i18n
  [453ms] - invoke:
  [453ms] - interceptor: prepare
  [453ms] - invoke:
  [453ms] - interceptor: staticParams
  [453ms] - invoke:
  [453ms] - interceptor: actionMappingParams
  [453ms] - invoke:
  [453ms] - interceptor: params
  [453ms] - invoke:
  [453ms] - interceptor: conversionError
  [453ms] - invoke:
  [453ms] - interceptor: validation
  [125ms] - invoke:
  [125ms] - interceptor: workflow
  [125ms] - invoke:
  [0ms] - invokeAction: ConfirmarAgregar_Cliente
  [125ms] - executeResult: success
```

Podemos igualmente crear perfiles de las actividades específicas con los métodos `push()` y `pop()` de la clase estática `UtilTimerStack`. El siguiente es un ejemplo para realizar el seguimiento únicamente de los accesos a una base de datos.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.util.profiling.UtilTimerStack;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {
   ...
      public String execute()
      {
             String activite="database access";
             UtilTimerStack.push(activite);
             try
             {
                  System.out.println("En el método de la
acción y el acceso a la base de datos");
             }
             catch(Exception e)
             {
                  UtilTimerStack.pop(activite);
             }
```

En resumen

Este capítulo ha explicado en primer lugar cómo gestionar los registros con el framework Struts. Struts utiliza un sistema de visualización muy detallado y debe configurarse correctamente en la fase de desarrollo. En un segundo punto, hemos presentado la etiqueta XHTML `<s:debug/>` incorporada por defecto con la biblioteca de etiquetas de Struts. Por último, el último párrafo explica el funcionamiento de la creación de perfiles en los desarrollos con la herramienta Struts.

Presentación

Struts utiliza un controlador principal para la administración del enrutamiento y de los parámetros. Así, podemos concentrarnos en la realización y la codificación de las distintas acciones del proyecto. Cada acción de Struts está definida en el archivo de configuración *struts.xml* (o en las clases con la técnica de configuración cero). Struts permite igualmente la utilización de expresiones regulares y las invocaciones dinámicas de métodos.

Clases de acción

La creación de acciones es el trabajo más importante de la fase de desarrollo que se debe realizar con la ayuda del framework Struts. Por ejemplo, tendremos acciones para mostrar la página de inicio, administrar los artículos, ocuparse de la autenticación del cliente y otras. Algunas operaciones son simples y no requieren de ninguna clase, sino que realizan simplemente una redirección, mientras que otras efectúan procesamientos complejos asociados a una clase de acción.

Una clase de acción es simplemente una clase Java. Por lo tanto, esta clase contiene atributos y métodos. Sin embargo, es necesario respetar algunas reglas para tener una verdadera clase de acción.

- Cada atributo debe estar asociado a sus descriptores de acceso (*getter* y *setter*). Las reglas son las mismas que para los JavaBeans.
- Una clase de acción debe tener un constructor por defecto sin argumento. Sin embargo, si no creamos ningún constructor por defecto, Struts lo hará por nosotros durante la compilación. También podemos utilizar otro constructor (por inicialización), pero en este caso el constructor por defecto sigue siendo obligatorio.
- Una clase de acción debe tener al menos un método para realizar la acción. Por defecto, este método se llama `execute()`, pero puede llevar cualquier nombre asociado con la etiqueta`<action/>` y el atributo `method` presentes en el archivo de configuración de la acción *struts.xml*.
- Una clase de acción puede estar asociado a varios métodos. En este caso, la clase de acción gestionará, por ejemplo, la visualización de los clientes, el formulario de creación, la confirmación de la creación, la consulta, la visualización del formulario de modificación, la confirmación de la modificación y, por último, la eliminación.
- Una clase de acción no necesita heredar de una clase específica o de una interfaz determinada. Sin embargo, es muy fácil y aconsejable heredar de la clase `ActionSupport`presente en el paquete `com.opensymphony.xwork2.action`.

La clase `com.opensymphony.xwork2.actionSupport` es la clase de acción predeterminada de Struts. El framework crea una instancia de esta clase si no se ha especificado ninguna declaración. Si implementamos la clase `ActionSupport`, estarán disponibles las siguientes constantes:

- *SUCCESS*: esta constante indica que la ejecución de la acción es correcta y que se debe mostrar la página de confirmación adaptada.
- *NONE*: esta constante indica que la ejecución de la acción es correcta, pero que no se debe devolver ningún resultado.
- *ERROR*: esta constante indica que la ejecución de la acción es incorrecta y que el usuario debe ser redirigido a una página de error.
- *INPUT*: esta constante indica un error de validación de las entradas o los datos introducidos por el usuario. Por lo general, será necesario actualizar la página de introducción de datos sin perder los datos.
- *LOGIN*: esta constante indica que la acción no puede ejecutarse ya que el usuario no se ha autenticado y fuerza la visualización de la página de identificación.

Por lo tanto estos valores se utilizan para sustituir las cadenas de caracteres *success*, *input*... Además, la herencia de la clase `ActionSupport` permite igualmente sobrecargar varios métodos como la función`validate()` para realizar validación es del usuario en programación.

Una de las principales ventajas de utilizar Struts es su administración de las resistencias de los atributos con la simple declaración en las clases y los descriptores de acceso. Esto es posible gracias al interceptor params que administra la pila de los datos. Esta pila de datos es accesible en las páginas JSP.

Código: ejemplo02.Cliente.java

```
package ejemplo02;

import com.opensymphony.xwork2.actionSupport;
import com.opensymphony.xwork2.util.profiling.UtilTimerStack;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;

        public Cliente()
        {

        }
        // getter identificador
        public String getIdentificador() {
               return identificador;
        }
        // setter identificador
        public void setIdentificador(String identificador) {
               this.identificador= identificador;
        }
        // getter contrasena
        public String getContrasena() {
               return contrasena;
        }
        // setter contrasena
        public void setContrasena(String contrasena) {
               this.contrasena=  contrasena;
        }

        public String execute()
        {
               System.out.println("En el método de la acción");
               return "success";
        }
}
```

Dado que las clases de acción de Struts han pasado a ser clases de POJO, la realización de pruebas o el uso de clases es relativamente simple. Podemos instanciar las clases y acceder directamente a las propiedades mediante los descriptores de acceso.

El siguiente es un ejemplo de código para la utilización de la clase `Cliente.java`.

```
Cliente cliente=new Cliente() ;
cliente.setIdentificador("jlafosse");
cliente.setContrasena("jerome");
String resultat=cliente.execute();
if(resultat.equals("success"))
{
    System.out.println("Correcto");
}
else
{
    System.out.println("Error");
}
```

Administración de los recursos

Las clases de acción permiten el acceso a los recursos externos como los Servlets de Java. Los recursos externos son los siguientes:

- la clase `ServletContext`;
- la clase `HttpSession`;
- la clase `HttpServletRequest`;
- la clase `HttpServletResponse`.

Struts ofrece la posibilidad de acceder a estos recursos utilizando la herencia de clases o la implementación de interfaces.

1. Acceso a los recursos por clase

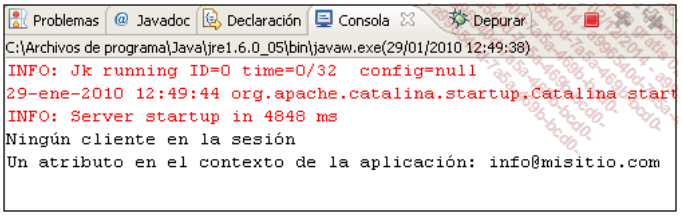
Existen dos clases que permiten acceder a los recursos, `com.opensymphony.xwork2.actionContext` y`org.apache.struts2.ServletActionContext`. La clase estática `ServletActionContext` ofrece tres métodos estáticos `getRequest()`, `getResponse()` y `getServletContext()` para acceder a los recursos.

Un punto muy importante de la utilización de la clase estática `ServletActionContext` es su ámbito de aplicación. De hecho, es imposible utilizar esta clase en un constructor de una clase de acción. En el constructor, la clase de acción todavía no conoce la clase estática ya que esta última aún no se le ha sido pasada.

A continuación, se muestra un pequeño ejemplo de utilización de la clase estática`ServletActionContext` en un método de acción.

```
Código: ejemplo02.Cliente.java
package ejemplo02;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.util.profiling.UtilTimerStack;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

  ...

      public String execute()
      {
              HttpServletRequest
request=ServletActionContext.getRequest();
              HttpSession session=request.getSession();
              if(session.getAttribute("cliente")==null)
              {
                   System.out.println("Ningún cliente en la sesión");
              }
              else
              {
                   System.out.println("Un cliente en la sesión");
              }

              ServletContext
context=ServletActionContext.getServletContext();
              System.out.println("Un atributo en el contexto de
la aplicación: "+context.getInitParameter("email"));


              return "success";
      }
}
```

```
Código: /WEB-INF/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">
    ...
      <!-- Parámetros globales -->
      <context-param>
      <param-name>email</param-name>
      <param-value>info@misitio.com</param-value>
      </context-param>

</web-app>
```



Acceso a los recursos

2. Acceso a los recursos por interfaz

El framework proporciona cuatro interfaces para acceder a los recursos externos:

- *org.apache.struts2.util.ServletContextAware*,
- *org.apache.struts2.interceptor.ServletRequestAware*,
- *org.apache.struts2.interceptor.ServletResponseAware*,
- *org.apache.struts2.interceptor.SessionAware*.

La interfaz *ServletContextAware* posee un método llamado `setServletContext()` que permite acceder al objeto *ServletContext* en la clase de acción. Cuando se ejecuta una acción, el framework comprueba si la interfaz se ha implementado y si una función sobrecarga la declaración. En el interior de este método, debemos asignar una variable *servletContext* para recuperar el objeto.

La interfaz *ServletRequestAware* posee un método llamado `setServletRequest()` que permite acceder al objeto *HttpServletRequest* en la clase de acción. Cuando se ejecuta una acción, el framework comprueba si la interfaz se ha implementado y si una función sobrecarga la declaración. En el interior de este método, debemos asignar una variable *request* para recuperar el objeto.

La interfaz *ServletResponseAware* posee un método llamado `setServletResponse()` que permite acceder al objeto *HttpServletResponse* en la clase de acción. Cuando se ejecuta una acción, el framework comprueba si la interfaz se ha implementado y si una función sobrecarga la declaración. En el interior de este método, debemos asignar una variable *response* para recuperar el objeto.

La interfaz *SessionAware* posee un método llamado `setSession()` que permite acceder al objeto*HttpSession* en la clase de acción. Cuando se ejecuta una acción, el framework comprueba si la interfaz se ha implementado y si una función sobrecarga la declaración. En el interior de este método, debemos asignar una variable *session* para recuperar la colección de atributos presentes en la sesión.

Struts transmite una instancia de la clase `org.apache.struts2.dispatcher.SessionMap` como parámetro. Esta clase hereda de la clase `AbstractMap` que a su vez implementa la clase `Map`. La clase`SessionMap` ofrece varios métodos para poder manipular los objetos de tipo *HttpSession*. Los métodos propuestos por la clase `SessionMap` son los siguientes:

- `invalidate()`: este método permite eliminar la sesión.
- `clear()`: este método permite borrar todos los atributos del objeto *HttpSession*.
- `get(cle)`: este método devuelve un atributo determinado por la clave configurada. Este método devuelve *null* si el objeto no existe o no se encuentra.
- `put(clave, valor)`: este método permite guardar un atributo de la sesión. Si el objeto*HttpSession* es *null*, se creará un nuevo objeto *HttpSession*.
- `remove(clave)`: este método permite suprimir un atributo específico de la sesión.

El siguiente ejemplo permite aplicar el acceso a los recursos externos utilizando las interfaces que ofrece Struts. El ejemplo sigue el mismo principio que el proyecto *ejemplo02*. El archivo de administración de la aplicación *struts.xml* es el siguiente:

```
Código: 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="true" />

    <package name="ejemplo03" namespace="/" extends="struts-
default">
        <default-action-ref name="Agregar_Cliente" />

        <action name="Agregar_Cliente">
           <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo03.Cliente" method="agregar">
        <result name="input">/jsp/AgregarCliente.jsp</result>
        <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>

        <action name="Eliminar_Cliente" class="ejemplo03.Cliente"
method="eliminar">
        <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>
    </package>
</struts>
```

La acción *Agregar_Cliente* permite mostrar únicamente el formulario JSP de registro de un nuevo cliente. La acción *ConfirmarAgregar_Cliente* permite agregar la información del cliente a la sesión si no se ha dejado ningún campo vacío. De lo contrario, el internauta será redirigido a la página de introducción de datos. Por último, la acción *Eliminar_Cliente* permite eliminar la información del cliente de la sesión. Las expresiones de JSP Expression Language *${identificador}* y *${contrasena}*presentes en el atributo `value` de las etiquetas `<input/>` permiten conservar los datos introducidos por el usuario.

La siguiente página permite visualizar los valores de los atributos presentes en la sesión, que son*Identificadorsesion* y *contrasenasesion*. El enlace propuesto ejecuta el método `eliminar()` de la clase de acción para eliminar la sesión.

Por último, la clase `Cliente` permite administrar los atributos del usuario en la sesión durante una autenticación. La clase `Cliente` accede a la sesión del usuario ya que se implementa la interfaz`SessionAware` y se sobrecarga del método `setSession(Map map)`. La acción *agregar()* permite registrar los parámetros del cliente en la sesión y el método `eliminar()` permite borrar esta misma información de la sesión. También se utiliza una comprobación de los datos introducidos (no vacíos) en esta clase.

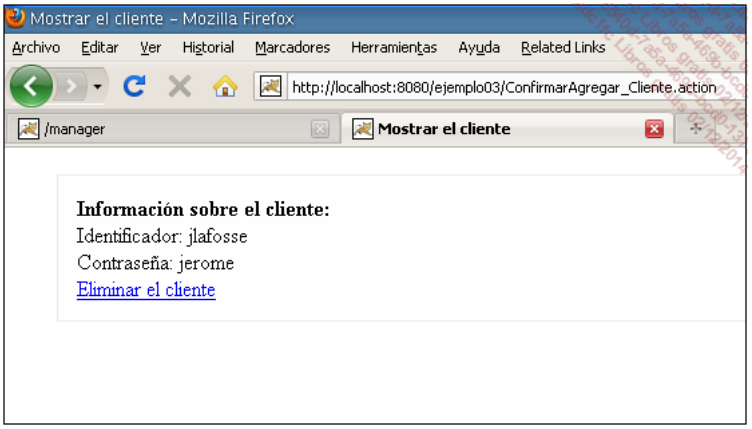
Podemos probar la aplicación *ejemplo03* con este enlace haciendo pruebas con campos vacíos y campos completados. Por último podemos eliminar los atributos de la sesión utilizando un enlace adaptado.

http://localhost:8080/ejemplo03/Agregar_Cliente.action

```
Código: /jsp/AgregarCliente.jsp
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <form method="post" action="ConfirmarAgregar_Cliente.action">
      <table>
      <tr>
            <td>Identificador:</td>
            <td><input type="text" name="identificador" value="$
{identificador}"/></td>
      </tr>
      <tr>
            <td>Contrase&ntilde;a:</td>
            <td><input type="text" name="contrasena" value="$
{contrasena}"/></td>
      </tr>
      <tr>
            <td colspan="2" align="center"><input type="submit"
value="Agregar el cliente"/></td>
      </tr>
      </table>
      </form>
</div>
```

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

```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4>Informaci&oacute;n sobre el cliente:</h4>
            Identificador: <s:property
value="#session.identificadorsesion"/><br/>
            Contrase&ntilde;a: <s:property
value="#session.contrasenasesion"/><br/>
      </p>
      <a href="Eliminar_Cliente.action">Eliminar el cliente</a>
</div>
</body>
</html>
```



Visualización de la información del cliente ejemplo03

3. Trasladar parámetros

Las declaraciones de acción también pueden recibir parámetros estáticos en el archivo de configuración *struts.xml*. Esta técnica permite asignar valores a las propiedades de la clase. Si retomamos nuestro ejemplo anterior, podemos agregar una etiqueta `<param/>` en la declaración de la acción. Cada parámetro corresponde a una propiedad de la clase de acción (setter y getter). El interceptor *staticParams* se ocupa del mapping entre los parámetros declarados en el archivo*struts.xml* y el código de la acción.

```
Código: 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>
   ...
        <action name="ConfirmarAgregar_Cliente"
class="ejemplo03.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success">/jsp/MostrarCliente.jsp</result>
            <param name="pagina">frontoffice</param>
        </action>
   ...
</struts>
```

```
Código: ejemplo03.Cliente.java
package ejemplo03;

import java.util.Map;
import org.apache.struts2.dispatcher.SessionMap;
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport implements SessionAware
{
    private String identificador;
    private String contrasena;
    private String pagina;
...
    public String getPagina() {
            return pagina;
    }

    public void setPagina(String pagina) {
            this.pagina= pagina;
    }
...
}
```

Código: /jsp/MostrarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4>Informaci&oacute;n sobre el cliente:</h4>
            Identificador: <s:property
value="#session.identificadorsession"/><br/>
            Contrase&ntilde;a: <s:property
value="#session.contrasenasession"/><br/>
            P&aacute;gina: <s:property value="pagina"/><br/>
      </p>
      <a href="Eliminar_Cliente.action">Eliminar el cliente</a>
</div>
</body>
</html>
```

Uso de los parámetros de acción de ejemplo03

Administración dinámica del mapping

Un proyecto final contiene varias declaraciones de acciones y puede limitar la legibilidad y la facilidad de mantenimiento del archivo de configuración *struts.xml*. En la primera versión de Struts, el archivo de configuración podía contener muchas líneas y, en ocasiones, declaraciones prácticamente idénticas (por ejemplo, administración de los artículos, clientes, categorías...).

Para ofrecer soluciones a estos problemas, Struts ofrece ahora declaraciones de acciones en forma de expresiones regulares o modelos llamados wildcard.

Podemos utilizar un nuevo proyecto llamado *ejemplo04* y agregar una declaración dinámica para el formulario de creación con el fin de mostrar cualquier prefijo introducido antes de la frase_*Cliente.action*.

```
Código: 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>
   ...
        <action name="*_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>
   ...
</struts>
```

El carácter * permite capturar las URL compuestas de cualquier cadena de caracteres. Así, podemos ejecutar la siguiente URL http://localhost: 8080/ejemplo04/Crear_Cliente.action para mostrar el formulario de creación. La parte de la URL que se captura por el carácter * está disponible con el término {1}. Si utilizamos varios caracteres de escape, existen tantos parámetros como caracteres de escape ({1}, {2}, …).

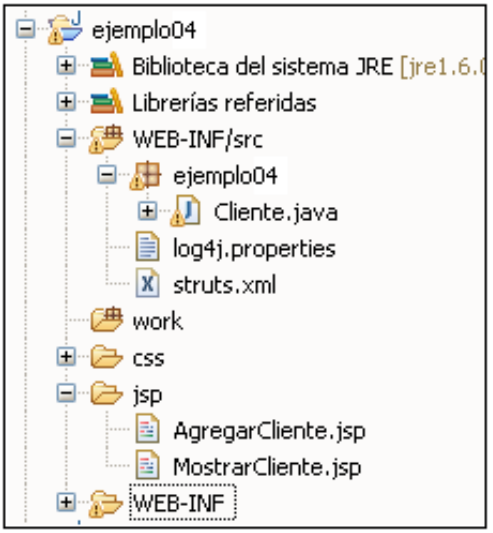
Así, podemos reducir considerablemente el código de nuestro ejemplo utilizando los escapes y la invocación dinámica de métodos.

```
Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="*_Cliente" class="ejemplo04.Cliente"
method="{1}">
            <result name="input">/jsp/cliente/Agregar.jsp</result>
            <result name="success">/jsp/cliente/{1}.jsp</result>
        </action>

    </package>
</struts>
```

Así, la URL http://localhost:8080/ejemplo04/Agregar_Cliente.action ejecutará la acción llamada *_*Cliente* y el método `Agregar()` de la clase `ejemplo04.Cliente`.

De igual modo, la URL http://localhost:8080/ejemplo04/ConfirmarAgregar_Cliente.action ejecutará la

acción llamada *_*Cliente* y el método `ConfirmarAgregar()` de la clase `ejemplo04.Cliente`. Cada método de la clase `Cliente` devuelve, en caso de éxito, a las páginas JSP del mismo nombre. Retomamos ahora las vistas JSP */jsp/cliente/Agregar.jsp* y */jsp/cliente/ConfirmarAgregar.jsp*.



Árbol del proyecto ejemplo04

Este ejemplo, aunque es muy útil, no representa un verdadero proyecto de plataforma Web. Si estudiamos por ejemplo una configuración de un sistema de administración de clientes, deberemos declarar las siguientes acciones:

- Consultar un cliente.
- Mostrar el formulario de creación de un cliente.
- Validar la creación de un cliente.
- Mostrar el formulario de modificación de un cliente.
- Validar la modificación de un cliente.
- Eliminar un cliente.

Sin la utilización de invocaciones dinámicas, este es un ejemplo del archivo de configuración para la administración de los clientes.

```
Código: 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="backoffice" namespace="/admin" extends="struts-
default">

        <action name="listado_Cliente" class="ejemplo.Cliente"
method="listado">
<result>/vistas/administradores/Cliente/listadoCliente.jsp</result>
        </action>

        <action name="editar_Cliente" class="ejemplo.Cliente"
method="editar">
<result>/vistas/administradores/Cliente/editarCliente.jsp</result>
        </action>
```

```
        <action name="agregarmodificar_Cliente"
class="ejemplo.Cliente" method="agregarmodificar">
            <result name="listadoCliente"
type="redirectAction">listado_Cliente</result>
            <result
name="input">/vistas/administradores/Cliente/editarCliente.jsp</result>
        </action>

        <action name="consultar_Cliente" class="ejemplo.Cliente"
method="consultar">
<result>/vistas/administradores/Cliente/consultarCliente.jsp</result>
        </action>

        <action name="Eliminar_Cliente" class="ejemplo.Cliente"
method="eliminar">
            <result name="listadoCliente"
type="redirectAction">listado_Cliente</result>
        </action>

</package>
</struts>
```

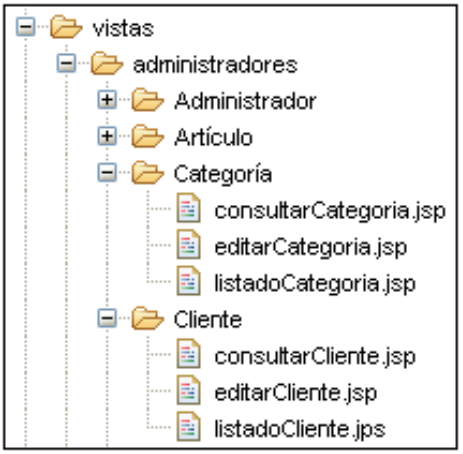
Por supuesto, se puede duplicar este código para, por ejemplo, la administración de los artículos.

```
...
<action name="listado_Articulo" class="ejemplo.Articulo"
method="listado">
  <result>/vistas/administradores/Cliente/listadoArticulo.jsp</result>
</action>

<action name="editar_Articulo" class="ejemplo.Articulo"
method="editar">
  <result>/vistas/administradores/Articulo/editarArticulo.jsp</result>
</action>
...
```

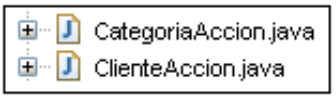
Enseguida podemos observar las repeticiones de declaraciones en el archivo de configuración y la complejidad del código a nivel de etiquetado. Con la utilización de las invocaciones dinámicas de métodos, podemos sustituir todo lo anterior por el siguiente código:

```
...
<action name="*_*" class="ejemplo.{2}Accion" method="{1}">
   <result name="listado{2}"
type="redirectAction">listado_{2}</result>
   <result name="input">/vistas/administradores/
{2}/editar{2}.jsp</result>
   <result>/vistas/administradores/{2}/{1}{2}.jsp</result>
</action>
...
```

Estas cinco líneas de declaraciones genéricas sustituyen a docenas de líneas de código. El funcionamiento será idéntico y totalmente operativo para administrar clientes, artículos o incluso categorías. Esta técnica es muy útil, pero requiere de un cierto rigor en la estructura del sitio.



Árbol genético



Clases de acción genéricas

En este ejemplo todo es dinámico, la llamada de clases con el parámetro {2}, la llamada de métodos de clases con el parámetro {1} y la invocación de páginas JSP con la combinación de dos parámetros.

El parámetro {0} contiene el URI recibido por la acción.

Invocación dinámica de métodos

Existe con Struts un carácter especial que puede utilizarse para la invocación dinámica de métodos. A este carácter *!* se le conoce como notación bang.

Por defecto, la invocación dinámica de métodos está habilitada en el archivo de propiedades*default.properties*. La declaración es la siguiente:

```
struts.enable.DynamicMethodInvocation = true
```

Esta declaración también puede especificarse en el archivo XML *struts.xml* con la etiqueta adaptada:

```
Código: 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="true" />
    ...
</struts>
```

Para poder utilizar la invocación dinámica de métodos es necesario declarar el parámetro value con valor true si esta etiqueta se utiliza en un archivo de configuración de la aplicación. No se recomienda utilizar la invocación dinámica de métodos en una aplicación profesional por razones evidentes de seguridad. De hecho, las URL pueden ser complejas y tener diferentes formatos, por lo que se dificulta en gran medida la realización de pruebas de verificación. Por último, el SEO efectuado por los principales motores de búsqueda en las URL por invocación dinámica es muy malo. Para desactivar este servicio, podemos utilizar nuestro archivo struts.xml y la siguiente declaración: `<constant name="struts.enable.DynamicMethodInvocation" value="false" />`

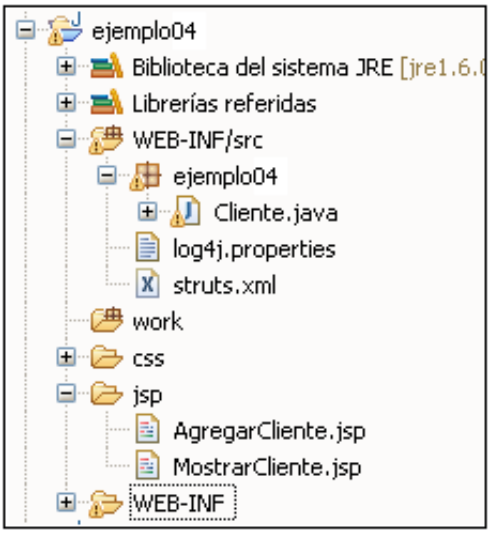
Podemos retomar el proyecto *ejemplo04* y utilizar la invocación dinámica de métodos. Para ello, se adapta el archivo *struts.xml* en consecuencia, así como los diferentes enlaces de la aplicación.

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

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="Gestion_Cliente" />

        <action name="Gestion_Cliente" class="ejemplo04.Cliente">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="agregar">/jsp/AgregarCliente.jsp</result>
            <result name="mostrar">/jsp/MostrarCliente.jsp</result>
        </action>

    </package>
</struts>
```

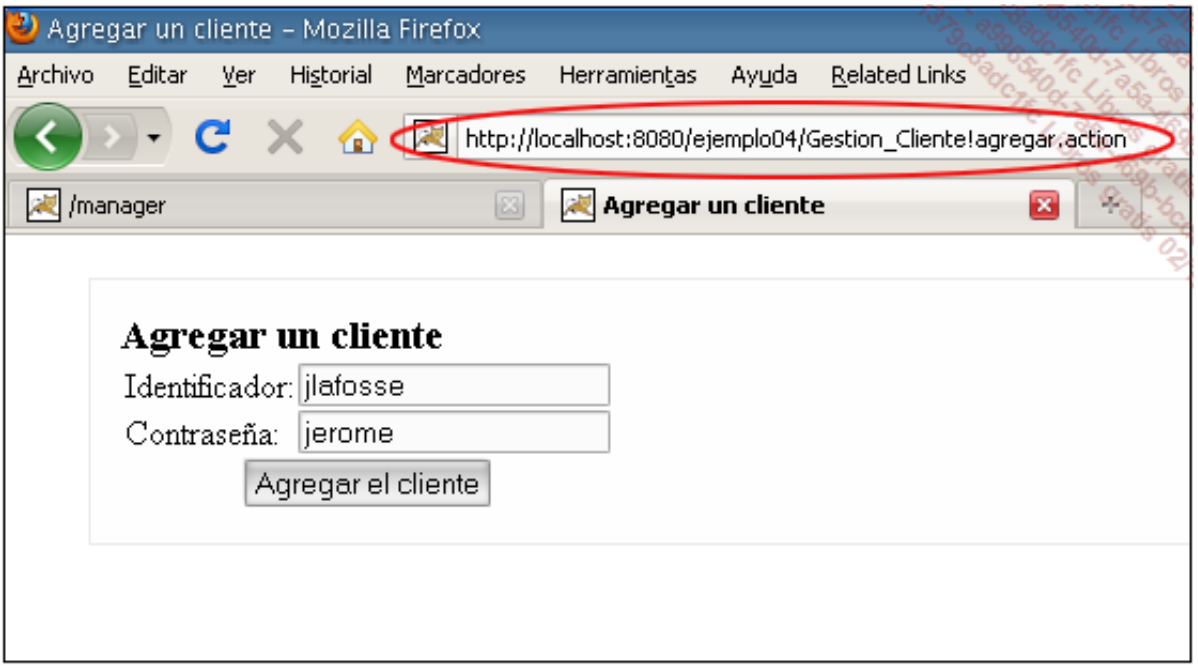
```
Código: ejemplo04.Cliente.java
package ejemplo04;

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

```
@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;

      public Cliente()
      {

      }
      ...

      // Mostrar el formulario de edición (agregar o modificar)
      public String agregar()
      {
             return "agregar";
      }

      // agregar la información del cliente en la sesión
      public String ConfirmarAgregar()
      {
              // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
              if(this.identificador.equals("") ||
this.contrasena.equals(""))
              {
                  return "input";
              }
              // ningún error
              else
              {
                   return "mostrar";
              }
      }
}
```

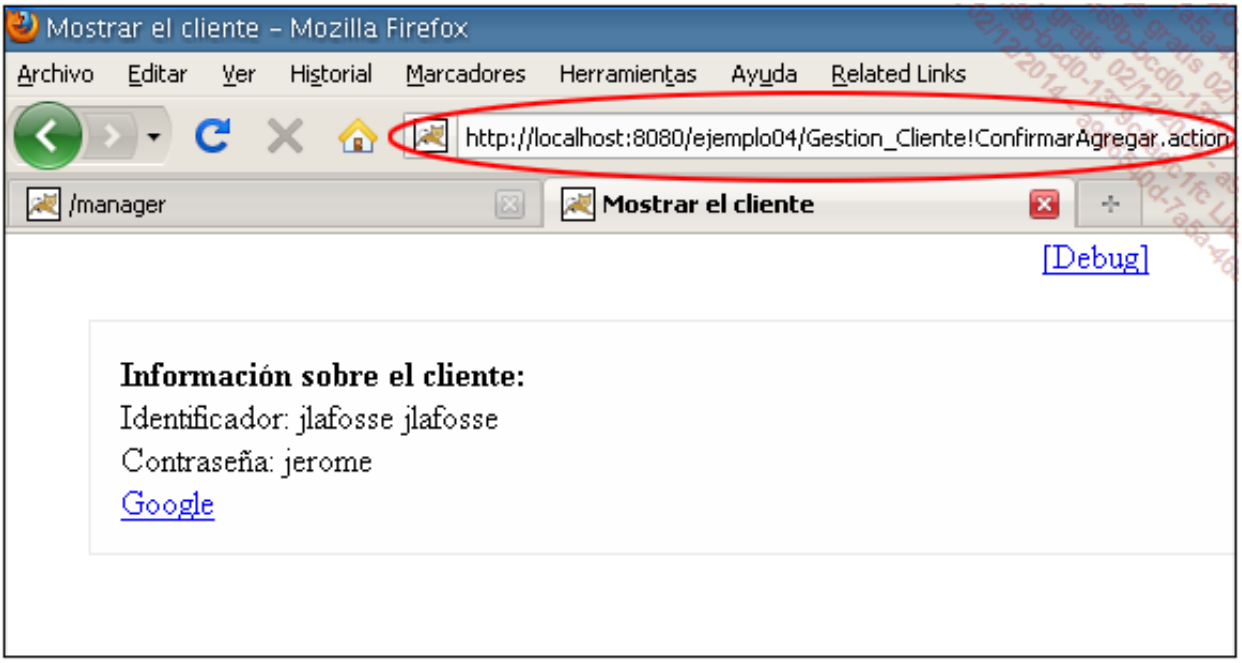
```
Código: /jsp/AgregarCliente.jsp
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <form method="post" action="Gestion_Cliente!
ConfirmarAgregar.action">
     ...
     </form>
</div>
</body>
</html>
```



Árbol de ejemplo04 con invocación dinámica



Formulario de registro con invocación dinámica



Visualización del formulario con invocación dinámica

Administración de los resultados

Las clases de acción de Struts devuelven cadenas de caracteres en relación con el procesamiento de la acción. Por ejemplo, si la acción contiene los resultados *success*, *input* y *error*, la declaración de la acción en el archivo *struts.xml* debe tener tres resultados con los valores de los atributos `name`siguientes:

```
<action name="miaccion" class="nombrepaquete.nombreclase">
    <result name="success">...</result>
    <result name="input">...</result>
    <param name="error">...</param>
</action>
```

El resultado se representa por medio de la etiqueta `<result/>` y de sus atributos, que son `name` y`type`. El atributo `name` permite dar un nombre al resultado que corresponda al valor *return* de la acción. El valor por defecto de este atributo es *success*. El atributo `type` permite definir el tipo de resultado. El valor por defecto de este atributo es *dispatcher* (redirección simple).

Por ejemplo, las siguientes declaraciones son idénticas:

```
<result name="success" type="dispatcher">/jsp/MostrarCliente.jsp
</result>
<result name="success">/jsp/MostrarCliente.jsp</result>
<result>/jsp/MostrarCliente.jsp</result>
```

El tipo *dispatcher*, correspondiente a una redirección a una página JSP en la mayoría de los casos, es el tipo más utilizado. La lista propuesta a continuación presenta todos los tipos de resultados posibles que podemos indicar en el atributo `type` de la etiqueta `<result/>`.

Además de estos distintos tipos de resultados, el desarrollador puede utilizar complementos con resultados específicos o incluso desarrollar sus propios complementos con resultados adaptados. En el capítulo Complementos de Struts abordaremos la creación de complementos con Struts.

- *dispatcher*: este tipo permite redirigir la acción a una página JSP de resultado.
- *redirect*: este tipo permite realizar una redirección completa a otra URL. La diferencia entre este tipo y *dispatcher* reside en la forma de redirección. Con *dispatcher* no se cambia la URL del navegador y se conservan los parámetros presentes en la consulta. Con *redirect* se cambia la URL en el navegador y se pierden los parámetros presentes en la consulta.
- *redirectAction*: este tipo permite realizar una redirección a otra acción.
- *chain*: este tipo permite utilizar el encadenamiento de acciones (varias acciones enlazadas).
- *freemarker*: este tipo permite utilizar el motor de plantillas FreeMarker.
- *velocity*: este tipo permite utilizar el motor de plantillas Velocity.
- *httpheader*: este tipo permite devolver encabezados HTTP específicos al navegador.
- *stream*: este tipo permite devolver un resultado en forma de flujo *InputStream* al navegador (imágenes, vídeos…).
- *xslt*: este tipo permite utilizar resultados XML y hojas de estilo XSLT.
- *plaintext*: este tipo permite devolver el resultado en formato de texto para visualizar el código fuente de una página.

1. Redirección con parámetros

El tipo *dispatcher* es el más utilizado en los resultados. Este tipo permitirá realizar una redirección a un recurso como una página JSP (o una página HTML/XHTML) sin perder los parámetros. Este parámetro debe apuntar a un recurso interno del proyecto, pero no permite realizar una redirección a un recurso externo o una URL absoluta (por ejemplo, www.gdawj.com).

```
<action name="Eliminar_Cliente" class="ejemplo03.Cliente"
method="eliminar">
   <result name="success"
type="dispatcher">/jsp/MostrarCliente.jsp</result>
</action>
```

2. Redirección sin parámetros

El tipo *redirect* permite realizar una redirección a un recurso interno o externo, pero con pérdida de parámetros. La principal ventaja de utilizar este tipo de resultado es que se puede redirigir al usuario a un recurso externo. Este tipo de redirección también es más rápida. Por último, este tipo de redirección se utiliza a menudo para administrar el doble envío (double submit). De hecho, con una redirección simple, cuando un usuario añade, por ejemplo, un producto a su compra, si por cualquier motivo actualiza su navegador, se agregará un segundo artículo. La redirección sin parámetros permite eliminar los parámetros de la consulta y dirigir al usuario a otra URL.

También podemos transmitir parámetros a una redirección utilizando las propiedades. La sintaxis*${propiedad}* permite transmitir los datos presentes a la clase de acción con sus descriptores de acceso.

```
<result name="success" type="redirect">/jsp/MostrarCliente.jsp?
identificador=${identificador}</result>
```

Si tomamos un nuevo proyecto llamado *ejemplo04*, realizaremos una primera redirección al motor de búsqueda Google y una segunda a una acción del sitio. Para realizar una redirección a un recurso externo es necesario utilizar el tipo *redirect*. El archivo de configuración de la aplicación *struts.xml* es el siguiente:

```
Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success"
type="redirect">/jsp/MostrarCliente.jsp?identificador=$
{identificador}</result>
        </action>

        <action name="Google">
```

```
            <result type="redirect">http://www.google.es</result>
        </action>
    </package>
</struts>
```

El archivo de la acción simple permite administrar únicamente respuestas.

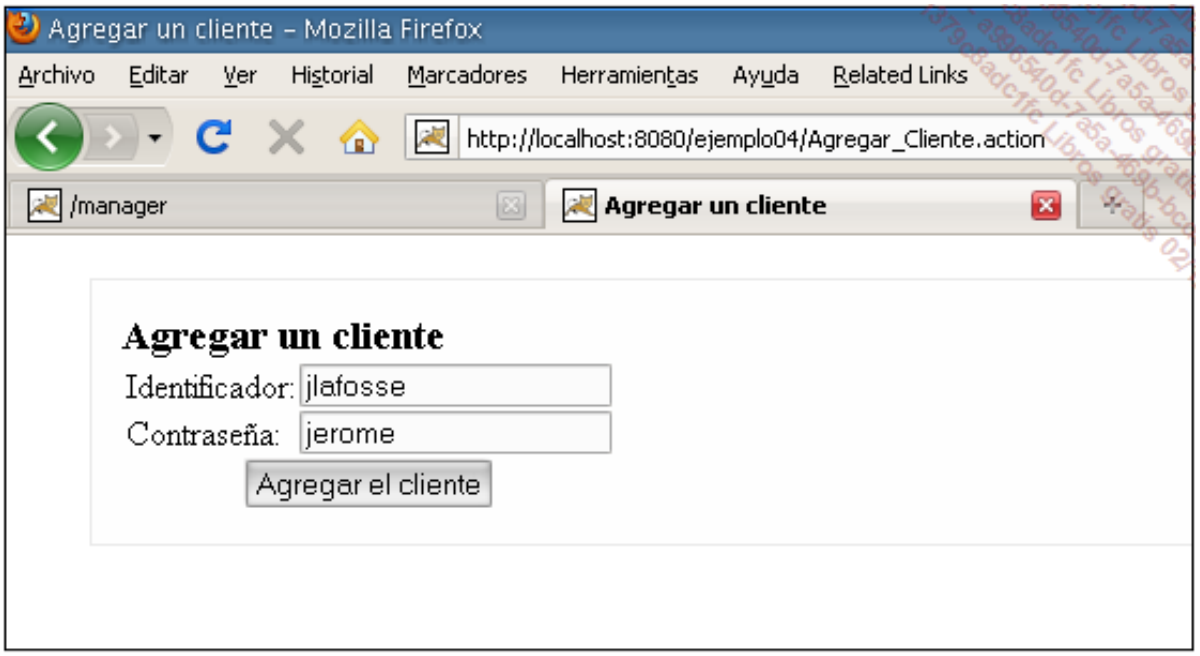
```
Código: ejemplo04.Cliente.java
package ejemplo04;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;

      public Cliente()
      {

      }

      //setter y getter

      // agregar la información del cliente en la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                return "input";
            }
            // ningún error
            else
            {
                return "success";
            }
      }
}
```

Por último, la página de visualización del cliente permite devolver los valores de los atributos. Podemos destacar que los valores se pierden, pero si se devuelve el parámetro *identificador*.

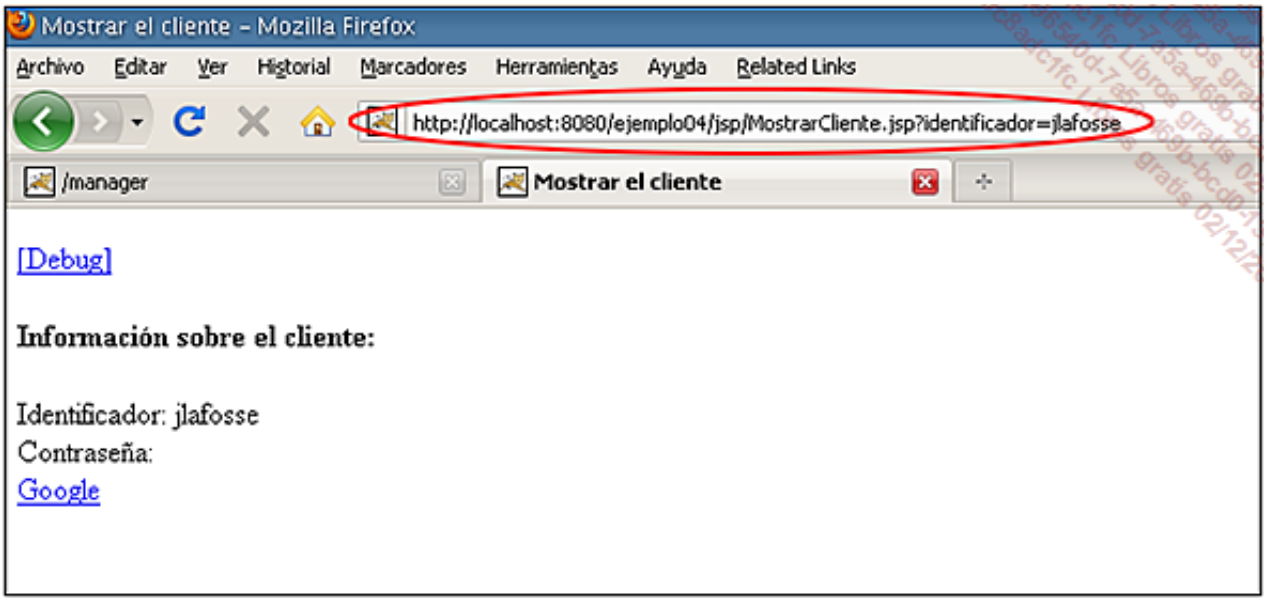
```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Mostrar el cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
      <h4>Informaci&oacute;n sobre el cliente:</h4>
      Identificador: <s:property value="identificador"/>
<%=request.getParameter("identificador") %><br/>
      Contrase&ntilde;a: <s:property value="contrasena"/><br/>
      <a href="../Google.action">Google</a>
      </p>
```

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

Es posible probar el proyecto *ejemplo04* en esta dirección: http://localhost:8080/ejemplo04/Agregar_Cliente.action



Formulario de registro de un nuevo cliente ejemplo04



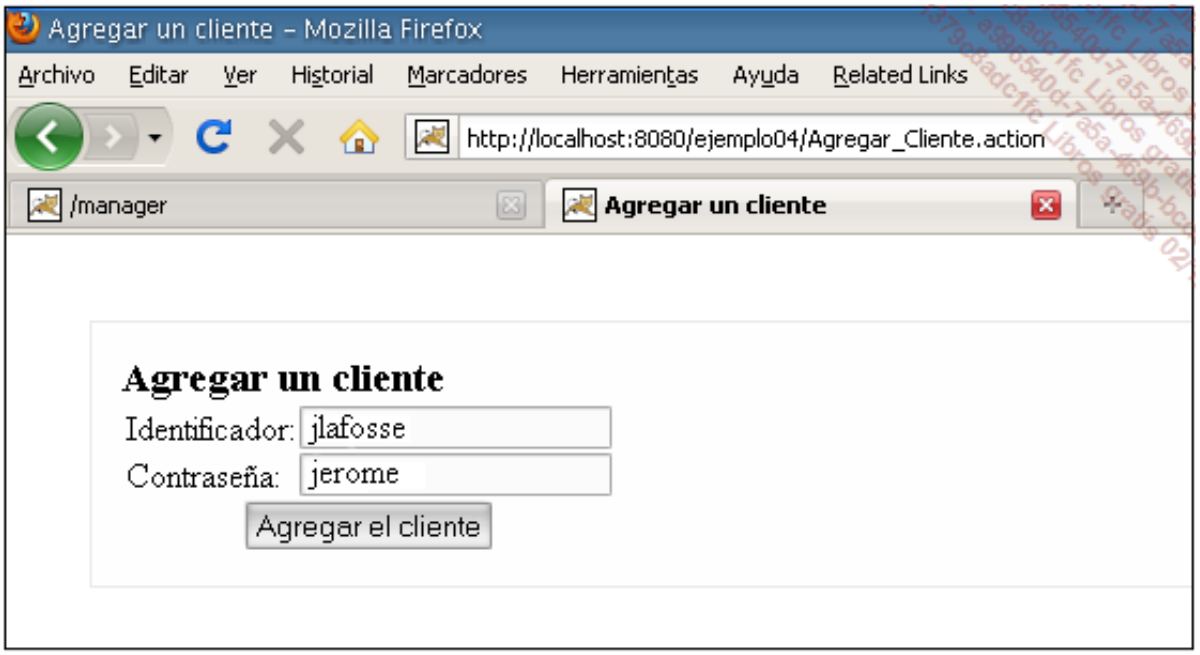
Visualización de la información del cliente ejemplo04

3. Redirección a una acción

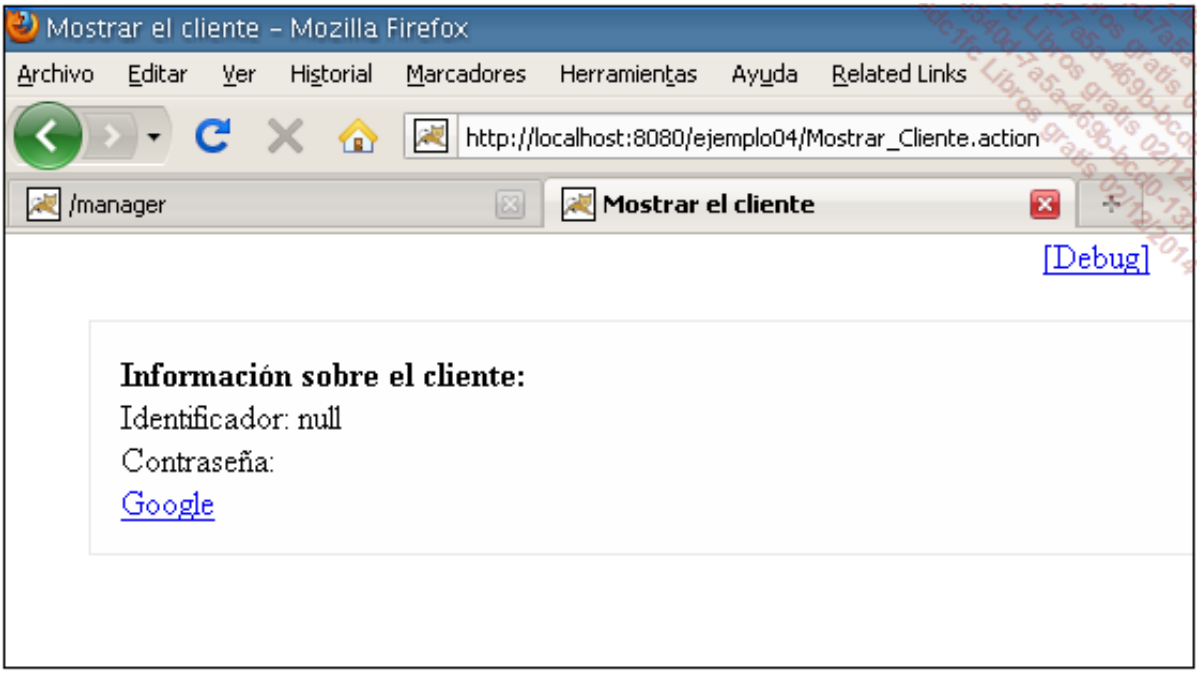
El tipo *redirectAction* se utiliza para realizar una redirección a otra acción de manera similar al tipo*redirect*. El nombre de la acción se determina en la etiqueta `<result/>` sin extensión (*.action*). La siguiente redirección permite redirigir el proyecto *ejemplo04* a otra acción con pérdida de parámetros.

La confirmación del registro del cliente redirige al usuario a la acción *Mostrar_Cliente*.

```
Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success"
type="redirectAction">Mostrar_Cliente</result>
        </action>

        <action name="Mostrar_Cliente" class="ejemplo04.Cliente">
            <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>
    </package>
</struts>
```



Formulario de registro de un cliente ejemplo04

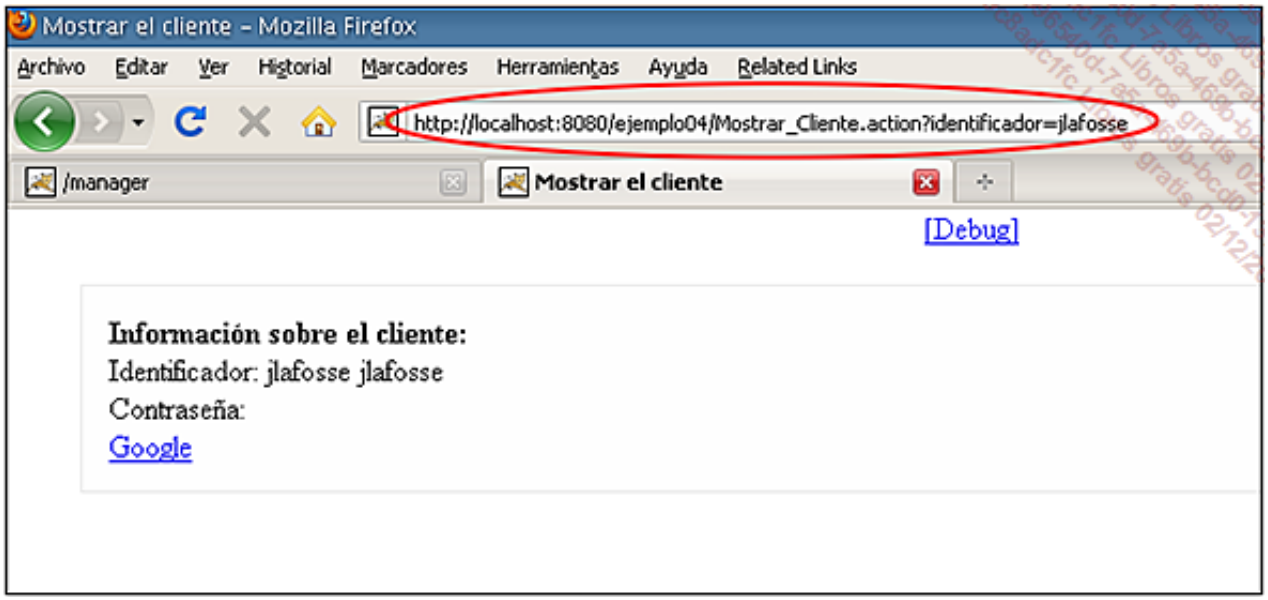


Visualización del cliente con redirección ejemplo04

No es necesario incluir la extensión de la acción (.action), el sufijo no es necesario. Esta técnica permite utilizar nombres de extensiones a su elección (.action, .do...).

También podemos realizar una redirección siguiendo el mismo principio anterior. Se agregarán los parámetros a la consulta y se transmitirán en la URL, lo cual reduce las posibilidades de escritura.

```
Código: 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>
...
        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success" type="redirectAction">
                <param name="actionName">Mostrar_Cliente</param>
                <param name="identificador">${identificador}</param>
            </result>
        </action>
...
</struts>
```



Redirección con parámetros

4. Redirección encadenada

El tipo *chain* permite realizar acciones encadenadas, es decir, una redirección a otra acción conservando el estado de la acción original en la acción de destino. También podemos encadenar acciones hacia otro paquete de Struts 2, especificando el nombre de este último. A continuación, retomaremos el ejemplo anterior y encadenar hemos la redirección de la confirmación de registro hacia la visualización. Si una acción se encadena con otra acción, la primera acción almacena la el valor de la pila de ejecución y lo transmitirá a la segunda acción y así sucesivamente.

Este tipo de redirección permite realizar un enrutamiento sin pérdida de parámetros. En el ejemplo anterior, el encadenado permite no perder la información introducida por el usuario.

Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-
default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success"
type="chain">Mostrar_Cliente</result>
        </action>
```

```
        <action name="Mostrar_Cliente" class="ejemplo04.Cliente">
            <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>

    </package>
</struts>
```

Visualización de los parámetros con redirección encadenada

En general, no es muy recomendable usar redirecciones encadenadas. Si una acción debe llamar a otra acción, esto se realiza con una redirección como respuesta a la primera acción o bien se puede llamar al código de la segunda acción en la primera.

---

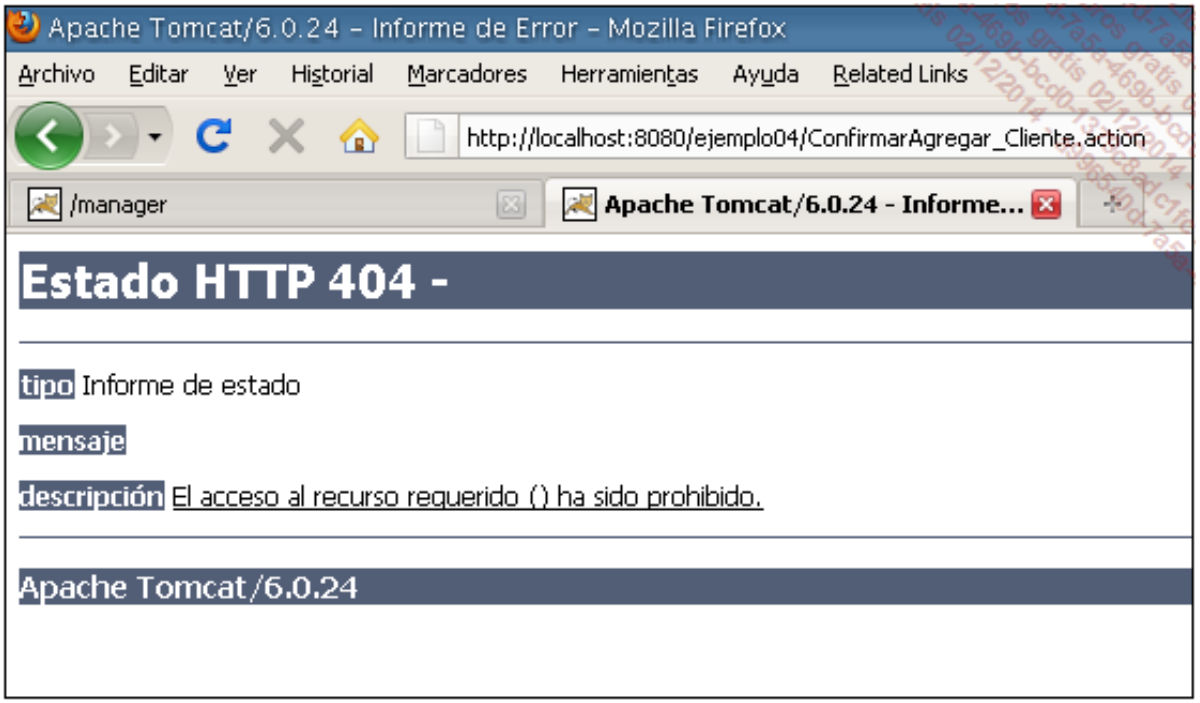
5. FreeMarker y Velocity

Estos tipos de resultados se utilizan para las presentaciones (templates) y las plantillas de visualización. El capítulo Los motores de plantillas explica en detalle estos dos tipos de resultados.

6. HttpHeader

Este tipo de resultados se utiliza para reenviar una respuesta HTTP al navegador. Los códigos de estado HTTP permiten al navegador tener información del lado del servidor. Por ejemplo, el código 404 indica al navegador que el recurso no se ha podido encontrar. El código 500 indica un error interno relativo al servidor.

Por ejemplo, podemos reenviar una respuesta 403 al navegador durante la autenticación para indicar que el acceso a este recurso está prohibido.



Tipos de respuestas HTTP

Gracias a esta técnica, Struts podrá administrar respuestas específicas. Si las páginas 403 se gestionan a través del descriptor de implementación web.xml, únicamente se mostrará esta página.

```
Código: /WEB-INF/web.xml
<error-page>
  <error-code>403</error-code>
  <location>/403.html</location>
</error-page>
```

7. Stream

Este tipo de resultado representado por una secuencia de bytes se utiliza para las imágenes, vídeos u otros. El capítulo Complementos de Struts dedicado al complemento JFreeChart explica en detalle este tipo de resultado.

8. XSLT

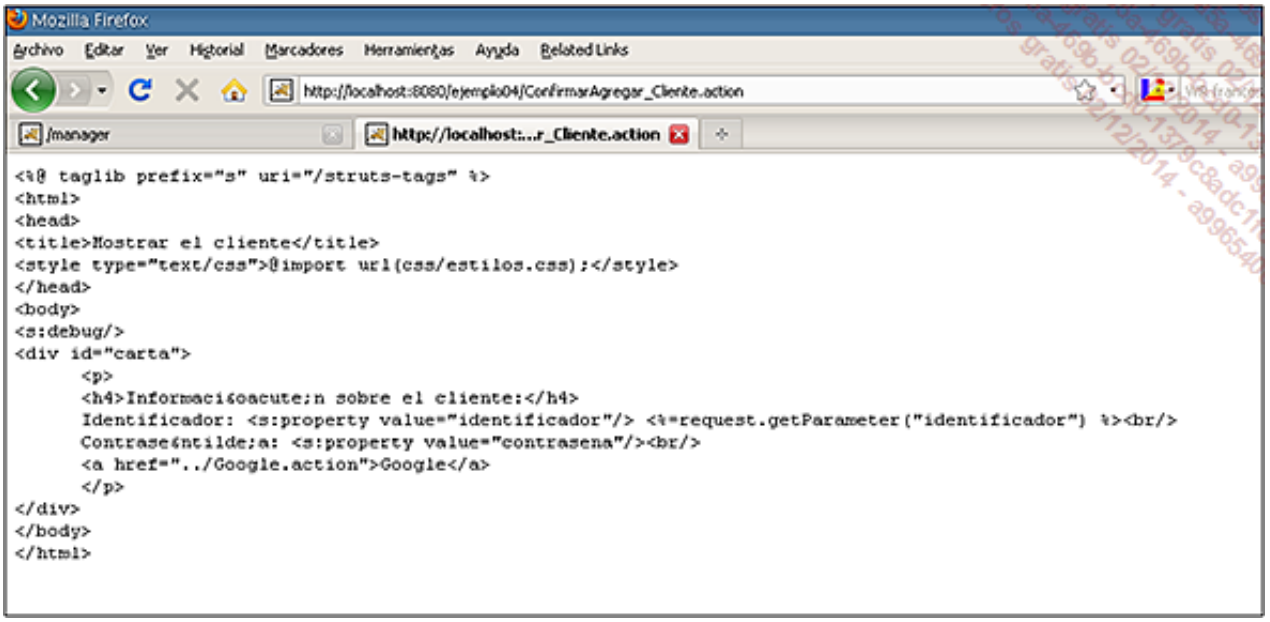
Este tipo de resultado representado por información en formato XML se explica en detalle en el capítulo XSLT.

9. PlainText

Este tipo permite devolver texto plano, es decir, el contenido textual de una página o de un archivo fuente. Podemos retomar nuestro proyecto *ejemplo04* y devolver tras la autenticación el código fuente de la página JSP.

```
Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success"
type="plainText">/jsp/MostrarCliente.jsp</result>
        </action>
    </package>
</struts>
```



Visualización en modo plainText de ejemplo04

Administración de las excepciones

La administración de las excepciones es una fase fundamental de un proyecto. Las pruebas permiten poner de relieve los fallos de los programas, pero la administración de las excepciones proporciona seguridad de procesamiento al código. Cada ejecución de excepción estará asociada a un error 500 HTTP del lado del servidor.

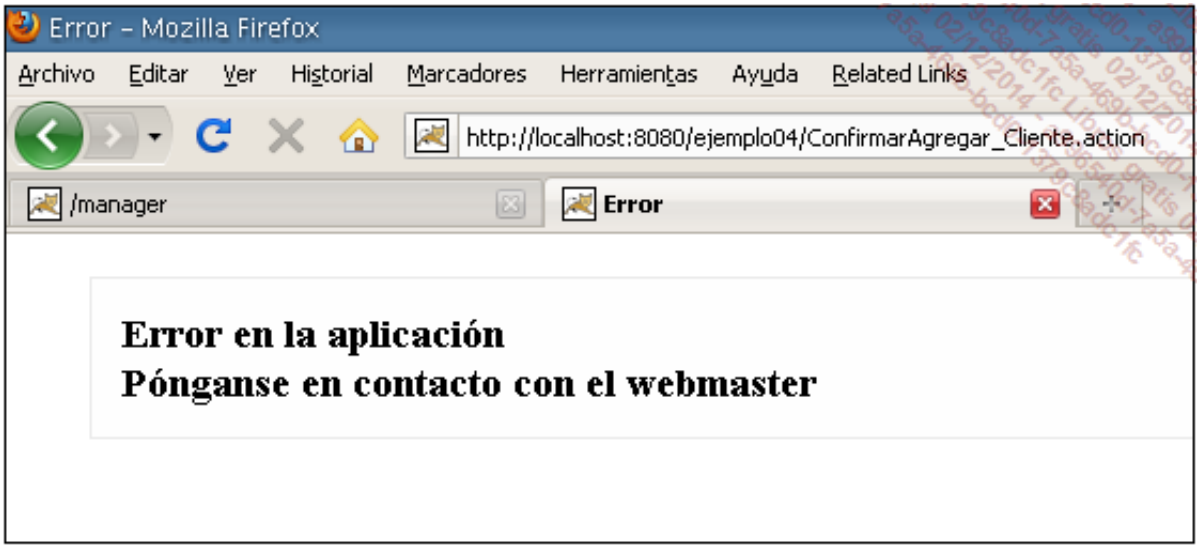
Para administrar las excepciones, Struts ofrece la etiqueta `<exception-mapping/>` que permite capturar las excepciones no procesadas en las acciones. Esta etiqueta contiene dos atributos,`exception`, para especificar el tipo de excepción que se va a capturar y `result`, que permite especificar el resultado que se ejecutará en caso de producirse una excepción. Esta acción de resultado puede especificarse de manera local en la acción o de manera global con la etiqueta `<globals-results/>`.

Si retomamos nuestro proyecto *ejemplo04*, podemos agregar voluntariamente el procesamiento*ConfirmarAgregar.action* de una conversión de cadena de caracteres en entero para ejecutar una excepción.

```
Código: ejemplo04.Cliente.java
package ejemplo04;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;

      public Cliente()
      {

      }

...
      // agregar la información del cliente en la sesión
      public String agregar()
      {
            // forzar la ejecución de una excepción
            int exception=Integer.parseInt(this.contrasena);

            // verificar los datos introducidos, en caso de error
devolver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                return "input";
            }
            // ningún error
            else
            {
                return "success";
            }
      }
}
```

```
Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <exception-mapping result="error"
exception="java.lang.Exception"/>
            <result name="error">/jsp/Error.jsp</result>
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>
    </package>
</struts>
```



Administración de las excepciones

```
Código: /jsp/Error.jsp
<html>
<head>
<title>Error</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Error en la aplicaci&oacute;n<br/>P&oacute;ngase en
contacto con el webmaster</h3>
</div>
</body>
</html>
```

La etiqueta `<global-exception-mappings/>` permite administrar los grupos de excepciones. Por el contrario, una excepción declarada en una etiqueta `<global-exception-mappings/>` debe hacer referencia a un resultado declarado en una etiqueta `<global-results/>`. Podemos adaptar nuestro ejemplo anterior con este tipo de declaración de excepción.

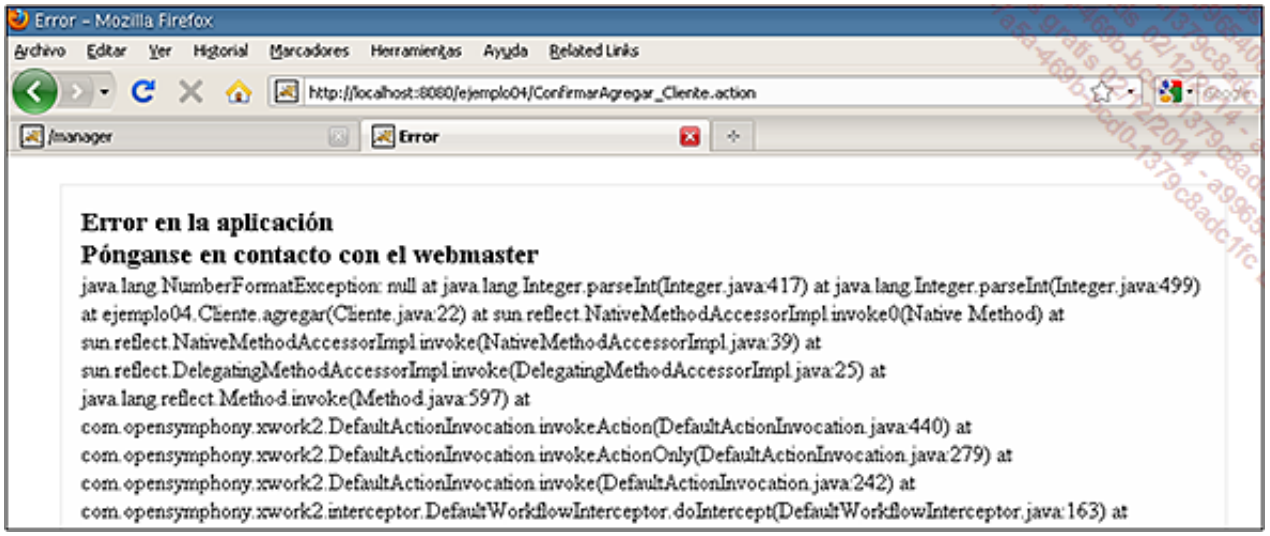
La etiqueta de Struts `<s:property/>` permite visualizar la información sobre las excepciones en las vistas JSP. Esta etiqueta utiliza el atributo `exception.message` para utilizar el mensaje asociado a las excepciones en el archivo de propiedades y el atributo `exceptionStack` para mostrar los registros de la pila de excepción de forma detallada.

Código: 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="true" />

    <package name="ejemplo04" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <global-results>
            <result name="error">/jsp/Error.jsp</result>
        </global-results>
        <global-exception-mappings>
            <exception-mapping result="error"
exception="java.lang.Exception"/>
        </global-exception-mappings>

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo04.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success">/jsp/MostrarCliente.jsp</result>
        </action>
    </package>
</struts>
```

Código: /jsp/Error.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Error</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Error en la aplicaci&oacute;n<br/> P&oacute;nganse en
contacto con el webmaster</h3>
      <s:property value="exceptionStack"/>
      <s:property value="exception.message"/>
</div>
</body>
</html>
```



Los atributos exceptionStack y exception.message

En resumen

Este capítulo ha presentado la administración de acciones de Struts. En primer lugar, se han detallado las clases de acción y sus resultados asociados. La segunda parte del capítulo explica las distintas técnicas de acceso a los recursos y el procesamiento de los parámetros. Los siguientes párrafos se han centrado en la administración del mapping y la invocación dinámica de métodos. El penúltimo párrafo presenta en forma de ejemplos los distintos tipos de resultados con Struts y la última parte se centra en el procesamiento de las excepciones.

Presentación

El framework Struts 2 incluye una librería de etiquetas, para las páginas JSP denominadas vistas. La biblioteca de etiquetas de Struts se compone de una primera categoría para la gestión de datos y una segunda para las propiedades, parámetros y estructuras de controles (condicionales, bucles...).

Dichas etiquetas permiten programar servicios dinámicos del lado del cliente sin utilizar código Java. De este modo, podemos rellenar campos de formularios, conservar datos o mostrar objetos fácilmente.

Las etiquetas de formulario

Estas etiquetas XHTML se utilizan en las páginas JSP para mostrar datos. La biblioteca de etiquetas de Struts se facilita con el archivo *struts2-core-x.jar*. Para utilizar esta librería en nuestras páginas JSP, debemos incluir la siguiente declaración en la parte superior de cada página y añadir el prefijo`<s:nombreetiqueta/>` a nuestras etiquetas:

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

Las etiquetas de Struts pueden mostrar contenido estático o contenido dinámico con la ayuda de una expresión OGNL (*Object-Graph Navigation Language*). Las etiquetas de Struts se declaran en el paquete *org.apache.struts2.components* y se basan en declaraciones JSTL corrientes. En función del tipo de atributo utilizado (visualización, lectura, presentación, etc.), existen diversos parámetros, como la clase de CSS utilizada para la presentación, el nombre, el título o las acciones JavaScript asociadas.

1. La etiqueta <s:form/>

Esta etiqueta permite mostrar un formulario HTML en el documento. Existen parámetros habituales disponibles como la acción que se va a ejecutar, el método HTTP o el tipo de codificación utilizado. El proyecto *ejemplo05*, basado en la aplicación anterior *ejemplo04*, utiliza la etiqueta `<s:form/>` en la página */jsp/AgregarCliente.jsp*.

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador"
label="Identificador" />
            <s:textfield name="contrasena" label="Contraseña"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```

Las etiquetas de Struts cuentan con numerosos parámetros utilizados habitualmente para la presentación, los estilos CSS que se van a aplicar o las acciones JavaScript asociadas. La etiqueta`<s:form/>` es traducida automáticamente por el motor y transformada en tabla. Struts realiza automáticamente la presentación. La alineación y el estilo son gestionados por los atributos de cada etiqueta. Podemos mostrar el código fuente de esta página con el fin de observar el código XHTML generado por Java.

Código fuente generado: /jsp/AgregarCliente.jsp

```
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
<form id="ConfirmarAgregar_Cliente" name="ConfirmarAgregar_Cliente"
action="/ejemplo05/ConfirmarAgregar_Cliente.action" method="post">
<table class="wwFormTable">
<tr>
    <td class="tdLabel"><label
for="ConfirmarAgregar_Cliente_identificador"
class="label">Identificador:</label></td>
    <td><input type="text" name="identificador" value=""
id="ConfirmarAgregar_Cliente_identificador"/></td>
</tr>
<tr>
    <td class="tdLabel"><label
for="ConfirmarAgregar_Cliente_contrasena" class="label">
Contrasena:</label></td>
    <td><input type="text" name="contrasena" value=""
id="ConfirmarAgregar_Cliente_contrasena"/></td>
</tr>
<tr>
    <td colspan="2"><div align="right"><input type="submit"
id="ConfirmarAgregar_Cliente_0" value="Agregar el cliente"/>
</div>
</td>
</tr>
</table>
</form>
</div>
</body>
</html>
```

2. Los temas de presentación y el atributo theme

Como hemos indicado anteriormente, por defecto Struts utiliza un tema de presentación que permite generar una tabla XHTML para mostrar el contenido. Resumiendo, la etiqueta *<s:form/>* se transforma en *<form…><table…/></form>*. Esta práctica presentación permite desarrollos rápidos y adaptados a las partes administrativas de los sitios Web pero no obligatoriamente a las partes públicas. En realidad, existe tendencia a utilizar presentaciones optimizadas con etiquetas `<span/>` y`<div/>`, así como a aplicar hojas de estilo de forma masiva.

Con Struts, las presentaciones, denominadas también plantillas, se generan con FreeMarker, que analizaremos en el capítulo reservado a esta herramienta. Los temas propuestos por FreeMarker son los siguientes:

- *xhtml*: es el tema predeterminado. Este modelo permite mostrarlos formularios en forma de tabla XHTML.
- *simple*: este tema permite traducir todas las etiquetas de Struts de la forma más simple, es decir, sin formato para la visualización. Con este tema, las etiquetas se traducirán sin modificación.
- *css_xhmtl*: este tema permite traducir las etiquetas con un modelo adaptado a las hojas de estilo CSS, es decir, con las etiquetas `<span/>` y `<div/>`.
- *ajax*: este tema permite facilitar un modelo para la utilización de Ajax.

Los temas pueden utilizarse en el nivel más alto de la jerarquía, en la etiqueta `<s:form/>`, o a nivel local en cada etiqueta XHTML (por ejemplo: `<s:textfield/>`). Por defecto se utiliza el tema *xhtml*, por lo que no es necesario especificarlo. Si deseamos cambiar el tipo, podemos utilizar el atributo`theme` de cada etiqueta de presentación.

Utilizaremos de nuevo nuestro *ejemplo05* y la página JSP *AgregarCliente.jsp* para modificar el tema.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente"
theme="simple">
            <s:textfield name="identificador"
label="Identificador" />
            <s:textfield name="contrasena" label="Contraseña"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```

Formulario de registro de un cliente con el tema simple

Formulario de adición de un cliente con el tema css_xhtml

Por supuesto, podemos definir nuestros propios temas con FreeMarker. También podemos utilizar un tema para toda la página haciendo uso de un atributo denominado `theme` con el alcance *page*,*request*, *session* o *application*. Además, es posible asignar un tema a todo el proyecto con la propiedad *struts.ui.theme* presente en el archivo de propiedades *struts.properties*.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente"
theme="css_xhtml">
            <s:textfield name="identificador"
label="Identificador" />
            <s:textfield name="contrasena" label="Contraseña"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```

```
Código fuente generado: /jsp/AgregarCliente.jsp
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
<h3>Agregar un cliente</h3>
<form id="ConfirmarAgregar_Cliente" name="ConfirmarAgregar_Cliente"
action="/ejemplo05/ConfirmarAgregar_Cliente.action" method="post">
<div id="wwgrp_ConfirmarAgregar_Cliente_identificador" class="wwgrp">
<div id="wwlbl_ConfirmarAgregar_Cliente_identificador" class="wwlbl">
<label for="ConfirmarAgregar_Cliente_identificador" class="label">
Identificador:
</label></div> <br /><div
id="wwctrl_ConfirmarAgregar_Cliente_identificador" class="wwctrl">
<input type="text" name="identificador" value=""
id="ConfirmarAgregar_Cliente_identificador"/></div> </div>
<div id="wwgrp_ConfirmarAgregar_Cliente_contrasena" class="wwgrp">
<div id="wwlbl_ConfirmarAgregar_Cliente_contrasena" class="wwlbl">
<label for="ConfirmarAgregar_Cliente_contrasena" class="label">
Contrasena:
</label></div> <br /><div
id="wwctrl_ConfirmarAgregar_Cliente_contrasena" class="wwctrl">
<input type="text" name="contrasena" value=""
id="ConfirmarAgregar_Cliente_contrasena"/></div> </div>
<div align="right" id="wwctrl_ConfirmarAgregar_Cliente_0"><input
type="submit" id="ConfirmarAgregar_Cliente_0" value="Agregar el
cliente"/>
</div>
</form>
</div>
</body>
</html>
```

3. La etiqueta <s:textfield/>

La etiqueta `<s:textfield/>` permite crear un campo de texto en la página JSP. Encontraremos los atributos HTML habituales con `maxlength`, `readonly` y `size`.

4. La etiqueta <s:password/>

La etiqueta `<s:password/>` permite crear un campo de tipo contraseña en la página JSP. Encontraremos los atributos HTML habituales con `maxlength`, `readonly` y `size`.

5. La etiqueta <s:submit/>

La etiqueta `<s:submit/>` permite crear un botón de validación del formulario. Si utilizamos de nuevo nuestra página JSP *AgregarCliente.jsp*, podemos observar el uso de las etiquetas `<s:textfield/>`,`<s:password/>` y `<s:submit/>`.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
```

Utilización de la etiqueta `<s:submit/>`

```
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" label="Identificador"
labelposition="top" cssClass="input"
            cssErrorClass="inputerror" tooltip="Introduzca su
identificador"
tooltipConfig="#{'tooltipDelay':'500','tooltipIcon':'/images/ayuda.
jpg'}"/>
            <s:password name="contrasena" label="Contraseña"
labelposition="top"/>
            <s:submit value="Agregar el cliente"/>
       </s:form>
</div>
</body>
</html>
```

6. La etiqueta <s:reset/>

La etiqueta `<s:reset/>` permite validar los datos del formulario. Esta etiqueta recoge las propiedades de las etiquetas anteriores.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" label="Identificador"
labelposition="top" cssClass="input"/>
            <s:password name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar el cliente"/>
            <s:reset/>
      </s:form>
</div>
</body>
</html>
```

7. La etiqueta <s:label/>

La etiqueta `<s:label/>` permite crear una etiqueta HTML con el fin de asignar un vínculo HTML que permita posicionar el cursor en un campo determinado.

8. La etiqueta <s:head/>

La etiqueta `<s:head/>` permite crear una etiqueta HTML de encabezado para el contenido de los archivos asociados, por ejemplo archivos JavaScript o Ajax. Esta etiqueta no se utiliza con frecuencia.

9. La etiqueta <s:textarea/>

La etiqueta `<s:textarea/>` permite crear una etiqueta HTML de tipo campo de texto de varias líneas. Contiene principalmente los atributos importantes siguientes `cols` y `rows` para precisar respectivamente el número de columnas y filas.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textarea name="contrasena" label="Contraseña"
labelposition="top" cols="18" rows="5"/>
            <s:submit value="Agregar el cliente"/>
            <s:reset/>
      </s:form>
</div>
</body>
</html>
```

Utilización de la etiqueta `<s:textarea/>`

10. La etiqueta <s:checkbox/>

La etiqueta `<s:checkbox/>` permite crear una casilla de verificación HTML Esta etiqueta de Struts permite gestionar de manera dinámica el estado de la casilla (seleccionada o no). La respuesta enviada a la clase de acción es *false* si la casilla no está seleccionada, y *true* en caso de estarlo. Las etiquetas checkbox van acompañadas de una etiqueta HTML `hidden`.

Si la etiqueta no está seleccionada, se presenta la etiqueta sin el parámetro *checkbox*. Todos los atributos deben contener sus descriptores de acceso en la clase de acción para poder funcionar.

```
Código: ejemplo05.Cliente.java
package ejemplo05;
```

También es posible utilizar colecciones Java para gestionar listas de casillas de verificación. Así, podemos crear una lista con los objetos que se mostrarán en cada casilla de verificación con el texto y el estado asociado.

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

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private boolean Boletin;

      public Cliente()
      {

      }

      // getter y setter

      public boolean isBoletin() {
             return Boletin;
      }

      public void setBoletin(boolean Boletin) {
             this.Boletin = Boletin; }

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
             // verificar los datos introducidos, en caso de
error volver a la página de introducción de datos
             if(this.identificador.equals("") ||
this.contrasena.equals(""))
              {
                 return "input";
              }
              // sin errores
              else
              {
                  return "mostrar";
              }
      }
}
```

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
<h3>Agregar un cliente</h3>
<s:form method="post" action="ConfirmarAgregar_Cliente">
<s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
<s:checkbox name="Boletin"  label="Inscripción al Boletin"/>
<s:submit value="Agregar el cliente"/>
<s:reset/>
</s:form>
</div>
</body>
</html>
```

11. La etiqueta <s:select/>

La etiqueta `<s:select/>` permite crear una lista desplegable HTML Este tipo de componente es bastante difícil de gestionar en la programación clásica cuando se realiza el posicionamiento del ítem adecuado en la lista. El atributo utilizado por Struts para gestionar los valores transmitidos se denomina `list`. Este atributo se utiliza igualmente para los botones radio, que también representan las opciones.

El tipo de este atributo `list` puede ser *String*, una enumeración *java.util.Enumeration*, un elemento de iteración *java.util.Iterator*, una colección *java.util.Map* o HashMap, ArrayList... Podemos utilizar el ejemplo anterior con una colección.

