PROGRAMACIÓN DINÁMICA CON JAVA: SERVLETS Y JSP

Índice de contidos

1 Servlets

1.1 Introducción

Fueron introducidos por Sun en 1996 como pequeñas aplicaciones Java para añadir funcionalidad dinámica a los servidores web. Los Servlets, al igual que los scripts CGI, reciben una petición del cliente y generan los contenidos apropiados para su respuesta, aunque el esquema de funcionamiento es diferente.

Todos los servlets asociados con un servidor se ejecutan dentro de un proceso simple. Cada vez que llega una petición, la JVM (*Java Virtual Machine*) crea un hilo Java para manejar la petición, reduciendo así la sobrecarga del sistema (ver Figura 1).

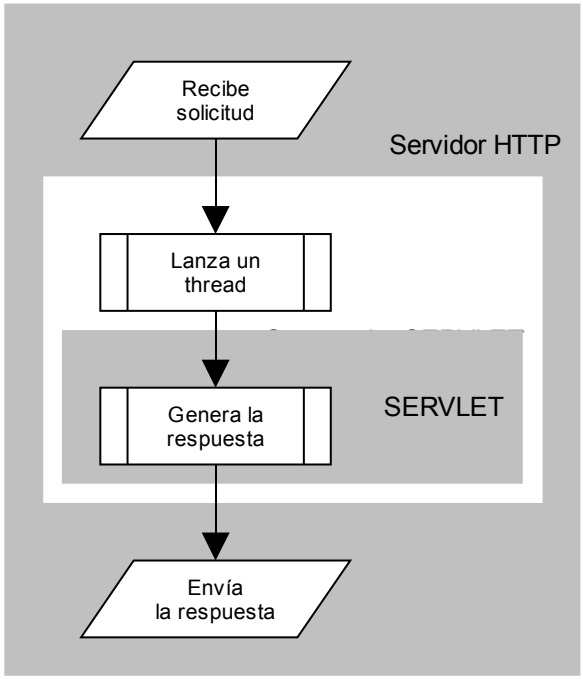

Figura 1: Esquema de ejecución de un servlet

Una desventaja de esta aproximación es que el contenido (estático y dinámico) de la página HTML generada, reside en el código fuente del servlet. Como consecuencia cualquier cambio estático en la página (cambio de una URL de un enlace o una imagen), requiere la intervención del programador y la recompilación del servlet.

Los servlets son el equivalente de los applets en el lado del servidor, aunque a diferencia de los primeros no tienen interfaz gráfico. Se escriben utilizando la API Servlet (que se distribuye como una extensión estándar del JDK 1.2 y posteriores, sin formar parte del núcleo de la plataforma Java).

1.2 Instalación

El primer paso es descargar el software que implementa las especificaciones Java Servlet 2.4 y Java Server Pages 2.0. Se puede obtener una versión gratuita, conocida como Tomcat @ Jakarta en http://jakarta.apache.org/tomcat/.

A continuación es necesario incluir el fichero *servlet-api.jar* (que incorpora todos los paquetes necesarios para trabajar con Servlets y JSP) a la variable *CLASSPATH*, con la finalidad de que el compilador de java *javac*, encuentre las clases necesarias.

Para poder ejecutar un servlet, el servidor HTTP que se esté utilizando debe dar soporte a esta tecnología. La siguiente lista muestra algunos servidores Web que se pueden utilizar:

- Apache.
- Netscape FastTrack o Enterprise.
- Microsoft IIS.
- Weblogic Tengah.
- Lotus Domino Go Webserver.
- IBM Internet Connection Server.
- etc.

En la dirección http://java.sun.com/products/servlet/industry.html se muestra un listado completo de servidores Web que soportan el API Servlet.

Para instalar un servlet en un servidor Web se copia el fichero .class correspondiente al servlet compilado, en el directorio classes del servidor (personalizable para cada usuario).

1.3 Formas de ejecutar un servlet

1. Escribiendo la URL del servlet en el campo de dirección del navegador. Debe seguir la siguiente sintaxis:

   http://your.host.com:port/servlet/ServletName.class[?argumentos]

   Los argumentos opcionales se representan mediante una cadena de consulta estándar codificada en la URL. No siempre debe encontrarse el servlet en el directorio con el nombre *servlet* del servidor, aunque suele ser lo habitual. Depende de la configuración del servidor.

2. También desde una página HTML puede llamarse a un servlet. Para ello se debe emplear la etiqueta adecuada. En el caso de que se trate simplemente de un enlace:

   *<A href="http://localhost:8080/servlet/miServlet">*Click Aquí*</A>*

   Si se trata de un formulario, habrá que indicar la URL del servlet en la propiedad ACTION de la etiqueta <FORM> y especificar el método HTTP (GET o

POST) en la propiedad METHOD de la misma etiqueta.

3. Al tratarse de clases Java como las demás, pueden crearse objetos de dicha clase (ser instanciadas), aunque siempre con el debido cuidado de llamar a aquellos métodos de la clase instanciada que sean necesarios. En ocasiones es muy útil escribir servlets que realicen una determinada función y que sólo puedan ser llamados por otros servlets. En dicho caso, será preciso redefinir su método *service()* de la siguiente forma:

```
public void service(ServletRequest req, ServletResponse resp) throws ServletException, IOException
{
     throw new UnavailableException(this,
        "Este servlet solamente acepta llamadas de otros servlets");
} // fin del método service()
```

En este caso el programador debe llamar explícitamente desde otro servlet los métodos del servlet que quiere ejecutar. Habitualmente los métodos *service()*, *doPost()*, *doGet()*, ..., etc. son llamados automáticamente cuando el servidor HTTP recibe una solicitud de servicio de un formulario introducido en una página HTML.

1.4 Características de los servlets

Además de lo comentado anteriormente, los servlets poseen las siguientes características:

1. Son independientes del servidor utilizado y de su sistema operativo, lo que quiere decir que a pesar de estar escritos en Java, el servidor puede estar escrito en cualquier lenguaje de programación.

2. Los servlets pueden llamar a otros servlets, e incluso a métodos concretos de otros servlets (en la misma máquina o en una máquina remota). De esta forma se puede distribuir de forma más eficiente el trabajo a realizar. Por ejemplo, se podría tener un servlet encargado de la interacción con los clientes y que llamara a otro servlet para que a su vez se encargara de la comunicación con una base de datos.

3. Los servlets pueden obtener fácilmente información acerca del cliente (la permitida por el protocolo HTTP), tal como su dirección IP, el puerto que se utiliza en la llamada, el método utilizado (GET, POST), ..., etc.

4. Permiten además la utilización de *cookies* y *sesiones*, de forma que se puede guardar información específica acerca de un usuario determinado, personalizando de esta forma la interacción cliente/servidor. Una clara aplicación es mantener la sesión con un cliente.

5. Los servlets pueden actuar como enlace entre el cliente y una o varias bases de datos en arquitecturas *cliente-servidor de 3 capas* (si la base de datos está en un servidor distinto).

6. Asimismo, pueden realizar tareas de *proxy* para un applet. Debido a las

restricciones de seguridad, un applet no puede acceder directamente a un servidor de datos localizado en cualquier máquina remota, pero sí podría hacerlo a través de un servlet.

7. Al igual que los programas CGI, los servlets permiten la generación dinámica de código HTML. Así, pueden emplearse servlets para la creación de contadores, banners, etc.

Los paquetes javax.servlet y javax.servlet.http

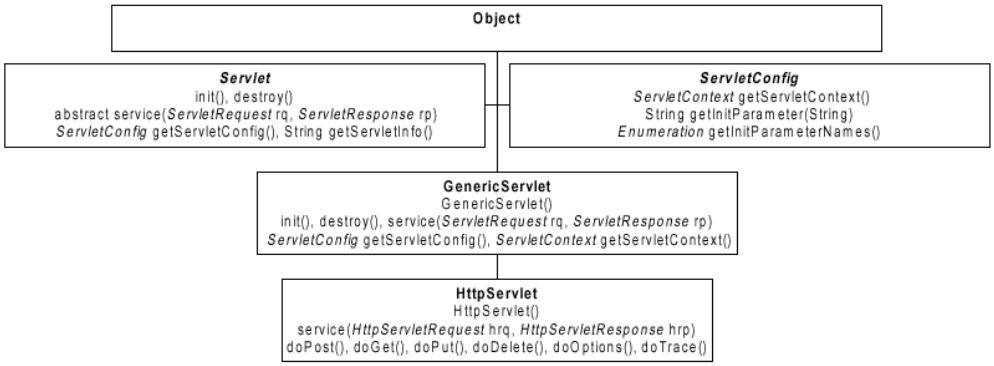

Figura 2: Jerarquía de clases para los servlets

El paquete javax.servlet

El paquete **javax.servlet** define las siguientes clases e interfaces:

Interfaces	
	Servlet
	ServletRequest
	ServletResponse
	ServletConfig
	ServletContext
	SingleThreadModel
Clases	
	GenericServlet
	ServletInputStream
	ServletOutputStream

- **Servlet**: debe ser implementada por todos los servlets. El servidor invoca a los métodos *init()* y *destroy()* para iniciar y detener un servlet. Los métodos *getServletConfig()* y *getServletInfo()* se implementan para devolver información acerca de un servlet. Al método *service()* lo invoca el servidor, de forma que el servlet pueda realizar su servicio. Contiene dos parámetros: uno de la interfaz *ServletRequest* y otro de la interfaz *ServletResponse*.

- **ServletRequest**: encapsula la solicitud de servicio de un cliente. Define una serie de métodos tendentes a obtener información acerca del servidor, solicitante y solicitud. El método *getInputStream()* devuelve un objeto de la clase *ServletInputStream* susceptible de ser utilizado para leer información

sobre solicitudes que hubiera enviado el cliente.

- **ServletResponse**: es utilizada por el servlet para responder a una solicitud enviando información al solicitante. El método *getOutputStream()* devuelve un objeto de la clase *ServletOutputStream* que se usa para enviar información de respuesta al cliente. El método *getWriter()* devuelve un objeto *PrintWriter* que se usa para la comunicación de cliente. El método *setContentType()* establece el tipo MIME de la información de respuesta. El método *setContentLength()* especifica la longitud de la respuesta en bytes. El método *getCharacterEncoding()* devuelve el tipo MIME que viene asociado a la respuesta.

- **ServletConfig**: esta interfaz la utiliza el servidor para pasar información sobre la configuración a un servlet. Sus métodos los utiliza el servlet para recuperar esta información.

- **ServletContext**: define el entorno en el que se ejecuta un servlet. Proporciona métodos utilizados por servlets para acceder a la información sobre el entorno.

- **SingleThreadModel**: se usa para identificar a los servlets que no pueden ser accedidos por más de un hilo al mismo tiempo. Si un servlet implementa esta interfaz, el servidor web no ejecutará al mismo tiempo el método *service()* de más de una instancia del servlet.

- **GenericServlet**: implementa la interfaz *Servlet*. Se puede heredar de esta clase para implementar servlets de usuario.

- **ServletInputStream**: se utiliza para acceder a información de solicitudes que suministra un cliente web. El método *getInputStream()* de la interfaz *ServletRequest* devuelve un objeto de esta clase.

- **ServletOutputStream**: se usa para enviar información de respuesta a un cliente web. El método *getOutputStream()* de la interfaz *ServletResponse* devuelve un objeto de esta clase.

El paquete javax.servlet.http

El paquete **javax.servlet.http** se usa para definir servlets específicos de HTTP. Define las siguientes clases e interfaces:

Interfaces	
	HttpServletRequest
	HttpServletResponse
	HttpSession
	HttpSessionBindingListener
	HttpSessionContext
Clases	
	Cookie

HttpServlet
HttpSessionBindingEvent
HttpUtils

- **HttpServletRequest**: amplía la interfaz *ServletRequest* y agrega métodos para acceder a los detalles de una solicitud HTTP.

- **HttpServletResponse**: amplía la interfaz *ServletResponse* y agrega constantes y métodos para devolver respuestas específicas de HTTP.

- **HttpSession**: la implementan los servlets que permiten dar soporte a las sesiones entre el navegador y el servidor (la sesión existe durante un determinado período de tiempo, e identifica al usuario en múltiples páginas web). HTTP es un protocolo sin estado, el estado se mantiene externamente por medio de *cookies* del lado del cliente o la escritura de una nueva URL. Esta interfaz proporciona métodos para leer y escribir valores de estado.

- **HttpSessionBindingListener**: la implementan clases cuyos objetos están asociados a sesiones HTTP. El método *valueBound()* se utiliza para notificar a un objeto que está asociado a una sesión, mientras que el método *valueUnboud()* se emplea para notificar a un objeto que ya no está ligado a una sesión.

- **HttpSessionContext**: se utiliza para representar a una colección de objetos *HttpSession* que están asociados a los identificadores de sesión. El método *getIds()* devuelve una lista de identificadores de sesión. El método *getSession()* devuelve el objeto *HttpSession* que está asociado a un identificador de sesión determinado. Los identificadores de sesión se implementan con objetos *String*.

- **Cookie**: esta clase representa un cookie HTTP. Las cookies se utilizan para mantener el estado de sesión sobre solicitudes HTTP múltiples. Se trata de valores de datos con nombre que se crean en el servidor web y se almacenan en los navegadores cliente de cada usuario. La clase *Cookie* proporciona métodos para obtener y establecer valores de cookie y de otros atributos.

- **HttpServlet**: amplía la clase *GenericServlet* con el fin de utilizar las interfaces *HttpServletRequest* y *HttpServletResponse*.

- **HttpSessionBindingEvent**: esta clase implementa el evento que se genera cuando un objeto está ligado o se desliga de una sesión HTTP.

- **HttpUtils**: proporciona el método *parseQueryString()* para analizar sintácticamente una cadena de consulta contenida en una solicitud HTTP.

1.5 Ejemplos básicos

Para ejemplificar el desarrollo de servlets, se detallan en lo que sigue varios ejemplos prácticos comentando su programación y funcionamiento.

TimeServlet

A continuación se lista el código del servlet *TimeServlet.java*, que muestra en el navegador del cliente la fecha y la hora a la que se ha conectado con el servidor.

```
import javax.servlet.*;
import java.util.*;
import java.io.*;

public class TimeServlet extends GenericServlet
{
          public String getServletInfo()
          {
               return "Time Servlet";
          }

          public void service(ServletRequest peticion, ServletResponse respuesta) throws ServletException, IOException
          {
               String fecha = new Date().toString();

               PrintStream outputStream = new PrintStream(respuesta.getOutputStream());
               outputStream.println(fecha);
          }
};
```

Para trabajar con servlets es necesario importar el paquete *javax.servlet*.

La clase *TimeServlet* extiende la clase *GenericServlet*, y sobreescribe los métodos *getServletInfo()* y *service()*.

