Unidad III – Programación Distribuida

Autor: Ing. Tymoschuk, Jorge
Colaboración: Ing. Polliotto, Martin

Breve historia de Java y la Red

En los años sesenta, los ordenadores comenzaron a proliferar, los Estados Unidos construyeron una red militar de ordenadores llamada ARPANET, que unía ordenadores clave por todo el país.
La red estaba descentralizada, y por lo tanto, era inmune a un ataque a gran escala a los Estados Unidos.

El buen diseño de la arquitectura de la red aseguró su supervivencia durante muchos años y la comunidad académica pronto superó su uso militar. Pasó a ser principalmente una herramienta educativa y fue rebautizada como Internet.

Los primeros días de Internet no fueron para el gran público. Poca gente era consciente de que Internet existía, menos todavía tenía acceso a ella. Estos primeros usuarios tuvieron que aprender muchas utilidades basadas en la línea de comandos como Telnet, FTP y Gopher para poder hacer algo útil. La sencillez de uso no era una preocupación.

En 1989, Tim Berners-Lee, un científico informático que trabajaba para la Organización europea para la investigación nuclear (CERN), inventó la World Wide Web(la red). Berners-Lee pretendía crear un sistema de hipertexto interactivo sobre la ya existente Internet que facilitaría, en gran medida, la comunicación entre la comunidad mundial de físicos. Hipertexto se refiere a cualquier sistema en el que las palabras funcionan como vínculos a otros documentos o secciones de un documento.

La red comenzó a adquirir importancia y en 1993 se componía de aproximadamente 50 servidores Web. En este momento, ocurrió un acontecimiento que encendería la mecha de la explosión de Internet; el centro nacional para aplicaciones de supercomputadoras (NCSA) en la universidad de Illinois hizo pública la primera versión de navegador Web de Mosaic para los sistemas Unix, PC y Macintosh.

Con los cimientos de Mosaic echados, el año 1994 fue testigo de la emergencia de la Red en la cultura popular. La Red se convirtió en la "aplicación estrella" que animó al público en general a explorar Internet por ellos mismos. Ese mismo año, algunas de las personas que habían creado Mosaic, fundaron una pequeña compañía en Silicon Valley, que finalmente se convertiría en Netscape.

Cómo funciona la Red

Puede haber cierta confusión en cuanto a qué es exactamente Internet y en qué se diferencia de la Red. Internet es la red de ordenadores física que une ordenadores de todo el mundo. La Red, por otro lado, es un servicio que se asienta sobre los fundamentos de Internet. La Red permite que los ordenadores interconectados hagan cosas útiles. La Red es uno de los muchos servicios que utilizan Internet; otros son, por ejemplo, el correo electrónico, los contenidos multimedia y los juegos multijugador.

Como servicio, la Red define cómo dos partes, un cliente Web (o navegador Web) y un servidor Web que utilizan Internet para comunicarse. Cuando se visita un sitio Web, se está creando una relación entre estas dos partes. En dicha relación, las dos partes se comunican enviándose una serie de mensajes breves. En primer lugar, el navegador Web envía un mensaje al servidor Web pidiendo una determinada página Web que desea recibir y el servidor Web le responde con un mensaje apropiado. Por cada página adicional que se visualiza, el navegador Web envía otra petición al servidor Web que, de la misma manera, responde con mensajes adecuados.

Este tipo de relación se denomina modelo de petición/respuesta. El cliente, en este caso el navegador Web, pide un recurso específico (por ejemplo, una página Web) y entonces el servidor responde con el recurso requerido, si está disponible. La Red se basa en este modelo de petición/respuesta, que está

implementado utilizando el protocolo de transferencia de hipertextos (HTTP). Un "protocolo" en el sentido de interconexión es una definición de cómo un dispositivo o programa se comunica con otro. HTTP es un protocolo de red que define cómo se comunican un cliente Web y un servidor Web.

Lo que hay que recordar, por su importancia, es que la petición que el cliente envía al servidor se conoce como petición HTTP y la respuesta enviada por el servidor al cliente se denomina respuesta HTTP.

Common Gateway Interface (CGI) (Interfaz Común de Pasarela)

Cuando navega por Internet, seguro que va a encontrarse tanto con páginas estáticas, que devuelven el mismo documento a todos los usuarios, como con páginas generadas sólo para Usted. Aunque se puede obtener una buena cantidad de uso con la presentación de archivos estáticos, la emoción y utilidad reales vienen al crear documentos HTML dinámicos.

El CGI proporcionó el mecanismo original a través del cual los usuarios podían realmente ejecutar programas en servidores Web y no simplemente pedir páginas HTML. Bajo el modelo CGI:

1. El navegador Web envía una petición de la misma forma que lo haría si se tratase de una página HTML.

2. El servidor Web ejecuta un programa externo, pasándole la petición HTTP que recibió del navegador, recibe su respuesta y la devuelve al navegador, como si se tratara de una respuesta HTTP.

CGI fue enormemente popular en los primeros momentos de la Red como el método estándar de generar páginas Web de forma instantánea (se dice que dichas páginas son dinámicas). Casi cualquier lenguaje de programación imaginable ha sido utilizado para implementar algún tipo de solución basada en CGI; Perl ha sido un lenguaje especialmente importante para el desarrollo de CGI.

Sin embargo, cuando las demandas de tráfico situadas en sitios Web aumentaron, CGI no era lo suficientemente eficiente. Esto es debido a que con CGI, cada vez que se recibe una petición, el servidor Web tiene que empezar a ejecutar una nueva copia del programa externo. Si es sólo un pequeño número de usuarios los que piden un programa CGI a la vez, no hay problema.

Pero éste no es el caso si cientos o miles de usuarios piden el recurso al mismo tiempo. Imagine el servidor Web tratando de lanzar una nueva copia de un programa CGI para cada una de esas peticiones. Los recursos del servidor se verían rápidamente agotados. Peor todavía si consideramos que los programas CGI escritos en intérpretes de lenguaje como Perl requerirían de intérpretes.

Alternativas a CGI

Con el paso de los años, han aparecido muchas soluciones alternativas a CGI. Los sustitutos de CGI que han tenido éxito proporcionan un entorno que vive dentro de un servidor Web existente o que incluso funcionan como servidores Web por sí mismos. .

Muchos de dichos sustitutos se han construido sobre la base del popular servidor Web de código abierto Apache (http://WWW.apache.org/). Esto es debido al popular módulo API de Apache, que permite a los desarrolladores ampliar la funcionalidad de Apache con los programas persistentes. Los módulos se cargan en la memoria cuando Apache se pone en marcha y Apache pasa las peticiones HTTP adecuadas a aquellos módulos residentes en la memoria y transfieren las respuestas HTTP otra vez al navegador.
Esto significa que el tiempo de carga de un intérprete en la memoria desaparece y los scripts pueden comenzar a ejecutarse mucho antes.

Existen muchos módulos creados por terceros que proporcionan una base sobre la que los desarrolladores crean aplicaciones que son mucho más eficientes que las CGI normales. Algunos ejemplos:

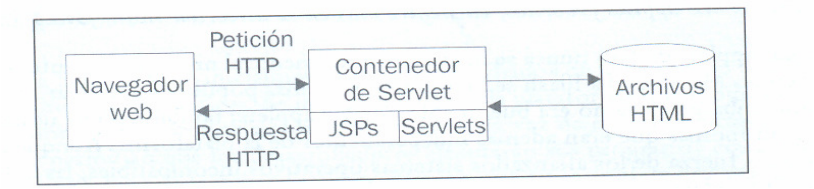
- mod _per1, Mantiene el intérprete Perl en la memoria.
- mod _php4, idem para PHP.
- mod_fastcgi, parecido al CGI, pero los programas permanezcan residentes en la memoria.

Java y la Red

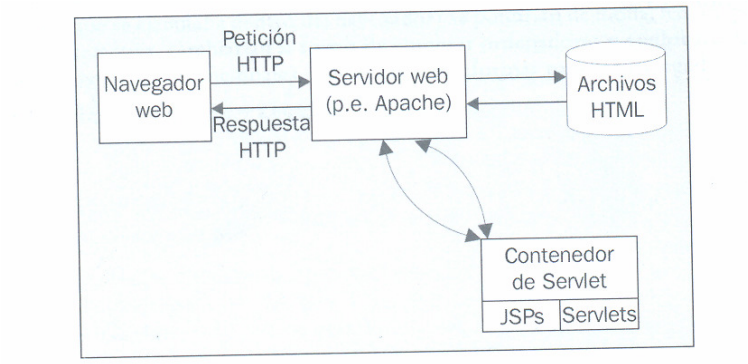
Java fue inicialmente lanzado a mediados de los 90 como una forma de animar páginas Web aburridas y estáticas, con sus applets("mini-aplicación" que se ejecutaba dentro del navegador). Java es independiente de la plataforma y permitían que los desarrolladores ejecutaran sus programas correctamente en el navegador Web.

Los novedosos applets de Java nunca se impusieron realmente al nivel que la gente predijo. Otras tecnologías como Macromedia Flash se hicieron mucho más populares por crear sitios Web realmente elegantes. Sin embargo, Java no era bueno sólo por los applets; también podía utilizarse para crear aplicaciones autónomas que eran además independientes de la plataforma. Y aquí aparecen los Servlets de Java. Los Servlets (un "mini-servidor") son una tecnología alternativa a CGI. Los Servlets, por ellos mismos, no son aplicaciones autónomas; se cargan en la memoria mediante un contenedor de servlet.

Este contenedor de servlet funciona entonces como un servidor Web, recibiendo peticiones HTTP de los navegadores Web y pasándolas a los servlets.



Otra posibilidad es que el contenedor de servlets puede integrarse en un servidor Web existente (por ejemplo, Apache integra Tomcat, contenedor de servlet).



La simplicidad del lenguaje de programación Java, su naturaleza independiente de la plataforma, el código abierto de Sun y la actitud hacia Java de la comunidad y la elegancia del propio modelo servlet han hecho de los Servlets de Java una solución inmensamente popular para crear contenido Web dinámico.

Páginas de Servidor Java (JSP, Java Server Pages)

Para hacer la creación de contenido Web dinámico todavía más fácil, Java ha introducido las Páginas de Servidor Java (también llamadas páginas JSP).

Mientras que los Servlets pueden requerir un conocimiento bastante extenso de Java para su creación, un novato en Java puede aprender cómo hacer cosas bastante claras con JSP rápidamente. Las páginas JSP fueron diseñadas también para facilitar a los usuarios de la tecnología de Páginas de Servidor Activas de Microsoft (ASP) su migración a Java.

La tecnología JSP está realmente construida sobre la base de los servlets; como veremos posteriormente en el libro, las dos tecnologías trabajan de hecho bien juntas. Es normal utilizar las dos en la misma aplicación Web.

JavaScript

Es una tecnología que permite a las páginas Web tener alguna funcionalidad de programación en el navegador. Mientras los applets de Java son aplicaciones aisladas que simplemente se muestran en la página Web, JavaScript trabaja con la página HTML y puede manipularla.

JavaScript no es Java; es un lenguaje de programación completamente distinto que fue desarrollado aproximadamente al mismo tiempo que se dio a conocer Java.

Para comprender mejor la distinción entre JavaScript y JSP, podría ayudarle recordar que el código JavaScript se ejecuta generalmente por el cliente Web (navegador) después de que el servidor Web haya enviado la respuesta HTTP al navegador y JSP es ejecutado por el servidor Web antes de que éste envíe la respuesta HTTP. Así, se dice que JavaScript es una tecnología aplicada al cliente y su código subyacente puede ser visto (y copiado) por usuarios Web, mientras que JSP es una tecnología del lado del servidor y su código subyacente no está expuesto a los usuarios Web; es procesado por el servidor Web antes de que llegue al cliente.

Elementos scriplet

- **Comentarios**

Los comentarios están diseñados para ayudarle a hacer que su JSP sea más sencilla de comprender. El contenido de un elemento comentario no se transfiere al cliente; los comentarios se desprenden de la página antes de su transmisión. Si desea que sus comentarios sean visibles en la imagen enviada al cliente necesita colocarlos como comentarios HTML.

Los comentarios JSP tienen la siguiente sintaxis:

```
<%-- Esto es un comentario de línea --%>
```

También puede escribir comentarios como en el código Java, de línea y párrafo.

```
<% // Este tipo de comentario Java de línea %>

<%
      /* Este tipo de comentario Java comienza con una barra inclinada y un
      asterisco y continúa en tantas líneas como desee hasta un asterisco y una
      barra inclinada. */
%>
```

- **Declaraciones**

Las declaraciones se utilizan para definir variables y métodos que se utilizarán en scriplets y expresiones posteriores o en otras declaraciones. La sintaxis de una declaración es:

```
<%! JavaDeclaration %>
```

donde JavaDeclaration es una o más definiciones de enunciados o métodos declarativos de Java, por ejemplo:

```
<% ! int contador = O; %>


<%! double calculoInteres(double loan, double interestRate, int numYears){
      double interest = numYears * (interestRate * loan/100);
      return interest;
    )
%>
```

- **Expresiones**

Los elementos expresión son una evaluación de una expresión de Java; la sintaxis general es:

```
<%= JavaExpression %>
```

donde JavaExpression es una expresión de Java como las que ya hemos visto. El resultado de la expresión será incluido después en el resultado de la página. Por ejemplo, si tuviéramos un String llamado helloString que contuviera la cadena literal "Hello World" e incluyéramos la expresión:

```
<%= helloString %>
```

en una página JSP, entonces Hello World sería incluido en la salida de la página. Nota: no se incluye punto y coma en los elementos expresión.

- **Scriptlets**

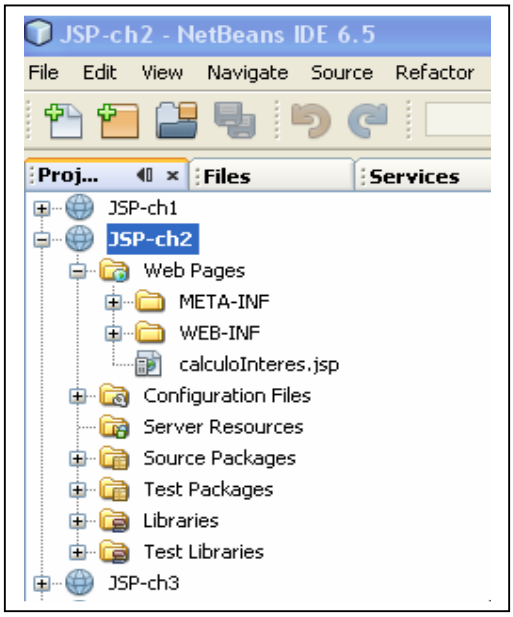
Puede incluir cualquier código Java dentro de elementos scriplet. Su sintaxis:

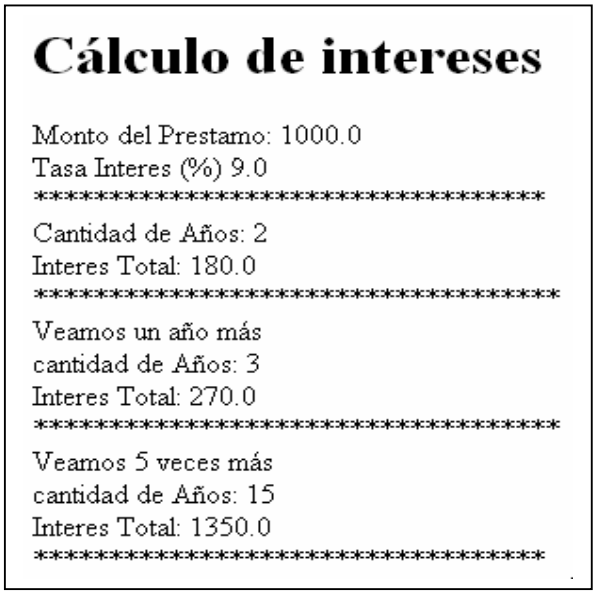
```
<% JavaCode %>
```

Veamos cómo podemos utilizar estos distintos elementos de escritura en una página JSP que calcula el pago de un préstamo.

```
<%--
    Documento   : Calculo del interés
    Creado : 30-ene-2009, 16:09:53
    Autor     : Tymos
--%>

<%-- Abajo declaramos variables --%>
<%! double montoPrestamo; %>
<%! double tasaInteres; %>
<%! int cantAños; %>

<%-- DECLARACION DE METODOS --%>
<%!     //  A continuación la declaración de un método
        double calculoInteres(double tasa,
                              double montoPrestamo,
                              int cantAños){
            return cantAños * montoPrestamo * tasa / 100;
        }
%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Cálculo de interes</title>
    </head>
```

```
    <body>
         <h1> Cálculo de intereses </h1>
        <%  //  Aquí usamos un scriplet conteniendo
            //  varias expresiones Java
            montoPrestamo = 1000;
            tasaInteres = 9;
            cantAños = 2;
        %>

        Monto del Prestamo: <%= montoPrestamo %>
        <br />
          Tasa Interes (%) <%= tasaInteres %>
        <br />
        **********************************
        <br />
          Cantidad de Años: <%= cantAños %>
        <br />
          Interes Total: <%= calculoInteres(montoPrestamo,tasaInteres,
                                                                cantAños)%>
       <br />
       **********************************
        <br />
            Veamos un año más <%cantAños++; %>
        <br />
        cantidad de Años: <%= cantAños %>
        <br />
        Interes Total: <%= calculoInteres(montoPrestamo,tasaInteres,
                                                              cantAños)%>
       <br />
       **********************************
        <br />
            Veamos 5 veces más <%cantAños*=5; %>
        <br />
        cantidad de Años: <%= cantAños %>
        <br />
        Interes Total: <%= calculoInteres(montoPrestamo,tasaInteres,
                                                            cantAños)%>
       <br />
       **********************************
    </body>
</html>     //    El proyecto es:   [calculoInteres.jsp] <Run file>
```





Obtención de datos desde el navegador

Ya hemos visto como utilizar scriptlets para insertar fragmentos del código lava dentro de JSP, cómo declarar métodos y variables, cómo devolver datos al navegador. Pero devolver datos al navegador desde el servidor representa sólo una mitad en el desarrollo de una aplicación Web. Para hacer que una aplicación Web sea más interactiva, los usuarios de esa aplicación deberían ser capaces de introducir datos en su navegador y enviarlos para un mayor procesamiento.

Comenzaremos por crear aplicaciones Web interactivas, en las que el cliente introduce datos. Luego hablaremos de las técnicas que se pueden utilizar para pasar estos datos al servidor, de manera que puedan ser leídos y procesados.

Presentaremos y utilizaremos varios controles de formulario HTML, entre los que se incluyen:

- Cuadros de texto y áreas de texto
- Botones de activación.
- Casillas de verificación.
- Control de selección (lista desplegables y lista multiselección).
- Botones de envío y de reinicio.

Los datos del cliente enviados al servidor son atributos del objeto request; Este objeto se instancia automáticamente, Java lo hace por nosotros. Lo que si deberemos es, en el servidor, extraer estos datos para su procesamiento.

Recuperación de los datos del cliente utilizando formularios HTML

Existe más de una forma de capturar datos del cliente y enviarlos al servidor. Puede hacerse desde una aplicación Java, (Clase que extiende de JFrame),también desde un Applet…
Pero el método normalmente empleado para realizar esta tarea consiste en pedirle al usuario que rellene un formulario HTML. La información proporcionada por el usuario puede ser textual (por ejemplo, el nombre del usuario), o puede ser una selección de un número de opciones ("elija su destino de vacaciones"). Una vez terminado el formulario, deberá hacer clic en el botón enviar para iniciar la petición (la transferencia de los datos del cliente al servidor).

HTML le proporciona una variedad de controles de formulario distintos que puede presentar al usuario.

Utilización del elemento <form>

Todos los formularios dentro de una página HTML están incluidos dentro de etiquetas de elemento de inicio <form> y de cierre </form>. Sin embargo, el **elemento <form>** por sí mismo no presenta ningún control visual en la pantalla de navegador. Colocamos controles específicos de formulario dentro de **<form>** y entonces estos controles pasan a formar parte del formulario. Son los controles presentados en el navegador para que el cliente interactúe con ellos.

La etiqueta <form> contiene atributos, que son entornos de configuración para el formulario. Por ejemplo, hay más de un método que podemos utilizar para enviar los datos del cliente al servidor, y debemos elegir el más adecuado. Además, el formulario necesita saber a qué página dirigirse una vez que el formulario ha sido enviado. A continuación, analicemos más detenidamente los atributos de la etiqueta <form>.

El atributo action

El atributo <action> se utiliza para especificar la página a la que debe dirigirse cuando el formulario es enviado. Necesitamos especificar el recurso en el servidor (por ejemplo, una página JSP) que se encarga del procesamiento de los datos del formulario enviados por el usuario. A continuación, este recurso

del servidor puede generar una página HTML que se envía de nuevo al cliente como respuesta a su petición.

El nombre del atributo de formulario utilizado para especificar esta fuente es action y su valor se puede especificar de diversas formas. Primero, como una URL absoluta, tal como se muestra a continuación:

```
<form action="http://myServer.com/process.jsp"></form>
```

Otra posibilidad es que la URL puede especificarse de forma relativa a la página que contiene el formulario:

```
<form action="process.jsp"></form>
```

En este caso, si la HTML actual es generada por: //myServer.com/input.jsp, la petición se envía al recurso http://myServer.com/process.jsp. En otras palabras, la página a la que nos dirigimos está en el mismo directorio Web que la página que contiene el formulario.

Finalmente, si un formulario es enviado sin un atributo action, la petición es enviada de nuevo a la página que contiene el formulario. .

El atributo name

Si vamos a transmitir datos, necesitamos también etiquetarlos, ya que de lo contrario no sabremos de qué formulario o de qué control de formulario proceden los datos. De esta forma, el atributo name se utiliza para identificar un formulario o control de formulario en particular dentro de la página HTML.
Veremos ejemplos de este atributo más adelante.

El atributo method

De igual forma que podemos elegir dónde enviar los datos del formulario, también podemos elegir la manera de enviarlos. El "cómo" se especifica mediante el valor del atributo method. Aunque potencialmente podríamos enviar los datos a cualquier página, existen sólo dos métodos comúnmente utilizados entre los que puede elegir para enviar los datos:

- GET
- POST

Especificaremos cual de ellos utilizamos de la siguiente forma:

```
<form action="process.jsp" method="post"></form>
```

Aquí estamos enviando datos de formulario a la página process.jsp utilizando el método post. El valor de method por defecto es get.

Observemos más detenidamente cómo funcionan estos métodos.

Utilización de Get y Post

Cuando el usuario envía los datos del formulario, el navegador recopila los datos introducidos o seleccionados por el usuario en pares de nombre/valor. El nombre en este par es el valor del atributo name del control, mientras que el valor denota el valor de los datos. Por ejemplo, supongamos que tenemos un cuadro de texto llamado nombre de usuario para que el usuario introduzca su nombre. El par de nombre/valor para un usuario "Anna Karenina" sería:

```
?userName = Anna+Karenina
```

Observe cómo el par de nombre/valor está precedido por el símbolo? y el espacio en blanco en el medio del valor del nombre ha sido reemplazado por el carácter +. Esto nos lleva a la pregunta: ¿qué ocurre cuando escribimos un carácter + en

el cuadro de texto? La respuesta es que algunos caracteres son reemplazados por un código especial. Esto se conoce como codificación URL.

Ahora, si elegimos utilizar el método GET para transferir los datos al servidor, este par nombre/valor se transmite al servidor poniéndolo como apéndice al final de la URL de la página destino a la que estamos enviando el formulario. Por ejemplo:

Aquí, la página de destino para los datos del formulario es process.jsp. Observe que podemos incluir varios pares nombre/valor en esta cadena de búsqueda uniéndolos en la URL con un carácter&:

process.jsp?**userName**=Anna+Karenina&**userGender**=Male

Aquí tenemos dos campos de entrada en el formulario denominados userName y userGender en los que el usuario ha introducido los datos Anna+Karenina y Male.

Una posible desventaja de esta técnica es la colocación de los pares nombre/valor en la cadena de caracteres de la URL es una forma bastante pública de pasar información. ¿Qué ocurre si tenemos datos que son personales o privados y no queremos que sean visibles para que todo el mundo los vea? Aquí es donde entra el método POST. Cuando especificamos qué datos deberían ser transferidos utilizando POST, los pares nombre/valor se envían, en este caso, dentro del cuerpo de la petición HTTP.

Entonces, ¿cuándo deberíamos utilizar GET y cuándo POST? Existen distintas opiniones al respecto:

- algunos sostienen que se debería usar siempre POST debido a la naturaleza pública de las peticiones GET.
- Sin embargo, GET también tiene ventajas: por ejemplo, las páginas que se cargan utilizando POST no pueden ser agregadas a la lista de favoritos correctamente, mientras que las cargadas con GET (por ejemplo, el resultado de un formulario enviado en Alta Vista) resultan fáciles de agregar.

El atributo target

Hemos observado previamente que el recurso de servidor identificado en el atributo action podría generar una página HTML cuando el formulario es enviado. El atributo target puede utilizarse para identificar un cuadro o ventana dentro del navegador al que debería enviarse a la página HTML resultante, si es distinta del cuadro o ventana que contiene el formulario. Este atributo puede tomar los siguientes valores.

- _blank: La página HTML resultante se envía a una nueva ventana.
- _parent: La página HTML resultante se envía al cuadro en el que estamos trabajando.
- _self: La página HTML resultante sobreescribe la página actual. Éste es el valor que se usa por defecto cuando no se especifica uno distinto.
- _top: La página HTML resultante ocupará toda la ventana del navegador, ignorando todas los barras de navegación configuradas.

Ya sabemos crear un formulario, veamos cómo rellenarlo con controles.

Utilización de controles HTML

Los controles HTML se utilizan para definir los componentes de una interfaz gráfica: cuadros, listados y botones que podrán ser utilizados por los usuarios para introducir los datos o eventos que quieren enviar al servidor. Los controles HTML que pueden utilizarse dentro de un formulario pueden definirse utilizando los siguientes elementos.

- <input>, utilizado para presentar controles como campos de texto, botones de activación y casillas de verificación.

- <select>, utilizado para presentar cuadros de listas multiselección, y cuadros de listas desplegables.
- <textarea>, utilizado para presentar controles de edición multilínea.

El elemento HTML <input>

Engloba los controles utilizados para crear cuadros de texto, botones de activación y casillas de verificación dentro de los formularios HTML. Su sintaxis es:

<input [lista de atributos]>

Uno de los atributos más importantes es el de **tipo** de input.

- **TEXT** Se utiliza para presentar campos de texto. Éste es el valor por defecto si no se especifica el atributo.
- **PASSWORD** Se utiliza para presentar controles de contraseña.
- **HIDDEN** Se utiliza para definir los controles ocultos. Los elementos de formulario ocultos se utilizan para almacenar información en la página que no tiene que mostrarse a los usuarios. Son típicamente valores de almacenamiento utilizados por la lógica de procesamiento del servidor frente a peticiones múltiples.
- **CHECKBOX** Se utilizan para presentar casillas de verificación.
- **RADIO** Se utiliza para presentar botones de activación.
- **RESET** Se utiliza para presentar un botón de control usado para restablecer los contenidos del formulario a sus valores originales por defecto.
- **SUBMIT** Se utiliza para presentar un botón de control usado para enviar el formulario.
- **IMAGE** Se utiliza para presentar imágenes.
- **BUTTON** Utilizado para presentar un botón de control que puede vincularse con el script del lado del cliente.
- **FILE** Utilizado para presentar archivo de control que puede ser utilizado para ver y seleccionar archivos del sistema de archivos local para que sea cargado por el servidor.

Para crear un cuadro de texto:

```
<input type="TEXT">
```

El atributo name

Después de haber elegido el tipo de control que necesitamos, debemos asignarle un nombre.

```
<input type="text" name="direccion">
```

Aquí hemos creado un cuadro de texto llamado dirección. Esta información es enviada al servidor como un par **nombre/valor**, el nombre será direccion y el valor será lo que el usuario haya introducido en el cuadro de texto antes de enviar el formulario.

El atributo maxlength

Se utiliza para tipos text o password. Indica el número máximo de caracteres que pueden ser introducidos en el control.

```
<input type="text" name="direccion" maxlength="30">
```

El atributo size

Se utiliza para tipos text o password. Define la anchura visible del control en número de caracteres.

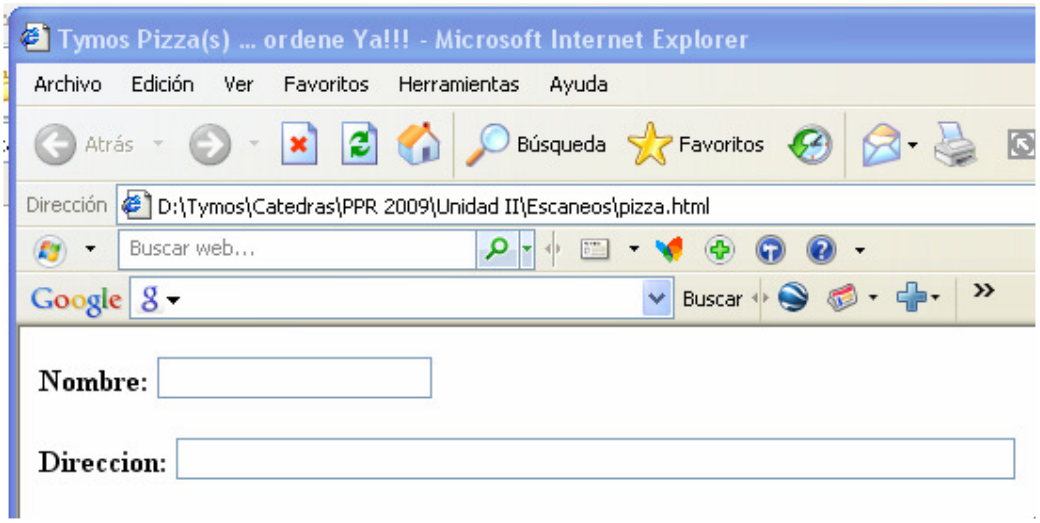
```
<input type="text" name="address" maxlength="30" size="30">
```

Veamos un ejemplo que muestra cómo crear una página con unos pocos controles en ella.

Vamos a construir un formulario que un cliente podría usar para pedir pizza por Internet. Primero, vamos a necesitar dos cuadros de texto para introducir el nombre y la dirección del cliente. Guarde la siguiente página como pizza.html:

```
<html>
<head>
<title>Tymos Pizza(s) … ordene Ya!!! </title>
</head>
<body>
      <form action = “procesar.jsp” method = “post”>
            <b> Nombre: </b>
            <input type = “text” name = “nombre” size = “30”<br>
            <br>
            <br>
            <b>Direccion:</b>
            <input type="text" name="direccion" size="70">
      </form>
</body>
</html>
```

y esto es todo lo que necesitamos para crear unos pocos cuadros de texto.
Si ahora hacemos doble clic sobre pizza.html



El atributo checked

Puede utilizarse para los elementos radio o checkbox. Causa la activación por defecto. Lo veremos en el ejemplo del pedido de pizzas.

El atributo value

Funciona de forma distinta dependiendo de los contenidos del atributo type.

- Controles seleccionables, definen el valor que se envía como parte del par nombre/valor al enviar el formulario.
- text y password, puede ser utilizado para prerellenar el control con datos específicos.
- Botones de control, los contenidos de este atributo se muestran en el botón.

Un ejemplo de cómo utilizaríamos el atributo value con un cuadro de texto:

```
<form action="process.jsp"> <input type="text" name="name" value="Tomas">
</form>
```

El formulario anterior presentará un campo de texto llamado nombre prerellenado con la cadena "Tomas".

Más de un control puede tener el mismo valor para el atributo name; esto ocurre cuando tenemos un grupo de controles que queremos utilizar juntos. Por ejemplo:

```
<form action="process.jsp">
      <input type="radio" name="rdb" value="123">
      <input type="radio" name="rdb" value="456">
</form>
```

El usuario puede seleccionar sólo uno de los botones. Si el primer botón, el par nombre/valor rdb=123 es enviado al servidor. Si el segundo botón, se envía el par nombre-valor rdb=456.

Para otros controles los usuarios pueden seleccionar cualquier combinación de ellos. Por ejemplo:

```
<form action="process.jsp">
      <input type="checkbox" name="cbx" value="123">
      <input type="checkbox" name="cbx" value="456">
</form>
```

Cuando el formulario se envía, si ambos cuadros de textos son activados, ambos pares nombre/valor son enviados.

Finalmente, también podemos utilizar el atributo value con botones. En este caso, el valor del atributo value es el texto presentado en la parte superior del formulario. Por ejemplo, la línea siguiente crea un botón de reinicio llamado dataReset que muestra el nombre Reset Form:

```
<input name="dataReset" "type="reset" value="Reset Form"/>
```

Otro ejemplo de creación de un botón de envío llamado dataSubmit que muestra el nombre formulario de envío en la parte superior:

```
<input name="dataSubmit" type="submit" value="Formulario de Envío"/>
```

Cómo añadir botones y casillas de verificación a un formulario HTML

Mejoremos aun más la página pizza.html. Agregamos

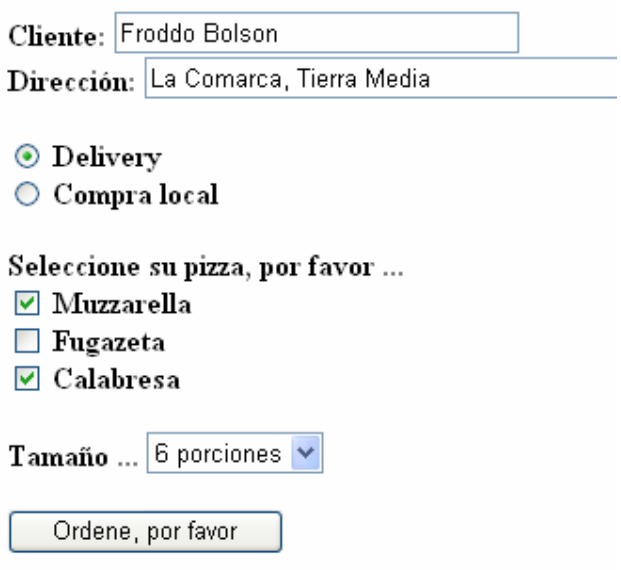
- Un conjunto de casillas de verificación para elegir ingredientes.
- Un par de botones de activación para que pueda elegir si quiere que se la lleven a casa o no
- Un botón de envío para enviar su pedido a la página process.jsp.
- Una lista desplegable con tamaños de la pizza solicitada

A continuación pizza.html completo, en negrita el agregado

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
  <head>
    <title>Pedir pizza</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
      <form action="proveerPizza.jsp" method="post">
          <b>Cliente:</b>
          <input type="text" name="cliente" size="30"><br>
          <b>DirecciÃ³n:</b>
          <input type="text" name="direccion " size="70">
        <br><br>
          <b>Seleccione su pizza, por favor ...</b>
        <br><br>
          <input type="radio" name="tipoCompra" value="Delivery" checked>
          <b>Delivery</b>
        <br>
          <input type="radio" name="tipoCompra" value="Compra local">
          <b>Compra local</b>
        <br><br>
        <br>
          <input type="checkbox" name="quiero" value="Muzzarella">
          <b>Muzzarella</b>
        <br>
          <input type="checkbox" name="quiero" value="Fugazeta">
          <b>Fugazeta</b>
        <br>
          <input type="checkbox" name="quiero" value="Calabresa">
          <b>Calabresa</b>
        <br><br>
          <b>TamaÃ±o ...</b>
          <select name= "tamanio">
              <option>4 porciones</option>
              <option>6 porciones</option>
              <option>8 porciones</option>
          </select>
        <br><br>
          <input type="submit" value="Ordene, por favor">
      </form>
    </body>
</html>
```

Una vez guardado el archivo, doble clic sobre pizza.html y en nuestro navegador Web la página de pedido de pizza aparece con estas nuevas opciones



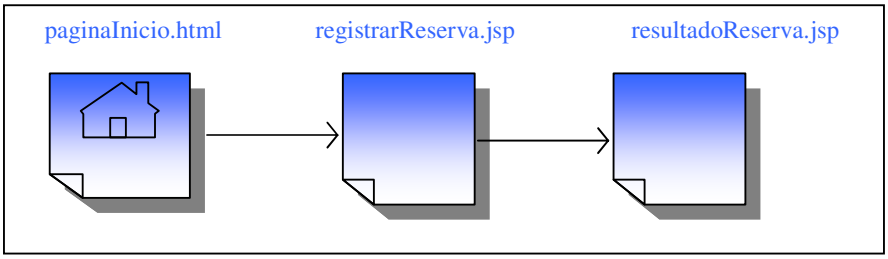
Diseñar páginas html/jsp usando la Paleta

- Las páginas **html/jsp** pueden diseñarse usando una paleta específica
- Ya usamos una paleta para diseñar formas para formularios basados en Swing.
- Aquí el objetivo es el mismo, la forma de trabajar algo distinta

Para ejemplificar, definiremos el proyecto Web *ReservasOnLineBis*
La aplicación tiene por finalidad

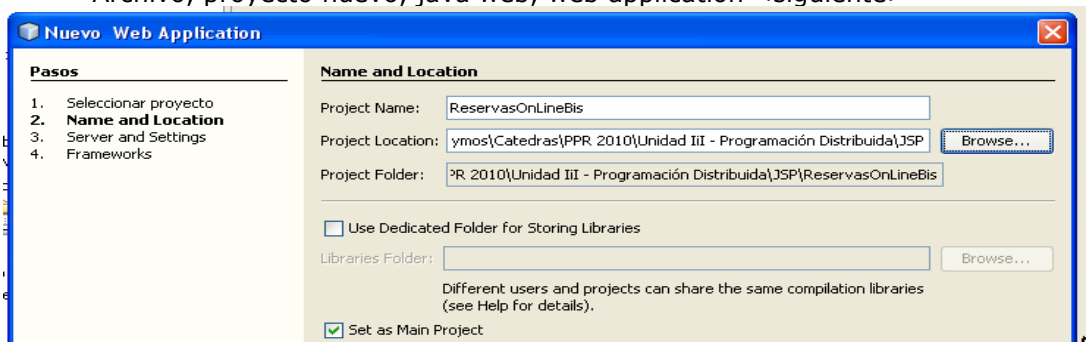
- permitir a los clientes del restaurante realizar una reserva vía Web.
  - El sistema deberá tomar los datos de la reserva,
  - verificar la disponibilidad para el día y hora solicitados
  - informar al cliente si la reserva pudo realizarse exitosamente o no.

El siguiente gráfico muestra el diagrama de páginas intervinientes.

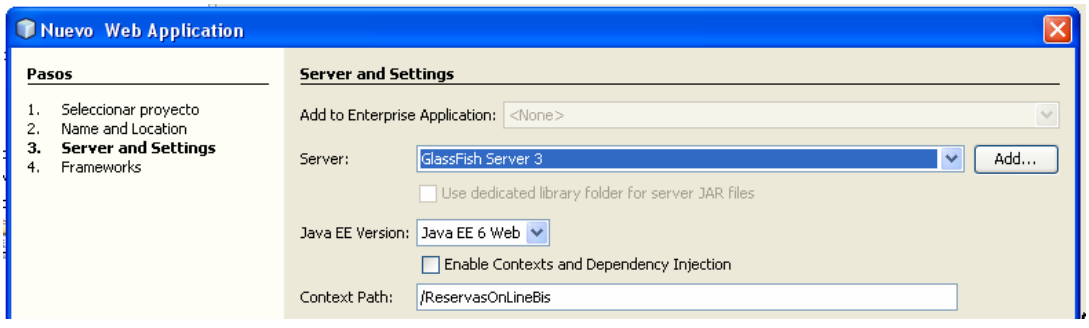


Definiendo el proyecto

Archivo, proyecto nuevo, java web, web application <siguiente>

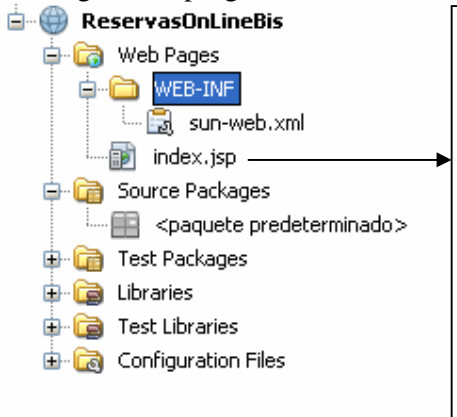


<Siguiente>



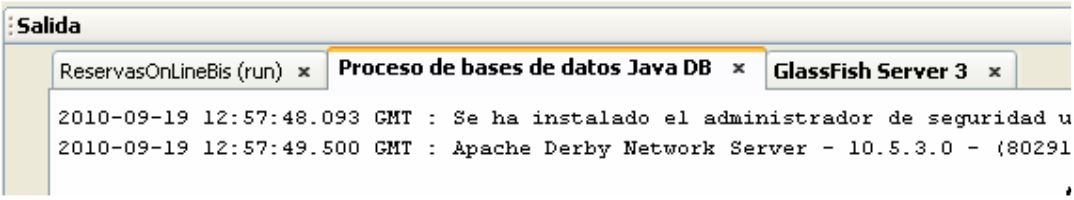
<Terminar>

NetBeans nos genera un proyecto




