REALIZACION DE UNA GUIA METODOLOGICA PARA EL DESARROLLO WEB CON JAVASERVER PAGES Y JAVASERVER FACES

REALIZACION DE UNA GUIA METODOLOGICA PARA EL DESARROLLO WEB CON JAVASERVER PAGES Y JAVASERVER FACES

RICARDO ARTURO GUACARI VILLALBA
CODIGO: 0205901
MARIO ANDRES VERGARA DOMINGUEZ
CODIGO: 9905505

UNIVERSIDAD TECNOLOGICA DE BOLIVAR
FACULTAD DE INGENIERIA DE SISTEMAS
CARTAGENA DE INDIAS D. T. y C.
2005

REALIZACION DE UNA GUIA METODOLOGICA PARA EL DESARROLLO WEB CON JAVASERVER PAGES Y JAVASERVER FACES

RICARDO ARTURO GUACARI VILLALBA
MARIO ANDRES VERGARA DOMINGUEZ

ASESOR:

GIOVANNY RAFAEL VASQUEZ MENDOZA
Ingeniero de Sistemas

UNIVERSIDAD TECNOLOGICA DE BOLIVAR
FACULTAD DE INGENIERIA DE SISTEMAS
CARTAGENA DE INDIAS D. T. y C.
2005

Cartagena, enero 24 del 2006.

Señores:

UNIVERSIDAD TECNOLOGICA DE BOLIVAR.
Comité de evaluación de proyectos.
LC.

Respetados Señores:

De la manera mas cordial, nos permitimos presentar a ustedes para su estudio, consideración y aprobación el Trabajo de Final titulado "**GUIA METODOLOGICA PARA EL DESARROLLO DE PAGINAS WEB CON JAVASERVER PAGES Y JAVASERVER FACES**", Trabajo Final presentado para aprobar el Minor de Ingeniería de Software.

Esperamos que este trabajo sea de su total agrado.
Cordialmente,

______________________________	______________________________
Ricardo Arturo Guacarí Villalba.	Mario Andrés Vergara Domínguez.
Código: 0205901.	Código: 9905505.

Cartagena, enero 24 del 2006.

Señores:

UNIVERSIDAD TECNOLOGICA DE BOLIVAR.
Comité de evaluación de proyectos.
LC.

Respetados Señores:

Tengo el agrado de presentar a su consideración, estudio y aprobación, la monografía titulada "**GUIA METODOLOGICA PARA EL DESARROLLO DE PAGINAS WEB CON JAVASERVER PAGES Y JAVASERVER FACES**", desarrollado por los estudiantes Ricardo A. Guacarí Villalba y Mario A. Vergara Domínguez.

Al respecto me permito comunicar que he dirigido el citado trabajo, el cual considero de gran importancia y utilidad.

Atentamente,

____

Giovanny R. Vásquez Mendoza.
Ingeniero de Sistemas.

Nota de aceptación.

____

____

____

____

Presidente del jurado

____

Jurado

____

Jurado

AGRADECIMIENTOS

nuestros más sinceros agradecimientos son primero que todo a Dios, padre celestial que bendice, ilumina e inspira nuestras vidas, a nuestros padres por brindarnos apoyo incondicional, a nuestros seres queridos que aunque lejos siempre se preocupan por nuestro futuro.

A nuestro asesor Giovanny Vásquez que con su sabiduría y consejos, nos guío durante la realización de nuestro proyecto.

A todas esas personas que de una u otra forma ayudaron en cada momento para así culminar con éxito este trabajo monográfico.

OBJETIVOS

Objetivo General:

Presentar a la comunidad académica de la UTB un documento que describa los conceptos y características importantes de las tecnologías JSP y JSF para facilitar la elaboración de paginas Web y que a su vez sirva de guía para los interesados en hacer uso de estas especificaciones

Objetivos Específicos:

- Ofrecer una visión global e integrada de las especificaciones JSP y JSF como nuevas alternativas para el desarrollo de aplicaciones Web.

- Señalar las características importantes de las especificaciones de JSP y JSF.

- Describir las principales etiquetas que posee JavaServer Faces.

- Elaborar un pequeño prototipo en donde se haga uso de las especificaciones JSP y JSF.

TABLA DE CONTENIDO

1. INTRODUCCIÓN 12
2. Marco teórico 13
    2.1. Java 13
    2.2. HTML 14
    2.3. JDBC 15
    2.4. APACHE 15
    2.5. TOMCAT 16
3. DISEÑO METODOLOGICO 17
4. INTRODUCCIÓN A JSP 18
    4.1. ¿Por qué utilizar JSP? 19
    4.2. ¿Qué se necesita para trabajar con JSP? 20
    4.3. Creación y compilación del Servlet 20
    4.4. ¿Cómo se usan las páginas JSP? 21
    4.5. Sumario de Sintaxis 22
    4.6. Plantilla de Texto: HTML estático 25
    4.7. Elementos de Script JSP 25
    4.8. Expresiones JSP 26
    4.9. Scriptlets JSP 27
    4.10. Declaraciones JSP 29
    4.11. Directivas JSP 30
        4.11.1. La directiva page 31
            4.11.1.1. Sintaxis de la Directiva page 32
    4.12. La directiva include JSP 35
        4.12.1. Clases de ficheros que se pueden incluir: 35
        4.12.2. Incluir Ficheros JSP 36
        4.12.3. Incluir Ficheros Estáticos 36
    4.13. Variables Predefinidas 37
        4.13.1. request 38

4.13.1.1. Uso de los Métodos GET y POST 39
4.13.2. response 41
4.13.3. out 41
4.13.4. session 42
4.13.5. application 42
4.13.6. config 42
4.13.7. pageContext 42
4.13.8. page 43
4.14. Acciones Estándar JSP 43
4.14.1. Acción jsp:include 43
4.14. 2. Acción jsp:useBean 44
4.14. 2.1. ¿Cómo funciona el jsp:useBean? 46
4.14. 3. Acción jsp:setProperty 47
4.14. 4. Acción jsp:getProperty 51
4.14.5. Acción jsp:forward 51
4.14.6. Acción jsp:plugin 53
4.15. El Juego de la Adivinanza Numérica 54
4.16. Acceso a Bases de Datos 57
4.16.1. Configurando ODBC para Access 58
4.16.2. Estableciendo la conexión 59
5.Introducción a JAVASERVER FACES 64
5.1. ¿Por qué utilizar JSF? 66
5.2. ¿Qué se necesita para trabajar con JSF? 68
5.3. ¿Qué es una Aplicación JavaServer Faces? 68
5.4. El Ciclo de Vida de una Página JavaServer Faces 71
5.4.1. Escenarios de Procesamiento del Ciclo de Vida de una Petición 71
5.4.2. Ciclo de Vida Estándar de Procesamiento de Peticiones 73
5.4.2.1. Reconstituir el Árbol de Componentes 74
5.4.2.2. Aplicar Valores de la Petición 74
5.4.2.3. Procesar Validaciones 75

5.4.2.4. Actualizar los Valores del Modelo 75
5.4.2.5. Invocar Aplicación 76
5.4.2.6. Renderizar la Respuesta 77
5.5. usando las etiquetas core 77
5.6. usando el componente de etiquetas html 85
5.6.1. atributos de los componentes de etiquetas UI 85
5.6.2. Modelo de Componentes de Interface de Usuario 87
5.6.2.1. Las Clases de los Componentes del Interface de Usuario 88
5.6.2.2. El Modelo de Conversión (Renderizado) de Componentes 90
5.6.2.3. Modelo de Eventos y Oyentes 99
5.6.2.4. Modelo de Validación 100
5.7. Modelo de Navegación 101
5.8. Formato Básico para Trabajar con JavaServer Faces 102
5.9 Ejemplo JavaServer Faces 103
6. CONCLUSION 106
7. RECOMENDACIONES 107
8. GLOSARIO 108
8. BIBLIOGRAFIA 111

1. INTRODUCCIÓN

La presente guía se enmarca en los avances tecnológicos que existen hoy día y que están siendo utilizados por gran parte de los desarrolladores de aplicaciones Web a nivel mundial como lo con JSP y JSF. Los cuales permiten que las entidades implementen sus recursos de funcionamiento y desarrollo, manipulando sistemas capaces de hacer que la optimización a nivel global en una entidad sea más eficiente.

Desde hace unos diez años hasta la actualidad, el fenómeno de la información digital el cual es llamado Internet, ha venido creciendo a pasos agigantados, ofreciendo cada vez mas comodidad para los usuarios pero a la vez exigiendo calidad. Al referirnos a calidad, se da a entender que los desarrolladores de paginas Web de hoy en día buscan la perfección en ellas.

Bien sabemos que existen las paginas Web estáticas y las paginas Web dinámicas, y que estas segundas están imponiéndose ante las primeras por la simple razón de que por medio de estas se puede ingresar información y por otras características que son incrustadas en el código fuente de las mismas.

Por lo anterior suscitado, para la puesta en marcha de este sistema se tomaron en cuenta los últimos avances tecnológicos de la ingeniería de software, poniendo en practica técnicas que nos facilitaron el aprovechamiento de conocimientos y por supuesto nos ayudaron a implementar y dar soluciones a la entidad permitiéndole ser mas competitiva en el mercado.

2. MARCO TEORICO

Cuando hablamos de desarrollo de paginas Web, en realidad no solo nos referimos al estar viviendo en un mundo moderno y novedoso, sino que también ocupamos un mundo mucho más tecnológico, puesto que con el rápido desarrollo de Internet y el surgimiento del comercio electrónico se ha obligado a la continuación de la búsqueda de la perfección. Y es aquí en donde se han dado los nacimientos de nuevas metodologías y herramientas que proporcionan medios para la consecución de soluciones a los problemas. Tal es el caso de Sun Microsystems que ha proporcionado al desarrollo de software nuevas facetas, entre ellas esta J2EE (Java 2 Enterprise Edition) que abarca el campo de desarrollo de nuestra monografía.

2.1. JAVA

Es un lenguaje de programación que fue creado para el desarrollo de software avanzados, para una amplia variedad de dispositivos de red y sistemas distribuidos, además es actualmente una de las tecnologías informáticas con mayor difusión dentro de este sector, su propietaria ***Sun Microsystems***, la creo a mediados de la década de los 90, para un uso en dispositivos electrónicos de consumo. Sin embargo, pronto se entendió que sus características se adaptaban perfectamente a las condiciones de Internet, y fue aquí, donde finalmente fue aplicado. En los pocos años que lleva de vida a sufrido un gran desarrollo y aceptación. Se caracteriza por ser un lenguaje robusto, un lenguaje de propósito general, de alto nivel, simple, orientado a objetos de arquitectura neutral, seguro, multihilo y lo mas importante es que para hacer uso de este

lenguaje no se necesita de ningún tipo de licencia, debido a que este Software es gratuito.

2.2. HTML

(*hyper text markup language***)**, lenguaje para marcado de hipertexto. Es un lenguaje muy sencillo que permite describir hipertexto, es decir, texto presentado en forma estructurada y agradable, con enlaces (hyperlinks) que conducen a otros documentos o fuentes de información relacionadas, y con inserciones multimedia (gráficos, sonidos). La descripción se basa en especificar en el texto la estructura lógica del contenido, así como los diferentes efectos que se quieran y dejar que luego la presentación final de dichos hipertextos se realice por un programa especializado.

HTML es una herramienta de fácil manejo que cumple con las siguientes características básicas:

- Permite introducir enlaces.
- Seleccionar tamaños o intercalar imágenes.
- Crear paginas a partir de una base de datos.

Vale la pena anotar que a pesar de estas características HTML no deja de ser una herramienta limitada a la hora de concebir paginas versátiles y extensas; Pero ofrece la creación de esta clase de paginas con una combinación de Scripts.

2.3. JDBC

La conectividad de base de datos Java (Java DataBase Conectivity, JDBC) es una especificación de la interfaz de aplicación de programa

(Application Program Interface, API) para conectar los programas escritos en Java a los datos, almacenados en bases de datos populares. La interfaz de aplicación nos permite codificar ordenes de solicitud de acceso en lenguaje estructurado de solicitud (SQL) que luego pasan al programa que administra la base de datos. La JDBC es muy similar a la conectividad abierta a base de datos (ODBC), con un pequeño programa "puente" podemos usar la interfaz JDBC para acceder a bases de datos a través de la interfaz ODBC.

2.4. APACHE

Es un software que provee servicios de comunicación con el protocolo http. Por tal razón es un software considerado servidor. Nació como situación para el servidor httpd 1.3 desarrollado por la NCSA.
Según lo expuesto por sus creadores, apache es uno de los mejores servidores de Web utilizados en la Red Internet desde hace mucho tiempo únicamente le hace competencia el servidor de Microsoft llamado, el IIS. Por lo que este servidor es uno de los mayores triunfos del software libre, que tanto gusta a los usuarios de **LINUX.** Es un servidor de Web flexible, rápido y eficiente, continuamente actualizado y adaptado a los nuevos protocolos http.

2.5. TOMCAT

Es un contenedor de JSP y Servlet de libre distribución y código abierto usado como referencia tanto en JSP como con Servlet. Incluye numerosas particularidades que hacen que sea una de las plataformas favoritas para el desarrollo de aplicaciones y aplicaciones Web, Tomcat puede ser usado a modo de servidor Web merced a un Servlet dedicado a servicio de fichero. Tomcat es la implementación de referencia para las JavaServer Pages (JSP) y la especificaciones Java Servlet. Esto significa que es el servidor Java disponible que más se ajusta a los estándares. Tomcat puede utilizarse como un contenedor solitario (principalmente para desarrollo y depuración) o como plugin para un servidor Web existente.

3. DISEÑO METODOLÓGICO

Para alcanzar el objetivo propuesto se procederá de la siguiente manera:

1. Hacer una investigación bibliográfica sobre el tema. Esta fase implica consultar las diferentes fuentes de información sobre el tema.

2. Se leerán las fuentes bibliográficas y se clasificará la información relevante para nuestro trabajo, ya que es posible encontrar mucha información sobre el tema que no sea relevante para nuestro trabajo.

3. Se elaborará una síntesis sobre los temas que se incluirán en la guía. Esta actividad es la más importante porque en ella vamos a realizar la verdadera investigación a través de la lectura. Todo esto servirá para realizar la síntesis del tema que quedará evidenciada en el documento guía.