El método *service()* implementa el manejo real de la solicitud y respuesta del servlet. El servidor web invoca a este método cuando se solicita la URL del servlet. El servidor pasa los argumentos *ServletRequest* y *ServletResponse* al servlet.

TimeServlet crea un objeto *Date* nuevo, lo convierte en un *String* y lo almacena por medio de la variable *fecha*. Utiliza el método *getOutputStream()* de la clase *ServletResponse* para crear un objeto *ServletOutputStream* que envíe información de respuesta de vuelta al navegador cliente.

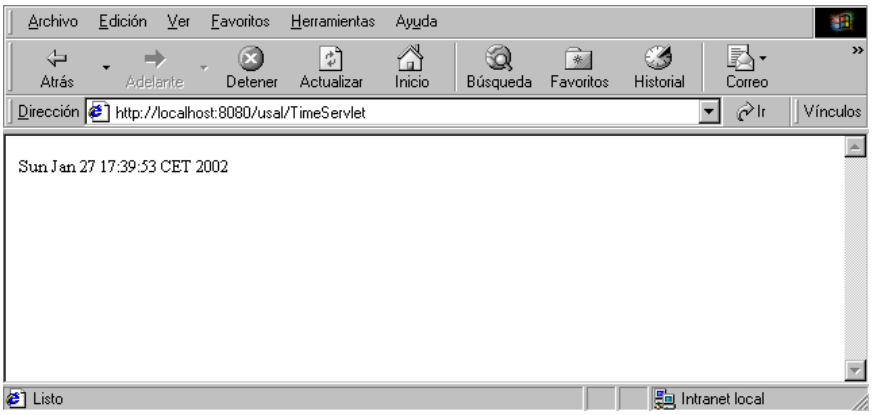

Figura 3: Resultado de la ejecución de TimeServlet.class

EchoRequest

El servlet mostrado a continuación es invocado desde un formulario contenido en un fichero *.html*. Su finalidad es procesar la información que le llega en forma de parámetros *ServletRequest* y enviárselos de nuevo al cliente con información ampliada.

El fichero *inscripcion.html* que llama al servlet contiene el siguiente código.

```
<FORM action="servlet/EchoRequest.class" method="POST"
        <P>
        NOMBRE: <BR> <INPUT type="text" name="Nombre">
        <P>
        E-MAIL: <BR> <INPUT type="password" name="Email">
        <P>
        TIPO DE INSCRIPCIÓN: <BR>
                <INPUT type="radio" name="Inscripcion" value="Normal"> Normal <BR>
                <INPUT type="radio" name="Inscripcion" value="Estudiante"> Estudiante <BR>
                <INPUT type="radio" name="Inscripcion" value="Socio de ATI"> Socio de ATI
        <P>
        TIPO DE COMUNICACIÓN: <BR>
                <INPUT type="checkbox " name="TIPO_1" value="paper"> Artículo <BR>
                <INPUT type="checkbox " name="TIPO_2" value="poster"> Poster <BR>
                <INPUT type="checkbox " name="TIPO_3" value="demo"> Demo Comercial
        <P>
        ORGANIZACIÓN: <BR> <INPUT type="text" name="Organizacion">
        <P>
        PAÍS: <BR>
              <SELECT size="5" name="Pais">
                  <OPTION>ESPAÑA</OPTION>
                  <OPTION>PORTUGAL</OPTION>
                  <OPTION>FRANCIA</OPTION>
                  <OPTION>ALEMANIA</OPTION>
                  <OPTION>ITALIA</OPTION>
                  <OPTION>REINO UNIDO</OPTION>
                  <OPTION>EEUU</OPTION>
                  <OPTION>AFRICA</OPTION>
              </SELECT>
        <P>
        <INPUT type="submit" value="Realizar Inscripción">
        <INPUT type="reset" value="Borrar">
</FORM>
```

Cuando el usuario pulse el botón Realizar Inscripción se ejecutará el servlet codificado a continuación.

```
import javax.servlet.*;
import java.util.*;
import java.io.*;

public class EchoRequest extends GenericServlet
{
        public String getServletInfo()
        {
             return "Echo Request Servlet";
        }

        public void service(ServletRequest peticion, ServletResponse respuesta) throws ServletException, IOException
        {
             respuesta.setContentType("text/plain");

             PrintStream outputStream = new PrintStream(respuesta.getOutputStream());
             outputStream.print("Servidor: " + peticion.getServerName() + ":");
             outputStream.println (peticion.getServerPort());
             outputStream.print("Cliente: " + peticion.getRemoteHost() + " ");
             outputStream.println(peticion.getRemoteAddr());
             outputStream.println("Protocolo: " + peticion.getProtocol());

             Enumeration params = peticion.getParameterNames();
             if (params != null)
             {
                    while (params.hasMoreElements())
                    {
                           String parametro = (String) params.nextElement();
                           String valor = peticion.getParameter(parametro);
                           outputStream.println(parametro + " = " + valor);
                    }
             }
        }
};
```

Si se carga el fichero *.html* en un navegador y se realiza la inscripción el servlet generará la salida mostrada en la Figura 4.

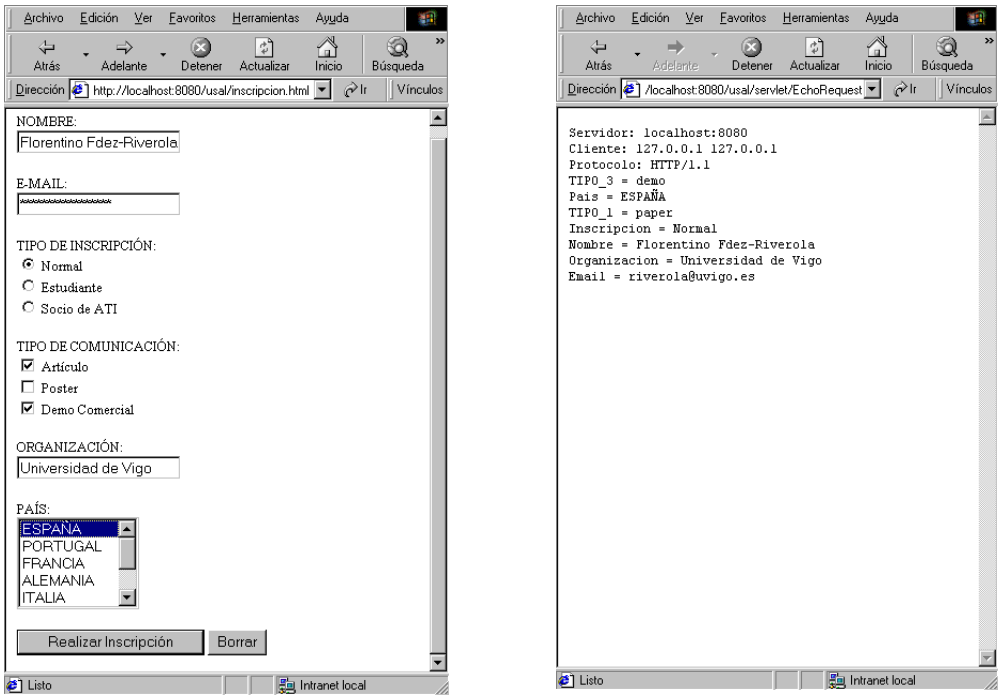

Figura 4: Resultado de la carga del fichero *inscripción.html* y ejecución del servlet *EchoRequest.class*

HTMLServlet

A continuación se lista el código del servlet *HTMLServlet.java*, que genera una página web (fichero *.html*) que será visualizado por el cliente.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HTMLServlet extends HttpServlet
{
          public String getServletInfo()
          {
               return "HTML Servlet";
          }

          public void service(ServletRequest peticion, ServletResponse respuesta) throws ServletException, IOException
          {
               respuesta.setContentType("text/html");
               PrintWriter out = respuesta.getWriter();
               out.println("<HTML>");
               out.println("<HEAD> <TITLE>Página genereda por el servlet HTMLServlet</TITLE> <HEAD>");
               out.println("<BODY BGCOLOR=\"BLACK\" TEXT=\"WHITE\">");
               out.println("<CENTER><H1>PÁGINA GENERADA POR HTMLSERVLET</H1></CENTER>");
               out.println("<P>");
               out.println("<CENTER>para " +
                                        peticion.getRemoteHost() +
                                        " en " +
                                        peticion.getRemoteAddr() + "</CENTER>");
               out.println("</BODY>");
               out.println("</HTML>");
          }
};
```

Se puede ejecutar el servlet directamente desde el navegador con la línea.

http://localhost:8080/usal/servlet/HTMLServlet

El resultado de la ejecución del servlet *HTMLServlet* se muestra en la Figura 5.

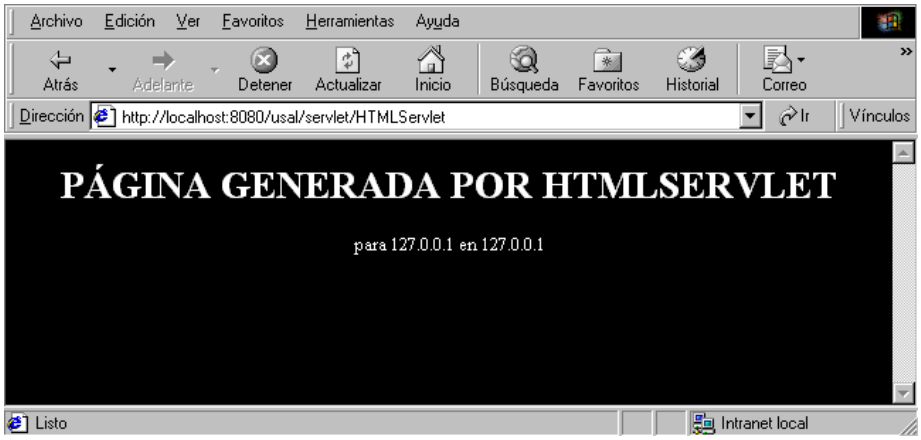

Figura 5: Resultado de ejecución del servlet *HTMLServlet.class*

Lectura de cabeceras de solicitud

Cuando un cliente HTTP (por ejemplo, un navegador) envía una petición, es necesario que suministre una línea de petición (normalmente GET o POST). Es posible también enviar un número de cabeceras que acompañan a la petición y que son opcionales excepto *Content-Length*, que es requerida sólo para peticiones POST. La Tabla 1 muestra las cabeceras más comunes que pueden estar presentes en una petición HTTP.

Cabecera HTTP	Significado
Accept	Tipos MIME que prefiere el navegador.
Accept-Charset	Conjunto de caracteres que espera el navegador.
Accept-Encoding	Tipos de codificación de datos (como gzip) para que el navegador sepa cómo decodificarlos. Los servlets pueden comprobar explícitamente el soporte para gzip y devolver páginas HTML comprimidas con gzip para navegadores que las soportan, seleccionando la cabecera de respuesta *Content-Encoding* para indicar que están comprimidas con gzip. En muchos casos, esto puede reducir el tiempo de descarga por un factor de cinco o diez.
Accept-Language	Idioma que está esperando el navegador, en el caso de que el servidor tenga versiones en más de un idioma.
Authorization	Información de autorización, usualmente en respuesta a una cabecera *www-Authenticate* enviada desde el servidor.
Connection	¿Se usan conexiones persistentes?. Si un servlet recupera un valor *Keep-Alive* de esta cabecera, u obtiene una línea de petición con HTTP 1.1 (donde las conexiones son persistentes por defecto), es posible ahorrar un tiempo considerable en el envío de páginas web que incluyen muchas piezas pequeñas (imágenes o applets). Para hacer esto, es necesario enviar una cabecera *Content-Length* en la respuesta, escribiendo en un *ByteArrayOutputStream*, y preguntando por el tamaño antes de escribir la salida.
Content-Length	En mensajes POST especifica el número de datos que se han añadido.
Cookie	Manejo de Cookies. (se explica porteriormente).
From	Especifica la dirección de correo electrónico del usuario que realiza la petición. Sólo es usado por aceleradores web, no por clientes personalizados ni por navegadores.
Host	Host y puerto escuchado en la URL original
If-Modified-Since	Sólo devuelve documentos más nuevos que el especificado, de otra forma se envía una respuesta *304 "Not Modified" response*.
Pragma	El valor *no-cache* indica que el servidor debería devolver un documento nuevo, incluso si es un proxy con una copia local.
Referer	URL de la página que contiene el enlace que el usuario siguió para obtener la página actual.
User-Agent	Especifica el tipo de navegador. Útil cuando el servlet está devolviendo contenido específico para un navegador.
UA-Pixels, UA-Color, UA-OS, UA-CPU	Cabeceras no estándar enviadas por algunas versiones de Internet Explorer, indicando el tamaño de la pantalla, la profundidad del color, el sistema operativo, y el tipo de CPU usada por el sistema del navegador.

Tabla 1: Cabeceras utilizadas en el protocolo HTTP

La lectura de cabeceras es muy sencilla, sólo se necesita llamar al método *getHeader()* de *HttpServletRequest*, que devuelve un *String* si se suministró la cebecera en esta petición, y *null* si no se suministró. Sin embargo, hay un par de cabeceras que se usan de forma tan común que tienen métodos de acceso especiales. El método *getCookies()* devuelve el contenido de la cabecera *Cookie*, lo analiza y lo almacena en un array de objetos *Cookie*. Los métodos *getAuthType()* y *getRemoteUser()* dividen la cabecera *Authorization* en sus componentes. Los métodos *getDateHeader()* y *getIntHeader()* leen la cabecera específica y la convierten a valores *Date* e *int*, respectivamente.

En vez de buscar una cabecera particular, se puede utilizar el método *getHeaderNames()* para obtener una *Enumeration* de todos los nombres de cabecera de una petición particular. Finalmente, además de buscar las cabeceras de petición, se puede obtener información sobre la propia línea de petición principal. El método *getMethod()* devuelve el método de petición principal (normalmente GET o POST, pero también son valores válidos HEAD, PUT, y DELETE). El método *getRequestURI()* devuelve la URI (parte de la URL que viene después del host y el puerto, pero antes de los datos del formulario). El método *getRequestProtocol()* devuelve la tercera parte de la línea de petición que generalmente es "HTTP/1.0" o "HTTP/1.1".

El siguiente código muestra todas las cabeceras enviadas por el navegador cliente cuando invoca al servlet desde la barra de direcciones. El resultado se muestra en la Figura 6.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestHeader extends HttpServlet  {

    public void doGet(HttpServletRequest peticion, HttpServletResponse respuesta) throws IOException, ServletException
    {
          respuesta.setContentType("text/html");
          PrintWriter out = respuesta.getWriter();

                out.println("<HTML>");
                out.println("<HEAD><TITLE>Página genereda por el servlet RequestHeader</TITLE><HEAD>");

                out.println("<BODY BGCOLOR=\"WHITE\" TEXT=\"BLACK\">");
                out.println("<H1>MOSTRANDO LAS CABECERAS ENVIADAS POR EL CLIENTE</H1>");
                out.println("<P>");
                out.println("Request Method: " + peticion.getMethod() + "<BR>");
                out.println("Request URI: " + peticion.getRequestURI() + "<BR>");
                out.println("Request Protocol: " + peticion.getProtocol() + "<BR>");
                out.println("<P>");

                out.println("<table border=\"1\">");

                Enumeration e = peticion.getHeaderNames();
                while (e.hasMoreElements())
                {
                        String nombre = (String)e.nextElement();
                        String valor = peticion.getHeader(nombre);
                out.println("<tr><td>" + nombre + "</td><td>" + valor + "</td></tr>");
                }
                out.println("</table>");
                out.println("</BODY>");
                out.println("</HTML>");
    }
}
```

