

Java Server Pages (JSP)

Contenido

1. Primer vistazo a la tecnología JSP

- Qué es y para qué sirve JSP
- Primer ejemplo *Hola Mundo y la fecha*
- Uso de objetos implícitos y ejemplo

-> *Ejercicio: instalación y ejecución de la primera aplicación web con tecnología JSP*

1. Elementos básicos de una página JSP

- **Directivas** <%@ (page | include | taglib)
- **Código Java** <% (declaraciones | java | expresiones)

-> *Ejercicios de manejo de ejemplos simples*

- **Acciones estándar** <jsp:acción (include | forward | usebean | getProperty | setProperty)

-> *Ejercicios de manejo de JavaBeans*

1. Elementos Avanzados de una página JSP

- **Acciones personalizadas** (etiquetas)
 - Definición de la estructura (biblioteca.tld y ésta en web.xml)
 - **Nombre, clase controladora**, atributos, cuerpo, etc.
 - Definición de la funcionalidad (JavaBean que deriva de taglib)
 - Declaración en página con <%@taglib y uso posterior

-> *Ejercicios de manejo de Etiquetas personalizadas*

Introducción a JSP

- JSP es una especificación de Sun Microsystems
 - Sirve para crear y gestionar **páginas web dinámicas**
 - Permite mezclar en una página código HTML para generar la parte estática, con contenido dinámico generado a partir de marcas especiales <% %>
 - El contenido dinámico se obtiene, en esencia, gracias a la posibilidad de incrustar dentro de la página código Java de diferentes formas → **Hay 3 formas de añadir contenido dinámico**
 - Su objetivo final es separar la interfaz (presentación visual) de la implementación (lógica de ejecución)
-

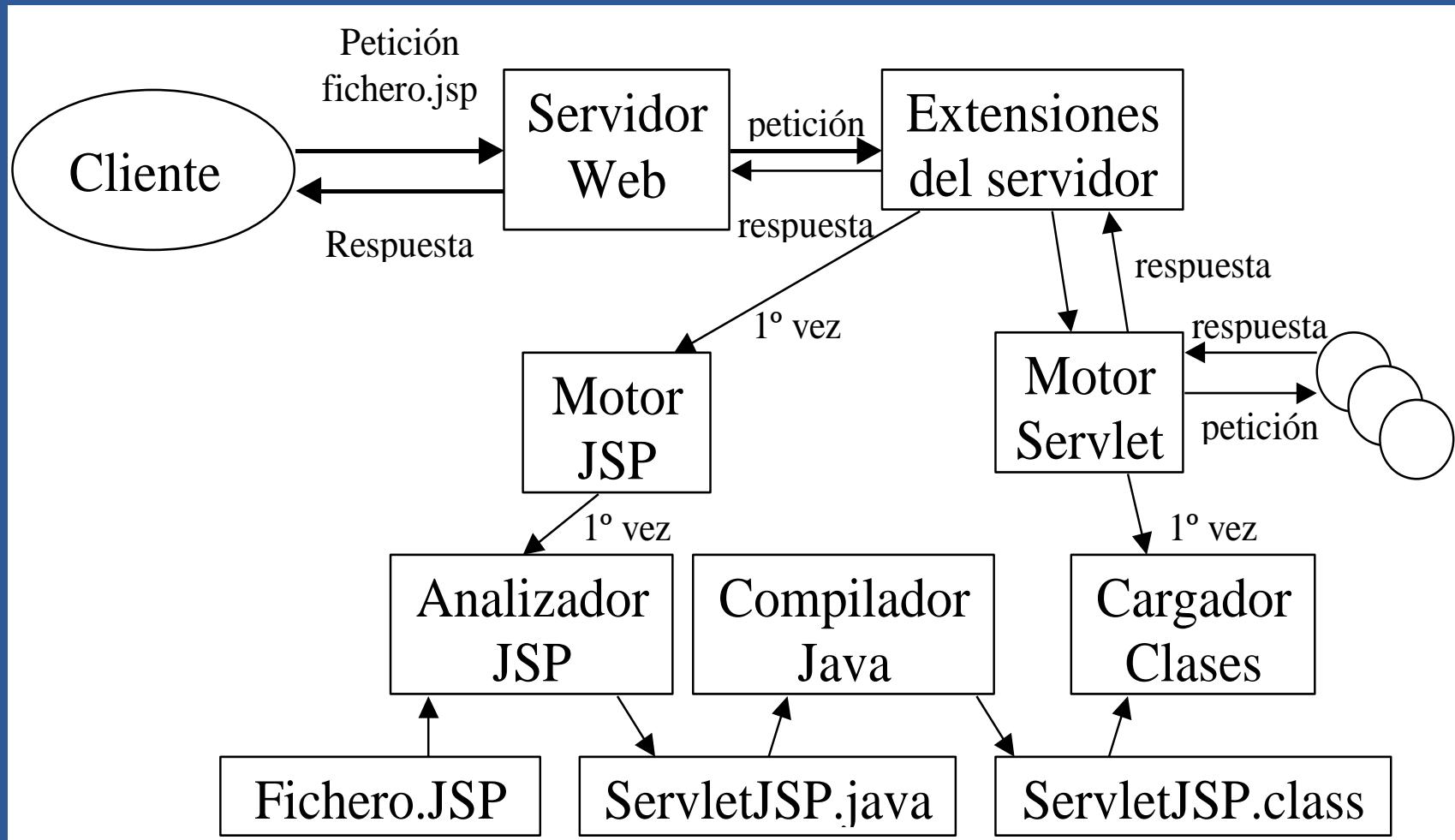
Introducción a JSP

Página JSP → Servlet

- La página JSP se convierte en un servlet
- La conversión la realiza en la máquina servidora el *motor o contenedor JSP*, la primera vez que se solicita la página JSP
- Este servlet generado procesa cualquier petición para esa página JSP
- Si se modifica el código de la página JSP, entonces se regenera y recompila



Funcionamiento



Primer ejemplo de JSP

Ejemplo de página que dice Hola y escribe la fecha actual (fichero ej1_hola.jsp)

```
<%@ page info="Un Hola Mundo" import="java.util.Date" %>
<HTML>
<head> <title> Hola, Mundo </title> </head>
<body> <h1> ¡Hola, Mundo! </h1>
La fecha de hoy es:    <%= new Date().toString() %>
</body>
</HTML>
```

- En esta página se mezcla código HTML con código Java incrustado con unas marcas especiales
- En este caso es una expresión, que se sustituye en la página por el resultado de evaluarla
- En otros casos es un trozo de código Java que simplemente se ejecuta

Ciclo de vida del servlet generado

Cuando se llama por primera vez al fichero JSP, se genera un servlet con las siguientes operaciones

- jsplninit()
 - Inicializa el servlet generado
 - Sólo se llama en la primera petición
- jspService(petición,respuesta)
 - Maneja las peticiones. Se invoca en cada petición, incluso en la primera
- jspDestroy()
 - Invocada por el motor para eliminar el servlet
-

Servlet generado por un JSP

En el método `_jspService` se introduce automáticamente el contenido dinámico de la página JSP.

- El código html se transforma en una llamada al objeto out donde se vuelca el contenido En el ejemplo:

```
out.write("\r\n\r\n<HTML>\r\n<head> <title>\n  Hola, Mundo </title> </head>\r\n<body> <h1>\n    ¡Hola, Mundo! </h1> \r\nLa fecha de hoy es: ");
```

- El código dinámico se traduce en función del contenido

Ej: El código jsp `<%= new Date().toString() %>` se traduce por `out.print(new Date().toString());`

Servlet generado por un JSP

En la primera invocación de esta página se genera automáticamente el siguiente servlet:

```
public class ej1_0005fhola$jsp extends HttpJspBase {  
....  
public final void _jspx_init() throws JasperException {  
...  
public void jspService(HttpServletRequest request,  
HttpServletResponse response)  
throws 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;
```

Servlet generado por un JSP

```
try {  
    if (_jspx_initied == false) {  
        _jspx_init();  
        _jspx_initied = true;  
    }  
    _jspxFactory = JspFactory.getDefaultFactory();  
  
    response.setContentType("text/HTML;charset=ISO-8859-1");  
    pageContext =  
        jjspxFactory.getPageContext(this, request, response,"",  
        true, 8192, true);  
  
    application = pageContext.getServletContext();  
    config = pageContext.getServletConfig();  
    session = pageContext.getSession();  
    out = pageContext.getOut();  
}
```

Servlet generado por un JSP

```
// HTML
// begin [file="C:\\web\\jakarta-
tomcat3.2.2\\webapps\\mariam\\hola.jsp";from=(0,40);to=(6,20)]
    out.write("\r\n\r\n<HTML>\r\n<head> <title> Hola, Mundo
</title> </head>\r\n\r\n<body> <h1> ¡Hola, Mundo! </h1>
\r\nLa fecha de hoy es: ");
// end
// begin [file="C:\\web\\jakarta-
tomcat3.2.2\\webapps\\mariam\\hola.jsp";from=(6,23);to=(6,46)]
    out.print( new Date().toString() );
// end
// HTML // begin [file="C:\\web\\jakarta-
tomcat3.2.2\\webapps\\mariam\\hola.jsp";from=(6,48);to=(11,0)]
    out.write("\r\n\r\n</body>\r\n</HTML>\r\n\r\n");
// end
```

Objetos implícitos

- JSP utiliza los objetos implícitos, basados en la API de servlets.
- Estos objetos están disponibles para su uso en páginas JSP y son los siguientes:
 - **Objeto request**
 - Representa la petición lanzada en la invocación de service(). Proporciona entre otras cosas los parámetros recibidos del cliente, el tipo de petición (GET/POST)
 - **Objeto response**
 - Instancia de HttpServletResponse que representa la respuesta del servidor a la petición. Ámbito de página

Objetos implícitos

- **out**: Es el PrintWriter usado para enviar la salida al cliente. 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**: Este es el objeto HttpSession asociado con la petición. Las sesiones se crean automáticamente, salvo que se use 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.

Objetos implícitos

- **application**: El ServletContext obtenido mediante getServletConfig().getContext().
- **config**: El objeto ServletConfig.
- **pageContext**: 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".
- **page**: 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.

Ámbitos

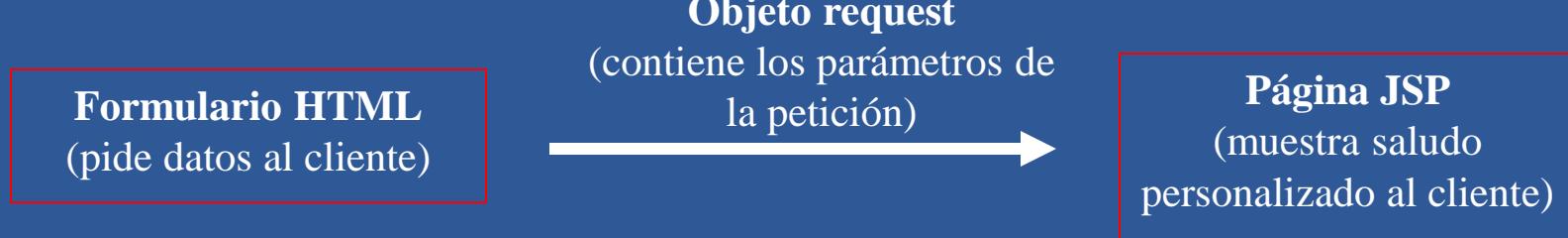
- Define dónde y durante cuánto tiempo están accesibles los objetos (Objetos implícitos, JavaBeans, etc)

Tipos de ámbitos:

- de página. El objeto es accesible por el servlet que representa la página
- de petición
- de sesión. El objeto es accesible durante toda la sesión, desde los servlets a los que se accede
- de aplicación. El objeto es accesible por el servlet que representa la página

Ejemplo de objetos implícitos

Aplicación que pide el nombre al usuario y le devuelve un saludo . Utiliza un fichero HTML como formulario que pide los datos al cliente y se los pasa a una página JSP que muestra el saludo con estos datos. El paso de los datos del formulario al JSP se realiza a través de un objeto implícito: objeto request



