

4 Un primer vistazo a los Servlets

4.1 El servidor de aplicaciones

Actualmente, cuando se programa una aplicación web normalmente se toma ventaja de un servidor de aplicaciones. Un servidor de aplicaciones es un software de infraestructura que proporciona una serie de servicios a aplicaciones que corren en su interior. Es un concepto similar al de sistema operativo: un sistema operativo es quien controla la ejecución de las aplicaciones en un ordenador, y además les proporciona una serie de servicios a estas aplicaciones (como por ejemplo, compartir la CPU y memoria RAM entre varios programas de un modo transparente para el programador, acceso al sistema de ficheros...). El servidor de aplicaciones va a ser quien ejecute y controle la ejecución de nuestras aplicaciones web. A su vez, nos prestará una serie de servicios como por ejemplo permitir almacenar estado entre distintas peticiones del cliente, facilitar el almacenamiento de información de modo persistente (en una base de datos, por ejemplo), repartir la carga de la aplicación web entre varios servidores, etc.

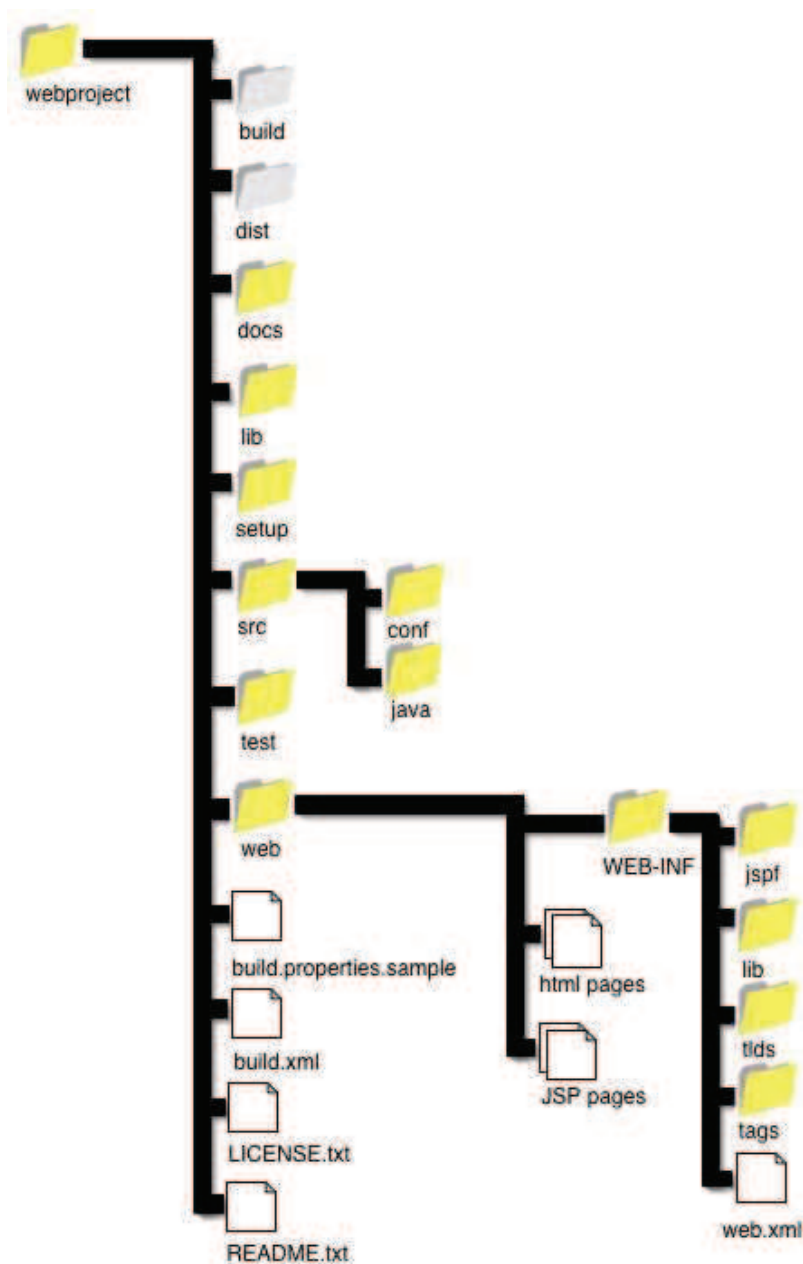
El servidor de aplicaciones será quien realmente reciba la petición http. Él analizará dicha petición y almacenará toda la información relativa a ella (como las cabeceras o la información de los campos de un formulario) en objetos Java. A continuación, si existe algún programa registrado para gestionar peticiones a la URL a la cual ha llegado la petición, invocará a dicho programa y le pasará como parámetros objetos Java que contienen toda la información relativa a la petición. Ese es el momento en el cual se comienza a ejecutar nuestro código fuente. Empleando el API que el servidor proporciona con este fin, nuestro programa Java podrá acceder a la información de la petición, procesarla, y generar la respuesta adecuada. Para generar la respuesta adecuada nuevamente el servidor nos proporciona un API compuesta por varios objetos Java. Nosotros invocaremos métodos sobre esos objetos Java, y el servidor finalmente generará una respuesta a la petición a partir de ellos, respuesta que enviará por la red.

En este tutorial emplearemos el servidor de aplicaciones Glassfish 3.1 junto con Netbeans 7.0 para la ejecución de los ejemplos. No obstante, la gran mayoría de la

información contenida en el tutorial es completamente independiente del servidor de aplicaciones Java EE que empleemos, o del entorno de desarrollo.

4.2 Estructura de directorios de una aplicación web

La imagen muestra la estructura de directorios recomendada para una aplicación web Java EE:



El directorio **build** es empleado para guardar la salida resultante de la compilación del código Java de la aplicación. El directorio **dist** se emplea para guardar el producto que finalmente será distribuido; esto es, la aplicación web al completo incluyendo los recursos compilados, librerías y los recursos estáticos que puedan formar parte de ella. El directorio **docs** se emplea para guardar el javadoc de la aplicación. El directorio **lib** para guardar librerías de terceras partes que estemos empleando en nuestra aplicación. El directorio **setup** se emplea para guardar archivos de configuración que puedan formar parte de nuestra aplicación. El código fuente debe estar colgando del directorio **src**, en una estructura de subdirectorios que refleje los distintos paquetes que se están empleando en el código fuente, al igual que sucede en cualquier aplicación Java.

Es habitual que en el directorio raíz de la aplicación encontremos ficheros con información acerca de la licencia de la aplicación e información de cómo usarla. También es aquí donde a menudo se encuentra un script de la herramienta de compilación Apache Ant (<http://ant.apache.org/>) para trabajar con el proyecto (el fichero **build.xml**).

El directorio más importante posiblemente sea el directorio **web**. De este directorio cuelga la aplicación web al completo; el contenido de este directorio (posiblemente conveniente empaquetado en un fichero war) es lo que nuestro entorno de desarrollo o Ant copiarán al directorio **dist**. En la raíz de este directorio, o bien directamente en una estructura de directorios a nuestro criterio, debemos colocar el contenido estático de nuestra aplicación como, por ejemplo, imágenes, hojas de estilo, páginas web HTML estáticas...

Del directorio **web** cuelga un directorio con nombre **WEB-INF**. En la raíz de este directorio podemos encontrar el descriptor de despliegue de la aplicación; esto es, un fichero XML con nombre **web.xml** que contiene distintos parámetros de configuración de nuestra aplicación (a lo largo de este tutorial veremos qué información contiene dicho fichero).

Del directorio **WEB-INF** también debe colgar el resultado de compilar nuestro código Java, empleando la adecuada estructura de directorios según los paquetes que emplee nuestro código. Es decir, aquí es donde debemos colocar el contenido del directorio **build**.

4.3 Nuestro primer Servlet

Los Servlets son los programas Java que corren dentro de un servidor de aplicaciones Java EE. A menudo a la parte web de un servidor de aplicaciones Java EE se le llama contenedor de Servlets, ya que lo que hace es "contener" dentro estos programas. Cuando al contenedor le llega una petición http, comprueba si hay algún Servlet registrado para responder a dicha petición; en caso afirmativo, invocará dicho código. Hasta Java EE 6 (que incorporó la versión 3.0 de la especificación de los Servlets) todos los Servlets debían extender la clase `HttpServlet`. En Java EE 6 sigue estando disponible esta opción, pero también es posible convertir en un Servlet a cualquier POJO (Plain Old Java Object) simplemente anotándolo.

La clase `HttpServlet` define (entre otros) dos métodos: `doGet` y `doPost`; si la petición que ha llegado para el Servlet era una petición tipo GET invocará al primer método y si era tipo POST invocará al segundo método.

Veamos nuestro primer Servlet:

```
package tema4;

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

@WebServlet(name="HolaMundoServlet", urlPatterns={"/HolaMundo"})
public class HolaMundoServlet extends HttpServlet {

    @Override
    protected void doGet(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>Hola mundo</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hola mundo!</h1>");
        out.print("<img src=\"./logojH.jpg\"></img>");
        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}
}

```

El método doGet es invocado cuando una petición tipo GET para la cual se haya registrado nuestro Servlet llega al servidor. La línea que indica qué peticiones quiere responder nuestro Servlet es la de la anotación:

```
@WebServlet(name="HolaMundoServlet", urlPatterns={"/HolaMundo"})
```

en esta línea se le está dando el nombre HolaMundoServlet al Servlet y se indica que el Servlet quiere registrarse para responder a las peticiones del tipo

```
http://máquina:puerto/contexto/HolaMundo
```

al indicar los patrones de URL a los cuales desea responder el Servlet es posible emplear *; por ejemplo, si se registra para responder a:

```
@WebServlet(name="HolaMundoServlet", urlPatterns={"/HolaMundo/*"})
```

sería invocado por el contenedor para todas las siguientes peticiones:

```
http:// máquina:puerto/contexto/HolaMundo/synopsis
http:// máquina:puerto/contexto/HolaMundo/complete?date=today
http:// máquina:puerto/contexto/HolaMundo
```

También es posible indicar múltiples patrones de URL a los cuales va a responder; por ejemplo:

```
@WebServlet (name="HolaMundoServlet", urlPatterns={"/HolaMundo/*",
".saludo"})
```

haría que el Servlet respondiese a todos los ejemplos anteriores y además a URLs como:

```
http:// máquina:puerto/contexto/Hola.saludo
http://
máquina:puerto/contexto/directorio/directorio2/compl?date=today.saludo
http:// máquina:puerto/contexto/directorio/fichero.saludo
```

Toda esta información también puede especificarse en el descriptor de despliegue de la aplicación web. De hecho, hasta la especificación 3.0 de los Servlets, ésta era la única forma de especificar esta información. Este descriptor de despliegue es un fichero con nombre web.xml que debe de situarse en la raíz del directorio WEB-INF. Si quisiésemos eliminar la anotación de nuestro Servlet y especificar toda esta información en el descriptor de despliegue el contenido de dicho descriptor debería ser:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <servlet>
```

```

        <servlet-name>HolaMundoServlet</servlet-name>
        <servlet-class>tema4.HolaMundoServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HolaMundoServlet</servlet-name>
        <url-pattern>/HolaMundo</url-pattern>
    </servlet-mapping>
</web-app>

```

La etiqueta `<servlet>` permite registrar un Servlet en el contenedor. Dentro de esta etiqueta deberemos colocar la etiqueta `<servlet-name>`, que indica el nombre del Servlet, y la etiqueta `<servlet-class>`, que indica la clase a la que pertenece; la clase debe especificarse indicando el nombre completo (con el paquete). La etiqueta `<servlet-mapping>` es la que permite especificar a qué URLs va a responder cada Servlet. Dentro de ella debemos indicar el nombre del Servlet y el patrón de URL al cual queremos que éste responda.

El descriptor de despliegue puede sobrescribir lo que nosotros hayamos indicado en el código fuente de los Servlet mediante anotaciones. Este es un comportamiento bastante útil. En el código fuente del Servlet podemos especificar unos patrones de URL por defecto de un modo sencillo y menos verboso que con XML empleando una anotación. Una vez terminada la aplicación, si queremos cambiar estos patrones de URL sería bastante tedioso cambiar la anotación: requeriría modificar el código fuente de la aplicación, volverla a compilar y desplegarla en el servidor de aplicaciones. Otra opción es simplemente modificar el descriptor de despliegue de la aplicación web. Lo que indiquemos en este descriptor va a sobrescribir a las anotaciones. Por ejemplo, si en nuestro ejemplo en el descriptor de despliegue usamos:

```

<servlet-mapping>
    <servlet-name>HolaMundoServlet</servlet-name>
    <url-pattern>/Hola</url-pattern>
</servlet-mapping>

```

ahora una petición a `http://máquina:puerto/contexto/HolaMundo` devolverá un error 404 (el recurso no está disponible). Si queremos acceder a nuestro Servlet deberemos acceder a: `http://máquina:puerto/contexto/Hola`, independientemente de lo que hayamos especificado en la anotación.

Volvamos al código fuente de `HolaMundoServlet`. El método `doGet` toma dos parámetros:

```
doGet (HttpServletRequest request, HttpServletResponse response)
```

el primero es un objeto Java que envuelve la petición que ha llegado al servidor. Empleando métodos de este objeto, podremos acceder a información como a las cabeceras de la petición, parámetros de un formulario si la petición se corresponde con el envío de un formulario, etcétera. En nuestro ejemplo, este objeto no se usa absolutamente para nada. El segundo objeto será el que el servidor web emplee para generar la respuesta. Habitualmente, se genera obteniendo un objeto tipo `PrintWriter` contenido en el objeto `HttpServletResponse` y empleándolo para generar la respuesta. Un `PrintWriter` proporciona una serie de métodos que permiten escribir a un cauce de salida de texto. Como se puede ver en el código de `HolaMundoServlet`, lo que hacemos es generar una página web estática que incluye un saludo y una imagen. La imagen debe estar situada en la raíz del directorio `./WEB-INF` para poder acceder a ella empleando la ruta indicada en el código.

Si desplegamos este Servlet en Glassfish empleando Netbeans, Netbeans abrirá nuestro navegador por defecto y nos mostrará esta página:



Observa la URL que se muestra en el navegador. La máquina es localhost. El protocolo (aunque Chrome lo omite) http. El puerto en el cual está el servidor es el puerto 8080. La aplicación está corriendo dentro del contexto "Servlet1" del servidor de aplicaciones. Y el recurso de la aplicación al cual estamos accediendo es HolaMundo. Aunque en estas capturas de pantalla el contexto de la aplicación estaba configurado para ser "Servlet1", cuando empaqueté el código fuente del curso cambié el contexto "contexto". La URL que tú deberás usar por tanto para acceder a este ejemplo es:

```
http://localhost:8080/contexto/HolaMundo
```



Video

https://www.youtube.com/watch?v=v0cKNDOHu_M

Este sería un buen momento para ver el segundo video que acompaña a este tutorial; en él se te explicará cómo crear Servlets, como ejecutarlos desde Netbeans, como para el servidor de aplicaciones, como depurarlos empleando del debugger de Netbeans, etc.

4.4 HttpServletRequest y HttpServletResponse

En este punto del tutorial recomiendo al lector que examine el javadoc de `javax.servlet.HttpServletRequest` y de `javax.servlet.HttpServletResponse`. Aquí simplemente vamos a presentar brevemente algunos de sus métodos que se emplean con más frecuencia. Para `HttpServletRequest` estos métodos son:

- **Object `getAttribute(String name)`:** devuelve el atributo asociado con el nombre que se le pasa con el argumento, o null si no hay ningún atributo asociado con dicho nombre. Veremos su utilidad en capítulos posteriores.
- **java.util Enumeration `getAttributeNames()`:** devuelve una enumeración que contiene los nombres de los atributos presentes en la petición.
- **String `getParameter(String name)`:** devuelve el parámetro asociado con el nombre que se le pasa como argumento, o null si dicho parámetro no existe. Veremos su utilidad en capítulos posteriores.
- **java.util.Map `getParameterMap()`:** devuelve un mapa con los parámetros de la petición.
- **String[] `getParameterValues(String name)`:** devuelve un array de cadenas de caracteres conteniendo todos los valores asociados con un determinado parámetro, o null si dicho parámetro no existe. Veremos su utilidad en capítulos posteriores.
- **String `getContextPath()`:** devuelve el fragmento de la URL que indica el contexto de la petición. En un mismo servidor de aplicaciones puede haber desplegadas varias aplicaciones Java EE. El servidor se encarga de aislar las unas de las otras, de tal modo que no puedan compartir información y que un error en una de ellas no pueda afectar a las demás. Cada una de estas aplicaciones tiene su propio contexto, y ese contexto es el primer fragmento de la URL que va justo después del nombre del servidor. Después del contexto, en la URL va la ruta del fichero o recurso al que queremos acceder en nuestra aplicación.

- **Cookie[] getCookies():** devuelve las cookies asociadas con esta petición. En un tema posterior hablaremos más en detalle de las cookies.
- **String getHeader(String name):** devuelve el contenido asociado con la cabecera http especificada en el parámetro.
- **java.util.Enumeration<String> getHeaderNames():** devuelve una enumeración conteniendo todas las cabeceras http de la petición.
- **java.util.Enumeration<String> getHeaders(String name):** devuelve todos los valores asociados con una determinada cabecera http.
- **int getIntHeader(String name):** devuelve el valor de una determinada cabecera http como un entero.
- **String getMethod():** indica el método http empleado en la petición; por ejemplo GET, POST, o PUT.
- **HttpSession getSession():** devuelve la sesión asociada con esta petición. Si la petición no tiene una sesión asociada, crea una. En un tema más adelante hablaremos más en detalle acerca de las sesiones.
- **HttpSession getSession(boolean create):** similar al método anterior, pero sólo crea la sesión si se le pasa el parámetro true.

Los métodos más comúnmente empleados de HttpServletResponse son:

- **java.io.PrintWriter getWriter() throws java.io.IOException:** devuelve un objeto PrintWriter asociado con la respuesta que será enviada al cliente. Habitualmente, crearemos la página web que se va a mostrar al usuario escribiendo contenido en el cauce de salida representado por este objeto.
- **void setContentType(String type):** establece el tipo de contenido de la respuesta que se va a enviar al cliente. Por ejemplo, "text/html;charset=UTF-8" para indicar que vamos a generar una página web empleando UTF-8 para codificar los caracteres, o "text/plain;charset=UTF-8" para indicar que lo que vamos a generar es texto plano empleando el mismo encoding que en el caso anterior.

- **void addCookie(Cookie cookie):** añade una cookie a la respuesta.
- **void addHeader(String name, String value):** añade una nueva cabecera, con el valor especificado en el segundo parámetro, a la respuesta.
- **boolean containsHeader(String name):** comprueba si la respuesta contiene o no una determinada cabecera.
- **String encodeRedirectURL(String url):** prepara una URL para ser usada por el método sendRedirect. Más adelante abordaremos en más profundidad este método.
- **String getHeader(String name):** devuelve el valor de la cabecera http especificada.
- **java.util.Collection<String> getHeaderNames():** devuelve los nombres de las cabeceras http de la respuesta.
- **java.util.Collection<String> getHeaders(String name):** devuelve los valores asociados con una determinada cabecera http.
- **void sendError(int sc):** envía al servidor una respuesta con el código de error especificado.
- **void sendRedirect(String location):** redirecciona al cliente a una determinada a URL.
- **ServletOutputStream getOutputStream():** devuelve un objeto tipo ServletOutputStream; empleando este objeto es posible construir Servlets que generen respuestas binarias (por ejemplo, una imagen) y no sólo texto.

A continuación mostramos el código de un Servlet que genera una respuesta al usuario donde se muestran todas las cabeceras de la petición http que ha recibido y el primero de sus valores (es posible que la cabecera tenga múltiples valores asociados; existe un método que permite obtener todos los valores, pero el código emplea un método que simplemente devuelve el primero de dichos valores; dejo como ejercicio para el lector el mostrar todos los valores). El patrón de URL asociado con el ejemplo es /cabeceras.

Este Servlet ha sobrescrito los métodos doGet y doPost de tal modo que ambos invocan al método processRequest pasándole los parámetros con los que han sido

invocados. Aunque un Servlet permite responder tanto a peticiones GET como POST (es más, la clase HTTP Servlet también define métodos que son invocados por peticiones tipo DELETE, HEAD, OPTIONS, PUT y TRACE) lo habitual es que un Servlet realice una única tarea, independientemente del método de la petición http que lo ha alcanzado. Por ello es una práctica común sobrescribir ambos métodos y reenviar la petición a un único método que realizará la misma tarea independientemente del método http empleado.

```
...
@WebServlet(name="CabecerasServlet", urlPatterns={"/cabeceras"})
public class CabecerasServlet 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 CabecerasServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Cabeceras: </h1>");

            out.println("<ul>");
            Enumeration<String> nombresDeCabeceras =
request.getHeaderNames();
            while (nombresDeCabeceras.hasMoreElements()) {
                String cabecera = nombresDeCabeceras.nextElement();
                out.println("<li><b>" + cabecera + ": </b>"
                    + request.getHeader(cabecera) + "</li>");
            }
            out.println("</ul>");
            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);
    }
}

```

Observa como en esta ocasión parte de la salida que genera el Servlet no es estática (como es el caso de la salida generada por HolaMundoServlet, que nunca cambia) sino que texto estático se combina con variables Java que pueden tener valores diferentes, generando una página dinámica que puede cambiar en función de los valores de dichas variables:

```

out.println("<li><b>" + cabecera + ": </b>"
    + request.getHeader(cabecera) + "</li>");

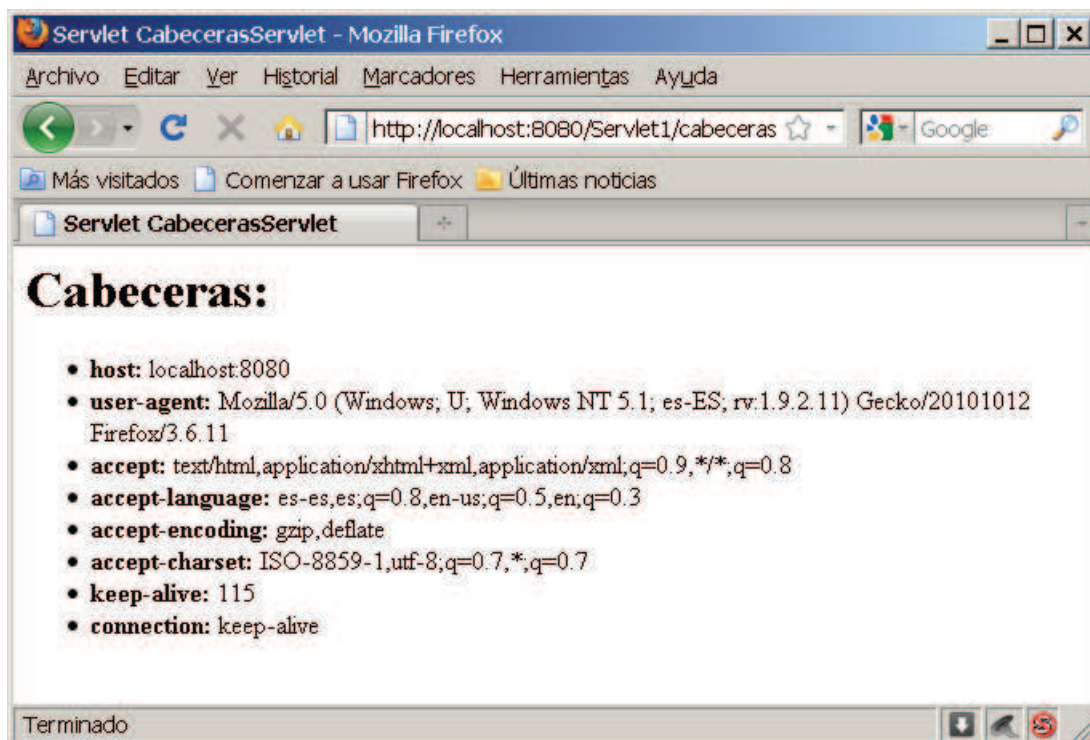
```

El HolaMundoServlet, más allá de ser un ejemplo académico, no tendría absolutamente ningún sentido ya que la salida que genera es completamente estática en el sentido de que no cambia; es siempre la misma. Para eso sería mucho más fácil haber empleado una página HTML estática. Sin embargo, lo que hace CabecerasServlet nunca lo podríamos replicar con una página HTML.

Si accedemos a CabecerasServlet empleando Google Chrome veremos algo como esto:

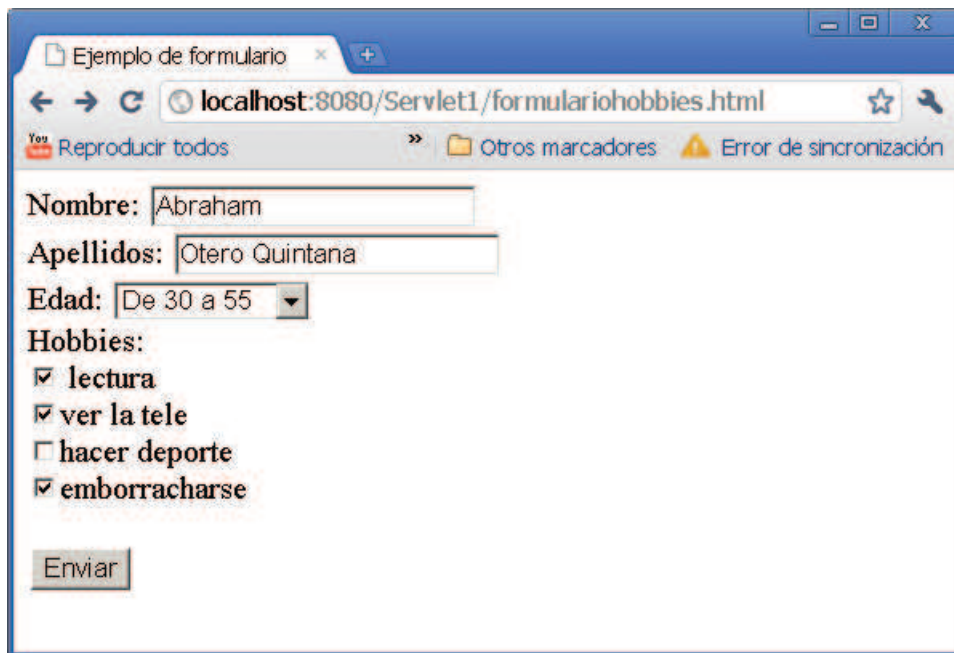


Mientras que si accedemos empleando Firefox veremos algo como esto:



4.5 Un Servlet que procesa un formulario

Lo más habitual es emplear un Servlet para procesar información que el cliente nos está mandando, y el cliente habitualmente envía información a través de formularios. Supongamos que tenemos el siguiente (extremadamente cutre) formulario:



Este formulario deberá colocarse en la raíz del directorio WEB-INF; en el proyecto de Netbeans que acompaña a este tutorial es el archivo que está dentro del directorio web, en el subdirectorio tema4, y cuyo nombre es formulariohobbies.html. El código HTML del formulario es:

```
<form method="get" action="/contexto/formulario" name="datos">
Nombre: <input name="nombre"><br>
Apellidos: <input name="apellidos"><br>
Edad:
  <select name="edad">
    <option>Menor de 18</option>
    <option>De 18 a 30</option>
    <option>De 30 a 55</option>
```



```

        <option>Mayor de 55</option>
    </select>
    <br>
Hobbies:<br>
    <input name="hobbies" value="lectura" type="checkbox"> lectura<br>
    <input name="hobbies" value="tele" type="checkbox">ver la tele<br>
    <input name="hobbies" value="deporte" type="checkbox">hacer
deporte<br>
    <input name="hobbies" value="emborracharse"
type="checkbox">emborracharse<br>
    <br>
    <button>Enviar</button></form>

```

Observa que el formulario va a enviar sus datos a la URL formada al retirar el nombre del formulario de la URL de la página web donde se aloja este, y añadir "/formulario". Por ejemplo, si el formulario estaba en `http://localhost:8080/Contexto/formulariohobbies.html`, la URL a la cual se enviará la petición es: `http://localhost:8080/contexto/formulario` empleando el método GET. En esa URL debe haber un Servlet preparado para procesar la información. Cuando este formulario envía todos sus parámetros se enviarán como parte de la URL, ya que hemos empleado el método GET; al rellenar el formulario tal y como se muestra en la figura de la página anterior ésta es la URL que obtendríamos:

```

http://localhost:8080/contexto/formulario?nombre=Abraham&apellidos=Otero+Quintana&edad=De+30+a+55&hobbies=lectura&hobbies=tele&hobbies=emborracharse

```

Si hubiésemos indicado que el formulario se enviase empleando el método POST sólo veríamos la URL `http://localhost:8080/contexto/formulario`, y el resto de la información se enviaría dentro del cuerpo del mensaje.

Cuando la petición llegue al servidor, tanto si se envía mediante el método GET como mediante el método POST, el contenedor de Servlets recogerá todos los parámetros del formulario y los guardará en el objeto `HttpServletRequest`. Podemos acceder a ellos a

través del método **getParameter(String)** de dicho objeto. El parámetro de este método es el valor del atributo "**name**" de cada uno de los campos del formulario. Por ejemplo, el primer campo de texto del formulario tiene como valor para este atributo "nombre", así que este será el parámetro que le pasemos al método `getParameter` para obtener el nombre que el usuario ha introducido en el formulario. Veamos un ejemplo de Servlet que procesa un formulario:

```
...
@WebServlet(name="FormularioServlet", urlPatterns={"/formulario"})
public class FormularioServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {

        String nombre = request.getParameter("nombre");
        String apellidos = request.getParameter("apellidos");
        String edad = request.getParameter("edad");
        String[] hobbies = request.getParameterValues("hobbies");

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet que procesa un formulario
basico</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>" + "Hola " + nombre + " " + apellidos+
"</h1>");
            out.println("Tu Franja de edad es " + edad + " y tus
hobbies son:");

            out.println("<ul>");
            for (String hobby : hobbies) {
                out.println("<li>" + hobby + "</li>");
            }
        }
```

```

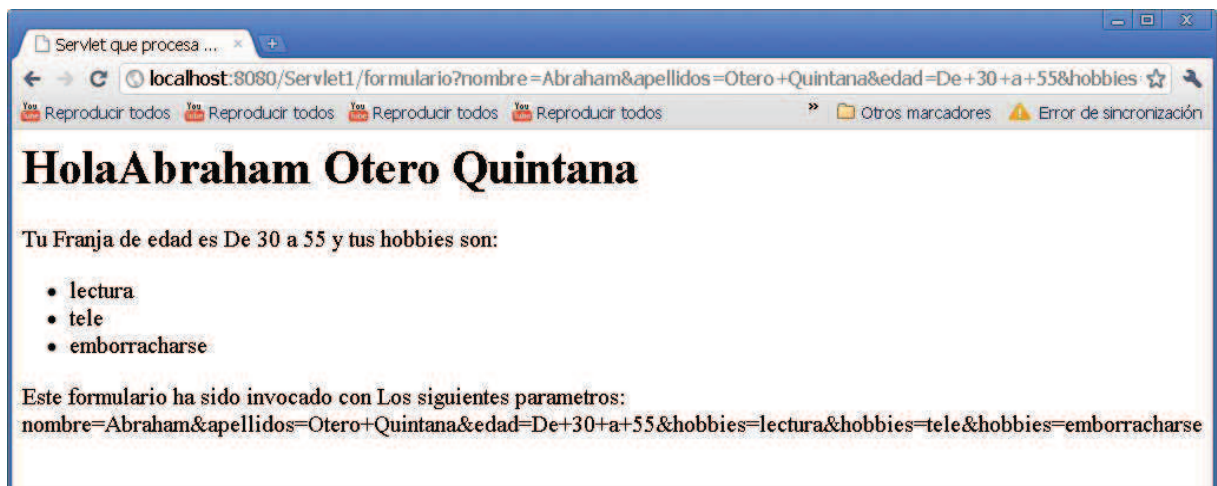
        out.println("</ul>");
        out.println("Este formulario ha sido invocado con Los
siguientes parametros:<br/>");
        out.println(request.getQueryString());

        out.println("</body>");
        out.println("</html>");