4. Se elaborará el documento guía. Esta actividad y la anterior pueden realizarse paralelamente. En esta fase del proyecto, estaremos escribiendo nuestra síntesis, de los temas investigados, en el documento final del proyecto.

4. INTRODUCCIÓN A JAVASERVER PAGES

Los archivos JSP son archivos HTML con etiquetas especiales que contienen código Java con los cuales se hace la parte dinámica de la pagina.

JSP está basado en la tecnología de **servlets**. Cuando es combinado con el uso de componentes JavaBeans, JSP proporcionar una capacidad que es al menos tan poderosa como los Servlets, posiblemente más que un servlet en crudo, y potencialmente mucho más fácil de usar.

Muchas páginas Web que están construidas con programas CGI son casi estáticas, puesto que CGI solo es un software que facilita la comunicación entre un servidor Web y los programas que funcionan fuera del servidor, limita la parte dinámica a muy pocas localizaciones. Pero muchas variaciones CGI, incluyendo los servlets, hacen que generemos la página completa mediante nuestro programa, incluso aunque la mayoría de ella sea siempre lo mismo. JSP nos permite crear dos partes de forma separada. Con lo cual podemos tener una persona que diseñe páginas HTML y otra que inserte código Java para generar la parte dinámica de la aplicación.

4.1. ¿Por qué utilizar JSP?

JSP es fácil de aprender y permite a los desarrolladores de páginas Web crear aplicaciones de una forma estándar. JSP está basado en Java (lenguaje orientado a objetos).

La principal razón para utilizar JSP es:
Es una tecnología multiplataforma (puede ejecutarse en servidores Web con diferentes sistemas operativos: Unix, Windows...) con capacidad de reutilizar componentes mediante JavaBeans y EnterpriseJavaBeans (EJB).

4.2. ¿Qué se necesita para trabajar con JSP?

Lo que se necesita para comenzar a desarrollar paginas Web con JSP son:

- Java Development Kit (JDK)
- Servidor Web con soporte para JSP (Apache TomCat, Blazix 1.1)
- Un navegador Web (Internet Explorer, Mozilla Firefox, Netscape Navigator, Opera...)
- Editor de textos (Notepad por ejemplo).

4.3. Creación y compilación del Servlet

La extensión del los archivos JSP es .jsp en vez de .html o .htm. Cuando desde un navegador Web se pide una página JSP, el servidor lo transforma a código fuente de Servlet y luego lo compila generando el archivo .class. La creación y compilación automática del servlet ocurre la primera vez que se accede a la página. Dependiendo del comportamiento del servidor web, el servlet será grabado durante algún periodo de tiempo para utilizarlo una y otra vez sin necesidad de recrearlo y recompilarlo. Una vez compilado las siguientes peticiones directamente ejecutan el .class del Servlet devolviendo el resultado en formato HTML. Por eso, la primera vez que se accede a la página, podría haber una pausa mientras que el servidor web crea y compila el servlet. Después de esto, los accesos a la página serán muchos más rápidos.

4.4. ¿Cómo se usan las páginas JSP?

El uso de JSP hace posible combinar las mejores capacidades del HTML con los componentes de software reutilizables para crear aplicaciones del lado del servidor.

Esto hace muy práctico separar la lógica del negocio de la representación de los datos. Con esto, los programadores especializados en escribir JavaBeans que implementen la lógica del negocio, y los diseñadores de páginas especializados en HTML pueden realizar llamadas a JavaBeans desde el HTML sin necesidad de convertirse en expertos programadores Java.

JavaServer pages son, en el sentido más básico, páginas Web con código java embebido. El código java embebido es ejecutado en el servidor antes que la página web sea retornada al navegador.
Veamos, enseguida, el ejemplo más sencillo y clásico de una página JSP:

```
<html>
<body>
<%
   out.println("<h1>Hello World!</h1>");
%>
</body>
</html>
```

Como puede verse, la página está compuesta de etiquetas HTML estándar con un poco de código java contenido entre las etiquetas <% y %>.

Al bloque de código encerrado entre las etiquetas <% y %> se le llama un scriptlet.

4.5. Sumario de Sintaxis

En esta sesión, presentaremos la sintaxis general de etiquetas JSP y, posteriormente, presentaremos ejemplos que ilustran la sintaxis que enseguida se explica.

- **Expresión JSP:** La Expresión es evaluada y situada en la salida.

```
<%= expression %>;
```

- **Scriptlet JSP:** El código se inserta en el método service.

```
<% code %>;
```

- **Declaración JSP:** El código se inserta en el cuerpo de la clase del servlet, fuera del método service.

```
<%! Code %>
```

- **Directiva page JSP:** Dirige al motor servlet sobre la configuración general.

```
<%@ page att="val" %>
```

Los atributos legales son (con los valores por defecto en negrita):

- import="**package.class**"
- contentType="**MIME-Type**"
- isThreadSafe="**true**|false"
- session="**true**|false"
- buffer="**size**kb|none"
- autoflush="**true**|false"
- extends="**package.class**"
- info="**message**"
- errorPage="**url**"

    - isErrorPage="true|**false**"
    - language="java"

- **Directiva include JSP:** Un fichero del sistema local se incluirá cuando la página se traduzca a un Servlet.

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

  La URL debe ser relativa. Usamos la acción jsp:include para incluir un fichero en el momento de la petición en vez del momento de la traducción.

- **Comentario JSP:** Comentario ignorado cuando se traduce la página JSP en un servlet.

  ```
  <%-- comment --%>
  ```

  Si queremos un comentario en el HTML resultante, usamos la síntaxis de comentario normal del HTML <-- comment -->.

- **Acción jsp:incluye:** Incluye un fichero en el momento en que la página es solicitada.

  ```
  <jsp:include page="relative URL" flush="true"/>
  ```

  en algunos servidores, el fichero incluido debe ser un fichero HTML o JSP, según determine el servidor (normalmente basado en la extensión del fichero).

- **Acción jsp:useBean:** Encuentra o construye un Java Bean.

  <jsp:useBean id=**“name”** cope=**“page” | ”request” | ”session” | ”application”** Detalles del Bean **/>**

  Detalles del Bean:

class="Nombre de la Clase"
class="Nombre de la Clase" type="Nombre de Tipo" beanName="Nombre del Bean" type="Nombre de Tipo"
type=**"Nombre de Tipo"**

- **Acción jsp:setProperty:** Selecciona las propiedades del bean, bien directamente o designando el valor que viene desde un parámetro de la petición.

<jsp:setProperty att=val*/>

Los atributos legales son:

- name="**beanName**"
- property="**propertyName**|*"
- param="**parameterName**"
- value="**val**"

- **Acción jsp:getProperty:** Recupera y saca las propiedades del Bean.

<jsp:getProperty name="propertyName" value="val"/>

- **Acción jsp:forward:** Reenvía la petición a otra página.

<jsp:forward page="relative URL"/>

- **Acción jsp:plugin:** Genera etiquetas OBJECT o EMBED, apropiadas al tipo de navegador, pidiendo que se ejecute un applet usando el Java Plugin.

<jsp:plugin attribute="value"*> ... </jsp:plugin>

4.6. Plantilla de Texto: HTML estático

En muchos casos, un gran porcentaje de nuestras páginas JSP consistirá en HTML estático, conocido como **plantilla de texto**. En casi todos lo aspectos, este HTML se parece al HTML normal, sigue la mismas reglas de sintaxis, y simplemente "pasa a través" del cliente por el servlet creado para manejar la página. No sólo el **aspecto** del HTML es normal, puede ser **creado** con cualquier herramienta que usemos para generar páginas Web.

La única excepción a la regla de que "la plantilla de texto se pasa tal y como es" es que, si queremos tener "<%" en la salida, necesitamos poner "<\%" en la plantilla de texto.

4.7. Elementos de Script JSP

Los elementos de script nos permiten insertar código Java dentro del servlet que se generará desde la página JSP actual. Hay tres formas:

- Expresiones de la forma <%= expresión %> que son evaluadas e insertadas en la salida.

- Scriptlets de la forma <% código %> que se insertan dentro del método service del servlet.

- Declaraciones de la forma <%! código %> que se insertan en el cuerpo de la clase del servlet, fuera de cualquier método existente.

4.8. Expresiones JSP

Una expresión JSP se usa para insertar valores Java directamente en la salida. Tiene la siguiente forma:

```
<%= expresión Java %>
```

La expresión Java es evaluada, convertida a un string, e insertada en la página. Esta evaluación se realiza durante la ejecución (cuando se solicita la página) y así tiene total acceso a la información sobre la solicitud. Por ejemplo, esto muestra la fecha y hora en que se solicitó la página:

```
<%= Math.sqrt(2) %>
<%= items[i] %>
<%= a + b + c %>
<%= new java.util.Date() %>
```

Para simplificar estas expresiones, hay un gran número de variables predefinidas[1] que podemos usar. Estos objetos implícitos se describen más adelante con más detalle, pero para el propósito de las expresiones, los más importantes son:

- **request**, el HttpServletRequest;
- **response**, el HttpServletResponse;
- **session**, el HttpSession asociado con el request (si existe), y
- **out**, el PrintWriter (una versión con buffer del tipo JspWriter) usada para envíar la salida al cliente.

Ejemplo:

```
Your hostname: <%= request.getRemoteHost() %>
```

Este fragmento de código devuelve el nombre de nuestra máquina.

[1] Variables predefinidas (página 34, sección 4.13)

Ejemplo de uso de Expresiones

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

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

4.9. Scriptlets JSP

Si queremos hacer algo más complejo que insertar una simple expresión, los scriptlets JSP nos permiten insertar código arbitrario dentro del método servlet que será construido al generar la página. Los Scriptlets tienen la siguiente forma:

```
<% Código Java  %>
```

Los Scriptlets tienen acceso a las mismas variables predefinidas que las expresiones. Por eso, por ejemplo, si queremos que la salida aparezca en la página resultante, tenemos que usar la variable out:

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

Observa que el código dentro de un scriptlet se insertará **exactamente** como está escrito, y cualquier HTML estático (plantilla de texto) anterior o posterior al scriptlet se convierte en sentencias print. Esto significa que los scriptlets no necesitan

completar las sentencias Java, y los bloques abiertos pueden afectar al HTML estático fuera de los scriplets. Por ejemplo, el siguiente fragmento JSP, contiene una mezcla de texto y scritplets:

```
<% if (Math.random() < 0.5) { %>
Have a <B>nice</B> day!
<% } else { %>
Have a <B>lousy</B> day!
<% } %>
```

que se convertirá en algo como esto:

```
if (Math.random() < 0.5) {
  out.println("Have a <B>nice</B> day!");
} else {
  out.println("Have a <B>lousy</B> day!");
}
```

Si queremos usar los caracteres "%>" dentro de un scriptlet, debemos poner "%\>".

Ejemplo de uso de Scriplets

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

```
<HTML>
	<head>
		<title> Página de ejemplo de scriptlet </title>
	</head>
	<body>
		<h1> Página de ejemplo de scriptlet </h1>
		<%
			for (int i=0; i<10; i++;)
			{		out.println("<b> Hola a todos. Esto es un
```

```
                    ejemplo de scriptlet " + i + "</b><br>");
                    System.out.println("Esto va al stream
                    System.out" + i );
                //Esto último va a la consola del Java, no al cliente.
                //out a secas es para la respuesta al cliente.
                }
            %>
    </body>
</HTML>
```

4.10. Declaraciones JSP

Una **declaración** JSP nos permite definir métodos o campos que serán insertados dentro del cuerpo principal de la clase servlet (fuera del método service que procesa la petición). Tienen la siguiente forma:

```
<%! Código Java%>
```

Como las declaraciones no generan ninguna salida, normalmente se usan en conjunción con expresiones JSP o scriptlets. Por ejemplo, aquí tenemos un fragmento de JSP que imprime el número de veces que se ha solicitado la página actual desde que el servidor se arrancó (o la clase del servlet se modificó o se recargó):

Ejemplo:

```
<%! int i = 0; %>
<%! int a, b; double c; %>
<%! Circle a = new Circle(2.0); %>
<%! private int accessCount = 0; %>
<%= ++accessCount %>
```

Ejemplo de uso

```
<HTML>
```

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

Como con los scriptlet, si queremos usar los caracteres "%>", ponemos "%\>".

4.11. Directivas JSP

Una **directiva** JSP afecta a la estructura general de la clase servlet. Normalmente tienen la siguiente forma:

```
<%@ directive attribute="value" %>
```

Sin embargo, también podemos combinar múltiples selecciones de atributos para una sola directiva, de esta forma:

```
<%@ directive attribute1="value1"
        attribute2="value2"
        ...
        attributeN="valueN" %>
```

Hay dos tipos principales de directivas: page, que nos permite hacer cosas como importar clases, personalizar la superclase del servlet, etc. e include, que nos permite insertar un fichero dentro de la clase servlet en el momento que el fichero JSP es traducido a un servlet. La especificación también menciona la directiva taglib, que no está soportada en JSP 1.0, pero se pretende que permita que los autores de JSP definan sus propias etiquetas. Se espera que sea una de las principales contribuciones a JSP 1.1.

4.11.1 La directiva page

La directiva **Page** se usa para definir atributos que se aplican a una página JSP entera.

La directiva **page** se aplica a una página JSP completa, y a cualquier fichero estático que incluya con la directivas **include**" o **<jsp:include>**, que juntas son llamadas una unidad de traducción.

Una directiva **page** puede usarse para establecer valores para distintos atributos que se pueden aplicar a la página JSP. Podemos usar la directiva **page** más de una vez en una página JSP (unidad de traducción). Sin embargo, (excepto para el atributo **import**), sólo podemos especificar un valor para atributo una sóla vez. El atributo **import** es similar a la directiva **import** en un programa Java, y por eso podemos usarlas más de una vez.

Podemos situar el directiva **page** en cualquier lugar de la unidad de traducción y se aplicará a toda la unidad de traducción. Por claridad y facilidad de entendimiento, el mejor lugar podría ser al principio del fichero JSP principal.

4.11.1.1. Sintaxis de la Directiva page

Aquí podemos ver la sintaxis de la directiva **page**. Los valores por defecto se muestran en **negrita**. Los corchetes ([...]) indican un término opcional. La barra vertical (|) proporciona una elección entre dos valores como true y false.

