Apuntes

de

J2EE

Tema 3:
Tecnologías
Java distribuídas

Uploaded by

Ingteleco

http://ingteleco.webcindario.com
ingtelecoweb@hotmail.com

3.- TECNOLOGÍAS JAVA DISTRIBUIDAS

3.1.- TECNOLOGÍAS WEB: SERVLETS Y JSP

3.1.1.- Justificación de este tipo de tecnologías

Como ya se ha comentado anteriormente, el funcionamiento tradicional de la web adoptaba un enfoque totalmente estático que implicaba que las aplicaciones basadas en la web carecieran de dinamismo y se redujeran únicamente a la solicitud de páginas web estáticas almacenadas en un servidor web y a la presentación de éstas en un navegador.

Con este enfoque la interacción entre el usuario y la aplicación era mínima, no permitiéndose que el usuario enviase datos desde el cliente hasta la aplicación ni que ésta generase una respuesta personalizada al usuario acorde a los datos que éste había enviado.

Todo ello provocó la necesidad de que surgieran tecnologías que permitieran dotar de un mayor dinamismo a las aplicaciones web, permitiendo el envío de datos desde el cliente hasta la aplicación y la generación de contenidos dinámicamente, principalmente en forma de páginas HTML.

Las dos tecnologías Java que permiten lograr estos objetivos son los *Servlets* y las *JSP*.

3.1.2.- Características básicas de los Servlets y las JSP

Existe una tecnología precursora de los Servlets y las JSP como son los CGI. Los CGI son unos programas que se ejecutan en el servidor web y que permiten recibir una petición de un cliente y generar una respuesta dinámicamente en forma de página HTML. Sin embargo, los CGI presentan una serie de problemas:

- Son dependientes de la plataforma en la que se ejecutan, de forma que el código de los CGI debe ser compilado específicamente para la plataforma sobre la que se vaya a ejecutar, no pudiéndose ejecutar sobre otra plataforma sin previamente ser modificados y recompilados para esa otra plataforma.

- Los CGI crean un nuevo proceso para atender a cada petición del cliente. Esto implica que cada vez que un cliente realiza una petición es necesario crear un nuevo proceso, asignarle recursos e inicializarlo. Esto ocasiona varios problemas:

    - El tiempo de respuesta es muy grande ya que es necesario realizar la inicialización de los procesos para cada petición.

    - La sobrecarga puede ser muy grande en el caso de que el número de peticiones así lo sea, ya que es necesario asignar recursos para cada proceso que se crea.

    - Al crearse un nuevo proceso para cada petición, no es posible mantener información sobre el estado de una petición a otra.

Los Servlets y las JSP, al igual que los CGI, son tecnologías de servidor centradas en la generación dinámica de contenidos web, esto quiere decir que son tecnologías que residen y se ejecutan en el

servidor web y que tienen como finalidad generar dinámicamente paginas web que posteriormente sean visualizadas en un browser. Sin embargo y a diferencia de los CGI, los Servlets y las JSP son tecnologías basadas en la plataforma Java, gracias a lo cual permiten superar muchas de las limitaciones que planteaban los CGI:

- Al estar basadas en la plataforma Java, son tecnologías independientes de la plataforma sobre la que se vayan a ejecutar (tecnologías multiplataforma) y que hacen uso de todas las facilidades que la plataforma Java proporciona, usando el mecanismo JDBC para el acceso a bases de datos, realizando conexiones a máquinas remotas, comunicándose con componentes Enterprise JavaBeans, etc.

- El rendimiento es muy superior al de los CGI ya que no se crea un proceso para cada petición del cliente sino que, gracias a su naturaleza multithreading, se crea un único proceso en memoria, se inicializa una única vez y es compartido por todas las peticiones que realicen los cliente. Las ventajas de este nuevo enfoque son múltiples:

  - El tiempo de respuesta disminuye al realizarse la inicialización una única vez y no una para cada petición.

  - La sobrecarga disminuye ya que solamente tenemos un único proceso independientemente del número de peticiones.

  - Es posible mantener el estado de una petición a otra, así como todos los recursos que tengamos asignados tales como conexiones a bases de datos, sin que sea necesario volver a establecer estas conexiones cada vez que se realiza una nueva petición.

Según todo lo anterior, las tres características básicas que podemos destacar de los Servlets y las JSP son las siguientes:

- Son tecnologías que residen y se ejecutan en el servidor web.

- Tienen como principal finalidad la generación dinámica de contenidos web.

- Son tecnologías basadas en la plataforma Java.

Sin embargo, aunque son tecnologías muy similares en cuanto a la esencia de su funcionamiento, presentan ciertos aspectos que las hacen diferentes.

3.1.3.- Diferencias entre los Servlets y las JSP

Tanto los Servlets como las JSP describen la manera en que procesar una petición de un cliente HTTP y crear una respuesta dinámicamente (generalmente en forma de página HTML). Sin embargo, mientras los Servlets se implementan mediante clases Java normales y corrientes, siendo por tanto íntegramente código Java, las páginas JSP son documentos de texto que incluyen una combinación de código HTML, etiquetas JSP y código Java.

Los **Servlets** son una tecnología que fue desarrollada como un mecanismo para aceptar peticiones de los browsers, consultar datos de bases de datos u otras aplicaciones, procesar estos datos y formatearlos (en forma de documentos HTML, principalmente) para su presentación en los browsers del cliente.

Los Servlets usan sentencias de impresión, sentencias tipo "*print*" dentro de las cuales se incluye código HTML embebido, para devolver la respuesta al browser que realizó la petición.

El uso de código HTML embebido en sentencias de impresión provoca que tanto el código HTML estático como el código HTML generado dinámicamente se encuentre mezclado y sea tratado de la misma manera, lo que conlleva una serie de problemas:

- El diseñador de la página web resultado no puede ver la apariencia de ésta hasta la ejecución del Servlet.
- Cuando el diseñador de la página web resultado quiera modificar algún aspecto de ésta, la localización de la parte de código HTML a modificar es muy complicado al estar éste embebido dentro de las sentencias Java de impresión.
- Cualquier cambio que se quiera realizar sobre el código HTML estático de la página web a devolver, implica recompilar y volver a inicializar el Servlet en el servidor web, ya que aunque es código HTML y en principio no debería ser necesario recompilar nada para ver los resultados de una modificación, al estar embebido dentro del código Java del Servlet es necesario recompilar éste para poder ver los resultados.

La **tecnología JSP** permite solucionar los problemas que plantean los Servlets gracias a que esta tecnología posee como característica fundamental el tener perfectamente separado el contenido HTML estático, de la lógica de presentación encargada de la generación del código HTML dinámico (encontrándose esta última separada entre las etiquetas <% y %>). Las páginas JSP contienen tanto código HTML estático como código encargado de la generación de código HTML dinámicamente, pero la diferencia radica en que, a diferencia de los Servlets, lo tiene perfectamente separado y realiza un tratamiento distinto de uno y otro.

Esta separación aporta múltiples ventajas:

- Cuando el diseñador de la página JSP realice algún cambio sobre ella, ésta será recompilada y e inicializada en el servidor automáticamente.
- Permite separar claramente la labor del desarrollador encargado de implementar en Java la lógica que se encarga de generar el código HTML dinámico y la labor del diseñador de la página web que se encarga de generar el código HTML estático de la página JSP. De esta forma, el desarrollador Java no necesita tener grandes conocimientos de diseño de interfaces mediante HTML y el diseñador de páginas web no necesita tener conocimiento alguno de Java para realizar su trabajo.
- Además, las páginas JSP suelen tener la lógica de negocio encargada de la generación de contenidos dinámicos implementada mediante componentes *JavaBeans*, lo cual permite que la lógica de presentación de una página JSP sea reutilizada por otras páginas JSP.

3.1.4.- Servlets

Para poder hacerse una idea del funcionamiento de los Servlets y del aspecto que tienen los mismos, lo mejor es estudiar un ejemplo sencillo. Imagínese que en una página web se desea recabar la opinión de los usuarios sobre un determinado sitio web así como algunos de sus datos personales, con el fin de realizar un estudio estadístico. Dicha información podría ser almacenada en una base de datos para su

posterior análisis. Sin embargo, como queremos que el ejemplo sobre el que vamos a trabajar sea sencillo, eliminaremos del mismo el almacenamiento de las opiniones y su posterior estudio estadístico y nos limitaremos a recoger los datos del usuario y su opinión y a devolver una pagina HTML que visualice esta información.

La primera tarea sería diseñar un formulario en el que el visitante pudiera introducir los datos. Para ello usaremos las etiquetas que proporciona el lenguaje HTML (<FORM>, <ACTION>, <TYPE>,...).

3.1.4.1.- Formulario

El formulario contendrá dos campos de tipo **TEXT** donde el visitante introducirá su nombre y apellidos. A continuación, deberá indicar la opinión que le merece la página visitada eligiendo una entre tres posibles (Buena, Regular y Mala). Por último, se ofrece alusuario la posibilidad de escribir un comentario si así lo considera oportuno. La figura 3.1 muestra el diseño del formulario creado.