```
Código: ejemplo05.Cliente.java
package ejemplo05;

import java.util.HashMap;
import java.util.Map;
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private boolean Boletin;
    private Map<Integer,String> listaProfesiones=new
HashMap<Integer,String>();
    private int profesion;


      public Cliente()
      {

      }

      public int getProfesion() {
             return profesion;
      }

      public void setProfesion(int profesion) {
             this.profesion = profesion;
```

La etiqueta `<s:select/>` contiene el atributo `label` que corresponde al texto que se mostrará en la página HTML. El atributo `list` permite crear el vínculo con la información que se debe mostrar en la lista. Por último, el atributo `name` permite nombrar la lista desplegable. El atributo `list` está asociado con los descriptores de acceso presentes en la clase `Client` permitiendo crear el vínculo con los valores automáticamente. El atributo `name` está asociado con los descriptores de acceso de todo el tipo presentes en la clase `Client`, permitiendo crear el vínculo con la elección del cliente.

De esta forma, ante la introducción de datos o errores en los formularios, la visualización se posicionará automáticamente en el ítem adecuado de la lista desplegable.

Vamos a utilizar un segundo ejemplo con la clase `Profesion` asociada al cliente y una lista desplegable denominada *profesion* asociada a una colección *listaProfesiones* de tipo *Lista*. La dirección para acceder a la aplicación es la siguiente: http://localhost:8080/ejemplo05/Agregar_Cliente.action

Utilización de la etiqueta `<s:select/>`

```
        }

      public Map<Integer, String> getListaProfesiones() {
             this.listaProfesiones.put(1, "Informático");
             this.listaProfesiones.put(2, "Formador");
             return listaProfesiones;
        }

      public void setListaProfesiones(Map<Integer, String>
listaProfesiones) {
             this.listaProfesiones = listaProfesiones; }

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
             // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
             if(this.identificador.equals("") ||
this.contrasena.equals(""))
             {
                  return "input";
             }
             // sin errores
             else
             {
                  return "mostrar";
             }
      }
}
```

Código: 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="true" />

    <package name="ejemplo05" namespace="/" extends="struts-default">
        <default-action-ref name="Agregar_Cliente" />

        <action name="Agregar_Cliente" class="ejemplo05.Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo05.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="success"
type="chain">Mostrar_Cliente</result>
        </action>
    </package>
</struts>
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
<h3>Agregar un cliente</h3>
<s:form method="post" action="ConfirmarAgregar_Cliente">
<s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
<s:checkbox name="Boletin"  label="Inscripción al Boletin"/>
<s:select name="profesion" label="Profesión"
list="listaProfesiones"/>
<s:submit value="Agregar el cliente"/>
</s:form>
</div>
</body>
</html>
```

Código: ejemplo05.Cliente.java

```
package ejemplo05;

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

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private int profesion;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();

      public Cliente()
      {
```

```
	}

	public List<Profesion> getListaProfesiones() {
		listaProfesiones.add(new Profesion(1,
"Informático"));
		listaProfesiones.add(new Profesion(2,
"Formador"));
		listaProfesiones.add(new Profesion(3, "SGBDM"));
		listaProfesiones.add(new Profesion(4, "Responsable
de red"));
		return listaProfesiones;
	}

	public void setListaProfesiones(List<Profesion>
listaProfesiones) {
		this.listaProfesiones = listaProfesiones;
	}

	public int getProfesion() {
		return profesion;
	}

	public void setProfesion(int Profesion) {
		this.profesion = profesion;
	}

	// getter y setter

	// agregar los datos del cliente a la sesión
	public String agregar()
	{
		// verificar los datos, en caso de error
volver a la página de introducción de datos
		if(this.identificador.equals("") ||
this.contrasena.equals(""))
		{
		    return "input";
		}
		// sin errores
		else
		{
		     return "mostrar";
		}
	}
}


// Clase de gestión de las profesiones
class Profesion
{
    private int idProfesion;
    private String nombre;
    public Profesion(int idProfesion, String nombre)
    {
      this.idProfesion=idProfesion;
      this.nombre=nombre;
    }

      public int getIdProfesion() {
		return idProfesion;
	}

	public void setIdProfesion(int idProfesion) {
		this.idProfesion=idProfesion;
	}

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

```
Clase: /vistas/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
	<h3>Agregar un cliente</h3>
	<s:form method="post" action="ConfirmarAgregar_Cliente">
		<s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
		<s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
		<s:select name="profesion" label="Profesión"
labelposition="top" list="listaProfesiones" listKey="idProfesion"
listValue="nombre"/>
		<s:submit value="Agregar el cliente"/>
	</s:form>
</div>
</body>
</html>
```

12. La etiqueta <s:optgroup/>

La etiqueta `<s:optgroup/>` permite, junto con la etiqueta `<s:select/>`, crear una lista desplegable con grupos de datos en formato HTML. Este tipo de componente, al igual que las listas simples, es bastante difícil de gestionar en la programación clásica con el fin de recolocar la lista en el ítem adecuado cuando se realiza una

selección. Vamos a retomar el ejemplo anterior y a agregarle una etiqueta `<s:optgroup/>` para gestionar el nivel de cualificación asociado a la profesion.

Código: ejemplo05.Cliente.java

```
package ejemplo05;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private int profesion;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();
    private Map<Integer,String> nivelProfesion=new
HashMap<Integer,String>();

      public Cliente()
      {

      }

      // getter y setter

      public Map<Integer, String> getNivelProfesion() {
             nivelProfesion.put(1, "Sec.");
             nivelProfesion.put(2, "Diplom.");
             nivelProfesion.put(3, "Lic.");
             return nivelProfesion;
      }

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
             // verificar los datos, en caso de error
volver a la página de introducción de datos
             if(this.identificador.equals("") ||
this.contrasena.equals(""))
             {
                 return "input";
             }
             // sin errores
             else
             {
                 return "mostrar";
             }
      }
}


// Clase de gestión de las profesiones
class Profesion
{
    private int idProfesion;
    private String nombre;

      public Profesion(int idProfesion, String nombre)
      {
        this.idProfesion=idProfesion;
        this.nombre=nombre;
      }

      // getter y setter

}
```

Utilización de la etiqueta `<s:optgroup/>`

La gestión de la etiqueta `<s:optgroup/>` no requiere la utilización del setter asociado.

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:select name="profesion" label="Profesión"
labelposition="top" list="listaProfesiones" listKey="idProfesion"
listValue="nombre"/>
                <s:optgroup label="Nivel"
list="nivelProfesion"/>
            </s:select>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```

13. La etiqueta <s:radio/>

La etiqueta `<s:radio/>` permite utilizar botones radio HTML. Los botones radio con Struts recogen el principio de uso de las listas. Cada botón radio se representa en una lista y la elección única es gestionada por el framework con el fin de aligerar el trabajo del desarrollador. El principio de utilización es idéntico al de las listas,

es necesario hacer uso de los descriptores de acceso tanto para la lista como para la elección del botón.

Vamos a agregar una lista de botones radio para el tipo de contrato del cliente.

Utilización de la etiqueta `<s:radio/>`

Código: ejemplo05.Cliente.java

```
package ejemplo05;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private int profesion;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();
    private Map<Integer,String> nivelProfesion=new
HashMap<Integer,String>();
    private SortedMap<Integer,String> listaContratos=new
TreeMap<Integer,String>();
    private int contrato;

      public Cliente()
      {

      }

// getter y setter

      public Map<Integer, String> getNivelProfesion() {
             nivelProfesion.put(1, "Sec.");
             nivelProfesion.put(2, "Diplom.");
             nivelProfesion.put(3, "Lic.");
             return nivelProfesion;
      }

      public SortedMap<Integer, String> getListaContratos() {
             listaContratos.put(1, "Tiempo parcial");
             listaContratos.put(2, "Temporal");
             listaContratos.put(3, "Jornada completa");
             return listaContratos;
      }

      public void setListaContratos(SortedMap<Integer, String>
listaContratos) {
             this.listaContratos = listaContratos;
      }

      public int getContrato() {
             return contrato;
      }

      public void setContrato(int contrato) {
             this.contrato = contrato;
      }

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
             // verificar los datos, en caso de error
volver a la página de introducción de datos
             if(this.identificador.equals("") ||
this.contrasena.equals(""))
             {
                 return "input";
             }
             // sin errores
             else
             {
                 return "mostrar";
             }
      }
}


// Clase de gestión de las profesiones
class Profesion
{
    private int idProfesion;
    private String nombre;
    public Profesion(int idProfesion, String nombre)
    {
      this.idProfesion=idProfesion;
      this.nombre=nombre;
    }

    // getter y setter
}
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
```

```
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:select name="profesion" label="Profesión"
labelposition="top" list="listaProfesiones" listKey="idProfesion"
listValue="nombre"/>
                <s:optgroup label="Nivel" list="nivelProfesion"/>
            </s:select>
            <s:radio name="contrato" label="Tipo de contrato"
list="listaContratos"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```

14. La etiqueta <s:checkboxlist/>

La etiqueta `<s:checkboxlist/>` permite crear una casilla de verificación HTML. Estos componentes utilizan tablas de cadenas de caracteres o una colección de tipos primitivos. Cuando se valida una casilla de verificación, la propiedad asociada en la clase de acción se inicializa, por lo que es necesario implementar una tabla de números enteros para gestionar la elección del usuario.

Vamos a continuar nuestro ejemplo con casillas de verificación para la gestión de las comidas del cliente.

Utilización de la etiqueta `<s:checkboxlist/>`

```
Código: ejemplo05.Cliente.java
package ejemplo05;

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

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private int profesion;
    private int[] comida;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();
    private List<Comida> listaComida=new ArrayList<Comida>();

      public Cliente()
      {

      }

      // getter y setter

      public List<Comida> getListaComida() {
            listaComida.add(new Comida(1, "Comida"));
            listaComida.add(new Comida(2, "Cena"));
            return listaComida;
      }

      public void setListaComida(List<Comida> listaComida) {
            this.listaComida = listaComida;
      }

      public int[] getComida() {
            return comida;
      }

      public void setComida(int comida[]) {
            this.comida = comida;
      }

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                return "input";
            }
            // sin errores
            else
            {
                return "mostrar";
            }
      }
}

// Clase de gestión de las profesiones
class Profesion
{
    private int idProfesion;
    private String nombre;
    public Profesion(int idProfesion, String nombre)
    {
      this.idProfesion=idProfesion;
      this.nombre=nombre;
    }

    // getter y setter
```

```
}

//Clase de gestión de las comida
class Comida
{
    private int id;
    private String nombre;
    public Comida(int id, String nombre)
    {
      this.id=id;
      this.nombre=nombre;
    }

    // getter y setter
}
```

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
<h3>Agregar un cliente</h3>
<s:form method="post" action="ConfirmarAgregar_Cliente">
<s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
<s:select name="profesion" label="Profesión" labelposition="top"
list="listaProfesiones" listKey="idProfesion" listValue="nombre"/>
<s:checkboxlist name="comida" label="Comida" list="listaComida"
listKey="id" listValue="nombre" labelposition="top"/>
<s:submit value="Agregar el cliente"/>
</s:form>
</div>
</body>
</html>
```

15. Otras etiquetas de los formularios de Struts

Struts propone otras etiquetas para facilitar la configuración de componentes HTML. Volvemos a encontrar la etiqueta `<s:combobox/>` que propone una etiqueta `<input/>` asociada a una lista desplegable `<select/>`, esta etiqueta permite enviar el texto y el valor de la etiqueta `<option/>` de la lista desplegable. La etiqueta `<s:updownselect/>` permite crear listas múltiples con la posibilidad de clasificar los elementos. La etiqueta `<s:optiontransferselect/>` permite crear un objeto complejo con una lista principal y una lista de destino con el fin de desplazar los ítems de la primera lista a la segunda. Por último, la etiqueta `<s:doubleselect/>` permite crear dos listas desplegables HTML asociadas. Las elecciones en la primera lista provocan modificaciones en la segunda lista.

Todas estas etiquetas, aunque en ocasiones son útiles, no se utilizan con frecuencia en el desarrollo de Internet. Por experiencia, es preferible desarrollar diferentes herramientas en DHTML con bibliotecas JavaScript o Ajax (como JQuery o Prototype).

Las etiquetas de control

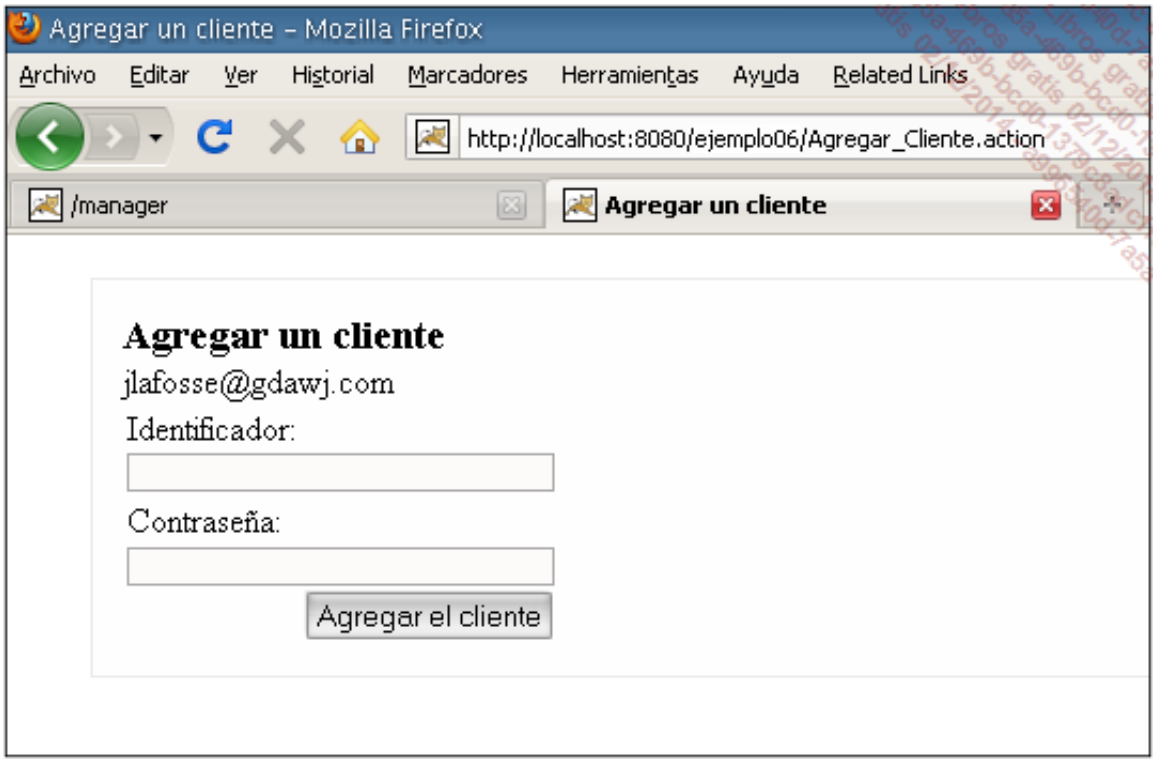
Las etiquetas de Struts estudiadas anteriormente permiten gestionar la presentación y visualización de los datos en las vistas. No obstante, el framework facilita también un conjunto de etiquetas para la gestión de los accesos a los datos, las condicionales o los bucles de programación, así como para la manipulación de las propiedades de la aplicación.

1. La etiqueta <s:property/>

La etiqueta `<s:property/>` permite mostrar, a partir de expresiones OGNL, datos presentes en la clase de acción asociada con sus descriptores de acceso o en el contexto de la aplicación (application, session, request, parameters, attr). El atributo `value` permite nombrar la propiedad y el parámetro `espace` permite evitar los caracteres HTML especiales (‘,”, &, < y >).

Vamos a retomar el ejemplo anterior *ejemplo05* y utilizar las etiquetas de control en nuestro nuevo proyecto *ejemplo06*.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:property value="emailWebmaster"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```



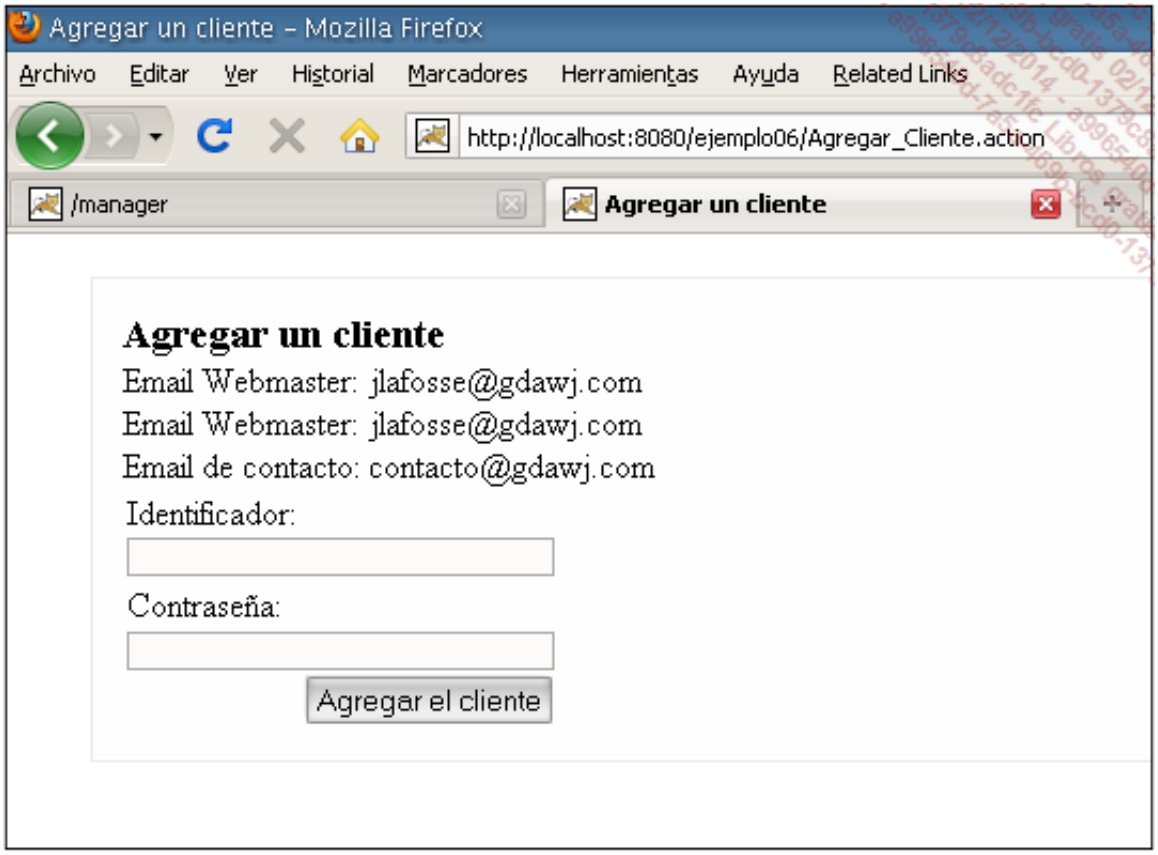
Utilización de la etiqueta `<s:property/>`

La notación JSP EL Expression Language también puede utilizarse con el dólar y las llaves: ${emailWebmaster}.

El segundo ejemplo permite leer un valor presente en el contexto de la aplicación. El alcance de la variable se precisa en el nombre de la misma con el carácter almohadilla #. Agregaremos un parámetro de aplicación en el archivo */WEB-INF/web.xml* para la dirección de correo electrónico de contacto.

```
Código: /WEB-INF/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>emailContacto</param-name>
        <param-value>contacto@gdawj.com</param-value>
     </context-param>

    <filter>
        <filter-name>struts2</filter-name>
        <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAnd
ExecuteFilter</filter-class>
    </filter>

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

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            Email Webmaster: <s:property
value="emailWebmaster"/><br/>
            Email Webmaster: ${emailWebmaster}<br/>
            Email de contacto: <s:property
value="#aplicación.emailContacto"/><br/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
</div>
</body>
</html>
```



Utilización de la notación EL

2. La etiqueta <s:a/>

La etiqueta `<s:a/>` permite crear un vínculo HTML conforme. Esta etiqueta se utiliza muy poco con Struts ya que en realidad no aporta ventajas en relación con la etiqueta HTML estándar, exceptuando que permite gestionar el id de la sesión.

3. La etiqueta <s:action/>

La etiqueta `<s:action/>` se utiliza para ejecutar una acción y recuperar el resultado de la misma en una variable.

Por ejemplo, podemos ejecutar una acción y almacenar el resultado en una variable denominada*objeto*.

```
<s:action var="objeto" name="miAccionAEjecutar"/>
```

4. La etiqueta <s:param/>

La etiqueta `<s:param/>` permite pasar parámetros a una acción. Ésta se utiliza en vínculos o formularios.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            Email Webmaster: <s:property
value="emailWebmaster"/><br/>
            Email Webmaster: ${emailWebmaster}<br/>
            Email de contacto: <s:property
value="#aplicación.emailContacto"/><br/>
            <s:submit value="Agregar el cliente"/>
      </s:form>

      <s:url id="mostrarURL" action="Mostrar" >
         <s:param name="páginaActual">${header.host}</s:param>
      </s:url>
      <s:a href="%{mostrarURL}">Pasar un par&aacute;metro a
una acci&oacute;n</s:a>
</div>
</body>
</html>
```

Se define una URL, así como un parámetro denominado *paginaActual* que permite recuperar el nombre del sistema local. El vínculo de Struts se configura automáticamente con estos datos.

5. La etiqueta <s:bean/>

La etiqueta `<s:bean/>` permite crear una instancia de una clase JavaBean. Esta etiqueta es similar a la etiqueta JSP `<jsp:useBean/>`. Esta etiqueta facilita el atributo `name`, que permite definir una clase JavaBean, y el atributo `id`, que permite nombrar la instancia. Esta etiqueta `<s:bean/>` se utiliza frecuentemente asociada con la etiqueta `<s:param/>` para asignar los valores al JavaBean.

Vamos a definir una nueva clase JavaBean denominada `Profesion` y a utilizar la etiqueta `<s:bean/>`.

```
Código: ejemplo06.Profesion.java
package ejemplo06;

public class Profesion {

	private int id;
	private String nombre;

	public Profesion()
	{

	}

	// getter y setter
}
```

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
	<h3>Agregar un cliente</h3>
	<s:bean name="ejemplo06.Profesion" id="profesion">
		<s:param name="id" value="1"/>
		<s:param name="nombre" value="'Formador'"/>
	</s:bean>
	Profesi&oacute;n: <s:property value="#profesion.id"/>
<s:property value="#profesion.nombre"/>
</div>
</body>
</html>
```

La inicialización de una propiedad de tipo cadena de caracteres en Struts se realiza con comillas. En el ejemplo vemos cómo comienza la propiedad nombre de la profesion. `<s:param name="nombre" value="'Formador'"/>`.

---

6. La etiqueta <s:date/>

La etiqueta `<s:date/>` se utiliza para configurar la visualización de las fechas. Esta etiqueta es muy útil para mostrar las fechas en función del país del usuario habitual. Por ejemplo, un usuario español preferirá ver las fechas con formato dd/mm/aaaa, mientras que un inglés preferirá el formato aaaa-mm-dd.

La modificación de la vista JSP permite gestionar dos objetos de fecha con los dos tipos de formato anteriores.

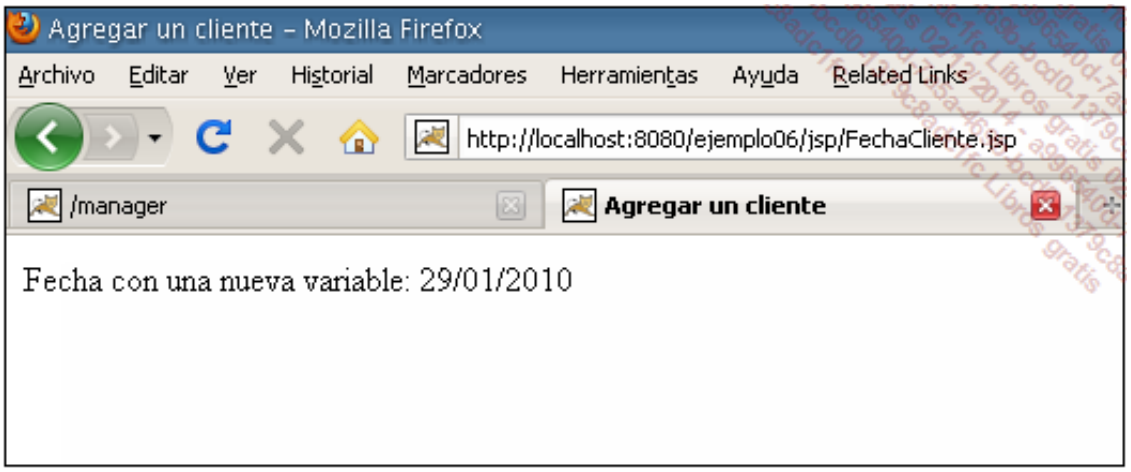
```
Código: /jsp/FechaCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
	<s:bean name="java.util.Fecha" id="fecha"/>
```

```
      Fecha: <s:property value="#fecha"/><br/>
      Fecha ES: <s:date name="#fecha" var="formato_es"
format="dd/MM/yyyy"/><s:property value="formato_es"/><br/>
      Fecha EN: <s:date name="#fecha" var="formato_en"
format="yyyy-MM-dd"/><s:property value="formato_en"/><br/>
</div>
</body>
</html>
```

7. La etiqueta <s:set/>

La etiqueta `<s:set/>` permite crear una propiedad asociada a su valor dinámico (tipo primitivo u objeto). La variable se puede crear en el contexto de la aplicación, en la sesión, en la solicitud actual o en la página.

```
Código: /jsp/FechaCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <s:bean name="java.util.Fecha" id="fecha"/>
      <s:set name="fechasolicitud" value="#fecha"/>
      Fecha con nueva variable: <s:property
value="#fechasolicitud"/>
</div>
</body>
</html>
```



Utilización de la etiqueta `<s:set/>`

Esta etiqueta se utiliza con frecuencia para almacenar un objeto complejo en una variable con el fin de acceder a sus valores en el resto del código.

8. La etiqueta <s:push/>

La etiqueta `<s:push/>` es muy similar a la etiqueta `<s:set/>`. Permite inicializar un objeto pero utilizarlo entre la etiqueta de inicio `<s:push>` y la etiqueta de finalización `</s:push>`.

9. La etiqueta <s:url/>

La etiqueta `<s:url/>` permite crear URL dinámicas que más tarde serán utilizadas en vínculos o formularios HTML. Esta etiqueta se utiliza en el ejemplo anterior para configurar un vínculo HTML con la ayuda de la etiqueta `<s:param/>`.

```
<s:url id="mostrarURL" action="Mostrar" >
  <s:param name="páginaActual">${header.host}</s:param>
</s:url>
<s:a href="%{mostrarURL}">Pasar un par&aacute;metro a
una acci&oacute;n</s:a>
```

10. Las etiquetas <s:if/>, <s:else/> y <s:elseif/>

Las etiquetas `<s:if/>`, `<s:else/>` y `<s:elseif/>` se utilizan para realizar pruebas condicionales. Estas etiquetas permiten comprobar si un atributo es nulo.

```
<s:if test="#profesion.id==null">
      Sin profesion
</s:if>
<s:else>
      Profesion <s:property value="#profesion.id"/>
</s:else>
```

11. La etiqueta <s:iterator/>

La etiqueta `<s:iterator/>` permite recorrer una tabla o una colección Java. Retomaremos el ejemplo de las profesiones para mostrar los valores.

```
Código: ejemplo06.Profesion.java
package ejemplo06;

public class Profesion {

      private int id;
      private String nombre;

      public Profesion()
      {

      }

      public Profesion(int id, String nombre)
      {
        this.id=id;
        this.nombre=nombre;
      }

      // getter y setter
}
```

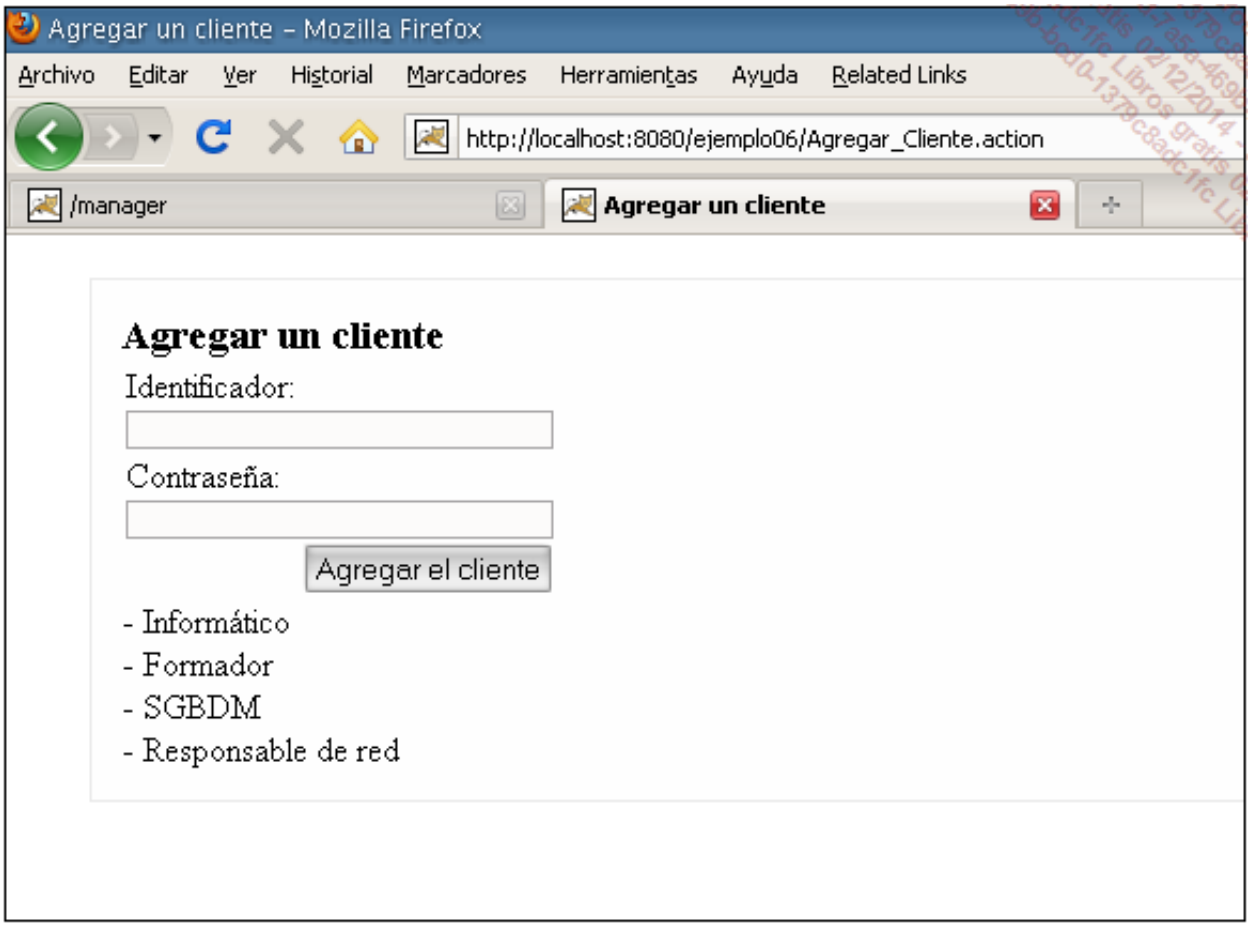
```
Código: ejemplo06.Cliente.java
package ejemplo06;

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

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {
```

```
    private String identificador;
    private String contrasena;
    private String emailWebmaster="jlaffosse@gdawj.com";
    private int profesion;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();

      public Cliente()
      {

      }

      public List<Profesion> getListaProfesiones() {
             listaProfesiones.add(new Profesion(1,
"Informático"));
             listaProfesiones.add(new Profesion(2,
"Formador"));
             listaProfesiones.add(new Profesion(3, "SGBDM"));
             listaProfesiones.add(new Profesion(4, "Responsable
de red"));
             return listaProfesiones;
      }

      public void setListaProfesiones(List<Profesion>
listaProfesiones) {
             this.listaProfesiones = listaProfesiones;
      }

      public int getProfesion() {
             return profesion;
      }

      public void setProfesion(int profesion) {
             this.profesion = profesion;
      }

      // getter y setter


      // agregar los datos del cliente a la sesión
      public String agregar()
      {
             // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
             if(this.identificador.equals("") ||
this.contrasena.equals(""))
             {
                    return "input";
             }
             // sin errores
             else
             {
                    return "mostrar";
             }
      }
}
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
```

```
</head>
<body>
<div id="carta">
<h3>Agregar un cliente</h3>
<s:form method="post" action="ConfirmarAgregar_Cliente">
<s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
<s:submit value="Agregar el cliente"/>
</s:form>
<s:iterator value="listaProfesiones">
<s:property value="id"/> - <s:property value="nombre"/><br/>
</s:iterator>
</div>
</body>
</html>
```

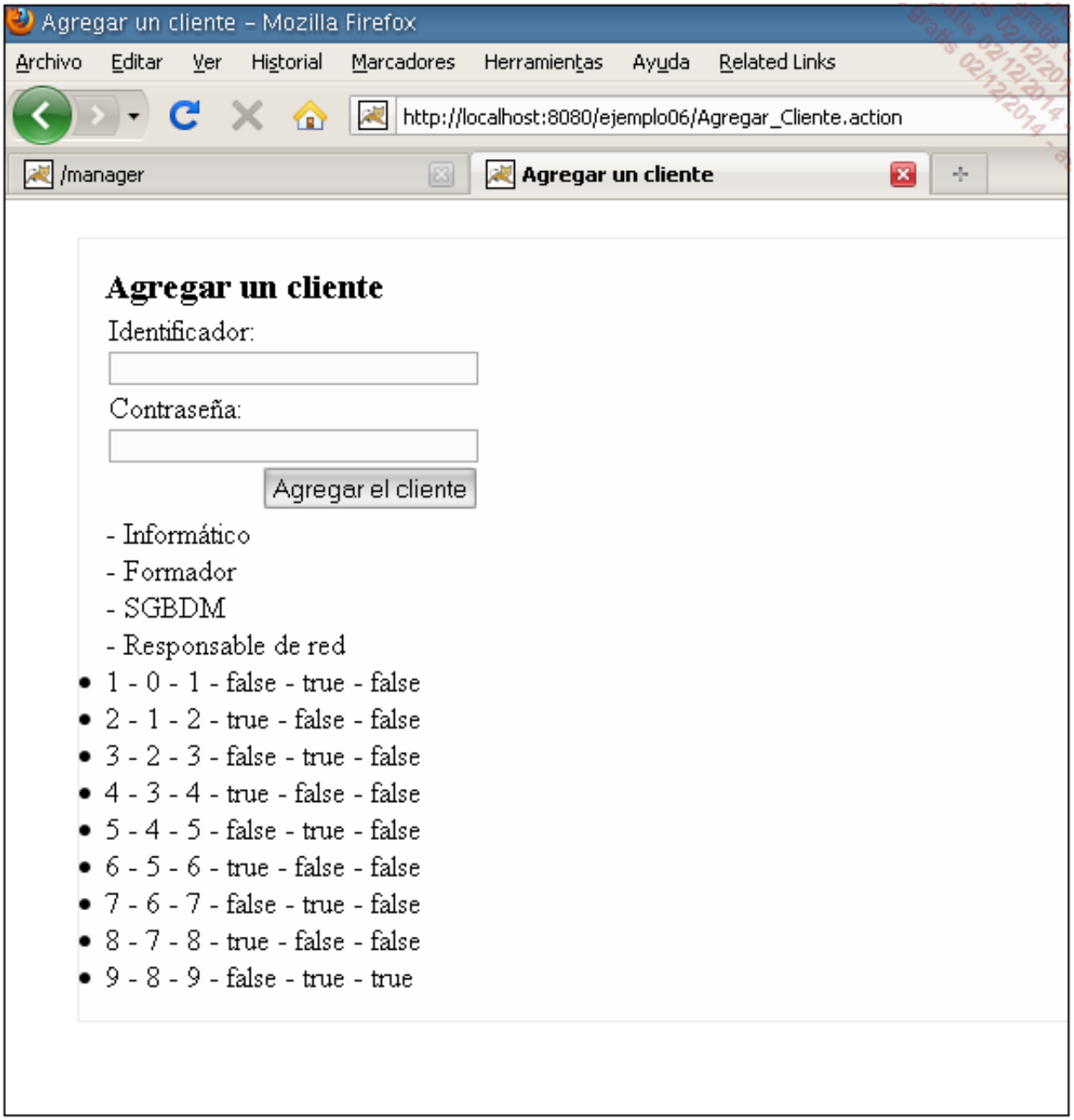


Utilización de la etiqueta `<s:iterator/>`

La etiqueta `<s:iterator/>` permite realizar bucles en los valores con el fin de simular el recorrido de los mismos. Este segundo ejemplo de la vista JSP permite mostrar una lista numerada.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
```

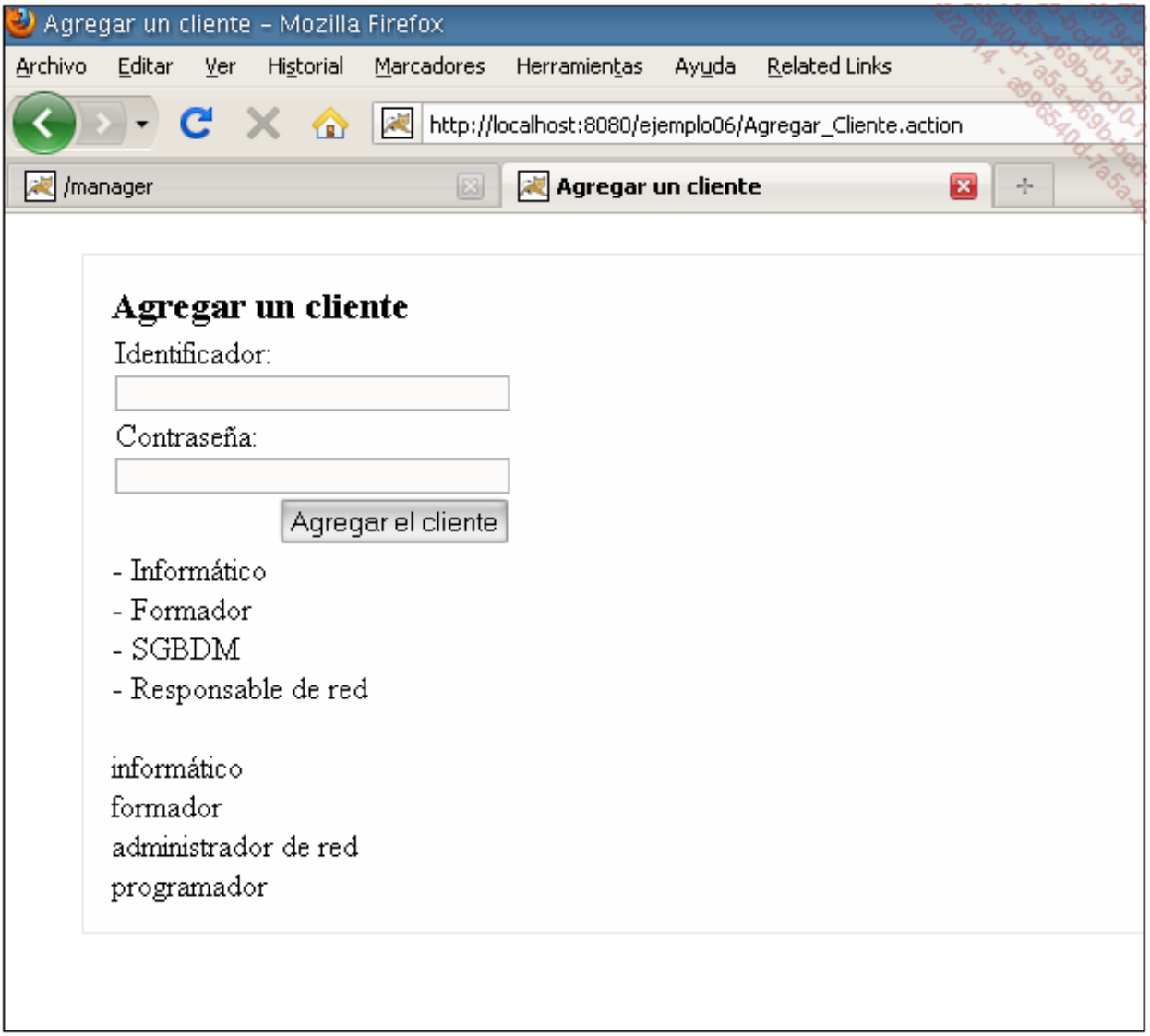
```
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
      <s:iterator value="listaProfesiones">
       <s:property value="id"/> - <s:property value="nombre"/><br/>
      </s:iterator>
      <ul>
      <s:iterator value="{1,2,3,4,5,6,7,8,9}" status="status">
            <li><s:property/> - <s:property
value="#status.index"/> - <s:property value="#status.count"/>
            - <s:property value="#status.even"/> - <s:property
value="#status.odd"/> - <s:property value="#status.last"/>
            </li>
      </s:iterator>
      </ul>
</div>
</body>
</html>
```



12. La etiqueta <s:append/>

La etiqueta `<s:append/>` permite concatenar elementos de iteración para las listas de datos. Esta etiqueta se utiliza principalmente para crear una nueva lista a partir de otras dos.

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
      <s:iterator value="listaProfesiones">
       <s:property value="id"/> - <s:property value="nombre"/><br/>
      </s:iterator>

      <br/>
      <s:set var="lista1" value="{'informático','formador'}"/>
      <s:set var="lista2" value="{'administrador
de red','programador'}"/>
      <s:append var="listacompleta">
            <s:param value="lista1"/>
            <s:param value="lista2"/>
      </s:append>
      <s:iterator value="#listacompleta">
            <s:property/><br/>
      </s:iterator>
</div>
</body>
</html>
```



Utilización de la etiqueta `<s:append/>`

13. La etiqueta <s:sort/>

La etiqueta `<s:sort/>` se utiliza para clasificar los componentes de un elemento de iteración de una colección.

```
Código: ejemplo06.Cliente.java
package ejemplo06;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private String emailWebmaster="jlaffosse@gdawj.com";
    private int profesion;
    private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();
```

```
      public Cliente()
      {

      }

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
            return "input";
            }
            // sin errores
            else
            {
                return "mostrar";
            }
      }

      // comparar dos objetos
      public Comparator<Object> getMyComparator()
      {
            return new Comparator<Object>()
            {
                public int compare(Object o1, Object o2)
                {
                    return
o1.toString().compareTo(o2.toString());
                 }
            };
      }
}
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" label="Contraseña"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar el cliente"/>
      </s:form>
      <s:sort comparator="myComparator" source="listaProfesiones"
id="clasifListaProfesiones"/>
      <s:iterator value="#atr.clasifListaProfesiones">
            <s:property value="id"/> - <s:property
value="nombre"/><br/>
      </s:iterator>
</div>
</body>
```

```
</html>
```

En resumen

En este capítulo se presentan las diferentes etiquetas de Struts para formatear los datos y las acciones. En la primera parte se detallan las etiquetas de los formularios utilizadas en las páginas JSP o vistas, con el fin de facilitar el formateo de los datos. El segundo párrafo presenta, siempre a través de ejemplos, las etiquetas de controles para acceder a los datos, las estructuras y las clasificaciones.

Presentación

La gestión de los mensajes de error y confirmación es una tarea muy importante durante el desarrollo de aplicaciones informáticas. La internacionalización en programación y desarrollo consiste en gestionar los idiomas y así facilitar páginas multilingües. El término i18n se ha estandarizado para la internacionalización y corresponde en realidad a los 18 caracteres que componen la palabra internacionalización entre la i y la n. En programación, el idioma se presenta mediante la configuración regional. Dicha configuración se define en función del idioma y el país.

La configuración regional se compone de dos parámetros:

La primera parte, el *idioma*, guión bajo, y la segunda parte, el *país*.

Así, por ejemplo el español de España se representaría de la siguiente forma: *es_ES*.

El español de Argentina: *es_AR*.

El inglés de EE.UU.: *en_US*.

Para la localización de los mensajes en Java, utilizamos un conjunto de archivos denominados bundle en inglés. Se utiliza un archivo de propiedades para cada idioma/configuración regional. Cada archivo cuenta con un prefijo común, *basename*, y debe tener la extensión *.properties*.

Los archivos de idiomas concretos deben tener el mismo prefijo seguido de guión bajo y del código del idioma. Para ser accesibles, estos archivos deben estar incluidos en el *classpath*.

La internacionalización permite cambiar la visualización del texto o las imágenes por ejemplo, en función del tipo de usuario. La plataforma Java EE se ha concebido para ser compatible con las aplicaciones multilingües mediante un principio simple y eficaz que en cierta medida se utiliza en Struts. Las clases de acción de Struts utilizan el método `getText()` para leer los mensajes presentes en un archivo de texto de propiedades en función del idioma.

Por último, todas las aplicaciones profesionales deberían utilizar el mecanismo de internacionalización incluso en los desarrollos monolingües (un solo idioma). Recurrir a un archivo de propiedades, incluso en un mismo idioma, permite mejorar el futuro mantenimiento y modificar los textos estáticos sin necesidad de volver a compilar la aplicación.

Aplicación

Como hemos dicho anteriormente, la configuración regional está definida por un idioma y un país. Tenemos por ejemplo el inglés de EE.UU. (*en_US*) y el inglés de Canadá (*en_CA*). El idioma es la parte más importante de la definición y en la mayoría de los casos se puede utilizar sin el país (por ejemplo:*propiedades_es.properties*, *propiedades_en.properties*).

Las aplicaciones multilingües utilizan archivos de texto compuestos de una clave/valor para cada configuración regional. Cada par de valores se compone de una clave única y de su valor asociado. Las claves son de tipo cadena de caracteres y los valores de las claves pueden ser de cualquier tipo.

El nombre de los archivos de propiedades utilizados para leer la información debe estar compuesto de la forma siguiente:

- *basename_idioma_país.properties*

Así, tenemos el campo *basename* que corresponde al nombre del archivo (por ejemplo: recursos), el idioma (por ejemplo: es), el país (por ejemplo: ES) y por último la extensión del archivo (*.properties*). De este modo, para nuestros ejemplos tendríamos dos archivos: *recursos_es.properties* y*recursos_en.properties*.

La utilización de la internacionalización con Struts es muy sencilla, ya que cada clase heredera de la clase `ActionSupport` puede utilizar la internacionalización. Asimismo, la interfaz*com.opensymphony.xwork2.TextProvider* facilita el acceso a los bundles del proyecto. El método`getText(String clave)` permite recuperar el valor asociado a la clave convertida en parámetro. El resultado de este método será *null* si la clave no se encuentra en el archivo de propiedades.

La segunda escritura del método `getText(String clave, String valorpordefecto)` permite recuperar el valor asociado a la clave convertida en parámetro y el valor predeterminado si dicha clave no se encuentra en el archivo de propiedades.

La tercera escritura `getText(String clave, String[] formato)` permite recuperar el valor asociado a la clave convertida en parámetro, en relación con el formato especificado en los parámetros.

La escritura `getText(String clave, String valorpordefecto, String[] formato)` permite recuperar el valor asociado a la clave convertida en parámetro, en relación con el valor predeterminado y el formato asociado.

Acceso a las propiedades

El acceso a los datos presentes en los archivos de propiedades mediante los métodos dedicados como `getText(...)` se realiza en el orden siguiente:

- A un archivo de propiedades que tenga el mismo nombre que la clase de acción y esté presente en el directorio */WEB-INF/clases* al mismo nivel que la clase de acción.

  Por ejemplo: la clase `Cliente.java` está asociada con el archivo de propiedades*Cliente.properties*.

  Esta técnica, aunque de sencilla implementación, se utiliza poco por razones de agilidad en el mantenimiento (un archivo por clase).

- A un archivo de propiedades que tenga el mismo nombre que una interfaz utilizada por cada clase que lo necesite. Por ejemplo: la interfaz *Recursos* utilizada por cada clase usuaria y el archivo *Recursos.properties*.

- A un archivo de propiedades que tenga el mismo nombre que una clase heredada por cada clase que lo necesite. Por ejemplo: el archivo de propiedades *ActionSupport.properties* utilizado por cada clase heredera de la clase `ActionSupport`.

- A un archivo de modelo si nuestras clases implementan la interfaz*com.opensymphony.xwork2.ModelDriven*.

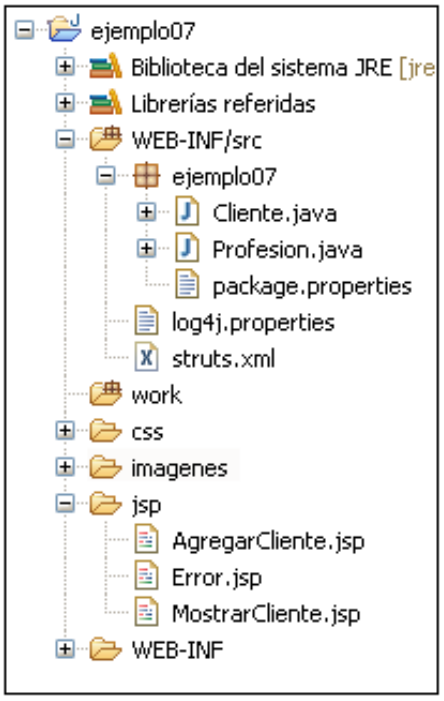
- A un archivo de propiedades presente en el paquete predeterminado del proyecto. Por ejemplo: *paquete ejemplo07*, */ejemplo07/recursos.properties*.

- A un archivo de propiedades presente en un paquete secundario de la aplicación. Por ejemplo: *ejemplo07.recursos*.

Para mostrar los datos presentes en el archivo de propiedades, podemos utilizar la etiqueta`<s:property/>` o el atributo `label` de las etiquetas asociadas al método `getText(...)`: `%{getText('clave')}`. Vamos a utilizar de nuevo el ejemplo anterior de gestión de las cuentas clientes con el fin de manipular los mensajes del archivo de propiedades.

El nuevo proyecto utilizado está disponible en la siguiente dirección:

http://localhost:8080/ejemplo07/Agregar_Cliente.action



Árbol del proyecto ejemplo07

Código: 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="ejemplo07" namespace="/" extends="struts-default">
        <default-action-ref name="agregar_Cliente" />

        <action name="Agregar_Cliente" class="ejemplo07.Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo07.Cliente" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="SUCCESS">/jsp/MostrarCliente.jsp</result>
        </action>

    </package>
</struts>
```

Código: ejemplo07.Cliente.java

```
package ejemplo07;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;

      public Cliente()
      {

      }

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                return "input";
            }
            // sin errores
            else
            {
                return SUCCESS;
            }
      }
}
```

Código: /jsp/AgregarCliente.jsp

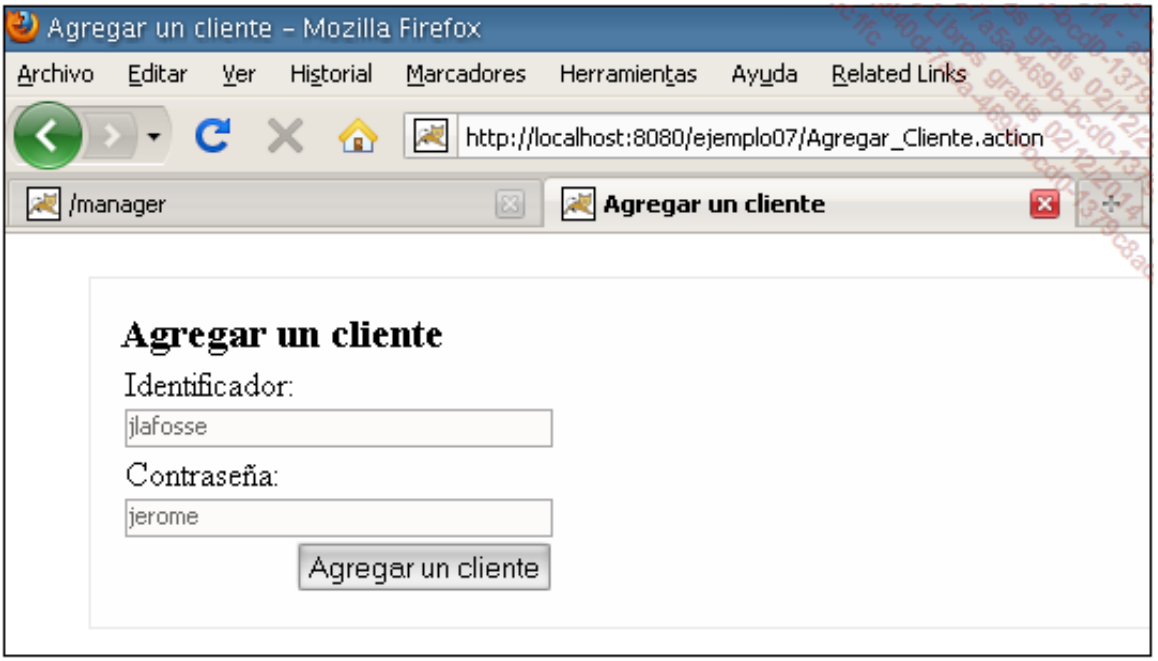
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.agregar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3><s:property value="%{getText('cliente.agregar')}"/></h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="%{getText('cliente.identificador')}" labelposition="top"
cssClass="input"/>
            <s:textfield name="contrasena" label="%
{getText('cliente.contrasena')}" labelposition="top"
cssClass="input"/>
            <s:submit value="%{getText('cliente.agregar')}"/>
      </s:form>
</div>
</body>
</html>
```

Código: /jsp/MostrarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%{getText('cliente.identificador')}"/>:
<s:property value="identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/>:
<s:property value="contrasena"/><br/>
      </p>
</div>
</body>
</html>
```

Código: /ejemplo07/package.properties

```
cliente.agregar=Agregar un cliente
cliente.identificador=Identificador
cliente.contrasena=Contraseña
cliente.mostrar=Mostrar un cliente
```



Agregar un cliente ejemplo07

Podemos observar que en nuestro ejemplo, el acceso a los datos se realiza en el archivo*/ejemplo07/package.properties* presente en el paquete predeterminado y al que por tanto podrán acceder todas las clases de dicho paquete. Este archivo está compuesto de parejas clave/valor en formato de texto. En las vistas, el acceso a dichos datos se realiza de dos formas, con la etiqueta`<s:property/>` y con el atributo `label`:

```
<s:property value="%{getText('cliente.agregar')}"/>
<s:textfield name="identificador" id="identificador" label="%
{getText('cliente.identificador')}"/>
```

Esta aplicación, aunque monolingüe, permite gestionar de forma sencilla los mensajes simplemente editando los archivos de propiedades estáticos sin compilar de nuevo los archivos de origen.

Datos multilingües

En el ejemplo anterior hemos mostrado las dos técnicas que ofrece Struts para acceder a los datos de los archivos de propiedades, la etiqueta `<s:property/>` y el atributo `label` de las diferentes etiquetas. Asimismo, Struts propone la etiqueta `<s:text/>` para el acceso a las propiedades. Esta etiqueta es equivalente al uso del método `getText(...)` en la etiqueta `<s:property/>`, y permite únicamente una escritura más ligera del acceso a los datos.

Las declaraciones siguientes también son equivalentes:

```
<s:property value="%{getText('cliente.agregar')}"/>
<s:text name="cliente.agregar"/>
```

Si el atributo `id` está presente en la etiqueta `<s:text/>`, esta no se muestra pero se almacena en la variable indicada en la propiedad. Así, esta propiedad puede mostrarse con la etiqueta`<s:property/>`.

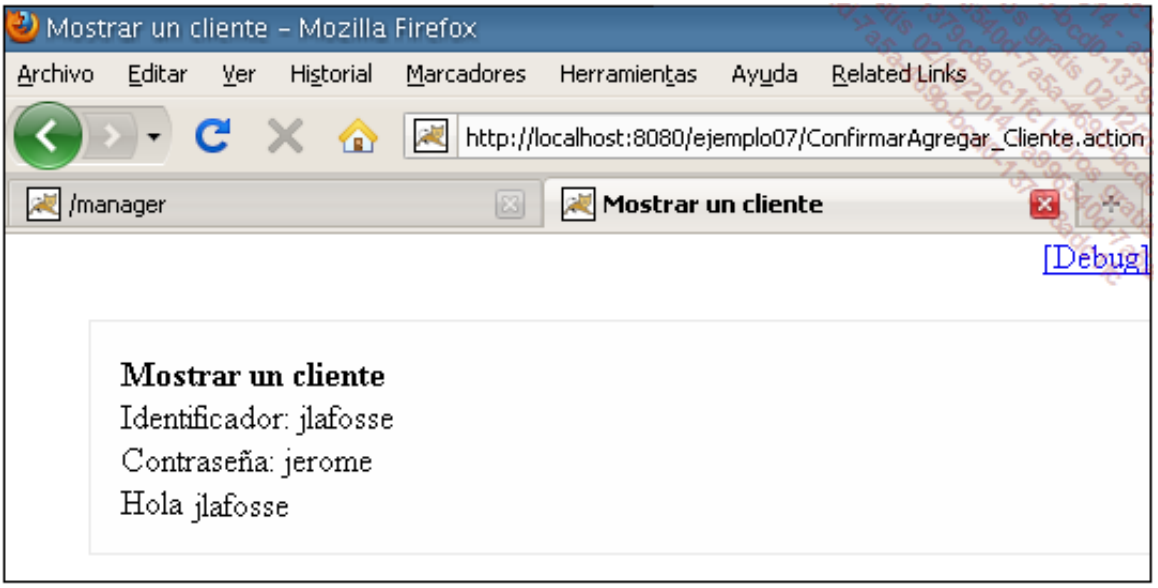
```
<s:text name="cliente.agregar" id="msj_clienteagregar"/>
<s:property value="#msj_clienteagregar"/>
```

Por último, la etiqueta `<s:text/>` permite transmitir parámetros presentes en el archivo de propiedades. Esta técnica se utiliza frecuentemente para mejora la ergonomía. Vamos a agregar una línea a nuestro archivo de propiedades *paquete.properties*.

```
cliente.bienvenida=Hola {0}
```

Ahora podemos transmitir el identificador de la persona autenticada a través de la etiqueta `<s:text/>`a la página */jsp/MostrarCliente.jsp*.

```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
>: <s:property value="contrasena"/><br/>
            <s:text name="cliente.bienvenida">
                <s:param><s:property
value="identificador"/></s:param>
            </s:text>
      </p>
</div>
</body>
</html>
```

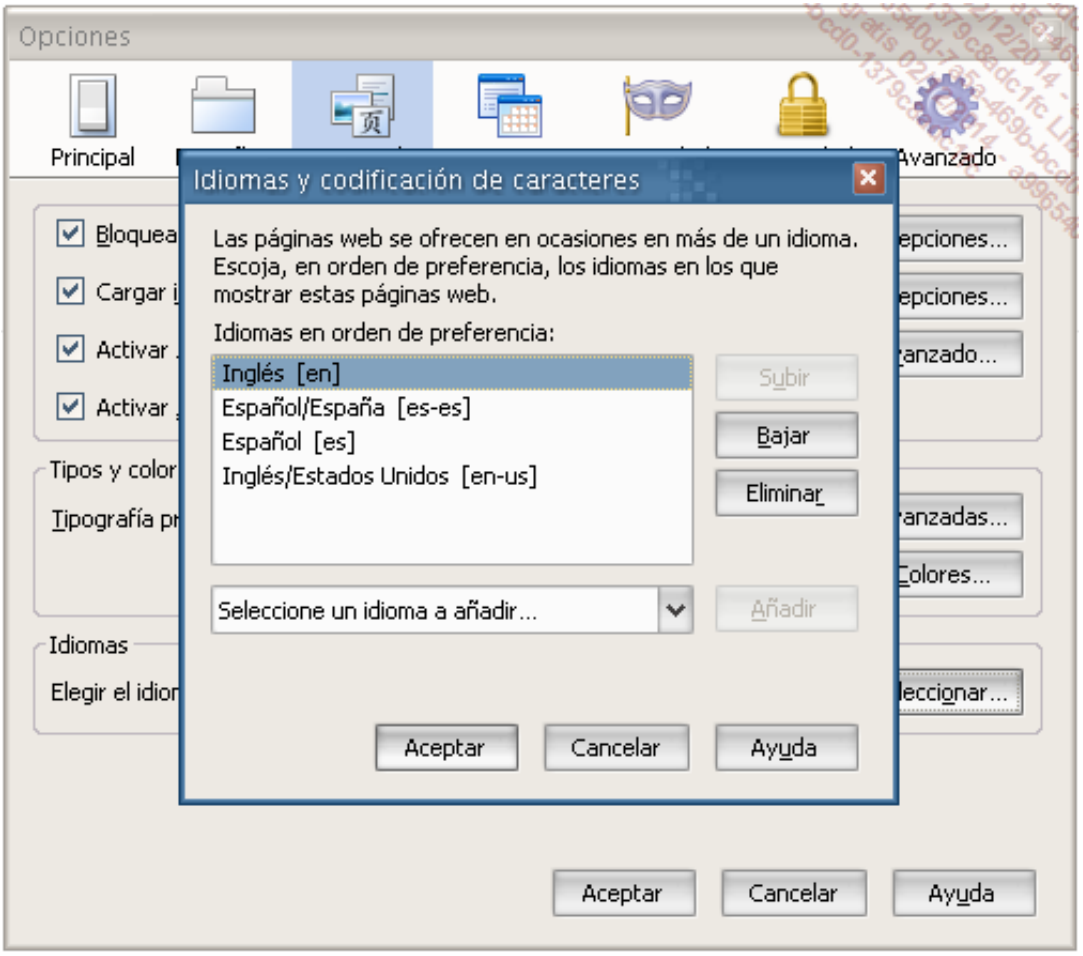


Visualización de la información del cliente ejemplo07

Ahora vamos a utilizar un segundo archivo de propiedades *package_en.properties* para gestionar la visualización en inglés. Para ello, simplemente copiamos/pegamos el archivo con las traducciones asociadas.

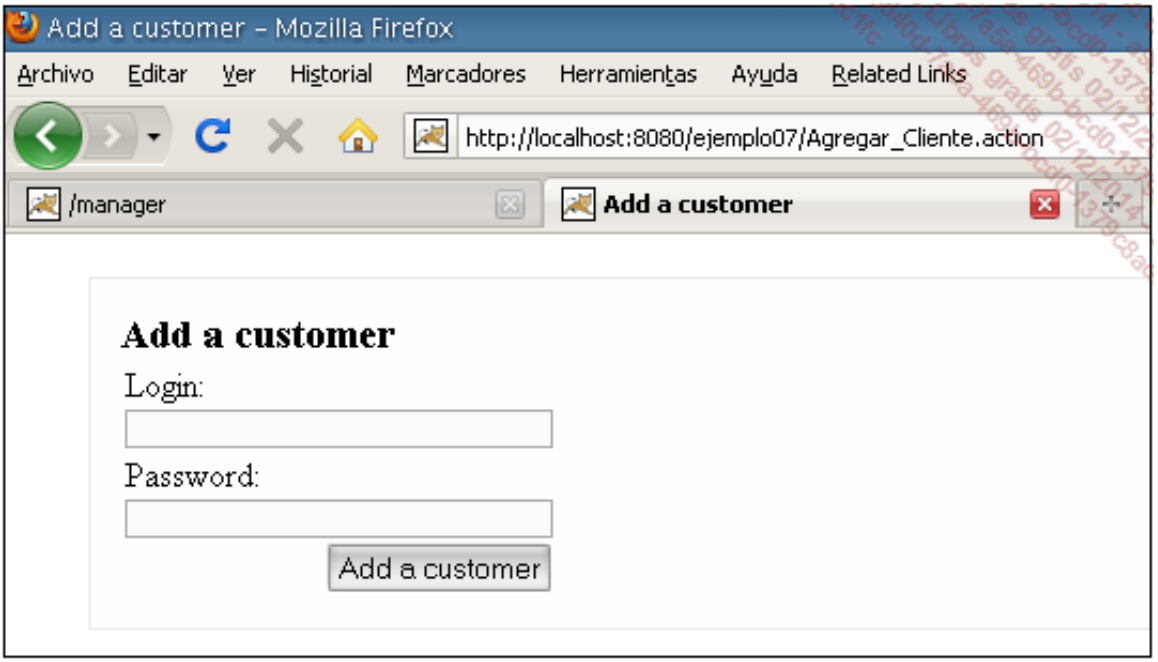
```
Código: /ejemplo07/package_en.properties
cliente.agregar=Add a customer
cliente.identificador=Login
cliente.contrasena=Password
cliente.mostrar=Show a customer
cliente.bienvenida=Hello {0}
```

Antes de comprobar sus características, configuraremos nuestro navegador en inglés. Para ello, en Firefox utilizamos el menú **Herramientas - Opciones - Contenido - Idiomas - Seleccionar** y el idioma:**Inglés**. Para que estos cambios tengan efecto, es necesario cerrar el navegador y abrir una nueva ventana.



Modificación del idioma predeterminado del navegador

Ahora podemos abrir de nuevo el navegador e iniciar la página del formulario del cliente. Así, el sitio utilizará automáticamente las traducciones presentes en el archivo asociado a la configuración regional *_en* (*package_en.properties*).



Formulario de cliente inglés ejemplo07

El principio es idéntico independientemente del número de idiomas utilizados. Tendremos

tantos archivos de propiedades con el sufijo correspondiente a la configuración regional adaptada como idiomas utilizados en la aplicación.

La etiqueta `<s:i18n/>` permite cargar de manera dinámica recursos o bundles. Esta técnica se utiliza principalmente para asociar una clave a un objeto complejo dinámico que no sea una cadena de caracteres. La etiqueta `<s:i18n/>` contiene un atributo `name` que permite especificar el bundle que desea que se utilice.

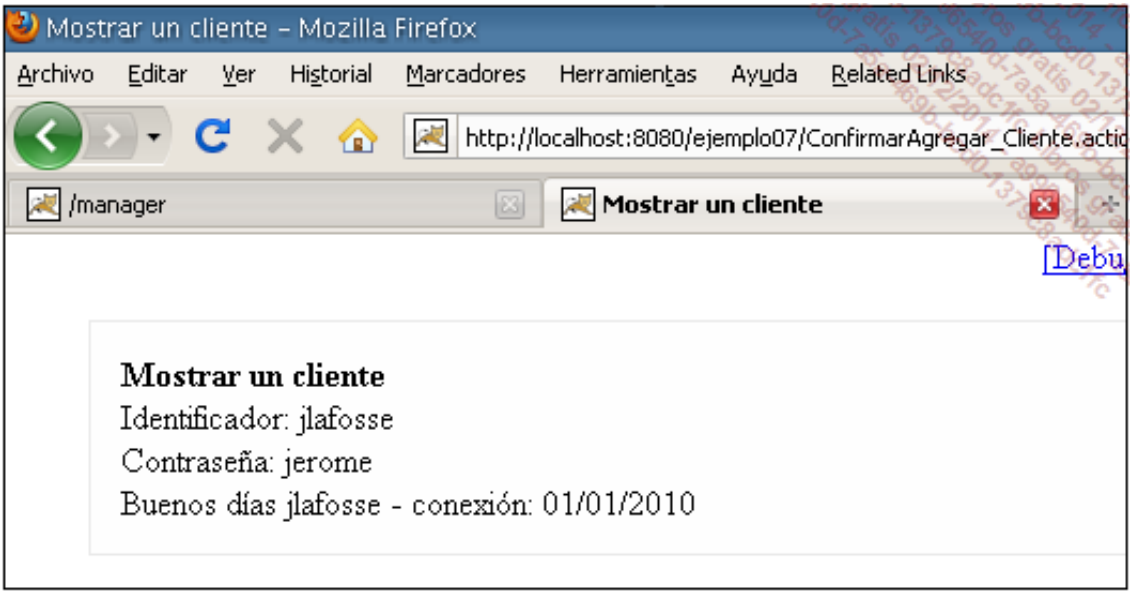
Para ilustrar nuestro ejemplo, vamos a definir una clase denominada `MisRecursos.java` que devolverá un contenido de tipo complejo adaptado al tipo de autenticación. Así, si el usuario conectado lo hace en español, la fecha de conexión se mostrará automáticamente con el formato dd/mm/aaaa, y si lo hace en inglés, la fecha de conexión se mostrará con el formato aaaa-mm-dd.

Las clases de utilización de los bundles dinámicos deben ser herederas de la clase`java.util.ListResourceBundle`.

```
Código: ejemplo07.MisRecursos.java
package ejemplo07;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.ListResourceBundle;

public class MisRecursos extends ListResourceBundle{

// objeto fecha y presentación
      Date fecha=new Date();
      SimpleDateFormat fechadia=new
SimpleDateFormat("dd/MM/yyyy");

      // devolver el contenido adaptado
      public Object[][] getContents()
      {
            return traducciones;
      }

      // contenido dinámico (clave/valor)
      Object[][] traducciones= {
            {"cliente.bienvenida","Buenos días{0} - conexión:
"+fechadia.format(fecha)}
      };
}
```

```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
```

```
>: <s:property value="contrasena"/><br/>

            <s:i18n name="ejemplo07.MisRecursos">
            <s:text name="cliente.bienvenida">
                  <s:param name="identificador"><s:property
value="identificador"/></s:param>
            </s:text>
            </s:i18n>
      </p>
</div>
</body>
</html>
```

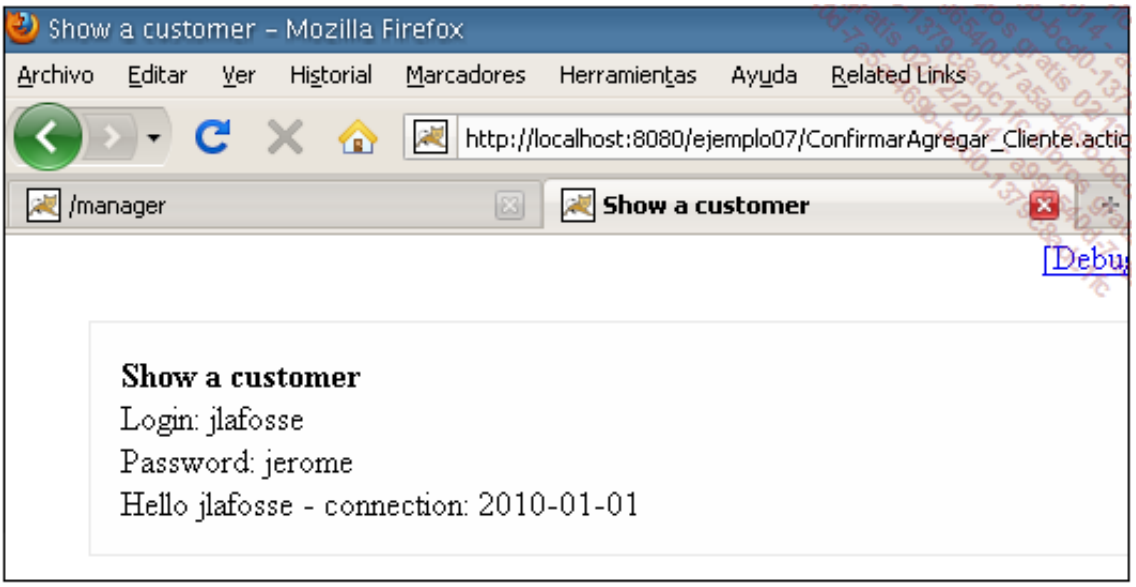


Visualización del cliente ejemplo07

Ahora, si deseamos utilizar un mensaje de autenticación adaptado para el inglés (configuración regional *en*), debemos crear una clase con el mismo nombre pero con el sufijo de la configuración regional (*MisRecursos_en.java*).

```
Código: ejemplo07.MisRecursos_en.java
package ejemplo07;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.ListResourceBundle;

public class MisRecursos_en extends ListResourceBundle{

      // objeto fecha y presentación
      Date fecha=new Date();
      SimpleDateFormat fechadia=new SimpleDateFormat("yyyy-mm-dd");

      // devolver el contenido adaptado
      public Object[][] getContents()
      {
            return traducciones;
      }

      // contenido dinámico (clave/valor)
      Object[][] traducciones= {
            {"cliente.bienvenida","Hello {0} - connection:
"+fechadia.format(fecha)}
      };
```

```
}
```



Visualizar el cliente ejemplo07

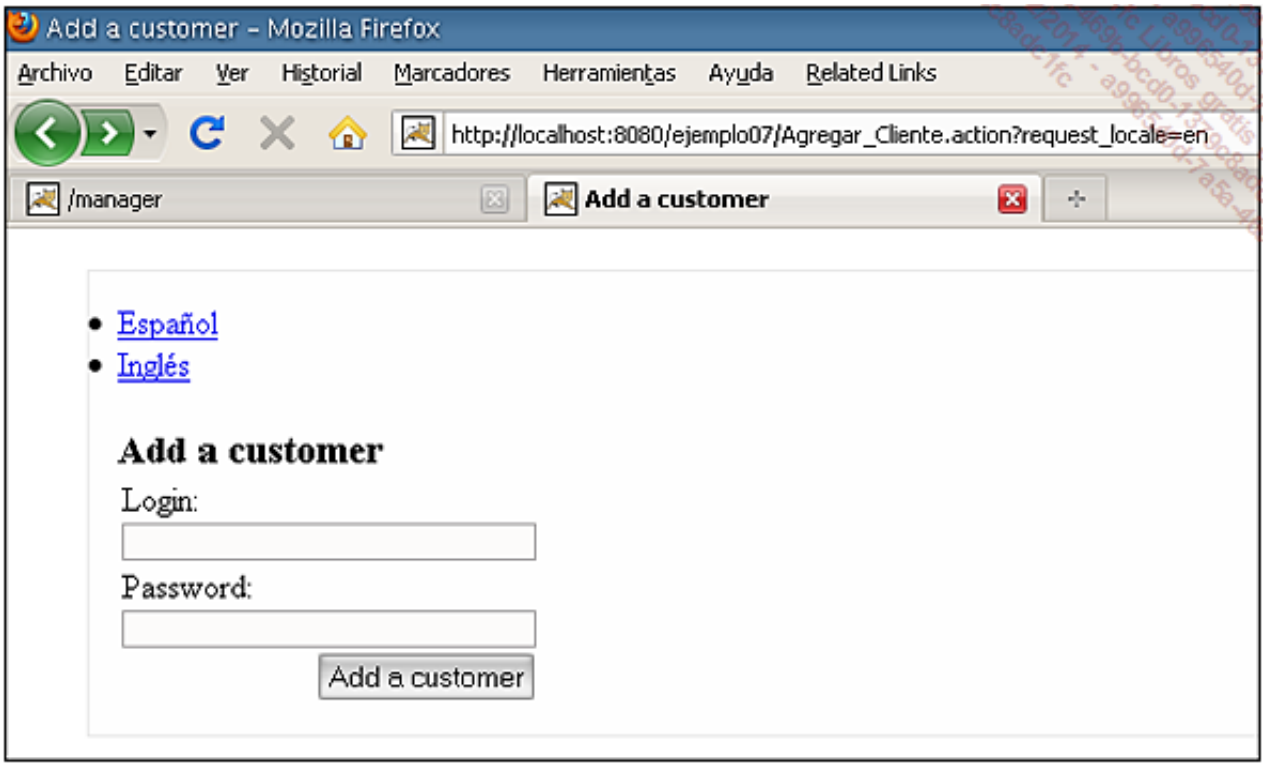
Selección dinámica de archivos

Los archivos de propiedades utilizados hasta ahora se cargan en función de la configuración regional del navegador y por tanto de la selección del idioma. Hemos utilizado las propiedades del navegador para realizar la elección del idioma. No obstante, esta manipulación requiere que se cargue de nuevo el navegador. En una aplicación profesional, los idiomas deben poder modificarse de manera dinámica, por ejemplo a través de pequeñas banderas que representen los idiomas. Para ello, la clase`ActionSupport` propone una solución de internacionalización.

Para poder cambiar de manera dinámica el idioma de la aplicación, Struts facilita el parámetro*request_locale*. Vamos a utilizar este parámetro para cambiar el idioma a través de un vínculo HTML. Estos vínculos se situarán en la primera acción y en la vista asoaciada *AgregarCliente.jsp*. Por supuesto, podríamos utilizar cualquier página JSP o JSPF para gestionar los idiomas (por ejemplo:*encabezado.jspf*).

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.agregar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <s:url action="Agregar_Cliente" id="urlIdiomaES">
            <s:param name="request_locale">es</s:param>
      </s:url>
      <s:url action="Agregar_Cliente" id="urlIdiomaEN">
            <s:param name="request_locale">en</s:param>
      </s:url>
      <ul>
            <li><s:a href="%
{urlIdiomaES}">Español</s:a></li>
            <li><s:a href="%{urlIdiomaEN}">Inglés</s:a></li>
      </ul>
      <br/>
      <h3><s:property value="%{getText('cliente.agregar')}"/></h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="%{getText('cliente.identificador')}" labelposition="top"
cssClass="input"/>
            <s:textfield name="contrasena" label="%
{getText('cliente.contrasena')}" labelposition="top"
cssClass="input"/>
            <s:submit value="%{getText('cliente.agregar')}"/>
      </s:form>
</div>
</body>
</html>
```

La primera parte del código permite únicamente crear vínculos HTML con los parámetros. Puede sustituirse por las líneas siguientes: `<ul><li><a href="Agregar_Cliente.action?request_locale=es">Espa&ntilde;ol</a></li><li><a href="Agregar_Cliente.action?request_locale=en">Ingl&eacute;s</a></li></ul>`



Gestión de idiomas con la etiqueta `<s:i18n/>`

Acceso a los recursos de las clases

En ocasiones, durante el desarrollo necesitamos acceder a los recursos de los archivos de propiedades de las clases de una aplicación para gestionar los mensajes de error de confirmación. Como hemos comentado anteriormente, todas las clases herederas de la clase `ActionSupport` pueden utilizar los archivos de propiedades. El acceso a las parejas de datos clave/valor se realiza a través de la función `getText(...)` estudiada con anterioridad.

Vamos a modificar nuestra clase `Cliente.java` con el fin de agregar un mensaje dinámico posterior a la autenticación del cliente. Para ello, utilizaremos una variable de clase denominada *mensaje* así como su getter asociado para acceder a su valor en las vistas JSP.

Código: ejemplo07.Cliente.java

```
package ejemplo07;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class Cliente extends ActionSupport {

    private String identificador;
    private String contrasena;
    private String mensaje;

      public Cliente()
      {

      }

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                 return "input";
            }
            // sin errores
            else
            {
                 // mensaje dinámico presente en el archivo
de propiedades
                  this.mensaje=getText("cliente.inicio");
                  return SUCCESS;
            }
      }
}
```

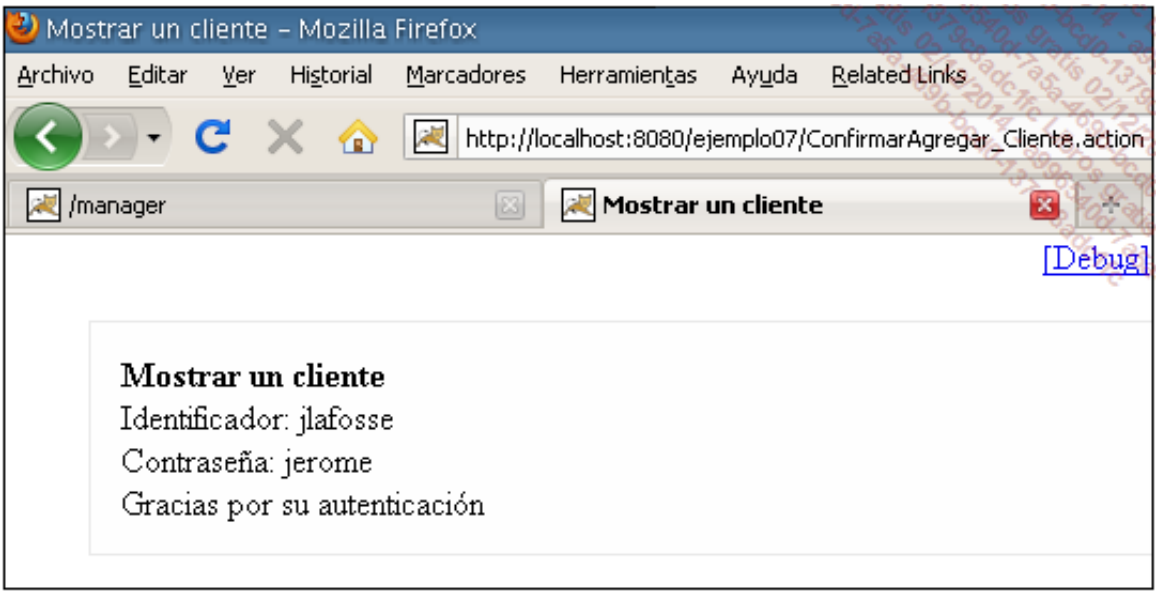
Código: ejemplo07.package.properties

```
...
cliente.inicio=Gracias por su autenticación
```

Código: /jsp/MostrarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
```

```
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
>: <s:property value="contrasena"/><br/>

            <s:property value="mensaje"/>
      </p>
</div>
</body>
</html>
```



Visualización de la información del cliente con acceso a los mensajes

En resumen

En este capítulo se ha presentado la gestión de mensajes de propiedades y el mecanismo de internacionalización de Struts. Las aplicaciones profesionales deben utilizar este mecanismo avanzado que permite un fácil mantenimiento y la evolución de un proyecto tanto monolingüe como multilingüe.

Presentación

La validación de datos permite mejorar la seguridad de un sistema, así como su solidez, mediante reglas y controles. Las validaciones permiten también conservar la coherencia y homogeneidad de dicho sistema, ya que cada dato u objeto presente en la base de datos se verifica antes de su inserción o modificación.

Las verificaciones y controles realizados en los campos de un sistema son complicados y representan una etapa de compleja implementación dentro de las aplicaciones Java EE. Para ello, Struts facilita un método de validación sencillo y eficaz basado en el framework de validación XWork.

Los validadores de Struts no requieren programación. Cada definición de una regla se configura en relación con una propiedad o un objeto en un archivo de texto en formato XML de fácil mantenimiento o con anotaciones Java. Asimismo, Struts propone, de acuerdo con sus estándares, el uso de reglas en los archivos de validación, la asociación de mensajes con los archivos de propiedades o bundles y las validaciones de la programación en las clases.

Aplicación

En Struts, se utilizan dos tipos de validadores:

- Los validadores de tipo campo o atributo. Estos validadores se utilizan en relación con los campos de formularios HTML y están asociados con una o varias reglas de validación.
- Los validadores de tipo condicional, que no están asociados con los campos de formularios pero permiten realizar comprobaciones de la programación en las clases de acción de las aplicaciones.