<%@ page
[language="**java**"]
[extends="*package.class*"]
[import= "{ *package.class*|package.*}, ..."]
[session="**true**|false"]
[buffer="none|**8kb**|*size*kb"]
[autoFlush="**true**|false"]
[isThreadSafe="**true**|false"]
[info="*text*"]
[errorPage="*URLrelativa*"]
[contentType="*mimeType*[;charset=*characterSet*]" |
"text/html; charset=ISO-8859-1"]
[isErrorPage="true|**false**"]
%>

Ejemplos de Directivas Page

```
<%@ page import="java.util.Date" %>
<%@ page import="java.awt.*" %>
<%@ page info="Información sobre la página" %>
```

La directiva page nos permite definir uno o más de los siguientes atributos sensibles a las mayúsculas:

- *import*="**package.class**" o import="**package.class1,...,package.classN**". Esto nos permite especificar los paquetes que deberían ser importados. Por ejemplo:

  <%@ page import="java.util.*" %> El atributo import es el único que puede aparecer múltiples veces.

- *contentType*="**MIME-Type**" o contentType="**MIME-Type**; charset=**Character-Set**" Esto especifica el tipo MIME de la salida. El valor por defecto es text/html. Por ejemplo, la directiva: <%@ page contentType="text/plain" %> tiene el mismo valor que el scriptlet <% response.setContentType("text/plain"); %>
- *isThreadSafe*="true|false". Este atributo especifica si la seguridad de threads está implementada en el fichero JSP. El valor por defecto, **true**, significa que el motor puede enviar múltiples solicitudes concurrentes a la página.

  Si usamos el valor por defecto, varios threads pueden acceder a la página JSP. Por lo tanto, debemos sincronizar nuestros métodos para proporcionar seguridad de threads.

  Con **false**, el motor JSP no envía solicitudes concurrentes a la página JSP. Probablemente no querremos forzar esta restricción en servidores de gran volumen porque puede dañar la habilidad del servidor de enviar nuestra página JSP a múltiples clientes.

- *session*="true|false". Un valor de true (por defecto) indica que la variable predefinida session (del tipo HttpSession) debería unirse a la sesión existente si existe una, si no existe se debería crear una nueva sesión para

unirla. Un valor de false indica que no se usarán sesiones, y los intentos de acceder a la variable session resultarán en errores en el momento en que la página JSP sea traducida a un servlet.

- *buffer*="**size**kb|none". Este atributo especifica el tamaño del buffer en kilobytes que será usado por el objeto **out** para manejar la salida enviada desde la página JSP compilada hasta el navegador cliente. El valor por defecto es 8kb.

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

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

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

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

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

- *language*="java". En algunos momentos, esto está pensado para especificar el lenguaje a utilizar. Por ahora, no debemos preocuparnos por él ya que java es tanto el valor por defecto como la única opción legal.

4.12. La directiva include JSP

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

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

La URL especificada normalmente se interpreta como relativa a la página JSP a la que se refiere, pero, al igual que las URLs relativas en general, podemos decirle al sistema que interpreta la URL relativa al directorio home del servidor Web empezando la URL con una barra invertida. Los contenidos del fichero incluido son analizados como texto normal JSP, y así pueden incluir HTML estático, elementos de script, directivas y acciones.

Por ejemplo, muchas sites incluyen una pequeña barra de navegación en cada página. Debido a los problemas con los marcos HTML, esto normalmente se implementa mediante una pequeña tabla que cruza la parte superior de la página o el lado izquierdo, con el HTML repetido para cada página de la site. La directiva *include* es una forma natural de hacer esto, ahorrando a los desarrolladores el mantenimiento engorroso de copiar realmente el HTML en cada fichero separado.

4.12.1. Clases de ficheros que se pueden incluir:

El fichero incluido puede ser un fichero JSP, un fichero HTML, o un fichero de texto. También ser un fichero de código escrito en lenguaje Java.

Hay que ser cuidadoso en que el fichero incluido no contenga las etiquetas <html>, </html>, <body>, or </body>. Porque como todo el contenido del fichero

incluido se añade en esa localización del fichero JSP, estas etiquetas podrían entrar en conflicto con las etiquetas similares del fichero JSP.

4.12.2. Incluir Ficheros JSP

Si el fichero incluido es un fichero JSP, las etiquetas JSP son analizadas y sus resultados se incluyen (junto con cualquier otro texto) en el fichero JSP.

Sólo podemos incluir ficheros estáticos. Esto significa que el resultado analizado del fichero incluido se añade al fichero JSP justo donde está situada la directiva. Una vez que el fichero incluido es analizado y añadido, el proceso continúa con la siguiente línea del fichero JSP llamante.

4.12.3. Incluir Ficheros Estáticos

Un **include** estático significa que el texto del fichero incluido se añade al fichero JSP.

Además en conjunción con otra etiqueta JSP, <jsp:include>: podemos incluir ficheros estáticos o dinámicos:

Un fichero estático es analizado y su contenido se incluye en la página JSP llamante. Un fichero dinámico actúa sobre la solicitud y envía de vuelta un resultado que es incluido en la página JSP.

Ejemplo de uso de la Directiva Include:
archivo jsp: incluye el contenido de dos archivos (una pagina HTML y una pagina JSP).
<HTML>

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

archivo.html

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

archivo.jsp

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

4.13. Variables Predefinidas

Para simplificar el código en expresiones y scriplets JSP, tenemos ocho variables definidas automáticamente, algunas veces llamadas objetos implícitos. Las variables disponibles son: request, response, out, session, application, config, pageContext, y page.

4.13.1. request

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

Podríamos encontrar útiles algunos de estos métodos para trabajar con objetos **request**:

Método	**Definido en**	**Función**
GetRequest	*javax.servlet.jsp.PageContext*	Devuelve el Objeto **request** actual
getParameterNames	*javax.servlet.ServletRequest*	Devuelve los nombres de los parámetros contenidos actualmente en **request**
getParameterValues	*javax.servlet.ServletRequest*	Devuelve los valores de los parámetros contenidos actualmente en **request**
GetParameter	*javax.servlet.ServletRequest*	Devuelve el valor de un parámetro su proporcionamos el nombre

También podemos encontrar otros métodos definidos en ServletRequest, HttpServletRequest, o cualquier otra subclase de ServletRequest que implemente nuestra aplicación.

El motor JSP siempre usa el objeto **request** detrás de la escena, incluso si no la llamamos explícitamente desde el fichero JSP.

4.13.1.1. Uso de los Métodos GET y POST

Los métodos HTTP GET y POST envían datos al servidor. En una Aplicación JSP, GET y POST envían los datos a un Bean, servlet, u otro componente del lado del servidor que está manejando los datos del formulario.

En teoría, GET es para obtener datos desde el servidor y POST es para enviar datos. Sin embargo, GET añade los datos del formulario (llamada una **query string (string de solicitud)**) a una URL, en la forma de parejas clave/valor desde el formulario HTML, por ejemplo, name=john. En el String de solicitud las parejas de clave/valor se separan por caracteres &, los espacios se convierten en caracteres +, y los caracteres especiales se convierten a sus correspondientes hexadecimales. Como el String de solicitud está en la URL, la página puede ser añadida a los bookmarks o enviada por e-mail con el string de solicitud. Este string normalmente está limitado a un número relativamente pequeño de caracteres.

Sin embargo, el método POST, pasa los datos de una longitud ilimitada como un cuerpo de solicitud de cabecera HTTP hacia el servidor. El usuario que trabaja en el navegador cliente no puede ver los datos que están siendo enviados, por eso la solicitud POST es ideal para enviar datos confidenciales (como el número de una tarjeta de crédito) o grandes cantidades de datos al servidor.

Ejemplo de uso de la variable predefinida Request

Este es el archivo HTML que pide los datos al cliente

```
<HTML>
	<head>
		<title> Formulario de petición de nombre </title>
```

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

Fichero JSP que opera dinámicamente con los datos del cliente y muestra los resultados

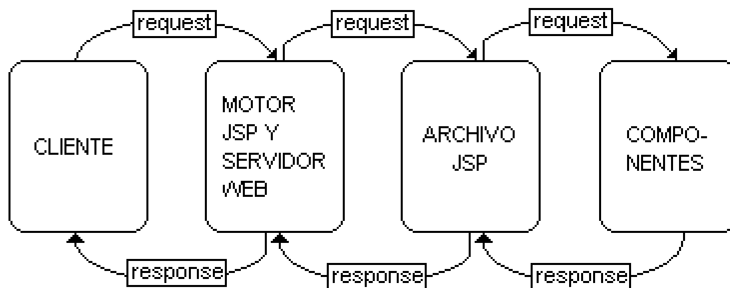
```
<HTML>
	<head>
		<title> Saludo al cliente </title>
	</head>
	<body>
		<h1> Saludo al cliente</h1>
		<!-- Los parámetros que le pasa el cliente en la petición se obtienen
		del objeto implícito request -->
		<%
			String nombre = request.getParameter(”nombre");
			out.println(”Encantado de conocerle,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

```
	</body>
</HTML>
```

4.13.2. response

Este es el HttpServletResponse asociado con la respuesta al cliente. Observa que, como el stream de salida (ver out más abajo) tiene un buffer, es legal seleccionar los códigos de estado y cabeceras de respuesta, aunque no está permitido en los servlets normales una vez que la salida ha sido enviada al cliente.

La siguiente gráfica ilustra detalladamente el proceso de comunicación y paso de datos a través de los objetos **request** y **response:**



4.13.3. out

Este es el PrintWriter usado para enviar la salida al cliente. Sin embargo, para poder hacer útil el objeto response, esta es una versión con buffer de PrintWriter llamada JspWriter. Observa que podemos ajustar el tamaño del buffer, o incluso desactivar el buffer, usando el atributo buffer de la directiva page. También

observa que out se usa casi exclusivamente en scriptlets ya que las expresiones JSP obtienen un lugar en el stream de salida, y por eso raramente se refieren explícitamente a out.

4.13.4 session

Este es el objeto HttpSession asociado con la petición. Recuerda que las sesiones se crean automáticamente, por esto esta variable se une incluso si no hubiera una sesión de referencia entrante. La única excepción es usar el atributo session de la directiva page (ver la Sección 5) para desactivar las sesiones, en cuyo caso los intentos de referenciar la variable session causarán un error en el momento de traducir la página JSP a un servlet.

4.13. 5. application

Es el ServletContext que se obtiene mediante getServletConfig().getContext().

4.13. 6. config

Este es el objeto ServletConfig para la página.

4.13. 7. pageContext

JSP presenta una nueva clase llamada PageContext para encapsular características de uso específicas del servidor como JspWriters de alto rendimiento. La idea es que, si tenemos acceso a ellas a través de esta clase en vez directamente, nuestro código seguirá funcionando en motores servlet/JSP "normales".

4.13. 8. page

Esto es sólo un sinónimo de this, y no es muy útil en Java. Fue creado como situación para el día que el los lenguajes de script puedan incluir otros lenguajes distintos de Java.

4.14. Acciones Estándar JSP

Las **acciones** JSP usan construcciones de sintaxis XML para controlar el comportamiento del motor de Servlets. Podemos insertar un fichero dinámicamente, reutilizar componentes JavaBeans, reenviar al usuario a otra página, o generar HTML para el plug-in Java. Las acciones disponibles incluyen:
Recuerda que, como en XML, los nombre de elementos y atributos son sensibles a las mayúsculas.

4.14.1. Acción jsp:include

Esta acción nos permite insertar ficheros en una página que está siendo generada. La síntaxis se parece a esto:

```
<jsp:include page="relative URL" flush="true" />
```

Al contrario que la directiva include, que inserta el fichero en el momento de la conversión de la página JSP a un Servlet, esta acción inserta el fichero en el momento en que la página es solicitada. Esto se paga un poco en la eficiencia, e imposibilita a la página incluida de contener código JSP general (no puede seleccionar cabeceras HTTP, por ejemplo), pero se obtiene una significante flexibilidad.

Ejemplo de uso de la Acción Include

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

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

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

4.14.2. Acción jsp:useBean

Esta acción nos permite cargar y utilizar un JavaBean en la página JSP. Esta es una capacidad muy útil porque nos permite utilizar la reusabilidad de las clases Java sin sacrificar la conveniencia de añadir JSP sobre servlets solitarios. El sintaxis más simple para especificar que se debería usar un Bean es:

<jsp:useBean id="**name**" class="**package.class**" />

Esto normalmente significa "ejemplariza un objeto de la clase especificada por class, y únelo a una variable con el nombre especificado por id". Sin embargo, también podemos especificar un atributo scope que hace que ese Bean se asocie con más de una sola página. En este caso, es útil obtener referencias a los beans existentes, y la acción jsp:useBean especifica que se ejemplarizará un nuevo objeto si no existe uno con el mismo nombre y ámbito.

Ahora, una vez que tenemos un bean, podemos modificar sus propiedades mediante jsp:setProperty, o usando un scriptlet y llamando a un método explícitamente sobre el objeto con el nombre de la variable especificada anteriormente mediante el atributo id. Recuerda que con los beans, cuando decimos "este bean tiene una propiedad del tipo **X** llamada foo", realmente queremos decir "Esta clase tiene un método getFoo que devuelve algo del tipo **X**, y otro método llamado setFoo que toma un **X** como un argumento". La acción jsp:setProperty se describe con más detalle en la siguiente sección, pero ahora observemos que podemos suministrar un valor explícito, dando un atributo param para decir que el valor está derivado del parámetro de la petición nombrado, o sólo lista las propiedades para indicar que el valor debería derivarse de los parámetros de la petición con el mismo nombre que la propiedad. Leemos las propiedades existentes en una expresión o scriptlet JSP llamando al método **getXxx**, o más comúnmente, usando la acción jsp:getProperty.

Observa que la clase especificada por el bean debe estar en el path normal del servidor, no en la parte reservada que obtiene la recarga automática cuando se modifican. Por ejemplo, en el Java Web Server, él y todas las clases que usa deben ir en el directorio classes o estar en un fichero JAR en el directorio lib, no en el directorio servlets.

además de id y class, hay otros tres atributos que podemos usar: scope, type, y beanName:

- id: Da un nombre a la variable que referenciará el bean. Se usará un objeto bean anterior en lugar de ejemplarizar uno nuevo si se puede encontrar uno con el mismo id y scope.

- Class: Designa el nombre completo del paquete del bean.

- Scope: Indica el contexto en el que el bean debería estar disponible. Hay cuatro posibles valores: page, request, session, y application. El valor por defecto, page, indica que el bean estará sólo disponible para la página actual (almacenado en el PageContext de la página actual). Un valor de request indica que el bean sólo está disponible para la petición actual del cliente (almacenado en el objeto ServletRequest). Un valor de session indica que el objeto está disponible para todas las páginas durante el tiempo de vida de la HttpSession actual. Finalmente, un valor de application indica que está disponible para todas las páginas que compartan el mismo ServletContext. La razón de la importancia del ámbito es que una entrada jsp:useBean sólo resultará en la ejemplarización de un nuevo objeto si no había objetos anteriores con el mismo id y scope. De otra forma, se usarán los objetos existentes, y cualquier elemento jsp:setParameter u otras entradas entre las etiquetas de inicio jsp:useBean y la etiqueta de final, serán ignoradas.

- Type: Especifica el tipo de la variable a la que se referirá el objeto. Este debe corresponder con el nombre de la clase o ser una superclase o un interface que implemente la clase. Hay que recordar que el nombre de la variable se designa mediante el atributo id.

- beanName: Da el nombre del bean, como lo suministraríamos en el método instantiate de Beans. Esta permitido suministrar un type y un beanName, y omitir el atributo class.

4.14.2.1. ¿Cómo funciona el jsp:useBean?

- El contenedor web localiza el objeto (bean) usando el id y el scope especificado.

- Si encuentra el objeto y se especificó el atributo type, entonces el contenedor web castea el objeto hallado con el tipo indicado. Si el casting falla, dispara la excepción java.lang.ClassCastException.

- Si no encuentra al objeto en el alcance especificado, pueden suceder tres cosas:

    - Si no se especificó el atributo class ni el beanName, se dispara la excepción java.lang.InstantiationException.

    - Si se especificó el atributo class, se crea una instancia de la clase especificada, invocando al constructor público sin argumentos y se liga el objeto a la variable id en el scope indicado. Si la clase es abstracta o no tiene constructor público sin argumentos, se dispara la excepción java.lang.InstantiationException.

    - Si se especificó beanName, se invoca al método instantiate() de la clase java.beans.Beans(con el beanName especificado como argumento) y se liga el objeto a la variable id en el scope indicado.

Si el tag jsp:useBean tiene cuerpo no-vació, este se procesa. En este punto, la variable (id) está inicializada y disponible en el cuerpo del bean. Se pueden escribir scripts, la acción estándar jsp:setProperty, texto para ser enviado como respuesta.

4.14.3. Acción jsp:setProperty

Usamos jsp:setProperty para obtener valores de propiedades de los beans que se han referenciado anteriormente. Podemos hacer esto en dos contextos. Primero,

podemos usar antes jsp:setProperty, pero fuera de un elemento jsp:useBean, de esta forma:

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

En este caso, el jsp:setProperty se ejecuta sin importar si se ha ejemplarizado un nuevo bean o se ha encontrado uno ya existente. Un segundo contexto en el que jsp:setProperty puede aparecer dentro del cuerpo de un elemento jsp:useBean, de esta forma:

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

Aquí, el jsp:setProperty sólo se ejecuta si se ha ejemplarizado un nuevo objeto, no si se encontró uno ya existente.

Los valores de las propiedades de un bean, pueden asignarse de las siguientes formas:

- En el momento que se realiza el requerimiento, mediante los valores de parámetros del objeto request.

- En el momento que se realiza el requerimiento, mediante la evaluación de expresiones.

- Escribiendo constantes String.

El contendor web usa introspección para descubrir las propiedades del bean: sus nombres, si se trata de propiedades simples o indexadas, el tipo de dato, sus métodos de acceso (setters y getters).

Aquí tenemos los cuatro atributos posibles de jsp:setProperty:

- name: Este atributo requerido designa el bean cuya propiedad va a ser seleccionada. El elemento jsp:useBean debe aparecer antes del elemento jsp:setProperty.

- property: Este atributo requerido indica la propiedad que queremos seleccionar. Sin embargo, hay un caso especial: un valor de "*" significa que todos los parámetros de la petición cuyos nombres correspondan con nombres de propiedades del Bean serán pasados a los métodos de selección apropiados.

- value: Este atributo opcional especifica el valor para la propiedad. Los valores string son convertidos automáticamente a números, boolean, Boolean, byte, Byte, char, y Character mediante el método estándard valueOf en la fuente o la clase envolvente. Por ejemplo, un valor de "true" para una propiedad boolean o Boolean será convertido mediante Boolean.valueOf, y un valor de "42" para una propiedad int o Integer será convertido con Integer.valueOf. No podemos usar value y param juntos, pero si está permitido no usar ninguna.

- param: Este parámetro opcional designa el parámetro de la petición del que se debería derivar la propiedad. Si la petición actual no tiene dicho parámetro, no se hace nada: el sistema no pasa null al método seleccionador de la propiedad. Así, podemos dejar que el bean suministre los valores por defecto, sobrescribiéndolos sólo cuando el parámetro dice que lo haga. Por ejemplo, el siguiente código dice "selecciona el valor de la

propiedad numberOfItems a cualquier valor que tenga el parámetro numItems de la petición, si existe dicho parámetro, si no existe no se hace nada"

```
<jsp:setProperty name="orderBean"
  property="numberOfItems"
  param="numItems" />
```

Si omitimos tanto value como param, es lo mismo que si suministramos un nombre de parámetro que corresponde con el nombre de una propiedad. Podremos tomar esta idea de automaticidad usando el parámetro de la petición cuyo nombre corresponde con la propiedad suministrada un nombre de propiedad de "*" y omitir tanto value como param. En este caso, el servidor itera sobre las propiedades disponibles y los parámetros de la petición, correspondiendo aquellas con nombres idénticos.

Si los valores de las propiedades, son asignados usando los valores de los parámetros del requerimiento o por constantes Strings, el contenedor realiza las siguientes conversiones de tipos:

Tipo de la Propiedad	Conversión Usada
boolean o Bolean	Java.lang.Boolean.valueOf(String)
byte o Byte	Java.lang.Byte.valueOf(String)
char o Carácter	Java.lang.Character.valueOf(String)
double o Double	Java.lang.Double.valueOf(String)
int o Integer	Java.lang.Integer.valueOf(String)
float o Float	Java.lang.Float.valueOf(String)
long o Long	Java.lang.Long.valueOf(String)

4.14.4. Acción jsp:getProperty

Este elemento recupera el valor de una propiedad del bean, lo convierte a un string, e inserta el valor en la salida. Los dos atributos requeridos son name, el nombre de un bean referenciado anteriormente mediante jsp:useBean, y property, la propiedad cuyo valor debería ser insertado.

Ejemplo:

```
<jsp:useBean id="itemBean" ... />
...
<UL>
  <LI>Number of items:
      <jsp:getProperty name="itemBean" property="numItems" />
  <LI>Cost of each:
      <jsp:getProperty name="itemBean" property="unitCost" />
</UL>
```

Para convertir el valor de la propiedad a String, el contenedor web invoca al método toString()si el valor es un objeto o, es convertido directamente si se trata de un valor primitivo.

4.14.5. Acción jsp:forward

```
<jsp:forward page="URL"/> o <jsp:forward page="URL">
<jsp:param name="Nombre del Parámetro" value="Valor del Parámetro"/>
……..
</jsp>
```

Este tag permite redireccionar el requerimiento a otra página JSP, a un servlet o a un recurso estático que pertenezca a la misma aplicación Web. La ejecución del JSP finaliza cuando encuentra el tag <jsp:forward>. Se limpia el buffer y el requerimiento original es modificado, incluyendo los parámetros especificados.

Tiene un sólo atributo, page, que debería consistir en una URL relativa. Este podría ser un valor estático, o podría ser calculado en el momento de la petición, como en estos dos ejemplo:

Ejemplo de uso de Acción Forward

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

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

• *Página JSP que lo ejecuta*

– No tiene nada de HTML

– En función de los valores de los parámetros de la petición redirige a una segunda página JSP (si es un usuario y una clave determinadas) o bien recarga la página inicial (incluyéndola)

– Mezcla código Java puro con acciones estándar

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

• El programa saludoforward.jsp podría ser el siguiente:

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

4.14.6. Acción jsp:plugin

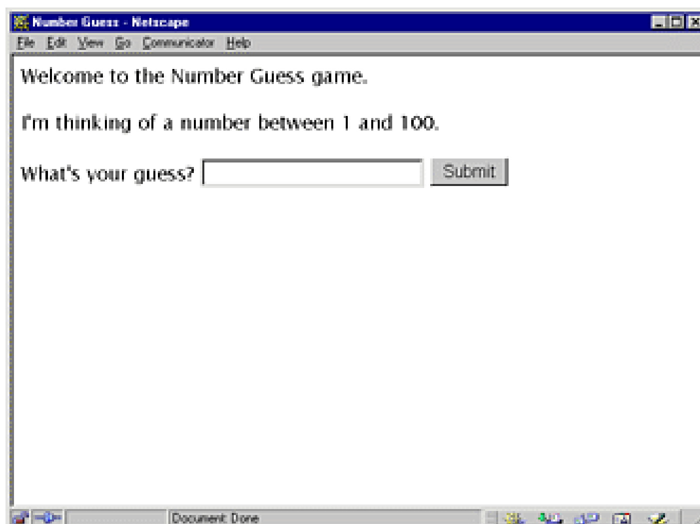
Esta acción nos permite insertar un elemento OBJECT o EMBED específico del navegador para especificar que el navegador debería ejecutar un applet usando el Plug-in Java.

4.15. El Juego de la Adivinanza Numérica

Este juego hace buen uso de scriptlets y expressions, y también del manejo de formularios HTML.

A continuación se muestra el código que compone la pagina de este juego:

numguess.jsp




```
<%@ page import = "num.NumberGuessBean" %>
<jsp:useBean id="numguess" class="num.
      NumberGuessBean" scope="session" />
<jsp:setProperty name="numguess" property="*" />
<html>
<head><title>Number Guess</title></head>
```

```
<body bgcolor="white">
<font size=4>
<% if (numguess.getSuccess() ) { %>
  Congratulations!  You got it.
  And after just <%= numguess.getNumGuesses() %>
    tries.<p>
  <% numguess.reset(); %>
  Care to <a href="numguess.jsp">try again</a>?
<% } else if (numguess.getNumGuesses() == 0) { %>
  Welcome to the Number Guess game.<p>
  I'm thinking of a number between 1 and 100.<p>
    <form method=get>
    What's your guess? <input type=text name=guess>
    <input type=submit value="Submit">
    </form>
<% } else { %>
    Good guess, but nope.  Try <b><%= numguess.
    getHint() %></b>.
    You have made <%= numguess.getNumGuesses() %>
    guesses.
    I'm thinking of a number between 1 and 100.<p>
    <form method=get>
    What's your guess? <input type=text name=guess>
    <input type=submit value="Submit">
    </form>
<% } %>
</font>
</body>
</html>
```

NumberGuessBean.java

```
package num;
        import java.util.*;
        public class NumberGuessBean {
          int answer;
          boolean success;
          String hint;
          int numGuesses;
        public NumberGuessBean() {
          reset();
        }
        public void setGuess(String guess) {
          numGuesses++;
          int g;
          try {       g = Integer.parseInt(guess);
          }
          catch (NumberFormatException e) { g = -1;}
          if (g == answer) {success = true;}
          else if (g == -1) { hint = "a number next time";}
          else if (g < answer) {hint = "higher";}
          else if (g > answer) {hint = "lower";}
        }
          public boolean getSuccess() {return success;}
          public String getHint() {return "" + hint;}
          public int getNumGuesses() {return numGuesses;}
          public void reset() {
                    answer = Math.abs(new Random().nextInt() % 100)+1;
                    success = false;
                numGuesses = 0;
        }
}
```

4.16. Acceso a Bases de Datos

JDBC es una API pura de Java que se usa para ejecutar comandos de SQL. Suministra una serie de clases e interfaces que permiten al desarrollador de web escribir aplicaciones que gestionen Bases de Datos.

La interacción típica con una base de datos consta de los siguientes cuatro pasos básicos:

- Abrir la conexión a la base de datos
- Ejecutar consultas contra la base de datos
- Procesar los resultados
- Cerrar la conexión a la base de datos

Y sin más demora, vamos a ver como se usan los cuatro pasos anteriores:

Paso 1. Abrir la conexión a la base de datos.

```
Connection conexion =
DriverManager.getConnection("jdbc:odbc:Nombre_ODBC","usuario","password");
```

Paso 2. Ejecutar consultas a la base de datos.

```
Statement Estamento = conexion.creStatement();
ResultSet rs = Estamento.executeQuery("select dni,nombre,apellidos,edad from
agenda");
```

Paso 3. Procesar los resultados. En este caso los muestra en pantalla.

```
while (rs.next()) {
        out.println("DNI ->"+ rs.getString("dni"));
        out.println("NOMBRE ->"+ rs.getString("nombre"));
        out.println("APELLIDOS ->"+ rs.getString("apellidos"));
        out.println("EDAD ->"+ rs.getInt("edad"));
}
```

Paso 4 . Cerrar la conexión a la base de datos.