```
<%--
    Document   : index
    Created on : 19/09/2010, 09:24:50
    Author     : Usuario
--%>

<%@page contentType="text/html"
pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
   <head>
        <meta http-equiv="Content-Type"
       content="text/html; charset=UTF-8">
        <title>JSP Page</title>
   </head>
   <body>
      <h1>Hello World!</h1>
    </body>
</html>
```

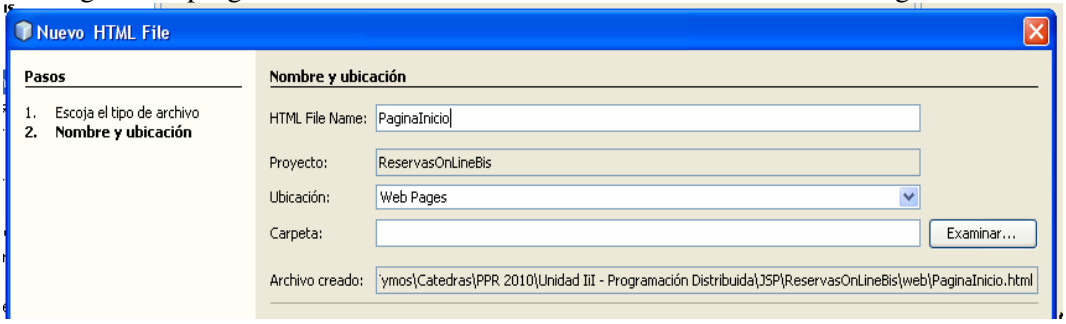
Este proyecto ya hace algo: Ejecutar, Ejecutar main project



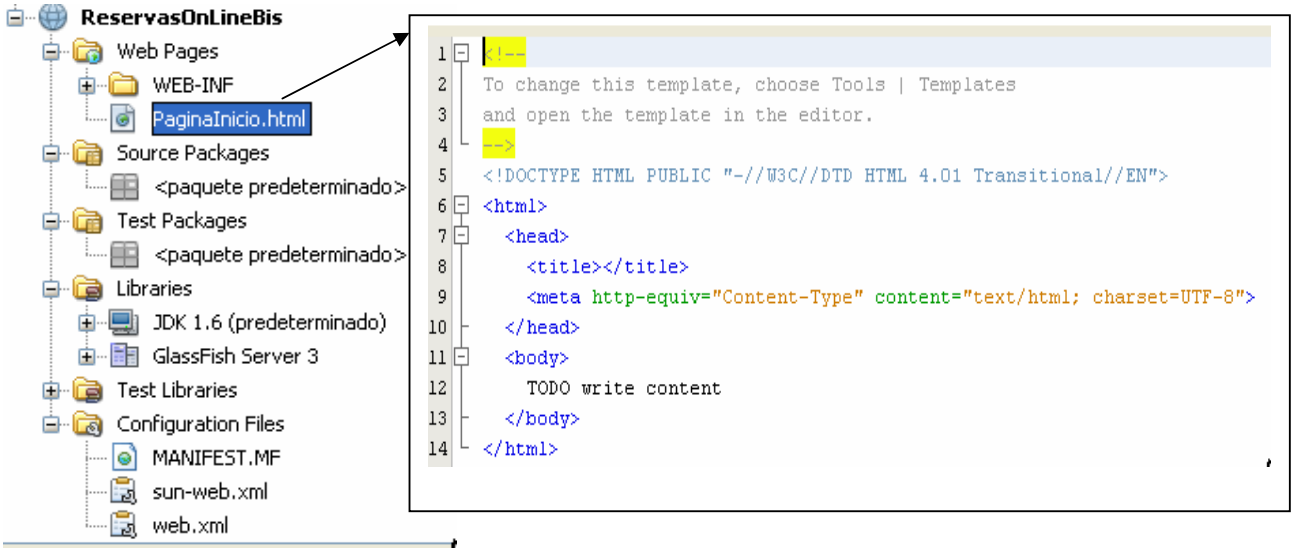
Y en nuestro navegador web

Borramos index.jsp e incorporamos PaginaInicio.html

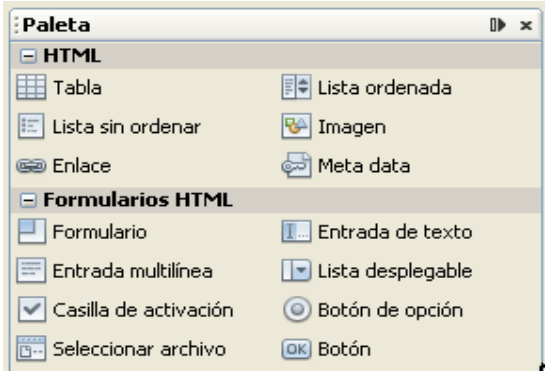
Archivo, archivo nuevo, web, HTML, <siguiente>



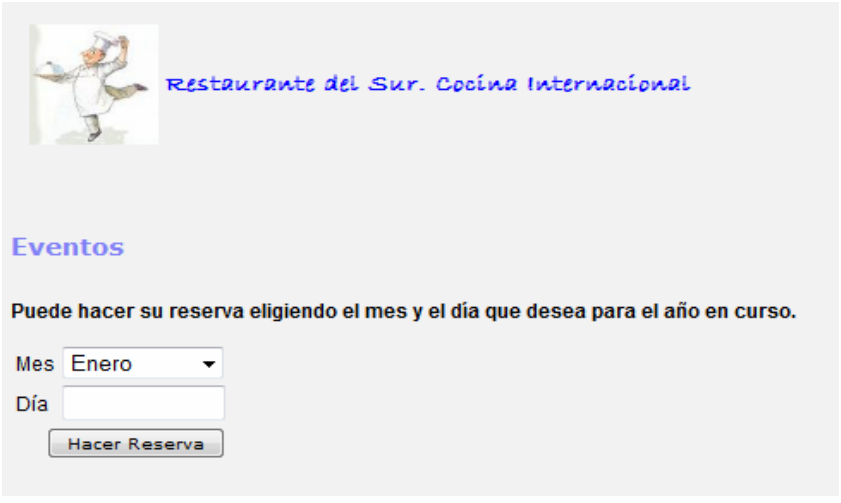
<Terminar> NetBeans nos muestra la incorporación de la página



Ademas de la edición de PaginaInicio NetBeans nos ofrece la paleta



El formulario que queremos presentar a nuestros clientes:

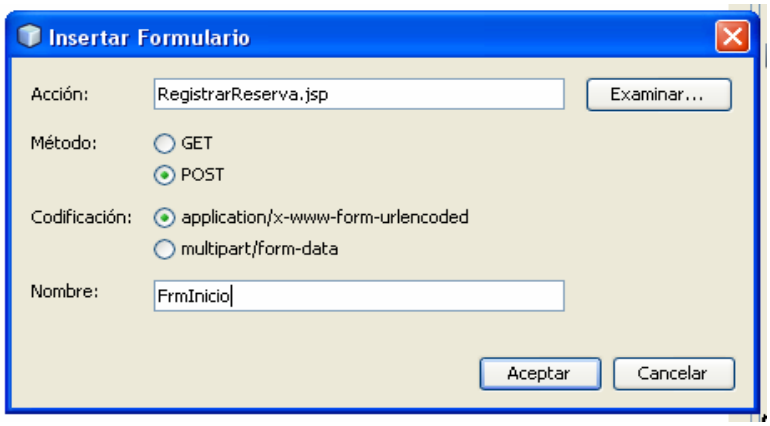


Esta página debe tener

- un formulario<form name="**frmInicio**"
- action="**RegistrarReserva.jsp**"
- method="**POST**">

Para insertar esta etiqueta en la página:

- En PaginaInicio posicionamos cursor detrás de <Body> <enter>
  (Esto define el punto de carga del siguiente componente)
- Paleta, Formularios HTML, Formulario, <<click>>



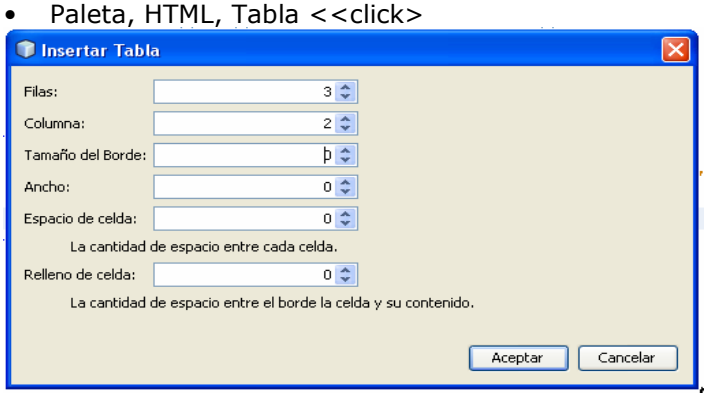
<Aceptar>

La codificación de PaginaInicio.html ahora es:

```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
      <form name="FrmInicio" action="RegistrarReserva.jsp" method="POST">
      </form>
  </body>
</html>
```

Una vez insertado el <form/> dentro de nuestro <Body>

- Debemos insertar un tabla (sin bordes) de tres filas y dos columnas para posicionar:
  - el combo de meses,
  - la entrada de texto para el día de la reserva,
  - el botón de hacer reserva
- Posicionamos cursor detrás de </form> y <Enter>
- Paleta, HTML, Tabla <<click>



<Aceptar>

En este caso se nos solicitan, además de la cantidad de filas y columnas a dibujar,

- Tamaño del borde: Valor 0 para indicar que no queremos borde.
- Ancho.
  - en porcentaje o en pixeles.
  - Si indicamos 0 tomará todo el ancho disponible.
- Espacio de celda
- Relleno de celda.

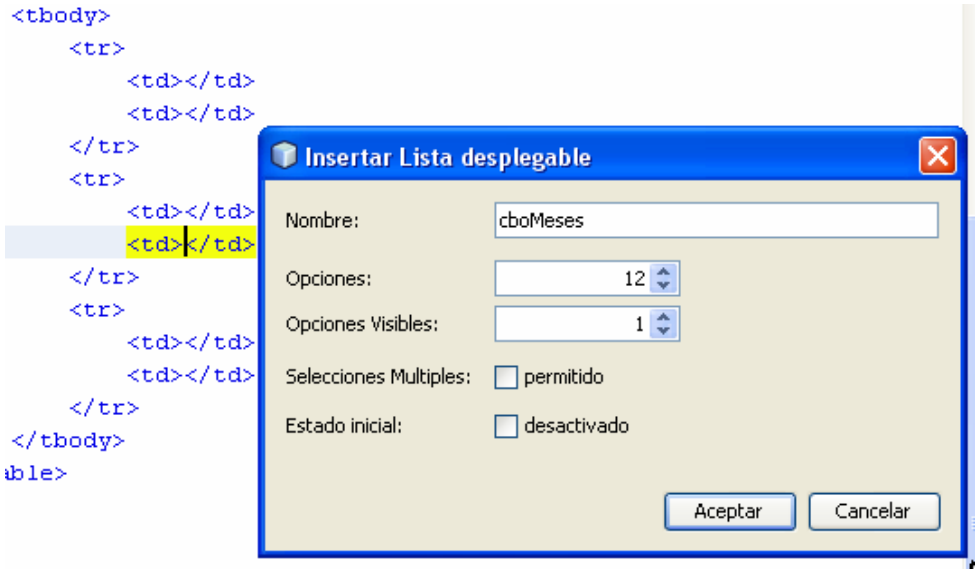
La codificación del cuerpo de **PaginaInicio.html** se modifica a:

```
<body>
    <form name="FrmInicio" action="RegistrarReserva.jsp" method="POST">
    </form>
    <table border="0">
        <thead>
            <tr>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td></td>
                <td></td>
            </tr>
            <tr>
                <td></td>
                <td></td>
            </tr>
            <tr>
                <td></td>
                <td></td>
            </tr>
        </tbody>
    </table>
</body>
```

Ahora rellenamos los casilleros con los componentes necesarios.

Lista combinada de meses

- Paleta, Formularios HTML, Lista desplegable
- arrastramos el componente, posicionamos cursor en el casillero tboby[1][2]
- Una línea vertical gruesa indica posición del cursor, levantamos dedo
- Aparece una forma solicitando más información.



<Aceptar>

El codigo de PaginaInicio.html se modifica a:

```
<tbody>
    <tr>
        <td></td>
        <td><select name="cboMeses">
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
                <option></option>
            </select></td>
    </tr>
```

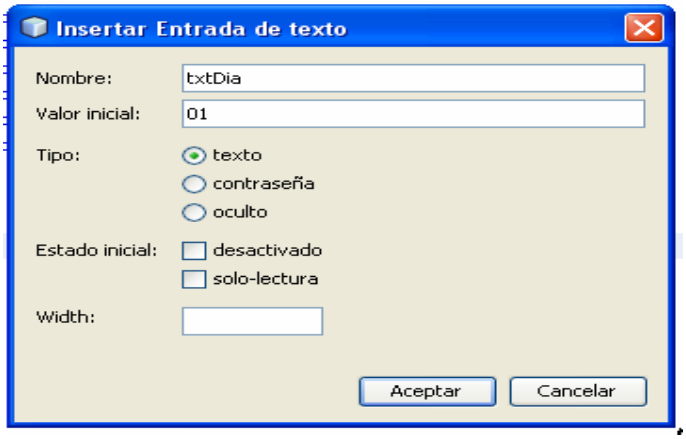
Codificamos **Mes** en el primer casillero
Codificamos los nombres de los meses para c/opción.

```
<tbody>
    <tr>
        <td>Mes</td>
        <td><select name="cboMeses">
                <option>Enero</option>
                <option>Febrero</option>
                <option>Marzo</option>
                <option>Abril</option>
                <option>Mayo</option>
```

.....

Entrada de texto para ingresar el día de reserva:

- Paleta, Formularios HTML, Entrada de texto
- arrastramos el componente, posicionamos cursor en el casillero tboby[2][2]
- Una línea vertical gruesa indica posición del cursor, levantamos dedo
- Aparece una forma solicitando más información.



Tipeamos Día en tbody[2][1]

Botón de opción Para generar la solicitud al servidor:

- Paleta, Formularios HTML, Botón
- arrastramos el componente, posicionamos cursor en el casillero tboby[3][2]
- Una línea vertical gruesa indica posición del cursor, levantamos dedo
- Aparece una forma solicitando más información.

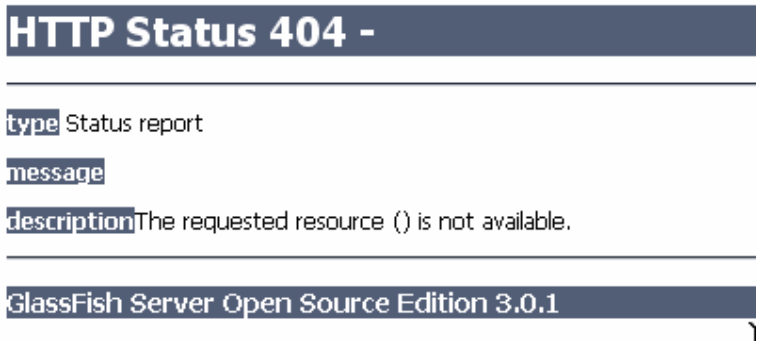
El tipo para los botones puede ser:

- **Enviar: Crea** un botón de envío (valor predeterminado). Cuando es presionado, el formulario al que corresponde ➔ servidor.
- **restablecer**: Crea un botón "restablecer". Cuando es presionado, todos los campos en el formulario vuelven a sus valores iniciales.
- **estandard**: Crea un botón "push". No tiene una acción predeterminada. Son usualmente definidos con scripts personalizados que manejas sus eventos.

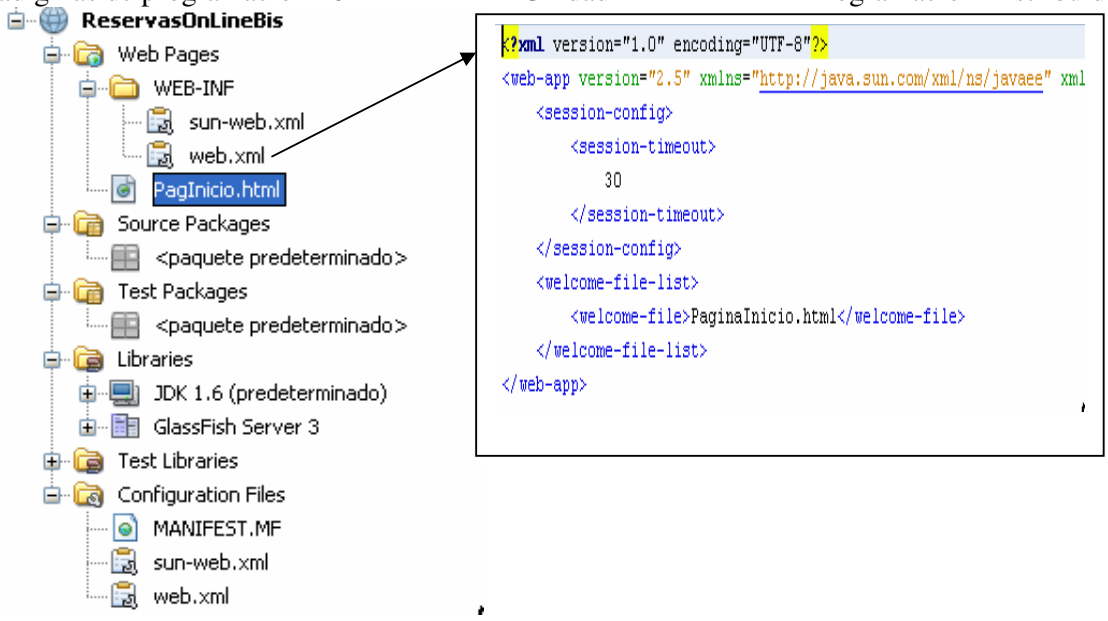
Quiero ver que tengo

Cursor sobre PaginaInicio.html <Derecho> <Vista>

Funcionará?
Borramos **index.jsp** y **Ejecutar, Ejecutar Main Project**



Ocurre que index.jsp no existe mas.
NetBeans no asume que PaginaInicio.html sea el pto de partida.
Necesitamos informárselo.
Para ello existe el descriptor de despliegue, web.xml.



Ejecutar, Ejecutar main Project

En el proyecto ReservasOnline esta interfaz es:

Completemos PaginaInicio.html

- En la etiqueta <title> tipeamos Restaurante del sur. Cocina Internacional
- Inmediatamente debajo de <form> incorporamos
  <H5>Eventos</H5>
  ** Puede hacer su reserva eligiendo el mes y el día que desea para el año en curso. **

- **La parte artística: El cocinero es una imagen incluida en otra página, headRestaurant.html**

headRestaurant.html

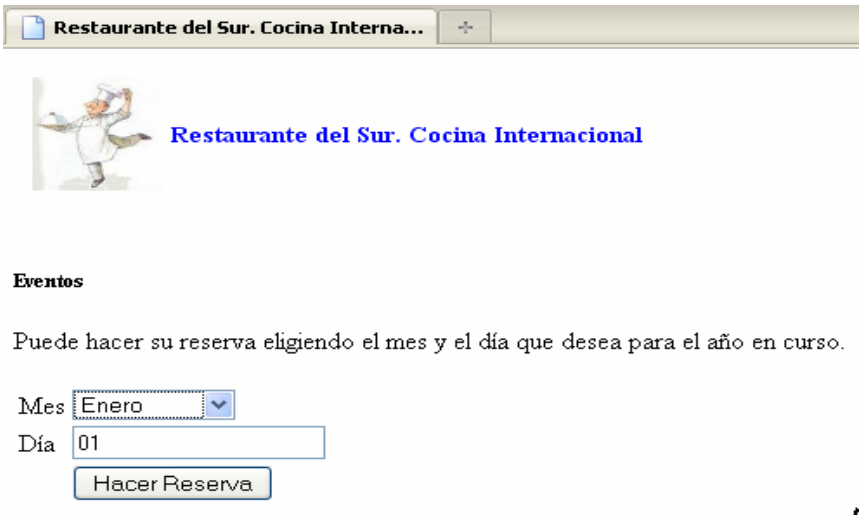
```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <table>
        <tr>
            <td>
                <img src="cocinero.jpg" width="75" height="75"/>
            </td>
            <td style="font-family:Bradley Hand ITC;
                font-size:15px; color:blue; font-weight:bold;">
                Restaurante del Sur. Cocina Internacional
            </td>
        </tr>
    </table>
</html>
```

- Incluimos (Copiar/pegar) en nuestro proyecto, web pages
  - headRestaurant.html
  - cocinero.jpg
- Incluir headRestaurant.html en PaginaInicio.html:
  Como se hace?

```
<!-- La etiqueta object provee una forma de ejecutar
aplicaciones externas.
Generalmente, es usado para ejecutar applets, animaciones flash
o para mostrar imágenes u otras páginas.
En este caso se uso con la finalidad de 'incluir' una página
dentro de otra (el encabezado de la página). -->

 <object data="headRestaurant.html" type="html/text"
         height="115" width="500"></object>
```

Copiar del proyecto original, PaginaInicio.html
  - el comentario
  - la etiqueta object
- Pegar en el mismo sitio

Ejecutar, Ejecutar main project



Bastante bien, no?

La codificación de PaginaInicio.html:

```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Restaurante del Sur. Cocina Internacional</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <!-- La etiqueta object provee una forma de ejecutar
        aplicaciones externas.
        Generalmente, es usado para ejecutar applets, animaciones flash
        o para mostrar imágenes u otras páginas.
        En este caso se uso con la finalidad de 'incluir' una página
        dentro de otra (el encabezado de la página). -->

         <object data="headRestaurant.html" type="html/text"
                 height="115" width="500"></object>
  </head>
  <body>
      <form name="FrmInicio" action="RegistrarReserva.jsp" method="POST">
            <h5>Eventos</h5>
            <p>Puede hacer su reserva eligiendo el mes y el día que desea
        <table border="0">
          <thead>
              <tr>
                  <th></th>
                  <th></th>
              </tr>
          </thead>
```

```
        <tbody>
            <tr>
                <td>Mes</td>
                <td><select name="cboMeses">
                        <option>Enero</option>
                        <option>Febrero</option>
                        <option>Marzo</option>
                        <option>Abril</option>
                        <option>Mayo</option>
                        <option>Junio</option>
                        <option>Julio</option>
                        <option>Agosto</option>
                        <option>Setiembre</option>
                        <option>Octubre</option>
                        <option>Noviembre</option>
                        <option>Diciembre</option>
                    </select></td>
            </tr>
            <tr>
                <td>Día</td>
                <td><input type="text" name="txtDia" value="01" /></td>
            </tr>
            <tr>
                <td></td>
                <td><input type="submit" value="Hacer Reserva" name="btnHacerReserva"
            </tr>
        </tbody>
    </table>
  </form>
</body>
</html>
```

Procesamiento de peticiones

Hasta ahora hemos visto cómo los formularios HTML y los elementos de formulario pueden ser utilizados para presentar diferentes controles de entrada en la pantalla del navegador del usuario, generando una interfaz gráfica que le permite introducir datos en el formulario. Esta es sólo una cara del desarrollo de aplicaciones Web. Ya hemos visto que cuando los usuarios envían los formularios, sus datos se envían al servidor. El servidor debe ser capaz de leer e interpretar estos datos. La manera en la que se lleva a cabo esta tarea en el servidor depende mucho de que es lo tenemos en el servidor. A continuación veremos cómo se puede utilizar la tecnología JSP/Servlet API para procesar las peticiones enviadas al servidor desde el navegador del cliente.

Utilización del objeto request

En Java antes de utilizar una variable tenemos que declararla especificando sus tipos de datos. En cambio, los contenedores Web J2EE proporcionan un conjunto de variables implícitas que pueden utilizarse dentro de sus páginas JSP sin tal declaración explícita. Una de esas variables es el objeto request, que contiene datos de las peticiones del cliente; puede utilizar este objeto, dentro de sus páginas JSP, para acceder a los datos pedidos, como los datos de formulario.

El objeto request tiene un método denominado getParameter() que nos permite recuperar el valor de un par nombre/valor desde el objeto, si le decimos su nombre. Un scriplet con esto:

```
<%
      String specialReq = request.getParameter("specialRequest");
%>
```

El tipo de dato, o sea la clase, de nuestro objeto request se llama HttpServletRequest. Como toda clase, tiene su propia colección asociada de métodos que puede utilizar y getParameter() es uno de ellos.

Veamos otro caso. Supongamos que utilizamos un cuadro de lista de multiselección y seleccionamos más de un componente de la lista. (Por ejemplo, tildamos Muzzarela y Calabresa, queremos ½ pizza de cada). Ya sabemos que en este caso varios pares nombre/valor, todos con el mismo nombre, se transfieren al servidor con los datos del formulario. En este caso, tenemos que utilizar el método getParameterValues(), otro scriplet, de esta forma:

```
<%
    String[] quieroPizza = request.getParameterValues("quiero");
%>
```

El método getParameterValues() devuelve un array de Strings.
Hay que tener en cuenta una última cosa sobre el objeto request: a diferencia de las variables y de los objetos de muchos otros tipos de datos, este objeto es implícito. Esto significa que no tenemos que declarar el objeto para crearlo; se crea para nosotros.

Ahora vamos a crear la página a la que nuestra página pizza.html envía los datos del formulario. Esta página, se llama proveerPizza.jsp:

```
<%--
    Document   : index
    Created on : 15-mar-2009, 17:31:43
    Author     : usuario
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Su pizza</title>
    </head>
    <body>
        <h1>Su pizza, ya!</h1>
        <b>Cliente:  </b>
        <% out.println(request.getParameter("cliente")); %><br>
        <b>DirecciÃ³n:</b>
        <% out.println(request.getParameter("direccion")); %><br>
        <b>Modalidad:</b>
        <% out.println(request.getParameter("tipoCompra"));%><br>
       <b>Va pizza</b>
       <%String[] quieroPizza = request.getParameterValues("quiero");
            for(int i = 0;i < quieroPizza.length; i++)
                out.println(quieroPizza[i]+", ");%><br>
       <b>TamaÃ±o</b>
        <% out.println(request.getParameter("tamanio")); %><br>
   </body>
</html>
```

Funcionamiento

Veamos cómo recuperar el primer valor, el del nombre del cliente:

```
<b>Name:</b>
<% out.println(request.getParameter("cliente")); %><br>
```

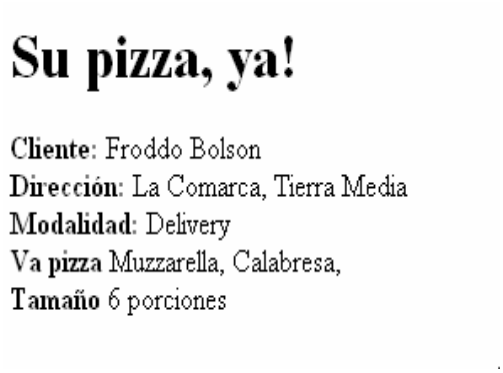
El valor null

Si no fuera que la primera casilla de verificación en la página de pedidos tiene la cláusula **checked**, y si el cliente olvidara tildar el tipo de entrega, el valor que recibiríamos del objeto request no sería "no" como podría esperarse, sino algo llamado null. ¿Qué es?

Una entrada null representa datos que faltan. Debería observar que mientras no pertenece a ningún tipo particular de datos, un null puede utilizarse para representar datos que faltan de cualquier tipo de datos. Un null devuelto por una variable significa simplemente que no hay ningún valor disponible para esta variable. Esto no significa que la variable contenga un cero, cero es un valor.

Para variables String, no significa que la variable contenga un String vacío, o un string compuesto por uno o más espacios en blanco, porque estos siguen siendo valores de String. Cualquier proceso aritmético o de otro tipo que implique un null devolverá null también; después de todo, no podemos procesar datos de forma adecuada si falta alguno de esos datos. Recuerde que un **null es nada:** ni un tipo de datos ni un valor.

Si llenamos los datos pedidos por en el formulario html, la página que devuelve proveerPizza.jsp puede ser la siguiente:



Presentación de JavaBeans

Ya hemos visto los scriptlets y conocemos el problema asociado con su utilización: estamos dejando de lado toda la orientación objetos.

En este capítulo vamos a aprender cómo sortear estos problemas organizando nuestro código siguiendo la citada orientación: Debemos mover los datos y el comportamiento para tratarlos a JavaBeans ("beans").

Por lo tanto, en este capítulo veremos cómo crear y utilizar JavaBeans.

Separación de papeles

Si consideramos un sitio Web típico de comercio electrónico que utilice JSP, el mantenimiento de este sitio es a menudo responsabilidad de dos grupos de gente diferentes: el diseñador y el desarrollador. El papel del diseñador es mantener la presentación o imagen del sitio. El desarrollador es el responsable de la funcionalidad o la lógica de la aplicación. La lógica de la aplicación es el código Java que proporciona cosas como acceso a bases de datos o reglas de negocios.

Puede existir una separación clara entre estos dos papeles o puede darse un cierto grado de solapamiento entre ellos. El punto clave es que cuando piensa en el diseño de la página no quiere tener que preocuparse por el código Java que le proporciona la funcionalidad. Como diseñador quiere tratar tareas como, por ejemplo, el acceso a bases de datos como si fueran "cajas negras" es decir, pone algo y obtiene algo, pero no le interesa lo que realmente ocurre dentro. Como desarrollador, no quiere preocuparse sobre cómo se muestran los datos que proporciona ni cómo se pasan a los clientes del sitio Web.

Esta separación de papeles lleva a la organización de la aplicación en capas.

- JSP es el nivel de presentación de la arquitectura J2EE.
- Hay otras partes de la arquitectura responsables del acceso a los datos y la lógica del negocio.

Componentes

Existen dos objetivos importantes cuando diseñamos una aplicación Web JSP:

- la reutilización del código
- el agrupamiento del código en capas.

La propuesta lógica es dividir el código en componentes. Esto ya se vio en AED, Capítulo III, Estrategias de resolución. Allí tratamos como problemas complejos se resuelven mediante el concurso de objetos de diversas clases que interactúan entre si. Allí hemos demostrado que dividir el código en componentes en lña forma de programar. Entonces, ¿cómo hacemos esto en una aplicación Web basada en JSP?

Introducción a JavaBeans

Ya sabemos qué son los componentes y por qué querría utilizados en mis aplicaciones Web. Cómo utilizar realmente los componentes con páginas JSP? La respuesta es que se utilizan JavaBeans (a menudo denominadas 'beans').

JavaBeans son simplemente objetos que exponen partes de datos llamados propiedades. (Recuerda Ud los gets()). Podemos utilizamos beans para almacenar estos datos y recuperarlos posteriormente. Por ejemplo, podríamos tener un bean que modelara una cuenta bancaria. Este bean podría tener propiedades de número de cuenta y de saldo.

Formalmente, un JavaBean no es más que una clase que mantiene algunos datos (las propiedades) y sigue unas determinadas convenciones de codificación. Estas convenciones proporcionan un mecanismo para el soporte automatizado. Este

soporte automatizado significa que el motor de JSP, por ejemplo Tomcat, puede inspeccionar el bean y descubrir las propiedades que tiene.

Propiedades

Cada conjunto de información expuesto por un bean se denomina propiedad.
Para encontrar un ejemplo de propiedades públicas no tenemos que irnos más lejos que a HTML.

```
<font face="Arial, Helvetiea, sans-serif" color="ff330099">
      JSP Bean Properties Syntax
</font>
```

En HTML, refiriéndonos por ejemplo a la etiqueta <font>, face y color son los dos ejemplos de propiedades que pueden establecerse frente a la etiqueta <font> estándar de HTML. Seguramente este tipo de propiedad le es ya muy familiar.

Igual que las propiedades HTML, las propiedades en JavaBeans proporcionan una aproximación simple a cómo poder pasar información para establecer o recuperar un valor, con el fin de utilizarlo en su código JSP.

Las propiedades en un JavaBean se exponen públicamente utilizando métodos getter y setter. Estos métodos siguen convenciones de denominación simples, que entenderemos más fácilmente con un ejemplo. Si tenemos una propiedad denominada color, los métodos getter y setter se llamarían getColor () y setColor () . Los nombres del método son sólo el nombre de la propiedad comenzando por letra mayúscula y precedido por get o set.

Para construir un JavaBean propio todo lo que tenemos que hacer es escribir una clase Java.

Construcción de un JavaBean

¿Qué aspecto tiene un JavaBean? Definamos una clase JavaBean llamada CarBean que podríamos utilizar como componente en un sitio Web de venta de coches. Será un componente que modelará un coche y tendrá una propiedad: la marca del coche.
Usemos el entorno NetBeans, abrimos un proyecto nuevo (Por ejemplo JSP-ch4, package FirstBean, y codificamos CarBean.java

```
package FirstBean;

import java.io.Serializable;
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author Tymos
 */
public class CarBean implements Serializable{
    private String make = "Gol 1000";
    public CarBean(){
    }

    public String getMake(){
        return make;
    }
    public void setMake(String make){
        this.make = make;
    }
}
```

Está UD. viendo algo nuevo? Claro que no.

Utilización de un JavaBean

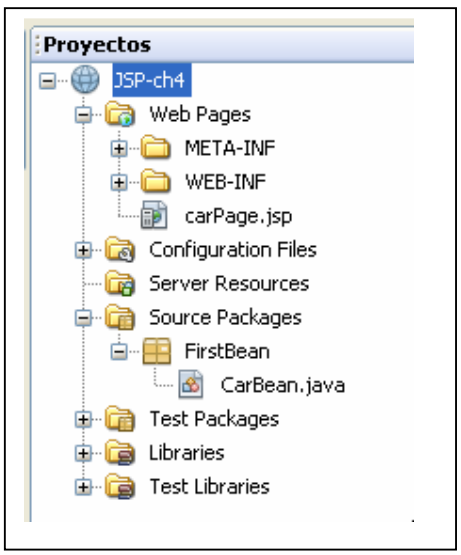
Ahora codifiquemos la página carPage.JSP que utilizará nuestro bean:

```
<%--
    Document   : carpage.jsp
    Created on : 15-mar-2009, 18:13:51
    Author     : Tymos
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Usando un primer javaBean</title>
    </head>
    <body>
        <h2>Usando mi primer javaBean</h2>
        <% FirstBean.CarBean myCar = new FirstBean.CarBean(); %>
        Me gusta mi <%= myCar.getMake() %> <br/>
        <% myCar.setMake("Ferrari Testa Rosa"); %>
        pero también un <%= myCar.getMake() %>
    </body>
</html>
```

Que tenemos?

- En <head>, el título, a continuación del nombre del navegador
- En <body>
  - Un encabezado tamaño h2, es parte de la página
  - Un scriplet construyendo el objeto mycar
  - Una línea de página compuesta: Constante + retorno scriplet.
  - Un scriplet modificando atributo de Carbean
  - Una línea de página compuesta: Constante + retorno scriplet.

NetBeans nos muestra donde están almacenados ambos archivos



Queremos ver como esto funciona.

Posicionamos cursor en carPage.jsp, botón derecho, run File.

Vemos que el NetBeans levanta Tomcat, luego aparece su explorador Web predeterminado, y en su área de edición:

Usando mi primer javaBean

Me gusta mi Gol 1000
pero tambien un Ferrari Testa Rosa

Eso es todo

Ya hemos visto cómo crear nuestra propia clase bean y utilizarla en una página JSP. Podríamos dar propiedades y métodos adicionales a nuestra clase CarBean para hacerla más interesante y útil. Sin embargo, una parte de nuestro objetivo

era eliminar la utilización excesiva de scriplet, para que el código fuera más sencillo de seguir, y hasta el momento no hemos hecho más que introducir más scriplets que contienen llamadas a métodos bean dentro de código JSP. Aquí es donde podemos introducirle el poder real de JavaBeans.

Es importante que los métodos getter y setter sigan estrictas convenciones de denominación? Esto todavía no ha quedado claro, pero muy pronto lo estará.
A continuación introduciremos las etiquetas bean, que nos **permitirán eliminar la necesidad de los scriplets** para llamar a métodos bean en la JSP.

Etiquetas bean

JSP proporciona un método de utilización de JavaBeans que se basa en el concepto de etiquetas. Estas etiquetas no son en realidad más complicadas que las etiquetas HTML estándar: tienen un nombre y toman atributos.

Las etiquetas están diseñadas para que el trabajo del desarrollador de la página sea más sencillo, ya que permiten al diseñador utilizar JavaBeans sin saber nada de Java. Existen tres etiquetas estipuladas por la especificación para sostener la utilización de JavaBeans en sus páginas JSP:

- **<jsp:useBean>**
  Localiza y hace una llamada a un JavaBean. Por ejemplo:

  ```
  <jsp:useBean id="myCar" class="FirstBean.CarBean"/>
  ```

  Un objeto del paquete.clase FirstBean.CarBean será localizado y creado.

  Podremos referirnos a este bean más adelante en la página, utilizando el valor establecido para el atributo id(myCar).

  Esta etiqueta tiene dos formas de cierre.
  - Notación corta /> mostrada previamente.
  - Final de etiqueta completo </jsp: useBean>:

  ```
  <jsp:useBean id="myCar" class="com.wrox.cars.CarBean" ></jsp:useBean>
  ```

- **<jsp:setProperty>**

  Establece el valor de una propiedad del bean utilizando el método setter.

  Si hemos llamado a un JavaBean con un id de myCar, como hicimos en la anterior sección, podemos utilizar la siguiente etiqueta para establecer el valor de una propiedad del bean.

  ```
  <jsp:setProperty name="myCar" property="make" value="Ferrari"/>
  ```

  Aquí es donde la utilidad del método de convenciones de nombramiento se hace evidente. Esta etiqueta toma la llamada al bean con el id de myCar y le aplica el método setMake(). Transfiere el argumento “Ferrari” al método setMake() y estamos asignando “Ferrari” a la propiedad make.

- **<jsp:getProperty>**

  Obtiene el valor de una propiedad utilizando el método getter y devuelve el valor de la propiedad a la página JSP que lo demanda.