Ejemplo de objetos implícitos

Fichero HTML que pide los datos al cliente
(ej2_saludo.html)

```
<HTML>
  <head>
    <title> Formulario de petición de nombre </title>
  </head>
  <body>
    <h1> Formulario de petición de nombre </h1>
    <!-- Se envía el formulario al JSP "saludo.jsp" -->
    <form method="post" action="saludo.jsp" >
      <p> Por favor, introduce tu nombre:
          <input type="text" name="nombre">
      </p>
      <p>
        <input type="submit" value="enviar información">
      </p>
    </form> </body>
</HTML>
```

Ejemplo de objetos implícitos

Fichero JSP que opera dinámicamente con los datos del cliente y muestra los resultados (ej2_saludo.jsp)

```
<HTML>
<head><title> Saludo al cliente </title></head>
<body>
    <h1> Saludo al cliente</h1>
    <!-- Los parámetros que le pasa el cliente en la
        petición se obtienen del objeto implícito request -->
    <%
        String nombre = request.getParameter("nombre");
        out.println("Encantado de conocerle, " + nombre);
    %>
    <!-- Al evaluarse el código, hay que escribir
        explícitamente en la salida (obj. implícito out) -->
</body>
</HTML>
```

Elementos de una página JSP

- **Código HTML**

Además de código HTML la página JSP puede incluir marcadores que se agrupan en tres categorías:

- **Directivas.**
 - Afectan a toda la estructura del servlet generado
- **Elementos de Scripting** (guiones)
 - Permiten insertar código Java en la página JSP
- **Acciones**
 - Afectan al comportamiento en tiempo de ejecución del JSP

Directivas JSP

- Utilizadas para definir y manipular una serie de atributos dependientes de la página que afectan a todo el JSP.
- Las directivas existentes son las siguientes:
 - Page
 - Include
 - Taglib

Directivas de JSP - Directiva Page

Sintaxis

```
<%@ page ATRIBUTOS %>
```

- Donde ATRIBUTOS son parejas:
 nombre="valor"
- Ejemplo:

```
<%@ page language="Java" import="java.rmi.* , java.util.*"  
session="true" buffer="12kb" %>
```

- Existe una lista de atributos que pueden ser usados
-

Directivas de JSP - Directiva Page

Algunos de los atributos más usados:

- `import = "package.class"`.
 - Lista de paquetes o clases, separados por comas, que serán importados para utilizarse dentro del código java.
- `session = "true | false"`
 - Especifica si la página participa en una sesión HTTP. Si se inicializa a true, está disponible el objeto implícito sesión.
- `buffer = "tamañoKB"`
 - Especifica el tamaño de un buffer de salida de tipo stream, para el cliente.

Directivas de JSP - Directiva Page

- **autoflush = "true | false".**
 - Un valor de true (por defecto) indica que el buffer debería desacargarse 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 que el servidor podría utilizar una superclase personalizada.
- **info = "message".**
 - Define un string que puede usarse para ser recuperado mediante el método getServletInfo.

Directivas de JSP - Directiva Page

- `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.
- `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`. Tiene el mismo valor que el scriptlet usando “`response.setContentType`”.

Directivas de JSP -

Directiva Page

- `isThreadSafe="true | false"`
 - Un valor de true (por defecto) indica un procesamiento del servlet normal, donde múltiples peticiones pueden procesarse simultáneamente con un sólo ejemplar del servlet, bajo la suposición que el autor sincroniza los recursos compartidos. Un valor de false indica que el servlet debería implementar SingleThreadModel.
 - `language="java"`
 - En algunos momentos, esto está pensado para especificar el lenguaje a utilizar. Por ahora, no debemos preocuparnos por él ya que java es tanto el valor por defecto como la única opción legal.
-

Directivas de JSP - Directiva Include

- Indica al motor JSP que incluya el contenido del fichero correspondiente en el JSP, insertándolo en el lugar de la directiva del JSP.
- El contenido del fichero incluido es analizado en el momento de la traducción del fichero JSP y se incluye una copia del mismo dentro del servlet generado.
- Una vez incluido, si se modifica el fichero incluido no se verá reflejado en el servlet
- El tipo de fichero a incluir puede ser un
 - fichero HTML (estático)
 - fichero jsp (dinámico)

Sintaxis: **<%@ include file="Nombre del fichero" %>**

Ejemplo de la Directiva Include

Ejemplo: Página JSP que incluye el contenido de dos ficheros (una página HTML y una página JSP)

```
<HTML>
  <head>
    <title> Página de prueba de directivas de
    compilación </title>
  </head>
  <body>
    <h1> Página de prueba de directivas de
    compilación </h1>
    <%@ include file="/fichero.html" %>
    <%@ include file="/fichero.jsp" %>
  </body>
</HTML>
```

Ejemplo de la Directiva Include

- Siendo, por ejemplo el fichero HTML el siguiente:

```
<HTML>
<head> <title> Hola, Mundo </title> </head>
<body> <h1> ¡Hola, Mundo! </h1>
</body>
</HTML>
```

- y el fichero JSP el siguiente:

```
<%@ page info="Un ejemplo Hola Mundo"
   import="java.util.Date" %>
La fecha de hoy es:    <%= new Date().toString() %>
```

Directivas de JSP - Directiva Taglib

- Permite extender los marcadores de JSP con etiquetas o marcas generadas por el propio usuario (etiquetas personalizadas).
- Se hace referencia a una biblioteca de etiquetas que contiene código Java compilado definiendo las etiquetas que van a ser usadas, y que han sido definidas por el usuario

Sintaxis

```
<%@ taglib uri="taglibraryURI" prefix="tagPrefix" %>
```

Elementos de una página JSP

- Código HTML
- Directivas
 - page
 - include
 - taglib
- Elementos de Scripting (guiones)
- Acciones (marcas estandar)

Elementos Scripting

- Permiten la inserción de **Declaraciones**, **Código Java** (scriptlets) y **Expresiones** dentro de una página JSP
- Hay 3 categorías:
 - Declaraciones
 - Código Java arbitrario
 - Expresiones

Elementos Scripting - Declaraciones

- Usadas para definir variables y métodos **con ámbito de clase** para el servlet generado
- Estas variables o métodos declarados pasarán a ser variables de instancia de la clase servlet generada
- Esto significa que serán globales a todo el servlet generado para la página
- Sintaxis
`<%! Declaración %>`
- Ejemplo:
`<%! int contador >`
-

Ejemplo de uso de Declaraciones

Uso de un contador que indica el número de veces
que se accede a una página
(ej5_declaraciones.jsp)

```
<HTML>
<head><title> Página de control de declaraciones
    </title></head>
<body>
    <h1> Página de control de declaraciones </h1>
    <%! int i=0 ; %> <!-- Esto es una declaración -->
    <% i++; %> <!-- scriptlet (código Java) que se ejecuta-->
    HOLA MUNDO
    <%= "Este JSP ha sido accedido " + i + " veces" %>
    <!-- Esto es una expresión que se evalúa y se sustituye en
        la página por su resultado -->
</body>
</HTML>
```

Elementos Scripting - Scriptlets

- Un scriptlet es un bloque de código Java insertado en la página y ejecutado durante el procesamiento de la respuesta
- El código introducido se inserta directamente en el método `_jspService()` del servlet generado para la página

Sintaxis

```
<% código Java %>
```

- Ejemplo

```
<% int i,j;  
    for (i=0;i<3;i++) {  
        j=j+1;  
    }  
%>
```

Ejemplo de uso de Scriptlets

- Página JSP que usa código Java para repetir 10 veces un saludo

```
<HTML>
<head><title> Página de ejemplo de scriptlet
    </title></head>
<body>
    <h1> Página de ejemplo de scriptlet </h1>
    <%
        for (int i=0; i<10; i++) {
            out.println("<b> Hola a todos. Esto es un ejemplo de
scriptlet " + i + "</b><br>");
            System.out.println("Esto va al stream System.out" + i
);
            //Esto último va a la consola del Java, no al
cliente,
            //out a secas es para la respuesta al cliente.
        }
    %>
</body>
</HTML>
```

Elementos Scripting - Expresiones

- Notación abreviada que envía el valor de una expresión Java al cliente.
- La expresión se traduce por la llamada al método `println` del objeto `out` dentro del método `_jspService()`, con lo que en cada petición, la expresión es evaluada y el resultado **se convierte a un String y se visualiza**

Sintaxis

`<%= Expresión Java a evaluar %>`

- Ejemplo

```
<%= "Esta expresión muestra el valor de un contador " +  
     contador %>
```

*Nota: será necesario que previamente contador haya tomado
un valor a través de un scriptlet*

Ejemplo de uso de Expresiones

En esta página JSP la expresión consiste en crear un objeto y llamar a uno de sus métodos. El resultado es un string que se muestra al cliente

```
<HTML>
  <head>
    <title> Página de ejemplo de expresiones </title>
  </head>
  <body>
    <h1> Página de ejemplo de expresiones </h1>
    Hola a todos, son las <%= new Date().toString() %>
  </body>
</HTML>
```

Ejercicios

1. Con los ejemplos existentes en el directorio Ejecutarlos, visualizar el contenido de los ficheros, visualizar el contenido de alguno de los servlets generados para ellos
1. Modificar el fichero ej4_incluido.html y comprobar que dichas modificaciones no aparecen reflejadas por la directiva
2. Crear una aplicación que funcione como calculadora con todas las operaciones.
 - Basarse en ej2_saludo.html para hacer el formulario y sumadora.jsp
 - Usar la instrucción if(operación.equals("restar")) resultado=...
3. Crear una aplicación que funcione como euroconversor

Elementos de una página JSP

- Código HTML
- Directivas
 - page
 - include
 - taglib
- Elementos de Scripting (guiones)
 - Declaraciones
 - Código Java arbitrario
 - Expresiones
- Acciones
 - Acciones estándar
 - Acciones personalizadas
-

Acciones estándar

- Son marcas estándar, con formato XML, que afectan al comportamiento en tiempo de ejecución del JSP y la respuesta se devuelve al cliente.
- En la traducción de JSP al servlet, la marca se reemplaza por cierto código Java que define a dicha marca. Una marca por tanto define un cierto código Java (es como una macro)
- Constan de un prefijo y un sufijo además de una serie de atributos. El prefijo es siempre “**jsp**” en las acciones estándar

Sintaxis

```
<jsp:sufijo    atributos/>
```

- Ejemplo

```
<jsp:include page="mijsp.jsp" "flush="true" />
```

Acciones estándar existentes

- <jsp:include>
- <jsp:forward>
- <jsp:param>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:plugin>
-

Acción jsp:include

- Permite incluir un recurso especificado por la URL, en la petición JSP en **tiempo de ejecución**.
 - Cuando se realiza la traducción de JSP al servlet, dentro del método `_jspService()` se genera el código que comprueba si existe el recurso (página) y si no se crea, invocándolo a continuación.
 - Cuando se ejecuta el servlet, se invoca al recurso que realiza la operación y devuelve el resultado al servlet
 - El elemento incluido puede acceder al objeto request de la página padre, y además de los parámetros normales, a los que se añadan con `<jsp:param>`
-

Acción jsp:include

Sintaxis

```
<jsp:include page="URL" flush="true">  
  <jsp:param name="nombre clave" value="valor" /> (no obligatorios)  
  ....  
</jsp:include>
```

Acción include vs Directiva include

Es importante distinguir entre directiva include y acción include

- Directiva <%@ include file="Nombre fichero" /> se añade el código al servlet que se genera para la página en tiempo de compilación y se incluye el contenido EXISTENTE EN EL MOMENTO INICIAL.
- Acción <jsp:include> no se añade código al servlet, sino que se invoca al objeto en tiempo de ejecución y se ejecuta el contenido EXISTENTE EN EL MOMENTO DE LA PETICIÓN

Ejemplo de la acción include

```
<HTML>
  <head>
    <title> Inclusión de fichero </title>
  </head>
  <body>
    <h1> Inclusión de ficheros </h1>
    <%@ include file="incluido.jsp" %>
    <jsp:include page="incluido.jsp" />
  </body>
</HTML>
```

- Fichero incluido ("*incluido.jsp*")

```
<%@ page import="java.util.Date" %>
<%= "Fecha actual es " + new Date() %>
```

-

Tipos de acciones existentes

- <jsp:include>
- <jsp:param>
- <jsp:forward>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:plugin>
-

Acción jsp:param

- Se usa como submarca dentro de cualquier otra marca
- Sirve para pasar parámetros a un objeto

Sintaxis

```
<jsp:....      >  
  <jsp:param name="nombre clave" value="valor" /> (no obligatorios)  
....  
</jsp:....    >
```

Tipos de acciones existentes

- <jsp:include>
- <jsp:param>
- <jsp:forward>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:plugin>
-

Acción jsp:forward

- Esta marca permite que la petición sea redirigida a otra página JSP, a otro servlet o a otro recurso estático
- Muy útil cuando se quiere separar la aplicación en diferentes vistas, dependiendo de la petición interceptada.
- Cuando se ejecuta el servlet se redirige hacia otro servlet y no se vuelve al servlet original

Sintaxis

```
<jsp:forward page="URL">  
  <jsp:param name="nombre clave" value="valor" /> (no obligatorios)
```

....