    } finally {
        out.close();
    }
}
...

```

En el caso de los hobbies, hemos usado el método `String [] getParameterValues (String)` ya que el campo del formulario correspondiente con los hobbies es de selección múltiple y, por tanto, es posible que haya múltiples valores asociados con él. El método `getQueryString` se emplea para mostrar, con propósitos didácticos, los parámetros enviados al servidor. Éste es un ejemplo de la salida producida por dicho Servlet:



4.5.1 El ejemplo anterior apestá

Aunque entiendo que a estas alturas el lector va a poder hacer bastante poco acerca del problema, quiero que ya seas consciente de él a partir de ya. El ejemplo anterior, apestá. En el mundo real, sería completamente inaceptable. El ejemplo anterior haría que la web que lo albergase fuese tremendamente vulnerable a ataques de cross site scripting (XSS, Cross Site Scripting, http://es.wikipedia.org/wiki/Cross-site_scripting): nadie le impide al usuario introducir en el campo nombre (por ejemplo) del formulario lo siguiente:

```
<iframe src ="http://javaHispano.org" width="100%"  
height="300"></iframe> <iframe src  
="http://www.youtube.com/watch?v=7cj56j9o-eU" width="100%"  
height="300"></iframe>
```

Lo cual empotraría dentro de la página web generada por el Servlet la web de javaHispano y un video de Youtube (el correspondiente con la final de la JavaCup 2009):



En este caso, aunque el comportamiento no es el esperado, el resultado no es grave. Por un lado, porque las páginas web que han sido empotradas son inocuas. Por otro, porque ningún contenido de esta página web queda almacenado en el servidor y no volverá a ser presentado a ningún usuario. Sin embargo, supongamos que la aplicación fuese (por ejemplo) un foro de Internet donde lo que un usuario escribe queda en la web visible para cualquier otro usuario que visite la página. Si en vez de haber empotrado esas dos web inocuas se hubiese empotrado una web que contenga malware, el equipo del usuario que está visitando nuestro foro podría infectarse por el contenido de nuestra página. Este formulario también podría ser empleado por los spammers para llenar nuestra web con enlaces a otras web donde venden Viagra y demás productos, enlaces que no sólo llenarán nuestro foro de mensajes molestos para los usuarios, sino que harán que baje nuestro ranking en Google, o que incluso Google nos elimine de su índice por considerarlos una granja de enlaces de spam.

El lector podría pensar "esto tiene una solución fácil, puedo limitar el número de caracteres que dejo que el usuario meta en cada uno de los campos de texto empleando el atributo "maxlength". Es más, alguien con más conocimiento de HTML podría pensar en emplear, por ejemplo, JavaScript para validar los campos del formulario. Ambas soluciones son completamente inútiles. No es necesario para nada el formulario para realizar el ataque XSS. Por ejemplo, simplemente copiando esta URL en un navegador web:

```
http://localhost:8080/contexto/formulario?nombre=%3Ciframe+src+%3D%22h  
ttp://javaHispano.org%22+width%3D%22100%25%22+height%3D%22300%22%3E%3C  
/iframe%3E+%3Ciframe+src+%3D%22http://www.youtube.com/watch%3Fv%3D7cj5  
6j9o-  
eU%22+width%3D%22100%25%22+height%3D%22300%22%3E%3C/iframe%3E&apellido  
s=&edad=Menor+de+18
```

obtendremos el mismo efecto, independientemente de todas las validaciones que hayamos incluido en la página web `formulariohobbies.html`, que ahora no estaremos empleando para nada en el ataque.

La validación de los parámetros recogidos de un formulario debe realizarse **siempre** en el servidor (emplear, por ejemplo, JavaScript en el cliente para validar los campos de un formulario antes de enviarlos puede incrementar la usabilidad de la aplicación; el usuario no tiene que esperar por la respuesta del servidor para darse cuenta de que ha introducido algo mal; pero no es ningún mecanismo que garantice que la información que llega al servidor es la que "debería" llegar). Ahora el lector podría pensar "vale, es fácil, basta con buscar, por ejemplo, los caracteres "<" y ">"; si alguien me esta atacando y está inyectando etiquetas HTML aparecerán esos caracteres en el mensaje". Incorrecto. Mira la URL esta un par de párrafos arriba. ¿Ves alguno de esos símbolos?. Sanitizar los datos recogidos en un formulario es una tarea tremendamente compleja, sobre todo porque el atacante puede jugar con el encoding de los caracteres que nos envía y emplear otros trucos. Para la mayor parte de las aplicaciones, puede que una solución casera sea más o menos aceptable. Pero si la aplicación está manipulando datos críticos, gestiona dinero, o simplemente está en una web con mucho tráfico, recibirás

ataques seguro y necesitarás emplear un nivel más alto de seguridad. La mejor solución en estos casos (realmente, en cualquier caso) es emplear una librería de una tercera parte que ha sido construida por alguien que realmente sabe cómo hacer el trabajo de sanitizar los parámetros del usuario. Un ejemplo de esta librería es OWASP (http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project).

Si estás leyendo este tutorial, supongo que tu formación sobre programación web con Java es nula o muy básica. Por ello en este momento no debes preocuparte de temas relacionados con la seguridad; esos conceptos más avanzados deberán ser abordados más tarde en tu formación. Pero sí quiero que seas consciente del problema. No creas que después de leer este tutorial vas estar en condiciones para desarrollar una aplicación para que tu banco local permita a sus usuarios acceder a banca online.

5 Ciclo de vida de los Servlets

Los Servlets son gestionados por el contenedor. El contenedor es quien decide cuándo se debe de crear un determinado Servlet, y cuándo se debe destruir. El programador no controla directamente el ciclo de vida del Servlet, pero el API de Java si que proporciona "hooks" (puntos de enganche) para responder a los distintos eventos del ciclo de vida de un Servlet. Estos hooks son métodos de la clase `javax.servlet.Servlet`, el padre de `HttpServlet`. Si el lector está familiarizado con el funcionamiento de los Applets Java, la idea es muy parecida. El navegador web es quien decide cuándo crea y cuándo destruye al Applet. Este lo único que hace es reaccionar ante estos eventos.

Cuando el contenedor crea un Servlet, antes de que éste responda a ninguna petición el contenedor va a ejecutar su método `init(ServletConfig config)`. La especificación de los Servlet garantiza que el Servlet no responderá a ninguna petición hasta que haya terminado la ejecución de este método. A menudo este método realiza distintas labores de inicialización, inicialización que podría depender de parámetros de configuración que se le han pasado al Servlet.

En cualquier momento, el contenedor va a alojar a una única instancia del Servlet por cada definición de dicho Servlet que encontremos en el descriptor de despliegue de la aplicación. Es decir, salvo que en el descriptor de despliegue de la aplicación indiquemos que queremos crear múltiples instancias para un Servlet con la etiqueta `<servlet>`, proporcionando varios nombres (`<servlet-name>`) para una misma clase de Servlet (`<servlet-class>`), el contenedor sólo va a tener una única instancia del Servlet para responder a todas las peticiones que le lleguen. Esto no quiere decir que vaya a haber un único thread que ejecute dichas peticiones; todo lo contrario: múltiples thread pueden estar ejecutando el cuerpo del Servlet para responder a múltiples peticiones de múltiples usuarios. Esto acarrea potenciales problemas de concurrencia que serán abordados en otro capítulo de este tutorial.

Cuando el contenedor destruya nuestro Servlet, antes de destruirlo invocará al método `destroy()`, dándonos la oportunidad de realizar cualquier labor de limpieza de recursos que hayamos podido adquirir.

Otro método interesante definido en la clase Servlet (aunque no relacionado con el ciclo de vida de estos) es el método `String getServletInfo()`. Este método puede ser sobrescrito por el programador para hacer que devuelva información como el autor, la versión, copyright, etcétera. Lo que devuelve es una cadena de texto descriptiva del Servlet.

Una de las tareas que a menudo se lleva a cabo en la inicialización de un Servlet es leer parámetros de configuración. Veamos cómo podemos especificar dichos parámetros.

5.1 Configurando los Servlet: ServletConfig

Esta clase es empleada por el contenedor para pasarle información al Servlet durante su inicialización. Estos son sus métodos:

- **`String getInitParameter(String name)`**: devuelve el parámetro de inicialización asociado con el nombre que se le pasa como argumento.
- **`java.util.Enumeration<String> getInitParameterNames()`**: devuelve los nombres de los parámetros de inicialización. Si no hay ningún parámetro, la enumeración que devuelve estará vacía.
- **`ServletContext getServletContext()`**: devuelve una referencia al contexto del Servlet.
- **`String getServletName()`**: devuelve el nombre de esta instancia de Servlet.

¿Y cómo hacemos para especificar los parámetros de inicialización de un Servlet? Tenemos dos formas diferentes. Podemos emplear una anotación en el código fuente del Servlet si estamos empleando Java EE 6 o superior. Y en cualquier caso siempre podemos emplear el descriptor de despliegue. Las anotaciones toman esta forma:

```
@WebServlet(name="ConfigurableServlet",  
urlPatterns={"/ConfigurableServlet"},  
initParams={@WebInitParam(name="parametro1", value="Valor1")},
```

```
@WebInitParam(name="parametro2", value="Valor2"))
```

Mediante `initParams` indicamos dentro de la anotación `@WebServlet` los parámetros de configuración. Cada parámetro se indica con una anotación `@WebInitParam`. Estas anotaciones tienen dos atributos: el nombre (`name`) del parámetro y su valor (`value`).

Otra forma de indicar los parámetros de configuración es en el descriptor de despliegue. Por ejemplo, este fragmento del descriptor de despliegue sería análogo a la anterior anotación (exceptuando la definición del patrón de URL al cual responde el Servlet, que debería realizarse en una etiqueta `<servlet-mapping>` dentro del descriptor de despliegue):

```
<servlet>
  <servlet-name>ConfigurableServlet</servlet-name>
  <servlet-class>tema5.ConfigurableServlet</servlet-class>
  <init-param>
    <param-name>parametro1</param-name>
    <param-value>Valor1</param-value>
  </init-param>
  <init-param>
    <param-name>parametro2</param-name>
    <param-value>Valor2</param-value>
  </init-param>
</servlet>
```

Si para un determinado Servlet definimos parámetros de configuración a través de anotaciones y en el descriptor de despliegue, todos estos parámetros se le pasarán al Servlet. Si definimos un determinado parámetro mediante una anotación, y después en el descriptor de despliegue volvemos a definir el mismo parámetro con un valor diferente, el descriptor de despliegue (como siempre) toma precedencia sobre las anotaciones y sobrescribirá sus valores, siendo los valores que recibe el Servlet finalmente los especificados en el descriptor de despliegue.

5.2 Un ejemplo

A continuación mostramos el código fuente de un Servlet que lee todos los parámetros de configuración que se le han pasado y los muestra en una página web. En el método `init()` del Servlet se leen sus parámetros de configuración y se almacenan en variables instancia. El método que responde a las peticiones http itera sobre estos parámetros y genera una lista mostrándoselos al usuario. Empleando este Servlet es fácil comprobar cómo al especificar parámetros en el descriptor de despliegue y en anotaciones, el Servlet los recibe todos, así como el hecho de que el descriptor de despliegue toma precedencia sobre las anotaciones.

```
@WebServlet(name="ConfigurableServlet",
urlPatterns={"/ConfigurableServlet"},
initParams={@WebInitParam(name="parametro1", value="Valor1"),
             @WebInitParam(name="parametro2", value="Valor2")})
public class ConfigurableServlet extends HttpServlet {
    private Map<String,String> mapaDeParametrosDeConfiguracion =
        new ConcurrentHashMap<String,String>();

    @Override
    public void init(ServletConfig config){
        Enumeration<String> nombresParametros =
config.getInitParameterNames();
        while (nombresParametros.hasMoreElements()) {
            String nombreParametro =
nombresParametros.nextElement();
            mapaDeParametrosDeConfiguracion.put(nombreParametro,
config.getInitParameter(nombreParametro));
        }
    }

    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 que toma parametros de
configuracion</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Parametros de configuracion:</h1>");
        out.println("<ul>");
        Set<String> s= mapaDeParametrosDeConfiguracion.keySet();

        for (String h : s) {
            out.println("<li>" + h + ": "+
                mapaDeParametrosDeConfiguracion.get(h) + "</li>");
        }
        out.println("</ul>");
        out.println("</body>");
        out.println("</html>");

    } finally {
        out.close();
    }
}

```

6 Recordando al usuario: sesiones y cookies

Como ya hemos visto, el protocolo http es un protocolo sin estado. En el protocolo http, el cliente realiza una petición de un determinado recurso al servidor, y el servidor responde sirviendo ese recurso. Si a continuación el cliente vuelve a pedir otro recurso, el servidor no es consciente de si ha o no servido algún otro recurso en el pasado al usuario, o de qué recursos ha servido. Esto es una limitación muy importante. Básicamente, hace que cada vez que un usuario visite una página web de nuestro portal sea como si ésta fuese la primera vez que visita dicha página web. No podemos diseñar flujos de navegación que, por ejemplo, hagan pasar al usuario por una serie de formularios para completar una tarea (al menos no sin persistir la información a una base de datos, cosa que raramente es recomendable en este escenario). Si nuestra aplicación requiere algún tipo de autenticación, no podremos recordar al usuario y éste tendrá que autenticarse cada vez que visite nuestro portal (realmente, tendría que hacerlo cada vez que su navegador solicite un recurso). Obviamente, esto no es viable.

El protocolo http contempla una forma de almacenar pequeñas piezas de información en el cliente, piezas de información que el navegador web del cliente nos va a devolver cada vez que realice una petición. Son las cookies. Por otro lado, los servidores de aplicaciones Java EE permiten asociar una sesión con cada usuario, en la cual podemos almacenar información relativa a dicho usuario.

6.1 HttpSession

Uno de los servicios que el contenedor de Servlet proporciona a las aplicaciones es la gestión de la sesión. Podemos asociar a cada visitante de nuestra página web una sesión. Esta sesión es un objeto Java que estará ligado de modo unívoco al visitante. Siempre que recibamos una petición de dicho visitante podremos acceder a su sesión, y sólo las peticiones de ese visitante podrán acceder a ella.

La sesión funciona básicamente como un mapa de pares (clave, valor), donde las claves son cadenas de caracteres y los valores pueden ser cualquier objeto Java. Empleando la sesión podemos, por ejemplo, almacenar un determinado objeto Java en la sesión del usuario y responder a su petición. Más adelante, recibimos una segunda petición del usuario, y podemos volver a acceder a la misma sesión y recuperar el objeto Java que guardamos en ella. Se trata de un mecanismo simple y efectivo para gestionar el estado del usuario. Ahora podemos saber si hemos visto o no en el pasado a este usuario, y recordar cosas sobre él.

La sesión en sí es un objeto Java, que por tanto ocupa memoria. Lo mismo sucede con todos los objetos que se almacenen dentro de ella. El servidor no puede crear sesiones de modo indefinido, y almacenarlas para siempre porque agotará su memoria. En algún momento debe destruir esas sesiones. Lo habitual es especificar en el descriptor de despliegue de la aplicación cuánto tiempo en minutos deseamos mantener vivas las sesiones de usuario. La idea detrás de este parámetro de configuración es que es posible que hayamos creado una sesión para un usuario y después el usuario abandona la web y deja de interactuar con la aplicación. Por ejemplo, está en la mitad del proceso de comprar algo, tiene varios artículos en su carrito de la compra y decide que no los quiere y simplemente cierra el navegador web. No existe ninguna forma para que el servidor sepa que el usuario ha cerrado el navegador web. El servidor continuaría esperando la confirmación del usuario para decir que quiere comprar los artículos, y todos esos artículos posiblemente estuviesen representados por objetos Java almacenados en la sesión del usuario. La idea es que después de un tiempo razonable, asumamos que el usuario no va a volver a la web (o si vuelve no va a continuar con la operación que tenía en curso) y vamos a destruir su sesión.

Empleando estas etiquetas en el servidor de despliegue:

```
<session-config>
  <session-timeout>
    30
  </session-timeout>
</session-config>
```

estaremos indicando que después de 30 minutos de inactividad en la sesión del usuario la invalidaremos y, por tanto, destruiremos todos los objetos almacenados en ella y perderemos toda esa información. También es posible indicar programáticamente que queremos destruir una sesión (esto es lo que muy a menudo hace el botón de "desconectarse" o "log out" de las aplicaciones web, botón que casi siempre los usuarios ignoran y simplemente cierran la lengüeta del navegador o el navegador).

Teóricamente, todos los objetos que se almacenen en la sesión de un servidor de aplicaciones Java EE deben implementar la interfaz `Serializable`. Esto es algo que a menudo se ignora, porque nunca da problemas en la máquina de desarrollo, ni los dará en la de despliegue mientras sólo usemos una instancia del servidor. El problema aparece cuando comenzamos a usar un cluster con varios servidores. En este escenario, es posible que en un determinado momento uno de los servidores decida que está muy ocupado y que va a ser otro el que responda una petición del usuario. Para responder a esa petición del usuario, debe transferir la sesión a ese servidor. Y para ello, lo hace serializándola a través de la red. Si la sesión no se puede serializar, lanzará una excepción. Esta es una de esas cosas que a menudo se ignora, y después un día explota en producción.

Estos son los métodos más comúnmente utilizados de `HttpSession`:

- **Object `getAttribute(String name)`:** devuelve un objeto java almacenado en la sesión del usuario, o null si no hay ningún objeto asociado con el nombre que se le pasa como parámetro.
- **java.util.Enumeration `getAttributeNames()`:** devuelve una enumeración de cadenas de caracteres con todos los nombres de todos los objetos almacenados en la sesión del usuario.
- **long `getCreationTime()`:** devuelve el instante en el que fue creada la sesión medido en milisegundos desde el uno de enero de 1970.
- **String `getId()`:** devuelve una cadena de caracteres conteniendo un identificador único para la sesión.
- **long `getLastAccessedTime()`:** devuelve el instante de tiempo en el cual el cliente realizó una petición asociada con esta sesión; el tiempo se mide como el tiempo en milisegundos transcurrido desde el uno de enero de 1970.

- **int getMaxInactiveInterval():** devuelve el máximo tiempo de inactividad, medido en segundos, que se va a permitir para esta sesión. Si la sesión está inactiva más de ese tiempo, será destruida por el contenedor.
- **ServletContext getServletContext():** devuelve el ServletContext al cual pertenece esta sesión.
- **void invalidate():** invalida la sesión y elimina los objetos que estuviesen ligados a ella.
- **void removeAttribute(String name):** elimina de la sesión el atributo asociado al nombre que se le pasa como parámetro.
- **void setAttribute(String name, Object value):** añade a la sesión el objeto que se le pasa como segundo parámetro, siendo la clave asociada con este objeto la cadena de caracteres que se pasa como primer parámetro.
- **void setMaxInactiveInterval(int interval):** especifica el tiempo en segundos que el contenedor esperará antes de invalidar esta sesión.

6.1.1 Ejemplo: SessionServlet

Vamos a ver un ejemplo de un Servlet que almacena información en la sesión. En este ejemplo académico, el usuario es presentado con un formulario (realmente con dos formularios HTML, aunque él no lo sabe) donde hay dos campos de texto y dos botones. El primero de esos botones le permite al usuario añadir un atributo y un valor asociado a su sesión. El atributo y el valor asociado son los especificados por los campos de texto del formulario. El segundo botón del formulario (realmente ese botón pertenece a un segundo formulario) le permite invalidar su sesión. Además, tanto el formulario que permite añadir atributos y valores a la sesión como el formulario que permite invalidar la sesión tienen un campo oculto cuyo atributo name toma el valor "accion" en ambos casos; en el caso del formulario que permite añadir contenido a la sesión el atributo value del campo oculto toma el valor "añadir", mientras que en el caso del formulario que permite invalidar la sesión este atributo toma el valor "invalidar".

Este es un uso típico para los campos hidden de los formularios. Nos permiten en el servidor saber si la información que estamos recibiendo ha llegado de un formulario o

de otro. Para ello, en el servidor recuperaremos el campo cuyo nombre es "accion", y después miraremos su valor. Según el valor de este campo, sabremos si el usuario quiere añadir un nuevo atributo su sesión, o invalidar la sesión. En este ejemplo, podríamos argumentar que habría otras formas de conseguir esto. Por ejemplo, mirando si la petición del usuario contiene o no la información correspondiente a los dos campos de texto, ya que sólo uno de los formularios tiene estos dos campos de texto. Sin embargo, hay situaciones en el mundo real en las cuales ambos formularios tienen exactamente los mismos campos (por ejemplo, el formulario de dar de alta y el de modificar) y el añadir un campo hidden a uno de ellos es una forma fácil para distinguir entre ambos formularios en el servidor.

Este es el código HTML de los formularios:

```
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
  http-equiv="content-type">
  <title>Formulario para añadir cosas a la sesión</title>
</head>
<body>
<form method="get" action="/contexto/SessionServlet"
  name="datos">
Atributo: <input name="parametro"><br>
Valor: <input name="valor"><br>
  <input name="accion" value="anadir" type="hidden"><br>
  <button>Añadir parámetro a la sesión</button>
</form>
<form method="get" action="/contexto/SessionServlet">
  <input name="accion" value="invalidar" type="hidden">
  <button>Invalidar sesion</button>
</form>
</body>
</html>
```

y a continuación se muestra el código del Servlet que lo procesa. Podemos observar cómo en el código lo primero que se hace es recuperar el campo "accion" para saber qué

acción debemos llevar a cabo. Después, añadimos el nuevo atributo a la sesión y generamos el código HTML necesario para mostrar el contenido de la sesión, o invalidamos la sesión. En ambos casos se muestra un enlace que permite volver al formulario, para seguir añadiendo más atributos a la sesión.

```
...
@WebServlet(name = "SessionServlet", urlPatterns =
{"/SessionServlet"})
public class SessionServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        String nuevoAtributo = request.getParameter("parametro");
        String valor = request.getParameter("valor");
        String action = request.getParameter("accion");

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet que muestra el contenido de la
sesion</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>El contenido de tu sesion es:</h1>");

            HttpSession s = request.getSession();
            if (action.equals("invalidar")) {
                s.invalidate();
                out.println("<h1>Sesion invalidada:</h1>");
            } else {
                s.setAttribute(nuevoAtributo, valor);
                out.println("<ul>");

                Enumeration<String> nombresDeAtributos =
s.getAttributeNames();
```

```

        while (nombresDeAtributos.hasMoreElements()) {
            String atributo =
nombresDeAtributos.nextElement();
            out.println("<li><b>" + atributo + ": </b>"
                + s.getAttribute(atributo) + "</li>");
        }

        out.println("</ul>");
    }
    out.println("<a
href=/contexto/tema6/FormularioSesion.html>" + "Volver a la página
anterior</a><br/>");
    out.println(request.getQueryString());

    out.println("</body>");
    out.println("</html>");
} finally {
    out.close();
}
}
...

```

6.2 Las Cookies

Otro mecanismo para "recordar" información de usuarios son las cookies. Este mecanismo forma parte de la especificación http. Consiste en la posibilidad de mandar al cliente pares (valor, clave), donde tanto el valor como la clave son texto. El navegador web del cliente (salvo que el usuario haya deshabilitado esta opción, cosa que permiten todos los navegadores modernos) almacenará las cookies que el servidor le mande en el disco duro del usuario. Cuando el usuario vuelva a esa web, el servidor mirará si tiene alguna cookie y, en caso afirmativo, se la enviará.

Cuando una cookie ha sido establecida por una página web que se encuentra en un determinado directorio del servidor web, el navegador web del usuario devolverá la cookie a cualquier página web que esté alojada en el mismo directorio, o en un subdirectorio de dicho directorio. En cualquier otro caso, el navegador no devolverá la cookie.

Para recuperar las cookies que el navegador del usuario nos está enviando emplearemos el método `Cookie[] getCookies()` del objeto `HttpServletRequest`. Para enviarle una cookie al usuario la añadiremos a la respuesta empleando el método `void addCookie(Cookie)` del objeto `HttpServletResponse`.

Éstos son los métodos más comunes, y el constructor, de la clase `Cookie`:

- **`Cookie(String name, String value)`**: construye una cookie con el nombre especificado y el valor. El nombre de una cookie no puede ser cambiado una vez la cookie ha sido creada. El nombre de la cookie no puede ser null, ni vacío, ni contener caracteres ilegales como espacios en blanco, comas, o puntos y comas. El formato de las cookies está descrito por el RFC 2109.
- **`int getMaxAge()`**: devuelve la edad máxima de esta cookie, medida en segundos.
- **`String getName()`**: devuelve el nombre de la cookie.
- **`String getPath()`**: devuelve la ruta del servidor a la cual el navegador web devolverá la cookie.
- **`String getValue()`**: devuelve el valor actual de la cookie. Es posible modificar el valor de la cookie una vez ha sido establecida.
- **`void setMaxAge(int expiry)`**: establece la máxima vida de esta cookie, medida en segundos. Un valor negativo (-1 habitualmente) quiere decir que la cookie será eliminada cuando el navegador web se cierre. Si se establece como valor 0, la cookie será eliminada inmediatamente.
- **`void setPath(String uri)`**: especifica la ruta a la cual la cookie deberá ser devuelta por el navegador web.
- **`void setValue(String newValue)`**: establece un nuevo valor para la cookie.

6.2.1 Un ejemplo:

CookieServlet es un ejemplo de Servlet que emplea cookies. El Servlet espera peticiones enviadas desde un formulario que permite al usuario crear cookies. En el formulario, el usuario indica el nombre de la cookie, su valor y opcionalmente su duración en segundos. Aquí tienes el código HTML de FormularioCookies.html:

```
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
  http-equiv="content-type">
  <title>Formulario para añadir cookies</title>
</head>
<body>
<form method="get" action="/contexto/CookieServlet" name="datos">
  Cookie: <input name="cookie"><br>
  Valor: <input name="valor"><br>
  Duración: <input name="duracion"><br>
  <button>Añadir cookie</button>
</form>
</body>
</html>
```

Cuando el Servlet recibe la petición, genera una página web donde se listan todas las cookies que ha recibido del navegador web del usuario, junto con sus valores y el atributo maxAge. Este es un ejemplo del listado que genera el Servlet:



Cuando el lector juegue con este formulario, podrá comprobar que posiblemente haya una cookie que él no ha creado : JSESSIONID. Éste es el nombre de una cookie que los servidores de aplicaciones Java EE emplean para asociar de modo unívoco cada usuario con su sesión. El valor de esa cookie permite al servidor de aplicaciones identificar cuál es la sesión del usuario. Si antes de emplear este Servlet el usuario emplea SessionServlet, y por tanto tiene una sesión creada en el servidor, verá aparecer esta cookie.

Empleando este ejemplo el usuario podrá crear cookies que tengan tiempo de expiración cortos (por ejemplo, 20 segundos) y podrá comprobar cómo a los 20 segundos la cookie ha desaparecido. Si en el formulario no se introduce ninguna información en los campos de texto, el Servlet no creará ninguna cookie nueva, sino que simplemente mostrará las que ya existen. Este es el código del Servlet:

```
@WebServlet(name = "CookieServlet", urlPatterns = {"/CookieServlet"})
public class CookieServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        String nombreNuevaCookie = request.getParameter("cookie");
        String valor = request.getParameter("valor");
        int duracion;
        try {
            duracion =
Integer.parseInt(request.getParameter("duracion"));
        } catch (NumberFormatException e) {
            duracion = -1;
        }

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet que muestra cookies</title>");
            out.println("</head>");
            out.println("<body>");
```

```

        out.println("<h1>El contenido de tu sesion es:</h1>");
        out.println("<ul>");

        if (nombreNuevaCookie != null && valor != null
            && !nombreNuevaCookie.equals("")) {
            Cookie nuevaCookie = new Cookie(nombreNuevaCookie,
valor);

            nuevaCookie.setMaxAge(duracion);
            response.addCookie(nuevaCookie);
            out.println("<li><b>" + nuevaCookie.getName() + ":
</b>" + nuevaCookie.getValue() + "; fecha de expiracion: "
                + nuevaCookie.getMaxAge() + "</li>");
        }

        Cookie[] todasLasCookies = request.getCookies();
        if (todasLasCookies != null) {
            for (Cookie cookie : todasLasCookies) {
                out.println("<li><b>" + cookie.getName() + ":
</b>" + cookie.getValue() + "-, fecha de expiracion: "
                    + cookie.getMaxAge() + "</li>");
            }
        }
        out.println("</ul>");
        out.println("<a
href=/contexto/tema6/FormularioCookies.html>" + "Volver a la pagina
anterior</a><br/>");
        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}

```

6.3 Sesiones vs Cookies

Cabría preguntarse ¿y qué uso para mantener el estado del cliente? ¿cookies o la sesión?. Cada solución tiene sus ventajas y sus inconvenientes. Uno de los principales

inconvenientes de las cookies es que sólo permiten almacenar una cadena de caracteres (e incluso hay ciertos caracteres prohibidos que no pueden formar parte de dicha cadena). Además, su tamaño suele estar limitado a 4 kB por los navegadores web, así como su número. No son adecuadas, por tanto, para guardar datos binarios o datos complejos.

Otra de sus grandes desventajas es que el usuario puede borrarlas empleando los menús de su navegador web, por lo que perderemos esa información. Es más, si el usuario es experto, podría modificar las cookies. Nunca debemos almacenar información confidencial o información de mucho valor en la cookie. Por ejemplo, si la web de un banco almacenase una cookie que permitiese a sus usuarios hacer login sin introducir ningún tipo de password, si un hacker se hace con esa cookie podrá suplantar al usuario. Del mismo modo, si tenemos una web que use cookies para determinar las acciones que el usuario puede realizar en nuestra web, el usuario podría modificar su cookie para ganar permisos. En general, en las cookies deberá almacenarse información no crítica, información que podemos asumir que en cierto momento podría perderse o podría ser modificada con fines malignos.

La sesión, por la contra, está en el servidor. El usuario no la puede eliminar, el usuario no puede modificarla, ni los hackers pueden robarla. Además, permite almacenar en ella objetos Java todo lo complejos que queramos, y no sólo cadenas de caracteres de tamaño limitado.

La desventaja de la sesión es que está ocupando memoria RAM en el servidor, por lo que inevitablemente tiene que tener una duración temporal. La sesión no nos vale para recordar cosas sobre el usuario en visitas realizadas en diferentes días, por ejemplo. La sesión sólo vale para mantener el estado durante un conjunto de interacciones relativamente continuadas que el usuario está realizando.

Las cookies, por la contra, pueden tener la duración que nosotros queramos: días, meses o años. Además, se almacenan en el cliente y no en el servidor, por lo que no consumen ningún recurso en el servidor.

Como puedes ver, ambas opciones tienen puntos fuertes y puntos débiles. Dependiendo de lo que se quiera hacer, una u otra puede ser más adecuada.

7 Compartiendo información entre Servlets

Existen varios escenarios donde resulta interesante compartir información entre distintos Servlets. Tanto el uso de la sesión como de las cookies es una posible solución para esto. Sin embargo, están lejos de ser óptimas en todos los escenarios. Hay ocasiones en las cuales un Servlet recibe la petición, realiza una parte del procesamiento y luego se lo envía a un segundo Servlet. El primer Servlet puede querer comunicarle algo al segundo Servlet, algo que sólo es relevante para esta petición y no para peticiones futuras por lo que no es óptimo colocarlo en la sesión del usuario.

También hay escenarios donde queremos compartir información entre usuarios diferentes. Dos o más usuarios diferentes de la aplicación modifican un mismo dato que es compartido entre uno o varios Servlet. Con este fin, sería imposible usar la sesión de los usuarios ya que esta sesión sólo es accesible a sus propias peticiones.

Antes de mostrar los mecanismos que los contenedores Java EE nos proporcionan para guardar información sólo relevante para una petición, o para compartir información entre varios usuarios de la aplicación, vamos a hablar de la clase `ServletContext`, que emplearemos con varios fines a lo largo de este tema.

7.1 ServletContext

Este objeto permite a un Servlet comunicarse con su contenedor. Existe un único contexto por cada aplicación web y por cada instancia de la máquina virtual Java. Esto convierte al contexto en un mecanismo simple para comunicar información entre todos los Servlet que corran en una misma aplicación web (siempre y cuando esa aplicación web corra en un mismo servidor; si tenemos varios servidores, y por tanto varias máquinas virtuales, el contexto de la aplicación no es un sitio adecuado para guardar información a la que queramos acceder globalmente; en estos escenarios suele recurrirse a una base de datos).

El contexto de la aplicación web puede obtenerse a través del método `ServletContext` `getServletContext()` que está definido tanto en `ServletConfig` como en `HttpServletRequest`. Los principales métodos de `ServletContext` son:

- **Object `getAttribute(String name)`:** devuelve un objeto Java (atributo del contexto) asociado con la cadena de caracteres que se le pasa como argumento.
- **java.util Enumeration<String> `getAttributeNames()`:** devuelve una enumeración con todos los nombres de los atributos almacenados en el contexto.
- **String `getContextPath()`:** devuelve la ruta del contexto de la aplicación web. La ruta del contexto es la porción de la URL que se emplea para seleccionar el contexto de una petición. Siempre empieza con una `"/`.
- **String `getInitParameter(String name)`:** devuelve una cadena de caracteres conteniendo los valores de inicialización del contexto expresados en el descriptor de despliegue de la aplicación. Si no existe el parámetro de inicialización, devuelve `null`.
- **java.util Enumeration<String> `getInitParameterNames()`:** devuelve los nombres de todos los parámetros de inicialización del contexto.
- **RequestDispatcher `getRequestDispatcher(String name)`:** devuelve un objeto `RequestDispatcher` que actúa de wrapper para el recurso localizado en la ruta que se le pasa como argumento. Un objeto tipo `RequestDispatcher` puede emplearse para redirigir la petición del usuario a otro recurso, que puede ser tanto estático como dinámico. Para ello, debemos solicitar el objeto `RequestDispatcher` indicando su ruta de modo relativo al actual contexto de la aplicación, y después invocar el método `forward(ServletRequest request, ServletResponse response)` del objeto `RequestDispatcher`.
- **RequestDispatcher `getNamedDispatcher(String name)`:** devuelve un objeto `RequestDispatcher` que actúa de wrapper para un objeto `Servlet` cuyo nombre (esto es, el nombre del `Servlet` especificado en el descriptor de despliegue o mediante la correspondiente anotación) es la cadena de caracteres que se le pasa como argumento al método.

- **java.net.URL getResource(String path):** devuelve una URL que se corresponde con la ruta del recurso que se le pasa como parámetro. La ruta del recurso debe comenzar como un "/" y se interpreta como una ruta relativa a la raíz del contexto de la aplicación, o relativa al directorio /META-INF/resources de un archivo jar que esté contenido en /WEB-INF/lib .
- **java.io.InputStream getResourceAsStream(String path):** este método es similar al anterior, sólo que devuelve el recurso como un cauce de entrada del cual podremos leer datos.
- **void removeAttribute(String name):** elimina el atributo asociado con la cadena de caracteres que se le pasa como argumento.
- **void setAttribute(String name, Object object):** guarda del objeto que se le pasa como segundo parámetro en el contexto de la aplicación y lo asocia con la cadena de caracteres que se le pasa como primer parámetro.
- **boolean setInitParameter(String name, String value):** Fija el valor de un parámetro de inicialización con el valor que se le pasa como segundo argumento.

Podemos fijar un parámetro de inicialización del contexto empleando la anotación `@WebInitParam`:

```
@WebServlet(urlPatterns="/patrones", initParams=
    {@WebInitParam(name="nombre", value="valor")})
```

O en el descriptor de despliegue con etiqueta `<context-param>`.

