4

Java Server Pages

TABLA DE CONTENIDOS

INTRODUCCIÓN

Java Server Pages (JSP) es una tecnología de Sun Microsystems que permite combinar en un mismo fichero de texto código estático HTML, con código Java para construir páginas dinámicas. El código java se separa del HTML encapsulándolo dentro de etiquetas, que generalmente comienzan por '<%' y terminan por '%>'. Normalmente la extensión de este fichero será .jsp.

De acuerdo con la estructura propuesta por las páginas JSP se puede separar la parte estática de la parte dinámica del código. Esto, agiliza y especializa el desarrollo de aplicaciones web, pudiendo desarrollar la parte estática con herramientas tradicionales de diseño por artistas gráficos, creando plantillas HTML u hojas de estilo, mientras que la parte dinámica las implementan los programadores. No es necesario que un programador conozca el trabajo de otro, ya que ambas partes son independientes. No se necesita de un conocimiento profundo de Java para desarrollar páginas dinámicas porque con un conjunto reducido de instrucciones es posible obtener páginas con funcionalidades avanzadas, por ejemplo a través del uso de Java Beans en páginas JSP.

Las páginas JSP están basadas en la tecnología de los servlets. Cuando un usuario solicita una página JSP por primera vez, el contenedor web compila la página traduciéndola a un servlet y redirigiendo la salida a la respuesta estándar del servlet. De este modo, es más fácil y rápido escribir código HTML directamente, frente a los servlets donde se necesitarían multitud de líneas println para tal labor. Las solicitudes posteriores de la misma página JSP no necesitarán recompilarse, mejorando el tiempo de respuesta del servidor de cara al cliente, ya que estás permanecerán en memoria.

Se emplearán servlets para implementar la lógica de la aplicación como gestión de peticiones de servicio y redirección de las mismas, acceso a bases de datos u otros recursos, en contraposición a las páginas JSP cuyo uso orientamos a la presentación de la salida al usuario final.

Veamos un ejemplo de una página web simple que muestra un mensaje de saludo creado con ambas tecnologías y el servlet resultante de la compilación de la página JSP Código del ejemplo implementado como un servlet:

```
package cap10;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloServlet extends HttpServlet
{
public void service(HttpServletRequest request,HttpServletResponse response)
                        throws ServletException,   IOException
{
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<html>");
    pw.println("<head>");
    pw.println("</head>");
    pw.println("<body>");
    pw.println("<h3> Hola " + request.getParameter("userName") + </h3>");
    pw.println("</body>");
    pw.println("</html>");
}
}
```

Código del ejemplo implementado como una página JSP:

```
<html>
<body>
<h3>Hola <%=request.getParameter("userName") %></h3>
</body>
</html>
```

Como podemos ver en el servlet obtenido de la compilación de la página JSP, éste extiende a la clase HttpJspPage, que es la interface que implementa los siguientes 3 métodos:

- JspInit() : Es el equivalente al método init() del ciclo de vida de un servlet. Lo hereda de la interface JspPage a la cuál extiende (opcional).
- JspDestroy(): equivale al método destroy() del ciclo de vida de un servlet. También heredado de JspPage (opcional).
- _jspService(javax.servlet.http.HttpServletRequest request, javax.servlet.http.HttpServletResponse response): Método generado automáticamente cuando el motor de JSP's traduce la página a un servlet. Su funcionalidad es homónima a la del método service() de un servlet tradicional.

Código del servlet obtenido de la compilación de la página JSP:

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;
public class Hello$JSP extends HttpJspBase {
    static {
    }
    public Hello$JSP( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws     org.apache.jasper.runtime.JspException {
    }
public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;ISO-8859-1");
```

```
            pageContext = _jspxFactory.getPageContext(this, request, response,
                                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();

            session = pageContext.getSession();
            out = pageContext.getOut();

            // HTML // begin [file="/Hello.jsp";from=(0,0);to=(2,9)]
                out.write("<html>\r\n<body>\r\n<h3>Hola ");

            // end
            // begin [file="/Hello.jsp";from=(2,12);to=(2,45)]
                out.print(request.getParameter("userName") );
            // end
            // HTML // begin [file="/Hello.jsp";from=(2,47);to=(5,0)]
                out.write("</h3>\r\n</body>\r\n</html>\r\n");

            // end

        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(t);
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
        }
    }
```

USO DE ETIQUETAS EN PÁGINAS JSP.

Para insertar código Java en las páginas JSP se emplean etiquetas. De acuerdo con la funcionalidad del código Java a embeber, emplearemos un tipo de etiqueta u otro.

Distinguimos 4 tipos de etiquetas distintas:

1. Directivas.
2. Declaración
3. Scriptlets
4. Etiquetas de expresión

DIRECTIVAS

Las etiquetas directivas se emplean para proporcionar información al motor de JSP's acerca de cómo construir el servlet; por ejemplo, cómo importar un paquete, declarar una página para reportar errores, especificar librerías de etiquetas…

La sintaxis general de las etiquetas directivas es:

```
<%@ directiva {atributo="valor"} %>
```

Las directivas que podemos utilizar son:

- page: proporciona información acerca de la página JSP para su compilación. De sus atributos el más importante es la cláusula import:
  - import: sirve para especificar los paquetas y clases Java que se quieren utilizar en el servlet resultante de la compilación de la página JSP. Es equivalente al import de java. El código declarado en esta directiva se añadirá a la cabecera del servlet obtenido, antes de la declaración de la clase, al igual que en un fichero java común.

