Servlets

Y

JSP

Contenido

Arquitectura de las Aplicaciones Web 5
Las Capas de la Aplicación 5
Capas de las Aplicaciones Web 6
La Capa de Dominio 6
La Capa de la Interfaz de Usuario 6
La Capa Web 7
La Capa de Servicios 7
La Capa de Acceso a Datos 7
Introducción Servlets 9
Arquitectura de Aplicación Servlet/JSP 9
El Hypertext Transfer Protocol (HTTP) 10
La Petición HTTP 11
La Respuesta HTTP 11
Servlets 13
Interfaz `Servlet` 13
Configuración Básica de un Servlet 14
Estructura de Directorios de la Aplicación 14
Interfaz `ServletRequest` 15
Interfaz `ServletResponse` 15
Interfaz `ServletConfig` 15
Interfaz `ServletContext` 16
`GenericServlet` 16
HTTP `Servlets` 17
Interfaz `HttpServletRequest` 18
Interfaz `HttpServletResponse` 18
Trabajar con Formularios HTML 19
Uso del Descriptor de Despliegue 19
Gestión de Sesión 20
Objetos `HttpSession` 20
JavaServer Pages 22
Visión General de JSP 22
Objetos Implícitos 23
Directivas 24
La Directiva page 24
La Directiva `include` 25

Elementos de Scripting 25
Scriptlets 25
Expresiones 26
Declaraciones 26
Inhabilitar los Elementos de Scripting 27
Acciones 27
`useBean` 27
`setProperty` y `getProperty` 27
`include` 28
`forward` 28
Lenguaje de Expresión EL 29
Sintaxis del Lenguaje de Expresión 29
Palabras reservadas 29
Los operadores `[]` y `.` 29
La regla de evaluación 29
Acceso a JavaBeans 30
Objetos Implícitos 30
Otros operadores EL 33
Operadores Aritméticos 33
Operadores relacionales 34
Operadores lógicos 34
Operador condicional 34
El operador `empty` 34
Configuración de EL en las versiones 2.0 y posteriores 34
Inhabilitación de scripts en JSP 34
Desactivar la evaluación EL 35
JSTL 36
Acciones de Propósito General 36
La Etiqueta `out` 36
La Etiqueta `set` 37
La Etiqueta `remove` 38
Acciones Condicionales 38
La Etiqueta `if` 38
Las Etiquetas `choose`, `when` y `otherwise` 39
Acciones de Iteración 40

La Etiqueta `forEach` 40
La Etiqueta `forTokens` 41
Acciones de Formateado 42
La Etiqueta `formatNumber` 42
La Etiqueta `formatDate` 43
La Etiqueta `timeZone` 45
La Etiqueta `setTimeZone` 45
La etiqueta `parseNumber` 45
La Etiqueta parseDate 46
Filtros 48
El API `Filter` 48
`init` 48
`doFilter` 48
`destroy` 48
Configuración de Filtros 49
Orden de los Filtros 50

Arquitectura de las Aplicaciones Web

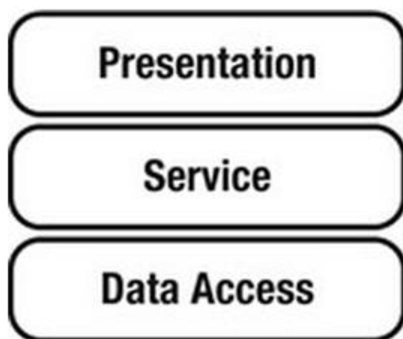
Una de las arquitecturas que más se utiliza en aplicaciones Web o de empresa es aquella en que la aplicación se divide en una serie de capas, cada una de ellas desempeñando una funcionalidad claramente diferenciada.

Se va a ver las capas en la que se divide una aplicación web. También se verá los distintos roles que desempeñan cada capa en la aplicación.

Las Capas de la Aplicación

Una aplicación consta de varias capas. Cada capa representa un área de responsabilidades de la aplicación. Por lo tanto su usan capas para conseguir una separación de responsabilidades. Por ejemplo, la vista no debería estar entrelazada con lógica de negocio o acceso a datos, ya que son todas diferentes responsabilidades y normalmente se colocan en diferentes capas.

Las capas se pueden ver como limites conceptuales, pero no tienen por qué estar físicamente separadas unas de otras. La siguiente figura muestra una imagen generalizada de las capas de una aplicación.

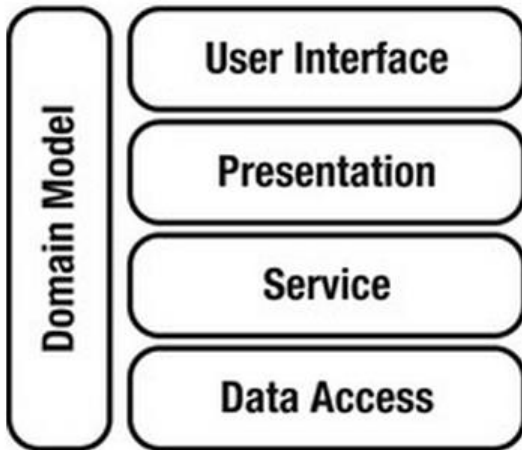


La capa de acceso a datos esta en el fondo de la aplicación, la de presentación el la cima, y la de servicios en el medio.

Capa	Descripción
`Presentación`	Debe ser tan ligera como sea posible. También debe ser posible proporcionar capas de presentación alternativas como páginas web o fachadas de servicios web.
`Servicios`	Punto de entrada al sistema conteniendo la lógica de negocio. Proporciona la interfaz que permite el uso del sistema. También es la capa que especifica los límites transaccionales del sistema, y también probablemente la seguridad. No debe saber nada de la tecnología usada en la capa de presentación ni de la vista.
Acceso a Datos	Capa basada en interfaz que proporciona acceso a la tecnología de acceso de datos subyacente, pero sin exponerla a las capas superiores. Abstrae el framework de persistencia real (JDBC, JDO, JPA, Hibernate, iBatis, etc.). No tiene que contener lógica de negocio.

La comunicación entre la capas es desde la cima a la base. Es decir, la capa de servicios puede acceder a la capa de acceso a datos, pero la capa de acceso a datos no puede acceder a la capa de servicios. Las dependencias circulares, o las dependencias de la base a la cima, son síntomas de un mal diseño.

Esta disposición en capas de las aplicaciones web, se puede descomponer un poco más. Así, en una aplicación web típica, se puede identificar cinco capas conceptuales. Se puede descomponer la capa de presentación en una capa web y una capa de interfaz de usuario. Por otro lado, la aplicación también incluirá una capa de dominio, que atraviesa todas las capas porque es usada en todas las capas desde la capa de acceso a datos a la capa de interfaz de usuario.



Capas de las Aplicaciones Web

Ahora se va a introducir las cinco capas que se han definido antes. Se verá el papel que juegan cada una de estas capas y que debe ir en cada capa.

La Capa de Dominio

Es la más importante de la aplicación. Es la representación del problema de negocio que se esta resolviendo, y contiene la reglas de negocio del dominio.

Una técnica usada para determinar el modelo del dominio es usar los nombres en las descripciones de los casos de uso como objetos del dominio. Estos objetos tienen tanto estado, atributos del objeto, como comportamiento, los métodos del objeto. Estos métodos normalmente son más específicos que los métodos de la capa de servicios.

La Capa de la Interfaz de Usuario

Esta capa presenta la aplicación al usuario. Esta capa transforma la respuesta generada por el servidor en el tipo pedido por el cliente del usuario. Por ejemplo, un navegador web probablemente pedirá un documento HTML, un servicio web podría querer un documento XML, y otro cliente podría pedir un documento PDF o Excel.

Se separa la capa de presentación en capa de interfaz de usuario y capa web porque, a pesar del amplio rango de diferentes tecnologías de vista, se quiere reutilizar tanto código como sea posible. El objetivo es reimplementar solo la interfaz de usuario. Se debería ser capaz de cambiar la interfaz de usuario sin tener que tocar la parte del servidor de la aplicación.

La interfaz de usuario en general tiene dependencias en la capa de dominio. A veces, es conveniente exponer y crear la vista del modelo de dominio directamente.

La capa de interfaz de usuario se puede implementar con diferentes tecnologías.

La Capa Web

La capa web tiene dos responsabilidades. La primera responsabilidad es guiar al usuario a través de la aplicación web. La segunda es ser la capa de integración entre la capa de servicio y HTTP.

La navegación del usuario a través de la aplicación puede ser tan sencilla como una hacer corresponder una URL a una vista o una solución de flujo de páginas. La navegación está solo ligada a la capa web, y no debe de haber ninguna lógica de navegación en las capas de dominio o de servicios.

Como capa de integración, la capa web debes ser tan liguera como sea posible. Debe ser la capa que convierte peticiones HTTP de entrada a algo que pueda ser tratado por la capa de servicio, y después transformar el resultado, si hay alguno, del servidor a respuestas para la interfaz de usuario. La capa web no debe contener ninguna lógica de negocio, esta es solo responsabilidad de la capa de servicios.

La capa web también esta formada por cookies, cabeceras HTTP, y probablemente una sesión HTTP. Es responsabilidad de la capa web gestionar todos estos elementos de forma consistente y transparente.

La capa web depende de la capa de dominio y la capa de servicios. En la mayoría de los casos, se quiere transformas la petición entrante en un objeto del dominio y llamar a un método en la capa de servicios para hacer algún procesamiento con ese objeto del dominio.

La Capa de Servicios

La capa de servicios es una capa muy importante en la arquitectura de una aplicación. Se puede considerar el corazón de la aplicación porque expones la funcionalidad (casos de uso) del sistema al usuario. Lo hace proporcionando un API con funcionalidades más generales. Se tiene que hacer una única llamada a un método para que el cliente complete un único caso de uso.

De forma ideal, una función general debe representar una única unidad de trabajo que o tiene éxito o fracasa. El usuario puede utilizar diferentes clientes (ej. Aplicación web, servicio web, o aplicación de escritorio), no obstante, esos clientes deben ejecutar la misma lógica de negocio. Por lo tanto, la capa de servicios debe ser el único punto de entrada al sistema.

El beneficio añadido de tener un único punto de entrada al sistema con funcionalidades generales en la capa de servicios es que se puede aplicar de forma simple transacciones y seguridad a esta capa. Ahora los requerimientos transaccionales y de seguridad son parte integral del sistema.

En un entorno web, tendremos varios usuarios operando en los servicios al mismo tiempo. Por esto, los servició deben ser sin estado, por esto es una buena practica hacer los servicios como singletons.

Mantener la capa de servicio como un único punto de entrada, sin estado, y aplicar transacciones y seguridad en esta capa permite exponer la capa de servicios a diferentes clientes.

La capa de servicios depende de la capa de dominio para ejecutar la lógica de negocio. No obstante, también depende de la capa de acceso a datos para almacenar y recuperar objetos del almacén de datos. La capa de servicio debe coordinar que objetos de dominio necesita y como interactúan entre ellos.

La Capa de Acceso a Datos

La capa de acceso a datos es la responsable de interactuar con el mecanismo de persistencia subyacente. Esta capa sabe como almacenar y recuperar objetos del almacén de datos. Lo hace de tal forma que la capa de servicios no sabe que almacén de datos se esta utilizando (puede ser una base de datos, o consistir en fichero planos en el sistema de ficheros).

La razón principal para crear una capa de acceso de datos separada es que se pretende manejar la persistencia de una forma transparente, sin que la capa de servicios tenga ningún conocimiento del tipo de almacén de datos que se está utilizando. El objetivo es poder cambiar el tipo y tecnología del almacén de datos, sin tener que modificar la capa de servicios.

Introducción Servlets

La tecnología Servlet de Java es la tecnología base para desarrollar aplicaciones web en Java. Lanzada en 1996 para competir con las aplicaciones CGI que entonces era el estándar para generar contenido dinámico en la web. El principal problema con las aplicaciones CGI era el hecho de que arrancaba un nuevo proceso por cada nueva petición HTTP. Un servlet, por otro lado, es un mecanismo mucho más rápido que CGI porque el servlet permanece en memoria después de dar servicio a la primera petición, esperando a subsiguientes peticiones.

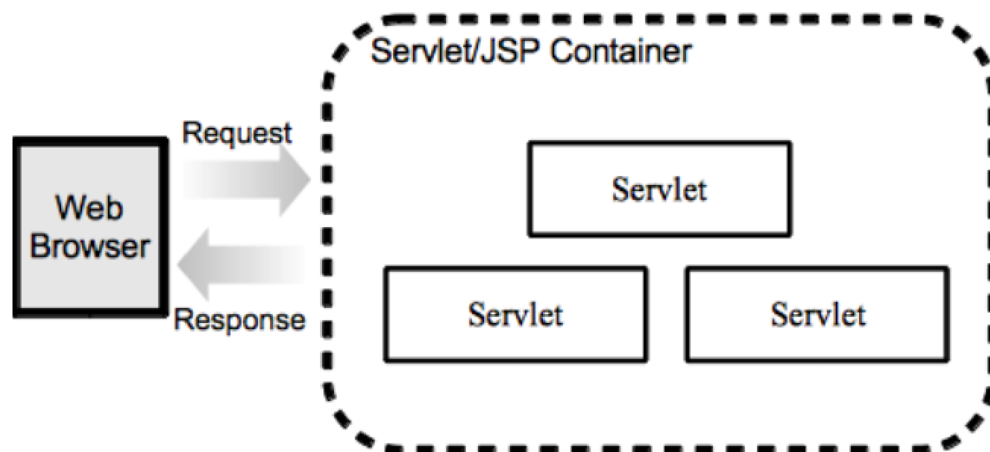
Los `Servlets` son clases Java que se ejecutan en un contendor de servlets. Un contenedor de servlets o motor de servlets es como un servidor web pero que tiene la habilidad de generar contenido dinámico, no solo servir recursos estáticos.