```
<web-app>
...
<context-param>
  <param-name>nombre</param-name>
  <param-value> valor </param-value>
</context-param>
```

```
...  
</web-app>
```

como siempre, si un mismo parámetro se define empleando la anotación y en el descriptor de despliegue, el parámetro del descriptor de despliegue toma precedencia.

7.2 Compartir información dentro de una misma petición

En ocasiones, resulta útil tener un Servlet inicial que procesa parte de la petición, y después redirige la petición a otro Servlet. Es posible que el primer Servlet necesite comunicar parte de su trabajo al segundo. Por ejemplo, supongamos que tenemos un Servlet cuya función es autenticar al usuario (ver si tiene o no permisos para realizar una determinada acción). Una vez que el primer Servlet ha determinado esto, otro Servlet generará la respuesta que finalmente va a ver el usuario. Pero la respuesta que va a generar depende de si se autenticó con éxito (en este caso seguramente le mostraremos cierta información privada) o no (en este caso seguramente le mostraremos un mensaje de error).

En este escenario, la información sobre el fallo o éxito en la autenticación podría guardarse en la sesión. Sin embargo, si esta información no va a ser usada en futuras peticiones del usuario, no tendría sentido dejar esa información en la sesión indefinidamente. El segundo Servlet una vez ha comprobado si el usuario se autenticó con éxito o no para ver qué respuesta debe generar, debería retirar esa información de la sesión.

Aunque esto es posible, es innecesariamente tedioso. Tenemos otro mecanismo más sencillo para almacenar información que sólo es relevante a una única petición del usuario: el método `void setAttribute (String clave, Object atributo)` del objeto `HttpServletRequest` nos permite guardar un objeto Java como un atributo de esta petición. Una vez hayamos generado la respuesta para el usuario y la petición se destruya, ese objeto también será destruido.

AutenticacionServlet es un Servlet que recoge de un formulario con un nombre de usuario y password, y comprueba si el usuario es el administrador y el password es el correcto. En caso afirmativo, guarda un atributo con clave "autenticado" en la sesión y con valor true. Y a continuación redirige la petición al Servlet AnhadiArticuloServlet empleando un RequestDispatcher. En caso de que la autenticación no tenga éxito, muestra un mensaje de error al usuario empleando nuevamente un RequestDispatcher, que en este caso redirige la petición a un recurso estático: la página "/tema7/error.html". Si la autenticación se realiza de modo adecuado, el Servlet también guardará como un atributo del objeto petición un elemento (una cadena de caracteres) que fue enviada desde el formulario que accede al Servlet:

```
@WebServlet(name="AutenticacionServlet",
urlPatterns={"/AutenticacionServlet"})
public class AutenticacionServlet extends HttpServlet {

    private String passwordAdministrador;
    ...

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

        String usuario = request.getParameter("usuario");
        String password = request.getParameter("password");
        String elemento = request.getParameter("elemento");
        ServletContext contexto = request.getServletContext();

        if (usuario!= null && usuario.equals("administrador")
            && password.equals(passwordAdministrador)){
            request.setAttribute("autenticado", true);
            request.setAttribute("elemento", elemento);
            RequestDispatcher anhadirServlet =
contexto.getNamedDispatcher("AnhadiArticuloServlet");
            anhadirServlet.forward(request, response);
        } else {
            RequestDispatcher paginaError =
contexto.getRequestDispatcher("/tema7/error.html");
            paginaError.forward(request, response);
        }
    }
}
```

El password del administrador es obtenido por AutenticacionServlet a partir de un fichero de configuración guardado en el servidor. Dicho fichero de configuración se le pasa a la aplicación web a través de un parámetro de configuración del Servlet:

```
<servlet>
    <servlet-name>AutenticacionServlet</servlet-name>
    <servlet-class>tema7.AutenticacionServlet</servlet-class>
    <init-param>
        <param-name>ficheroUsuarios</param-name>
        <param-value>/tema7/ficheroUsuarios.prop</param-value>
    </init-param>
</servlet>
```

En el método init() de AutenticacionServlet se accede al valor de dicho parámetro de configuración (esto es, a la ruta del fichero de configuración), y empleando el método getResourceAsStream() del contexto de la aplicación obtenemos dicho fichero como un cauce de entrada. El fichero es un fichero de propiedades, y se lee empleando la clase java.util.Properties. El contenido de este fichero, que debe estar situado en /web/tema7/ficheroUsuarios.prop, es:

```
administradorPassword = administrador
```

En caso de que se produzca algún error al abrir el fichero donde están almacenados los password, se lanzará una excepción y se escribe al log un mensaje de error con severidad grave.

```
@Override
public void init(ServletConfig config) {
    String ficheroUsuarios =
config.getInitParameter("ficheroUsuarios");
```

```

        ServletContext contexto = config.getServletContext();
        InputStream is =
contexto.getResourceAsStream(ficheroUsuarios);
        Properties ficheroPropiedades = new Properties ();
        try {
            ficheroPropiedades.load(is);
            passwordAdministrador =
ficheroPropiedades.getProperty("administradorPassword");
        } catch (Exception ex) {
            Logger.getLogger(AuthenticacionServlet.class.getName()).log(
Level.SEVERE, "No se pudo cargar el fichero con los password", ex);
        }
    }
}

```

AuthenticacionServlet colabora con otros dos Servlets. Uno es AnhadirArticuloServlet, que añade al contexto de la aplicación el elemento que envió el usuario, en caso de que el usuario se haya autenticado con éxito. Sólo es posible acceder a AnhadirArticuloServlet cuando el usuario está correctamente autenticado. AnhadirArticuloServlet finalmente redirige la petición a ListarArticulosServlet, que realiza un listado de todos los artículos disponibles.

ListarArticulosServlet puede ser accedido por cualquier usuario de la aplicación, conozca o no el password del administrador. En la siguiente sección veremos estos otros dos Servlets, que emplean el contexto de la aplicación para comunicarse entre ellos.

7.3 Guardando información en el contexto de la aplicación

El contexto de la aplicación nos permite compartir información dentro de toda una aplicación web, y no sólo a nivel de petición (como permite el objeto HttpServletRequest) o entre todas las peticiones que realice un mismo usuario de la aplicación web (como permite la sesión de dicho usuario). AnhadirArticuloServlet recupera de la petición el atributo "autenticado"; en caso de que el usuario no esté autenticado le mostrará la página estática de error empleando un RequestDispatcher.

AuthenticacionServlet sólo va a redirigir la petición del usuario a AnhadirArticuloServlet

cuando el usuario se ha autenticado con éxito. Pero nada impide al usuario teclear la URL correspondiente con `AnhadiArticuloServlet` en la barra de su navegador, por lo tanto tenemos que volver a comprobar si realmente se ha autenticado o no, y esto lo comprobamos a través de la atributo de la petición del usuario. El usuario no tiene ninguna forma de modificar estos atributos (sólo puede añadir parámetros simulando el envío de un formulario servidor, pero no tiene forma de añadir atributos a la petición).

Si el usuario está debidamente autenticado, intentamos recuperar del contexto de la aplicación un atributo con nombre "lista". Esta es la lista de todos los artículos disponibles. El formulario de `AnhadiArticuloServlet` permite añadir un nuevo artículo a la lista, cuando el usuario que está añadiendo dicho artículo está autenticado correctamente (es decir, cuando en el formulario ha introducido el nombre de usuario y el password correcto). Si en el contexto no existiese ningún objeto asociado con el atributo "lista" (porque la lista no se ha creado) la crearemos. En ambos casos, añadimos el nuevo artículo a la lista y finalmente redirigimos la petición al Servlet `ListarArticulosServlet` empleando un `RequestDispatcher`:

```
@WebServlet(name = "AnhadiArticuloServlet", urlPatterns =
{"/AnhadiArticuloServlet"})
public class AnhadiArticuloServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

        ServletContext contexto = request.getServletContext();
        Boolean autenticado = (Boolean)
request.getAttribute("autenticado");
        if (!autenticado) {
            RequestDispatcher paginaError =
contexto.getRequestDispatcher("/tema7/error.html");
            paginaError.forward(request, response);
        }
        else {
            List<String> lista = (List<String>)
contexto.getAttribute("lista");
            if (lista == null) {
```



```

        lista = new LinkedList<String>();
        contexto.setAttribute("lista", lista);
    }
    lista.add((String) request.getAttribute("elemento"));
    RequestDispatcher listarArticulosServlet =
contexto.getNamedDispatcher("ListarArticulosServlet");
    listarArticulosServlet.forward(request, response);
}
}

```

El Servlet `ListarArticulosServlet` puede ser accedido por usuarios autenticados o no autenticados. Este Servlet recupera el objeto `lista` del contexto de la aplicación, y realiza un listado de su contenido, sin realizar ninguna modificación a la lista que contiene todos los artículos:

```

@WebServlet(name="ListarArticulosServlet",
urlPatterns={"/ListarArticulosServlet"})
public class ListarArticulosServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        ServletContext contexto = request.getServletContext();
        List<String> lista = (List<String>)
contexto.getAttribute("lista");

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet
ListarArticulosServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Lista:</h1>");

```

```

        out.println("<ul>");
        if (lista != null) {
            for (String articulo : lista) {
                out.println("<li>" + articulo + "</li>");
            }
        }
        out.println("</ul>");

        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}
...

```

El lector puede realizar el siguiente experimento con la aplicación: abre un navegador web (por ejemplo FireFox, y vete a <http://localhost:8080/contexto/tema7/login.html>. Empleando este formulario, añade varios elementos a la lista. Después, arranca otro navegador web diferente (Chrome, o si no queda más remedio, Internet Explorer) y vete a la URL: <http://localhost:8080/contexto/ListarArticulosServlet>. Podrás ver como en el segundo navegador aparece la lista con todos los elementos que has añadido. Al estar empleando dos navegadores web diferentes, para el servidor de aplicaciones a todos los efectos son dos usuarios diferentes que están accediendo a la aplicación. Cada uno de los navegadores web tiene asociada su propia sesión, por lo que no podrían compartir atributos empleando la sesión. Pero si lo pueden hacer empleando el contexto.

7.4 Recapitulando: opciones para "recordar" información

Dentro de la aplicación web tenemos tres ámbitos diferentes para guardar información. El ámbito más reducido es el formado por los atributos que mantenemos en el objeto `HttpServletRequest`. Los objetos almacenados de este modo sólo duran mientras se procesa una petición del usuario, petición que a veces se va redirigiendo entre distintos

Servlets o recursos estáticos del servidor. Una vez que hemos terminado de procesar la petición, el objeto `HttpServletRequest` y todos sus atributos se destruyen.

El siguiente ámbito sería el dado por la sesión del usuario. Los objetos que mantenemos con atributos de la sesión permanecerán en ella hasta que la sesión se invalide, o bien porque el usuario ha permanecido demasiado tiempo inactivo, o bien porque la hemos invalidado programáticamente.

Tanto los objetos que se almacenan en la petición, como los que se almacenan en la sesión del usuario sólo son accesibles para el propio usuario. No es posible que otro usuario acceda de ningún modo alguno de estos objetos.

El tercer ámbito es el dado por contexto de la aplicación web. Los objetos almacenados en dicho contexto son accesibles a todos los componentes de la aplicación web, independientemente del usuario que esté accediendo a dicho componente. Esto convierte al contexto en una forma sencilla de almacenar información global para toda la aplicación, como por ejemplo un catálogo de productos que va a ser compartido por varios Servlet.

Una cuarta opción para "recordar" información son las cookies. En este caso, también sólo se pueden emplear para recordar información sobre un único usuario, y no para compartir información entre varios usuarios.

8 Gestión declarativa de errores en la aplicación

En las aplicaciones Java EE tenemos dos formas de gestionar los posibles errores que sucedan: declarativamente o programáticamente. El modo declarativo de gestión es simple de emplear: basta con añadir las etiquetas adecuadas al descriptor de despliegue de la aplicación e indicar en las etiquetas cómo se va a gestionar la excepción: mostrando un recurso estático, o invocando a un Servlet. La otra opción es gestionar errores programáticamente: dentro del código fuente de nuestro Servlet, incluir el código fuente requerido para la gestión de los errores, o incluir redirecciones al recurso estático o dinámico que va a gestionar la excepción.

La gestión programática de excepciones es más flexible, más potente y más tediosa. Además, tiene la desventaja de mezclar el código fuente de la lógica de la aplicación con el código fuente de la gestión de los errores. La gestión declarativa de excepciones es menos potente y menos flexible. Por ejemplo, no es posible implementar asociaciones muchos a uno entre las excepciones y los recursos que las van a gestionar; requiere exponer las URL de los recursos que los van a gestionar al usuario (por lo que el usuario podría acceder a ellos directamente) y en ocasiones no es suficientemente flexible: al gestionar declarativamente las excepciones, haremos siempre lo mismo para cada tipo de excepción, mientras que al gestionarlas programáticamente podríamos hacer cosas diferentes para un mismo tipo de excepción dependiendo de dónde y cómo se ha generado ésta.

Como ya hemos dicho, la gestión programática consiste simplemente en escribir dentro de los Servlet el código fuente requerido para gestionar el error. No tiene más que explicar. En este tema vamos a abordar la gestión declarativa de errores.

8.1 Gestión de códigos de error http

En el descriptor de despliegue de nuestra aplicación web podemos incluir una o varias etiquetas `<error-page>`. Dentro de cada una de estas etiquetas, indicaremos un recurso estático o dinámico destinado a gestionar el error http correspondiente de la aplicación (recuerdo al lector que en el tema 2 puede encontrar una lista con los errores http). El servidor de aplicaciones va a mostrar unas páginas de error por defecto cuando se produce uno de estos errores http. Sin embargo estas páginas son bastante poco amistosas para el usuario de la aplicación, y además tienen una apariencia muy diferente a la apariencia del portal web, ya que la página está completamente generada por el servidor de aplicaciones y no respeta el look and feel de la web. De ahí el interés en proporcionar unas páginas para gestionar estos errores más adecuadas.

En nuestro descriptor de despliegue se han incluido las siguientes etiquetas para gestionar errores http:

```
...
<error-page>
  <error-code>404</error-code>
  <location>/tema8/404error.html</location>
</error-page>

<error-page>
  <error-code>500</error-code>
  <location>/tema8/500error.html</location>
</error-page>
...
```

cuando se produzca un error 404 (el error http que se produce cuando el usuario solicita un recurso que no existe en el servidor) el servidor de aplicaciones redirigirá la petición a la página de error que hayamos indicado nosotros. Las que se proporcionan como ejemplo en este tutorial son todavía más malas que las del servidor de aplicaciones, pero servirán para que el lector se haga una idea de cómo funciona el mecanismo.

Las páginas de error que se proporcionan con el tutorial incluyen un curioso comentario HTML... debido a que por algún motivo algunos navegadores como Internet Explorer se niegan a mostrar páginas de error 404 que tengan una longitud inferior a 256 bytes.

8.2 Gestión declarativa de excepciones

En el descriptor de despliegue de la aplicación también es posible indicar qué recursos se va a emplear para gestionar una determinada excepción Java. Por ejemplo:

```
...
<error-page>
  <exception-type>NullPointerException</exception-type>
  <location>/GestorDeExcepcionesServlet</location>
</error-page>
<error-page>
  <exception-type>NumberFormatException</exception-type>
  <location>/GestorDeExcepcionesServlet</location>
</error-page>
...
```

Estas etiquetas en el descriptor de despliegue hacen que cuando se produzca una `NullPointerException` o una `NumberFormatException` en nuestra aplicación, independientemente de cuál haya sido el Servlet que la ha producido, se llame al recurso `/GestorDeExcepcionesServlet`.

Cuando el contenedor web gestione una de estas excepciones, redirigirá la petición al recurso que se indique en el descriptor de despliegue, enviando en la petición los objetos `HttpServletRequest` y `HttpServletResponse` originales. Además, en el objeto `HttpServletRequest` va a añadir dos nuevos atributos. El primero de ellos está asociado con la clave `"javax.servlet.error.exception"` y es la excepción que se ha producido. El segundo de ellos está asociado con la clave `"javax.servlet.error.request_uri"` y es la URL del recurso que ha producido la excepción.

GestorDeExcepcionesServlet informa al usuario de que se ha producido un error en la aplicación, y a continuación muestra la traza de la excepción. Después, muestra la URL del recurso que produjo el error:

```
@WebServlet(name = "GestorDeExcepcionesServlet", urlPatterns =
{"/GestorDeExcepcionesServlet"})
public class GestorDeExcepcionesServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException {
        Exception exception =
            (Exception)
request.getAttribute("javax.servlet.error.exception");
        String urlFuenteDelError = (String)
request.getAttribute("javax.servlet.error.request_uri");

        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet
GestorDeExcepcionesServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Se ha producido el siguiente
error:</h1>");
            out.println("<pre>");
            exception.printStackTrace(out);
            out.println("</pre>");
            out.println("<br/>Al intentar acceder al siguiente
recurso:<br/>");
            out.println("<pre>");
            out.println(urlFuenteDelError);
            out.println("</pre>");
            out.println("</body>");
            out.println("</body>");
```

```

        out.println("</html>");
    } finally {
        out.close();
    }
}

```

Para probar este Servlet he construido otro Servlet con nombre `ExceptionServlet`. Este Servlet genera un número aleatorio y, en base al valor de dicho número, la mitad de las veces lanza una `NumberFormatException`, y la otra mitad una `NullPointerException` (es difícil escribir un Servlet más inútil y con un comportamiento peor que éste):

```

@WebServlet (name="ExceptionServlet",
urlPatterns={"/ExceptionServlet"})
public class ExceptionServlet extends HttpServlet {

    protected void processRequest (HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        if (((int) (Math.random()*10)) %2 == 1) {
            Integer.parseInt("Esto no es un número");
        }
        else {
            Object o = null;
            o.toString();
        }
    }
}

```

El lector accediendo a la URL `http://localhost:8080/contexto/ExceptionServlet` y refrescando varias veces la pantalla podrá ver los mensajes correspondientes con ambos tipos de excepciones. Éstos son los dos tipos de errores diferentes que se pueden generar:

Servlet GestorDeExc... x +

localhost:8080/contexto/ExceptionServlet

Reproducir todos Reproducir todos Reproducir todos Reproducir todos

Otros marcadores Error de sincronización

Se ha producido el siguiente error:

```
java.lang.NumberFormatException: For input string: "Esto no es un numero"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:449)
    at java.lang.Integer.parseInt(Integer.java:499)
    at tema8.ExceptionServlet.processRequest(ExceptionServlet.java:18)
    at tema8.ExceptionServlet.doGet(ExceptionServlet.java:29)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:734)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:847)
    at org.apache.catalina.core.StandardWrapper.service(StandardWrapper.java:1523)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:279)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:188)
    at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:641)
    at com.sun.enterprise.web.WebPipeline.invoke(WebPipeline.java:97)
    at com.sun.enterprise.web.PESessionLockingStandardPipeline.invoke(PESessionLockingStandardPipeline.java:85)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:185)
    at org.apache.catalina.connector.CoyoteAdapter.doService(CoyoteAdapter.java:332)
    at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:233)
    at com.sun.enterprise.v3.services.impl.ContainerMapper.service(ContainerMapper.java:165)
    at com.sun.grizzly.http.ProcessorTask.invokeAdapter(ProcessorTask.java:791)
    at com.sun.grizzly.http.ProcessorTask.doProcess(ProcessorTask.java:693)
    at com.sun.grizzly.http.ProcessorTask.process(ProcessorTask.java:954)
    at com.sun.grizzly.http.DefaultProtocolFilter.execute(DefaultProtocolFilter.java:170)
    at com.sun.grizzly.DefaultProtocolChain.executeProtocolFilter(DefaultProtocolChain.java:135)
    at com.sun.grizzly.DefaultProtocolChain.execute(DefaultProtocolChain.java:102)
    at com.sun.grizzly.DefaultProtocolChain.execute(DefaultProtocolChain.java:88)
    at com.sun.grizzly.http.HttpProtocolChain.execute(HttpProtocolChain.java:76)
    at com.sun.grizzly.ProtocolChainContextTask.doCall(ProtocolChainContextTask.java:53)
    at com.sun.grizzly.SelectionKeyContextTask.call(SelectionKeyContextTask.java:57)
    at com.sun.grizzly.ContextTask.run(ContextTask.java:69)
    at com.sun.grizzly.util.AbstractThreadPool$Worker.doWork(AbstractThreadPool.java:330)
    at com.sun.grizzly.util.AbstractThreadPool$Worker.run(AbstractThreadPool.java:309)
    at java.lang.Thread.run(Thread.java:619)
```

Al intentar acceder al siguiente recurso:

/contexto/ExceptionServlet

Servlet GestorDeExc... x +

localhost:8080/contexto/ExceptionServlet

Reproducir todos Reproducir todos Reproducir todos Reproducir todos

Otros marcadores Error de sincronización

Se ha producido el siguiente error:

```
java.lang.NullPointerException
    at tema8.ExceptionServlet.processRequest(ExceptionServlet.java:22)
    at tema8.ExceptionServlet.doGet(ExceptionServlet.java:29)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:734)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:847)
    at org.apache.catalina.core.StandardWrapper.service(StandardWrapper.java:1523)
    at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:279)
    at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:188)
    at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:641)
    at com.sun.enterprise.web.WebPipeline.invoke(WebPipeline.java:97)
    at com.sun.enterprise.web.PESessionLockingStandardPipeline.invoke(PESessionLockingStandardPipeline.java:85)
    at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:185)
    at org.apache.catalina.connector.CoyoteAdapter.doService(CoyoteAdapter.java:332)
    at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:233)
    at com.sun.enterprise.v3.services.impl.ContainerMapper.service(ContainerMapper.java:165)
    at com.sun.grizzly.http.ProcessorTask.invokeAdapter(ProcessorTask.java:791)
    at com.sun.grizzly.http.ProcessorTask.doProcess(ProcessorTask.java:693)
    at com.sun.grizzly.http.ProcessorTask.process(ProcessorTask.java:954)
    at com.sun.grizzly.http.DefaultProtocolFilter.execute(DefaultProtocolFilter.java:170)
    at com.sun.grizzly.DefaultProtocolChain.executeProtocolFilter(DefaultProtocolChain.java:135)
    at com.sun.grizzly.DefaultProtocolChain.execute(DefaultProtocolChain.java:102)
    at com.sun.grizzly.DefaultProtocolChain.execute(DefaultProtocolChain.java:88)
    at com.sun.grizzly.http.HttpProtocolChain.execute(HttpProtocolChain.java:76)
    at com.sun.grizzly.ProtocolChainContextTask.doCall(ProtocolChainContextTask.java:53)
    at com.sun.grizzly.SelectionKeyContextTask.call(SelectionKeyContextTask.java:57)
    at com.sun.grizzly.ContextTask.run(ContextTask.java:69)
    at com.sun.grizzly.util.AbstractThreadPool$Worker.doWork(AbstractThreadPool.java:330)
    at com.sun.grizzly.util.AbstractThreadPool$Worker.run(AbstractThreadPool.java:309)
    at java.lang.Thread.run(Thread.java:619)
```

Al intentar acceder al siguiente recurso:

/contexto/ExceptionServlet

9 Problemas de concurrencia con los Servlets

En una gran cantidad de escenarios, el modelo de programación de los Servlets es un tremendo éxito en cuanto gestión de la concurrencia se refiere. Permite que el programador escriba un programa sin tener en cuenta consideraciones de concurrencia; el programa se escribe como si tuviese un usuario accediendo a cada recurso. Y es el contenedor de Servlets el que se encarga de permitir que múltiples thread estén ejecutando de modo concurrente el mismo código.

En muchos escenarios, este modelo funciona perfectamente y permite a desarrolladores sin amplios conocimientos en concurrencia desarrollar aplicaciones altamente concurrentes. Un éxito que todavía no ha sido replicado fuera de las aplicaciones de servidor en la plataforma Java.

Pero no es la solución más perfecta, y hay algunos problemas de concurrencia de los cuales no nos abstrae completamente este modelo de programación. El propósito de este tema es comprender esos problemas de concurrencia y aprender a resolverlos.

9.1 Un Servlet con problemas de concurrencia

Échale un vistazo a este código:

```
@WebServlet(name="ProblemaDeConcurrenciaServlet",
urlPatterns={"/ProblemaDeConcurrenciaServlet"})
public class ProblemaDeConcurrenciaServlet extends HttpServlet{
    public static final int REPETICIONES = 3000000;
    private int valor;
```

```

        protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
            valor=0;
            int[] numerosAleatorios = new int[REPETICIONES];
            for (int i = 0; i < REPETICIONES; i++) {
                double a = Math.random()*10;
                numerosAleatorios[i] = (int)a;
                valor += numerosAleatorios[i];
            }
            for (int i = 0; i < REPETICIONES; i++) {
                valor -= numerosAleatorios[i];
            }
            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out = response.getWriter();
            try {
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Problema de concurrencia</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("El resultado es " + valor + "");
                out.println("</body>");
                out.println("</html>");
            } finally {
                out.close();
            }
        }
    }

```

ProblemaDeConcurrenciaServlet genera REPETICIONES números enteros aleatorios, que almacena en un array. El Servlet suma todos estos números a una variable instancia con nombre valor. A continuación, resta todos esos números y genera una página web donde se muestra el valor de la variable valor. En principio, parece que valor siempre debería ser 0. Sin embargo, esto no tiene por qué ser así.

Recordemos que el contenedor de Servlets va a crear una única instancia de nuestro ProblemaDeConcurrenciaServlet. Esa instancia puede ser ejecutada por múltiples thread para responder a múltiples peticiones de múltiples usuarios. Supongamos que llega una primera petición a ProblemaDeConcurrenciaServlet. Un thread comienza a ejecutar su

código, asigna a valor 0, y comienza a generar números aleatorios, a sumarlos y a restarlos. En mitad de este proceso, llega una segunda petición para ProblemaDeConcurrenciaServlet. El contenedor de Servlets seleccionará otro thread de su pool de threads y hará que este nuevo thread también comience a ejecutar el código de la misma instancia de ProblemaDeConcurrenciaServlet. Por tanto, el segundo thread hará que la variable instancia valor vuelva a valer cero (cuando el primer thread estaba en medio de sus operaciones, sumando y restando números) y comenzará también él a restar y a sumar números. Obviamente, necesitamos mucha suerte para que después de realizar todas estas operaciones "valor" acabe teniendo un valor de 0.

Si el usuario quiere probar el código de ProblemaDeConcurrenciaServlet puede hacerlo accediendo a la URL `/contexto/tema9/ProblemaDeConcurrenciaServlet.html`. ProblemaDeConcurrenciaServlet.html es una página web que contiene múltiples iframes; cada frame es para lo que a nosotros nos atañe una página web independiente. Cada uno de estos frames va a incluir el contenido de la URL `/contexto/ProblemaDeConcurrenciaServlet`. Este es el código de la página web:

```
<iframe src="/contexto/ProblemaDeConcurrenciaServlet" width="19%" >
</iframe>

<iframe src="/contexto/ProblemaDeConcurrenciaServlet" width="19%">
</iframe>

<iframe src="/contexto/ProblemaDeConcurrenciaServlet" width="19%" >
</iframe>

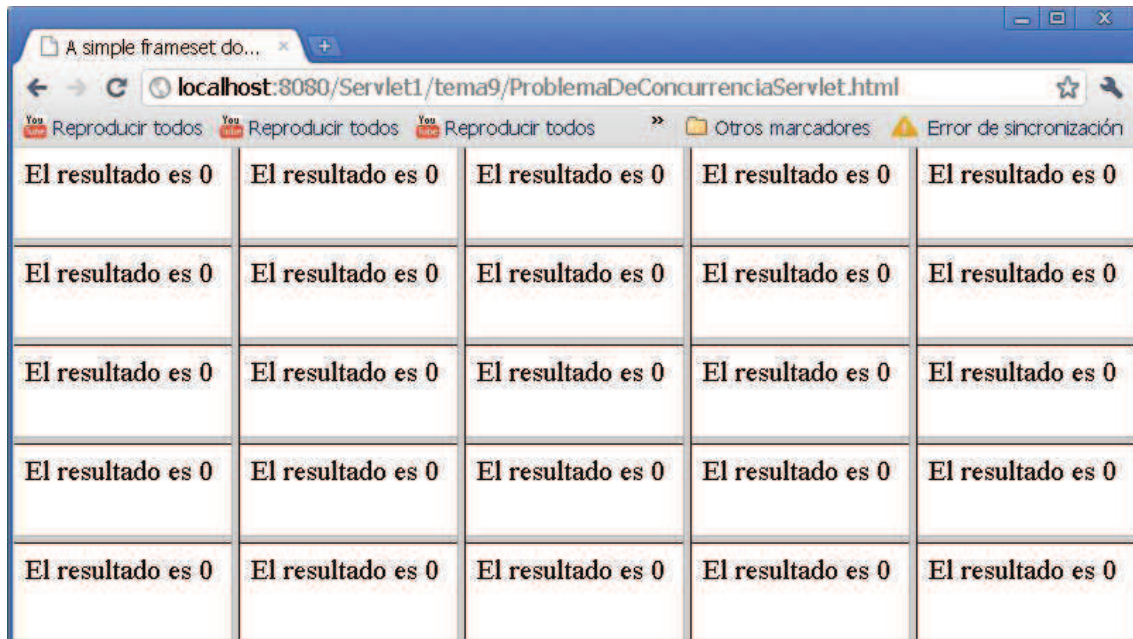
<iframe src="/contexto/ProblemaDeConcurrenciaServlet" width="19%">
</iframe>

