3

Java Servlets

TABLA DE CONTENIDOS

PROGRAMACIÓN DE SERVLETS SENCILLOS

A continuación se expone un ejemplo sencillo de servlet, que servirá como toma de contacto con esta tecnología:

```
package cap02;

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

public class HelloWorldServlet extends HttpServlet
{
    public void service(      HttpServletRequest request,
                        HttpServletResponse response)
                              throws ServletException,
                              IOException
    {
           response.setContentType("text/html");

           PrintWriter pw = response.getWriter();
           pw.println("<html>");
           pw.println("<head>");
           pw.println("</head>");
           pw.println("<body>");
           pw.println("<h3>Viva Java!</h3>");
           pw.println("</body>");
           pw.println("</html>");
    }
}
```

Para crear un servlet es necesario implementar una clase que extienda a HttpServlet (perteneciente a javax.servlet.http) y sobrescribir los métodos doGet(), doPost() o ambos. En este caso redefinimos el método service (HttpServletRequest request, HttpServletResponse response). El método service() se ejecutará automáticamente cada vez que el servlet tenga que atender una petición de servicio. La clase HttpServlet no es abstracta, luego no sería necesaria la reescritura del método service() ya que cuenta con una implementación propia. En la práctica, no se suele sobrescribir el método service(), sino uno de sus métodos más especializados, ya sea doGet() o doPost(), en función si queremos extraer los datos de un formulario en cuyo código html se ha definido con el método GET (METHOD="GET") o POST respectivamente.

El primer argumento del método service(), de tipo HttpServletRequest, es un objeto que hace referencia a toda la información proveniente de la solicitud, es decir del cliente. Por lo tanto, a partir de este objeto podemos obtener información relacionada con el cliente; por ejemplo el navegador que usa e información acerca de su ordenador; también, si la petición proviniese del envío de un formulario HTML se podrían obtener los valores de los campos de éste. Del mismo modo, el segundo argumento del método, de tipo HttpServletResponse, define un objeto que nos permite establecer una comunicación en sentido contrario, desde el servidor hacia el cliente, pudiendo por lo tanto dar respuesta a la solicitud del usuario.

En este caso nuestro método service es el responsable de construir una página HTML que muestra el Titulo "Viva Java". Para ello, lo primero que hacemos es establecer a partir del HttpServletResponse el tipo de respuesta que recibirá el navegador del usuario por medio del método SetContentType (String str). Para indicar que se trata de una página web usamos el tipo "text/html". Posteriormente obtendremos a partir del mismo objeto un PrintWriter en donde iremos escribiendo los datos que queremos que el cliente reciba. Mediante el método println de PrintWriter se escribirán aquellas líneas que recibirá el navegador para visualizar en la pantalla del cliente. Como en este caso no necesitamos ninguna información del cliente, no utilizaremos para nada el parámetro HttpServletRequest.

La API Servlet es una API de extensión estándar, lo que significa que no forma parte del núcleo de la plataforma Java. Todas las interfaces y clases necesarias a la hora de programar Servlets se incluyen en los paquetes javax.servlet y javax.servlet.http que importamos en la cabecera de nuestro fichero de ejemplo.

EL PAQUETE JAVAX.SERVLET

Destacamos como más importantes las siguientes interfaces y clases:

- **Interfaces:**
  - Servlet → Define métodos que todos los Servlets deben implementar. Para implementar esta interface existen 2 opciones. Se puede escribir un servlet genérico que extienda a javax.servlet.GenericServlet o un servlet HTTP que extienda a javax.servlet.http.HttpServlet. Esta interface define métodos para inicializar el servlet (init()), para atender las peticiones de servicio (service()), y para eliminar el servlet del servidor (destroy()).
  - ServletRequest → Encapsula la solicitud de servicio de un cliente. Define una serie de métodos destinados a obtener información de la solicitud. Se puede obtener información adicional referente al protocolo específico (Por ejemplo, los datos HTTP, como la información concerniente a las cabeceras HTTP, se obtienen por medio del objeto HttpServletRequest que veremos más adelante.
  - ServletResponse → Este objeto lo utiliza el servlet para poder responder a una solicitud enviando información al solicitante. Para enviar un flujo de datos binarios como respuesta, empleamos el método getOutputStream() que nos devuelve un objeto ServletOutputStream. Para enviar datos de texto se usa el objeto PrintWriter, que es devuelto por getWriter(). Mediante el método setContentType fijamos el tipo de contenido de la respuesta. Si no se define ninguno juego de caracteres se toma por defecto el ISO-8859-1.
  - ServletConfig → La utiliza el servidor para meter la información referente a la configuración del servlet. El servlet, normalmente en su método init, utiliza los métodos de este objeto para recuperar esa información.
  - ServletContext → Define un conjunto de métodos que usa un servlet para comunicarse con su contenedor de Servlets; por ejemplo, para atender peticiones, obtener el tipo MIME de un fichero o información sobre el entorno en el que se está ejecutando. Existe un contexto para cada aplicación web (Una aplicación web es una colección de Servlets contenidos bajo la raíz de una URL concreta).Se accede al objeto ServletContext a través del objeto ServletConfig.

- **Clases:**
  - GenericServlet → Implementa la interfaz Servlet. El programador puede extender esta clase para definir sus propios Servlets. Pero normalmente, extiende HttpServlet que es una subclase de GenericServlet.

- ServletInputStream → Se utiliza para acceder a la información que suministra un cliente Web cuando hace la petición. 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 JVAX.SERVLET.HTTP

Se usa para definir Servlets específicos para el protocolo http. Las interfaces y clases más importantes de este paquete son las siguientes:

- **Interfaces:**
  - HttpServletRequest → Extiende la interfaz ServletRequest y agrega métodos para acceder a los detalles de una solicitud HTTP.
  - HttpServletResponse → Extiende la interfaz ServletResponse y agrega constantes y métodos para devolver respuestas específicas HTTP.
  - HttpSession → Define una forma de identificar a un usuario a través de las peticiones de más de una página o la visita de un sitio web y almacena la información acerca del usuario. El contenedor de Servlets usa esta interface para crear una sesión entre el cliente HTTP y el servidor HTTP, ya que éste carece de estado. La sesión persiste por un periodo de tiempo especificado, a lo largo de más de una conexión o petición de una página web por parte del usuario. El servidor puede mantener la sesión de distintas formas, como por ejemplo por medio del uso de cookies o la reescritura de URL's.

