

Java Enterprise Edition

Desarrollo de aplicaciones web con JEE 6

Thierry GROUSSARD

colección
Recursos Informáticos



Resumen

Este libro de JEE 6 se dirige a **desarrolladores Java** y proporciona todas las bases para usar el lenguaje Java en el **desarrollo de aplicaciones web dinámicas**.

Tras la **presentación de la plataforma JEE**, el lector descubrirá el funcionamiento del **protocolo http**, omnipresente en las aplicaciones web. El siguiente capítulo presenta los **servlets**, base de todas las aplicaciones web desarrolladas en lenguaje java. A continuación, se detalla la presentación de los datos al usuario con las páginas jsp. Los dos capítulos siguientes detallan las soluciones disponibles para **mejorar la productividad** usando las etiquetas de la librería **JSTL** o creando una librería de etiquetas propia, adaptada a las necesidades de la aplicación.

Ya que la mayoría de las aplicaciones usa actualmente una base de datos para almacenar la información, el último capítulo presenta **el acceso a los datos** con java mediante la **api jdbc** y las particularidades de las aplicaciones web en ese ámbito.

En este libro, cada concepto se ilustra con **ejemplos concretos**, que pueden adaptarse fácilmente según las necesidades. La estructura de los principales **archivos de configuración** se documenta como guía para realizar el despliegue de la aplicación.

Los ejemplos citados en el libro se pueden descargar en la web www.ediciones-eni.com.

Capítulos del libro : Prefacio – Presentación de JEE – El protocolo HTTP – Servlets – Uso de sesiones – Las páginas JSP – La librería JSTL – Etiquetas JSP personalizadas – JDBC – Anexos

El autor

Después de dedicarse al análisis y desarrollado durante más de 10 años, **Thierry Groussard** se orientó hacia la formación y, más en particular, en el terreno del desarrollo. Su profundo conocimiento de las necesidades de la empresa y sus cualidades pedagógicas hacen que sus obras estén particularmente adaptadas al aprendizaje y puesta en práctica del desarrollo en Java.

Este libro ha sido concebido y se difunde respetando los derechos de autor. Todas las marcas citadas han sido registradas por su editor respectivo. Reservados todos los derechos. El contenido de esta obra está protegido por la Ley, que establece penas de prisión y/o multas, además de las correspondientes indemnizaciones por daños y perjuicios, para quienes reprodujeren, plagiaren, distribuyeren o comunicaren públicamente, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución artística fijada en cualquier tipo de soporte o comunicada a través de cualquier medio, sin la preceptiva autorización.

Este libro digital integra varias medidas de protección, entre las que hay un marcado con su identificador en las imágenes principales

Introducción

Desde su creación por SUN en 1999, la plataforma JEE (originalmente J2EE) se ha convertido en un elemento esencial en el desarrollo de aplicaciones Web. El elemento principal del éxito de JEE es sin duda la ausencia de restricciones relativas al uso de una herramienta de desarrollo en particular o a la utilización de un tipo concreto de servidor. Elegir JEE para desarrollar una aplicación Web es tener a su disposición decenas de herramientas accesibles para el desarrollo así como un gran número de servidores a su elección para el despliegue de la aplicación. Muchos de ellos provienen del mundo Open Source, con lo que el coste de una aplicación se reduce considerablemente. La elección de una herramienta de desarrollo o de un servidor no es irreversible ya que es muy fácil migrar sin afectar al trabajo ya realizado.

La creación de aplicaciones Web con la plataforma JEE requiere el conocimiento de las bases del lenguaje java y del lenguaje HTML. Hay varias obras disponibles en esta misma colección que tratan estas materias.

Esta obra no trata todas las tecnologías de la plataforma JEE, un solo libro no sería ni por asomo suficiente para abordarlas todas. Se centra primeramente en los dos pilares fundamentales, los servlets y las páginas JSP que son indispensables para la creación de aplicaciones Web. El uso de la biblioteca JSTL y la creación de etiquetas personalizadas, a continuación, le permitirán mejorar su productividad.

Las bases de datos son elementos indispensables de una aplicación. El último capítulo de esta obra presenta el uso de la API JDBC para la comunicación con una base de datos.

Cada nuevo concepto se ilustra con varios ejemplos que pueden adaptarse fácilmente a sus necesidades y de este modo usarse en sus aplicaciones.

Introducción

La tecnología JEE (*Java Enterprise Edition*) constituye la solución propuesta por Sun para el desarrollo de aplicaciones distribuidas. La base de esta solución se sustenta en el lenguaje Java, también creado por Sun. Este lenguaje básico también es conocido con el término JSE (*Java Standard Edition*).

De hecho, JEE no es una mejora de JSE. La versión JSE es suficientemente rica por sí sola. JEE puede más bien ser considerada como una normativa que describe todos los elementos que constituyen e intervienen para el funcionamiento de una aplicación distribuida.

Define por ejemplo los elementos siguientes:

- Cómo deben desarrollarse los diferentes componentes de una aplicación (servlet, páginas JSP...).
- Cómo estos diferentes componentes pueden comunicarse entre ellos o con otras aplicaciones (JDBC, jndi, JavaMail...).
- Cómo deben organizarse estos componentes para construir una aplicación (descriptor de despliegue).
- Las restricciones que tienen que respetar los servidores encargados de albergar estas aplicaciones.

El cumplimiento de esta normativa permite a bastantes sociedades desarrollar sus propios servidores que serán capaces de hospedar cualquier aplicación que también respete esta normativa. Varias decenas de servidores distintos están disponibles actualmente con un rendimiento y una capacidad diferentes y, por consiguiente, con un precio diferente (desde gratuito hasta varias decenas de miles de euros).

La gran ventaja de JEE con relación a tecnologías propietarias reside en que en esta inmensa selección cabe la posibilidad de evolucionar hacia otro servidor con más rendimiento sin tener que realizar grandes modificaciones en la aplicación.

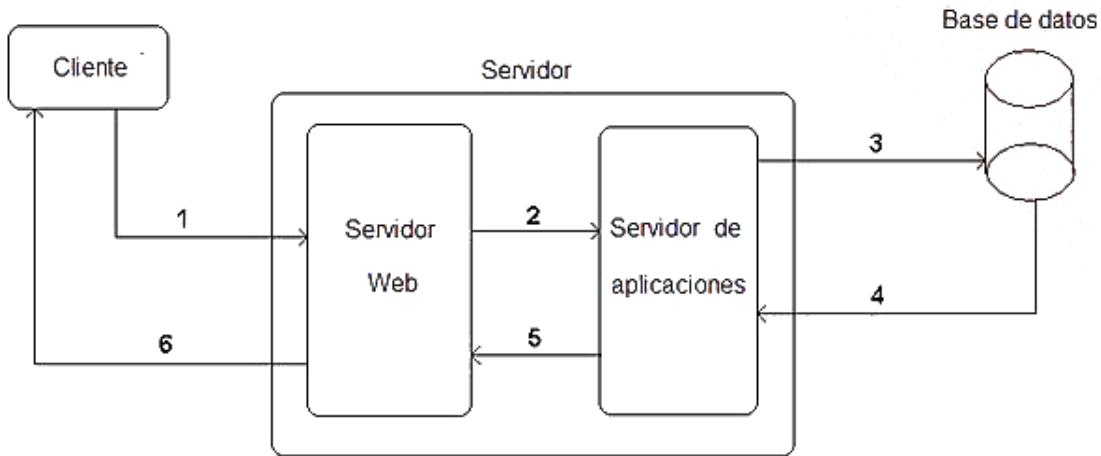
Servidores Web y servidores de aplicaciones

En ocasiones reina una notable confusión entre los términos servidor Web y servidor de aplicaciones y sin embargo hay una diferencia importante entre ambos elementos.

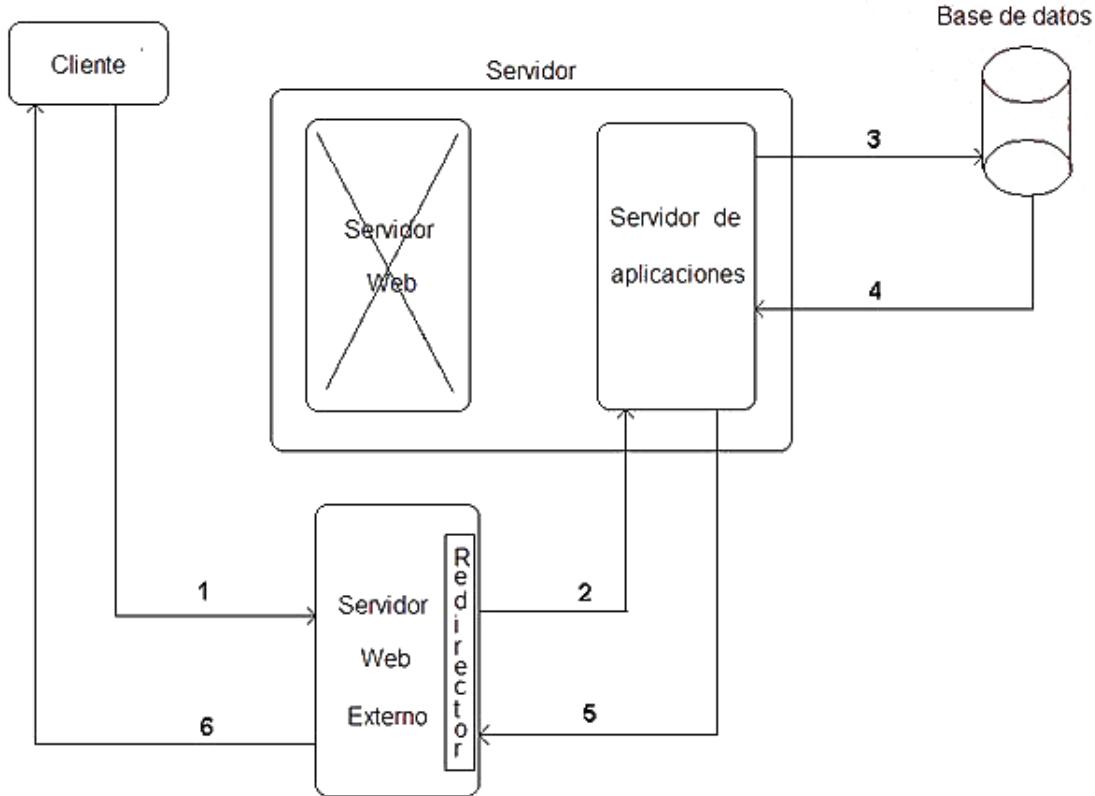
Un servidor Web no es más que un simple servidor de archivos. Los clientes se dirigen a éste mediante el protocolo HTTP para obtener un recurso. Cuando el servidor Web recibe una petición HTTP, extrae simplemente de la petición el nombre del recurso solicitado, lo busca en el disco y "lo envuelve" dentro de una respuesta HTTP para transmitirlo al cliente. Éste es el único trabajo que puede realizar un servidor Web. Un servidor Web no realiza ningún tratamiento en el recurso antes de transmitirlo al cliente. Por lo tanto, puede transmitir de manera indiferente a un cliente una página HTML, una imagen, un archivo de sonido o incluso un archivo ejecutable. El tipo de contenido del recurso solicitado le es totalmente indiferente.

La función de un servidor de aplicaciones es radicalmente distinta ya que los recursos que le son confiados no son simples archivos estáticos, sino que contienen el código que se va a encargar de ejecutar en nombre de los clientes que realicen la petición. Cuando un servidor de aplicaciones recibe una solicitud HTTP, éste también analiza la petición para determinar qué recurso se le ha solicitado. Generalmente, la petición concierne código ejecutable alojado en el servidor. Contrariamente a lo que haría un servidor Web en la misma situación, no transfiere al cliente el código sino que lo ejecuta y es el resultado de la ejecución de este código lo que se reenvía al cliente.

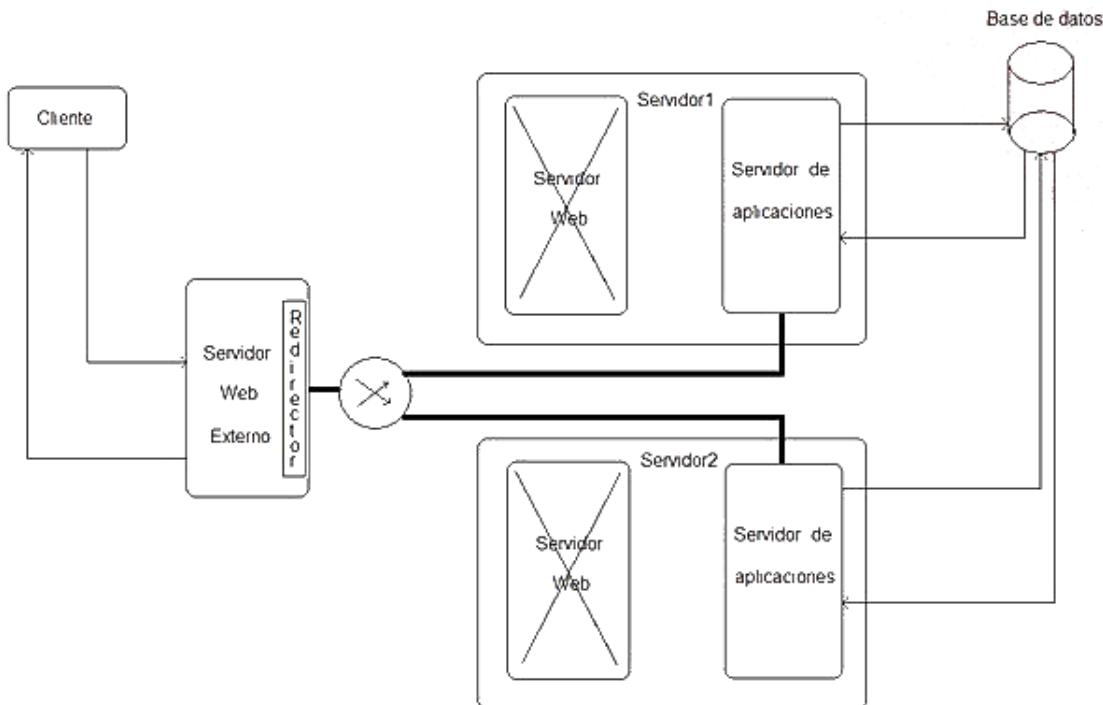
De hecho, la confusión se produce frecuentemente entre estos dos elementos porque generalmente un servidor de aplicaciones toma también las funciones de servidor Web. Cuando el servidor recibe una petición HTTP proveniente del exterior, es la parte del servidor Web la que recibe esta petición y la analiza. Si concierne a un recurso estático, el servidor Web realiza su función yendo a buscar el recurso y reenviándolo al cliente en una respuesta HTTP. Si la petición concierne a un recurso dinámico (código), el servidor Web no sabe tratar esta petición, por lo que la transfiere a la parte correspondiente al servidor de aplicaciones del servidor. Éste realiza su función ejecutando el código correspondiente y generando una respuesta HTTP. Si así lo requiriera, el servidor de aplicaciones puede contactar con otro servidor o una base de datos para poder construir la respuesta. Esta respuesta HTTP se transmite al servidor Web que a su vez se encarga de reenviarla al cliente.



La parte correspondiente al servidor Web de un servidor de aplicaciones suele tener menos rendimiento que un servidor Web dedicado. En ocasiones es posible reemplazarla por un servidor Web de verdad que no tiene ninguna otra función salvo la propia. Basta con adjuntarle un elemento llamado redirector que se encargará de transferir al servidor de aplicaciones las peticiones HTTP que conciernen a recursos dinámicos. Los recursos estáticos se gestionarán por el propio servidor Web y el enlace entre el redirector y el servidor de aplicaciones lo proporcionará un protocolo de red propietario.



Esta solución también es posible cuando el servidor de aplicaciones está altamente solicitado. El redirector puede en este caso actuar como balanceador de carga repartiendo las peticiones HTTP entre varios servidores de aplicaciones. En este contexto, es necesario que los servidores de aplicaciones sean idénticos.



Clients ligeros y clients pesados

En el mundo de las aplicaciones informáticas, se suele llamar cliente ligero a una solución que permite usar una aplicación desde un ordenador cliente sin que se le tenga que instalar ningún programa. Generalmente, un navegador web realiza esta función. La mayoría de los sistemas operativos proporcionan este tipo de productos en su paquete de servicios básicos. El servidor de aplicaciones se encarga del conjunto de tratamientos en este caso. La ventaja indiscutible de este tipo de soluciones es la ausencia de intervención en los ordenadores cliente antes de poder usar la aplicación. Como contrapartida, el servidor de aplicaciones se queda con una fuerte carga de solicitudes y el aspecto visual de la aplicación queda limitado a las funcionalidades del lenguaje HTML. No obstante, existen multitud de tecnologías que se añaden a HTML para mejorar considerablemente la renderización gráfica de una aplicación.

Por contra, el cliente pesado no sufre ninguna limitación en este ámbito. Además, permite aligerar la carga del servidor ya que éste puede simplemente enviar al cliente los resultados en bruto de un tratamiento dejando al cliente la tarea de gestionar él mismo la presentación de estos resultados. Como contrapartida, esta solución requiere la instalación de la aplicación encargada del diálogo con el servidor y la presentación de los resultados en todos los ordenadores cliente. Cuando se crea una nueva versión de la aplicación, hay que volver a desplegar esta aplicación en todos los ordenadores cliente.

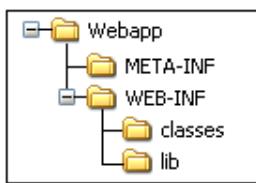
Lo ideal sería tener las ventajas de ambas soluciones. Esto es lo que permite el uso de applets. Un applet es una aplicación Java casi normal en el sentido que no hay prácticamente limitaciones en todo lo que puede hacer un applet. La única diferencia con una aplicación normal es que el sistema no se encarga directamente de su ejecución, sino que se ejecuta internamente en el navegador en el contexto de una página HTML. La ventaja respecto a una aplicación es que no requiere la realización de despliegues en los ordenadores cliente. Es el servidor el que transfiere al navegador el código del applet que éste debe ejecutar. Si se modificara el código fuente del applet, simplemente bastaría con poner a disposición de los clientes la nueva versión en el servidor para que se realice su transferencia. Cuando el servidor trata una petición HTTP, proporciona en este caso al cliente los resultados del tratamiento y el código para que éste los muestre. La creación de los applets se detalla en la obra JAVA 6 - Los fundamentos del lenguaje Java, de esta colección.

Estructura de una aplicación JEE

La normativa JEE también describe cómo debe organizarse una aplicación para que pueda ser soportada por cualquier servidor de aplicaciones compatible. Una aplicación web generalmente se compone de los elementos siguientes:

- De recursos estáticos: páginas HTML, imágenes, sonidos, hojas de estilo...
- De recursos dinámicos: servlets, JSP, Java Bean.
- De librerías de clases utilizadas por los diferentes componentes dinámicos.
- De un descriptor de despliegue que permite definir los parámetros de funcionamiento de la aplicación en el servidor, los enlaces entre las URL y los recursos dinámicos de la aplicación, las páginas por defecto y de error de la aplicación, la seguridad de la aplicación, etc.

Los archivos que contienen estos elementos tienen que organizarse en una forma de árbol concreta para ser fácilmente accesibles por el servidor de aplicaciones. Este árbol básico se presenta en la figura siguiente:



El directorio applicationWeb de este ejemplo representa la raíz de la aplicación. El nombre de esta carpeta no viene impuesto. Sin embargo, hay que tener en cuenta que algunos servidores usan el nombre de este directorio como nombre por defecto de la aplicación. De todas formas, se puede dar un nombre distinto cuando se despliegue la aplicación en el servidor. Todos los elementos contenidos en esta carpeta son accesibles por los clientes. Generalmente, se ponen las páginas HTML, las páginas JSP, imágenes, archivos de sonido... Estos distintos recursos pueden organizarse en carpetas diferentes para evitar mezclarlos. Por supuesto, hay que tener en cuenta estas carpetas cuando se deseé utilizar alguno de estos elementos.

La carpeta META-INF contiene el archivo MANIFEST.MF generado por la herramienta de archivado jar. Contiene información descriptiva del archivo cuando la aplicación se despliega con esta forma (ver a continuación).

La carpeta WEB-INF contiene elementos únicamente accesibles por el servidor. Es en esta carpeta donde se encuentra el archivo web.xml que es el descriptor de despliegue de la aplicación. La estructura de este archivo se detalla en el anexo.

El subdirectorio classes del directorio WEB-INF contiene el código compilado de todas las clases necesarias para el funcionamiento de la aplicación. Si las clases están definidas en paquetes, esta carpeta debe contener un árbol con estructura idéntica a la de los paquetes de la aplicación. Los archivos de esta carpeta nunca se transfieren a los clientes. Solamente el servidor tiene acceso para instanciar las distintas clases utilizadas en la aplicación.

El subdirectorio lib del directorio WEB-INF contiene todas las librerías indispensables para el buen funcionamiento de la aplicación. Estas librerías generalmente se ubican en esta carpeta en forma de archivo Java (jar). Aquí se puede encontrar, por ejemplo, la librería que contiene un driver de acceso a una base de datos o librerías de etiquetas JSP personalizadas.

Empaquetado de una aplicación

Para facilitar el despliegue de una aplicación, se puede incluir todo el conjunto de archivos necesarios para el funcionamiento de la aplicación en un archivo empaquetado Java. El archivo correspondiente lleva la extensión war (web archive). Este archivo simplemente se copia en la estructura de directorios del servidor, el cual se encarga automáticamente del despliegue de la aplicación web que contiene. Se genera mediante el uso de la utilidad jar.

La manipulación de un archivo empaquetado Java (archivo jar o war) retoma los mismos principios que la manipulación de archivos en el mundo Unix con el comando tar. Las opciones del comando jar que permiten manipular un archivo empaquetado Java son, por cierto, extrañamente parecidas a las del comando tar de Unix. El formato utilizado internamente en los archivos empaquetados Java es ampliamente conocido ya que se trata del formato ZIP. Los archivos empaquetados Java pueden manipularse con herramientas dedicadas al manejo de archivos ZIP.

1. Creación de un empaquetado

La sintaxis básica de creación de un empaquetado Java es la siguiente:

```
jar cf nombreDelEmpaquetado listaArchivos
```

Por supuesto, el parámetro c se destina a indicar al comando jar que se desea crear un empaquetado. En cuanto al parámetro f, indica que el comando tiene que generar un archivo. El nombre del empaquetado se indica en el tercer parámetro de este comando. Por convenio, la extensión de este empaquetado para una aplicación web será .war. El último elemento representa el o los archivos que se incluirán en el empaquetado. Si se desea incluir varios archivos en el archivo, sus nombres tienen que estar separados por un espacio. El carácter comodín * también está permitido en la lista. Si un nombre de directorio está presente en la lista, todo su contenido se añade al empaquetado.

El empaquetado se genera en el directorio actual. También están disponibles las opciones siguientes:

- -v muestra el nombre de los archivos a medida que se van añadiendo al empaquetado.
- -0 desactiva la compresión del empaquetado.
- -c elimina el nombre del directorio en el empaquetado.

2. Visualización del contenido

El contenido de un empaquetado puede visualizarse con el comando siguiente:

```
jar tf estudio.war
```

El comando muestra en consola el contenido del empaquetado (a continuación se muestra un extracto).

```
META-INF/
META-INF/MANIFEST.MF
Visualizacion.html
Animador/
Animador/EnTrabajo.html
Animador/IdentificacionAnimador.html
Animador/MainAnimador.html
Animador/MenuAnimador.jsp
Animador/ZonaAnimador.html
default.html
Error/
Error/Error.jsp
prueba/
prueba/MiApplet.class
Formaciones/
Formaciones/GestFormacion.html
Formaciones/GestFormacion.jar
Formaciones/LasFormaciones.jsp
menuPrincipal.html
perdido.html
ruta.html
```

```
Estudio/
Estudio/Conexion1_estudio.dbxmi
ssm.js
```

Se puede obtener información adicional añadiendo la opción `v` al comando. La fecha de modificación y el tamaño del archivo se añaden al resultado del comando.

```
jar tvf estudio.war
```

```
 0 Tue Jan 19 20:11:52 CET 2010 META-INF/
 68 Tue Jan 19 20:11:52 CET 2010 META-INF/MANIFEST.MF
523 Thu Jun 10 16:28:32 CEST 2004 Visualizacion.html
 0 Fri Aug 08 11:28:08 CEST 2008 Animador/
705 Tue Jun 15 12:04:28 CEST 2004 Animador/EnTrabajo.html
1040 Mon Mar 03 13:17:20 CET 2008 Animador/IdentificacionAnimador.html
 650 Wed Jul 09 15:36:56 CEST 2008 Animador/MainAnimador.html
6023 Thu Feb 07 10:39:24 CET 2008 Animador/MenuAnimador.jsp
 410 Tue Jun 15 11:33:58 CEST 2004 Animador/ZonaAnimador.html
 673 Tue Jun 15 11:29:50 CEST 2004 default.html
 0 Fri Aug 08 11:28:08 CEST 2008 Error/
 770 Fri Aug 17 10:54:50 CEST 2007 Error/Error.jsp
 0 Wed Jul 23 15:14:48 CEST 2008 prueba/
1468 Tue Aug 28 17:27:16 CEST 2007 prueba/MiApplet.class
 0 Fri Aug 08 11:28:08 CEST 2008 Formaciones/
1156 Thu Feb 07 10:44:54 CET 2008 Formaciones/GestFormacion.html
6737 Tue Jun 15 13:23:04 CEST 2004 Formaciones/GestFormacion.jar
 886 Thu Jun 10 16:18:02 CEST 2004 Formaciones/LasFormaciones.jsp
1201 Wed Aug 29 12:28:08 CEST 2007 menuPrincipal.html
 550 Wed Aug 29 12:48:56 CEST 2007 perdido.html
 956 Wed Aug 29 12:41:02 CEST 2007 ruta.html
```

Las rutas de acceso a los archivos se muestran con el carácter `/` como separador y son relativas a la raíz del archivo. Por supuesto, el contenido del empaquetado no se modifica por la ejecución de este comando.

3. Extracción

Los archivos pueden extraerse del empaquetado con el comando siguiente:

```
jar xvf estudio.war
```

Los archivos albergados en el empaquetado se vuelven a crear en disco en el directorio actual. Si el archivo contiene una ruta, ésta se vuelve a crear en el directorio actual. Los posibles archivos y directorios existentes se sobreescreiben por los que están en el interior del archivo. La extracción de un archivo puede ser selectiva indicando con un parámetro adicional la lista de archivos que se desea extraer del empaquetado separando el nombre de éstos mediante espacios. El comando siguiente permite extraer del empaquetado únicamente el archivo web.xml ubicado dentro de la carpeta WEB-INF.

```
jar xvf estudio.war WEB-INF/web.xml
```

El contenido del empaquetado no se modifica por este comando.

4. Actualización

El contenido de un empaquetado puede actualizarse mediante la adición de archivos después de su creación. En este caso hay que usar el comando siguiente:

```
jar uf estudio.war imágenes/logo.gif
```

El último parámetro de este comando representa la lista de archivos que se actualizarán en el archivo. Si estos archivos no existían en el empaquetado, se añaden, sino se reemplazan por la nueva versión. Si el empaquetado contiene directorios, la ruta de acceso completa tiene que especificarse en la lista de archivos que se añadirán.

Presentación

El protocolo HTTP se creó a comienzos de los años 90 por investigadores del CERN. El objetivo de estos investigadores era proporcionar un protocolo simple y eficaz para la Web que en aquellos días se encontraba en sus inicios. El uso principal de este protocolo tenía que ser el intercambio de documentos de hipertexto. Desde su creación, este protocolo se ha utilizado para transferir otros elementos a parte de documentos de hipertexto.

1. Funcionamiento

El principio de funcionamiento del protocolo HTTP se basa en el tandem pregunta/respuesta. El cliente genera una pregunta con la forma de una petición HTTP. Esta petición contiene por lo menos los datos que permiten identificar el recurso solicitado por el cliente. Generalmente se le añade más información de distinta índole. La petición HTTP simplemente es un bloque de texto que transita del cliente al servidor. Este bloque de texto tiene que tener un formato concreto para que sea reconocido por el servidor. Se compone de dos partes separadas por una línea en blanco (que contiene simplemente un retorno de carro/salto de línea). La primera parte, llamada cabecera de la petición HTTP, es obligatoria. La segunda parte, llamada cuerpo de la petición HTTP, es opcional. Su presencia depende del tipo de petición HTTP. Incluso si no hay cuerpo en la petición, la línea en blanco de separación es obligatoria.

Cuando el servidor recibe la petición HTTP procedente del cliente, la analiza y realiza los tratamientos necesarios para construir la respuesta HTTP. Como sucede con las peticiones, ésta también se construye siempre con un bloque de texto separado en dos por una línea en blanco. La primera parte, que corresponde a la cabecera de la respuesta HTTP, es obligatoria. La segunda parte, que corresponde al cuerpo de la respuesta HTTP, es opcional y depende del tipo de la respuesta HTTP.

Una respuesta HTTP solamente puede existir si se ha enviado previamente una petición HTTP al servidor. El servidor nunca toma la iniciativa de enviar datos a un cliente si éste no los ha solicitado. De hecho, es ciertamente imposible que esto sucediera debido a que de todo cliente que no le ha enviado una petición ignora simple y llanamente su existencia.

Generalmente, el protocolo TCP se utiliza para el transporte de los dos bloques de texto que forman la petición y la respuesta HTTP. Por defecto la conexión mediante este protocolo TCP se establece para cada par petición/respuesta. Sin embargo, para mejorar el uso del ancho de banda de red, la versión 1.1 del protocolo HTTP propone una solución que permite el transporte de varios pares petición/respuesta con la misma conexión TCP.

Para visualizar claramente el aspecto de un par petición/respuesta HTTP, a continuación se muestra el resultado de una captura de los datos transmitidos por la red en la generación de una petición HTTP por un navegador y la respuesta correspondiente realizada por el servidor (el cuerpo de la respuesta HTTP ha sido intencionadamente cortado en este ejemplo).

La petición HTTP relativa a la página de inicio del sitio www.eni-ecole.fr:

```
GET / HTTP/1.1
Host: www.eni-ecole.fr
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0; fr;
rv:1.9.0.15) Gecko/2009101601 Firefox/3.0.15 (.NET CLR 3.5.30729)
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,en;q=0.8,fr-fr;q=0.6,en-us;q=0.4,zh;q=0.2
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

La respuesta HTTP realizada por el servidor (el cuerpo de la respuesta está truncado).

```
HTTP/1.1 200 OK
Date: Mon, 04 Jan 2010 11:20:50 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.3 with Suhosin-Patch
X-Powered-By: PHP/5.2.4-2ubuntu5.3
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

<?xml version="1.0" encoding="iso-8859-1"?><!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
```

```

<head>
 .<META HTTP-EQUIV="CACHE-CONTROL" CONTENT="NO-CACHE">
 .<META HTTP-EQUIV="PRAGMA" CONTENT="NO-CACHE">
 .<META HTTP-EQUIV="EXPIRES" CONTENT="0">
<meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1" />
<title>ENI Ecole Informatique - Une formation, un dipl.me, un emploi</title>
<link href="eniStyle.css" rel="stylesheet" type="text/css" />
<!--[if IE 6]>
    <link rel="stylesheet" href="eniStyle6.css" type="text/css"
/>
<![endif]-->
</head>
<body onLoad="ges_menu(document.getElementById('item7'));">
<div class="cadre">
<div class="haut ; bleuFd">
...
...
...
</div>
</html>

```

Para demostrar que el protocolo HTTP puede usarse para transportar cualquier tipo de contenido, a continuación se muestra un par petición/respuesta que permite obtener de un servidor una imagen en formato gif.

```

HTTP/1.1 200 OK
Date: Mon, 04 Jan 2010 11:20:50 GMT
Server: Apache/2.2.8 (Ubuntu) PHP/5.2.4-2ubuntu5.3 with Suhosin-
Patch
Last-Modified: Tue, 30 Sep 2008 09:45:01 GMT
ETag: "50220-1fe-45819d661c140"
Accept-Ranges: bytes
Content-Length: 510
Keep-Alive: timeout=15, max=98
Connection: Keep-Alive
Content-Type: image/gif

GIF89a...4.....
.....
.....
.....,4....@.pH4`...1.l:...tJ."...V..z...8....3n.i....|N.
<.....w.....5.....-.->...."....!....%....2.....7.<<.....=6.....3..$....'....#
.....??
.....
,...,4..0.....
.Hp.....*\..P....#B.@..E..2j..... C:.Ar...(.S.....
b...)....89....Sg.#.@"
.J4...H..X....P.J.J....X.j.j5...;
```

En el caso del ejemplo, el navegador extraerá el cuerpo de la respuesta HTTP y lo tratará como una imagen en formato gif.

2. Las URL

Las URL (*Uniform Resource Locator*) están íntimamente relacionadas al protocolo HTTP. De hecho, gracias a ellas se pueden localizar los recursos que se desea recuperar mediante peticiones HTTP. Se componen de una cadena de caracteres compuesta de cinco campos. El formato que se tiene que respetar es el siguiente:

protocolo://identificador del servidor:número de puerto/recurso?parámetros

La información que contiene cada campo se describe a continuación:

- **protocolo:** indica el protocolo utilizado para acceder al recurso. En el caso del ejemplo que nos interesa, usaremos por supuesto HTTP (hay otros protocolos disponibles como ftp o mailto).
- **identificador del servidor:** esta parte de la URL permite localizar el servidor en la red. Corresponde

generalmente al FQDN (*fully qualified domain name*) del servidor. Este elemento tiene que permitir identificar sin ambigüedades el servidor en la red. Esta parte de la URL deberá transformarse en la dirección IP para que el protocolo TCP pueda establecer una conexión con el servidor. Generalmente, un servidor DNS (*Domain Name System*) se encarga de esta operación. También es posible utilizar directamente una dirección IP en una URL. Sin embargo, es más fácil de recordar un FQDN que una dirección IP (y aún lo será más cuando se utilice la versión 6 del protocolo IP).

- **número de puerto:** este dato proporciona el número de puerto TCP con el que se debe establecer la conexión. El número de puerto le sirve al protocolo TCP para identificar una aplicación particular en una máquina. De hecho, gracias a este número de puerto una máquina puede alojar varias aplicaciones, cada una de ellas usando un número de puerto diferente. El puerto 80 se asigna por defecto al protocolo HTTP. Si el servidor utiliza este número de puerto por defecto éste puede omitirse en la petición HTTP.
- **recurso:** esta parte de la URL permite determinar el recurso que se desea obtener. Puede estar compuesto por varios elementos separados por el carácter /.
- **parámetros:** cuando la petición HTTP se realiza sobre un recurso dinámico (código que se desea que ejecute el servidor) a veces es preferible poder pasar parámetros a este código. Esto es lo que permite hacer esta parte de la URL que proporciona estos datos con la forma del par nombreDeParámetro=valorDelParámetro. La aplicación tiene que estar preparada para tratar estos parámetros. Si se va a enviar más de un parámetro, hay que separar cada par correspondiente en la URL con el carácter &.

En una URL ciertos caracteres tienen un significado específico. Es el caso, por ejemplo, de los caracteres /, ?, &. Si estos caracteres tienen que usarse para otro uso que para el que han sido diseñados en una URL (en un valor de un parámetro, por ejemplo), hay que realizar una codificación de estos caracteres para asegurar la coherencia de la URL. La codificación consiste en reemplazar en la URL el carácter en cuestión por la secuencia %HH, donde HH corresponde al código ASCII del carácter expresado en hexadecimal. La tabla mostrada a continuación contiene la correspondencia de los caracteres usados más frecuentemente y su codificación asociada.

carácter	codificación
Espacio	%20 o a veces +
"	%22
%	%25
&	%26
+	%2B
.	%2E
/	%2F
?	%3F
,	%60

Esta operación de codificación se realiza automáticamente por los navegadores cuando estos construyen una URL. Por contra, si una URL tiene que construirse mediante código, será necesario realizar manualmente la codificación de los caracteres especiales. La clase URLEncoder dispone del método estático encode al que hay que proporcionar como primer parámetro la cadena de caracteres en la que se reemplazarán los caracteres especiales por su codificación. El segundo parámetro indica el juego de caracteres que se utilizará.

El código siguiente:

```
out.println(URLEncoder.encode("java&html", "UTF-8"));
```

muestra la cadena codificada:

```
java%26html
```

► Ciertos navegadores o servidores relativamente antiguos imponen un límite de 255 caracteres para la longitud de la cadena de caracteres que representa la URL.

Las peticiones HTTP

Las peticiones HTTP constituyen el punto de partida del diálogo entre un navegador y un servidor Web. La creación y el envío de la petición HTTP son, en la mayoría de los casos, transparentes para el usuario debido a que el navegador se encarga automáticamente de estas dos acciones. En general, construye la petición HTTP en función de los datos contenidos en una página HTML. Estos datos tienen su origen en un enlace de hipertexto o en un formulario. La estructura de la petición generada es siempre la misma. La parte de la cabecera empieza siempre por una línea con el mismo formato. Esta línea contiene el tipo de la petición HTTP seguido del identificador del recurso solicitado y finaliza con la versión del protocolo HTTP utilizado (cada vez más, HTTP/1.1).

1. Los distintos tipos de petición

Originalmente, la versión 1.0 del protocolo HTTP disponía de tres tipos de petición HTTP:

- La petición GET permite obtener un recurso disponible en el servidor. Este es el tipo de petición que genera un navegador cuando se introduce directamente una URL o cuando se activa un enlace de hipertexto. Este tipo de petición no contiene ningún tipo de dato en el cuerpo de la petición. Si tienen que enviarse parámetros al servidor, éstos se añaden a continuación de la URL.
- La petición POST se destina más bien al envío al servidor de los datos recopilados en un formulario HTML. Aunque también se pueda realizar con una petición GET, el uso de una petición POST presenta varias ventajas:
 - No hay ningún límite teórico a la cantidad de información que se envía hacia el navegador porque esta información se inserta en el cuerpo de la petición HTTP. Con el método GET, la información que se envía se añade mediante parámetros en la URL, con el riesgo de sobrepasar el límite de tamaño máximo de ésta.
 - Los datos que se transfieren en el cuerpo de la petición HTTP no se pueden ver en la barra de dirección del navegador, como sucede en el caso de una petición HTTP GET.
- La petición HEAD tiene un funcionamiento similar al de la petición GET excepto en que el servidor no devuelve nada en el cuerpo de la respuesta HTTP. Los navegadores utilizan este tipo de peticiones para la gestión del almacenamiento en caché de datos.

La versión 1.1 del protocolo HTTP añade nuevos tipos de petición.

- La petición PUT funciona de manera opuesta a la de la petición POST, ya que permite enviar un recurso al servidor para que éste lo guarde de forma permanente.
- La petición DELETE permite solicitar al servidor la eliminación de un recurso determinado.
- La petición TRACE se usa sobre todo en la fase de pruebas ya que permite solicitar al servidor la devolución en el cuerpo de la respuesta HTTP de una copia de la petición HTTP que acaba de recibir.
- La petición OPTIONS permite obtener datos sobre las opciones que se pueden utilizar para obtener un recurso.

Entre todas estas peticiones, algunas son potencialmente peligrosas para el servidor (PUT, DELETE) y solamente se debe permitir su acceso después de una etapa de autenticación del cliente.

2. Las cabeceras de petición

El navegador añade las cabeceras de petición en el momento en que construye la petición HTTP. Cada tipo de navegador tiene su propia técnica para construir una petición HTTP y las cabeceras utilizadas no son forzosamente siempre las mismas entre un navegador y otro. Sin embargo, hay un conjunto de cabeceras que se pueden calificar como estándar y que están prácticamente siempre presentes en una petición HTTP. Las cabeceras se usan principalmente para proporcionar al servidor información adicional acerca de la petición. A continuación se facilita un listado de las cabeceras que los navegadores usan con más frecuencia:

- **Accept:** esta cabecera permite indicar al servidor qué tipos de datos se aceptan como respuesta. Los datos tienen que facilitarse formateados en una lista de tipo MIME (*Multipurpose Internet Mail Extensions*). Si el navegador no tiene ninguna preferencia, puede añadir la cabecera siguiente:

```
Accept: */*
```

- **Accept-Charset:** el navegador utiliza esta cabecera para indicar sus preferencias relativas al juego de caracteres que puede usar el servidor para construir la respuesta HTTP.

```
Accept-Charset: ISO-8859-1,utf-8
```

Los servidores pueden usar esta cabecera de diferentes formas. Algunos devolverán un error al cliente si no tienen a su disposición el recurso con uno de los juegos de caracteres especificados, otros servidores podrán devolver al cliente el recurso con el juego de caracteres por defecto disponible en el servidor.

- **Accept-Encoding:** el navegador indica con esta cabecera los métodos de compresión aceptados.

```
Accept-Encoding: gzip,deflate
```

- **Accept-Language:** esta cabecera indica al servidor en qué lengua se desea preferentemente obtener el recurso solicitado.

```
Accept-Language: es,en
```

- **Connection:** esta cabecera es específica de la versión 1.1 del protocolo HTTP. Determina cómo se comportarán las conexiones TCP utilizadas para transportar la petición y la respuesta. Si el navegador desea mantener abierta la conexión TCP después del envío de esta petición, tiene que especificar el valor *keep-alive* para esta cabecera. Cuando desee el cierre de la conexión después del tratamiento de una petición, usará el valor *close*.

```
Connection: keep-alive
```

- **Host:** esta cabecera especifica el FQDN (*fully qualified domain name*) y el número de puerto del servidor que alberga el recurso solicitado. Este nombre puede corresponder a un servidor físico o a uno virtual. En todo caso, es necesario que pueda transformarse en una dirección IP para que el protocolo TCP pueda transportar la petición HTTP. En el caso que se estén usando servidores virtuales, hay varios FQDN con la misma dirección IP. La referencia hacia el servidor virtual adecuado se realiza gracias a esta cabecera. Esta cabecera es específica de la versión 1.1 del protocolo HTTP.

```
Host: www.eni-ecole.fr
```

- **Referer:** esta cabecera contiene la URL del documento a partir del cual la petición HTTP ha sido generada. Se informa cuando una petición HTTP se construye después de la activación de un enlace de hipertexto o de la validación de un formulario. El ejemplo siguiente es un extracto de una petición HTTP generada por el uso de un enlace hipertextual.

```
GET /index.php HTTP/1.1
Referer: http://www.eni-ecole.fr/
```

- **User-Agent:** esta cabecera contiene información que permite identificar el tipo de navegador en el origen de la petición. Puede utilizarse para realizar estadísticas o para permitir al servidor optimizar el código enviado en función del tipo de navegador. Puede considerarse como la firma del navegador.