La aplicación de las validaciones en una aplicación de Struts se divide en tres etapas:

- Etapa 1: definición de la acción que debe verificar sus entradas.
- Etapa 2: creación del archivo de validación asociado. El nombre del archivo debe adaptarse al modelo siguiente: *NombreClaseAccion-validacion.xml*.
- Etapa 3: definición de la página a la que se deberá acudir en caso de error en la introducción de datos. Esta definición se realiza en el archivo de configuración *struts.xml* a través de la etiqueta `<result name="input"/>`.

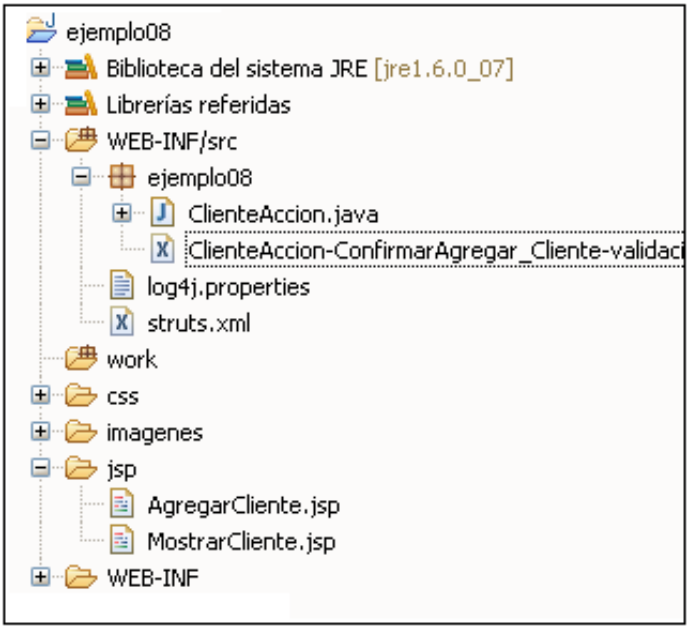
El nombre del archivo de configuración depende de las validaciones que deseemos llevar a cabo. El modelo presentado a continuación, con el nombre de la clase seguido del término validación, es el más frecuente (NombreClaseAccion_validacion.xml). No obstante, si deseamos realizar únicamente la verificación del método `agregarmodificar_Cliente()` de la clase de acción `ClienteAccion`, tendremos un archivo de tipo ClienteAccion-agregarmodificar_Cliente-validacion.xml.

Esta técnica permite así aplicar las validaciones a una acción concreta del controlador o utilizar un archivo de validación por ejemplo para crear un cliente y otro para realizar modificaciones con reglas diferentes. Los archivos de validación de Struts deben comenzar por la definición de la gramática adaptada. La etiqueta raíz del documento XML es `<validators/>`. Esta etiqueta puede contener etiquetas `<field/>` relacionadas con campos de formularios o etiquetas `<validator/>` para la definición de validadores asociados. La etiqueta `<field/>` contiene un atributo denominado `name` que establece el vínculo con el nombre de un campo de formulario HTML que debe validarse. Podemos realizar tantas validaciones de campos de formularios como deseemos.

La etiqueta `<field/>` contiene en sí misma una o varias etiquetas `<field-validator/>` asociadas con un atributo `type` que define el tipo de validación que se va a llevar a cabo (campo obligatorio, e-mail, número entero...).

También podemos transmitir parámetros a una etiqueta `<field/>` con el fin de precisar un valor o un espacio. Por último, la etiqueta `<message/>` presente en las etiquetas `<field-validator/>` permite precisar el mensaje que debe mostrarse en caso de error de validación.

Vamos a aplicar las validaciones a partir de nuestro formulario de cliente. El nuevo proyecto, denominado *ejemplo08* se basa en la aplicación *ejemplo07*. Cambiaremos el nombre de la clase de acción `Cliente` por `ClienteAccion` con el fin de respetar las convenciones de Struts.



Árbol del proyecto ejemplo08

Código: 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="true" />

    <package name="ejemplo08" namespace="/" extends="struts-default">
        <default-action-ref name="Agregar_Cliente" />

        <action name="Agregar_Cliente"
class="ejemplo08.ClienteAccion">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo08.ClienteAccion" method="agregar">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="SUCCESS">/jsp/MostrarCliente.jsp</result>
        </action>

    </package>
</struts>
```

Código: ejemplo08.ClienteAccion.java

```
package ejemplo08;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private String identificador;
    private String contrasena;
```

```
      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                  return "input";
            }
            // sin errores
            else
            {
                  return SUCCESS;
            }
       }
}
```

Código: /ejemplo08/ClienteAccion-ConfirmarAgregar_Cliente-validation.xml

```
<!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="identificador">
        <field-validator type="requiredstring">
            <message>El campo identificador es obligatorio</message>
        </field-validator>
    </field>
    <field name="contrasena">
        <field-validator type="requiredstring">
            <message>El campo contraseña es obligatorio</message>
        </field-validator>
    </field>
</validators>
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" id="contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
            <s:submit value="Agregar un cliente"/>
      </s:form>
</div>
</body>
</html>
```

Las definiciones configuradas son simples y se basan en los campos *identificador* y *contraseña* del formulario. La relación entre los campos del formulario y el archivo de validación se realiza a través de las etiquetas `<field name="identificador">` y `<field name="contrasena">`. En nuestro caso, verificamos únicamente si los campos están vacíos. La definición del enrutamiento se realiza en el

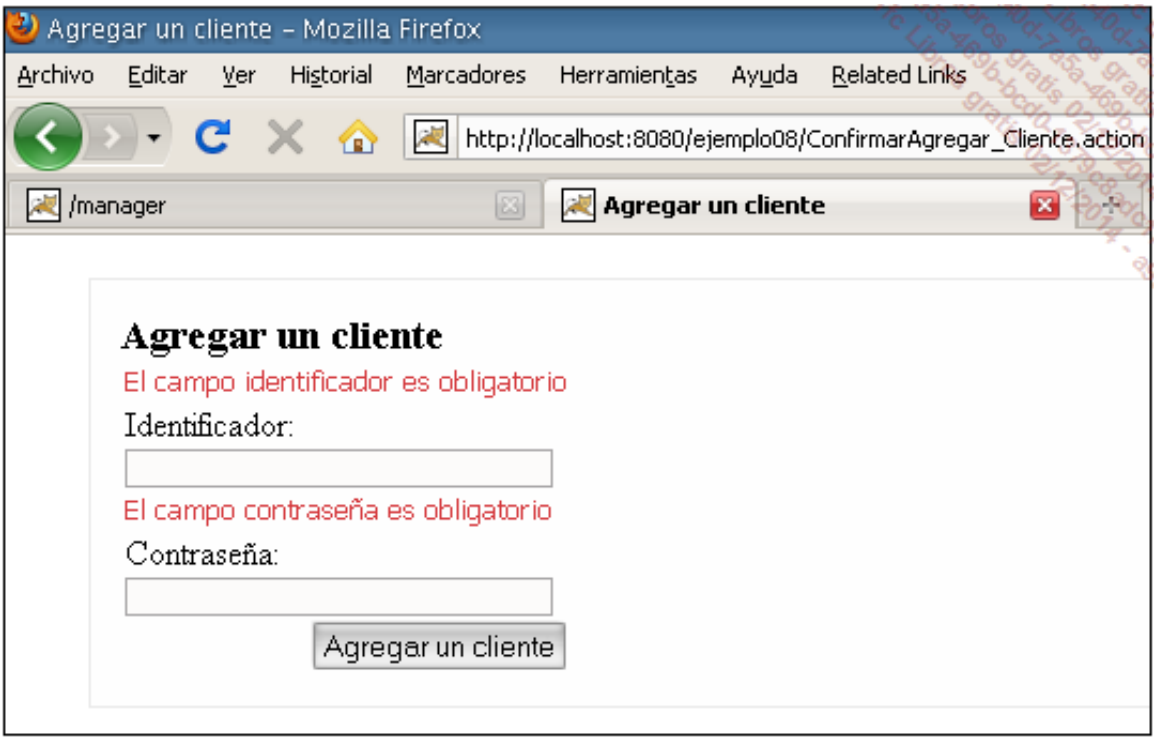
archivo *struts.xml* y permite precisar que en caso de error de validación, se acuda a la página precisada por el parámetro *input* sin perder datos.

```
<action name="ConfirmarAgregar_Cliente"
class="ejemplo08.ClienteAccion" method="agregar">
  <result name="input">/jsp/AgregarCliente.jsp</result>
  <result name="SUCCESS">/jsp/MostrarCliente.jsp</result>
</action>
```

Podemos definir la clase CSS `.errorMessage` propuesta por Struts con el fin de configurar un color adaptado (rojo) para mostrar los mensajes de error. Se agregará el estilo siguiente al archivo*/css/estilos.css*.

```
.errorMessage
{
      color:#D53A3E;
      font-family:tahoma, verdana, arial, sans-serif;
      font-size:13px;
}
```

Ahora, podemos intentar agregar una cuenta de cliente sin introducir el identificador ni la contraseña.



Formulario de adición de cliente y validación de datos introducidos

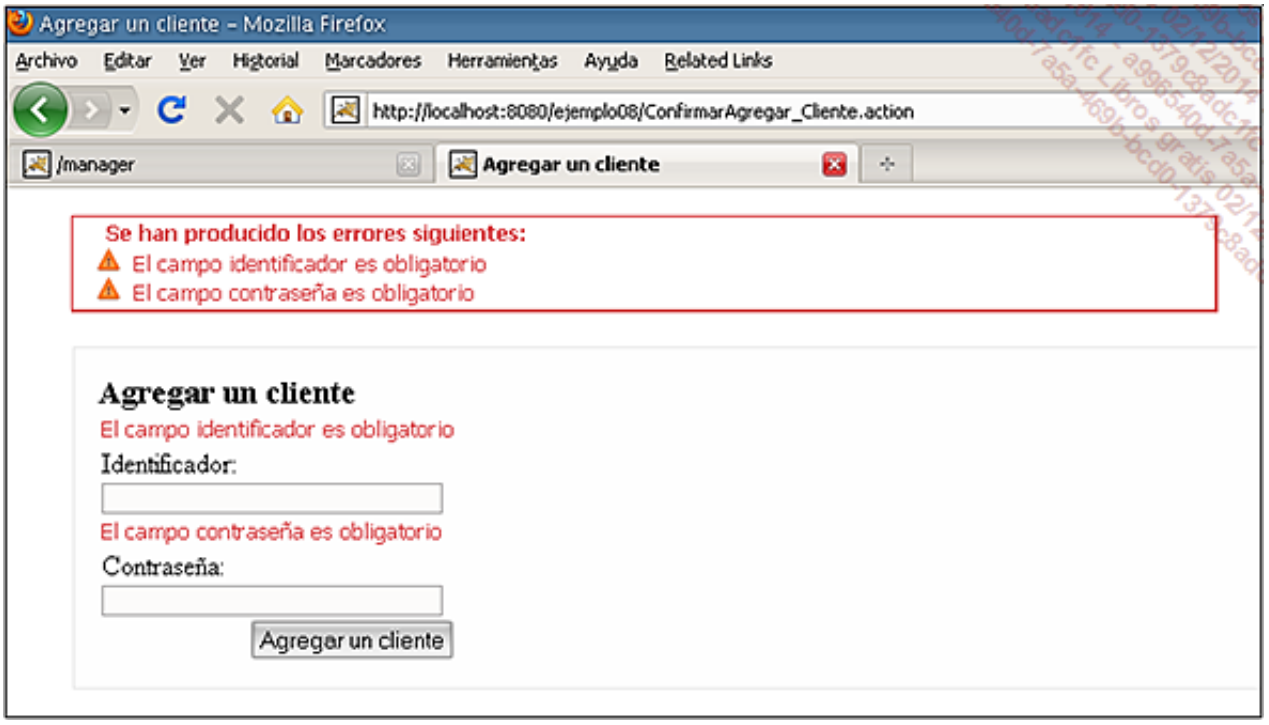
Ahora, podemos mejorar la visualización utilizando la etiqueta propuesta por Struts `<s:fielderror/>`con el fin de mostrar los mensajes de error en las vistas adaptadas. De esta forma, la etiqueta se añade con un estilo CSS adaptado para la presentación.

```
Código: /css/estilos.css
#mensaje_error
{
      margin-top:16px;
      margin-bottom:0px;
      margin-left:40px;
      padding-bottom:2px;
      padding-top:2px;
      padding-left:20px;
```

```
      padding-right:50px;
      text-align:justify;
      border-style:solid;
      border-top-width:1px;
      border-top-color:#D53A3E;
      border-right-width:2px;
      border-right-color:#D53A3E;
      border-bottom-width:2px;
      border-bottom-color:#D53A3E;
      border-left-width:1px;
      border-left-color:#D53A3E;
      width:600px;
}

#mensaje_error ul
{
      padding-top:0px;
      margin-top:2px;
      padding-bottom:0px;
      padding-right:20px;
      margin-bottom:0px;
      margin-left:8px;
      margin-right:10px;
      color:#D53A3E;
      font-family:tahoma, verdana, arial, sans-serif;
      font-size:13px;
      font-weight:normal;
      list-style-image:url(../imagenes/importante.gif);
}

#mensaje_error label
{
      color:#D53A3E;
      font-family:tahoma, verdana, arial, sans-serif;
      font-size:12px;
      font-weight:bold;
}
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los errores siguientes: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
...
</body>
</html>
```



Formulario de adición de cliente y hoja de estilos CSS

Validaciones

En el ejemplo anterior, hemos utilizado la validación de tipo *requirestring*, que corresponde a la verificación de la presencia de una cadena de caracteres que no esté vacía. Struts incluye varias validaciones estándar para facilitar la tarea: *required*, *requiredstring*, *int*, *date*, *expression*,*fieldexpression*, *e-mail*, *url*, *visitor*, *conversion*, *stringlenght* o *regexp*. A continuación, presentaremos cada una de estas validaciones con el fin de comprender su interés.

1. required

Esta validación permite verificar que un campo no tenga el valor *null*. No obstante, las cadenas de caracteres vacías no son nulas, están presentes pero vacías, algo diferente.

2. requiredstring

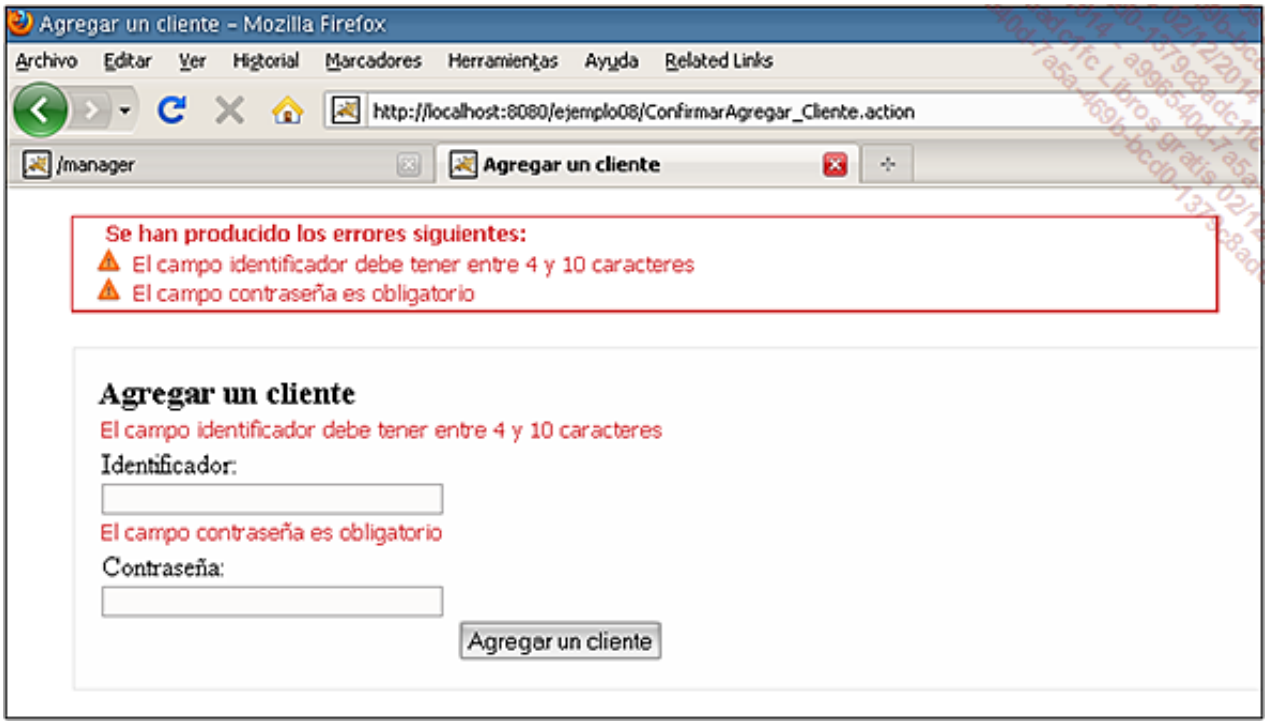
Esta validación permite comprobar que un campo no tenga el valor *null* y no esté vacío. Esta validación contiene un parámetro denominado *trim* que corresponde a los espacios situados antes y después de los datos.

```
<field-validator type="requiredstring">
   <param name="trim">true</param>
   <message>El campo identificador es obligatorio</message>
</field-validator>
```

3. stringlength

Esta validación permite validar la longitud de un campo que no esté vacío. Así, podemos especificar la longitud mínima y máxima de un campo de formulario. En nuestro ejemplo, el campo *identificador*debe tener un mínimo de 4 caracteres y un máximo de 10.

```
Código: /ejemplo08/ClienteAccion-ConfirmarAgregar_Cliente-validation.xml
<!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="identificador">
        <field-validator type="stringlength">
              <param name="minLength">4</param>
              <param name="maxLength">10</param>
            <message>El campo identificador debe tener entre 4 y 10
caracteres</message>
        </field-validator>
    </field>
    <field name="contrasena">
        <field-validator type="requiredstring">
            <message>El campo contraseña es obligatorio</message>
        </field-validator>
    </field>
</validators>
```



Formulario de cliente y validación de la longitud de los campos

4. int

Esta validación permite verificar si un campo puede convertirse en número entero y si se utilizan los parámetros *min* y *max*, además de verificar que los datos se han introducido en el espacio indicado.

```
<field-validator type="int">
    <param name="min">0</param>
    <param name="max">20</param>
    <message>El campo identificador debe estar comprendido entre 0
y 20</message>
</field-validator>
```

5. date

Esta validación permite verificar si un campo es de tipo *fecha* y se encuentra en un espacio determinado. En cambio, el formato de la fecha depende de la configuración regional actual de la aplicación. La validación siguiente permite verificar si la fecha introducida no es posterior al 31 de diciembre de 2000.

```
<field-validator type="date">
    <param name="max">31/12/2000</param>
    <message>El campo no debe ser posterior al
31/12/2000</message>
</field-validator>
```

6. e-mail

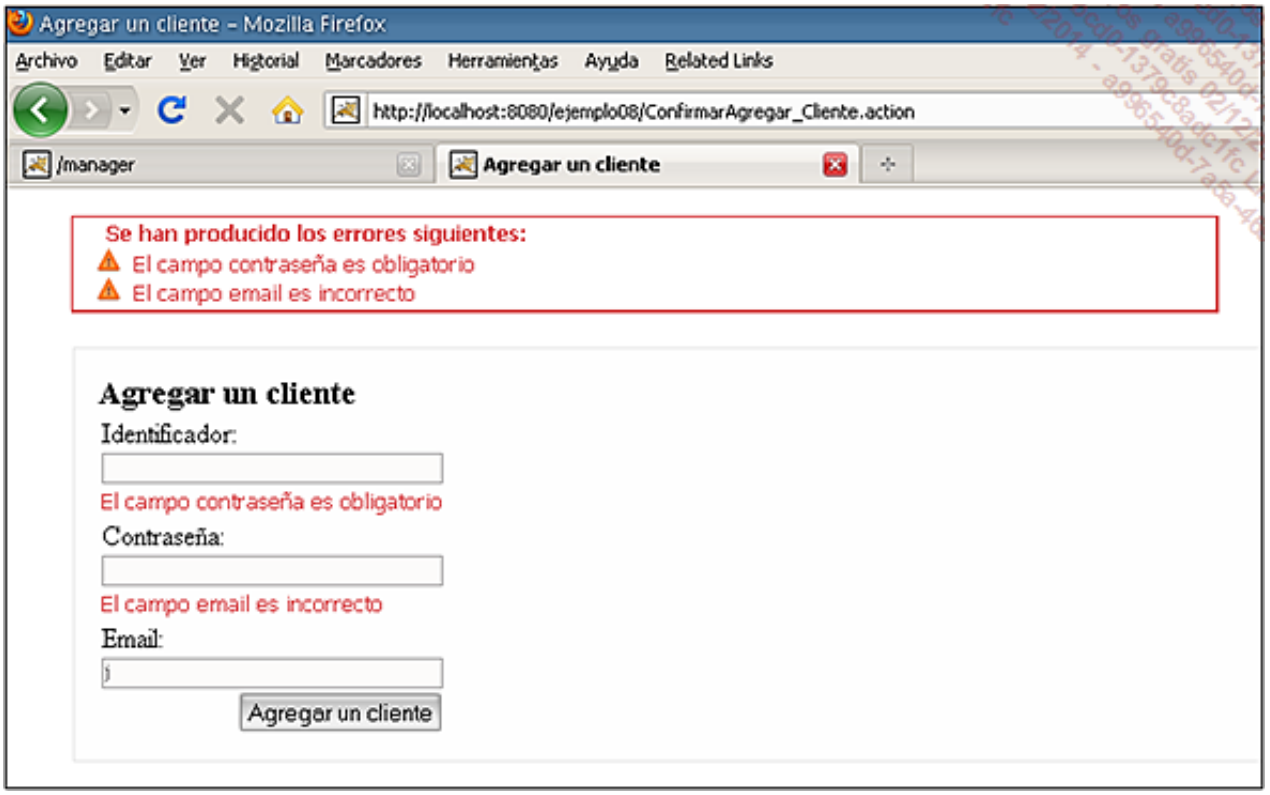
Esta validación permite verificar si un campo de tipo cadena de caracteres es una dirección de correo electrónico válida. Retomaremos nuestro *ejemplo08* y agregaremos un campo e-mail asociado a su validador.

```
Código: ejemplo08.ClienteAccion.java
package ejemplo08;
```

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

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private String identificador;
    private String contrasena;
    private String email;

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                  return "input";
            }
            // sin errores
            else
            {
                  return SUCCESS;
            }
      }
}
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los errores siguientes: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" id="contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
            <s:textfield name="email" id="email" label="Email"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar un cliente"/>
      </s:form>
</div>
</body>
</html>
```



Formulario de cliente y validación del e-mail

7. url

Esta validación permite verificar si un campo de tipo cadena de caracteres es una URL correcta. Este validador utiliza la clase `java.net.URL` para verificar la sintaxis.

```
<field-validator type="url">
    <message>El campo URL es incorrecto</message>
</field-validator>
```

8. regex

Esta validación permite especificar una expresión regular para verificar un campo de formulario. Este validador utiliza la clase `java.lang.regexp.Pattern` para verificar la sintaxis.

```
<field name="cliente.códigoPostal">
    <field-validator type="regex">
        <param name="expression"><![CDATA[^\d{5}$]]></param>
        <message>El campo debe ser un código postal con el formato NNNNN
    </field-validator>
</field>
```

9. fieldexpression y expression

Estas validaciones permiten utilizar expresiones OGNL para validar campos de formulario. Las validaciones de tipo *fieldexpression* utilizan dos parámetros, *min* y *max*, que permiten limitar los datos introducidos por el usuario.

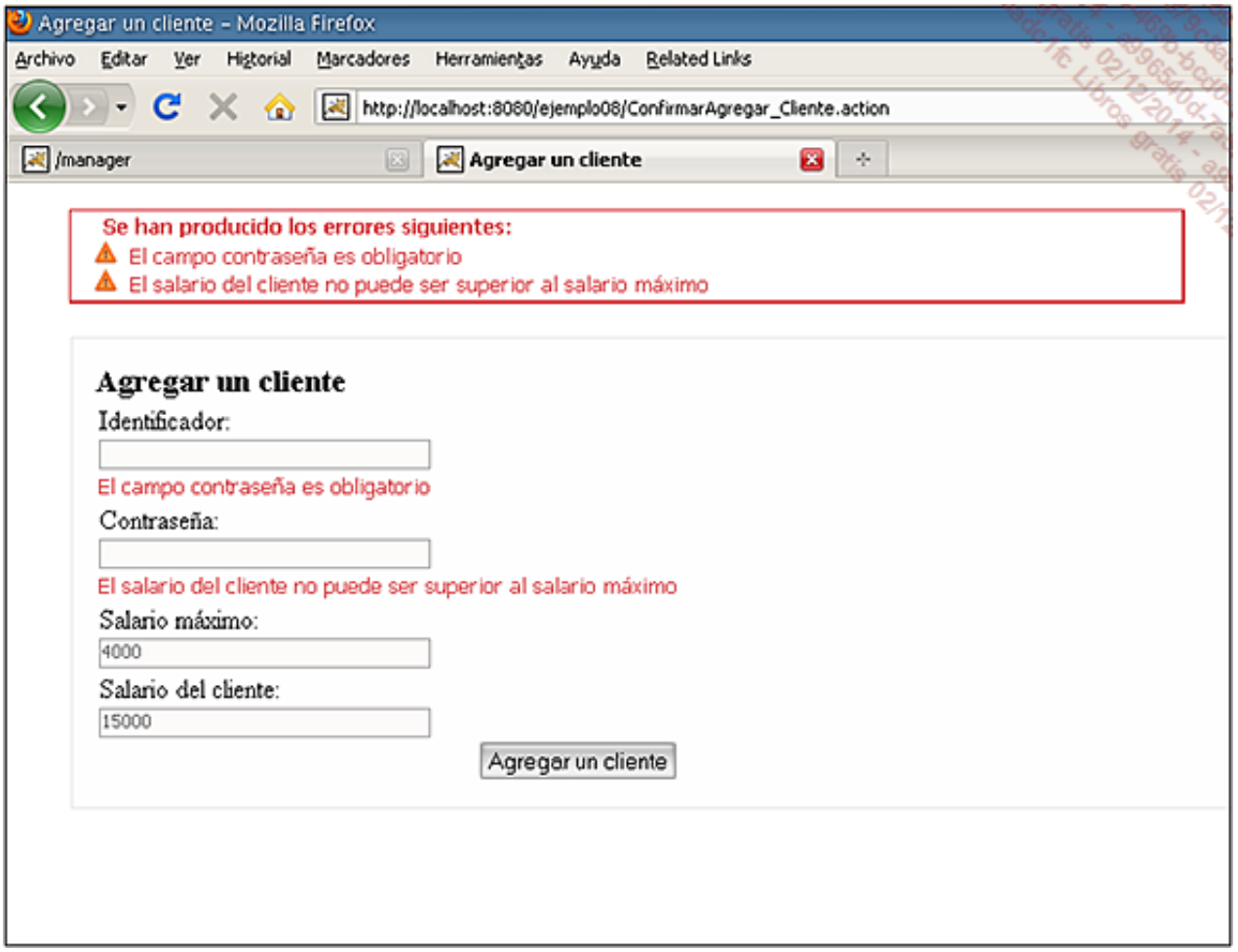
Mostraremos esta validación con la ayuda de dos nuevos campos para la gestión del salario del cliente.

```
Código: ejemplo08.ClienteAccion.java
```

```
package ejemplo08;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private String identificador;
    private String contrasena;
    private String email;
    private int salarioMáx=4000;
    private int salarioCliente;

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                  return "input";
            }
            // sin errores
            else
            {
                  return SUCCESS;
            }
      }
}
```

```
Código: /ejemplo08/ClienteAccion-ConfirmarAgregar_Cliente-validation.xml
<!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="identificador">
        <field-validator type="stringlength">
              <param name="minLength">4</param>
              <param name="maxLength">10</param>
            <message>El campo identificador debe tener entre 4 y 10
caracteres</message>
        </field-validator>
    </field>
    <field name="contrasena">
        <field-validator type="requiredstring">
            <message>El campo contraseña es obligatorio</message>
        </field-validator>
    </field>
    <field name="email">
        <field-validator type="email">
            <message>El campo email es incorrecto</message>
        </field-validator>
    </field>
    <field name="salarioCliente">
        <field-validator type="fieldexpression">
              <param name="fieldName">salarioMax</param>
              <param name="expression">
                     salarioMax > salarioCliente
              </param>
            <message>El salario del cliente no puede ser
superior al salario máximo</message>
```

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



Formulario de cliente y gestión de un espacio de validación

En la prueba, el salario máximo debe ser superior al salario del cliente (mínimo) para que no se produzca un error.

Las validaciones de tipo *expression* utilizan dos parámetros, *min* y *max*, que permiten definir una limitación a nivel de los datos introducidos por el usuario pero al contrario que la validación*fieldexpression*, desencadenan un error de acción (action error) y no un error de campo (field error). Durante la utilización de las validaciones de tipo expression, la etiqueta de Struts `<s:action/>`permite mostrar los mensajes de error.

10. conversion

Esta validación permite mostrar un error cuando una acción se encuentra con un error de conversión. Retomaremos nuestro ejemplo para agregar un campo y gestionar la edad del cliente en forma de número entero.

```
Código: ejemplo08.ClienteAccion.java
package ejemplo08;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {
```

```
    private String identificador;
    private String contrasena;
    private int edad;

      // getter y setter

      // agregar los datos del cliente a la sesión
      public String agregar()
      {
            // verificar los datos introducidos, en caso de error
volver a la página de introducción de datos
            if(this.identificador.equals("") ||
this.contrasena.equals(""))
            {
                  return "input";
            }
            // sin errores
            else
            {
                  return SUCCESS;
            }
      }
}
```

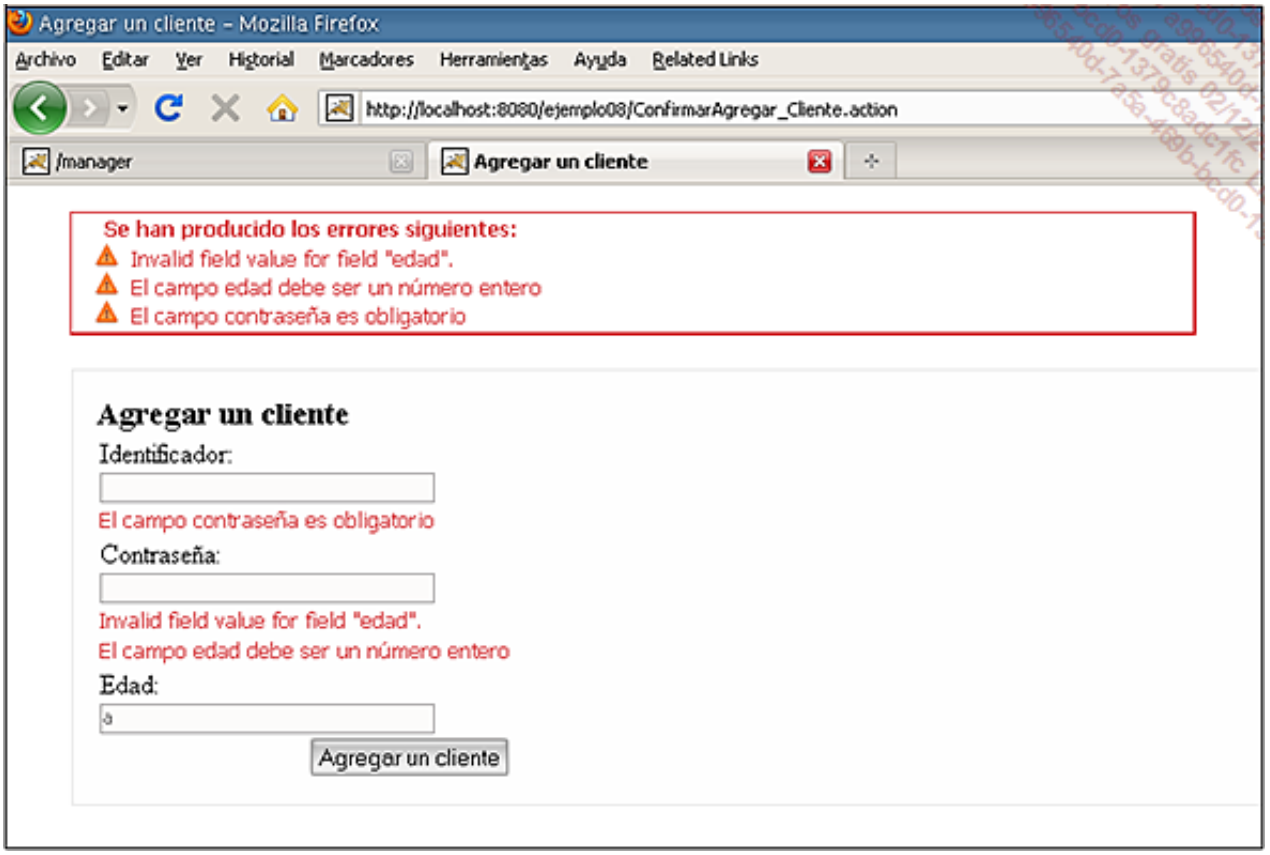
Código: /ejemplo08/ClienteAccion-ConfirmarAgregar_Cliente-validation.xml

```
<!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="identificador">
        <field-validator type="stringlength">
              <param name="minLength">4</param>
              <param name="maxLength">10</param>
            <message>El campo identificador debe tener entre 4 y 10
caracteres</message>
        </field-validator>
    </field>
    <field name="contrasena">
        <field-validator type="requiredstring">
            <message>El campo contraseña es obligatorio</message>
        </field-validator>
    </field>
    <field name="edad">
        <field-validator type="conversion">
            <message>El campo edad debe ser un número entero</message>
        </field-validator>
    </field>
</validators>
```

Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
       <label>Se han producido los errores siguientes: </label>
       <ul><s:fielderror/></ul>
```

```
</div>
</s:if>
<div id="carta">
      <h3>Agregar un cliente</h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" id="contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
            <s:textfield name="edad" id="edad" label="Edad"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar un cliente"/>
      </s:form>
</div>
</body>
</html>
```



Formulario de cliente y validación de la edad

Struts muestra por defecto el mensaje Invalid field value for field "nombredelcampo". Este mensaje se puede sobrecargar o modificar en el archivo de propiedades. Esta técnica se explica detalladamente en el capítulo Gestión de tipos y conversiones.

---

11. visitor

Esta validación permite mejorar el mantenimiento y reutilizar un sistema si deseamos compartir reglas de validación que serán usadas por diversos formularios. En nuestro ejemplo de formulario de creación de un cliente, utilizamos los campos *identificador* y *contrasena*. A continuación se asignarán reglas precisas a los campos para los formularios de creación y modificación.

Ahora, si deseamos aplicar esas mismas reglas al formulario de autenticación, que también contendrá los campos identificador y contraseña, podemos utilizar este validador.

Por ejemplo, la definición del identificador presente en nuestro archivo *ClienteAccion-ConfirmarAgregar_Cliente-validation.xml* es la siguiente:

```
<field name="identificador">
        <field-validator type="stringlength">
              <param name="minLength">4</param>
              <param name="maxLength">10</param>
            <message>El campo identificador debe tener entre 4 y 10
caracteres</message>
        </field-validator>
</field>
```

Ahora, si deseamos utilizar la misma validación para un campo identificador de otra acción, utilizaremos la definición siguiente:

```
<field name="identificador">
        <field-validator type="visitor">
            <message>El campo identificador debe tener entre 4 y 10
caracteres</message>
        </field-validator>
</field>
```

El vínculo entre las validaciones se realiza a través del atributo `name` de las etiquetas `<field/>`.

Aplicación de un ejemplo completo

En el ejemplo anterior, hemos utilizado una clase de acción `ClienteAccion`, así como todos sus descriptores de acceso, y en ocasiones una clase interna como `Profesion`. Esta técnica, aunque totalmente eficaz, no es demasiado rápida y requiere muchas líneas de código en la clase de acción.

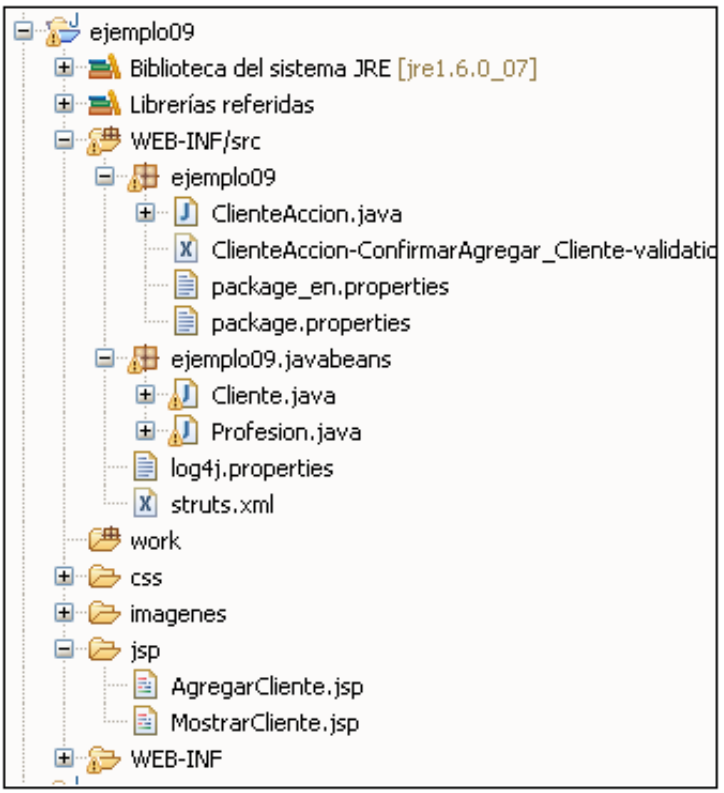
Vamos a utilizar un nuevo proyecto *ejemplo09* con una clase de acción `ClienteAccion` y dos JavaBeans, *Cliente.java* y *Profesion.java*. En la clase de acción `ClienteAccion`, no se declarará cada propiedad en conformidad con sus descriptores de acceso sino el propio objeto *cliente*. Por tanto sólo habrá un getter y un setter para trabajar con el objeto *cliente* y no existirán descriptores de acceso para cada propiedad. Además, las vistas y archivos de validación se adaptarán en consecuencia.

El archivo de propiedades también se utiliza para la gestión de los mensajes y los idiomas. El primer archivo *package.properties* contiene los mensajes informativos, así como los textos asociados a las validaciones. El segundo archivo *package_en.properties* es idéntico al primero pero contiene las traducciones de los mensajes. Este archivo de propiedades es avanzado y utiliza métodos de Struts con el fin de mostrar de manera dinámica el nombre de los campos concernidos por una validación con el método `getText(...)`. Por ejemplo, la línea siguiente permite mostrar el mensaje de error durante una validación estableciendo el vínculo con el nombre del campo de forma dinámica.

```
cliente.identificador=Identificador
campoobligatorio=El campo ${getText(fieldName)} es obligatorio
```

En caso de error en el campo *cliente.identificador*, el método `${getText(fieldName)}` sustituirá el campo asociado y mostrará el mensaje siguiente: *El campo Identificador es obligatorio*.

La aplicación permite agregar un nuevo cliente a partir de un identificador, una contrasena y la elección de la profesion en una lista dinámica. La clase de acción devuelve esta lista y el mecanismo de descriptor de acceso permite establecer el vínculo entre las diferentes propiedades de la aplicación. Por último, la página JSP */jsp/MostrarCliente.jsp* permite mostrar los datos introducidos por el cliente en la interfaz a través del acceso al objeto *cliente*.



Árbol del proyecto ejemplo09

```
Código: 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="true" />

    <package name="ejemplo09" namespace="/" extends="struts-default">
        <default-action-ref name="Agregar_Cliente" />

        <action name="Agregar_Cliente"
class="ejemplo09.ClienteAccion">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="ConfirmarAgregar_Cliente"
class="ejemplo09.ClienteAccion">
            <result name="input">/jsp/AgregarCliente.jsp</result>
            <result name="SUCCESS">/jsp/MostrarCliente.jsp</result>
        </action>

    </package>
</struts>
```

```
Código: ejemplo09.ClienteAccion.java
package ejemplo09;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo09.javabeans.Cliente;
import ejemplo09.javabeans.Profesion;


@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

      // objeto JavaBean
      private Cliente cliente;
      // lista de profesiones
      private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();

      public Cliente getCliente() {
            return cliente;
      }

      public void setCliente(Cliente cliente) {
            this.cliente = cliente;
      }

      public List<Profesion> getListaProfesiones() {
            listaProfesiones.add(new Profesion(0,""));
            listaProfesiones.add(new Profesion(1,
"Informático"));
            listaProfesiones.add(new Profesion(2,"Formador"));
            listaProfesiones.add(new Profesion(3,
"Administrador"));
            listaProfesiones.add(new Profesion(4,
"Programador"));
            return listaProfesiones;
      }
```

```
      public void setListaProfesiones(List<Profesion>
listaProfesiones) {
              this.listaProfesiones = listaProfesiones;
      }
}
```

Código: ejemplo09.Cliente.java

```
package ejemplo09.javabeans;

@SuppressWarnings("serial")
public class Cliente {

    private String identificador;
    private String contrasena;
    private Profesion profesion;


      public Cliente() {

      }

      // getter y setter
}
```

Código: ejemplo09.Profesion.java

```
package ejemplo09.javabeans;

@SuppressWarnings("serial")
public class Profesion {

    private int idProfesion;
    private String nombre;

    public Profesion() {

    }
    public Profesion(int idProfesion, String nombre) {
      this.idProfesion=idProfesion;
      this.nombre=nombre;
    }

    // getter y setter
}
```

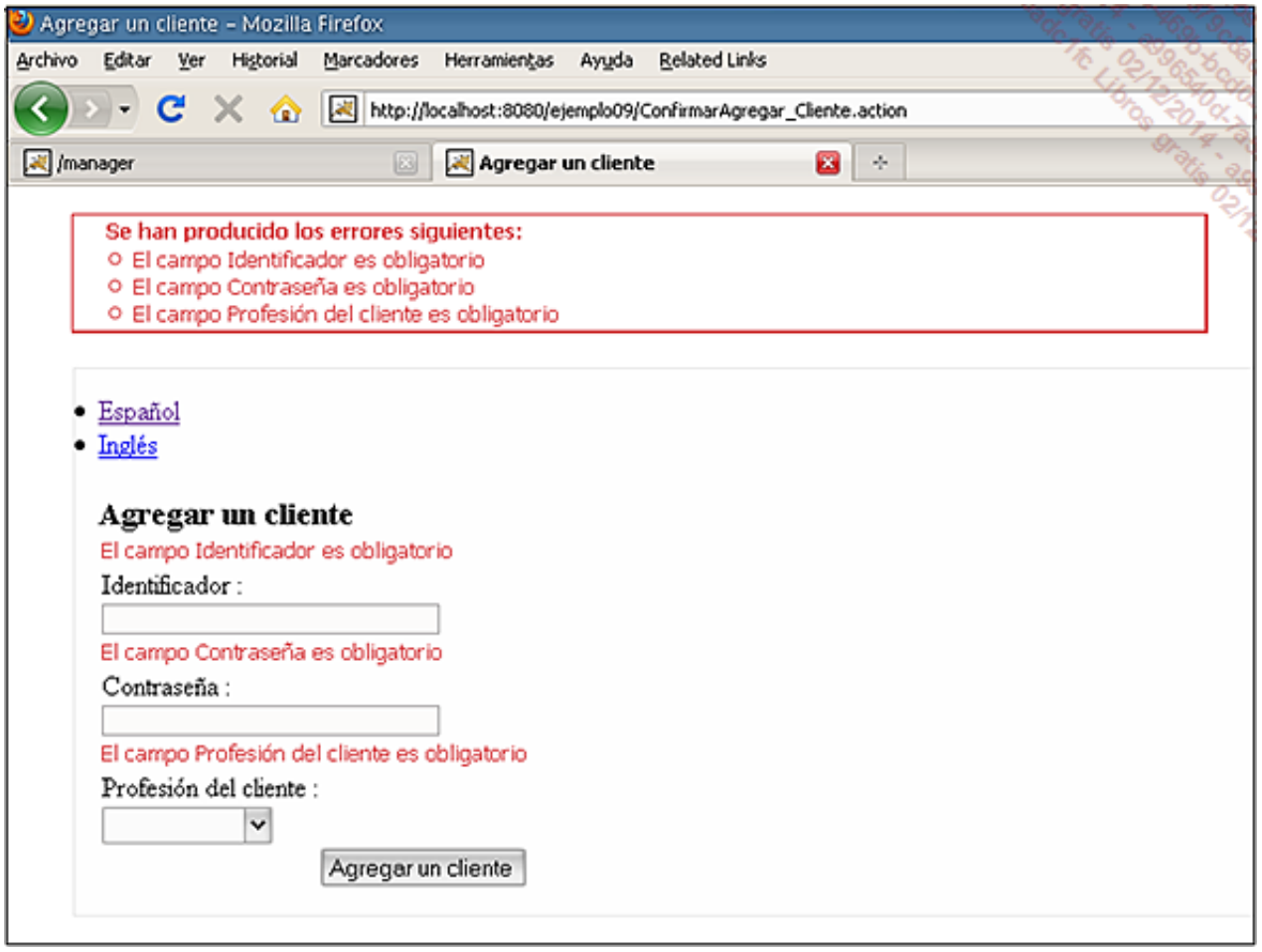
Código: /jsp/AgregarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>
<s:property value="%{getText('cliente.agregar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label><s:property value="%{getText('error')}"/></label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
```

```
<div id="carta">
      <ul>
            <li><a href="Agregar_Cliente.action?
request_locale=es">Español</a></li>
            <li><a href="Agregar_Cliente.action?
request_locale=en">Ingl&eacute;s</a></li>
      </ul>
      <br/>
      <h3><s:property value="%{getText('cliente.agregar')}"/></h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="cliente.identificador"
id="cliente.identificador" label="%{getText('cliente.identificador')}"
labelposition="top" cssClass="input"/>
            <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="%{getText('cliente.contrasena')}"
labelposition="top" cssClass="input"/>
            <s:select name="cliente.profesion.idProfesion"
id="cliente.profesion.idProfesion" label="%
{getText('cliente.profesion.idProfesion')}" labelposition="top"
list="listaProfesiones" listKey="idProfesion" listValue="nombre"/>
            <s:submit value="%{getText('cliente.agregar')}"/>
      </s:form>
</div>
</body>
</html>
```

Código: /jsp/MostrarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>
<s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="cliente.identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
>: <s:property value="cliente.contrasena"/><br/>
            <s:property value="%
{getText('cliente.profesion.idProfesion')}"/>: <s:property
value="cliente.profesion.idProfesion"/><br/>
      </p>
</div>
</body>
</html>
```



Formulario de cliente con validaciones

Gestión de mensajes de error y SUCCESS

Los ejemplos presentados en las validaciones y la internacionalización utilizan colecciones para mostrar los mensajes de error durante las validaciones. Struts gestiona estos mensajes de manera dinámica y éstos permiten mostrar textos independientes o insertados en los archivos de propiedades.

En Struts, existen tres tipos de mensajes asociados a colecciones de datos:

- *actionErrors* contiene la lista de mensajes de error correspondiente al tratamiento de las acciones en asociación con la variable *errorMessages*.
- *fieldErrors* contiene la lista de mensajes de error correspondiente al tratamiento de las validaciones de los campos de formularios en asociación con la variable *errors*.
- *actionMessages* contiene la lista de mensajes de confirmación correspondiente al tratamiento.

Así, podemos gestionar estos tres tipos de mensajes con los archivos de validación XML, pero también en forma de software, con los métodos de programación.

Para agregar un error de acción *actionErrors*, se debe agregar el método siguiente a las clases de acción:

```
addActionError("Texto para justificar el error");
```

Ahora, es necesario utilizar la etiqueta de Struts adaptada para mostrar los errores de acción en la vista:

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

Para agregar un error de validación *fieldErrors*, se debe agregar el método siguiente a las clases de acción:

```
addFieldError("campo", "Texto para justificar el error de
validación");
```

Ahora, es necesario utilizar la etiqueta de Struts adaptada para mostrar los errores de acción en la vista:

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

Para agregar un mensaje de confirmación *actionMessages*, se debe agregar el método siguiente a las clases de acción:

```
addActionMessage("Texto para justificar el SUCCESS");
```

Ahora, es necesario utilizar la etiqueta de Struts adaptada para mostrar los mensajes de confirmación en la vista:

```
<s:actionmessage/>
```

A continuación, mejoraremos nuestro proyecto *ejemplo09* para comprobar si la creación del cliente se corresponde con el identificador *jlafosse*. De no ser así, se agregará un mensaje de error dinámico, además del mensaje del campo *cliente.identificador*, o si no se mostrará un mensaje de SUCCESS. Las vistas se modifican para mostrar los mensajes adaptados.

La clase de acción `ClienteAccion` cuenta con un nuevo método `verificar()` configurado en el archivo de configuración *struts.xml*.

Código: struts.xml

```
...
<action name="ConfirmarAgregar_Cliente"
class="ejemplo09.ClienteAccion" method="verificar">
```

```
    <result name="input">/jsp/AgregarCliente.jsp</result>
    <result name="success">/jsp/MostrarCliente.jsp</result>
</action>
...
```

Código: ejemplo09.ClienteAccion.java

```
package ejemplo09;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo09.javabeans.Cliente;
import ejemplo09.javabeans.Profesion;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

      ...

      // método requerido tras la verificación de los
datos introducidos por parte del validador
      public String verificar()
      {
            // verificar si el identificador es correcto
            if(!this.cliente.getIdentificador().equals("jlafosse"))
            {
                  addFieldError("cliente.identificador",
getText("campologin"));
                  addActionError(getText("erroraccion.login"));
                  return "input";
            }
            // sin errores
            else
            {
                  addActionMessage(getText("correcto"));
                  return SUCCESS;
            }
      }
}
```

Código: /jsp/AgregarCliente.jsp

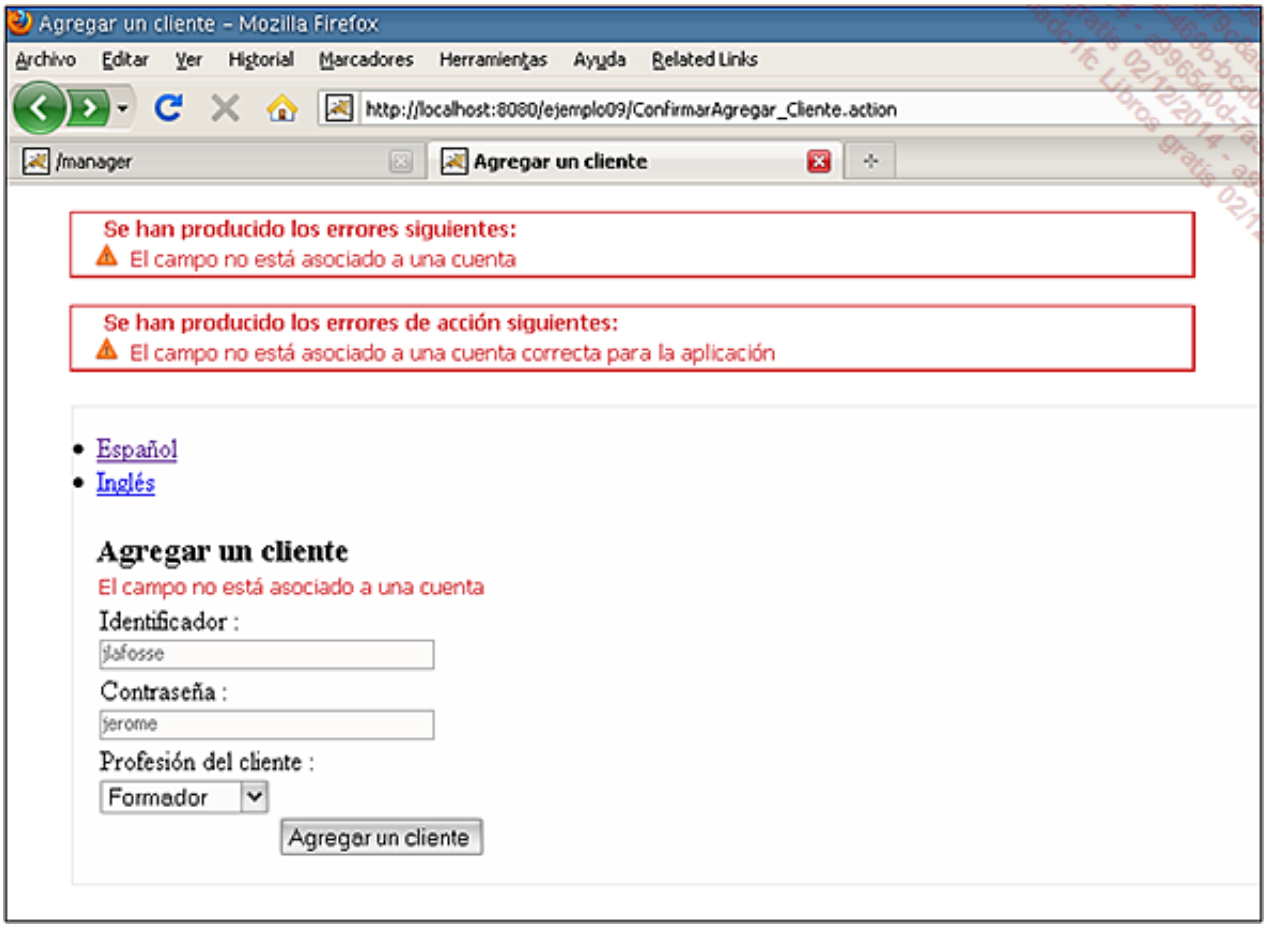
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%getText('cliente.agregar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label><s:property value="%{getText('error')}"/></label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<!-- Mensaje de error durante acciones -->
<s:if test="errorMessages.size()>0">
<div id="mensaje_error">
      <label><s:property value="%
{getText('erroraccion')}"/></label>
      <ul><s:actionerror/></ul>
</div>
</s:if>
```

```
<div id="carta">
      <ul>
            <li><a href="Agregar_Cliente.action?
request_locale=es">Espa&ntilde;ol</a></li>
            <li><a href="Agregar_Cliente.action?
request_locale=en">Ingl&eacute;s</a></li>
      </ul>
      <br/>

      <h3><s:property value="%{getText('cliente.agregar')}"/></h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="cliente.identificador"
id="cliente.identificador" label="%{getText('cliente.identificador')}"
labelposition="top" cssClass="input"/>
            <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="%{getText('cliente.contrasena')}"
labelposition="top" cssClass="input"/>
            <s:select name="cliente.profesion.idProfesion"
id="cliente.profesion.idProfesion" label="%
{getText('cliente.profesion.idProfesion')}" labelposition="top"
list="listaProfesiones" listKey="idProfesion" listValue="nombre"/>
            <s:submit value="%{getText('cliente.agregar')}"/>
      </s:form>
</div>
</body>
</html>
```

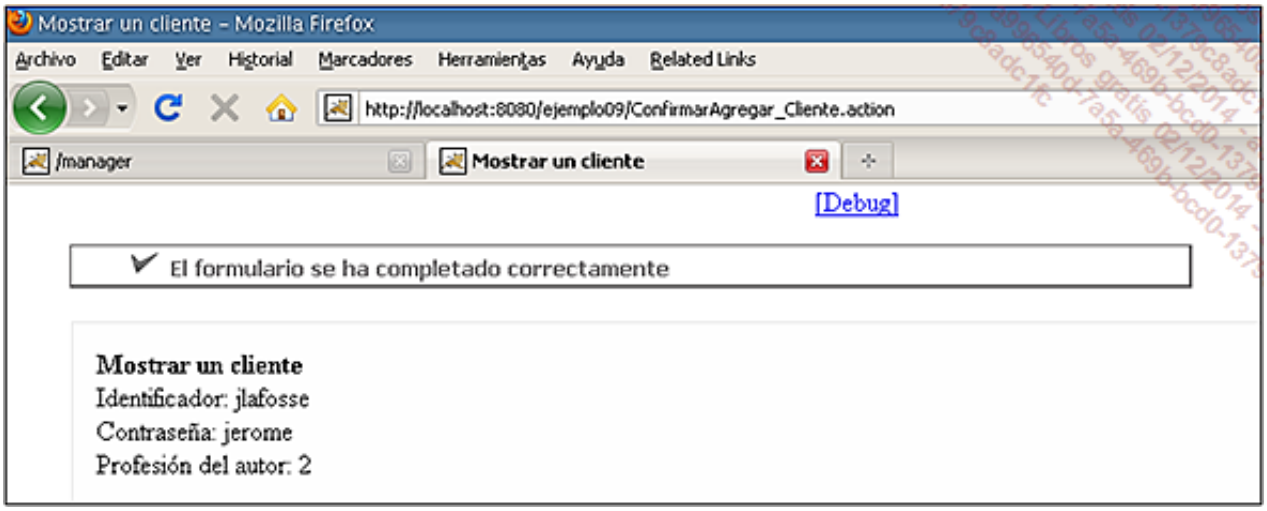
```
Código: /jsp/MostrarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<!-- Mensaje de confirmación -->
<s:if test="actionMessages.size()>0">
      <div id="mensaje_informacion">
            <ul><s:actionmessage/></ul>
      </div>
</s:if>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="cliente.identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
>: <s:property value="cliente.contrasena"/><br/>
            <s:property value="%
{getText('cliente.profesion.idProfesion')}"/>: <s:property
value="cliente.profesion.idProfesion"/><br/>
      </p>
</div>
</body>
</html>
```

```
Código: package.properties
cliente.agregar=Agregar un cliente
cliente.identificador=Identificador
cliente.contrasena=Contraseña
```

```
cliente.profesion.idProfesion=Profesión del cliente
cliente.mostrar=Mostrar un cliente

error=Se han producido los errores siguientes:
erroraccion=Se han producido los errores de acción siguientes:
campoobligatorio=El campo ${getText(fieldName)} es obligatorio
campominimo=El campo ${getText(fieldName)} debe tener entre $
{getText(minLength)} y ${getText(maxLength)} caracteres
campologin=El campo no está asociado a una cuenta
erroraccion.login=El campo no está asociado a una cuenta correcta
para la aplicación
correcto=El formulario se ha completado correctamente
```



Formulario de cliente y validaciones avanzadas multilingües

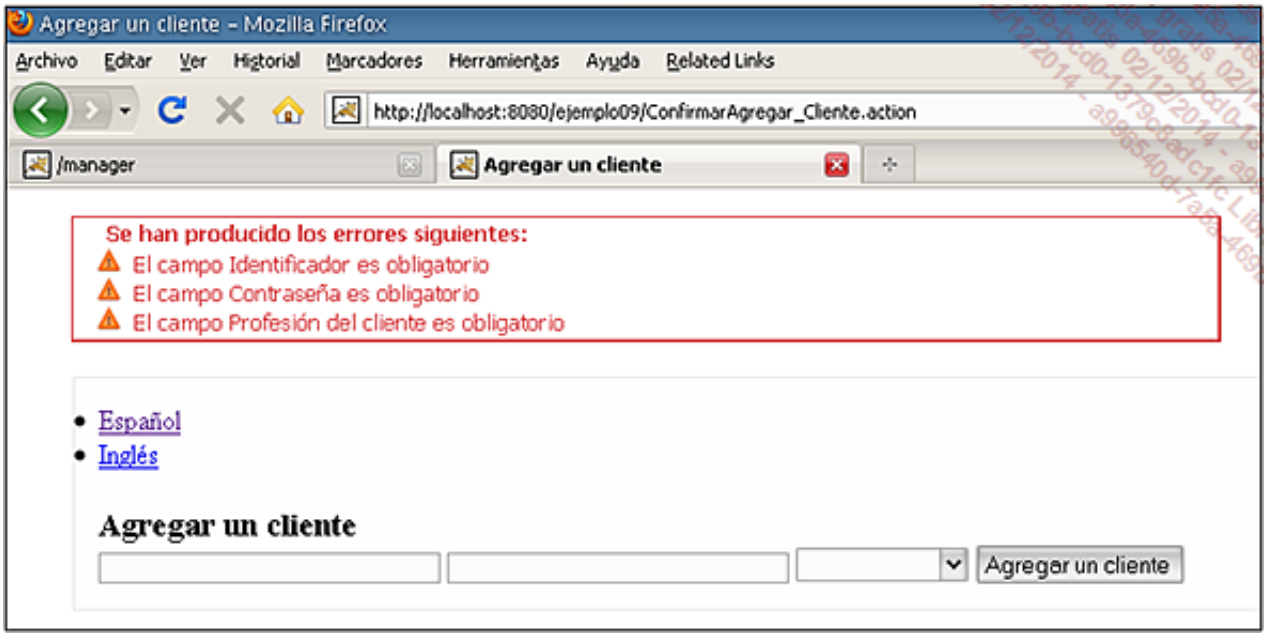


Visualización del cliente tras la validación

Los mensajes de error se muestran a través de la etiqueta `<s:fielderror/>` en el cuadro situado en la parte superior de la página, pero también aparecen automáticamente junto a cada campo del formulario. Esta presentación es gestionada por el tema predeterminado de la página xhtml.

En nuestro formulario, el tema no se ha precisado, por lo que se utiliza el tema predeterminado,*xhtml*. Con este modo de visualización en forma de tabla HTML, los mensajes de error se asocian automáticamente a los campos. Para desactivar este servicio, podemos cambiar el tema y utilizar el tema simple, que no aplica ningún tipo de formato. Podemos precisar este tema en la etiqueta`<s:form/>` de nuestra página JSP *AgregarCliente.jsp*.

```
<s:form method="post" action="ConfirmarAgregar_Cliente"
theme="simple">
```



Formulario de cliente y tema simple

Escribir un validador

Los validadores que incluye Struts son suficientes para la mayoría de las aplicaciones Web. No obstante, en ocasiones es necesario desarrollar validadores específicos en función de una necesidad particular. Los validadores de Struts implementan la interfaz *Validator* definida en el paquete*com.opensymphony.xwork2.validator*.

Struts utiliza también un interceptor para la gestión de las validaciones. El validador se ocupa de cargar y ejecutar los validadores. El interceptor, que para recordárselo funciona como un filtro, desencadena el método `setValidatorContext()` para asignar el validador en el contexto adaptado. En un segundo momento, el interceptor desencadena el método `validate(Object objetavalider)` con el objeto que debe validarse como parámetro. Este método es responsable del tratamiento de la validación y es el que debemos sobrecargar para llevar a cabo nuestras validaciones.

1. La interfaz Validator y las clases ValidatorSupport y FieldValidatorSupport

Para poder sobrecargar el método `validate(Object objetoavalidar)`, es más sencillo heredar de la clase `ValidatorSupport` o `FieldValidatorSupport` que implementar la interfaz `Validator`. La clase`ValidatorSupport` se utiliza para definir validaciones complejas (de tipo acción) y la clase`FieldValidator` se hereda para las validaciones de los campos de formularios. Si nuestro validador debe utilizar parámetros, el principio es idéntico al de las clases de acción. En realidad, basta con declarar la variable en la clase validador y definir su getter y setter.

La clase `ValidatorSupport` incluye varios métodos para la gestión de las validaciones:

- `getFieldValue(String nombre, Object objeto)`: permite devolver el valor de un campo especificado.
- `addActionError(Object actionError)`: permite agregar un error de tipo acción.
- `addFieldError(String nombre, Object objeto)`: permite agregar un error de campo.

2. Declarar los validadores

Los validadores se declaran de manera estándar en el archivo *default.xml* presente en la clase siguiente: `com.opensymphony.xwork2.validator.validators` que se encuentra en el archivo*xwork.jar*. Durante la creación de nuevos validadores, es necesario declarar los mismos en un archivo *validators.xml* presente en el directorio */WEB-INF/clases* o */WEB-INF/src* de la aplicación.

A continuación se muestra el contenido del archivo*com.opensymphony.xwork2.validator.validators.default.xml* incluido con Struts:

```
<?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">

<!-- START SNIPPET: validators-default -->
<validators>
   <validator name="required"
class="com.opensymphony.xwork2.validator.validators.RequiredField
Validator"/>
   <validator name="requiredstring"
class="com.opensymphony.xwork2.validator.validators.RequiredString
Validator"/>
   <validator name="int"
class="com.opensymphony.xwork2.validator.validators.IntRangeField
Validator"/>
```

```
   <validator name="long"
class="com.opensymphony.xwork2.validator.validators.LongRangeField
Validator"/>
   <validator name="short"
class="com.opensymphony.xwork2.validator.validators.ShortRangeField
Validator"/>
   <validator name="double"
class="com.opensymphony.xwork2.validator.validators.DoubleRangeField
Validator"/>
   <validator name="date"
class="com.opensymphony.xwork2.validator.validators.DateRangeField
Validator"/>
   <validator name="expression"
class="com.opensymphony.xwork2.validator.validators.Expression
Validator"/>
   <validator name="fieldexpression"
class="com.opensymphony.xwork2.validator.validators.FieldExpression
Validator"/>
   <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.VisitorField
Validator"/>
   <validator name="conversion"
class="com.opensymphony.xwork2.validator.validators.ConversionError
FieldValidator"/>
   <validator name="stringlength"
class="com.opensymphony.xwork2.validator.validators.StringLength
FieldValidator"/>
   <validator name="regex"
class="com.opensymphony.xwork2.validator.validators.RegexField
Validator"/>
   <validator name="conditionalvisitor"
class="com.opensymphony.xwork2.validator.validators.Conditional
VisitorFieldValidator"/>
</validators>
<!--  END SNIPPET: validators-default -->
```

3. Aplicación

Vamos a utilizar un nuevo proyecto *ejemplo10* basado en el ejemplo anterior para configurar un validador. Definiremos un validador complejo para el identificador del cliente. Este identificador debe tener una longitud mínima, una longitud máxima y no puede estar ya en uso. Para ello, el validador verificará si el identificador no se encuentra en una lista. Por supuesto, en una aplicación profesional, esta lista podría encontrarse en una base de datos en vez de formar parte del código de manera permanente.

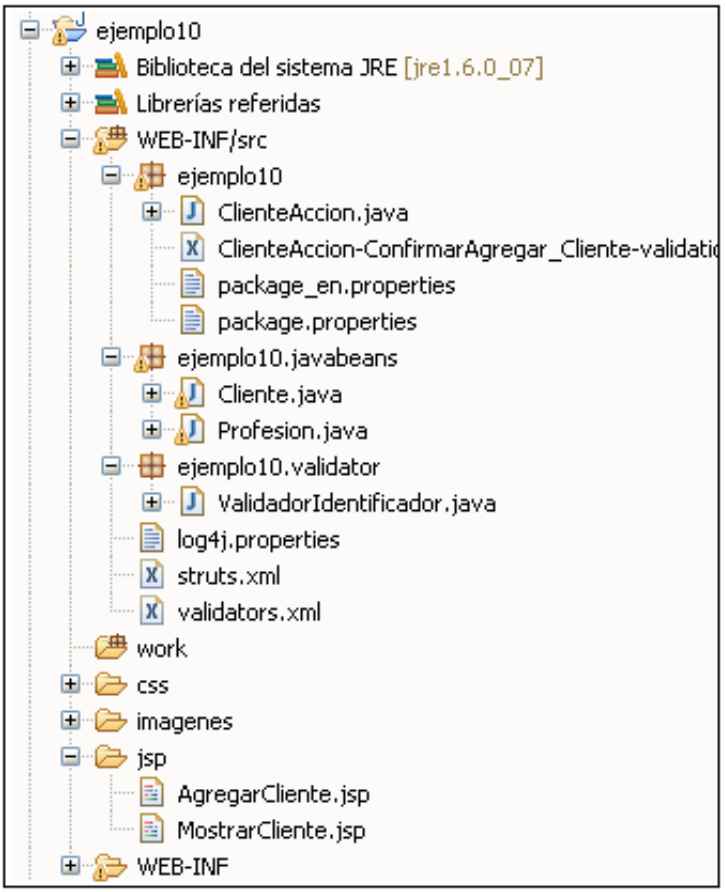
Empezaremos creando una clase denominada `ValidadorIdentificador` heredera de la clase`FieldValidatorSupport`. Con el fin de conservar un sistema homogéneo, vamos a crear un paquete dedicado a los validadores llamado *ejemplo10.validator*.

El archivo de validación *validators.xml* se encuentra en el directorio */WEB-INF/src* de la aplicación y contiene la declaración del validador. Ahora que hemos guardado el validador con el nombre*validadoridentificador*, podemos utilizarlo en los archivos de validación del mismo modo que cualquier validador.

```
Código: /WEB-INF/src/validators.xml
<?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="validadoridentificador"
        class="ejemplo10.validator.ValidadorIdentificador"/>
</validators>
```

La clase de validación `ValidadorIdentificador` contiene dos parámetros denominados *minLength* y*maxLength* que serán transmitidos por el archivo de validación *ClienteAccion-ConfirmarAgregar_Cliente-validation.xml* con las etiquetas `<param name="minLength"/>` y `<param name="maxLength"/>`. A continuación, el método `validate(Object objetoavalidar)` desencadenado en cada validación, verificará si el identificador se encuentra en la lista siguiente: *jlafosse*, *asoto*,*crenato*, *amartín*. Estos identificadores no podrán utilizarse para validar el formulario.



Árbol del proyecto ejemplo10

```
Código: /ejemplo10/ClienteAccion-ConfirmarAgregar_Cliente-validation.xml
<!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="cliente.identificador">
        <field-validator type="requiredstring">
            <message key="campoobligatorio"/>
        </field-validator>
    </field>
    <field name="cliente.identificador">
        <field-validator type="validadoridentificador">
              <param name="minLength">4</param>
              <param name="maxLength">20</param>
            <message key="validadoridentificador"/>
        </field-validator>
    </field>
```

```
    <field name="cliente.contrasena">
        <field-validator type="requiredstring">
            <message key="campoobligatorio"/>
        </field-validator>
    </field>
    <field name="cliente.profesion.idProfesion">
        <field-validator type="int">
              <param name="min">1</param>
            <message key="campoobligatorio"/>
        </field-validator>
    </field>
</validators>
```

Código: ejemplo10.validator.ValidadorIdentificador

```
package ejemplo10.validator;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.validator.ValidationException;
import com.opensymphony.xwork2.validator.validators.FieldValidatorSupport;

public class ValidadorIdentificador extends FieldValidatorSupport{

      private int minLength=0;
      private int maxLength=0;

      public int getMinLength() {
             return minLength;
      }

      public void setMinLength(int minLength) {
             this.minLength = minLength;
      }

      public int getMaxLength() {
             return maxLength;
      }

      public void setMaxLength(int maxLength) {
             this.maxLength = maxLength;
      }

      // validación a realizar en el campo
      public void validate(Object objeto) throws ValidationException
      {
             String nombreCampo=this.getFieldName();
             String
valorCampo=(String)this.getFieldValue(nombreCampo, objeto);

             // verificar la longitud mínima del identificador
             if(valorCampo.length() < this.minLength)
             {
                  addFieldError(nombreCampo,objeto);
                  return;
             }
             if(valorCampo.length() > this.maxLength)
             {
                  addFieldError(nombreCampo,objeto);
                  return;
             }
             // verificar que el identificador sólo contiene estos
caracteres
             if(isUtilizaIdentificador(valorCampo))
             {
```

```
                addFieldError(nombreCampo,objeto);
                return;
            }
      }

      // verificar si el identificador no está utilizado
      public boolean isUtilizaIdentificador(String identificador)
      {
            List<String> listaIdentificadores=new
ArrayList<String>();
            listaIdentificadores.add("jlafosse");
            listaIdentificadores.add("asoto");
            listaIdentificadores.add("crenato");
            listaIdentificadores.add("amartín");

            if(listaIdentificadores.contains(identificador))
            {
                System.out.println(identificador+" está en la
lista");
                return true;
            }
            System.out.println(identificador+" no está en la
lista");
            return false;
      }
}
```

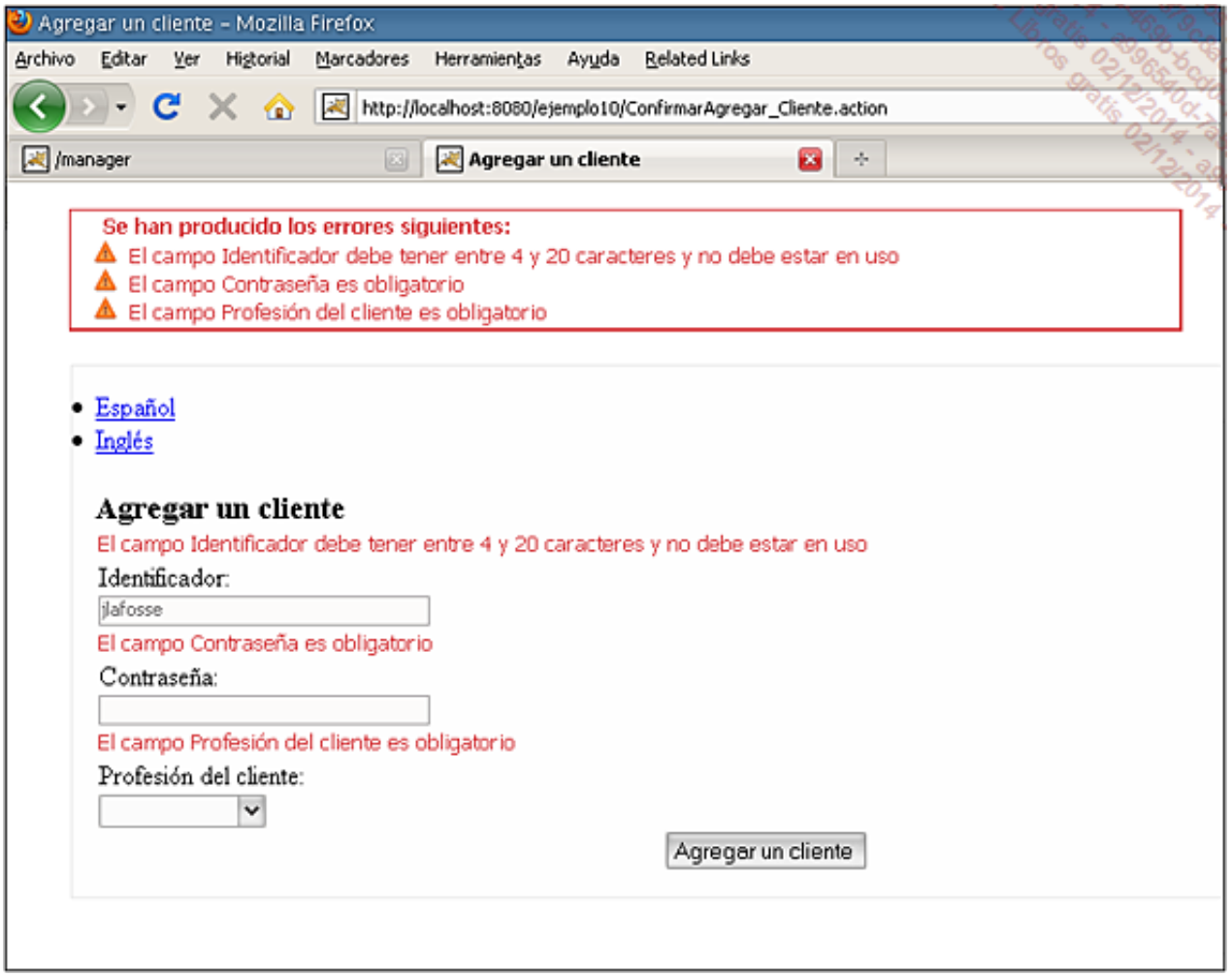
Código: ejemplo10.ClienteAccion.java

```
package ejemplo10;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo10.javabeans.Cliente;
import ejemplo10.javabeans.Profesion;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

      // objeto JavaBean
      private Cliente cliente;
      // lista de profesiones
      private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();

      // getter y setter


      // método para agregar un mensaje de confirmación
      public String verificar()
      {
          addActionMessage(getText("correcto"));
          return SUCCESS;
      }
}
```

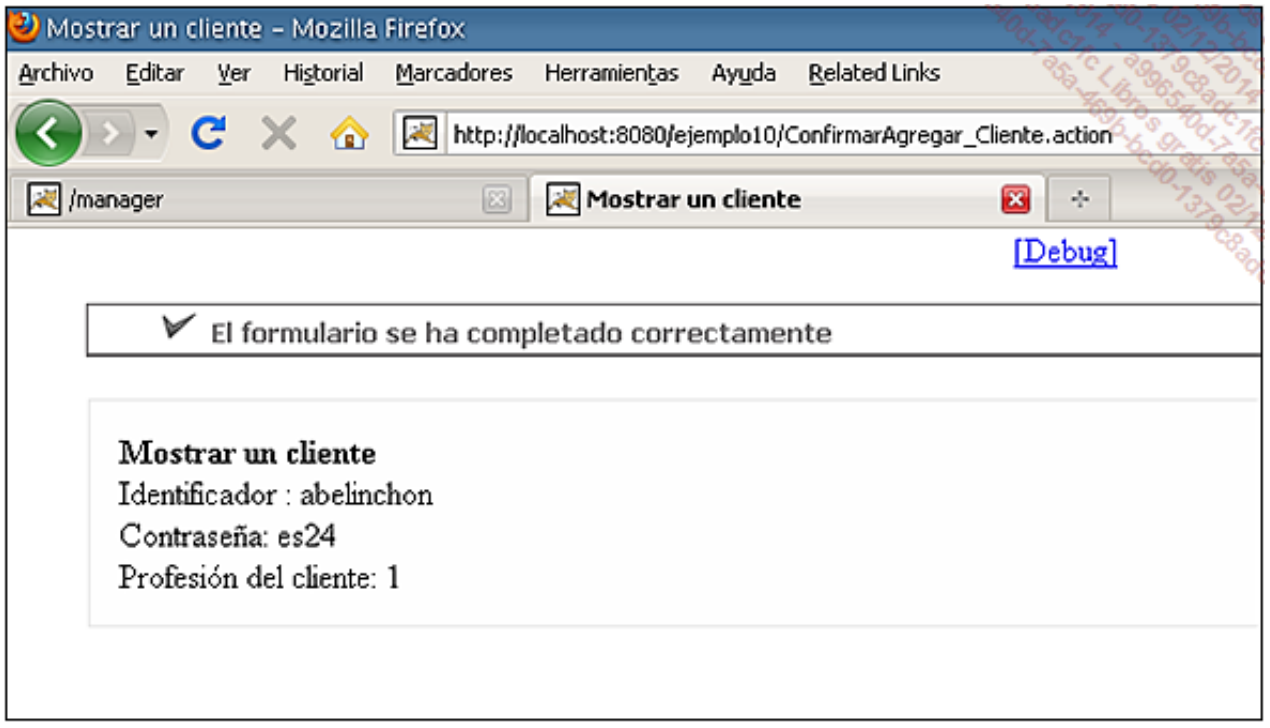
Código: ejemplo10.package.properties

```
...
validadoridentificador=El campo ${getText(fieldName)} debe tener
entre ${getText(minLength)} y ${getText(maxLength)} caracteres y
no debe estar en uso
```

```
Código: ejemplo10.package_en.properties
...
validadoridentificador=The field ${getText(fieldName)} must be
between ${getText(minLength)} and ${getText(maxLength)} characters
and must not be used
```



Formulario de cliente multilingüe con validaciones personales



4. Validación en las clases de acción

El párrafo anterior explica la aplicación de una validación XML para los campos utilizados de manera declaratoria. Struts incluye también validaciones de software en forma de código Java, que puede integrarse en las clases de acción. La interfaz *com.opensymphony.xwork2.Validateable* puede implementarse en las clases de acción para facilitar una validación por programación. Esta interfaz requiere una sola escritura de método denominada `validate()`. El método `validate()` contiene toda la lógica de validación de las propiedades de los JavaBeans.

Si una clase de acción implementa la interfaz `Validateable`, el método `validate()` se ejecutará automáticamente. Si nuestra clase de acción es heredera de la clase `ActionSupport`, no tenemos necesidad de implementar la interfaz `Validateable`.

Vamos a utilizar esta técnica para imponer el identificador *adurán* durante la creación de una cuenta de cliente. De lo contrario, se agregará un mensaje de tipo *fieldError*.

Struts ofrece una programación sencilla y permite combinar las técnicas de validación. En nuestro ejemplo, hemos utilizado validaciones XML (bundle validation) por definición, un validador específico y por último una validación de programación, con el método `validate()`en la clase de acción.