- **Clases:**
  - HttpServlet → Extiende a la clase GenericServlet con el fin de utilizar las interfaces HttpServletResponse y HttpServletRequest. Esta clase es la que normalmente extiende el programador.

JAKARTA-TOMCAT

Es uno de los proyecto de código abierto publicados por Apache Software Fundation. Es una aplicación web basada en Java y creada para ejecutar Servlets y páginas JSP. Más concretamente, Tomcat es un contenedor de Servlets con un entorno JSP. Un contenedor de Servlets es un shell de ejecución que maneja e invoca Servlets por cuenta del usuario.

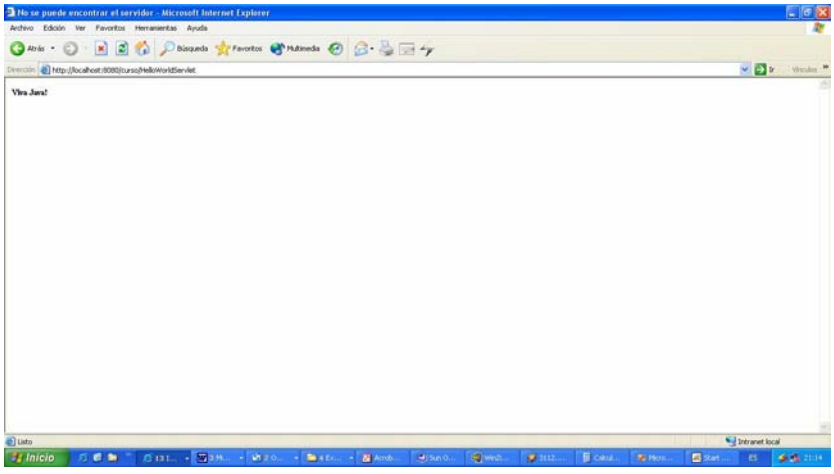
Tomcat puede utilizarse como contenedor de Servlets en solitario para fines de desarrollo y depuración o bien como plugin externo para un servidor web ya existente. En este nuestro primer ejemplo nos centraremos en el primer caso, con el fin de probar el Servlet que acabamos de desarrollar.

Lo primero sería descargarse Tomcat (*http://jakarta.apache.org/*) para después ejecutar el programa de instalación de Tomcat en nuestro equipo. En capítulos posteriores se profundizará en la estructura de directorios y configuración de Tomcat. Por el momento nos limitaremos a aquellos pasos necesarios para lanzar nuestro servlet por medio de una petición de servicio que llevará a cabo el navegador.

Tomcat por defecto permanece a la escucha en el puerto 8080 del ordenador en donde esté instalado. De esta forma si se introduce como URL en el navegador la dirección: http://localhost:8080/*direcciónWeb* lo que hará Tomcat será lanzar el servlet que haya asignado a dicha dirección. Los pasos a seguir serán los siguientes:

1. Crear el directorio $TOMCAT_HOME$/webapps/curso/WEB-INF/classes/cap02
2. Compilar el fichero fuente y volcar el fichero cap02/HelloWorldServlet.class obtenido al directorio $TOMCAT_HOME$/webapps/curso/WEB-INF/classes/cap02/
3. Compilar el fichero fuente y volcar el fichero cap02/HelloWorldServlet.class obtenido al directorio $TOMCAT_HOME$/webapps/curso/WEB-INF/classes/cap02/
4. Añadir el fichero web.xml al directorio $TOMCAT_HOME$/webapps/curso/WEB-INF. (El fichero web.xml es un fichero xml que almacena la configuración para los distintos contextos de Tomcat junto con el fichero server.xml referente a la configuración global). En el fichero web.xml existe una directiva que indica que cuando se reciba una petición para ejecutar HelloWorldServlet se ejecute el fichero compilado HelloWorldServlet.class del directorio cap02:
5. Se lanza el servidor Tomcat a la escucha mediante el icono resultante de la instalación "Start Tomcat".
6. Se introduce en el navegador la URL: *http://localhost:8080/curso/HelloWorldServlet*

```
…
        <servlet>
          <servlet-name>HelloWorldServlet</servlet-name>
          <servlet-class>cap02.HelloWorldServlet</servlet-class>
    </servlet>
…
```



ESTRUCTURA DE ARCHIVOS DE UNA APLICACIÓN WEB

Antes de nada es necesario conocer el significado de aplicación Web. Ésta es un conjunto de clases Java, Servlets, páginas JSP, un descriptor de despliegue (web.xml), páginas estáticas generadas por código HTML, XHTML, imágenes… y otros recursos que pueden agruparse y ejecutarse en distintos servidores que no tienen porque ser del mismo proveedor. Una aplicación Web, sería pues la capa Web de cualquier aplicación.

Con el fin de evitar colisiones entre los recursos de aplicaciones Web distintas, se define un importante objeto conocido como ServletContext. Se asocia un único ServletContext a cada aplicación Web.

Por lo tanto, el contenedor (Tomcat) que aloja una aplicación Web no es más que una jerarquía de directorios donde almacenar y organizar todos aquellos ficheros que componen una aplicación Web cualquiera. Por ello, el primer paso en el desarrollo de una aplicación Web es la creación de la jerarquía de directorios que albergará los componentes que conforman está. Tomando como raíz de dicha jerarquía el directorio webapps alojado en la carpeta de instalación de Tomcat, al cual describiremos por $TOMCAT_HOME, la estructura de directorios a partir de ese punto ($TOMCAT_HOME$/webapps) debería tener el siguiente aspecto para una aplicación Web llamada MiPrimeraAplicacionWeb:

- **/MiPrimeraAplicacionWeb:** Éste es el directorio raíz de la aplicación Web. En él se almacenarán todos los archivos HTML y JSP que requiera la aplicación. Se podrían crear otro subdirectorios adicionales para guardar otros recursos estáticos necesarios. En este directorio se almacenan pues todos aquellos recursos que son de acceso público para el cliente.