```
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0;
SLCC1; .NET CLR 2.0.50727; InfoPath.2; .NET CLR 3.5.21022; .NET
CLR 1.1.4322; .NET CLR 3.5.30729; .NET CLR 3.0.30618)
```

Otras cabeceras relacionadas con la gestión del almacenamiento en caché se estudiarán un poco más adelante en este capítulo.

Las respuestas HTTP

Después de la recepción y el tratamiento de una petición HTTP, el servidor genera una respuesta HTTP para transmitir el recurso solicitado al cliente. La estructura de la respuesta HTTP es similar a la de la petición HTTP. También se descompone en dos partes, la cabecera de la respuesta y el cuerpo de la respuesta. Estas dos partes están separadas por una línea en blanco. Puede haber respuestas HTTP sin cuerpo.

La primera línea de la parte de la cabecera de la respuesta ejerce una función específica debido a que esta línea también se llama línea de estado de la respuesta. Permite, principalmente, definir el tipo de la respuesta. Contiene un valor numérico correspondiente al código de estado de la respuesta asociada a un mensaje.

1. Los diferentes tipos de respuesta

Las respuestas HTTP se organizan en cinco categorías. La primera cifra del código de estado determina la categoría.

1XX: información

2XX: éxito

3XX: redirección

4XX: error provocado por el cliente

5XX: error provocado por el servidor

En cada categoría hay varios tipos de respuesta. Los más vistos son los listados a continuación:

100 continúa: el servidor genera esta respuesta cuando recibe una petición HTTP en varias partes. El cliente necesita esta respuesta para saber que el comienzo de la petición se ha recibido satisfactoriamente y que el servidor está a la espera de la continuación.

200 OK: ésta es la respuesta más frecuente generada por los servidores, por suerte, ya que indica que la petición se ha tratado correctamente.

201 creado: el servidor genera esta respuesta típicamente cuando recibe una petición HTTP put. Indica de este modo que el recurso se ha creado correctamente en el servidor.

204 sin contenido: este tipo de respuesta la envía el servidor cuando solamente los datos contenidos en la cabecera de la respuesta son los que el cliente espera. No hay, en este caso, cuerpo de respuesta HTTP.

301 movimiento permanente: el recurso solicitado por el cliente se encuentra ahora en otra ubicación. Esta nueva ubicación queda indicada por la cabecera location.

302 movimiento temporal: el recurso solicitado por el cliente se encuentra temporalmente en otra ubicación. El servidor tiene que devolver en la respuesta un enlace de hipertexto hacia la nueva ubicación.

304 sin modificar: el servidor genera esta respuesta en relación a una petición condicional del cliente. Indica que el recurso presente en el servidor es idéntico al que ya está disponible en el cliente.

400 petición incorrecta: la sintaxis de la petición HTTP es incorrecta.

401 no autorizado: el acceso al recurso solicitado requiere la autenticación del cliente.

403 acceso prohibido: el acceso a este recurso está prohibido.

404 no encontrado: el recurso solicitado no se ha encontrado.

405 método no admitido: el tipo de petición HTTP utilizado para acceder a este recurso no está admitido.

407 autenticación proxy: un servidor proxy envía este tipo de respuesta cuando requiere la autenticación del cliente antes de transmitir la petición a un servidor.

500 error interno del servidor: un problema de funcionamiento impide al servidor cursar con éxito la petición.

503 servicio no disponible: una sobrecarga del servidor impide tratar la petición.

505 versión HTTP no compatible: el servidor no puede utilizar la versión del protocolo HTTP especificada en la petición.

2. Las cabeceras de respuesta

El servidor genera las cabeceras de respuesta en el momento en que construye la respuesta HTTP. Cada tipo de servidor tiene su propia técnica para construir una respuesta HTTP y las cabeceras no son forzosamente las mismas entre un servidor y otro. Sin embargo, hay un conjunto de cabeceras a las que se puede calificar como estándar y

que están prácticamente siempre presentes en una respuesta HTTP. Las cabeceras se usan principalmente para proporcionar al navegador información adicional acerca de la respuesta. A continuación se muestra una lista de las cabeceras más frecuentemente usadas por los servidores.

location: esta cabecera de respuesta se encuentra en respuestas HTTP de redirección (3XX) para indicar la nueva ubicación donde se encuentra el recurso solicitado.

server: esta cabecera de respuesta contiene información acerca del tipo de servidor que ha generado la respuesta.

via: esta cabecera de respuesta se añade a la respuesta de origen realizada por el servidor cuando la respuesta ha pasado por un servidor proxy.

retry-after: esta cabecera de respuesta se encuentra principalmente en las respuestas HTTP de tipo 503 (servicio no disponible) para indicar una duración estimada de la no disponibilidad del servidor. Esta duración se expresa en segundos de espera o con una fecha y hora a partir de la cual el cliente podrá reformular la petición.

proxy-authenticate: este tipo de cabecera es incluida por un servidor proxy en una respuesta HTTP 407. Indica el método de autenticación esperado por el servidor proxy. El navegador utiliza esta información para reformular la petición HTTP que contiene los datos de autenticación del usuario. Entonces éste añade en la cabecera de la nueva petición que ha generado la cabecera de petición proxy-authorization con los datos identificativos del usuario.

allow: este tipo de cabecera se encuentra en las respuestas HTTP 405 (método no admitido) para indicar cuáles son los métodos HTTP aceptados para acceder a un recurso determinado.

connection: esta cabecera es propia de la versión 1.1 del protocolo HTTP. Indica cómo se tiene que gestionar la conexión TCP una vez el cliente ha recibido la respuesta.

content-encoding: esta cabecera indica el método de compresión utilizado por los datos contenidos en el cuerpo de la respuesta HTTP.

content-language: esta cabecera indica el idioma del documento contenido en el cuerpo de la respuesta HTTP.

content-length: esta cabecera indica el número de bytes contenidos en el cuerpo de la respuesta HTTP.

content-MD5: esta cabecera permite la verificación de la integridad del contenido de la respuesta. La firma MD5 calculada por el servidor sobre el cuerpo de la respuesta se añade a continuación a la respuesta con esta cabecera. El cliente realiza la misma operación cuando recibe la respuesta y compara a continuación el resultado obtenido con el valor de esta cabecera para detectar una posible alteración de la respuesta durante su transferencia.

content-type: esta cabecera indica el tipo MIME (*Multipurpose Internet Mail Extensions*) del documento contenido en la respuesta. Ciertos navegadores necesitan esta cabecera para poder interpretar correctamente la respuesta. Si esta cabecera no está en la respuesta se trata el documento como si fuera texto en bruto.

date: esta cabecera contiene la fecha y la hora de generación de la respuesta HTTP.

last-modified: esta cabecera contiene la fecha y la hora de la última modificación del recurso en el servidor.

WWW-authenticate: este tipo de cabecera se incluye en las respuestas HTTP 401 (no autorizado) para reclamar al cliente los datos de su identidad. Generalmente los navegadores muestran automáticamente un cuadro de diálogo que permite al usuario introducir un nombre y una contraseña. La validación de este cuadro de diálogo provoca nuevamente el envío de la petición HTTP hacia el servidor con los datos recopilados.

Algunas cabeceras de respuesta que están relacionadas con la gestión del almacenamiento en caché se describen en la sección siguiente.

Gestión del almacenamiento en caché

Generalmente en una aplicación, el contenido de algunos recursos cambia con muy poca frecuencia. Para optimizar las transferencias entre el cliente y el servidor, el navegador puede almacenar temporalmente estos recursos. Cuando tiene que realizar de nuevo una petición HTTP para obtener uno de estos recursos, puede usar la versión que ha almacenado localmente en vez de contactar de nuevo con el servidor. Sin embargo, tiene que asegurarse previamente que la versión del recurso que mantiene localmente sigue siendo válida. Para ello, puede usar varias técnicas como una petición condicional o realizar previamente una petición HEAD para obtener únicamente la parte de la cabecera de la respuesta HTTP por parte del servidor.

1. Gestión realizada por el cliente

Hay varias cabeceras de petición HTTP para darle un carácter condicional a una petición HTTP. Este carácter condicional lleva, generalmente, una fecha o un identificador asociado al recurso.

La primera solución para el navegador consiste en añadir la cabecera `if-modified-since` seguida de la fecha y hora de creación del recurso que ya tiene en caché. De este modo, solicita al servidor que le transfiera el recurso si éste ha sido modificado posteriormente a esta fecha. Si éste es el caso, el servidor devuelve al cliente el recurso en una respuesta HTTP 200. Si el recurso no se ha modificado en el servidor desde esa fecha, éste genera una respuesta HTTP 304 y el navegador utiliza la versión del recurso que ya tenía en la memoria caché.

La segunda solución es algo similar, pero además usa una etiqueta asociada por el servidor al recurso. Éste modifica el valor de la etiqueta en cada modificación del recurso.

Para generar este tipo de petición, el navegador añade la cabecera `if-none-match` seguida de la etiqueta correspondiente a la versión del recurso que tiene en caché. Como sucede con la solución anterior, el servidor devuelve al cliente el recurso si no tiene la misma versión. Si no, el servidor genera una respuesta HTTP 304.

2. Gestión realizada por el servidor

Para que los dos mecanismos mostrados en el apartado anterior funcionen, el servidor tiene que añadir la información necesaria en las respuestas que realiza al cliente. Esta información se añade mediante las cabeceras de respuesta siguientes:

`date`: esta cabecera contiene la fecha y hora de generación de la respuesta HTTP. El navegador la utiliza como referencia para realizar peticiones HTTP condicionales.

`Etag`: esta cabecera representa el valor de la etiqueta asociada por el servidor al recurso cuando éste se transmite al cliente.

`expires`: con esta cabecera el servidor indica al navegador hasta cuando puede conservar en caché el recurso enviado. Una vez que la fecha haya pasado, el navegador tiene que realizar una nueva petición HTTP si necesita usar este recurso de nuevo.

El servidor también puede impedir el almacenamiento en caché de un recurso por el navegador añadiendo la cabecera de respuesta `cache-control` con el valor `no-cache`.

Ese capítulo no pretende reemplazar la norma RFC2616 que describe detalladamente en varias centenas de páginas el funcionamiento del protocolo HTTP. Simplemente permite comprender mejor los elementos que vamos a manipular en los capítulos siguientes.

Presentación

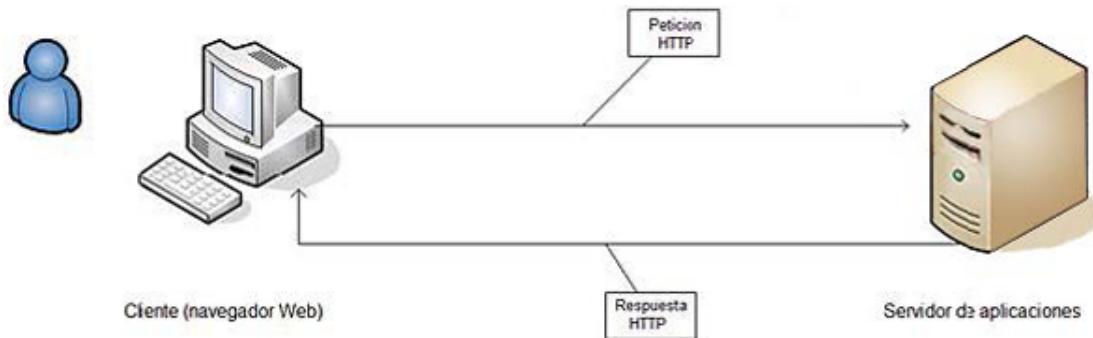
La primera pregunta que uno se plantea cuando comienza a desarrollar aplicaciones Web concierne en general al aspecto que puede tener un servlet. De hecho, un servlet es una simple clase Java que permite añadir funcionalidades a un servidor de aplicaciones. Para que el servidor pueda encargarse de esta nueva clase, ésta tiene que respetar algunas restricciones.

Cuando un cliente desea ejecutar el código añadido en el servidor de aplicaciones, tiene que realizar la solicitud al servidor mediante el protocolo HTTP. Esta solicitud generalmente toma la forma de una petición HTTP enviada por el cliente con destino la URL asociada por el servidor al servlet. Éste ejecuta el tratamiento y genera una respuesta HTTP para transmitirle al cliente el resultado de su ejecución.

1. Diálogo con un servlet

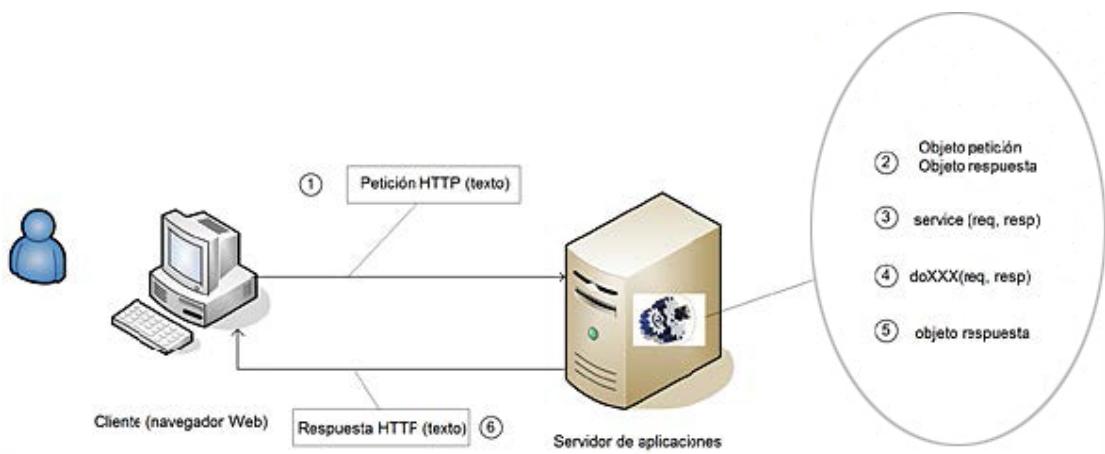
Los fundamentos del diálogo entre un cliente y un servlet se sustentan por lo tanto en el protocolo HTTP. Este protocolo se basa en el uso de un par petición-respuesta. La petición se usa para transportar información del cliente al servidor y la respuesta, obviamente, se usa para transportar información del servidor al cliente. La información añadida en la respuesta HTTP está relacionada generalmente con los resultados de la ejecución del código del servlet. En general, la creación y el envío de la petición HTTP se confían a un navegador web. Éste recopila los datos introducidos por el usuario mediante un documento HTML y realiza las transformaciones de formato de estos datos generando una petición HTTP. Esta petición se envía por la red con destino el servidor de aplicaciones. El servidor recibe y analiza esta petición y ejecuta el servlet al que concierne la petición HTTP. Entonces, genera la respuesta HTTP y siempre la envía al cliente por la red. El cliente (navegador) recibe esta respuesta y analiza su contenido para determinar cómo debe interpretarla (página HTML, imagen, sonido...).

Este funcionamiento se resume en el esquema siguiente.



2. Tratamientos realizados

Cuando el servidor de aplicaciones recibe una petición HTTP, éste transforma la petición HTTP, que hasta el momento era una cadena de caracteres, en un objeto Java. Los datos contenidos en la petición HTTP se transfieren a las propiedades de este objeto Java. También se crea un objeto Java que representa la respuesta HTTP. El servlet se sirve de éste para construir la respuesta que se enviará al cliente. El servidor extrae seguidamente de la petición el nombre del servlet que tiene que ejecutar. El control se transfiere al servlet con la llamada a su método `service`. Los objetos que representan la petición y la respuesta HTTP se pasan como parámetro a este método. El trabajo principal de éste consiste en determinar el tipo de petición HTTP (GET, POST...). La última etapa del servlet consiste en ejecutar el método apropiado para el tratamiento de este tipo de petición (`doGet`, `doPost`...). Estos métodos también reciben los dos objetos creados para representar la petición y la respuesta HTTP. El contenido de cada uno de estos métodos no viene fijado, sino que será redefinido por el creador del servlet. Nuestro principal trabajo con los servlets es por lo tanto crear el contenido de estos métodos. En la gran mayoría de los casos, estos métodos extraerán de la petición HTTP los parámetros recibidos del cliente, realizarán un tratamiento y finalmente construirán la respuesta. Cuando el tratamiento del servlet ha finalizado, el servidor de nuevo colabora para ejecutar esta vez la operación inversa de la que acaba de realizar. El objeto que representa la respuesta se transforma en respuesta HTTP con el aspecto de texto y ésta se devuelve al cliente. El tratamiento de la petición por el servlet ha finalizado.



3. Clases e interfaces utilizadas

Para realizar el tratamiento de una petición HTTP, el servidor utiliza muchas clases y interfaces. Cuando el servidor transforma la petición HTTP tal cual (con el formato de una cadena de caracteres) en un objeto Java utiliza una clase que implementa la interfaz `javax.servlet.http.HttpServletRequest`. Esta interfaz hereda a su vez de la interfaz `javax.servlet.ServletRequest`. El objeto obtenido permite de este modo acceder fácilmente a la información principal transportada por la petición HTTP. Su utilización se detalla en la sección Utilizar la petición HTTP de este capítulo.

Otro objeto que implementa la interfaz `javax.servlet.http.HttpServletResponse`, que a su vez hereda la interfaz `javax.servlet.ServletResponse`, también se crea para la construcción de la respuesta HTTP por el servlet. El uso de este objeto se detalla en la sección Construir la respuesta HTTP de este capítulo.

Para que el servidor pueda encargarse de la ejecución del servlet, éste tiene que respetar ciertas características. Estas características se definen en la interfaz `javax.servlet.Servlet` que se implementa en la clase abstracta `javax.servlet.GenericServlet`. Esta clase es especializada por la clase `javax.servlet.http.HttpServlet`, que define las características de un servlet que puede ser contactado gracias al protocolo HTTP. Prácticamente siempre es ésta la clase que se usa como clase base para los servlets. La sección siguiente detalla los métodos principales disponibles que pueden redefinirse en esta clase.

Ciclo de vida de un servlet

Como cualquier clase Java, un servlet tiene que instanciarse para poder usarse. Generalmente, cuando se necesita una instancia de una clase, se tiene que utilizar el operador `new` para crearla. El problema con un servlet es que no se sabe exactamente en qué momento se le necesitará. Son los clientes los que deciden cuando una instancia de un servlet es necesaria generando una petición HTTP para solicitar su ejecución. El servidor de aplicaciones es el mejor ubicado para detectar esta solicitud y es por lo tanto el responsable de la creación y la destrucción de las instancias de servlets. Para ello, utiliza la estrategia siguiente:

En el momento de la recepción de una petición HTTP, determina si ésta concierne a un servlet. Si éste es el caso, verifica si ya hay una instancia de este servlet en memoria y llama entonces al método `service` de esta instancia de servlet. Si no hay ninguna instancia disponible de este servlet, el servidor crea una y llama al método `init` de esta nueva instancia. Una vez hecho este paso adicional, puede llamar al método `service` del servlet.

Con esta técnica, una misma instancia de servlet tratará las peticiones HTTP de muchos clientes. Cuando el servidor estima que no se necesita más a este servlet, destruye automáticamente la instancia correspondiente. Esta situación se produce en general únicamente en el momento de parada del servidor o cuando se publica una nueva versión del servlet. Antes de la eliminación de la instancia de un servlet se ejecuta su método `destroy`.

1. Método init

El servidor invoca el método `init` inmediatamente después de la instanciación del servlet. Por defecto, la implementación de este método no realiza ninguna operación. De hecho existe para permitir la inicialización de recursos que requiere el servlet para ejecutar su tratamiento. Puede por ejemplo utilizarse para establecer una conexión con un servidor de base de datos o para abrir un archivo en el que el servlet registrará información (registro de información). Si la ejecución del método `init` no tiene éxito se lanza una excepción del tipo `ServletException`. Esta excepción permite al servidor detectar la no disponibilidad del servlet.

2. Parámetros de inicialización

Se puede evitar poner en el código algunos datos que el servlet va a necesitar para su inicialización. Por ejemplo, en el caso en que éste requiera establecer una conexión con un servidor de base de datos, los datos de conexión al servidor puede que no se conozcan hasta el momento en que se despliegue la aplicación en el servidor de producción. Si estos datos están escritos directamente en el código, obligatoriamente habrá que modificarlos y se tendrá que recompilar la aplicación. Para evitar esta manipulación, este tipo de datos se puede colocar en el descriptor de despliegue de la aplicación (`web.xml`). El método `init` podrá recuperar los datos de este archivo. Los datos de inicialización de cada servlet se añadirán en el interior de la etiqueta `<servlet>` mediante el uso de la etiqueta `<init-param>`. Esta etiqueta tiene dos elementos obligatorios:

- `<param-name>`: representa el nombre del parámetro. Este nombre tiene que usarse en el método `init` del servlet para acceder al parámetro.
- `<param-value>`: representa el valor asignado al parámetro.

El elemento `<description>`, aunque es opcional, resulta muy útil para la claridad del descriptor de despliegue.

La declaración de un servlet en el archivo `web.xml` tiene por lo tanto el aspecto siguiente:

```
<servlet>
    <servlet-name>PrimerServlet</servlet-name>
    <servlet-class>es.eni.ri.PrimerServlet</servlet-class>
    <init-param>
        <description>
            dirección del servidor de base de datos
        </description>
        <param-name>direccionIpBDD</param-name>
        <param-value>127.0.0.1</param-value>
    </init-param>
</servlet>
```

La recuperación de los parámetros en el método `init` se realiza gracias al método `getInitParameter`. Este método espera como parámetro una cadena de caracteres que representan el nombre del parámetro en el descriptor de despliegue. Esta función devuelve el valor del parámetro tipado siempre como una cadena de caracteres. Si el parámetro indicado no se encuentra en el descriptor de despliegue, se devuelve el valor `null`.

El método `init` del servlet tendría el aspecto siguiente:

```
@Override  
public void init() throws ServletException  
{  
    super.init();  
    String ipServidor;  
    ipServidor=getInitParameter("direccionIpBDD");  
    if (ipServidor==null)  
    {  
        throw new ServletException();  
    }  
    cnx=abrirConexionBDD(ipServidor);  
    if (cnx==null)  
    {  
        throw new ServletException();  
    }  
}
```

3. Método `destroy`

Este método se invoca cuando el servidor decide destruir una instancia del servlet. Esta situación se produce generalmente cuando se para el servidor. Este método puede ser útil para liberar los recursos usados por el servlet durante su funcionamiento. Por ejemplo, una conexión a un servidor de base de datos abierta por el método `init` del servlet puede cerrarse en este método.

Sin embargo, hay que ser muy precavido con este método ya que se ejecuta solamente en una parada normal del servidor, un apagado repentino del servidor no permitiría la ejecución de este método. Si hay información importante para el funcionamiento de la aplicación que debería guardarse en un archivo o en una base de datos, es mejor hacerlo periódicamente durante el funcionamiento del servlet mejor que en el método `destroy`.

4. Método `service`

Cuando el servidor recibe una petición HTTP, realiza un análisis completo de los datos contenidos en la petición. Este análisis permite en primer término determinar el host y la aplicación implicada por la petición HTTP. El servidor determina a continuación qué servlet tiene que ejecutarse. Entonces, invoca el método `service` de este servlet. Éste va a realizar una última selección analizando la petición HTTP para determinar su tipo (GET, POST, PUT...). El método `doGet`, `doPost`, `doPut`... del servlet se llama en función del tipo de petición HTTP recibido.

5. Métodos `doXXXX`

Estos métodos constituyen realmente el corazón de un servlet. Sus implementaciones por defecto no realizan ninguna operación. La redefinición de algunos de estos métodos es por lo tanto prácticamente obligatoria para que un servlet tenga interés. Sin embargo, no es obligatorio redifinir todos los métodos `doXXX` de un servlet. Generalmente sólo los métodos `doGet` y `doPost` se redefinen. Corresponden a una invocación de un servlet por una petición HTTP GET o POST. Las operaciones realizadas en los métodos `doGet` y `doPost` pueden ser distintas. Sin embargo, se aconseja conservar una cierta coherencia entre ambos métodos realizando los mismos tratamientos en el método `doGet` y en el método `doPost`. La técnica utilizada para solicitar la ejecución de un servlet no debería tener impacto alguno en su ejecución.

La primera idea para conservar la coherencia entre ambos métodos consiste en copiar el mismo código en el método `doPost` y el método `doGet`. El peligro de esta solución radica en que las modificaciones que sin duda se realizarán en uno de los métodos puede que no se transmitan al otro. Es mejor tener solamente un único ejemplar de código. Éste puede estar ubicado indiferentemente tanto en el método `doGet` como en el método `doPost`. El método que no tenga el código basta que tenga la llamada al otro pasándole la petición HTTP y la respuesta HTTP.

Con esta solución, las dos versiones siguientes de servlet son equivalentes.

Versión 1:

```
public class TestServlet extends HttpServlet {  
  
    @Override
```

```

protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    //código del servlet
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    doGet(request, response);
}
}

```

Versión 2:

```

public class TestServlet extends HttpServlet {

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    doPost(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    //código del servlet
}
}

```

Algunas herramientas de desarrollo, como por ejemplo NetBeans, utilizan una tercera solución colocando el código en un método "normal" que es llamado desde el método `doGet` y desde el método `doPost`.

Versión NetBeans:

```

public class TestServlet extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    try
    {
        /*
         * código del servlet
        */
    }
    finally
    {
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)

```

```
throws ServletException, IOException
{
    processRequest(request, response);
}
```

Con esta solución la independencia del tipo de petición HTTP usada para solicitar la ejecución del servlet es un poco más clara que con las dos soluciones anteriores.

Utilizar la petición HTTP

La petición HTTP es el único medio disponible para hacer transitar datos desde el cliente hasta el servidor que aloja un servlet. Esta petición generalmente se crea automáticamente por el navegador en el momento en que se usa un enlace de hipertexto o en el momento en que se valida un formulario. El navegador recopila todos los datos necesarios y genera la petición HTTP con la estructura de una cadena de caracteres (ver capítulo El protocolo HTTP para el formato de esta cadena de caracteres). La petición HTTP se envía a continuación al servidor para tratarla. En la recepción de esta cadena de caracteres, el servidor la analiza para determinar qué recurso se está solicitando. Además, transforma esta cadena de caracteres en un objeto `HttpServletRequest` en el que sus propiedades se inicializan con los datos contenidos en la petición HTTP. A continuación, el objeto `HttpServletRequest` se pasa como parámetro al método `doXXXX` llamado en función del tipo de petición HTTP.

1. Obtener información sobre la URL

Un gran número de métodos permiten obtener información sobre la URL usada por el cliente para contactar con el servidor y solicitar la ejecución del servlet. Estos métodos permiten extraer los diferentes elementos que forman una URL (ver descripción en el capítulo que trata el protocolo HTTP).

`getServerName()`: este método extrae de la URL el elemento que identifica el servidor hacia el que se ha enviado la petición HTTP. Generalmente este elemento corresponde al FQDN (*Fully Qualified Domain Name*) del servidor. Éste se transforma en dirección IP gracias al mecanismo de resolución de nombres, generalmente un servidor DNS (*Domain Name System*). Esta dirección IP se usa a continuación para hacer transitar la petición hasta el servidor para tratarla.

`getServerPort()`: este método permite obtener el número de puerto TCP (*Transmission Control Protocol*) hacia el que la petición HTTP se ha enviado. El puerto 80 está asociado por defecto al protocolo HTTP. Para el desarrollo de una aplicación, algunos servidores usan por defecto el puerto 8080.

`getContextPath()`: este método recupera el nombre de la aplicación que aloja el servlet que debe ser contactado. Este nombre se asocia a la aplicación cuando ésta se despliega en el servidor. Los entornos de desarrollo utilizan en general el nombre asociado al proyecto para identificar la aplicación en el servidor. Este método es muy práctico para generar automáticamente enlaces de hipertexto que se adaptan al nombre usado para identificar la aplicación en el servidor. La siguiente porción de código genera un enlace de hipertexto correcto sea cual sea el nombre utilizado para el despliegue de la aplicación.

```
out.println("<a href=" + request.getContextPath() +  
"/pag2.html>continuar</a>");
```

`getServletPath()`: este método proporciona el último eslabón de la cadena utilizada para contactar al servlet desde el cliente. El valor devuelto por este método identifica el servlet dentro de una aplicación. Los nombres asociados a los servlets de una aplicación son definidos por las etiquetas `<servlet-mapping>` del descriptor de despliegue de la aplicación (`web.xml`). Esta técnica permite tener una total independencia entre los nombres reales de las clases que constituyen los servlets y los nombres a través de los cuales son accesibles.

`getMethod()`: esta función retorna el tipo de petición HTTP utilizado para contactar al servlet. Puede ser útil si un mismo método se utiliza para tratar varios tipos de petición HTTP y así poder determinar el tipo concreto de petición.

`getQueryString()`: este método permite obtener los parámetros pasados en la URL. La cadena de caracteres corresponde exactamente a la generada por el navegador cuando ha construido la petición HTTP. Si hay caracteres que han sido codificados por el navegador, no se descodifican en el retorno de esta función.

`getRequestURL`: este método proporciona la URL usada para contactar el servlet. Los posibles parámetros no se incluyen en el resultado de esta función. El resultado se devuelve usando un objeto `StringBuilder` y no con una cadena de caracteres para facilitar el uso de operaciones de concatenación.

`getLocalAddr()`: este método retorna la dirección IP de la tarjeta de red desde la que la petición HTTP ha llegado al servidor. Este método se usa generalmente cuando se crean filtros para limitar el acceso de un servlet (ver sección Filtros de este capítulo).

`getLocalName()`: este método tiene prácticamente la misma finalidad que el anterior, salvo que permite obtener el nombre del host asociado a la tarjeta de red desde la que se ha recibido la petición HTTP en el servidor. Este método tiene que usarse con paciencia, ya que la resolución inversa de una dirección IP en un nombre de host consume muchos recursos.

`getLocalPort()`: el objetivo de esta función es obtener el número de puerto TCP en el que ha llegado la petición HTTP.

`getRemoteAddr()`: este método devuelve la dirección IP del cliente desde el que se ha generado la petición HTTP. Generalmente, este método se utiliza en la creación de filtros para limitar el acceso a un servlet (ver sección Filtros de este capítulo).

`getRemoteHost()`: este método devuelve el FQDN (*Fully Qualified Domain Name*) del cliente desde el que ha sido generada la petición HTTP. Este método tiene que usarse con paciencia ya que la resolución inversa de una dirección IP en un nombre de host consume muchos recursos.

El ejemplo de código mostrado a continuación ilustra estos distintos métodos:

```
package es.eni.ri;

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

public class InfoURL extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet Información sobre URL</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("nombre del servidor (getServerName): ");
            out.println(request.getServerName());
            out.println("<br>");
            out.println("número de puerto del servidor (getServerPort): ");
            out.println(request.getServerPort());
            out.println("<br>");
            out.println("nombre de la aplicación en el servidor
(getContextPath): ");
            out.println(request.getContextPath());
            out.println("<br>");
            out.println("identificación del servlet (getServletPath): ");
            out.println(request.getServletPath());
            out.println("<br>");
            out.println("método http (getMethod): ");
            out.println(request.getMethod());
            out.println("<br>");
            out.println("parámetros pasados en la petición
getQueryString): ");
            out.println(request.getQueryString());
            out.println("<br>");
            out.println("URL completa (getRequestURL): ");
            out.println(request.getRequestURL());
            out.println("<br>");
            out.println("dirección IP del servidor (getLocalAddr): ");
            out.println(request.getLocalAddr());
            out.println("<br>");
            out.println("nombre del servidor (getLocalName): ");
            out.println(request.getLocalName());
            out.println("<br>");
            out.println("puerto de recepción de la petición (getLocalPort): ");
            out.println(request.getLocalPort());
            out.println("<br>");
            out.println("dirección IP del cliente (getRemoteAddr): ");
            out.println(request.getRemoteAddr());
            out.println("<br>");
            out.println("nombre de la máquina cliente (getRemoteHost): ");
            out.println(request.getRemoteHost());
            out.println("<br>");
            out.println("</body>");
            out.println("</html>");

        }
        finally
        {
    }
```

```

        out.close();
    }
}
@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}

```

2. Leer parámetros

Los parámetros son ciertamente los elementos más usados de una petición HTTP. De hecho, es gracias a ellos que los datos pueden enviarse a un servlet. El protocolo HTTP dispone de dos técnicas distintas para enviar los parámetros del cliente al servidor:

- Añadiendo a continuación de la URL los parámetros usando una cadena de caracteres con la estructura siguiente:

```
?nombre_del_parámetro1=valor_del_parámetro1&nombre_del_parámetro2=valor_del
_parámetro2&...
```

- Añadiendo estos parámetros en el cuerpo de la petición HTTP.

La elección de una técnica u otra está íntimamente relacionada con el tipo de petición HTTP (seguida a la URL para el método GET y en el cuerpo de la petición para el método POST).

Indistintamente del método usado para transmitir los parámetros, éstos se recuperan del mismo modo en el servlet. Puesto que el protocolo HTTP sólo es capaz de transportar texto, los parámetros son transmitidos y recibidos usando cadenas de caracteres. Si alguno de ellos representa un valor numérico, hay que realizar una conversión en el servlet. Los métodos siguientes están disponibles para la manipulación de parámetros de una petición HTTP:

`getParameter(String name)`: este método permite la recuperación del parámetro cuyo nombre se ha especificado. Si el parámetro esperado no existe en la petición HTTP este método devuelve el valor `null`.

`getParameterValues(String name)`: en algunos casos un solo parámetro puede estar presente varias veces en una petición HTTP. Es el caso por ejemplo en el que los datos se recopilan a partir de un formulario HTML que contiene casillas de verificación o una lista que permita selecciones múltiples. Este método permite obtener los valores del parámetro cuyo nombre es pasado como argumento. Devuelve el resultado en una tabla de cadenas de caracteres. Esta tabla se dimensiona automáticamente en función del número de valores del parámetro en especificado. Este método devuelve `null` si el parámetro solicitado no existe en la petición HTTP.

El uso de los dos métodos anteriores exige el conocimiento del nombre del parámetro del que se desea obtener el valor. Es por tanto importante que el creador de páginas HTML y el desarrollador de los servlets se pongan de acuerdo acerca de los nombres utilizados por los parámetros y sobre la información que representan.

Sin embargo, hay disponibles dos métodos para acceder a los parámetros de una petición HTTP incluso si no se conocen sus nombres.

El método `getParameterNames()` permite obtener en un objeto `Enumeration` los nombres de todos los parámetros de una petición HTTP. A continuación, el valor de cada parámetro puede ser obtenido gracias al nombre retirado de la enumeración.

Sin embargo, esta solución no permite saber si un mismo parámetro aparece varias veces en la petición ya que, si es el caso, su nombre sólo está una vez en la enumeración.

El método `getParameterMap` permite proporcionar una solución a este problema devolviendo un objeto `Map` en el que las claves se construyen con los nombres de los parámetros y los valores por tablas de cadena de caracteres que representan el contenido de cada uno de los parámetros.

El ejemplo mostrado a continuación ilustra el uso de cada uno de estos métodos.

```
package es.eni.ri;
```

```

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Iterator;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author tgroussard
 */
public class LecturaParametros extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String nombreParametro=null;
        String[] valoresParametro=null;
        Iterator it;
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet de lectura de parámetros</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<hr>");
            // recuperación de un parámetro único por su nombre
            out.println("apellidos: ");
            out.println(request.getParameter("apellidos"));
            out.println("<br>");
            out.println("nombre: ");
            out.println(request.getParameter("nombre"));
            out.println("<br>");
            // recuperación de valores múltiples de un parámetro
            String[] aficiones=null;
            aficiones=request.getParameterValues("aficiones");
            out.println("aficiones:");
            for(String aficion:aficiones)
            {
                out.println(aficion + " ");
            }
            out.println("<hr>");
            out.println("<br>");
            // recuperación de los nombres de los parámetros
            Enumeration parametros=null;
            out.println("lista de los nombres de los parámetros");
            out.println("<br>");
            parametros=request.getParameterNames();
            while (parametros.hasMoreElements())
            {
                out.println(parametros.nextElement());
            }
            out.println("<hr>");
            out.println("<br>");
            // recuperación de los nombres de los parámetros
            out.println("lista de los parámetros sin saber el nombre");
            out.println("<br>");
            // recuperación de un iterador sobre las claves
            (nombre de los parámetros)
                it=request.getParameterMap().keySet().iterator();
                // bucle de recorrido de la lista de los parámetros
                while(it.hasNext())
                {
                    nombreParametro=(String)it.next();
                    out.println("nombre del parámetro: ");
                    out.println(nombreParametro);

```

```

        out.println("valores del parámetro:");
        valoresParametro=(String[])request.getParameterMap().
get(nombreParametro);
        for(String valorParametro:valoresParametro)
        {
            out.println(valorParametro);
            out.println(";");
        }
        out.println("<br>");
    }
} finally {
    out.close();
}
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}

```

El formulario utilizado para contactar con el servlet:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
    <form action="LecturaParametros" method="get"><table border="0">
        <tbody>
            <tr>
                <td>apellidos</td>
                <td><input type="text" name="apellidos" value="" size="20" /></td>
            </tr>
            <tr>
                <td>nombre</td>
                <td><input type="text" name="nombre" value="" size="20" /></td>
            </tr>
            <tr>
                <td>aficiones</td>
                <td>
                    <input type="checkbox" name="aficiones" value="musica">música</input>
                    <input type="checkbox" name="aficiones" value="cine">cine</input>
                    <input type="checkbox" name="aficiones" value="jardineria">jardinería</input>
                    <input type="checkbox" name="aficiones" value="bricolaje">bricolaje</input>
                </td>
            </tr>
            <tr>
                <td></td>
                <td> <input type="submit" value="aceptar" name="validacion" /></td>
            </tr>
        </tbody>
    </table>
</form>

```

```

        </table>
    </form>
</body>
</html>

```

3. Leer cabeceras

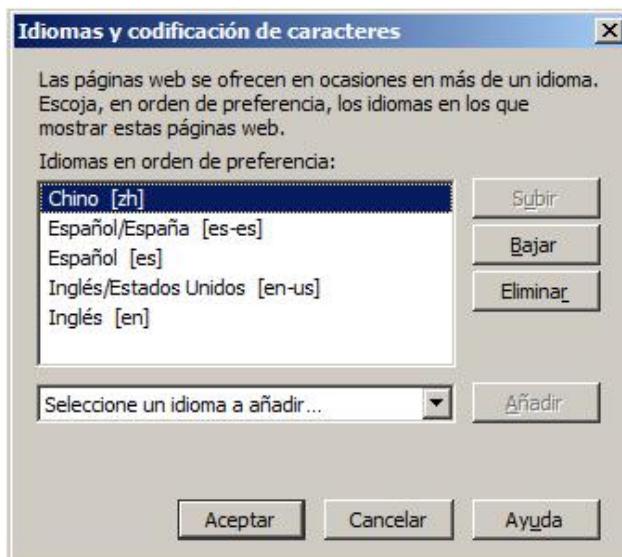
Las cabeceras de petición HTTP se usan para transportar datos del cliente (el navegador) hacia el servidor. Contrariamente a lo que sucede con los parámetros, cuyos valores son facilitados por el usuario, las cabeceras son generadas por el navegador. Es por lo tanto éste el que determina en el momento de la creación de la petición HTTP qué cadenas se agregarán y cuáles son sus respectivos valores. Despues, estos datos se analizan en el lado servidor para adaptar la respuesta HTTP. En muchos casos, el servidor usa directamente las cabeceras cuando analiza una petición HTTP antes de delegar el tratamiento en un servlet.

El objeto `HttpServletRequest` proporciona varios métodos para la manipulación de cabeceras. Algunos de estos métodos se especializan en la manipulación de una cabecera determinada. Típicamente es el caso de las cabeceras HTTP estándar. Para el resto, hay métodos universales que permiten la manipulación de cualquier tipo de cabecera HTTP.

`getContentLength()`: este método devuelve el tamaño en bytes del cuerpo de la petición HTTP. Si no hay cuerpo en la petición, como en el caso de una petición GET, este método devuelve -1.

`getContentType()`: este método permite determinar el tipo de datos presente en el cuerpo de la petición HTTP. Sólo es útil para las peticiones de tipo POST. Para una petición HTTP generada a partir de datos recopilados de un formulario HTML este método devuelve el valor `application/x-www-form-urlencoded`.

`getLocale()`: este método proporciona el idioma preferido para la respuesta HTTP. Si el servlet es capaz de generar un resultado en varios idiomas, si está disponible, debe ser éste el usado para la respuesta HTTP. Si este idioma no está disponible, la respuesta debe realizarse con el idioma por defecto del servlet. La técnica de configuración del idioma preferido depende del navegador. Algunos de ellos se basan en el idioma configurado a nivel de sistema, otros proponen una opción de configuración específica. A continuación se muestra el cuadro de diálogo de Firefox que permite realizar esta configuración y el efecto que produce esta configuración en la visualización de la página de inicio de Google.





`getLocale()`: el navegador también puede proporcionar una lista de idiomas favoritos que puede recuperarse en un objeto de tipo `Enumeration` con este método.

Cada navegador utiliza su propia técnica para la construcción de una petición HTTP y genera peticiones HTTP que pueden contener muchas cabeceras distintas. Por tanto, no se puede tener un método específico para la recuperación de cada uno de estos tipos de cabecera. El método `getHeader(String name)` proporciona una solución que permite la recuperación de la cabecera cuyo nombre se le ha pasado por parámetro. El valor de la cabecera devuelta es una cadena de caracteres. Este valor tiene que convertirse a continuación en el tipo correcto de datos para poder utilizarse.

Los métodos `getDateHeader(String name)` y `getIntHeader(String name)` son un poco más eficaces porque además realizan una conversión a tipo `long` que representa una fecha o a tipo `int` en la recuperación de la cabecera. Por contra, ambos métodos son más peligrosos porque pueden desencadenar una excepción si no es posible la conversión al tipo deseado.

Algunos navegadores pueden generar varias veces la misma cabecera con valores diferentes en vez de crear una única cabecera con una lista de valores. Para este caso un poco particular, el método `getHeaders(String name)` devuelve en un objeto `Enumeration` los valores de todas las cabeceras que tengan el nombre indicado por el parámetro `name`.

Los nombres de las cabeceras no están totalmente estandarizados. Es difícil imaginar todas las cabeceras que un navegador puede colocar en una petición HTTP. Para permitir el uso de estas cabeceras específicas el método `getHeaderNames()` devuelve en un objeto `Enumeration` la lista de todos los nombres de las cabeceras utilizadas en la petición. El o los valores pueden obtenerse con los métodos `getHeader(String name)` o `getHeaders(String name)`.

En el siguiente ejemplo muestra el nombre y el valor de cada cabecera recibida en la petición HTTP.

```
package es.eni.ri;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.Enumeration;
import java.util.Iterator;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
```

```

import javax.servlet.http.HttpServletResponse;

public class LecturaCabeceras extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    String nombreCabecera=null;
    Enumeration nombresCabeceras=null;
    Enumeration valoresCabeceras=null;
    try {

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet LecturaCabeceras</title>");
        out.println("</head>");
        out.println("<body>");
        nombresCabeceras=request.getHeaderNames();
        out.println( "<table border=\"1\" cellpadding=\"10\">" );
        out.println("<tbody>");
        while (nombresCabeceras.hasMoreElements())
        {
            nombreCabecera=(String)nombresCabeceras.nextElement();
            out.println("<tr>");
            out.println("<td>" + nombreCabecera + "</td>");
            valoresCabeceras=request.getHeaders(nombreCabecera);
            while (valoresCabeceras.hasMoreElements())
            {
                out.println("<td>" +
valoresCabeceras.nextElement() + "</td>");
            }
            out.println("</tr>");
        }
        out.println("</tbody>");
        out.println("</table>");
        out.println("<br>");
        out.println("</body>");
        out.println("</html>");
    }

    finally {
        out.close();
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}

```

Las dos impresiones de pantalla siguientes ilustran las diferencias en las cabeceras de las peticiones HTTP en función de los navegadores (en este caso Internet Explorer y Firefox).

Servlet LecturaCabeceras - Windows Internet Explorer

accept	image/gif, image/jpeg, image/pjpeg, image/png, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */*
accept-language	es
user-agent	Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.1)
accept-encoding	gzip, deflate
host	192.168.1.33:8080
connection	Keep-Alive

Servlet LecturaCabeceras - Mozilla Firefox

host	192.168.1.33:8080
user-agent	Mozilla/5.0 (Windows; U; Windows NT 5.1; es-ES; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.3
accept	text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language	es-es,es;q=0.8,en-us;q=0.5,en;q=0.3
accept-encoding	gzip,deflate
accept-charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive	115
connection	keep-alive
cache-control	max-age=0

4. Añadir información a la petición

Después de su transformación en objeto `HttpServletRequest`, la petición HTTP transita por muchos tratamientos. Pasa por ejemplo por el método `service` después por un método `doXXX` y finalmente puede transferirse a una página JSP. Los distintos tratamientos por los que pasa pueden añadirle elementos adicionales mediante los atributos. Un servlet puede por ejemplo extraer datos de una base de datos y transmitirlos a una página JSP para su visualización.

Los cuatro métodos siguientes permiten la manipulación de los atributos de un objeto `HttpServletRequest`.

`setAttribute(String name, Object o)`: este primer método se encarga del almacenamiento de un objeto en el objeto `HttpServletRequest`. El primer parámetro de tipo cadena de caracteres identifica el objeto en la lista de atributos. El segundo parámetro representa el objeto que se almacenará.

`getAttribute(String name)`: este método es el complementario del anterior debido a que permite extraer un objeto que se ha almacenado previamente. Acepta como parámetro una cadena de caracteres que representa el nombre utilizado para buscar el objeto en los atributos. Esta cadena tiene que ser idéntica a la usada cuando se almacenó el objeto en los atributos. Este método devuelve un tipo `Object` lo que obliga con mucha frecuencia a realizar conversión de tipos para utilizar el objeto extraído de la lista de atributos.

`getAttributeNames()`: este método proporciona una `Enumeration` de la lista de nombres de atributos de la petición.

`removeAttribute(String name)`: este método permite la eliminación de un objeto almacenado en la lista de atributos de la petición. No se usa con frecuencia ya que en general es la destrucción del objeto `HttpServletRequest` lo que provoca la destrucción de los atributos.

Construir la respuesta HTTP

La respuesta HTTP permite el tránsito de datos desde el servidor al cliente. Como en el caso de la petición, la respuesta HTTP se forma con una cadena de caracteres con un formato determinado. Se divide en dos partes, la cabecera de la respuesta y el cuerpo de la respuesta. Es ésta cadena de caracteres la que el servidor envía al cliente. Para facilitar la manipulación de la respuesta HTTP, el servidor crea una instancia de una clase que implementa la interfaz `HttpServletResponse`. Las manipulaciones en la respuesta HTTP se realizan, por tanto, mediante los métodos de esta clase.

1. Definir el estado de la respuesta

El estado de la respuesta representa la primera línea de la cabecera de respuesta. Se usa por el cliente para determinar el resultado del tratamiento de la petición por el servidor. Se construye con un valor numérico y un mensaje informativo. Los códigos de estado se reparten en cinco categorías.

1XX: informativo

2XX: éxito

3XX: redirección

4XX: error imputable al cliente

5XX: error imputable al servidor

El detalle de cada uno de estos códigos está disponible en la RFC 2616 (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>).

El código de estado por defecto de una respuesta HTTP es 200 para indicar que el tratamiento de la petición se ha desarrollado correctamente.

El método `setStatus(int sc)` permite especificar el estado de la respuesta enviada al cliente. El parámetro esperado por este método corresponde al código de estado. Para una mejor legibilidad del código se aconseja utilizar una de las constantes definidas en la interfaz `HttpServletResponse`. El mensaje asociado es el mensaje estándar tal y como se define en la RFC 2616.

Para personalizar este mensaje, hay que utilizar el método `sendError(int sc, String msg)`. Este método espera como parámetros el código de estado y el mensaje personalizado que se desea asociarle.

El ejemplo siguiente devuelve al cliente una respuesta con un código de estado 503 y un mensaje personalizado.

```
package es.eni.ri;

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

public class ResponseHTTP extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {

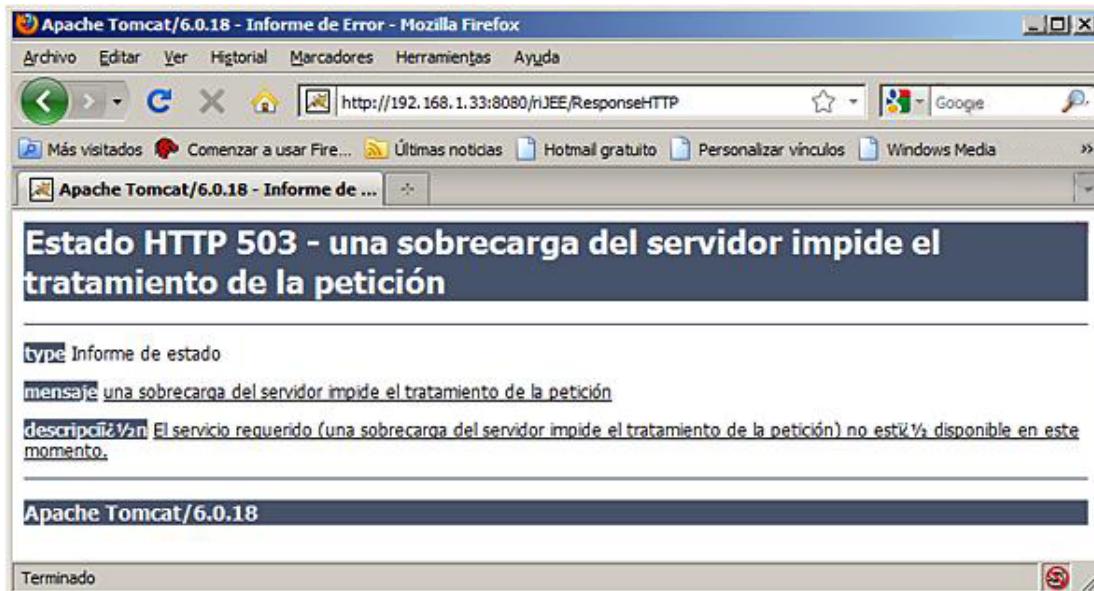
            response.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE ,
"una sobrecarga del servidor impide el tratamiento de la petición");
        }
        finally
        {
            out.close();
        }
    }
}
```

```

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
}

```

El navegador muestra la página siguiente:



2. Agregar cabeceras

Las cabeceras de respuesta se usan para añadir a la respuesta HTTP información adicional que facilita el tratamiento que le realizará el navegador. El servidor ya añade automáticamente algunas cabeceras en la respuesta HTTP. Para otros casos, hay que intervenir a nivel de código del servlet y añadir manualmente las cabeceras de respuesta. Se ofrece un abanico de métodos para la manipulación de cabeceras de respuesta. Algunos de los cuales se han creado para gestionar específicamente una cabecera en concreto.