```
rs.close();
Estamento.close();
conexion.close();
```

Lo primero que se debe hacer para poder atacar a bases de datos con páginas JSP o con Servlets, es instalar el Driver correcto de la base de datos que se desee utilizar.

Por ejemplo, supongamos que tenemos una base de datos Microsoft Access. Para instalar el puente JDBC-ODBC, simplemente bajamos el JDK y seguimos la directrices de instalación del mismo. El puente JDBC-ODBC está incluido en JDK desde la versión 1.1.

4.16.1. Configurando ODBC para Access

El puente JDBC-ODBC no necesita pasos específicos para su instalación, pero ODBC si. Por ejemplo si asumimos que estamos utilizando un máquina con Windows/9x o NT. Necesitaremos configurar nuestra conexión mediante ODBC, si has llegado hasta aquí estoy seguro que alguna vez has configurado una conexión mediante ODBC a cualquier base de datos, ya sea Access, Oracle o SQL Server, así que la configuración de la conexión de ODBC, no vamos a detallarla demasiado, sólo informaré de los pasos básicos, que se ven en la siguiente lista:

1. Ir al Panel de Control.
2. Fuentes de datos ODBC.
3. Agregar.
4. Seleccionar Driver Controlador para Microsoft Access (*.mdb).
5. Facilitar a la conexión ODBC las características básicas de la base de datos.

Mediante estos pasos suponemos que ya hemos configurado el ODBC.

4.16.2. Estableciendo la conexión

La primera cosa que debe hacerse para trabajar con bases de datos es establecer la conexión con la base de datos, para ello tenemos dos simples pasos, cargar el driver y establecer la conexión.

Cargar el driver es una operación muy simple que requiere sólo una única línea de código.Si vamos a cargar una fuente ODBC utilizaremos una clase llamada sun.jdbc.odbc.jdbcOdbcDriver y la instrucción de código quedaría así
Class.forName("sun.jdbc.odbc.JdbcOdbDriver);

Cuando se llama a Class.forName, lo que se está haciendo es crear una instancia al driver y registrarlo con el DriverManager. Esto es lo único que hay que hacer para cargar el driver Después de esto, necesitamos establecer una conexión a la base de datos mediante el ODBC que hemos creado anteriormente, para ello bastaría con esta línea de código.
conexion =
DriverManager.getConnection("jdbc:odbc:NombreODBC","nombre_usuario","clave
");

La conexión a la base de datos, ya está realizada, esto es debido al método estático DriverManager.getConnection(); que lo que hace es devolver una conexión a la base de datos.

Los parámetros son tres, uno es la URL dónde se encuentra la base de datos, y los otros se explican por si sólo, el segundo es el nombre de usuario y tercero es el password para poder acceder a la base de datos.

Imaginemos ahora que tenemos una base de datos ORACLE llamada BASE en un servidor cuya dirección HTTP es 171.10.10.1 en el puerto 1515.El nombre de usuario es miguel y la contraseña es gato.

En este caso la conexión se haría de la siguiente manera:

```
String dbURL = "jdbc:oracle:thin:@171.10.10.1:1515:BASE";
String usuario = "miguel";
String pwd = "gato";
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conexion = DriverManager.getConnection(dbURL,usuario,pwd);
```

Con estas instrucciones tan simples y teniendo el diver adecuado de Oracle se haría una conexión a una base de datos ORACLE.

Para ejecutar cualquier instrucción SQL contra la base de datos debemos usar el objeto estamento. Para crearlo tiene que llamarse al método Connection.createStatements(). Este devolverá un estamento que sirve para enviar las consultas a la base de datos. Con la siguiente instrucción se crea un estamento.

```
Statement estamento = conexion.createStatement();
```

Para los que estén familiarizados con Visual Basic, podrán entenderlo mejor si lo consideran como si fuera el objeto Workspace.

El paso siguiente es ejecutar instrucciones SQL, estas instrucciones pueden ser cualquiera que nos haga falta para gestionar nuestra base de datos. Por ejemplo, gracias al Estamento, podemos ejecutar instrucciones como SELECT, UPDATE o DELETE. Veamos un ejemplo completo:

```
import java.sql.*;
public class CrearTablaAgenda{
public void CrearTablas(){
Connection conexion = null;
String strSQL = "";
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
conexion =
DiverManager("jdb:odbc:NombreODBC","nombre_usuario","clave");
Statement Estamento = conexion.createStatement();
strSQL = "CREATE TABLE AGENDA (DNI VARCHAR(5),NOMBRE
VARCHAR(30),APELLIDOS VARCHAR(30))";
Estamento.executeUpdate(strSQL);
} catch (SQLException excepcion_SQL) {
System.err.println(excepcion_SQL.getMessage());
} catch (ClassnotFoundException excepcion_ClassForName) {
System.err.println(excepcion_ClassForName.getMessage());
} catch (Exception excepcion_general) {
System.err.println(excepcion_general.getMessage());
} finally {
try {
if (conexion != null) {
conexion.close();
}
}
}catch (SQLException sql_excepcion) {
System.err.println(sqq_excepcion.getMessage());
```

```
}
}
}
public static void main(String[] argumentos) {
CrearTablaAgenga tablas_agenda = new CrearTablaAgenga();
tablas_agenda.CrearTablas();
}
}
```

En realidad puede llegar a asustar tanto código, pero la verdad es que lo que más ocupa es el control de las excepciones de error, como todos sabemos los errores pueden o no controlarse, es recomendable controlarlos al máximo y además es una buena práctica a la hora de programar. Si nos fijamos la conexión a la base de datos, y la ejecución de la sentencia SQL ocupa sólo cinco líneas (cuatro si quitamos la variable strSQL).

Las instrucciones INSERT, UPDATE y DELETE al ser su codificación igual que la que acabamos de ver, no vamos a remarcar más en ellos, pero lo que si vamos a ver es la cláusula SELECT, ya que tiene un manejo distinto de los resultados.

Lo más común es mostrar los resultados en pantalla de una sentencia SELECT, para ello valga el siguiente fragmento de código:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
cn = DiverManager("jdb:odbc:NombreODBC","nombre_usuario","clave");
StringBuffer strsql = new StringBuffer();
strsql.append(" SELECT ");
strsql.append(" DNI, ");
strsql.append(" NOMBRE, ");
strsql.append(" APELLIDOS, ");
```

```
strsql.append(" EDAD ");
strsql.append(" FROM AGENDA ");
strsql.append(" ORDER BY NOMBRE ");
Statement s=cn.createStatement();
ResultSet rs = s.executeQuery(strsql.toString());
out.println("<TABLE>");
out.println("<TR>");
out.println("<TD>DNI</TD>");
out.println("<TD>NOMBRE</TD>");
out.println("<TD>APELLIDOS</TD>");
out.println("<TD>EDAD</TD>");
out.println("</TR>");
while (rs.next()){
out.println("<TR>");
out.println("<TD>" + rs.getString("DNI") + "</TD>");
out.println("<TD>" + rs.getString("NOMBRE") + "</TD>");
out.println("<TD>" + rs.getString("APELLIDOS") + "</TD>");
out.println("<TD>" + rs.getString("EDAD") + "</TD>");
out.println("</TR>");
}
out.println("</TABLE>");
rs.close();
s.close();
cn.close();
```

5. INTRODUCCIÓN A JAVASERVER FACES

La tecnología JavaServer Faces es un marco de trabajo de interfaces de usuario del lado de servidor para aplicaciones Web basadas en tecnología Java

Los principales componentes de la tecnología JavaServer Faces son:

- Un API y una implementación de referencia para: representar componentes UI y manejar su estado; manejo de eventos, validación del lado del servidor y conversión de datos; definir la navegación entre páginas; soportar internacionalización y accesibilidad; y proporcionar extensibilidad para todas estas características.

- Una librería de etiquetas JavaServer Pages (JSP) personalizadas para dibujar componentes UI dentro de una página JSP.

Este modelo de programación bien definido y la librería de etiquetas para componentes UI facilitan de forma significativa la tarea de la construcción y mantenimiento de aplicaciones Web con UIs del lado del servidor. Con un mínimo esfuerzo, podemos:

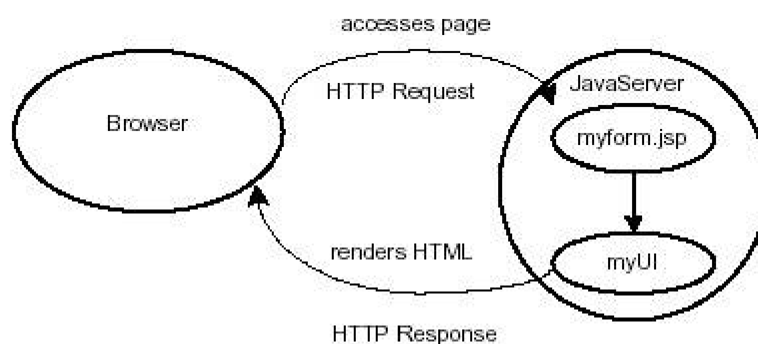
- Conectar eventos generados en el cliente a código de la aplicación en el lado del servidor.

- Mapear componentes UI a una página de datos del lado del servidor.

- Construir un UI con componentes reutilizables y extensibles.

- Grabar y restaurar el estado del UI más allá de la vida de las peticiones de servidor.

Como se puede apreciar en la siguiente figura, el interface de usuario que creamos con la tecnología JavaServer Faces (representado por myUI en el gráfico) se ejecuta en el servidor y se renderiza (convierte) en el cliente.



La página JSP, myform.jsp, dibuja los componentes del interface de usuario con etiquetas personalizadas definidas por la tecnología JavaServer Faces. El UI de la aplicación Web (representado por myUI en la imagen) maneja los objetos referenciados por la página JSP:

- Los objetos componentes que mapean las etiquetas sobre la página JSP.

- Los oyentes de eventos, validadores, y los conversores que está registrados en los componentes.

- Los objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación.

5.1. ¿Por qué utilizar JSF?

Una de las grandes ventajas de la tecnología JavaServer Faces es que ofrece una clara separación entre el comportamiento y la presentación. Las aplicaciones Web construidas con tecnología JSP conseguían parcialmente esta separación. Sin embargo, una aplicación JSP no puede mapear peticiones HTTP al manejo de eventos específicos del componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JavaServer Faces nos permite construir aplicaciones Web que implementan una separación entre el comportamiento y la presentación tradicionalmente ofrecidas por arquitectura UI del lado del cliente.

La separación de la lógica de la presentación también le permite a cada miembro del equipo de desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas. Por ejemplo, los Autores de páginas sin experiencia en programación pueden usar las etiquetas de componentes UI de la tecnología JavaServer Faces para enlazar código de la aplicación desde dentro de la página Web sin escribir ningún script.

Otro objetivo importante de la tecnología JavaServer Faces es mejorar los conceptos familiares de componente-UI y capa-Web sin limitarnos a una tecnología de script particular o un lenguaje de marcas. Aunque la tecnología JavaServer Faces incluye una librería de etiquetas JSP personalizadas para representar componentes en una página JSP, los APIs de la tecnología JavaServer Faces se han creado directamente sobre el API JavaServlet. Esto nos permite hacer algunas cosas: usar otra tecnología de presentación junto a JSP,

crear nuestros propios componentes personalizados directamente desde las clases de componentes, y generar salida para diferentes dispositivos cliente.

Pero lo más importante, la tecnología JavaServer Faces proporciona una rica arquitectura para manejar el estado de los componentes, procesar los datos, validar la entrada del usuario, y manejar eventos.

5.2. ¿Qué se necesita para trabajar con JSF?

Para trabajar con Java Server Faces, es necesario tener lo siguiente:

- El Java SDK 1.4.1 o superior (http://java.sun.com/j2se)
- El servidor Web Apache Tomcat (http://jakarta.apache.org/tomcat/).
- Un editor de texto cualquiera.

5.3. ¿Qué es una Aplicación JavaServer Faces?

En su mayoría, las aplicaciones JavaServer Faces son como cualquier otra aplicación Web Java. Se ejecutan en un contenedor Servlet Java, y típicamente contienen:

- Componentes JavaBeans (llamados objetos del modelo en tecnología JavaServer Faces) conteniendo datos y funcionalidades específicas de la aplicación.
- Oyentes de Eventos.
- Páginas, cómo páginas JSP.

- Clases de utilidad del lado del servidor, como beans para acceder a las bases de datos.

Además de estos ítems, una aplicación JavaServer Faces también tiene:

- Una librería de etiquetas personalizadas para dibujar componentes UI en una página.

- Una librería de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones.
- Componentes UI representados como objetos con estado en el servidor.

- Validadores, manejadores de eventos y manejadores de navegación.

Toda aplicación JavaServer Faces debe incluir una librería de etiquetas personalizadas que define las etiquetas que representan componentes UI y una librería de etiquetas para representar otras acciones importantes, como validadores y manejadores de eventos. La implementación de JavaServer Faces proporciona estas dos librerías.

La librería de etiquetas de componentes “HTML” elimina la necesidad de codificar componentes UI en HTML u otro lenguaje de marcas, resultando en componentes completamente reutilizables. Y, la librería "core" hace fácil registrar eventos, validadores y otras acciones de los componentes.

La librería de etiquetas de componentes puede ser la librería html_basic incluida con la implementación de referencia de la tecnología JavaServer Faces, o podemos definir nuestra propia librería de etiquetas que dibuje componentes personalizados o que dibuje una salida distinta a HTML.

Otra ventaja importante de las aplicaciones JavaServer Faces es que los componentes UI de la página están representados en el servidor como objetos con estado. Esto permite a la aplicación manipular el estado del componente y conectar los eventos generados por el cliente a código en el lado del servidor.

Finalmente, la tecnología JavaServer Faces nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos en el lado del servidor.

Para hacer uso de cualquier etiqueta de JavaServer Faces, es necesario incluir la directiva taglib con la librería que contiene la etiqueta a utilizar. La manera de incluir estas librerias es la siguiente:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
```

Donde el valor del atributo uri únicamente identifica la librería a utilizar. El valor del atributo prefix es usado para diferenciar las etiquetas utilizadas para esa librería de etiquetas. La etiqueta form debe ser referenciada en la pagina con el prefijo *h:* porque la precedencia de la directiva de Liberia de etiquetas usa la *h* para distinguir las etiquetas definidas en html_basic.tld.

```
<h:form …>
```

Una pagina que contenga etiquetas JavaServer Faces esta representada por un árbol de componentes. Como la raíz del árbol esta el componente UIViewRoot. La etiqueta view representa este componente en la pagina. Toda etiqueta de componentes de etiqueta en la pagina deben ser agregado en la etiqueta view, el cual es definida en el jsf_core TLD.