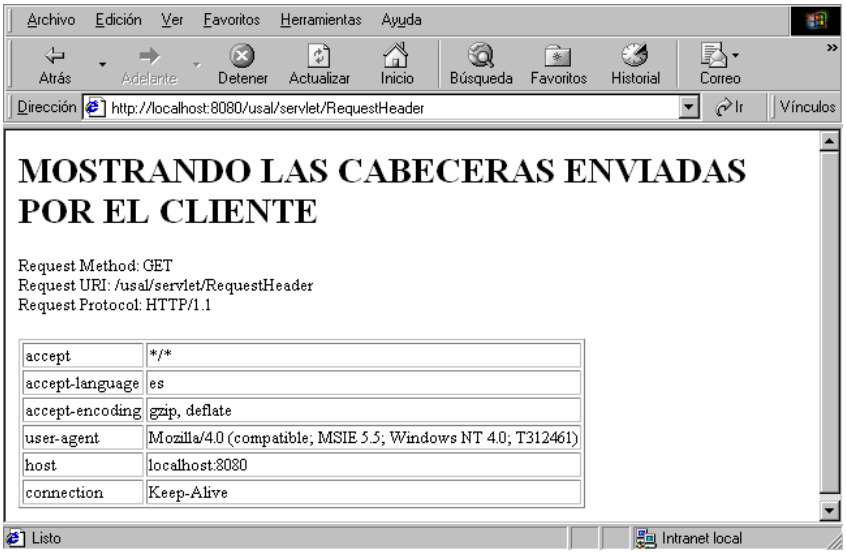

Figura 6: Resultado de ejecución del servlet *RequestHeader.class*

Cookies

Las *cookies* en Java son objetos de la clase *Cookie*, definida en el paquete *javax.servlet.http*. El empleo de cookies en el seguimiento de un cliente que realiza varias conexiones al servidor web, requiere que dicho cliente sea capaz de soportarlas.

Cada cookie tiene un nombre que puede ser el mismo para varias cookies. De esta forma, se puede mantener información acerca del cliente durante días, ya que esa información queda almacenada en el ordenador del cliente (aunque no indefinidamente, pues las cookies tienen una fecha de caducidad).

Las cookies se añaden en la cabecera de un mensaje HTTP, pudiendo enviarse más de una al mismo tiempo. Las cookies almacenadas en el cliente son enviadas, en principio, sólo al servidor que las originó. Por eso, los servlets que se ejecutan en un mismo servidor comparten las mismas cookies.

Los pasos para crear y enviar una cookie son los siguientes:

1. Crear un objeto *Cookie*.
2. Establecer sus atributos.
3. Enviar la cookie.

Por otra parte, para obtener información de una cookie, es necesario:

1. Recoger todas las cookies de la petición del cliente.
2. Encontrar la cookie precisa.
3. Obtener el valor asignado a la misma.

Crear un objeto cookie

La clase *javax.servlet.http.Cookie* tiene un constructor que presenta como argumentos un *String* con el nombre de la cookie y otro *String* con su valor. Es

importante hacer notar que toda la información almacenada en cookies lo es en forma de *String*, por lo que será preciso convertir cualquier valor a *String* antes de añadirlo a una cookie.

El nombre de la cookie no puede empezar por el símbolo dólar ($). El valor asignado a un nombre, no puede contener espacios en blanco ni ninguno de los siguientes caracteres: [ ] ( ) = , " / ¿ @ : ;

Es necesario crear la cookie antes de acceder al *Writer* del objeto *HttpServletResponse*, pues las cookies son enviadas al cliente en la cabecera del mensaje.

Un ejemplo de creación de una cookie se muestra en el siguiente código:

```
String IdObjetoAComprar = new String("301");
if(IdObjetoAComprar != null)
              Cookie miCookie = new Cookie("Compra", IdObjetoAComprar);
```

Establecer los atributos de la cookie

La clase *Cookie* proporciona varios métodos para establecer los valores de una cookie y sus atributos. Entre otros los mostrados en la Tabla 2:

void *setComment*(String)	Si un navegador presenta esta cookie al usuario, el cometido de la cookie será descrito mediante este comentario.
void *setDomain*(String)	Establece el patrón de dominio a quien permitir el acceso a la información contenida en la cookie. Por ejemplo *.yahoo.com*.
void *setMaxAge*(int)	Establece el tiempo de caducidad de la cookie en segundos. Un valor –1 indica al navegador que borre la cookie cuando se cierre. Un valor 0 borra la cookie de inmediato.
void *setPath*(String)	Establece la ruta de acceso del directorio de los servlets que tienen acceso a la cookie. Por defecto es aquel que originó la cookie.
void *setSecure*(boolean)	Indica al navegador que la cookie sólo debe ser enviada utilizando un protocolo seguro (HTTPS).
void *setValue*(String)	Establece el valor de la cookie.
void *setVersion*(int)	Establece la versión del protocolo de la cookie.

Tabla 2: Métodos comunes de la clase Cookie

Todos estos métodos tienen sus métodos *getX()* correspondientes incluidos en la misma clase.

Por ejemplo, se puede cambiar el valor de una cookie de la siguiente forma:

```
Cookie miCookie = new Cookie("Nombre", "ValorInicial");
miCookie.setValue("ValorFinal");
```

o hacer que el cliente la elimine al cerrar el navegador.

```
miCookie.setMaxAge(-1);
```

Envío de la cookie

Las cookies son enviadas como parte de la cabecera de la respuesta al cliente. Por ello, tienen que ser añadidas a un objeto *HttpServletResponse* mediante el método

addCookie(Cookie). Esto debe realizarse antes de llamar al método *getWriter()* de ese mismo objeto. El siguiente código muestra un ejemplo:

```
public void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException
{
      //  ...
      Cookie miCookie = new Cookie("Nombre","Valor");
      miCookie.setMaxAge(-1);
      miCookie.setComment("Esto es un comentario");
      resp.addCookie(miCookie);

      PrintWriter out = resp.getWriter();
      //  ...
}
```

Recogida de cookies

Los clientes devuelven las cookies como parte integrante del header de la petición al servidor. Por este motivo, las cookies enviadas deberán ser recogidas del objeto *HttpServletRequest* mediante el método *getCookies()*, que devuelve un array de objetos *Cookie*. A continuación se muestra un ejemplo donde se recoge la primera cookie del array de cookies:

```
Cookie miCookie = null;
Cookie[] arrayCookies = req.getCookies();
miCookie = arrayCookies[0];
```

Hay que tener cuidado, pues tal y como se ha mencionado con anterioridad, puede haber más de una cookie con el mismo nombre, por lo que habrá que detectar de alguna manera cuál es la cookie que se necesita.

Obtener el valor de una Cookie

Para obtener el valor de una cookie se utiliza el método *getValue()* de la clase *Cookie*. El siguiente ejemplo supone que se tiene una tienda virtual de libros, y que un usuario ha decidido eliminar un libro del carro de la compra. Se tienen dos cookies con el mismo nombre (*compra*), pero con dos valores (*libro1*, *libro2*). Si se quiere eliminar el valor *libro1* es necesario el siguiente código:

```
//  ...
String libroABorrar=req.getParameter("Borrar");
//  ...
if(libroABorrar != null)
{
      Cookie[] arrayCookies = req.getCookies();
      for(i=0;i<arrayCookies.length;i++)
      {
            Cookie miCookie=arrayCookies[i];
            if(miCookie.getName().equals("compra") && miCookie.getValue.equals(libroABorrar)
            {
                  miCookie.setMaxAge(0); // Elimina la cookie
            } // fin del if
      } // fin del for
} // fin del if
//  ...
```

La página *.html* mostrada en la Figura 7 permite establecer las cookies que son enviadas posteriormente al navegador cliente.

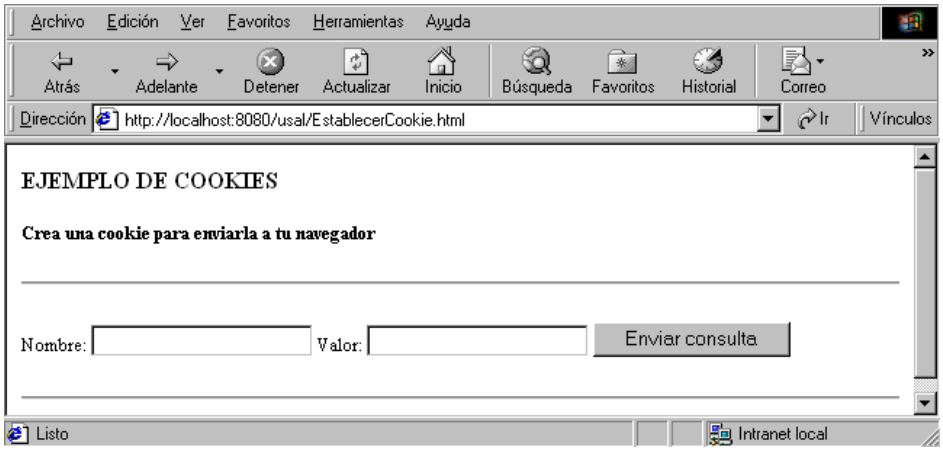

Figura 7: Vista de la página *EstablecerCookie.html*

El código del servlet encargado de crear las cookies, enviarlas al navegador cliente y visualizar las existentes se muestra a continuación.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SetCookie extends HttpServlet  {

    public void doGet(HttpServletRequest peticion, HttpServletResponse respuesta) throws IOException, ServletException
    {
        String nombre, valor;
        respuesta.setContentType("text/html");
        PrintWriter out = respuesta.getWriter();

        // Se visualizan la cookies existentes
        Cookie[] cookies = peticion.getCookies();
        if (cookies != null)
               for (int i = 0; i < cookies.length; i++)
               {
                       Cookie c = cookies[i];
                       nombre = c.getName();
                       valor = c.getValue();
                       out.println(nombre + " = " + valor + "<BR>");
               }

        // Se añade la cookie recibida a la respuesta
        nombre = peticion.getParameter("NombreCookie");
        if (nombre != null && nombre.length() > 0)
        {
               valor = peticion.getParameter("ValorCookie");
               Cookie c = new Cookie(nombre, valor);
               respuesta.addCookie(c);
        }
    }

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

Manejo de Sesiones (Session Tracking)

Una sesión es una conexión continuada de un mismo navegador a un servidor web durante un intervalo prefijado de tiempo. Este tiempo depende habitualmente del servidor, aunque a partir de la versión 2.1 de la API Servlet puede establecerse mediante el método *setMaxInactiveInterval(int)* de la interfaz *HttpSession*. Esta interfaz es la que proporciona los métodos necesarios para mantener sesiones.

La forma de obtener una sesión es mediante el método *getSession(boolean)* de un objeto *HttpServletRequest*. Si el parámetro es true se crea una sesión nueva si es necesario, mientras que si es false, el método devolverá la sesión actual.

Una vez que se tiene un objeto *HttpSession*, es posible mantener una colección de pares nombre/valor, de forma que pueda almacenarse todo tipo de información sobre la sesión. Este valor puede ser cualquier objeto de la clase *Object* que se desee. La forma de añadir valores a la sesión es mediante el método *putValue(String, Object)* de la interfaz *HttpSession*, y la de obtenerlos mediante el método *getValue(String)* del mismo objeto.

Además de estos métodos mencionados, la interfaz *HttpSession* define los métodos mostrados en la Tabla 3.

long *getCreationTime*()	Devuelve el momento en que fue creado la sesión (en milisegundos).
long *getLastAccessedTime*()	Devuelve el último momento en que el cliente realizó una petición con el identificador asignado a una determinada sesión (en milisegundos).
String [] *getValueNames*()	Devuelve un array con todos los nombre de los objetos asociados con la sesión.
void *invalidate*()	Invalida la sesión en curso.
boolean *isNew*()	Devuelve un boolean indicando si la sesión es nueva.
void *removeValue*(String)	Elimina el objeto asociado con una determinada clave.

Tabla 3: Métodos comunes de la interfaz HttpSession

La página *.html* mostrada en la Figura 8 permite establecer pares objeto/valor en la sesión actual.