Éste es el caso por ejemplo del método `setContentType(String type)`. Este método permite indicar la naturaleza de los datos incluidos en el cuerpo de la respuesta. El parámetro de tipo cadena de caracteres representa el tipo MIME (*Multipurpose Internet Mail Extensions*) del documento incluido en el cuerpo de la respuesta. Para algunos navegadores esta información es esencial y en su ausencia consideran el documento recibido como de texto plano. Sin embargo, la gran mayoría es capaz de determinar el tipo del documento en función de su contenido.

El idioma en el que se envía el documento que compone la respuesta puede indicarse utilizando el método `setLocale(Locale loc)`. Este método espera como parámetro un objeto `Locale` representante del idioma del documento. Este objeto puede obtenerse con una de las muchas constantes definidas en la clase `Locale`.

Los métodos siguientes son más generales puesto que permiten agregar cualquier cabecera a una respuesta HTTP. El método `setHeader(String name, String value)` añade a la cabecera especificada por el primer parámetro el valor especificado en el segundo parámetro. Si una cabecera de igual nombre ya existe en la respuesta, su valor es sobreescrito.

Para evitar este problema el método `addHeader(String name, String value)` tiene un funcionamiento un poco diferente debido a que no sobreescribe nunca un valor ya existente sino que añade a la cabecera un valor adicional. Ésta es sin lugar a dudas la solución que utiliza el servidor para agregar sus propias cabeceras a la respuesta HTTP.

No existe sin embargo un método para eliminar una cabecera ya agregada a una respuesta HTTP.

La manipulación de cabeceras de respuesta tiene que realizarse de forma obligada antes de la escritura de

información en el cuerpo de la respuesta HTTP.

3. Construir el cuerpo de la respuesta

La escritura de datos en el cuerpo de la respuesta es muy sencilla de implementar debido a que consiste en enviar los datos hacia un flujo de salida. Es exactamente el mismo principio que el usado para la escritura en la consola Java. Hay que hacer justamente una pequeña distinción en función del tipo de los datos que se envían al cliente.

Para escribir texto en el cuerpo de la respuesta HTTP se utiliza preferentemente un objeto `PrintWriter`. Este objeto es accesible por el método `getWriter` del objeto `HttpServletResponse`.

Los datos binarios se añaden gracias a un objeto `ServletOutputStream` accesible mediante el método `getOutputStream` del objeto `HttpServletRequest`.

Nunca hay que usar simultáneamente ambos objetos ya que en este caso se desencadena una excepción de tipo `IllegalStateException`.

El ejemplo de servlet mostrado a continuación devuelve una imagen en formato JPEG al cliente.

```
package es.eni.ri;

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

public class ResponseHttp extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        ServletOutputStream out = null;

        try {
            response.setContentType("image/jpg");
            RandomAccessFile raf = new RandomAccessFile(new
File(getServletContext().getRealPath("/images/titi.jpg")), "r" );
            response.setContentLength( (int) raf.length() );
            out = response.getOutputStream();
            byte [] datas = new byte [ (int) raf.length() ];
            while ( (raf.read( datas )) > 0 )
            {
                out.write( datas );
            }
            out.flush();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
```

```
{
    processRequest(request, response);
}
}
```

El mismo resultado puede obtenerse con la versión siguiente del servlet que es más simple.

```
package es.eni.ri;

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

public class ResponseHttp2 extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        response.sendRedirect("images/titi.jpg");
    }

    @Override
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        processRequest(request, response);
    }
}
```

Sin embargo, el funcionamiento no es realmente idéntico porque en el primer ejemplo el servlet devuelve él mismo el contenido de la imagen al navegador. En el segundo ejemplo devuelve una respuesta HTTP de redirección al navegador con lo que provoca que éste cree una nueva petición HTTP para obtener el contenido de la imagen. Por tanto, en este caso son dos peticiones HTTP las que se necesitan para visualizar la imagen.

La información escrita en la respuesta HTTP no es inmediatamente transmitida al cliente sino que se copia en un buffer. El tamaño y el uso de este buffer pueden controlarse con los métodos siguientes.

`setBufferSize(int size)`: este método especifica el tamaño del buffer usado para la construcción del cuerpo de la respuesta HTTP. Un tamaño grande permite limitar el número de paquetes TCP enviados por la red para la transmisión de la respuesta al cliente. Como contrapartida, el servidor necesitará una cantidad mayor de memoria para funcionar. En cambio, un tamaño menor permite al cliente recibir más rápidamente los primeros datos que componen la respuesta además de usar menos recursos de memoria en el servidor. Este método tiene que llamarse obligatoriamente antes de la escritura de datos en la respuesta HTTP.

`int getBufferSize()`: este método devuelve el tamaño actual del buffer de memoria.

`reset()`: este método reinicializa completamente el buffer de la respuesta HTTP eliminando el código de estado, las cabeceras de respuesta y el cuerpo de la respuesta. No se puede invocar este método si los datos que conciernen a la respuesta ya han sido transmitidos al cliente (no es posible borrar datos que ya han llegado a destino en el cliente!).

`resetBuffer()`: este método es menos radical que el anterior ya que sólo elimina el cuerpo de la respuesta HTTP. Sin embargo, tiene la misma restricción que el método `reset`.

`boolean isCommitted()`: este método determina si los datos ya han sido enviados al cliente para poder llamar sin riesgo a uno de los dos métodos anteriores.

`flushBuffer()`: este método fuerza el envío del contenido del buffer al cliente. Después de la llamada a este método, la cabecera de respuesta HTTP no puede modificarse.

Para ilustrar el uso de estos métodos, retomamos el ejemplo anterior y lo modificamos ligeramente para que el archivo contenedor de la imagen no se lea en una sola operación sino que se lea byte a byte. Para evidenciar la importancia del buffer, basta con variar el tamaño de éste modificando la línea `response.setBufferSize(2000);`.

Tomemos por ejemplo una prueba con un tamaño de 2000 bytes y luego con un tamaño algo mayor. En el primer caso verá aparecer la imagen en bandas sucesivas. En el segundo caso la imagen aparecerá de una sola vez.

Para llevar la experiencia al extremo, puede colocar la instrucción `response.flushBuffer();` al interior del bucle `while`. La imagen se ha transmitido entonces byte a byte al navegador.

```
package es.eni.ri;

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

public class ResponseHttp extends HttpServlet
{

protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    ServletOutputStream out = null;

try {
    response.setContentType("image/jpg");
    //línea que se modifica
    response.setBufferSize(2000);
    RandomAccessFile raf = new RandomAccessFile(new
File(getServletContext().getRealPath("/images/PICT0871.jpg")), "r" );
    response.setContentLength( (int) raf.length() );
    out = response.getOutputStream();
    int b;
    while ( (b=raf.read( )) !=-1 )
    {
        out.write( (byte)b );
    }
    // la línea siguiente puede colocarse en
    // el interior del bucle
    response.flushBuffer();
}
@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
}
```

Elementos accesibles desde un servlet

Una aplicación web dispone de varios espacios de almacenamiento que le permiten guardar en memoria datos esenciales para su buen funcionamiento. Los objetos puestos en estos espacios de almacenamiento poseen cada uno una duración de vida distinta y una visibilidad específica.

1. Contexto de aplicación

Una aplicación generalmente está constituida por varios servlets, páginas JSP, páginas HTML y otros recursos. Estos elementos son agrupados por el servidor de aplicaciones en un contenedor llamado contexto de aplicación. Además de contener los servlets y las páginas JSP de la aplicación, éste también puede alojar datos relacionados con la configuración de la aplicación y datos guardados por los servlets y las páginas JSP. Este contexto de aplicación se representa con una instancia de una clase que implementa la interfaz `ServletContext`. Esta interfaz de clase es accesible desde un servlet o una página JSP mediante el método `getServletContext()`.

La información relacionada con la configuración de la aplicación es accesible con el método `String getInitParameter(String name)`. La cadena de caracteres devuelta por este método contiene el valor del parámetro de inicialización de la aplicación cuyo nombre se le proporciona como parámetro. Si el parámetro no existe, este método devuelve el valor `null`. Los parámetros de inicialización tienen que declararse en el descriptor de despliegue de la aplicación. Para cada parámetro hay que añadir en la sección `<web-app>` una etiqueta `<context-param>` con el nombre del parámetro y su valor. Cuando se inicia la aplicación, estos datos son transferidos por el servidor en el contexto de aplicación. La porción siguiente del archivo `web.xml` define dos parámetros de inicialización.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>nombreciente</param-name>
    <param-value>eni ecole</param-value>
  </context-param>
  <context-param>
    <param-name>logocliente</param-name>
    <param-value>eniecole.gif</param-value>
  </context-param>
</web-app>
```

Estos parámetros pueden usarse en todos los servlets y en todas las páginas JSP de la aplicación. Los siguientes ejemplos de código utilizan estos parámetros para mostrar la página y el logo de la empresa desde un servlet y desde una página JSP.

Ejemplo de servlet:

```
package es.eni.ri;

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

public class ParametrosInicializacion extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ParametrosInicializacion</title>");
```

```

        out.println("</head> ");
        out.println("<body> ");
        out.println("<h1> ");
        out.println(" <img src=\"images/" +
getServletContext().getInitParameter("logoCliente")+" \""
width="\\"50\\" height="\\"50\\" alt=\"logo\"/>\"");
        out.println(getServletContext().getInitParameter("nombreCliente") +
"</h1> ");
        out.println("</body> ");
        out.println("</html> ");

    }
    finally
    {
        out.close();
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
}

```

Ejemplo de página JSP:

```

<%--
 Document      : ParametrosInicializacion
 Created on   : 10 nov. 2009, 11:34:05
 Author       : tgroussard
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>
            " width="50" height="50" alt="logo"/>
            <%=getServletContext().getInitParameter("nombreCliente")%>
        </h1>
    </body>
</html>

```

- Hay que poner especial atención cuando se utilice el método `getInitParameter` del contexto de aplicación ya que este método también existe para un servlet y permite obtener los parámetros de inicialización de éste.

Los atributos del contexto de aplicación funcionan según el mismo principio que los atributos de petición vistos en la sección Añadir información a la petición de este capítulo. Los métodos `void setAttribute(String name, Object`

object) y Object getAttribute(String name) permiten respectivamente añadir un atributo en el contexto de aplicación o leer un atributo del contexto de aplicación. Todos los elementos de código de la aplicación tienen acceso a estos atributos. Son considerados a veces como variables globales de la aplicación.

El ejemplo siguiente guarda en el contexto de la aplicación la fecha y la hora de inicio de un servlet. Esta información es a continuación utilizada en una página JSP.

Código del servlet:

```
package es.eni.ri;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.GregorianCalendar;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AtributosContext extends HttpServlet {

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        getServletContext().setAttribute("horaArranque", new
GregorianCalendar());
    }

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet AtributosContext</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("");
        out.println("el primer arranque de este servlet se ha
efectuado el ");
        out.println((GregorianCalendar)getServletContext().getAttribute
("horaArranque")).getTime();
        out.println(" <a href=\"parametrosAplicacion.jsp\">a la
página jsp </a>");
        out.println("</body>");
        out.println("</html>");

    } finally {
        out.close();
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}
```

Código de la página JSP:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="java.util.GregorianCalendar"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    el servlet de origen se ha iniciado el
<%=((GregorianCalendar)getServletContext().getAttribute("horaArranque"))
.getTime()%>
  </body>
</html>
```

2. Sesión

El principio de sesión es casi lo mismo que el contexto de aplicación ya que permite el almacenamiento de datos accesibles desde toda la aplicación. Sin embargo, existe una diferencia muy importante porque con la utilización del contexto de aplicación el mismo dato se comparte entre todos los clientes que acceden a la aplicación. Con la sesión los datos almacenados están asociados a cada cliente. El uso y el funcionamiento de las sesiones se abordan en un capítulo específico.

Utilización de otros recursos

El tratamiento de una petición HTTP puede compartirse entre varios servlets o un servlet y una página JSP. También es posible que pueda necesitar a veces recursos estáticos como por ejemplo una página HTML. Hay tres soluciones posibles para obtener la colaboración entre recursos en el transcurso del tratamiento de una petición HTTP.

1. Utilización del RequestDispatcher

El único medio para obtener un recurso de la parte del servidor es enviándole una petición HTTP. Esta obligación se aplica incluso en el caso de un servlet que quiere utilizar el recurso. Para permitir la transferencia de una petición HTTP a otro recurso, hay que obtener un objeto RequestDispatcher. Este objeto puede obtenerse desde el contexto de aplicación. Éste contiene todos los recursos de la aplicación y es por tanto capaz de proporcionar un objeto que puede transferir una petición HTTP a uno de ellos. El método `getRequestDispatcher` devuelve un objeto RequestDispatcher. Espera simplemente como parámetro una cadena de caracteres que identifiquen el recurso hacia el que la petición HTTP tiene que transferirse. Esta cadena de caracteres tiene que empezar con el carácter "/" para indicar que la ruta se expresa desde la raíz de la aplicación.

Para utilizar un recurso disponible en otra aplicación albergada en el mismo servidor, el proceso es un poco más complejo, ya que debido a que hay que obtener previamente una referencia al contexto de la aplicación que alberga el recurso deseado. El método `getContext` que está disponible desde el contexto actual espera como argumento el nombre de esta aplicación (comenzando con "/" para indicar que se especifica el nombre desde la raíz del servidor). A continuación, el objeto RequestDispatcher se obtiene desde este contexto. Finalmente, a partir de este objeto RequestDispatcher, hay dos soluciones disponibles para utilizar el recurso.

a. Include

Esta solución permitirá incluir otro recurso en la respuesta HTTP generada por el servlet. Este mecanismo se suele usar para insertar en la respuesta fragmentos de código HTML escrito en documentos independientes. Evita la presencia en un servlet de muchas instrucciones `println` colocando en el cuerpo de la respuesta código HTML. Otra ventaja no despreciable reside también en la facilidad de mantenimiento y de modificación de la aplicación ya que el código HTML que se insertará es independiente de todo servlet.

El método `include` del objeto RequestDispatcher espera como parámetros la petición HTTP y la respuesta HTTP que a continuación podrá utilizar el recurso incluido. Sin embargo, hay una pequeña limitación que concierne a la respuesta transferida al recurso incluido debido a que éste no puede modificar la parte de la cabecera de esta respuesta HTTP. Por contra, puede actuar libremente en el cuerpo de la respuesta HTTP. Sin embargo, no se presenta ninguna limitación en cuanto al uso de la petición HTTP por el recurso incluido.

El ejemplo siguiente presenta esta solución en la que el inicio y el final de la página que constituye la respuesta se externalizan en los archivos `cabecera.html` y `pie.html`. El servlet realiza la inclusión de estos dos archivos en el contenido de la respuesta que genera.

Archivo `cabecera.html`

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <table>
      <tr>
        <td>
          
        </td>
        <td>
          <h1>ENI ECOLE</h1>
        </td>
      </tr>
    </table>
    <hr>
```

Archivo `pie.html`

```
<hr>
```

```

<h5>calle Benjamin Franklin, 44800 Saint Herblain (Francia)</h5>
</body>
</html>

```

Código del servlet:

```

package es.eni.ri;

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

public class Inclusion extends HttpServlet {

protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    try {
        RequestDispatcher rd=null;
        rd=getServletContext().getRequestDispatcher("/cabecera.html");
        rd.include(request, response);
        out.println("este texto lo ha generado el servlet");
        rd=getServletContext().getRequestDispatcher("/pie.html");
        rd.include(request, response);

    } finally {
        out.close();
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
}

```

b. Forward

Con esta solución el tratamiento de la petición HTTP se ejecutará en varias etapas. El primer servlet que recibe la petición realizará un primer tratamiento de ésta y después delegará a otro recurso la tarea de construir la respuesta HTTP. El esquema clásico consiste en realizar en el primer servlet una búsqueda de información en una base de datos sin preocuparse de la presentación del resultado. Los datos en bruto procedentes de la base de datos se asocian a la petición HTTP en forma de atributos. A continuación, la petición HTTP se transmite a otro recurso (generalmente una página JSP) que extraerá los datos de los atributos de la petición y después se encarga de la presentación para el cliente generando la respuesta HTTP. Como sucede con el método `include`, el método `forward` espera recibir por parámetro la petición y la respuesta HTTP. Después, ambos elementos pasan a ser accesibles desde el recurso al que se ha confiado el final del tratamiento. Este método tiene también varias restricciones.

- El servlet de origen no puede haber enviado ya datos al cliente. Si éste es el caso, se produce el

desencadenamiento de una excepción `IllegalStateException`.

- Si hay datos en el buffer de respuesta pero todavía no han sido enviados al cliente, éstos se borran cuando el control pasa al segundo recurso.
- Cuando la petición HTTP se transfiere al segundo recurso, la URL de origen se reemplaza por la URL de este segundo recurso. Si la URL de origen es necesaria para continuar el tratamiento, hay que copiarla y guardarla antes de la llamada al método `forward`.

El ejemplo siguiente ilustra la utilización de esta técnica. Permite calcular el rendimiento de una inversión financiera. Una página HTML solicita la introducción del capital inicial, de la duración y del interés. Estos datos se envían a un servlet que realiza los cálculos y genera una tabla de doble que contiene los resultados. A continuación el control pasa a una página JSP que se encarga de la presentación de estos resultados.

Página HTML de introducción de los valores de cálculo:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <title></title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" >
    </head>
    <body>
        <form method="GET" action="RendimientoInversion">
            <table border="0">
                <tbody>
                    <tr>
                        <td>capital inicial</td>
                        <td><input type="text" name="capital" value="" /></td>
                    </tr>
                    <tr>
                        <td>interés</td>
                        <td><input type="text" name="interes" value="" /></td>
                    </tr>
                    <tr>
                        <td>duración</td>
                        <td><input type="text" name="duracion" value="" /></td>
                    </tr>
                    <tr>
                        <td></td>
                        <td><input type="submit" value="calcular" /></td>
                    </tr>
                </tbody>
            </table>
        </form>
    </body>
</html>
```

Código servlet que realiza los cálculos:

```
package es.eni.ri;

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

public class RendimientoInversion extends HttpServlet
{
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
    }
}
```

```

PrintWriter out = response.getWriter();
try
{
    int duracion;
    double capital;
    double interes;
    duracion=Integer.parseInt(request.getParameter("duracion"));
    capital=Double.parseDouble(request.getParameter("capital"));
    interes=Double.parseDouble(request.getParameter("interes"));
    double resultado[][][];
    resultado=new double[duracion+1][3];
    resultado[0][0]=0;
    resultado[0][1]=0;
    resultado[0][2]=capital;
    for (int i=1;i<=duracion;i++)
    {
        resultado[i][2]=resultado[i-1][2]*(1+(interes/100));
        resultado[i][1]=resultado[i][2]-capital;
        resultado[i][0]=resultado[i-1][2]*(interes/100);
    }

    RequestDispatcher rd=null;
    rd =getServletContext().getRequestDispatcher("/resultados.jsp");
    request.setAttribute("resultado", resultado);
    rd.forward(request, response);
} finally {
    out.close();
}
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    processRequest(request, response);
}
}

```

Página JSP que muestra los resultados:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="java.text.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <%
            double resultado[][][];
            resultado=(double[][][]) request.getAttribute("resultado");
        %>
        <table border="1">
            <thead>
                <tr>
                    <th>año</th>
                    <th>intereses del año</th>

```

```

        <th>intereses acumulados</th>
        <th>capital</th>
    </tr>
</thead>
<tbody>
<%
    DecimalFormat ft=null;
    ft=new DecimalFormat("0.00");
    for (int i=0;i<resultado.length;i++)
    {
%
%>

    <tr>
        <td><%=i%></td>
        <td><%=ft.format(resultado[i][0])%></td>
        <td><%=ft.format(resultado[i][1])%></td>
        <td><%=ft.format(resultado[i][2])%></td>
    </tr>
<%
}
%
%>
</tbody>
</table>

</body>
</html>

```

2. Redirección

El funcionamiento de las redirecciones puede parecer similar al uso de un forward en un servlet pero hay sin embargo una gran diferencia en su modo de funcionamiento. En el caso de un forward, el servidor transfiere la petición HTTP al recurso de destino. Con las redirecciones el servidor informa al navegador que tiene que realizar una nueva petición HTTP con destino la URL indicada. Para ello, el servidor devuelve al navegador una respuesta HTTP 301 o 302 que contiene la URL hacia la que el navegador tiene que mandar una nueva petición HTTP. La redirección puede realizarse de dos formas distintas.

La primera solución consiste en modificar el estado de la respuesta HTTP llamando al método `setStatus` con una de las dos constantes siguientes como parámetro:

- `HttpServletResponse.SC_MOVED_PERMANENTLY` para una respuesta HTTP de tipo 301.
- `HttpServletResponse.SC_MOVED_TEMPORARILY` para una respuesta HTTP de tipo 302.

La distinción entre ambos tipos de redirección es mínima y afecta principalmente a las acciones de los motores de búsqueda. Si el motor de búsqueda recibe una respuesta HTTP 301 cuando intenta indexar un recurso, considerará que el recurso ha cambiado definitivamente de dirección y por tanto actualizará los datos que tenía previamente sobre este recurso. Si recibe una respuesta HTTP 302, considerará que la eliminación es temporal y que el recurso próximamente volverá a estar disponible en su ubicación original.

Para que el navegador sepa hacia qué URL tiene que generar la nueva petición HTTP, hay que añadir la cabecera de respuesta llamada `location` y especificar como valor para ésta la URL hacia la que la nueva petición tiene que enviarse.

```

response.setStatus(HttpServletResponse.SC_MOVED_PERMANENTLY);
response.setHeader("location", "http://www.eni-ecole.fr");

```

La segunda solución se basa en el uso del método `sendRedirect` al que hay que proporcionar como parámetro la URL hacia la que tiene que enviarse la nueva petición HTTP. Obligatoriamente, el tipo de respuesta en este caso es 302 (redirección temporal).

```

response.sendRedirect("http://www.eni-ecole.fr");

```

Filtros

Los filtros se usan para añadir un tratamiento para que se realice en una petición HTTP antes de que el servlet la reciba o en una respuesta HTTP antes de que se transmita al cliente. Entre los usos más comunes de los filtros, cabe mencionar:

- La descompresión de los datos procedentes de los clientes o la compresión de los datos antes del envío a los clientes.
- El descifrado de datos procedentes de los clientes o el cifrado de datos antes del envío a los clientes.
- La conversión de datos recibidos de clientes o enviados a los clientes.
- El control de acceso realizado por los clientes.
- El registro de acceso.

Varios filtros pueden ejecutarse sucesivamente en el trayecto de una petición o de una respuesta HTTP. El orden de ejecución se corresponde con el orden de declaración de los filtros en el descriptor de despliegue de la aplicación (web.xml).

1. Creación

Un filtro se compone de una clase Java que implementa la interfaz `javax.servlet.Filter`. Esta interfaz define los métodos utilizados por el servidor de aplicaciones durante la utilización del filtro. El principio es prácticamente similar al ya usado por un servlet ya que hay tres métodos que tienen que estar disponibles para ser ejecutados durante la vida del filtro en el servidor.

El servidor ejecuta el método `void init(FilterConfig filterConfig)` una vez que éste ha creado una instancia del filtro. El parámetro `FilterConfig` permite acceder a los datos de configuración del filtro definidos en el descriptor de despliegue de la aplicación. Gracias a este parámetro, se puede acceder a los siguientes datos:

- El nombre del filtro se obtiene con el método `String getFilterName()`.
- El contexto de la aplicación en el que se encuentra el filtro lo proporciona el método `ServletContext getServletContext()`.
- Los parámetros de inicialización se recuperan individualmente con el método `String getInitParameter(String name)` al que hay que proporcionar el nombre del parámetro utilizado en el descriptor de despliegue.
- El método `Enumeration getInitParameterNames()` devuelve los nombres de los parámetros de inicialización en un objeto `Enumeration`.

El método `void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)` se invoca cada vez que el filtro tiene que entrar en acción. Este método recibe la petición y la respuesta HTTP para analizarlas así como un objeto `FilterChain` que se utiliza para llamar al filtro siguiente si hay varios filtros configurados para este servlet en el descriptor de despliegue. El método `doFilter(ServletRequest request, ServletResponse response)` de este objeto permite transferir el control al filtro siguiente o al recurso web solicitado en la petición HTTP si el filtro es el único que está configurado o es el último de la lista. Si este método no se invoca, el tratamiento de la petición se detiene y no se devolverá ninguna respuesta al cliente.

El método `void destroy()` es el último que se ejecuta por el filtro ya que se invoca por el servidor antes de que éste destruya el filtro.

2. Declaración

El servidor de aplicaciones es el responsable de la ejecución del primer filtro durante el tratamiento de una petición HTTP. Por lo tanto, tiene que saber cuándo llega una petición. Como en el caso de los servlets, los filtros tienen que declararse en el descriptor de despliegue de la aplicación. El principio de declaración es sin duda muy similar ya que un filtro se declara en dos etapas:

- La declaración de la clase que constituye el filtro.
- La asociación del filtro con una o varias URL o servlets.

La estructura de declaración de un filtro tiene por lo tanto el siguiente aspecto.

```
<filter>
    <description>.....</description>
    <filter-name>.....</filter-name>
    <filter-class>.....</filter-class>
    <init-param>
        <description>.....</description>
        <param-name>.....</param-name>
        <param-value>.....</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>.....</filter-name>
    <url-pattern>.....</url-pattern>
    <servlet-name>.....</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>.....</filter-name>
    <url-pattern>.....</url-pattern>
    <servlet-name>.....</servlet-name>
    <dispatcher>.....</dispatcher>
</filter-mapping>
```

El elemento `<filter>` es la declaración del filtro. Éste incluye una descripción opcional del filtro, un nombre del filtro (obligatorio) para poder referenciarlo en el resto del descriptor de despliegue y el nombre de la clase Java que el servidor deberá instanciar en la creación del filtro. Si el filtro necesita parámetros de inicialización, éstos tienen que declararse en una o varias secciones `<init-param>`. Cada sección `<init-param>` contiene una descripción opcional del parámetro, el nombre del parámetro y el valor del parámetro. El nombre permite acceder a él desde el código del filtro.

El o los elementos para los que el filtro debe aplicarse se describen en una o varias secciones `<filter-mapping>`. Hay dos opciones para la definición de estas secciones.

La primera alternativa permite asociar un filtro a un recurso únicamente cuando éste es llamado por una petición HTTP procedente del exterior del servidor (desde un navegador). El elemento `<filter-name>` retoma el nombre del filtro y los elementos `<url-pattern>` o `<servlet-name>` definen para qué recurso tiene que estar activo el filtro.

La segunda alternativa permite configurar un filtro para que se ejecute cuando un recurso sea llamado incluso si la llamada proviene de la misma aplicación gracias al objeto `RequestDispatcher`. La sintaxis de declaración es idéntica a la primera solución pero además incluye uno o varios elementos `<Dispatcher>` que indican para qué tipos de acceso al recurso tiene que estar activo el filtro.

Se pueden usar los siguientes valores:

`REQUEST`: el filtro se activa cuando la petición HTTP proviene directamente del cliente.

`FORWARD`: el filtro se activa cuando el recurso se invoca desde el método `forward` de un objeto `RequestDispatcher`.

`INCLUDE`: el filtro se activa cuando el recurso se invoca desde el método `include` de un objeto `RequestDispatcher`.

`ERROR`: el filtro se activa cuando el mecanismo de gestión de errores entra en acción seguido al lanzamiento de una excepción o de un error HTTP.

El ejemplo siguiente implementa un filtro para realizar estadísticas sobre los tipos de navegador que acceden a la aplicación.

Código del filtro

```
package es.eni.ri;

import java.io.*;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
```

```

public class FiltroNavegador implements Filter
{
    FileOutputStream archivo=null;
    PrintWriter outArchivo=null;
    String idIE=null;
    String idFF=null;
    public void init(FilterConfig filterConfig) throws ServletException
    {
        try
        {
            archivo = new
FileOutputStream(filterConfig.getInitParameter("nombreArchivoLog"));
            outArchivo=new PrintWriter(archivo);
            idIE=filterConfig.getInitParameter("idIE");
            idFF=filterConfig.getInitParameter("idFF");
        } catch (FileNotFoundException ex)
        {
            Logger.getLogger(FiltroNavegador.class.getName()).log(Level.SEVERE
E, null, ex);
        }
    }

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException
    {
        HttpServletRequest peticion;
        String navegador;
        String mensaje;
        peticion=(HttpServletRequest)request;
        navegador=requete.getHeader("user-agent");
        if(navegador.toUpperCase().contains(idIE.toUpperCase()))
        {
            mensaje="Internet Explorer" + new
GregorianCalendar().getTime().toString();
            outArchivo.println(mensaje);

        }
        else if (navegador.toUpperCase().contains(idFF.toUpperCase()))
        {
            mensaje="FireFox" + new GregorianCalendar().getTime().toString();
            outArchivo.println(mensaje);
        }
        else
        {
            mensaje="navegador desconocido" + new
GregorianCalendar().getTime().toString();
            outArchivo.println(mensaje);
        }
        outArchivo.flush();
        chain.doFilter(request, response);
    }

    public void destroy()
    {
        try {
            outArchivo.close();
            archivo.close();
        } catch (IOException ex) {
            Logger.getLogger(FiltroNavegador.class.getName()).log(Level.SEVERE
E, null, ex);
        }
    }
}

```

Declaración del filtro en el descriptor de despliegue

```
<filter>
```

```
<description>ghghjg</description>
<filter-name>estadisticas del navegador</filter-name>
<filter-class>es.eni.ri.FiltroNavegador</filter-class>
<init-param>
    <description>nombre del archivo de registro
de estadísticas</description>
    <param-name>nombreArchivoLog</param-name>
    <param-value>c:\logs\estadoNavegador.log</param-value>
</init-param>
<init-param>
    <description>identificador de un cliente Internet
Explorer</description>
    <param-name>idIE</param-name>
    <param-value>MSIE</param-value>
</init-param>
<init-param><description>identificador de un cliente
FireFox</description>
    <param-name>idFF</param-name>
    <param-value>FIREFOX</param-value>
</init-param>
</filter>
<filter-mapping>
    <filter-name>estadisticas del navegador</filter-name>
    <url-pattern>/RendimientoInversion</url-pattern>
</filter-mapping>
```

Eventos

En el transcurso del funcionamiento de una aplicación en el servidor, éste instancia varios objetos en el inicio de la aplicación y los destruye cuando ésta se detiene. Los eventos nos permiten estar informados de estas operaciones y permiten ejecutar acciones que tienen asociadas. Estas acciones tienen que definirse en una o varias clases que implementan las interfaces asociadas a cada tipo de evento. El principio de funcionamiento es muy similar al usado para el tratamiento de eventos en una aplicación swing. La única pequeña diferencia aparece a nivel de la asociación entre estas clases y el contexto de aplicación, ya que se tiene que hacer en el descriptor de despliegue de la aplicación. Esta declaración se realiza añadiendo uno o varios elementos <listener> que contienen el nombre de la clase encargada de gestionar los eventos.

```
<listener>
  <listener-class>.....</listener-class>
</listener>
```

1. Eventos asociados a la aplicación

Este tipo de eventos se desencadenan a través del objeto `ServletContext` que está asociado a la aplicación. En la inicialización del contexto y en su destrucción es cuando se activan. La manipulación de atributos de contexto de aplicación también desencadena eventos en las siguientes situaciones:

- cuando se añade un atributo;
- cuando se modifica un atributo;
- cuando se elimina un atributo.

La gestión de estos eventos debe delegarse a una clase que implemente la interfaz `ServletContextListener` para los eventos asociados al ciclo de vida del contexto y la interfaz `ServletContextAttributeListener` para los eventos asociados a la manipulación de atributos de contexto de aplicación. Para poder administrar estos dos tipos de eventos, la clase tiene que contener los métodos siguientes:

- `void contextInitialized(ServletContextEvent sce)` que se invoca cuando el servidor crea el contexto.
- `void contextDestroyed(ServletContextEvent sce)` que se invoca cuando el servidor destruye el contexto.
- `void attributeAdded(ServletContextAttributeEvent scab)` que se invoca cuando se añade un atributo al contexto de la aplicación.
- `void attributeReplaced(ServletContextAttributeEvent scab)` que se invoca cuando se modifica un atributo en el contexto de la aplicación.
- `void attributeRemoved(ServletContextAttributeEvent scab)` que se invoca cuando se borra un atributo del contexto de la aplicación.

Para los eventos asociados al ciclo de vida de la aplicación, el parámetro de tipo `ServletContextEvent` permite obtener, gracias al método `getServletContext()`, una referencia al contexto de la aplicación que ha disparado el evento.

En los eventos asociados al uso de atributos el parámetro de tipo `ServletContextAttributeEvent` proporciona, gracias a los métodos, `String getName()` y `Object getValue()` el nombre y el valor del atributo que ha desencadenado el evento.

El ejemplo siguiente ilustra estos conceptos guardando en un archivo un registro de los distintos eventos.

Código de la clase de gestión de eventos

```
package es.eni.ri;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
```

```

import java.io.PrintWriter;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class EvtContexto implements
ServletContextListener,ServletContextAttributeListener
{
    FileOutputStream archivo=null;
    PrintWriter outArchivo=null;
    public void contextInitialized(ServletContextEvent sce)
    {
        try {
            archivo = new
FileOutputStream(sce.getServletContext().getInitParameter("nombreArchivoLog"));
            outArchivo=new PrintWriter(archivo);
            outArchivo.println("*****");
            outArchivo.println("*****");
            outArchivo.println(new
GregorianCalendar().getTime().toString() + " arranque de la aplicación ");
            outArchivo.flush();
        } catch (FileNotFoundException ex) {
            Logger.getLogger(EvtContexte.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    public void contextDestroyed(ServletContextEvent sce)
    {
        try {
            outArchivo.println("*****");
            outArchivo.println("*****");
            outArchivo.println(new
GregorianCalendar().getTime().toString() + " parada de la aplicación ");
            outArchivo.flush();
            outArchivo.close();
            archivo.close();
        } catch (IOException ex) {
            Logger.getLogger(EvtContexte.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    public void attributeAdded(ServletContextAttributeEvent scab)
    {
        outArchivo.println("*****");
        outArchivo.println(new
GregorianCalendar().getTime().toString() + " adición de un atributo");
        outArchivo.println("nombre del atributo:" + scab.getName());
        outArchivo.println("valor del atributo:" + scab.getValue());
        outArchivo.flush();
    }

    public void attributeRemoved(ServletContextAttributeEvent scab)
    {
        outArchivo.println("*****");
        outArchivo.println(new
GregorianCalendar().getTime().toString() + " eliminación de un atributo");
        outArchivo.println("nombre del atributo:" + scab.getName());
        outArchivo.flush();
    }

    public void attributeReplaced(ServletContextAttributeEvent scab)

```

```

    {
        outArchivo.println("*****");
        outArchivo.println(new
GregorianCalendar().getTime().toString() + " modificación de un atributo");
        outArchivo.println("nombre del atributo:" + scab.getName());
        outArchivo.println("valor del atributo:" + scab.getValue());
        outArchivo.flush();
    }
}

```

Declaración del listener en el descriptor de despliegue

```

<listener>
    <description>tratamiento de los eventos del contexto
de aplicación</description>
    <listener-class>es.eni.ri.EvtContexto</listener-class>
</listener>

```

2. Eventos asociados a sesiones

El uso de sesiones en una aplicación también provoca el desencadenamiento de eventos en el interior de la aplicación. Las situaciones en las que se desencadenan eventos asociados a sesiones son bastante similares a los ya vistos para una aplicación. El principio de gestión es sin duda idéntico. Las interfaces que deben implementar la clase encargada de gestionar los eventos asociados son la interfaz `HttpSessionListener` para los eventos de creación y destrucción de sesiones y la interfaz `HttpSessionAttributeListener` para los eventos asociados a la manipulación de elementos en el interior de la sesión. Por lo tanto, los métodos siguientes tienen que estar disponibles en esta clase.

- `void sessionCreated(HttpSessionEvent se)`: este método se ejecuta cuando se crea una sesión.
- `void sessionDestroyed(HttpSessionEvent se)`: este método se ejecuta cuando se destruye una sesión. La destrucción puede tener su origen por la llamada al método `invalidate` de la sesión o porque ha vencido el intervalo de expiración.
- `void attributeAdded(HttpSessionBindingEvent se)`: este método se ejecuta cuando se añade un atributo a la sesión.
- `void attributeRemoved(HttpSessionBindingEvent se)`: este método se ejecuta cuando se borra un atributo de la sesión.
- `void attributeReplaced(HttpSessionBindingEvent se)`: este método se ejecuta cuando se modifica un atributo de la sesión.

Para los eventos asociados a la creación y a la destrucción de la sesión, el parámetro de tipo `HttpSessionEvent` permite obtener, gracias al método `getSession()`, una referencia a la sesión que ha desencadenado el evento.

En el caso de los eventos asociados al uso de atributos el parámetro de tipo `HttpSessionBindingEvent` proporciona, gracias a los métodos `String getName()` y `Object getValue()`, el nombre y el valor del atributo que ha desencadenado el evento. También se puede acceder a la sesión que contiene este atributo con el método `getSession()`.

El siguiente ejemplo de código permite guardar en un archivo de registro los eventos asociados a la sesión.

Código de la clase que gestiona los eventos

```

package es.eni.ri;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.GregorianCalendar;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;

```

```

import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class EvtSesion implements HttpSessionAttributeListener,
HttpSessionListener
{
    FileOutputStream archivo=null;
    PrintWriter outArchivo=null;
    public void attributeAdded(HttpSessionBindingEvent se)
    {
        try
        {
            archivo = new
OutputStream(se.getSession().getServletContext().getInitParamete
ter("nombreArchivoLogSesiones"), true);
            outArchivo = new PrintWriter(archivo);
            outArchivo.print("adición de un atributo de tipo ");
            outArchivo.print(se.getValue().getClass().getName());
            outArchivo.println(" a la sesión " +
se.getSession().getId());
            outArchivo.flush();
            outArchivo.close();
            archivo.close();
        }
        catch (IOException ex)
        {
            Logger.getLogger(EvtSession.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    public void attributeRemoved(HttpSessionBindingEvent se)
    {
        try
        {
            archivo = new
OutputStream(se.getSession().getServletContext().getInitParamete
ter("nombreArchivoLogSesiones"), true);
            outArchivo = new PrintWriter(archivo);
            outArchivo.print("eliminación de un atributo de tipo ");
            outArchivo.print(se.getValue().getClass().getName());
            outArchivo.println(" de la sesión " + se.getSession().getId());
            outArchivo.flush();
            outArchivo.close();
            archivo.close();
        }
        catch (IOException ex)
        {
            Logger.getLogger(EvtSession.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

    public void attributeReplaced(HttpSessionBindingEvent se)
    {
        try
        {
            archivo = new
OutputStream(se.getSession().getServletContext().getInitParamete
ter("nombreArchivoLogSesiones"), true);
            outArchivo = new PrintWriter(archivo);
            outArchivo.println("*****");
            outArchivo.print("modificación de un atributo de tipo ");
            outArchivo.print(se.getValue().getClass().getName());
            outArchivo.println(" de la sesión " + se.getSession().getId());
            outArchivo.flush();
            outArchivo.close();
        }
    }
}

```

```

        archivo.close();
    }
}
catch (IOException ex)
{
    Logger.getLogger(EvtSession.class.getName()).log(Level.SEVERE,
null, ex);
}
}

public void sessionCreated(HttpSessionEvent se)
{
    try
    {
        se.getSession().setAttribute("arranque", new GregorianCalendar());
        archivo = new
FileOutputStream(se.getSession().getServletContext().getInitParamete
ter("nombreArchivoLogSesiones"), true);
        outArchivo = new PrintWriter(archivo);
        outArchivo.println("*****");
*****");
        outArchivo.println(new
GregorianCalendar().getTime().toString() + " inicio de la sesión
" + se.getSession().getId());
        outArchivo.flush();
        outArchivo.close();
        archivo.close();
    }
    catch (IOException ex)
    {
        Logger.getLogger(EvtSession.class.getName()).log(Level.SEVERE,
null, ex);
    }
}

public void sessionDestroyed(HttpSessionEvent se)
{
    GregorianCalendar arranque=null;
    arranque=(GregorianCalendar)se.getSession().getAttribute("arranque");
    long duracion;
    duracion=new GregorianCalendar().getTimeInMillis()-
arranque.getTimeInMillis();
    outArchivo.close.println("*****");
*****");
    System.out.println("duración: " + duracion);

    try
    {
        archivo = new
FileOutputStream(se.getSession().getServletContext().getInitParamete
ter("nombreArchivoLogSesiones"),true);
        outArchivo=new PrintWriter(archivo);
        outArchivo.println("*****");
*****");
        outArchivo.print(new
GregorianCalendar().getTime().toString() + " duración de la sesión
" + se.getSession().getId() + ": ");
        outArchivo.println(duracion/1000 + " Segundos");
        outArchivo.flush();
        outArchivo.close();
        archivo.close();
    }
    catch (IOException ex)
    {
        Logger.getLogger(EvtSession.class.getName()).log(Level.SEVERE,
null, ex);
    }
}
}

```

}

Declaración del listener en el descriptor de despliegue

```
<listener>
  <listener-class>es.eni.ri.EvtSession</listener-class>
</listener>
```

Sincronización de servlets

Cuando el servidor recibe una petición HTTP, la analiza y determina si involucra a un servlet en su tratamiento. Si éste es el caso, el servidor busca una instancia del servlet en cuestión. Si no hay ninguna instancia disponible, hay que crear una nueva. A continuación el servidor arranca un nuevo thread para poder delegar al servlet el tratamiento de la petición HTTP. Este thread ejecuta principalmente el método `service` del servlet y todos los métodos que le siguen.

Si hay varias peticiones simultáneas, el servidor crea varios threads para tratar en paralelo estas distintas peticiones. Un mismo método de la instancia de un servlet puede por lo tanto ejecutarse simultáneamente por varios threads. El contenido de los atributos del servlet puede por consiguiente modificarse por cada thread independientemente de los demás, lo que provoca riesgos de perturbar el correcto funcionamiento global de la aplicación.

La solución a este problema consiste en asegurar que el código del servlet no se podrá ejecutar por más de un thread a la vez.

1. Utilización de la interfaz SingleThreadModel

La primera solución se obtiene implementando en el servlet la interfaz `SingleThreadModel`. La implementación de esta interfaz no exige la creación de métodos específicos. Se usa para marcar la clase para definir el comportamiento del servidor en el momento en que usará la clase. Dependiendo del servidor, esta interfaz puede traducirse en dos implementaciones distintas, con el objetivo de garantizar que el código del servlet no se ejecute por más de un solo thread a la vez.

- El servidor puede crear una sola instancia de la clase correspondiente al servlet y gestionar una cola de espera para los threads encargados de tratar las peticiones HTTP. Cada uno de estos threads de la cola de espera recibe la autorización de usar el servlet en función de su llegada a la cola de espera.
- El servidor también puede utilizar otra estrategia creando una instancia de la clase correspondiente al servlet para cada petición HTTP. Estas instancias a continuación se almacenan en un pool para una posible reutilización.

Ambas soluciones conllevan sin embargo varios inconvenientes:

Si se usa una única instancia, el tratamiento de las peticiones será mucho más lento y los clientes rápidamente percibirán una falta de actividad por parte del servidor.

Por contra, si una instancia de servlet se usa para cada petición el tratamiento será rápido, pero lamentablemente si muchas peticiones llegan simultáneamente el consumo de recursos de memoria en el servidor se verá fuertemente aumentado.

Además, ambas soluciones son ineficaces si el servlet usa otras clases en los tratamientos.

Por todas estas razones, SUN recomienda no utilizar esta solución y reemplazarla por la creación de bloques de código sincronizados.

2. Utilización de bloques de código sincronizados

La utilización de bloques de código sincronizados permite gestionar de forma más eficaz la sincronización de algunas porciones de código utilizadas en un servlet.

Para que los bloques de código se beneficien de un candado para garantizar que no se ejecutarán por más de un único thread a la vez tienen que definirse con el uso de la palabra clave `synchronized`. Se recomienda limitar al mínimo estricto el volumen de código colocado en estos bloques.