```
<%@ import java.util.* %>
```

Otros atributos posibles son:

- contentType="MIME-Type"
- isThreadSafe="true|false"
- session="true|false"
- buffer="sizekb|none"
- autoflush="true|false"
- extends="package.class"
- info="message"
- errorPage="url"
- isErrorPage="true|false"
- language="java"

- include: incluye un fichero de texto en el momento de la traducción de la página JSP a un servlet. Este fichero puede ser un fichero html o un fichero JSP. Esto posibilita la inclusión de directivas, código HTML, scriptlets… lo que resulta muy útil en la reutilización de código.

Su sintaxis es la siguiente:

```
<%@ include file="url relativa" %>
```

Imaginemos que queremos incluir un mismo banner en la cabecera de cada página que compone nuestra aplicación web; para ello, bastaría con definir dentro de cada página JSP la cláusula include con la dirección relativa del fichero HTML que contiene la imagen del banner. Nuestra página JSP tendría el siguiente aspecto:

```
<HTML>
<HEAD>
    <TITLE> Tutorial de JSP y Servlets </TITLE>
</HEAD>
<BODY>
<%@ include file="banner.html" %>
<H1> Encima de esta línea se encuentra el banner que hemos añadido </H1>
… (Código perteneciente a la página)
```

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

ETIQUETAS DE DECLARACIÓN

Las etiquetas de declaración sirven para declarar variables y métodos de clase. Es similar a la declaración de variables en Java. El código encapsulado entre estas etiquetas se inserta fuera del método _jspService() del servlet generado. Su sintaxis es:

```
<%! Código Java %>
```

Por ejemplo para declarar una variable que almacene el número de visitas a nuestra página JSP, utilizaremos lo siguiente:

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

SCRIPTLETS

Los scriptlets nos permiten insertar código java dentro del método _jspService() del servlet obtenido al traducir la página JSP. Implícitamente un scriptlet tiene acceso a los siguientes objetos sin necesidad de declararlos:

- request: Representa la petición del cliente. Es normalmente una subclase de HttpServletRequest
- response: Representa la respuesta del servidor web al cliente y es una subclase de HttpServletRequest.
- out: objeto que representa la salida de texto en la pantalla del cliente.
- session: representa al objeto HttpSession asociado a la petición de servicio.
- application: se identifica con el objeto ServletContext que se obtiene al llamar a getServletConfig().getServletContext().

La sintaxis de un scriptlet es:

```
<% Código Java %>
```

A continuación veamos un ejemplo del uso de estas etiquetas para obtener el parámetro nombreUsuario pasado en la petición del cliente a la página JSP, para identificar si se trata de un nuevo usuario o no.

```
<HTML>
<BODY>
…
<%-- Esto es un comentario en JSP.  --%>
<% if(request.getParameter("nombreUsuario")=="nuevoUsuario" ){ %>
```

```
            Bienvenido <B> newbie </B> a nuestro portal
        <%} else {%>
            Otra vez por aquí
        …
        </BODY>
        </HTML>
```

El anterior código una vez compilado a un servlet quedaría:

```
…
public void doGet(HttpServletRequest request, HttpServletResponse response) {
        …
if(request.getParameter("nombre")=="nuevoUsuario") {
        out.println("Bienvenido <B> newbie </B> a nuestro portal");
    } else {
    out.println(" Otro vez por aquí");
    ...
    }
```

ETIQUETAS DE EXPRESIÓN

Se emplean para insertar código embebido dentro de la página HTML. Cualquier código situado entre las marcas '<%=' y '%>', que identifican una etiqueta de expresión, se evalúa, se convierte a cadena automáticamente, y se agrega a la salida. En las etiquetas de expresión se pueden utilizar los mismos objetos implícitos que en los scriptlets.

Hay que hacer el inciso de que el código introducido en una expresión no ha de finalizar nunca en ";". Esto es así porque el motor JSP pondrá la expresión automáticamente como parámetro de la instrucción out.println().

Retomando el ejemplo de declaración anterior, ahora incrementaremos en uno el contador de visitas, mostrando el resultado por pantalla:

```
    …
    <%! private int contador=0 %>
    <%= contador++%>
    …
    //Mostremos ahora el valor de la variable i en un bucle:
    …
<% for (int i=0; i<=5;i++) { %>
<%=i%>
<% } %>
…
```

USO DE JAVA BEANS EN PÁGINAS JSP

Un componente Java Bean es una clase Java fácil de reutilizar y manipular. Ésta ha de seguir las siguientes convenciones en el diseño:

- Tiene un constructor sin argumentos.
- Para leer y modificar sus propiedades hay que usar los métodos:
- tipoPropiedad getNombrePropiedad()

- void **setNombrePropiedad**(tipoPropiedad valor)

Las propiedades en un Java Bean son "variables" que podemos clasificar como de:

- sólo lectura.
- sólo escritura.
- lectura/escritura

En función del tipo de propiedad a implementar definiremos los métodos getXxx, setXxx o ambos. El acceso a las propiedades solamente podrá llevarse a cabo a través de estos métodos. Hay que tener en cuenta que el nombre de la propiedad en la declaración del nombre del método ha de comenzar con mayúsculas.

Veamos como ejemplo un Java Bean que gestiona el acceso a una base de datos para recuperar y modificar los productos almacenados en ella, para una aplicación web orientada al comercio electrónico:

```
package dataBase;
public class VentaOnLine {
private String productoId = "0";
//  database es una variable que referencia el objeto que instancia la base de
// datos que vamos a emplear.
private VentaOnLineDBAO database = null;
public VentaOnLine() {
}