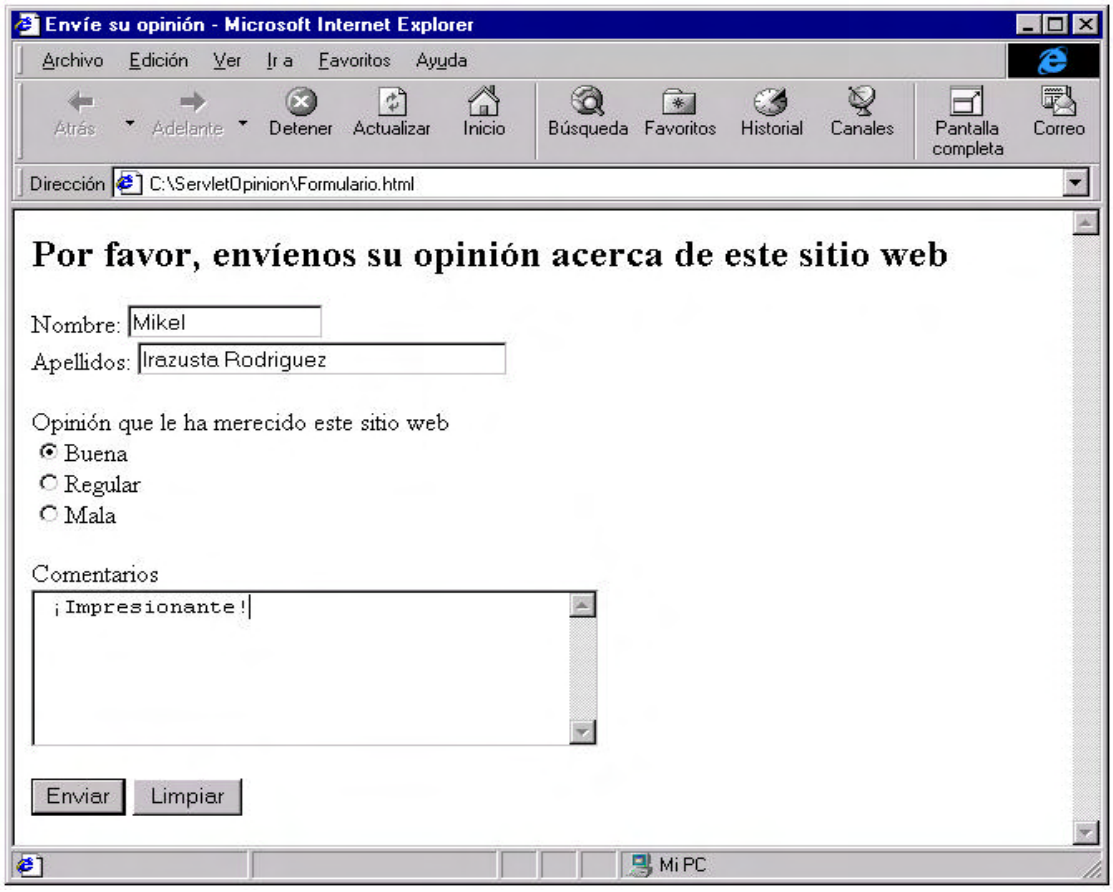

Figura 3.1: Diseño del formulario de adquisición de datos.

El código correspondiente a la página HTML que contiene este formulario es el siguiente:

```
<HTML>
<HEAD>
   <TITLE> Envíe su opinión </TITLE>
</HEAD>
<BODY>
<H2>Por favor, envíenos su opinión acerca de este sitio web</H2>
<FORM ACTION="http://localhost:8080/examples/servlet/ServletOpinion"
```

```
 METHOD="GET">
   Nombre: <INPUT TYPE="TEXT" NAME="nombre" SIZE=15><BR>
   Apellidos: <INPUT TYPE="TEXT" NAME="apellidos" SIZE=30><P>

   Opinión que le ha merecido este sitio web<BR>
   <INPUT TYPE="RADIO" CHECKED NAME="opinion" VALUE="Buena">Buena<BR>
   <INPUT TYPE="RADIO" NAME="opinion" VALUE="Regular">Regular<BR>
   <INPUT TYPE="RADIO" NAME="opinion" VALUE="Mala">Mala<P>

   Comentarios <BR>
   <TEXTAREA NAME="comentarios" ROWS=6 COLS=40> </TEXTAREA><P>
   <INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
   <INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">
</FORM>
</BODY>
</HTML>
```

Figura 3.2: Código de la página HTML del formulario.

En el código anterior, hay algunas cosas que merecen ser comentadas. En primer lugar, es necesario asignar un identificador único (es decir, un valor de la propiedad **NAME**) a cada uno de los campos del formulario, ya que la información que reciba el Servlet estará organizada en forma de **pares de valores**, donde uno de los elementos de dicho par será un *String* que contendrá el nombre del campo. Así, por ejemplo, si se introdujera como nombre del visitante "Mikel", el Servlet recibiría del browser el par **nombre=Mikel**, que permitirá acceder de una forma sencilla al nombre introducido mediante el método **getParameter()**, tal y como se explicará posteriormente al analizar el código del Servlet de este ejemplo. Por este motivo, es importante no utilizar nombres duplicados en los elementos del formulario.

Por otra parte, puede observarse que en la etiqueta **<FORM>** se han utilizado dos propiedades, **ACTION** y **METHOD**. Con la propiedad **METHOD** indicamos que el método empleado para la transmisión de datos desde el cliente hasta el servidor web es el método **HTTP GET**. También se podría haber usado el método **HTTP POST**. Mediante la propiedad **ACTION** especificamos el **URL** del Servlet que debe procesar los datos que introducimos en el formulario. El **URL** presentado en el ejemplo tiene las siguientes características:

- El Servlet se encuentra ubicado en la máquina local (su nombre es **localhost** y la dirección IP es **127.0.0.1**). El nombre que debe aparecer en la URL deberá ser el nombre o la dirección IP de la máquina donde esté ubicado el Servlet.

  ```
  <FORM ACTION="http://localhost:8080/examples/servlet/ServletOpinion" ...>
  ```

  o de otra forma,

  ```
  <FORM ACTION="http://127.0.0.1:8080/examples/servlet/ServletOpinion" ...>
  ```

- En la URL (y seguido del nombre de la máquina) deberemos incluir el puerto donde se encuentra "escuchando" el servidor web. En nuestro ejemplo, el servidor web que contiene al Servlet se encuentra "escuchando" por el **puerto 8080**.

  ```
  <FORM ACTION="http://localhost:8080/examples/servlet/ServletOpinion" ...>
  ```

- El Servlet se encuentra situado en un **subdirectorio** (virtual) del servidor web llamado **examples/servlet**.

```
<FORM ACTION="http://localhost:8080/examples/servlet/ServletOpinion" ...>
```

- El **nombre del Servlet** de nuestro ejemplo es **ServletOpinion**, y es éste el que recibirá la información enviada por el cliente al servidor a través del formulario, y quien se encargará de diseñar la respuesta y de pasarla al servidor para que éste a su vez la envíe de vuelta al cliente. Al final se ejecutará una clase llamada **ServletOpinion.class**.

3.1.4.2.- Código del Servlet

Como se ha mencionado anteriormente, el Servlet que gestionará toda la información del formulario se llamará **ServletOpinion**. Como un Servlet es una clase Java, deberá encontrarse almacenado en un fichero con el nombre **ServletOpinion.java**. Por mantener lo más sencillo posible este ejemplo, nuestro Servlet se limitará a responder al usuario con una página HTML que muestre de nuevo la información introducida por éste en el formulario, dejando como una posible ampliación todo lo relacionado con el almacenamiento de los datos y su análisis estadístico. El código fuente de la clase ServletOpinion es el siguiente:

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

public class ServletOpinion extends HttpServlet{

   // Declaración de variables miembro correspondientes a los campos
   // de formulario
   private String nombre;
   private String apellidos;
   private String opinion;
   private String comentarios;

   // Este método se ejecuta una única vez (al ser inicializado el servlet)
   // Se suelen inicializar variables y realizar operaciones costosas en
   // tiempo de ejecución (conexiones a BD,...)
   public void init(ServletConfig config) throws ServletException{
      // Llamada al método init() de la superclase (HttpServlet)
      // Así se asegura una correcta inicialización del servlet
      super.init(config);
      System.out.println("Iniciando ServletOpinion...");
   }

   // Este método es llamado por el servidor web al "apagarse" (shutdown)
   // Sirve para proporcionar una correcta desconexión de una base de
   // datos, cerrar ficheros,...
   public void destroy(){
      System.out.println("No hay nada que hacer...");
   }

   // Método llamado automáticamente cuando se realiza una invocación
   // al servlet a través del método HTTP GET.
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException{

      // Adquisición de los valores del formulario a través del objeto req
      nombre = req.getParameter("nombre");
      apellidos = req.getParameter("apellidos");
```

```
      opinion = req.getParameter("opinion");
      comentarios = req.getParameter("comentarios");

      // Devolver al usuario una página HTML con los valores adquiridos

      // En primer lugar se establece el tipo de contenido MIME de la respuesta
      resp.setContentType("text/html");

      // Se obtiene un PrintWriter donde escribir (sólo para mandar texto)
      PrintWriter out = null;
      try{
         out = resp.getWriter();
      }catch (IOException io){
         System.out.println("Se ha producido una excepción");
      }

      // Se genera el contenido de la página HTML
      out.println("<html>");
      out.println("<head>");
      out.println("<title>Valores recogidos en el formulario</title>");
      out.println("</head>");
      out.println("<body>");
      out.println("<b><font size=+2>Valores recogidos del formulario:
                  </font></b>");
      out.println("<p><font size=+1><b>Nombre: </b>" + nombre + "</font>");
      out.println("<br><font size=+1><b>Apellido: </b>" + apellidos
                  + "</font>");
      out.println("<p><font size=+1><b>Opini&oacute;n: </b><i>" + opinion
                  + "</i></font>");
      out.println("<br><font size=+1><b>Comentarios: </b>" + comentarios
                  + "</font>");
      out.println("</body>");
      out.println("</html>");

      // Se fuerza la descarga del buffer y se cierra el PrintWriter.
      // De esta forma se libera el recurso.
      out.flush();
      out.close();
   }