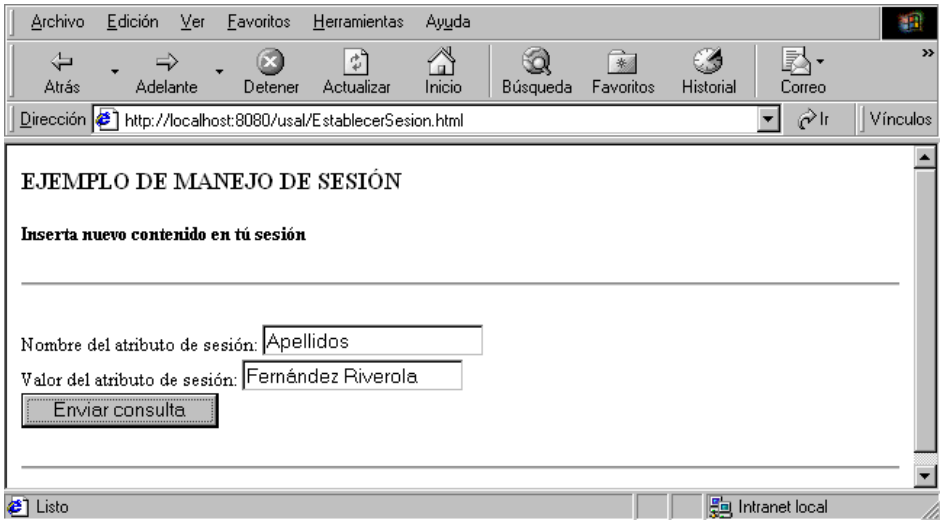

Figura 8: Vista de la página *Establecer Session.html*

El código del servlet encargado de crear la sesión, agregar nuevos pares objeto/valor y visualizar los existentes se muestra a continuación.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SetSession extends HttpServlet  {

    public void service(HttpServletRequest peticion, HttpServletResponse respuesta) throws IOException, ServletException
    {
        respuesta.setContentType("text/html");
        PrintWriter out = respuesta.getWriter();

        out.println("<H3>Ejemplo de Sesiones</H3>");

        // Se crea la sesion si es necesario
        HttpSession session = peticion.getSession(true);
```

```
        // Visualizar informacion de la sesion
        Date created = new Date(session.getCreationTime());
        Date accessed = new Date(session.getLastAccessedTime());

        out.println("<hr>");
        out.println("<b>ID:</b> " + session.getId() + "<br>");
        out.println("<b>Fecha de creación:</b> " + created + "<br>");
        out.println("<b>Fecha de último acceso:</b> " + accessed + "<br>");
        out.println("<hr><p>");

        // Se almacena nueva informacion el el objeto HttpSession
        String nombreDato = peticion.getParameter("NombreDato");
        if (nombreDato != null && nombreDato.length() > 0)
        {
                String valorDato = peticion.getParameter("ValorDato");
                session.setAttribute(nombreDato, valorDato);
        }

        out.println("<b><i>Los siguientes datos están en tu sesión:</b></i><br>");

        // Visualizar objetos contenidos en la sesion actual
        Enumeration e = session.getAttributeNames();
        while (e.hasMoreElements())
        {
                String nombre = (String)e.nextElement();
                String valor = session.getAttribute(nombre).toString();
                out.println(nombre + " = " + valor + "<BR>");
        }
    }
}
```

El resultado de incluir nueva información en la sesión se muestra en la Figura 9.

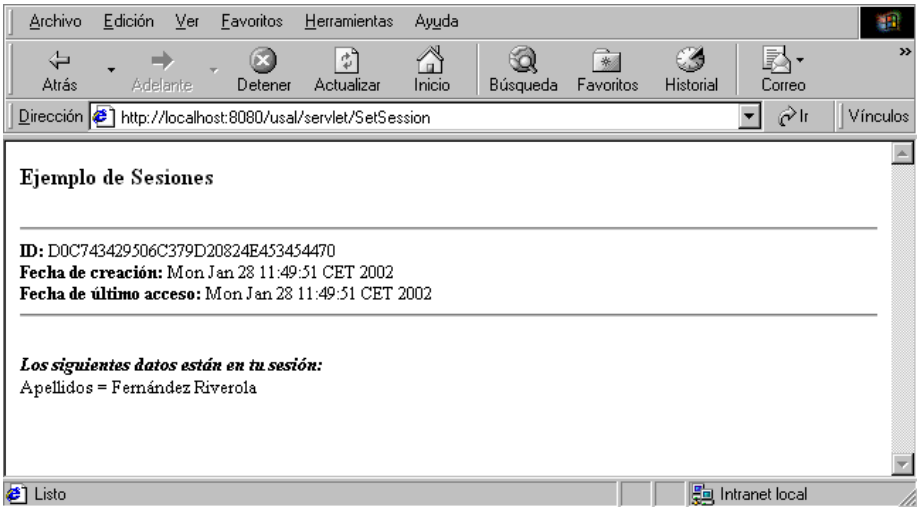

Figura 9: Resultado de ejecución del servlet *SetSession.class*

Reescritura de URL's

A pesar de que la mayoría de los navegadores más extendidos soportan las cookies en la actualidad, para poder emplear sesiones con clientes que, o bien no soportan cookies o bien las rechazan, debe utilizarse la reescritura de URL's.

Para emplear está técnica lo que se hace es incluir el código identificativo de la sesión (*sessionId*) en la URL de la petición. Los métodos que se encargan de reescribir la URL si fuera necesario son *encodeURL()* y *encodeRedirectURL()*. El primero de ellos lee un *String* que representa una URL y si fuera necesario la reescribe añadiendo el identificativo de la sesión, dejándolo inalterado en caso contrario. El segundo realiza lo mismo sólo que con URL's de redirección, es decir, permite reenviar la petición del cliente a otra URL.

Si en el ejemplo anterior incluimos reescritura de URL's quedaría como sigue:

```
// ... continuación del código anterior

        // Visualizar objetos contenidos en la sesion actual
        Enumeration e = session.getAttributeNames();
        int contValores = 1;
        while (e.hasMoreElements())
        {
            ++contValores;
            String nombre = (String)e.nextElement();
            String valor = session.getAttribute(nombre).toString();
            out.println(nombre + " = " + valor + "<BR>");
        }

        out.print("<p><a href=\"");
        out.print(respuesta.encodeURL( "SetSession?NombreDato=nombre" +
                                                contValores + "&ValorDato=valor" + contValores));
        out.println("\" >URL encoded </a>");
    }
```

Manejo de códigos de Estado en HTTP

Cuando un servidor web responde a una petición de un navegador u otro cliente web, la respuesta consiste típicamente en una línea de estado, algunas cabeceras de respuesta, una línea en blanco, y el documento. Aquí tenemos un ejemplo mínimo:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hola mundo
```

La línea de estado consta de la versión HTTP y un entero que se interpreta como código de estado, además de un mensaje muy corto que corresponde a dicho código de estado. En la mayoría de los casos, todas las cabeceras son opcionales excepto *Content-Type*, que especifica el tipo MIME del documento que sigue. Aunque muchas respuestas contienen un documento, algunas no lo tienen. Por ejemplo, las respuestas a peticiones HEAD nunca deberían incluir un documento, y hay una gran variedad de códigos de estado que esencialmente indican fallos, y o no incluyen un documento o sólo incluyen un pequeño "mensaje de error de documento".

Los servlets pueden realizar una variedad de tareas manipulando la línea de estado y las cabeceras de respuesta. Por ejemplo, reenviar al usuario a otros sitios web, indicar que el documento adjunto es una imagen, un fichero Acrobat o un fichero HTML, decirle al usuario que se requiere una password para acceder al documento, ..., etc.

La forma de hacer esto es mediante el método *setStatus()* de *HttpServletResponse*. Este método recibe un dato de tipo *int* (el código de estado) como argumento, pero en vez de usar los números explícitamente, es más claro y legible usar las constantes definidas en *HttpServletResponse*. El nombre de cada constante está derivado del mensaje estándar HTTP 1.1 para cada constante, todo en mayúsculas con un prefijo SC (por *Status Code*) y subrayados en vez de espacios. Así, como el mensaje para 404 es *Not Found*, la constante equivalente en *HttpServletResponse* es *SC_NOT_FOUND*. Sin embargo, hay dos excepciones: El código 302 se deriva del mensaje HTTP 1.0, y la constante para el código 307 tampoco existe.

Seleccionar el código de estado no significa que no se necesite devolver un documento. Por ejemplo, aunque la mayoría de los servidores generarán un pequeño mensaje *File Not Found* para respuestas 404, un servlet podría personalizar esta respuesta. Sin embargo, se necesita llamar a *setStatus()* antes de enviar el contenido mediante *PrintWriter*.

Aunque el método general de selección del código de estado es simplemente llamar a *setStatus(int)*, hay dos casos comunes para los que se proporciona un método atajo en *HttpServletResponse*: El método *sendError()*, que genera una respuesta 404 junto con un mensaje corto formateado dentro de un documento HTML, y el método *sendRedirect()*, que genera una respuesta 302 junto con una cabecera Location indicando la URL del nuevo documento.

Códigos de Estado HTTP 1.1 y sus significados

La Tabla 4 muestra una lista con los códigos de estado disponibles en HTTP 1.1 junto con sus mensajes asociados y su interpretación. Se debería tener cuidado al utilizar los códigos de estado que están disponibles sólo en HTTP 1.1, ya que muchos navegadores sólo soportan HTTP 1.0. Si es necesario utilizar códigos de estado específicos para HTTP 1.1, en la mayoría de los casos se deberá comprobar explícitamente la versión HTTP de la petición (mediante el método *getProtocol()* de *HttpServletRequest*) o reservarlo para situaciones donde no importe el significado de la cabecera HTTP 1.0.

Código de Estado	Mensaje Asociado	Significado
100	Continue	Continúa con petición parcial (nuevo en HTTP 1.1).
101	Switching Protocols	El servidor cumplirá con la cabecera *Upgrade* y cambiará a un protocolo diferente (nuevo en HTTP 1.1).
200	OK	Todo está bien. Los documentos enviados por peticiones GET y POST. Este es el valor por defecto para los servlets a menos que se utilice *setStatus()*.
201	Created	El servidor creó un documento. La cabecera *Location* indica la URL.
202	Accepted	La petición se está realizando, el proceso no se ha completado.
203	Non-Authoritative Information	El documento está siendo devuelto normalmente, pero algunas cabeceras de respuesta podrían ser incorrectas porque se está usando una copia del documento (nuevo en HTTP 1.1).
204	No Content	No hay un documento nuevo. El navegador continua mostrando el documento anterior. Esto es útil si el usuario recarga periódicamente una página y existe la posibilidad de determinar que la página anterior ya está actualizada. Sin embargo, esto no funciona para páginas que se recargan automáticamente mediante cabeceras de respuesta Refresh o su equivalente <META HTTP-EQUIV="Refresh" ...>, ya que al devolver este código de estado se pararán futuras recargas.
205	Reset Content	No hay documento nuevo, pero el navegador debería resetear el documento. Usado para forzar al navegador a borrar los contenidos de los campos de un formulario CGI (nuevo en HTTP 1.1).
206	Partial Content	El cliente envía una petición parcial con una cabecera *Range*, y el servidor la ha completado (nuevo en HTTP 1.1).
300	Multiple Choices	El documento pedido se puede encontrar en varios sitios. Serán listados en el documento devuelto. Si el servidor tiene una opción preferida, debería listarse en la cabecera de respuesta *Location*.
301	Moved Permanently	El documento pedido está en algún lugar, y la URL se da en la cabecera de respuesta *Location*. Los navegadores deberían seguir automáticamente el enlace a la nueva URL.
302	Found	Similar a 301, excepto que la nueva URL debería ser interpretada como reemplazada temporalmente, no permanentemente. El mensaje era *Moved Temporarily* en HTTP 1.0, y la constante en *HttpServletResponse* es *SC_MOVED_TEMPORARILY*, no *SC_FOUND*. Cabecera muy útil, ya que los navegadores siguen automáticamente el enlace a la

		nueva URL. Este código de estado es tan útil que hay un método especial para él: *sendRedirect()*. Usar *sendRedirect(url)* tiene un par de ventajas sobre hacer *setStatus(SC_MOVED_TEMPORARILY)* y *setHeader("Location", url)*. Primero, es más fácil. Segundo, con *sendRedirect()*, el servlet construye automáticamente una página que contiene el enlace (para mostrar a los viejos navegadores que no siguen las redirecciones automáticamente). Finalmente, *sendRedirect()* puede manejar URL's relativas, que se traducen automáticamente a absolutas. Este código de estado es usado algunas veces de forma intercambiada con 301. Por ejemplo, si erróneamente se pide *http://host/~user* (olvidando la última barra), algunos servidores enviarán 301 y otros 302. Técnicamente, se supone que los navegadores siguen automáticamente la redirección si la petición original era GET. Ver cabecera 307 para más detalles.
303	See Other	Igual que 301/302, excepto que si la petición original era POST, el documento redirigido (dado en la cabecera *Location*) debería ser recuperado mediante GET (nuevo en HTTP 1.1).
304	Not Modified	El cliente tiene un documento en la caché y realiza una petición condicional (normalmente suministrando una cabecera *If-Modified-Since* indicando que sólo quiere documentos posteriores a la fecha especificada). El servidor le especifica al cliente que el viejo documento de su caché todavía está en uso.
305	Use Proxy	El documento pedido debería recuperarse mediante el proxy listado en la cabecera *Location* (nuevo en HTTP 1.1).
307	Temporary Redirect	Es idéntica a 302 (*Found o Temporarily Moved*). Fue añadido a HTTP 1.1 ya que muchos navegadores siguen erróneamente la redirección de una respuesta 302 incluso si el mensaje original fue un POST, y sólo se debe seguir la redirección de una petición POST en respuestas 303. Esta respuesta es algo ambigua: sigue el redireccionamiento para peticiones GET y POST en el caso de respuestas 303, y en el caso de respuesta 307 sólo sigue la redirección de peticiones GET. Nota: por alguna razón no existe una constante en *HttpServletResponse* que corresponda con este código de estado (nuevo en HTTP 1.1).
400	Bad Request	Mala sintaxis de la petición.
401	Unauthorized	El cliente intenta acceder a una página protegida por password sin la autorización apropiada. La respuesta debería incluir una cabecera *WWW-Authenticate* que el navegador debería usar para mostrar la caja de diálogo usuario/password, que viene de vuelta con la cabecera *Authorization*.
403	Forbidden	El recurso no está disponible, sin importar la autorización. Normalmente indica la falta permisos de fichero o directorios en el servidor.
404	Not Found	No se pudo encontrar el recurso en esa dirección. La respuesta estándar es *no such page*. El método *sendError(message)* de *HttpServletResponse* está disponible para este código de error. La ventaja de *sendError()* sobre *setStatus()* es que con el primero el servidor genera automáticamente una página que muestra un mensaje de error.
405	Method Not Allowed	El método de la petición (GET, POST, HEAD, DELETE, PUT, TRACE, ..., etc.) no estaba permitido para este recurso particular (nuevo en HTTP 1.1).
406	Not Acceptable	El recurso indicado genera un tipo MIME incompatible con el especificado por el cliente mediante su cabecera Accept (nuevo en HTTP 1.1).
407	Proxy Authentication Required	Similar a 401, pero el servidor proxy debería devolver una cabecera *Proxy-Authenticate* (nuevo en HTTP 1.1).
408	Request Timeout	El cliente tarda demasiado en enviar la petición (nuevo en HTTP 1.1).
409	Conflict	Usualmente asociado con peticiones PUT. Usado para situaciones como la carga de una versión incorrecta de un fichero (nuevo en HTTP 1.1).
410	Gone	El documento se ha ido. No se conoce la dirección de reenvío. Difiere de la 404 en que se sabe que el documento se ha ido permanentemente, no sólo no está disponible por alguna razón desconocida como en 404 (nuevo en HTTP 1.1).
411	Length Required	El servidor no puede procesar la petición a menos que el cliente envíe una cabecera *Content-Length* (nuevo en http 1.1).
412	Precondition Failed	Alguna condición previa especificada en la petición era falsa (nuevo en HTTP 1.1).
413	Request Entity Too Large	El documento pedido es mayor que lo que el servidor puede manejar ahora. Si el servidor cree que puede manejarlo más tarde, debería incluir una cabecera *Retry-After* (nuevo en HTTP 1.1).
414	Request URI Too	La URI es demasiado larga (nuevo en HTTP 1.1).

