J2EE, Servlet y JSP

Contenido

SERVLETS

1.- Introducción a los Servlets.

- Los Servlets son aplicaciones que implementan los servidores orientados a petición-respuesta, como los servidores web compatibles con Java. Por ejemplo, un servlet podría ser responsable de recoger los datos de un formulario que se ha mandado desde un cliente remoto en HTML y aplicarle la lógica de negocios[1] utilizada para actualizar la base de datos de la aplicación.
- Pueden ser incluidos en muchos servidores diferentes porque el API Servlet, el que se utiliza para escribir Servlets, se adapta al entorno o protocolo del servidor. Los servlets se están utilizando ampliamente dentro de servidores HTTP[2]; muchos servidores Web[3] soportan el API Servlet.
- En nuestro caso podremos utilizar Tomcat como servidor web en Java, que es un módulo independiente del servidor http Apache. Otra alternativa es usar un servidor de aplicaciones como Glassfish.
- Los servlets no tienen interface gráfico de usuario, pero pueden usar los componentes de las páginas HTML.
- Proporcionan una forma de generar páginas dinámicas que son fáciles de escribir y rápidos en ejecutarse.
- Los Servlets facilitan una plataforma genérica para desarrollar software en el lado del servidor, con una extensión estándar de Java, la API Servlet.
- Los servlets se utilizan para manejar peticiones de cliente HTTP. Por ejemplo, un servlet diseñado para procesar datos de un formulario de página HTML con protocolo HTTP. En el formulario pueden viajar datos personales o requeridos por la aplicación. El servlet podría formar parte de un sitio web en el que los datos se procesaran usando tablas en bases de datos.
- Un servlet puede manejar múltiples peticiones concurrentes, y puede sincronizarlas.
- Reenviar peticiones. Los Servlets pueden reenviar peticiones a otros servidores y servlets. Con esto pueden ser utilizados para repartir la carga entre servidores que contengan la misma información, y para particionar un único servicio lógico en varios servidores (por ejemplo, uso de bases de datos distribuidas).

En : http://tomcat.apache.org/tomcat-5.5-doc/servletapi/index.html
disponemos de toda la documentación referente a los servlets.

[1] El software principal de la aplicación distribuida
[2] Además de servidor Web, satisface transferencia de ficheros, correo, etc...
[3] Es una aplicación que se ejecuta sobre un servidor que escucha las peticiones HTTP (páginas) que le llegan y las satisface.

2.- Ejemplo de Servlet.

La siguiente clase define completamente un servlet que devuelve la hora del servidor.

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

public class ServletHola extends HttpServlet {

  private static final String CONTENT_TYPE = "text/html; charset=windows-1252";
  public void init(ServletConfig config) throws ServletException  {
    super.init(config);}

  public void doGet(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException  {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head><title>Mi primer Servlet</title></head>");
    out.println("<body>");
    out.println("<p>Hola soy un servlet.</p>");
    out.println("<br><br> La fecha del servidor es: "+new java.util.Date());
    out.println("</body></html>");
    out.close();
  }
}
```

Comentarios al programa:

import javax.servlet.*

Importa el paquete servlet, que proporciona clases e interfaces para escribir servlets. La arquitectura de este paquete se describe a continuación.

import javax.servlet.http.*

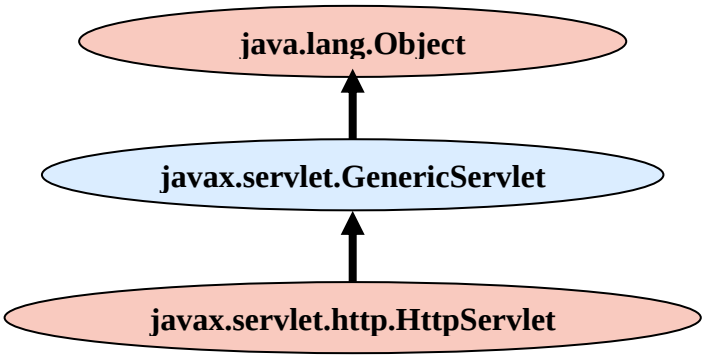
Indica que el servlet va a manejar peticiones http.

import java.io.PrintWriter

Para comunicarnos con el cliente, utilizaremos el método *out.println( )*, que forma parte de la clase PrintWriter.

*public class ServletHola **extends HttpServlet***

La funcionalidad principal del API Servlet la proporciona la interface **Servlet**. Todos los servlets implementan este interface, bien directamente o, más comúnmente, extendiendo una clase que ya lo incorpora, lo implementa, como **HttpServlet**
Esquema:



El interface **Servlet** declara, pero no implementa, métodos que manejan el Servlet y su comunicación con los clientes. Deberemos codificar algunos de esos métodos para desarrollar un servlet.

*public void **doGet(HttpServletRequest** request, **HttpServletResponse** response)*
*throws **ServletException**, IOException*

Este es uno de los métodos de obligada escritura para codificar servlets.
Un Servlet HTTP maneja peticiones del cliente a través de su método **service**. Este método soporta peticiones estándar de cliente HTTP despachando cada petición a un método designado para manejar esa petición. Por ejemplo, el método **service** llama al método **doGet** mostrado anteriormente.
Este método utiliza un objeto que representa las peticiones del cliente (**HttpServletRequest**) y otro que lo hace con las respuestas del servlet (**HttpServletResponse**). Estos objetos se proporcionan al método **service** y a los métodos que **service** llama para manejar peticiones HTTP.

Es decir, la información del cliente que recibimos a través de la página html, está encapsulada en el objeto **HttpServletRequest**, de aquí obtenemos los parámetros pasados en un hipotético formulario para procesarlos posteriormente.
Cuando contestamos al cliente, lo hacemos encapsulando la información en el objeto **HttpServletResponse.**

Es obligatorio avisar de un posible error de entrada/salida o alguno interno de servlet con la clausula throws.

*out.println("

 La fecha del servidor es: "+new java.util.Date());*

El código que hay dentro del método get(..) tiene dos esquemas diferentes, por una parte se trata de un formato típico de página html utilizando un objeto de tipo PrintWriter, que se trasladará hasta el cliente sin ningún tipo de variación. Por otra parte se observa que existe código puro en java; éste antes de enviarse al cliente se ejecutará en el servidor y se enviará su resultado, en este caso la fecha del sistema.

Aquí se desvela el carácter **dinámico** de los servlets frente al estático de las páginas html. Un servlet puede variar la respuesta al cliente dependiendo del momento en que se solicita. Las páginas html siempre mandarán la misma versión.

3.- Peticiones y Respuestas

Como ya se ha apuntado, los métodos de la clase **HttpServlet** que manejan peticiones de cliente toman dos argumentos.

- Un objeto **HttpServletRequest**, que encapsula los datos desde el cliente.
- Un objeto **HttpServletResponse**, que encapsula la respuesta hacia el cliente.

3.1.- Objetos HttpServletRequest

Un objeto **HttpServletRequest** proporciona acceso a los datos de cabecera HTTP, cualquier cookie encontrada en la petición, y el método HTTP con el que se ha realizado la misma. El objeto **HttpServletRequest** también permite obtener los argumentos que el cliente envía como parte de la petición.

Para acceder a los datos del cliente:

- El método **getParameter** devuelve el valor de un parámetro nombrado. Si nuestro parámetro pudiera tener más de un valor, deberíamos utilizar **getParameterValues** en su lugar. El método **getParameterValues** devuelve un array de valores del parámetro nombrado. (El método **getParameterNames** proporciona los nombres de los parámetros).

Ejemplo: Obtener todos los valores de los parámetros; cada parámetro tiene un único valor.

```
Enumeration parametros=request.getParameterNames();
while(parametros.hasMoreElements()){
 String param=(String)parametros.nextElement();
 String valor=request.getParameter(param);
 out.println(param+" = "+valor);
 }
```

- Para peticiones GET de HTTP, el método **getQueryString** devuelve en un **String** una línea de datos desde el cliente. Debemos analizar estos datos nosotros mismos para obtener los parámetros y los valores.
- Para peticiones POST, PUT, y DELETE de HTTP.
  - Si esperamos los datos en formato texto, el método **getReader( )** devuelve un **BufferedReader** utilizado para leer la línea de datos.
  - Si esperamos datos binarios, el método **getInputStream( )** devuelve un **ServletInputStream** utilizado para leer la línea de datos.

3.2.- Objetos HttpServletResponse

Un objeto **HttpServletResponse** proporciona dos formas de devolver datos al usuario.

- El método **getWriter( )** devuelve un **Writer.**
- El método **getOutputStream( )** devuelve un **ServletOutputStream**.

Se utiliza el método **getWriter()** para devolver datos en formato texto al usuario y el método **getOutputStream()** para devolver datos binarios.
Si cerramos el **Writer** o el **ServletOutputStream** después de haber enviado la respuesta, permitimos al servidor saber cuándo la respuesta se ha completado.

Cabecera de Datos HTTP
Debemos seleccionar la cabecera de datos HTTP antes de acceder a **Writer** o a **OutputStream**. La clase **HttpServletResponse** proporciona métodos para acceder a los datos de la cabecera. Por ejemplo, el método **setContentType** selecciona el tipo del contenido. (Normalmente esta es la única cabecera que se selecciona manualmente).

Problemas con los Hilos
Los Servlets HTTP normalmente pueden servir a múltiples clientes concurrentes. Si los métodos de nuestro Servlet no funcionan con clientes que acceden a recursos compartidos, deberemos sincronizar el acceso a estos recursos o crear un servlet que maneje sólo una petición de cliente a la vez.

Descripciones de Servlets
Además de manejar peticiones de cliente HTTP, los servlets también son llamados para suministrar descripción de ellos mismos. Por ejemplo sobrescribiendo el método **getServletInfo()**, que suministra una descripción del servlet.

4.- Manejar Peticiones GET y POST

Los métodos en los que delega el método **service** las peticiones HTTP, incluyen:

doGet, para manejar GET, GET condicional, y peticiones de HEAD
doPost, para manejar peticiones POST
doPut, para manejar peticiones PUT
doDelete, para manejar peticiones DELETE

Método	Descripción
doGet	Permite a un cliente leer información del servidor web mediante una cadena de consulta añadida a una dirección URL que indica al servidor qué información debe devolver. Suele contener parámetros. Tiene longitud limitada. Ejemplo: **http://www.mihost.com/mipath/miservlet.class?nombre1=valor1&nombre2=valor2** Los datos enviados así son susceptibles de ser interceptados. También se produce un GET cuando se escribe una dirección, se pulsa un enlace o se utiliza un marcador.
doPost	Permite que el cliente envíe datos de longitud ilimitada al servidor web en una sola vez. También se produce un POST por defecto, cuando en un formulario de una página html se pulsa el botón Enviar "submit" , aunque es configurable. Se utiliza para enviar información confidencial (p.e. tarjeta de crédito); no la envía en forma de parámetros, sino que el navegador codifica los datos en el propio mensaje. Es el método más empleado.
doPut	Permite a un cliente colocar un archivo en el servidor, de forma similar al envío de un archivo mediante FTP.
doDelete	Permite a un cliente eliminar un documento o página web del servidor.

Por defecto, estos métodos devuelven un error **BAD_REQUEST (400)**. Nuestro servlet debería sobrescribir el método o métodos diseñados para manejar las interacciones HTTP que soporta. Los métodos para manejar las peticiones HTTP más comunes son GET y POST.
Ejemplo de servlet con control asociado de métodos doGet y doPost para peticiones GET y POST respectivamente.