  ```
  <jsp:getProperty name="myCar" property="make" />
  ```

  Obtenemos el valor de la propiedad make de la llamada al bean myCar.

Ahora volvamos a escribir nuestros ejemplos anteriores utilizando estas tres etiquetas en vez del código Java en scriptlets.

El código .jsp modificado queda:

```
<%--
    Document   : carPage1
    Created on : 15-mar-2009, 18:44:10
    Author     : Tymos
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Usando un primer javaBean (Bis)</title>
    </head>
    <body>
        <h2>Usando mi primer javaBean (Bis)</h2>
        <jsp:useBean id = "myCar" class = "FirstBeanBis.CarBean" />
            Me conformo con mi <jsp:getProperty name = "myCar" property="make"/>
      <br/>
        <jsp:setProperty name ="myCar" property = "make" value = "Ferrari Testa
            Rosa"/>
        pero muero por un <jsp:getProperty name = "myCar" property = "make"/>
    </body>
</html>
```

En el navegador Web vemos:

Note que en esta ocasión la respuesta fue mucho mas rápida, ya que el contenedor Web Tomcat estaba instalado.

Un examen más detallado de los métodos JavaBean

Aunque sólo hemos utilizado los métodos getter y setter en nuestro bean hasta ahora, debería damos cuenta de que no tenemos por qué utilizar métodos sólo para obtener y establecer propiedades.

- **Los métodos del bean deberían contener cualquier tipo de funcionalidad.** Por ejemplo, podríamos incluir un método que establezca una conexión a una base de datos, y este método sería llamado por otro método en el bean. Sin embargo, a menos que incluya métodos getter y setter, no será capaz de recuperar/establecer valores en los beans utilizando etiquetas.

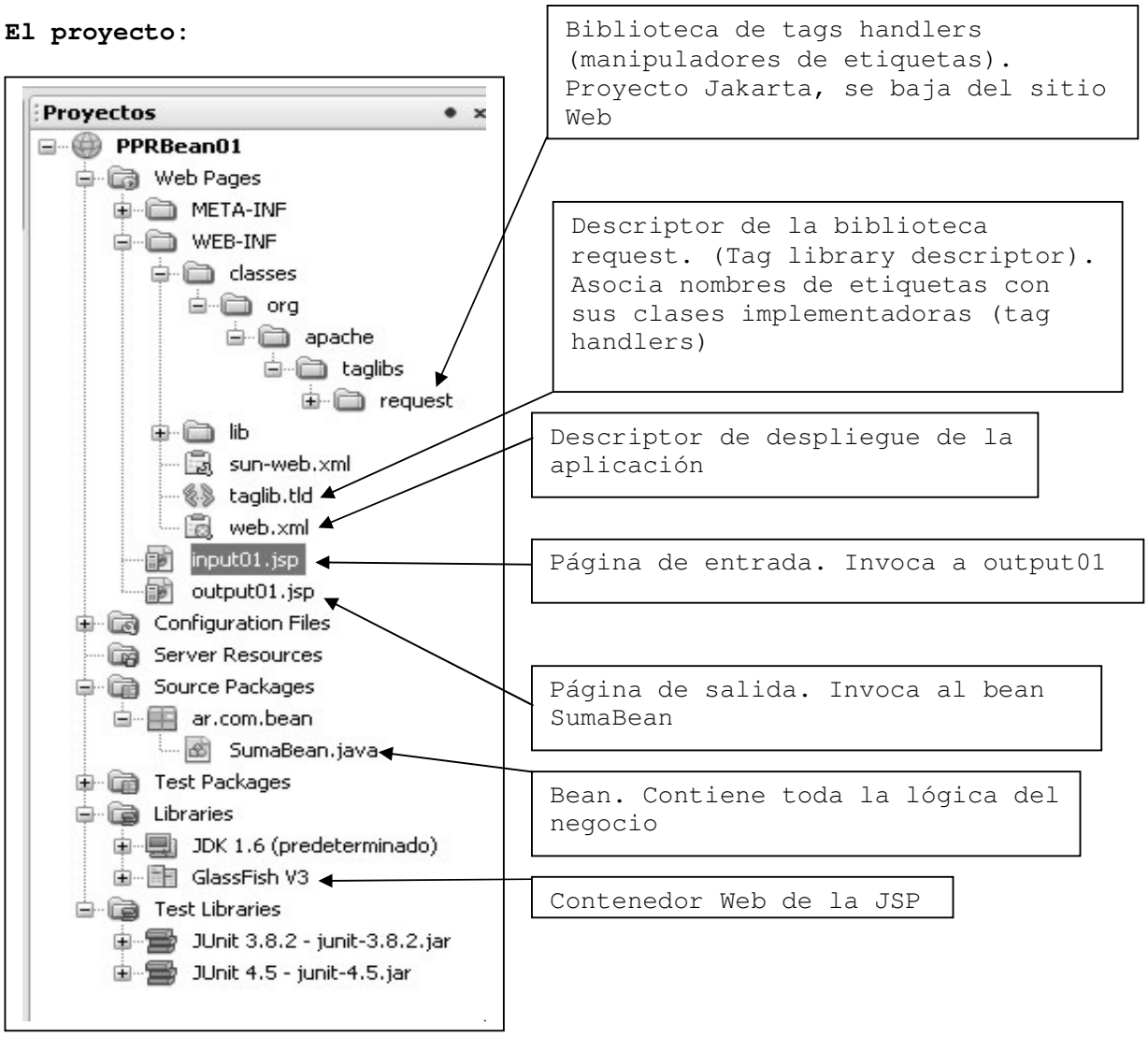
- **las propiedades que obtenemos no tienen por qué ser atributos bean.** podemos devolver el valor de cualquier variable desde el método getter utilizando una etiqueta, no sólo atributos de objeto.

Veamos ahora un segundo Bean, un poco mas elaborado.

Proyecto PPRBean01

Enunciado: Se tiene una sucesión de números enteros definida a partir de valores desde/hasta informados. El cliente puede pedir diversas sumas de sus términos.

El proyecto:



El descriptor de despliegue **web.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>input01.jsp</welcome-file>
    </welcome-file-list>
     <taglib>
        <taglib-uri>/WEB-INF/taglib.tld</taglib-uri>
         <taglib-location>/WEB-INF/taglib.tld</taglib-location>
     </taglib>
</web-app>
```

Descriptor de la biblioteca **request**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0.1</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>request</shortname>
  <uri>http://jakarta.apache.org/taglibs/request-1.0</uri>
  <info>The REQUEST custom tag library contains tags which can be used
     to access all the information about the HTTP request for a JSP
     page.Tags are provided to access information in the HTTP request
      for HTTP input parameters from a POST or GET, HTTP Headers,
     Cookies, request attributes, and session information related to
     this request.
  </info>
  <tag>
    <name>log</name>
    <tagclass>org.apache.taglibs.request.LogTag</tagclass>
    <bodycontent>JSP</bodycontent>
  </tag>
  <tag>
    <name>request</name>
    <tagclass>org.apache.taglibs.request.RequestTag</tagclass>
    <teiclass>org.apache.taglibs.request.RequestTEI</teiclass>
    <bodycontent>empty</bodycontent>
    <attribute>
      <name>id</name>
      <required>yes</required>
      <rtexprvalue>no</rtexprvalue>
    </attribute>
  </tag>

     … Sigue descripción de los restantes tags
```

Página de entrada **input01.jsp**

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
1">
   <title>PPRBean01, sumando en una sucesión</title>
</head>
<body bgcolor="#FFFFFF">
    <h3>Sumando según distintos criterios en una sucesión</h3>
    <br>
    <form action="output01.jsp">
    Por favor, su identificación <br>
    (<b>nickName</b>)<input type="text" name="nickName"
     size="20"><br><br>

    Y su celular donde enviarle los resultados<br>
    (<b>telCel</b>)<input type="text" name="telCel" size="15"><br><br>

    Por favor, indíquenos cual es el intervalo que desea trabajar<br>
    Desde:(<b>desde</b>) <input type="text" name="desde" size="5">
    , Hasta:(<b>hasta</b>) <input type="text" name="hasta"
```

```
      size="5"><br><br>

    Indíquenos cuales sumas necesita, por favor...<br>
    <input type="checkbox" name="sumaTodos" value= "X">
     Suma de todos sus términos<br>
    <input type="checkbox" name="sumaPares"  value  ="X">
     Suma de sus términos pares<br>
    <input type="checkbox" name="sumaImpares" value="X">
     Suma de sus términos impares<br>
    <input type="checkbox"name="sumaPrimos"  value="X">
     Suma de sus términos primos<br>
    <input type="checkbox" name="sumaNoPrimos" value="X">
     Suma de sus términos no primos<br><br><br>
    Click en <input type="submit" name="submit" value="submit">para el
     resultado<br>
    que la página <b>output01.jsp</b> nos devuelve.
    </form>
</body>
</html>
```

Se visualiza:

Sumando según distintos criterios en una sucesión

Por favor, su identificación
(**nickName**) Petete

Y su celular donde enviarle los resultados
(**telCel**) 15622222

Por favor, indíquenos cual es el intervalo que desea trabajar
Desde:(**desde**) 100 , Hasta:(**hasta**) 200

Indíquenos cuales sumas necesita, por favor...
- ☑ Suma de todos sus términos
- ☑ Suma de sus términos pares
- ☐ Suma de sus términos impares
- ☑ Suma de sus términos primos
- ☑ Suma de sus términos no primos

Click en [submit] para el resultado
que la página **output01.jsp** nos devuelve.

Al clickear submit entra en acción la página output01.jsp

- **req:parameter:** Usa el tag handler parameter de la biblioteca cuyo prefijo es req para obtener el dato llamado(name) del objeto automático request.
- Define el objeto **jsp:useBean id="sumaBean"**
- Le pasa atributos **jsp:setProperty name="sumaBean"**
- Recibe resultados **jsp:getProperty name="sumaBean"**

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
                                                                    1">
   <title>PPRBean01 - Salida de datos simple</title>
</head>
<body bgcolor="#FFFFFF">
    <%@ taglib uri="/WEB-INF/taglib.tld" prefix="req" %>
    Estimado Sr(a). <b><req:parameter name="nickName"/></b>
    Tel/cel: <b><req:parameter name="telCel"/></b><br><br>

    Hemos registrado su siguiente solicitud de servicios:<br><br>
     Intervalo desde: <b><req:parameter name="desde"/></b>,
              hasta: <b><req:parameter name="hasta"/></b><br>

     Suma de todos sus términos.........
     <b><req:parameter name="sumaTodos"/></b><br>

     Suma de sus términos pares.........
     <b><req:parameter name="sumaPares"/></b><br>

     Suma de sus términos impares.....
     <b><req:parameter name="sumaImpares"/></b><br>

     Suma de sus términos primos.......
     <b><req:parameter name="sumaPrimos"/></b><br>

     Suma de sus términos no primos.
     <b><req:parameter name="sumaNoPrimos"/></b><br><br><br>

    Resultados del procesamiento de datos solicitado<br><br>

    <%// A continuación pasamos los datos del cliente a  SumaBean %>

    <jsp:useBean id="sumaBean" class="ar.com.bean.SumaBean"/>
    <jsp:setProperty name="sumaBean" property="desde"/>
    <jsp:setProperty name="sumaBean" property="hasta"/>
    <jsp:setProperty name="sumaBean" property="sumaTodos"/>
    <jsp:setProperty name="sumaBean" property="sumaPares"/>
    <jsp:setProperty name="sumaBean" property="sumaImpares"/>
    <jsp:setProperty name="sumaBean" property="sumaPrimos"/>
    <jsp:setProperty name="sumaBean" property="sumaNoPrimos"/>

    <%// A continuación obtenemos los datos requeridos de SumaBeam%>
    Intervalo procesado desde: <jsp:getProperty name="sumaBean"
                                              property="desde"/>
    , hasta : <jsp:getProperty name="sumaBean" property="hasta"/><br>
    <jsp:getProperty name="sumaBean" property="sumaTodos"/><br>
    <jsp:getProperty name="sumaBean" property="sumaPares"/><br>
    <jsp:getProperty name="sumaBean" property="sumaImpares"/><br>
    <jsp:getProperty name="sumaBean" property="sumaPrimos"/><br>
    <jsp:getProperty name="sumaBean" property="sumaNoPrimos"/><br><br>
    Agradecemos la atención dispensada.
</body>
</html>
```

Se visualiza:

Estimado Sr(a). **Petete** Tel/cel: **15622222**

Hemos registrado su siguiente solicitud de servicios:

Intervalo desde: **100**, hasta: **200**
Suma de todos sus términos..... X
Suma de sus términos pares..... X
Suma de sus términos impares.....
Suma de sus términos primos..... X
Suma de sus términos no primos. X

Resultados del procesamiento de datos solicitado

Intervalo procesado desde: 100 , hasta : 200
Suma todos 14950
Suma pares 7450
Suma impares .....No solicitado
Suma primos 3167
Suma no primos ..11783

Agradecemos la atención dispensada.

Importante: Los atributos con el mismo nombre en la página y el bean pueden ser transferidos en forma genérica usando la forma

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

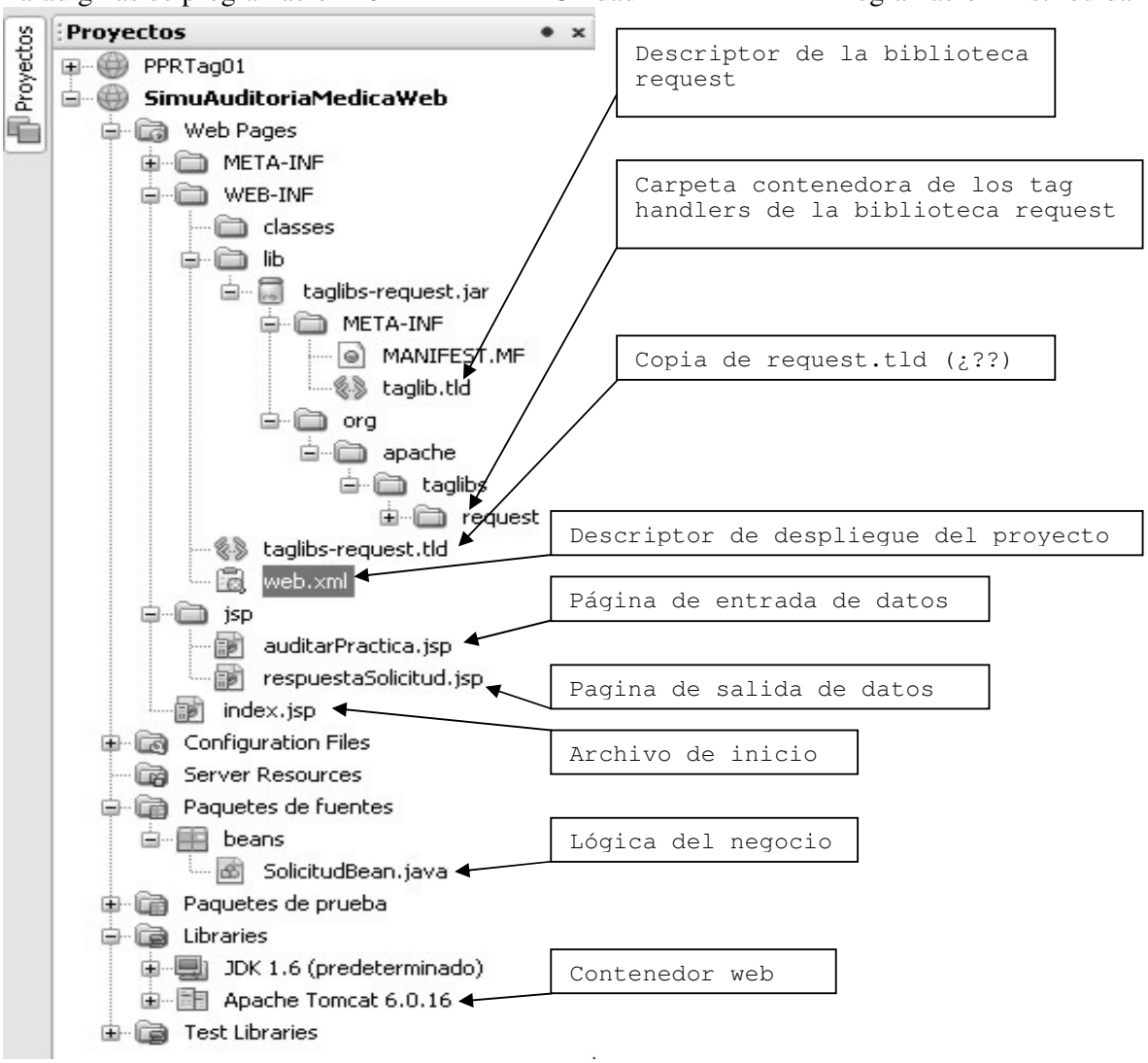
Para completar, un tercer Java Bean, colaboración del Ing. Martín Polliotto

El enunciado dice:

Una importante empresa de la ciudad dedicada a brindar servicios de medicina prepaga nos solicita un sistema para simular el proceso de autorización de prácticas médicas. Básicamente el proceso es el siguiente:

- ✓ Diariamente la empresa recibe pedidos de autorización de prácticas médicas. De cada solicitud (o pedido) de autorización se tienen los siguientes datos:
  - practica (String): nombre o código alfanumérico de la práctica,
  - cantidad (int): a solicitar
  - estado (String): almacena el estado la solicitud. Este puede ser:
    - **Pendiente de Autorización**. Cuando se envía la solicitud al Auditor y aún no se tuvo respuesta.
    - **Autorizada.** Validación exitosa del médico auditor.
    - **Rechazada**. Validación no exitosa del médico auditor.
  - motivo (String): cadena que contiene el motivo de la aceptación o rechazo de la solicitud según corresponda al valor del estado de la misma.

Un capture del proyecto **SimulaAuditoríaMédicaWeb**, entorno NetBeans nos muestra:



Vamos viendo los contenidos:

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
    <taglib>
        <taglib-uri>http://jakarta.apache.org/taglibs/request-1.0</taglib-uri>
        <taglib-location>/WEB-INF/taglibs-request.tld</taglib-location>
    </taglib>
</web-app>
```

index.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>PREPAGA - Autorizar práctica Web</title>
    </head>
    <body>
        <jsp:forward page="/jsp/auditarPractica.jsp"/>
    </body>
</html>
```

/WEB-INF/taglibs-request.tld

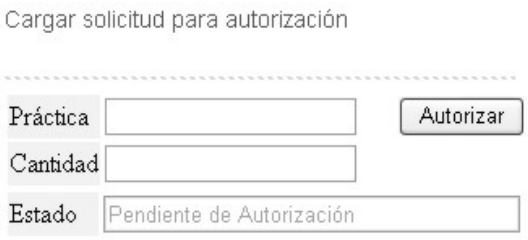
Ya la hemos visto en el proyecto Jakarta.

auditarPractica.jsp

```
<%--
    Document   : auditarPractica
    Created on : 03/07/2009, 15:39:25
    Author     : Martín Pollioto
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>PREPAGA - Autorizar práctica Web</title>
    </head>
    <body>
        <form name="formSolicitud" action="jsp/respuestaSolicitud.jsp"
                                                        method="POST">
            <div style="width:315px">
                <p style="font-family:Arial;color:blue;font-size:14px;">
                  Cargar solicitud para autorización</p>
                <hr style="border-style: dotted; border-color: #D8D8D8; border-
                                                          width: 1px;"></hr>
            </div>
            <table border="0">
                <tbody>
                    <tr>
                        <td bgcolor="#F2F2F2">Práctica</td>
                        <td><input type="text" name="practica" value=""
                                                    id="practica" /></td>
                        <td align="right" width="100px"><input type="submit"
                              value="Autorizar" name="btnAutorizar"/></td>
                    </tr>
                    <tr>
                        <td bgcolor="#F2F2F2">Cantidad</td>
                        <td><input type="text" name="cantidad" value=""
                                                    id="cantidad"/></td>
                    </tr>
                </tbody>
            </table>
            <table>
                <tr>
                    <td bgcolor="#F2F2F2" width="54px">Estado</td>
```

```
                    <td><input type="text" id="estado" name="estado"
                        value="Pendiente de Autorización" disabled="disabled"
                        size="37"/></td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

Un capture de la página generada por auditarPractica.jsp

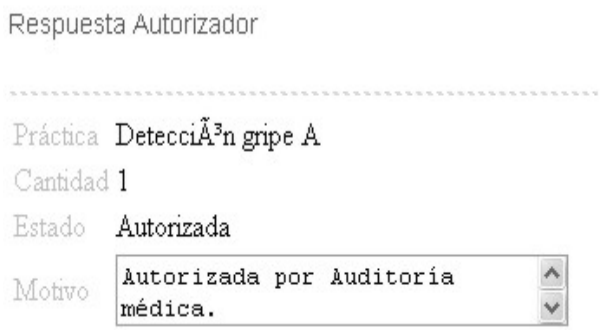


respuestaSolicitud.jsp

```
<%--
    Document   : respuestaSolicitud
    Created on : 03/07/2009, 22:04:42
    Author     : Martin Polliotto
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://jakarta.apache.org/taglibs/request-1.0" prefix="req" %>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>PREPAGA - Autorizar práctica Web</title>
    </head>
    <body>
      <div style="width:340px">
        <p style="font-family:Arial;color:blue;font-size:14px;">Respuesta
                                                          Autorizador</p>
        <hr style="border-style: dotted; border-color: #D8D8D8; border-width:
                                                            1px;"></hr>
      </div>
      <jsp:useBean  id="solicitud" scope="request" class="beans.SolicitudBean">
      <jsp:setProperty name="solicitud" property="cantidad" param="cantidad"/>
      </jsp:useBean>

      <table>
         <tbody>
            <tr>
              <td style="color:#BDBDBD;">Práctica</td>
                    <td><req:parameter name="practica"/></td>
            </tr>
            <tr>
              <td style="color:#BDBDBD;">Cantidad</td>
              <td><req:parameter name="cantidad"/></td>
            </tr>
            <tr>
              <td style="color:#BDBDBD;">Estado</td>
              <td><jsp:getProperty name="solicitud" property =
                                        "estadoSolicitud"/></td>
           </tr>
           <tr>
              <td style="color:#BDBDBD;">Motivo</td>
```

```
                <td><textarea readonly cols="30">
                        <jsp:getProperty name="solicitud" property="motivo"/>
                    </textarea>
                </td>
            </tr>
        </tbody>
    </table>
  </body>
</html>
```

Si lo visualizamos



Por ultimo, la lógica de negocios esta en:

SolicitudBean.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package beans;

/**
 *
 * @author Martin Polliotto
 */
public class SolicitudBean {
    //atributos
    private int cantidad;
    private String motivo;
    private static final String motivosRechazo[]={"Cantidad excedida", "Fuera de
      Cobertura", "Falta análisis"};

    public SolicitudBean(){
    }

    public int getCantidad() {
        return cantidad;
    }

    public void setCantidad(int cantidad) {
        this.cantidad = cantidad;
    }

    public String getMotivo() {
        return motivo;
    }

    public void setMotivo(String motivo) {
        this.motivo = motivo;
    }      /* propiedad adicional (de solo lectura) que permite simular el
           trabajo del * auditor al autorizar o rechazar la solicitud. Determina
           el estado (Autorizada/Rechazada) y el motivo correspondiente en
           función de la  cantidad solicitada...     */
```

```
      public String getEstadoSolicitud(){
        String estado = "";
        int random = (int) (Math.random()*10); // simulamos un valor aleatorio
        int valor = random*cantidad;// determinamos valor

        // determina la aprobación/rechazo de la practica...
        if(valor%2==0)//Si es par autorizamos, impar rechazamos
        {
            estado="Autorizada";
            motivo="Autorizada por Auditoría médica.";
        }
        else
        {
            estado="Rechazada";
            motivo=motivosRechazo[(int)(Math.random()*3)];
        }
        return estado;
    }
}
```

Introducción a las bibliotecas de etiquetas

Ya hemos visto las razones existentes para separar aplicaciones en niveles y utilizar componentes para proporcionar tanto una reutilización eficiente del código como facilidad de mantenimiento. Ha visto cómo crear JavaBeans simples y cómo utilizarlas dentro de una página JSP utilizando etiquetas bean.
Estas etiquetas tuvieron mucho éxito al eliminar el código Java del nivel JSP.

A menudo es necesario tomar decisiones en una página JSP, quizás basadas en la petición hecha por el cliente. Las etiquetas que vimos en el capítulo de java beans no tienen esta capacidad. Estas etiquetas se limitan a interactuar con JavaBeans; no tienen ningún control sobre la página JSP.

Por ejemplo, **no tienen acceso a los objetos request (petición) y response (respuesta)**. En cambio, las bibliotecas de etiquetas a medida sí pueden proporcionar esta funcionalidad. Por eso, a continuación:

- Explicaremos por qué necesitamos bibliotecas de etiquetas.
- Analizaremos los componentes de una biblioteca de etiquetas.
- Explicaremos cómo desplegar y utilizar la biblioteca de etiquetas Request proporcionada por el proyecto Jakarta, como ejemplo.
- Hablaremos de lo que ocurre cuando utilizamos bibliotecas de etiquetas desde dentro de una página JSP.

Comencemos explicando el primer punto.

La necesidad de las bibliotecas de etiquetas

Supongamos que tenemos una página con un frondoso código scriplet. Ya hemos hablado mal de los scriplets, veamos como se reemplaza esto por una etiqueta a medida.

Supongamos también que la aplicación que usa esta página necesita que el cliente especifique el país en el que se encuentra cuando hace una compra online, para calcular correctamente los gastos de envío, por ejemplo.

Si se almacenaran en una base de datos los países del mundo, junto con los gastos de envío asociados a cada uno, podríamos escribir una JSP que buscara los países y nos los mostrara para que el usuario informara el suyo. A continuación mostramos una versión simple de este código. No intente ejecutar el código, sólo trate de comprender lo que está intentando hacer.

```
<%@ page import="java.util.*" %>
<%@ page import="java.awt.*" %>
<%@ page import="java.sql.*" %>
<html>
<body>
    <% String url = "jdbc:odbc:countrydb";
        String user  = "";
        String password = "".
       try {
              Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
              Connection con = DriverManager.getConnection(url, user, password) ;
              Statement stmt = con.createStatement();
              ResultSet rs = stmt.executeQuery("Select * from countries");
              while (rs.next()){
                     String result    rs.getString("name");
                     <%= result %> <br />
               }
              rs.close();
       }catch (Exception e){System.out.println(e);
    %>
</body>
</html
```

¿Lo ha entendido? Que hace este scriplet:
Se conecta con la base de datos countrydb.
se emite una petición de datos.
se usa una colección ResultSet ,
el ciclo while recorre la colección
su objeto rs recibe cada pais de dicha base
los va mostrando de a uno por línea
finalizado el ciclo se cierra se cierra la conexión. Eso es todo

En comparación con el código anterior, observe lo siguiente:

```
<%@ taglib uri="getCountriesTag" prefix="countryTag" %>
<html>
      <body>
            <countryTag:getCountries />
      </body>
</html>
```

Si quisiera reemplazar el código scriptlet por una etiqueta a medida, la página equivalente conducida por dicha etiqueta probablemente sería algo muy similar a las líneas anteriores. De hecho, la sintaxis utilizada en la etiqueta hace que la funcionalidad sea mucho más clara para quien explore la escritura.

Evidentemente, a un diseñador le resulta mucho más sencillo trabajar con este tipo de sintaxis transparente que si intenta acceder a una base de datos utilizando el código Java con el que quizá no esté familiarizado.

La otra importante ventaja es que la etiqueta se puede reutilizar fácilmente, cosa que no se logra utilizando scriptlets, atados a la página .jsp que los contiene.

En definitiva, para un diseñador de páginas tiene sentido tener acceso a un grupo de etiquetas que se parezcan a los marcadores HTML con los que está acostumbrado a trabajar.

¿Por qué no podemos utilizar simplemente JavaBeans?

Podría estar pensando, ¿por qué no podemos hacer todo esto utilizando simplemente JavaBeans? La respuesta es que, en la mayoría de las ocasiones:
JavaBeans son de hecho el enfoque que debería utilizar.
Puede utilizar JavaBeans para agrupar partes de similar funcionalidad.
Sin embargo, los JavaBeans no pueden interactuar con la página JSP: (no tienen acceso a los objetos request y response) no pueden utilizarse para los procesos de tomas de decisiones. (Sentencias de control)

Supongamos un caso: Podría ocurrir que tuvieramos algunos datos en una matriz que el diseñador necesitara usar para producir una página JSP. Por ejemplo, podríamos tener una matriz de nombres de productos mostrados, dependiendo de si el usuario ha comprado antes un producto similar. Si tratáramos de hacer esto utilizando etiquetas bean terminaríamos incorporando código de presentación al bean, y justamente una de las principales razones para introducir beans, que reemplazaban a los scriplets, era evitar esta mezcla de lógica de presentación y de negocios.

Las bibliotecas de etiquetas a medida (etiquetas de funcionalidad afín agrupadas) nos permiten eliminar todo el código Java de la página JSP. Esto significa que un diseñador Web que no sepa Java puede mantener un sitio Web basado en JSP.

Una vez que hemos comprendido las ventajas que representan la utilización de una biblioteca de etiquetas, pasemos a analizar las partes que componen dicha biblioteca.

Dentro de una biblioteca de etiquetas

No pretendemos construir una biblioteca de etiquetas propia, pero si es importante que entendamos de qué se compone una biblioteca de etiquetas, para poder utilizar una.

Hay tres partes importantes en la comprensión de cómo utilizar una biblioteca de etiquetas:

- Clases de manipuladores de etiquetas, proporcionan la funcionalidad de las etiquetas.
- Un descriptor de la biblioteca de etiquetas (TLD), describe las etiquetas y hace coincidir las etiquetas con las clases de manipuladores de etiquetas.
- La directriz taglib, ésta es una directriz situada en la parte superior de su JSP, que le permite utilizar una biblioteca de etiquetas en particular.

Estudiemos cada uno de estos componentes.

Los manipuladores de etiquetas

Un manipulador de etiquetas(Tag handler) es un tipo de clase especial que contiene el código que ejecuta una etiqueta; es decir que la funcionalidad de la etiqueta se encuentra dentro del manipulador de etiquetas. (Como los métodos que vimos en AED, con funcionalidad definida, se encuentran dentro de clases específicas).

A diferencia de los JavaBeans, que son conjuntos de funciones comunes, un manipulador de etiquetas tiene una finalidad estrictamente definida, debe responder a una única etiqueta. Una biblioteca de etiquetas tiene su manipulador de etiquetas para cada etiqueta a medida.

No necesita saber cómo implementar un manipulador de etiquetas para utilizar bibliotecas de etiquetas a medida.

Nota: clarificando un poco. No estamos presentando nada que el alumno no conozca, cuando hablamos del encapsulamiento en Java, vimos que había el nivel del programador de aplicaciones, que usaba métodos de clases ya listas, y del desarrollador de clases, un nivel superior, que las codificaba. Aquí estamos en la misma división del trabajo, el diseñador de páginas, cuando utiliza una etiqueta en una página JSP necesita conocer de que etiquetas dispone y su funcionalidad, saber hacerlas es incumbencia de otro profesional.

El motor JSP necesita saber qué clases de manipuladores de etiquetas corresponden a unas etiquetas determinadas en una página]SP, y qué métodos llamar en esas clases. Esta información se almacena en un descriptor de la biblioteca de etiquetas (TLD), que explicaremos a continuación.

El descriptor de bibliotecas de etiquetas

Un descriptor de bibliotecas de etiquetas (TLD) es un documento XML que contiene información sobre una o más etiquetas a medida. Un TLD es un archivo que describe las etiquetas a medida y las relaciona con sus clases de manipuladores de etiquetas.

El lenguaje de marcado extensible (XML) es un conjunto de reglas para describir datos (proporcionando etiquetas que describen la naturaleza de los datos) de forma neutral respecto a la plataforma e independiente respecto al lenguaje.

Como norma general, un archivo TLD se denomina como la clase a la que se refiere. Debe ser guardado con extensión .tld y almacenado en el directorio WEB – INF de la aplicación Web.

Las bibliotecas de etiquetas existentes vendrán con un archivo TLD, por lo que mientras no escriba sus propias bibliotecas de etiquetas no necesita entender los contenidos del archivo TLD, simplemente tiene que comprender lo que hace.

La directriz taglib

Para utilizar las etiquetas de una biblioteca de etiquetas a medida en sus páginas JSP, debe añadir una directriz taglib en la parte superior se su página JSP. Por ejemplo:

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/request-1.O"
prefix="req" %>
```

Se necesita dicha directriz para cada biblioteca de etiquetas que desee utilizar en su página JSP, y los atributos uri y prefix tienen valores específicos para cada biblioteca de etiquetas.

El valor de un atributo uri identifica la biblioteca de etiquetas. A menudo consta de una URL de la organización que mantiene la biblioteca de etiquetas. En realidad, el sistema no intenta acceder a la URL; únicamente existe para identificar la biblioteca de etiquetas. La utilización de una URL simplemente ayuda a documentar sobre dónde se originó la biblioteca de etiquetas y para garantizar que tiene un nombre único. Sin embargo, URI puede ser una ruta de acceso con un nombre de archivo añadido.

En este caso, el sistema asume que este archivo es su archivo TLD e intenta cargarlo. Si la ruta de acceso no apunta al archivo TLD, el sistema buscará el archivo TLD en otro sitio de la aplicación Web.

Siempre que un motor JSP encuentra una extensión de etiquetas en una JSP, mira el valor del atributo uri en la directriz taglib y después analiza sintáctica mente el descriptor de la biblioteca de etiquetas para encontrar la clase de manipulación de etiquetas requerida. Entonces genera un código para que interactúe con el manipulador de etiquetas y le permita a su JSP utilizar las etiquetas.

El valor del atributo prefix se utiliza para identificar una etiqueta en una página JSP como parte de una biblioteca de etiquetas en particular. De esta forma, si tenemos dos etiquetas con el mismo nombre, pero procedentes de distintas bibliotecas de etiquetas, es posible identificarlas de forma única. Entonces, si tuviéramos una etiqueta con el nombre myTag y un prefijo myPrefix, la etiqueta se identificaría de la siguiente forma:

```
<myPrefix:myTag/>
```

No existen restricciones sobre lo que puede utilizar como valor de prefix, excepto que no puede utilizar jsp, jspx, java, javax, servlet, sun, o sunw, ya que son palabras reservadas.

Fijemos ahora estos conceptos con un ejemplo.

Utilización de una biblioteca de etiquetas

Vamos a utilizar una de las bibliotecas de etiquetas creadas por las personas que construyeron Tomcat, el proyecto Jakarta.

El proyecto Jakarta ha creado un gran número de bibliotecas de etiquetas a medida. La que vamos a utilizar es la Biblioteca de Etiquetas Request(de petición). Esta Biblioteca de Etiquetas Request contiene etiquetas que pueden ser utilizadas para acceder a toda la información sobre la petición http para una página JSP. Ya hemos utilizado el objeto request para conseguir información del cliente.
Ahora veremos cómo puede proporcionar la misma funcionalidad en sus páginas JSP sin tener que saber ni una sola línea de Java.

Un par de Biblioteca de Etiquetas disponibles

Puede descargarse la última entrega de la Biblioteca de Etiquetas Request y DateTime del sitio Web de Yakarta

http://jakarta.apache.org/site/downloads/, en downloads opte por taglibs, de la página de Taglib Downloads baje Request Taglib y DateTime Taglib (en Keys opte por el archivo .zip) a una carpeta adecuada, por ejemplo Yakarta-Taglibs.