- </jsp:forward>

Ejemplo de la acción forward

Formulario HTML que pide nombre y password y los envía a una página jsp que lo analiza (*forward.jsp*)

```
<HTML>
  <head>  <title> Ejemplo de uso del forward
  </title> </head>
  <body>
    <h1> Ejemplo de uso del forward </h1>
    <form method="post" action="forward.jsp">
      <input type="text" name="userName">
      <br> y clave:
      <input type="password" name="password">
      </p>
      <p><input type="submit" name="log in">
    </form>
  </body>
</HTML>
```

Ejemplo de la acción forward

- Página JSP que lo ejecuta
 - No tiene nada de HTML
 - En función de los valores de los parámetros de la petición redirige a una segunda página JSP (si es un usuario y una clave determinadas) o bien recarga la página inicial (incluyéndola)
 - Mezcla código Java puro con acciones estándard
-

Ejemplo de la acción forward

```
<% if  
((request.getParameter("userName").equals(  
"Ricardo")) &&  
(request.getParameter("password").equals(  
"xyzzy")) ) {  
%>  
    <jsp:forward page="saludoforward.jsp" />  
<% } else { %>  
<%@ include file="forward.html"%>  
<% } %>
```

Ejemplo de la acción forward

El programa *saludoforward.jsp* podría ser el siguiente:

```
<HTML>
  <head>
    <title> Saludo al cliente </title>
  </head>
  <body>
    <h1> Saludo al cliente</h1>
    <%>
      out.println("Bienvenido a la nueva
      aplicación");
    %>
  </body> </HTML>
```

Tipos de acciones existentes

- <jsp:include>
- <jsp:forward>
- <jsp:param>
- <jsp:useBean>
- <jsp:setProperty>
- <jsp:getProperty>
- <jsp:plugin>
-

Acción jsp:useBean

- Esta marca sirve para instanciar un JavaBean si no existe, o localizar una instancia ya existente, para su uso desde la página.
- **Los JavaBeans son objetos Java que cumplen ciertas características en cuanto a su diseño: son serializables y tienen un constructor implícito.**
- Se utilizan para reducir al máximo el código Java insertado en una página JSP. En lugar de meterlo directamente en el fichero JSP se mete en una clase y una instancia de ésta se llama desde el JSP
- Permite separar la lógica de ejecución (en el JavaBean) de la presentación (en el servlet generado)
 - Se encapsula el código Java en un objeto (JavaBean) y se instancia y usa con el JSP.

Acción jsp:useBean

- Los JavaBeans se caracterizan porque a sus atributos (llamados propiedades) se acceden (por convenio) a través de los métodos `setNombreAtributo` y `getNombreAtributo`
 - Ojo, si el nombre va en minúsculas el método lleva la inicial del nombre en mayúsculas para “nombre” se pone “`getNombre`”.
- Si se usa un JavaBean en una página **habrá que definir la clase correspondiente**, creando los métodos set y get para los atributos definidos. Normalmente se suele definir dentro de un paquete.
- Dentro del servlet generado se puede llamar a métodos de un JavaBean que se encarguen de realizar ciertas operaciones y el servlet muestra el resultado de las mismas

Acción jsp:useBean

Sintaxis

```
<jsp:useBean id="nombre" scope="nombreámbito"  
detalles />
```

Características de los atributos de esta acción:

- En id se define el nombre asignado al JavaBean (identificador asociado)
- El ámbito se refiere a dónde puede referenciarse el JavaBean. Permite compartir objetos en una sesión
 - “page”, “request”, “session” y “application”

Acción jsp:useBean

- Los detalles pueden ser:
 - **class=“Nombre de la clase del JavaBean”** (es lo que más se usa)
 - **id:** Da un nombre a la variable que referencia al bean. Se usará un objeto bean anterior en lugar de instanciar uno nuevo si se puede encontrar uno con el mismo id y scope.
 - **class:** Designa el nombre completo del paquete del bean.
 - **scope:** Indica el contexto en el que el bean debería estar disponible. Hay cuatro posibles valores: page, request, session, y application.
 - **type:** Especifica el tipo de la variable a la que se referirá el objeto.
 - **beanName:** Da el nombre del bean, como lo suministraríamos en el método instantiate de Beans. Esta permitido suministrar un type y un beanName, y omitir el atributo class.

Acción jsp:setProperty

- Esta marca se utiliza junto con la marca useBean para asignar valor a las propiedades del Bean
 - En el método `_jspService()` del servlet generado se invoca al método `set` de la propiedad deseada.
 - 2 usos:
 - Despues de un useBean.

```
<jsp:useBean id="myName" ... />  
...  
<jsp:setProperty name="myName"  
    property="someProperty" ... />
```
- Se ejecuta siempre que haya una solicitud.

Acción jsp:setProperty

- Dentro de un useBean

```
<jsp:useBean id="myName" ... >
```

```
...
```

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

```
</jsp:useBean>
```

Sólo se ejecuta cuando haya que instanciar un bean.

Acción jsp:setProperty

- Sintaxis

```
<jsp:setProperty name="identificador del Bean"  
detalles de la propiedad />
```

- Donde los detalles pueden ser

- **property:** Este atributo requerido indica la propiedad que queremos seleccionar. Sin embargo, hay un caso especial: un valor de "*" significa que todos los parámetros de la petición cuyos nombres correspondan con nombres de propiedades del Bean serán pasados a los métodos de selección apropiados.
- **value:** Este atributo opcional especifica el valor para la propiedad. Los valores string son convertidos automáticamente a lo que corresponda mediante el método standard valueOf. No se pueden usar value y param juntos, pero si está permitido no usar ninguna.
- **param:** Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición

Acción jsp:setProperty

- property="*" (se cogen como propiedades y valores todos los parámetros del objeto request)
- property="Nombre" (se coge un parámetro con el mismo nombre del objeto request)
- property="Nombre" param="NombreParámetro" (si se desean nombres distintos)
- property="Nombre" value="valor parámetro" (se asignan propiedades arbitrarias con valores concretos)

Acción jsp:getProperty

- Se utiliza para obtener el valor de las propiedades de un Bean.
- Dentro del método `_jspService()` del servlet generado se accede al valor de una propiedad, **lo convierte a string y lo imprime en la salida del cliente (objeto out)**.

Sintaxis

```
<jsp:getProperty> name="nombre del Bean"  
    property="Nombre de la propiedad" />
```

Ej. de uso de JavaBean (Formulario cliente)

(lenguaje/beans.html)

```
<HTML>
  <head>
    <title> Página de prueba del uso de beans </title>
  </head>
  <body>
    <h1> Página de prueba del uso de beans </h1>
    <form method="post" action="beans.jsp" >
```

Se envía el formulario al servicio cuyo fichero es “beans.jsp”

```
      <p> Por favor, introduce tu nombre:<br/>
      <input type="text" name="nombre">
      <br> ¿Cuál es tu lenguaje de programación favorito?
      [...]
```

Ej. de uso de JavaBean (Formulario cliente)

[...]

```
<select name="lenguaje">
    <option value="Java"> Java
    <option value="C++"> C++
    <option value="Perl"> Perl
</select>
</p>
<p>
    <input type="submit" value="enviar información">
</p>
</form>
</body>
</HTML>
```

Ej. de uso de JavaBean (Fichero jsp)

- Fichero *lenguaje/beans.jsp* que usa un Bean para elaborar los resultados y los muestra

```
<jsp:useBean id="lenguajeBean" scope="page"  
 class="es.uniovi.di.servinfo.lenguaje.LenguajeBean">
```

Usa un Bean generado a partir de la clase denominada “LenguajeBean” que está en el paquete “es.uniovi.di.servinfo.lenguaje” con ámbito de página

```
<jsp:setProperty name="lenguajeBean" property="*"/>
```

Las propiedades del Bean las toma del objeto petición

```
</jsp:useBean>
```

```
<HTML>
```

```
<head><title> Resultado de prueba del uso de beans  
</title> </head>
```

Ej. de uso de JavaBean (Fichero jsp)

```
<body> <h1> Resultado de prueba del uso de beans </h1>
<p> Hola
<jsp:getProperty name="lenguajeBean" property="nombre" />.
</p>
coge el valor de la propiedad indicada y lo imprime para lo cual se ejecuta un método del Bean con el nombre de la propiedad
<p> Tu lenguaje favorito es
<jsp:getProperty name="lenguajeBean" property="lenguaje"
/>. </p>
<p> Mis comentarios acerca del lenguaje
<p> <jsp:getProperty name="lenguajeBean"
property="comentariosLenguaje" />. </p>
</body> </HTML>
```

Ej. de uso de JavaBean (Definición de la clase)

Clase LenguajeBean

(classes\es.uniovi.di.servinfo.lenguaje.LenguajeBean.java)

```
package es.uniovi.di.servinfo.lenguaje;

public class LenguajeBean {
    // propiedades
    private String nombre;
    private String lenguaje;

    // constructor隐式的, no hace falta declararlo
    public LenguajeBean() {}
```

- [...]