```
<f:view>
  ... otras etiquetas JavaServer Faces, posiblemente mezcladas con otras
  contenido...
```

```
</f:view>
```

En resumen, una pagina JSP que usa etiquetas JavaServer Faces podría verse como esto:

```
        <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
        <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

    <f:view>
        <h:form>
            otras etiquetas JavaServer Faces y etiquetas core, incluyendo
            uno o mas  componentes de botones para en manejo
            de la forma.
        </h:form>
    </f:view>
```

5.4. El Ciclo de Vida de una Página JavaServer Faces

El ciclo de vida de una página JavaServer Faces es similar al de una página JSP: El cliente hace una petición HTTP de la página y el servidor responde con la página traducida a HTML. Sin embargo, debido a las características extras que ofrece la tecnología JavaServer Faces, el ciclo de vida proporciona algunos servicios adicionales mediante la ejecución de algunos pasos extras.

Los pasos del ciclo de vida se ejecutan dependen de si la petición se originó o no desde una aplicación JavaServer Faces y si la respuesta es o no generada con la fase de renderizado del ciclo de vida de JavaServer Faces. Esta sección explica los diferentes escenarios del ciclo de vida. Luego explica cada uno de estas fases del ciclo de vida utilizando el ejemplo guessNumber.

5.4.1. Escenarios de Procesamiento del Ciclo de Vida de una Petición

Una aplicación JavaServer Faces soporta dos tipos diferentes de respuestas y dos tipos diferentes de peticiones:

- Respuesta Faces: Una respuesta servlet que se generó mediante la ejecución de la fase Renderizar la Respuesta del ciclo de vida de procesamiento de la respuesta.

- Respuesta No-Faces: Una respuesta servlet que no se generó mediante la ejecución de la fase Renderizar la Respuesta. Un ejemplo es una página JSP que no incorpora componentes JavaServer Faces.

- Petición Faces: Una petición servlet que fue enviada desde una Respuesta Faces previamente generada. Un ejemplo es un formulario enviado desde un componente de interface de usuario JavaServer Faces, donde la URI de la petición identifica el árbol de componentes JavaServer Faces para usar el procesamiento de petición.

- Petición No-Faces: Una petición Servlet que fue enviada a un componente de aplicación como un servlet o una página JSP, en vez de directamente a un componente JavaServer Faces.

La combinación de estas peticiones y respuestas resulta en tres posibles escenarios del ciclo de vida que pueden existir en una aplicación JavaServer Faces:

- *Escenario 1:* Una Petición No-Faces genera una Respuesta Faces:
  Un ejemplo de este escenario es cuando se pulsa un enlace de una página HTML que abre una página que contiene componentes JavaServer Faces.

Para dibujar una Respuesta Faces desde una petición No-Faces, una aplicación debe proporcionar un mapeo FacesServlet en la URL de la página que contiene componentes JavaServer Faces. FacesServlet acepta peticiones entrantes y pasa a la implementación del ciclo de vida para su procesamiento.

- *Escenario 2:* Una Petición Faces genera una Respuesta No-Faces:

  Algunas veces una aplicación JavaServer Faces podría necesitar redirigir la salida a un recurso diferente de la aplicación Web diferente o generar una respuesta que no contiene componentes JavaServer Faces. En estas situaciones, el desarrollador debe saltarse la fase de renderizado (Renderizar la Respuesta) llamando a FacesContext.responseComplete. FacesContext Contiene toda la información asociada con una Petición Faces particular. Este método se puede invocar durante las fases Aplicar los Valores de Respuesta, Procesar Validaciones o Actualizar los Valores del Modelo.

- *Escenario 3:* Una Petición Faces genera una Respuesta Faces:

  Es el escenario más común en el ciclo de vida de una aplicación JavaServer Faces. Este escenario implica componentes JavaServer Faces enviando una petición a una aplicación JavaServer Faces utilizando el FacesServlet. Como la petición ha sido manejada por la implementación JavaServer Faces, la aplicación no necesita pasos adicionales para generar la respuesta. Todos los oyentes, validadores y conversores serán invocados automáticamente durante la fase apropiada del ciclo de vida estándar, como se describe en la siguiente sección.

5.4.2 Ciclo de Vida Estándar de Procesamiento de Peticiones

La mayoría de los usuarios de la tecnología JavaServer Faces no necesitarán conocer a fondo el ciclo de vida de procesamiento de una petición. Sin embargo, conociendo lo que la tecnología JavaServer Faces realiza para procesar una página, un desarrollador de aplicaciones JavaServer Faces no necesitará preocuparse de los problemas de renderizado asociados con otras tecnologías UI. Un ejemplo sería el cambio de estado de los componentes individuales. Si la selección de un componente como un checkbox afecta a la apariencia de otro componente de la página, la tecnología JavaServer Faces manejará este evento de la forma apropiada y no permitirá que se dibuje la página sin reflejar este cambio.

5.4.2.1. Reconstituir el Árbol de Componentes

Cuando se hace una petición para una página JavaServer Faces, como cuando se pulsa sobre un enlace o un botón, la implementación JavaServer Faces comienza el estado Reconstituir el Árbol de Componentes.

Durante esta fase, la implementación JavaServer Faces construye el árbol de componentes de la página JavaServer Faces, conecta los manejadores de eventos, los validadores y graba el estado en el FacesContext.

5.4.2.2. Aplicar Valores de la Petición

Una vez construido el árbol de componentes, cada componente del árbol extrae su nuevo valor desde los parámetros de la petición con su método decode. Entonces el valor es almacenado localmente en el componente. Si falla la conversión del valor, se genera un mensaje de error asociado con el componente y se pone en la cola de FacesContext. Este mensaje se mostrará durante la fase Renderizar la Respuesta, junto con cualquier error de validación resultante de la fase Procesar Validaciones.

Si durante esta fase se produce algún evento, la implementación JavaServer Faces emite los eventos a los oyentes interesados.
En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

En el caso del componente userNumber de la página greeting.jsp, el valor es cualquier cosa que el usuario introduzca en el campo. Como la propiedad del objeto del model unida al componente tiene un tipo Integer, la implementación JavaServer Faces convierte el valor de un String a un Integer.

En este momento, se han puesto los nuevos valores en los componentes y los mensajes y eventos se han puesto en sus colas.

5.4.2.3. Procesar Validaciones

Durante esta fase, la implementación JavaServer Faces procesa todas las validaciones registradas con los componentes del árbol. Examina los atributos del componente que especifican las reglas de validación y compara esas reglas con el valor local almacenado en el componente. Si el valor local no es válido, la implementación JavaServer Faces añade un mensaje de error al FacesContext y el ciclo de vida avanza directamente hasta la fase Renderizar las Respuesta para que la página sea dibujada de nuevo incluyendo los mensajes de error. Si había errores de conversión de la fase Aplicar los Valores a la Petición, también se mostrarán.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

5.4.2.4. Actualizar los Valores del Modelo

Una vez que la implementación JavaServer Faces determina que el dato es válido, puede pasar por el árbol de componentes y configurar los valores del objeto de modelo correspondiente con los valores locales de los componentes. Sólo se actualizarán los componentes que tenga expresiones valueRef. Si el dato local no se puede convertir a los tipos especificados por las propiedades del objeto del modelo, el ciclo de vida avanza directamente a la fase Renderizar las Respuesta, durante la que se dibujará de nuevo la página mostrando los errores, similar a lo que sucede con los errores de validación.

En este punto, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se han disparado eventos durante esta fase, la implementación JavaServer Faces los emite a los oyentes interesados.

5.4.2.5. Invocar Aplicación

Durante esta fase, la implementación JavaServer Faces maneja cualquier evento a nivel de aplicación, como enviar un formulario o enlazar a otra página.

En este momento, si la aplicación necesita redirigirse a un recurso de aplicación Web diferente o generar una respuesta que no contenga componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Luego la implementación JavaServer Faces configura el árbol de componentes de la respuesta a esa nueva página. Finalmente, la implementación JavaServer Faces transfiere el control a la fase Renderizar la Respuesta.

5.4.2.6. Renderizar la Respuesta

Durante esta fase, la implementación JavaServer Faces invoca las propiedades de codificación de los componentes y dibuja los componentes del árbol de componentes grabado en el FacesContext.

Si se encontraron errores durante las fases Aplicar los Valores a la Petición, Procesar Validaciones o Actualizar los Valores del Modelo, se dibujará la página original. Si las páginas contienen etiquetas output_errors, cualquier mensaje de error que haya en la cola se mostrará en la página.

Se pueden añadir nuevos componentes en el árbol si la aplicación incluye renderizadores personalizados, que definen cómo renderizar un componente. Después de que se haya renderizado el contenido del árbol, éste se graba para que las siguientes peticiones puedan acceder a él y esté disponible para la fase Reconstituir el Árbol de Componentes de las siguientes llamadas.

5.5. USANDO LAS ETIQUETAS CORE

Las etiquetas incluidas en la librerías de etiquetas core de JSF son usadas para realizar acciones independientes de un particular kit de conversión.
estas etiquetas están listadas a continuación:

<f:actionListener>

Registra una instancia ActionListener en el UIComponent asociado con el padre.

Sintaxis:

<f:actionListener type=”*fully-qualified-classname*”/>

Nombre del Atributo	Tipo	Descripción
Type	String	Limita completamente el nombre de la clase Java de un ActionListener para ser creado y registrado.

<f:attribute>

Agrega un atributos configurables a un componente padre.

Sintaxis:

<f:attribute name=”*attribute-name*” value=”*attribute-value*”/>

<f:attribute type=”*attribute-type*” value=”*attribute-value*”/>

Nombre del Atributo	Tipo	Descripción
Name	String	Nombre del componente attribute a utilizar.
Type	Object	Valor del componente attribute a utilizar.

<f:convertDateTime>

Registra una instancia del convertidor DateTime en un componente asociado con el padre.

Sintaxis:

<f:convertDateTime
[dateStyle="{default|short|medium|long|full}"]
[locale="{*locale*" | string}]
[pattern="*pattern*"]
[timeStyle="{default|short|medium|long|full}"]
[timeZone="{*timeZone*| string}"]
[type="{date|time|both}"]/>

Nombre del Atributo	**Tipo**	**Descripción**
dateStyle	String	Estilo de formato predefinido que determina como el componente fecha de una cadena fecha es formateada.
Locale	Locale or String	Locale de estilos predefinidos para fechas y tiempos que son usados durante el formateo.
Pattern	String	Formato de patrón que determina como la cadena date/time debe ser formateada.
timeStyle	String	Estilo de formato predefinido que determina como el componente time de una fecha es formateada.
timeZone	timezone or String	Time zone es el que interpreta algún tipo de información en la cadena fecha (date).
Type	String	Especifica si el valor de la cadena contiene una fecha, tiempo o ambos.

<f:convertNumber>

Registra una instancia del NumberConverter en un componente asociado con el padre.

Sintaxis:

```
<f:convertNumber
[currencyCode="currencyCode"]
[currencySymbol="currencySymbol"]
[groupingUsed="{true|false}"]
[integerOnly="{true|false}"]
[locale="locale"]
[maxFractionDigits="maxFractionDigits"]
[maxIntegerDigits="maxIntegerDigits"]
[minFractionDigits="minFractionDigits"]
[minIntegerDigits="minIntegerDigits"]
[pattern="pattern"]
[type="{number|currency|percent}"]/>
```

<f:converter>

Registra una llamada instancia Converter arbitrariamente en el UIComponent asociado con el padre.

Sintaxis:

```
<f:converter converterId="converterId"/>
```

Nombre del Atributo	Tipo	Descripción
converterIdname	String	Identificador Converter del convertidor que se creará.

<f:facet>

Registra una faceta llamada desde el UIComponent asociado con e padre.

Sintaxis:

<f:facet name="*facet-name*"/>

Nombre del Atributo	Tipo	Descripción
Name	String	Nombre de la faceta que se creará.

<f:loadBundle>

Especifica un ResourceBundle que es expuesto como un mapa.

Sintaxis:

<f:loadBundle basename="*resource-bundle-name*" var="*attributeKey*"/>

Nombre del Atributo	Tipo	Descripción
Basename	String	Nombre base del recurso a cargar.

<f:param>

Sustituye parámetros en una instancia de MessageFormat y agrega pares de cadena (name-value) a una URL.

Sintaxis:

Sintaxis 1: Valor no llamada

<f:param [id="*componentId*"] value="*parameter-value*"

[binding="*componentReference"]*/>

Sintaxis 2: Valor llamada

<f:param [id="*componentId*"]

[binding="*componentReference"]*

name="*parameter-name*" value="*parameter-value*"/>

<f:selectItem>

Representa un item en una lista de ítems en un componente UISelectMany o UISelectOne.

Sintaxis:

Sintaxis 1: Valor directamente especificado

<f:selectItem [id="*componentId*"]

[binding="*componentReference"]*

[itemDisabled="{true|false}"]

itemValue="*itemValue"*

itemLabel="*itemLabel"*

[itemDescription="*itemDescription"]*/>

Sintaxis 2: Especifica un valor indirecto

<f:selectItem [id="*componentId*"]

[binding="*componentReference"]*

value="*selectItemValue"/*>

<f:selectItems>

Representa un conjunto de ítems en un componente UISelectOne o UISelectMany.

Sintaxis:

<f:selectItems [id="*componentId*"]

[binding="*componentReference"]*

value="*selectItemsValue"*/>

<f:subview>

Contiene todas las etiquetas de JSF en una pagina que esta incluida en otra pagina JSP que contiene etiquetas JSF.

Sintaxis:

<f:subview id="*componentId*"

[binding="*componentReference"]*

[rendered="{true|false}"]>

</f:subview>

<f:validateDoubleRange>

Registra un DoubleRangeValidator en un componente.

Sintaxis:

Sintaxis 1: se especifica solo el máximo

<f:validateDoubleRange maximum="*543.21"*/>

Sintaxis 2: se especifica solo el mínimo

<f:validateDoubleRange minimum="*123.45*"/>

Sintaxis 3: especifica ambos valores, máximo y mínimo

<f:validateDoubleRange maximum="*543.21*" minimum="*123.45*"/>

<f:validateLength>

Registra un LengthValidator en un componente.

Sintaxis:

Sintaxis 1: se especifica solo el máximo

<f:validateLength maximum="*16"*/>

Sintaxis 2: se especifica solo el mínimo

<f:validateLength minimum="*3*"/>

Sintaxis 3: especifica ambos valores, máximo y mínimo

<f:validateLength maximum="*16*" minimum="*3*"/>

<f:validateLongRange>

Registra un LongRangeValidator en un componente.

Sintaxis:

Sintaxis 1: se especifica solo el máximo

<f:validateLongRange maximum="*543"*/>

Sintaxis 2: se especifica solo el mínimo

<f:validateLongRange minimum="*123*"/>

Sintaxis 3: especifica ambos valores, máximo y mínimo

<f:validateLongRange maximum="*543*" minimum="*123*"/>

<f:validator>

Registra un custom Validator en un UIComponente.

Sintaxis:

<f:validator validatorId="*validatorId*"/>

<f:valueChangeListener>

Registra un valor, cambia el listener en un componente padre.

Sintaxis:

<f:valueChangeListener type="*fully-qualified-classname*"/>

<f:verbatim>

Genera una pequeña instancia UIOutput en el UIComponent asociado con el padre.

Sintaxis:

<f:verbatim [escape="{true|false}"]/>

<f:view>

Agrega todas las etiquetas JSF en la página.

Sintaxis:

<f:view [locale="*locale*">

…

</f:view>

5.6. USANDO EL COMPONENTE DE ETIQUETAS HTML

Las etiquetas definidas por el Standard HTML de JavaServer Faces suministra los kits de librerías de etiquetas representados por los controles HTML form y otros elementos básicos de HTML. A su vez, estos controles muestran datos o aceptan datos de el usuario. Este dato es recolectado como parte de una forma y es sometido al servidor usualmente cuando el usuario hace clic en un botón.

5.6.1. ATRIBUTOS DE LOS COMPONENTES DE ETIQUETAS UI

En general, muchos de los componentes de etiquetas soportan estas atributos:

- *Id:* únicamente identifica el componente. Si no incluimos un atributo *id*, la implementación JavaServer Faces automáticamente genera un componente ID.

- *Inmediate:* si es verdadero, indica que algún evento, validación y conversión asociada con el componente debería aplicar la solicitud del valor de la fase seguida que la posterior.

  Asumamos que tenemos una pagina con un botón y un campo para introducir la cantidad de un libro en un carro de compra. Si ambos atributos *immediate* del botón y del campo son colocados a true (verdadero), el nuevo valor del campo estaría disponible para cualquier proceso asociado con el evento que es generado cuando al botón se le hace click.

Si el atributo *immediate* del boton es true pero el atributo *immediate* del campo es false, el evento asociado con el boton es procesado sin actualizar el valor local del campo.

- *Style:* especiita un estilo de cascada (CSS Cascading Style Sheet) para la etiqueta.

- *styleClass:* especifica una clase CSS que contiene definiciones de estilos o styles.

