JAVA - Web

1. Tecnología Web
2. Modelo Cliente- Servidor
3. Acceso a Datos
4. Servlets
5. Seguimiento de cliente con Servlets
6. Directivas
7. Declaraciones
8. Scriptles
9. Variables Predefinidas
10. Procesamiento de un JSP
11. El motor JSP
12. EJBS
13. Ejemplos de código

Introducción a la tecnología WEB con J2EE

La Web dinámica se ha desarrollado desde un sistema de información distribuido (HTML) basado en red que ofrecía información estática hasta un conjunto de portales y aplicaciones en Internet que ofrecen un conjunto variado de servicios.

Las soluciones de *primera generación* incluyeron **CGI**, que es un mecanismo para ejecutar programas externos en un servidor web. El problema con los scripts CGI es la escalabilidad; se crea un nuevo proceso para cada petición.

Las soluciones de *segunda generación* incluyeron vendedores de servidores Web que proporcionaban plug-ins y APIs para sus servidores. El problema es que sus soluciones eran específicas a sus productos servidores. Microsoft proporcionó las páginas activas del servidor (ASP) que hicieron más fácil crear el contenido dinámico. Sin embargo, su solución sólo trabajaba con Microsoft IIS o Personal Web Server. Otra tecnología de segunda generación son los **Servlets**. Los Servlets hacen más fácil escribir aplicaciones del lado del servidor usando la tecnología Java. El problema con los CGI o los Servlets, sin embargo, es que tenemos que seguir el ciclo de vida de escribir, compilar y desplegar .

Las páginas **JSP** son una solución de *tercera generación* que se pueden combinar fácilmente con algunas soluciones de la segunda generación, creando el contenido dinámico, y haciendo más fácil y más rápido construir las aplicaciones basadas en Web que trabajan con una variedad de otras tecnologías: servidores Web, navegadores Web, servidores de aplicación y otras herramientas de desarrollo.

La tecnología Java Server Pages™ (JSP) nos permite poner segmentos de código servlet directamente dentro de una página HTML estática. Cuando el navegador carga una página JSP, se ejecuta el código del servlet y el servidor de aplicaciones crea, compila, carga y ejecuta un servlet en segundo plano para ejecutar los segmentos de código servlet y devolver una página HTML o imprimir un informe XML.

Modelo cliente-servidor

Cuando se utiliza un servicio en Internet, como consultar una base de datos, transferir un fichero o participar en un foro de discusión, se establece un proceso en el que entran en juego dos partes. Por un lado, el usuario, quien ejecuta una aplicación en el ordenador local: el denominado *programa cliente*. Este programa cliente se encarga de ponerse en contacto con el ordenador remoto para solicitar el servicio deseado. El ordenador remoto por su parte responderá a lo solicitado mediante un programa que esta ejecutando. Este último se denomina *programa servidor*. Los términos *cliente* y *servidor* se utilizan tanto para referirse a los programas que cumplen estas funciones, como a los ordenadores donde son ejecutados esos programas.

El programa o los programas cliente que el usuario utiliza para acceder a los servicios de Internet realizan dos funciones distintas. Por una parte, se encargan de gestionar la comunicación con el ordenador servidor, de solicitar un servicio concreto y de recibir los datos enviados por éste; y por otra, es la herramienta que presenta al usuario los datos en pantalla y que le ofrece los comandos necesarios para utilizar las prestaciones que ofrece el servidor.

El navegador es una especie de aplicación capaz de interpretar las órdenes recibidas en forma de código HTML fundamentalmente y convertirlas en las páginas que son el resultado de dicha orden.

Fig. Lenguajes del lado cliente y del lado servidor

Cuando pinchamos sobre un enlace hipertexto, en realidad lo que pasa es que establecemos una petición de un archivo HTML residente en el servidor (un ordenador que se encuentra continuamente conectado a la red) el cual es enviado e interpretado por nuestro navegador (el cliente).

Sin embargo, si la página que pedimos no es un archivo HTML, el navegador es incapaz de interpretarla y lo único que es capaz de hacer es salvarla en forma de archivo. Es por ello que, si queremos emplear lenguajes accesorios para realizar un sitio web, es absolutamente necesario que sea el propio servidor quien los ejecute e interprete para luego enviarlos al cliente (navegador) en forma de archivo HTML totalmente legible por él.

De modo que, cuando pinchamos sobre un enlace a una pagina que contiene un script en un lenguaje comprensible únicamente por el servidor, lo que ocurre en realidad es que dicho script es ejecutado por el servidor y el resultado de esa ejecución da lugar a la generación de un archivo HTML que es enviado al cliente.

Así pues, podemos hablar de lenguajes de lado servidor que son aquellos lenguajes que son reconocidos, ejecutados e interpretados por el propio servidor y que se envían al cliente en un formato comprensible para él. Por otro lado, los lenguajes de lado cliente (entre los cuales no

sólo se encuentra el HTML sino también JavaScript) son aquellos que pueden ser directamente "ejecutados" por el navegador y no necesitan un pre-procesamiento.