  // Método para fijar la propiedad id del producto
  public void setProductoId(String productoId) {
      this.productoId = productoId;
  }
// Método que define la base de datos a la que vamos a acceder.
  public void setDatabase(VentaOnLineDBAO database) {
      this.database = database;
  }
 // Método que recupera las características de un producto
  public ProductoCaracteristicas getProductoDetalles() throws
    ProductoNoEncontradoException {
        return (ProductoCaracteristicas)database.getProductoDetalles(bookId);
  }
}

//  Este método devuelve la lista de productos disponibles en la base de datos
encapsulada en
    //un objeto del tipo Collection.
      public Collection getProductos() throws ProductoNoEncontradoException {
        return database.getListaProductos();
  }

//Método que establece cual es el producto que vamos a comprar por medio de su
id
  public void setProductoComprarId(productoId id)
    throws PedidoException {
        database.comprarProducto(id);
  }
// Método que recupera el valor de la propiedad de solo lectura
numeroDeProductos.
  public int getNumeroDeProductos () throws ProductoNoEncontradoException {
        return database.getNumeroDeProductos();
  }
```

La potencia del uso de los Java Beans en páginas JSP radica en la no necesidad por parte del programador de conocer Java para implementar páginas dinámicas usando JSP's, ya que la mayor parte de la funcionalidad necesaria para nuestra página, como acceso a bases de datos, ficheros, construcción de gráficos dinámicos y estadísticas, recuperar los datos introducidos por un cliente en un formulario HTML… se puede conseguir mediante la reutilización de código en forma de Java Beans.

Para declarar el uso de un Java Bean dentro de una página JSP utilizamos la etiqueta <jsp:useBean>. Tenemos 2 posibles sintaxis:

```
<jsp:useBean id="nombreBean" class="nombreClaseBean" scope="scope"/>
```

o

```
<jsp:useBean id="beanName" class="fully_qualified_classname"
scope="scope">
  <jsp:setProperty .../>
</jsp:useBean>
```

La segunda forma la empleamos cuando queremos inicializar los valores de las propiedades del Java Bean.

Veamos qué significa cada uno de los atributos que acompañan a la cláusula jsp:useBean en su declaración:

- id : declara el nombre del Java Bean que deberemos emplear en el resto de la página JSP para referirnos a él y acceder a su funcionalidad. Se usará un objeto Java Bean anterior en lugar de instanciar uno nuevo si el motor de JSP's encuentra uno con la misma id y scope
- scope : define el alcance y vida del Java Bean en la aplicación web. Su valor se decidirá de acuerdo con los principios de diseño del programador. Puede tomar los siguientes valores:
  - page : indica que el Java Bean sólo estará disponible para la página actual. Éste es el scope por defecto de un Java Bean.
  - request : el Java Bean sólo estará disponible para la petición actual del cliente.
  - session : el Java Bean está disponible para todas las páginas durante el tiempo de vida de la sesión actual.
  - application : indica que el Java Bean estará disponible para todas la páginas que compartan el mismo ServletContext o hasta que se destruya.
  - class : Define el nombre completo del Java Bean, por lo que un Java Bean no puede declararse sino está contenido en un paquete. Para el Java Bean anterior tendríamos que el valor de class sería "dataBase.VentaOnLine"

MANIPULACIÓN DEL VALOR DE LAS PROPIEDADES DE UN JAVA BEAN

La forma estándar de fijar el valor de las propiedades de un Java Bean es a través de la cláusula jsp:setProperty. Podemos llevar a cabo la inicialización en dos contextos distintos; uno fuera de la cláusula jsp:useBean. En este caso se inicializan las propiedades asumiendo que el Java Bean ya existe:

```
<jsp:useBean id="nombreBean"…>
<jsp:setProperty name="nombreBean"
                 property="nombrePropiedad"
                 value="valor"/>
```

En el segundo contexto, se encapsula *jsp:setPropety* dentro de la acción jsp:useBean; sólo se inicializarán las propiedades en caso de que el Java Bean se haya instanciado:

```
<jsp:useBean id…
       <jsp:setProperty name=…/>
</jsp:useBean>
```

Veamos el significado de los atributos que acompañan a la etiqueta:

- name: nombre del Java Bean. Debe de coincidir con el nombre usado en la declaración de éste, en el atributo id.
- property: nombre la propiedad que se desea modificar. Admite la posibilidad de introducir el carácter '*' para denotar a todas aquellas parámetros de la petición de solicitud que coincidan con nombres de propiedades del Java Bean se pasen a los métodos de selección adecuados de éste.
- value: Este atributo es opcional. Especifica el valor de la propiedad, convirtiendo automáticamente el valor de la cadena introducida al tipo de dato de la propiedad. En su lugar podemos utilizar el atributo param, opcional también y con la misma sintaxis, para que la propiedad tome el valor del parámetro de la petición de servicio del mismo nombre, en caso de que exista.

Veamos un ejemplo en el que modificamos el valor de la propiedad productoId del Java Bean que hemos creado anteriormente:

```
…
<jsp:useBean id="miPrimerBean"
              class="dataBase.ventaOnLine"
              scope="session"/>
…
<jsp:setProperty name="miPrimerBean"
                 property="productoId"
                 value="5"/>
```

Igualmente, utilizamos la etiqueta:

```
<jsp:getProperty name="nombreBean" property="nombrePropiedad"/>
```

para recuperar el valor de una determinada propiedad e inserta el resultado en forma de cadena en la respuesta al cliente. Por ejemplo:

```
…
<jsp:getProperty name="miPrimerBean" property="numeroDeProductos"/>
…
```

JAVA BEANS Y FORMULARIOS

Los Java Beans son una herramienta muy potente para extraer y procesar de forma sencilla, los datos introducidos en formularios HTTP que son la piedra angular de Internet para mejorar la interactividad con el cliente. Veremos un ejemplo en el que se pone en práctica todo lo visto. El programa consistirá en un formulario web para introducir los datos del cliente que posteriormente serán extraídos por el JSP mediante el uso de Java Beans:

Código del Formulario HTML:

```
<HTML>
<BODY>
// Declaramos el método POST como paso de parámetros y el JSP DatosCliente.jsp
como el
//responsable de procesarlos.
<FORM METHOD=POST ACTION="DatosCliente.jsp">
What's your name? <INPUT TYPE=TEXT NAME=nombreUsuario SIZE=20><BR>
What's your e-mail address? <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
What's your age? <INPUT TYPE=TEXT NAME=edad SIZE=4>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Implementación del Java Bean que procesará la información descrita en el formulario:

```
package formulario;
public class DatosCliente {
       String nombreUsuario, email;
       int edad;

       public void setNombreUsuario( String valor ){
              nombreUsuario = valor;  }

       public void setEmail( String valor ){
              email = valor;      }

       public void setEdad( int valor )    {
              edad = valor;    }
       public String getNombreUsuario() { return nombreUsuario; }

       public String getEmail() { return email; }

       public int getEdad() { return edad; }
}
```

El siguiente código corresponde al JSP (DatosCliente.jsp) que extraerá a partir del Java Bean la información obtenida en el formulario, mostrándola luego por pantalla mediante una página construida dinámicamente:

```
<jsp:useBean id="usuario" class="Formulario.DatosCliente" scope="session"/>
// Obtenemos todos los parámetros de la petición que coincidan con el nombre
de parámetros
//declarados en el Java Bean.
<jsp:setProperty name="usuario" property="*"/>

<HTML>
<BODY>
    Los datos introducidos por el usuario en el formulario son:<BR>
    Nombre: <%= usuario.getNombreUsuario() %><BR>
    Email: <%= usuario.getEmail() %><BR>
    Edad: <%= usuario.getEdad() %><BR>
</BODY>
</HTML>
```

ETIQUETAS A MEDIDA: CONCEPTOS GENERALES

Las etiquetas estándar vistas hasta ahora simplifican el desarrollo y el mantenimiento de las páginas JSP. Las etiquetas a medida permiten separar mejor la lógica de presentación de la de proceso. Entre las ventajas del uso de etiquetas a medida encontramos:

- Permiten reducir e incluso eliminar el código Java en nuestras páginas JSP (scriptlets y expresiones, redundantes en la mayor parte de los casos). Uno de los principales problemas de mezclar código Java con código HTML, es que éste se vuelve ilegible y se hace muy costoso de mantener y depurar. Así, mediante el uso de etiquetas personalizadas podríamos implementar todo tipo de herramientas que necesitásemos para crear nuestra aplicación web, ya sea acceso a bases de datos, email, redirección de servlets, procesamiento de formularios… Todo esto ya podía ser hecho con el uso de Java Beans, sin embargo obligaba al desarrollador a tener un conocimiento básico de Java, al menos para inicializar las propiedades. Con las etiquetas a medida cualquier parámetro que necesite se puede pasar como atributo de la etiqueta o en el cuerpo de la misma, ya sean estos estáticos o dinámicos, sin necesidad de incluir código Java para ello. El código Java queda encapsulado en ficheros, al cuál hacen referencia las etiquetas.
- Su sintaxis es menos compleja al presentar un estilo similar al código HTML.
- Permiten su reutilización en otras aplicaciones mejorando la productividad en el desarrollo. Con scriptlets la reutilización de código sólo se podría hacer mediante la difundida y no demasiado buena técnica de cortar-pegar.
- Al igual que los scriptlets y las expresiones JSP, tienen acceso directo a todos los objetos disponibles en las páginas JSP, como request, response, out… Su integración con el entorno es total.

ELEMENTOS DE UNA ETIQUETA A MEDIDA

- Tag Handler: Es el corazón de una etiqueta a medida, es una clase java que implementa una serie de interfaces especializadas que trataremos más adelante. La clase debe implementar toda la funcionalidad referente a la etiqueta. Tiene

acceso a toda la información disponible en la página JSP (request, session,…). Una vez finalizada la ejecución de la clase, esta devuelve su salida a la página JSP.

- Tag Library Descriptor (fichero .tld): Es un fichero XML donde se definen las propiedades, atributos y localización del fichero Tag Handler. Esta información se utiliza en tiempo de ejecución para comprobar el uso correcto de la etiqueta a medida por parte del programador.
- Una declaración de la librería de Tags en la página JSP: Mediante una directiva declaramos la existencia de nuestros tags dentro de la página JSP. De esta forma ya podemos utilizar libremente nuestros tags en la página.
- El fichero web.xml: En este fichero definiremos la asociación entre la declaración que hace el usuario de la librería y el fichero tld correspondiente. Esta asociación se explica más adelante.