Ej. de uso de JavaBean (Definición de la clase)

```
// Coloca el valor a la propiedad "Nombre"
public void setNombre(String nombre) {
    this.nombre=nombre;
}
// Recupera el valor de la propiedad nombre
public String getNombre() {
    return nombre;
}
// Coloca el valor a la propiedad "lenguaje"
public void setLenguaje(String lenguaje) {
    this.lenguaje=lenguaje;
}
// Consigue el valor de la propiedad "Lenguaje"
public String getLenguaje() {
    return lenguaje; } [ . . . ]
```

Ej. de uso de JavaBean (Definición de la clase)

```
/* Consigue el valor de la propiedad "comentariosLenguaje"
 */
public String getcomentariosLenguaje () {
    if (lenguaje.equals("Java")) {
        return "El rey de los lenguajes Orientados a
objetos";    }
else if (lenguaje.equals("C++")) {
    return "Demasiado complejo";
}
else if (lenguaje.equals("Perl")) {
    return "OK si te gusta el código incomprensible";    }
else {
    return "Lo siento, no conozco el lenguaje " +
lenguaje ;
}
}
```

Acción jsp:plugin

- **Permite incluir applets en la jsp que la define**, sin que el programador deba preocuparse ni por su correcta visualización en el cliente ni de que sus recursos se hospeden en el mismo servidor que la jsp. No obstante, cuando en la máquina cliente hay instaladas varias versiones del JRE, puede haber problemas.
- Lo que hace esta acción es identificar el tipo y versión del cliente web mediante el encabezado de petición User-Agent y generar código html específico para ese cliente que permita asegurar la correcta visualización del applet sin importar versión ni tipo de cliente.
- Tiene muchos atributos coincidentes con los de la etiqueta <applet> asociada a los applets Java. Los principales son los siguientes:

Acción jsp:plugin - Atributos

- **type**
 - El valor “applet” es el más normal.
 - Indica el tipo de componente que va a insertarse
 - Si incluimos un bean gráfico, utilizaríamos “bean”
 - Obligatorio
- **code**
 - Nombre de la clase del applet
 - Obligatorio

Acción jsp:plugin - Atributos

- **width**
 - Anchura en píxeles
 - Obligatorio
- **height**
 - Altura en píxeles
 - Obligatorio
- **codebase**
 - En el caso de que no se encuentre donde su archivo html asociado

Acción jsp:plugin - Atributos

- Se utiliza para ejecutar applets ubicados en un servidor distinto de aquel en el que está el jsp
- No es obligatorio
- **archive**
 - Indica el fichero jar donde están almacenados el applet y sus recursos
 - No es obligatorio
- Hay más, pero estos son los más utilizados

Acción jsp:plugin - Ejemplo

```
<html>
<head><title>Boletin de noticias con
    applet</title></head>
<body>
<h1>Boletin de noticias CON APPLET</h1>
<b>Jueguecillo de capitales (es un
    applet):</b><p>

<jsp:plugin type="applet"
    code="JuegoCapitales.class" width="700"
    height="350">
</jsp:plugin><hr><p>
```

Acción jsp:plugin - Ejemplo

```
<!-- Si los recursos del applet se encuentran en  
    un servidor distinto de aquél en  
que se encuentra la jsp donde se declara la  
acción -->  
<%-- <jsp:plugin type="applet"  
code="JuegoCapitales.class"  
codebase="http://www10.brinkster.com/trilc  
ejf/applets"  
width="700" height="350">  
</jsp:plugin> --%>
```

Acción jsp:plugin - Ejemplo

```
<b>Resumen de noticias:</b><p>
<ol>
    <li><jsp:include
        page="/Noticia1.html"/><hr><p>
    <li><jsp:include
        page="/Noticia2.html"/><hr><p>
    <li><jsp:include page="/Noticia3.jsp">
        <jsp:param name="origen" value="El País"
    />
        </jsp:include><hr><p>
</ol>
</body>
```

JSP 2.0

- Algunas nuevas características de JSP 2.0 frente a 1.2 orientadas a simplificar su desarrollo son:
 - Simple Expression Language (EL)
 - Fragmentos JSP
 - Ficheros de Tags
 - Manejadores de Etiquetas Simplificados

Simple Expression Language

- La Expression Language, está inspirada en los lenguajes de expresiones de ECMAScript y XPath
 - Simplifican el uso de expresiones en JSPs.
 - Permite la ejecución de expresiones fuera de los elementos de scripting de JSP
 - Fue introducida con JSTL 1.0 como un mecanismo alternativo al uso de expresiones en Java para asignar valores a atributos
- Desde JSP 2.0 el JSP container entiende expresiones en EL
- EL es mucho más tolerante sobre variables sin valor (`null`) y realiza conversiones automáticas de datos
- Se puede habilitar (por defecto) o deshabilitar el uso de expresiones EL:

```
<%@ page isScriptingEnabled="true|false"  
isELEnabled="true|false"%>
```
-

Sintaxis de EL

- Una expresión en EL contiene variables y operadores
- \${expr}, donde expr es:
 - Literales:
 - true o false
 - Integer
 - Floating point
 - String
 - null
- Ejemplos:
 \${false} <%-- evaluates to false --%>
 \${3*8}

Sintaxis de EL

- \${expr}, donde expr es:
 - Operadores:
 - Aritméticos: +, -, *, /, div, %, mod, -
 - Lógicos: and, &&, or, ||, !, not
 - Relacionales: ==, eq, !=, ne, <, lt, <, gt, <=, le, >=, ge
 - Vacío, empty, valida si una variable es null o una colección no tiene elementos
 - Llamadas a función, donde func es el nombre de la función y args es 0 o más argumentos separados por comas.
 - Condicional: A ? B: C, como en C y Java
- Ejemplos:

```
 ${ (6 * 5) + 5 } <%-- evaluates to 35 --%>
 ${empty name}
```

Sintaxis de EL

- \${expr}, donde expr es:
 - Objetos implícitos:
 - pageContext → contexto del JSP, puede usarse para acceder a objetos implícitos como request, response, session, out, servletContext, etc.
 - Ejemplo: \${pageContext.response}
 - param
 - \${param.name} <-> request.getParameter(name)
 - paramValues
 - \${paramvalues.name} <-> request.getParameterValues(name)
 - header
 - \${header.name} <-> request.getHeader(name)
 - headervalues
 - \${headervalues.name} <-> request.getHeaderValues(name)
 - cookie
 - \${cookie.name.value} → devuelve el valor de la primera cookie con el nombre dado
 - initParam
 - initParam.name → ServletContext.getInitparameter(String name)

Sintaxis de EL

- \${expr}, donde expr es:
 - Objetos implícitos:
 - pageScope → puede acceder a objetos dentro del contexto de página
 - \${pageScope.objectName}
 - \${pageScope.objectName.attributeName}.
 - requestScope
 - \${requestScope.objectName}
 - \${requestScope.objectName.attributeName}
 - sessionScope
 - \${sessionScope.name}
 - applicationScope

Ejemplos EL

- `${amount + 5}` , añade 5 a una variable amount
- `${order.amount + 5}`, añade 5 a la propiedad amount del bean order
 - Lo mismo sería hacer: `${order['amount'] + 5}`
- Para asignar un valor dinámico a un atributo podemos hacer:

```
<input name="firstName"  
value="${customer.firstName}">
```
-

Ejemplos EL

- El operador ?:

```
<select name="artist">
  <option value="1" ${param.artist == 1 ? 
    'selected' : ''}>
    Vesica Pisces
  <option value="2" ${param.artist == 2 ? 
    'selected' : ''}>
    Cortical Control
  <option value="3" ${param.artist == 3 ? 
    'selected' : ''}>
    vida vierra
</select>
```

Páginas de Error JSP

- Errores en servlets and JSPs desde su versión 2.0 vienen dadas en la variable `javax.servlet.error.exception`
- La propiedad `errorData` del objeto implícito `pageContext` expone información sobre el problema.
- `javax.servlet.jsp.ErrorData` tiene las siguientes propiedades:
 - `requestURI` → la URI para la que falló la petición
 - `servletName` → el nombre del servlet o JSP que falló
 - `statusCode` → el código del fallo
 - `throwable` → la excepción que causó la invocación de la página de error

Páginas de Error JSP

```
<%@ page isErrorPage="true" contentType="text/html" %>
<%@ taglib prefix="log" uri="http://jakarta.apache.org/taglibs/log-1.0" %>
```

Sorry, but things didn't work out as planned. I've logged as much as I know about the problem, so rest assured that my master will look into what's wrong as soon as he's sober.

```
<jsp:useBean id="now" class="java.util.Date" />
<log:fatal>
-----
${now}
Request that failed: ${pageContext.errorData.requestURI}
Status code: ${pageContext.errorData.statusCode}
Exception: ${pageContext.errorData.throwable}
-----
</log:fatal>
```

Páginas de Error JSP

- En el `web.xml` se puede indicar que esta es la página de error mediante los siguientes elementos:

...

```
<error-page>
    <exception-
type>java.lang.Throwable</exception-type>
    <location>/error.jsp</location>
</error-page>
<error-page>
    <exception-code>500</exception-code>
    <location>/error.jsp</location>
</error-page>
```

...

web.xml en JSP 2.0

- Aunque se sigue utilizando web.xml hay dos diferencias principales:
 - Las reglas de web.xml están definidas en un XML Schema
 - Las configuraciones específicas de JSP han sido movidas a un nuevo elemento XML
- Usar Schemas en vez de DTDs nos permite colocar los elementos XML de primer nivel en cualquier orden y realizar aún mas verificaciones sobre la sintaxis de web.xml
- Todos los elementos específicos a la configuración de un JSP se encuentran agrupados dentro del nuevo elemento <jsp-config>.

Elemento jsp-config

- Su subelemento <jsp-property-group> es muy interesante.
 - Permite aplicar configuraciones a un grupo de JSPs que conforman con un patrón de url específico

```
...
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
...
```

Elemento jsp-config

Elemento	Descripción
<el-ignored>	Indica si ignorar o no las expresiones EL dentro de un JSP correspondiente al patrón URL indicado. Por defecto es false.
<scripting-invalid>	Si es true no se pueden introducir scriplets en el JSP.
<page-encoding>	Indica la codificación de caracteres para un conjunto de JSPs
<include-coda>	Indica el path correspondiente a un fichero a añadir al final de cada página JSP..
<include-prelude>	Lo mismo pero al comienzo de cada JSP.
<is-xml>	Si se asigna a true la sintaxis de los JSPs es en XML.



Ejercicios

1. Modificar la página ej4_directivainclude.jsp para incluir también el fichero “incluido.jsp” como acción include
2. Cambiar la página HTML o JSP y volver a ejecutar comprobando resultados
3. Ejecutar el ejemplo del Bean que evalúa el lenguaje preferido del cliente
4. Crear un Bean que funcione como sumadora.
 - Usar como base sumadora.html para sumadorabean.html, lenguajebean.jsp para sumadorabean.jsp
 - lenguajebean.java y sumadora.jsp para sumadorabean.java
5. Ampliar la sumadora a calculadora con más operaciones

Elementos de una página JSP - Resumen

- Código HTML
- Directivas <%@
 - page <%@ page ATRIBUTOS %>
 - include <%@ include file="Nombre del fichero" %>
 - taglib <%@ taglib uri="taglibraryURI" prefix="tagPrefix" %>
- Elementos de Scripting (guiones) <%
 - Declaraciones <% ! Declaración %>
 - Código Java arbitrario (scriptlets) <% código Java %>
 - Expresiones <%= Expresión Java a evaluar %>

Elementos de una página JSP - Resumen

- **Acciones** estándar *<jsp:*
 - <jsp:useBean>
 - <jsp:setProperty>
 - <jsp:getProperty>
 - <jsp:include>
 - <jsp:param>
 - <jsp:forward>
 - <jsp:plugin>
-
- Acciones personalizadas *<etiq:*

Problemas con JSPs

- JavaBeans nos permiten separar la parte de presentación de una página JSP de la implementación de una regla de negocio
 - Sin embargo, sólo 3 elementos acción en JSP se pueden usar para acceder a un bean:
 - `jsp:useBean`
 - `jsp:getProperty`
 - `jsp:setProperty`
 - Por tanto a menudo tenemos que incluir código en un JSP
 - JSP (1.1 en adelante) define custom tags que pueden ser usadas para definir acciones propietarias, de manera que en nuestro JSP únicamente tenemos código de marcado

Etiquetas personalizadas

Las acciones personalizadas están definidas por el programador de la aplicación web mediante el uso de etiquetas personalizadas.

Una etiqueta personalizada permite ocultar bajo ella un conjunto de acciones (definidas con instrucciones java) evitando que las mismas se incluyan en el fichero JSP.

Así pues, para incluir lógica de programa en una aplicación web, es posible realizarlo de tres modos

- - Incluyendo el correspondiente código Java en una página JSP
 - Incluyéndolo en un Java Bean que se llama desde la página JSP

Etiquetas personalizadas - Características

Ventajas que proporcionan

- Permiten reutilizar código
 - Usando bibliotecas de etiquetas con las funcionalidades más extendidas
- Permiten separar las funciones del diseñador web (que usa HTML y XML) de las del programador web (que usa Java)
 - Permiten invocar funcionalidad sin necesidad de introducir código Java en la página JSP
- Son más potentes que los JavaBeans
(página siguiente)

Etiquetas personalizadas - Características

- Son más potentes que los JavaBeans
 - El uso exclusivo de Beans no es suficiente para evitar todo tipo de código Java en una página JSP
 - Las marcas propias pueden lograr directamente lo que los beans pueden sólo lograr en conjunción con los scriptlets

Ej: Una etiqueta que compruebe que un usuario existe (u otra condición) y se redirija a una página de éxito o de error en función de la comprobación.