```
package paquetecontraseña;

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

public class ServletContraseña extends HttpServlet {

    protected void processRequest (HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            /* Business logic */
                String vUsuario = "";
                String vPassword = "";
                try {
                  vUsuario = request.getParameter("Tusuario");
                  vPassword = request.getParameter("Tpassword");
                }catch(Exception e){
                  e.printStackTrace();
                }
             /* Answer to the client */
                out.println("<html>");
                out.println("<head><title>Contraseña</title></head>");
                out.println("<body>");
                if(vUsuario.equals("java")&&vPassword.equals("servlet"))
                  out.println("<p>O.K.</p>");
                else{
                  out.println("<p>ERROR</p>"+
                  "<a href='index.jsp'>Volver </a>");
                 }
                out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }
```

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

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

    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>
}
```

El servlet extiende la clase **HttpServlet** y sobreescribe los métodos **doGet() y doPost()**, derivando su tratamiento a un método nuevo llamado **processRequest()** que no pertenece a la API de **HttpServlet** .

Dentro del método **processRequest**, el método **getParameter** obtiene los argumentos esperados por el servlet.
Para responder al cliente, el método **doGet()** utiliza un **Writer** - del objeto **HttpServletResponse -** para devolver datos en formato texto al cliente. Antes de acceder al writer, el ejemplo selecciona la cabecera del tipo del contenido. Al final del método **processRequest**, después de haber enviado la respuesta, el **Writer** se cierra.

El fichero **index.jsp** asociado al servlet :

```
<%--
    Document   : index
    Created on : 18-oct-2012, 21:33:21
    Author     : Daniel
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
   <BODY bgcolor="cyan">
          <br><br>
           <form action="ServletContrase_a">
          <table  cellspacing="2" cellpadding="1" border="1" width="50%">
      <tr>
         <td>Usuario</td>
         <td><input type="text" name="Tusuario" size="15" maxlength="10"></td>
      </tr>
      <tr>
         <td>Password</td>
         <td><input type="text" name="Tpassword" size="15" maxlength="10"></td>
      </tr>
      <tr>
         <td></td>
         <td><input type="submit" value="Enviar">
                <input type="Reset"></td>
      </tr>
      </table>
      </form>
      <br><br>

    </body>
</html>
```

Problemas con los Threads.

Los servlets HTTP normalmente pueden servir a múltiples clientes concurrentemente. Si los métodos de nuestro servlet trabajan con clientes que acceden a recursos compartidos, podemos manejar la concurrencia creando un servlet que maneje sólo una petición de cliente a la vez, también se puede sincronizar el acceso a los recursos.
Para hacer que el servlet maneje solo un cliente a la vez, tiene que implementar el interface **SingleThreadModel** además de extender la clase **HttpServlet**.
Implementar el interface **SingleThreadModel** no implica escribir ningún método extra. Sólo se declara que el servlet implementa el interface, y el servidor se asegura de que nuestro servlet sólo ejecute un método **service** cada vez.
Por ejemplo, si un Servlet acepta un nombre de usuario y un número de tarjeta de crédito, y le agradece al usuario su pedido. Si este servlet actualizara realmente una base de datos, entonces la conexión con la base de datos podría ser un recurso compartido. El servlet podría sincronizar el acceso a ese recurso, o implementar el interface **SingleThreadModel**. Si el servlet implementa este interface, el único cambio en el código es la línea mostrada en **negrita**.

```
public class SimpleServlet extends HttpServlet  implements SingleThreadModel {

   public void doPost(HttpServletRequest request, HttpServletResponse response)
                                                  throws ServletException, IOException {
         ...
   }
   ...
}
```

5.-Proporcionar Información de un Servlet

Algunas aplicaciones, como La Herramienta de Administración del Java Web Server, obtienen información sobre el servlet y la muestran. La descripción del servlet es un string que puede describir el propósito del servlet, su autor, su número de versión, o aquello que el autor del servlet considere importante.
El método que devuelve esta información es **getServletInfo**, que por defecto devuelve **null**. No es necesario sobreescribir este método, pero las aplicaciones no pueden suministrar descripción de nuestro servlet a menos que nosotros lo hagamos.
El siguiente ejemplo muestra la descripción de SimpleServlet.

```
public class SimpleServlet extends HttpServlet {
   ...
   public String getServletInfo() {
      return "El presente Servlet es un ejemplo académico";
   }
}
```

6.- El Ciclo de Vida de un Servlet.

Cada servlet tiene el mismo ciclo de vida.

- Un servidor carga e inicializa el servlet.
- El servlet maneja cero o más peticiones de cliente.
- El servidor elimina el servlet. (Algunos servidores sólo cumplen este paso cuando se desconectan).

Inicializar un Servlet

Cuando un servidor carga un servlet, ejecuta el método **init()** del servlet. La inicialización se completa antes de manejar peticiones de clientes y antes de que el servlet sea destruido.
Aunque muchos servlets se ejecutan en servidores multihilo, los servlets no tienen problemas de concurrencia durante su inicialización. El servidor llama sólo una vez al método **init()**, cuando carga el servlet, y no lo llamará de nuevo a menos que vuelva a recargar el servlet. El servidor no puede recargar un servlet sin antes haberlo destruido llamando al método **destroy()**.
Ejemplo:

```
public void init(ServletConfig config) throws ServletException  {
  super.init(config);
}
```

El método **init()** proporcionado por la clase **HttpServlet** inicializa el servlet y graba la inicialización con todos los parámetros pertinentes en un objeto de la clase ServletConfig.
En el ejemplo anterior, lo que hacemos es llamar explícitamente al método init de la clase HttpServlet, que es lo que hace de manera automática si no se sobreescribe.
Para hacer una inicialización específica de nuestro servlet, debemos sobreescribir el método **init** y lanzar una **ServletException** por si ocurre un error que haga que el servlet no pueda manejar peticiones de cliente, por ejemplo la imposibilidad de establecer una conexión requerida.
En el método init de un Servlet se debe incluir las conexiones a tablas de bases de datos o apertura de ficheros, ya que se hace una sola vez para las sucesivas consultas.

Interactuar con Clientes

Después de la inicialización, el servlet puede manejar peticiones de clientes, lo hace implementando los métodos doGet y doPost. La lógica de negocio estará escrita en dichos métodos.

Destruir un Servlet

Los servlets se ejecutan hasta que el servidor los destruye, por ejemplo, a petición del administrador del sistema. Cuando un servidor destruye un servlet, ejecuta el método **destroy()** del propio servlet. Este método, al igual que init, sólo se ejecuta una vez.
Aquí es donde se cerrarán conexiones o ficheros que se hayan abierto en init.
Si no sobrescribimos este método, se ejecutará el de la clase **HttpServlet**, de manera similar a como se hacía con init.
El servidor no ejecutará de nuevo el servlet, hasta haberlo cargado e inicializado de nuevo.

7.- Guardar el Estado del Cliente

HTTP es un protocolo sin estado. Cada conexión asociada a peticiones y a respuestas es independiente de las demás.
Entre una petición y otra (del mismo usuario), el servidor HTTP olvida la petición previa. Por eso el contenedor Web debe crear un mecanismo para almacenar la información de sesión de un mismo usuario.
El API Servlet proporciona dos formas de seguir la pista al estado de un cliente.

Cookies
Las Cookies son un mecanismo que el servlet utiliza para mantener en el cliente una pequeña cantidad de información asociada con el usuario.
En las cookies se puede guardar información del usuario como contraseñas, preferencias, enlaces visitados, etc.

Seguimiento de Sesión
El seguimiento de sesión es un mecanismo que los servlets utilizan para conservar el estado de un mismo cliente sobre la serie de peticiones durante algún periodo de tiempo.

8.- Utilizar Cookies

Las Cookies son un mecanismo mediante el cual un servidor (o un servlet, como parte de un servidor) envía información al cliente para ser almacenada. Posteriormente el servidor puede recuperar esos datos en la siguiente conexión del cliente.
Los servlet envían cookies al cliente añadiendo campos a las cabeceras de respuesta HTTP.
Los clientes devuelven las cookies automáticamente añadiendo campos a las cabeceras de peticiones HTTP.

Aspectos fundamentales de las cookies:
- Son pequeños ficheros de texto.
- Las envía el servidor Web.
- Se almacenan en el ordenador del cliente.
- El navegador las devuelve cuando se visita el mismo sitio Web o dominio.
- Todas las cookies asociadas a un dominio son enviadas en cada petición al servidor web.
- Las cookies tienen un tiempo de vida limitado y son eliminadas por el navegador del cliente al final de ese proceso.

Ventajas:
- Identificar a un usuario durante una sesión de comercio electrónico.
- Evitar escribir el nombre de usuario y la password cada vez que se visita un determinado dominio.
- Personalizar un dominio, colores, elementos.
- Publicidad enfocada.
- No son un serio problema de seguridad.
- Las cookies nunca son interpretadas o ejecutadas de ninguna forma, y no pueden usarse para insertar virus o atacar nuestro sistema.

Desventajas:

- Aunque no presentan un serio problema de **seguridad**, si que presentan un significante problema de **privacidad.**
- A ciertos usuarios no les gusta que los motores de búsqueda puedan recordar que ellos son las personas que usualmente buscan por uno u otro tópico.
- Debido a estos problemas reales de privacidad, algunos usuarios desactivan las cookies.
- Por eso, puede ser un problema que nuestra web dependa de las cookies.
- Como programadores de servlets que podemos usar cookies, no deberíamos confiar a las cookies información particularmente sensible, ya que el usuario podría correr el riesgo de que alguien accediera a su ordenador o a sus ficheros de cookies.

Para enviar una cookie:

- Crear un objeto **Cookie**.
- Seleccionar cualquier atributo.
- Enviar la cookie

Para obtener información de una cookie:

- Recuperar todas las cookies de la petición del usuario.
- Buscar la cookie o cookies con el nombre que te interesa, utilizando las técnicas de programación estándar.
- Obtener los valores de las cookies que hayas encontrado.

Cada navegador y/o sistema operativo guarda y gestiona las cookies de forma diferente.
Es interesante averiguar dónde se encuentran para tener el control total sobre ellas.