	Long	
415	Unsupported Media Type	La petición está en un formato desconocido (nuevo en HTTP 1.1).
416	Request Range Not Satisfiable	El cliente incluyó una cabecera Range no satisfactoria en la petición (nuevo en HTTP 1.1).
417	Expectation Failed	No se puede conseguir el valor de la cabecera *Expect* (nuevo en HTTP 1.1).
500	Internal Server Error	Mensaje genérico *server is confused*. Normalmente es el resultado de programas CGI o servlets que se quedan colgados o retornan cabeceras mal formateadas.
501	Not Implemented	El servidor no soporta la funcionalidad de rellenar peticiones. Usado, por ejemplo, cuando el cliente envía comandos como PUT que el servidor no soporta.
502	Bad Gateway	Usado por servidores que actúan como proxies o gateways. Indica que el servidor inicial obtuvo una respuesta errónea desde el servidor remoto.
503	Service Unavailable	El servidor no puede responder debido a mantenimiento o sobrecarga. Por ejemplo, un servlet podría devolver esta cabecera si algún pool de threads o de conexiones con bases de datos están llenos. El servidor puede suministrar una cabecera *Retry-After*.
504	Gateway Timeout	Usado por servidores que actúan como proxies o gateways. Indica que el servidor inicial no obtuvo una respuesta a tiempo del servidor remoto (nuevo en HTTP 1.1).
505	HTTP Version Not Supported	El servidor no soporta la versión de HTTP indicada en la línea de petición (nuevo en HTTP 1.1).

Tabla 4: Códigos de estado del protocolo HTTP 1.1

1.6 Ejemplo: Motor de búsqueda

A continuación se muestra un ejemplo que hace uso de los dos códigos de estado más comunes distintos de 200: 302 y 404. El código 302 se selecciona mediante el método *sendRedirect()*, mientras que el 404 se selecciona mediante *sendError()*. En esta aplicación, se muestra un formulario HTML que permite al usuario elegir una cadena de búsqueda, el número de los resultados por página, y el motor de búsqueda a utilizar. Cuando se envía el formulario, el servlet extrae estos tres parámetros, construye una URL con los parámetros embebidos en una forma apropiada para el motor de búsqueda seleccionado, y redirige al usuario a esa dirección. Si el usuario falla al elegir el motor de búsqueda o envía un nombre de motor de búsqueda no conocido, se devuelve una página de error 404 diciendo que no hay motor de búsqueda o que no se conoce.

Los listados siguientes muestran el código fuente de las clases *SearchSpec* y *SearchEngines* respectivamente:

```
package hall;

class SearchSpec
{
          private String name, baseURL, numResultsSuffix, extra;

          private static SearchSpec[] commonSpecs =
          {
               new SearchSpec( "google",
                               "http://www.google.com/search?q=",
                               "&num=",
                               ""),
               new SearchSpec( "infoseek",
                               "http://srch.overture.com/d/search/p/go/?Partner=go_home&Keywords=",
                               "&nh=",
                               ""),
               new SearchSpec( "lycos",
                               "http://lycospro.lycos.com/cgi-bin/pursuit?query=",
                               "&maxhits=",
                               ""),
               new SearchSpec( "hotbot",
                               "http://www.hotbot.lycos.es/result.html?query=",
                               "&numresult_field=",
               "&bool=all&z=11111212112111111211&languagefield=any&description_field=full&uk=ww&hs=s")
                    };

          public SearchSpec(String name, String baseURL, String numResultsSuffix, String extra)
          {
               this.name = name;
```

```
                this.baseURL = baseURL;
                this.numResultsSuffix = numResultsSuffix;
                this.extra = extra;
          }

          public String makeURL(String searchString, String numResults)
          {
                return(baseURL + searchString + numResultsSuffix + numResults + extra);
          }

          public String getName()
          {
                return(name);
          }

          public static SearchSpec[] getCommonSpecs()
          {
                return(commonSpecs);
          }
}
```

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;

public class SearchEngines extends HttpServlet
{
          public void doGet(HttpServletRequest peticion, HttpServletResponse respuesta) throws ServletException, IOException
          {
                String searchString = URLEncoder.encode(peticion.getParameter("searchString"));
                String numResults = peticion.getParameter("numResults");
                String searchEngine = peticion.getParameter("searchEngine");

                SearchSpec[] commonSpecs = SearchSpec.getCommonSpecs();
                for(int i=0; i<commonSpecs.length; i++)
                {
                     SearchSpec searchSpec = commonSpecs[i];
                     if (searchSpec.getName().equals(searchEngine))
                     {
                        String url = respuesta.encodeURL(searchSpec.makeURL(searchString, numResults));
                        respuesta.sendRedirect(url);
                        return;
                     }
                }
                respuesta.sendError(respuesta.SC_NOT_FOUND,
                                    "Motor de búsqueda especificado no reconocido.");
          }

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

La Figura 10 muestra el aspecto de la página *SearchEngines.html* creada para invocar al servlet *SearchEngine.class*.

Figura 10: Página web creada para realizar una búsqueda en Internet

1.7 Subida de ficheros al servidor

Al desarrollar una aplicación web, es posible que surja la necesidad de crear un sistema de administración para simplificar las labores de mantenimiento. Esta administración, posiblemente estará formada por unos cuantos formularios que interactuarán con una base de datos. Pero a veces, también resulta necesario poder subir ficheros al servidor.

Para subir un fichero al servidor, se debe crear un formulario especial via POST e indicando que es una subida multi-parte tal como se muestra a continuación:

```
<HTML>
<HEAD>
<TITLE></TITLE>
</HEAD>
<BODY>
<center>
<form method="POST" enctype='multipart/form-data' action="UploadFichero">
Por favor, seleccione el trayecto del fichero a cargar
<br><input type="file" name="fichero">
<input type="submit">
</form>
</center>
</BODY>
</HTML>
```

En el formulario anterior, se puede ver una invocación a un servlet llamado uploadFichero. El aspecto del formulario se puede ver en la Figura 11.

Figura 11: Aspecto del formulario Upload.

A continuación, se debe que crear un servlet que sea capaz de procesar la respuesta. Para facilitar la labor se pueden emplear las clases que porciona un subproyecto en Apache llamado fileupload (http://jakarta.apache.org/commons/fileupload/). Se descarga el fichero jar correspondiente y se incluye en el classpath. Para el funcionamiento correcto de esta clase será necesario incluir,también incluir en el classpath la implementación de Yakarta commons IO que se puede descargar de http://jakarta.apache.org/commons/io/.

A continuación se procede a codificar el servlet empleando la clase DiskFileUpload del proyecto fileupload que facilita el procesarmiento de la petición. El código del servlet se ha construido como se especifica a continuación:

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

import org.apache.commons.fileupload.*;
import java.util.*;
/**
 * Subir ficheros al servidor
 * @author José Ramón Méndez Reboredo
 * @version 1.0
 */
public class UploadFichero extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet</title></head><body>");

        procesaFicheros(request,out);

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

    public boolean procesaFicheros(HttpServletRequest req, PrintWriter out ) {
        try {
            // construimos el objeto que es capaz de parsear la perición
            DiskFileUpload fu = new DiskFileUpload();

            // maximo numero de bytes
            fu.setSizeMax(1024*1024*8); // 8 MB

            // tamaño por encima del cual los ficheros son escritos directamente en disco
            fu.setSizeThreshold(4096);

            // directorio en el que se escribirán los ficheros con tamaño
            // superior al soportado en memoria
            fu.setRepositoryPath("/tmp");

            // ordenamos procesar los ficheros
            List fileItems = fu.parseRequest(req);

            if(fileItems == null)
            {
                return false;
            }

            out.print("<br>El número de ficheros subidos es: " +  fileItems.size());

            // Iteramos por cada fichero
              Iterator i = fileItems.iterator();
            FileItem actual = null;

            while (i.hasNext())
            {
                actual = (FileItem)i.next();
                String fileName = actual.getName();
                out.println("<br>Nos han subido el fichero" + fileName);

                // construimos un objeto file para recuperar el trayecto completo
                File fichero = new File(fileName);

                // nos quedamos solo con el nombre y descartamos el path
                fichero = new  File("c:\\ficherossubidos\\" + fichero.getName());

                // escribimos el fichero colgando del nuevo path
                actual.write(fichero);
            }

        }
        catch(Exception e) {
            return false;
        }

        return true;
    }

    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

1.8 Generar imágenes desde servlets

Muchas veces resulta necesario generar imágenes de forma dinámica como gráficas o direcciones de correo electrónico. En este sentido, es necesario recurrir al API Java 2D mediante la creación de una ventana de Swing.

A nivel general, los pasos a seguir son los siguientes:

1- Generar una imagen con la clase BufferedImage para aprovechar la API Java 2D y sus capacidades gráficas:

```
Graphics contextoGrafico=null;
int w = 200;
int h = 50;

BufferedImage image = new BufferedImage(w, h, BufferedImage.TYPE_3BYTE_BGR);
```

2- Crear una imagen que represente el fondo de la ventana creada en el paso anterior:

```
contextoGrafico = image.getGraphics();
```

3- Pintar según lo requerido

```
contextoGrafico.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 20));
contextoGrafico.draw3DRect(0,0,w-1,h-1,true);
Date ahora = new Date();
SimpleDateFormat formateador = new SimpleDateFormat("dd-MM-yyyy HH:mm");
contextoGrafico.drawString(formateador.format(ahora), 20, 30);
```

4- Cambiar el tipo de respuesta del servlet

```
res.setContentType("image/gif");
```

5- Codificar nuestra imagen como un fichero GIF y escribirlo en el buffer de salida

```
GifEncoder encoder = new GifEncoder(image, out);
encoder.encode();

out.flush();
out.close();
```

Para codificar la imagen (Objeto Image), se puede usar una clase que se puede descargar gratuitamente de http://www.acme.com/java/software/Acme.JPM.Encoders.GifEncoder.html. Esta clase es GifEncoder y permite codificar imágenes en formato GIF. De la misma forma, se podría emplear las clases JPEGCodec, JPEGEncodeParam y JPEGImageEncoder del paquete com.sun.image.codec.jpeg que se distribuye con el propio JavaSE. Es importante recordar que si se escoge esta opción el tipo de respuesta del servlet será `"image/jpeg"` y que no es necesario descargar ningún software adicional.

```
JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(image);
param.setQuality(1.0f, false);
encoder.setJPEGEncodeParam(param);
encoder.encode(image);

out.flush();
out.close();
```

A continuación se muestra el código fuente del servlet creado para construir la imagen así como una página web que permite su prueba.

```
import java.awt.*;
import java.awt.image.*;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
```

```
import java.text.*;

import com.sun.image.codec.jpeg.*;

/**
  * Genera una imagen para colocar en la web
  * @author José Ramón Méndez Reboredo
  * @version 1.0
  */
public class SimpleJPGServlet extends HttpServlet {

  /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
    * @param request servlet request
    * @param response servlet response
    */
  protected void processRequest(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, java.io.IOException {

       ServletOutputStream bufferSalida = res.getOutputStream();

       Graphics contextoGrafico=null;

       try {
            // Construir un contexto de gráficos mediante una
                        //buffered image
            int w = 200;
            int h = 50;

            BufferedImage image = new BufferedImage(w, h, BufferedImage.TYPE_3BYTE_BGR);

                        contextoGrafico=image.getGraphics();

            contextoGrafico.setFont(new Font("Arial", Font.BOLD | Font.ITALIC, 20));

            contextoGrafico.draw3DRect(0,0,w-1,h-1,true);

            Date ahora = new Date();
            SimpleDateFormat formateador = new SimpleDateFormat("dd-MM-yyyy HH:mm");

            contextoGrafico.drawString(formateador.format(ahora), 20, 30);

            // Encode the off screen graphics into a GIF and send it to the client
            res.setContentType("image/jpeg");

            JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(bufferSalida);
            JPEGEncodeParam param = encoder.getDefaultJPEGEncodeParam(image);
            param.setQuality(1.0f, false);
            encoder.setJPEGEncodeParam(param);
            encoder.encode(image);

            bufferSalida.flush();
            bufferSalida.close();

       } finally {
            if (contextoGrafico != null) contextoGrafico.dispose();
       }
  }

  /** Handles the HTTP <code>GET</code> method.
    * @param request servlet request
    * @param response servlet response
    */
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
     processRequest(request, response);
  }

  /** Handles the HTTP <code>POST</code> method.
    * @param request servlet request
    * @param response servlet response
    */
  protected void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, java.io.IOException {
     processRequest(request, response);
  }
}
```

Por otro lado el código de la página web que invoca al servlet:

```
<html>
<head>
    <meta http-equiv="Content-Language" content="es">
    <title>Mostrar la imagen generada por un Servlet</title>
</head>
<body>
    <center>
        <h2>Bienvenidos a nuestro servidor</h2>
        <p>Os vamos a mostrar los fácil que es invocar al servlet</p>
        <img src="SimpleJPGServlet">
    </center>
</body>
</html>
```

Finalmente en la se puede ver el resultado del servlet generador de imágenes.

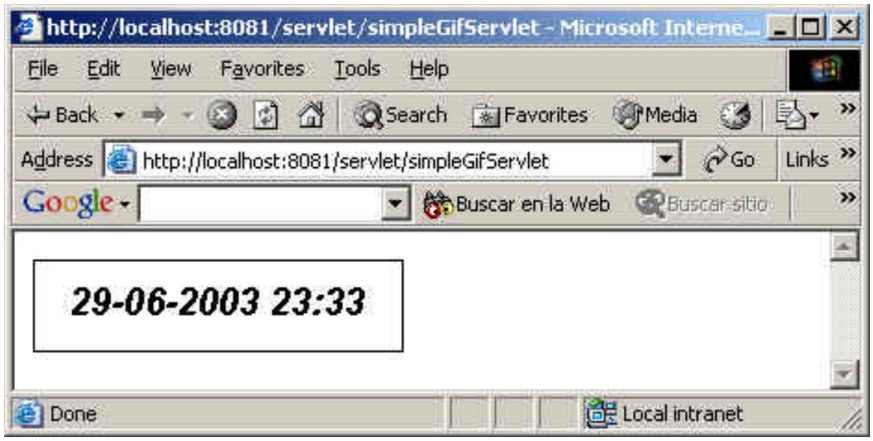

Figura 12: Resultado de generación de imágenes.

2 JSP

2.1 Introducción

JSP es una tecnología basada en Java que simplifica el proceso de desarrollo de sitios web dinámicos. Las *Java Server Pages* son ficheros de texto (normalmente con extensión *.jsp*) que sustituyen a las páginas HTML tradicionales. Los ficheros JSP contienen etiquetas HTML y código embebido que permite al diseñador de la página web acceder a datos desde código Java que se ejecuta en el servidor.

Cuando la página JSP es requerida por un navegador a través de una petición HTTP, las etiquetas HTML permanecen inalterables, mientras que el código que contiene dicha página es ejecutado en el servidor, generando contenido dinámico que se combina con las etiquetas HTML antes de ser enviado al cliente. Este modo de operar implica una separación entre los aspectos de presentación de la página y la lógica de programación contenida en el código.

Es una tecnología híbrida basada en *template systems*. Al igual que ASP, SSJS y PHP puede incorporar scripts para añadir código Java directamente a las páginas *.jsp*, pero también implementa, al estilo de ColdFusion, un conjunto de etiquetas que interaccionan con los objetos Java del servidor, sin la necesidad de aparezca código fuente en la página.