Código: ejemplo10.ClienteAccion.java

```
package ejemplo10;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo10.javabeans.Cliente;
import ejemplo10.javabeans.Profesion;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

	// objeto JavaBean
	private Cliente cliente;
	// lista de profesiones
	private List<Profesion> listaProfesiones=new
ArrayList<Profesion>();

	// getter y setter

	// método para verificar que el identificador utilizado sea
adurán
	public void validate()
	{
		if(cliente!=null)
		{
			if(!
cliente.getIdentificador().equals("adurán"))
			{
addFieldError("identificadorusuario",getText("identificador
usuario"));
			 }
		}
	}

	// método para agregar un mensaje de confirmación
	public String verificar()
```

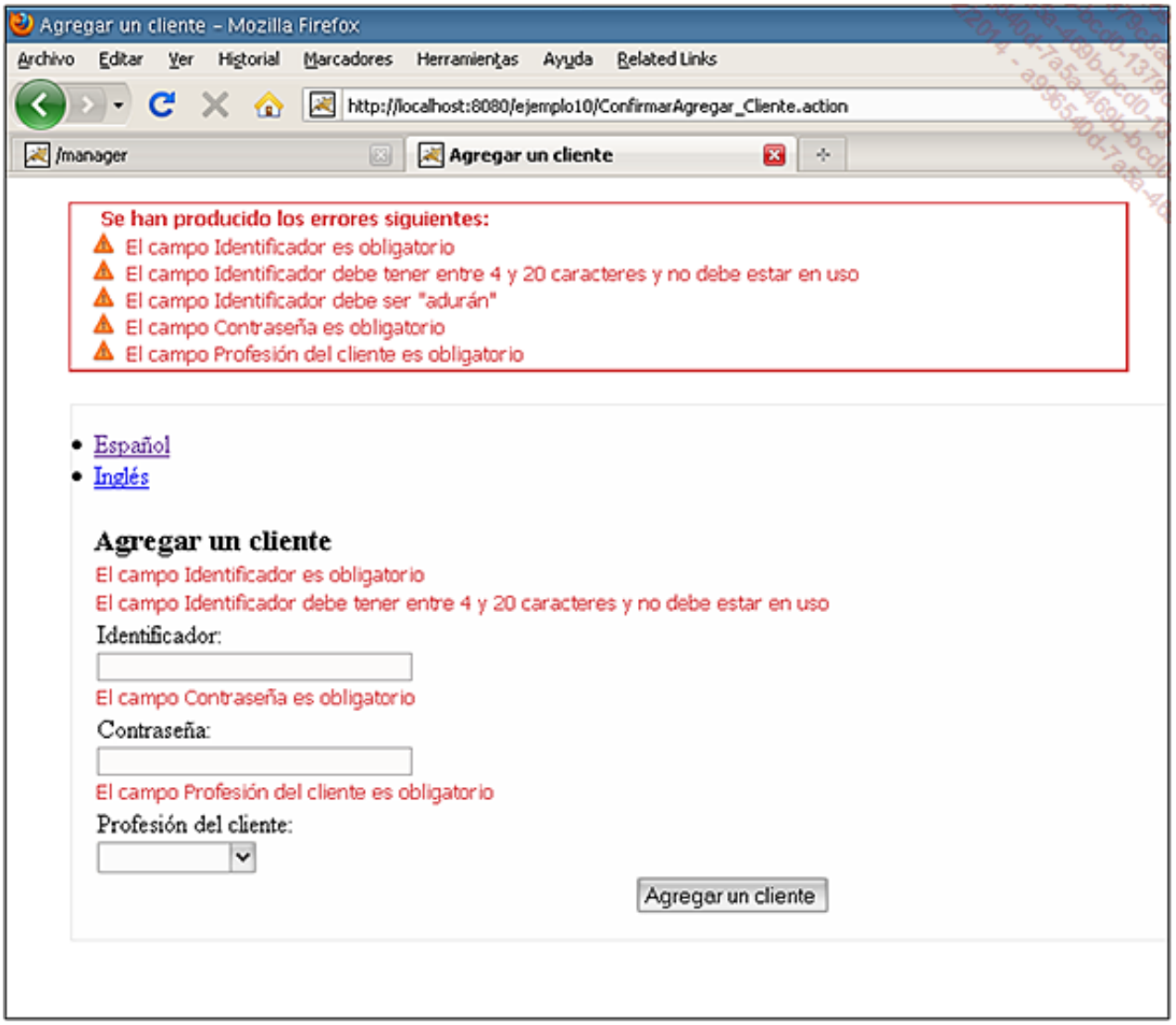
```
        {
                addActionMessage(getText("correcto"));
                return SUCCESS;
        }
}
```

Código: ejemplo10.package.properties

```
...
identificadorusuario=El campo Identificador debe ser "adurán"
```

Código: ejemplo10.package_en.properties

```
...
identificadorusuario=The field Login must be "adurán"
```



Formulario de cliente con validación personal del identificador

En resumen

En este capítulo se explica la configuración de validaciones en una aplicación de Struts. Para ello, el framework ofrece varias herramientas basadas en la elaboración de archivos XML, la creación de validadores personales y en casos más concretos, de la configuración de validaciones de software en la programación. Asimismo, Struts propone combinar estas tres técnicas en proyectos de gran envergadura.

Presentación

En los capítulos anteriores, hemos estudiado cómo recuperar los datos introducidos en los formularios y llevar a cabo validación es más o menos complejas.

Cada parámetro transmitido mediante el protocolo HTTP se considera como una cadena de caracteres de tipo *String*. Por ejemplo, la introducción de la edad del cliente en el formulario se verificará por medio de los validadores como un entero, pero se recibirá como una cadena de caracteres. Para evitar las conversiones múltiples y costosas, con Struts es posible utilizar servicios simples de conversiones.

Administración de las conversiones

Struts utiliza un interceptor llamado *params*, responsable de realizar el mapping entre los campos de formularios y sus descriptores de acceso respectivos. Gracias a este interceptor, las cadenas de caracteres recibidos por el protocolo HTTP se convierten automáticamente al tipo definido en la clase de acción, como el nuestro ejemplo anterior sobre la administración de la cuenta del cliente. El ID de la profesión del cliente se recibe en forma de una cadena de caracteres, pero se convierte automáticamente en un entero.

Struts lleva a cabo conversiones entre los distintos tipos y un problema de conversión (por ejemplo, de caracter a entero) no tiene por qué detener el framework. La interfaz*com.opensymphony.xwork2.ValidationAware* es la encargada de administrar las conversiones de tipos. Si nuestra clase de acción no implementa esta interfaz, se pueden producir excepciones.

Los mensajes de error de conversión se administran a través del archivo de propiedades. La introducción de un valor incorrecto a nivel de tipo generará un mensaje como el siguiente: *Invalid field value for field fieldName*.

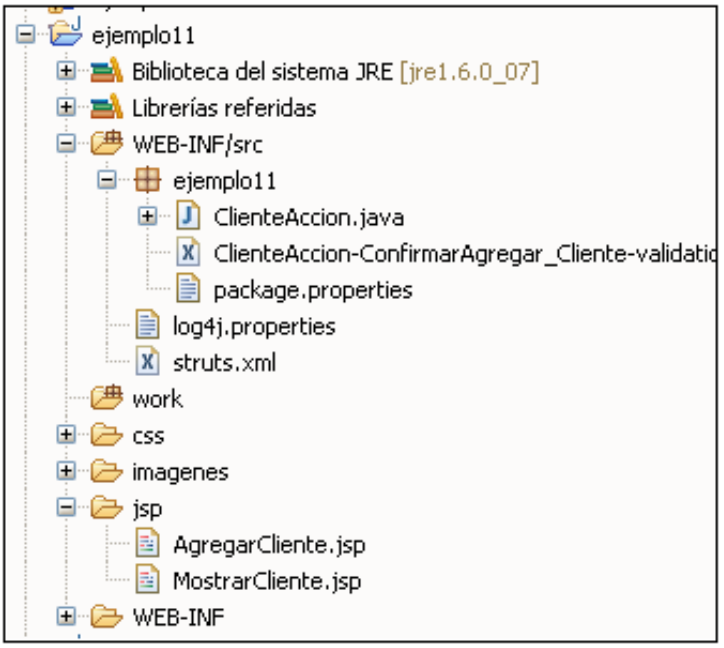
Ya nos hemos encontrado con este tipo de mensaje en el proyecto *ejemplo08* cuando hemos introducido un carácter para la edad del cliente. Podemos modificar las visualizaciones relativas a los tipos sobrecargando estos mensajes por defecto en nuestro archivo de propiedades.

En nuestro caso, podemos utilizar esta definición en el archivo *package.properties*.

```
Invalid.fieldvalue.NombreDelCampo=Error de tipo en un campo
del formulario
```

El parámetro *NombreDelCampo* corresponde al nombre del campo utilizado en los formularios y las validaciones.

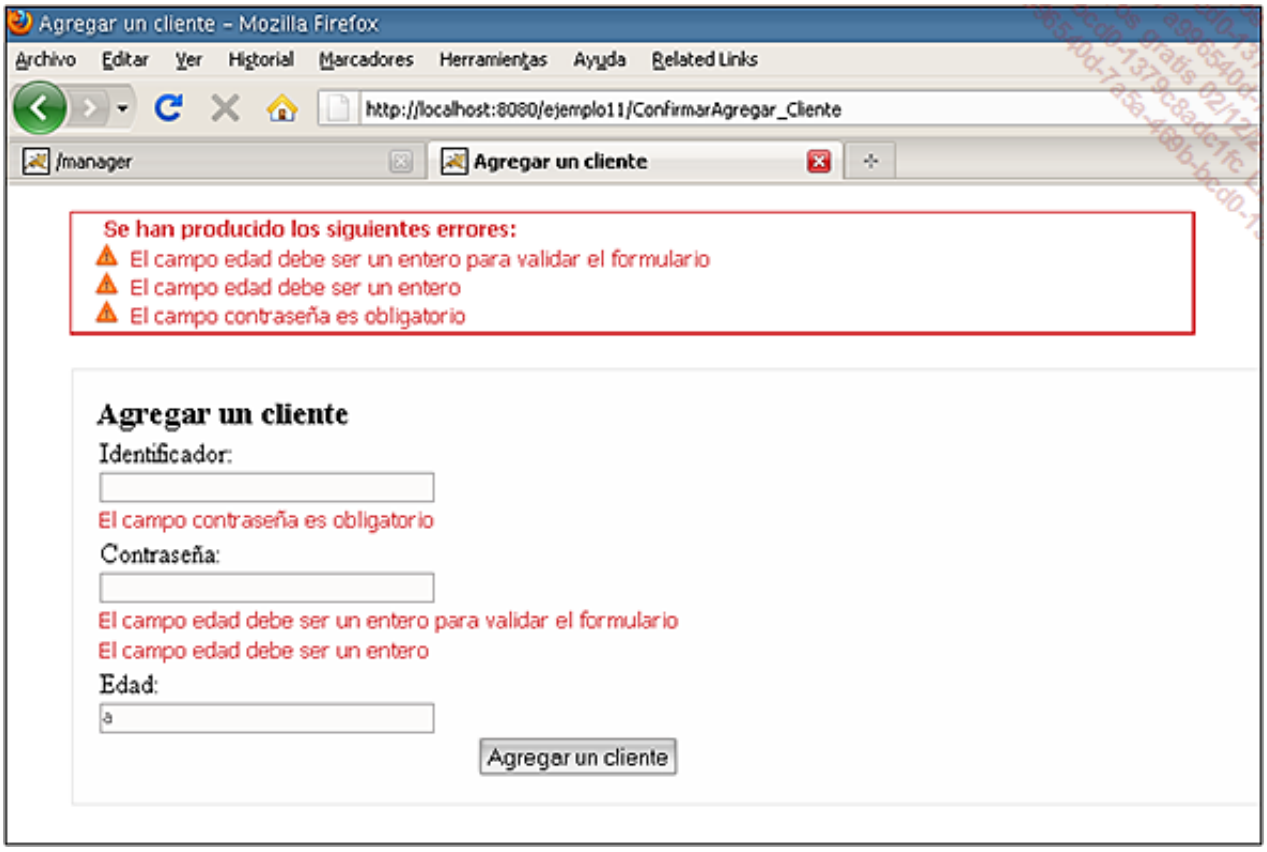
Podemos retomar nuestra aplicación *ejemplo08* y utilizarla para nuestro nuevo proyecto *ejemplo11*.



Árbol del ejemplo11

Código: ejemplo11.package.properties

```
invalid.fieldvalue.edad=El campo edad debe ser un número entero para
poder confirmar el formulario
```



Formulario de registro de cliente y administración de los tipos

---

El archivo de validación que contiene los mensajes (ejemplo11.package.properties) también puede ser nombrado según el nombre de la clase ubicarse en el mismo entorno que éste dentro del árbol del proyecto. En nuestro ejemplo, habríamos podido utilizar el archivo ejemplo11.ClientAction.properties.

Administración de los tipos

Como hemos explicado en el párrafo anterior, Struts proporciona conversores de tipos simples que pueden utilizarse en gran medida en la mayoría de los proyectos. Sin embargo, no es posible realizar conversiones de tipos complejos con el sistema que se incluye por defecto. Para ello, debemos crear nuestros propios conversores de tipos. Un conversor de tipos debe implementar la interfaz*ognl.TypeConverter* o heredar de la clase `DefaultTypeConverter` o `StrutsTypeConverter`.

Esta interfaz tiene una única firma de método llamada `convertValue(...)`, que permite administrar la conversión al tipo adaptado. La implementación de este método permite recuperar la propiedad afectado por el tipo, así como la clase de la propiedad.

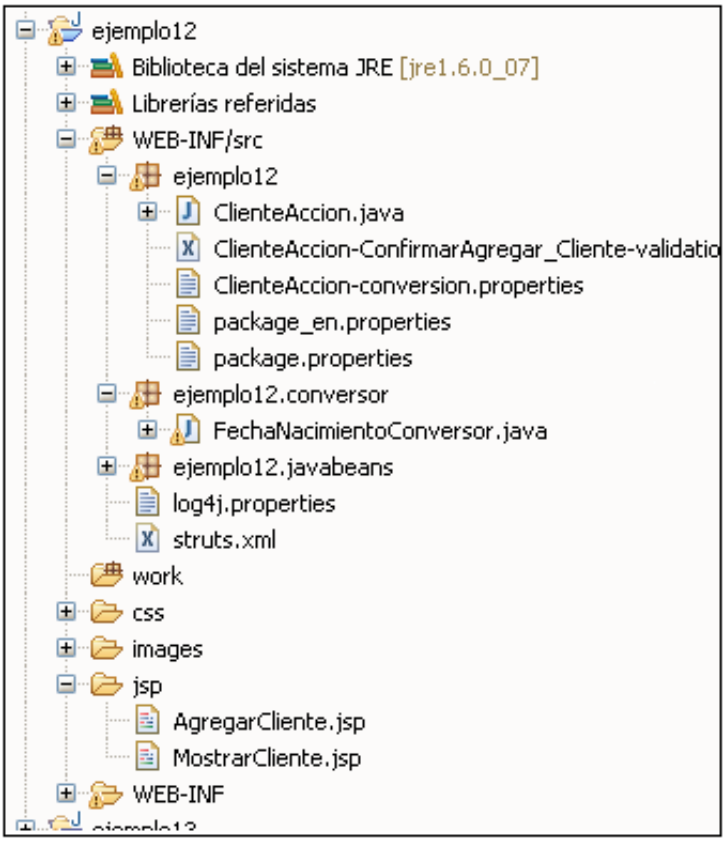
Aplicación

La aplicación de las conversiones automáticas de tipos necesita una configuración previa. Para ello, debemos crear un archivo que deberá contener el nombre de la clase de acción con el sufijo*conversion*. Para nuestro ejemplo de la clase `ClienteAccion`, se colocará el archivo de conversión en el mismo directorio que la clase y llevará el nombre *ClienteAccion-conversion.properties*.

El archivo de configuración debe encontrarse en el mismo directorio que la clase de acción.

Este archivo de propiedades contiende los nombres de los campos que deben asociarse a las conversiones de tipos, así como la clase asociada a la verificación del tipo.. Las clases de administración de tipos asociadas a una acción siempre se ejecutarán, incluso en el caso de utilizar validaciones. Por lo tanto, nuestra clase de validación del tipo para el campo *fechaNacimiento* se ejecutará por el interceptor al mismo tiempo que el setter de la propiedad afectada.

Para aplicar la administración de los tipos y las conversiones vamos a retomar nuestro proyecto*ejemplo10* basado en la administración de cuentas de cliente y añadiremos una propiedad*fechaNacimiento* de tipo *java.util.Date*.



Árbol del proyecto ejemplo12

Esta aplicación contiene un nuevo campo para la fecha de nacimiento del cliente. El archivo de propiedades *ClienteAccion-conversion.properties* permite realizar el enlace entre el campo que se va a verificar y la clase encargada de realizar esta operación.

```
Código: ejemplo12.ClienteAccion-conversion.properties
cliente.fechaNacimiento=ejemplo12.conversor.FechaNacimiento
Conversor
```

La clase que permite administrar la conversión de tipo es simple, recuperar los datos introducidos y realiza una conversión un modelo especificado (en español en este caso). Si la conversión es correcta, devolver el resultado en el objeto afectado, en caso contrario saldrá del conversor sin bloquear la aplicación.

Los datos introducidos correctamente serán procesados y reenviados al formulario mientras que los datos introducidos incorrectamente no serán reenviados al formulario.

Código: ejemplo12.conversor.FeachaNacimientoConversor

```
package ejemplo12.conversor;

import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import ognl.DefaultTypeConverter;

public class FechaNacimientoConversor extends
DefaultTypeConverter {

    public Object convertValue(Map context, Object valor, Class
clase)
    {
       System.out.println("En el conversor");
       // ¿los datos introducidos son de tipo Date
        if (clase == Date.class)
        {
           DateFormat dateFormat=new
SimpleDateFormat("dd/MM/yyyy");
           dateFormat.setLenient(false);
           try
           {
               String[] s=(String[])valor;
               Date date=dateFormat.parse(s[0]);
               System.out.println("en la fecha:
"+dateFormat.format(date));
               return date;
           }
           catch(ParseException e)
           {
             //System.out.println("Error:" + e);
           }
        }
        return null;
    }
}
```

Código: /jsp/AgregarCliente.jsp

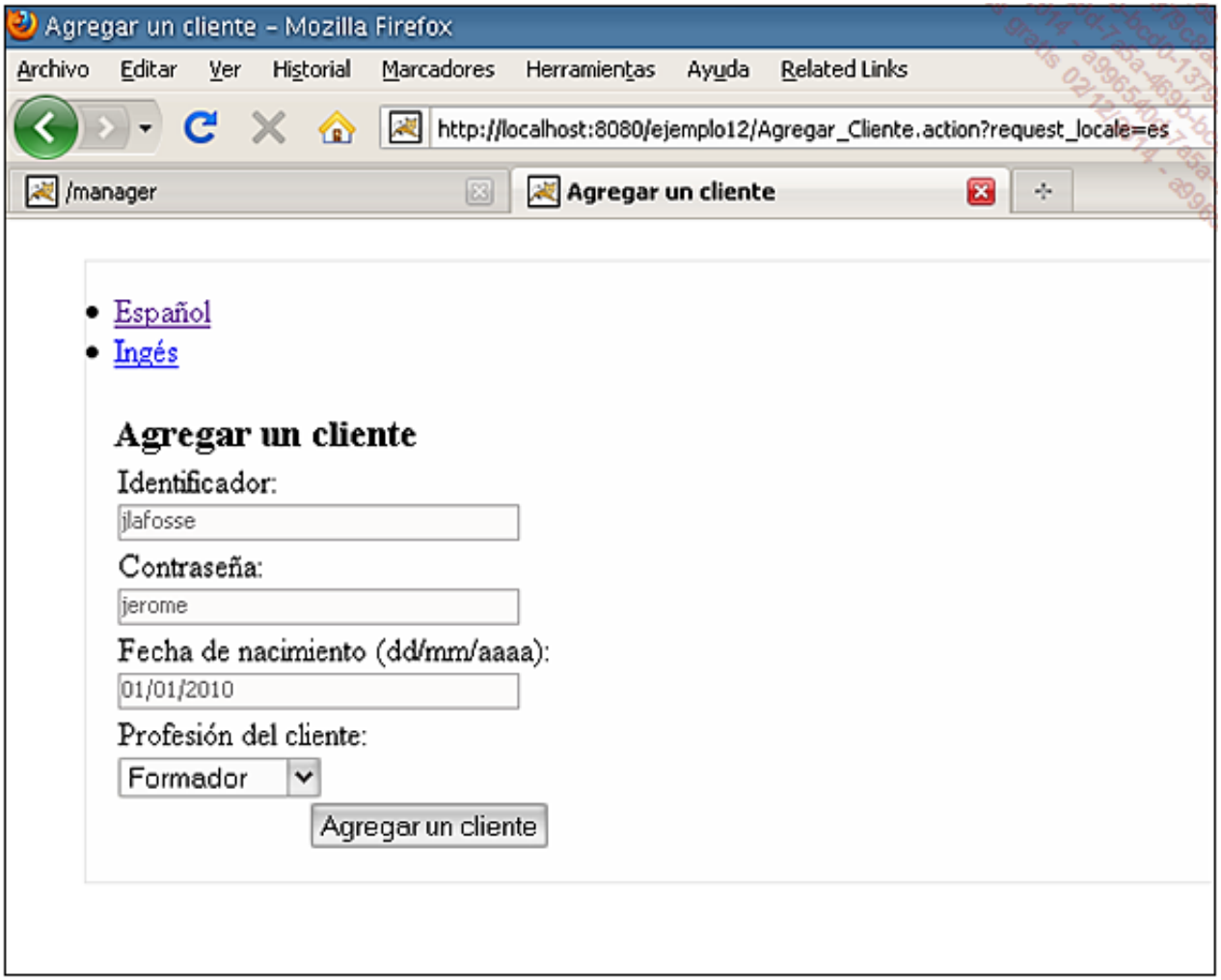
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.agregar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>

<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label><s:property value="%{getText('error')}"/></label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<!-- Mensaje de error durante las acciones -->
<s:if test="errorMessages.size()>0">
```

```
<div id="mensaje_error">
      <label><s:property value="%
{getText('erroraccion')}"/></label>
      <ul><s:actionerror/></ul>
</div>
</s:if>
<div id="carta">
      <ul>
            <li><a href="Agregar_Cliente.action?
request_locale=es">Espa&ntilde;ol</a></li>
            <li><a href="Agregar_Cliente.action?
request_locale=en">Inge&eacute;s</a></li>
      </ul>
      <br/>
      <h3><s:property value="%{getText('cliente.agregar')}"/></h3>
      <s:form method="post" action="ConfirmarAgregar_Cliente">
            <s:textfield name="cliente.identificador"
id="cliente.identificador" label="%{getText('cliente.identificador')}"
labelposition="top" cssClass="input"/>
            <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="%{getText('cliente.contrasena')}"
labelposition="top" cssClass="input"/>
            <s:textfield name="cliente.fechaNacimiento"
id="cliente.fechaNacimiento" label="%
{getText('cliente.fechaNacimiento')}" labelposition="top"
cssClass="input" />
            <s:select name="cliente.profesion.idProfesion"
id="cliente.profesion.idProfesion" label="%
{getText('cliente.profesion.idProfesion')}" labelposition="top"
list="listaProfesiones" listKey="idProfesion" listValue="nombre"/>
            <s:submit value="%{getText('cliente.agregar')}"/>
      </s:form>
</div>
</body>
</html>
```

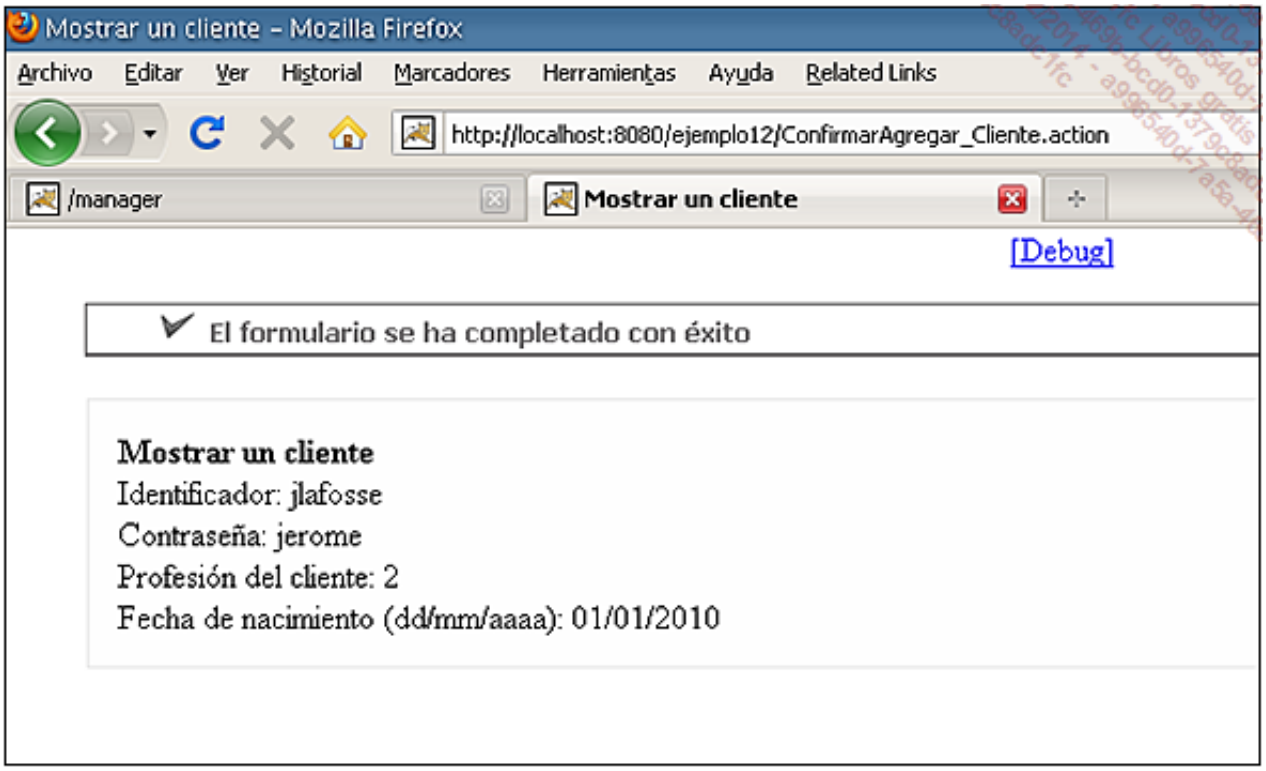
Código: /jsp/MostrarCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title><s:property value="%{getText('cliente.mostrar')}"/></title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<!-- Mensaje de confirmación -->
<s:if test="actionMessages.size()>0">
      <div id="mensaje_informacion">
            <ul><s:actionmessage/></ul>
      </div>
</s:if>
<div id="carta">
      <p>
            <h4><s:property value="%
{getText('cliente.mostrar')}"/></h4>
            <s:property value="%
{getText('cliente.identificador')}"/>: <s:property
value="cliente.identificador"/> <br/>
            <s:property value="%{getText('cliente.contrasena')}"/
>: <s:property value="cliente.contrasena"/><br/>
            <s:property value="%
{getText('cliente.profesion.idProfesion')}"/>: <s:property
value="cliente.profesion.idProfesion"/><br/>
            <s:property value="%
```

```
{getText('cliente.fechaNacimiento')}"/>: <s:date
name="cliente.fechaNacimiento" format="dd/MM/yyyy"/>
      </p>
</div>
</body>
</html>
```



Formulario de registro de un cliente ejemplo12



Visualización de la cuenta del cliente

En resumen

El protocolo HTTP utiliza únicamente cadenas de caracteres para transmitir información a las aplicaciones. Esta restricción de tipo hace necesario realizar conversiones (de cadenas de caracteres a entero, fecha un objeto). Con Struts es posible una administración automática de las conversiones para los tipos simples a partir de las cadenas de caracteres recibidas. Además, podemos definir nuestras propias clases de conversiones y de administración de tipos a partir de una nomenclatura precisa y eficaz.

Presentación

La mayoría de las aplicaciones Web utilizan una base de datos para garantizar la persistencia de la información. Por ejemplo, los sitios comerciales almacenan los datos relativos a los clientes, artículos o encargos. La mayoría de las bases de datos comercializadas son de naturaleza cliente-servidor y recurren al lenguaje SQL para manipular los datos que contienen. Java cuenta con una API para trabajar con las bases de datos. Esta tecnología, denominada JDBC (*Java DataBase Connectivity*), es una biblioteca de interfaces y clases, utilizada para acceder a los SGBDR (Sistemas de Gestión de Bases de Datos Relacionales). JDBC ofrece a los desarrolladores todas las herramientas necesarias para permitir que los programas cliente se conecten con las bases de datos y les envíen sus consultas. Estas consultas están escritas en lenguaje SQL.

Las aplicaciones actuales Java EE utilizan el modelo de diseño MVC (*Modelo Vista Controlador*). Hasta el momento hemos estudiado la parte Vista con las páginas JSP y la parte Controlador con las clases de acción, por lo que nos falta la parte de persistencia y el acceso a los datos con la capa Modelo.

Arquitectura MVC

Las aplicaciones se dividen en varias capas o niveles, con el fin de separar cada funcionalidad del sistema. La comunicación entre dos niveles (vista con controlador, controlador con modelo y viceversa) se basa en la invocación de los descriptores de acceso (getter y setter) y la transmisión de parámetros adaptados. Struts se considera un framework de presentación y se ocupa principalmente de las capas Vista y Controlador. La configuración de la capa Modelo es tarea del desarrollador.

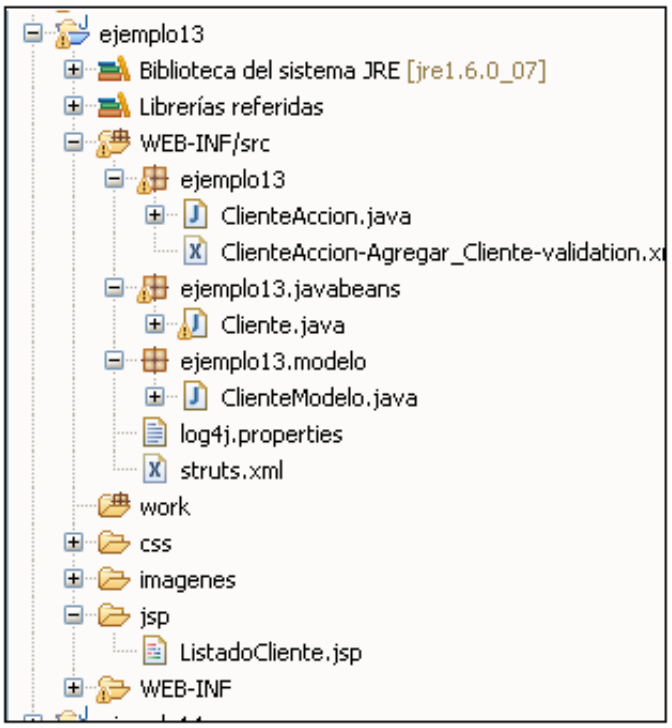
Así, la última capa será responsable de la persistencia de los objetos utilizados o del acceso al SGBD para transformar los datos objeto en datos relacionales. Esta técnica se denomina mapping relacional hacia objeto. La técnica contraria, que consiste en transformar un objeto en datos en una base de datos relacional, se conoce como mapping objeto hacia relacional, y se realiza mediante consultas SQL.

Por convención, las clases de acción llevan el sufijo *Acción* (por ejemplo: `ClienteAccion.java`), y están asociadas con sus clases modelo, que cuentan con el término *Modelo* como sufijo o prefijo (por ejemplo: `ClienteModelo.java` o `ModeloCliente.java`). Estas dos clases manipulan un JavaBean correspondiente al objeto en sí (sus atributos y métodos) (por ejemplo: *Cliente.java*). Las clases de acción se consideran el controlador y llevan a cabo los tratamientos importantes de una aplicación. La clase modelo realiza todos los tratamientos de acceso o de modificación de datos (por ejemplo: consultar, agregar, modificar, eliminar y realizar un listado).

Aplicación

Podemos utilizar el modelo MVC con el ejemplo del formulario del cliente reducido al identificador y la contraseña. La capa modelo será realizada/simulada por una clase estática que contiene una lista dinámica de cuentas de cliente y podrá ser actualizada.

El nuevo proyecto *ejemplo13* utiliza la clase JavaBean `Cliente` para gestionar el identificador y la contraseña. Esta clase se utiliza en la clase de acción `ClienteAccion` y en el modelo `ClienteModelo`. El árbol del proyecto debe ser el siguiente:



Árbol del proyecto ejemplo13

El archivo de configuración de la aplicación *struts.xml* cuenta con dos acciones, una para la creación de una nueva cuenta (*Agregar_Cliente.action*) y otra para mostrar la información (*Listado_Cliente.action*). La acción *Listado_Cliente.action* desencadena el método `listado()` de la clase de acción y la acción `Agregar_Cliente.action` está asociada al método `agregar()` de la clase de acción.

```
Código: 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="true" />

    <package name="ejemplo13" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo13.ClienteAccion" method="listado">
            <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
```

```
class="ejemplo13.ClienteAccion" method="agregar">
            <result name="input">/jsp/ListadoCliente.jsp</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

    </package>
</struts>
```

La clase de acción es simple y cuenta únicamente con sus descriptores de acceso y dos métodos muy útiles, `listado()` y `agregar()`. El método `listado()` recupera la información presente en el modelo para asignársela a la colección que se mostrará en la vista a través de su getter.

Código: ejemplo13.ClienteAccion.java

```
package ejemplo13;

import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo13.javabeans.Cliente;
import ejemplo13.modelo.ClienteModelo;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private Cliente cliente;
    private List<Cliente> listaClientes;

      public Cliente getCliente() {
             return cliente;
      }

      public void setCliente(Cliente cliente) {
             this.cliente = cliente;
      }

      public List<Cliente> getListaClientes() {
             listaClientes=ClienteModelo.getListaClientes();
             return listaClientes;
      }

      public void setListaClientes(List<Cliente> listaClientes) {
             this.listaClientes = listaClientes;
      }

      // devolver la lista de clientes tras la recuperación
      public String listado()
      {
             listaClientes=ClienteModelo.getListaClientes();
       return SUCCESS;
      }

      // agregar el cliente al modelo
      public String agregar()
      {
             ClienteModelo.agregar(this.cliente);
          return SUCCESS;
      }
}
```

La clase JavaBean `Cliente`, muy sencilla, está compuesta por sus atributos y sus descriptores de acceso.

Código: ejemplo13.javabeans.Cliente.java

```
package ejemplo13.javabeans;

@SuppressWarnings("serial")
public class Cliente {

    private int idCliente;
    private String identificador;
    private String contrasena;

    public Cliente() {

    }

    public Cliente(int idCliente,String identificador, String
contrasena){
        this.idCliente=idCliente;
        this.identificador=identificador;
        this.contrasena=contrasena;
    }

        // descriptores de acceso de la clase
}
```

Por último, el modelo permite devolver la lista de los clientes creados y agregar un nuevo cliente a la lista. Esta etapa utiliza una lista estática, pero en un proyecto avanzado debería sustituirse por una base de datos.

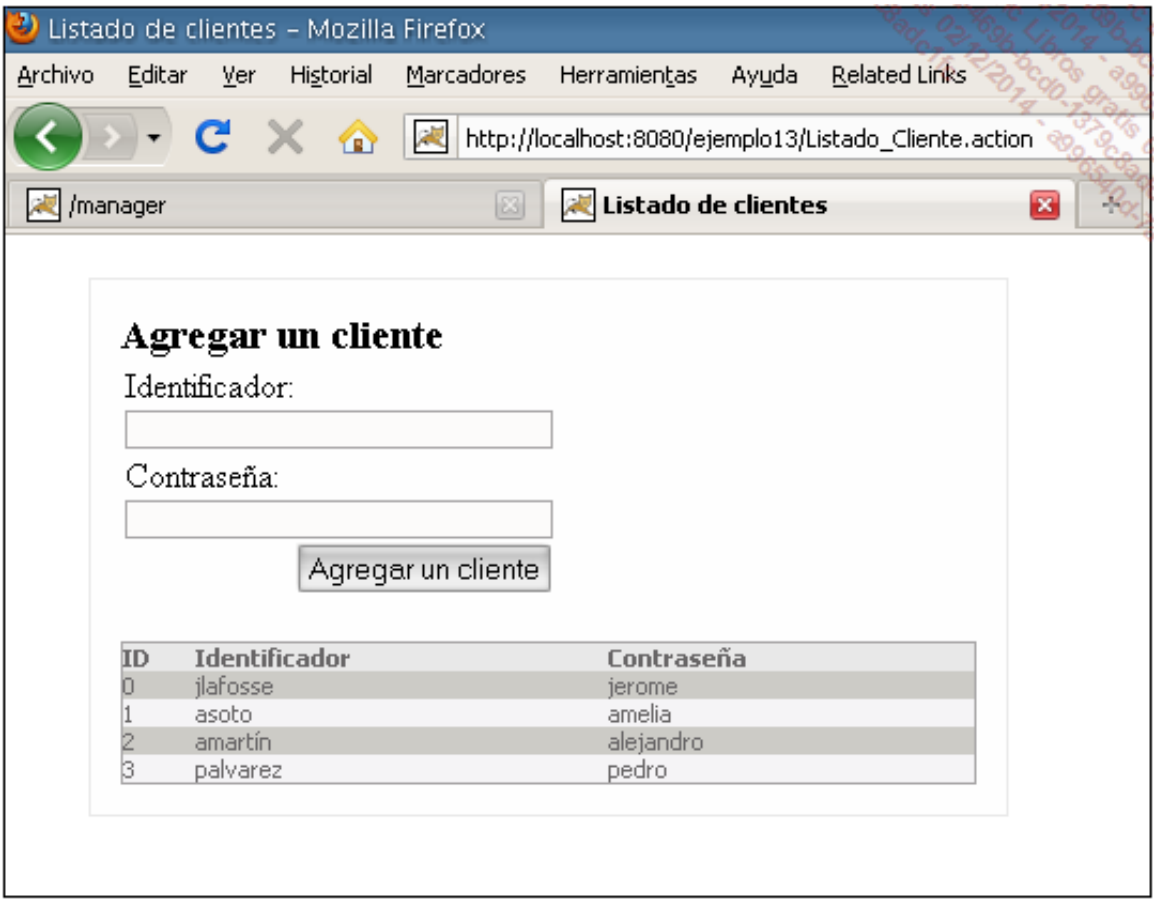
Esta clase modelo utiliza una lista estática que será compartida por el conjunto de objetos usuario.

Código: ejemplo13.modelo.ClienteModelo.java

```
package ejemplo13.modelo;

import java.util.ArrayList;
import java.util.List;
import ejemplo13.javabeans.Cliente;

public class ClienteModelo {
    private static List<Cliente> listaClientes;
    private static int id=0;

    static
    {
        listaClientes=new ArrayList<Cliente>();
        listaClientes.add(new Cliente(id++, "jlafosse", "jerome"));
        listaClientes.add(new Cliente(id++, "asoto", "amelia"));
        listaClientes.add(new Cliente(id++, "amartín", "alejandro"));
        listaClientes.add(new Cliente(id++, "palvarez", "pedro"));
    }

    // devolver la lista de clientes
    public static List<Cliente> getListaClientes() {
              return listaClientes;
       }

       public static void setListaClientes(List<Cliente>
listaClientes) {
               ClienteModelo.listaClientes = listaClientes;
       }

       // agregar un cliente a la lista
       public static void agregar(Cliente cliente) {
```

```
            cliente.setIdCliente(id++);
            listaClientes.add(cliente);
        }
}
```

Puede acceder al proyecto en la dirección http://localhost:8080/ejemplo13/Listado_Cliente.action y mostrar directamente, gracias al método `listado()` de la clase de acción y a la línea*listaClientes=ClienteModelo.getListaClientes()*, la lista de los clientes presentes en el modelo. Ahora podemos agregar una cuenta correcta y validar la creación, a continuación la lista se actualizará automáticamente.



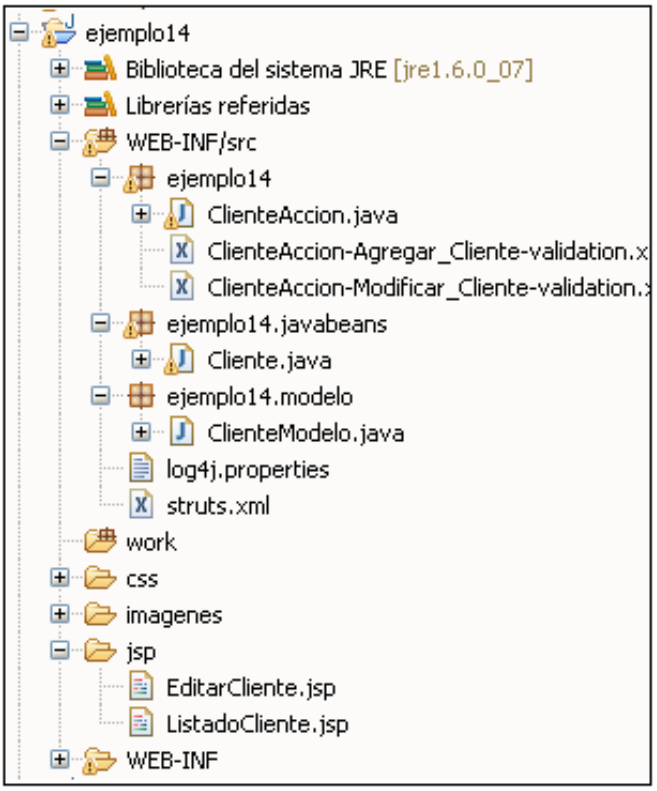
Administración de los clientes

La utilización del tipo redirectAction en el resultado de la adición <result name="success" type="redirectAction">Listado_Cliente</result> permite redirigir totalmente el formulario a la acción Listado_Cliente.action para evitar así creaciones dobles en caso de actualización de la página (F5) o de hacer doble clic en el botón de adición. Asimismo, dado que este tipo es una redirección real, se perderán todos los parámetros y se limpiarán los datos introducidos por el usuario (identificador y contraseña).

Ahora, si intentamos crear una cuenta con un error de introducción (por ejemplo: sin introducir la contraseña), observaremos que se conservan los datos pero no se muestra la lista de clientes. Es algo totalmente normal puesto que la acción *listado()* de la clase de acción que devuelve la lista desde el modelo no se ejecuta en caso de error en la introducción de datos. Por tanto, los datos ya no se muestran. El interceptador *Preparable* permite evitar este problema.

El interceptor Preparable

El interceptor *Preparable* acude al método `prepare()` cada vez que se ejecuta la clase de acción, si este interceptor está asociado con la acción, por supuesto. Para ello, la clase de acción debe implementar la interfaz *com.opensymphony.xwork2.Preparable*. Vamos a configurar un servicio completo denominado *ejemplo14* de realización de listado, creación, modificación y eliminación de clientes a partir de este interceptor.



Árbol del proyecto ejemplo14

Comenzaremos modificando el archivo de declaración de las acciones *struts.xml*.

```
Código: 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="true" />

    <package name="ejemplo14" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo14.ClienteAccion" method="listado">
            <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo14.ClienteAccion" method="agregar">
            <result name="input">/jsp/ListadoCliente.jsp</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
```

```
        </action>

        <action name="Editar_Cliente"
class="ejemplo14.ClienteAccion" method="editar">
            <result name="success">/jsp/EditarCliente.jsp</result>
        </action>

        <action name="Modificar_Cliente"
class="ejemplo14.ClienteAccion" method="modificar">
            <result name="input">/jsp/EditarCliente.jsp</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Eliminar_Cliente"
class="ejemplo14.ClienteAccion" method="eliminar">
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

    </package>
</struts>
```

Existen cinco acciones declaradas que permiten efectuar tareas muy concretas:

- *Listado_Cliente.action*: esta acción permite ejecutar el método `listado()` de la clase de acción para recuperar la lista completa de clientes a través del modelo. Esta acción realiza una redirección hacia la vista *ListaCliente.jsp* para mostrar el formulario de creación y la lista de cuentas de clientes.

- *Agregar_Cliente.action*: esta acción permite ejecutar el método `agregar()` de la clase de acción y agregará directamente el objeto *cliente* (informado por el interceptor *params*) al modelo. Esta acción utiliza el archivo de validación *ClienteAccion-Agregar_Cliente-validación.xml* y realiza una redirección completa tras una creación hacia la acción de realización del listado de clientes.

- *Editar_Cliente.action*: esta acción permite mostrar el formulario de edición (modificación) de una cuenta de cliente. Para ello, la acción utiliza un parámetro denominado *idClienteActual*transmitido al vínculo y que corresponde al id. del cliente que desea editar.

- *Modificar_Cliente.action*: esta acción permite modificar el objeto *cliente* a través del modelo utilizando los nuevos valores introducidos en el formulario. Al igual que el formulario de creación, esta acción utiliza un archivo de validación de los datos introducidos llamado*ClienteAccion-Modificar_Cliente-validación.xml*. Podemos observar así la flexibilidad de Struts, que permite utilizar validaciones diferentes dependiendo de la acción que deba realizarse (creación o modificación). Por supuesto, si las validaciones son idénticas, podemos utilizar un*visitor*, como se ha explicado en el capítulo Biblioteca de etiquetas de Struts, dedicado a las validaciones. Esta acción realiza una redirección completa a la acción de creación de una nueva cuenta y de realización del listado de clientes.

- *Eliminar_Cliente.action*: esta acción permite eliminar un cliente a partir del parámetro*idClienteActual*, que corresponde al id. del cliente que desea eliminar. En caso de eliminación, esta acción redirige hacia la página de realización del listado.

Código: ejemplo14.ClienteAccion.java

```
package ejemplo14;

import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.opensymphony.xwork2.Preparable;
import ejemplo14.javabeans.Cliente;
```

```
import ejemplo14.modelo.ClienteModelo;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport implements
Preparable, ModelDriven{

    private Cliente cliente;
    private List<Cliente> listaClientes;
    private int idClienteActual;

       public void prepare() throws Exception {
              // en creación, crear un nuevo objeto vacío
              if(idClienteActual==0)
              {
                     cliente=new Cliente();
              }
              // en modificación, devolver la información del objeto
              else
              {
       cliente=ClienteModelo.getCliente(idClienteActual);
              }
       }

       public Object getModel() {
              return cliente;
       }

       public int getIdClienteActual() {
              return idClienteActual;
       }

       public void setIdClienteActual(int idClienteActual) {
              this.idClienteActual = idClienteActual;
       }

       public Cliente getCliente() {
              return cliente;
       }

       public void setCliente(Cliente cliente) {
              this.cliente = cliente;
       }

       public List<Cliente> getListaClientes() {
       listaClientes=ClienteModelo.getListaClientes();
return listaClientes;
       }

       public void setListaClientes(List<Cliente> listaClientes) {
              this.listaClientes = listaClientes;
       }

       // devolver la lista de clientes tras la recuperación
       public String listado()
       {
              listaClientes=ClienteModelo.getListaClientes();
              return SUCCESS;
       }

       // agregar el cliente al modelo
       public String agregar()
       {
              ClienteModelo.agregar(cliente);
              return SUCCESS;
       }
```

```
	// mostrar el formulario en edición
	public String editar()
	{
		return SUCCESS;
	}

	// modificar un cliente
	public String modificar()
	{
		ClienteModelo.modificar(cliente);
		return SUCCESS;
	}

	// eliminar un cliente a partir del parámetro recibido llamado
idCliente
	public String eliminar()
	{
		ClienteModelo.eliminar(idClienteActual);
		return SUCCESS;
	}
}
```

Código: ejemplo14.modelo.ClienteModelo.java

```
package ejemplo14.modelo;

import java.util.ArrayList;
import java.util.List;
import ejemplo14.javabeans.Cliente;

public class ClienteModelo {
    private static List<Cliente> listaClientes;
    private static int id=1;

    static
    {
       listaClientes=new ArrayList<Cliente>();
       listaClientes.add(new Cliente(id++, "jlafosse", "jerome"));
       listaClientes.add(new Cliente(id++, "asoto", "amelia"));
       listaClientes.add(new Cliente(id++, "amartin", "alejandro"));
       listaClientes.add(new Cliente(id++, "palvarez", "pedro"));
    }

       // devolver la lista de clientes
       public static List<Cliente> getListaClientes() {
              return listaClientes;
       }

       public static void setListaClientes(List<Cliente>
listaClientes) {
               ClienteModelo.listaClientes = listaClientes;
       }

       // agregar un cliente a la lista
       public static void agregar(Cliente cliente) {
         cliente.setIdCliente(id++);
         listaClientes.add(cliente);
        }

       // eliminar un cliente de la lista
       public static void eliminar(int idCliente) {
              for(int i=0;i<listaClientes.size();i++)
              {
                     Cliente c=listaClientes.get(i);
```

```
                        if(c.getIdCliente()==idCliente)
                        {
                                listaClientes.remove(c);
                        }
                }
        }

        // modificar un cliente de la lista
        public static void modificar(Cliente cliente) {
                int idCliente=cliente.getIdCliente();
                for(int i=0;i<listaClientes.size();i++)
                {
                        Cliente c=listaClientes.get(i);

                        if(c.getIdCliente()==idCliente)
                        {

        c.setIdentificador(cliente.getIdentificador());

        c.setContrasena(cliente.getContrasena());
                                break;
                        }
                }
        }

        // buscar un cliente en la lista
        public static Cliente getCliente(int idCliente) {
                for(int i=0;i<listaClientes.size();i++)
                {
                        Cliente c=listaClientes.get(i);
                        if(c.getIdCliente()==idCliente)
                        {
                                return c;
                        }
                }
                return null;
        }
}
```

```
Código: /jsp/ListadoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Lista de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los errores siguientes: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<div id="carta">

       <h3>Agregar un cliente</h3>
       <s:form method="post" action="Agregar_Cliente">
              <s:textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
              <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
```

```
              <s:submit value="Agregar un cliente"/>
       </s:form>

       <table border="0" id="tabla" cellpadding="0"
cellspacing="0">
       <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td colspan="2"
align="center"><b>Gestión</b></td></tr>
       <s:iterator value="listaClientes" status="linea">
       <s:if test="#linea.odd"><tr class="linea1"></s:if>
       <s:if test="#linea.even"><tr class="linea2"></s:if>
       <td><s:property value="idCliente"/></td>
       <td><s:property value="identificador"/></td>
       <td><s:property value="contrasena"/></td>
       <td align="center"><a href="Editar_Cliente.action?
idClienteActual=${idCliente}"/><img src="imagenes/editarcliente.png"
alt="Editar" title="Editar" border="0"/></a></td>
       <td align="center"><a href="Eliminar_Cliente.action?
idClienteActual=${idCliente}"/><img src="imagenes/eliminarcliente.png"
alt="Eliminar" title="Eliminar" border="0"/></a></td>
       </tr>
       </s:iterator>
       </table>
</div>
</body>
</html>
```

Código: /jsp/EditarCliente.jsp

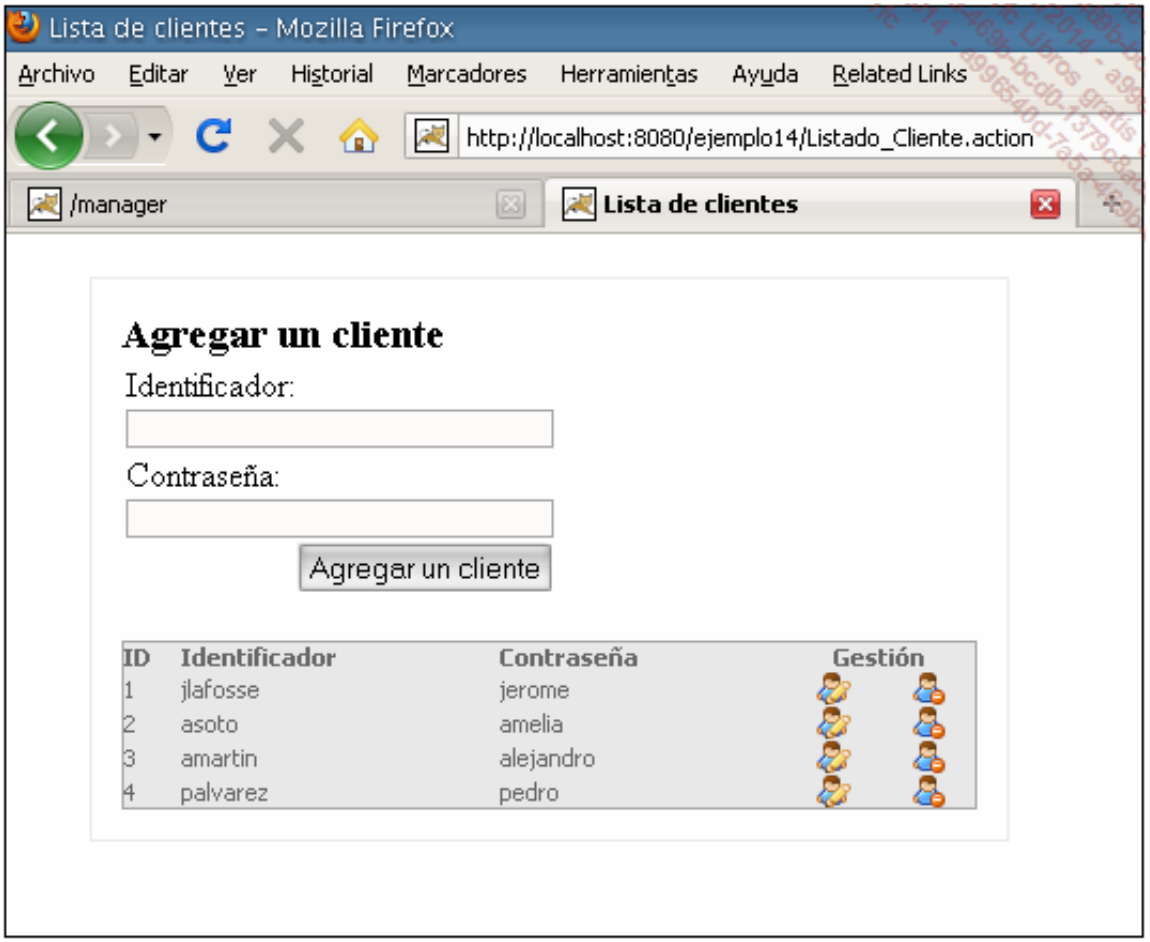
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Editar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los errores siguientes: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<div id="carta">

       <h3>Editar un cliente</h3>
       <s:form method="post" action="Modificar_Cliente">
              <s:hidden key="cliente.idCliente"/>
              <s:textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
              <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
              <s:submit value="Modificar un cliente"/>
       </s:form>
</div>
</body>
</html>
```

La administración de los clientes ha terminado. Observamos que los servicios de visualización, adición y eliminación funcionan correctamente, al contrario que el servicio de edición. En realidad, al hacer clic en el vínculo de edición/modificación, el formulario está vacío. Se trata de algo totalmente normal, el método `prepare()` de nuestra clase de acción, que permite buscar al cliente concernido

por*idClienteActual* no se ha ejecutado. Para ejecutar este método automáticamente cada vez que se acuda a la acción, es necesario configurar el interceptor *Preparable*. La configuración se realiza incluyendo la línea siguiente *<interceptor-ref name="paramsPrepareParamsStack"/>* en el archivo de configuración struts.xml.

```
Código: 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>
...
        <action name="Editar_Cliente"
class="ejemplo14.ClienteAccion" method="editar">
                    <interceptor-ref
name="paramsPrepareParamsStack"/>
           <result name="success">/jsp/EditarCliente.jsp</result>
        </action>
...
</struts>
```

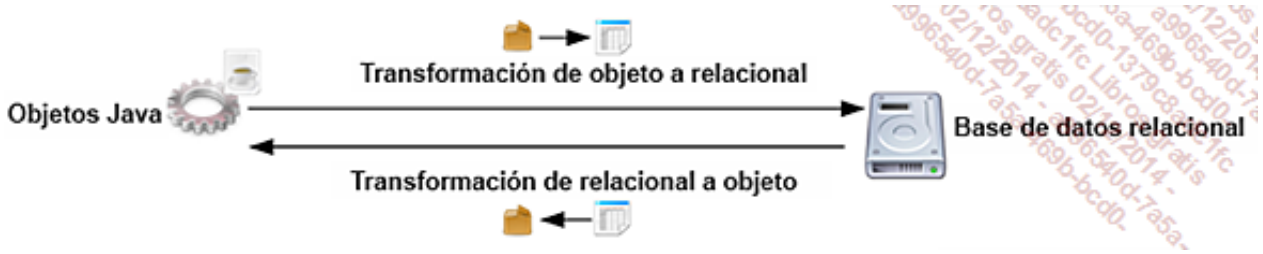
A partir de ahora, nuestra aplicación *ejemplo14* está totalmente operativa.



Administración completa de los clientes

Acceso y manipulación de datos

En los ejemplos anteriores, los datos se almacenan en una colección dinámica Java que contiene las cuentas de los clientes. Durante la utilización de proyectos de gran envergadura, las bases de datos se usan para garantizar la persistencia de los datos. Los modelos objeto-relacional y relacional-objeto se utilizarán para gestionar la persistencia de los datos en Java.



Transformaciones de objeto en relacional y relacional en objeto

Las transformaciones objeto-relacional (objetos Java a base de datos) y relacional-objeto (base de datos a objeto Java) se denominan mapping. Actualmente existe un gran número de herramientas, más o menos funcionales, para llevar a cabo estos servicios. Los más conocidos son la librería OpenSource Hibernate y Java Persistence API (JPA). Sin necesidad de utilizar una herramienta de mapping, es totalmente posible llevar a cabo este servicio incluyendo diversas líneas de programación a nivel del software. El mecanismo implementado por el desarrollador utiliza el modelo o pattern Data Access Object (DAO), suficiente para la mayoría de aplicaciones profesionales.

El modelo o capa de persistencia, ofrece un conjunto de métodos para consultar, agregar, modificar, eliminar o realizar un listado de los objetos, imágenes de los valores de la base de datos.

1. El modelo Data Access Object DAO

Este modelo facilita un conjunto de reglas o consignas que deben seguirse para configurar un sistema de persistencia de objetos. Con este modelo de diseño, cada clase utilizada en el sistema debe poseer su propia clase modelo para gestionar su persistencia. Por ejemplo, tendremos una clase JavaBean `Cliente.java` y una clase `ClienteModeloDAO.java`. La clase modelo contiene los métodos de modificación del estado del objeto (agregar, modificar y eliminar) y de consulta (consultar, realizar listado).

El modelo DAO de base se utiliza para cada clase de acción de Struts que debe llevar a cabo una operación en la base de datos. El modelo DAO prevé también la configuración de una clase principal (o varias) para gestionar los métodos comunes y la declaración de las conexiones al SGBD. La interfaz `DAO` es implementada por al clase DAO principal para indicar que se debe sobrecargar, y por tanto utilizar, el método `getConnection()`. La clase `ModeloDAO` se sitúa en la parte superior de la jerarquía e implementa la anterior interfaz `DAO`. Esta clase permite la implementación del método`getConnection()` que será utilizado por las clases secundarias, así como por el setter, para la transmisión de la conexión.

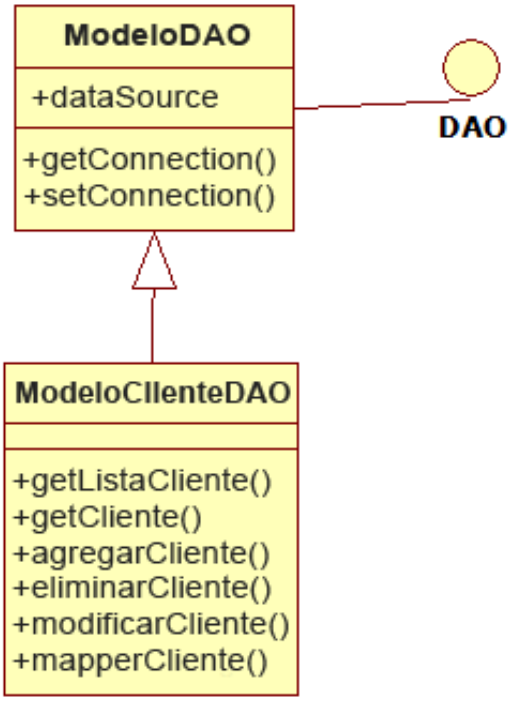
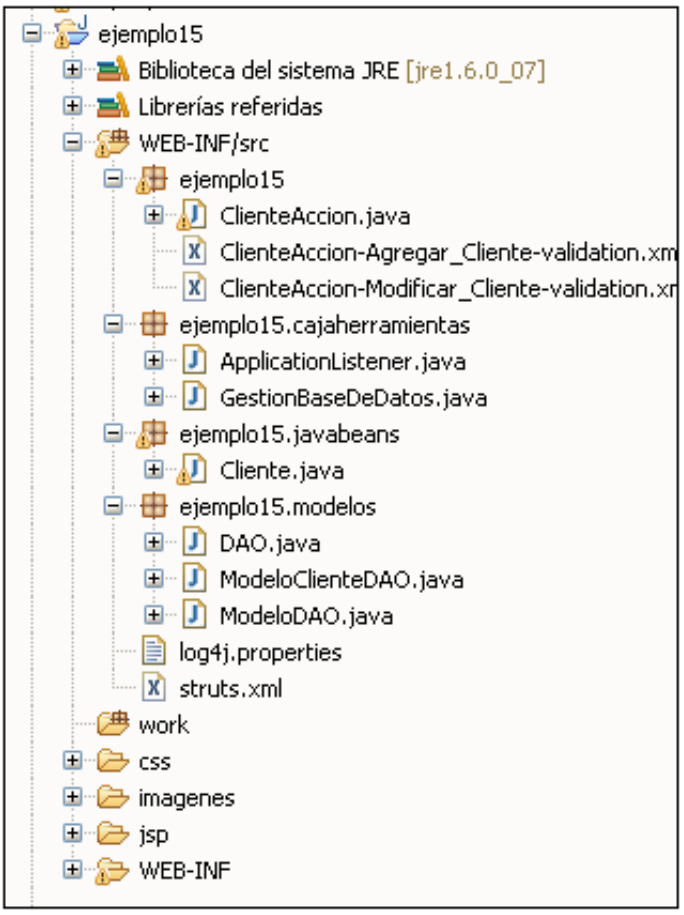


Diagrama UML del pattern DAO

Vamos a utilizar un nuevo proyecto *ejemplo15* basado en el ejemplo anterior para hacer uso de la parte modelo DAO. La base de datos utilizada se denomina *struts2*.



Árbol del proyecto ejemplo15

La interfaz DAO requiere una sola escritura del método `getConnection()` para la conexión al SGBD.

Código: ejemplo15.modelos.DAO.java

```
package ejemplo15.modelos;

import java.sql.Connection;

public interface DAO {

	// Definición del método a declarar en las clases
de los usuarios
	public Connection getConnection();
}
```

La clase `ModeloDAO` propone la implementación del método `getConnection()` a partir de un pool de conexión (DataSource). Este método devuelve una conexión SQL que puede ser utilizada por cualquier clase secundaria del modelo. Los pools de conexiones (*javax.sql.DataSource*) permiten declarar conexiones a partir del archivo XML de la aplicación (*web.xml*) y gestionar más fácilmente la configuración de la misma. Por tanto, la conexión con la fuente de datos se almacena en el contexto de la aplicación (*ServletContext*).

Código: ejemplo15.modelos.ModeloDAO.java

```
package ejemplo15.modelos;

import java.sql.Connection;
import java.sql.SQLException;
import javax.servlet.ServletContext;
import javax.sql.DataSource;
import org.apache.struts2.ServletActionContext;

// Clase de conexión
public class ModeloDAO  implements DAO
{
	DataSource dataSource=null;

	// Recuperar una conexión
	public Connection getConnection()
	{
		ServletContext
servletContext=ServletActionContext.getServletContext();
		if(this.dataSource==null)
		{


dataSource=(DataSource)servletContext.getAttribute("dataSource");
		}
		Connection connection=null;
		if(dataSource!=null)
		{
			try
			{
				connection=dataSource.getConnection();
			}
			catch(SQLException e)
			{
				System.out.println(e);
			}
		 }

		// devolver la conexión
		return connection;
	}

	// Posicionar una dataSource
	public void setConnection(DataSource dataSource)
```

```
	{
		this.dataSource=dataSource;
	}
}
```

Para aplicar la conexión al SGBD, utilizaremos un escuchador (listener) activado al iniciar la aplicación y configurado en el archivo de gestión de la aplicación *web.xml*. También se declaran dos etiquetas para la conexión al SGBD (`<context-param/>` y `<resource-ref/>`). Para ello, podemos utilizar la clase`javax.servlet.ServletContextListener` y los métodos `contextInitialized()` y `contextDestroyed()`. Esta clase se declara en un paquete específico con una clase estática `GestionBaseDeDatos.java` que permite gestionar el cierre de las conexiones.

```
Código: /WEB-INF/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">
    <!-- Cargador de datasource -->
    <listener>
        <listener-
class>ejemplo15.cajaherramientas.ApplicationListener</listener-class>
    </listener>

    <!-- Parámetros globales -->
    <context-param>
        <param-name>dataSourceJNDI</param-name>
        <param-value>java:/comp/env/jdbc_struts2_MySQL</param-
value>
    </context-param>

    <filter>
        <filter-name>struts2</filter-name>
        <filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExec
uteFilter</filter-class>
    </filter>

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

    <!-- Información de conexión a la base de datos -->
       <resource-ref>
              <description>Conexión a la base de datos
MySQL</description>
              <res-ref-name>jdbc_struts2_MySQL</res-ref-name>
              <res-type>javax.sql.DataSource</res-type>
              <res-auth>Container</res-auth>
       </resource-ref>
</web-app>
```

```
Código: ejemplo15.cajaherramientas.ApplicationListener.java
package ejemplo15.cajaherramientas;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletContext;
```

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.sql.DataSource;

public class ApplicationListener implements
ServletContextListener{

	Context context=null;

	//función llamada durante la creación del iniciador
	public void contextInitialized(ServletContextEvent
servletContextEvent)
	{
		ServletContext
servletContext=servletContextEvent.getServletContext();
		String
dataSourceJNDI=servletContext.getInitParameter("dataSourceJNDI");

		try
		{
			context=new InitialContext();
			DataSource
dataSource=(DataSource)context.lookup(dataSourceJNDI);
			if(dataSource==null)
			{
				System.out.println("No hay DataSource
para el proyecto: ejemplo15");
			}
			else
			{
				System.out.println("DataSource:
¡ejemplo15 cargado!");
			}
			servletContext.setAttribute("dataSource",
dataSource);
		 }
		 catch(NamingException e)
		 {
			throw new RuntimeException();
		 }
		 finally
		 {
			try
			{
				//cerrar el contexto
				if(context!=null)
				{
					context.close();
				}
			 }
			 catch(Exception e)
			 {
				 System.out.println("¡Error en
initCtx!");
			 }
		}
	}

	//función llamada durante la destrucción del iniciador
	public void contextDestroyed(ServletContextEvent
servletContextEvent)
	{
		try
		{
			 //cerrar el contexto
```

```
                        if(context!=null)
                        {
                                context.close();
                        }
                }
                catch(Exception e)
                {
                        System.out.println("¡Error en
initCtx!");
                }
        }
}
```

Código: ejemplo15.cajaherramientas.GestionBaseDeDatos.java

```
package ejemplo15.cajaherramientas;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class GestionBaseDeDatos
{
        // Permite cerrar un resultset
        public static void closeResulset(ResultSet resultado)
        {
                if(resultado!=null)
                {
                        try
                        {
                                resultado.close();
                        }

                        catch(Exception e)
                        {
                                System.out.println("Error en el
cierre de la conexión de un resultset");
                        }
                }
        }

        // Cierre de una consulta
        public static void closeRequest(Statement consulta)
        {
                if(consulta!=null)
                {
                        try
                        {
                                consulta.close();
                        }
                        catch(Exception e)
                        {
                                System.out.println("Error en el
cierre de una consulta");
                        }
                }
        }

        // Cierre de una conexión
        public static void closeConnection(Connection connection)
        {
                if(connection!=null)
                {
                        try
                        {
```

```
                        connection.close();
                    }

                    catch(Exception e)
                    {
                        System.out.println("Error en el
cierre de una conexión");
                    }
                }
        }
}
```

La configuración del pool de conexión casi ha finalizado, sólo nos falta la declaración del pool en el archivo de configuración del proyecto (*/tomcat/conf/Catalina/localhost/ejemplo15.xml*) o del servidor (*/tomcat/conf/Catalina/server.xml*).

```
Código: /tomcat/conf/Catalina/localhost/ejemplo15.xml
<Context path="/ejemplo15" reloadable="true"
docBase="C:\JAVA\PROJECTOWEB\ejemplo15"
workDir="C:\JAVA\PROJECTOWEB\ejemplo15\work">
       <Resource name="jdbc_struts2_MySQL"
                 auth="Container"
                 type="javax.sql.DataSource"
                 username="root"
                 password=""
                 driverClassName="com.mysql.jdbc.Driver"
                 url="jdbc:mysql://localhost:3306/struts2"
                 maxActive="20"
                 maxIdle="10"
                 validationQuery="SELECT 1"
        />
</Context>
```

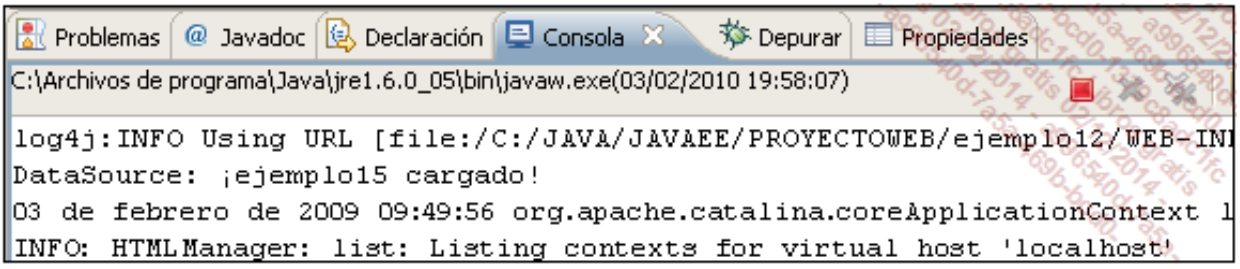
La aplicación de una conexión con un SGBD requiere la instalación del piloto adecuado en el directorio /Tomcat/lib o /WEB-INF/lib de la aplicación. Para el proyecto, se utiliza el piloto mysql-connector-java-3.1.11-bin.jar.

Para gestionar la persistencia de los datos vamos a crear la base de datos MySQL *struts2* con una tabla *cliente* y tres campos: *idCliente*, *identificador* y *contraseña*. La tabla *cliente* se completa con varios almacenamientos para comprobar la aplicación.

```
INSERT INTO `cliente` VALUES (1, 'jlafosse', 'jerome');
INSERT INTO `cliente` VALUES (2, 'asoto', 'amelia');
INSERT INTO `cliente` VALUES (3, 'amartín', 'alejandro');
INSERT INTO `cliente` VALUES (4, 'pálvarez', 'pedro');
```

Estructura MySQL de la base de datos struts2

Podemos empezar por comprobar nuestra conexión al SGBD iniciando la aplicación y verificando si el listener gestiona correctamente el pool.



Seguimiento de la carga del pool de conexión en la consola

La clase de acción de Struts prácticamente no se modifica, utiliza el modelo dinámico*ModeloClienteDAO* en vez del modelo de la clase estática del ejemplo anterior.

```
Código: ejemplo15.ClienteAccion.java
package ejemplo15;

import java.util.ArrayList;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.opensymphony.xwork2.Preparable;
import ejemplo15.javabeans.Cliente;
import ejemplo15.modelos.ModeloClienteDAO;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport implements
Preparable, ModelDriven{

    private Cliente cliente;
    private List<Cliente> listaClientes;
    private int idClienteActual;

       public void prepare() throws Exception {
              ModeloClienteDAO ModeloClienteDAO=new
ModeloClienteDAO();
              // en creación, crear un nuevo objeto vacío
              if(idClienteActual==0)
              {
                     cliente=new Cliente();
              }
              // en modificación, devolver la información del objeto
              else
              {

      cliente=modeloClienteDAO.getCliente(idClienteActual);
              }
      }

      public Object getModel() {
             return cliente;
      }

      public int getIdClienteActual() {
             return idClienteActual;
      }

      public void setIdClienteActual(int idClienteActual) {
             this.idClienteActual = idClienteActual;
      }

      public Cliente getCliente() {
             return cliente;
```

```
	}

	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

	public List<Cliente> getListaClientes() {
		ModeloClienteDAO ModeloClienteDAO=new
modeloClienteDAO();

listaClientes=(ArrayList<Cliente>)modeloClienteDAO.getListaClientes();
		 return listaClientes;
	}

	public void setListaClientes(List<Cliente> listaClientes) {
		this.listaClientes = listaClientes;
	}

	// devolver la lista de clientes tras la recuperación
	public String listado()
	{
		ModeloClienteDAO modeloClienteDAO=new
ModeloClienteDAO();

listaClientes=(ArrayList<Cliente>)modeloClienteDAO.getListaClientes();
		 return SUCCESS;
	}

	// agregar el cliente al modelo
	public String agregar()
	{
		ModeloClienteDAO modeloClienteDAO=new
ModeloClienteDAO();
		ModeloClienteDAO.agregarCliente(cliente);
		return SUCCESS;
	}

	// mostrar el formulario en edición
	public String editar()
	{
		return SUCCESS;
	}

	// modificar un cliente
	public String modificar()
	{
		ModeloClienteDAO modeloClienteDAO=new
ModeloClienteDAO();
		modeloClienteDAO.modificarCliente(cliente);
		return SUCCESS;
	}

	// eliminar un cliente a partir del parámetro recibido llamado
idCliente
	public String eliminar()
	{
		ModeloClienteDAO modeloClienteDAO=new
ModeloClienteDAO();
		modeloClienteDAO.eliminarCliente(idClienteActual);
		return SUCCESS;
	}
}
```

Por último, el modelo se basa en el esquema UML (*Unified Modeling Language*) anterior y propone las características necesarias para la consulta, modificación, eliminación y adición de clientes, así como el

método de mapping relacional-objeto. Los métodos de la clase corresponden a la definición del del pattern DAO modelo CRUD (Create, Request, Update, Delete) del pattern DAO con las funciones que permiten realizar listados de clientes, crear un nuevo cliente (C), recuperar la colección (R), modificarla (U) y por último eliminarla (D).

Código: ejemplo15.modelos.ModeloClienteDAO.java

```
package ejemplo15.modelos;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;
import ejemplo15.cajaherramientas.GestionBaseDeDatos;
import ejemplo15.javabeans.Cliente;

public class ModeloClienteDAO extends ModeloDAO{

	// Variables
	Connection conexion=null;
	ResultSet resultado=null;
	private static List<Cliente> listaClientes;

	// devolver la lista de clientes
	public List<Cliente> getListaClientes()
	{
		// Variables
		PreparedStatement consulta=null;
		Cliente cliente=null;
		String consultaString=null;
		listaClientes=new ArrayList<Cliente>();

		try
		{
			// Apertura de una conexión
			conexion=super.getConnection();

			// consulta de lista de clientes
			consultaString="SELECT * FROM cliente WHERE 1
ORDER BY idCliente";

	consulta=conexion.prepareStatement(consultaString);

			// Ejecución de la consulta
			resultado=consulta.executeQuery();

			// Se almacena el resultado en una lista
			if(resultado!=null)
			{
				while(resultado.next())
				{
					// Se efectúa el mapping de
los atributos con los campos de la tabla SQL
					cliente=mapperCliente(resultado);

					// Se añade el objeto a la lista
de clientes

	listaClientes.add((Cliente)cliente);
				}
			}
		}
		catch(Exception e)
		{
```

```
                    System.out.println("Error en la consulta
de la clase ModeloClienteDAO función getListaClientes");
             }
             finally
             {
                    try
                    {
                           // Cierre de la conexión
                           if(resultado!=null)
                           {

       GestionBaseDeDatos.closeResulset(resultado);
                           }
                           if(consulta!=null)
                           {

       GestionBaseDeDatos.closeRequest(consulta);
                           }
                           if(conexion!=null)
                           {

       GestionBaseDeDatos.closeConnection(conexion);
                           }
                    }
                    catch(Exception ex)
                    {
                           System.out.println("Error en el
cierre de la conexion con la base de datos en la clase
ModeloClienteDAO función getListaClientes");
                    }
             }

             // Devolver la lista de clientes
             return listaClientes;
       }

       // buscar un cliente en la base
       public Cliente getCliente(int idCliente)
       {
             // Variables
             PreparedStatement consulta=null;
             Cliente cliente=null;
             String consultaString=null;

             try
             {
                    // Apertura de una conexión
                    conexion=super.getConnection();

                    // Creación de la consulta
                    consultaString = "SELECT * FROM cliente WHERE
idCliente=?";

                    // Se prepara la consulta

       consulta=conexion.prepareStatement(consultaString);
                    consulta.setInt(1,idCliente);

                    // Ejecución de la consulta
                    resultado=consulta.executeQuery();

                    // Se almacena el resultado en el objeto cliente
                    if(resultado!=null)
                    {
                           if(resultado.next())
```

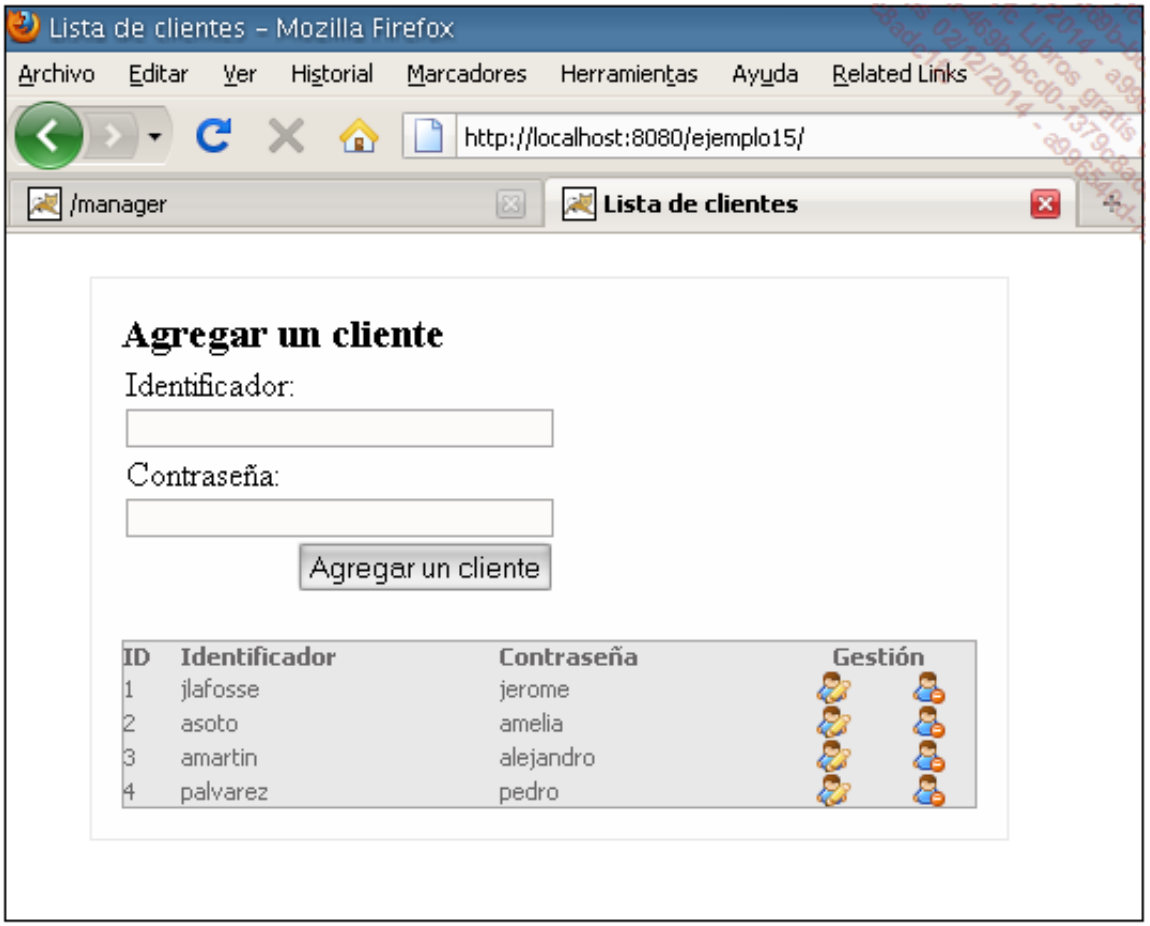
```
                        {
                                // Se realiza el mapping de
los atributos con los campos de la tabla SQL
                                cliente=mapperCliente(resultado);
                        }
                }
            }
            catch(Exception e)
            {
                cliente=null;
                System.out.println("Error en la consulta
en la clase ModeloClienteDAO función getCliente");
            }
            finally
            {
                try
                {
                        // Cierre de la conexión
                        if(resultado!=null)
                        {

       GestionBaseDeDatos.closeResulset(resultado);
                        }
                        if(consulta!=null)
                        {

       GestionBaseDeDatos.closeRequest(consulta);
                        }
                        if(conexion!=null)
                        {

       GestionBaseDeDatos.closeConnection(conexion);
                        }
                }
                catch(Exception ex)
                {
                        System.out.println("Error en el
cierre de la conexión con la base de datos en la clase
ModeloClienteDAO función getCliente");
                }
            }

            // Devolver objeto cliente
            return cliente;
       }

       // agregar un cliente a la base
       public int agregarCliente(Cliente cliente)
       {
            // Variables
            PreparedStatement consulta=null;
            String consultaString=null;
            int codigoError=0;

            try
            {
                // Apertura de una conexión
                conexion=super.getConnection();

                // Creación de la consulta
                consultaString="INSERT INTO cliente
(identificador,contrasena) VALUES(?,?)";

                // Preparación de la consulta
```

```
	consulta=conexion.prepareStatement(consultaString);
                        consulta.setString(1,
cliente.getIdentificador());
                        consulta.setString(2, cliente.getContrasena());

                        // Se vacía el cliente por seguridad
                        cliente=null;

                        // Ejecución de la consulta
                        codigoError=consulta.executeUpdate();
                }
                catch(Exception e)
                {
                        codigoError=0;
                        System.out.println("Error en la consulta
de la clase ModeloClienteDAO función agregarCliente");
                }
                finally
                {
                        try
                        {
                                // Cierre de la conexión
                                if(resultado!=null)
                                {

	GestionBaseDeDatos.closeResulset(resultado);
                                }
                                if(consulta!=null)
                                {

	GestionBaseDeDatos.closeRequest(consulta);
                                }
                                if(conexion!=null)
                                {

	GestionBaseDeDatos.closeConnection(conexion);
                                }
                        }
                        catch(Exception ex)
                        {
                                System.out.println("Error en el
cierre de la conexión con la base de datos en la clase
ModeloClienteDAO función agregarCliente");
                        }
                }

                // Devolver el código de error
                return codigoError;
        }


        // eliminar un cliente en la base
        public int eliminarCliente(int idCliente)
        {
                // Variables
                PreparedStatement consulta=null;
                String consultaString=null;
                int codigoError=0;

                try
                {
                        // Apertura de una conexión
                        conexion=super.getConnection();

                        // Eliminar el cliente
```

```
                    consultaString="DELETE FROM cliente WHERE
idCliente=?";

        consulta=conexion.prepareStatement(consultaString);
                    consulta.setInt(1,idCliente);

                    // Ejecución de la consulta
                    codigoError=consulta.executeUpdate();
              }
              catch(Exception e)
              {
                    codigoError=0;
                    System.out.println("Error en la consulta
de la clase ModeloClienteDAO función eliminarCliente");
              }
              finally
              {
                    try
                    {
                          // Cierre de la conexión
                          if(resultado!=null)
                          {

        GestionBaseDeDatos.closeResulset(resultado);
                          }
                          if(consulta!=null)
                          {

        GestionBaseDeDatos.closeRequest(consulta);
                          }
                          if(conexion!=null)
                          {

        GestionBaseDeDatos.closeConnection(conexion);
                          }
                    }
                    catch(Exception ex)
                    {
                          System.out.println("Error en el
cierre de la conexión con la base de datos en la clase
ModeloClienteDAO función eliminarCliente");
                     }
              }

              // Devolver el código de error
              return codigoError;
        }


        // modificar un cliente en la base
        public int modificarCliente(Cliente cliente)
        {
              // Variables
              PreparedStatement consulta=null;
              String consultaString=null;
              int codigoError=0;

              try
              {
                    // Apertura de una conexión
                    conexion=super.getConnection();

                    // Creación de la consulta
                    consultaString="UPDATE cliente set
identificador=?,contrasena=? WHERE idCliente=?";
```

```
       consulta=conexion.prepareStatement(consultaString);
                     consulta.setString(1,
cliente.getIdentificador());
                     consulta.setString(2, cliente.getContrasena());
                     consulta.setInt(3, cliente.getIdCliente());

                     // Se vacía el cliente por seguridad
                     cliente=null;

                     // Ejecución de la consulta
                     codigoError=consulta.executeUpdate();
              }
              catch(Exception e)
              {
                     System.out.println("Error en la consulta
de la clase ModeloClienteDAO función modificarCliente");
              }
              finally
              {
                     try
                     {
                            // Cierre de la conexión
                            if(resultado!=null)
                            {

       GestionBaseDeDatos.closeResulset(resultado);
                            }
                            if(consulta!=null)
                            {

       GestionBaseDeDatos.closeRequest(consulta);
                            }
                            if(conexion!=null)
                            {

       GestionBaseDeDatos.closeConnection(conexion);
                            }
                     }
                     catch(Exception ex)
                     {
                            System.out.println("Error en el
cierre de la conexión con la base de datos en la clase
ModeloClienteDAO función modificarCliente");
                     }
              }

              // Devolver el código de error
              return codigoError;
       }


    // Realizar el mapping relacional hacia objeto
    public Cliente mapperCliente(ResultSet resultado)
    {
       // Variables
       Cliente cliente=new Cliente();

       try
       {
              if (resultado.getString("idCliente")==null)
              {
                     cliente.setIdCliente(0);
              }
              else
```

```
                {

        cliente.setIdCliente(resultado.getInt("idCliente"));
                }

                if (resultado.getString("identificador")==null)
                {
                        cliente.setIdentificador("");
                }
                else
                {

        cliente.setIdentificador(resultado.getString("Identificador"));
                }

                if (resultado.getString("contrasena")==null)
                {
                        cliente.setContrasena("");
                }
                else
                {

        cliente.setContrasena(resultado.getString("contrasena"));
                }
        }
         catch (Exception e)
         {
                //Si se produce un error durante el mapping
de atributos
                cliente=null;
                System.out.println("Error en el mapping de
atributos de un cliente de la clase ModeloClienteDAO, función
mapperCliente");
         }

         // Devolver objeto cliente
         return cliente;
    }

}
```


Lista de clientes – Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda Related Links
http://localhost:8080/ejemplo15/
/manager
Lista de clientes
Agregar un cliente
Identificador:
Contraseña:
Agregar un cliente
ID Identificador Contraseña Gestión
1 jlafosse jerome
2 asoto amelia
3 amartin alejandro
4 palvarez pedro

En resumen

En este capítulo se ha presentado la configuración de la persistencia de objetos Java a través de una base de datos relacional. El modelo de persistencia DAO se ha detallado mediante el ejemplo de gestión de clientes. A partir de ahora, el proyecto dispone de todas las capas de una aplicación profesional Modelo Vista Controlador MVC y permite un mantenimiento y evolución más sencillos mediante la división de los diferentes módulos a través de modelos de diseño adaptados.