```
Print Writer out;
out = resp.getWriter();
synchronized (this)
{ contador++;
}
out.println("valor del contador:"+contador);
```

Obtener el seguimiento de la sesión

Muchas aplicaciones necesitan que las peticiones HTTP de un mismo cliente puedan asociarse las unas a las otras. El ejemplo más clásico es el de las aplicaciones de compra en línea. La búsqueda de artículos, su selección y la colocación de los artículos en un carrito virtual hasta la validación del pedido requieren muchas peticiones HTTP enviadas desde un mismo cliente. Estas peticiones deberían poderse asociar unas a otras. El protocolo HTTP es un protocolo sin estado (cada petición es absolutamente independiente de las otras), con lo que es la aplicación quien tiene la obligación de mantener un estado. Hay dos soluciones posibles para guardar los datos propios de cada cliente. La primera solución consiste en confiar al cliente (navegador) la responsabilidad de guardar los datos necesarios para el buen funcionamiento de la aplicación. Para ello, se usa el mecanismo de las cookies.

Con la segunda opción, el servidor tiene que encargarse él mismo de guardar los datos asociados a cada cliente. Las sesiones definen un espacio de almacenamiento en el servidor en el que se guardan los datos de cada cliente.

1. Uso de cookies

Una cookie representa un dato enviado por el servidor con destino al navegador. El servidor inserta esta información en la cabecera de la respuesta HTTP. Cuando el navegador analiza esta respuesta HTTP, extrae la o las cookies y las almacena en el ordenador cliente. Este almacenamiento puede realizarse en disco o en memoria en función de las características asignadas a la cookie en el momento de su creación por el servidor. Cada navegador posee su propia técnica para administrar las cookies que el servidor le ha confiado. Generalmente, el navegador almacena las cookies en un archivo de texto que asocia a una determinada aplicación.

Si el navegador realiza una nueva petición HTTP con destino en una aplicación de la cual ya tiene cookies, éste las inserta automáticamente en la cabecera de la petición HTTP. La aplicación puede entonces, analizando la petición HTTP, volver a tener los datos que anteriormente había confiado al cliente.

Sin embargo, las cookies presentan un pequeño inconveniente debido a que los navegadores pueden configurarse para rechazar las cookies enviadas por los servidores. Por lo tanto, en algunos casos es importante prevenir este problema con otra solución para asegurarse de que hay seguimiento de la información asociada al cliente.

a. Creación y envío de cookies

La primera fase en el uso de una cookie consiste en crearla y después determinar sus características antes de incluirla en una respuesta HTTP. La clase `javax.servlet.http.Cookie` permite representar una cookie en forma de objeto Java. Las características de la cookie vienen determinadas por las diferentes propiedades de este objeto. Para la manipulación de estas características disponemos de los siguientes métodos.

`Cookie(String name, String value)`: este constructor permite determinar, en el momento de la creación, el nombre asociado a la cookie y el valor memorizado en la cookie. Estos datos se forman con dos cadenas de caracteres. El nombre dado a la cookie tiene que respetar algunas reglas bastante estrictas ya que sólo puede contener caracteres alfanuméricos (sin signos de puntuación, espacios, carácter \$...). Este nombre no puede modificarse después de la creación de la cookie. Por supuesto es gracias a éste que la cookie podrá ser identificada cuando el navegador la reenvíe al servidor. Las reglas acerca del valor de la cookie son más flexibles ya que la única limitación hace referencia al tamaño de la cookie que se limita a cuatro kilobytes.

Si un objeto Java tiene que transferirse en una cookie conviene transformarlo previamente en cadena de caracteres antes de almacenarlo en una cookie.

`void setDomain(String pattern)`: este método permite asociar un nombre de dominio a la cookie. Este método espera como parámetro una cadena de caracteres que comience con el carácter . (punto). Por defecto, desde que se crea, la cookie se asocia al dominio al que pertenece el servidor que la acaba de crear. El navegador utiliza esta información para clasificar las cookies y determinar las que tiene que poner en una petición HTTP destinada a un servidor en concreto.

`void setPath(String uri)`: este método permite establecer que únicamente la aplicación (cuyo nombre se le pasa por parámetro) es del ámbito de la cookie. El navegador sólo incorpora en la petición HTTP las cookies asociadas a la aplicación a la que la petición HTTP va dirigida. Por defecto la cookie se asocia a la aplicación en el origen de su envío al navegador.

`void setMaxAge(int expiry)`: este método permite fijar el tiempo de vida de la cookie (en segundos). El navegador tiene que conservarla durante este tiempo y reenviarla en todas las peticiones HTTP destinadas al servidor o a la aplicación que se la han transmitido. Hay dos casos específicos:

La llamada a este método pasando como parámetro el valor -1 se usa para que el navegador no guarde de forma permanente la cookie. Ésta permanece simplemente en memoria hasta el cierre del navegador.

Este método también puede llamarse pasando como parámetro el valor 0 para que el navegador pueda eliminar la cookie.

`void setSecure(boolean flag)`: llamando a este método y pasando como parámetro el valor `true`, se indica al

navegador que no debe incluir la cookie en las futuras peticiones si éstas no se envían con el protocolo HTTPS.

`void setValue(String newValue):` este método permite modificar el valor contenido en una cookie.

Después de la creación de la cookie, ésta puede enviarse al navegador. Para ello, hay que incorporarla en la respuesta HTTP. El método `void addCookie(Cookie cookie)` asocia a la respuesta HTTP la cookie que se pasa por parámetro. Este método tiene que usarse varias veces si múltiples cookies tienen que enviarse al cliente.

b. Recuperación y uso de cookies

Cuando el navegador genera una petición HTTP con destino en una aplicación, añade automáticamente a la petición las cookies que la aplicación anteriormente le ha transferido. La aplicación tiene que extraer las cookies de la petición HTTP y explotar su contenido.

El acceso a las cookies de una petición HTTP es menos fácil que el acceso a otros datos de la petición HTTP. En efecto, no se puede extraer de la petición una cookie en concreto. La única solución disponible consiste en obtener en forma de tabla el conjunto de las cookies de la petición mediante el método `Cookie[] getCookies()`. Este método devuelve una tabla que contiene todas las cookies presentes en la petición HTTP o `null` si no se ha encontrado ninguna cookie.

Las cookies presentes en la tabla tienen que tratarse una a una. Los métodos `String getName()` y `String getValue()` permiten obtener los datos propios de cada cookie.

El siguiente ejemplo ilustra el uso de las cookies. Este extracto de aplicación guarda en una cookie los datos de conexión del usuario.

Página JSP de conexión

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <%
        String login="";
        String password="";
        String infos=null;
        Cookie[] gatos;
        // recuperación de las cookies presentes en la petición http
        gatos=request.getCookies();
        // verificación de si había cookies
        if (gatos!=null)
        {
            for(int i=0;i<gatos.length;i++)
            {
                // búsqueda de una cookie login
                if(gatos[i].getName().equals("login"))
                {
                    // extracción de los datos almacenados en la cookie
                    infos=gatos[i].getValue();
                    login=infos.split("-")[0];
                    password=infos.split("-")[1];
                }
            }
        }
    %>
    <body>
        <h1>Identifíquese</h1>
        <form method="post" action="Autenticacion">
            <table>
                <tbody>
                    <tr>
                        <td>usuario</td>
                        <!-- campo de introducción del nombre de usuario -->
                        <!-- el contenido se inicializa con el
```

```

valor almacenado en la cookie-->
    <td><input type="text" name="login"
value=<%=login%>" size="20" /></td>
    </tr>
    <tr>
        <td>contraseña</td>
        <!-- campo de introducción de la contraseña del
usuario -->
        <!-- el contenido se inicializa con el
valor almacenado en la cookie-->
        <td><input type="password" name="password"
value=<%=password%>" size="20" /></td>
    </tr>
    <tr>
        <td>
            <input type="checkbox" name="memo"
value="memo">Guardar sus datos de conexión</input>
        </td>
    </tr>
    <tr>
        <td></td>
        <td> <input type="submit" value="validar"
name="validacion" /></td>
    </tr>
</tbody>
</table>
<%
    String message;
    message=(String)request.getAttribute("message");
    if (message!=null)
    {
        out.println(message);
    }
%>
</form>
</body>
</html>

```

Servlet de validación de las conexiones

```

package fr.eni.ri;

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

public class Autenticacion extends HttpServlet
{
    String login;
    String password;
    String memo;
    Cookie gato;
    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException
    {
        RequestDispatcher rd=null;
        login=request.getParameter("login");
        password=request.getParameter("password");
        memo=request.getParameter("memo");
        if (Login.validacion(login, password))
        {
            // verificación que permite saber si el usuario
desea guardar

```

```

        // sus datos de conexión
        if (memo==null)
        {
            // creación de una cookie para poder eliminar una
possible cookie
            // ya presente en el navegador
            gato=new Cookie("login", "");
            gato.setMaxAge(0);
        }
        else if( memo.equals("memo") )
        {
            // creación de una cookie que permite guardar
los datos de conexión
            // en el navegador
            gato=new Cookie("login", login + "-" + password);
            // el navegador conservará los datos durante siete días
            gato.setMaxAge(24*3600*7);
        }
        response.addCookie(gato);
        rd=getServletContext().getRequestDispatcher("/continuar.jsp");
        rd.forward(request, response);
    }
    else
    {   request.setAttribute("message", "datos de conexión inválidos");
        rd=getServletContext().getRequestDispatcher("/inicio.jsp");
        rd.forward(request, response);
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}

```

2. Utilización de la sesión

La sesión representa un espacio de almacenamiento que el servidor asocia a cada cliente. Para poder realizar el enlace entre un cliente en concreto y el espacio de almacenamiento que se le dedicará en el servidor, éste tiene que poder identificar de forma segura el origen de la petición HTTP. Los únicos datos disponibles para realizar esta identificación tienen que encontrarse en la petición HTTP. Una primera idea podría ser la de usar el nombre de host del cliente o su dirección IP para realizar esta identificación. Desgraciadamente, esta solución no es realista ya que muchos clientes utilizan los servicios de un servidor proxy para acceder a una aplicación web. En este caso hipotético, las peticiones HTTP generadas por los clientes se transmiten al servidor proxy que proporciona el enrutamiento hasta el servidor de aplicaciones. Por lo tanto, es la identidad del servidor proxy la que es accesible desde los datos almacenados en la petición HTTP recibida por el servidor de aplicaciones.

La solución consiste en que el servidor de aplicaciones genere una clave que le permita establecer el enlace entre un cliente y un espacio de almacenamiento que le esté asociado en el servidor. A esta clave se la llama identificador de sesión. Cada petición HTTP tiene que tener este identificador de sesión para poder estar asociada a un cliente en concreto. El problema a resolver es, por lo tanto, la transferencia de este identificador de sesión al cliente para que éste pueda a continuación incluirlo en cada petición HTTP emitida con destino al servidor. Hay tres posibles soluciones:

- La reescritura de la URL.
- Los campos ocultos de formulario.

- Las cookies.

a. Reescritura de la URL

La reescritura de la URL consiste en que el servidor añada en todas las URL presentes en la respuesta HTTP transmitida al cliente un parámetro correspondiente al identificador de sesión. Este mecanismo no es muy complejo de implementar gracias a los métodos `String encodeURL(String url)` y `String encodeRedirectURL(String url)` disponibles en el objeto `HttpServletResponse`. Estos dos métodos esperan como parámetro una cadena de caracteres que representa la URL a transformar a la que éstos añadirán un parámetro correspondiente al identificador de sesión. Además, contienen un mecanismo que permite verificar automáticamente si no hay otra solución que se pueda usar (las cookies) para transmitir el identificador de sesión al cliente. Si detectan que hay otra solución disponible para asegurar esta transferencia, dejan sin modificar la cadena de caracteres que se les ha pasado por parámetro. Esta solución obliga a estar alerta en el desarrollo sin olvidar el uso de uno de los dos métodos para cada URL transmitida al cliente en la respuesta HTTP. Los formularios HTML, los enlaces de hipertexto y las respuestas de redirección son los principales elementos involucrados. Además, para usar este mecanismo se requiere que todas las respuestas HTTP transmitidas al cliente estén generadas por el servidor (servlet o página JSP). El uso de un documento HTML estático inhabilita el seguimiento de sesión con esta solución. Los ejemplos mostrados a continuación ilustran los distintos casos hipotéticos.

- Caso de un formulario HTML generado por un servlet.

```
out.println("<FORM action=\"" +  
response.encodeURL("/riJEE/Seguir") + "\">");
```

el código HTML construido por esta instrucción

```
<FORM  
action="/riJEE/Seguir;jsessionid=43A9FBA967BBA4B68AE3114B7E4745CA">
```

- Caso de un formulario contenido en una página JSP.

```
<form action="<% = response.encodeURL("/riJEE/Seguir") %>">
```

el código HTML recibido después del tratamiento de la página JSP por el servidor

```
<form  
action="/riJEE/Seguir;jsessionid=AAD4B8EB4B069D13F543FBC7C0BC8C1E">
```

b. Campos de formulario ocultos

El uso de los campos de formulario ocultos es un poco más limitado debido a que el servidor únicamente puede ser contactado con una petición HTTP generada por la validación del formulario. Este formulario tiene que contener un campo de texto invisible para el usuario y que contenga el identificador de sesión. Este campo de texto tiene que llamarse `jsessionid` y su contenido tiene que generarse dinámicamente por la recuperación del identificador de sesión con el método `getId()` del objeto `Session`. Los ejemplos de código siguientes presentan esta solución tanto para un servlet como para una página JSP.

- Caso de un formulario generado por un servlet.

```
out.println("<input type=\"hidden\" name=\"jsessionid\" value=\"" +  
request.getSession().getId() + "\">");
```

el código HTML generado por esta instrucción

```
<input type="hidden" name="jsessionid"  
value="ABC7A4851D93D92C7959EF88A85D36E4">
```

- Caso de un formulario HTML de una página JSP.

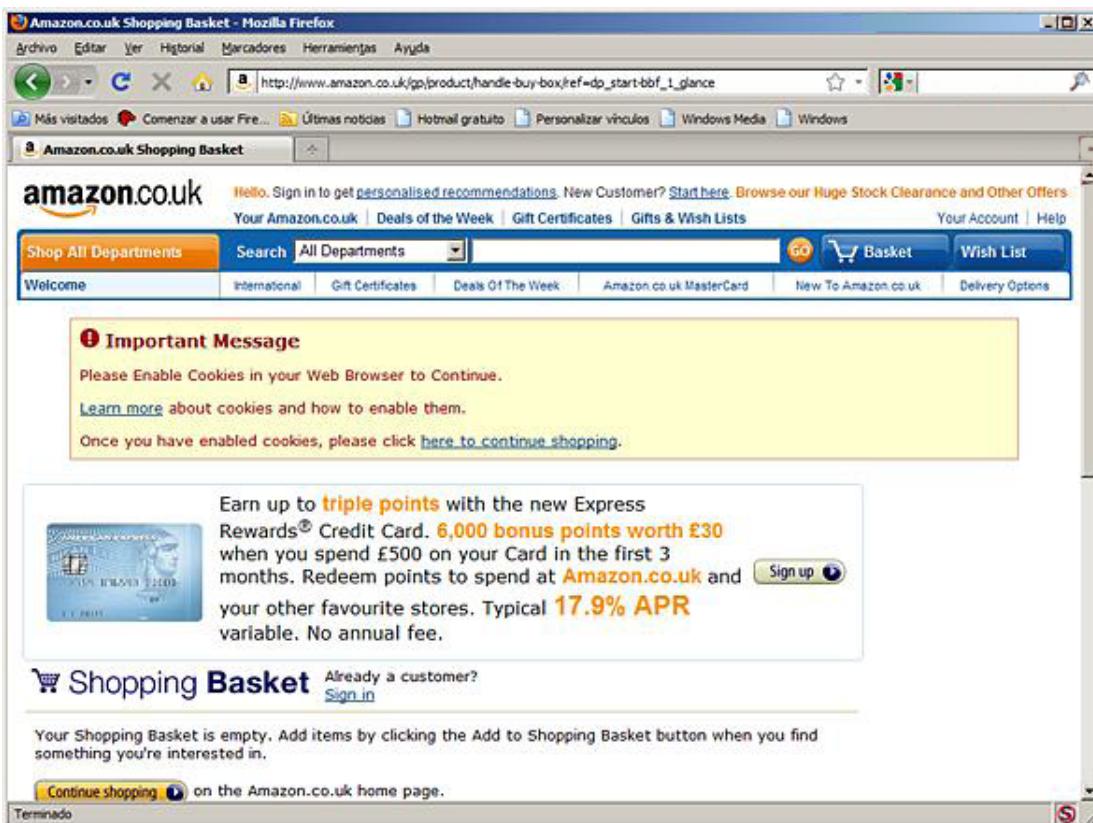
```
<input type="text" name="jsessionid" value="<% = session.getId() %>">
```

el código HTML recibido después del tratamiento de la página JSP por el servidor

```
<input type="hidden" name="jsessionid"
value="ABC7A4851D93D92C7959EF88A85D36E4">
```

c. Cookies

La última solución utiliza las cookies para transportar el identificador de sesión entre el cliente y el servidor. Esta solución es sin duda la más sencilla de implementar debido a que no hay que hacer absolutamente nada. El servidor y el navegador se encargan de todo automáticamente. Cuando el servidor crea una sesión para un usuario, genera automáticamente una cookie en la que guarda el identificador de sesión. Esta cookie se añade automáticamente a la respuesta HTTP transmitida al cliente. Todas las nuevas peticiones con destino esta aplicación y que procedan de este cliente contendrán la cookie de sesión. El único fallo de este mecanismo se encuentra a nivel del cliente que puede haber prohibido las cookies en su navegador. El seguimiento de sesión en este caso es imposible si las cookies son la única solución implementada para transportar la cookie de sesión. Cuando esto sucede, es necesario detectar esta situación y advertir al cliente.



Lo ideal, por supuesto, es tener de antemano una alternativa en el caso de que las cookies estén desactivadas en el navegador cliente.

Utilización de la sesión

Es muy sencillo implementar la manipulación de sesiones debido a que la mayor parte de las operaciones están automatizadas al máximo. El trabajo más importante consiste en buscar la sesión desde el identificador de sesión. Este identificador se transmite en la petición HTTP (ya sea en forma de cookie o como parámetro), por lo que es lógico que sea el objeto `HttpServletRequest` el que proponga un método que permita obtener una sesión.

1. Obtener una sesión

El objeto `HttpServletRequest` proporciona los dos métodos siguientes que permiten obtener una sesión. Esta sesión se representa con un objeto que implementa la interfaz `HttpSession`.

`HttpSession getSession():` la llamada a este método devuelve la sesión asociada a la petición HTTP o una nueva sesión si no hay ningún identificador en la petición HTTP. El identificador de sesión se busca en las cookies de la petición y después en los parámetros de la petición HTTP.

`HttpSession getSession(boolean create):` este método aporta una sutil diferencia en relación a la versión anterior. El booleano esperado como parámetro modifica el comportamiento de este método si no hay identificador de sesión en la petición HTTP. En el caso de que este booleano tenga el valor `true`, el comportamiento es idéntico al de la versión anterior. Si el booleano tiene el valor `false`, no se crea la nueva sesión y este método devuelve `null`.

Se puede verificar después de la llamada a este método si la sesión se acaba de crear o se trata de una sesión ya utilizada. El método `boolean isNew()` devuelve un booleano igual a `true` si la sesión acaba de crearse y `false` en caso contrario.

2. Almacenar, consultar y suprimir elementos

Los métodos que permiten la manipulación de elementos en el interior de una sesión son muy parecidos a los utilizados para la manipulación de los atributos de una petición HTTP. La inserción o la modificación de un elemento se realiza con el método `void setAttribute(String name, Object value)`. El primer parámetro representa el nombre utilizado para identificar el atributo en la sesión. El segundo parámetro representa el objeto que se desea almacenar en la sesión. Contrariamente a lo que sucede con las cookies, se puede poner un objeto en la sesión en vez de una simple cadena de caracteres. Si el nombre del parámetro ya está en uso en esta sesión, el valor correspondiente se sobreescribe por el nuevo.

La consulta de un elemento de la sesión se realiza con el método `Object getAttribute(String name)`. Este método devuelve el objeto previamente almacenado en la sesión con el nombre indicado o `null` si no hay ningún elemento con este nombre en la sesión. Muy frecuentemente, al objeto obtenido de la sesión hay que hacerle una conversión de tipo antes de usarlo.

El método `void removeAttribute(String name)` permite eliminar de la sesión el elemento cuyo nombre se pasa como parámetro. Si no hay ningún elemento en la sesión con este nombre, no realiza ninguna acción.

El método `EnumerationgetAttributeNames()` permite obtener en forma de enumeración la lista de todos los nombres de los elementos de la sesión.

Cuando es necesario obtener el seguimiento de sesión por otro método distinto que el del uso de cookies es indispensable poder obtener el identificador de sesión generado por el servidor. El método `String getId()` lo devuelve en forma de cadena de caracteres generada por el servidor.

3. Finalizar la sesión

El uso de sesiones consume recursos de memoria en el servidor. No es posible que el servidor conserve indefinidamente las sesiones de todos los clientes que se han conectado. Hay dos soluciones posibles para liberar recursos de memoria asociados a las sesiones.

La aplicación puede finalizar voluntariamente una sesión llamando al método `void invalidate()` que provoca la destrucción inmediata de la sesión y de todos los elementos que contenía.

Si la aplicación no finaliza voluntariamente la sesión, es el servidor el que lo hace automáticamente después de una cierta duración de inactividad de la sesión. Múltiples métodos permiten controlar el funcionamiento de este mecanismo.

El método `int getMaxInactiveInterval()` permite obtener el tiempo de vida en segundos de las sesiones sin uso en el servidor. Este valor por defecto es configurable en el descriptor de despliegue de la aplicación con la etiqueta siguiente:

```
<session-config>
    <session-timeout>10</session-timeout>
</session-config>
```

-
-  Observe que en esta etiqueta el tiempo de vida de la sesión se expresa en minutos.
-

La duración máxima de inactividad de una sesión puede también configurarse con el método void setMaxInactiveInterval(int interval). En este método el tiempo de vida máximo de inactividad de la sesión se tiene que expresar en segundos y sólo se aplica en esta sesión en concreto. El valor por defecto configurado en el descriptor de despliegue seguirá aplicándose siempre para el resto de sesiones.

Presentación

En una aplicación Web, se usan dos tipos de recursos:

- Los recursos estáticos cuyo contenido no cambia entre dos peticiones HTTP.
- Los recursos dinámicos cuyo contenido se adapta en función de los datos recibidos en la petición HTTP.

Sea cual sea el tipo de recurso, el lenguaje HTML se usa prácticamente siempre para la codificación de estos recursos. Para los recursos estáticos, se genera durante el desarrollo de la aplicación, generalmente gracias a una herramienta especializada en el diseño de este tipo de recursos. Estas herramientas permiten crear rápidamente páginas HTML complejas, muchas veces sin ni siquiera tener la necesidad de escribir manualmente una sola línea de código HTML. A veces, es sorprendente la complejidad del código generado.

Para los recursos dinámicos, este código HTML tiene que generarse completamente con instrucciones Java escribiendo directamente en el cuerpo de la respuesta HTTP. Con esta solución, un servlet puede llegar a tener varias centenas o incluso varios miles de líneas de código únicamente para la generación de la parte estática de la página.

Las páginas JSP aportan una solución eficaz que permite mezclar código HTML para la parte estática de la página y código Java para generar únicamente las partes dinámicas de la página. El código Java se inserta en el interior de la página JSP mediante un conjunto de etiquetas específicas. El ejemplo de código mostrado a continuación utiliza la etiqueta JSP más sencilla posible para insertar una parte de código Java en medio de etiquetas HTML. Este ejemplo tiene que guardarse en un archivo con extensión .jsp.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Visualización de la fecha y hora</title>
  </head>
  <body>
    <h1>Buenos días</h1>
    estamos a <% out.println(new
java.util.GregorianCalendar().getTime().toLocaleString());%>
  </body>
</html>
```

Cuando un navegador envía una petición HTTP para obtener la página JSP, recibe como respuesta el código HTML siguiente.

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

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Visualización de la fecha y hora</title>
  </head>
  <body>
    <h1>Buenos días</h1>
    estamos a 01-may-2010 00:28:30

  </body>
</html>
```

En el código no hay ningún rastro de etiquetas JSP específicas y, por supuesto, la hora se actualiza en cada refresco de la página. Cuando el servidor ha construido la respuesta HTTP con destino el navegador, ha reemplazado las etiquetas que contienen el código Java por el resultado de la ejecución de éste. Como las páginas JSP son una tecnología de script del lado servidor, lógicamente es éste el que ejecuta el código. El navegador sólo se encarga de visualizar el código HTML que el servidor le ha devuelto. Esta técnica no se tiene que confundir con una tecnología script del lado cliente, como JavaScript, ya que en este caso es el navegador el que se encarga de ejecutar el código. A continuación se muestra la página HTML que permite obtener el mismo resultado pero que utiliza script del lado cliente.

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

```

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Visualización de la fecha y hora</title>
    </head>
    <body>
        <h1>Buenos días</h1>
        estamos a
<script type="text/javascript" language="javascript">
    d = new Date();
    document.write(d.toLocaleDateString()+' '+
d.toLocaleTimeString()+'.');
</script>
</body>
</html>

```

De hecho, obtenemos casi el mismo resultado ya que en el caso de la página JSP tenemos la fecha y hora del servidor pues es éste el que ejecuta el código. En el caso de la página HTML que contiene JavaScript tenemos la fecha y hora de la máquina cliente ya que es ésta la que ejecuta el código.

1. Tratamiento de una página JSP por el servidor

En el párrafo anterior, hemos constatado que el servidor simplemente reemplaza las etiquetas específicas JSP por el resultado de su ejecución sin intervenir a nivel de código HTML. De hecho, la técnica utilizada por el servidor es casi la opuesta ya que en una primera fase realiza un tratamiento en todas las etiquetas HTML de la página generando código fuente Java que permite enviar en la respuesta HTTP estas distintas etiquetas (tranquilos: el servidor es más rápido que nosotros escribiendo código Java). Cuando en el transcurso de su análisis encuentra etiquetas específicas JSP, inserta, en el código fuente que genera, el contenido de estas etiquetas. Cuando finaliza este primer tratamiento ya tiene un archivo fuente Java que contiene la definición de una clase. Esta clase es extremadamente parecida a un servlet. A continuación, el servidor provoca la compilación del código fuente para obtener código ejecutable por la máquina virtual Java. Seguidamente, el servidor crea una instancia de esta clase y la ejecuta como un servlet llamando al método `service` de esta clase. La misma instancia se usará de nuevo si una nueva petición HTTP que implique este recurso llega al servidor. Para entender el trabajo realizado por el servidor, vamos a comparar el contenido de la página JSP con el código fuente generado por el servidor.

El directorio donde se encuentra el código generado depende del tipo de servidor y de la configuración de la aplicación. Para poder encontrar el archivo correspondiente en su disco, es práctico realizar una búsqueda con el nombre del archivo. El servidor conserva el nombre del archivo al que añade el sufijo `_jsp` y después termina con la extensión `.java` (es un archivo fuente Java). Por ejemplo, el archivo `index.jsp` se transforma después del tratamiento por el servidor en un archivo `index_jsp.java`. El archivo compilador respeta también este convenio Java y conserva el mismo nombre pero con la extensión `.class`.

Archivo `index.jsp`

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Visualización de la fecha y hora</title>
    </head>
    <body>
        <h1>Buenos días</h1>
        estamos a <% out.println(new
java.util.GregorianCalendar().getTime().toLocaleString());%>
    </body>
</html>

```

Archivo `index_jsp.java`

```

package org.apache.jsp;

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

```

```

import javax.servlet.jsp.*;

public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    private static final JspFactory _jspxFactory =
        JspFactory.getDefaultFactory();

    private static java.util.List _jspx_dependants;

    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.AnnotationProcessor _jsp_annotationprocessor;
    public Object getDependants() {
        return _jspx_dependants;
    }

    public void _jspInit() {
        _el_expressionfactory =
        _jspxFactory.getJspApplicationContext(getServletConfig()).getServletCon
text().getExpressionFactory();
        _jsp_annotationprocessor = (org.apache.AnnotationProcessor)
getServletConfig().getServletContext().getAttribute(org.apache.
AnnotationProcessor.class.getName());
    }

    public void _jspDestroy() {
    }

    public void _jspService(HttpServletRequest request,
    HttpServletResponse response)
        throws java.io.IOException, ServletException {

        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try {
            response.setContentType("text/html;charset=UTF-8");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;

            out.write("\n");
            out.write("\n");
            out.write("\n");
            out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"\n");
            out.write("      \"http://www.w3.org/TR/html4/loose.dtd\">\n");
            out.write("\n");
            out.write("<html>\n");
            out.write("      <head>\n");
            out.write("          <meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\">\n");
            out.write("          <title> Visualización de la fecha y hora </title>\n");
            out.write("      </head>\n");
            out.write("      <body>\n");
            out.write("          <h1>Buenos días</h1>\n");
            out.write("          estamos a ");
            out.println(new

```

```
java.util.GregorianCalendar().getTime().toLocaleString());
        out.write("\n");
        out.write("      </body>\n");
        out.write("</html>\n");
    } catch (Throwable t) {
        if (!(t instanceof SkipPageException)){
            out = _jspx_out;
            if (out != null && out.getBufferSize() != 0)
                try { out.clearBuffer(); } catch (java.io.IOException e) {}
            if (_jspx_page_context != null)
                _jspx_page_context.handlePageException(t);
        }
    } finally {
        _jspxFactory.releasePageContext(_jspx_page_context);
    }
}
```

Se pueden encontrar en este archivo todas las etiquetas HTML de la página JSP simplemente pasadas como parámetro al método `write` de un objeto `JspWriter`. El código Java de la página JSP está directamente incorporado en el archivo fuente. Por contra, muchas otras líneas se han añadido para el funcionamiento del servlet generado.

Será necesario recordar este ejemplo cuando realice pruebas de una página JSP ya que si se produjera un error durante la ejecución de la página JSP el número de la línea en la que el error se ha producido corresponde al archivo Java y no al archivo JSP.

2. Elementos que constituyen una página JSP

El elemento principal que forma una página JSP es, por supuesto, código HTML. Este código se conserva idéntico en el tratamiento de la página JSP. También sucede lo mismo con el código Java: se puede añadir en la página y se conservará idéntico. Los otros elementos disponibles son las directivas JSP y las etiquetas específicas que se transformarán en código Java en el tratamiento de la página JSP realizado por el servidor. Los párrafos siguientes describen estos elementos y las condiciones en las que pueden usarse.

Directivas JSP

Las directivas se utilizan para proporcionar información al servidor cuando éste está tratando la página. No actúan directamente sobre el contenido devuelto al navegador. La sintaxis general de uso de una directiva es la siguiente:

```
<%@nombreDeLaDirectiva atributos >
```

Tres directivas diferentes se pueden usar en una página JSP. La directiva `page` permite configurar características del código generado durante el tratamiento de la página JSP. La directiva `include` permite insertar automáticamente texto o código mientras se trata la página JSP. La directiva `taglib` se usa para la declaración de librerías de etiquetas externas.

1. La directiva page

La directiva `page` se usa para definir las características del código generado durante el tratamiento de la página JSP. Esta directiva puede aparecer en cualquier parte de la página JSP pero por evidentes razones de legibilidad es mejor colocarla al comienzo del archivo. Puede usarse varias veces en el mismo archivo con la condición de que no redefina un atributo ya definido. La única excepción a esta regla es el uso del atributo `import`.

Se pueden usar los atributos siguientes en esta etiqueta:

- `language="lenguaje"`: este atributo indica qué lenguaje se utiliza en el código de la página JSP. El valor por defecto de este atributo es, por supuesto, el lenguaje Java.
- `extends="paquete.nombreDeClase"`: determina el nombre de la clase de la que va a heredar la clase generada por el tratamiento de la página JSP realizado por el servidor. El nombre tiene que incluir además el nombre del paquete.
- `import="nombreDePaquete1, nombreDePaquete2..."`: este atributo permite establecer los paquetes importados en el código generado por el servidor. Se pueden definir múltiples paquetes simultáneamente separando sus nombres por una coma en el valor asignado a este atributo. Si hay que importar muchos paquetes es mejor utilizar varias directivas `page` con el atributo `import`, éste es, por otra parte, el único caso en que un atributo puede usarse varias veces en la directiva `page`. Para simplificar la escritura de una página JSP, tres paquetes se importan por defecto en la transformación de la página JSP en código Java. Por lo tanto, el servidor añade automáticamente las siguientes líneas en el código que genera.

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

- `session="booleano"`: este atributo determina si la sesión es accesible desde la página JSP. Si éste es el caso, la variable `session`, accesible desde la página JSP, le hace referencia.
- `buffer="tamaño del buffer"`: este atributo establece el tamaño del buffer utilizado por el objeto `JspWriter` de la página JSP que sirve para escribir los datos en la respuesta HTTP enviada al cliente. El valor `none` puede usarse para desactivar el uso de un buffer y escribir los datos directamente al cliente.
- `autoFlush="booleano"`: este atributo indica si el contenido del buffer tiene que enviarse automáticamente al cliente cuando esté lleno. Si este atributo es igual a `false`, se lanzará una excepción cuando se llene el buffer. Este atributo sólo puede usarse si hay un buffer activo para la página JSP.
- `isThreadSafe="booleano"`: este atributo indica el modelo de thread usado para el tratamiento de la petición HTTP relativa a la página JSP. Si este atributo es igual a `true`, el servidor considera que el código de la página JSP puede utilizarse por múltiples threads simultáneamente con total seguridad. Esto supone que las partes de código peligrosas están ubicadas en bloques sincronizados. Si el valor de este atributo es igual a `false`, una sola petición se tratará a la vez para cada página JSP. Esto es equivalente a la implementación de la interfaz `SingleThreadModel` de un servlet.
- `errorPage="url relativa"`: cuando se lanza una excepción en la página JSP y no está controlada en ningún bloque `try catch`, el servidor llama automáticamente al recurso indicado en esta etiqueta.

- `isErrorPage="booleano"`: este atributo indica si la página se ha creado para el tratamiento de excepciones. Si éste es el caso, la variable `exception` permite recuperar la excepción que ha provocado la visualización de esta página.
- `contentType="tipo Mime"`: éste indica cuál es el tipo de documento contenido en la respuesta HTTP generada por el tratamiento de la página JSP. El valor por defecto de este atributo es `text/html`.
- `pageEncoding="tipo de codificación"`: este atributo determina el tipo de codificación de los caracteres que forman parte de la respuesta HTTP.

Ejemplo de directiva page:

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

2. La directiva include

Esta directiva permite incluir otro recurso en una página JSP. Este recurso puede ser un documento HTML, texto, JSP o XML. Sin embargo, no se puede incluir un servlet ya que la inclusión se realiza en el momento de la conversión de la página JSP. El contenido de la etiqueta `include` simplemente se reemplaza por el contenido del archivo indicado por el atributo `file`.

Ejemplo de uso de la directiva include:

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Visualización de la fecha y hora</title>
    </head>
    <body>
        <%@include file="/logo.html" %>
        <h1>Buenos días</h1>
        estamos a <% out.println(new
GregorianCalendar().getTime().toLocaleString());%>
    </body>
</html>
```

El archivo logo.html:

```
<table>
    <tr>
        <td>
            
        </td>
        <td>
            <h1>ESCUELA ENI</h1>
        </td>
    </tr>
</table>
```

Extracto del archivo Java generado por el servidor:

```
out.write("<html>\n");
    out.write("    <head>\n");
    out.write("        <meta http-equiv=\"Content-Type\"
content=\"text/html; charset=UTF-8\">\n");
    out.write("        <title>Visualización de fecha y hora</title>\n");
    out.write("    </head>\n");
    out.write("    <body>\n");
```

```

out.write("      ");
out.write(" <table>\n");
out.write("       <tr>\n");
out.write("         <td>\n");
out.write("           <img
src=\"images/escuelaeni.gif\" width=\"65\" height=\"65\""
alt="escuelaeni\"/>\n");
out.write("           </td>\n");
out.write("           <td>\n");
out.write("             <h1>ESCUELA ENI</h1>\n");
out.write("           </td>\n");
out.write("       </tr>\n");
out.write(" </table>\n");
out.write("\n");
out.write("      <h1>Buenos días</h1>\n");
out.write("      estamos a ");
out.println(new GregorianCalendar().getTime().toLocaleString());
out.write("\n");
out.write("      \n");
out.write("    </body>\n");
out.write("</html>\n");

```

3. La directiva taglib

Esta directiva permite referenciar librerías de etiquetas externas. Tiene que insertarse en la página JSP cuando se desea utilizar librerías de etiquetas personalizadas. El atributo `uri` contiene la URI (*Uniform Resource Identifier*) asociada al archivo de descripción de la librería. El valor de este atributo tiene que corresponder al valor definido en el archivo `tld` (*Tag Library Descriptor*). Este archivo contiene un documento XML que describe la librería. Contiene datos referentes a la librería y las etiquetas que contiene. El servidor utiliza este archivo para validar la sintaxis de las etiquetas personalizadas en la página JSP.

El segundo atributo `prefix` define qué prefijo se usará en la página JSP para referenciar las etiquetas de esta librería.

Por ejemplo, para poder utilizar etiquetas de la librería cuyo archivo `tld` se muestra a continuación:

```

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <description>JSTL 1.1 i18n-capable formatting library</description>
  <display-name>JSTL fmt</display-name>
  <tlib-version>1.1</tlib-version>
  <short-name>fmt</short-name>
  <uri>http://java.sun.com/jsp/jstl/fmt</uri>

  <validator>
    <description>
      Provides core validation features for JSTL tags.
    </description>
    <validator-class>
      org.apache.taglibs.standard.tlv.JstlFmtTLV
    </validator-class>
  </validator>
  ...
</taglib>

```

Hay que añadir la directiva siguiente en la página JSP:

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

Las etiquetas de esta librería son accesibles desde la página JSP usando el prefijo `fmt`.

```

<fmt:setBundle basename="cuenta.recursos" />
<fmt:message key="inicio"/>

```

Scriptlets

Los scriplets permiten insertar código Java en una página JSP. Hay cuatro etiquetas distintas para insertar código Java.

1. Declaración de variables o de métodos <%! %>

Esta etiqueta tiene que utilizarse cuando se necesita definir en el interior de una página JSP variables o métodos que a continuación se usarán en otros scriplets de la página. La declaración se tiene que hacer en el interior de la etiqueta <%! %>. En su interior sólo se pueden escribir declaraciones de variables o declaraciones de métodos. No está permitido poner instrucciones Java directamente en esta etiqueta. Por contra, la ubicación de esta etiqueta en la página JSP no tiene importancia y puede colocarse incluso al final de la página JSP aunque los elementos que contiene hayan sido ya usados en la página. Sin embargo, lógicamente, para mejorar la legibilidad se recomienda colocar esta etiqueta al inicio de la página.

Ejemplo de declaración de una variable y de un método:

```
<%!
    double iva=1.16;
    double calcularTarifa(double precio)
    {
        return precio*iva;
    }
%>
```

2. Inserción de instrucciones Java <% %>

La etiqueta <% %> puede contener una o varias instrucciones Java. Cuando el servidor analiza la página, las instrucciones colocadas en esta etiqueta se ponen, tal cual, en el método `_jspService` de la clase generada. Esta etiqueta también puede contener declaraciones de variables. Sin embargo, estas variables sólo pueden usarse después de sus declaraciones. Por contra, se prohíbe la declaración de métodos en esta etiqueta. En efecto, siempre hay que tener presente que el contenido de esta etiqueta se inserta en el método `_jspService` de la clase generada. Si esta etiqueta contiene la declaración de un método, se encontraría con un método definido en el interior de otro método, lo que está claramente prohibido en Java.

Ejemplo de uso de la etiqueta <% %>:

```
<%
    double iva=1.16;
    out.println( Double.parseDouble(request.getParameter("precio"))*iva);
%>
```

La variable `iva` declarada en este ejemplo puede usarse en otros scriplets colocados después de éste en la página JSP.

3. Evaluación de una expresión <%= %>

Esta etiqueta permite insertar en la respuesta HTTP enviada al cliente, el resultado de la expresión que contiene. La expresión se convierte en una cadena de caracteres. Si la expresión contiene un objeto, el método `toString` de este objeto se invoca automáticamente. La expresión no tiene que finalizar con punto y coma ya que en el momento en el que el servidor la traduce en código Java ésta se pasa como parámetro al método `out.print`.

Ejemplo de expresión:

```
<%=new GregorianCalendar().getTime().toLocaleString()%>
```

Esta expresión se traduce por la línea siguiente en el código Java generado por el servidor:

```
out.print(new GregorianCalendar().getTime().toLocaleString());
```

4. Comentarios <%-- --%>

Todo lo que se encuentra en el interior de esta etiqueta se ignora en el proceso de análisis de la página JSP realizado por el servidor. No hay nada que deba insertarse en la respuesta enviada al cliente ni siquiera en el código Java generado por el servidor.

```
<%-- cálculo del precio con iva --%>
<%
    double iva=1.16;
    out.println( Double.parseDouble(request.getParameter("precio"))*iva);

%>
```

Objetos implícitos

Para simplificar al máximo el código de los scriptlets de una página JSP, ciertos objetos de uso común son accesibles directamente mediante unas variables predefinidas.

`request`: esta variable permite obtener una referencia al objeto `HttpServletRequest` utilizado para contactar con la página JSP.

`response`: esta variable permite obtener una referencia al objeto `HttpServletResponse` que se usa para transferir al cliente el resultado del tratamiento de la página JSP. Este objeto no es muy útil debido a que hay muchas otras alternativas disponibles para insertar información en la respuesta HTTP realizada al cliente.

`pageContext`: esta variable permite acceder al objeto `pageContext` asociado a la página JSP. Es posible acceder mediante este objeto al resto de objetos de la página JSP.

`session`: esta variable contiene una referencia al objeto `HttpSession` asociado al cliente. Para que esta variable sea accesible no hay que haber desactivado el uso de sesiones en la directiva `page` de la página JSP (ver sección Directivas JSP - La directiva `page` de este capítulo).

`application`: esta variable referencia al objeto `ServletContext` asociado a la aplicación web que contiene la página JSP.

`out`: esta variable representa el flujo de salida en forma de un objeto `JspWriter` que permite escribir en el cuerpo de la respuesta HTTP. El scriptlet `<%= %>`, entre otros, usa este objeto.

`config`: esta variable permite referenciar al objeto `ServletConfig` asociado a la página JSP. Las propiedades de este objeto se inicializan con los datos que conciernen a esta página JSP resultantes del descriptor de despliegue.

`exception`: esta variable está disponible únicamente en una página JSP dedicada a la gestión de errores. Permite obtener una referencia al objeto `Exception` en el origen de la redirección a esta página de gestión del error.

Etiquetas JSP

Las etiquetas JSP se pueden usar para insertar en una página JSP acciones de uso común en la creación de ésta. El objetivo de estas etiquetas es el de limitar al máximo la utilización de scriptlets en las páginas JSP. Cuando el servidor analiza la página JSP, estas etiquetas se reemplazan por su equivalente en forma de código Java. Los atributos de estas etiquetas se usan para personalizar el código generado.

1. Etiqueta <jsp:useBean>

El objetivo de esta etiqueta es recuperar o crear una instancia de un objeto javaBean. La primera operación efectuada por esta etiqueta consiste en buscar en la ubicación especificada por el atributo `scope` si una instancia del JavaBean existe con el nombre indicado por el atributo `id`. Si no se encuentra ninguna instancia en esta ubicación con el nombre correspondiente, la etiqueta crea una nueva instancia con el constructor por defecto de la clase identificada por el atributo `class`. Seguidamente, la instancia creada se pone en la ubicación indicada por el atributo `scope` con el nombre indicado por el atributo `id`. A continuación, el cuerpo de la etiqueta se ejecuta para permitir la inicialización de la nueva instancia que se ha creado.

Cuando el JavaBean se ha instanciado o se ha recuperado en su ubicación de origen, se crea una variable que permite referenciarlo en la página JSP. Para llamar a esta variable se usa el valor del atributo `id`.

La sintaxis de esta etiqueta es, por lo tanto, la siguiente:

```
<jsp:useBean id="..." scope="..." type="..." class="..."  
beanName="..." > código de inicialización </jsp:useBean>
```

En esta etiqueta, los atributos tienen el significado siguiente:

`id`: en una primera etapa, este atributo representa el nombre con el que el JavaBean se busca. A continuación, se usa para nombrar la variable de acceso desde la página JSP que permite la manipulación del JavaBean.

`scope`: este atributo indica la ubicación donde buscar el JavaBean y donde se almacenará si la etiqueta tiene que instanciarlo. Estos son los valores posibles para este atributo:

- `page`
- `request`
- `session`
- `application`

`class`: este atributo indica el nombre completo de la clase (con el paquete) utilizada para instanciar el JavaBean en el caso de que éste no se encuentre en la ubicación indicada por el atributo `scope`.

`type`: este atributo determina el tipo de la variable utilizada para referenciar el JavaBean en la página JSP. Este tipo tiene que ser el mismo que el indicado por el atributo `class` o un supertipo de éste.

`beanName`: este atributo se utiliza en vez del atributo `class`. Si el JavaBean no existe, será, en este caso, creado a partir de un archivo de serialización.

El ejemplo siguiente busca en la petición un JavaBean con el nombre `elCliente`:

```
<jsp:useBean  
    id="elCliente"  
    type="pk1.Persona"  
    class="pk1.Cliente"  
    scope="request">  
</jsp:useBean>
```

Tras el análisis realizado por el servidor, esta etiqueta se transforma en el siguiente código Java:

```
pk1.Persona cliente = null;  
synchronized (request)  
{  
    cliente = (pk1.Persona)
```

```

_jspx_page_context.getAttribute("cliente", PageContext.REQUEST_SCOPE);
if (cliente == null)
{
    cliente = new pk1.Cliente();
    _jspx_page_context.setAttribute("cliente", cliente,
PageContext.REQUEST_SCOPE);
}
}

```

2. Etiqueta <jsp:getProperty>

Esta etiqueta se usa conjuntamente con la etiqueta anterior para insertar en la página JSP el valor de una propiedad de un JavaBean. Esta etiqueta necesita dos atributos:

- **name:** este atributo representa el nombre del JavaBean presente en la página JSP desde la cual se obtiene la propiedad. Tiene que corresponder al valor del atributo **id** de la etiqueta <jsp:useBean>. La etiqueta <jsp:useBean> tiene que aparecer en la página JSP antes que la etiqueta <jsp:getProperty>.
- **property:** este atributo especifica el nombre de la propiedad que se buscará. El método `getXXXXXX` se utiliza en el JavaBean para obtener el valor de esta propiedad.

El ejemplo siguiente inserta el valor de la propiedad apellidos del JavaBean llamado elCliente en la página JSP.

```
<jsp:getProperty name="elCliente" property="apellidos"/>
```

El código Java correspondiente:

```

out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString((
(pk1.Cliente)_jspx_page_context.findAttribute("elCliente")).getApellidos(
)));

```

3. Etiqueta <jsp:setProperty>

Esta etiqueta permite la asignación de un valor a una propiedad de un JavaBean. Como en el caso de la etiqueta <jsp:getProperty>, antes se debe haber usado una etiqueta <jsp:useBean> para que el JavaBean esté disponible en la página. Esta etiqueta también puede utilizarse para transferir los valores resultantes de un formulario a las propiedades de un JavaBean. Los atributos siguientes permiten la configuración de esta etiqueta.

name: este atributo permite identificar el JavaBean cuya propiedad o propiedades se modificarán.

property: este atributo precisa el nombre de la propiedad que va a modificar esta etiqueta. También puede usarse el valor * en este atributo. En este caso, la etiqueta busca en la petición HTTP los parámetros que tengan el mismo nombre que las propiedades del JavaBean y transfiere el valor de estos parámetros a las propiedades correspondientes de éste. Por supuesto, en el diseño del formulario de introducción de datos HTML hay que utilizar en los campos los mismos nombres que los usados para las propiedades del JavaBean que se actualizará.

value: este atributo tiene que usarse cuando se desea modificar una propiedad de un JavaBean. Es el que indica el valor a transferir a la propiedad del JavaBean.