- **/MiPrimeraAplicacionWeb/WEB-INF:** Directorio que almacena los recursos que no deben ser de acceso público. Ninguno de los archivos que contenga este directorio podrán ser enviados directamente al cliente por el servidor Web. Aquí se guarda el fichero web.xml, nuestro descriptor de despliegue que establece la configuración de la aplicación Web.

- **/MiPrimeraAplicacionWeb/WEB-INF/classes:** Contiene los Servlets compilados además de aquellas clases java que dependan de los Servlets.

- **/MiPrimeraAplicacionWeb/WEB-INF/lib:** Alberga los ficheros jar de los que dependa la aplicación Web. Si por ejemplo necesitáramos acceder a bases de datos por medio de un driver JDBC, aquí almacenaríamos los ficheros comprimidos .jar que contienen el driver.

EL DESCRIPTOR DE DESPLIEGUE

Como se ha comentado antes, el descriptor de despliegue no es más que un fichero que determina la configuración de una aplicación web en particular. Su contenido está escrito en XML. Tomemos como ejemplo el fichero web.xml del capítulo anterior:

```
<web-app>
             <display-name>SCWCD</display-name>
             <description>Ejemplos del libro.</description>
             <servlet>
                 <servlet-name>HelloWorldServlet</servlet-name>
                 <servlet-class>cap02.HelloWorldServlet</servlet-class>
```

```
        </servlet>
         …
<servlet-mapping>
             <servlet-name>HelloWorldServlet</servlet-name>

         <url-pattern>/HelloWorldServlet</url-pattern>
        </servlet-mapping>

        …

</web-app>
```

Toda la información del descriptor de despliegue debe de ir encapsulada entre las marcas <Web-app> y </Web-app>. Las etiquetas utilizadas son cerradas, después de abrirlas se declara su cierre precediendo su nombre por /. Ej.: <servlet> para abrir la etiqueta y </servlet> para cerrarla. Entre las marcas para la configuración de la aplicación más relevantes destacaremos:

- **<display-name> :** Nombre que describe al descriptor de despliegue. Este es visible por las herramientas de desarrollo. (opcional)

- **<description>:** Breve descripción (opcional)

- **<servlet>:** Contiene la declaración del servlet. Este define el nombre del Servlet y el fichero compilado donde se encuentra. (requerido).
  - **<servlet-name>:** Define el nombre del servlet que se empleará en el fichero web.xml cada vez que haya que hacer referencia al servlet en cuestión. No tiene por que coincidir con el nombre dado al servlet en el fichero java. (requerido)
  - **<servlet-class>:** Define el nombre de la clase del servlet. Este nombre debe incluir el paquete. (requerido)

- **</servlet-mapping> :** Describe el mapeo de direcciones entre el servlet y una url específica, es decir, determina la url que el cliente ha de introducir en el navegador para que el servidor ejecute el servlet.
  - **<servlet-name>:** Determina el servlet al que vamos a asignar una dirección URL. Este nombre ha de coincidir con el definido por la subetiqueta <servlet-name> de la etiqueta <servlet>.
  - **<url-pattern>:** Describe el patrón de la URL que se le asignará al servlet. Aquí se introduce la porción de la URL siguiente a: **http:**//nombreServidor:puerto/AplicacionWeb/... (parte de la url a introducir).

  Ejemplos de url's válidas:

  - /HelloWorldServlet
  - /curso/*
    Si la dirección url de nuestra aplicación Web fuese http://www.cleformacion.com, el anterior patrón lanzaría el servlet para toda dirección de la forma:
    http://www.cleformacion.com/curso/ …
  - /curso/manual/*

- **<context-param> :** Contiene la declaración de parámetros de inicialización de configuración de la aplicación Web. Este elemento ha de definirse antes de cualquier declaración de servlet dentro del fichero web.xml . (opcional)
  - **<param-name>:** Define el nombre del parámetro a tomar valor.
  - **<param-value>:** Declara el valor asignado al parámetro definido anteriormente.
  -

Se puede consultar esta información dinámicamente desde cualquier servlet que pertenezca al contexto de la aplicación. A través del objeto ServletContext, toda la información almacenada en él es global a toda la aplicación Web. Ejemplo:

CONTENIDO DEL FICHERO WEB.XML:

```
…
<context-param>
    <param-name> userId </param-name>
    <param-value> cle </param-value>
</context-param>
…
```

CÓDIGO PERTENECIENTE AL SERVLET:

```
…   ServletContext configuracion;
configuracion = getServletContext();…
//Para recuperar esta información
String usuario= configuracion.getInitParameter("userId");
```

LECTURA DE ARGUMENTOS PASADOS A TRAVÉS DE UN FORMULARIO

El html (HyperText Markup Lenguaje) es una implementación específica SGML (Standard Generalizad Markup Lenguaje), estándar internacional para la definición de texto electrónico independiente de dispositivos, sistemas y aplicaciones. Este supone el corazón del motor que mueve la información en Internet.

Por el momento no existe un estándar único del lenguaje de marcas HTML (actualmente por su versión 4) debido a la guerra desatada entre los distintos navegadores existentes en la actualidad que se empeñan en incluir etiquetas (elemento en que se fundamenta este leguaje descriptivo) que no son compatibles entre los navegadores de distintas compañías como Internet Explorer o Netscape.

Las directivas o etiquetas html pueden ser de dos tipos: abiertas o cerradas. Las cerradas son aquellas que tienen una palabra clave para delimitar el comienzo y el final de la directiva. Entre las etiquetas inicial y final pueden definirse nuevas directivas. Las abiertas constan de una única palabra clave. A su vez, algunas directivas pueden contener parámetros que se especifican a continuación de la etiqueta. Para diferenciar las directivas del resto del texto estas se encierran entre los caracteres '<' y '>'.