Arquitectura de Aplicación Servlet/JSP

Un servlet es un programa Java. Una aplicación servlet contiene un o más servlets. Una página JSP se traduce y compila a un servlet.

Una aplicación servlet se ejecuta en un contenedor de servlets y no puede ejecutarse por si misma. Un contenedor de servlets pasa peticiones del usuario a la aplicación servlet y respuestas desde la aplicación servlet de vuelta al usuario.

Los usuarios web utilizan navegadores web para acceder a las aplicaciones servlet. A los navegadores web se les llama clientes web. La siguiente figura muestra la arquitectura de una aplicación servlet/JSP.



El servidor web y el cliente web se comunican en un lenguaje que ambos comprenden: el Protocolo de Transferencia de Hipertexto (HTTP).

Un contenedor servlet/JSP es un servidor web especial que puede procesar servlets a si como servir contenido estático. Apache `Tomcat` y `Jetty` son los contenedores servlet/JSP mas populares y son gratuitos y de código abierto.

`Servlet` y `JSP` son dos de las múltiples tecnologías definidas en la especificación `Java Enterprise Edition (EE)`. Entre tras tecnologías `Java EE`, están `Java Message Service` (`JMS`), `Enterprise JavaBeans` (`EJB`), `JavaServer Faces` (`JSF`), y `Java Persistence` API.

Para ejecutar una aplicación `Java EE`, se necesita un contenedor `Java EE`, como `GlassFish`, `JBoss`,

Oracle WebLogic, e IBM WebSphere. Se puede desplegar una aplicación servlet/JSP en un contenedor Java EE, pero un contenedor servlet/JSP es suficiente y más ligero que un contenedor Java EE. Tomcat y Jetty no son contenedores Java EE, así que no pueden ejecutar EJB ni JMS.

El Hypertext Transfer Protocol (HTTP)

El protocolo HTTP permite a los servidores y navegadores web intercambiar datos sobre Internet o una intranet.

Un servidor web se ejecuta permanentemente esperando clientes web que se conecten a él y pidan recursos. En HTTP el cliente es siempre quien inicia una conexión, un servidor nunca esta en disposición de contactar con un cliente. Para localizar recursos se utiliza URLs (Uniform Resource Locator), por ejemplo:

```
http://google.con/index.html
```

La primera parte de la URL es `http`, que identifica el protocolo usado. No todas las URLs usan HTTP.

En general una URL tiene el siguiente formato:

```
protocolo://[host.]dominio[:puerto[/contexto][/recurso][?cadena consulta]
```

o

```
protocolo://direccion IP[:puerto[/contexto][/recurso][?cadena consulta]
```

Una dirección IP (Internet Protocol) es una etiqueta numérica asignada a una computadora u otro dispositivo. Una computadora puede albergar más de un dominio, así que varios dominios pueden tener la misma dirección IP.

La parte host puede estar presente e identifica a localizaciones totalmente diferentes en Internet o una intranet. Con el paso de los años www ha sido el nombre host más popular y se ha convertido el nombre host por defecto.

80 es el puerto por defecto de HTTP. Por esto, si un servidor web se ejecuta en el puerto 80, no es necesario el número de puerto para alcanzar el servidor. Pero a veces, un servidor web no se ejecuta en el puerto 80 y se necesita especificar el número de puerto. Por ejemplo, Tomcat se ejecuta por defecto en el puerto 8080, así que se necesita proporcionar el número de puerto

```
http://localhost:8080
```

`localhost` es un nombre reservado que se usas para hacer referencia a la computadora local, es decir, la misma computadora en la cual se ejecuta el navegador web.

El contexto en una URL hace referencia al nombre de la aplicación, pero también es opcional. Un servidor web puede ejecutar múltiple contextos (aplicaciones) y uno de ellos se puede configurar para que sea el contexto por defecto. Para pedir un recurso en el contexto por defecto, se puede omitir la parte del contexto en la URL.

Finalmente, un contexto puede tener uno o más recursos por defecto. Una URL sin un nombre de recurso se considera que identifica a un recurso por defecto. Por supuesto, si existe más de un recurso por defecto en un contexto, se devolverá el de mayor prioridad cuando un cliente no especifique un nombre de recurso.

Después del nombre del recurso viene una o más cadenas de consulta. Una cadena de consulta es un par clave/valor que se le puede pasar el servidor para ser procesada.

La Petición HTTP

Una petición HTTP consta de tres componentes:

- Método – URI (Uniform Resource Identifier) – Protocolo/versión
- Cabeceras de la petición
- Cuerpo de la petición

Un ejemplo de petición HTTP:

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Langue: en-gb
Connection: Keep-Alive
Host: localhost
User-Agent: Mozilla/5.0 (Macintosh; U; Inter Mac OS X 10.5; en-US;rv:1.9)
Gecko/20100625 Firefox/3.6.6
Content-Length: 30
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

lastName=Blanks&firstName=Mike
```

El método de petición – la URI – El protocolo y su versión aparecen el la primera línea de la petición.

Una petición HTTP puede usar uno de los distintos métodos especificados en el estándar HTTP. HTTP 1.1 soporta siete tipos de peticiones: GET, POST, HEAD, OPTIONS, PUT, DELETE, y TRACE. GET y POST son los más comúnmente usados en las aplicaciones web.

La URI especifica un recurso. Se interpreta normalmente como relativo al directorio raíz del servidor. En consecuencia, siempre debe comenzar con /. Una URL es realmente un tipo de URI.

En una petición HTTP, las cabeceras de la petición contienen información útil a cerca del entorno del cliente y del cuerpo de la petición. Cada cabecera esta separada por una secuencia de retorno de caro/avance de línea (CRLF).

Entre las cabeceras y el cuerpo hay una line en blanco (CRLF), que le dice al servidor HTTP donde comienza el cuerpo.

El cuerpo, si lo hay, consisten en paras de claves/valor separados por el símbolo &.

La Respuesta HTTP

Al igual que la petición HTTP, una respuesta HTTP también consisten en tres partes:

- Protocolo – Código de estado – Descripción
- Cabeceras de respuesta
- Cuerpo

Un ejemplo de respuesta HTTP:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Date: Thu, 5 Jan 2012 13:13:33 GMT
Last-Modified: Wed, 4 Jan 2012 13:13:12 GMT
```

```
Content-Length: 30
<html>
<head>
<title>Ejemplo Respuesta HTTP</title>
</head>
<body>
   Hola como estamos
</body>
</html>
```

La primera línea de la respuesta es similar a la primera línea de la petición. Indica cual es el protocolo utilizado y el código de estado. Un código de estado `200` significa que el recurso pedido ha sido encontrado, existen otros códigos para expresar otras situaciones. Por ejemplo, el código `404` significa que el recurso pedido no se ha podido localizar, `401` es acceso no autorizado, el `405` método no permitido, etc.

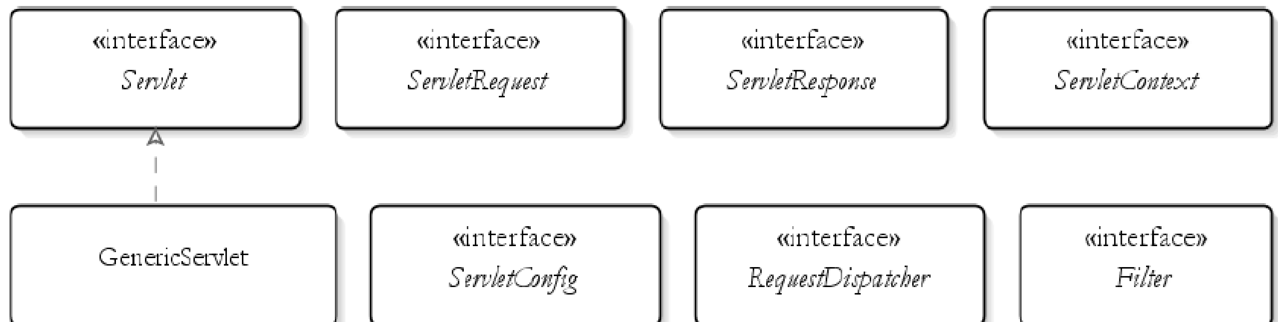
La cabecera de respuesta contiene información útil similar a las cabeceras de la petición.

El cuerpo de la respuesta es el contenido del recurso pedido. Normalmente será en formato `HTML`.

Servlets

`Servlet` es la tecnología principal para desarrollar servlets.

`Servlet`, es una interfaz que todas las clases servlet deben implementar o bien directamente o bien indirectamente.



La interfaz `Servlet` define un contrato entre un servlet y el contenedor de servlets. El contrato se reduce a la promesa por parte del contenedor de cargar en memoria la clase servlet y llamar a unos métodos específicos en la instancia del servlet. Solo puede haber una instancia de cada tipo servlet en una aplicación.

Una petición de usuario provoca que el contenedor de servlets llame al método `service` del servlet, pasándole una instancia de `ServletRequest` y de `ServletResponse`. `ServletRequest` encapsula la petición HTTP actual de tal forma que no se tenga que transformar y manipular los datos HTTP. `ServletResponse` representa la respuesta HTTP para el usuario actual y facilita enviar la respuesta de vuelta al usuario.

Para cada aplicación el contenedor de servlets también crea una instancia de `ServletContext`. Este objeto encapsula los detalles de entorno del contexto (aplicación). Solo existe un `ServletContext` por cada contexto. Para cada instancia de servlet, también hay un `ServletConfig` que encapsula la configuración del servlet.

Interfaz Servlet

La interfaz `Servlet` define cinco métodos:

```
void init(ServletConfig config) throws SerlvetException;
void service(ServletRequest request, ServletResponse response)
   throws SerlvetException, IOException;
void destroy();
String getServletInfo();
SerlvetConfig getServletConfig();
```

`init`, `service`, y `destroy` son métodos de ciclo de vida. El contenedor de servlets invoca estos métodos de a cuerdo con las siguientes reglas.

- `init`. El contenedor de servelts invoca este método la primera vez que se hace una petición al servlet. Este método no es llamado en subsiguientes peticiones. Se usa este método para escribir código de inicialización. Cuando es invocado este método, el contenedor de servlets le pasa un objeto

ServletConfig. Normalmente, se asignará el objeto ServletConfig a un atributo del servlet para que pueda ser usado por otras partes de la clase servlet.

- service. El contenedor de servlets invoca este método cada vez que llega una petición para el servlet. En este método se pone el código que realiza la funcionalidad asociada al serlvet.
- destroy. El contenedor de servlets invoca esta método cuando el servlet va ha ser destruido. Esto ocurre cuando la aplicación de desinstalada o cuando el contenedor de servlets se para. Normalmente, se escribe código de liberación de recursos en este método.

Los otros dos métodos, getServletInfo y getServletConfig en Servlet son métodos normales:

- getServletInfo. Este método devuelve la descripción del servlet. Se puede devolver cualquier cadena que pudiera ser útil, o incluso null.
- getServletConfig. Este método devuelve el objeto ServletConfig pasado por el contenedor al método init. No obstante, para que getServletConfig devuelva un valor no nulo, se debe asignar el objeto ServletConfig pasado al método init a un atributo de la clase servlet.

Una instancia de un servlet se comparte por todo los usuarios de la aplicación, de tal forma que no es recomendado tener definidos en la clase servlet atributos de instancia.

Configuración Básica de un Servlet

La configuración más sencilla es mediante anotaciones.

La anotación a nivel de clase @WebServlet se usa para declarar un servlet.

Se puede dar un nombre al servlet mediante el atributo name de esta anotación. Este atributo es opcional.

El atributo urlPatterns, que aunque opcional casi siempre presente, le dice al contenedor de servlets que patrón URL debe invocar al servlet. El patrón siempre debe comenzar por la barra inclinada (/).

Estructura de Directorios de la Aplicación

Una aplicación servlet se debe desplegar con una estructura de directorios determinada.

El directorio es el directorio de la aplicación. Bajo el directorio de la aplicación esta el directorio WEB-INF. Este a su vez tiene dos subdirectorios:

- classes. En este directorio residen las clases de los servlets y otras clases Java. Los directorios bajo el directorio classes reflejan los paquetes de las clases.
- lib. En este directorios van los ficheros jar necesarios para la aplicación.

Una aplicación servlet/JSP normalmente tiene páginas JSP, ficheros HTML, ficheros de imágenes, y otros recursos. Estos deben ir bajo el directorio de la aplicación y se organizan normalmente en subdirectorios.

Cualquier recurso puesto bajo el directorio de la aplicación es directamente accesible al usuario introduciendo la URL del recurso en el navegador. Si se quiere que un determinado recurso sea accesible solo por los servlets, y no accesible por el usuario, se deben poner bajo el directorio WEB-INF.

EL método recomendado para desplegar una aplicación servlet/JSP es desplegarlo como un fichero war. Un fichero war es un fichero jar con la extensión war. Se pueden crear ficheros war de diferentes maneras, mediante la aplicación jar que viene con el JDK de Java, o con una herramienta de compresión como WinZip, o con una herramienta de construcción como Ant o Maven.

Interfaz ServletRequest

Para cada petición HTTP, el contenedor de servlets crea una instancia de la interfaz `ServletRequest` y la pasa al método `service` del servlet. `ServletRequest` encapsula información a cerca de la petición.