TIPOS DE ETIQUETAS A MEDIDA

Distinguiremos los siguientes tipos de etiquetas a medida:

- parametrizados con cuerpo

  <prefijo:nombre atributo=…>

  cuerpo

  </prefijo:nombre>

- parametrizados

  <prefijo:nombre atributo=…/>

En ambos casos, pasamos uno o más atributos como parámetros en la llamada de la etiqueta. Al igual que los parámetros que se pasan en la llamada a un método o función en programación, los atributos nos permiten pasar la información necesaria para el procesamiento del cuerpo de la etiqueta. En el siguiente capítulo profundizaremos más en éste y otros detalles.

DESARROLLO DE ETIQUETAS A MEDIDA SENCILLAS

En este capítulo veremos un ejemplo sencillo de una etiqueta a medida parametrizada y con cuerpo. Implementaremos la etiqueta IfTag que tiene como parámetro *condition*. Ésta tiene un comportamiento similar al de una instrucción *if* en programación si se omite su cláusula *else*. Si el valor de su parámetro *condition* es *true* entonces se procesa el cuerpo de la etiqueta, en caso contrario no se procesa.

Veamos la implementación de los distintos tipos de elementos que intervienen en el desarrollo de una etiqueta a medida para este caso en particular.

- Tag Handler: A continuación se detalla el contenido de la clase Java que contiene la codificación de la etiqueta IfTag:

```
package sampleLib;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
```

```
public class IfTag implements Tag {

    private PageContext pageContext;
    private Tag parentTag;

    public void setPageContext(PageContext pageContext) {
           this.pageContext = pageContext;
    }
    public void setParent(Tag parentTag) {
           this.parentTag = parentTag;
    }

    public Tag getParent() {
           return this.parentTag;
    }
    private boolean condition = false;

    public void setCondition(boolean condition) {
           this.condition = condition;
    }

    public int doStartTag() throws JspException {
           if (condition)
                  return EVAL_BODY_INCLUDE;
           else
                  return SKIP_BODY;
    }

    public int doEndTag() throws JspException {
           return EVAL_PAGE;
    }

    public void release() {
    }
}
```

La clase IfTag que implementa nuestra etiqueta, extiende la interface **Tag** (*javax.servlet.jsp.tagext.Tag*) que define los métodos necesarios para codificar una etiqueta a medida y se pueda utilizar en una página JSP. Veamos uno a uno sus métodos y la finalidad de estos:

- public void setPageContext(PageContext pageContext) : método que define el contexto en el que actuará la etiqueta. A través de PageContext es posible acceder a la misma información que está disponible desde la página JSP, es decir, a HtttpServletRequest, HttpSession… La página JSP llama antes a este método que a doStartTag()
- public void setParent(Tag parentTag) : Establece cual es el manejador de la etiqueta padre. Es nulo cuando la etiqueta no está anidada dentro de otra, como es el caso de nuestro ejemplo, y distinto de nulo en caso contrario. El motor lo llama, al igual que en el anterior caso, antes de llamar a doStartTag().
- public Tag getParent(): Devuelve una instancia de la etiqueta padre.
- public void setCondition(boolean condition) : Método que emplea el motor de JSP para establecer el valor de los parámetros de nuestra etiqueta. Sigue las mismas convenciones que el método setParámetro() de los Java Beans. En el ejemplo actual, la llamada se producirá siempre que aparezca la etiqueta ifTag en la página JSP, ya que el atributo condition es obligatorio.
- public int doStartTag() throws JspException : Este método lo ejecuta el motor de JSP para procesar la etiqueta de apertura de nuestro Tag. De acuerdo con nuestro ejemplo, si la condición es cierta le indicamos a la página JSP que evalúe el cuerpo

de la etiqueta mediante la constante predefinida en la interface Tag EVAL_BODY_INCLUDE. Si es falso devolvemos a la página JSP SKIP_BODY para indicarle que no procese el cuerpo y pase a procesar la etiqueta de cierre de ifTag. En caso de error lanzamos una excepción del tipo JspException para comunicar al servlet tal hecho y que no siga procesándola.

- public int doEndTag() throws JspException : Método que ejecuta el motor JSP tras llamar a doStartTag y ejecutar el cuerpo si la condición fuese cierta. Devolvemos EVAL_PAGE para indicar al motor de JSP's que continúe con el procesamiento del resto de la página. Si por cualquier motivo quisiéramos dejar de evaluar el resto de la página, el resultado que tendríamos que devolver sería SKIP_PAGE..
- public void release(): Lo invoca el motor de JSP con fines de tareas de limpieza. Si por ejemplo hubiésemos definido una etiqueta para realizar una conexión a una base de datos, podríamos definir aquí el cierre de dicha conexión o la liberación de los recursos que hayamos empleado en la implementación de ésta.

En la mayoría de los casos, no necesitaremos definir muchos de estos métodos, ya que lo mejor es extender la clase TagSupport que implementa la interface Tag.

TAG LIBARY DESCRIPTOR

Veamos el contenido del fichero descriptor de nuestra librería de etiquetas (fichero taglib.tld) que define los parámetros, atributos y ruta de nuestra etiqueta.

```
<!DOCTYPE taglib
              PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
                     "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">

         <taglib>
    <!-- INFO ABOUT THIS LIBRARY -->
           <tlib-version>1.0</tlib-version>
           <JSP-version>1.2</JSP-version>
           <short-name>curso</short-name>

    <!-- URI FOR IMPLICIT MAPPING  -->
           <uri>http://localhost/libreria</uri>            …

    <!-- IF TAG -->
    <tag>
           <name>if</name>
           <tag-class>sampleLib.IfTag</tag-class>
           <body-content>JSP</body-content>
           <attribute>
                  <name>condition</name>
                  <required>true</required>
                  <rtexprvalue>true</rtexprvalue>
           </attribute>
    </tag>

</taglib>
```

Lo primero que vemos <!DOCTYPE…> define el tipo de documento e información relevante de éste. A continuación la descripción de la librería se encapsularía entre las etiquetas cerradas <taglib> y </taglib>. Veamos el significado de las etiquetas que componen el cuerpo de la librería:

- <tlib-version>: Señala la versión actual de librería.

