# J2EE – JSP, Serviets y Strusts

Roberto Casado-Vara [1]

[1] University of Salamanca, Plaza de los Caídos s/n – 37002 – Salamanca, Spain
rober@usal.es

**Resumen**. En este capítulo vamos a describir la tecnología J2EE y sus entornos tecnológicos mediante los cuales se pueden desarrollar aplicaciones web usando el lenguaje de programación Java. Un servidor de aplicaciones es un ordenador que funciona como contenedor de aplicaciones permitiendo la ejecución de las mismas dentro de la red. El término también hace referencia al software instalado en tal ordenador para facilitar la ejecución de esas aplicaciones. Mediante los servidores de aplicaciones se podrán desplegar las aplicaciones en la web y podrán ser usadas por los usuarios finales. Por último, en este capitulo se explican los Servlets y los structs presentes en la tecnología J2EE.

**Palabras clave:** J2EE; JSP; Servlets.

**Abstract.** In this chapter we are going to describe the J2EE technology and its technological environments through which web applications can be developed using the Java programming language. An application server is a computer that works as a container of applications allowing the execution of the same within the network. The term also refers to the software installed on such a computer to facilitate the execution of those applications. Through the application servers the applications can be deployed on the web and used by end users. Finally, this chapter explains the Servlets and structs present in J2EE technology.

**Keywords**: J2EE; JSP; Servlets



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

# 1. Introducción JSP - Servlets

## 1.1 Arquitectura Cliente – Servidor

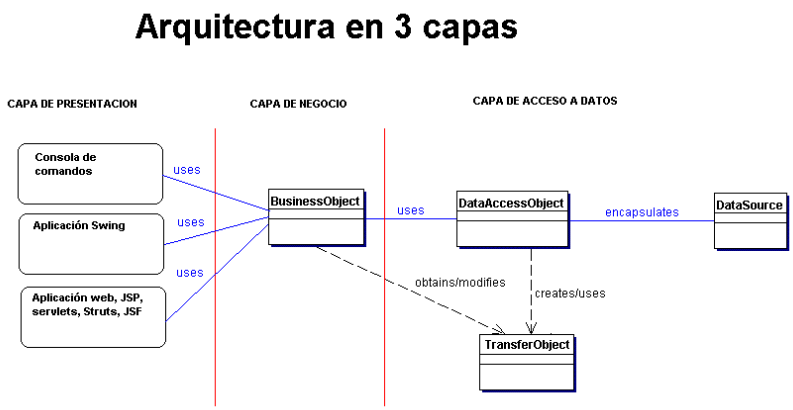
### 1.1.1 Introducción

La **arquitectura cliente-servidor** llamado modelo cliente-servidor es una forma de dividir y especializar programas y equipos de procesamiento a fin de que la tarea que cada uno de ellos realizada se efectúe con la mayor eficiencia, y permita simplificarlas. De esta manera se libera al equipo servidor de la realización de tareas propias de los clientes optimizando los recursos del mismo.

### 1.1.2 Arquitectura en 3 capas

Dentro de la parte servidora puede adoptarse una programación por capas, dicha programación facilita las labores de mantenimiento y de abstracción entre elementos de distinta índole en una aplicación. Dentro de esta arquitectura multicapa la más extendida es la versión con 3 capas:

- Capa de datos: encargada del "diálogo" con las bases de datos.
- Capa de negocio: la que contiene la funcionalidad de la aplicación.
- Capa de presentación: la encargada de interactuar con el usuario para mostrar y recoger datos.




### 1.1.3 Ventajas de la arquitectura cliente-servidor

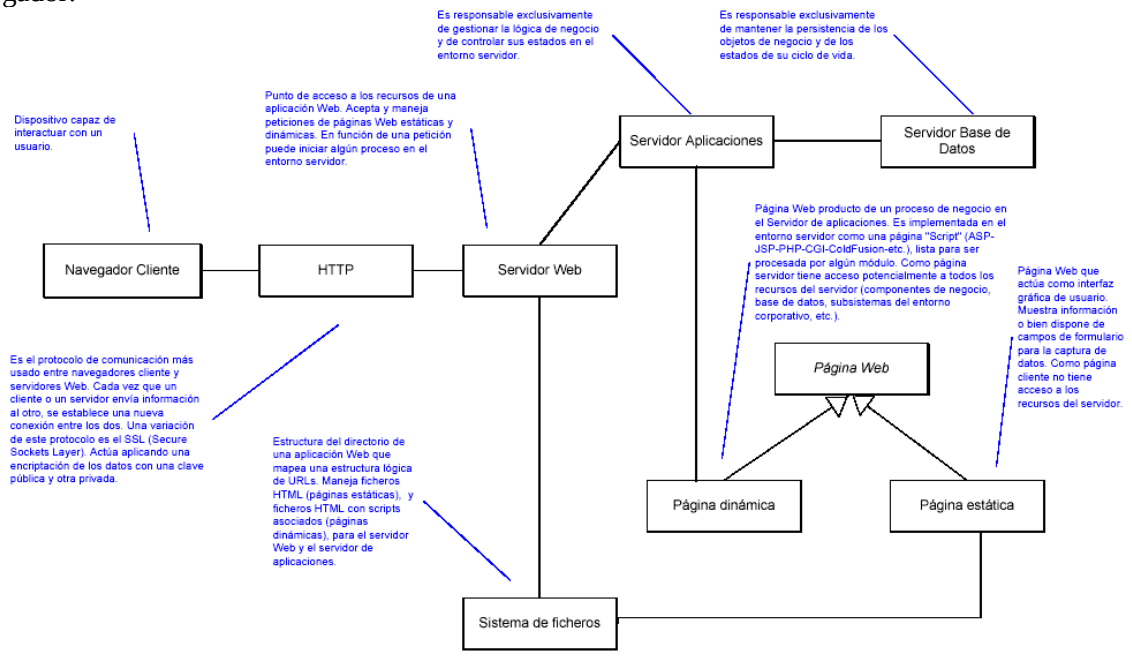
El servidor no necesita tanta potencia de procesamiento, parte del proceso se reparte con los clientes.
Se reduce el tráfico de red considerablemente. Idealmente, el cliente se conecta al servidor cuando es estrictamente necesario, obtiene los datos que necesita y cierra la conexión dejando la red libre para otra conexión.

## 1.2 Arquitectura Web

Denominamos arquitectura web a aquella arquitectura **cliente–servidor** que está especializada en trabajar con información para la web. De esta forma el cliente será un usuario con acceso a internet

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

mediante el uso de un navegador web y que interactúa con dicha aplicación mediante ese navegador.




## 1.3 Servidor Web

Un **servidor web** es un programa que implementa el procotolo HTTP (hypertext transfer protocol). Este protocolo está diseñado para transferir lo que llamamos hipertextos, páginas web o páginas HTML (hypertext markup language); es decir textos complejos con enlaces, figuras, formularios, botones y objetos incrustados como animaciones o reproductores de sonidos.
Un servidor web se encarga de mantenerse a la espera de peticiones HTTP llevada a cabo por un cliente HTTP que solemos conocer como navegador. El navegador realiza una petición al servidor y éste le responde con el contenido que el cliente solicita [1-8].
Sobre el servicio web clásico podemos disponer de aplicaciones web. Éstas son fragmentos de código que se ejecutan cuando se realizan ciertas peticiones o respuestas HTTP. Hay que distinguir entre:

- **Aplicaciones en el lado del cliente**: el cliente web es el encargado de ejecutarlas en la máquina del usuario. Son las aplicaciones tipo Java o Javascript: el servidor proporciona el código de las aplicaciones al cliente y éste, mediante el navegador, las ejecuta. Es necesario, por tanto, que el cliente disponga de un navegador con capacidad para ejecutar aplicaciones (también llamadas scripts). Normalmente, los navegadores permiten ejecutar aplicaciones escritas en lenguaje javascript y java, aunque pueden añadirse más lenguajes mediante el uso de plugins.
- **Aplicaciones en el lado del servidor**: el servidor web ejecuta la aplicación; ésta, una vez ejecutada, genera cierto código HTML; el servidor toma este código recién creado y lo envía al cliente por medio del protocolo HTTP.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

## 1.4 Servidor de Aplicaciones

### 1.4.1 Introducción

Un **servidor de aplicaciones** es un ordenador que funciona como contenedor de aplicaciones permitiendo la ejecución de las mismas dentro de la red. El término también hace referencia al software instalado en tal ordenador para facilitar la ejecución de esas aplicaciones.

### 1.4.2 Servidores de aplicación Java EE

Como consecuencia del éxito del lenguaje de programación Java, el término servidor de aplicaciones usualmente hace referencia a un servidor de aplicaciones Java EE (aunque también se utiliza para otros lenguajes como por ejemplo .NET).
WebSphere (IBM), Oracle Application Server (Oracle Corporation) y WebLogic (BEA) están entre los servidores de aplicación Java EE comerciales más conocidos.
JBoss es otro servidor de aplicaciones libre muy popular en la actualidad. Tomcat (Apache Software Foundation) si bien no es un servidor de aplicaciones, si no un contenedor de servlets suele también incluirse dentro de este grupo.
Java EE provee estándares que le permiten a un servidor de aplicaciones servir como "contenedor" de los componentes que conforman dichas aplicaciones. Estos componentes, escritos en lenguaje Java, usualmente se conocen como Servlets, Java Server Pages (JSPs) y Enterprise JavaBeans (EJBs) y permiten implementar diferentes capas de la aplicación, como la interfaz de usuario, la lógica de negocio, la gestión de sesiones de usuario o el acceso a bases de datos remotas.
La portabilidad de Java también ha permitido que los servidores de aplicación Java EE se encuentren disponibles sobre una gran variedad de plataformas, como Microsoft Windows, Unix y GNU/Linux.

### 1.4.3 Características comunes

Los servidores de aplicación típicamente incluyen también middleware (o software de conectividad) que les permite intercomunicarse con variados servicios, para efectos de confiabilidad, seguridad, no-repudiación, etc. Los servidores de aplicación también brindan a los desarrolladores una Interfaz para Programación de Aplicaciones (API), de tal manera que no tengan que preocuparse por el sistema operativo o por la gran cantidad de interfaces requeridas en una aplicación web moderna.
Los servidores de aplicación también brindan soporte a una gran variedad de estándares, tales como HTML, XML, IIOP, JDBC, SSL, etc., que les permiten su funcionamiento en ambientes web y la conexión a una gran variedad de fuentes de datos, sistemas y dispositivos.

### 1.4.4 Usos

Un ejemplo común del uso de servidores de aplicación (y de sus componentes) son los portales de Internet, que permiten a las empresas la gestión y divulgación de su información, y un punto único de entrada a los usuarios internos y externos. Teniendo como base un servidor de aplicación, dichos portales permiten tener acceso a información y servicios (como servicios Web) de manera segura y transparente, desde cualquier dispositivo.

## 1.5 Java EE

Java EE (Java Enterprise Edition, antes conocido como J2EE) es un estándar para el desarrollo, despliegue y gestión de aplicaciones distribuidas multicapa, basadas en componentes, que surge

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

como resultado de una iniciativa de la industria de desarrollo software dirigida por Sun Microsystems. Java EE combina un conjunto de tecnologías en una arquitectura con un modelo de programación de aplicaciones razonable, así como un conjunto de pruebas de compatibilidad para el desarrollo de aplicaciones en el lado del servidor.
Direcciones:

- **http://java.sun.com/**
- **http://java.sun.com/reference/api/**

Java EE es parte de lo que Sun ha denominado plataforma Java, que incluye además: JME (Java Micro Edition) y Java SE (Java Standard Edition). Esta distinción responde a la idea de realizar tres "empaquetados" básicos de la tecnología Java adaptando así su uso a distintas necesidades:

- Java SE son los entornos de ejecución, las herramientas y APIs que requieren los desarrolladores para escribir, desplegar y ejecutar applets y aplicaciones en Java. Pretende constituirse en un entorno de desarrollo rápido para aplicaciones cliente.
- JME es un entorno de ejecución Java muy pequeño, altamente optimizado para poder ser ejecutado en dispositivos consumibles, desde teléfonos, buscas, móviles, PDAs, etc. Las APIs utilizables en JME son un subconjunto de las que incluye Java SE.

# 2 Servlets

## 2.1 Introducción

Fueron introducidos por Sun en 1996 como pequeñas aplicaciones Java para añadir funcionalidad dinámica a los servidores web. Los Servlets reciben una petición del cliente y generan los contenidos apropiados para su respuesta por lo tanto están diseñados para trabajar en un modelo de procesamiento petición/respuesta.
Cada vez que llega una petición, la JVM (Java Virtual Machine) crea un hilo Java para manejar la petición, reduciendo así la sobrecarga del sistema. Solamente hay una instancia del servlet que responde todas las peticiones concurrentemente. Esto es un dato que hay que tener en cuenta para evitar problemas de datos entre distintas peticiones.
Una vez que es invocado se queda en memoria hasta que el servidor de aplicaciones lo descarga. Una desventaja de esta aproximación es que el contenido (estático y dinámico) de la página HTML generada, reside en el código fuente del servlet, por lo tanto ante cualquier operación de mantenimiento que se quiera hacer sobre las páginas generadas conlleva la compilación de nuevo de todo el código Java.

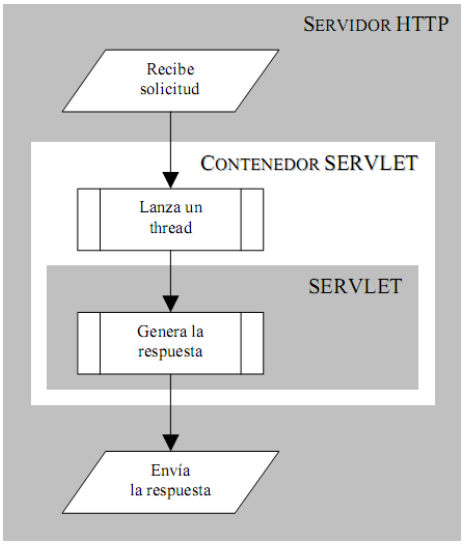
## 1.1. Características

- Son independientes del servidor.
- Permiten que un applet (aplicación Java incrustada en una página web del cliente) se comunique con un servidor web.
- Desde un servlet se puede invocar a otro servlet (local o remoto).
- Permiten obtener muy fácilmente información del cliente ya que están basados en el protocolo http, sólo tienen que analizar los datos de la petición o request.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

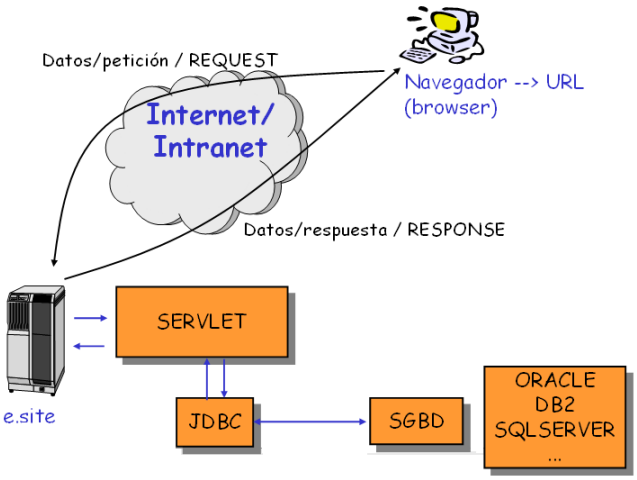
- Permiten responder directamente al cliente generando una respuesta o invocar a otro servlet o JSP para que muestre la respuesta.

## 2.2 Funcionamiento




1. Un cliente lanza una petición a un servlet.
2. Un contenedor de servlets (normalmente un servidor de aplicaciones, aunque no tiene por qué llegar a serlo) recibe una petición sobre un servlet concreto y procesa dicha petición.
3. Una vez procesada genera la respuesta que dará al cliente.
4. Por último envía esa respuesta por el mismo canal que ha recibido la petición.

Veamos a continuación un gráfico que muestra este funcionamiento sobre una aplicación:




ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

1. Un cliente lanza mediante su navegador web una petición a un servlet alojado en el servidor *e.site*. Por ejemplo una petición para consultar una disponibilidad de entradas de cine.
2. El contenedor de servlets procesa la llamada al servlet. En este caso recibiría los datos de la película a ver, fechas y cine. Con esos datos consultaría la base de datos para ver la disponibilidad.
3. Una vez consultada la disponibilidad generaría una respuesta con un listado de asientos disponibles.
4. Devolvería el código HTML necesario para que al usuario se le muestre en pantalla un listado con los asientos disponibles.

## 2.3 Requisitos

Para trabajar con Servlets y poder desplegarlos es necesario disponer de un contenedor de Servlets que cumpla la especificación **Java Servlet**. Las versiones más usadas en la actualidad son la 2.4 y 2.5 (quedando ya bastante obsoleta la 2.3).

### 2.3.1 Contenedor de Servlets Vs. Servidor de aplicaciones

Normalmente estos términos suelen confundirse y realmente no son iguales ni tienen las mismas funcionalidades aunque comparten muchas de ellas.

**Contenedor de Servlets**
Se denomina así al software que es capaz de gestionar servlets (y por ello procesar las peticiones a los mismos de manera correcta). Además de servir servlets también sirve páginas JSPs (que luego veremos con más detalle pero no son otra cosa que un tipo especial de servlets).
A la hora de procesar JSPs también es necesario que cumpla una especificación, en este caso la **Java Server Pages**. Las versiones más usadas en la actualidad son la 2.0 y 2.1 (quedando ya bastante obsoleta la 1.2).