Descomprima los archivos allí mismo. Tendremos 2 carpetas

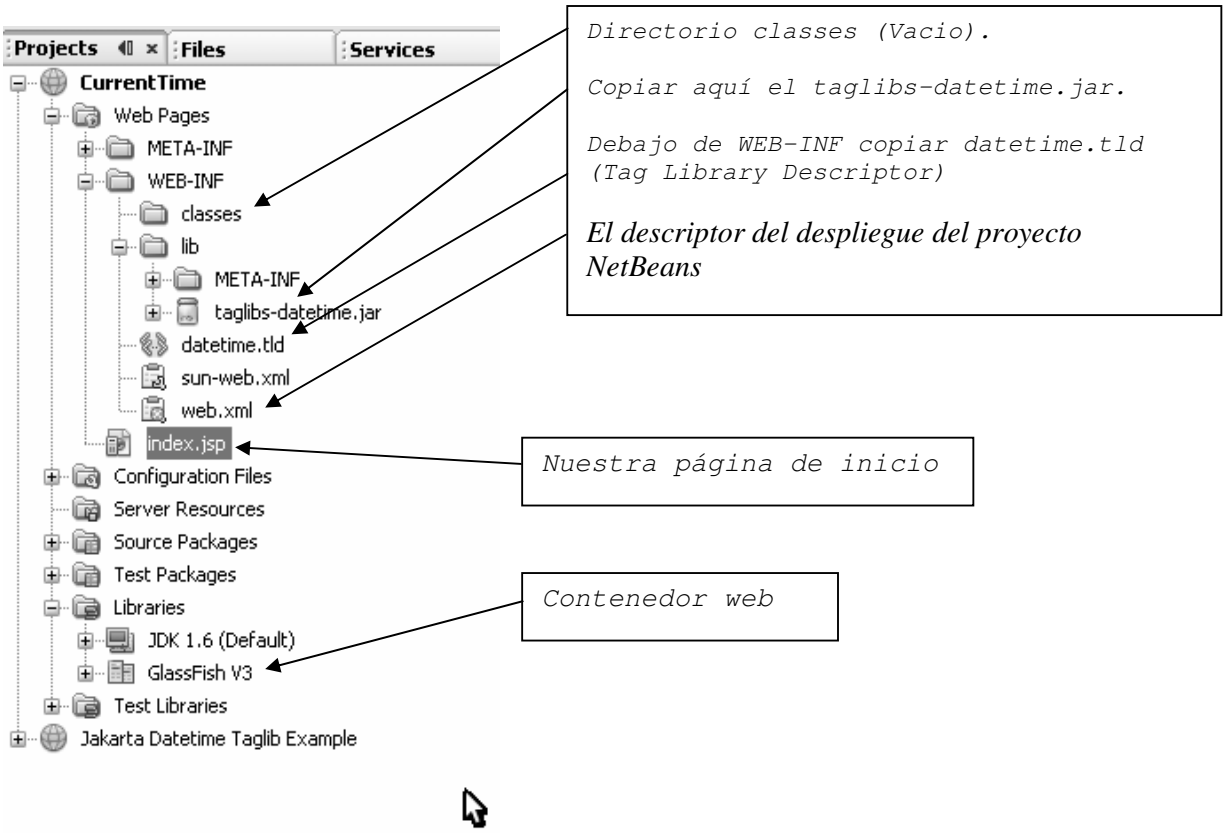
Si vemos el contenido de la carpeta Jakarta-Taglibs-Datetime-1.0.1 datetime.jar, contiene las clases de manipuladores de etiquetas datetime.tld, es el archivo descriptor para la biblioteca.
datetime-doc.war, contiene documentación.
datetime-examples.war, ejemplo de una aplicación Web

Si editamos el datetime.tld:

jakarta.apache.org/taglibs/datetime-1.0

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
  <tlibversion>1.0.1</tlibversion> <jspversion>1.1</jspversion>
  <shortname>datetime</shortname>  <uri>http://</uri>
  <info>
```

```
      The DATETIME custom tag library contains tags which can be used to handle
  </info>
  <tag>
    <name>currentTime</name>
    <tagclass>org.apache.taglibs.datetime.CurrentTimeTag</tagclass>
    <bodycontent>empty</bodycontent>
  </tag>
  <tag>
    <name>format</name> la descripción de format sigue con sus atributos…
Otras etiquetas descriptas en este archivo:
Parse, timeZone, timeZones, months, weekdays, amPms, eras,
```



Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns=http://java.sun.com/xml/ns/javaee
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <session-config> <session-timeout> 30 </session-timeout> </session-config>
    <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

index.jsp

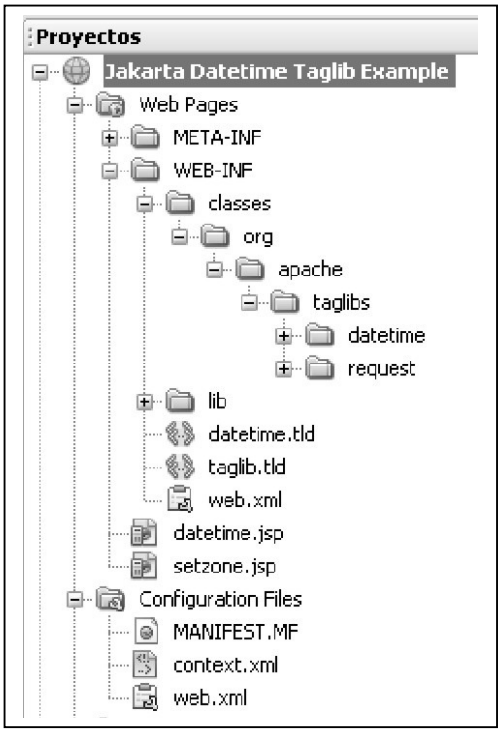
```
<%--
    Document   : index
    Created on : 13-may-2009, 18:48:58
    Author     : Tymos
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
   <%@taglib uri="http://jakarta.apache.org/taglibs/datetime-1.0" prefix="dt" %>
```

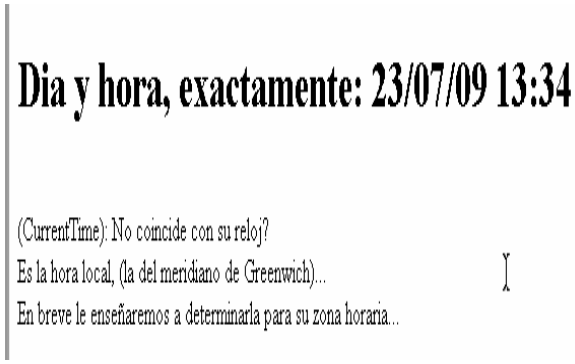
```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Proyecto CurrentTime, etiqueta currentTime, biblioteca dateTime
        </title>
    </head>
    <body>
        <h1>Dia y hora, exactamente:
            <dt:format>
                <dt:currentTime/>
            </dt:format>
        </h1>
        <br>
        (CurrentTime): No coincide con su reloj?
        <br>
        Es la hora local, (la del meridiano de Greenwich)...
        <br>
        En breve le enseñaremos a determinarla para su zona horaria...
    </body>
</html>
```

Correr el proyecto

En el menú tag de NetBeans <Ejecutar> <Clean and Build main Project>
Si la compilación no detectó errores, vamos al proyecto, posicionamos cursor sobre index.jsp <boton derecho> <Run File>

Esto es lo que deberíamos ver:





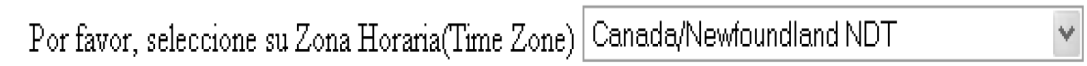
A continuación un ejemplo que es una adaptación de una aplicación Web tomada de datetime-examples.war, Proyecto Yakarta.

Datetime.jsp

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <title>Jakarta DATETIME Taglib Example(Entrada)</title>
</head>
<body bgcolor="#FFFFFF">
 <center><h1>Ejemplo con las Bibl.cas Jakarta DATETIME y REQUEST  </h1>
 </center>
<br>

<%@ taglib uri="http://jakarta.apache.org/taglibs/datetime-1.0"
prefix="dt" %>
<p>
<form action="setzone.jsp">
A continuación presentamos una lista de selección usando la etiqueta
<b>timezone</b>
<br>
<br>
Por favor, seleccione su Zona Horaria(Time Zone)
<select name="timezone">
    <dt:timeZones id="tz">
    <option value="<jsp:getProperty name="tz" property="zoneId"/>">
        <jsp:getProperty name="tz" property="zoneId"/>
        <jsp:getProperty name="tz" property="displayName"/>
    </dt:timeZones>
</select>
<br>
<br>
Y ahora, listas de selección usando las etiquetas <b>months, day, year</b>
<br>
<br>
Mes:
<select name="month">
    <option value="Select">Select
    <dt:months id="mon" locale="true">
        <option value="<jsp:getProperty name="mon"
property="monthOfYear"/>">
        <jsp:getProperty name="mon" property="month"/>
    </dt:months>
</select>
Día del mes:
<select name="day">
    <option value="Select">Select
    <option value="01">1
    <option value="02">2
    <option value="03">3
    <option value="04">4
      … muchos …
    <option value="30">30
    <option value="31">31
</select>
Año:
<select name="year">
    <option value="Select">Select
    <option value="1997">1997
    <option value="1998">1998
      … muchos …
    <option value="2009">2009
    <option value="2010">2010
</select>
<br>
<br>
```

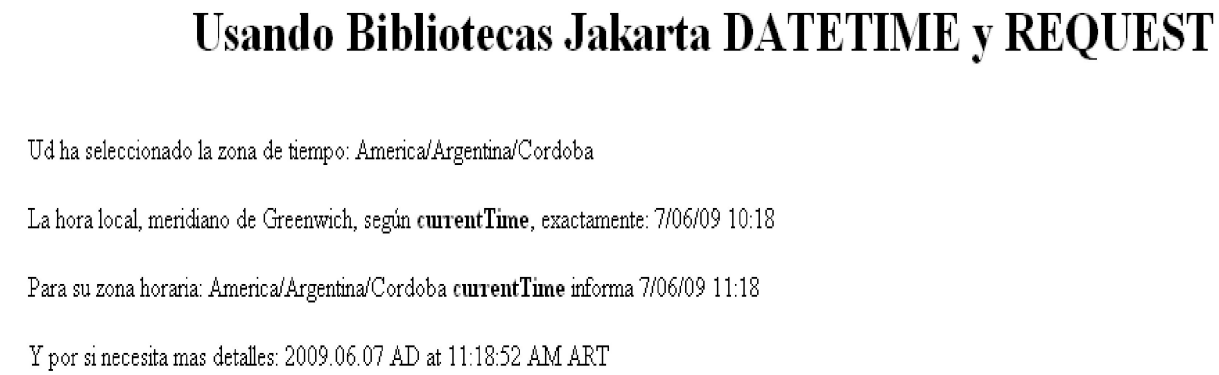
```
En que día de la semana estamos? (Etiqueta <b>weekdays</b>), por favor:
<select name="weekday">
   <option value="Select">Select
      <dt:weekdays id="day" locale="true">
        <option value="<jsp:getProperty name="day"
property="shortWeekday"/>">
        <jsp:getProperty name="day" property="weekday"/>
      </dt:weekdays>
</select>
<br>
<br>
Es de mañana o tarde? (Etiqueta <b>amPms</b>):
<select name="ampm">
   <option value="Select">Select
   <dt:amPms id="ampm" locale="true">
      <option value="<jsp:getProperty name="ampm" property="name"/>">
      <jsp:getProperty name="ampm" property="name"/>
   </dt:amPms>
</select>
<br>
<br>
En que era vivimos? (etiqueta <b>eras</b>):
<select name="era">
  <option value="Select">Select
  <dt:eras id="era" locale="true">
    <option value="<jsp:getProperty name="era" property="name"/>">
    <jsp:getProperty name="era" property="name"/>
  </dt:eras>
</select>
<br>
<br>
Click en <input type="submit" name="submit" value="submit"> para ver el
resultado que la página setzone.jsp nos devuelve, usando la zona horaria y fecha
informados.
</form>
<br>
<br>
</body>
</html>
```

Un par de captures mostrando funcionamiento



(No consigo capturar las listas desplegables, …)

Si después de una adecuada selección capturamos la página retornada:



Analizamos como obtenemos la zona horaria y hora correspondiente.
Vemos primero como se describe la tag timeZone en datetime.tld

```
<tag>
  <name>timeZone</name>
  <tagclass>org.apache.taglibs.datetime.TimeZoneTag</tagclass>
  <bodycontent>JSP</bodycontent>
  <attribute>
    <name>id</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

```
<dt:timeZone id="tz">
    <req:parameter name="timezone"/>
</dt:timeZone>
```

y que interpretamos?

- La descripción del tag dice que la etiqueta **timeZone**
  - Su cuerpo es tipo JSP
  - tiene un atributo,
  - cuyo nombre es **id**
  - de carácter obligatorio
  - y no requiere evaluación en tiempo real.

- En la invocación
  - Apertura: definimos que el atributo **id**, de tipo timeZone de la biblioteca cuyo prefijo es **dt**(datetime) tendrá como nombre **"tz"**.
  - Cuerpo: "tz" recibe el valor que la etiqueta **parameter** de la biblioteca **req**(request) extrae del objeto request, atributo timezone, el cual fué seleccionado por el usuario antes de clickear submit. Se ha configurado un par nombre/valor
  - cierre

como se describe la tag **format** en datetime.tld

```
<tag><name>format</name>
 <tagclass>org.apache.taglibs.datetime.FormatTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
     … pattern, patterned …
    <attribute>
      <name>timeZone</name>
      <required>no</required>
      <rtexprvalue>no</rtexprvalue>
    </attribute>
    … date, default, locale, localeRef …
</tag>
```

```
Para su zona horaria: <req:parameter name="timezone"/>
<b>currentTime</b> informa
<dt:format timeZone="tz">
    <dt:currentTime/>
</dt:format>
```

y que interpretamos?

- La descripción del tag dice de la etiqueta format:
  - Su cuerpo es JSP
  - tiene 7 atributos,
  - ninguno es obligatorio (Tienen valores default)
  - (detallamos el atributo usado: timeZone)

- En la invocación
  - **currentTime informa** son ctes literales
  - Apertura: definimos que el atributo **id**, de tipo timeZone de la biblioteca cuyo prefijo es **dt**(datetime) será **"tz"**, que ya trae asociado su valor.
  - Cuerpo: Se invoca la etiqueta currentTime de datetime. El retorno de esta invocación es el parámetro de entrada de format, que así sabe de que lugar del mundo es el usuario que quiere saber la hora.
  - Cierre

setzone.jsp completa

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
   <title>Jakarta DATETIME Taglib Example(Respuesta)</title>
</head>
<body bgcolor="#FFFFFF">

<center>
    <h1>        Usando Bibliotecas Jakarta DATETIME y REQUEST
    </h1>
</center>
<br>
<%@ taglib uri="http://jakarta.apache.org/taglibs/datetime-1.0" prefix="dt" %>
<%@ taglib uri="http://jakarta.apache.org/taglibs/request-1.0" prefix="req" %>
<p>
Ud ha seleccionado la zona de tiempo:
    <req:parameter name="timezone"/>
<br>
<br>

La hora local, meridiano de Greenwich, según <b>currentTime</b>, exactamente:
<dt:format>
    <dt:currentTime/>
</dt:format>
<br>
<br>

<% /* Insertamos aquí un comentario de mas de una linea
    para decir que en lo que sigue estamos determinando
    la zona horaria informada en datetime.jsp, para luego
    obtener la hora que allí es.
*/%>

<dt:timeZone id="tz">
    <req:parameter name="timezone"/>
</dt:timeZone>

Para su zona horaria: <req:parameter name="timezone"/>
 <b>currentTime</b> informa
<dt:format timeZone="tz">
    <dt:currentTime/>
```

```
</dt:format>
<br><br>
Y por si necesita mas detalles:
<dt:format timeZone="tz" pattern="yyyy.MM.dd G 'at' hh:mm:ss a zzz">
   <dt:currentTime/>
</dt:format>
<br>
<br>
</body>
</html>
```

Cómo escribir bibliotecas de etiquetas

Acabamos de ver como se puede, mediante bibliotecas de etiquetas a medida (Standard) incluir funcionalidad en nuestras páginas JSP, sin llenarlas de complicados scriptlets. Esto hace que su página sea más fácil de mantener y puede proporcionarle una biblioteca de componentes reutilizables, un principio básico en la programación orientada a objetos.

Permanentemente se están creando una variedad de bibliotecas de etiquetas, gratuitamente pero claro, sólo cubren las funciones más comunes.

Es importante saber como crear nuevas etiquetas para las necesidades específicas de su aplicación Web.

Una biblioteca de etiquetas personalizada

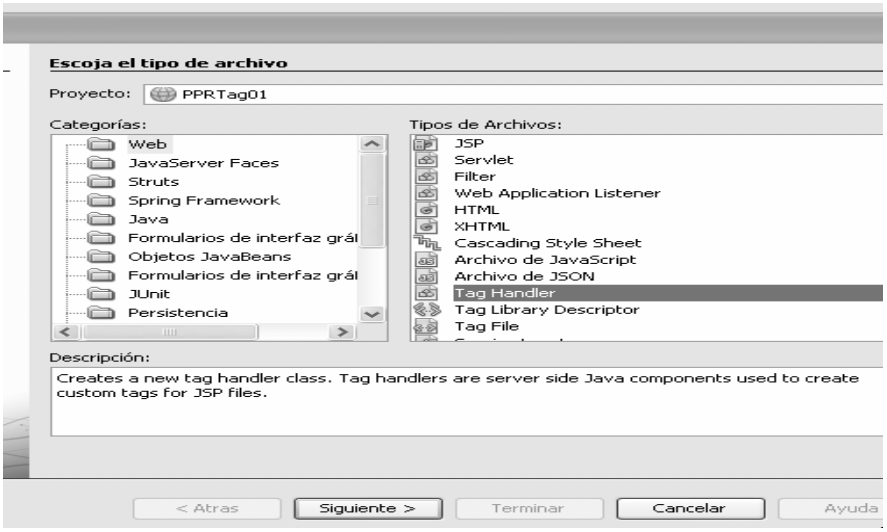
Ya vimos que una biblioteca de etiquetas se compone de dos elementos principales:

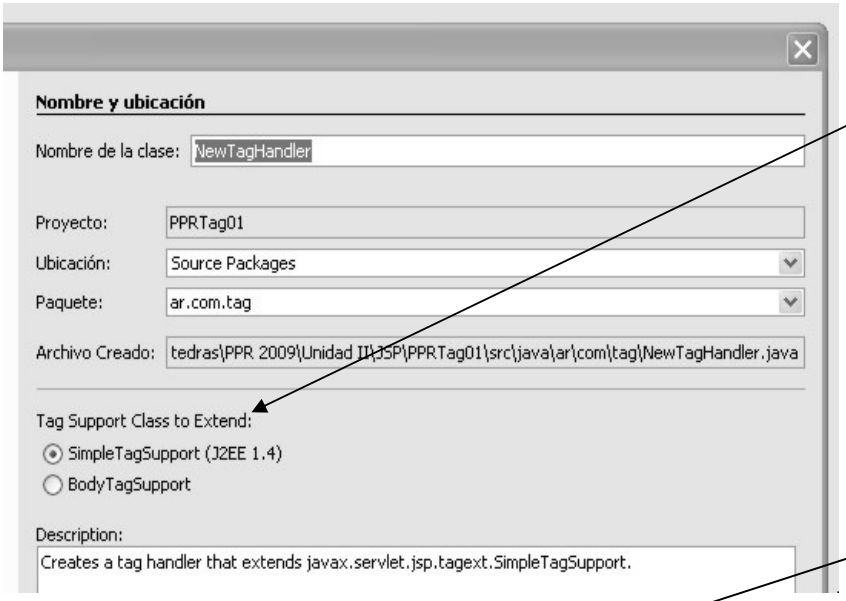
- Un archivo descriptor de la biblioteca de etiquetas, (**.tld**) que presenta las bibliotecas de etiquetas a sus clases de manipulación de etiquetas adecuadas. Hemos visto parcialmente datetime.tld
- Las clases de manipulación de etiquetas, (**Tags handlers**) contienen el código que ejecutan las etiquetas. Hemos trabajado con datetime.jar y al extraer su contenido obtuvimos una estructura de carpetas, debajo de la última (datetime) se encuentran 14 archivos .class. Veamos los pasos.

Construcción de una etiqueta personalizada

Se debe crear una clase de manipulador de etiquetas (**tag handler**) y presentarla a su correspondiente etiqueta personalizada utilizando un archivo descriptor de bibliotecas de etiquetas (**MisTags.tld**, por ejemplo).

Usando NetBeans, la secuencia de pasos para crear un tag handler:
Archivo, Archivo Nuevo, **Web**, **Tag Handler** <**Siguiente**>





Aquí se nos ofrecen 2 alternativas:

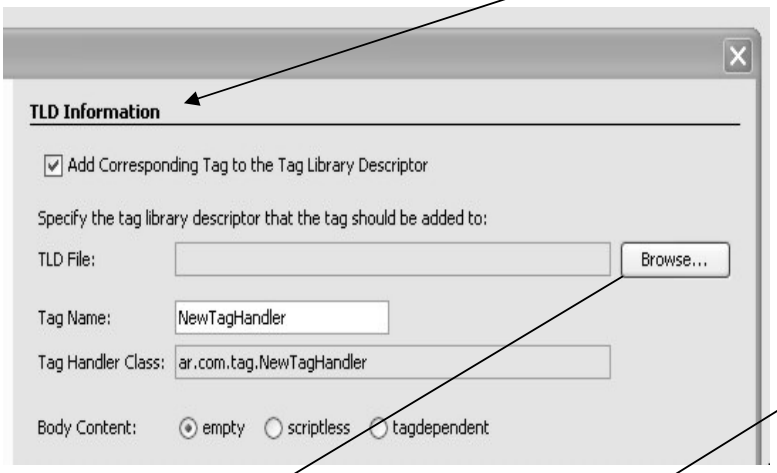
SimpleTagSupport:
2 interfaces,
8 métodos

Adecuado para aplicaciones simples

BodyTagSupport:
5 interfaces
8 métodos

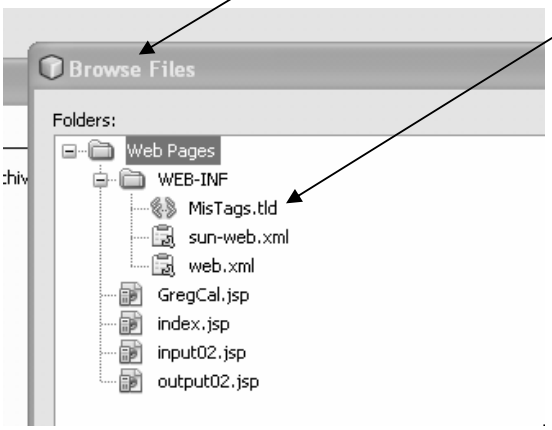
Mucho más potente

<siguiente>



Definimos Body Content: **empty** (Vacio)

Debemos <**Browse**> para especificar nuestro .tld



Seleccionamos **MisTags.tld**

(Tome esto como una introducción, cuando desarrollemos proyectos usando tags vamos ir paso por paso, UD. verá que lo primero es justamente definir el descriptor de biblioteca Mistags,tld)

Supongamos que hubiésemos seleccionado **BodyTagSupport**

Estos manipuladores de etiquetas implementan 5 interfaces y para cumplimentarlas contienen 8 métodos, siendo los más importantes **doStartTag()** y **doEndTag()**.

Cuando se lee la etiqueta de apertura (por ejemplo **<example:time>**, el contenedor llama al método **doStartTag()** del manipulación de etiquetas(Tag Handler).

Cuando se lee la correspondiente etiqueta de cierre **</example:time>**, el contenedor llama al método **doEndTag()**.

Por lo general, crear un manipulador de etiquetas no es nada más que definir un código personalizado para que sea ejecutado por cada uno de estos métodos.

Para asegurarse de que el objeto manipulador de etiquetas contiene todos los métodos necesarios, debe implementar una interfaz de manipuladores de etiquetas. La especificación JSP 1.2 proporciona interfaces de manipuladores de etiquetas:

- **Tag**
- **IterationTag**
- **BodyTag**
- **Serializable**
- **JSPTag**

La interfaz Tag es la **interfaz de manipuladores de etiquetas básica** que define los métodos requeridos por todos los manipuladores de etiquetas. La interfaz define seis métodos:

- doStartTag(),
- doEndTag(),
- getParent(),
- setParent(),
- release() y
- setPageContext()

Si necesita implementar sólo uno o dos de estos métodos en su clase de manipuladores de etiquetas, existe una clase de ayuda, **TagSupport**, que ya implementa la interfaz Tag. Si su clase de manipulador de etiquetas extiende TagSupport, sólo tenemos que implementar los métodos que necesitamos para conseguir nuestro objetivo.

Cuando extendemos la clase TagSupport, los dos métodos que normalmente rescribimos se relacionan con la JSP que lee la etiqueta a medida.

Al llegar a la **etiqueta de apertura**, se llama al método **doStartTag();** el valor que devuelve este método determina lo que se hace después. Los valores válidos devueltos incluyen:

- **EVAL_BODY_INCLUDE** hará que se evalúe el contenido del cuerpo de la etiqueta y que esté disponible para su uso en el manipulador de etiquetas.
- **SKIP_BODY**, se salta el cuerpo de la etiqueta, por lo que debería ser utilizado en etiquetas que no tuvieran cuerpo o en etiquetas en las que no quisiéramos evaluar el cuerpo.
- **doStartTag()** puede devolver mas valores válidos.

Al llegar a la **etiqueta de cierre**, se llama a **doEndTag()**. Si tiene una etiqueta vacía, este método sigue siendo llamado, pero será llamado justo después de **doStartTag()**. Valores válidos de devolución son

- **EVAL_PAGE** la ejecución de la JSP continúa normalmente después de la etiqueta a medida.(Podemos tener mas etiquetas en esta página)
- **SKIP_PAGE** la Ejecución DE LA JSP se detendrá después de evaluar la etiqueta a medida y se saltará el resto de su JSP, si lo hubiere.

Si hubiéramos optado por SimpleTagSupport (Usaremos este), se implementan 2 interfaces y se necesitan también 8 métodos.

El archivo descriptor de bibliotecas de etiquetas

Los manipuladores de etiquetas se vinculan a las etiquetas personalizadas. El mecanismo de unión es un archivo descriptor de bibliotecas de etiquetas (TLD). Ya vimos un ejemplo en la sección anterior, al describir **datetime.tld**

Un TLD (**Tag Library Descriptor**) es un documento tipo XML que describe su biblioteca de etiquetas a medida (personalizada). No se preocupe si no ha utilizado XML antes. Introduciremos las distintas partes del TLD utilizadas según aparezcan.

La primera parte de un TLD es siempre igual. Es un encabezamiento. Para amoldarse a un documento XML la primera línea es siempre la declaración XML.

```
<?xml version="l.O" encoding="ISO-8859-1"?>
```

La siguiente línea comienza la etiqueta que une el documento XML con una definición y un mecanismo de validación para un TLD. Esto es igual para todos los archivos TLD de JSP 1.2

```
<DOCTYPE taglib PUBLlC
      "-//Sun Microsystems, lnc.//DTD JSP Tag Library 1.2//EN"
            "http://java.sun.com/dtd/Web-jsptaglibrary_l_2.dtd">
```

El elemento raíz definido para un descriptor de bibliotecas de etiquetas es **<taglib>**.

El elemento **<taglib>** debe aparecer después de los dos encabezamientos predeterminados y debe circundar el resto de los elementos que describen nuestra biblioteca de etiquetas.

Dentro del elemento **<taglib>** debe incluir algunos otros elementos para describir su biblioteca de etiquetas. Una lista de subelementos <taglib> importantes que debe especificar son:

Elemento	Descripción
<tlib-version>	La versión de la implementación de la biblioteca de etiquetas. Es arbitraria y establecida por los desarrolladores de una biblioteca de etiquetas.
<jsp-version>	La versión de la especificación JSP de la que depende la biblioteca de etiquetas.
<short-name>	Un nombre corto y fácil de recordar para esta taglib.
<uri>	Un URI que identifica de forma única esta taglib.
<display-name>	Un nombre corto para taglib, que se supone será mostrado por las herramientas.
<tag>	Soporta información de una única etiqueta en esta biblioteca de etiquetas. Si tiene más de una etiqueta, utilice tantos elementos <tag> como necesite.
<description>	Una cadena que describe el propósito de la biblioteca de etiquetas.

Veamos la información especificada en el elemento <tag>.

El elemento < tag>

Cada etiqueta a medida debe tener un elemento <tag> que la una a un manipulador de etiquetas a medida. Todos los elementos <tag> deben contener, al menos, los dos elementos **<name> y <tag-class>** y dos etiquetas no pueden tener el mismo nombre. Por ejemplo:

```
<taglib>
      <tag>
            <name>time</name>
            <tag-class>com.wrox.chlO.timeTag</tag-class>
      </tag>
</taglib>
```

El elemento **<tag>** tiene sub elementos adicionales para más información; los más importantes:

Elemento	Requerido	Descripción
name	Sí	Un nombre único para la etiqueta. Este nombre se corresponde biunívoca mente con el elemento name de la etiqueta personalizada en su JSP.
tag-class	Si	La clase de manipuladores de etiquetas que implementa la interfaz Tag, IterationTag, o BodyTag.
tei-class	Si	Una clase opcional de Información Extra de Etiquetas (Tag Extra Info)
display-name	Si	Un nombre corto para la etiqueta, que se supone será mostrado por las herramientas.
Description	No	Una breve descripción de la etiqueta.
Attribute	No	Información sobre un atributo utilizado por la etiqueta. Si la etiqueta utiliza más de un atributo, utilice tantos elementos <attribute> como sea necesario.
variable	No	Crea una variable scripting que puede ser utilizada por la etiqueta

Sabemos lo básico sobre lo que tenemos que hacer, construyamos una etiqueta a medida (custom tag) simple para nosotros. Por ejemplo, que muestre la hora actual (En formato usual) en una JSP. Para ello:

Paso 1) – Crearemos un proyecto que llamaremos **PPRTag01.**

Archivo, Proyecto Nuevo, Java Web, Web Application **<Siguiente>**.
Project Name: **PPRTag01;**
Project Folder, Project Location, aceptar las sugeridas.
Use Dedicated Folder for Storing Libraries: No
Set as Main Project: Si **<Siguiente>**

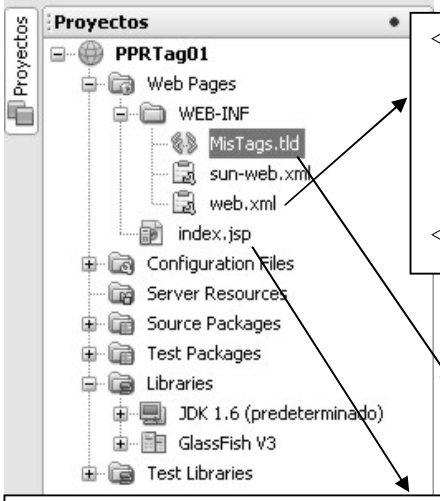
Add to Enterprice Application: None
Server: GlassFish V3
Use dedicated Library for folder for server jar files: No
Java EE version: Jave EE 5
Context Path: /PPRTag01 **<Terminar>**

Paso 2) – Creamos una .tld (Tag Library Descriptor).

Allí describiremos tags.

Archivo, Archivo Nuevo, Web, Tag Library Descriptor
TLD Name: **MisTags**
Proyect, Location, Folder, Created File, URI, prefix: **Aceptar los valores sugeridos**. (queda URI: **/WEB-INF/MisTags** y prefix: **mistags**)

Una imagen del proyecto **PPRTag01** en este punto y sus partes importantes




```
<web-app version="2.5"
   <session-config><session-timeout>30</session-
                                        timeout>
   </session-config>
   <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
</web-app>
```

```
<taglib version="2.0"
  <tlib-version>1.0</tlib-version>
   <short-name>mistags</short-name>
   <uri>/WEB-INF/MisTags</uri>
</taglib>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
    </body>
</html>
```

Paso 3) – Crear el tag handler de la etiqueta.

File, New File, Web, Tag Handler,<Next>
- Class Name: **BadTimeTag**
- Location: Source package
- Package: ar.com.tag
- Tag Support Class to extend: SimpleTagSupport, **<Next>**

TLD Information
- Add corresponding information to Tag Library Description: Si, tildar
- TLD File: <Browse>, seleccionar MisTags.
- Body Content: Empty **<Finish>**
- Vemos que en el proyecto PPRTag01 aparece, debajo de Source Packages /ar.com.tag/BadTimeTag.java.

```
package ar.com.tag;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 *
 * @author Tymos
 */

public class BadTimeTag extends SimpleTagSupport {

    /**
     * Called by the container to invoke this tag.
     * The implementation of this method is provided by the tag library
     *developer, and handles all tag processing, body iteration, etc.
     */
    @Override
    public void doTag() throws JspException {
        JspWriter out = getJspContext().getOut();
```

```
        try {
            // TODO: insert code to write html before writing the body content.
            // e.g.:
            //
            // out.println("<strong>" + attribute_1 + "</strong>");
            // out.println("    <blockquote>");

            JspFragment f = getJspBody();
            if (f != null) f.invoke(out);

            // TODO: insert code to write html after writing the body content.
            // e.g.:
            //
            // out.println("    </blockquote>");

        } catch (java.io.IOException ex) {
            throw new JspException("Error in BadTimeTag tag", ex);}
    } //    doTag()
}     //    BadTimeTag
```

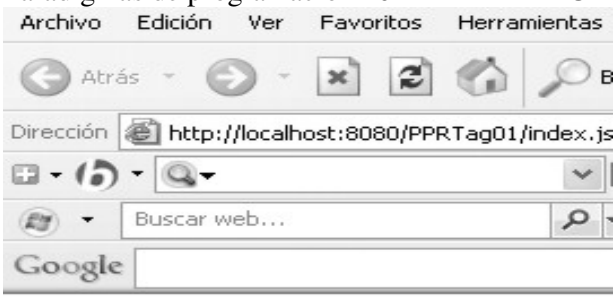
Verificamos MisTags.tld (NetBeans habrá incluido la etiqueta?)

```
<taglib> ……
  <tag>
    <name>BadTimeTag</name>
    <tag-class>ar.com.tag.BadTimeTag</tag-class>
    <body-content>empty</body-content>
  </tag>
</taglib>
```

Queremos que **BadTimeTag** nos informe la hora. Completemos la codificación.

Paso 4) – Modificamos **index.jsp**, para usar nuestra tag, **en negrita**:

```
<%--
    Document   : index
    Created on : 06-jul-2009, 20:26:52
    Author     : Tymos
--%>
<%@taglib prefix="mistags" uri="/WEB-INF/MisTags"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Mi biblioteca de tags</title>
    </head>
    <body>
        Resultado de la invocación a <br>
        la etiqueta BadTimeTag de la biblioteca<br>
        identificada por el prefijo mistags<br>
        <b><mistags:BadTimeTag></mistags:BadTimeTag></b><br>
        (claro, en el meridiano de Greenwich)
    </body>
</html>
```



Si consultamos el reloj, vemos que la hora informada está **Exactamente** atrasada 1 hora.

Será que no podemos dar una hora mas adecuada?

Si, y ya lo hemos hecho usando la biblioteca datetime en nuestro proyecto Yakarta Datetime Taglib Example.

Pero no lo podemos hacer nosotros? Eso de depender para todo de terceros…
Investiguemos …

En el proyecto Yakarta Datetime Taglib Example seleccionamos una zona horaria: "America/Argentina/CordobaART". Eso debe ser parámetro de algún método en una clase TimeZone o algo así… Investiguemos, veamos el Help…
Investiguemos GregorianCalendar. En el help hay un ejemplo muy completo.
En el **mismo proyecto PPRTag01**, construyamos otra custom tag, la llamemos **GregCalTag**, la incluiremos en **MisTags.tld**. Entonces arrancamos del:

Paso 3) – Crear el tag handler de la etiqueta. Seguimos los pasos indicados en el punto 3) de BadTime y basándonos en un ejemplo que el help tiene para la clase GregorianCalendar obtenemos:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ar.com.tag;

import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import java.util.TimeZone;
import java.util.SimpleTimeZone;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Date;
/**
 *
 * @author usuario
 */
public class GregCalTag extends SimpleTagSupport {

    /**
     * Called by the container to invoke this tag.
     * The implementation of this method is provided by the tag library
     * developer, and handles all tag processing, body iteration, etc.
     */
    @Override
    public void doTag() throws JspException {
        JspWriter out = getJspContext().getOut();
        String time = "Current Time for GMT-08:00<br>";
               time+= "(Pacific Standard Time)<br>";
        // get the supported ids for GMT-08:00 (Pacific Standard Time)
        String[] ids = TimeZone.getAvailableIDs(-8 * 60 * 60 * 1000);
        // if no ids were returned, something is wrong. get out.
        if (ids.length == 0)
            System.exit(0);
```

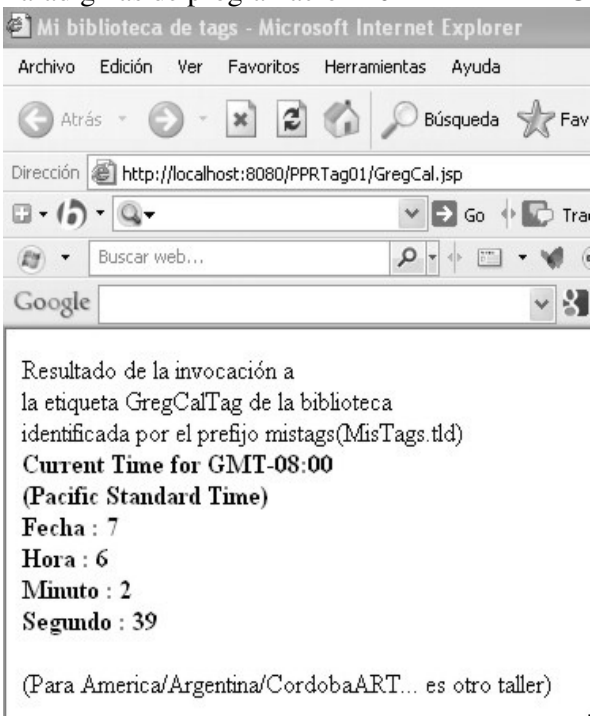
```
        // begin output

        SimpleTimeZone pdt = new SimpleTimeZone(-8 * 60 * 60 * 1000, ids[0]);

        // set up rules for daylight savings time
        pdt.setStartRule(Calendar.APRIL,1,Calendar.SUNDAY, 2*60*60*1000);
        pdt.setEndRule(Calendar.OCTOBER,-1,Calendar.SUNDAY,2*60*60*1000);

        // create a GregorianCalendar with the Pacific Daylight time zone
        // and the current date and time
        Calendar calendar = new GregorianCalendar(pdt);
        Date trialTime = new Date();
        calendar.setTime(trialTime);
        // print out a bunch of interesting things

        time += "Fecha   : " + calendar.get(Calendar.DATE)+"<br>";
        time += "Hora    : " + calendar.get(Calendar.HOUR)+"<br>";
        time += "Minuto  : " + calendar.get(Calendar.MINUTE)+"<br>";
        time += "Segundo : " + calendar.get(Calendar.SECOND)+"<br>";

       try {
            out.println(time);
        } catch (java.io.IOException ex) {
            throw new JspException("Error in GregCalTag tag", ex);}
    }   //  doTag()
}       //  GregCalTag
```

Necesitamos de una página .jsp invocante. Decidimos usar una página nueva, entonces, en el entorno NetBeans:

Paso 4) - Archivo, archivo nuevo, Web, JSP, <Siguiente>
JSP File name: GregCal,
Options: JSP File (Standard syntax) <Terminar>
Vemos que bajo Web Pages aparece GregCal.jsp.
Modificamos la codificación base generada por el IDE, el resultado es:

```
<%--
    Document   : GregCal
    Created on : 07-jul-2009, 0:11:19
    Author     : Tymos
--%>
<%@taglib prefix="mistags" uri="/WEB-INF/MisTags"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Mi biblioteca de tags</title>
    </head>
    <body>
        Resultado de la invocación a <br>
        la etiqueta GregCalTag de la biblioteca<br>
        identificada por el prefijo mistags(MisTags.tld)<br>
        <b><mistags:GregCalTag></mistags:GregCalTag></b><br>
        (Para America/Argentina/Cordoba... es otro taller)<br>
    </body>
</html>
```

Un capture de lo que GregCalTag devuelve



Crear etiquetas con atributos

El par de etiquetas creadas funcionan bien, pero su funcionalidad no requiere de datos de entrada. Ocurre que el grueso de las aplicaciones informáticas procesa datos de entrada proporcionando resultados. Inclusive ya hemos visto un par de Beans que trabajaban de esta manera. El segundo, recibía los límites de un intervalo de números y a partir de las opciones indicadas en casillas de texto (Check Box) retornaba determinados resultados. El beam **SumaBean.java** está en el proyecto PPRBean01.

El concepto de atributos utilizados con tags handler puede resultarle una novedad, pero desde el punto de vista de la codificación, los atributos le resultarán familiares. Ya vimos qué son los javaBeans y cómo implementar los métodos getter y setter en variables. Los tags handler implementan un atributo utilizando un método setter del mismo nombre en su código. Estos métodos setter son los mismos que vimos con los JavaBean.

El elemento <attribute> Los atributos para una etiqueta tienen que ser primero declarados en el archivo TLD. El elemento <attribute> declarado en el archivo TLD es un subelemento de <tag>. Tiene que haber un elemento <attribute> para cada atributo de una etiqueta personalizada.