- <JSP-version>: Indica al contenedor alojado en el servidor web la versión de JSP's a utilizar para poder emplear las etiquetas que en la librería se definen.
- <short-name>: nombre nemotécnico asignado a la librería por el programador. Éste será el nombre emplearemos en el JSP para hacer referencia a la librería
- <uri>: dirección relativa que identifica unívocamente a la librería.
- <tag>: Define en su cuerpo las propiedades de la etiqueta en cuestión. Sus atributos son:
  - <name>: Declara un nombre único para nuestra etiqueta. Así para poder referirnos a ella en la página JSP, emplearemos la siguiente sintaxis:

<prefijo:nombreEtíqueta>

-
- <tagClass> : Nombre de la ruta completa de la clase que implementa la etiqueta. En nuestro caso, sampleLib.IfTag es la clase que hemos visto antes.
- <body-content> : Específica el tipo del contenido del cuerpo. Los posibles valores son empty para etiquetas sin cuerpo y JSP para aquellas con cuerpo.
- <attribute> : Encapsula en su interior los campos que definirán las propiedades del atributo.

  - <name>: nombre del parámetro.
  - <required> : determina si el parámetro es obligatorio u opcional.
  - <rtexprvalue> : establece si el valor del atributo puede ser calculado dinámicamente.

- Fichero web.xml

En él especificamos la localización de nuestra librería

```
<webapp>
    …

<taglib>
  <taglib-uri>http://localhost/libreria</taglib-uri>
  <taglib-location>taglib.tld</taglib-location>
</taglib>
…
</webapp>
```

El significado de los campos que declaran el uso de la librería son:

- <taglib> : etiqueta que señala al motor de JSP's el uso de una nueva librería.
  - <taglib-uri> : dirección relativa de la librería de etiquetas.
  - <taglib-location> : determina el nombre del fichero .tld que almacena la librería. Este nombre es relativo al directorio WEB-INF. Para que la dirección fuese absoluta respecto a la raíz de la aplicación debería haber comenzado por el carácter '/'.

- Página JSP

Una vez que 1) hemos implementado la etiqueta ifTag, 2) incluido en un librería de etiquetas (taglib.tld), y 3) declarado su existencia al servidor web mediante la modificación del fichero web.xml, lo único que nos quedaría sería codificar una página JSP para su uso. Para tal objetivo, debemos incluir una directiva en la página que indique al motor de JSP's que la página actual va a hacer uso de la etiqueta: <%@taglib prefix="*nombrePrefijo*" uri="*rutaLibrería*">

```
<%@ taglib prefix="test" uri="http://localhost/libreria" %>
<html><body><pre>
Hello<br>
<% boolean debug = "true".equals(request.getParameter("debug")); %>
<test:if condition="<%=debug%>" >
    DEBUG INFO:...
</test:if>
</pre></body></html>
```

El atributo de la directiva *prefix* establece el prefijo que emplearemos por convenio a la hora de invocar a la etiqueta, mientras que *uri* señala la dirección de la librería a la que pertenece. Esta dirección ha de coincidir con la declarada en el fichero web.xml.

El JSP toma el valor del parámetro debug pasado en la petición, por medio de un scriptlet (<% boolean … %>, que será asignado a *condition* en la llamada a la etiqueta usando una expresión JSP. Para finalizar, bastaría invocar a nuestra página JSP para probar el correcto funcionamiento de la etiqueta:

http://localhost:8080/*contexto_aplicacion*/htmlTags/ifTest.jsp?debug=true

DISEÑO DE APLICACIONES WEB SEGÚN EL MODELO MVC

El modelo MVC (Model-View-Controller) se introdujo por primera vez como parte del lenguaje de programación Smalltalk. El objetivo del diseño de programas siguiendo este patrón era agilizar y facilitar el desarrollo de aplicaciones, encapsulando la funcionalidad de éstas en componentes o entidades independientes. De este modo, en caso de error o actualización del software solamente es necesario modificar los componentes implicados sin que estos cambios repercutan en el resto. Además plantea una clara separación de los roles y responsabilidades en el equipo de desarrollo entre programadores y diseñadores gráficos.

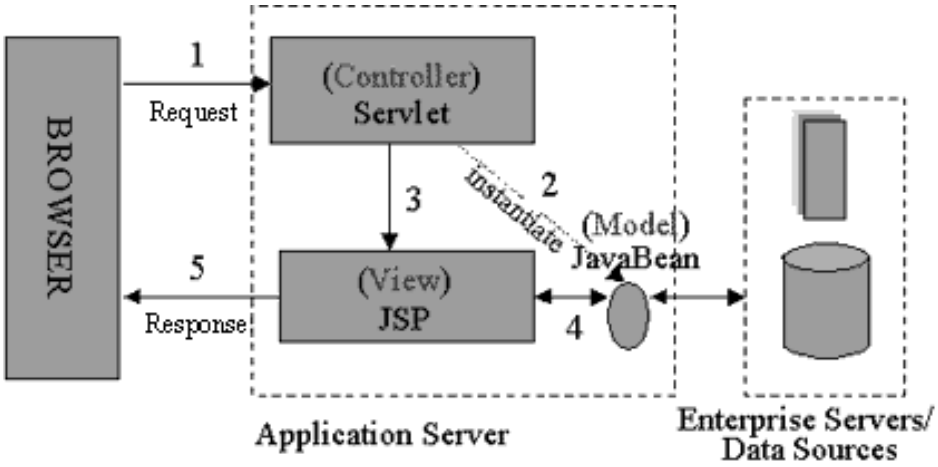
Este modelo fue dirigido al desarrollo de aplicaciones GUI, lo cual se adapta también al diseño de aplicaciones web, ya que ambos comparten una filosofía similar, una interfaz gráfica para poder comunicarse con el usuario.