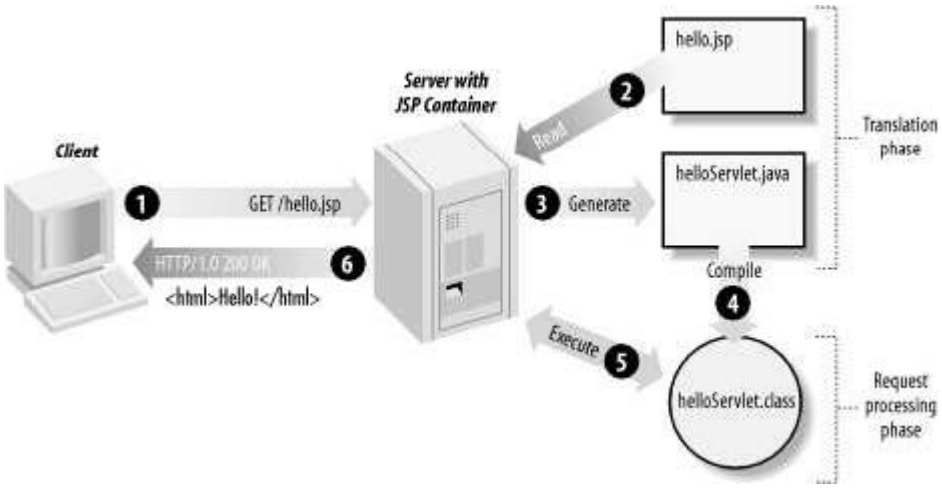


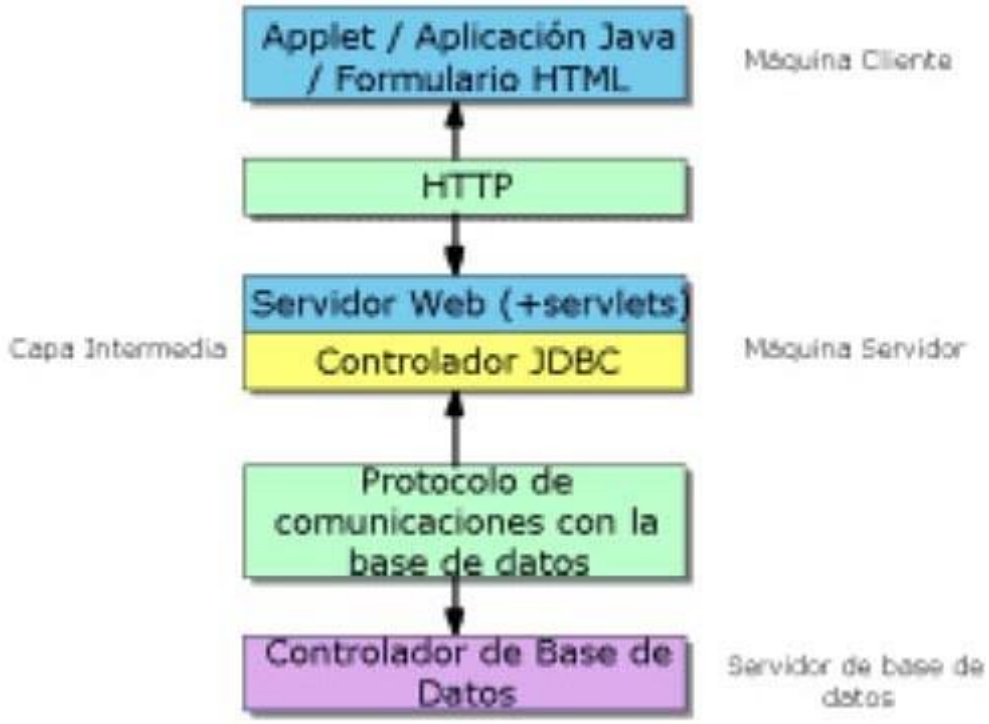
Fig. Ejecución en el modelo web cliente-servidor

En resumen, los programadores de aplicaciones JAVA mediante la especificación J2EE escriben componentes de aplicación J2EE. Un componente J2EE es una unidad de software funcional auto-contenida que se ensambla dentro de una aplicación J2EE y que se comunica con otros componentes de aplicación. La especificación J2EE define los siguientes componentes de aplicación:

- Componentes de Aplicación Cliente: Navegador y páginas *jsp* renderizadas
- Componentes JavaBeans Enterprise: *clases de negocio (EJBs)*
- Componentes Servlets y JavaServer Pages (también llamados componentes Web): *clases controlodaras*.
- Applets: *pequeñas aplicaciones que se ejecutan en el cliente*

Acceso a Datos

Una de las tareas más importantes y más frecuentemente realizadas por los ***servlets*** es la conexión a ***bases de datos*** mediante ***JDBC***. Esto es debido a que los ***servlets*** son un componente ideal para hacer las funciones de capa media en un sistema con una arquitectura de tres capas como la mostrada en la Figura:



Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los *drivers* ***JDBC*** no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de *driver*. En cualquier caso, tanto el ***Servidor http*** como el ***Servidor de Base de Datos*** pueden estar en la misma máquina, aunque en sistemas empresariales de cierta importancia esto no suele ocurrir con frecuencia.

JDBC (***Java DataBase Connectivity***) es una parte del API de ***Java*** que proporciona clases para conectarse con bases de datos. Dichas clases forman parte del package ***java.sql***, disponible en el ***jdk 1.1.7*** y en ***jdk 1.2***. El nombre ***JDBC*** es fonéticamente similar a ***ODBC*** (***Open DataBase Connectivity***), que es el estándar más extendido para conectar PCs con bases de datos.