Un ejemplo de contenedor de Servlets sería *Apache Tomcat* (en cualquiera de sus versiones).

**Servidor de aplicaciones**

Como hemos visto en la introducción, denominamos servidor de aplicaciones J2EE al software que permite manejar tanto servlets, JSPs y EJBs. Es decir sería un paso más sobre un contenedor de servlets. O visto de otra manera un contenedor de servlets sería un servidor de aplicaciones al que le faltan cosas (EJBs,…)

Un ejemplo de servidor de aplicaciones J2EE sería *Jboss*.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

## 2.4 Estructura de una aplicación web

Toda aplicación web para Java que desee exponerse en un contenedor de servlets debe de tener siempre una estructura fija de modo que el contenedor sepa siempre en donde buscar los elementos necesarios para exponer los Servlets y operar con los mismos. Esta estructura viene definida por una serie de características:

- **Nombre y tipo de fichero**: toda aplicación web será un fichero *.zip* con una extensión especial, en concreto con la extensión **.war**.
- **Carpeta WEB-INF**: colgando de la carpeta raíz del fichero war habrá siempre una carpeta de nombre **WEB-INF** con la siguiente estructura:
  - **/lib**
  - **/clases**
  - **web.xml**

Dentro de la carpeta lib irán aquellas bibliotecas (ficheros con extensión **.jar**) que sean necesarias para que la aplicación funcione (frameworks, utilerías,…)

Dentro de la carpeta clases irán los ficheros **.class** con las clases Java compiladas que forman parte del proyecto.

El fichero **web.xml** es el fichero de descripción de la aplicación. Contiene toda la configuración de la misma y tiene un aspecto similar al siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app 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">
  <servlet>
    <description>This is the description of my J2EE component</description>
    <display-name>This is the display name of my J2EE component</display-name>
    <servlet-name>ServletPrueba</servlet-name>
    <servlet-class>es.test.ServletPrueba</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ServletPrueba</servlet-name>
    <url-pattern>/servlet/ServletPrueba</url-pattern>
  </servlet-mapping>


  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

## 2.5 Creación de Servlets

A la hora de crear un servlet para una aplicación web es necesario realizar dos pasos:

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

1. Creación de una clase Java (el servlet en sí mismo) que tiene que implementar la interfaz ***javax.servlet.Servlet***; también puede optarse por heredar de la clase ***javax.servlet.http.HttpServlet*** que ya tiene métodos implementados con funcionalidades para trabajar en el ámbito de las peticiones http.
2. Una vez creada la clase es necesario "mapearla" (configurarla) dentro del fichero de configuración de la aplicación: **web.xml**.

### 2.5.1 Creación del Servlet

Si tomamos como base la clase HttpServlet podemos crear una nueva clase en la que podremos sobrescribir (entre otros) los siguientes métodos:

- **doGet():** Es invocado cuando el cliente realiza una petición GET.
- **doPost():** Es invocado cuando el cliente realiza una petición POST.
- **doHead():** Es invocado como respuesta a una petición HEAD.
- **doDelete():** Es invocado como respuesta a una petición DELETE.

```
package es.test;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ServletPrueba extends HttpServlet {
	protected void doDelete(HttpServletRequest req, HttpServletResponse resp)
				throws ServletException, IOException {
			super.doDelete(req, resp);
	}
	protected void doGet(HttpServletRequest req, HttpServletResponse resp)
				throws ServletException, IOException {
			super.doGet(req, resp);
	}
	protected void doHead(HttpServletRequest req, HttpServletResponse resp)
				throws ServletException, IOException {
			super.doHead(req, resp);
	}
	protected void doPost(HttpServletRequest req, HttpServletResponse resp)
				throws ServletException, IOException {
			super.doPost(req, resp);
	}
}
```

### 2.5.2 Configuración del Servlet

Para configurar el servlet recién creado es necesario indicar en el fichero web.xml una serie de datos:

- **Clase Java** que contiene el Servlet.
- **Nombre** (único) que se dará a ese Servlet.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **Ruta** (o URL) en la cual es accesible el Servlet.

Para ello primero indicaremos los datos de Clase y nombre:

```
<servlet>
  <description>This is the description of my J2EE component</description>
  <display-name>This is the display name of my J2EE component</display-name>
  <servlet-name>ServletPrueba</servlet-name>
  <servlet-class>es.test.ServletPrueba</servlet-class>
</servlet>
```

Como puede verse además, se pueden configurar otros parámetros como una descripción y un nombre a mostrar. Pero los obligatorios son **servlet-name** y **servlet-class**.

Una vez tenemos declarado el Servlet el siguiente paso es configurar la URL desde la que será accesible:

```
<servlet-mapping>
  <servlet-name>ServletPrueba</servlet-name>
  <url-pattern>/servlet/ServletPrueba</url-pattern>
</servlet-mapping>
```

## 2.6 El contenedor Ciclo de vida

Los servlets se ejecutan dentro de la plataforma del servidor web (contenedor web). El contenedor web es responsable de la inicialización, invocación y destrucción del servlet.

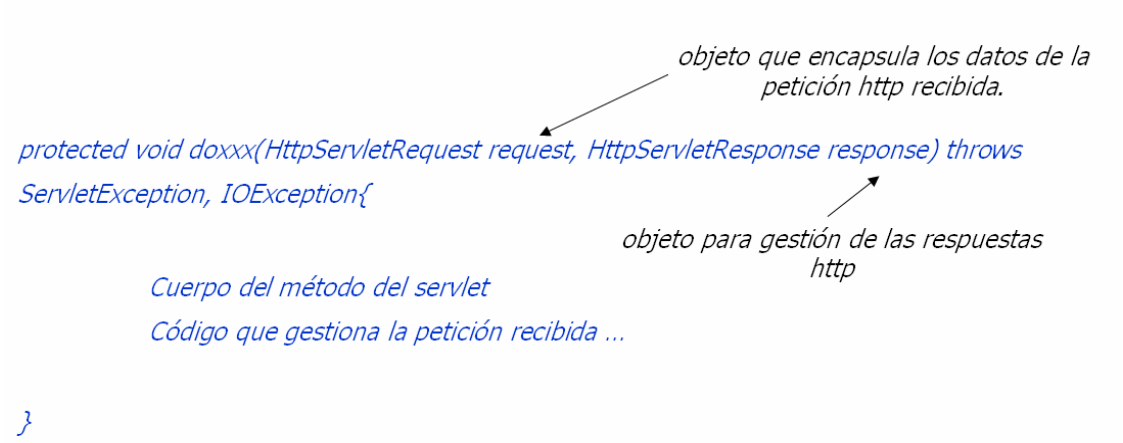
El contenedor web se comunica con el servlet a través de la interfaz ***javax.servlet.Servlet***.

- **init():** Es invocado cuando el servlet se carga por primera vez.
- **service():** Es invocado cada vez se realiza una petición desde el cliente. Recibe como parámetro un objeto ServletRequest con los datos de la petición y un objeto SevletResponse para generar la respuesta.
- **destroy():** Es invocado antes de que el servlet sea descargado.

## 2.7 Manejo de peticiones y respuestas

Los servlets reciben peticiones por parte de clientes http y dichas peticiones (en función del tipo de la misma (Get o Post) son gestionadas por el método *doGet()* o *doPost()* del servlet requerido) . Estos métodos son los encargados de llevar a cabo las tareas pertinentes para atender la petición [9-15].

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239




### 2.7.1 Peticiones

Todas las peticiones que se hacen al servlet van encapsuladas en el objeto HttpServletRequest. Este objeto lleva la información de dichas peticiones tanto los datos que se quieren enviar al Servlet como otros datos informativos del cliente.
HttpServletRequest es usado dentro de los métodos doXXX para acceder a la información de la petición. Se usa el siguiente método:

- **getParameter(String)**: Devuelve el valor del argumento que se le pasa como parámetro

Ejemplo:

```
Si se envía al servlet el parámetro nombre, la forma de recuperar este valor del request sería la siguiente:
        String nombre = request.getParameter("nombre");
```

En ocasiones, es necesario introducir valores en el request de manera que estos datos sean tratados en otros destinos.
Por ejemplo, guardar en el request los datos de un usuario (nombre, apellidos,…). Para ello, es necesario utilizar el método:

- **setAttribute(String, Object)**: del objeto *HttpServletRequest*. Mediante este método es posible introducir parámetros (atributos) en el request de una forma bastante sencilla.

Ejemplo (guardar en el request):

```
//Introducir los valores dentro del request
        request.setAttribute("nombre", nombre);
        request.setAttribute("apellidos", apellidos);
```

Si lo que se desea es obtener un atributo del request el método a usar es:

- **getAttribute(String)**: Mediante este método es possible recuperar el valor de un atributo.

Ejemplo (obtener del request):

```
//Obtener valores almacenados en el request
        request.getAttribute("nombre");
        request.getAttribute("apellidos");
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

### 2.7.2 Respuestas

Se encuentran encapsuladas en el objeto HttpServletResponse que proporciona la forma de devolver datos al usuario. Para ello se utiliza uno de los siguientes métodos:

- **getWriter():** Devuelve un objeto PrintWriter que permite enviar datos al usuario en formato texto.
- **getOutputStream():** Devuelve un objeto ServletOutputStream que permite enviar datos en formato binario.

### 2.7.3 Redirección

Mediante los objetos HttpServletRequest y HttpServletResponse, es posible llevar a cabo redireccionamientos desde un servlet a otro servlet o bien a una jsp o página html.

**Uso de HttpServletRequest**

HttpServletRequest se utiliza cuando además del redireccionamiento interesa reenviar el objeto HttpServletRequest y HttpServletResponse (por ejemplo para que en esa redirección se puedan mantener atributos introducidos en el HttpServletRequest).

El objeto HttpServletRequest proporciona un método para la obtención de la dirección a la que redireccionar:

- **getRequestDispatcher(String):** el cual devuelve un objeto del tipo *RequestDispatcher*.

El objeto obtenido proporciona el método:

- **forward(HttpServletRequest, HttpServletResponse)** que permite llevar a cabo el redireccionamieto. Como puede observarse, el método recibe como parámetro los objetos httpServletRequest y httpServletResponse puesto que serán reenviados al destino.

Ejemplo:

```
protected void doGet(HttpServletRequest request,   HttpServletResponse response)                 throws ServletException,IOException {
        //Obtener el objeto RequestDispatcher
        RequestDispatcher rd = request.getRequestDispatcher("destino.jsp");
        //Efectuar el reenvío
        rd.forward(request, response);
}
```

En el ejemplo, cuando el servlet reciba una petición tipo get se va a redireccionar a la página "destino.jsp" y además la página recibe los objetos request y response.

**Uso de HttpServletResponse**

Se utiliza cuando se desea realizar un simple redireccionamiento sin necesidad de pasarle request ni response.

El objeto HttpServletResponse proporciona un método para redireccionar hacia donde se desee:

- **sendRedirect(String)**: que redirecciona hacia la página indicada.

Ejemplo:

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

```
protected void doGet(HttpServletRequest request,   HttpServletResponse response)
                                throws ServletException,IOException {
        //Efectuar el reenvío
        response.sendRedirect("destino.jsp");
}
```

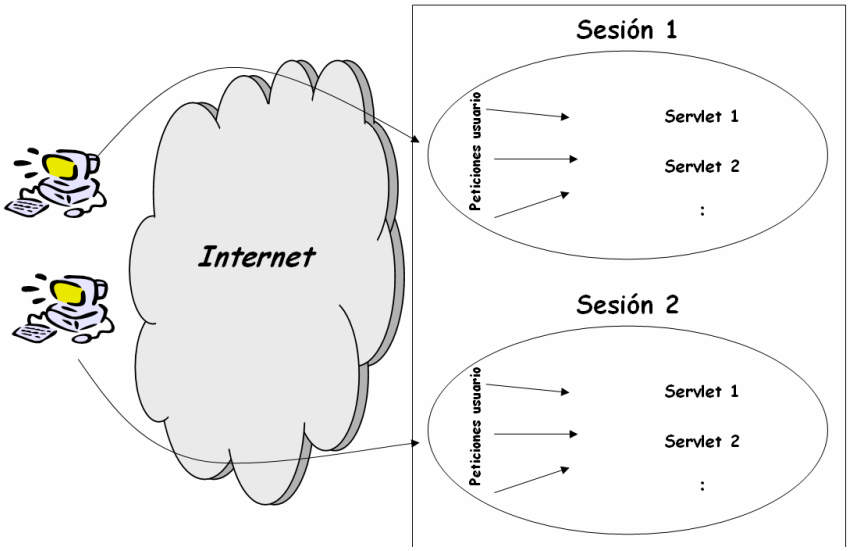
En el ejemplo, cuando el servlet reciba una petición tipo **get**, se efectúa una redirección a la página "**destino.jsp**".

## 2.8 Manejo de sesiones

Dentro del entorno web aparece el término sesión. Se podría definir una sesión como un contexto lógico-temporal dentro del cual el usuario de una aplicación web lleva a cabo un proceso de navegación. Este contexto contiene información que puede ser utilizada para diversos fines.

Imaginemos dos usuarios (usuario1 y usuario2). El usuario1 accede a la página de sun. En ese mismo instante, el servidor genera una sesión para el usuario1. El usuario1 irá navegando por las distintas páginas del site y lo hará bajo la misma sesión. Por otro lado, el usuario2 accede también a la página de sun generándose una nueva sesión que en ningún caso coincidirá con las sesiones activas anteriores. De esta forma, ambos usuarios están trabajando bajo distintas sesiones (contextos lógico- temporales independientes).

En java, la clase que permite actuar con la sesión es la **HttpSession**.




### 2.8.1 Interfaz HttpSession

La interfaz HttpSession proporciona métodos que almacenan y recuperan propiedades de sesión estándar y datos de la aplicación.
Para obtener el objeto de sesión se hará siempre desde HttpServletRequest mediante el método:


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **getSession:** dispone de un argumento boolean que si se pone a true, generará una nueva sesión, si es false se devolverá la sesión actual.

Para asociar un objeto a una sesión se utiliza el método:

- **setAttribute(String, Object)**: que proporciona un nombre para el objeto y su valor:

```
HttpSession sesion = request.getSession();
sesion.setAttribute("nombre", new String("Jorge"));
```

Para obtener el valor de un dato de la sesión se utiliza el método:

- **getAttribute(String)**