`getParameter` es el método más frecuentemente usado de `ServletRequest`. Su uso más común es el de devolver el valor de un campo de formulario HTML. También se usa para recuperar el valor de un par nombre/valor de la cadena de consulta.

Hay que observar `getParameter` devuelve un valor nulo si el parámetro no existe.

Además de `getParameter`, se puede también usar `getParameterNames`, `getParameterMap`, y `getParameterValues` para recuperar nombres y valores de campos de formulario o pares clave/valor de una cadena de consulta.

Interfaz ServletResponse

La interfaz `ServletResponse` representa la respuesta de un servlet. Antes de invocar al método `service` de un servlet, el contenedor de servlets crea una instancia de `ServletResponse` y la pasa como el segundo argumento al método `service`. `ServletResponse` oculta la complejidad del envío de la respuesta al navegador.

Uno de los métodos definidos en `ServletResponse` es el método `getWriter`, que devuelve un objeto `java.io.PrintWriter` que puede enviar texto al cliente.

Cuando se envía la respuesta al cliente, la mayoría de las veces es en formato HTML.

Antes de enviar cualquier etiqueta HTML, se debe establecer en tipo de contenido de la respuesta llamando al método `setContentType`, pasándole `"text/html"` como su argumento. Es así como se le dice al navegador que el tipo de contenido es HTML.

Interfaz ServletConfig

El contenedor de servlet pasa una instancia de la interfaz `ServletConfig` al método `init` del servlet cuando el contenedor inicializa el servlet. `ServletConfig` encapsula información de configuración que se puede paras al servlet a través de la anotación `@WebServlet` o el descriptor de despliegue. Cada pieza de información pasada de esta manera se le llama parámetro inicial. Un parámetro inicial tiene dos componentes: una clave y un valor.

Para recuperar el valor de un parámetro inicial desde dentro de un servlet se llama al método `getInitParameter` en la instancia `ServletConfig` pasada por el contenedor de servlets al método `init` del servlet.

```
String getInitParameter(String name)
```

Además, el método `getInitParameterNames` devuelve una instancia de `Enumeration` con todos los nombres de parámetros iniciales.

```
Enumeration<String> getInitParameterNames()
```

`ServletConfig` ofrece otro método útil, `getServletContext`. Este método se usa para recuperar la instancia `ServletContext` desde dentro del servlet.

Interfaz ServletContext

ServletContext representa la aplicación servlet. Solo existe un contexto por aplicación web. En un entorno distribuido donde una aplicación es desplegada simultáneamente en varios contenedores, hay un objeto ServletContext por Maquina Virtual Java.

Se puede obtener el ServletContext llamando al método getServletContext de ServletConfig.

ServletContext se utiliza para compartir información para que sea accedida por todos los recursos de la aplicación y para permitir el registro dinámico de objetos web. Lo primero se hace almacenando objetos en un Map interno dentro de ServletContext. Los objetos almacenados en ServletContext se denominan atributos.

Los siguientes métodos en ServletContext tratan los atributos:

```
Object getAtrribute(String name)
Enumeration<String> getAttributeNames()
void setAttribute(String name, Object Object)
void removeAttribute(String name)
```

GenericServlet

GenericServlet es una implementación abstracta de las interfaces Servlet y ServletConfig y realiza las siguientes tareas.

- Asigna el ServletConfig en el método init a un atributo de instancia de tal manera que se puede recupera llamando al método getServletConfig.
- Proporciona implementaciones por defecto para todos los métodos de la interfaz Servlet.
- Proporciona métodos que envuelven los métodos de la interfaz ServletConfig.

GenericServlet mantiene el objeto ServletConfig asignándolo a un atributo de instancia servletConfig en el método init.

No obstante, si se sobre escribe este método en un servlet, el método init en el servlet será llamado en su lugar y se tendrá que hacer una llamada super.init(servletConfig) para mantener el objeto ServletConfig. Para ahorra hace esto, GenericServlet proporciona un segundo método init, que no tiene parámetros. Este método se llamado por le primer método init después de que el objeto ServletConfig haya sido asignado al atributo servletConfig:

```
public void init(ServletConfig servletConfig) throws ServletException{
   this.serlvetConfig = servletConfig;
   this.init();
}
```

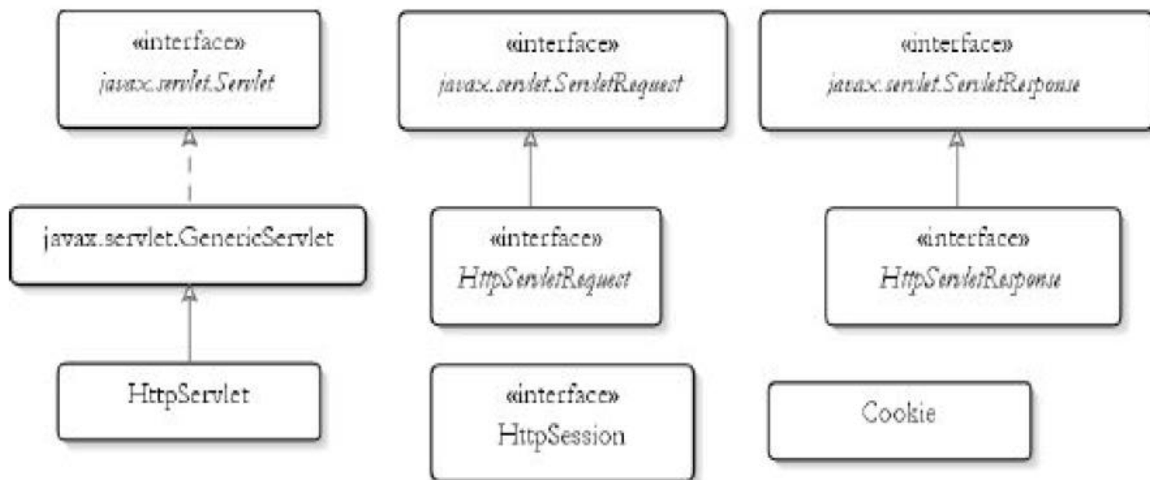
Esto significa, que se puede escribir código de inicialización sobre escribiendo el método init sin parámetros y el objeto ServletConfig se guardará en la instancia de GenericServlet.

Como se puede observar, extendiendo GenericServlet no es necesario sobre escribir métodos que no se quiere implementar. Como resultado, se tiene un código más limpio.

Aun que GenericServlet es un buen avance con respecto a la implementación manual de Servlet, se utiliza poco, ya que no tiene tantas ventajas como extender de HttpServlet.

HTTP Servlets

La mayoría de las aplicaciones servlets trabajan con HTTP. Esto significa que se puede utilizar las características que ofrece HTTP.



La clase `HttpServlet` sobre escribe la clase `GenericServlet`. Si se usa `HttpServlet`, también se trabaja con objetos `HttpServletRequest` y `HttpServletResponse` que representan la petición y la respuesta servlet, respectivamente. La interfaz `HttpServletRequest` extiende la interfaz `ServletRequest` y la interfaz `HttpServletResponse` extiende la interfaz `ServletResponse`.

`HttpServlet` sobre escribe el método service de `GenericServlet` y añade otro método service con la siguiente especificación:

```
protected void service(HttpSerlvetRequest request,
   HttpServletResponse response)
   throws ServletException, IOException
```

La diferencia entre el nuevo método `service` y el especificado por la interfaz `Servlet` es que la primera acepta objetos `HttpServletRequest` y `HttpServletResponse`, en lugar de los objetos `ServletRequest` y `ServletResponse`.

El contenedor de servlets, como siempre, llama al método `service` original en `Servlet`, el cual en `HttpServlet` esta implementado como sigue:

```
public void service(ServletRequest req, ServletResponse res)
   throws ServletException, IOException {
   HttpServletRequest request;
   HttpServletResponse response;
   try{
      request = (HttpServletRequest) req;
      response = (HttpServletResponse) res;
   catch(ClasCastException e) {
      throw new ServletException("non-HTTP request or response");
   }
   service(request, response);
}
```

La conversión explicita siempre dará un resultado correcto porque el contenedor de servlets siempre pasa un objeto `HttpServletRequest` y un objeto `HttpServletResponse` cuando llama al método `service` del servlet, como anticipación al uso de HTTP.

El nuevo método `service` en `HttpServlet` examina entonces el método HTTP usado para enviar la petición (llamando al `request.getMethod`) y llama a uno de los siguientes métodos: `doGet`, `doPost`, `doHead`, `doPut`, `doTrace`, `doOptions`, y `doDelete`. Cada uno de estos siete métodos representa los métodos del protocolo HTTP posibles. `doGet` y `doPost` son los más usados habitualmente. Como resultado, raramente se tiene que sobre escribir el método `service`. En su lugar, se tiene que sobre escribir `doGet` o `doPost` o ambos.

Para resumir, hay dos características en `HttpServlet` que no se encuentran en `GenericServlet`:

- En lugar del método service, se sobre escribirá `doGet`, `doPost` o ambos. En casos muy excepcionales también se sobre escribirán los otros métodos: `doHead`, `doPut`, `doTrace`, `doOptions`, `doDelete`.
- Ahora se trabaja con `HttpServletRequest` y `HttpServletResponse`, en lugar de `ServletRequest` y `ServletResponse`.

Interfaz `HttpServletRequest`

La interfaz `HttpServletRequest` representa la petición del servlet en el entorno HTTP. Extiende la interfaz `ServletRequest` a añade varios métodos. Algunos de los métodos añadidos son los siguientes:

`String getContextPath()`

Devuelve la porción de la URI de petición que indica el contexto de la aplicación.

`Cookie[] getCookies()`

Devuelve una matriz de objetos `Cookie`.

`String getHeader(String name)`

Devuelve el valor de la cabecera HTTP especificada.

`String getMethod()`

Devuelve el nombre del método HTTP con el cual se realizo la petición.

`String getQueryString()`

Devuelve la cadena de consulta en la URL de la petición.

`HttpSession getSession()`

Devuelve el objeto sesión asociado con esta petición. Si no se encuentra, se crea un nuevo objeto sesión.

`HttpSession getSession(boolean créate)`

Devuelve el objeto sesión actual asociado con esta petición. Si no se encuentra y el argumento create es `true`, se crea un nuevo objetos sesión.

Interfaz `HttpServletResponse`

La interfaz `HttpServletResponse` representa la respuesta servlet en un entorno HTTP. Algunos de los métodos definidos en el son:

```
void addCookie(Cookie cookie)
```

Añade una cookie a este objeto respuesta.

```
void addHeader(String name, String value)
```

Añade una cabecera a este objeto respuesta.

```
void sendRedirect(String location)
```

Envía una código de respuesta que re direcciona al navegador a la localización especificada.

Trabajar con Formularios HTML

Una aplicación web casi siempre contiene uno o más formularios HTML para recoger entradas del usuario. Se puede enviar fácilmente un formulario HTML desde un servlet al navegador. Cuando el usuario envía el formulario, los valores introducidos en los elementos del formulario son enviados al servidor como parámetros de la petición. El valor de un campo de entrada HTML (un campo de texto, un campo oculto, o un campo de contraseña) o un área de texto es enviado al servidor como una cadena de caracteres. Un campo de entrada o un área de texto vacío envían una cadena de caracteres vacía. Por lo tanto, `ServletRequest.getParameter` que coge el nombre de campo de entrada nunca devuelve `null`.

Un elemento de selección HTML también envía una cadena de caracteres al servidor. Si ninguna de las opciones es seleccionada en el elemento de selección, se envía el valor de la opción que se muestra actualmente.

Un elemento de selección múltiple (un elemento de selección que permite múltiples selecciones y se indica mediante `<select multiple>`) envía una matriz de cadenas de caracteres y tiene que ser tratada con `ServletRequest.getParameterValues`.

Un `checkbox` es un poco especial. Un `checkbox` chequeado envía una cadena `"on"` al servidor. Un `checkbox` no chequeado no envía nada al servidor y `ServletRequest.getParameter(fieldName)` devuelve `null`.

Los `radio buttons` envían el valor del botón seleccionado al servidor. Si ninguno de los botones esta seleccionado, no se envía nada al servidor y `ServletRequest.getParameter(fieldName)` devuelve `null`.

Si un formulario contiene múltiples elementos de entrada con el mismo nombre, todos los valores serán enviados y se tiene que usar `ServletRequest.getParameterValues` para recuperarlos. `ServletRequest.getParameter` solo devolverá el último elemento.

Uso del Descriptor de Despliegue

Un aspecto de despliegue es configurar las correspondencias de los servlets con una ruta. Hasta ahora, se mapeaba un servlet con una ruta usando la anotación `@WebServlet`. El uso del descriptor de despliegue es otra forma de configurar una aplicación servlet. El descriptor de despliegue siempre tiene el nombre `web.xml` y esta localizado bajo el directorio `WEB-INF`.

Hay algunas ventajas del uso del descriptor de despliegue. Por un lado, se puede incluir elementos que no tiene equivalente en la anotación `@WebServlet`, como el elemento `load-on-starup`. Este elemento carga el servlet en el momento de arrancar la aplicación, en lugar de cargarlo la primera vez que es llamado. Esto es especialmente útil si el método `init` del servlet tarda un poco en completarse.

El descriptor de despliegue también permite sobre escribir los valores especificados por una anotación servlet. Una anotación `@WebServlet` en un servlet que es también declarado en el descriptor de despliegue no tendrá efecto.

Gestión de Sesión

La gestión de sesión es un tema muy importante en el desarrollo de aplicaciones web. Esto es así debido al hecho de que HTTP es un protocolo sin estado. Un servidor web por defecto no sabe si una petición HTTP viene desde un usuario que se conecta por primera vez o uno que ya ha estado conectado.