Con el API de la clase Cookie tendremos toda la información y posibilidades de operación sobre ellas:

API de Cookie

Constructor Summary

Cookie(java.lang.String name, java.lang.String value)	Constructs a cookie with a specified name and value.

Method Summary

java.lang.Object	**clone**() Overrides the standard java.lang.Object.clone method to return a copy of this cookie.
java.lang.String	**getComment**() Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.
java.lang.String	**getDomain**() Returns the domain name set for this cookie.
int	**getMaxAge**() Returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
java.lang.String	**getName**() Returns the name of the cookie.

java.lang.String	**getPath**() Returns the path on the server to which the browser returns this cookie.
boolean	**getSecure**() Returns true if the browser is sending cookies only over a secure protocol, or false if the browser can send cookies using any protocol.
java.lang.String	**getValue**() Returns the value of the cookie.
int	**getVersion**() Returns the version of the protocol this cookie complies with.
void	**setComment**(java.lang.String purpose) Specifies a comment that describes a cookie's purpose.
void	**setDomain**(java.lang.String pattern) Specifies the domain within which this cookie should be presented.
void	**setMaxAge**(int expiry) Sets the maximum age of the cookie in seconds.
void	**setPath**(java.lang.String uri) Specifies a path for the cookie to which the client should return the cookie.
void	**setSecure**(boolean flag) Indicates to the browser whether the cookie should only be sent using a secure protocol, such as HTTPS or SSL.
void	**setValue**(java.lang.String newValue) Assigns a new value to a cookie after the cookie is created.
void	**setVersion**(int v) Sets the version of the cookie protocol this cookie complies with.

Crear una Cookie

El constructor por defecto de la clase **javax.servlet.http.Cookie** crea una cookie con un nombre inicial y un valor (que siempre será un objeto). Se puede cambiar el valor posteriormente utilizando el método **setValue**.
Lo más lógico es utilizar el constructor: ***Cookie****(java.lang.String* ***name****, java.lang.String* ***value****)*
Ejemplo:

```
Cookie cok=new Cookie("user",usuario);
```

Seleccionar los Atributos de una Cookie

La clase **Cookie** proporciona varios métodos para seleccionar y modificar los valores de la cookie y sus atributos.
Estos métodos están explicados en el javadoc para la clase **Cookie**. Podéis ver un resumen en el API descrita en el punto anterior.

```
cok.setMaxAge(60*3);
```

Enviar Cookies

Las cookies se envían como cabeceras en la respuesta al cliente, se añaden con el método **addCookie** de la clase **HttpServletResponse**. Si estamos utilizando un **Writer** para devolver texto, debemos llamar a **addCookie** antes de llamar al método **getWriter** de **HttpServletResponse**.

```
response.addCookie(cok);
```

Recuperar Cookies

Los clientes devuelven las cookies como campos añadidos a las cabeceras de petición HTTP. Para recuperar una cookie, debemos recuperar todas las cookies utilizando el método **getCookies** de la clase **HttpServletRequest**.

El método **getCookies** devuelve un array de objetos **Cookie**, en el que podemos buscar la cookie o cookies que queramos. Para obtener el nombre de una cookie, utilizamos el método **getName**.

```
Cookie [] misCookies=request.getCookies();
```

La primera posición del array de Cookies la ocupa una que se crea por defecto al entrar en el servidor, es una identificación de sesión realmente (se verá después) y tiene una descripción parecida a:
misCookies[0] : JSESSIONID 8BCBFB00A604975D20B703D76B9CE251

Obtener el valor de una Cookie

Para obtener el valor de una cookie, se utiliza el método **getValue**.

```
for(int i=0;i<misCookies.length;i++) {
   Cookie unaCookie=misCookies[i];
   if(unaCookie.getName().equals(nombreC)) {
     valorC=unaCookie.getValue();
   }
  }
```

Ejemplo de Cookie

```
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-1252"/>
    <TITLE>Creando Cookies</TITLE>
  </HEAD>
  <BODY>
    <H2>Introduce nombre y te crearé una cookie.</H2>
    <form action="HazCookie">
    Nombre : <input type="text" name="nombre"
      size="15" maxlength="15">
    <input type="submit" value="Crear Cookie">
    </form>
  </BODY>
</HTML>
```

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

public class HazCookie extends HttpServlet {
  private static final String CONTENT_TYPE = "text/html; charset=windows-1252";
  public void init(ServletConfig config) throws ServletException{
    super.init(config);
}
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws
                                                          ServletException, IOException{
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();

    String usuario=request.getParameter("nombre");
    Cookie cok=new Cookie("user",usuario); //necesita un objeto para cada parámetro
    cok.setMaxAge(60*3);
    response.addCookie(cok);

    out.println("<html>");
    out.println("<head><title>HazCookie</title></head>");
    out.println("<body>");
    out.println("<p>He creado una cookie con el nombre : "+usuario+"</p>");
    out.println("</body></html>");
    out.close();
  }
}
```

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

public class LeerCookie extends HttpServlet
{
 private static final String CONTENT_TYPE = "text/html; charset=windows-1252";

 public void init(ServletConfig config) throws ServletException
 {
  super.init(config);
 }

 public void doGet(HttpServletRequest request, HttpServletResponse response) throws
                                                          ServletException, IOException

 {
  response.setContentType(CONTENT_TYPE);
  PrintWriter out = response.getWriter();

  Cookie [ ] misCookies=request.getCookies();
  String nombreC="user", valorC=null;
  for(int i=0;i<misCookies.length;i++)
  {
   Cookie unaCookie=misCookies[i];
   if(unaCookie.getName().equals(nombreC))
   {
    valorC=unaCookie.getValue();
   }
  }

  out.println("<html>");
  out.println("<head><title>LeerCookie</title></head>");
  out.println("<body>");
  if(valorC!=null){
  out.println("<p>Hola  "+valorC+"</p>");
  }else
  {
   out.println("<p>Hola usuario desconocido/p>");
  }
  out.println("</body></html>");
  out.close();
 }
}
```

Ejercicio propuesto:

Implementa un proyecto Web Application donde en un solo Servlet simulemos la creación y detección de una cookie.
La primera vez que accedamos a nuestro sitio web detectará que no llevamos cookie, dará un mensaje de bienvenida y creará una cookie con una cadena descriptiva.
Las veces posteriores detectará la presencia de la cookie, dará un mensaje alusivo, rescatará su valor y lo escribirá.
Se podrán simular las situaciones en las que la cookie no tiene tiempo de expiración, etc.

9.- Seguimiento de Sesión

Como ya se ha apuntado, el seguimiento de sesión es otro mecanismo que los servlets utilizan para mantener el estado sobre la serie de peticiones desde un mismo usuario (peticiones originadas desde el mismo navegador) durante un periodo de tiempo.

Aspectos fundamentales de las sesiones:

- Es otra alternativa a las Cookies para suplir las carencias del protocolo HTTP.
- La información se almacena en el servidor.
- Se utiliza la interfaz HttpSession, de la API de Servlets.
- El uso de sesiones trae asociada una búsqueda en el servidor dependiendo de la petición del cliente.
- Cuando el cliente tiene una sesión abierta y se vuelve a conectar, el Servidor conoce el espacio de memoria del servlet que corresponde con la sesión y la utiliza.
- Internamente usa cookies para relacionar cliente con espacio de memoria, si es posible; si no puede utilizar otros métodos (como reescritura de URL).
- Usar sesiones en servlets es bastante sencillo.
- Las sesiones son compartidas por los servlets a los que accede el cliente. Esto es conveniente para aplicaciones compuestas por varios servlets.
- Si la sesión se ha diseñado para que persista más allá del cierre del navegador, en la cookie viajará el id de la sesión. Al volver al mismo sitio web, la cookie llevará el id previo y el servidor nos reconocerá con una sesión pendiente.

Para utilizar el seguimiento de sesión debemos:

- Obtener una sesión (un objeto **HttpSession**) para un usuario.
- Almacenar u obtener datos desde el objeto **HttpSession**.
- Invalidar la sesión (opcional).

API de HttpSession

Method Summary

java.lang.Object	**getAttribute**(java.lang.String name) Returns the object bound with the specified name in this session, or null if no object is bound under the name.
java.util.Enumeration	**getAttributeNames**() Returns an Enumeration of String objects containing the names of all the objects bound to this session.
long	**getCreationTime**() Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
java.lang.String	**getId**() Returns a string containing the unique identifier assigned to this session.
long	**getLastAccessedTime**() Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT, and marked by the time the container recieved the request.
int	**getMaxInactiveInterval**() Returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
ServletContext	**getServletContext**() Returns the ServletContext to which this session belongs.
HttpSessionContext	**getSessionContext**() **Deprecated.** *As of Version 2.1, this method is deprecated and has no replacement. It will be removed in a future version of the Java Servlet API.*
java.lang.Object	**getValue**(java.lang.String name) **Deprecated.** *As of Version 2.2, this method is replaced by getAttribute(java.lang.String).*
java.lang.String[]	**getValueNames**() **Deprecated.** *As of Version 2.2, this method is replaced by getAttributeNames()*
void	**invalidate**() Invalidates this session then unbinds any objects bound to it.
boolean	**isNew**() Returns true if the client does not yet know about the session or if the client chooses not to join the session.

void	**putValue**(java.lang.String name, java.lang.Object value) **Deprecated.** *As of Version 2.2, this method is replaced by setAttribute(java.lang.String, java.lang.Object)*
void	**removeAttribute**(java.lang.String name) Removes the object bound with the specified name from this session.
void	**removeValue**(java.lang.String name) **Deprecated.** *As of Version 2.2, this method is replaced by removeAttribute(java.lang.String)*
void	**setAttribute**(java.lang.String name, java.lang.Object value) Binds an object to this session, using the name specified.
void	**setMaxInactiveInterval**(int interval) Specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Obtener una Sesión

El método **getSession()** del objeto **HttpServletRequest** devuelve una sesión de usuario. Cuando llamamos al método con su argumento **create** como **true (**ver API) , la implementación creará una sesión si no está previamente creada ; si ya lo está, la localiza y la abre. Si el argumento es **false,** se indica que la página no puede tener sesiones.
Para mantener la sesión correctamente, debemos llamar a **getSession** antes de escribir cualquier respuesta. (Si respondemos utilizando un **Writer**, entonces debemos llamar a **getSession** antes de acceder al **Writer**, no solo antes de enviar cualquier respuesta).