- El uso exclusivo de Beans no es suficiente.
 - Podríamos usar un bean para comprobar que el usuario existe.
 - El hecho de redirigir a una página o a otra en función de si existe o no debe hacerse en una página JSP (puesto que desde un bean no se puede reenviar a una página JSP) y debe utilizarse código Java para realizar la comparación (if...)
- ○ JavaBeans no permiten interactuar con la JSP.

Etiquetas personalizadas - Ejemplo de uso

- Si se desea solicitar una lista de libros de una base de datos con JSP podría hacerse utilizando un Bean que devuelva la lista de libros.

```
<jsp:useBean id="biblio" class="Libros" />
<% ResultSet res=getLibros("Libros");%>
```

- Posteriormente se recorre la lista y se muestra el atributo “título de cada libro”. Esto no se puede hacer en el Bean porque en él no se puede acceder al objeto out para devolver el resultado al cliente, por lo que hay que incluirlo en el fichero JSP

```
<% while (res.next()) { %>
    <%=res.getString("titulo") %>
• <% } %>
```

Etiquetas personalizadas - Ejemplo de uso

- La inclusión de las acciones personalizadas permite ocultar todo el código Java al diseñador web, permitiendo que éste utilice etiquetas como si de cualquier otra etiqueta HTML se tratase

```
<%@ taglib uri="/bibliolib" prefix="etiq" %>
<etiq:getLibro id="libros" />
<ul>
<etiq:bucle name="libros" bucleId="biblio" >
  <li> <jsp:getProperty name="libros" property="titulo"
  />
</etiq:bucle>
```

Etiquetas personalizadas - Ventajas

- Reducen o eliminan código Java en las páginas JSP
- Sintaxis muy simple, como código HTML
- **Tienen acceso a todos los objetos implícitos** de las páginas JSP
- Facilitan la reutilización de código porque pueden ser usadas en cualquier aplicación web
- Se pueden anidar, consiguiendo interacciones complejas
- Pueden ser personalizadas a través de atributos y determinadas de forma estática o dinámica

Etiquetas personalizadas - Definición

1. Definición de la estructura en un fichero de **definición de etiquetas** (biblioteca de etiquetas)
 - Fichero con extensión .tld “Tag Library Descriptor (TLD)” es un fichero XML que describe la biblioteca
 - Los .tld se guardan en un directorio TLDS dentro de WEB-INF de la aplicación
 - **La biblioteca (.tld) debe estar a su vez definida en el fichero web.xml, que existirá en el directorio WEB-INF de la aplicación**
2. Definición de la funcionalidad asociada a la etiqueta (lo que queremos que haga cuando se use la etiqueta) . Se define a través de un **controlador de la etiqueta** (clase que implementa la etiqueta)
 - ○ Es un JavaBean, con propiedades que coinciden con los¹⁰² atributos XML de la etiqueta

Etiquetas personalizadas - Uso en una página JSP

- Cuando se desee usar la etiqueta, se incluye en el fichero JSP la directiva JSP *taglib* para importar las etiquetas

```
<%@ taglib uri="dirección de la biblioteca" prefix="tagPrifix"
%>
```

- Posteriormente se usa la etiqueta dentro de la página

Ejemplos:

<eti:holamundo/> esto es una etiqueta que muestra un mensaje
de hola Mundo

<eti:hola nombre=Marian/> Muestra un mensaje hola Marian

- **<eti:suma 3 4/>** muestra la suma de dos valores

Etiquetas personalizadas - Ejemplo de uso en un fichero JSP

- Caso simple, sin atributos ni cuerpo, que saca código HTML y contenido dinámico (etiquetas\ejtag1_hola.jsp)

```
<%@ taglib uri="/WEB-INF/tlds/etiquetas.tld"
prefix="ejemplo" %>

<HTML>
  <head><title> Saludo al cliente mediante el uso de una
  etiqueta </title></head>
  <body>
    <h1> Saludo al cliente con una etiqueta </h1>
    <ejemplo:Hola />
  </body>
</HTML>
```

Etiquetas personalizadas - Partes de un fichero TLD

- Descriptor de la biblioteca TLD *Tag Library Descriptor*
 1. Información acerca de la biblioteca (cabecera)
 2. Definición de la estructura de cada etiqueta
-

Descriptor de la Librería de Etiquetas (Tag Library Descriptor)

- Fichero XML que define una librería de tags y sus etiquetas. Consta de:
 - Prólogo XML como todo documento XML
 - El elemento raíz **<taglib>**, que tiene como sub-elementos:
 - **tlib-version** → versión de la librería de etiquetas
 - **jsp-version** → la versión de JSP, actualmente 2.0
 - **short-name** → nombre corto para la librería
 - **description** → información para documentación
 - **uri** → enlace a más información sobre tag library
 - **tag** → elemento más importante, puede haber varios y tiene sub-elementos

Elemento <tag> del descriptor de una librería de etiquetas

- Puede tener los siguientes sub-elementos (**name** and **tag-class** son los únicos obligatorios):
 - **name** → identificador de la etiqueta
 - **tag-class** → nombre de la clase completa que realiza el procesamiento de esta etiqueta
 - **tei-class** → clase de ayuda de esta etiqueta
 - **body-content** → el tipo de cuerpo de una etiqueta:
 - **empty** → no hay cuerpo
 - **JSP** → cuerpo es un conjunto de elementos
 - **tagdependent** → cuerpo debe ser interpretado por la etiqueta
 - **description**
 - **attribute**: cero o más atributos que puede tener tres subelementos:
 - **name** (obligatorio)
 - **required: true** o **false** (valor por defecto **false**)
 - **rtextrvalue**: determina si el valor de este atributo se puede determinar en tiempo de ejecución
 - ...

Definición de la estructura de una etiqueta

- En la definición de una etiqueta personalizada debe aparecer al menos:

<tag>

- Indica comienzo de definición

<name> Hola </name>

- Nombre de la etiqueta

<tagclass> tagPaquete.HolaTag </tagclass>

- Nombre de la clase que implementa la funcionalidad

</tag>

- Fin de la etiqueta

- Además puede llevar

Atributos, anidaciones y cuerpo

Ejemplo de definición de estructura de una etiqueta

TLD Tag Library Descriptor (WEB-INF\tlds\etiquetas.tld)

Información acerca de la biblioteca (cabecera)

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib >
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname> ejemplos </shortname>
    <info> Biblioteca de ejemplos sencillos </info>
```

Información de cada etiqueta

```
<tag>
    <name> Hola </name>
    <tagclass> es.uniovi.di.servinfo.etiquetas.HolaTag
    </tagclass>
    <info> Ejemplo simple </info>
</tag>
• </taglib>
```

La API de Etiquetas Personalizadas

- Paquete `javax.servlet.jsp.tagext`
- Un manejador de etiquetas (Tag Handler) es una clase que está ligada a una etiqueta personalizada y es invocada cada vez que el contenedor de JSPs encuentra la etiqueta.
- En JSP 1.2, la clase `javax.servlet.jsp.tagext` tiene 4 interfaces y 12 clases
 - Los dos interfaces más importantes son `Tag` y `BodyTag`
 - Estos interfaces dictan el ciclo de vida de un manejador de etiquetas

Manejadores de etiquetas (Tag Handlers)

- Un manejador de etiquetas tiene acceso a un API a través del cual se puede comunicar con una página JSP. El punto de entrada a la API es el objeto `javax.servlet.jsp.PageContext`, a través del cual el `TagHandler` puede recuperar todos los otros objetos implícitos (`request`, `session`, and `application`) accesibles desde una página JSP.
- Los objetos implícitos pueden tener atributos con nombre asociados con ellos. Tales atributos son accesibles usando métodos `[set|get]Attribute`.
- Si la etiqueta está anidada, un Tag Handler también tiene acceso al Tag Handler (llamado *parent*) de la etiqueta padre.
- Un conjunto de clases Tag Handler relacionadas constituyen una tag library y son empaquetadas como un fichero JAR.

Ciclo de vida de las etiquetas

Cuando el motor JSP encuentra la etiqueta durante la traducción de la página JSP al servlet

1. Se crea un objeto de la clase etiqueta si no existe
2. Se invocan sus métodos ***set*** de cada uno de sus atributos
3. Se invoca el método ***doStartTag()***. Este método finaliza con una de estas opciones
 1. SKIP_BODY (si no hay cuerpo)
 2. EVAL_BODY_INCLUDE
4. Si no hay cuerpo se invoca ya a ***doEndTag()*** y se acaba

Ciclo de vida de las etiquetas

Si hay cuerpo,

1. Se invoca el método `setBodyContent()`
Cualquier salida de la etiqueta se deriva al objeto `BodyContent`
2. Se invoca el método `doInitBody()` permite realizar acciones antes de ser evaluado el cuerpo. La salida se guarda en el buffer `BodyContent`
3. Se procesa el cuerpo de la etiqueta (se pasa el contenido al buffer del `BodyContent`)
4. Se invoca el método `doAfterBody()`. Al final del método se puede determinar por dónde seguirá el ciclo de vida, devolviendo una de las siguientes constantes:
 - o `EVAL_BODY_AGAIN`
 - o `SKIP_BODY`
5. Si `EVAL_BODY_AGAIN` se vuelve al paso 3
6. Se invoca el método `doEndTag()` y se acaba

El interfaz Tag

- Este interfaz define los siguientes métodos:
 - `doStartTag`
 - `doEndTag`
 - `getParent`
 - `setParent`
 - `setPageContext`
 - `release`

Definición de la funcionalidad de una etiqueta

- La funcionalidad de una etiqueta está definida en una clase controladora (clase Java)
- Permite instanciar un objeto que será invocado por el servlet generado para la página JSP que usa la etiqueta
- JSP 1.2 proporciona 3 interfaces de manejadores de etiquetas:
 - *Tag*
 - *IterationTag*
 - *BodyTag*
- Para evitar que el programador tenga que codificar todos los métodos también proporciona varias clases de ayuda:
 - *TagSupport*, implementa todos los métodos de la interfaz

Definición de la funcionalidad de una etiqueta

- La forma básica de definición de funcionalidad es construir una clase que herede de una clase de ayuda, por ejemplo **TagSupport**, y redefinir los métodos que queramos cambiar.
- Dos métodos básicos:
 - *doStartTag()* llamado cuando se abre la etiqueta
 - *doEndTag()* llamado cuando se cierra
 - Si la etiqueta tiene atributos hay que definir los métodos set y get para los mismos
 - Otros métodos son *doInitBody*, *doAfterBody()*, si tiene cuerpo, etc.
- Desde el servlet para la página se invoca a estos métodos

Ejemplo de definición de la clase controladora

(WEB-INF\classes\es\uniovi\di\servinfo\etiquetas\HolaTag.java)

```
package tagPaquete;

import java.io.IOException;
import java.util.Date;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HolaTag extends TagSupport {

    public int doStartTag() throws JspTagException {
        // el motor JSP llama este método cuando encuentre el
        // comienzo de una marca implementada por esta clase
        return SKIP_BODY;
    }
}
```

[...]

Ejemplo de definición de la clase controladora

```
public int doEndTag() throws JspTagException {  
    // el motor llama este método cuando encuentre el  
    // final de una marca implementada por esta clase  
    String dateString = new Date().toString();  
    try {  
        pageContext.getOut().write("Hola mundo.<br/>");  
        pageContext.getOut().write("Mi nombre es" +  
            getClass().getName() + "y son las " +  
            dateString + "<p/>");  
    } catch (IOException ex) {  
        throw new JspTagException  
            ("Error: la etiqueta hola no puede escribir en la  
            salida del JSP");  
    }  
    return EVAL_PAGE;  
}  
  
} // clase HolaTag
```

Usando atributos en una etiqueta personalizada I

- Objetivo: Etiqueta personalizada para imprimir un saludo que incluya el nombre del usuario
- El manejador debe definir un método `setNombre` para asignar un atributo al valor

Etiqueta con atributos: definición en la tld

```
<taglib>
[...]