```
<tag>
      <attribute>
      </attribute>
</tag>
```

El elemento <**attribute**> tiene subelementos adicionales que contienen más información sobre el atributo (sólo **name** es un atributo obligatorio):

```
Elemento    Descripción
<name>      El nombre del atributo.
<required>  Especifica si se necesita el atributo para la etiqueta:
               o  true para hacer que el atributo sea necesario.
               o  false, el atributo es opcional. (valor por defecto)
<rtexprvalue> Especifica si puede ser dinámicamente creado por un
            scriptlet en el tiempo de ejecución:
               o  true
               o  false
<description> Una breve descripción del atributo.
```

Vamos a la tarea.

Ejemplo 1) - Codificar una etiqueta que retorne una línea de texto invertida.

Paso 3) - Crear el tag handler de la etiqueta.

File, New File, Web, Tag Handler,<Next>
1. Class Name: **InvertTag**
2. Location: Source package
3. Package: ar.com.tag
4. Tag Support Class to extend: SimpleTagSupport, **<Next>**

TLD Information
5. Add corresponding information to Tag Library Description: Si, tildar
6. TLD File: <Browse>, seleccionar **MisTags**.
7. Body Content: Empty **<Finish>**
8. Vemos que en el proyecto PPRTag01 aparece, debajo de Source Packages /ar.com.tag/**InvertTag**.java.

Incorporar a la etiqueta **InvertTag** el atributo **textoLeido**. Nos queda:

```
<tag>
  <name>InvertTag</name>
  <tag-class>ar.com.tag.InvertTag</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>textoLeido</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
```

Adecuar la codificación de **InvertTag** suministrada por NetBeans a nuestro propósito. Nos queda:

```
package ar.com.tag;

import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 *
 * @author Tymos
 */
public class InvertTag extends SimpleTagSupport {
    /**
     * Called by the container to invoke this tag.
     * The implementation of this method is provided by the tag library
     * developer, and handles all tag processing, body iteration, etc.
     */
    private String textoLeido = "Texto leido";
    private String textoInver = "";
    private char car;
    @Override
    public void doTag() throws JspException {
        JspWriter out = getJspContext().getOut();
        for(int i = textoLeido.length(); i >= 0;i--){
            car = textoLeido.charAt(i);
            textoInver+=car;
        }
        try {
            // TODO: insert code to write html before writing the body content.
            // e.g.:
```

```
            //
            // out.println("<strong>" + attribute_1 + "</strong>");
            // out.println("    <blockquote>");
            out.println(textoInver);
        } catch (java.io.IOException ex) {
            throw new JspException("Error in InvertTag tag", ex);
        }
    }
}
```

Decidimos gestionar esta aplicación con 2 páginas .jsp, de manera similar a como se hace en el proyecto Yakarta Datetime Taglib Example. Usaremos **input02.jsp** como entrada de datos y **output02.jsp** como invocante de la tag InvertTag y retorno de datos.
Entonces, en el entorno NetBeans:

Paso 4) - Archivo, archivo nuevo, Web, JSP, <Siguiente>
JSP File name: input02,
Options: JSP File (Standard syntax) <Terminar>
Vemos que bajo Web Pages aparece input02.jsp.
Modificamos la codificación base generada por el IDE, el resultado es:

```
<%--
    Document   : input02
    Created on : 09-jul-2009, 8:15:18
    Author     : Tymos
--%>
<%@taglib prefix="mistags" uri="/WEB-INF/MisTags"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
      <h3>Entrada de datos para InvertTag</h3>
      <p>
      <form action="output02.jsp"/>
      Por favor, su identificación <br>
      (Nombre o apodo)<input type="text" name="identific" size="20"><br><br>
      Introduzca un texto corto, tenga la bondad <br>
      El texto es: <input type="text" name="textoLeido" size="20"><br><br>
      Click en <input type="submit" name="submit" value="submit">para el
      resultado<br>que la página <b>output02.jsp</b> nos devuelve.
    </body>
</html>
```

Entrada de datos para InvertTag

Por favor, su identificación
(Nombre o apodo) Puccini

Introduzca un texto corto, tenga la bondad
El texto es: Nessun dorma! Nessun..

Click en [submit] para el resultado
que la página **output02.jsp** nos devuelve.

Paso 4) - Archivo, archivo nuevo, Web, JSP, <Siguiente>

```
      JSP File name: output02,
      Options: JSP File (Standard syntax) <Terminar>
      Vemos que bajo Web Pages aparece output02.jsp.
      Modificamos la codificación base generada por el IDE, el resultado es:
<%--
    Document   : output02
    Created on : 09-jul-2009, 8:42:15
    Author     : Tymos
--%>
<%@ taglib prefix="mistags" uri="/WEB-INF/MisTags"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h3>output02.jsp respondiendo a input02.jsp</h3>
        <% String usuario = request.getParameter("identific");%>
        Estimado Sr(a). <b><%= usuario %></b><br>
        Hemos registrado su siguiente solicitud de servicios:<br><br>
        <% String texto = request.getParameter("textoLeido");%>
        Invertir el texto de: <b><%= texto %></b><br><br>
        y el resultado de la invocación a<br>
        <b>mistags:InvertTag</b> (con atributo texto)<br>
        es el siguiente texto invertido:
        <b><mistags:InvertTag textoLeido = "<%= texto %>">
        </mistags:InvertTag></b><br>
    </body>
</html>      // Aquí hay varias líneas de código que requieren profundización:
```

- **String usuario = request.getParameter("identific");** es una expresión java contenida en una página .jsp, o sea un **scriplet**. Utiliza el método **getParameter** del objeto **request** para asignar el valor del atributo **identific** al objeto usuario.
- **String texto = request.getParameter("textoLeido");** idem anterior
- **mistags:InvertTag textoLeido = "<%= texto %>"** Es una invocación a la etiqueta **InvertTag** de la biblioteca prefijo **mistags** con parámetro **textoLeido**. Este parámetro se informa entre "" y puede asumir un par de formas:
  - "Expresión literal": El valor literal estático de la Expresión literal es el lo que se pasa al tag InvertirTag.
  - **"<%= texto %>"**. Es lo que tenemos en este caso. Entre comillas tenemos una expresión java, un scriplet, cuya evaluación es lo que se pasa al tag.

output02.jsp respondiendo a input02.jsp

Estimado Sr(a). **Puccini**
Hemos registrado su siguiente solicitud de servicios:

Invertir el texto de: **Nessun dorma! Nessun..**

y el resultado de la invocación a
mistags:InvertTag (con atributo texto)
es el siguiente texto invertido: **..nusseN !amrod nusseN**

Ejemplo 2) - Procesar una línea de texto cumpliendo una serie de requerimientos opcionales:

1) - Cuantas veces existe una clave.
2) - En que posiciones del texto se encuentra.
3) - Cuantas palabras tiene la frase.
4) - En que posición se encuentra la más larga.

Comenzamos por el **tag handler**, **FraseTag**. Necesitamos atributos:

```
private String frase, clave, ctasClave, posClave, ctasPal, posLarga;
private String respuesta = "";
```

La primera linea describe los atributos de entrada, el único obligatorio es frase.
La segunda línea contiene la respuesta de la etiqueta a los requerimientos
La forma de definirlo, hemos visto la parte principal

Paso 3) - Crear el tag handler de la etiqueta.

File, New File, Web, Tag Handler,<Next>

1. Class Name: **FraseTag**
2. Location: Source package
3. Package: ar.com.tag
4. Tag Support Class to extend: SimpleTagSupport, **<Next>**

TLD Information

5. Add corresponding information to Tag Library Description: Si, tildar
6. TLD File: <Browse>, seleccionar **MisTags**.
7. Body Content: tagdependent

Atributes. En la parte inferior de la interface **TLD Information** aparece un área para definir los atributos. Ejemplificaremos el atributo **frase**: **<new>**

Al <Aceptar> el atributo se incorpora en:

- la forma **Atributes**.
- en la descripción de la etiqueta **FraseTag**.
- En la codificación de la tag handler.

Repetimos esta acción tantas veces como atributos tenga la etiqueta.

Incorporado el último < **Finish>**

8. Vemos que en el proyecto PPRTag01 aparece, debajo de Source Packages /ar.com.tag/FraseTag.java con todos los atributos y doTag().
9. Vemos que en MisTags.tld se ha incorporado la descripción de la etiqueta FraseTag con todos sus atributos:

```
<tag>
  <name>FraseTag</name>
  <tag-class>ar.com.tag.FraseTag</tag-class>
  <body-content>tagdependent</body-content>
  <attribute>
    <name>frase</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
```

```
                <name>clave</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
              </attribute>
              <attribute>
                <name>ctasClave</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
              </attribute>
              <attribute>
                <name>posClave</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
              </attribute>
              <attribute>
                <name>ctasPal</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
              </attribute>
              <attribute>
                <name>posLarga</name>
                <required>false</required>
                <rtexprvalue>true</rtexprvalue>
              </attribute>
            </tag>
```

Dejaremos la codificación definitiva del método doTag() para mas adelante. Vamos ahora con las JSP.

Definimos la primera, que llamamos **input03** como habitualmente:

Paso 4) - Archivo, archivo nuevo, Web, JSP, <Siguiente>
JSP File name: **input03**,
Options: JSP File (Standard syntax) <Terminar>
Vemos que bajo Web Pages aparece input03.jsp.
Modificamos la codificación base generada por el IDE, el resultado es:

```
<%--
    Document   : input03
    Created on : 11-jul-2009, 20:53:01
    Author     : Tymos
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Trabajando con una frase</title>
    </head>
    <body bgcolor="#FFFFFF">
    <h3>Brindando diversos servicios opcionales sobre una frase</h3>
    <br>
    <form action="output03.jsp">
     La frase a  trabajar:<input type="text" name="frase" size="40"><br>
     La clave a buscar es: <input type="text" name="clave" size="5"><br><br>

     Por favor, indíquenos en que servicios está interesado<br>

    <input type="checkbox" name="ctasClave" value="X">
     Cuantas veces ocurre la clave informada<br>
    <input type="checkbox"name="posClave"  value="X">
     En que posiciones la clave ocurre <br>
```

```
    <input type="checkbox" name="ctasPal" value= "X">
     Cuantas palabras contiene la frase<br>
    <input type="checkbox" name="posLarga" value="X">
     En que posición se encuentra la palabra mas larga<br>
    Click en <input type="submit" name="submit" value="submit">para el
resultado<br>
    que la página <b>output03.jsp</b>, invocante de<br>
    la tag handler <b>FraseTag</b> nos devuelve.
    </form>
    </body>
</html>
```

Si ejecutamos input03.jsp en nuestro navegador predeterminado aparece:

Brindando diversos servicios opcionales sobre una frase

La frase a trabajar: En un lejano pais moraba un tiránico rey.

La clave a buscar es: pais

Por favor, indíquenos en que servicios está interesado

- [x] Cuantas veces ocurre la clave informada
- [ ] En que posiciones la clave ocurre
- [x] Cuantas palabras contiene la frase
- [x] En que posición se encuentra la palabra mas larga

Click en [submit] para el resultado
que la página **output03.jsp**, invocante de
la tag handler **FraseTag** nos devuelve.

Paso 4 bis) - Archivo, archivo nuevo, Web, JSP, <Siguiente>
JSP File name: **output03**,
Options: JSP File (Standard syntax) <Terminar>
Vemos que bajo Web Pages aparece **output03.jsp**.
Modificamos la codificación base generada por el IDE, el resultado es:

```
<%--
    Document   : output03
    Created on : 11-jul-2009, 21:37:15
    Author     : Tymos
--%>
<%@ taglib prefix="mistags" uri="/WEB-INF/MisTags"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Trabajando Frases</title>
    </head>
    <body>
        <h3>output03.jsp respondiendo a input03.jsp</h3>
        Estimado Sr(a). Usuario. Hemos recibido el siguiente conjunto<br>
        <% String fraseLeid  = request.getParameter("frase");%>
```

```
        <% String claveLeid  = request.getParameter("clave");%>
        <% String ctasClaves = request.getParameter("ctasClave");%>
        <% String posClaves  = request.getParameter("posClave");%>
        <% String ctasPals   = request.getParameter("ctasPal");%>
        <% String posLargas  = request.getParameter("posLarga");%>

        Frase a tratar:<b><%= fraseLeid%></b>, clave <b><%= claveLeid% ></b>
        <br><br>
        Sobre esta frase se requieren los siguientes servicios<br>
        Cuantas veces ocurre la clave
        <b><%= claveLeid%></b>:<b><%= ctasClaves%></b><br>
        En que posiciones ocurre la clave: <b><%= posClaves%></b><br>
        Cuantas palabras tiene la frase <b><%= ctasPals%></b><br>
        En que posicion está la mas larga: <b><%= posLargas%></b><br><br>
        El resultado de los servicios requeridos es:<br>

        <b><mistags:FraseTag frase = "<%= fraseLeid %>"
                             clave = "<%= claveLeid %>"
                             ctasClave = "<%= ctasClaves %>"
                             posClave = "<%= posClaves %>"
                             ctasPal  = "<%= ctasPals %>"
                             posLarga = "<%= posLargas %>">
        </mistags:FraseTag></b><br>
     </body>
</html>
```

FraseTag.java tiene la siguiente codificación:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ar.com.tag;

import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 *
 * @author Tymos
 */
public class FraseTag extends SimpleTagSupport {
    /**
     * Called by the container to invoke this tag.
     * The implementation of this method is provided by the tag library
     * developer, and handles all tag processing, body iteration, etc.
     */

    private String frase, clave, ctasClave, posClave, ctasPal, posLarga;
    private String respuesta = "";

    //  Un método seter por cada atributo
    public void setFrase(String frase){this.frase = frase;}
    public void setClave(String clave){this.clave = clave;}
    public void setCtasClave(String ctasClave){this.ctasClave= ctasClave;}
    public void setPosClave(String posClave){this.posClave = posClave;}
    public void setCtasPal(String ctasPal){this.ctasPal = ctasPal;}
    public void setPosLarga(String posLarga){this.posLarga = posLarga;}

    @Override
    public void doTag() throws JspException {
        JspWriter out = getJspContext().getOut();
        //  Instanciamos objeto hileras
```

```
        Hileras hilera = new Hileras(frase);

        //  Cuantas veces ocurre una clave
        if(clave!=null &&ctasClave!=null)
            respuesta+=hilera.cuantasVeces(clave);

         //   En que posiciones ocurre la clave
        if(clave!=null &&posClave!=null)
            respuesta+=hilera.dondeOcurre(clave);

         //  Cuantas palabras tiene la frase
        if(ctasPal!=null)
            respuesta+=hilera.cuantasPalabras();

         //  Cual es la palabra mas larga
        if(posLarga!=null)
            respuesta+=hilera.palabraMasLarga();
        try {
            out.println(respuesta);
        } catch (java.io.IOException ex) {
            throw new JspException("Error in FraseTag tag", ex);
        }
    }   //  doTag()
}   //  FraseTag
```

El método **doTag** de **FraseTag** utiliza objetos Hileras, su clase:

```
package ar.com.tag;
public class Hileras{
      private char carAct, carAnt; private String hilera;
      private int contPal;

      public Hileras(String hilera){                  //  Constructor,
        this.hilera = hilera+" ";contPal = 0;
      }

    public String cuantasVeces(String clave){
        String auxRet = "La clave "+clave+" ocurre ";
        int fromIndex, posic, cuantas=0;
        boolean hayMas = true;
        for(fromIndex = 0; hayMas;){
            if((posic = hilera.indexOf(clave, fromIndex)) >= 0){// encontramos
               fromIndex = posic+1; cuantas++;
            } else hayMas = false;
        }
        auxRet+=cuantas + " veces<br>";
            return auxRet;
    }   //  cuantasVeces

    public String dondeOcurre(String clave){
        String auxRet = "La clave "+clave+" ocurre en posiciones: ";
        int fromIndex, posic=0;
        boolean hayMas = true;
        for(fromIndex = 0; hayMas;){
            if((posic = hilera.indexOf(clave, fromIndex)) >= 0){// encontramos
               auxRet+=(fromIndex+posic)+", ";
               fromIndex = posic+1;
            }
            else hayMas = false;
        }
            auxRet+="<br>";
            return auxRet;
    }   //  dondeOcurre
```

```
    public String cuantasPalabras(){
        carAnt = hilera.charAt(0);
        for(int ind = 1;ind < hilera.length();ind++){
            carAct = hilera.charAt(ind);
            if(finPal(carAnt, carAct))contPal++;
            carAnt = carAct;
        }
        return "La frase contiene " + contPal +" palabras<br>";
      }

    public boolean finPal(char ant, char act){       // Es fin palabra?
        boolean fPal = false;
        Caracter antCar = new Caracter(ant);
        Caracter actCar = new Caracter(act);
        if  (antCar.pertPal() && !actCar.pertPal())fPal = true;
            return fPal;
      }

    public String palabraMasLarga(){
        return "Palabra mas larga, a cargo del alumno, está bien?<br>";
    }
}  // public class Hileras
```

La respuesta de FraseTag al requerimiento de servicios de **Input03.jsp** es:

output03.jsp respondiendo a input03.jsp

Estimado Sr(a). Usuario. Hemos recibido el siguiente conjunto
Frase a tratar:**En un lejano pais moraba un tiranico rey**, clave **pais**

Sobre esta frase se requieren los siguientes servicios
Cuantas veces ocurre la clave **pais**:X
En que posiciones ocurre la clave: **null**
Cuantas palabras tiene la frase X
En que posicion está la mas larga: X

El resultado de los servicios requeridos es:
La clave pais ocurre 1 veces
La frase contiene 8 palabras
Palabra mas larga, a cargo del alumno, está bien?

Seguir la pista a usuarios

Muchos sitios Web con éxito dependen de un enfoque personalizado para atraer y retener visitantes. Para crear un servicio personalizado, el sitio tiene que ser capaz de recordar al usuario en múltiples visitas.
Tomemos como ejemplo a Amazon, que es una gran tienda online visitada cada día por millones de clientes que vuelven. Para aumentar sus ventas, Amazon necesita destacar productos que puedan suscitar el interés del cliente. Por lo tanto, como la mayoría de las tiendas Web, Amazon recordará si el cliente ha visitado antes el sitio, qué cosas le interesan y qué productos han comprado en el pasado.

Los sitios tienen que recordar también a los usuarios durante las visitas. Por ejemplo, cuando un cliente compra productos el sitio le permite adquirir múltiples productos y le permite pagarlos todos juntos al mismo tiempo. Analizaremos los detalles de cómo se sigue la pista del usuario en este tipo de aplicación,
donde se sigue al usuario en una visita.

Nos concentraremos en la utilización de JSP para seguir la pista de los usuarios.

¿Qué es una sesión?

Para construir aplicaciones Web eficaces, es necesario asociar lógicamente una serie de peticiones diferentes de un cliente en particular. En sitios Web de compras, cada compra será puesta en su carrito, para que al finalizar pueda pagarlo todo junto. El carrito recuerda todos los artículos que ha decidido comprar, incluso si ha navegado por cientos de páginas y ha comprado decenas de cosas. Esto es posible porque estas aplicaciones Web son capaces de seguir la pista de las sesiones. Esto se denomina seguimiento de una sesión.

Se puede seguir una sesión identificando al cliente de forma única en cada petición hecha al servidor.

El protocolo HTTP

HTTP, el protocolo estándar que se utiliza en toda la red (WWW) para acceder a archivos no tiene estado por diseño. Los protocolos de Internet pueden ser divididos en dos tipos: con estado y sin estado. Un protocolo como el protocolo de transferencia de ficheros (FTP), que se utiliza para transferir archivos, recuerda a un cliente desde su entrada en el sistema hasta su salida. Es decir, recuerda el estado de la conexión con el cliente. Por lo tanto, puede llevar a cabo varias operaciones en una sola sección.

HTTP, por el contrario, es un protocolo sin estado. Es inherente al protocolo HTTP que no hay manera de asociar una petición con otra; pueden entrar en momentos diferentes y en puertos diferentes. Por lo tanto, no existe una asociación de peticiones por parte del servidor. Si cada vez que añadiera un nuevo artículo a su carrito olvidara todos los artículos anteriores y no recordara cual era su carrito, ¡sería muy difícil construir una tienda online con éxito!

¿Cómo funciona el protocolo HTTP?

Veamos la secuencia de acontecimientos que ocurre cuando pide una página Web, digamos por ejemplo, http://www2.lavoz.com.ar

- El navegador abre una conexión HTTP con el servidor Web en lavoz.
- Envía una petición HTTP para una página Web.
- El servidor Web responde con la página pedida o un mensaje de error.
- Se cierra la conexión entre el servidor Web y el navegador.

La próxima vez que envíe una petición para otra página Web al mismo servidor Web se abre una nueva conexión; el servidor Web no será capaz de decir si la petición es de un nuevo usuario o el mismo.

De vuelta a las sesiones

Una sesión puede ser vista como un objeto asociado a un solo cliente, que ayuda a generar y administrar datos específicos del usuario en una aplicación Web. Dentro de nuestra aplicación, la sesión se trata como un objeto único asociado con un conjunto de datos relacionados. Una sesión puede ser creada y destruida cómo y cuándo se necesite. Además, una sesión puede tomarse un descanso y expirar. Los datos como la información sobre un usuario habitual pueden asociarse con una sesión. Por ejemplo, muchos sitios Web necesitan que el usuario se registre en el sistema para acceder a sus servicios y que salga del sistema cuando termine la sesión. Así, una vez que entra en el sistema, se crea una sesión que se llenará con información sobre el usuario. Esta información puede ser leída y escrita varias veces, según lo requiera la aplicación. Al salir del sistema, los datos asociados con la sesión pueden ser almacenados en el servidor o destruidos.

Los datos almacenados en el servidor podrían ser información sobre pedidos o preferencias anteriores, de forma que la siguiente vez que entre en el sitio será como si nunca lo hubiera dejado.
El seguimiento de sesiones es importante en las aplicaciones Web de comercio electrónico actuales. Como hemos visto antes, un carrito de compra tiene que

reflejar de forma precisa las elecciones hechas por el cliente en cada paso del proceso de compra.

Seguimiento de sesiones

Se han desarrollado tres mecanismos principales para permitir el seguimiento del cliente:
o Cookies
O Reescritura URL
O Campos de formulario ocultos

Cookies

Las cookies es el mecanismo de seguimiento de sesiones más utilizado. Una cookie es un archivo de texto almacenado en la máquina del cliente con la ayuda de un navegador Web como Netscape Navigator o Microsoft Internet Explorer, etc.

- Las cookies almacenan información utilizando los pares nombre/valor
- la devuelven al servidor que la creó en posteriores peticiones.
- El servidor crea una cookie, la llena de información relevante
- y la envía al navegador del cliente.
- El cliente almacena esta cookie en el disco duro
- El cliente envía la cookie de nuevo al servidor en posteriores peticiones.
- Todas las cookies tiene una fecha de caducidad establecida,
- esta fecha puede ser de un futuro lejano.

Sin embargo, la utilización de cookies presenta inconvenientes. Por convención, una cookie no debería tener un tamaño mayor de 4KB y ningún dominio debería tener más de 20 cookies. Además, la información confidencial como los detalles de una tarjeta de crédito no debería almacenarse en una cookie. Ha habido mucha polémica en lo relativo a preocupaciones de privacidad creadas por el uso de cookies. Por estas razones, algunos usuarios inhabilitan las cookies en sus navegadores Web, evitando que las cookies se utilicen como un mecanismo de seguimiento de sesiones. Una aplicación que dependa de cookies para seguir sesiones tendría, en este caso, problemas.

Aunque las JSP tienen una interfaz de alto nivel y fácil de utilizar para las cookies, existen todavía algunos detalles relativamente tediosos que necesitan ser tratados:

- La extracción de la cookie que almacena el identificador de la sesión de las otras cookies.
- El establecimiento de una fecha de caducidad adecuada para la cookie.
- La asociación de la información que está en el servidor con el identificador de la sesión (puede haber demasiada información para poder almacenarla realmente en la cookie, además de que datos confidenciales como los números de una tarjeta de crédito no deberían ir nunca en cookies).

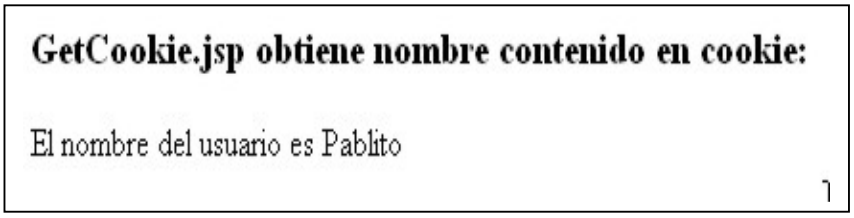
Vamos a un ejemplo simple, usando dos páginas .jsp

- **AddCookie.jsp** escribe una cookie en el navegador del cliente
- **GetCookie.jsp** obtiene las cookies procedentes del cliente cuando devuelve el valor de la cookie que fue establecida porAddCookie.jsp.

```
<'--AddCookie.jsp-->
<html>
   <head>
      <title>Esta página almacena una cookie en su navegador</title>
   </head>
   <body>
      <h3>AddCookie.jsp instala cookie en su navegador!</h3>
      <% // Un pequeño scriplet…
         Cookie myCookie = new Cookie("usuario", "Pablito");
         myCookie.setMaxAge(24*60*60);
         response.addCookie(myCookie);
      %>
```

```
      <a href="GetCookie.jsp">Read Cookie</a>
   </body>
</html>
```

```
<!--GetCookie.jsp-->
<html>
   <head>
      <title>Esta página obtiene la cookie almacenada en su navegador</title>
   </head>
   <body>
      <h3></h3>
      <% // Otro pequeño scriplet
         Cookie[] cookieList = request.getCookies();
         for(int i = 0; i < cookieList.length;i++){
            Cookie myCookie = cookieList[i];
            if (myCookie.getName().equals("usuario"))
               out.println("El nombre del usuario es " + myCookie.getValue());
         }
      %>
   </body>
/html>
```



Reescritura URL

La reescritura URL se utiliza para añadir datos que identifiquen la sesión al final de cada URL. El servidor asocia el identificador con los datos sobre la sesión que ha almacenado, de forma que el ID pueda ser utilizado en cada petición posterior para asociar la petición con un cliente en particular.
Esta técnica apenas se usa porque existen métodos más sencillos y menos indiscretos de seguir sesiones.

Campos de formulario ocultos

Los formularios HTML pueden tener una entrada con el siguiente aspecto:

```
<input type="hidden" name="session" value="12345">
```

El hecho de declarar un tipo como "hidden" puede ocultar un campo de formulario. No obstante, al enviar el formulario, el nombre y el valor especificados se incluyen en los datos GET o POST. Esto se puede utilizar para almacenar información sobre la sesión. Por ejemplo, cuando un usuario compra el producto, los identificadores de los artículos que el usuario ha añadido a su carrito de la compra podrían haber sido añadidos como campos de formulario ocultos.
Esta técnica es admitida por todos los navegadores principales y no necesita ninguna forma de autorización del cliente; sin embargo, su utilidad es limitada, porque sólo funciona para una secuencia de formularios generados dinámicamente y la gente puede ver la fuente HTML para ver los datos almacenados.

Se han desarrollado muchas estrategias como las que hemos descrito previamente para seguir las sesiones de los usuarios. Pero la mayoría de ellas resultan difíciles o problemáticas para ser directamente aplicadas por el programador.

Utilización de sesiones en páginas JavaServer

Las JSP proporcionan un método sencillo para controlar sesiones, utilizando el objeto implícito llamado session. Este objeto utiliza internamente uno de los métodos descritos antes para seguir sesiones, pero los detalles no se exponen al desarrollador. Por ejemplo, Tomcat utiliza cookies para identificar el cliente cuando puede, si no recurre a utilizar reescritura URL.

Analicemos ahora cómo utilizar el objeto session para seguir sesiones, como se define en la especificación JSP. Es bastante sencillo de utilizar; los pasos básicos para utilizar el objeto son:

- Declarar que la página participa en una sesión.
- Leer o escribir al objeto session.
- Terminar la sesión bien finalizándola o bien no haciendo nada por lo que finalmente caducará por sí misma.

Las propiedades de una JSP se declaran utilizando la directriz **<%@page%>**. La directriz que se muestra a continuación indica que:

- el lenguaje de scripting está basado enJava,
- que las clases declaradas en el paquete java.util están disponibles, de forma directa, para el código scripting,
- que la página participa en sesiones:

%@page language="java " import=java.utíl.*" session="true "%

Si el valor de session es "true " , una variable implícita de lenguaje de script (como el objeto request) del tipo javax.servlet.http.HttpSession se encuentra disponible para su uso en la página, con el nombre session. Representará la sesión existente, o si no hay ninguna sesión existente, se creará una nueva sesión.
Si el valor de session es "false", entonces la página no puede participar en sesiones. Si trata de escribir un código con inicio de sesión en dicha página, obtendrá un error de compilación.
Por defecto, el valor es "true", por lo que si no declara explícitamente el valor como false, se asume entonces que la página está interesada en seguir sesiones.
Debería observar que existe una sesión para cada aplicación Web en su servidor.

La aplicación está contenida en una carpeta e incluye sub carpetas. Esto significa que aunque las JSP de una carpeta puedan compartir datos de la sesión entre
ellas, no pueden compartirlos con JSP que se encuentren en otra carpeta al lado.

El objeto Session en detalle

Como hemos visto, el objeto session está disponible para páginas que tengan el atributo session establecido como "true". Veamos ahora qué podemos hacer con el objeto session.

- Almacenamiento de datos en Session.
- Comprobación de una nueva seseión.
- Obtención del identificador de la sesión.
- Eliminación de la sesión.
- Tiempo de espera superado de la sesión.

Almacenamiento de datos en Session
Esto se hace llamando a su atributo setAttribute():

```
Integer iFirstVal = new Integer(l);
session.setAttribute("firstVal".iFirstVal);
```

El objeto iFirstVal (Integer) es almacenado en la sesión con el nombre "firstVal".
(Cada objeto almacenado se asocia con un único nombre, que lo identifica).

Esto es similar a la forma en la que especificamos un atributo id único en la acción <jsp: useBean> como vimos.
Una vez almacenado, un objeto unido a la sesión está disponible para cualquier otra JSP que esté en la misma sesión y contexto, llamando al método getAttribute() del objeto sesión y especificando el mismo nombre que se utilizó cuando se lo almacenó.

```
Integer iPrevVal = (Integer) session.getAttribute("firstVal");
```

Al obtener valores de la sesión, tenemos que asignar nuestro objeto de nuevo a su clase particular. Esto es debido a que el tipo devuelto por getAttribute() se declara como Object.

Si desea obtener todos los nombres de todos los objetos de la sesión use elmétodo getAttributeNames(), que devuelve una Enumeration de los atributos.
Finalmente, el método removeAttribute() elimina un objeto nombrado de la sesión.

Comprobación de una nueva sesión
Podemos comprobar si la sesión es nueva haciéndole al objeto session la pregunta:

```
if (session.isNew(){
      //    Tratamiento de nueva sesion
}
```

La sesión se considera nueva hasta que un cliente se une a la sesión. En otras palabras, la información relacionada con la sesión que se proporciona al cliente tiene que ser devuelta al servidor en una petición posterior, completando así el circuito y diciendo que la sesión ha sido establecida con éxito.
Veamos un ejemplo:

- Un cliente envía una petición HTTP a un servidor Web, → aplicación.
- El servidor genera un **ID único** para identificar al cliente
- **ID único** en una cookie + respuesta → cliente.
- Cliente recibe la respuesta y almacena el ID.
- Cliente envía la segunda petición + cookie → servidor.
- Servidor asume que el cliente está participando en la sesión.
- Sesión continúa siendo nueva si el cliente fracasa en cerrar el circuito.

Obtención del identificador de la sesión
Cada sesión es definida por un único ID de sesión, que se utiliza para seguir peticiones múltiples del mismo cliente al servidor y asociar el cliente con sus datos de sesión.
Podemos obtener el identificador de la sesión, que es una cadena única asignada a una sesión específica, utilizando el método getId():

```
      String id – session.getId();
```

Eliminación de la sesión
Necesitamos ser capaces de cerrar la sesión cuando ya no la necesitemos y esto se lleva a cabo utilizando el método **session.invalidate().**

Tiempo de espera superado de la sesión
Desgraciadamente, el protocolo HTTP no nos dice si el cliente está activo todavía o no y si no nos deshacemos periódicamente de sesiones antiguas acabaríamos por quedarnos sin espacio de almacenamiento en nuestro servidor. Como HTTP no tiene estado, el único mecanismo que podemos utilizar para determinar cuando un usuario está inactivo es utilizando un tiempo de espera y para asegurarnos de que las sesiones se eliminan de forma periódica, los servidores nos permiten definir tiempos de espera para las sesiones.

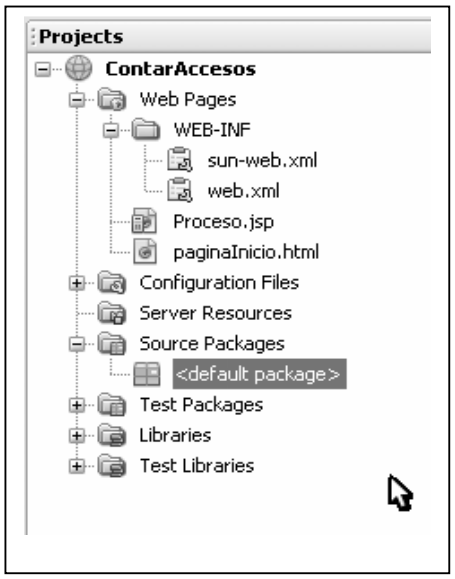
Los servidores como Tomcat definen por defecto el tiempo de espera para una sesión, que se puede obtener mediante el método **getMaxInactiveInterval()** del

objeto session. El desarrollador puede cambiar también este tiempo de espera utilizando el método **setMaxInactiveInterval ();** el tiempo de espera utilizado por estos métodos se define en segundos. Si se establece el tiempo de espera para una sesión en -1, la sesión no caduca nunca. Cuando utilice su aplicación Web en Tomcat, puede definir el tiempo de espera por defecto de la sesión utilizando la etiqueta <session-timeout> en el archivo Web.xml que especifica la información de configuración para la aplicación. Por ejemplo, el código que aparece a continuación especifica que las sesiones de esta aplicación no tendrán tiempo de espera por defecto; para utilizar esto debería guardarlo en el Web.xml en la carpeta WEB-INF dentro de la carpeta de su aplicación:

```
<webapp>
      <session-config>
            <session-timeout>-1</session-timeout>
      </session-config>
</webapp
```

Cómo utilizar el objeto session

Veamos el proyecto **ContarAccesos**, un ejemplo que demuestra la utilización del objeto session en una JSP. Esta página seguirá la pista del número de veces que un usuario accede a la página, almacenando el contador de acceso en la sesión. Cada vez que un usuario acceda a la página, el contador de acceso se incrementará y se almacenará de nuevo en la sesión. Además, mostraremos como el ID de la sesión.



PaginaInicio.jsp

Contando accesos mediante cookies

Nombre cliente: Frodo

Ejecutar => submit

Proceso.jsp

Proceso.jsp respondiendo a paginaInicio.html

Estimado Sr cliente: **Frodo**

UD. a visitado la pagina **2** veces

--

PaginaInicio.html

```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
   <head>
        <title>Contador de accesos usando cookies</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
      <form action="Proceso.jsp">
        <table border="0">
          <thead>
```

```
            <tr>
                <th>Contando accesos</th>
                <th>mediante cookies</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Nombre cliente:</td>
                <td><input type="text" name="cliente" value="" size="20"></td>
            </tr>
            <tr>
                <td>Ejecutar =></td>
                <td><input type="submit" value="submit" name="submit" ></td>
            </tr>
        </tbody>
      </table>
    </form>
  </body>
</html>

<%--
     Document   : Proceso.jsp
    Created on : 04/08/2009, 14:03:41
    Author     : MaqTymos
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Proceso.jsp</title>
    </head>
    <body>
        <h3>Proceso.jsp respondiendo a paginaInicio.html</h3>
        <% String cliente  = request.getParameter("cliente");
            String sCuenta  = null;
            Cookie[] cookies = request.getCookies();
            // Obtener el valor actual de la cookie de nombre "usuario" buscando
            // entre las cookies recibidas. Puede haber varias ...
            if (cookies != null){ // si hay cookies...
                for (int i = 0; i < cookies.length; i++){
                    // Buscar la cookie "usuario"
                    if (cookies[i].getName().equals("usuario")){
                        // y obtener el valor asociado
                        sCuenta = cookies[i].getValue();
                        break;
                    }
                }   //  for
            }   //  if (cookies != null)

            // Incrementar el contador para esta página. El valor es
            // guardado en la cookie de nombre "usuario".
            // Después, asegurarse de enviársela al cliente con la
            // respuesta (response).
            Integer objCuenta = null; // contador
            if (sCuenta == null) // si no se encontró "usuario"
                objCuenta = new Integer(1);
            else                 // se encontró, sCuenta tiene valor != null
                objCuenta = new Integer(Integer.parseInt(sCuenta)+1);
            // Crear una nueva cookie con la cuenta actualizada
            Cookie c = new Cookie("usuario", objCuenta.toString());
            // Añadir la cookie a las cabeceras de la respuesta HTTP
```

```
            response.addCookie(c);
      %>
      <%    // Responder al cliente %>
           Estimado Sr cliente: <b> <%= cliente %> </b> <br>
           UD. a visitado la pagina <b> <%= sCuenta %> </b> veces <br>
      <%    // Cerrar el flujo
              out.close();
      %>
    </body>
</html>
```

Bueno, hemos logrado seguir una sesión y contar las veces que un cliente accesa el sitio. Lo que no está demasiado bien es el como: Hemos incluido un robusto código script dentro de Proceso.jsp y ya habiamos dicho que esto no debe hacerse, por el tema de la reusabilidad del soft. Veamos de trasladar el grueso de esta codificación a un Java Bean o a una Custom Tag. (Tema pendiente)

Ya vimos la utilización de la acción <jsp:useBean> para concretar los componentes JavaBeans de una JSP, pero establecimos siempre el valor del atributo scope como "page". De hecho, un objeto declarado en una JSP utilizando la acción <jsp:useBean> puede ser declarado en distintos ámbitos:
El scope define la disponibilidad de un objeto:

- ámbito **page**: Disponible solo para la pɹgina que manipulamos.
- ámbito **request**: Disponible para la pɹgina que manejamos y para cualquier página a la que le pase control.
- ámbito **session**: Disponible para cualquier JSP dentro de la misma sesión.
- ámbito **application**: Disponible para cualquier componente de la misma aplicación Web.

No puede crear objetos con ámbito session en una JSP que no participe en las sesiones. Otra advertencia: todos los objetos asociados con la sesión se liberan una vez que se destruye la sesión y si se intenta utilizar uno de ellos se emitirɹ una excepción. El tema es bastante vasto, trataremos de ejemplificarlo con casos concretos.

API Java Servlet

El API Java Servlet tiene mucho en común con la tecnología JSP. Gran parte de sus prestaciones se derivan de los servlets Java y, de hecho, antes de ejecutar una página JSP, se compila en un servlet. El proceso es el siguiente:

- El programador diseña una página JSP utilizando texto de plantillas HTML, etiquetas JSP y JavaBeans. Seguidamente, implementa un contenedor Web (un servidor Web compatible con Java).
- Cuando recibe la primera solicitud para un determinada página JSP, la página se transforma en la clase de Java correspondiente a las instrucciones definidas en el código JSP. El contenedor se encarga de crear este objeto que, realmente, es un servlet.
- Las solicitudes clientes se pasan a este objeto servlet y las respuestas del mismo al cliente, por medio del servidor.
- Si la página JSP se modifica posteriormente, el servidor recibirá el cambio y convertirá la nueva versión en una nueva clase Java.

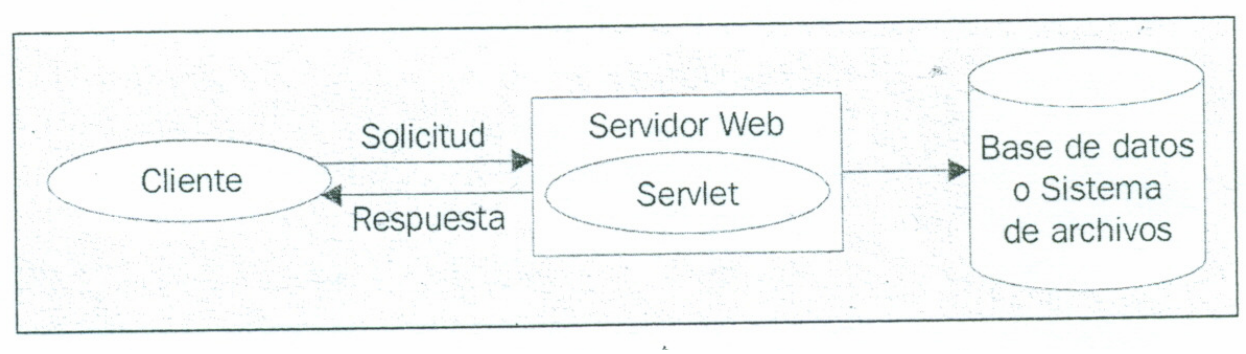
Por esta razón, el uso de servlets es fundamental en JSP. Veremos los servlets desde el punto de vista de un programador de páginas JSP. Nos centraremos principal
mente en los siguientes aspectos:

- Arquitectura de un servlet y su funcionamiento.
- Proceso que se sigue al ejecutar una página JSP.
- Aplicaciones Web y el archivo web.xml que se utiliza para configurarlas.
- Conceptos avanzados: seguimiento de sesiones, el contexto del servlet y cómo reenviar e incluir solicitudes.

Tecnología Java Servlet

Un servlet es un componente o programa que genera contenido Web dinámico. Los servlets se definen por medio del API Java Servlet y se gestionan por medio de un servidor o contenedor Web como Tomcat.
El protocolo HTTP sobre el que se construye la Web, funciona por medio de un mecanismo de solicitudes y respuestas en el que un servidor recibe una solicitud, la procesa y devuelve la respuesta correspondiente. El API Java servlet modela este proceso y lo orienta a objetos para que el código que se programa pueda procesar solicitudes del proveedor y responderlas de forma automática. Por ejemplo, un servlet puede utilizar los datos de un formulario HTML de introducción de pedidos para actualizar la base de datos de pedidos de una empresa.

Como hemos mencionado, los servlets se ejecutan dentro de un servidor compatible con Java (un contenedor Web) como puede ser Tomcat, como se aprecia en la imagen:



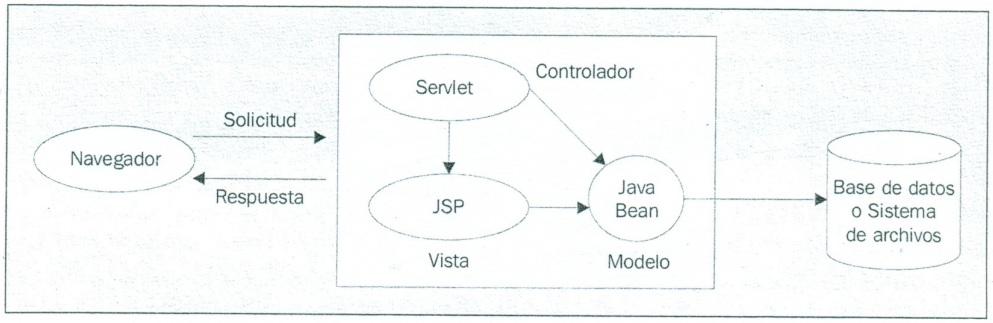
El contenedor Web carga, ejecuta y administra el servlet, siguiendo el proceso que se describe a continuación:

- El cliente envía una solicitud de página al contenedor.
- En caso de que el servlet no se encuentre cargado, el contenedor lo carga. Una vez cargado tras la primera solicitud, permanece en este estado hasta que el contenedor decide descargado (habitualmente cuando se cierra el propio contenedor).
- El contenedor envía la información de la solicitud al servlet, creando un nuevo hilo para cada solicitud a ejecutar.
- El servlet procesa la solicitud, construye una respuesta y la transmite al contenedor.
- El contenedor envía la respuesta de nuevo hasta al cliente.

El contenedor es uno de los elementos más importantes de todo el proceso, ya que se encarga de cargar y de inicializar el servlet. Puede procesar varias instancias de un servlet, determina el servlet que debe recibir una determinada solicitud, comprueba que se devuelve la respuesta al cliente y, por último, elimina el servlet una vez terminada su vida operativa.