```
HttpSession sesion = request.getSession();
String nombre = (String) sesion.getAttribute("nombre");
```

# 3 JSP (Java Server Pages)

## 3.1 Introducción

Java Server Pages es una tecnología orientada a crear páginas web con programación en Java. Las páginas JSP están compuestas de código HTML/XML mezclado con etiquetas especiales para programar scripts de servidor en sintaxis Java. La extensión de fichero de una página JSP es ".jsp" en vez de ".html" o ".htm", y eso le indica al servidor que esta página requiere un tratamiento especial [16-23].

Las páginas JSP permiten separar la parte dinámica de la estática en una página web. Cada página es compilada automáticamente en un servlet por el motor JSP la primera vez que se accede a ella.

Existen varios objetos implícitos: request, response, out, session, application, config, pageContext, page y exception.

Beneficios que aporta JSP:

- Separación de la presentación de la implementación o JavaBeans que gestionan la información.
- Las frecuentes modificaciones de una página se realizan más eficientemente.
- Los JavaBeans utilizados en páginas .jsp pueden ser utilizados en servlets, applets o aplicaciones Java.
- Teóricamente se pueden desarrollar por separado.
- Utilización de procesos ligeros (hilos Java) para el manejo de las peticiones.

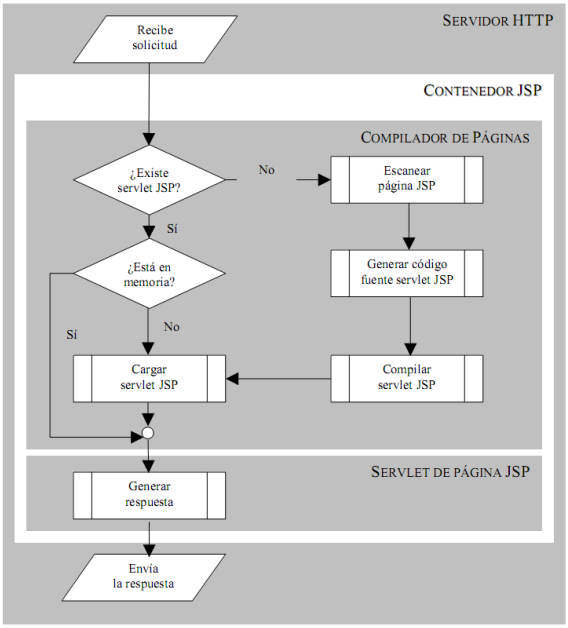
ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

## 3.2 Modo de trabajo de JSP

El componente más importante es un servlet especial denominado compilador de páginas. Este servlet junto con sus clases Java asociadas, se conoce con el nombre de contenedor JSP. El contenedor está configurado para llamar al compilador de páginas para todas las peticiones que coincidan con una página .jsp. Su misión es la de compilar cada página .jsp en un servlet cuya finalidad es la de generar el contenido dinámico especificado por el documento .jsp original.

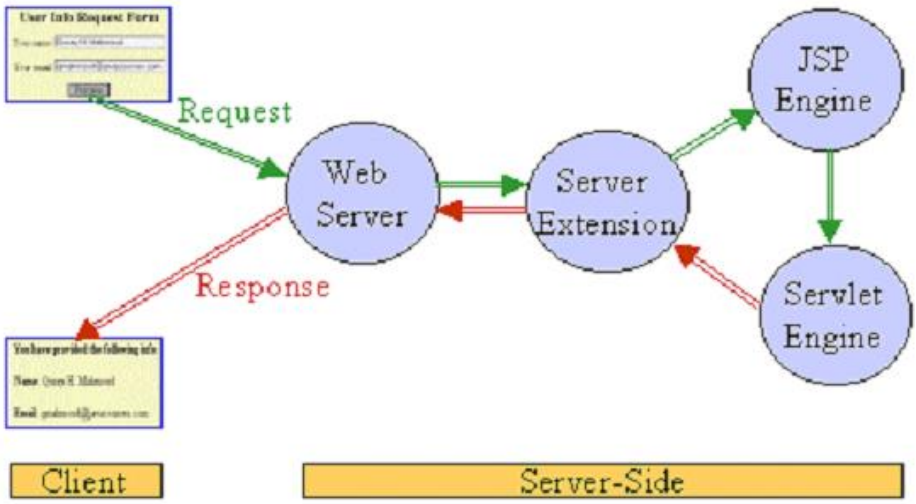
El compilador de páginas escanea el documento en busca de etiquetas JSP, generando el código Java correspondiente para cada una de ellas. Las etiquetas estáticas HTML son convertidas a Strings de Java. Las etiquetas que hacen referencia a JavaBeans son traducidas en los correspondientes objetos y llamadas a métodos. Una vez que el código del servlet ha sido construido (se ha codificado su método *service()*), el compilador de páginas llama al compilador de Java para compilar el código fuente y añade el fichero de bytecodes resultante al directorio apropiado del contenedor JSP.

Una vez que el servlet correspondiente a la página .jsp ha sido generado, el compilador de páginas invoca al nuevo servlet para generar la respuesta al cliente. Mientras que el código de la página .jsp no se altere, todas las referencias a la página ejecutarán el mismo servlet. Esto supone una cierta demora en la primera referencia a la página, aunque existen mecanismos en JSP para precompilar la página .jsp antes de que se haya producido la primera petición.




Cuando se realiza una llamada a una página JSP, será compilada (por el motor JSP) en un Servlet Java. En este momento el Servlet es manejado por el motor Servlet que cargará la clase Servlet (usando un cargador de clases) y lo ejecutará para crear HTML dinámico y enviarlo al navegador.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc




## 3.3 Sintaxis

### 3.3.1 Introducción

JSP proporciona 4 tipos principales de etiquetas:

- **Directivas**: conjunto de etiquetas que contienen instrucciones para el contenedor JSP con información acerca de la página en la que se encuentran. No producen ninguna salida visible, pero afectan a las propiedades globales de la página JSP influyendo en la generación del servlet correspondiente.
- **Elementos de Script**: se utilizan para encapsular código Java que será insertado en el servlet correspondiente a la página .jsp.
- **Comentarios**: se utilizan para añadir comentarios a la página .jsp. JSP soporta múltiples estilos de comentarios, incluyendo aquellos que hacen que los comentarios aparezcan en la página HTML generada como respuesta a la petición del cliente.
- **Acciones**: soportan diferentes comportamientos. Pueden transferir el control entre páginas, applets e interaccionar con componentes JavaBeans del lado del servidor. Además se pueden programar etiquetas JSP personalizadas para extender las funcionalidades del lenguaje en forma de etiquetas de este tipo.

### 3.3.2 Directivas

Permiten controlar la estructura general de la jsp. Las directivas se encuentran delimitadas por: <%@ y %>
Sintaxis: <%@ page atributo1=”valor1” atributo2=”valor2” atributo3=... %>

**Directiva page**

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239

| Atributo | Valor | Valor por defecto |
|---|---|---|
| info | Cadena de texto. | "" |
| language | Nombre del lenguaje de script. | "java" |
| contentType | Tipo MIME y conjunto de caracteres. | "text/html"<br>"ISO-8859-1" |
| extends | Nombre de clase. | Ninguno |
| import | Nombres de clases y/o paquetes. | java.lang,<br>java.servlet,<br>javax.servlet.http,<br>javax.servlet.jsp |
| session | Booleano. | "true" |
| buffer | Tamaño del buffer o "none". | "8kb" |
| autoFlush | Booleano. | "true" |
| isThreadSafe | Booleano. | "true" |
| errorPage | URL local. | Ninguno |
| isErrorPage | Booleano. | "false" |

Se pueden incluir múltiples directivas de este tipo en una página JSP. No se puede repetir el uso de un atributo en la misma directiva ni en otras de la misma página excepto para import. Los atributos que no sean reconocidos generarán un error en tiempo de ejecución (el nombre de los atributos es sensible a mayúsculas y minúsculas).

**Directiva include**

Permite incluir el contenido de otra página HTML o JSP. El fichero es identificado mediante una URL y la directiva tiene el efecto de remplazarse a sí misma con el contenido del archivo especificado. Puede ser utilizada para incluir cabeceras y pies de páginas comunes a un desarrollo dado.
Sintaxis: <%@ include file="localURL" %>

```
<%@ include file="pagina.html" %>
```

**Directiva taglib**

Esta directiva se utiliza para notificar al contenedor JSP que la página utilizará una o más librerías de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas personalizadas, que pueden ser utilizadas para extender la funcionalidad básica de JSP.
Sintaxis: <%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>

```
<%@ taglib uri="miTagLib.tld" prefix="prefijo" %>
```

### 3.3.3 Elementos de script

Las etiquetas de este tipo permiten encapsular código Java en una página .jsp. JSP proporciona tres tipos diferentes de etiquetas de script: declaraciones, scriptlets y expresiones.

- **Declaraciones:** permiten definir variables y métodos para una página.
  - Sintaxis: **<%! declaraciones %>**

```
<%! private int contador = 0;  %>
```

- **Scriptlets:** son bloques de código preparados para ejecutarse.
  - Sintaxis: **<% scriptlet %>**

```
<% for (long x = 0; x <=20; ++x) {%>
<tr><td><%= x %></td><td><%= factorial (x) %></td></tr>
<% } %>
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **Expresiones:** son líneas simples de código cuya evaluación es insertada en la salida de la página en el lugar de la expresión original.
    - Sintaxis: **<%= expresión %>**

```
<%= (horas < 12)? “AM” : “PM” %>
```

### 3.3.4 Comentarios

En una JSP, pueden existir tres tipos de comentarios, aunque se dividen en dos categorías:

- Los que son transmitidos al cliente como parte de la respuesta generada por el servidor.
    - **Comentario de contenido**: **<!-- comentario -->** Comentario Standard html, transmitido al cliente.
- Los que sólo son visibles en el código fuente de la página JSP. (típicamente usaremos estos)
    - **Comentario de JSP**: **<%-- comentario --%>** Comentario no transmitido al cliente.
    - **Comentarios del lenguaje de script**: **<% /* comentario */ %>** Comentario no transmitido al cliente.

### 3.3.5 Acciones

Las acciones permiten la transferencia de control entre páginas, permiten a las páginas .jsp interaccionar con componentes JavaBeans residentes en el servidor, crear y usar librerías de etiquetas personalizadas [24-30].

**Acción forward**

Permite reenviar la petición a otra página. Se usa para transferir el control de una página JSP a otra localización. Cualquier código generado hasta el momento se descarta, y el procesado de la petición se inicia en la nueva página. El navegador cliente sigue mostrando la URL de la página .jsp inicial.

Sintaxis: **<jsp:forward page=”localURL” />**

```
Ejemplo básico:        <jsp:forward page=“pagina.html”/>
Página configurable:   <jsp:forward page=“<%=variable%>”/>
Paso de parámetros:    <jsp:forward page=”factorial.jsp” >
                                <jsp:param name=”numero” value=”25” />
                       </jsp:forward>
```

**Acción include**

Proporciona una forma sencilla de incluir contenido generado por otro documento local en la salida de la página actual. En contraposición con la etiqueta forward, include transfiere el control temporalmente:

Sintaxis: **<jsp:include page=”localURL” flush=”true” />**

```
<jsp:include page=”localURL” flush=”true”
                        <jsp:param name=”nombreParametro1” value=”valorParametro1” />
                        ...
                        <jsp:param name=”nombreParametroN” value=”valorParametroN” />
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

```
</jsp:include>
```

**Acción useBean**

Permite cargar y utilizar un componente JavaBean en la JSP.

```
<jsp:useBean id="MiBean" class="package.class"/>
```

Una vez que tenemos un bean, podemos leer o modificar sus propiedades mediante <jsp:getProperty>, <jsp:setProperty> o usando un Scriptlet. Del siguiente modo, el contenido sólo es ejecutado cuando se crea por primera vez el Bean.

```
<jsp:useBean id="MiBean"class="package.class">
        <jsp:setProperty name="MiBean" property="p" value="x"/>
</jsp:useBean>
```

Los posibles atributos de <jsp:useBean> son:

- **id**="name"
- **scope**="page|request|session|application"
- **class**="package.class"
- **type**="package.class"
- **beanName**="package.class"

**Acción setProperty**

Permite asignar un valor a una propiedad de un Bean.

```
<jsp:useBean id="MiBean" scope="session" class="JSP.MiBean" />
<jsp:setProperty name="MiBean" property="mensaje" value="valor" />
```

Los posibles atributos de <jsp:setProperty> son:

- **name**="beanName"
- **property**="propertyName|*"
- **param**="parameterName"
- **value**="val"

**Acción getProperty**

Permite recuperar el valor de una propiedad de un Bean, lo convierte a String y lo visualiza.

```
<jsp:useBean id="MiBean" scope="session" class="JSP.MiBean" />
<jsp:getProperty name="MiBean" property="mensaje" />
```

Los posibles atributos de <jsp:getProperty> son:

- **name="beanName"**
- **property="propertyName|*"**
- **param="parameterName"**
- **value="val"**

**Etiqueta param**

Permite pasar parámetros a las etiquetas **<jsp:include>**, **<jsp:forward>**, **<jsp:plugin>**

```
<jsp:forward page="pagina.html">
```



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
                    <jsp:param name="param1" value="valor1"/>
                    <jsp:param name="param2" value="valor2"/>
                </jsp:forward>
```

**Etiqueta params**

Permite agrupar varias etiquetas <jsp:param>. Sólo es posible anidarla dentro de la etiqueta <jsp:plugin>

```
        <jsp:plugin type="applet" code="Clock2.class"
        codebase="/examples/jsp/plugin/applet" jreversion="1.2"
        width="160" height="150" >
                    <jsp:params>
                        <jsp:param name="param1" value="valor1"/>
                        <jsp:param name="param2" value="valor2"/>
                    </jsp:params>
                </jsp:plugin>
```

**Acción pluging**

Permite insertar un elemento OBJECT o EMBED específico del navegador para decirle al navegador que debería ejecutar un applet usando el Plugin Java en vez de la máquina virtual del navegador.

```
        <jsp:plugin type="applet" code="Clock2.class"
        codebase="/examples/jsp/plgin/applet" jreversion="1.2" width="160"                    height="150" >
                    <jsp:fallback>
                        Plugin tag OBJECT or EMBED not supported by browser
                </jsp:fallback>
        </jsp:plugin>
```

**Etiqueta fallback**

Si no se puede ejecutar la etiqueta <jsp:plugin>, se ejecuta la etiqueta <jsp:fallback>. Sólo es posible anidarla dentro de la etiqueta <jsp:plugin>

### 3.3.6 Ejemplo: uso de javabean desde jsp

Ejemplo del JavaBean:

```
package JSP.Bean;

import java.beans.*;

public class MiBean implements Serializable {

        private String mensaje = "mensaje por defecto";
        public String getMensaje() {   return mensaje;   }
        public void setMensaje(String valor) {  mensaje = valor;  }

}
```

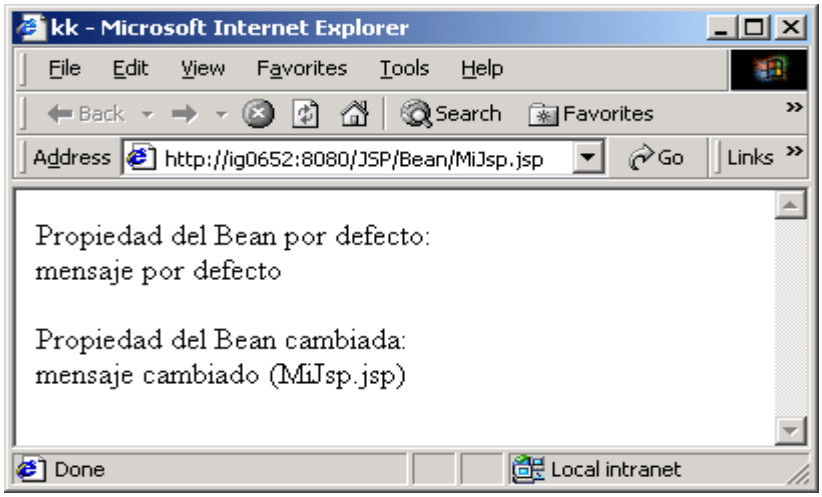
Ejemplo de la JSP:

```
<html><head><title>Mi título</title></head><body>

        <jsp:useBean id="miBean" scope="session" class="JSP.Bean.MiBean"/>
                Propiedad del Bean por defecto:<br>

        <jsp:getProperty name="miBean" property="mensaje" />
                <br><br>Propiedad del Bean cambiada:<br>

        <jsp:setProperty name="miBean" property="mensaje"
                        value="mensaje cambiado (MiJsp.jsp)" />

        <jsp:getProperty name="miBean" property="mensaje" />

</body></html>
```

Resultado del ejemplo:

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239




## 3.4 Objetos implícitos en las JSP

### 3.4.1 Objetos implícitos

El contenedor JSP exporta un número de objetos internos para su uso desde las páginas .jsp. Estos objetos son asignados automáticamente a variables que pueden ser accedidas desde elementos de script de JSP. A continuación vemos los objetos disponibles y la API a la que pertenecen de modo que podamos ver sus propiedades y métodos.

| Objeto | Descripción | Tipo | Ámbito |
|---|---|---|---|
| application | Representa el contexto en el que se ejecutan las JSP | javax.servlet.ServletContext | Application |
| session | Representa la sesión de un cliente | javax.servlet.http.HttpSession | Session |
| request | Extrae datos del cliente | javax.servlet.http.HttpServletRequest | Request |
| response | Envía datos al cliente | javax.servlet.http.HttpServletResponse | Page |
| out | Envía la salida al cliente | javax.servlet.jsp.JspWriter | Page |
| exception | Información del último error | java.lang.Throwable | Page |
| config | Representa la configuración del Servlet | javax.servlet.ServletConfig | Page |
| pageContext | Es de donde se obtienen algunos objetos implícitos | javax.servlet.jsp.PageContext | Page |
| page | Representa la propia página JSP | java.lang.Object | Page |

Los objetos proporcionados por JSP se clasifican en cuatro categorías:

1. Objetos relacionados con el servlet correspondiente a la página .jsp.
2. Objetos relacionados con la entrada y salida de la página.



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

3. Objetos que proporcionan información sobre el contexto.
4. Objetos para manejo de errores.

Estos objetos tienen la funcionalidad común de establecer y recuperar valores de atributos como mecanismo de intercambio de información.

| Método | Descripción |
|---|---|
| *setAttribute*(key, value) | Asocia un atributo con su valor correspondiente. |
| *getAttributeNames*() | Retorna los nombres de todos los atributos asociados con la sesión. |
| *getAttribute*(key) | Retorna el valor del atributo asociado con key. |
| *removeAttribute*(key) | Borra el atributo asociado con key. |

La mayoría de los objetos implícitos provienen de la clase abstracta: javax.servlet.jsp.PageContext.
Cuando se compila la JSP, el servlet que se obtiene, contiene la creación de estos objetos implícitos dentro del método _jspService:

```
public void _jspService(HttpServletRequest request, HttpServletResponse  response)
                        throws IOException, ServletException {
                                PageContext pageContext = null;
                                HttpSession session = null;
                                ServletContext application = null;
                                ServletConfig config = null;
                                JspWriter out = null;
                                Object page = this;
                                 pageContext = _jspxFactory.getPageContext(this, request, response,"", true, 8192, true);
                                 application = pageContext.getServletContext();
                                 config = pageContext.getServletConfig();
                                 session = pageContext.getSession();
                                 out = pageContext.getOut();
                         }
```

Los objetos implícitos y también los Bean pueden tener diferentes ámbitos:

- **Page**: el objeto existe sólo en una página.
- **Request**: el objeto existe en diferentes páginas cuando la petición es pasada de una a otra.
- **Session**: el objeto existe a lo largo de toda la sesión del cliente, cada cliente tiene el suyo.
- **Application**: el objeto existe siempre y todos los clientes lo comparten.