<iframe src="/contexto/ProblemaDeConcurrenciaServlet" width="19%" >
</iframe>

...
```

Y así hasta sumar un total de 25 iframes. En cada uno de ellos se ha indicado que debe tener un ancho de 19% (el motivo de esto es que si se especifica un ancho de un 20% en algunos navegadores los cinco iframes no cogen ya que acaban ocupando más de un 100% por los píxeles de separación que dejan entre ellos). Es decir, esta página define

dentro de nuestro navegador un grid de 5×5 , y en cada una de las casillas de ese grid incluye el contenido de ProblemaDeConcurrenciaServlet. Esta técnica (el crear una página web HTML que incluya múltiples veces otra página web) a menudo resulta útil en la depuración de aplicaciones web. Si accedes a esta página web con un navegador verás algo como:

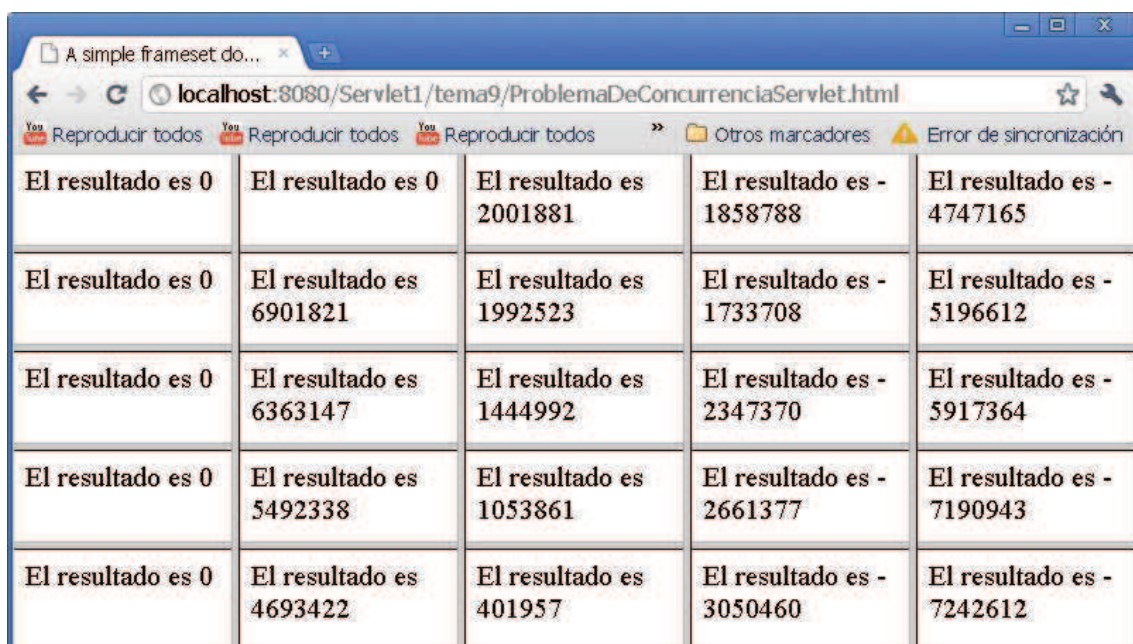


ProblemaDeConcurrenciaServlet ha sido invocado 25 veces y, cómo podría esperar un lector no familiarizado con problemas de concurrencia, el resultado siempre ha sido 0. El motivo por el cual nada ha fallado es que nuestro navegador web ha realizado las 25 peticiones sucesivamente, sin iniciar una petición hasta la que la anterior ha terminado.

Vamos a volver a visitar la URL, pero esta vez vamos a abrir dos navegadores web diferentes (tiene que ser necesariamente dos navegadores web diferentes; no vale con que abras dos lengüetas en un navegador web, o abras dos veces un mismo navegador web). Visitaremos la URL primero en uno de los navegadores, y después en el segundo. El código de ProblemaDeConcurrenciaServlet está preparado para tardar en ejecutarse unos 15 o 20 segundos cuando es invocado 25 veces en la mayor parte de los ordenadores modernos. Si cuando abras la página web ProblemaDeConcurrenciaServlet.html con tu navegador, de un modo prácticamente instantáneo ves los 25 resultados y no te da tiempo a abrir la misma página web en el segundo navegador una de dos: o tienes un equipo que es una máquina de la leche (¡enhorabuena!) o estás leyendo este tutorial muchos años después de que ya lo haya

escrito. En cualquier caso, incrementa el valor de la variable REPETICIONES para que a tu ordenador le lleve al menos unos 15 segundos renderizar la página ProblemaDeConcurrenciaServlet.html. Su valor actual es 3 millones. Cuando incrementes este valor, ten cuidado de no agotar toda la memoria del heap del servidor de aplicaciones (algo que casi con total seguridad sucederá si por ejemplo haces que REPETICIONES valga 300 millones; aunque no debería ser necesario poner un valor tan alto en la variable a no ser que estés leyendo este tutorial muchos, muchos años después de que yo lo haya escrito).

El resultado que deberías ver al abrir esta página en dos navegadores diferentes, de modo simultáneo, es algo como esto (hago énfasis una vez más en que es necesario emplear dos navegadores; la mayor parte de los navegadores web no abren múltiples conexiones de modo simultáneo contra una misma página web, aunque abras dos lengüetas; si abres esta página en dos lengüetas de un navegador, con toda probabilidad verás que no aparece ningún contenido en una de ellas hasta que ha terminado de rellenarse la otra):



El resultado es 0	El resultado es 0	El resultado es 2001881	El resultado es - 1858788	El resultado es - 4747165
El resultado es 0	El resultado es 6901821	El resultado es 1992523	El resultado es - 1733708	El resultado es - 5196612
El resultado es 0	El resultado es 6363147	El resultado es 1444992	El resultado es - 2347370	El resultado es - 5917364
El resultado es 0	El resultado es 5492338	El resultado es 1053861	El resultado es - 2661377	El resultado es - 7190943
El resultado es 0	El resultado es 4693422	El resultado es 401957	El resultado es - 3050460	El resultado es - 7242612

A simple frameset document - Mozilla Firefox

Archivo Editar Ver Historial Marcadores Herramientas Ayuda

http://localhost:8080/Servlet1/tema9/ProblemaDeC

Más visitados Comenzar a usar Firefox Últimas noticias

A simple frameset document

El resultado es -8138758	El resultado es -3374802	El resultado es 543797	El resultado es 4101066	El resultado es 0
El resultado es -7534365	El resultado es -2872091	El resultado es 1235431	El resultado es -785440	El resultado es 0
El resultado es -6811387	El resultado es -2458009	El resultado es 1309287	El resultado es 5580769	El resultado es 0
El resultado es -6027301	El resultado es -1624728	El resultado es 1712026	El resultado es 5923492	El resultado es 0
El resultado es -5526326	El resultado es -1083979	El resultado es 2375807	El resultado es 6484102	El resultado es 0

Terminado

En mi caso, lancé primero el navegador Chrome. Como puedes ver, los seis primeros resultados que obtuvo son correctos porque en este momento había un único thread ejecutando el cuerpo de ProblemaDeConcurrenciaServlet (no me había dado tiempo a refrescar la página web en FireFox). En cuanto el segundo navegador comienza a realizar peticiones, dos threads diferentes comienzan a ejecutar el código de ProblemaDeConcurrenciaServlet. El resultado por tanto no es el correcto. Al final, nuevamente sólo un thread ejecuta ese código, ya que las peticiones de Chrome han terminado y sólo Firefox está realizando peticiones. Por ello los cinco últimos resultados que ha obtenido FireFox vuelven a ser correctos.

Espero que con este ejemplo te quede claro en qué consisten los problemas de concurrencia con los Servlets.

9.2 ¿Con qué variables pueden aparecer los problemas de concurrencia?

Nunca vas a tener un problema de concurrencia con una variable local de un método (sea o no ese método de un Servlet). Las variables locales de los métodos están confinadas en el stack, y cada thread tiene su propio stack. Por tanto, todas las variables definidas dentro de los métodos de los Servlets, y que no han sido exportadas a una variable con un ámbito mayor (por ejemplo, una variable de instancia) son thread-safe.

También es thread-safe cualquier objeto que esté almacenado en la petición. El objeto `HttpServletRequest` sólo es accesible para cada una de las peticiones que se realicen al servidor. Por tanto, peticiones realizadas por dos usuarios diferentes, u otras peticiones realizadas por un mismo usuario, no van a compartir nunca un objeto `HttpServletRequest`.

Cualquier otra variable, no es thread-safe. Y aquí incluimos variables de instancia del Servlet (como la variable `valor` del ejemplo de la sección anterior), variables de la clase del Servlet (esto es, variables estáticas que hayas definido dentro de la clase del Servlet) y objetos que hayas guardado como atributos del contexto (que no sólo van a poder ser accedidos por un mismo Servlet cuyo código está siendo ejecutado de modo simultáneo por varios thread, sino que además pueden ser accedidos por código de distintos Servlets que se están ejecutando simultáneamente en el contenedor).

Tampoco son thread-safe los objetos que se guardan en la sesión del usuario. La sesión del usuario no se comparte entre distintos usuarios, así que en este caso no corremos el riesgo de que dos usuarios accedan a la vez al mismo Servlet. Pero sí que corremos el riesgo de que un mismo usuario realice dos peticiones simultáneas (por ejemplo, abriendo dos lengüetas en su navegador web) y por tanto haya dos thread en el servidor para responder a dichas peticiones, ambos accediendo a la misma sesión.

9.3 Una primera (cutre) solución a los problemas de concurrencia

Una solución, que actualmente está "drepecada" y no se recomienda su uso para solucionar algunos problemas de concurrencia es el implementar la interface `SingleThreadModel`. Cuando un Servlet implementa esta interface sin métodos el contenedor nos garantiza que un único thread va a ejecutar de modo simultáneo el código de cada instancia del Servlet. Por tanto, dejamos de tener problemas de concurrencia con las variables instancia (los problemas más comunes). No obstante, los problemas de concurrencia con todos los demás tipos de variables mencionados en el apartado anterior persisten.

Cuando implementamos la interface `SingleThreadModel` el contenedor puede optar por varias opciones para garantizar que un único thread ejecuta el código de cada instancia de nuestro Servlet. Si el contenedor crea una única instancia de dicho Servlet (como se suele hacer con los Servlet normales) el contenedor creará una cola de espera para las peticiones, y hasta que una petición termine no permitirá que la siguiente se ejecute. Por tanto, el rendimiento de la aplicación se va a degradar considerablemente.

Otra opción, que es la realmente implementada por la mayor parte de los contenedores de Servlets, es crear un pool de instancias de nuestro Servlet, en vez de tener una única instancia, e ir repartiendo las peticiones que lleguen entre las distintas instancias.

La ventaja de emplear la interface `SingleThreadModel` es que es tremendamente fácil. Para resolver el problema de concurrencia de `ProblemaConcurrenciaServlet` lo único que tenemos que hacer es que implemente esta interfaz. El contenedor se encargará del resto. Esto es lo que hace `STMConcurrenciaServlet`:

```
public class STMConcurrenciaServlet extends HttpServlet implements  
SingleThreadModel {...
```

el resto del código de `STMConcurrenciaServlet` es idéntico al de `ProblemaConcurrenciaServlet`. `STMConcurrenciaServlet.html` es una página web que incluye 25 frames, cada uno de los cuales incluye el resultado generado por

STMConcurrenciaServlet. Si abres esta página web en dos navegadores diferentes de modo simultáneo, podrás comprobar como en este caso valor siempre vale 0. Aunque hay dos thread ejecutando nuestro código, Glassfish ha creado dos instancias diferentes de STMConcurrenciaServlet, y cada uno de los thread trabaja sobre una instancia diferente. Cada una de estas instancias tiene su propia variable valor, así que no comparten este recurso.

Las desventajas de esta solución es que emplea un mecanismo de gestión de concurrencia muy burdo y muy poco eficiente. Si el contenedor de Servlets opta por crear una cola con las peticiones y tener una única instancia del Servlet, la última petición que llega al servidor tiene que esperar a que se completen todas las demás para ser gestionada. Esto hará que nuestra aplicación web tenga un comportamiento muy pobre. Si el contenedor crea múltiples instancias de nuestro Servlet, está consumiendo más recursos que si crease una sola instancia. Y a pesar de haber creado múltiples instancias, no se han resuelto todos los problemas de concurrencia. Sólo se han resuelto los problemas de concurrencia relativos a las variables de instancia, pero seguimos teniendo problemas de concurrencia con las variables de clase, con las almacenadas en el contexto y con las almacenadas en la sesión. Actualmente, se recomienda no emplear esta solución para resolver problemas de concurrencia.

9.4 Una mejor solución a los problemas de concurrencia

La solución actualmente recomendada a los problemas de concurrencia es emplear sincronización manual sobre las variables que puedan provocar estos problemas. Esto es lo que hace el código de ManualConcurrenciaServlet. En este código, tenemos una variable de instancia adicional: lockDeValor, cuyo tipo de dato es Object. Hemos tenido que crear esta variable porque sólo se puede aplicar sincronización sobre referencias, y no sobre tipos de datos primitivos (valor es un tipo de dato primitivo). El código del Servlet adquiere un lock empleando esta referencia antes de comenzar a modificar la variable valor (synchronized (lockDeValor)) y no lo libera hasta que ha terminado de trabajar con esta variable; esto es, hasta que no la ha mostrado en el HTML.

```

@WebServlet(name = "ManualConcurrenciaServlet", urlPatterns =
{"/ManualConcurrenciaServlet"})
public class ManualConcurrenciaServlet extends HttpServlet {

    public static final int REPETICIONES = 3000000;
    public final Object lockDeValor = new Object();
    private int valor;

    protected void processRequest(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
        synchronized (lockDeValor) {
            valor = 0;
            int[] numerosAleatorios = new int[REPETICIONES];
            for (int i = 0; i < REPETICIONES; i++) {
                double a = Math.random() * 10;
                numerosAleatorios[i] = (int) a;
                valor += numerosAleatorios[i];
            }
            for (int i = 0; i < REPETICIONES; i++) {
                valor -= numerosAleatorios[i];
            }

            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out = response.getWriter();
            try {
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Problema de
concurrencia</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("El resultado es " + valor + "");
                out.println("</body>");
                out.println("</html>");
            } finally {
                out.close();
            }
        }
    }
}

```

Para probar esta clase el lector puede usar la página `ManualConcurrenciaServlet.html`.

Personalmente, recomiendo que nunca se emplee para sincronizar un bloque de código dentro de un Servlet `synchronized (this)`. Esta sincronización es de grano muy grueso. Seguramente podemos hacer algo mejor en la mayor parte de los casos. Posiblemente haya varias variables que sean accedidas de modo concurrente, y posiblemente no todo el código del Servlet acceda a todas las variables a la vez, sino que habrá un fragmento de código que accede a una variable, después otro fragmento que trabaja con otra variable, etcétera. En este caso, es preferible tener varios bloques `synchronized ()` con el ámbito más reducido posible. De este modo, será posible que varios thread estén ejecutando de modo simultáneo secciones diferentes de nuestro código fuente, secciones tales que cada una de ellas puede ejecutarse concurrentemente con las demás.

En el caso de nuestro simple ejemplo académico, sólo tenemos una única variable instancia (valor) a la cual estamos accediendo continuamente para sumar, restar e incrustar en el HTML. Por ello al final necesitamos sincronizar todo el bloque de código. A menudo en ejemplos reales se puede encontrar una solución mejor.

Finalmente resaltar que esta solución, aunque en nuestro ejemplo la hayamos empleado sobre una variable de instancia, puede emplearse sobre cualquier variable susceptible de padecer problemas de concurrencia, sea ésta de sesión, de contexto, de clase o de instancia.

10 Bases de datos y los Servlets

Muy a menudo las aplicaciones web emplean una base de datos relacional para persistir información. Este documento no pretende ser un tutorial de sobre cómo trabajar con bases de datos desde Java. Si el lector no tiene experiencia previa con ello, le recomiendo que le eche un vistazo al tutorial "El ABC de JDBC": http://www.javahispano.org/contenidos/es/el_abc_de_jdbc/.

Tenemos varias alternativas para realizar la persistencia de la información a la base de datos. Una, que raramente es aconsejable, es emplear directamente el API JDBC como lo haríamos en una aplicación Java SE. Otra, es emplear un pool de conexiones. La tercera, que no será cubierta en este tutorial, es emplear un motor de persistencia como JPA.

10.1 Empleando JDBC directamente

Siguiendo nuestra línea de ejemplos ridículamente sencillos, vamos a construir un Servlet que recibe dos campos de un formulario: el nombre de una persona y su edad. El Servlet los recupera y los guarda en una base de datos. La conexión con la base de datos se establece en el método `init` del Servlet:

```
@WebServlet(name = "JDBCServlet", urlPatterns = {"/JDBCServlet"})
public class JDBCServlet extends HttpServlet {

    private Statement statment = null;
    private Connection conexion = null;

    @Override
    public void init(ServletConfig config) {
        try {
            Class.forName("org.apache.derby.jdbc.ClientDriver");
            conexion = DriverManager.getConnection(
                "jdbc:derby://localhost:1527/HOLAJDBC", "administrador",
                "administrador");
        }
    }
}
```

```

        statment = conexion.createStatement();
    } catch (ClassNotFoundException ex) {
        Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
            "No se pudo cargar el driver de la base de datos", ex);
    } catch (SQLException ex) {
        Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
            "No se pudo obtener la conexión a la base de datos", ex);
    }
}

```

Si este método se ejecuta con éxito, tendremos una variable instancia tipo Statement que nos permitirá enviar consultas a la base de datos. Este Servlet será nuestro primer Servlet que realice una acción diferente cuando es alcanzado por un método http GET y por un método POST. Este último método suele emplearse para enviar algún contenido al servidor. Este será el método que empleemos para enviar el nombre y la edad de la persona que vamos a guardar en la base de datos.

En el método doPost de nuestro Servlet recuperamos los parámetros del formulario (el fichero que se encuentra en tema10/crearpersona.html) y construye una consulta tipo insert para la base de datos. Si hay algún problema en la consulta llamará a un método que realiza logging del problema y redirige la petición del usuario a una página de error estática (/tema10/error.html). Si la consulta tiene éxito, redirigimos al usuario de nuevo al formulario que le permite crear otra entrada la base de datos.

```

@Override
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    String nombre = request.getParameter("nombre");
    int edad;
    try {
        edad = Integer.parseInt(request.getParameter("edad"));
    } catch (NumberFormatException e) {
        edad = -1;
    }
    ServletContext contexto = request.getServletContext();
}

```

```

        String query = null;

        try {
            query = "insert into\"PERSONAS\" values('"
                + nombre + "','" + edad + ")";
            synchronized(statement){
                statement.executeUpdate(query);
            }
        } catch (SQLException ex) {
            gestionarErrorEnConsultaSQL(ex, request, response);
        }

        RequestDispatcher paginaAltas =
contexto.getRequestDispatcher("/tema10/crearpersona.html");
        paginaAltas.forward(request, response);

    }

    private void gestionarErrorEnConsultaSQL(SQLException ex,
HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException {
        ServletContext contexto = request.getServletContext();
        Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
"No se pudo ejecutar la consulta contra la base de datos", ex);
        RequestDispatcher paginaError =
contexto.getRequestDispatcher("/tema10/error.html");
        paginaError.forward(request, response);
    }
}

```

Observa como el código emplea sincronización sobre el objeto Statement. Este objeto es una variable de instancia y puede ser compartida por múltiples peticiones. Observa como en esta ocasión la sincronización afectada a una única línea del cuerpo del método del Servlet. Todas las demás líneas del método pueden ser ejecutadas en paralelo por múltiples threads; sólo esta línea es la que puede ser ejecutada por un único thread de modo concurrente. Mientras un thread está ejecutando esta línea, nada impide que otros threads estén ejecutando otras líneas del cuerpo del método.

Para que este ejemplo funcione correctamente, es necesario tener una base de datos creada con Derby (una base de datos que forma parte del JDK 6) en el equipo local corriendo en el puerto 1527. La base de datos tiene una única tabla con nombre PERSONAS con un campo de texto (el nombre) y un campo numérico (la edad). El comando empleado en la creación de la base de datos del ejemplo fue:

```
CREATE TABLE PERSONAS (  
    NOMBRE VARCHAR(100),  
    EDAD INT);
```

También será necesario tener el driver de la base de datos en el CLASSPATH del proyecto. En los videos que acompañan este tutorial se muestra cómo crear dicha base de datos empleando Netbeans.

crearpersona.html realmente tienen dos formularios. El primero envía al servidor el nombre de la persona y su edad empleando el método POST. El segundo simplemente realiza una petición GET contra JDBCServlet. El método doGet de JDBCServlet lista todas las personas introducidas en la base de datos, junto con su edad. Para ello, realizamos un SELECT * (esperemos que no haya muchos registros...) en la base de datos e iteramos sobre el ResultSet. Observa nuevamente como volvemos a emplear sincronización sobre el objeto Statement.

```
@Override  
protected void doGet(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 Listar Personas</title>");  
        out.println("</head>");  
        out.println("<body>");  
        out.println("<h1>Lista de las personas:</h1>");  
        out.println("<ul>");
```



```

        String query = null;
        query = "select *" + "from \"PERSONAS\"";
        ResultSet resultSet = null;
        try {
            synchronized(statement){
                resultSet = statement.executeQuery(query);
            }
            while (resultSet.next()) {
                out.println("<li>" + resultSet.getString("NOMBRE")
+ " edad: " + resultSet.getInt("EDAD") + "</li>");
            }
        }
        catch (SQLException ex) {
            gestionarErrorEnConsultaSQL(ex, request, response);
        }
        finally {
            if (resultSet != null) {
                try {
                    resultSet.close();
                } catch (SQLException ex) {
                    Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
                        "No se pudo cerrar el Resulset", ex);
                }
            }
        }
        out.println("</ul>");
        out.println("</body>");
        out.println("</html>");
    } finally {
        out.close();
    }
}

```

Este Servlet crea en su método `init ()` un recurso (la conexión con la base de datos) que es necesario liberar de modo explícito cuando el Servlet es destruido por el contenedor. Por tanto vamos a tener que sobrescribir el método `destroy ()` de la clase Servlet (este es el método que es invocado por el contenedor justo antes de destruir el Servlet).

```

@Override
public void destroy () {
    try {
        statment.close();
    } catch (SQLException ex) {
        Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
            "No se pudo cerrar el objeto Statement", ex);
    }
    finally {
        try {
            conexion.close();
        } catch (SQLException ex) {
            Logger.getLogger(JDBCServlet.class.getName()).log(Level.SEVERE,
                "No se pudo cerrar el objeto Conexion", ex);
        }
    }
}
}

```

Esta estrategia para acceder a la base de datos tiene una desventaja: aunque hemos limitado la sincronización al máximo, (una única sentencia en cada caso) al final tenemos una única conexión a la base de datos, y por tanto sólo es posible ejecutar de modo simultáneo una única consulta contra ella. Si varios usuarios están empleando la aplicación, todas esas consultas acabarán en una cola y se irán ejecutando de una en una. Esto está lejos de ser óptimo desde el punto de vista de rendimiento.

El lector podría pensar que una posible opción es crear el objeto Statment y el objeto Connection dentro de los métodos doGet y doPost, de tal modo que se conviertan en variables confinadas en el stack. Tendríamos que crear ambas variables dentro de los métodos, ya que si sólo creamos el objeto Statement en el método tendríamos que aplicar sincronización sobre el objeto Connection y nos encontraríamos en exactamente la misma situación que en el caso anterior. Esta opción sería también bastante poco óptima. Abrir y cerrar conexiones contra una base de datos son operaciones costosas. No es una buena idea realizar esta operación por cada petición http que recibamos. Probablemente, la primera aproximación (aún siendo mala) sea mejor que ésta.

La solución óptima sería no tener una única conexión con la base de datos, sino varias. Cuando llega una petición http, empleamos una conexión que no se esté usando en ese momento. Cuando una petición http termina, liberamos esa conexión. Es decir, implementamos manualmente un pool de conexiones. Esto es tedioso, y propenso a fallos. Es complicado realizar una buena implementación altamente eficiente, sin problemas de concurrencia y que libere adecuadamente todos sus recursos. No obstante, podríamos hacerlo. Pero no va a hacer falta: ya hay muchos implementados en Java y podemos emplearlos.

10.2 Empleando el pool de conexiones

La forma más adecuada de emplear las conexiones a la base de datos dentro de un servidor de aplicaciones Java EE es emplear un pool de conexiones que esté gestionado por el propio servidor. Para ello, empleando la consola de administración del servidor, deberemos configurar un pool de conexiones dentro de él. En el tercer video que acompaña a este tutorial mostraremos como hacer esto con Glassfish. Además de configurarlo, deberemos asignarle un nombre JNDI (Java Naming and Directory Interface) a este recurso. JNDI es un API que permite acceder de modo uniforme a múltiples servicios de directorio. Nos va a permitir localizar estos servicios a través de su nombre.

Para configurar adecuadamente el pool de conexiones en la aplicación, deberemos modificar el descriptor de despliegue de la aplicación declarando el recurso:

```
<resource-ref>
  <res-ref-name>jdbc/HolaMundoPool</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

jdbc/HolaMundoPool es el nombre que le dimos en la consola de administración al pool. Además, en Glassfish deberemos modificar también un descriptor de despliegue propio de este contenedor cuyo nombre es sun-web.xml:

```
<resource-ref>
    <res-ref-name>jdbc/HolaMundoPool</res-ref-name>
    <jndi-name>jdbc/HolaMundoPool</jndi-name>
</resource-ref>
```

ConexionPoolServlet es un Servlet que hace exactamente lo mismo que JDBCServlet, pero obtiene su conexión a partir de un pool de conexiones con nombre jdbc/HolaMundoPool. Los recursos JDBC deben ser almacenados en el contexto JNDI java:comp/env/jdbc. Para localizar nuestro pool, creamos un objeto InitialContext() y empleando su método lookup recuperamos un objeto Context ligado a los objetos de subespacio "java:comp/env". A continuación, volviendo a emplear el método lookup del nuevo objeto de tipo Context, recuperamos nuestro pool de conexiones, cuyo nombre es "jdbc/HolaMundoPool". La interface con este pool es un objeto de tipo DataSource, el cual tiene un método Connection getConnection(). Este método nos devolverá una conexión del pool, que podremos emplear para ejecutar nuestras sentencias SQL.

Por tanto, ahora el método init () de ConexionPoolServlet deberá obtener el DataSource que previamente ha sido registrado en el servidor de aplicaciones:

```
DataSource pool;

@Override
public void init(ServletConfig config) throws ServletException {
    Context env = null;
    try {
        env = (Context) new
InitialContext().lookup("java:comp/env");
        pool = (DataSource) env.lookup("jdbc/HolaMundoPool");
        if (pool == null) {
            throw new ServletException("No se encontró el
DataSource");
        }
    }
```

```

        }
        catch (NamingException ne) {
            throw new ServletException(ne.getMessage());
        }
    }
}

```

Y los métodos `doGet` y `doPost` de `ConexionPoolServlet` emplearán este `DataSource` para conseguir una conexión:

```

...
String nombre = request.getParameter("nombre");
int edad;
try {
    edad =
Integer.parseInt(request.getParameter("edad").trim());
} catch (NumberFormatException e) {
    edad = -1;
}
ServletContext contexto = request.getServletContext();
String query = null;

try {

    Connection conexion = pool.getConnection();
    Statement statment = conexion.createStatement();
    query = "insert into\"PERSONAS\" values('\"
        + nombre + "\",\" + edad + \"\")";
    statment.executeUpdate(query);
    statment.close();
    conexion.close();
} catch (SQLException ex) {
    gestionarErrorEnConsultaSQL(ex, request, response);
}

...

```

En este código no se está creando realmente ninguna conexión, sino que estamos pidiéndole al pool de conexiones una que esté libre en ese momento. De este modo, contamos con múltiples conexiones que permitirán ejecutar de modo simultáneo múltiples consultas contra la base de datos, pero no estamos continuamente abriendo y cerrando conexiones contra ésta. De un modo similar, el método `close` realmente no está cerrando la conexión, sino liberándola al pool. Observa también como no hace falta emplear sincronización, ya que el propio pool de conexiones se encarga de no dar la misma conexión a dos Servlets de modo simultáneo.

Para probar este segundo Servlet puedes emplear el formulario `/tema10/crearpersonaPool.html`.

Antes de jugar con el código fuente de este capítulo, especialmente con el ejemplo que emplea el pool de conexiones, te recomiendo que veas el tercer video que acompaña a este tutorial. En él se muestra como emplear la funcionalidad que tiene Netbeans para ayudarnos a trabajar con bases de datos y con servidores de aplicaciones, así como a definir un pool de conexiones en la consola de administración de Glassfish.



Video

<https://www.youtube.com/watch?v=Pq7euWVGKxc>

11 Reaccionando a eventos en el contenedor: listeners y filtros

En ocasiones, antes de arrancar nuestra aplicación web querremos realizar ciertas tareas de inicialización. Por ejemplo, quizás tengamos un catálogo de productos almacenado en la base de datos. Salvo que estemos tratando con cantidades muy grandes de productos, podría tener sentido cargar todos esos productos en memoria (por ejemplo, almacenándolos en el contexto de la aplicación) para que cuando consultemos el catálogo las operaciones vayan más rápido. O quizás, por cualquier motivo, queramos abrir un archivo con el cual vamos a trabajar en la aplicación web. En este último caso, cuando la aplicación web deje de ejecutarse deberíamos cerrar el archivo.

También resulta interesante a veces realizar algún tipo de acción cada vez que se crea una sesión del usuario, o simplemente cada vez que un determinado Servlet es usado. Esto podría valernos, por ejemplo, para crear estadísticas de uso de nuestra aplicación, o estadísticas sobre los patrones de comportamiento de los usuarios de la aplicación.

En este tema vamos a ver cómo responder a este tipo de eventos del contenedor y a interceptar peticiones que llegan al servidor.

11.1 ServletContextListener: arranque y parada de la aplicación

La interfaz `ServletContextListener` nos permite crear clases que pueden responder ante eventos de arranque y parada de la aplicación web. La interfaz `WebServletContextListener` define dos métodos:

- **`public void contextInitialized(ServletContextEvent sce)`**: este método es invocado cuando la aplicación web es arrancada. Este método será ejecutado antes de que cualquier Servlet o filtro de la aplicación (en este tema veremos qué

son los filtros) sea inicializado (y por tanto, antes de que puedan responder a una petición de un usuario).

- **public void contextDestroyed(ServletContextEvent sce):** este método es invocado cuando la aplicación va a ser detenida. Cuando este método es invocado, el método destroy() de todos los Servlets y filtros del contenedor ha sido ejecutado.