JSP se implementa utilizando la tecnología Servlet. Cuando un servidor web recibe una petición de una página *.jsp*, la redirecciona a un proceso especial dedicado a manejar la ejecución de servlets (*servlet container*). En el contexto de JSP, este proceso se llama *JSP container*.

Beneficios que aporta JSP

Los beneficios ofrecidos por JSP, como alternativa a la generación de contenido dinámico para la web, se resumen a continuación:

Mejoras en el rendimiento

- Utilización de procesos ligeros (hilos Java) para el manejo de las peticiones.
- Manejo de múltiples peticiones sobre una página *.jsp* en un instante dado.
- El contenedor servlet puede ser ejecutado como parte del servidor web.
- Facilidad para compartir recursos entre peticiones (hilos con el mismo padre: *servlet container*).

Soporte de componentes reutilizables

- Creación, utilización y modificación de JavaBeans del servidor.
- Los JavaBeans utilizados en páginas *.jsp* pueden ser utilizados en servlets, applets o aplicaciones Java.

Separación entre código de presentación y código de implementación

- Cambios realizados en el código HTML relativos a cómo son mostrados los datos, no interfieren en la lógica de programación y viceversa (ver Figura 3.32).

División del trabajo

- Los diseñadores de páginas pueden centrarse en el código HTML y los programadores en la lógica del programa.
- Los desarrollos pueden hacerse independientemente.
- Las frecuentes modificaciones de una página se realizan más eficientemente.

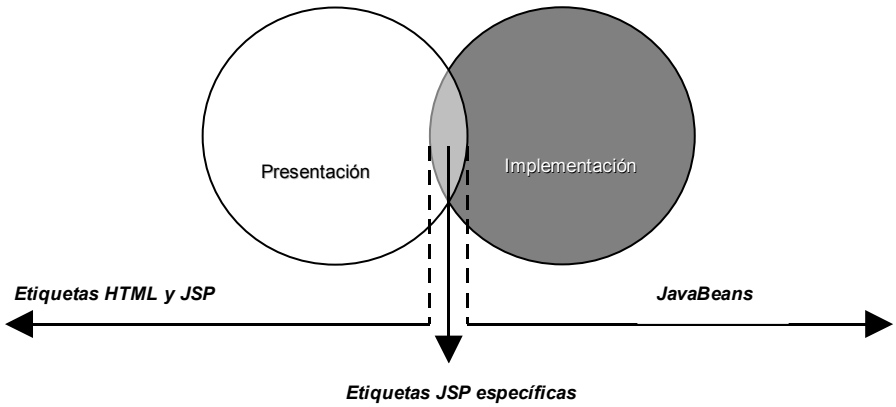

Figura 13: Separación entre código de presentación y código de implementación en JSP

¿Cómo trabaja JSP?

Para poder utilizar esta tecnología es necesario un servidor web que de soporte a

páginas *.html*, y código que implemente un contenedor JSP donde ejecutar las etiquetas JSP. Existen servidores web que incorporan dicha capacidad dentro de su código (*Netscape Enterprise* y *Aplication Server 5.0*), así como servidores escritos íntegramente en Java (*Java Web Server* de Sun y *Jigsaw* del W3consortium) que dan soporte a esta tecnología directamente.

Sin embargo, la mayoría de servidores web, están escritos en lenguajes de programación tradicional, compilados en código nativo por razones de eficiencia. Para estos servidores es necesario añadir código suplementario que implemente el contenedor JSP. Para ello se han proporcionado API's del servidor para poder extender su funcionalidad, así Netscape proporciona NSAPI (*Netscape Server Application Programming Interface*), y Microsoft proporciona ISAPI (*Internet Server Application Programming Interface*). Mediante estas API's es posible desarrollar herramientas de soporte para JSP.

Para el servidor Apache, se dispone de módulos. En versiones antiguas del servidor era necesario recompilar el código del servidor para soportar un nuevo módulo, actualmente son cargados dinámicamente basándose en ficheros de configuración.

Una vez que el contenedor JSP ha sido instalado y configurado, los ficheros *.jsp* se tratan igual que los ficheros *.html*, situándolos en cualquier lugar de la jerarquía de directorios. Cualquier clase Java que se utilice en un fichero *.jsp*, debe estar disponible en la variable *CLASSPATH* del contenedor JSP.

Aunque la especificación JSP no presupone nada sobre la implementación que da soporte a esta tecnología, la mayoría de las implementaciones disponibles están basadas en servlets.

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

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

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

de que se haya producido la primera petición.

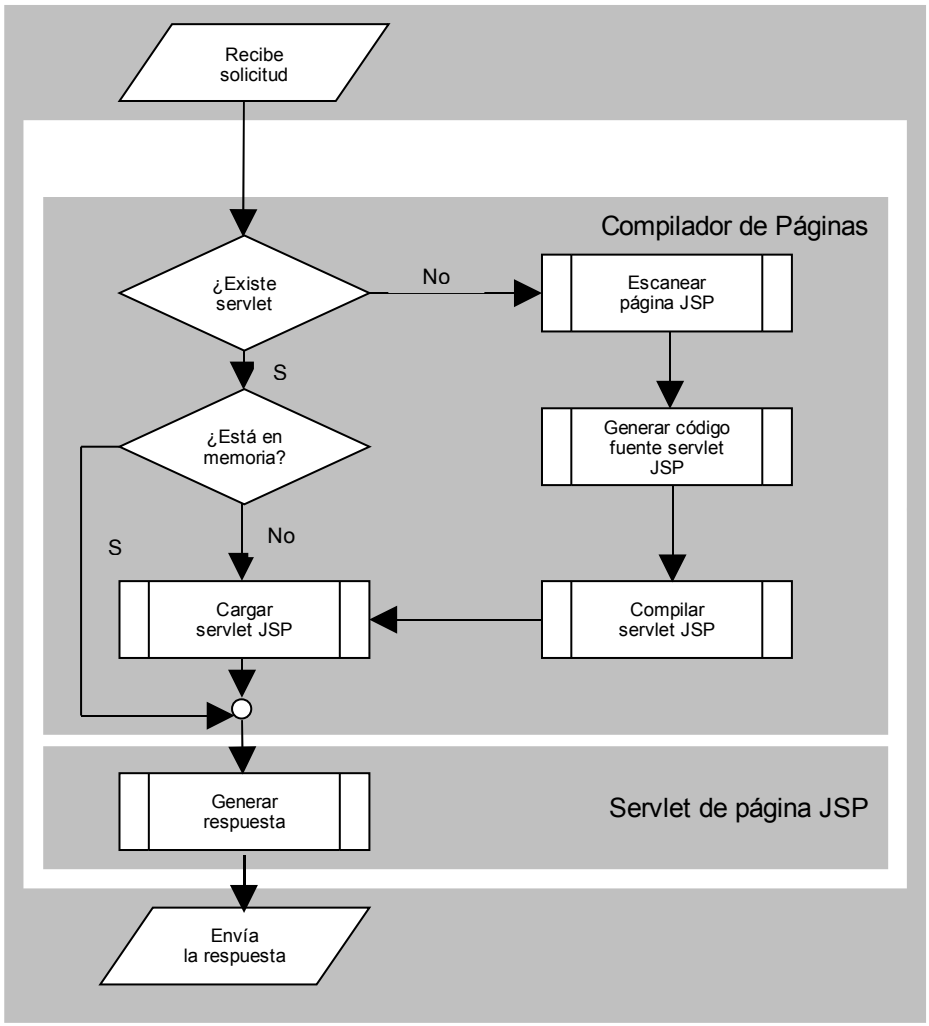

Figura 14: Esquema de ejecución de una página JSP

Salida con buffer

El protocolo HTTP permite enviar al cliente información adicional en forma de cabeceras (información sobre el cuerpo de la respuesta, control del cache, establecimiento de cookies, ..., etc). Esta información debe enviarse antes que el cuerpo de la página HTML que el cliente debe visualizar, pero el código JSP que podría encargarse de generar estas cabeceras se encuentra inmerso en el cuerpo del documento *.jsp*.

La especificación JSP requiere que todas las implementaciones de JSP soporten salida con buffer. Esto supone que el contenido generado por una página *.jsp* no se envía directamente al navegador cuando está siendo generado, sino que el contenido de la página a enviar se guarda en un buffer de salida. Este buffer sólo se vacía cuando la página *.jsp* completa ha sido procesada. Esto facilita la creación de cabeceras desde código JSP inmerso en una página, o incluso permite descartar el envío de la página

por la detección cualquier error en tiempo de ejecución.

El tamaño por defecto del buffer que maneja JSP es de 8K, que se supone más que suficiente para la mayoría de las páginas con contenido dinámico, aunque es posible incrementar el tamaño o incluso desactivar el buffer.

Manejo de sesiones

Una característica del protocolo HTTP es que carece de estado, lo que significa que el servidor no puede mantener información sobre los clientes que se conectan a él una y otra vez. Cuando se recibe una petición de un cliente y se envía la respuesta, el servidor olvida todo sobre el ordenador que generó la petición, y la siguiente vez que el mismo cliente se conecte será tratado como si nunca antes se hubiera conectado. Esto es un problema cuando se desea que el usuario se autentifique y rellene un conjunto de formularios. ¿Cómo saber que se trata del mismo cliente y no otro totalmente distinto el que se conecta a una página en concreto?.

El proceso de tratar de mantener el estado a través de múltiples peticiones HTTP se conoce con el nombre de manejo de sesión (*session management*). La idea es que todas las peticiones que genere un cliente durante un período de tiempo dado, pertenecen a la misma sesión interactiva.

JSP incluye soporte para el manejo de sesiones, dado que se implementa utilizando la API Servlet. Los servlets pueden utilizar *cookies* o *reescritura de URL's* para implementar el manejo de sesiones, pero los detalles están ocultos al nivel de JSP.

2.2 Sintaxis del lenguaje

JSP proporciona 4 tipos principales de etiquetas:

- **Directivas**: conjunto de etiquetas que contienen instrucciones para el contenedor JSP con información acerca de la página en la que se encuentran. No producen ninguna salida visible, pero afectan a las propiedades globales de la página *.jsp* que tienen influencia en la generación del servlet correspondiente.

- **Elementos de Script**: se utilizan para encapsular código Java que será insertado en el servlet correspondiente a la página *.jsp*.

- **Comentarios**: se utilizan para añadir comentarios a la página *.jsp*. JSP soporta múltiples estilos de comentarios, incluyendo aquellos que hacen que los comentarios aparezcan en la página HTML generada como respuesta a la petición del cliente.

- **Acciones**: soportan diferentes comportamientos. Pueden transferir el control entre páginas, applets e interaccionar con componentes JavaBeans del lado del servidor. Se pueden programar etiquetas JSP personalizadas para extender las funcionalidades del lenguaje en forma de etiquetas de este tipo.

Directivas

Se usan para enviar información especial sobre la página al contenedor JSP (especificar el lenguaje de script, incluir contenido de otra página, o indicar que la página usa una determinada librería de etiquetas). Las directivas JSP van encerradas entre **<%@** y **%>**.

page

<%@ page atributo1=”valor1” atributo2=”valor2” atributo3=... **%>**

o también en formato XML:

<jsp:directive.page
atributo1=”valor1”
atributo2=”valor2”
atributo3=... **/>**

Los posibles valores de los atributos se muestran en la Tabla 5.

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

Tabla 5: Valores por defecto de los atributos de la directiva page

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

Un ejemplo de una página *.jsp* sencilla se muestra a continuación.

```
<HTML>
<HEAD>
        <TITLE>La primera página con JSP</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">

<%@ page info=”Probando JSP...” %>
<%@ page language=”java” %>
```

```
<%@ page contentType="text/plain; charset=ISO-8859-1" %>
<%@ page import="java.util.*, java.awt.*" %>
<%@ page session="false" %>
<%@ page buffer="12kb" %>
<%@ page autoFlush="true" %>
<%@ page isThreadSafe="false" %>
<%@ page errorPage="/webdev/misc/error.jsp" %>
<%@ page isErrorPage="false" %>

Hola mundo!
</BODY>
</HTML>
```

include

<%@ include *file*="localURL" **%>**

Permite incluir el contenido de otra página *.html* o *.jsp*. El fichero es identificado mediante una URL y la directiva tiene el efecto de remplazarse a sí misma con el contenido del archivo especificado. Puede ser utilizada para incluir cabeceras y pies de páginas comunes a un desarrollo dado.

taglib

<%@ taglib *uri*="tagLibraryURI" *prefix*="tagPrefix" **%>**

Esta directiva se utiliza para notificar al contenedor JSP que la página utilizará una o más librerías de etiquetas personalizadas. Una librería de etiquetas es una colección de etiquetas personalizadas, que pueden ser utilizadas para extender la funcionalidad básica de JSP.

El valor del atributo *uri* indica la localizacion del TLD (*Tag Library Descriptor*), mientras que el atributo *prefix* especifica el identificador del espacio de nombres en XML que será anexado a todas la etiquetas de esa librería que se utilicen en la página *.jsp*.

Por razones de seguridad, la especificación JSP permite utilizar solamente TLD's del sistema de archivos local.

Elementos de script

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

Las declaraciones permiten definir variables y métodos para una página. Los scriptlets son bloques de código preparados para ejecutarse y las expresiones son líneas simples de código cuya evaluación es insertada en la salida de la página en el lugar de la expresión original.

Declaraciones

Las declaraciones se usan para definir variables y métodos específicos para una página *.jsp*. Las variables y los métodos declarados son accesibles por cualquier otro elemento de script de la página, aunque estén codificados antes que la propia declaración. La sintaxis es:

<%! declaracion (es) **%>**

Algunos ejemplos se muestran a continuación:

```
<%! private int x = 0, y = 0; private String unidad="cm"; %>
```

```
<%! static public int contador = 0; %>
```

```
<%! public long factorial (long numero)
  {
         if (numero == 0) return 1;
         else return numero * factorial(numero - 1);
  } %>
```

Un uso importante de la declaración de métodos es el manejo de eventos relacionados con la inicialización y destrucción de la página *.jsp*. La inicialización ocurre la primera vez que el contenedor JSP recibe una petición de una página *.jsp* concreta. La destrucción ocurre cuando el contenedor JSP descarga la clase de memoria o cuando el contenedor JSP se reinicia.

Estos eventos se pueden controlar sobreescribiendo dos métodos que son llamados automáticamente: *jspInit()* y *jspDestroy()*.

```
<%! public void jspInit ()
    {
         // El código de inicialización iría aquí.
    }
    public void jspDestroy()
    {
         // El código de finalización iría aquí.
    }
%>
```

Obviamente su codificación es opcional.