Ejemplo de directiva cerrada:

```
<CENTER> Curso de Jsp y Servlets </CENTER>
```

Ejemplo de directiva abierta:

```
<HR>
```

Ejemplo de directiva con parámetros:

```
<BODY bgcolor="#FFFFFF"> </BODY>
```

A continuación se verá la estructura básica de cualquier documento HTML:

```
<HTML>
     <HEAD>
          <TITLE>
          </TITLE>
     </HEAD>
     <BODY>
     </BODY>
</HTML>
```

```
Indica el inicio del documento.
Inicio de la cabecera.
Inicio del título del documento.
Final del título del documento.
Final de la cabecera del documento.
Inicio del cuerpo del documento.
Final del cuerpo del documento.
Final del documento.
```

FORMULARIOS HTML

Los formularios son las herramientas de las que dispone la tecnología HTML para mejorar la interactividad con el usuario, pudiendo a través de ellos, recopilar información del cliente para posteriormente procesarla y poder responder adecuadamente a sus peticiones o acciones. En un formulario podemos solicitar diferentes datos (campos) cada uno de los cuales quedará asociado a una variable.

La declaración de un formulario tiene lugar entre las directivas cerradas de HTML <form> y </form>. En el interior de la declaración del formulario tiene lugar la definición de las variables de entrada. Para la introducción de las variables se utiliza la directiva <INPUT>. Esta directiva tiene el parámetro type que indica el tipo de variable a introducir y name que indica el nombre que se le dará al campo.

Cada tipo de variable tiene sus propios parámetros, pudiendo algunos de ellas almacenar varios valores, como por ejemplo los checkbox o los campos de tipo radio.

Existen 3 tipos de datos: variables de entrada, variables de selección y áreas de texto.

Ejemplos de algunas directivas para leer datos de entrada:

- type = submit → Representa un botón. Al pulsar este botón la información de todos los campos se envía al programa indicado en <FORM>. Tiene el parámetro value = "texto" que indica el texto que aparecerá en el botón.
- type= text name = campo → Indica que el campo a introducir será un texto.
- type = password name = campo → Indica que el campo será una palabra de paso. Mostrará asteriscos (*) en lugar de las letras escritas.

La directiva form tiene los parámetros action y method:

- Action=Programa →Indica la URL del programa que procesará la información enviada por el cliente una vez que este pulse el botón de enviar (Submit). En nuestro caso, el servlet que extraerá y procesará dicha información.
- Method = POST/GET→ Indica el método a emplear para el envío de la información proporcionada por el usuario. El método GET concatena los valores de las variables de entrada a la URL introducida en action (separando la raíz por el carácter '?' y los pares de nombres de variables-valor por '&'). Por ejemplo este es el método empleado cuando realizamos una búsqueda en google:

```
http://www.google.com/search?hl=es&ie=UTF-8&oe=UTF-
8&q=manual+de+programacion&lr=
```

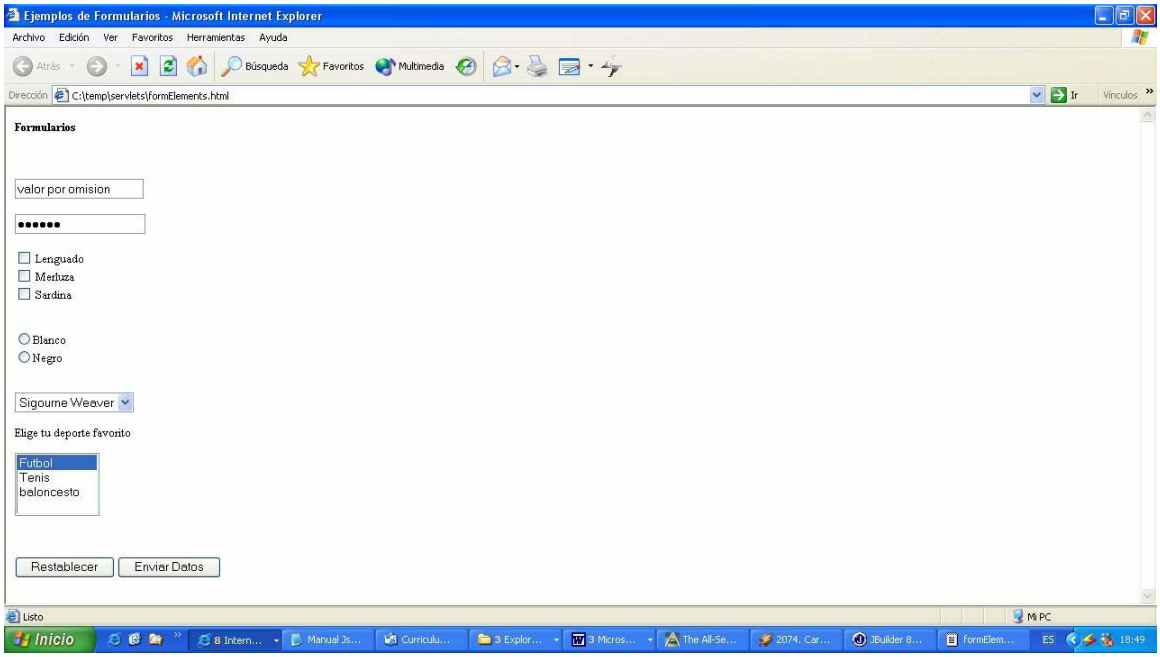
El método GET impone una limitación máxima de 4KB para el tamaño de la información a enviar.

El método POST, ha diferencia del anterior, carece de esta restricción y los datos enviados no modifica la URL para codificarlos, sino que se encapsula en el cuerpo de la petición, no siendo así visible en la URL la información enviada al servidor. Este último método es ideal cuando se desea transmitir grandes volúmenes de datos o bien no queremos que la información enviada sea visible, por ejemplo, al enviar claves o datos confidenciales por motivos de seguridad.

LECTURA DE PARÁMETROS DESDE UN SERVLET

Una vez que el usuario ha rellenado los datos del formulario y los ha enviado, es necesario que el servlet extraiga dicha infamación y la procese convenientemente. Para el proceso de extracción de los datos enviados por el usuario se emplearan los métodos definidos por la clase ServletRequest. Más concretamente, mediante el método getParameterNames() obtendríamos un objeto de tipo Enumeration que almacenaría el nombre de todas las variables de entrada encapsuladas en la petición de envío del formulario. A partir de esta lista de nombres, podríamos obtener el valor de estas variables mediante el método *getParameter(String nombreParámetro)* o bien por medio del método *getParamenterValues(String nombreParámetro)* en el caso de que la variable de entrada almacenase más de un valor, como es el caso del tipo de entrada checkbox.