<tag>
  <name>saludo</name>
  <tagclass>es.uniovi.di.servinfo.etiquetas.HolaTagAtribu
  to</tagclass>
  <bodycontent> empty</bodycontent>
  <info>Esta es una etiqueta muy simple de saludo
  </info>
  <attribute>
    <name>nombre</name>
    <required>false</required>
    <rtextvalue>false</rtextvalue>
  </attribute>
</tag>

</taglib>
```

Etiqueta con atributos: definición de la clase controladora

```
package es.uniovi.di.servinfo.etiquetas;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HolaTag extends TagSupport {
    // Atributo de la etiqueta definido en el
    // fichero TLD
    private String nombre = null;

    public void HolaTag() { }
```

Etiqueta con atributos: definición de la clase controladora

```
// Métodos set y get del atributo "nombre" que
// se ha definido en el fichero TLD para esta etiqueta
public void setNombre( String _nombre ) {
    nombre = _nombre;
}

public String getNombre() {
    return( nombre );
}
```

Etiqueta con atributos: definición de la clase controladora

```
// Al inicio de la etiqueta
public int doStartTag() throws JspTagException {
    try {
        JspWriter out = pageContext.getOut();
        out.println( "<table border=1>" );
        if( nombre != null ) {
            out.println( "<tr><td> Hola " +nombre+ "
</td></tr>" );
            }
        else { out.println( "<tr><td>Hola Mundo
JSP</td></tr>" ); }
    } catch( Exception e ) { throw new
JspTagException( e.getMessage() ); }
    return( SKIP_BODY );
}
```

-

Etiqueta con atributos: definición de la clase controladora

```
// Al cierre de la etiqueta
public int doEndTag() throws JspTagException {
    try {
        /* Se utiliza el pageContext para obtener el
        objeto de salida, sobre el que colocar la etiqueta
        HTML de cierre de la tabla */
        pageContext.getOut().print( "</table>" );
    } catch( Exception e ) {
        throw new JspTagException( e.getMessage() );
    }
    return( SKIP_BODY );
}
```

Etiqueta con atributos: definición de la clase controladora

```
public void release() {  
    /* Llama al método release() del padre, para que se  
    devuelvan todos los recursos utilizados al sistema.  
    Esta es una buena práctica, sobre todo cuando se  
    están utilizando jerarquías de etiquetas */  
    super.release();  
}  
} // de la clase
```

Etiqueta con atributos: uso de la etiqueta

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01  
Transitional//EN">  
  
<%@ taglib uri="/WEB-INF/tlds/ejemplo2.tld"  
prefix="etiq" %>  
  
<html>  
<head>  
    <title>Ejemplo Etiquetas, Etiqueta Saludo</title>  
    <meta http-equiv="Content-Type" content="text/html;  
        charset=iso-8859-1">  
</head>
```

Etiqueta con atributos: uso de la etiqueta

```
<body>
<h2>Ejemplo de biblioteca de etiquetas:
<i>HolaTagAtributo.java</i></h2>
```

En esta página se muestra el uso de esta etiqueta, que se invoca en primer lugar especificando un valor para el atributo *nombre* y luego sin ningún argumento.

```
<p>
<hr><center>
  <etiq:saludo nombre="Agustín"/>
</center><hr>
<etiq:saludo />
</body>
</html>
```

-

El interfaz IterationTag

- Extiende el interfaz Tag añadiendo un nuevo método `doAfterBody`, que puede devolver:
 - `Tag.SKIP_BODY` → el cuerpo se ignora y el contenedor llama a `doEndTag`
 - `IterationTag.EVAL_BODY AGAIN` → `doAfterBody` es llamado de nuevo

Ejemplo IterationTag I

- Objetivo: calcular una potencia: $2^3=8$

```
package es.deusto.customtags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class PowerTag
    //extends TagSupport {
    implements IterationTag {

    private PageContext pageContext;
    private int number;
    private int power;
    private int counter = 0;
    private int result = 1;

    // the setter for number
    public void setNumber(int number) {
        System.out.println("number: " + number);
        this.number = number;
    }

    public int getNumber() {
        return this.number;
    }

    ...
}
```

Ejemplo IterationTag II

```
// the setter for power
public void setPower(int power) {
    System.out.println("power: " + power);
    this.power = power;
}

public int getPower() {
    return this.power;
}
public void setParent(Tag t) {
}

public void setPageContext(PageContext p) {
    System.out.println("setPageContext");
    pageContext = p;
}

public void release() {
    System.out.println("doAfterBody");
}

public Tag getParent() {
    return null;
}
```

Ejemplo IterationTag III

```
public int doStartTag() {
    System.out.println("doStartTag");
    return EVAL_BODY_INCLUDE;
}

public int doAfterBody() {
    System.out.println("doAfterBody");
    counter++;
    result *= number;
    if (counter >= power)
        return SKIP_BODY;
    else
        return EVAL_BODY_AGAIN;
}

public int doEndTag() throws JspException {
    System.out.println("doEndTag");
    try {
        JspWriter out = pageContext.getOut();
        out.println(number + "^" + power + "=" + result);
        this.counter = 0;
        this.result = 1;
    }
    catch (Exception e) {
    }
    return EVAL_PAGE;
}
}
```

Ejemplo IterationTag III

- JSP para evaluar el código:

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag number="2" power="3">.</easy:myTag>
```

- Fichero TLD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name>myTag</short-name>
    <tag>
        <name>myTag</name>
        <tag-class>es.deusto.customtags.PowerTag</tag-class>
        <body-content>tagdependent</body-content>
        <attribute>
            <name>number</name>
            <required>true</required>
            <rexprvalue>true</rexprvalue>
        </attribute>
        <attribute>
            <name>power</name>
            <required>true</required>
            <rexprvalue>true</rexprvalue>
        </attribute>
    </tag>
</taglib>
```

Manipulando el contenido de una etiqueta personalizada

- Una etiqueta personalizada puede tener un cuerpo:

```
<%@ taglib uri="/myTLD" prefix="x"%>
<x:theTag>This is the body
content</x:theTag>
```

- Para manipular el cuerpo de una etiqueta es necesario utilizar BodyTag y BodyContent

La interfaz BodyTag

- Extiende la `IterationTag` con 2 métodos
- Tiene un ciclo de vida similar a `IterationTag`, sin embargo:
 - `doStartTag` puede devolver `SKIP_BODY`, `EVAL_BODY_INCLUDE` (el cuerpo se evalua como con `IterationTag`) o `EVAL_BODY_BUFFERED` (un objeto de tipo `BodyContent` es creado al cuerpo de la etiqueta personalizada)
- `public void setBodyContent(BodyContent bodyContent)`
 - Llamado después de `doStartTag`, seguido de `doInitBody`, pero no se invoca si:
 - La custom tag no tiene cuerpo
 - La custom tag tiene cuerpo pero `doStartTag` devuelve `SKIP_BODY` o `EVAL_BODY_INCLUDE`
- `public void doInitBody() throws java.servlet.jsp.JspException`
 - No se invoca si se cumple alguna de las mismas condiciones que para `setBodyContent`

La clase BodyContent

- Representa el cuerpo de una etiqueta personalizada
- Ejemplo:
“Codificar el contenido HTML de una etiqueta personalizada y visualizar su contenido en el navegador”

Ejemplo de manipulación del cuerpo de custom tag I

```
package es.deusto.customtags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class EncoderTag
    implements BodyTag {
    PageContext pageContext;
    BodyContent bodyContent;
    /* Encode an HTML tag so it will be displayed as it is on the browser. Particularly, this method searches the passed in String and replace every occurrence of
     * the following character:
     * '<' with "&lt;"'
     * '>' with "&gt;"'
     * '&' with "&amp;"'
     * '\"' with "&quot;"'
     * ' ' with "&nbsp;"'
    private String encodeHtmlTag(String tag) {
        if (tag == null) return null;
        int length = tag.length();
        StringBuffer encodedTag = new StringBuffer(2 * length);
        for (int i = 0; i < length; i++) {
            char c = tag.charAt(i);
            if (c == '<')
                encodedTag.append("&lt;");'
            else if (c == '>')
                encodedTag.append("&gt;");'
            else if (c == '&')
                encodedTag.append("&amp;");'
            else if (c == '\"")
                encodedTag.append("&quot;");'
                //when trying to output text as tag's value as in
                // values="???".
            else if (c == ' ')
                encodedTag.append("&nbsp;");'
            else
                encodedTag.append(c);
        }
        return encodedTag.toString();
    }
}
```



Ejemplo de manipulación del cuerpo de custom tag II

```
public void setParent(Tag t) {  
}  
public void setPageContext(PageContext p) {  
    pageContext = p;  
}  
public void release() {  
}  
public Tag getParent() {  
    return null;  
}  
public int doStartTag() {  
    return EVAL_BODY_BUFFERED;  
}  
public void setBodyContent(BodyContent bodyContent) {  
    this.bodyContent = bodyContent;  
}  
public void doInitBody() {  
}  
public int doAfterBody() {  
    String content = bodyContent.getString();  
    try {  
        JspWriter out = bodyContent.getEnclosingWriter();  
        out.print(encodeHtmlTag(content));  
    }  
    catch (Exception e) {}  
  
    return SKIP_BODY;  
}  
public int doEndTag() throws JspException {  
    return EVAL_PAGE;  
}  
}
```

Ejemplo de manipulación del cuerpo de custom tag III

- Ejemplo JSP que usa EncoderTag:

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag><BR> means change line</easy:myTag>
```

- TLD file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag
Library 1.2//EN"
      "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name/>
  <tag>
    <name>myTag</name>
    <tag-class>es.deusto.customtags.EncoderTag</tag-class>
    <body-content>tagdependent</body-content>
  </tag>
</taglib>
```

Clases de ayuda

- Clases que implementan interfaces Tag, IterationTag y BodyTag:
 - `public class TagSupport implements IterationTag, java.io.Serializable`
 - `public class BodyTagSupport extends TagSupport implements Bodytag`
- Ahora sólo es necesario sobre-escribir los métodos que quieran utilizarse en el procesamiento de custom tags

Ejemplo con BodyTagSupport I

```
package es.deusto.customtags;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class CapitalizerTag extends BodyTagSupport {

    public int doAfterBody() {
        String content = bodyContent.getString();
        try{
            Jspwriter out = bodyContent.getEnclosingWriter();
            out.print(content.toUpperCase());
        }
        catch(Exception e) {}
        return SKIP_BODY;
    }

}
```

Ejemplo con BodyTagSupport II

- JSP que utiliza CapitalizerTag:

```
<%@ taglib uri="/myTLD" prefix="easy"%>
<easy:myTag>See the big picture?</easy:myTag>
```

- Fichero TLD:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP
    Tag Library 1.2//EN"
        "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name/>
    <tag>
        <name>myTag</name>
        <tag-class>es.deusto.customtags.CapitalizerTag</tag-
class>
        <body-content>tagdependent</body-content>
    </tag>
</taglib>
```

Etiquetas anidadas

- Cuando dos o más etiquetas personalizadas están anidadas es posible obtener una referencia a la clase padre a través de `findAncestorwithClass`
- Ejemplo:

```
OuterTag1 parent1 =  
(OuterTag1)findAncestorwithClass(th  
is, OuterTag.class);
```

Variables de script

- La clase Tag Extra Info (TEI) se usa para permitir la creación de variables de script.
 - Hay que definir en la clase `TagExtraInfo` el método `getVariableInfo`
 - Este método crea un array de objetos `VariableInfo`
 - Se creará uno de estos objetos por cada variable a definir, especificándose:
 - Nombre variable
 - Clase de la variable
 - Boolean indicando si habrá que crear una nueva variable
 - Scope de la variable:
 - `AT_BEGIN` → variable disponible en interior etiqueta y el resto del JSP
 - `NESTED` → variable disponible en el interior de la etiqueta
 - `AT_END` → variable disponible en el resto del JSP

Ejemplo: definir un iterador I