Una vez hayamos creado una clase que implemente esta interfaz, tenemos que registrarla en el servidor de aplicaciones. Esto podemos hacerlo a través del descriptor de despliegue:

```
<listener>
  <listener-class>
    tema11.ServletListenerDeContexto
  </listener-class>
</listener>
```

O empleando la anotación `@WebListener` si estamos empleando un contenedor de Servlets que soporte la especificación 3.0 o posterior de los Servlets:

```
@WebListener
public class ServletListenerDeContexto implements
ServletContextListener {
  ...
}
```

Es posible definir en una aplicación web cuantos listeners de contexto queramos. Si estos son declarados empleando el descriptor de despliegue, serán invocados en el mismo orden en el cual fueron definidos en el descriptor de despliegue cuando la aplicación arranque, y en orden inverso cuando la aplicación se detenga.

A continuación mostramos un ejemplo de clase que implementa la interfaz. Esta clase se limita a mostrar mensajes en la consola cuando el servidor arranca, mostrando el

contexto en el cual está arrancando la aplicación y el parámetro de configuración que se le ha pasado:

```
<context-param>
    <param-name>parametro</param-name>
    <param-value>valor </param-value>
</context-param>
```

Cuando la aplicación es detenida, mostrará otro mensaje en la consola y volverá a mostrar el contexto de la aplicación que se está deteniendo.

```
package temall;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class ServletListenerDeContexto implements
ServletContextListener {

    public void contextInitialized(ServletContextEvent contextEvent) {
        ServletContext contexto= contextEvent.getServletContext();
        System.out.println("Aplicacion arrancando en el contexto " +
            contexto.getContextPath() +
            " con el siguiente parámetro de configuracion: " +
            contexto.getInitParameter("parametro"));
    }

    public void contextDestroyed(ServletContextEvent contextEvent) {
        ServletContext contexto= contextEvent.getServletContext();
        System.out.println("Aplicacion del contexto " +
            contexto.getContextPath() + " deteniéndose.");
    }
}
```

11.2 Escuchando eventos de sesión

También es posible dentro de una aplicación web escuchar los eventos relativos a la creación y destrucción de las sesiones de los usuarios. Para ello debemos implementar la interfaz `HttpSessionListener` que define los siguientes métodos:

- **`public void sessionCreated(HttpSessionEvent se)`:** método que será invocado cada vez que se cree una sesión en la aplicación. A partir del evento que se le pasa como parámetro es posible obtener dicha sesión.
- **`public void sessionDestroyed(HttpSessionEvent se)`:** método que será invocado cada vez que se destruya una sesión en la aplicación. A partir del evento que se le pasa como parámetro es posible obtener dicha sesión.

Al igual que sucedía con los listeners de contexto, es necesario registrar los listeners de sesión, o bien a través de la anotación `@WebListener` o en el descriptor de despliegue:

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

Una de las utilidades de escuchar estas sesiones puede ser realizar login relativo a la creación y destrucción de sesiones. Esto es lo que (de un modo chapucero) realiza el siguiente código. Cada vez que se crea una sesión, muestra un mensaje en la consola indicando la hora en la cual fue creada la sesión, su identificador y su máximo tiempo de inactividad. Cada vez que se destruye una sesión, muestra otro mensaje en la consola mostrando el identificador de la sesión.

```

package temall;

import java.util.Date;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class ListenerDeSesion implements HttpSessionListener {

    public void sessionCreated(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.print("A las " + getTime() + " se creo la sesion
con ID: " +
            session.getId() + " MaxInactiveInterval=" +
            session.getMaxInactiveInterval());
    }

    public void sessionDestroyed(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        System.out.println("A las " + getTime() + " se destruyo la
sesion con ID: "
            + session.getId());
    }

    private String getTime() {
        return new Date(System.currentTimeMillis()).toString();
    }
}

```

También existen otros Listeners como **HttpSessionBindingListener**, que nos permite obtener notificaciones del contenedor cuando un objeto es ligado o desligado a una sesión. **HttpSessionAttributeListener** permite obtener notificaciones cuando hay cambios en los atributos contenidos en una sesión. **HttpSessionActivationListener** permite saber si el contenedor está haciendo pasiva una sesión, es esto es, la está eliminando de la memoria principal y pasándola a un dispositivo de almacenamiento secundario (habitualmente esto sucede cuando una sesión lleva bastante tiempo sin

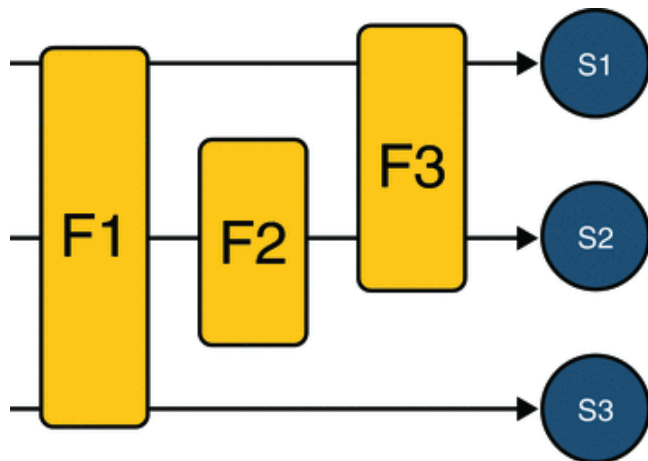
usarse, aunque no ha expirado, y se está acabando la memoria). Todas estas interfaces se encuentran en el paquete `javax.servlet.http`, cuyo javadoc puede consultar el lector si está interesado en obtener más información sobre ellas.

11.3 Filtros

En una aplicación Java EE 6 es posible definir una serie de filtros que interceptan las peticiones que llegan desde el usuario. Estos filtros definen patrones de las URLs que desean interceptar. Cualquier petición que llegue al servidor empleando una URL compatible con dicho patrón será pasada al filtro antes de llegar al recurso estático o dinámico al cual se dirige la petición.

En una aplicación web puede haber definidos múltiples filtros. En este caso, los filtros serán invocados en el orden en el cual fueron definidos en el descriptor de despliegue de la aplicación. El hecho de que pueda haber múltiples filtros hace que a menudo se hable de una "cadena de filtros". No todos los filtros tienen que interceptar todas las peticiones, o las mismas peticiones necesariamente. Pero puede ser que dos o más filtros hayan indicado que quieren interceptar patrones de URL que en ocasiones son compatibles el uno con el otro, y por tanto todos esos filtros deberán ser aplicados a la petición. En este caso, la petición pasará primero por el primer filtro definido, el cual una vez haya hecho su trabajo deberá pasar la petición al siguiente filtro de la cadena, y así sucesivamente. El último filtro de la cadena pasa la petición al recurso solicitado por el usuario; esto suponiendo que los filtros finalmente decidan que la petición va a llegar al recurso. Podría darse la situación de que los filtros decidan que la petición no está autorizada, o no debe llegar a ese recurso por cualquier otro motivo, y decidan redirigir la petición a algún otro recurso. O incluso que ellos mismos generen la respuesta.

La siguiente figura ilustra el funcionamiento de los filtros: con círculos se representan tres Servlets. Las flechas representan peticiones. Hay peticiones que pasan por uno de los filtros, otras por dos de los filtros, y otras por los tres filtros. También podría haber peticiones que no pasen por ningún filtro.



Para definir filtros debemos crear una clase que implemente la interfaz `javax.servlet.Filter`. Esta clase define tres métodos:

- **`public void init(FilterConfig filterConfig)`**: este método es equivalente al método de inicialización de los Servlets. Cuando el filtro es creado, se invoca a este método para inicializarlo. En el objeto que se le pasa como argumento es posible que encontremos información relativa a parámetros de configuración del filtro que han sido especificados en el descriptor de despliegue de la aplicación o mediante anotaciones.
- **`public void destroy()`**: este método es equivalente al método de destrucción de los Servlets. Es invocado cuando el filtro va a ser destruido.
- **`public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`**: éste es el método que es invocado cada vez que ha llegado una petición compatible con los patrones de URL que desea interceptar el filtro. Los dos primeros objetos que se le pasan son los objetos de petición y respuesta asociados a la petición del usuario. El filtro puede añadir o modificar parámetros en estos objetos, o incluso en ocasiones envuelve estos objetos en un wrapper modificando su funcionalidad (es decir, encapsula los objetos originales en otros objetos que presentan la misma interfaz, y cuyos métodos posiblemente acaben delegando su acción a los métodos del objeto original, pero que también probablemente realicen alguna acción adicional). El tercer parámetro del

método, el objeto `FilterChain`, tiene un único método `doFilter(ServletRequest request, ServletResponse response)` al cual deberemos invocar (en caso que no hayamos decidido detener la petición y redireccionarla a otro recurso) una vez este filtro haya terminado su trabajo para pasar la petición al siguiente filtro, o para enviarla finalmente al recurso que la debe recibir.

Podemos registrar un filtro empleando el descriptor de despliegue de la aplicación:

```
<filter>
  <filter-name>Filtro</filter-name>
  <filter-class>tema11.Filtro</filter-class>
</filter>
```

En este caso, también deberemos indicar en el descriptor de despliegue los patrones de URL que deseamos interceptar con este filtro:

```
<filter-mapping>
  <filter-name>Filtro</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Mediante esta declaración estaríamos indicando que queremos que el filtro intercepte todas las peticiones a nuestra aplicación. Mediante esta otra:

```
<filter-mapping>
  <filter-name>Filtro</filter-name>
  <url-pattern>*.map</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Filtro</filter-name>
  <url-pattern>/Catalogo/*</url-pattern>
```

```
</filter-mapping>
```

Estaríamos indicando que queremos que nuestro filtro intercepte todas las peticiones a todos los recursos que terminen con ".map", o a todos los recursos que se encuentren en el directorio "Catalogo" de nuestra aplicación.

Otra forma de registrar los filtros es a través de la anotación `@WebFilter`:

```
@WebFilter(filterName = "Filtro",  
urlPatterns = {"/*"},  
initParams = {  
    @WebInitParam(name = "parametro", value = "valor")})
```

En este segundo caso, además de especificar que la clase donde hemos puesto la anotación es un filtro, e indicar qué URLs queremos que intercepte, estamos pasando parámetros de inicialización al filtro, parámetros que podríamos recuperar a partir del objeto `FilterConfig`, y en base a los cuales podríamos realizar distintas acciones en el método `init (FilterConfig filterConfig)` del filtro.

11.3.1 Un ejemplo de filtro

A continuación mostramos un ejemplo de filtro. El filtro intercepta todas las peticiones que llegan a nuestra aplicación e imprime por consola un mensaje indicando la IP de la máquina desde la cual el usuario ha realizado la petición. En el caso de que la aplicación se corresponda con un envío de un formulario, va a extraer todos los parámetros del formulario y también los va a mostrar por consola.

```
package temall;  
  
import java.io.IOException;  
import java.util.Enumeration;  
import javax.servlet.Filter;  
import javax.servlet.FilterChain;
```

```

import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.HttpServletRequest;
import javax.servlet.HttpServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter(filterName = "Filtro", urlPatterns = {"//*"})
public class Filtro implements Filter {

    public Filtro() {

    }

    public void doFilter(HttpServletRequest request, HttpServletResponse
response, FilterChain chain) throws IOException, ServletException {

        StringBuilder buffer = new StringBuilder();
        for (Enumeration parametrosPeticion =
request.getParameterNames();
            parametrosPeticion.hasMoreElements();) {
            String nombreParametro = (String)
parametrosPeticion.nextElement();
            String valoresDeParametros[] =
request.getParameterValues(nombreParametro);
            int numeroParametros = valoresDeParametros.length;
            buffer.append(nombreParametro);
            buffer.append(" = ");
            for (int i = 0; i < numeroParametros; i++) {
                buffer.append(valoresDeParametros[i]);
                buffer.append(" | ");
                if (i < numeroParametros - 1) {
                    buffer.append(",");
                }
            }
        }
        System.out.println("Recibida peticion Desde la IP: "
            + request.getRemoteAddr());
        if (!buffer.toString().equals("")) {
            System.out.println("\tla peticin tiene los parametros: " +
buffer);
        }
    }
}

```



```

        chain.doFilter(request, response);
    }

    public void destroy() {
    }

    public void init(FilterConfig filterConfig) {
    }
}

```

Observa como una vez el filtro ha hecho su trabajo, continúa la cadena de filtros reenviando la petición al siguiente filtro, o si no hay más filtros al recurso que debe recibirla:

```
chain.doFilter(request, response);
```

Si el lector accede al formulario

```
http://localhost:8080/contexto/tema6/FormularioSesion.html
```

y después teclea en el primer campo del formulario "esto es el atributo" y en el segundo "esto es el valor" y lo envía, ésta es la salida que produce en la consola de Glassfish el filtro:

```

INFO: Recibida peticion Desde la IP: 127.0.0.1
INFO: Recibida peticion Desde la IP: 127.0.0.1
INFO:          la peticin tiene los parametros: parametro = esto es el
atributo | valor = esto es el valor | accion = anhadir |

```

En el primer caso se nos muestra simplemente que hemos recibido la petición, ya que la petición era una petición GET y sin parámetros. En el segundo caso, además de indicar que se ha recibido la petición, nos está mostrando los parámetros (incluido el campo oculto del formulario). Recomendando al lector que experimente con este filtro empleando

los formularios que vienen con este curso, u otros que diseñe el mismo. Sería especialmente interesante que probase a definir distintos patrones de filtro, y a mostrar toda la información que fuese posible a acerca de las peticiones, o a interceptar las peticiones y redirigirlas a otro recurso.

En vista de este ejemplo, es fácil comprender la potencia de los filtros. Son una excelente herramienta para realizar logging de las aplicaciones. También pueden emplearse para añadir funcionalidad no relacionada con la lógica de negocio de la aplicación y que es completamente transversal a ésta, pero que hace falta en el mundo real muchas veces. Por ejemplo, autenticación, autorización, comprimir y/o descomprimir las peticiones y/o las respuestas del servidor, lanzar un trigger ante determinados eventos, permitir generar analíticas muy detalladas de uso de la aplicación web, realizar logging, etc.

12 Páginas JSP

Un Servlet podría verse como una herramienta que nos permite generar HTML desde código Java. Una de las partes más tediosas de la programación de Servlets es precisamente la generación del código HTML. Todos los ejemplos vistos en este tutorial son extremadamente simples, especialmente en cuanto a la generación de HTML se refiere. Siempre estamos generando la cantidad mínima de HTML para poder visualizar los resultados de las tareas que hemos llevado a cabo. En una web real, especialmente si queremos que tenga una apariencia razonablemente buena, tendremos que generar cantidades de HTML mucho mayores. Si hacemos esto desde un Servlet, será increíblemente tedioso.

Por un lado, no podremos tomar ventaja de herramientas gráficas como, por ejemplo, la herramienta opensource KompoZer (<http://kompozer.net/>). Estas herramientas nos permiten crear páginas web de un modo bastante visual (de un modo simplista, podríamos decir que son algo parecido al diseñador de swing que tiene incorporado Netbeans dentro, pero para HTML). Si bien podríamos emplear estas herramientas inicialmente, con la aproximación de los Servlets tendríamos que hacer copiar y pegar del HTML generado a nuestro código Java, y una vez que empotremos allí el código HTML no podemos volver a emplear la herramienta gráfica para editarlo.

Por otro lado, nuestro "código Java" con HTML empotrado sería enorme y su legibilidad bajaría considerablemente por estar lleno de cadenas de caracteres inmensas. Por no hablar de que cambiar el código HTML sería una tarea bastante tediosa.

Los Servlets son fundamentalmente programas Java. Y son una buena herramienta para hacer cosas relacionadas con programación, como la implementación de la lógica de negocio de una aplicación web, almacenar o recuperar datos de una base de datos, e incluso, hasta un cierto punto, definir la lógica de navegación de una aplicación web. Pero los Servlets son una herramienta pésima para crear la capa de presentación de la aplicación web (esto es, para generar el HTML que finalmente se va a renderizar en el navegador web del usuario).

Dentro de Java EE tenemos una herramienta mucho más adecuada para generar este HTML: las páginas JSP¹. Hasta un buen punto, las páginas JSP pueden considerarse justo lo contrario de un Servlet: son código HTML que tiene código Java empotrado. Pero son principalmente código HTML; simplemente empotrar pequeñas cantidades de código Java para crear sus partes dinámicas.

Desde un punto de vista funcional, las páginas JSP y los Servlets se complementan mutuamente. Las páginas JSP son un sitio nefasto para escribir grandes cantidades de código Java, y por tanto son un sitio malo para implementar la lógica de la aplicación (especialmente si esta lógica es compleja), para realizar manipulaciones complejas de datos (como por ejemplo almacenarlos o recuperarlos de bases de datos) o, en general, son un mal lugar para realizar cualquier tarea que se realiza de modo fácil con un programa Java. Para lo que son buenas es para generar HTML de modo dinámico.

A nivel de implementación, las páginas JSP son realmente Servlets (de ahí que sólo hablemos de contenedores de Servlets, y no de “contenedores de Servlets y páginas JSP”). Cuando un usuario de nuestra aplicación solicite por primera vez una página JSP, el contenedor web va a generar un archivo ".java" que contiene un Servlet que genera todo el HTML que contiene la página JSP, más el código Java que contenía dicha página. A continuación, compila este código Java generando el archivo Class correspondiente. Finalmente, despliega el Servlet resultante en el contenedor web, y envía la petición del usuario ha dicho Servlet.

Si todo este proceso te parece largo y complicado ¡estás en lo cierto!. La primera vez que una página JSP es alcanzada por una petición del usuario, el servidor va a tardar un tiempo anormalmente alto en responder. Va a tener que traducir la página JSP a código

¹ Realmente, en estos momentos la tecnología JSP se considera (al menos por parte de Oracle) como una tecnología legacy cuyo uso no se recomienda en la actualidad. En estos momentos se recomienda emplear Java Server Faces (JSF) en su lugar. Esta otra tecnología es más compleja de entender que JSP ya que envuelve bastante "magia" (cosas que suceden de modo transparente para el programador) difícil de comprender para un programador sin experiencia en aplicaciones web. Mi consejo para el lector es que se familiarice con la tecnología JSP; muchos conceptos e incluso sintaxis que va a aprender siguen siendo válidos para JSF. Y JSP es una tecnología mucho mejor que JSF desde el punto de vista educacional por ser más transparente (involucra menos "magia"), si bien no pongo en duda que para desarrollar aplicaciones empresariales reales la tecnología JSF, o algún otro framework web, a menudo son una mejor opción.

Java, compilarlo, y desplegar el nuevo Servlet en el contenedor. El usuario de la aplicación web (tú mismo cuando comiences experimentar) notará este tiempo anormalmente alto de respuesta. A partir de la primera petición, la página JSP ya ha sido transformada a un Servlet, y su velocidad de respuesta será similar a la de cualquier otro Servlet.

12.1 Despliegue de las páginas JSP

Las páginas JSP se sitúan dentro de un proyecto Java EE de un modo similar a las páginas HTML estáticas; esto es, en desarrollo suelen estar colgando de la raíz de un directorio con nombre "web", posiblemente organizadas en una estructura de directorios adecuada para nuestra aplicación. En despliegue, se sitúan en el directorio de nivel más alto de la aplicación web. Nuevamente, pueden estar organizadas en una estructura de directorios adecuada para la organización de la aplicación web.

A menudo no queremos que un usuario pueda llegar directamente una página JSP. Puede que esa página esté preparada, por ejemplo, para recibir los datos de un formulario. O que la página suponga que habrá algún dato guardado en la sesión del usuario. Sin embargo, si publicamos la página JSP como cualquier otra página HTML estática, nada le va a impedir al usuario teclear en su navegador la URL correspondiente con la página y acceder a ella directamente. Con las páginas HTML estáticas, esto no suele ser un problema. Con las página JSP, esto podría dar lugar a un error en la aplicación porque la página ha sido accedida de un modo que no estaba previsto.

Un pequeño truco que a menudo se emplea para resolver este problema es localizar las página JSP dentro del directorio /WEB-INF. Este directorio nunca es accesible directamente a través de una URL (entre otras cosas, porque en ese directorio van a estar las clases Java compiladas; si se pudiese acceder a ese directorio un hacker podría descargar esas clases, descompilarlas y a partir del código fuente ganar conocimiento acerca de la estructura interna de nuestra aplicación e, indirectamente, acerca de las máquinas en las que corre, información que le sería muy útil al realizar un ataque contra nuestro servidor). Si colocamos las página JSP en el directorio /WEB-INF el usuario nunca va a poder acceder a ellas directamente. Sólo va a poder llegar a ellas a través de una redirección realizada por un Servlet o por otra página JSP.

Obviamente, esta solución sólo es válida para aquellas páginas JSP a las que no nos interesa que el usuario pueda acceder a ellas directamente. Si queremos que el usuario tenga acceso directo a la página, tendremos que colocarla fuera de ese directorio.

Las páginas JSP no necesitan incluir ningún tipo de información en el descriptor de despliegue de la aplicación; la URL que permite acceder a ellas es simplemente la misma URL que permitiría su acceso si se tratase de un recurso estático. Si la página estuviese en el directorio /WEB-INF, como ya hemos indicado, no será accesible mediante ningún tipo de URL, sino que habrá que llegar a ella a través de una redirección.

12.2 Mi primera página JSP

A continuación mostramos una primera página JSP, que podrás encontrar en `tema12/HolaMundoJSP.jsp`

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
    <title>Hola mundo JSP</title>
  </head>
  <body>
    <h1>Hola mundo JSP</h1>
    <p>La fecha actual en el servidor es <%= 2+2 %> </p>
  </body>
</html>
```

Como podrás observar, su código es prácticamente idéntico al de una página HTML. La única diferencia es la etiqueta `<%= new java.util.Date() %>`. Esta etiqueta, como veremos más adelante, se emplea para empotrar dentro del documento HTML una expresión Java, que será transformada en una cadena de caracteres invocando a su método `toString()`. En este caso la expresión Java es simplemente `"2+2"`. Empleando

esa etiqueta estamos haciendo algo que no podemos hacer con HTML: mostrar la hora del servidor (no del cliente) en la página HTML.

La forma de acceder a esta página sería similar a si fuese un recurso estático; es decir, podemos hacerlo a través de la URL:

```
http://localhost:8080/contexto/tema12/HolaMundoJSP.jsp
```

Este ejemplo te permite hacerte una idea de la apariencia típica que van a tener las páginas JSP: HTML con código Java empotrado. Pasemos ahora a ver la sintaxis de las páginas JSP.

12.3 Sintaxis de las páginas JSP

Existen cinco elementos de script diferentes que podemos encontrar dentro de las páginas JSP: comentarios, directivas, declaraciones, expresiones y scriptlets. Veamos lo que son cada uno de ellos.

12.3.1 Comentarios en las páginas JSP

Existen tres tipos diferentes de comentarios que podemos incluir en las páginas JSP. El primer tipo es comentarios HTML estándar:

```
<!--AdivinarNumeros.jsp-->
```

Este mensaje se envía al ordenador del usuario cuando realiza la petición, aunque no se renderiza (como todos los comentarios HTML). Por tanto, estos comentarios también serán incluidos dentro del código del Servlet que será generado a partir de esta página JSP.

El siguiente tipo de comentario es el comentario de página JSP:

```
<!-- Comentario de página JSP-- %>
```

Estos comentarios sólo se ven en las páginas JSP. No se incluyen de ningún modo en el Servlet durante la fase de producción, ni se envían al navegador del usuario.

El último tipo de comentario es el comentario Java: donde sea posible escribir código Java en el cuerpo de la página JSP también será posible escribir un comentario Java. Estos comentarios se traducen al código fuente del Servlet que se generará a partir de la página JSP, pero no se envían al usuario cuando realiza una petición.

12.3.2 Etiquetas de declaración

Dentro de una página JSP podemos incluir declaraciones de variables instancia (que por tanto estarán accesible desde todo el código de la página JSP) o declaraciones de métodos, empleando la etiqueta:

```
<%! ... %>
```

Por ejemplo:

```
<%! int ultimo = 0;%>
```

estaría declarando una variable entera llamada "ultimo". Esta variable será una variable instancia del Servlet que se genere a partir de la página JSP. Y este código:

```
<%! public String hora() {  
    return (new java.util.Date()).toString();  
}%>
```


sería un método de dicho Servlet. En general, cualquier cosa que pongamos dentro de una página JSP dentro de un etiqueta `<%!... %>` será añadido al código del Servlet fuera de cualquier método.

Si necesitamos escribir algún tipo de código de inicialización de la página JSP deberemos emplear una de estas directivas para sobrescribir el método **`jspInit()`**, que es equivalente al método **`init(ServletConfig)`** de los Servlets. Para realizar labores de limpieza de recursos, deberemos sobrescribir el método **`jspDestroy()`**.

12.3.3 Etiquetas de scriptlet

Todo el código HTML que incluyamos dentro de la página JSP, así como las etiquetas de scriptlet, van a ser copiados a un método con nombre `_jspService()` del Servlet que se va a generar a partir de nuestra página JSP. El código HTML, será escrito a la salida tal cual. Para ello, tendrá que ser rodeado de comillas, como todas las cadenas de caracteres Java, y ser escrito al `OutputStream` de la respuesta del usuario. Los scriptlets, al igual que el código HTML, son copiados al método `_jspService()`, pero son considerados código Java y por tanto no se rodean entre comillas. Su sintaxis es:

```
| <% código Java %>
```

por ejemplo:

```
| <% int variableLocal = 0;%>
```

estaría definiendo una variable local del método `_jspService()`. Y el código:

```
| <% ultimo += 10; %>
```

le estaría sumando 10 a la variable ultimo.

El hecho de que el código de los scriptlets se copie al método `_jspService()`, junto con el HTML, nos permite hacer cosas como la siguiente:

```
<% if (saludar) { %>
    <p> ¡Hola!</p>
<% }
else { %>
    <p> Adios</p>
<% } %>
```

donde suponemos que la variable `saludar` es de tipo boolean. Si vale `true`, mostraremos en la página el saludo. En caso contrario, diremos "adiós". Recuerda; el código de los scriptlets va a ser copiado tal cual al método `_jspService()`, mientras que el HTML se va a mandar a la respuesta del usuario. Por tanto, en el método `_jspService()` el código que las líneas anteriores van a generar será:

```
if (saludar) {
    out.println("<p> ¡Hola!</p>");
}
else {
    out.println("<p> Adios</p>");
}
```

De este modo podemos hacer condicionales que elijan entre un fragmento de HTML u otro para mostrar en la salida que será enviada al usuario; o bucles que hagan que un determinado fragmento de HTML se repita múltiples veces.

12.3.4 Etiquetas de expresiones

Las etiquetas de expresiones van a incluirse en la salida generada por los Servlet, combinándose con el HTML que tienen alrededor empleando el operador "+" en el código del Servlet. Su sintaxis es:

```
<%= expresión Java %>
```

por ejemplo:

```
<p> El resultado de hacer 8*4 es: <%= 8*4 %> </p>
```

que dará lugar al siguiente código Java:

```
out.println("<p> El resultado de hacer 8*4 es: "+ 8*4 + "</p>");
```

Si en la expresión incluimos un objeto Java, como sucede siempre que se concatena una cadena de caracteres con un objeto Java, se invocará a su método `toString()`:

```
<p> La hora actual es: <%= new java.util.Date() %> </p>
```

Dará lugar a:

```
out.println("<p> La hora actual es: " + (new  
java.util.Date()).toString() + "</p>");
```

Veamos un pequeño ejemplo de página JSP que emplea declaraciones y scriptlets. Nuestra página tendrá una variable global que actúa a modo de contador. Comienza valiendo 0. Cada vez que el usuario alcanza la página, se genera una tabla donde se muestran los 10 números consecutivos al contador. El contador, al ser una variable global del Servlet, va a conservar su valor entre una visita y otra del usuario. Además, le mostraremos la hora en el servidor empleando un método que crea un objeto Date y lo convierte a una cadena de caracteres:

```
<!--SimpleJSP.jsp-->
<%! int ultimo = 0;%>

<%! public String hora() {
    return (new java.util.Date()).toString();
}%>

<html>
  <head><title>Simple JSP</title></head>
  <body>
    <h1>Una bonita tabla:</h1>
    <table border=2>
      <%
        for (int i = ultimo; i < ultimo + 10; i++) {%>
          <tr>
            <td>Numero</td>
            <td><%= i %></td>
          </tr>
        <%}
        ultimo += 10;
      %>
    </table>
    <p>La hora actual es: <%= hora() %></p>
  </body>
</html>
```

Nuevamente, resulta interesante resaltar este código de la página JSP:

```

<table border=2>
  <%
    for (int i = ultimo; i < ultimo + 10; i++) {%>
      <tr>
        <td>Numero</td>
        <td><%= i %></td>
      </tr>
    <%>
    ultimo += 10;
  %>
</table>

```

que acabará generando el siguiente código Java al ser traducido a un Servlet:

```

out.println("<table border=2>");
for (int i = ultimo; i < ultimo + 10; i++) {
    out.println("<tr>");
    out.println("<td>Numero</td>");
    out.println("<td>" + i + "</td>");
    out.println("</tr>");
}
ultimo += 10;
out.println("</table>");

```

12.3.5 Etiquetas de directivas

Las etiquetas de directiva tienen la siguiente sintaxis:

```

<%@ directiva [atributo= "valor"]* %>

```

Donde el contenido entre corchetes puede aparecer de 0 a n veces. Sólo vamos a ver dos tipos diferentes de directivas.

Directiva page

Esta directiva modifica la forma en la que se traduce la página JSP a un Servlet. Puede tomar diferentes atributos, aunque sólo puede aparecer una vez cada atributo dentro de una misma página JSP, a excepción del atributo **import**. Este atributo se emplea para indicar que la clase necesita importar un paquete o clase:

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

Es posible indicar con una única directiva que se debe importar varios paquetes:

```
<%@ page import= "java.util.*, java.text,java.sql.Date" % >
```

Otro posible atributo es **isThreadSafe**, que puede tomar valores true y false. Este atributo indica si es posible ejecutar esta página de modo concurrente por varios usuarios. Si toma valor true, significa que la página puede ser ejecutada por varios usuarios de modo concurrente sin problemas. Si toma valor false, significa que la página sólo puede atender a una única petición a la vez. En este caso, se hará que el Servlet correspondiente implemente la interfaz SingleThreadModel:

```
<%@ page isThreadSafe= "false" % >
```

Por defecto este atributo toma el valor true; es decir, se asume que la página puede ejecutarse de modo concurrente.

El atributo `errorPage` permite especificar otra página JSP que se va a encargar de gestionar las posibles excepciones que se lancen dentro de esta página. El valor de este atributo debe ser la URL de la página que va a gestionarlas excepciones, URL que debe ser relativa a la posición de la página web actual, o relativa al contexto de la aplicación:

```
<%@ page errorPage= "GestionaError.jsp" % >
```

El atributo **session** define si una página JSP puede participar en la sesión http del usuario, y por tanto acceder a su contenido o almacenar nuevo contenido en ella. Este atributo puede tomar dos valores: true y false. El valor por defecto es true; es decir, por defecto las página JSP participan en la sesión del usuario. Si queremos hacer que una página JSP no participe en la sesión podríamos incluir esta directiva:

```
<%@ page session= "false" %>
```

El atributo **isErrorPage** indica que la página JSP ha sido diseñada para ser página de error. Estas páginas tendrán acceso de modo automático a una variable implícita (más adelante en este capítulo hablaremos sobre las variables implícitas) que contiene la excepción generada en la página donde se lanzó la excepción.

El atributo **contentType** define el tipo MIME de la salida generada por la página. El valor por defecto es "text/html". El atributo **@page.encoding** permite especificar el juego de caracteres que se debe de emplear cuando la página sea enviada al navegador web. Es equivalente a la correspondiente etiqueta HTML.