```
<h:dataTable id="books"
 ...
 styleClass="list-background"
 value="#{bookDBAO.books}"
 var="book">
```

- *Value:* identifica un dato externo y une los valores de los componentes a esto.

- *Binding:* identifica una propiedad de un bean y une la instancia del componente a esto.

Todas los atributos de las etiquetas de los componentes UI (excepto “id “& “var”) son valores unidos y activos, lo cual significa que son expresiones aceptadas por JavaServer Fases EL.(Expession Languaje).

5.6.2. Modelo de Componentes de Interface de Usuario

Los componentes UI JavaServer Faces son elementos configurables y reutilizables que componen el interface de usuario de las aplicaciones JavaServer Faces. Un componente puede ser simple, como un botón, o compuesto, como una tabla, que puede estar compuesta por varios componentes.

La tecnología JavaServer Faces proporciona una arquitectura de componentes rica y flexible que incluye:

- Un conjunto de clases UIComponent para especificar el estado y comportamiento de componentes UI.

- Un modelo de renderizado que define cómo renderizar los componentes de diferentes formas.

- Un modelo de eventos y oyentes que define cómo manejar los eventos de los componentes.

- Un modelo de conversión que define cómo conectar conversores de datos a un componente.

- Un modelo de validación que define cómo registrar validadores con un componente.

5.6.2.1. Las Clases de los Componentes del Interface de Usuario

La tecnología JavaServer Faces proporciona un conjunto de clases de componentes UI, que especifican toda la funcionalidad del componente, cómo mantener su estado, mantener una referencia a objetos del modelo, y dirigir el manejo de eventos y el renderizado para un conjunto de componentes estándar.

Estas clases son completamente extensibles, lo que significa que podemos extenderlas para crear nuestros propios componentes personalizados.

Todas las clases de componentes UI de JavaServer Faces descienden de la clase UIComponentBase, que define el estado y el comportamiento por defecto de un UIComponent. El conjunto de clases de componentes UI incluido en la última versión de JavaServer Faces es:

- UICommand: Representa un control que dispara acciones cuando se activa. El componente UICommand realiza una acción cuando se activa.

- UIForm: Encapsula un grupo de controles que envían datos de la aplicación. Este componente es análogo a la etiqueta form de HTML.

- *UIcolumn:* El componente de UIColumn representa una columna de datos en un componente de UIData.

- *UIGraphic:* Muestra una imagen.

- *UIInput:* Toma datos de entrada del usuario. Esta clase es una subclase de UIOutput.

- *UIOutput:* Muestra la salida de datos en un página.

- *UIPanel:* Muestra una tabla.

- *UIParameter:* Representa la sustitución de parámetros.

- *UISelectItem:* Representa un sólo ítem de un conjunto de ítems.

- *UISelectItems:* Representa un conjunto completo de ítems.

- *UISelectBoolean:* Permite a un usuario seleccionar un valor booleano en un control, seleccionándolo o deseleccionándolo. Esta clase es una subclase de UIInput.
- *UISelectMany:* Permite al usuario seleccionar varios ítems de un grupo de ítems. Esta clase es una subclase de UIInput.

- *UISelectOne:* Permite al usuario seleccionar un ítem de un grupo de ítems. Esta clase es una subclase de UIInput.

La mayoría de los autores de páginas y de los desarrolladores de aplicaciones no tendrán que utilizar estas clases directamente. En su lugar, incluirán los componentes en una página usando la etiqueta correspondiente al componente. La mayoría de estos componentes se pueden renderizar de formas diferentes. Por ejemplo, un UICommand se puede renderizar como un botón o como un hiperenlace.

5.6.2.2. El Modelo de Conversión (Renderizado) de Componentes

La arquitectura de componentes JavaServer Faces está diseñada para que la funcionalidad de los componentes se defina mediante las clases de componentes, mientras que la conversión de los componentes se puede definir mediante un convertidor separado. Este diseño tiene varios beneficios:

- Los escritores de componentes pueden definir sólo una vez el comportamiento de un componente, pero pueden crear varios convertidores, cada uno de los cuales define una forma diferente de dibujar el componente para el mismo cliente o para diferentes clientes.

- Los autores de páginas y los desarrolladores de aplicaciones pueden modificar la apariencia de un componente de la página seleccionando la etiqueta que representa la combinación componente/convertidor apropiada.

Un kit de conversión (Renderkit) define como se mapean las clases de los componentes a las etiquetas de componentes apropiadas para un cliente particular. La implementación JavaServer Faces incluye un kit de conversión estándar para cambiar a un cliente HTML.

Por cada componente UI que soporte un kit de conversión, éste define un conjunto de objetos Renderer. Cada objeto Renderer define una forma diferente de dibujar el componente particular en la salida definida por el kit de conversión. Por ejemplo, un componente UISelectOne tiene tres Renderer diferentes. Uno de ellos dibuja el componente como un conjunto de botones de radio. Otro dibuja el componente como un ComboBox. Y el tercero dibuja el componente como un ListBox.

Cada etiqueta JSP personalizada en el kit de conversión de HTML está compuesta por la funcionalidad del componente, definida en la clase UIComponent, y los atributos de conversión, definidos por el convertidor.

La parte command de las etiquetas corresponde con la clase UICommand, y especifica la funcionalidad, que es disparar un action. Las partes del botón y el hiperenlace de las etiquetas corresponden a un convertidor (renderizador) independiente, que define cómo dibujar el componente.

La implementación de referencia de JavaServer Faces proporciona una librería de etiquetas personalizadas para renderizar componentes en HTML. Soporta todos los componentes listados en la siguiente tabla:

input_text: Permite al usuario introducir un string. Equivale a un elemento <input type=text> en HTML. Su aspecto es la de un campo de texto.

<h:inputText value="#{form.testString}"
readonly="true"/>

<h:inputText value="Texto ingresado"
style="color: Black; background:
Teal;"/>

<h:inputText value="1234567"
size="5"/>

<h:inputText value="1234567890"
maxlength="6" size="10"/>

input_secret: Permite al usuario introducir un string sin que aparezca el string real en el campo. Corresponde a un elemento <input type=password> de HTML. Su apariencia es un campo de texto, que muestra una fila de caracteres en vez del texto real introducido.

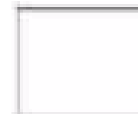
<h:inputSecret value="#{form.passwd}"
redisplay="true"/>

<h:inputSecret value="#{form.passwd}"
redisplay="false"/>

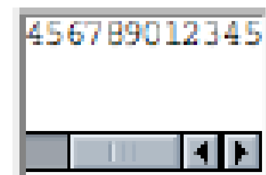
input_textarea: su función es la de Permitir al usuario introducir un texto multi-líneas. Equivale a un elemento <textarea> en HTML. Su apariencia es n campo de texto multi-línea.

<h:inputTextarea rows="5"/>

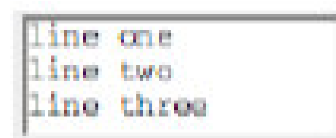
<h:inputTextarea cols="5"/>



<h:inputTextarea
value="123456789012345"
Rows="3" cols="10"/>



<h:inputTextarea
value="#{form.dataInRows}"
Rows="2" cols="15"/>



El h:inputTextarea tiene atributos cols y rows para especificar el numero de columnas filas respectivamente en el área de texto.

input_date: tiene la función de permitirle al usuario introducir una fecha. Su equivalencia en HTML es la de un elemento <input type=text>. Su apariencia es la de un string de texto, formateado con un ejemplar de java.text.DateFormat.

input_datetime: Esta permite al usuario introducir una fecha y una hora. Su equivalencia en HTML es la de un elemento <input type=text>. Su apariencia es la de un string de texto, formateado con un ejemplar de java.text.SimpleDateFormat.

input_hidden: su función consiste en permitir que se introduzca una variable oculta en una página. Corresponde a un elemento <input type=hidden> de HTML. No tiene ninguna apariencia.

input_number: Permite al usuario introducir un número. Representa un elemento <input type=text> de HTML. Tiene apariencia de un string de texto, formateado con un ejemplar de java.text.NumberFormat.

input_time: Permite al usuario introducir una hora. Representa un elemento <input type=text> de HTML. Su apariencia es la de un string de texto, formateado con un ejemplar de java.text.DateFormat.

output_text: Muestra una línea de texto. La manera de mostrar el mensaje es como texto normal.

```
<h:outputText
value="#{form.testString}"/>
```

```
<h:outputText value="Number
#{form.number}"/>
```

Number 1000

```
<h:outputText  value="<input  type='text'
value='hello'/>"/>
```

hello

```
<h:outputText escape="true"
value="<input type='text'
value='hello'/>"/>
```

<input type="text" value="hello">

output_date: Muestra una fecha formateada. Su forma de mostrar la fecha es como texto normal. Tiene aspecto de un string de texto, formateado con un ejemplar de java.text.DateFormat.

output_datetime: Muestra una fecha y hora formateados. Muestra los datos como texto normal. Su apariencia es un string de texto, formateado con un ejemplar de java.text.SimpleDateFormat.

Output_errors: Muestra mensajes de error a manera de texto normal. Su apariencia, de igual manera es la de un texto normal.

output_label: su función es la de Mostrar un componente anidado como una etiqueta para un campo de texto especificado. Corresponde a un elemento <label> de HTML y los muestra a manera de texto normal.

output_message: Muestra un mensaje localizado (internacionalizado). La manera de mostrar el mensaje es como texto normal.

output_number: Muestra un número formateado. La manera de mostrar el mensaje es como texto normal. Corresponde a un string de texto, formateado con un ejemplar de java.text.NumberFormat .

output_time: Muestra una hora formateada La manera de mostrar el mensaje es como texto normal. Corresponde a un string de texto, formateado con un ejemplar de java.text.DateFormat.

graphic_image: Esta etiqueta simplemente muestra una imagen. Es traducido como un elemento <img> en HTML.

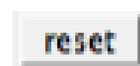
```
<h:graphicImage
value="/tjefferson.jpg"/>
```

```
<h:graphicImage value="/tjefferson.jpg"
style="border: thin solid black"/>
```

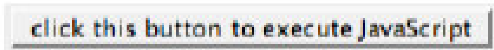
command_button: esta etiqueta envía un formulario a la aplicación, representa a un elemento <input type=type> en HTML, donde el valor del tipo puede ser submit, reset, o image. Su apariencia es la de un Botón.

```
<h:commandButton value="submit"
type="submit"/>
```

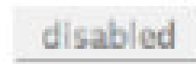
```
<h:commandButton value="reset"
type="reset"/>
```



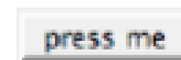
```
<h:commandButton value="click this
button..."
onclick="alert('button clicked')"
type="button"/>
```




```
<h:commandButton value="disabled"
disabled="#{not form.buttonEnabled}"/>
```




```
<h:commandButton
value="#{form.buttonText}"
type="reset"/>
```



commandLink: Enlaza a otra pagina o localización en otra pagina; corresponde a un elemento <a href> en HTML. Su apariencia es la de un Hiperenlace.