Por ejemplo, en una aplicación web en la que el usuario necesita registrarse, una vez que el usuario introduces el nombre de usuario y la contraseña correctos, la aplicación no debería pedir al usuario que se registrada de nuevo para acceder a las diferentes partes de la aplicación. La aplicación necesita recordar que usuarios ya se han registrado correctamente. En otras palabras, necesita ser capaz de gestionar sesiones de usuario.

Objetos `HttpSession`

De las diferentes técnicas de gestión de sesión, los objetos `HttpSession` son los más potentes y versátiles. Un usuario puede tener uno o ningún objeto `HttpSession` y solo puede acceder a su propio objeto `HttpSession`. Un objeto `HttpSession` se crea automáticamente cuando un usuario visita una página por primera vez. Se recupera el objeto `HttpSession` de un usuario llamando al método `getSession` en el objeto `HttpServletRequest`. Hay dos métodos sobre cargados getSession:

```
HttpSession getSession()
HttpSession getSession(boolean créate)
```

El método sin argumentos devuelve el objeto `HttpSession` actual o crea y devuelve uno si no existe ninguno. `getSession(false)` devuelve el objeto `HttpSession` actual si existe uno o un valor nulo si no existe ninguno. `getSession(true)` devuelve el objeto `HttpSession` actual si existe uno o crea uno nuevo si no existe. `getSession(true)` es igual que `getSession()`.

El método `setAttribute` de `HttpSession` pone un valor en el objeto `HttpSession`. Su sintaxis es la siguiente:

```
void setAttribute(String name, Object value)
```

Un valor puesto en el objeto `HttpSession` se almacena en memoria. Por lo tanto, se debe solo almacenar objetos lo mas pequeños posible y no demasiados de ellos en la sesión. Y aunque los contenedores de servlets modernos son capaces de mover objetos en las sesiones al almacenamiento secundario cuando se va a quedar sin memoria, esto causa un problema de rendimiento.

Un valor añadido al objeto `HttpSession` no tiene por qué ser un `String` sino que pude ser cualquier objeto Java en tanto en cuanto implemente la interfaz `Serializable`, de tal forma que el objeto almacenado en la sesión pude ser serializado a un fichero o a una base de datos cuando el contenedor de servlets crea conveniente hacerlo. Se puede almacenar en sesión objetos no serializables, no obstante si el contenedor de servlets intentara serializarlo, fallaría y lanzaría una excepción.

El método `setAttribute` espera un nombre diferente para cada objeto diferente. Si se pasa un nombre de atributo que ha sido usado previamente, el nombre se desasociará del valor antiguo y se asociará al valor nuevo. Se puede recuperar un objeto almacenado en sesión llamando al método `getAttribute` en el objeto `HttpSession`, pasándole un nombre de atributo. La sintaxis de este método es la siguiente:

```
Object getAttribute(String name)
```

Otro método útil en HttpSession es getAttributeNames, que devuelve un objeto Enumeration para iterar todos los atributos en un objeto HttpSession.

```
Enumeration<String> getAttributeNames()
```

Observe que los valores almacenados en el objeto HttpSession no son enviados al cliente. En su lugar, el contendor de servlets genera un identificador único para cada objeto HttpSession que crea y envía este identificador como un dato al navegador, o bien en forma de cookies con nombre JSESSIONID o añadiéndolo a las URLs como un parámetro jsessionid. En subsiguientes peticiones, el navegador envía de vuelta este datos al servidor, que le dirá que usuario esta haciendo la petición. Cualquiera que sea la forma escogida por el contenedor de servlets para transmitir el identificador de sesión, esta ocurre de forma automática. Se puede recuperar el identificador del objeto HttpSession llamando al método getId en dicho objeto HttpSession.

```
String getId()
```

También hay un método invalidate definido en HttpSession. Este método fuerza a que la sesión expire y desenlaza todos los objetos enlazados a él. Por defecto, un objeto HttpSession espera después de algún periodo de inactividad del usuario. Se puede configurar tiempo de expiración para toda la aplicación a través del elementos sessiontimeout del descriptor de despliegue. Si este elemento no está configurado, el tiempo de expiración será determinado por el contenedor de servlets.

En la mayoría de los casos, se querrá destruir instancia HttpSession no usadas antes de su tiempo de expiración para poder liberar algo de memoria.

Se puede llamar al método getMaxInactiveInterval para averiguar cuanto tiempo más vivirá un objeto HttpSession después de la última visita del usuario. Este método devuelve el número de segundos que le queda de vida a la instancia HttpSession. El método setMaxInactiveInterval permite establecer un valor diferente para el tiempo de expiración para un objeto HttpSession individual.

```
void setMaxInactiveInterval(int seconds)
```

Pasando 0 a este método causa que el objeto HttpSession nunca expire. Esto no es una buena idea ya que la memoria dinámica ocupada por el objeto HttpSession nunca se liberará hasta que la aplicación sea descargada o se pare el contenedor de servlets.

JavaServer Pages

Existen dos inconvenientes que los servlets no son capaces de resolver. Primero, todas las etiquetas HTML escritas en un servlet deben ser encerradas en cadenas de caracteres Java, haciendo que mandar respuestas HTPP sea una tarea tediosa. Segundo, todas las etiquetas HTML están codificadas directamente en los servlets.

`JavaServer Pages` (`JSP`) resuelven estos dos problemas de los servlets. `JSP` no remplaza a los servlets. Más bien, lo complementa. Las aplicaciones web modernas Java usan tanto servlets como páginas `JSP`.

Visión General de JSP

Una página `JSP` es esencialmente un servlet. No obstante, trabajar con páginas `JSP` es más fácil que con servlet por dos razones. Primera, no se tiene que compilar las páginas `JSP`. Segunda, las páginas `JSP` son básicamente ficheros de texto con extensión `jsp` y se puede usar cualquier editor de texto para escribirlas. Las páginas `JSP` se ejecutan en un contendor `JSP`. Un contendor de servlets normalmente es también un contenedor `JSP`.

La primera vez que una página `JSP` es solicitada, un contenedor servlet/JSP hace dos cosas:

1. Traducir la página `JSP` en una clase implementadora de la página `JSP`, que es una clase Java que implementa la interfaz `JspPage` o su subinterfaz `HttpJspPage`. `JspPage` es, a su vez, una subinterfaz de `Servlet` y esto hace a toda página `JSP` un servlet. Si hubiera un error de traducción, se enviaría un mensaje de error al cliente.
2. Si la traducción fue correcta, el contenedor servlet/JSP compila la clase servlet. Después, el contenedor carga e instancia lo `bytecodes Java` y realiza las operaciones del ciclo de vida que normalmente hacen los servlets.

Para peticiones sub siguientes para la misma página `JSP`, el contenedor servlet/JSP comprueba si la página `JSP` ha sido modificada desde la última vez que fue traducida. Si es así, será traducida, compilada y ejecutada de nuevo. Si no, el servlet `JSP` que ya esta en memoria se ejecuta. De esta forma, la primera invocación a una página `JSP` siempre tarda más tiempo que las siguientes peticiones porque esto implica traducción y compilación. Para evitar este problema, se puede hace una de las dos siguientes acciones:

- Configurar la aplicación de tal forma que todas las páginas `JSP` sean llamadas (y, como efecto, traducidas y compiladas) cuando se arranca la aplicación, en lugar de en la primera petición.
- Pre compilar las páginas `JSP` y desplegarlas como servlets.

Una página puede contener plantillas de datos y elementos sintácticos. Un elemento sintáctico es algo con un significado especial para le traductor `JSP`. Por ejemplo, `<%` es un elemento sintáctico porque denota el comienzo de un bloque de código `Java` dentro de una página JSP. `%>` es también un elemento sintáctico porque denota la finalización de un bloque de código `Java`. Todo lo demás que no es un elemento sintáctico es plantilla de datos. Las plantillas de datos se envían tal cual al navegador. Por ejemplo, las etiquetas HTML y el texto en una página `JSP` son plantillad de datos.

Otro aspecto donde las páginas `JSP` son diferentes de un servlet es que las primera no necesitan ser anotadas o mapeadas a una URL en el descriptor de despliegue. Cada página `JSP` en el directorio de la aplicación puede ser invocada introduciendo la ruta de la página en el navegador. Ahora bien, si las páginas `JSP` están bajo el directorio `WEB-INF` no podrán ser accedidas de esta manera.

Hay dos cosas que resaltar. Primero, el código Java puede aparecer en cualquier lugar dentro de las páginas JSP siempre que este encerradas entre los elementos sintácticos <% y %>. Segundo, para importar un tipo Java usado en una página JSP, se usa el atributo import de la directiva @page. Sin importar un tipo Java, se tiene que escribir el nombre totalmente cualificado del tipo Java en el código.

El bloque <% … %> se denomina scriplet.

Objetos Implícitos

El contenedor de servlets pasa varios objetos al servlet que se esta ejecutando. Por ejemplo, Se tiene un objeto HttpServletRequest y un objeto HttpServletResponse en el método service del servlet y un objeto ServletConfig en el método init. Además, se puede obtener un objeto HttpSession llamando al método getSession en el objeto HttpServletRequest.

En una JSP se puede recupera estos objetos usando los objetos implícitos de la JSP:

Objeto	Tipo
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext
page	javax.servlet.jsp.HttpJspPage
exception	java.lang.Throwable

pageContext hace referencia a un objetos PageContext creado por la página. Proporciona información útil del contexto y acceso a varios objetos relacionados con el servlet por medio de métodos como, getRequest, getResponse, getServletContext, getServletConfig, y getSession. Estos métodos no son muy útiles en scriptlets ya que los objetos que devuelven se pueden acceder directamente a través de los objetos implícitos request, response, session, y application. No obstante, pageContext permite que estos objetos sean accedidos mediante el uso del EL (Expression Language).

Otro conjunto de métodos interesantes ofrecidos por PageContext son aquellos para obtener y establecer atributos, los métodos getAttribute y setAttribute. Los atributos pueden ser almacenados en uno de cuatro ámbitos: página, petición, sesión, y aplicación. El ámbito página es el ámbito más reducido y los atributos almacenados aquí solo están disponibles en la misma página JSP. El ámbito de petición hace referencia al actual ServletRequest, el ámbito sesión al HttpSession actual, y el ámbito aplicación al ServletContext.

El método setAttribute en PageContext tiene la siguiente definición:

```
public abstact void setAttribute(String name, Object value, int scope)
```

El valor de scope puede ser uno de las siguientes constantes de PageContext: PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE, y APPLICATION_SCOPE.

Para almacenar un atributo en el ámbito de página, se puede usar esta sobre carga de setAttribute:

```
public abstact void setAttribute(String name, Object value)
```

Por ejemplo, el siguiente scriptlet almacena un atributo en el objeto ServletRequest.

```
<%
   // product es un objeto Java
   pageContext.setAttribute("product", product,
      pageContext.REQUEST_SCOPE);
%>
```

El código anterior tiene el mismo efecto que este:

```
<%
   request.setAttribute("product", product);
%>
```

El objeto implícito out hace referencia a la clase JspWriter, que es parecido a PrintWriter que se obtiene llamando a getWriter en el objeto HttpServletResponse. Se puede llamar a sus métodos sobre cargados print como se haría con PrintWriter para mandar mensajes al navegador. Por ejemplo:

```
out.println("Bienvenido");
```

Hay que decir que por defecto el compilador JSP establece el tipo de contenido de una página JSP a text/html. Si se está mandando otro tipo de contenido diferente, se debe especificar llamando a response.setContentType o usando la directiva @page.

También hay que comentar que le objeto implícito page representa la página JSP actual y no se suele usar en la codificación de la página JSP.

Directivas

Las directivas son el primer tipo de elementos sintácticos JSP. Son instrucciones para el traductor de JSP que le dicen como se debe traducir la página JSP en un servlet. Hay varias directivas definidas en JSP, pero solo se va a dos de las más importantes, page e include.

La Directiva page

Se utiliza la directiva @page para dar instrucciones al traductor JSP en ciertos aspectos de la página JSP actual.

La directiva @page tiene la siguiente sintaxis:

```
<%@ page atributo1="valor1" atributo2="valor2" ... %>
```

El espacio entre @ y page es opcional y atributo1, atributo2, etc. son atributos de la directiva page. Algunos de ellos son:

- import. Especifica un tipo Java o varios tipos Java que serán importado y utilizable por el código Java en esta página. Se puede usar el carácter comodín * para importar un paquete entero. Todos los tipos en los siguientes paquetes son importados explícitamente: java.lang, javax.servlet, javax.servlet.http, javax.servlet.jsp.

- contentType. Especifica el tipo de contenido del objeto implícito response de este página. Por defecto, el valor es text/html.
- pageEncoding. Especifica la codificación de caracteres para la página. Por defecto el valor es ISO.8859-1.
- isELIgnored. Indica si la expresión EL (lenguaje de expresión) se ignoran.
- language. Especifica el lenguaje de scripting usado en la página. Por defecto su valor es java y es el único valor posible en JSP 2.2.

La directiva @page puede aparecer en cualquier lugar de la página. La excepción es cuando contiene los atributos contentType o pageEncoding. En estos casos debe aparecer antes de cualquier plantilla de datos y antes de enviar cualquier contenido que usa código Java. Esto es así porque el tipo de contenido y la codificación de caracteres debe estar establecida antes de enviar cualquier contenido.