Directiva include

La directiva include permite incluir un fragmento de HTML o de página JSP dentro de esta página JSP. Los fragmentos, como su propio nombre indica, son fragmentos de HTML o de página JSP, pero no páginas completas. Son fragmentos que pueden ser empotrados en una determinada sección de una página JSP. En muchos portales, hay contenido repetitivo como un pie de página con información de copyright o de contacto, los menús, la cabecera, etcétera. La idea de estas directiva es incluir este contenido que se repite en múltiples páginas en un fichero independiente, y empleando esta directiva incluir dicho contenido en cada una de las páginas que lo va a usar. De este modo, cuando es necesario modificar ese contenido basta con modificar el fichero independiente y no es necesario modificar todas las páginas donde aparece. Su sintaxis es:

```
<%@ include file="banner.jsp" %>
```

12.4 Variables implícitas en las páginas JSP

El contenedor de Servlets hace que todas las páginas JSP tengan acceso en las etiquetas de expresión y en las etiquetas de scriptlet (es decir, al código que se copia al método `_jspService()`) a un conjunto de variables que representan objetos que a menudo son necesarios para generar la respuesta del usuario, como por ejemplo el objeto correspondiente con la sesión del usuario, o el objeto `HttpRequest` o `HttpResponse`. Estas variables implícitas simplemente "están ahí" cuando estamos escribiendo código dentro de una página JSP. Estas variables implícitas son:

- **application**: objeto de tipo `javax.servlet.ServletContext` que se corresponde con el objeto `ServletContext` de esta aplicación.
- **config**: objeto de tipo `javax.servlet.ServletConfig` que se corresponde con el objeto `ServletConfig`.
- **exception**: objeto de tipo `java.lang.Throwable` que se corresponde con una excepción que fue lanzada en una página JSP que tenía especificada como página de error la página actual. Esta variable implícita sólo está presente en aquellas páginas JSP que hayan indicado que `isErrorPage= "true"` empleando la correspondiente directiva.
- **out**: objeto de tipo `javax.servlet.jsp.JspWriter`; este objeto se corresponde con el flujo de salida que se va a enviar al usuario.
- **page**: objeto de tipo `java.lang.Object` que se corresponde con la propia página JSP. Esta variable implícita es equivalente a `"this"` en Java.
- **PageContext**: objeto de tipo `javax.servlet.jsp.PageContext` que define un ámbito igual a la propia página.
- **request**: objeto de tipo `javax.servlet.HttpServletRequest` que se corresponde con el objeto petición del usuario.

- **response**: objeto de tipo `javax.servlet.HttpServletResponse` que se corresponde con el objeto respuesta.
- **session**: objeto de tipo `javax.servlet.http.HttpSession` que se corresponde con la sesión del usuario.

Aunque muchas de estas variables implícitas podrían obtenerse las unas de las otras (por ejemplo, podríamos obtener la sesión del usuario a través del objeto request) resulta útil para el desarrollador que siempre estén disponibles, sin necesidad de escribir código para obtenerlas.

12.5 Un ejemplo de una página JSP

A continuación mostramos una página JSP que genera un número aleatorio entre 1 y 100. La página presenta un formulario al usuario para que éste intente adivinar el número. Si el usuario lo adivina, se le muestra un mensaje de felicitación y se le permite volver a jugar al juego de adivinar el número. Si no, se le indica si el número a adivinar es mayor o menor que el que él ha introducido (la cadena de caracteres con este mensaje se genera empleando un método declarado en la página JSP). Además, la página JSP va contando el número de intentos que el usuario ha realizado para adivinar cada número.

```
<!--AdivinarNumeros.jsp-->

<%! int adivinar = (int) (java.lang.Math.random() * 100)+1;%>
<%! int intentos = 0;%>

<%! /*Este metodo genera un pequeño mensaje de ayuda*/

    public String ayuda(int numero) {
        if (numero > adivinar) {
            return "El numero es menor";
        }
        return "El numero es mayor";
    }%>
```

```

<html>
  <head><title>Adivinar el nmero</title></head>
  <body>
    <% String parametro = request.getParameter("adivinar");
      int numero;
      if (parametro != null) {
        numero = Integer.parseInt(parametro);
        intentos++;
      }
      else {
        numero = -1;
      }
      if (parametro != null && adivinar == numero) {%>
        <p> ¡Enhorabuena! Lo has adivinado despues de <%=
intentos%> intentos.</p>
        <% adivinar = (int) (java.lang.Math.random() * 100)+1;
          intentos = 0;
          %>
          ¿Quieres probar otra vez <a
href="/contexto/tema12/AdivinarNumeros.jsp"></a>?
        <% }
        else if (intentos == 0) {%>
          <p>Adivina el numero; esta entre uno y 100:</p>
        <% }
        else {%>
          <p>No es ese. <%= ayuda(numero)%>.</p>
          <p>Has hecho un total de <%= intentos%> intentos.</p>
        <% }%>

        <form method=get>
          ¿Cual es el numero? <input type=text name="adivinar">
          <input type=submit value="Submit">
        </form>
      </body>
</html>

```

Puedes encontrar esta página JSP en /tema12/AdivinarNumeros.jsp.

12.6 Tipos de errores en las páginas JSP

Cuando estamos desarrollando una página JSP podemos tener tres tipos de errores diferentes en ellas. El primer tipo de error sucede en el momento de la traducción de la página JSP al Servlet, y surge por problemas en las etiquetas de la página. Por ejemplo, puede ser que hayamos abierto una determinada etiqueta pero no la hayamos cerrado.

El segundo tipo de error son errores de compilación que se generan cuando el código Java del Servlet resultado de la traducción de la página JSP es compilado. Uno de estos errores podría surgir, por ejemplo, porque en tiempo de compilación no se encuentra una clase que estamos usando porque nos hemos olvidado de incluir en la página la correspondiente directiva importando dicha clase.

El tercer tipo de error son errores en tiempos de ejecución. Por ejemplo, que se genere una `NullPointerException` cuando un usuario realiza una petición contra la página JSP.

En general, depurar errores en las páginas JSP es complicado. Por un lado, muchos de estos errores pueden estarse produciendo en la compilación o ejecución del código de un Servlet que realmente nosotros no estamos viendo, ya que el código Java ha sido generado por el contenedor. Esto hace que sea difícil interpretar un error de compilación que se está produciendo en una línea de un archivo Java que tú no has escrito y no estás viendo; tú lo que estás viendo es tu página JSP.

Por otro lado, escribir código Java en las páginas JSP tampoco es óptimo. Los entornos de desarrollo no suelen proporcionar funcionalidad de edición de código Java tan avanzada en el editor de páginas JSP como lo hacen en el editor de código fuente. Aunque, por ejemplo, puedas tener autocompletado, seguramente no tendrás refactoring. Y los chequeos que el propio entorno de desarrollo realiza respecto a la sintaxis y corrección del código Java empotrado en una página JSP son mucho menores que los que realiza sobre el código Java normal.

Todo esto hace altamente desaconsejable incluir mucho código Java dentro de una página JSP. Lo ideal es que las páginas JSP se limiten a presentar resultados al usuario, y que no contengan la lógica de negocio de la aplicación. Esta lógica de negocio, habitualmente, se expresa mucho mejor en Java. Y el sitio adecuado para escribir código Java dentro de una aplicación web son los Servlets, no las páginas JSP. Las

páginas JSP son adecuadas para tareas relacionadas con presentación (generar el HTML).

13 Acciones estándar JSP

Las acciones estándar de las páginas JSP son etiquetas con aspecto XML que se pueden utilizar dentro de las páginas JSP para llevar a cabo distintas acciones en tiempo de ejecución, acciones que podrían conseguirse también escribiendo código Java. Su propósito es tratar de reducir la cantidad de código Java necesario dentro de una página JSP, así como permitir que sea posible crear páginas JSP compuestas (idealmente) de un modo completo por etiquetas.

A menudo, en el desarrollo de una aplicación web compleja existen distintos roles entre los desarrolladores. Existen programadores que se especializan en el Back-End y suelen encargarse de implementar la lógica de negocio de la aplicación, y su persistencia. Estos programadores trabajan fundamentalmente con código Java. Existen otros programadores que se especializan en el Fron-End de la aplicación (diseñadores), y se encargan de la generación del código HTML. Para ello, suelen emplear herramientas que están preparadas para trabajar con etiquetas. Y ellos mismos están acostumbrados a trabajar con etiquetas, y no con código Java. De ahí que sea útil para ellos el poder conseguir acciones como recuperar un objeto de la sesión del usuario, o acceder a sus propiedades a través de etiquetas (la sintaxis que más conocen y están acostumbrados a usar) en vez de escribiendo código Java.

Por otro lado, incluso para un programador Java, a menudo estas etiquetas resultan más compactas y prácticas de usar que el código Java. No obstante, todas las acciones que permiten realizar las etiquetas de las acciones estándar JSP podrían conseguirse escribiendo código Java dentro de las páginas JSP. En este capítulo vamos a realizar una (breve) introducción a las acciones estándar JSP.

13.1 Sintaxis de las acciones estándar JSP

Las acciones estándar JSP siguen la siguiente sintaxis:

```
<jsp:acción [atributo= "valor"]*/>
```

donde el texto que va entre corchetes puede aparecer de cero a n veces.

13.1.1 Manipulación de Java Beans

Una de las acciones estándar más comunes es

```
<jsp:useBean id="nombre" scope="ámbito" class="clase de la instancia" />
```

esta etiqueta intenta localizar un Java Bean en el ámbito que se le indica con el atributo scope y cuyo nombre coincida con el atributo id. En caso de no localizarlo, creará una instancia de la clase que se le indica con el atributo class, y la almacenará en el ámbito especificado empleando el identificador indicado en el atributo id. Por ejemplo:

```
<jsp:useBean id="carrito" scope="session" class="MiCarrito" />
```

intenta recuperar de la sesión del usuario una instancia de la clase MiCarrito con nombre carrito. En caso de no existir dicha instancia, la página JSP creará la instancia y la almacenará en la sesión del usuario.

Existen cuatro valores posibles para el atributo scope (ámbitos diferentes):

- **page:** el ámbito será la propia página JSP. Cuando el control sea transferido a otra página JSP, u otro Servlet, esta variable desaparecerá. En caso de no indicar el valor del atributo scope en la etiqueta, éste es su valor por defecto.
- **request:** el ámbito será la petición. Cuando finalice la petición, se destruirá el objeto.
- **session:** el ámbito será la sesión del usuario. Para que una página JSP pueda almacenar y recuperar información de la sesión del usuario el valor del atributo session de la directiva page debe ser "true" (el valor por defecto): `<%@ page session="true" %>`.
- **application:** el ámbito será el contexto de la aplicación.

Si al crear un Java Bean mediante esta acción estándar queremos inicializar algunas de sus propiedades podemos hacerlo mediante la etiqueta `jsp:setProperty`:

```
<jsp:setProperty name="carrito" property="balance" value="0.0" />
```

El primer atributo de esta etiqueta es el nombre del Java Bean cuya propiedad queremos modificar; el segundo es el nombre de la propiedad y el tercero su valor. Para emplear esta etiqueta, es necesario haber usado antes la etiqueta `jsp:useBean`, ya que el atributo name de `jsp:setProperty` es el atributo id de `jsp:useBean`.

También es posible acceder a los atributos de un Java Bean empleando etiquetas:

```
<jsp:getProperty name="carrito" property="balance" />
```

donde el primer atributo es el nombre del objeto al cual queremos acceder, y el segundo es el nombre de la propiedad del objeto. Empleando esta etiqueta podremos hacer algo como:

```
<p> Balance del carrito de la compra: <jsp:getProperty name="carrito"
property="balance" /> </p>
```

De este modo no tenemos que emplear una expresión o un scriptlet JSP para incluir un valor en el HTML que se está generando dinámicamente; podemos emplear una etiqueta.

13.1.2 Accediendo a los parámetros de un formulario

La etiqueta `jsp:setProperty` también puede ser empleada para guardar en un Java Bean los parámetros de un formulario que han sido enviados a una página JSP. Para enviar un formulario a una página JSP simplemente especificaremos en el atributo `action` del formulario la URL de la página JSP a la cual queremos enviar el formulario. Una vez dentro de la página, usamos una etiqueta como:

```
<jsp:setProperty name="carrito" property="balance" />
```

observa como en esta ocasión no hemos indicado el valor para la propiedad. Cuando en una página JSP aparece una etiqueta de este estilo, el contenedor de Servlets intentará buscar un parámetro de la petición cuyo nombre coincida con el nombre especificado para la propiedad del Java Bean ("balance" en este caso). Si lo encuentra, almacenará dicho parámetro de la petición en el atributo del Java Bean. Si el atributo ya es una cadena de caracteres, no es necesario ningún tipo de conversión. Si el atributo es un tipo de dato primitivo, la conversión se lleva a cabo invocando el método `valueOf()` de la correspondiente clase wrapper (`Float.valueOf(String)`, `Integer.valueOf(String)`, `Boolean.valueOf(String)`, etc.).

Muy a menudo vamos a tener un formulario que tiene un número determinado de campos, y un objeto Java Bean que tiene una propiedad por cada uno de los campos del formulario. En la aplicación, tendremos que recuperar los campos del formulario y almacenarlos en los atributos del objeto Java. Podríamos hacer esto empleando una

etiqueta `jsp:setProperty` para cada uno de los atributos del objeto Java. O podemos simplemente emplear las siguientes sintaxis:

```
<jsp:setProperty name="firmante" property="*" />
```

El "*" en el valor de la propiedad indica que vamos a tratar de asignar todos los campos del formulario a propiedades del objeto Java. La asignación se realizará buscando propiedades del objeto Java cuyo nombre coincida con el atributo "name" de los campos del formulario. Por ejemplo, supongamos que tenemos este objeto Java Bean:

```
package tema13;

public class Usuario {

    private String nombre;
    private String email;
    private int edad;

    public String getNombre() {
        return nombre;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public int getEdad() {
        return edad;
    }
}
```

```

    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}

```

Y este formulario HTML:

```

...
<form method="post" action="/contexto/tema13/AnhadirFirma.jsp">
¿Cual es tu nombre? <input name="nombre" size="20" type="text"><br>
¿Cuál es tu e-mail? <input name="email" size="20" type="text"><br>
¿Cuál es tu edad? <input name="edad" size="4" type="text">
    <p><button>Enviar</button></p>
</form>
...

```

Si queremos construir una página JSP que reciba los tres campos del formulario y los almacene en los tres atributos de Usuario simplemente tendremos que hacer:

```

<jsp:useBean id="firmante" class="tema13.Usuario" scope="session"/>
<jsp:setProperty name="firmante" property="*/>

```

Con la primera etiqueta estaremos creando una instancia de la clase Usuario que se va a almacenar en la sesión. Su identificador será "firmante". Con la segunda etiqueta, estamos diciendo que se busque en la petición del usuario parámetros cuyo nombre coincida con los atributos del Java Bean (esto es, parámetros con nombre "nombre", "email" y "edad"). Los dos primeros parámetros serán copiados directamente a los campos del objeto Java. El tercero, se convertirá empleando el método estático Integer.valueOf(String).

Como puedes ver, con sólo tener cuidado de emplear una nomenclatura consistente en los formularios HTML y en los atributos de nuestros objetos del modelo, esta etiqueta nos va a ahorrar un montón de trabajo.

13.1.3 Redirigiendo peticiones entre páginas JSP

Para redirigir una petición a otro recurso estático o dinámico de la aplicación desde una página JSP podríamos hacer un scriptlet que lo haga del mismo modo que se hace en los Servlets (empleando un `RequestDispatcher`). Pero tenemos una forma más fácil de hacerlo: empleando la etiqueta `jsp:forward` Por ejemplo:

```
<jsp:forward page="/tema13/Libro.jsp" />
```

el atributo `page` es la URL a la cual se va a redirigir la petición. La URL puede ser relativa a la posición de la página JSP actual, o puede ser absoluta respecto al contexto de la aplicación. En el último caso, la URL debe comenzar con un `/`. La URL nunca debe de contener el nombre del protocolo, puerto, servidor o contexto. Se dará directamente desde el contexto de la aplicación.

13.2 Poniendo todo esto junto: aplicación para salvar gatitos

Vamos a hacer una aplicación web de firmas para apoyar causas justas. Vamos a tener un primer formulario donde el usuario introducirá su nombre, su e-mail y su edad:

Firmar petición


localhost:8080/contexto/tema13/firmarpeticion.html

Reproducir todos

Otros marcadores

Error de sincronización

Firma la petición para salvar al gatito:



¡Sino la firmas el gatito morirá!

¿Cual es tu nombre?

¿Cuál es tu e-mail?

¿Cuál es tu edad?

Esta información será recibida por una página JSP que almacena los campos del formulario en los atributos de la clase `tema13.Usuario`. Una vez almacenados, la página JSP mostrará esta información al usuario, y le mostrará dos enlaces, uno para realizar la firma de la petición, y otro para rectificar la información que vuelve a mostrar el formulario. Este es el código de esa página:

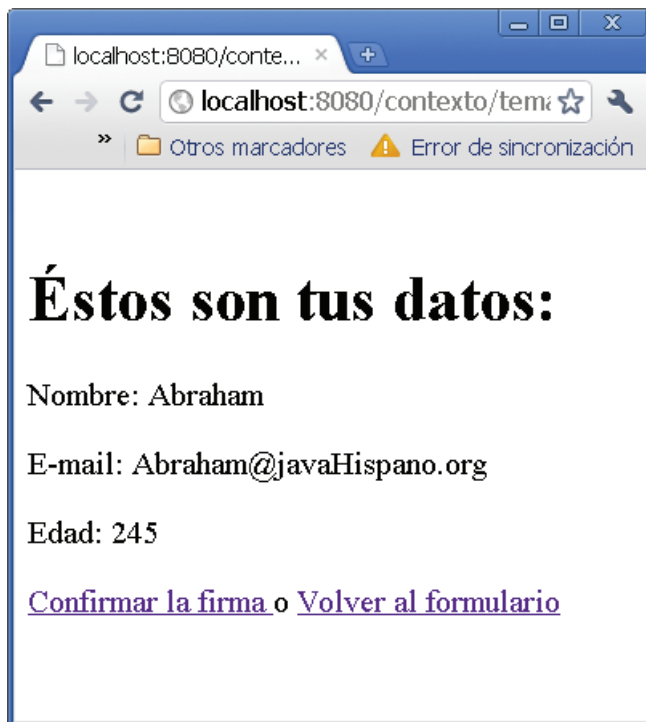
```

<jsp:useBean id="firmante" class="tema13.Usuario" scope="session"/>
<jsp:setProperty name="firmante" property="*" />
<html>
<head>
</head>
<body>
    <h1> Éstos son tus datos:</h1>
    <p>Nombre: <jsp:getProperty name="firmante" property="nombre"
/></p>
    <p>E-mail: <jsp:getProperty name="firmante" property="email"
/></p>
    <p>Edad: <jsp:getProperty name="firmante" property="edad" /></p>
<a href="/contexto/tema13/Anhadir.jsp">Confirmar la firma </a> o
<a href="/contexto/tema13/firmarpeticion.html">Volver al
formulario</a>
</body>
</html>

```

La primera etiqueta de la página JSP crea una instancia de la clase Usuario a la cual llama firmante; esta instancia será guardada en la sesión del usuario. La segunda etiqueta trata de rellenar los atributos de esta instancia empleando los parámetros del formulario. En el cuerpo de la página JSP empleamos la etiqueta jsp:getProperty para acceder a la información que se ha almacenado en la instancia de Usuario y volvérsela a mostrar al firmante de la petición.

Esta es la apariencia de la página web:



Si el usuario hace un clic en el enlace de confirmación de firma, irá a la página `Anhadir.jsp`. Esta página recuperará el objeto donde hemos almacenado la información del usuario, objeto que se encuentra en la sesión, y lo añadirá al libro de firmas. Esta es la clase que emplearemos para representar el libro de firmas:

```
package tema13;

import java.util.LinkedList;
import java.util.List;

public class LibroDeFirmas {

    private List<Usuario> firmantes = new LinkedList<Usuario>();

    public List<Usuario> getFirmantes() {
        return firmantes;
    }

    public void setFirmantes(List<Usuario> firmantes) {
        this.firmantes = firmantes;
    }
}
```

```

    public void anhadirFirmante(Usuario firmante) {
        this.firmantes.add(firmante);
    }
}

```

Este libro de firmas va a ser compartido por todos los usuarios que visiten la web. Por tanto, no podrá ser almacenado en la sesión de ningún usuario; lo almacenaremos en el contexto de la aplicación. Este es el código de la página Anhadir.jsp

```

<jsp:useBean id="firmante" class="tema13.Usuario" scope="session"/>
<jsp:useBean id="libro" class="tema13.LibroDeFirmas"
scope="application"/>
<html>
<head>
</head>
<body>
    <% libro.anhadirFirmante(firmante); %>
    <jsp:forward page="/tema13/Libro.jsp" />
</body>
</html>

```

La primera etiqueta recupera de la sesión el Java Bean donde hemos almacenado la información del firmante de esta petición. La segunda etiqueta recupera del contexto de la aplicación (o crea si no existe) una instancia de la clase LibroDeFirmas. En el código de la página JSP simplemente hay un scriptlet que añade al libro de firmas el nuevo firmante. Después, empleando la etiqueta jsp:forward, envía al firmante a Libro.jsp.

Libro.jsp muestra los nombres y los e-mail de todas las personas que han firmado la petición. También intenta recuperar de la petición del usuario el Java Bean "firmante". Si dicho Java Bean existe, le mostrará al usuario un saludo personalizado (esto es, después de la palabra "Hola" va el nombre del firmante). A continuación, muestra una lista con toda la gente que ha firmado la petición.

Si alguien accede directamente a esta página JSP, simplemente se realizará el listado de la gente que ha firmado la petición. Este es el código de Libro.jsp

```

<%@ page import= "tema13.*" %>
<%@ page import= "java.util.List" %>
<jsp:useBean id="firmante" class="tema13.Usuario" scope="session"/>
<jsp:useBean id="libro" class="tema13.LibroDeFirmas"
scope="application"/>
<html>
<head>
</head>
<body>
    <% if (firmante.getNombre() != null) { %>
        <h1> Hola <jsp:getProperty name="firmante" property="nombre"
/>!/</h1>
    <% }
    else { %>
        <h1> Hola!</h1>
    <% } %>



    <p>Esta es toda la gente que ha salvado al gatito: </p>
    <ul>
    <% for(Usuario firma : libro.getFirmantes()) {
        out.print("<li>" + firma.getNombre() + " (" + firma.getEmail() +
")</li>");
    }
    session.invalidate();
    %>
    </ul>
</body>
</html>

```

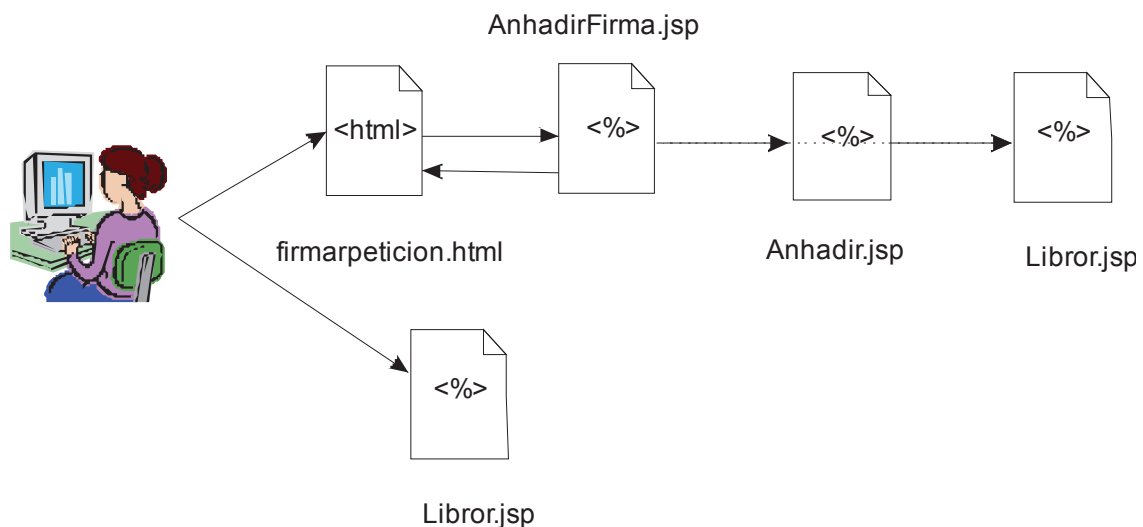
Las directivas page son necesarias en esta página JSP ya que en ella hay scriptlets que acceden a las clases que están importando dichas directivas. En esta página también se está invalidando la sesión del usuario. Si no se invalidase la sesión del usuario, cuando tu comiences a hacer pruebas con la página, y trates de enviar varias veces el formulario, la página JSP que recibe el formulario (AnhadirFirma.jsp) verá que en tu

sesión ya existe un Java Bean con nombre "firmante", y en vez de crear uno nuevo modificará los atributos de este objeto, y finalmente los añadirá al libro de firmas. Esto hará que el libro de firmas contenga realmente una única instancia de la clase Usuario, repetida dos veces (realmente, una vez por cada envío del formulario). Los atributos de esa instancia serán los atributos del último envío del formulario. Invalidar la sesión solución a este problema, ya que hará que la próxima vez sea necesario crear una nueva sesión y, por tanto, un nuevo objeto Usuario.

Esta es la apariencia de la página del libro de firmas:



En la siguiente figura representamos los posibles flujos de navegación de la aplicación web de recogida de firmas. El flujo representado en la parte superior se corresponde con un usuario que firma la aplicación, visitando un primer lugar el formulario de HTML estático, y después dando su consentimiento en la página JSP. El segundo flujo de navegación se corresponde con usuario que simplemente visite libro de firmas para ver quién ha firmado la petición.



13.2.1 Problemas con esta aplicación

Esta aplicación, que intencionadamente ha sido hecha completamente mediante páginas JSP, estaría mejor diseñada empleando un Servlet que sustituyese a la página `Anhadir.jsp`. Como habrás podido ver, esta página JSP no se emplea para generar absolutamente ningún HTML. Se emplea para ejecutar código Java y para redirigir la petición a otra página. Esto podríamos haberlo conseguido de un modo simple con un Servlet.

Una ventaja adicional de haber usado un Servlet es que es posible tener Servlets desplegados en el servidor que no son accesibles al usuario a través de una URL. Para ello basta con no definir ningún mapeo para el Servlet. ¿Y por qué esto resulta interesante?. Pues porque en nuestra aplicación del libro de firmas el usuario podría ir directamente a `Anhadir.jsp` empleando la URL:

Lo cual tendría como efecto añadir un firmante de la petición cuyo nombre fuese "null" y su dirección de correo "null". En general, no es buena idea dejar publicados al usuario recursos a los cuales él no debería de acceder directamente a través de una URL. Esto puede conseguirse de un modo simple con los Servlets, o combinando Servlets con páginas JSP.

13.3 Creando etiquetas a medida

Es posible crear librerías de acciones (también llamadas librerías de etiquetas) con una sintaxis similar a las acciones estándar JSP, etiquetas que después podremos emplear en nuestras páginas JSP evitando el emplear scriptlets. De este modo, podemos conseguir una separación completa entre el código fuente Java, que estará completamente en archivos Java, y la capa de presentación de nuestra aplicación, que estará formada por páginas JSP, las cuales incluirán etiquetas estándar y etiquetas que hemos implementado nosotros para cubrir nuestras propias necesidades.

Para construir una una etiqueta debemos crear una clase Java que extienda la interfaz **javax.servlet.jsp.tagext.Tag** o, en su defecto, que extienda la clase **TagSupport**, la cual proporciona implementaciones por defecto para todos los métodos de Tag. La interfaz Tag define varias variables (que como todas las variables definidas dentro de una interfaz son static y final) y varios métodos. En esta sección vamos a realizar una introducción muy básica a las librerías de etiquetas, y sólo vamos a emplear uno de los métodos definidos en la interfaz Tag. Con el contenido de esta sección será suficiente para que el lector cree etiquetas básicas. Pero si va a realizar una cantidad significativa de programación con JSP, debería estudiar por su cuenta más detalle las librerías de etiquetas, así como el ciclo de vida de las etiquetas (algo que intencionalmente se está omitiendo completamente en esta sección por no ser necesario para crear etiquetas sencillas).

El método de la interfaz Tag que a nosotros nos interesa es

```
public int doStartTag() throws JspException
```

Éste es el método será invocado para crear el contenido HTML que deseamos insertar en nuestra página JSP. Dentro de este método tenemos acceso a un objeto de la clase `javax.servlet.jsp.PageContext` con nombre **pageContext**. Dicho objeto tiene métodos que nos permiten acceder al flujo de salida asociado con la respuesta que se va a generar para el usuario (`getOut()`) a la sesión asociada con esta petición, al contexto de la aplicación... y en general a cualquier información que podamos necesitar para generar el cuerpo de la respuesta que se va a enviar al usuario.

El entero que devuelve este método puede tomar los valores representados por las constantes de la interfaz Tag **SKIP_BODY** o **EVAL_BODY_INCLUDE**. Como su nombre indica, el primero quiere decir que se va a omitir el cuerpo de la etiqueta, mientras que el segundo quiere decir que se va a incluir el cuerpo de la etiqueta. Las etiquetas que nosotros desarrollemos para nuestras librerías, al igual que cualquier etiqueta HTML, puede tener un cuerpo. A partir del valor de retorno de `doStartTag()`, el contenedor decide si se incluye o no el cuerpo de la etiqueta en la respuesta.

Aparte de sobrescribir el método `doStartTag()`, en nuestra clase podemos crear atributos que se van a corresponder con atributos de la etiqueta. Para ello, simplemente creamos un atributo normal y corriente Java, y creamos un método setter para dicho atributo (no hace falta crear el getter). Con sólo hacer eso en nuestro código Java, podremos desde las páginas JSP pasarle atributos HTML a nuestra etiqueta, empleando la misma sintaxis que se emplea en HTML para especificar cualquier atributo. Veamos un ejemplo de clase Java que implementa una etiqueta:

```
package tema13;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
```

```

public class HolaMundoTag extends TagSupport {

    private String texto;
    private int repeticiones = 2;

    @Override
    public int doStartTag() throws JspException {
        try {
            JspWriter out = pageContext.getOut();
            for (int i = 0; i < repeticiones; i++) {
                out.print(texto + "<br/>");
            }
        } catch (IOException e) {
            throw new JspException("Error: IOException" +
e.getMessage());
        }
        if (Math.random() * 10 > 5) {
            return SKIP_BODY;
        }
        return EVAL_BODY_INCLUDE;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }

    public void setRepeticiones(int repeticiones) {
        this.repeticiones = repeticiones;
    }
}

```

HolaMundoTag define dos atributos, uno es un texto que se va a mostrar al usuario. El otro, que toma valor por defecto 2, es el número de veces que se va a mostrar dicho texto. Observa como hay métodos set para dichos atributos. Por tanto, nuestra etiqueta va a tener dos atributos, uno es un texto que queremos mostrar al usuario, y el otro es un número entero indicando cuántas veces lo queremos mostrar.

Como puedes observar, el método `getOut()` devuelve un objeto de tipo **JspWriter**, que es realmente un wrapper en torno al flujo de salida asociado con la respuesta que se va a generar para el usuario.

Al final de todo del cuerpo del método, se genera un número aleatorio entre 0 y 10; si dicho número es mayor que 5 se va a ignorar el cuerpo de la etiqueta, mientras que si dicho número es menor que 5 se va a incluir el cuerpo de la etiqueta en la respuesta que se envía al usuario. Obviamente, este comportamiento no determinista de la etiqueta está implementado aquí sólo con fines académicos, para poder demostrar ambos comportamientos.