El primer ejemplo, mostrado a continuación, recupera un JavaBean en la sesión y le asigna un valor a su propiedad horaConexion.

```

<jsp:useBean id="cliente" scope="session" class="pk1.Cliente"
type="pk1.Cliente"/>
<jsp:setProperty name="cliente" property="horaConexion"
value="<%=new Date().toLocaleString()%>"/>

```

El código Java generado por estas dos etiquetas:

```

k1.Cliente cliente = null;
synchronized (session)
{
    cliente = (pk1.Cliente)
_jspx_page_context.getAttribute("cliente", PageContext.SESSION_SCOPE);
    if (cliente == null)

```

```

    {
        cliente= new pk1.Cliente();
        _jspx_page_context.setAttribute("cliente", cliente,
PageContext.SESSION_SCOPE);
    }
}
org.apache.jasper.runtime.JspRuntimeLibrary.handle SetProperty(_jspx
_page_context.getAttribute("cliente"), "horaConexion",
new Date().toLocaleString());

```

El segundo ejemplo utiliza el formulario HTML siguiente para permitir la introducción del nombre y los apellidos del cliente.

```

<html>
<head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<form action="inicioCliente.jsp">
    <table>
        <tbody>
            <tr>
                <td>nombree</td>
                <td><input type="text" name="nombre" value="" /></td>
            </tr>
            <tr>
                <td>apellidos</td>
                <td><input type="text" name="apellidos" value="" /></td>
            </tr>
            <tr>
                <td></td>
                <td></td>
            </tr>
            <tr>
                <td><input type="submit" value="Validar" name="validar" /></td>
                <td><input type="submit" value="Anular" name="anular" /></td>
            </tr>
        </tbody>
    </table>
</form>
</body>
</html>

```

Los datos recopilados en este formulario se envían a la página JSP mostrada a continuación que inicializa un JavaBean y muestra a continuación sus propiedades.

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page import="java.util.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title></title>
    </head>
    <body>
        <jsp:useBean id="cliente" scope="session"
class="pk1.Cliente" type="pk1.Cliente"/>
        <jsp:setProperty name="cliente" property="*"/>
        <jsp:setProperty name="cliente" property="horaConexion"
value="<%new Date().toLocaleString()%"/>
        Bienvenido
        <jsp:getProperty name="cliente" property="nombre"/>
        <jsp:getProperty name="cliente" property="apellidos"/>
        hora de conexión
        <jsp:getProperty name="cliente" property="horaConexion"/>
    </body>

```

```
</html>
```

Las etiquetas JSP generan el código siguiente:

```
pk1.Cliente cliente = null;
    synchronized (session) {
        cliente = (pk1.Cliente)
_jspx_page_context.getAttribute("cliente", PageContext.SESSION_SCOPE);
    if (cliente == null){
        cliente = new pk1.Cliente();
        _jspx_page_context.setAttribute("cliente", cliente,
PageContext.SESSION_SCOPE);
    }
}
out.write("\n");
out.write("      ");
org.apache.jasper.runtime.JspRuntimeLibrary.introspect(_jspx_page_
context.findAttribute("cliente"), request);
out.write("\n");
out.write("      ");
org.apache.jasper.runtime.JspRuntimeLibrary.handleSetProperty(_jspx
_page_context.findAttribute("cliente"), "horaConexion",
new Date().toLocaleString());
out.write("\n");
out.write("      Bienvenido\n");
out.write("      ");
out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString((((
pk1.Cliente)_jspx_page_context.getAttribute("cliente")).getNombre())));
out.write("\n");
out.write("      ");
out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString((((
pk1.Cliente)_jspx_page_context.getAttribute("cliente")).getApellidos())));
out.write("\n");
out.write("      hora de conexion\n");
out.write("      ");
out.write(org.apache.jasper.runtime.JspRuntimeLibrary.toString((((
pk1.Cliente)_jspx_page_context.getAttribute("cliente")).getHoraConexion())));
```

4. Etiqueta <jsp:include>

Esta etiqueta realiza la misma función que el método `include` del objeto `RequestDispatcher` debido a que permite incluir en la respuesta el contenido de otro recurso estático o dinámico. Cuando se incluye un recurso dinámico, por ejemplo un servlet, se le pueden pasar parámetros añadiendo en el interior de esta etiqueta una o varias etiquetas `<jsp:param>`. Esta etiqueta acepta dos atributos.

`page`: la URL del recurso que se incluirá en la respuesta HTTP se indica con este parámetro. El recurso que se incluirá tiene que pertenecer a la misma aplicación.

`flush`: el valor booleano de este atributo indica si el buffer tiene que ser enviado al cliente antes de que el recurso se haya incluido.

La etiqueta `<jsp:param>`, que permite transmitir parámetros al recurso incluido, espera el atributo `name` para especificar el nombre del parámetro y el atributo `value` para especificar el valor del parámetro.

El ejemplo siguiente de página JSP realiza una llamada a un servlet que buscará en la base de datos el número de pedidos del cliente cuyo código se ha pasado en una petición HTTP.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Usted ha enviado
        <jsp:include page="/NumPedidos">
```

```

<jsp:param name="codigo"
value='<%=request.getParameter("codigoCliente") %>' />
</jsp:include>
pedidos </h1>
</body>
</html>

```

El código Java generado por esta etiqueta:

```

out.write("      <h1>Usted ha enviado\n");
out.write("      ");
org.apache.jasper.runtime.JspRuntimeLibrary.include(request,
response, "/NumPedidos" + (("/NumPedidos").indexOf('?')>0? '&':
'?') + org.apache.jasper.runtime.JspRuntimeLibrary.URLEncode("codigoCliente",
request.getCharacterEncoding())+ "=" +
org.apache.jasper.runtime.JspRuntimeLibrary.URLEncode(String.valueOf
(request.getParameter("codigoCliente")) ),
request.getCharacterEncoding(), out, false);
out.write("\n");
out.write("      pedidos </h1>\n");

```

El código del servlet llamado:

```

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

public class NumPedidos extends HttpServlet {

protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
    PrintWriter out = response.getWriter();
    try
    {
        InitialContext cx;
        Connection cn;
        DataSource ds;
        cx=new InitialContext();
        ds=(DataSource)cx.lookup("java:comp/env/jdbc/northwind");
        cn=ds.getConnection();
        PreparedStatement sm;
        ResultSet rs;
        sm=cn.prepareStatement("select count(orderid) from
orders where customerid=?");
        sm.setString(1,request.getParameter("codigo"));
        rs = sm.executeQuery();
        rs.next();
        out.println(rs.getString(1));

    } catch (SQLException ex) {
        Logger.getLogger(NumPedidos.class.getName()).log(Level.SEVERE,
null, ex);
    } catch (NamingException ex) {

```

```

        Logger.getLogger(NumPedidos.class.getName()).log(Level.SEVERE,
null, ex);
    }
}

@Override
protected void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}

}

```

5. Etiqueta <jsp:forward>

Esta etiqueta realiza la misma función que el método `forward` de un objeto `RequestDispatcher`. Los atributos de esta etiqueta son completamente idénticos a los de la etiqueta `<jsp:include>`. Sin embargo, contrariamente a ésta, los elementos que vienen detrás de esta etiqueta no se tratan.

6. Etiqueta <jsp:plugin>

Esta etiqueta gestiona la inserción de un applet en una página JSP con el objetivo de que ésta se ejecute en el navegador. Cuando el servidor analiza la página, esta etiqueta se reemplaza por el código HTML necesario para la inserción de un applet. Este código se adapta en función del tipo de navegador. Esta etiqueta tiene muchos atributos.

`type`: este atributo indica el tipo de objeto que debe ser ejecutado por el navegador: `bean` o `applet`.

`code`: este atributo especifica el nombre de la clase principal que debe ser ejecutada por el navegador. Se tiene que incluir la extensión `.class`.

`codebase`: este atributo indica el directorio en el que se encuentra el archivo que contiene la clase Java que se ejecutará. Si este código se encuentra en el mismo directorio que la página JSP, hay que utilizar el carácter `.` como nombre de directorio.

`archive`: cuando un applet necesita varios archivos para funcionar, es preferible juntarlos todos en un empaquetado Java (archivo `jar`). Este atributo proporciona el nombre del archivo que el navegador se descargará.

`name`: este atributo permite asignar un nombre a esta instancia del applet. Este nombre se utiliza cuando se desea establecer una comunicación entre varios applets de una página HTML.

`align`: este atributo permite posicionar el applet en la página HTML. Este posicionamiento se realiza en relación a la línea de texto actual. Se admiten los siguientes valores:

- bottom
- top
- middle
- left
- right

`width`: cuando el navegador analiza y muestra la página HTML, tiene que reservar espacio para la visualización del applet. Este atributo indica el número de píxeles o el porcentaje de la anchura de la página asignado a la

visualización del applet.

height: este atributo también se utiliza para el dimensionamiento del applet pero concierne a la altura del applet.

hspace y vspace: estos dos atributos indican los márgenes horizontal y vertical en píxeles entre los bordes del applet y los elementos que lo rodean.

jreversion: este atributo indica la versión del JRE necesario para que el navegador pueda ejecutar el applet.

nspluginurl: este atributo indica la ubicación donde un navegador basado en Netscape podrá descargar el plugin necesario para la ejecución del applet.

iepluginurl: este atributo indica la ubicación donde un navegador Internet Explorer podrá descargar el plugin necesario para la ejecución del applet.

Si el plugin necesario para la ejecución del applet no se ha podido obtener, el mensaje indicado en la etiqueta <jsp:fallback> se visualizará.

A veces es útil proporcionar información al applet antes de su ejecución. Para ello, la etiqueta <jsp:param> permite definir en la página los parámetros que pueden ser recuperados por el applet gracias al método getParameter. Esta etiqueta espera los atributos name y value que indican el nombre y el valor del parámetro. Esta etiqueta tiene que colocarse en el interior de la etiqueta <jsp:plugin>.

Ejemplo de utilización de la etiqueta <jsp:plugin>

```
<jsp:plugin code="PruebaApplet2.class" codebase=". "
type="applet"
align="left,top"
width="50%"
archive="codigoapplet.jar"
height="200"
hspace="25"
vspace="10"
name="gestion"
jreversion="1.6">
<jsp:params>
<jsp:param name="color" value="rojo"/>
<jsp:param name="talla" value="50" />
</jsp:params>
<jsp:fallback>imposible ejecutar el applet</jsp:fallback>
</jsp:plugin>
```

El navegador recibe el código HTML siguiente:

```
<object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
name="gestion" width="50%" height="200" hspace="25" vspace="10"
align="left,top"
codebase="http://java.sun.com/products/plugin/1.2.2/jinstall-
1_2_2-win.cab#Version=1,2,2,0">
<param name="java_code" value="PruebaApplet2.class">
<param name="java_codebase" value=". ">
<param name="java_archive" value="codigoapplet.jar">
<param name="type" value="application/x-java-applet;version=1.6">
<param name="color" value="rojo">
<param name="talla" value="50">
<comment>
<EMBED type="application/x-java-applet;version=1.6" name="gestion"
width="50%" height="200" hspace="25" vspace="10" align="left,top"
pluginspage="http://java.sun.com/products/plugin/"
java_code="PruebaApplet2.class" java_codebase=". "
java_archive="codigoapplet.jar" color="rojo" talla="50"/>
<noembed>
imposible ejecutar el applet
</noembed>
</comment>
</object>
```

Sintaxis XML

La sintaxis clásica de una página JSP no constituye un documento XML bien formado (respecto a las reglas de sintaxis del lenguaje XML). Sin embargo, a veces es conveniente que una página JSP esté representada por un documento XML bien formado. Para ello, prácticamente todos los elementos JSP descritos en las secciones anteriores tienen una versión XML. A continuación se muestra la correspondencia entre la versión JSP clásica y la versión XML de estos elementos.

Sintaxis JSP	Sintaxis XML	Observaciones
<%@page %>	<jsp:directive.page />	Los atributos son idénticos en ambas versiones.
<%@include %>	<jsp:directive.include />	Los atributos son idénticos en ambas versiones.
<%@taglib %>	Sin correspondencia	Utilizar el atributo xmlns:XXX de la etiqueta <jsp:root>.
<%! %>	<jsp:declaration> </jsp:declaration>	
<% %>	<jsp:scriptlet> </jsp:scriptlet>	
<%-- --%>	Sin correspondencia	
<%= %>	<jsp:expression> </jsp:expression>	

Cuando una página utiliza la sintaxis XML, hay que utilizar la etiqueta <jsp:root> como elemento raíz de la página.

Presentación

Es frecuente que el desarrollo de una aplicación se asigne a varias personas en función de sus especialidades. El aspecto y la presentación de una aplicación generalmente se delega en una persona que domina perfectamente el lenguaje HTML y todas las tecnologías asociadas (hojas de estilo, grafismo...). En el diseño de una página JSP, estas competencias, por supuesto, también son necesarias pero también es indispensable dominar el lenguaje Java con el fin de poder crear las partes dinámicas de la página. Para un grafista puede que sea a veces muy molesto tener que intervenir a este nivel. El objetivo de la librería JSTL (*java standard tag library*) es el de proporcionar a estos perfiles la posibilidad de insertar porciones dinámicas en una página JSP sin estar obligados a aprender el lenguaje Java. Esta librería permite realizar un conjunto de operaciones de uso común en el diseño de la parte dinámica de una página JSP. Propone un conjunto de etiquetas XML que permiten insertar en la página JSP partes dinámicas. Los diseñadores de páginas web, que están habituados a este estilo de lenguaje, se verán mucho más capaces utilizando esta técnica.

Para que esta librería de etiquetas sea accesible desde una página JSP, basta simplemente con referenciarla en la página con la etiqueta <%@taglib %>. Esta etiqueta contiene los atributos *uri* y *prefix*. El atributo *uri* identifica la librería de etiquetas que se usará y el atributo *prefix* se usa para indicar el prefijo que se utilizará para acceder a sus distintas etiquetas. Para las librerías estándar proporcionadas por SUN se usarán los valores siguientes.

Librería	prefix	uri
Librería básica	c	http://java.sun.com/jsp/jstl/core
Librería XML	x	http://java.sun.com/jsp/jstl/xml
Librería de internacionalización	fmt	http://java.sun.com/jsp/jstl/fmt
Librería SQL	sql	http://java.sun.com/jsp/jstl/sql

Las declaraciones que se incluirán en una página JSP son las siguientes:

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

<%@taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>

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

Las expressions language

Las expressions language se utilizan en el interior de una página JSP con el fin de facilitar la manipulación de datos que son accesibles desde esta página. Estos datos pueden estar presentes en la página, la petición HTTP, la sesión o la misma aplicación. Las expressions language reemplazan el uso de scriptlets de acceso a estos objetos.

Pueden aparecer en los atributos de etiquetas JSP o como texto simple en la página JSP.

1. Declaración de una expression language

La sintaxis general de una expression language tiene el aspecto siguiente:

```
 ${ expresión }
```

La expresión identifica el objeto que se desea obtener. Esta expresión se descompone en dos partes. La primera identifica el contexto o ámbito donde tiene que realizarse la búsqueda y la segunda identifica el objeto de esta ubicación. Esta segunda parte puede ser un índice en una tabla o en una lista, una clave en un objeto de tipo Map o la propiedad de un objeto. La especificación de índices o de claves tiene que ponerse entre los caracteres [y]. Según el caso, será una cadena de caracteres o un entero. El acceso a una propiedad se realiza con el operador . del lenguaje Java.

Varios elementos predefinidos permiten identificar los contextos donde pueden realizarse las búsquedas. Si no se indica ninguno, la búsqueda se realiza en este orden: página, petición, sesión y finalmente aplicación. Se puede acceder a los siguientes contextos:

`pageScope`: este elemento permite obtener una referencia a los atributos del objeto `PageContext` asociados a la página JSP. También hay que proporcionar mediante una cadena de caracteres la clave asociada al atributo del que se desea obtener el valor.

El ejemplo siguiente representa una expresión que permite el acceso a un atributo llamado `nombreCliente` en la página JSP.

```
 ${pageScope[ "nombreCliente" ]}
```

 Se trata de un atributo de la página JSP y no de una variable declarada en un scriptlet.

`pageContext`: este elemento permite el acceso al objeto `PageContext` asociado a la página JSP. Desde este objeto se puede acceder a los objetos implícitos disponibles en una página JSP (petición, respuesta, sesión...) así como a sus propiedades. Basta con completar la expresión con el nombre del objeto y la propiedad que se desea utilizar.

El ejemplo siguiente permite obtener el tamaño del buffer asociado a la respuesta HTTP.

```
 ${pageContext.response.bufferSize}
```

`requestScope`: este elemento permite obtener una referencia a los atributos del objeto `request` utilizado para obtener la página JSP. También hay que proporcionar la clave con una cadena de caracteres asociada al atributo del que se desea obtener su valor.

El ejemplo siguiente representa una expresión que permite el acceso a un atributo llamado `cliente` en la petición HTTP.

```
 ${requestScope[ "cliente" ]}
```

`sessionScope`: esta expresión permite extraer un elemento almacenado a nivel de sesión. También hay que proporcionar una cadena de caracteres que representa el nombre utilizado cuando el objeto se almacenó en la sesión. La expresión puede completarse con el nombre de una propiedad.

El ejemplo siguiente representa una expresión que permite obtener la propiedad `nombre` de un objeto `cliente` que se ha almacenado previamente en la sesión.

```
 ${sessionScope[ "cliente" ].nombre}
```

`applicationScope`: este elemento permite obtener una referencia a los atributos del objeto `servletContext` asociado a la aplicación. También hay que proporcionar mediante una cadena de caracteres la clave asociada al atributo del que se desea obtener su valor.

El ejemplo siguiente representa a una expresión que permite el acceso a un atributo llamado `tasas` en el contexto de la aplicación.

```
 ${applicationScope[ "tasas" ]}
```

initParam: este elemento permite extraer un elemento de los parámetros de inicialización de la aplicación. Éstos tienen que estar definidos en el descriptor de despliegue de la aplicación (web.xml). Además, hay que especificar el nombre del parámetro que se desea extraer.

Declaración de un parámetro en web.xml

```
<context-param>
    <param-name>nombreArchivoLog</param-name>
    <param-value>logApli.txt</param-value>
</context-param>
```

Código de acceso al parámetro desde una página JSP

```
 ${initParam[ "nombreArchivoLog" ]}
```

param: este elemento permite acceder a los parámetros de la petición HTTP. El nombre del parámetro tiene que añadirse para completar la expresión.

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

paramValues: este elemento tiene la misma función que el anterior pero tiene que usarse cuando un mismo parámetro aparece varias veces en la petición HTTP. Es, por ejemplo, el caso de una petición HTTP generada a partir de un formulario HTML que contiene una lista de selección múltiple. Los parámetros se obtienen recopilados en una colección. A continuación, hay que realizar una iteración para recorrer todos los parámetros y después para cada parámetro realizar una segunda iteración para obtener el valor o valores de este parámetro.

```
<!-- bucle para recorrer parámetros -->
<!-- y visualización del nombre del parámetro -->
<c:forEach var='parametro' items='${paramValues}'>
    <c:out value='${parametro.key}' /> =
    <!-- bucle para recorrer los valores de un parámetro -->
    <!-- y visualización de estos valores -->
    <c:forEach var='valor' items='${parametro.value}'>
        <c:out value='${valor}' escapeXml="false"/><br>
    </c:forEach>
</c:forEach>
```

header: este elemento permite acceder a las cabeceras de la petición HTTP. El nombre de la cabecera de la que se desea obtener el valor tiene que añadirse para completar la expresión.

```
 ${header[ "user-agent" ]}
```

headerValues: este elemento funciona siguiendo el mismo principio que el elemento paramValues pero en las cabeceras de la petición HTTP.

```
<!-- bucle para recorrer las cabeceras -->
<!-- y visualización del nombre de la cabecera -->
<c:forEach var='cabecera' items='${headerValues}'>
    <c:out value='${cabecera.key}' /> =
    <!-- bucle para recorrer los valores de la cabecera -->
    <!-- y visualizar estos valores -->
    <c:forEach var='valor' items='${cabecera.value}'>
        <c:out value='${valor}' escapeXml="false"/><br>
    </c:forEach>
</c:forEach><br>
```

Cookie: este elemento permite el acceso a las cookies asociadas a la petición HTTP. El nombre de la cookie de la que se desea obtener su valor tiene que añadirse a la expresión seguido de la propiedad que se desea utilizar. La propiedad value, generalmente, es la más utilizada debido a que permite obtener el contenido de la cookie.

```
 ${cookie[ "identificador" ].value}
```

2. Operadores en expressions language

Hay varios operadores disponibles para efectuar tratamientos en las expressions language. Generalmente, se pueden usar los operadores del lenguaje Java. Sin embargo, algunos de ellos tienen un significado particular en los lenguajes HTML o XML. Éste es el caso, por ejemplo, de los operadores de comparación < y >. Para evitar conflictos hay una versión textual de estos operadores. Las tablas mostradas a continuación presentan los distintos operadores disponibles.

Operadores matemáticos

Operador	Versión textual	Función realizada
+		suma
-		resta
*		multiplicación
/	div	división
%	mod	módulo

Estos operadores sólo pueden usarse con valores numéricos.

Operadores de comparación

Operador	Versión textual	Función realizada
==	eq	igualdad
!=	ne	desigualdad
<	lt	menor que
>	gt	mayor que
<=	le	menor o igual que
>=	ge	mayor o igual que

Cuando los operadores de igualdad y desigualdad trabajan con objetos, utilizan el método `equals` que será necesario redefinir.

Los operadores de inferioridad y superioridad también trabajan con objetos y requieren que los objetos que se compararán implementen la interfaz Comparable. Esta implementación implica la existencia del método `compareTo` utilizado por estos operadores.

Operadores lógicos

Operador	Versión textual	Función realizada
&&	and	y lógico
	or	o lógico
!	not	negación

Estos operadores se utilizan con operandos booleanos o con expresiones que generan resultados booleanos.

Operadores varios

Operador	Versión textual	Función realizada
	empty	prueba si el operando es null, una cadena vacía, una tabla vacía...

()

modifica el orden de evaluación del resto de operadores.

3. Tratamiento de excepciones en las expressions language

Cuando se evalúa una expression language, puede producirse con frecuencia una excepción. Las excepciones que se producen con más frecuencia son las `NullPointerException` y las `ArrayOutOfBoundsException`. Estos dos tipos de excepciones se generan automáticamente por las expressions language. Si una de estas excepciones se lanza en mitad del tratamiento de una expresión, el resultado de la expresión toma automáticamente el valor `null`. Si este resultado tiene que visualizarse en la página JSP, automáticamente se reemplaza por una cadena de caracteres vacía. Para ilustrar la utilidad de las expressions language, a continuación se muestra una pequeña experiencia que consiste en extraer de la sesión un objeto cliente y mostrarlo en una página JSP. La primera versión, mostrada más abajo, utiliza un scriptlet para efectuar esta operación en la página JSP.

```
<%
cuenta.Cliente c;
c=(cuenta.Cliente)session.getAttribute("cliente");
out.println("apellidos del cliente: ");
out.println(c.getApellidos());
out.println("<br>");
out.println("nombre del cliente: ");
out.println(c.getNombre());
out.println("<br>");
out.println("calle: ");
out.println(c.getDireccion().getCalle());
out.println("<br>");
out.println("codigo postal: ");
out.println(c.getDireccion().getCp());
out.println("poblacion: ");
out.println(c.getDireccion().getPoblacion());
%>
```

En esta primera versión no se toma ningún tipo de precaución y se exige que el objeto cliente esté presente en la sesión y esté correctamente inicializado. Si éste no es el caso, se obtiene el resultado siguiente.

Estado HTTP 500 -

```
tipo Informe de excepción
mensaje
descripción El servidor ha encontrado un error interno () que le ha impedido
satisfacer la petición.
excepción
org.apache.jasper.JasperException: An exception occurred processing JSP
page /visualizarCliente1.jsp at line 28

25:         out.println(c.getNombre());
26:         out.println("<br>");
27:         out.println("calle: ");
28:         out.println(c.getDireccion().getCalle());
29:         out.println("<br>");
30:         out.println("codigo postal: ");
31:         out.println(c.getDireccion().getCp());
```

El error proviene de la línea relacionada con la dirección del cliente porque ésta no ha sido inicializada cuando se ha creado el cliente.

Para evitar estos problemas, hay que proporcionar un tratamiento de excepciones cada vez que el objeto cliente se utilice. A continuación se muestra una segunda versión más segura de la página JSP.

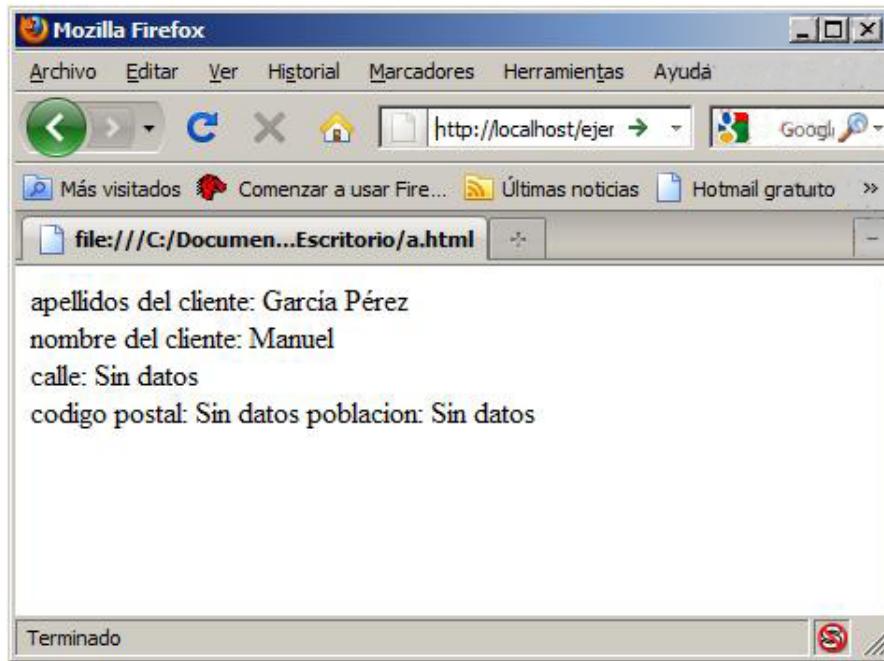
```
<%
cuenta.Cliente c;
c=(cuenta.Cliente)session.getAttribute("cliente");
out.println("apellidos del cliente: ");
try
{
    out.println(c.getApellidos());
```

```

}
catch (Exception ex)
{
    out.println("Sin datos");
}
out.println("<br>");
out.println("nombre del cliente: ");
try
{
    out.println(c.getNombre());
}
catch (Exception ex)
{
    out.println("Sin datos");
}
out.println("<br>");
out.println("calle: ");
try
{
    out.println(c.getDireccion().getCalle());
}
catch (Exception ex)
{
    out.println("Sin datos");
}
out.println("<br>");
out.println("codigo postal: ");
try
{
    out.println(c.getDireccion().getCp());
}
catch (Exception ex)
{
    out.println("Sin datos");
}
out.println("poblacion: ");
try
{
    out.println(c.getDireccion().getPoblacion());
}
catch (Exception ex)
{
    out.println("Sin datos");
}
%>

```

Ya no hay problemas en la ejecución.



Como contrapartida, el código de la página JSP se complica y necesita un buen conocimiento del lenguaje Java. En la tercera versión, que utiliza las expressions language y las etiquetas JSTL, se simplifica enormemente el contenido de la página JSP.

```
apellido del cliente: <c:out  
value='${sessionScope["cliente"].apellidos}' default="Sin datos"/>  
<br>  
nombre del cliente : <c:out  
value='${sessionScope["cliente"].nombre}' default="Sin datos"/>  
<br>  
calle : <c:out  
value='${sessionScope["cliente"].direccion.calle}' default="Sin datos"/>  
<br>  
codigo postal : <c:out  
value='${sessionScope["cliente"].direccion.cp}' default="Sin datos"/>  
poblacion : <c:out  
value='${sessionScope["cliente"].direccion.poblacion}' default="Sin datos"/>
```

Las posibles excepciones se tratan automáticamente, dejando el resultado generado inicializado a null si se lanza una excepción. Posteriormente, la etiqueta se encarga del reemplazo de este valor null por la cadena de caracteres especificada por el atributo default.

La librería básica

Como su nombre indica, las etiquetas de esta librería permiten realizar los tratamientos básicos que se pueden necesitar en una página JSP. Estas etiquetas pueden organizarse en cinco categorías.

- gestión de variables,
- operaciones condicionales,
- iteraciones,
- gestión de las URL,
- etiquetas varias.

1. Etiquetas de gestión de variables

Hay dos etiquetas que permiten la creación y la destrucción de una variable.

a. Asignación de un valor a una variable

La etiqueta `<c:set>` se utiliza para crear una nueva variable o para modificar el contenido de una variable existente. Los atributos siguientes permiten la configuración de esta etiqueta.

`var:` indica el nombre de la variable que se creará o se modificará.

`scope:` indica el ámbito donde se almacenará la variable. Los ámbitos estándar son accesibles con las palabras clave `page`, `request`, `session` y `application`. Si no se especifica ningún ámbito, se usa la página por defecto.

`value:` el valor que se guardará en esta variable. Este valor también se puede especificar en el cuerpo de la etiqueta. Esto permite hacer el código más legible cuando el volumen de datos es importante. Evidentemente, se puede obtener este valor con una `expression language`.

`target:` este atributo se utiliza cuando se desea modificar una propiedad de un objeto. Permite identificar el objeto cuya propiedad se desea modificar. El objeto tiene que existir en el ámbito definido por el atributo `scope` antes del uso de esta etiqueta. Puede crearse mediante la etiqueta `<jsp:usebean>` como se muestra en los ejemplos de más adelante.

`property:` este atributo se utiliza conjuntamente con el anterior para identificar la propiedad del objeto que se desea modificar. Esta propiedad tiene que ser accesible mediante un método `setXXXX`.

Ejemplos de uso:

- Creación en la sesión de una variable llamada `navegador` que contiene la cabecera `user-agent` de la petición HTTP.

```
<c:set var="navegador" scope="session" value='${header["user-agent"]}'/>
```

- Creación en la sesión de una variable que contiene una tabla HTML en la que se han incluido los apellidos y el nombre de un objeto cliente que él mismo extrae de los atributos de la petición HTTP.

```
<c:set var="tabla" scope="session">
<table border="1">
<thead>
<tr>
<th>apellidos</th>
<th>nombre</th>
</tr>
</thead>
<tbody>
<tr>
```

```

        <td>${requestScope["cliente"].apellidos}</td>
        <td>${requestScope["cliente"].nombre}</td>
    </tr>
</tbody>
</table>
</c:set>

```

Esta tabla puede, a continuación, insertarse en cualquier elemento de la aplicación recuperando el contenido de esta variable en la sesión.

```
out.println(session.getAttribute("tabla"));
```

- Creación en los atributos de la petición HTTP de un objeto cliente cuyos apellidos y nombre se inicializan con los parámetros apellidoscliente y nombreciente de la petición HTTP.

```

<jsp:useBean scope="request" id="cliente" class="cuenta.Cliente"/>
<c:set target ="${requestScope['cliente']}"
value='${param["apellidoscliente"]}' property="apellidos"/>
<c:set target= "${requestScope['cliente']}"
value='${param["nombreciente"]}' property="nombre"/>

```

b. Destrucción de una variable

La eliminación de una variable se realiza con la etiqueta `<c:remove>`. Esta etiqueta acepta los atributos `name` y `scope` que permiten identificar el nombre de la variable y dónde se encuentra.

2. Operaciones condicionales

Hay dos etiquetas disponibles para reemplazar las estructuras `if` y `switch` del lenguaje Java.

a. Etiqueta `<c:if>`

Esta etiqueta es fácil de usar debido a que sólo espera un atributo obligatorio. Este atributo, llamado `test`, contiene la expresión que será evaluada. Esta expresión tiene que generar un resultado de tipo booleano. Si el booleano generado es igual a `true`, el cuerpo de la etiqueta se inserta en la página.

```

<c:if test="${!empty sessionScope['cliente'].apellidos }">
    ${sessionScope["cliente"].apellidos }
</c:if>

```

Otra sintaxis permite almacenar en una variable el resultado booleano de la evaluación de la condición. En este caso, hay que utilizar los atributos `var` y `scope` para identificar el nombre y el ámbito de la variable que contendrá el resultado de dicha evaluación. Por supuesto que esta variable se puede usar en otra expresión.

```
<c:if test="${empty sessionScope['cliente'].apellidos }" var="t"/>
```

b. Etiqueta `<c:choose>`

Esta etiqueta realiza el mismo tratamiento que la instrucción `switch` del lenguaje Java. La estructura de uso es, sin lugar a dudas, muy similar, ya que realiza la llamada a otras dos etiquetas que corresponden a las instrucciones `case` y `default` del lenguaje Java. La etiqueta `<c:when>` reemplaza la instrucción `case`. Como ésta, esta etiqueta sólo puede usarse en el interior de la estructura. Por lo tanto, tiene que ser una etiqueta descendiente de la etiqueta `<c:choose>`. La etiqueta `<c:otherwise>` reemplaza la intrucción `default` y está sometida a la misma restricción que la etiqueta `<c:when>`.

Las etiquetas `<c:choose>` y `<c:otherwise>` no aceptan ningún atributo. Solamente la etiqueta `<c:when>` contiene el atributo `test`, utilizado para determinar si el contenido de la etiqueta se insertará o no en el documento. El ejemplo siguiente modifica el color de fondo de la página en función del valor del parámetro `color` recibido en la petición HTTP.

```
<body bgcolor="
```

```

<c:choose>
    <c:when test="${param['color']}==1">
        red
    </c:when>
    <c:when test="${param['color']}==2">
        green
    </c:when>
    <c:when test="${param['color']}==3">
        blue
    </c:when>
    <c:otherwise>
        yellow
    </c:otherwise>
</c:choose>
"

```

3. Iteraciones

Hay dos estructuras disponibles para construir bucles. La naturaleza de los datos en los que tiene que realizarse la iteración determina el tipo de operador que se usará. La etiqueta `<c:foreach>` se especializa en el recorrido de una colección, una lista o una tabla. La etiqueta `<c:forTokens>` se especializa en el recorrido de una cadena de caracteres.

a. Etiqueta `<c:forEach>`

Esta etiqueta ofrece dos modos de funcionamiento:

- Recorrido de una lista de elementos de tipo tabla o colección.
- Bucle clásico que incrementa un contador en cada iteración.

Los atributos de la etiqueta determinan el modo de funcionamiento. Si el atributo `items` está presente, el bucle recorrerá el contenido del elemento indicado por esta propiedad. Si el atributo `items` no aparece en la etiqueta, ésta obligatoriamente tendrá los atributos `begin` y `end` para fijar los extremos de inicio y final de la iteración. Por defecto, el paso de iteración es de uno salvo que el atributo `step` indique un valor distinto. Si estos dos atributos se usan conjuntamente con el atributo `items`, determinan el índice de inicio y de final de las iteraciones en el contenido del elemento `items`.

Sea cual sea el caso, una variable declarada con el atributo `var` se usa para referenciar el elemento actual o el valor actual del contador.

También es posible declarar una variable que permita seguir la evolución del bucle. El nombre de esta variable se define con el atributo `varStatus`. Esta variable ofrece las propiedades siguientes para seguir la evolución del bucle.

`index`: representa el índice actual del recorrido de la colección.

`current`: representa el objeto actual de la iteración.

`count`: indica el número de pasada en el bucle.

`first`: booleano que indica si se trata de la primera pasada del bucle.

`last`: booleano que indica si se trata de la última pasada del bucle.

```

<!-- bucle de recorrido de cabeceras -->
<!-- y visualizacion de los apellidos en la cabecera -->
<c:forEach var='cabecera' items='${headerValues}' varStatus="estado">
    <!-- visualizacion del numero de la cabecera y de los apellidos -->
    <c:out value='${estado.count}'/>
    <c:out value='${cabecera.key}'/> =
    <!-- bucle de recorrido de valores de una cabecera -->
    <!-- y visualizacion de estos valores -->
    <c:forEach var='valor' items='${cabecera.value}' varStatus="b2">
        <c:out value='${valor}' escapeXml="false"/><br>
    </c:forEach>
</c:forEach><br>

```

b. Etiqueta <c:forTokens>

Esta etiqueta permite reemplazar el método `split` de la clase `String` debido a que realiza el corte de una cadena de caracteres en función de uno o varios caracteres de delimitación. El atributo `items` contiene la cadena en la que se realizará el corte. Los caracteres de delimitación se indican con el atributo `delims`. Las diferentes palabras que se encuentren en la cadena de caracteres están disponibles sucesivamente mediante el atributo `var`. Como en el caso de la etiqueta `<c:forEach>`, los atributos `begin`, `end` y `step` permiten especificar la posición de inicio, la posición final y el incremento en cada "paso" respectivamente.

```
<c:forTokens items="${pageContext.request.queryString}"  
delims=";" var="palabra" varStatus="estado">  
    <c:if test="${estado.first}">  
        <table border="1">  
            <tr>  
                <td> numero </td>  
                <td>parametro</td>  
            </tr>  
    </c:if>  
    <tr>  
        <td>  
            <c:out value="${estado.count}" />  
        </td>  
        <td>  
            <c:out value="${palabra}" />  
        </td>  
    </tr>  
    <c:if test="${estado.last}">  
        </table>  
    </c:if>  
</c:forTokens>
```

4. Manipulación de URL

Hay tres etiquetas disponibles para manipular URL:

- La etiqueta `<c:import>` se usa para insertar un recurso.
- La etiqueta `<c:redirect>` se usa para generar una respuesta de redirección.
- La etiqueta `<c:url>` se usa para generar una URL con un identificador de sesión.

a. Etiqueta <c:import>

Esta etiqueta permite la inserción de un recurso en el interior de la página JSP. Es, sin embargo, más flexible que la etiqueta `<jsp:include>` dado que el recurso que se incluirá no es necesariamente un recurso de la aplicación. El nombre del recurso se proporciona con el atributo `url`. Este atributo puede corresponder a una URL absoluta que identifica un recurso alojado en otro servidor. Así como puede corresponder a una URL relativa alojada en el mismo servidor. Si este recurso pertenece a otra aplicación alojada en este servidor, el atributo `context` indica el nombre de la aplicación en el servidor. En estos dos atributos las URL tienen que empezar obligatoriamente con el carácter `/`.

Por defecto, el contenido del recurso se inserta en la página JSP. Si los atributos `session` y `scope` se especifican en la etiqueta, el contenido del recurso se transfiere a la variable cuyo nombre y ámbito han sido indicados por estos dos atributos.

El siguiente ejemplo recupera en una variable el código HTML generado por un servlet y muestra el código en la página JSP.

```
<c:import url="/PrimerServlet" context="/servlet01"  
var="codigoHtml" scope="page"/>  
<c:out value="${codigoHtml}" escapeXml="true" />
```

Si desea importar un recurso en el mismo servidor pero ubicado en otra aplicación, hay que añadir en el archivo

context.xml el atributo `crossContext="true"` en la etiqueta context.

b. Etiqueta <c:redirect>

Esta etiqueta permite enviar una respuesta de redirección al cliente. El atributo `url` representa una URL relativa o absoluta hacia el recurso al que se debe realizar la redirección. Una URL relativa se expresa en relación al contexto de la aplicación actual. Para contactar con un recurso disponible en el mismo servidor pero de otra aplicación, hay que añadir el atributo `context` a esta etiqueta para identificar el nombre de la aplicación en la que se encuentra el recurso. El valor de este atributo siempre comienza con el carácter `/`.

```
<c:redirect url="http://www.google.es" />
```

c. Etiqueta <c:url>

El objetivo principal de esta etiqueta es añadir en las URL presentes en una página JSP la información necesaria para asegurar el seguimiento de sesión en el caso de que el navegador cliente no acepte cookies.

La URL reescrita puede registrarse en la variable indicada por el atributo `var` almacenada en el ámbito indicado por el atributo `context`. Si la reescritura de URL no es necesaria, es debido a uno de los motivos siguientes:

- El cliente acepta cookies.
- El recurso se encuentra en otra aplicación.
- El recurso se encuentra en otro servidor.

la etiqueta <c:url> no realiza ningún tratamiento.

```
<a href=<c:url value='/visualizarCliente.jsp' />>  
    resultado de la búsqueda  
</a>
```

Código HTML correspondiente:

```
<a  
href="/jstl/visualizarCliente.jsp;jsessionid=22456B5750E4868D7EE259027A1E9F72" >  
    resultado de la búsqueda  
</a>
```

d. Etiqueta <c:param>

Esta etiqueta permite añadir un parámetro a una URL. Se puede utilizar como etiqueta hija de todas las etiquetas anteriores para los casos en que la URL necesite el paso de uno o varios parámetros. Esta etiqueta requiere que se informen el atributo `name` que representa el nombre del parámetro insertado en la URL y el atributo `value` que representa el valor del mismo.

El ejemplo siguiente realiza una redirección hacia el motor de búsqueda Google pasando un parámetro llamado `q`. El valor de este parámetro se obtiene a partir del parámetro llamado `query` recibido en la petición HTTP original.

```
<c:redirect url="http://www.google.es/search" >  
    <c:param name="q" value="${param['query']}" />  
</c:redirect>
```

La librería XML

Esta librería se especializa en la manipulación de documentos XML. Propone un conjunto de etiquetas sustancialmente idénticas a las disponibles en la librería estándar debido al hecho de que ambas trabajan con contenidos XML. Utiliza el lenguaje XPath para identificar los elementos del documento XML en los que se van a realizar los tratamientos. Los ejemplos siguientes utilizarán el documento mostrado a continuación como origen XML.

```
<?xml version="1.0" encoding="utf-8"?>
<restaurante xmlns:resto="http://www.ejemploXml.es/restaurante">
    <menu tipo="gastronomico">
        <entrantes>
            <nombre calorias="50">tomate al horno</nombre>
            <nombre calorias="300">ensalada verde</nombre>
            <nombre calorias="350">ensalada de pasta</nombre>
        </entrantes>
        <platos>
            <nombre calorias="1000">lenguado a la plancha</nombre>
            <nombre calorias="2000">escalope a la milanesa</nombre>
            <nombre calorias="1700">fabada asturiana</nombre>
        </platos>
        <embutidos>
            <nombre calorias="240">jamón serrano</nombre>
            <nombre calorias="300">chorizo</nombre>
            <nombre calorias="120">jamón york</nombre>
        </embutidos>
        <postres>
            <nombre calorias="340" sabor="chocolate">helado</nombre>
            <nombre calorias="250" fruta="manzana">tarta</nombre>
            <nombre calorias="400">crema catalana</nombre>
        </postres>
    </menu>
    <menu tipo="economico">
        <entrantes>
            <nombre calorias="50">pan</nombre>
        </entrantes>
        <platos>
            <nombre calorias="1700">lomo adobado</nombre>
        </platos>
        <embutidos>
            <nombre calorias="240">fuet</nombre>
        </embutidos>
        <postres>
            <nombre calorias="340" sabor="vainilla">helado</nombre>
        </postres>
    </menu>
</restaurante>
```

1. Etiquetas básicas

Las tres etiquetas siguientes son esenciales para la manipulación de un documento XML.

La etiqueta `<x:parse>` realiza el trabajo más importante ya que es la que analiza el documento XML.

La etiqueta `<x:set>` permite declarar una variable que contenga un extracto de un documento XML.

La etiqueta `<x:out>` se utiliza para insertar una porción de documento XML en la página JSP.

a. Etiqueta `<x:parse>`

Esta etiqueta es esencial para la manipulación de un documento XML ya que es la responsable de la transformación del documento XML en el objeto que contiene una lista de nodos en forma de árbol. El resto de etiquetas trabajarán con esta lista.

Primeramente, hay que indicar con el atributo `doc` de esta etiqueta el contenido XML que se debe analizar. Si el documento que se analizará se encuentra en un archivo, previamente hay que transferir el contenido de este documento a una variable que será utilizada por la etiqueta `<x:parse>`. La etiqueta `<c:import>` se encarga de esta

operación.

```
<c:import url="/Restaurante.xml" var="menus" />
```

El resultado del análisis se almacena en una variable indicada por el atributo `var` en el ámbito especificado por el atributo `scope`.

```
<x:parse doc="${menus}" var="arbolMenus" scope="page">
```

b. Etiqueta <x:set>

Esta etiqueta tiene la misma función que la etiqueta `<c:set>` de la librería básica. Permite la creación y la modificación de una variable. El nombre de la variable se indica con el atributo `var` y la ubicación de la variable se indica con el atributo `scope`. El contenido de la variable se genera en función de la evaluación de la expresión XPath asociada al atributo `select`. El primer elemento de la expresión XPath corresponde al nombre de la variable que contiene el documento XML que se tratará. Por supuesto que esta variable tiene que haber sido inicializada con una etiqueta `<c:parse>`.

```
<c:import url="/Restaurante.xml" var="menus" />
<x:parse doc="${menus}" var="arbolMenus" scope="page"/>
<x:set var="platos"
select="$arbolMenus/restaurante/menu[@tipo='economico']" />
<x:out select="$platos" />
```

c. Etiqueta <x:out>

Esta etiqueta es la correspondencia exacta de la etiqueta `<c:out>` con la excepción de que trabaja más con una expresión XPath que con una variable. Envía a la página JSP el resultado de la evaluación de la expresión XPath.

```
<x:out select="$arbolMenus/restaurante/menu[@tipo='gastronomico']" />
```

2. Etiquetas condicionales

Encontramos, una vez más, en esta librería dos etiquetas que permiten reemplazar los operadores `if` y `switch` del lenguaje Java.

a. Etiqueta <x:if>

Esta etiqueta utiliza en su atributo `select` una expresión XPath que puede evaluarse como un booleano. Si esta expresión se evalúa a `true` el cuerpo de la etiqueta se inserta en la página JSP.

```
<x:if select="$arbolMenus/restaurante/menu/platos/nombre[@calorias]>'500'" >
    <h1 style="color:red"> hipercalórico </h1>
</x:if>
```

b. Etiqueta <x:choose>

Esta etiqueta tiene exactamente la misma estructura que la etiqueta `<c:choose>`. Las diferentes condiciones se indican en etiquetas `<x:when>` o en una etiqueta `<x:otherwise>`. Para la etiqueta `<x:when>` el atributo `select` contiene una expresión XPath que puede evaluarse como un booleano determinando si el cuerpo de la etiqueta tiene que insertarse o no en la página JSP.

```
<x:choose>
    <x:when
select="$arbolMenus/restaurante/menu/platos/nombre[@calorias<='1000']">
        <h1 style="color:green"> bueno para conservar la línea </h1>
    </x:when>
    <x:when
select="$arbolMenus/restaurante/menu/platos/nombre[@calorias<='1700']">
        <h1 style="color:yellow"> normal </h1>
```

```

        </x:when>
        <x:when
select="$arbolMenus/restaurante/menu/platos/nombre[@calorias>'1700']">
            <h1 style="color:red"> hipercalórico </h1>
        </x:when>
</x:choose>

```

3. Etiqueta de iteración

La etiqueta `<x:forEach>` permite realizar una iteración sobre varios elementos de un documento XML. Los elementos tratados se extraen del documento a partir de la expresión XPath especificada en el atributo `select` de esta etiqueta. El atributo `var` permite acceder al cuerpo del bucle para cada elemento que forma parte de la iteración.