La directiva page también puede aparecer múltiples veces. No obstante, un atributo que aparezca en varias directivas page debe tener el mismo valor. Una excepción a esta regla es el atributo import. El efecto del atributo import que aparece en varias directivas page es acumulativo. Por ejemplo, las siguientes directivas page importan tanto java.util.ArrayList como java.util.Date

```
<%@page import="java.util.ArrayList"%>
<%@page import="java.util.Date"%>
```

Esto es lo mismo que

```
<%@page import="java.util.ArrayList, java.util.Date"%>
```

La Directiva include

Se usa da directiva include para incluir contenido de otro fichero en la página JSP actual. Se puede usar varias directivas include en una página JPS.

La sintaxis de la directiva include es la siguiente:

```
<%@ include file="url"%>
```

donde el espacio entre @ e include es opcional y url representa la ruta relativa al fichero a incluir. Si url comienza por /, se interpreta como una ruta absoluta en el servidor. Si no existe, se interpreta como relativa a la página JSP actual.

El traductor JSP traduce la directiva include remplazando la directiva con el contenido del fichero a incluir.

Por convención un fichero incluido tiene una extensión jspf, que significa fragmento o segmento JSP.

Hay que decir que se puede incluir también fichero HTML estáticos.

Elementos de Scripting

El segundo tipo de elementos sintácticos JSP, los elementos de scripting incorporan código Java a una página JSP. Existen tres tipos de elementos de scripting: scriptlets, declaraciones, y expresiones.

Scriptlets

Un scriptlet es un bloque de código Java. Un scriptlet comienza con <% y termina con %>.

Hay que decir que las variables definidas en un scriptlet son visibles en otro scriptlet debajo de él.

Expresiones

Una `expresión` es evaluada y su resultado proporcionado al método `print` del objeto implícito `out`. Una expresión comienza con <%= y termina con %>. Por ejemplo:

```
Hoy es <%=java.util.Calendar.getInstance().getTime()%>
```

Hay que decir que no se tiene que poner el punto y coma detrás de la expresión.

Con esta `expresión`, el contenedor JSP evalúa `java.util.Calendar.getInstance.getTime()`, y después pasa el resultado a `out.print()`. Esto es lo mismo que es siguiente scriptlet:

```
Hoy es
   <%
         out.print(java.util.Calendar.getInstance().getTime());
   %>
```

Declaraciones

Se puede declarar variables y métodos que pueden ser usados en una página JSP. La declaración va encerrada entre <%! y %>.

```
<%!public java.util.Date getFechaHoy(){
   return new java.util.Date();
}%>
<html>
<head>
<title>Declaraciones</title>
</head>
<body>
   Hoy es
   <%=getFechaHoy()%>
</body>
</html>
```

Una declaración puede aparecer en cualquier lugar dentro de una página JSP y puede haber múltiples declaraciones en la misma página.

Se puede usar declaraciones para sobre escribir los métodos init y `destroy` en la clase implementadora. Para sobre escribir `init`, se declara el método `jspInit`. Para sobre escribir `destroy`, se declara el método `jspDestroy`.

- `jspInit`. Este método es similar al método `init` en Servlet. `jspInit` es invocado cuando se inicializa la página JSP. A diferencia del método `init`, `jspInit` no toma argumentos. Se puede aún obtener el objeto `ServletConfig` a través del objeto implícito `config`.
- `jspDestroy`. Este método es similar al método `destroy` en Servlet y es invocado cuando la página JSP se va a destruir.

```
<%!
   public void jspInit(){
      System.out.println("jspInit ...");
   }
```

```
    public void jspDestroy(){
       System.out.println("jspDestroy ...");
    }
%>
```

Inhabilitar los Elementos de Scripting

Con la introducción de EL en JSP, la practica recomendada es usar EL para acceder a los objetos del lado del servidor y no escribir código Java en la páginas JSP.

Por esta razón, a partir de JSP 2.0 los elementos de scripting se pueden inhabilitar definiendo un elemento <scripting-invalid> dentro del elemento <jsp-propertygroup> en el descriptor de despliegue.

```
<jsp-property-group>
   <url-patter>*.jsp</url-pattern>
   <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

Acciones

Las acciones son el tercer tipo de los elementos sintácticos. Se traducen en código Java que realiza una operación, como el acceso a un objeto Java o la invocación de un método. Se va a ver acciones estándares que deben ser soportados por todos los contenedores JSP. Además de estas acciones estándar, también se puede crear etiquetas particularizadas para realizar ciertas operaciones.

useBean

Esta acción crea una variable de scripting asociada con un objeto Java. Fue uno de los primero esfuerzos por separar la lógica de presentación y la de negocio. Ahora, gracias a otras tecnologías como la etiquetas particularizadas y EL, useBean ya casi no se usa.

```
<html>
<head>
<title>
</head>
<body>
   <jsp:useBean id="hoy" class="java.util.Date" />
   <%=hoy%>
</body>
</html>
```

setProperty y getProperty

La acción setProperty establece una propiedad de un objeto Java y getProperty imprime la propiedad de un objeto Java.

```
<jsp:useBean id="empleado" class="es.insa.formacion.entidades.Empleado"/>
<jsp:setProperty name="empleado" property="nombre" value="Antonio"/>
Nombre: <jsp:getProperty name="empleado" property="nombre" />
```

include

La acción include se usa para incluir otro recurso dinámicamente. Se puede incluir otra página JSP, un servlet, o una página HTML estática.

```
<html>
<head>
<title>Include action</title>
</head>
<body>
   <jsp:include page="jspf/menú.jsp">
      <jsp:param name="text" value="¿Que tal?" />
   </jsp:include>
</body>
</html>
```

Es importante comprender la diferencia entre la directiva include y la acción include. Con la directiva include, la inclusión ocurre en el momento de traducción de la página, es decir, cuando el contenedor JSP traduce la página al servlet generado. Con la acción include, la inclusión ocurre en tiempo de petición. Por esto, se puede pasar parámetros usando una acción include, pero no una directiva include.

La segunda diferencia es que con la directiva include, la extensión del fichero del recurso a incluir no tiene importancia. Con la acción include, la extensión del fichero debe ser jsp para que sea procesado como una página JSP. Usar jspf en la acción include, por ejemplo, hará que el segmento JSP sea tratado como un fichero estático.

forward

La acción forward relanza la página actual a un recurso diferente.

```
<jsp:forward page="jspf/login.jsp">
   <jsp:param name="text" value="Please login" />
</jsp:forward>
```

Lenguaje de Expresión EL

El lenguaje de expresión (EL) permite acceder a los datos de aplicación desde las páginas JSP. EL esta diseñado para hacer posible y facilitar la creación de páginas JSP libres de script, es decir, páginas que no usan declaraciones, expresiones o scriptlets JSP.

Sintaxis del Lenguaje de Expresión

Las expresiones EL comienzan con ${ y terminan con }. El constructor de una expresión EL es el siguiente:

```
${expresión}
```

La secuencia de caracteres ${, denota el comienzo de una expresión EL. Si se quiere enviar el literal ${, necesitar utilizar una secuencia de escape para el primer carácter: \${.

Palabras reservadas

Las siguientes palabras son reservadas y no se deben usar como identificadores:

```
andeq    gt    true  instanceof
or ne    le    false empty
notlt    ge    null  div   mod
```

Los operadores [] y .

Una expresión EL puede devolver cualquier tipo de dato. Si una expresión EL da como resultado un objeto que tiene una propiedad, puedes utilizar los operadores [] o . para acceder a la propiedad. Estos dos operadores tiene una función similar; [] es una forma más generalizada, pero . proporciona una forma resumida.

Para acceder a una propiedad de un objeto, puedes usar una de las siguientes formas:

```
${objeto["propiedad"]}
${objeto.propiedad}
```

No obstante, solo se puede usar el operador [] si propiedad no es un nombre de variable Java valido. Por ejemplo para acceder a la cabecera accept-language solo se puede usar el operador [] ya que accept-languaje no es un nombre de variable Java valido.

Para acceder a propiedad de un objeto que a su vez es una propiedad de otro objeto, se utiliza de forma recursiva los operadores.

```
${objeto["objetointerior"]["propiedad"]}
${objeto. Objetointerior.propiedad}
```

La regla de evaluación

Una expresión EL se evalúa de izquierda a derecha. Para una expresión de la forma expr-a[expr-b] la forma de evaluación es la siguiente:

1. Se evalúa expr-a para obtener valor-a.
2. Si valor-a es null, se devuelve null.
3. Se evalúa expr-b para obtener valor-b.

4. Si `valor-b` es `null`, se devuelve `null`.
5. Si el tipo de dato de `valor-b` es `java.util.Map`, se comprueba que `valor-b` es una clave en `Map`. Si es así, se devuelve `valor-a.get(valor-b)`. Si no, se devuelve `null`.
6. Si el tipo de dato de `valor-b` es `java.util.List` o una matriz, se hace lo siguiente:
    a. Se convierte `valor-b` a `int`. Si la conversión falla, se lanza una excepción.
    b. Si `valor-a.get(valor-b)` lanza `IndexOutOfBoundsException` o si `Array.get(valor-a, valor-b)` lanza `ArrayIndexOutOfBoundsException`, se devuelve `null`.
    c. En caso contrario, se devuelve `valor-a.get(valor-b)` si `valor-a` es una `List`, o `Array.get(valor-a, valor-b)` si `valor-a` es una matriz.
7. Si `valor-a` no es un `Map`, una `List`, o un array, `valor-a` debe ser un JavaBean. En este caso, se convierte `valor-b` a un `String`. Si `valor-b` es una propiedad legible de `valor-a`, se llama al método `getter` de la propiedad y se devuelve el valor de la propiedad. Si el método `getter` lanza una excepción, la expresión es inválida.

Acceso a JavaBeans

Se puede usar tanto el operador `[]` como el operador `.` para acceder a las propiedades de un bean:

```
${nombreBean["nombrePropiedad"] }
${nombreBean.nombrePropiedad}
```

Si la propiedad es un objeto que a su vez tiene una propiedad, se puede acceder a la propiedad del segundo objeto utilizando los operadores de forma recursiva.

Objetos Implícitos

Al igual que las páginas JSP tiene sus objetos implícitos que pueden ser accedidos directamente desde scripts JSP, EL proporciona sus propios objetos implícitos para ser accedidos desde una expresión EL. La siguiente tabla lista los objetos implícitos de EL:

Objeto	Descripción
pageContext	El `javax.servlet.jsp.PageContext` para la JSP actual
initParam	Un `Map` que contiene todos los parámetros de inicialización del contexto (aplicación) con los nombres de los parámetros como claves.
param	Un `Map` que contiene todos los parámetros de la petición con los nombre de los parámetros como claves. El valor de cada clave es el primer valor del parámetro del nombre especificado. Por ello, si hay dos parámetros de petición con el mismo nombre, solo el primero podre ser recuperado usando el objeto `param`.
paramValues	Un `Map` que contiene todos los parámetros de la petición con los nombre de los parámetros como claves. El valor de cada clave es un array de cadenas que contiene todos los valores para el nombre del parámetro específico. Si el parámetro solo tiene un valor, se devuelve un array con un solo elemento.
header	Un `Map` que contiene las cabeceras de la petición con los nombres de las cabeceras como claves. El valor de cada clava es el primer valor de cabecera del nombre de cabecera especificado.

Objeto	Descripción
	Si una cabecera tiene más de un valor, solo se devuelve el primer valor.
headerValues	Un Map que contiene todos las cabeceras de la petición con los nombre de las cabeceras como claves. El valor de cada clave es un array de cadenas que contiene todos los valores para el nombre de la cabecera específico. Si la cabecera solo tiene un valor, se devuelve un array con un solo elemento.
cookies	Un Map que contiene todos los objetos Cookies en el objeto petición actual. Los nombres de las cookies son las claves del Map, y cada clave se corresponde con un objeto Cookie.
applicationScope	Un Map que contiene todos los atributos en el objeto ServletContex con los nombres de los atributos como claves.
sessionScope	Un Map que contiene todos los atributos en el objeto HttpSession con los nombres de los atributos como claves.
requestScope	Un Map que contiene todos los atributos en el objeto HttpServletRequest con los nombres de los atributos como claves.
pageScope	Un Map que contiene todos los atributos en el ámbito de la página con los nombres de los atributos como claves.

pageContext

El objeto pageContext representa el objeto javax.servlet.jsp.PageContext para la JSP actual. Contiene todos los objetos implícitos JSP, que son los dados en la siguiente lista.

Objeto	Objeto referenciado
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext
page	javax.servlet.jsp.HttpJspPage
exception	java.lang.Throwable

Por ejemplo, para obtener el objeto HttpServletRequest actual se puede usar alguna de las siguientes expresiones.