```
<h:commandLink> <h:outputText
value="register"/> </h:commandLink>
```

```
<h:commandLink style="font-style: italic">
<h:outputText value="#{msgs.linkText}"/>
</h:commandLink>
```

```
<h:commandLink>
<h:outputText value="#{msgs.linkText}"/>
<h:graphicImage value="/registration.jpg"/>
</h:commandLink>
```

```
<h:commandLink value="welcome"
actionListener="#{form.useLinkValue}"
```

```
action="#{form.followLink}">

<h:commandLink>
<h:outputText value="welcome"/>
<f:param name="outcome" value="welcome"/>
</h:commandLink>
```

form: Representa un formulario de entrada. Las etiquetas internas del formulario reciben los datos que serán enviados con el formulario. Es equivalente a un elemento <form> de HTML.

panel_data: Itera sobre una colección de datos. Tiene como apariencia un conjunto de filas en una tabla

panel_grid: Muestra una tabla. Corresponde a un elemento <label> HTML. con elementos <tr> y %lt;td>. Tiene como apariencia una tabla.

panel_group: Su función es la de agrupar un conjunto de paneles bajo un padre y su apariencia es la de una fila en una tabla.

panel_list: Muestra una tabla de datos que vienen de una collection, un array, un iterator, o un map. Representa un elemento <table> de HTML. con elementos <tr> y %lt;td>. Su apariencia es de una tabla.

selectboolean_checkbox: Permite al usuario cambiar el valor de una elección booleana. Equivale a un elemento <input type=checkbox> de HTML. Su apariencia es la de un checkBox.

selectitem: Representa un ítem de una lista de ítems en un componente UISelectOne. Equivale a un elemento <option> HTML. No tiene ningún tipo de apariencia.

selectitems: Representa una lista de ítems en un componente UISelectOne. Equivale a un elemento <option> HTML. No tiene ningún tipo de apariencia.

selectmany_checkboxlist: Muestra un conjunto de checkbox, en los que el usuario puede seleccionar varios. Representa un conjunto de elementos <input> en HTML. La forma en que se visualiza es como un conjunto de CheckBox.

selectmany_listbox: Permite a un usuario seleccionar varios ítems de un conjunto de ítems, todos mostrados a la vez. Representa Un conjunto de elementos <select> HTML.. su apariencia es de un ListBox.

selectmany_menu: Permite al usuario seleccionar varios ítems de un grupo de ítems. Se representa como un conjunto de elementos <select> HTML. Su apariencia es de un comboBox.

selectone_listbox: Permite al usuario seleccionar un ítem de un grupo de ítems.se representa un conjunto de elementos <select> HTML. Su apariencia es de un listBox.

selectone_menu: Permite al usuario seleccionar un ítem de un grupo de ítems. Se representa como un conjunto de elementos <select> HTML. Su apariencia es un comboBox.

selectone_radio: Permite al usuario seleccionar un ítem de un grupo de ítems. Se representa un conjunto de elementos <input type=radio> HTML y su apariencia es de un conjunto de botones de radio.

5.6.2.3. Modelo de Eventos y Oyentes

Un objetivo de la especificación JavaServer Faces es mejorar los modelos y paradigmas existentes para que los desarrolladores se puedan familiarizar rápidamente con el uso de JavaServer Faces en sus aplicaciones. En este espíritu, el modelo de eventos y oyentes de JavaServer Faces mejora el diseño del modelo de eventos de JavaBeans, que es familiar para los desarrolladores de GUI y de aplicaciones Web.

Al igual que la arquitectura de componentes JavaBeans, la tecnología JavaServer Faces define las clases Listener y Event que una aplicación puede utilizar para manejar eventos generados por componentes UI. Un objeto Event identifica al componente que lo generó y almacena información sobre el propio evento. Para ser notificado de un evento, una aplicación debe proporcionar una implementación de la clase Listener y registrarla con el componente que genera el evento. Cuando el usuario activa un componente, como cuando pulsa un botón, se dispara un evento. Esto hace que la implementación de JavaServer Faces invoque al método oyente que procesa el evento. JavaServer Faces soporta dos tipos de eventos: eventos value-changed y eventos action.

Un evento value-changed ocurre cuando el usuario cambia el valor de un componente. Un ejemplo es seleccionar un checkbox, que resulta en que el valor del componente ha cambiado a true. Los tipos de componentes que generan estos eventos son los componentes UIInput, UISelectOne, UISelectMany, y UISelectBoolean. Este tipo de eventos sólo se dispara si no se detecta un error de validación.

Un evento action ocurre cuando el usuario pulsa un botón o un hiperenlace. El componente UICommand genera este evento.

5.6.2.4. Modelo de Validación

La tecnología JavaServer Faces soporta un mecanismo para validar el dato local de un componente durante la fase del Proceso de Validación, antes de actualizar los datos del objeto modelo.

Al igual que el modelo de conversión, el modelo de validación define un conjunto de clases estándar para realizar chequeos de validación comunes. La librería de etiquetas jsf-core también define un conjunto de etiquetas que corresponden con las implementaciones estándar de Validator.

La mayoría de las etiquetas tienen un conjunto de atributos para configurar las propiedades del validador, como los valores máximo y mínimo permitidos para el dato del componente. El desarrollador de la página registra el validador con un componente anidando la etiqueta del validador dentro de la etiqueta del componente.

Al igual que el modelo de conversión, el modelo de validación nos permite crear nuestras propias implementaciones de Validator y la etiqueta correspondiente para realizar validaciones personalizadas.

5.7. Modelo de Navegación

Virtualmente todas las aplicaciones Web están hechas de un conjunto de páginas. Uno de los principales problemas de un desarrollador de aplicaciones Web es manejar la navegación entre esas páginas.

El nuevo modelo de navegación de JavaServer Faces facilita la definición de la navegación de páginas y el manejo de cualquier procesamiento adicional necesario para elegir la secuencia en se que cargan las páginas. En muchos

casos, no se requiere código para definir la navegación. En su lugar, la navegación se puede definir completamente en el fichero de configuración de la aplicación usando un pequeño conjunto de elementos XML. La única situación en que necesitaremos proporcionar algo de código es si necesitamos algún procesamiento adicional para determinar qué página mostrar luego.

Para cargar la siguiente página en una aplicación web, el usuario normalmente pulsa un botón. Una pulsación de un botón genera un evento action. La implementación de JavaServer Faces proporciona un nuevo oyente de eventos action por defecto para manejar este evento. Este oyente determina la salida del evento action, como success o failure. Esta salida se puede definir como una propiedad String del componente que generó el evento o como el resultado de un procesamientro extra realizado en un objeto Action asociado con el componente. Después de determinar la salida, el oyente la pasa al ejemplar de NavigationHandler asociado con la aplicación. Basándose en la salida devuelta, el NavigationHandler selecciona la página apropiada consultando el fichero de configuración de la aplicación.

5.8. Formato Básico Para Trabajar con Javaserver Faces

Para trabajar con JSF hay que seguir una estructura con ciertos elementos mencionados anteriormente en esta guía:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
…
```

```
<BODY>
…
        <h:form>
        …
        </h:form>
…
</BODY>
</f:view>
```

5.9. Ejemplo JavaServer Faces

jsfevent.jsp

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

<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
```

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<f:view>
        <html>
                <body              bgcolor="cyan">
                        <h:form       id="form1" >
                                <h:commandButton   value="mathew"
        action="success" />
                                <h:commandButton  value="amy"  action="failure" />
                        </h:form>
                </body>
        </html>
</f:view>
```

mathew.jsp

```
<html>
        <body>
              I  AM  MATHEW
        </body>
</html>
```

amy.jsp

```
<html>
        <body>

                        I AM AMY
```

```
        <br>
    </body>
</html>
```

faces-config.xml

```
<navigation-rule>

    <from-view-id>
    /jsfevent.jsp
    </from-view-id>

     <navigation-case>
         <from-outcome>success</from-outcome>
         <to-view-id>/mathew.jsp</to-view-id>
     </navigation-case>

     <navigation-case>
         <from-outcome>failure</from-outcome>
         <to-view-id>/amy.jsp</to-view-id>
      </navigation-case>

</navigation-rule>
```

6. CONCLUSION

Con el desarrollo de esta investigación concluimos que las tecnologías JSP y JSF son herramientas orientadas básicamente en etiquetas que facilitan el desarrollo de aplicaciones Web.

JSP es una tecnología con una característica muy importante que es la de combinar o incrustar código java junto con código HTML . Esta combinación es posible realizarla de dos formas como lo son el insertar código java a través de

Scripts, y la de realizar un llamado de un archivo contenedor de código java. El trabajar con JSP le aporta grandes beneficios al gremio de programadores Web, debido a la facilidad de proteger el código, logrando así un alto grado de confidencialidad de su forma de programar. En pocas palabras JSP hace muy práctico separar la lógica del negocio de la representación de los datos.

La especificación JSF se destaca por ofrecer una clara separación entre el comportamiento y la presentación. Esto se da gracias a sus librerías de etiquetas Core y HTML; la primera consta de un conjunto de etiquetas personalizadas para representar manejadores de eventos, validadores, y otras acciones; la segunda que corresponde al conjunto de etiquetas personalizadas para dibujar componentes UI en una página. Además la tecnología JSF nos permite convertir y validar datos sobre componentes individuales y reportar cualquier error antes de que se actualicen los datos en el lado del servidor.

Para que las características y especialidades de ambas tecnologías sean entendibles, esta documentación esta provista de sencillos ejemplos que ayudan a que el aprendiz mantenga un ritmo adecuado para su enseñanza, obteniendo así información de cada uno de los conceptos básicos que necesita para comenzar a incursionar en estas especificaciones de la tecnología Java.

7. RECOMENDACIONES

Es necesario tener en cuenta que el presente documento solo es una pequeña introducción a lo que respecta a las especificaciones JavaServer Pages y JavaServer Faces, y que solo nos muestra una parte general de estas, dejándole así al aprendiz el compromiso de seguir investigando y avanzando cada vez mas en estas tecnologías. Y es por esto que vale destacar la recomendación de seguir afianzando los conocimientos obtenidos de esta documento guía con otros libros y documentos que bien pueden ser encontrados a través de Internet.

Sería conveniente que para la utilización de este documento guía, el aprendiz de estas especificaciones tenga cierto grado de conocimiento a cerca del lenguaje de programación JAVA, pues, como se ha visto anteriormente JSP y JSF se basan prácticamente en la utilización del lenguaje de programación JAVA junto a HTML.

Para nosotros seria de mucho agrado que la Universidad Tecnológica de Bolívar creara centros de investigación o generara proyectos con ente tipo de especificaciones (JSP y JSF); teniendo en cuenta que estas hacen parte de la tecnología JAVA, la cual esta cruzando por una etapa de mucho furor debido a sus características especiales, y que ayudarían mucho mas a la proliferación y evolución de dicha tecnología.

8. GLOSARIO

API: Interfaz de programación de aplicaciones (Applications Programming Interface): una serie de funciones que están disponibles para realizar programas para un cierto entorno.

APPLET: Pequeñas aplicaciones escritas en Java que se incluye página Web (HTML) y que se puede ejecutar en cualquier navegador que disponga de un

intérprete Java, sin que para su uso necesite intercambiar Información con el servidor ya que siempre se ejecuta en el “cliente”.

BEAN: Bean un componente software que tiene la particularidad de ser reutilizable y que evitan la tediosa tarea de programarlos uno a uno. Se puede decir que existen con la finalidad de ahorrarnos tiempo al programar. Es el caso de la mayoría de componentes que manejan los editores visuales más comunes.

BUFER: Memoria intermedia que almacena temporalmente datos que están en ruta de la memoria principal a otra computadora o a un dispositivo de entrada/salida.

CGI: software que facilita la comunicación entre un servidor Web y los programas que funcionan fuera del servidor. Por ejemplo, aquellos que procesan formularios interactivos o que busquen información en las bases de datos.

COMPONENTE: Parte discreta de un sistema capaz de operar independientemente, pero diseñada, construida y operada como parte integral del sistema.

DRIVER: pequeño programa cuya función es controlar el funcionamiento de un dispositivo del ordenador bajo un determinado sistema operativo.

FICHEROS: Un archivo o fichero informático es una entidad lógica compuesta por una secuencia finita de bytes, almacenada en un sistema de archivos ubicada en la memoria secundaria de un ordenador. Los archivos son agrupados en directorios dentro del sistema de archivos y son identificados por un nombre de archivo.

HTTP: (HyperText Transmission Protocol) Protocolo para transferir archivos o documentos hipertexto a través de la red. Se basa en una arquitectura cliente/servidor.

INTERFACE: Una interfaz es la parte de un programa informático que permite a éste comunicarse con el usuario o con otras aplicaciones permitiendo el flujo de información.

PLUGIN: Un plugin (o plug-in) es un programa de ordenador que interactúa con otro programa para aportarle una función o utilidad específica, generalmente muy específica.

RENDERIZAR: Es la acción de asignar y calcular todas las propiedades de un objeto antes de mostrarlo en pantalla. -Proceso mediante el cual el ordenador crea una imagen partiendo de la descripción de las características de los objetos que contiene.

SCRIPT: En la programación de computadoras es un programa o una secuencia de instrucciones que es interpretado y llevado a cabo por otro programa en lugar de ser procesado por el procesador de la computadora.

SCRIPTLETS: es una pieza de código Java incrustado en código HTML.

SERVLET: Los servlets son objetos que corren dentro del contexto de un servidor de aplicaciones (ej: Tomcat) y extienden su funcionalidad.

SITES: Conjunto de páginas de Internet dedicadas a mostrar las actividades de una empresa o entidad.

STREAM: Un flujo continuo de datos, generalmente codificado de manera digital, diseñados para ser procesados de forma secuencial.

THREADS: Muchos lenguajes de programación (como Java), y otros entornos de desarrollo soportan los llamados hilos o hebras (en inglés, threads). Los hilos son similares a los procesos en que ambos representan una secuencia simple de instrucciones ejecutada en paralelo con otras secuencias. Los hilos son una forma de dividir un programa en dos o más tareas que corren simultáneamente.

URL: (Uniform Resource Locator) Refiere la situación de un documento en Internet. Dirección global de documentos y otras fuentes en la World Wide Web.

WEB: Del inglés, tela de araña. Conjunto de páginas de Internet reunidas bajo un mismo tema. Últimamente, se dedica más este término a las páginas personales, utilizando site para las empresas.

9. BIBLIOGRAFIA

- **Laxxuss**, Core JavaServer Faces [JSF] : Prentice Hall 2004

- **Laxxuss**, Core JSP : Prentice Hall 2004
- **Hans Bergsten**, JavaServer Pages_2nd Edition : O'Reilly 2002
- **Miguel Angel García**, Tutorial de JavaServer Pages, Diciembre 2004.
- Http://www.sun.com
- Http://www.java.sun.com/product/jsp/jsp
- Http://www.lawebdelprogramador.com
- Http://www.java.sun.com/webservices/docs/1.2/tutorial/doc/JSPIntro.html#wp69778
- Http://www.programacionfacil.com
- http://www.programacion.com/java/tutorial/servlets_jsp/11/#servlets_jsp_jsp10
- http://roseindia.net/jsf/r/introducingjsf.shtml
- http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html
- http://www.programacion.net/tutorial/jsf_intro/1/#cap_1