```
package es.deusto.customtags;

import java.util.Collection;
import java.util.Iterator;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class IteratorTag
    extends BodyTagSupport {

    private Collection collection;
    private Iterator iterator;
    private PageContext pageContext;

    public void setPageContext(PageContext p) {
        System.out.println("setPageContext");
        pageContext = p;
    }
    ...
}
```

Ejemplo: definir un iterador II

```
// the setter for number
public void setCollection(Collection collection) {
    this.collection = collection;
}

public int doStartTag() throws JspException {
    return collection.size() > 0 ? EVAL_BODY_BUFFERED : SKIP_BODY;
}

public void doInitBody() throws JspException {
    iterator = collection.iterator();
    this.pageContext.setAttribute("item", iterator.next());
}

...
```

Ejemplo: definir un iterador III

```
public int doAfterBody() throws JspException {  
    if (iterator == null) {  
        iterator = collection.iterator();  
    }  
    if (iterator.hasNext()) {  
        this.pageContext.setAttribute("item", iterator.next());  
        return EVAL_BODY_AGAIN;  
    }  
    else {  
        try {  
            getBodyContent().writeOut(getPreviousout());  
        }  
        catch (java.io.IOException e) {  
            throw new JspException(e.getMessage());  
        }  
        return SKIP_BODY;  
    }  
}  
public void release() {  
    collection = null;  
    iterator = null;  
}  
● }
```

Ejemplo: definir un iterador IV

- Definir un objeto TagExtraInfo

```
package es.deusto.customtags;
import java.util.Collection;
import java.util.Iterator;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class IteratorTagInfo extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
        {
            new VariableInfo("item",
                "java.lang.Object",
                true,
                VariableInfo.AT_BEGIN)
        };
    }
}
```

Ejemplo: definir un iterador V

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
 "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
    <tlib-version>1.0</tlib-version>
    <jsp-version>1.2</jsp-version>
    <short-name/>
    <tag>
        <name>iterate</name>
        <tag-class>es.deusto.customtags.IteratorTag</tag-class>
            <tei-class>es.deusto.customtags.IteratorTagInfo</tei-class>
        <body-content>JSP</body-content>
            <!--<variable>
                <name-given>item</name-given>
                <name-from-attribute>item</name-from-attribute>
                <variable-class>java.lang.String</variable-class>
                <declare>true</declare>
                <scope>AT_BEGIN</scope>
            </variable>
            <attribute>
                <name>id</name>
                <required>true</required>
                <rtpvalue>true</rtpvalue>
            </attribute>-->
            <attribute>
                <name>collection</name>
                <required>true</required>
                <rtpvalue>true</rtpvalue>
            </attribute>
        </tag>
    </taglib>
```

Ejemplo: definir un iterador VI

```
<html><head><title>An Iterator</title></head>

<%@ taglib uri="/myTLD" prefix="it"%>
<body>
<%
java.util.Vector vector = new java.util.Vector();
vector.addElement("one");
vector.addElement("two");
vector.addElement("three");
vector.addElement("four");
%>

Iterating over <%= vector %> ...
```

```
<p>
<it:iterate collection="<%=>vector%>">
    Item: <%= item %><br>
</it:iterate>
</p>
</body>
</html>
```

Custom Tags en JSP 2.0

- JSP 2.0 permite la declaración en un fichero TLD de funciones invocables desde cualquier JSP
- JSP 2.0 define Simple Tag Handler API, apropiada para etiquetas personalizadas donde el cuerpo sólo contenga:
 - Código de marcado
 - Expresiones EL
 - Elementos de acción
-

Invocando Funciones desde JSPs

- La EL de JSP 2.0 permite invocar a un método público de una clase usando la siguiente sintaxis:
 `${prefix:methodName(param1, param2, ...)}`
- La función JSP debe ser declarada en un tag library descriptor (TLD):

```
<function>
    <name>methodName</name>
    <function-class>className</function-class>
    <function-signature>
        returnType methodName(param1Type,
param2Type, ...)
    </function-signature>
</function>
```
- La clase Java que lo implementa no tiene que utilizar ninguna interfaz especial, el método debe ser público y estático.

Ejemplo Función JSP

- Revisar ejemplo en:
`examples\customtag\ej7_static_function_jsp`
- Copiar contenido a `webapps\myapp`
- Ejecutar:
`http://localhost:8080/ej7_static_function_jsp/TestFunction.jsp`

Ejemplo Simple Tag

- Una tag simple debe implementar la interfaz `javax.servlet.jsp.tagext.SimpleTag`
 - La clase `javax.servlet.jsp.tagext.SimpleTagSupport` implementa la interfaz
- Hay que añadir elementos setter por cada atributo e implementar `doTag`
 - Sólo se invoca un método no tres como antes: `doStartTag()` , `doAfterBody()` , `doEndTag()`
- **IMPORTANTE: los elementos de scripting no se permiten en el cuerpo del elemento siempre que la custom tag utilice un simple tag handler**
- Revisar ejemplo
`customtag\ej8_simpletag\ej8_simpletag`

Ejemplo Simple Tag

```
package com.mycompany.mylib;
import java.io.IOException;
import java.util.Iterator;
import java.util.Map;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class PollTag extends SimpleTagSupport {
    private String question;
    private Map answers;
    private String votesMapName;
    private String answersMapName;
    public void setQuestion(String question) {
        this.question = question;
    }
    public void setAnswers(Map answers) {
        this.answers = answers;
    }
    public void setVotesMapName(String votesMapName) {
        this.votesMapName = votesMapName;
    }
    public void setAnswersMapName(String answersMapName) {
        this.answersMapName = answersMapName;
    }
}
```

Ejemplo Simple Tag

```
public void doTag() throws JspException, IOException {
    Jspwriter out = getJspContext().getOut();
    JspFragment body = getJspBody();
    if (body != null) {
        out.println("<p>");
        body.invoke(null);
        out.println("</p>");
    }
    out.print("Question:");
    out.print(question);
    out.println("<br>");
    out.println("<form action=\"result.jsp\" target=\"result\">");
    out.print("<input type=\"hidden\" name=\"question\" value=\"\"");
    out.print(question);
    out.println(">");
    out.print("<input type=\"hidden\" name=\"votesMapName\" value=\"\"");
    out.print(votesMapName);
    out.println(">");
    out.print("<input type=\"hidden\" name=\"answersMapName\" value=\"\"");
    out.print(answersMapName);
    out.println(">");
    Iterator i = answers.keySet().iterator();
    while (i.hasNext()) {
        String key = (String) i.next();
        String value = (String) answers.get(key);
        out.print("<input type=\"radio\" name=\"vote\" value=\"\"");
        out.print(key);
        out.print(">");
        out.print(value);
        out.println("<br>");
    }
    out.println("<input type=\"submit\" value=\"Vote\">");
    out.println("</form>");
```

Ficheros Tag en JSP 2.0

- JSP 2.0 permite desarrollar una acción propietaria como un fichero de tag (tag file).
- Un tag file es un fichero de texto donde se utilizan elementos JSP para todas las partes dinámicas
 - Tiene la misma funcionalidad que un Tag Handler
- Se diferencia de un JSP en que:
 - Tiene extensión .tag
 - Usa una directiva tag en vez de page
 - Permite especificar entrada y salida con directivas válidas sólo en tag files.
-

Ejemplo de Fichero de Tag

```
<%@ tag body-content="empty" %>
<%@ attribute name="question" required="true" %>
<%@ attribute name="answers" required="true"
    type="java.util.Map" %>
<%@ attribute name="votesMapName" required="true" %>
<%@ attribute name="answersMapName" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

```
Question: ${question}<br>
<form action="result.jsp" target="result">
    <input type="hidden" name="question" value="${question}">
    <input type="hidden" name="votesMapName" value="${votesMapName}">
    <input type="hidden" name="answersMapName" value="${answersMapName}">
    <c:forEach items="${answers}" var="a">
        <input type="radio" name="vote" value="${a.key}">${a.value}<br>
    </c:forEach>
    <input type="submit" value="Vote">
</form>
```

Explicación Fichero Tag

- La directiva **tag** es similar a **page**
 - Declara características generales del fichero.
 - Su atributo **body-content** indica:
 - **empty** → el elemento XML representando la acción no tiene cuerpo
 - **scriptless** → el cuerpo no puede contener código Java
 - **tagdependent** → el contenedor pasa el cuerpo al evaluador de la tag
- La directiva **attribute** declara atributos válidos para la acción definida
 - Su atributo **type** indica el tipo de datos, por defecto **string**
- Los Map de respuestas y votos y las preguntas son codificados como campos hidden en el formulario, para así poderse transferir a la página procesando los votos

Página Procesando Votos

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <head>
        <title>Poll Results</title>
    </head>
    <body bgcolor="white">
        <c:set target="\${applicationScope[param.votesMapName]}" 
               property="\${param.vote}" 
               value="\${applicationScope[param.votesMapName][param.vote] + 1}" />
        <p>
            Question: \${param.question}<br>
            <c:forEach items="\${applicationScope[param.answersMapName]}" 
                       var="a">
                \${a.key}) \${a.value}:
                \${applicationScope[param.votesMapName][a.key]}<br>
            </c:forEach>
        </p>
    </body>
</html>
```

Utilizando el poll tag

```
<%@ page contentType="text/html" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="my" tagdir="/WEB-INF/tags/mytags" %>

<html>
  <head>
    <title>My Page</title>
  </head>
  <body bgcolor="white">
    <jsp:useBean id="myAnswers" scope="application"
      class="java.util.TreeMap">
      <c:set target="\${myAnswers}" property="1" value="Yes" />
      <c:set target="\${myAnswers}" property="2" value="No" />
      <c:set target="\${myAnswers}" property="3" value="Maybe" />
    </jsp:useBean>
    <jsp:useBean id="myVotes" scope="application"
      class="java.util.HashMap" />
    ...
    <p>
      <my:poll question="will you start using tag files?"
        answers="\${myAnswers}"
        answersMapName="myAnswers" votesMapName="myVotes" />
    </p>
    ...
  </body>
</html>
```

Ejercicios

1. Ejecutar los ficheros
 - o ejtag1_hola.jsp (Etiqueta sin atributos ni cuerpo)
 - o ejtag2_hola.jsp (Etiqueta con atributos)
 - o ejtag3_hola.jsp (etiqueta con atributos y cuerpo)
2. Comprobar el contenido del fichero web.xml
3. Comprobar el contenido del fichero etiquetas.tld
4. Hacer una etiqueta denominada suma, que efectúe la suma de los dos números pasados como parámetros

web

Así pues, la funcionalidad de una aplicación puede ser integrada de tres modos:

- Como código Java dentro de las páginas JSP
 - No separa la interfaz de la implementación
- Con el uso de JavaBeans llamados desde las páginas JSP
 - Separa la interfaz de la implementación en gran medida
- Con el uso de etiquetas personalizadas
 - Evitan la necesidad de inclusión de cualquier código Java en la página JSP
- Un buen diseño pasa por evitar en la medida de lo posible la primera de las tres opciones
-

Mantenimiento de sesiones

- Dado que el JSP se convierte en un servlet, todo lo visto en servlets es válido con JSP.
- Uso de Interfaz HttpSession para la gestión de sesiones
- El objeto “session” se crea automáticamente en JSP (puede suprimirse con page)
- El objeto “session” es un diccionario al que se pueden asociar objetos (atributos) y darles un nombre (también eliminar)
 - `session.setAttribute("mipaqute.sesion.nombre", miObjeto);`
- Se puede recuperar la información desde otras peticiones posteriores de la misma sesión
 - `Clase unObjeto = session.getAttribute("mipaqute.sesion.nombre");`
- También se puede eliminar un objeto asociado
 - `session.removeAttribute("mipaqute.sesion.nombre");`