Formas de seguir la trayectoria de los usuarios (clientes)

Los ***servlets*** permiten seguir la trayectoria de un cliente, es decir, obtener y mantener una determinada información acerca del cliente. De esta forma se puede ***tener identificado a un cliente*** (usuario que está utilizando un browser) durante un determinado tiempo. Esto es muy importante si se quiere disponer de aplicaciones que impliquen la ejecución de varios ***servlets*** o la ejecución repetida de un mismo ***servlet***. Un claro ejemplo de aplicación de esta técnica es el de los ***comercios vía Internet*** que permiten llevar un ***carrito de la compra*** en el que se van guardando aquellos productos solicitados por el cliente. El cliente puede ir navegando por las distintas secciones del comercio virtual, es decir realizando distintas conexiones ***HTTP*** y ejecutando diversos ***servlets***, y a pesar de ello no se pierde la información contenida en el carrito de la compra y se sabe en todo momento que es un mismo cliente quien está haciendo esas conexiones diferentes.

El mantener información sobre un cliente a lo largo de un proceso que implica múltiples conexiones se puede realizar de tres formas distintas:

1.1 Mediante ***cookies***
1.2 Mediante ***seguimiento de sesiones*** (***Session Tracking***)
1.3 Mediante la ***reescritura*** de URLs y paso de parámetros en formulario (***Request***)

Introducción a JSP

La tecnología JSP (*Java Server Pages*) es una especificación abierta desarrollada por Sun Microsystems como un alternativa a Active Server Pages (ASP) de Microsoft, y son un componente dominante de la especificación de Java 2 Enterprise Edition (J2EE). Muchos de los servidores de aplicaciones comercialmente disponibles (como BEA WebLogic, IBM WebSphere, Live JRun, Orion, etcétera) ya utilizan tecnología JSP.

Esta tecnología permite desarrollar páginas web con contenido dinámico y supone una evolución frente a la tecnología CGI, y los Servlets. Un fichero JSP puede contener etiquetas HTML normales, y elementos especiales para generar el contenido dinámico.

Al mismo tiempo permite una separación en n capas de la arquitectura de la aplicación web y se integra perfectamente con todas las API's empresariales de Java: JDBC, RMI (y CORBA), JNDI, EJB, JMS, JTA, ...

Estructura de una página JSP

Una página JSP es básicamente una página Web con HTML tradicional y código Java. La extensión de fichero de una página JSP es ".jsp" en vez de ".html" o".htm", y eso le dice al servidor que esta página requiere un manejo especial que se conseguirá con una extensión del servidor o un plug-in.

Un ejemplo sencillo:

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

<html>
   <head>
         <title>Hola, mundo!!</title>
   </head>
   <body>
         <h1>Hola, mundo!</h1>
         Hoy es <%= new java.util.Date() %>.
   </body>
</html>
```

Directivas

Las directivas JSP son instrucciones procesadas por el motor JSP cuando la página JSP se traduce a un servlet. Las directivas usadas en este ejemplo le dicen al motor JSP que incluya ciertos paquetes y clases. Las directivas están encerradas entre etiquetas de directiva <%@ y %>.

```
<%@ page import="javax.naming.*" %>
<%@ page import="javax.rmi.PortableRemoteObject" %>
<%@ page import="Beans.*" %>
```

Declaraciones

Las declaraciones JSP nos permiten configurar variables para su uso posterior en expresiones o scriptlets. También podemos declarar variables dentro de expresiones o scriptlets en el momento de usarlas. El ámbito es toda la página JSP, no hay concepto de variables de ejemplar. Es decir, no tenemos que declarar variables de ejemplar para usar en más de una expresión o scriptlet. Las declaraciones van encerradas entre etiquetas de declaración <%! y %>. Podemos tener varias declaraciones. Por ejemplo,

```
<%! double bonus; String text; %> .
<%! String strMult, socsec; %>
<%! Integer integerMult; %>
<%! int multiplier; %>
<%! double bonus; %>
```

Scriptlets

Los scriptlets JSP nos permiten embeber segmentos de código java dentro de una página JSP. El código embebido se inserta directamente en el servlet generado que se ejecuta cuando se pide la página. Este scriptlet usa las variables declaradas en las directivas descritas arriba. Los Scriptlets van encerrados entre etiquetas <% y %>.

```
<%
strMult = request.getParameter("MULTIPLIER");
socsec = request.getParameter("SOCSEC");
integerMult = new Integer(strMult);
multiplier = integerMult.intValue();
bonus = 100.00;
%>
```

Variables Predefinidas

Un scriptlet puede usar las siguientes variables predefinidas: session, request, response, out, e in. Este ejemplo usa la variable predefinida request, que es un objeto HttpServletRequest. De igual forma, response es un objeto HttpServletResponse, out es un objeto PrintWriter, e in es un objeto BufferedReader. Las variables predefinidas se usan en los scriptlets de la misma forma que se usan en los servelts, excepto que no las declaramos.

```
<%
strMult = request.getParameter("MULTIPLIER");
socsec = request.getParameter("SOCSEC");
integerMult = new Integer(strMult);
multiplier = integerMult.intValue();
bonus = 100.00;
%>
```

Expresiones

La expresiones JSP nos permiten recuperar dinámicamente o calcular valores a insertar directamente en la página JSP. En este ejemplo, una expresión recupera el número de socio desde el bean de entidad Bonus y lo pone en la página JSP.