El ejemplo siguiente muestra en una tabla la lista de entradas de todos los menús. Cada elemento se muestra en un color distinto en función del número de calorías.

```





```

La librería de internacionalización y de formato

El objetivo de esta librería es el de simplificar la vida del diseñador de aplicaciones Web cuando éstas tienen que gestionar varias versiones en función del idioma favorito del cliente. Además, esta biblioteca gestiona los formatos utilizados para la visualización de fechas y valores numéricos.

1. Internacionalización de una aplicación

Para que una aplicación pueda adaptarse al idioma favorito del cliente es evidente que hay que disponer de los recursos que se tienen que mostrar al usuario en varios idiomas. La primera solución consiste en gestionar varias versiones de cada recurso cada uno en un idioma distinto. Esta solución es muy costosa de implementar debido a que para cada recurso hay que pensar en una versión diferente para cada uno de los idiomas gestionados por la aplicación. Las cosas son aún peor durante el mantenimiento de la aplicación ya que una simple modificación en una página JSP debe trasladarse a cada una de las versiones de la página. La solución propuesta para simplificar las cosas consiste en gestionar una sola versión de cada recurso y colocar los datos variables en función del idioma en uno o varios archivos externos. El recurso contiene las instrucciones para realizar la búsqueda de estos datos en el archivo adecuado en función del idioma favorito del cliente.

a. Definición de recursos

Todos los recursos del tipo cadena de caracteres tienen que colocarse en uno o varios archivos externos. Estos archivos contienen pares clave/valor. Las claves se usan en las páginas JSP para obtener el valor correspondiente del archivo.

Los nombres de los archivos que contienen los recursos tienen que respetar el convenio siguiente. El comienzo del nombre del archivo es completamente libre. Tiene que seguirse de un indicador del idioma para el que se ha creado este archivo de recursos. Hay que usar los códigos de país tal y como están definidos para los navegadores en la cabecera de la petición HTTP `accept language`. Finalmente, este archivo tiene que tener la extensión `.properties`.

En una aplicación se puede encontrar por tanto los archivos siguientes:

recursos_es.properties: para la versión española de las cadenas de caracteres.

recursos_en.properties: para la versión inglesa de las cadenas de caracteres.

recursos.properties: para la versión por defecto de las cadenas de caracteres. Esta versión se utiliza si no existe una versión específica de los recursos para el idioma preferido del cliente.

Obviamente, los archivos tienen que contener las mismas claves.

b. Uso de los recursos

Los archivos de recursos creados de este modo tienen que estar referenciados en las páginas JSP por el uso de las etiquetas `<fmt:setBundle>` o `<fmt:bundle>`. Estas dos etiquetas utilizan el atributo `basename` para identificar el archivo que contiene los recursos. La etiqueta `<fmt:setBundle>` acepta opcionalmente los atributos `var` y `scope` que permiten asignar a una variable el archivo que contiene los recursos. Esta técnica es útil cuando se usan varios archivos de recursos en la misma página JSP.

El o los archivos de recursos referenciados con la etiqueta `<fmt:setBundle>` pueden usarse en el conjunto de la página JSP. La etiqueta `<fmt:bundle>` es más restrictiva debido a que los recursos referenciados no son accesibles desde el contenido de la etiqueta.

```
<fmt:setBundle basename="cuenta.recursos" />
```

Las cadenas de caracteres que están en los archivos de recursos se incorporan en la página JSP mediante la etiqueta `<fmt:message>`. Esta etiqueta utiliza el atributo `key` para identificar la cadena de caracteres del archivo de recursos por defecto. Es esta cadena de caracteres la que debe insertarse en la página JSP en el lugar de la etiqueta `<fmt:message>`. Para acceder a un recurso en otro archivo hay que añadir a esta etiqueta el atributo `bundle` para identificar el archivo de donde se extraen los recursos. Este archivo tiene que estar previamente referenciado por la etiqueta `<fmt:setBundle>` que contiene el atributo `var` para identificar este archivo.

El extracto de página JSP mostrado a continuación contiene un formulario HTML para el que las etiquetas y el texto de los botones se han extraído de archivos de recursos.

```
<fmt:setBundle basename="recursos" />
<fmt:message key="txtLogin"/>
<table border="0" cellspacing="10" cellpadding="10">
```

```

<tbody>
    <tr>
        <td><fmt:message key="txtNombre"/></td>
        <td><input type="text" name="txtNombre" value="" /></td>
    </tr>
    <tr>
        <td><fmt:message key="txtPassword"/></td>
        <td><input type="password" name="txtPassword" value="" /></td>
    </tr>
    <tr>
        <td><input type="submit" value=<fmt:message
key="btnAceptar"/>" /></td>
        <td><input type="reset" value=<fmt:message
key="btnCancelar"/>" /></td>
    </tr>
</tbody>
</table>

```

La versión española del archivo de recursos: recursos_es.properties

```

txtLogin=Identificación
txtNombre=Nombre
txtPassword=Contraseña
btnAceptar=Aceptar
btnCancelar=Cancelar

```

La versión inglesa del archivo de recursos: recursos_en.properties

```

txtLogin=login
txtNombre=name
txtPassword=password
btnAceptar=OK
btnCancelar=cancel

```

2. Formato de valores numéricos y de fechas

Es frecuente tener que mostrar en una página JSP fechas y horas o valores numéricos. Para una mejor legibilidad, es importante que estos datos se presenten al usuario en un formato con el que esté familiarizado. Las etiquetas de formato permiten adaptar automáticamente el formato de visualización en función de los parámetros regionales de cada usuario.

a. Definición de la franja horaria que se usará

La etiqueta `<fmt:setTimeZone>` permite establecer la franja horaria utilizada para el formato de las horas realizado por la etiqueta `<fmt:formatDate>`. La franja horaria que se utilizará puede configurarse para toda la página o para ser usada caso por caso en cada etiqueta de formato.

El atributo `value` permite especificar la franja horaria que debe usarse. Tiene el formato siguiente:

`GMT +/- valor numérico.`

El valor numérico indica el desfase en relación al meridiano de Greenwich.

Si la franja horaria tiene que usarse caso por caso, hay que utilizar los atributos `var` y `scope` para indicar la variable que contiene la franja horaria. El ejemplo siguiente define la franja horaria por defecto para la página y una franja horaria específica almacenada en la variable `tzMoscu`.

```

<fmt:setTimeZone value="GMT+1"/>
<fmt:setTimeZone value="GMT+3" var="tzMoscu" scope="page" />

```

b. Dar formato a una fecha y a una hora

La etiqueta `<fmt:formatDate>` realiza el formato de una fecha y/o de una hora. Muchos atributos permiten personalizar el funcionamiento de la etiqueta.

El atributo `value` representa el objeto `Date` a partir del cual el formato se realiza.

Mediante el atributo `type` se indica a lo que se desea realizar el formato. Para este atributo están disponibles los tres valores siguientes:

- `date`: para mostrar únicamente la parte de la fecha de este objeto.
- `time`: para mostrar únicamente la parte de la hora de este objeto.
- `both`: para mostrar la fecha y la hora de este objeto

Los atributos `dateStyle` y `timeStyle` indican el formato utilizado para la fecha y la hora. Estos atributos aceptan los valores siguientes:

- `default`
- `short`
- `medium`
- `long`
- `full`

El atributo `pattern` se utiliza cuando se desea obtener un formato de visualización personalizado. Este atributo tiene como valor una cadena de caracteres que sigue el mismo convenio que el usado en la clase `SimpleDateFormat`.

- `y` representa el año
- `M` representa el mes
- `w` representa el número de semana del año
- `D` representa el número de día del año
- `d` representa el número de día del mes
- `E` representa el día de la semana
- `H` representa la hora de 0 a 23
- `K` representa la hora de 0 a 11
- `a` representa el indicador am/pm
- `m` representa los minutos
- `s` representa los segundos
- `z` o `Z` representa el nombre de la franja horaria

El atributo `TimeZone` indica qué franja horaria debe usarse para dar el formato a la fecha/hora. Esta franja horaria tiene que haber sido previamente definida por una etiqueta `<fmt:setTimeZone>`.

El ejemplo siguiente presenta diferentes tipo de formatos y utiliza varias franjas horarias.

```
<fmt:setTimeZone value="GMT+1"/>
hora de Madrid en formato por defecto:
<fmt:formatDate value="${requestScope['dateValidation']}"
timeStyle="long" type="both"/>
```

```

<fmt:setTimeZone value="GMT+3" var="tzMoscu" scope="page"/>
<fmt:setTimeZone value="GMT-5" var="tzWashington" scope="page"/>
<br>
hora de Moscú en formato corto:
<fmt:formatDate value="${requestScope['dateValidation']}" 
dateStyle="short" timeStyle="short" type="both" timeZone="${tzMoscu}"/>
<br>
hora de Washington en formato medio:
<fmt:formatDate value="${requestScope['dateValidation']}" 
dateStyle="medium" timeStyle="medium" type="both" timeZone="${tzWashington}"/>
<br>
hora de Moscú en formato personalizado:
<fmt:formatDate value="${requestScope['dateValidation']}" 
pattern="HH:mm dd-MMM-yyyy" type="both" timeZone="${tzMoscu}"/>

```

Resultado correspondiente:

```

hora de Madrid en formato por defecto: 21 dic. 2009 23:42:54 GMT+01:00
hora de Moscú en formato corto: 22/12/09 01:42
hora de Washington en formato medio: 21 dic. 2009 17:42:54
hora de Moscú en formato personalizado: 01:42 22-dic-2009

```

c. Dar formato a valores numéricos

La etiqueta <fmt:formatNumber> permite dar formato a valores numéricos según tres estilos.

- Formato monetario.
- Formato de porcentaje.
- Formato de valores numéricos normales.

En esta etiqueta el atributo `style` determina el tipo de formato realizado. Los valores `currency`, `percent` y `number` corresponden a los tres estilos anteriores.

El formato se aplica al valor indicado por el atributo `value` o al valor contenido en el cuerpo de la etiqueta si el atributo `value` no se informa.

Para dar formato a valores monetarios, los atributos `currencyCode` o `currencySymbol` indican la característica o características utilizadas como símbolo monetario. El atributo `currencyCode` tiene que usar uno de los valores definidos en la norma ISO 4217.

El formato de la parte numérica se puede configurar con los atributos siguientes:

`maxIntegerDigits`: número máximo de cifras para la parte entera del número.

`minIntegerDigits`: número mínimo de cifras para la parte entera de número. Se pueden añadir ceros para obtener este valor mínimo de cifras en la visualización si el número es demasiado pequeño.

`maxFractionDigits`: número máximo de cifras para la parte fraccionaria del número.

`minFractionDigits`: número mínimo de cifras para la parte fraccionaria de número. Se pueden añadir ceros para completar.

`groupingUsed`: indica si los caracteres de separación de millares se añaden para la representación de números grandes.

Se puede especificar un formato personalizado con el atributo `pattern`. Este atributo tiene que respetar el mismo convenio que la clase `DecimalFormat`.

El ejemplo siguiente ilustra los principales usos de esta etiqueta.

```

total del pedido:
<fmt:formatNumber value="1256.5" currencySymbol="€"
type="currency" minFractionDigits="2" />
<br>
descuento:
<fmt:formatNumber value="0.085" type="percent"
minIntegerDigits="2" minFractionDigits="2" />
<br>

```

```
peso del paquete:  
<fmt:formatNumber value="18645" type="number" groupingUsed="true"/>  
gramos  
<br>
```

Resultado correspondiente:

```
total del pedido: 1 256,50 €  
descuento: 08,50%  
peso del paquete: 18 645 gramos
```

Librería de acceso a bases de datos

Las etiquetas de esta librería permiten fácilmente realizar operaciones elementales en una base de datos. Esta librería no tiene que usarse para el diseño completo de una aplicación, para lo cual es evidentemente preferible y más eficaz usar JDBC. Para poder funcionar, esta librería necesita tener una conexión al servidor de base de datos. Esta conexión se puede obtener por el servidor de aplicaciones mediante un objeto `DataSource` o configurada directamente en la página JSP.

1. Configurar una conexión

La etiqueta `<sql:setDataSource>` permite configurar para la página JSP el medio para obtener una conexión con la base de datos. Si el servidor dispone de un objeto `DataSource`, es mejor usar éste último.

La etiqueta `<sql:setDataSource>` toma el aspecto siguiente:

```
<sql:setDataSource dataSource="jdbc/northwind" scope="page"/>
```

Esta etiqueta necesita la declaración de un parámetro en el descriptor de despliegue de la aplicación.

```
<context-param>
    <param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
    <param-value>jdbc/northwind</param-value>
</context-param>
```

El valor de este parámetro tiene que corresponder al nombre JNDI utilizado en el servidor para obtener el objeto `DataSource`.

Si el servidor no dispone de ningún objeto `DataSource`, entonces hay que configurar la etiqueta `<sql:setDataSource>` con todos los parámetros necesarios para establecer la conexión con el servidor de base de datos. Los datos que se tienen que proporcionar mediante atributos son los siguientes:

`driver`: nombre de la clase correspondiente al driver que se usará.

`url`: cadena de caracteres que contiene los datos indispensables para la conexión. El formato de esta cadena depende de la base de datos.

`user`: nombre de la cuenta utilizada para establecer la conexión.

`password`: contraseña utilizada para establecer la conexión.

La etiqueta `<sql:setDataSource>` toma el aspecto siguiente:

```
<sql:setDataSource var="ds" scope="page"
driver="com.microsoft.sqlserver.jdbc.SQLServerDriver"
url="jdbc:sqlserver://localhost:1433;databaseName=northwind" user="sa"/>
```

Con esta solución los datos de conexión están presentes directamente en las páginas JSP y será necesario intervenir a nivel de estas páginas en caso de que sea necesaria una modificación de estos parámetros.

El objeto `DataSource` obtenido por esta etiqueta también puede almacenarse en una variable para ser usado en el resto de etiquetas de esta librería. La variable se define con los atributos `var` y `scope`.

2. Ejecución de una instrucción select

La etiqueta `<sql: query>` ejecuta una instrucción `select` en una base de datos. Los datos se devuelven en un objeto de tipo `javax.servlet.jsp.jstl.sql.Result` cuya función es extremadamente similar a la del objeto `ResultSet` de JDBC.

Los atributos siguientes se utilizan para configurar esta etiqueta:

`sql`: contiene la instrucción `select` que se ejecutará.

`maxRows`: determina el número máximo de filas devueltas por la ejecución de la instrucción `select`.

`startRow`: indica el índice de la primera fila del resultado devuelto. El valor por defecto de este atributo es igual a cero, correspondiente a la primera fila.

`var`: nombre de la variable que contendrá los resultados.

`scope`: ámbito de la variable que contendrá los resultados.

El ejemplo siguiente permite obtener las diez primeras filas de la tabla orders.

```
<sql:query maxRows="10" var="filas" sql="select * from orders"></sql:query>
```

3. Utilizar los resultados

El objeto `javax.servlet.jsp.jstl.sql.Result` tiene las siguientes propiedades para obtener los datos devueltos por la ejecución de la instrucción select.

`columnNames`: tabla de cadenas de caracteres que contiene los nombres de las columnas de la petición.

`rowCount`: número de líneas reenviadas por la petición select.

`rows`: esta propiedad contiene una tabla de objetos `SortedMap` que contiene cada una de las filas del resultado. En cada objeto `SortedMap` los nombres de los campos en la base de datos se utilizan como claves. La selección se realiza con los nombres de los campos.

`rowsByIndex`: esta propiedad contiene una tabla en la que cada elemento contiene también una tabla. El primer nivel corresponde a fila del resultado y el segundo nivel contiene todos los valores de esa fila.

`limitedByMaxRows`: esta propiedad de tipo booleano permite saber si el número de filas del resultado se ha limitado o no por el atributo `maxRows` de la etiqueta `query`.

El ejemplo mostrado a continuación utiliza la propiedad `rows` para obtener la redacción de los campos y llenar las cabeceras de la tabla HTML. A continuación utiliza la propiedad `rowsByIndex` para obtener el valor de cada campo y completar la tabla HTML.

```
<sql:setDataSource dataSource="jdbc/northwind" scope="page"/>
<sql:query maxRows="10" var="filas" sql="select * from
orders"></sql:query>
<table border="1">
    <thead>
        <c:forEach var="n" items="${filas.columnNames}">
            <th>${n}</th>
        </c:forEach>
    </thead>
    <tbody>
        <c:forEach var="unaFila" items="${filas.rowsByIndex}" >
            <tr>
                <c:forEach var="unValor" items="${unaFila}">
                    <td>
                        ${unValor}
                    </td>
                </c:forEach>
            </tr>
        </c:forEach>

    </tbody>
</table>
```

En la ejecución obtenemos el siguiente resultado.

OrderID	CustomerID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	Freight	ShipName	ShipAddress	ShipCity	ShipRegion	ShipPostalCode	ShipCountry
10248	VINET	5	1996-07-04 00:00:00	1996-08-01 00:00:00	1996-07-16 00:00:00	3	32.3800	Vinos y cervezas Rodríguez	Orense, 41	Madrid		22044	Spain
10249	TOMSP	6	1996-07-05 00:00:00	1996-08-16 00:00:00	1996-07-10 00:00:00	1	11.6100	Toms Spezialitäten	Luisenstr. 48	Münster		44087	Germany
10251	VICTE	3	1996-07-08 00:00:00	1996-08-05 00:00:00	1996-07-15 00:00:00	1	41.3400	Viandas en stock	2, rue du Commerce	Lyon		69004	France
10252	SUPRD	4	1996-07-09 00:00:00	1996-08-06 00:00:00	1996-07-11 00:00:00	2	51.3000	Suprêmes délices	Boulevard Thiers, 255	Charleroi		B-6000	Belgium
10254	CHOPS	5	1996-07-11 00:00:00	1996-08-08 00:00:00	1996-07-23 00:00:00	2	22.9800	Chop-suey Chinese	Hauptstr. 31	Bern		3012	Switzerland
10255	RICSU	9	1996-07-12 00:00:00	1996-08-09 00:00:00	1996-07-15 00:00:00	0	143.3100	Rucher Supermarché	Starenweg 5	Genève		1204	Switzerland
10257	HILAA	4	1996-07-16 00:00:00	1996-08-13 00:00:00	1996-07-22 00:00:00	3	31.9100	HILARION- Abartes	Carrera 22 con Ave. Carlos Soublette #3-35	San Cristóbal	Táchira	5022	Venezuela
10258	ERNSH	1	1996-07-17 00:00:00	1996-08-14 00:00:00	1996-07-23 00:00:00	1	140.5100	Ernst Handel	Kirchgasse 6	Graz		8010	Austria
10259	CENTC	4	1996-07-18 00:00:00	1996-08-15 00:00:00	1996-07-25 00:00:00	3	32.2000	Centro comercial Macarons	Sierras de Granada 9993	Méjico D.F.		05022	Méjico
10260	OTTIK	4	1996-07-19 00:00:00	1996-08-16 00:00:00	1996-07-29 00:00:00	1	55.0900	Cemex Kaseladen	Mehrheimerstr. 369	Köln		50739	Germany

4. Ejecución de una instrucción SQL cualquiera

La etiqueta `<sql:update>` tiene que usarse para la ejecución de cualquier instrucción SQL a excepción de la instrucción select. Los atributos disponibles son idénticos a los de la etiqueta `<sql:query>` con la excepción de los que conciernen a la gestión del número de filas (`maxRows` y `startRows`) que no existen para esta etiqueta.

La variable configurable con el atributo `var`, contendrá para esta etiqueta el número de filas afectadas por la ejecución de la instrucción SQL.

El ejemplo siguiente modifica los gastos de transporte de los pedidos en España y muestra el número de pedidos modificados.

```
<sql:update var="resultado" sql="update orders set Freight=0
where ShipCountry='Spain'"></sql:update>
${resultado} pedidos han sido modificados
```

5. Utilización de parámetros en el código SQL

Con frecuencia se tiene que ejecutar varias veces una sentencia SQL con apenas una pequeña modificación entre dos ejecuciones. El ejemplo clásico corresponde a una sentencia de selección con una restricción.

```
select * from orders where customerId='VINET'
```

El valor que se pone en la condición generalmente lo introduce el usuario de la aplicación y en este caso está disponible en un parámetro de la petición HTTP. La primera solución que viene en mente consiste en construir una query SQL concatenando cadenas de caracteres.

```
<sql:query var="filas" sql="select * from orders where
customerId='${param['codigoCliente']}'"></sql:query>
```

Con esta solución hay que ir con mucho cuidado en el uso de los caracteres ' y ". Hay que ser consciente que después del tratamiento de la página JSP por parte del servidor tiene que quedar una instrucción SQL correcta. Si se usan muchos parámetros, la sintaxis rápidamente se volverá compleja. Para limitar los riesgos, la librería SQL autoriza el uso de parámetros en las instrucciones SQL. La ubicación de estos parámetros tiene que reservarse con el carácter ? en la instrucción SQL. A continuación hay que proporcionar con etiquetas `<sql:param>` los valores que se usarán reemplazando estos diferentes parámetros. Las etiquetas `<sql:param>` deben usarse en el interior de la etiqueta `<sql:query>` o `<sql:update>`. El reemplazo se realiza en el orden de aparición de las etiquetas `<sql:param>`.

El ejemplo anterior toma con esta solución un aspecto más claro.

```
<sql:query var="filas" sql="select * from orders where customerId=?"
<sql:param value="${param['codigoCliente']}" />
</sql:query>
```

Presentación

El objetivo de las etiquetas personalizadas es el de mejorar la legibilidad del código en una página JSP, aumentar la productividad del desarrollador y facilitar el mantenimiento de la aplicación. El capítulo sobre la librería JSTL ya ha demostrado estas ventajas cuando se necesita ejecutar acciones estándar en una página JSP. Lamentablemente, esta librería no contiene etiquetas correspondientes a todas las necesidades que se pueden encontrar en la creación de páginas JSP. Con frecuencia, se tiene la obligación de añadir porciones de código Java en forma de scriptlet en las páginas JSP. Cuando debe realizarse el mismo tratamiento en varias páginas JSP, hay que duplicar el código en cada una de las páginas JSP. El objetivo de las etiquetas personalizadas es el de externalizar este código en archivos independientes. A continuación, este código se puede reutilizar en el interior de las distintas páginas JSP de la aplicación, o incluso utilizarse en varias aplicaciones. Es por este principio que funciona la librería JSTL. Hasta la aparición de la versión 2.0 de la api JSP, esta solución era compleja de implementar y necesitaba mucho tiempo de desarrollo. La creación de etiquetas personalizadas ahora se ha simplificado enormemente desde que es posible crear etiquetas personalizadas sin incluso escribir una sola línea de código Java. Las etiquetas personalizadas pueden crearse en forma de archivo de etiquetas (tag file) o de clase Java.

Los archivos de etiquetas

Los archivos de etiquetas son simples archivos de texto que contienen un extracto de código JSP que puede ser reutilizado como una etiqueta normal. Como una etiqueta normal, estas porciones de código se traducen en Java cuando el servidor analiza la página. Por contra, el código Java generado por la traducción de estas etiquetas no se incluye directamente en el código de la página JSP. De hecho, el servidor crea una nueva clase que hereda de la clase `javax.servlet.jsp.tagext.SimpleTagSupport` que contiene la traducción del contenido de la etiqueta. Cuando esta etiqueta se encuentra en una página JSP mientras el servidor la analiza, éste reemplaza la etiqueta por la creación de una llamada a una función. Esta función crea una instancia de la clase correspondiente a la etiqueta y a continuación invoca el método `doTag` de esta instancia para ejecutar el contenido de la etiqueta. Para esclarecer todo esto, vamos a escribir nuestra primera etiqueta personalizada.

1. Creación de un archivo de etiquetas

Una etiqueta tiene que crearse en un simple archivo de texto con la extensión `.tag`. Todos los archivos de etiquetas tienen que colocarse en el directorio `WEB-INF/tags` de la aplicación. Si se utilizan varias etiquetas personalizadas en la aplicación, éstas pueden estar agrupadas en un archivo (`.jar`). Este archivo tiene que estar en el directorio `WEB-INF/lib` de la aplicación. En el caso de ejemplo, es obligatorio otro archivo de descripción de las etiquetas personalizadas (`.tld`). Veremos su sintaxis un poco más adelante en este capítulo.

Nuestra primera etiqueta va a ser muy simple ya que simplemente inserta en la página JSP la fecha y hora actuales. A continuación se muestra el archivo correspondiente:

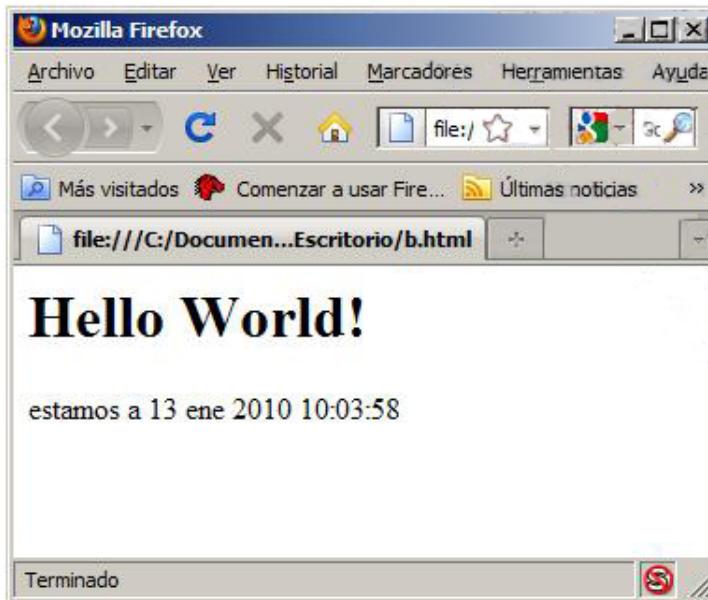
```
<%
    java.util.Date d;
    d=new java.util.Date();
%>
estamos a <%=d.toLocaleString()%>
```

La utilización de esta nueva etiqueta es similar al uso de las etiquetas disponibles en la librería JSTL. En primer lugar es necesario declarar que se desea usar una librería externa de etiquetas añadiendo la directiva `taglib` al comienzo de la página JSP. A continuación, las etiquetas de esta librería son accesibles utilizando el prefijo asociado a la librería de donde éstas provengan.

El código de la página JSP utilizando nuestra etiqueta nueva:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">
<%@taglib prefix="perso" tagdir="/WEB-INF/tags" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <perso:fechaYHora/>
    </body>
</html>
```

En la ejecución obtendremos el resultado siguiente:



Como en el caso de una página JSP normal, el navegador sólo recibe código HTML.

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

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Hello World!</h1>

    estamos a 13 ene 2010 10:03:58

  </body>
</html>
```

Por contra, las cosas se complican cuando se examina el código generado por el servidor en el análisis de la página JSP. Se generan los dos archivos siguientes. El primero corresponde a la traducción de la etiqueta personalizada y el segundo corresponde a la traducción de la página JSP.

El archivo fechaYHora_tag.java

```
package org.apache.jsp.tag.web;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class fechaYHora_tag
    extends javax.servlet.jsp.tagext.SimpleTagSupport
    implements org.apache.jasper.runtime.JspSourceDependent {

    private static final JspFactory _jspxFactory =
        JspFactory.getDefaultFactory();

    private static java.util.List _jspx_dependants;

    private JspContext jspContext;
    private java.io.Writer _jspx_sout;
    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.AnnotationProcessor _jsp_annotationprocessor;
```

```

public void setJspContext(JspContext ctx) {
    super.setJspContext(ctx);
    java.util.ArrayList _jspx_nested = null;
    java.util.ArrayList _jspx_at_begin = null;
    java.util.ArrayList _jspx_at_end = null;
    this.jspContext = new
org.apache.jasper.runtime.JspContextWrapper(ctx, _jspx_nested,
_jspx_at_begin, _jspx_at_end, null);
}

public JspContext getJspContext() {
    return this.jspContext;
}

public Object getDependants() {
    return _jspx_dependants;
}

private void _jspInit(ServletConfig config) {
    _el_expressionfactory =
_jspxFactory.getJspApplicationContext(config.getServletContext()).
getExpressionFactory();
    _jsp_annotationprocessor = (org.apache.AnnotationProcessor)
config.getServletContext().getAttribute(org.apache.AnnotationProces
sor.class.getName());
}

public void _jspDestroy() {
}

public void doTag() throws JspException, java.io.IOException {
    PageContext _jspx_page_context = (PageContext)jspContext;
    HttpServletRequest request = (HttpServletRequest)
_jspx_page_context.getRequest();
    HttpServletResponse response = (HttpServletResponse)
_jspx_page_context.getResponse();
    HttpSession session = _jspx_page_context.getSession();
    ServletContext application = _jspx_page_context.getServletContext();
    ServletConfig config = _jspx_page_context.getServletConfig();
    JspWriter out = jspContext.getOut();
    _jspInit(config);
    jspContext.getELContext().putContext(JspContext.class,jspContext);

    try {
        out.write('\n');
        out.write('\n');

java.util.Date d;
d=new java.util.Date();

        out.write("\n");
        out.write("estamos a ");
        out.print(d.toLocaleString());
    } catch( Throwable t ) {
        if( t instanceof SkipPageException )
            throw (SkipPageException) t;
        if( t instanceof java.io.IOException )
            throw (java.io.IOException) t;
        if( t instanceof IllegalStateException )
            throw (IllegalStateException) t;
        if( t instanceof JspException )
            throw (JspException) t;
        throw new JspException(t);
    } finally {
        jspContext.getELContext().putContext(JspContext.class,super.getJsp
Context());
        ((org.apache.jasper.runtime.JspContextWrapper)
jspContext).syncEndTagFile();
    }
}

```

}

El archivo index_jsp.java

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    private static final JspFactory _jspxFactory =
JspFactory.getDefaultFactory();

    private static java.util.List _jspx_dependants;

    static {
        _jspx_dependants = new java.util.ArrayList(1);
        _jspx_dependants.add("/WEB-INF/tags/fechaYHora.tag");
    }

    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.AnnotationProcessor _jsp_annotationprocessor;

    public Object getDependants() {
        return _jspx_dependants;
    }

    public void _jspInit() {
        _el_expressionfactory =
_jspxFactory.getJspApplicationContext(getServletConfig()).getServle
tContext().getExpressionFactory();
        _jsp_annotationprocessor = (org.apache.AnnotationProcessor)
getServletConfig().getServletContext().getAttribute(org.apache.An
notationProcessor.class.getName());
    }

    public void _jspDestroy() {
    }

    public void _jspService(HttpServletRequest request,
HttpServletResponse response)
        throws java.io.IOException, ServletException {

        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;

        try {
            response.setContentType("text/html;charset=UTF-8");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;

            out.write("\n");
            out.write("\n");
        }
    }
}
```

```

out.write("\n");
out.write("<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN\n");
out.write("      \"http://www.w3.org/TR/html4/loose.dtd\">\n");
out.write("      \n");
out.write("<html>\n");
out.write("      <head>\n");
out.write("          <meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">\n");
out.write("      <title>JSP Page</title>\n");
out.write("      </head>\n");
out.write("      <body>\n");
out.write("          <h1>Hello World!</h1>\n");
out.write("      ");
if (_jspx_meth_perso_005ffechaYHora_005f0(_jspx_page_context))
    return;
out.write("\n");
out.write("      </body>\n");
out.write("</html>\n");
} catch (Throwable t) {
    if (!(t instanceof SkipPageException)) {
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try { out.clearBuffer(); } catch (java.io.IOException e) {}
        if (_jspx_page_context != null)
_jspx_page_context.handlePageException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}

private boolean
_jspx_meth_perso_005ffechaYHora_005f0(PageContext
_jspx_page_context)
    throws Throwable {
PageContext pageContext = _jspx_page_context;
JspWriter out = _jspx_page_context.getOut();
// perso:fechaYHora
org.apache.jsp.tag.web.fechaYHora_tag
_jspx_th_perso_005ffechaYHora_005f0 = new
org.apache.jsp.tag.web.fechaYHora_tag();
org.apache.jasper.runtime.AnnotationHelper.postConstruct(_jsp_annotationprocessor, _jspx_th_perso_005ffechaYHora_005f0);
_jspx_th_perso_005ffechaYHora_005f0.setJspContext(_jspx_page_context);
_jspx_th_perso_005ffechaYHora_005f0.doTag();
org.apache.jasper.runtime.AnnotationHelper.preDestroy(_jsp_annotationprocessor, _jspx_th_perso_005ffechaYHora_005f0);
return false;
}
}

```

2. Añadir atributos a una etiqueta personalizada

Las etiquetas personalizadas también pueden aceptar atributos. Éstos se usan para proporcionar datos que permiten modificar el comportamiento de la etiqueta. Los atributos tienen que declararse en el archivo de etiquetas con la directiva @attribute.

Esta directiva espera los atributos siguientes:

name: nombre del atributo añadido a la etiqueta personalizada.

required: booleano que indica si el atributo es obligatorio u opcional.

type: nombre completo de la clase del atributo. Por defecto, los atributos se consideran cadenas de caracteres.

Para ilustrar el uso de los atributos, vamos a crear una etiqueta personalizada que permita calcular una fecha de entrega en función del modo de entrega elegido por el cliente. Esta etiqueta exigirá la presencia de un atributo

llamado modoEntrega que le servirá para el cálculo de la fecha prevista de recepción de la mercancía. El contenido de este atributo se obtiene por un parámetro de la petición HTTP. En el ejemplo anterior hemos usado únicamente scriptlets en el archivo de etiquetas. Para éste, por el contrario, vamos a utilizar únicamente otras etiquetas JSP. Evidentemente se puede utilizar una mezcla de los dos y éste es el caso más común.

El archivo de etiquetas:

```
<%-- referencia a las librerías utilizadas --%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%-- declaración del atributo --%>
<%@attribute name="modoEntrega" required="true" %>
<%-- creación de un bean que contiene la fecha y hora --%>
<jsp:useBean id="ahora" class="java.util.Date" />
<%-- creación de un bean que contendrá la fecha de entrega --%>
<jsp:useBean id="fechaEntrega" class="java.util.Date" />
<%-- en función del valor del atributo modoEntrega
    el número de días de transporte se guarda en la variable duracion--%>
<c:choose>
    <c:when test="${modoEntrega == 'DHL'}">
        <c:set var="duracion" value="1" />
    </c:when>
    <c:when test="${modoEntrega == 'correo'}">
        <c:set var="duracion" value="3" />
    </c:when>
    <c:when test="${modoEntrega == 'transportista'}">
        <c:set var="duracion" value="5" />
    </c:when>
</c:choose>
<%-- cálculo de la fecha de entrega --%>
<jsp:setProperty name="fechaEntrega" property="time"
value="${ahora.time + (24*60*60*1000 * duracion)}" />
<%-- formato y visualización de la fecha de entrega --%>
<fmt:formatDate value="${fechaEntrega}" type="date" dateStyle="full"/>
```

El código de la página JSP:

```
<%@taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<%@ taglib prefix="dl" tagdir="/WEB-INF/tags" %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Confirmación del pedido</title>
    </head>
    <body>

        <h3>Gracias por su pedido</h3><br>
        <em>con el envío por ${param.modoEntrega}</em>,
        su pedido llegará el
        <dl:fecha_entrega modoEntrega="${param.modoEntrega}" />
    </body>
</html>
```

3. Variables en archivos de etiquetas

Acabamos de ver que con los atributos se puede hacer transitar datos desde la página JSP a la etiqueta personalizada. A veces es útil hacer la operación inversa. Por ejemplo cuando una etiqueta realiza un cálculo y el resultado de este cálculo tiene que usarse en varios sitios en la página JSP. En este caso, hay que memorizar el resultado del cálculo en una variable que se podrá utilizar varias veces en la página JSP. La variable tiene que declararse en el archivo de etiquetas con una directiva @variable. Esta directiva tiene varios atributos para definir las características de la variable.

name-given: este atributo especifica el nombre de la variable. Con este nombre la variable será referenciada en la página JSP.

name-from-attribute: este atributo permite especificar el nombre de otro atributo que contiene el nombre de la variable. Esta solución permite parametrizar el nombre de la variable cuando se usa la etiqueta en una página JSP.

alias: este atributo se utiliza en conjunción con el anterior. Permite especificar el nombre de una variable local a la etiqueta. El contenido de esta variable local se copia a continuación en la variable accesible desde la página JSP.

variable-class: indica el nombre completo de la clase de esta variable. Por defecto las variables son de tipo cadena de caracteres.

scope: este atributo indica cómo se inicializa el contenido de la variable. Son posibles los tres valores siguientes.

AT-BEGIN: el contenido se transfiere a la variable antes de la ejecución de cada fragmento de código alojado en la etiqueta y también al final de la ejecución de la etiqueta.

NESTED: el contenido inicial de la variable se guarda al comienzo de la ejecución de la etiqueta. Se transfiere a la variable antes de cada ejecución de fragmento de código contenido en la etiqueta y después el valor inicial se restaura al final de la ejecución de la etiqueta.

AT-END: el contenido se transfiere a la variable al final de la ejecución de la etiqueta.

El archivo de etiquetas y la página JSP mostrados a continuación permiten probar estos distintos valores.

El archivo de etiquetas

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@variable name-given="variable" scope="AT_END" %>
contenido de la variable al inicio de la etiqueta <i>${variable}</i><br>
modificación de la variable al interior de la etiqueta <br>
<c:set var="variable" value="nuevo valor desde la etiqueta
antes del doBody" />
<jsp:doBody/>
contenido de la variable después del doBody <i>${variable}</i><br>
<c:set var="variable" value="nuevo valor de la etiqueta desde la etiqueta
antes del final de la etiqueta" />
```

La página JSP de prueba

```
%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/HTML4/loose.dtd">
<%@taglib prefix="perso" tagdir="/WEB-INF/tags" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    modificación de la variable desde la página JSP antes de la etiqueta <br>
    <c:set var="variable" value="valor desde la página JSP"/>
    <perso:prueba>
      valor de la variable en el cuerpo de la etiqueta
    <i>${variable}</i> <br>
    </perso:prueba>
    valor de la variable desde la página JSP después de la
etiqueta<i> ${variable}</i>
  </body>
</html>
```

Las etiquetas diseñadas en Java

Esta segunda solución para crear etiquetas personalizadas consiste en crear una clase Java que represente la etiqueta. Esta clase evidentemente debe respetar ciertas restricciones. Éstas se definen en la interfaz `SimpleTag`. Las clases que corresponden a etiquetas tienen, por lo tanto que implementar esta interfaz. Otra solución puede ser la de crear una clase que herede de la clase `SimpleTagSupport` que ya implementa la interfaz `SimpleTag`. Todos los métodos de esta clase tienen acceso a las variables implícitas de la página JSP.

El método más importante de esta clase es el método `doTag` que se llama cuando la etiqueta se encuentra en una página JSP. Evidentemente este método se sobrecargará en la clase de la etiqueta.

Los archivos compilados de estas clases tienen que colocarse en el directorio `/WEB-INF/classes` de la aplicación. Si estas clases están disponibles en un empaquetado Java éste tiene que ponerse en el directorio `/WEB-INF/lib` de la aplicación.

Este tipo de etiquetas requiere la creación de un archivo de descripción (tag lib descriptor) colocado en el directorio `WEB-INF` de la aplicación.

1. Etiquetas simples sin cuerpo

Este tipo de etiquetas es el que tiene un desarrollo más sencillo debido a que basta simplemente con crear una clase que herede de la clase `SimpleTagSupport` y redefinir el método `doTag` en esta clase. Este método se invoca automáticamente cuando el servidor encuentra la etiqueta durante el análisis de la página JSP. Los objetos implícitos de la página JSP son accesibles gracias al método `getJspContext`.

El código de la etiqueta que visualiza la hora tiene el siguiente aspecto:

```
package pk1;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class MuestraHora-1 extends SimpleTagSupport
{
    public void doTag() throws JspException, IOException
    {
        getJspContext().getOut().write(new
SimpleDateFormat("dd/MM/yyyy").format(new Date()));

    }
}
```

Esta etiqueta requiere el archivo de declaración mostrado a continuación (archivo `MuestraHora-1.tld`):

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd">
    <tlib-version>2.1</tlib-version>
    <short-name>perso</short-name>
    <tag>
        <name>mostrarHora-1</name>
        <tag-class>pk1.MuestraHora-1</tag-class>
        <body-content>empty</body-content>
    </tag>
</taglib>
```

El uso de esta etiqueta en una página JSP es similar al uso de una etiqueta normal, simplemente hay que añadir la directiva `taglib` referenciando el archivo de descripción de etiquetas.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="perso" uri="/WEB-INF/MuestraHora-1.tld" %>
```

```

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

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <perso:mostrarHora/>
  </body>
</html>

```

Si la etiqueta acepta atributos, hay que definir en la clase de la etiqueta una variable para cada atributo así como un método setXXX que permite la asignación de un valor a esta variable. Este método se invoca cuando es necesario transferir el valor del atributo de la página JSP a la instancia de la clase que representa la etiqueta. Podemos por ejemplo añadir un atributo para elegir el formato de visualización de la hora. La etiqueta toma entonces el aspecto siguiente:

```

package pk1;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class MuestraHora-2 extends SimpleTagSupport
{
    String formato;
    public void setFormat(String formato)
    {
        this.format=formato;
    }
    public void doTag() throws JspException,IOException
    {
        getJspContext().getOut().write(new
SimpleDateFormat(formato).format(new Date()));
    }
}

```

Los atributos tienen que estar declarados también en el archivo de descripción de etiquetas.

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd">
    <tlib-version>2.1</tlib-version>
    <short-name>perso</short-name>
    <tag>
        <name>mostrarHora-2</name>
        <tag-class>pk1.MuestraHora-2</tag-class>
        <body-content>empty</body-content>
        <attribute>
            <name>formato</name>
            <required>true</required>
        </attribute>
    </tag>
</taglib>

```

La utilización de esta etiqueta requiere ahora la adición del atributo formato.

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="perso" uri="/WEB-INF/MuestraHora-2.tld" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/html4/loose.dtd">

```

```

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <perso:mostrarHora formato="dd/MM/yyyy"/>
    </body>
</html>

```

2. Etiquetas con cuerpo

Una etiqueta personalizada con cuerpo tiene que gestionarse de modo diferente según el uso que hace de los datos contenidos en el cuerpo de la etiqueta. Si la etiqueta no tiene necesidad de manipular los datos contenidos en el cuerpo, simplemente hay que hacer que los datos se transfieran a la página enviada al navegador. El método `getJspBody` permite obtener una referencia a un objeto `JspFragment` que representa el cuerpo de la etiqueta. El método `invoke` de este objeto `JspFragment` permite transferir su contenido a la respuesta HTTP. Este método espera como parámetro un `Writer` que asegure esta transferencia. Si ningún objeto `Writer` es dado, se utiliza el asociado a la página JSP. El método `invoke` llama por lo tanto en este caso al método `JspContext.getOut()` para obtener una referencia a este objeto.

Cuando una etiqueta puede tener cuerpo, hay que modificar el elemento `<body-content>` de esta etiqueta en el archivo de descripción.

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-
jsptaglibrary_2_0.xsd">
    <tlib-version>2.1</tlib-version>
    <short-name>perso</short-name>
    <tag>
        <name>mostrarHora-3</name>
        <tag-class>pk1.MuestraHora-3</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
            <name>formato</name>
            <required>true</required>
        </attribute>
    </tag>
</taglib>

```

En el ejemplo mostrado a continuación la etiqueta transfiere el contenido de su cuerpo directamente en la respuesta HTTP.

```

package pk1;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class MuestraHora-3 extends SimpleTagSupport
{
    String formato;
    public void setFormat(String formato)
    {
        this.format=formato;
    }
    public void doTag() throws JspException,IOException
    {
        getJspBody().invoke(null);
        getJspContext().getOut().write(new
SimpleDateFormat(formato).format(new Date()));
    }
}

```

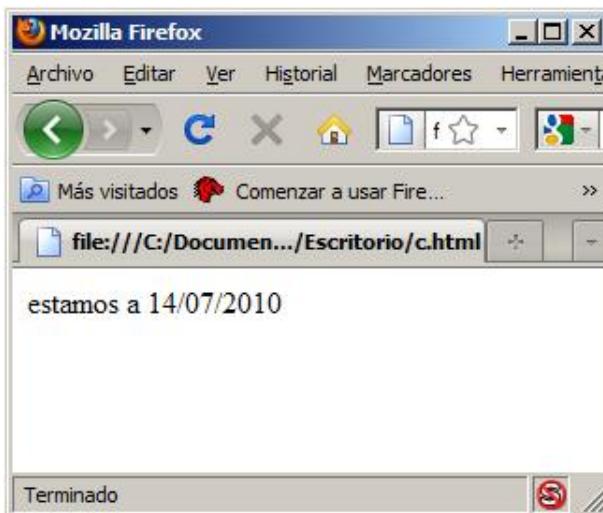
```
}
```

Esta etiqueta puede usarse con la forma siguiente en una página JSP.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="perso" uri="/WEB-INF/AfficheHeure-3.tld" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/HTML4/loose.dtd">

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <perso:muestraHora formato="dd/mm/yyyy">
      estamos a
    </perso:muestraHora>
  </body>
</html>
```

El documento HTML recibido por el navegador tiene en este caso la forma siguiente:



Cuando una etiqueta tiene que manipular el contenido de su cuerpo, hay que recuperarlo en una variable para que pueda ser usado en el método `doTag`. La solución consiste en utilizar de nuevo el método `invoke` del objeto `JspFragment` que contiene el cuerpo de la etiqueta pero pasándole esta vez como parámetro un objeto `StringWriter`. Este objeto `StringWriter` permite recuperar más adelante el cuerpo de la etiqueta en forma de cadena de caracteres.

Después de la manipulación, el cuerpo de la etiqueta puede transferirse a la respuesta por el objeto `Writer` asociado por defecto a la página JSP.

```
package pk1;

import java.io.IOException;
import java.io.StringWriter;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class MuestraHora-4 extends SimpleTagSupport
{
    String formato;
    public void setFormato(String formato)
    {
        this.formato=formato;
    }
}
```

```
public void doTag() throws JspException, IOException
{
    StringWriter lector;
    lecteur=new StringWriter();
    getJspBody().invoke(lector);
    getJspContext().getOut().write(lector.toString().toUpperCase());
    getJspContext().getOut().write(new
SimpleDateFormat(formato).format(new Date())));
}
```

Principio de funcionamiento de una base de datos

Las bases de datos se han convertido en elementos esenciales en la mayoría de las aplicaciones. Reemplazan el uso de archivos gestionados por el propio desarrollador. Con su aporte, se obtiene una importante ganancia de productividad en el desarrollo y una mejora significativa del rendimiento de las aplicaciones. Facilitan la compartición de datos entre usuarios. Para poder utilizar una base de datos, tiene que conocer un mínimo de vocabulario propio de esta tecnología.