En el ejemplo posterior :

```
HttpSession session = request.getSession(true);
```

Almacenar y Obtener Datos desde la Sesión

El Interface **HttpSession** proporciona métodos para crear y recuperar sesiones.

setAttribute(java.lang.String name, java.lang.Object value)
Establece un nombre de atributo y un valor para la sesión.
El nombre debe ser un String, y el valor cualquier objeto Java.

```
session.setAttribute("accessCount", accessCount);
```

java.lang.Object **getAttribute**(java.lang.String name)
Devuelve el objeto especificado por su nombre como parámetro.

```
Integer accesos =(Integer)session.getAttribute("accessCount");
```

boolean **isNew( )**
Indica si el cliente tiene una sesión previa abierta o no.

```
if (session.isNew()){....}
```

Aunque no creemos una sesión propia, de forma implícita el servidor crea una por defecto, que es la que observamos en el servlet anterior con forma de cookie.

Invalidar la Sesión

Una sesión de usuario puede ser invalidada manual o automáticamente, dependiendo de donde se esté ejecutando el servlet. (Por ejemplo, el Java Web Server, invalida una sesión cuando no hay peticiones de página por un periodo de tiempo determinado).
Invalidar una sesión significa eliminar el objeto **HttpSession** y todos sus valores del sistema.
Para invalidar manualmente una sesión, se utiliza el método **invalidate()** de "session".
session.invalidate();
Algunas aplicaciones tienen un punto natural en el que invalidar la sesión.

Comunicación entre Servlets

Utilizando el mecanismo de las sesiones podemos conseguir comunicar datos entre Servlets.
Como ya se ha apuntado, las sesiones son espacios de memoria comunes a todos los servlets que forman el sitio web. Si disponemos de un objeto Session llamado Datos, que sería un ArrayList de una clase Cuadro, podríamos establecer el siguiente ejemplo :

```
HttpSession Datos = request.getSession(true);
Datos.setAttribute("ArrayCuadros",ArrayCuadros);
```

y se puede utilizar en otro Servlet

```
ArrayCuadros=(ArrayList)Datos.getAttribute("ArrayCuadros");
```

Llamar a un Servlet desde otro

La interfaz RequestDispatcher se utiliza para encaminar solicitudes a otros Servlet o JSP dentro de la misma aplicación, permitiendo delegar el procesamiento de la solicitud a ellos.
Es una típica acción de un Servlet Controlador (MVC) que se encarga de atender las peticiones y dirigir la aplicación.

```
ServletContext sc=getServletContext();
RequestDispatcher rd;
rd=sc.getRequestDispatcher("/OtroServlet");
rd.forward(request,response);
```

Objetos implicados:

public interface **ServletContext**
Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

ServletContext	**getContext**(java.lang.String uripath) Returns a ServletContext object that corresponds to a specified URL on the server.

This method allows servlets to gain access to the context for various parts of the server, and as needed obtain RequestDispatcher objects from the context. The given path must be begin with "/", is interpreted

relative to the server's document root and is matched against the context roots of other web applications hosted on this container.
In a security conscious environment, the servlet container may return null for a given URL.

RequestDispatcher	**getRequestDispatcher**(java.lang.String path) Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.

Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path. A RequestDispatcher object can be used to forward a request to the resource or to include the resource in a response. The resource can be dynamic or static.
The pathname must begin with a "/" and is interpreted as relative to the current context root. Use getContext to obtain a RequestDispatcher for resources in foreign contexts. This method returns null if the ServletContext cannot return a RequestDispatcher.

Ejemplo de getServletContext(): **org.apache.catalina.core.ApplicationContextFacade@16adb67**

public interface **RequestDispatcher**
Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. The servlet container creates theRequestDispatcher object, which is used as a wrapper around a server resource located at a particular path or given by a particular name.

void	**forward**(ServletRequest request, ServletResponse response) Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.

Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server. This method allows one servlet to do preliminary processing of a request and another resource to generate the response.
For a RequestDispatcher obtained via getRequestDispatcher(), the ServletRequest object has its path elements and parameters adjusted to match the path of the target resource.
forward should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an IllegalStateException. Uncommitted output in the response buffer is automatically cleared before the forward.
The request and response parameters must be either the same objects as were passed to the calling servlet's service method or be subclasses of the
ServletRequestWrapper or ServletResponseWrapper classes that wrap them.

Ceder el control desde un servlet hacia otro o una jsp

Es tan sencillo como escribir:

response.sendRedirect("OfertaPartidos2012JSP.jsp");

Ejemplo de Sesión

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import java.util.*;
public class MostrarSession extends HttpServlet {
  public void doGet(HttpServletRequest  request, HttpServletResponse response)
                                                          throws ServletException, IOException{
    HttpSession session = request.getSession(true); //hay un sólo objeto por sesión
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String cabecera;
    Integer accessCount = new Integer(0);
    if (session.isNew()) {
      cabecera = "Bienvenido, Nuevo Usuario";
    }
    else {
      cabecera = "Bienvenido Usuario";
      Integer accesos =(Integer)session.getAttribute("accessCount");
      if (accesos != null) {
          accessCount =new Integer(accesos.intValue()+ 1);
      } }
    session.setAttribute("accessCount", accessCount); //se pueden añadir varios atributos al objeto
    out.println("<HTML><BODY BGCOLOR='#FDF5E6'>\n" +
      "<H1 ALIGN='CENTER'>"+ cabecera +"</H1>\n"+
      "<H2>Información de tu Session:</H2>\n" +
      "<TABLE BORDER=1 ALIGN=CENTER>\n"+
      "<TR BGCOLOR='#FFAD00'>\n" +
      "<TH>Información<TH>Valor\n" +
      "<TR>\n <TD>ID <TD>"+ session.getId() +
      "\n <TR>\n <TD>Fecha de Creación\n <TD>" +
      new Date(session.getCreationTime()) + "\n" +
      "<TR>\n<TD>Fecha de último acceso\n<TD>" +
      new Date(session.getLastAccessedTime())+
      "\n<TR>\n" +
      "<TD>Número de accesos previos \n <TD>" +
      accessCount + "\n</TABLE>\n </BODY></HTML>");
  }
  public void doPost(HttpServletRequest request, HttpServletResponse response)throws
ServletException, IOException {
    doGet(request, response);
  }
}
```

JSP (Java Server Pages)

1.- Fundamentos.

JSP es una tecnología basada en Java que simplifica el proceso de desarrollo de sitios web dinámicos. Las *Java Server Pages* son ficheros de texto (normalmente con extensión *.jsp*) que sustituyen a las páginas HTML tradicionales. Los ficheros JSP contienen etiquetas HTML y código embebido que permite al diseñador de la página web acceder a datos desde código Java que se ejecuta en el servidor.
Cuando la página JSP es requerida por un navegador a través de una petición HTTP, las etiquetas HTML permanecen inalterables, mientras que el código que contiene dicha página es ejecutado en el servidor, generando contenido dinámico que se combina con las etiquetas HTML antes de ser enviado al cliente. Este modo de operar implica una separación entre los aspectos de presentación de la página y la lógica de programación contenida en el código.
El objetivo de la tecnología JSP es dar soporte a la separación entre lógica de presentación y la de negocio o programación.

- Los diseñadores Web pueden diseñar y actualizar sus páginas sin necesidad de saber programar en Java.
- Los programadores pueden escribir código sin tener que preocuparse por el diseño de las páginas Web.

Ejemplo:

```
<HTML>
<HEAD>
<TITLE>Bienvenido al mundo JSP</TITLE>
</HEAD>
<BODY>
<H1>Bienvenido al mundo JSP</H1>
<%
java.util.Date fecha = new java.util.Date();
out.println("La fecha es : "+fecha);
%>
</BODY>
</HTML>
```

JSP es una tecnología híbrida basada en *template systems*. Al igual que ASP, SSJS y PHP puede incorporar scripts para añadir código Java directamente a las páginas *.jsp*, pero también implementa, al estilo de ColdFusion, un conjunto de etiquetas que interaccionan con los objetos Java del servidor, sin la necesidad de que aparezca código fuente en la página.
JSP se implementa utilizando la tecnología Servlet. Cuando un servidor web recibe una petición de una página *.jsp*, la redirecciona a un proceso especial dedicado a manejar la ejecución de servlets (*servlet container*). En el contexto de JSP, este proceso se llama *JSP container*.

Beneficios que aporta JSP

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

Mejoras en el rendimiento:

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

Soporte de componentes reutilizables:

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

Separación entre código de presentación y código de implementación:

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

División del trabajo:

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

Partes de una JSP.

- La mayor parte de una página JSP es un patrón de texto HTML.
- Otra parte es donde se aloja el código Java, el cual va entre <% ..... %>.
- Esta página debe almacenarse con extensión .jsp
- Sobre todo nos facilita escribir código sin tanto *out.println(" ");* para componer toda la página HTML.

2.- Mecanismos de ejecución de las JSP.

— Las páginas JSP se deben alojar en directorios donde el Web Server las pueda localizar, normalmente en la misma jerarquía de directorio de las páginas HTML.

— La primera vez que se realiza una petición de una página JSP, el contenedor Web convierte el fichero JSP en un servlet que pueda responder a la petición HTTP.

  – En primer lugar traduce la página JSP en un fichero fuente Java que contiene la definición de una clase servlet.
  – En segundo lugar el contenedor Web compila el código fuente del servlet en una clase Java (.class).
  – El bytecode de esta clase Java se carga en la JVM del contenedor Web utilizando un cargador de clases.

- En el tercer paso, el contenedor Web crea una instancia de la clase servlet y lleva a cabo la fase de inicialización del ciclo de vida invocando al método especial jspInit.
- Finalmente, el contenedor Web puede invocar el método _jspService de la página JSP convertida, de modo que pueda servir las peticiones HTTP del cliente.

Ejemplo:

```
<HTML>
<HEAD><TITLE>Rapel</TITLE></HEAD>
<BODY>
Hoy tendrás un día :
<% if (Math.random() < 0.5) { %>
Buenísimo <B>SUERTE</B> increíble!
<% } else { %>
Malísimo <B>GAFE</B> total!
<% } %>
</BODY>
</HTML>
```

3.- Variables implícitas.

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

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

— **request :** El objeto HttpServletRequest asociado con la petición.
— **response :** El objeto HttpServletResponse asociado con la respuesta que se envía al navegador.
— **out :** El objeto JspWriter asociado al flujo del salida de la respuesta.
— **session :** El objeto HttpSession asociado con la sesión de cierto usuario. Esta variable sólo tiene sentido si la página JSP forma parte de una sesión HTTP

- **application :** El objeto ServletContext de la aplicación Web.
- **config :** El objeto ServletConfig asociado con el servlet para esta página JSP.
- **pageContext :** Este objeto encapsula el entorno de una petición para esta página JSP.
- **page :** Esta variable es equivalente a la variable this.
- **exception :** Objeto que fue lanzado por alguna otra página JSP. Solo disponible en una página de error.
- Los nueve objetos proporcionados por JSP se clasifican en cuatro categorías: objetos relacionados con el servlet correspondiente a la página *.jsp*, objetos relacionados con la entrada y salida de la página, objetos que proporcionan información sobre el contexto y objetos para manejo de errores.
- Estas cuatro categorías tienen la funcionalidad común de establecer y recuperar valores de atributos como mecanismo de intercambio de información. Los métodos estándares para manejo de atributos son los que se muestran en la siguiente tabla.

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

4.- Elementos de las JSP's.

- Los elementos están empotrados entre las marcas <% %> y son procesadas por el motor JSP durante la traducción de la página JSP.
- Cualquier otro texto en la página JSP se considerará parte de la respuesta y se copiará textualmente al flujo de respuesta HTTP que se envía al navegador Web.
- Hay 5 tipos de elementos de Script.
  - Comentarios : <%-- comentario --%>
  - Etiqueta de directiva : <%@ directiva %>
  - Etiqueta de declaración : <%! declaración %>
  - Etiqueta de scriptlet : <% código %>
  - Etiqueta de expresión : <%= expresión %>.

Comentarios.

Hay 3 tipos de comentarios en una página JSP:

- Comentarios HTML. <!-- comentario. HTML -- >
- Comentarios de página JSP. <%-- comentario. --%>
- Comentarios Java. <% /* comentario */ %>.

Scriptlet.

<% *scriptlet* %>

- Se utilizan para encapsular código Java que será insertado en el servlet correspondiente a la página *.jsp*.
- Los scriptlets pueden contener cualquier tipo de sentencias Java, pudiendo dejar abiertos uno o más bloques de sentencias que deberán ser cerradas en etiquetas de scritplets posteriores.
- Se pueden incluir sentencias condicionales *if-else if-else*,

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

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

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

Declaraciones.

- Permite incluir miembros en la clase servlet de la JSP, tanto atributos como métodos.
- Las declaraciones se usan para definir variables y métodos específicos para una página *.jsp*. Las variables y los métodos declarados son accesibles por cualquier otro elemento de script de la página, aunque estén codificados antes que la propia declaración.
- Tiene un uso bastante dudoso, ya que se demuestra que sin declaración, las variables pueden ser usadas en otro bloque de script.
- La sintaxis es:

<%! *declaracion (es)* %>

Algunos ejemplos se muestran a continuación.

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

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

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

Otros ejemplos:

```
<%! public int Contador = 0; %>
<%! public void incrementa(int cantidad){
Contador=Contador+cantidad;}
%>.
```

— Un uso importante de la declaración de métodos es el manejo de eventos relacionados con la inicialización y destrucción de la página *.jsp*. La inicialización ocurre la primera vez que el contenedor JSP recibe una petición de una página *.jsp* concreta.
— La destrucción ocurre cuando el contenedor JSP descarga la clase de memoria o cuando el contenedor JSP se reinicia.
— Estos eventos se pueden controlar sobrescribiendo dos métodos que son llamados automáticamente: **jspInit()** y **jspDestroy()**.

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

Obviamente su codificación es opcional.

Expresiones.

— Contiene código Java que se evalúa durante la petición HTTP. El resultado de la expresión se incluye en el flujo de respuesta HTTP.
— Las expresiones están ligadas a la generación de contenidos. Se pueden utilizar para imprimir valores de variables o resultados de la ejecución de algún método. Todo resultado de una expresión es convertido a un String antes de ser añadido a la salida de la página. Las expresiones no terminan con un punto y coma (;).

<%= *expresión* %>

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

*Diez es <%= (2*5) %>*

Bienvenido Sr. <%=name %>

Fecha actual: <%= new java.util.Date() %>

Directivas.

— Especifica una información que afectará a la traducción de la página JSP.
— Conjunto de etiquetas que contienen instrucciones para el contenedor JSP con información acerca de la página en la que se encuentran. No producen ninguna salida visible, pero afectan a las propiedades globales de la página *.jsp* que tienen influencia en la generación del servlet correspondiente.
— Las directivas JSP van encerradas entre **<%@ .... %>**.

<%@ page session=“false” %>
<%@ page import =“java.util.Date” %>
<%@ include file = “copyright.html ” %>

Directiva page

— La directiva **page** nos permite definir uno o más de los siguientes atributos:

import="*package.class*s" o
import="*package.class1*,...,*package.class*N"**.

Esto nos permite especificar los paquetes que deberían ser importados. Por ejemplo:
<%@ page import="java.util.*" %>

El atributo **import** es el único que puede aparecer múltiples veces.

contentType="*MIME-Type*" o
contentType="*MIME-Type*;
charset=*Character-Set*"
Esto especifica el tipo MIME de la salida. El valor por defecto es **text/html**. Por ejemplo, la directiva:
<%@ page contentType="text/plain" %>

tiene el mismo valor que el scriptlet

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

session="true|false".
Un valor de **true** (por defecto) indica que la variable predefinida **session** (del tipo **HttpSession**) debería unirse a la sesión existente si existe una, si no existe se debería crear una nueva sesión para unirla. Un valor de **false** indica que no se usarán sesiones, y los intentos de acceder a la variable **session** resultarán en errores en el momento en que la página JSP sea traducida a un servlet.

buffer="*size*kb|none".
Esto especifica el tamaño del buffer para el **JspWriter** out. El valor por defecto es específico del servidor, debería ser de al menos **8**kb.

autoflush="true|false".
Un valor de **true** (por defecto) indica que el buffer debería descargase cuando esté lleno. Un valor de **false**, raramente utilizado, indica que se debe lanzar una excepción cuando el buffer se sobrecargue. Un valor de **false** es ilegal cuando usamos
buffer="none"

extends="*package.class*".
Esto indica la superclase del servlet que se va a generar. Debemos usarla con extrema precaución, ya el servidor podría utilizar una superclase personalizada.

info="*message*".
Define un string que puede usarse para ser recuperado mediante el método **getServletInfo**.

errorPage="*url*".
Especifica una página JSP que se debería procesar si se lanzará cualquier **Throwable** pero no fuera capturado en la página actual.

isErrorPage="true|false".
Indica si la página actual actúa o no como página de error de otra página JSP. El valor por defecto es **false**.

Como resumen, los posibles valores de los atributos se muestran a continuación.

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

Ejemplo de utilización de page

```
<HTML>
<HEAD>
<TITLE>Mi primera página  JSP</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<%@ page info="Probando página JSP....." %>
<%@ page language="java" %>
<%@ page contentType="text/plain; charset=ISO-8859-1" %>
<%@ page import="java.util.*, java.awt.*" %>
<%@ page session="false" %>
<%@ page buffer="12kb" %>
<%@ page autoFlush="true" %>
<%@ page isThreadSafe="false" %>
<%@ page errorPage="/webapps/varios/error.jsp" %>
<%@ page isErrorPage="false" %>
Hola a todos!
</BODY>
</HTML>
```

Directiva include

— Esta directiva nos permite incluir ficheros en el momento en que la página JSP es traducida a un servlet.
— Los ficheros pueden ser documentos estáticos como páginas HTML, o documentos dinámicos como páginas JSP.
— Facilita la reutilización de un documento en diversas páginas.
— El fichero incluido es identificado mediante una URL y la directiva tiene el efecto de remplazarse a sí misma con el contenido del archivo especificado. Puede ser utilizada para incluir cabeceras y pies de páginas comunes a un desarrollo dado.

Ejemplo: <%@ include file="url relativa" %>.

5.- Gestión de Excepciones en JSP

— En cuanto al manejo de excepciones, cuando surge un error durante la ejecución de una página *.jsp*, puede hacerse de tres modos distintos :
  – El comportamiento por defecto es el de mostrar un mensaje en la ventana del navegador
  – También se puede utilizar el atributo *errorPage* de la directiva *page* para especificar una página de error alternativa

    <%@ page errorPage="errorpge.jsp" %>

    Y hace falta otra página, indicando que es una página de error,

    <%@ page isErrorPage="true" %>
    <h1> El error es <%= exception.getMessage() %>
    Volver.
  – La tercera opción es utilizar el mecanismo de manejo de excepciones de Java (Ver apuntes de excepciones) usando scriptlets.

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

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Page</title>
   </head>
      <BODY BGCOLOR="#FFFFFF">
      <%! int x = 14; %>
      <%! public long factorial (long numero) throws IllegalArgumentException {
        if ((numero < 0) || (numero > 20))
          throw new IllegalArgumentException("Fuera de Rango");
        if (numero == 0) return 1;
          else return numero * factorial(numero - 1); } %>
      <% try {
          long resultado = factorial (x); %>
          <p align=center> <%= x %>! = <%= resultado %></p>
      <% } catch (IllegalArgumentException e) { %>
          <p>El argumento para calcular el factorial esta fuera de rango.</p>
      <% } %>
   </body>
</html>
```

Otro ejemplo utilizando páginas de error:

```
<!-- JSPVotacion.jsp -->
<html>
<head>
<title>
JSP Votacion
</title>
</head>
<body>
<%@ page errorPage="paginaerror.jsp" %>
<%! String valor;%>
<% valor=request.getParameter("partido");%>
<% if(valor!=null){%>
   <p>GRACIAS POR VOTAR</p>
   HA VOTADO : <%=valor%>
 <%} else throw new Exception("Debes elegir un partido");%>
</body>
</html>
```

```
<!-- paginaerror.jsp -->
<html>
<head>
<title>
Pagina de Error
</title>
</head>
<body bgcolor="red">
<%@ page isErrorPage="true" %>
<h1><%= exception.getMessage() %> </h1>
<a href="ofertaPartidos.htm" >Volver</a>.
</body>
</html>
```

Ejercicio propuesto:

Adapta el código JSP de la función factorial anteriormente descrito a una gestión de excepciones con página de error.

6.- Sesiones.