Función de un servlet en una aplicación Web

Las distintas partes de una aplicación Web se pueden separar por medio de las acciones <jsp: forward> y <jsp: include> y su relación con la arquitectura "Model 2" o "Modelo Vista Controlador". Una de las funciones de un servlet es la de controlador dentro de una aplicación Web, ya que disponen de todas las prestaciones de Java, no como JSP. Un servlet controlador actúa en función de la solicitud del cliente y coordina o delega en otros servlets, objetos compartidos, archivos de recursos, bases de datos y cualquier otro tipo de dispositivos disponibles.
Veamos un ejemplo típico de arquitectura Model 2:



El proceso se divide en componentes de información (modelo) y presentación (vista), dirigidos por el controlador. El papel del servlet controlador consiste en procesar las solicitudes, crear componentes de información como javaBeans u otros objetos utilizados por el componente presentación (JSP). Habitualmente, el controlador también determina el componente de presentación al que se debe reenviar la solicitud.
El modelo MVC permite estructurar la aplicación Web por lo que resulta más sencillo desarrollarla y extenderla. En este caso, la aplicación se divide en tres partes: **modelo**, **vista y controlador**.

- **Modelo**. Contiene la parte central de la funcionalidad de la aplicación. El modelo representa el estado de la aplicación, sin prestar atención a la vista o al controlador. Los JavaBeans pueden desempeñar este papel ya que se pueden diseñar para procesar la mayor parte de la lógica de negocio de una aplicación. Pueden interactuar con una base de datos o con un sistema de archivos, por lo que su misión consiste en mantener los datos de la aplicación.

- **Vista**. Ofrece la presentación del modelo, en otras palabras, decide la forma en que se presentan los datos al usuario. La vista puede acceder a los datos del modelo pero no puede modificarlos. Además, no se ocupa en absoluto del controlador. Se puede notificar a la vista el momento en el que se producen cambios en el modelo (datos). Un programador de interfaces Web no tiene porque saber lo que sucede en la base de datos o en el componente de lógica de negocio. Deben tener amplios conocimientos de HTML pero no de Java o de otros lenguajes de programación. Una página JSP puede desempeñar este papel ya que su diseño le permite utilizar la cantidad minina de un código que no sea HTML. **Veremos esto muy en detalle un poco mas adelante, cuando tratemos JSTL y EL.**

- **Controlador**. El controlador reacciona a las entradas introducidas por el usuario. Crea y proporciona entradas para el modelo. Un servlet puede contener código Java y HTML, por lo que puede adoptar solicitudes HTTP, tomar decisiones sobre la creación de los correspondientes JavaBeans y notificar a la vista los cambios efectuados en el modelo.

Los casos en que resulta más indicado utilizar esta arquitectura son:

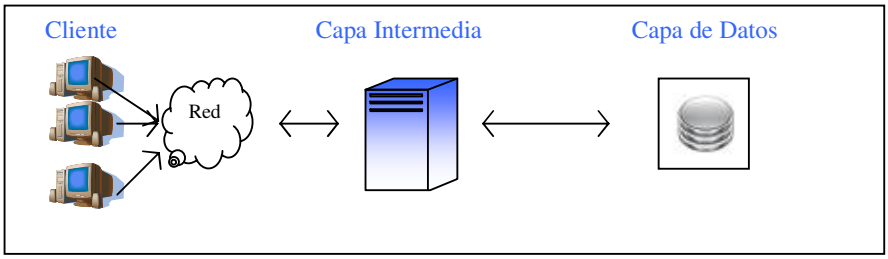
- La página Web se basa en información proporcionada por el usuario durante el tiempo de ejecución. Por ejemplo, páginas de resultados en motores de búsqueda o aplicaciones que procesan pedidos de sitios de comercio electrónico.
- Los datos se actualizan con frecuencia en la página Web. Por ejemplo, páginas con pronósticos meteorológicos o con titulares de noticias.
- La página Web debe utilizar información de bases de datos corporativas o comerciales, como puede ser el caso de una página de una tienda online que muestre los precios actuales y los productos en stock.

El tema del modelo MVC es de gran importancia en la correcta distribucion de incumbencias, responsabilidades en una aplicación Web, por ello incluimos aquí un aporte de un docente de la catedra, Ing. Martin Polliotto.

ARQUITECTURA WEB. Patrón de Diseño Modelo-Vista-Controlador.

ARQUITECTURA DE TRES CAPAS

Una aplicación Web es un programa informático que puede dar servicio *simultáneamente* a múltiples usuarios que lo ejecutan a través de Internet. Este tipo de aplicaciones se basa en lo que se conoce como una arquitectura de tres capas, donde diferentes actores y elementos implicados se encuentran distribuidos en tres bloques o capas.



Capa Cliente

Se trata de la capa con la que interactúa el usuario de la aplicación, normalmente a través de un navegador Web. Básicamente en esta capa se realizan dos funciones principales:

- Se encarga de capturar los datos de usuario con lo que interactúa la capa intermedia y enviárselo a ésta.
- Presentar al usuario los resultados generados por la aplicación.

Capa Intermedia

En una arquitectura de tres capas la capa intermedia esta constituida por la aplicación en sí. Esta se encuentra instalada en una máquina independiente, conocida como servidor, a la que acceden los clientes a través de la red utilizando el protocolo HTTP.

La aplicación de la capa intermedia es ejecutada por un motor de aplicación especial capaz de permitir que una instancia de ella pueda dar servicio a múltiples clientes. Además de este motor, los servidores necesitan otro software conocido servidor Web, que sirva de interfaz entre a aplicación y el cliente, realizando el diálogo http con éste.

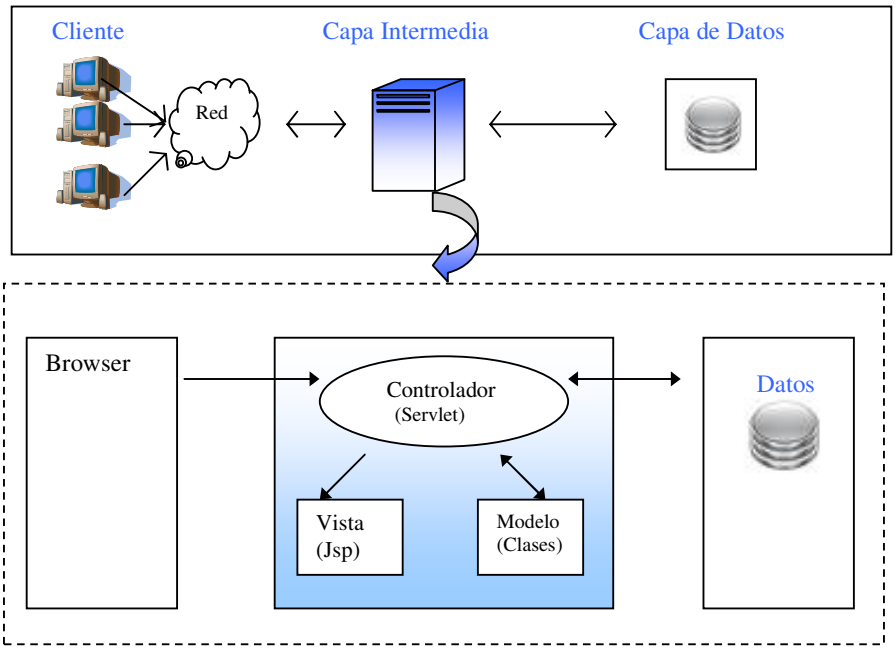
De manera resumida podemos mencionar las tres funciones principales de esta capa:

- Recoger los datos enviados desde la capa cliente.
- Procesar la información y, en general, implementar la lógica de aplicación, incluyendo el acceso a datos.
- Generar las respuestas para el cliente.

Capa de datos

La capa de datos tiene como misión el almacenamiento permanente de la información manejada por la aplicación y la gestión de la seguridad de los mismos. Esta tarea es soportada, generalmente, por las bases de datos relacionales (Oracle, MySQL, SQL Server, etc.).

Nuestro desafío como desarrolladores consiste en implementar exitosamente la capa intermedia de la aplicación. Para ello necesitamos definir un modelo formado por una serie de bloques o componentes, de modo que cada uno pueda *desarrollarse de manera independiente con responsabilidades claramente definidas.* En este contexto es donde juega un papel fundamental el patrón Modelo-Vista-Controlador (MVC).



El controlador

Se puede decir que el **controlador** es el cerebro de la aplicación. **Todas las peticiones a la capa intermedia que se realicen desde el cliente son dirigidas al controlador**, cuya misión es determinar las acciones a realizar por cada una de estas peticiones e invocar al resto de los componentes de la aplicación (sean jsp o clases del modelo) para que realicen las acciones requeridas en cada caso, encargándose también de la coordinación de todo el proceso.

Por ejemplo, en el caso de que una petición requiera enviar como respuesta al cliente determinada información existente en una base de datos (o en una simple colección, como lo ejercicios que propone la materia), el controlador solicitará los datos necesario al modelo y, una vez recibidos, se los proporcionará a la vista.

En una aplicación JEE, el controlador es implementado a través de un servlet central, que dependiendo de la cantidad de peticiones que debe gestionar, puede apoyarse en otros servlets auxiliares para procesar cada petición.

La vista

La vista será la encargada de generar las respuestas (generalmente HTML o XML) que deben ser enviadas al cliente. Cuando esta respuesta tiene que incluir datos proporcionados por el controlador, el código HTML de la página no será fijo sino que deberá ser generado de forma dinámica, por lo que su implementación correrá a cargo de una página JSP.

El modelo

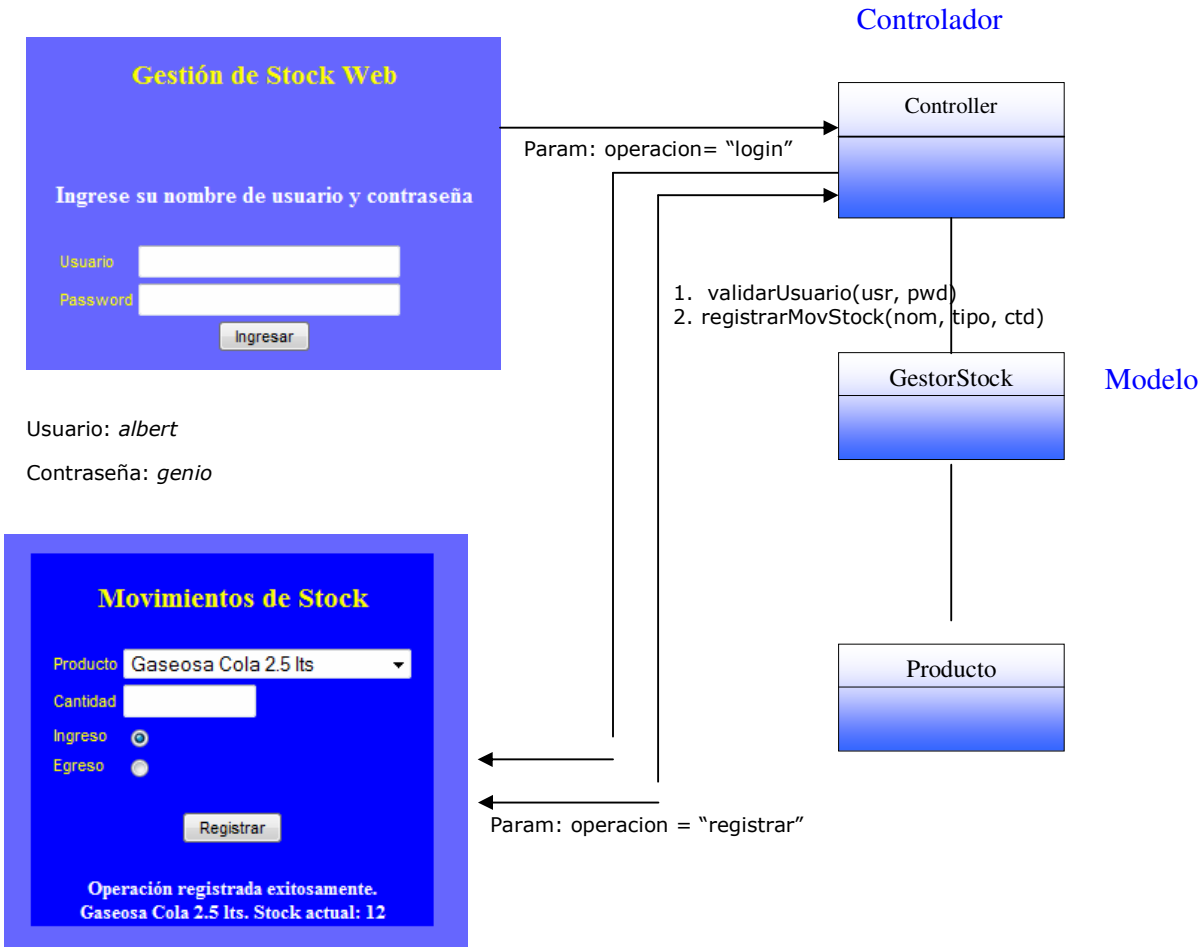
En la arquitectura MVC la lógica de negocio de la aplicación, incluyendo el acceso a datos y su manipulación, esta encapsulada dentro del modelo. En una aplicación JEE el modelo puede ser implementado mediante clases estándar Java o mediante Enterprise JavaBeans.

APLICACIÓN DE MVC. GestionStockWeb.

A continuación se ejemplifica el uso del modelo MVC en un caso práctico de estudio. Este propone una aplicación Web donde se registran los movimientos de stock de determinados

productos de un negocio. Para poder registrar ingresos o egresos de los artículos, el usuario deberá previamente identificarse mediante un nombre y contraseña específicos.
Al momento de registrar un egreso de productos, la aplicación deberá controlar que el stock actual sea suficiente para la cantidad solicitada. En caso de serlo, además deberá informar si el stock resultante se equipara con el nivel mínimo de seguridad previsto para el artículo. Si no existe disponibilidad o la cantidad ingresada no es correcta, deberá notificarse al usuario.

El esquema de resolución es el siguiente:



Usuario: *albert*

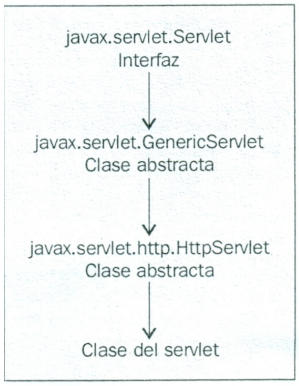
Contraseña: *genio*

Fin Aporte Polliotto

Arquitectura de un servlet

A continuación diseñaremos un servlet. Todos los servlets utilizan interfaces y clases definidas en los paquetes javax.servlet y javax.servlet.http. El primer paquete contiene definiciones compatibles con servlets genéricos (el diseño original de los servlet permitía utilizarlos entre distintos protocolos, no solamente con HTTP). El segundo se ocupa de los servlets diseñados específicamente
para HTTP.

Todos los servlets deben implementar la interfaz javax.servlet.Servlet, que declara métodos para administrar el servlet y comunicación con el cliente. El programador del servlet debe cerciorarse de que se implementan dichos métodos a la hora de diseñar un nuevo servlet. Afortunadamente, disponemos de la clase javax.servlet. hhtp.HttpServlet que permite implementar la interfaz mencionada anteriormente. Por esta razón, la mayoría de las clases de servlet que diseñaremos para aplicaciones Web extienden la clase HttpServlet. En el siguiente diagrama se puede comprobar la herencia de una típica clase de servidor:



Cuando el servlet acepta una llamada de un cliente, recibe dos objetos de las clases:

- javax.servlet.ServletRequest: comunicación cliente ➔ servidor, (request) que implementa javax.servlet.http.HttpServletRequest
- javax.servlet.ServletResponse: comunicación servlet ➔ cliente, (response) que implementa javax.servlet.http.HttpServletResponse.

Procesamiento de solicitudes

Como hemos visto anteriormente, un servlet acepta una solicitud de un cliente y la procesa para crear una respuesta, que devuelve de nuevo al cliente. La interfaz básica Servlet define un método service() para procesar las solicitudes del cliente y que se invoca para cada una de las solicitudes que el contenedor pasa al servlet.
La solicitud se guarda en el objeto HttpServletRequest y la respuesta en el objeto
HttpServletResponse. La solicitud se analiza por medio de los métodos que proporciona la clase HttpServletRequest y la respuesta se escribe en el objeto HttpServletResponse que el contenedor reenvía al cliente.

HttpServlet es una clase abstracta que implementa la interfaz Servlet. También añade otros métodos, lo que significa que apenas tendremos que proporcionar un método service() propio. Por el contrario, los métodos que ofrecemos a continuación se invocan de forma automática por el método service() prediseñado, en función del tipo de solicitud:

- doGet(). Se invoca para procesar solicitudes HTTP GET.
- doPost(). Se invoca para procesar solicitudes HTTP POST.

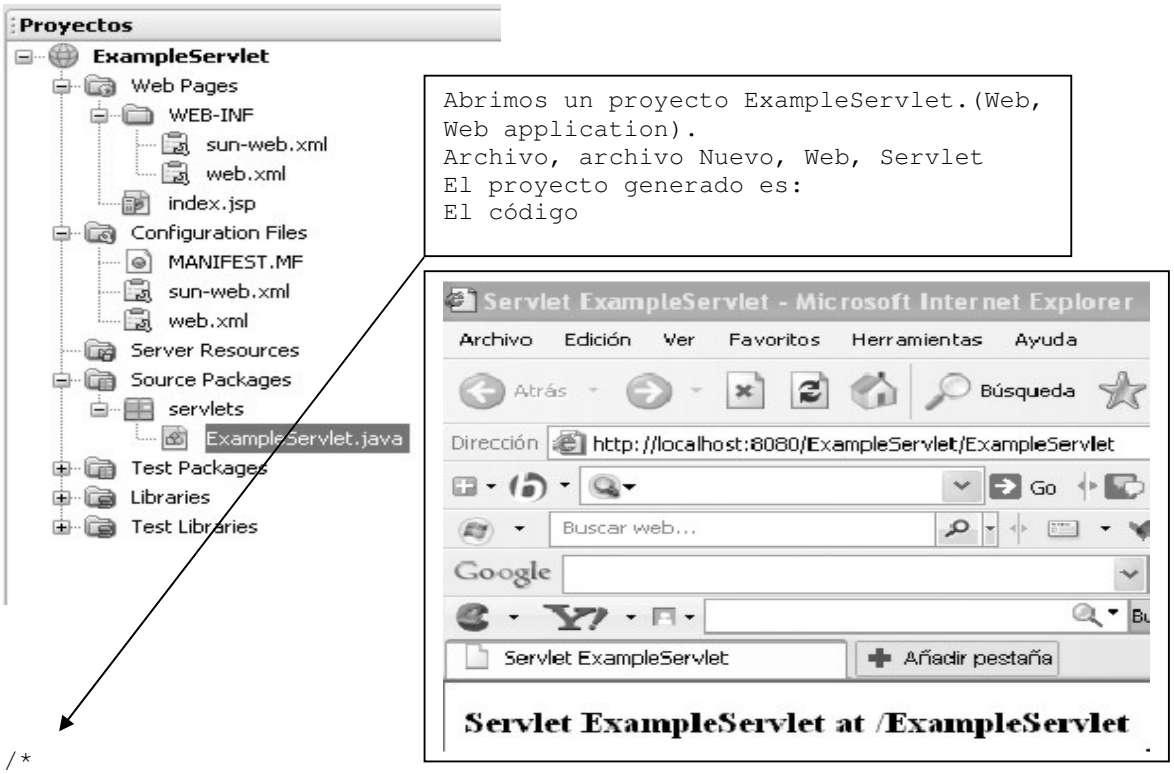
Habitualmente, al diseñar un servlet basado en HTTP, el programador debe preocuparse exclusivamente de los métodos doGet() y doPost() . Cuando se realiza una solicitud de una página representada por el servlet, el contenedor la reenvía al servlet invocando el método service() de la clase base javax.servlet.Servlet. La implementación de HttpServlet, en función del tipo de solicitud, invoca bien el método doGet() o bien doPost() , por lo que el servlet debe anular uno de los dos métodos.
Sin embargo, existen cinco métodos similares que se utilizan con menor frecuencia:

- doHead(). Se invoca para procesar solicitudes HTTP HEAD. En HttpServlet, este método especializado ejecuta el método doGet(), pero solamente devuelve al cliente los encabezados generados por el método doGet().
- doOptions(). Procesa solicitudes HTTP OPTIONS. Este método determina de forma automática los métodos HTTP directamente compatibles con el servlet y devuelve dicha información al cliente.
- doTrace(). Método que se invoca para procesar solicitudes HTTP TRACE. Genera una respuesta con un mensaje que contiene todos los encabezados enviados en la solicitud TRACE.
- doPut(). Se invoca para solicitudes HTTP PUT.
- doDelete(). Se invoca para solicitudes HTTP DELETE.

HttpServlet dispone de algunos métodos de interés:

- `init() y destroy()`, que nos permiten administrar recursos asignados al servlet durante toda su vida operativa.
- `getServletInfo()`, método que utiliza el servlet para conseguir información sobre si mismo.

Empezaremos con un ejemplo sencillo (simplemente el que nos genera nuestro IDE) y utilizaremos algunas de las clases que hemos mencionado anteriormente. ExampleServlet extiende la clase HttpServlet que, a su vez, implementa la interfaz Servlet. Devuelve una sencilla página HTML. Utilizaremos este ejemplo como referencia cuando analicemos el ciclo vital del servlet.

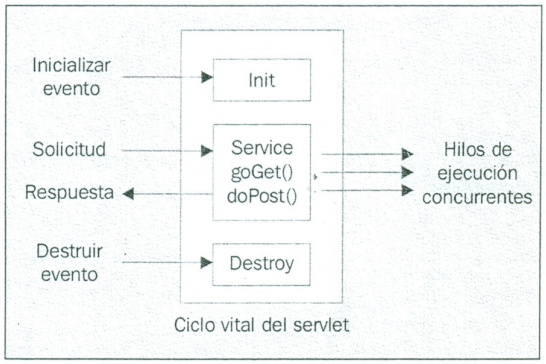



```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package servlets;

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

/**
 *
 * @author Tymos
 */
public class ExampleServlet extends HttpServlet {
    /**
     * Processes requests for both HTTP <code>GET</code> and code> POST </code>
     * methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
```

```
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                 throws ServletException, IOException{
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            // TODO output your page here
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet ExampleServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h3>Servlet ExampleServlet at " +
            request.getContextPath () + "</h3>");
            out.println("</body>");
            out.println("</html>");
        } finally {
            out.close();
        }
    }

    // <editor-fold default state="collapsed" desc="HttpServlet methods. Click
    on the + sign on the left to edit the code.">
    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {

            processRequest(request, response);

    }

    /**
     * Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
@Override
    protected void doPost(HttpServletRequest request,
                  HttpServletResponse response)
                  throws ServletException, IOException {

        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>

}
```

Ciclo vital del servlet

El contenedor se encarga de este proceso, crea los objetos servlet correspondientes así como los métodos necesarios. En la siguiente imagen se puede apreciar el ciclo vital del servlet en términos generales:



Carga, creación de instancias e inicialización

Un servidor carga y crea instancias de un servlet de forma dinámica cuando se solicitan sus servicios por primera vez. También se puede configurar el servidor Web para cargar y crear instancias de servlets específicos cuando se inicialice el servidor, en especial en aquellos casos en los que el proceso de inicialización implica operaciones de larga duración, como abrir una conexión a una base de datos.
Puede resultar muy útil para mejorar el tiempo de respuesta a la primera solicitud. El método init() del servlet es el encargado de la inicializacin y se invoca para cada una de las instancias de servlet, antes de procesar cualquier solicitud. Este método se hereda de HtppServlet, por lo que solamente se puede anular en caso de que se deba ejecutar una determinada función al inicio. Por ejemplo, se puede utilizar el método init() para cargar datos predeterminados o crear conexiones a bases de datos.

Procesamiento de solicitudes

Una vez inicializado adecuadamente el servlet, el contenedor lo puede utilizar para procesar solicitudes.
En cada una, se pasa un objeto ServletRequest a los métodos doGet() o doPost() del servlet, que representa la solicitud y un objeto ServletResponse que se puede utilizar para crear la respuesta que recibirá el clienre.
En el caso de solicitudes HTTP, el contenedor ofrece los objetos request y response como implementación de las interfaces HttpServletRequest y HttpServletResponse. Cuando llega una solicitud al contenedor, busca una instancia del servlet correspondiente. Si no lo encuentra, carga uno e invoca doGet() o doPost(), en función de cómo se enviara la solicitud. Esto significa que en cada solicitud, existe un hilo distinto que ejecuta el mismo método doGet() o doPost() . En este caso, la seguridad de los hilos es fundamental, como veremos más adelante.

El objeto Request

La interfaz HttpServletRequest permite al servlet acceder a información sobre los parámetros de la solicitud pasada por el cliente. También le proporciona acceso al objeto de flujo de entrada, ServletImputStream, a través del cual puede leer datos de los clientes que utilizan protocolos de la aplicación como el método HTTPPOST.
Ya mencionamos el objeto HttpServletRequest. El objeto JSP request es una instancia de esta clase.
El objeto request contiene toda la información de la solicitud cliente. En el protocolo HTTP, esta información se puede transmitir del cliente al servidor por medio de una cadena de consulta, en encabezados HTTP y en el cuerpo del mensaje de la solicitud.

La interfaz HttpServletRequest es una extensión de ServletRequest y contiene los métodos
necesarios para acceder a información de encabezados específicos de HTTP, como pueden ser las cookies encontradas en la solicitud. También nos permite acceder a los parámetros enviados por el cliente junto a la solicitud.
Para ello, disponemos de los siguientes métodos:

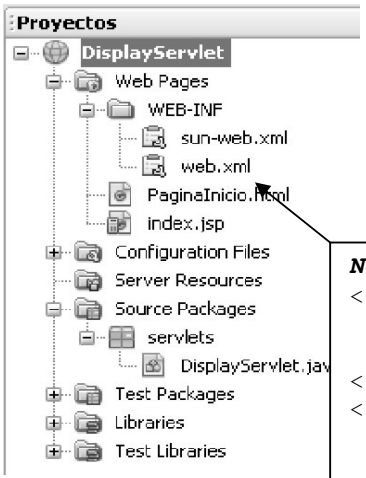
- **getParameter()** devuelve el valor de un determinado parámetro de la solicitud,
  proporcionando el nombre del parámetro. Si el parámetro dispone de más de un valor, puede utilizar el método **getParameterValues()**, que devuelve todos los valores.
- Para solicitudes HTTP GET, puede utilizar el método **getQueryString()**, que devuelve una cadena de datos del cliente. En este caso, tendré que escribir personalmente el código que permite extraer esta información.
- En métodos HTTP POST, puede utilizar tanto **getReader()** como **getInputStream()**, en
  función de si espera texto o datos binarios. El método getReader() devuelve un objeto BufferedReader y getInputStream() un objeto ServletInputStream.

El objeto Response

La interfaz HttpServletResponse define métodos servlet para contestar al cliente. La función de un servlet consiste en procesar la solicitud y generar la respuesta adecuada. El objeto response contiene toda la información que el servidor debe devolver al cliente. En el protocolo HTTP, esta información se transmite del servidor al cliente en los encabezados HTTP y en el cuerpo de mensaje de la solicitud.
El objeto **HttpServletResponse** ofrece dos formas para devolver datos al usuario:

- getWriter(), devuelve un objeto Writer (texto)
- getOutputStream(),devuelve un objeto ServletOutputStream (datos binarios)
- métodos adicionales
  - sendRedirect().Redirige al cliente a una dirección URL distinta, que debe ser absoluta.
  - sendError() . Envía un mensaje de error

DisplayServlet



A continuación codificamos DisplayServlet, usando algunas de los métodos de los objetos request y response que acabamos de ver.
Usaremos PaginaInicio.html (del proyecto ContarAccesos),
Como entrada de datos y aprovechado su distribución tabular le incorporamos nuevos componentes. Esta página invocara a DisplayServlet quien elabora y responde.

Nota: *Al incorporar el servlet aparecen, en* ***web.xml***

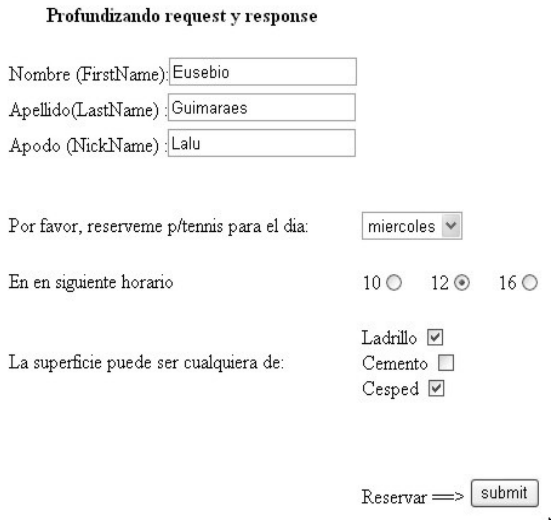
```
<servlet>
  <servlet-name>DisplayServlet</servlet-name>
  <servlet-class>servlets.DisplayServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>DisplayServlet</servlet-name>
  <url-pattern>/DisplayServlet</url-pattern>
</servlet-mapping>
```

PaginaInicio.html

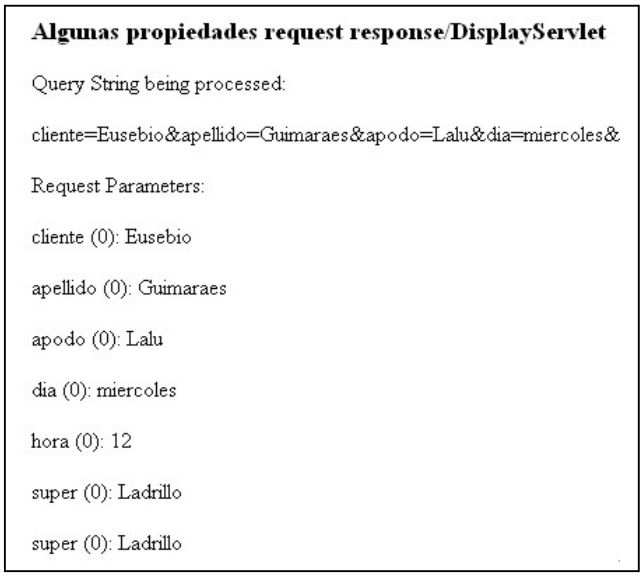
```
<!--
To change this template, choose Tools | Templates
and open the template in the editor.
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
   <head>
        <title>Probando opciones</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
      <form action="DisplayServlet">
        <table border="0">
          <thead>
              <tr>
                  <th>Profundizando request y response</th>
                  <th></th>
              </tr>
          </thead>
          <tbody>
              <tr>
                  <td><br>
                      Nombre  (FirstName):<input type="text" name="cliente"
                                                value="" size="20"></td>
                  <td></td>
              </tr>
               <tr>
                  <td>Apellido(LastName) :<input type="text" name="apellido"
                                                value="" size="20"></td>
                  <td></td>
              </tr>
               <tr>
                  <td>Apodo   (NickName) :<input type="text" name="apodo"
                                               value="" size="20"></td>
                  <td></td>
              </tr>
              <tr>
                  <td><br><br>Por favor, reserveme p/tennis para el dia:</td>
                  <td><br><br>
                        <select name="dia">
                            <option>lunes <option>miercoles <option>jueves
                        </select>
                  </td>
              </tr>
              <tr>
                  <td><br>En en siguiente horario</td>
                  <td><br>
                        10<input type="radio" name="hora" value="10" checked>
                           
                        12<input type="radio" name="hora" value="12">
                           
                        16<input type="radio" name="hora" value="16">
                  </td>
              </tr>
              <tr>
                  <td><br>La superficie puede ser cualquiera de:</td>
                  <td><br>Ladrillo
                        <input type="checkbox" name="super" value="Ladrillo">
                        <br>Cemento
                        <input type="checkbox"  name="super" value="Cemento">
                        <br>Cesped
                        <input type="checkbox"  name="super" value="Cesped">
                  </td>
              </tr>
               <tr>
                  <td>
                  </td>
                  <td><br><br><br>Reservar ==> <input type="submit"
                                                value="submit"

                                                                name="submit">
                  </td>
              </tr>
             </tbody>
        </table>
      </form>
   </body>
</html>
```

El documento generado:



Responde DisplayServlet (capture parcial)



El código de **DisplayServlet**

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package servlets;

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

/**
 *
 * @author usuario
 */
public class DisplayServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                  throws ServletException, IOException{
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {// TODO output your page here
            out.println("html");
            out.println("head");
            out.println("<title>Algunas propiedades request response</title>");
```

```
            out.println("</head>");
            out.println("<body>");
            out.println("<h3>Algunas propiedades request response"
                               + request.getContextPath () + "</h3>");
            out.println("Query String being processed:<p>");
            out.println(request.getQueryString());
            out.println("<p>");
            out.println("Request Parameters:<p>");
            Enumeration enumParam = request.getParameterNames();
            while(enumParam.hasMoreElements()){
                String paramName = (String) enumParam.nextElement();
                String paramValues[] = request.getParameterValues(paramName);
                if (paramValues != null){
                    String auxTxt ="";
                    for (int i = 0; i < paramValues.length; i++){
                        auxTxt += paramName;
                        auxTxt += " (";
                        auxTxt += i + "): ";
                        auxTxt += paramValues[i] + "<p>";
                        out.println(auxTxt);
                    }   //  for
                }   //  if
            }   //  while
            out.println("</body>");
            out.println("</html>");
        } finally {out.close();}
    } // protected void processRequest

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods.
      Click on the + sign on the left to edit the code.">
    /**
     * Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
    throws ServletException, IOException {processRequest(request, response);}


 /**
     * Returns a short description of the servlet.
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>
```

```
}   //  public class DisplayServlet extends HttpServlet {
```

Descargas

Un servlet se descarga de la memoria cuando se cierra el contenedor. El contenedor invoca el método **destroy()** del servlet, método heredado de la clase GenericServlet y que solamente es necesario para anularla en el servlet en caso de que haya que ejecutar determinadas acciones al cerrar el contenedor. También es extensible al hecho de cerrar cualquier archivo o conexiones a bases de datos que se encuentren abiertas.

¿Qué sucede en la página JSP?

Como mencionamos anteriormente, tanto la tecnología JSP como la de servlets ofrecen ventajas e inconvenientes. Los servlets facilitan a los programadores la tarea de diseñar código para generar contenidos HTML dinámicos basados en lógica de negocio. No obstante, el código se llena en exceso de HTML y resulta complicado integrar el trabajo de diseñar interfaces Web con el de programar Java. El enfoque JSP se ha diseñado para que el diseño de código HTML sea más sencillo e independiente del código Java que contenga lógica de negocio. De hecho, JSP atraviesa una fase de traducción al principio de su ciclo vital (la primera vez que lo solicita un usuario) para convertirse en servlet, lo mismo que una crisálida se transforma en mariposa(¿??)…

Colaboración entre servlets

De la misma forma que es posible que varias páginas JSP colaboren en una única solicitud del usuario, con un servlet se pueden realizar las mismas operaciones, como veremos más adelante.

Seguimiento de sesiones

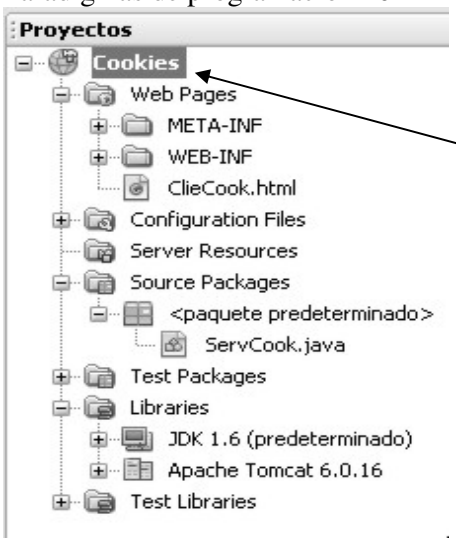
El protocolo HTTP es, por su diseño, un protocolo sin estado. Para diseñar aplicaciones Web eficaces, es necesario asociar de forma lógica series de diferentes solicitudes entre sí. Con el paso del tiempo, muchas de las estrategias de seguimiento de sesión han evolucionado, pero resulta demasiado complicado para el programador utilizarlas de forma directa. Por esta razón, el API Java Servlet ofrece una sencilla interfaz, javax.servlet.http. HttpSession, que permite a los contenedores servlet realizar el seguimiento de una sesión de usuario sin necesidad de que el programador se implique en el proceso.
Esta interfaz se ha analizado cuando vimos JSP; el objeto session de JSP permite implementarla.
El seguimiento de sesión permite a los servlets conservar en el tiempo la información de estado sobre una serie de solicitudes efectuadas por el mismo usuario. Para poder utilizar este mecanismo es necesario:

- Obtener el objeto HttpSession de HttpServletRequest.
- Almacenar y recuperar datos del objeto session.
- Cuando los datos de la sesión pierdan su utilidad, destruir el objeto session.

El método getSession() del objeto HttpServletRequest devuelve la sesión actual asociada a la solicitud; en caso de que no exista sesión actual, se crea una. Al almacenar los datos en la sesión, se crean pares nombre-valor.
La interfaz HttpSession dispone de métodos que nos permiten almacenar, recuperar y eliminar atributos:

- setAttribute()
- getAttribute()
- getAttribueNames()
- removeAttribute()

El método invalidate() sirve para invalidar la sesión, lo que implica la destrucción de todos los objetos de la misma.



Ya vimos **Seguimiento de sesiones** cuando vimos **Seguir la pista a los usuarios**, qué es una sesión? Pg 73. En esa ocasión lo resolvimos codificando un scriplet en la página JSP. Ahora codificaremos un servlet **Cookies** que utilice el seguimiento de sesiones para saber cuántas veces ha accedido a la misma un determinado usuario, y diferenciaremos la sesión según la identificación del cliente/usuario.

ClieCook.html

```
<!--
    Document   : ClieCook
    Created on : 26-oct-2008, 16:52:31
    Author     : Tymos
-->
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Probando Cookies ...</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h3>Contador de accesos usando cookies</h3>
    <form action="ServCook">
      El cliente:<br><input type="text" name="cliente" size="60">
      <br><br>Solicita acceso al servlet ServCook
      <br><br>
      <input type="submit" value="Acceder ServCook">
    </form>
  </body>
</html>
```

Esto se ve:

Contador de accesos usando cookies

El cliente:

Bernardo

Solicita acceso al servlet ServCook

Acceder ServCook

Pruebe de, sin salir de la forma, cambiar el nombre del cliente. Si el nombre aparece en esta sesión por primera vez, verá que ServCook le informa de su primera visita. Si, en cambio, es una reiteración, verá que ServCook, usando el array de cookies recuerda cuantas visitas anteriores UD ha realizado y lo contabiliza correctamente.

ServCook.java responde:

Hola, Sr(a) Bernardo
Has visitado esta página 1 vez.
Veamos... En que pudiera serle útil?

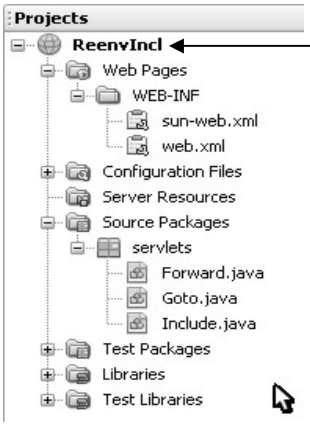
La codificación del servlet:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
// Author Tymoschuk, Jorge
public class ServCook extends HttpServlet{
    // Daremos un poco de variedad a los mensajes con que el servlet responde
    String[] msgs = {"Siempre es un placer recibirle",
                     "Par Ud. tengo todo ... todo el tiempo",
                     "Pero hoy mi agenda está completa, completa",
                     "Veré que dice mi secretaria, mis compromisos",
                     "Solo dispongo de contados segundos ...",
                     "Ud me sorprende. Pareciera creer soy igual ...",
                     "Veamos... En que pudiera serle útil?"};
    String mensa; int indice;

  protected void doGet(HttpServletRequest request,
           HttpServletResponse response)throws ServletException, IOException{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Obtener el valor actual de la cookie "contador.cook" buscando
    // entre las cookies recibidas.
    String sCuenta  = null;
    String cliente  = null;
        cliente     = request.getParameter("cliente");
    Cookie[] cookies = request.getCookies();
    if (cookies != null){ // si hay cookies...
        for (int i = 0; i < cookies.length; i++){
            // Buscar la cookie con el nombre del cliente
            if (cookies[i].getName().equals(cliente)){
                // y obtener el valor asociado
                sCuenta = cookies[i].getValue();
                break;
            }
        }   //  for
    }   //  if (cookies != null)

    // Incrementar el contador para esta página. El valor es
    // guardado en la cookie con el nombre del cliente.
    // Después, asegurarse de enviársela al cliente con la
    // respuesta (response).
    Integer objCuenta = null; // contador
    if (sCuenta == null) // si no se encontró nombre del cliente
        objCuenta = new Integer(1);
    else                 // se encontró el cliente en cuestion
        objCuenta = new Integer(Integer.parseInt(sCuenta)+1);
    // Crear una nueva cookie con la cuenta actualizada
    Cookie c = new Cookie(cliente, objCuenta.toString());
    // Añadir la cookie a las cabeceras de la respuesta HTTP
    response.addCookie(c);

    // Responder al cliente
    out.println("<html>");
    out.println("Hola, Sr(a) "+cliente);
    out.println("<br>");
    out.println("Has visitado esta página " + objCuenta.toString() +
                ((objCuenta.intValue() == 1) ? " vez." : " veces."));
    out.println("<br>");
    indice = (int)(7*Math.random());
```

```
    mensa = msgs[indice];
    out.println(mensa);
    out.println("</html>");

    // Cerrar el flujo
    out.close();
  }

  // Devuelve una descripción breve.
  public String getServletInfo(){return "Servlet CuentaCk";}
}
```

Cómo reenviar e incluir solicitudes

Ya hemos visto como diseñar una aplicación Web para que reenvíe solicitudes a otros recursos para su posterior procesamiento o incluir el resultado de un servlet o JSP dentro de otro. En JSP, esta operación se consigue por medio de las acciones <jsp:forward> y <jsp:include>; un servlet también puede reenviar o incluir otro recurso con ayuda de la interfaz javax.servlet.RequestDispatcher.