**Eventos en las JSP:**

- Evento **jspInit()**, Salta cuando la JSP es cargada
- Evento **jspDestroy()**: Salta cuando la JSP es destruida

Estos eventos son a nivel de cada página JSP

### 3.4.2 Objeto request

Permite extraer información del cliente. Extrae los parámetros de una página y los datos de un formulario con el método getParameter():

```
String p = request.getParameter("param1");
```

También extrae los datos de una cookie mediante el método getCookies():

```
Cookie[] cookies = request.getCookies();
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

Algunos métodos del objeto request:

- ***getCharacterEncoding():*** devuelve el juego de caracteres del cliente
- ***getContentLength():*** devuelve la longitud de la petición
- ***getContentType():*** devuelve el tipo MIME de la petición
- ***getParameter("param1"):*** devuelve el valor de un parámetro
- ***getParameterNames():*** devuelve los parámetros
- ***getParameterValues("select1"):*** devuelve los valores de par.
- ***getQueryString():*** devuelve los parámetros de la página
- ***getRemoteAddr():*** devuelve la IP del cliente
- ***getRemoteHost():*** devuelve el nombre del host del cliente
- ***getServerName():*** devuelve el nombre del servidor
- ***getCookies():*** obtiene la cookie del cliente
- ***getRequestURI():*** url petición
- ***getHeader("User-Agent"):*** navegador cliente

Cuando se mandan datos a través de un hipervínculo se recogen con el objeto request.

Para enviar datos desde una página HTML a una página JSP a través de un hipervínculo se debe hacer lo siguiente:

- Primero, en la página HTML se crea el hipervínculo:

```
<a href="pag.jsp?codigo=123&nombre=luis">Ir a pag.jsp</a>
```

- Segundo, en la página JSP se recogen los datos:

```
<%
        String cod, nom;
        cod = request.getParameter("codigo");
        nom = request.getParameter("nombre");
%>
```

**Envío de un formulario a un bean**

La información que el usuario introduce en el formulario se almacena en el objeto request. Al enviar el formulario a una página JSP, los datos se recogen con el método getParameter() del objeto request:

```
<%= request.getParameter("codigo") %>
```

Otros métodos útiles para el manejo de formularios:

- **getRequest():** devuelve el objeto request
- **getParameterNames():** devuelve los nombre de los parámetros
- **getParameterValues():** devuelve los valores de los parámetros


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **getParameter():** devuelve el valor de un parámetro dado el nombre del control HTML.

Los datos del formulario se pueden enviar además a otra página JSP, a un componente JavaBean o a un Servlet. Generalmente los datos se envían al Bean, donde se encuentra la lógica de negocio. Para ello los pasos a realizar son:

- Crear un formulario en una página JSP
- Escribir un componente JavaBean haciendo coincidir los nombres de las propiedades con los nombres de los controles HTML
- Añadir una etiqueta **<jsp:useBean>** para crear y usar el Bean
- Añadir una etiqueta **<jsp:setProperty>** para indicar que las propiedades del Bean coinciden con los nombres de los controles HTML
- Añadir etiquetas **<jsp:getProperty>** para recuperar los datos desde el Bean.
- En la página JSP, se puede omitir el atributo ACTION para que los datos se envíen al Bean especificado en **<jsp:useBean>**

Ejemplo del JavaBean:

```
package JSP.Formularios;
import java.beans.*;

public class FormBean  {

	private String codigo = "", nombre = "";
	public String getCodigo() {  return codigo;  }
	public void setCodigo(String valor) { codigo = valor; }
	public String getNombre() { return nombre; }
	public void setNombre(String valor) { nombre = valor; }
	public String cliente() { return codigo + " " + nombre; }
}
```

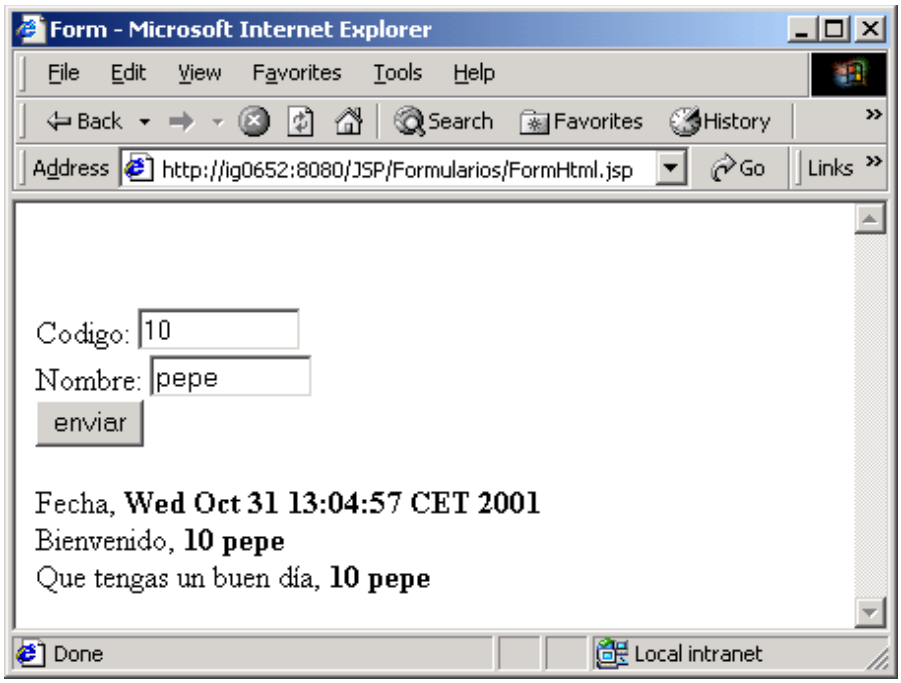
Ejemplo de la JSP:

```
<html><head><title>Form</title></head><body>

<jsp:useBean id="FormBean" scope="session"  class="JSP.Formularios.FormBean" />

<jsp:setProperty name="FormBean" property="*" />

	<form method="post">
		Codigo:
			<input type="text" name="codigo" size="10"
		value="<%= request.getParameter("codigo")==null ? "" :
	request.getParameter("codigo")%>"><br>
		Nombre:
			<input type="text" name="nombre" size="10"
			value="<%= request.getParameter("nombre")==null ? "" :
			request.getParameter("nombre")%>"><br>
				<input type="submit" value="enviar"><br>
	</form>

<% if (request.getParameter("codigo") != null &&	!request.getParameter("codigo").equals("")) { %>

		Fecha, <b><%= new java.util.Date() %></b>
		<br>Bienvenido, <b>

		<jsp:getProperty name="FormBean" property="codigo" />
		<jsp:getProperty name="FormBean" property="nombre" />
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

```
                </b><br>
                Que tengas un buen día,
                <%= "<b>" + FormBean.getCodigo() + " " +       FormBean.getNombre() + "<b>" } %>

</body>
</html>
```

Resultado del ejemplo:




### 3.4.3 Objeto response

Permite enviar información al cliente. También envía una cookie al cliente mediante el método addCookie():

***response.addCookie(miCookie);***

Algunos métodos del objeto response:

- ***sendRedirect****("Request.jsp"):* redireciona a otra página al cliente
- ***sendError****(401):* envía un error al cliente
- ***setStatus****(401):* envía un código de estado al cliente
- ***addCookie****(miCookie):* envía una cookie al cliente

### 3.4.4 Objeto out

Permite enviar la salida al cliente. La salida, por defecto, es almacenada en un buffer de 8 kb:

```
<%@ page autoFlush="true" buffer="8"%>
```

Esto se puede cambiar con la directiva page:


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
<%@ page autoFlush=“false"%>
```

Donde se le indica que la salida no se almacene en el buffer.

Algunos métodos del objeto out:

- ***clear():*** limpia el contenido del buffer
- ***flush():*** escribe el contenido del buffer
- ***getBufferSize():*** devuelve el tamaño del buffer
- ***getRemaining():*** devuelve el espacio no usado del buffer
- ***isAutoFlush():*** indica si se utiliza el buffer o no
- ***println(“mensaje"):*** visualiza un mensaje con salto de línea
- ***print(“mensaje”):*** visualiza un mensaje sin salto de línea
- ***close():*** cierra el buffer visualizando el contenido

### 3.4.5 Objeto exception

La gestión de excepciones en las JSP se puede realizar de la siguiente manera:
Escribir un Bean, EJB, Servlet u otro componente para que lance excepciones en determinadas condiciones. También si se produce un error en una JSP.

```
public Object metodo() throws NullPointerException { ... }
```

Escribir una JSP que será la página de error usando la directiva page con isErrorPage=“true”. Es decir que si ocurre un error en otras JSP que usen el componente, se ejecutará esta página

```
<%@ page isErrorPage=“true” import=“java.util.*” %>
```

En la página de error usar el objeto exception para obtener información de la excepción

```
<%= exception.toString() %>
<% exception.printStackTrace(); %>
```

En las JSPs que usan el componente, indicar qué página se ejecutará si se producen algún error con la directiva page estableciendo errorPage a la página de error

```
<%@ page isThreadSafe=“false” import=“java.util.*” errorPage=“error.jsp” %>
```

### 3.4.6 Objeto session

Cuando un usuario se conecta a un sitio web el servidor le crea una sesión que es independiente de las sesiones de otros usuarios [31-37].
En términos de programación una sesión es un objeto (objeto session) en el que se puede almacenar información propia de cada usuario. El objeto sesión existe mientras el usuario está visitando la web, lo que permite compartir los datos que hay dentro del objeto sesión entre diferentes páginas JSP. Cada usuario tiene su propio identificador de sesión llamado session Id.
Un ejemplo típico es el carrito de la compra; lo que vamos comprando se va acumulando en las páginas, y cada carrito de usuario es diferente.
Para guardar un objeto dentro del objeto session usar el método setAttribute():

```
String s = “hola”;

session.setAttribute(“nomObj”, s);
```

Para recuperar un objeto dentro del objeto session usar el método getAttribute():

```
String s = (String)session.getAttribute(“nomObj”);
```

Algunos métodos del objeto session:

- ***getCreationTime():*** cuándo fue creada la sesión

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

- ***getId():*** id. de sesión
- ***getLastAccessedTime():*** último acceso del cliente
- ***setMaxInactiveInterval(2000):*** máxima duración de la sesión
- ***getMaxInactiveInterval():*** obtiene duración se la sesión
- ***isNew():*** devuelve si es nueva la sesión
- ***setAttribute("obj1", "hola"):*** guarda un objeto en el obj.sesión
- ***getAttribute("obj1"):*** obtiene un objeto del obj. sesión
- ***getAttributesNames():*** obtiene los objetos del obj. sesión
- ***removeAttribute("obj1"):*** borra un objeto del obj. sesión
- ***invalidate():*** elimina una sesión

#### 3.4.7 Objeto application

Todos los clientes comparten el mismo objeto application donde se pueden almacenar otros objetos. Representa el contexto en el que se ejecuta la JSP.
Para guardar un objeto usar el método setAttribute():

```
String s1 = "hola";

application.setAttribute("valor", s1);
```

Para recuperar un objeto utilizar el método getAttribute():

```
String s2 = (String)application.getAttribute("valor");

out.println(s2 + "<br>");
```

Algunos métodos del objeto application:

- ***getMajorVersion():*** versión superior del Servlet API
- ***getMinorVersion():*** versión inferior del Servlet API
- ***getRealPath("/"):*** path físico de un path virtual
- ***getResource("/"):*** da el URL de una path
- ***getServerInfo():*** inf. del contenedor del Servlets
- ***setAttribute("valor", s1):*** guarda un objeto en application
- ***getAttribute("valor"):*** recupera un objeto

# 4 Struts

## 4.1 Struts Framework - Introducción

### 4.1.1 Introducción al “Modelo Vista Controlador” (MVC)

MVC o Model View Controller es un patrón de diseño aportado originariamente por el lenguaje SmallTalk a la Ingeniería del Software. El paradigma MVC consiste en dividir las aplicaciones en tres partes:

- El **Controlador** es el encargado de redirigir o asignar una aplicación (un Modelo) a cada petición; el Controlador debe poseer, de algún modo, un "mapa" de correspondencias entre peticiones y respuestas (aplicaciones o Modelo) que se les asignan.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

El Controlador es el nexo entre el Modelo y la Vista. Define el flujo de la aplicación. Convierte las interacciones del usuario en acciones que ejecutará en el Modelo. Los diferentes tipos de clientes requerirán diferentes tipos de Controladores. En Struts las acciones (Action), los ActionServlet y los objetos de configuración (creados desde el struts-config-xml) actuarán como Controlador. El Controlador acepta los eventos de usuario que vengan desde la Vista, que son peticiones de cargas de página (HTTP GET) y de envíos de formularios (HTTP POST) y será cuestión del Controlador seleccionar la siguiente vista.

- El **Modelo** lo componen los datos y las reglas de negocio de la aplicación. El Modelo sería la aplicación que responde a una petición, es la lógica de negocio a fin de cuentas, quien se va a encargar de modificar los datos de la aplicación. Debe ser independiente de Struts o de la API JSP/Servlet. El framework de Struts no proporciona clases propias para el Modelo deben desarrollarse para cada aplicación.

- La **Vista** es lo que el usuario final ve. La Vista muestra partes del Modelo al usuario. Los diferentes tipos de clientes (web, rich GUI, etc.) pueden tener diferentes Vistas. Nunca modificará el modelo de datos directamente. Una vez realizadas las operaciones necesarias el flujo vuelve al Controlador y éste devuelve los resultados a una Vista asignada. Struts no especifica que se deba usar una tecnología concreta para la Vista. De todas formas, proporciona algunas etiquetas JSP personalizadas para facilitar el desarrollo de la misma.

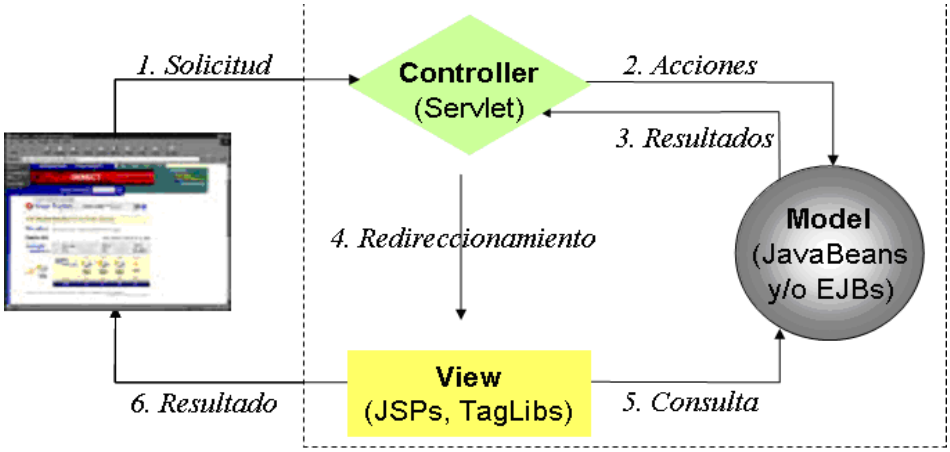
### 4.1.2 MVC vs. Modelos Clásicos

Vemos las diferencias que supone el modelo con los modelos convencionales:

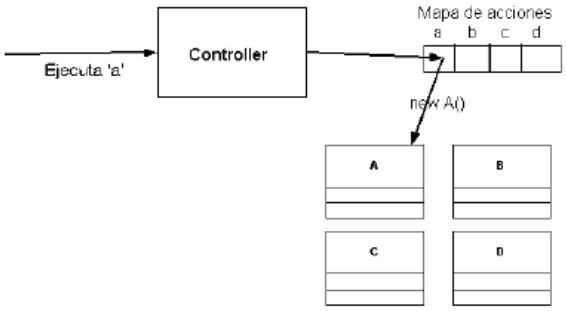

En el esquema más básico de programa, tenemos una entrada o parámetros que llegan (INPUT), se procesan y se muestra el resultado (OUTPUT).
En el caso del patrón MVC el procesamiento se lleva a cabo entre sus tres componentes. El Controlador recibe una orden y decide quién la lleva a cabo en el modelo. Una vez que el modelo (la lógica de negocio) termina, sus operaciones devuelven el flujo y vuelve al Controlador y éste envía el resultado a la capa de presentación.

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239




El Controller, en cierta forma, debe tener un registro de la relación entre órdenes que le pueden llegar y la lógica de negocio que le corresponde (es como una operadora de teléfono que recibe una petición y une dos líneas). En el siguiente gráfico se representa ese funcionamiento:




**¿Qué ventajas obtenemos de este modelo?**

Obviamente una separación total entre lógica de negocio y presentación. A esto se le pueden aplicar opciones como el multilenguaje, distintos diseños de presentación, etc., sin alterar la lógica de negocio.

La separación de capas como presentación - lógica de negocio - acceso a datos, es fundamental para el desarrollo de arquitecturas consistentes, reutilizables y más fácilmente mantenibles, lo que al final resulta en un ahorro de tiempo en desarrollo en posteriores proyectos.

### 4.1.3 ¿Qué es Struts?

Struts es un framework para aplicaciones web java que implementan el modelo MVC.

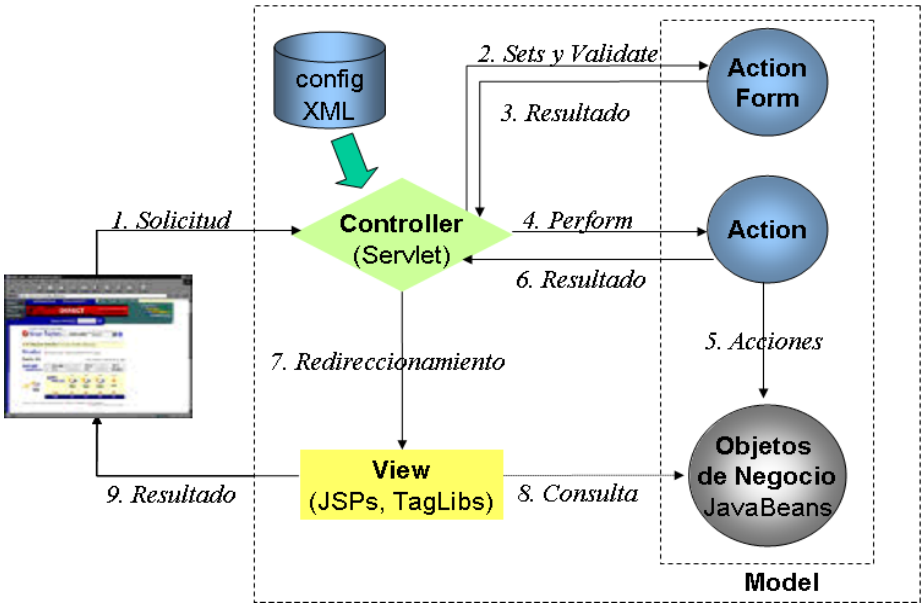
Un framework es la extensión de un lenguaje mediante una o más jerarquías de clases que implementan una funcionalidad y que (opcionalmente) pueden ser extendidas. El framework puede involucrar TagLibraries. (Librerías de tags de apoyo en la programación de las vistas).

Realmente provee un conjunto de clases y TAG-LIBS que conforman el Controlador, la integración con el Modelo (o lógica de negocio) y facilitan la construcción de Vistas.

Naturalmente, el Modelo o lógica de negocio es la parte que nos corresponde desarrollar. Por eso, Struts es una plataforma sobre la que montamos la lógica de negocio, y esta plataforma nos permite dividir la lógica de la presentación entre otras cosas.

La arquitectura del framework de Struts tiene bastante que ver con el MVC lógicamente. A continuación se muestra la arquitectura.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc




4.1.4 ¿Qué capacidades aporta el Framework Struts?

**Introducción general**

Evidentemente, como todo framework intenta, Struts simplifica notablemente la implementación de una arquitectura según el patrón MVC. Él mismo separa muy bien lo que es la gestión del workflow de la aplicación, del modelo de objetos de negocio y de la generación de interfaz.

El controlador ya se encuentra implementado por Struts, aunque si fuera necesario se puede heredar y ampliar o modificar, y el workflow de la aplicación se puede programar desde un archivo XML. Las acciones que se ejecutarán sobre el modelo de objetos de negocio se implementan basándose en clases predefinidas por el framework. La generación de interfaz se soporta mediante un conjunto de Tags predefinidos por Struts cuyo objetivo es evitar el uso de Scriplets (los trozos de código Java entre "<%" y "%>"), lo cual genera ventajas de mantenimiento [38-45].

**Logísticamente, separa claramente el desarrollo de interfaz del workflow y lógica de negocio permitiendo desarrollar ambas en paralelo o con personal especializado.**

También es evidente que potencia la reutilización, el soporte de múltiples interfaces de usuario (HTML, sHtml, Wml, Desktop applications, etc.) y de múltiples idiomas, localismos, etc.

**Control del flujo de la aplicación**

Como se ha mencionado al inicio, el framework, mediante el Controlador, permite gestionar el flujo de la aplicación. El cliente lanza una petición, la petición es capturada por el Controlador y en función de su configuración, el flujo va a un sitio u otro. Más adelante se verá esto con más detalle.

**Validación de formularios**

El framework permite además controlar y realizar la validación de formularios de manera que la robustez de la aplicación se incrementa al mismo tiempo que el proceso de actuación frente al envío de un formulario incorrecto se ve simplificado. Más adelante se entrará en profundidad.

**Internacionalización**

Debido al auge del entorno web, el acceso a las aplicaciones web se puede llevar a cabo desde cualquier parte del mundo. Por ello, este tipo de aplicaciones han de estar preparadas para mostrar los contenidos en el lenguaje del cliente que está accediendo a la aplicación. Inicialmente, esta situación se solventaba creando el contenido html en cada uno de los idiomas deseados. Es decir,

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

los proyectos se duplicaban, triplicaban,… para que estuvieran disponibles en los idiomas deseados.
Para evitar esta situación, se introducen los ficheros de propiedades (**.properties**). Dentro de estos ficheros de propiedades, el desarrollador introduce los contenidos en el idioma que desea, es decir, **se crea un fichero de propiedades para cada idioma**. Por tanto, una vez que acceda un usuario a la aplicación, tan solo queda determinar el idioma del cliente para seleccionar el fichero de propiedades correspondiente. Los contenidos de las páginas salen de estos ficheros de propiedades.
**De este modo, es posible desarrollar aplicaciones internacionalizadas sin necesidad de duplicar n veces las páginas de la aplicación.**
Struts introduce un sistema bastante sencillo para llevar a cabo el proceso de internacionalización. Más adelante se entrará en detalle.

**Tags de presentación**

Struts presenta una serie de bibliotecas de tags que permiten gestionar de una manera bastante eficiente el apartado vista. Bibliotecas a destacar:

- *Html:* ***http://struts.apache.org/1.x/struts-taglib/dev_html.html***
- *Bean:* ***http://struts.apache.org/1.x/struts-taglib/dev_bean.html***
- *Logic:* ***http://struts.apache.org/1.x/struts-taglib/dev_logic.html***
- *Nested:* ***http://struts.apache.org/1.x/struts-taglib/dev_nested.html***

## 4.2 Elementos Básicos del Struts Framework

### 4.2.1 Action

**Introducción**

- Un **action** es el ejecutor de acciones sobre los Objetos de Negocio. Forma parte del nexo de unión entre la capa de presentación y la capa de lógica de negocio de la aplicación.
- Los action son invocados por el elemento controlador en función de la petición recibida del cliente.
- Mediante el fichero de configuración **struts-config.xml**, se establecen las relaciones entre peticiones y actions.
- Generalmente, los Action Beans siempre realizan las siguientes acciones:
  - Obtener los valores necesarios del Action Form, JavaBean, request, session o de donde sea.
  - Llamar a los objetos de negocio del Model.
  - Analizar los resultados, y según los mismos, retornar el ActionForward (destino) correspondiente.
- Extiende de la clase ***Action*** de Struts.
- La clase action tiene un método principal denominado ***execute(),*** el cual es invocado en el momento de acceder al action. Dentro de dicho método, se llevan a cabo las acciones del action.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

**Ejemplo de Action**

Ejemplo sencillo de action:

```
public final class PruebaAction extends Action {
                        public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response) {
                                        try{
                                        //Invocar negocio …
                                        si resultado correcto
                                                        destino = “OK”;
                                         si resultado incorrecto
                                                        destino = “FAIL”;
                                         }catch(Exception e){
                                                        e.printStacktrace();
                                                        destino = “FAIL”;
                                        }finally{
                                        return mapping.findForward(destino);
                                         }
                         }
}
```

La navegación definida en el fichero struts-config.xml, que indicará al Controlador dónde se debe navegar será similar a:

```
<action-mappings>
            […]
            <action path="/pruebaAction" type=“...[paquete_completo]...PruebaAction“ name=“miFormulario " scope=“request "
validate="true" input=“/jsp/miFormulario.jsp”>
                        <forward name=“OK " path=“/jsp/ok.jsp " />
                        <forward name=“FAIL” path=“/jsp/fail.jsp” />
            </action>
            […]
</action-mappings>
```

**Detalles Action**

En el ejemplo anterior vemos que la clase PruebaAction extiende de Action.
Dentro del action se encuentra el método execute que recibe como parámetros:

- ***ActionMapping mapping:*** Este objeto, mediante el método findForward, permite al desarrollador reconducir el flujo de la aplicación en función de los resultados obtenidos.
- ***ActionForm form:*** Este objeto encapsula los datos del formulario origen de la petición al action.
- ***HttpServletRequest request:*** Petición http solicitada por el navegador cliente que visita la página.
- ***HttpServletResponse response:*** Respuesta http a servir al navegador cliente que visita la página.

A continuación, dentro del método, se invoca al negocio (modelo) y en función de los resultados obtenidos, se modifica el destino (flujo).
Finalmente, se invoca al método ***findForward*** del objeto **m*apping*** para redirigir al destino que proceda.
Más adelante se verá un ejemplo más detallado.

### 4.2.2 ActionForm

**Introducción**

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239

Los ActionForm de Struts son los elementos del framework que se encargan de interactuar con los formularios html. Para ello cuentan con una serie de características que facilitan dicho proceso.
Los ActionForm son clases que extienden *ActionForm* y que implementan métodos ***get*** y ***set*** para cada una de los inputs de un form de una página, y los métodos ***validate*** y ***reset***.
Una de las tareas que durante el desarrollo de una aplicación consume mucho trabajo (aunque en realidad no lo merezcan) es la interacción con formularios, ya sea para editar u obtener nueva información. Las comprobaciones, la gestión de errores, el volver a presentarle el mismo formulario al usuario con los valores que puso y los mensajes de error y un largo etcétera están soportadas por Struts a fines de hacernos la vida un poco más fácil.
La idea es la siguiente: todo el trabajo de comprobaciones y generación de mensajes de error se implementa en los *ActionForm* y todo el trabajo de generación de interfaz en la/s JSP.
Cuando un usuario completa un formulario y lo envía, el Controlador busca en el ámbito especificado el ActionForm Bean correspondiente (todo esto configurado en el ***struts-config.xml***) y si no lo encuentra, lo crea. Luego realiza un set por cada input del formulario y finalmente llama al método ***validate***. Si éste retornara uno o más errores, el Controlador llamaría a la JSP del formulario para que ésta lo volviera a generar (con los valores establecidos por el usuario) e incluyera el o los mensajes de error correspondientes. Si todo estuviese bien, llamaría al método ***perform*** del ***Action*** (también configurado en el ***struts-config.xml***) pasándole el ***ActionForm*** Bean como parámetro para que sea utilizado para obtener los valores de los datos.

**Ejemplo de ActionForm**

```
public final class ClienteForm extends ActionForm {
private String nombre = null;

        public String getNombre() {
                return this.nombre;
        }

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

        public ActionErrors validate (ActionMapping mapping, HttpServletRequest request) {
                ActionErrors errors = new ActionErrors();
                if(nombre == null || nombre.equals("")){
                        errors.add("nombre", new ActionMessage("nombre.obligatorio"));
                }
                return errors;
        }

        public void reset (ActionMapping mapping, HttpServletRequest request) {
                this.nombre = null;
         }
}
```

**Detalles de ActionForm**

Al momento de escribir un *ActionForm* debemos tener en mente los siguientes principios:

- No debe tener nada que corresponda a la lógica de negocio.
- No debería tener más que implementaciones de getters y setters (obligatoriamente una par por cada input del form; si el input se llama nombre entonces tendremos *getNombre() y setNombre(String nombre)),* y de los métodos ***reset*** y ***validate***.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **Debe actuar como un Firewall entre el usuario y el Action deteniendo todo tipo de errores de datos incompletos o inconsistencia.**
- Si el formulario se desarrolla en varias páginas (por ejemplo, en las interfaces de tipo "Wizard"/"Asistentes") el ActionForm y el Action deberán ser los mismos, lo que permitirá, entre otras cosas, que los input se puedan reorganizar en distintas páginas sin cambiar los ActionForm ni los Action.

## 4.3 Descripción del fichero Struts-config

### 4.3.1 Introducción

Para que los elementos del framework puedan relacionarse entre sí, es necesario establecer estas vinculaciones en algún fichero. Este fichero de configuración se denomina ***struts-config.xml***.
Dentro del ***struts-config.xml*** existen diversos apartados los cuales permiten albergar las diferentes configuraciones de nuestra aplicación.
Si la aplicación está basada en este framework, la configuración de la misma, es leída de este fichero en el momento en el que la aplicación arranca. La presencia de este fichero de configuración es notificada al servidor gracias al fichero ***web.xml***, donde aparece la referencia al mismo.
Ejemplo:

```
<init-param>
        <param-name>strutsConfig</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
```

### 4.3.2 Ejemplo de fichero struts-config.xml

A continuación se muestra un ejemplo de este fichero de configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN" "lib/dtd/struts-config_1_2.dtd">
<struts-config>
        <!– declaración de form beans -->
        <form-beans>
                <form-bean name="validarUsuarioAF" type="es.gpm.formacion. aplicacion.presentacion.finals.ValidarUsuarioAF" />
                <form-bean name="mostrarProductoAF" type="es.gpm.formacion. aplicacion.presentacion.finals.MostrarProductoAF" />
                <form-bean name="carritoAF" type="es.gpm.formacion. aplicacion.presentacion.finals.CarritoAF" />
        </form-beans>

        <!– declaración de actions -->
        <action-mappings>
                <action path="/actionLogin" forward="/jsp/login.jsp"></action>
                <action path="/acceso" forward="/jsp/login.jsp"></action>
                <action path="/irprincipal" forward="/jsp/login.jsp"></action>
                <action path="/validarUsuarioA" type="es.gpm.formacion. aplicacion.presentacion.finals.ValidarUsuarioA" name="validarUsuarioAF" validate="true" input="/jsp/login.jsp">
        <!– fordwards del action -->
                        <forward name="OK" path="/jsp/carrito.jsp"></forward>
                        <forward name="error" path="/jsp/error.jsp"></forward>
                        <forward name="login" path="/jsp/login.jsp"></forward>
                </action>
                <action path="/irValidarUsuarioA" type="es.gpm.formacion. aplicacion.presentacion.finals.ValidarUsuarioA">
        <!– fordwars del action -->
                        <forward name="OK" path="/jsp/carrito.jsp" />
                        <forward name="error" path="/jsp/error.jsp" />
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

```
                                <forward name="login" path="/jsp/login.jsp" />
                        </action>
                </action-mappings>

    <!– forwards globales -->
                <global-forwards>
                        <forward name="FAIL" path="/jsp/error.jsp" />
                </global-forwards>

    <!– declaración ficheros de recursos -->
                <message-resources parameter="es.gpm.formacion.estructura. recursos.aplicacion" />
                <message-resources key="validaciones" parameter="es.gpm.formacion. estructura.recursos.validaciones" />
                <message-resources key="errores" parameter="es.gpm.formacion. estructura.recursos.errores" />
</struts-config>
```

### 4.3.3 Descripción del fichero struts-config.xml

Dentro del fichero *struts-config.xml* se encuentran las siguientes zonas:

**Declaración de los ActionForms**

- Esta zona está definida entre los tags:

```
<form-beans> </form-beans>
```

- A su vez, dentro de esta zona se encuentra la definición de cada uno de los ActionForm. Los ActionForm vienen definidos dentro de los tags

```
<form-bean> </form-bean>
```

Ejemplo:

```
<form-bean name="validarUsuarioAF" type="es.gpm.formacion. aplicacion.presentacion.finals.ValidarUsuarioAF>
</form-bean >
```

o La propiedad **name** indica el nombre del ActionForm. Este nombre será utilizado para referenciar al ActionForm cuando sea requerido.

o La propiedad **type** especifica la ubicación de la clase ActionForm dentro de la aplicación.

**Declaración de los Action**

Esta zona está definida entre los tags:

```
<action-mappings> </action-mappings >
```

A su vez, dentro de la misma, aparece la declaración de cada uno de los Action. Éstos están definidos por los tags:

```
<action> </action>
```

Ejemplo 1

```
<action path="/validarUsuarioA" type="es.gpm.formacion.aplicacion. presentacion.finals.ValidarUsuarioA" name="validarUsua-
rioAF" validate="true" input="/jsp/login.jsp">
        <forward name="OK" path="/jsp/carrito.jsp"></forward>
        <forward name="error" path="/jsp/error.jsp"></forward>
        <forward name="login" path="/jsp/login.jsp"></forward>
</action>
```



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- La propiedad ***path*** indica el patrón necesario para invocar al Action.
- La propiedad ***type*** indica la ubicación de la clase Action dentro del proyecto.
- La propiedad ***name*** indica el nombre del ActionForm asociado al Action.
- La propiedad ***validate*** indica si el formulario se valida o no.
- La propiedad ***input*** indica la procedencia de la petición a ese Action.
- Los Action tienen asociados forwads. Estos elementos permiten definir los posibles redireccionamientos de un action. En este caso, desde el Action ValidarUsuarioA, es posible ir a la página carrito.jsp, error.jsp o login.jsp.
    - La propiedad ***name*** de los forward permite especificar el nombre del forward. Este nombre será utilizado a la hora de invocar al método ***findForward*** del objeto ***mapping***.
    - La propiedad ***path*** indica la ruta a la que se desea acudir.

**Ejemplo 2**

```
<action path="/actionLogin input="/jsp/login.jsp">
```

En este caso, se ha creado un elemento que tan sólo se encarga de realizar redireccionamientos. Es lo más parecido al uso en html del href*:*

- El parámetro ***path*** indica el patrón necesario para invocar el forward.
- El parámetro ***forward*** indica el destino del forward que se desea alcanzar.

**Declaración de los recursos utilizados**

En esta zona se incluyen las referencias de los recursos utilizados. En este caso, se han incluido los ficheros de mensajes de la aplicación.
La forma de definir los recursos se lleva a cabo de la siguiente forma:

```
<message-resources key="validaciones" parameter="es.gpm. formacion.estructura.recursos.validaciones">
</message-resources>
```

- El parámetro ***key*** indica cómo referenciar a ese recurso desde la aplicación.
- El parámetro ***parameter*** indica la ubicación del recurso en cuestión.
- Es conveniente especificar un recurso que no contenga la propiedad ***key***.

**Declaración de global-forwards**

La sección *<global-forwards>* permite redireccionar solicitudes de un modo global.
Ejemplo:
En este caso, si desde alguna acción de Struts se redirige a *"FAIL"*, la solicitud será redireccionada al archivo: */jsp/error.jsp"*.