```
<H1>Calculo</H1>
Número de socio:
<%= record.getSoc() %>
<P>
calculo: <%= record.getsocio() %>
<P>
```

Etiquetas específicas de JSP

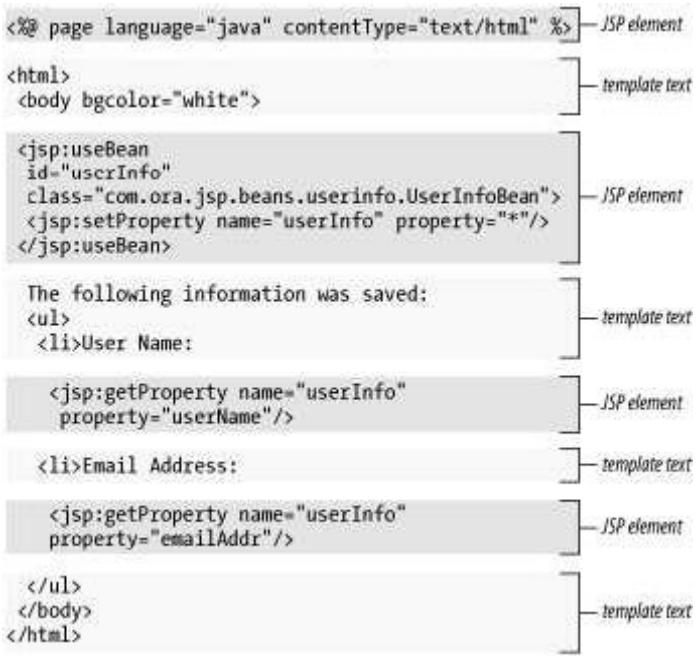
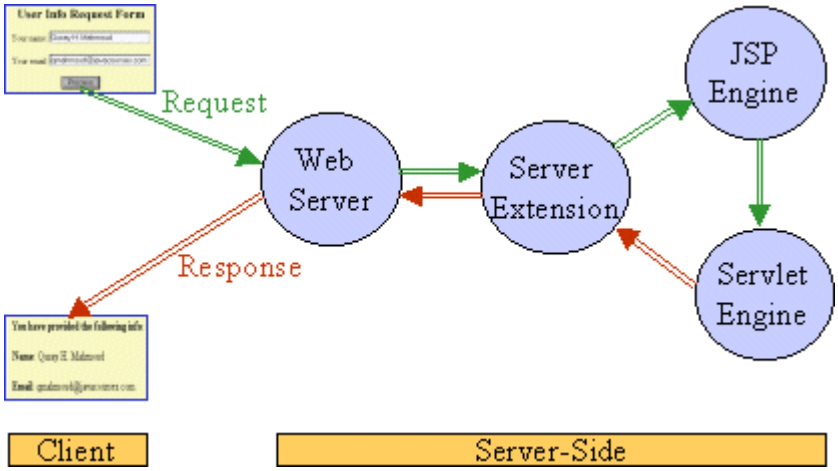


Fig. Estructura de una página JSP

La especificación JavaServer Pages define etiquetas específicas de JSP que nos permiten extender la implementación JSP con nuevas características y ocultar mucha complejidad a los diseñadores visuales que necesitan buscar la página JSP y modificarla

Procesamiento de la página JSP

Cuando se llame a la página `(date.jsp)`, será compilada (por el motor JSP) en un Servlet Java. En este momento el Servlet es manejado por el motor Servlet como cualquier otro Servlet. El motor Servlet carga la clase Servlet (usando un cargador de clases) y lo ejecuta para crear HTML dinámico para enviarlo al navegador, como se ve en la Figura 2. Para este ejemplo, el Servlet crea un objeto Date y lo escribe como un String en el objeto out, que es el stream de salida hacia el navegador.



Objetos implícitos

El Motor de JSP

El motor JSP nos ofrece instancias de un conjunto de clases. Son objetos ya establecidos, que no tenemos más que usar (no hay que instanciarlos). Deben utilizarse dentro del código Java.

Algunos objetos implícitos:

page (*javax.servlet.jsp.HttpJspPage*.)**:** Instancia del servlet de la página. Esto es sólo un sinónimo de this, y no es muy útil en Java. Fue creado como situación para el día que el los lenguajes de script puedan incluir otros lenguajes distintos de Java.

config (*javax.servlet.ServletConfig***):** Datos de configuración del servlet.

request (*javax.servlet.http.HttpServletRequest***):** Datos de la petición, incluyendo los parámetros. Este es el HttpServletRequest asociado con la petición, y nos permite mirar los parámetros de la petición (mediante getParameter), el tipo de petición (GET, POST, HEAD, etc.), y las cabeceras HTTP entrantes (cookies, Referer, etc.). Estrictamente hablando, se permite que la petición sea una subclase de ServletRequest distinta de HttpServletRequest, si el protocolo de la petición es distinto del HTTP. Esto casi nunca se lleva a la práctica.

response (*javax.servlet.http.HttpServletResponse*)**:** Datos de la respuesta. Este es el HttpServletResponse asociado con la respuesta al cliente. Como el stream de salida tiene un buffer, es legal seleccionar los códigos de estado y cabeceras de respuesta, aunque no está permitido en los servlets normales una vez que la salida ha sido enviada al cliente.

out (*javax.servlet.jsp.JspWriter***):** Flujo de salida para el contenido de la página. Este es el PrintWriter usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto response esta es una versión con buffer de PrintWriter llamada JspWriter. Podemos ajustar el tamaño del buffer, o incluso desactivar el buffer, usando el atributo buffer de la directiva page. Se usa casi exclusivamente en scriptlets ya que las expresiones JSP obtienen un lugar en el stream de salida, y por eso raramente se refieren explícitamente a out.

session (*javax.servlet.http.HttpSession***):** Datos específicos de la sesión de un usuario. Este es el objeto HttpSession asociado con la petición. Las sesiones se crean automáticamente, por esto esta variable se une incluso si no hubiera una sesión de referencia entrante. La única excepción es usar el atributo session de la directiva page para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable session causarán un error en el momento de traducir la página JSP a un servlet.

application (*javax.servlet.ServletContext***):** Datos compartidos por todas las páginas de una aplicación. El ServletContext obtenido mediante getServletConfig().getContext().

pageContext (*javax.servlet.jsp.PageContext***):** Datos de contexto para la ejecución de la página. JSP presenta una nueva clase llamada PageContext para encapsular características de uso específicas del servidor como JspWriters de alto rendimiento. La idea es que, si tenemos acceso a ellas a través de esta clase en vez directamente, nuestro código seguirá funcionando en motores servlet/JSP "normales".

exception (*java.lang.Throwable***):** Errores o excepciones no capturadas.

Ejemplo:

```
<%
String strParam = request.getParameter("nombre_del_parametro");
out.println( strParam );
%>
```

EJBs

Un EJB es un componente del lado del servidor que encapsula la lógica del negocio de una aplicación. En cualquier aplicación, los *beans enterprise* implementan los métodos de la lógica del negocio, que pueden ser invocados por clientes remotos para acceder a los servicios importantes proporcionados por la aplicación.

Beneficios

Los EJBs simplifican el desarrollo de grandes aplicaciones empresariales seguras y distribuidas por las siguientes razones:

- **Los desarrolladores pueden concentrarse en solventar la lógica del negocio**: el contenedor EJB proporciona servicios a nivel del sistema como el control de transaciones y las autorizaciones de seguridad. Por lo tanto, los desarrolladores no tienen que preocuparse de estos problemas.
- **Clientes pequeños**: Los desarrolladores no tienen que desarrollar código para las reglas de negocio o accesos a bases de datos; pueden concentrarse en la presentación del cliente. El resultado final es que los clientes son pequeños, y esto es especialmente importante para clientes que se ejecutan en pequeños dispositivos con recursos limitados.
- **Desarrollo rápido**: Los EJBs son componentes portables, y por lo tanto los ensambladores de aplicaciones pueden construir nuevas aplicaciones desde beans existentes. Las aplicaciones resultantes se pueden ejecutar en cualquier servidor compatible J2EE.

Componentes

Hay dos tipos principales de componentes EJB : **session** y **entity**. Un EJB de sesión se usa para realizar una tarea para un cliente, y un EJB de entidad es específico del dominio y se usa para representar un objeto de entidad del negocio que existe en un almacenamiento persistente. Sin embargo, los beans de entidad y de sesión tienen algunas diferencias que podemos ver en la siguiente tabla:

EJB de Sesión	EJB de Entidad
Transitorio	Persistente
Puede ser usado por un sólo cliente.	Puede ser usado por muchos clientes.
No tiene identidad	Tiene una identidad (como una clave primaria)

Desarrollar EJBs

Para desarrollar EJBs, todo bean enterprise necesita:

- Un **interface remoto** que exponga los métodos que soporta bean enterprise.
- Un **interface home** que proporciona los métodos del ciclo de vida del bean enterprise.
- Una clase de implementación, incluyendo la lógica de negocio que necesite.

EJBs contra Servlets

A primera vista, los EJBs y los Servlets son tecnologías similares porque ambos son componentes distribuidos del lado del servidor. Sin embargo, hay una diferencia importante entre los dos en el tipo de solución que ofrecen; los EJBs no pueden aceptar peticiones HTTP. En otras palabras, los EJBs no peuden servir peticiones que vienen directamente desde un navegador Web, y los servlets si pueden. Servlets y JSPs se pueden usar para implementar presentación y control web, pero al contrario que los EJBs, no pueden manejar transacciones distribuidas. Los EJBs pueden ser llamados desde cualquier cliente basado en Java.

¿Cúando usar EJBs?

Los EJBs son buenos para las aplicaciones que tienen alguno de estos requerimientos:

- **Escalabilidad**: si tenemos un número creciente de usuarios, los EJBs nos permitirán distribuir los componentes de nuestra aplicación entre varias máquinas con su localización transparente para los usuarios.
- **Integridad de Datos**: los EJBs nos facilitan el uso de transacciones distribuidas.
- **Variedad de clientes**: si nuestra aplicación va a ser accedida por una variedad de clientes (como navegadores tradicionales o navegadores WAP), se pueden usar los EJBs para almacenar el modelo del negocio, y se puede usar una variedad de clientes para acceder a la misma información.

Algunos ejemplos

Elementos de Script

En el ejemplo `date.jsp` se usa todo el nombre de la clase `Date` incluyendo el nombre del paquete, lo que podría llegar a ser tedioso. Si queremos crear un ejemplar de la clase `Date` usando simplemente: `Date today = new Date();` sin tener que especificar el path completo de la clase, usamos la directiva `page` de esta forma:

Ejemplo #1: fecha.jsp