Proceso de autenticación de usuarios

- Todo lo visto en servlets es aplicable en JSP
- Mecanismos
 - Declarativo. Usa un mecanismo proporcionado por el motor de servlets
 - Indicar restricciones de seguridad
 - Indicar el modo de realizar la autenticación
 - Indicar los usuarios definidos en el sistema
 - Por programa
 - Generar formulario html para nombre y clave
 - Generar página JSP que accede a Base de Datos, comprueba clave y se fijan los parámetros de seguridad en el JavaBean, cuyo ámbito se restringe a la sesión que acaba de establecer el usuario

XML y JSP trabajando juntos

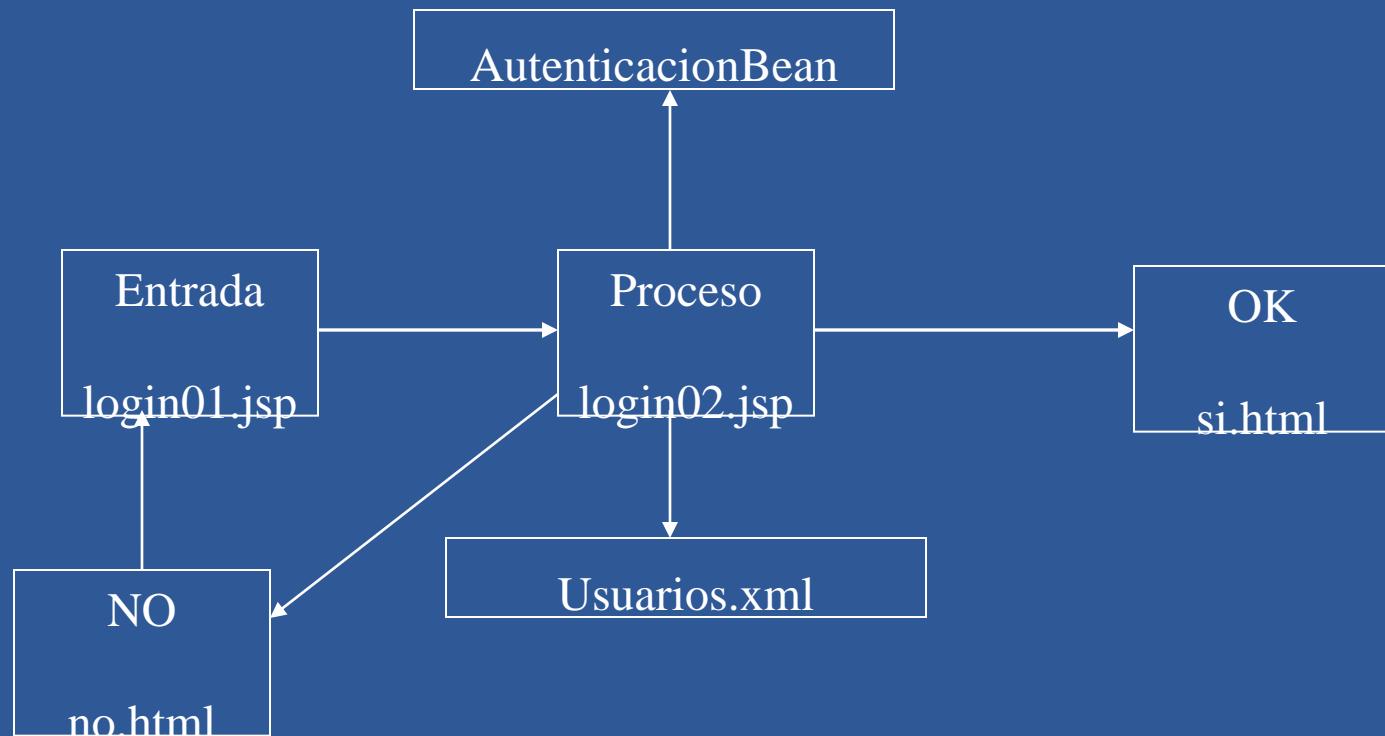
- Para utilizar un documento XML en una aplicación es preciso analizarlo
- Existen dos modelos de analizadores aceptados:
 - DOM (Modelo de objetos de documento)
 - SAX (API simple para XML)
- Analizador de XML tipo DOM
 - El analizador crea una estructura de datos en memoria en forma de árbol, donde se coloca todos los elementos del documento
 - Ofrece métodos para acceso a los nodos y a la información de los mismos a través de una API
- Analizador SAX
 - En lugar de crear un árbol, va leyendo el archivo XML y ejecuta acciones según las etiquetas encontradas.

XML y JSP trabajando juntos

- Si se desea usar un documento XML en una aplicación JSP habrá que
 - Incluir el código Java para la manipulación del documento XML
 - o bien incluir un JavaBean que lo manipule
 - o bien definir una etiqueta personalizada que envuelva la operación dentro de su clase
- En cualquier caso se trata de manejar XML con Java

Ej. de análisis XML con DOM en una página JSP

- Veamos cómo realizar la autenticación de usuarios en base a un fichero XML con los datos de autenticación



Ej. de análisis XML con DOM en una página JSP

Supongamos el siguiente contenido de la página
xml de usuarios autorizados

```
<?xml version='1.0' encoding='iso-8859'?>
<usuariosAutorizados>
    <usuario nombre="Agustín" pwd="agustin"
        nivel="administrador">
    </usuario>
    <usuario nombre="Invitado" pwd="invitado"
        nivel="usuario">
    </usuario>
</usuariosAutorizados>
```

Ej. de análisis XML con DOM en una página JSP

Formulario de entrada (login01.jsp)

```
<form method="POST" accion="login02.jsp"
      name="autenticacion">
  <table border="0" cellpadding="0" cellspacing="0"
        width="200">
    <tr>
      <td width="50%">Nombre</td>
      <td width="50%"><input type="text" name="nombre"
          size="16"></td>
    </tr>
    <tr>
      <td width="50%">Contraseña</td>
      <td width="50%"><input type="password" name="pwd"
          size="16"></td>
    </tr>
    <tr>
      <td width="50%" colspan="2" align="center">
        <input type="submit" value="Entrar" size="16">
      </td>
    </tr> </table> </form>
```

Ej. de análisis XML con DOM en una página JSP

Autenticación (login02.jsp)

- Importación de los paquetes para el control del fichero XML

```
<%@ page import="javax.xml.parsers.*" %>
```

- Definición de las variables para información de entorno

```
<%! String usuario=""; %>
```

```
<%! String pwd=""; %>
```

```
<%! String redireccionURL =""; %>
```

- Instanciación del JavaBean responsable de mantener información del usuario mientras dure la sesión

```
<jsp:useBean id="login" scope="session"
```

- class="login.Autenticacion"/>

Ej. de análisis XML con DOM en una página JSP

- Recuperación de los datos de identificación del usuario enviados desde el formulario HTML de la página previa

```
if (request.getParameter("nombre") != null)
    {usuario=request.getParameter("nombre");}
if (request.getParameter("pwd") !=null)
    { pwd= request.getParameter("pwd"); }
```

- Inicialización de variables para el procesamiento del documento XML

```
Document documento;
DocumentBuilderFactory.newInstance();
redireccionURL="no.html";
```

Ej. de análisis XML con DOM en una página JSP

- Generación del conjunto de nodos siguiendo el modelo DOM

```
// Apertura del archivo
```

```
URL url=new URL(archivoXml);
```

```
InputStream datosXML=url.openStream();
```

```
// Construcción del documento XML
```

```
DocumentBuilder
```

```
    builder=factory.newDocumentBuilder();
```

```
documento=builder.parse(datosXML);
```

```
//Generación de lista de nodos en base a la clave “usuario”
```

```
NodeList
```

```
listaNodos=documento.getElementsByTagName("usuario");
```

Ej. de análisis XML con DOM en una página JSP

- Comparación de los atributos introducidos con los existentes en el árbol

```
for (int i=0; i<listaNodos.getLength();i++) {  
    Node actNodo=listaNodos.item(i);  
    // para este nodo recogemos el valor del “nombre” y  
    // contraseña  
    Element actElemento=(Element)listaNodos.item(i);  
    String actUsuario=actElemento.getAttribute("nombre");  
    String actPwd=actElemento.getAttribute("pwd");  
    // si el usuario es correcto le dejamos que siga  
    if (actUsuario.equals(usuario) && actPwd.equals(pwd)) {  
        redireccionURL="si.html";  
    }  
}
```

Ej. de análisis XML con DOM en una página JSP

//Actualizamos el bean para indicar que el usuario ya está autorizado

```
logon.setNombre(usuario);  
logon.setAutorizado();  
break bucle();  
}  
}
```

- Código JavaScript que permite el reenvío de una página a otra, de forma que se pueda presentar al visitante la página adecuada dependiendo de que haya sido posible la autenticación o no

```
<script language="javascript">  
setTimeout ("document.location='<%= redireccionURL  
%>'",100)  
● </script>
```

Ej. de análisis XML con DOM en una página JSP

- Control de la sesión
 - El JavaBean Autenticacion.java lleva a cabo el control
 - Es posible comprobar en cada página web si el usuario está autorizado llamando al JavaBean
 - Los métodos set de las propiedades nombre y clave se invocan desde las páginas de control de la lógica de aplicación (login02.jsp) una vez que el usuario ha sido validado
 - Los métodos get devuelven objetos String con la identificación y la autorización del usuario

Ej. de análisis XML con DOM en una página JSP

```
package login;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class Autenticacion {
    // Propiedad "nombre" del bean, que corresponde al
    // nombre
    // del usuario para el que se implementa el bean
    String nombre = "";
    // Variable de clase que indica si el usuario para el
    // que se crea el bean en la sesión que ha
    establecido
    // con el servidor está identificado ante el sistema
    y se
    // le concede autorización
    boolean autorizado = false;
```

Ej. de análisis XML con DOM en una página JSP

```
// Métodos get() y set() de la propiedad "nombre"
public String getNombre() {
    return( nombre );
}
public void setNombre( String _nombre ) {
    nombre = _nombre;
}
// Métodos get() y set() de la variable de clase que
// controla
// la autenticación del usuario ante el sistema
public void setAutorizado() {
    autorizado = true;
}
public boolean getAutorizado() {
    return( autorizado );    }    }
```

Qué se ha visto en esta sesión

1. Primer vistazo a la tecnología JSP
 - Qué es y para qué sirve JSP
 - Primer ejemplo *Hola Mundo* y la fecha
 - Uso de objetos implícitos y ejemplo
2. Elementos básicos de una página JSP
 - **Directivas** <%@ (page | include | taglib)
 - **Código Java** <% (declaraciones | java | expresiones)
 - **Acciones estándar** <jsp:acción (include | forward | usebean | getProperty | setProperty)
3. Elementos Avanzados de una página JSP
 - **Acciones personalizadas** (etiquetas)
 - Definición de la estructura (biblioteca.tld y ésta en web.xml)
 - **Nombre, clase controladora**, atributos, cuerpo, etc.
 - Definición de la funcionalidad (JavaBean que deriva de taglib)

Principales conclusiones de diseño

1. La tecnología JSP permite diseño de aplicaciones web con generación dinámico de páginas web tras un procesamiento
2. La lógica de la aplicación (procesamiento) se invoca desde una página html mediante tres posibles modos
 1. Inclusión de código Java
 2. Invocación de objetos (JavaBeans) que implementen la lógica
 3. Invocación de etiquetas personalizadas que implementen la lógica
3. Para realizar un buen diseño que separe interfaz de lógica no se debe incluir directamente el código Java. De este modo están separadas las funciones:
 - Diseñador de interfaces (JSP sin código Java -> html y marcas)
 - Programador de la lógica de la aplicación (JavaBeans que implementan la funcionalidad)
4. Una página JSP se transforma automáticamente en un servlet. La ventaja es que el diseño de la interfaz no se mezcla con la programación en Java

Java Server Pages (JSP)

Fin de la sesión