```
<global-forwards>
        <forward name="FAIL" path="/jsp/error.jsp" />
</global-forwards>
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

## 4.4 Ficheros de propiedades / internacionalización (i18n)

### 4.4.1 Ficheros de propiedades

Los ficheros de propiedades son recursos utilizados para guardar en ellos datos e información referente a la aplicación.
Los ficheros de propiedades tienen la extensión ***.properties***.
La estructura básica de un fichero de propiedades está formada por el par ***clave***, ***valor.***
Ejemplo:

nombre.requerido (clave) = El campo nombre es obligatorio (valor)

**Internacionalización (i18n)**

Una de las capacidades de Struts es la internacionalización. Para llevar a cabo este proceso se emplean los ficheros de propiedades.
Los ficheros de propiedades pueden contener títulos de páginas, etiquetas de formulario, mensajes, etc.
Para cada idioma alternativo se creará un archivo nuevo que se llame igual pero que termine en***"_xx.properties"*** siendo xx el código ISO de idioma (ej: mensajes_en.properties).
De esta manera podemos tener algo así:

***mensajes.properties***

```
validacion.usuario.nombre.requerido = El nombre del usuario es requerido
validacion.usuario.edad.requerido = La edad del usuario es requerida
```

…

***mensajes_en.properties***

```
validacion.usuario.nombre.requerido = User´s name mandatory
validacion.usuario.edad.requerido = User´s age mandatory
```

…

Como se observa, el mismo mensaje está en dos idiomas diferentes. Cuando se determine el idioma de presentación, la aplicación utilizará el fichero de propiedades correspondiente para mostrar los mensajes.
Ejemplo: La manera de mostrar un mensaje internacionalizado dentro de una JSP sería:

```
<bean:message bundle="labels" key="label.url"/>
```

El fichero struts-config.xml deberá tener la siguiente declaración del fichero .properties donde se encuentra la clave buscada y su correspondiente traducción.

```
<message-resources key="labels" parameter="es.gpm.formacion.estructura. recursos.validaciones">
</message-resources>
```

## 4.5 Tags de Struts

Las etiquetas personalizadas de struts se agrupan en tag libraries. El framework de Struts aprovecha las capacidades de las librerías de tags de JSP para incluir varias categorias diferentes de etiquetas y así ayudar a hacer más manejable y reusable la capa de presentación. Con el uso de las librerías de tags, los desarrolladores puede interactuar con el resto del Framework sin necesidad de incluir código Java en las páginas JSP.
Esta taglib contiene etiquetas que se usan para crear formularios de introducción de datos de Struts, así como otras etiquetas generalmente usadas para la creación de interfaces de usuario basadas en HTML.
La definición de una taglib se realiza en un Tag Library Definition (archivo xml con extensión TLD donde se define el nombre del tag, la clase TagHandler que lo atiende, la definición de sus atributos, si tiene body, etc.)
A su vez se debe declarar en el fichero web.xml de la siguiente forma:

```
<web-app>
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
        <taglib>
                <taglib-uri> nombreTagLib</taglib-uri>
                <taglib-location>/directorios/nombreArchivo.tld
                </taglib-location>
        </taglib>
</web-app>
```

En la JSP donde se utilizará se incluirá de la siguiente forma:

```
<%@ taglib uri="nobreTagLib" prefix="prefijoTagLib" %>
```

Finalmente, el Tag que se usa en la JSP consiste en:

```
 <prefijoTagLib:nombreTag atributo=valor ... >
```

Para más info., ver la página oficial de Sun sobre TagLibraries en http://java.sun.com/products/jsp/taglibraries.html

### 4.5.1 Tags struts-html

Los tags de la librería HTML forman un puente entre una vista JSP y otros componentes de la aplicación web. Desde que las aplicaciones web dinámicas empezaron a depender de los datos introducidos por un usuario, los formularios de entrada juegan un importante papel en el framework de Struts. Por esta razón, la mayoría de los tags de esta librería corresponden a formularios HTML.
Dentro de esta librería también se encuentran etiquetas que dan soporte al apartado de mensajes, mensajes de error, hipervínculos e internacionalización.
Con la sentencia siguiente se añade la taglib a la JSP:

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

**Etiquetas text, form, submit**

- Lo que se encuentre dentro de la etiqueta *<html:form>* estará dentro de un formulario. La propiedad *action="/altaEmpleado"* indica la acción que se ejecutará cuando se haga submit del formulario.
- Las etiquetas *<html:text/>* permiten mostrar cajas de texto de introducción de datos dentro del formulario. La *"property"* de cada *<html:text>* indica la propiedad asociada del ActionForm. Cada etiqueta *<html:errors/>* permite mostrar los errores asociados a cada propiedad del formulario, si los hubiera.
- *<html:submit>* permite mostrar un botón que al pulsarlo enviará el formulario y se ejecutará la acción especificada en él. *<html:cancel>* muestra igualmente un botón, pero cancela cualquier acción que se esté realizando.

Ejemplo: altaEmpleado.jsp

```
<%@ page language="java"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-bean" prefix="bean"%>
<%@ taglib uri="http://jakarta.apache.org/struts/tags-html" prefix="html"%>
<html>
<head>
<title>JSP for altaEmpleadoForm form</title>
</head>
<body>
        <html:form action="/altaEmpleado">

        Id del Empleado : <html:text property="idEmpleado"/>
        <html:errors property="idEmpleado"/><br/>

        Nombre : <html:text property="nombre"/>
        <html:errors property="nombre"/><br/>
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
            Cargo : <html:text property="cargo"/>
            <html:errors property="cargo"/><br/>

            Provincia : <html:text property="codigoprovincia"/>
            <html:errors property="codigoprovincia"/><br/>

            Sector : <html:text property="codigosector"/>
            <html:errors property="codigosector"/><br/>

            Comisión : <html:text property="comision"/>
            <html:errors property="comision"/><br/>

            Salario : <html:text property="salario"/>
            <html:errors property="salario"/><br/>

            Fecha de alta : <html:text property="fechaalta"/>
            <html:errors property="fechaalta"/><br/>

<html:submit/><html:cancel/>

</html:form>
</body>
</html>
```

**Etiqueta checkbox**

En la **JSP** declaramos la etiqueta dándole un nombre para poder recuperar el valor del check seleccionado (*property=”elCheck”*). El *<bean:message >* es el texto que aparecerá a la izquierda del check. El valor asociado a cada check se puede establecer con la propiedad *“value”* o simplemente poniendo el valor entre los tags de apertura y cierre [46-56].

**ActionForm**: la propiedad asociada al checkbox debe ser siempre de tipo boolean. También podemos darle un valor por defecto para cada vez que carguemos el formulario, se hace en el método **reset**. También podemos dar ese valor en el action que carga esa página, se haría de la misma manera.

El reset quedaría así (ActionForm):

```
public void reset(ActionMapping mapping, HttpServletRequest request) {
elCheck=true;
 }
```

La JSP contendría el siguiente código:

```
<bean:message key="index.elcheck" /><html:checkbox property="elCheck" value="on"/>
```

**Etiqueta multibox**

Es una colección de *checkbox*, en el ejemplo siguiente vemos su implementación.

En la JSP declaramos la etiqueta dándole un nombre para poder recuperar el valor del check seleccionado (*property=”selected”*). Evidentemente, todos los check tienen el mismo nombre, ya que son parte del multibox. El *<bean:message >* que aparece antes de cada elemento del multibox es el texto que aparecerá a la izquierda del check. El valor asociado a cada check se puede establecer con la propiedad *“value”* o simplemente poniendo el valor entre los tags de apertura y cierre.

Ejemplo:

```
<table>
            <tr>
                  <bean:message key="index.opcion1"/>
                  <html:multibox property="selected">1 </html:multibox>
            </tr>
            <tr>
                  <bean:message key="index.opcion2" />
                  <html:multibox property="selected" value="2"/>
</tr>
            <tr>
                  <bean:message key="index.opcion3" />
```



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
                <html:multibox property="selected"> 3 </html:multibox>
            </tr>
            <tr>
                <bean:message key="index.opcion4" />
                <html:multibox property="selected"> 4 </html:multibox>
            </tr>
</table>
```

Estableceremos una propiedad en el ActionForm asociado a la página para referirnos al multibox, ha de ser un array de strings, que es lo que nos devuelve el multibox. Obligatoriamente hay que poner el multibox en el método reset para que no se produzcan errores en el caso de no seleccionar nada.

```
private String[] selected;
    public String[] getSelected() {
    return selected;
}
public void setSelected(String[] selected) {
    this.selected = selected;
}
public void reset(ActionMapping mapping, HttpServletRequest request) {
    selected=null;
}
```

En el **ActionForm** también podemos darle un valor por defecto para que cada vez que se cargue el formulario, aparezcan chequeados los elementos deseados. Desde el método **reset**, es posible iniciar estos valores. Se trabaja sobre el value del control no sobre la posición.

```
public void reset(ActionMapping mapping, HttpServletRequest request) {
    selected = new String{"1","3"};
}
```

También podemos dar ese valor en el **Action** que carga esa página, se haría de la misma manera.

```
ListarTemaForoForm formulario = (ListarTemaForoForm)form;
String[] loseleccionado = formulario.getSelected();
if (loseleccionado!=null){
    for (int i=0; i<loseleccionado.length;i++){
        System.out.println("Valor " + i + "en Array: " + loseleccionado[i].toString());
    }
    retorno = "pruebamulti";
}
```

**Etiquetas select, option, options, options collection**

En el **ActionForm** es necesario que exista el bean del que vamos a obtener las propiedades para rellenar el combo.

```
public class ListarTemaForoForm extends ActionForm {
//El combo
private String codPais;
//El bean
private ArrayList listaPaises;

public String getCodPais() {
        return codPais;
}

public void setCodPais(String codPais) {
        this.codPais = codPais;
}

public ArrayList getListaPaises() {
        return listaPaises;
}
public void setListaPaises(ArrayList listaPaises) {
        this.listaPaises = listaPaises;
}
```

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239

```
}
```

La JSP quedaría de la siguiente manera:

```
<html:select styleClass="verdana10normalazul" property="idPais">
<html:option value="-1">
<bean:message bundle="textos" key="combo.seleccion"/>
</html:option>
<html:optionsCollection name=" listarTemaForoForm " label=" nomPais " value=" idPais " property=" listaPaises " />
</html:select>
```

Si queremos que el combo sea de **selección múltiple**, se añade el atributo *"multiple"* con un valor numérico cualquiera, los valores de este combo se recuperan igual que el multibox, con un array de String:

```
<html:select property="codPais2" multiple="3">
```

Lo mismo con optionsCollection:

```
<html:select property="codPais">
<html:option value="-1">Seleccione</html:option>
<html:optionsCollection name="listaTemasForm" property="listaPaises"  label="paiDescripcion"  value="paiCodPais"/>
</html:select>
```

- La propiedad ***name*** es el nombre que tiene el formulario en el struts-config.
- La propiedad ***property*** es el nombre del bean del que queremos obtener datos (un ArrayList en este caso).
- La propiedad ***label*** es el nombre que muestra el elemento en el combo.
- La propiedad ***value*** es el valor que se almacena para cada elemento en el combo.

## Etiqueta radio

En el **Form**, mediante el método reset, se inicia el valor del componente.

```
public class ListarTemaForoForm extends ActionForm {
private String elRadio;
public String getElRadio() {
        return elRadio;
}
public void setElRadio(String elRadio) {
        this.elRadio = elRadio;
}

public void reset(ActionMapping mapping, HttpServletRequest request) {
        elRadio="1";
}
}
```

En el Action:

```
public class ListarTemasForoAction extends Action {
public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse re-
sponse) {

if (form.getElRadio()!=null){
        String valorRadio=form.getElRadio();
        System.out.println("RADIO: " + valorRadio);
}
return mapping.findForward(retorno);

}
}
```

En la JSP:

```
<bean:message key="r.nombre"/><html:radio property="elRadio" value="1"/>
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
<bean:message key="r.nombr2"/><html:radio property="elRadio" value="2"/>
<bean:message key="r.nombr3"/><html:radio property="elRadio" value="3"/>
```

**Etiqueta submit**

Es un botón que, como su nombre indica, hace un submit del formulario. Se envía el formulario de modo que se pasen los valores al **ActionForm:**

```
public class AltaTemaForoForm extends ActionForm implements AltaTemaForoForm{
private String aceptar;
public String getAceptar() {
        return aceptar;
}
public void setAceptar(String aceptar) {
        this.aceptar = aceptar;
}
}
```

En el Action:

```
if (formulario.getAceptar() != null){
        retorno=”aceptado”;
}else{
        retorno=”cancelado”;
}
return mapping.findForward(retorno);
```

En la JSP:

```
<html:submit property="aceptar" >
        <bean:message key="altaTemaForo.botonAceptar" />
</html:submit>
```

**Etiqueta cancel**

Es un botón normal, con la particularidad de que no se ejecutara la función *validate()* del formulario asociado, la codificación es igual que la de *<html:submit >*.

**Etiqueta file**

Sirve para importar y tratar ficheros.

En el **Form** se declara una variable de tipo *FormFile*, que es la que va a contener el fichero:

```
public class AltaTemaForoForm extends ActionForm {
private FormFile fichero;

public FormFile getFichero() {
        return fichero;
}

public void setFichero(FormFile fichero) {
        this.fichero = fichero;
}
}
```

En la JSP se pone la propiedad enctype en el <html:form >, y declaramos el <html:file >

```
<html:form action="altaTemaForo.do" enctype="multipart/form-data">
        EL FILE  <html:file property="fichero" size="50"/>
```

**Etiqueta errors**

Sirve para mostrar mensajes de tipo error en la página, generalmente se controla en el *validate()*.

En el Form:

```
public ActionErrors validate( ActionMapping mapping, HttpServletRequest request) {
ActionErrors errores=new ActionErrors();

if ((cfoTitulo==null)||(cfoTitulo.equals(""))){
        errores.add(“camponombre", new ActionError(“nombre.obligatorio"));
}
return errores;
}
```

En la JSP:

```
<html:errors bundle=“ficheropropiedades” property=“camponombre”/>
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

### 4.5.2 Tags struts-logic

Gran parte de los tags contenidos dentro de esta librería se encuentran disponibles también dentro de la *JSTLJavaServer Pages Standard Tag Library*).
Esta librería contiene tags que son útiles para la gestión de generación de salidas de texto condicionales, para recorrer colecciones de objetos, para repeticiones de salida de texto y para la gestión del flujo de la aplicación.

**Etiqueta iterate**

Permite iterar los valores de un bean.
En la **JSP** se itera el ArrayList "listaPaises", que contiene los bean con la descripción de cada país:

- **id** es para darle nombre a cada elemento que iteremos, en este caso, cada objeto bean del ArrayList.
- **name** es para especificar el vector, arraylist, etc que queremos iterar.

```
<logic:iterate id="pais" name="listaTemasForm" property="listaPaises">
        <br>
        <bean:write name="pais" property="paisDescripcion"/>
</logic:iterate>
```

**Etiquetas present, not present**

La etiqueta <logic:present > evalúa un parámetro que le llega por request, y, dependiendo de su existencia o no, se lleva a cabo una acción o no.

En el **Action**:

```
String parametro="prueba logic";
request.setAttribute("parametro",parametro);
retorno="validar";
```

En la JSP:

```
<logic:present name="parametro" scope="request">
        <bean:message key="mensajes.prueba.present" bundle="mensajes"/>
</logic:present>
```

La etiqueta <logic:notPresent > evalúa un parámetro que le llega por *request*, y si no existe, se lleva a cabo la acción correspondiente. Funciona igual que <logic:present > .

**Etiqueta equal**

Tag que evalúa si una variable especificada es igual a un determinado valor.