En el proyecto **ReenvIncl** (Reenviar/incluir) crearemos tres servlets:

- **Include**: Incluimos el Servet Goto
- **Forward**: Re direccionamos al servlet Goto
- **Goto**: este servlet es utilizado por los dos anteriores. Su lógica le permite detectar cual es la solicitut origen y proceder en consecuencia.

Nota: Al crear los servlets dentro del entorno NetBeans el descriptor de despliegue web.xml es actualizado automáticamente.

Forward.java

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package servlets;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.RequestDispatcher;
/**
 *
 * @author Tymos
 */
public class Forward extends HttpServlet{
    /**
     * processes requests for both HTTP <code>GET</code> and
     * <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     * @throws ServletException if a servlet-specific error occurs
     * @throws IOException if an I/O error occurs
     */
    String forwardingAddress = "Goto";
    protected void processRequest(HttpServletRequest request,
```

```
                                HttpServletResponse response)
                                throws ServletException, IOException{
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            // TODO output your page here
            out.println("<h3>Servlet Forward, redireccionando a Goto</h3>");
            //    Linea anterior no sale, investigar
            request.setAttribute("option", "forward");
            RequestDispatcher dispatcher =
            request.getRequestDispatcher(forwardingAddress);
            dispatcher.forward(request, response);
            out.println("<h3>Si todo anda bien, esta linea no debe salir</h3>");
            //    Linea anterior no sale, correcto
        } finally {out.close();}
    }
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                    throws ServletException, IOException {
        processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }
}     //    public class Forward
```

Include.java `// Mostramos solo lo diferente`

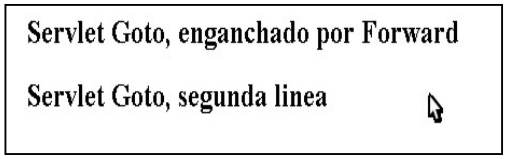
```
public class Include extends HttpServlet {

    String forwardingAddress = "Goto";
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            // TODO output your page here
            request.setAttribute("option", "include");
            RequestDispatcher dispatcher =
                    request.getRequestDispatcher(forwardingAddress);
            out.println("<h3>Servlet Include, por incluir servlet Goto</h3>");
            dispatcher.include(request,response);
            out.println("<h3>Servlet Include, de vuelta ... (Ok?)</h3>");
        } finally {out.close();}
    }
}
```

Goto.java `//    Mostramos solo lo diferente`

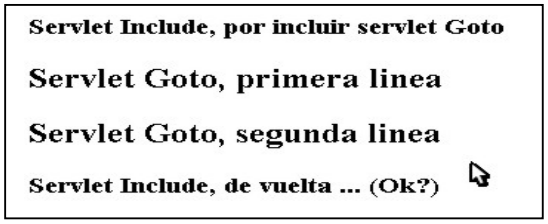
```
public class Goto extends HttpServlet {

  protected void processRequest(HttpServletRequest request,
                 HttpServletResponse response)
                 throws ServletException, IOException {
   response.setContentType("text/html;charset=UTF-8");
   PrintWriter out = response.getWriter();
   try {
```

```
      // TODO output your page here
      String option = (String) request.getAttribute("option");
      if (option != null)
       if (option.equals("forward")){
         out.println("<h3>UD proviene del servlet Forward y ha sido </h3>");
         out.println("<h1> redireccionado</h1>");
         out.println("<h3>a la pagina del servlet Goto... Todo bien?...</h3>");
       }else
          if (option.equals("include")){
            out.println("<h3>UD ha ejecutado el servlet Include; esta
                                                                  linea</h3>");
            out.println("<h1> será incluida</h1>");
            out.println("<h3> en la respuesta del servlet Goto... OK?...</h3>");
           }
       }
   finally {out.close();}
}
    @Override
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

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

Si ejecutamos **Forward.java**



Si ejecutamos **Include.java**



¿Qué es JSTL? (JSP Standard Tag Library)

Introducción

- **La especificación JSP ahora se ha convertido en una tecnología estándar para la creación de sitios Web dinámicos en Java.**
- **La librería JSTL es un componente dentro de la especificación del Java 2 Enterprise Edition (J2EE) y es controlada por Sun MicroSystems. JSTL no es más que un conjunto de librerías de etiquetas simples y estándares que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP. Las etiquetas JSTL están organizadas en 4 librerías:**
  - **core:** Comprende las funciones script básicas como loops, condicionales, y entrada/salida.
  - **xml:** Comprende el procesamiento de xml
  - **fmt:** Comprende la internacionalización y formato de valores como de moneda y fechas.
  - **ql:** Comprende el acceso a base de datos.

¿Cuál es el problema con los scriptlets JSP?

Ejemplo: Un scriptlet JSP: **<% int contador = 100; %>**

- El código Java embebido en scriptlets es desordenado.
- Un programador que no conoce Java no puede modificar el código Java embebido, anulando uno de los mayores beneficios de los JSP: permitir a los diseñadores y personas que escriben la lógica de presentación que actualicen el contenido de la página.
- El código de Java dentro de scriptlets JSP no pueden ser reutilizados por otros JSP, por lo tanto la lógica común termina siendo re-implementado(Repetido) en múltiples páginas.
- La recuperación de objetos fuera del HTTP Request y Session es complicada. Es necesario hacer el Casting de objetos y esto ocasiona que tengamos que importar más Clases en los JSP.

¿Como mejora esta situación la librería JSTL?

- Debido a que las etiquetas JSTL son XML, estas etiquetas se integran limpia y uniformemente a las etiquetas HTML.
- Las 4 librerías de etiquetas JSTL incluyen la mayoría de funcionalidad que será necesaria en una página JSP. Las etiquetas JSTL son muy sencillas de usarlas para personas que no conocen de programación, a lo mucho necesitarán conocimientos de etiquetas del estilo HTML.
- Las etiquetas JSTL encapsulan la lógica como el formato de fechas y números. Usando los scriptlets JSP, esta misma lógica necesitaría ser repetida en todos los sitios donde es usada, o necesitaría ser movida a Clases de ayuda.
- Las etiquetas JSTL pueden referenciar objetos que se encuentren en los ambientes Request y Session sin conocer el tipo del objeto y sin necesidad de hacer el Casting.
- Los JSP EL (Expression Language) facilitan las llamadas a los métodos Get y Set en los objetos Java. Esto no es posible en la versión JSP 1.2, pero ahora está disponible en JSP 2.0. EL es usado extensamente en la librería JSTL.

Ejemplo: JSTL sin scriptlets

Para demostrar el concepto de ordenamiento y mejor mantenimiento de los JSP, veamos 2 versiones de una página simple. La primera versión usa etiquetas scriptlets, y la segunda usa etiquetas JSTL. Ambas páginas implementan la misma lógica. Graban una lista de objetos AddressVO del Request y luego iteran a través de la lista, imprimiendo el atributo apellido de cada objeto (si el apellido no es null y de longitud diferente a 0). En cualquier otro caso, imprimirá "N/A". Finalmente, la página imprime la fecha actual.

Con scriptlets JSP:

```
 1  <%@ page import="com.ktaylor.model.AddressVO, java.util.*"%>
 2
 3  <p><h1>Customer Names</h1></p>
 4  <%        //     inicia scriplet
 5  List addresses = (List)request.getAttribute("addresses");
 6  Iterator addressIter = addresses.iterator();
 7   while(addressIter.hasNext()) {       //     Inicia bloque while
 8     AddressVO address = (AddressVO)addressIter.next();
 9     if((null != address) &&
10     (null != address.getLastName()) &&
11     (address.getLastName().length() > 0)) {   //     Inicio bloque if
12  %> //     interrumpe scriplet, necesitamos expresion Java
13     <%=address.getLastName()%><br/>    //     expresión Java
14  <% //     continuamos scriplet
15     }      //     fin bloque if
16     else { //     inicio bloque else
17  %>        //     interrumpimos scriplet
18     N/A<br/>      //     etiqueta avance
19  <%        //     continuamos scriplet
20     }      //     fin bloque else
21  }         //     fin bloque while
22  %>        //     termina scriplet
23  <p><h5>Last Updated on: <%=new Date()%></h5></p>
```

Nota: Observe la complicación resultante de tener que usar scriplets con sus cuerpos {} entremesclados con etiquetas conteniendo expresiones Java

Con JSTL (Unicamente):

```
 1  <%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
 2
 3  <p><h1>Customer Names</h1></p>
 4
 5  <c:forEach items="${addresses}" var="address">
 6    <c:choose>
 7       <c:when test="${not empty address.lastName}" >
 8         <c:out value="${address.lastName}"/><br/>
 9       </c:when>
10       <c:otherwise>
11         N/A<br/>
12       </c:otherwise>
13    </c:choose><br/>
14  </c:forEach><br/>
15
16  <jsp:useBean id="now" class="java.util.Date" />
17  <p><h5>Last Updated on: <c:out value="${now}"/></h5></p>
```

Nota: Codigo mucho mas corto y mucho mas inteligible

A continuación entramos en detalle. Usamos un material aportado por el Ing. Martin Polliotto, docente de la catedra.

Expression Language (EL)

La especificación 2.0 de JSP ha introducido una nueva librería estandar de etiquetas, denominada JSTL. Estas etiquetas tratan de abstraer la complejidad de introducir código Java (scriptlet) dentro de JSP, del mismo modo que trata de evitar que cada equipo de desarrollo cree un juego de etiquetas no estándar para las mismas labores.

Dentro de estas etiquetas, se utiliza un lenguaje llamado EL, lenguaje de expresiones, que pretende ser un lenguaje más sencillo que Java, para realizar operaciones. Uno de los primeros contenedores de JSP que soporta estas capacidades es Tomcat5.

Fichero descriptor

Lo primero que hay que hacer es cambiar el fichero descriptor:

Queremos interpretar EL **<el-ignored>false</el-ignored>**
No soportamos scriplet **<scripting-invalid>true</scripting-invalid>**
Queremos una cabecera por defecto <include-prelude> prelude.jspf
</includeprelude>
Y tambien un pie <include-coda> coda.jspf </include-coda>

Ejemplo:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd"
version="2.4">
<description>Ejemplos de EL</description>
<display-name>Ejemplos de JSP 2.0 </display-name>
<jsp-config>
     <jsp-property-group>
          <description>Definicion de atributos</description>
          <display-name>ConfiguracionJSP</display-name>
          <url-pattern>/*</url-pattern>
          <el-ignored>false</el-ignored>
          <page-encoding>ISO-8859-1</page-encoding>
          <scripting-invalid>true</scripting-invalid>
          <include-prelude>prelude.jspf</include-prelude>
          <include-coda>coda.jspf</include-coda>
     </jsp-property-group>
</jsp-config>
</web-app>
```

EL

Antes de JSTL, JSP usaba Java para referenciar atributos dinámicos. Con JSTL ya no es necesario. Compara por ejemplo, la lectura de un parámetro:

```
<%-- con JSP, usando una expresión Java --%>
<%= request.getParameter("login") %>

<%-- con JSTL, usando Expresión Language  --%>
${param.login}
```

En JSTL se evita el uso de Java proporcionando un lenguaje de expresiones llamado EL (Expression Language). EL no es un lenguaje de programación en sí mismo, su único propósito es

- Referenciar objetos y sus propiedades
- Escribir expresiones simples.

EL se basa en las siguientes normas:
Las expresiones JSTL comienzan con ${ y terminan con }. Lo que hay en medio es tratado como una expresión Java.

- Las expresiones se componen de
  - identificadores. Hay once identificadores reservados que corresponden a once objetos implícitos (los veremos luego). El resto de identificadores sirven para crear variables.
  - Literales. Son números, cadenas delimitadas por comillas simples o dobles, y los valores true, false, y null.
  - Operadores. Permiten comparar y operar con identificadores y literales.
  - Operadores de acceso. Se usan para referenciar propiedades de los objetos.
- Podemos usar expresiones en cualquier parte del documento, o como valores de los atributos de etiquetas JSTL, exceptuando los atributos var y scope, que no aceptan expresiones.
- En cualquier sitio donde sea valido colocar una expresión, también será válido colocar más de una.
- Por ejemplo: value="Hola ${nombre} ${apellidos}".
- Las expresiones pueden contener operaciones aritméticas, comparaciones, operaciones booleanas, y agruparse mediante paréntesis. También pueden decidir cuando una variable existe o no (usando empty).
- Para acceder al campo de un bean java, o a un elemento de una colección (array, o Map), se usa el operador punto, o el operador corchete:

  ```
  ${bean.propiedad}
  ${map.elemento}
  ${header['User-Agent']}
  ```

- Si un campo de un bean es otro bean, podemos encadenar puntos para acceder a una propiedad del segundo bean:

  ```
  ${bean1.bean2.propiedad}
  ```

- Las expresiones pueden aparecer como parte del valor de un atributo de una etiqueta:

  ```
  <c:out value="${2+2}"/>
  <c:out value="${nombre}"/>
  <c:if test="${tabla.indice % 2 == 0}">es par</c:if>

  O independientemente junto a texto estático como el HTML:
  <input type="text" name = "usuario" value =
                        "${requestScope.usuario.nombre}"/>
  ```

Operadores

Los operadores y sus significados son los mismos que en otros lenguajes de programación

Resumen de operadores

Operador	Descripción
.	Acceso a propiedad o elemento de un Map.
[]	Acceso a elemento de un array o de un List.
()	Agrupación de expresiones.
: ?	Expresión condicional: condición ? valorSiCierto: valorSiFalso
+,-,*,/(o div),%(o mod)	Suma, resta, multiplicación, división, resto.
==(o eq),!=(o ne),<(o lt), >(o gt),<=(o le),>= (o ge)	Igualdad, desigualdad, menor que, mayor, menor o igual que, mayor o igual que.
&&(o and),||(o or),!(o not)	Operadores lógicos and, or, y not.
empty	Comprueba si una colección (array, Map, Collection) es null o vacía.
función(argumentos)	Llamada a una función.

Resumen de precedencia de operadores

[], .
()
- unario, not, !, empty
*, /, div, %, mod
+, - binario
(), <, >, <=, >=, lt, gt, le, ge
==, !=, eq, ne
&& (o and)
|| (o or)

Acceso a datos

- El operador punto (.) permite recuperar la propiedad de un objeto por su nombre. Ejemplo:
  ${libro.paginas}
- Las propiedades se recuperan usando las convenciones de los JavaBeans. Por ejemplo, en el código anterior, invocábamos el método libro.getPaginas() del objeto libro.
- El operador corchete ([]) permite recuperar una propiedad con nombre o indexada por número. Ejemplo:
  ${libro["paginas"]}
  ${libro[0]}
  ${header['User-Agent']}
- Dentro del corchete puede haber una cadena literal, una variable, o una expresión.

Operadores aritméticos

Operador	Descripción
+	suma
-	resta
*	multiplicación
/ div	división
% mod	resto

Operadores lógicos

Operador	Descripción
&& and	true si ambos lados son ciertos
|| or	true si alguno de los lados son ciertos
! not	true si la expresión es falsa

Operadores relacionales

Operador	Descripción	Ejemplo	Resultado
== eq	comprobación de igualdad	${5 == 5}	true
!= ne	comprobación de desigualdad	${5 != 5}	false
< lt	menor que	${5 < 7}	true
> gt	mayor que	${5 > 7}	false
<= le	menor o igual que	${5 le 5}	true
>= ge	mayor o igual que	${5 ge 6}	false

Operador empty

- El operador empty comprueba si una colección o cadena es vacía o nula.
  ${empty param.login}
- Otra manera de hacerlo es usando la palabra clave null:
  ${param.login == null}

Literales

En expresiones EL podemos usar números, caracteres, booleanos (palabras clave true, false), y nulls (palabra clave null).

Objetos implícitos

- Ciertos objetos son automáticamente accesibles a cualquier etiqueta JSP.
- A través de ellos es posible a cualquier variable de los ámbitos page, request, session, application,a parametros HTTP, cookies, valores de cabeceras, contexto de la página, y parámetros de inicialización del contexto.
- Todos, excepto pageContext, están implementados usando la clase java.util.Map.

Objetos implicitos	contiene
pageScope	Variables de ámbito página.
requestScope	Variables de ámbito request.
sessionScope	Variables de ámbito session.
applicationScope	Variables de ámbito application.
param	Parámetros del request como cadenas.
paramValues	Parámetros del request como array de cadenas.
header	Cabeceras del request HTTP como cadenas.
headerValues	Cabeceras del request HTTP como array de cadenas.
cookie	Valores de las cookies recibidas en el request.
initParam	Parametros de inicialización de la aplicación Web.
pageContext	El objeto PageContext de la página

Ejemplo:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>

<%-- ACCESO A TODOS LOS ELEMENTOS --%>
<c:forEach items="${header}" var="item">
     <c:out value="${item}"/><br/>
</c:forEach>
<br/>

<%-- ACCESO POR NOMBRE CON PUNTO: objeto.propiedad --%>
Host: <c:out value="${header.Host}"/> <br/>
```

```
<%-- ACCESO POR NOMBRE CON CORCHETES: objeto['propiedad']
     Usar este formato evita que se interprete el nombre como una
     expresión. Por ejemplo: queremos la propiedad "Accept-Language" no
     una resta entre las variables Accept y Language.
--%>

Host: <c:out value="${header['Host']}"/> <br/>
Accept-Language: <c:out value="${header['Accept-Language']}"/> <br/>
```

El resultado es:

```
accept-encoding=gzip, deflate
cache-control=no-cache
connection=Keep-Alive
referer=http://127.0.0.1:8080/sampleJSTL/
host=127.0.0.1:8080
accept-language=es
user-agent=Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; MyIE2;
.NET CLR
1.1.4322)
cookie=JSESSIONID=E58368E8FFA066A654750C4CED08AA88
accept=*/*
Host: 127.0.0.1:8080
Host: 127.0.0.1:8080
Accept-Language: es
```

pageContext

Aquí se listan algunos de los valores disponibles en el contexto de la página. Existen otros que podemos consultar en el javadoc. Por ejemplo, pageContext.request.authType esta ejecutando el método ttpServletRequest.getAuthType().

De igual modo podemos acceder al resto de métodos.

Expresión	**Descripción**
${pageContext.exception.message}	Devuelve una descripción del error cuando la página en curso es una página de error JSP.
${pageContext.errorData}	Información de un error JSP.
${pageContext.request.authType}	Tipo de autentificación usado en la página.
${pageContext.request.remoteUser}	Identificador del usuario (cuando esta en uso la autentificación del contenedor).
${pageContext.request.contextPath}	Nombre (también llamado contexto)de la aplicación.
${pageContext.request.cookies}	Array de cookies.
${pageContext.request.method}	Método HTTP usado para acceder a la página.
${pageContext.request.queryString}	Query de la página (el texto de la URL que viene después del PATH)
${pageContext.request.requestURL}	URL usada para acceder a la página.
${pageContext.session.new}	Contiene true si la sesión es nueva, false si no lo es.
${pageContext.servletContext.serverInfo}	Información sobre el contenedor JSP.

pageContext.errorData

- En JSP 2.0, cuando se produce un error la excepción ocurrida está disponible en el ámbito request con el nombre javax.servlet.error.exception. Ademas, se adjunta al contexto de la

página, información adicional en forma de bean con nombre errorData. Este bean tiene las siguientes propiedades:

Propiedad	Tipo Java	Descripción
requestURI	String	URI de la petición fallida.
servletName	String	Nombre de la página o servlet que lanzó la excepción.
statusCode	int	Código HTTP del fallo.
throwable	Throwable	La excepción que produjo el fallo.

- Para declarar una página de error JSP que responda a un código o excepción, podemos usar el web.xml:

```
<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>
```

- O una declaración para una página concreta:

```
<%@ page errorPage="/error2.jsp" contentType="text/html" %>
```

_ Ejemplo de página de error:

```
<%@ page isErrorPage="true" contentType="text/html" %>
<pre>
     URI de la petición fallida: ${pageContext.errorData.requestURI}
     Quien lanzo el error: ${pageContext.errorData.servletName}
     Código del fallo: ${pageContext.errorData.statusCode}
     Excepción: ${pageContext.errorData.throwable}
     La misma excepción anterior:
     ${requestScope['javax.servlet.error.exception']}
</pre>
```

Funciones EL

- Las funciones representan un modo de extender la funcionalidad del lenguaje de expresiones. Se usan dentro de cualquier expresión EL, ejemplo: Mi nombre de usuario tiene ${fn:length(username)} letras.
- En el descriptor de la biblioteca de funciones, se establece una correspondencia entre el nombre de la función y un método estático de una clase Java.

```
<function>
     <description> Returns the number of items in a collection or the
          number of characters in a string.
     </description>
     <name>length</name>
     <function-class> org.apache.taglibs.standard.functions.Functions
     </function-class>
     <function-signature>int length(java.lang.Object) </function-
                                                          signature>
</function>
```

- En JSTL 1.1 existen 16 funciones de manipulación de cadenas:
  - Alternar una cadena entre mayúsculas/minúsculas: toLowerCase, toUpperCase.

- Obtener una subcadena: substring, substringAfter, substringBefore.
- Eliminar el espacio en blanco en ambos extremos de una cadena: rim.
- Reemplazar caracteres en una cadena: replace.
- Comprobar si una cadena contiene a otra: indexOf, startsWith, endsWith, contains, containsIgnoreCase.
- Convertir una cadena en array (split), o un array en cadena (join).
- Codificar los caracteres especiales del XML: escapeXml.
- Obtener la longitud de una cadena: length.

- La función length también puede aplicarse a instancias de java.util.Collection, caso en el cual, devolverá el número de elementos de una colección.
- Todas las funciones EL interpretan las variables no definidas o definidas con valor igual a null, como cadenas vacías.

fn:contains

- Devuelve true si una cadena contiene a otra, false en caso contrario.
  fn:contains(string, substring) # boolean
  - Si la cadena a buscar es vacía, siempre existirá coincidencia.

Ejemplo:

```
<%-- Muestra mensaje si la variable query contiene la cadena 'sex' --%>
<c:if test="${fn:contains('sex', query)}">
   Access control configuration prevents your request from being allowed.
   Offending word: sex
</c:if>
```

fn:containsIgnoreCase

• Devuelve true si una cadena contiene a otra sin distinguir entre minúsculas/mayúsculas, false en caso contrario.

fn:containsIgnoreCase(string, substring) # boolean

• Si la cadena a buscar es vacía, siempre existirá coincidencia.

Ejemplo:

```
<%-- Muestra un mensaje si 'james stratchan' es subcadena de username
     sin distinguir entre mayúsculas o minúsculas. --%>

<c:set var="username" value="James Stratchan"/>
<c:if test="${fn:containsIgnoreCase('james stratchan', username)}">
Hi James Stratchan!
</c:if>
```

fn:endsWith

• Devuelve true si una cadena termina con el sufijo indicado, false en caso contrario.

fn:endsWith(string, suffix) # boolean

• Si la cadena a buscar es vacía, siempre existirá coincidencia.

• Ejemplo:

```
<c:set var="username" value="James Stratchan"/>
<c:if test="${fn:endsWith(username, 'tratchan')}">
     Hi James Stratchan!
</c:if>
```

fn:escapeXml
• Codifica los caracteres que podrían ser interpretados como parte de las etiquetas XML.
fn:escapeXml(string) # String
• Es equivalente a c:out con atributo xmlEscape="true".

fn:indexOf
• Devuelve el índice (empezando desde cero) dentro de una cadena donde comienza otra cadena especificada.

fn:indexOf(string, substring) # int
• Si la cadena a buscar es vacía devuelve cero (comienzo de la cadena). Si la cadena a buscar no se encuentra devuelve -1.

fn:join
• Convierte en cadena los elementos de un array. El separador se usa para
separar los elementos en la cadena resultante.

fn:join(array, separator) # String
• Si el array es nulo devuelve la cadena vacía.

fn:length
• Devuelve el número de caracteres en una cadena, o de elementos en una colección.

fn:length(input) # integer
• Devuelve cero para cadenas vacías.

fn:replace
• Recorre una cadena (inputString) reemplazando todas las ocurrencias de una cadena (beforeSubstring) por otra (afterSubstring).

fn:replace(inputString, beforeSubstring, afterSubstring) # String
• El texto reemplazado no es vuelto a procesar para realizar más reemplazos. Si afterSubstring es vacía se eliminan todas las ocurrencias de beforeSubstring.

fn:split
• Divide una cadena en subcadenas, usando como separador cualquier carácter de otra cadena. Los caracteres de la segunda cadena no forman parte del resultado.

fn:split(string, delimiters) # String[]
• No se realiza reemplazo alguno cuando:
La cadena a dividir es vacía (se devuelve la cadena vacía).
La cadena de delimitadores es vacía (se devuelve la cadena original

fn:startsWith
• Comprueba si una cadena comienza por otra.

fn:startsWith(string, prefix) # boolean
• Devuelve true si la cadena a buscar es vacía.

fn:substring
• Devuelve un subconjunto de una cadena. Los índices comienzan en cero. La cadena resultante comienza en beginIndex y llega hasta el caracter en endIndex-1 (incluido). La longitud de la cadena es endIndex-beginIndex.

fn:substring(string, beginIndex, endIndex) # String
_ Si beginIndex es menor que cero se ajusta su valor a cero.
_ Si beginIndex es mayor que la longitud de la cadena se ajusta su valor a la longitud de la cadena.
_ Si endIndex es menor que cero o mayor que la longitud de la cadena, se ajusta su valor a la longitud de la cadena.
_ Si endIndex es menor que beginIndex se devuelve una cadena vacía.

fn:substringAfter
• Devuelve la porción de una cadena que viene a continuación de Cierta subcadena. La cadena devuelta comienza en el siguiente caracter tras la subcadena buscada.

fn:substringAfter(string, substring) # String
_ Si no se encuentra la subcadena o si esta es vacía se devuelve la cadena original.

fn:substringBefore
• Devuelve la porción de una cadena que viene antes de cierta subcadena. La cadena devuelta va desde el primer carácter hasta el carácter anterior a la subcadena buscada.

fn:substringBefore(string, substring) # String
• Si no se encuentra la subcadena o si esta es vacía se devuelve la cadena original.

fn:toLowerCase
• Convierte todos los caracteres a minúsculas.

fn:toLowerCase(string) # String

fn:toUpperCase
• Convierte todos los caracteres a mayúsculas.

fn:toUpperCase(string) # String

fn:trim
• Elimina el espacio en blanco en los extremos de una cadena.

fn:trim(string) # String

Material sobre este mismo tema y autor, resumido en filminas, sigue

JSP y EL

JSP Expression Language

- Mecanismo de evaluación de expresiones
- La especificación de **JSP 2.0** introduce el Expression Language (EL) junto con **JSTL**
- Diseñado para ser mas amigable que el manejo de scriptlets
- Similar a JavaScript
- Permite acceder a objetos implícitos, variables locales y objetos **Beans** de una manera mas sencilla que utilizando scriptles

EL – Sintaxis básica

${ expresion }

- EL ha sido creado para reemplazar a los scriptlets
- Deshabilitación de scriptlets en web.xml

```
<jsp-config>
  <jsp-property-group>
      <url-pattern>*.jsp</url-pattern>
      <scripting-enabled>false</scripting-
  enabled>
  </jsp-property-group>
</jsp-config>
```


Deshabilitacion de EL

- Directiva de pagina

```
<%@ page isELIgnored="true" %>
```

- En web.xml

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-enabled>false</el-enabled>
</jsp-property-group>
```


Evaluacion aritmetica con EL

- Addition: +
- Subtraction: –
- Multiplication: *
- Exponents: E
- Division: / or div
- Modulus: % or mod

Operadores en EL

- Operadores lógicos

&&	ó	and
||	ó	or
!	ó	not

- Otros operadores

  Para acceder a object.property se emplea:

  object["property"]

Comparación en EL

- Resultado de la comparación: **true** o **false**
- Operadores de comparación

==	ó	eq
!=	ó	ne
<	ó	lt
>	ó	gt
<=	ó	le
>=	ó	ge

JavaBeans y EL

```
<jsp:getProperty name="myBean"
  property="name" />
```

- Una alternativa con empleo de scriptlet es

```
<%= myBean.getName()%>
```

- En EL

```
${myBean.name}
```


Busqueda de variables en EL

${producto}

- Esta expresión buscará el atributo producto en la página, en el pedido, en la sesión y en el alcance de la aplicación e imprimirá su valor.
- Si el atributo no es encontrado **null** es el valor retornado

Ejemplo EL – templateText.jsp

```
<html>
<body>
<table border="1"> <tr> <td>Hola
  ${param['name']}</td> <td> </td> </tr>
<tr>
<form action="templateText.jsp" method="post">
<td><input type="text" name="name"></td>
<td><input type="submit"></td>
</form>
</tr>
</table>
</body>
</html>
```


EL y JavaBeans

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>SimpleBean</title>
  </head>
  <body> <body> <h2>EL y JavaBeans</h2>
    <jsp:useBean id="person" scope="request" class="bean.SimpleBean">
      <jsp:setProperty name="person" property="*"/>
    </jsp:useBean>
    <table border="1">
      <tr> <td>${person.name}</td> <td>${person.age}</td>
        <td> </td> </tr>
      <tr>
      <form action="simpleBean.jsp" method="post">
        <td><input type="text" name="name"></td>
        <td><input type="text" name="age"></td>
        <td><input type="submit"></td>
      </form>
    </tr> </table>
</body>  </html>
</html>
```


EL y JavaBeans

nombres que los parámetros

```
<jsp:useBean id="person" class="com.ch03.Person" scope="request">
    <jsp:setProperty name="person" property="*"/>
</jsp:useBean>
```

- Acceder a las propiedades de un JavaBean

```
<tr>
<td>${person.name}</td> <td>${person.age}</td>
</tr>
ó
<tr>
<td>${person["name"]}</td> <td>${person["age"]}</td>
</tr>
```


Acceso a propiedades anidadas

```
<tr>
<td>${person.name}</td>
<td>${person.age}</td>
<td>${person["address"].line1}</td>
<td>${person["address"].town}</td>
<td>${person.address.phoneNumbers[0].std}
  ${person.address.phoneNumbers[0].number}</td
  >
<td>${person.address.phoneNumbers[1].std}
  ${person.address.phoneNumbers[1].number}</td
  >
</tr>
```

Objetos implicitos

Objeto	Significado
request	el objeto HttpServletRequest asociado con la petición
response	el objeto HttpServletResponse asociado con la respuesta
out	el Writer empleado para enviar la salida al cliente. La salida de los JSP emplea un buffer que permite que se envíen cabeceras HTTP o códigos de estado aunque ya se haya empezado a escribir en la salida (out no es un PrintWriter sino un objeto de la clase especial JspWriter).
session	el objeto HttpSession asociado con la petición actual. En JSP, las sesiones se crean automáticamente, de modo que este objeto está instanciado aunque no se cree explícitamente una sesión.
application	el objeto ServletContext, común a todos los servlets de la aplicación web.
config	el objeto ServletConfig, empleado para leer parámetros de inicialización.
pageContext	permite acceder desde un único objeto a todos los demás objetos implícitos
page	referencia al propio servlet generado (tiene el mismo valor que this).Como tal, en Java no tiene demasiado sentido utilizarla, pero está pensada para el caso en que se utilizara un lenguaje de programación distinto.
exception	Representa un error producido en la aplicación. Solo es accesible si la página se ha designado como página de error (mediante la directiva page isErrorPage).

Ejemplos de objetos implicitos

```
<tr> <td>${pageContext.request.requestURI}</td>
</tr>
<tr> <td>${sessionScope.session.person.name}</td>
</tr>
<tr> <td>${requestScope.request.person.name}</td>
</tr>
<tr> <td>${param["name"]}</td> </tr>
<tr> <td>${paramValues.multi[1]}</td> </tr>
```


Funciones en EL

- Las funciones representan un modo de extender la funcionalidad del **EL**
- Se usan dentro de cualquier expresión **EL**, ejemplo:

Mi nombre de usuario tiene {fn:length(username)} letras.

Deshabilitacion de EL

- Directiva de pagina

```
<%@ page isELIgnored="true" %>
```

- En web.xml

```
<jsp-property-group>
  <url-pattern>*.jsp</url-pattern>
  <el-enabled>false</el-enabled>
</jsp-property-group>
```


Funciones en EL

En JSTL 1.1 existen 16 funciones de manipulación de cadenas:

- Alternar una cadena entre mayúsculas/minúsculas: **toLowerCase**,
- **toUpperCase**.
- Obtener una subcadena: **substring**, **substringAfter**, **substringBefore**.
- Eliminar el espacio en blanco en ambos extremos de una cadena: **trim**
- Reemplazar caracteres en una cadena: **replace**.
- Comprobar si una cadena contiene a otra: **indexOf**, **startsWith**,
- **endsWith**, **contains**,
- **containsIgnoreCase**.
- Convertir una cadena en array (**split**), o un array en cadena (**join**).
- Codificar los caracteres especiales del XML: **escapeXml**.
- Obtener la longitud de una cadena: **length**.

- La función **length** también puede aplicarse a instancias de java.util.Collection, caso en el cual, devolverá el número de elementos de una colección.
- Todas las funciones EL interpretan las variables no definidas o definidas con valor igual a null, como cadenas vacías.

A continuación, del mismo autor, filminas con el detalle del contenido de bibliotecas JSTL

Que es JSTL (Java Standard Tag Library)

¿Qué es JSTL?

- JavaServer Pages **S**tandard **T**ag **L**ibrary
- La librería JSTL es un componente dentro de la especificación del Java 2 Enterprise Edition (J2EE) y es controlada por Sun MicroSystems. **JSTL no es más que un conjunto de librerías de etiquetas simples y estándares** que encapsulan la funcionalidad principal que es usada comúnmente para escribir páginas JSP.
- **Objetivo**:
  - Simplificar y agilizar el desarrollo de aplicaciones web permitiendo generar dinámicamente páginas WEB de una forma muy sencilla

¿Cómo se compone JSTL?