```
<%@page import="java.util.*" %>
<HTML>
      <HEAD>
        <TITLE>JSP Example</TITLE>
      </HEAD>
      <BODY BGCOLOR="ffffcc">
        <CENTER>
        <H2>Date and Time</H2>
          <%
          java.util.Date today = new java.util.Date();
          out.println("Today's date is: "+today);
          %>
        </CENTER>
      </BODY>
</HTML>
```

Todavía hay otra forma de hacer lo mismo usando la etiqueta `<%=` escribiendo: `Today's date is: <%= new Date() %>`

Como podemos ver, se puede conseguir el mismo resultado usando diferentes etiquetas y técnicas. Hay varios elementos de script JSP. Hay algunas reglas convencionales que nos ayudarán a usar más efectivamente los elementos de Script JSP.

- Usamos `<% ... %>` para manejar declaraciones, expresiones, o cualquier otro tipo de código válido.
- Usamos la directiva `page` como en `<%@page ... %>` para definir el lenguaje de script. También puede usarse para especificar sentencias `import`. Aquí hay un ejemplo:

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

- Usamos `<%!.... %>` para declarar variables o métodos. Por ejemplo:

  ```
  <%! int x = 10; double y = 2.0; %>
  ```

- Usamos `<%=.... %>` para definir una expresión y forzar el resultado a un `String`. Por
  ejemplo: `<%= a+b %>` o `<%= new java.util.Date() %>`.
- Usamos la directiva `include` como en `<%@ include .... %>` para insertar el contenido
  de otro fichero en el fichero JSP principal. Por ejemplo:

  ```
  <%@include file="copyright.html" %>
  ```

Manejar Formularios

Una de las partes más comunes en aplicaciones de web es un formulario HTML donde el usuario introduce alguna información como su nombre y dirección. Usando JSP, los datos del formulario (la información que el usuario introduce en él) se almacenan en un objeto `request` que es enviado desde el navegador hasta el contenedor JSP. La petición es procesada y el resultado se envía a través de un objeto `response` de vuelta al navegador. Estos dos objetos están disponibles implícitamente para nosotros.

Para demostrar como manejar formularios HTML usando JSP, aquí tenemos un formulario de ejemplo con dos campos: uno para el nombre y otro para el email. Como podemos ver, el formulario HTML está definido en un fichero fuente JSP. Se utiliza el método `request.getParameter` para recuperar los datos desde el formulario en variables creadas usando etiquetas JSP.

La página `procesar.jsp` imprime un formulario o la información proporcionada por el usuario dependiendo de los valores de los campo del formulario. Si los valores del formulario son `null` se muestra el formulario, si no es así, se mostrará la información proporcionada por el usuario. Observa que el formulario es creado y manejado por el código del mismo fichero JSP.

Ejemplo #2: procesar.jsp

```
<HTML>
   <HEAD>
         <TITLE>Formulario Ejemplo</TITLE>
   </HEAD>

   <BODY BGCOLOR="#ffffcc">
         <% if (request.getParameter("name")==null &&
               request.getParameter("email")== null) { %>
         <CENTER>
         <H2>User Info Request Form</H2>
         <FORM METHOD="GET" ACTION="procesar.jsp">
            <P>Nombre: <input type="text" name="name" size=26>
            <P>email: <input type="text" name="email" size=26>
            <P><input type="submit" value="Process">
         </FORM>
         </CENTER>
         <% } else { %>
         <%! String name, email; %>
         <% name = request.getParameter("name");
               email = request.getParameter("email"); %>
         <P><B>Ha introducido la siguiente información</B>:
         <P><B>Nombre</B>: <%= name %>
         <P><B>Email</B>: <%= email %>
         <% } %>
   </BODY>
</HTML>
```

Comentarios: En el contexto de JSP, los JavaBeans contienen la lógica de negocio que devuelve datos a un script en una página JSP, que a su vez formatea los datos devueltos por el componente JavaBean para su visualización en el navegdor. Una página JSP utiliza un componente JavaBean fijando y obteniendo las propiedades que proporciona.

Hay muchos beneficios en la utilización de JavaBeans para mejorar las páginas JSP:

- Componentes Reutilizables: diferentes aplicaciones pueden reutilizar los mismos componentes.

- Separación de la lógica de negocio de la lógica de presentación: podemos modificar la forma de mostrar los datos sin que afecte a la lógica del negocio.
- Protegemos nuestra propiedad intelectual manteniendo secreto nuestro código fuente.

Uso de JavaBeans con JSP

Ahora, veamos como modificar el ejemplo anterior, `procesar.jsp` para usar JavaBeans. En el formulario anterior había dos campos: `name` y `email`. En JavaBeans, son llamados propiedades. Por eso, primero escribimos un componente JavaBean con métodos `setX getX`, donde X es el nombre de la propiedad. Por ejemplo, si tenemos unos métodos llamados `setName` y `getName` entonces tenemos una propiedad llamada `name`. El ejemplo #3 muestra un componente `FormBean`.

Los buenos componentes deben poder interoperar con otros componentes de diferentes vendedores. Por lo tanto, para conseguir la reutilización del componente, debemos seguir dos reglas importantes (que son impuestas por la arquitectura JavaBeans):

1.1 Nuestra clase bean debe proporcionar un constructor sin argumentos para que pueda ser creado usando `Beans.instantiate`.
1.2 Nuestra clase bean debe soportar persistencia implementando el interface `Serializable` o `Externalizable`.

Ejemplo #3: FormBean.java