Dentro de este modelo distinguiremos 3 tipos de elementos o componentes: el modelo, la vista y el controlador.

- El modelo: Es el motor de nuestra aplicación, manipula los datos del programa, sus transformaciones y contiene la lógica de la aplicación. No tiene conocimiento específico de los controladores o de las vistas. La comunicación entre estos tiene lugar de forma dinámica.
- La vista: Es la representación visual del modelo. La vista se registra con el modelo para recibir los eventos asociados al cambio de datos. Cuando se producen estos cambios, presenta al usuario los datosmodificados. Otra responsabilidad de la vista es mandar al controlador las acciones del usuario.

- El controlador: gestiona la interacción entre el usuario y el modelo. El controlador recibe las acciones del usuario e invoca el método adecuado del controlador. También es el encargado de seleccionar la vista.

Veamos ahora mediante un diagrama como se ajustan estas 3 entidades al desarrollo de una aplicación web, donde intervendrá todo lo estudiado hasta ahora:



- El modelo es implementado como un JavaBean para la encapsulación del procesamiento de la información, por ejemplo, para la manipulación de una base de datos.
- Para las vistas emplearemos páginas Jsp que presentarán al usuario los datos por pantalla obtenidos de un JavaBean y nos permitirán interactuar con el usuario. La finalidad de cualquier código java contenido en esta parte del modelo debería orientarse al paso de parámetros al controlador y en obtener datos del modelo.
- El controlador será el responsable de gestionar las peticiones de usuario, el encaminamiento de estas, tratamiento de errores así como el control del flujo de la aplicación. También podríamos incluir aquí algo de código destinado al pre-procesamiento de la información si fuera necesario. El controlador se codificará como un servlet. Este no debe de contener ninguna sentencia del tipo out.println() para mostrar información por pantalla, ya que se violarían los principios de diseño del modelo.
  Las peticiones del controlador podrían provenir tanto de un navegador como de una aplicación java con RMI (invocación remota de métodos) que actuaría como cliente. El comportamiento del servlet se dirige bien por el paso de parámetros en la petición del cliente o bien por el pathinfo de ésta. A partir de está información el servlet elegirá el bean adecuado para la petición del usuario y la vista a emplear para devolver esa información al usuario.

Para comprender mejor como emplear el modelo MVC en el desarrollo de aplicaciones web, implementaremos una guia telefónica. A partir del nombre y los apellidos introducidos por el cliente en un formulario devolveremos el teléfono y la dirección de email de aquellas personas que coincidan con los datos introducidos. Necesitaremos crear una base de datos con una tabla llamada "GuiaTelefonica" que contenga los campos

nombre, apellidos, teléfono y dirección de email. La página principal de nuestra aplicación será la vista BuscarTelefono.jsp que se detalla a continuación:

```
<html>
  <head>
    <title>Guia telefónica</title>
  </head>
  <body bgcolor="#FFFFFF">
   <p><b>  Guia Telefonica CLE Online</b></p>
   <form name="form1" method="get"
     action="GuiaTelefonica.BuscarTelefonoServlet">
   <table border="0" cellspacing="0" cellpadding="6"> <tr> <td >Buscar por:
</td>
   <td>
   </td> </tr> <tr> <td><b>
   Nombre
   </b></td> <td>
   <input type="text" name="Nombre"> <tr> Y/O </td> </tr> <tr> <td ><b>
   Apellidos
   </b></td> <td >
   <input type="text" name="Apellidos"></td> </tr> <tr> <td ></td> <td >
   <input type="submit" name="Submit" value="Enviar">
   </td> </tr> </table>
   </form>
  </body>
</html>
```