Expresiones

<%= expresión **%>**

Las expresiones están ligadas a la generación de contenidos. Se pueden utilizar para imprimir valores de variables o resultados de la ejecución de algún método. Todo resultado de una expresión es convertido a un *String* antes de ser añadido a la salida de la página. Las expresiones no terminan con un punto y coma (;).

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

Scriptlets

<% scriptlet **%>**

Los scriptlets pueden contener cualquier tipo de sentencias Java, pudiendo dejar abiertos uno o más bloques de sentencias que deberán ser cerradas en etiquetas de scritplets posteriores.

Se pueden incluir sentencias condicionales *if-else if-else*,

```
<% if (numero < 0) { %>
        <p>No se puede calcular el factorial de un número negativo.</p>
<% } else if (numero > 20) { %>
        <p>Argumentos mayores que 20 causan un desbordamiento de pila.</p>
<% } else { %>
        <p align=center><%= x %>! = <% factorial (x) %></p>
<% } %>
```

Se pueden incluir sentencias de iteración: *for*, *while* y *do/while*,

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

En cuanto al manejo de excepciones, el comportamiento por defecto cuando surge un error durante la ejecución de una página *.jsp*, es el de mostrar un mensaje en la ventana del navegador. También se puede utilizar el atributo *errorPage* de la directiva *page* para especificar una página de error alternativa. La tercera opción es utilizar el mecanismo de manejo de excepciones de Java usando *scriptlets*.

El ejemplo mostrado anteriormente para calcular el factorial de un número dado, utilizando manejo de excepciones quedaría como sigue.

```
<!-- factorialestry.jsp -->

<HTML>
<HEAD>
                <TITLE>La segunda página con JSP</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">

<%! public long factorial (long numero) throws IllegalArgumentException
  {
                if ((numero < 0) || (numero > 20))
                     throw new IllegalArgumentException("Fuera de Rango");

                if (numero == 0) return 1;
                else return numero * factorial(numero - 1);
  } %>

<% try {
          long resultado = factorial (x); %>

          <p align=center> <%= x %>! = <%= resultado %></p>
<% } catch (IllegalArgumentException e) { %>
          <p>El argumento para calcular el factorial esta fuera de rango.</p>
<% } %>

</BODY>
</HTML>
```

Comentarios

En una página *.jsp*, pueden existir tres tipos de comentarios, aunque se dividen en dos grandes categorías: (i) los que son transmitidos al cliente como parte de la respuesta generada por el servidor y (ii) los que sólo son visibles en el código fuente de la página *.jsp*.

Comentarios de contenido

Este tipo de comentarios serán incluidos en la salida de la página *.jsp*, y por lo tanto serán visibles en el código fuente que le llegue al cliente. La sintaxis es:

<!-- comentario **-->**

Es el tipo estándar de comentario para HTML y XML. Se puede incluir contenido dinámico en el comentario (expresiones JSP), y el valor de la expresión aparecerá como parte del comentario en el cliente.

```
<!-- long en Java es de 64 bits, por lo tanto 20! = <%= factorial(20) %> es el límite superior -->
```

Comentarios JSP

Los comentarios JSP son independientes del tipo de contenido producido por la página y del lenguaje de script utilizado, Estos comentarios sólo pueden ser vistos examinando la página *.jsp* original. La sintaxis es:

<%-- comentario **--%>**

El cuerpo del comentario es ignorado por el contenedor JSP.

Comentarios del lenguaje de Script

Pueden ser introducidos dentro de los scriptlets o las expresiones JSP, utilizando los comentarios nativos de la sintaxis del lenguaje de script. La sintaxis es.

<% /* comentario */ **%>**

Estos comentarios no serán visibles para el cliente, pero sí aparecerán en el servlet correspondiente a la página *.jsp*.

Acciones

Las acciones permiten la transferencia de control entre páginas, soportan la especificación applets de Java de un modo independiente al navegador del cliente, capacitan a las páginas *.jsp*, para interaccionar con componentes JavaBeans residentes en el servidor y permiten la creación y uso de librerías de etiquetas personalizadas.

forward

<jsp:forward *page*=”localURL” **/>**

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

El valor del parámetro *page*, puede ser un documento estático, un CGI, un servelt u otra página JSP. Para añadir flexibilidad a este parámetro, se puede construir su valor como concatenación de valores de variables.

```
<jsp:forward page='<%= "message" + statusCode + ".html" %>' />
```

Puede resultar útil pasar pares parámetro/valor al destino de la etiqueta. Para ello se utiliza **<jsp:param** ... **/>**.

```
<jsp:forward page="factorial.jsp"
             <jsp:param name="numero" value="25" />
             ...
</jsp:forward>
```

include

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

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

El valor del atributo *flush*, determina si se debe vaciar el buffer de la página actual antes de incluir el contenido generado por el programa incluido.

Como en el caso anterior, se puede completar esta etiqueta con el uso de **<jsp:param>** para proporcionar información adicional al programa llamado.

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

Plug-in

La finalidad de esta etiqueta es generar código HTML específico para el navegador del cliente cuando se invocan applets que utilizan el *Plug-in* de Sun.

Beans

JSP proporciona tres acciones diferentes para interaccionar con JavaBeans del lado del servidor: **<jsp:useBean>**, **<jsp:setProperty>** y **<jsp:getProperty>**.

Objetos implícitos

El contenedor JSP exporta un número de objetos internos para su uso desde las páginas *.jsp*. Estos objetos son asignados automáticamente a variables que pueden ser accedidas desde elementos de script de JSP. La Tabla 6 muestra los objetos disponibles y la API a la que pertenecen.

Objeto	Clase o Interfaz	Descripción
page	javax.servlet.jsp.HttpJspPage	Instancia del servlet de la página .jsp.
config	javax.servlet.ServletConfig	Datos de configuración del servlet.
request	javax.servlet.http.HttpServletRequest	Petición. Parámetros incluidos.
response	javax.servlet.http.HttpServletResponse	Respuesta.
out	javax.servlet.jsp.JspWriter	Stream de salida para el contenido de la página.
session	javax.servlet.http.HttpSession	Datos específicos de la sesión de usuario.
application	javax.servlet.ServletContext	Datos compartidos por todas las páginas.
pageContext	javax.servlet.jsp.PageContext	Contexto para la ejecución de las páginas.
exception	java.lang.Throwable	Excepción o error no capturado.

Tabla 6: Objectos accesibles desde una página JSP

Los nueve objetos proporcionados por JSP se clasifican en cuatro categorías: (i) objetos relacionados con el servlet correspondiente a la página *.jsp*, (ii) objetos relacionados con la entrada y salida de la página, (iii) objetos que proporcionan información sobre el contexto y (iv) objetos para manejo de errores.

Estas cuatro categorías tienen la funcionalidad común de establecer y recuperar valores de atributos como mecanismo de intercambio de información. Los métodos estándares para manejo de atributos son los que se muestran en la Tabla 7.

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

Tabla 7: Métodos para manejo de atributos

2.3 Ejemplos básicos

Para ejemplificar el desarrollo de páginas JSP, se detallan en lo que sigue varios ejemplos prácticos comentando su programación y funcionamiento.

Factoriales

A continuación se lista el código de la página *factoriales.jsp*, que muestra en el navegador del cliente el factorial de los números 0 a 20.

```
<!-- factoriales.jsp -->
<HTML>
<HEAD>
          <TITLE>Cálculo de factoriales con JSP</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">

<%! public long factorial (long numero)
  {
          if (numero == 0) return 1;
          else return numero * factorial(numero - 1);
  } %>

<H1>TABLA DE FACTORIALES</H1>
<table border="1">
<tr><th><i>x</i></th><th><i>x</i>!</th></tr>
<% for (long x = 0; x <=20; ++x) { %>
     <tr><td><%= x %></td><td><%= factorial (x) %></td></tr>
<% } %>
</table>

</BODY>
</HTML>
```

Figura 15: Cálculo de factoriales utilizando JSP

Manejo de excepciones

A continuación se muestra cómo es posible manejar excepciones desde una página JSP utilizando el ejemplo anterior.

```
<!-- factorialestry.jsp -->
<HTML>
<HEAD>
          <TITLE>Cálculo de factoriales con JSP y manejo de excepciones</TITLE>
</HEAD>

<BODY BGCOLOR="#FFFFFF">

<%! int x = 21; %>

<%! public long factorial (long numero) throws IllegalArgumentException
  {
               if ((numero < 0) || (numero > 20))
                      throw new IllegalArgumentException("Fuera de Rango");

               if (numero == 0)
                      return 1;
               else
               return numero * factorial(numero - 1);
  } %>

<% try {
               long resultado = factorial (x); %>
               <p align=center> <%= x %>! = <%= resultado %></p>
```

```
<% } catch (IllegalArgumentException e) { %>
                <p>El argumento para calcular el factorial esta fuera de rango.</p>
<% } %>

</BODY>
</HTML>
```

Envío de e-mails

El código de la página JSP mostrada a continuación permite enviar los campos de cualquier formulario por e-mail a una dirección especificada.

```
<HTML>

<HEAD>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.event.*" %>
<%@ page import="javax.mail.internet.*" %>

<%! String HOST = "lsi2.ei.uvigo.es"; %>
<%! String REMITENTE = "user1@lsi2.ei.uvigo.es"; %>
<%! String DESTINATARIO = "friverola@yahoo.com"; %>
<%! String ASUNTO = "REGISTRO VIA WEB DE FORMULARIOS"; %>
<%! String CONTENIDO; %>

<%! String cadError = ""; %>
<%! String nomParam; %>
<%! String valParam; %>

<%! Enumeration parametros; %>
<%
          parametros = request.getParameterNames();
          for (CONTENIDO=""; parametros.hasMoreElements(); )
          {
                    nomParam = parametros.nextElement().toString();
                    valParam = request.getParameter(nomParam);
                    CONTENIDO += "<STRONG>" + nomParam + "</STRONG>: <i>" + valParam + "</i><BR>";
    }
%>

<%!       class ConnectionHandler extends ConnectionAdapter
          {
                    public void opened(ConnectionEvent e)
                              { // System.out.println("Connection opened."); }

                    public void disconnected(ConnectionEvent e)
                              { // System.out.println("Connection disconnected.");  }

                    public void closed(ConnectionEvent e)
                              { // System.out.println("Connection closed.");  }
          }
%>

<%!       class TransportHandler extends TransportAdapter
          {
                    public void messageDelivered(TransportEvent e)
                    { // System.out.println("Message delivered.");  }

                    public void messageNotDelivered(TransportEvent e)
                              { // System.out.println("Message NOT delivered.");  }

                    public void messagePartiallyDelivered(TransportEvent e)
                              { // System.out.println("Message partially delivered.");  }
          }
%>

<% // Envía el mensaje a la dirección especificada con los parámetros que le lleguen. %>
<% // TRUE si todo va bien, FALSE si se produce algún error.  %>
<%! public boolean SendMessage()
{
                    Properties properties = new Properties();
                    properties.put("mail.smtp.host", HOST);
                    properties.put("mail.from", REMITENTE);
                    Session session = Session.getInstance(properties, null);

                    try
                    {
                              Message message = new MimeMessage(session);
                              InternetAddress[] address = {new InternetAddress(DESTINATARIO)};
                              message.setRecipients(Message.RecipientType.TO, address);
```

```
                                message.setFrom(new InternetAddress(REMITENTE));
                                message.setSubject(ASUNTO);
                                message.setContent(CONTENIDO,"text/html");
                                Transport transport = session.getTransport(address[0]);
                                transport.addConnectionListener(new ConnectionHandler());
                                transport.addTransportListener(new TransportHandler());
                                transport.connect();
                                transport.sendMessage(message,address);
                        }
                        catch(Exception e)
                        {
                                cadError = e.toString();
                                return false;
                        }
                        return true;
}
%>
</HEAD>


<BODY BGCOLOR="#BBBBBB">
<% if (!SendMessage()) { %>
          <FONT color="#1F5EB1" size="4" face="Tahoma"><STRONG>¡<%= cadError %>!</STRONG></FONT>
          <img src="imagenes/duke_405.gif" align="left" border="0" hspace="0">
<% } else { %>
          <FONT color="#1F5EB1" size="4" face="Tahoma"><STRONG>Sus datos fueron enviados correctamente                    a <%=
DESTINATARIO %>:</STRONG>
          <p>
          <%= CONTENIDO %>
          </FONT>
<% } %>
</BODY>
```

Para que la página compile correctamente se deben incluir en la variable *CLASSPATH* los siguiente ficheros JAR: *activation.jar*, *mail.jar* y *pop3.jar*.

El mensaje que visualiza el navegador cliente después de ejecutar la página *SendTo.jsp* se muestra en la Figura 16.

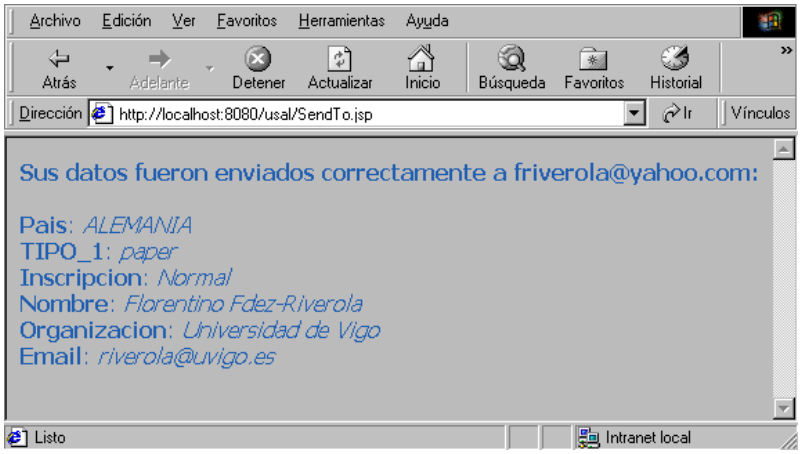

Figura 16: Resultado de la ejecución de la página *SendTo.jsp*

Acceso a bases de datos

Otra utilidad importante es la de acceder a bases de datos utilizando un controlador JDBC. El siguiente ejemplo muestra la página *notas.jsp* que accede a la base de datos *notas* gestionada por *mySQL*.

```
<HTML>

<HEAD>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.Vector" %>

<%! String DNI; %>
```

```
<% DNI = request.getParameter("dni"); %>

<%! String MI_TABLA; %>
<% MI_TABLA = request.getParameter("tabla"); %>

<%! Vector Titulos = new Vector(); %>
<%! Vector Valores = new Vector(); %>
<%! String cadError = ""; %>


<% // TRUE si todo va bien, FALSE si se produce algún error.  %>
<%! public boolean consultaDNI()
{
          String consultaSQL;

          try {
                    Class.forName("org.gjt.mm.mysql.Driver");
                    Connection miCon = DriverManager.getConnection("jdbc:mysql://localhost/notas");
                    Statement sentencia = miCon.createStatement();

                    consultaSQL = "SELECT count(*) from " + MI_TABLA + " where dni='" + DNI + "'";
                    ResultSet miRes = sentencia.executeQuery(consultaSQL);
                    miRes.next();

                    if ( !((miRes.getString(1)).equalsIgnoreCase("1")) )
                    {
                              cadError = "DNI INCORRECTO";
                              return false;
                    }
                    else
                    {
                              consultaSQL = "SELECT * from " + MI_TABLA + " where dni='" + DNI + "'";
                              miRes = sentencia.executeQuery(consultaSQL);

                              ResultSetMetaData meta = miRes.getMetaData();
                              int columnas = meta.getColumnCount();
                              int fila = 0;
                              miRes.next();

                              Titulos.removeAllElements();
                              Valores.removeAllElements();
                              for (int i=1; i<=columnas; i++)
                              {
                                        Titulos.addElement(meta.getColumnLabel(i).toUpperCase());
                                        Valores.addElement(miRes.getObject(i));
                              }
                    }

                    miRes.close();
                    sentencia.close();
                    miCon.close();

          } catch (ClassNotFoundException e) {
                    cadError = "Controlador JDBC no encontrado";
                    return false;
          } catch (SQLException e) {
                    cadError = "Excepción capturada de SQL " + e;
                    return false;
          } return true;
} %>
</HEAD>


<BODY BGCOLOR="#BBBBBB">
<% if (!consultaDNI()) { %>
          <FONT color="#1F5EB1" size="4" face="Tahoma"><STRONG>¡<%= cadError %>!</STRONG></FONT>
          <img src="imagenes/duke_405.gif" align="left" border="0" hspace="0">
<% } else { %>
          <HR>
          <CENTER><TABLE border="1" width=75%>
          <TR>
          <% for (int x = 0; x < Titulos.size(); x++) { %>
                    <TH><%= Titulos.elementAt(x) %> </TH>
          <% } %>
          </TR>

          <TR>
          <% for (int x = 0; x < Valores.size(); x++) { %>
                <TD>
                <CENTER><FONT color="#1F5EB1"><%= Valores.elementAt(x) %> </FONT></CENTER>
                </TD>
          <% } %>
          </TR>
          </TABLE></CENTER>
          <HR>
<% } %>
</BODY>
</HTML>
```