Como ya se ha apuntado anteriormente, existe un objeto implícito para crear sesiones llamado **session**, y se puede utilizar directamente, como si ya estuviese creado.
Como ejemplo se describe el código del programa "Acierta un Número" utilizando JSP.
En el ejemplo, para la gestión de sesiones utilizamos el objeto implícito **session.**

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@page session="false"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Page</title>
   </head>
   <body>
   <p align="center"><font size="+1" color="#993300"><b><font size="+2">
         <i>Juego de Acierta un número</i></font></b></font></p>
    <br> <form action="jspAcierta.jsp">Dime un número :
    <input type="text" name="intento"
    size="5" maxlength="5">
    <br><br>
    <input type="submit" value="Enviar">
    </form>
   </body>
</html>
```

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title>JSP Acierta</title>
   </head>
<body>
<%@page import= "javax.servlet.*"%>
<%@page import= "javax.servlet.http.*"%>
<%@page import= "java.io.PrintWriter"%>
<%@page import= "java.io.IOException"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%!int prueba=0;%>
<% try
 {
  prueba = Integer.parseInt(request.getParameter("intento"));
 }
 catch(Exception e)
 {
  e.printStackTrace();
 }%>
```

```
<%!  int pensado=0;%>
<%!  Integer pensa;%>
<%!  Integer veces;%>
<%!  int vecesJugador=0;%>
<% veces=new Integer(0);%>
<% if(session.isNew()) {
       pensado=(int)(Math.random()*100)+1;
       pensa=new Integer(pensado);
       session.setAttribute("numeroPC",pensa);
       session.setAttribute("vecesJuego",veces);
   }else{
     pensa=(Integer)session.getAttribute("numeroPC");
     veces=(Integer)session.getAttribute("vecesJuego");
     pensado=pensa.intValue(); }
   vecesJugador=veces.intValue();
   vecesJugador++;
   veces=new Integer(vecesJugador);
   session.setAttribute("vecesJuego",veces); %>
<%    if(prueba<pensado) {%>
   <p>El número pensado es mayor</p>
   <p align="center"><font size="+1" color="#993300"><b><font size="+2"><i>Juego de Acierta
un número</i></font></b></font></p>
     <br> <form action='jspAcierta.jsp'>Dime un número :
     <input type='text' name='intento'
     size='5' maxlength='5'>
     <br><br>
     <input type='submit' value='Enviar'>
     </form>
  <% }
   else if(prueba>pensado) {%>
     <p>El número pensado es menor</p>
   <p align="center"><font size="+1" color="#993300"><b><font size="+2"><i>Juego de Acierta
un número</i></font></b></font></p>
     <br> <form action='jspAcierta.jsp'>Dime un número :
     <input type='text' name='intento'
     size='5' maxlength='5'>
     <br><br>
     <input type='submit' value='Enviar'>
     </form>
  <% }else {%>
      <p>Has acertado</p>
      <%session.invalidate();%>
      <A HREF='index.jsp'>Jugar de nuevo</A>
  <% }%>
   <br>llevas <%= vecesJugador%> intentos
 </body>
</html>
```

7.- Acciones

- Las *acciones* JSP usan construcciones de sintaxis XML para controlar el comportamiento del motor de servelts.
- Las acciones permiten la transferencia de control entre páginas, soportan la especificación applets de Java de un modo independiente al navegador del cliente, capacitan a las páginas *.jsp* para interaccionar con componentes JavaBeans residentes en el servidor y permiten la creación y uso de librerías de etiquetas personalizadas.
- Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans o reenviar al usuario a otra página.
- Como en XML, los nombre de elementos y atributos son sensibles a las mayúsculas

Tipos de Acciones:

jsp:include - Incluye un fichero en el momento de petición de esta página.
jsp:useBean - Encuentra o instancia un JavaBean.
jsp:setProperty - Selecciona la propiedad de un JavaBean.
jsp:getProperty - Inserta la propiedad de un JavaBean en la salida.
jsp:forward - Reenvía al peticionario a una nueva página.

Acción jsp:include

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

- Esta acción nos permite insertar ficheros en una página que está siendo generada.
- Al contrario que la directiva include, que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada..
- Proporciona una forma sencilla de incluir contenido generado por otro documento local en la salida de la página actual. En contraposición con la etiqueta *forward*, *include* transfiere el control temporalmente.
- El valor del atributo *flush*, determina si se debe vaciar el buffer de la página actual antes de incluir el contenido generado por el programa incluido.
- Se puede completar esta etiqueta con el uso de <jsp:param> para proporcionar información adicional al programa llamado.

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

Accion jsp:useBean

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

Acción jsp:forward

<jsp:forward ***page=”localURL”*** **/>**

— Se usa para transferir el control de una página *.jsp* a otra localización. Cualquier código generado hasta el momento se descarta, y el procesado de la petición se inicia en la nueva página. El navegador cliente sigue mostrando la URL de la página *.jsp* inicial.
— El valor del parámetro *page*, puede ser un documento estático, un CGI, un servlet u otra página JSP. Para añadir flexibilidad a este parámetro, se puede construir su valor como concatenación de valores de variables.

<jsp:forward page=’**<%=** “message” + statusCode + “.html” **%>**’ **/>**

— Puede resultar útil pasar pares (parámetro/valor) al destino de la etiqueta, para ello se utiliza <jsp:param .../>.

<jsp:forward *page=”factorial.jsp”*
<jsp:param *name=”numero”*
value=”25” ***/>***
...
</jsp:forward>

8.- Ejemplo de uso de acción jsp:forward

```
<html>
<!-- forward.jsp -->
<%
  double freeMem = Runtime.getRuntime().freeMemory();
  double totlMem = Runtime.getRuntime().totalMemory();
  double percent = freeMem/totlMem;
  if (percent < 0.5) {
%>
<jsp:forward page="one.jsp"/>
<% } else { %>
<jsp:forward page="two.html"/>
<% } %>
</html>
```

```
<html>
<!-- one.jsp -->
<body bgcolor="white">
<font color="red">
Uso de Memoria Virtual < 50%.
</html>
```

```
<html>
<!-- two.html -->
<body bgcolor="white">
<font color="red">
Uso de Memoria Virtual > 50%.
</html>
```

9.- Comunicación entre páginas .JSP

Existen tres formas fundamentales de comunicar datos entre páginas JSP o servlet.

1. Mediante enlaces con parámetros.
   <a href="http://localhost:8080/detalle.jsp?codigo=<%=codigo%>">
   posteriormente, en detalle.jsp se extrae el parámetro
   <%parametro=request.getParameter("codigo");%>

2. Mediante formularios, de igual manera, investigando los parámetros transmitidos.

3. Implementando sesiones.
   Como ya se ha apuntado, las sesiones son espacios de memoria comunes a todos los servlets o jsp que forman el sitio web. En JSP, existe predefinido un único objeto sesión para todo el sitio web, pero con la posibilidad de crear tantos atributos como sea necesario.

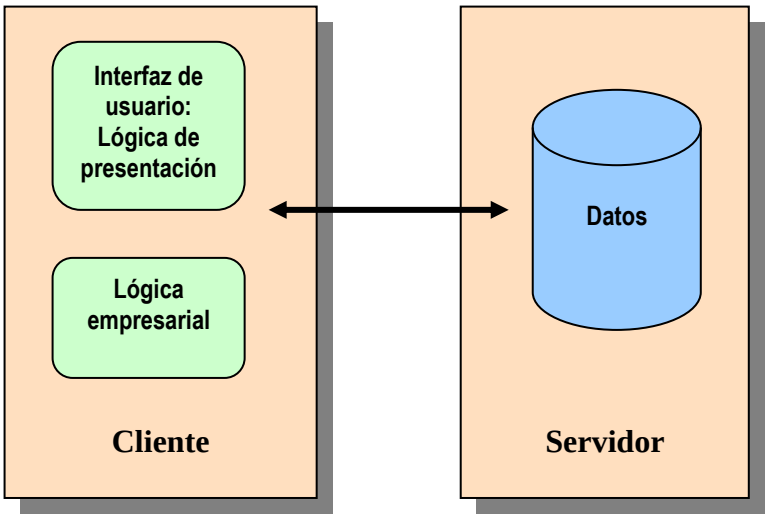
   Ejemplo, un ArrayList de Cuadros que se crea en detalle.jsp
   session.setAttribute("ArrayCuadros",ArrayCuadros);
   y se utiliza en comprarCuadro.jsp
   ArrayCuadros=(ArrayList)session.getAttribute("ArrayCuadros");

PLATAFORMA J2EE

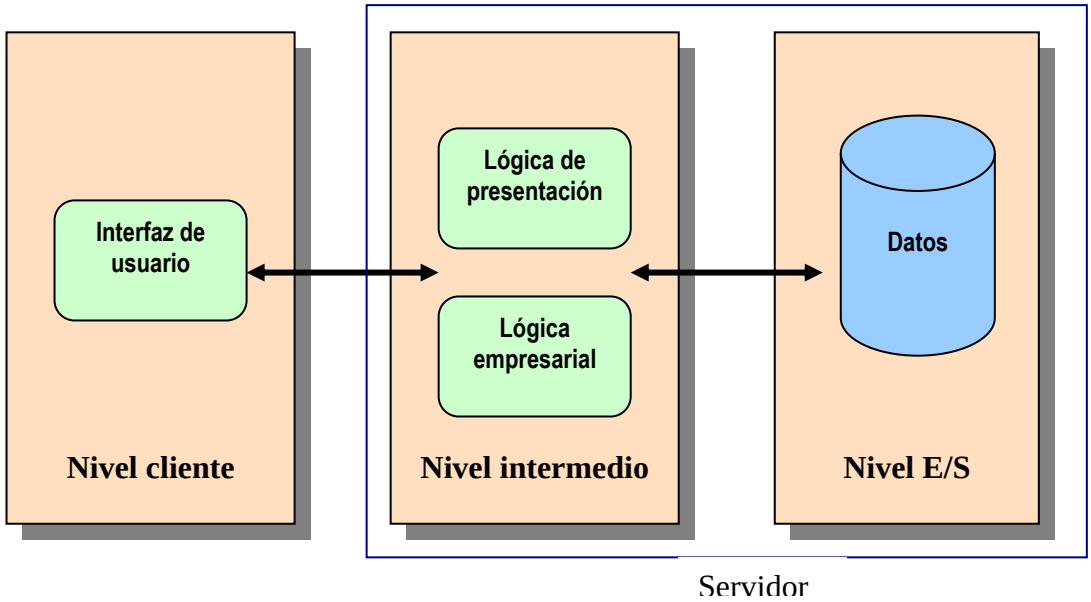
— Java™2 Platform Enterprise Edition (J2EE™) es una arquitectura para una plataforma de desarrollo en Java construida para aplicaciones empresariales (de gran envergadura) distribuidas, desarrollada por Sun Microsystems, con aportaciones de distintos desarrolladores, incluido Borland

1.-Ventajas de las aplicaciones J2EE

A principios de los años 90, los sistemas de información utilizaban, con frecuencia, una arquitectura cliente-servidor. La interfaz de usuario se ejecutaba en un ordenador personal. Este era el nivel del cliente. Los datos empresariales a los que tenía acceso la aplicación cliente se encontraban en una base de datos y los "servía" un servidor. Se conseguía una compartición y centralización de los datos, así como una mayor funcionalidad y capacidad de crecimiento.