Presentación

La configuración de la carga de archivos del cliente en el servidor a menudo es una operación compleja. Para facilitar esta tarea, Struts utiliza la librería *commonsFileUpload* del consorcio Jakarta. Sin la utilización de bibliotecas adaptadas, el usuario debe hacer uso de los métodos de gestión de flujo (`InputStream` y `OutputStream`). La configuración de la carga de archivos requiere un formulario HTML con la etiqueta `<form/>`. El atributo `enctype` de esta etiqueta debe estar posicionado en*multipart/form-data* y el método `post` debe utilizarse como parámetro. Esta declaración permite indicar que el formulario contiene datos multimedia.

```
<form action="miAccion" enctype="multipart/form-data"
method="post">
```

La etiqueta `<form/>` está asociada a un campo *<input/>* de tipo *file*. Este componente HTML permite integrar un botón **examinar** que sirve para seleccionar el archivo que desea enviar al servidor.

```
<form action="miAccion" enctype="multipart/form-data"
method="post">
<input type="file" name="miarchivo"/>
</form>
```

La etiqueta <s:file/>

La configuración de la carga de archivos en Struts requiere el uso del interceptor *fileUpload*. La carga de archivos es progresiva. El formulario indica la acción a la que se debe acudir para la carga, así como el archivo que se va a cargar. La acción de Struts declara un objeto de tipo *java.io.File* utilizado para establecer el vínculo con el archivo enviado. Durante la utilización de una lista de archivos, podemos declarar una colección de archivos e iterar esta lista para cada archivo.

```
<s:form action="uploadAction" enctype="multipart/form-data">
<s:file name="miArchivo" label="Archivo"/>
<s:submit/>
</s:form>
```

A continuación, debemos crear una acción con las propiedades siguientes, así como los descriptores de acceso respectivos:

```
private File miArchivo;
private String miArchivoFileName; // sintaxis nombredelarchivoFileName
private String miArchivoContentType; // sintaxis
nombredelarchivoContentType
```

El interceptor fileUpload

El interceptor *fileUpload* gestiona la carga del o los archivos de la consulta HTTP. Este interceptor posee dos propiedades que permiten gestionar el tamaño máximo del archivo que se va a enviar y los tipos permitidos:

- *maximumSize*: esta propiedad permite limitar el tamaño (en bytes) del archivo que se va a enviar. El tamaño por defecto es 2 megabytes.
- *allowedTypes*: esta propiedad permite precisar los tipos de archivo permitidos.

Para nuestra proyecto, vamos a gestionar la imagen del cliente (avatar) con un sistema de carga de imágenes Web (gif, jpeg y png). Este proyecto *ejemplo16* se desarrolla a partir de la aplicación*ejemplo13*, que permite realizar un listado y agregar clientes. La declaración de la gestión de la carga se encuentra en el archivo *struts.xml*.

Código: 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="true" />

    <package name="ejemplo16" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo16.ClienteAcción" method="listado">
        <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo16.ClienteAcción" method="agregar">
            <interceptor-ref name="fileUpload">
              <param name="maximumSize">102400</param>
              <param
name="allowedTypes">imagen/gif,imagen/jpeg,imagen/png</param>
            </interceptor-ref>
            <interceptor-ref name="basicStack"/>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>
    </package>
</struts>
```

El interceptor fileUpload puede declararse de forma independiente, pero en caso de utilización de otros tipos de acción con en nuestro ejemplo (redirección hacia otra acción), el interceptor basicStack debe declararse tras fileUpload.

La acción *Agregar_Cliente* utiliza el interceptor para la gestión de la carga de archivos y declara un tamaño máximo de archivo de 100 kilobytes, así como las imágenes de tipo gif, jpeg y png permitidas.

Carga única

Vamos a realizar una carga única de archivo para la gestión de la imagen del cliente. La clase JavaBean `Cliente` se modifica ligeramente para gestionar la imagen en forma de cadena de caracteres que representa la ruta del directorio de almacenamiento.

```
Código: ejemplo16.javabeans.Cliente.java
package ejemplo16.javabeans;

@SuppressWarnings("serial")
public class Cliente {

    private int idCliente;
    private String identificador;
    private String contrasena;
    private String imagen;

    public Cliente() {

    }

      public Cliente(int idCliente,String identificador, String
contrasena, String imagen){
      this.idCliente=idCliente;
      this.identificador=identificador;
      this.contrasena=contrasena;
      this.imagen=imagen;
    }

    // descriptores de acceso
}
```

El código de la clase de acción es simple y permite recuperar automáticamente el archivo, copiarlo en el directorio de imágenes de los clientes y aplicar el nombre de la imagen al objeto.

```
Código: ejemplo16.ClienteAccion.java
package ejemplo16;

import java.io.File;
import java.util.List;
import javax.servlet.ServletContext;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo16.javabeans.Cliente;
import ejemplo16.modelo.ClienteModelo;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private Cliente cliente;
    private List<Cliente> listaClientes;
    private File imagen;
    private String imagenFileName;
    private String imagenContentType;

      public File getImagen() {
             return imagen;
      }

      public void setImagen(File imagen) {
             this.imagen = imagen;
      }
```

```
	public String getImagenFileName() {
		return imagenFileName;
	}

	public void setImagenFileName(String imagenFileName) {
		this.imagenFileName = imagenFileName;
	}

	public String getImagenContentType() {
		return imagenContentType;
	}

	public void setImagenContentType(String imagenContentType) {
		this.imagenContentType = imagenContentType;
	}

	public Cliente getCliente() {
		return cliente;
	}

	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

	public List<Cliente> getListaClientes() {
		listaClientes=ClienteModelo.getListaClientes();
		return listaClientes;
	}

	public void setListaClientes(List<Cliente> listaClientes) {
		this.listaClientes = listaClientes;
	}

	// devolver la lista de clientes tras la recuperación
	public String listado()
	{
		listaClientes=ClienteModelo.getListaClientes();
		return SUCCESS;
	}

	// agregar el cliente en el modelo
	public String agregar() throws Exception
	{
		// ubicar el nombre del archivo en el cliente
		if(this.imagenFileName!=null)
		{
			cliente.setImagen(this.imagenFileName);
			// copiar el archivo en el directorio de
la aplicación imagenesclientes
			ServletContext
context=ServletActionContext.getServletContext();
			String
directorioImagenesCliente=context.getRealPath("imagenesclientes");
			File almacenamientoImagen=new
File(directorioImagenesCliente,this.imagenFileName);
			this.imagen.renameTo(almacenamientoImagen);
			ClienteModelo.agregar(this.cliente);
		}


	    return SUCCESS;
	}
}
```

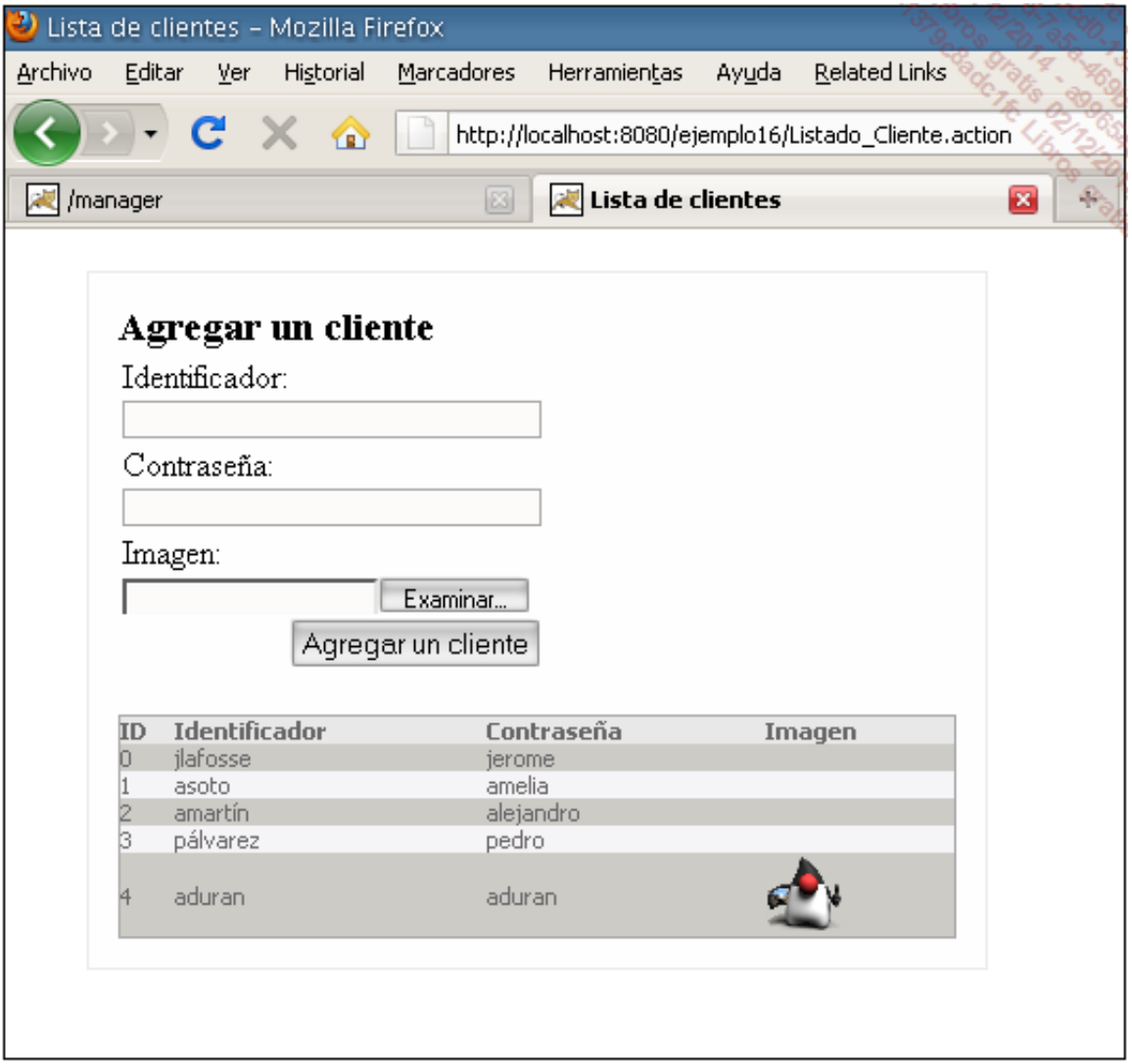
Por último, la vista JSP también se modifica ligeramente para mostrar las imágenes de los clientes si éstas no son nulas.

Código: /jsp/ListadoCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Lista de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los errores siguientes: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<!-- Mensaje de error durante las acciones -->
<s:if test="errorMessages.size()>0">
<div id="mensaje_error">
      <label>Se han producido los siguientes errores de acción:
</label>
      <ul><s:actionerror/></ul>
</div>
</s:if>
<!-- Mensaje de confirmación -->
<s:if test="actionMessages.size()>0">
      <div id="mensaje_informacion">
           <ul><s:actionmessage/></ul>
      </div>
</s:if>
<div id="carta">

      <h3>Agregar un cliente</h3>
      <s:form method="post" action="Agregar_Cliente"
enctype="multipart/form-data">
            <s:textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
            <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
            <s:file name="imagen" id="imagen" label="Imagen"
labelposition="top" cssClass="input"/>
            <s:submit value="Agregar un cliente"/>
      </s:form>
      <table border="0" id="tabla" cellpadding="0"
cellspacing="0">
      <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td><b>Imagen</b></td></tr>
      <s:iterator value="listaClientes" status="linea">
      <s:if test="#linea.odd"><tr class="linea1"></s:if>
      <s:if test="#linea.even"><tr class="linea2"></s:if>
      <td><s:property value="idCliente"/></td>
      <td><s:property value="identificador"/></td>
      <td><s:property value="contrasena"/></td>
      <s:if test="imagen!=null">
            <td><img src="imagenesclientes/${imagen}" width="40"
height="40"/></td>
      </s:if>
      <s:else>
            <td> </td>
      </s:else>
```

```
      </tr>
      </s:iterator>
      </table>
</div>
</body>
</html>
```

A partir de ahora podemos agregar un nuevo cliente y cargar su imagen desde el formulario.



Carga de la imagen del cliente

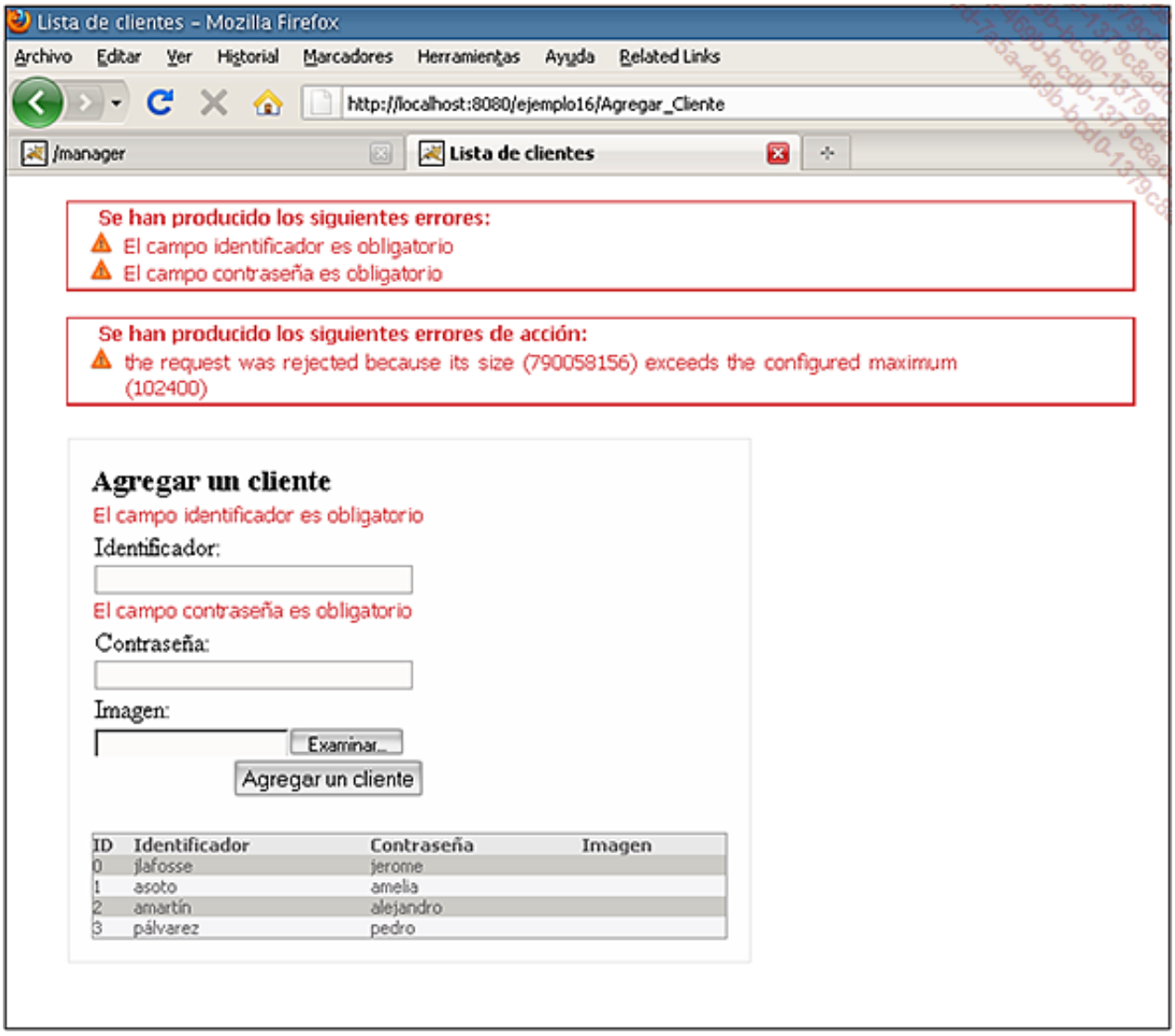
Podemos observar que el archivo de validación *ClienteAccion-Agregar_Cliente-validación.xml* no es gestionado por el interceptor *fileUpload* y que, en consecuencia, las validaciones ya no se implementan. Si deseamos agregar la validación de los campos y la utilización del resultado de tipo*input*, debemos agregar el interceptor *validationWorkflowStack* presente en el archivo *struts-default.xml*y que contiene otros interceptores.

```
<interceptor-stack name="validationWorkflowStack">
   <interceptor-ref name="basicStack"/>
   <interceptor-ref name="validation"/>
   <interceptor-ref name="workflow"/>
</interceptor-stack>
```

```
Código: 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="true" />

    <package name="ejemplo16" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo16.ClienteAccion" method="listado">
            <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo16.ClienteAccion" method="agregar">
              <interceptor-ref name="fileUpload">
              <param name="maximumSize">102400</param>
              <param
name="allowedTypes">imagen/gif,imagen/jpeg,imagen/png</param>
               </interceptor-ref>
              <interceptor-ref name="validationWorkflowStack"/>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
            <result name="input">/jsp/ListadoCliente.jsp</result>
        </action>
    </package>
</struts>
```

Ahora que el interceptor de gestión de las validaciones ya está operativo, vamos a enviar un archivo demasiado pesado (tamaño superior a 100 Kb) y de un tipo incorrecto (que no sea una imagen) a propósito. Observamos que los errores gestionados por el interceptor de carga son de tipo acción<*s:actionerror/*>. Los mensajes mostrados están en inglés y se definen en el archivo *struts-messages.properties* presente en el paquete *org.apache.struts2* del archivo *struts2-core-x.jar*.

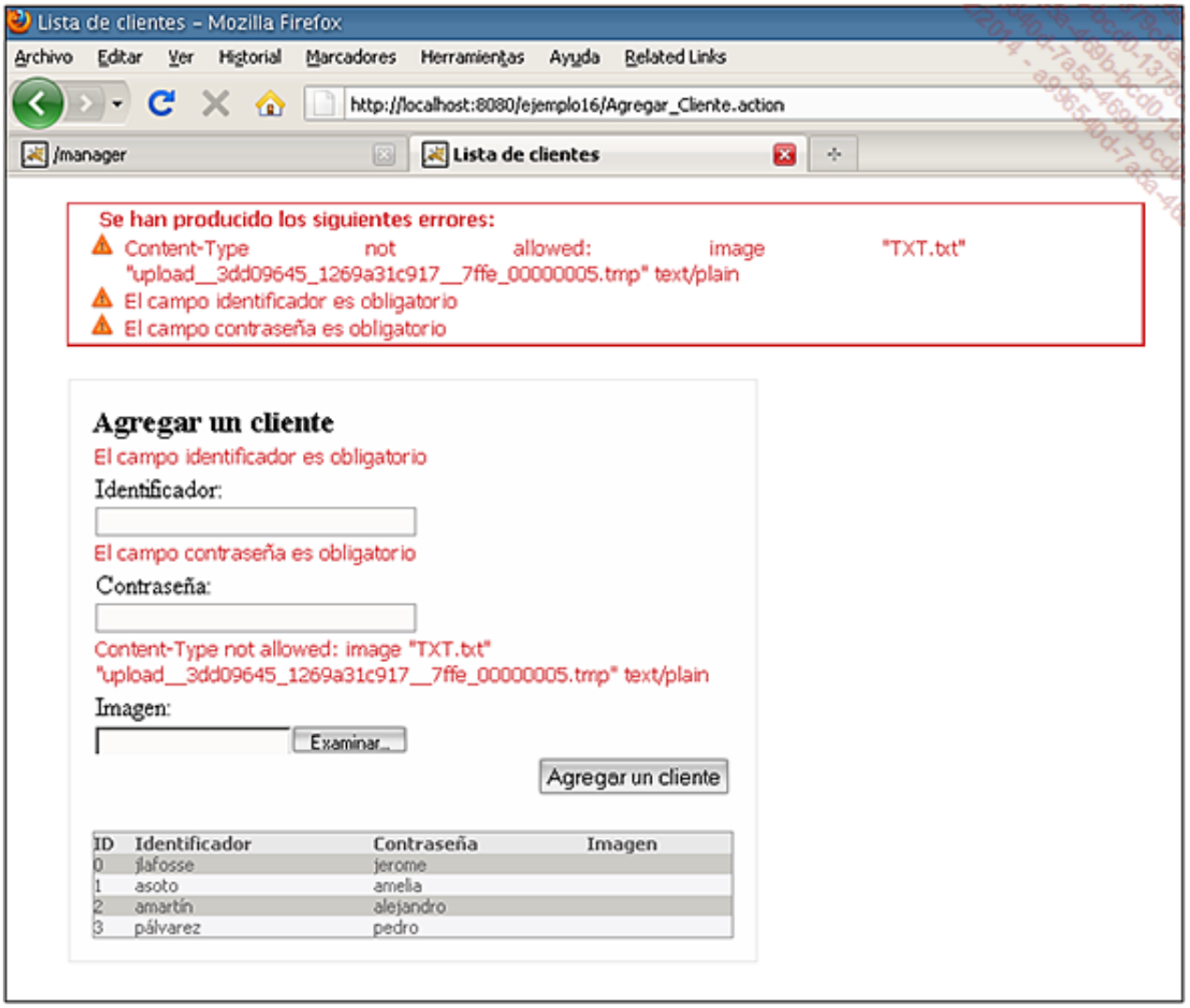


Gestión de los errores de carga de archivos

Para sobrecargar los mensajes mostrados en el seguimiento de los errores, podemos crear un archivo denominado *struts-mensajes.properties* que se encuentre en el paquete de la clase implicada.

```
Código: /WEB-INF/src/struts-mensajes.properties
struts.mensajes.error.uploading=Error durante la carga del archivo
: {0}
struts.mensajes.error.content.type.not.allowed=Error. Tipo de
archivo no permitido
struts.mensajes.error.file.too.large=Error. El archivo
es demasiado grande
```

Con un archivo de tamaño inferior a 100 Kb pero de un formato inadecuado (por ejemplo: un archivo de texto), el mensajes reenviado será el siguiente:

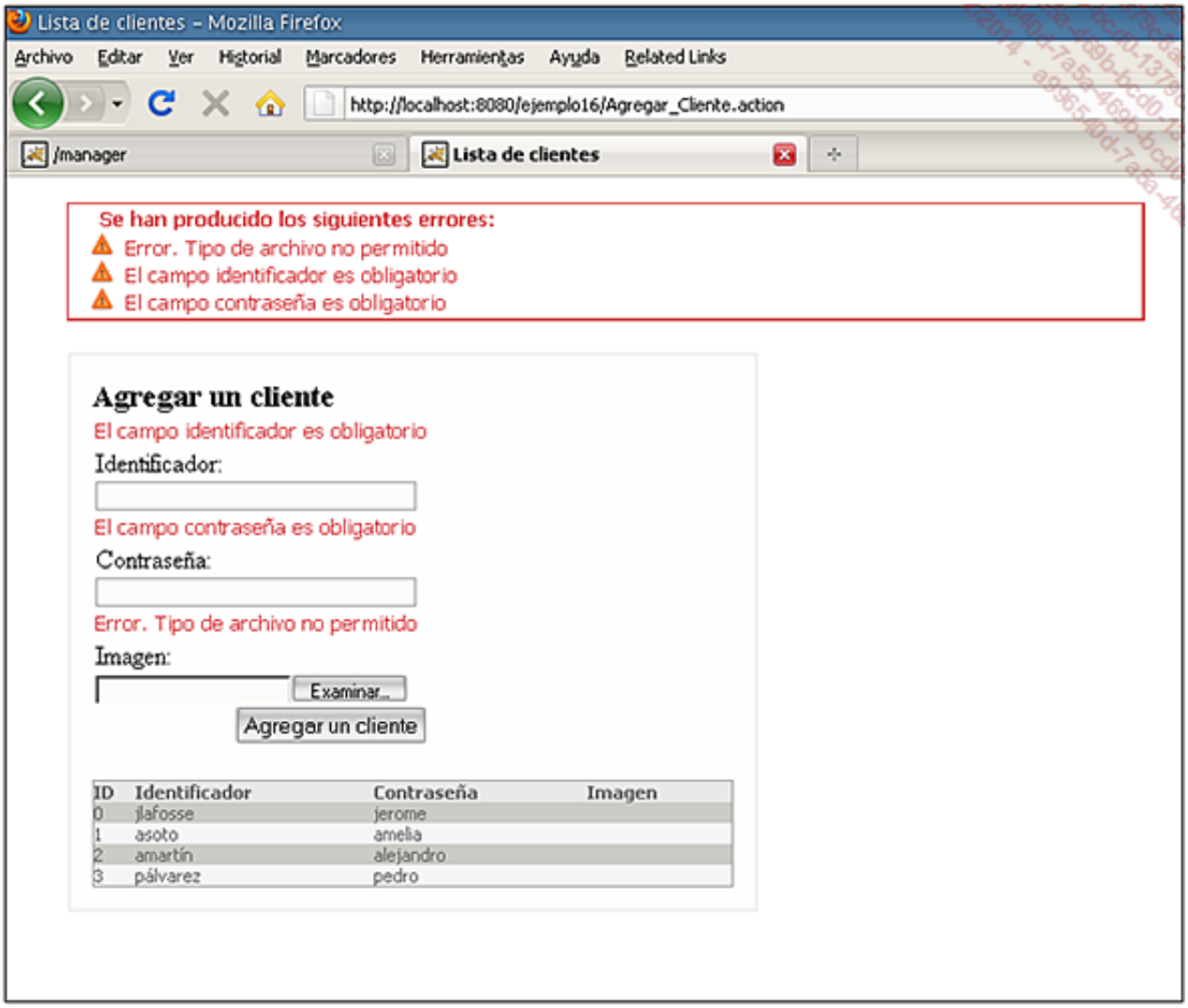


Visualización de mensajes de error adaptados

La verificación del tipo de contenido se ha realizado correctamente, pero los mensajes de error siguen siendo los definidos por defecto en Struts. Para que se utilice nuestro archivo *struts-mensaje.properties*, debemos sobrecargar el archivo de propiedades de Struts creando un archivo*struts.properties* en el directorio */WEB-INF/src* de la aplicación e indicando a través del parámetro*struts.custom.i18n.resources* el archivo que debe utilizarse para los mensajes. Aprovecharemos además para definir el tamaño máximo del archivo que se va a cargar en este archivo de configuración.

```
Código: /WEB-INF/src/struts.properties
struts.multipart.parser=org.apache.struts2.dispatcher.multipart.Ja
kartaMultiPartRequest
struts.multipart.maxSize=102400
struts.custom.i18n.resources=struts-mensajes
```

En caso de gestión del sitio en varios idiomas, será necesario crear tantos archivos struts.mensaje.properties como idiomas, añadiéndole a estos el sufijo de su configuración regional (por ejemplo: struts-mensaje_en.properties).



Gestión de mensajes de error por idioma

Carga múltiple

El interceptor *fileUpload* permite gestionar también diversos archivos del mismo formulario de una sola vez. El principio consiste en crear las etiquetas `<s:file/>` necesarias con nombres idénticos.

```
<s:file name="imagen" id="imagen" label="Imagen1"
labelposition="top" cssClass="input"/>
<s:file name="imagen" id="imagen" label="Imagen 2"
labelposition="top" cssClass="input"/>
```

Los archivos se envían a la clase de acción en forma de tabla y de este modo podrán ser tratados en consecuencia.

```
@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private Cliente cliente;
    private List<Cliente> listaClientes;
    private File[] imagen;
    private String[] imagenFileName;
    private String[] imagenContentType;

      ...

      // agregar el cliente al modelo
      public String agregar() throws Exception
      {

            ServletActionContext.getServletContext();
            String
directorioImagenesCliente=context.getRealPath("imagenesclientes");

            for(int i=0;i<imagen.length;i++)
            {
                  File almacenamientoImagen=new
File(directorioImagenesCliente,this.imagenFileName[i]);
                  this.imagen[i].renameTo(almacenamientoImagen);
            }

            ...

         return SUCCESS;
      }
}
```

Carga en Ajax

La carga de archivos estudiada anteriormente es perfectamente funcional pero no permite visualizar el estado de progreso del archivo enviado. Los archivos de gran tamaño pueden tardar varios minutos en cargarse, dejando al usuario a la espera sin información adicional. Java permite utilizar librerías avanzadas que permiten cargar archivos en segundo plano con la tecnología Ajax y hacer uso del lenguaje JavaScript para informar al usuario sobre el estado de progreso.

Existen numerosos complementos o librerías Ajax Upload más o menos específicos y funcionales. Vamos a utilizar la librería Ajax File Upload Plug-in (*AjaxFileUpload-x.jar*) del proyecto Struts 2 http://www.davidjc.com/ajaxfileupload/demo!input.action

La aplicación de este complemento de Struts 2 requiere la instalación de las librerías siguientes en el*classpath*:

- *ajaxFileUpload-x.jar*: esta librería permite gestionar la carga de archivos.
- *commons-fileupload-x.jar*: esta librería desarrollada por el consorcio Jakarta, permite gestionar la carga de archivos.
- *commons-io-x.jar*: esta librería permite gestionar las entradas/salidas para las cargas.
- *json-lib-x.jar*: esta librería se basa en JSON (*JavaScript Object Notation*), una librería JavaScript que utiliza la tecnología Ajax para el acceso a los datos.

Para aplicar la carga de archivos en modo asíncrono y con una barra de progreso, crearemos un nuevo proyecto, *ejemplo17*. A continuación, debemos agregar dos etiquetas en nuestra vista usuaria, (librería de etiquetas o taglib) para configurar las librerías necesarias y el formulario dinámico.

```
<%@ taglib uri="http://www.davidjc.com/taglibs" prefix="djc"%>
//Para incluir la librería de etiquetas
<djc:head /> //Etiqueta para incluir automáticamente los archivos
JavaScript
```

Debemos precisar el nombre *ajaxFileUploadForm* en el id. y el parámetro *onsubmit* como *false*. A continuación, el nombre del archivo debe ser *upload*. El botón de validación del formulario *<s:submit/>*tiene que poseer el atributo `onclick` con el método Ajax que se debe ejecutar y la barra de progreso, llamada *fileuploadProgress*.

```
Código: /jsp/ImagenCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ taglib uri="http://www.davidjc.com/taglibs" prefix="djc" %>
<html>
<head>
<title>Agregar una imagen</title>
<style type="text/css">@import url(css/estilos.css);</style>
<djc:head />
</head>
<body>
<div id="carta">

      <h3>Agregar una imagen</h3>
      <s:form method="post" action="Uploader_Image"
enctype="multipart/form-data" id="ajaxFileUploadForm"
onsubmit="return false">
            <s:file name="upload" id="upload" label="Imagen"
labelposition="top" cssClass="input" accept="*/*"/>
            <s:submit value="Enviar la imagen"
labelposition="top" onclick="return
davidjc.AjaxFileUpload.initialise(undefined, undefined);"/>
      </s:form>
      <div id="fileuploadProgress">
```

```
            <span id="uploadFilename">Iniciando,
por favor espere...</span>
            <div id="progress-bar"><div id="progress-
bgrd"> </div></div>
            <div id="progress-text"> </div>
            <br/>
      </div>
</div>
</body>
</html>
```

La declaración de la acción también debe adaptarse. El interceptor empleado se denomina*fileUploadStack* y se utiliza un resultado de tipo *httpheader* para la información reenviada en Ajax.

Código: 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="true" />

    <package name="ejemplo17" namespace="/" extends="struts-
default">
        <default-action-ref name="Agregar_Imagen" />

         <action name="Agregar_Imagen">
            <result>/jsp/ImagenCliente.jsp</result>
         </action>

        <action name="Uploader_Image"
class="ejemplo17.ClienteAccion">
              <interceptor-ref name="fileUploadStack">
              </interceptor-ref>
              <result name="success" type="httpheader">
                        <param name="status">200</param>
                    </result>

              <interceptor-ref name="validationWorkflowStack"/>
            <result name="success"
type="redirectAction">Agregar_Imagen</result>
         </action>

    </package>
</struts>
```

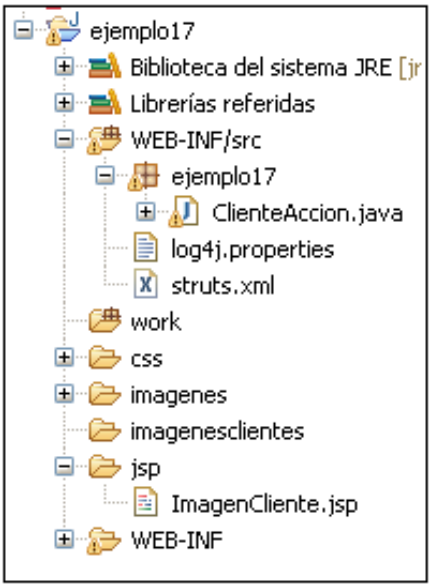
El código de la clase de acción es simple, posee los diferentes atributos necesarios para la gestión de la carga del archivo. Los métodos `getUpload()`, `getUploadContentType()` y `getUploadFileName()` se utilizan para realizar las acciones de carga.

Código: ejemplo17.ClienteAccion.java

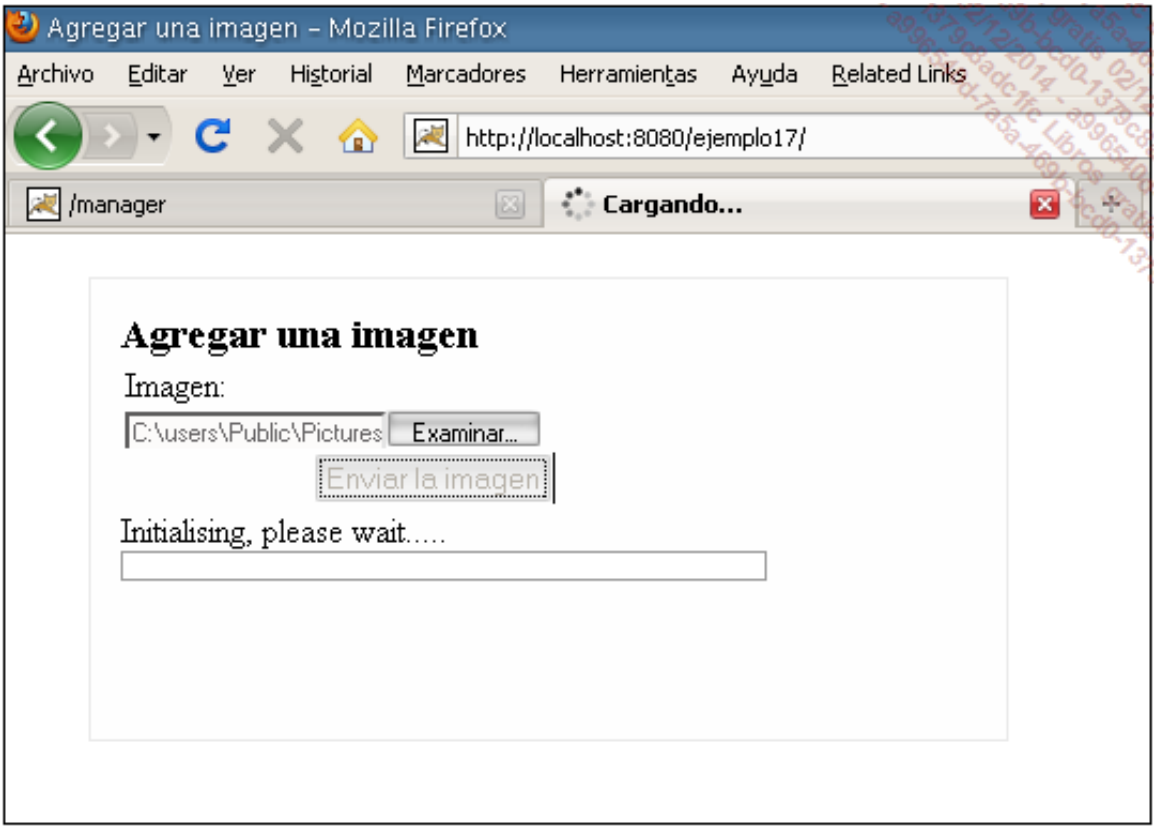
```
package ejemplo17;

import java.io.File;
import javax.servlet.ServletContext;
import org.apache.struts2.ServletActionContext;
import com.davidjc.ajaxfileupload.action.FileUpload;
import com.opensymphony.xwork2.Action;
```

```
@SuppressWarnings("serial")
public class ClienteAccion extends FileUpload {

    private File imagen;
    private String imagenFileName;
    private String imagenContentType;

      // agregar el cliente al modelo
      public String execute()
      {
             this.imagen=this.getUpload();
             this.imagenContentType=this.getUploadContentType();
             this.imagenFileName=this.getUploadFileName();

             // ubicar el nombre del archivo en el directorio
             if(this.imagen!=null)
             {
                    // copiar el archivo en el directorio de
la aplicación imagenesclientes
                    ServletContext
context=ServletActionContext.getServletContext();
                    String
directorioImagenesCliente=context.getRealPath("imagenesclientes");
                    File almacenamientoImagen=new
File(directorioImagenesCliente,this.imagenFileName);
                    this.imagen.renameTo(almacenamientoImagen);
             }

         return Action.SUCCESS;
      }

}
```

Nuestra aplicación *ejemplo17* funciona y podemos comprobarla con un archivo de gran tamaño con el fin de visualizar la carga asíncrona y la barra de progreso.



Árbol del proyecto ejemplo17



Carga asíncrona de archivos en Ajax

En resumen

En este capítulo se ha detallado la configuración de la gestión de cargas de archivos, del cliente en el servidor con en framework de Struts. El primer ejemplo explica en detalle la aplicación de esta técnica a partir de la imagen del formulario del cliente. El segundo párrafo le introduce a la carga múltiple y por último, la tercera parte le permite implementar la carga de archivos asíncronos con las tecnologías JavaScript y Ajax.

Presentación

La aplicación de la descarga de archivos desde el servidor al cliente se ofrece de manera estándar con el framework de Struts. El envío dinámico de archivos se utiliza, por ejemplo, en las fichas de productos o para la gestión del CV de los clientes. La descarga de archivos funciona en forma de flujo y por ello puede reenviar un vídeo o imagen directamente al navegador del cliente. Esta técnica también se utiliza en ocasiones para proteger el acceso a contenidos confidenciales y facilitar el flujo tras autenticación o descifrado.

El envío de un archivo desde el servidor al navegador del cliente se realiza utilizando el atributo`Content-Type` en el encabezado HTTP, en función del tipo de archivo que deba enviarse. El navegador también utiliza los atributos `Content-Disposition` y el parámetro*attachment;filename=nombredelarchivo* (a continuación este nombre de archivo se utilizará para guardar el contenido).

Desde el punto de vista de la programación, el archivo se lee del lado servidor a través de la clase`FileInputStream` y se envía al navegador con una instancia de la clase `OutputStream`.

El código siguiente permite utilizar esta función en un Servlet Java EE:

```
FileInputStream flectura=new FileInputStream(archivo);
BufferedInputStream blectura=new BufferedInputStream(flectura);
byte[] bytes=new byte[blectura.available()];
response.setContentType(contentType);
OutputStream salida=response.getOutputStream();
blectura.read(bytes);
salida.write(bytes);
```

Resultado stream

Struts propone utilizar el resultado de tipo *stream* para la descarga de archivos. Durante la implementación del resultado de tipo *stream*, ya no es obligatorio el uso de una página JSP para el resultado, ya que el flujo se envía directamente al navegador.

Vamos a proponer un nuevo proyecto, *ejemplo18*, que nos permitirá descargar la imagen del cliente mostrándola directamente en el navegador en forma de flujo o descargándola directamente de manera forzada. Esta diferencia se establece a través del parámetro *Content-Type* del encabezado. El parámetro *Content-Type imagen/png* obliga al navegador a mostrar la imagen y el parámetro *Content-Type aplicación/byte-stream* fuerza la descarga de la misma.

El resultado de tipo *stream* de Struts dispone de los siguientes parámetros:

- *inputName*: este parámetro permite precisar la función de la clase de acción que va a devolver el objeto de tipo *InputStream*. Por tanto, la clase de acción deberá declarar el setter asociado.
- *bufferSize*: este parámetro permite precisar el tamaño del búfer utilizado para leer los datos que deben enviarse.
- *contentType*: este parámetro permite precisar el tipo de respuesta que debe reenviarse al encabezado del paquete HTTP.
- *contentLength*: este parámetro permite precisar el tamaño de la respuesta reenviada.
- *contentDisposition*: este parámetro permite precisar el nombre del objeto que se mostrará en el navegador o se guardará en el disco del cliente.

Para nuestro proyecto *ejemplo18*, presentaremos el archivo *struts.xml* con las declaraciones de las dos acciones y los parámetros asociados. Las dos acciones son prácticamente iguales, con la diferencia de que el parámetro *contentType* permite a la acción *Mostrar_Imagen.action* devolver la imagen al navegador y a la acción *Descargar_Imagen.action* forzar el resultado en flujo de bytes y por tanto la descarga.

```
Código : 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="true" />

    <package name="exemple18" namespace="/" extends="struts-
default">
        <default-action-ref name="Gestionar_Imagen" />

         <action name="Gestionar_Imagen">
            <result>/jsp/ImagenCliente.jsp</result>
         </action>

        <action name="Mostrar_Imagen"
class="ejemplo18.DescargarAccion">
            <result name="success" type="stream">
                  <param name="inputName">inputStream</param>
                  <param name="contentType">imagen/png</param>
                  <param
name="contentDisposition">filename="elnombredemiimagen.png"</param>
                  <param name="bufferSize">2048</param>
            </result>
        </action>

        <action name="Descargar Imagen"
```

```
class="ejemplo18.DescargarAccion">
            <result name="success" type="stream">
                  <param name="inputName">inputStream</param>
                  <param name="contentType">aplicación/byte-
stream</param>
                  <param
name="contentDisposition">filename="elnombredemiimagen.png"</param>
                  <param name="bufferSize">2048</param>
            </result>
        </action>

    </package>
</struts>
```

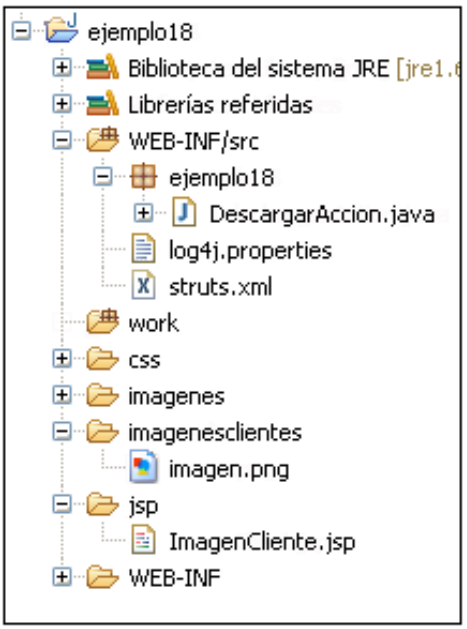
La clase de acción contiene el setter para la ruta del archivo que se va a manipular y el método `getInputStream()` asociado al parámetro *inputName*, que permite devolver el flujo.

```
Código: ejemplo18.DescargarAccion.java
package ejemplo18;

import java.io.InputStream;
import javax.servlet.ServletContext;
import org.apache.struts2.util.ServletContextAware;
import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class DescargarAccion extends ActionSupport implements
ServletContextAware{

    private String rutaArchivo;
    private ServletContext servletContext;

    public void setServletContext(ServletContext servletContext) {
        this.servletContext=servletContext;
    }

    public void setRutaArchivo(String rutaArchivo) {
        this.rutaArchivo = rutaArchivo;
    }

    public InputStream getInputStream() throws Exception {
        return servletContext.getResourceAsStream(rutaArchivo);
    }
}
```

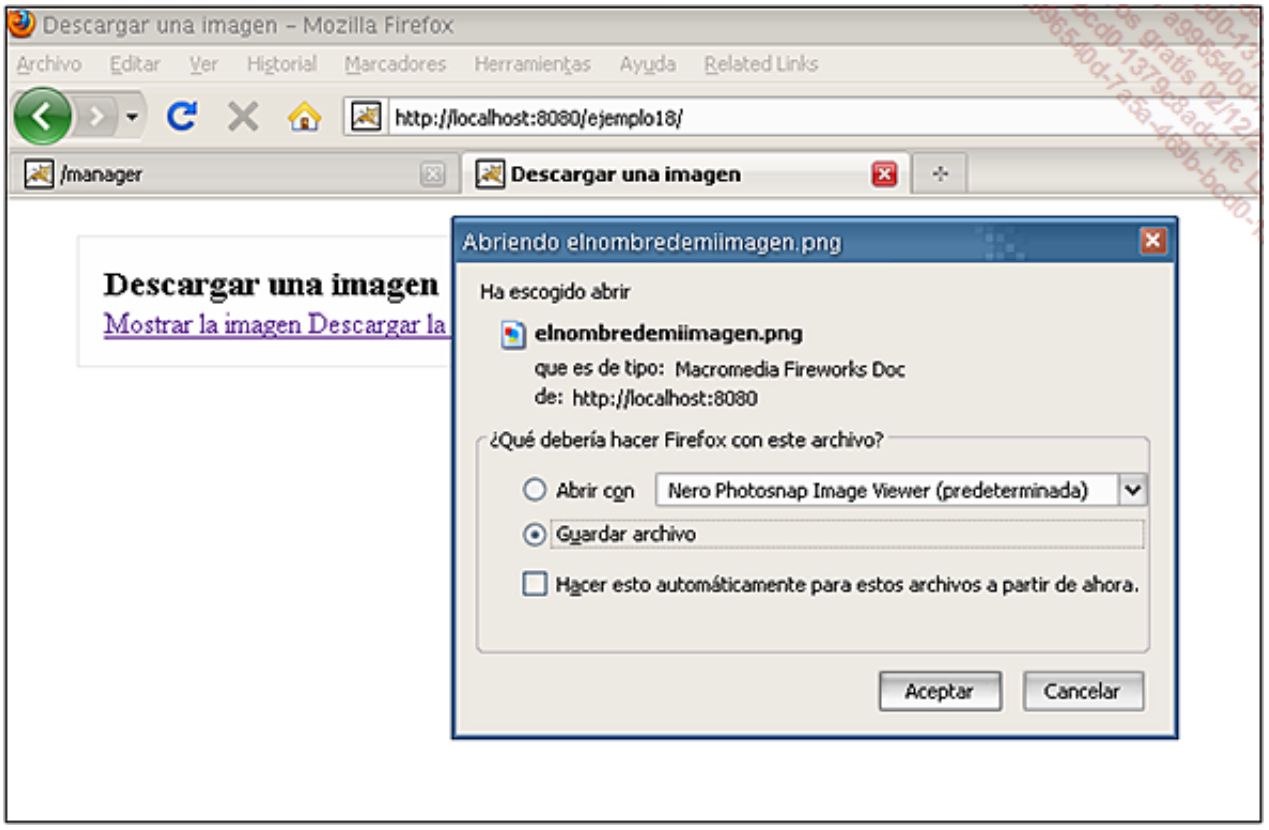
La vista JSP permite establecer los dos vínculos adaptados con la ruta del archivo que va a mostrarse o descargarse como parámetro. La etiqueta `<img/>` de inserción de imagen permite mostrar el contenido de la imagen de manera dinámica a partir de la acción. Este principio se utiliza habitualmente para proteger contenidos.

```
Código: /jsp/ImagenCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Descargar una imagen</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Descargar una imagen</h3>
      <a href="Mostrar_Imagen.action?
rutaArchivo=/imagenesclientes/imagen.png">
```

```
      Mostrar la imagen
      </a>

      <a href="Descargar_Imagen.action?
rutaArchivo=/imagenesclientes/imagen.png">
      Descargar la imagen
      </a>
      <br/>
      <img src="Mostrar_Imagen.action?
rutaArchivo=/imagenesclientes/imagen.png" align="bottom"/>
</div>
</body>
</html>
```



Árbol del proyecto ejemplo18



Descarga de un archivo en bytes

Descarga dinámica de archivos

La aplicación anterior permite mostrar cómo forzar las descargas o devolver flujos de bytes directamente accesibles por el navegador. Vamos a desarrollar una nueva aplicación, *ejemplo19*, que permita descargar los archivos de origen del proyecto o mostrar el código en forma de contenido textual en el navegador.

El archivo de configuración de la aplicación *struts.xml* contiene la acción *Mostrar_Archivo.action*, que permite descargar los archivos en forma de documentos de texto (*contentType=text/plain*). Esta declaración está asociada al método `inputStreamMostrar()`. La segunda acción, denominada*Descargar_Mostrar.action*, está asociada al método de la acción `inputStreamDescargar()`. Este método, más complejo que el anterior, permite precisar de forma dinámica el nombre del archivo que se va a descargar (atributo `contentDisposition`) en función del nombre recibido para beneficiarse de un sistema genérico y verificar que éste existe.

```
Código: 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="true" />

    <package name="ejemplo19" namespace="/" extends="struts-
default">
        <default-action-ref name="Gestionar_Archivo" />

         <action name="Gestionar_Archivo">
            <result>/jsp/ArchivoCliente.jsp</result>
         </action>

        <action name="Mostrar_Archivo"
class="ejemplo19.DescargarAccion">
              <result name="success" type="stream">
                    <param name="inputName">inputStreamMostrar</
param>
                    <param name="contentType">text/plain</param>
              </result>
         </action>

        <action name="Descargar_Archivo"
class="ejemplo19.DescargarAccion">
              <result name="success" type="stream">
                    <param
name="inputName">inputStreamDescargar</param>
              </result>
         </action>

    </package>
</struts>
```

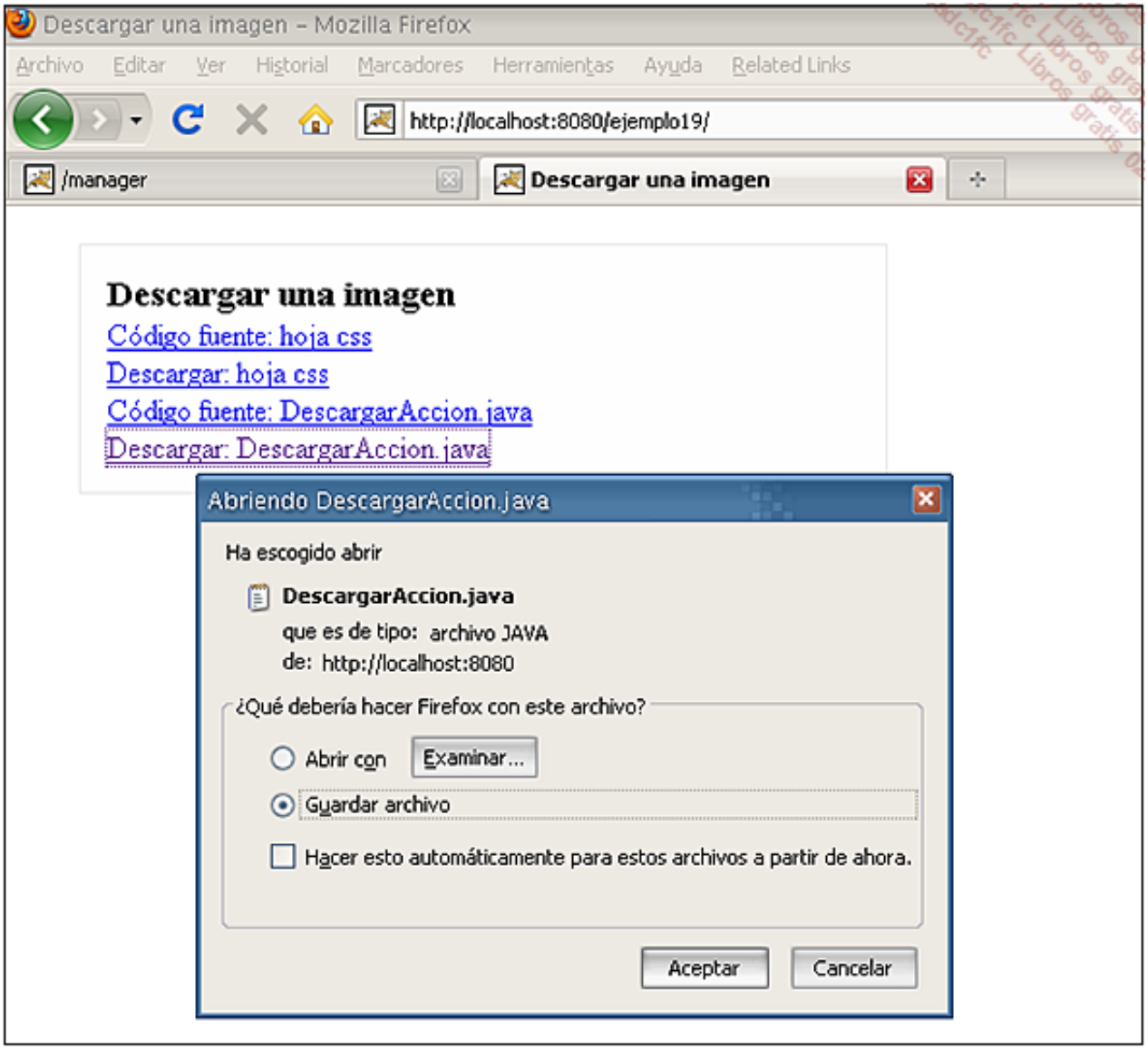
```
Código: ejemplo19.DescargarAccion.java
package ejemplo19;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;
```

```
import javax.servlet.ServletContext;
import org.apache.struts2.dispatcher.StreamResult;
import org.apache.struts2.util.ServletContextAware;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.Result;
import com.opensymphony.xwork2.ActionContext;

@SuppressWarnings("serial")
public class DescargarAccion extends ActionSupport implements
ServletContextAware{

    private String rutaArchivo;
    private String directorioArchivo;
    private ServletContext servletContext;

      public void setServletContext(ServletContext
servletContext) {
            this.servletContext=servletContext;
      }

      public void setRutaArchivo(String rutaArchivo) {
            this.rutaArchivo = rutaArchivo;
      }

      public void setDirectorioArchivo(String directorioArchivo)
{
            this.directorioArchivo = directorioArchivo;
      }

      // Mostrar un archivo (contentType=text/plain)
      public InputStream getInputStreamMostrar() throws Exception {

            return
servletContext.getResourceAsStream(rutaArchivo);
      }

      // Descargar un archivo (contentType=aplicación/byte-
stream)
      public InputStream getInputStreamDescargar() throws
Exception {

            String
directorio=servletContext.getRealPath(this.directorioArchivo);
            File archivo=new
File(directorio,this.rutaArchivo);

      if(archivo.exists())
      {
            Result
result=ActionContext.getContext().getActionInvocation().getResult(
);
            if(result!=null && result instanceof StreamResult)
            {
                  StreamResult stream=(StreamResult)result;


stream.setContentDisposition("filename="+archivo.getName());
                  stream.setContentType("aplicación/byte-
stream");
            }
            try
            {
                  return new FileInputStream(archivo);
            }
            catch(Exception e)
```

```
            {
                  System.out.println(e);
            }
      }

      return null;
    }
}
```

Código: /jsp/ArchivoCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Descargar una imagen</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Descargar una imagen</h3>

      <a href="Mostrar_Archivo.action?
rutaArchivo=/css/estilos.css">
      C&oacute;digo fuente: hoja css
      </a><br/>
      <a href="Descargar_Archivo.action?
directorioArchivo=/css&rutaArchivo=estilos.css">
      Descargar: hoja css
      </a><br/>

      <a href="Mostrar_Archivo.action?rutaArchivo=/WEB-
INF/src/ejemplo19/DescargarAccion.java">
     Código fuente: DescargarAccion.java
     </a><br/>
     <a href="Descargar_Archivo.action?directorioArchivo=/WEB-
INF/src/ejemplo19/&rutaArchivo=DescargarAccion.java">
     Descargar: DescargarAccion.java
     </a>
</div>
</body>
</html>
```



Visualización y descarga dinámica de archivos

En resumen

En este capítulo se ha presentado la aplicación de la gestión de las descargas de archivos desde el servidor al cliente con la ayuda del framework de Struts. El primer ejemplo presenta la aplicación del resultado *stream* y la declaración de las descargas en el archivo de configuración *struts.xml*. Se han detallado las dos formas de descarga, a saber, la descarga en forma de flujo para la visualización en el navegador y la descarga forzada para el almacenamiento en el disco duro. El último ejemplo, más avanzado, permite gestionar los dos tipos de descarga de forma dinámica y genérica con la ayuda de parámetros y tratamientos de la clase de acción.

Presentación

La carga o visualización de una página durante un procesamiento complejo puede incluso llevar varios minutos. Por lo tanto, en ocasiones es necesario mostrar el progreso de la carga, pero esta no es una tarea sencilla de programación de Internet. El framework ofrece para ello un interceptor dedicado que simula la carga y el proceso de los procesamientos.

El funcionamiento del interceptor *execandWait* se basa en una sesión y unos parámetros actualizados automáticamente para informar sobre el progreso del procesamiento. El principio reposa sobre un proceso o thread que se ejecuta en segundo plano y devuelve información al usuario en la acción actual para informarle sobre el tiempo restante para que finalice totalmente la ejecución.

La aplicación de la carga de páginas se realiza siguiendo los siguientes puntos:

- Definición de la etiqueta HTML `<meta/>` que permite actualizar la página.
- Definición de la acción con el interceptor *execAndWait* en el archivo *struts.xml*.
- Definición del método de acción y de la función *getComplete()*.

La etiqueta `<meta/>` permite recargar la página actual (u otra página) cada x segundos. Por ejemplo, es habitual establecer que se actualice la página cada segundo.

```
<meta http-equiv="refresh"
content="1;url=/ejemplo20/Carga.action"/>
```

El interceptor *execAndWait* contiene tres parámetros para gestionar la ejecución. El parámetro*threadPriority* permite asignar la prioridad al thread. El parámetro *delay* determina el número de milisegundos de espera antes de enviar el resultado al cliente (el valor por defecto es 0). Por último, el parámetro *delaySleepInterval* especificará el número de milisegundos del thread principal que efectúa la verificación al final de la ejecución. El método `getComplete()` es obligatorio cuando se aplica el interceptor *execAndWait* y se gestiona el progreso de carga, permite devolver un entero que representa el progreso del proceso.

Aplicación

El proyecto *ejemplo20* permite visualizar el progreso de la carga de la página a partir de una acción simple que efectúa una espera simulando un procesamiento largo. El resultado se reenvía a la página*/jsp/Cargador.jsp* cada segundo si el parámetro *delay* se ha establecido en 1000 milisegundos. La etiqueta `<meta/>` vuelve a cargar la acción en segundo plano.

El resultado llamado *wait* permite especificar la página que se mostrará durante la espera. El resultado ejecutado en caso de éxito se determina mediante el tributo `name`.

El servicio de espera es funcional, pero muestra una barra de espera sin información sobre el tiempo de procesamiento, se utiliza este principio para informar al usuario, pero en ningún caso para mostrar el progreso del procesamiento efectuado.

```
Código : 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="true" />

    <package name="ejemplo20" namespace="/" extends="struts-
default">
        <default-action-ref name="Cargador" />

         <action name="Cargador" class="ejemplo20.CargadorAccion">
              <interceptor-ref name="defaultStack"/>
              <interceptor-ref name="execAndWait">
                    <param name="delay">1000</param>
              </interceptor-ref>
              <result name="wait">/jsp/Cargador.jsp</result>
              <result name="success">/jsp/Completado.jsp</result>
         </action>

    </package>
</struts>
```

```
Código: ejemplo20.Cargador.action
package ejemplo20;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class CargadorAccion extends ActionSupport {

      public String execute() {

            System.out.println("En la acción");
      try
      {
            Thread.sleep(10000);
      }
      catch(Exception e)
      {
            System.out.println(e);
      }
```

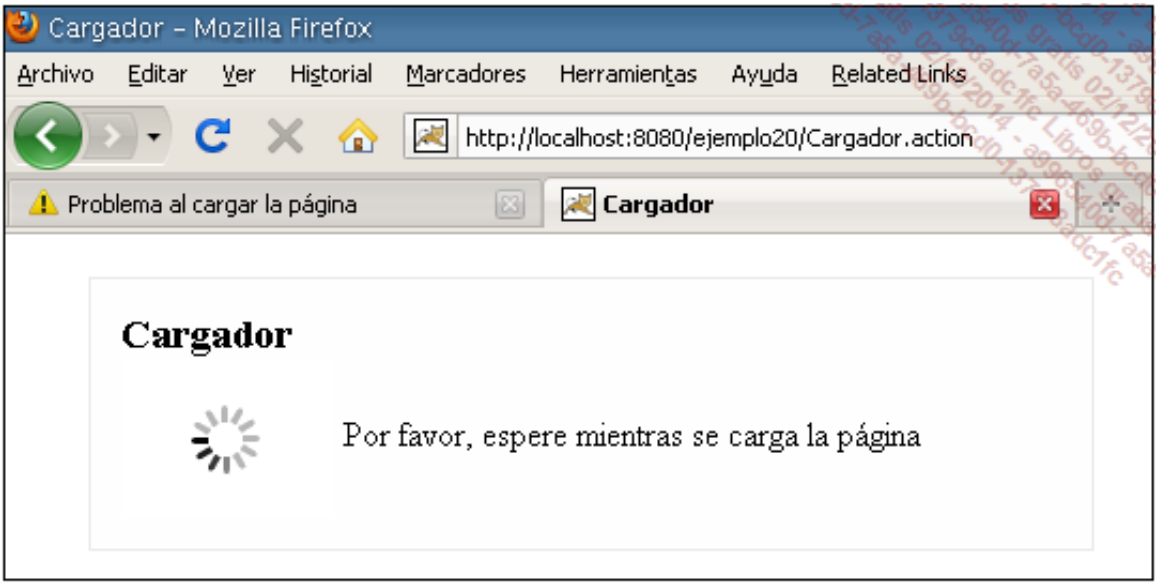
```
        return SUCCESS;
    }
}
```

Código: /jsp/Cargador.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Cargador</title>
<meta http-equiv="refresh"
content="1;url=/ejemplo20/Cargador.action"/>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Cargador</h3>
      <img src="imagenes/cargacircular.gif"
align="absmiddle"/> Por favor, espere mientras se carga la
p&aacute;gina
</div>
</body>
</html>
```

Código: /jsp/Completado.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Cargador completado</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Carga completa</h3>
      Se ha completado el procesamiento
</div>
</body>
</html>
```



Progreso de la carga

El segundo *ejemplo21* utiliza el método `getComplete()` y la propiedad *complete* en la clase de acción para mostrar el progreso de la carga al usuario. La clase de acción `Cargador.action`, así como la vista JSP */jsp/Cargador.jsp* se modifican para procesar el parámetro *complete*.

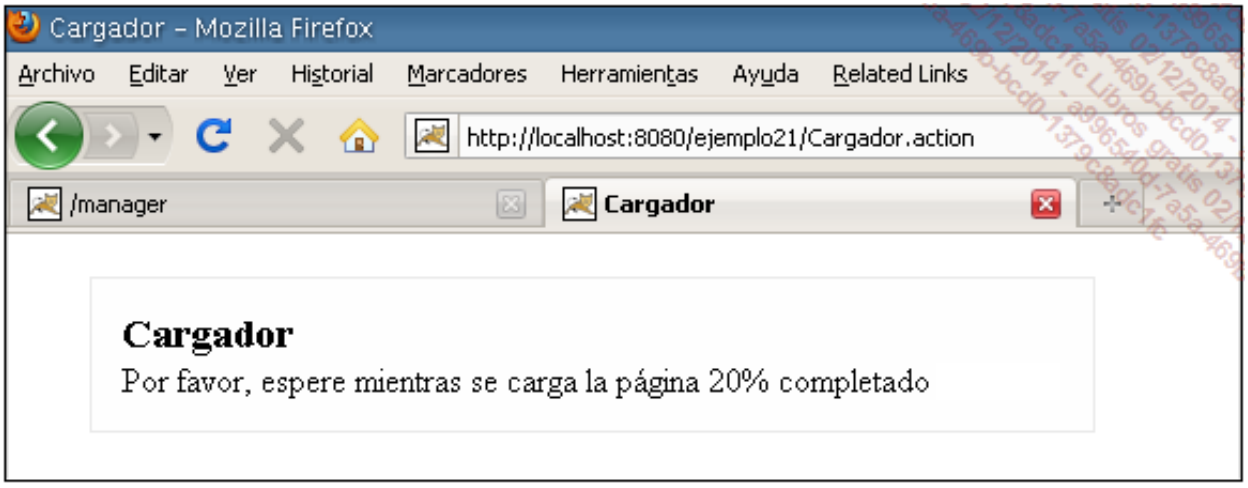
El parámetro *complete* se incrementa en 10 cada segundo, sabiendo el tiempo de espera es de 10 segundos, el tiempo de carga será entonces correcto y preciso. Por supuesto, esta cifra deberá ajustarse y no es muy precisa, pero permite indicar el progreso al usuario.

Código: ejemplo21.Cargador.action

```
package ejemplo21;

import com.opensymphony.xwork2.ActionSupport;

@SuppressWarnings("serial")
public class CargadorAccion extends ActionSupport {

      private int complete=0;

      public String execute() {

            System.out.println("En la acción");
      try
      {
            Thread.sleep(10000);
      }
      catch(Exception e)
      {
            System.out.println(e);
      }

      return SUCCESS;
    }

      public int getComplete() {
             complete+=10;
             return complete;
      }
}
```

Código: /jsp/Cargador.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Cargador</title>
<meta http-equiv="refresh"
content="1;url=/ejemplo21/Cargador.action"/>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Cargador</h3>
      Por favor, espere mientras se carga la p&aacute;gina
      <s:property value="complete"/>% completado
</div>
</body>
</html>
```



Carga de archivos y progreso

En resumen

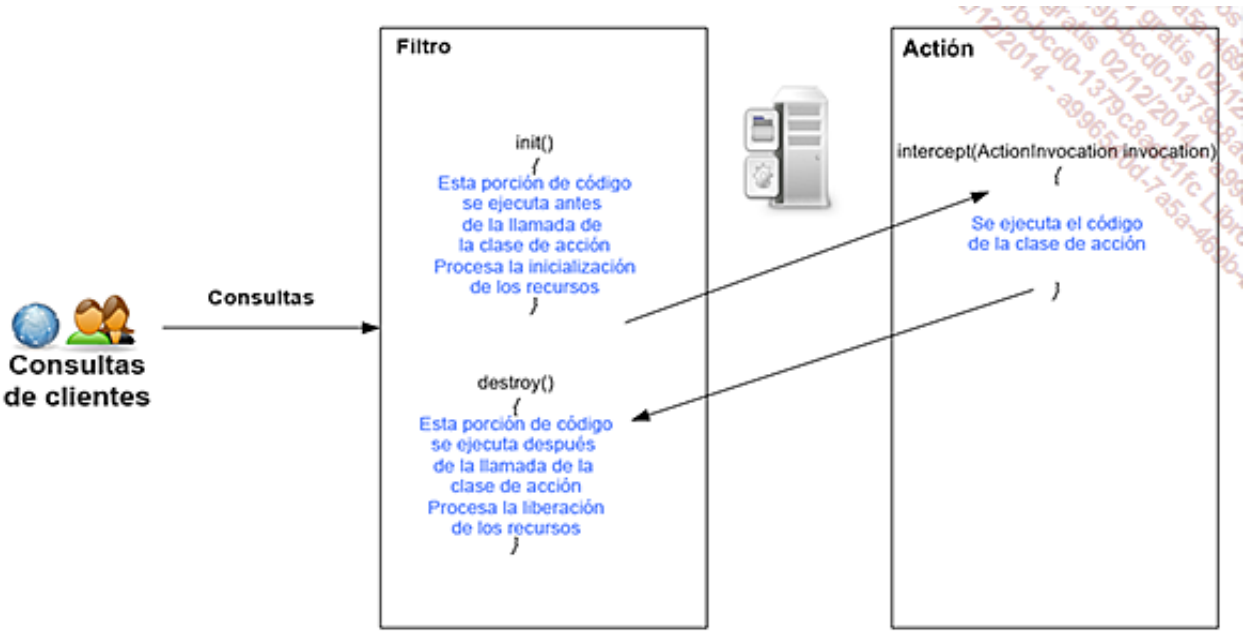
En este capítulo se ha explicado cómo gestionar la carga de páginas con la ayuda del interceptor*execAndWait*. El objetivo de este interceptor es hacer esperar al usuario y proporcionar información sobre los procesamientos realizados en segundo plano. El servicio incluido por defecto permite llamar a una acción en intervalos de tiempo regulares y ofrecer información acerca del progreso gracias a un parámetro dedicado.

Presentación

Struts se basa en una lista de interceptores encargados de prestar servicios específicos para la gestión de los parámetros, la validación de formularios o incluso la carga de archivos. Estos interceptores utilizan el principio de los filtros para el ciclo de vida del proceso. La mayoría de los interceptores que incluye Struts son suficientes para las aplicaciones profesionales. Sin embargo, en ocasiones es necesario desarrollar un interceptor para un servicio determinado.

Un interceptor no es ni más ni menos que una clase Java que implementa la interfaz*com.opensymphony.xwork2.Interceptor*. Esta interfaz ofrece las mismas funciones que un filtro, es decir las firmas de los métodos `init()`, `destroy()`, pero añade una nueva llamada`intercept(ActionInvocation invocation)`. Se llama al método `init()` antes de que se haya creado el interceptor para inicializar de los recursos y únicamente cuando la aplicación esté cargada. Se llama al método `intercept()` cada vez que una acción se envía al interceptor realizar el procesamiento de las operaciones. Se llama al método `destroy()` después de que se haya ejecutado el interceptor para liberar los recursos y únicamente cuando la aplicación esté descargada.

Así, el framework llama al método `intercept()` de cada interceptor declarado por la acción. El objeto instancia de la clase `ActionInvocation` representa el estado de la acción y le permite recuperar los objetos *Action* y *Result* asociados. La clase `AbstractInterceptor` implementa la interfaz *Interceptor* y proporciona una implementación estándar de los métodos `init()` y `destroy()`.



Funcionamiento de los filtros Java

Escribir un interceptor

Para aplicar el desarrollo de la creación de un interceptor, vamos a crear una herramienta de gestión de auténticación del cliente, para el acceso a las acciones del usuario. Los interceptores personales se utilizan principalmente para la auténticación, la conexión al SGBD (mediante un pool JNDI) o incluso el difrado/descifrado de datos.

El proyecto *ejemplo22* utiliza la clase `ejemplo22.AutenticacionInterceptor.java` como interceptor encargado de gestionar la autenticación para el acceso a la acción *Proteger.action*.

El archivo de configuración *struts.xml* contiene la definición del interceptor con la etiqueta`<interceptors/>` y la referencia a la clase `ejemplo22.AutenticacionInterceptor.java`. Se declaran cuatro acciones, la accion *Inicializar.action* permite mostrar la página JSP para hacer una lista de los recursos.

La acción *Proteger.action* utiliza el interceptor declarado mediante la etiqueta `<interceptor-ref/>` y proporciona dos resultados. Por último, existen dos acciones dedicadas a la gestión de la autenticación la etapa de conexión con la colocación de dos paramétros para la información de conexión (*identificadorPredeterminado* y *contrasenaPredeterminada*) y la etapa de desconexión.