```
${pageContext["request"]}
${pageContext.request}
```

Y, el método de la petición se puede obtener usando alguna de las siguientes expresiones:

```
${pageContext["request"]["method"]}
```

```
${pageContext["request"].method}
${pageContext.request["method"]}
${pageContext.request.method}
```

Los parámetros de petición son accedidos con más frecuencia que otros objetos implícitos; por esto, se proporcionan dos objetos implícitos, `param` y `paramValues`.

`initParam`

El objeto implícito `initParam` se usa para recuperar el valor de un parámetro de contexto. Por ejemplo, para acceder a un parámetro de contexto nombrado `password`, se utiliza la siguiente expresión:

```
${intiParam.password}
```

o

```
${intiParam["password"]}
```

`param`

El objeto implícito `param` se usa para recuperar un parámetro de petición. Este objeto representa un `Map` que contiene todos los parámetros de petición. Por ejemplo, para recuperar el parámetro llamado `userName`, se usa una de las siguientes expresiones:

```
${param.userName}
```

```
${param["userName"]}
```

`paramValues`

Se usa el objeto implícito `paramValues` para recuperar los valores de un parámetro de petición. Este objeto representa un `Map` que contiene todos los parámetros de la petición con los nombres de los parámetros como claves. El valor de cada clave es una matriz de cadenas de caracteres que contiene todos los valores para un nombre de parámetro específico. Si el parámetro solo tiene un valor, también se devuelve una matriz con un solo elemento. Por ejemplo, para obtener el primer y segundo valor del parámetro `selectedOption` se usa las siguientes expresiones:

```
${paramValues.selectedOptions[0]}
${paramValues.selectedOptions[1]}
```

`header`

El objeto implícito `header` representa un `Map` que contiene todas las cabeceras de la petición. Para recuperar un valor de una cabecera, se usa el nombre de la cabecera como clave. Por ejemplo, para recuperar el valor de la cabecera `accept-language`, se usa la siguiente expresión:

```
${header.["accept-language"]}
```

Si el nombre de la cabecera es un nombre de variable Java válido, como `connection`, también se puede utilizar el operador `.`:

```
${header.connection}
```

`headerValues`

El objeto implícito `headerValues` representa un `Map` que contiene todas las cabeceras de la petición con los nombres de las cabeceras como claves. A diferencia de `header`, el `Map` devuelto por el objeto implícito `headerValues` devuelve una matriz de cadenas de caracteres. Por ejemplo, para obtener el primer valor

de la cabecera accept-language, se usa la siguiente expresión:

```
${header.["accept-language"][0]}
```

cookie

Se usa el objeto implícito cookie para recuperar una cookie. Este objeto representa un Map que contiene todas las cookies en el objeto HttpServletRequest actual. Por ejemplo, para recuperar el valor de una cookie denominada jsessionid, se usa la siguiente expresión:

```
${cookie.jsessionid.value}
```

Para obtener la ruta de la cookie jsessionid, se usa la siguiente expresión:

```
${cookie.jsessionid.path}
```

applicationScope, sessionScope, requestScope, y pageScope

Se usa el objeto implícito applicationScope para obtener el valor de una variable del ámbito de aplicación. Por ejemplo, si se tiene una variable del ámbito de aplicación denominada miVariable, se puede usar la siguiente expresión para acceder al atributo:

```
${applicationScope.miVariable}
```

En la programación sevlet/JSP un objeto se dice que tiene un ámbito si esta colocado como atributo en alguno de los siguientes objetos: PageContext, ServletRequest, HttpSession, o ServletContext. Los objetos implícitos sessionScope, requestScope, y pageScope son similares a appliactionScope pero su ámbito son sesión, petición, y página, respectivamente.

Un objeto de un ámbito determinado también puede ser accedido en una expresión EL sin indicar el ámbito. En este caso, el contenedor JSP devolverá el primer objeto con nombre igual en el PageContext, ServletRequest, HttpSession o ServletContext. La búsqueda del objeto se realiza desde en el ámbito más reducido (PageContext) al más amplio (ServletContext). Por ejemplo, la siguiente expresión devolverá el objeto referenciado por hoy en cualquier ámbito:

```
${hoy}
```

Otros operadores EL

EL proporciona otros operadores: operadores aritméticos, operadores relacionales, operadores lógicos, el operador condicional, y el operador empty. No obstante, como el objetivo de EL es facilitar la construcción de JSP libres de scripts, estos operadores EL son de uso limitado, excepto el operador condicional.

Operadores Aritméticos

Son cinco:

- Adición (+)
- Substracción (-)
- Multiplicación (*)
- División (/ y div)
- Resto/modulo (% y mod)

La precedencia de los operadores aritméticos es la siguiente, de mayor a menor:

```
* / div % mod
```

+ -

Operadores relacionales

La siguiente es la lista de operadores relacionales:

- Igualdad (== y eq)
- Desigualdad (¡= y ne)
- Mayor que (> y gt)
- Mayor o igual que (>= y ge)
- Menor que (< y lt)
- Menor o igual que (<= y le)

Operadores lógicos

Hay tres operadores lógicos:

- Y (&& y and)
- O (|| y or)
- NO (! y not)

Operador condicional

El operador condicional tiene la siguiente sintaxis:

```
${sentencia?A:B}
```

Si sentencia se evalúa a verdadero, la salida de la expresión es A. En caos contrario, la salida es B.

El operador empty

El operador empty se usa para examinar si un valor es null o vacío. En el ejemplo siguiente:

```
${empty X}
```

Si X es null o si X es una cadena de longitud cero. La expresión devuelve verdadero. También devuelve verdadero si X es un Map, un array, o una colección vacía.

Configuración de EL en las versiones 2.0 y posteriores

JSP 2.0 u posteriores proporciona la posibilidad de inhabilitar el scripting en todas las páginas JSP. Por lo tanto se puede forzar a escribir páginas JSP libres de scripts. También se puede inhabilitar EL en las aplicaciones.

Inhabilitación de scripts en JSP

Para inhabilitar la posibilidad de escribir scripts JSP en la página, se usa el elemento jsp-property-group con dos subelementos: url-pattern y scripting-invalid, que establece en el descriptor de despliegue. El elemento url-pattern define el patrón URL al cual se aplica la inhabilitación de scripts.

```
<jsp-config>
   <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <scripting-invalid>true</scripting-invalid>
   </jsp-property-group>
</jsp-config>
```

Solo puede haber un elemento `jsp-config` en el descriptor de despliegue.

Desactivar la evaluación EL

Hay dos formas para desactivar la evaluación EL en una JSP.

Primero, puedes establecer a `true` el atributo `isELIgnored` de la directiva page:

```
<%@page isELIgnored="true"%>
```

El valor por defecto para `isELIgnored` es `false`. El uso del atributo `isELIgnored` es recomendado si se quiere desactivar la evaluación EL en una o unas pocas páginas.

Segundo, se puede usar el elemento `jsp-property-group` en el descriptor de despliegue. Este elemento va acompañado con los dos subelementos `url-pattern` y `el-ignore`. . El elemento `url-pattern` define el patrón URL al cual se aplica la desactivación EL.

```
<jsp-config>
   <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <el-ignore>true</el-ignore>
   </jsp-property-group>
</jsp-config>
```

El valor establecido en el descriptor de despliegue tiene precedencia sobre el establecido en la página.

Por otro lado, si usas el descriptor de despliegue que cumple con especificación `2.3` de servlet y anteriores, la evaluación `El` esta ya desactivas por defecto.

JSTL

La Librería de Etiquetas Estándar de JavaServer Pages (JSTL JavaServer Pages Standard Tag Library) es una colección de librerías de etiquetas particularizadas para resolver problemas comunes como iterar sobre un mapa, comprobaciones condicionales o incluso manipulación de datos.

JSTL esta actualmente en la versión 1.2. Esta compuesto por dos partes, el API JSTL y la implementación JSTL. Para poder utilizar JSTL en una aplicación web se tiene que poner los dos componentes (se pueden encontrar en un solo jar) bajo el directorio WEB-INF/lib.

Para usar la librería JSTL en una página JSP, de tiene que usar la directiva JSP taglib, cuyo formato es el siguiente:

```
<%@ taglib uri="uri" prefix="prefijo" %>
```

Por ejemplo, para usar la librería Core, se tiene poner la siguiente declaración al principio de la página JSP:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Acciones de Propósito General

Existen tres acciones de propósito general en la librería Core que se usan para manipular variables con diferentes ámbitos: out, set, remove.

La Etiqueta out

La etiqueta out evalúa una expresión y saca el resultado al JspWriter actual. La sintaxis de out tiene dos formas, con y sin cuerpo.

```
<c:out value="valor" [escapeXml="{true|false}"]
   [default="valorPorDefecto"]/>
<c:out value="valor" [escapeXml="{true|false}"]>
   valor por defecto
</c:out>
```

El cuerpo para out es JSP. La lista de atributos es:

Atributo	Tipo	Descripción
value*+	Objeto	Expresión a ser evaluada.
escapeXml+	boolean	Indica si los caracteres <, >, &, ', y " en el resultado serán convertidos a sus correspondientes códigos de entidad de caracteres, ej. < a <, etc.
default+	Objeto	Valor por defecto

Por ejemplo, la siguiente etiqueta out imprime el valor de la variable x:

```
<c:out value="${x}"/>
```

Por defecto, out codifica los caracteres especiales <, >, ', ", y & a sus correspondientes códigos de entidad de carácter <, >, ', ", and &, respectivamente.

Antes de JSP 2.0, la etiqueta out era la forma más fácil de imprimir el valor de un objeto. En JSP 2.0 o posterior, a menos que se necesite un valor con secuencia de escape XML, se puede usar si ningún problema la expresión EL:

```
${x}
```

El atributo default en out permite asignar un valor por defecto que se mostrará si la expresión EL asignada a su atributo value devuelve null. Al atributo defaul se le puede asignar un valor dinámico. Si este valor dinámico devuelve null, la etiqueta out mostrará una cadena de caracteres vacía.

Por ejemplo, en la siguiente etiqueta out, si la variable MiVar no se encuentra en HttpSession, el valor de la variable MiVar en el ámbito de aplicación será mostrada. Si esta última no se encuentra, se enviara a la salida una cadena de caracteres vacía.

```
<c:out value="${sessionScope.miVar}"
   default="${applicationScope.miVar"/>
```

La Etiqueta set

Se puede utilizar la etiqueta set para hacer lo siguiente:

1. Crear una cadena de caracteres y una variable que referencia a la cadena en alguno de los ámbitos.
2. Crear una variable el algún ámbito que referencie a un objeto de algún ámbito.
3. Establecer la propiedad de un objeto en algún ámbito.

Si set se usa para crear una variable en algún ámbito, la variable se puede utilizar en la misma página JSP después de la aparición de la etiqueta.

La sintaxis de la etiqueta set tiene cuatro formas. La primera forma se usa para crear una variable en algún ámbito en la cual el atributo value especifica la cadena de caracteres a ser creada o un objeto de algún ámbito ya existente.

```
<c:set value="value" var="varName"
   [scope="{page|request|session|application}"]/>
```

Donde el atributo scope especifica el ámbito de la variable.

La segunda forma es similar a la primera, excepto que la cadena de caracteres a crear o el objeto de algún ámbito a referenciar se pasan como contenido del cuerpo.

```
<c:set var="varName" [scope="{page|request|session|application}"]>
   body content
</c:set>
```

La segunda forma permite tener código JSP en el contenido del cuerpo.

La tercera forma establece el valor de una propiedad de un objeto de algún ámbito. El atributo target especifica el objeto de algún ámbito y el atributo property la propiedad del objeto. El valor a asignar a la propiedad es especificado por el atributo value.

```
<c:set target="target" property="propertyName" value="value"/>
```

Hay que decir que se debe usar una expresión EL en el atributo target para referenciar el objeto de algún ámbito.

La cuarta forma es similar a la tercera forma, pero el valor asignado se pasa como contenido del cuerpo.

```
<c:set target="target" property="propertyName">
   body content
</c:set>
```

Atributo	Tipo	Descripción
value+	Object	La cadena a ser creada, o el objeto de algún ámbito a referenciar, o el nuevo valor de una propiedad.

Atributo	Tipo	Descripción
var	String	La variable a ser creada.
scope	String	El ámbito de la variable creada.
target+	Object	El objeto de algún ámbito a cuya propiedad se le asignara un nuevo valor; este debe ser una instancia JavaBean o un objeto java.util.Map.
property+	String	El nombre de la propiedad a la que se asigna un nuevo valor.

La Etiqueta remove

Se puede usar la etiqueta remove para eliminar una variable en algún ámbito.

```
<c:remove var="varName"
   [scope="{page|request|session|application}"]/>
```

Hay que observare que el objeto referenciado por la variable no es eliminado. Por lo tanto, si otra variable de algún ámbito también esta referenciando al mismo objeto, se puede acceder aún al objeto a través de la segunda variable.

La lista de atributos de la etiqueta remove son los siguientes:

Atributo	Tipo	Descripción
var	String	El nombre de la variable de algún ámbito a eliminar.
scope	String	El ámbito de la variable a eliminar.

Como ejemplo, la siguiente etiqueta remove elimina la variable job del ámbito de página.