```
package userinfo;
import java.io.*;

public class FormBean implements Serializable {
  private String name;
  private String email;
  public FormBean() {
   name = null;
   email = null;
  }
  public void setName(String name) {
   this.name = name;
  }
  public String getName() {
   return name;
  }
  public void setEmail(String email) {
   this.email = email;
  }
  public String getEmail() {
   return email;
  }
}
```

Para poder usar el componente `FormBean` en el fichero JSP, necesitamos ejemplarizar el componente. Esto se hace usando la etiqueta `<jsp:useBean>`. La siguiente línea `<jsp:setProperty>` se ejecuta cuando se ha ejemplarizado el bean, y se usa para inicializar

sus propiedades. En este caso, ambas propiedades (`name` y `email`) se configuran usando una sola sentencia. Otra posible forma de configurar las propiedades es hacerlo una a una, pero primero necesitamos recuperar los datos desde el formulario. Aquí tenemos un ejemplo de como configurar la propiedad `name`:

```
<%! String yourname, youremail; %>
<% yourname = request.getParameter("name"); %>
<jsp:setProperty name="formbean" property="name"
value="<%=yourname%>"/>
```

Una vez que se han inicializado las propiedades con los datos recuperados del formulario, se recuperan los valores de las propiedades usando `<jsp:getProperty>` en la parte `else`, como se ve en el Ejemplo #4:

Ejemplo #4: procesar2.jsp

```
<jsp:useBean id="formbean" class="userinfo.FormBean"/>
<jsp:setProperty name="formbean" property="*"/>
<HTML>
<HEAD>
<TITLE>Form Example</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffcc">
<% if (request.getParameter("name")==null
 && request.getParameter("email") == null) { %>
<CENTER>
<H2>User Info Request Form </H2>
<form method="GET" action="procesar2.jsp">
<P>
Your name: <input type="text" name="name" size=27>
<p>
Your email: <input type="text" name="email" size=27>
<P>
<input type="submit" value="Process">
</FORM>
</CENTER>
<% } else { %>
<P>
<B>You have provided the following info</B>:
<P>
<B>Name</B>: <jsp:getProperty name="formbean" property="name"/>
<P>
<B>Email</B>: <jsp:getProperty name="formbean" property="email"/>
<% } %>
</BODY>
</HTML>
```

JDBC

JDBC es una API para el lenguaje de programación Java que define cómo un cliente puede acceder a una base de datos. Proporciona métodos para consultar y actualizar datos en una base de datos. JDBC está orientado a bases de datos relacionales. Desde un punto de vista técnico, la API es un conjunto de clases en el paquete java.sql. Para usar JDBC con una base de datos en particular, necesitamos un controlador JDBC para esa base de datos.

JDBC es una piedra angular para la programación de bases de datos en Java. Hoy en día, se considera de muy bajo nivel y propenso a errores. Se crearon soluciones como MyBatis o JdbcTemplate para aliviar la carga de la programación JDBC. Sin embargo, bajo el capó, estas soluciones todavía usan JDBC. JDBC es parte de la plataforma Java Standard Edition.

JDBC gestiona estas tres actividades principales de programación:

- conectarse a una base de datos;
- enviar consultas y declaraciones de actualización a la base de datos;
- recuperar y procesar los resultados recibidos de la base de datos en respuesta a la consulta.

Conector MySQL / J

Para conectarse a MySQL en Java, MySQL proporciona MySQL Connector / J, un controlador que implementa la API JDBC. MySQL Connector / J es un controlador JDBC Tipo 4. La designación de Tipo 4 significa que el controlador es una implementación Java pura del protocolo MySQL y no se basa en las bibliotecas cliente de MySQL.

Cadena de conexión

Una conexión de base de datos se define con una cadena de conexión. Contiene información como el tipo de base de datos, el nombre de la base de datos, el nombre del servidor y el número de puerto. También puede contener pares clave / valor adicionales para la configuración. Cada base de datos tiene su propio formato de cadena de conexión.

La siguiente es una sintaxis de una cadena de conexión MySQL:

```
jdbc: mysql: // [host1] [: puerto1] [, [host2] [: puerto2]] ... [/ [base de datos]]
   [? propertyName1 = propertyValue1 [& propertyName2 = propertyValue2] ...]
```

Es posible especificar varios hosts para una configuración de conmutación por error del servidor. Los elementos entre corchetes son opcionales. Si no se especifica ningún host, el nombre de host predeterminado es localhost. Si no se especifica el puerto para un host, el valor predeterminado es 3306, el número de puerto predeterminado para los servidores MySQL.

```
jdbc: mysql: // localhost: 3306 / testdb? useSSL = false
```

Este es un ejemplo de una cadena de conexión MySQL. Se jdbc:mysql://conoce como subprotocolo y es constante para MySQL. Nos conectamos al localhost puerto estándar 3306 de MySQL. El nombre de la base de datos es testdb. Los pares clave / valor adicionales siguen al signo de interrogación (?). El useSSL=false le dice a MySQL que no habrá una conexión segura.

Pasos para conectar JAVA a una base de datos

1- Primero que todo debemos descargar el archivo .jar conocido como MySQL Connector y añadirlo a las librerías de nuestro proyecto, en NetBeans este archivo .jar ya viene incorporado si utilizás la versión 8,2 y anteriores; luego crearemos en nuestro proyecto, un paquete al cual llamaremos "Metodos", y dentro de este, crearemos una clase la cual llamaremos "ConexionMySQL".

2- Después de "public class ConexionMySQL {", deberás incorporar este fragmento de código que se utiliza para importar las Clases y/o librerías necesarias para poder trabajar con ciertas funcionalidades y métodos de nuestra Clase:

```
import java.sql.*;
```

3- El código completo debe quedar asi:

```
package Metodos;
import java.sql.*;
import javax.swing.JOptionPane;


public class ConexionMySQL {

    public String db = "bdclientes";
    public String url = "jdbc:mysql://localhost/"+db;
    public String user = "root";
    public String pass = "";


   public Connection Conectar(){

       Connection link = null;

       try{

            Class.forName("com.mysql.jdbc.Driver");

          link = DriverManager.getConnection(this.url, this.user, this.pass);

       }catch(Exception ex){

          JOptionPane.showMessageDialog(null, ex);
       }

       return link;

   }

}
```

4- La primera parte de este código, consiste en crear 4 variables de tipo String y con la propiedad public. En la primera variable llamada "db", guardamos el nombre de la Base de datos a la cual nos queremos conectar; en la segunda variable url, guardaremos la ruta de donde se encuentra ubicada nuestra Base de datos, si nos damos cuenta deberemos emplear la api "jdbc:mysql" lo cual permite indicar que queremos conectar nuestra aplicación Java con una Base de datos en MySQL, además debemos consignar la dirección de la máquina que contiene la Base de datos, en este caso como trabajaremos en una sola máquina colocamos "localhost", en el caso dado que trabajemos de forma remota debemos reemplazar este valor por la dirección IP (ej. 192.168.1.2) de la máquina a la cual necesitamos conectarnos, en esta misma variable concatenamos la variable que instanciamos al principio "+db"; luego la tercera y cuarta variable "user" y "pass", corresponden al usuario y la clave para poder tener acceso a la Base de datos.

```
public String db = "bdcontactos";
public String url = "jdbc:mysql://localhost/"+db;
public String user = "root";
public String pass = "123";
```

5- Lo siguiente es un método, este contiene la propiedad "public", y devolverá un valor de tipo "Connection", que en otras palabras sería la conexión con nuestra Base de datos; y se le ha asignado a este método el nombre de "Conectar":

```
public Connection Conectar(){}
```

6- Ahora instanciaremos un objeto de tipo "Connection", al cual asignaremos el valor "null". Esto irá dentro del método del punto anterior:

```
Connection link = null;
```

7- Proseguiremos a crear un bloque "try-catch", debido a que el código que hemos de emplear podría producir una excepción y/o error. Este bloque irá dentro del método mencionado en el punto 5:

```
try{

    }catch(Exception ex){


    }
```

8- En esta línea de código lo que haremos será cargar la Clase "Driver", que se encuentra ubicada dentro del ".jar" MySQL Connector que agregamos a nuestras librerías. Esta línea irá dentro del "try{}" del punto anterior:

```
 Class.forName("com.mysql.jdbc.Driver");
```

9- Ahora utilizando el método DriverManager.getConnection, y empleando las variables que instanciamos al inicio de esta Clase, obtendremos la conexión con nuestra Base de datos, y procederemos administrarla y almacenarla en el objeto tipo Connection instanciado al principio del método "Conectar". Esta línea de código irá dentro del "try{}".

```
link = DriverManager.getConnection(this.url, this.user, this.pass);
```

10- En el caso dado de que ocurra un error al tratar de conectarnos con nuestra Base de datos, procederemos a mostrar dicha excepción a través de una pequeña ventana, usando el componente JOptionPane. Esta parte irá dentro del "catch(){}" del punto 7:

```
JOptionPane.showMessageDialog(null, ex);
```

11- Por último retornaremos la conexión obtenida, la cual podría ser exitosa o nula. Esta línea irá por fuera del try-catch, pero dentro del método Conectar:

```
return link;
```

12- En cuanto al código para el test de conexión, en el código, lo que hacemos es instanciar un objeto de nuestra Clase ConexionMySQL.java y luego en un objeto tipo Connection, administramos y/o capturamos la conexión que se nos devuelve al llamar el método Conectar de nuestra Clase mencionada anteriormente:

```
ConexionMySQL mysql = new ConexionMySQL();

java.sql.Connection cn= mysql.Conectar();
```

14- En el resto de código, empleamos un condicional "if", en el cual verificamos que no se nos devuelva una conexión nula, si se cumple esta condición, mostraremos una ventana (JOptionPane) en la cual mostraremos el mensaje "Conectado"; luego dentro de un try-catch, procedemos a cerrar la conexión con la Base de datos, y en dado caso de que ocurra una excepción a la hora de realizar la desconexión, mostraremos dicho error por consola:

```
if(cn!=null){
        JOptionPane.showMessageDialog(null, "Conectado");
        try{
           cn.close();
        }catch(SQLException ex){
           System.out.println("Error al desconectar "+ex);
        }
    }
```

15- Ahora sólo basta con ejecutar tu proyecto y hacer las respectivas pruebas.

Código completo de la prueba de la conexión:

```
ConexionMySQL mysql = new ConexionMySQL();
    java.sql.Connection cn= mysql.Conectar();

    if(cn!=null){
        JOptionPane.showMessageDialog(null, "Conectado");
        try{
            cn.close();
        }catch(SQLException ex){
            System.out.println("Error al desconectar "+ex);
        }
    }
```