La anterior vista llama al controlador de nuestra aplicación (GuiaTelefonia.java) cuando el usuario hace un submit del formulario. El controlador será el responsable de obtener el nombre y los apellidos introducidos por el usuario comprobando que sean correctos, construirá la sentencia SQL y establecerá la conexión con la base de datos (pre-procesamiento de datos), cuyo resultado pasará al JavaBean GuiaTelefonicaBean para que éste extraiga de la base de datos la información requerida.

```
package GuiaTelefonica;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;
import java.net.*;

public class BuscarTelefonoServlet extends HttpServlet {
   public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {

      String query = null;
      String where = null;
      String nombre = null;
      String apellidos = null;
      ResultSet rs = null;

      res.setContentType("text/html");
      PrintWriter out = res.getWriter();

      // Comprobamos que los campos introducidos en el formulario no sean
      //vacíos
      if (req.getParameter("Nombre").length() > 0)
         nombre = req.getParameter("Nombre");
```

```
      else nombre = null;

      if (req.getParameter("Apellidos").length() > 0)
         apellidos = req.getParameter("Apellidos");
      else apellidos = null;

// Contruimos la sentencia SQL que pasaremos al Bean para que este ejecute la
//consulta sobre la base de datos.
      if ((nombre != null) && (apellidos != null)){
         where = "nombre ='";
         where += nombre;
         where += "' AND ";
         where += "apellidos ='";
         where += apellidos;
         where += "'";
      }
      else if ((nombre == null) && (apellidos != null)){
         where = "apellidos ='";
         where += apellidos;
         where += "'";
      }
      else if ((nombre != null) && (nombre == null)){
         where = "nombre ='";
         where += nombre; where += "'";
      }
      query = "SELECT nombre,apellidos,telefono,emilio
      FROM GuiaTelefonica WHERE " + where;
      ConnectDB ConexionBD= new ConnectDB();

      ConexionBD.setConexion("sun.jdbc.odbc.JdbcOdbcDriver",
      "jdbc:odbc:GuiaTelefonica","anonimo","");

      GuiaTelefonicaBean consultaBD=new ConsultaTelefonicaBean();
      ResultSet rs=ConsultaBD.getResultSet(query,ConexionBD);
      //Almacenamos el resultado en la sesion para poder recuperarlo
//posteriormente desde la vista.
  HttpSession sesion=request.getSession(true);
  sesion.setAttribute("resultadoBusqueda",rs);
      RequestDispatcher rd = null;
      rd = getServletConfig().getServletContext().getRequestDispatcher
       ("/mostrarResultados.jsp");
      // Redirigimos la respuesta a la vista mostrarResultados.jsp
      if (rd != null) {
         rd.forward(req, res);
      }

   }
}
```

A continuación tenemos el código del único modelo de nuestra aplicación. Nuestro JavaBean hace uso de la clase ConnectDB para establecer la conexión con la base de datos. Esta última es una clase reutilizable que podemos emplear en cualquier proyecto que necesite del acceso a bases de datos.

```
package GuiaTelefonica;
import java.io.*;
import java.net.*;
import java.sql.*;
import java.util.*;

public class GuiaTelefonicaBean {
    private Connection con = null;
    private Statement stmt = null;
    private ResultSet rs = null;
    private String query = null;

    public GuiaTelefonicaBean() {}

    public ResultSet getResultSet(String query,ConnectDB conexionBD) {
    // Establecemos la conexión con la base de datos
      con = conexionBD.getConnection();
      try{
         stmt = con.createStatement();
         // Ejecutamos la consulta para obtener el result set
         rs = stmt.executeQuery(query);
      }
      catch(SQLException sqlex){
      sqlex.printStackTrace();
    }
    catch (RuntimeException rex) {
    rex.printStackTrace();
   }
   catch (Exception ex) {
   ex.printStackTrace(); }

   return rs;
  }
}
```

Clase ConnectDB:

```
package GuiaTelefonica;
import java.io.*;
import java.net.*;
import java.sql.*;
import java.util.*;

/**
* Clase reutilizable para establecer una connexion con una base de datos.
*/
public class ConnectDB {
   // setup connection values to the database
   private String DB_DRIVER;
   private String URL;
   private String USERNAME;
   private String PASSWORD;

   public void setConnection(String db_Driver, String url, String username,
String password) {
     //Inicializamos los datos de configuracion de la conexion
     DB_DRIVER=db_Driver;
     URL=url;
     USERNAME=username;
     PASSWORD=password;
```

```
     //Inicializamos la conexion con la base de datos.
     try {
        Class.forName(DB_DRIVER).newInstance();
     }
     catch (ClassNotFoundException cnfx) {
        cnfx.printStackTrace();
     }
     catch (IllegalAccessException iaex){
        iaex.printStackTrace();
     }
     catch(InstantiationException iex){
      iex.printStackTrace ();
   }
   }

/**
* Devuelve una connexion a la base de datos.
*/
public Connection getConnection() {
   Connection con = null;
   try {
      con = DriverManager.getConnection(URL, USERNAME, PASSWORD);
   }
   catch (Exception e) {
      e.printStackTrace();
   }
   finally {
      return con;

   }
}

/**
* Método estático que libera la connexion a la base de datos.
*/
public void closeConnection(Connection con) {
   try {
      if (con != null) con.close();
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
}
```

Por ultimo necesitamos crear la vista mostrarResultados.jsp que será llamada por nuestro controlador para presentar los datos solicitados por el cliente:

```
<html>
  <%@page import ="java.sql.*" %>
  <head>
     <title>Guia Telefonica -- Resultados de la busqueda:</title>
     <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  </head>
  <body bgcolor="#FFFFFF"> <b>Search Results</b> <p>
  <% ResultSet rs = (ResultSet)session.getAttribute("resultadoBusqueda");
  %>
  <% if (rs.wasNull()) { %>
     "No existe ningun teléfono de un usuario que coincide con ese nombre en
      nuestra base de datos"
  <% } else %>
  <table> <tr> <td> <div align="center">Nombre</b></div> </td> <td> <div
    align="center">Apellidos</font></b></div> </td> <td> <div
    align="center">Telefono</font></b></div> </td> <td> <div
    align="center">Direccion de email</font></b></div> </td> </tr>
    <% while(rs.next()) { %> <tr> <td>
    <%= rs.getString("nombre") %></td> <td><%= rs.getString("apellidos")
    %></td>
    <td><%= rs.getString("telefono") %>
    </td> <td>
    <%= rs.getString("emilio") %>
    </td> </tr>
    <% } %>
  </table>
  </body>
</html>
```