1. Terminología

En el contexto de las bases de datos, los términos siguientes se usan con frecuencia:

Base de datos relacional: una base de datos relacional es un tipo de base de datos que utiliza tablas para almacenar información. Se usan valores de dos tablas para asociar los datos de una tabla con los datos de otra tabla. Por regla general, en una base de datos relacional, los datos sólo se almacenan una vez.

Tabla: una tabla es un componente de una base de datos que almacena datos en registros (filas) y en campos (columnas). Los datos generalmente se agrupan por categorías al nivel de tablas. Por ejemplo, tendremos la tabla de Clientes, de Productos o de Pedidos.

Registro: el registro es el conjunto de datos relativos a un elemento de una tabla. Los registros son el equivalente a nivel lógico de las filas de una tabla. Por ejemplo un registro de la tabla Clientes contiene las características de un cliente en particular.

Campo: un registro se compone de varios campos. Cada campo de un registro contiene un solo dato acerca del registro. Por ejemplo un registro Cliente puede contener los campos CódigoCliente, Nombre, Apellidos...

Clave primaria: una clave primaria se utiliza para identificar de manera única cada fila de una tabla. La clave primaria es un campo o una combinación de campos cuyo valor es único en la tabla. Por ejemplo el campo CódigoCliente es la clave primaria de la tabla Cliente. No puede haber dos clientes con el mismo código.

Clave foránea: una clave foránea representa uno o varios campos de una tabla que hacen referencia a los campos de la clave primaria de otra tabla. Las claves foráneas indican la manera en que las tablas se asocian.

Relación: una relación es una asociación establecida entre dos campos comunes en dos tablas. Una relación puede ser uno a uno, uno a muchos o muchos a muchos. Gracias a las relaciones, los resultados de las querys pueden tener datos de varias tablas. Una relación de uno a muchos entre la tabla Cliente y la tabla Pedido permite a una query reenviar todos los pedidos correspondientes a un cliente.

2. El lenguaje SQL

Antes de poder escribir una aplicación Java utilizando datos, tiene que familiarizarse con el lenguaje SQL (*Structured Query Language*). Este lenguaje permite dialogar con la base de datos. Existen diferentes versiones del lenguaje SQL en función de la base de datos utilizada. Sin embargo, SQL tiene una sintaxis elemental normalizada independiente de toda base de datos.

a. Búsqueda de información

El lenguaje SQL permite especificar los registros que se quieren extraer así como el orden en el que se desean extraer. Puede crear una instrucción SQL que extraiga datos de varias tablas simultáneamente, o puede crear una instrucción que extraiga únicamente un registro específico. La instrucción SELECT se utiliza para reenviar campos específicos de una o varias tablas de la base de datos.

La instrucción siguiente reenvía la lista de los nombres y apellidos de todos los registros de la tabla Cliente.

```
SELECT Nombre, Apellidos FROM Cliente
```

Puede utilizar el símbolo * en vez de la lista de los campos cuyo valor desea saber.

```
SELECT * FROM Cliente
```

Puede limitar el número de registros seleccionados utilizando uno o varios campos para filtrar el resultado de la instrucción. Hay diferentes cláusulas para realizar este filtrado.

Cláusula WHERE

Esta cláusula permite especificar la lista de condiciones que deberán cumplir los registros para formar parte de los

resultados devueltos. El ejemplo siguiente permite encontrar todos los clientes que habitan en Valencia.

```
SELECT * FROM Cliente WHERE Poblacion='Valencia'
```

La sintaxis de la cláusula necesita el uso de la comilla simple para delimitar las cadenas de caracteres.

Cláusula WHERE ... IN

Puede utilizar la cláusula `WHERE ... IN` para obtener todos los registros que responden a una lista de criterios. Por ejemplo puede buscar todos los clientes que habitan en Francia y en España.

```
SELECT * FROM Cliente WHERE Pais IN ('Francia','España')
```

Cláusula WHERE ... BETWEEN

También puede obtener una selección de registros que se sitúan entre dos criterios especificados. La petición siguiente permite recuperar la lista de los pedidos pasados el mes de noviembre de 2005.

```
SELECT * from Pedidos WHERE FechaPedido BETWEEN '01/11/05' AND '30/11/05'
```

Cláusula WHERE ... LIKE

Puede usar la cláusula `WHERE ... LIKE` para obtener todos los registros para los que existe una condición particular para un campo dado. Por ejemplo la sintaxis siguiente selecciona todos los clientes cuyo nombre empieza por d.

```
SELECT * FROM Cliente WHERE Nombre LIKE 'd%'
```

En esta instrucción el símbolo % se utiliza para reemplazar una secuencia de caracteres cualquiera.

Cláusula ORDER BY ...

Puede utilizar la cláusula `ORDER BY` para obtener los registros en un orden particular. La opción `ASC` indica un orden creciente y la opción `DESC` indica un orden decreciente. Varios campos pueden especificarse como criterio de ordenación. Se analizan de izquierda a derecha. En caso de igualdad en el valor de un campo, el campo siguiente es usado.

```
SELECT * FROM Cliente ORDER BY Nombre DESC,Apellidos ASC
```

Esta instrucción devuelve los clientes ordenados en orden decreciente para el nombre y en caso de igualdad por orden creciente en los apellidos.

b. Añadir datos

La creación de registros en una tabla se realiza con el comando `INSERT INTO`. Tiene que indicar la tabla en la que desea insertar la fila, la lista de campos para los que especifica un valor y finalmente una lista de los valores correspondientes. La sintaxis completa es por tanto la siguiente:

```
INSERT INTO cliente (codigoCliente,nombre,apellidos) VALUES (1000,'Pedro',  
'Rodríguez García')
```

Cuando se añade este nuevo cliente sólo el código, el nombre y los apellidos se pondrán en la tabla. Al resto de campos se les asignará el valor `NULL`. Si la lista de los campos no se introduce, la instrucción `INSERT` exige que se especifique un valor por cada campo de la tabla. Por lo tanto, estará con la obligación de utilizar la palabra clave `NULL` para indicar que para un campo en concreto no hay información. Si la tabla Cliente se compone de cinco campos (codigoCliente, nombre, apellidos, dirección, país) la instrucción anterior puede escribirse con la sintaxis siguiente:

```
INSERT INTO cliente VALUES (1000,'Pedro','Rodríguez García',NULL,NULL)
```

Observe que en este caso las dos palabras clave `NULL` son obligatorias para los campos dirección y país.

c. Actualización de datos

La modificación de campos para registros existentes se realiza con la instrucción `UPDATE`. Esta instrucción puede actualizar varios campos de varios registros de una tabla a partir de expresiones que se le han proporcionado. Tiene

que introducir el nombre de la tabla que se actualizará así como el valor que se asignará a cada campo. La lista de campos se indica con la palabra clave `SET` seguida de la asignación del nuevo valor para cada uno de los campos. Si quiere que las modificaciones sólo se lleven a cabo en un conjunto limitado de registros, tiene que especificar la cláusula `WHERE` para limitar el ámbito de la actualización. Si no se indica ninguna cláusula `WHERE`, la modificación se realizará sobre todos los registros de la tabla.

Por ejemplo, para modificar la dirección de un cliente en concreto puede utilizar la instrucción siguiente:

```
UPDATE Cliente SET direccion = 'Calle Cartagena 52, 08021 Barcelona' WHERE  
codigoCliente=1000
```

Si la modificación se realiza sobre todos los registros de la tabla, la cláusula `WHERE` es inútil. Por ejemplo, si quiere aumentar el precio de todos sus artículos, puede utilizar la instrucción siguiente:

```
UPDATE CATALOGO SET precioUnitario=precioUnitario*1.1
```

d. Eliminación de datos

La instrucción `DELETE FROM` permite eliminar uno o varios registros de una tabla. Tiene que introducir al menos el nombre de la tabla en la que desea realizar el borrado de información. Si no da más detalles, se borrarán todas las filas de la tabla. En general se añade una cláusula `WHERE` para limitar el alcance del borrado. El comando siguiente borra todos los registros de la tabla `Cliente`:

```
DELETE FROM Cliente
```

El comando siguiente es menos radical y sólo elimina un registro en concreto:

```
DELETE FROM Cliente WHERE codigoCliente=1000
```

El lenguaje SQL evidentemente es mucho más completo que lo visto hasta ahora y no se resume en cinco instrucciones. Sin embargo, son suficientes para la manipulación de datos desde Java. Si desea profundizar más en el aprendizaje del lenguaje SQL, le aconsejo que consulte alguna de las obras disponibles en esta misma colección que trate esta materia de un modo más detallado.

Acceso a una base de datos desde Java

Cuando se desea manipular una base de datos desde un lenguaje de programación hay dos soluciones disponibles:

- Comunicar directamente con la base de datos.
- Utilizar una capa lógica que proporcione el diálogo con la base de datos.

La primera solución conlleva varios requisitos:

- Tiene que dominar perfectamente la programación en red.
- Tiene que conocer al detalle el protocolo usado por la base de datos.

Este tipo de desarrollo suele ser muy largo y lleno de escollos. ¿Por ejemplo las especificaciones del protocolo son accesibles?

Todo su trabajo deberá ser revisado si cambia el tipo de base de datos ya que, por supuesto, los protocolos no son compatibles con las distintas bases de datos. A veces, es incluso peor el caso en que se actualiza a una siguiente versión una misma base de datos.

La segunda solución evidentemente es preferible y es la que los diseñadores de Java han elegido. Por lo tanto, han desarrollado la librería JDBC para el acceso a una base de datos. Más concretamente, la librería JDBC se compone de dos partes. La primera parte, albergada en el paquete `java.sql`, se compone esencialmente de interfaces. Estas interfaces las implementa el driver JDBC. Este driver no se desarrolla por parte de Sun sino generalmente por el creador de la base de datos. En efecto, es éste último quien domina mejor la técnica para comunicarse con la base de datos. Hay cuatro tipos de drivers JDBC con características y rendimientos distintos.

Tipo 1: driver jdbc-odbc

Este tipo de driver no es específico a una base de datos sino que traduce simplemente las instrucciones JDBC en instrucciones ODBC. A continuación, es el driver ODBC el que ofrece la comunicación con la base de datos. Esta solución sólo ofrece rendimientos mediocres. Esto es principalmente debido al número de capas lógicas usadas en la implementación. Las funcionalidades JDBC están además limitadas por las funcionalidades de la capa ODBC. Sin embargo, esta solución le permite acceder a prácticamente cualquier base de datos. Esta solución tiene que usarse cuando no existe ningún otro tipo de driver disponible.

Tipo 2: driver nativo

Este tipo de driver no se escribe completamente en Java. La parte de este driver escrita en Java realiza simplemente llamadas a funciones del driver nativo. Estas llamas se realizan gracias a la API JNI (*Java Native Interface*). Como sucede con los pilotos de tipo 1, hay por lo tanto una traducción necesaria entre el código Java y la base de datos. Sin embargo este tipo de driver es más eficaz que los drivers jdbc-odbc.

Tipo 3: driver que utiliza un servidor intermediario

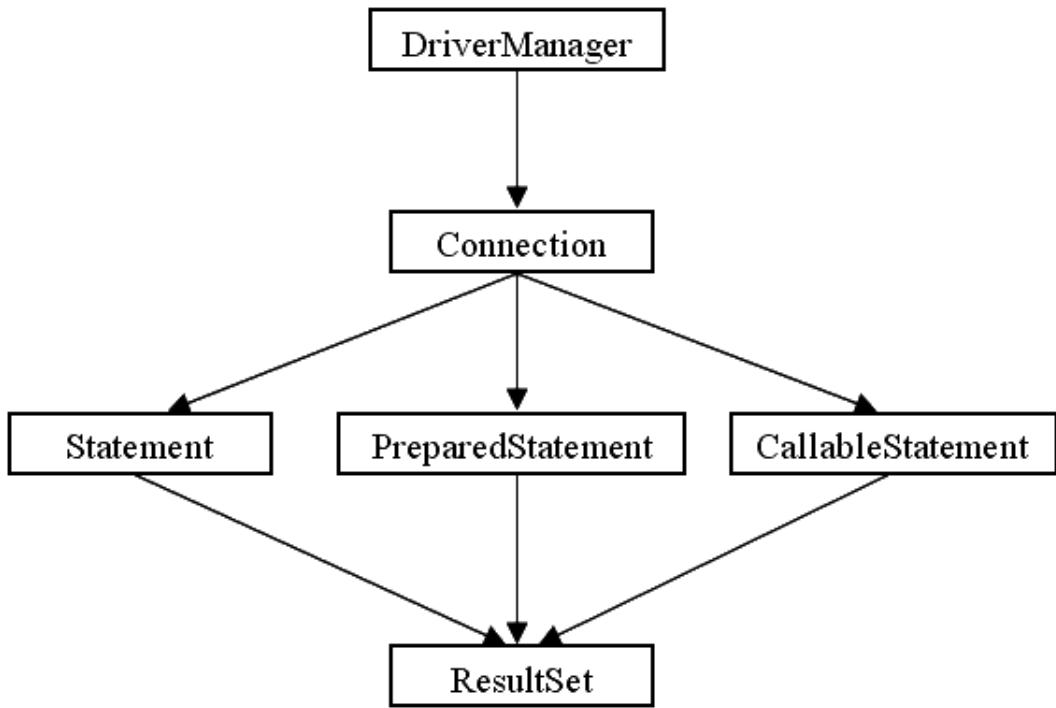
Este driver desarrollado completamente en Java no se comunica directamente con la base de datos sino con una dirección en un servidor intermediario. El diálogo entre el driver y el servidor intermediario es estándar independientemente del tipo de base de datos que haya detrás. A continuación, es el servidor intermediario el que transmite las instrucciones a la base de datos usando un protocolo específico de la misma.

Tipo 4: driver completamente escrito en Java

Este tipo de driver representa la solución ideal debido a que no hay ningún intermediario. El driver transmite directamente las peticiones a la base de datos utilizando un protocolo propio de la base de datos. La mayoría de los drivers son ahora de este tipo.

1. Presentación de JDBC

La librería JDBC proporciona un conjunto de clases y sobre todo de interfaces que permiten la manipulación de una base de datos. Estos elementos representan todo lo necesario para acceder y operar con los datos desde una aplicación Java. El esquema siguiente retoma el camino lógico desde el driver a los datos.



La clase `DriverManager` es nuestro punto de partida. Es la que ofrece el enlace con el driver. Es gracias a su uso como intermediario que podemos obtener una conexión con la base de datos. Ésta se representa con una instancia de clase que implementa la interfaz `Connection`. Esta conexión se utiliza para transmitir instrucciones a la base de datos. Las instrucciones simples se ejecutan gracias a la interfaz `Statement`, las instrucciones con parámetros se usan con la interfaz `PreparedStatement` y los procedimientos almacenados con la interfaz `CallableStatement`.

Los posibles registros seleccionados por la instrucción SQL son accesibles con un elemento `ResultSet`. Vamos a detallar estas distintas etapas en los párrafos siguientes.

2. Establecer y manipular la conexión

Después de su carga el driver es capaz de proporcionarnos una conexión al servidor de base de datos.

a. Carga del driver

La primera etapa indispensable es obtener el driver JDBC adaptado a su base de datos. En general este driver está disponible para su descarga en el sitio web del creador de la base de datos. Para nuestros ejemplos usaremos el driver proporcionado por Microsoft para acceder a un servidor de base de datos SQL Server 2005. Éste puede descargarse en la dirección siguiente:

<http://msdn.microsoft.com/en-us/data/aa937724.aspx>

Después de su descompresión, obtendrá un directorio que contendrá el driver en sí mismo en forma de empaquetado Java (`sqljdbc.jar`) y muchos archivos de ayuda sobre el uso de este driver. El archivo empaquetado contiene las clases desarrolladas por Microsoft que implementan las distintas interfaces JDBC. Este archivo deberá ser accesible cuando se compile y se ejecute la aplicación.

Este driver tiene que cargarse mediante el método `forName` de la clase `Class`. Este método espera como parámetro una cadena de caracteres que contengan el nombre del driver. Es indispensable en este nivel recorrer la documentación del driver para obtener el nombre de la clase. En nuestro caso, la clase tiene el nombre siguiente:

`com.microsoft.sqlserver.jdbc.SQLServerDriver`

El registro del driver se realiza entonces con la instrucción siguiente:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

El driver que ofrece el acceso a un origen de datos ODBC está disponible directamente con el JDK. Puede cargarse con la sintaxis siguiente:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Las cadenas de caracteres pasadas por parámetro representan los nombres de las clases, por lo tanto se diferencian las mayúsculas y las minúsculas. Por otra parte, la instrucción `forName` tiene que estar protegida por un bloque `try catch` ya que es susceptible de lanzar una excepción de tipo `ClassNotFoundException`.

b. Establecer la conexión

El método `getConnection` de la clase `DriverManager` se encarga de este trabajo. Este método ofrece tres soluciones para establecer una conexión. Cada una de las versiones de este método espera como parámetro una cadena de caracteres que representa una URL que contiene los datos necesarios para establecer la conexión. El contenido de esta cadena es específico para cada driver y hay que consultar de nuevo la documentación para obtener la sintaxis correcta. El inicio de esta cadena de caracteres es, sin embargo, estándar con la forma siguiente `jdbc :nombreDelProtocolo`. El nombre del protocolo es propio a cada driver y es gracias a este nombre que el método `getConnection` es capaz de identificar al driver adecuado. El resto de la cadena es completamente específico a cada driver. Contiene en general los datos que permiten identificar el servidor y la base de datos en este servidor con la que la conexión tiene que establecerse. Para el driver de Microsoft, la sintaxis básica es la siguiente:

```
jdbc:sqlserver://localhost;databaseName=northwind;  
user=thierry;password=secret;
```

Para el driver ODBC, la sintaxis es más sencilla ya que basta simplemente con especificar el nombre del origen ODBC al que se desea conectar. Este origen ODBC evidentemente tiene que haberse creado con anterioridad.

```
jdbc:odbc:nwd
```

En caso de éxito, el método `getConnection` devuelve una instancia de la clase que implementa la interfaz `Connection`. En algunos casos es mejor usar la segunda versión del método `getConnection` ya que permite indicar el usuario y la contraseña utilizados para establecer la conexión como parámetros separados de la URL de conexión. A continuación se muestran dos ejemplos que permiten establecer una conexión a una misma base de datos. El primero utiliza el driver Microsoft.

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class ConexionDirecta  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");  
        }  
        catch (ClassNotFoundException e)  
        {  
            System.out.println("error durante la carga del driver");  
        }  
        Connection cnxDirecta=null;  
        try  
        {  
            cnxDirecta=DriverManager.getConnection("jdbc:sqlserver://local  
host;databaseName=northwind; user=thierry;password=secret");  
        }  
        catch (SQLException e)  
        {  
            System.out.println("error durante la conexión");  
        }  
    }  
}
```

El segundo usa el driver ODBC.

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;  
  
public class ConexionDirectaODBC
```

```

{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException e)
        {
            System.out.println("error durante la carga del driver");
        }
        Connection cnxOdbc=null;
        try
        {
            cnxOdbc=DriverManager.getConnection("jdbc:odbc:nwd","thierry",
"secret");
        }
        catch (SQLException e)
        {
            System.out.println("error durante la conexión");
        }
    }
}

```

c. Manipular la conexión

Una conexión está abierta una vez ha sido creada. No hay por lo tanto un método que permita abrir una conexión. Por contra, una conexión puede cerrarse llamando al método `close`. Después del cierre de una conexión, ya no se podrá usar esa conexión y tiene que volver a crearse para usarse de nuevo. La función `isClosed` permite verificar si una conexión ya está cerrada. Si esta función devuelve un booleano igual a `false` se puede pensar que la conexión está abierta y que se permite el diálogo con la base de datos. Sin embargo, no siempre es así. A veces, la conexión puede estar en un estado intermedio: no está cerrada pero no puede utilizarse para transferir instrucciones al servidor. Para verificar la disponibilidad de la conexión puede utilizar el método `isValid`. Este método comprueba realmente la disponibilidad de la conexión intentando enviar una instrucción SQL y verificando que obtiene una respuesta desde el servidor. Este método no está implementado en todos los drivers y si no lo está, su llamada desencadena una excepción del tipo `java.lang.UnsupportedOperationException` o un error del tipo `java.lang.AbstractMethodError`.

Si únicamente realiza operaciones de lectura en la base de datos, puede optimizar la comunicación indicando que la conexión es de sólo lectura. El método `setReadOnly` permite modificar este parámetro de la conexión. Su estado puede a continuación ser comprobado utilizando el método `isReadOnly`. Para algunos drivers esta funcionalidad no está implementada y el método `setReadOnly` no tiene ningún efecto. La función siguiente verifica si esta funcionalidad está disponible para la conexión que se le pasa por parámetro.

```

public static void compruebaSoloLectura(Connection cnx)
{
    boolean estado;
    try
    {
        estado = cnx.isReadOnly();
        cnx.setReadOnly(!estado);
        if (cnx.isReadOnly() != estado)
        {
            System.out
                .println("el modo de
sólo lectura es compatible con este driver");
        }
        else
        {
            System.out
                .println("el modo de
sólo lectura no es compatible con este driver");
        }
        cnx.setReadOnly(estado);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

```

```
    }
}
```

Cuando se ejecuta una instrucción SQL, el servidor puede detectar algún problema y por ello puede generar avisos. Estos avisos pueden recuperarse con el método `getWarnings` de la conexión. Este método devuelve un objeto `SQLWarning` representativo del problema encontrado por el servidor. Si el servidor encuentra varios problemas, genera varios objetos `SQLWarning` encadenados los unos a los otros. El método `getNextWarning` permite obtener el elemento siguiente o `null` si el actual es el último de la lista. La lista se puede vaciar con el método `clearWarnings`. La función siguiente muestra todos los avisos recibidos por la conexión.

```
public static void visualizarWarnings(Connection cnx)
{
    SQLWarning aviso;
    try
    {
        aviso=cnx.getWarnings();
        if (aviso==null)
        {
            System.out.println("no hay avisos");
        }
        else
        {
            while (aviso!=null)
            {
                System.out.println(aviso
.getMessage());
                System.out.println(aviso
.getSQLState());
                System.out.println(aviso
.getErrorCode());
                aviso=aviso
.getNextWarning();
            }
        }
        cnx.clearWarnings();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
```

En general en el momento de la creación de la URL de conexión, uno de los parámetros de esta URL determina el nombre de la base de datos con la que se desea establecer una conexión. En el caso en que se utilice un origen de datos ODBC, es el nombre de este origen de datos el que debe indicarse, el nombre de la base de datos tiene que haberse dado cuando se creó el origen de datos. En este caso el nombre de la base de datos puede obtenerse con el método `getCatalog`. La modificación del nombre de la base de datos a la que está conectado se realiza con el método `setCatalog` al que hay que pasarle por parámetro el nombre de otra base de datos presente en el mismo servidor. Por supuesto, es necesario que la cuenta con la que ha abierto la conexión tenga suficientes derechos de acceso para esta base de datos. Observe que con este método, cambiamos de base de datos pero la conexión sigue siempre en el mismo servidor. No hay ninguna forma de cambiar de servidor sin crear una nueva conexión.

El código siguiente:

```
System.out.println("base de datos actual: " +cnxDirecta.getCatalog());
System.out.println("cambio de base de datos");
cnxDirecta.setCatalog("garaje");
System.out.println("base de datos actual: " +cnxDirecta.getCatalog());
```

... muestra:

base de datos actual: northwind

cambio de base de datos

base de datos actual: garaje

La estructura de la base de datos puede obtenerse con el método `getMetaData`. Este método devuelve un objeto `DataBaseMetaData` que proporciona gran cantidad de información sobre la estructura de la base de datos. La función siguiente muestra un análisis rápido de esta información.

```
public static void infoBaseDeDatos(Connection cn)
```

```

{
    ResultSet rs;
    DatabaseMetaData dbmd;
    try
    {
        dbmd=cn.getMetaData();
        System.out.println("tipo de base de datos: " +
dbmd.getDatabaseProductName());
        System.out.println("versión: " +
dbmd.getDatabaseProductVersion());
        System.out.println("nombre del driver: " +
dbmd.getDriverName());
        System.out.println("versión del driver: " +
dbmd.getDriverVersion());
        System.out.println("nombre del usuario: " +
dbmd.getUserName());
        System.out.println("url de conexión: " +
dbmd.getURL());
        rs=dbmd.getTables(null,null,"%",null);
        System.out.println("estructura de la base de datos");
        System.out.println("base de datos\tesquema\tnombre de tabla\
t tipo de tabla");
        while(rs.next())
        {
            for (int i = 1; i <=4 ; i++)
            {
                System.out.print(rs.getString(i)+"\t");
            }
            System.out.println();
        }
        rs.close();
        rs=dbmd.getProcedures(null,null,"%");
        System.out.println("procedimientos almacenados");
        System.out.println("base de datos\tesquema\t nombre del
procedimiento");
        while(rs.next())
        {
            for (int i = 1; i <=3 ; i++)
            {
                System.out.print(rs.getString(i)+"\t");
            }
            System.out.println();
        }
        rs.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
}

```

Todos estos métodos le podrán ser útiles algún día pero el objetivo principal de una conexión es el de permitir la ejecución de instrucciones SQL. Es por tanto la conexión la que nos proporcionará los objetos necesarios para la ejecución de instrucciones. Se pueden ejecutar tres tipos de instrucciones SQL:

- instrucciones sencillas
- instrucciones precompiladas
- procedimientos almacenados

A cada uno de estos tipos le corresponde un objeto JDBC:

- Statement para las intrucciones sencillas
- PreparedStatement para las instrucciones precompiladas

- `CallableStatement` para los procedimientos almacenados

Y finalmente cada uno de estos objetos puede obtenerse por un método diferente de la conexión:

- `createStatement` para los objetos `Statement`
- `prepareStatement` para los objetos `PreparedStatement`
- `prepareCall` para los objetos `CallableStatement`

La utilización de estos métodos y de estos objetos se detalla en la sección siguiente.

3. Ejecución de instrucciones SQL

Antes de la ejecución de una instrucción SQL tiene que elegir el tipo de objeto JDBC más apropiado. En las secciones siguientes se describen los tres tipos de objetos disponibles y su utilización.

a. Ejecución de instrucciones de base de datos con el objeto Statement

Este objeto se obtiene con el método `createStatement` de la conexión. Hay dos versiones de este método, la primera no espera ningún parámetro. En este caso, si el objeto `Statement` se utiliza para ejecutar una instrucción SQL generando un conjunto de registros (`select`), será de sólo lectura y con recorrido hacia adelante.

Los datos presentados en este conjunto de registro no se podrán modificar y el recorrido del conjunto de registros sólo se podrá realizar desde el primer registro. La segunda versión permite escoger las características del conjunto de registros generado. Acepta dos parámetros. El primero determina el tipo de conjunto de registros. Se han definido las siguientes constantes:

`ResultSet.TYPE_FORWARD_ONLY`: el conjunto de registros sólo se podrá recorrer hacia adelante.

`ResultSet.TYPE_SCROLL_INSENSITIVE`: el conjunto de registros se podrá recorrer en los dos sentidos pero será insensible a los cambios realizados por otros usuarios en la base de datos.

`ResultSet.TYPE_SCROLL_SENSITIVE`: el conjunto de registros se podrá recorrer en ambos sentidos y será sensible a los cambios realizados por otros usuarios en la base de datos.

El segundo parámetro determina las posibilidades de modificación de los datos contenidos en el conjunto de registros. Se han definido las dos constantes siguientes:

`ResultSet.CONCUR_READ_ONLY`: los registros son de sólo lectura.

`ResultSet.CONCUR_UPDATABLE`: los registros pueden modificarse en el conjunto de registros.

Este objeto es el más elemental de los que permiten ejecutar instrucciones SQL. Puede encargarse de la ejecución de cualquier tipo de instrucción SQL. Por lo tanto, puede ejecutar tanto instrucciones DDL (*Data Definition Language*) como instrucciones DML (*Data Manipulation Language*). Simplemente hay que elegir en este objeto el método que más se adapte a la ejecución del código SQL. La elección de este método viene dictada por el tipo de resultado que tiene que proporcionar la instrucción SQL. Hay cuatro métodos posibles:

`public boolean execute(String sql)`: este método permite la ejecución de cualquier instrucción SQL. El booleano devuelto por este método indica si un conjunto de registros se ha generado (`true`) o si simplemente la instrucción ha modificado registros en la base de datos (`false`). Si un conjunto de registros se ha generado, puede obtenerse con el método `getResultSet`. Si hay registros que han sido modificados en la base de datos, el método `getUpdateCount` devuelve el número de registros modificados. La función siguiente ilustra el uso de estos métodos.

```
public static void compruebaExecute(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String instrucion;
    ResultSet rs;
    boolean resultado;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
```

```

        System.out.println("introduzca la instrucción SQL:");
        instrucion=br.readLine();
        resultado=stm.execute(instrucion);
        if (resultado)
        {
            System.out.println("la instrucción ha generado un
conjunto de registros");
            rs=stm.getResultSet();
            rs.last();
            System.out.println("contiene " +
rs.getRow() + " registros");
        }
        else
        {
            System.out.println("la instrucción ha modificado
registros en la base de datos");
            System.out.println("número de registros
modificados: " + stm.getUpdateCount());
        }
    }
    catch (SQLException e)
    {
        System.out.println("la instrucción no ha funcionado
correctamente");
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

public Resultset executeQuery(String query): este método se ha creado especialmente para la ejecución de instrucciones select. El conjunto de registros está disponible directamente como valor devuelto por la función.

public int executeUpdate(String instrucion): este método se adapta perfectamente a la ejecución de instrucciones que modifican el contenido de la base de datos, como son las intrucciones INSERT, UPDATE y DELETE. El número entero devuelto por esta función indica el número de registros afectados por la modificación.

public int[] executeBatch(): este método permite ejecutar un conjunto de instrucciones SQL por lotes. El lote de instrucciones que se ejecutarán tiene que prepararse previamente con los métodos addBatch y clearBatch. El primero recibe como parámetro una cadena de caracteres que representa la instrucción SQL que se añadirá al lote. El segundo permite reinicializar el lote de instrucciones. No se puede eliminar una instrucción en concreto del lote. Además, no puede añadir al lote una instrucción SQL que genere un conjunto de resultados, ya que en este caso el método updateBatch desencadena una excepción del tipo BatchUpdateException. Esta función devuelve una tabla de enteros que permite obtener información sobre la ejecución de cada una de las instrucciones del lote. Cada celda de la tabla contiene un entero que representa el resultado de la ejecución de la instrucción correspondiente en el lote. Un valor mayor o igual a 0 indica un funcionamiento correcto de la instrucción y representa el número de registros modificados. Un valor igual a la constante Statement.EXECUTE_FAILED indica que la ejecución de la instrucción ha sido incorrecta. En este caso algunos drivers detienen la ejecución del lote mientras que otros continúan con la ejecución de la siguiente instrucción del lote. Un valor igual a la constante Statement.SUCCESS_NO_INFO indica que la instrucción se ha ejecutado correctamente pero que el número de registros modificados no se ha podido determinar. Este método es muy práctico para ejecutar modificaciones en varias tablas asociadas. Podría ser el caso en una aplicación de gestión comercial con una tabla para los pedidos y una tabla para las líneas de los pedidos. La eliminación de un pedido debe implicar en este caso la eliminación de todas sus líneas correspondientes. La función siguiente le permite introducir varias instrucciones SQL y ejecutarlas por lotes.

```

public static void compruebaExecuteBatch(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String instrucion="";
    int[] resultados;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("introduzca sus instrucciones SQL y
después run para ejecutar el lote:");
        instrucion=br.readLine();
    }

```

```

        while (!instruccion.equalsIgnoreCase("run"))
        {
            stm.addBatch(instruccion);
            instruccion=br.readLine();
        }
        System.out.println("ejecución del lote de instrucciones");
        resultados=stm.executeBatch();
        for (int i=0; i<resultados.length;i++)
        {
            switch (resultados[i])
            {
                case Statement.EXECUTE_FAILED:
                    System.out.println("la ejecución de la
instrucción " + i + " ha fallado");
                    break;
                case Statement.SUCCESS_NO_INFO:
                    System.out.println("la ejecución de la
instrucción " + i + " ha tenido éxito");
                    System.out.println("el número de
registros modificados es desconocido");
                    break;
                default:
                    System.out.println("la ejecución de la
instrucción " + i + " ha tenido éxito");
                    System.out.println("ha modificado "
+ resultados[i] + " registros");
                    break;
            }
        }

    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

Hay otros métodos que pueden intervenir en el comportamiento del objeto Statement.

`public void setQueryTimeOut(int duracion):` este método indica la duración máxima permitida para la ejecución de una instrucción SQL antes de desencadenar una excepción.

`public void close():` cuando un objeto Statement deja de usarse en una aplicación, es preferible cerrarlo explícitamente llamando a este método. Éste provoca la liberación de todos los recursos utilizados por este objeto. El cierre de un objeto Statement provoca también el cierre del conjunto de registros asociado.

`public void setMaxRows(int numero):` este método limita el número de filas de los conjuntos de registros generados por este objeto Statement. Si una instrucción SQL genera un conjunto de caracteres con más filas, las filas excedentes simplemente se ignoran (sin más información).

`public void setMaxFieldSize(int tamaño):` este método limita el tamaño de algunos tipos de campos en el conjunto de registros. Los tipos de campos afectados son los campos de una base de datos que pueden tener un tamaño variable como por ejemplo los campos de caracteres o binarios. Los datos excedentes simplemente se ignoran. Los campos afectados pueden quedar inutilizables en la aplicación.

`public void setFetchSize(int numFilas):` cuando una instrucción SQL genera un conjunto de registros, los datos correspondientes se transfieren del servidor de base de datos a la memoria de la aplicación Java. Esta transferencia puede realizarse por bloques en función del uso de los datos. Este método indica al driver el número de filas de cada bloque transferido de la base de datos a la aplicación.

`public boolean getMoreResults():` si el método execute se utiliza para ejecutar varias instrucciones SQL, por ejemplo dos instrucciones select, en este caso se generan dos conjuntos de registros. El primero se obtiene con el método getResultset. El segundo no será accesible hasta que se haya llamado al método getMoreResults. Esta función permite desplazar el puntero sobre el resultado siguiente y devuelve un booleano igual a true si el resultado siguiente es un conjunto de registros. Si éste es el caso, será también accesible con el método getResultset. Si la función getMoreResults devuelve un booleano false significa que el resultado siguiente no es un conjunto de registros o que simplemente no hay resultado siguiente. Para salir de dudas, hay que llamar a la

función `getUpdateCount` y verificar el valor devuelto por esta función. Si es igual a -1 es que no hay resultado siguiente si no el valor devuelto representa el número de registros modificados en la base de datos.

La función siguiente permite la ejecución de varias instrucciones SQL separadas por puntos y coma:

```
public static void compruebaExecuteMultiple(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String instrucion;
    ResultSet rs;
    boolean resultado;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("introduzca sus instrucciones SQL
separadas por ::");
        instrucion=br.readLine();
        resultado=stm.execute(instrucion);
        int i=1;
        // tratamiento del resultado generado por la primera
instrucción
        if (resultado)
        {
            System.out.println("la instrucción Nº "
+ i + " ha generado un conjunto de registros");
            rs=stm.getResultSet();
            rs.last();
            System.out.println("contiene " +
rs.getRow() + " registros");
        }
        else
        {
            System.out.println("la instrucción Nº "
+ i + " ha modificado registros en la base de datos");
            System.out.println("número de
registros modificados:" + stm.getUpdateCount());
        }
        i++;
        // desplazamiento del puntero a un posible
resultado siguiente
        resultado=stm.getMoreResults();
        // bucle mientras haya aún resultados de tipo
conjunto de registros -> resultado==true
        // o que haya resultados de tipo número de
registros modificados -> getUpdateCount != -1
        while (resultado || stm.getUpdateCount()!=-1)
        {
            if (resultado)
            {
                System.out.println("la instrucción
Nº " + i + " ha generado un conjunto de registros");
                rs=stm.getResultSet();
                rs.last();
                System.out.println("contiene " +
rs.getRow() + " registros");
            }
            else
            {
                System.out.println("la instrucción
Nº " + i + " ha modificado registros en la base de datos");
                System.out.println("número
de registros modificados:" + stm.getUpdateCount());
            }
            i++;
            // desplazamiento del puntero a un posible
resultado siguiente
            resultado=stm.getMoreResults();
        }
    }
}
```

```

        }
    }
    catch (SQLException e)
    {
        System.out.println("la instrucción no ha
funcionado correctamente");
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

```

b. Ejecución de instrucciones parametrizadas con el objeto PreparedStatement

Frecuentemente, se da el caso de tener que ejecutar varias veces una intrucción SQL sólo con una pequeña modificación entre dos instrucciones. El ejemplo clásico corresponde a una selección con una restricción.

```

stm=cnx.createStatement	ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CON
CUR_READ_ONLY;
resultado=stm.execute("select * from customers where customerID='ALFKI'");

```

El valor en el que se realiza la restricción lo introduce generalmente el usuario de la aplicación y en este caso está disponible en una variable. La primera solución que viene en mente consiste en construir una intrucción SQL concatenando varias cadenas de caracteres.

```

public static void compruebaInstruccionConcat(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String instrucion;
    String codigo;
    ResultSet rs;
    boolean resultado;
    try
    {
        stm=cnx.createStatement	ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY;
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("introduzca el código del cliente
buscado:");
        codigo=br.readLine();
        instrucion="select * from customers where
customerID=' " + codigo + " '";
        resultado=stm.execute(instrucion);
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    } catch (IOException e) {
        // TODO Bloque catch autogenerado
        e.printStackTrace();
    }
}

```

Esta solución presenta varios inconvenientes:

La concatenación de cadenas de caracteres consume muchos recursos.

El servidor tiene que analizar cada vez una nueva instrucción.

La sintaxis se volverá rápidamente compleja si hay varias cadenas que tienen que concatenarse. Siempre hay que tener en mente que al final tenemos que obtener una instrucción SQL correcta.

Los diseñadores de JDBC han previsto una solución eficaz para paliar estos inconvenientes. El objeto PreparedStatement aporta una solución eficaz a este problema que nos permite la creación de instrucciones con parámetros. En este tipo de instrucción, los parámetros se reemplazan por símbolos de interrogación. Antes de la ejecución de la instrucción, hay que proporcionar al objeto PreparedStatement los valores que se deben usar para

reemplazar los diferentes símbolos de interrogación. Un objeto `PreparedStatement` puede crearse usando el mismo principio que para la creación de un objeto `Statement`. El método `prepareStatement`, accesible desde una conexión, devuelve un objeto `PreparedStatement`. Este método está disponible en dos versiones. La primera espera como argumento una cadena de caracteres que representa la instrucción SQL. En esta instrucción la ubicación de los parámetros tiene que reservarse usando los interrogantes. Si un conjunto de registros se crea por la ejecución de esta instrucción éste sera de sólo lectura y de recorrido hacia adelante solamente. La segunda versión del método `prepareStatement` espera, además de la cadena de caracteres, un argumento que indique el tipo de conjunto de registros y otro que determine las posibilidades de modificación de los datos contenidos en el mismo. Se pueden usar las mismas constantes que para el método `createStatement`.

```
stm=cnx.prepareStatement("select * from customers where
customerID=?",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
```

Antes de la ejecución de la instrucción SQL, tendrá que proporcionar un valor para cada uno de los interrogantes que representan un parámetro. Para ello el objeto `PreparedStatement` dispone de muchos métodos que permiten la asignación de un valor a un parámetro. Cada uno de estos métodos corresponde al tipo de datos SQL que se insertará en el sitio del interrogante. Estos métodos se llaman según el esquema: `setXXXX` donde `XXXX` representa un tipo de datos SQL. Cada uno de estos métodos espera como primer parámetro un entero que corresponde a la posición del parámetro en la instrucción SQL. El primer parámetro se sitúa en la posición 1. El segundo parámetro corresponde al valor que se asignará al parámetro. El tipo de este parámetro evidentemente se corresponde al tipo de datos SQL a transferir como parámetro. El driver JDBC convierte a continuación el tipo Java en tipo SQL.

Por ejemplo el método `setInt(int indiceParam, int value)` realiza una conversión de tipo int Java a tipo `INTEGER` sql. Los valores almacenados en los parámetros se conservan de ejecución en ejecución de la instrucción SQL. El método `clearParameters` permite reinicializar el conjunto de parámetros.

Los parámetros pueden ser usados en una instrucción SQL reemplazando valores pero nunca para reemplazar un nombre de campo y aún menos para reemplazar un operador. La sintaxis siguiente obviamente no está permitida:

```
stm=cnx.prepareStatement("select * from customers where ?=
?",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
stm.setString(1,"customerID");
stm.setString(2,"ALFKI");
rs=stm.executeQuery();
```

El resto de métodos disponibles con un objeto `PreparedStatement` son completamente idénticos a los definidos para un objeto `Statement` ya que la interfaz `PreparedStatement` hereda directamente de la interfaz `Statement`.

```
public static void compruebaPreparedStatement(Connection cnx)
{
    {
        PreparedStatement stm;
        BufferedReader br;
        String codigo;
        ResultSet rs;
        try
        {
            stm=cnx.prepareStatement("select * from
customers where customerID like ?",ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
            br=new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("introduzca el código del
cliente que se buscará:");
            codigo=br.readLine();
            stm.setString(1,codigo);
            rs=stm.executeQuery();
            while (rs.next())
            {
                for (int i = 1; i
<=rs.getMetaData().getColumnCount(); i++)
                {
                    System.out.print(rs.getString(i)+"\t");
                }
                System.out.println();
            }
        }
        catch (SQLException e)
        {
            e.printStackTrace();
        }
    }
}
```

```

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

```

c. Ejecución de procedimientos almacenados con el objeto CallableStatement

Un procedimiento almacenado representa código SQL almacenado en el servidor. Esta aproximación intenta obtener las siguientes ventajas:

- Los tratamientos y el tiempo de respuesta mejoran.
- El código SQL puede compartirse entre varias aplicaciones.

Por contra las aplicaciones se vuelven más dependientes del servidor de base de datos. El cambio de servidor le obligará a reescribir sus procedimientos almacenados ya que la sintaxis de un procedimiento almacenado es propia de cada servidor.

Los procedimientos almacenados son accesibles desde Java gracias al objeto `CallableStatement`. Como sucede con los objetos `Statement` y `PreparedStatement`, es también una conexión la que nos proporcionará una instancia de esta clase. En este caso, el método `prepareCall` es el que debe usarse. Este método espera como argumento una cadena de caracteres que identifique el procedimiento almacenado que se invocará. Por contra, la sintaxis de esta cadena de caracteres es un poco especial debido a que no basta con indicar el nombre del procedimiento almacenado. Hay que tener en cuenta dos posibles situaciones según si el procedimiento almacenado devuelve o no un valor. Si el procedimiento almacenado no devuelve un valor, la sintaxis de esta cadena de caracteres es la siguiente:

```
{call nombreProcedimiento( ?, ?,... )}
```

En esta sintaxis los interrogantes representan los parámetros esperados por el procedimiento almacenado. Como para el objeto `PreparedStatement` los valores de estos parámetros tienen que proporcionarse con los métodos `setXXXX` correspondientes al tipo de parámetro.

Si el procedimiento almacenado devuelve un valor, hay que usar la sintaxis siguiente que añade un parámetro adicional para albergar el retorno del procedimiento almacenado.

```
{ ?=call nombreProcedimiento( ?, ?,... )}
```

En la ejecución del procedimiento se guardará un valor en el primer parámetro que se utiliza para el retorno del procedimiento almacenado. Es necesario informar el objeto `CallableStatement` invocando al método `registerOutParameter`. Este método espera como primer parámetro el índice del parámetro, 1 para el valor de retorno del procedimiento almacenado, pero también podría ser cualquier otro parámetro que se utilizara en salida y como segundo parámetro el tipo SQL del parámetro. Este tipo puede indicarse con una de las constantes definidas en la interfaz `java.sql.Types`. Después de la ejecución del procedimiento almacenado, el valor de los parámetros utilizados de salida es accesible con los métodos `getXXXX` donde `xxxx` representa el tipo SQL del parámetro. Estos métodos esperan como argumento el índice del parámetro en la invocación del procedimiento almacenado.

Como sucede con los métodos `createStatement` y `prepareStatement` el método `prepareCall` ofrece una segunda sintaxis que permite indicar las características de un posible conjunto de resultados generado por la ejecución del procedimiento almacenado.

Para ilustrar el uso de este objeto `CallableStatement` vamos a utilizar los dos procedimientos almacenados siguientes:

```

create PROCEDURE pedidosPorCliente @codigo nchar(5)
AS
SELECT OrderID,
       OrderDate,
       RequiredDate,
       ShippedDate
FROM Orders
WHERE CustomerID = @codigo
ORDER BY OrderID

```

```

CREATE procedure numPedidos @codigo nchar(5) as
declare @num int
select @num=count(*) from orders where customerid=@codigo
return @num

```

El primero devuelve la lista de todos los pedidos del cliente cuyo código se pasa por parámetro. El segundo devuelve un entero correspondiente al número de pedidos realizados por el cliente cuyo código se le ha pasado por parámetro.

```

public static void compruebaProcedimientoAlmacenado(Connection cnx)
{
    CallableStatement cstml,cstm2;
    BufferedReader br;
    String codigo;
    ResultSet rs;
    int numPedidos;
    try
    {
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("introduzca el código del cliente
buscado: ");
        codigo=br.readLine();
        cstml=cnx.prepareCall("{ ?=call numPedidos ( ? )}");
        cstml.setString(2,codigo);
        cstml.registerOutParameter(1,java.sql.Types.INTEGER);
        cstml.execute();
        numPedidos=cstml.getInt(1);
        System.out.println("número de pedidos realizados
por el cliente " + codigo + " : " + numPedidos );
        cstm2=cnx.prepareCall("{ call pedidosPorCliente
( ? )}",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
        cstm2.setString(1,codigo);
        rs=cstm2.executeQuery();
        System.out.println("detalle de los pedidos");
        System.out.println("número de pedido\tfecha del pedido");
        while (rs.next())
        {
            System.out.print(rs.getInt("OrderID") + "\t");
            System.out.println(new
SimpleDateFormat("dd/MM/yy").format(rs.getDate("OrderDate")));
        }
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
}

```

4. Utilización de conjuntos de registros con la interfaz ResultSet

Cuando una instrucción SQL SELECT se ejecuta con el método executeQuery de un objeto Statement, PreparedStatement o CallableStatement, ésta devuelve un objeto ResultSet. Es mediante este objeto ResultSet que vamos a poder intervenir sobre el conjunto de registros. Nuestras posibilidades de acción en este conjunto de registros quedan determinadas por las características del objeto ResultSet. Estas características se fijan en el momento de creación de los objetos Statement, PreparedStatement o CallableStatement en función de los argumentos pasados en el momento de la invocación de los métodos createStatement, prepareStatement o prepareCall.