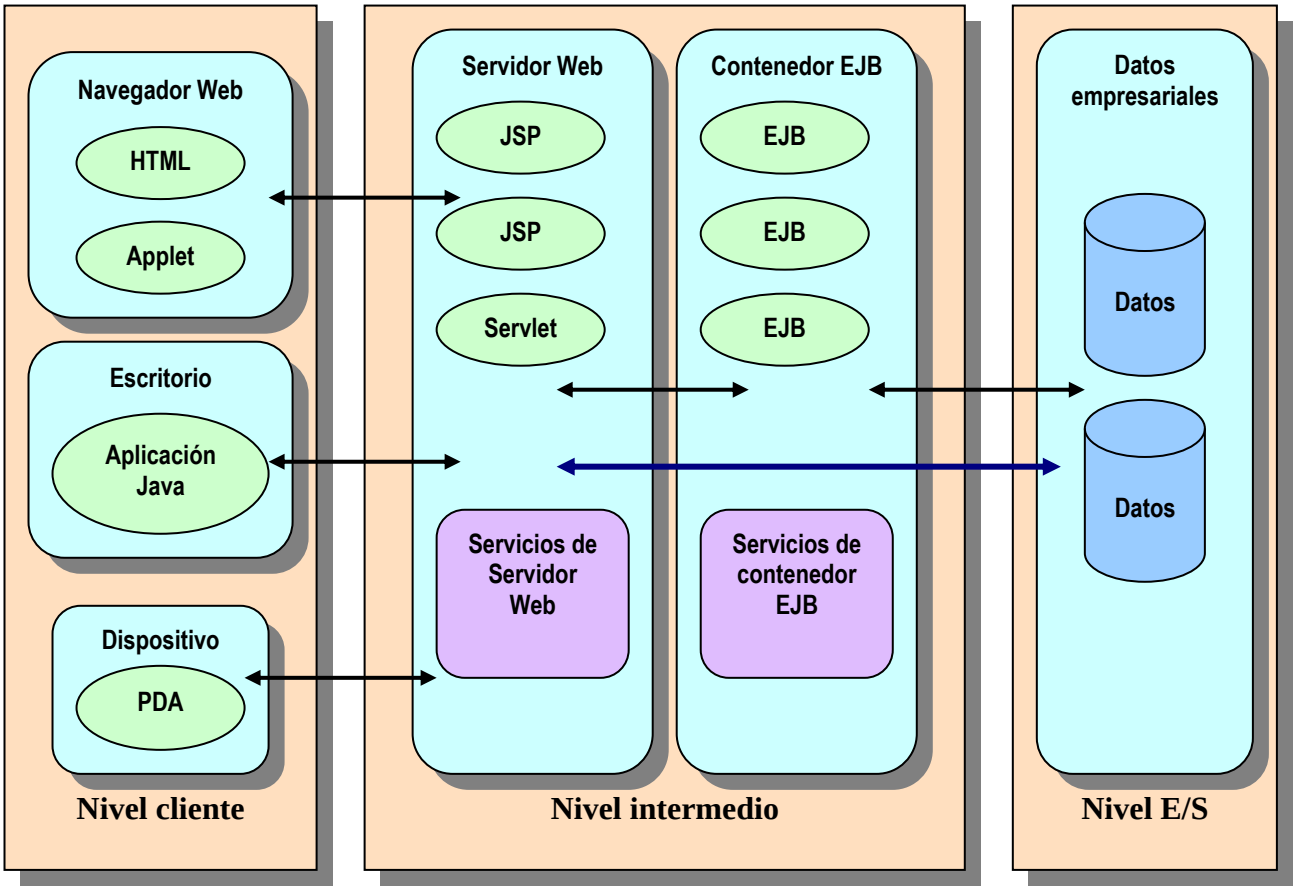


Con la experiencia, se vio que era muy complicado crear y mantener un sistema distribuido flexible utilizando el modelo cliente-servidor. Por ejemplo, la lógica empresarial (*'el programa'*) estaba en la aplicación cliente. Cada vez que había que modificar la lógica, había que instalar la aplicación cliente revisada en cada equipo. El mantenimiento era muy costoso e improductivo.
Estas aplicaciones también tenían que gestionar las transacciones, preocuparse de la seguridad y procesar los datos de forma eficaz, todo ello con una interfaz atractiva y fácil de comprender por parte del usuario.
Se necesitaban expertos en todos los campos, lo cual es un objetivo tarea muy difícil de conseguir.
Una arquitectura cliente-servidor puede ser adecuada para algunos entornos, pero la mayoría de las grandes empresas requieren mucho más de lo que puede ofrecer el modelo cliente-servidor.
Una vez que se hicieron evidentes las limitaciones del modelo cliente-servidor, se empezó a buscar un modelo mejor. El resultado es el modelo multinivel.



En el modelo multinivel, la lógica necesaria para presentar la interfaz de usuario reside en el nivel intermedio. Ahora la lógica empresarial también se encuentra en el nivel intermedio. Si es necesario realizar cambios, se pueden llevar a cabo solamente en un lugar, sin necesidad de realizarlos en todos los equipos.

Este diagrama ampliado muestra los componentes que se ejecutan en cada nivel:



La aplicación cliente en una aplicación J2EE, puede ser una página HTML o un applet que se ejecute en un visualizador, una aplicación Java de un ordenador de escritorio o, incluso, una aplicación cliente Java de algún dispositivo portátil, como un PDA (asistente personal digital) o un teléfono móvil.
El nivel intermedio puede contar con páginas de JavaServer™ o Servlets que se ejecuten en un servidor web. Estos elementos componen la lógica de presentación del servidor.
Los contenedores EJB proporcionan un entorno de ejecución para Enterprise JavaBeans™, que contiene la lógica empresarial de la aplicación. Tanto el servidor web como el contenedor Enterprise JavaBeans (EJB) ofrecen servicios a los componentes que se ejecutan en ellos. Gracias a que estos servicios siempre están disponibles, los programadores no necesitan incluirlos en los componentes que crean.
Muy pocas aplicaciones J2EE cuentan con todos estos componentes. Por ello, se pueden mezclar y hacer que coincidan de forma muy flexible para que cubran las necesidades de la empresa. Por ejemplo, se puede asignar la responsabilidad empresarial a las JSP o servlets, ejerciendo la doble función.
En el presente módulo utilizaremos Servlets y/o JSP para conectar los clientes con los datos empresariales.

El modelo multinivel permite además la separación de roles en la codificación de aplicaciones distribuidas:

- Un profesional puede encargarse de la codificación del nivel de cliente (páginas htm, applets, etc).
- Otro puede encargarse de la lógica empresarial en el servidor (qué hay que hacer con los datos que vienen en las páginas htm o qué mandamos al cliente en la página), normalmente codificado en servlets o JSP.
- Un tercer profesional encargado de administrar la base de datos que gestiona la aplicación.

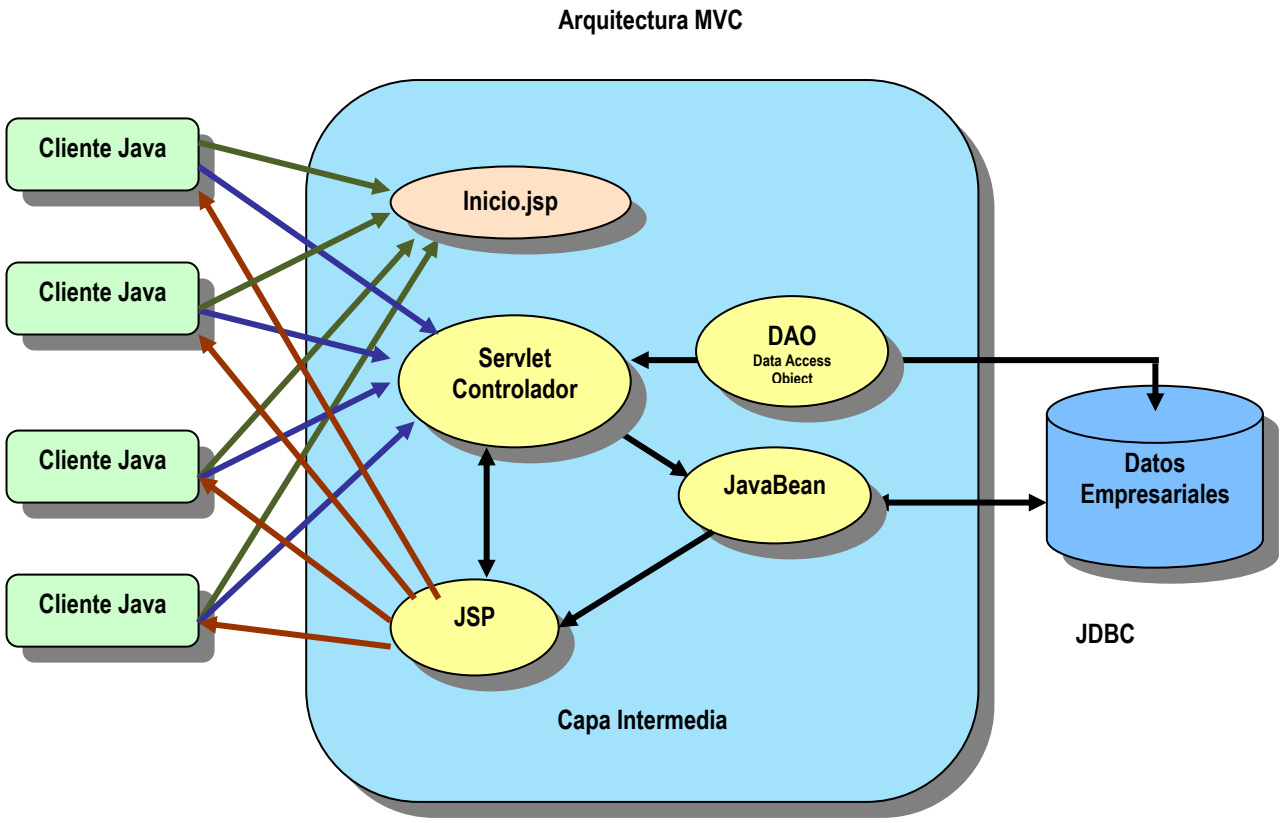
2.- Patrón MVC

Existe una recomendación o norma a la hora de diseñar arquitecturas con tecnologías web, se denomina patrón Modelo Vista Controlador.