Veamos algunos ejemplos para la lectura de parámetros en un formulario. Dado el siguiente formulario HTML, veremos a continuación el servlet necesario para extraer la información introducida:



Código HTML correspondiente al anterior formulario (emplea el método POST):

```
<HTML>
<HEAD>
<TITLE>Ejemplos de Formularios</TITLE>
</HEAD>
<BODY>
<B>Formularios</B>
<FORM ACTION="LecturaParametros" METHOD="POST">

<BR><BR>
<INPUT TYPE='text' NAME='nombre' VALUE='valor por omision' SIZE='20'>
<BR><BR>

<INPUT TYPE='password' NAME='password' VALUE='secret' MAXSIZE='16'>
<BR><BR>

<INPUT TYPE='checkbox' NAME='pescado' VALUE='lenguado'>  Lenguado <BR>
<INPUT TYPE='checkbox' NAME='pescado' VALUE='merluza'> Merluza <BR>
<INPUT TYPE='checkbox' NAME='pescado' VALUE='sardina'> Sardina <BR>
<BR><BR>

<INPUT TYPE='radio' NAME='color' VALUE='B' SELECTED>Blanco<BR>
<INPUT TYPE='radio' NAME='color' VALUE='N'>Negro<BR>
<BR><BR>
```

```
<SELECT NAME='Artista de Cine'>
  <OPTION VALUE='Al Pacino'> Al Pacino
  <OPTION VALUE='Sigourne Weaver' SELECTED> Sigourne Weaver
  <OPTION VALUE='Paz Vega'> Paz Vega
</SELECT>
<BR><BR>
```

```
Elige tu deporte favorito
<BR><BR>
<SELECT NAME='deporte' MULTIPLE>
  <OPTION VALUE='futbol' SELECTED> Futbol
  <OPTION VALUE='tenis'> Tenis
  <OPTION VALUE='baloncesto'> baloncesto
</SELECT>
<BR><BR>
<BR><BR>

<INPUT TYPE='reset'>
<INPUT TYPE='submit' VALUE='Enviar Datos'> <BR>

</FORM>
</BODY>
</HTML>
```

Código correspondiente al servlet que procesará los datos obtenidos por el formulario:

```
package cap04;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LecturaParametros extends HttpServlet {
/*Con independencia del método empleado para codificar los datos, ya que para
los servlets estos son tratados de igual forma, ambas peticiones se redirigen
al método processRequest, para atender la petición*/
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
           throws IOException {
           processRequest(request, response);
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
           throws IOException {
           processRequest(request, response);
    }
```

```
private void processRequest(HttpServletRequest request, HttpServletResponse
response)
            throws IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<HTML>");
            out.println("<HEAD>");
            out.println("<TITLE>Lectura Parámetros 1</TITLE>");
            out.println("</HEAD>");
            out.println("<BODY BGCOLOR='white'>");
            out.println("<B>" + obtainParameter("nombre", request) + "</B>");
            out.println("<B>" + obtainParameter("password", request) + "</B>");
            out.println("<B>" + obtainParameter("pescado", request) + "</B>");
            out.println("<B>" + obtainParameter("color", request) + "</B>");
            out.println("<B>" + obtainParameter("Artista de Cine", request) +
"</B>");
            out.println("<B>" + obtainParameter("deporte", request) + "</B>");
            out.println("</BODY>");
out.println("</HTML>");
            out.close();
    }

/* El siguiente método extrae los datos del formulario, mediante una llamada
al método getParameter(nombreParámetro) del objeto request, con el nombre de
parámetro que le pasa processRequest. */
private String obtainParameter(String name, HttpServletRequest request) {
            String result = "";
            if (name != null && name.length() > 0) {
                  result = name + ": ";
                  String value = request.getParameter(name);
                  if (name == null || name.length() == 0) {
                        result += "nulo";                }
                  else {
                        result += value;
                  }

                  result += "<BR>";
            }
            return result;
    }
}
```

CICLO DE VIDA DE UN SERVLET

Los servlets corren en un servidor web como parte del mismo proceso que ejecuta el propio servidor. El servidor web es el responsable de inicializar, llamar y destruir cualquier instancia de un servlet.

El servidor web se comunica con los servlets a través de una interface simple, como es *javax.servlet.Servlet*. Esta interface consta de tres métodos principales:

- init()
- service()
- destroy()

Y dos métodos destacados en segundo plano:

- getServletContext()
- getServletInfo()

Todos los servlets tienes el mismo ciclo de vida:

- Un servidor carga e inicializa el servlet.
- Un servidor maneja cero o más peticiones de cliente.
- El servidor elimina el servlet.

EL MÉTODO INIT()

Cuando un servlet se carga por primera vez, lo primero que hace el servidor web es ejecutar su método init(). Esto posibilita al servlet llevar a cabo cualquier tarea de configuración o inicialización, como por ejemplo el establecer conexiones con bases de datos u otros servidores así como abrir ficheros para el procesamiento de configuraciones.

La llamada al método init se realiza una única vez, a no ser que el servlet sea recargado por el servidor, antes de procesar ninguna otra llamada, por ejemplo al método service().

Como veremos en el capítulo 7, es posible acceder a los parámetros iniciales del servlet, definidos en el descriptor web.xml mediante la cláusula <context-param>, invocando al método getInitParameter(*nombreParámetro*) del objeto ServletContext.

EL MÉTODO SERVICE()

El método service() constituye el corazón de un servlet. Es el responsable de atender las peticiones de servicio de los clientes y proporcionar una respuesta a las mismas a través de los objetos pasados como argumento en su llamada:

- Un objeto del tipo ServletRequest con información del cliente. Esta información está principalmente estructurada como un conjunto de tuplas nombreParámetro-valor y un InputStream. La mayoría de los métodos de esta clase están orientado a recopilar dicha información del cliente.