```
Código : 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="true" />

    <package name="exemple21" namespace="/" extends="struts-
default">

      <interceptors>
      <interceptor name="autenticacionInterceptor"
class="ejemplo22.AutenticacionInterceptor">
           <param name="identificadorPredeterminado">jlafosse</param>
           <param name="contrasenaPredeterminada">jerome</param>
      </interceptor>
      </interceptors>

        <default-action-ref name="Inicializar" />

         <action name="Inicializar">
          <result>/jsp/Lista.jsp</result>
         </action>

         <action name="Proteger">
            <interceptor-ref name="createSession"/>
            <interceptor-ref name="defaultStack"/>
              <interceptor-ref
name="autenticacionInterceptor"/>
               <result
name="autenticacion">/jsp/Autenticacion.jsp</result>
             <result>/jsp/Recurso.jsp</result>
         </action>

         <action name="Conectar"
class="ejemplo22.AutenticarAccion" method="conectar">
             <param name="identificadorPredeterminado">jlafosse</param>
                  <param name="contrasenaPredeterminada">jerome</param>
                  <result
```

```
name="input">/jsp/Autenticacion.jsp</result>
                    <result type="redirectAction">Proteger</result>
         </action>

         <action name="Desconectar"
class="ejemplo22.AutenticarAccion" method="desconectar">
                    <result type="redirectAction">Inicializar</result>
         </action>

    </package>
</struts>
```

La clase `AutenticarAccion.java` permite verificar la autenticación de los clientes en función de los parámetros predeterminados, declarados en el archivo de configuración de la aplicación *struts.xml*. En caso de éxito, se guardará un parámetro llamado *autenticacion* en caso contrario, se volverá a mostrar el formulario de autenticación.

Código: ejemplo22.AutenticarAccion.java

```
package ejemplo22;

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

@SuppressWarnings("serial")
public class AutenticarAccion extends ActionSupport implements
SessionAware {

      private String identificador;
      private String contrasena;
      private String identificadorPredeterminado;
      private String contrasenaPredeterminada;
      private Map<String,Object> sessionMap;

      public void setSession(Map<String,Object> map)
      {
            this.sessionMap=map;
      }

      // descriptores de acceso

      public String conectar() {

            // verificar si el identificador y la contraseña son
correctos
            if(identificador!=null && contrasena!=null)
            {
                  if(identificador.equals(identificadorPredeterminado) &&
contrasena.equals(contrasenaPredeterminada))
                  {
                        // autenticación correcta,
guardar el valor en la sesión
                        this.sessionMap.put("autenticacion",
true);
                        return SUCCESS;
                   }
             }

           return INPUT;
    }

    public String desconectar() {
            // vaciar la sesión del usuario
            this.sessionMap.clear();
```

```
            return SUCCESS;
      }
}
```

La clase del interceptor *AutenticacionInterceptor.java* utiliza estos tres métodos y los parámetros de la sesión para verificar si el usuario puede acceder a la acción y a las páginas JSP afectadas. Los desarrolladores podrán, de este modo, añadir el sistema de autenticación para la totalidad de un servicio utilizando el siguiente código:

```
<action name="action">
  ...
  <interceptor-ref name="autenticacionInterceptor"/>
  ...
</action>
```

```
Código: ejemplo22.AutenticacionInterceptor.java

package ejemplo22;

import java.util.Map;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;

@SuppressWarnings("serial")
public class AutenticacionInterceptor extends
AbstractInterceptor{

      // método ejecutado antes de la acción
      public void init()
      {
            System.out.println("Antes del método de acción");
      }

      // método del interceptor
      public String intercept(ActionInvocation invocation) throws
Exception
      {
            System.out.println("En el método del interceptor");

            // recuperar los objetos de la sesión
            Map<String,Object>
session=invocation.getInvocationContext().getSession();

            if(session.get("autenticacion")==null)
            {
                  // el usuario no se autentica
regresar a la página de autenticación
                  return "autenticacion";
            }
            else
            {
                  // verificar la autenticación
                  boolean autenticacion=(Boolean)
(session.get("autenticacion"));
                  if(autenticacion)
                  {
                        // el usuario se han autenticado,
continuar la acción
                        return invocation.invoke();
                  }
                  else
                  {
```

```
                        // el usuario no se ha autenticado
regresar a la página de autenticación
                        return "autenticacion";
                  }
            }
      }

      // método ejecutado después de la acción
      public void destroy()
      {
            System.out.println("Despues del método de acción");
      }
}
```

Las páginas JSP permite mostrar respectivamente un enlace a los recursos protegidos por autenticación (/jsp/Lista.jsp), un formulario de autenticación (/jsp/Autenticacion.jsp) y los propios recursos protegidos (/jsp/Rescurso.jsp).

Código: /jsp/Lista.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Lista</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Lista</h3>
      <a href="Proteger.action">Acceder al recurso</a>
</div>
</body>
</html>
```

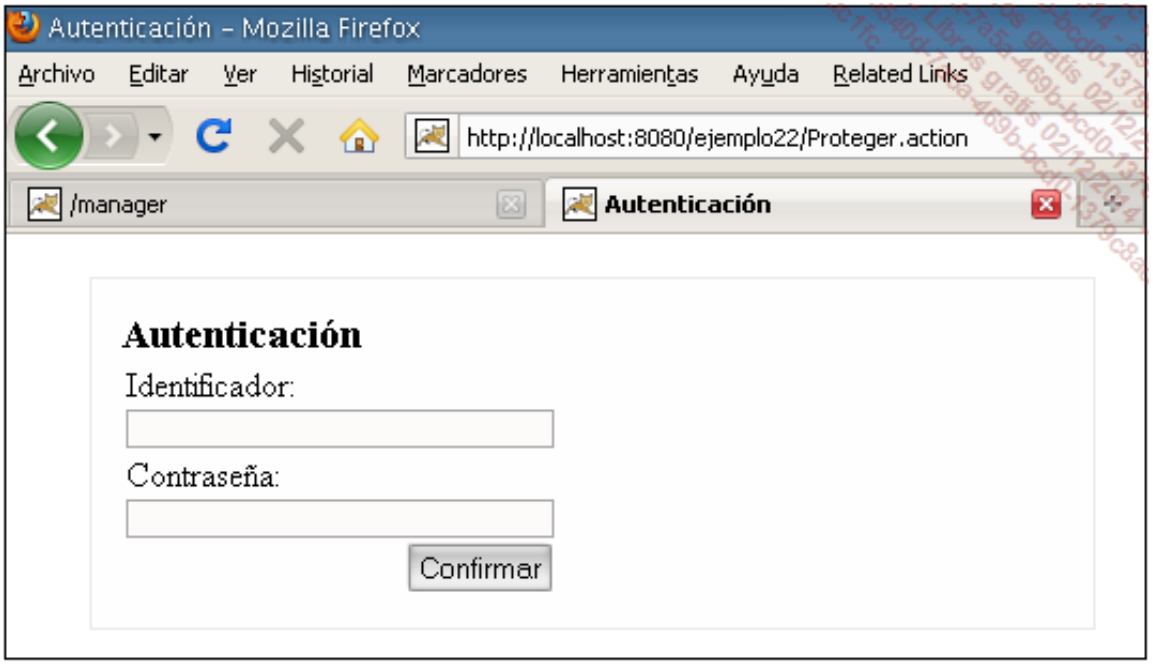
Código: /jsp/Autenticacion.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Autenticaci&oacute;n</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

      <h3>Autenticaci&oacute;n</h3>
      <s:form method="post" action="Conectar">
            <s:textfield name="identificador" id="identificador"
label="Identificador" labelposition="top" cssClass="input"/>
            <s:textfield name="contrasena" id="contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
            <s:submit value="Confirmar"/>
      </s:form>
</div>
</body>
</html>
```

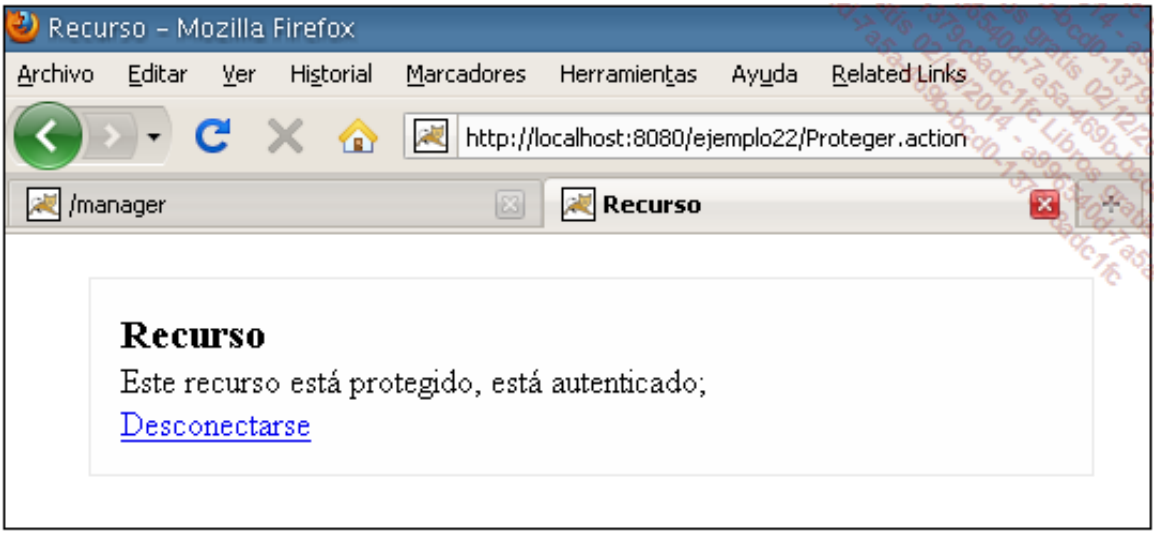
Código: /jsp/Recurso.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Recurso</title>
<style type="text/css">@import url(css/estilos.css);</style>
```

```
</head>
<body>
<div id="carta">

      <h3>Recurso</h3>

      Este recurso est&aacute; protegido, est&aacute;
autenticado;
     <br/><a href="Desconectar.action">Desconectarse</a>
</div>
</body>
</html>
```



Formulario de autenticación del cliente



Acceso protegido por un interceptor de Struts personalizado

En resumen

En este capítulo se ha presentado la escritura personalizada de interceptores mejorar los servicios que ofrece Struts mediante la implementación de la interfaz *Interceptor* o heredando de la clase abstracta `AbstractInterceptor`. La programación de interceptores permite, por lo tanto, simplificar las clases, las estructuras y la división de servicios.

Presentación

Struts incluye distintos tipos de resultados utilizados a lo largo del libro como *Dispatcher*, *Stream* o*redirectAction*. El framework también ofrece la posibilidad de crear resultados personalizados en función de una necesidad.

Un resultado de Struts debe implementar la interfaz *com.opensymphony.xwork2.Result*. Esta interfaz cuenta con un solo método llamado `execute(ActionInvocation invocation)`, que se activa cuando se ejecuta el resultado o `doExecute(String finalLocation, ActionInvocation invocation)`, que permite especificar el destino final de la acción para el enrutamiento. El objetivo de la creación de resultados personalizados es el de escribir el código que se ejecutará cuando se reenvíe el resultado. El principio es prácticamente idéntico al del desarrollo de los interceptores y consiste en heredar de la clase`org.apache.struts2.dispatcher.StrutsResultSupport` que a su vez implementa la interfaz *Result*.

Escribir un resultado

El proyecto *ejemplo23* basado en la aplicación *ejemplo14* (gestión de cuentas de clientes como una colección) proporciona un resultado que permite devolver la lista de las cuentas de usuario en forma de flujo RSS en lugar de texto.

RSS (*Really Simple Syndication*) es un formato de intercambio de datos en formato XML. Las fuentes son ahora distribuidores, reutilizables y diseñadas para el flujo RSS o sindicación. Los navegadores actuales pueden leer directamente los archivos RSS, pero también podemos utilizar un software especializado llamado el lector RSS. Un documento RSS es un archivo en formato XML que contiene la etiqueta `<rss/>` y, al menos, un canal con la etiqueta `<channel/>` que proporcionen la información del sitio. Este canal está compuesto de un determinado número de artículos y de etiquetas `<item/>` que representan la información.

El archivo *struts.xml* contiene la definición del resultado con la etiqueta `<result-types/>` y la asociación de la clase:

```
<result-types>
    <result-type name="rss" class="ejemplo23.rssResult"/>
</result-types>
```

El archivo *struts.xml* también define una acción llamada *ListadoRss_Cliente.action* que contiene un resultado de tipo *rss* para el procesamiento de información.

```
Código: 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="true" />

    <package name="ejemplo23" namespace="/" extends="struts-
default">
      <result-types>
              <result-type name="rss" class="ejemplo23.rssResult"/>
      </result-types>

        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo23.ClienteAccion" method="listado">
              <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="ListadoRSS_Cliente"
class="ejemplo23.ClienteAccion" method="listado">
              <result type="rss">/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo23.ClienteAccion" method="agregar">
              <result name="input">/jsp/ListadoCliente.jsp</result>
              <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Editar_Cliente"
class="ejemplo23.ClienteAccion" method="editar">
```

```
                    <interceptor-ref
name="paramsPrepareParamsStack"/>
                <result name="success">/jsp/EditarCliente.jsp</result>
        </action>

        <action name="Modificar_Cliente"
class="ejemplo23.ClienteAccion" method="modificar">
                <result name="input">/jsp/EditarCliente.jsp</result>
                <result name="success"
type="redirectAction">Listar_Cliente</result>
        </action>

        <action name="Eliminar_Cliente"
class="ejemplo23.ClienteAccion" method="eliminar">
                <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

    </package>
</struts>
```

El código del interceptor permite recuperar la instancia de la respuesta HTTP y definir un resultado en forma de contenido XML (*setContentType*). La lista dinámica de clientes, presente en el objeto de invocación y devuelta por la clase de acción, se recupera y se le da formato para respetar la presentación RSS (canal y artículos).

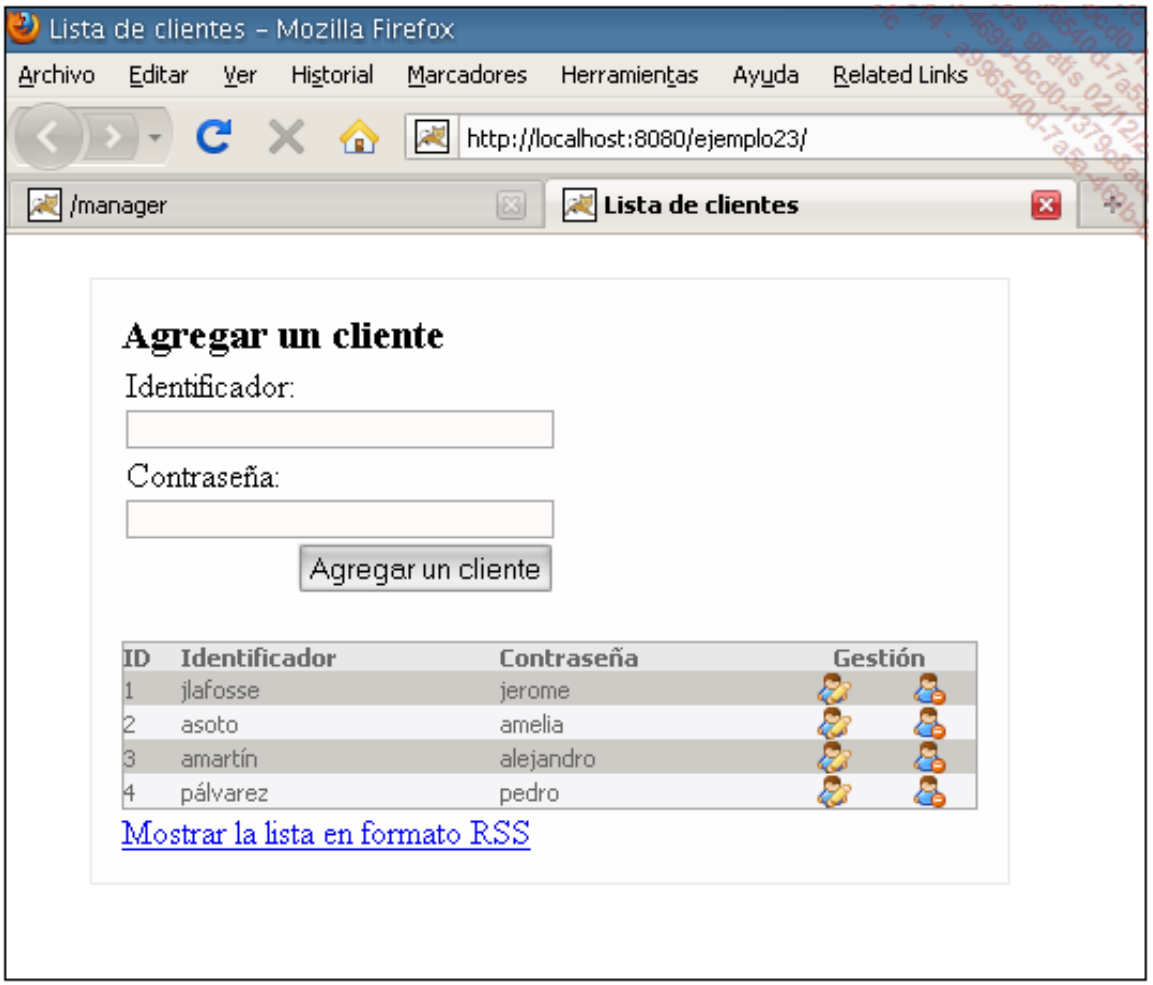
Código: ejemplo23.rssResult.java

```
package ejemplo23;

import java.io.PrintWriter;
import java.util.List;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts2.StrutsStatics;
import org.apache.struts2.dispatcher.StrutsResultSupport;
import com.opensymphony.xwork2.ActionInvocation;
import ejemplo23.javabeans.Cliente;

@SuppressWarnings("serial")
public class rssResult extends StrutsResultSupport{

    // método ejecutado por el resultado
    public void doExecute(String finalLocation,ActionInvocation
invocation) throws Exception
      {
            // recuperar el objeto response
            HttpServletResponse
response=(HttpServletResponse)invocation.getInvocationContext().get
(StrutsStatics.HTTP_RESPONSE);
            // recuperar la lista de clientes de la pila
de ejecución
            List<Cliente>
listaClientes=(List<Cliente>)invocation.getStack().findValue("lista
Clientes");

            // tipo de respuesta
            response.setContentType("application/xml");
            PrintWriter out=response.getWriter();
            out.println("<?xml version=\"1.0\" encoding=\"UTF-
8\"?>");
            out.println("<rss version=\"2.0\">");
            // crear un canal
            out.println("<channel>");
            out.println("<title>Flujo RSS de los clientes</title>");
```

```
out.println("<link>http://localhost:8080/ejemplo23/</link>");
            out.println("<description>Flujo RSS de
los clientes</description>");
            // crear los artículos de los clientes
            for(int i=0;i<listaClientes.size();i++)
            {
                  Cliente cliente=(Cliente)listaClientes.get(i);
                  out.println("<item>");
                        out.println("<title> Cliente:
"+cliente.getIdentificador()+"</title>");

out.println("<link>http://localhost:8080/ejemplo23/Editar_Cliente.
action?idClienteActual="+cliente.getIdCliente()+"</link>");
                        out.println("<description> Cliente:
"+cliente.getIdCliente()+" - "+cliente.getIdentificador()+" -
"+cliente.getContrasena()+"</description>");
                  out.println("</item>");
             }
             out.flush();
             out.close();
      }
}
```

Se añade un enlace a la acción *ListadoRss_Cliente.action* en la vista JSP */jsp/ListadoCliente.jsp*, responsable de la visualización de la información.

```
<a href="ListadoRSS_Cliente.action">Mostrar la lista en formato
RSS</a>
```



Gestión de los clientes en formato RSS

Gestión del resultado en formato RSS

En resumen

En este capítulo se ha explicado en detalle la escritura de resultados personalizados que se utilizarán en las acciones de Struts. El ejemplo propuesto permite manipular los datos enviados, modificarlos y devolver el resultado en diferentes formatos. Esta técnica se utiliza en los resultados propuestos por Struts para realizar redirecciones, presentaciones en XML o para devolver flujos de bytes.

Presentación

Al navegar entre las diferentes páginas de una aplicación de Internet, sucede a menudo que una acción es llamada 2 veces consecutivas o que se ha añadido un artículo a su carrito de la compra dos veces por error. Este doble clic o doble envío es más problemático cuando se encuentra en la fase de pago (monto retirado dos veces en la cuenta del cliente) o de inserción de datos (creación de dos artículos en lugar de uno).

Esta validación doble se produce cuando el procesamiento del lado del servidor es largo y el usuario hace clic varias veces sobre el botón de enviar, o cuando el usuario actualiza la página actual que realiza la acción. Struts ofrece una técnica de gestión del doble envío que puede aplicarse a otros idiomas. La técnica consiste en utilizar un testigo (token) único para cada usuario, guardado en el servidor y con una copia insertada en cada formulario HTML.

Cuando se envía el formulario al servidor, la aplicación, para enviado con el que se encuentra en el servidor, realiza la acción en caso de equivalencia y a continuación elimina el testigo del servidor. Así, cuando se produce un doble envío por error o de manera involuntaria, con el testigo del servidor eliminado, la acción no se ejecutará.

Struts proporciona la etiqueta `<s:token/>` para gestionar el testigo del lado del cliente. Esta etiqueta debe colocarse en una etiqueta `<s:form/>` y generar un campo de caché de tipo *<hidden/>* para guardar el testigo. Para colocar un testigo en una acción es necesario declarar el interceptor *Token* o*TokenSession*. El interceptor *Token* devuelve un resultado llamado *invalid.token* de la lista de errores de acción *<s:actionerror/>*.

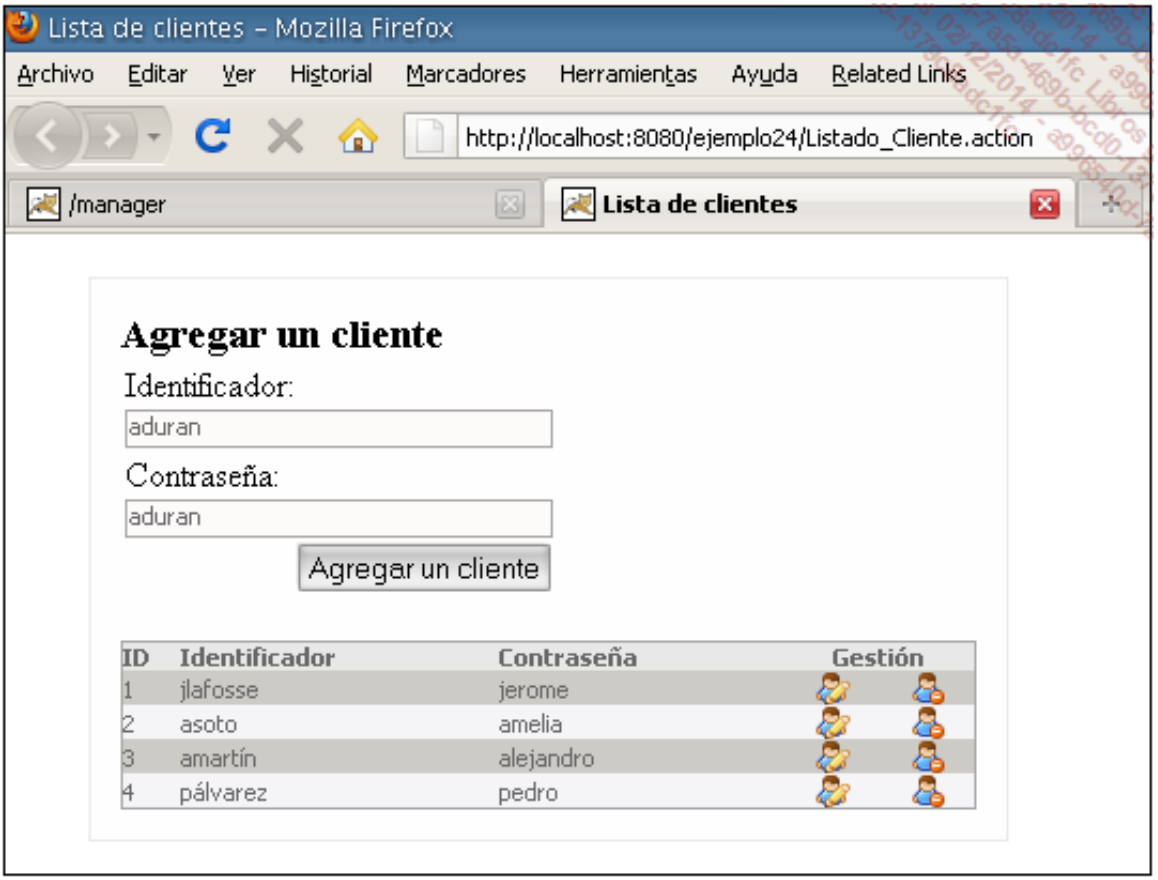
Para adaptar el mensaje al idioma deseado, podemos crear la variable struts.messages.invalid.token en el archivo del idioma.

Aplicación del testigo

Para aplicar la gestión del testigo y del interceptor *TokenInterceptor*, utilizaremos el ejemplo de gestión de clientes *ejemplo14* en un nuevo proyecto llamado *ejemplo24*. El método `agregar()` permite crear un nuevo cliente y se modificará para abordar el problema del doble envío. Para ello, crearemos un thread de espera de varios segundos.

```
Código: ejemplo24.ClienteAccion.java
package ejemplo24;

import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.ModelDriven;
import com.opensymphony.xwork2.Preparable;
import ejemplo24.javabeans.Cliente;
import ejemplo24.modelo.ClienteModelo;
@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport implements
Preparable, ModelDriven{
    ...
      // agregar el cliente al modelo
      public String agregar()
      {
            try
            {
            Thread.sleep(4000);
            }
            catch(Exception e)
            {

            }
            ClienteModelo.agregar(cliente);
          return SUCCESS;
      }

      ...
}
```

El thread de espera permite simular una carga larga de la página cuando se añade el cliente. A continuación, podemos visualizar el proyecto y crear una nueva cuenta. Si el tiempo de carga de la página es largo, podemos hacer clic en el botón Agregar (**Agregar un cliente**) varias veces consecutivas. Entonces observamos las inserciones múltiples a la lista.



Formulario de registro del cliente

Visualización múltiple de las cuentas de los clientes

Para evitar este doble envío, utilizaremos el interceptor *token* y el nombre *invalid.token* en el resultado para gestionar la visualización del testigo en la definición de la acción *Agregar_Cliente.action*. También utilizamos el interceptor *validationWorkflowStack* para conservar la aplicación de las validaciones del formulario.

```
<action name="Agregar_Cliente" class="ejemplo24.ClienteAccion"
method="agregar">
       <interceptor-ref name="token"/>
       <interceptor-ref name="validationWorkflowStack"/>
       <result name="invalid.token">/jsp/ListadoCliente.jsp</result>
       <result name="input">/jsp/ListadoCliente.jsp</result>
       <result name="success"
type="redirectAction">Listado_Cliente</result>
</action>
```

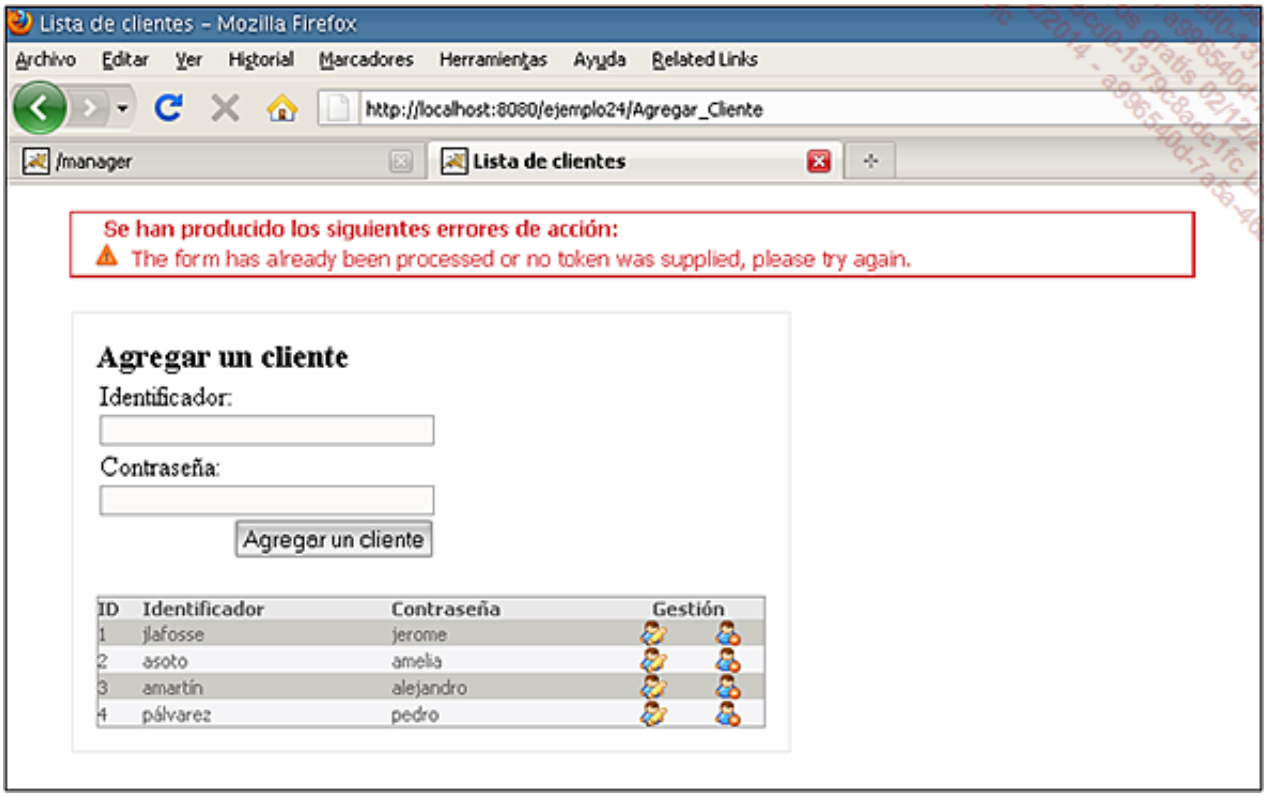
Se mostrará el error del testigo mediante la etiqueta `<s:actionerror/>`.

Para gestionar el testigo del interceptor, añadimos la etiqueta `<s:token/>` en el formulario de registro*/jsp/ListadoCliente.jsp*.

```
Código: /jsp/ListadoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Listado de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los siguientes errores: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<!-- Mensaje de error de las acciones -->
<s:if test="errorMessages.size()>0">
<div id="mensaje_error">
      <label>Se han producido los siguientes errores de acción:
</label>
      <ul><s:actionerror/></ul>
</div>
</s:if>
<!-- Mensaje de confirmación -->
<s:if test="actionMessages.size()>0">
      <div id="mensaje_informacion">
            <ul><s:actionmessage/></ul>
      </div>
</s:if>
<div id="carta">

      <h3>Agregar un cliente</h3>
      <s:form method="post" action="Agregar_Cliente">
            <s:token/>
            ...
</div>
</body>
</html>
```

El código fuente generado por la etiqueta `<s:token/>` en el formulario HTML podría ser como el siguiente: `<input type="hidden" name="struts.token" value="LQV6RKHKOLETXKRZSR32B5VPGXIWTJA4" />`

Podemos probar de nuevo la aplicación realizando varias validaciones del formulario a continuación. Se mostrará el mensaje de error vinculado al testigo y se realizará una única creación.



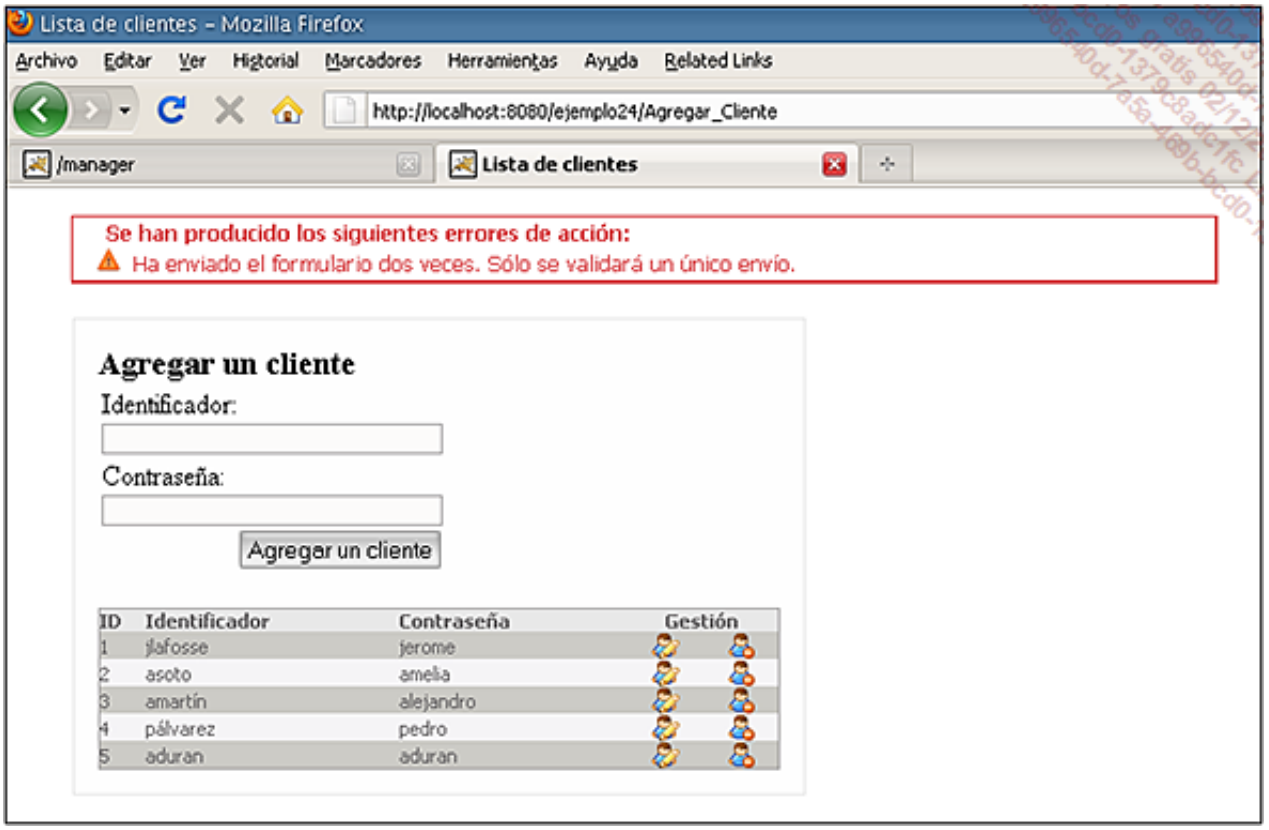
Gestión del doble clic con el mensaje por defecto

El mensaje de error asociado a la gestión de testigos es la clave *struts.messages.invalid.token* que se encuentra en el archivo *struts-messages.properties* incluido con Struts. La clase de gestión del interceptor es *org.apache.struts2.interceptor.TokenInterceptor*. Para sobrecargar del mensaje mostrado en el error de la acción, es necesario crear un archivo llamado *TokenInterceptor.properties* que se encuentra en el directorio */WEB-INF/classes/org/apache/struts2/interceptor*.

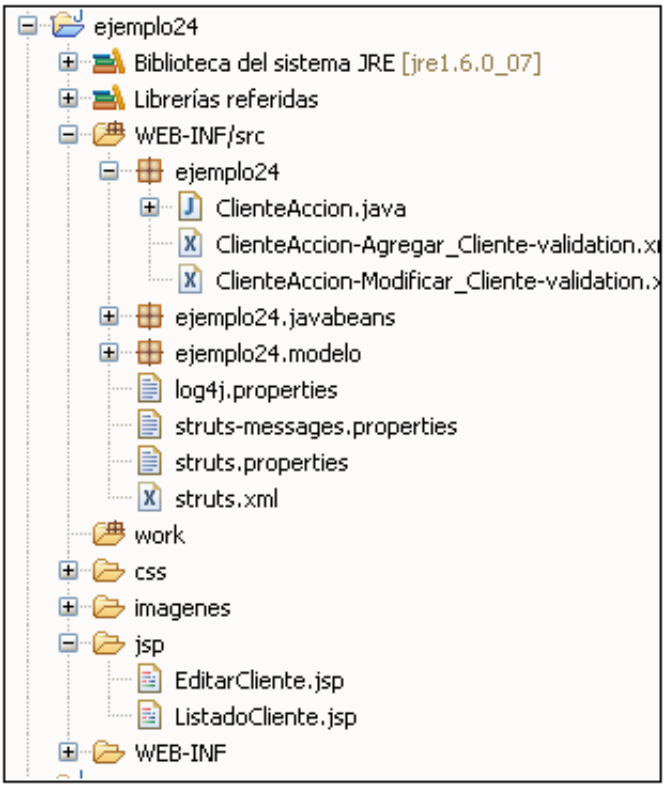
Una 2ª técnica consiste en crear, como en los ejemplos anteriores del libro, un archivo*struts.properties* en la carpeta */WEB-INF* de la aplicación con el parámetro *struts.custom.i18n.resources*y un segundo archivo llamado *struts-messages.properties* con la clave y su valor.

```
Código: struts.properties
struts.custom.i18n.resources=struts-messages
```

```
Código: struts-messages.properties
struts.messages.invalid.token=Ha enviado el formulario dos
veces. Sólo se validará un único envío.
```



Formulario de cliente con gestión del doble clic multilingüe



Árbol del proyecto ejemplo24

El interceptor tokenSession es similar al interceptor token, la diferencia reside en la forma de guardar los datos al validar el formulario. El interceptor tokenSession guarda los datos del formulario en la sesión del usuario.

En resumen

En este capítulo se ha explicado en detalle cómo evitar los errores relacionados con el doble clic o el doble envío cuando el usuario hace clic varias veces sobre el botón de confirmación o actualizar la página. La técnica utilizada para evitar el doble envío consiste en utilizar un testigo o token, similar del lado del cliente y del servidor y válido para un único intercambio. Para aplicar esta técnica, Struts ofrece la posibilidad de utilizar la etiqueta `<s:token/>` en las vistas y los interceptores *token* y*tokenSession*.

Presentación

En las primeras versiones de Struts 2 se incluía el complemento *Dojo*, un framework JavaScript OpenSource que ofrecía un conjunto de etiquetas para gestionar componentes Ajax. Esta librería contaba por ejemplo con la etiqueta `<sx:head/>` para insertar automáticamente las librerías JavaScript, la etiqueta `<sx:div/>` para gestionar las llamadas Ajax o la etiqueta `<sx:submit/>` para publicar formularios utilizando la tecnología Ajax (sin volver a cargar la página).

Desafortunadamente, la versión de *Dojo* presente en el complemento es una versión antigua y no es posible actualizar esta librería. Asimismo, esta librería, basada en la versión 0.4 de Dojo, es particularmente lenta. Los diseñadores de Struts han precisado además que el uso del complemento Dojo para Struts superior a la versión 2.1 no es adecuado y que el mantenimiento no es compatible con esta librería. Los desarrolladores de Struts trabajan actualmente con un complemento basado en la librería JavaScript *JQuery*, pero han aclarado que con Struts los desarrolladores pueden utilizar cualquier framework JavaScript.

En la actualidad, los protagonistas de Internet hablan a menudo de las tecnologías Web 2.0. Tras este término se esconde un conjunto de tecnologías y herramientas basados en la ergonomía de las interfaces. La expresión se ha propuesto para designar las nuevas tecnologías y la renovación de Internet. Este término fue inventado por un miembro de la sociedad O'Reilly para designar el renacimiento de la Web. El fin es utilizar las tecnologías de Internet acercándose a interfaces hombre/máquina atractivas del software instalado en los equipos personales.

La definición exacta de Web 2.0 aún no está muy clara, pero se sabe que un sitio Web 2.0 cuenta con las características siguientes:

- Se facilita la gestión de la información del sitio (realizar listados, consultar, modificar y eliminar).
- El sitio se puede utilizar en su totalidad con un navegador estándar.
- El sitio utiliza los estándares de la tecnología.

Desde el punto de vista tecnológico, la infraestructura Web 2.0 es bastante compleja, ya que incluye a la vez el o los servidores, la sindicación del contenido, la mensajería y las aplicaciones.

Instalación del framework JavaScript

Desde hace años, se vienen desarrollando numerosos framework especializados en JavaScript específicos que permiten ofrecer un conjunto de servicios Web 2.0. Podemos citar por ejemplo*Prototype* (http://www.prototypejs.org/), así como *Scriptaculous* (http://script.aculo.us/) basado en Prototype y que ofrece nuevas funcionalidades Ajax y DHTML. La librería más potente, pero también la más pesada y compleja, es actualmente, sin ninguna duda, *ExtJs* (http://extjs.com/). Por último, la más utilizada y recomendada por los diseñadores de Struts es *JQuery* (http://jquery.com/). Esta librería, muy ligera en una versión comprimida (alrededor de 20 Kb), permite manipular documentos HTML y XHTML. Asimismo, ofrece un sencillo sistema de acceso a las etiquetas de los documentos mediante notación de puntos. Este framework propone cientos de complementos para gestionar la visualización, los formularios, Ajax, las validaciones e incluso el uso del ratón.

La configuración de este framework es muy simple y está compuesta por tres etapas:

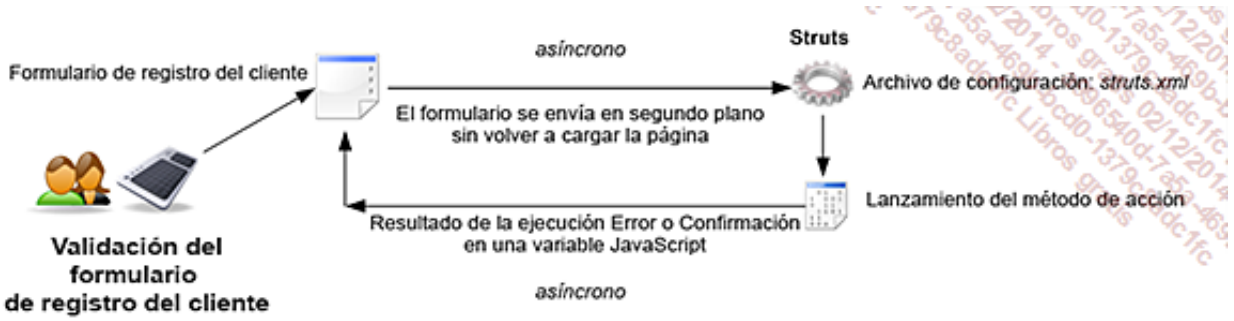
- Descarga de la librería JavaScript y de manera opcional de los complementos necesarios.
- Instalación de los archivos *.js* descargados en un directorio de la aplicación (por ejemplo:*/jscripts*).
- Declaración de las inclusiones de librerías en las páginas JSP usuarias (por ejemplo: `<script src="jscripts/jquery.min.js" type="text/javascript"></script>`).

Tecnología Ajax

AJAX (*Asynchronous JavaScript And XML*) es una técnica de desarrollo de Internet basada en el lenguaje JavaScript que permite efectuar consultas HTTP sin necesidad de volver a cargar las páginas. Esta tecnología hace que los sitios sean más interactivos y propone a los usuarios una ergonomía similar a la de los programas informáticos. Ajax se basa en el uso de tecnologías HTML y CSS, así como en el de DOM (*Document Object Model*) para la presentación de objetos, y por último del objeto JavaScript *XMLHTTPRequest* para la realización de consultas en segundo plano.

La tecnología Ajax obliga sin embargo a replantear el desarrollo de las aplicaciones. En realidad, se acude a los servicios en segundo plano y estos se devuelven en forma de variable.

En el caso de nuestro formulario de adición de un cliente a partir de su identificador y contraseña, debemos modificar la respuesta ya que el formulario no se volverá a cargar. Sólo deberán reenviarse y tratarse las respuestas de error y éxito.



Gestión del formulario de adición de clientes en Ajax

En nuestro ejemplo vamos a utilizar el framework *JQuery*, así como un conjunto de complementos, para llevar a cabo las principales operaciones útiles de una aplicación Ajax Web 2.0. Para ello, comenzaremos un nuevo proyecto, *ejemplo25*, que nos permitirá gestionar los clientes y la librería*JQueryForm Plug-in* (http://malsup.com/jquery/form/) para enviar el formulario en Ajax sin necesidad de volver a cargar la página.

La configuración se modifica desde el punto de vista de la estructura. En realidad, las respuestas a los errores de validación *<result name="input">* y de éxito *<result name="success">* siempre deben volver a la misma página de visualización de mensajes (errores, errores de acción y éxito). El resto de visualizaciones se gestionan en Ajax para evitar volver a cargar las páginas. El archivo de gestión de las acciones *struts.xml* cuenta con las declaraciones para la realización de listados, la adición, la modificación y la eliminación. Estas acciones se realizan en segundo plano y únicamente devuelven el mensaje adaptado en función de la acción (validación de campos, error, éxito). La acción*Formulario_Cliente.action* permite volver al archivo de gestión del cliente añadido o modificado dependiendo de la presencia o ausencia del objeto *cliente*. Esta acción utiliza el interceptor*paramsPrepareParamsStack* para gestionar la transmisión de parámetros. Finalmente, la última acción*Indice_Cliente.action* permite mostrar la página principal de gestión.

```
Código : 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="true" />

    <package name="exemple25" namespace="/" extends="struts-
default">
        <default-action-ref name="Indice_Cliente" />
```

```
        <action name="Indice_Cliente"
class="ejemplo25.ClienteAccion">
            <result>/jsp/IndiceCliente.jsp</result>
        </action>

        <action name="Formulario_Cliente"
class="ejemplo25.ClienteAccion" method="editar">
            <interceptor-ref name="paramsPrepareParamsStack"/>
            <result>/jsp/FormularioCliente.jsp</result>
        </action>

        <action name="Listado_Cliente"
class="ejemplo25.ClienteAccion" method="listado">
            <result>/jsp/ListadoCliente.jsp</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo25.ClienteAccion" method="agregar">
            <result name="input">/jsp/RespuestaAjax.jsp</result>
            <result name="success">/jsp/RespuestaAjax.jsp</result>
        </action>

        <action name="Modificar_Cliente"
class="ejemplo25.ClienteAccion" method="modificar">
            <result name="input">/jsp/RespuestaAjax.jsp</result>
            <result name="success">/jsp/RespuestaAjax.jsp</result>
        </action>

        <action name="Eliminar_Cliente"
class="ejemplo25.ClienteAccion" method="eliminar">
            <result name="input">/jsp/RespuestaAjax.jsp</result>
           <result name="success">/jsp/RespuestaAjax.jsp</result>
        </action>

    </package>
</struts>
```

La vista */IndiceCliente.jsp* contiene la declaración de los archivos JavaScript para el framework *JQuery* y su complemento de gestión de formularios. Esta sencilla página contiene dos bloques utilizados para mostrar el contenido del formulario y la lista de clientes.

Código: /jsp/IndiceCliente.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Lista de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
<!-- Utilizar la librería JavaScript JQuery -->
<script src="javascript/jquery.min.js"
type="text/javascript"></script>
<script src="javascript/jquery.form.js"
type="text/javascript"></script>
<script src="javascript/ejemplojform.js"
type="text/javascript"></script>
</head>
<body>
<!--  etiqueta utilizada para mostrar las respuestas Ajax -->
<div id="CuadroRespuestaAjax"></div>
<div id="carta">

      <!--  etiqueta utilizada para agregar/modificar un cliente en
Ajax -->
      <div id="CuadroFormularioCliente"></div>
```

```
      <!--  etiqueta utilizada para mostrar la lista de clientes en
Ajax -->
      <div id="CuadroListadoCliente"></div>
</div>
</body>
</html>
```

La página */jsp/FormularioCliente.jsp* es muy sencilla y permite mostrar el formulario del cliente que está siendo creado o modificado de acuerdo con el parámetro recibido.

```
Código: /jsp/FormularioCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>

<!--  formulario de registro o modificación -->
<s:if test="cliente.idCliente!=0">
<h3>Editar un cliente</h3>
<s:set id="accion">Modificar_Cliente.action</s:set>
</s:if>
<s:else>
<h3>Agregar un cliente</h3>
<s:set id="accion">Agregar_Cliente.action</s:set>
</s:else>

<s:form method="post" action="%{accion}" id="Formulario_Cliente"
name="Formulario_Cliente">
<s:hidden key="cliente.idCliente"/>
<s:textfield name="cliente.identificador" id="cliente.identificador"
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="cliente.contrasena" id="cliente.contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
<s:submit value="Confirmar"/>
</s:form>
```

La página */jsp/ListadoCliente.jsp* retoma el código de visualización de los datos del cliente.

```
Código: /jsp/ListadoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>

<table border="0" id="tabla" cellpadding="0" cellspacing="0">
      <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td colspan="2"
align="center"><b>Gestión</b></td></tr>
      <s:iterator value="listaClientes" status="linea">
      <s:if test="#linea.odd"><tr class="linea1"></s:if>
      <s:if test="#linea.even"><tr class="linea2"></s:if>
      <td><s:property value="idCliente"/></td>
      <td><s:property value="identificador"/></td>
      <td><s:property value="contrasena"/></td>
      <td align="center"><a href="JavaScript:modificarCliente('$
{idCliente}')"><img src="imagenes/editarcliente.png" alt="Editar"
title="Editar" border="0"/></a></td>
      <td align="center"><a href="JavaScript:eliminarCliente('$
{idCliente}')"><img src="imagenes/eliminarcliente.png" alt="Eliminar"
title="Eliminar" border="0"/></a></td>
      </tr>
      </s:iterator>
</table>
```

Por último, la página */jsp/RespuestaAjax.jsp*, utilizada por las acciones, permite simplemente mostrar los mensajes recibidos en Ajax.

```
Código: /jsp/RespuestaAjax.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
      <label>Se han producido los siguientes errores: </label>
      <ul><s:fielderror/></ul>
</div>
</s:if>
<!-- Mensaje de error durante las acciones -->
<s:if test="errorMessages.size()>0">
<div id="mensaje_error">
      <label>Se han producido los siguientes errores de acción:
</label>
      <ul><s:actionerror/></ul>
</div>
</s:if>
<!-- Mensaje de confirmación -->
<s:if test="actionMessages.size()>0">
      <div id="mensaje_informacion">
            <ul><s:actionmessage/></ul>
      </div>
</s:if>
```

Ahora el tratamiento de la aplicación se gestiona en el archivo JavaScript utilizado junto con el framework *JQuery*. El archivo JavaScript *ejemplojform.js* utiliza el método`$(document).ready(function())`, que permite realizar acciones una vez cargada la página. Utilizamos este método con el fin de cargar el formulario de registro o modificación del cliente y ejecutar la acción que permite realizar listados de los registros.

Este archivo contendrá cuatro métodos que corresponden a las acciones a llevar a cabo:

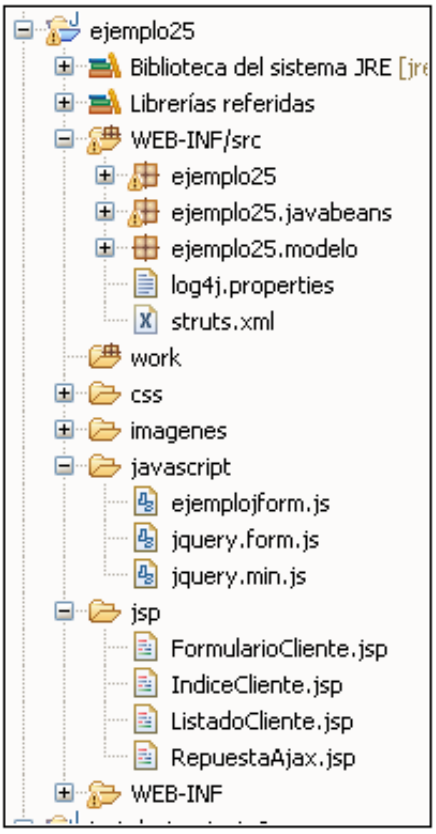
- `agregarCliente()`: este método permite acudir de manera asíncrona a la acción`Formulario_Cliente.action` y precisar que el formulario utilizará le complemento *ajaxForm*para el envío de la información.
- `listadoCliente()`: este método permite acudir de manera asíncrona a la acción*Listado_Cliente.action* y mostrar el contenido en el cuadro *CuadroListadoCliente* de la página actual.
- `eliminarCliente (idClienteActual)`: este método permite acudir de manera asíncrona a la acción *Eliminar_Cliente.action* y transmitir el cliente relevante. Cuando se ejecuta el resultado, se desencadena de nuevo el método de realización de listados para actualizar la lista.
- `modificarCliente (idClienteActual)`: este método permite acudir de manera asíncrona a la acción *Formulario_Cliente.action* y mostrar el formulario que está siendo modificado asignando el complemento *ajaxForm* para convertir el formulario en asíncrono. Durante la modificación, se actualiza el listado de clientes y se vuelve a mostrar el formulario de adición.

```
Código: /javascript/ejemplojsform.js
// acción realizada al inicio
$(document).ready(function() {

    // devolver la lista de clientes
    listadoCliente();

    // devolver el formulario de adición de clientes
    agregarCliente();
});

// devolver el formulario de clientes utilizando Ajax
function agregarCliente()
{
```

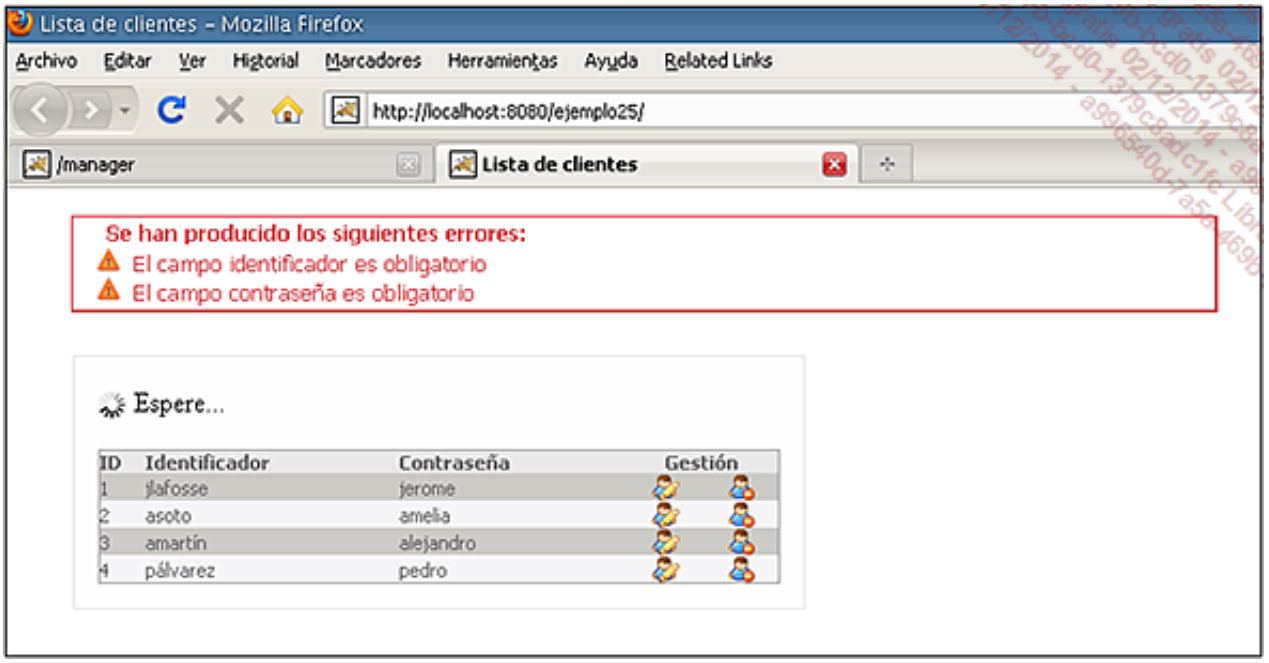
```
      //enviar los datos en POST
      $.ajax(
      {
            type: "POST",
            url: "Formulario_Cliente.action",
            dataType: "html",
            timeout : 8000,
            error: function(){
        alert('Desconexion del servidor');
        },
            beforeSend : function()
            {
                  $("#CuadroFormularioCliente").html('<img
src="imagenes/cargacircularpequena.gif" align="absmiddle"
width="16" height="16" style="padding-right:5px;"/>Espere...');
            },
            success: function(html)
            {
                  // incluir el resultado recibido en la etiqueta
de la página actual
          $("#CuadroFormularioCliente").html(html);

          // utilizar el formulario ajaxForm
                $('#Formulario_Cliente').ajaxForm({
                    // precisar el destino de la
respuesta (etiqueta)
                    target: '#CuadroRespuestaAjax',
                    // función desencadenada tras el envío del
formulario
                    success: function() {

                      // mostrar las respuestas (errores,
errores de acción, confirmación)
                      $('#CuadroRespuestaAjax').fadeIn('slow');
                      // adición realizada correctamente
                      if($
('#CuadroRespuestaAjax').html().indexOf("mensaje_informacion")!=-1)
                      {
                        // recuperar la nueva lista de cliente
                              listadoCliente();
                              // vaciar los campos del formulario
                              $
('#Formulario_Cliente').resetForm();
                              $
('#Formulario_Cliente').clearForm();
                      }

                    }
                });
            }
      });
}

// devolver la lista de clientes utilizando Ajax
function listadoCliente()
{
      //enviar los datos en POST
      $.ajax(
      {
            type: "POST",
            url: "Listado_Cliente.action",
            dataType: "html",
            timeout : 8000,
            error: function(){
        alert('Desconexion del servidor');
```

```
        },
            beforeSend : function()
            {
                  $("#CuadroListadoCliente").html('<img
src="imagenes/cargacircularpequena.gif" align="absmiddle"
width="16" height="16" style="padding-right:5px;"/>Espere...');
            },
            success: function(html)
            {
                  // incluir el resultado recibido en la etiqueta
de la página actual
           $("#CuadroListadoCliente").html(html);
            }
      });
}

// eliminar el cliente
function eliminarCliente(idClienteActual()
{
      //enviar los datos en POST
      $.ajax(
      {
            type: "POST",
            url: "Eliminar_Cliente.action",
            dataType: "html",
            data: "idClienteActual="+idClienteActual,
            timeout : 8000,
            error: function(){
        alert('Desconexión del servidor');
        },
            beforeSend : function()
            {
                  $("#CuadroListadoCliente").html('<img
src="imagenes/cargacircularpequena.gif" align="absmiddle"
width="16" height="16" style="padding-right:5px;"/>Espere...');
            },
            success: function(html)
            {
            // mostrar la respuesta ajax
            $('#CuadroRespuestaAjax').html(html);
            // eliminación realizada correctamente
                  if($
('#CuadroRespuestaAjax').html().indexOf("mensaje_informacion")!=-1)
                  {
                        listadoCliente();
            }
            }
      });
}

// modificar el cliente
function modificarCliente(idClienteActual()
{
      //enviar los datos en POST
      $.ajax(
      {
            type: "POST",
            url: "Formulario_Cliente.action",
            dataType: "html",
            data: "idClienteActual="+idClienteActual,
            timeout : 8000,
            error: function(){
        alert('Desconexión del servidor');
        },
            beforeSend : function()
```

```
            {
                  $("#CuadroFormularioCliente").html('<img
src="imagenes/cargacircularpequena.gif" align="absmiddle"
width="16" height="16" style="padding-right:5px;"/>Espere...');
            },
            success: function(html)
            {
                  // incluir el resultado recibido en la etiqueta
de la página actual
            $("#CuadroFormularioCliente").html(html);

          // utilizar el formulario ajaxForm
                $('#Formulario_Cliente').ajaxForm({
                    // precisar el destino de la
respuesta (etiqueta)
                    target: '#CuadroRespuestaAjax',
                    // función desencadenada tras el envío
del formulario
                    success: function() {

                        // mostrar las respuestas (errores,
errores de acción, confirmación)
                         $('#CuadroRespuestaAjax').fadeIn('slow');
                        // modificación realizada correctamente
                        if($
('#CuadroRespuestaAjax').html().indexOf("mensaje_informacion")!=-1)
                        {
                            // recuperar la nueva lista de cliente
                                  listadoCliente();
                                  // volver a mostrar el formulario
de creación
                                  agregarCliente();
                        }

                    }
                });
            }
      });
}
```



Árbol del proyecto ejemplo25



Gestión de clientes en Ajax

Optimizaciones

El ejemplo anterior es ampliamente funcional. Ahora vamos a mejorarlo con el proyecto *ejemplo26*, mediante servicios de tipo Web 2.0 con el lenguaje DHTML, las hojas de estilos y los complementos*JQuery*.

Los optimizaciones utilizadas son las siguientes:

- Utilización de botones dinámicos.
- Gestión de cuadros (box) dinámicos para las confirmaciones y mensajes.
- Utilización del complemento de gestión de componentes o widgets de tipo googletoolkit.
- Utilización de un calendario dinámico para la gestión de fechas.
- Utilización de un servicio de autocompletado para las búsquedas.
- Gestión de clasificaciones dinámicas.

1. Utilización de botones dinámicos

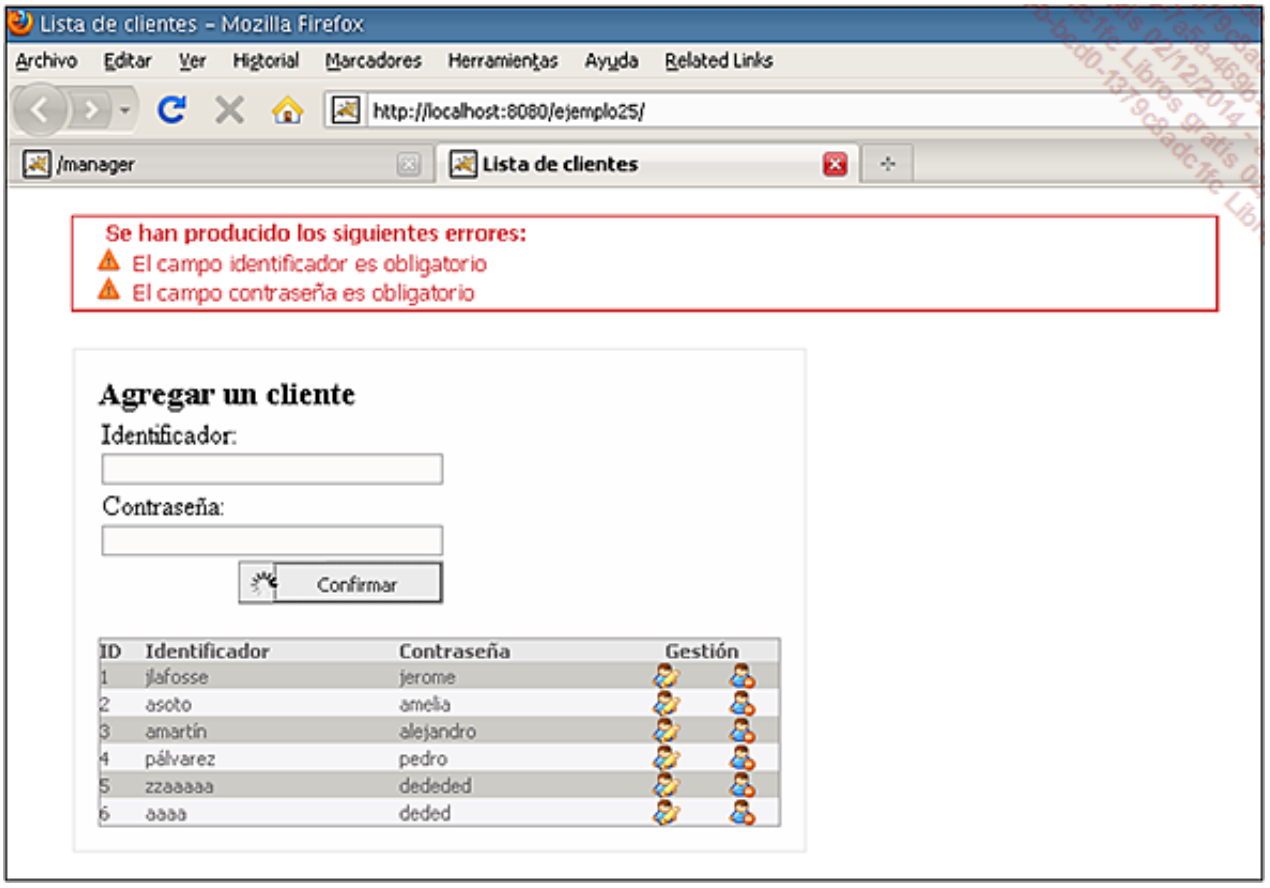
Vamos a agregar un nuevo estilo con un icono para el botón de confirmación con el fin de mostrar la imagen de carga en el botón al igual que en los programas informáticos.

```
#botonconfirmar
{
      background-image: url(../imagenes/confirmar.png);
      background-repeat:no-repeat;
      background-position:5% 65%;
      height:20px;
      padding-left:20px;
      width:120px;
      height:25px;
      color:#333333;
      font-family: tahoma, verdana, arial, sans-serif;
      font-size:11px;
      font-weight:normal;
      margin-left:17px;
      background-color:#ebebeb;
      border:1px solid;
      border-top-color:#999999;
      border-left-color:#999999;
      border-right-color:#666666;
      border-bottom-color:#666666;
}
```

A continuación, agregaremos las líneas *JQuery* siguientes, que permiten cambiar la imagen de fondo del botón, desactivar el mismo y mostrarlo de nuevo cuando se produce la respuesta.

```
// utilizar el formulario ajaxForm
$('#Formulario_Cliente').ajaxForm({
            ...
// función desencadenada antes del envío del formulario
beforeSubmit : function() {
$("#botonconfirmar").css("background-
image","url(./imagenes/cargacircularpequena.gif)");
$("#botonconfirmar").attr("disabled","disabled");
                    },
// función desencadenada tras el envío del formulario
success: function() {
```

```
// volver a colocar el botón
$("#botonconfirmar").css("background-
image","url(./imagenes/confirmar.png)");
$("#botonconfirmar").removeAttr("disabled");
...
```



Utilización de botones dinámicos

2. Gestión de cuadros (box) dinámicos para las confirmaciones y mensajes

Vamos a agregar un nuevo complemento *JQuery* denominado *JQueryUI* (http://jqueryui.com/). Este complemento permite gestionar calendarios, barras de carga, tablas, etc. La librería se copia en el directorio */javascript/jqueryui* con el directorio */css* para las hojas de estilos y el directorio */js* para los archivos JavaScript. Vamos a utilizar la librería para crear cuadros de diálogo dinámicos Web 2.0 con botones, con el fin de gestionar el redimensionamiento de los cuadros, el desplazamiento de estos y la gestión de la colocación en primer plano.

A continuación pasamos a la configuración de las librerías en la página */jsp/IndiceCliente.jsp*:

```
<!--  Utilizar la biblioteca JavaScript JQueryUI -->
<style type="text/css">@import url(javascript/plugin/jqueryui/css/
smoothness/jquery-ui-1.7.1.custom.css);</style>
<style type="text/css">@import url(javascript/plugin/jqueryui/css/
css.css);</style>
<script type="text/javascript" src="javascript/plugin/jqueryui/js/
jquery-ui-1.7.1.custom.min.js"></script>
<script type="text/javascript"
src="javascript/ejemplojqueryui.js"></script>
```

La librería está configurada y ahora podemos modificar los vínculos para la eliminación de clientes con el fin de desencadenar una nueva función JavaScript de confirmación avanzada.

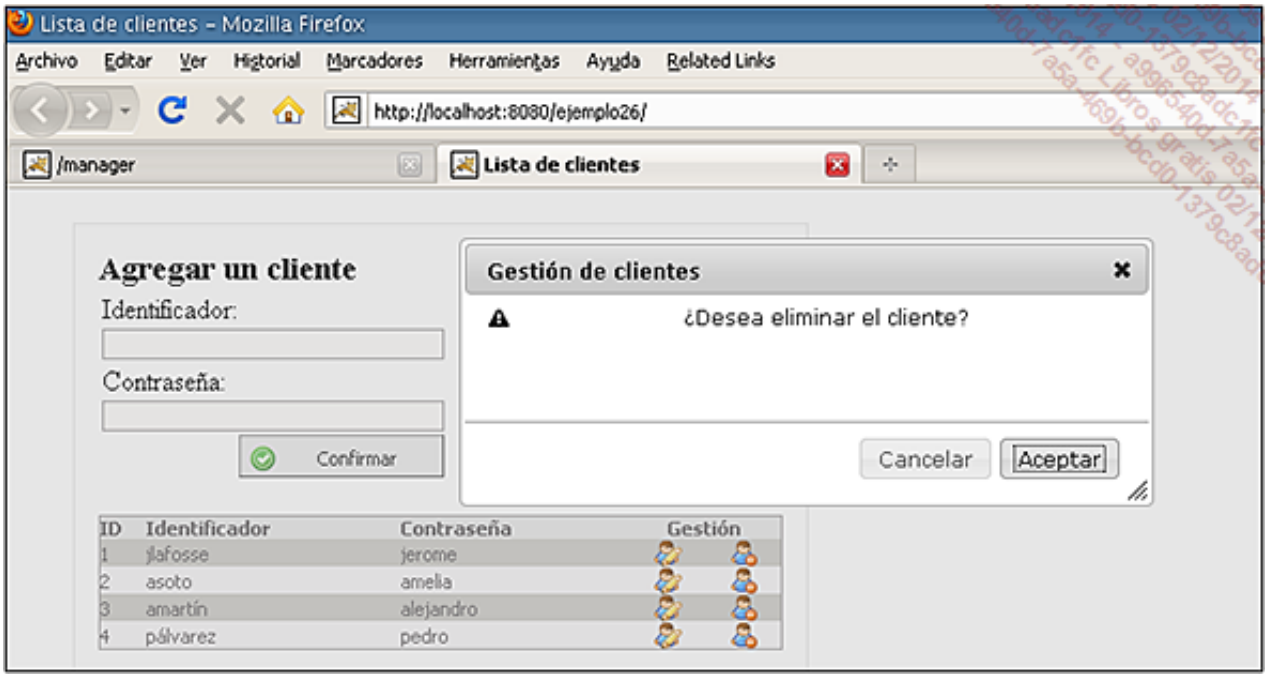
```
<a href="JavaScript:confirmarEliminarCliente('${idCliente}')"><img
src="imagenes/eliminarcliente.png" alt="Eliminar" title="Eliminar"
border="0"/></a>
```

La gestión de los cuadros dinámicos casi ha finalizado, sólo falta codificar la función*confirmarEliminarCliente(idClienteActual)* en el archivo JavaScript */javascript/ejemplojform.js* para poder mostrar el cuadro de diálogo modal. Este cuadro se muestra en el centro (position) y dispone de dos botones (**Aceptar** y **Cancelar**). El botón de confirmación desencadena el método`eliminarCliente(idClienteActual)`.

```
// función que permite mostrar una ventana de confirmación
antes de eliminar
function confirmarEliminarCliente(idClienteActual)
{
      $(function(){
                        // Configuración del cuadro
                        $('#CuadroConfirmacion').dialog({
                              autoOpen: false,
                              width: 400,
                              modal: true,
                              position: 'middle',
                              buttons: {
                                    "Aceptar": function() {
                                          $
(this).dialog("close");

      eliminarCliente(idClienteActual;
                                    },
                                    "Cancelar": function() {
                                          $
(this).dialog("close");
                                    }
                              }
                        });

      });
      // Mostrar el cuadro tras un clic en el ratón
      $('#CuadroConfirmacion').dialog('open');
}
```

La página principal */jsp/IndiceCliente.jsp* se modifica ligeramente para incluir el cuadro de diálogo oculto al inicio de la aplicación.

```
<!--  cuadro utilizado para las confirmaciones -->
<div id="CuadroConfirmacion" title="Gestión de clientes"
style="display:none">
       <p><span class="ui-icon ui-icon-alert" style="float:left;
margin:0 7px 20px 0;"></span>&iquest;Desea eliminar el cliente?</p>
</div>
```



Cuadro de diálogo Web 2.0

Para la gestión de los cuadros de mensaje (errores, errores de acción y éxito), vamos a utilizar un código JavaScript basado en un Timer (thread), que mostrará u ocultará los cuadros de acuerdo con las necesidades.

```
// tiempo de espera de los mensajes en milisegundos
var tiempoespera=3000;
// función que permite mostrar u ocultar los mensajes durante
un determinado tiempo
function mensajeDinamico(cache,animation)
{
        if($("#mensaje_error")!=null || $
("#mensaje_informacion")!=null)
        {
             if(cache)
             {
                $("#mensaje_error").effect("explode",{},500);
                $("#mensaje_informacion").effect("explode",{},500);
             }
             else
             {
                $("#mensaje_error").show("slow");
                $("#mensaje_informacion").show("slow");
             }
             if(animation) setTimeout("mensajeDinamico("+!
cache+",false)",tiempoespera);
        }
}
```

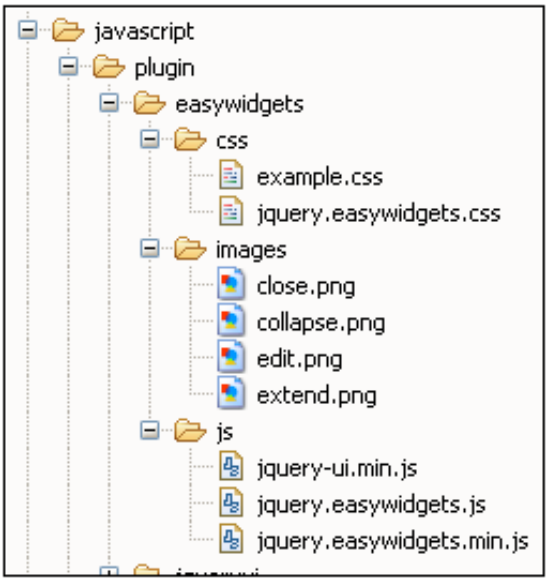
Por tanto, este método utiliza un tiempo de espera de 3 segundos (configurable) para ocultar los mensajes con un efecto de animación. A continuación se acude a la función*mensajeDinamico(cache,animation)* en cada método que devuelve un valor.

```
...
// función desencadenada tras el envío del formulario
success: function() {
      // mostrar las respuestas (errores, errores de acción,
confirmación)
      $('#CuadroRespuestaAjax').fadeIn('slow');
```

```
      // modificación realizada correctamente
      if($
('#CuadroRespuestaAjax').html().indexOf("mensaje_informacion")!=-1)
      {
            // recuperar la nueva lista de cliente
            listadoCliente();
            // volver a mostrar el formulario de creación
            agregarCliente();
      }
      // gestión dinámica de los mensajes
      mensajeDinamico(false,true);
}
...
```

3. Utilización del complemento Widget

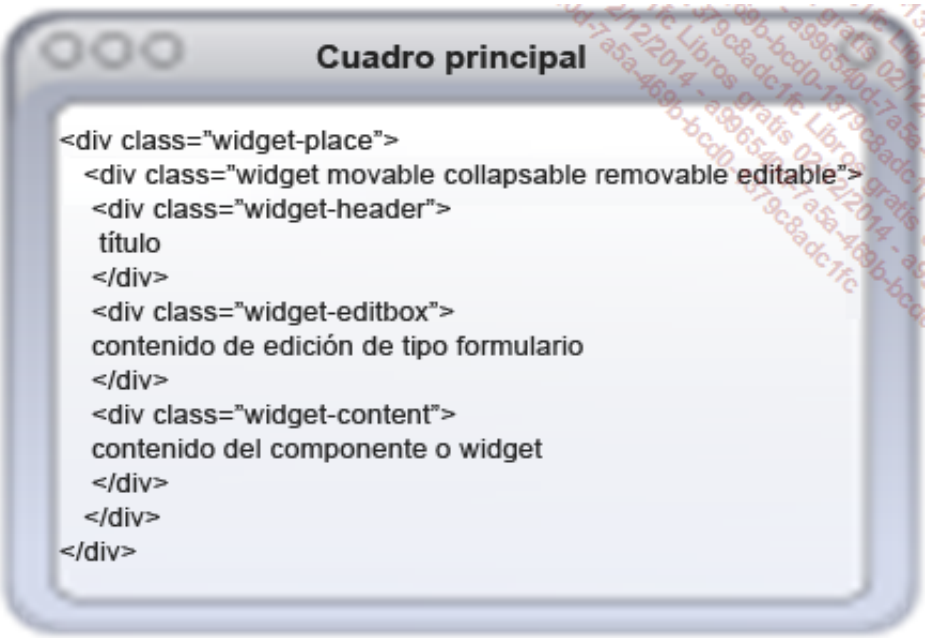
Los componentes o widgets se utilizan cada vez más en las aplicaciones de Internet. Así, encontramos cuadros que se pueden ampliar, desplazar, cerrar y cuya utilización es similar a la de los cuadros en los programas informáticos. El complemento *JQuery easywidgets* permite crear componentes avanzados. Este complemento utiliza una hoja de estilos específica, así como un archivo JavaScript *jquery.easywidgets.js* e imágenes, para los cuadros.



Árbol JavaScript del complemento JQuery easywidgets

A continuación se realiza la gestión del complemento con código JavaScript ubicado en nuestro archivo */javascript/ejemplo26.js*. Para crear un widget, debemos utilizar las etiquetas `<div/>` con las clases específicas y utilizar un nombre único con el atributo `id` para cada widget con el fin de poder hacer referencia a los mismos. El cuadro debe estar asociado con la clase CSS `widget-place`, y el cuadro secundario con la clase `widget` y con las diferentes propiedades que deben gestionarse (desplazable, editable, eliminable...). El bloque del título está asociado con el estilo `widget-header`, el cuadro secundario cuenta con el estilo `widget-editbox` y el contenido principal está vinculado con el estilo `widget-content`.

En resumen, la estructura de un widget (componente) es la siguiente:



Estructura de las etiquetas y estilos CSS del complemento easywidget

Vamos a modificar nuestra página principal insertando el título en la etiqueta `widget-header`, el formulario de registro en la parte *widget-editbox* y la tabla de lista en la sección *widget-content*. Además, agregaremos un segundo widget de ayuda para mostrar el alcance de los desplazamientos de los cuadros. Para ocultar o mostrar todos los cuadros se insertan dos vínculos. A continuación se indica el nuevo código de la página JSP.

```
Código: /jsp/IndiceCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Lista de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
<!-- Utilizar la librería JavaScript JQuery -->
<script src="javascript/jquery.min.js"
type="text/javascript"></script>
<script src="javascript/jquery.form.js"
type="text/javascript"></script>
<!--  archivo de gestión -->
<script src="javascript/ejemplo26.js"
type="text/javascript"></script>
<!--  Utilizar la librería JavaScript JQueryUI -->
<style type="text/css">@import url(javascript/plugin/jqueryui/css/
smoothness/jquery-ui-1.7.1.custom.css);</style>
<style type="text/css">@import url(javascript/plugin/jqueryui/css/
css.css);</style>
<script type="text/javascript" src="javascript/plugin/jqueryui/js/
jquery-ui-1.7.1.custom.min.js"></script>
<script type="text/javascript"
src="javascript/ejemplojqueryui.js"></script>
<!--  Utilizar la librería JavaScript easywidgets -->
<style type="text/css">@import
url(javascript/plugin/easywidgets/css/jquery.easywidgets.css);</style>
<style type="text/css">@import
url(javascript/plugin/easywidgets/css/example.css);</style>
<script type="text/javascript" src="javascript/plugin/easywidgets/
js/jquery.easywidgets.js"></script>
<script type="text/javascript" src="javascript/plugin/easywidgets/
js/jquery-ui-min.js"></script>
</head>
<body>
```

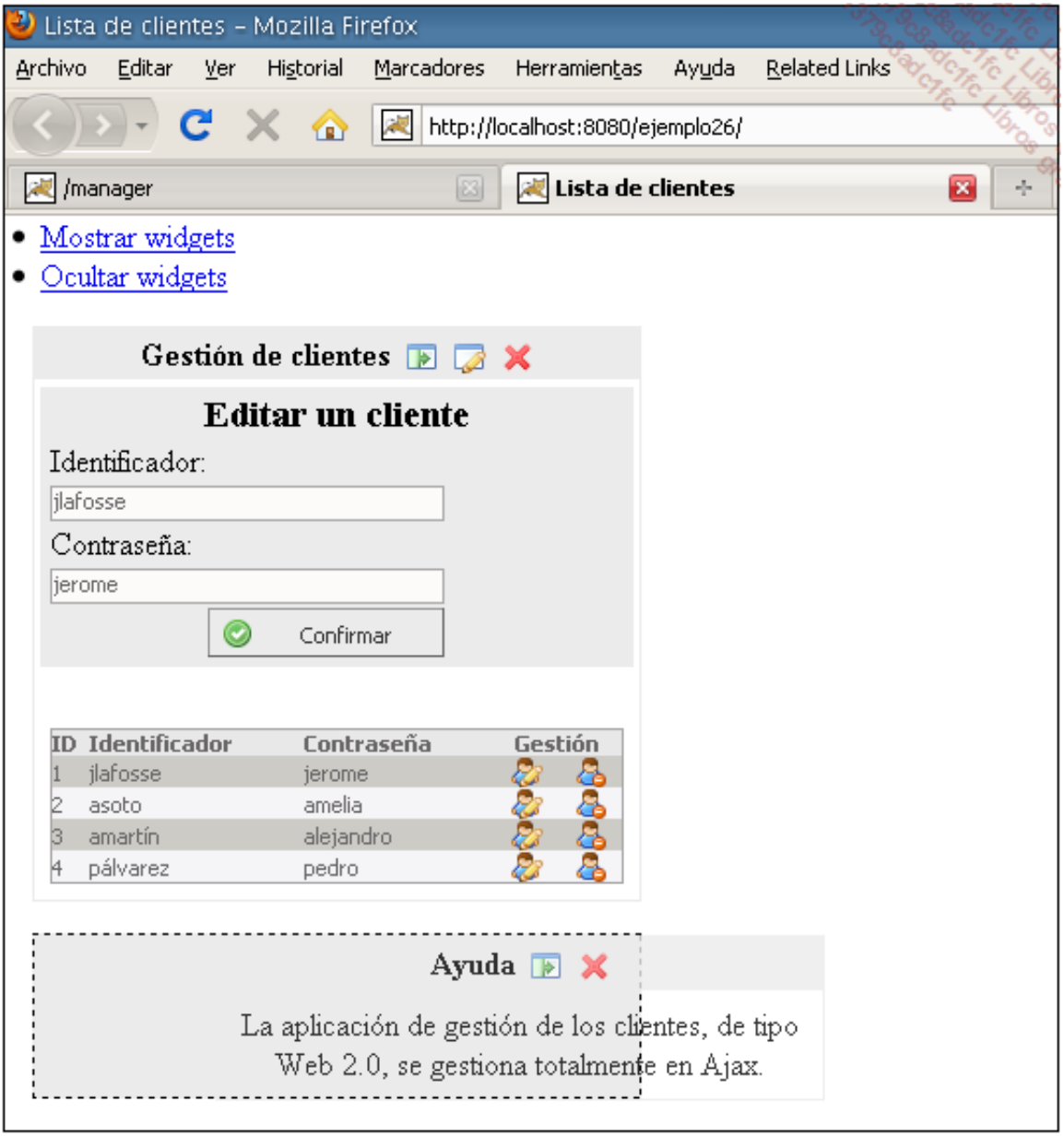
```
<!--  etiqueta utilizada para mostrar las respuestas Ajax -->
<div id="CuadroRespuestaAjax"></div>

<div align="left" style="margin-left:20px">
<ul>
      <li><a onClick="$.fn.ShowEasyWidgets(); return false"
href="#" title="Mostrar widgets">Mostrar
widgets</a></li>
      <li><a onClick="$.fn.HideEasyWidgets(); return false"
href="#" title="Ocultar widgets">Ocultar widgets</a></li>
</ul>
</div>

<!-- Widget 1 -->
<div class="widget-place" id="widget-place-1">
  <div class="widget movable collapsable removable closeconfirm
editable collapse" id="identificarwidget-1">
    <div class="widget-header"><strong>Gesti&oacute;n de
clientes</strong></div>
    <div class="widget-editbox">
      <!--  etiqueta utilizada para agregar/modificar un cliente en
Ajax -->
        <div id="CuadroFormularioCliente"></div>
    </div>
    <div class="widget-content">
         <!--  etiqueta utilizada para mostrar la lista de clientes
en Ajax -->
         <div id="CuadroListadoCliente"></div>
    </div>
  </div>
</div>

<!-- Widget 2 -->
<div class="widget-place" id="widget-place-2">
<div class="widget movable collapsable removable closeconfirm
collapse" id="identificarwidget-2">
    <div class="widget-header"><strong>Ayuda</strong></div>
    <div class="widget-content">
         La aplicaci&oacute;n de gesti&oacute;n de los clientes,
de tipo Web 2.0, se gestiona totalmente en Ajax.
    </div>
  </div>
</div>

<!--  cuadro utilizado para las confirmaciones -->
<div id="CuadroConfirmacion" title="Gestión de clientes"
style="display:none">
      <p><span class="ui-icon ui-icon-alert" style="float:left;
margin:0 7px 20px 0;"></span>&iquest;Desea eliminar el cliente?</p>
</div>
</body>
</html>
```

La gestión de los widgets se realiza con la ayuda del archivo JavaScript *ejemplo26.js*. Agregaremos el bloque de internacionalización i18n de gestión de textos e imágenes mostrados para la manipulación de los cuadros. Asimismo, utilizamos el servicio de gestión de cookies, que permite conservar la ubicación de nuestros widgets (la ubicación de los cuadros) en la siguiente utilización.