   // Función que permite al servidor web obtener una pequeña descripción del
   // servlet, que cometido tiene, nombre del autor, etc.
   public String getServletInfo(){
      return  "Este servlet lee los datos de un formulario" +
              " y los muestra en pantalla";
   }
}
```

Figura 3.3: Código del Servlet "ServletOpinion"

El resultado obtenido en el browser tras la ejecución del Servlet puede apreciarse en la figura 3.4:

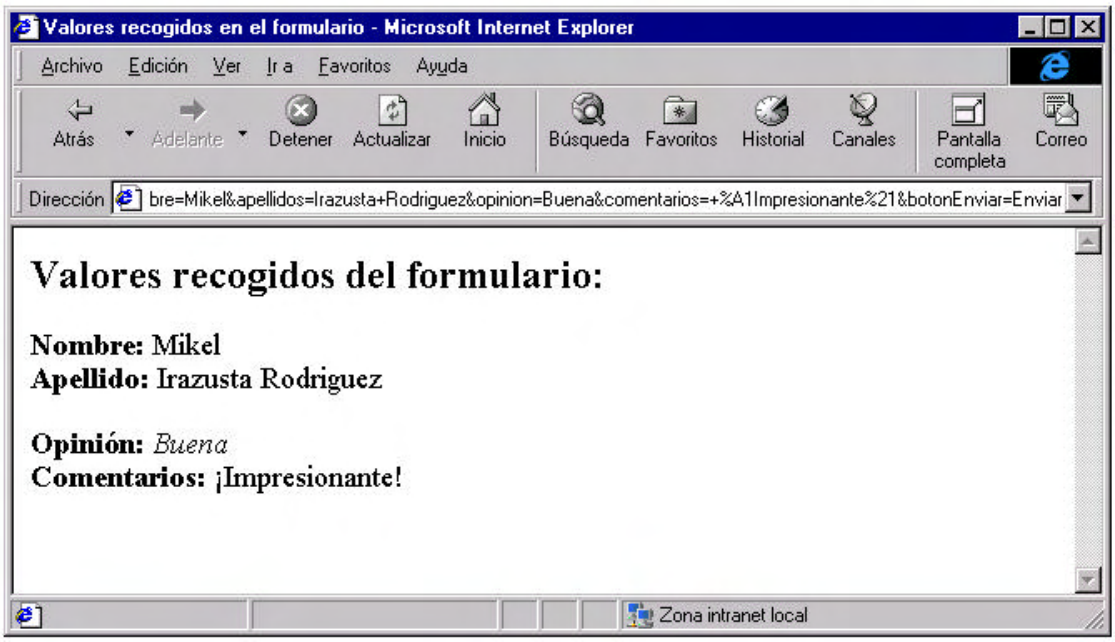

Figura 3.4: Página HTML devuelta por el Servlet.

Para lograr una mayor simplicidad en esta primera aproximación a los Servlets, se ha evitado tratar de conseguir un código más sólido, que debería realizar las comprobaciones pertinentes (verificar que los *String* no son *null* después de leer el parámetro, excepciones que se pudieran dar, etc.), y se ha optado por mostrar únicamente las partes que son esenciales para entender su funcionamiento.

Se puede observar como el aspecto de un Servlet es muy similar al de cualquier otra clase Java. Sin embargo, sí existen una serie de aspectos particulares en el código de los Servlets que es necesario comentar más detalladamente.

3.1.4.2.1.- Paquetes a importar en los Servlets

Para el desarrollo de Servlets se dispone de dos paquetes: el paquete `javax.servlet` y el paquete `javax.servlet.http`. Todas las clases e interfaces necesarias para la programación de Servlets se encuentran en estos dos paquetes, razón por la cual los hemos importado en nuestro ejemplo.

El paquete **`javax.servlet`** contiene una serie de clases generales para el desarrollo de Servlets con independencia del protocolo que se vaya a usar para acceder a éstos.

El paquete **`javax.servlet.http`** es un paquete específico para el desarrollo de Servlets que sebasan en el protocolo HTTP. Los Servlets basados en el protocolo HTTP son el tipo de Servlets más común y, por tanto, el que usaremos en nuestro ejemplo.

3.1.4.2.2.- La clase HTTPServlet

La forma más sencilla de implementar un Servlet basado en el protocolo HTTP es heredando de la clase **HTTPServlet**.

La clase **HTTPServlet** proporciona gran parte de la funcionalidad necesaria para el desarrollo de Servlets basados en el protocolo HTTP, de tal forma que la manera más sencilla de implementar un Servlet de este tipo es heredando de ella toda la funcionalidad que aporta y redefiniendo aquellos métodos que queramos que tengan un comportamiento especifico para nuestro Servlet.

En nuestro ejemplo, **la clase ServletOpinion hereda de la clase HTTPServlet**, indicando de esta manera que la clase ServletOpinion va a ser un Servlet que hace uso del protocolo HTTP. Una vez hecho esto, nos encargaremos de ir redefiniento algunos de sus métodos.

3.1.4.2.3.- Método Init

Cuando un Servlet es cargado por primera vez, el método **init()** es llamado por el servidor web. Este método no será llamado nunca más mientras el Servlet se esté ejecutando. Esto permite al Servlet efectuar cualquier operación de inicialización potencialmente costosa en términos de CPU, ya que solo se ejecutará la primera vez. Esto es una ventaja importante frente a los programas CGI, que son cargados en memoria cada vez que hay una petición por parte del cliente. Por ejemplo, si en un día hay 500 consultas a una base de datos, mediante un CGI habría que abrir una conexión con la base de datos 500 veces, frente a una única apertura que sería necesaria con un Servlet, pues dicha conexión podría quedar abierta a la espera de recibir nuevas peticiones.

Si el servidor web permite precargar los Servlets, el método **init()** será llamado al iniciarse el servidor. Si el servidor web no tiene esa posibilidad, será llamado únicamente la primera vez que se realice una petición al Servlet por parte del cliente (esta segunda opción es la que se da con el servidor web que usaremos en las prácticas, el método **init()** sólo se ejecutará la primera vez que se invoque al Servlet).

El método **init()** tiene un único argumento, que es una referencia a un objeto de la interfaz **ServletConfig**, que proporciona los parámetros de inicialización del Servlet.

Siempre que se redefina el método **init()** de la clase base **HTTPServlet** (tal y como se ha hecho en nuestro ejemplo), será preciso llamar al método **init()** de la clase padre (como así se ha hecho en nuestro caso), a fin de garantizar que la inicialización se efectúe correctamente.

3.1.4.2.4.- Método Destroy

El método **destroy()** no tiene ninguna función en el Servlet de nuestro ejemplo, ya que no se ha utilizado ningún recurso adicional que necesite ser liberado.

Sin embargo, este método tiene mucha importancia en aquellos Servlets que tengan asignados ciertos recursos, ya que permite realizar una descarga correcta del Servlet de forma que no queden recursos ocupados indebidamente. Tareas propias de este método son, por ejemplo, el cierre de conexiones con bases de datos, el cierre de ficheros, etc.

3.1.4.2.5.- Métodos doGet y doPost

Estos dos métodos son **el núcleo de todo Servlet** que se base en el protocolo HTTP y serán los dos métodos fundamentales que deberemos redefinir cuando implementemos un Servlet.

Los métodos **doGet()** y **doPost()** se invocan cada vez que un cliente realiza una petición al Servlet a través del mecanismo **HTTP GET** y **HTTP POST** respectivamente, siendo éstos los encargados de

generar la respuesta a dicha petición. El funcionamiento se basa en que la clase **HTTPServlet**, de la que heredamos, tiene implementado un método que se encarga de recibir la petición del cliente HTTP, distinguir que tipo de método se ha usado en la petición (**HTTP GET** o **HTTP POST**) e invocar el método **doGet()** o **doPost()** según corresponda.

De esta forma, cuando implementemos un Servlet, lo único que debemos hacer es heredar de la clase **HTTPServlet** e **implementar al menos uno de los métodos doGet() y doPost()** para que el Servlet realice el comportamiento que nosotros deseemos cada vez que reciba una petición.

Además, la clase **HTTPServlet** es bastante "inteligente" ya que es capaz de saber qué métodos han sido redefinidos en una subclase, de forma que puede comunicar al cliente que tipos de métodos soporta el Servlets en cuestión. Así, si **en la clase ServletOpinion sólo se ha redefinido el método doGet**, si el cliente realiza una petición de tipo HTTP POST el servidor lanzará automáticamente un mensaje de error similar al siguiente:

501 Method POST Not Supported

Los métodos **doGet()** y **doPost()** reciben como parámetros dos objetos: un objeto de la interfaz **HTTPServletRequest** y un objeto de la interfaz **HTTPServletResponse**.

- El objeto **HTTPServletRequest** contiene información acerca de la petición de servicio que el cliente realiza al Servlet. Este objeto contendrá la información que el cliente le envía al Servlet (a través del formulario) en forma de parejas de parámetros *clave/valor*, así como información acerca del cliente que realiza la petición (máquina y browser).

  La interfaz **HTTPServletRequest** posee varios métodos que permiten acceder a la información del objeto. El más usado (y el que hemos usado en nuestro ejemplo) es el **`getParameter(String)`** que permite consultar el valor de cada uno de los pares *clave/valor* que ha enviado el cliente a través del formulario

- El objeto **HTTPServletResponse** permite al Servlet enviar la respuesta al cliente que ha solicitado el servicio. Esta interfaz dispone de métodos que permiten obtener *Streams* con los que enviar al cliente la respuesta.

  El proceso de preparación de la respuesta consiste básicamente en lo siguiente:

  - Indicar el tipo de contenido MIME de la respuesta mediante el método **`setContentType()`**. En nuestro ejemplo, la respuesta es una página HTML y por tanto el contenido MIME es HTML:

    ```
    resp.setContentType("text/html");
    ```

  - Obtener un *Stream* hacia el cliente (*PrintWriter* cuando haya que mandar texto y *ServletOutputStream* para datos binarios). En nuestro ejemplo usaremos un *PrintWriter*:

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

  - Ir enviando la respuesta (sentencias HTML) a través del *PrintWriter* (mediante la sentencia de impresión **`println()`**):

    ```
    out.println("<title>Valores recogidos en el formulario</title>");
    ```

3.1.4.2.6.- Método getServletInfo

El método **getServletInfo()** proporciona datos acerca del Servlet (autor, fecha de creación, funcionamiento,...) al servidor web. No es en ningún caso obligatoria su utilización, aunque puede ser interesante cuando se tienen muchos Servlets funcionando en un mismo servidor y puede resultar complejo la identificación de los mismos.

3.1.4.2.7.- Interface ServletContext

Un Servlet vive y muere dentro delos límites del servidor web. Por este motivo, puede ser interesante en un determinado momento obtener información acerca del entorno en que se está ejecutando el Servlet. Todo Servlet tendrá a su disposición esta información a través del objeto **ServletContext**. Un Servlet puede obtener dicho objeto mediante el método **getServletContext()** aplicable al objeto **ServletConfig**.

3.1.5.- JSP (Java Server Pages)

Las páginas JSP son una tecnología usada en la capa de presentación de las aplicaciones web. Esta tecnología se sitúa por encima de la otra tecnología Java para la capa de presentación, los Servlets, permitiendo generar dinámicamente contenidos HTML de una forma más sencilla.

Las JSP nos permiten mezclar en una misma página contenidos HTML estáticos con código de tipo *Script* para la generación de contenidos dinámicos.

Sin embargo, la característica más destacada de esta tecnología es que permite separar muy claramente el código HTML estático de la parte encargada de la generación dinámica de contenidos. De esta forma, trabajar con JSP se reduce a escribir código HTML normal y corriente, usando cualquier herramienta de edición de páginas HTML, y a encerrar después el código encargado de la generación de contenidos dinámicos con unas etiquetas especiales que empiezan por <% y terminan por %>.

A pesar de que la apariencia de una página JSP es más parecida a una página HTML normal que a un Servlet, el funcionamiento interno de las páginas JSP se soporta a través de Servlets. De tal manera que una página JSP es convertida automáticamente a un Servlet, transformando las sentencias HTML de la página JSP a sentencias tipo `print()` en el Servlet y copiando el código *Script* (código Java) directamente en el código del Servlet.

La transformación de una página JSP a un Servlet se realiza generalmente la primera vez que ésta es invocada, siendo, por tanto, la primera invocación de una JSP bastante más lenta que las invocaciones posteriores.

La tecnología JSP se presenta como una puerta para el desarrollo de aplicaciones basadas en componentes. De tal manera que podemos desarrollar aplicaciones en las que la lógica de negocio esté soportada por componentes *JavaBeans* o *Enterprise JavaBeans* y la capa de presentación sea soportada por páginas JSP encargadas de generar dinámicamente páginas HTML que muestren los resultados generados por los componentes.

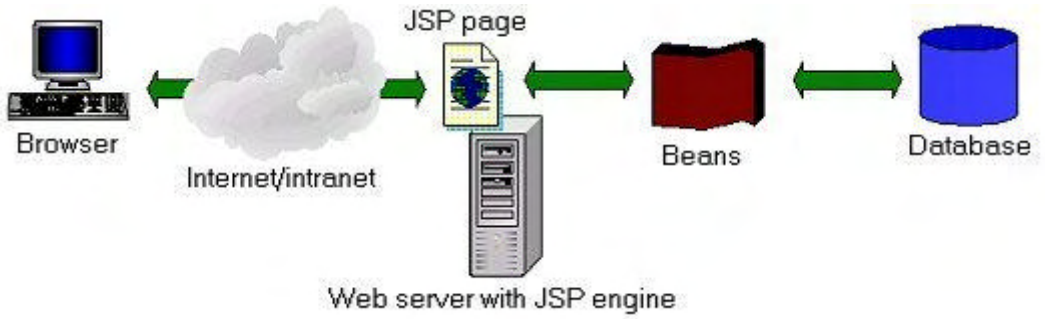

Figura 3.5: Arquitectura básica de una aplicación basada en JSP

La figura 3.5 muestra un modelo básico de aplicación basado en JSP. En ella se puede observar como el código de la página JSP interactúa con los componentes *JavaBeans* para que éstos realicen los procesos de negocio y luego la página JSP se encarga de mostrar los resultados mediante una mezcla de código HTML generado dinámicamente y código HTML estático.

A continuación se irán comentando algunos de los elementos más importantes que pueden aparecer en una página JSP:

- Código HTML estático.
- Elementos JSP de *Script*
- Directivas JSP
- Variables predefinidas
- Actions

3.1.5.1.- Código HTML Estático

El mayor porcentaje de código de una página JSP suele estar constituido por código HTML estático. El código HTML estático de una página JSP es código HTML normal y corriente que sigue las reglas marcadas por el lenguaje HTML y que será devuelto directamente al cliente que invoque la página. De hecho, este código puede ser generado directamente mediante herramientas de edición de páginas HTML.

3.1.5.2.- Elementos JSP de Script

Los elementos JSP de *Script* nos permiten insertar código Java dentro de la página JSP. Existen tres elementos de *Script* diferentes:

- **Declaraciones:** permiten definir variables globales o métodos Java que pueden ser accedidos desde cualquier sitio dentro de la página. Las declaraciones aparecen entre las etiquetas <%!...%>.
- **Scriptlets:** nos permiten insertar el código Java que se encarga de la lógica de presentación de la página JSP. El código Java de los *Scriptlets* aparece entre las etiquetas <%...%>.

- **Expresiones:** permiten mostrar de una manera sencilla los resultados producidos por expresiones Java. Las expresiones aparecen entre las etiquetas <%=...%>.

3.1.5.2.1.- Declaraciones

Las declaraciones nos permiten definir en una página JSP métodos y variables que podrán ser accedidos desde el código situado en cualquier otro punto de la misma página.

Lo habitual suele ser que los métodos se definan dentro de los componentes que se encargan de los procesos de negocio de la aplicación, sin embargo, en algunos casos es mejor definir ciertos métodos dentro de la propia página, sobre todo cuando la funcionalidad que implementan esos métodos sólo tenga sentido dentro de esa página y, por tanto, sólo vayan a ser invocados desde dentro de ella.

Las declaraciones de métodos y variables se escriben entre las etiquetas **<%!...%>**.

Es importante tener en cuenta que las declaraciones no generan ningún código de salida en las páginas JSP y, por tanto, su utilidad se fundamenta en el uso que hagan de ellas las *expresiones* y los *Scriptlets*, ya que son éstos los que en última instancia se encargarán de generar los resultados de las páginas JSP.

Ejemplo:

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

3.1.5.2.2.- Scriptlets

Los *Scriptlets* nos permitan insertar código Java en una página JSP. Aparecerán entre las etiquetas **<%...%>**.

El código Java que insertemos se encargará de todo lo relacionado con la lógica de presentación de la página y con la generación de contenidos HTML dinámicamente.

El código Java de un *Scriptlet* puede ser cualquier código que podamos usar en la programación Java tradicional, podemos usar cualquier clase del API de Java (que previamente hayamos importado mediante una directiva), acceder a los métodos y variables definidas mediante las declaraciones, invocar a componentes, realizar accesos a bases de datos, usar variables predefinidas, etc.

Ejemplo:

```
<% if( name.length() == 0 ){
     longitud = “cero”;
   }else{
     longitud = “más que cero”;
   }
%>
```

Como hemos comentado anteriormente, las páginas JSP se transforman internamente en Servlets. Puede ser interesante entender como se realiza esta transformación para comprender un poco mejor como se trata el código de los *Scriptlets*. Dos reglas:

1. El código Java del *Scriptlet* se inserta directamente en el código del Servlet.

2. El código HTML estático de la página JSP se inserta en el Servlet mediante sentencias `println()`.

Esto quiere decir que el código Java de un *Scriptlet* no tiene porque contener sentencias Java completas sino que podemos dejar una sentencia incompleta, insertar en medio de *Scriptlet* sentencias de código HTML estático y luego insertar otro *Scriptlet* que termine de completar la sentencia Java que dejamos incompleta anteriormente, siempre y cuando el código que resulte al traducir la página JSP a un Servlet sea correcto.

El siguiente código contiene mezclado código HTML estático con *Scriptlets* y muestra un ejemplo de lo comentado anteriormente.

```
<% if (Math.random() < 0.5){ %>
   Have a <B>nice</B> day!
<% }else{ %>
   Have a <B>lousy</B> day!
<% } %>
```

El código que se generará cuando se traduzca a un Servlet será totalmente correcto:

```
if (Math.random() < 0.5){
   out.println("Have a <B>nice</B> day!");
}else{
   out.println("Have a <B>lousy</B> day!");
}
```

De hecho, ésta es una técnica muy empleada para la generación dinámica de código HTML ya que nos permite decidir en tiempo de ejecución qué código HTML insertar en la página HTML resultado y cual no.

3.1.5.2.3.- Expresiones

Una expresión JSP permite insertar el resultado de una expresión Java directamente en la página HTML resultado. Las expresiones tienen el siguiente formato:

<%= Expresión Java %>

La expresión Java será evaluada en tiempo de ejecución, su resultado se convertirá en un *String* y éste será insertado dinámicamente en la página HTML resultado.

Las expresiones se suelen usar generalmente para visualizar en puntos concretos de la página JSP valores de variables o valores devueltos por métodos.

Ejemplos:

```
<H2>Welcome, <%= myBean.getName() %> </H2>

Current Time: <%= new java.util.Date() %>
```

Este último ejemplo mostrará en la página HTML resultado la fecha en la que la página JSP fue invocada. Observar, además, como las expresiones Java que se incluyen entre las etiquetas <%= y %> no llevan el punto y como al final de la línea.

3.1.5.3.- Directivas JSP

Las directivas JSP permiten configurar ciertos aspectos de las páginas JSP, afectando éstas a todo el contenido de la página. La sintaxis general de las directivas JSP es la siguiente:

<%@ directiva atributo=”valor” %>

En el caso de que una directiva tenga más de un atributo, la sintaxis usada será la siguiente:

```
<%@ directiva  atributo1=”valor1”
               atributo2=”valor2”
               ...
               atributoN=”valorN” %>
```

Existen dos tipos de directivas básicas:

- La **directiva PAGE** que permite configurar ciertos atributos de la página JSP como, por ejemplo, el lenguaje de *Script* a utilizar, el tipo de contenido generado por la página, las clases a importar, etc.
- La **directiva INCLUDE** que permite insertar en una página JSP el contenido de otros ficheros (como cabeceras o pies de página) que estarán almacenados independientemente. El contenido del fichero será insertado en la página JSP en el momento en que ésta sea transformada a su correspondiente Servlet.

3.1.5.3.1.- Directiva PAGE

De entre todos los atributos que nos permite definir la directiva PAGE hemos destacado los siguientes:

- **`import=“package.class”`** o bien **`import=“package.class1,..., package.classN”`**

  Este atributo nos permite indicar que clases vamos a importar en la página JSP. Una vez que tengamos importadas las clases, podremos usarlas en el código Java que insertemos dentro de la página JSP.

  Ejemplo:

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

  El código del ejemplo anterior nos permitirá usar dentro de la página JSP todas las clases del paquete `java.util`.

- **`contentType=”tipoMIME”`**

  Este atributo nos permite indicar el tipo de contenido que va a generar la página JSP. Normalmente las páginas JSP generan como resultado una página HTML y, por tanto, el tipo de contenido por defecto es “*text/html*”.

  Ejemplo:

  ```
  <%@ page contentType=”text/html” %>
  ```

- **`info="mensaje"`**

  Este atributo nos permite definir un String que identifique a la página JSP. Este valor puede ser consultado mediante el método `getServletInfo()`.

- **`errorPage="URL"`**

  Este atributo nos permite indicar la URL de la página de errorque será invocada cuando se produzca una excepción en el código de la página JSP que no haya sido capturada.

- **`isErrorPage="true|false"`**

  Este atributo permite indicar si la página actual puede o no actuar como página de error de otra página JSP. Por defecto su valor es "false".

- **`language="java"`**

  Este atributo nos permite indicar el tipo de lenguaje que vamos a usar como *Script* en la página JSP. Actualmente, el lenguaje por defecto y el único permitido es Java.

3.1.5.3.2.- Directiva INCLUDE

Esta directiva nos permite incluirel contenido de un fichero dentro de la página JSP. El contenido del fichero será insertado en el momento en que la página JSP es transformada en un Servlet. La sintaxis de esta directiva es la siguiente:

<%@ include file="URL" %>

La URL identifica el fichero a incluir dentro de la página. El contenido de este fichero es incrustado directamente como parte de la página JSP y, por tanto, puede contener cualquier elemento de los que habitualmente aparecen en una página JSP: HTML estático, elementos de *Script*, directivas y *actions*.

Esta directiva se suele usar cuando queremos generar páginas que contengan todas ellas un mismo encabezado o un mismo logotipo, ya que de esta forma no tenemos que incluir el código HTML del logotipo en todas las páginas JSP sino quelo tenemos guardado en un fichero aparte y lo incrustamos en todas las páginas mediante la directiva INCLUDE.

A continuación se muestra una página JSP que usa la directiva INCLUDE para incrustar como cabecera de todas las páginas HTML que genera, el logotipo que se encuentra en el fichero "logotipo.html".