Nuestra etiqueta está lista. Pero necesitamos crear un archivo de configuración para que el contenedor sepa "cómo funciona". Estos archivos son archivos XML con una extensión `.tld` que se sitúan en el directorio `WEB-INF`, habitualmente en un subdirectorio de `WEB-INF` con nombre `tlds`. En dicho subdirectorio es donde podrás encontrar el archivo de configuración correspondiente con `HolaMundoTag`, cuyo nombre es `MiTagLib.tld`, y cuyo contenido es:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd">

    <tlib-version>1.0</tlib-version>
    <short-name>newtag_library</short-name>
    <uri>MiLibreriaDeTags</uri>
    <tag>
        <name>holamundo</name>
        <tag-class>tema13.HolaMundoTag</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
            <name>texto</name>
            <required>true</required>
        </attribute>
        <attribute>
            <name>repeticiones</name>
```

```
        <required>false</required>
    </attribute>
</tag>
</taglib>
```

Las etiquetas **<tlib-version>** y **<short-name>** están indicando la versión del nombre de la librería de etiquetas. Dentro de nuestra librería de etiquetas (**<taglib>**) podemos tener definidas varias etiquetas, empleando para ello múltiples repeticiones de la etiqueta **<tag>**. En nuestro caso, sólo tenemos una. **<name>** indica el nombre de la etiqueta; éste será el nombre que emplearemos en el archivo JSP para referirnos a dicha etiqueta. **<tag-class>** indica la clase que implementa dicha etiqueta; debe ser un nombre completo incluyendo el paquete.

La etiqueta **<body-content>** puede tomar tres valores:

- **empty**: indica que la etiqueta no va a tener contenido. Se trata de una etiqueta compacta.
- **scriptless**: indica que la etiqueta va a tener contenido, y dicho contenido van a ser etiquetas: etiquetas estándar, etiquetas nuestras , o simplemente etiquetas HTML. Pero el contenido no puede ser un scriptlet.
- **tagdependent**: se emplea para cualquier otro tipo de contenido.

En nuestro caso, hemos indicado que la etiqueta puede contener otras etiquetas dentro. A continuación tenemos la declaración de los atributos admisibles para la etiqueta. Emplearemos una etiqueta **<attribute>** por cada atributo. Para cada atributo, debemos indicar su nombre y si dicho atributo es obligatorio o no (**<required>**). Si marcamos un atributo como obligatorio mediante **<required>true</required>**, obligatoriamente tendremos que indicar ese atributo cuando empleemos la etiqueta. Si lo marcamos como opcional, **<required>false</required>**, podremos especificarlo o no.

Ahora ya tenemos creada nuestra etiqueta y podemos emplearla desde las páginas JSP. Para ello, tendremos que emplear una directiva de página JSP indicando que vamos a emplear una librería de etiquetas; debemos especificar cuál es la URI de dicha librería

de etiquetas, así como el prefijo que emplearemos en el documento para referirnos al espacio de nombre correspondiente con dicha librería:

```
<%@ taglib uri="MiLibreriaDeTags" prefix="mistags" %>
```

El atributo **uri** de la directiva taglib tiene que corresponderse obligatoriamente con el valor indicado en la etiqueta <uri> del archivo tld correspondiente con la librería. El valor del atributo **prefix** puede ser lo que nosotros queramos, y será el espacio de nombre donde se encontrarán todas las etiquetas definidas en la uri. En nuestro caso, la única etiqueta que está definida en MiLibreriaDeTags tendrá como nombre completo mistags:holamundo, es decir, el nombre del espacio de nombres indicado en la directiva, y el nombre que hemos indicado en el archivo tld para la etiqueta.

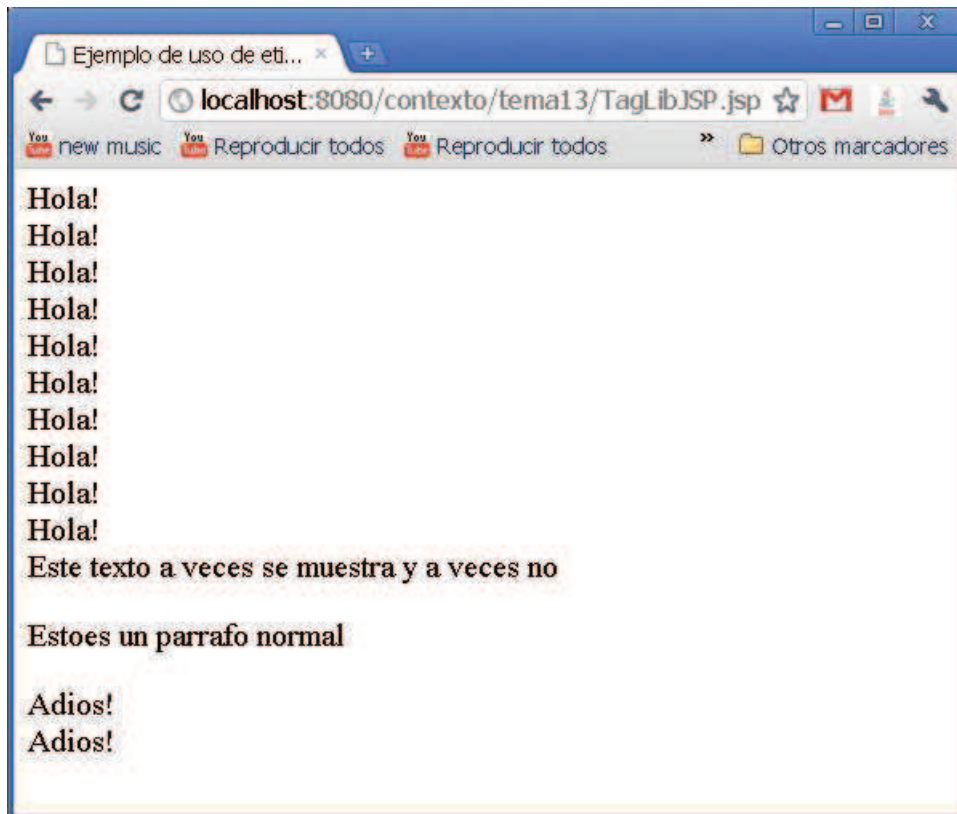
A continuación mostramos un ejemplo de páginas JSP donde se emplea nuestra etiqueta:

```
<%@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">

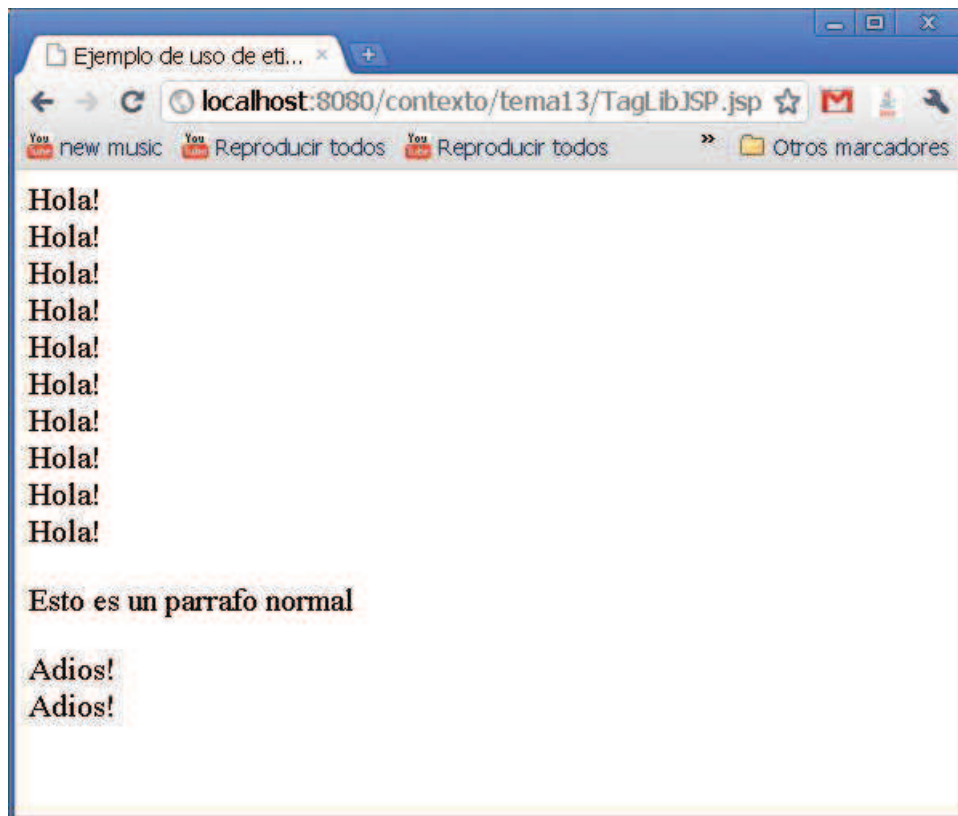
<%@ taglib uri="MiLibreriaDeTags" prefix="mistags" %>
<html>
    <head>
        <title>Ejemplo de uso de etiquetas propias</title>
    </head>
    <body>
        <mistags:holamundo texto= "Hola!" repeticiones= "10">Este
texto a veces se muestra y a veces no</mistags:holamundo>
        <p>Esto es un parrafo normal</p>
        <mistags:holamundo texto= "Adios!"/>
    </body>
</html>
```


Observa como la etiqueta se ha empleado dos veces. La primera vez, la etiqueta tiene como atributo de texto "Hola!", que deberá repetirse 10 veces. Además, la etiqueta tiene cuerpo. Ese cuerpo a veces se mostrará y a veces no debido a la implementación de nuestro método doStartTag(). Después hemos escrito un párrafo normal, y otra vez volvemos a usar la etiqueta, esta vez de modo compacto (sin tener cuerpo). Esta segunda vez tampoco estamos indicando el número de repeticiones que debe emplearse para el texto. El atributo repeticiones era opcional, y en nuestra clase toma el valor 2; por tanto el texto "Adios!" se mostrará dos veces.

Este es el resultado de visualizar TagLibJSP.jsp cuando se muestra el texto de la primera etiqueta:



Y este es el resultado cuando dicho texto no aparece:



14 Poniéndolo todo junto en una aplicación: BasicCRUDWebApp

Todos los ejemplos que hemos visto hasta este momento han sido extremadamente sencillos, estando formados a lo sumo por dos o tres Servlets o páginas JSP bastante básicas. En este capítulo también vamos a ver un ejemplo sencillo, pero no tanto como los anteriores ;)

El ejemplo que vamos a ver es una aplicación web que implementa las operaciones CRUD (Create, Read, Update and Delete) para un modelo muy básico; la llamaremos BasicCRUDWebApp. Esta aplicación sigue un esquema típico Modelo-Vista-Controlador (MVC). Comenzamos explicando brevemente en qué consiste el patrón MVC.

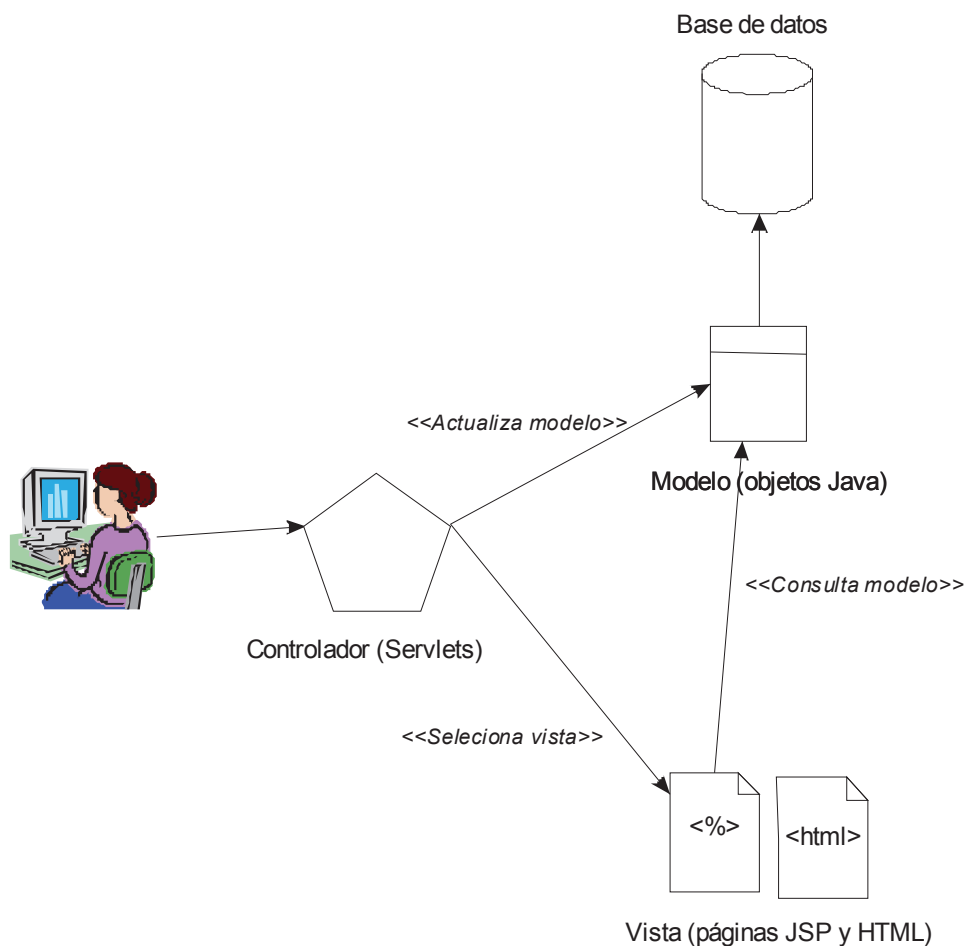
14.1 Patrón MVC

El patrón de diseño MVC nació en un lenguaje de programación llamado Smalltalk como una solución para construir interfaces gráficas de usuario. Su propósito es separar tres responsabilidades dentro de una aplicación. Éstas tres responsabilidades son:

- **Modelo:** se encarga de representar los objetos del dominio y contener la lógica de negocio.
- **Vista:** se encarga de presentar los datos al usuario.
- **Controlador:** acepta las acciones del usuario, las convierte en manipulaciones sobre el modelo y selecciona vistas adecuadas para representar la información relacionada con las acciones del usuario.

En el desarrollo de aplicaciones web con Java, el modelo idealmente está construido por POJOS (especialmente en versiones antiguas de la plataforma Java EE esto no siempre era posible). La vista, está formada tanto por páginas HTML estáticas como por páginas

JSP. Y el controlador está formado por Servlets. La siguiente imagen representa el funcionamiento típico de este patrón en una aplicación web Java. El controlador es el que recibe las peticiones del usuario, y en base a ellas es posible que realice alguna manipulación del modelo. A continuación, selecciona la vista adecuada para responder al usuario (una página JSP o HTML) y le pasa la información que debe representar:

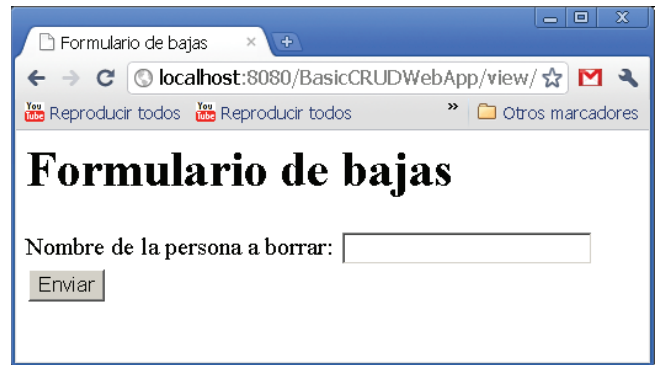


14.2 BasicCRUDWebApp

BasicCRUDWebApp es una aplicación sencilla que permite dar de alta personas ("amigos" del usuario de la aplicación web), recuperar la información de una persona, actualizar la información de una persona y borrar personas. Consta de una primera página principal desde la cual se puede acceder a todas estas acciones:



Desde esta primera página HTML estática accedemos a otras cuatro páginas HTML estáticas que contienen un formulario en el cual se le pide al usuario la información necesaria para llevar a cabo cada una de las operaciones CRUD:



En esta aplicación, el nombre de la persona actuará a modo de identificador único de cada uno de los registros que se den de alta. Veamos ahora la estructura interna de la aplicación.

14.3 El modelo y la persistencia (patrón DAO)

En nuestro caso, el modelo propiamente dicho está formado por una única clase, un Java Bean muy simple y sin ninguna lógica de negocio:

```
package model;
import java.io.Serializable;

public class Friend implements Serializable{

    private String name = null;
    private String phone = null;
    private String address = null;
    private String comments = null;
    private int age = 0;
    private int friendshipLevel = 0;

    //getters y setters
}
```

También tenemos un conjunto de clases que se encargan de persistir la información de la aplicación. La persistencia sigue un patrón DAO. En este patrón tenemos una interfaz que define las operaciones de persistencia; en nuestro caso es esta interfaz:

```
package persistence;

import model.Friend;

public interface FriendDAO {
    public boolean createFriend(Friend friend);

    public boolean deleteFriend(String name);

    public Friend readFriend(String name);

    public boolean updateFriend(String name, Friend friend);

    public boolean setUp(String url, String driver, String user,
String password);

    public boolean disconnect();
}
```

Sus métodos dan soporte a las cuatro operaciones CRUD sobre el modelo. Además, tenemos un método para configurar el mecanismo de persistencia, y otro que debe ser invocado cuando se detenga la aplicación para liberar cualquier posible recurso que el mecanismo de persistencia esté empleando.

Todas las operaciones que la aplicación realice relativas a la persistencia las hará sobre esta interfaz. De este modo, si es necesario soportar una nueva base de datos que emplee un dialecto especial de SQL, o un mecanismo diferente de una base de datos SQL para persistir la información, todo lo que tendremos que hacer será implementar esta interfaz y dar soporte a las operaciones.

BasicCRUDWebApp proporciona tres implementaciones diferentes de la interfaz: FriendDAOJDBCImplementation, FriendDAOPoolImplementation y FriendDAOFileImplementation.

La implementación FriendDAOJDBCImplementation emplea JDBC directamente contra la base de datos, sin tomar ventaja del pool de conexiones. El método setUp() abre una conexión contra la base de datos, conexión que se libera en el método disconnect(). La clase necesita usar sincronización sobre el objeto statement que se emplea para realizar todas las operaciones CRUD de la aplicación web (mismo para todos los usuarios que estén empleando la aplicación de modo concurrente). Se trata de una implementación pobre y poco escalable ya que sólo podemos realizar una operación contra la base de datos de modo simultáneo. A continuación mostramos uno de los métodos de FriendDAOJDBCImplementation a modo de ejemplo, junto con el método que inicializa la conexión con la base de datos:

```
private Connection conexion = null;
private static FriendDAOJDBCImplementation persistenceManager =
null;
private static final Logger logger =
Logger.getLogger(FriendDAOJDBCImplementation.class.getName());
...

@Override
public boolean setUp(String url, String driver, String user,
String password) {
    try {
        Class.forName(driver);
        conexion = DriverManager.getConnection(url, user,
password);
    } catch (ClassNotFoundException ex) {
        logger.log(Level.SEVERE, "No se encontro el driver para la
base de datos", ex);
        return false;
    } catch (SQLException ex) {
        logger.log(Level.SEVERE, "No se pudo establecer la
conexion con la base de datos", ex);
```



```

        return false;
    }
    return true;
}

@Override
public boolean createFriend(Friend friend) {
    String query = "insert into\FRIENDS\" values(?,?,?,?,?,?)";
    PreparedStatement statement;
    try {
        synchronized (lockOfTheConexion) {
            statement = conexion.prepareStatement(query);
        }
        statement.setString(1, friend.getName());
        statement.setString(2, friend.getAddress());
        statement.setString(3, friend.getPhone());
        statement.setString(4, friend.getComments());
        statement.setInt(5, friend.getAge());
        statement.setInt(6, friend.getFriendshipLevel());
        statement.execute();
        return true;
    } catch (SQLException ex) {
        logger.log(Level.SEVERE, "Error al crear un amigo", ex);
        return false;
    }
}
}

```

Observa como en esta ocasión, a diferencia de los ejemplos del tema 10, hemos empleado un PreparedStatement. Los PreparedStatement no sólo suelen ser más eficientes que los Statement cuando una misma consulta va a realizarse varias veces, sino que además son bastante menos vulnerables a inyecciones SQL (http://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL).

Como su nombre indica, FriendDAOPoolImplementation emplea un pool de conexiones, que debe estar correctamente declarado en el descriptor de despliegue de la aplicación:

```

<!--Configuracion del pool de conexiones-->
<resource-ref>
    <res-ref-name>jdbc/myfriends</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>

```

Y en el fichero sun-web.xml que requiere Glassfish:

```

<resource-ref>
    <res-ref-name>jdbc/myfriends</res-ref-name>
    <jndi-name>jdbc/myfriends</jndi-name>
</resource-ref>

```

Este es el contenido de los descriptores de despliegue de la aplicación que va con el tutorial. Para que funcione en tu equipo, deberás crear el pool de conexiones, y o bien darle el nombre JNDI que se emplea en estos descriptores, o modificar estos descriptores para que sean coherentes con el nombre de tu pool.

FriendDAOPoolImplementation obtiene su conexión del pool del servidor, pero como no me gusta repetir código fuente, emplea una instancia de FriendDAOJDBCImplementation para ejecutar las operaciones contra la base de datos. Cada método que realiza una de las operaciones CRUD crea una instancia de FriendDAOJDBCImplementation y lo configura adecuadamente introduciendo dentro de él a través de un método setter un objeto de tipo Statement. A continuación, ejecuta la operación contra la base de datos empleando el objeto tipo FriendDAOJDBCImplementation, y finalmente libera la conexión de la base de datos. A modo de ejemplo, pongo a continuación uno de sus métodos, así como los métodos auxiliares en los que se apoya:

```

@Override
public boolean createFriend(Friend friend) {
    FriendDAO jdbcFriendDAO = prepareForExecutingQuery();
    if (jdbcFriendDAO == null) {
        return false;
    }
    boolean isExecutedOk = jdbcFriendDAO.createFriend(friend);
    releaseQueryResources(jdbcFriendDAO);
    return isExecutedOk;
}

private FriendDAO prepareForExecutingQuery() {
    FriendDAOJDBCImplementation jdbcPersistenceManager = new
FriendDAOJDBCImplementation ();
    Connection connection;
    try {
        connection = pool.getConnection();
    } catch (SQLException ex) {
        logger.log(Level.SEVERE, "No se pudo abrir la conexion
contra la base de datos", ex);
        return null;
    }
    jdbcPersistenceManager.setConnection(connection);
    return jdbcPersistenceManager;
}

private void releaseQueryResources( FriendDAO friendDAO) {
    friendDAO.disconnect();
}

```

De este modo, la lógica relacionada con la ejecución de las sentencias SQL se encuentra en un único sitio (en la clase FriendDAOJDBCImplementation).

El tercer mecanismo de persistencia está implementado en FriendDAOFileImplementation, y como su nombre indica emplea un archivo para persistir los datos. Los datos cuando la aplicación se está ejecutando se mantienen en memoria en un mapa. Cuando la aplicación se detiene, se guardan a un archivo en el disco duro (esto se hace en el método diconnect). Cuando la aplicación arranca, se

comprueba si existe dicho archivo, y si existe se cargan los datos del archivo (esto se hace en el método setUp).

FriendDAOFileImplementation es una prueba conceptual de las bondades del patrón DAO; seguramente el lector hasta ahora había asumido que la aplicación web siempre estaba persistiendo la información en una base de datos. Pues no, la aplicación web puede funcionar perfectamente sin una base de datos (lo cual es una ventaja para realizar pruebas rápidas sobre ella: no hay que instalar ni configurar una base de datos). A modo de ejemplo, a continuación mostramos uno de los métodos de FriendDAOFileImplementation:

```
@Override
public synchronized boolean createFriend(Friend friend) {
    if (friendMap.containsKey(friend.getName())) {
        return false;
    } else {
        friendMap.put(friend.getName(), friend);
        return true;
    }
}
```

Recapitulando, tenemos una interfaz (FriendDAO) que define todas las operaciones que debe soportar el mecanismo de persistencia. Y tenemos tres implementaciones diferentes de la interfaz: FriendDAOJDBCImplementation, FriendDAOPoolImplementation y FriendDAOFileImplementation. La selección del mecanismo concreto de persistencia que se va a emplear se realiza a través de una factoría:

```
package persistence;
public class FriendPersistFactory {

    public static FriendDAO getFriendDAO(String persistenceMechanism) {
        if (persistenceMechanism.equals("file")) {
            return
FriendDAOFileImplementation.getJDBCPersistenceManager();
        }
    }
}
```

```

        }
        else if (persistenceMechanism.equals("JDBC")){
            return
FriendDAOJDBCImplementation.getJDBCPersistenceManager();
        }
        else if (persistenceMechanism.equals("pool")){
            return
FriendDAOPoolImplementation.getFriendDAOPoolImplementation();
        }
        else{
            return null;
        }
    }
}

```

Invocando al método estático de esta factoría obtendremos un objeto tipo FriendDAO que nos permitirá realizar todas las operaciones relativas a la persistencia. El código que invoca a ese método es un Listener de contexto cuyo nombre es StartUpListener. Cuando arranca la aplicación, este Listener recupera un conjunto de parámetros de configuración del contexto de la aplicación que se encuentran en el descriptor de despliegue. Estos parámetros son la URL de la base de datos (en caso de estar usando persistencia a archivo, este parámetro será el nombre del archivo), el usuario y el password de la base de datos, el driver de la base de datos (que sólo es necesario cuando se está usando la implementación que emplea directamente JDBC) y un parámetro (persistenceMechanism) que indica el mecanismo de persistencia que se va a emplear. Este es el parámetro que se le pasará a la factoría para crear el mecanismo de persistencia.

```

package controller;

import java.util.logging.Level;
import java.util.logging.Logger;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletContextEvent;
import persistence.FriendDAO;

```

```

import persistence.FriendPersistFactory;

public class StartUpListener implements ServletContextListener {

    private FriendDAO friendDAO;

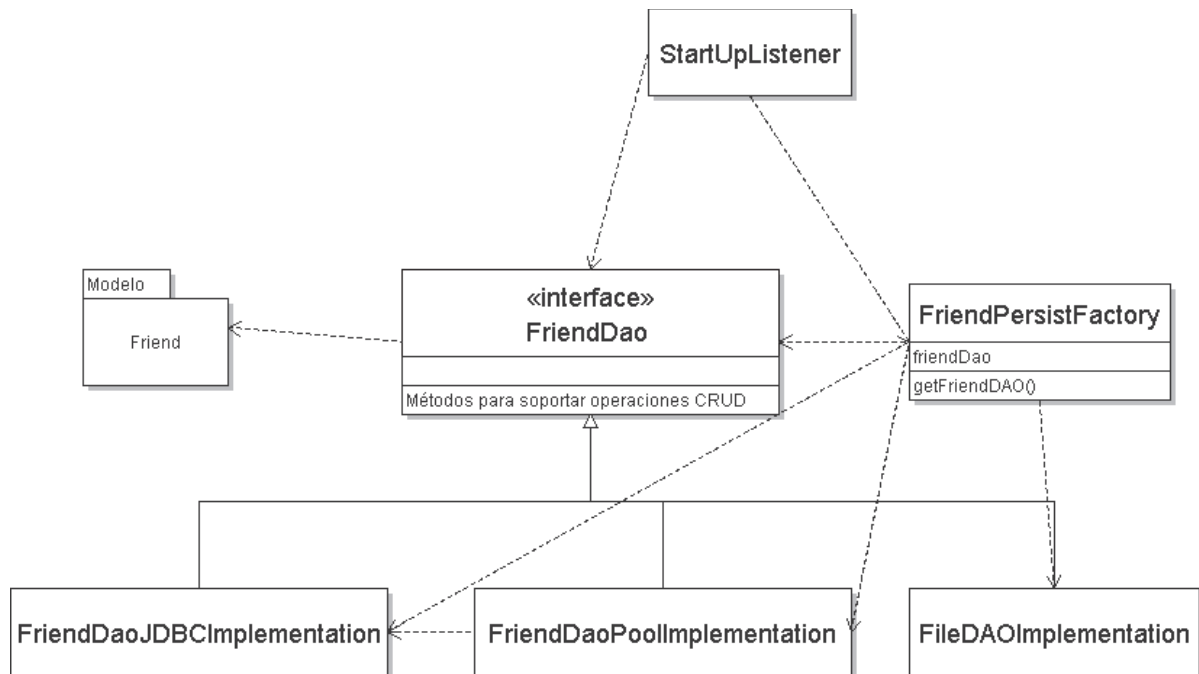
    @Override
    public void contextInitialized(ServletContextEvent evt) {
        String url, driver, user, password, persistenceMechanism;
        ServletContext context = evt.getServletContext();
        url = context.getInitParameter("databaseURL");
        driver = context.getInitParameter("databaseDriver");
        user = context.getInitParameter("databaseUser");
        password = context.getInitParameter("databasePassword");
        persistenceMechanism =
context.getInitParameter("persistenceMechanism");
        friendDAO =
FriendPersistFactory.getFriendDAO(persistenceMechanism);
        boolean ok = friendDAO.setUp(url, driver, user, password);
        if (!ok) {
            context.setAttribute("persistenceMechanism", "none");
        }
    }

    @Override
    public void contextDestroyed(ServletContextEvent evt) {
        boolean ok = friendDAO.disconnect();
        if (!ok) {
            Logger.getLogger(StartUpListener.class.getName()).log(Level.SEVERE,
                "No se encontro el driver para la base de datos");
        }
    }
}

```

A continuación se muestra un diagrama UML donde se describe el funcionamiento del patrón DAO en la aplicación. Observa como el modelo no tiene dependencias con nadie. La factoría es la única clase que necesita conocer exactamente qué distintos tipos de implementaciones existen para el mecanismo de persistencia. StartUpListener, y el

resto de la aplicación, sólo necesitan conocer la interfaz que implementan los distintos mecanismos de persistencia, y la factoría para obtener la instancia que deben utilizar de dicha interfaz.



Lo bonito del patrón DAO es que abstrae completamente a la aplicación del mecanismo de persistencia que está empleando. Sólo la factoría tiene una cantidad mínima de lógica para elegir el mecanismo concreto de persistencia que se debe emplear. El resto del código de la aplicación, no tienen ni idea de qué mecanismo se está empleando, ni necesita saberlo. Y lo único que tenemos que hacer para cambiar de un mecanismo de persistencia a otro es modificar el descriptor de despliegue de la aplicación:

```

<!--Este parmetroConfigura el tipo de persistencia; sus posibles
valores son
    pool: usa el pool de persistencia (que tiene que estar
configurado el servidor de aplicaciones)
    JDBC: usa JDBC directamente; es necesario especificar la URL
de la base de datos, el driver el usuario y el password
    file: usa un fichero; en este caso databaseURL es el nombre
del fichero-->

```

```
<context-param>
    <param-name>persistenceMechanism</param-name>
    <param-value>file</param-value>
</context-param>
```

Otra ventaja que tiene el patrón DAO es que, aunque la aplicación no vaya a emplear varios mecanismos de persistencia, su uso da como resultado un código bien organizado y estructurado, y además un código que sigue una organización estandarizada que será fácil de comprender por otros desarrolladores que lo lean.

La aplicación que va con el curso se distribuye empleando como mecanismo de persistencia un archivo. De este modo el lector podrá ejecutarla sin configurar ni instalar ninguna base de datos.

14.4 El controlador

Todas las peticiones que llegan a la aplicación llegan a través de un único Servlet con nombre FrontController. El nombre viene de otro patrón de diseño: el Front Controller. De un modo muy simple, este patrón consiste en hacer que todas las peticiones de la aplicación pasen por un controlador frontal. Esto tiene la ventaja de que es muy simple introducir en este controlador frontal requerimientos de la aplicación transversales a la lógica de negocio, como autorización, autenticación, logging... así como cambiar el flujo de navegación de la aplicación.

Nuestro Servlet controlador frontal, y todos los Servlets de la aplicación, heredan de un Servlet con nombre MyCoolServlet (no sabía qué llamarle... y como su propósito es ahorrar trabajo le puse ese nombre). Este Servlet define un conjunto de cadenas de caracteres que se inicializan a partir de parámetros de configuración del contexto de la aplicación. Esas cadenas de caracteres definen tanto las vistas como los controladores secundarios de la aplicación (otros Servlets). Todas las vistas, y todos los Servlets de la aplicación se referencian a partir de esas cadenas de caracteres, que se configuran en el descriptor de despliegue. Como veremos más adelante, con sólo cambiar un parámetro de configuración en el descriptor de despliegue podemos cambiar las vistas o los controladores.

MyCoolServlet también definen dos métodos de utilidad para redirigir una petición a un Servlet (gotoNamedResource) o a una página HTML o JSP (gotoURL). Estos métodos de utilidad, que van a heredar sus hijos, permiten redirigir la petición en una única sentencia, sin necesidad de crear el RequestDispatcher adecuado para cada caso.

MyCoolServlet define un método abstracto (processRequest) y sobre escribe los métodos doGet y doPost de tal modo que redirigen las peticiones que les llegan al método abstracto. De este modo evitaremos sobrescribir estos dos métodos en todos los Servlets de la aplicación (como hemos hecho a lo largo de todo este tutorial en cada uno de los Servlets... copiar y pegar es malo pero hasta ahora quería que cada ejemplo fuese autocontenido). Este es el código de MyCoolServlet:

```
public abstract class MyCoolServlet extends HttpServlet {
    protected String errorForm = null;
    protected String displayForm = null;
    protected String successForm = null;
    protected String createServlet = null;
    protected String deleteServlet = null;
    protected String updateServlet = null;
    protected String readServlet = null;
    protected String persistenceMechanism = null;

    protected abstract void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException;

    protected void gotoNamedResource(String address,
        HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
        RequestDispatcher dispatcher =
        getServletContext().getNamedDispatcher(address);
        dispatcher.forward(request, response);
    }

    protected void gotoURL(String address, HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
        RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(address);
```

```

        dispatcher.forward(request, response);
    }

    @Override
    public void init() {
        ServletConfig config = getServletConfig();
        ServletContext context = config.getServletContext();
        displayForm = context.getInitParameter("displayForm");
        errorForm = context.getInitParameter("errorForm");
        successForm = context.getInitParameter("successForm");
        createServlet = context.getInitParameter("createServlet");
        deleteServlet = context.getInitParameter("deleteServlet");
        readServlet = context.getInitParameter("readServlet");
        updateServlet = context.getInitParameter("updateServlet");
        persistenceMechanism =
context.getInitParameter("persistenceMechanism");
    }

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

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

```

El Servlet FrontController va a ser el que reciba el envío de todos los formularios de la aplicación. FrontController sobrescribe el método abstracto de su padre, y se limita a extraer un parámetro oculto que está presente en todos los formularios: "form". Este campo oculto indica el recurso al cual se quiere dirigir el formulario. En caso de que el formulario no tenga dicho campo oculto, en caso de que no exista mecanismo de persistencia, o en caso de que este campo oculto no tome uno de los valores preestablecidos de antemano, mostraremos una página de error. Los valores que toma el

campo oculto de los formularios se corresponden con cada una de las operaciones CRUD y con la ocurrencia de un error inesperado. Éste es el código del FrontController:

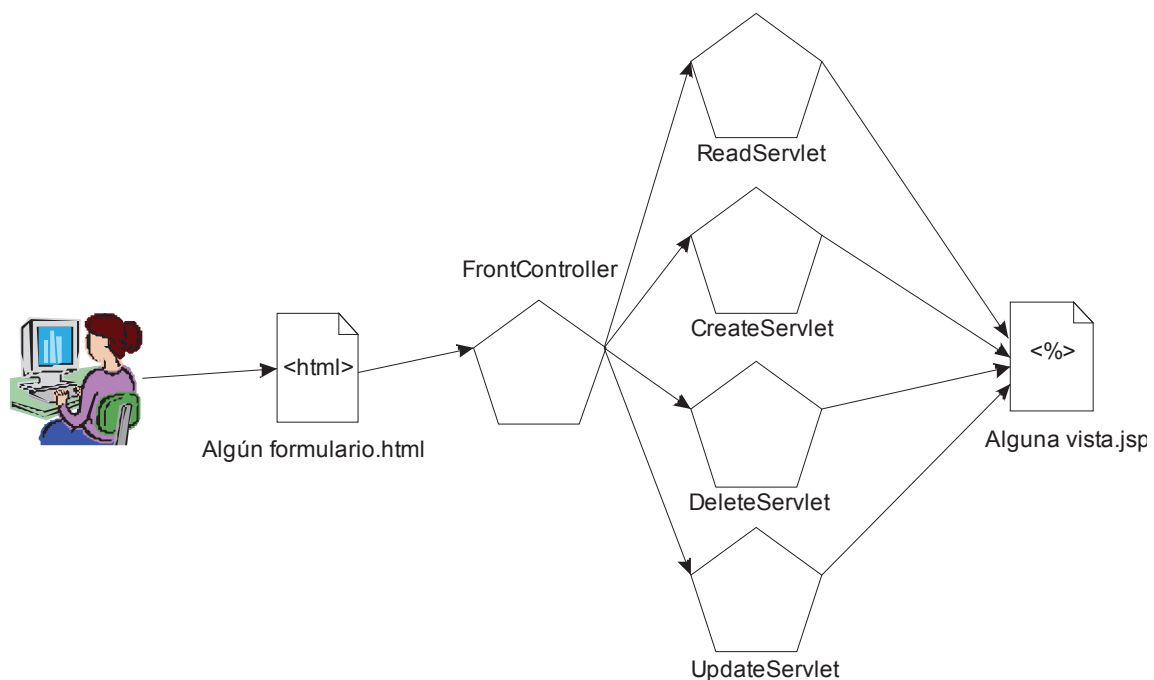
```
public class FrontController extends MyCoolServlet {

    @Override
    protected void processRequest(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
        String form = request.getParameter("form");
        if (form == null || (persistenceMechanism != null
            && persistenceMechanism.equals("none"))) {
            gotoURL(errorForm, request, response);
        } else if (form.equals("errorForm")) {
            gotoNamedResource(errorForm, request, response);
        } else if (form.equals("createServlet")) {
            gotoNamedResource(createServlet, request, response);
        } else if (form.equals("updateServlet")) {
            gotoNamedResource(updateServlet, request, response);
        } else if (form.equals("deleteServlet")) {
            gotoNamedResource(deleteServlet, request, response);
        } else if (form.equals("readServlet")) {
            gotoNamedResource(readServlet, request, response);
        } else {
            gotoURL(errorForm, request, response);
        }
    }
}
```

Además de este Servlet, tendremos otro Servlet por cada una de las operaciones CRUD. Cada uno de estos Servlets obtiene a partir de la factoría la implementación de FriendDAO que debe emplear para llevar a cabo su tarea; realiza la tarea que se le ha solicitado empleando dicha implementación (lo cual a menudo requiere extraer parámetros de un formulario); almacena en la petición información que empleará más adelante la vista para informar al usuario de lo que ha ocurrido, y redirige la petición a

la vista adecuada (suele haber dos, una correspondiente con el éxito de la operación y otra con un error inesperado).

La siguiente figura ilustra el flujo de navegación típico de la aplicación: desde un formulario HTML estático se llega al controlador frontal, quien recibe absolutamente todas las peticiones de formularios de la aplicación. El controlador decide qué Servlet debe gestionar la petición, y redirige la petición al Servlet adecuado. El Servlet lleva a cabo la acción solicitada por el usuario, y selecciona la vista adecuada (normalmente una página JSP). Los Servlets, como suele ser habitual, se dibujan mediante pentágonos:



A continuación, a modo de ejemplo mostramos el código de CreateServlet. Se han excluido un conjunto de métodos destinados a validar los campos del objeto que se está creando:

```
package controller;  
  
import java.io.*;  
import javax.servlet.*;
```

```

import javax.servlet.http.*;
import persistence.FriendDAO;
import persistence.FriendPersistFactory;
import model.Friend;

public class CreateServlet extends MyCoolServlet {

    @Override
    protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        FriendDAO persistenceManager =
        FriendPersistFactory.getFriendDAO(persistenceMechanism);
        Friend friend = generateFriendFromRequest(request);
        if (friend != null && persistenceManager.createFriend(friend))
        {
            request.setAttribute("friend", friend);
            request.setAttribute("message", "ha sido creado con
éxito");
            gotoURL(successForm, request, response);
        } else {
            gotoURL(errorForm, request, response);
        }
    }

    Friend generateFriendFromRequest(HttpServletRequest request)
    throws NumberFormatException {
        Friend friend = new Friend();
        String name = request.getParameter("name");
        String phone = request.getParameter("phone");
        String address = request.getParameter("address");
        String comments = request.getParameter("comments");
        String ageString = request.getParameter("age").trim();
        String friendshipLevelString =
        request.getParameter("friendshipLevel");
        boolean valid = valiateName (name) && valiatePhone(phone) &&
            valiateAge(name) &&
        valiateFriendshipLevel(friendshipLevelString);
        if (!valid) {
            return null;
        }
        int age = Integer.parseInt(ageString);
    }
}

```

```

        int friendshipLevel = Integer.parseInt(friendshipLevelString);
        friend.setName(name);
        friend.setPhone(phone);
        friend.setAddress(address);
        friend.setComments(comments);
        friend.setAge(age);
        friend.setFriendshipLevel(friendshipLevel);
        return friend;
    }

    //métodos de validación
    //...
}

```

CreateServlet es, con diferencia, el Servlet más complejo de la aplicación. Tiene que realizar bastante trabajo: extraer un montón de parámetros de un formulario, validarlos, y hacer que los campos de un objeto tomen esos parámetros. Todos los demás Servlets son bastante sencillos. Inclusive UpdateServlet, que tiene que hacer todo lo que hace CreateServlet, y alguna cosa más. Pero su código fuente, y el código fuente de CreateServlet, fue diseñado para que ambos puedan cooperar y no haya que repetir trabajo. CreateServlet es el padre de UpdateServlet, y UpdateServlet aprovecha todo el trabajo que hace su padre extrayendo los campos del formulario, validándolos, y creando el objeto que hay que persistir. Éste es el código completo de UpdateServlet:

```

public class UpdateServlet extends CreateServlet {

    @Override
    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        FriendDAO persistenceManager =
        FriendPersistFactory.getFriendDAO(persistenceMechanism);
        String formerName = request.getParameter("formerName");
        Friend friend = super.generateFriendFromRequest(request);
        if (persistenceManager.updateFriend(formerName, friend)) {

```

```

        request.setAttribute("friend", friend);
        request.setAttribute("message", "ha sido modificada con
éxito");
        gotoURL(successForm, request, response);
    } else {
        gotoURL(errorForm, request, response);
    }
}
}

```

DeleteServlet y ReadServlet tienen una extensión similar a UpdateServlet. Pondremos aquí el código de ReadServlet sólo para hacer énfasis de nuevo en el patrón MVC: observa cómo incluso el Servlet que tiene como misión simplemente presentar datos al usuario, no realiza ninguna tarea relacionada con la generación de HTML. Simplemente consulta al modelo, y le pasa los datos que hay que mostrar a la vista (una página JSP) empleando para ello la sesión del usuario:

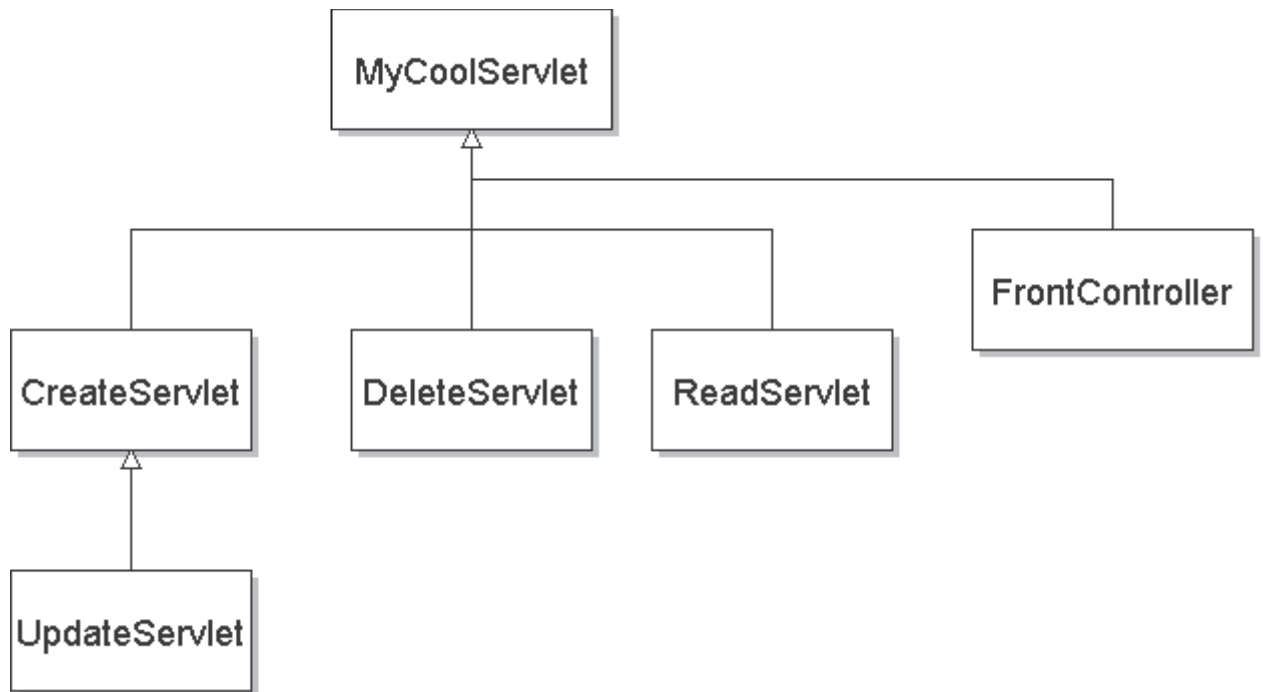
```

public class ReadServlet extends MyCoolServlet {

    @Override
    protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        FriendDAO persistenceManager =
        FriendPersistFactory.getFriendDAO(persistenceMechanism);
        String name = request.getParameter("name");
        Friend friend = persistenceManager.readFriend(name);
        if (friend != null) {
            request.setAttribute("message", "tienen la siguiente
informacion almacenada:");
            request.setAttribute("friend", friend);
            gotoURL(displayForm, request, response);
        } else {
            gotoURL(errorForm, request, response);
        }
    }
}

```

A continuación mostramos un diagrama UML mostrando las relaciones de herencia entre los distintos Servlets de la aplicación:



14.5 La vista

Lo único que nos queda ya es la vista de la aplicación. Una buena parte de la vista de la aplicación ya la hemos cubierto: la página inicial que permite seleccionar la operación CRUD a realizar, más los cuatro formularios de HTML estáticos para cada una de las operaciones (sí, la vista de la aplicación es muy cutre; ya lo advertí desde el principio del tutorial: mi objetivo es explicar la programación del servidor, no del cliente; para el cliente siempre voy a generar el HTML mínimo para salir del paso).

Al margen de las páginas HTML estáticas, tenemos dos página JSP en la aplicación: `displayForm.jsp`, y `successForm.jsp`. Ambas páginas se han situado dentro del directorio `WEB-INF` de la aplicación, dentro de un subdirectorio con nombre "view". De ese modo los usuarios de la aplicación web no podrán realizar peticiones de modo directo a

estos recursos; estos recursos sólo serán accesibles a través de una redirección. La primera, se limita a recuperar de la sesión un objeto de tipo Friend con id "friend" y mostrar sus campos. Esta es la vista que emplea el formulario de las operaciones "Read":



successForm.jsp es una vista genérica de operación realizada con éxito. Esta página JSP también intenta recuperar de la sesión un objeto de tipo Friend con id "friend" y mostrar sus campos. Además, intenta recuperar de la sesión una cadena de texto como mensaje informativo para el usuario. Si consigue recuperar estos objetos con éxito, muestra la cadena de texto con el mensaje informativo y muestra los campos del objeto Friend. En caso negativo, muestra un mensaje genérico diciendo que la operación se realizó con éxito. Éste su código:

```
<html>
  <head>
  </head>
  <body>
    <jsp:useBean id="friend" scope="request" class="model.Friend"
  />

    <jsp:useBean id="message" scope="request"
class="java.lang.String" />
    <meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
```

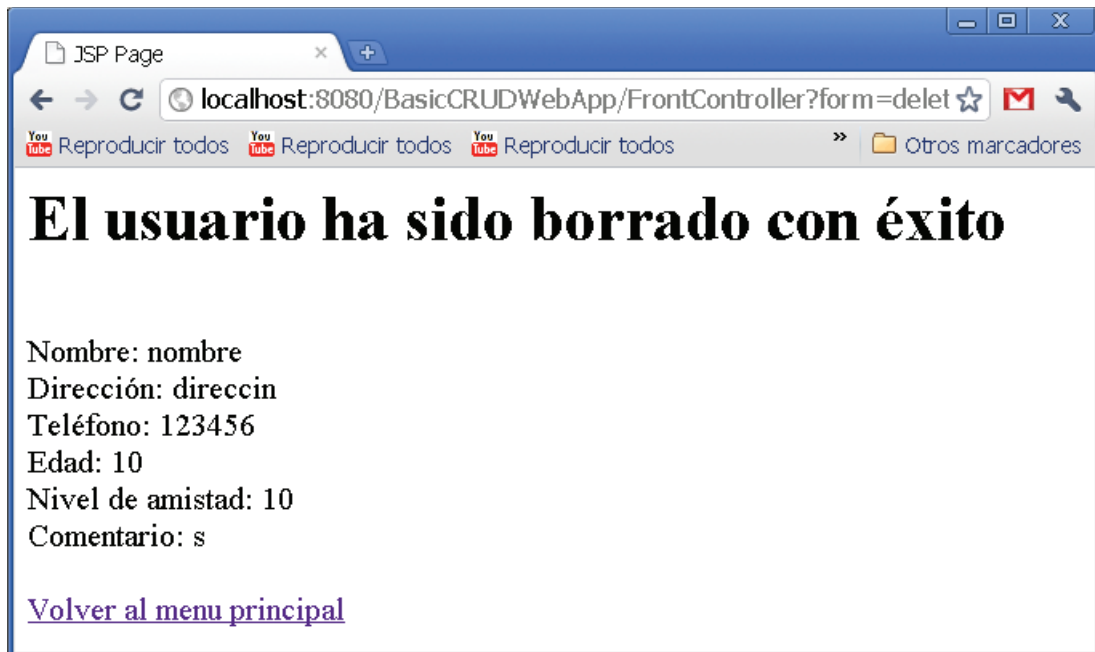
```

<title>JSP Page</title>
<% if(message == null){
    message = ":";
}
%>
<% if(friend.getName() != null){%>
<h1>El usuario <%= message %></h1><BR>
Nombre: <jsp:getProperty name="friend" property="name" /><BR>
Dirección: <jsp:getProperty name="friend" property="address"
/><BR>
Teléfono: <jsp:getProperty name="friend" property="phone"
/><BR>
Edad: <jsp:getProperty name="friend" property="age" /><BR>
Nivel de amistad: <jsp:getProperty name="friend"
property="friendshipLevel" /><BR>
Comentario: <jsp:getProperty name="friend" property="comments"
/><BR><BR>
<%}
else{ %>
Operación realizada con éxito.
<%} %>
<a
href="/BasicCRUDWebApp<%=application.getInitParameter("mainForm")%>">V
olver al menu principal</a>
<% session.removeAttribute("friend");
%>
</body>
</html>

```

Como puedes ver, la vista de la aplicación contiene absolutamente ninguna lógica. Se limita a presentar cosas. Dado el diseño de la aplicación (sobre todo por el uso del controlador frontal que decide a qué recurso hay que dirigir cada petición) resulta trivial modificar esta aplicación para cambiar cualquiera de sus vistas (también para cambiar los Servlets que se encargan de realizar las cuatro operaciones CRUD). Aunque la infraestructura de soporte está ahí, esto no ha sido implementado en la aplicación, más que para un caso: el de la página de éxito. Con sólo variar un parámetro de configuración de contexto en el descriptor de despliegue de la aplicación, podemos

cambiar la página que se muestra cuando una operación ha sucedido con éxito. La que mostramos a continuación es la cutre página correspondiente con una operación de borrado exitosa; ésta es la que se muestra por defecto en la aplicación:



Esta otra página (HTML estático), que todavía es más cutre, es la otra página de éxito de borrado:



Éste es el fragmento de XML del descriptor de despliegue que permite elegir entre ambas páginas:

```

    <!--Cambiando este parametro usamos la pantalla de exito cutre en
    vez de la menos cutre-->

    <context-param>
        <param-name>successForm</param-name>
        <param-value>/WEB-INF/view/successJSP.jsp</param-value>
    </context-param>

    <!-->context-param>
        <param-name>successForm</param-name>
        <param-value>/view/successForm.html</param-value>
    </context-param><!-->

```

La aplicación que acompaña al curso emplea la página menos cutre como vista de éxito por defecto.

Dejo como ejercicio para el lector, por ejemplo, cambiar la vista correspondiente con la página de error (ahora mismo esta vista es el formulario error.html). Sería un ejercicio interesante crear una página de error dinámica que muestre un mensaje explicando en qué consiste el error, e incluso quizás una traza de una excepción, si se generó una excepción, (esto como ejercicio académico; no tiene demasiado sentido mostrar excepciones a los usuarios finales).

Otra extensión interesante sería modificar el flujo de navegación de la operación correspondiente con una actualización; en una primera pantalla se pediría el nombre de la persona, y en una segunda pantalla se mostraría un formulario dinámico (una página JSP) que mostrase en sus campos como valores por defecto los valores actuales de la persona en la base de datos. De ese modo, el usuario sólo tendrá que modificar aquellos campos que quiera cambiar.

14.6 Cada una de las interacciones en más detalle

Para permitir al lector comprender con más detalle para qué vale cada uno de los componentes de la aplicación web, en esta sección mostramos diagramas correspondientes con la navegación de cada una de las cuatro operaciones CRUD. En los diagramas se omite siempre la página inicial de la aplicación (index.html) que permite seleccionar la operación CRUD a realizar. Los nombres que se emplean en los diagramas para referenciar a cada recurso estático (página HTML) o dinámico (Servlet o página JSP) son los nombres (en el caso de los Servlets) o las rutas (en el caso de las páginas HTML y JSP) que habría que emplear en la aplicación para redirigir la petición del usuario a ese recurso empleando un RequestDispatcher.

Diagrama de navegación correspondiente con la creación de personas:

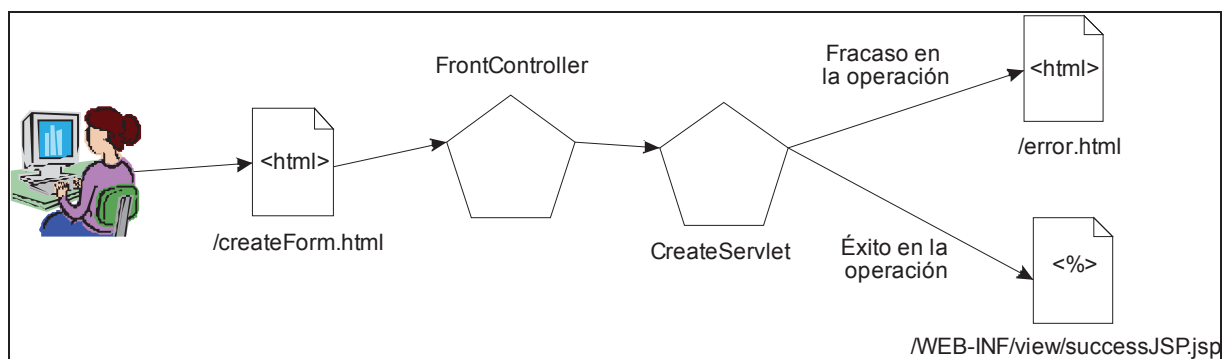


Diagrama de navegación correspondiente con la consulta de una persona:

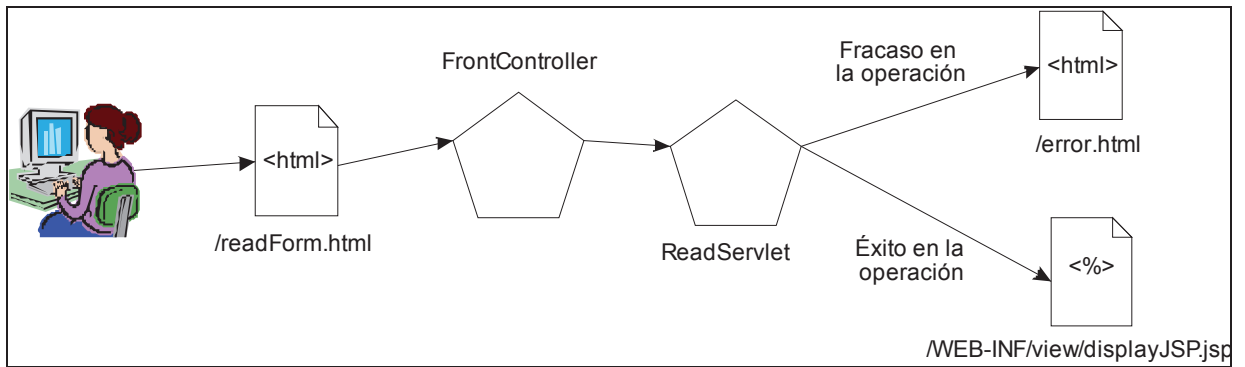


Diagrama de navegación correspondiente con el borrado de personas:

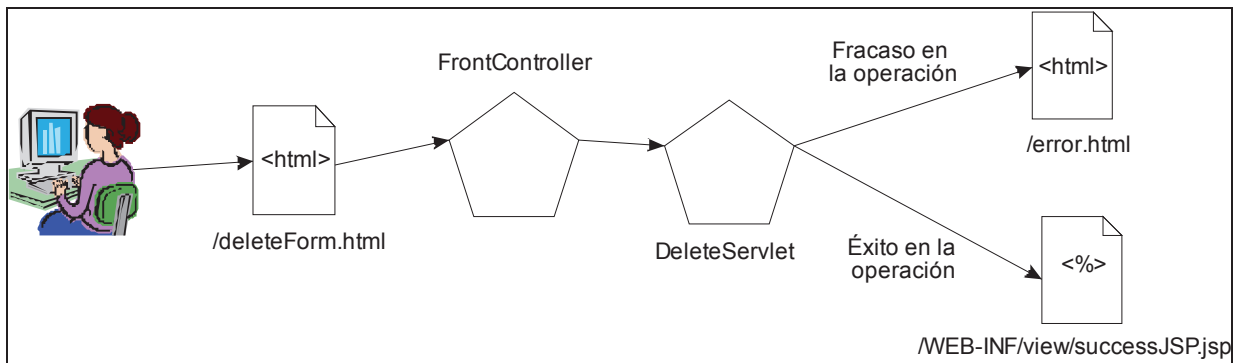
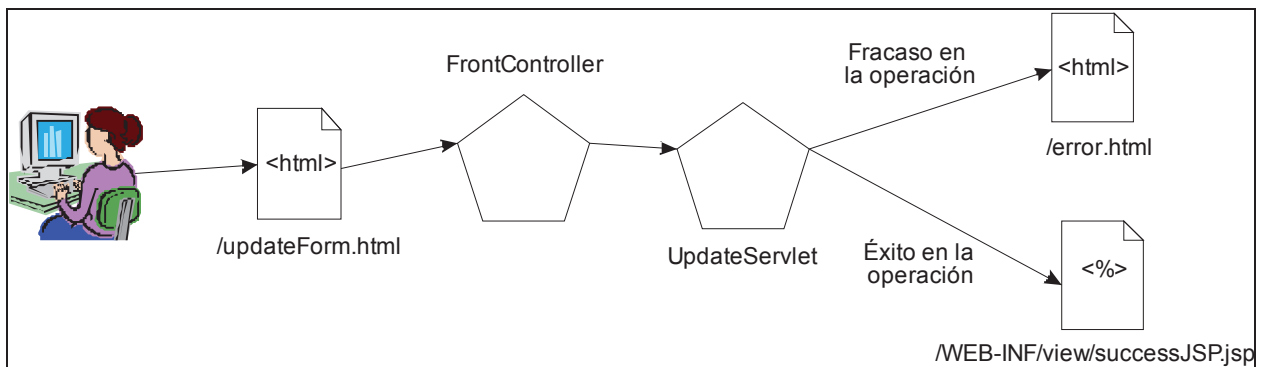


Diagrama de navegación correspondiente con la actualización de personas:



En el descriptor de despliegue de la aplicación se indica que ante cualquier error que se produzca en ella (error 404, error 500 o que se lance cualquier excepción no gestionada en el código Java) se muestre la página estática error.html.

15 ¿Y ahora qué?

Pues ahora yo me planto (que ya he trabajado bastante...) y ahora sigues tú solo. Desde aquí lo que tendría sentido es estudiar algún motor de persistencia como JPA o Hibernate. A día de hoy, lo habitual no es emplear directamente desde una aplicación Java consultas SQL contra una base de datos (con o sin pool de conexiones). En muchas ocasiones, con sólo escribir unas pocas anotaciones en los objetos del modelo y tenemos la mayor parte del trabajo hecho: el motor de persistencia se encarga del resto.

Cosas que no hemos tocado en el tutorial son los temas de autenticación y autorización. Los servidores de aplicaciones Java EE proporcionan mecanismos estandarizados para dar soporte a este tipo de requerimientos. Tampoco hemos hablado nada de servicios web. Dos temas que deberías investigar por tu cuenta.

Por otro lado, en aplicaciones reales tampoco suelen emplearse Servlets y páginas JSP directamente, sino que se suele emplear algún framework web. Spring MVC, Struts 2, GWT y JSF son actualmente los más populares y usados dentro del lenguaje Java. Si estás por la labor de aprender Groovy, puedes ir por Grails. En cualquier caso, ten en cuenta que esta recomendación es válida en el momento de escribir estas líneas (finales de 2010). Quizás cuando tú leas esto las cosas ya hayan cambiado completamente.

Lo que sí que te puedo asegurar, por más que hayan cambiado las cosas, es que te olvides de buscar el mejor framework web. No existe. Cada framework web tiene sus ventajas y desventajas, y se adapta más o menos a un determinado tipo de aplicaciones. Si realmente existiese uno que es "el mejor", todo el mundo lo usaría y no habría más que uno. ¿Y cómo sabes cual se adapta más a tus problemas? ¿O cuál es el mejor para tu futuro profesional?. Nuevamente, puedo dar recomendaciones válidas a finales de 2010 (lo he hecho en el párrafo anterior). Pero si ya ha pasado bastante tiempo, esta recomendación ya habrá caducado. Lo que debes hacer es mantenerte al día. Hay un montón de páginas web en Internet (como javaHispano.org) que deberías visitar de modo regular y leer las noticias, ver de lo que habla la gente, qué frameworks hacen más ruido y menos ruido, de cuáles se habla mal y de cuáles se habla bien... Esto, junto con probarlo tu mismo y ver si te convence, es lo único que realmente funciona.

15.1 Si has llegado hasta aquí estás en deuda con la comunidad

Si has llegado hasta aquí, si te has leído todo este tutorial, visto los videos y te has aprovechado del código fuente, ahora mismo tienes una deuda considerable con la comunidad. Te has beneficiado considerablemente de este material que está disponible de modo gratuito en Internet. Pero tienes la obligación moral de devolver el favor.

¿Cómo? Colabora con la comunidad (con cualquiera, no tiene que ser con javaHispano). Visita los foros y responde dudas de otros que todavía no ha llegado donde tú estás. No sólo vayas al portal de la comunidad a leer las noticias; publica tu también noticias (esto todavía te hace estar más al día que consumir pasivamente el contenido de otros; créeme, sé de qué estoy hablando...). O escribe tú un tutorial. ¡Escribir un documento es una forma fantástica de aprender sobre un nuevo tema!.

Elige tú lo que quieres hacer, pero éstas endeudan con la comunidad, y tienes la obligación moral de pagarla.

Me despido ya. Para cualquier duda, consulta, insulto o tirón de orejas sobre este tutorial puedes ponerte en contacto conmigo enviando un e-mail a abraham@javahispano.org. Espero que te haya resultado útil y ¡suerte con tus aplicaciones!.