- El patrón MVC distingue claramente el rol de cada tecnología:
  - Un servlet se usa como **Controlador**.
  - Las páginas JSP se usan como **Vistas**.
  - Las clases Java, componentes JavaBeans, ficheros de imagen o texto, los demás servlets además de las tablas JDBC se usan como el **Modelo**.
- El servlet Controlador recibe los formularios del cliente y los procesa, enviando la petición del mismo al Modelo. De igual manera extrae datos del modelo que envía a la Vista.
- La página Vista JSP envía los datos que tiene al cliente a través de un formato html

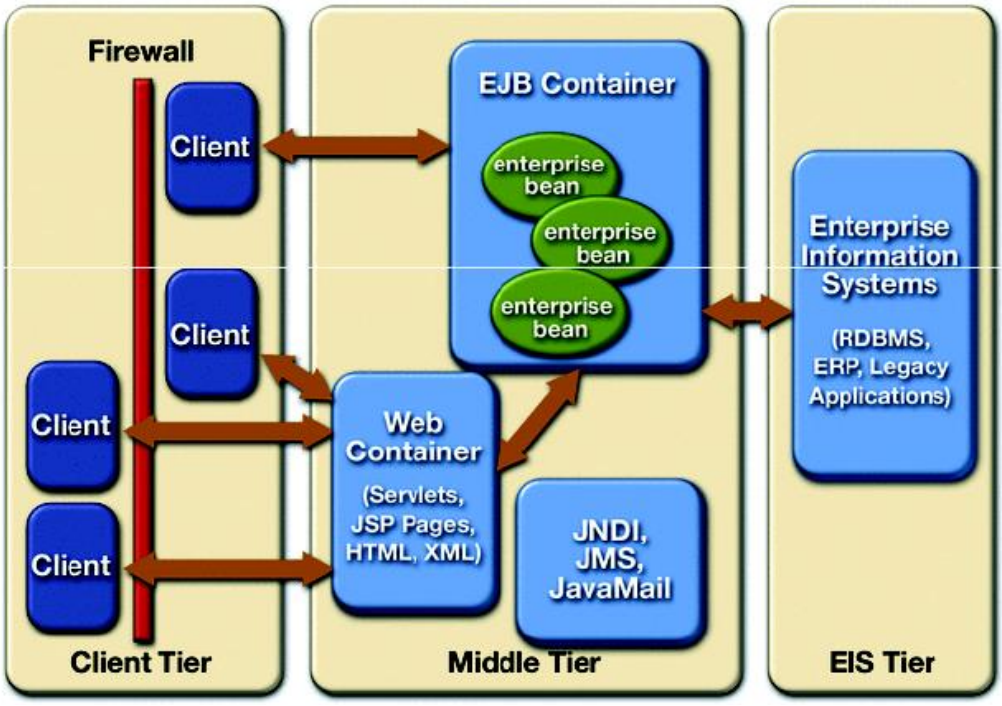
En la implementación de nuestros proyectos, utilizaremos las páginas JSP como vistas con código embebido java, e inclusión puntual de objetos JSP.
De la misma manera, los servlets siempre serán los controladores.
Si fuese necesario, diseñaremos un Dispatcher para gestionar el menú de entrada.
Aportaremos una nueva capa (DAO) para realizar las transacciones entre los controladores y las tablas de la base de datos.
La primera llamada del cliente supondrá el envío de una página *.html* o *.jsp* de inicio.

Esquema :



3.- Arquitectura en varias capas

— Se basa en aplicaciones distribuidas divididas en componentes según la función que realizan y distribuidas en múltiples capas.
  - Capa de componentes cliente que se ejecutan en la máquina del usuario.
  - Capa de componentes web que se ejecutan en un servidor JavaEE.
  - Capa de componentes de negocio que se ejecutan en un servidor JavaEE.
  - Capa de sistema de información empresarial que se ejecutan en un EIS Server.

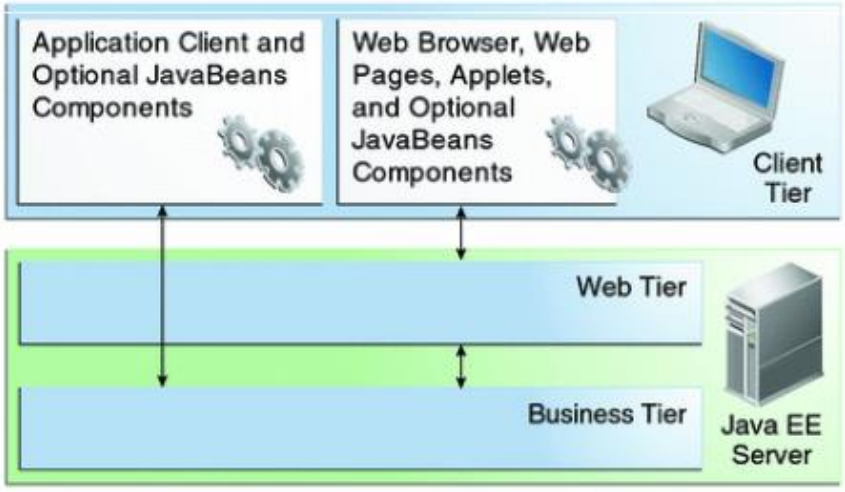


4.- Componentes

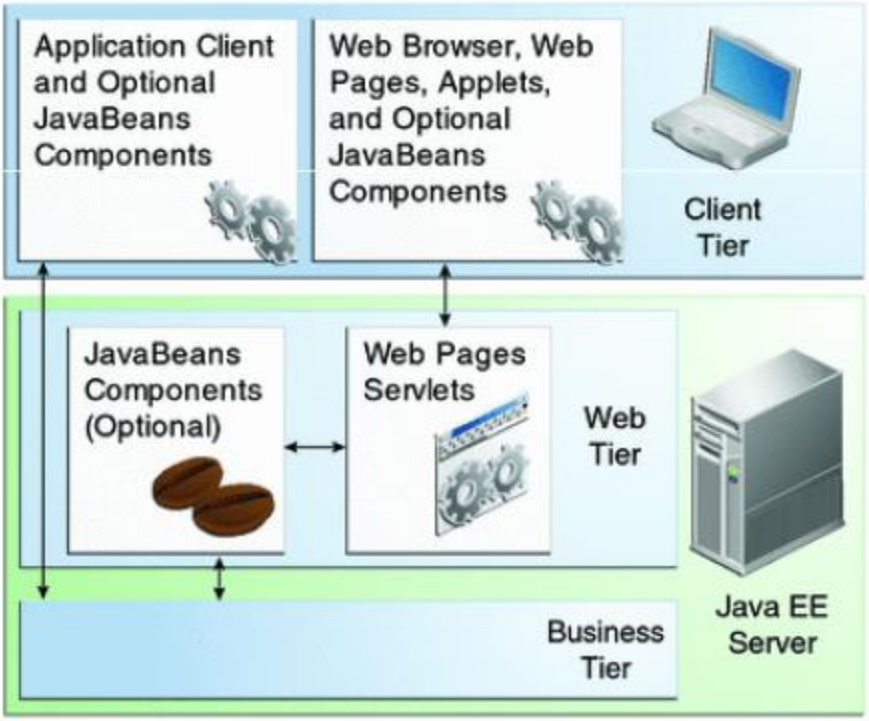
— La aplicaciones JavaEE están formadas por componentes.
— Un componente es unidad de software escrita en Java dentro de una aplicación JavaEE que se comunica con otros componentes.

— Tipos de componentes:
  – Componentes de la capa cliente.
  – Componentes web.
  – Componentes de negocio.

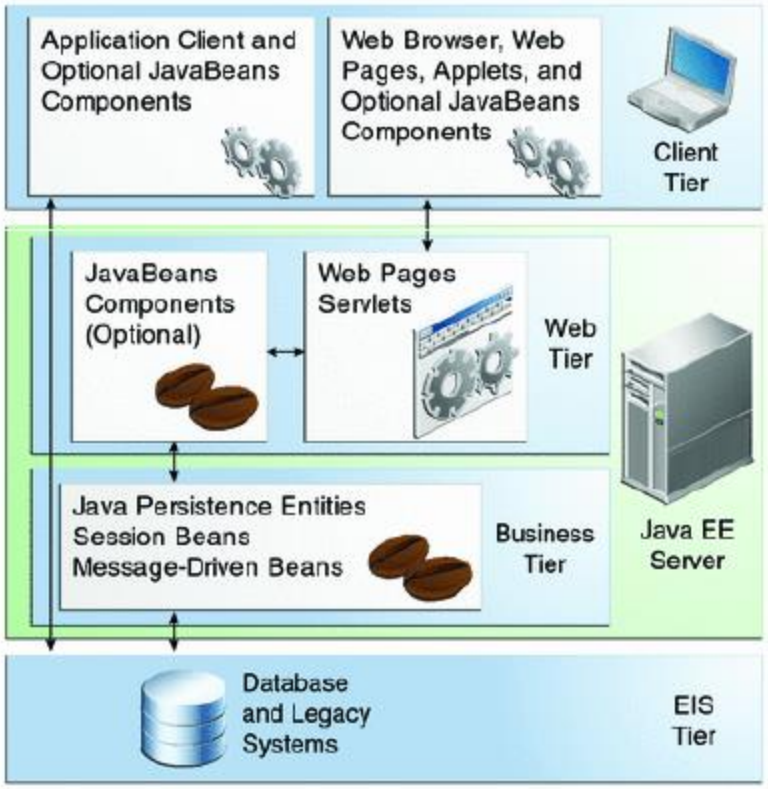
— Componentes de la capa cliente
  – Aplicaciones cliente.
  – Aplicaciones web cliente.
  – Applets.



— Componentes web
  - Servlets, JSPs y JSFs.



— Componentes de negocio
  – EJBs.



5.- Contenedores

— Contenedor:
  – Entorno de ejecución específico para un conjunto de objetos de un determinado tipo.
  – Interface entre componentes y las funcionalidades de bajo nivel que especifica Java EE.
— Implementan parte de las especificaciones JavaEE.
— Proporcionan servicios a los programadores (APIs).
— El proceso de despliegue consisten en instalar un componente JavaEE en un contenedor JavaEE.

— Otras definiciones :
  – Un contenedor es un sistema que proporciona un entorno estandarizado en periodo de ejecución, para gestionar los componentes de aplicación, JSP, Servlet,…y cualquier otro tipo de componentes Java. Estos entornos proporcionan una serie de servicios que liberan al desarrollador de ciertas tareas ya estandarizadas.
  – Un contenedor es un entorno de ejecución para un componente, el componente vive en el contenedor y éste proporciona servicios al componente.
  – De modo similar un contenedor vive dentro de un servidor de aplicaciones que le proporciona un entorno de ejecución para él y para otros contenedores.

— Tipos de contenedores
  – Contenedor de aplicaciones web.

- Contenedor de applets (navegador y un Plug-in Java).
- Contenedor web
  - Proporciona funciones de comunicación con clientes HTTP.
  - Aloja componentes (Servlets, JSPs).
- Contenedor de Enterprise JavaBeans (EJB).
  - Proporciona servicios comunes a la lógica de negocio(Distribución de objetos, Persistencia, Concurrencia,Transacciones).
  - Aloja componentes (beans).

6.- Servidores de aplicaciones

— Un servidor de aplicaciones es un producto que implementa (se ajusta) al estándar JavaEE y ofrece un contenedor Web y un contenedor EJB, como mínimo
— Implementa las especificaciones (APIs) y contenedores JavaEE.
— Proporciona el marco para ejecutar las aplicaciones ofreciendo servicios de seguridad, acceso a datos, soporte transaccional, balanceo de carga, gestión de aplicaciones, …

Oracle GlassFish Server 3.x

JBoss Application Server 7.x

IBM WebSphere Application Server 8.0

Oracle WebLogic Server

Apache Geronimo 3.0-beta-1

Fujitsu Interstage Application Server
powered by Windows Azure

Hitachi uCosminexus Application Server
v9.0

Apache TomEE 1.0