```
<HTML>
   <HEAD>
      <TITLE>Ejemplo de la directiva INCLUDE</TITLE>
   </HEAD>
   <BODY>
      <%@ include file="/logotipo.html" %>
      ...
   </BODY>
</HTML>
```

Figura 3.6: Página JSP que usa la directiva INCLUDE

Es importante recordar que la directiva INCLUDE inserta el contenido del fichero en el momento en que la página JSP es convertida en un Servlet. Esto implica que si el logotipo cambia después de que la página ha sido transformada, los cambios no se verán reflejados. Por tanto, la directiva INCLUDE es muy eficiente en cuanto al tiempo de respuesta, pero sólo debe usarse en aquellos casos en que el contenido del fichero que se inserta no se vaya a modificar frecuentemente. En caso contrario, será necesario usar una *action* de tipo JSP:INCLUDE ya que ésta incluye el contenido del fichero cada vez que la página es invocada, de forma que es menos eficiente en cuanto al tiempo de respuesta pero más flexible a cambios.

3.1.5.4.- Ejemplo de uso de elementos Script y directivas

A continuación se muestra el código de una página JSP que hace uso de los elementos de *Script* y directivas vistas hasta este momento:

```
<HTML>
<HEAD>
<TITLE>Ejemplo de uso de elementos Script y directivas</TITLE>
</HEAD>
<BODY>
<%@ include file="cabecera.html" %>
<%@ page import="java.util.Date" %>
<%! int numVisitantes = 0; %>
<% numVisitantes++; %>

Eres el visitante número <%= numVisitantes %> de esta página.
<P>La hora en la que se te visualizó esta página fue <%= new Date() %><P>
Para generar esta página HTML hemos usado una JSP con los siguientes
elementos:
<UL>
   <LI><B>Una Directiva Include</B> que incrusta el título de esta página<BR>
   <LI><B>Una Declaración</B> que define una variable donde guardar el número
   de visitantes<BR>
   <LI><B>Un Scriptlet</B> que va incrementando el número de visitantes.<BR>
   <LI><B>Una Expresión</B> que visualiza el número de visitantes<BR>
   <LI><B>Una Directiva Page</B> que importa la clase java.util.Date<BR>
   <LI><B>Otra Expresión</B> que visualiza la hora en que se descargó la
   página<BR>
</UL>
</BODY>
</HTML>
```

Figura 3.6: Código de una página JSP que usa elementos de Script y directivas

```
<CENTER>
<TABLE WIDTH="50%" BORDER="5">
   <TR>
   <TD ALIGN="CENTER"><FONT SIZE="+3">Ejemplo JSP</FONT></TD>
   </TR>
</TABLE>
</CENTER><BR>
```

Figura 3.7: Código del fichero "cabecera.html"

Esta página JSP genera como resultado una página HTML que contiene como contenidos dinámicos el número de visitantes que tiene la página hasta ese momento y la fecha y hora en la que se realizó el

último acceso a esa página. La página HTML generada, además, mostrará al usuario los elementos JSP que fueron usados para generar sus contenidos dinámicos. La página HTML que se generará en nuestro ejemplo es la siguiente:

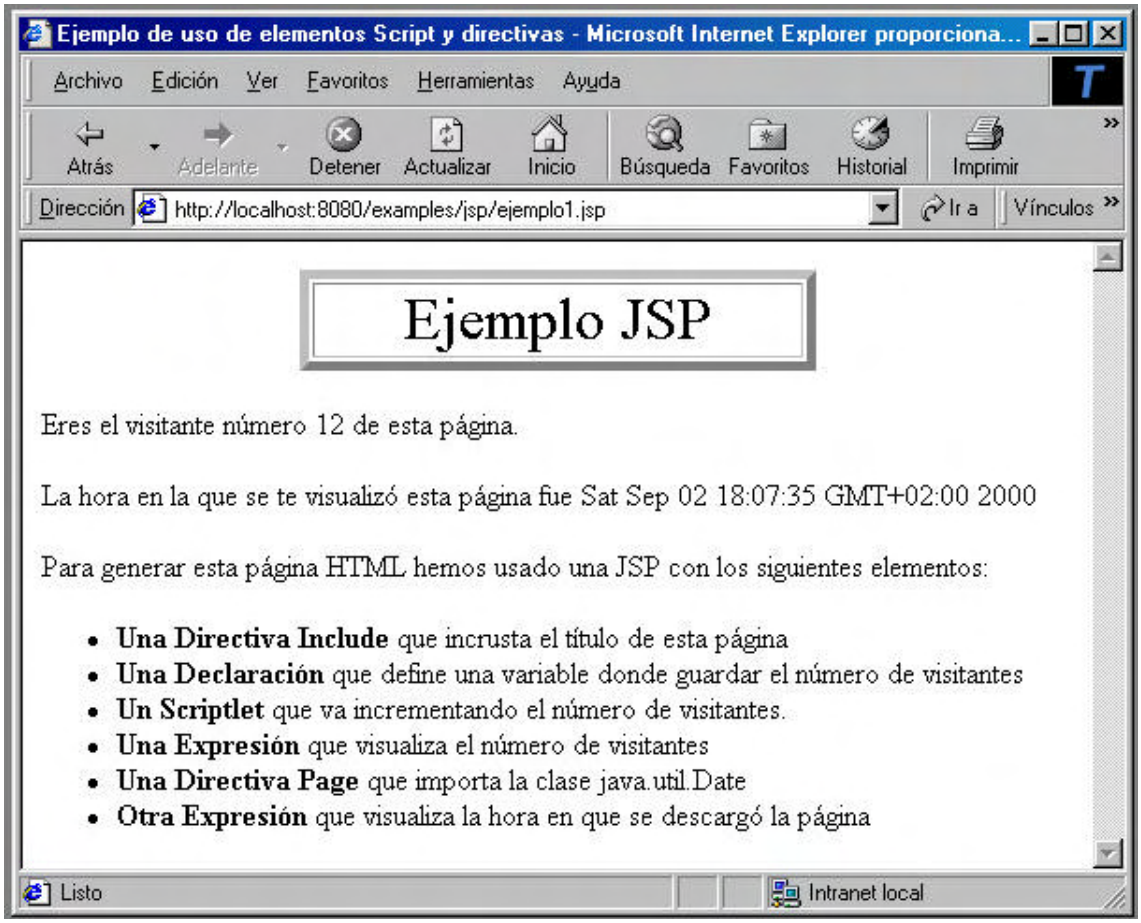

Figura 3.8: Página HTML generada por la página JSP del ejemplo.

3.1.5.5.- Variables Predefinidas

Las JSP definen una serie de *variables implícitas* que pueden ser usadas dentro de las *expresiones JSP* y de los *Scriptlets*.

Las más importantes son las siguientes:

- **OUT**: el objeto `out` pertenece a la clase `javax.servlet.jsp.JSPWriter` y nos permite, mediante su método `println()`, imprimir contenidos directamente desde dentro de los *Scriptlets*.

  Ejemplo:

  ```
  <% out.println(<H2>Uso del objeto out </H2>); %>
  ```

- **REQUEST:** el objeto `request` pertenece a la clase `javax.servlet.http.HttpServletRequest` y nos permite, a través de su método `getParameter()`, obtener los valores de los parámetros que le llegan a la página JSP a través de un formulario HTML.

Ejemplo:

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

- **RESPONSE:** el objeto `response` pertenece a la clase `javax.servlet.http.HttpServletResponse` y contiene una serie de métodos que nos permiten trabajar sobre la respuesta (página HTML) que genera la página JSP.

  Ejemplo:

  ```
  <% response.setContentType("text/html"); %>
  ```

  que sería equivalente a usar la directiva:

  ```
  <%@ page contentType="text/html" %>
  ```

Como se puede apreciar, todas estas variables se corresponden con los objetos que manipulábamos cuando trabajábamos con Servlets y, por tanto, su utilidad es la misma que en éstos.

3.1.5.6.- Ejemplo de uso de variables predefinidas

En este ejemplo veremos la interacción entre un formulario y una página JSP que recoge los datos del formulario, los procesa y genera una página HTML cuyos contenidos variarán en función de los datos que se hayan introducido en elformulario.

3.1.5.6.1.- El formulario

El formulario tiene el siguiente aspecto:

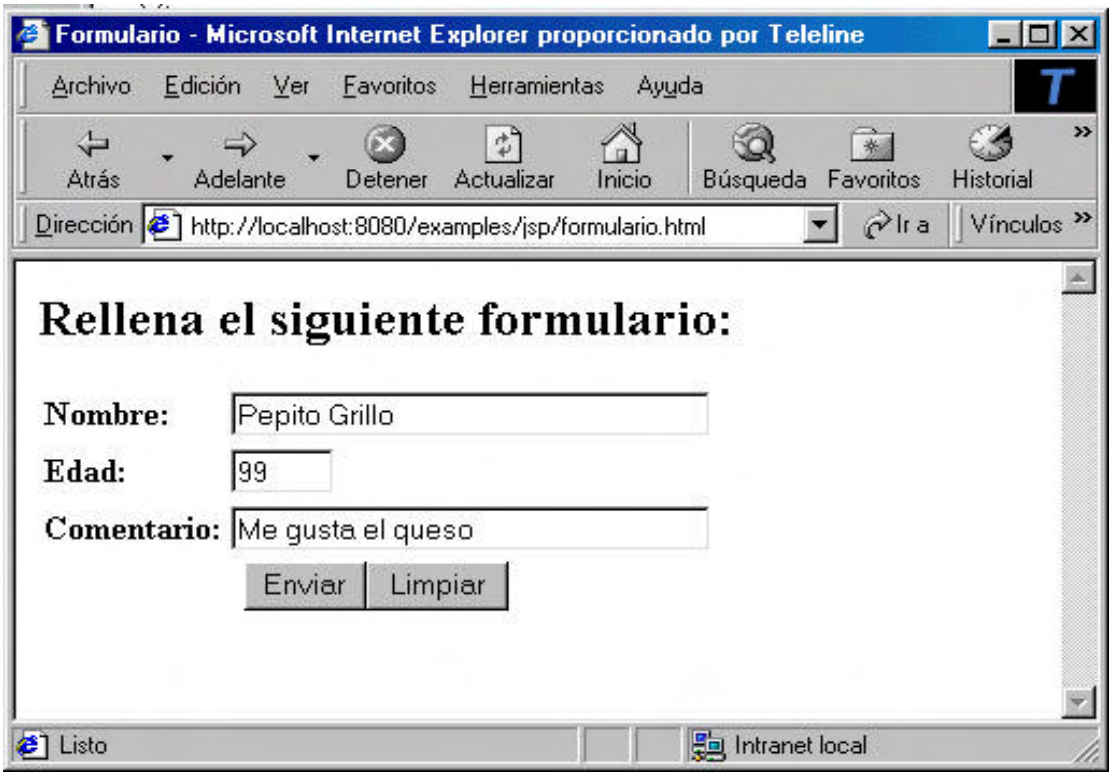

Figura 3.9: Formulario que debe rellenar el usuario

El código HTML que corresponde al formulario anterior es el siguiente:

```
<HTML>
  <HEAD>
    <TITLE>Formulario</TITLE>
  </HEAD>
  <BODY>
    <H2>Rellena el siguiente formulario: </H2>
    <FORM action="ejemplo2.jsp" method="GET">
    <TABLE WIDTH="50%">
      <TR>
        <TD WIDTH="90"><B>Nombre:</B></TD>
        <TD WIDTH="254">
        <P><INPUT TYPE="TEXT" NAME="nombre" SIZE="30"></P></TD>
      </TR>
      <TR>
        <TD WIDTH="90"><B>Edad:</B></TD>
        <TD WIDTH="254">
        <P><INPUT TYPE="TEXT" NAME="edad" SIZE="5"></P></TD>
      </TR>
      <TR>
        <TD WIDTH="90"><B>Comentario:</B></TD>
        <TD WIDTH="254">
        <P><INPUT TYPE="TEXT" NAME="comentario" SIZE="30"></P></TD>
      </TR>
      <TR>
        <TD COLSPAN="2" ALIGN="CENTER" WIDTH="344">
        <P><INPUT TYPE="SUBMIT" VALUE="Enviar">
       <INPUT TYPE="RESET" VALUE="Limpiar"></P></TD>
      </TR>
    </TABLE>
    </FORM>
  </BODY>
</HTML>
```

Figura 3.10: Código HTML del formulario

3.1.5.6.2.- La página JSP

La página JSP se encargará de validar los datos que se hayan introducido en el formulario. En primer lugar comprobará que el nombre y el comentario no se hayan dejado sin rellenar y en caso de que así sea generará un mensaje de error. En caso de que ambos se hayan rellenado, comprobará si el comentario se corresponde con una frase predefinida, la frase "Me gusta el queso", en cuyo caso visualizará un mensaje especial. En caso de que el comentario sea cualquier otro, simplemente visualizará un mensaje de agradecimiento.

```
<HTML>
<HEAD>
   <TITLE>Resultados</TITLE>
</HEAD>

<%!
   // No se permite que el nombre y el comentario queden en blanco con
   // lo que habrá que validarlo
   boolean validarEntrada(String nombre, String comentario){
      boolean resultado = true;
      // si el nombre o el comentario está en blanco, devolvemos false
```

```
     // para indicar que la entrada es invalida
     if (nombre.length() == 0){
        resultado = false;
     }
     if (comentario.length() == 0){
        resultado = false;
     }
     return resultado;
   }

   // Generamos un resultado en función de el comentario introducido en
   // en el formulario
   String getResultado(String comentario){
      String queso = "Me gusta el queso";
      String resultado;

      if (comentario.compareTo(queso) == 0)
         resultado = "¡A nosotros también nos gusta el queso!";
      else
         resultado = "Es una pena, esperamos que algún día te guste el queso.";

      return resultado;
   }
%>

<%
   // Obtenemos los datos introducidos en el formulario
   String nombre = request.getParameter("nombre");
   String edad = request.getParameter("edad");
   String comentario = request.getParameter("comentario");

   boolean esValida;
   esValida = validarEntrada(nombre,comentario);

   // Generamos una respuesta diferente en función de si el usuario
   // a dejado los campos nombre y comentario en blanco o no
   if (esValida){
%>

<H2>¡Gracias por tu información, <%= nombre %>!</H2>
<H3>

<%
     //Generamos el resultado en función del comentario
     out.println(getResultado(comentario));
   }else{
     out.println("Has dejado sin rellenar tu nombre o tu comentario.");
%>

</H3>
Por favor, <a href=formulario.html> rellenalo de nuevo</a>

<%
   }
%>
</BODY>
</HTML>
```

Figura 3.11: Código de la página JSP

De la página JSP anterior cabe destacar lo siguiente:

- Se ha usado una sección para las declaraciones (el código que está entre las etiquetas <%! y %>) donde se han definido una serie de métodos. Como se puede apreciar, estos métodos se han definido exactamente igual que como se definen los métodos de una clase Java normal y corriente. Estos métodos son invocados posteriormente en los *Scriptlets* de la página JSP.

- Se ha usado la expresión `<%= nombre %>` para visualizar en la página HTML resultado el nombre que introdujo el usuario.

- Se ha dividido una sentencia Java `if...else` entre dos *Scriptlets* diferentes. Esto es perfectamente valido tal y como se justificó anteriormente.

- Se ha usado el método `request.getParameter()` para acceder a los valores que el usuario introdujo en el formulario y se han asignado estos valores a variables temporales. Los datos que se introducen en el formulario llegan a la pagina JSP en el objeto `request` (en forma de pares clave/valor) y mediante el método `getParameter()` podemos consultar el valor de cada uno de ellos. Esta es la forma más habitual de recoger los datos de un formulario.

- En ciertas ocasiones se ha empleado el método `out.println()` para insertar código HTML en la página resultado.

3.1.5.6.3.- La página HTML resultado

A continuación se muestra la página HTML que generaría como resultado nuestra página JSP en caso de que los datos que se introdujeran en el formulario fueran los mismos que los que aparecen en la figura 3.9

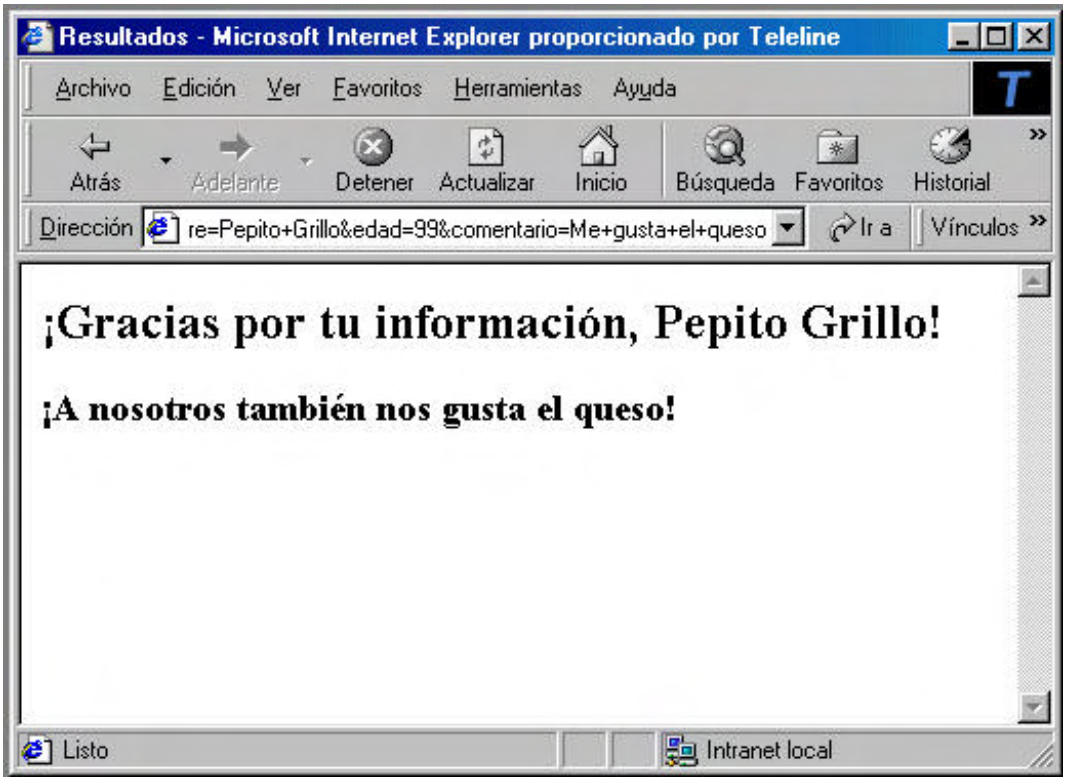

Figura 3.12: Página HTML generada por la página JSP

3.1.5.7.- Actions

La tecnología JSP dispone de unos elementos llamados *actions* que nos permiten dar funcionalidad a las páginas JSP sin necesidad de tener que usar código JAVA en forma de *Script*.

La utilidad que proporcionan las *actions* es muy variada, permiten desde insertar dinámicamente el contenido de un fichero en la página JSP o invocar otras páginas JSP hasta, y quizás la utilidad más importante, trabajar sobre componentes *JavaBeans*.

A continuación nos centraremos en algunas de las *actions* más destacadas:

3.1.5.7.1.- La Action JSP:Include

Esta *action* nos permite insertar el contenido de un fichero en una página JSP. Su sintaxis es la siguiente:

<jsp:include page="URL relativa" flush="true" />

La diferencia con la directiva INCLUDE radica en que esta *Action* inserta el contenido del fichero cada vez que la página JSP es invocada, y en cambio la directiva INCLUDE lo hacía en el momento en que la página JSP era traducida a un Servlet (una única vez por tanto).

Esto supone una pequeña penalización en el tiempo de respuesta ya que cada vez que la página es solicitada, se debe insertar el contenido de un fichero externo. Pero en cambio la flexibilidad frente a los cambios es mayor.

3.1.5.7.2.- Action JSP:UseBean

La *action* JSP:USEBEAN nos permite declarar que vamos a usar un determinado componente *JavaBean* en nuestra página JSP. Su sintaxis es la siguiente:

<jsp:useBean id="nombre" class="package.class" scope="page|request| session|application" />

Ejemplo:

```
<jsp:useBean id="myBeanInstance" class="com.myPackage.myBeanClass"
             scope="page" />
```

Como se puede observar, esta *action* admite tres atributos diferentes:

- El **atributo class** que indica la clase de *JavaBean* que vamos a usar.
- El **atributo id** que indica el nombre de la instancia que vamos a usar.
- El **atributo scope** que indica el ámbito de la instancia con la que vamos a trabajar.

Esta sentencia generalmente implica instanciar un nuevo bean de la clase especificada mediante el atributo **class**, asignarle a esa instancia el nombre indicado con el atributo **id** y asociarle el ámbito indicado con el atributo **scope**.

Sin embargo, en el caso de que ya exista instanciado un bean con el mismo nombre y el ámbito de ese bean le haga estar todavía en curso, esta sentencia no implicará crear una nueva instancia sino que por el contrario, implicará obtener una referencia al bean ya existente.

El atributo **scope** define el ámbito de un bean y nos permite crear beans cuyo ámbito englobe más de una página. Esto da lugar a que en muchas ocasiones la *action* JSP:USEBEAN sea usada para obtener referencias a beans ya existentes, en lugar de para instanciar nuevos beans. La *action* sólo implica la creación de un nuevo bean en el caso de que no exista ya un bean disponible con ese mismo **id**.

3.1.5.7.2.1.- Ambito de un Bean

El ámbito de un bean se indica mediante el **atributo scope** de la *action* JSP:USEBEAN e indica el contexto dentro del cual el bean está disponible (ciclo de vida del bean).

Existen cuatro tipos de ámbitos:

- **PAGE:** Es el ámbito por defecto e indica que el bean sólo estará disponible dentro de la página en la que fue creado.
- **REQUEST:** El bean solamente estará disponible durante la petición en curso.
- **SESSION:** El bean estará disponible para todas las páginas durante el tiempo de vida de una sesión de usuario.
- **APPLICATION:** El bean estará disponible para todas las páginas durante todo el tiempo que dure la ejecución de la aplicación.

Como ya se ha comentado, el ámbito de un bean tiene una gran importancia ya que además de delimitar el contexto en el que el bean está disponible, permite determinar si la *action* JSP:USEBEAN debe crear una nueva instancia del bean o por el contrario debe obtener la referencia a un bean ya existente.

Los beans de ámbito **page** y **request** se destruyen una vez que la petición del cliente ha sido servida y por tanto, la *action* JSP:USEBEAN deberá crear una nueva instancia del bean de tipo **page** o **request** para cada nueva petición.

En cambio, los beans de ámbito **session** se mantienen durante todo el tiempo que dura una sesión de usuario y por tanto, la *action* JSP:USEBEAN solo deberá crear una nueva instancia del bean en el caso de que este no exista todavía para la sesión en curso; en caso de que ya exista, simplemente obtendrá una referencia al bean ya existente.

3.1.5.7.2.2.- JSP:UseBean con cuerpo de inicialización

Existe otra forma de usar la *action* JSP:USEBEAN que permite añadir un cuerpo de inicialización:

```
<jsp:useBean id="nombre" class="myClass" scope="page|...|application">
        {Cuerpo de Inicialización}
</jsp:useBean>
```

El cuerpo de inicialización que se sitúa entre las etiquetas **<jsp:useBean>** y **</jsp:useBean>** sólo será ejecutado en el caso en que se cree un nuevo bean, y no en el caso en que se use un bean ya existente.

El cuerpo de inicialización nos permitirá inicializar las propiedades del bean mediante la *action* JSP:SETPROPERTY.

3.1.5.7.3.- Action JSP:SetProperty

La *action* JSP:SETPROPERTY nos permite asignar valores a las propiedades de los beans. Su sintaxis es la siguiente:

```
<jsp:setProperty    name="myName"    property="someProperty"
                    value="someValue"    param="someParam" />
```

El significado de sus cuatro atributos es el siguiente:

- **NAME**: Este atributo es obligatorio e identifica al bean cuyas propiedades van a ser modificadas. Se debe corresponder con el atributo **id** de alguna *action* de tipo JSP:USEBEAN usada anteriormente.

- **PROPERTY**: Este atributo es obligatorio e identfica la propiedad del bean que va a ser modificada. También se le puede asignar el valor "*" que es un valor comodín que será explicado posteriormente.

- **VALUE**: Es un atributo opcional que indica el valor a asignar a la propiedad. El valor de este atributo siempre aparece en forma de *String*. La conversión del tipo *String* al tipo específico de la propiedad se realizará automáticamente.

- **PARAM**: Es un atributo opcional que identifica qué parámetros de los que llegan a la página JSP provenientes de un formulario (llegarán en la variable **request**) serán asignados a la propiedad del bean.

  En caso de que no exista ningún parámetro con ese nombre, no se modificará el valor de la propiedad (¡Ojo!, el valor de la propiedad no se pone a *null* sino que se deja tal y como estaba).

Por ejemplo, la siguiente sentencia indica que a la propiedad "numberOfItems" del bean llamado "OrderBean" se le debe asignar el valor del parámetro llamado "numItems" que llegará procedente del formulario que llama a la página JSP. En caso de que ese parámetro no existiera, la propiedad "numberOfItems" no se modificaría.

```
<jsp:setProperty name="OrderBean" property="numberOfItems" param="numItems" />
```

En caso de que se omitan los atributos **value** y **param**, el comportamiento será el mismo que si se proporcionara un atributo **param** con el mismo nombre que el de la propiedad. De esta forma, se asignaría a la propiedad el valor del parámetro que tenga su mismo nombre.

Muy relacionado con lo anterior está el uso del valor "*" en el atributo **property**. En este caso lo que se hará es ir emparejando todas las propiedades del bean con los parámetros que tengan su mismo nombre, de forma que a todas las propiedades a las que se las consiga emparejar con algún parámetro se les asignará su valor.

La *action* JSP:SETPROPERTY puede ser usada en dos contextos diferentes:

- Se puede usar después de una *action* de tipo JSP:USEBEAN, pero fuera de ella:

```
<jsp:useBean id="myName" ... />
        ...
<jsp:setProperty name="myName" property="someProperty" .../>
```

En este caso, la *action* JSP:SETPROPERTY se ejecutará siempre, independientemente de si la *action* JSP:USEBEAN crea un nuevo bean "myName" o referencia a uno ya existente.

- También se puede usar dentro del cuerpo de inicialización de una *action* JSP:USEBEAN:

```
<jsp:useBean id="myName" ... >
        ...
        <jsp:setProperty name="myName" property="someProperty" .../>
        ...
</jsp:useBean>
```

En este caso, la *action* JSP:SETPROPERTY sólo se ejecutará en caso de que la *action* JSP:USEBEAN precedente haya creado una nueva instancia llamada "myName" y no en el caso de que haya encontrado una ya existente.

Como ya se ha comentado anteriormente, los valores que se asignan a las propiedades de los beans pueden provenir de dos fuentes distintas: de un valor que le dé el programador explícitamente (mediante el **atributo value**), o de los parámetros que llegan procedentes de un formulario (mediante el **atributo param**). A continuación mostraremos un ejemplo de cada tipo.

Ejemplo 1:

```
<jsp:useBean id="myBeanInstance" class="com.package.myBeanClass"
             scope="request" >

     <jsp:setProperty  name="myBeanInstance" property="myProperty"
                       value="123" />

</jsp:useBean>
```

Ejemplo 2:

```
<jsp:useBean id="myBeanInstance" class="com.package.myBeanClass"
             scope="request" >

     <jsp:setProperty  name="myBeanInstance" property="myProperty"
                       param="myFormElementName" />

</jsp:useBean>
```

Obviamente, no tiene sentido usar juntos en una misma *action* el **atributo value** y el **atributo param**.

3.1.5.7.4.- Action JSP:getProperty

La *action* JSP:GETPROPERTY permite obtener el valor de una propiedad de un bean, convertirlo a un *String* e insertarlo en la página HTML de salida generada por la página JSP. Su sintaxis es la siguiente:

<jsp:getProperty name="myName" property="myProperty" />

- El **atributo name** identifica el bean sobre el que se va a consultar las propiedades. Deberá ser el nombre de un bean referenciado anteriormente por una *action* de tipo JSP:USEBEAN.

- El **atributo property** identifica la propiedad del bean de la que se va a obtener su valor.

Ejemplo:

```
<jsp:useBean id="myBeanInstance" class="com.package.myBeanClass"
             scope="request" />
<H2>My Prop: <jsp:getProperty name="myBeanInstance" property="myProp"/></H2>
```

Como se puede observar, estamos mezclando código HTML estático con la *action* JSP:GETPROPERTY para generar dinámicamente contenidos HTML.

3.1.5.7.5.- Propiedades de los JavaBeans

Los *JavaBeans* son un tipo de componentes de la plataforma Java que, como todos los componentes pertenecientes a un modelo, cumplen una serie de características propias del modelo al que pertenecen.

De entre todas las características que deben tener los JavaBeans nos centraremos únicamente en aquellas que se necesitan para su uso en páginas JSP, que es todo lo relativo al tratamiento de sus propiedades.

Todas las propiedades de un JavaBean deben tener una **pareja de métodos get/set** que permitan consultar y modificar los valores de esas propiedades. Así, si decimos que un JavaBean tiene una **propiedad llamada xxx de tipo Y**, esto significa que el JavaBean tiene un **método llamado getXxx() que devuelve un objeto** (o tipo primitivo) **de tipo Y** y que permite consultar el valor de la propiedad xxx, y otro **método llamado setXxx() que recibe como parámetro un objeto** (o tipo primitivo) **de tipo Y** y que permite modificar el valor de la propiedad xxx.

Ejemplo de propiedad "*nombre*" de tipo *String*:

```
String getNombre();
void setNombre(String n);
```

De hecho, el nombre de las propiedades en los JavaBeans quedan definidos mediante los pares de métodos get/set y no mediante los atributos que puedan tener asociados a esas propiedades.

Así, podemos tener un JavaBean con la propiedad "*nombre*", sin que necesariamente tengamos que tener un atributo que se llame "*nombre*", tal y como se demuestra en este ejemplo:

```
public class PersonaBean{

   String n;

   public String getNombre(){
      return n;
   }
   public void setNombre(String n){
      this.n = n;
   }
}
```

Figura 3.13: JavaBean "PersonaBean" con su propiedad "nombre"

Este JavaBean tendrá una propiedad llamada "*nombre*" (tal y como definen sus métodos) y no una propiedad llamada "*n*".

Desde un JSP podremos consultar y modificar los valores de las propiedades de un bean, bien mediante las *actions* JSP:GETPROPERTY y JSP:SETPROPERTY, o bien invocando directamente a los métodos get/set del bean desde un Scriptlet o desde una expresión. La forma más elegante es mediante el uso de las *actions*.

3.1.5.7.6.- Ejemplo del uso de JavaBeans en una página JSP

Este ejemplo muestra el uso de componentes *JavaBeans* en una página JSP. El ejemplo consistirá en una página JSP que, haciendo uso de un componente *JavaBean*, calculará una serie de números primos en base a una serie de características indicadas por el usuario a través de un formulario.

3.1.5.7.7.- El formulario

Se mostrará al usuario un formulario donde se le pedirá que introduzca cuántos números primos quiere que se calculen y a partir de qué número quiere que se comiencen a calcular los números primos. Este es el aspecto del formulario:

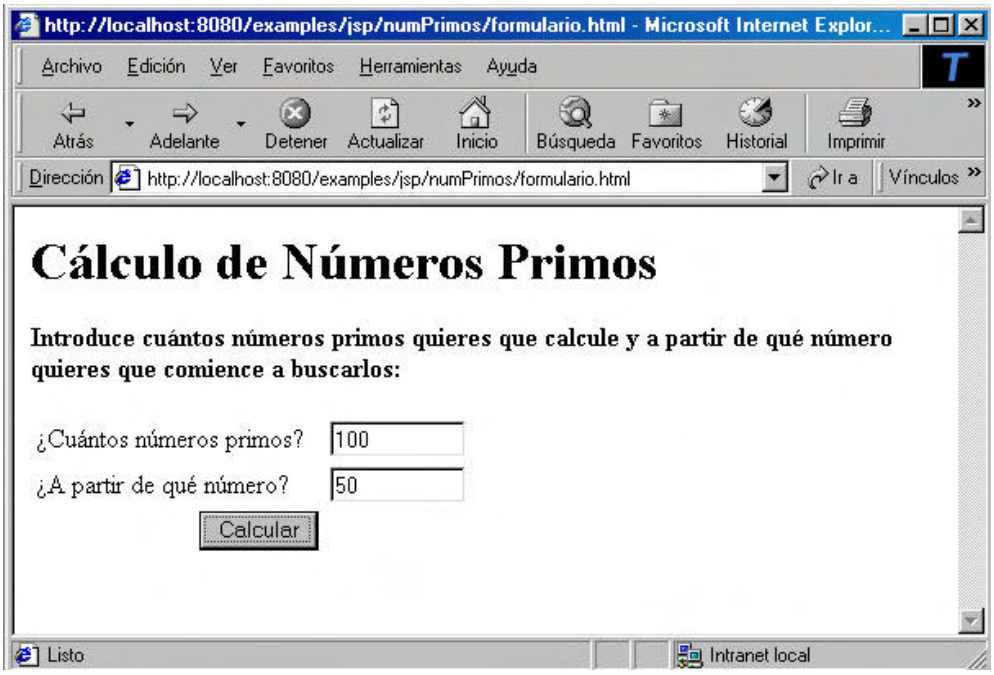

Figura 3.14: Formulario a rellenar por el usuario

El código HTML que define este formulario es el siguiente:

```
<HTML>
  <HEAD>
    <TITLE></TITLE>
  </HEAD>
  <BODY>
    <H1>Cálculo de Números Primos</H1>
    <P><B>Introduce cuántos números primos quieres que calcule y
      a partir de qué número quieres que comience a buscarlos:</B>
    </P>
```

```
    <FORM ACTION="numPrimos.jsp" METHOD="GET">
    <TABLE WIDTH="50%">
      <TR>
        <TD WIDTH="180">¿Cuántos números primos?</TD>
        <TD WIDTH="93">
        <P><INPUT TYPE="TEXT" NAME="cuantosPrimos" SIZE="10"></P></TD>
      </TR>
      <TR>
        <TD WIDTH="180">¿A partir de qué número?</TD>
        <TD WIDTH="93">
        <P><INPUT TYPE="TEXT" NAME="aPartirNumero" SIZE="10"></P></TD>
      </TR>
      <TR>
        <TD COLSPAN="2" ALIGN="CENTER" WIDTH="273">
        <P><INPUT TYPE="SUBMIT" VALUE="Calcular"></P></TD>
      </TR>
    </TABLE>
    </FORM>
  </BODY>
</HTML>
```

Figura 3.15: Código HTML del formulario

3.1.5.7.7.1.- El componente JavaBean

El componente *JavaBean* usado en nuestro ejemplo se encargará de almacenar la información que el usuario ha introducido en el formulario y en base a ella realizará el cálculo de los números primos. Este es el código del componente *JavaBean*:

```
package numPrimos;

public class PrimosBean{

   String listaPrimos;  // La lista de numeros primos generada
   int cuantosPrimos;   // El numero de primos a generar
   int desdeNumero;     // A partir de que número se generan los primos

   public void setCuantosPrimos(int num){
      this.cuantosPrimos = num;
   }

   public void setDesdeNumero(int num){
      this.desdeNumero = num;
   }

   public String getListaPrimos(){
      return listaPrimos;
   }

   public int getCuantosPrimos(){
      return cuantosPrimos;
   }

   public int getDesdeNumero(){
      return desdeNumero;
   }

   public void calcularNumerosPrimos(){
   // Algoritmo que calcula los números primos
```

```
     listaPrimos = "";
     int primo = desdeNumero;
     for (int i=0;i<cuantosPrimos;){
        int divisor = 2;
        boolean noEncontrado = true;
        while(noEncontrado && (divisor < primo)){
           if (primo%divisor == 0){
              noEncontrado = false;
           }else{
              divisor++;
           }
        }
        if (noEncontrado){
           listaPrimos = listaPrimos + "  " + primo;
           i++;
        }
        primo++;
     }
   }
}
```

Figura 3.16: Código del JavaBean PrimosBean

Este *JavaBean* contiene tres propiedades, tal y como se puede deducir al observar las parejas de métodos *get/set* que implementa:

- Propiedad **cuantosPrimos**
- Propiedad **listaPrimos**
- Propiedad **desdeNumero**

3.1.5.7.7.2.- La página JSP

Este es el código de la página JSP:

```
<HTML>
<HEAD>
   <TITLE>Resultados</TITLE>
</HEAD>
<BODY>
   <jsp:useBean id="bean" class="numPrimos.PrimosBean" scope="page" />
   <jsp:setProperty name="bean" property="cuantosPrimos" />
   <jsp:setProperty name="bean" property="desdeNumero" param="aPartirNumero" />

   <% bean.calcularNumerosPrimos(); %>
   <H1>Resultados Obtenidos</H1>
   <P><B>Estos son los <jsp:getProperty name="bean" property="cuantosPrimos"/>
   primeros números primos que existen a partir del número
   <jsp:getProperty name="bean" property="desdeNumero" />:</B></P>

   <P><jsp:getProperty name="bean" property="listaPrimos" /></P>

   <P><A HREF="formulario.html">Volver al formulario</A></P>
</BODY>
</HTML>
```

Figura 3.17: Código de la página JSP

De la página JSP anterior cabe destacar lo siguiente:

- Se han usado las *actions* JSP:USEBEAN, JSP:SETPROPERTY y JSP:GETPROPERTY para crear el Bean y para almacenar yrecuperar los valores de sus propiedades.

- Se ha usado el *Scriptlet* `<% bean.calcularNumerosPrimos() %>` para acceder a los métodos de negocio del Bean. De esta forma, la página JSP delega la responsabilidad del cálculo de los números primos en el componente*JavaBean.*

3.1.5.7.7.3.- La página HTML resultado

A continuación se muestra la página HTML que generaría como resultado nuestra página JSP en caso de que los datos que se introdujeran en el formulario fueran los mismos que los que aparecen en la figura 3.14

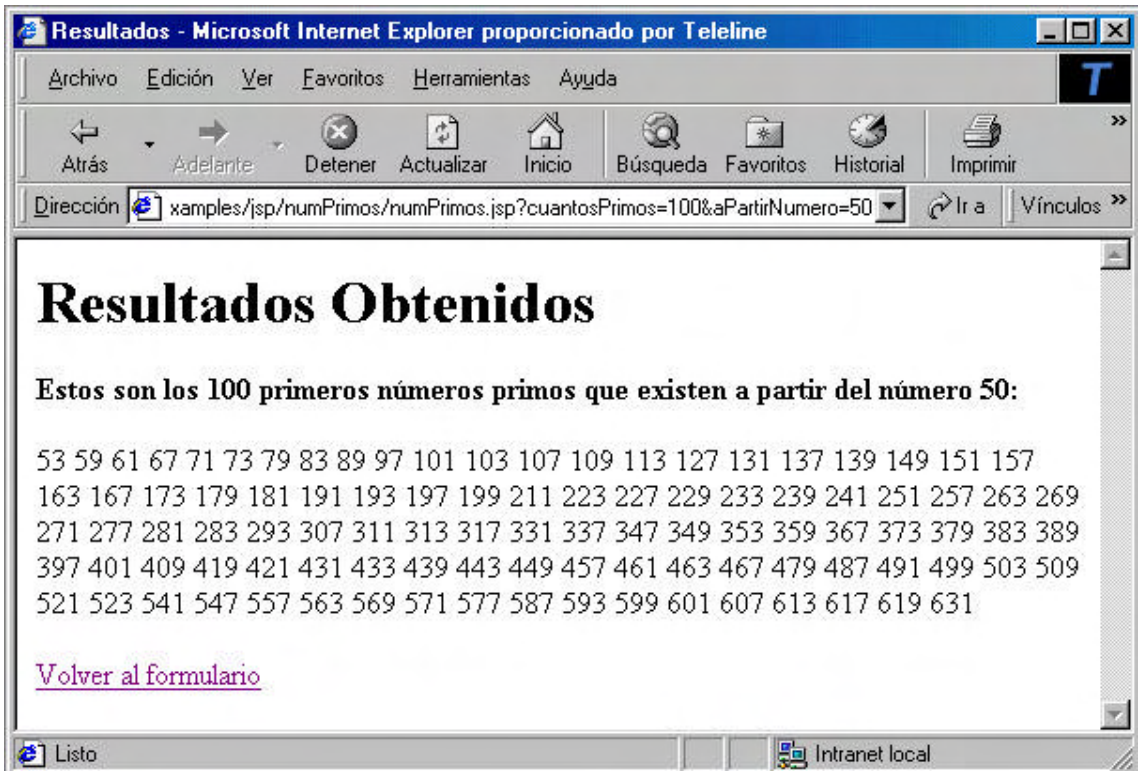

Figura 3.18: Página HTML resultado generada por la página JSP

3.1.5.7.8.- Action JSP:Forward

La *action* JSP:FORWARD permite redirigir una petición desde una página JSP hacia otra. Posee un único **atributo page** que contiene la URL relativa de la página hacia la que se redirige la petición. Esta URL puede ser tanto un valor estático como un valor obtenido en tiempo de ejecución.

Ejemplo:

```
<jsp:forward page="/utils/errorReporter.jsp" />
<jsp:forward page="<%= someJavaExpresion %>" />
```

Cuando se usa la *action* JSP:FORWARD, el control pasa a la página redireccionada y éste ya no vuelve a la página de origen.

3.1.5.7.9.- Comentarios JSP

Los comentarios en JSP se indican de la siguiente forma:

```
<%--
        Comentario –
%>
```

Todo lo que se incluya entre las etiquetas **<%--** y **--%>** será ignorado por el intérprete JSP y no se ejecutará.