El primer argumento determina el tipo de conjunto de registros. Se han definido las constantes siguientes:

`ResultSet.TYPE_FORWARD_ONLY`: el conjunto de registros se recorrerá hacia adelante solamente.

`ResultSet.TYPE_SCROLL_INSENSITIVE`: el conjunto de registros podrá recorrerse en ambas direcciones pero será insensible a los cambios realizados en la base de datos por el resto de usuarios.

`ResultSet.TYPE_SCROLL_SENSITIVE`: el conjunto de registros podrá recorrerse en ambas direcciones y será sensible a los cambios realizados en la base de datos por otros usuarios.

El segundo argumento determina las posibilidades de modificación de los datos albergados en el conjunto de registros. Se han definido las dos siguientes constantes:

`ResultSet.CONCUR_READ_ONLY`: los registros están en modo sólo lectura.

`ResultSet.CONCUR_UPDATABLE`: los registros pueden modificarse en el conjunto de registros.

Evidentemente, es necesario que las acciones ejecutadas en el objeto `Resultset` sean compatibles con estas características sino se lanzará una excepción. Se puede verificar las características de un objeto `Resultset` usando los métodos `getType` y `getConcurrency`.

```
public static void infosResultSet(ResultSet rs)
{
    try {
        switch (rs.getType())
        {
            case ResultSet.TYPE_FORWARD_ONLY:
                System.out.println("el Resultset se
recorre solamente hacia adelante");
                break;
            case ResultSet.TYPE_SCROLL_INSENSITIVE:
                System.out.println("el Resultset se
puede recorrer en ambos sentidos");
                System.out.println("no es sensible
a las modificaciones realizadas por otros usuarios");
                break;
            case ResultSet.TYPE_SCROLL_SENSITIVE:
                System.out.println("el Resultset se
puede recorrer en ambos sentidos");
                System.out.println("es sensible
a las modificaciones realizadas por otros usuarios");
                break;
        }
        switch (rs.getConcurrency())
        {
            case ResultSet.CONCUR_READ_ONLY:
                System.out.println("los datos
albergados en el ResulSet están en modo sólo lectura");
                break;
            case ResultSet.CONCUR_UPDATABLE:
                System.out.println("los datos
albergados en el ResulSet son modificables");
                break;
        }
    } catch (SQLException e)
    {
        e.printStackTrace();
    }
}
```

a. Posicionamiento en un ResultSet

El objeto `ResultSet` gestiona un puntero de registros que determina sobre qué registro intervendrán los métodos ejecutados en el mismo `ResultSet`. Este registro se denomina a veces registro activo o registro actual. El objeto `ResultSet` contiene siempre dos registros ficticios que sirven de marca para el inicio del `ResultSet` (`BOF`) y el final del `ResultSet` (`EOF`). El puntero de registros puede posicionarse sobre uno de estos dos registros pero nunca delante del registro `BOF` ni después del registro `EOF`. Estos registros no contienen datos y una operación de lectura o de escritura sobre estos registros desencadena una excepción. En la creación del `ResultSet` el puntero se posiciona antes del primer registro (`BOF`). Hay muchos métodos disponibles para gestionar el puntero de registros.

`boolean absolute(int posicion)`: desplaza el puntero de registros al registro especificado. La numeración de los registros empieza por 1. Si el valor del parámetro es negativo el desplazamiento se realiza empezando por el final del `ResultSet`. Si no existe el número de registro, el puntero se posiciona en el registro `BOF` si el valor es negativo e inferior al número de registros o en el registro `EOF` si el valor es positivo y superior al número de registros. Este método devuelve `true` si el puntero se posiciona en un registro válido y `false` en caso contrario (`BOF` o `EOF`).

`boolean relative(int desplazamiento):` desplaza el cursor el número de registros especificado por el parámetro `desplazamiento`. Si el valor de este argumento es positivo, el cursor desciende en el conjunto de registros y su el valor es negativo el cursor asciende en el conjunto de registros. Este método devuelve `true` si el puntero se posiciona en un registro válido y `false` en caso contrario (BOF o EOF).

`void beforeFirst():` desplaza el puntero de registros hasta antes del primer registro (BOF).

`void afterLast():` desplaza el puntero de registros hasta después del último registro (EOF).

`boolean first():` desplaza el puntero de registros hasta el primer registro. Este método devuelve `true` si hay algún registro en el `ResultSet` y `false` en caso contrario.

`boolean last():` desplaza el puntero de registros al último registro. Este método devuelve `true` si hay algún registro en el `ResultSet` y `false` en caso contrario.

`boolean next():` desplaza el puntero de registros al registro siguiente respecto al actual. Este método devuelve `true` si el puntero de registros está en un registro válido y `false` en el caso contrario (EOF).

`boolean previous():` desplaza el puntero de registros al registro anterior respecto al actual. Este método devuelve `true` si el puntero de registros está en un registro válido y `false` en el caso contrario (BOF).

Para todos estos métodos salvo para el método `next` se requiere que el `ResultSet` sea de tipo `SCROLL_SENSITIVE` o `SCROLL_INSENSITIVE`. Si el `ResultSet` es de tipo `FORWARD_ONLY` sólo el método `next` funciona y en este caso el resto de métodos desencadenan el lanzamiento de una excepción. Los métodos siguientes permiten comprobar la posición del puntero de registros.

`boolean isBeforeFirst():` devuelve `true` si el puntero está colocado antes del primer registro (BOF).

`boolean isAfterLast():` devuelve `true` si el puntero está colocado después del último registro (EOF).

`boolean isFirst():` devuelve `true` si el puntero está colocado en el primer registro.

`boolean isLast():` devuelve `true` si el puntero está colocado en el último registro.

`int getRow():` devuelve el número del registro sobre el que se encuentra el puntero de registros. Se devuelve 0 si no hay registro actual (BOF o EOF).

```
public static void posicionRs(ResultSet rs)
{
    try {
        if (rs.isBeforeFirst())
        {
            System.out.println("el puntero está antes
del primer registro");
        }
        if (rs.isAfterLast())
        {
            System.out.println("el puntero está después
del último registro");
        }
        if (rs.isFirst())
        {
            System.out.println("el puntero está en el
primer registro");
        }
        if (rs.isLast())
        {
            System.out.println("el puntero está en el
último registro");
        }
        int posicion;
        posicion=rs.getRow();
        if (posicion!=0)
        {
            System.out.println("es el registro
número " + posicion);
        }
    } catch (SQLException e) {
        // TODO Bloque catch auto-generado
        e.printStackTrace();
    }
}
```

b. Lectura de datos en un ResultSet

El objeto `ResultSet` proporciona muchos métodos que permiten la lectura de campos de un registro. Cada uno de estos métodos tiene un tipo de datos SQL. Hay que usar prioritariamente el método adaptado al tipo de campo cuyo valor se desea obtener. Sin embargo, algunos de estos métodos son relativamente flexibles y permiten la lectura de varios tipos de datos. La tabla mostrada a continuación recoge los principales tipos de datos SQL y los métodos que realizan la lectura de un `ResultSet`. Los métodos marcados con el símbolo ☺ son los métodos recomendados. Los métodos marcados con el símbolo ☹ son posibles pero con riesgos de pérdida de información.

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	STRUCT	JAVA OBJECT	
getByte	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getShort	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getInt	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getLong	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getFloat	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getDouble	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getBigDecimal	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getBoolean	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺										
getString	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺		
getBytes														☺	☺	☺										
getDate											☺	☺	☺				☺	☺	☺							
getTime											☺	☺	☺				☺	☺	☺							
getTimeStamp											☺	☺	☺				☺	☺	☺							
getAsciiStream											☺	☺	☺	☺	☺	☺	☺									
getUnicodeStream											☺	☺	☺	☺	☺	☺	☺									
getBinaryStream												☺	☺	☺	☺	☺	☺									
getBlob																		☺								
getBlob																			☺							
getArray																				☺						
getRef																					☺					
getCharacterStream														☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	
getObject	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	☺	

Cada uno de estos métodos está disponible en dos formas. La primera acepta como argumento el número de la columna cuyo valor se desea obtener. La numeración comienza por 1. La segunda versión acepta una cadena de caracteres que representa el nombre de la columna en la base de datos. Si la instrucción que ha sido utilizada para crear el `ResultSet` contiene alias entonces las columnas tienen el nombre del alias y no el nombre del campo en la base de datos. Para una mejor legibilidad del código, evidentemente es preferible usar los nombre de las columnas en lugar de los índices. Cuando en la base de datos un campo no contiene ningún valor (`DBNull`) los métodos devuelven un valor igual a 0 para los campos numéricos, un valor `false` para los campos booleanos y un valor `null` para el resto de tipos. En algunos casos, hay por tanto una duda posible sobre el valor real del campo en la base de datos. Por ejemplo el método `getInt` puede devolver un valor igual a cero porque este valor está efectivamente en la base de datos o porque este campo no se ha informado en la base de datos. Para salir de dudas, el método `wasNull` devuelve un booleano igual a `true` si el campo sobre el que la última operación de lectura en el `ResultSet`

conténía efectivamente el valor null.

```
public static void lecturaRs(Connection cnx)
{
    Statement stm;
    String instrucion;
    ResultSet rs;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
        instrucion="select * from products ";
        rs=stm.executeQuery(instrucion);
        System.out.println("codigoProducto
\tnombre\tprecio unitario\tcantidad\tagotado\tfechaCaducidad");
        while(rs.next())
        {
            System.out.print(rs.getInt("ProductID")+"\t");
            System.out.print(rs.getString("ProductName")+"\t");
            System.out.print(rs.getDouble("UnitPrice")+"\t");
            rs.getShort("UnitsInStock");
            if (rs.wasNull())
            {
                System.out.print("desconocido\t");
            }
            else
            {
                System.out.print(rs.getShort("UnitsInStock")+"\t");
            }
            System.out.print(rs.getBoolean("Discontinued")+"\t");
            if (rs.getDate("DLC")!=null)
            {
                System.out.println(rs.getDate("DLC"));
            }
            else
                System.out.println("no perecedero");
        }
        rs.close();
        stm.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
```

c. Modificación de los datos en un ResultSet

La modificación de los datos se realiza simplemente utilizando los métodos updateXXX donde XXX corresponde al tipo de datos de la columna que se actualizará. Como sucede con los métodos getXXX, éstos están disponibles en dos versiones, una espera por parámetro el índice de la columna que se actualizará y la segunda espera por parámetro una cadena de caracteres que representa el nombre de la columna en la base de datos. Si la instrucción que se ha usado para crear el ResultSet contiene alias entonces las columnas tienen el nombre del alias y no el nombre del campo en la base de datos. Para una mejor legibilidad del código, es preferible usar los nombres de las columnas en lugar de sus índices. El tipo del segundo parámetro esperado por estos métodos se corresponde al tipo de datos que se actualizará en el ResultSet. Las modificaciones tienen que validarse a continuación con el método updateRow o cancelarlas con el método cancelRowUpdates. El ResultSet tiene que ser necesariamente del tipo CONCUR_UPDATABLE para poderse modificar. En caso contrario, la ejecución del método updateXXX desencadena el lanzamiento de una excepción.

```
public static void modificacionRs(Connection cnx)
{
    Statement stm;
    String instrucion;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String respuesta;
```

```

try
{
    stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
    instrucion="select * from products ";
    rs=stm.executeQuery(instrucion);
    System.out.println("número de fila\tcódigo del
producto\tnombre\tprecio unitario\tcantidad\ttagotado\tfechaCaducidad");
    while(rs.next())
    {
        num++;
        System.out.print(num + "\t");
        System.out.print(rs.getInt("ProductID")+"\t");
        System.out.print(rs.getString("ProductName")+"\t");
        System.out.print(rs.getDouble("UnitPrice")+"\t");
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("desconocido\t");
        }
        else
        {
            System.out.print(rs.getShort("UnitsInStock")+"\t");
        }
        System.out.print(rs.getBoolean("Discontinued")+"\t");
        if (rs.getDate("DLC")!=null)
        {
            System.out.println(rs.getDate("DLC"));
        }
        else
            System.out.println("no perecedero");
    }
    br=new BufferedReader(new InputStreamReader(System.in));
    System.out.println("¿qué fila desea modificar? ");
    respuesta=br.readLine();
    rs.absolute(Integer.parseInt(respuesta));
    System.out.println("nombre actual " +
rs.getString("ProductName"));
    System.out.println("introduzca el nuevo valor o enter
para conservar el valor actual");
    respuesta=br.readLine();
    if (!respuesta.equals(""))
    {
        rs.updateString("ProductName",respuesta);
    }
    System.out.println("precio unitario " +
rs.getDouble("UnitPrice"));
    System.out.println("introduzca el nuevo valor o enter
para conservar el valor actual");
    respuesta=br.readLine();
    if (!respuesta.equals(""))
    {
        rs.updateDouble("UnitPrice",Double.parseDouble(respuesta));
    }
    rs.getShort("UnitsInStock");
    if (rs.wasNull())
    {
        System.out.println ("cantidad en stock
actual desconocida");
    }
    else
    {
        System.out.println("cantidad en stock actual "
+ rs.getShort("UnitsInStock"));
    }
    System.out.println("introduzca el nuevo valor o enter
para conservar el valor actual");
    respuesta=br.readLine();
    if (!respuesta.equals(""))

```

```

        {
            rs.updateShort("UnitsInStock", Short.parseShort(respuesta));
        }
        System.out.println("¿desea validar estas
modificaciones s/n");
        respuesta=br.readLine();
        if (respuesta.toLowerCase().equals("s"))
        {
            rs.updateRow();
        }
        else
        {
            rs.cancelRowUpdates();
        }
        System.out.println("valores actuales ");
        System.out.print(rs.getString("ProductName")+"\t");
        System.out.print(rs.getDouble("UnitPrice")+"\t");
        rs.getShort("UnitsInStock");
        if (rs.wasNull())
        {
            System.out.print("desconocido\t");
        }
        else
        {
            System.out.print(rs.getShort("UnitsInStock")+"\t");
        }
        rs.close();
        stm.close();
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

d. Eliminación de datos en un ResultSet

La eliminación de una línea se realiza de forma muy sencilla posicionando el puntero en la línea que se desea eliminar y después invocando al método `deleteRow`. La línea se elimina inmediatamente del `ResultSet` y en la base de datos. La posición del puntero de registros después de la eliminación depende del driver de base de datos utilizado. Algunos desplazan el puntero al registro siguiente, otros lo desplazan al registro anterior y finalmente algunos no modifican la posición del puntero. En este caso hay que utilizar uno de los métodos de desplazamiento para posicionar el puntero en un registro utilizable. El `ResultSet` tiene que ser del tipo `CONCUR_UPDATABLE` para poder eliminar datos. En caso contrario, la ejecución del método `deleteRow` desencadena una excepción.

```

public static void eliminacionRs(Connection cnx)
{
    Statement stm;
    String instrucion;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String respuesta;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,Result
Set.CONCUR_UPDATABLE);
        instrucion="select * from products ";
        rs=stm.executeQuery(instrucion);
        System.out.println("número de fila\tcódigo del
producto\tnombre\tprecio unitario\tcantidad\tagotado\tfechaCaducidad");
        while(rs.next())
        {
            num++;
            System.out.print(num + "\t");
            System.out.print(rs.getInt("ProductID")+"\t");

```

```
System.out.print(rs.getString("ProductName")+"\t");
System.out.print(rs.getDouble("UnitPrice")+"\t");
rs.getShort("UnitsInStock");
if (rs.wasNull())
{
    System.out.print("desconocido\t");
}
else
{
    System.out.print(rs.getShort("UnitsInStock")+"\t");
}
System.out.print(rs.getBoolean("Discontinued")+"\t");
if (rs.getDate("DLC")!=null)
{
    System.out.println(rs.getDate("DLC"));
}
else
    System.out.println("no perecedero");
}
br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("¿qué fila desea eliminar?");
respuesta=br.readLine();
rs.absolute(Integer.parseInt(respuesta));
rs.deleteRow();
System.out.println("el puntero está ahora
en la línea " + rs.getRow());
}
catch (Exception e)
{
    e.printStackTrace();
}
```

e. Adición de datos en un ResultSet

Cada objeto `ResultSet` contiene una fila especial destinada a la inserción de datos. El puntero de registros tiene que posicionarse previamente en esta fila especial con la instrucción `moveToInsertRow`. Una vez que el puntero se ha posicionado en esta fila, ésta puede actualizarse con los métodos `updateXXX`. La inserción de la fila tiene que validarse a continuación con el método `insertRow`. Este método provoca la actualización de la base de datos. La fila de inserción se vuelve en ese momento una fila normal del `ResultSet` y el puntero de registros se posiciona sobre esta fila. Puede volver a posicionar el puntero de registros en la fila en que se encontraba antes de la inserción con el método `moveToCurrentRow`. Si no facilita los valores de todas las columnas, se insertarán valores `null` en la base de datos para las columnas no informadas. La base de datos tiene que aceptar valores nulos para esos campos, sino se desencadenará una excepción. El `ResultSet` tiene que ser del tipo `CONCUR_UPDATABLE` para poder insertar datos.

En caso contrario, la ejecución del método `moveToInsertRow` desencadena una excepción.

```
public static void adicionRs(Connection cnx)
{
    Statement stm;
    String instrucion;
    ResultSet rs;
    int num=0;
    BufferedReader br;
    String respuesta;
    try
    {
        stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
        instrucion="select * from products ";
        rs=stm.executeQuery(instrucion);
        br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("introduzca los valores para la
nueva fila");
        rs.moveToInsertRow();
        System.out.print("código del producto: ");
        respuesta=br.readLine();
        respuesta=respuesta.toUpperCase();
        rs.insertRow();
        num++;
        System.out.println("la fila se ha insertado con éxito");
    }
}
```

```

        rs.updateInt("ProductID", Integer.parseInt(respuesta));
        System.out.print("Nombre: ");
        respuesta=br.readLine();
        rs.updateString ("ProductName", respuesta);
        System.out.print("Precio unitario: ");
        respuesta=br.readLine();
        rs.updateDouble("UnitPrice",Double.parseDouble(respuesta));
        System.out.print("Cantidad en stock: ");
        respuesta=br.readLine();
        rs.updateDouble("UnitsInStock",Short.parseShort(respuesta));
        rs.insertRow();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

5. Gestión de transacciones

Las transacciones permiten garantizar que un conjunto de instrucciones SQL se ejecutará con éxito para cada una de ellas o que ninguna de ellas se ejecutará. La transferencia de una cantidad de dinero entre dos cuentas bancarias es el ejemplo clásico donde una transacción es necesaria. Imagine la situación siguiente: nuestro banco tiene que ejecutar el cobro de un cheque de 1000 € a débito desde la cuenta número 12345 y a la cuenta 67890. Por medidas de seguridad, para cada operación realizada en una cuenta (débito o crédito), se edita un informe. A continuación se muestra un extracto del código que puede realizar estas operaciones.

```

public static void movimiento(String cuentaDebito,String
cuentaCredito,double cantidad)
{
    try
    {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        Connection cnx=null;
        cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;da
tabaseName=banco; user=sa;password=");
        PreparedStatement stm;
        stm=cnx.prepareStatement("update cuentas set
saldo=saldo + ? where numero=?");
        stm.setDouble(1,cantidad * -1);
        stm.setString(2,cuentaDebito);
        stm.executeUpdate();
        impresionInforme(cuentaDebito, cantidad);
        stm.setDouble(1,cantidad);
        stm.setString(2,cuentaCredito);
        stm.executeUpdate();
        impresionInforme(cuentaCredito, cantidad);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

Para el 99,9999% de los movimientos efectuados con este código no hay ningún problema, hasta el día en que la impresora que se encarga de las ediciones se bloquea y este bloqueo lanza una excepción en el método impresionInforme. A priori, esta excepción no parece ser un problema debido a que la llamada a este método está dentro de un bloque try y hay un bloque catch que trata la excepción. Sin embargo, no hay que olvidar que si una excepción se desencadena y un bloque catch se ejecuta, la ejecución sigue por la instrucción siguiente al bloque catch. En este ejemplo, las instrucciones colocadas entre la desencadenante de la excepción y el final del bloque try simplemente no se ejecutarán. En nuestro caso, esto puede ser un problema si esta excepción se desencadena en el método impresionInforme ejecutado inmediatamente después de la operación de débito. La operación de crédito simplemente no tiene lugar y por tanto la cantidad se pierde. Una solución consiste en anular el efecto de las instrucciones SQL ya ejecutadas realizando el tratamiento inverso, en nuestro caso devolviendo el dinero en la cuenta.

De hecho, éste es el mecanismo que se implementa en una transacción pero evidentemente de forma automática. Es a nivel de conexión al servidor de base de datos donde se gestionan las transacciones.

a. Implementación de transacciones

Hasta ahora no nos habíamos preocupado por las transacciones y, sin embargo, nos damos cuenta que desde nuestra primera instrucción ejecutada con JDBC están ahí. El funcionamiento por defecto de JDBC consiste efectivamente en incluir cada instrucción ejecutada en una transacción y después validar la transacción si la instrucción se ha ejecutado correctamente (commit) o anular la instrucción en caso contrario (rollBack). A este modo de funcionamiento se le denomina modo autoCommit. Si desea gestionar usted mismo la finalización de una transacción validando o anulando todas las instrucciones que ésta contiene tiene que desactivar el modo autoCommit invocando el método `setAutoCommit(false)` en el objeto `Connection`. Usted será ahora el responsable de la finalización de las transacciones. Los métodos `commit` y `rollback` del objeto `Connection` permiten validar o anular las instrucciones ejecutadas desde el inicio de la transacción. Una nueva transacción comienza automáticamente a partir del final de la anterior o desde la apertura de la conexión. El código de nuestro método que permite transferir una cantidad entre dos cuentas tiene que tomar por lo tanto la forma siguiente.

```
public static void movimiento1(String cuentaDebito, String cuentaCredito, double cantidad)
{
    Connection cnx=null;
    try
    {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;databaseName=banco; user=sa;password=");
        cnx.setAutoCommit(false);
        PreparedStatement stm;
        stm=cnx.prepareStatement("update cuentas set saldo=saldo + ? where numero=?");
        stm.setDouble(1,cantidad * -1);
        stm.setString(2,cuentaDebito);
        stm.executeUpdate();
        impressionRapport(cuentaDebito, cantidad);
        stm.setDouble(1,cantidad);
        stm.setString(2,cuentaCredito);
        stm.executeUpdate();
        impressionRapport(cuentaCredito, cantidad);
        cnx.commit();
    }
    catch (Exception e)
    {
        try {
            cnx.rollback();
        }
        catch (SQLException e1)
        {
            e1.printStackTrace();
        }
        e.printStackTrace();
    }
}
```

b. Puntos de retorno

Cuando se invoca al método `rollback`, todas las instrucciones SQL ejecutadas desde el comienzo de la transacción se cancelan. Este método tiene una segunda versión que acepta por parámetro un objeto `SavePoint`. Este objeto representa un punto de retorno en la ejecución de instrucciones SQL. Se crea con el método `setSavePoint`. La llamada al método `rollback` con un parámetro del tipo `SavePoint` provoca la cancelación de todas las instrucciones hasta este punto de retorno.

c. Niveles de aislamiento

Mientras que una transacción está activa, la base de datos puede bloquear los datos modificados por las instrucciones ejecutadas en el interior de la transacción para evitar conflictos e incoherencias. Hay disponibles distintos tipos de bloqueos para una transacción. Para comprender bien sus efectos, previamente hay que identificar los problemas que pueden acaecer cuando una transacción está activa.

Lectura errónea: esta anomalía se produce cuando una aplicación accede a datos que están siendo modificados por

una transacción que todavía no se ha validado.

Lectura no reproducible: esta anomalía se produce cuando las ejecuciones sucesivas de una misma instrucción SELECT no producen el mismo resultado. Éste es el caso en que usted lee datos que están siendo modificados por otra transacción.

Lectura fantasma: esta anomalía se produce si ejecuciones sucesivas de una misma query reenvían datos de más o de menos. Esto puede ser debido a que otra transacción esté añadiendo o borrando datos de la tabla.

JDBC tiene previstos varios niveles de aislamiento. Estos niveles de aislamiento determinan la manera en la que se bloquean los datos durante la transacción. Este bloqueo puede imponerse en lectura, en escritura o incluso en lectura y escritura en los datos a los que acceden las instrucciones de la transacción.

El método `setTransactionIsolationLevel` permite fijar qué tipos de problemas queremos evitar. Este método acepta como argumento una de las constantes presentadas en la siguiente tabla. Esta tabla presenta además el efecto que tiene cada nivel de aislamiento sobre los datos.

Constante	Lectura errónea	Lectura no reproducible	Lectura fantasma
TRANSACTION_READ_UNCOMMITTED	possible	possible	possible
TRANSACTION_READ_COMMITTED	impossible	possible	possible
TRANSACTION_REPEATABLE_READ	impossible	impossible	possible
TRANSACTION_SERIALIZABLE	impossible	impossible	impossible

El hecho de especificar un nivel de aislamiento para una transacción puede a veces tener efecto en el resto de aplicaciones que acceden a los datos manipulados en la transacción debido a que pueden ser bloqueados y por tanto ser inaccesibles al resto de aplicaciones.

Pool de conexiones y Datasource

Cuando una aplicación clásica necesita acceder a una base de datos, hay que establecer una conexión con el servidor de base de datos. Por esta conexión van a transitar las instrucciones necesarias para realizar los tratamientos solicitados por el cliente. El caso de una aplicación web es similar pero con un pequeño detalle que tiene su importancia. La aplicación tiene que realizar los tratamientos para varios clientes simultáneamente con varias conexiones establecidas con el servidor de base de datos. Estando limitado el número de conexiones al servidor de base de datos, suele ser indispensable cerrar y abrir las conexiones en función de las necesidades. Lamentablemente esto consume muchos recursos y a veces se llega a una situación en la que no hay más conexiones disponibles. JDBC propone una solución eficaz a este problema.

1. Principio y utilización de un pool de conexiones

La solución propuesta por JDBC consiste en mantener abiertas un conjunto de conexiones que las aplicaciones pueden usar. Esta técnica se denomina pool de conexiones. Cuando la aplicación ya no tiene necesidad de seguir usando la conexión, la vuelve a poner en el pool. Los accesos son más rápidos porque no hay que esperar a la apertura de la conexión cada vez que tienen que usarse. Este mecanismo se implementa por varias clases que completan las clases JDBC básicas. El servidor de aplicaciones utiliza estas clases porque es él el que se encarga de gestionar este mecanismo. Las aplicaciones que deseen obtener una conexión a un servidor de base de datos tienen por tanto que realizar la petición al servidor indicando el nombre del recurso que desean obtener. El desarrollador de la aplicación y el administrador del servidor tienen que ponerse de acuerdo con el nombre usado. Esta solución presenta además la ventaja de que no hay que definir en la aplicación Java los parámetros de la conexión al servidor de base de datos. Estos datos suelen variar entre el servidor usado para hacer pruebas en la aplicación y el servidor de producción. No hay necesidad por tanto de tener que recompilar una parte de la aplicación justo antes de ponerla en producción únicamente para que cargue los parámetros de conexión a la base de datos.

2. Configuración de un pool de conexiones

La configuración del pool de conexiones tiene que realizarse en el mismo servidor. Esta configuración depende del tipo de servidor. Algunos servidores ofrecen una interfaz gráfica para esta configuración.

El ejemplo mostrado a continuación es un extracto del archivo server.xml de un servidor Tomcat.

```
<Resource  
    name="jdbc/northwindDB"  
    type="javax.sql.DataSource"  
    maxActive="10"  
    maxIdle="3"  
    maxWait="10000"  
    username="jefe"  
    password="password"  
    driverClassName="com.microsoft.sqlserver.jdbc.SQLServerDriver"  
    url="jdbc:sqlserver://localhost:1433;databaseName=northwind"  
/>
```

El atributo `name` de esta etiqueta indica el nombre del recurso en el servidor. Este es el nombre que tiene que usarse en el código para obtener el recurso del servidor. Este recurso se representa por una instancia de la clase `DataSource`. Ésta se crea en el arranque del servidor y después se coloca en el servicio de directorio asociado al servidor.

3. Utilización de un pool de conexiones

La utilización de un pool de conexiones es muy fácil de implementar. En primer lugar, la aplicación tiene que obtener una referencia a la raíz del servicio de directorio del servidor. Para ello, hay que crear un objeto `javax.naming.InitialContext`.

```
InitialContext raiz;  
raiz=new InitialContext();
```

A partir de este objeto `InitialContext` hay que buscar a continuación el recurso correspondiente al objeto `DataSource` instanciado por el servidor. Esta búsqueda se realiza por el nombre usado para identificar el recurso en la configuración del servidor. La norma Sun dicta que hay que anteponer al nombre del recurso la cadena "java:comp/env/".

La instrucción siguiente permite, por tanto, obtener una referencia al objeto `DataSource` proporcionado por el servidor.

```
DataSource ds;  
ds= (DataSource) raiz.lookup("java:comp/env/jdbc/northwind");
```

Para obtener una conexión, basta simplemente con solicitar al objeto `DataSource` que consiga una conexión con el método `getConnection`.

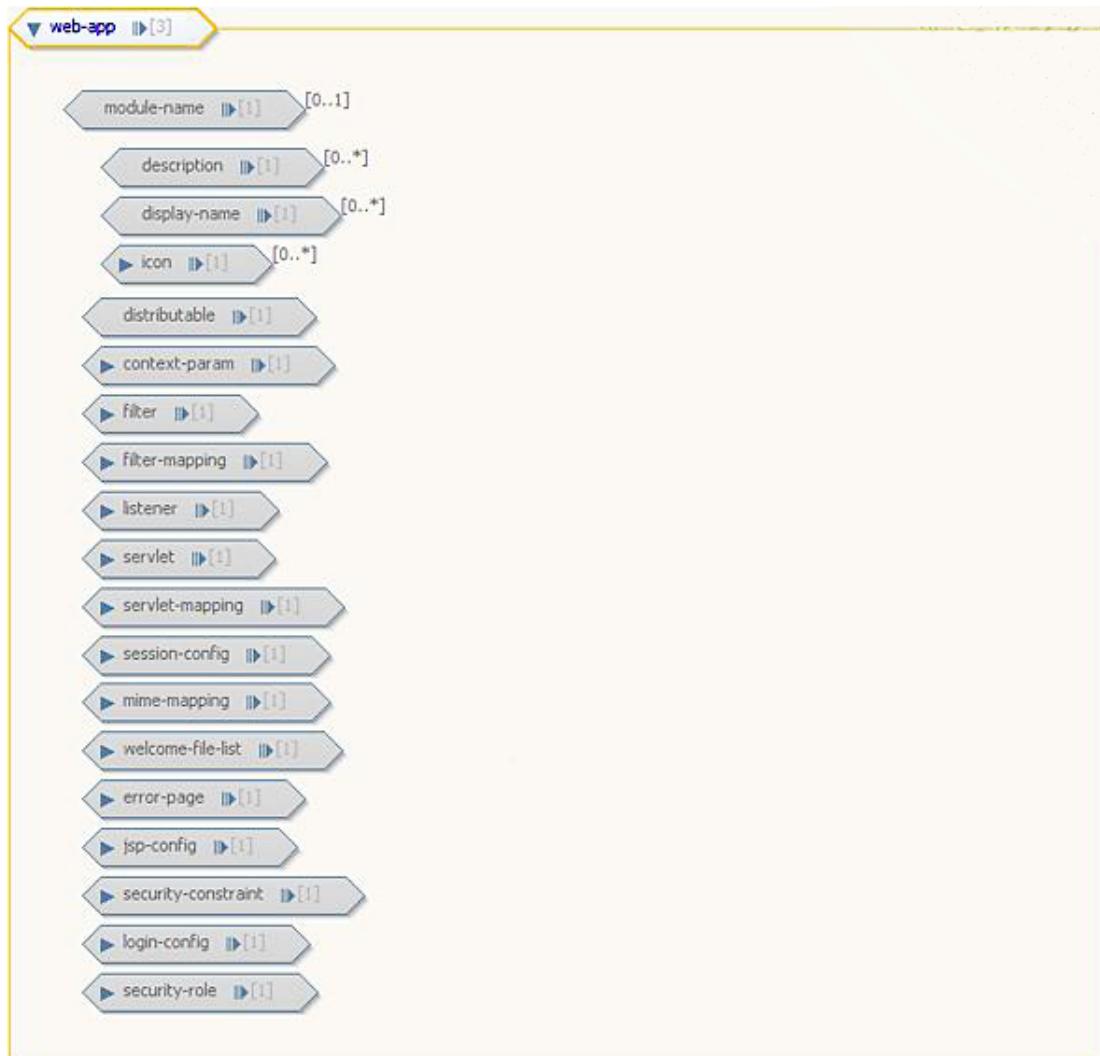
```
Connection cnx;  
cnx=ds.getConnection();
```

La conexión obtenida se puede usar directamente debido a que ya está abierta. Después de su uso la conexión tiene que devolverse al pool de conexiones para ser de nuevo usada por la aplicación. Hay que invocar al método `close` para volverla a colocar en el pool.

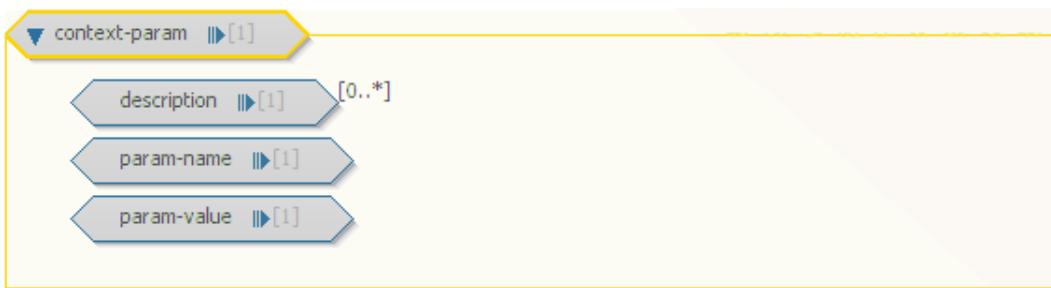
-
-  La llamada de este método no provoca el cierre de la conexión sino simplemente la vuelve a colocar en el pool.
-

No se debe olvidar llamar a este método ya que rápidamente el servidor corre el riesgo de quedarse sin conexiones disponibles si éstas no se devuelven inmediatamente después de su uso.

Estructura general del descriptor de despliegue



Sección context-param



Función de la etiqueta: definición de los parámetros de la aplicación.

Multiplicidad de esta etiqueta: [0 a N].

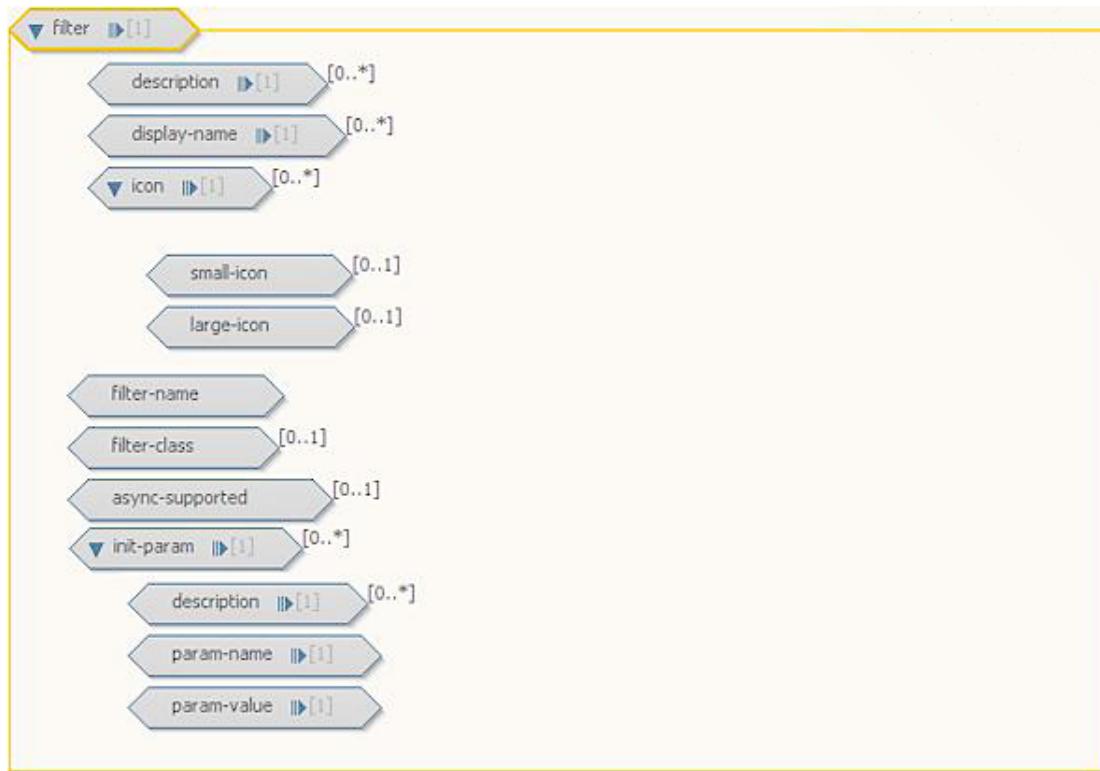
Elementos hijo:

<description> [0 a N]: comentario sobre la función del parámetro.

<param-name> [1]: nombre del parámetro.

<param-value> [1]: valor del parámetro.

Sección filter



Función de la etiqueta: definición de los filtros aplicados a las peticiones y respuestas HTTP de un servlet.

Multiplicidad de esta etiqueta: [0 a N].

Elementos hijo:

<description> [0 a N]: comentario sobre la función del filtro.

<display-name> [0 a N]: nombre que se usa para mostrar el filtro en una herramienta de desarrollo.

<small-Icon> y <large-Icon> [0 a N]: iconos que se usan para mostrar el filtro en una herramienta de desarrollo.

<filter-name> [1]: nombre del filtro.

<filter-class> [1]: clase Java del filtro.

<async-supported> [0 a 1]: indica si el filtro soporta los tratamientos asíncronos (true o false).

<init-param> [0 a N]: definición de los parámetros del filtro.

Elementos hijo:

<description> [0 a N]: comentario sobre la función del parámetro.

<param-name> [1]: nombre del parámetro.

<param-value> [1]: valor del parámetro.

Sección filter-mapping



Función de la etiqueta: definición de la asociación entre un filtro y un servlet.

Multiplicidad de esta etiqueta: [0 a N].

Elementos hijo:

<filter-name> [1]: nombre del filtro.

<url-pattern> [0 a N]: URL sobre la que el filtro se debe aplicar.

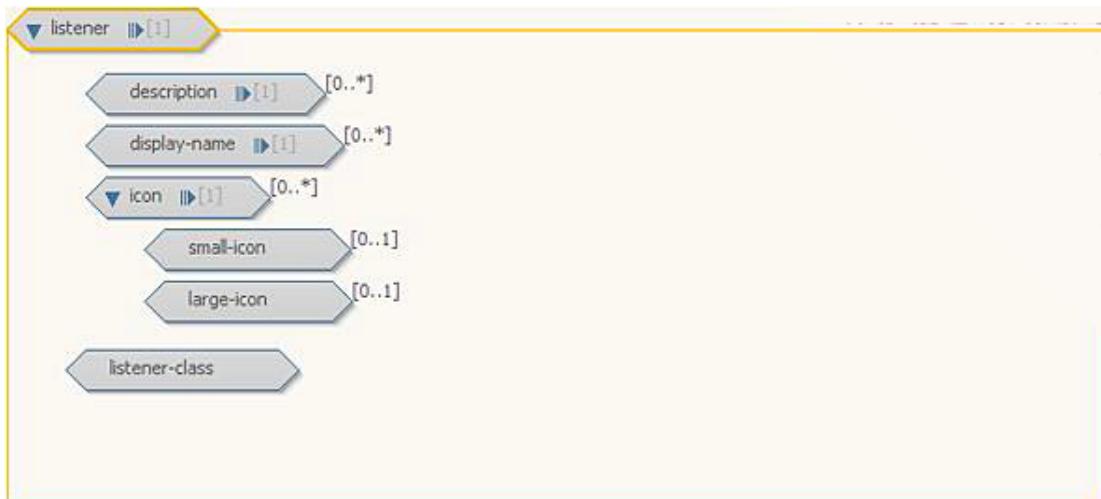
<servlet-name> [0 a 1]: nombre del servlet sobre el que el filtro se debe aplicar.

<dispatcher> [0 a 1]: origen de la petición sobre la que el filtro se debe aplicar.

Posibles valores:

REQUEST
FORWARD
INCLUDE
ERROR

Sección listener



Función de la etiqueta: declaración de los detectores de eventos de la aplicación.

Multiplicidad de esta etiqueta: [0 a N].

Elementos hijo:

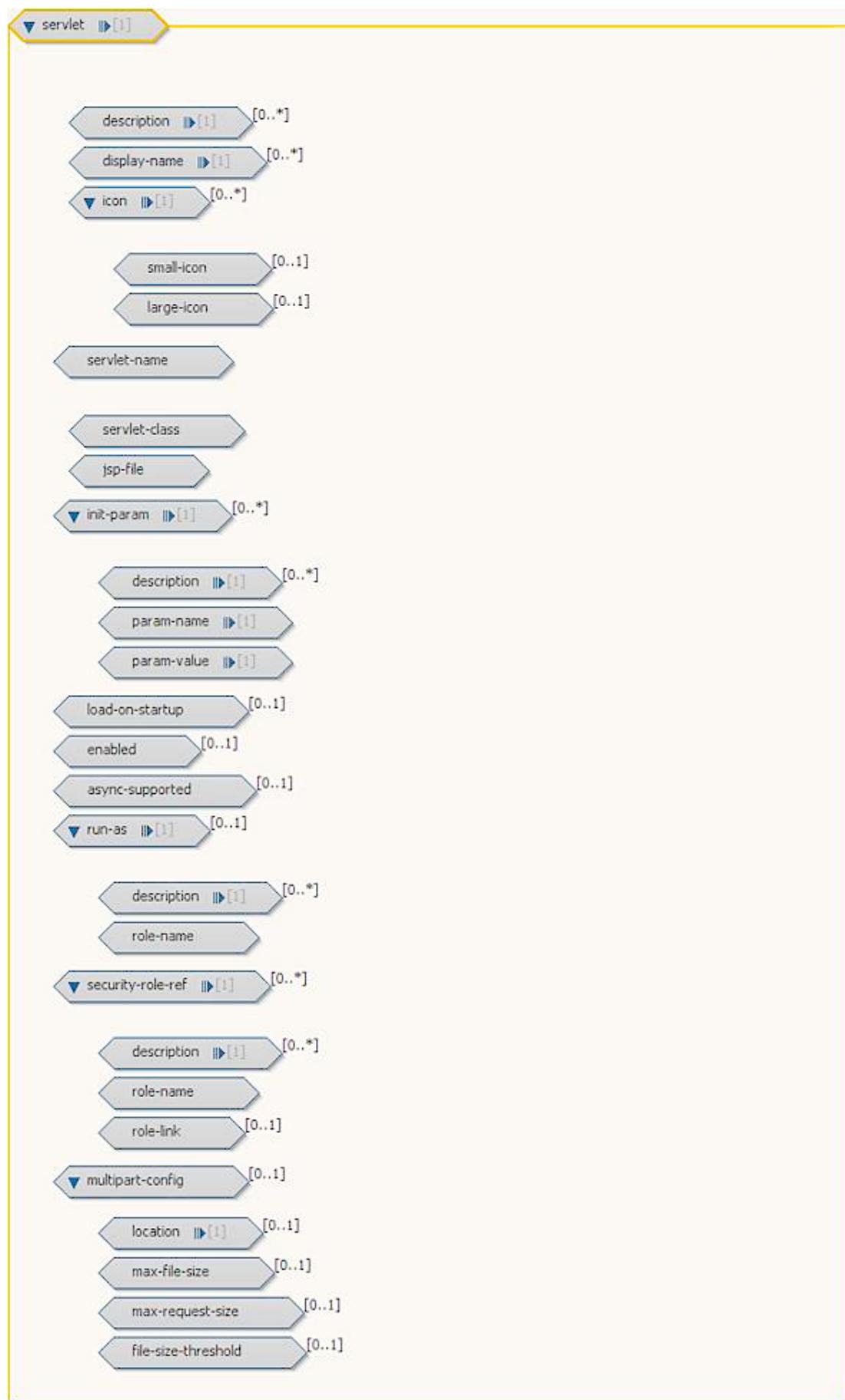
<description> [0 a N]: comentario sobre la función del detector.

<display-name> [0 a N]: nombre que se usa para mostrar el detector en una herramienta de desarrollo.

<small-Icon> y <large-Icon> [0 a N]: iconos que se usan para mostrar el detector en una herramienta de desarrollo.

<listener-class> [1]: clase Java del detector.

Sección servlet



Función de la etiqueta: declaración de los servlets de la aplicación.

Multiplicidad de esta etiqueta: [0 a N].

Elementos hijo:

<description> [0 a N]: comentario sobre la función del servlet.

<display-name> [0 a N]: nombre que se usa para mostrar el servlet en una herramienta de desarrollo.

<small-Icon> y <large-Icon> [0 a N]: iconos que se usan para mostar el servlet en una herramienta de desarrollo.

<servlet-name> [1]: nombre que permite identificar el servlet en el descriptor de despliegue.

<servlet-class> [0 a 1]: clase Java del servlet.

<jsp-file> [0 a 1]: nombre de la página JSP para la que se quiere configurar el servlet. Se debe usar en lugar del elemento <servlet-class>.

<init-param> [0 a N]: definición de los parámetros de inicialización del servlet.

Elementos hijo:

<description> [0 a N]: comentario sobre la función del parámetro.

<param-name> [1]: nombre del parámetro.

<param-value> [1]: valor del parámetro.

<load-on-startup> [0 a 1]: indica si el servlet se debe cargar durante el inicio del servidor. El valor numérico positivo de esta etiqueta indica la orden de carga del servlet.