En el **Action**:

```
String parametro="probando equal"
request.setAttribute("parametro",parametro);
retorno="validar";
PaisVO pais=new PaisVO();
pais.setPaisCodPais("1");
pais.setPaisDescripcion("España");
request.setAttribute("pais",pais);
```

En la JSP:
Se puede pasar un bean *(PaisVO*), en cuyo caso se referencia con *name* y la propiedad que se desee comparar con *property*, el valor a comparar es *value;* o es posible pasarle un parámetro, en cuyo caso se usa *parameter* para referenciarlo, y se compara también con el value.
Ejemplo 1

```
<logic:equal name="pais" property="paisDescripcion" value="España">
<bean:message key="mensajes.equal.present" bundle="mensajes"/>
</logic:equal>
```

Ejemplo2


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
<logic:equal parameter="parametro" value="probando equal">
<bean:message key="mensajes.equal.present" bundle="mensajes"/>
</logic:equal>
```

### 4.5.3 Tags struts-bean

Algunas de las características en esta librería están también disponibles en la *JSTL* (*JavaServer Pages Standard Tag Library*). El equipo de Struts recomienda utilizar los tags de la JSTL cuando sea posible en lugar de los específicos de Struts.
Esta librería contiene etiquetas que permiten utilizar beans y sus propiedades desde la jsp.

**Etiqueta message**

Tag que permite mostrar un mensaje internacionalizado. Se debe especificar, directamente, el *key* del mensaje, o indirectamente, usando los atributos *name* y *property* para obtenerlo del bean. El atributo *bundle* permite especificar el nombre del alcance de la aplicación donde se encuentra el *MessageResources*. Si no se especifica el local, el *Locale* se obtendrá del la sesión usando la clave *Action.LOCALE_KEY.*

```
<td><bean:message key="global.user.firstName"/>:</td>
```

**Etiqueta write**

Tag que permite mostrar el valor de una propiedad de un bean. La tag *write* utiliza las siguientes reglas:

- Si se especifica el atributo *format*, el valor se formateará en función del formato string y el *locale* por defecto. El atributo *formatKey* también se puede usar especificando el nombre del formato de string del resource bundle.
- El atributo *formatKey* se usa para especificar un formato desde el resourse bundle. Se puede especificar qué resource bundle y qué locale usar. Si no se especifica, se usará el resource bundle por defecto y el locale actual.

## 4.6 DispatchAction

Supongamos el caso de una funcionalidad CRUD típica (**C**: create, **R**: read, **U**: update, **D**: delete). Sería deseable englobar estas operaciones en una sola clase, ocultando los métodos y fomentando la reusabilidad. Para tal fin se creó la clase DispatchAction.
DispatchAction es una clase abstracta que extiende de la clase Action, y para su uso se debe sobrescribir. Su estructura se compone de un método por cada acción lógica que se desee englobar, y se especifica el método a invocar en función de un parámetro almacenado en la request.
Para usar DispatchAction, se deben seguir los siguientes pasos:

1. Crear una clase que extienda de DispatchAction.
2. Crear un método para cada acción lógica.
3. Crear un action mapping para esa clase, especificando el atributo *"attribute"* para especificar el parámetro de la request que almacenará el nombre del método que se desea invocar.
4. Pasar al action un parámetro por request que indique el método a invocar.

Detalladamente:

- **Creación de la clase y de los métodos para cada acción lógica:**

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

```
public class UsuarioDispatchAction extends DispatchAction {

 public ActionForward remove(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

            System.out.println("REMOVE USER");
            ...
            return mapping.findForward("success");
    }

    public ActionForward save(
        ActionMapping mapping,
        ActionForm form,
        HttpServletRequest request,
        HttpServletResponse response)
        throws Exception {

            System.out.println("SAVE USER");
             ...
            return mapping.findForward("success");
    }

}
```

Estos métodos tienen la misma forma que el Action.execute estándar, salvo el nombre, por supuesto.

- **Y crear el action mapping:**

```
<action  path="/dispatchUsuarioSubmit" type="action.UsuarioDispatchAction" parameter="method" input="/form/usuarioForm.jsp"
name="usuarioForm" scope="request" validate="false">
            <forward name="success" path="/success.jsp" />
</action>
```

Así, el DispatchAction que hemos creado usa el valor del parámetro *"method"* de la request para elegir el método apropiado a ejecutar.
El atributo *"parameter"* de este mapping indica cuál es el nombre de la variable que se buscará en la request.
Como último paso, se debe pasar al action un parámetro en la request donde indiquemos el nombre del método que deseamos ejecutar. Para ello se puede usar cualquiera de las alternativas a nuestra disposición: javascript, hidden, actionForm…
Supongamos el uso de un combo para elegir la acción a ejecutar. La **JSP** sería como sigue:

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean"%>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html"%>
...

    <html:link action="home">Home</html:link>

    <html:form action="/dispatchUserSubmit">
            ...
        action:
         <html:select property="method" size="2">
           <html:option value="save">Save</html:option>
           <html:option value="remove">Remove</html:option>
        </html:select> <br/>

        <html:submit/><html:cancel/>
    </html:form>
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

...

Al fin y al cabo lo único necesario es disponer de un par clave-valor entre los valores de la request en que la clave se llame igual que el atributo *"parameter"* especificado en el mapping del struts-config.xml y la clave contenga un nombre de método válido.

## 4.7 Validator Framework

### 4.7.1 Validaciones

En la inmensa mayoría de las aplicaciones web J2EE nos encontraremos con formularios de toma de datos. Estos formularios son presentados al usuario como la forma de interactuar con la aplicación y contendrán, según la complejidad de los mismos, distintos tipos y formatos de datos como pueden ser:

- Campos numéricos
- Enteros
- Decimales
- Campos de fecha
- Formatos (dd/MM/yyyy, dd/MM/yy, MM/dd/yyyy)
- Campos alfanuméricos
- Tamaño máximo etc.

Estos son unos de los pocos ejemplos posibles de tipos de datos que puede haber en un formulario. La mayoría de los Frameworks comerciales tienen formas de realizar validaciones sobre los datos proporcionados en los formularios. Por ejemplo, los métodos validate() de los ActionForm de Struts permiten realizar validaciones sobre los datos de los formularios.
Existe el problema de la descentralización y repetición de código, es decir, para dos ActionForm distintos tendremos distintas validaciones.
Si los formularios tienen, por ejemplo, dos campos de fecha tendremos que repetir el código de validación en los dos ActionForm para realizar la validación de la fecha. Si en un futuro el formato de fecha cambia, tendremos que tocar todos nuestros ActionForm que tengan validaciones de fecha, lo que lo hace poco mantenible.
Para solucionar este problema se creó VALIDATOR que nos proporciona una centralización de las rutinas de validación, tanto para presentación como para negocio.

### 4.7.2 Beneficios del uso de Validator

- Una sola definición de las reglas de validación para una aplicación.
- Las reglas de validación de cliente y servidor pueden estar definidas en un solo lugar.
- Configurar nuevas reglas y/o cambios en las reglas existentes es más simple.
- Soporta internacionalización.
- Soporta expresiones regulares.
- Se puede utilizar tanto para aplicaciones web como para aplicaciones estándar de Java.

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239

### 4.7.3 Configuración de Validator

El framework de validator se distribuye empaquetado con el framework de Struts. Sin embargo, tenemos que habilitarlo en el fichero de configuración struts-config.xml:

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

De este modo estamos diciendo a struts que cargue e inicialice el plugin de Validator para la aplicación.
Una vez hecha la inicialización, el plugin carga los archivos de configuración de Validator especificados en la propiedad *pathnames*. En *value* debemos especificar la ruta donde están situados estos dos ficheros de configuración: *validator-rules.xml* y *validation.xml*.

**validator-rules.xml y validation.xml**

- **Validator – rules.xml**: Contiene el cómo, es decir, contiene las reglas de validación propiamente dichas. Por defecto, el validator-rules.xml contiene una serie de reglas predefinidas que nos servirán para validar la mayoría de los formatos de campos estándar; No obstante se pueden ampliar el número de reglas de forma sencilla para que cubran todos los requisitos de validación que necesiten los formularios de nuestra aplicación.

- **Validation.xml**: Contiene el qué, es decir, contiene una definición de qué reglas de validación tenemos que aplicar a qué formulario.

**Configuración de validator-rules.xml, ejemplo de regla de validación**

A continuación podemos ver un ejemplo de regla de validación que trae por defecto la distribución del framework validator para cantidades enteras:

```
<validator name="integer" classname="org.apache.struts.validator.FieldChecks"
          method="validateInteger"
          methodParams="java.lang.Object,
           org.apache.commons.validator.ValidatorAction,
           org.apache.commons.validator.Field,
           org.apache.struts.action.ActionMessages,
           org.apache.commons.validator.Validator,
           javax.servlet.http.HttpServletRequest"
          depends="" msg="errors.integer"
          jsFunctionName="IntegerValidations" />
```

Para definir una regla necesitamos aportar la siguiente información:

- **Name**: hay que especificar un nombre a la regla de validación.
- **Classname**: La clase que se encarga de realizar la validación.
- **Method**: Método de la clase que contendrá la regla de validación.
- **MethodParams**: Los objetos que va a recibir como parámetro el método de validación.
- **Depends**: Son las reglas de las que depende esta regla.
- **msg**: Es el mensaje internacionalizado que va a aparecer por defecto si la regla de validación falla.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

- **jsFunctionName**: Es la función javascript que se lanzará en caso de que esta regla de negocio se esté invocando desde presentación.

En el fichero de validation.xml, se configura el cómo, es decir a qué formulario se va a aplicar la regla. Dentro del formulario habrá que especificar a qué campo o campos aplicaremos la regla.

```
<form name="insertarEntradaValidator">
<field property="idTipo" depends="required,integer">
                                    <arg0 key="Tipo" resource="false" />
</field>
</form>
```

En el ejemplo se aplicará al formulario llamado “insertarEntradaValidator”. Se validará que el campo idTipo sea obligatorio (required) y que además sea de tipo entero (integer). Se pasa como argumento “Tipo” esto nos servirá para mostrar mensajes por pantalla del tipo. “El campo Tipo es obligatorio” o “El campo Tipo no es de tipo entero”.
Algunas validaciones básicas proporcionadas por el framework:

| **Nombre** | **Descripción** |
|---|---|
| byte,short,integer, long,float,double | Comprueba si el valor puede ser convertido de forma segura a su correspondiente tipo primitivo. |
| creditCard | Comprueba si el campo es un número de tarjeta de crédito válido. |
| date | Comprueba si el campo es una fecha válida. |
| email | Comprueba si el campo es una dirección e-mail válida. |
| mask | Tiene éxito si el campo empareja la máscara regular correspondiente de la expresión. |
| maxLength | Comprueba si la longitud del valor es menor o igual a la longitud máxima dada. |
| minLength | Comprueba si la longitud del valor es mayor o igual a la longitud mínima dada. |
| range | Comprueba si el valor está dentro de un rango mínimo y máximo. |
| required | Comprueba si el campo no es nulo y la longitud del campo es mayor de cero, no incluyendo espacios en blanco. |

### 4.7.4 Configuración del struts-config.xml

Es importante apuntar que, para que la validación se lleve a cabo, hay que especificar validate="true“, al declarar la acción en el struts-config-xml

```
<action path="/insertarEntrada" type="es.paquete.ejemplo.web.actions. UsuariosDispatchAction" name=" insertarEntradaValidator"
scope="session" parameter="method" validate=“true”
</action>
```

ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

### 4.7.5 Creación de validaciones propias

El framework validator también permite programar validaciones personalizadas de modo que es posible crear nuevas clases para realizar nuestras propias validaciones. Para ello tenemos que:

1. Crear una nueva clase para la validación:
2. Creamos una nueva clase que implemente la interfaz Serializable

```
public class ValidationClass implements Serializable
```

3. Creamos un método para realizar la validación

```
public static boolean validacionPersonalizada(Object bean,
        ValidatorAction va, Field field, ActionErrors errors, HttpServletRequest request)
```

4. Insertamos la regla de validación en el fichero validator-rules-xml

```
<validator name="personalizada" classname="org.apache.struts. validator.FieldChecks" method="validacionPersonalizada"
methodParams="java.lang.Object,
org.apache.commons.validator.ValidatorAction,
org.apache.commons.validator.Field,
org.apache.struts.action.ActionMessages,
org.apache.commons.validator.Validator,
javax.servlet.http.HttpServletRequest"
depends="" msg="errors.mensaje.personal" jsFunctionName="" />
```

5. Insertamos la regla de validación en el fichero validation.xml

```
<form name="insertarEntradaValidator">
        <field property="idTipo" depends=" personalizada">
                <arg0 key="Tipo" resource="false" />
        </field>
</form>
```

### 4.7.6 Validaciones en presentación

Como hemos dicho, el framework validator permite usar las mismas validaciones para presentación y negocio.
Para validar el formulario en presentación tendremos que:

- Meter la tag *<html:javascript />* en la jsp del formulario:

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

- Esta tag recibirá el nombre del formulario mapeado en nuestro archivo de configuración validation.xml. En nuestro ejemplo "insertarEntradaValidator"
- Por defecto este tag generará una función javascript de nombre validate formName. En el ejemplo anterior será validateInsertarEntradaValidator(). Dicha función contendrá la validaciones javascript necesarias para validar el formulario.
- Podemos modificar el nombre por defecto rellenando el parámetro *method* de la tag *<html:javascript>*
- Finalmente deberemos de llamar a la función javascript que nos generará el framework validator:

```
<html:form action="insertarEntrada.do" method="post" onsubmit="return
        validateInsertarEntradaValidator(this);">
```

Hay que destacar que las funciones javascript a las que llamará para hacer las validaciones deberán de estar definidas jsFunction de cada una de las reglas de validación configuradas en el validation-rules.xml.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

## 4.8 Tiles

### 4.8.1 Soluciones a repetición de código

Normalmente en el desarrollo de una aplicación java con tecnología JEE tendemos a repetir gran cantidad de código e incluirlo en todas las páginas de la aplicación.
Cuando la aplicación es de un tamaño considerable realizar un cambio que afecte a una parte común de todas las páginas es una tarea, pesada, repetitiva y lenta.
La solución ofrecida por muchos de los desarrollos web actuales es el realizar inclusiones de páginas:

```
<jsp:include page="/header.jsp" />
```

Esta solución no es óptima ya que aunque reutilizamos código en todas nuestras páginas, cada página de la aplicación está creando su propio diseño. Si cambia el diseño general de aplicación tendremos que cambiar todas las páginas de la aplicación perdiendo mucho tiempo en realizar el cambio.

### 4.8.2 Solución: Tiles y Struts

El modo de organizar la información propuesta por esta solución es el uso de plantillas para todas las páginas de la aplicación.
La plantilla usada en toda la aplicación contendrá:

- El diseño principal de la aplicación.
- Una referencia a las diferentes páginas que formarán el contenido de la página.
- Todos los ficheros que necesitamos incluir en nuestra aplicación:
    - Funciones javascript
    - Estilos CSS
    - IFrames

En caso de que necesitemos cambiar el diseño general de la aplicación, únicamente tendremos que cambiar el contenido de dicha plantilla y se cambiará para toda la aplicación.

### 4.8.3 Configuración de Tiles

Existen 2 documentos que hay que definir o configurar

- **tiles-def.xml:** En este fichero se definirá la plantilla que vamos a aplicar en la aplicación. Pueden existir más de un fichero de Tiles-def.xml, generalmente existirá uno general donde configurar la plantilla llamado Tiles-def.xml y otro por cada uno de los módulos llamado tiles-nombreModulo-def.xml.
- **struts-config.xml:** Tendremos que configurar nuestro fichero de configuración de Struts indicándole cada uno de los fichero de Tiles que tengamos configurados para nuestra aplicación.

### 4.8.4 Configuración tiles-def.xml

Un fichero tiles-def.xml podría tener el siguiente aspecto:

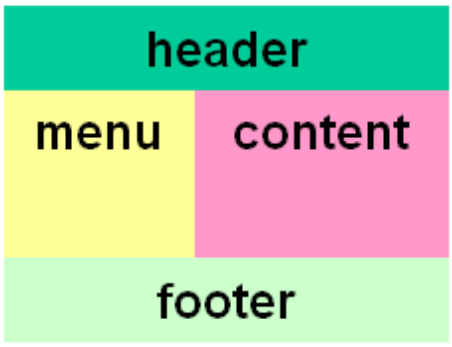
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"  "http://ja-
karta.apache.org/struts/dtds/tiles-config_1_1.dtd">
```

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

```
<tiles-definitions>
<definition name=".templateHome" path="/templates/templateHome.jsp">
<put name="title" value="${title}" />
        <put name="content" value="${content}" />
        <put name="header" value="/templates/headerHome.jsp" />
        <put name="menu" value="/templates/menuHome.jsp" />
        <put name="footer" value="/templates/footerHome.jsp" />
</definition>
</tiles-definitions>
```

Así la plantilla quedaría divida en secciones tal y como se muestra en la siguiente imagen.




Dentro de las etiquetas </tiles-definitions> tendremos definidas todas las plantillas que necesitemos usar en nuestra aplicación. En el ejemplo tenemos definida una única plantilla; Para definir la plantilla tendremos que usar la tag <definition/>.
Para realizar la definición de nuestra plantilla tendremos que especificar:

- Un nombre *name=".templateHome"*.
- Una ruta a la página jsp que compondrá el diseño de nuestra plantilla en el ejemplo *path="/templates/templateHome.jsp"*.

Mediante el uso de las etiquetas <put /> pasaremos parámetros a nuestra plantilla que nos servirán para, por ejemplo, especificar los includes que tendrá nuestra página, en el ejemplo anterior nos servirán para pasar como parámetros a la plantilla un título, el contenido principal, la cabecera, el menú y el pie.
Si abrimos la jsp que forma nuestra plantilla "templateHome.jsp" tenemos las siguientes tags dentro de la jsp:

```
<tiles:useAttribute name="title" />
<title> ${title}</title>
</tiles:useAttribute>
```

Es decir, que el parámetro que teníamos definido con <put name="title" .../>, lo estamos usando dentro de la jsp que nos sirve como plantilla.
Dentro del la jsp templateHome.jsp tenemos además:

```
<tiles:get name="header"/>
<tiles:get name="menu"/>
<tiles:get name="content"/>
<tiles:get name="footer"/>
```

Este tag nos servirá para realizar un *include* de la jsp que pasemos como parámetro.

**Configuración tiles-def.xml (Herencia)**

Tenemos una plantilla llamada templateHome que tiene definida una serie de includes de páginas:

```
<put name="header" value="/templates/headerHome.jsp" />
<put name="menu" value="/templates/menuHome.jsp" />
<put name="footer" value="/templates/footerHome.jsp" />
```


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

Y además podemos especificarle dos parámetros, el título y el contenido:

```
<put name="title" value="${title}" />
<put name="content" value="${content}" />
```

Podemos usar además "herencia" para crear a partir de nuestra plantilla principal templateHome el resto de páginas de nuestra aplicación del siguiente modo:

```
<definition name=".usuarios.home" extends=".templateHome">
        <put name="title" value="Gestión de usuarios" />
        <put name="content" value="/jsp/usuarios/gestionUsuarios.jsp" />
</definition>
```

Con esto estamos diciendo que nuestra página usuarios.home tendrá el aspecto definido en templateHome.jsp, pero tendrá un <title> y un contenido central personalizados.