```
// Gestión de widgets
$(function(){

  // Gestión de la internacionalización para los cuadros
  $.fn.EasyWidgets({
```

```
    i18n : {
      editText : '<img
src="./javascript/plugin/easywidgets/imagenes/edit.png" alt="Edit"
width="16" height="16" title="Editar"/>',
      closeText : '<img
src="./javascript/plugin/easywidgets/imagenes/close.png" alt="Close"
width="16" height="16" title="Cerrar"/>',
      collapseText : '<img
src="./javascript/plugin/easywidgets/imagenes/collapse.png"
alt="Close" width="16" height="16" title="Cerrar esta herramienta"/>',
      cancelEditText : '<img src="./javascript/plugin/easywidgets/
images/edit.png" alt="Edit" width="16" height="16" />',
      extendText : '<img
src="./javascript/plugin/easywidgets/imagenes/extend.png"
alt="Close" width="16" height="16" title="Abrir esta herramienta"/>',
      confirmMsg : '¿Cerrar esta herramienta?',
    }
  });

  // Utilización del complemento cookie para memorizar las ubicaciones
  $.fn.EasyWidgets({

    behaviour : {
      useCookies : true
    }

  });
});
```

El servicio está totalmente operativo y ahora podemos desplazar los cuadros, abrirlos, mostrar el formulario de edición, etc. También podemos colocar los cuadros a nuestro gusto, cerrar el navegador y volver a la página para comprobar que las ventanas siguen colocadas correctamente.



Utilización de widgets para los formularios

4. Utilización de herramientas dinámicas

Para esta parte del proyecto de Struts 2 en Ajax, vamos a crear un nuevo proyecto, *ejemplo27*, a partir del anterior y a agregar dos atributos a la clase JavaBean `Cliente` con el fin de gestionar la fecha de nacimiento del cliente y un campo de descripción para sus datos (cv, competencias...).

Estas dos nuevas propiedades se asociarán a continuación con los servicios dinámicos DHTML para mostrar un calendario para la fecha de nacimiento. El modelo se modifica ligeramente para la inicialización y la modificación de los objetos del cliente.

```
Código: /ejemplo27/.javabeans.Cliente.java
package ejemplo27.javabeans;

@SuppressWarnings("serial")
public class Cliente {

    private int idCliente;
    private String identificador;
    private String contrasena;
```

```
    private String fechadenacimiento;
    private String descripcion;

      public Cliente() {

    }

      public Cliente(int idCliente,String identificador, String
contrasena, String fechadenacimiento, String descripcion){
      this.idCliente=idCliente;
      this.identificador=identificador;
      this.contrasena=contrasena;
      this.fechadenacimiento=fechadenacimiento;
      this.descripcion=descripcion;
    }

     // getter y setter
      ...
}
```

```
Código: /ejemplo27/modelo.ClienteModelo.java
package ejemplo27.modelo;

import java.util.ArrayList;
import java.util.List;
import ejemplo27.javabeans.Cliente;

public class ClienteModelo {
    private static List<Cliente> listaClientes;
    private static int id=1;

    static
    {
      listaClientes=new ArrayList<Cliente>();
      listaClientes.add(new Cliente(id++, "jlafosse", "jerome",
"01/01/1968", "informático"));
      listaClientes.add(new Cliente(id++, "asoto", "amelia",
"02/02/2004", "contable"));
      listaClientes.add(new Cliente(id++, "amartín", "alejandro",
"16/05/1954", "repartidor"));
      listaClientes.add(new Cliente(id++, "pálvarez", "pedro",
"04/05/1992", "músico"));
    }

      // modificar un cliente de la lista
      public static void modificar(Cliente cliente) {
             int idCliente=cliente.getIdCliente();
             for(int i=0;i<listaClientes.size();i++)
             {
                   Cliente c=listaClientes.get(i);

                   if(c.getIdCliente()==idCliente)
                   {

      c.setIdentificador(cliente.getIdentificador());

      c.setContrasena(cliente.getContrasena());

      c.setFechadenacimiento(cliente.getFechadenacimiento());

      c.setDescripcion(cliente.getDescripcion());
                         break;
                   }
            }
```

```
      }
...
}
```

El código de la vista responsable de la visualización se modifica ligeramente para gestionar la fecha de nacimiento y la descripción del cliente.

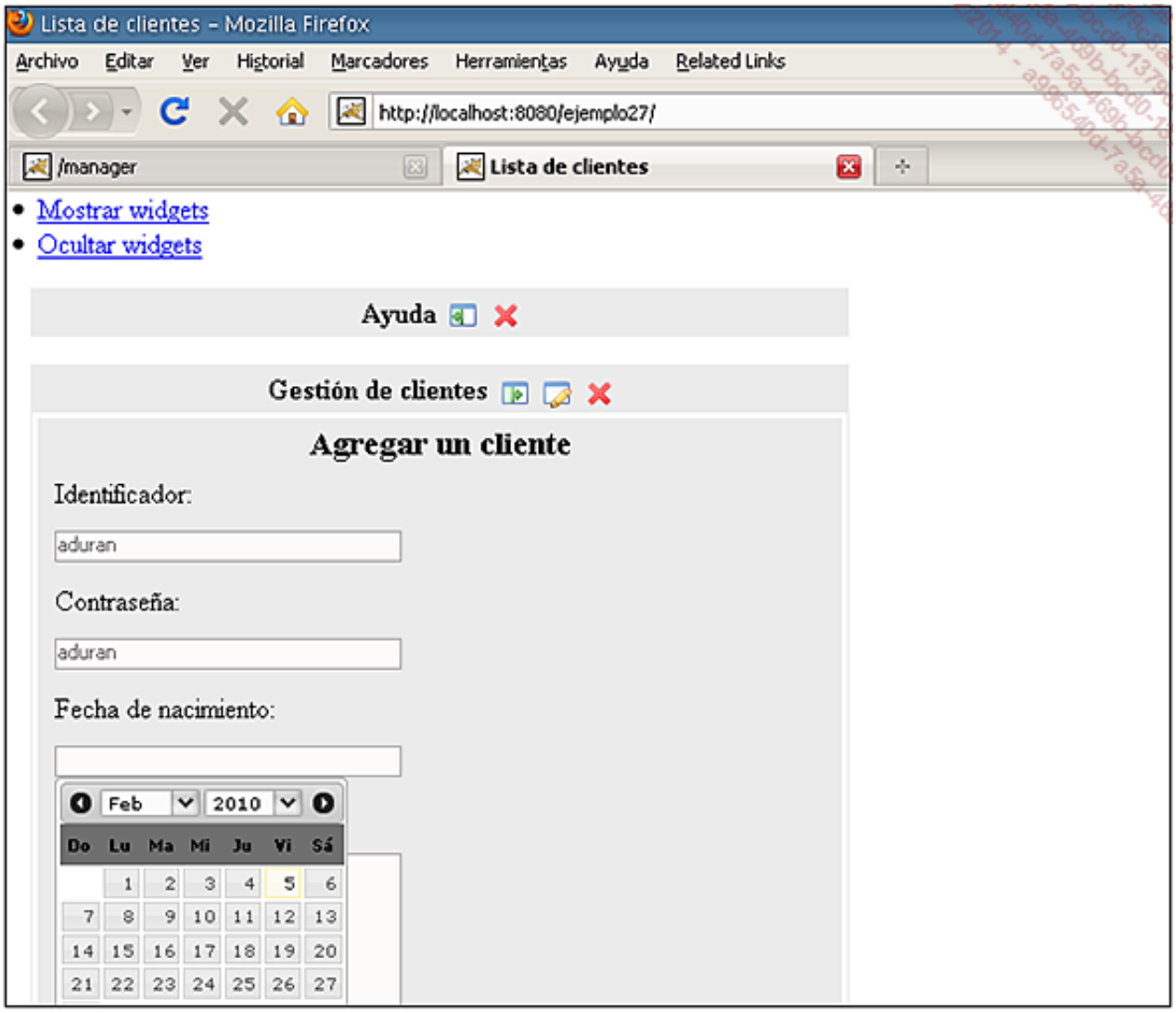
```
Código: /jsp/ListadoCliente.jsp
 <%@ taglib prefix="s" uri="/struts-tags" %>

<table border="0" id="tabla" cellpadding="0" cellspacing="0">
      <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td><b>Fecha de
nacimiento</b></td><td><b>Descripci&oacute;n</b></td><td colspan="2"
align="center"><b>Gestión</b></td></tr>
      <s:iterator value="listaClientes" status="linea">
      <s:if test="#linea.odd"><tr class="linea1"></s:if>
      <s:if test="#linea.even"><tr class="linea2"></s:if>
      <td><s:property value="idCliente"/></td>
      <td><s:property value="identificador"/></td>
      <td><s:property value="contrasena"/></td>
      <td><s:property value="fechadenacimiento"/></td>
      <td><s:property value="descripcion"/></td>
      <td align="center"><a href="JavaScript:modificarCliente('$
{idCliente}')"><img src="imagenes/editarcliente.png" alt="Editar"
title="Editar" border="0"/></a></td>
      <td align="center"><a
href="JavaScript:confirmarEliminarCliente('${idCliente}')"><img
src="imagenes/eliminarcliente.png" alt="Eliminar" title="Eliminar"
border="0"/></a></td>
      </tr>
      </s:iterator>
</table>
```

Ahora podemos pasar a la parte dinámica del formulario del cliente, modificando la página JSP*/jsp/FormularioCliente.jsp* con el fin de agregar un calendario dinámico gestionado por el complemento *JQuery UI* (http://jqueryui.com/demos/datepicker/#default). Para ello, debemos agregar el código JavaScript que permite aplicar el calendario a un objeto a partir de su atributo `id` y configuraremos el idioma en español con la ayuda del código JavaScript y de la librería *jquery-ui-i18n.js*. El tamaño del calendario es gestionado por el estilo css *ui-datepicker-div*.

```
#ui-datepicker-div
{
      font-size:8%;
}

Código : /jsp/FormularioCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>

<!--  formulario de adición o modificación -->
<s:if test="cliente.idCliente!=0">
<h3>Editar un cliente</h3>
<s:set id="accion">Modificar_Cliente.action</s:set>
</s:if>
<s:else>
<h3>Agregar un cliente</h3>
<s:set id="accion">Agregar_Cliente.action</s:set>
</s:else>

<s:form method="post" action="%{accion}" id="Formulario_Cliente"
name="Formulario_Cliente">
<s:hidden key="cliente.idCliente"/>
<s:textfield name="cliente.identificador" id="cliente.identificador"
```

```
label="Identificador" labelposition="top" cssClass="input"/>
<s:textfield name="cliente.contrasena" id="cliente.contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
<s:textfield name="cliente.fechadenacimiento"
id="cliente_fechadenacimiento" label="Fecha de nacimiento"
labelposition="top" cssClass="input"/>
<s:textarea  name="cliente.descripcion" id="cliente.descripcion"
label="Descripción" labelposition="top" cssClass="textarea"/>
<s:submit value="Confirmar" id="botonconfirmar"/>
</s:form>

<script type="text/javascript" src="javascript/plugin/jqueryui/js/
i18n/jquery-ui-i18n.js"></script>
<script type="text/javascript">
      $(function() {
            // mostrar los meses y años
            $("#cliente_fechadenacimiento").datepicker({
                  changeMonth: true,
                  changeYear: true
            }
            );
            // cambiar el idioma a español
            $
("#cliente_fechadenacimiento").datepicker($.datepicker.regional['es']
);
            $("#cliente_fechadenacimiento").datepicker('option',
$.extend({showMonthAfterYear: false},
$.datepicker.regional['es']));
      });
</script>
```

La notación de puntos (cliente.fechadenacimiento) no está permitida para acceder a un objeto con el framework JQuery. En el ejemplo utilizamos el guión bajo (_).



Utilización de un calendario dinámico

5. Utilización de un servicio de autocompletado para las búsquedas

La gestión del autocompletado o proposición en las búsquedas es un servicio habitual en las aplicaciones de Internet Web 2.0. Para ejemplificar este servicio, vamos a utilizar de nuevo un complemento *JQuery* denominado *autocomplete* con sus archivos JavaScript y su hoja de estilos.

La configuración comienza por la instalación de los archivos en el directorio */javascript/plugin* de la aplicación *ejemplo27*. La vista especializada de la visualización de la lista de clientes se modifica con el fin de gestionar el formulario de búsqueda.

```
Código: /jsp/ListadoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>

<!-- Formulario de búsqueda -->
<div id="buscar" align="left">
 <a href="javascript:mostrarBusqueda();"><img
src="imagenes/lupabusqueda.png" title="Hacer clic para mostrar
el formulario de búsqueda" alt="Buscar" align="absmiddle"
border="0"/></a>
</div>
<s:form method="post" id="formulariobusqueda"
name="formulariobusqueda" theme="simple">
<table cellspacing="4" cellpadding="0" id="tabla" width="440px"
border="0">
```

```
<tr>
<td>Buscar: <s:textfield name="busqueda" id="busqueda"
label="Búsqueda" labelposition="left" cssClass="input"/></td>
<td>
<select name="tipobusqueda" id="tipobusqueda"
onchange="cambiarTipoBusqueda();" class="listadesplegable">
      <option value="cliente.identificador">Identificador</option>
      <option value="cliente.contrasena">Contraseña</option>
      <option value="cliente.fechadenacimiento">Fecha de
nacimiento</option>
</select>
</td>
<td><input type="imagen" src="imagenes/buscar.gif"
align="absmiddle"
onclick="JavaScript:listadoCliente('busqueda')"/></td>
</tr>
</table>
</s:form>
<table border="0" id="tabla" cellpadding="0" cellspacing="0">
      <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td><b>Fecha de
nacimiento</b></td><td><b>Descripci&oacute;n</b></td><td colspan="2"
align="center"><b>Gestión</b></td></tr>
      <s:iterator value="listaClientes" status="linea">
      <s:if test="#linea.odd"><tr class="linea1"></s:if>
      <s:if test="#linea.even"><tr class="linea2"></s:if>
      <td><s:property value="idCliente"/></td>
      <td><s:property value="identificador"/></td>
      <td><s:property value="contrasena"/></td>
      <td><s:property value="fechadenacimiento"/></td>
      <td><s:property value="descripcion"/></td>
      <td align="center"><a href="JavaScript:modificarCliente('$
{idCliente}')"><img src="imagenes/editarcliente.png" alt="Editar"
title="Editar" border="0"/></a></td>
      <td align="center"><a
href="JavaScript:confirmarEliminarCliente('${idCliente}')"><img
src="imagenes/eliminarcliente.png" alt="Eliminar" title="Eliminar"
border="0"/></a></td>
      </tr>
      </s:iterator>
</table>

<script type="text/javascript">
// Gestión de búsquedas
$(function() {
      if($("#tipobusqueda") != null)
      {
            // Desencadenamiento de la acción para inicializar
el autocompletado
            cambiarTipoBusqueda();

            var busqueda=$('#busqueda').val();

            if(busqueda== null || busqueda== '')
            {
                  $("#formulariobusqueda").hide();
            }
            else
            {
                  $("#formulariobusqueda").show();
            }
      }
});
</script>
```

La página responsable de mostrar los términos del autocompletado es muy simple y permite mostrar cada respuesta con la ayuda de una coleccion.

Código: /jsp/AutoComplete.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-
1" pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<s:iterator value="lista" >
      <s:property/>
</s:iterator>
```

El código JavaScript de gestión del formulario se modifica con el fin de gestionar la visualización del formulario de búsqueda y la confirmación del envío. La función *listadoCliente()* gestiona a partir de ahora la búsqueda para el autocompletado.

Código: /javascript/ejemplo27.js

```
...
// iniciar la búsqueda por tipo de campo
function cambiarTipoBusqueda()
{
      var atributo=$('#tipobusqueda').val();

      if(atributo!=null)
      {
            atributo=jQuery.trim(atributo);

            if(atributo!='')
            {
                  url="AutoComplete.action?atributo="+atributo;
                  $("#busqueda").autocomplete(url, {
                          delay: 400,
                          width:400,
                          cacheLength:1,
                          matchSubset:false,
                          mustMatch : true,
                          minChars:1,
                          autoFill: false
                      });
            }
      }
}

// visualización del formulario de búsqueda
function mostrarBusqueda()
{
      if($("#formulariobusqueda").is(":hidden"))
      {
      $("#formulariobusqueda").slideDown("fast");
      }
   else
      {
         $("#formulariobusqueda").slideUp("fast");
      }
}

// devolver la lista de clientes utilizando Ajax
function listadoCliente()
{
      //búsqueda del usuario
      var busqueda=$("#busqueda").val();
      var atributo=$('#tipobusqueda').val();
      // no busqueda
      if($("#busqueda").html()==null || $
```

```
("#tipobusqueda").html()==null)
      {
            busqueda="";
            atributo="";
      }

      //enviar los datos en POST
      $.ajax(
      {
            type: "POST",
            url: "Listado_Cliente.action",
            dataType: "html",
            data:
"busqueda="+busqueda+"&atributo="+atributo+"&idClienteActual=0",
            timeout : 8000,
            error: function(){
        alert('Desconexion del servidor');
        },
            beforeSend : function()
            {
                  $("#CuadroListadoCliente").html('<img
src="imagenes/cargacircularpequena.gif" align="absmiddle"
width="16" height="16" style="padding-right:5px;"/>Espere...');
            },
            success: function(html)
            {
                  // incluir el resultado recibido en la etiqueta de
la página actual
          $("#CuadroListadoCliente").html(html);
          if(busqueda!="")
                  {
             // gestión dinámica de mensajes
                  //mensajeDinámico(false,true);
               }
            }
      });
}
// realizar listado de todos los clientes
function listadoClienteInicializar()
{
      // vaciar los campos y desencadenar la lista
      $("#busqueda").val("");
      $('#tipobusqueda').val("");
      listadoCliente();
}
...
```

Por último, el modelo *ClienteModelo* se modifica para gestionar las búsquedas del usuario en la lista de clientes y se devuelven únicamente los objetos relevantes.

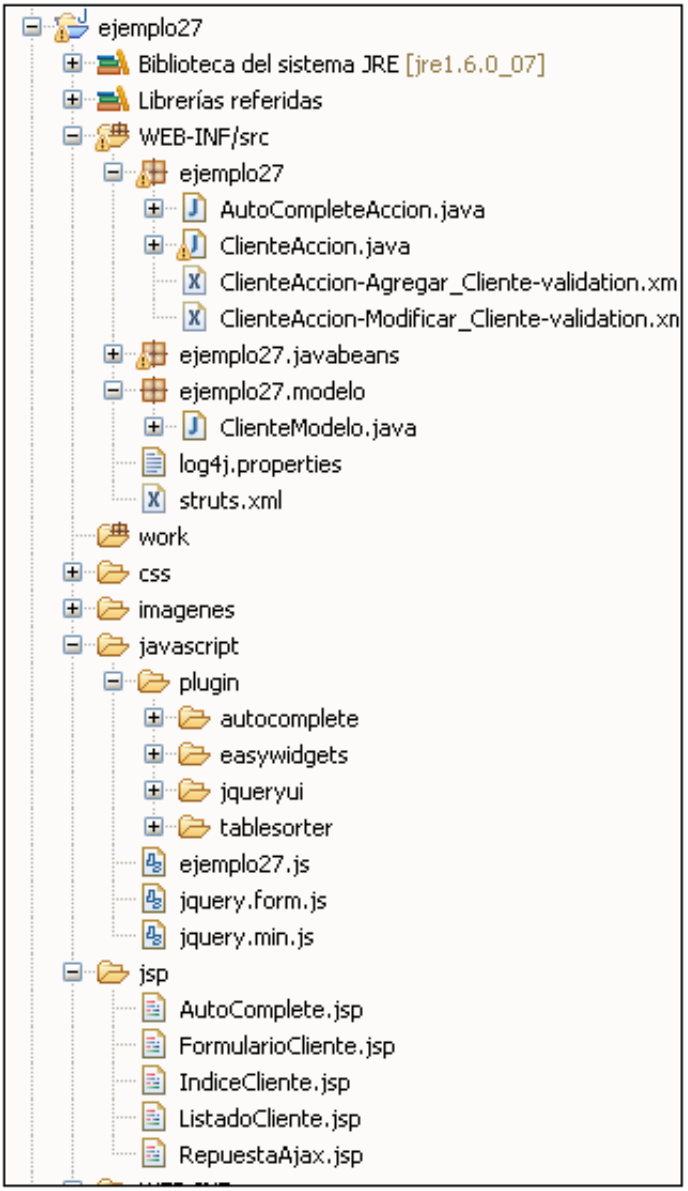
```
Código: /ejemplo27/modelo/ClienteModelo.java
package ejemplo27.modelo;

import java.util.ArrayList;
import java.util.List;
import ejemplo27.javabeans.Cliente;

public class ClienteModelo {
    private static List<Cliente> listaClientes;
    private static int id=1;

    static
    {
       listaClientes=new ArrayList<Cliente>();
```

```
       listaClientes.add(new Cliente(id++, "jlafosse", "jerome",
"01/01/1968", "informático"));
       listaClientes.add(new Cliente(id++, "asoto", "amelia",
"02/02/2004", "contable"));
       listaClientes.add(new Cliente(id++, "amartín", "alejandro",
"16/05/1954", "repartidor"));
       listaClientes.add(new Cliente(id++, "pálvarez", "pedro",
"04/05/1992", "músico"));
    }


    // devolver la lista de clientes
    public static List<Cliente> getListaClientes(String busqueda,
String atributo) {

      // lista para las búsquedas
      List<Cliente> listaClientesBusqueda=new ArrayList<Cliente>();

      // búsqueda usuario
      if(busqueda!=null && !busqueda.equalsIgnoreCase(""))
      {
             // examinar cada cliente
             for(Cliente c : listaClientes)
             {
                    // búsqueda en contraseña
                    if(atributo.equals("cliente.contrasena"))
                    {

      if(c.getContrasena().contains(busqueda))
                          {
                                listaClientesBusqueda.add(c);
                          }
                    }
                    // búsqueda en la fecha
                    else
if(atributo.equals("cliente.fechadenacimiento"))
                    {

      if(c.getFechadenacimiento().contains(busqueda))
                          {
                                listaClientesBusqueda.add(c);
                          }
                    }
                    // búsqueda en el identificador
                    else
                    {

      if(c.getIdentificador().contains(busqueda))
                          {
                                listaClientesBusqueda.add(c);
                          }
                   }
             }
             // devolver la nueva lista de clientes
             return listaClientesBusqueda;
      }

             return listaClientes;
      }

      public static void setListaClientes(List<Cliente>
listaClientes) {
              ClienteModelo.listaClientes = listaClientes;
      }
```

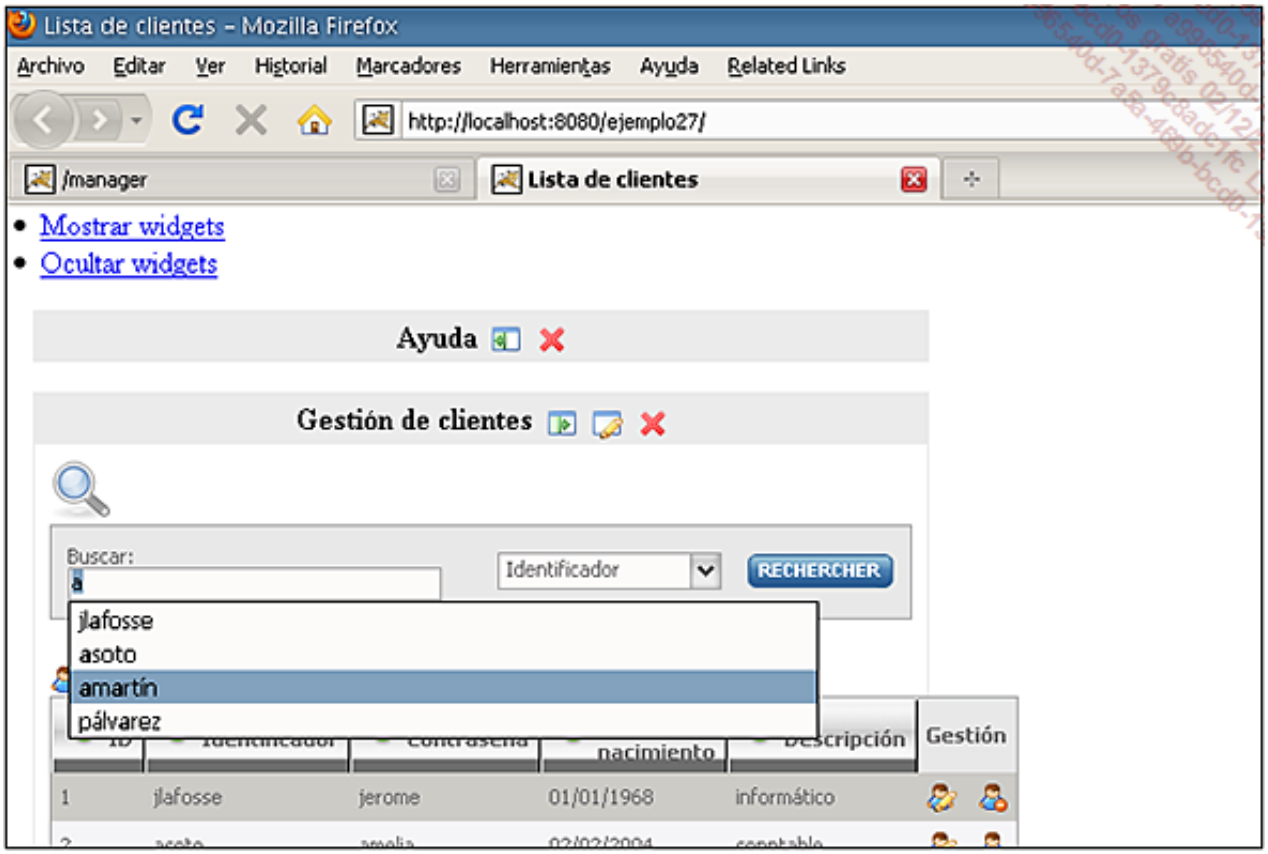
```
      // agregar un cliente a la lista
      public static void agregar(Cliente cliente) {
        cliente.setIdCliente(id++);
        listaClientes.add(cliente);
    }

      // eliminar un cliente de la lista
      public static void eliminar(int idCliente) {
             for(int i=0;i<listaClientes.size();i++)
             {
                    Cliente c=listaClientes.get(i);
                    if(c.getIdCliente()==idCliente)
                    {
                          listaClientes.remove(c);
                    }
             }
      }

      // modificar un cliente de la lista
      public static void modificar(Cliente cliente) {
             int idCliente=cliente.getIdCliente();
             for(int i=0;i<listaClientes.size();i++)
             {
                   Cliente c=listaClientes.get(i);

                   if(c.getIdCliente()==idCliente)
                   {

      c.setIdentificador(cliente.getIdentificador());

      c.setContrasena(cliente.getContrasena());

      c.setFechadenacimiento(cliente.getFechadenacimiento());

      c.setDescripcion(cliente.getDescripcion());
                        break;
                   }
             }
      }

      // buscar un cliente en la lista
      public static Cliente getCliente(int idCliente) {
             for(int i=0;i<listaClientes.size();i++)
             {
                   Cliente c=listaClientes.get(i);
                   if(c.getIdCliente()==idCliente)
                   {
                         return c;
                   }
             }
             return null;
      }


      // realizar una búsqueda en las listas
      public static ArrayList<String> buscar(String introduccion,
String atributo) {

             // lista de búsquedas
             ArrayList<String> listaBusquedas=new
ArrayList<String>();

             // examinar cada cliente
             for(Cliente c : listaClientes)
             {
```

```
                    // búsqueda en contraseña
                    if(atributo.equals("cliente.contrasena"))
                    {

      if(c.getContrasena().contains(introduccion))
                          {

      listaBusquedas.add(c.getContrasena().trim());
                          }
                     }
                     // buscar en la fecha
                     else
if(atributo.equals("cliente.fechadenacimiento"))
                    {

      if(c.getFechadenacimiento().contains(introduccion))
                          {

      listaBusquedas.add(c.getFechadenacimiento());
                          }
                    }
                    // búsqueda en el identificador
                    else
                    {

      if(c.getIdentificador().contains(introduccion))
                            {

      listaBusquedas.add(c.getIdentificador().trim());
                            }
                    }
            }

            return listaBusquedas;
      }

}
```

La aplicación está operativa y las búsquedas por autocompletado le permiten beneficiarse de un proyecto más ergonómico y funcional.



Árbol del proyecto ejemplo27



Utilización del autocompletado para las búsquedas

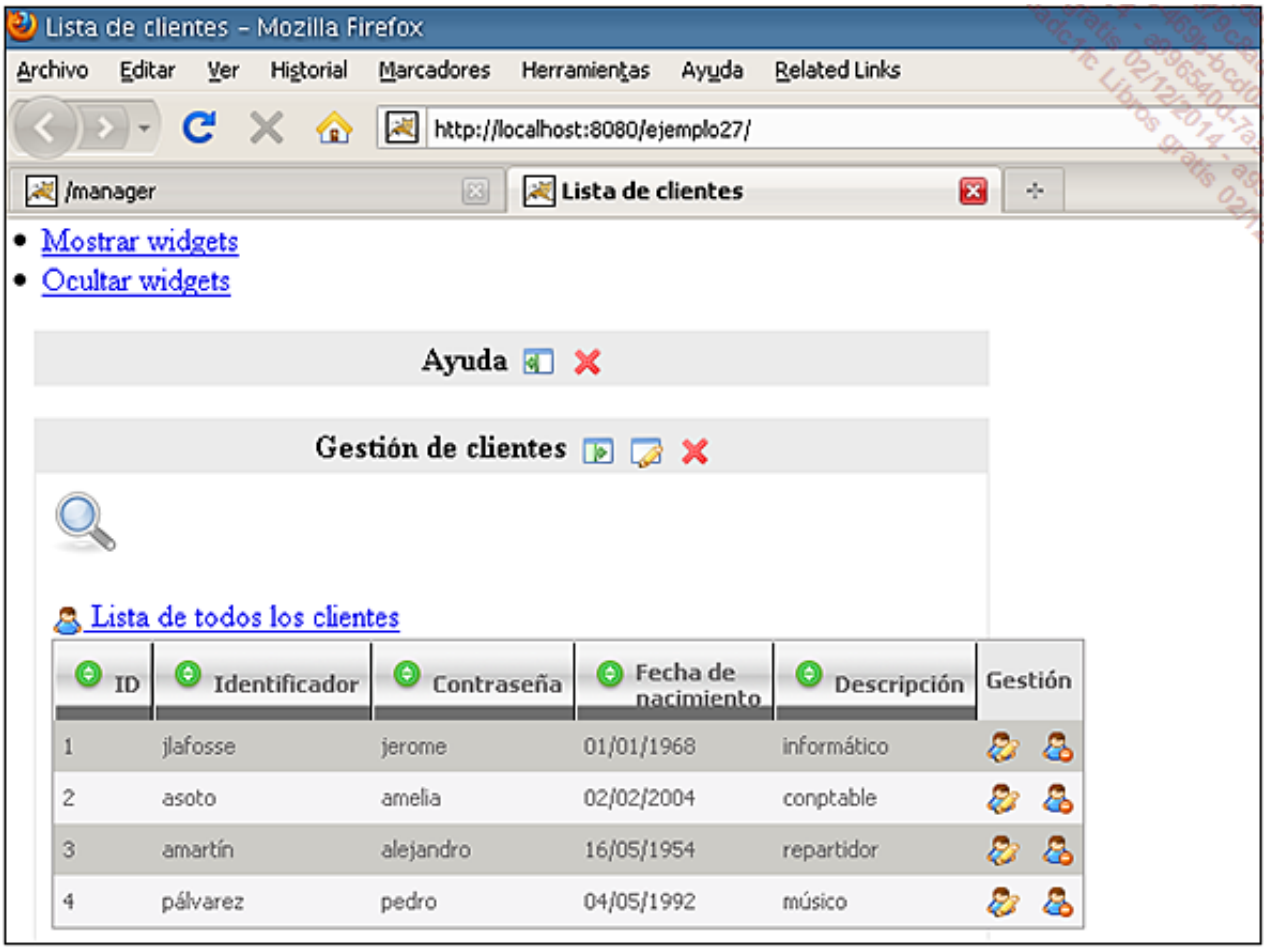
6. Gestión de clasificaciones dinámicas

El servicio termina con el uso del complemento *JQuery tablesorter* (http://tablesorter.com/docs/), que permite clasificar las columnas de una tabla. Este complemento se instala en el directorio*/javascript/plugin/tablesorter*. Para activar las clasificaciones, las tablas deben contar con un identificador único (parámetro *id*), así como con un formato a partir de etiquetas `<thead/>` y`<tbody/>`. Nuestra página de visualización se modifica ligeramente para respetar esta estructura y colocar en JavaScript las clasificaciones de la tabla de clientes.

```
Código: /jsp/ListadoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<!-- Formulario de búsqueda -->
<div id="buscar" align="left">
 <a href="javascript:mostrarBusqueda();"><img
src="imagenes/lupabusqueda.png" title="Hacer clic para mostrar el
formulario de busqueda" alt="Buscar" align="absmiddle"
border="0"/></a>
</div>
<s:form method="post" id="formulariobusqueda"
name="formulariobusqueda" theme="simple">
<table cellspacing="4" cellpadding="0" class="tabla" width="440px"
border="0">
<tr>
<td>Buscar: <s:textfield name="busqueda" id="busqueda"
label="Busqueda" labelposition="left" cssClass="input"/></td>
<td>
<select name="tipobusqueda" id="tipobusqueda"
onchange="cambiarTipoBusqueda();" class="listadesplegable">
      <option value="cliente.identificador">Identificador</option>
      <option value="cliente.contrasena">Contrase&ntilde;a</option>
      <option value="cliente.fechadenacimiento">Fecha de
```

```
nacimiento</option>
</select>
</td>
<td><input type="imagen" src="imagenes/buscar.gif"
align="absmiddle"
onclick="JavaScript:listadoCliente('busqueda')"/></td>
</tr>
</table>
</s:form>

<br/><div align="left"><a
href="JavaScript:listadoClienteInicializar();"><img
src="imagenes/cliente.png" align="absmiddle" border="0"/> Listado de todos
los clientes</a></div>

<table border="0" id="tablaCliente" cellpadding="0"
cellspacing="0">
      <thead><tr>
            <th>ID</th>
            <th>Identificador</th>
            <th>Contrase&ntilde;a</th>
            <th>Fecha de nacimiento</th>
            <th>Descripcion</th>
            <td colspan="2" align="center"><b>Gesti&oacue;n</b></td>
      </tr>
      </thead>
      <tbody>
      <s:iterator value="listaClientes" status="linea">
      <s:if test="#linea.odd"><tr class="linea1"></s:if>
      <s:if test="#linea.even"><tr class="linea2"></s:if>
      <td><s:property value="idCliente"/></td>
      <td><s:property value="identificador"/></td>
      <td><s:property value="contrasena"/></td>
      <td><s:property value="fechadenacimiento"/></td>
      <td><s:property value="descripcion"/></td>
      <td align="center"><a href="JavaScript:modificarCliente('$
{idCliente}')"><img src="imagenes/editarcliente.png" alt="Editar"
title="Editar" border="0"/></a></td>
      <td align="center"><a
href="JavaScript:confirmarEliminarCliente('${idCliente}')"><img
src="imagenes/eliminarcliente.png" alt="Eliminar" title="Eliminar"
border="0"/></a></td>
      </tr>
      </s:iterator>
      </tbody>
</table>

<script type="text/javascript">
// Gestión de busquedas
$(function() {
      if($("#tipobusqueda") != null)
      {
            // Desencadenamiento de la acción para inicializar
el autocompletado
            cambiarTipoBusqueda();

            var busqueda=$('#busqueda').val();

            if(busqueda== null || busqueda== '')
            {
                  $("#formulariobusqueda").hide();
            }
            else
            {
                  $("#formulariobusqueda").show();
```

```
            }
        }

        // Gestión de clasificaciones
        $("#tablaCliente").tablesorter();
});
</script>
```



Utilización del complemento JavaScript de gestión de clasificaciones

En resumen

En este capítulo se presenta la configuración de servicios Web 2.0 con la ayuda del lenguaje JavaScript y de la tecnología Ajax. Se detallan los principales servicios utilizados en las aplicaciones profesionales. Se explican los formularios XHTML sin recarga de páginas detalladamente, así como los cuadros de diálogo y mensajes. El párrafo siguiente propone la utilización de un calendario dinámico para la gestión de las fechas. Por último, los dos últimos párrafos le introducen a los mecanismos de autocompletado y las clasificaciones dinámicas sin necesidad de recargar las páginas.

Velocity

Los motores de plantillas permiten mejorar el código y separar la parte de procesamiento de datos de la presentación XHTML. Los motores de plantillas o de presentación aportan soluciones para la presentación mediante etiquetas dedicadas. Un motor permite insertar contenido dinámico y gestionar la división de las partes de las páginas. El objetivo de estos motores es separar el código de la presentación. El motor Velocity (http://velocity.apache.org/) del consorcio Apache es un producto OpenSource incluido con Struts 2 en el paquete *struts2-core-2.x.x.jar*.

Las aplicaciones utilizan por defecto las páginas JSP para la gestión de las vistas y de la presentación. Sin embargo, también es posible utilizar el motor Velocity (al igual que FreeMarker) para la manipulación de datos. El acceso a los datos y la comprensión de la herramienta son muy similares a los del lenguaje JSP.

Velocity permite colocar las vistas en la aplicación y empaquetarlas, al contrario que las páginas JSP. además, si queremos implementar una aplicación como complemento de Struts, Velocity permite incluir las vistas en el mismo paquete *.jar*.

Velocity utiliza el signo del dólar (*$*) para el acceso a los datos. Con Struts se incluye la definición de los resultados de tipo *velocity* en el archivo *struts-default.xml* y, por lo tanto, se pueden utilizar sin problemas.

Cómo usar Velocity

Al igual que JSP, Velocity proporciona objetos implícitos que pueden utilizarse sin creación previa.

- *response*: el objeto *HttpServletResponse*.
- *res*: el alias del objeto *response*.
- *request*: el objeto *HttpServletRequest*.
- *req*: el alias del objeto *request*.
- *session*: el objeto *HttpSession*.
- *application*: el objeto *ServletContext*.
- *base*: la ruta del contexto de la aplicación (*path*).
- *action*: el objeto *action*.
- *stack*: el valor de la pila de ejecución.

Velocity también ofrece varias etiquetas o tags. Estas etiquetas se parecen a los tags de Struts, pero la sintaxis es diferente y se basa en la utilización de caracteres *#nombre*.

Cuando las páginas utilizan únicamente tags de Velocity, la etiqueta de inclusión de la biblioteca de Struts no es obligatoria `<%@ taglib prefix="s" uri="/struts-tags" %>`. No es posible empaquetar proyectos con páginas JSP, ya que las páginas JSP necesitan un motor Java para ejecutarse.

Esta es una lista no exhaustiva de etiquetas utilizadas con Velocity: *#if*, *#else*, *#elseif*, *#end*, *#foreach*,*#include*, *#stextfield*, *#sform*.

Para utilizar Velocity con todas sus funcionalidades, es necesario descargar las siguientes bibliotecas en formato *.jar* e instalarlas en la carpeta */WEB-INF/lib* de la aplicación.

- *velocity-x.jar*: la biblioteca principal del motor de plantillas.
- *velocity-x-dep.jar*: la biblioteca con las dependencias.
- *velocity-tools-x.jar*: las herramientas para el motor de plantillas.
- *commons-digester-x.jar*: la biblioteca que permite leer archivos de configuración en formato XML.

Para ilustrar el uso del motor de plantillas Velocity, vamos a crear un nuevo proyecto con el nombre*ejemplo28* a partir de la aplicación *ejemplo14*. El archivo de gestión de la aplicación *struts.xml* se modifica para devolver resultados de tipo velocity.

```
Código : 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="true" />

    <package name="ejemplo28" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
```

```
class="ejemplo28.ClienteAccion" method="listado">
            <result
type="velocity">/velocity/ListadoCliente.vm</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo28.ClienteAccion" method="agregar">
            <result name="input"
type="velocity">/velocity/ListadoCliente.vm</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Editar_Cliente"
class="ejemplo28.ClienteAccion" method="editar">
                    <interceptor-ref
name="paramsPrepareParamsStack"/>
            <result name="success"
type="velocity">/velocity/EditarCliente.jsp</result>
        </action>

        <action name="Modificar_Cliente"
class="ejemplo28.ClienteAccion" method="modificar">
            <result name="input"
type="velocity">/velocity/EditarCliente.vm</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Eliminar_Cliente"
class="ejemplo28.ClienteAccion" method="eliminar">
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

    </package>
</struts>
```

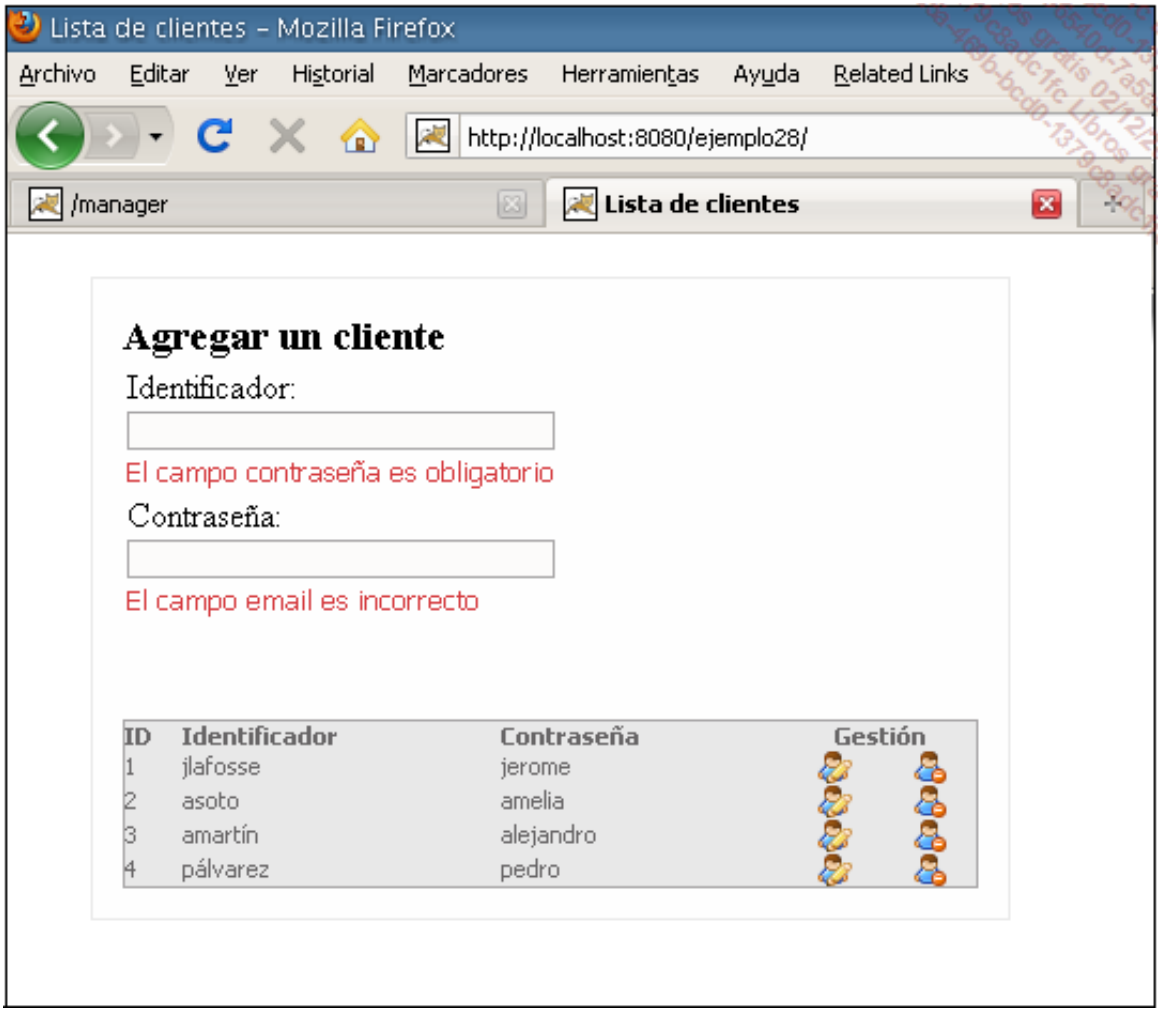
Ya no se utilizarán las páginas JSP, en su lugar usaremos archivos de plantillas Velocity en formato.*vm*, una sintaxis más simple que permite gestionar la visualización de los mensajes de Struts (errores de formularios, errores de acciones y confirmaciones).

```
Código: /velocity/ListadoCliente.vm
<html>
<head>
<title>Listado de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

       <h3>Agregar un cliente</h3>
       #sform ("action=Agregar_Cliente")
              #stextfield ("name=cliente.identificador"
"id=cliente.identificador" "label=Identificador" "labelposition=top"
"cssClass=input")
              #stextfield ("name=cliente.contrasena"
"id=cliente.contrasena" "label=Contraseña" "labelposition=top"
"cssClass=input")
              #ssubmit ("value=Agregar un cliente")
       #end

       <table border="0" id="tabla" cellpadding="0"
cellspacing="0">
```

```
	<tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td colspan="2"
align="center"><b>Gestion</b></td></tr>
	#foreach( $name in $listaClientes )
	<tr>
		<td>$name.idCliente</td>
		<td>$name.identificador</td>
		<td>$name.contrasena</td>
		<td align="center"><a href="Editar_Cliente.action?
idClienteActual=$name.idCliente"/><img
src="imagenes/editarcliente.png" alt="Editar" title="Editar"
border="0"/></a></td>
		<td align="center"><a href="Eliminar_Cliente.action?
idClienteActual=$name.idCliente"/><img src="imagenes/eliminarcliente.png"
alt="Eliminar" title="Eliminar" border="0"/></a></td>
	</tr>
	#end
	</table>
</div>
</body>
</html>
```

Código: /velocity/EditarCliente.vm

```
<html>
<head>
<title>Editar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">

	<h3>Editar un cliente</h3>
	#sform ("action=Modificar_Cliente")
		#stextfield ("name=cliente.identificador"
"id=cliente.identificador" "label=Identificador" "labelposition=top"
"cssClass=input")
		#stextfield ("name=cliente.contrasena"
"id=cliente.contrasena" "label=Contraseña" "labelposition=top"
"cssClass=input")
		#ssubmit ("value=Modificar un cliente")
	#end
</div>
</body>
</html>
```



Gestionar los clientes con el motor de plantillas Velocity

FreeMarker

FreeMarker (http://freemarker.org/) al igual que Velocity, es un motor de plantillas que puede utilizarse con Struts. De hecho, la biblioteca de etiquetas facilitada por Struts está basada en este motor. Para utilizar FreeMarker con Struts no es necesario disponer ninguna biblioteca adicional ya que la herramienta está incluida con el framework y la biblioteca *freemarker-x.jar*. Al igual que Velocity, FreeMarker permite empaquetar aplicaciones en un archivo con formato *.jar*.

FreeMarker ofrece los mismos objetos implícitos que los utilizados por Velocity. Las tags de FreeMarker se pueden utilizar además de las etiquetas de Struts y quien en la siguiente sintaxis: *<@s.nombre/>*. Los archivos de FreeMarker tienen la extensión de formato *.ftl* y a los tipos de resultados se les llama *freemarker*. Vamos a crear un nuevo proyecto con el nombre *ejemplo29* a partir del anterior utilizando FreeMarker como motor de plantillas en lugar de JSP o Velocity. Teniendo en cuenta que desarrollamos con el modelo de diseño MVC, únicamente se cambiará la visualización y su enrutamiento sin afectar a toda la aplicación.

```
Código : 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="true" />

    <package name="ejemplo29" namespace="/" extends="struts-
default">
        <default-action-ref name="Listado_Cliente" />

        <action name="Listado_Cliente"
class="ejemplo29.ClienteAccion" method="listado">
            <result
type="freemarker">/freemarker/ListadoCliente.ftl</result>
        </action>

        <action name="Agregar_Cliente"
class="ejemplo29.ClienteAccion" method="agregar">
            <result name="input"
type="freemarker">/freemarker/ListadoCliente.ftl</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Editar_Cliente"
class="ejemplo29.ClienteAccion" method="editar">
                    <interceptor-ref
name="paramsPrepareParamsStack"/>
            <result name="success"
type="freemarker">/freemarker/EditarCliente.ftl</result>
        </action>

        <action name="Modificar_Cliente"
class="ejemplo29.ClienteAccion" method="modificar">
            <result name="input"
type="freemarker">/freemarker/EditarCliente.ftl</result>
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

        <action name="Eliminar_Cliente"
```

```
class="ejemplo29.ClienteAccion" method="eliminar">
            <result name="success"
type="redirectAction">Listado_Cliente</result>
        </action>

    </package>
</struts>
```

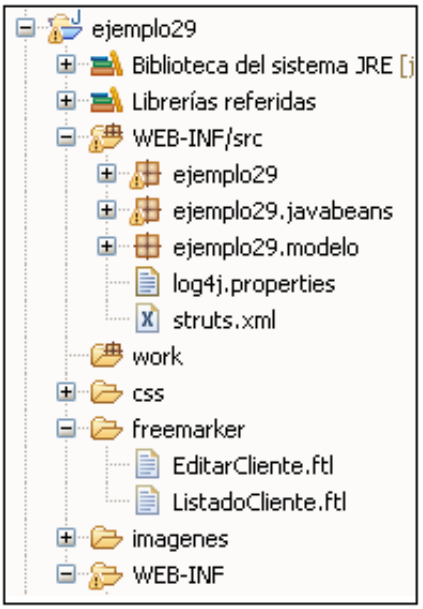
Código: /freemarker/ListadoCliente.ftl

```
<html>
<head>
<title>Listado de clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>

<div id="carta">

       <h3>Agregar un cliente</h3>
       <@s.form method="post" action="Agregar_Cliente">
               <@s.hidden key="cliente.idCliente"/>
               <@s.textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
               <@s.textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
               <@s.submit value="Agregar un cliente"/>
       </@s.form>

       <table border="0" id="tabla" cellpadding="0"
cellspacing="0">
       <tr><td><b>ID</b></td><td><b>Identificador</b></td><td><b>
Contraseña</b></td><td colspan="2"
align="center"><b>Gestión</b></td></tr>
       <@s.iterator value="listaClientes" status="linea">
       <@s.if test="#linea.odd"><tr class="linea1"></@s.if>
       <@s.if test="#linea.even"><tr class="linea2"></@s.if>
       <td><@s.property value="idCliente"/></td>
       <td><@s.property value="identificador"/></td>
       <td><@s.property value="contrasena"/></td>
       <td align="center"><a href="Editar_Cliente.action?
idClienteActual=${idCliente}"/><img src="imagenes/editarcliente.png"
alt="Editar" title="Editar" border="0"/></a></td>
       <td align="center"><a href="Eliminar_Cliente.action?
idClienteActual=${idCliente}"/><img src="imagenes/eliminarcliente.png"
alt="Eliminar" title="Eliminar" border="0"/></a></td>
       </tr>
       </@s.iterator>
       </table>
</div>
</body>
</html>
```

Código: /freemarker/EditarCliente.ftl

```
<html>
<head>
<title>Editar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>

<div id="carta">
```

```
	<h3>Editar un cliente</h3>
	<@s.form method="post" action="Modificar_Cliente">
		<@s.hidden key="cliente.idCliente"/>
		<@s.textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
		<@s.textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
		<@s.submit value="Modificar un cliente"/>
	</@s.form>
</div>
</body>
</html>
```



Árbol del proyecto ejemplo29

En resumen

Velocity es un motor de plantillas utilizado para la capa Vista del modelo MVC en lugar de JSP en una aplicación de Struts. Esta herramienta se ha explicado a partir de un ejemplo concreto. Struts también incluye y utiliza otro motor de plantillas llamado FreeMarker para su biblioteca de etiquetas. Esta biblioteca utilizada en una aplicación web es totalmente empaquetable y convertible al formato *.jar*.

Presentación

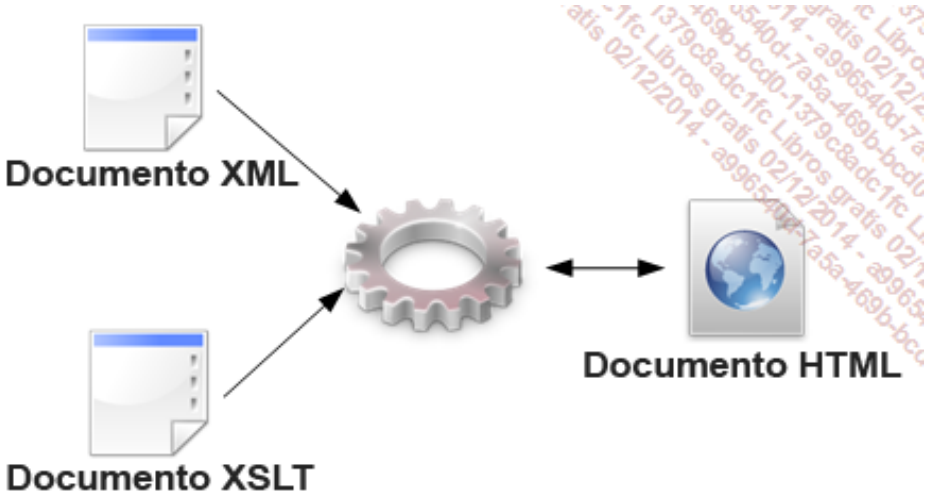
XML (*eXtended Markup Language*) deriva del lenguaje SGML (*Standard Generalized Markup Language*) desarrollado en los años 80. Se propuso una versión más simple de este lenguaje para la presentación de documentos Web el HTML (*HyperText Markup Language*). XML utiliza la simplicidad del HTML con la flexibilidad de SGML.

En un documento XML, la presentación y se separa completamente de los datos. La información (contenido) se separa de la apariencia (continente). Por lo tanto, se pueden ofrecer varios tipos de salidas para un mismo archivo de datos (imagen, archivo HTML, archivo XML, archivo PDF...).

XSL (*eXtensible Stylesheet Language*) permite gestionar los estilos y el formato de un documento XML, de igual forma que CSS lo hace para HTML.

Los documentos XML se utilizan para el intercambio de datos o la utilización de un estándar en la estructuración de la información. Los datos XML pueden ser, por lo tanto, manipulados por parte de los clientes y del servidor. Un documento XML puede transformarse en otro documento XML (cambio de nombre de las etiquetas, programación...) o en un documento XHTML, de texto, PDF u otros.

El siguiente esquema presenta la técnica que permite obtener un documento HTML/XHTML a partir de un documento XML y de una hoja de estilo XSL y el motor XSLT. Los Datos están completamente separados de la Presentación.



Técnica de transformación de XML-XSLT a HTML

Utilización

Utilizaremos varios ejemplos para efectuar transformaciones de XML a XML, de XML a flujo RSS y de XML a XHTML. Para ello, Struts proporciona un resultado de tipo XSLT con parámetros adicionales utilizados por la hoja de estilo XSLT.

- *stylesheetLocation*: ruta de la ubicación de la hoja de estilo.
- *excudingPattern*: lista de los elementos que se excluirá a partir de un modelo.
- *matchingPattern*: especificación de los modelos.
- *parse*: ruta de la hoja interpretada o no con una expresión OGNL.

Las hojas de estilo XSLT están presentes por defecto en el caché con Struts. Para cambiar esta configuración, es necesario modificar el parámetro *struts.xslt.nocache* en el archivo *struts.properties* o*default.properties*.

El proyecto *ejemplo30* permite crear un archivo XML a partir de los datos introducidos por el usuario. Este archivo XML toma forma a partir de los datos introducidos en un formulario HTML y de una hoja de estilo XSL. Cada acceso se realiza automáticamente a partir del nodo *result* y de los nodos hijos (por ejemplo, */result/cliente/identificador del cliente.getIdentificador()*).

```
Código: 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="true" />

    <package name="ejemplo30" namespace="/" extends="struts-
default">
        <default-action-ref name="Agregar_Cliente" />

        <action name="Agregar_Cliente">
            <result>/jsp/AgregarCliente.jsp</result>
        </action>

        <action name="XMLXSL" class="ejemplo30.ClienteAccion">
            <result name="success" type="xslt">
              <param name="stylesheetLocation">
               /xsl/Cliente.xsl
              </param>
            </result>
        </action>

    </package>
</struts>
```

```
Código: /jsp/AgregarCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Agregar un cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
```

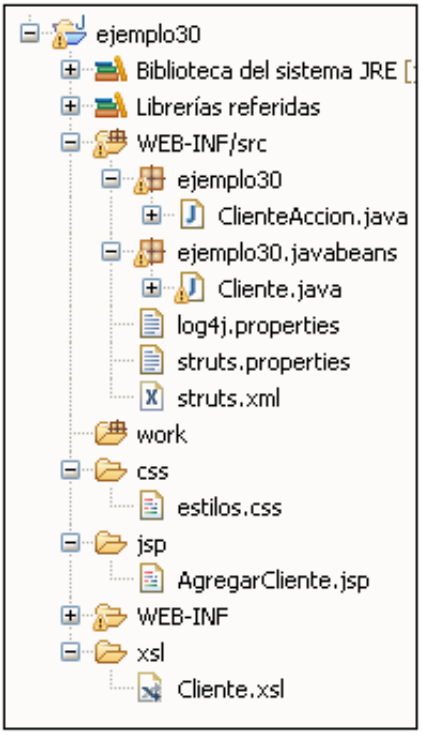
```
      <h3>Agregar un cliente</h3>
       <s:form method="post" action="XMLXSL"
id="Formulario_Cliente" name="Formulario_Cliente">
       <s:textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
       <s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
       <s:textfield name="cliente.fechadenacimiento"
id="cliente_fechadenacimiento" label="Fecha de nacimiento"
labelposition="top" cssClass="input"/>
       <s:textarea  name="cliente.descripcion" id="cliente_descripcion"
label="Descripción" labelposition="top" cssClass="textarea"/>
       <s:submit value="Confirmar" id="botonconfirmar"/>
       </s:form>
</div>
</body>
</html>
```

```
Código: ejemplo30.ClienteAccion.java
package ejemplo30;

import com.opensymphony.xwork2.ActionSupport;
import ejemplo30.javabeans.Cliente;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private Cliente cliente;

      public Cliente getCliente() {
             return cliente;
      }

      public void setCliente(Cliente cliente) {
             this.cliente = cliente;
      }

      // procesar la información
      public String execute() {
             return SUCCESS;
      }
}
```

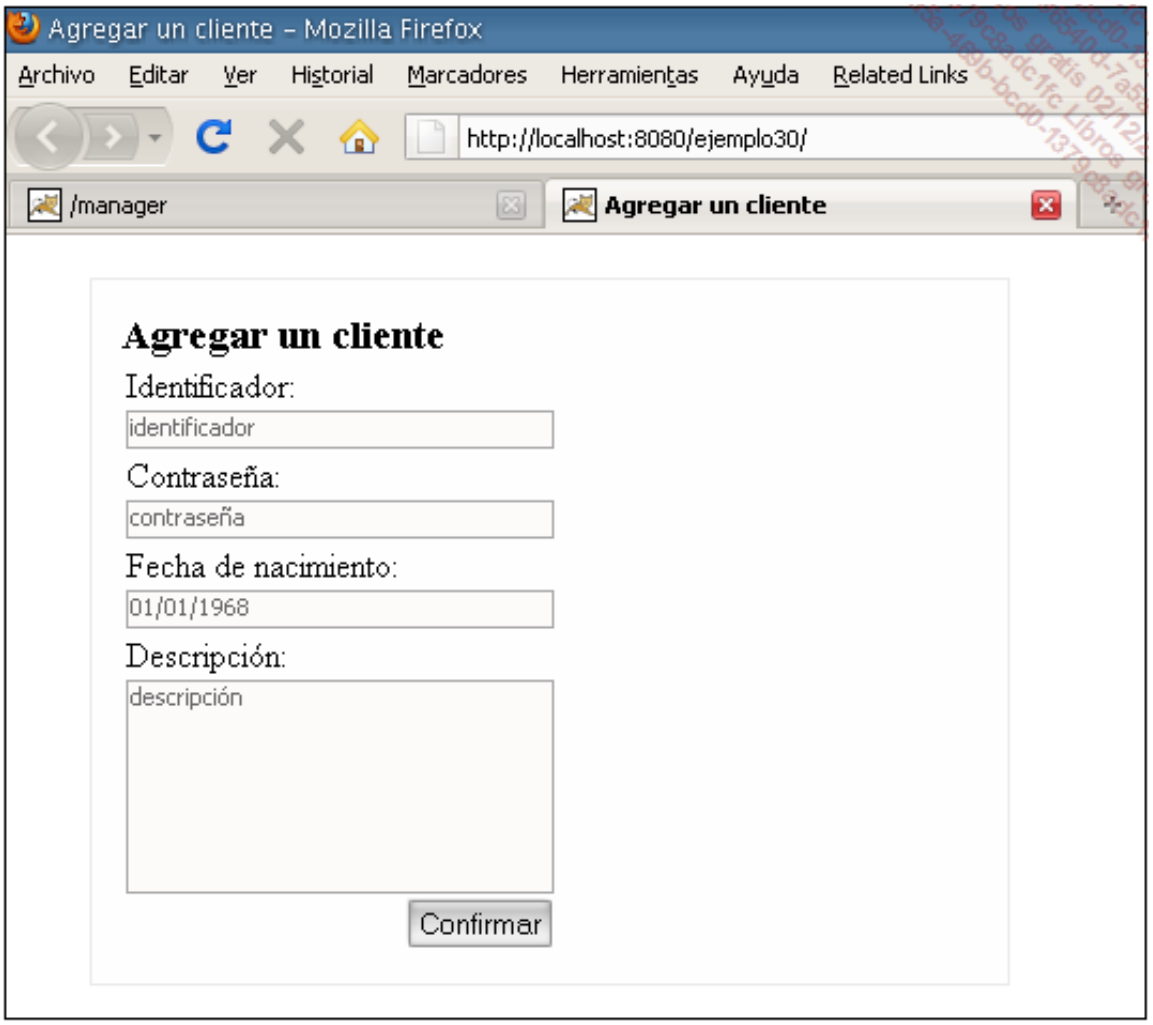
```
Código: /xsl/Cliente.xsl
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">

    <cliente>
        <identificador>
            <xsl:value-of select="/result/cliente/identificador"/>
        </identificador>
        <contrasena>
            <xsl:value-of select="/result/cliente/contrasena"/>
        </contrasena>
        <fechadenacimiento>
            <xsl:value-of select="/result/cliente/fechadenacimiento"/>
        </fechadenacimiento>
        <descripcion>
```

```
            <xsl:value-of select="/result/cliente/descripcion"/>
        </descripcion>
    </cliente>

    </xsl:template>

</xsl:stylesheet>
```

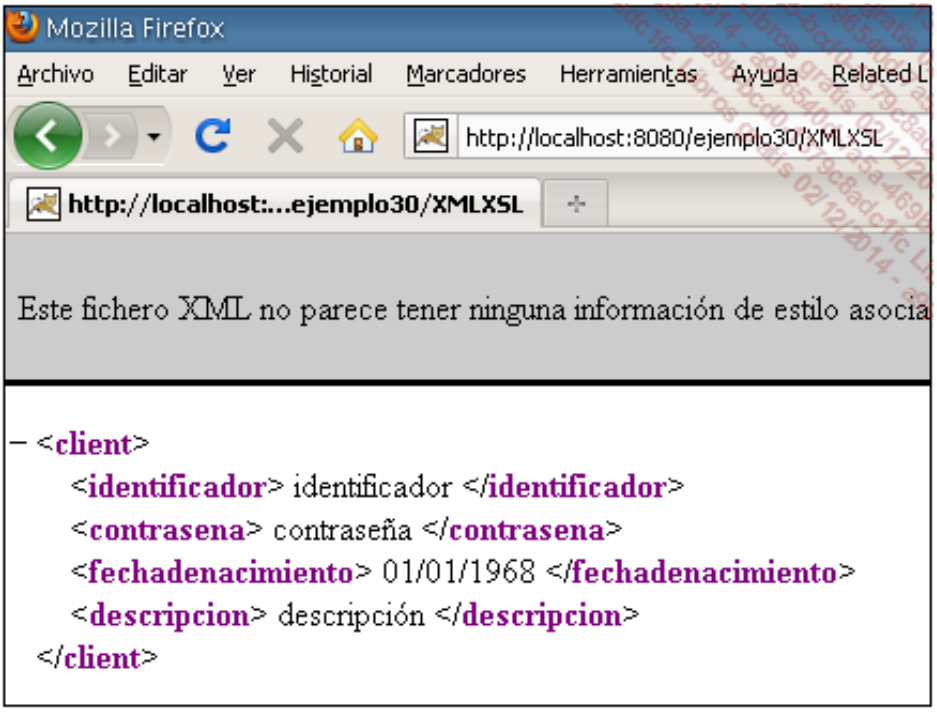
```
Código: struts.properties
struts.xslt.nocache=true
```



Árbol del proyecto ejemplo30



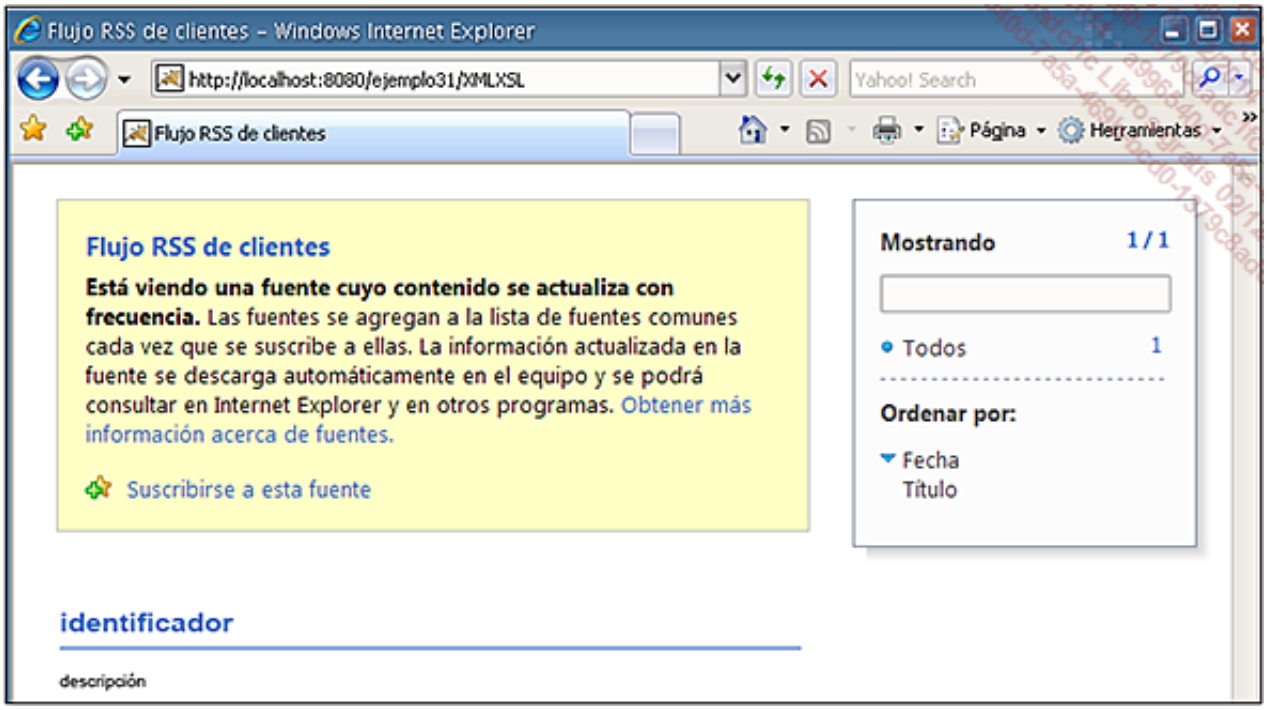
Formulario de registro de una cuenta de cliente



Visualización de la información del cliente en formato XML

El segundo proyecto *ejemplo31* permite mostrar el resultado en forma de un flujo RSS modificando únicamente la parte de la VISTA, es decir, la hoja de estilo XSL.

```
Código: /xsl/Cliente.xsl
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
    <rss version="2.0">
      <channel>
      <title>Flujo RSS de clientes</title>
      <link>http://localhost:8080/ejemplo31/</link>
      <description>Flujo RSS de clientes</description>
      <!-- cliente -->
      <item>
        <title>
          <xsl:value-of select="/result/cliente/identificador"/>
        </title>
        <description>
          <xsl:value-of select="/result/cliente/descripcion"/>
        </description>
        </item>
        </channel>
   </rss>
    </xsl:template>
</xsl:stylesheet>
```

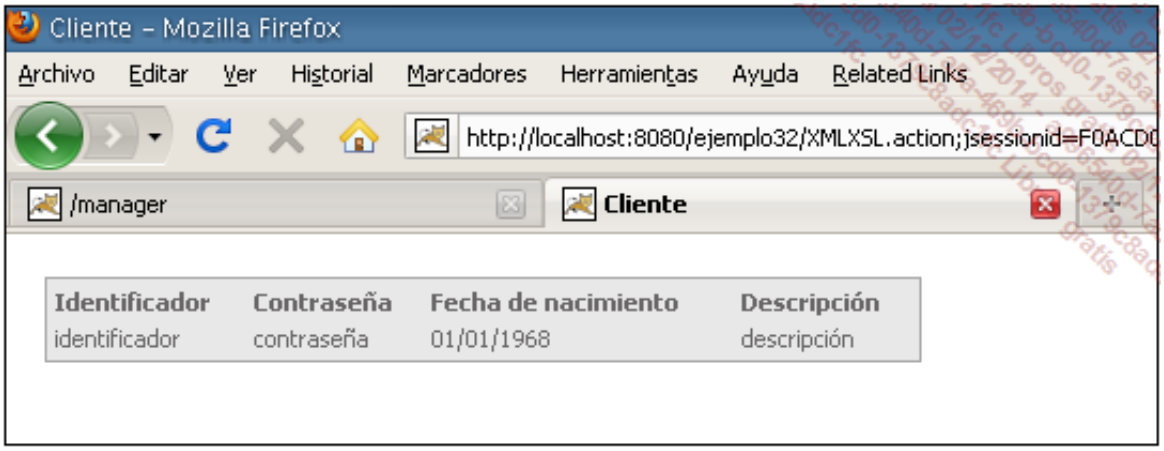


Visualización de la información del cliente en formato RSS

El último proyecto *ejemplo32* permite mostrar en formato XHTML la información introducida por el usuario con la ayuda de una hoja de estilo XSL. Observamos las ventajas de el procesamiento y la presentación.

```
Código: /xsl/Cliente.xsl
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes" encoding="ISO-8859-1"/>
    <xsl:template match="/">
    <html>
      <head>
            <title>Cliente </title>
```

```
            <!-- hojas de estilo -->
            <link rel="stylesheet" type="text/css"
href="css/estilos.css" title="predeterminado" />
      </head>
      <body>
          <table border="0" id="tabla" cellpadding="0"
cellspacing="4">
            <tr><td><b>Identificador</b></td><td><b>Contraseña</b></
td><td><b>Fecha de nacimiento</b></td><td><b>Descripción</b></td></tr>
          <tr>
            <td><xsl:value-of
select="/result/cliente/identificador"/></td>
            <td><xsl:value-of
select="/result/cliente/contrasena"/></td>
            <td><xsl:value-of
select="/result/cliente/fechadenacimiento"/></td>
            <td><xsl:value-of
select="/result/cliente/descripcion"/></td>
            </tr>
        </table>
       </body>
     </html>
     </xsl:template>
</xsl:stylesheet>
```



Visualización de la cuenta del cliente con una hoja XSLT

En resumen

Struts ofrece un medio simple y eficaz para gestionar los resultados de tipo XSLT. La respuesta se eenviarán en formato XML asociado con una hoja de estilo XSL. Esta arquitectura es la más utilizada para la gestión de salidas estándar en diferentes formatos como XML, RSS, XHTML o PDF sin tocar el código y manipulando únicamente la capa Vista del modelo MVC.

Presentación

Struts ofrece un sistema de complementos para aumentar las funcionalidades del framework. Un complemento (plug-in) de Struts es un archivo con formato *.jar* compuesto de clases de Java, de plantillas de Velocity o FreeMarker y de un archivo *struts-plugin.xml*.

Los complementos Struts se facilitan en forma de archivos *.jar* ubicados en el directorio */WEB-INF/lib*de la aplicación. En el archivo del complemento, un archivo *struts-plugin.xml* permite definir el empaquetado de sus resultados. Como recordatorio, Struts carga los archivos de gestión en este orden:

- El archivo *struts-default.xml* ubicado en la biblioteca *struts2-core-2.x.jar*.
- Todos los archivos llamados *struts-plugin.xml* implementados en la aplicación.
- El archivo principal de la aplicación, *struts.xml*.

Todos los complementos de Struts pueden utilizar y declarar nuevos paquetes, tipos de resultados, interceptores, acciones o bibliotecas de etiquetas. Existe un gran número de complementos más o menos útiles en Internet que se pueden utilizar con el framework.

El complemento JFreeChart

JFreeChart (http://www.jfree.org/jfreechart/) es una biblioteca Java OpenSource que permite crear gráficos evolucionados. La biblioteca, que se puede descargar desde el sitio del autor, se compone de los siguientes archivos que deben copiarse en el directorio */WEB-INF/lib* de la aplicación para su utilización:

- *jfreechart-x.jar*: la biblioteca gráfica de JFreeChart.
- *jcommon-x.jar*: la biblioteca dependiente para la creación de los gráficos.
- *struts2-jfreechart-plugin-2.x.jar*: el complemento JFreeChart para Struts.

La biblioteca struts2-jfreechart-plugin-2.x.jar es necesaria para la gestión de los gráficos. De hecho, esta biblioteca contiene el archivo struts-plugin.xml que definen el resultado de tipo chart. Además, la versión del complemento debe ser idéntica a la versión de Struts.

La creación de una aplicación de Struts JFreeChart se realiza siguiendo los siguientes puntos:

- Creación de un paquete de tipo *jfreechart-default*.
- Utilización de la clase *JFreeChart* para crear el gráfico.
- Definición de un resultado de tipo *chart* he inicialización de los parámetros *width* y *height* para el tamaño del gráfico.
- Definición de un atributo llamado *chart* en la clase de acción con su getter para devolver el objeto que se va a mostrar.