```
<c:remove var="job" scope="page"/>
```

Acciones Condicionales

Las acciones condicionales se utilizan para tratar con situaciones en las cuales la salida de una página depende de el valor de cierta entrada, lo que en Java se resuelve utilizando if, if … else, y la sentencia switch.

Existen cuatro etiquetas para realizar acciones condicionales en JSTL: if, choose, when, y otherwise.

La Etiqueta if

La etiqueta if comprueba una condición y procesa el contenido de su cuerpo si la condición se evalúa a verdadera. El resultado de la comprobación se almacena en un objeto Boolean, y se crea una variable en un ámbito para referenciar al objeto de tipo Boolean. Se puede especificar el nombre de la variable creada en el ámbito usando el atributo var y el ámbito en el atributo scope.

La sintaxis de if tiene dos formas. La primera forma no tiene contenido en el cuerpo:

```
<c:if test="testCondition" var="varName"
   [scope="{page|request|session|application}"]/>
```

En este caso, normalmente el objeto especificado por var será comprobado por otra etiqueta posteriormente en la misma JSP.

La segunda forma se usa con un contenido en el cuerpo:

```
<c:if test="testCondition" [var="varName"]
      [scope="{page|request|session|application}"]>
      body content
```

```
</c:if>
```

El contenido del cuerpo es JSP y será procesado si la condición se evalúa a verdadero. La siguiente lista muestra los atributos de la etiqueta `if`.

Atributo	Tipo	Descripción
test+	Boolean	La condición a comprobar que determina si el contenido del cuerpo existente se debe procesar.
var	String	El nombre de la variable de algún ámbito que referencia el valor de la condición; el tipo de var es Boolean.
scope	String	El ámbito de la variable especificada por var.

Para similar un `else`, se usa dos etiquetas `if` con condiciones opuestas.

Las Etiquetas choose, when y otherwise

Las etiquetas `choose` y `when` actúan de forma parecida a `switch` y `case` en `Java`. Se usan para proporcionar el contexto para una ejecución condicional mutuamente excluyente. La etiqueta `choose` debe tener uno o más etiquetas `when` anidadas en ella, y cada etiqueta `when` representa un caso que puede ser evaluado y procesado. La etiqueta `otherwise` se usa para un bloque condicional por defecto que será procesado si ninguna condición de las etiquetas `when` se evalúa a verdadero. Si esta presente, la etiqueta `otherwise` debe aparecer después del último `when`.

Las etiquetas `choose` y `otherwise` no tiene atributos. La etiqueta `when` debe tener un atributo `test` especificando la condición que determina si el contenido del cuerpo se debe procesar.

Como ejemplo, el siguiente código comprueba el valor de un parámetro denominado `status`. Si el valor de `status` es `pleno`, se muestra `"Eres un miembro pleno"`. Si el valor es `estudiante`, se muestra `"Eres miembro estudiante"`. Si el parámetro `status` no existe o su valor no es `pleno` ni `estudiante`, el código no muestra nada.

```
<c:choose>
   <c:when test="${param.status=='pleno'}">
      Eres un miembro pleno
   </c:when>
   <c:when test="${param.status==estudiante'}">
      Eres un miembro estudiante
   </c:when>
</c:choose>
```

El siguiente ejemplo es parecido al anterior, pero se utiliza la etiqueta `otherwise` para mostrar `"Por favor, registrese"` si el parámetro `status` no existe o si su valor no es `pleno` ni `estudiante`.

```
<c:choose>
   <c:when test="${param.status=='pleno'}">
      Eres un miembro pleno
   </c:when>
   <c:when test="${param.status==estudiante'}">
      Eres un miembro estudiante
   </c:when>
   <c:otherwise>
      Por favor, registrese
   </c:otherwise>
</c:choose>
```

Acciones de Iteración

Las acciones de iteración son útiles cuando se necesita iterar un número de veces o sobre una colección de objetos. JSTL proporciona dos etiquetas que realizan acciones de iteración, forEach y forToken.

La Etiqueta forEach

La etiqueta forEach itera el contenido del cuerpo un número de veces o itera sobre una colección de objetos. Los objetos sobre los que se puede iterar incluyen todas las implementaciones de java.util.Collection y java.util.Map, y matrices de objetos o tipos primitivos. También se puede iterar sobre un java.util.Iterator y java.util.Enumeration, pero no se debe utilizar Iterator o Enumeration en más de una acción porque ni Iterator ni Enumerator serán reinicializados.

Atributo	Tipo	Descripción
var	String	El nombre de la variable que referencia el elemento actual de la iteración.
items+	Cualquier tipo soportado.	Colecciones de objetos sobre los que iterar.
varStatus	String	El nombre de la variable que mantiene el estado de la iteración. El valor es del tipo javax.servlet.jsp.jstl.core.LoopTagStatus.
begin+	int	Si se especifica items, la iteración comienza en el elemento localizado en el índice especificado, donde el primer elemento de la colección tiene el índice 0. Si items no se especifica, la iteración comienza con el índice establecido en el valor especificado. Si se especifica, el valor debe comenzar en cero o mayor.
end+	int	Si el elemento items se especifica, la iteración terminara en el elemento localizado en el índice especificado (incluido). Si no se especifica items, la iteración termina cuando índice alcance el valor especificado.
step+	int	La iteración procesará cada elemento de la colección, comenzando por el primero. Si está presente, el valor de step debe ser mayor o igual que 1.

La sintaxis d la etiqueta forEach tiene dos formas. La primera forma es para repetir el contenido del cuerpo un número fijo de veces:

```
<c:forEach [var="varName"] begin="begin" end="end" step="step">
   body content
</c:forEach>
```

La segunda forma se usa para iterar sobre una colección de objetos:

```
<c:forEach items="collection" [var="varName"]
   [varStatus="varStatusName"] [begin="begin"] [end="end"]
   [step="step"]>
      body content
</c:forEach>
```

El contenido del cuerpo es JSP.

Por ejemplo, la siguiente etiqueta forEach muestra "1, 2, 3,4, 5".

```
<c:forEach var="x" begin="1" end="5">
   <c:out value="${x}"/>,
</c:forEach>
```

Y, la siguiente etiqueta forEach itera sobre la propiedad teléfono de una variable dirección.

```
<c:forEach var="telefono" items="${direccion.telefono}">
   ${telefono}"<br/>
</c:forEach>
```

Para cada iteración, la etiqueta forEach crea una variable cuyo nombre se especifica por el atributo var. La variable solo esta disponible entre las etiquetas de comienzo y fin de forEach, será eliminada justo antes de la etiqueta de cierre de forEach.

La etiqueta forEach tiene una variable del tipo javax.servlet.jsp.jstl.core.LoopTagStatus, varStatus. La interfaz LoopTagStatus tiene una propiedad count que devuelve la "cuenta" de la iteración actual. El valor de status.count es uno para la primera iteración, 2 para la segunda iteración, etc. Por ejemplo, comprobando el resto de count%2, se puede saber si la etiqueta está procesando un elemento con índice par o impar.

Se puede también usar forEach para iterar sobre un mapa. Se hace referencia a una clave y un valor de un mapa utilizando las propiedades key y value, respectivamente. El código para iterar sobre un mapa es el siguiente:

```
<c:forEach var="mapItem" items="map">
   ${mapItem.key} : ${mapItem.value}
</c:forEach>
```

La Etiqueta forTokens

Se usa la etiqueta forTokens para iterar sobre sobre los elementos que están separados por un delimitador especifico. La sintaxis para esta acción es la siguiente:

```
<c:forTokens items="stringOfTokens" delims="delimiters"
   [var="varName"] [varStatus="varStatusName"]
   [begin="begin"] [end="end"] [step="step"]>
      body content
</c:forTokens>
```

El contenido del cuerpo es JSP. La lista de atributos de la etiqueta forTokens es la siguiente:

Atributo	Tipo	Descripción
var	String	El nombre de la variable que hace referencia al elemento actual de la iteración.
items+	String	La cadena de caracteres de elementos sobre los que iterar.
varStatus	String	Nombre de la variable que mantiene los datos de estado de la iteración. El valor es de tipo javax.servlet.jsp.jstl.core.LoopTagStatus.
begin+	int	El índice de comienzo de las iteraciones, donde el índice comienza en cero. Si se especifica, begin debe ser 0 o mayor.
end+	int	El índice final de la iteración, donde el índice comienza en cero.
step+	int	La iteración procesará cada elemento de la cadena de caracteres, comenzando por el primero. Si está presente, el valor de step debe ser mayor o igual que 1.
delims+	String	El conjunto de delimitadores.

Un ejemplo de forTokem:

```
<c:forTokens var="item" items="Argentina,Brazil,Chile" delims=",">
   <c:out value="${item}"/><br/>
</c:forTokens>
```

Acciones de Formateado

JSTL proporciona etiquetas para formatear y convertir números y fechas. Las etiquetas son formatNumber, formatDate, timeZone, setTimeZone, parseNumber, y parseDate.

La Etiqueta formatNumber

Se usa formatNumber para formatear números. La etiqueta proporciona la flexibilidad de usar diversos atributos para obtener el formato que se adapte a las necesidades. La sintaxis de formatNumber tiene dos formas. La primera se usa sin contenido en el cuerpo.

```
<fmt:formatNumber value="numericValue"
   [type="{number|currency|percent}"]
   [pattern="patronParticular"]
   [currencyCode="codigoMoneda"]
   [currencySymbol="simboloMoneda"]
   [groupingUsed="{true|false}"]
   [maxIntegerDigits="maxDigitosParteEntera"]
   [minIntegerDigits="minDigitosParteEntera"]
   [maxFractionDigits="maxDigitosParteDecimal"]
   [minFractionDigits="minDigitosParteDecimal"]
   [var="nombreVar"]
   [scope="{page|request|session|application}"]
/>
```

La segunda forma se usa con contenido en el cuerpo:

```
<fmt:formatNumber [type="{number|currency|percent}"]
   pattern="patronParticular"]
   [currencyCode="codigoMoneda"]
   [currencySymbol="simboloMoneda"]
   [groupingUsed="{true|false}"]
   [maxIntegerDigits="maxDigitosParteEntera"]
   [minIntegerDigits="minDigitosParteEntera"]
   [maxFractionDigits="maxDigitosParteDecimal"]
   [minFractionDigits="minDigitosParteDecimal"]
   [var="nombreVar"]
   [scope="{page|request|session|application}"]>
      numeric value to be formatted
</fmt:formatNumber>
```

El contenido del cuerpo es JSP. Los atributos de formatNumber son:

Atributo	Tipo	Descripción
value+	String o Number	Valor numérico a formatear.
type+	String	Indica si el valor va ha ser formateado como número, moneda o porcentaje. El valor de este atributo es uno de los siguientes: number, currency, percent.
pattern+	String	Patrón de formato particularizado.
currencyCode+	String	Código ISO 4217.
currencySymbol+	String	Símbolo de moneda.
groupingUsed+	Boolean	Indica si la salida contendrá separador de millares.

Atributo	Tipo	Descripción
maxIntegerDigits+	int	Número máximo de dígitos en la parte entera de la salida.
minIntegerDigits+	int	Número mínimo de dígitos en la parte entera de la salida.
maxFractionDigits+	int	Número máximo de dígitos en la parte decimal de la salida.
minFractionDigits+	int	Número mínimo de dígitos en la parte decimal de la salida.
var	String	El nombre de la variable para almacenar la salida como un String.
scope	String	El ámbito de var. Si el atributo scope esta presente, se debe especificar el atributo var.

Uno de los usos de formatNumber es para formatear números como moneda. Para ello, se puede especificar el atributo currencyCode para especificar el código ISO 4217 de moneda. Algunos de estos códigos son los siguientes.

Moneda	Código ISO 4217	Nombre de unidad mayor	Nombre de unidad menor
Dólar canadiense	CAD	dollar	cent
Yuan chino	CNY	yuan	jiao
Euro	EUR	euro	euro-cent
Yen japonés	JPY	yen	sen
Libra esterlina	GBP	pound	pence
Dólar Americano	USD	dollar	cent

Algunos ejemplos del uso de formatNumber:

Acción	Resultado
<fmt:formatNumber value="12" type="number"/>	12
<fmt:formatNumber value="12" type="number" minIntegerDigits="3"/>	012
<fmt:formatNumber value="12" type="number"minFractionDigits="2"/>	12.00
<fmt:formatNumber value="123456.78" pattern=".000"/>	123456.780
<fmt:formatNumber value="123456.78" pattern="#,#00.0#"/>	123,456.78
<fmt:formatNumber value="12" type="currency"/>	$12.00
<fmt:formatNumber value="12" type="currency" currencyCode="GBP"/>	GBP 12.00
<fmt:formatNumber value="0.12" type="percent"/>	12%
<fmt:formatNumber value="0.125" type="percent" minFractionDigits="2"/>	12.50%

Cuando se formatea monedas, si el atributo currencyCode no se especifica, se usa el idioma del navegador.

La Etiqueta formatDate

Se utiliza la etiqueta formatDate para formatear fechas. La sintaxis es la siguiente:

```
<fmt:formatDate value="fecha"
   [type="{time|date|both}"]
   [dateStyle="{default|short|medium|long|full}"]
   [timeStyle="{default|short|medium|long|full}"]
   [pattern="patronParticular"]
   [timeZone="ZonaHoraria"]
```

```
    [var="nombreVar"]
    [scope="{page|request|session|application}"]
/>
```

El contenido del cuerpo es JSP. Los atributos de la etiqueta formatDate son:

Atributo	Tipo	Descripción
value+	java.util.Date	Fecha y/o hora a ser formateada.
type+	String	Indica si la hora, la fecha, o ambos componentes van ha ser formateados.
dateStyle+	String	Estilo de formateado predefinido para fechas que sigue la semántica definida en java.text.DateFormat.
TimeStyle+	String	Estilo de formateado predefinido para las horas que sigue la semántica definida en java.text.DateFormat.
Pattern+	String	Patrón de formateado particularizado.
timezone+	String o java.util.TimeZone	Zona horaria en la que representar la hora.
var	String	Nombre de la variable en la que almacenar el resultado como una cadena de caracteres.
scope	String	Ámbito de var.

Hay distintos valores para el atributo timeZone.

El siguiente código usa la etiqueta formatDate para formatear el objeto java.util.Date referenciado por la variable now.

```
Default: <fmt:formatDate value="${now}"/>
Short: <fmt:formatDate value="${now}" dateStyle="short"/>
Medium: <fmt:formatDate value="${now}" dateStyle="medium"/>
Long: <fmt:formatDate value="${now}" dateStyle="long"/>
Full: <fmt:formatDate value="${now}" dateStyle="full"/>
```

El siguiente ejemplo de la etiqueta formatDate se usa para formatear horas.