Generalmente se suele definir un tiles-def.xml principal en el cual tendremos nuestra o nuestras plantillas padres, de las cuales heredarán el resto de las plantillas de nuestra aplicación. Por otro lado tendremos otra serie de ficheros tiles-modulo-def.xml donde definiremos todas las páginas de la aplicación que pertenecen a un módulo concreto.

**Configuración struts-config.xml**

Finalmente deberemos de configurar el struts-config.xml con el fin de que sepa localizar las plantillas definidas en nuestros ficheros de configuración de tiles. Para ello tendremos que incluir:

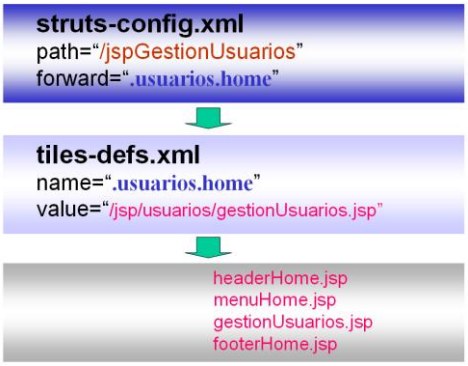
```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
        <set-property property="definitions-config"
value="/WEB-INF/tiles/tiles-def.xml,
WEB-INF/tiles/tiles-usuarios-def.xml" />
        <set-property property="definitions-parser-validate" value="true" />
        <set-property property="moduleAware" value="true" />
</plug-in>
```

De esta manera los nombres dados en nuestro fichero de configuración pueden ser usados en los ficheros de struts-config.xml para realizar las redirecciones; Por ejemplo para redirigir a la página de gestión de usuarios .usuarios.home, podremos tener una acción definida en el struts-config como ésta:

```
        <action path="/jspGestionUsuarios" forward=".usuarios.home" />
```

Con la configuración explicada en los pasos anteriores tendremos que al poner en nuestro navegador http://Direccion_servidor:puerto/contexto_aplicacion/jspGestionUsuarios.do , se navegará a la página que tenemos definida .usuarios.home, que será una página compuesta por las páginas *headerHome.jsp, menuHome.jsp, gestionUsuarios.jsp y footerHome.jsp.*

En gestionUsuarios.jsp tendremos la funcionalidad y en el resto tendremos las partes comunes a toda la aplicación.




ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

## 5 Enlaces de interes

- Página oficial de Struts:
  - http://struts.apache.org/
- Librería de etiquetas BEAN:
  - http://struts.apache.org/1.x/struts-taglib/dev_bean.html
- Librería de etiquetas HTML:
  - http://struts.apache.org/1.x/struts-taglib/dev_html.html
- Librería de etiquetas LOGIC:
  - http://struts.apache.org/1.x/struts-taglib/dev_logic.html
- Librería de etiquetas NESTED:
  - http://struts.apache.org/1.x/struts-taglib/dev_logic.html
- Guía rápida de Struts:
  - http://www.strutskickstart.com/
- Conceptos básicos Struts:
  - http://struts.apache.org/userGuide/index.html
- Struts Validator Guide:
  - http://struts.apache.org/1.2.4/userGuide/dev_validator.html
- Struts Validator API:
  - http://struts.apache.org/1.2.4/api/org/apache/struts/validator/package-summary.html#package_description
- Creating an Input Form JSP Page Using the Struts Validator to Validate Data Entry:
  - http://www.oracle.com/technology/obe/obe_as_1012/j2ee/develop/client/strutsvalidator/validator.htm
- Check Your Form with Validator:
  - http://otn.oracle.com/oramag/oracle/04-jan/o14dev_struts.html
- Struts Validator two fields match:

### References

1. Adrián Sánchez-Carmona, Sergi Robles, Carlos Borrego (2015). Improving Podcast Distribution on Gwanda using PrivHab: a Multiagent Secure Georouting Protocol. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 1
2. Baruque, B., Corchado, E., Mata, A., & Corchado, J. M. (2010). A forecasting solution to the oil spill problem based on a hybrid intelligent system. Information Sciences, 180(10), 2029–2043. https://doi.org/10.1016/j.ins.2009.12.032
3. Casado-Vara, R., Chamoso, P., De la Prieta, F., Prieto J., & Corchado J.M. (2019). Non-linear adaptive closed-loop control system for improved efficiency in IoT-blockchain management. Information Fusion.
4. Casado-Vara, R., de la Prieta, F., Prieto, J., & Corchado, J. M. (2018, November). Blockchain framework for IoT data quality via edge computing. In Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems (pp. 19-24). ACM.
5. Casado-Vara, R., Novais, P., Gil, A. B., Prieto, J., & Corchado, J. M. (2019). Distributed continuous-time fault estimation control for multiple devices in IoT networks. IEEE Access.


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

6. Casado-Vara, R., Vale, Z., Prieto, J., & Corchado, J. (2018). Fault-tolerant temperature control algorithm for IoT networks in smart buildings. Energies, 11(12), 3430.
7. Casado-Vara, R., Prieto-Castrillo, F., & Corchado, J. M. (2018). A game theory approach for cooperative control to improve data quality and false data detection in WSN. International Journal of Robust and Nonlinear Control, 28(16), 5087-5102.
8. Chamoso, P., González-Briones, A., Rivas, A., De La Prieta, F., & Corchado J.M. (2019). Social computing in currency exchange. Knowledge and Information Systems.
9. Chamoso, P., González-Briones, A., Rivas, A., De La Prieta, F., & Corchado, J. M. (2019). Social computing in currency exchange. Knowledge and Information Systems, 1-21.
10. Chamoso, P., González-Briones, A., Rodríguez, S., & Corchado, J. M. (2018). Tendencies of technologies and platforms in smart cities: A state-of-the-art review. Wireless Communications and Mobile Computing, 2018.
11. Chamoso, P., Rivas, A., Martín-Limorti, J. J., & Rodríguez, S. (2018). A Hash Based Image Matching Algorithm for Social Networks. In Advances in Intelligent Systems and Computing (Vol. 619, pp. 183–190). https://doi.org/10.1007/978-3-319-61578-3_18
12. Chamoso, P., Rodríguez, S., de la Prieta, F., & Bajo, J. (2018). Classification of retinal vessels using a collaborative agent-based architecture. AI Communications, (Preprint), 1-18.
13. Choon, Y. W., Mohamad, M. S., Deris, S., Illias, R. M., Chong, C. K., Chai, L. E., … Corchado, J. M. (2014). Differential bees flux balance analysis with OptKnock for in silico microbial strains optimization. PLoS ONE, 9(7). https://doi.org/10.1371/journal.pone.0102744
14. Corchado, J. A., Aiken, J., Corchado, E. S., Lefevre, N., & Smyth, T. (2004). Quantifying the Ocean's CO2 budget with a CoHeL-IBR system. In Advances in Case-Based Reasoning, Proceedings (Vol. 3155, pp. 533–546).
15. Corchado, J. M., & Fyfe, C. (1999). Unsupervised neural method for temperature forecasting. Artificial Intelligence in Engineering, 13(4), 351–357. https://doi.org/10.1016/S0954-1810(99)00007-2
16. Corchado, J. M., Borrajo, M. L., Pellicer, M. A., & Yáñez, J. C. (2004). Neuro-symbolic System for Business Internal Control. In Industrial Conference on Data Mining (pp. 1–10). https://doi.org/10.1007/978-3-540-30185-1_1
17. Corchado, J. M., Corchado, E. S., Aiken, J., Fyfe, C., Fernandez, F., & Gonzalez, M. (2003). Maximum likelihood hebbian learning based retrieval method for CBR systems. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 2689, pp. 107–121). https://doi.org/10.1007/3-540-45006-8_11
18. Corchado, J. M., Pavón, J., Corchado, E. S., & Castillo, L. F. (2004). Development of CBR-BDI agents: A tourist guide application. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 3155, pp. 547–559). https://doi.org/10.1007/978-3-540-28631-8
19. Corchado, J., Fyfe, C., & Lees, B. (1998). Unsupervised learning for financial forecasting. In Proceedings of the IEEE/IAFE/INFORMS 1998 Conference on Computational Intelligence for Financial Engineering (CIFEr) (Cat. No.98TH8367) (pp. 259–263). https://doi.org/10.1109/CIFER.1998.690316
20. Costa, Â., Novais, P., Corchado, J. M., & Neves, J. (2012). Increased performance and better patient attendance in an hospital with the use of smart agendas. Logic Journal of the IGPL, 20(4), 689–698. https://doi.org/10.1093/jigpal/jzr021
21. Cristian Peñaranda, Jorge Agüero, Carlos Carrascosa, Miguel Rebollo, Vicente Julián (2016). An Agent-Based Approach for a Smart Transport System. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 2
22. David Griol, Jose Manuel Molina, Araceli Sanchís De Miguel (2014). Developing multimodal conversational agents for an enhanced e-learning experience. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 3, n. 1
23. Di Mascio, T., Vittorini, P., Gennari, R., Melonio, A., De La Prieta, F., & Alrifai, M. (2012, July). The Learners' User Classes in the TERENCE Adaptive Learning System. In 2012 IEEE 12th International Conference on Advanced Learning Technologies (pp. 572-576). IEEE.
24. Fdez-Riverola, F., & Corchado, J. M. (2003). CBR based system for forecasting red tides. Knowledge-Based Systems, 16(5–6 SPEC.), 321–328. https://doi.org/10.1016/S0950-7051(03)00034-0
25. Fernández-Riverola, F., Díaz, F., & Corchado, J. M. (2007). Reducing the memory size of a Fuzzy case-based reasoning system applying rough set techniques. IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews, 37(1), 138–146. https://doi.org/10.1109/TSMCC.2006.876058


ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

26. Fyfe, C., & Corchado, J. (2002). A comparison of Kernel methods for instantiating case based reasoning systems. Advanced Engineering Informatics, 16(3), 165–178. https://doi.org/10.1016/S1474-0346(02)00008-3
27. Fyfe, C., & Corchado, J. M. (2001). Automating the construction of CBR systems using kernel methods. International Journal of Intelligent Systems, 16(4), 571–586. https://doi.org/10.1002/int.1024
28. Gabriel Santos, Tiago Pinto, Zita Vale, Isabel Praça, Hugo Morais (2016). Enabling Communications in Heterogeneous Multi-Agent Systems: Electricity Markets Ontology. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 2
29. García Coria, J. A., Castellanos-Garzón, J. A., & Corchado, J. M. (2014). Intelligent business processes composition based on multi-agent systems. Expert Systems with Applications, 41(4 PART 1), 1189–1205. https://doi.org/10.1016/j.eswa.2013.08.003
30. García, O., Chamoso, P., Prieto, J., Rodríguez, S., & De La Prieta, F. (2017). A serious game to reduce consumption in smart buildings. In Communications in Computer and Information Science (Vol. 722, pp. 481–493). https://doi.org/10.1007/978-3-319-60285-1_41
31. Glez-Bedia, M., Corchado, J. M., Corchado, E. S., & Fyfe, C. (2002). Analytical model for constructing deliberative agents. International Journal of Engineering Intelligent Systems for Electrical Engineering and Communications, 10(3).
32. Glez-Peña, D., Díaz, F., Hernández, J. M., Corchado, J. M., & Fdez-Riverola, F. (2009). geneCBR: A translational tool for multiple-microarray analysis and integrative information retrieval for aiding diagnosis in cancer research. BMC Bioinformatics, 10. https://doi.org/10.1186/1471-2105-10-187
33. Gonzalez-Briones, A., Chamoso, P., De La Prieta, F., Demazeau, Y., & Corchado, J. M. (2018). Agreement Technologies for Energy Optimization at Home. Sensors (Basel), 18(5), 1633-1633. doi:10.3390/s18051633
34. González-Briones, A., Chamoso, P., Yoe, H., & Corchado, J. M. (2018). GreenVMAS: virtual organization-based platform for heating greenhouses using waste energy from power plants. Sensors, 18(3), 861.
35. Gonzalez-Briones, A., Prieto, J., De La Prieta, F., Herrera-Viedma, E., & Corchado, J. M. (2018). Energy Optimization Using a Case-Based Reasoning Strategy. Sensors (Basel), 18(3), 865-865. doi:10.3390/s18030865
36. Jesús Ángel Román Gallego, Sara Rodríguez González (2015). Improvement in the distribution of services in multi-agent systems with SCODA. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 3
37. Jörg Bremer, Sebastian Lehnhoff. (2017) Decentralized Coalition Formation with Agent-based Combinatorial Heuristics. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 3
38. José Alemany, Stella Heras, Javier Palanca, Vicente Julián (2016). Bargaining agents based system for automatic classification of potential allergens in recipes. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 2
39. Laza, R., Pavn, R., & Corchado, J. M. (2004). A reasoning model for CBR_BDI agents using an adaptable fuzzy inference system. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 3040, pp. 96–106). Springer, Berlin, Heidelberg.
40. Leonor Becerra-Bonache, M. Dolores Jiménez López (2014). Linguistic Models at the Crossroads of Agents, Learning and Formal Languages. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 3, n. 4
41. Li, T., Sun, S., Corchado, J. M., & Siyau, M. F. (2014). A particle dyeing approach for track continuity for the SMC-PHD filter. In FUSION 2014 - 17th International Conference on Information Fusion. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84910637583&partnerID=40&md5=709eb4815eaf544ce01a2c21aa749d8f
42. Li, T., Sun, S., Corchado, J. M., & Siyau, M. F. (2014). Random finite set-based Bayesian filters using magnitude-adaptive target birth intensity. In FUSION 2014 - 17th International Conference on Information Fusion. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84910637788&partnerID=40&md5=bd8602d6146b014266cf07dc35a681e0
43. Lima, A. C. E. S., De Castro, L. N., & Corchado, J. M. (2015). A polarity analysis framework for Twitter messages. Applied Mathematics and Computation, 270, 756–767. https://doi.org/10.1016/j.amc.2015.08.059
44. Mata, A., & Corchado, J. M. (2009). Forecasting the probability of finding oil slicks using a CBR system. Expert Systems with Applications, 36(4), 8239–8246. https://doi.org/10.1016/j.eswa.2008.10.003
45. Méndez, J. R., Fdez-Riverola, F., Díaz, F., Iglesias, E. L., & Corchado, J. M. (2006). A comparative performance study of feature selection methods for the anti-spam filtering domain. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4065 LNAI, 106–120.



ISBN: 978-84-9012-859-6
Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc

Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-33746435792&partnerID=40&md5=25345ac884f61c182680241828d448c5
46. Omar Jassim, Moamin Mahmoud, Mohd Sharifuddin Ahmad (2014). Research Supervision Management Via A Multi-Agent Framework. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 3, n. 4
47. Sittón-Candanedo, I., Alonso, R. S., Corchado, J. M., Rodríguez-González, S., & Casado-Vara, R. (2019). A review of edge computing reference architectures and a new global edge proposal. Future Generation Computer Systems, 99, 278-294.
48. Paula Andrea Rodríguez Marín, Néstor Duque, Demetrio Ovalle (2015). Multi-agent system for Knowledge-based recommendation of Learning Objects. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 1
49. Ricardo Silveira, Guilherme Klein Da Silva Bitencourt, Thiago Ângelo Gelaim, Jerusa Marchi, Fernando De La Prieta (2015). Towards a Model of Open and Reliable Cognitive Multiagent Systems: Dealing with Trust and Emotions. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 3
50. Rodriguez-Fernandez J., Pinto T., Silva F., Praça I., Vale Z., Corchado J.M. (2018) Reputation Computational Model to Support Electricity Market Players Energy Contracts Negotiation. In: Bajo J. et al. (eds) Highlights of Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection. PAAMS 2018. Communications in Computer and Information Science, vol 887. Springer, Cham
51. Rodríguez, S., De La Prieta, F., Tapia, D. I., & Corchado, J. M. (2010). Agents and computer vision for processing stereoscopic images. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 6077 LNAI). https://doi.org/10.1007/978-3-642-13803-4_12
52. Rodríguez, S., Gil, O., De La Prieta, F., Zato, C., Corchado, J. M., Vega, P., & Francisco, M. (2010). People detection and stereoscopic analysis using MAS. In INES 2010 - 14th International Conference on Intelligent Engineering Systems, Proceedings. https://doi.org/10.1109/INES.2010.5483855
53. Rodríguez, S., Tapia, D. I., Sanz, E., Zato, C., De La Prieta, F., & Gil, O. (2010). Cloud computing integrated into service-oriented multi-agent architecture. IFIP Advances in Information and Communication Technology (Vol. 322 AICT). https://doi.org/10.1007/978-3-642-14341-0_29
54. Sigeru Omatu, Tatsuyuki Wada, Pablo Chamoso (2013). Odor Classification using Agent Technology. DCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 4
55. Sittón, I., & Rodríguez, S. (2017). Pattern Extraction for the Design of Predictive Models in Industry 4.0. In International Conference on Practical Applications of Agents and Multi-Agent Systems (pp. 258–261).
56. Tapia, D. I., & Corchado, J. M. (2009). An ambient intelligence based multi-agent system for alzheimer health care. International Journal of Ambient Computing and Intelligence, v 1, n 1(1), 15–26. https://doi.org/10.4018/jaci.2009010102

ISBN: 978-84-9012-859-6

Colección Aquilafuente: AQ-239
Ediciones Universidad de Salamanca - cc by nc dc