Podemos obtener el InputStream a partir del método getInputStream() que devolverá un objeto de tipo ServletInputStream que se puede utilizar para recopilar información adicional del cliente. Hay que tener en consideración para el diseño e implementación del servlet, que el método service() atiende simultáneamente múltiples peticiones de usuarios a la vez, lo cual podría degenerar en problemas de concurrencia al acceder a recursos comunes, como son bases de datos, ficheros…

- Una instancia del objeto ServletResponse que representa la información de respuesta del servidor al cliente. Previamente se debe de fijar el contenido o formato de la respuesta mediante *setContentType()*. A su vez, mediante los métodos *getOutputStream()* o *getWriter()*, se abren las vías de comunicación con el cliente para que este puede recibir información del servidor.

EL MÉTODO DESTROY()

El método destroy() es llamado por el servidor para destruir un servlet, por ejemplo a petición del administrador o bien con fines de limpieza, como la liberación de recursos asignados a éste (recursos externos, ficheros, bases de datos…). Así, si no se necesitan de tareas de liberación de recursos la implementación de este método podría ser vacía.

El servidor espera a destruir el servlet a que todas las peticiones realizadas al método service() terminen o bien pase una determinada cantidad de tiempo, de forma que sea posible la eliminación del servlet a pesar de que se presenten peticiones que precisen de

un largo periodo de tiempo para su procesamiento y finalización. Al igual que el método init(), éste es sólo llamado una vez.

GESTIÓN DEL ESTADO DEL CLIENTE: HTTPSESSION

Como ya hemos visto anteriormente, una de las principales ventajas de los servlets es la posibilidad de llevar a cabo un seguimiento de la sesión de un usuario. Al carecer de estado el protocolo http, es imposible almacenar la información de las acciones o peticiones del cliente (con un mismo navegador) a lo largo de sus movimientos por un determinado sitio web durante un determinado período de tiempo.

Pasa solventar este problema y posibilitar al servidor web mantener el estado, la tecnología Java Servlet propone el uso de sesiones. Éstas no son más que una API que gira en torno al objeto HttpSession, que nos permite almacenar información como si de una tabla se tratase.

Las sesiones las comparten todos los servlets que conforman una aplicación web a los que accede el cliente, lo cual resulta muy útil en aplicaciones compuestas por más de un servlet.

OBJETO HTTPSESSION

Los pasos a seguir para utilizar el seguimiento de sesiones empleando un objeto HttpSession propio de los servlets es:

- Obtener una sesión para un cliente. Para ello basta con obtener un objeto HttpSession.
- Administrar la sesión (consultar o modificar la información almacenada en HttpSession).
- Invalidar la sesión si fuese necesario.

OBTENER UNA SESIÓN

Para obtener una instancia de la clase HttpSession invocamos al método getSession(boolean crear) de HttpServletRequest, antes de proporcionar ninguna respuesta a una petición. Si pasamos *true* como parámetro, el servlet la creará en el caso de que no exista previamente.

```
HttpSession sesion = HttpServletRequest.getSession(true);
```

Cuando se crea una sesión, el servidor le asigna un número de identificación único denominado *session id* con el que podrá identificar unívocamente las sesiones de los distintos clientes que acceden concurrentemente al servidor. La id se transmite al cliente utilizando para ello cookies o la reescritura de la URL.

Posteriormente cada vez que el cliente accede a un recurso del servidor (servlets, jsp's), el cliente provee al servlet o jsp el id que previamente le había sido asignado.

ADMINISTRAR LA SESIÓN

Una sesión es como una pizarra de la que se puede leer y escribir. Cualquier objeto Java se puede añadido a HttpSession. A Todo objeto asociado a un objeto HttpSession se le

debe asignar un nombre único. Hay que tener en cuenta los posibles conflictos entre nombre de objetos distintos al compartir todos los servlets de una misma aplicación web las sesiones de los clientes. Los procesos de lectura y escritura se realizan por medio de los métodos de HttpSession getAttribute() y setAttribute() respectivamente.

Ejemplo:

- Escritura de datos en HttpSession

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
…
HttpSession sesion= request.getSession(true);
Integer anyoSesion=new Integer(2001);
sesion.setAttribute("miItemDeSesion", anyoSesion);
…
}
```

- Lectura de datos de HttpSession

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
…
HttpSession sesion= request.getSession(true);
//getAttribute devuelve null si no se asigno previamente.
Integer item_sesion= (Integer) sesion.getAttibute("miItemDeSesion");
int contador= item_sesion.intValue();}
```

Otros métodos destacables de la clase HttpSession son:

- **getId:** Este método devuelve el identificador único generado para cada sesión.
- **isNew:** Esto devuelve *true* si el cliente (navegador) nunca ha visto la sesión, normalmente porque acaba de ser creada en vez de empezar una referencia a una petición de cliente entrante. Devuelve *false* para sesiones preexistentes.
- **getCreationTime**: Devuelve la hora, en milisegundos desde 1970, en la que se creo la sesión.
- **getLastAccessedTime**: Esto devuelve la hora, en milisegundos desde 1970, en que la sesión fue enviada por última vez al cliente.
- **getMaxInactiveInterval**: Devuelve la cantidad de tiempo, en segundos, que la sesión debería seguir sin accesos antes de ser invalidada automáticamente. Un valor negativo indica que la sesión nunca se debe desactivar.

INVALIDAR LA SESIÓN

Una sesión de usuario puede ser invalidada manual o automáticamente; por ejemplo el servidor invalida una sesión tras un cierto tiempo de inactividad por parte del cliente. Al eliminar una sesión se borra el objeto HttpSession asociado y la información en él contenida. Para invalidar una sesión manualmente basta con invocar al método *invalidate()*. Por ejemplo podríamos querer invalidar una sesión en una página de comercio electrónico, tras hacer efectivas las compras por parte del cliente.

Compartición de recursos: ServletContext

En una aplicación Web todos los Servlets que la conforman comparten un contexto único. La información almacenada en el contexto, es accesible por todas las páginas JSP y Servlets del proyecto de la aplicación. Así es posible obtener información común a todos ellos. Es a su vez, un método de comunicación con el contenedor de Servlets alojado en el servidor Web, para obtener información relevante, tal como el entorno sobre el que se ejecuta.