```
Default: <fmt:formatDate type="time" value="${now}"/>
Short: <fmt:formatDate type="time" value="${now}" timeStyle="short"/>
Medium: <fmt:formatDate type="time" value="${now}"
   timeStyle="medium"/>
Long: <fmt:formatDate type="time" value="${now}" timeStyle="long"/>
Full: <fmt:formatDate type="time" value="${now}" timeStyle="full"/>
```

El siguiente ejemplo de la etiqueta formatDate se usa para formatear tanto fechas como horas.

```
Default: <fmt:formatDate type="both" value="${now}"/>
Short date short time: <fmt:formatDate type="both"
value="${now}" dateStyle="short" timeStyle="short"/>
Long date long time format: <fmt:formatDate type="both"
value="${now}" dateStyle="long" timeStyle="long"/>
```

El siguiente ejemplo de la etiqueta formatDate se usa para formatear horas con zona horaria.

```
Time zone CT: <fmt:formatDate type="time" value="${now}"
   timeZone="CT"/><br/>
Time zone HST: <fmt:formatDate type="time" value="${now}"
   timeZone="HST"/><br/>
```

El siguiente ejemplo de la etiqueta formatDate se usa para formatear tanto fechas como horas usando un patrón particularizado.

```
<fmt:formatDate type="both" value="${now}" pattern="dd.MM.yy"/>
<fmt:formatDate type="both" value="${now}" pattern="dd.MM.yyyy"/>
```

La Etiqueta timeZone

La etiqueta timeZone se usa para especificar la zona horaria en la cual la información horaria en su contenido del cuerpo tiene que ser formateada o analizada.

```
<fmt:timeZone value="timeZone">
   body content
</fmt:timeZone>
```

El contenido del cuerpo es JSP. Al atributo value se le puede pasar un valor dinámico de tipo String o java.util.TimeZone.

Si el atributo value es null o vacío, se usa la zona horaria GMT.

```
<fmt:timeZone value="GMT+1:00">
   <fmt:formatDate value="${now}" type="both"
      dateStyle="full" timeStyle="full"/>
</fmt:timeZone>
```

La Etiqueta setTimeZone

Se usa la etiqueta setTimeZone para almacenar la zona horaria especificada en una variable o la variable de configuración horaria. La sintaxis de setTimezone es la siguiente:

```
<fmt:setTimeZone value="timeZone" [var="varName"]
   [scope="{page|request|session|application}"]
/>
```

Atributo	Tipo	Descripción
value+	String o java.util.TimeZone	La zona horaria.
var	String	Nombre de la variable que contendrá la zona horaria de tipo java.util.TimeZone
scope	String	El ámbito de var o de la variable de configuración horaria.

La etiqueta parseNumber

Se usa parseNumber para analizar una representación en forma de cadena de un número, una moneda, o un porcentaje en formato sensible al lenguaje en un número. La sintaxis tiene dos formas. La primera se usa sin contenido de cuerpo.

```
<fmt:parseNumber value="numericValue"
   [type="{number|currency|percent}"]
   [pattern="customPattern"]
   [parseLocale="parseLocale"]
   [integerOnly="{true|false}"]
   [var="varName"]
   [scope="{page|request|session|application}"]
/>
```

La segunda forma se usa con contenido en el cuerpo:

```
<fmt:parseNumber [type="{number|currency|percent}"]
   [pattern="customPattern"]
   [parseLocale="parseLocale"]
```

```
   [integerOnly="{true|false}"]
   [var="varName"]
   [scope="{page|request|session|application}"]>
      numeric value to be parsed
</fmt:parseNumber>
```

El contenido del cuerpo es JSP. Los atributos de la etiqueta `parseNumber` son:

Atributo	Tipo	Descripción
value+	String	Cadena de caracteres a ser analizada.
type+	String	Indica si la cadena de caracteres a ser analizada va a ser analizada como un número, una moneda, o un porcentaje.
pattern+	String	Patrón de formato particularizado que determina como la cadena de caracteres del atributo `value` tiene que ser analizada.
parseLocale+	String o java.util.Locale	Localización cuyo patrón de formato por defeco va a ser usado durante la operación de análisis, o al cual el patrón especificado por medio del atributo `pattern` es aplicado.
integerOnly+	Boolean	Indica si la solo la parte entera del valor dado debes ser analizada.
var	String	Nombre de la variable que mantendrá el resultado.
scope	String	Ámbito de `var`.

```
<fmt:parseNumber var="formattedNumber" type="number" value="${quantity}"/>
```

La Etiqueta parseDate

La etiqueta `parseDate` analiza una representación en forma de cadena de caracteres de una fecha y hora en un formato sensible al lenguaje. La sintaxis tiene dos formas. La primera forma se usa sin contenido en el cuerpo.

```
<fmt:parseDate value="dateString"
   [type="{time|date|both}"]
   [dateStyle="{default|short|medium|long|full}"]
   [timeStyle="{default|short|medium|long|full}"]
   [pattern="customPattern"]
   [timeZone="timeZone"]
   [parseLocale="parseLocale"]
   [var="varName"]
   [scope="{page|request|session|application}"]
/>
```

La segunda forma se usa con contenido en el cuerpo:

```
<fmt:parseDate [type="{time|date|both}"]
   [dateStyle="{default|short|medium|long|full}"]
   [timeStyle="{default|short|medium|long|full}"]
   [pattern="customPattern"]
   [timeZone="timeZone"]
   [parseLocale="parseLocale"]
   [var="varName"]
   [scope="{page|request|session|application}"]>
      date value to be parsed
</fmt:parseDate>
```

El contenido del cuerpo es JSP. La lista de atributos es la siguiente:

Atributo	Tipo	Descripción
value+	String	Cadena de caracteres a ser analizada.
type+	String	Indica si la cadena de caracteres a ser analizada va a ser analizada como un número, una moneda, o un porcentaje.
dateStyle+	String	Estilo de formato de la fecha.
timeStyle+	String	Estilo de formato de la hora.
pattern+	String	Patrón de formato particularizado que determina como la cadena de caracteres del atributo value tiene que ser analizada.
timeZone+	String o java.util.TimeZone	Zona horaria en la cual interpretar cualquier información horaria en la cadena de caracteres que representa la fecha.
parseLocale+	String o java.util.Locale	Localización en la cual se usara el patrón de formato por defecto cuando se realice la operación de análisis, o al cual se aplicara el patrón especificado por medio del atributo pattern.
var	String	Nombre de la variable que mantendrá el resultado.
scope	String	Ámbito de var.

```
<c:set var="myDate" value="12/12/2005"/>
<fmt:parseDate var="formattedDate" type="date"
   dateStyle="short" value="${myDate}"/>
```

Filtros

Un filtro es un objeto que intercepta una petición y procesa el objeto `ServletRequest` o `ServletResponse` pasado al recurso que ha sido pedido. Los filtros se puede usar para registro de trazas, encriptación y de encriptación, comprobación de sesión, etc. Los filtros se pueden configurar para interceptar uno o múltiples recursos.

La configuración de los filtros se puede realizar mediante anotaciones o mediante el descriptor de despliegue. Si varios filtros son aplicados a un mismo recurso o el mismo conjunto de recursos, el orden de invocación puede ser significativo y en este caso se necesita configurarlos en el descriptor de despliegue.

El API `Filter`

Un filtro debe implementar la interfaz `javax.servlet.Filter`. Esta interfaz expone tres métodos de ciclo de vida para un filtro: `init`, `doFilter`, y `destroy`.

`init`

El método `init` es llamado por le contenedor de servlet cuando un filtro va a ser puesto en servicio, es decir, cuando se arranca la aplicación. En otras palabras, el método `init` no espera hasta que un recurso asociado con él es invocado. Este método solo es llamado una vez y debe contener código de inicialización para el filtro. La sintaxis de método es la siguiente:

```
void init(FilterConfig filterConfig)
```

El contenedor de servlet pasa un objeto `FilterConfig` al método `init`.

`doFilter`

El método `doFilter` es llamado por el contenedor de servlet cada vez que se invoca un recurso asociado con el filtro. Este método recibe un objeto `ServletRequest`, uno `ServletResponse`, y uno `FilterChain`.

```
void doFilter(ServletRequest request, ServletResponse response,FilterChain filterChain)
```

Al recibir un objeto `ServletRequest` y `ServletResponse` se puede añadir algún atributo a `ServletRequest` o alguna cabecera a `ServletResponse`.

La última línea de código en la implementación del método `doFilter` debe ser una llamada al método `doChain` sobre el objeto `FilterChain` pasado como tercer argumento a `doFilter`.

```
filterChain.doFilter(request, response)
```

Un recurso puede estar asociado con varios filtros (formado una cadena de filtros), y `FilterChain.doFilter()` básicamente provoca que sea invocado el siguiente filtro en la cadena. La llamada de este método en el último filtro de la cadena provoca que se invoque al propio recurso pedido.

El método `doFilter` de `FilterChain` solo tiene dos parámetros en lugar de tres.

`destroy`

El último método en el ciclo de vida de un filtro es `destroy`.

```
void destroy()
```

Este método es invocado por el contenedor de servlet antes de que el filtro se saque de servicio, que normalmente es cuando se para la aplicación.

A menos que una clase de tipo filtro sea declarada en varios elementos `filter` en el descriptor de despliegue, el contenedor de servlet solo creará una única instancia de cada tipo de filtro. Ya que una aplicación Servlet/JSP normalmente es una aplicación multiusuario, una instancia de filtro puede ser accedida por múltiples hilos de ejecución al mismo tiempo y es necesario manejar con cuidado los temas multi-hilos.

Configuración de Filtros

La configuración de filtros tiene los siguientes objetivos:

- Determinar que recursos el filtro intercepta.
- Establecer valores iniciales a ser pasados al método `init` de filtro.
- Dar un nombre al filtro. Esto puede ser útil para ver el orden de invocación de los filtros.

La interfaz `FilterConfig` permite acceder al `ServletContext` a través de su método `getServletContext`.

Si un nombre tiene nombre, el método `getFilterName` de `FilterConfig` devuelve el nombre.

Se pueden pasar parámetros iniciales al filtro. Existen dos métodos que se puede usar para manejar los parámetros iniciales, el primero tiene la siguiente sintaxis:

```
java.util.Enumeration<java.lang.String> getInitParameterNames()
```

Este método devuelve un objeto `Enumeration` de los nombres de los parámetros del filtro. Si no existen parámetros iniciales para el filtro, este método devuelve un objeto `Enumeration` vacío.

El segundo método para tratar los parámetros iniciales tiene la siguiente sintaxis:

```
java.lang.String getInitParameter(java.lang.String parameterName)
```

Hay dos maneras de configurar un filtro. Se puede configurar un filtro usando la anotación `@WebFilter` a nivel de tipo o registrándolo en el descriptor de despliegue.

La anotación `@WebFilter` tiene los siguientes atributos, todos opcionales:

Atributo	**Tipo**	**Descripción**
`asyncSupported`	`Boolean`	Especifica si el filtro soporta modo de operación asíncrono.
`description`	`String`	Descripción del filtro.
`dispatcherTypes`	`DispatcherType[]`	Los tipos de dispatchers a los que se aplica el filtro
`displayName`	`String`	El nombre de visualización del filtro
`filterName`	`String`	Nombre del filtro
`initParams`	`WebInitParam[]`	Parámetros iniciales
`largeIcon`	`String`	Nombre del icono grande para el filtro
`servletNames`	`String[]`	Nombres de los sevlets a los que se aplica el servlet
`smallIcon`	`String`	Nombre de icono pequeño para el filtro
`urlPatterns`	`String[]`	Patrón URL a que se aplica el filtro
`value`	`String[]`	Patrón URL a que se aplica el filtro

Por ejemplo, el siguiente ejemplo asigna el nombre `MiFiltro` al filtro y se aplica a todos los recursos.

```
@WebFilter(filterName="DataCompressionFilter", urlPatterns={"/*"})
```

Esto es equivalente a la declaración de los elementos `filter` y `filter-mapping` el descriptor de desplie-

gue.

```
<filter>
   <filter-name>DataCompressionFilter</filter-name>
   <filter-class>
      the fully-qualified name of the filter class
   </filter-class>
</filter>
<filter-mapping>
   <filter-name>DataCompresionFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Otro ejemplo, este especificando parámetros iniciales:

```
@WebFilter(filterName="Security Filter", urlPatterns={"/*"},
   initParams = {
      @WebInitParam(name="frequency", value="1909"),
      @WebInitParam(name="resolution", value="1024")
   }
)
```

Su equivalente en el descriptor de despliegue.

```
<filter>
   <filter-name>Security Filter</filter-name>
   <filter-class>filterClass</filter-class>
   <init-param>
      <param-name>frequency</param-name>
      <param-value>1909</param-value>
   </init-param>
   <init-param>
      <param-name>resolution</param-name>
      <param-value>1024</param-value>
   </init-param>
</filter>
<filter-mapping>
   <filter-name>DataCompresionFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

Orden de los Filtros

Si tienes varios filtros aplicados al mismo recurso y el orden de invocación es importante, se tiene que usar el descriptor de despliegue para gestionar que filtro debe ser invocado primero. Por ejemplo, si `Filtro1` tiene que ser invocado antes de `Filtro2`, la declaración del `filter-mapping` del `Filtro1` debe aparecer antes de la declaración `filter-mapping` del `Filtro2` en el descriptor de despliegue.

```
<filter-mapping>
   <filter-name>Filtro1</filter-name>
   <url-pattern>
      algún patron
   </url-pattern>
</ filter-mapping >
< filter-mapping >
   <filter-name>Filtro2</filter-name>
```

```
      <url-pattern>
         algún patron
      </url-pattern>
   </ filter-mapping >
```

No es posible gestionar el orden de invocación de filtros sin el descriptor de despliegue.