- **Es una colección de cuatro librería de tags**

1. **Core**: acciones de propósito generales
2. **Internalización y formateo:** adaptaciones a varios lenguajes y regiones
3. **Acceso a base de datos relacionales**
4. **Procesamiento XML**

JavaServer Pages Standard Tag Library – JSTL

- La especificacion JSTL 1.0 fue liberada en junio de 2002
- Para su uso se requiere al menos Servlet 2.3 y JSP 1.2
- Apache Jakarta Project provee una implementación de JSTL, NetBeans trae una implementación de SUN
- Se necesitan dos JAR: jstl.jar y standard.jar en el subdirectorio WEB-INF/lib

Configuración del deployment descriptor – web-xml

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
   version="2.4">
<taglib>
   <taglib-uri>http://java.sun.com/jstl/core</taglib-uri>
   <taglib-location>/WEB-INF/tlds/c.tld</taglib-location>
</taglib>
<taglib>
   <taglib-uri>http://java.sun.com/jstl/fmt</taglib-uri>
   <taglib-location>/WEB-INF/tlds/fmt.tld</taglib-location>
</taglib>
<taglib>
   <taglib-uri>http://java.sun.com/jstl/xml</taglib-uri>
   <taglib-location>/WEB-INF/tlds/x.tld</taglib-location>
</taglib>
<taglib>
   <taglib-uri>http://java.sun.com/jstl/sql</taglib-uri>
   <taglib-location>/WEB-INF/tlds/sql.tld</taglib-location>
</taglib>
</web-app>
```

Archivos TLDs

- Dentro del directorio META-INF del archivo standard.jar se encuentran los archivos c.tld, fmt.tld, sql.tld y x.tld, que deberan ubicarse en el directorio WEB-INF/tld
- Actualmente JSTL viene incorporado al estandar JSP 2.0

Librerias de Tags gemelas

- Debido a que JSTL 1.0 fue liberado anterior a JSP 2.0, soporta completamente los conteiners JSP 1.2
- Para soportar ambos mundos, scripting (rtexprvalues) y EL (elexprvalues), se crearon un conjunto de librerias gemelas
- Los URI de las dos librerias son similares, pero a las librerias de runtime se le agrega "_rt" tanto al URI como al prefix
- Es posible mezclar el uso de acciones runtime y EL

Expression Language Tag Libraries

Functional Area	URI	Prefix
Core	http://java.sun.com/jstl/core	c
XML processing	http://java.sun.com/jstl/xml	x
I18N and formatting	http://java.sun.com/jstl/fmt	fmt
Relational database access	http://java.sun.com/jstl/sql	sql

Core Tag Library

La librería JSTL core cubre las siguiente areas funcionales:

- Manipulacion de variables
- Condicionales
- Looping e iteraciones
- Manipulacion URL

Acción <c:out> – Manipulación variables

- Es equivalente a <%= expresion %>

```
<%@ taglib uri="http://java.sun.com/jstl/core"
  prefix="c" %>
<c:out value="Good Afternoon!" />
<c:out value="${book.author.name}"
  default="Unknown"/>

<%@ taglib uri="http://java.sun.com/jstl/core"
  prefix="c" %>
<c:out value="${book.author.name}">
  Unknown
</c:out>
```


Acción <c:set> – Manipulación variables

- Establece un valor de una variable en un particular alcance (pagina, request, session, o aplicación)

```
<%@ taglib uri="http://java.sun.com/jstl/core"
  prefix="c" %>
<c:set var="browser" value="${header['User-
  Agent']}" scope="session" />
Your Browser User Agent is : <c:out
  value="${browser}"/>
```

Accion <c:remove> – Manipulacion variables

- Remueve una variable de un determinado alcance

```
<%@ taglib uri="http://java.sun.com/jstl/core"
  prefix="c" %>
<c:set var="browser" value="${header['User-
  Agent']}" scope="session" />
<c:remove var="browser" scope="session">
Your Browser User Agent is : <c:out
  value="${browser}"/>
```


Accion <c:catch> – Manipulacion variables

- Provee un mecanismo para capturar cualquier excepción java.lang.Throwable que puede ocurrir durante la ejecución de cualquier accion anidada

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c"
  %>
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt"
  %>
<c:catch var="exception">
  fmt:parseDate var="dob" value="${param.birthDate}"
  pattern="yyyy-MM-dd" />
</c:catch>
<c:if test="${exception != null}">
  <jsp:useBean id="dob" class="java.util.Date" />
</c:if>
```


Accion <c:if> – Condicionales

- **Si el resultado de evaluar la expresión es true, el contenido del cuerpo será procesado**

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<%@ page import="com.ch04.CalendarBean"%>

<jsp:useBean id="cal" class="com.ch04.CalendarBean"/>
The time is currently : <BR><BR>
<c:out value="${cal.time}"/>,
<c:set var="hour" value="${cal.hour}" scope="request" />
<b>
<c:if test="${hour >= 0 && hour <=11}">
   Good Morning!
</c:if>
<c:if test="${hour >= 12 && hour <=17}">
   Good Afternoon!
</c:if>
<c:if test="${hour >= 18 && hour <=23}">
   Good Evening!
</c:if>
</b>
```


<c:choose>, <c:when>, y <c:otherwise>

- Un numero ilimitado de <c:when> puede existir en una acción <c:choose>

```
<c:choose>
   <c:when test="${hour >= 0 && hour <=11}">
      Good Morning!
   </c:when>
   <c:when test="${hour >= 12 && hour <=17}">
      Good Afternoon!
   </c:when>
   <c:otherwise>
      Good Evening!
   </c:otherwise>
</c:choose>
```


Accion <c:forEach> – Looping e iteraciones

- **Sintaxis 1 – Iteraccion sobre una colección de objetos**

```
<c:forEach[var="varName"] items="collection"
  [varStatus="varStatusName"] [begin="begin"]
  [end="end"] [step="step"]>
  body content
</c:forEach>
```

- **Sintaxis 2 – Iteraccion de un numero fijo de veces**

```
<c:forEach [var="varName"]
  [varStatus="varStatusName"] begin="begin"
  end="end" [step="step"]>
  body content
</c:forEach>
```

<c:forEach>

Iteractuando sobre una colección (sintaxis 1), el atributo items puede ser un objeto de uno de los siguiente tipos:

- Un array
- Una implementacion de java.util.Collection
- Una implementacion de java.util.Iterator
- Una implementacion de java.util.Enumeration
- Una implementacion de java.util.Map
- Un string de valores separados por coma

<c:forEach>

Attribute	Required	Default	Description
`items`	No	None	Collection of items to iterate over. If `null` then no iteration takes place
`begin`	No	0	If specified, must be >=0 *If `items` specified:* Iteration begins at the item located at the specified value *If `items` not specified:* Iteration stats from specified value
`end`	No	Last object in the collection	If specified must be >=`begin` *If `items` specified:* Iteration stops at the item at the specified index (inclusive) *If `items` not specified:* Iteration stops when index reaches specified value
`step`	No	1 (process all objects)	If specified must be >= 1 Iteration processes every `step` item in the collection
`var`	No	None	Name of the scoped variable that represents the current item of the iteration
`varStatus`	No	None	Name of the scoped variable that represents the status of the iteration Object is of type `javax.servlet.jsp.jstl.core.LoopStatus`

<c:forEach>

```
<c:forEach var="book"
  items="${sessionScope.books}">
  <tr>
    <td align="right" bgcolor="#ffffff">
      <c:out value="${book.title}"/>
    </td>
  </tr>
</c:forEach>
```


Accion <c:forTokens>

- Iteractua sobre un string separado por un conjunto de delimitadores

```
<c:forTokens items="stringOfTokens"
  delims="delimiters" [var="varName"]
  [varStatus="varStatusName"] [begin="begin"]
  [end="end"] [step="step"]>
  body content
</c:forEach>
```


<c:forTokens>

Attribute	Required	Default	Description
items	Yes	None	The string to tokenize
delims	Yes	None	The delimiter characters that separate the tokens of the string
begin	No	0	If specified, must be >=0 The token to start with
end	No	Last token in the string to be tokenized	If specified must be >=begin The token to end with
step	No	1 (process all objects)	If specified must be >= 1 Iteration processes every step item in the collection
var	No	None	Name of the scoped variable that represents the current item of the iteration
varStatus	No	None	Name of the scoped variable that represents the status of the iteration. Object is of type javax.servlet.jsp.jstl.core.LoopStatus

<c:forTokens>

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<c:set var="queryResult" value="Dan,Jepp,Male,26,Java
  Developer,London" scope="request" />

<html> <body> <table border="1">
<tr> <th>First Name</th> <th>Last Name</th> <th>Sex</th>
    <th>Age</th> <th>Occupation</th> <th>Location</th>
</tr>
<tr>
<c:forTokens items="${queryResult}" delims="," var="token">
  <td><c:out value="${token}"/></td>
</c:forTokens>
</tr>
</table> </body> </html>
```


Accion <c:import> – Manipulacion de URL

- Importa el contenido de un recurso basado en un URL

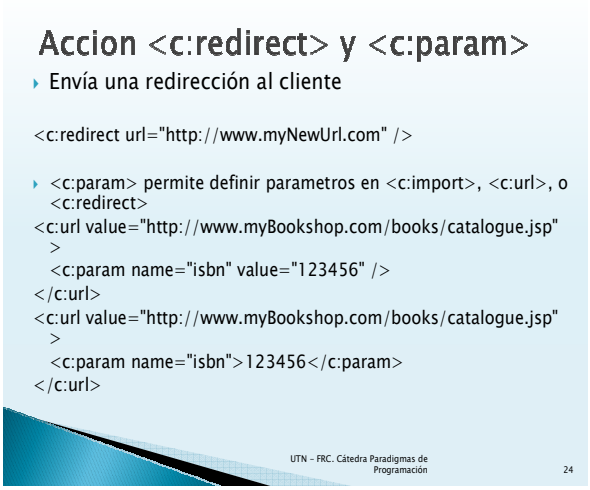
```
<%@ taglib uri="http://java.sun.com/jstl/core"
  prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/xml"
  prefix="x" %>

<c:import url="http://mybokstore.com/book.xml"
  var="url" />
<x:parse xml="${url}" var="book" scope="session"
  />
```

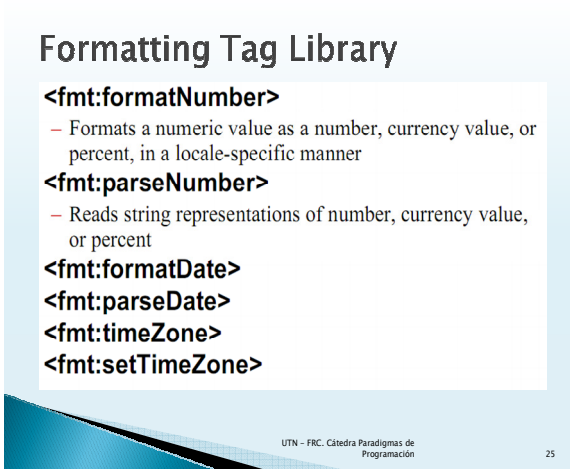

Accion <c:url> – Manipulacion de URL

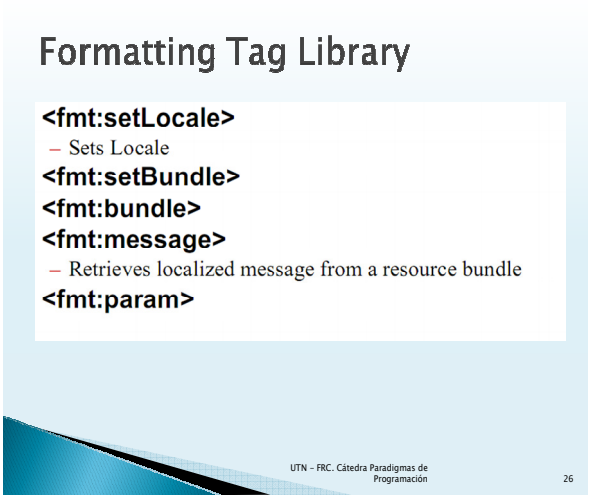
- Provee una forma sencilla de construir correctamente una URL, aplicando las reglas de reescritura. Una reescritura de URL luce como:

http://www.myserver.com/shop/checkout.jsp;jsessionid=42eab543dc2

```
<c:url
  value="http://www.myserver.com/shop/checkout
  .jsp" />
```







A continuación 4(cuatro) proyectos Net Beans, ejemplos proporcionados por el Ing. Martin Pollioto, docente de la catedra, usando distintas alternativas. Cada proyecto contiene un enunciado y es acompañado por su respectiva codificacion. Los proyectos son los siguientes:

- Modelo MVC, proyecto **MVCBookStore**
- Modelo Custom Tag, proyecto **PycJTagSinBody**
- Modelo combinado, proyecto PycProdeRegional2010
- Modelo 2 JSP + JavaBean, proyecto

Tema Principal: Introducción a la Programación Distribuida.

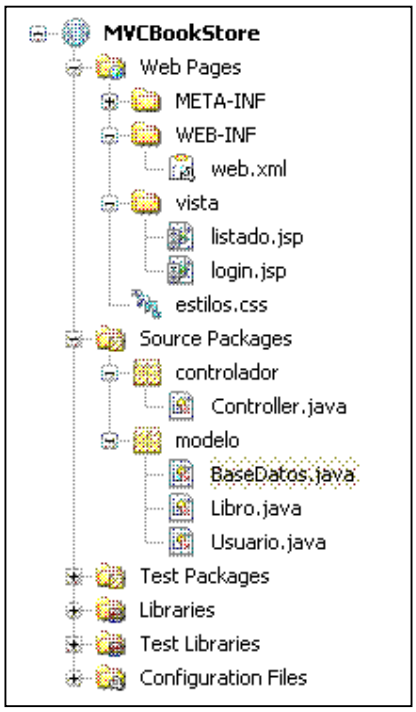
Temas Particulares: Aplicaciones Distribuidas con Java. Servlets. Introducción al manejo de JSPs. Elementos HTML. Patrón MVC. JSTL/EL.

Caso de estudio: "BookStore"

Se nos ha solicitado desarrollar una aplicación Web que permita gestionar las consultas de libros en una importante librería con sucursales en distintas provincias. Se necesita poder acceder, a través de una conexión de Internet al servidor de la empresa (http://localhost:8084/MVCBookStore/), de modo de poder consultar el stock de libros de la librería. Como política de seguridad, la empresa solicitó desarrollar una página de login que permita validar los usuarios registrados de la empresa. Una vez validado el usuario podrá realizar las consultas pertinentes.

El proyecto y las interfaces Web modeladas se muestran a continuación:

Proyecto



Login.jsp

listado.jsp

Veamos sus componentes

Vista/login.jsp

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%--
    Document   : login
    Created on : 01/08/2009, 23:58:18
    Author     : fenix
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <title>Solo usuarios registrados</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <link rel="stylesheet" href="estilos.css" type="text/css">
    </head>

    <body>
        <form name="frmLogin" action="Controller" method="POST">
            <input type="hidden" value="login" name="operacion">
            <!--
                El input 'operacion' permite enviar, sin mostrar al usuario,
                la acción que deberá procesar nuestro servlet Controller
                cuando reciba la petición desde este formulario... --><br/>
            <div>
                <table>
                    <h3>Librería Web <br>Sistema de consulta Online</h3>
                    <br/>
                    <h4 style="color:white;">Ingrese su nombre de usuario y
                                                          contraseña</h4>
                    <tbody style="">
                        <tr>
                            <td>Usuario</td>
                            <td><input type="text" name="usuario" value=""
                                                           size="25" /></td>
                        </tr>
                        <tr>
                            <td>Password</td>
                            <td><input type="password" name="clave" value=""
                                                          size="25" /></td>
                        </tr>
                    </tbody>
                </table>
                <input type="submit" value="Ingresar" name="btnLogin"
                             style="font-family:arial;font-size:11px;"/><br/>
                <!-- código JSTL/EL -->
                <c:if test="${requestScope.msgError ne null }">
                    <h5 style="color:red;">
                        <c:out value="${requestScope.msgError}"/>
                    </h5>
                </c:if>

                <!--   Codigo scriplet
                <h5 style="color:red;">
                   <%String msgError = (String)request.getAttribute("msgError");
                       if(msgError!=null) out.println(msgError);
                    %>
                </h5> -->
            </div>
        </form>
    </body>
</html>
```

Vista/listado.jsp

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@page import="java.util.*,modelo.*"  %>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Listado de Usuarios registrados</title>
        <link rel="stylesheet" href="estilos.css" type="text/css">
    </head>
    <body>
        <form action="Controller" method="POST">
            <input type="hidden" value="consultar" name="operacion">
            <div>
            <h3>Librería Web <br>Seleccione un libro.</h3>
            <table border="0">
                <tbody>
                    <tr>
                        <td>Libro:</td>
                        <!-- Código SCRIPTLET
                        <td>
                            <select name="cboLibros">
                                <% ArrayList libros =
                                   (ArrayList)request.getAttribute("libros");
                                    if(libros!=null)
                                        for(int i = 0; i< libros.size();i++){
                                            Libro l = (Libro)libros.get(i);%>
                                     <option><%=l.getNombre()%></option>
                                <%}%>
                            </select>
                        </td> -->

                        <td>
                         <select name="cboLibros">
                          <c:forEach var="libro"
                            items="${requestScope['libros']}">
                             <option><c:out value="${libro.nombre}"/></option>
                           </c:forEach>
                          </select>
                        <td>
                         <input type="submit" value="Consultar"
                                                name="btnConsultar"/>
                        </td>
                    </tr>
                    <tr>
                      <table border="1">
                       <thead style="color:white;">
                        <tr>
                         <th>Nombre</th><th>Autor</th><th>Disponibilidad</th>
                        </tr>
                       </thead>
                         <tr><!--Código JSTL/EL -->
                           <td><c:out value="${libro.nombre}"/></td>
                           <td><c:out value="${libro.autor}"/></td>
                           <td><c:out value="${libro.stock}"/></td>
                         </tr>
                      </table>
                    </tr>
                </tbody>
            </table>
            </div>
        </form>        </body>        </html>
```

Controlador/Controler.java

```
package controlador;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import modelo.BaseDatos;
public class Controller extends HttpServlet {
    private BaseDatos bd;
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                   throws ServletException, IOException {
        String operacion = request.getParameter("operacion");
        RequestDispatcher rq = null;
        //Si la acción es login...
        if (operacion.equals("login")) {
            String usuario = request.getParameter("usuario");
            String clave = request.getParameter("clave");
            if (bd.validarUsuario(usuario, clave)) {
               request.setAttribute("libros", bd.getLibros());
               //encapsulamos el objeto que procesará la salida.
               rq = request.getRequestDispatcher("vista/listado.jsp");
             } else {
               request.setAttribute("msgError", "Usuario o contraseña
                                                              incorrectos");
               rq = request.getRequestDispatcher("vista/login.jsp");
            }
        } else {
            String libroSel = request.getParameter("cboLibros");
            request.setAttribute("libros", bd.getLibros());
            request.setAttribute("libro", bd.consultarLibro(libroSel));
            rq = request.getRequestDispatcher("vista/listado.jsp");
        }
        //delegamos la salida a otro recurso (en nuestro caso a otra JSP)
        rq.forward(request, response);
    }
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
                                                                  response)
                                       throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse
                                                                  response)
                                       throws ServletException, IOException {
        processRequest(request, response);
    }
    @Override
    public String getServletInfo() {
        return "Servlet de Validación de Usuarios";
    }// </editor-fold>
    @Override
    //sobreescribimos el método 'init()' para que cuando se cargue e inicialice
    //nuestro servlet se creen los usuarios de nuestra base de usuarios.
    public void init() throws ServletException {
        super.init();
        bd = new BaseDatos();
    }
}
```

modelo/BasedeDatos.java

```
package modelo;


import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

/**
 *
 * @author fenix
 */
public class BaseDatos {

    private List usuarios;
    private List libros;

    public BaseDatos(){
        //usuarios...
        usuarios = new ArrayList();
        usuarios.add(new Usuario("Nick", "nick"));
        usuarios.add(new Usuario("admin", "123"));
        //libros...
        libros = new ArrayList();
        libros.add(new Libro("interfaces graficas y aplicaciones para internet",
                                                    "J. Ceballos Sierra"));
        libros.add(new Libro("J2EE Aplicaciones Web con Java", "Don Java"));
        libros.add(new Libro("Todo HTML", "W3c"));

    }

    public boolean  validarUsuario(String usr, String pwd){
       if(usr ==null || pwd ==null)
           return false;

       Iterator i = usuarios.iterator();
       while(i.hasNext()){
           Usuario user = (Usuario)i.next();
           if(user.getNombre().equals(usr)&&user.getClave().equals(pwd))
               return true;
       }
       return false;
    }

    public Libro consultarLibro(String n){
       Iterator i = getLibros().iterator();
       while(i.hasNext()){
           Libro l = (Libro)i.next();
           if(l.getNombre().equals(n))
               return l;
       }
       return null;
    }

    public List getLibros() {
        return libros;
    }
}
```

modelo/Libro.java

```
package modelo;
public class Libro {
    private String nombre;
    private String autor;
    private int stock;
    public Libro(String n, String a) {
        nombre = n;
        autor = a;
        stock = (int) (Math.random() * 100);
    }
    public String getNombre() {return nombre;}
    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getAutor() {return autor;}
    public void setAutor(String autor) {this.autor = autor;}
    public int getStock() {return stock;}
    public void setStock(int stock) {this.stock = stock;}
}
```

modelo/Usuario.java

```
package modelo;
public class Usuario {
    private String nombre;
    private String clave;
    private String perfil;
    public Usuario(String n, String c){
        nombre = n;
        clave = c;
        int random = (int)(Math.random()*2);
        if(random>0)
            perfil="ADMINISTRADOR";
        else
            perfil="USUARIO";
    }

    public String getNombre() {return nombre;}
    public void setNombre(String nombre) {this.nombre = nombre;}
    public String getClave() {return clave;}
    public void setClave(String clave) {this.clave = clave;}
    public String getPerfil() {return perfil;}
    public void setPerfil(String perfil) {this.perfil = perfil;}
}
```

Tema Principal: Introducción a la Programación Distribuida.

Temas Particulares: Propuesta de enunciado para resolución por modelos MVC (Jsp/Servlet/Modelo) y Modelo 2.

Caso de estudio: "JProde"

Les proponemos desarrollar una aplicación Java Web que nos permita gestionar la carga de apuestas de un **Prode** con los resultados de los partidos de la primera ronda del torneo regional de fútbol.

La aplicación debe permitir cargar resultados de los partidos, eligiendo de los combos de selección mostrados en la imagen anterior, los equipos (Local y Visitante) y el resultado correspondiente. Una vez ingresados los datos del partido, se deberá validar en el servidor que la combinación de equipos Local-Visitante es correcta según el feacture de los partidos e informar con un mensaje si el resultado fue registrado con éxito o no.

apostarPartido.jsp

La opción >>Cargar otro resultado permite regresar a la pantalla anterior para continuar con la carga de los resultados de los 6 partidos a disputarme en la primera ronda del regional. La opción Ver Aciertos, en cambio deberá mostrar el porcentaje de aciertos obtenido hasta el momento. Es decir si solo se cargaron 3 partidos, podemos ver el % de aciertos sobre 3 apuestas.

registrarApaciertos. jsp

SU TRABAJO.

En su equipo tendrá disponible un proyecto **incompleto**, con los fuentes de la aplicación Web enunciada anteriormente. Ud. deberá completar el código que le proporcionamos de

modo de lograr un funcionamiento correcto de toda la aplicación. Queda bajo su responsabilidad elegir la propuesta de resolución: puede

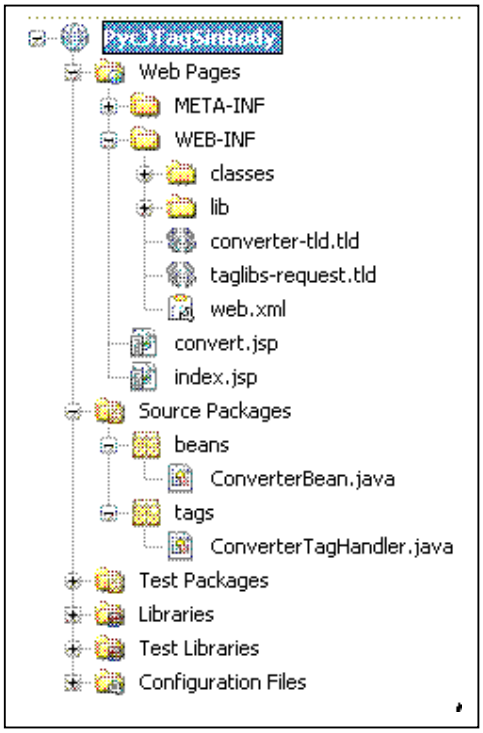
- implementar un JavaBean y comunicación entre páginas .jsp
- utilizar un esquema Model-View-Controller.

Para ello tiene las siguientes tareas:

- Completar las páginas .jsp con la/s líneas necesarias. No deberá armar nada desde cero. Considere utilizar etiquetas correctas y completarlas con los atributos adecuados

- Codificar las clases:
  - **Esquema JAVABEAN/JSPs:** `ProdeBean.java` con los atributos y métodos que considere necesarios, más estos dos métodos obligatorios:
    - **public String getResultadoApuesta(){}.** Permite registrar la apuesta y retornar el mensaje de confirmación de carga o error cuando se envían los datos de los equipos y el resultado apostado al servidor.
    - **public String getPorcentajeAciertos(){}**. Retorna el % de aciertos según los partidos cargados hasta el momento.
  - **Esquema MODEL-VIEW-CONTROLLER:** `ProdeServlet.java` completando el método que proceso la petición y gestiona el flujo entre las páginas.

  - `GestorProde.java` **solo** los métodos:
    - public String cargarResultado(String eLocal, String eVisitante, String resultado){}.
      Busca en su base de partidos, los equipos recibidos por parámetro, y registra el resultado en el partido correspondiente. Si es exitoso el proceso retorna: "Apuesta registrada con éxito", caso contrario: "Revise el feacture, seleccione correctamente los equipos."
    - public float calcularPorcentajeAciertos(){}.
      Retorna el valor del % calculado sobre los aciertos registrados contra los partidos apostados, no sobre los 6 partidos en total. Por ejemplo si solo cargamos 3 apuestas y solo 1 coincide con el resultado simulado por la aplicación, entonces debería retornar 2/3.



Vamos por partes

Convert-tld.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
  <tlib-version>1.0</tlib-version>
  <short-name>converter-tld</short-name>
  <uri>/WEB-INF/converter-tld</uri>
  <tag>
    <name>convertTo</name>
    <tag-class>tags.ConverterTagHandler</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>cant</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>float</type>
    </attribute>
    <attribute>
      <name>coefConv</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
      <type>float</type>
    </attribute>
  </tag>
</taglib>

<%--
    Document   : convert
    Created on : 23-may-2010, 14:38:01
    Author     : fenix
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<%@taglib uri="http://jakarta.apache.org/taglibs/request-1.0" prefix="req"%>
<%@taglib uri="/WEB-INF/converter-tld.tld" prefix="convert"%>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JConvert</title>
        <style>
            body{
                background-color:#CEF6CE; font-family: Arial;  font-size:11px; }
            div{
                position:relative; left:400px; top:50px;  border-style:double;
                height: 200px; width:475px;
            }
        </style>
    </head>
```

```
 <body>
     <div>
       <h1 style="color:teal;position:relative;left:10px;">
            Sitio exclusivo para conversiones de dinero
       </h1>
       <h3 style="color:teal;position:relative;left:10px;">Resultado...</h3>
       <jsp:useBean id="convertBean" class="beans.ConverterBean">
          <jsp:setProperty name="convertBean" property="*"/>
       </jsp:useBean>

       <form action="index.jsp">
          <table border="0" style="position:relative;left:10px;">
            <tbody>
              <tr>
                 <td align="right">Cantidad convertida</td>
                 <td><req:parameter name="cantidad"/>
                       <jsp:getProperty name="convertBean" property="monedaOrigen"/>
                 </td>
              </tr>
              <tr>
                 <td align="right">Cantidad final</td>
                 <td><convert:convertTo cant="${convertBean.cantidad}"
                       coefConv="${convertBean.coefConversion}"/>
                       <jsp:getProperty name="convertBean" property="monedaDestino"/>
                 </td>
              </tr>
            </tbody>
          </table>
          <h4 style="position:relative;left:10px;"> Gracias por usar nuestro sitio.</h4>
       </form>
     </div>
   </body>
</html>
```

beans/ConverterBeans.java

```
package beans;

public class ConverterBean {

    private float matConvert[][];
    private String monedas[];
    private int monedaOri,  monedaDes;
    private int cantidad;

    public ConverterBean() {
        monedas = new String[3]; monedas[0] = "Peso/s"; monedas[1] = "Dolar/es";
        monedas[2] = "Euro/s";

        /**
         * Matriz de coeficientes de conversión.
         * Los valores son utilizados por el javaBean para determinar que
         * factor a utilzar en los cálculos.
         */
        matConvert = new float[3][3];

        //Peso a...
        matConvert[0][0] = 1; matConvert[0][1] = 0.251f;  matConvert[0][2] =
0.209f;
```

```
        //Dolar a...
        matConvert[1][0] = 3.98f; matConvert[1][1] = 1; matConvert[1][2] =
0.83f;

        //Euro a...
        matConvert[2][0] = 4.78f; matConvert[2][1] = 1.2f; matConvert[2][2] = 1;
    }

    public int getMonedaDes() {return monedaDes;    }
    public void setMonedaDes(int monedaDes) { this.monedaDes = monedaDes;    }
    public int getMonedaOri() { return monedaOri;    }
    public void setMonedaOri(int monedaOri) { this.monedaOri = monedaOri;    }
    public float getCoefConversion(){return matConvert[monedaOri][monedaDes];}
    public String getMonedaOrigen(){  return monedas[monedaOri];    }
    public String getMonedaDestino(){ return monedas[monedaDes];    }
    public int getCantidad() { return cantidad;    }
    public void setCantidad(int cantidad) {     this.cantidad = cantidad;    }
}
```

tags/ConverterTaghandler.java

```
package tags;

import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

/**
 *
 * @author fenix
 */
public class ConverterTagHandler extends SimpleTagSupport {

    private float cant;
    private float coefConv;

    /**
     * Called by the container to invoke this tag.
     * The implementation of this method is provided by the tag library
     * developer, and handles all tag processing, body iteration, etc.
     */
    @Override
    public void doTag() throws JspException {
        JspWriter out = getJspContext().getOut();
        try {
            out.print(cant * coefConv);
        } catch (java.io.IOException ex) {
            throw new JspException("Error in ConverterTagHandler tag", ex);
        }
    }

    public void setCant(float cant) {
        this.cant = cant;
    }

    public void setCoefConv(float coefConv) {
        this.coefConv = coefConv;
    }
}
```

`Modelo combinado, proyecto PycProdeRegional2010`

Tema Principal: Introducción a la Programación Distribuida.

Temas Particulares: Propuesta de enunciado para resolución por modelos MVC (Jsp/Servlet/Modelo) y Modelo 2.

Caso de estudio: "JProde"

Les proponemos desarrollar una aplicación Java Web que nos permita gestionar la carga de apuestas de un **Prode** con los resultados de los partidos de la primera ronda del torneo regional de fútbol.

La aplicación debe permitir cargar resultados de los partidos, eligiendo de los combos de selección mostrados en la imagen anterior, los equipos (Local y Visitante) y el resultado correspondiente. Una vez ingresados los datos del partido, se deberá validar en el servidor que la combinación de equipos Local-Visitante es correcta según el feacture de los partidos e informar con un mensaje si el resultado fue registrado con éxito o no.

apostarPartido.jsp

La opción >>Cargar otro resultado permite regresar a la pantalla anterior para continuar con la carga de los resultados de los 6 partidos a disputarme en la primera ronda del regional. La opción Ver Aciertos, en cambio deberá mostrar el porcentaje de aciertos obtenido hasta el momento. Es decir si solo se cargaron 3 partidos, podemos ver el % de aciertos sobre 3 apuestas.

registrarApuesta.jsp

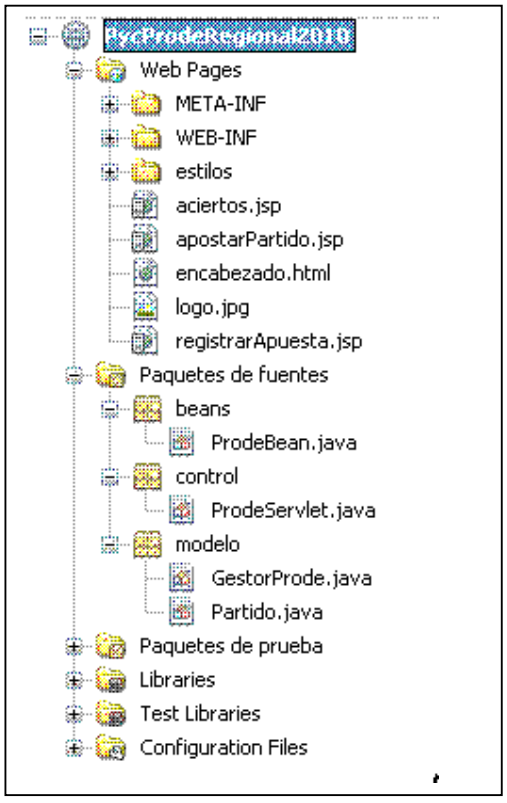
aciertos.jsp

SU TRABAJO.

En su equipo tendrá disponible un proyecto **incompleto**, con los fuentes de la aplicación Web enunciada anteriormente. Ud. deberá completar el código que le proporcionamos de modo de lograr un funcionamiento correcto de toda la aplicación. Queda bajo su responsabilidad elegir la propuesta de resolución: puede implementar un JavaBean y comunicación entre páginas .jsp o bien utilizar un esquema Model-View-Controller.

Para ello tiene las siguientes tareas:

- Completar las páginas .jsp con la/s líneas necesarias. No deberá armar nada desde cero. Considere utilizar etiquetas correctas y completarlas con los atributos adecuados
- Codificar las clases:
  - **Esquema JAVABEAN/JSPs:** `ProdeBean.java con los atributos y métodos que considere necesarios, más estos dos métodos obligatorios:`
    - **public String getResultadoApuesta(){}.** Permite registrar la apuesta y retornar el mensaje de confirmación de carga o error cuando se envían los datos de los equipos y el resultado apostado al servidor.
    - **public String getPorcentajeAciertos(){}**. Retorna el % de aciertos según los partidos cargados hasta el momento.
  - **Esquema MODEL-VIEW-CONTROLLER:** `ProdeServlet.java completando el método que proceso la petición y gestiona el flujo entre las páginas.`
  - `GestorProde.java` **solo** `los métodos:`
    - public String cargarResultado(String eLocal, String eVisitante, String resultado){}. Busca en su base de partidos, los equipos recibidos por parámetro, y registra el resultado en el partido correspondiente. Si es exitoso el proceso retorna: "Apuesta registrada con éxito", caso contrario: "Revise el feacture, seleccione correctamente los equipos."
    - public float calcularPorcentajeAciertos(){}. Retorna el valor del % calculado sobre los aciertos registrados contra los partidos apostados, no sobre los 6 partidos en total. Por ejemplo si solo cargamos 3 apuestas y solo 1 coincide con el resultado simulado por la aplicación, entonces debería retornar 2/3.



Vamos por partes

aciertos.jsp

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%--
    Document   : aciertos
    Created on : 13-jun-2010, 15:32:37
    Author     : fenix
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<link rel="stylesheet" href="estilos/mundial.css" type="text/css">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JProde 2010</title>
    </head>
    <body>
        <%@include file="encabezado.html"%>

        <h3 style="color:#5FB404;"><%=request.getAttribute("msjAciertos")
                                                                    %></h3>
        <p style="color:blue;font-weight: bold;">Gracias por usar JPRODE
                                                        REGIONAL 2010.</p>
    </body>
</html>
```

apostarPartido.jsp

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JProde 2010</title>
        <link rel="stylesheet" href="estilos/mundial.css" type="text/css">
    </head>
    <body>
        <%@include file="encabezado.html"%>
        <table>
            <tr>
                <td>
                    <table border="1">
                        <tbody>
                            <tr class="row">
                                <td class="col">Defensores</td>
                                <td class="col">Flor de Ceibo</td>
                            </tr>
                            <tr class="row">
                                <td class="col">Central</td>
                                <td class="col">Juventud Unida</td>
                            </tr>
                            <tr class="row">
                                <td class="col">Juventud Unida</td>
                                <td class="col">Flor de Ceibo</td>
                            </tr>
                            <tr class="row">
                                <td class="col">Defensores</td>
                                <td class="col">Central</td>
                            </tr>
                            <tr class="row">
                                <td class="col">Flor de Ceibo</td>
                                <td class="col">Central</td>
                            </tr>
```

```
                            <tr class="row">
                                <td class="col">Juventud Unida</td>
                                <td class="col">Defensores</td>
                            </tr>
                        </tbody>
                    </table>
                </td>
                <td>
                    <form action="ProdeServlet" method="post">
                        <input type="hidden" name="accion"
                                                value="cargarApuesta"/>
                        <table border="0">
                            <tbody>
                                <tr>
                                    <td align="right">Equipo local</td>
                                    <td><select name="local">
                                            <option>Defensores</option>
                                            <option>Central</option>
                                            <option>Juventud Unida</option>
                                            <option>Flor de ceibo</option>
                                        </select></td>
                                </tr>
                                <tr>
                                    <td align="right">Equipo visitante</td>
                                    <td><select name="visitante">
                                            <option>Defensores</option>
                                            <option>Central</option>
                                            <option>Juventud Unida</option>
                                            <option>Flor de ceibo</option>
                                        </select></td>
                                </tr>
                                <tr>
                                    <td align="right">Resultado:</td>
                                    <td><select name="resultado">
                                          <option value="L">L-Local</option>
                                          <option value="E">E-Empate</option>
                                          <option value="V">V-Visitante</option>
                                        </select></td>
                                </tr>
                                <tr>
                                    <td colspan="2" align="right">
                                        <input type="submit" value="Cargar
                                                                    Resultado" />
                                    </td>
                                </tr>
                            </tbody>
                        </table>
                    </form>
                </td>
            </tr>
        </table>
        <p style="font-size:11px;font-family:Courier New;"> Selecciona los
             equipos según el feacture y elegí
                <b>L</b> para apostar<br/>por el Local,
                <b>E</b> si pensas en un empate y
                <b>V</b> si tu favorito es el <br/>
                                equipo Visitante
        </p>
    </body>
</html>
```

registrarApuesta.jsp

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JProde 2010</title>
        <link rel="stylesheet" href="estilos/mundial.css" type="text/css">
    </head>
    <body>
        <%@include file="encabezado.html"%>
        <form action="ProdeServlet" method="post">
            <input type="hidden" name="accion" value="verAciertos"/>
            <table border="0">
                <tbody>
                    <tr>
                        <td colspan="2"><%=
                                    request.getAttribute("rtdoApuesta")%></td>
                        <td>>><a href="apostarPartido.jsp">Cargar otro
                                    resultado</a></td>
                    </tr>
                    <tr>
                        <td colspan="2"><input type="submit" value="Ver
                                    Aciertos" /></td>
                    </tr>
                </tbody>
            </table>
        </form>
    </body>
</html>
```

beans/ProdeBean.java

```
package beans;
import modelo.GestorProde;
public class ProdeBean {
    private GestorProde feacture;
    private String local;
    private String visitante;
    private String resultado;
    public ProdeBean() {feacture = new GestorProde();}
    public GestorProde getFeacture() {return feacture;}
    public void setFeacture(GestorProde feacture) {this.feacture = feacture;}
    public String getLocal() {return local;}
    public void setLocal(String local) {this.local = local;}
    public String getVisitante() {return visitante;}
    public void setVisitante(String visitante) {this.visitante = visitante;}
    public String getResultado() {return resultado;}
    public void setResultado(String resultado) {this.resultado = resultado;}
    public String getResultadoApuesta(){
        return feacture.cargarResultado(local, visitante, resultado);
    }
    public String getPorcentajeAciertos(){
        return "% de Aciertos hasta el momento: " +
        feacture.calcularPorcentajeAciertos()*100;
    }
}
```

control/ProdeServlet.java

```
package control;
import java.io.IOException; import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException; import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse; import modelo.GestorProde;

public class ProdeServlet extends HttpServlet {
    private GestorProde gestor;
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
                                  throws ServletException, IOException {
        String accion = request.getParameter("accion");
        String forward = null;
        RequestDispatcher rd = null;
        if (accion.equals("cargarApuesta")) {
            String local = request.getParameter("local");
            String visitante = request.getParameter("visitante");
            String resultado = request.getParameter("resultado");
            request.setAttribute("rtdoApuesta", gestor.cargarResultado(local,
                                                    visitante, resultado));
            forward = "registrarApuesta.jsp";
        } else {
            //acción: "VerAciertos"...
            request.setAttribute("msjAciertos", "% de Aciertos hasta el momento:
                                 " + gestor.calcularPorcentajeAciertos() * 100);
            forward = "aciertos.jsp";
        }
        rd = request.getRequestDispatcher(forward);
        rd.forward(request, response);
    }
    public void init() throws ServletException {
        super.init();
        gestor = new GestorProde();
    }
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {
        processRequest(request, response);
    }
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
                          throws ServletException, IOException {
        processRequest(request, response);
    }
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>
}
```

modelo/gestorProde.java

```
package modelo;
import java.util.LinkedList;
import java.util.List;
public class GestorProde {
    private List<Partido> partidos;
    public GestorProde() {
        partidos = new LinkedList<Partido>();
        partidos.add(new Partido("Defensores", "Flor de Ceibo"));
        partidos.add(new Partido("Central", "Juventud Unida"));
        partidos.add(new Partido("Juventud Unida", "Flor de Ceibo"));
        partidos.add(new Partido("Defensores", "Central"));
        partidos.add(new Partido("Flor de Ceibo", "Central"));
        partidos.add(new Partido("Juventud Unida", "Defensores"));
    }
```

```
    public String cargarResultado(String eLocal, String eVisitante,
                                              String resultado){
        for (Partido p : partidos) {
            if(p.getLocal().equalsIgnoreCase(eLocal) &&
               p.getVisitante().equalsIgnoreCase(eVisitante)){
                  p.setResultApostado(resultado);
                  return "Apuesta registrada con éxito";
            }
        }
        return "Revise el feacture, seleccione correctamente los equipos.";
    }

    public float calcularPorcentajeAciertos(){
        int c=0;
        for (Partido p : partidos) {
            if(p.getResultApostado()!=null &&
               p.getResultApostado().equals(p.getResultSimulado()))
                c++;
        }
        return (((float)c)/6);
    }

    public List<Partido> getPartidos() {return partidos; }
}
```

modelo/Partido.java

```
package modelo;
public class Partido {
    private String local;
    private String visitante;
    private String resultApostado;
    private String resultSimulado;
    public Partido(String eLocal, String eVisitante) {
        this.local = eLocal;
        this.visitante = eVisitante;
        resultApostado = null;
        resultSimulado = iniciarResultadoPartido();
    }
    public String getLocal() {return local;}
    public void setLocal(String local) {this.local = local;}
    public String getVisitante() {return visitante;}
    public void setVisitante(String visitante) {this.visitante = visitante;}
    public void setResultApostado(String resultApostado) {
        this.resultApostado = resultApostado;}
    public String getResultSimulado() {return resultSimulado;}
    public void setResultSimulado(String resultSimulado) {
        this.resultSimulado = resultSimulado;}
    public String getResultApostado() {return resultApostado;}
    private String iniciarResultadoPartido() {
        int valor = ((int) (Math.random() * 10));
        String r = null;
        if (valor <= 3) {r = "L";
        } else if (valor <= 6) { r = "E";
        } else {r = "V";}
        return r;
    }
    public String toString() {
        return local + " - " + visitante + " Apuesta: " + resultApostado +
               "Resultado: " + resultSimulado;
    }
}
```

Modelo 2 JSP + JavaBeans, verlo en http://labsys.frc.utn.edu.ar, Sitios de las catedras, Paradigmas de programación 2011, Unidad III - Programación distribuida, Codigo Suficiente!!!