INFORMACIÓN DE INICIALIZACIÓN DEL SERVLET

La información de inicialización la obtiene el servlet a través de llamada a getServletContext(). Esta llamada devuelve un objeto de tipo ServletContext y, a partir de él, obtenemos la información de inicialización. Esto es útil, por ejemplo para guardar datos como el identificador del controlador que gestiona el acceso a bases de datos, la dirección de correo del servidor, hojas de estilo URL para la aplicación de manera que todas las páginas de un determinado proyecto tengan la misma presentación…

Estos datos son añadidos al fichero web.xml empleando las etiquetas <param-name> y <param-value>, que son atributos de la cláusula <context-param>. El primero de ellos define el nombre del parámetro mientras que el segundo almacena el valor de este.

Por ejemplo, queremos guardar la dirección de correo electrónico de nuestro servidor para que se pueda acceder a ella desde cualquier página de la aplicación. Para ello, añadimos las siguientes líneas al fichero web.xml:

```
<web-app>
…
<context-param>
    <param-name> direccionEmail </param-name>
    <param-value> cleFormacion@cle.com </param-value>
</context-param>
…
</web-app>
```

Para recuperar dicha información lo haremos en el método init() del servlet; llamamos al método getInitParameter(String *nombreParámetro*) del objeto ServletContext que obtenemos mediante la llamada a getServletContext()

```
public void init() {
ServletContext contexto;
//Obtenemos el contexto del servlet a partir de ServletConfig
   contexto= getServletContext();
// Ahora en la variable direccionCorreo almacenaremos el valor del parámetro
//direccionEmail que había guardado en el fichero web.xml
String direccionCorreo= contexto.getInitParameter("direccionEmail");
    …
    }
```

ACCESO A ATRIBUTOS COMPARTIDOS.

El acceso a esta información tiene lugar en tiempo de ejecución, luego es conveniente que las operaciones de lectura y escritura de atributos se hagan en el cuerpo de los métodos doGet() o doPost() del servlet.

A partir del objeto ServletContext obtenido durante la inicialización, disponemos de un conjunto de métodos para obtener información relevante del entorno, tal como, el nombre y la versión del contenedor de servlets, la dirección relativa del directorio raíz de servidor web sobre el que corre…

El contexto de una aplicación web también sirve para almacenar y modificar atributos en el formato de pares de nombre-valor, es decir, datos comunes a los que todos los servlets que componen nuestro proyecto debieran tener acceso. Esta información puede ser consulta y modificada dinámicamente desde cualquier servlet que pertenezca al contexto de la aplicación a través de los métodos de ServletContext *setAttribute()* y *getAttribute()* respectivamente.

Para comunicarse dos Servlets el nombre de un determinado usuario, el servlet que obtiene el nombre lo difundiría al contexto por medio de la siguiente línea:

```
contexto.setAttribute("nombreUsuario", "Antonio");
```

Y el servlet que quiere leer el nombre del usuario implementaría dentro de su método service() (al igual que el anterior):

```
String str = (String)contexto.getAttribute("nombreUsuario");
```

ASPECTOS DE CONCURRENCIA EN LOS SERVLETS.

Una de las principales ventajas de los servlets frente a otras tecnologías, como por ejemplo las páginas CGI, es la potencia. Ésta radica en que en que el contenedor web no lanza un proceso para atender cada petición de servicio de un cliente,[1] sino que crea un thread para atenderla. Existe una única copia del servlet en el servidor pero múltiples hilos que actúan contra este servlet. De este modo, se reducen los requerimientos hardware necesarios por parte del servidor web.

No obstante, esta ventaja podría tornarse en desventaja, si en la programación del servlet no se tiene en cuenta este aspecto, dándose la posibilidad de problemas de concurrencia al intentar acceder todos los hilos a datos comunes dentro del servlet.

Estos problemas de concurrencia nunca tendrán lugar dentro del cuerpo del método init(), ya que a éste se le llama una única vez en el momento de la creación del servlet o cuando es necesario descargarlo de memoria para volverlo a lanzar de nuevo. Además los métodos del ciclo de vida de un servlet, service() y destroy(), no se pueden invocar hasta que la ejecución del método init() haya finalizado.

Sin embargo un servidor web puede llamar al método service() múltiples veces para atender peticiones simultáneamente de los clientes. Si el método service() utiliza recursos externos como bases de datos, ficheros, instancias de variables no locales al método… podrían darse problemas de concurrencia en el acceso a los mismos por parte de varios hilos a la vez.

Por ejemplo, supóngase el uso de un contador, declarado como instancia de clase, que controle el número de peticiones simultaneas que atiende el método service() simultáneamente:

[1] El crear un proceso en el sistema operativo es una operación muy costosa que consume muchos recursos.

```
import javax.servlet;
    import javax.servlet.http;
    public class MiServlet extends GenericServlet {
           …
           private int contador;
           …
        public void service(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
// Incrementamos en uno el contador al comenzar a procesar una nueva
//petición.
             int contadorAux=contador + 1; //línea 1
             contador=contadorAux;           //línea 2
             …
             // Antes de salir del método service y salir del método service()
decrementamos
            //el contador de las peticiones que se atienden concurrentemente.
            contador=contador-1;
            }  // Fin del método service()
    …
    }
```

Si dos servlets ejecutan a la vez el método service(), existe la posibilidad que mientras uno no ha terminado de ejecutar la línea 1, el otro esté ejecutando la línea 2, sin embargo, el valor de contadorAux no sería consistente entre ambos, tendría diferente valor, lo que falsearía el resultado.

Una de las soluciones posibles sería encapsular la parte de código susceptible de ser problemática entre cláusulas *synchronized*, de manera que un único hilo pueda acceder a la vez a esa parte del código, mientras el otro permanece a la espera de que el primero finalice.

```
…
    synchronized(this) {
             Int contadorAux=contador + 1; //línea 1
             contador=contadorAux;           //línea 2
    }
    …
    synchronized(this) {
           contador=contador-1;
    }
```

Los peligros potenciales de concurrencia de múltiples hilos contra el método service() se reducen a:

- La utilización de variables de clase como hemos visto.
- El uso de variables estáticas, ya que de éstas por su naturaleza sólo existe una única instancia de ella en la clase, común a todas.
- El acceso a la sesión. Los datos almacenados en la sesión son compartidos por todos los servlets e hilos. Se podría dar el caso de un usuario actuando simultáneamente sobre la misma aplicación web desde dos ventanas distintas, de forma que las acciones tomadas en una ventana no tengan constancia en la otra.