Los gráficos se devuelven en forma de imágenes en formato PNG o JPEG. La aplicación *ejemplo33*permite introducir la división del tiempo de trabajo del cliente informático (porcentaje de tiempo para realizar el análisis, el desarrollo, las pruebas y el mantenimiento) y mostrar una gráfica circular (pie) para representar la distribución.

```
Código 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="true" />

        <package name="ejemplo33" namespace="/"
extends="jfreechart- default">
        <default-action-ref name="Tiempo_Cliente" />

        <action name="Tiempo_Cliente">
            <result>/jsp/TiempoCliente.jsp</result>
        </action>

        <action name="Grafica" class="ejemplo33.ClienteAccion">
         <result name="success" type="chart">
                   <param name="value">chart</param>
                   <param name="type">png</param>
                   <param name="width">600</param>
                   <param name="height">400</param>
            </result>
         </action>
```

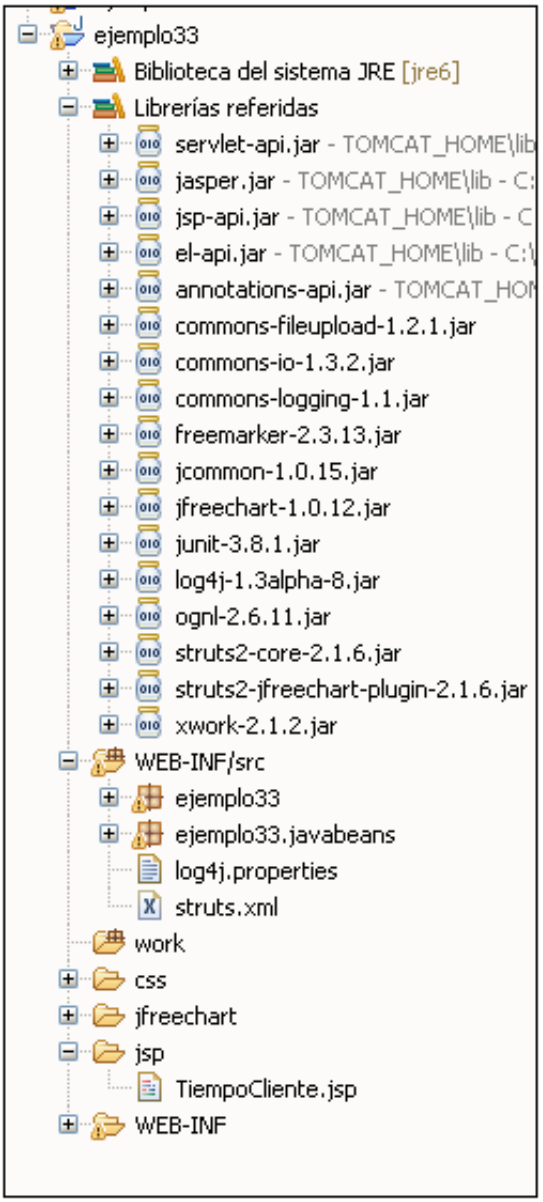
```
    </package>
</struts>
```

```
Código : ejemplo33.ClienteAccion.java
package ejemplo33;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PiePlot3D;
import org.jfree.data.general.DefaultPieDataset;
import org.jfree.util.Rotation;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo33.javabeans.Cliente;

public class ClienteAccion extends ActionSupport {

      // objeto para la gráfica
      private JFreeChart chart;
      // objeto cliente
      private Cliente cliente;

      public Cliente getCliente() {
             return cliente;
      }

      public void setCliente(Cliente cliente) {
             this.cliente = cliente;
      }

      // generar la gráfica a partir de la información
      public String execute() throws Exception {

      // crear la lista de datos que se mostrarán en la gráfica
      DefaultPieDataset datos=new DefaultPieDataset();
      datos.setValue("Analisis",
this.cliente.getPorcentajeAnalisis());
      datos.setValue("Desarrollo",
this.cliente.getPorcentajeDesarrollo());
      datos.setValue("Pruebas",
this.cliente.getPorcentajePruebas());
      datos.setValue("Mantenimiento",
this.cliente.getPorcentajeMantenimiento());

      // crear la gráfica
      chart = ChartFactory.createPieChart3D(
               "Distribución del tiempo de trabajo del cliente",  //
título
                datos,                // datos que se van a mostrar
                true,                 // mostrar leyenda
                true,
                true
            );

      // crear la gráfica circular
      PiePlot3D plot=(PiePlot3D) chart.getPlot();
      // ángulo de visualización
      plot.setStartAngle(190);
      // rotation
      plot.setDirection(Rotation.CLOCKWISE);
      // transparencia del esquema
      plot.setForegroundAlpha(0.4f);
      plot.setNoDataMessage("No hay datos para mostrar");
```

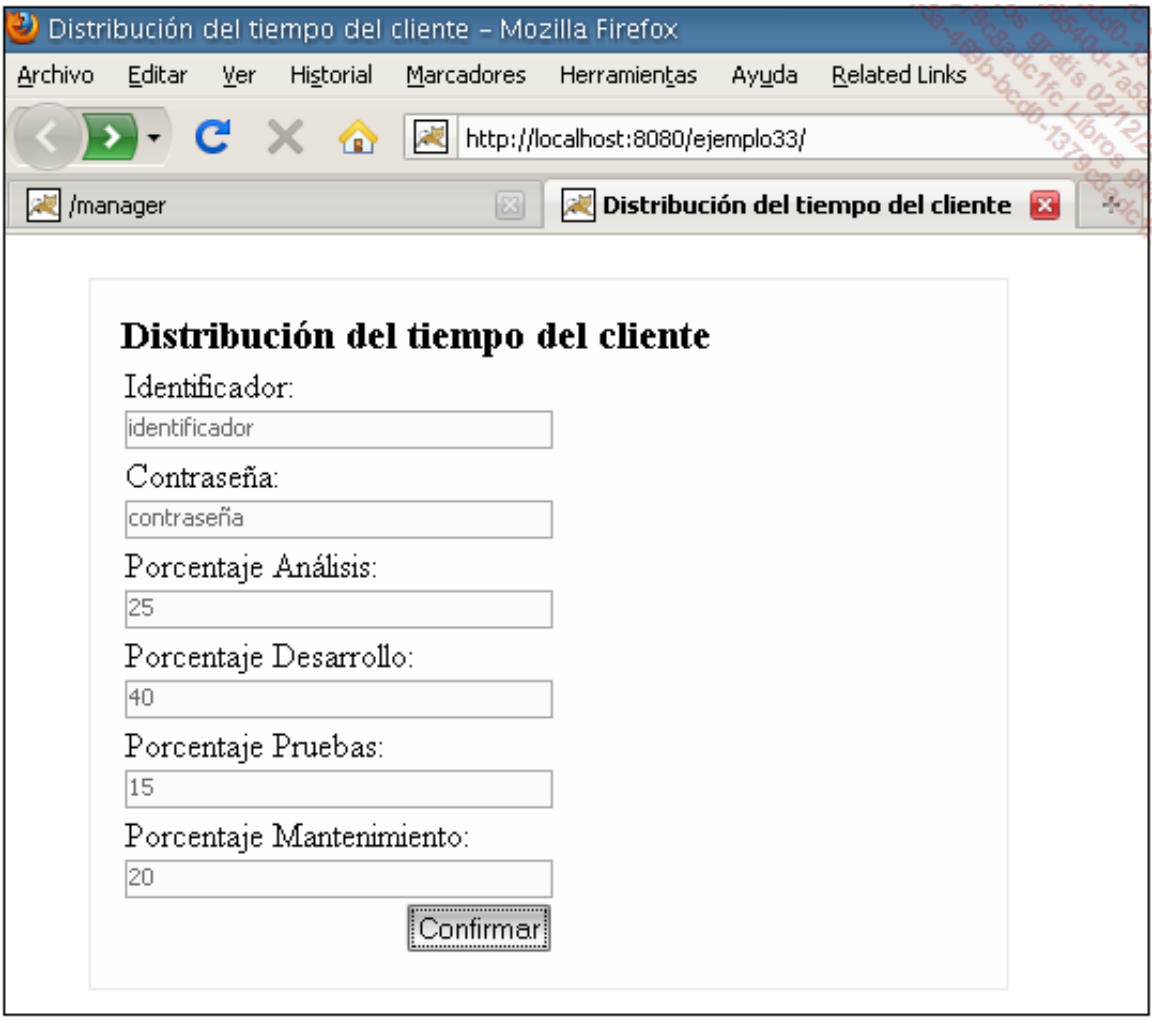
```
	return SUCCESS;
	}

	// getter para devolver la imagen
	public JFreeChart getChart() {
		return chart;
	}
}
```

Código: /jsp/TiempoCliente.jsp

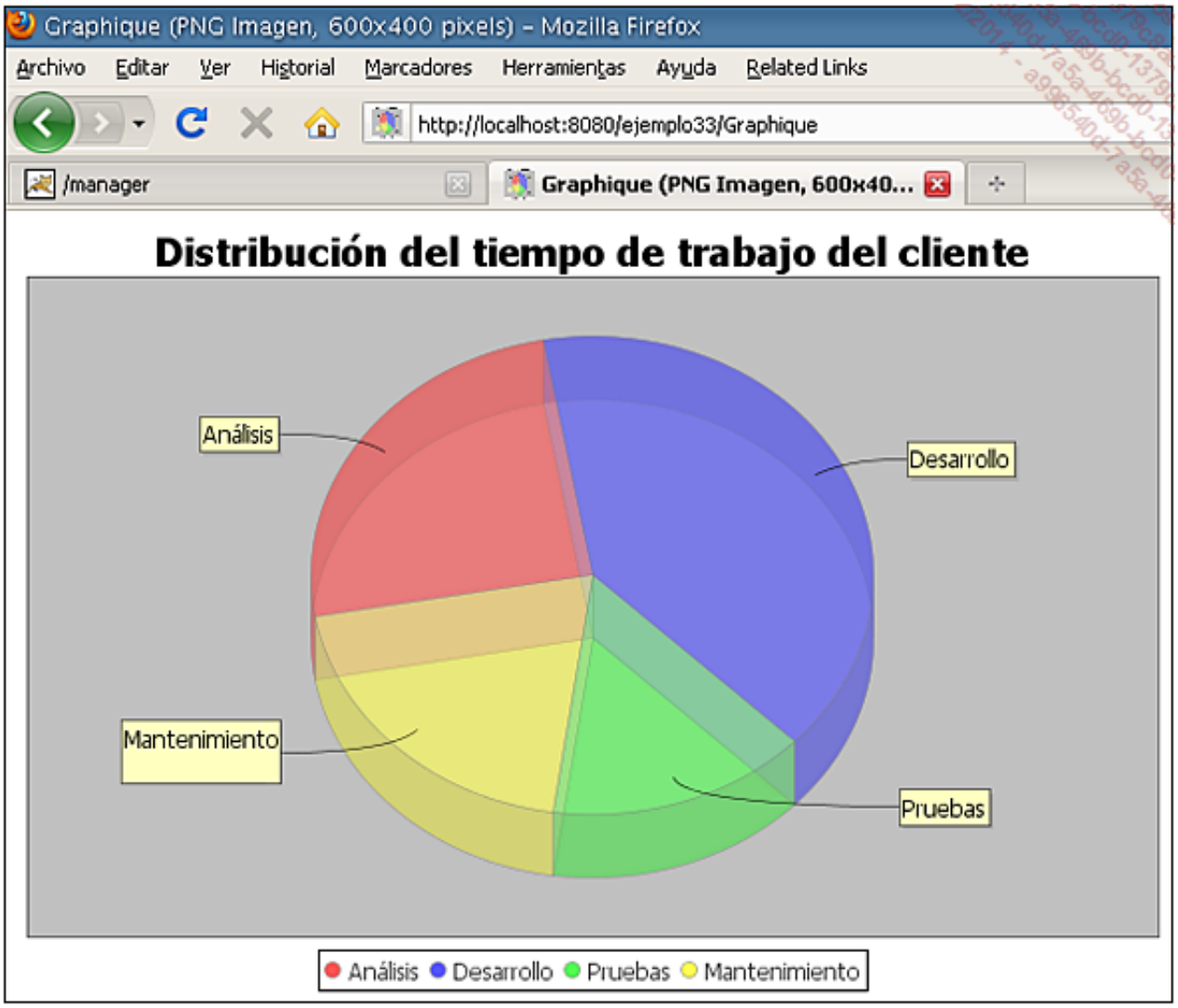
```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Distribuci&oacute;n del tiempo del cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<div id="carta">
	<h3>Distribuci&oacute;n del tiempo del cliente</h3>
	<s:form method="post" action="Grafica"
id="Formulario_Cliente" name="Formulario_Cliente">
	<s:textfield name="cliente.identificador"
id="cliente.identificador" label="Identificador" labelposition="top"
cssClass="input"/>
	<s:textfield name="cliente.contrasena"
id="cliente.contrasena" label="Contraseña" labelposition="top"
cssClass="input"/>
	<s:textfield name="cliente.porcentajeAnalisis"
id="cliente.porcentajeAnalisis" label="Porcentaje Analisis"
labelposition="top" cssClass="input"/>
	<s:textfield name="cliente.porcentajeDesarrollo"
id="cliente.porcentajeDesarrollo" label="Porcentaje
Desarrollo" labelposition="top" cssClass="input"/>
	<s:textfield name="cliente.porcentajePrueba"
id="cliente.porcentajePrueba" label="Porcentaje Pruebas"
labelposition="top" cssClass="input"/>
	<s:textfield name="cliente.porcentajeMantenimiento"
id="cliente.porcentajeMantenimiento" label="Porcentaje Mantenimiento"
labelposition="top" cssClass="input"/>
	<s:submit value="Confirmar" id="botonconfirmar"/>
	</s:form>
</div>
</body>
</html>
```



Árbol del proyecto ejemplo33



Formulario de introducción de valores del cliente



Visualización de la gráfica JFreeChart

También podemos insertar directamente una gráfica en una imagen de un documento XHTML con la ayuda del siguiente código: **. El nuevo *ejemplo33ajax* permite mostrar dinámicamente gráficas de forma aleatoria con la ayuda de la etiqueta XHML `<img/>` y de código JavaScript.

```
Código: /jsp/TiempoCliente.jsp
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Distribuci&oacute;n del tiempo del cliente</title>
<style type="text/css">@import url(css/estilos.css);</style>
<!--  archivo de gestión -->
<script src="javascript/ejemplo33ajax.js"
type="text/javascript"></script>
</head>
<body>
<div id="carta">

      <img src="" id="imagen"/>

      <script language="JavaScript">
      // acción efectuada después de la carga
      window.load=mostrarGrafica();
      </script>
</div>
</body>
```

```
</html>
```

Código: /javascript/ejemplo33ajax.js

```
// función que permite la visualización del gráfico en Ajax
function mostrarGrafica()
{
      // enlace de la acción con un parámetro aleatorio para
forzar la actualización de la imagen y evitar el caché
      var action="Grafica.action?"+new Date().getTime();

    // insertar la imagen en la etiqueta
    var im=new Image();
      im.src=action;
      document.images["imagen"].src=im.src;

      // cambiar la visualización cada 2 segundos
      setTimeout("mostrarGrafica()",2000);
}
```

El complemento Tiles

El complemento Tiles permite gestionar la apariencia y la división de las aplicaciones Web. Los proyectos de Internet normalmente utilizan divisiones en forma de encabezados, menú, contenido y pie de página. Para realizar esta división, los desarrolladores utilizan tablas HTML o aún mejor, definiciones en las hojas de estilo CSS y etiquetas `<span/>`, `<div/>` u otras. El proyecto se compone de estas páginas (fragmentos) llamadas, por ejemplo *encabezado.jspf*, *menu.jspf* y *piedepagina.jspf*. A continuación, cada vista del sitio utiliza inclusiones dinámicas a partir de la directiva JSP *<%@ include file="..." %>* o del tag *<jsp:include/>*.

El proyecto *ejemplo34* pasado en la aplicación *ejemplo08* utiliza una división del sitio a partir de fragmentos de páginas JSP y de la etiqueta `<%@ include file=''...'' />`.

Código: /jspf/encabezado.jspf

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Gestión de los clientes</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<!-- Mensaje de error -->
<s:if test="errors.size()>0">
<div id="mensaje_error">
 <label>Se produjeron los siguientes errores: </label>
 <ul><s:fielderror/></ul>
</div>
</s:if>
<div id="encabezado">Gestión de clientes</div>
<div id="menu">
MEN&Uacute;<br/>
<a href="/ejemplo34">Mostrar el formulario</a>
</div>
<div id="carta">
```

Código: /jsp/AgregarCliente.jsp

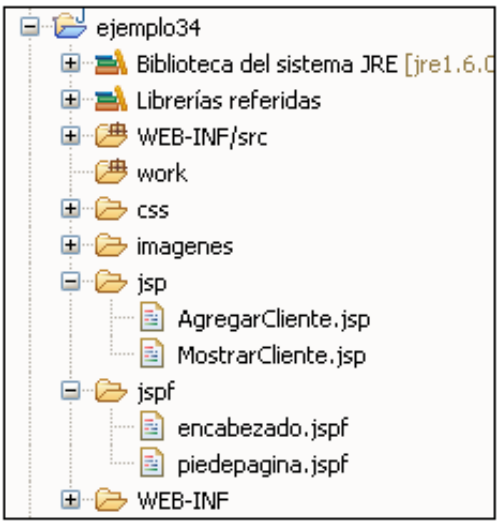
```
<%@ include file="../jspf/encabezado.jspf" %>
<h3>Agregar un cliente</h3>
<s:form method="post" action="ConfirmarAgregar_Cliente">
 <s:textfield name="identificador" id="identifiant"
label="Identificador" labelposition="top" cssClass="input"/>
 <s:textfield name="contrasena" id="contrasena"
label="Contraseña" labelposition="top" cssClass="input"/>
 <s:submit value="Agregar un cliente"/>
</s:form>
<%@ include file="../jspf/piedepagina.jspf" %>
```

Código: /jsp/MostrarCliente.jsp

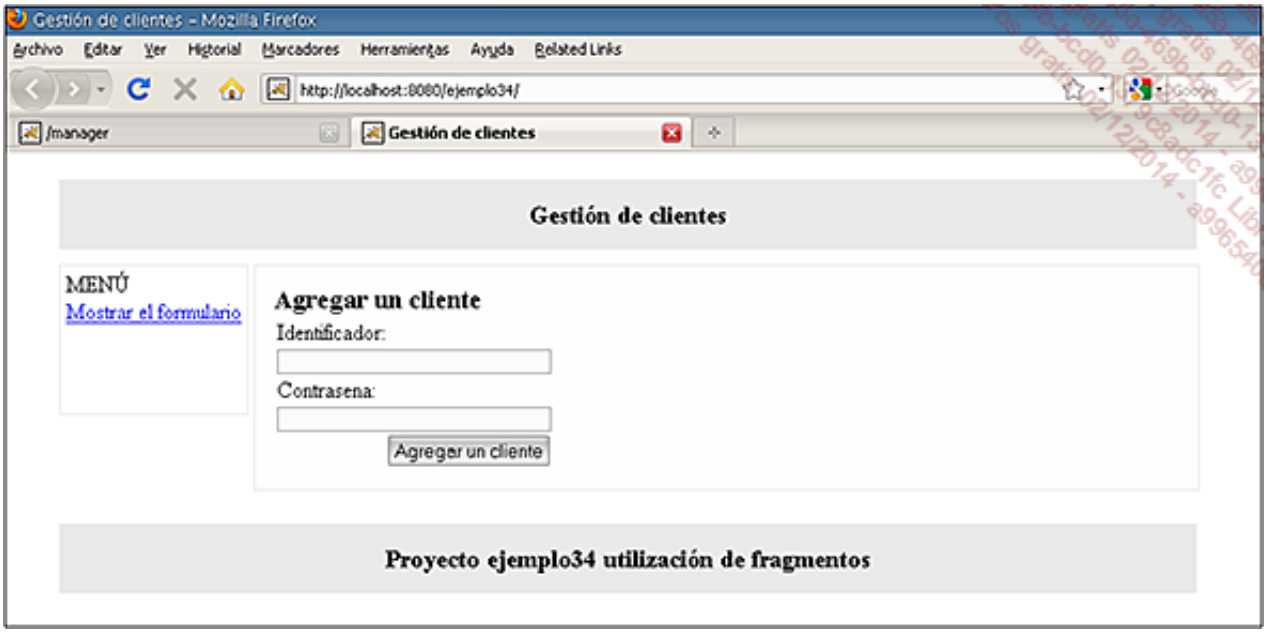
```
<%@ include file="../jspf/encabezado.jspf" %>
<p>
<h4><s:property value="%{getText('cliente.mostrar')}"/></h4>
<s:property value="%{getText('cliente.identificador')}"/>:
<s:property value="identificador"/> <br/>
<s:property value="%{getText('cliente.contrasena')}"/>: <s:property
value="contrasena"/><br/>
</p>
<%@ include file="../jspf/piedepagina.jspf" %>
```

Código: /jspf/piedepagina.jspf

```
</div>
<div id="piedepagina">Proyecto ejemplo34 utilizaci&oacute;n de
fragmentos</div>
</body>
</html>
```



Árbol del proyecto ejemplo34



Visualización del formulario del cliente con la ayuda de fragmentos

Esta técnica, muy utilizada en la mayoría de las aplicaciones Web, también es muy limitada para proyectos de envergadura. De hecho, si la presentación cambia (el nombre de los fragmentos) o la presentación, debemos modificar cada página del usuario. Para evitarlo, el complemento Tiles proporciona una biblioteca de etiquetas que permite definir la presentación para todas las páginas de usuario del sitio. Un cambio en la definición de las páginas conlleva una modificación instantánea de todas las páginas de usuario. Para ello, Tiles se basa en una definición de páginas en formato JSP o XML, que permite determinar la estructura del sitio.

Tiles es un componente desarrollado en su origen por Struts 1. Sin embargo, gracias a su popularidad, Tiles pasó a ser un proyecto de Apache por completo http://tiles.apache.org. El complemento de Struts, Tiles se incluye con las bibliotecas de Struts struts-2.1.X-lib.zip (tiles-core-2.X.jar, tiles-api-2.X.jar, tiles-jsp-2.x.jar y struts2-tiles-plugin-2.1.X.jar). Para que el complemento Tiles funcione correctamente con Struts 2, deberán instalarse también las siguientes bibliotecas dependientes: commons-beanutils-1.X.jar, commons-collections-3.X.jar y commons-digester-1.X.jar.

El complemento Tiles se basa en dos componentes: la plantilla y la definición. La plantilla de las páginas se define a partir de una página JSP. Cada página JSP utilizada por la plantilla de división será declarada en la página. Ahora, si queremos cambiar toda la plantilla de presentación de la aplicación, basta con cambiar una sola página JSP de definición para aplicar los cambios. Para aplicar este complemento, vamos a definir un nuevo proyecto con el nombre *ejemplo35* a partir del anterior.

1. Página del formato de la plantilla

La siguiente página JSP permite definir la plantilla de presentación del sitio. Al principio del archivo encontramos la inclusión de la biblioteca de etiquetas y la utilización de dos etiquetas:

- `<tiles:getAsString name="titulopagina"/>`: permite devolver una cadena de caracteres definida en el archivo de configuración.
- `<tiles:insertAttribute name="encabezado"/>`: permite incluir una página en la definición de la plantilla de presentación.

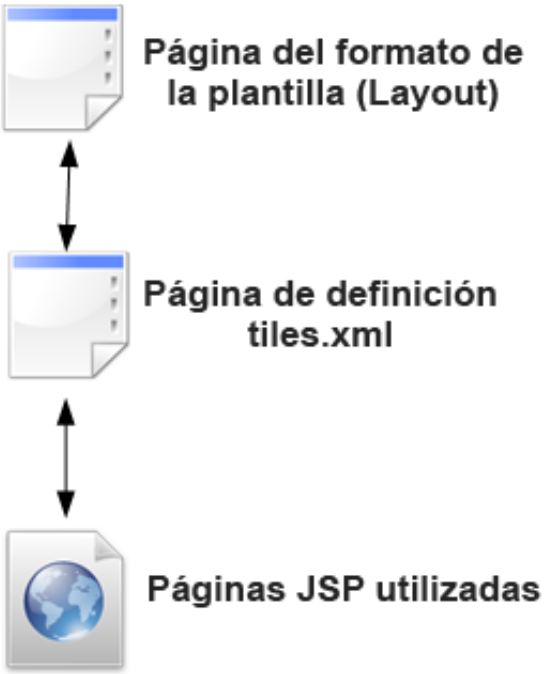
La etiqueta `<tiles:/>` posee el atributo role, que permite especificar la función del usuario necesaria para la ejecución de la página.

Código: /jsp/PlantillaPresentacion.jsp

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles"
prefix="tiles"%>
<html>
<head>
<title><tiles:getAsString name="tituloPagina"/></title>
</head>
<body>
<tiles:insertAttribute name="encabezado"/>
<tiles:insertAttribute name="contenido"/>
<tiles:insertAttribute name="piedepagina"/>
</body>
</html>
```

2. Definición de la plantilla

Se utiliza una definición de plantilla entre la página de presentación de la plantilla y las páginas JSP del usuario. Por analogía con Java, la página de presentación de la plantilla (layout) se compara a menudo con una interfaz y la definición de la plantilla con la clase de usuario de la interfaz, que proporciona una implementacion de ésta.



Arquitectura de Tiles

La página de definición del plantilla se establece en un archivo *tiles.xml* ubicado en el directorio*/WEB-INF* de la aplicación Struts. El archivo *tiles.xml* del projecto *ejemplo35* se presenta a continuación con la ayuda de la etiqueta `<definition/>` y de los atributos `name` y `template`.

```
Código: /WEB-INF/tiles.xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
 "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
 "http://struts.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
<definition name="AgregarCliente"
template="/jsp/PlantillaPresentacion.jsp">
<put-attribute name="titulopagina" value="Registro de un cliente"/>
<put-attribute name="encabezado" value="/jspf/encabezado.jspf"/>
<put-attribute name="contenido" value="/jsp/AgregarCliente.jsp"/>
<put-attribute name="piedepagina" value="/jspf/piedepagina.jspf"/>
</definition>

<definition name="MostrarCliente"
template="/jsp/PlantillaPresentacion.jsp">
<put-attribute name="titulopagina" value="Visualización del cliente"/>
<put-attribute name="encabezado" value="/jspf/encabezado.jspf"/>
<put-attribute name="contenido" value="/jsp/MostrarCliente.jsp"/>
<put-attribute name="piedepagina" value="/jspf/piedepagina.jspf"/>
</definition>
</tiles-definitions>
```

La etiqueta `<definition/>` contiene una o varias etiquetas `<put-attribute/>` utilizadas en referencia a la página de presentación de la plantilla. Para nuestro proyecto, la página de presentación utilizada para la plantilla es */jsp/PlantillaPresentacion.jsp*.

La definición *AgregarCliente* permite declarar una variable llamada *titulopagina* asociada a la presentación de la plantilla y tres páginas para insertar respectivamente el encabezado, el contenido y el pie de página.

Posteriormente, en el archivo de configuración de la aplicación *struts.xml*, se asociaron los

resultados de tipo *tiles* con las diferentes definiciones del archivo *tiles.xml* (*AgregarCliente* y*MostrarCliente*).

3. Aplicación del complemento Tiles

La aplicación del complemento Tiles se realiza en cuatro pasos:

- Copia de los archivos *tiles-core-2.X.jar*, *tiles-api-2.X.jar*, *tiles-jsp-2.x.jar* y *struts2-tiles-plugin-2.1.X.jar* en el directorio */WEB-INF/lib* de la aplicación.
- Definición del listener asociado a Tiles en el archivo de configuración */WEB-INF/web.xml*.

```
       <listener>
        <listener-
class>org.apache.struts2.tiles.StrutsTilesListener</listener-
class>
       </listener>
```

- Definición de los resultados de tipo *tiles* en el archivo de configuración *struts.xml* o utilización del paquete *tiles-default*.

```
       <result-types>
        <result-type name="tiles"
class="org.apache.struts2.views.tiles.TilesResult"/>
       </result-types>
```

- Utilización de los resultados de tipos *tiles* en los resultados de las acciones de Struts.

El proyecto *ejemplo35* está terminando, el archivo *struts.xml* contiene la definición de las acciones y de los resultados de tipo *tiles*. Estos últimos están relacionados con el archivo de definición de la plantilla *PlantillaPresentacion.jsp* y las definiciones del archivo *tiles.xml*.

Los resultados definidos en el archivo *struts.xml* se envían a la plantilla de presentación*PlantillaPresentacion.jsp* y utilizar los datos presentados en las definiciones *AgregarCliente* y*MostrarCliente* del archivo *tiles.xml*.

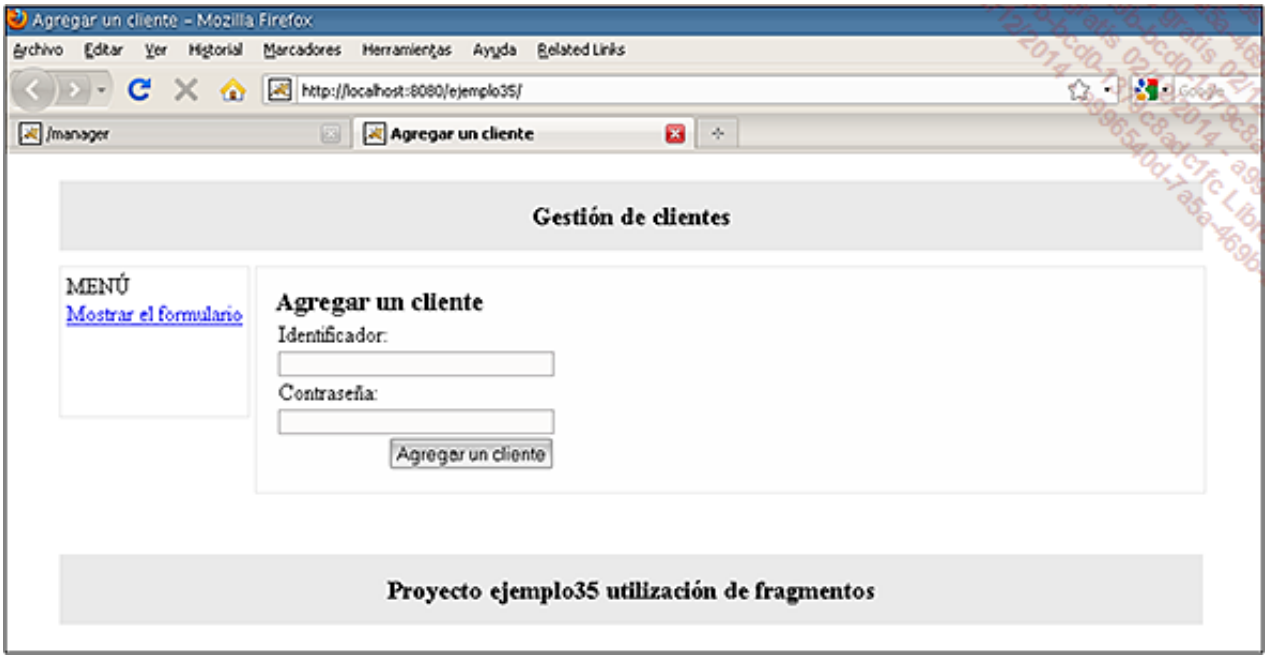
```
Código: /WEB-INF/web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="ejemplo35" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
<listener>
<listener-
class>org.apache.struts2.tiles.StrutsTilesListener</listener-
class>
</listener>
<filter>
<filter-name>struts2</filter-name>
<filter-
class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExec
uteFilter</filter-class>
</filter>
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

Código: 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="true" />
<package name="ejemplo35" namespace="/" extends="struts-default">
<result-types>
<result-type name="tiles"
class="org.apache.struts2.views.tiles.TilesResult"/>
</result-types>
<default-action-ref name="Agregar_Cliente" />
<action name="Agregar_Cliente" class="ejemplo35.ClienteAccion">
<result type="tiles">AgregarCliente</result>
</action>
<action name="ConfirmarAgregar_Cliente"
class="ejemplo35.ClienteAccion" method="agregar">
<result name="input" type="tiles">AgregarCliente</result>
<result name="success" type="tiles">MostrarCliente</result>
</action>
</package>
</struts>
```

Código: /jsp/PlantillaPresentacion.jsp

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles"
prefix="tiles"%>
<html>
<head>
<title><tiles:getAsString name="titulopagina"/></title>
</head>
<body>
<tiles:insertAttribute name="encabezado"/>
<tiles:insertAttribute name="contenido"/>
<tiles:insertAttribute name="piedepagina"/>
</body>
</html>
```

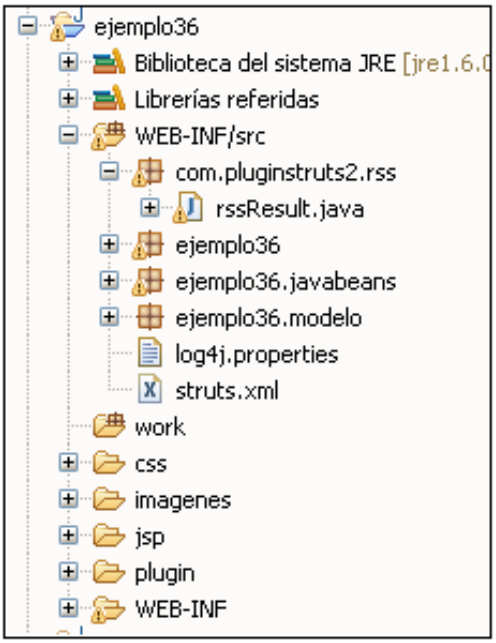


Utilización del complemento Tiles para la arquitectura de las páginas

Escribir un complemento

La escritura de un complemento no es una tarea muy compleja con el framework Struts. El principio es el mismo que en la creación de archivos en formato *.jar*. Vamos a escribir un complemento que permita generar directamente una colección de objetos en forma de un flujo RSS (lo retomamos del proyecto*ejemplo23*).

El árbol de la aplicación utiliza una escritura de paquete completamente validada para poder ser exportada:



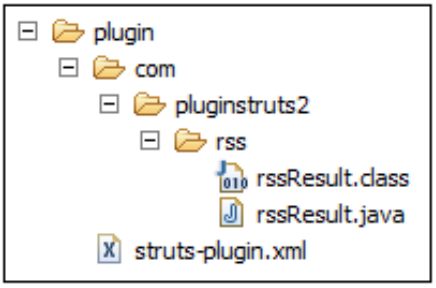
Árbol del proyecto ejemplo36

El paquete *com.pluginstruts2.rss* contiene nuestra clase `rssResult.java` que permite generar una colección de objetos en forma de flujo RSS. Esta clase utiliza dos parámetros, *coleccionObjeto* para especificar el nombre de la colección que se va a mostrar y *enlace* para realizar los enlaces de cada elemento del flujo rss.

```
Código : /com/pluginstruts2/rss/rssResult.java
package com.pluginstruts2.rss;

import java.io.PrintWriter;
import java.util.List;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts2.StrutsStatics;
import org.apache.struts2.dispatcher.StrutsResultSupport;
import com.opensymphony.xwork2.ActionInvocation;

@SuppressWarnings("serial")
public class rssResult extends StrutsResultSupport{

	// recuperar el nombre del objeto
	private String coleccionObjeto;
	// enlace para el archivo rss
	private String enlace;

	public String getColeccionObjeto() {
		return coleccionObjeto;
	}

	public void setColeccionObjeto(String coleccionObjeto) {
		this.coleccionObjeto = coleccionObjeto;
```

```
	}

	public String getEnlace() {
		return enlace;
	}

	public void setEnlace(String enlace) {
		this.enlace = enlace;
	}

	// método ejecutado por el resultado
	public void doExecute(String finalLocation,ActionInvocation
invocation) throws Exception
	{
		// recuperar el objeto response
		HttpServletResponse
response=(HttpServletResponse)invocation.getInvocationContext().get
(StrutsStatics.HTTP_RESPONSE);

		// recuperar la lista de clientes de la pila
de ejecución
		List<Object>
listaObjects=(List<Object>)invocation.getStack().findValue(this.getColeccionO-
bjeto());

		// tipo de respuesta
		response.setContentType("application/xml");
		PrintWriter out=response.getWriter();
		out.println("<?xml version=\"1.0\" encoding=\"ISO-
8859-1\"?>");
		out.println("<rss version=\"2.0\">");
		// crear un canal
		out.println("<channel>");
		out.println("<title>Flujo RSS de los objetos</title>");
		out.println("<link>"+this.getEnlace()+"</link>");
		out.println("<description>Flujo RSS de
objetos</description>");
		// crear los objetos
		for(int i=0;i<listaObjects.size();i++)
		{
			Object o=(Object)listaObjects.get(i);
			out.println("<item>");
				out.println("<title>"+o+"</title>");
				out.println("<link>"+this.getEnlace()
+"</link>");

	out.println("<description>"+o+"</description>");
			out.println("</item>");
		}
		out.flush();
		out.close();
	}
}
```

La etapa de generación del archivo (*.jar*) consiste en crear un directorio independiente, que conserve el árbol de paquetes, para colocar el archivo completado *.class* y crear el archivo *struts-plugin.xml* en la raíz del paquete.



Árbol de un complemento de Struts

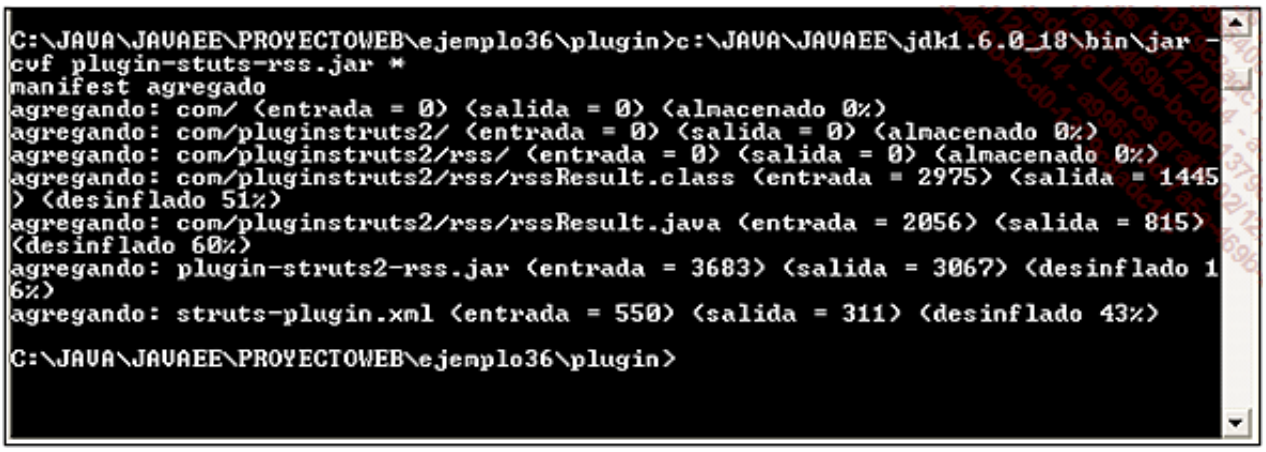
El archivo *struts-plugin.xml* comienza por la definición del nombre del paquete que se utilizará más adelante en el resto de paquetes. Se especifica un resultado de tipo *rss* con la asociación de la clase que se va a ejecutar para este tipo. Por último, se inicializan dos parámetros por defecto para el nombre de la colección y el enlace de cada elemento del flujo rss.

```
Código: struts-plugin.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="rss-default" extends="struts-default">
     <result-types>
                <result-type name="rss"
class="com.pluginstruts2.rss.rssResult" default="false">
                        <param
name="coleccionObjeto">listaObjetos</param>
                        <param name="enlace">http://localhost</param>
                </result-type>
     </result-types>
     </package>
</struts>
```

Hemos terminado con la estructura, ahora podemos pasar a la creación del archivo con la herramienta*jar* incluida en el jdk de Java. Para ello, es necesario situarse en el directorio del complemento y ejecutar el siguiente comando:

```
jar -cvf plugin-struts2-rss.jar *
```

Se creará un nuevo paquete con el nombre *plugin-struts2-rss.jar* en la raíz del árbol del complemento.



Generar un archivo para el complemento de Struts

Utilizar el complemento

Ya sea creado el complemento, ahora podemos pasar a su utilización copiando la biblioteca *plugin-struts2-rss.jar* en el directorio */WEB-INF/lib* de una aplicación. El complemento se utiliza en el archivo de configuración *struts.xml* para declarar un paquete que lo utiliza. El archivo de configuración comienza por la definición del paquete que utiliza el complemento (`extends="rss-default"`) y la definición del resultado de tipo *rss* con los enlaces que se realizarán en el flujo RSS y al nombre de la colección de objetos que se debe mostrar.

```
Código : 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="ejemplo36" namespace="/" extends="rss-default">

        <default-action-ref name="ListadoRSS_Cliente" />

        <action name="ListadoRSS_Cliente"
class="ejemplo36.ClienteAccion" method="listado">
            <result type="rss">
              <param name="enlace">http://www.google.es</param>
              <param name="coleccionObjeto">listaClientes</param>
            </result>
        </action>

    </package>
</struts>
```

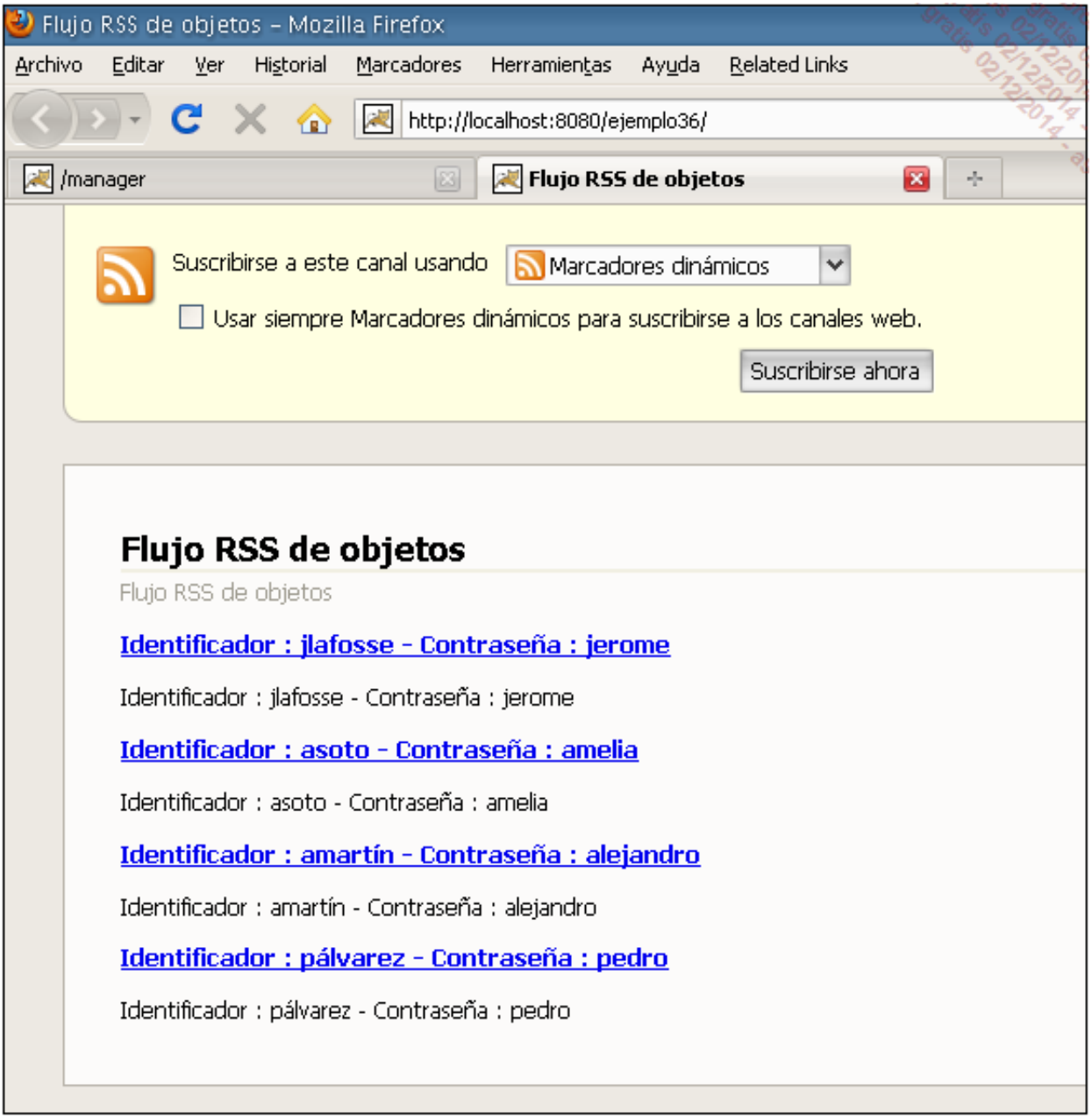
Por último, la clase de acción `ClienteAccion.action` utiliza la colección estática con el nombre*listaClientes* y el complemento *rss*. Ahora la aplicación es totalmente funcional.

```
Código: ejemplo36.ClienteAccion.java
package ejemplo36;

import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo36.javabeans.Cliente;
import ejemplo36.modelo.ClienteModelo;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

    private Cliente cliente;
    private List<Cliente> listaClientes;


      public Object getModelo() {
             return cliente;
      }

      public Cliente getCliente() {
             return cliente;
      }

      public void setCliente(Cliente cliente) {
```

```
		this.cliente = cliente;
	}

	public List<Cliente> getListaClientes() {
		return listaClientes;
	}

	public void setListaClientes(List<Cliente> listaClientes) {
		this.listaClientes = listaClientes;
	}

	// devolver la lista de clientes después que la recuperación
	public String listado()
	{
		listaClientes=ClienteModelo.getListaClientes();
	  return SUCCESS;
    }
}
```

Código: ejemplo36.javabeans.Cliente.java

```
package ejemplo36.javabeans;

@SuppressWarnings("serial")
public class Cliente {

	private int idCliente;
	private String identificador;
	private String contrasena;

	public Cliente() {

	}

	public Cliente(int idCliente,String identificador, String
contrasena){
		this.idCliente=idCliente;
		this.identificador=identificador;
		this.contrasena=contrasena;
	}

	// getter y setter

	public String toString() {
		String res="Identificador: "+this.getIdentificador()+"
- Contraseña: "+this.getContrasena();
		return res;
	}
}
```


Flujo RSS de objetos – Mozilla Firefox
Archivo Editar Ver Historial Marcadores Herramientas Ayuda Related Links
http://localhost:8080/ejemplo36/
/manager
Flujo RSS de objetos
Suscribirse a este canal usando
Marcadores dinámicos
Usar siempre Marcadores dinámicos para suscribirse a los canales web.
Suscribirse ahora
Flujo RSS de objetos
Flujo RSS de objetos
Identificador : jlafosse - Contraseña : jerome
Identificador : jlafosse - Contraseña : jerome
Identificador : asoto - Contraseña : amelia
Identificador : asoto - Contraseña : amelia
Identificador : amartín - Contraseña : alejandro
Identificador : amartín - Contraseña : alejandro
Identificador : pálvarez - Contraseña : pedro
Identificador : pálvarez - Contraseña : pedro

Otros complementos

El sitio http://cwiki.apache.org/S2PLUGINS/home.html ofrece una colección de distintos complementos disponibles para el framework Struts. Podemos encontrar complementos para Hibernate, Ajax, JavaScript, imágenes o incluso la facilitación del uso de tablas.

La comunidad de complementos de Struts se encuentra actualmente en fase de desarrollo y prácticamente cada día aparecen nuevos complementos, más o menos útiles y funcionales, disponibles en línea. Un desarrollador verá como se agiliza su trabajo con la ayuda de estos complementos.

En resumen

El framework Struts ofrece una forma simple y eficaz de crear complementos. En este capítulo se han presentado el mecanismo de complemento de Struts, así como la utilización a partir de archivos y del archivo de configuración *struts.xml*. El complemento JFreeChart permite crear gráficas complejas con la ayuda de un resultado de tipo *chart*. El complemento Tiles se utiliza para la arquitectura de las vistas y con el fin de facilitar el mantenimiento del conjunto. Por último, los dos últimos párrafos explican el desarrollo de complementos personalizados y la utilización a partir de archivos fuente y del archivo en formato *.jar*.

Presentación

Como hemos visto desde el comienzo de este libro, la configuración de las acciones, validaciones o resultados es una tarea simple con Struts, se realiza a partir de archivos XML. El framework ofrece, sin embargo, un segundo enfoque llamado configuración cero o zero configuration. En lugar de utilizar el archivo *struts.xml* para especificar las clases y el enrutamiento, las clases se anotan.

Configuración

Si deseamos utilizar la configuración cero, es decir sin archivo XML, debemos indicar a Struts que debe utilizar un paquete o una lista de paquetes como clases de acción. La primera versión de Struts 2 ofrecía la posibilidad de utilizar el sistema *Zero Config*, que permitía especificar el paquete que se anotaría en el archivo */WEB-INF/web.xml*. Esta técnica se ha mejorado en beneficio del complemento*CodeBehind*, que permite la simplificación de las declaraciones de paquetes que se anotarán y el funcionamiento global. Desde la versión 2.1 de Struts, estas 2 técnicas han pasado a ser obsoletas en beneficio del complemento *Convention*.

El complemento *Convention* (*struts2-convention-plugin-2.x.jar*) está incluido en el archivo *struts2.X-lib.rar*, que se puede descargar desde el sitio oficial de Struts 2. Este archivo contiene las bibliotecas estándar, así como los complementos que pueden utilizarse con la versión indicada del framework. Este complemento permite realizar anotaciones de acciones, de interceptores, de validaciones o de tipo de conversiones. A continuación se resumen las principales funcionalidad es de este complemento:

- Notación de acciones.
- Convención de nombramiento para los resultados.
- Convención de nombramiento para los accesos a los nombres de clases por URL.
- Convención de nombramiento para los paquetes.
- Notación de interceptores.
- Notación de espacios de nombres.
- Notación de paquetes XWork.

Para utilizar el complemento *Convention*, es necesario copiar la biblioteca *struts2-convention-plugin-2.x.jar* en el directorio */WEB-INF/lib* de la aplicación. El archivo *struts.xml* de configuración de aplicaciones ya no es obligatorio, la aplicación utiliza automáticamente el complemento *Convention* si este está instalado y está presente en la *classpath*.

En el caso de una distribución de la aplicación en forma de paquete con la configuración cero, es necesario establecer el parámetro struts.convention.action.disableJarScanning en true.

Utilización

Por defecto, el complemento *Convention* utiliza resultados automáticos presentes en el directorio*/WEB-INF/content* de la aplicación. Esta configuración se puede modificar con la propiedad*struts.convention.result.path* presente en el archivo de propiedades de Struts. El mapping de las URL se realiza con la ayuda de los nombres. Así, la acción que se ejecuta con la URL http://localhost:8080/ejemplo37/cliente se asociará automáticamente a la vista JSP */WEB-INF/content/cliente.jsp*.

1. Nomenclatura

Para aplicar la configuración cero, utilizaremos un nuevo proyecto llamado *ejemplo37* a partir de la aplicación *ejemplo14*. Por defecto, el complemento *Convention* busca todas las clases que heredan de la clase `com.opensymphony.xwork2.Action` o que tienen el sufijo *Action* en paquetes específicos.

Los paquetes utilizados por el complemento *Convention* deben respetar una convención de nomenclatura y se les llama *struts*, *struts2*, *action* o *actions*.

Cada paquete que contenga uno de estos nombres será considerado como un paquete del complemento *Convention*. A continuación, el complemento examine si las clases de estos sub-paquetes heredan de la clase `com.opensymphony.xwork2.Action` o si tienen el sufijo *Action*.

La llamada de las URL se basa en el árbol de los paquetes:

La clase `ejemplo37.acciones.Cliente.java` será llamada por la URL */cliente* y el nombre de espacio */*.

La clase `ejemplo37.actions.paginacion.Cliente.java` será llamada por la URL */paginacion/cliente* y el nombre de espacio */paginacion*.

También podemos indicar al complemento que ignore determinados paquetes (para no aplicar las acciones) con el parámetro *struts.convention.exclude.packages* en el archivo de configuración*struts.xml*. Asimismo, podemos realizar la acción a la inversa, especificar a Struts en qué paquete aplicar las acciones con el parámetro *struts.convention.action.packages*.

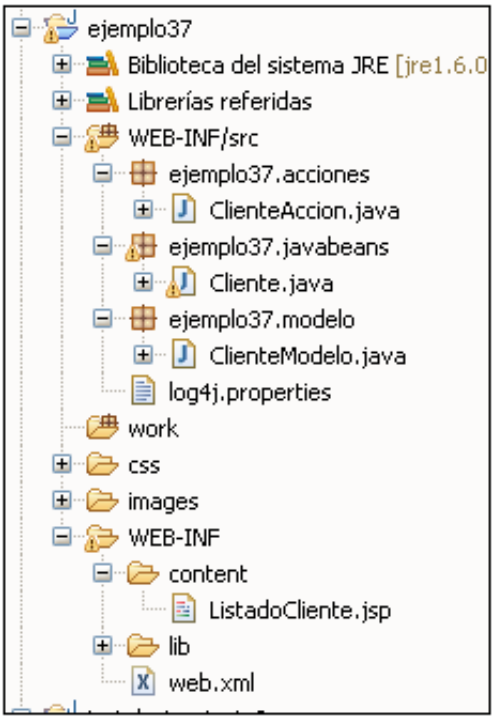
2. Notación de acciones

La utilización de anotaciones de acciones permite especificar la URL que ejecutará la acción para la aplicación. La anotación *@Action* permite declarar una URL y la anotación *@Actions* permite asignar varias URL. La notación de URL es sensible a mayúsculas y minúsculas, pero el sufijo *.action* es utilizado automáticamente en las URL. Podemos especificar tantas anotaciones de acciones como funcionalidades que se van a ejecutar posee la clase.

El proyecto *ejemplo37* permite hacer una lista de los clientes a partir de la colección del modelo. La siguiente clase de acción permite realizar el procesamiento de la aplicación con la ayuda de anotaciones de acciones.

```
Código: ejemplo37.actions.ClienteAccion.java
package ejemplo37.actions;

import java.util.List;
import org.apache.struts2.convention.annotation.Action;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo37.javabeans.Cliente;
import ejemplo37.modelo.ClienteModelo;

@SuppressWarnings("serial")
public class ClienteAccion extends ActionSupport {

      // lista de clientes
      private List<Cliente> listaClientes;
```

```
	// anotación del complemento Convention
	@Action("/ListadoCliente")
	public String execute()
	{
	  System.out.println("Registró en el método de lista");
	  listaClientes=ClienteModelo.getListaClientes();
	  return SUCCESS;
	}

	public List<Cliente> getListaClientes() {
		return listaClientes;
	}

	public void setListaClientes(List<Cliente> listaClientes) {
		this.listaClientes = listaClientes;
	}
}
```



Árbol del proyecto ejemplo37

Para definir la acción por defecto, el parámetro *<default-action-ref name="ListadoCliente" />* del archivo de configuración *struts.xml* puede sustituirse por una definición de acción en forma de anotación. En nuestro proyecto, la acción ejecutada por la URL actual (value="/") es la función*Listado()* de nuestro proyecto.

```
// anotación del complemento Convention
@Actions({
		@Action(value="/",
		results={@Result(name="success", location="/WEB-
INF/content/ListadoCliente.jsp")}
		),
			@Action(value="/ListadoCliente",
			results={@Result(name="success", location="/
WEB-INF/content/ListadoCliente.jsp")}
			)
	})
public String listado()
{
```

```
    System.out.println("Registro en el método de lista");
    listaClientes=ClienteModelo.getListaClientes();
    return SUCCESS;
}
```

3. Anotaciones de resultados

Por defecto, como ya hemos explicado en este capítulo, el complemento *Convention* utiliza el directorio */WEB-INF/content*, así como cada página asociada a la acción para realizar el enrutamiento. La anotación *@Results* ofrece 2 tipos de resultados: globales o locales. Los resultados globales permiten compartir los resultados con todos los métodos de la clase de acción. Al igual que las acciones, los resultados se definen en forma de anotaciones. De manera inversa, los resultados locales permite especificar los resultados acción por acción.

Para lo siguiente utilizamos un nuevo proyecto llamado *ejemplo38*, adaptado del proyecto *ejemplo14*para aplicar el principio de la configuración cero y la gestión de resultados.

4. Anotación de interceptores

Para gestionar los clientes utilizamos el interceptor *paramsPrepareParamsStack* con la siguiente declaración en el archivo de configuración de la aplicación *struts.xml*:

```
<action name="Editar_Cliente" class="ejemplo14.ClienteAccion"
method="editar">
 <interceptor-ref name="paramsPrepareParamsStack"/>
 <result name="success">/jsp/EditarCliente.jsp</result>
</action>
```

La utilización del interceptor con la configuración cero se realiza con la anotación *@InterceptorRef* y el nombre del interceptor que se utilizará antes de la definición de la clase de acción.

```
Código: ejemplo38.acciones.ClienteAccion.java
...
@SuppressWarnings("serial")
@InterceptorRef("paramsPrepareParamsStack")
public class ClienteAccion extends ActionSupport implements
Preparable{
      ...
}
```

La anotación *@InterceptoRefs* permite declarar varios interceptores en la misma clase de acción.

```
@InterceptorRefs({
    @InterceptorRef("paramsPrepareParamsStack"),
    @InterceptorRef("validation")
})
public class ClienteAccion extends ActionSupport implements
Preparable{
      ...
}
```

También es posible colocar las anotaciones de interceptores al nivel de cada declaración de acción. Para ello, utilizamos el parámetro de acción llamado *interceptorRefs*. Del mismo modo, el atributo`params` de la anotación *@InterceptorRef* se compone de una tabla de claves que permite especificar la configuración del interceptor: {*"clave1","valorclave1","clave2","valorclave2"...*}.

En nuestro proyecto *ejemplo38*, debemos utilizar, además del parámetro *paramsPrepareParamsStack*, el interceptor *validation* para la comprobación de los datos introducidos del cliente en los formularios de registro y de edición. La tabla *params* contiene también un conjunto de propiedades para especificar que las validaciones se realizan con la ayuda de un archivo XML sin declaración en la

clase.

```
interceptorRefs=@InterceptorRef(value="validation",params={"progra
mmatic", "true", "declarative", "false"})
```

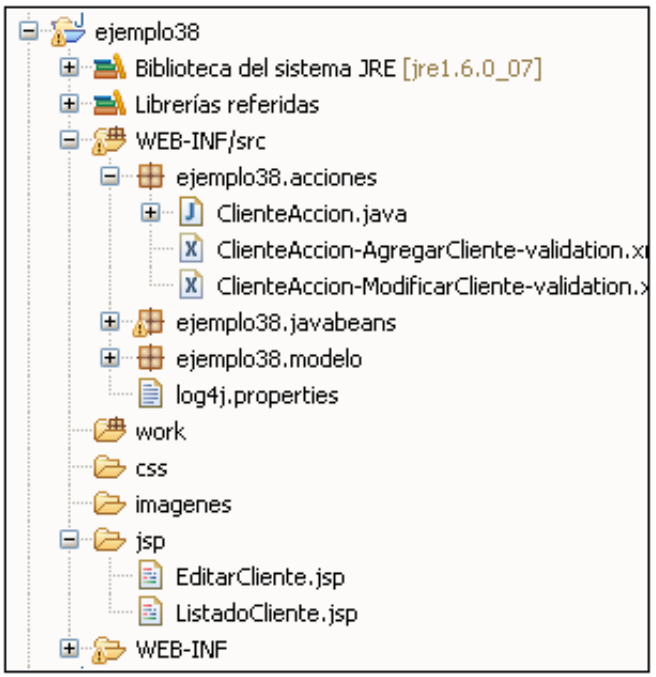
El proyecto ejemplo38 ahora puede ser adaptado a partir del proyecto completo que utilizaba anteriormente el archivo de configuración *struts.xml*.

Código: ejemplo38.acciones.ClienteAccion.java

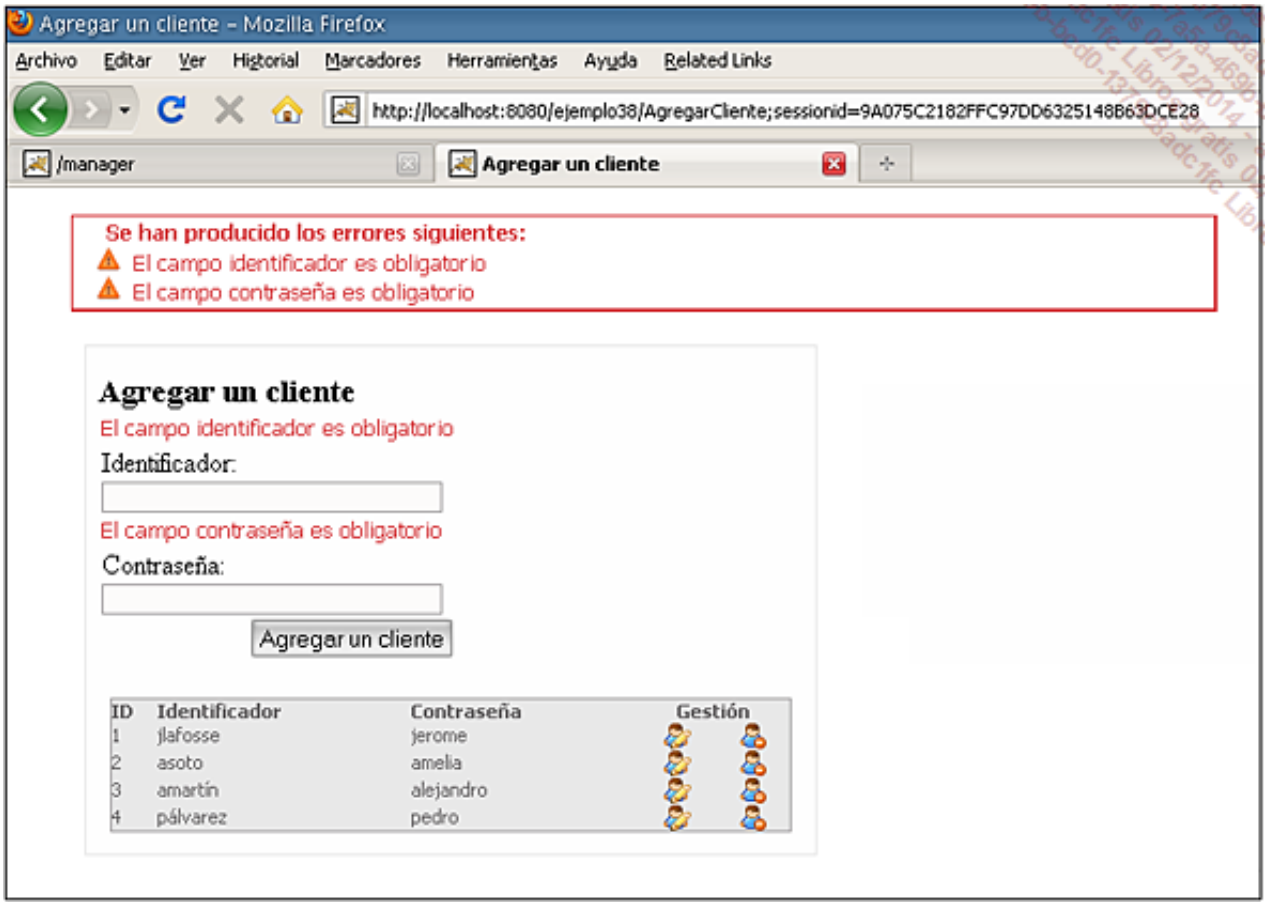
```
package ejemplo38.acciones;

import java.util.List;
import org.apache.struts2.convention.annotation.Action;
import org.apache.struts2.convention.annotation.Actions;
import org.apache.struts2.convention.annotation.InterceptorRef;
import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.Preparable;
import org.apache.struts2.convention.annotation.Result;
import ejemplo38.javabeans.Cliente;
import ejemplo38.modelo.ClienteModelo;

@SuppressWarnings("serial")
@InterceptorRef("paramsPrepareParamsStack")
public class ClienteAccion extends ActionSupport implements
Preparable{

      // lista de clientes
      private List<Cliente> listaClientes;
      // objeto cliente
      private Cliente cliente;
      // cliente que se va a modificar
      private int idClienteActual;

      public void prepare() throws Exception {
       // durante la creación, crear un nuevo objeto vacío
       if(idClienteActual==0)
       {
            cliente=new Cliente();
       }
       // durante la modificación, devolver la información del objeto
       else
       {
            cliente=ClienteModelo.getCliente(idClienteActual);
       }
      }

      // anotación del complemento Convention
      @Actions({
         @Action(value="/",
       results={@Result(name="success",
location="/jsp/ListadoCliente.jsp")}
       ),
             @Action(value="/ListadoCliente",
             results={@Result(name="success",
location="/jsp/ListadoCliente.jsp")}
             )
       })
       public String listado()
       {
         listadoCliente=ClienteModelo.getListadoCliente();
         return SUCCESS;
       }
```

```
	// anotación del complemento Convention
	@Action(value="/AgregarCliente",
	results={
			@Result(name="success",
location="/ListadoCliente", type="redirect"),
			@Result(name="input",
location="/jsp/ListadoCliente.jsp")
			},

interceptorRefs=@InterceptorRef(value="validation",params={"progra
mmatic", "true", "declarative", "false"})
	)
	public String agregar()
	{
		ClienteModelo.agregar(cliente);
	    return SUCCESS;
	}


	// Mostrar el formulario de edición
	@Action(value="/EditarCliente",
	results={@Result(name="success",
location="/jsp/EditarCliente.jsp")}
	)
	public String editar()
	{
		return SUCCESS;
	}


	// modificar un cliente
	@Action(value="/ModificarCliente",
			results={
			@Result(name="success",
location="/ListadoCliente", type="redirect"),
			@Result(name="input",
location="/jsp/EditarCliente.jsp")
			}
	)
	public String modificar()
	{
		ClienteModelo.modificar(cliente);
		return SUCCESS;
	}


	// eliminar un cliente a partir del parámetro recibido llamado
idCliente
	@Action(value="/EliminarCliente",
	results={@Result(name="success", location="/ListadoCliente",
type="redirect")}
	)
	public String eliminar()
	{
		ClienteModelo.eliminar(idClienteActual);
		return SUCCESS;
	}


	public List<Cliente> getListaClientes() {
		listaClientes=ClienteModelo.getListaClientes();
		return listaClientes;
	}

	public void setListaClientes(List<Cliente> listaClientes) {
```

```
		this.listaClientes = listaClientes;
	}

	public Cliente getCliente() {
		return cliente;
	}

	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

	public int getIdClienteActual() {
		return idClienteActual;
	}

	public void setIdClienteActual(int idClienteActual) {
		this.idClienteActual = idClienteActual;
	}
}
```



Árbol del proyecto ejemplo38



Gestión completa de los clientes a partir de anotaciones

5. Anotaciones de espacios de nombre

La anotación de namespace permite cargar la convención de nomenclatura utilizada por defecto con Struts. Cuando esta anotación se coloca al nivel de la clase de acción, ésta se aplica a todos los métodos de la acción. Podemos modificar nuestro proyecto anterior *ejemplo38* añadiendo un nuevo namespace para acceder al proyecto por URL diferentes.

```
Código: ejemplo38.acciones.ClienteAccion.java
package ejemplo38.acciones;
...
@Namespace("/GestionCliente")
@SuppressWarnings("serial")
@InterceptorRef("paramsPrepareParamsStack")
public class ClienteAccion extends ActionSupport implements
Preparable{
...
```

Ahora se puede acceder a la aplicación desde las siguientes URL:

http://localhost:8080/ejemplo38/ListadoCliente.action

http://localhost:8080/ejemplo38/GestionCliente/ListadoCliente.action

6. Anotaciones de las vistas

La anotación *@ResultPath* permite especificar la ruta de almacenamiento de los resultados. Podemos utilizar esta anotación antes de la declaración de la clase de acción.

```
Código: ejemplo38.acciones.ClienteAccion.java
package ejemplo38.acciones;
...
@ResultPath("/jsp")
@SuppressWarnings("serial")
@InterceptorRef("paramsPrepareParamsStack")
public class ClienteAccion extends ActionSupport implements
Preparable{
...
```

7. Anotaciones de las excepciones

La anotación *@ExceptionMappings* se define al nivel de la declaración de la clase de acción y contiene anotaciones *@ExceptionMapping* que permiten definir los tipos de excepciones que se van a gestionar.

```
Código: ejemplo38.acciones.ClienteAccion.java
package ejemplo38.acciones;
...
@ExceptionMappings({
    @ExceptionMapping(exception =
"java.lang.NullPointerException", result = "success", params =
{"parametro1", "valorparametro1"})
})
@SuppressWarnings("serial")
@InterceptorRef("paramsPrepareParamsStack")
public class ClienteAccion extends ActionSupport implements
Preparable{
...
```

8. Carga automática de la configuración

El complemento *Convention* puede recargarse automáticamente sin reiniciar el contexto de la aplicación como el archivo de configuración *struts.xml* y la directiva *<constant name="struts.devMode" value="true" />*.

Para recargar automáticamente el complemento *Convention*, debemos utilizar la siguiente constante en el archivo *struts.xml* de la aplicación: *<constant name="struts.convention.classes.reload" value="true" />* o en el archivo de propiedades.

En resumen

El framework Struts ofrece una técnica llamada configuración cero o zero configuration que permite no crear y escribir el archivo de configuración de la aplicación *struts.xml* y que no sea necesario declarar cada acción, resultado o interceptor. Esta técnica utiliza para ello el complemento *Convention* incluido en el paquete y que se puede descargar desde el sitio del framework. La aplicación de esta configuración cero se basa en las anotaciones Java 5 situadas al principio de las clases o métodos de clases. Cada paquete de la aplicación debe respetar una convención de nomenclatura para poder beneficiarse de la configuración por anotación simplificada.

Cuando se respeta esta convención de nomenclatura, los desarrolladores pueden crear anotaciones para las acciones, los resultados, los interceptores que se van a utilizar, el espacio de nombre de la aplicación, las vistas que se van a mostrar y la gestión de las excepciones. El proyecto *ejemplo38*ilustra en detalle la aplicación de esta técnica a partir de las principales funcionalidades de una aplicación de Internet (hacer listados, agregar, modificar, eliminar).

Presentación

En este capítulo se explicará en detalle los distintos interceptores, así como el lenguaje de manipulación de expresiones Java Object-Graph Navigation Language OGNL. Este lenguaje de manipulación de datos simplifica el acceso a la información de los getters y setters de los JavaBeans. De igual modo, esta biblioteca puede utilizarse en asociación con las taglibs de JSTL.

Interceptores de Struts

Los interceptores son filtros que permiten realizar tareas para simplificar y mejorar el trabajo de los desarrolladores. En la mayoría de los casos, los interceptores incluidos con Struts son suficientes y es fundamental entender las ventajas que cada uno de ellos aporta. Estos son los interceptores incluidos por defecto con el framework.

params

Gestión del mapping entre los parámetros de las consultas y las propiedades de las acciones.

staticParams

Gestión de los parámetros estáticos declarados en las definiciones y clases de acción.

prepare

Gestión del acceso a las plantillas de clases.

scope

Gestión del mecanismo de las sesiones.

servletConfig

Gestión del acceso a las clases `HttpServletRequest` y `HttpServletResponse`.

validation

Gestión de las validaciones de formularios.

token

Gestión del doble envío o double submit.

tokenSession

Gestión del doble envío o double submit con un parámetro de sesión.

profiling

Gestión de la creación de perfiles de las acciones.

timer

Gestión del tiempo de ejecución de las acciones.

roles

Gestión de las funciones de los Realms para la seguridad de las aplicaciones.

modelDriven

Gestión de los modelos de persistencia.

scopedModelDriven

Gestión de los modelos de persistencia como el interceptor anterior, pero con gestión de sesión.

logger

Gestión de las acciones.

execAndWait

Gestión de la carga de páginas en procesamientos más o menos largos.

debugging

Gestión de la depuración de las aplicaciones.

exception

Gestión del procesamiento de las excepciones.

i18n

Gestión de la internacionalización para las aplicaciones multilingües.

fileUpload

Gestión de las cargas de archivos.

store

Gestión de los ámbitos de aplicación de los mensajes de aplicación y de errores.

chain

Gestión del encadenado de acciones y de la transmisión de parámetros.

createSession

Gestión de la sesión del usuario actual.

conversionError

Gestión de los errores de conversión en las acciones.

alias

Gestión de los nombres de parámetros en las consultas HTTP.

checkbox

Gestión de las casillas de verificación en los formularios.

cookie

Gestión de la cookie del usuario.

workflow

Gestión de la llamada de los métodos de validación en las clases de acción.

actionMappingParams

Gestión del mapping de los parámetros de las acciones.

Object-Graph Navigation Language OGNL

OGNL es un lenguaje de descripción que permite acceder y modificar las propiedades de los objetos Java. OGNL forma parte del proyecto opensymphony (http://www.opensymphony.com y http://www.ognl.org/) y puede utilizarse como complemento de las taglibs de Java EE.

OGNL hace referencia a los objetos en función de su ámbito de aplicación y de la colección de datos (Context Map):

- *application*: esta colección contiene los atributos presentes en el contexto de la aplicación*ServletContext*.
- *session*: esta colección contiene los atributos presentes en la sesión del usuario.
- *request*: esta colección contiene todos los atributos presentes en la consultar actual.
- *parameters*: esta colección contiene los parámetros de la consulta actual.
- *attr*: esta colección busca los atributos en el siguiente orden: `request`, `session` y `application`.

Para leer objetos en la pila de propiedades, podemos utilizar la etiqueta `<s:property value=''nombredelapropiedad''/>` con el parámetro *value* adaptado. Dado que la acción hace referencia a los parámetros de la pila, el caracter almohadilla (*#*) es opcional. Por contra, cuando deseamos acceder a los objetos en el contexto (*session*, *application* o *attr*) debemos utilizar la notación *#*.

Por ejemplo, las siguientes notaciones permiten leer los parámetros según el ámbito de aplicación:

```
<s:property value="client.identificador"/>
<s:property value="#request.cliente['identificador']"/>
<s:property value="#session.contrasena"/>
<s:property value="#application.emailContacto"/>
```

La notación de puntos (y por tabla) se utiliza para acceder a los objetos del contexto. Podemos acceder, por ejemplo, a la propiedad *identificador* del objeto *cliente* de las siguientes formas:

```
<s:property value="cliente.identificador"/>
<s:property value="cliente['identificador']"/>
<s:property value="#request.cliente['identificador']"/>
```

OGNL también permite ejecutar los métodos de clases para mostrar un procesamiento específico mediante el carácter arroba @. Para ello, debemos especificar en la llamada, el nombre completamente cualificado de la clase seguido del nombre del método.

Por ejemplo, para ejecutar el método estático `Visualizacion()` de la clase estática `CajaHerramientas`presente en el paquete *ejemplo39*.

```
<s:property value="@ejemplo39.CajaHerramientas@Visualizacion"/>
```

OGNL también permite manipular las tablas de valores (*Array*) mediante el getter asociado y las distintas propiedades dedicadas. La tabla de países declarados en la clase de acción se utiliza en la vista mediante OGNL.

```
// Lista de países
private  String[] listaPaises;
{
    listaPaises=new String[]
"España","Inglaterra","Alemania","Suiza"};
}
```

Podemos mostrar directamente el contenido de la tabla mediante su nombre, su tamaño con el

atributo `length` y acceder a los elementos por índice.

```
Lista de pa&iacute;ses: <s:property value="listaPaises"/><br/>
Lista de pa&iacute;ses: <s:property value="[0].listaPaises"/> <br>
Tama&ntilde;o de la tabla de los pa&iacute;ses: <s:property
value="listaPaises.length"/><br/>
Acceso al primer &iacute;ndice de la tabla: <s:property
value="listaPaises[0]"/><br/>
```

OGNL también permite manipular las listas de valores en forma de claves/valores (*Map*) mediante el getter asociado y las distintas propiedades dedicadas. La lista de ciudades declaradas en la clase de acción se utiliza en la vista mediante OGNL.

```
// Lista de ciudades
private  Map<String,String> listaCiudades=new
HashMap<String,String>();
{
    listaCiudades.put("MA", "Madrid");
    listaCiudades.put("BA", "Barcelona");
    listaCiudades.put("VA", "Valencia");
}
```

Podemos mostrar directamente el contenido de la lista mediante su nombre, su tamaño con el atributo `size` y acceder a los elementos por clave.

```
Lista de ciudades: <s:property value="listaCiudades"/><br/>
Lista de ciudades: <s:property value="[0].listaCiudades"/> <br>
Tama&ntilde;o de la lista de ciudades: <s:property
value="listaCiudades.size"/><br/>
Acceso al primer &iacute;ndice de la tabla: <s:property
value="listaCiudades['BA']"/><br/>
```

OGNL también permite manipular las listas de objetos (*List*) mediante el getter asociado y las distintas propiedades dedicadas. La lista de clientes declarados en la clase de acción se utiliza en la vista mediante OGNL.

```
// lista de clientes
private  List<Cliente> listaClientes=new ArrayList<Cliente>();
{
    Cliente c1=new Cliente();
    c1.setIdentificador("jlafosse");
    c1.setContrasena("jerome");
    Cliente c2=new Cliente();
    c2.setIdentificador("asoto");
    c2.setContrasena("amelia");

    listaClientes.add(c1);
    listaClientes.add(c2);
}
```

Podemos mostrar directamente el contenido de la lista mediante su nombre, su tamaño con el atributo `size` y acceder a los elementos por clave.

```
Lista de clientes: <s:property value="listaClientes"/><br/>
Lista de clientes: <s:property value="[0].listaClientes"/> <br>
Tama&ntilde;o de la lista de clientes: <s:property
value="listaClientes.size"/><br/>
Acceso al primer &iacute;ndice de la tabla: <s:property
value="listaClientes[0].getIdentificador()"/><br/>
&iquest;Lista vac&iacute;a? : <s:property value="listaClientes.isEmpty"/><br/>
```

OGNL permite ejecutar métodos de la clase de acción. La técnica es la misma que para el acceso a los objetos de la clase y de la internacionalización.

```
// método de la clase
public String getVisualizacion() {
    return "Método de la clase de acción: "+new
GregorianCalendar().getTime();
}
```

Podemos mostrar directamente el resultado de la ejecución del método de acción mediante su nombre.

```
Ejecutar un m&eacute;todo de clase de acci&oacute;n:
<s:property value="%{getVisualizacion()}"/>
```

El nuevo proyecto *ejemplo39* ilustra detalladamente los ejemplos anteriores y presenta las distintas manipulaciones de OGNL.

```
Código: 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="true" />

    <package name="ejemplo39" namespace="/" extends="struts-
default">
        <default-action-ref name="OGNL" />

        <action name="OGNL" class="ejemplo39.OgnlAccion">
            <result>/jsp/Ognl.jsp</result>
        </action>

    </package>
</struts>
```

```
Código: ejemplo39.OgnlAccion.java
package ejemplo39;

import java.util.ArrayList;
import java.util.GregorianCalendar;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.apache.struts2.interceptor.SessionAware;
import com.opensymphony.xwork2.ActionSupport;
import ejemplo39.javabeans.Cliente;

@SuppressWarnings("serial")
public class OgnlAccion extends ActionSupport implements
SessionAware{

      // objeto cliente
      private Cliente cliente;
      // objeto session
      private Map<String,Object> sessionMap;
```

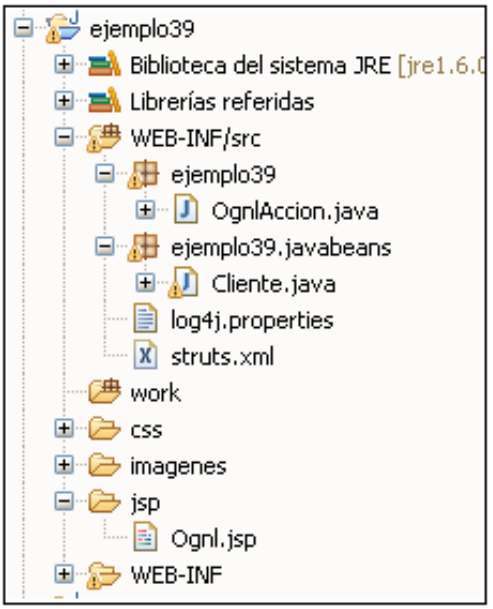
```
	// tabla de países
	private  String[] listaPaises;
	{
		listaPaises=new String[]
{"España","Inglaterra","Alemania","Suiza"};
	}

	// Lista de ciudades
	private  Map<String,String> listaCiudades=new
HashMap<String,String>();
	{
		listaCiudades.put("MA", "Madrid");
		listaCiudades.put("BA", "Barcelona");
		listaCiudades.put("VA", "Valencia");
	}

	// lista de clientes
	private  List<Cliente> listaClientes=new ArrayList<Cliente>();
	{
		Cliente c1=new Cliente();
		c1.setIdentificador("jlafosse");
		c1.setContrasena("jerome");
		Cliente c2=new Cliente();
		c2.setIdentificador("asoto");
		c2.setContrasena("amelia");

		listaClientes.add(c1);
		listaClientes.add(c2);
	}

	public Cliente getCliente() {
		return cliente;
	}
	public void setCliente(Cliente cliente) {
		this.cliente = cliente;
	}

	public void setSession(Map<String,Object> map)
	{
		this.sessionMap=map;
	}

	public String[] getListaPaises() {
		return listaPaises;
	}

	public Map<String, String> getListaCiudades() {
		return listaCiudades;
	}

	public List<Cliente> getListaClientes() {
		return listaClientes;
	}

	// método de la clase
	public String getVisualizacion() {
		return "Método de la clase de acción: "+new
GregorianCalendar().getTime();
	}

	// añadir la información del cliente a la sesión
	public String execute()
	{
		// creación de un objeto cliente
		cliente=new Cliente();
```

```
            cliente.setIdentificador("jlafosse");
            cliente.setContrasena("jerome");

            // añadir la contraseña a la sesión
            sessionMap.put("contrasena", "lafosse");

            return SUCCESS;
      }
}
```

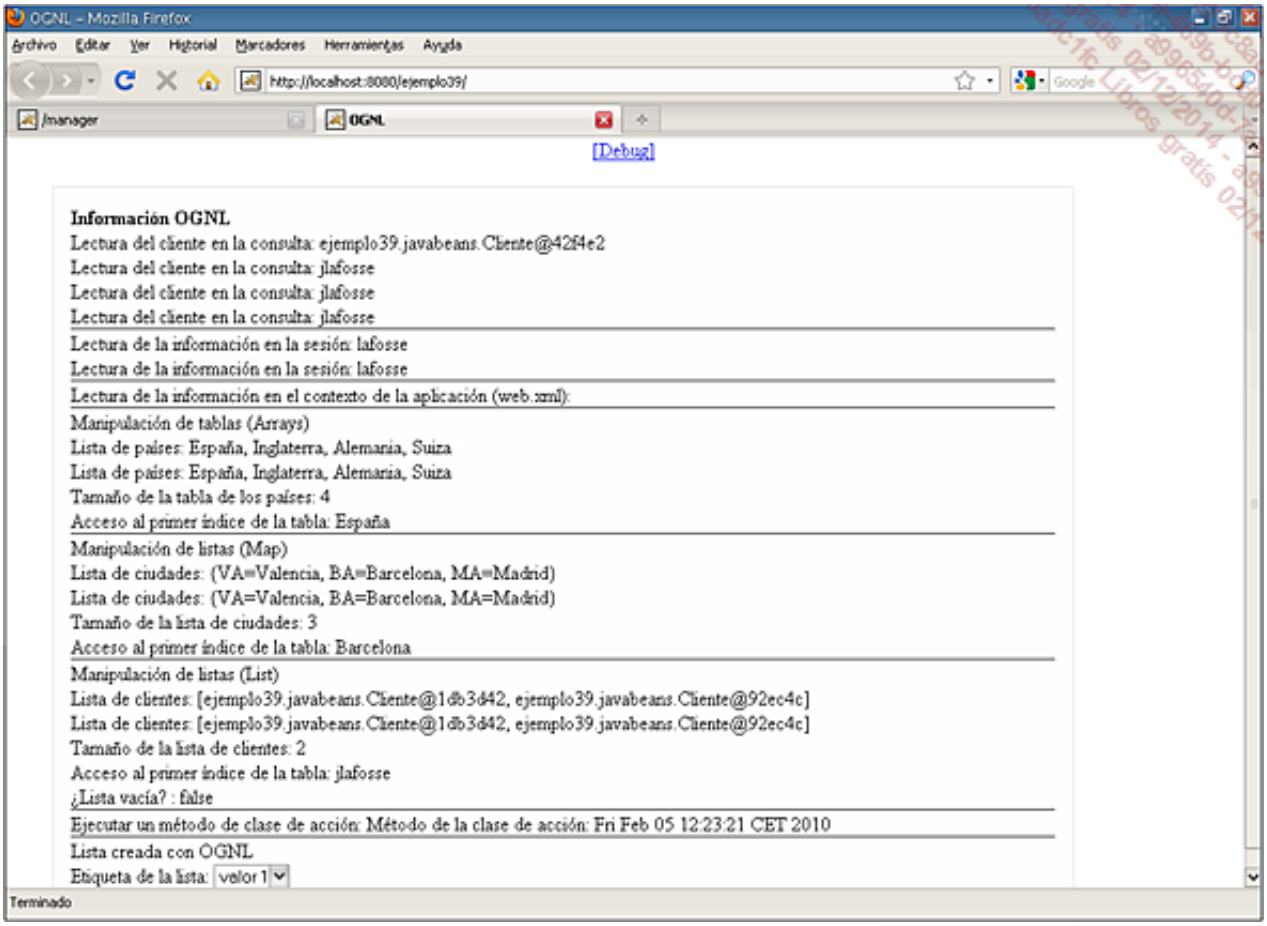
Código: /jsp/Ognl.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>OGNL</title>
<style type="text/css">@import url(css/estilos.css);</style>
</head>
<body>
<s:debug/>
<div id="carta">
      <h4>Informaci&oacute;n OGNL</h4>
      Lectura del cliente en la consulta:
<s:property value="cliente"/><br/>
      Lectura del cliente en la consulta:
<s:property value="cliente.identificador"/><br/>
      Lectura del cliente en la consulta:
<s:property value="cliente['identificador']"/><br/>
      Lectura del cliente en la consulta:
<s:property value="#request.cliente['identificador']"/><br/>
      <hr/>
      Lectura de la informaci&oacute;n en la sesi&oacute;n:
<s:property value="#session.contrasena"/><br/>
      Lectura de la informaci&oacute;n en la sesi&oacute;n:
<s:property value="#session['contrasena']"/><br/>
      <hr/>
      Lectura de la informaci&oacute;n en el contexto de
la aplicaci&oacute;n (web.xml): <s:property
value="#application.emailContacto"/><br/>
      <hr/>
      Manipulaci&oacute;n de tablas (Arrays)<br/>
      Lista de pa&iacute;ses: <s:property value="listaPaises"/><br/>
      Lista de pa&iacute;ses: <s:property value="[0].listaPaises"/>
<br>
      Tama&ntilde;o de la tabla de los pa&iacute;ses: <s:property
value="listaPaises.length" /><br/>
      Acceso al primer &iacute;ndice de la tabla:
<s:property value="listaPaises[0]"/><br/>
      <hr/>
      Manipulaci&oacute;n de listas (Map)<br/>
      Lista de ciudades: <s:property
value="listaCiudades"/><br/>
      Lista de ciudades: <s:property
value="[0].listaCiudades"/> <br>
      Tama&ntilde;o de la lista de ciudades: <s:property
value="listaCiudades.size"/><br/>
      Acceso al primer &iacute;ndice de la tabla:
<s:property value="listaCiudades['BA']"/><br/>
      <hr/>
      Manipulaci&oacute;n de listas (List)<br/>
      Lista de clientes: <s:property
value="listaClientes"/><br/>
      Lista de clientes: <s:property
value="[0].listaClientes"/> <br>
      Tama&ntilde;o de la lista de clientes: <s:property
```

```
value="listaClientes.size"/><br/>
      Acceso al primer &iacute;ndice de la tabla:
<s:property value="listaClientes[0].getIdentificador()"/><br/>
      &iquest;Lista vac&iacute;a? : <s:property
value="listaClientes.isEmpty"/><br/>
      <hr/>
      Ejecutar un m&eacute;todo de clase de acci&oacute;n:
<s:property value="%{getVisualizacion()}"/>
      <hr/>
      Lista creada con OGNL<br/>
      <s:select label="Etiqueta de la lista" name="nombre"
list="{'valor 1','valor 2','valor 3'}"/>
</div>
</body>
</html>
```



Árbol del proyecto ejemplo39



Manipulación de la información con la ayuda de OGNL

En resumen

En este último capítulo se ha presentado una lista detallada de los interceptores utilizados por el framework Struts. De igual modo, la sección Object-Graph Navigation Language OGNL explica de forma extensa el lenguaje de manipulación OGNL, que permite acceder y modificar los objetos Java.