Para que la página compile correctamente se deben incluir en la variable *CLASSPATH* la ruta para localizar el controlador JDBC.

La página encargada de ejecutar *notas.jsp* es *notas.html*, cuyo código se muestra a continuación.

```
<HTML>

<HEAD>
</HEAD>

<BODY>
<FORM action= “notas.jsp" method="POST">
          <STRONG>DNI:</STRONG>
          <INPUT type="text" name="dni" size=8 maxlength=8>

          <!--<INPUT type="hidden" name="tabla" value="proafeb01">-->

          <STRONG>CONVOCATORIA:</STRONG>
          <SELECT name="tabla"><OPTION selected>PROA_FEB_01<OPTION>PROA_SEP_01</SELECT>

          <INPUT type="submit" value="BUSCAR">
          <INPUT type="reset" value="BORRAR">
</FORM>
</BODY>
</HTML>
```

La Figura 17 muestra la salida generada por la página *notas.jsp*.

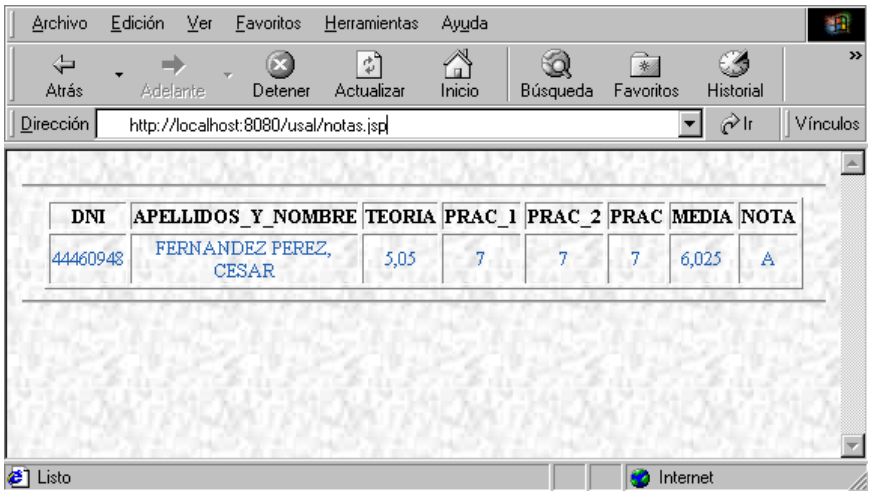

Figura 17: Resultado de la ejecución de la página *notas.jsp*

3 Tendencias actuales

Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. El patrón MVC se emplea frecuentemente en aplicaciones web, donde la vista es la página HTML ó XHTML.

El modelo es la representación específica del dominio de la información sobre la cual funciona la aplicación. El modelo es otra forma de llamar a la capa de dominio. La lógica de dominio añade significado a los datos; por ejemplo, calculando si hoy es el cumpleaños del usuario o los totales, impuestos o portes en un carrito de la compra. Por otro lado, la vista presenta el modelo en un formato adecuado para interactuar,

usualmente un elemento de interfaz de usuario. Muchas aplicaciones utilizan un mecanismo de almacenamiento persistente (como puede ser una base de datos) para almacenar los datos. MVC no menciona específicamente esta capa de acceso a datos. Finalmente el controlador es un elemento que permite responder a eventos, usualmente acciones del usuario e invoca cambios en el modelo y probablemente en la vista. Es común pensar que una aplicación tiene tres capas principales: presentación, lógica de negocio, y acceso a datos. En MVC, la capa de presentación está partida en controlador y vista.

Cuando se desarrolló la tecnología JSP, esta fue entendida por la comunidad de desarrolladores de sítios web como una tecnología equivalente a los servlets. Las páginas JSP se desplegaban en un contenedor que tenía la capacidad de transformarlas en Servlets de forma dinámica y de ahí la equivalencia. Con el tiempo y motivado por la introducción de la novedosa idea del MVC de separar las acciones llevadas a cabo sobre el modelo de la forma de las vistas del modelo, servlets y JSP empezaron a emplearse con diferentes objetivos. Los servlets ejercían el papel de controladores y se empleaban para realizar acciones sobre el modelo (eliminación, modificación o inserción de datos en el modelo) el cual estaba implementado habitualmente con Java Beans. Finalmente, las páginas JSP actuaban como vistas, de tal forma que únicamente permitían construir distintas vistas sobre los datos. Este esquema se describe en la Figura 18 y se conoce como MVC Model 2.

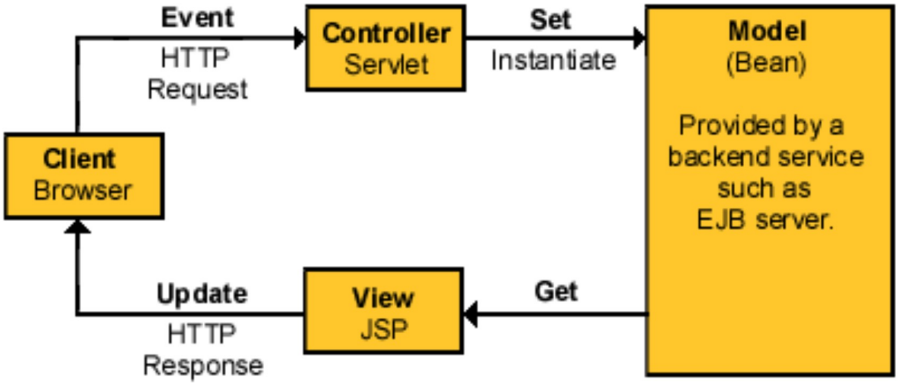

Figura 18: MVC Model 2

De esta forma el flujo de control de una página web era el siguiente:

1. El usuario interactua con la interfaz (por ejemplo pulsando un enlace con destino a un servlet capaz de realizar una acción sobre los datos).
2. El servlet recibe la petición y modifica los datos del modelo en la forma indicada por el usuario.
3. El servlet delega a las páginas JSP de vista. Las páginas JSP de vista obtienen los datos necesarios del modelo y generan una interfaz adecuada para el usuario donde reflejan los cambios realizados en el modelo.

Esta es la forma más primitiva de emplear el patrón MVC. Sin embargo, con el paso del tiempo aparecieron frameworks que distribuían implementaciones de este patrón que aportaban numerosas mejoras. Este es el caso de Structs. La Figura 19 representa la configuración típica de un sitio web basado en Struts.

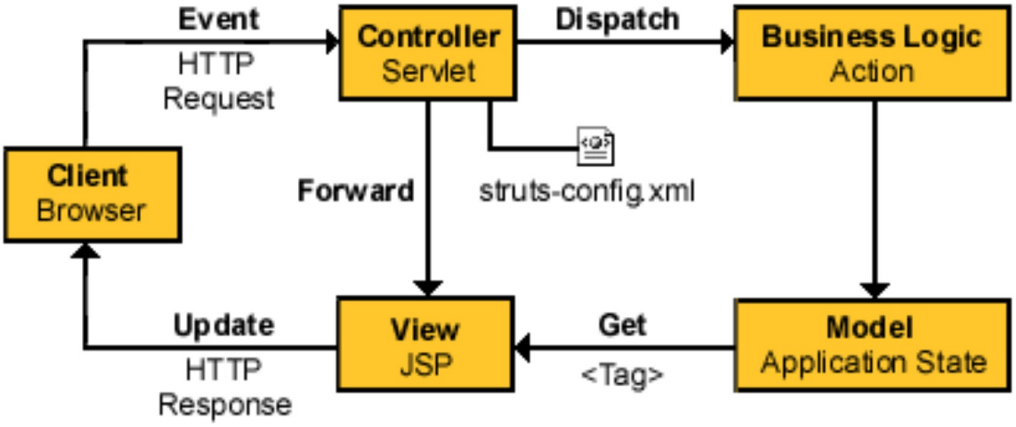

Figura 19: MVC Model 2 implementado con Struts

Como se puede ver, los servlets que implementaban los controladores se cambiaron por un Servlet proporcionado por Struts y varias clases que heredan de Action. Estas clases se configuran mediante el fichero struts-config.xml. Cada clase que hereda de Action implementa una acción sobre el modelo. La vista, el modelo y el navegador se mantienen de la misma forma que en la versión anterior.

En Struts, el flujo de operaciones se realiza según se describe a continuación:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace)
2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se refleja los cambios en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El modelo no debe tener conocimiento directo sobre la vista.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente.

Además, Struts incorpora un lenguaje de etiquetas propio y soporte para internacionalización de los sítios web. Tras el éxito de Struts surgió Webwork, un framework similar a Struts orientado al desarrollo de sítios web que incorpora, además de una implementación de MVC model 2, un sistema de validaciones, un lenguaje de expresiones, un conjunto de interceptores avanzados que permiten especificar ortogonalmente acciones para el control de la aplicación y soporte para la internacionalización.

4 JPA con Hibernate

Hibernate (http://www.hibernate.org) implementa la Java Persistence API (en adelante JPA) permitiendo un mapeo directo desde Java Beans a bases de datos relacionales como MySQL, Postgre o cualquier otro gestor. Esta API es a día de hoy la implementación directa de los Entity Beans de la Arquitectura Empresarial de Java (JEE, Java Enterprise Edition). JPA es una implementación concreta de una tecnología conocida con el nombre de ORM (Object Relational Mapping) o mapeo de objetos a relaciones.
Para usar Hibernate simplemente hay que descargarlo de la página web (http://www.hibernate.org), y desarrollar Beans con las anotaciones propias de java empresarial tal como se puede ver en el siguiente código:

```
package es.ei.uvigo;
@Entity
@Table( name="users" )
public class User implements Serializable{
   @Id
   @Column(name = "username", nullable = false)
   private String username;

    @Column(nullable = false, length = 50, unique = true)
    private String clave;

   public String getUsername(){
      return username;
   }
   public String getClave(){
      return clave;
   }
   public void setUsername(String username){
      this.username=username;
   }
   public void setClave(String clave){
      this.clave=clave;
   }
}
```

JPA es "default driven" lo que significa que todos los comportamientos se entienden por defecto. En este sentido no hace falta emplear gran cantidad de anotaciones para configurar el comportamiento concreto de los beans sinó que se escogerán unos comportamientos habituales como los por defecto y sólo habrá que configurar aquellos que resulten más particulares. De las anotaciones empleadas en el código fuente únicamnete @Entity y @Id son necesarias para determinar el comportamiento del bean.
Una vez creado el bean se debe crear un fichero de configuración que se colocará en el directorio WEB-INF del fichero .jar que se emplee para guardar los beans persistentes y que deberá tener el nombre de persistence.xml. Este fichero de configuración determina propiedades como el nombre de la base de datos, el servidor, el usuario o la clave de acceso. Este fichero tiene una configuración similar a esta:

```
<persistence> <!-- Colocalo en el directorio META-INF del fichero .jar y llámalo persistence.xml -->
   <persistence-unit name="manager1" transaction-type="RESOURCE_LOCAL">
     <class>es.ei.uvigo.User</class>
     <class>es.ei.uvigo.Cars</class>
     <!-- enumera todas las clases persistentes -->
     <properties>
       <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
       <property name="hibernate.connection.driver_class" value="org.hsqldb.jdbcDriver"/>
       <property name="hibernate.connection.username" value="sa"/>
       <property name="hibernate.connection.password" value=""/>
       <property name="hibernate.connection.url" value="jdbc:hsqldb:."/>
       <property name="hibernate.max_fetch_depth" value="3"/>
```

```
        <!-- cache configuration -->
        <property name="hibernate.ejb.classcache.org.hibernate.ejb.test.Item" value="read-write"/>
        <property name="hibernate.ejb.collectioncache.org.hibernate.ejb.test.Item.distributors"
          value="read-write, RegionName"/>

      </properties>
    </persistence-unit>
</persistence>
```

Como se puede ver, en el fichero de configuración debe existir una entrada <class> </class> para cada uno de los beans persistentes. Una vez creado el fichero de configuración, para realizar la persistencia de un bean, el código necesario es muy sencillo y se expone en la siguiente a continuación:

```
// Use persistence.xml configuration
EntityManagerFactory
    emf = Persistence.createEntityManagerFactory("manager1");
// Retrieve an application managed entity manager
EntityManager em = emf.createEntityManager();

     User u=new User();
     u.setUsername("moncho");
     u.setClave("xjmon764d");
     em.persist(u);


em.close();
emf.close(); //close at application end
```

Como se puede ver la persistencia es muy sencilla. Además, JPA permite la persistencia automática de objetos que contengan relacciones y eliminando la necesidad de realizar un diseño a nivel de base de datos y la codificación de instrucciones SQL. La conclusión es que esta tecnología es muy interesante para el desarrollo de cualquier tipo de aplicación Java incluyendo aplicaciones web.