COLABORACIÓN ENTRE SERVLETS: HTTPSERVLETREQUEST, HTTPSESSION Y SESSIONCONTEXT. EL REQUESTDISPATCHER.

La potencia de los servlets, no sólo radica en la gestión de múltiples threads simultáneamente con bajos recursos, sino en las posibilidades de comunicación entre éstos. La tecnología servlets nos brinda una serie de objetos para tal fin, como son HttpServletRequest, HttpSession, SessionContext y el RequestDispatcher. Los tres primeros ya fueron vistos en capítulos anteriores, sin embargo, estudiaremos ahora su alcance.

El objeto HttpServletRequest tiene su existencia limitada al tiempo necesario para procesar una petición determinada. Una vez que ésta finaliza, el objeto se elimina de memoria. El objeto HttpServletRequest se emplea para obtener información acerca de la petición de servicio de un cliente en un determinado servlet. Existe la posibilidad de redirigir dicha petición a otro servlet para que sea éste último el que la atienda en su lugar. Este hecho es muy útil a la hora de estructurar adecuadamente una aplicación web, ya que así, se podrían crear una jerarquía de servlets que se encarguen de las distintas etapas que componen el procesamiento de una solicitud. Por ejemplo, se podría asignar el acceso a bases de datos a un servlet mientras otro distinto se encargase de construir y suministrar la información obtenida, en páginas dinámicas enviadas al navegador del cliente. Otra posibilidad la podemos encontrar en un proyecto web de comercio electrónico, donde cada uno de los servlets se especializase en la gestión de la venta de los distintos tipos de productos: Electrónica, Ropa, Ocio… Para redirigir las peticiones empleamos el objeto RequestDispatcher que se verá más adelante en este capítulo.

Mediante el objeto HttpSession es posible almacenar información relevante a las acciones que ha efectuado un cliente a lo largo de su navegación por un sitio web. Se asocia un objeto HttpSession a cada cliente conectado en un determinado instante. El alcance de este objeto es global a la aplicación y visible por tanto desde cualquier servlet.

SessionContext nos permite almacenar información dentro del contexto único que posee la aplicación Web. Estos datos son accesibles desde cualquier servlet perteneciente a la aplicación; por lo tanto, su alcance es global. También existe la posibilidad de acceder al contexto de aplicaciones web distintas, siempre y cuando estén asociadas a un mismo sistema virtual principal.

RequestDispatcher define una interface que posibilita recibir una petición y redirigirla para que sea tratada por otro componente; como por ejemplo, un servlet, una página HTML o JSP, un script CGI… el cual pasa a tomar el control definitivamente, recibiendo los datos de sesión, contexto y petición. Para ello, define el método **forward**(ServletRequest request, ServletResponse response). Otra alternativa que ofrece esta interface es permitir a un servlet incluir en su respuesta la devuelta por otro servlet o página JSP. Con este fin, cede el control al servlet o página JSP hasta que finalice su ejecución, momento en el cual vuelve a tomar control. Para ello, define el método **include**(ServletRequest request, ServletResponse response).

Para obtener un objeto que implemente la interface RequestDispatcher se ha de llamar al método getRequestDispatcher(String URI) del objeto ServletContext. Este método toma como argumento una cadena que representa la ruta del recurso a emplear. Esta ruta ha de comenzar por '/' y se interpreta como relativa a la raíz de la aplicación.

Veamos ahora con más detalle, cómo utilizar estos dos métodos.

- Dentro de un servlet o de una página JSP, el método ***include()*** del interface RequestDispatcher permite incluir la respuesta generada por otro servlet, página JSP, fichero... Es típico el uso de cabeceras y pies HTML comunes en todas las páginas que componen una aplicación. Así por ejemplo, el siguiente servlet utiliza en su respuesta el contenido de los ficheros encabezado.bmp y pie.bmp:

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

public class MiServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, java.io.IOException {
        //Incluimos la cabecera
        RequestDispatcher rd =
getServletContext().getRequestDispatcher("/imagenes/encabezado.bmp");
        rd.include(request, response);

        //Mostramos la información del libro
        PrintWriter out = response.getWriter();
        out.println("<h1>BOOKS-WORLD … TIENDA DE VENTA DE LIBROS ONLINE </h1>");
        …
        //Añadimos un pie al final
        RequestDispatcher rd2 =
getServletContext().getRequestDispatcher("/imagenes/pie.bmp");
        rd2.include(request, response);
    }
}
```

- El método ***forward()*** posibilita el redireccionar una petición recibida por un servlet a otro de los recursos soportados por el motor de servlets como páginas JSP, otro servlet… Una vez que se ha pasado el control a ese otro recurso, éste no vuelve al servlet original. Obsérvese al diferencia con el método ***include().*** Con el método incluye() una vez que se ha ejecutado el recurso llamado, el control vuelve al servlet original. Mediante el método forward() podemos estructurar mejor el código de la aplicación, lo que agiliza la depuración de errores, así como reducir el tiempo de desarrollo y los costes. La sintaxis de include() es la misma que la de forward().

```
public void doPut(HttpServletRequest req,
                  HttpServletResponse res)
        throws ServletException, IOException{
        ...

    // Obtenemos el requestDispatcher para mostrar una página de
    //agradecimiento a nuestro cliente
    RequestDispatcher Agradecido =
      getServletContext().getRequestDispatcher(
          "/servlet/Gracias");

    // Sino se encuentra el servlet en el servidor contruimos la
    //respuesta desde éste.
    if (Agradecido == null) {
    res.setContentType("text/html");
        PrintWriter respuesta = res.getWriter();
        respuesta.println("<html>");
    respuesta.println("lt;title>Gracias por su compra!</title>");
        respuesta.println("Su pedido sera enviado en menos de 48
                  horas!");
    respuesta.println("</html>");
    respuesta.close();
        return;
    }
```